Aikido

Programming Language Reference Manual

Issue 1.10

David Allison

Sun Microsystems Laboratories

4150 Network Circle
Santa Clara, CA
95054
USA

Sun Microsystems Laboratories

Chapter 1. A tour of Aikido

Aikido is an interpreted, dynamically typed language that can be used for general purpose programming but is best suited for prototyping and scripting. It has been derived from the ideas present in a large number of languages including Pascal, Ada, C, C++, Java™, JavaScript and Verilog. This chapter will give an overview of the features of the language. Full information is available in the following chapters.

1.1. Heritage

Aikido has been a dream of mine for about 20 years now. In 1980 I began to program computers for fun and learn more about the various languages out there. The first features that have become Aikido began when I wrote a macro assembler for a 6502 machine. The assembler had a large amount of the features found in Aikido today. The *inner statement* of the macro handlers came from that assembler. When I began to learn more about high level languages I noticed the beauty of symmetry in language design. Pascal was the first beautiful language I'd come across in university.

The structure of a Aikido program resembles that of a Pascal program. It is block structured, allowing functions to be defined anywhere in the program, not just at the top level as in C, or just inside a class like in the Java™ programming language. The syntax of the program looks like C++ with the usual C++ operators and syntactic elements. For example, Aikido uses braces to enclose compound statements and block bodies rather than the **begin/end** pairs of Pascal or Ada.

In designing the language, I chose C++ as the main language on which to be based. Where a feature is present in multiple languages, I chose the C++ way of doing it. The reason for this is that C++ is a very common language with which most professional programmers have had some experience. The Java™ language is also based on C++, so I would be in good company. The C++ language has gone through rigorous standardization recently and has been defined by recognized experts in language design.

The name Aikido comes from the heritage of the language. It has evolved from all the languages with which I am familiar, just as the Japanese martial art of the same name has evolved from the best parts of other martial arts

Unlike most other languages, Aikido is an interpreted language. This means that there is no compiler that produces object code to be run. The source text of the Aikido program is read directly by the interpreter. All input to the interpreter is in the form of source code – there is no object code at all. Even the system-supplied libraries are read as source code. This has advantages and disadvantages. The obvious disadvantage is that the size of the input to the interpreter is larger than it would be if it was pre-compiled. The size of the interpreter is also larger since it also incorporates a parser for the source code. The speed of execution is also much slower than compiled languages.

The main advantage is that there is no separate compile phase, meaning that the time from editing the source code to running is much reduced from a compiled language. This contributes to Aikido's utility for rapid prototyping where a fast turnaround from editing to running is essential.

1.2. Dynamic types

Aikido is a dynamically typed language. This is in contrast to all of the languages, from which it is derived, which are all statically typed. The difference is that a variable in Aikido can take a value of any type rather than having to explicitly say what type it is when a variable is declared. For example, in C++ you declare variables as follows:

```
// C++ declarations
int maxval = 1000 ;
std::string name = "Dave" ;
const double pi = 3.14159265 ;
User *u = new User (name) ;
```

This shows a set of simple variable definitions (and declarations). Each of them is initialized with a value (although this is not necessary, it is a good idea). Notice that each one specifies a type twice: once for the declared type and once implied by the initializer.

The same set of declarations in Aikido:

```
var maxval = 1000
var name = "Dave"
const pi = 3.14159265
var u = new User (name)
```

Notice that there is no type specification for the variable – the type is inferred from the initial value (variables and constants must be initialized in Aikido). A variable may be declared by the **var**, **const** and **generic** keywords. A variable declared using the **var** keyword is a regular variable that can change value but not type; one declared using the **const** keyword is a constant that cannot change value; the **generic** keyword declares a variable that can change both value and type.

Aikido provides the following types:

- Integer - 64 bit signed integer
- Real - floating point number
- Character - 8 bit ASCII code
- Byte - 8 bit value
- String - sequence of characters
- Vector - sequence of values
- Bytevector - sequence of bytes
- Map - set of mappings from one value to another
- Object - user defined object
- Stream - Input/output communication channel
- Function - enclosure
- Thread - enclosure that executes in parallel with others
- Class - user defined type
- Monitor - user defined type with mutual exclusion protection
- Package - user defined program section
- Interface - user defined contract
- Enumeration - set of defined names
- Enumeration constant - a user defined member of an enumeration
- None - no type defined

All of the usual operators are present in Aikido and can be applied to any of the types, subject to certain rules.

1.3. Generic variables

A variable declaration that is done using the **generic** keyword and defines a variable that can take any value of any type at any time. For example:

```
generic x                    // generic variable
x = 100                      // assign integer value
```

```
x = "hello"                    // assign string value
x = func                       // assign a function value
```

The template feature in C++ provides a generic type mechanism of sorts. For example, the following template function is generally quoted in the literature:

```
template <class T>
T max (T a, T b) {
   return a < b ? b : a ;
}
```

It is used to calculate the maximum of 2 values. It can be called with values of any type for which the less than operator is defined. For example, the following calls can be made:

```
int ix, iy ;
std::string s1, s2 ;

int m = max (ix, iy) ;
std::string s = max (s1, s2)
```

The first call will instantiate a different copy of the template than the second call. The template mechanism in C++ is a source level macro-like facility that actually compiles different copies of the code for each instantiation.

In Aikido , generic variables simplify this substantially. The parameters to a Aikido function are generic (unless otherwise specified). The following example shows the equivalent function to the C++ template shown above:

```
function max (a, b) {
   return a < b ? b : a
}
```

The function can be called with any type for which the less than operator is defined. The difference between this and the C++ template is that there is only ever one copy of the function. The Aikido functions are truly generic, rather than the template facility that shoehorns generic functions into C++.

1.4. Block structure

Aikido programs are organized in a block structure similar to Pascal and Ada. The outer level block in Aikido is called 'main' and is created automatically by the parser. This allows a simple Aikido program to consist simply of a series of statements without any functions. For example, a program to count from 0 to 99 and print to the screen could be coded as:

```
foreach x 100 {
   System.println (x)
}
```

That's the whole file. In C++ you would have to put this in a function called main. In the Java™ language you would have to define a class with a static function called main.

Inside the top level block can be other blocks. A block in Aikido is one of the following entities:

- Package
- Function

- Thread
- Class
- Monitor
- Interface

A *package* is a block that represents a section of a program. A *function* is a block that is called, does something and returns. A *thread* is a block that executes in parallel with other threads in the program. A *class* allows the definition of a user-defined type. And a *monitor* is a user-defined type that is protected by a mutual-exclusion lock.

An *interface* defines a contract for a set of members of a block. If a block *implements* an interface it is mandated to provide at least the set of members defined in the interface.

Here is a very contrived example of a set of nested blocks:

```
// package to provide counter
package CounterExample {
    const LIMIT = 1000                    // max range for counter

    // count
    public function count (from, to, inc) {
        if ((to – from) > LIMIT) {
            throw “too many iterations”
        }
        var x = from                                  // current counter

        // function to increment the current counter
        function increment() {
            x += inc
        }

        // function to print the current counter
        function print {
            System.println (x)
        }

        while (x < to) {
            print()
            increment()
        }
    }
}

// check the arguments
if (sizeof (args) != 3) {
    throw “usage: count <from> <to> <inc>”
}

// args are strings, need them as ints
var from = cast<int>(args[0])
var to = cast<int>(args[1])
var inc = cast<int>(args[2])

// run the example
CounterExample.count (from, to, inc)
```

The package *CounterExample* provides one public function *count*. This takes 3 parameters and counts, printing the values to the screen. The *count* function has 2 internal functions (*increment* and *print*). These are not really necessary in such a short example, but serves to illustrate the concept of nested blocks. The nested blocks have access to all the variables of their enclosing block (the function *count*).

1.5. Multithreaded programming

Aikido provides a simple mechanism for writing programs with multiple threads of control. The threading mechanism is very simple to use but very powerful. Thread synchronization facilities are provided to allow threads to share data. The mainstay of the thread synchronization facilities is the *monitor*. This is an Ada-like structure that allows only one thread inside it at one time. It also provides the ability for a thread to wait for resources and notify other threads when resources become available.

The following example shows a simple threaded program:

```
monitor Semaphore (count = 0) {                    // simple monitor for a
semaphore

   public function take() {                        // take the semaphore
      while (count <= 0) {
            wait()
      }
      count--
   }

   public function give() {                        // give the semaphore back
      count++
      notify()
   }
}

// write to standard output with a  semaphore to protect
thread writer (semaphore, s) {
   semaphore.take()
   System.println (s)
   semaphore.give()
}

var lock = new Semaphore(1)            // an instance of the semaphore

foreach w 100 {
   writer (lock, "writer: " + w)       // start writer thread
}
```

For full information on multithreaded programs see Chapter 10.

1.6. Stream input and output

Contrary to C++ and the Java™ language, Aikido provides full I/O facilities built in to the language. The operator -> is the stream operator that allows the contents one value to be copied to another. A stream is a type of value in Aikido that is a communications channel usually connected to a device. There are 3 predefined streams: stdin, stdout and stderr. These are connected to the standard input, standard output and standard error devices of the operating system.

To write a value to a stream, the stream operator may be used:

```
var name = “Dave”
name -> stdout
```

This writes the string *name* to the stream *stdout*. Streams can also be used for inputting values to the program. For example:

```
var n = 0
stdin -> n
```

will read from standard input to the integer variable ‘n’. The usual conversions (in this case from ASCII to binary) are performed when reading and writing to streams.

Streams may be connected to files, terminals, pipes, networks, threads and objects. A stream connected to a network allows for a simple way to write network-based applications. A program talking to a network does not even need to be aware that it is doing so.

1.7. Expressions and statements

Aikido provides the usual set of expression operators and control statements. In addition it provides much higher level constructs than is normally seen in a programming language. These include:

- Strings. Many string manipulation operators are provided. Strings can be used as easily as integers
- Vectors. Vectors are sets of values. Many operations are available
- Maps. A map is an associative container associating one value with another.

These high level constructs make it much faster to write code as most of the functionality is already available for you. For example, the string operations make string handling as easy as it would be in an old BASIC program:

```
var name = “Aikido”
var s = name[0:2]                       // s = “Aik”

const version = 1.0                     // real number
var s1 = name + “ version “ + version   // s1 = “Aikido version 1.0”
```

Automatic conversions from one type to another are preformed seamlessly. Conversions from string to integer are automatic (where you would have to call the atoi() function in C)

The set of expression operators found in C++ and the Java™ language are provided with the addition of the stream operator for input and output. A powerful member-testing operator is provided.

A powerful feature called ‘inline blocks’ allows statements to be used in an expression. This can be used instead of defining a function that is called only once.

Blocks (functions, classes, etc) may be declared anonymously and used in an expression.

Statements are those usually found in languages such as C++ or The Java™ language, with the addition of a powerful *foreach* loop.

Statements provided include:

- **if...else** statement with **elif** clause
- **switch** statement that operates on any type, not just integer
- **try...catch** statement for exception handling

- **for, do...while** and **while** loops
- **foreach** loop. Provides convenient iterator functionality
- **import** statement to allow for modularity
- **using** statement for namespace management
- **break, continue** and **return**

Full information on expressions and statements can be found in Chapter 6 and Chapter 7.

1.8. Object Orientation

Aikido is fully object-oriented. Although the language provides a rich set of types itself, Aikido allows the user to define completely new types through use of classes. Classes are very similar to other object-oriented languages, however in Aikido a class is a close relative to a function. This may be hard to see at first, but a function is really an object whose lifetime is the time it takes to call the function and return. The memory for the object is allocated when the call is made and deleted when the call completes.

Given that it is object-oriented, Aikido provides the ability for a class to inherit the characteristics of another class. In fact, because classes and functions are so similar, you make a function inherit the characteristics of another function (or any other block type).

A class defines a user-defined type. This type can also provide a set of operators that operate on objects of that type. Aikido, like C++, allows the built-in operators of the language to be overridden for a class.

Aikido also provides the Java™ *interface* model for block contracts.

1.9. Block extension

A unique feature of Aikido is the ability to extend an existing block. A block is defined in section 1.4. In a regular language if you wanted to add a function (say) to a class you would have to either modify the source code for the class, or derive a new class using inheritance. The former may not be possible or desirable. The latter means that you have introduced a new type and now must replace all uses of the old type with the new one.

Aikido allows a function (and anything else) to be added to the class (or other block) directly. Consider the ubiquitous example of adding a function to an existing string class:

```
extend String {
   public function compareIgnoreCase (s) {
      // code for comparing
   }
}
```

The class String will now have a new function *compareIgnoreCase*().

The extension mechanism can be used to add anything to any block so it is even possible to add code to a function.

1.10. Enumerated types

Languages have had enumerated types for years. Pascal used them extensively. C and C++ have the *enum* type specifier that introduces a set of integer names into the program. Aikido takes the Pascal approach but improves on it. An enumeration in Aikido is syntactically similar to C and C++ but semantically similar to Pascal. A constant of an enumeration is a name that can be assigned a constant integral value. A variable that has a value that is an enumeration constant can only be assigned other constants of the enumeration. Writing out the variable writes out the enumeration name.

Basically Aikido's enumerated types are pure and true to the philosophy.

1.11. Late binding

The term *late binding* refers to the time at which the item referred to is linked to a real definition. Let's make this more concrete. If you refer to a variable in an expression the parser searches for the variable with a certain name and places a pointer to a data structure representing the variable in the expression. This will not change later on in the program. Similarly, if you refer to a function, the parser will find the function and make a hard pointer to it – not to be changed later.

In C++, the term late binding refers to a call to a virtual function. If you refer to a member function of an object through a pointer or reference the C++ compiler does not make a hard reference to a particular function. Rather it makes a reference to a description of the function. The real function is determined at runtime when the real type of the object is known. The Java™ language also does this.

In Aikido this is extended further. In C++ a reference to a non-virtual function or data member of an object is bound at compile time. In Aikido all references to members of blocks are determined at runtime. This allows a reference to a block member to be made in the program before the block is actually defined. It also means that a reference to a block member is made concrete when the actual type of the block is determined at runtime.

The penalty for this feature is runtime speed. It is obviously slower to do a runtime search for a block member than it is to statically determine the member at compile time.

In some respects this is similar to The Java™ language's *call through interface* ability, where a call made to a function through an interface (as opposed to a class) causes a runtime search for the method signature. Aikido, because of its dynamic type nature, extends this to all references to block members rather than just methods (functions).

The C++ concept of a virtual function is the normal access method for all blocks in Aikido. If a derived block provides a member that itself is a block and that member matches the name of a block in a superblock then it behaves as what C++ calls a method override. In effect it replaces the superblock's member.

1.12. Access protection

Like most object-oriented languages, Aikido provides the programmer with control over the access to members of blocks. Any declaration is Aikido has an access mode associated with it. The access modes provided are:

- Public – available to anyone
- Protected – only available to the block itself and derived blocks
- Private – only available within the block itself

The access to a block member is determined at runtime when the actual block member is determined (see section 1.11). This has a runtime cost.

It is important to note that in Aikido any member has an access mode, not just class members. A function can provide public or protected members. This would be meaningless in the absence of the ability of a function to be subject to inheritance. In essence, a function can provide a public interface that is accessible to other functions or classes derived from it. For example:

```
function A {
   private var a = 1
   public var b = 2
```

```
}

function B extends A {
    var x = a                    // error: a in not accessibla
    var y = b                    // ok: b is a public member
}
```

1.13. Exception handling

Exceptions are nothing new. Languages like C++ and the Java™ language have had them for the few years. Even good old BASIC had an exception mechanism. Aikido's exception system is based on the trusted try/catch/throw mechanism of C++ and The Java™ language. Because Aikido is supposed to be lightweight and quick to program, the exception system is not onerous to use. You can simply throw anything you like at any time and an exception handler will catch it. There is no type specifier in the catch clause so all exceptions are caught. For example:

```
try {
    do_something()
} catch (e) {
    if (typeof (e) == "string") {          // pass a string on
        throw e
    }
}
```

If an exception is uncaught and makes it all the way to the top level of the program a runtime error occurs and the program terminates. This is unlike C++ where the program is aborted with a core dump. This allows exceptions to be used as regular errors. For example, it is common to check the arguments of a program on startup:

```
if (sizeof (args) < 2) {
    throw "usage: myprog file command"
}
```

1.14. Why choose Aikido

I am often asked why Aikido is any different or better than any other programming language. People often see Aikido as another Perl, or a slow C++. Yes, Aikido can be viewed as just another programming language written by someone keen on programming languages, but it is much more than that.

So why would you choose to write programs in Aikido as opposed to C++ (or even Perl if you are that way inclined and can read line noise)?

I suppose the trite answer is "why not?" but that is avoiding the question. There is no simple answer but there are some features of Aikido that make it much easier to write programs than it would be in another language:

- Aikido's syntax is very similar to C++ and Java™. There is a large community of people who know and like this syntax. If you know either of these 2 languages you can write a Aikido program with little or no learning. This is one criticism I have of Perl. The language syntax really does get in the way no matter how is can be justified. This is, of course, just my opinion. Those who feel differently can write their programs in Perl.

- Aikido is interpreted. This seems a minor point but does make a lot of difference when you are actually writing the programs. The fact that Aikido has no 'compile and link' phase means that the turnaround time from editing your code to running it is much less than with a traditional compiled language.
- Aikido is dynamically typed. When writing code in C++ or the Java™ language you have to spend a lot of time taking care of the types of the variables. You have to declare a variable with a certain type and then make sure that all the functions it is passed to have a parameter of a matching type. The static nature of C++ and the Java™ language certainly mean that once the code is compiling it is closer to being correct than before. In Aikido, all the type checking is done at runtime so getting is correct means that the code must be executed. This delays the checking for correctness until later in the cycle but the checks are still done.
- Aikido has high level constructs. These include commonly used types such as vectors, maps and strings. A vector is one of the most common classes used in a modern C++ program. They have effectively replaced simple arrays as they are extendible and have little or no access overhead. Mapping one value to another is a feature that is needed in almost every program but requires the use of a library class. Strings, of course, are very common. All these are built into Aikido and are as easy to use as common integers. There is also a high-level *foreach* statement that is used to iterate though any expression.
- Aikido has builtin stream input and output facilities. Nearly all programs take input and produce output. Using Aikido the input and output of values is simple to use and can be directed to any device. Writing to a network is as easy as writing to the screen. Streams can also be layered allowing protocols and filters to be implemented very easily.
- Aikido is multithreaded. Writing a program using threads is very easy in Aikido, vastly simpler than C++. Builtin monitors allow thread synchronization to be done with ease.
- Garbage collection is provided in the interpreter thus removing the burden from the user. Aikido uses reference counting garbage collection thus spreading the cost throughout the runtime of the program.
- Aikido is object-oriented. This has been a part of Aikido from its birth and not shoehorned into the language as an afterthought. Writing object-oriented programs in Aikido is very easy. There are even features not found in other languages, such as object extension and function inheritance.

The only justifiable reason for choosing one language over another is one of cost. If it saves time and money to use a certain language then that is a compelling reasons for do so. If the price of one language is similar to another, the choice of a language depends on personal taste and knowledge. Certain languages, however, do lend themselves to certain programming tasks better than others.

1.14.1. What Aikido is good at

Aikido's many high level features make it suitable for programming a certain type of application. It is true to say that Aikido can be used for more or less any programming task but it is better at some than others. Here is a list of some of the application is could be used for:

- Rapid prototyping.

  When writing a program for the first time in order to prove the concepts it is good to be able to produce a solution quickly. If you code in C++ or the Java™ language you spend a lot of time getting the program to compile without errors – there are just so many details to take care of. This usually involves checking types and inserting cast expressions to perform conversions. The dynamic types of Aikido more or less free you from this burden. Aikido lets you write a program very quickly to test the algorithms. The lack of a compile and link phase lets you insert debug and trace statements in the program and immediately run it. Then, when the program is running (albeit slower than it would in C++), you can easily edit it to convert it into your final language. I have used Aikido for this on many occasions and it is definitely quicker (and therefore cheaper) to program in Aikido and convert to C++ after the program is working than to code in C++ initially.

- Network programming

Writing code that talks to network connections is non-trivial in most languages. This is because the network support is provided by an external library rather than being built in to the language. Thus the mechanisms used in the language for regular input and output cannot be used for talking to a network without an abstraction mechanism. In Aikido the network support is built in to the language – you can read and write a network connection as easily as you read and write a file. The basic construct for this is the stream. Streams in Aikido can be layered in order to write protocol handlers and filters.

- Multithreaded programming
  Like network programming, writing programs with multiple threads is non-trivial. The thread support is usually not provided as a language feature, but is an external library. Aikido changes this so that thread support is a natural part of the language. Writing a program with threads becomes as easy (notwithstanding the issues of synchronization) as writing a program with functions.

- Scripting
  Aikido has been used very successfully for writing scripts. Writing a script is not much different from writing a regular program. The tasks involved are similar, possibly leaning more toward running OS commands and processing the output. A scripting language usually has little or no structural components – you write a file containing a set of commands and the script interpreter executes the commands. Aikido can be written in a similar fashion. Like prototyping, the high level features of the language make it easy to write scripts. Once you begin writing scripts in an object oriented language, you won't go back.

- String manipulation
  Handling strings of characters and files in Aikido is very easy. The regular expression handling features that are incorporated into the language (instead of being in a separate library) make string processing very powerful. Whenever regular expressions are involved, it is easy for the program to degenerate into something resembling line noise – with lots of special characters and backslashes. I think this is a necessary evil when dealing with complex string manipulation. Aikido also supports slicing of strings and vectors in a very flexible way.

- Tools
  Conceptually there is not much difference between tools and scripts. Tools tend be more functional that scripts and can include higher level parsing and text manipulation features. Aikido has been used for writing various build tools and source level parsing tools. It is much easier to write a complex tool in Aikido than use a scripting language such as Perl or one of the shells available on UNIX®.

- GUI Prototyping
  Aikido has an interface to the popular public-domain GUI toolset called GTK+. The interface converts the C-style GTK+ functions into an object-oriented form. It is very easy to prototype a GUI application using Aikido. It is even fast enough to run real applications without having to rewrite them in C++ or Sun's Java™ language.

One last thing that Aikido is good at is being the control language for an assembler. This is, after all, what it was designed to do.

1.14.2. What Aikido is not good at

So what can Aikido not do? It is not a replacement for C++. The very features that make Aikido suitable for prototyping make it slow and not safe for full scale implementation of a reliable software project. The dynamic types mean that the program cannot be type-checked unless all of the code has been run. The fact

that it is interpreted makes it much slower to execute than a C++ program (possibly up to 100 times slower).

Chapter 2. So, what is a Aikido program anyway?

Before we get to the interesting stuff we have to define what constitutes a program written in Aikido.

All the statements and declarations in a Aikido program reside in a *package* called *main*. The main package is automatically by the Aikido interpreter before the program is read from the disk. It is as if the programmer declared:

```
package main {
   const args = [<program arguments>]
}
```

and included all his statements inside the package body.

A package in Aikido is an enclosure that contains code to be executed and other declarations. The main package is automatically given a constant vector of strings containing the set of arguments passed to the program by the operating system.

As stated, the body of a package consists of a series of declarations and code statements. A declaration is a variable, class, function or other enclosure. A code statement is something that is executed.

No reference manual for any language would be complete without the obligatory “hello world” program. Here’s a version in Aikido:

```
System.println (“Hello world”)
```

That’s it...

What this does it to invoke the *println* function inside the *System package* to print the given string to standard output followed by a line feed character.

We could, of course, make it more complex.

That was too trivial to act as a real example, so here’s a more complex piece of code that actually does something useful.

```
// search the given stream for a regular expression

function grep (regex, instream) {
  while (!System.eof (instream)) {          // until end of file
     var line = “”                          // variable to hold line
     instream -> line                       // read the line
     if (sizeof (line[regex]) != 0) {       // contains regex?
        System.println (line)               // yes, so print line
     }
  }
}

// check the arguments
if (sizeof (args) < 2) {
```

```
    throw "usage: grep expr files..."       // exception: args bad
}

var regex = args[0]                         // get regular expression
delete args[0]                              // remove from arg list

// now go through each remaining arg, calling the grep function

foreach file args {
    var instream = System.openin (file)     // open a stream to the file
    grep (regex, instream)                          // call the grep function
    System.close (instream)                 // close the stream
}
```

This program shows a simple regular expression matcher for a set of files. Regular expressions are familiar to those who use UNIX®. They allow the user to specify an expression that contains special characters that match a range of characters in the input. The UNIX® program grep is a system-supplied program that searches files for regular expressions. This example shows how a grep-like program may be written in Aikido.

The first thing to notice is the complete lack of semicolon characters at the end of the lines. In other programming languages (C, C++, Java™, Pascal, etc), you have to terminate all statements with a semicolon character to satisfy the grammar of the language. In most cases, the semicolon is at the end of the line, immediately followed by a line feed character. In Aikido, the line feed character is significant in certain circumstances, this removing the need for an artificial end-of-statement character. In all languages there is a *natural end* for most statements and expressions. Aikido makes use of this natural end to work out what the programmer means. See section 7.2 for details on the rules and choices for this.

The program defines one function (called grep) taking 2 arguments: the regular expression to search for; and the stream to search. The arguments to functions (and other *blocks*) do not need a type specified for them. Aikido is a dynamic typed language and the type of a variable can be read at runtime. In this case, the function is internal to the program and we know exactly what types will be passed to it.

The first argument is a regular expression. This is a string that may contain special characters that match ranges of characters in the input.

The second argument is a stream. This is a builtin type that is a channel connected to an IO port. In this case it is connected to an open file.

The body of the function is pretty obvious. It enters a loop, reading each line into a string variable until end of file is reached. As each line is read, it is searched for the regular expression and, if found, output to standard output. The search function uses the subscript operator to subscript a string with a string. This results in a vector whose size is non-zero if a match is found.

The code of the main program first checks that we have been given the correct number of arguments. Arguments are passed in the vector *args*, which is a vector of strings. There is one element in the vector for each argument.

We extract the first argument (*args[0]*) as the regular expression and then delete the first element of the args vector. This shifts all the remaining element down one.

We then enter a *foreach* loop, looping once for each element in the vector. We open a stream attached to the file whose name is the element, call the grep function, and close the stream.

2.1. The basics

Let's get down to basics. The input to the Aikido interpreter consists of a sequence of characters in a file. The file is divided into a series of lines, each terminated by a line feed character. The characters in a line are grouped into a sequence of lexical tokens. Aikido is a case-sensitive language, meaning that an upper case letter is treated as distinct from the lower case version of the same letter.

2.1.1. Comments

A comment is identical to those in C++. Comments are ignored by the parser and form no part of the program. Both single and multi-line comments are supported. A single line comment begins with the character sequence // and ends at the line feed at the end of the line.

A multi-line comment is a sequence of characters starting with the character sequence /* and ending with ***/.** It can contain line feed characters, but cannot contain the sequence /* (multi-line comments do not nest).

2.1.2. Reserved Words

Every programming language contains a set of lexical tokens that are defined as *reserved*. This means that you cannot use an identifier with the same spelling as one of the reserved words. Aikido is no exception, the following is the complete set of reserved words in the Aikido language:

class	new	delete	import	package	public
for	thread	const	try	catch	throw
function	switch	case	default	break	continue
macro	if	else	elif	foreach	while
operator	generic	null	native	monitor	cast
private	protected	enum	static	using	extend
return	true	false	var	sizeof	typeof
interface	implements	do	instanceof	synchronized	in
extends					

Each of the reserved words is in lower case.

Note: Because the word 'in' is common in programs the keyword *in* can be used as an identifier in the program.

2.1.3. Literals

Like any other programming language, Aikido allows the programmer to use *literals*. A literal is a lexical token that is interpreted literally, like a number or a character constant, or a string.

2.1.3.1. Numbers

Aikido supports 2 different types of numbers: floating point and integer. A floating-point number is a sequence of digits containing a decimal point. The number can be in exponent form by adding the normal exponent suffix, that is 'e' or 'E' followed by a signed exponent.

An integer number in Aikido has a *base* and a *value*. The base is the mathematical base of the number and is one of: decimal, hexadecimal, octal or binary. As in C or C++, the base of the number literal is determined from the initial characters of the number:

- 0x - hexadecimal
- 0b - binary
- 0 - octal

Any other number specifies a decimal number literal. The set of valid characters in the number is determined by its base:

- Hexadecimal: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, a, b, c, d, e, f
- Octal: 0, 1, 2, 3, 4, 5, 6, 7
- Binary: 0, 1
- Decimal: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

The end of a number literal is reached when a character not valid for the base is detected in the input stream.

Examples:

```
1234          // decimal
0xffe0        // hexadecimal
033           // octal
0b110101      // binary
3.1415927     // floating point
1.23e+40      // floating point number in exponent form
```

2.1.3.2. Characters

A character constant is a single character enclosed in single quote marks. Only one character is allowed within the quotes (unlike C++). The value of a character constant is the ASCII code for the character. UNICODE or other non-english character mappings are not supported.

Like C and C++, a backslash in a character constant is used to escape special codes. The backslash is followed either with a special character code or an octal or hexadecimal number. The following special character codes are supported:

\a	alert character
\r	return
\n	line feed
\t	tab
\b	backspace
\v	vertical space

If the backslash is followed by the character ‘x’, then it is interpreted as a hexadecimal number. If it is followed by a number in the range 0..7, is it interpreted as an octal number. Otherwise the character following the backslash is read literally. In particular, the character constant ‘\\’ is a backslash character.

2.1.3.3. String literals

A string literal is a sequence of characters enclosed in double quote characters (“). Any double quote character in the string literal must be escaped by preceding it with a backslash character.

Inside the quote marks, backslashes can be used as in a character constant to specify a special character or non-ASCII character.

```
"this is a string"                  // simple string
"\tPress any key to continue\n"     // string with tab and linefeed characters
"\033[31m"                          // escape sequence (change color to red)
"Please enter value for \"name\"" // escaped quotes
```

2.1.4. Identifiers

An identifier represents a name in a program. An identifier is a sequence of characters that starts either with a letter or an underscore character, and contains letters, numbers and underscore characters. As Aikido is a case-sensitive language, upper and lower case letters are considered as different in an identifier. Thus the identifiers *mapset* and *MapSet* are considered as distinct identifiers. You can't use a reserved word as an identifier (except *in*)

Valid identifiers:

```
count
__end
quite_a_long_identifier
AnotherName
result34_4
in
```

Invalid identifiers:

```
2beOrNot2be         // starts with a number
sys$login           // illegal character
thread              // reserved word
```

Chapter 3. Values

A program in any language manipulates a set of *values*. It creates them, copies them from one location to another, inputs and outputs them and otherwise interprets them. A *variable* is a memory location that holds a single value. Any value in a program has an associated *type*.

A value in a Aikido program has one of the following types:

- Integer - 64 bit signed integer
- Real - floating point number
- Character - 8 bit ASCII code
- Byte - 8 bit byte value
- String - sequence of characters
- Vector - sequence of values
- Byte vector - sequence of bytes
- Map - set of mappings from one value to another
- Object - user defined object
- Stream - Input/output communication channel
- Function - enclosure
- Thread - enclosure that executes in parallel with others
- Class - user defined type
- Monitor - user defined type with mutual exclusion protection
- Package - user defined program
- Interface - contract for interface contents
- Enumeration - set of defined names
- Enumeration constant - a user defined member of an enumeration
- Memory - the address of a block of raw memory
- Pointer - an address inside a block of raw memory
- None - no type defined

3.1. Integer

An integer is a 64 bit signed integral number. The range of an integer is –9223372036854775808 to 9223372036854775807 (approximately –9e18 to 9e18). Every value in the range is usable.

Integers can be manipulated by performing arithmetic on them. There is no overflow detection for integers.

3.2. Real

A real number is what is a floating-point number in IEEE double precision format. This allows numbers in the range: 2.2250738585072014e-308 to 1.7976931348623157e+308.

As this is stored in floating point format, not all the values in the range are possible. Inaccuracies are also probable during calculations.

Real numbers may be manipulated like integers, except there are certain operations that are not allowed: shifting and bitwise operations being 2 examples.

3.3. Character

A character is a value that maps to the American Standard Code for Information Interchange (commonly known by the acronym ASCII). This defines a mapping for the characters commonly found in the English speaking world.

Although there are many other characters present in use in the world, Aikido only supports those 256 characters mapped by the ASCII standard. This restriction may be lifted in future.

A character may be though of as an integer in the range 0..255 and behaves as such with respect to arithmetic operations. It does, however, behave differently when input and output operations are applied to it.

3.4. Byte

A byte is a value that holds a number between 0 and 255 (inclusive). That is, it is an 8-bit quantity. Bytes are only really useful when extracted or inserted into bytevectors.

3.5. String

A string is an ordered sequence of characters. It has a certain length (possibly zero). A string with zero length is called the 'null string' and is represented as "" in the Aikido program.

Strings may be manipulated by a restricted set of operators in the Aikido language. In particular, they may be appended using the '+' operator and subscripted using square brackets.

3.6. Vector

A vector is a value that holds a sequence of other values. It can be of any length (including zero). Each of the values in the vector can be of a different type. Vectors are represented in the Aikido language by a sequence of values (possibly none) enclosed in square brackets and separated by commas. For example, the following is a valid vector literal:

[1,2,3,5,7,11,13,17,19]

representing the set of prime numbers below 20.

Vectors may be manipulated by various operators in the language. In particular, new values may be inserted into them and others deleted. They may also be subscripted.

3.7. Byte vector

Whereas a vector contains values of any type, a byte vector contains a set of values which are of type *byte*. There is no literal representation of a byte vector, but it can be created using the *new* keyword:

```
var vec = new byte [100]                // vector of 100 bytes
```

3.8. Map

A map is a value holding a set of mappings from one value to another. Put another way, a map contains a series of 'value = value' pairs. It is an associative container in that it associates one value to another.

A map literal is represented in the language by a series of 'value = value' pairs enclosed in braces and separated by commas. For example, the following is a map:

```
{"Danville" = 15000, "San Ramon" = 12000, "Oakland" = 500000}
```

which may represent the population of various Bay Area cities.

Each of the values in a map may be of a different type.

Maps may be manipulated by a restricted set of operators in the language. In particular, new mappings may be added to them, others deleted and they may be searched by subscripting them.

3.9. Object

An object is an instance of a user defined type (defined using a *class* or *monitor*). They are created using the *new* operator of the language. The contents of an object are accessed by use of the '.' operator.

Objects may be deleted either by the *garbage collector* or by explicitly using the *delete* operator.

The reserved word *null* specifies the null object.

3.10. Stream

A stream is a channel attached to an IO port and used for sending and receiving data. Streams may be attached to a file, screen, keyboard, network, or anything else. Standard streams are provided for the standard file descriptors available to all programs (called *stdin, stdout* and *stderr*).

A stream is created when a file is opened or a network connection is made. You can read from and write to a stream using the '->' operator. There is an abundant set of library functions available to manipulate streams.

3.11. Function

A function is an enclosure containing declarations and code. When the program invokes a function, control transfers to the code of the function, which is executed. When the function returns, control is returned to the statement after the function call.

A function can take a set of parameters, which are assigned values when the function is called. The function can also return a single value to its caller.

The only valid operations on a function are to pass it around, to extend it, derive from it and call it.

A function is defined like:

```
function name (arguments) {
   // function body
}
```

3.12. Thread

A thread is very similar to a function, except that when it is invoked, the invoker continues execution without waiting for the thread to return. The thread then executes in parallel with the invoker and all other threads in the program.

Like a function, a thread may be passed a set of parameters, but may not return a value to the caller.

The same set of operations as a function may be applied to threads.

A thread is defined like:

```
thread name (arguments) {
   // thread body
}
```

3.13. Class

A class is a user defined type. The Aikido language provides a rich set of predefined types for use in the programs. By defining a class, you are extending the set of provided types with one of your own. When you define a class, you give it a body and a set of operators with which to manipulate the body.

A class may be passed parameters, which are assigned values when the *new* operator is applied to the class, thus creating a new instance of it (an *Object*). The body of a class may contain code that is executed when the instance is created. This code is known as the constructor.

A class definition is very much like that of a function:

```
class name (arguments) {
   // class body
}
```

3.14. Monitor

A monitor is a user defined type with a special property – it guarantees mutual exclusion on any accesses to it. This is essential when programming with threads as their nature means that certain data must be protected against simultaneous update by multiple threads.

An instance of a monitor is created with the *new* operator (just like a class). The object created by the monitor behaves just like a regular object, except that any access to it (invoking a method, reading or writing an attribute) first gains a mutual exclusion lock on the memory. The lock is released when access is no longer required.

A monitor definition is very similar to a class:

```
monitor name (arguments) {
   // monitor body
}
```

3.15. Package

A package is an enclosure defining a section of a user program. See Chapter 1 for full information on packages.

A package is defined as:

```
package name {
   // package body
}
```

The package name can contain multiple identifiers separated by dots. For example:

```
package com.sun.Aikido {
   // contents
}
```

This is equivalent to:

```
package com {
   public package sun {
       public package Aikido {
          // contents
       }
   }
}
```

3.16. Interface

An interface is a contract. An interface contains a set of members that have no definition but serve to define a contract for a block to be defined later. In other words, if a block implements an interface it must provide all the members defined in the interface.

The members of an interface do not have an access mode – the access is assigned by the implementing block. They also do not have any bodies as these are provided by the blocks implementing the interface. Here is an example of an interface:

```
interface Event {
   function isFatal : int
   function operate (para : integer = 0)
   operator -> (stream, isout)
}
```

This defines an interface that contains 3 members; 2 functions and an operator. Any block that implements this interface must provide these 3 members, otherwise an error occurs. The members provided by the block implementing the inteface must match the interface signature precisely. This means that it must have:

1. the same number of parameters
2. the same return type (if it is a function)
3. the same variable argument status (…)
4. each parameter must have:
    1. the same type (if defined)
    2. the same default value (if defined)

Another purpose of an interface is to allow block equivalence to be tested. This can be done using the *instanceof* operator. Consider the following example:

```
interface A {
}

class B implements A {
}

var b = new B                          // instance of B
```

```
var x = b instanceof A                    // is B an instance of A?  yes.
```

3.17. Enumerations

An enumeration defines a set of constants (*enumeration constants*). A value whose type is enumeration may only be assigned one of the member enumeration constants. Effectively an enumeration is a user-defined type whose values are limited to the constants held within.

For example:

```
enum Color {
   RED, GREEN, BLUE
}
```

Defines an enumeration (enumerated type) that contains the 3 constants RED, GREEN and BLUE. In C and C++, an enumerated type contains a set of named integer constants, which (especially in C) can be used interchangeably with regular integers.

In Aikido, an enumeration is very much like that of Pascal or Ada. The constants in an enumeration are not integers but may also be assigned an integer value.. An enumeration should be used when the set of values taken by an object are known when the program is written and the object cannot take any value outside that set.

The enumeration in Aikido is pure. Once a variable has a value that is a member of an enumeration it cannot be then made to take a value outside of the enumeration. For example, if you decide that a variable has a value RED (indicating that it is a member of the Color enumeration), it doesn't make any sense to let the program assign it the value 1000, or "red".

The identifiers in an enumeration can be considered as integer constants. The default values of the constants are similar to that in C. The first constant has value 0 and each subsequent constant has a value one greater than the previous constant. The value of an enumeration constant can be obtained by casting the constant to an integer. In the enumeration *Color*, the values of the constants are 0, 1, and 2 respectively. If the default values are not desired, the constants may be given values by appending a constant expression of integral type. Consider the following enumeration:

```
enum Type {
   INTEGER = 1
   REAL = 2
   STRING = 4
}
```

This defines an enumerated type with 3 constants (supposedly representing the types in a compiler). Each enumeration constant has an integral value. The values may be an integral constant expression. For example:

```
enum Opcode {
   IDENTIFIER = 1 << 0,
   NUMBER = 1 << 1,
   CHAR = 1 << 2
}
```

Most of the normal expression operators may be used in the constant expressions. The operands of the expressions can be an integer, an enumeration constant or the literals *true* and *false*.

The following operators are valid in a constant expression:

```
|       ^       &       ==      !=      <       >       <=
>=      <<      >>      >>>     +       -       *       /
%       +(unary)        -(unary)        !       ~
```

The constant expression may also include a parenthesized constant expression.

To obtain the value of an enumeration constant you can cast it to an integer (see section 6.15 for information on casts). For example:

```
var x = cast<int>(NUMBER)              // sets x to the value 1 << 1 (2)
```

You can also use the enumeration constant as an index into a vector or in a range expression for the *in* operator and *foreach* statement.

Once a variable has been assigned an enumeration constant you can perform the following manipulations on it:

- Move the value to the next (n) value in the enumeration. The value of n can't be greater than that which put you at the last value in the enumeration
- Move the value to the previous (n) value in the enumeration. You can't go past the start
- Send it to a stream

Consider the following examples:

```
var color  = GREEN                     // assign member of Color
var nc  = color + 1                    // nc = BLUE
color = RED                            // reassign to RED
color += 2                             // color = BLUE
color -= 1                             // color = GREEN
color -> stdout                        // send to stream
```

Sending an enumeration constant to a stream will cause the name of the enumeration to be sent to the stream. In the above example, the stream gets the characters: GREEN

Moving beyond the limits of the enumeration will cause a runtime error.

You can iterate through the members of an enumeration using the **foreach** statement:

```
foreach col Color {
   System.println (col)
}
```

You can also use an enumeration constant as an index into a vector:

```
var cols = new [sizeof (Color)]                // declare a new vector
cols[GREEN] = 50
cols[RED] = "hello"
var a = BLUE
var b = cols[a]
```

3.17.1. Extending enumerations

An enumerated type may be extended in 2 ways:

- By deriving a new type from it through inheritance
- By directly extending it

Consider the following enumerated type that would be present in a lexical analyzer of a compiler:

```
enum Tokens {
   IDENTIFIER,
   NUMBER,
   STRING,
   CHAR
}
```

This could be a base set of tokens that are recognized by the lexical analyzer. Say we are going to be using the lexical analyzer in our project. If it is to be of any use, we need to be able to add to the set of tokens. In C or C++, you would have to define a new enumeration containing the additional tokens and then use casts to satisfy the compiler and stop it grumbling about type mismatches. For example, a function called match() could be defined in C++ as follows:

```
bool Lex::match (Tokens tok) ;
```

This looks at the current token to see if it matches the one passed as a parameter, and if so moves on to the next token and returns true. If there is no match is just returns false.

We can call this with one of our defined tokens with no problems:

```
if (match (CHAR)) {
   // process character matched
}
```

Say we defined a new set of tokens for our extended lexical analyzer:

```
enum NewTokens {
   SEMICOLON,
   DOT,
   STAR,
   SLASH,
   PLUS,
   MINUS
}
```

If we wanted to use these as a parameter to the match function we would have to use a cast:

```
if (match ((Tokens)PLUS)) {
   // process PLUS token
}
```

This makes for untidy code.

In Aikido we use a similar technique, but instead of defining a completely new type, we can just derive from the original enumeration:

```
enum NewTokens extends Tokens {
```

```
    SEMICOLON,
    DOT,
    STAR,
    SLASH,
    PLUS,
    MINUS
}
```

Now a constant in the NewTokens enumeration is just treated as a member of the Tokens enumeration.

Another approach is to extend the Tokens enumeration as follows:

```
extend Tokens {
    SEMICOLON,
    DOT,
    STAR,
    SLASH,
    PLUS,
    MINUS
}
```

The difference in the 2 approaches is that in the first case a new type name is formed and in the second, the type name remains the same. This may be important if you are doing type checking in the program. In the absence of type checking, the two approaches are similar.

The integer values of the constants added to the enumeration continue on from the values used in the enumeration being extended or derived from. For example:

```
enum Numbers {
    ZERO, ONE, TWO, THREE
}

enum MyNumbers extends Numbers {
    FOUR                                    // value 4
}
```

or:

```
extend Numbers {
    FOUR                                    // again, value 4
}
```

3.18. Memory and Pointer

Memory values are created by calling the function *System.malloc*(). They represent the address of a block of raw memory. The block is subject to normal garbage collection rules. A *pointer* is created by adding an integer value to a *memory* value. The *pointer* must remain inside the block of memory allocated by the *malloc* function. Adding or subtracting an integer to/from a *pointer* will yield another *pointer*, again inside the memory block.

See section 15.3.9 for further information about raw memory functions.

Consider the following examples:

```
var mem = malloc (1024)                 // raw memory block
```

```
var p1 = mem + 1000          // pointer to memory
var p2 = p1 – 100            // another pointer
var p3 = p1 – 1001           // error: outside memory
var p4 = p2 + 200            // error: outside memory
poke (p2, 1234,4)            // poke value into memory
poke (p2+200, 1234, 4)       // error: pointer is outside memory
```

3.19. Closures

A closure is a construct that contains enough information for a function, or other block, to be executed in any context. Basically, this means that the block is packaged up with the complete 'static chain' required for it to execute. The static chain contains the set of blocks that enclose the block. A closure is created when a block is passed as an argument to or returned from another block. It is a garbage-collected entity that will be deleted when it is no longer referenced. A consequence of closures is that stack frames are also garbage collected and will be deleted only when there are no references to them.

The main use of closures is to enable a function to be passed to another function and called. Without closures (as in older versions of Aikido), the calling function would have to be at the same lexical scope as the function passed as an argument to it. Consider the following example:

```
var s = System.transform ("hello", ctype.toupper)
```

This calls the library function 'transform' and passed a function from another package to it. The transform function calls this function for every item is its first argument (the characters of the string "hello" in this case).

The main advantage of closures is that the programmer doesn't have to worry about the proper lexical scoping of functions passed as arguments, thus relieving a heavy burden.

3.20. None

This is a special value that means "no value". It is used in certain occasions when there is no value that can reasonably be assigned. For example, the contents of a vector created by the '*new []*' operator have the value *none*.

Tip: a convenient way to assign the value *none* to a variable is to use the predefined identifier 'none', or to assign it the expression {}[0]. This is an index of an empty map, which always results in the value *none*.

Chapter 4. Packaging up your code

Any sizeable program needs to be divided up into meaningful sections. In C you can divide it into a set of files. In C++, you divide it into files and also use *namespaces*. The Java™ language has the idea of a *package*, where you specify the name of the directory in which the code will reside when compiled.

Aikido takes this one step further. Take the example of a grep program presented in Chapter 1.

All declarations and code at the top level in Aikido are in the package *main*. When the program executes, it starts with the first statement in main and continues to the last one. This is useful for writing small programs that do one thing, but if you want to write something that may be useful to other programs (reusable code) you have to rethink the layout. One possibility is to create a single function for the whole program and call it when you want to invoke the code.

```
function mygrep (regex, args) {
   // contents of the grep code here
}

// later in the program
mygrep (regex, args)
```

This is rather limiting, in that the function can only do one thing.

Another, better, alternative is to create a class containing the program and create an instance of it. You can then invoke the methods of the class from anywhere. If we want to do this, it would be better to name the class something that is more generic as it will be called upon to do more than one thing. Let's call it FileUtils.

```
class FileUtils {
   public function grep (regex, args) {
      // contents of grep here
   }
}

// later in the program
fileutils = new FileUtils()

fileutils.grep (regex, args)
```

This is more flexible. The FileUtils class can contain other functions that can be invoked. The disadvantage of this is that the variable *fileutils* needs to be available to anything that wants to invoke a method on the class. This means either passing it around as a parameter or making it a global variable.

The best approach for this problem is to use a *package*.

```
package FileUtils {
   public function grep (regex, args) {
      // contents of grep here
   }
}

// later in the program
```

```
FileUtils.grep (regex, args)
```

A package cannot have any parameters. It defines an enclosure that contains block members. The members have access protection. The names within the package are local to that package and do not clash with names in other packages.

A package name can be a sequence of identifiers separated by dots. This allows a 'distinguished name' to be used for packages. Consider:

```
package java.lang [
   public class System {
      // body
   }
}
```

This defines a package whose name is 'java.lang' and which contains a single class 'System'. This is a shortcut for:

```
package java {
   public package lang {
       public class System {
       }
   }
}
```

The 'System' class inside the package has the distinguished name 'java.lang.System'.

Another characteristic of a package is that you can add things to it after it has been defined. For example, if I wanted to add a copy() function to the FileUtils package:

```
// somewhere in the program
package FileUtils {
   public function copy (from, to) [
      // contents of copy function
   }
}

// later in program
FileUtils.copy ("a.txt", "b.txt")
```

This piece of code can be placed anywhere in the program. What it does is to extend the package by adding the function. Why would you want to do this? Maybe the file containing the original FileUtils package code is not writable, or owned by someone else and you want to add your own function.

4.1. Package scope

A package looks a lot like a class but there is one major difference: you cannot instantiate a package. A package has no storage associated with it. When a package is declared, all the variables defined inside the package are placed in the enclosing block – there is not a new block allocated for it.

In technical terms, a package introduces a new lexical scope but not a new scope level. This distinction is important because it enables the variables inside a package to be accessed even though there is no instance of the package.

Thus a package introduces a new scope in which its variable names live.

4.2. Packages as namespaces

A namespace is an encapsulation for a set of names used in a program. Names used in one namespace to not collide with those defined in another namespace. The use of namespaces removes the pernicious problem of 'namespace pollution' where a program affects the names in another program by defining its own names globally.

Every block in a Aikido program defines a new namespace. A scope is a namespace. This is not, however, what is usually meant when people talk of the concept of a namespace. Generally speaking, a namespace has evolved to mean a division of the main program in which the names are enclosed in a namespace identifier.

For example, in C++, a namespace is defined like:

```
namespace mynames {
   int x ;
   void f() ;
}
```

This defines a namespace called *mynames* containing 2 names: 'x' and 'f'. The names 'x' and 'f' do not clash with names in other namespaces. Outside the *mynames* namespace, the variables can be referred to as:

```
mynames::x = 1 ;
mynames::f() ;
```

That is, by prefixing the names by the name of the namespace, you can access the contents of the namespace.

Another feature of the namespace mechanism in C++ is the ability to reach into a namespace, extract a name and make an alias for it in your own namespace. This is done using the 'using' declarations and directives. For example, if we wanted to make the function 'f' available as the local name 'f', we can do:

```
using mynames::f ;

// call the alias
f() ;
```

Aikido uses packages as namespaces. A package defines a block therefore it is automatically a namespace. You can prefix a name inside a package with the package name to access it:

```
package Lex {
   public var currentToken = 0
   // others
}

var c = Lex.currentToken
```

Aikido also supports the 'using' concept. After a package is defined, you can issue a 'using' statement:

```
 using Lex
var c = currentToken
```

This is a simplified form of the C++ *using*. It only supports what is called a *using directive*, not a *using declaration*. The difference is that a using directive takes all the names of a namespace and makes aliases for them in the local namespace. With a using declaration, you can selectively choose which names to extract.

One area where Aikido exceeds the C++ namespace mechanism is the ability to protect names inside a namespace. Since a namespace is a package, the normal member access protections apply to any accesses. Only those names that are marked public will be accessible from outside the namespace. Also, since they are packages, namespaces can be derived from one another.

4.3. Package dangers

A package is an open enclosure. This allows other source files to contain different parts of the package. It looks innocent enough to split a package over multiple files, and indeed it is a useful feature, but there is a danger of which you should be aware…

Consider the following code:

```
package Parser {                    // a package for a parser
   function getToken() {            // a function in the parser
      // body
   }
  // other bits
}


function print (s) {                // a function outside the package
   // do something
}

package Parser {                    // more parser (in a different file, maybe)
   function error (s) {             // an error function
      print (“Error: + s)           // results in “Cannot call a value of type none”
   }
}
```

If you follow the above code, the call to the function ’print’ in the function ‘error’ results in a runtime error. Why should this be?

The answer lies in the fact that the package is extended by the second package block and this results in the code for the second package block being appended to the first block. When this is executed it is before the function ‘print’ has been encountered and therefore the value of the variable holding the function body has not yet been assigned a value.

Chapter 5. Declarations

You've seen examples of declarations earlier in the document. The discussion on packages in the previous chapter showed the declaration of a package. Here we will see other declaration in all their gory detail.

A program cannot do anything useful without having variables to work on. A variable is an example of a declaration.

```
var nickname = "Dave"
```

Shows an example of a variable declaration. It declares a variable called *nickname* and assigns it an initial value of the string "Dave". This tells the interpreter to allocate space of the variable in the current *scope*.

5.1. Variables

A variable is a named location that can store a value. Variables are allocated space in the current scope. When a variable goes out of scope its value is destroyed.

Aikido is a stack based interpreter and in this respect it operates just like a C++. When a scope is entered, space is allocated for all the variables in that scope. All variables (with one notable exception) must be initialized at point of declaration. This serves to assign an initial type to the variable. Once a variable has been assigned a type, it cannot be assigned a value with a different type (unless the current type is *none*). The one exception to this rule is a *generic* variable, which can take a value of any type at any time. Variables that are generic do not need to be initialized when declared.

The **var**, **const** or **generic** declaration statements declare variables:

```
var name = "Joe"              // a string variable
var count = 1234              // an integer variable
const MAX = 1000              // an integer constant
const FILENAME = "/tmp/x"     // a string constant
generic rooms                 // a generic variable
```

Variables declared using ***var*** are known as *regular* variables. They are the most common sort of variables in a Aikido program. Their value can be changed at any time (subject to the type restrictions mentioned earlier). If the variable has been assigned a value of type *none*, then it can be changed to anything after that.

Those variables declared using **const** are constant and their value may not be changed once it has been assigned.

Generic variables do not need to be initialized and can be assigned any value at any time.

A variable doesn't have to hold a number or string. It can hold a value of any type including function, package and class. This allows functions and classes (and other blocks) to be treated as a scalar quantity and passed around the program.

Consider the following set of variables and statements.

```
function print {                    // a function
}
```

```
class Member {                          // a class
}

var name = "Dave"                       // string variable
name = 6                                // error: cannot change type to integer

const MAX = 100                         // integer constant
MAX = 101                               // error: cannot change value of constant

var p = print                           // function variable
p()                                     // call the function print

generic x
x = name                                // x == "Dave"
x = 6 * 5 +4                            // x == 34

var type = Member                       // class variable
var q = new type        ()                      // instance of Member
```

The **var**, **const** and **generic** declarations allow multiple variables to be declared at one time. Simply separate the declarations with commas:

```
var a = 2, b = 4
generic q, w, e
const max = 1000, min = 0
```

5.1.1. Constants

The keyword **const** is used to define a *constant*. A *constant* is a variable whose value cannot be changed by assignment. Since Aikido is dynamically typed there are only a few rudimentary checks that can be done to ensure that you don't try to overwrite the value of a constant. The parser will not let you assign a value to a constant once it has been set in the declaration. Subscripts of constants cannot be assigned to. Consider the following examples:

```
const  MAX = 100
const name = "fred"
const vals = [1,3,5,7]

var x = MAX                             // fine
MAX = 101                               // error, MAX is constant
name = "joe"                            // error, name is constant
name[2] = 'a'                           // error, name is constant
vals[100] = 66                          // error
```

When defining the formal parameters for a block it is useful to declare some parameters as const in order to prevent their modification by the block. For example:

```
function f (a, const vec) {
   vec[3] = a                                   // error, vec is constant
}
```

5.2. Scopes

Before going any further, we need to introduce the concept of a scope. A scope is a collection of declarations. There is one global scope in the program (corresponding to the package *main*). Each time you declare a new enclosure inside main, you create a new scope. An example of an enclosure would be a function or package.

A declaration inside a scope will not clash with declarations in other scopes. This means that you are free to name your declarations anything you like inside a scope without having to pay regard to the names given to declarations in other scopes.

When you are in a scope, you can reference the declarations in that scope. Scopes may be nested, meaning you can open a new scope inside a scope. When you are in a nested scope you have access to the variables within the scope and also variables in any enclosing scopes. The names in a nested scope take priority over the names in an enclosing scope.

Any variable (or other declaration) must be declared before it can be used.

Here is a contrived set of scopes to explain the rules:

```
var x = 1                       // variable x in scope main
var l = 10                      // variable l in scope main

function y {                    // function y in scope main
   x = l                        // assign main.l to main.x
  var y = x                     // declare new variable y and assign main.x
}

class p {                               // class main.p
   var x = 2                    // variable main.p.x
   x = l                        // main.p.x = main.l

   var l = 20                   // new variable main.p.l
   if (q) {
      var x = 3                 // new variable x inside block
      x = l                     // assign main.p.l to local x
   }

   function y (l) {         // function main.p.y
      x = l                     // main.p.x = argument l
   }
}

x = l                           // main.x = main.l
```

As can be seen, the scope rules are obvious and make it possible to hide declarations on enclosing scopes. In order to gain access to a declaration in an enclosing scope, you can qualify the declaration name with its scope name.

```
var x = 10                      // variable main.x

class p {
   var x = 20                   // variable main.p.x
   x = 40                       // main.p.x = 40
   main.x = 30                  // assign to main.x
}
```

5.3. Blocks

A block in Aikido is a named enclosure. Examples of blocks are packages, functions and classes. A block contains executable code and declarations enclosed in a set of braces. For example, the function declaration:

```
function func {
   // block body
}
```

Declares a function block. Blocks are referred to by name. The name is declared in the current scope and must be unique in that scope.

5.3.1. Block parameters

Blocks can be declared anywhere but must be declared before they can be used. Blocks can take parameters. These are appended after the block name in the declaration. The following is an example of a function declaration with parameters.

```
function print (indent, text) {
   // function body
}
```

The parameters are enclosed in parentheses and consist of the 2 variables *indent* and *text*. The parameter declaration does not specify a type for the parameters. This enables the parameters to be *generic* variables, capable of taking any type. The type of the parameters can be checked in the function using the *typeof* operator.

If there are no block parameters, you can either use an empty set of parentheses "()" or omit the parentheses altogether. Consider the following function definitions:

```
function dosomething() {
   // body
}

function dosomething_again {
}
```

Both definitions specify functions that take no parameters. Which form to use is a matter of personal taste. I tend to switch between them.

The parameters to a block may be declared constant by prefixing them with the keyword **const**. This prevents assignment to both the variable itself and subscripts of the variable. For example:

```
function sortArray (const array) {
   // sort the array, but not allowed to modify it
}
```

Packages cannot have parameters.

5.3.2. Parameter access control

The parameters to a block are subject to the same access control rules as regular variables. In fact, a parameter to a block is a regular variable inside the block whose value is assigned by the caller. By default

the access mode for a parameter is *private*, just like a variable. You can modify the access mode by using the access control keywords in the block parameter list. For example:

```
class Employee (public name, age, protected role) {
   // contents of class
}

var me = new Employee ("Dave",  37, ENGINEER)        // instance of Employee
print (me.name)                                      // OK, name is public
print (me.age)                                       // error: age is private
```

See section 5.3.17.2 for details on access control of block members.

5.3.3. Parameter types

Sometimes it is desirable to specify a type for a parameter. Aikido does not have any keywords for types. To specify a type for a parameter, use the following notation:

```
function print (indent : int, test : string) {
}
```

This is reminiscent of the way Pascal and Ada do it. The 'types' *int* and *string* are not types at all, but are constants initialized with values of the appropriate type. The notation after the ':' in the parameter list is an expression whose type is used to check the actual parameter type assigned to the parameter. The following would be the same thing:

```
function print (indent : 1234, text : "hello world") {
   // function body
}
```

The value of the type designator is always ignored, it's just the type that matters. When the call is made, the interpreter attempts to convert the actual parameter type to the type of the formal parameter as if by using a *cast* operator. If the conversion is not possible a runtime error results.

So when do you decide to use a parameter type instead of leaving it generic? The choice is fairly arbitrary for internal blocks. I tend to omit them if they are obvious. For an external block (one that is called from outside your code) you should include the parameter types to ensure that the types are correct. Of course, you can choose to omit them and do the checks for correct types inside the block body.

The following constants are created by the system as a convenience for use by the programmer when specifying types for parameter values and for the *cast* operator. They are not reserved words.

```
const int = 0
const integer = int
const string = ""
const vector = []
const bytevector
const map = {}
const char = 'a'
cont byte
const stream = stdout
const real = 0.0
const object = null
```

```
const none
```

5.3.4. Default parameters

Parameters can also be given default values. The default value for a parameter is used when the caller does not supply a value for it. In this case, the parameters with default values can be thought of as optional.

```
function writeString (text : string, stream = stdout) {
   // function body
}
```

Here, the function *writeString* takes to parameters: *text* (a string) and *stream* (a stream) with a default value of *stdout*. This function can be called as follows:

```
writeString ("hello", outstream)
writeString ("hi there")
```

The first call writes the text to the stream *outstream*, while the second writes it to standard output. Default parameters can only appear at the end of the parameter list. There cannot be any parameters without a default value after one with a default value. For example, the following is illegal:

```
function f (t = 0, x) {                    // x has no default value
}
```

5.3.5. Reference parameters

One last feature of parameters is the ability to pass a parameter by reference. This allows the parameter to be used as an output of the block. The syntax for this is the same as Pascal.

```
function getCoords (var x, var y) {
   x = screen.getX()
   y = screen.getY()
}
```

In this case, the actual parameters are passed by reference.

```
var x = 0
var y = 0
getCoords (x,y)
```

When the block is invoked, the values of the local variables *x* and *y* will be set. This can only be used where the actual value has an address (i.e. is a variable). The interpreter will report an error if an attempt is made to pass something else (referred in C literature as an *rvalue*) by reference. The only things that can be passed by reference are:

- Variables
- Block members using a member access expression
- Subscript expressions (e.g. vec[4])

5.3.6. Variable parameter list

You may define a block where the names and number of parameters is not specified in the formal parameter list.

```
function printf (format, ...) {
   // body of function
}
```

In this case, the caller must supply at least one parameter (for the parameter *format*), but may provide any number of additional parameters after that. The interpreter gathers the additional parameters into a vector and passes the vector to the block as a parameter named *args*.

The block can then refer to elements of the *args* vector by subscripting. The first additional parameter is at element *args*[0], the second as *args*[1], etc. The number of additional parameters can be obtained by the expression *sizeof (args)*

For example, this function prints all the arguments to the given stream:

```
function printArgs (stream, ...) {
   foreach arg args {
      arg -> stream
  }
}

// called with
printArgs (outstream, 1,2, "hello", Member)
```

It is also possible to make a function have only a variable set of arguments:

```
function f(...)
   // the args vector will contain all arguments
}
```

5.3.7. Understanding parameter passing

The semantics for passing parameters in Aikido is very similar to that of the Java™ language:

- Scalar values are passed by value.
- Compound values are passed by reference

A scalar is something that is held within a single value. Examples are integers and characters. A compound value is something that cannot be held in a single value, but need additional storage for it. Examples of compound values are vectors, strings and objects.

When you pass a scalar to a block, a copy of the scalar is made and that is passed. If you modify the parameter within the block, the actual value is not affected (because the formal value is a copy of the actual).

If you pass an object (or other compound value), you are passing a reference to contents of the object. In C++ speak, you are passing a pointer to the object. You can use this pointer to modify the object if you so desire.

If you want to prevent the block modifying the contents of the object you must copy the object before passing it to the block. This can be done by calling the function System.clone().

5.3.8. Static declarations

A declaration that is marked *static* is one whose lifetime is greater than the block in which it is defined. A static declaration is allocated memory at the top level of the program (*main* package). Consider the following:

```
package P {
   class A {
      var a = 1
      static var b = 2
   }
}
```

This shows 2 block declarations. The first is a package named 'P'. The second is a class within the class named 'A'. Inside the class there are 2 variables ('a' and 'b'). The variable 'a' is allocated space inside the class 'A', but 'b' is marked static, so it will be allocated space in the *main* package.

The effect of this is that the lifetime of 'b' is greater than that of 'a' because instances of the class 'A' can come and go.

Static declarations are accessed just like regular declarations. It is only their location that changes. One difference is that static blocks do not have a 'this' parameter passed to them and they can be invoked through the name of their enclosing block. Consider:

```
class A {
   static function f() {                    // a static function
   }
}

A.f()                                       // invoke the f() in A
```

Here the function 'f' is declared static. You can therefore invoke it through the class name rather than having to do it through an instance of the class.

The initializer for a static variable must contain only other static variable or constants.

5.3.9. Static initializers

The keyword static may also be used to declare a section of code that is used to initialize a block statically. This is very similar to the Java™ concept of static initializers. A static initializer is executed when the program starts. Its purpose is to initialize the members of a block (class, especially) that are statically allocated. The initializer is executed early and only once.

Consider the following code:

```
class A {
   static var freelist = null               // static variable

  static {
     freelist = new FreeItem()              // initialize once
  }
```

The static initializer for the class A will be executed when the program starts and will only be executed once.

5.3.10. Forward declarations

Sometimes it is necessary to declare a block but not supply the body for the block. This happens when you have 2 mutually recursive blocks (2 blocks that invoke each other). Although the Aikido interpreter can invent variables for you, it will not let you call a variable it has invented. This is because it is invariably an error.

A forward declaration of a block tells the parser that a block will exist in the a scope but that it is not yet defined. This means that if you reference it in a later block it will refer to the correct declaration rather than trying to invent one in the current scope.

The syntax for a forward declaration is:

blocktype name ...

Where blocktype is one of *function, class, thread, monitor* or *package*. The ellipsis notation (...) tells the parser that this is a forward declaration.

You cannot use a forward-declared block as a base block for inheritance. Consider the following example:

```
package P {
   function A ...              // forward declaration of A

   function B {                // definition of B – call A
      A()
   }

   function A {                // definition of A – call B
      B()
   }
}
```

You can also use block extension to achieve the same effect (see section 5.3.19). The above example using block extension would be:

```
package P {
   function A {
   }

   function B {
      A()
   }

   extend A {
      B()
   }
}
```

The choice is yours.

Another trick you can play to handle forward declarations is to reference it via the ***this*** variable. Any references to members of a block are performed at runtime rather than compile time so they are late bound. This means that you can delay the lookup of the forward-referenced entity until runtime. Consider:

```
package P {
   function A {
```

```
        this.B()                    // refer to B via this – late bound
    }

    function B {
    }
}
```

This will be slower to execute because of the additional runtime lookup of the function in the object.

5.3.11. Function result types

It is sometimes convenient to be able to specify that a function returns a specific type. This is useful where you want to ensure that the function actually returns the specified type rather than being able to return any type.

A function return type is a clause following the function declaration that specifies an expression whose type is used to cast the actual return value of the function. If the cast cannot be performed a runtime error will result. This can happen if the types cannot be cast.

If the function result type is omitted, the function can return any type and is said to be generic.

The following example shows a function return type specification:

```
function getValue (s) : int {
    // function body
}
```

The colon character is followed by a simple expression whose type only is used (the value is ignored). This is very similar to the parameter type specifications. The effect is to cast any values returned by the function to the type of the expression (integer in this case). So, if the function tries to return a real number or string this will be converted to an integer and returned.

5.3.12. Block equivalence

A block in Aikido is a scalar variable in many respects. This allows you to assign them and retrieve their value. Consider the following:

```
function A {
}

function B = A
```

This defines a function A and then make another function B to be equivalent to A. They are, in effect, the same function with 2 different names. You can do anything the B that is possible with A.

5.3.13. Nesting blocks

A block can contain any declarations. This not only refers to variables, but nested blocks also. A block declared inside another block is local to the enclosing block. It has access to both its own declarations and also the declarations of the enclosing block.

```
function write (thing, stream) {
    function writeInt {
        // code to write an integer
```

```
    }

    function writeString {
        // code to write a string
    }

    if (typeof (thing) == "integer") {
        writeInt ()
    } elif (typeof (thing) == "string") {
        writeString ()
    } else {
        throw "Invalid type"
    }
}
```

The function *write* can be used to write integers and strings (to somewhere). It defines 2 internal functions called *writeInt* and *writeString*. These functions are called with the appropriate parameter type as it is determined by the body of the *write* function. The 2 internal functions have access to the parameters of the write function so these do not need to be passed to them.

The ability to nest functions is present in the Pascal family of languages but is absent from the C family.

Although the examples given in this section all use functions as the block type, exactly the same rules apply to classes, threads, packages and monitors.

This block structure is known as 'lexical scoping' in some literature. It is a powerful way of encapsulating code and data. A nested block can access the contents of any direct parent block as if the contents were declared inside the nested block itself. The usual access control protection doesn't apply to access to parent blocks. For example:

```
package A {
    private var x = 1              // private variable

    function B {                   // nested function
        var y = x                  // OK, x is also private to B
    }
}

var y = A.x                        // error: no access to private variable A.x
```

5.3.14. Block inheritance

Aikido allows blocks to inherit from other blocks. Only single inheritance is supported (no multiple-inheritance). It is important to remember that a block in Aikido can be a function, class, package, thread or monitor. This implies that a function can inherit from another function (and indeed any other block type). This is a lot different from other object-oriented languages.

Inheritance allows a block to take advantage of the functionality provided by another block and add new functionality for its own use. It is not the intention of this manual to delve into object-oriented theory – there are numerous excellent books on that subject.

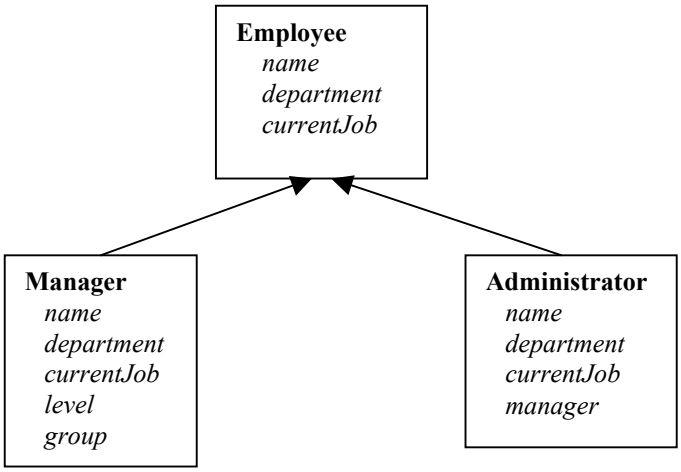
For the sake of clarity, we switch to using classes for examples of inheritance, just because that's the normal use of it. Let's build a hierarchy of classes to represent the people employed by a company.

```
class Employee (public name, public department) {        // an employee with a
// name and department
   protected var currentJob = “”                         // currently assigned job
public:
   function setJob (job) {                               // assign a job
      currentJob = job
  }
}

// a manager – also an employee, but with a management level attribute
class Manager (n, d, public level) extends Employee (n, d) {
   currentJob = “manager”                                // set current job
   var group = []                                        // group of employees
public:
   function addGroupMember (employee) {                 // add an employee to group
      employee -> group
   }
}

// an administrator employee who is assigned to a manager
class Administrator (n, d, public manager) : Employee (n,d) {
}
```

The above classes define a hierarchy as follows:



The syntax for block derivation is very similar to that of C++ except there is no private or protected inheritance.

What we have defined is a block hierarchy of 3 classes. Each class has a name and a set of attributes. The attributes are the variables inside the class. The blocks all take parameters and these parameters remain part of the block attributes (this is substantially different from C++). You will notice that some of the parameters have the keyword *public* in front of them. This allows the parameter to be accessible from outside the block. The default access protection for parameters is *private*.

The default access protection for all the members of a block is *private*. This prevents those members being accessed from outside the block. The keywords *protected* and *public* may be used to change the access protection of a member. See section 5.3.17.2 on page 5-61 for further details on access protection.

When an instance of the block is created, the parameters are assigned values and the body of the block is executed. When the body of a class, package or monitor is executed it is useful to refer to it as a *constructor*. In the case of the *Manager* class, the constructor executes and does 2 things: sets the variable *currentJob* to the string “manager”; and initializes the variable *group* to an empty vector. After that is done, the constructor returns and the object is constructed.

One step was omitted from the above discussion of constructors. Before any of the code of a constructor is executed, the code of the superblock (the parent in the derivation hierarchy) is executed. This sequence continues up the hierarchy until the top is reached (only one level in this example).

A subtlety of the inheritance mechanism is that there will actually be 2 copies of the attributes of the superblock when those attributes are passed as parameters. In the above example, the class *Manager* takes 3 parameters (*n*, *d*, and *level*). The *n* and *d* parameters are the *name* and *department* passed on to the superblock (*Employee*). Since parameters remain as part of the block, the *n* and *d* parameters will contain the same data as the *name* and *department* parameters of the *Employee* class. Thus there are actually 2 copies of the data. This is a little inefficient and care should be taken when designing a hierarchy.

So much for inheritance of classes - it is all pretty standard. A class is just a block, like packages, functions, threads and monitors. The inheritance of packages and monitors is similar to a class so is easily understood. The inheritance of a function or thread is a little different from other languages.

Let’s examine the inheritance of a function. Suppose we wanted to add a function to our class hierarchy to print out the attributes of each of the classes. We could add a function *print()* to each of the classes that prints to standard output.

```
class Employee (          ....) {          // as before
  // ...
public:
   function print {
      System.print ("name: " + name + " department: " + department)
   }
```

So much for the Employee class, what about the Manager? Well, it has to print the name and department too, so we could make the print function in 3 ways:

- print the name and department attributes from directly from the employee class
- call the employee class's print() function as the first thing done
- derive the manager's print function from the employee's print() function

The first is not good because we might add another attribute to the employee that we want printed and would then have to add it the print() function of the manager too.

The second is fine, but what happens if we introduce another level into the hierarchy between the manager and the employee? We would have to change the manager's print function accordingly.

The third is the most flexible as it insulates us from any changes to the hierarchy. This is how you would do it:

```
class Manager (......)      {          // as before
   // ....
   function print extends print {
      System.print (" level: " + level)
      System.print (" group: ")
      foreach e group {
         e.print()
      }
   }
}
```

So, a function can be derived from another function. Apart from the insulation issue discussed above, what is the difference between deriving from a function and just calling the would-be superblock function as the first statement?

The answer lies in the definition of a function in Aikido. A function is a block, just like a class. The only difference is in the lifetime of the memory allocated for the function's data. When you create an instance of a class, the memory is allocated until the class is deleted. When you call a function, memory is also allocated (usually on the program stack) but is freed when the function returns. If you call a function inside another function, there is no way to access any memory of the called function after it returns (the memory is free).

If you derive a function from another function, the call of the superblock function is made in the **same context** as the function being called. This allows the superblock function to expose some public data to its children (just like a class would). This makes for an efficient way to pass information between functions in the same derivation hierarchy.

Another use of function derivation is to enforce preconditions on a set of functions. Consider the following hierarchy:

```
function checkRange (n) {
   if (n < 0 || n > 255) {
      throw "Invalid number value: " + n
   }
}

function setX (x) extends checkRange (x) {
}

function setY (y) extends checkRange (y) {
}
```

This is a trivial example, but shows one use of function derivation.

5.3.15. Interface inheritance

Just like other blocks, interfaces (see section 5.3.18) may be derived from other interfaces. They cannot, however, be derived from other block types, and blocks (other than interfaces) cannot be derived from interfaces. Consider the following examples:

```
interface A {
   function f (a,b)
}

interface B extends A{          // OK, B is derived from A
   function g (c,d)
}

class C extends A {             // error: cannot derive class from interface
}

class D implements A {          // OK
   function f(a,b) {}
}

class E implements B {          // error: function g() is not defined in E
   function f(a,b)  {}
}

interface F implements B {      // error: interfaces cannot implement interfaces
}

class G extends  D {                    // OK, G also implements A
 }
```

5.3.16. "Virtual" functions

Aikido does not have the concept of a "virtual" function. Actually, it doesn't have the concept of a non-virtual function as all block members are inherently "virtual". If a derived block provides a member with the same name and number of parameters as an accessible member in a base block, then the derived block member overrides the member in the base block. This only applies to block members that are themselves blocks (functions, classes, etc.).

For example:

```
class A {
   private function f {                    // private function
   }
   public function g {                     // protected is also ok
   }
}

class B extends A {                              // A is base block of B
   private function f {                    // no override
   }
   public function g {                     // overrides A.g
   }
}
```

Because the resolution of block members is done at runtime rather than at compile time, the override of a block member only comes into affect if the overridden member is invoked from within the base class. For example:

```
class A {
   private function f {                    // private function
      System.println ("A.f")
   }

   public function g {
      System.println ("A.g")
   }

   function show {
      f()
      g()
    }
}

class B extends A {                              // A is base block of B
   private function f {                    // no override
      System.println ("B.f")
   }
   public function g {                     // overrides A.g
      System.println ("B.g")
  }
}

var a = new A()
var b  = new B()
a.f()                          // prints A.f
a.g()                          // print A.g
a.show()                       // prints A.f followed by A.g

b.f()                          // prints B.f
b.g()                          // print B.g
b.show()                       // prints A.f followed by B.g
```

This mechanism allows a derived block to change block type of a member in a base block. This can be useful if, for example, you wanted to change a function in a base block into a thread.

```
class X {
   public function run() {
   }
}

class Y extends X {
   public thread run() {
   }
}
```

I can't really think of a reason for overriding a class (say) with a function, but it is indeed possible to do so if someone finds a good application.

5.3.17. Block member resolution

When you need to access a member of a block you do so by use of the '.' operator. The actual resolution of the member is not done when the program is parsed, but rather when it is actually running. Because the language is dynamically typed, there is no way for the parser to know what the real type of a variable is until it is running.

This has some advantages and disadvantages. The disadvantage is that you will not get an early warning of access to undefined members in blocks. Other, statically typed, languages detect undefined members when the compiler runs.

The advantage is that there is rarely a need to cast an object of one type to one of another. For example, consider the following classes:

```
// a generic tree node
class TreeNode (opcode) {
public:
   var left = null                    // left pointer
   var right = null                   // right pointer
}

// tree node for a number
class Number (value) extends TreeNode (NUMBER) {
    public function getValue() {              // get the value of the number
      return value
   }
}

// tree node for an identifier
class Identifier (spelling) extends TreeNode (IDENTIFIER) {
   public function getSpelling() {
   return spelling
   }
}


// print a whole tree, a node at a time
function printTree (tree) {
   if (tree = null) {
```

```
        return
    }

    printTree (tree.left)                    // print left tree

    switch (tree.opcode) {                   // look at opcode
    case NUMBER:
        System.print ("NUMBER: ")
        System.println (tree.getValue())     // just call the getValue() function
        break

    case IDENTIFIER:
        System.print ("IDENTIFIER: )
        System.println (tree.getSpelling())
        break
    }

    printTree (tree.right)                   // print the right node
}
```

If you can follow the *printTree*() function, it first calls itself recursively to print the left node, then it prints the node it is passed, then does the same for the right node. The important section is in the switch statement. It switches on the opcode and has cases for NUMBER and IDENTIFIER. A number is a tree node that holds the value of the number. We want to print this as part of the print routine.

In a static typed language, you would have to cast the *tree* pointer to the appropriate type before dereferencing it to call the getValue() function. Let's see what the equivalent function would be in C++ (given the same classes and functions).

```
void printTree (TreeNode *tree) {
    if (tree == NULL) {
        return ;
    }

    printTree (tree->left) ;

    switch (tree->opcode) {
    case NUMBER: {
        Number *num = static_cast<Number*>(tree) ;          // cast to Number*
        std::cout << "NUMBER: "  << num->getValue() << '\n' ;
        break ;
        }

    case IDENTIFIER: {
        Identifier *id = static_cast<Identifier*>(tree) ;   // cast to Identifier
        std::cout << "IDENTIFIER: "  << id->getSpelling() << '\n' ;
        break ;
      }
    }
    printTree (tree->right) ;
}
```

Notice that the code is very similar, the only difference being the lack of a cast to the real type for the tree node.

5.3.17.1. The ***this*** variable

When in a block, a special variable with the name ***this*** is created by the interpreter. The ***this*** variable is set up to point to the current object so that you can access an members of the current object by using the notation:

```
this.member
```

Unlike C++ the name ***this*** is not a reserved word. It is the name of a system defined variable present in all blocks. Any block that is a (non-static) member of another block is passed the ***this*** variable as an implicit parameter when invoked.

The ***this*** variable is actually constant and cannot be written to by the program.

5.3.17.2. Access control

The access to the member is controlled by the access level assigned to the member in the block. There are 3 access levels:

- *private*: member cannot be accessed outside block
- *protected*: member can be accessed by derived blocks
- *public*: member can be accessed from anywhere

The default access mode for all members is private. You can change the access mode in 2 ways: prefix the member's declaration with the appropriate access keyword; or change the access mode for the whole block.

The first method is how it is done in the Java™ language:

```
class Employee {
   var salary = 0                          // private member
   protected function changeManager()      // protected member
   public function print()                 // public member
}
```

If you omit the access mode keyword, the default for the block is used.

The second method is to change the current access mode. This is the C++ style.

```
class Employee {
   var salary = 0                          // as before, private member
protected:                                 // current mode now protected
   function changeManager() {              // protected member
   }
public:                                    // current mode now public
   function print() {                      // public member
   }
}
```

Which to use is really a matter of style. The use of the Java™ method makes each member explicit about its access mode. Use of the C++ method saves typing and time.

5.3.18. Implementing interfaces

As in the Java™ language, Aikido allows a special block called an *interface* to be defined. This specifies that any block that *implements* this interface must provide at least the members defined in the interface. Consider the following interface:

```
interface WindowInterface{
   function open (x,y)
   function close
}
```

This defines an interface that contains 2 functions. Any block that implements this interface must provide at least those functions:

```
class Window implements WindowInterface {
   public function open (x,y) {
      // open the window at coords
   }

   public function close {
     // close the window
   }

  public function resize (w, h) {                    // extra function not in interface
     // resize the window
   }
}
```

If the block being defined does not honor the contract and provide the required members a compile time error results.

The use of interfaces allows stricter control over the type system in the language. In the above example, the name WindowInterface may be used to refer to any block that implements that interface.

Whereas Aikido only supports single inheritance of blocks (implementation inheritance in the parlance), it does support multiple interface inheritance.

```
interface Aint {
   function f()
}

interface Bint {
   function g()
}

class C implements Aint, Bint {
   function f() {}
   function g() {}
|
```

5.3.19. Block extension

Aikido allows any block to be extended by adding new members to it. This feature is not available in languages such as C++ where you have to either modify the original source code or derive a new class to extend a class.

Consider a package for string manipulation. Strings are always a good example for extension, as you can never design a package containing all the features that everyone will want. Suppose our string package is defined as follows:

```
package company.Strings {                    // package for all strings
    public class String (value) {            // a string class
    public:
      function append (s) {
        value += s
      }

      // and other members
    }
}
```

We notice that the *String* class does not provide a *toUpperCase*() function. How do we add one? One solution is to go into the source code and add it with a text editor. This may not be either desirable (if the code is shared) or possible (you may not have write access to the file).

Another solution is to derive a new class from the *String* class. Call it *MyString*. Now we add the function *toUpperCase*() to the *MyString* class. OK, but now we need to change all instances of *String* to *MyString* so that we may use the *toUpperCase*() function. This is definitely doable, but is messy and error prone.

Another solution: define the function *toUpperCase*() to take a *String* as an argument and operate directly on it. This might work if the contents of the *String* are publicly accessible, otherwise you will have to go in and modify the access mode for it.

The best solution is to extend the *String* class with the desired function. Here's how you would do it:

```
package company.Strings {                    // go into the correct package
   extend String {                           // extend the String class
      function toUpperCase() {               // add the function
         // implementation of function
      }
   }
}
```

After this is done, the *String* class now has a new function. All existing code that uses the old String class will still work, as they won't call the new function. You can now use the *toUpperCase*() function on any object of type *String*.

Notice that is was not necessary to extend the package. This is because packages are automatically extended. If the *String* class was, for example, inside a class rather than a package you would have to first extend the outer class, then extend the *String* class.

Extension is not limited to adding functions to classes. You can add anything to any block, including new code to a function.

If you extend a block you can add parameters to the extension. The parameters must be given default values as code that is ignorant of the extension will not pass values for the new parameters you add. For example, you can extend the String class and add a parameter:

```
package company.Strings {
   extend String (defaultLength = 0) {
      // any other extensions here
```

```
    }
}
```

Here, we added a new parameter to the String class in our company's string package. Any existing code will not see it and the value of 0 will be passed.

5.4. Functions

A function is a block that is called. When a function is called, control is transferred to the body of the function after all the parameter values have been assigned. The function then executes until it returns. It can return a value to the caller by use of the *return* statement.

```
function factorial (n) {
    if (n < 2) {
        return 1
    }
    return n * factorial (n – 1)
}
```

This shows the ubiquitous factorial function that calculates (inefficiently I might add) the formula n!. It is passed a single parameter and calls itself recursively until the value of the parameter reaches the value 1. Don't try this for large values of n as it will overflow the stack quite quickly. The function returns the value of the current calculation by use of the *return* statement.

Functions are invoked by a function call expression:

```
var f = factorial (10)                    // call the factorial function and assign to x
```

A function always returns when the thread of control drops off the end. If the caller of the function expects a value to be returned and one is not returned then a value of type *none* is returned to the caller.

5.4.1. Native functions

As with any interpreted language, there needs to be a way for a program written in Aikido to make calls out to code written in another language. Typically this is used to make calls to the operating system in order to provide extensions to the facilities present in the language.

Aikido allows a function to be defines as *native*. This tells the interpreter to look for a native symbol in the symbol table of the process and invoke it whenever the function is called. For example:

```
native function sin (d)                    // mathematical sin()
native cos (d)                  // cosine()
```

Note that the keyword *function* is optional.

It would be an inefficient use of resources to code these functions directly in Aikido since they already exist in the operating system.

The declaration of a native function specifies the names of the parameters passed to the function. The names are for documentation only but the number of parameters is important and is checked by the interpreter. Unlike a regular function, the parameters may not be given a type or a default value. You can, however, use the ellipsis notation to specify that the function takes a variable number of parameters. Consider the following examples:

```
native printf (format, ...)          // variable parameter list
```

```
native f (a,b,c,d,e,f)          // 6 parameters
```

The parameters for a native function can be declared to be passed by reference rather than by value. This is achieved by prefixing the name with the reserved word **var**. For example:

```
native func (var a, b)
```

The first parameter passed to the native function will be passed by reference rather than by value. This means that the native function can write to the caller's variable.

5.4.2. Raw native functions

Native functions come in 2 flavors:

- Regular native functions. Where the code for the function is written in C++ and is linked into the Aikido interpreter from a shared library. Regular native functions have an interface function that obeys the rules for a Aikido native function
- Raw native functions. Where a function is available in the operating system and is a 'simple' function you can call it directly from Aikido without having to write a special mapping function in C++0

Raw native functions are used when there is no regular native function available. This means that the language first looks for a regular native function and if it can't find it, a raw native function is created. Consider the following example:

```
native malloc (size)        // raw native
native sin (v)              // regular native
```

How do you know if a function is raw or regular? There is no real way to do it just by looking at the Aikido program. You will have to use the OS to find out if the symbol for the regular function exists. Symbols for regular native functions are always called 'Aikido__<name>'.

Declaring a function as 'raw native' tells the interpreter that this function exists in the operating system and the interpreter arranges to call it directly. There are certain rules for raw natives:

1. The symbol must exist in the operating system or in a library loaded into the interpreter
2. There is a maximum of 10 parameters
3. The parameter types must be simple:
   - Integers, characters, bytes or enumeration constants
   - char * pointers
4. The return type must be simple (same as parameters)
5. No floating point is allowed.

When the call is made, integral parameters are passed as ints and strings are passed as char * pointers.

5.5. Threads

A thread is very similar to a function. It is defined in the same way except by use of the *thread* reserved word rather than *function*.

```
thread serverloop (stream) {
   while (!System.eof (stream)) {
      // code for thread
   }
}
```

The behavior of a thread is, however, much different from that of a function. When a thread is invoked the invoker resumes execution immediately without waiting for the thread to execute. The thread executes in parallel with all the other threads in the system (including the main program). When the thread returns it terminates.

Consider the following code which could be seen in a server program.

```
foreach channel channels {
   var stream = getStream(channel)
   serverLoop (stream)
}
```

The loop creates a number of threads, all of which execute in parallel. The loop does not wait for the threads to finish. The return value from the call to the thread is a stream connected to the thread as its *input* and *output* streams. This can be used to communicate with the thread.

There can be many executing threads running at once in the program. Care must be taken to ensure the integrity of any data shared among threads (as with any multithreaded program).

See the Chapter 10 for further information on threads.

5.6. Classes

A *class* is a user-defined type. It is the mainstay of object-oriented programming. When you define a class, you are defining a data type and an associated set of operations (known as *methods* in the literature). You can create *instances* of a class using the *new* operator. An instance of a class created in this way is a value of type *object*.

A class is declared using the reserved word *class*.

For example, consider the following class declaration:

```
class User (name, password = "") {
public:
   function validatePassword (pass) {
      return pass == this.password
  }

   function getName {
      return name
   }

   System.println ("User " + name + " created")
}
```

This declares a class called *User* with 2 attributes (*name* and *password*). There are also 2 member functions (*validatePassword* and *getName*). In addition, the *password* attribute has a default value of the null string. The constructor for the class prints something to standard out (probably for debugging, but who knows)

Instances of this class may be created:

```
var me = new User ("Dave", "evaD")          // both name and password supplied
```

```
var you = new User ("Sandra")                    // only name supplied, password = ""
```

When each instance is created, the constructor prints information about the object just created to the screen.

5.6.1. Operator overloading

One powerful feature of object-oriented programming is the ability to use a user-defined type as if it is a built-in type. The types built in to the language allow the use of the expression operators to manipulate them. For example, given the following:

```
var x = 1
var y = 10
var z = 20
```

You can perform calculations using expression operators:

```
var result = x * y + z
```

This works because the language knows all about the type *integer* and how to manipulate it. The same can be said for the other built-in types. For example, if you say:

```
var outs = "The result is " + result
```

The language knows to convert the integer value *result* to a string before performing the addition.

In this sense, the built-in types are objects for which the language has built-in rules and has code to explicitly deal with them. What, then happens if you try to use an object for which the language has no rules? Consider the case of a complex number (used frequently in arithmetic and also in C++ documentation). This is usually implemented as a class:

```
class Complex (re, im) {
}
```

You can create instances of it using:

```
var v = new Complex (1, 3.1)
var c = new Complex (1.2, 2.1)
```

So far so good. What happens if you now want to add them together? One solution is to define a function taking 2 parameters of type Complex and perform the addition. This won't work unless the function has access to the internal representation of the class (either directly to the variables, or through accessor functions).

```
function complexAdd (a, b) {
   // add the Complex numbers a and b
   return result
}

// add 2 complex numbers together

var r = complexAdd (v, c)
```

This is rather untidy. A better approach would be the ability to overload the '+' operator and provide a definition for the Complex class.

```
class Complex (re, im) {
   public:
      operator+ (c) {
         // add the contents of this object to 'c'
      }
}
```

You can then use the normal expression syntax to perform the calculation:

```
var result = v + c
```

So, what are the rules for overloading operators? Firstly, the operator functions (they are treated exactly like functions) must be inside a class (or monitor of course). Secondly, you can only overload the following operators:

*	( )	->	[ ]	+	-	/
%	~	^	!	&	|	<<
>>	>>>	<	>	<=	>=	==
!=	sizeof	typeof	foreach	cast	in	

Thirdly, operators are not inherited from superblocks. This is because operators are only valid for classes and the superblock of a class may be something other than a class. Also, it might be dangerous to inherit an operator from another class when you are providing additional functionality on top of the class.

5.6.1.1. Binary and unary operators

A binary operator is defined as a function taking one argument. The function is called with the *this* variable set to the left side of the binary operation and the argument set to the right side. The expression on the right of the operator does not need to be the same type as the left.

For example:

```
class X {
   public operator * (p) {
   }

   public operator~() {
   }
}

var x = new X()
var r = x * 2
```

Here, the * operator has been overloaded. The operator function is called with ***this*** set to *x* and *p* set to the integer value 2.

For a unary operator, the operator function is defined with no arguments. For example, the operator for ones-complement (~) is overloaded in the above example. If this is called as:

```
var q = ~x
```

The function will be called with ***this*** set to x.

5.6.1.2. The subscript operator [ ]

The subscript operator is defined as follows:

```
operator[] (i, j = -1) {
}
```

It is passed either 1 or 2 parameters, depending on the form of the subscript operation. If it is passed one parameter, the value of the second parameter is –1.

Which form is used depends on whether the subscript is a simple element or a range of elements.

```
var r1 = val[2]                 // called with one parameter
var r2 = val[2:6]               // called with 2 parameters
```

The operator function can determine which form of subscript is required by looking at the value of the second parameter (j)

5.6.1.3. The function call operator ( )

By defining an operator of this type, you allow the programmer to call an object as if it is a function. The operator function can take any number of parameters as long as the calls to the function match those parameters.

```
class T {
   public operator() (a, b, c) {
      return a * b + c
   }
}

var t = new T()
var r = t (2,4,5)
```

sets the variable r to value 2 * 4 + 5 (13).

This form of operator overloading can be used to perform what are called ‘functors’ in the C++ standard library.

5.6.1.4. The stream operator (->)

Aikido has a built-in operator for sending and receiving data over streams. Usually this operator is used directly with values of type stream, but is it useful to be able to encapsulate the stream value in an object. It can also be useful to treat an object as if it was a stream.

For example, suppose we have a class defined to represent a window on a screen. It might be useful to be able to send data to the window and have it display the data, scrolling if necessary. It also might be useful to allow the window to be used as input and enable the data to be read from it. Let’s define 2 classes, one for a window to be used as output, and another for a window to be used as input.

```
class OutputWindow {
public:
   operator -> (data, isoutput) {
      if (isoutput) {
         // output to window, scrolling
```

```
        } else {
            throw "Output window cannot be used for input"
        }
    }
}

class InputWindow {
public:
    operator -> (stream, isoutput) {
        if (isoutput) {
            throw "Input window cannot be used for output"
        } else {
            // read from window and write to stream
        }
    }
}
```

They may be used as follows:

```
var out = new OutputWindow()        // new output window
var in  = new InputWindow()         // new input window

"hello world" -> out                // write string to output

var str = ""                        // variable to hold input
in -> str                           // read from input window
```

As can be seen, the stream operator takes 2 parameters. The first is the expression to the left or right of the -> operator. The second is a value of 1 if the first parameter is to the left of the -> operator, and 0 if it is to the right. Another way of thinking of this is that the *isoutput* parameter is set to 1 if the stream operator is pointing to the object, zero otherwise. That is, *isoutput* means that the object is being output to.

Which values are chosen depends on which of the operands of the -> operator contain the stream operator function. If both the operands contain a stream operator function, the version with the second parameter set to 0 is chosen.

In the above example, the stream operator for *OutputWindow* will be called to write the string. It will be passed the variable *out* as the ***this*** parameter, and the string "hello world" as the *data* parameter. The *isoutput* parameter will be set to 1.

For the case of reading from the *InputWindow*, the stream operator function will be passed the variable *str* as the *stream* parameter and the value 0 as the *isoutput* parameter.

So, how would the stream operator for InputWindow be implemented to work for all cases?

This is one possible implementation:

```
operator -> (stream, isoutput) {
  if (isoutput) {
      throw "Cannot use input window for output"
  } else {
      readString() -> stream
  }
}
```

The function *readString*() reads a string from the internal data structures of the window. The -> operator is then used to output this to the *stream* parameter. This operator can now be used to read from the window into any variable. Also, if the stream parameter is itself an object with a stream operator, it will be invoked. So, you can do this:

```
var in = new InputWindow()
var out = new OuputWindow()

in -> out
```

And copy the data directly from the input window to the output window.

5.6.1.5. The *sizeof* and *typeof* operators

The *sizeof* and *typeof* operators are used to calculate the size and type of value respectively. They can be used on any internal type. They can also be overloaded to work with any user-defined type. They both take no parameters. For illustration, consider the following class. The member's *type* and *size* can be anything meaningful to the class.

```
class T {
    var type = "something"                  // set to something meaningful
    var size = 1234                         // also set to something meaningful

    public operator sizeof() {
        return size
    }

public operator typeof() {
    return type
}

var t = new ()
var s = sizeof (t)
var t = typeof (t)
```

The *sizeof* operator returns an integer value representing the size of the class. This does not have to be the number of bytes allocated for the class, but can be anything meaningful for the class. For example, if the class contained a vector, the *sizeof* operator could return the sizeof the vector. If no sizeof operator is defined for the class, the default return value is the number of variables in the class.

The typeof operator is used to inquire about the type of an object. The operator can be applied directly to an object without the typeof operator overload, in which case the type of the class of which the object is an instance will be returned. The typeof operator does not need to return a string value – it can return anything that is useful. It can even make up the type on the fly:

```
class T (name) {
    public operator typeof() {
        return "T " + name
    }
}
```

This makes it a truly dynamic type.

5.6.1.6. The *foreach* operator

The *foreach* operator allows a class to provide an *iterator*. The *foreach* statement is used in Aikido to iterate though all the values of an expression. For built-in types, the *foreach* operator can determine what constitutes the elements of an expression (for example, the elements of a vector).

By overloading the *foreach* operator (called an operator, but really a statement) you can provide the programmer with the ability to use the *foreach* statement on an object of your class as if it is a built-in type.

For example, suppose we have a class called *Vector*, that implements features on top of the built-in *vector* type.

```
class Vector {
   var value = []
public:
   operator foreach() {
      return value
   }
  // other member functions and operators
}


var v = new Vector()
foreach element v {
   f (element)
}
```

The *foreach* operator has 2 forms: simple and complex

- Simple form. The operator has no parameters and must return something for which the *foreach* operator is valid. This is the form in the above example.
- Complex form. The operator takes one parameter that is a reference to a 'context'. This form is used then you have a complex object for which the simple form will not suffice.

Consider the following example of the complex form of the *foreach* operator:

```
class List {                          // single linked list
   var start = null                   // start of list
   var end = null                     // end of list

   class Item (public value) {        // a single list item with a value
      public var next = null                  // and a next pointer
   }

   public function insert (v) {       // insert value into list
      var item = new Item (v)         // make new list item
      if (end == null) {              // list empty?
         end = item                   // assign end of list
         start = item                 // assign start of list
      } else {
         end.next  = item             // link to end of list
         end = item                   // this is now end of list
      }
   }

   public operator foreach (var it) { // iterate through all items
```

```
    if (typeof (it) == "none") {        // first time in?
        it = start                      // yes, set context to first item
        return it.value                 // return item value
    } elif (it == end) {                // beyond last iteration?
        it = none                       // tell caller that we are finished
    } else {
        it = it.next                    // in the middle of list, move context on
        return it.value                 // and return current value
    }
}
```

In the above example the *foreach* operator takes a single parameter. This parameter is a reference to a variable used to hold the current iteration *context*. The caller of the foreach operator sets this context to the value *none* before the first call of the operator. This tells the operator that the caller wishes to start a new iteration. The operator sets the context to the current position in the list and returns the value of the item for that iteration. Then, for each additional iteration, the caller passes the context set by the operator. The operator increments this context and returns the item for each iteration.

When the operator detects that we have reached the end of the list it sets the context to *none* again and the caller stops the iterations.

If you think about what the context is: it is the context of the previous call to the operator by this caller. To get to the next context we need to move it on before returning the value.

5.6.1.7. The cast operator

When you convert a value of one type to a value of another type it is commonly referred to as 'casting' the type. An object may be defined to include an operator that is called when a cast from an instance of the object to another type is attempted. The operator is called the 'cast operator' and is defined as:

```
class A {
    public operator cast (v) {
        // convert to type of v
    }
}
```

Now, when you try to convert the object to another type, the cast operator function will be called:

```
var a = new A()
var b = cast<string>(a)
```

This will call the 'operator cast' function passing an expression of type 'string' to it as a parameter. The cast operator should examine the type of the expression it gets as a parameter to ensure that the conversion is possible. It should return a value of the appropriate type after the cast is done. For example:

```
class Name (n : string) {
    public operator cast (v) {
        if (typeof (v) != "string") {
            throw "Illegal cast of name to " + typeof (v)
        }
        return n                        // n is already a string
    }
}

var name = new Name ("Dave")                    // instance of Name object
```

```
var hello = "Hello " + cast<string>(name)          // cast to string and append
```

5.6.1.8. The in operator

The Aikido language has an operator named *in*. This is used to test a value for membership of another value. For example, it may be used on a vector to test if a value is a member of a vector:

```
var v = [1,2,3,4]
var x = 3 in v                    // is the value 3 a member of v (yes)
```

For user defined types, the block can provide an instance of the operator *in*. This operator will be called when it is desired to test for block for membership. For example:

```
class List {                              // single linked list
   var start = null                       // start of list
   var end = null                         // end of list

   class Item (public value) {            // a single list item with a value
      public next = null                  // and a next pointer
   }

   public function insert (v) {           // insert value into list
      var item = new Item (v)             // make new list item
      if (end == null) {                  // list empty?
         end = item                       // assign end of list
         start = item                     // assign start of list
      } else {
         end.next  = item                 // link to end of list
         end = item                       // this is now end of list
      }
   }

   public operator in (v) {                        // test for membership
      for (var item = start ; item != null ; item = item.next) {          // go through all
items
         if (item.value == v) {           // value matches?
            return true                   // return true
         }
      }
      return false                                 // no match
   }
```

As can be seen, the *in* operator takes a single parameter and returns *true* or *false* depending on whether the parameter is a member of the object or not.

The above example may be used as in:

```
var list = new List()
var found  = 10 in list
```

Chapter 6. Expressions

An expression consists of a series of operators and operands. An operator is a token that performs some function on the operands. For example, the expression:

```
(quantity * price) – 3.1
```

is an expression containing 2 operators and 3 operands. The operators are the '*' and '-' symbols, with the operands being the variables *quantity* and *price*, and the number 3.1.

All expressions produce a value. From the simplest expression (a single number or variable), to the most complex expression with many operators.

The type of an expression depends on the types of the operands in the expression. In the above example, the value would be *real*. An expression can have any type including vector, map, function, class, etc.

The Aikido language provides a rich set of operators and defines a meaning for them for all the built-in types. You can provide your own operator functions for user-defined types. See section 5.6.1 for details on how to overload operators for user-defined objects.

The expression operators are organized into a set or priorities. An operator with higher priority is always executed before one of a lower priority. For example, the multiply operator (*) is higher priority than the addition operator (+) so, consider the following:

```
1 + 7 * 4                    // value 29
(1 + 7) * 4                  // value 32
```

The following table shows the relative priority of all the operators. The highest priority is at the top of the table.

new operator
[ ] ( ) . ++ -- (postfix)
sizeof typeof cast ! - + ~ ++ -- (unary)
* / %
+ -
<< >> >>>
< > <= >= ***instanceof in***
== !=
&
^
|
&&
||
? :
->
= += -= *= /= %= <<= >>= >>>= &= |= ^=

6.1. Primary expressions

The simplest expression is the primary expression. These are the references to variables, numbers, strings and other literals. The primary expression is the highest level expression. They have priority over all the other expression operators. Primary expressions consist of the following:

```
( expression )              // parentheses
identifier                  // a variable
number                      // number literal (integer or real)
true                        // integer value 1
false                       // integer value 0
null                        // null object
[ expression list ]                 // vector literal
{ map element list }        // map literal
new-expression              // create a new object
direct-operator-expression  // call an operator directly
`statements`                // inline block
anonymous-block             // anonymous block definition
```

6.1.1. Identifiers

An identifier is a reference to a variable. The interpreter searches for the variable by traversing the scope levels in the program in the following order:

1. the current scope
2. the static scope list, consisting of the current block and traversing up the chain to the enclosing blocks
3. any blocks listed in the using statements of the blocks as the search progresses up the enclosing blocks

Usually the variable must exist before it can be used. However, the interpreter can invent the variable if it doesn't exist and you are assigning to it. This means you don't have to use a variable declaration to make a new variable – if the first reference to it is an assignment statement it will be declared for you.

6.1.2. Numbers and characters

A number may be an integer or real literal. The integer literal may be in decimal, octal, hexadecimal or binary. A real literal must include a decimal point (period) and may also include an exponent.

A character literal is a character enclosed in single quotes. A character literal is treated as an integer whose value is the ASCII code for the character. Only 8-bit characters are supported. An escaped hex or octal constant may also be used inside the quote marks.

6.1.3. Vector and Map literals

A vector literal defines a vector value. They consist of an optional series of expressions enclosed in square brackets and separated by commas. If there are no expressions in the brackets, the vector will be of zero length. Consider the following examples:

```
var x = [1,2,3]             // vector of 3 elements
var y = []                  // empty vector
var m = [[1,2], [3,4], [5,6]]   // vector of vectors
```

A vector literal may be used anywhere a value may appear.

A map literal defines a map value. It consists of a series of *value=value* pairs enclosed in braces and separated by commas. The map may be empty. Consider the following examples of map literals:

```
var m = {1 = "a", 2 = "b"}          // map of  2 elements
var n = {}                                  // empty map
```

A map literal may be used anywhere a value may appear.

6.1.4. Strings

A string literal is a set of characters enclosed in double quote marks. They may be used anywhere a value may appear.

```
"this is a \"string\"."
```

Any special characters in the string must be escaped my preceding them with a backslash. A string has a length and is not terminated by a zero byte as in C. The string may include any characters.

6.1.5. Inline blocks

An inline block is a section of code that is placed in an expression. The code may contain any valid statements (as can appear inside a function, for example). The code is executed when the expression is executed and is expected to yield a value. The '*return'* statement is used to return a value from the inline block.

An inline block is defined by enclosing the code in a pair of *backtick* characters (`). The blocks may be nested. This is similar to the use of backticks to execute subcommands in many shell programming languages such as the Bourne Shell. Consider the following example:

```
var x = `return y + 1`              // equivalent to x = y + 1

var sum = `var r = 0                // more complex statements
   for (var j = 0 ; j < y ; j++) {
      r += j*j
   }
   return r`
```

The code in the block is executed in its own scope, meaning that any variables declared in the block are unique to that block. Variables in enclosing blocks can, of course, be used in the block.

If the block does not return a value, a value of type *none* is returned.

The use of inline blocks is encouraged in cases where a small function would be defined for the code and called only once. They can decrease readability of the code, so care must be taken.

6.1.6. Anonymous blocks

An anonymous block is an expression that defines a block with no name. This may be used to define a function (say) for single use or to be passed as a parameter to another block. The blocks that may be defined in this way are:

- functions
- threads
- classes and monitors

Consider the following trivial example:

```
// function to test the value of a function
function tester (func, para, value) {
   if (func (para) != value) {                    // call the function and test value
      "FAILED\n" -> stderr
      System.exit (1)
}

tester (function (p) { return p * p }, 2, 4)
tester (function (p) { return p*p*p }, 2, 8)
```

The function 'tester' is called with 3 parameters: a function, a parameter to that function and the value expected from the function. The calls pass an anonymous function expression taking one parameter and performing some calculation on it.

As an aside, consider the following:

```
var func = function (x,y) {
   // code
   }
```

This operates in a very similar fashion to a regular function definition:

```
function func (x,y) {
   // code
}
```

With a major difference: the former case declares a variable that happens to contain a function value whereas the latter defines a block. The difference is subtle yet important. In the case of the former you cannot derive a new block from the variable and cannot extend it using the *extend* statement.

As an anonymous block is a block, passing it to another block as an argument creates a closure.

6.2. Arithmetic operators

The set of arithmetic operators provided by Aikido are:

ex * *ex*	Multiplication
ex / *ex*	Division
ex % *ex*	Modulus
ex + *ex*	Addition
ex – *ex*	Subtraction
- *ex*	Unary minus
+ *ex*	Unary plus

Each of the arithmetic operators can operate on a selection of built-in types as follows. Where an integer type is mentioned, the rule also applies to chars.

Type combination	Meaning	Result type
*integer * integer*	Multiply the 2 integers	*integer*
*integer * real*	Convert integer to real and perform multiply	*real*
*real * integer*	Convert right integer to real then multiply	*real*
*real * real*	Floating point multiplication	*real*
integer / integer	Integer division	*integer*
integer / real	convert integer to real then divide	*real*
real / integer	convert integer to real then divide	*real*
real / real	floating point division	*real*
integer % integer	modulus of 2 integers	*integer*
real % real	modulus of 2 reals	*real*
integer + integer	integer addition	*integer*
integer + string	integer converted to string then prepended to string	*string*
integer + vector	integer is prepended to vector	*vector*
integer + enumconst	integer is used to move to next (n) constants in enum	*enumconst*
integer + real	integer is converted to real and addition performed	*real*
real + integer	integer is converted to real then added	*real*
real + string	real is converted to string and prepended to string	*string*
real + vector	real is prepended to vector	*vector*
real + real	real addition	*real*
string + integer	integer is converted to string and appended to left operand	*string*
string + real	real is converted to string and appended to left operand	*string*
string + string	strings are joined	*string*
string + vector	string is prepended to vector	*vector*
string + enumconst	name of enumconst is appended to string	*string*

string + *block*	name of block is appended to string	*string*
vector + *vector*	vectors are appended	*vector*
vector + *any type*	right operand is appended to end of vector	*vector*
map + *map*	maps are appended	*map*
enumconst + *integer*	integer used to move to next (n) constant in enum	*enumconst*
enumconst + *string*	enumconst name is prepended to string	*string*
enumconst + *vector*	enumconst is prepended to vector	*vector*
block + *string*	block name is prepended to string	*string*
block + *vector*	block is prepended to vector	*vector*
object + *vector*	object is prepended to vector	*vector*

integer – *integer*	integer subtraction	*integer*
integer – *enumconst*	integer used to move to (n) previous constant in enum	*enumconst*
integer – *real*	integer converted to real and subtraction performed	*real*
real – *integer*	integer converted to real and subtracted	*real*
real – *real*	real subtraction	*real*
enumconst – *integer*	integer used to move to (n) previous constant in enum	*enumconst*
vector – *vector*	difference in vectors. Those elements not present in both vectors.	*vector*

- *integer*	unary minus of integer	*integer*
- *real*	unary minus of real	*real*

6.3. Bitwise operators

Aikido provides a comprehensive set of operators that operate on the bits of an integer value. They also apply quite naturally to strings and vectors in certain occasions.

The operators in this category are:

ex & *ex*	AND
ex | *ex*	OR
ex ^ *ex*	exclusive-OR

~ *ex*	ones complement
ex << *ex*	left shift
ex >> *ex*	arithmetic right shift
ex >>> *ex*	logical right shift

All of the bitwise operators apply to integer operands. The difference in the >> and >>> operator is whether the left operand's sign is extended into the result. The >> operator is the arithmetic shift right operator and will cause sign extension to be performed on the result. The >>> operator does not perform sign extension.

This notation comes from the Java™ language, C++ does not have a >>> operator, but relies on whether the left operand is a signed or unsigned quantity to determine the shift type.

Although the normal use of bitwise operators is to use them for integers, the following also apply to other operand types:

Type Combination	Meaning	Result type
vector & *vector*	intersection of 2 sets. The elements that are common to both vectors	*vector*
vector | *vector*	union of 2 sets. The elements in one or both sets.	*vector*
string << *integer*	shift the contents of the string left by the integer number of characters	*string*
string >> *integer*	shift the contents of the string right by the integer number of characters	*string*
vector << *integer*	shift the contents of the vector left by the integer number of elements	*vector*
vector >> *integer*	shift the contents of the vector right by the integer number of elements	*vector*

Shifting a string or vector left causes it to shorten by the shift count. If you shift more than the size of the string or vector, it will become empty. The elements shifted off the left side are discarded.

Shifting a string (or vector) right causes the rightmost elements of the string (or vector) to be deleted. The length of the string (or vector) is reduced by the shift count.

The use of the & and | operators for vectors (along with the – operator) allow the programmer to perform set manipulation operations. They are quite compute-intensive operations so care should be taken where performance is an issue.

6.4. Comparison and relational operators

It is always useful to be able to compare one value to another. Aikido provides the usual set of comparison and relational operators:

ex == ex	Equality
ex != ex	Inequality
ex > ex	Greater than
ex < ex	Less than
ex >= ex	Greater than or equal to
ex <= ex	Less than or equal to
! ex	boolean not
ex ***instanceof*** ex	Hierarchy membership
ex ***in*** ex	Membership

The result of a comparison is an integer value of 0 (if the comparison fails) or 1 (if the comparison succeeds).

For comparisons between integers, reals, characters and strings, the following rules hold:

Type Combination	Meaning
integer op integer	integer comparison
integer op real	integer converted to real then compared
integer op string	integer converted to string then compared
real op real	real comparison
string op string	string comparison
string op integer	as *integer op string*
string op real	as *real op string*

String comparisons are done using the usual alphanumeric comparison of the characters in the string. For example:

```
"abc" == "abc"          // true
"abc" == "abcd"  // false
"abc" < "def"           // true
"abc" < "abcd"          // true
```

For other types the meaning of the operator varies with the type used.

Type Combination	Meaning
vector == vector	The vectors contain exactly the same elements
stream == stream	Streams are the same stream
map == map	Maps contain exactly the same elements
enumconst == enumconst	Same constant in same enumeration
block < block	Left block is subblock of right block (derived from it)
block == block	The same block?
block > block	Left block is superblock of right block (base block)
object == object	The objects are at the same address
vector < vector	The number of elements in the left is less than number of elements in the right operand
map < map	As vector < vector

The set of comparisons that deserve more explanation is the *block op block* ones. This is used when one block (function, class, monitor, thread, package, or enum [in this case]) is compared with another one. When comparing for less than or greater than, you are testing the inheritance relationship of the blocks. Less than means that a block is inferior in the inheritance hierarchy, while greater than means it is superior.

For the equality comparison, you are testing whether the blocks are exactly the same block.

The *instanceof* operator may be used to check if an object is a instance of a certain block set.

Given the following blocks:

```
class A {
}

class B extends A {
}

class C extends A {
}
```

The comparisons:

```
B < A          // true: B is derived from A
A > B          // true: A is the superblock of B
A == B         // false: they are not the same object
B == B         // true: they are the same object
B < C          // false: there is no inheritance relationship
```

6.4.1. Instanceof

The ***instanceof*** operator allows the programmer to test whether an object is an instance of a block. Because blocks are arranged in inheritance trees, being an instance of a sub-block implies that the object is also an instance of a superblock.

Given the following instances of the above classes:

```
var a = new A()
var b = new B()
var c = new C()

a instanceof A   // true
b instanceof A   // true: B is subclass of A
b instanceof C   // false: B is not related to A
```

Where interfaces are used, the instanceof operator may be used to check if an object implements the contract for an interface:

```
interface Dint {
   function f
}

class D implements Dint {
   public function f() {}
}

class E {
   public function f() {}              // same function but no interface
}


var d = new D()
var e = new E()
```

```
d instanceof Dint          // true: implements Dint
e instanceof Dint          // false: does not implement Dint
```

6.4.2. The in operator

It is frequently useful to be able to test if a value is a member of some other value. The language provides an operator named *in* for just this purpose. The *in* operator is an infix operator that may be applied to values of any type (including object instances). The result is either true or false, with true meaning that the left value is a member of the right value.

The language allows a special syntax known as a range for the in operator: The syntax is:

```
in-expression: shift-expression in range-expression

range-expression: shift-expression
                  shift-expression .. shift-expression
                  shift-expression ... shift-expression
```

This allows the right side of an *in* expression to be a range of expressions. For example:

```
1 in 1 .. 3                // true
6 in 7 ... 9               // false

enum Color {
   RED, GREEN, BLUE
}

GREEN in RED .. BLUE       // true
BLUE in RED .. GREED       // false
```

The values on either end of the range must be integral.

For builtin types, the following table shows the result of performing the expression:

```
value in <Right Value>
```

Right value	Operation
vector	search vector for value
map	search map for value
string	search string for substring or character
object	if object has an operator in, call it, otherwise look for string as member of object
block	look for string value as member of block

All other types result in a runtime error. Where the type of the right side of the in operator is a block the left side must be a string. Consider the following examples:

```
var a = 1
class B {
   public function f() {}
```

```
}
var b = new B()
var m = {1 = 2, 3 =4}

a in [1,2,3]          // vector: a has value 1 therefore true
a in [2,3]            // vector: false
a in m                // map: true since 1 is a key in the map
2 in m                // map: false since 2 is not a key in the map
'e' in "hello"        // string: true
"el" in "hello"       // string: true
"ol" in "hello"       // string false
"f" in b              // object: true
"f" in B              // class: true
"g" in B              // class: false
1 in B                // runtime error
```

6.5. Assignment operators

The assignment operators allow a value to be assigned to a variable. The set of assignment operators is the same as in C++. There is the straight assignment (=); and there are a set of compound assignments that perform a calculation on the right operand before doing that assignment.

The complete set of assignment operators are:

=	Straight assignment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
&=	AND assignment
|=	OR assignment
^=	exclusive-OR assignment
<<=	left shift assignment
>>=	arithmetic right shift assignment
>>>=	logical right shift assignment

A compound assignment is the logical equivalent of a straight assignment and an operator. For example, for the += compound assignment:

a += b is the same as *a = a + b*

The rules for type conversion for the individual operators also apply to the assignment operators.

The language allows multiple assignments to be done in one expression:

```
a = b = c = 2
```

The left side of an assignment operator has to have an address. That is, it must be a variable or an expression that has an address (like a vector subscript or block member access). Consider the following:

```
var a = 1
var v = [1,2,3,4]

class C {
```

```
    public var x = 1
}
var c = new C()
var d = null

a = 10          // ok, a is a variable
v = []          // ok, vectors can be assigned
v[2] = 10       // ok, element of vector
v[5] = 1        // error, subscript out of range
c.x = 30        // ok, access to member
(a + 1) = 2     // error, expression has no address
2 = 30          // error, number does not have an address
d.x =1          // error: null pointer
c.y = 10        // error: no such block member C.y
```

6.6. Conditional operator

The conditional operator allows the value of an expression to be tested and results in one of 2 possible values. It is a 3-operand expression and is of the form:

```
ex1 ? ex2 : ex3
```

The expression *ex1* is evaluated and if it is non-zero (compared with integer 0), the result of the conditional expression is *ex2*, otherwise it results in *ex3*.

6.7. Stream operator

The stream operator provides a method for input and output in a Aikido program. The operator takes 2 operands and is the form:

```
stream1 -> stream2
```

The contents of *stream1* are copied to *stream2*. The operands can be of any type and the interpreter has code to deal with all the built-in types. For example:

```
var x = 0                           // integer variable
var s = ""                          // string variable
var v = []                          // vector variable

stdin -> stdout                     // copy standard input to standard output
 "hello world" -> stdout            // writes "hello world" to stdout
56 -> outstream                     // the integer 56
["the result is: ", result] -> output // vector literal.  Each element written in sequence
x -> output                         // write variable x

stdin -> x                          // read from stdin to x
input -> s                          // read string from input and write to s

x -> v                              // append variable x to vector v
```

The rules for the input and output of the various built-in types are:

Output type	Input type	Operation

integer	integer	copy left to right
	real	real converted to integer
	string	string converted to integer if possible, 0 otherwise
	vector	first element converted to integer
	map	first element converted to integer
	char	converted to integer
	block	integer set to address
	enumconst	index into enumeration
	object	call toInteger() if present, otherwise address of object
	stream	one integer read from stream
real	integer	converted to real
	real	copied
	string	string converted to real if possible, 0.0 otherwise
	vector	first element converted to real
	map	first element converted to real
	char	converted to integer then real
	block	converted to integer then real
	enumconst	converted to integer then real
	object	toReal() called if present, error otherwise
	stream	one floating point number read from stream
string	integer	converted to string
	real	converted to string
	string	copied
	vector	each element appended to string
	map	each element appended to string
	char	converted to string
	block	name of block
	enumconst	name of constant
	object	toString() called if present, blockname@address if not
	stream	one line read from stream
char	integer	truncated to 8 bits
	real	runtime error
	string	first character in string
	vector	first element converted to char
	map	first element converted to char
	block	first character of name
	enumconst	‘A’ = first const, ‘B’ = second, etc
	object	toChar() called if present, error if not
	stream	one char read from stream
vector	object	toVector() called if present, otherwise object appended to vector
	anything	appended to vector
map	object	toMap() called if present, otherwise appended
	anything	appended as {x = x}
function	scalar	function called with single argument
	vector	function called once for each element.

		Element passed as parameter
	map	function called for each element. Function has to arguments for left and right of map pair.
	stream	function called for each line of input
thread		like function
class		like function only new object created for each
package		like class
enum		runtime error
enumconst		runtime error
object		runtime error
stream	integer	converted to decimal character sequence
	real	converted to floating point character sequence
	string	each character written
	char	single character written
	vector	each element written
	map	each element written as left=right
	block	block name written
	enumconst	name of constant written
	object	"object " + address written
	stream	steam copied

In some respects, the stream operator acts like the ***cast*** operator for the arithmetic types.

The result of a stream operator is the result returned by its right operator. If the right operator is a block type (function, etc) or an overloaded stream operator then the result is the value returned by that block. All the results of the block call are appended to a vector. This allows stream operators to be linked together. Consider:

```
var lines = []                          // vector variable
lines = instream -> func()
```

This reads all the lines from the stream and applies the function *func*() to them one at a time. The vector *lines* will contain the result of the function *func*() for each line.

Another example:

```
var lines = []
instream -> func1() -> func2() -> lines
```

sets the vector *lines* to the results of applying *func1*() and *func2*() to each line of the input.

6.8. Increment and decrement operators

There are 4 operators in this category. Two of them are prefix operators and the other 2 are postfix operators. The operators are:

++ ex	Pre-increment
ex ++	Post-increment

-- ex	Pre-decrement
ex --	Post-decrement

The prefix operators increment or decrement their operand and the result is the value after the increment or decrement is done. The postfix operators increment or decrement, but the value is as it was before the operation was done. Consider the following examples:

```
var a = 1
var b = 20

var c = ++a          // value 2
var d = b++          // value 20 (b set to 21)
var e = a++          // value 2 (a now has value 3)
var f = ++b          // value 21 (b set to 22)
```

6.9. Logical operators

The logical operators allow short-circuit evaluation of a logical expression. There are 2 logical operators: && (logical AND) and || (logical OR).

Each of them takes 2 operands and executes from left to right. Once the value of the logical expression can be determined the evaluation stops (this is known as short-circuit evaluation).

In the case of the && operator, the value of the expression is known to be false whenever the left operand is false. If this is the case, the evaluation stops and the right operand is never evaluated. If the left operand evaluates to true, then the right operand is evaluated.

For the || operator, the value of the expression is known to be true when the left operand is true. If it is false, then the right operand is evaluated.

The logical operators are guaranteed to execute left to right and terminate whenever the value is known.

6.10. Call operator

The call operator is the way to invoke a block. What the invocation does depends on the block type. The syntax for the operator is postfix parentheses enclosing an optional actual parameter list:

```
expression ( optional expression list )
```

The optional expression list is a list of expressions separated by commas.

The action taken by the call operator is:

Block type	Action
function	call the function and wait for it to complete
thread	start the thread and continue
class	create an instance of the class
monitor	create an instance of the monitor

Likewise, the value of the call operator depends on the block type:

Block type	Value
function	value returned from function

thread	stream through which messages can be sent to the thread
class	object created
monitor	object created

The value is returned from a function using the *return* statement. If no value is returned from a function the value *none* is returned.

For a thread, the interpreter creates a stream through which you can send messages to the thread. The thread can read the messages from its *input* stream. For more details, see Chapter 1.

For classes and monitors, the object created by the call is returned. This is the same functionality as the *new* operator. See section 6.11.

If the block being called is inside an object, call mechanism assigns the ***this*** variable to the object on which the call is being made. Consider the following call examples:

```
class A (x) {
   public function f (a,b,c) {
   }
}

package P {
   public function g {
   }
}

var a = A(1)                    // instance of A
var v = a.f (1,2,3)             // call of A.f
var p = P.g()                   // call to package function g
```

6.10.1. Value and reference parameters

When a call is made, the values of the actual parameters are passed to the formal parameters. What is passed depends on the type of the actual parameters. Section 5.3.7 specified that scalar values are passed by value, and compound values by reference. The following table specifies how the various types are passed:

Type	Mode
integer	value
char	value
real	value
string	reference
vector	reference
map	reference
function	value
stream	value
class	value
thread	value
object	reference
package	value
enum	value
enumconst	value

monitor	value

So, if you pass an integer to a function then a copy of that integer value is made before the call is made. If you pass a vector then a copy of the vector is not made. This means that you can modify the contents of the vector in the function. If you want to prevent the function from modifying the contents of the vector, make a copy of it before the call. You can do this with the System.clone() function.

Consider the following examples:

```
function a (x : integer) {
   x = 2
}

function b (var x: int) {
   x = 2
}

function c (v : vector) {
   c[4] = 45
}

var q = 1
var vec = [1,2,3,4]

a (q)                                  // q not modified
b(q)                                   // q set to value 2
c(vec)                                 // vec[4] set to 45
c(System.clone (vec, false))           // vec unmodified
```

Vectors, maps, strings and objects are all passed by reference in this way.

6.11. *new* operator

The *new* operator allows the creation of instances of blocks. It can be used to create a single instance or a vector of instances. The syntax of the new operator is:

```
new [ size ]...                        // new empty vector
new type (opt expr list)               // new object
new type                               // new object with no arguments
new type (opt expr list) [ size]...    // new vector of objects with arguments
new type [ size ] ...                  // new vector of object with no argments
```

The simplest case of the *new* operator is that where a single object is created. The optional expression list (in parentheses) is passed as the actual arguments to the constructor for the object. This behaves exactly the same as a call operator applied to a class or monitor.

6.11.1. Creating vectors

The *new* operator can create multi-dimensional vectors. In the simplest case, the *new* operator creates a single dimensional vector all of the elements of which has the value *none*.

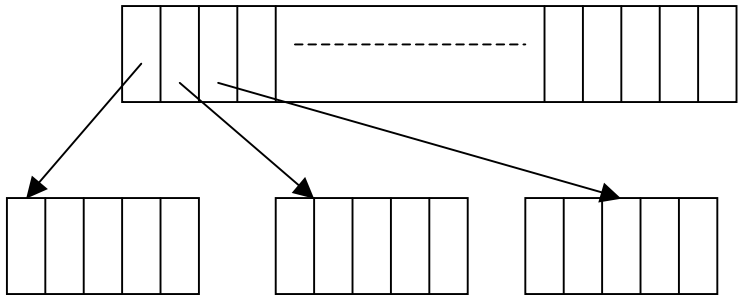
```
var vec = new [100]                    // create a 100 element vector
```

This returns a vector containing 100 elements.

A multi-dimensional vector (matrix) can be created by simply appending more dimensions to the operator:

```
var matrix = new [100][5]
```

This creates a matrix of 100 elements by 5 elements. Pictorially this is the layout.



This shows that the first dimension of the matrix is set up to point to 100 copies of the second dimension. If the matrix had even more dimension, then the second dimension vectors would be set up to point to the 3rd dimension, and so on.

In the form shown, each of the elements of the final dimension will have the value *none*. The new operator can be used to populate a vector with instances of objects or other values. The following example shows this:

```
class A (a) {
}

var avec = new A(2) [10]                    // vector of 10 instances of A
```

This creates a vector of 10 instance of the class A, each of which is constructed with the value of the parameter 'a' set to the number 2. If the class A had no parameters, you could omit the parentheses.

You can also use a simple type to create a vector of values.

```
var ivec = new int [50]                     // vector of 50 integers
```

There is no limit to the number of dimensions you can give a vector (or matrix). More than 4, however, is a little hard to understand (for my simple mind anyway).

6.11.1.1. Byte vectors

A regular vector is a sequence of values. Each element of the vector can take any type. This is not always exactly what is required. When you are writing programs that deal with network byte streams or raw memory it is more useful to be able to deal with the sequence of bytes directly without having to convert the bytes into a vector of values. This saves time and programming effort.

Aikido has a special type called a ‘bytevector’. This is (as the name suggests) a sequence of bytes. Like a vector it is variable in length and all the usual vector operations can be used on it.

A bytevector can be created using the *new* operator:

```
var buf = new byte [100]                // bytevector of 100 bytes
```

This creates a bytevector of 100 bytes in length. The individual bytes in the bytevector can be read and written like a normal vector:

```
0xff -> buf                 // append 0xff to buf
buf[5] = 0x75               // write to bfuf
var b = buf[5:9]            // slice of bytes
buf -> outfile              // write raw data to file
```

6.12. Subscript operator

The subscript operator can be used to extract and set the value of a single element, or range of elements of a value. It can be applied to integers, strings, vectors and maps (and objects when it is overloaded by the object).

There are 2 forms:

```
expr [ subscript ]                  // single subscript
expr [ sub1 : sub2 ]        // range subscript
```

For a range, the subscripts (*sub1* and *sub*) are inclusive. They may appear in any order in the brackets.

The subscript operator may be used both on the left and right side of an assignment. If used on the left, the subscript selects the element(s) to be set by the assignment.

Subscripts are checked at runtime and result in a runtime error if they are out of range.

6.12.1. Subscripting integers

An integer value can be subscripted. This has the effect of isolating the bits of the integer. The subscripts must be greater or equal to 0 and less than or equal to 63. That is:

```
0 <= subscript <= 63
```

Bit 0 is the least significant bit (rightmost as written down). Bit 63 is the MSB.

The range form of the subscript operator may be used to select a contiguous range of bits in an integer. Consider:

```
var x = 0x010203ff
var b0 = x[0]               // LSB of word (value 1)
var b = x[7:0]              // least significant byte of word (0xff)

x[23:16] = 0xee             // x now 0x01ee03ff
```

The use of integer subscripts is more efficient than the usual method of shifting masking and ORing values into a word.

6.12.2. Subscripting vectors, bytevectors and maps

Subscripting a vector allow the selection of elements within the vector. The subscript must lie in the range:

0 <= subscript < sizeof (vector)

Vector subscripts start at 0 and continue up to sizeof (vector) – 1. So, for example, a vector of 10 elements will have valid subscripts 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Consider:

```
var primes = [1, 2, 3, 5, 7, 11]          // a few prime numbers
primes[3]                                 // value 5
primes[-3]                                // error: subscript out of range
primes[6]                                 // error: only 6 elements
```

You can use a subscript to set the value of a vector element:

```
primes[2] = 20                            // primes now [1,2,20,5,7,11] (!)
primes[7] = 17                            // error: subscript out of range
```

Subscripting a vector using a range subscript produces a vector as a result. Subscripting with a single subscript can produce a value of any type. So, for example:

```
var x = primes[3]                         // value 5
var y = primes[3:3]                       // value [5] – a vector with only one element
```

Subscripting a map is a little different from vectors. For a start, the range type of subscript is not allowed. The use of a subscript for a map is to either add a new element to the map or to search the map for an element. If the subscript is on the left side of an assignment, then an element is added to the map, otherwise the map is searched. Consider:

```
var towns = {"San Ramon" = 15000, "Danville" = 12000}   // map of town/population
var pop = towns["Danville"]                             // value 12000
var pop2 = towns["Walnut Creek"]                        // value none
towns["Alamo"] = 9000                                   // insert "Alamo" = 9000
```

If a map is searched for a value and that value does not exist, then a value of type *none* is returned. This is the usual way to search a map:

```
if (typeof (towns["Danville"]) == "none") {
   // no map entry for Danville
}
```

Another way to insert a value into a map is by using the += operator:

```
towns += {"Dublin" = 20000, "Pleasanton" = 18000}
```

This allows multiple values to be inserted in one instruction.

If the value already exists in the map, it is overwritten with the new value.

6.12.3. Subscripting strings

Subscripting a string can take 2 forms. In the first case, you can subscript a string with an integer subscript (or range) and select a character (or substring) from the string. In the second you can search the string for a *regular expression*.

Consider the following:

```
var str = "now is the time"                 // simple string
var c = str[7]                              // character at [7] = 't'
str[0] = 'N'                                // string is now "Now is the time"
str[11:14] = "tune"                         // string is now "Now is the tune"

var ex1 = str["is"]                         // search for simple regular expression
var ex2 = str[".*is the ([a-z]+)"]          // complex regular expression
```

For the case of simple, integer subscripts the rules are very similar to vectors. The subscript must be in the interval:

0 <= subscript < sizeof (string)

The first character in the string is at subscript 0, the last at subscript sizeof (string) – 1. You can extract and overwrite the characters in the string.

Subscripting a string with a range subscript will produce a string (even if the subscripts are identical). Subscripting with a single index will produce a character. For example:

```
var x = str[7]                              // value 't' – single character
var y = str[7:7]                            // value "t" – string of one character
```

The System.find() (section 15.3.2) function may be used to search a string for a substring and return the index of the start of it.

Regular expressions are more complex and powerful…

6.12.3.1. Subscripting strings with regular expressions

Regular expressions provide a mechanism to extract specific substrings from a string. They are very powerful and complex. I will give an overview of the most common forms of regular expressions here. The regular expression support in Aikido is provided by a library implementing Perl 5 compatibility. This means that the regular expressions in Aikido are compatible (as far as possible) with the Perl language version 5.

For full details on the regular expression support please refer to Chapter 19 on page 19-285.

A regular expression is a string that contains special characters that can match other characters in the string being searched. There is a long list of special characters available.

If a character in a regular expression is not a special character then it matches the same character in the string being searched. So, for the simplest regular expression, consider:

```
var ex = str["is"]
```

This searches for the simple regular expression containing 2 characters ('i' and 's'). The special characters let you be ambiguous about what you would like to find. For example, the special character '.' (period) matches any character in the search string.

If you want to switch off the meaning of a special character, prefix it with a backslash character. Actually, because the regular expression is string literal in Aikido, you will have to escape the backslash with another backslash to allow it to be passed through as a backslash !!

The asterisk character means "match zero or more of the preceding character", so this allows you to find a series of characters all the same. That is not much use on it's own, but combine it with the fact that an asterisk can follow a special character (like '.') and you get a way to find a series of any characters.

Like the asterisk, the + character matches "one or more of the preceding character".

Another useful special character in a regular expression is the square bracket. A set of square brackets enclose a set of characters (or special characters) and will match any of that set. For example, consider:

```
var s = "abc1234abc"
s ["[0-9]+"]                    // matches 1234
```

This says: match a series of one or more characters in the range 0 to 9. In other words, extract a number from a string.

Regular expressions can get very unwieldy very quickly. The use of special characters and backslashes can turn what started out as a reasonably simple string into something resembling line noise. And then it starts to look like a Perl program.

Here is a list of the common regular expression "special characters"

Special Character (expression)	Meaning
.	any character
*	zero or more of preceding expression
+	one or more of preceding expression
[ ]	any of the enclosing expressions
[^]	anything except the enclosing expressions
( )	subexpression

When Aikido matches a regular expression it returns information about what it found. It returns this information in a vector of Regex objects. The Regex object is defined as:

```
class Regex {
   public var start = 0
   public var end = 0
}
```

The *start* and *end* members contain integer subscripts into the string being searched. They are inclusive. There is one element in the vector of Regex objects for each subexpression matched in the regular expression. A subexpression is an expression within the regular expression enclosed in parentheses.

The first element of the vector of Regex objects contains the subscripts for the whole regular expression.

If the regular expression is not matched in the string, an empty vector is returned.

For example, consider a system where a string is received consisting of 2 substrings separated by colons. We want to extract the 2 substrings:

```
function extractStrings  (command) {
   var expr = command ["([^:]+):([^:]+)"]                    // see explanation
   if (sizeof (expr) == 3) {                                 // need 3 expressions
      var sub1 = command[expr[1].start : expr[1].end]        // extract first
      var sub2 = command[expr[2].start : expr[2].end]        // extract second
      // process sub1 and sub2
   } else {
      // error bad format
   }
}
```

The regular expression looks complex, but is not really. It contains 2 subexpressions separated by a colon. Each subexpression is the same and says: "match one or more sequences of characters that doesn't contain a colon". The subexpression is enclosed in parentheses. Let's look in detail at the composition of the subexpression:

The first part is "[^:]". This matches any character except the colon character. This is followed by the + character, which says "one or more".

So, put the 2 subexpressions together separated by a colon and you get the whole regular expression.

The next step is to check if we actually got all the expressions we asked for. If all the matches succeed, the vector returned from the match should contain 3 elements. The first is the match for the whole expression; the second for subexpression 1 and the third for subexpression 2. Each element contains a RegEx object whose *start* and *end* fields have been filled in. The start field contains the subscipt of the start of the expression, the *end* contains the subscript of the last character in the string that matches the expression. So to extract the matched string, use a range subscript, passing the *start* and *end* fields of the appropriate vector element.

6.13. Member access operator

Members of a block may be accessed by use of the '.' operator. The member access operator is evaluated at runtime rather than at parse time. See section 5.3.17 for details.

The member access operator may appear on the left or right of an assignment expression. Consider the following examples:

```
class A {
   public var a = 0
   public var b = 0
}

var a = new A()                  // instance of class A
var x = a.a                      // reference to A.a
a.b = 12                                 // assignment to A.b
```

Access to the member is governed by the access mode of the member. See section 5.3.17.2 for full details.

6.13.1. Access to overloaded operators

If a class defines an overloaded operator function you can access it directly as a member. The syntax of this is:

```
object . operator op
```

The *op* is a valid overloadable operator (section 5.6.1). For example, consider the following:

```
class A {
   public operator + (x) {
      // add this to x
   }
}

var a = new A()                    // instance of A
var b = a.operator+ (4)            // direct call to overloaded operator +
```

6.14. *sizeof* and *typeof* operators

The *sizeof* operator is unlike the *sizeof* operator in C and C++. In Aikido, the *sizeof* operator is a dynamic operator, capable of calculating the size of anything at runtime. It can be used to get the number of elements in a vector or string. The following table shows the values returned by the *sizeof* operator for the various built-in types:

Type	Value of sizeof
integer	8
char	1
string	number of characters in string
vector	number of elements in vector
bytevector	number of bytes in vector
map	number of entries in map
object	number of variables in object
block	number of variables in block
enum	number of constants in enum
enumconst	1
real	8
memory	number of bytes in memory
pointer	number of bytes from pointer to end of memory
anything else	0

The syntax for the sizeof operator is:

```
sizeof ex
```

The value is the size of the expression *ex* (as from above table). Although parentheses around the expression are unnecessary it is probably good style to add them. Consider the following examples:

```
sizeof ("hello world")          // value: 11

var v = [1,2,3,4,5]
sizeof (v)                      // value: 5

if (sizeof (args) < 3) {
   // error
}
```

The *typeof* operator is very similar to the *sizeof* operator. The purpose is to obtain the dynamic type of an expression. The syntax for the typeof operator is:

```
typeof ex
```

Again, like the *sizeof* operator, parentheses are not necessary but style dictates that they should be added around the expression. The type of the value of the *typeof* operator depends on the type of the expression. The following table shows the typeof result for the built-in types:

Expression type	Value of typeof	Type of value of typeof
integer	"integer"	string
real	"real"	string
char	"char"	string
string	"string"	string
vector	"vector"	string
bytevector	"bytevector"	string
map	"map"	string
stream	"stream"	string
memory	"memory"	string
pointer	"pointer"	string
object (with value null)	"null"	string
object (with non null value)	block type of object	block type of object
block	block itself	block itself
none	"none"	string
enumconst	enumeration containing enumconst	enumeration containing enumconst
anything else	"unknown"	string

The type of the result of *typeof* may or may not be important depending on how the operator is used. If the result of a *typeof* is compared directly to the result expected then the type matters. Consider the following:

```
var t = typeof (e)
if (t == "integer") {
   // process integer
} elif (t == T) {
   // process block T
}
```

For this to work, you need to know that for an integer, the string "integer" is returned.

Another way of doing this is to use a comparison with another *typeof* operation:

```
if (typeof (e) == typeof (int)) {
   // process integer
} elif (typeof (e) == typeof (T)) {
   // process block T
}
```

If this is done, the actual value of the *typeof* operator is irrelevant, as long as it is the same every time for the same type.

6.15. *cast* operator

Given that Aikido is dynamically typed language there is rarely a need to cast one type to another. The main use of casts in C++ is to convert an object of one type to an object of another. Since the access to members in Aikido is done at runtime, there is never a need to cast the object (except for readibility reasons)

However, there is a need so be able to convert a value of one type to a value of another. Casting is a runtime operation (unlike C++ where it is done at compile time). The cast operator is, for example, able to convert a number to a string, or a string to a number.

The syntax for the cast operator follows the new C++ cast syntax:

cast < expression1 > (expression2)

The expression1 is an expression whose type is used rather than its value. This is very similar to the use of the expressions for types in block parameters (section 5.3.3).

The following table shows the operation performed when an expression of one type is converted to another type.

Cast from	Cast to	Operation
integer	real	integer converted to real
real	integer	real converted to integer
integer	string	integer formatted to decimal string
real	string	real formatted to string
integer	char	truncated to 8 bits
char	integer	extended to 64 bits
char	real	extended to 64 bits and converted to real
char	string	single character string
bytevector	string	characters in vector
object	integer	function toInteger() called if present
object	real	function toReal() called if present
object	char	function toChar() called if present
object	vector	function toVector() called if present
object	map	function toMap() called if present
object	string	function toString() called if present
string	integer	string parsed to integer if possible
string	real	string parsed to real if possible
string	char	first character extracted from string
string	bytevector	bytes in string
enumconst	integer	value of constant
enumconst	string	name of enum constant as string

For conversion from an object to a type, the interpreter can make calls to member functions of the object to do the conversion. The function called depends on the type to which the object is being converted. The above table shows the names of the functions called. Consider the following example:

```
class A {
   public function toString() {
      // return a string
   }
}

var a = new A()                    // instance of class A
var s = cast<string>(a)            // cast to string – call of toString()
```

Note: if the operator cast is overloaded in the object it will be called in preference to the above conversion operators.

6.16. Builtin member functions

The '.' operator is normally applied to instances of objects or members of packages. The builtin types, such as vectors, strings, maps, etc. are not objects and therefore the usual use of the '.' operator does not apply.

However, it is convenient to be able to use the same, object oriented, syntax to apply operations to the builtin types as you would with objects. The Aikido language provides a small set of operations that can be applied to the builtin types using the normal member selection operator. This is similar to the way in which the Java™ language provides the length attribute for an array when an array is not an instance of a standard object.

The set of operations that can be applied to the builtin types are:

Operator	Meaning
size()	sizeof()
type()	typeof()
print()	System.print()
println()	System.println()
clear()	Assignment of empty value
append(x)	Append to end of value
close()	Close a stream
eof()	End-of-file of stream
flush()	Flush the stream
rewind()	Rewind a stream
seek(off, whence)	Move to position in stream
clone (deep)	Copy object
fill (v, s, e)	Fill an object
resize (s)	Change the size of an object
sort()	Sort a vector
bsearch (v)	Search a vector for a value
find (val, index)	Find a value in a value
rfind (val, index)	Find a value in a value, searching backwards
split (sep)	Split a value at a separator

`transform (func)`	`Apply function to value`
`trim()`	`Trim white space`
`replace (find, rep, all)`	`Replace parts of a value`
`hash()`	`Generate hash code`
`insert (val, index)`	`Insert into a value`

Consider the following example:

```
var v = [1,2,3,4]
for (var i = 0 ; i <v.size() ; i++) {
   v.println ()
}
```

This is equivalent to:

```
var v = [1,2,3,4]
for (var i = 0 ; i <sizeof(v) ; i++) {
    println (v)
}
```

It's a matter of taste as to which you use. The former is more object oriented and might match other code in the program. The latter is more C-like. The latter will execute faster because there are fewer function calls made.

Chapter 7. Statements

A statement is the actual code that is executed by the interpreter. An expression (Chapter 1) is an example of a statement. Statements allow control of the execution of a program.

Here is a summary of the statements in Aikido:

declaration
expression
macro
{ *statement-list*$_{opt}$ }

if (*expression) statement1*
if (*expression) statement1* **else** *statement2*
if (*expression1) statement1* **elif** (*expression2) statement ....*
if (*expression1) statement1* **elif** (*expression2) statement2 ...* **else** *statementn*
switch (*expression) switch-block*

import *identifier-list*
import *string-literal*

using *package-expression*

while (*expression) statement*
for (*initialization*$_{opt}$; *expression*$_{opt}$; *expression*$_{opt}$) *statement*
do *statement* **while** *expression*
foreach *variable* **in**$_{opt}$ *expression statement*
foreach *variable* **in**$_{opt}$ *range-expression statement*

return *expression*$_{opt}$
break
continue

try *statement* **catch** (*variable*) *statement*
throw *expression*

delete *expression*

synchronized *(expression) statement*

Notable exceptions to other languages include the lack of a **goto** statement. Additions include the **elif** clause of the **if** statement and the **foreach** loop.

A macro is an expansion of a statement-level macro defined using the macro facility of the language. For further information see Chapter 1.

7.1. Declarations and expressions as statements

As can be seen a declaration and an expression are themselves statements and can therefore be used anywhere a statement can be used.

It is considered good practice to limit the scope of a variable to as small an area as possible.

7.2. Compound statements

A compound statement is a set of statements (possibly empty) enclosed in a pair of braces. The separator for the statements in the compound statement is either a line-feed character or a semicolon character. This is an unusual aspect of the Aikido language. In regular languages the separator (or terminator) for a statement is the semicolon character only and this must appear even if it is the last thing on a line.

When the programmer is writing code, she normally puts one statement on a line. This is considered good programming style. So, by the appearance of a line-feed character at certain positions in the source code, the parser can infer that the programmer meant to end the statement there.

This is not always the case, of course. The programmer might want to split a long line up across multiple lines, or put more than one statement on a single line. The Aikido parser allows this using the following rules:

1. Statements always end at their *natural end*, regardless of the presence or absence of a line-feed character.
2. A semicolon character will terminate the statement before its *natural end* and allow more on the same line
3. A line-feed character will be ignored if it appears within a statement before the *natural end*.
4. If the natural end of a statement is ambiguous, a line-feed character terminates the statement.

We need to define what the *natural end* of a statement means. At any point in the parse of a statement there is a set of valid tokens that can be next in the sequence. If the next token in the sequence is not among the valid tokens then the statement has reached its *natural end*. Most of the time the natural end of a statement is obvious. Sometimes, however, there are subtle syntactic ambiguities in the language that make is non-obvious. This occurs when an expression operator can appear as both a prefix and postfix (or infix) operator. Consider the simple case of the increment operator. This can appear both as postfix (after the object to be incremented) and prefix.

```
a = b++                 // postfix use of ++
++b                     // prefix use of ++
```

Suppose the parser has reached the identifier ‘b’ in the first line above. The next token in the input stream is the ++ operator. This can be read as either a postfix use of the operator for ‘b’ or as a prefix use for the next token. Thus there is no natural end for the statement at this point. It is ambiguous. If the programmer wrote:

```
x  = a
++b
```

He clearly means something different from:

```
x = a++
b
```

In this case, the line feed character is a significant token in the input. Of course, if the programmer is aware of the ambiguity he can use a semicolon to terminate the statement where he wants, but most people don’t want to have to think carefully about the end of every statement.

The ++ operator was one (sort of obvious) place where this occurs. There are others that are more subtle. Here is a list of them:

1. The + and – operators. Can be unary operators.
2. The ++ and -- operators. Can be used prefix too
3. The subscript operator [ ]. Can also be a vector literal
4. The call operator ( ). Can also be a parenthesized expression.

If the programmer really wants to split an expression over multiple lines, there are good and bad places to do so. Aside from the style issue, the rules are that if an expression is split over multiple lines, the split should occur where there cannot be a natural end for the statement. This only affects the operators listed above and should not happen in reality. Consider the following cases:

```
x = a_very_long_identifier_that_makes_the_line_too_long
   + b

a_function_call_with_long_argument_list
   ( a1, a2, a3, a4, s5 /* .... */)

x = a_long_array_name
   [ s ]
```

Each of these will be parsed as a different meaning than was desired by the programmer. To make them correct, reformat them as:

```
x = a_very_long_identifier_that_makes_the_line_too_long +
    b

a_function_call_with_long_argument_list (
   a1, a2, a3, a4, s5 /* .... */)

x = a_long_array_name [
   s ]
```

That is, make the split at the place in the statement where it has not reached its natural end.

Of course, if you wish, you can always insert semicolon characters at the end of a statement. I, personally, find that leaving out the semicolon characters is better because there is less typing and I rarely split expressions over lines.

One disadvantage of omitting the semicolons is if you ever want to convert a Aikido program to another language that needs the semicolons, you will have to manually insert them. Personally, I am willing to take that risk as it is only a matter of a couple of minutes with an editor when the port is attempted.

The following are examples of valid Aikido statement sequences:

```
if (a == 1) f() ; break

var x = 10 var y = 20

if (x ==
   10)
{ /* do something }

x = 20 y = 30
```

7.3. Selection statements

The selection statements allow a value to be tested and an action taken based on that value. The statements here are the ***if*** and ***switch*** statements.

if (*expression) statement1*
if (*expression) statement1* **else** *statement2*
if (*expression1) statement1* **elif** (*expression2) statement ....*
if *expression1) statement1* **elif** (*expression2) statement2 ...* **else** *statementn*
switch (*expression) switch-block*

7.3.1. The if statement

The ***if*** statement tests the *expression* and executes *statement1* if the value is non-zero. If the optional ***else*** clause is present and the value is 0 then *statement2* is executed. For example:

```
if (nickname == "root") {
   // perform root task
} else {
   throw "Access denied"
}
```

The optional ***elif*** clauses are a shorthand for:

if (*expression1) statement1* **else if** (*expression2) statement2 ...*

This is provided because the use of nested ***if*** statements is a common form found in programs. There may be many ***elif*** clauses in a single ***if*** statement. They may be followed by an ***else*** clause.

7.3.2. The switch statement

The ***switch*** statement is used to test a value and execute a set of statements depending on the value. The *switch-block* contains a set of statements labeled with either ***case*** or ***default***.

An example of a switch statement is:

```
switch (command) {
case LOGIN:
   // do login command
   break
case LOGOUT:
   // do logout command
   break
default:
   throw "unknown command"
}
```

This shows a switch statement that tests the value of the expression command. There are 2 ***case*** clauses inside the switch block and one ***default***. A ***case*** clause is of the form:

case *expression* : *statement-list*$_{opt}$

(Note that the *expression* is not a constant expression as would be mandated by other languages. See section 7.3.2.1 for more information).

The colon character is necessary (to keep syntactic compatibility with other languages). The colon is followed by a list of statements that are terminated by the next ***case*** clause, ***default*** clause or the end of the ***switch*** statement.

The ***default*** clause is similar to the ***case*** clause:

default : *statement-list*$_{opt}$

There is no expression in the ***default*** clause and there may be only one of them in the whole ***switch*** statement.

The operation of this ***switch*** statement is the same as:

```
if (command == LOGIN) {
   // do login command
} elif ( command == LOGOUT) {
   // do logout command
} else {
   throw "unknown command"
}
```

The ***switch*** statement compares the value under test to each of the ***case*** clauses in sequence until it finds a match. If a match is found then control is passed to the statement sequence of the *case* clause. If a match was not found then control is passed to the ***default*** clause if present. If there was no ***default*** clause no action is taken and control passes to the next statement after the ***switch*** statement.

Notice that the ***switch*** statement has ***break*** statements terminating the ***case*** clauses. If these were absent, control would flow from one ***case*** clause to the next one if that ***case*** clause was activated. For example:

```
switch (x) {
case 1:
   System.println ("one")

case 2:
   System.println ("two")

case 3:
   System.println ("three")
}
```

If this was called with the value of x == 1 it would output:

```
one
two
three
```

This may not be the expected result. The action of the ***break*** statements is to break out of the switch statement at that point. Usually the absence of a ***break*** statement for a ***case*** clause is a programming error, but occasionally they it is useful to be able to fall through to the next ***case*** clause. For example, consider the following ***switch*** statement:

```
switch (a) {
case 1000:
   // do something
   break
```

```
case 2000:
case 2001:
   // do something else
   break
}
```

The absence of a ***break*** for the 'case 2000' ***case*** clause is legitimate.

If it is not obvious that a fall through to the next ***case*** clause (or default) is desired, a well placed comment would be helpful to anyone reading the code.

7.3.2.1. Differences from C++ and Java™ switch statements

There are minor differences between the Aikido switch statements and those in C++ or the Java™ language (or C for that matter). The biggest difference is that the expressions in the case clauses are not constant expressions. Aikido allows any expression here and will perform a linear search through them comparing the value under test to each of them.

This means that the following is legal code:

```
switch (command) {
case "LOGIN":
   // do login
   break
case "LOGOUT":
   // do logout
   break
}
```

You can even call functions in the case clauses if you so desire.

This difference has a subtle performance issue. Because you can use any expression in the switch statement case clauses, there is no way for the parser to compile this to a branch table or any other such high-performance data structure. This makes the switch statement in Aikido have the same performance characteristics as an ***if .. elif .. else*** statement.

Another difference between Aikido and other languages is that there is no check done to make sure that two ***case*** clauses do not have the same value. Because they are expressions rather than constant expressions there is no way to do this without suffering a large performance penalty at runtime.

Another difference is that each case clause defines a new scope level. This means you can declare variables in a case clause. For example:

```
switch (x) {
case 1:
   var name = "xx"
   break
case 2:
   var name = "yy"
   break
}
```

Each variable 'name' will be in a different scope.

Finally, the last difference is that the ***default*** clause can contain an empty statement list. In C++ and the Java™ language it has to contain at least one statement.

These differences are due to the way in which ***switch*** statements are implemented in Aikido. In C++ and the Java™ language they are implemented as a branch table or search, with the ***case*** and ***default*** clauses being labels addressing blocks of code. Aikido does it differently. It can do so because it is an interpreted language rather than being compiled.

7.4. Import statement

The ***import*** statement causes the parser to insert a file at the top level scope of the program. This file contains other Aikido program text that is parsed in the context of the current program. This facility allows the program to be split into sections and also allows for user-defined and system provided libraries of classes and other facilities.

The syntax of the import statement is:

import *import-identifier-sequence*

or:

import *string-literal*

The import identifier sequence is a series of identifiers separated by periods. The sequence of identifiers is used to find the file to be imported by a system defined search. Once the file is found, it is parsed in the context of the top level scope of the program (scope main). The ***import*** statement will ensure that the same file is not imported twice. For example:

```
package T {
   import string          // import the string package

  var s = new String()    // create instance of object in string package
}
```

The sequence of identifiers may include wildcard characters. This allows a single import statement to be used to import many files. For example:

```
import chat.*
```

Will import all files that can be located in the ‘chat’ directory. The meaning of wildcard characters can be obtained by reading the UNIX® manual page for ‘glob’ (man –s3c glob).

The other form of the import statement uses a string literal as the name of a file to import. This is operating system dependent. When a string literal is used, it is taken literally as the name of a file to import. If this file doesn’t exist then the normal rules for finding a file are used. For example:

```
import “/project/Aikidofiles/lex.aikido”        // import a full file name
import “libfind.so.1.3”                         // for a specific shared library
```

Again, wildcards may be used to import many files at once.

It should be noted that, while it is referred to as a ‘*statement’* is it not truly a statement per se. The difference between an import statement and other proper statements is that you cannot enclose an *import* inside another statement and expect it to be controlled by it. For example:

```
if (System.operatingsystem == "Windows") {      // check for OS
   import windows                               // WILL NOT DO AS EXPECTED
} else {
   import goodos
}
```

This will not behave as you would expect if *import* was a proper statement. The above code will result in both the files being imported at the top level of the program and the 'if' statement being empty. Perhaps not what was intended…

7.5. Using statement

As touched upon on section 4.2, the names available inside a block may be augmented by the names in package by use of the *using* statement. The *using* statement adds the named package to the list of places to search when looking for the name of a variable, block or any other names in the program.

Normally Aikido searches for a name using the regular scope rules: inner scope first, then progressively outwards until the it reaches the outside. By inserting a using statement you are inserting side branches off the main scope search to the named packages. For example:

```
function A {
   using System
   function B {
     println ("hello")
   }
}
```

When inside function B, the identifier *println* is used as a call. The first place searched in function B itself. If it is not found here, the search moves on to function A. The println function is not defined in A, but there is a using statement for the System package, so that is searched next. The println function is indeed inside the System package, so the search ends there.

The appearance of a *using* statement inside a block affects the block itself and any nested blocks, but does not affect any other blocks.

The *using* statement affects the search from its point of issue to the end of the block in which it appears. Consider the following:

```
function A {
   function B {
      println ("hello")
   }

   using System

   function C {
      println ("hello")
   }
}
```

The call to println in function B will result in an error since the using statement has not yet been executed in the sequence. The one in function C will, however, be fine.

The use of a using statement can introduce ambiguity into the block. Generally there is no problem as the names defined locally take precedence over the names in any package you are using. But, if 2 packages share the same name and both are the subject of a using statement in the same block the language has no way of determining which one to use, so an error results.

Consider the following:

```
package A {
   public function print (x) {
   }
}

package B {
   public function print (x) {
   }
}

function C {
   using A
   using B

   print (1)                    // error: ambiguous use of print
}
```

Here, the function C has no way to determine which print function to call as there is one in both packages A and B.

Note that there is an implicit using statement of the System package done automatically by the language parser. This means that you can use any functions or other things defined in the System package without using the name of the package (provided they don't clash with user defined things of course).

7.6. Iteration statements

> **iterate** *('I-t&-"rAt)*: (*transitive verb*) to say or do again [C16: from Latin *iterare*, from *iterum* again]

Iteration statements allow the program to repeat a section of code while a condition is true. The following are the iteration statements in Aikido.

while (*expression) statement*
for ($initialization_{opt}$; $expression_{opt}$; $expression_{opt}$) *statement*
do *statement* **while** *(expression)*
foreach *variable* $\mathbf{in}_{opt}$ *expression statement*
foreach *variable* $\mathbf{in}_{opt}$ *range-expression statement*

7.6.1. The while, do and for statements

The ***while*** statement evaluates the *expression* and while it has a non-zero value executes the *statement*.

The **do** statement executes the statement while the expression is true. Unlike the **while** statement, the statement is always executed at least once.

The ***for*** statement is intended to be used for fairly regular loops where the initialization, termination and increment conditions can be expressed at the start of the statement.

The ***for*** statement takes 3 optional expressions and executes them as if the loop was written as follows:

for (*init* ; *condition* ; *increment*) *statement*

```
init
while (condition ) {
   statement
   increment
}
```

The *init* part of the ***for*** statement can be either an expression or a variable declaration. The variable declaration must start with the ***var*** keyword. This can be used to limit the scope of the control variable for the loop to the loop itself. Consider:

```
for (var x = 0 ; x < 10 ; x++) {
   // loop body
}
```

The variable *x* is limited in scope to the ***for*** loop itself.

All three of the expressions in the ***for*** loop are optional. If you omit them all you will get a loop that will never terminate (unless by some other means).

```
for (;;) {          // forever
   // loop body
}
```

A ***break*** statement will exit the for loop. A ***continue*** statement will cause a jump to the *increment* expression and the loop may continue.

7.6.2. The foreach statement

The ***foreach*** loop is a powerful statement that is used to iterate through all the members of a multi-valued set. The syntax of the ***foreach*** loop is as follows:

foreach *variable* ***in***$_{opt}$ *expression statement*
foreach *variable* ***in***$_{opt}$ *range-expression statement*

The *variable* is the name of a variable that will be declared by the ***foreach*** statement in the scope of the loop only. The *expression* is used to determine how many iterations will be made and what the value of the *variable* will be on each iteration. The *statement* is executed on each iteration.

Consider the following obvious ***foreach*** loop:

```
var primes  = [1,2,3,5,7,11,13,17,19]
foreach prime in primes {
   System.println (prime)
}
```

This iterates through all the members of the vector *primes*, setting the variable *prime* to each successive element. The statement prints the value of the *prime* variable.

Note: the 'in' keyword is entirely optional. It is allowed to be compatible syntactically with other languages that provide *foreach*.

One important note: the expression is only executed once for the lifetime of the loop. The interpreter evaluates the expression at the start of the loop and uses the value obtained to control the variable.

Another feature of foreach is the ability to iterate over a range of values. The 'expression' in the foreach statement may also be a range-expression. Consider:

```
foreach x 100..400 {
   f(x)
}
```

This iterates from 100 to 400 (inclusive) calling the function f() for each iteration. The restriction of this type of statement is that the range limits must be integral (enumerations are integral).

The foreach statement is used very frequently in a Aikido program. There are many occasions where you need to iterate through all the elements. The value of the expression in the foreach statement can be one of the following:

Expression	Action
positive integer	iterate from 0 up to the integer value in steps of 1
negative integer	iterate from 0 down to the integer value in steps of –1
vector	iterate through all the elements of the vector
string	iterate through each of the characters in the string
map	iterate through each of the elements in the map
enumeration	iterate through each of the constants in the enumeration
range	iterate over each element in range (inclusive)

It is common in a program to execute a loop a fixed number of times. You can do this with a ***for*** loop as follows:

```
for (var x = 0 ;x < 10 ; x++) {
   // loop body
}
```

But a much more convenient way to do it is with a ***foreach*** loop:

```
foreach x 10 {
   // loop body
}
```

or, if the start is not 0:

```
foreach x 1 .. 9 {
   // loop body
}
```

Iterating through the elements for a vector or string is a pretty obvious operation: the variable is set to each successive element on each iteration. Iterating through a map is a little more difficult to visualize.

A map is implemented as an associative array of value versus value. Each 'element' in a map consists of a pair of values. It you iterate through a map, what is the *variable* set to on each iteration?

The answer is that it is set to an instance of an object of the following type:

```
class Pair {
   public generic first
   public generic second
}
```

An element in a map can be thought of as a mapping from *first* to *second*. Consider the following:

```
var towns = {"San Ramon" = 15000, "Danville" = 12000, "Dublin" = 20000}

foreach town towns {
   var name = town.first
   var population = town.second
}
```

The map of Bay Area towns versus population is traversed one element at a time with the variable *town* being set the each successive element. The variable has its *first* field set to the town name and its *second* field set to the population of the town. There is no guarantee of the order in which the towns are returned – it depends on the nature of the map.

Note: there is no way to change the value of the loop variable from within the loop. If this is needed, you must use a different loop type (***for*** or ***while***). The ***break*** and ***continue*** statements operate as expected within a ***foreach*** loop.

7.6.3. Break and continue statements

When in a loop it is sometimes necessary to terminate the whole loop early or to terminate a particular iteration. Aikido provides the usual ***break*** and ***continue*** statements to enable this.

The ***break*** statement causes the innermost loop (or ***switch*** statement) to terminate immediately, with control passing out of the loop to the next statement after the loop. It is important to note that only one loop is terminated. If you are inside many nested loops only the innermost one is affected.

Aikido does not provide a **goto** statement. This much-vilified statement is present in C++ but not in the Java™ language. One of the legitimate uses of the goto statement is to get out of nested loops. This is not available in Aikido and you must use other means to do so. You could use a variable to detect a loop terminating condition, or perhaps use an exception (be careful of performance). It is true to say that wherever a goto statement occurs there is always another way of doing the same operation.

The ***continue*** statement causes the innermost loop to terminate its current iteration and continue on to the next one. In the case of a ***for*** loop, the next iteration includes executing the *increment* expression.

7.7. Return statement

The ***return*** statement causes a function to terminate and, possibly, return a value to the caller. The syntax of the return statement is:

return $expression_{opt}$

Used without the optional expression, the return statement causes the function to terminate immediately. If the caller was expecting a value to be returned from the function the she is out of luck.

Used with a value allows the function to return a value to the caller. There is no restriction on the types of the values returned from a function (functions are generic). Of course, if you return a type that was not expected by the caller you can expect to get a runtime error.

If you want to omit the expression from the return statement you must terminate the statement with either a semicolon or line-feed character (comments are, of course, ignored). This probably won't be a problem for anyone, but it is best to mention it for the sake of completeness.

7.8. Exception statements

Aikido supports a C++ style exception mechanism (see Chapter 1). The supporting statements are the ***try*** and ***throw*** statements.

The try statement sets up an exception handler for a block of code. The syntax of a try statement is:

try *statement* **catch** (*identifier*) **statement**

The statement is executed and if any exception occurs while the execution is taking place, the catch clause is invoked. Invoking the catch clause involves declaring a variable with the name specified and setting its value to the exception that has occurred, and then executing the statement in the catch clause.

Unlike C++, there can only be one catch clause in the try statement. It is always executed when an exception is thrown. The job of the catch clause is to determine if it should handle the exception or not. If it decides not to handle it (say the type isn't what it is expecting), it should re-throw the exception to the next handler.

To raise an exception, use the ***throw*** statement. This is of the form:

throw *expression*

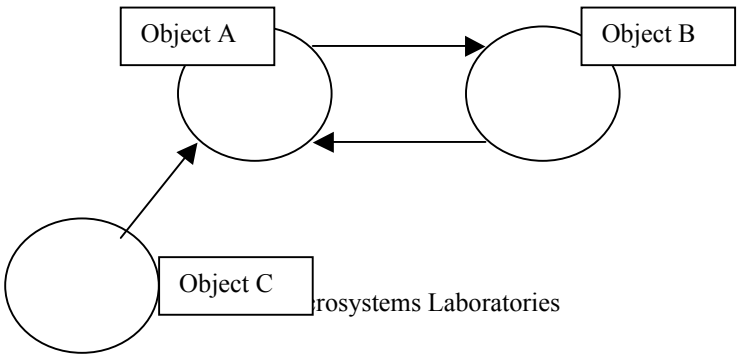
The expression can be of any type (including a block). The exception is thrown to be caught by the first handler on the way up the call stack. If there are no handlers in place the interpreter terminates and prints a runtime error to standard error.

There is one call stack per thread, so if a thread does not catch an exception, the whole program will terminate.

7.9. Delete statement

The ***delete*** statement removes a reference to a variable or part of a variable. Aikido is a garbage collected language. The type of garbage collection used is referred to as 'reference counting'. When the interpreter notices that the reference count to an object gets to zero (there are no more references to it), the object is deleted. For further information see Chapter 1.

The problem with reference counting garbage collection is that is doesn't work where you have 2 objects that refer to each other. Consider the following:



Objects A and B are mutually referential (they have pointers to each other), and object C points to object A. Suppose the reference count for object C goes to zero and it is therefore deleted. The link from C to A is removed but A is not deleted because it has still got a reference from B.

The solution to the above dilemma is to explicitly delete object A by artificially decrementing its reference count. This will bring it to 0 and therefore it will be removed. Once A is out of the way, B will have its reference count set to 0 and it will also be deleted.

The ***delete*** operator does just this. It artificially decreases the reference count of an object by 1.

The ***delete*** operator can be applied to regular objects or to an element of a vector or map. Consider the following:

```
class A {
   public var f = null
}

var a = new A()
var v = [1,2,3]
var m = {1 = 'a', 2 = 'b', 3 = 'c'}

delete a.f                  // object member
delete a                    // delete the object a
delete v[1]                 // delete the second element of the vector
delete v                    // delete the whole vector
delete m[0]                 // delete the first element in the map
```

Deleting an element from a vector or map actually removes the element from the vector or map. Consider the following:

```
var v = [1, 2, 3, 4]        // 4 integer values
delete v[2]                 // delete the 3rd value – v now contains [1,2,4]
```

7.10. Synchronized statement

This statement allows a temporary monitor to be created for an object for the duration of the statement. It is sometimes useful to be able to lock an object for a short period without having to declare the object as a monitor. This might be for performance reasons. The synchronized statement allows just this:

```
synchronized (obj) {
   // perform some task on obj knowing that we have a lock
}
```

When the synchronized statement is executed a monitor is created for the object referenced in the expression (if it doesn't already exist of course). This monitor is entered and the statement is executed. When the statement finishes execution the monitor is released.

Chapter 8. Exceptions

An exception is an occurrence that is not considered to be part of the normal behavior of the program. They usually signal an error condition where part of the program has been asked to do something it was not designed, or not willing, to do. An exception *occurs* at a given point in a program and is *handled* in another. The two points may coincide.

There are multiple ways to make an exception occur:

- Print a message to the output of the program. This is usually to the standard error stream
- Return an unusual or out-of-range value from a function. The null pointer is usually considered out of range for an object. For an integer-valued function, the value –1 is usually returned
- Throw an exception using the built-in exception mechanism

There are also multiple ways to handle an exception:

- Abort the program altogether
- Set a flag and continue
- Totally ignore it
- Invoke an exception handler

As the science of software engineering has evolved over the past half century or so, most of these methods have been considered state-of-the-art at one time or other. In most cases, there was not other options available due to system or language constraints. For others they were bad choices for exceptions pure and simple.

The current state-of-the-art for the occurrence and handling of exceptions is the language-supplied exception system.

So, let's say that I design an object to represent a user of a system. Let's also say that one of the attributes of a user is a password which the user must supply in order to log on to the system. One of the jobs of the user object is to provide validation of the password. We choose to implement the user as a class:

```
class User (name, password) {
   public function validatePassword (pass) {
      if (password != pass) {
         // do something
      }
   }

   public function getName() {
      return name
   }
}
```

The class provides a member function *validatePassword*() that takes an argument (*pass*) and checks to see if it is the same as that expected for the user. The check is done as a simple comparison. The design issue is what to do if the password is wrong.

We consider this as an occurrence of an exception, so our choices boil down to the list presented above. When deciding which option to choose we have to consider how the exception is handled. Realistically we have 2 choices here. We can either return a boolean value (**false**, say) to tell the caller that the password provided was not correct; or we can throw an exception.

If we decide to return a false value to the caller we have defined that the API to this class is that the caller must check the return value of *validatePassword*() to see if the password validated OK, or not. This is a valid thing to do. We have just delegated the responsibility of the password check failure to one of our callers (of which there potentially may be many). The callers now have to decide what to do when our *validPassword*() function tells them that the password was wrong and are faced with the same question as the original function.

For example, they may do the following:

```
function login (user, password) {
   if (!user.validatePassword (password)) {
      System.println ("Invalid password")
      System.exit (1)
   }
// perform login functions
}
```

or, perhaps the following:

```
function login (user, password) {
   if (!user.validatePassword (password)) {
      return false
   }
   // perform login functions
   return true
 }
```

The buck must stop somewhere. The first example shows an exception handler. The exception occurrence was detected and something was done about it. The program would never get past the *login*() function if the password check failed.

The second example just defers the problem and passes it on to the caller of the *login*() function.

Suppose we chose the first method of handling the exception. What impact does our choice have on the rest of the program? At first sight it appears that it is minimal – the caller of the *login*() function doesn't ever need to deal with the case of the password being incorrect. However, if the *login*() function was called as part of a thread, we do not really want all the other threads in the system to stop, just because one failed. We need a way to stop a single thread without the other threads in the system being adversely affected.

Unfortunately the only way to stop a thread is to return from the thread function itself, so this means we have to pass up the error condition all the way to the top. This points us toward the second method of exception handling where we return an error status from the *login*() function. We are forced to handle the exception condition at the top level.

This doesn't seem very satisfactory. Fortunately, the language comes to the rescue.

8.1. Throwing and catching exceptions

Aikido has an exception handling mechanism built directly into the language. It is based on the C++ and Java™ methods and therefore should be a tried and tested mechanism.

The basic idea is that when an exception occurs, an exception object is *thrown* at the point of occurrence. The exception then passes up the call stack to the first function that is willing to deal with it. When an exception handler is found, it is invoked and passed the exception object. If a handler is invoked and it is not willing to deal with the exception, it can *re-throw* the exception to the next handler up the chain. If we get to the end of the chain and nobody was willing to handle it, then we have reached a point where the program has no option but to terminate.

So, we talk of *throwing* an exception to a handler who *catches* it – must like the exception was a ball in a game. To throw an exception we use the **throw** statement (section 7.8), and we catch it using the **catch** clause of a **try** statement (also section 7.8).

A **try** statement sets up the conditions for an exception handler. It tells the interpreter to *try* to execute the code within the statement, but that it is willing to deal with any exceptions that occur during the execution. The phrase "try to execute" is important because, by use of the try statement, you are saying: "I know that this code may fail and it's my responsibility to deal with the case where it does". When the code does indeed fail (by throwing an exception) the **catch** clause of the **try** statement is invoked and the exception that has occurred is passed to it.

Here is an example of a try statement:

```
try {
   login (user, password)
   System.println ("User " + user.getName() + "  logged in")
   service (user)
} catch (e) {
   log ("Login for " + user.getName() + " failed due to: " + e)
   return
}
```

The code inside the try statement calls the *login*() function passing the user and supplied password. If the *login*() function succeeds, the user is logged in and the function *service*() is called for the user, which presumably services requests by the user (accepts commands or something).

If the login fails (possibly due to password mismatch, but could be anything else), the exception handler is invoked. The exception handler is passed the exception that was thrown (the variable 'e'), and it logs a message saying why the user failed to log in and returns from the function containing this code.

There can only be one **catch** clause for a **try** statement, in contrast to the C++ and Java™ exception mechanisms. This is because of the dynamically typed nature of the language. The Aikido mechanism is no less powerful than that of C++ or Java™, it is just different. In C++, the exception mechanism is responsible the doing the type comparisons between the exception being thrown and the exception handlers. A handler is only ever invoked if the type of exception it is willing to deal with matches that of the exception (subject to certain rules).

The Aikido mechanism always invokes the exception handler, passing it the responsibility of doing the type comparison for itself. It could be argued that this is actually a more powerful mechanism than that available in C++ because the comparison of exceptions can be done by the code in the program itself rather than it being restricted to a fixed set of rules designed by the language designer.

For example, if we define a set of simple exceptions as follows:

```
const BAD_PASSWORD  = 1          // password mismatch
const TOO_MANY_USERS = 2         // too many users logged in
```

```
const USER_BARRED = 3                    // user has been barred from logging in today
```

We can write exception handlers like:

```
try {
   login()                                          // try to log in
  // etc
} catch (e) {
   if (typeof (e) == typeof (int) && e < 0) {              // test for type and value
       System.println ("server failure, have to exit")
      System.exit (e)
   } elif (typeof (e) == typeof (int)) {                   // got an expected exception?
      switch (e) {
      case BAD_PASSWORD:
         System.println ("Invalid password, try again")
         continue                                      // allow a retry
      case TOO_MANY_USERS:
         System.println ("Too many users, try again later")
          return
      case USER_BARRED:
         System.println ("You are barred, try again tomorrow")
         return
      }
   } elif (typeof (e) == "string") {                   // got a string?
      System.println ("Failed to login due to: " + e)
      return
   } else {
      throw e                                       // rethrow exception to caller
   }
}
```

So, given that we can catch exceptions with a handler, how can we signal the occurrence of an exception in the first place? The **throw** statement.

Aikido allows the program to throw any *value*. Recall that a value is the basic holder of information in a Aikido program (see Chapter 1) and can have one of a number of built-in types. When you want to raise an exception condition, you create a value and throw it.

Consider the following examples:

```
throw 1                          // an integer 1
throw "Invalid password"         // a string
throw  InvalidPassword()                 // an object (could also use new Inv...)
throw getException (1)           // something returned from a function
throw false
throw null
throw new Exception ("Invalid password")
throw InvalidPassword            // a class
```

When a handler comes across an exception is it not prepared to deal with it can throw the exception to the next handler in the chain. This is done by simply throwing the variable that was caught.

```
try {
   // code
} catch (e) {
   if (typeof (e) != "string") {                   // I only like strings
```

```
        throw e                                    // anything else gets thrown on
    } else {
        // deal with string exception
    }
}
```

Of course, you can always handle the exception and then throw it on again:

```
var s = System.openin (filename)
try {
    // code
} catch (e) {                                  // for any exception
    System.close (s)                           // close the stream
    throw e                                    // and rethrow it
}
```

8.2. Uncaught exceptions and stack unwinding

What happens if the exception reaches the top level in a call stack? To answer this we must define what a call stack means.

Each thread in a program maintains its own stack. At any point in the execution of a thread, the stack contains information about where the thread has been. The most basic use of the stack is to hold information such as the return address for the currently executing function so that when the function returns, the thread knows what to execute next. It also holds the values of the local variables and other information for each level in the stack.

The call stack has a top and a bottom. The bottom of the stack contains information about the first function called by the thread (the thread function itself). The topof the stack keeps changing, but contains information about the currently executing function.

Suppose we have a set of function like this:

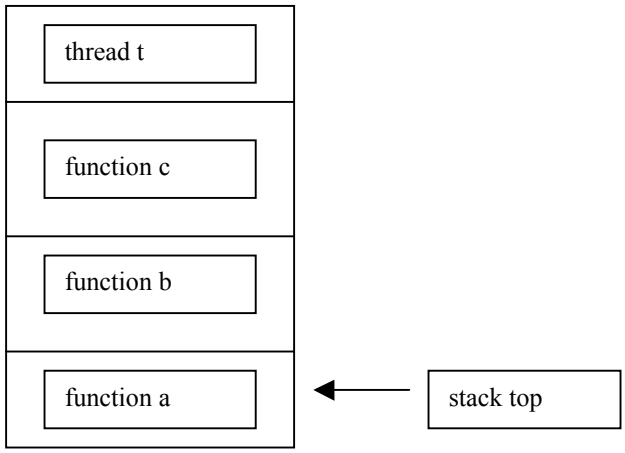
```
function a {
    var a = 1
}

function b {
    var b = 2
    a()
}

function c {
    var c = 3
    b()
}


thread t {
    var t = 4
    c()
}
```

If we are currently executing the function ‘a’, then the snapshot of the call stack at this moment in time would be:



If we return from function ‘a’, the topmost element of the stack would be popped off and we would be left with function ‘b’ at the top.

When you throw an exception the stack is rewound by traversing from the top to the bottom looking for an exception handler. Each time we pass from one level to another in the call stack we remove the level just passed, thus freeing all the memory associated with that level. It also destroys any variables that were created at that level.

When we come to an exception handler, the language interpreter arranges it so that the top of the stack contains the function in which the handler is situated and invokes the handler, first assigning the variable named in the handler with the exception being thrown.

If we get to the bottom of the stack and nobody had caught the exception, then the program must exit because we have arrived at a situation where an exception that is potentially dangerous has not been handled. This does not just terminate the current thread, but stops all threads and exits the whole program.

An uncaught exception will cause the interpreter to print a “runtime error” to the standard error stream before exiting.

So, consider the following (badly written) example:

```
thread userServer (stream) {
   var user = new User()                    // might throw an exception, but not handled
   var password = getPassword()
   try {
      login (user, password)
   } catch (e) {
      if (typeof (e) == “integer”) {
      switch (e) {
      // handle exceptions here
      }
   }                                        // oops, don’t handle anything else
}
```

If we get an exception thrown that is not an integer (say, a string), or thrown from somewhere outside the try statement we will see something like the following appear on the terminal:

Runtime Error : no such user fred

I leave it an exercise as to what was thrown.

8.3. Exceptions and runtime errors

When a Aikido program is running, there are occasions when the interpreter itself throws exceptions.. They can be caught like any other exception and handled before causing a Runtime Error and a program abort.

The types of the exceptions thrown by the interpreter may be a string or may be an instance of the class *System.Exception* (or one of its subclasses). The System.Exception class is described in section 15.3.11.

For example, the *openin*() function in the *System* package supplied with Aikido tries to open a file for input and returns a stream if successful. There is always the possibility that the file cannot be opened (maybe it doesn't exist, or is not accessible), so the *openin*() function throws an exception of type *System.FileException* if it cannot perform its task.

If you are prepared to deal with the inability of the openin() function to open the file, you can enclose the call in a try statement:

```
try {
    var instream = System.openin ("chatrc")
    processRCFile (instream)
} catch (e) {
    // can't open file, ignore
}
```

If it is a requirement of the program that the file exist, you can either let is cause a runtime error, or catch the exception and give a more appropriate error.

Chapter 9. Streams

Input and output are perhaps the most important aspects of an program. The incorrect handling of input can cause devastating program errors. We all know the old phrase Garbage In Garbage Out (GIGO). This succinctly sums up the problem. Even the most well written program cannot work properly if the input data is incorrect or is incorrectly handled.

Input and output is built into Aikido. This is a feature that was absent in the C family of languages and even Java™, both of which rely on system provided libraries and functions to perform required IO operations.

The Java™ language has the concept of a Stream. This is an object supplied by the *java.io* package that supplies methods for reading and writing. In Aikido, the language provides a full set of input and output operations on built-in values called *streams*.

A stream is an input/output channel usually connecting the program to a device. The simplest streams are those connected to files. Consider:

```
var instream = System.openin (filename)     // open the file for input
var lines = []                              // create an empty vector
instream -> lines                                  // read whole file into vector
System.close (instream)                     // close the stream
```

This code segment first creates a stream connected to a file. The file is opened for input (meaning that you can only read from it, and it must already exist). It then creates an empty vector and then reads all the lines from the stream into the vector. Finally it closes the stream.

The System library provides a set of functions to create and close streams. Once a stream is opened, it may be used as a normal value in the program. The special -> operator is the stream operator (section 6.7) and causes copies to and from streams. The function of the stream operator depends on the types of its operands. In the case above, the right side operand is a vector type. The definition of the operation of a stream write to a vector says that the whole stream is appended to the vector. Thus, this example reads each line in the file and appends that line to the vector until the end of the stream is detected.

9.1. Stream operations

In addition to creating, reading and writing streams, Aikido provides a set of operations that can be used to manipulate them. These operations are held in the System package.

The operations provided are:

Operation	Result
close (stream)	The stream is closed
select (stream, timeout)	Returns 1 if there is data waiting to be read from stream. Times out after *timeout* microseconds.
eof (stream)	Returns 1 if the stream is at the end of file
flush (stream)	Flush the data remaining in the stream buffers
getchar (stream)	Read a single character from the stream. Returns the character read
getbuffer (stream)	Read all the available characters in the stream buffer. Returns a string containing all the characters

availableChars (stream)	Returns the number of characters in the buffer
setStreamAttribute (stream, attr, value)	Set the value of a stream attribute
rewind (stream)	Rewind the stream to the start
seek (stream, offset, whence)	Move to a new position in a seekable stream

9.1.1. Stream buffering

Streams are not connected directly to their device. To do this would introduce performance penalties and erratic behavior due to the latencies of the hardware devices. Rather, the streams have a buffer embedded in them to insulate the user of the stream from the hardware. A buffer is an area of memory that is used to hold data that is either waiting to be read by the stream user, or is waiting to be written to the device.

Say we had a stream connected to a network connection. If we did not have buffering, any write to the stream would cause a packet to be transmitted over the network. This is not only undesirable in terms of performance, but also may not be what is expected by the receiver of the packets.

Buffering also helps on the incoming side by providing a FIFO for incoming data that has not yet been processed by the stream user.

The user of a stream must be aware of the buffering scheme being used by the stream. It may not be obvious, for example, that the data is not actually written to the stream until the output buffer is flushed, so the user may be happily sending data to it and the stream is simply buffering it up. This can occur especially with streams connected to networks, where the medium is best suited to large buffers.

The size of the buffer attached to a stream may be controlled by the *StreamAttributes.BUFFERSIZE* attribute. This may be set by the use of the *setStreamAttribute*() operation. For example, if we decide that we don't want a stream to be buffered, we could use the following on the open stream:

```
System.setStreamAttribute (s, System.StreamAttributes.BUFFERSIZE, 0)
```

Alternatively, we can choose to allocate a large buffer for, say, a network stream by:

```
System.setStreamAttribute (net, System.StreamAttributes.BUFFERSIZE, 8192)
```

The default size of the buffer is 512 bytes.

Buffers may be flushed to the hardware by use of the *flush*() operation. You can read the whole buffer into a string data type by calling the *getbuffer*() operation. You can look to see how many characters remain in the buffer by calling the *availableChars*() operation.

There is a system-supplied class called StreamBuffer() that provides many of the features you need to send data over streams. In particular, when you read from a stream into a StreamBuffer it will read all the characters available in the stream. And when you write a StreamBuffer to a stream it will automatically flush the buffer.

If the stream attribute AUTOFLUSH is set to true, the stream will be flushed when any data is sent to it. This should be used with care as performance may suffer. It is useful when the stream is being used for transmitting characters at a time.

9.2. Reading and writing streams

Streams can be read and written by use of the stream operator. The operator understands the type of data being read into or written from and behaves differently depending on the data type. The following table shows the combinations:

Data type	Reading	Writing
integer	Decimal integer converted to binary	Decimal integer
string	Whole line read – terminated by line feed character which is discarded	Characters written to stream.
real	ASCII for real value	Written as ASCII
char	Single character	Single character
vector	Each line of file appended to vector - line feed is retained	Each element written
bytevector	bytes read from input stream. All the bytes in the current buffer are read	All the bytes in the vector are written in raw mode
map	cannot read	Elements written as first=second
enumeration constant	cannot read	Name of enumeration constant

9.3. Standard streams

Like any programming language, Aikido provides a set of standard streams connected to the standard devices of the system. There is one connected to the standard output (*stdout*), on to the standard input (*stdin*) and one connected to the standard error device (*stderr*).

These are set up by the interpreter and are available to anything in the Aikido program. So, to write an error message to the standard error stream, you could write:

```
["Error: incorrect range: ", a, " to ", b, '\n'] -> stderr
```

This creates a vector literal and uses the stream operator to write it to the standard error. To read from the keyboard (usually connected to standard input, but may be redirected), you could write:

```
var limit = -1
stdin -> limit
```

In addition to the standard streams, each thread in a Aikido program has 2 streams connected to it. These are connected by the system and are called *input* and *output*. For the main program thread, the *input* is connected to *stdin* and the *output* is connected to *stdout*. The idea of these streams is to provide a stream that can be redirected without worrying about overwriting the standard stream variables and not being able to direct it back again.

Also, the input and output streams are automatically connected as communication channels to any thread created. Consider the following:

```
// server thread, sits waiting for input, processes input and writes result
thread server {
  while (!System.eof(input)) {                  // process until stream closed
    var command = ""
    input -> command                            // read command from stream
    var result = execute (command)              // execute command
    result -> output                            // write result to output
    System.flush (output)                       // flush the stream
  }

  var serverStream = server()           // create thread and stream

  var result = ""
```

```
"cat x.c\n" -> serverStream          // send command to server
System.flush (serverStream)          // flush the stream

serverStream -> result                        // wait for result
```

The above example shows a thread that acts as a server. It sits in a loop, reading commands from its input stream, executing them, and sending the result to the output stream. The main thread first creates the server thread and then sends a single command to it. It then waits for the result to come back.

As can be seen, when a thread is created, the return value from the thread creation call is a stream that is connected to the thread. The thread sees the other end of the stream as its *input* and *output* streams.

9.4. File streams

One of the most common uses of streams is for access to files. A stream can be attached to a file by opening the file using one of the file-opening functions provided in the System package. The file access functions are:

Function	Purpose
openin (filename)	Open the named file for input
openout (filename)	Open the named file for output
openup (filename)	Open the names file for update (reading and writing)
openfd (fd)	Open the integer file descriptor as a stream
open (filename, mode)	Open the named file with the given integer mode

When opening a file for input, the file must already exist. The file is opened and the current position is set to the start of the file. If the file doesn't exist, an exception is thrown.

Opening a file for output means that the file is created if it doesn't exist, or truncated if it does exist. In either case, the current position will be at the start of the file.

When opening a file for update, the file is created if it doesn't exist or truncated if it does. The stream is capable of reading and writing to the file. The current position is set to the end of the file.

The *openfd*() function connects a stream to an existing file descriptor provided by the operating system.

The *open*() function takes both the filename and an integral mode parameter. The mode parameter specifies a particular mode in which to open the file. The modes are enumerated in the System package and are:

Mode	Meaning
OpenMode.APPEND	Put file pointer at end of file
OpenMode.BINARY	Open the file in binary mode
OpenMode.IN	Open the file for input
OpenMode.OUT	Open the file for output
OpenMode.TRUNC	Truncate the file to zero length
OpenMode.ATEND	Open the file and place pointer at end
OpenMode.NOCREATE	If the file doesn't exist, don't create it

OpenMode.NOREPLACE	If opening for output and the file exists, don't replace it

The modes are defined as a set of bits that can be ORed together to form all the properties of the open File streams are seekable. This means that you can move the current position by use of the *rewind*() and *seek*() functions in the *System* package.

The *rewind*() function puts the file pointer at the start of the file. Any reads or writes after a rewind has been performed will happen at the start of the file.

The seek() function allows the file pointer to be moved to any position in the file. It moves the file pointer relative to the start, end or current position in the file. The 3rd parameter specifies from which position the move is relative:

Whence value	Meaning
SEEK_SET	Relative from the start of the file
SEEK_CUR	Relative from current position
SEEK_END	Relative from end of file

9.5. Network streams

Aikido has the ability to connect a stream to a network connection. A stream connected to a network port operates like any other stream except it is not seekable. The Network package contains functions to support network streams. In order to use the network package it is necessary to import it to your program. This is done by placing an import statement in your code:

```
import net
```

This will search for the file net.aikido in all the normal places. See Chapter 1 for full information on imports.

Function	Meaning
Network.open (addr,port)	Open an active network connection (TCP)
Network.openServer (addr, port, type)	Open a passive network connection (TCP or UDP
Network.lookupName (name)	Consult a naming service to convert a network name to an address
Network.lookupAddress (ipaddr)	Consult a naming service to convert an integer IP address into a host name
Network.accept()	Wait for and accept an incoming connection
Network.openSocket()	Open a UDP client socket for sending datagrams
Network.send (socket, addr, port, buffer)	Send a UDP datagram to the given socket. The addr and port specify the recipient. The buffer is the string or bytevector to send
Network.receive (socket, var addr, var port, maxbuffer = 4096)	Wait for an incoming UDP datagram. Blocks until a datagram is received, then returns the data as a bytevector. Also sets the addr and port to the sending address. The maxbuffer argument specifies the max size of data that can be received
Network.peek (socket, var addr, var port, maxbuffer = 4096)	As receive, but don't extract the data from the network. A call to receive() will read the

	data.
Network.formatIPAddress (addr)	Build a string of the form n.n.n.n out of an integer IP address.

Networks consist of a set of interconnected machines, each with a unique address. Each machine has a name that can be translated to an Internet Protocol (IP) address by looking it up in a naming service. One such common naming service is the Domain Naming Service (DNS).

A network name is a string like “agamemnon.eng.sun.com”. The DNS service converts this to a 4 byte integer. You may be familiar with the form of IP addresses. They are written down as a series of 4 numbers separated by dots. The numbers are in the range 0..255 (although certain numbers are reserved for special uses). For example, the name “agamemnon.eng.sun.com” may be translated into the IP address: 152.70.60.45.

The network stream creation functions take an address as the first parameter. An address can either be a string or an integer. If passed a string, the functions will either perform a lookup on the name, or if the string conforms to the standard IP address format (4 numbers separated by dots) will convert it to an integer. If passed an integer, the functions will use the bottom 32 bits of the integer as the IP address.

The port parameter of the functions specifies a TCP port to be used as the endpoint for the connection. It is an integer in the range 0..65535 (16 bits).

9.5.1. Passive and active connections

The terms “passive” and “active” come from the terminology used in the TCP/IP documentation. An active connection is a client side connection, and a passive connection is one for a server. Consider what constitutes a network connection. You have a machine out on the network that is sitting waiting for incoming connection requests. That machine has done what is called a “passive” open of a network port. Another machine wants to create a connection across the network to the first machine. It does this by opening a port in “active” mode. When a port is opened in active mode it will cause the underlying network software to send traffic across the network to the receiver (determined by the address/port/type triple).

So we have a server sitting waiting for incoming connections on a port it has opened passively and a client who is actively sending it traffic from a port it has opened actively.

A passive port can be opened in Aikido by issuing a call to the openserver():

```
var s = Network.openServer (address, port, Network.TCP)
```

The return value from the openServer() call is a “socket”. This is an integer that represents a passively open port. Just by doing this we have just told the network software in the operating system that this port should be opened. We cannot receive anything on it yet because there are no connections to the port.

As previously stated, a passively open port accepts incoming connections from other machines. In order to wait for an incoming connection we issue an accept() call. This blocks until we get a connection request, then creates a stream for the connection and returns the stream value to the caller.

```
var instream = Network.accept (s)                    // wait for incoming connection
```

When the accept() call returns we have an open stream to another machine on the network.

So, how do we do an active open to connect over the network from the client side? The call open() is used to do this:

```
var serverstream = Network.open (addr, port)
```

When this returns we have an open TCP stream to the server and can start to send messages through it.

9.5.2. Special considerations for network streams

When using network streams it is important to realize that the stream is not connected to a passive device line a disk drive, but to another program running on a different machine. When a stream is connected to a disk there is no harm in sending data through the stream at varying rates, even a character at a time. The only consideration is that of performance. An unbuffered stream attached to a disk will be slower than an buffered one, but it will still work.

When a stream is connected across the network, the packetization of the data is of utmost importance. The other end of the connection may not be a Aikido program, so it will be expecting packets containing certain data. If the Aikido program sending the data sends it in drips and drabs, the other end may see incomplete packets and will fail with protocol errors.

All data sent across a network is sent in a *packet*. This is a series of bytes of data enclosed in a protocol envelope. Usually the protocol is Transmission Control Protocol (TCP), but may be anything else. A packet's payload (the term given to the data held within the packet) can be of any length, but certain protocols impose the exact contents and length of the payload.

Consider the case where a server (written in C or some other non-Aikido language) gets a connection from a client. Say the server is something that a client logs in to. The server protocol for this application may require that the first packet seen from the client contain information such as the username and password for the person wishing to log in. Further, suppose that the contents of the first packet consists of 2 strings, each of which are terminated by a line-feed character. You could write the client in Aikido as follows:

```
// login to the server with username and password
function login (server, username, password) {
      function write (s) {                  // local function to write to stream
         [s, '\n'] -> server
         System.flush (server)
      }

   write (username)                         // write username
   write (password)                         // write password
}
```

The above program would cause a protocol error in the server because there will actually be 2 packets sent to the server. The appearance of the *flush*() call in the *write*() function will cause a packet to be sent or both the username and the password.

The program is very easy to fix – just move the call to the *flush*() function until all the data has been sent. In this case, it was trivial, but there are other subtleties that may come into play.

The default size of a buffer for a stream is 512 bytes. If the protocol for a server requires a packet that is larger than 512 bytes the packet will get split into 2 by the stream buffering mechanism. In this case you can use the setStreamAttibute() call to increase the local buffer size for the stream.

Another consideration is with the use of strings. When reading strings from a stream, the stream operator expects a line-feed character to be present at the end of the string. Sometimes this will not be present. If

so, you'll have to read a character at a time from the stream until you get to the terminating character of the string.

9.5.3. Datagrams

A network stream is a TCP connection. This is a connection-oriented protocol that uses the TCP protocol to send and receive data. Another protocol that may be used for sending data over a network is UDP. This is a connectionless protocol in which every packet sent over the network is addressed with the address of the intended recipient. Such a packet is known as a *datagram*.

Datagrams are sometimes used where a TCP connection is not suitable (broadcast messages for example) or the overhead of the TCP protocol is not wanted. For example, the common DHCP protocol for assignment of IP addresses on a LAN uses UDP datagrams.

A program wishing to be the recipient of a datagram can create a passive network socket using the Network.openServer() call and specifying Network,UDP as the '*type'* parameter:

```
var socket = Network.openServer (addr, port, Network.UDP)
```

Once a socket has been created, the server may receive a packet using the *Network.receive()* function. This waits for an incoming packet and when it gets one, it reads the sender's address and reads the data from the network. It returns the data is received as a string and sets two reference variables to the IP address and port number of the sender:

```
var addr = 0                                    // IP address of sender
var port = 0                                    // UDP port of sender
var data = Network.receive (socket, addr, port)     // receive a datagram
```

After a *receive()* call has completed, the '*addr'* and '*port'* variables are set to contain the IP address and UDP port number of the sender of the datagram. The data is returned from the function as a bytevector.

To send a datagram you can call the function *Network.send()*. This takes the socket to send on, the IP address and UDP port to send to, and the data to send:

```
Network.send ("computer2", 6456, "hello world")
```

The data must be a string or bytevector (or something that can be cast to a them).

The Network package also contains a stream filter (section 9.6) for sending repeated datagrams to the same address. This is called the *DatagramStream* class and may be used as follows:

```
var str = new Network.DatagramStream ("computer2", 6456)          // create stream

"hello world" -> str                                              // send string to stream
```

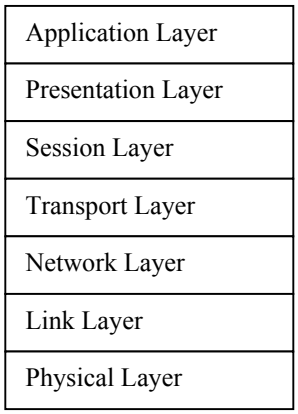
The *DatagramStream* may be *retargetted* by calling the '*retarget*()' function with the new address and port:

```
str.retarget ("computer3", 6456)
"hello again" -> str
```

The number of datagrams sent through the stream is available in the '*numDatagrams'* variable in the stream.

9.6. Layering streams: stream filters

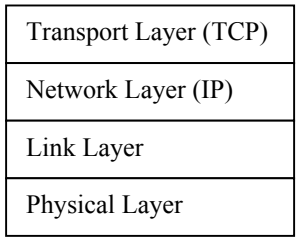
Communications protocols are comprised of a set of layers. These layers make up what is known as a *stack*. This is done to simplify the design of the protocols and to ease understanding. For example, the ISO Open Systems Interconnect (OSI) communications protocols standard is divided into at least 7 layers, each of which implements a part of the protocols. Each layer has an interface above and below and has been given a name. The standard OSI stack model is:



The idea of a layer is that its world consists of a set of 2 interfaces: one to the layer below it and one to the layer above it. The flow of data is through the layers. For outbound data, the flow is down the stack. Data enters at the top and is transformed by the layers until it reaches the bottom of the stack. For inbound data, the opposite it true.

It is also a property of the layering scheme that a higher layer provides higher level data structures than a lower layer. Take the Presentation layer in the OSI stack. The interface at the top of the layer is a set of abstract objects (an object model). The lower interface to the session layer is a flat byte stream. Therefore the presentation layer is responsible for converting to and from the abstract data types to the concrete data required by its lower interface.

Every protocol available on the network relies on some sort of stack of layers. The simplest would be the TCP or UDP protocols used throughout the Internet. These consist of 4 layers as follows:



The bottom 2 layers provide the very low level transmission and reception of data over a wire. The third layer (Internet Protocol) provides a way to address a packet of data and transmit it over the network to another IP layer on another machine. The top layer is the Transmission Control Protocol (TCP) layer and is responsible for building and maintaining connections to another TCP layer on another machine. It is also responsible for ensuring that data sent over the connections arrive in a timely manner and in the correct order.

One way to implement such a scheme would be for each layer to know exactly where it stands in the stack and be coded to only fit there. So each layer would provide an API (Application Program Interface) at its top and make calls to another API at its bottom. This scheme would certainly work but is a little inflexible. Consider a situation where an application program was sitting above an API and making calls to it. Now we wish to insert another layer in the stack. We cannot do this without making changes to the application itself.

A better way to implement the layering scheme is to use a stream mechanism. When AT&T developed the System V Release 4 (SVR4) version of UNIX® they introduced the concept of a stream access to devices. Traditionally, a device was accessed through the regular open, close, read, write and ioctl calls provided by the operating system. These were basically implemented as a set of function pointers in a table provided by he kernel. If you wanted to add to a device driver you had to either go in and modify the kernel code, or you could add an application program on top of the device driver and modify all your code to call it rather than going directly to the device driver.

The streams mechanism extends the traditional interface (open, close, read, write and ioctl) by allowing you to *push* a module on to a stream. This, in effect, inserted a piece of code between the application program making calls to the device driver and the device driver itself. The interface to the device driver remained the same, it was just that the calls made by the application program now went to the new module first, rather than going to the device driver directly.

The streams solution provided by AT&T was adopted as a good solution to the device driver access problems. You could now push a stream module on to a device to allow, for example, encryption of the data going to the device. The application program would not necessarily have to be aware of the new module in its path to the device.
Streams are a good generic mechanism for abstracting an interface. They are also applicable for higher level protocols as well as device drivers. By overloading the stream operator (->) in a Aikido program you are able to implements the same strategy for the streams used by Aikido programs.

Recall that a class can provide an operator function that will be called whenever the stream operator is applied to an instance of that class (see section 5.6.1.4). For example:

```
class A {
   public operator -> (data, isout) {
   }
}

var a = new A()                 // new instance of A
a -> outstream                  // write object to stream
instream -> a                   // read object from stream
```

The operator function will be called for all input and output operations on the instance ‘a’. It is passed 2 parameters: a data item, and a flag that has value 1 if the operator is being used for output to the object.

So, in the above example, the expression:

```
a -> outstream
```

Will call the operator function with *data* = *outstream* and *isout* = ***false*** (we are reading from the object). The expression:

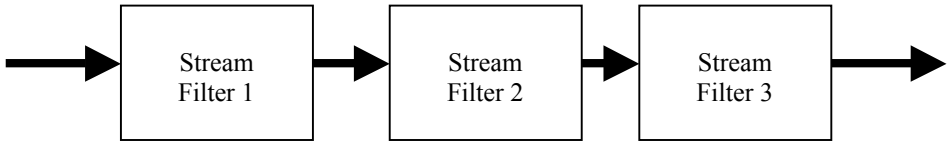
```
instream -> a
```

Will result in *data* = *instream* and *isout* = ***true*** as we are writing to the object.

An object providing an overload of the stream operator may by placed in the path of any stream. The user of the stream does not have to be aware of the existence of the object. Typically an object placed in the path of a stream will add something to any data passing through the stream on the way out and remove something on the way in.

An object placed in a stream for the purpose of modifying the data flowing through the stream is referred to as a **stream filter**.

For a stream filter, there are 2 directions in which data will flow. The *downstream* direction is where the data is sent to the filter for forwarded to the rest of the stream on the outbound side. The *upstream* direction is going the other direction. When you write to a stream the data flows downstream, for reading, the data flows upstream.



In the diagram, the flow from left to right is the downstream direction. This corresponds to the direction of the arrow suggested by the -> operator.

Consider a simple protocol filter that adds a length indicator to the data passing through it. Let's define is as prefixing the data with a marker (say the character 'A'), then the decimal length of the data, then the marker again (A). This is followed by the data itself. Let's define a class called *ProtocolA* that implements this layer:

```
import streambuffer                          // we use a streambuffer

class ProtocolA (instream, outstream) {      // takes input and output streams
   var buffer = new Streambuffer()           // make a new buffer

   public operator -> (data, isout) {        // stream operator overload
     if (isout) {                            // output to stream?
        buffer.put ('A')                     // add marker
        buffer.put (sizeof (data))           // add length
        buffer.put ('A')                     // add another marker
        buffer.put (data)                    // add the data
        buffer -> outstream                  // write to downstream
     } else {
        instream -> buffer                           // read from input stream
        if (buffer[0] != 'A') {              // check for initial marker
           throw "protocol error"
        }
        var ch = 1
        var len = 0
        while (buffer[ch] != 'A') {          // read the length (until marker)
           len = len * 10 + buffer[ch++] – '0'
        }
        ch++                                 // skip final marker
        buffer[ch:sizeof (buffer) – 2] -> data   // write data upstream
     }
```

```
    }
}
```

The StreamBuffer is used as a convenient way to gather the components required by the protocol into a single packet. It also allows the bytes received to the extracted.

As can be seen, the *ProtocolA* stream operator provides for both upstream and downstream flow. When used in the downstream direction it adds the markers and length to the data. When used in the upstream direction it checks for the markers, reads the length and sends the payload on its way.

The stream filtering mechanism allows any number of layers to be placed in the stream, each one of which is unaware of the existence of the others. Its primary use is in the implementation of network protocols, but may be used for any input and output to a stream. For example, it may be used to filter output to the screen to convert it to upper case:

```
class Shout (instream, outstream){
   import ctype                    // character typing

   public operator -> (data, isout) {
      if (isout) {
         var val = ""                        // result string
         foreach ch data {                   // for each char in input
            val += ctype.toupper (ch)        // convert to upper case
         }
         val -> outstream                    // write to out stream
      } else {
         instream -> data                    // transparent flow through
      }
   }
}
```

This filter only works for output. It just acts as a pass though when it is used as input.

Chapter 10. Multithreaded programming

Programming using multiple threads of control can be a daunting task. Conceptually it is very simple, but there are many subtle problems that can occur if you don't take care with your data. These are the sort of problems that do not show themselves until a rare combination of events forces them out.

Aikido supports multithreaded applications directly in the language. Much like its ancestors Ada and the Java™ language, Aikido has constructs and support structures to allow the user to write a program with multiple threads of control all running independently.

In a single threaded program there is one path the program can take. That path may pass through multiple functions and go back on itself many times, but there is no way to get off the path once you are on in. The introduction of threading support into operating systems allowed the program to take many paths at the same time. Sometimes those paths never meet and even go off into canyons never to come out again. Sometimes they cross and clash.

The basic support construct for a multithreaded program is the *thread*. A thread is basically a function that is called and executes in parallel with all other threads in the program. The program can spawn as many threads as it likes. It Aikido, a thread is defined in a very similar way to a function:

```
thread t (args) {
   // thread body
}
```

You start a thread by calling it. For example, the following code will start multiple threads:

```
foreach x 1000 {
   t()
}
```

Spawns 1000 copies of the thread 't'. Each copy of the thread executes effectively in parallel with all the other 999 copies of it.

Aikido uses a process or task model for threads. This is where you designate a function as a process and it always behaves that way. You can, of course, inherit a function from a thread, thus changing it from a process to a regular function. This is in contrast to the Java™ method which uses a Thread Object model. in this model a thread is a regular object with special functions (*start*() and *run*()). When the start function is called the Java™ interpreter spawns a thread and arranges it so that the first thing called when the thread starts is the *run*() function.

10.1. Threads

Everything in a Aikido program runs in a thread. The main program is actually a thread even though you don't explicitly name it as such. There are a set of operations provided by the system to give you control over threads. These are in the package 'main' so are accessible to everything.

The functions provided are:

Function	Parameters	Purpose
sleep	time in microseconds	Delay the current thread for a time
setPriority	integer priority	Set the priority level of the current thread
getPriority		Return the current priority level of the current thread
getID		Get the current integral thread id
join	stream connecting to thread	Wait for thread to terminate

Consider the following example:

```
thread test() {
   System.println ("thread " + getID() + " sleeping")
   sleep (10 * 1000000)                    // 10 seconds
   System.println ("woke up")
}

var t = test()
System.println ("waiting for test")
join (t)
System.println ("test completed")
```

This shows the use of the thread operations. The thread 'test' prints its identifier and sleeps for 10 seconds. It then wakes up and terminates. The main program spawns one copy of the thread and waits for it to finish.

Note that if you don't join a thread before terminating the main program, all the threads will be stopped.

The above example prints:

```
waiting for test
thread 4 sleeping
[10 seconds pass]
woke up
test completed
```

The order of the first 2 lines is non-deterministic as the thread may start before the print from the main program.

10.1.1. Thread priorities

All threads effectively execute in parallel. Each thread has a scheduling priority assigned to it that governs the amount of CPU time allocated to it relative to other threads. Aikido has 100 priority levels. Level 0 is the lowest priority and level 99 is the highest. A thread with a higher priority will get more CPU time than one at a lower priority.

The priority of a running thread can be set only by the thread itself and is an integer in the range 0..99. The function setPriority() sets the priority and the function getPriority() retrieves the current priority.

The following constants give the limits of the priority levels available:

Constant	Meaning
MIN_PRIORITY	The lowest priority level (0)
MAX_PRIORITY	The highest priority level (99)

The constants are in the System package.

10.1.2. Alternate threading model

The Aikido threads can be made to look like the Java™ threading model. As mentioned before, the Java™ language uses a Thread Object model where an object is defined with a couple of special functions. Let's see how to define such an object in Aikido.

```
class Thread {
   public function run() {                  // meant to be overridden by subclass
      throw "run called directly in thread"
   }

   private var id = 0                       // local copy of thread id

   public thread start() {                  // start the thread running
      id = getID()
      run()
   }

   public function setPriority (p) {
         main.setPriority (p)
   }

   public function getPriority() {
      return main.getPriority()
   }
}
```

Given this definition, we can now define subclasses of this Thread class to make our own threads. Consider:

```
class ServerThread (stream) extends Thread {
   public function run() {
      for (;;) {
          // run server functionality
      }
   }
}

// 2 instances of the server thread
var t1 = new ServerThread (stream1)
var t2 = new ServerThread (stream2)

// start the threads running
t1.start()
t2.start()
```

The Thread class is, of course, a simplified version of that available in the Java™ language. It does, however, serve to show that the Aikido thread model is very powerful.

10.2. Monitors

Threads frequently need to share data. When data is shared by multiple threads you always have to possibility that there will be a clash when 2 threads try to access the data at the same time. A monitor provides a mechanism that allows shared data to be protected by "Mutual Exclusion Lock" or *mutex*.

So why is it necessary to protect shared data? Consider the situation where we have 2 separate threads running simultaneously on 2 separate processors in a multiprocessor machine. The data to be shared between them is in memory attached to both processors. Say that the threads both wish to increment a counter in the shared memory.

In order to increment a counter the programs need to do the equivalent of the following code:

```
count = count + 1
```

This is decomposed into 3 steps:

1. Load the value of count from memory
2. Add 1 to the loaded value
3. Store the incremented value back to memory

Now consider the 2 threads happily running along. The current value of *count* is 1. Let's call the threads 'A' and 'B'. Consider one possible scenario where the threads are not trying to access the variable *count* at the same time:

Time	Thread A	Thread B
1	Loads count (1)	
2	Increments value to 2	
3	Stores count (2)	
4		Loads count (2)
5		Increments value o 3
6		Stores count (3)

So after the 2 threads have completed their increment, the value of *count* is (correctly) 3. Now consider the scenario where both threads arrive at the code to increment *count* at the same time:

Time	Thread A	Thread B
1	Loads value of count (1)	Loads value of count (1)
2	Increments value to 2	Increments value to 2
3	Stores value back to count (2)	Stores value back to count (2)

This time we get a problem. The value of *count* is incorrectly set to 2 after both threads have run. This is known as the "lost update" problem and is one of the classic reasons for failure in multithreaded programs.

You may think that the solution to the problem is to provide the variable *count* with a 'lock'. This is a variable that guards the access to *count*. Each thread could check the lock before trying to increment the *count* variable and loop until it the lock is freed. Consider the scenario where we put a lock on count and each thread checks the lock before accessing *count*. Each thread reaches the lock at the same time.

Time	Thread A	Thread B
1	Loads value of lock – gets 0	Loads value of lock – gets 0
2	checks if lock is 0 – true	Checks if lock is 0 – true
3	sets value of lock to 1	sets value of lock to 1
4	loads value of count (1)	loads value of count (1)
5	increments value to 2	increments value to 2
6	stores value of count (2)	stores value of count (2)
7	stores value 0 back to lock	stores value 0 back to lock

Thus we have not solved the problem, only moved the point of contention to the lock variable itself.

In order to solve this problem we need support from the hardware. We need the ability to perform an *atomic* 'lock' operation that is not divided into several stages. Most processors that can be used in multiprocessor systems have such an instruction. We need a way to access the instruction from Aikido (or any other language for that matter).

One solution to the problem is the *mutex* lock. This is a variable that is treated specially by the operating system and provides *lock* and *unlock* operations. These operations are atomic (probably coded in assembly language in the operating system to use the atomic instructions of the processor). The semantics of the lock operation are those we tried to do with the lock variable – if the *mutex* is locked we wait until it becomes unlocked. Consider the above scenario with the use of a mutex instead of a lock variable:

Time	Thread A	Thread B
1	Locks mutex – gets lock	Locks mutex – has to wait
2	loads value of count (1)	
3	increments value to 2	
4	stores value of count (2)	
5	unlocks mutex	Gets lock on mutex
6		loads value of count (2)
7		increments value (3)
8		stores value (3)
9		unlocks mutex

Thus, is we can guarantee that the *lock* operation on a *mutex* is atomic we have solved the lost update problem.

A *monitor* is a more functional solution to the problem. It provides an atomic locking mechanism like the *mutex* solution but also provides *wait* and *notify* operations. These operations can be used to implement resource allocation problems. A *mutex* itself cannot be used for resource allocation, you need another construct called a *condition variable*. Thus a monitor is a combination of a mutex and a condition variable.

A monitor can be thought of as an area of memory that can only be accessed by one thread at one time. All accesses to a monitor when it is occupied are blocked and the threads are placed on a queue until they can gain access.

Consider the code for the update of the *count* variable written in Aikido with a monitor.

```
monitor Count                       // monitor to protect count
   var count = 1                    // the counter needing protection

   public function inc() {          // function to increment count
      count++
   }
}

var count = new Count()             // instance of the monitor

thread A {                          // thread A
   for (;;) {
      count.inc()                   // increment count
   }
}
```

```
thread B {                          // thread B
   for (;;) {
      count.inc()                   // increment count
   }
}
```

Both threads are infinite loops, incrementing *count* for ever.

Notice that we used a function to increment the counter. If we had not done this, but provided access to the counter through the monitor:

```
thread A {
   for (;;) {
      count.count++
   }
}
```

Then we still have the problem. This is because the ++ operator is not atomic and breaks down to:

```
count.count = count.count + 1
```

The fact that the monitor guarantees exclusive access to the *count* variable will not help because you are making 2 separate accesses to it – one to read it and one to write it. Each one will be locked, of course.

10.2.1. Wait and notify for monitors

Monitors can be used to implement a queuing scheme for resource allocation. Resource allocation is where you have a resource or some sort (say, a disk drive or a network stream) and you have multiple users wishing to access the resource at the same time. If there were more resources available than users there would be no problem – just allocate each user its own resource. This is rarely the case.

Consider the case of an audio device on a computer. There is only one audio device and only one set of speakers. If we have a multithreaded application where each thread wishes to send something to the speakers we need some way to guard the access so that only one gets control at one time. If this was not done, the best scenario would be that you would get garbled sound from the speakers as the output from each thread is interleaved with the output from other threads. The worse case would be a crash due to contention on the device.

To support resource allocation the following functions are made available to monitors:

- ***wait***()
  When inside a monitor, release the monitor and put the thread on the list of those waiting for the monitor. When woken up, reacquire the monitor.
- ***timewait***(microseconds)
  When inside a monitor, release the monitor and wait for a specified time period. When time expires or when woken up by a call to *notify* or *notifyAll* reacquire the monitor.
- ***notify***()
  When inside a monitor, notify the first waiting for the monitor that it is about to be released
- ***notifyAll*** ()
  When inside a monitor, notify all those waiting for the monitor that it is about to be released

All these functions are available when a thread is inside a monitor object. That is, it must have gained access through the monitor's mutual exclusion mechanism before they are available.

There are 2 basic operations: *wait* and *notify*. Wait is used when a thread wishes to wait for the availability of a resource. Notify it used to notify those waiting that a resource is about to become available.

Consider the problem of the audio device. Let's write code to simulate it. The problem can be broken down to a set of threads wishing to queue a *track* to play on the speakers. Let's define a queue of tracks and a thread that extracts one at a time and plays that track on the speakers. The thread that plays the tracks is called the ***player*** and those queuing the tracks are called ***clients***.

First we need to define the track queue:

```
monitor Tracks {
   var tracks = []                    // vector of tracks
   var numTracks = 0                  // number of queued tracks

   public function addTrack (t) {
      // code to add a track to the queue
   }

   public function getTrack() {
      // code to get the first track on the queue
   }
}

var tracks = new Tracks()                    // an instance of the track queue
```

The addTrack and getTrack functions are the interface the to queue of tracks. Let's define that addTrack adds a track to the queue and getTrack extracts the first track on the queue. In addition, the getTrack function will not return until there is a track to play.

Let's define the player thread:

```
thread player() {
   for (;;) {
      var track = tracks.getTrack()
      System.println ("playing track " + track)
   }
}
```

This runs forever, getting tracks off the queue and playing them (actually printing the name in this example, but you get the idea).

A client can now be defined:

```
thread client (name) {
   foreach x 5 {                              // queue 5 tracks
      tracks.addTrack (name + x)
      sleep (System.rand() % 10000000 + 500000)        // wait for a random time
   }
}
```

This is really a test client that just queues 5 tracks at random times. The System.rand() function returns a pseudo-random value.

Now we need to write the addTrack and getTrack functions.

First the getTrack function. This must wait for a track to be available. We can do this with the wait() function.

```
public function getTrack() {
   while (numTracks == 0) {                    // wait until track is queued
      wait()
   }
   var t = tracks[0]                           // get first track on queue
   delete tracks[0]                            // delete the first element
   numTracks—                                  // one fewer tracks
   return t                                    // return track we got
}
```

The addTrack function will add a track to the vector called *tracks* and increment the number of tracks. When this is called the player may be waiting for a track to play, so the addTrack function uses the notify() function to tell the player that a track is ready.

```
public function addTrack(t) {
   t -> tracks                          // append to vector
   numTracks++                                 // another track added
   notify()                             // notify player
}
```

Now we can put it all together into a program that tests the player.

```
monitor Tracks {
   var tracks = []                      // vector of tracks
   var numTracks = 0                    // number of queued tracks

   public function addTrack(t) {
      t -> tracks                       // append to vector
      numTracks++                               // another track added
      notify()                          // notify player
   }

   public function getTrack() {
      while (numTracks == 0) {                 // wait until track is queued
         wait()
      }
      var t = tracks[0]                        // get first track on queue
      delete tracks[0]                         // delete the first element
      numTracks—                                     // one fewer tracks
      return t                                 // return track we got
   }
}

var tracks = new Tracks()                      // an instance of the track queue


// the player – play all queued tracks forever
thread player() {
   for (;;) {
      var track = tracks.getTrack()
      System.println ("playing track " + track)
   }
}
```

```
// a client – queue 5 tracks
thread client (name) {
   foreach x 5 {                                    // queue 5 tracks
      tracks.addTrack (name + x)
      sleep (System.rand() % 10000000 + 500000)          // wait for a random time
   }
}

// start player thread
var p = player()

// start 4 client threads
foreach c 4 {
   client ("client" + c)
}

// wait for player to finish (never)
join (p)
```

When the program is run, it output a random set of lines of the form:

```
playing track client00
playing track client01
playing track client30
playing track client20
```

And so on until all the tracks are exhausted. Then it loops forever waiting for more tracks. There won't be any.

10.2.2. Mutexes

It is sometimes easier to think in terms of locking a piece of memory before accessing it, then unlocking it afterwards. A monitor can be used for this purpose.

One problem with the Java™ method of locking (it uses monitors too) is that it lulls you into a false sense of security about the safety of your code. In the Java™ system, you place the keyword *synchronized* before a function definition and is makes a monitor out of the object referenced by the function. I have been caught thinking that just by a liberal sprinkling of *synchronized* throughout my code I was automatically multithread safe – I was wrong.

The same can be said of any monitor implementation. It makes the task of guarding protected data easy if you design the code to use them. It is very easy to just use monitors everywhere instead of classes. This will definitely work but it will be slower and may not fully protect you.

A simple mutex can be implemented by a monitor as follows:

```
monitor Mutex {
   var available = true                    // flag to say whether it is available

   // lock the mutex and return when lock granted
   public function lock() {
      while (!available) {
         wait()
```

```
        }
        available = false
    }

    // unlock the mutex.   Can only get here if we locked it
    public function unlock() {
        available = true
        notify()
    }
}
```

This can be used as follows. Let's take the previous example of incrementing a shared counter *count*. We can protect this variable with a mutex as follows:

```
var countLock = new Mutex()

function incCount() {
    countLock.lock()
    count++
    countLock.unlock()
}
```

Threads wishing to increment the count can either call this function directly of provide the equivalent to it inline.

10.2.3. Semaphores

Semaphores are another simple multithreading tool that can be easier to use than monitors. The too can be implemented by using a monitor.

A semaphore is a variable that maintains a counter. Threads increment and decrement this counter corresponding to the availability of the resource guarded by the semaphore. If a thread tries to decrement the counter and it is already at zero then it has to wait until the counter goes above zero by some other thread incrementing it.

Let's call the act of decrementing the counter *take*() and that of incrementing it *put*(). Here is one possible implementation of a simple counting semaphore.

```
monitor Semaphore (count = 0) {
    public function take() {
        while (count <= 0) {
            wait()
        }
        count--
    }

    public function put() {
        count++
        notify()
    }
}
```

The monitor takes a parameter specifying the initial value of the counter (default of zero). The take() function waits until the counter is above 0, then decrements it. The put function increments the counter and notifies the first thread in line waiting for the semaphore.

Note that this (and the mutex implementation) use notify() rather than notifyAll() in to wake up the threads after the resource becomes available. This is to guarantee a FIFO ordering for the requests. If notifyAll() was used, all the threads would wake up at once and there would be no guarantee of the order in which they will try to get the resource again.

Semaphores are usually pressed into service for resource allocation. Consider a resource protected by a semaphore. Say we have 10 instances of the resource available and we want to dish it out to the requesting threads on a FIFO manner.

```
var res = new Semaphore (10)

function grabResource() {
   res.take()
}

function releaseResource() {
   res.put()
}
```

The first 10 threads to call the grabResource() function will get the resource, with the rest waiting for one who has it to free it using the releaseResource() function.

10.3. Synchronization

A monitor is a class in which the interpreter gains a lock before any access is made to the internals of an instance of the class. Other languages have the keyword *synchronized* that can be used for the same effect. Aikido also provides this facility for compatibility reasons.

For example, a monitor can be defined in Aikido as follows:

```
synchronized class Semaphore {
   // contents
}
```

The *synchronized* keyword may also be used as a statement to create temporary locks on objects. See section 7.10 for details.

A function member of a class may also be synchronized by placing the synchronized keyword before the declaration. The effect of this is to transform the class into a monitor for the duration of the call to the function only. For example:

```
class Concordance {
public:
   synchronized function add (word) {
      // object is mutex locked here
   }
}
```

In the absence of early return from the function, this is equivalent to:

```
class Concordance {
public:
   function add (word) {
      synchronized (this) {
        // locked here
      }
   }
}
```

10.4. Thread streams

A novel feature of the Aikido threading mechanism is the ability to use a stream to communicate with a thread. A traditional threading model uses semaphores and shared memory in order for one thread to talk to another. When a thread is spawned in a Aikido program, the interpreter creates a stream connecting the thread itself and the caller of the thread.

Every thread in a program (including the main thread) gets 2 variables called *input* and *output*. These variables are connected to stdin and stdout respectively for the main thread. For a thread spawned inside the program they are connected to the stream created for the thread.

The input and output streams for a thread are the main means of communication to the thread. When the program creates a thread, the return value of the thread call is a stream connected to the thread. The caller can then use this stream to send data to and read data from the thread. Consider the following example:

In effect, the Aikido thread stream mechanism is similar to the Ada *rendezvous* where the threads arrange to meet at a particular time in order to exchange data.

The fact that threads have special input and output streams makes then symmetric to a process. In most operating systems, a process gets 2 or more streams through which it may communicate with the outside world.

See section 9.3 for an example of use of the thread streams.

Note that the *join*() call uses the stream connected to the thread rather than a traditional thread identifier.

Chapter 11. Writing reusable code

Any program of any reasonable size will have to be divided up into a set of modules. In a compiled programming language the modules consist of separately compiled files. These files are compiled into an object code format that describes the visible contents of the module.

Aikido takes a different approach. Everything in Aikido is in source code form. There is no compiler as such (although the parser does operate like a traditional compiler front end), so there is no object code. The visible state of a Aikido program is the public interfaces provided by the packages and classes.

A Aikido program is divided into a set of packages, classes, functions (etc. collectively known as blocks), and executable statements. The blocks form enclosures for other blocks and statements. Each block can define a set of its contents to be publicly accessible and therefore form its interface.

The program can be split into a set of separate files that are read by the parser when the it is run. All the files required by the program must be available to the parser either by explicitly telling it about their existence or by use of the import statement.

11.1. Import files

An import file is a subset of a program. It contains Aikido declarations and code. A file is imported by using the import statement of the language. This is similar in concept to taking the contents of the file named in the import statement and inserting them in place of the import statement. That is, the import statement itself is replaced by the contents of the file being imported.

This is not a precise definition of the import. If this were the case, contents of the import file would be inserted in exactly the same scope as the import statement. This would mean that if, for example, the import statement was inside a class, the contents of the file being imported would only be available to the class performing the import. This is not desirable because it would result in copies of the import files being inserted in multiple places in the program, unnecessarily increasing its size. Instead, the contents of the import file are placed in the top level scope of the program. There is a mechanism in place to stop a file being imported more than once in the whole program. Consider the following:

```
// in file myfile.aikido

function myprint (s, stream) {
   ["*", s] -> stream
}
```

This shows an import file that defines a single function called *myprint*. This function simply prints to a given stream, prefixing the output with an asterisk (for some unknown reason). If you wrote the following code:

```
thread printer {
   import myfile

   myprint ("printer running"
   // and other silly code
}
```

Then you could use the function myprint anywhere in your program (after the thread printer is defined of course). For example:

```
printer()                          // start the printer thread

myprint ("started printer")        // call the myprint function
```

So, you decide that you want to use an import file. You will want to do this if you are using any of the system library facilities or simply want to divide your program into pieces. Say you want to use the *Vector* monitor provided by the system library. To use it you must import the file containing its definition. Do this by:

```
import vector
```

This searches for the file named 'vector.aikido' in certain places (defined later) and when it finds the file it opens it and inserts the contents into the program after first moving to the top level scope (main).

The name of an import file need not just be a single identifier. It may be a set of identifiers separated by dots. This modifies the name of the file to search for. For example, you may import a file:

```
import com.sun.labs.nametool
```

This looks for a file consisting of the components com, sun, labs and nametool in that order. Where this file actually resides depends on the operating system. In UNIX®, the dots are replaced by slash characters so the file would be called: "com/sun/labs/nametool.aikido"

The import statement is also used to load native code into the interpreter. The native code is in files called *shared object* files. These are special files produced by the operating system with the conventional name of:

lib*name*.**so**

That is, the name of the file, prefixed with the sequence 'lib' and suffixed with the extension '.so'.

11.1.1. Search paths

So where does Aikido look for import files? This is a complex problem that perplexes most languages that allow files to be included. An import file can be either a Aikido source file (with the extension ".aikido") or a shared object file with the extension ".so" (on UNIX®).

The following algorithm is used to find an import file: First the suffix ".aikido" is appended to the file name formed from the identifier sequence in the import statement. Then the following places are searched in the order given:

1. The 'Aikido.zip' archive file
2. The directory in which the 'Aikido' executable resides
3. The set of directories in the AIKIDOPATH environment variable
4. The current directory

Then the '.aikido' suffix is removed and replace by ".so". The following places are searched for the new file:

1. The directories specified in the LD_LIBRARY_PATH environment variable
2. The directory in which the 'Aikido' executable resides
3. The set of directories in the AIKIDOPATH environment variable
4. The current directory

If the file cannot be found an error is reported by the parser. All import files must exist.

The file 'Aikido.zip' is a standard zip archive containing the system import files, such as 'vector.aikido', 'streambuffer.aikido', etc.

The system searches the directories contained in environment variables. The variables used are: LD_LIBRARY_PATH and AIKIDOPATH. The former is the standard search path for shared libraries.

AIKIDOPATH specifies a set of directories to search when looking for import files. It is the same format as the LD_LIBRARY_PATH variable, in that it consists of a set of directory names separated by colon characters. The search is made from the first directory to the last and it stops when the file is found.

For example, if the following command was issued in the shell:

```
% setenv AIKIDOPATH /usr/proj/imports:/usr/local/imports:/home/fred
```

Then the search for an import file will start with */usr/proj/imports*, then */usr/local/imports* and finally */home/fred*.

11.2. Native functions

Although Aikido provides a powerful set of language features and is capable of doing a large number of things, you still need to be able to code a function in another (compiled) language. You might want to do this if:

- The interpreted Aikido version is too slow
- You want access to a native function provided by the operating system
- You want to call into an existing library of functions or a package supplied as object code
- Aikido is too high level for what you want to do

The ability to mix things coded in Aikido and other things coded in another language is an asset to Aikido and is not discouraged. Unlike the Java™ language where the existence of a native function makes the program non-portable, Aikido is agnostic about what language something is written in.

When you write a module using a native function you will generate a shared object file. This file can be loaded into Aikido using an import statement just like any other import file. You will need to make a regular Aikido import file to declare the native functions to the parser. This regular Aikido file will also import the shared object file so its existence should be invisible to the users of your code.

11.2.1. Writing Native functions

Native functions are normally written in C++, but C is also possible (with a C++ wrapper). The steps involved are:

1. Decide on what native functions you need
2. Declare the native functions in a Aikido file
3. Make a '.cc' file containing the functions
4. Compile the C++ file using a C++ compiler
5. Link the object file to make a shared object file
6. Import the shared object file in the Aikido file
7. Place the Aikido file and shared object file in a location that can be searched with the import statement.

11.2.2. Deciding on what functions you need

Before writing any software you need to know what you are going to do. This is no different when deciding what native functions to provide. Use common sense. A function should be native only if it is really necessary to make it so. If a regular Aikido function will do, you should probably use one. Native functions are more hassle to write. You have to use a C++ compiler and maintain another set of files (the source and shared object files).

Use the criteria in section 11.2 to help you decide if a native function is justified. Aikido provides a lot of high level functionality. It will be better if you know what Aikido can do before making the decision. For example, before going off to write a function to string manipulation it would be better if you checked what Aikido can do with strings. A lot of work has gone into making the facilities in Aikido useful.

11.2.3. Declaring the native functions

Usually a native function will be provided as part of a bigger package (or class, etc). Say you are writing a screen manipulation package and need a native function to move the cursor around the screen. It has been decided that this would be much better to code in C++ because of the performance requirements. Let's call the package *tty*.

```
package tty {
   // contents of tty package
}
```

We decide to call the function tty_goto() taking 2 parameters: the row and column on the screen. The purpose of the function is to move the cursor to the given row and column. The declaration would be:

```
package tty {
   public native tty_goto (row, col)

   // rest of functions
}
```

11.2.4. Writing the C++ code

Writing the C++ code is pretty straightforward. Create a file called tty.cc and in it put:

```
#include "Aikido.h"

using namespace Aikido ;

extern "C" {

AIKIDO_NATIVE (tty_goto) {
   if (paras[1].type != T_INTEGER) {
      throw Exception ("Illegal type for row parameter")) ;
   }
   if (paras[2].type != T_INTEGER) {
      throw Exception ("Illegal type for col parameter")) ;
   }

   int row = paras[1].integer ;
   int col = paras[2].integer ;

   // code for moving the cursor to (row, col). Probably escape codes
```

```
    return 0 ;
}

}
```

The first thing to do is include the Aikido main header file “Aikido.h”. This defines everything you need to write the native functions. Everything in Aikido is in the C++ namespace “Aikido”, so we issue a using directive to dump this into the local namespace.

All the native functions must have C linkage (no mangling by the C++ compiler please), so the extern “C” declaration specifies this.

Next comes the function itself. The AIKIDO_NATIVE macro is defined in “Aikido.h” as a function declaration and expands to the following (for the tty_goto function):

```
Aikido::Value Aikido__tty_goto (Aikido::Aikido *b,
        Aikido::VirtualMachine *vm,
        Aikido::ValueVec &paras,
        Aikido::StackFrame *stack,
        Aikido::StackFrame *staticLink,
        Aikido::Scope *currentScope,
        int currentScopeLevel)
```

Then follows the body of the function. It is not necessary to understand what each of the parameters is. The only one you need is the ‘paras’ parameter. This is a vector of Aikido::Value structures containing the values passed as parameters to the native function. The Aikido::Value is paraphrased as:

```
struct Value {
    union {
        string *str ;
        INTEGER integer ;
       vector *vec ;
       map *map ;
       Function *func ;
       Thread *thread ;
       Stream *stream ;
       Class *cls ;
       Package *package ;
       Block *block ;
       Enum *en ;
       EnumConst *ec ;
       double real ;
       Object *object ;
    } ;
    Type type ;
}
```

That is, a structure containing a union of data elements for each of the built-in values and an element containing the type of the value.

A Type is an enumerated type with the following members:

Name	Meaning
T_INTEGER	64 bit signed integer
T_CHAR	single byte character
T_STRING	string of characters
T_VECTOR	vector of values
T_FUNCTION	function
T_STREAM	stream
T_MAP	map of value/value pairs
T_THREAD	thread
T_OBJECT	object (instance of class or monitor)
T_CLASS	class
T_INTERFACE	interface
T_PACKAGE	package
T_ENUM	enumeration
T_ENUMCONST	enumeration constant
T_REAL	real number
T_MONITOR	monitor
T_NONE	no value

In order to fully understand each of the data types it will be necessary to go and look at the Aikido.h header file. I leave that as an exercise to anyone writing native code.

The function we have written first verifies that the parameters are of the correct type. Since this function is defined inside a package (the tty package) the first parameter (paras[0]) is always passed as a pointer to the instance of the package. The first user parameter is paras[1]. If any of the types is incorrect an exception is thrown.

We then extract the row and column parameters from the paras vector.

The last thing we do is return a value from the function. Since this is effectively a void function we return a zero integer value.

11.2.5. Compiling the C++ code

So we have written the C++ code and now want to compile it. To do this we need a decent C++ compiler that supports the latest C++ standard. In particular we need:

- string support
- vector support
- map support
- namespaces and using directives
- exception support

The first 3 are components of the Standard Template Library (STL).

Once we have access to a good C++ compiler (the Sun C++ compiler version 6 or later is a good one) we can start to compile the code. We will need to know where the file Aikido.h can be found. Let's define this as an environment variable AIKIDODIR. Let's assume that the C++ compiler is called CC.

```
% CC  -I$AIKIDODIR –c  tty.cc
```

The '-I' flag tells the compiler where to look for header files. The '-c' flag tells it to generate an object file and not to run the linker. When this command completes we will have a file called tty.o in the same directory.

A better way to do this is to use a *makefile* for input to the *make* utility on UNIX®. This is left as an exercise.

11.2.6. Linking the object code

We now have a file called 'tty.o'. This is an object file compiled from 'tty.cc' and contains our native function. In order to import this file into the Aikido interpreter it is necessary to convert it into a shared object. This is done using the linker. The linker on UNIX® is called 'ld' and it takes a flag '-G' to tell it to make a shared object out of a set of object files. Issue the following command:

```
% ld –G –o libtty.so tty.o
```

This will produce a file called '*libtty.so*' in the current directory.

11.2.7. Importing the shared object

Now we need to import the shared object we just created into the Aikido import file. We do this by use of a standard import statement:

```
package tty {
   import libtty

   public native tty_goto (row, col)

   // rest of functions
}
```

Because we called the file *libtty.so*, there is no clash with the object file *tty.o* or the Aikido import file *tty.aikido*.

Now, any time you import *tty.aikido* you will also get *libtty.so*.

11.2.8. Placing the files

In order for Aikido to find the files you need to place them in a place that is searched when looking for import files. This can be any of the places mentioned in section 11.1.1. The easiest way to do this is to set the AIKIDOPATH variable to contain the path to the files.

Another way to do it is to pass the flag –I<dir> to the Aikido interpreter. This adds the named directory to the list of places searched for the files. There can be many –I flags and they are searched in the order given in the command.

11.3. Libraries

As far as Aikido is concerned, a library is a piece of source code that can be imported into a program. A library should contain code that can be reused in any circumstances. It general a file provide by a library should contain a package, although classes and monitors are also acceptable. The use of a block at the top

level in a library reduces the pollution of the global namespace and thus the possibility that your carefully chosen function name will clash with one that the programmer using your library has chosen.

Placing all the functions you want to provide inside a package allows the user to call them through the name of your package. This may be an appropriate interface. For example, suppose you were providing a complex number library. It would make sense to place this in a package named for the function of the library:

```
package ComplexNumbers {
   public class Complex (re, im) {
      //
   }
}
```

The user can then refer to the contents of the library as:

```
var c = new ComplexNumbers.Complex (1, 0.2)
```

If you just placed the class Complex at the top level and someone else also has a class called Complex in the same program then the interpreter would complain about a duplicate definition.

Of course, for some things this is inappropriate. For example, the monitor Vector provided by the system library probably should not be in a package as it will be used frequently and is a sufficiently well recognized concept. Having to type:

```
var v = new VectorPackage.Vector()
```

Is a little onerous for the programmer.

11.3.1. The main function

When writing a program in any language you are often faced with the question of whether to make the program standalone or make it reusable. For example, when coding in C++ you have to make the decision of whether to include a 'main()' function or not. If you do include main() you will be precluding some future use of the code in a library where main() will come from elsewhere.

Languages such as Pascal did not even have the concept of a program that was not meant to be executed directly (unless by an vendor-supplied extension).

The Java™ language, however, has a good approach: any class may have a function called 'main' with a defined return type and argument list. If the class name is provided to the interpreter as the starting class and that class contains the designated main function then execution starts at that main function. If other 'main' functions exist in other classes they are not used unless specified. This has many advantages.

Any code you write needs to be tested. The simplest way to test a piece of code is to write a test harness that talks to the code under test and provides a set of input data. This test harness needs to be executable. In a C++ program you may decide either to include the test harness in a separate file and link that file with the code under test. This way the test harness is not supplied as part of the code. The Java™ method let's you embed the test harness inside the class to be tested and supply it with the class. So, the class comes with its own test code that can be rerun at any time. Another approach that can be used in C++ is to provide a main function in a conditionally compiled section of a file. This can then be compiled in or out by setting a compile time macro.

The Aikido approach to this is very similar to that of the Java™ language. Consider the following trivial program:

```
function factorial (n) {
    if (n <= 1) {
        return 1
    } else {
        return n * factorial (n – 1)
    }
}
```

That's the whole file. We want to supply this as part of a mathematical library, but we need to test it. We could add the following code to the file:

```
// integer test
for (var  f = -1 ; f < 20 ; f++) {
    System.println ("factorial (" + f + ") = " + factorial (f))
}

// real test
for (var f = -1.0 ; f < 100.0 ; f++) {
    System.println ("factorial (" + f + ") = " + factorial (f))
}
```

If we do this and find that it all works we have a choice to make. We can either remove the code from the file (by comments or deletion), or we can keep it in. If we remove the code then the factorial function may be used as part of a library without executing the test code ever time we import it. Commenting out the code will work as long as there are no nested comments. Deleting the code will also work except we will not be able to rerun the test.

If we keep the code in the file then every time we import the file, the test harness code will be rerun. This is harmless in this case except for a number of lines of output, but may not be feasible if the code enters an infinite loop.

Another approach is to place the code in a function called 'main' at the top level in the file. When the interpreter runs it will look at the first file it is given to see if it contains a function called 'main' inside the main package. If it exists it will arrange to execute that function after all the code has been executed in the file. If the file just contains definitions of functions, packages, classes, etc. then the effect will be just to run the main function. So, we can rewrite our test harness as:

```
function main {
    // integer test
    for (var  f = -1 ; f < 20 ; f++) {
        System.println ("factorial (" + f + ") = " + factorial (f))
    }

    // real test
    for (var f = -1.0 ; f < 100.0 ; f++) {
        System.println ("factorial (" + f + ") = " + factorial (f))
    }
}
```

The file now contains 2 function definitions: factorial and main. If the file is imported the main function will never be run but if the file is given as the first argument to the interpreter it will be run after all the other code in the program. Since there is no other code in the program it will be the only thing run.

Chapter 12. Macros

Aikido support a statement-level macro system. A macro is a much-maligned facility available in the C family of languages that works at the source code level. What this means is that a macro actually replaces the source text instead of being processed by the language parser.

The C family of languages have the "C Preprocessor" available to them. This is a powerful, yet much misused feature of the language that has become unpopular these days with the introduction of more powerful language features such as templates and explicit constants. It is, nevertheless, a very popular way of controlling the input to the compiler.

Macros in Aikido are a lot different from the standard C mechanism. A Aikido macro is an identifier that maps on to a piece of text. The identifier can only be used "at the statement level" in a Aikido program. That is, it can be use wherever a statement may be used. This limits the power of the macro but also limits its potential for abuse.

A macro is defined in a Aikido scope by use of the **macro** statement:

> **macro** *identifier arguments*$_{opt}$ { ***linefeed***
> *macro body*
> }

Unlike a normal statement, a macro is line based. This means that the first line of the macro body must appear on the next line of source text. The macro body is enclosed in braces just like a regular compound statement.

The macro has a name and possibly a set of arguments. The arguments are a set of comma-separated identifiers. Note that there are no parentheses around the argument list for a macro.

The body of the macro contains any text. Whenever the macro name is used as a statement, the macro name and any arguments are replaced by the macro body before the parser gets a chance to parse it. Thus a macro provides the potential to let you make your own statements and enhance the language. The name of a macro may not be an existing reserved word, so you can't redefine the meaning of a builtin statement.

I tend to think that macros are of limited use, but are present in the language because of its assembler legacy. They are very useful in an assemler.

12.1. The Inner Statement

A macro is a scope of its own. This implies that the macro body is a complete statement and cannot be used as part of another statement. The reason for this is that the macro may define variables and these shouldn't affect the variables in the scope in which the macro is instantiated. Also, the macro may be instantiated many times within the same block of code and if it defined its own variables they would become multiply defined and cause errors.

Consider the following simple macro:

```
macro forever {
   for (;;)
}

forever {
```

```
    // do something
}
```

As it stands this macro will not work as expected. The expansion of the macro will be equivalent to:

```
{ for (;;) }
    // do something
}
```

Which contains a number of errors. The first is that the **for** loop does not have a body and will therefore give a syntax error in the parser. The second is that the open brace after the *forever* will be consumed as part of the macro instantiation and therefore the close brace at the end of the supposed macro body will be misplaced.

The correct definition of such a macro uses the *inner statement*. This is a special statement that can only be used within a macro body and is replaced by a block of text when the macro is instantiated. Consider the following correct macro definition:

```
macro forever {
    for (;;) {
        ...                         // inner statement
    }
}

forever {
    System.println ('.')
}
```

The ellipsis notation (...) is the inner statement. In the above example it is replaced textually with the text passed in the braces after the *forever* instantiation. The inner statement may only be used inside a macro body – outside will result in errors. If it is present, the macro must be followed by a block of text and that text must be on a new line from the macro instantiation. The text must be enclosed in braces and the open brace must be on the same line as the macro instantiation. The above example shows the correct form. Here is a couple of incorrect forms:

```
forever System.println ('.')        // error: too many macro parameters

forever                             // error: no open brace
    System.println ('.')

forever                             // error: open brace on new line
{
    System.println ('.')
}
```

Thus the inner statement is fairly restrictive syntactically.

There may only be one inner statement in a macro. See section 12.4.1 for details of the operation of the inner statement with respect to macro inheritance.

12.2. Macro arguments

A macro can contain a set of arguments. These are specified when the macro is defined as a series of identifiers separated by commas as follows:

```
macro testmac arg1, arg2, arg3 {
  // body
}
```

This shows a macro called *testmac* with 3 arguments. Inside the body of the macro the arguments can be used. In order to use a macro argument you have to prefix it with a dollar sign ($). Remember that macros are a text-level feature of Aikido so a macro argument is not a regular variable as far as the parser is concerned. In fact, the parser never sees a macro argument. Consider the following:

```
macro loop from, to, inc {
   for (var x = $from ; x < $to ; x += $inc) {
      ...
   }
}

function X {
   loop 3, 70, 4 {
      System.println ("loop: " + x)
   }
```

The invocation of the *loop* macro is textually replaced by:

```
for (var x = 3 ; x < 70 ; x += 4) {
   System.println ("loop: " + x)
}
```

The macro instantiator collects the arguments for the macro by scanning forward in the program text until it gets to a comma or an open brace. The comma and open brace are absorbed.

A macro argument may be given a default value. This default value must be a string and will be used if the macro argument is omitted. Unlike the arguments for blocks, you can omit a macro argument actual value at in any part of the argument list. Consider:

```
macro loop from = "0", to, inc = "1" {
   for (var x = $from ; x < $to ; x += $inc) {
      ...
   }
}

loop ,100 {
   System.println ("loop: " + x)
}
```

This macro invocation omits the first and last argument from the macro instance. The first is omitted by the appearance of a comma without any preceding text. The last is omitted by the appearance of an open brace. Thus the above would loop from 0 to 99 in steps of 1.

The use of a macro argument inside the macro body is controlled by the dollar sign ($) preceding the argument name. If the name is determinable by scanning forward in the text (that it, it is an identifier terminated by a non-letter or non-number character) the dollar can immediately precede the identifier. If the identifier is part of a larger string of letters and numbers then you must enclose the macro argument name in braces. Consider the following:

```
macro loop2 from, to, inc, prefix {
```

```
    for (var ${prefix}_x = $from ; ${prefix}_x < $to ; ${prefix}_x += $inc) {
        ...
    }
}

loop2 0, 100, 3, my {
    System.println ("loop: " + my_x)
}
```

Here the macro has an argument called prefix that is used to prefix the name of the loop variable. The macro argument name must be enclosed in braces because it is immediately followed by an underscore character in the macro body.

12.3. Macro scope

A macro is defined in a scope just like any other variable or block. When the parser is searching for a macro it uses the same rules as for the search for variables.

Macros can define variables and other items inside their body. The body of a macro acts like a brace-enclosed scope, much the same as a block (function, class, etc.). Thus any variables or blocks defined within the macro are local to that macro. This also includes other macros.

When a macro is instantiated there are actually 2 active scopes: the scope of the actual macro itself; and the scope in which it is being instantiated.

Consider the following example:

```
macro repeat n {
    var x = 0
    while (x < $n) {
        ...
        ++x
    }
}

function T {
    var t = 3
    repeat 10 {
        System.println ("t = " + t)
        ++t
    }
}
```

The *repeat* macro iterates through its inner statement a number of times controlled by the macro parameter 'n'. The function is expands to:

```
function T {
    var t = 3
    { var x = 0
        while (x < 10) {
        System.println ("t = " + t)
        ++t
        ++x
     }
    }
```

```
}
```

When inside the repeat macro, the scopes that are available are:

- The scope of the *repeat* macro itself. This is where the variable ‘x’ is declared.
- The scope of the function T. This is where the variable ‘t’ is located

12.4. Macro inheritance

Macros are allowed to be inherited from another macro. Just like blocks, a macro can be defined to extend another previously defined macro definition. Consider the following macros:

```
macro base x {
   // body of base
}

macro derived y extends base "." + $y {
   // body of derived
}
```

This shows a macro called *derived* that is derived from the macro *base*. The macro argument ‘x’ of the *base* macro is passed the concatenation of the string “.’ and the value of the ‘y’ argument of the *derived* macro. When the macro *derived* is instantiated it also instantiates the *base* macro (with the appropriate parameter value). Notice that the arguments of the *derived* macro must be prefixed by a dollar sign when used as actual arguments for the *base* macro. This is symmetric with the use of macro arguments inside the macro body.

This facility is useful when you have a set of macros that have common parts. The macros used as bases for derivation can include things like macros and variables that are made available to derived macros.

The regular access controls apply to macro inheritance as they do with block inheritance. Consider the following:

```
macro base x {
   var a = 1
   protected macro m {
   }
   public var b = 3
}

macro derived y extends base "." + $y {
   // body of derived
}
```

The macro *derived* has access to the macro ‘m’ and the variable ‘b’ from its base macro. The variable ‘a’ is not accessible.

12.4.1. Behavior of inner statement in inheritance

The inner statement of a macro is problematic when dealing with macro inheritance. The basic rule is the inner statement can only exist once in a whole macro inheritance tree. Consider the following:

```
macro loop from, to, inc {
   for (var x = $from ; x < $to ; x += $inc) {
      ...
   }
}

macro myloop from, to : loop $from, $to, 1 {
}

function X {
   myloop 3, 70 {
      System.println ("loop: " + x)
   }
```

This shows the definition of a new macro *myloop*, derived from the *loop* macro previously seen. The instantiation of the *myloop* macro would work fine in the function X because there is only one instance of an inner statement in the whole inheritance tree. If we had defined *myloop* as:

```
macro myloop from, to : loop $from, $to, 1 {
   ...
}
```

Then the macro is broken. Which of the inner statements will the parser take when the *myloop* macro is instantiated?

Chapter 13. Garbage collection

Garbage collection refers to the act of cleaning up after the program while it is running. During execution a program will allocate temporary values and discard previously allocated objects. The garbage collector is responsible for making sure that the memory occupied by the discarded items is returned to the pool of memory available for new allocations.

In a traditional C or C++ program the programmer can allocate objects using the *malloc*() function or *new operator*. This memory comes from the pool of memory made available by the operating system for free store (sometimes called the *heap*). Once the program allocates memory it is the programmers responsibility to make sure that the memory is freed when the program no longer requires it. Failure to do so could result in programs running out of memory.

Garbage collection seeks to remove the requirement placed on the programmer to manage the free store allocation and deallocation by himself. A garbage collector is a piece of software that is intimately aware of all the objects that have been allocated from the free store and when then may be put back into the free store. It keeps track of all the objects allocated by the program and their scope. When an object goes out of scope and is therefore no longer accessible by the program it can be returned to free store.

There are multiple methods of achieving garbage collection. It is not the intention of this book to go into depth on the various techniques available, but a quick overview wouldn't hurt. See note [1] for a book on garbage collection.

13.1. Mark and Sweep Garbage Collection

The basic idea of this is that the garbage collector makes a scan of every object in the heap marking it if it is referenced by another object. Once all the objects have been scanned, anything that is not marked is deemed garbage as it has no references. It then makes another pass and deletes all the garbage. So there are 2 passes: a *mark* pass and a *sweep* pass.

For this to work, the garbage collector must know where every object is in the heap. It must keep a directory of every object so that is may make the passes. This implies that the garbage collector must be part of the heap software itself and not a layer on top.

While the mark and sweep passes are operating, the program must be idle, as there cannot be any modifications to the objects in the heap. This means that the heap must be locked (mutex lock or semaphore, or some other method) during the garbage collection operation. The question is, when is the best time to do this?

One approach is to wait until the heap fills up and then perform the mark and sweep. This can work when the heap is bounded and small but may cause a large resident set of memory pages in a virtual memory system. Another approach is to wait until the program is idle and try the collection then. This has the problem that it is very hard to guess when the program will be idle. Past behavior is not a good predictor of future behavior.

Since the heap must be locked, the operation of mark and sweep garbage collection will necessarily cause a delay in the operation of the program while it is running. This may be unacceptable for a real-time system

[1] A good book on garbage collection is "Garbage Collection: Algorithms for Automatic Dynamic Memory Management" by Richard Jones and Rafael Lins. Published by Wiley in 1996.

or a server that must always be available and where a delay of a couple of seconds (or more) will cause uneven performance.

13.2. Reference Counting Garbage Collection

In this form of garbage collection, each object subject to garbage collection maintains a count of the number of objects that reference it (point to it). Each time a new object is allocated the reference count is set to zero, then when the object is referenced its reference count it incremented. When a reference is removed the counter is decremented. When the counter reaches zero the object has no references and is therefore garbage.

Unlike the mark and sweep methods, this does not require the heap to be locked while garbage collection is happening – it happens while the program is running. The downside is that any assignment of a pointer to an object is slower to execute, as the reference counts must be maintained. The cost of garbage collection is spread throughout the execution of the program. It just happens as a consequence of assigning and removing pointers to objects. This makes it more suited to real-time and server applications where the delay caused by mark and sweep would cause an unacceptable performance delay.

Reference counting garbage collection has one major disadvantage: it does not work where you have 2 objects that refer to each other. Say we have 2 objects (object A and object B). Object A has a reference to object B, and object B has a reference to object A. We also have another object referring to A. When the external reference to A is removed, its reference count is decremented, but will not get to zero because it still has a reference from B.

This obvious problem can be dealt with in a number of ways. One way is to ignore the problem and suffer the consequences of memory leakage due to objects that cannot be deleted. Another way is to combine reference counting with some form of mark and sweep collection. Yet another way is to allow the programmer to have some control over the reference count.

13.3. Copying Garbage Collection

In this form, the heap is divided into 2 sections. One section is the *active* section and this is where all the objects reside. This is similar to the mark and sweep collection technique described in section 13.1. The heap must be locked and but only 1 pass is performed. While the heap is locked, the collector visits every object and if it is referenced, copies the object from the active section of the heap to the other section. It must then keep track of the fact that it has been moved so the other references wont copy it again. When all the objects have been traversed, the roles of the 2 heap sections are reversed.

The advantage of this form of collection is that the heap is automatically compacted when the copy is done. This reduces heap fragmentation and can improve the performance of the program. The disadvantage is that twice the memory is required to run the system, or put another way, you can only use half the memory allocated to the program at any time. This may not be such a big problem as memory sizes increase.

13.4. Other Garbage Collection algorithms

There are other algorithms available for performing garbage collection. They all rely on the ability of the program to automatically detect whether an object is referenced or not. For full information on the current garbage collection techniques, the reader is referred to note 1 on page 165.

13.5. Garbage Collection in Aikido

Aikido uses the Reference Counting technique for its garbage collection functions (section 13.2). This relies on each object in the program having a counter for the number of references to that object. An object

in Aikido is not just those that are instances of a class, but anything that is allocated from the free store. This includes vectors, strings and maps. Integers and other scalars are not allocated from free store and are therefore not handled by the garbage collection routines.

When a program is running, Aikido keeps a count, in each object, of the references to that object. When the reference count goes to zero then the object is deemed garbage and is deleted.

The underlying memory system in Aikido is the basic *new* and *delete* system provided by C++. This maintains a single heap of memory blocks and arranges it so that it is efficient to both allocate new blocks and free existing ones. Fragmentation is always possible with this system but since the same memory allocation scheme is used by the operating system itself it is hoped that it is as efficient as possible.

The following values are subject to reference counting:

- Strings
- Streams
- Vectors
- Maps
- Objects (instances of blocks)

An overhead of 4 bytes is added to a value that is reference counted. This holds the reference count itself. The cost of the garbage collection is spread out though the execution time of the program. There is no heap locking performed. This means that the Aikido program will be slower to execute when garbage collected objects are used, but since Aikido is interpreted anyway the overhead is not noticeable.

Garbage collection happens all the time, sometimes when you don't expect it. Consider the following statements:

```
["the answer is ", answer, '\n'] -> stdout          // statement (1)

System.println ("the answer is " + answer)              // statement (2)

var a = "the answer is " + answer                       // statement (3)
System.println (a)                                  // statement (4)
```

In statement (1) a vector is created and sent to standard output. When the statement completes, the vector is deleted.

In statement (2), a string is created by the concatenation of a string and another value. This string is passed to println. The string is deleted when the statement completes.

Statement (3) creates a string by concatenation. It is not deleted because the variable 'a' references it. When 'a' goes out of scope (the function containing it returns or an object containing it is deleted), the string will also be deleted.

In statement (4), the variable 'a' is passed to println. The string is not deleted after the call because variable 'a' references it.

Aikido contains a *delete* statement. This seems unusual for a garbage-collected language but is necessary to get around the major flaw with reference counted systems. Remember that a reference counting garbage collector cannot deal with mutually referential objects (where they refer to each other). In section 13.2, there were three alternatives suggested for dealing with this problem. The method used in Aikido is to allow the programmer to explicitly decrement the reference count of an object. This will force the deletion of a set of mutually referential objects.

For this to work, the programmer must be aware of the structure of the program. You can, of course, totally ignore the problem and be prepared to handle memory leaks (don't care about them). You can also take the C++ approach and do all the memory allocation and deallocation yourself. Using *delete* on an object when you no longer need it does no harm and will help the garbage collector by seeding it with an object to start with. Alternatively, you can be aware of the Aikido garbage collection and only deal with those problematic object structures.

If you take the approach of deleting everything when it is not needed, the Aikido interpreter will take care of the rest of the temporary objects that are created for you (by string concatenation, say).

So, Aikido provides the best of both worlds with respect to garbage collection. If you are a C++ programmer who likes the deal with the problem yourself you can use the *delete* operator to explicitly delete the objects. If you are a Java™ programmer, or one who is lazy like myself, then the system will do the right thing in most circumstances. You just have to take care of those times where the garbage collector can fail.

13.6. The finalize function

Like the Java™ language, an object may include a function called 'finalize' to be called when the garbage collector decides to delete the object. This may be used to clean up anything that is required before the object memory is freed. For example, an open file may be closed.

```
class File (name)
    var stream = openout (name)
   // …

   public function finalize() {
    close (stream)
    }
}
```

Chapter 14. Dynamic Loading

As Aikido is an interpreted language, it is natural to expect that it is possible to dynamically load code into the interpreter at runtime. Aikido allows any source code to be loaded into it while it is running. Loading code into a running program presents a number of problems. In particular, in Aikido it is possible to extend blocks (classes, functions, etc) and add things to them. It is conceivable that code loaded into the interpreter might want to do this. If it does, what is the effect on existing blocks? Are they extended?

There are 2 facilities provided in Aikido for loading and executing source code at runtime. The first is the simplest: dynamic expression evaluation. This allows a string containing an expression to be parsed and evaluated at runtime. The expression can contain anything that could be used in an equivalent non-dynamic expression at that point in the program.

The second facility is more comprehensive: dynamic code loading. This allows any code contained in a file or generated by the program itself to be parsed and executed. There is no restriction on the contents of the code to be loaded. The effect of loading code in this way is the same as if the programmer had inserted the code at that point in the program inside a package.

The dynamic loading facilities are provided by functions inside the System package rather than directly by the language itself.

14.1. Dynamic expression evaluation

This is the simplest form of dynamic loading. The function used to do this is the *System.eval*() function. This takes a string as a parameter. The string is parsed as if it is a Aikido expression inserted in the code at the point of the call to the eval() function. Consider the following simple example:

```
function f {
   var x = 1                                  // local variable
   var y = System.eval ("x + 1")              // evaluate
}
```

The one and only argument to the *eval*() function is a string. The string is an expression that can contain anything that could be used at that point in the program if the expression was written directly. The above example is exactly the same as:

```
function f {
   var x = 1                                  // local variable
   var y = x + 1                              // evaluate
}
```

Although the above example doesn't show it, the expression can be much more complex and may be generated by the program or loaded from some other location. For example, a simple calculator may be written as:

```
function calculate {                        // calculate expressions typed in
   while (!System.eof(stdin)) {             // until end of input
      var expr = ""                         // string to hold expression
      System.print ("Expression: ")         // ask user for expression
      System.flush (stdout)
      stdin -> expr                         // read the expression
      generic result = System.eval (expr)   // calculate result
      System.println ("result: " + result)  // print result
   }
}
```

This function reads from standard input until it is closed, reading strings and printing the results of the calculation performed on the strings. When run, the program might produce the following:

```
Expression: 1 + 2
result: 3
Expression: 3.1415927 / 2
result: 1.57079635
```

If the strings provided to the *eval*() function contain only numbers and operators on numbers, the result is obvious (as in the above example run). What happens if you give it identifiers? The answer is that any expression is valid, including those containing identifiers and function calls. For example, the program may have been given the following:

```
Expression: System.println ("hello world")
hello world
0
```

Any identifiers can be passed to *eval*() provided that those identifiers are valid at the point at which the eval() function is called. The first example showed this: the identifier 'x' was passed to *eval*().

14.1.1. Creating new variables

When the Aikido parser comes across an identifier representing a variable of which it has no knowledge it has a couple of choices: it could report an error saying that the variable is undeclared; or it could invent a new variable. Which path it chooses depends on what follows the identifier in the code sequence. If the identifier is followed by an assignment token (=) then it can be inferred that the programmer omitted the 'var' from a variable declaration (of course, this may be a false inference and it might just be that the programmer mistyped the name of a variable).

So, if the parser comes across an unknown identifier followed by an assignment token (= sign) then it creates the variable as if the programmer had declared it. This feature is very useful for creating new variables in a dynamically loaded expression. An assignment to a variable is an expression in Aikido so it is possible to do the following:

```
function f {
   System.eval ("r = 1")
}
```

This creates a variable called 'r' and assigns the integer value 1. The variable is created in the scope of the caller of the *eval* function. This means that the variable exists for as long as the caller of the *eval* function exists. In the case of a function that calls *eval*, the variable will be destroyed when the function exits. In the case of eval being called in a class constructor, the variable will exist until the instance of the class is deleted.

This feature let's the simple calculator create variables that may be used as memory storage areas with no modification to the code:

```
Expression: m1 = 1234
result: 1234
Expression: m1 / 2
result: 617
```

14.2. Dynamic code loading

Another way to load code into a running program is to use the *System.load*() function, the more powerful cousin of *System.eval*(). Loading code into a running program is a very powerful but somewhat dangerous feature that should be used with caution.

The System.load() function allows any code to be loaded into the program while it is running. The code is taken from a stream or a vector and is in source code format. Typically, the code will be held in a file or be transmitted over a network.

Consider a file containing (just) the following code:

```
var myvar = 123
public var x = "hello"

public function print (v, stream) {
   ["(", v, ")\n"] -> stream
}
```

Say that this code is in the file 'code.aikido". The code may be loaded by the following calls:

```
var str = System.openin ("code.aikido")
var p = System.load (str)
System.close (str)
```

So what does this do? The code is loaded into a new class all of its own and created specially for it by the interpreter. This class has the name "<anon>" (by default, but you can specify something else). The class is automatically instantiated and the instance returned as the return value of the System.load() call.

Once loaded, the instance of the class may be used just as if it was coded as part of the program. For example:

```
p.x += " world"                    // append to variable x
p.print ("wonderful", stderr)      // call print function
var q = p.myvar                    // oops, ERROR, private variable
```

The System.load() function parses the code as if it was coded inline at the point of call of the function. This means that any variables that are in scope at that point are available to the dynamic code. Since the

code is loaded into an empty class any variables you declare are put in that class and do not clash with any existing variables.

There is no restriction on the code that can be used in dynamically loaded program. You can do anything in the file that could be coded inline. This includes extending existing blocks. If you extend a block it will take immediate effect. Care must be taken with this feature in the presence of threads. If you extend a block that is being used by a thread strange things might happen.

Chapter 15. System Library

Like most programming languages, Aikido does not have a built-in function for everything that the user might want to do. There are, however, a set of functions that are built on top of the basic features of the language and provided with the language interpreter. It is not the intention to provide an all-encompassing library of functions and classes that will suit every purpose. Rather, the system library provides a set of simple objects and functions that can be used as building blocks for larger, more functional libraries.

The Aikido system library is:

- Small
  No major learning task involved in getting to know the library contents. One of the big obstacles in learning a language is memorizing all those classes and functions conveniently provided by the language authors.

- Efficient
  The system library is used often by the programs. It therefore must be as efficient as possible. This is not a purist library where everything is written in Aikido. If something can be done more efficiently in C++ and is a critical library component, it is written in C++.

- Functional
  Although small, the Aikido library provides enough functionality to be useful. Libraries can go over the top and provide everything that someone might want in the future. This is a daunting, if not impossible, task. The other extreme is also possible, where the library is of no use to anyone. The Aikido library packages and classes do one thing well and try not to anticipate every forthcoming use.

- Extensible
  One of the nice features of Aikido is the ability to extend an object. This can be put to good use in the context of the library. The library does not provide everything that will eventually be needed. The objects in the library is extensible so the user can easily provide additional functionality for his own program, or provide an additional library for use by other programmers.

This chapter will provide a reference for the facilities contained in the Aikido system library.

15.1. Summary

The system library consists of the following packages and classes:

Name	Import File	Purpose
package System	<none>	Lowest level system functions. Always available without import
package Network	net.aikido	Network streams and associated functions
package ctype	ctype.aikido	Character type functions. Provides functions to interrogate and manipulate characters
package Math	math.aikido	Mathematics functions and constants
class String	string.aikido	A string object providing additional functionality on top of built-in string value
class Streambuffer	streambuffer.aikido	Buffering facility for streams
class Properties	properties.aikido	Facility for handling sets of properties
monitor Vector	vector.aikido	Vector object providing additional functionality above

		built-in type. Also supports mutual exclusion lock
monitor Map	map.aikido	Map object providing additional functionality above built-in type. Also supports mutual exclusion lock
monitor List	list.aikido	Linked list of values
monitor Stack	stack.aikido	LIFO stack of values
monitor Queue	queue.aikido	FIFO queue of values
monitor Hashtable	hashtable.aikido	Map making use of hash functions to provide faster lookup
interface Container	container.aikido	An interface for all container monitors
package Security	security.aikido	Encryption facilities
package Filename	filename.aikido	File name manipulation facilities
package Registry	registry.aikido	Windows® only. Registry access factilities
package GTK	gtk.aikido	GTK+ interface for GUI
package GDK	gdk.aikido	Low level GDK interface for GUI

The above table lists the contents of the system library in 3 columns. The first is the name of the class or package that is provided. The second is the file in which it is provided. This file must be imported into the program in order to use the facility. The third is a brief description of the purpose of the package or class.

In order to import a library facility you must use the **import** statement somewhere in your program. Once a facility has been imported it is available from that point on in the program. It is imported at the top scope level so if you import inside a class it will still be made available to the rest of the program. When importing you should drop the '.aikido' from the end of the file name. For example, to import the class Vector, you would type:

```
import vector
```

The System package is always available as it is automatically imported (from the file system.aikido) when the interpreter starts.

All the files are provided in the file Aikido.zip (a zip archive). They are always in source code form.

15.2. Extending the library

There are a number of ways to extend the library. It would be naïve of me to presume that this library as it stands will suffice all the requirements of any programmer. Indeed it is not intended to do so. It is my intention that the library be added to and extended by anyone using it as time progresses and more requirements are placed on it.

To extend the library you can either provide additional packages and classes, or you can extend the existing packages and classes to provide more functions. The functions that you provide may either be written in Aikido or may be in a native language (primarily C++, but maybe others are possible).

For example, suppose you need a way to convert a string to upper case. This is not supplied by the class String (although maybe it should be since I've used it a number of times in this book). You can do this in a number of ways, but let's decide to extend the String class by adding a *toUpperCase*() function. Let's see how to do this:

```
import string
import ctype

extend String {
```

```
    // convert string contents to upper case
    public function toUpperCase() {
        for (var ch = 0 ; ch < sizeof (value)  ; ch++ {
            value[ch] = ctype.toupper (value[ch])
        }
    }
}
```

Notice that we decided to overwrite the value in the string with the upper case version of itself. We could also have chosen to make a new string and return it from the *toUpperCase*() function.

We first needed to import the *String* class and the *ctype* package. The *ctype* package provides the *toupper*() function that we need to convert a character to upper case.

As we have extended the String class we can now call the toUpperCase() function at any time. See [1] for an alternative implementation.

This might be a nice feature to put in a library for your project, so it would be a good idea to save it to a disk and provide it for import by other people. You can do this by putting the above code in a file with the name, say, "mylib.aikido".

If you put the file mylib.aikido somewhere that is accessible by other people, they can then say:

```
import mylib
```

And get all the additional functionality you provide.

15.3. System Package

The System package contains a set of functions, classes and constants that deal with many operating system interfaces. Functions to access files, read directories, etc are provided.

Name	Arguments	Purpose
function println	value, stream	Print to stream, appending line-feed character. Stream defaults to output
function print	value, stream	Print to stream. Default output
function printf	format,...	C style printf. Only to output
package StreamAttributes		Names of attributes for setStreamAttribute
function clone	value, recurse	Clone a value, optionally recursing
function fill	object, value, start, end	Fill the object with the value from index start to end. Works with vectors or strings
function resize	object, size	Change the size of a string or vector
function find	object, value, index	Find a value. The search starts from the index parameter (default is 0). If the object is a vector, bytevector or string, the index of the value found is returned. If the object is a block or an object, 0 is returned if not found, or the block if it found.
function rfind	object, value.	Find a value, searching backwards rather

[1] This function can be done using the System.transform() function described in section 15.3.2

	index	then forwards.
function split	object, separator	Split a value into parts. Returns vector of parts. If value is block, gets all the variables.
function transform	value, function	Transform a value by applying a function to each element of it. A new value is returned and the old value is not modified.
function replace	val, find, repl, all	Replace parts of a value
function trim	value	Trim white space off value
function hash	val	Make hash code for value
function open	filename, mode	Open a file and returns stream
function openin	filename	Open a file for input and return stream
function openout	filename	Open a file for output and return stream
function openup	filename	Open a file for update and return stream
function openfd	file descriptor	Attach a stream to an existing file descriptor
function close	stream	Close a stream
function select	stream, timeout	Check a stream for ready input
function eof	stream	Check a stream for end-of-file condition
function flush	stream	Flush a stream's buffers
function getchar	stream	Read a single character from a stream
function getbuffer	stream	Read all the characters from a stream
function availableChars	stream	How many characters are available in a stream
function setStreamAttributes	stream, attr, value	Set a stream attribute
function error	string	Signal an error condition
function format	format, ...	Format a *printf* style string
function vformat	format, vector	Format using vector instead of args
function rewind	stream	Rewind a stream
function getStackTrace		Find out where you are in a program
function system	command, env, dir	Execute a command on the operating system, returning a vector of lines containing the output of the command. The env and dir parameters allow the environment and current directory to be set for the command. They are optional
function exec	command, outstream, errstream, env, dir	Execute a command and write the output to the given streams. All parameters are optional except the command. The default is to write to standard output and standard error
function getSystemStatus	none	Get the exit status of the previous system command
function pipe	command, redirectStderr, env, dir	Spawn a new process to run the command and return an open stream connected to the standard input and output of the command. The redirectStderr parameter is optional and defaults to false. If it is true then standard error will be sent to the stream. The env and dir parameters are as in system() and are optional.
function pipeclose	stream	Close the stream associated with a pipe created using the pipe() function. This also

		waits for the process to terminate and returns the status of the process.
function eval	expression	Evaluate an expression
function seek	stream, offset, whence	Move current position in a stream
function abort		Abort the program
function exit	code	Exit the program gracefully
function stat	filename	Get statistics on a file
function readdir	dirname	Read the contents of a directory into a vector of strings
function chdir	dirname	change directory
function getwd		return current working directory
function rand		Generate a random 32 bit number
function srand	seed	Set the seed of the random number generator
function sort	vector	Sort a vector
function bsearch	vector, value	Search a vector using binary search
function getenv	variable	Get the value of an environment variable
function setenv	variable, value	Set the value of an environment variable
function glob	pattern	Expand pattern as a set of filenames and return a vector of strings
class StackFrame		Record of current location in program
function printStackTrace	stack trace	Print stack trace information to stdout
function whereami		Print current location to stdout
class Regex		Regular expression result
function readfile	filename	Read the contents of a text file into a vector of lines
function time		Get the current time in microseconds from Jan 1, 1970 GMT
function date		Get the current date (in a Date object) from Jan 1, 1970 GMT
class Date		Holds the components of a date
class Stat		Holds file statistics
class User		Holds information on a user
class Pair		Holds information for map iteration
function redirectStream	stream, value	Redirect a stream
package OpenMode		Constants for open() function mode parameter
function load	value, name	Dynamically load code
class Exception		Generic exception
class FileException		File based exception
class ParameterException		Function parameter exception
function kill	pid, signal	Send signal to process
function sigset	signal, handler	Set up a signal handler for the given signal
function sighold	signal	Stop delivery of the given signal
function sigrelse	signal	Allow delivery of the given signal
function sigignore	signal	Ignore signal
function sigpause	signal	Allow delivery of signal and wait until one is received
package Signals		Signal numbers
function getlimit	name	Get value of OS limit variable

function setlimit	name, value	Set the value of an OS limit variable
function getUser	username	Get details of a user
function malloc	size	Allocate raw memory
function poke	addr, value,size	Write to raw memory
function peek	addr, size	Read from raw memory
function bitsToReal	bits	Convert bit sequence to real value
function loadLibrary	filename	Load dynamic library
class RegexMatch		Regular expression match object
function match	str, expr	Match using regular expression
function append	val, x	Append x onto val

Note: there is an automatic 'using' statement done for the System package. This means that it is not necessary to prefix all system facilities with 'System.". This documentation uses the full name of the facility in order to disambiguate it for the reader.

For example:

```
System.println ("hello world")

and:

println ("Hello world")
```

will call the same function. Which one to use is mostly a matter of taste, in the absence of any other requirements.

15.3.1. Output of values

There are a couple of simple functions provided to wrap the stream output facilities of the language. These functions are:

- **println (value = "", stream = output)**
  Print the value to the stream followed by a line-feed character. The default for the stream is output, which will map to standard output unless it has been redirected. The println function is defined as:

  ```
  function println (value, stream=output) {
     value -> stream
     '\n' -> stream
  }
  ```

  If the first parameter is omitted the function simply prints a blank line

- **print (value, stream = output)**
  Like the println function, this prints to the output stream. It does not append a line-feed character, however.
- **printf (format, ...)**
  This is very similar to the standard C printf function. It is provided for those who dislike the style of stream input/output and would prefer to do it using format characters. The output only goes to the output stream (or wherever it is redirected to). The function uses the '*vformat'* function to do its work. In fact, the function is defined as:

  ```
  function printf (f: string, ...) {
     print (vformat (f, args))
  ```

}

Example:

printf ("the average is %g\n", total/count)

If you want to put the result in a string or another stream, use the *format* function directly.

15.3.2. Operations on values

The following operations on values are available.

- **clone (object, recurse)**
  Make a copy of an object. If the recurse parameter is true then the subobjects of the object are also copied. The object may be any value, from an instance of a class to a vector, to a simple integer. The new value is returned
- **fill (object, value, start, end)**
  Fill an object with a value. The only objects that can be filled are vectors and strings. The value is inserted in all the elements from start to end indices. The end index is not included.
- **resize (object, size)**
  Change the size of a vector or string to that given.
- **sort (vec)**
  Sort a vector in ascending order of value. A new vector is created and sorted by this operation. The old vector is not modified.
- **bsearch (vec, val)**
  Search a sorted vector for the given value. Returns true if the value is found, false otherwise.
- **find (obj, val, index = 0)**
  Search an object for the given value. The search is linear and this works with vectors, maps, strings, blocks and objects. The index of the found value is returned for a vector and string. For a map an object of type Pair is returned. If the find() function is applied to an object the object is searched for a function named "find" taking one parameter. If this function is present, it is called. If it is absent, the type of the object is searched. For example, to find a block in the main program use: System.find (main, "blockname")
- **rfind (obj, val, index = sizeof(obj) - 1)**
  Search an object for the given value using a reverse search where appropriate The search is linear and this works with vectors, maps, strings, blocks and objects. The index of the found value is returned for a vector and string. For a map an object of type Pair is returned. If the find() function is applied to an object the object is searched for a function named "find" taking one parameter. If this function is present, it is called. If it is absent, the type of the object is searched. For example, to find a block in the main program use: System.rfind (main, "blockname")
- **split (obj, sep)**
  Split a value into parts. The 'sep' parameter (separator) is used to delimit the parts. The object may be a vector, string, object or block. If it is a block the separator is ignored and the list of components of the block are returned. The return value is a vector of the parts of the value. For a string, the separator may be a character or a string. Like find(), if the split() function is applied to an object, and the object contains a function called "split" (with one parameter), then that function is called. If the function is not defined then the split() function is applied to the type of the object.
- **transform (value, func)**
  Transform a value by applying a function to each part of it. What a 'part' means depends on the type of the value being transformed. If the value consists of multiple parts (string, vector or map), the function is applied to each element is turn. The result is a value of the same type as that passed but transformed by the function. A closure is created by the interpreter when the function is passed, therefore any function taking one argument and returning a value can be used.
  - **trim (value)**

Trim the white space off a value. If the value is a string the space and tab characters are trimmed off each end. If the value is a vector the values of type none are trimmed off. If the type is a bytevector, all zeroes are trimmed off. A new value is returned, the parameter is not modified by this operation.

- **replace (value, find, repl, all)**
  Replace parts of a value with other values. The first parameter is the value to operate upon. It is not modified by the operation. The second parameter is the value to find in the first value and replace with the third parameter. The function can replace either the first or all of the instances depending on the value of the final parameter.

  For example:

  var n = System.replace ("hello", 'l', 'L', true) // result: heLLo

  The type of the first parameter is not limited to strings: vectors and bytevectors can also be used.
- **hash (value)**
  Calculate a hash code for the value passed. The hash code is an integer that will be the same for identical values. Any type is acceptable as the value. If the value is an object, the function 'hash()' in the object will be called if it exists.
- **append (val, x)**
  Append the value x onto the value val.

15.3.3. Operations on files and streams

A substantial portion of the functions in the system library is devoted to the handling of files. When a Aikido program is running, all access to files it though a stream. There needs to be a way to create the stream and control it. The system library provides this ability. The following facilities are provided:

- **open (file, mode)**
  Open a named file. The mode parameter specifies how the file will be opened. Modes are in the OpenMode package included with the System package. If the file can be opened a stream attached to the file is returned.
- **openin (file)**
  Open a file for input. The file must exist and must be accessible. A stream attached to the file is returned.
- **openout (file)**
  Open a file for output. The file is created or truncated if it already exists. A stream attached to the file is returned.
- **openup (file)**
  Open a file for update. The file is created if it doesn't exist. If it already exists, the file pointer is set to the end of the file. A stream is returned.
- **openfd (fd)**
  Open a file to a raw file descriptor. This creates a stream attached to an Operating System file descriptor that is already open.
- **close (stream)**
  Close the given stream.
- **select (stream, timeout)**
  Check the given stream for available data. Returns true if there is data available for reading from the stream. The timeout parameter specifies how long to wait for data to arrive before giving up and returning false. A value of 0 means no timeout.
- **eof (stream)**
  Check for end-of-file condition on the given stream. This means that the stream has been closed. Returns true if the end-of-file condition has been met.
- **flush (stream)**
  Flush any buffered data to the device on an output stream.

- **getchar (stream)**
  Read a single character (byte) from an input stream.
- **getbuffer (stream)**
  Read all the buffered characters from an input stream. Creates a string containing all the bytes and returns that.
- **availableChars (stream)**
  Obtain a count of the number of characters that are available for reading from an input stream.
- **setStreamAttribute (stream, attr, value)**
  Set an attribute of a stream. The following attributes are supported:
  - StreamAttributes.BUFFERSIZE – integer: the size of the buffer, attached to the stream (bytes)
  - StreamAttributes.MODE – integer: 0: line mode, 1: character mode. Determines whether the stream is treated as containing lines or characters.
  - StreamAttributes.AUTOFLUSH – true or false. If true, the stream will be automatically flushed when any data is sent to it.
- **rewind (stream)**
  Set the current position back to the start of the stream. The stream must support the seek operation. That is, it must be a file or a device that has a current position and provides random access.
- **seek (stream, offset, whence)**
  Seek to a position in the stream. The values for 'whence' are SEEK_SET, SEEK_CUR and SEEK_END meaning to seek to an absolute position, relative from the current position or relative from the end of the stream respectively.
- **stat (file)**
  Get statistics on a named file. An object of type System.Stat is returned if the file exists and 'null' if it does not or is not accessible.
- **readdir (dirname)**
  Read the named directory and return a vector of strings. Each entry in the vector is the name of a file in the directory. All files except '.' and '..' are returned.
- **readfile (filename)**
  Read all the lines in a file into a vector of strings. Returns the vector.
- **chdir (dirname)**
  Change working directory to that given as the string argument.
- **getwd()**
  Read the current working directory
- **redirectStream (stream, newvalue)**
  Redirect a stream. This simply assigns the new value to the stream. No checking is done to ensure that either parameter is a streamable object.
- **pipeclose (stream)**
  Close a stream associated with a pipe (created by the pipe() function) and wait for the process to terminate. Return the status of the process when it terminates.
- **loadLibrary (filename)**
  Load the named file as a shared library.
- **glob (pattern)**
  Expand the pattern to a set of filenames. The pattern may contain wildcard characters. This is similar to the use of wildcard characters in the regular shells available on operating systems.

15.3.4. Date and time

These functions provide ways to obtain the current date and time. Times are measured in microseconds since Jan 1, 1970 at midnight GMT.

- **time()**
  Read the current time of day. This is the number of microseconds since the epoch (Jan 1 1970).

- **date()**
  Read the current date. This returns an instance of the class System.Date.
- **gmdate()**
  Read the current date as a GMT date. An instance of System.Date is returned
- **makedate(time)**
  Make a Date object out of a time value in local time. Return the System.Date object.
- **makegmdate (time)**
  Make a GMT Date object out of a time value. Return the System.Date() object.
- **parsedate (datestring, date)**
  Given a string representation of a date, convert it to a System.Date() object. A wide range of formats are supported. See the discussion of date formats below for details. The Date object must be preallocated and passed to this function. The object will be filled in by the function.

The System.Date() class holds the information about a date in time. It can be constructed by either calling one of the date functions (listed above) or by creating an instance of the System.Date() class and passing a date in a string form.

The System.Date() class is defined as:

```
public class Date (s : string = "") {
public:
    var sec = 0                 // seconds after the minute - [0, 61]
    var min = 0                 // minutes after the hour - [0, 59]
    var hour = 0                // hour since midnight - [0, 23]
    var mday = 1                // day of the month - [1, 31]
    var mon = 0                 // months since January - [0, 11]
    var year = 0                // years since 1900
    var wday = 0                // days since Sunday - [0, 6]
    var yday = 0                // days since January 1 - [0, 365]
    var isdst = 0               // flag for alternate daylight savings time
    var tzdiff = 0              // difference in seconds from local timezone to GMT
    var tz = ""                 // local time zone name

    function format (f)         // format date with given format
    function toString           // format date in default format
    function normalize          // normalize the fields of the date
    operator -> (stream, isout) // stream output
    operator + (secs)           // move forward by number of seconds
    operator - (secs)           // move backwards by number of seconds
    operator == (otherdate)
    operator != (d)
    operator < (otherdate)
    operator > (otherdate)
    operator <= (d)
    operator >= (d)
    operator in (d1, d2)
    static function makeDate (time)      // factory method to make a date from a time
    static function makeUTCDate (time)   // make a UTC date from a time
}
```

15.3.4.1. Date formats

The constructor for the System.Date() class can take a string parameter representing the date to be constructed. This is passed to the System.parsedate() function to convert it to a System.Date() object. The string is an English language representation of the date to be converted. A wide range of formats are available.

The date string consists of a series of ordered tokens. Each token is part of the date to be produced. If the token is a number it is taken as either a day, hour, minute or second depending on what the following character is. If the token is a sequence of letters, it must be from the following set of sequences:

sunday	**monday**	**tuesday**	**wednesday**	**thursday**
friday	**saturday**	**january**	**february**	**march**
april	**may**	**june**	**july**	**august**
september	**october**	**november**	**december**	**am**
pm	**gmt**	**utc**		

The case of the sequence is ignored. Also, abbreviated sequences are allowed, where the abbreviation is the initial sequence of letters in the sequence. In other words, you can abbreviate each name by just using its initial letters ('sep' for 'september' for example).

The general form of a date is:

<month> <day> <year> <hour>:<min>:<sec> <am/pm> <timezone>

The parser allows these in any order, it looks for the next character in the sequence to determine what the meaning of a field is. For example, if a number is found and it is followed by a colon character it is either an hour or a minute. The choice depends on whether the hour has already been set or not. The hour can be in 12 or 24 hour format.

If a field is omitted, it is filled in with the appropriate value from the current date. This means, that if you omit the time from the string, it will be filled in with the current time.

Time zones are specified by either GMT or UTC plus or minus a delta. The delta is specified as 2 fields representing the hours and minutes. For example, "GMT+1230" would be 12 hours 30 minutes ahead of GMT (somewhere in India I think). A negative delta is west of GMT (Greenwich is in London, UK, in case you were unaware). There is no direct translation of timezone names to the delta (currently).

The following are examples of valid dates (current date is "Thu Nov 14 13:14:31 PST 2002"):

```
"Wednesday Oct 23"      // at 1:14:31PM
"Jan 23 1981 10:0 am"
"13:30:1"               // Thursday November 14 13:30:1 PST 2002
"nov 15"                        // tomorrow at this time
"July 13 GMT-0800"      // July 13 13:14:31 PST 2002
```

15.3.4.2. Date manipulations and operations

Once an instance of System.Date() exists, is can be manipulated to produce other dates and operated upon to compare, format, normalize, etc. A date can be made changed to another date by adding or subtracting a number of seconds from the object (overloaded operators). For example:

```
var today= System.date()
var tomorrow = today + (60 * 60 * 24)
```

The individual fields of a date can be manipulated by adding and subtracting numbers from the fields. The above example could be recoded as:

```
today.mday--
today.normalize()                    // normalize the date
```

In this case we have to call the 'normalize()' function was we might have made the fields internally consistent (if mday was 1 it will now be 0 and thus out of range). You can perform this operation on any of the fields of the date object (they are all public).

Dates can be compared with other dates by use of the normal relational operators. Dates are said to be ordered in time. This means that if one date is less than another, the one that is less represents a time before the one that is greater. For example:

```
var today = System.date()
var yesterday = today – (60*24*24)

var x = today > yesterday                    // true
var y = yesterday > today                    // false
```

Because a date includes not only the day/month/year value, but also the time value, you have to be careful when comparing them. In the above example, the value of 'yesterday' is actually the value of the current time on the day before today. This is especially important when using the equality operator (==) for comparing dates. If you want to check if the 2 dates are on the same day, remember to zero the time fields before comparing.

Dates can be formatted for output. There are 2 ways to format a date:

1.format (fmt)
2.toString()

Both of these functions return a string. The simpler is the toString() function. It formats the date using the default format. This format is the same as that produced by the *date* command on UNIX®.

The more comprehensive of the formatting functions is the format() function. This takes a string that specifies how to format the date into a string. This is a printf() like format specifier using % characters to control the format. For full information see the manual page on a UNIX® system for the function *strftime:*

```
% man strftime
```

The default format for the toString() function is: "%a %b %e %T %Z %Y"

15.3.5. Miscellaneous operations

- **getStackTrace()**
  Get a trace of the current location in the program. Returns a vector of System.StackFrame objects. The first element in the vector is the topmost location on the stack.
- **printStackTrace (trace)**
  Print the contents of a stack trace obtained by getStackTrace to the standard output
- **whereami()**
  Print the current stack trace to standard output
- **abort()**
  Abort the current process. This will probably produce a core dump.
- **exit (code)**
  Exit from the current process gracefully. The code is returned as the status of the process.
- **rand()**
  Generate a 32-bit pseudo random number.
- **srand (seed)**
  Set the seed of the random number generator. This makes the random sequence predictable.
- **getenv (variable)**
  Get the value of an environment variable as a string. The value 'null' is returned if the variable does not exist.
- **setenv (variable, value)**
  Set the value of an environment variable to the string value given.
- **error (str)**
  Report an error in the program. This writes the error to the standard error stream and increments an internal error count.
- **format (str, ...)**
  Format a C printf style string and return the result as a string. Supports all printf arguments with the addition of %b for binary output.
- **vformat (str, args)**
  Format a C printf style string and return result as a string. Like format, except takes a vector of arguments rather than a variable argument list. This can be used from within a variable argument list function.
- **getUser (name)**
  Given a user name, look up the operating system for the user. If the user is found, an instance of the class *User* will be returned containing information about that user. The value *null* is returned if the user does not exist.
- **getlimit (limitname)**
  Read the value of an operating system limit variable. The names of the limits are:
  - cputime
  - filesize
  - datasize
  - stacksize
  - coredumpsize
  - descriptors
  - memorysize

- **setlimit (limitname, value)**
  Set the value of an operating system limit. The names of the limits are as in getlimit above.

15.3.6. Executing Operating System commands

There are 3 methods of executing commands on the native operating system:

- **system (command, env = [], dir = "")**
- **exec (command, outstream = output, errstream = stderr, env = [], dir = "")**

- **pipe (command, redirectStderr = false, env = [], dir = "")**
- **getSystemStatus()**

All of them take a string representing the command to execute. The command is passed to the system shell (/bin/sh on UNIX®) and may contain multiple commands separated by semicolons. The command is executed using the *system*() function on UNIX®. See the UNIX® manual, section 3s for information on the *system*() function.

The command executes in its own process. This process inherits the environment of the calling process (the Aikido interpreter). The parameters *env* and *dir* in both the functions allow the environment of the new process to be modified without affecting the Aikido interpreter. The modifications include adding and overriding environment variables and changing the current directory.

The *env* parameter is optional. If it exists, it should be a vector of strings. Each string consists of the name of an environment variable and a value and is of the form:

```
"name=value"
```

This string is added directly to the environment variable list of the process spawned for the system command.

The *dir* parameter is an optional string that can contain the name of a directory to which the current directory of the process is set.

The easiest to use is the *system*() function. It simply takes the command to execute and the *env* and *dir* parameters (optional). The command is executed and any output from the execution (standard output) is collected into a vector of strings. This vector is returned from the function. The standard error stream of the command is not redirected and appears on the terminal. Each of the strings in the returned vector is terminated by a newline character. Consider the following example:

```
var files = System.system ("cd /usr/dist ; ls")      // 2 commands
files -> stdout                                      // send output to stdout
```

The command executed consists of 2 commands (although it could have been done with a simple command). Notice that they are separated by a semicolon character. The return value is assigned to the variable *files*, and this is streamed to standard output. The exit status of the system command may be obtained by calling the function System.getSystemStatus() immediately after the system call.

The *exec*() function is more flexible. Like the *system*() function it takes a command to execute and the optional *env* and *dir* parameters, but is operates differently. With the *system*() function, the output of the command is buffered up into a vector and returned from the function. This has the effect of not allowing the output to be seen until the command completes. If there is a lot of output it will use a lot of memory to hold the vector. This may not be desirable.

The *exec*() function takes 2 additional (optional) parameters called *outstream* and *errstream*. These specify the locations to which the standard output and standard error (respectively) streams of the command. The output from the command is sent to streams as the command executes. The *outstream* and *errstream* parameters may be a stream, vector or an object that provides an overload of the stream operator. If they point to the same stream, the output to the standard output will always appear before the output to standard error. Consider the following:

```
System.exec ("ls")                   // perform a list of the current dir
var files = []
System.exec ("ls", files)            // list files into vector
```

In the first example, the output from the command is sent to the output stream of the thread running the command. In the second example, the output is sent to a vector.

The *mode* (StreamAttributes.MODE) of the stream determines how the output from the command is written to it. If the mode has value 0, then the output of the command is buffered up into lines and each line is written to the stream as a string. If the mode is 1, then each character is written to the stream as it is read.

The return value of the *exec*() function is the exit status of the command. Generally commands will exit with status 0 for success and anything else for an error. If the command terminates with an error code then the value of the operating system variable 'errno' is returned. If the command terminated with a zero code, then 0 is returned. If the command terminated due to other conditions (signal, or other cause) then the return value is that returned from the operating system function '*wait*()'. This value is a series of fields in one integer. The format of the integer and its associated flags may be found in the file <sys/wait.h> on a UNIX® system. See the manual pages for *wait*(2) and *wstat*(5) for full information.

The *pipe*() function is the most flexible of those available. It spawns a new process to run the command and opens a stream to the process. The stream is connected to the standard input and standard output of the process (and standard error if requested). The *pipe*() functions returns this stream as its return value. The caller can then use this stream to send and receive data. The *pipe*() function does not wait for the process to terminate before returning (unlike the system() and exec() functions). Consider the following example:

```
var str = System.pipe ("cat")
```

Here the stream 'str' is attached to the standard input and standard output of the process running the command 'cat'. Writing to the stream will cause the process to receive data from its standard input, while reading will read output produced by the process.

Unlike the *system*() and *exec*() functions, the *pipe*() function does not wait for the process to terminate. The process runs in parallel with the program that called the *pipe*() function.

Note: Since the *pipe*() function does not wait for the process to terminate, you must close the stream returned by the function using the *pipeclose*() function. This will wait for the process to terminate after closing the stream. The process will see an end-of-file condition on its standard input and should exit. The exit status will be returned from the *pipeclose*() function. Failure to call *pipeclose*() will result in zombie processes (on UNIX®) being created.

15.3.7. Dynamic loading operations

The following dynamic loading functions are available:

- **eval (expression)**
  Evaluate the expression in the current scope context. The expression is a string. The result of the expression is returned.
- **load (source, name = "<anon>")**
  Load source code from a stream, string or vector. The code is placed in a new package whose name is provided as the second parameter. The default is "<anon>". The package created is returned.

15.3.8. Signal handling

When running a Aikido program in a real environment (meaning one that is deployed and must behave properly under adverse conditions), the program must be able to deal with signals from the operating system. Signals are the operating system's way of telling a program that something external to it has

happened and needs attention. For example, if the user hits ctrl.-C when the program is running, the operating system sends a signal (SIGINT in this case) to the program to tell it that the user has requested an interrupt of its operation.

Signals are numbered. Each signal means something different to the program. The SIGINT signal is number 2. The numbers are assigned by the operating system but there is a standard to the numbering scheme used.

When a program gets a signal sent to it, it has a number of choices.

1. It may not even see it and the default signal handler is invoked by the operating system
2. It may say "do not disturb" and thus the signal will be ignored
3. It may catch the signal and invoke a function to handle it

The Aikido System package contains functions that provide control over the handling of signals. The names of the functions are identical to those provided by the Solaris Operating Environment, as are the names of the signals themselves.

The main signal manipulation function is *System.sigset*(). This takes 2 parameters:

1. The number of a signal
2. The signal "disposition"

The signal number is one of the constants in the *System.Signals* package. There is one constant for each signal provided by the operating system. The *disposition* is either a function or an integer. If it is a function then that function is called when the given signal is sent to the process. If it is an integer, it may be one of:

- System.SIG_DFL - default handler
- System.SIG_IGN - ignore signal
- System.SIG_HOLD - don't deliver signal
- any other integer value - a native signal handler function

The default handler for a signal is provided by the operating system and is different depending on the signal number. Some of the signals cause the program to exit (SIGINT for example), others cause a core dump to be written to the disk, some others are ignored. This is how the signals are handled in the absence of program control over them.

If a signal is ignored then it will never be sent to the program. The behavior of SIG_HOLD is different. A program can contain many threads running simultaneously. Since a signal is sent to a process as a whole, there will be, at any one time, a thread running on a CPU. When the signal is delivered, that thread will receive the signal and process it. SIG_HOLD causes the calling thread not to see the given signal. Other threads in the system can still see the signal.

A signal handler function is one that is called when the signal is received. There are restrictions on what function can be called for a signal. Since a signal is asynchronous, there is no way to know what the static chain of the signal handler function will be when it is called – it can happen at any time. Since the function must be able to interact with the rest of the program it must have a valid environment in which to execute. This means that the function must be in a known location.

The function called by a signal handler must take one parameter. It will be passed the number of the signal caught when it is invoked

The function can return a value, but that value will be ignored.

Consider the following trivial example of a ctrl-C counter:

```
function interrupt (sig) {
   static var count = 0
   System.println ("Signal " + sig + " caught: "  + count++)
   if (count > 10) {
      System.exit(0)
   }
}

System.sigset (System.Signals.SIGINT, interrupt)

// continue with rest of program
```

The function *interrupt* is a signal handler function. It is called when the user hits ctrl-C. It counts the number of interrupts and when it reaches 10 the program exits.

The return value of sigset() is either an integer or a function. If it is an integer it represents the disposition value of signal prior to setting it. This could be the address of a native signal handler function. This value may be used as a parameter to sigset() to reset the signal disposition to the previous value. If sigset() returns a function, then this function is the previous signal handler set by the program

The following signal manipulation functions are provided:

- **kill (pid, signal)**
  Send a signal to a process identified by the pid. The signal the number of the signal to send. The result is 0 if the signal has been sent, -1 otherwise. If the signal is 0 then it can be used to check for the validity of the process whose id is pid. See kill(2) on UNIX® for details.
- **sigset (signal, handler)**
  Set the disposition of the given signal to the handler. The handler may be either an integer or a function. If it is an integer it may be one of SIG_DFL, SIG_IGN or SIG_HOLD, or may be the value returned from a previous call to sigset. If it is a function, the function must be in the main package and must have one parameter. The function is called when the signal is received. The sigset function returns the previous value of the signal disposition. This may be either an integer or a function. If it is an integer it may represent the address of an native function that was previously set to handle the signal.
- **sighold (signal)**
  Add the given signal to the current thread's signal mask, thus preventing it from being delivered.
- **sigrelse (signal)**
  Remove the given signal from the current thread's signal mask, thus enabling its delivery.
- **sigignore (signal)**
  Prevent the signal being delivered to the process
- **sigpause (signal)**
  Remove the given signal from the thread's signal mask and wait until a signal is received.

15.3.9. Raw memory accesses

Sometimes it is necessary to be able to access raw memory from a Aikido program. Aikido provides 2 functions that should remind readers of the BASIC language facilities for doing this operation. The functions are *peek* and *poke*.

A raw memory address can be one of:

- An integer containing a valid machine address

- A *memory* value allocated using *System.malloc*()
- A *pointer* value created by adding to a *memory* value

The function System.malloc() allows the program to allocate a block of memory from the runtime heap. The parameter to the call is the number of bytes of memory to allocate. The result is a value of type *memory*. You can do a limited number of things to a raw *memory* value:

- Pass it to the peek and poke functions to read and write it
- Add an integer to it to produce a pointer

In the case of addition of an integer to a *memory*, a *pointer* is created. The interpreter makes sure that the *pointer* cannot be outside the memory address range set up by the *System.malloc*() function. Once a *pointer* is created, you can then add and subtract integers from it to form other *pointer* values. Again, you cannot go outside the range of the original *memory*.

The memory allocated by the *System.malloc*() function is reference counted and garbage collected like any other object in the system. When all the *memory* and *pointer* values referring to it are deleted, the memory itself is deleted. This means you do not have to take care of freeing the memory (there is, in fact, no free function).

The *peek* function allows raw memory to be read into either an integer or a bytevector depending on the size of the memory to be read.

The *poke* function allows raw memory to be written from either an integer or a bytevector, again depending on its size.

There is also a function called *bitsToReal* that takes a sequence of raw bits and treats it as a real number. This is necessary when there is a real number stored in raw memory and it needs to be read into a real value.

- **malloc (size)**
  Allocate a number of bytes off the program heap and return a *memory* value representing the start address of the memory. The memory is reference counted and subject to the normal garbage collection rules.
- **peek (address, size = 1)**
  Read memory. The address must be an integer representing a valid machine address. The size parameter is optional (default value is 1 byte) and represents the number of bytes to read. If the size is either 1, 2, 4 or 8, the value will be read into an integer. Correct alignment is assumed. If the size is any other value, the appropriate number of bytes will be read into a bytevector and that will be returned.
- **poke (address, value, size = 1)**
  Write memory. The address is a valid integral machine address. The value is typed appropriately for the size. If the size is 1, 2, 4 or 8 then the value must be an integral value. Correct alignment in memory is assumed. If the size is any other value, the value to be written must be in a bytevector.
- **bitsToReal (bits)**
  Convert a sequence or raw bits into a real value. This is different from casting an integer to real since there is no floating point conversion done.

If the address passed to *peek* and *poke* is an integer type, Aikido performs no checks whatsoever on the validity of the addresses for these functions. Crashes may occur for misuse. Care must be taken. For *memory* and *pointer* values, address validation is done.

15.3.10. Regular expression matching

The library provides a convenient wrapper for the standard regular expression matching facilities of the language. The wrapper consists of one function and a class:

```
class RegexMatch {
   public function expr (n)                    // retrieve expression n
   public funciton nExprs()                    // number of expressions
}
```

- **match (str, expr)**
  Look for the expression in the string and return a RegexMatch object instance describing the result. The value null is returned if there were no matches.

The RegexMatch class holds information about the regular expression match. A regular expression contains sub-expressions. The RegexMatch object holds all these sub-expressions and allows the user to query how many there are and get access to each one.

Consider the following example:

```
var str = "result:1234, value:3"
var m = System.match (str, "[^:]*:([0-9]*)[^:]*:([0-9]*)")

var n = m.nExprs()                  // 3 expressions matched
var result = m.expr(1)              // 1st sub-expression
var value = m.expr(2)               // 2nd sub-expression
```

The line noise in the System.match() function is a regular expression that means:

1. Any sequence of characters except :
2. A colon character
3. A sub-expression containing digits 0-9 only
4. Any sequence of characters except :
5. A colon character
6. A sub-expression containing digits 0-9 only

This will match the string passed and will return an object with 3 expressions in it. The first expression is the whole matched string. The second and third expressions are the sub-expressions.

Of course, this could all have been accomplished using the built-in features of the language, but this wrapper is perhaps a little more readable and encapsulates the more common usage of regular expressions: extraction of portions of a string.

15.3.11. Classes and packages

The System package provides the following set of classes and packages:

```
// An object representing a user of the computer as returned by getUser
class User {
   var name                         // username
   var uid                          // user id (integer)
   var gid                          // group id (integer)
   var fullname                     // user's full name (string)
   var dir                          // home directory
   var shell                        // shell executable
```

```
    var password                    // encrypted password
}

// A vector of StackFrame objects is returned by getStackTrace
class StackFrame {
    var filename                    // name of file
    var lineno                      // line number
    var block                       // name of block
}

// A vector of Regex objects is the value of a regular expression match
class Regex {
    var start                       // index of start of match
    var end                         // index of end of match
}

// A Date object is returned from the date function
class Date {
    var sec                         // seconds after minute – [0, 61]
    var min                         // minutes after hour – [0, 59]
    var hour                        // hour since midnight – [0, 23]
    var mday                        // day of month – [1, 31]
    var mon                         // months since January – [0, 11]
    var year                        // years since 1900
    var wday                        // days since Sunday – [0, 6]
    var yday                        // days since January 1 – [0, 365]
    var isdst                       // flag for alternate daylight savings time

    function toString()             // convert to string
    operator -> (stream, isout)     // stream to output stream
    function day()                  // get name of day
    function month()                // get name of month
}

// A Stat object is returned from the stat function
class Stat {
    var mode                        // file mode (see mknod (2))
    var inode                       // inode number
    var dev                         // id of device containing dir entry for file
    var rdev                        // id of device
    var nlink                       // number of links
    var uid                         // user id of owner
    var gid                         // group id of owner
    var size                        // size in bytes
    var atime                       // time of last access
    var mtime                       // time of last data modification
    var ctime                       // time of last status change
    var blksize                     // preferred IO block size
    var blocks                      // number of 512 byte blocks
}

// A Pair class is used in a foreach loop through a map
class Pair {
    generic first                   // key of map entry
    generic second                  // value of map entry
}
```

```
// Modes for the open() function
package OpenMode {
   const APPEND
   const BINARY
   const IN
   const OUT
   const TRUNC
   const ATEND
   const NOCREATE
   const NOREPLACE
}

// A generic exception
class Exception {
   function report (stream)                  // report exception to stream
   function printStackTrace (stream)         // show location of exception
   operator -> (stream, isout)               // stream exception to stream
   function toString()                       // convert to string
}

// Specific exception types
class FileException {                        // as Exception
   function getFileName()                    // get file name
}

class ParameterException {                   // as Exception
   function getFunction()                    // get function name
}

package Signals {
      const SIGHUP  = 1                      // hangup
      const SIGINT  = 2                      // interrupt (rubout)
      const SIGQUIT = 3                      // quit (ASCII FS)
      const SIGILL  = 4                      // illegal instruction (not reset when
caught)
      const SIGTRAP = 5                      // trace trap (not reset when caught)
      const SIGIOT  = 6                      // IOT instruction
      const SIGABRT = 6                      // used by abort, replace SIGIOT in the
future
      const SIGEMT  = 7                      // EMT instruction
      const SIGFPE  = 8                      // floating point exception
      const SIGKILL = 9                      // kill (cannot be caught or ignored)
      const SIGBUS  = 10                     // bus error
      const SIGSEGV = 11                     // segmentation violation
      const SIGSYS  = 12                     // bad argument to system call
      const SIGPIPE = 13                     // write on a pipe with no one to read it
      const SIGALRM = 14                     // alarm clock
      const SIGTERM = 15                     //software termination signal from kill
      const SIGUSR1 = 16                     // user defined signal 1
      const SIGUSR2 = 17                     // user defined signal 2
      const SIGCLD  = 18                     // child status change
      const SIGCHLD = 18                     // child status change alias (POSIX)
      const SIGPWR  = 19                     // power-fail restart
      const SIGWINCH = 20                    // window size change
```

```
        const SIGURG  = 21                  // urgent socket condition
        const SIGPOLL = 22                  // pollable event occured
        const SIGIO   = SIGPOLL             // socket I/O possible (SIGPOLL alias)
        const SIGSTOP = 23                  // stop (cannot be caught or ignored)
        const SIGTSTP = 24                  // user stop requested from tty
        const SIGCONT = 25                  // stopped process has been continued
        const SIGTTIN = 26                  // background tty read attempted
        const SIGTTOU = 27                  // background tty write attempted
        const SIGVTALRM = 28                // virtual timer expired
        const SIGPROF = 29                  // profiling timer expired
        const SIGXCPU = 30                  // exceeded cpu limit
        const SIGXFSZ = 31                  // exceeded file size limit
        const SIGWAITING = 32               // process's lwps are blocked
        const SIGLWP  = 33                  // special signal used by thread library
        const SIGFREEZE = 34                // special signal used by CPR
        const SIGTHAW = 35                  // special signal used by CPR
        const SIGCANCEL = 36                // thread cancellation signal used by
libthread
        const SIGLOST = 37                  // resource lost (eg, record-lock lost)
    }
```

15.3.12. System information variables

The System package contains a set of constants that contain information about the system on which the interpreter is running. These are read from the operating system. The variables are:

Variable	Type	Meaning
hostname	string	The name of the machine
username	string	The name of the current user
domainname	string	The name of the domain in which the machine resides
pid	integer	Process id of the process invoking the interpreter
ppid	integer	Process id of the parent process
pgrp	integer	Process group of the invoking process
ppgrp	integer	Process group of the parent of the invoking process
uid	integer	User identifier (number) for the invoking user
gid	integer	Group id (number) of the invoking user
operatingsystem	string	Common name of operating system
osinfo	string	Name and version of the operating system
machine	string	Type of the machine
architecture	string	Architecture name of machine
platform	string	Platform name
manufacturer	string	Manufacturer name
serialnumber	string	Serial number of machine
hostid	integer	Host identifier of machine
pagesize	integer	Size of a memory page on machine
numpages	integer	Number of physical pages of memory
numprocessors	integer	Number of online processors
fileSeparator	string	Pathname separator for OS (/ on UNIX®)
extensionSeparator	string	Separator for file suffix (. on UNIX®)

One noteworthy variable is System.operatingsystem. This contains the common name for the operating system and can be used to determine which operating system the program is running on. The currently supported operating systems are:

Operating System	System.operatingsystem value
Sun Microsystems Solaris	Solaris
Linux	Linux
Microsoft Windows	Windows
Mac OS X	Mac OS X

So, if necessary, a Aikido program can determine what OS it is running on and can act accordingly:

```
if (System.operatingsystem == "Windows") {
   throw "Sorry, Windows does not support this feature"
}
```

The System.osinfo variable can be used to determine the version of the operating system being run. The value of this differs depending on the OS. For example, on Solaris the System.osinfo variable may contain:

```
SunOS 5.7 Generic_106541-17
```

This shows the operating system name, version and patch level.

Your mileage may vary...

15.4. Network package

The Network package contains facilities to allow the use of network streams in Aikido. A network stream is a standard stream that is connected through the network to another machine. Aikido's network facilities are described in section 9.5. To use the network package you need to import the file 'net.aikido':

```
import net
```

Then you can call the functions through the package name (Network). For example:

```
var addr = Network.lookupName ("mycorp.co.ca")
```

This library provides the following facilities.

- **open (address, port)**
  Open an active TCP network stream to a server. The address is one of:
  1. integer IP address
  2. IP address in n.n.n.n format. This is a string. Example: "192.129.1.1"
  3. Host name. Also a string. Example: "www.acorn.co.uk"

  The port is a TCP port number. The return value is a stream open for reading and writing and connected to a machine over the network.
- **openServer (address, port, type)**
  Open a passive network stream. This is one that may be connected to by another machine. The parameters are the same as the 'open' call. The type is either Network.TCP or Network.UDP. The return value is a 'socket'. This is a number that may be passed to the 'accept' 'send' or 'receive' calls.
- **openSocket()**
  Open a UDP socket for sending or receiving UDP datagrams. The socket is returned.

- **lookupName (name)**
  Given the name of a machine on the network, consult a naming service to translate the name into an IP address. The return value is an integer containing the IP address. An exception is thrown if the name doesn't exist.
- **lookupAddress (addr)**
  Given an integral IP address, consult the naming service to convert it into a hostname. The hostname is returned as a string.
- **accept (socket)**
  For a TCP type stream, this waits for an incoming connection to a socket. It blocks until such a connection is made and returns a stream open for reading and writing.
- **send (socket, address, port, data)**
  Send a UDP datagram to the given address and port. The data must be a bytevector or a string (or can be cast to a string). The socket is one created by openServer() or openSocket().
- **receive (socket, var address, var port, maxbuffer = 4096)**
  Receive a datagram from the given socket. The reference parameters 'address' and 'port' are set to the IP address and UDP port number of the sender of the datagram. The data received is returned as a bytevector. The optional 'maxbuffer' parameter specifies the maximum size of the datagram we will accept.
- **peek (socket, var address, var port, maxbuffer = 4096)**
  Like receive except that the data is not extracted from the network – it is still there for a receive() call to get.
- **formatIPAddress (addr)**
  Convert the integral IP address into a n.n.n.n format string.

In addition, the Network package contains a DatagramStream class that may be used as a stream filter for sending a series of datagrams to the same address. The class contains:

- **retarget (addr, port)**
  Set the target address of the stream filter to the IP address and port given
- **getAddress (var addr, var port)**
  Get the current target address and port of the filter
- **operator -> (data, isout)**
  Stream data out through the filter to the target address and port. The data can be a vector, string or anything that can be cast to a string. The data is sent as one packet.
- **numDatagrams**
  Variable containing the number of datagrams sent through the filter.

15.5. Character typing package

This package contains a set of functions that can be used to determine the type of a character. The C language has had this forever and it has been called 'ctype' for just as long. To use the package, import the file 'ctype.aikido' and call the functions through the package name (ctype).

```
import ctype

if (ctype.isupper (ch)) {
}
```

The facilities provided are:

Function	Arguments	Purpose

isalpha	c	true if c is alphabetic
isdigit	c	true if c is a digit
isspace	c	true if c is a white space
isalnum	c	true if c is alphabetic or numeric
iscntrl	c	true if c is a control character
islower	c	true if c is a lower case letter
isupper	c	true if c is an upper case letter
ispunct	c	true if c is a punctuation character
isprint	c	true if c is a printable character
toupper	c	upper case version of c
tolower	c	lower case version of c

15.6. Mathematics package

This package includes a rich set of mathematical constants and functions. They mostly operate on real values (floating point numbers). To use the mathematics package you need to import the file 'math.aikido':

```
import math
```

Then you can use the functions by calling them though the package name. For example:

```
var theta = Math.sin (Math.PI)
```

15.6.1. Mathematical constants

The following constants are provided:

Constant	Meaning
E	Exponential
LOG2E	Base 2 logarithm of E
LOG10E	Base 10 logarithm of E
LN2	Natural logarithm of 2
LN10	Natural logarithm of 10
PI	pi
PI_2	pi / 2
PI_4	pi / 4
SQRT2	square root of 2
SQRT1_2	square root of _

15.6.2. Mathematical functions

Function	Arguments	Purpose
sin	x	Sine of x in radians
cos	x	Cosine of x in radians
tan	x	Tangent of x in radians
acos	x	Arccosine of x
asin	x	Arcsine of x
atan	x	Arctangent of x
sinh	x	Hyperbolic sine of x

cosh	x	Hyperbolic cosine of x
tanh	x	Hyperbolic tangent of x
exp	x	Exponent (e^x)
log	x	Natural logarithm
log10	x	Base 10 logarithm
sqrt	x	Square root
ceil	x	Ceiling
fabs	x	Absolute value
floor	x	Floor
atan2	x, y	Arctangent of x/y
pow	x, y	x raised to the power of y
ldexp	x, exp	$x * 2^{exp}$
fmod	x, y	Floating point remainder
trunc	x	integer value of floating point
round	x	rounded to nearest

I recommend that you either use the UNIX® manual pages or other source for the rigorous definitions of the mathematics associated with the functions. The Aikido functions map directly onto those in the operating system.

15.7. String object

Although Aikido provides string data types as part of the language, it is common to think of a string as an object that can be manipulated using member functions (methods if you like). The string object provided by the system library does just that. For those Java™ programmers among us, this should look familiar.

Creating an instance of the String class creates a String object. To do this, you need to import the file 'string.aikido':

```
import string

var q = new String()                    // empty
var s = new String ("hello")            // initialized with string
var p = new String (1234)               // initialized with integer
```

The parameter to the String constructor is anything that can be cast to a string value (see the cast operator in section 6.15). You can also omit the parameter, thus getting an empty object.

The following functions are provided:

- **append (s)**
  Append the given string to the end of the object.
- **length()**
  Get the length of the string
- **split (sep)**
  Split the string into sections. The separator (sep) is a regular expression that is used to determine the start and end of the sections. The return value is a vector of strings.
- **replace (s1, s2)**
  Replace all occurrences of substrings matched by the regular expression 's1' by the string 's2'. This does not overwrite the value of the object, it creates a new string and returns it.
- **substring (start, end = 0)**
  Given 2 indexes (start and end) return a substring of the string. If 'end' is omitted (or value 0) then the substring from 'start' to the end of the string is returned.
- **toInteger()**

Convert the string to an integer. The integer is returned. If the string does not contain a valid sequence of characters for an integer (as specified in the cast operator of the language) an exception is thrown.

- **indexOf (s)**
  Get the start index of the regular expression 's' in the string.
- **operator-> (stream, isout)**
  Stream the string into or out of a stream.
- **operator+ (s)**
  Concatenate a string to the end of the string object and return a new string object.
- **operator[] (i, j = -1)**
  Index the string object by either a single index or a range. If a single index is used, a character is returned, otherwise a String object is returned.
- **operator sizeof()**
  Get the size of the string

Since the string object provides overloaded operators you can use an instance of it like a regular string value. The split() and replace() functions allow regular expressions to be used to do the searching.

15.8. Streambuffer object

When a stream is connected to a network it is sometimes necessary to ensure that the data sent across the network is 'packetized' correctly for the protocol. Normally, when using streams, you are unaware of how the stream is buffered and don't really care anyway. If you do care about reading and writing chunks of data at a time you can use an instance of the Streambuffer class. A Streambuffer is an subclass of String so anything that a String can do, so can a Streambuffer.

A Streambuffer also acts like a stream in that you can read individual characters from it, one at a time. It maintains a 'current read position' pointer that is updated on a 'get'.

When reading a stream buffer from a stream, the buffer is read as an atomic entity. That is, all the available characters are read from the stream and inserted into the buffer. In the case of a network stream, this will mean that the whole packet is read at once.

When writing a stream buffer to a stream, the stream is flushed after the data is written. This ensures that the data is sent in the correct chunks.

To use a stream buffer you need to import the file 'streambuffer.aikido". The class Streambuffer can then be used:

```
import streambuffer

var buf = new Streambuffer()          // create new buffer
buf.put ("Hello")                     // append a string to it
buf -> output                         // send to output stream (and flush output)
```

The following facilities are provided:

- **clear()**
  Clear all data out of the object
- **put (v)**
  Put a string into the object. This is the same as String.append()
- **putchar (index, v)**
  Put a character into the buffer at the given index. The buffer must contain a string longer than the index.

- **getchar()**
  Get the next character from the streambuffer.
- **get (n = 1)**
  Get a series of characters from the streambuffer. The default is 1. Always returns a string value.
- **getall()**
  Get all remaining characters from the buffer. Returns a string value.
- **operator-> (stream, isout)**
  Read or write the buffer to a stream. If writing to a real stream (as opposed to a filter) the stream is flushed.
- **toString()**
  Convert the buffer contents to a string value
- **operator[] (i, j = -1)**
  Index the buffer as a string object
- **operator sizeof()**
  Get the size of the buffer.

15.9. Properties object

The Properties object is a very useful object for dealing with properties of a program. The Java™ langauge has a very similar object. The most common use of it is reading and writing properties files. These are files on disk that contain certain properties for a program. Properties each have a name and a value. They may be used for setting initial values of attributes in a program.

A Properties object can be populated from the contents of a *properties* file and may also be written out into a file, thus creating a properties file.

A properties file contains a set of lines of the form:

```
name = list of values
```

Where name is the name of a property meaningful to the program. The 'list of values' is a comma-separated list of strings.

To use the Properties object you must import the file "properties.aikido":

```
import properties
```

A new Properties object may be created:

```
var p = new Properties()
```

It may be populated from a stream or individual properties added one at a time.

To read a property from the object, use the normal subscript operator of the language. The value of a property is either a string (if it only has one value), or a vector of strings for multi-valued properties.

The properties object contains the following facilities:

- **put (prop, val)**
  Put a new property into the object. The 'prop' parameter is the name of the property. The 'val' parameter is either a string or a vector of strings.
- **remove (prop)**
  Remove a named property from the object.
- **replace (prop, value)**

Replace the value of a named property in the object.

- **operator[] (i, j = -1)**
  Retrieve the value of a property from the object. Only a single subscript may be used. The subscript is the name of a property to find. The return value is the value of the property.
- **operator-> (stream, isout)**
  Read or write the properties object to a stream.
- **operator foreach()**
  Iterate through each of the properties. Each element is an object of type System.Pair.

If we have a properties file called, say, "chat.props", we can read it as follows:

```
var strm = System.openin ("chat.props")      // open the file
var p = new Properties()                     // create the object
strm -> p                                    // read the properties
System.close (strm)                          // close the file
```

You can then query the properties object for individual properties:

```
var server = p["server"]                     // string property
var port = cast<integer>(p["port"])          // integer property
```

15.10. Containers

The system library contains a set of objects that can be used to hold other objects. These are:

- List
- Vector
- Map
- Queue
- Stack
- Hashtable
- Container (interface)

All containers are implemented as monitors to provide thread safety. All the classes implement the interface *Container*.

15.10.1. List

The List monitor is an object that provides a doubly-linked list of values. It is thread-safe. In order to use the List object, you must import the file 'list.aikido'

```
import list
```

Internally, the List object holds the values as a set of objects of type 'Item'. This contains the 'next' and 'prev' pointers as well as the data payload. Certain list operations require that an Item be passed (erase, insertBefore, etc.).

The List object has the following operations

- **insertEnd(v)**
  Insert the value 'v' at the end of the list
- **insertStart(v)**
  Insert the value 'v' at the start of the list.
- **insertBefore (item, v)**

Insert the value ‘v’ before the item ‘item’. The item for a value can be obtained using the ‘find’ function

- **insertAfter (item, v)**
  Insert the value ‘v’ after the item ‘item’. The item for a value can be obtained using the ‘find’ function
- **find (v, start=null)**
  Find the item associated with the given value, starting at ‘start’. If ‘start’ is omitted, the whole list is searched. Return either *null* or the item object.
- **erase (v)**
  Erase something from the list. The value ‘v’ is either an item or a value in the list. If it is a value, the list is searched for the list.
- **clear**
  Clear the list, deleting all the items
- **size**
  Number of items in the list
- **operator sizeof**
  As size()
- **operator in (v)**
  Check if the value ‘v’ is in the list, returns true or false
- **push_back (v)**
  Same as insertEnd()
- **push_front (v)**
  Same as insertStart()
- **operator foreach (var x)**
  Complex iterator. Returns each value in list in turn when used as an argument to the foreach command.
- **empty()**
  Check if the stack is empty, returns true if so.

15.10.2. Vector

Although the Aikido language contains a builtin value for vectors, the system library provides a monitor implementing a vector object. The reason for this is two-fold:

1. Those familiar with the Java™ and C++ languages will think of vectors as objects
2. The built-in vectors are not multithread safe.

To use the Vector monitor, import the file “vector.aikido”:

```
import vector
```

Since the Vector is implemented as a monitor, any access is automatically subject to a mutual exclusion lock. This means that it can be used safely in a multithreaded program without regard to any other locking mechanism.

The Vector monitor contains the following facilities:

- **push_back (v)**
  C++ style append. Value v is appended to the end of the vector
- **append (v)**
  Java™ style append. As push_back()
- **operator+ (v)**
  Add a value to the vector and return a new vector. Does not modify the object
- **getElements()**

Get the raw vector value from the object

- **erase (v)**
  Search for the value 'v' in the vector and erase the element. Throws an exception if the value is not found.
- **operator[] (i, j = -1)**
  Index the vector with either a single or range subscript.
- **size()**
  Get length of vector
- **operator sizeof()**
  Get the length of the vector
- **operator-> (stream, isout)**
  Read or write the vector from or to a stream
- **operator foreach()**
  Iterate through each element of the vector

15.10.3. Map

Like the Vector monitor (see section 15.10.2) the Map monitor implements an object-oriented Map object that is protected by a mutual exclusion locking mechanism. In order to use the Map object you must import the file "map.aikido":

```
import map
```

Aikido provides a built-in map data type. This object provides an object-oriented view of it.

The Map monitor contains the following facilities:

- **insert (key, data)**
  Insert a key/value pair into the map. Exception results if the key exists in the map already
- **isPresent (key)**
  Returns true if the key is present in the map.
- **find (key)**
  Find the value associated with the key in the map. An exception is thrown if it is not present.
- **erase (key)**
  Erase the key from the map. Exception thrown if not present
- **size()**
  Number of keys in the map
- **operator sizeof()**
  Number of keys in the map
- **operator foreach()**
  Iterate through the map. Each iteration results in a System.Pair object
- **operator[] (i, j = -1)**
  Index the map with a value. Behaves as get()
- **clear()**
  Clear all values out of the map
- **keys()**
  Get a vector containing all the keys in the map
- **get(key)**
  Retrieve the value of a key. Return none if not present
- **put (key, v)**
  Same as insert()
- **empty()**
  Is the map empty, true or false

15.10.4. Stack

A stack is a LIFO (Last In First Out) structure that is an extension of the List object. All the operations provided for List are also present for Stack. To use a Stack, you must import 'stack.aikido':

```
import stack
```

A stack implements 'push' and 'pop' operations. These always operate on the end of the underlying List.

In addition to the operations provided by the List (see 15.10.1), the Stack monitor provides the following operations:

- **push (v)**
  Push the given value onto the stack. This is inserted at the end of the list
- **top()**
  Get the value at the top of the stack. The value is not removed
- **pop()**
  Pop the value off the top of the stack

15.10.5. Queue

A queue is a FIFO (First In First Out) structure. Like Stack it is an extension of the List object. Items inserted into the queue are retrieved in the same order as they were inserted. To use the Queue, you need to import the file 'queue.aikido':

```
import queue
```

The Queue implements 'put' and 'get' operations.

- **put(v)**
  Insert the given value into the queue. This is inserted at the end of the queue.
- **get()**
  Get the value at the front of the queue. The value is removed from the queue.

15.10.6. Hashtable

A Hashtable is a fast-access map. It is implemented as a vector of maps. The keys are hashed by the System.hash() function and the value is used to select a map in the vector.

The regular Map object should provide enough speed for most applications, but if you are writing a program that requires a very large mapping, the Hashtable might be better.

To use the Hashtable, import the file 'hashtable.aikido':

```
import hashtable
```

A Hashtable object takes one ones optional parameter in its constructor: size of the table. This is fixed for the lifetime of the object and cannot be changed. The default is 1009 elements.

The Hashtable behaves exactly like a Map.

15.11. Security package

This provides a simple password entry and encryption package. Only 2 functions are supported.

The file "security.aikido" must be imported in order to use this package. The functions may be called through the package name (Security).

```
import security

var password = Security.getpassword ("Password:")
password = Security.encryp (password,"da")
```

- **getpassword (prompt)**
  Print the prompt to the terminal and read the user's password from the keyboard. Terminal echo is switched off while the password is being entered. The string entered is returned.
- **encrypt (key, salt)**
  Encrypt the string passed in 'key' using the 2-letter string 'salt'. The encrypted string is returned.

15.12. GTK+ Graphical Toolset package

GTK+ is a popular public-domain library that allows application to be written to use Graphical User Interfaces. It is primarily aimed at Unix® and Linux machines but there is also a Windows® port available on the internet.

Aikido provides an interface to the GTK+ libraries. It can be accessed by importing the file 'gtk.aikido':

```
import gtk
```

The GTK+ library is written in C, but the designers took a very object-oriented approach to it, implementing pseudo-objects in C and providing the equivalent of methods for each object. The Aikido interface to the library functions converts this into real object classes and methods. For each object class in the GTK+ hierarchy, the Aikido interface provides a class.

For example, the GTK+ functions:

```
GtkWidget *gtk_window_new (GtkWindowType type);
void gtk_window_set_title (GtkWindow *window, const gchar *title) ;
```

Are implemented as the following Aikido class:

```
package GTK {
   public class Window (type, _toplevel = true) extends Bin (false) {
      if (_toplevel) {
         object = gtk_window_new (type)
      }

      public function set_title (title) {
         gtk_window_set_title (object, title)
      }
   }
}
```

The class takes 2 parameters: the window type and an optional *_toplevel*. The *_toplevel* parameter is set to true if the Window is created directly and false if it is a superclass.

If the *toplevel* parameter is true, then the function gtk_window_new() is called to create the window. The result is assigned to the variable *object*. This variable exists in the very top level class in the hierarchy: the *Object* class. The value returned by gtk_window_new() is a pointer type. This is treated as an integer type in Aikido (integers are 64 bit so there is enough room). The *object* variable in a GTK Object is passed down to the GTK+ C functions as-is and is cast to a pointer.

The member function *set_title*() simply calls the GTK+ C function, passing the *object* variable and the title string.

Most of the GTK+ C functions are implemented as *raw native* functions (see section 5.4.2). Some of them are done as regular native functions and have an interface function coded for them. These are the ones that do not conform to the raw native rules.

Because the Aikido GTK+ interface is simply a mapping to a native library instead of being implemented in Aikido itself, the performance is very good. It is good enough to allow full GUI applications to be written in Aikido instead of having to resort to C code.

15.12.1. GTK+ resources

In order to use the GTK+ interface in Aikido, you will need to download the GTK+ libraries. These can be obtained from:

http://www.gtk.org

The web site contains some documentation on the GTK+ library and its requirements. I suggest that you examine the header files, GTK+ example code and the Aikido GTK+ interface files to get further information.

When you have installed GTK+ on your system you will need to tell Aikido where the libraries are. To do this, set the AIKIDOPATH environment variable to point to the directory containing them. You might also need to set the LD_LIBRARY_PATH environment variable to locate the GLIB libraries.

The library files should be called a specific name on the system. On Unix® systems the should be called: **libgtk.so** and **libgdk.so**. On Windows®, call them **gtk.dll** and **gdk.dll** (you might have to rename them).

15.12.2. Signals

In any GUI toolkit, the application communicates with the toolkit through an event mechanism. The GTK+ library uses the notion of *signals* to communicate events of interest. The idea is that an object can set it up so that a function is called upon receipt of a named signal. Signals names are strings.

The class GTK.Object contains the signal connection functions:

```
package GTK {
   public class Object {
      public function signal_connect (sig, func, data) {
      }
   }
}
```

The signal_connect() function takes 3 parameters: the name of a signal (a string); the function to call when the signal occurs; and an arbitrary datum to be passed to the function.

The following example shows the use of signals:

```
class ExitConfirmation (text) extends GTK.Dialog {
   set_border_width (10)
   set_modal(true)
   set_title ("Confirmation")
   set_position (GTK.WIN_POS_CENTER)

   // pack the label text into the box
   var vbox = get_vbox()
   var label = new GTK.Label (text)
   vbox.pack_start (label, true, true, 0)
   label.show()

   function no_clicked (widget, p) {        // called when the No button is clicked
      hide()
   }

   var actarea = get_action_area()

   // make the action buttons
   var yes = new GTK.Button ("Yes")
   actarea.pack_start (yes, true, true, 0)
   yes.show()

   var no = new GTK.Button ("No")
   actarea.pack_end (no, true, true, 0)
   no.show()

   // connect the signals
   yes.signal_connect ("clicked", function (a,b) { System.exit(0) }, null)      // anonymous
   no.signal_connect ("clicked", no_clicked, null)                               // in class
}
```

15.13. Filename package

Programs frequently manipulate filenames. This package provides facilities to perform many operations on strings that are in the format of file names. All the functions return a new string (with the exception of explode which returns a vector). In no case is the parameter modified by the function.

The filename package resides in the file ‘filename.aikido’ and should be imported before use:

```
import filename
```

- **explode (filename)**
  Given a string representing a filename, split it into a vector of strings, each element of which is a component of the path. If the filename is ‘rooted’, the first string in the vector will be empty.
- **implode (vector)**
  Given a vector of strings, form them into a path name and return the string formed
- **dirname (path)**
  Extract the directory portion of a path name

- **filename (path)**
  Extract the filename portion of a path name
- **suffix (path)**
  Extract the suffix (file extension) from a path name
- **basename (path)**
  Extract the portion of a filename without the suffix (extension)
- **export (path)**
  Create a full NFS path name for a file. Prefixes /net/machinename to the path if it is not already over NFS. If on Windows, the export path name is \\machinename\filename.

The filename package used the System.fileSeparator and System.extensionSeparator variables.

15.14. Lexical Analyzer package

Lexical analysis consists of taking an input text and applying transformations on it to split it into a series of tokens. The input text is said to consist of an ordered sequence of tokens separated by white space characters. The lexical analyzer package provides a class that can be used to perform the analysis of a vector of lines of input text.

A token has a type and some associated information. The token types recognized by the raw lexical analyzer (unextended) are:

- IDENTIFIER
  An identifier is a token consisting of an alphabetic character and containing a sequence of alphabetic and numeric characters.

- NUMBER
  A number is a based (base 2, 8, 10 or 16) sequence of valid characters for that base that represents a numeric quantity.

- STRING
  A sequence of characters (excluding double quotes and line feed characters) enclosed in double quotes. Can contain escape characters prefixed by a backslash character.

- CHAR
  A single character enclosed in single quote marks. Can contain escape characters.

- BAD
  Unrecognized token

- EOL
  End of line token. Can be returned when running in line based mode.

In addition, the analyzer supports blank lines comments. A comment is a started by either a ‘#’ character or a ‘//’ character sequence and extends to the end of line.

The lexical analyzer is in the file lex.aikido and must be imported using:

```
import lex
```

To use it, create an instance of the class Lex and pass a vector of strings the contain the source text, one line per string.

```
var lex = new Lex (lines)
```

15.14.1. Adding tokens

The raw lexical analyzer recognizes the tokens listed in the previous section. In order to be able to use it, you need to add other tokens to it. There are 2 types of tokens that you can add:

1. Reserved words
2. Operator tokens

Reserved words are identifiers whose spelling is deemed to be a token by itself. For example, the reserved word 'while' has a meaning to program what parses C. Operator tokens are a sequence of characters that form a token. Examples are '+=' or '<<'.

In order to add new tokens to the analyzer you need to extend the set of tokens that already exist. The lexical analyzer package contains an enumerated type called 'Tokens'. We use the Aikido 'block extension' feature to add to it:

```
extend Tokens {
   WHILE, DO, FOR,
   PLUSPLUS, MINUSMINUS
}
```

This adds 5 new tokens to the set of tokens recognized. Now we need to give the tokens a definition. To define a reserved word token use the function 'addReservedWord()'. To define an operator token, use the function 'addToken()'

```
lex.addReservedWord ("while", WHILE)
lex.addReservedWord ("do", DO)
lex.addReservedWord ("for", FOR)

lex.addToken ("++", PLUSPLUS)
lex.addToken ("—", MINUSMINUS)
```

15.14.2. Extracting the token sequence

Once the tokens have been added to the analyzer, the program can read the next token from the input text by calling the 'nextToken()' function. The analyzer is line based, so the function 'readLine' can be used to read the next line.

The algorithm for parsing a token sequence is as follows:

```
var lex = new Lex (lines)              // create the analyzer

extend Tokens {
   // new tokens
}

lex.addReservedWord (…)                // add reserved words
lex.addToken (…)                       // add tokens

lex.readLine()                         // read the first line
lex.nextToken()                        // read the first token
```

```
while (!lex.eof()) {                    // until end of tokens
   switch (lex.currentToken) {          // process the current token
    case <token>:
      // process the token
     lex.nextToken()                    // move on one token
   }
}
```

The main functions and variables for the analysis are:

- **readLine()**
  Read the next line from the input line set. Increment the current line counter. Skip blank lines and lines containing just comments

- **nextToken()**
  Read the next token from the input text. Set the variable 'currentToken' to the token type.

- **line**
  Contains the whole contents of the current line

- **ch**
  Contains the index into the current line of the start of the next token. A call to 'nextToken' will move this on to the next token.

- **lineno**
  The current line number, starting at 1 for the first line in the sequence

- **spelling**
  The spelling of the current token (if appropriate). This contains the characters that make up an identifier or string if the current token is IDENTIFIER or STRING

- **number**
  If the current token is NUMBER, this contains the value of the number.

- **currentToken**
  The current token type. This will be one of the built-in token types (IDENTIFIER. NUMBER, etc) or one of the tokens added by the user

- **eof()**
  The analyzer has reached the end of input text.

- **addReservedWord()**
  Add a reserved word and its corresponding token to the analyzer

- **addToken()**
  Add an operator token to the analyzer.

- **reset()**
  Reset the analyzer to the start of the sequence. No lines have been read and no tokens parsed.

- **match(token)**
  Look at the current token and if it matches the parameter, skip to the next token and return true. If no match, don't move the token pointer and return false

- **getIdentifier()**

Extract an identifier from the token sequence. The value of currentToken must be IDENTIFIER. This returns the spelling of the identifier. If there is no identifier in the sequence, throw an exception.

- **getNumber()**
  Line getIdentifier(), but extract a number from the stream. Throws an exception if the current token is not a number.

The analyzer always has a value for 'currentToken'. This is either the type of the token just extracted by 'nextToken()' or BAD. It starts out with BAD and each time 'nextToken()' is called, the value of 'currentToken' is set to the type of the currently recognized token.

The functions match(), getIdentifier() and getNumber() are utility functions that are used to extract certain token types from the input. The definition of match() is:

```
function match (t) {
   if (currentToken == t) {
      nextToken()
      return true
   }
   return false
}
```

The getIdentifier() and getNumber() functions are similar.

The variables 'spelling' and 'number' are used to provide additional information when the token type is IDENTIFIER, STRING and NUMBER. In the case of 'spelling', it contains the characters that make up the identifier or string. For a number, the variable 'number' contains the value of the number.

15.15. Registry package (Windows® only)

Aikido provided a package for use on the Microsoft® Windows® Operating system only. This package allows access to the Windows® Registry. This is a hierarchical database of keys and values and is used by all applications to store persistent data.

The registry package is in the file registry.aikido and is imported by:

```
import registry
```

The package is named *Registry* and provides the following functions:

- **openKey (key, name)**
  Open a subkey of an open key. There are a set of keys that are opened by the operating system. These are the top level keys. This function opens a subkey of a key and returns the handle for the subkey
- **closeKey (key)**
  Close an open key
- **enumKeys (key)**
  Retrieve the names of the subkeys of an open key. The value returned is a vector of strings, each of which is a subkey name
- **enumValues (key)**
  Retrieve the names of all the values of an open key. A vector of strings is returned.
- **getValue (key, valuename)**
  Retrieve the value of a named value of an open key. The value of the valuename is returned as a Aikido value. For example, if the registry value is a type REG_DWORD, the value returned is of type integer.

- **setValue (key, valuename, valuevalue, type = none)**
  Set the value of the named value of an open key. The 'valuevalue' parameter is a Aikido value whose type is appropriate for the key value. If the type of the value to be stored in the registry is not obvious from the type of the 'valuevalue' parameter, the additional 'type' parameter may be set to one of the REG_ values.

The top level keys that are always open are:

1. HKEY_CLASSES_ROOT
2. HKEY_CURRENT_CONFIG
3. HKEY_CURRENT_USER
4. HKEY_LOCAL_MACHINE
5. HKEY_USERS

The registry types that can be passed to the setValue function are:

Type name	Type in registry	Aikido type
REG_DWORD	32 bit integer	lower half of integer
REG_DWORD_LITTLE_ENDIAN	same as DWORD	as DWORD
REG_DWORD_BIG_ENDIAN	Big endian 32 bit integer	lower half of integer in big endian format
REG_QWORD	64 bit integer	whole integer
REG_QWORD_LITTLE_ENDIAN	same as QWORD	whole integer
REG_SZ	zero terminated string	string
REG_EXPAND_SZ	zero terminated string containing environment variables	string
REG_MULTI_SZ	zero terminated set of zero terminated strings	vector of strings
REG_BINARY	Binary data	bytevector
REG_NONE	No value	none

15.16. Java™ Object model

Aikido provides a subset of the Java™ object model for use by programmers who are familiar with the Java™ language. Use of the Java™ Object model will increase the size of the program and will result in lower performance because most of the operations executed by the interpreter rather than in native code. If you are writing a program that will be converted to the Java™ language, or don't want to learn another programming model, you can use the Java™ object model in your Aikido programs.

The Java™ Object model is stored in the aikido.zip file as a set of Aikido files. To use the object model you need to import the Aikido files. This is done using a regular import statement. For example, to use the Java™ Thread model:

```
import java.lang.Thread
```

All the Java™ objects are held in the package 'java'. There are sub-packages of this for the 'io' and 'lang' components of the model. Only a subset of the Java™ objects is currently included. The list of objects that are defined is:

- java.lang.System
- java.lang.Boolean

- java.lang.Object
- java.lang.Runtime
- java.lang.String
- java.lang.Thread
- java.lang.ThreadGroup
- java.lang.Exception
- java.lang.Process
- java.lang.StringBuffer
- java.io.DataOutputStream
- java.io.FileInputStream
- java.io.FilterOutputStream
- java.io.OutputStream
- java.io.File
- java.io.FileNotFoundException
- java.io.IOException
- java.io.PrintStream
- java.io.FileDescriptor
- java.io.FileOutputStream
- java.io.InputStream

When you import a Java™ object from the library it must be referenced by its full package name unless you place a 'using' statement in the program. For example:

```
import java.lang.Thread

class Mythread extends java.lang.Thread {
  // …
}
```

can also be coded as:

```
import java.lang.Thread

using java.lang

class Mythread extends Thread {
  // …
}
```

Chapter 16. Worked example: A chat service

Let's see how to develop a real program in Aikido. This example is an actual running program used in my office. It has proved to be very useful for improving team communication.

The example program we will develop in this chapter is a simple chat system. We are probably all familiar with the notoriety of public chat systems. Just mention the words "chat room" and you will evoke looks of horror and images of internet stalkers immediately spring to mind. The concept of a chat service has been around for a long time, ever since people started using mainframe computers and networks. The UNIX® Operating System has had the 'talk' command (a one-to-one chat system) for ever. Digital Equipment Company had a nice chat system on their VMS operating system (VAX Phone if I remember correctly).

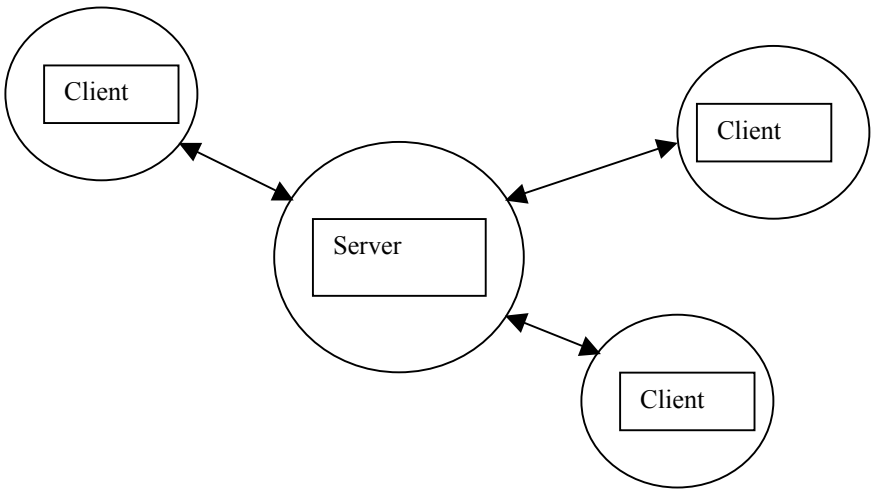
When the internet came along, the most popular chat service was Internet Relay Chat (IRC). This is a powerful and flexible multi-user chat system that worked worldwide on a vast scale. More recently, America Online (AOL) have popularized their Instant Messenger (IM) system and there millions of users logged on at any one time.

I am not going to try to write anything as complex as a multi-server IRC or an IM for an example of how to use a programming language. The chat system I will explain here is an extremely simple but effective system that omits most of the bells and whistles of the major systems. It will, however, be fully functional.

During this chapter we will show the development of 2 programs. Each one will be slowly built up from parts as we write those parts. There will necessarily be repetition of sections of code as the code grows.

16.1. Architecture

The chat service consists of a single server and a set of clients. The communication between the clients and the server is via the network using the TCP/IP protocol. All the clients connect directly to the server and messages are exchanged over the communication streams. The following diagram shows a top level view of the architecture:



Three clients are shown connected to the single server. The clients and server can each be running on a different machine. The communication is via a local ethernet or equivalent.

16.2. Protocol

For simplicity, let's make the messages sent between the client and server strings of characters. Let's define a single message as consisting of 3 parts - each part being separated by a colon character. The first part of the string will be a command identifier and will be an upper case word.

Consider what a client has to do to operate with a chat server:

- Connect to the server program at a given location on the network
- Log in to the server providing an identifier for who the user is and possibly some authentication information
- Join and leave "chat rooms". A chat room is where the users reside. Users in a chat room are sending messages to each other.
- Obtains lists of who is present in the chat rooms and other rooms on the system
- Send messages to a room. A message sent to a room is seen by all users in the room
- Log off from the server, leaving all the rooms and disconnecting the network connection

The connection from the client to the server will be handled by making a network stream. Once a connection is made, the client will start sending commands to the server.

In a chat system each user is identified by a user name or *nickname*. A nickname is a short identifier usually similar to the users first name in real life. The nickname is the primary means of identification in our chat system.

The following commands will provide the appropriate functions:

Command	Argument 1	Argument 2	Meaning
LOGIN	nickname of user	password	Log on and authenticate user
LOGOUT	nickname		Log the user out
JOIN	name of room		Join the conversation in a room
LEAVE	name of room		Leave a room
TALK	name of room		Set room as current room
MESSAGE	nickname	text of message	Send message to all in current room
BROADCAST	nickname	text of message	Send message to all logged in users
MOTD			Get the Message or the Day
USERS			Get a list of all users who are logged in
ROOMS			Get a list of all the rooms available
KICK	nickname	nickname to kick off	Remove a user from the server
KILL	nickname	user number	Select a user given his number and log him off
SHOWLOG	room	max lines	Show the log of messages for a given room
CLEARLOG	room		Clear the log file for a room
SHUTDOWN	time in seconds	reason	Shut the server down after a period of time

If an argument is listed as blank in the table it means that it is ignored by the command. However, it still must be present in the command string.

The commands listed above are those sent from the client to the server. We also need to be able to send messages from the server to the client. These will carry things such as notifications, messages to the client from other users and errors. Let's decide to use exactly the same format for the server to client messages as for the commands going the other way.

Message	Argument 1	Argument 2	Meaning
JOIN	nickname	room	The user named by nickname has joined the named room
LEAVE	nickname	room	The named user has left the named room
MESSAGE	nickname	message text	Incoming message from named user
ERROR	error text		Error from server
KICK	nickname	reason	Notification of user being kicked off by the named user
KILL	nickname		Notification of user being killed by named user
LOGIN	nickname		Named user has logged in
LOGOUT	nickname		Named user has logged out
SHUTDOWN	time in seconds	reason	Warning of impending server shutdown
ENDLOG			End of output from SHOWLOG command
TALKING	room		Notification of success of TALK command

16.2.1. Rooms

The protocol refers to rooms. We need to define what a room is. A room is a location in the server identified by a name. There can be any number of rooms in existence at once. A room contains a set of users. Rooms can be created at will. A room is created if a user joins it. Rooms are never deleted.

When a message is sent to a room all the users in the room receive the message.

16.2.2. Users

A user of the system is an entity that represents a person typing at a keyboard communicating with the other users of the system. A user is identified by a nickname but also has the attributes:

- Full name
- Password
- User session id

The full name is the name of the person represented by the user. The password is a secret word that is known only to the user and verified by the server. When a user logs in, he is assigned a unique user session identifier. This is used to differentiate between the sessions when a user logs in to the server twice (there is no limit to the number of times a user may be logged in).

A user may be in many rooms at a time. Typically when a user says something, a message will be sent to his 'current room'. The user may set his current room at any time using the TALK command. It is also possible for a user to set his current room to "*" and after that, any message sent by the user will be sent to all the rooms in which the user is present.

16.3. Server

Let's begin writing the chat server. The first thing we need is a package for the server:

```
package ChatServer {
}
```

Because the package has no parameters, an instance will be created automatically by the Aikido system. Any code placed in the body of the package will be executed when the chat server starts.

The chat server is a network server, so we will need to use network services in it. To do this we need to import the network package into the server package:

```
package ChatServer {
   import net
}
```

This imports the net.aikido file at the top level of the program.

16.3.1. The configuration file

The chat server will need information that remains static. The server is connected to a network so will need to know which machine it is running on and on what TCP port it is listening. The Aikido system library contains a package called *Properties*. This is basically a map of property name versus property value. The properties object can be read from and written to a file. The format of the file is:

```
name = value list
```

Where name is an identifier and value list is a set of strings separated by commas. The properties required by the chat server are simply the machine name and port number, so the file (let's call it "chat.props") will contain:

```
# chat server properties
server = databank1
port = 6000
```

Here we set the machine name (property "server") to a valid machine name for our network and pick a TCP port number not assigned to anything else running on the server. How do we pick a port number? This is one of the most problematic areas with the whole of TCP/IP programming. There is no right answer, short of consulting a port mapping service available for some protocols on some systems. For our purposes, the number 6000 seems ok.

So, once we have a properties file set up we need to be able to read it into the program and read the value of the properties. We do this by opening the file and streaming the contents into an object of type Properties:

```
import properties                       // import the package

var props = new Properties()            // create blank Properties object
var file = System.openin ("chat.props") // open the properties file
file -> props                           // read properties into object
System.close (file)                     // close the file

// now read the properties we care about
var server = props["server"]            // read server machine name
var port = cast<int>(props["port"])     // read port number as int
```

We now have read the properties from the file and have set 2 variables (*server* and *port*) to the properties we care about from the file. Notice that the *props* object is subscripted with the name of the property to read. This uses an overloaded operator in the Properties class. The 'port' property is supposed to be an integer type, but all properties come in from the file in the form of strings. Therefore we cast it to integer before assigning to the *port* variable.

16.3.2. Allowing connections from clients

The chat server is a typical server. It does nothing else but sit and wait for incoming connections from clients. The first thing we need to do is create a server socket for the connections:

```
var socket = Network.openServer (server, port, Network.TCP)
```

Then, once the socket is opened, we need to enter a loop accepting connections from the socket. The creation of the socket does a passive open of the network port, so it is ready to accept connections from clients. The loop is:

```
for (;;) {
   var s = Network.accept (socket)
   // got a connection, process it
}
```

Putting this all together with the package and the properties management we have:

```
package ChatServer {
   import net
   import properties                    // import the package

   var props = new Properties()         // create blank Properties object
   var file = System.openin ("chat.props")        // open the properties file
   file -> props                        // read properties into object
   System.close (file)                  // close the file

   // now read the properties we care about
   var server = props["server"]         // read server machine name
   var port = cast<int>(props["port"])  // read port number as int

   var socket = Network.openServer (server, port, Network.TCP)

   for (;;) {
      var s = Network.accept (socket)
      // got a connection, process it
   }
}
```

Notice that we have not dealt with any exceptions this far. Anything that goes wrong in this part of the server will be pretty serious so we are happy for any exceptions to be reported directly to the person running the server.

So now we have created a Aikido program that will run forever accepting incoming connections on a given port number on a given machine. When it gets the connection it will do nothing. Let's fix that little problem.

A server is inherently multithreaded. It must be able to process messages from clients and at the same time accept new connections from other clients. One way to handle this is to create a new thread for every client connection. We choose to do this because threads are relatively cheap and simple to program.

We define a thread called *userServer* to handle all the communication from a particular client. This thread takes, as a parameter, the stream which was created by the incoming client connection request. The thread has to run until the stream is closed by the client:

```
thread userServer (clientStream) {
   while (!System.eof (clientStream)) {
      // read and process client commands
   }
}
```

Putting this thread together with the main server package we get:

```
package ChatServer {
   import net
   import properties                    // import the package

   var props = new Properties()         // create blank Properties object
   var file = System.openin ("chat.props")      // open the properties file
   file -> props                        // read properties into object
   System.close (file)                  // close the file

   // now read the properties we care about
   var server = props["server"]         // read server machine name
   var port = cast<int>(props["port"])  // read port number as int

   thread userServer (clientStream) {
      while (!System.eof (clientStream)) {
         // read and process client commands
      }
   }

   var socket = Network.openServer (server, port, Network.TCP)

   for (;;) {
      var s = Network.accept (socket)           // accept new connection
      userServer (s)                            // spawn thread to handle it
   }
}
```

16.3.3. Processing client commands

Now that we have a thread running for each client connection we need to be able to process the commands those clients send us. The clients are connected to the server through network streams. We have one thread on the server (userServer) associated with each stream.

Remember that the commands from clients will consist of a string broken in 3 parts. We need to be able to read the string from the stream and break it up into its constituent parts.

First let's see how to read the command from the stream. The main loop of the *userServer* should be augmented as follows:

```
var command = ""                                    // string var to hold command
while (!System.eof (clientStream)) {
   clientStream -> command                          // read command from stream
   try {
       // process client command
   } catch (error) {                                // catch exceptions
      ["ERROR:", error, ":x\n"] -> clientStream     // report as errors to client
      System.flush (clientStream)
   }
```

The next step is to write the code to actually process the client commands. Let's define a function inside the *userServer* thread called *clientCommand*() that will do the whole processing for us.

Commands are strings of 3 fields separated by colon characters. A convenient way to extract the fields is to use a regular expression. What must this regular expression match? The first field is a series of characters ending in ':'. The second is also a series of characters ending in ':'. The third is the series of characters from after the second ':' to the end of the string. We want the 3 fields to be extracted without the colon characters.

A regular expression to do this is:

```
"([^:]+):([^:]+):(.*)"
```

This consists of 3 subexpressions (inside the parentheses). The first one says: "one or more of any character except colon". The second is the same as the first. The third says: "a sequence of zero or more characters"

The code to extract the fields (*command* is a parameter to the *clientCommand()* function):

```
var ex = command["([^:]+):([^:]+):(.*)"]
if (sizeof (ex) > 3) {                              // need 3 fields
   var cmd = command[ex[1].start:ex[1].end]
   var nick = command[ex[2].start:ex[2].end]
   var text = command[ex[3].start:ex[3].end]
} else {
    throw "Invalid command: " + command
}
```

We have chosen to name the fields after their normal use. This may sometimes be wrong but it's only a name. The "string subscripted by string" expression returns a vector of System.Regex objects. Each of these represents a substring within the string being subscripted. The first element in the vector is the start and end index of the whole regular expression. The follow an element for each subexpression in the regular expression. Each element contains the start and end subscripts of the section of the string matched by that subexpression. Thus, element 1 contains the subscipts into the string for the command part of the command string, element 2 contains the first argument and element 3 contains the second argument.

If the regular expression does not match the command, we throw an exception saying that we don't recognize the command. This will be caught by the *userServer* thread main loop.

Now that we have extracted the fields from the command we can write code to look a the command and do something with it:

```
switch (cmd) {
```

```
case "LOGIN":
   // process login command
   break
case "LOGOUT":
   // process logout command
   break
// etc
}
```

This switch statement switches on the 'cmd' field of the command. Aikido switch statements can use any type in the case limbs. In this case we use a string. We should include all the recognized commands in this switch statement.

16.3.4. Users and rooms

Before going any further with the server command processing we need to define some internal objects for the server. The server deals with users who are logged in and have joined conversations in rooms. We will need to keep a set of rooms and a set of logged in users.

We have chosen to make this server secure. This means we have to validate any incoming user to make sure she is who she says she is. The simplest way to do this is by a password mechanism. We have already defined that the LOGIN command contain a nickname and a password. We need a way to determine if this user has an account on the server and if so, if the password supplied matches that on record for the user.

To do this, we have another file. This file is very like the UNIX® /etc/passwd file in that it contains a set of lines, each of which contains information about a single account. The format of the lines is a set of fields separated by colons (seem familiar?). The fields are:

1. User's nickname (eg, joe)
2. User's full name (eg. Joe Soap)
3. Users password. Let's assume some sort of encryption for this. It doesn't matter for the server but it would be safer.

We can write a function to check that the user has an account and that the password matches. Since the information is stored in a file we have 2 choices. We can:

1. Open the file when the server starts and read it into a list. Then use this list to validate the user
2. Open the file every time we want to check a user

The first would be higher performance but means that if we add a new user we would have to stop and restart the server in order to read the file again. This is like having to reboot your PC when you add anything to it – the hardware settings are read by DOS when it starts.

Having to restart the server for a new user would be onerous. We choose the second option – open the file every time. The function to do this can be coded as:

```
function validateUser (name, password) {
   var accounts = System.readfile ("chat.users")
   foreach account accounts {
      var expr = account["([^:]+):([^:]+):(.*)"]
      if (name == account[expr[1].start:expr[1].end]) {
         if (password != account[expr[2].start:expr[2].end]) {
            throw "Invalid password"
         }
```

```
            return account[expr[3].start:expr[3].end – 1]
            }
        }
        throw “No such user”
    }
```

This function is called with the name and password for the user. It opens the file and reads all the lines into a vector. The *accounts* variable holds one element for each line in the file. The linefeed character is kept in the string. The function goes through each line in the vector and extracts the 3 fields from the string. If the first field matches then we have found the entry for the named user. Then we check if the password string matches that in the file. If it doesn't then we throw an exception saying that the password is wrong. If all is well, the function returns the full name of the user.

We need to define classes to hold a user and a room. Let's start with a room. A room has a name and a set of users. Rooms are shared among all the users, so the userServer threads will be accessing the rooms. We need to hold a vector of users in the room. Because the room is shared among the user threads, we need to protect the vector against simultaneous update by 2 threads at the same time. Fortunately, the System library contains a class the implements a protected vector. This class is called Vector and is implemented as a monitor. To use this class we need to import it:

```
import vector
```

So what else does a room need?

- A log file so that the messages sent in a room can be read back by someone who has missed part of a conversation
- Functions to send messages to all users in the room, add a new user and remove a user
- A function to list all the users in the room
- Functions to gain access to the room's name

Logging to a file is something that is common to a number of classes. We can define a function to do the logging:

```
function log (stream, command, from, message) {
    if (message == “”) {
        message = “xxx”
    }
     [command, ‘:’, from, ‘:’, message, ‘\n’] -> stream
    System.flush (stream)
}
```

This takes a stream and the 3 fields of a message. It writes the 3 fields to the stream, separated by colons. It is possible that there will be no third field, so the function checks for an empty message and fills it in if necessary.

With all these requirements in place, let's define a room class:

```
class Room (name) {
    var users = new Vector()                    // protected vector of users

    var logfile = System.openup (“room_” + name + “.log”)

    // send a message to all users in the room
    public function send (command : string, from : string, message : string) {
        log (logfile, command, from, message)                    // log message
        foreach user users {
```

```
            // send message to user
        }
    }

    // add a user
    public function join (user) {
        users.append (user)                                  // add the user
        send ("JOIN", user.getNick(), name)                  // tell the others
    }

    // remove a user
    public function leave (user) {
        send ("LEAVE", user.getNick(), name)         // tell other users
        users.erase (user)                           // remove the user
    }

    // list all the users in the room
    public function listUsers (stream) {
        ["Room \"", name, "\":\n"] -> stream
        foreach user users {
            // write user information to stream
        }
        '\n' -> stream                          // terminate line
    }

   // write the name of the room to a stream
    public function showName (stream) {
        ["\t", name, "\n"] -> stream
    }

    // get the name of this room
    public function getName() {
        return name
    }
}
```

I have omitted the calls to functions in a, yet to be defined, user object. I'll add these when I write the code for the user object.

So now we need to define a class for a user. Let's think about what a user object would need:

- A variable to hold the room in which the user is currently talking
- A variable to hold the full name of the user
- A variable for the current session id for this user
- Functions to get access to the name and nickname
- A function to send a message to the client
- Various utility functions

The user object will be created when a user logs into the server. The first thing it must do is validate that the user's nickname and password are correct. The side effect of this (the validateUser function) is to get the full name of the user.

The user object needs to send messages to its stream. The stream is that which was created when the client connected to the server. Sending a message to stream can be as simple as formatting a string and using the stream operator to output the string to the stream. For the sake of illustration we will use a class provided

in the System library to hold the data before sending to the stream. This is a Streambuffer class. Its purpose is to allow a packet to be formed before sending to a stream to ensure that the data is packed correctly in a packet. To use it, we need to import it:

```
import streambuffer
```

Let's go ahead and define the class:

```
class User (nickname, password, outstream, id) {
   var name = validateUser (nickname, password)          // validate the user

   public var currentRoom = null                         // room where I am talking

   public function getNick() {                           // get the nickname
      return nickname
   }

   public function getName() {                           // get the full name
      return name
   }

   public function getID() {                             // get the session id
      return id
   }

   var outputbuffer = new Streambuffer()                 // buffer for sending data

   var currentTXid = 0                                   // see section 16.3.4.1

   // send a message to the outstream
   public function send (txid, command, from, message) {
      if (txid != currentTXid) {
         outputbuffer.clear()
         outputbuffer.put (command + ":" + from + ":" + message)
         outputbuffer -> outstream
         currentTXid = txid
      }
   }

   // is this the root user
   public function isRoot() {
      return nickname == "root"
   }

   // close the stream
   public function close() {
      System.close (outstream)
   }

   // time last message was sent by this user
   var lastMessage = System.time()

   // how long has the user been idle (in seconds)
   public function getIdleTime() {
      return (System.time() – lastMessage) / 1000000
```

```
    }

    // reset the idle counter
    public function resetIdle() {
        lastMessage = System.time()
    }
}
```

The class has a few features we have not yet described. The isRoot() function looks at the nickname of the user and returns true if this is the superuser. The superuser has the name "root". The object keeps a note of the last time a message was sent by this user to other users. The difference between the current time and the time a message was last sent is the user's idle time.

16.3.4.1. Transactions

One subtle problem exists with a room. Since users can be present in multiple rooms at the same time and each room maintains a list of the users present in it there is no way to prevent a user seeing multiple copies of a message. Consider if a user is present in 3 rooms and another user sends a message to all the rooms occupied. He can do this either with a BROADCAST command or if he is talking in more than one room. In this case the user will get 3 copies of the message, one from each of the room he is occupying. This is undesirable.

The solution is to associate a "transaction identifier" (a number) to each incoming message. When messages are sent to users from rooms, the transaction identifier is compared with one held in the user object and the message is only sent if the identifiers do not match. Then the transaction identifier held in the user object is set to the one of the message. If a new transaction identifier is allocated for each incoming message then each user will only get the first message sent to it with a new transaction identifier.

To explain this further let's assume we have an integer variable that holds the current transaction identifier. Let's call this *messageTXid*.

```
var messageTXid = 0
```

Each time a userServer thread receives a new message it increments the messageTXid variable and sends the message to all the rooms it is talking in. Inside the user object is an integer holding the number of the last transaction identifier sent to the user.

```
class User .... {
  // bits of user

    var currentTXid = 0

    public function sent (txid,.....) {
        if (txid != currentTXid) {
            // send the message
            currentTXid = txid
        }
    }

  // more user body parts
}
```

The send function in a user object is passed a variable txid. This is the current transaction id of a message sent to a room. You can see that only the first message sent to a user object will get sent as the next message with the same txid will fail the test in the send function.

Since the "current transaction identifier" is a global variable and can be accessed by multiple user threads at the same time it needs to be protected in a monitor. Let's define a monitor for it:

```
monitor Transactions {
   public var messageTXid = 0

   public function newTransaction() {
      ++messageTXid
   }
}

// make an instance of the monitor
var transactions = new Transactions()
```

Now instead of accessing the *messageTXid* directly it will have to be done through the *transactions* monitor.

16.3.5. Server control

Now that we have defined the classes for the rooms and users we are in a position to put it all together. The server needs to keep track of a number of things:

1. A list of all users who have logged in
2. A list of all the rooms that have been created
3. The current user session id

All these can be accessed by multiple threads at once. For example, when a user logs in, the userServer thread associated with a client receives the LOGIN message. The thread will create a user object and then it needs to add it to the list of logged in users. This means that all the data needed by the server globally has to be protected by a monitor. Let's define it.

```
monitor ServerControl {
   var loggedInUsers = new Vector()      // list of all logged in users
   var rooms = new Map()                 // map of room name vs Room object
public:

   function broadcast (command, t1, t2) {
      foreach room rooms {
         room.second.send (command, t1, t2)
      }
   }

   var numUsers = 0                 // number of logged in users

   function addUser (user) {
      loggedInUsers.append (user)
      ++numUsers
   }

   function removeUser (user) {
      try {
         loggedInUsers.erase (user)
```

```
            --numUsers
        } catch (e) {
        }
    }

    private var uid = 0

    function getID() {
        return ++uid
    }

    // given a room name see if it exists.  If it cannot be found, create it and add
    // it to the list of rooms
    function findRoom (name) {
        var room = null
        try {
            room = rooms.find (name)
        } catch (e) {                                   // exception if not found
            room = new Room (name)
            rooms.insert (name, room)
        }
        return room
    }


    // get all rooms
    function allRooms() {
        return rooms
    }

    // get all users
    function allUsers() {
        return loggedInUsers
    }



    function listUsers (stream) {
        "Current user are:\n" -> stream
        foreach room rooms {
            room.second.listUsers (stream)
        }
        System.flush (stream)
    }

    function listRooms (stream) {
        "Available rooms:\n" -> stream
        foreach room rooms {
            room.second.showName (stream)
        }
    }

   // a few other functions
}

// an instance of the ServerControl monitor
```

```
var control = new ServerControl()
```

The list of all rooms is held in an instance of the class monitor Map(). This is a protected interface on top of the system defined map type. It is provided in the System library and therefore must be imported:

```
import map
```

16.3.5.1. Shutting the server down

There needs to be a way to shut the server down gracefully, Yes, we could just ^C the program but that would be unfriendly to the uses who are logged in. The SHUTDOWN command can be sent from the client to shut down the server. This is a protected command only available to the superuser ("root").

The shutdown command is supposed to give the users a grace time to allow them to say goodbye to their friends before the world is nuked below their feet. The simplest way to shutdown is just to call System.exit(). If we want to give a time for shutdown we need to start a thread that sleeps for a period of time and then calls System.exit().

```
thread shutdown (control, time : int, reason) {
   var microseconds = time * 1000000
   sleep (microseconds)
   control.broadcast ("SHUTDOWN", "now", reason)
   sleep (1000000)
   System.exit (0)
}
```

The thread is passed the control object so that it can broadcast messages. It is also passed the time in seconds and a reason for shutdown. Notice that the *time* parameter has been given a type of int. This is to save us casting the incoming string from the command to an integer before calling the shutdown thread.

The thread sleeps for the specified amount of time then broadcasts a message to all users telling them that the server is shutting down now. Then it sleeps for another second (to give the broadcast time to propagate) and the calls System.exit() thus stopping the program.

There is no way provided to stop a shutdown once it has started. Kind of like starting an auto self-destruct sequence on the Enterprise!

16.3.6. Executing user commands

We have laid the foundations for the server. Now we just need to process the command sent by the user by making calls to the functions and classes we have defined.

Let's revisit the userServer() thread. This is a thread that is spawned when a new connection arrives from a client. It runs until the stream from the client is closed.

The thread will receive a set of commands from the client. The first command it will receive is the LOGIN command. What we should do here is to create a new user object and add him to the main list of logged in users. Let's define a couple of new variables inside the userServer thread:

```
var user = null                    // user object
var rooms = new Vector()                  // all the rooms occupied by the user
```

Now we can write the code for the LOGIN command. Remember that the commands are processed by a switch statement with case limbs for each separate command:

```
switch (cmd) {
case "LOGIN":
   // code for login
    break
// more
}
```

The operation of a login is the following:

```
case "LOGIN":
   user = new User (nick, text, clientStream, control.getID())
   control.addUser (user)
   break
```

That is, create a new user object passing the parameters we received. The constructor for the user object validates that the password and nickname are correct. If all is well, we add the user to the server control monitor.

Let's now look at the JOIN command. This is sent by the client when a user wishes to join the conversation in a room:

```
case "JOIN":
   var room = control.findRoom (nick)          // find or create room
   room.join (user)                            // join the room
   rooms.append (room)                         // append to local list of rooms
   user.currentRoom = room                     // set as current room for talking
   break
```

We first find or create the room. The semantics of the findRoom() function in ServerControl is that it creates a room if it doesn't already exist. In any case it returns a valid room object. We then join the room and append it to our local list of rooms. The room is then set to be our current room where we will talk.

The main operation of the server is to send messages to users. A message originates at a client when the user types something. This is sent as a MESSAGE command to the server. The message is sent either to the room the user is currently talking in, or to all rooms occupied by the user. The variable currentRoom in the user object specifies where the message will go.

We have defined a concept of a super user. This is a user whose name is "root" and has privileges above and beyond those of a normal user. One such privilege is that any message sent by root is automatically sent to all the users in the system.

Let's look at the code for the MESSAGE command:

```
case "MESSAGE":
   if (user.isRoot()) {                        // root user?
      foreach room control.allRooms() {        // send to all users in all rooms
         room.second.send ("MESSAGE", nick, text)
      }
   } else {
      if (sizeof (rooms) == 0) {                          // check if we are in a room
         "ERROR:You are not in a room:x\n" -> clientStream
         System.flush (clientStream)
      }
```

```
        if (user.currentRoom != null) {                 // have we a current room?
            user.currentRoom.send (“MESSAGE”, nick, text)  // send to it only
        } else {
            foreach room rooms {                        // send to all rooms we occupy
                room.send (“MESSAGE”, nick, text)
            }
        }
        user.resetIdle()                                // reset idle counter
    }
    break
```

The rest of the command are very similar in nature.

16.3.7. Complete server program

Without further ado (drum roll please), here is the full listing of the complete chat server program.

```
package ChatServer {

    const version = "0.9"
    System.println ("Aikido chat server version " + version + " running...")

    import net                          // network support
    import streambuffer                 // utility buffering for networks
    import map                          // map objects
    import properties                   // property maps
    import vector                   // vector objects

    // this is a list of users that are logged in to the system, for validation purposes
    var loggedInUsers = new Vector()    // all logged in users

    // read the system properties file

    var props = new Properties()
    var file = System.openin ("chat.props")
    file -> props
    System.close (file)


    // a user has logged in, check that she has an account and that
    // the password is valid
    function validateUser (name, password) {
        var accounts = System.readfile ("chat.users")
        foreach account accounts {
            var expr = account["([^:]+):([^:]+):(.*)"]
            if (name == account[expr[1].start:expr[1].end]) {
                if (password != account[expr[2].start:expr[2].end]) {
                    throw "Invalid password"
                }
                return account[expr[3].start:expr[3].end - 1]
            }
        }
        throw "No such user"
    }
```

```
// read server address and port from the properties
var server = props["server"]
var port = cast<int>(props["port"])

// this holds the current message tranaction id to prevent multiple messages
// being sent to a single user when he is in multiple rooms.   Since the server
// is multithreaded and updates to this are done by the client threads, this needs
// to be protected by a mutex and is therefore a monitor

monitor Transactions {
   public var messageTXid = 0

   public function newTransaction() {
      ++messageTXid
   }
}

// instance of the transactions monitor
var transactions = new Transactions()


// the log file

var syslogfile = System.openup ("chat.log")

// log to a file
function log (stream, command, from, message) {
   if (message == "") {
      message = "xxx"
   }
   [command, ":", from, ":", message, "\n"] -> stream
   System.flush (stream)
}

// a Room.  This object represents a single chat room on the server.  It has a name
// and contains a set of users who are present in the room

class Room (name) {
   var users = new Vector()

   var logfile = System.openup ("room_" + name + ".log")

   // send a command to all users in the room
   public function send (command: string, from : string, message : string) {

      log (logfile, command, from, message)
      foreach user users {
         user.send (transactions.messageTXid, command, from, message)
      }
   }

   // allow a user to join this room
   public function join (user) {
      users.append (user)                                    // add the user
```

```
        send ("JOIN", user.getNick(), name)                    // tell other users
    }

    // a user is leaving
    public function leave (user) {
        send ("LEAVE", user.getNick(), name)      // tell the others she has gone
        users.erase (user)                                     // remove the user
    }

    // list all the users in the room to the stream
    public function listUsers (stream) {
        ["Room \"", name, "\":\n"] -> stream
        foreach user users {
            var idle = user.getIdleTime()
            ["    [", user.getID() , "] ", user.getNick(), " (", user.getName(), ") idle for ", idle, "
second", (idle == 1)?"":"s"] -> stream
            if (user.currentRoom == this) {
                " [currently talking here]" -> stream
            }
            '\n' -> stream
        }
    }

    public function showName (stream) {
        ["\t", name, "\n"] -> stream
    }

    public function getName() {
        return name
    }

}


//
// a User is someone who has an account and is logged on
//

class User (nick : string, password : string, outstream, id) {
    var name = validateUser (nick, password)
    public var currentRoom = null

    // return the nickname for the user
    public function getNick() {
        return nick
    }

    public function getName() {
        return name
    }


    public function getID() {
        return id
    }
```

```
    // this variable is a buffer for gathering output data
    var outputBuffer = new Streambuffer()

    // this is the transaction id of a set of messages
    var currentTXid = 0

    // send a command to the client via the stream
    public function send (TXid, command : string, from : string,message : string) {
        if (TXid != currentTXid) {
            outputBuffer.clear()
            outputBuffer.put (command + ":" + from + ":" + message + "\n")
            outputBuffer -> outstream
            currentTXid = TXid
        }
    }

    public function isRoot() {
        return nick == "root"
    }

    public function close() {
        System.close (outstream)
    }


    var lastMessage = System.time()              // last time user typed something

    public function getIdleTime() {
        return (System.time() - lastMessage) / 1000000              // idle time in seconds
    }

    public function resetIdle() {
        lastMessage = System.time()
    }
}

// shutdown after the given time in seconds
thread shutdown (control, time : int, reason) {
    var microseconds = time * 1000000
    sleep (microseconds)
    control.broadcast ("SHUTDOWN", "now", reason)
    sleep (1000000)
    System.exit (0)
}

// this monitor contains shared data and accesses to it.

monitor ServerControl {
public:
    // this variable holds all the rooms that have been created by users
    var rooms = new Map()

    // broadcast to all users
    function broadcast (command, t1, t2) {
        foreach room rooms {
            room.second.send (command, t1, t2)
```

```
    }
}

var numUsers = 0

function stats() {
    broadcast ("STATS", numUsers, sizeof (rooms))
}

// add a user to the list of those logged in
function addUser (user) {
    loggedInUsers.append (user)
    ++numUsers
}

// remove a user from the logged in list
function removeUser (user) {
    try {
        loggedInUsers.erase (user)
        --numUsers
    } catch (e) {
    }
}

var uid = 0                    // unique id for a user

// get the next user id
function getID() {
    return ++uid
}

// find a room given its name.  If no room exists, create it
function findRoom (name : string) {
    var room = null
    try {
        room = rooms.find (name)
    } catch (e) {
        room = new Room (name)
        rooms.insert (name, room)
    }
    return room
}

function allRooms() {
    return rooms
}

// list all the users logged into all rooms
function listAllUsers (stream) {
    "Current users are:\n" -> stream
    foreach room rooms {
        room.second.listUsers (stream)
    }
    System.flush (stream)
}
```

```
// list all the available rooms
function listRooms (stream) {
    "Available rooms:\n" -> stream
    foreach room rooms {
        room.second.showName (stream)
    }
}


// kick a given user off the system
function kickUser (nick, reason) {
    var user = null
    foreach u loggedInUsers {
        if (u.getNick() == nick) {
            user = u
            break
        }
    }
    if (user == null) {
        throw "User " + nick + " is not logged in"
    }
    log (syslogfile, "KICKED", "root", reason)
    user.send (transactions.messageTXid, "KICK", "root", reason)
    user.close()
}

// kill a user given his id
function killUser (nick, id) {
    var user = null
    foreach u loggedInUsers {
        if (u.getID() == id) {
            if (nick == "root" || nick == u.getNick()) {
                user = u
                break
            }
        }
    }
    if (user == null) {
        throw "Cannot kill user [" + id + "]"
    }
    log (syslogfile, "KILLED", nick, "x")
    user.send (transactions.messageTXid, "KILL", nick, "x")
    user.close()
}

generic shutdownThread                    // stream talking to shutdown thread
var shuttingDown = false

function startShutdown (time, reason) {
    if (shuttingDown) {
        return
    }
    broadcast ("SHUTDOWN", time, reason)
    if (time == "now") {
        System.exit (0)
    } else {
```

```
            shutdownThread = shutdown (this, time, reason)                    // keep stream
in case we cancel
          }
          shuttingDown = true
       }

    }

    // instance of the ServerControl monitor
    var control = new ServerControl()

    //
    // one of these is created for each user in the server.  The thread is started
    // when the user's client first connects.  The client will send messages
    // to the thread through the clientStream stream.  These messages are text
    // based and will drive the functionality of the user server
    //

    thread userServer (clientStream) {
       var user = null                                          // user object associated with
this user
       var rooms = new Vector()                        // rooms the user is in

       // process an incoming client command
       function clientCommand(command) {

          // the command consists of:
          //   series of chars ending in colon (command name)
          //   series of chars ending in colon (nickname)
          //   series of zero or more chars (text)
          // the regular expression matches these 3 subexpressions

          var commandexpr = command["([^:]+):([^:]+):(.*)"]

          if (sizeof (commandexpr) > 3) {              // need at least 3 expressions
             var cmd = command[commandexpr[1].start : commandexpr[1].end]
           // command name
             var nick = command[commandexpr[2].start : commandexpr[2].end]
           // nickname
             var text = command[commandexpr[3].start : commandexpr[3].end]
           // text

             switch (cmd) {
             case "LOGIN":
                log (syslogfile, "LOGIN", nick, "")
                user = new User (nick, text, clientStream, control.getID())
                control.addUser (user)
                //control.stats()
                break

             case "LOGOUT":
                log (syslogfile, "LOGOUT", nick, "")
                control.removeUser (user)
                //control.stats()
                return false
```

```
case "JOIN":
    var room = control.findRoom (nick)
    room.join (user)
    rooms.append (room)
    user.currentRoom = room
    //control.stats()
    break

case "TALK":                        // talk in a room
    if (nick == "*") {
        user.currentRoom = null
        ["TALKING:*:x\n"] -> clientStream
        System.flush (clientStream)
        break
    }
    var found = false
    foreach room rooms {
        if (room.getName() == nick) {
            user.currentRoom = room
            ["TALKING:", nick, ":x\n"] -> clientStream
            System.flush (clientStream)
            found = true
            break
        }
    }
    if (!found) {
        ["ERROR:No such room ", nick, ":x\n"] -> clientStream
        System.flush (clientStream)
    }
    break

case "LEAVE":
    try {
        var room = control.findRoom (nick)
        room.leave (user)
        rooms.erase (room)
        if (room == user.currentRoom) {
            if (sizeof (rooms) == 0) {
                user.currentRoom = null
                ["TALKING:*:x\n"] -> clientStream
            } else {
                user.currentRoom = rooms[0]
                ["TALKING:", user.currentRoom.getName(), ":x\n"] -> clientStream
            }
            System.flush (clientStream)
        }
    } catch (e) {
        ["ERROR:Can't leave ", nick, ":x\n"] -> clientStream
        System.flush (clientStream)
    }
    break

case "MESSAGE":
    if (user.isRoot()) {
        foreach room control.allRooms() {
            room.second.send ("MESSAGE", nick, text)
```

```
                }
            } else {
                if (sizeof (rooms) == 0) {
                    "ERROR:You are not in a room:x\n" -> clientStream
                    System.flush (clientStream)
                }
                if (user.currentRoom != null) {
                    user.currentRoom.send ("MESSAGE", nick, text)
                } else {
                    foreach room rooms {
                        room.send ("MESSAGE", nick, text)
                    }
                }
                user.resetIdle()
            }
            break

        case "BROADCAST":
            foreach user loggedInUsers {
                user.send (transactions.messageTXid, "MESSAGE", nick, text)
            }
            break

        case "MOTD":
            try {
                var motd = System.openin ("chat.motd")
                motd -> clientStream
                System.close (motd)
                System.flush (clientStream)
            } catch (e) {
            }
            break

        case "USERS":
            control.listAllUsers(clientStream)
            break

        case "ROOMS":
            control.listRooms (clientStream)
            "\nYou are present in\n" -> clientStream
            foreach room rooms {
                room.showName (clientStream) ;
            }
            if (user.currentRoom == null) {
                "** you are currently talking in all rooms\n" -> clientStream
            } else {
                ["** you are currently talking in ", user.currentRoom.getName(), '\n'] ->
clientStream
            }
            System.flush (clientStream)
            break

        case "KICK":                    // kick a user off the system
            if (user.isRoot()) {
                control.kickUser (nick, text)
            } else {
```

```
                "ERROR:Access denied:x\n" -> clientStream
                System.flush (clientStream)
            }
            break

        case "KILL":                    // kill a user given its id
            try {
                control.killUser (nick, text)
            } catch (e) {
                ["ERROR:", e, ":x\n"] -> clientStream
                System.flush (clientStream)
            }
            break

        case "SHOWLOG":
            function showlog (room) {
                try {
                    var tail = "/usr/bin/tail -" + text + " room_" + room.getName() + ".log"
                    //System.println (tail)
                    var lines = System.system (tail)
                    lines -> clientStream
                } catch (e) {
                    ["ERROR:", e, ":x\n"] -> clientStream
                    System.flush (clientStream)
                }
            }
            if (nick == "*") {
                foreach room rooms {
                    showlog (room)
                }
            } else {
                room = control.findRoom (nick)
                showlog (room)
            }
            "ENDLOG:x:x\n" -> clientStream
            System.flush (clientStream)
            break

        case "CLEARLOG":
            foreach room control.allRooms() {
                System.system ("/usr/bin/cp /dev/null room_" + room.second.getName() +
".log")
            }
            break

        case "SHUTDOWN":
            if (user.isRoot()) {
                control.startShutdown (nick, text)
            } else {
                "ERROR:Access denied:x\n" -> clientStream
                System.flush (clientStream)
            }
            break

        }
    }
```

```
            return true
        }

        var command = ""                        // variable to hold incoming command
        while (!System.eof(clientStream)) {
            clientStream -> command                     // read the command
            transactions.newTransaction()       // generate a new transaction for it
            try {
                if (!clientCommand(command)) {          // process the command
                    break                       // logged out?
                }
            } catch (error) {                   // uncaught error in command
                System.println (error)          // print it to screen
                ["ERROR:", error, ":x\n"] -> clientStream       // send it to client
                System.flush (clientStream)
                break                                   // stop loop
            }
        }
        foreach room rooms {                            // leave all the rooms
            room.leave (user)
        }
        control.removeUser (user)               // remove the user
        System.close (clientStream)                     // close the stream to the client
        //control.stats()
    }

    // open the network server port and start accepting connections from
    // clients

    var now = System.date()
    ["Started ", now.mon+1, "/", now.mday, "/", now.year+1900, " "] -> syslogfile
    [now.hour, ":", now.min, ":", now.sec, "\n"] -> syslogfile
    System.flush (syslogfile)

    var socket = Network.openServer (server, port, Network.TCP)         // create server
socket
    for (;;) {
        var s = Network.accept (socket)                 // accept a connection
        userServer (s)                                  // start a server thread for it
    }
}
```

16.4. Client

Now for the client that talks to the server. The client is the software that has the following tasks:

- Connect to the server
- Get the user's password
- Accept input from the keyboard and send it to the server
- Accept input from the server and send it to the screen
- Control the screen output
- Interpret several user commands to allow control of client

Let's start at the top level. The client is a class and will need network support.

```
class ChatClient (nickame) {
   import net
   import streambuffer

   // client code
}
```

The client package takes a single parameter – the nickname of the user. The client will usually read the nickname from a properties file, but this parameter allows the default nickname to be overridden. This is very useful for logging on as someone else temporarily (particularly as root).

16.4.1. Get the client properties

The client needs a set of properties that are used to control it. These are things like:

- The nickname to use to log in to the server
- Optionally the password to use
- The machine and port on which the server is running
- A set of rooms to join initially

Let's read the properties from a file called 'chat.props' in the user's home directory:

```
var home = System.getenv ("HOME")              // get home dir

var userprops  = new Properties()
var userfile = System.openin (home + "/chat.props")
userfile -> userprops
System.close (userfile)

// now read the properties
var server = userprops["server"]
var port = cast<int>(userprops["port"])
var nick = nickname == "" ? userprops["nick"] : nickname
```

16.4.2. Reading the user's password

A client needs to log in to the server with a password. Passwords are private data and are encrypted before sending to the server. The system-provided package Security provides simple methods to read and encrypt passwords. The user may put his password in the properties file if he wants to. If the client package was provided with a nickname then the password in the file is ignored as the nickname refers to another user:

```
function getPassword {
   const prompt = "Enter your chat password: "
   if (nickname != "") {                       // same user?
      return Security.getpassword (prompt)             // no, get password
   }

   var password = ""
   try {
      password = userprops["password"]         // look in properties
   } catch (e) {
```

```
        password = Security.getpassword (prompt)   // not present, ask user
    }
    return password
}

var password = getPassword()

// encrypt the password
password = Security.encrypt (password, "da")
```

The Security package contains 2 functions:

Function	Parameters	Purpose
getpassword	prompt	Print prompt to screen, switch off echo on terminal and read the password typed in by the user. Switch echo back on again
encrypt	string, salt	Encrypt the password using DES encryption. The salt string is used by the encryption routine and must be 2 characters long.

The result of the call to getPassword() and encrypt() is a string of unintelligible characters that bear no resemblance to the original text typed in by the user.

16.4.3. Get the default set of rooms

The user can provide a set of rooms to join when the client starts. These are held in a property with name 'rooms'. Let's read them:

```
generic rooms = []
try {
    rooms = userprops["rooms"]
} catch (e) {
}
```

The 'rooms' variable is generic because if there is only one value for the 'rooms' property then a string is returned, otherwise a vector of strings is returned.

16.4.4. Log on to the server

We now have all the information we need to connect and log on to the server. First we connect:

```
var serverStream = Network.open (server, port, Network.TCP)
```

Here we create a stream connection to the network. The function 'Network.open()' opens a network connection to the named network address and returns a stream we can read from and write to.

Once the stream to the server is opened we need to be able to read from it in order to receive messages. This is done by a thread called 'client' that is very similar to that in the server.

```
thread client {
    function print (text) {
        // print text to the screen
    }

    function serverCommand (command) {
```

```
        // process a command from the server
    }

    // read all messages from server
    var command = ""
    while (!System.eof (serverStream)) {
        serverStream -> command                    // read command from stream
        serverCommand (command)                    // process command
    }
    print ("*** server has closed connection, exiting\r\n")
    System.exit (0)
}
```

The purpose of the serverCommand() function is to process the command from the server. Commands coming this way (from the server to the client) are usually notifications of events. This may be an event of someone joining a conversation in a room, or a simple message sent to the client. The format of the commands coming to the client is the same as those going the other way. We use a regular expression to extract the fields and switch on the command.

The print() function is responsible for printing something to the screen. We'll delve into this later.

Now we can send commands to the server and receive them:

```
client()                    // start client thread

var buffer = new Streambuffer()          // utility buffer

// send LOGIN command to server
buffer.put ("LOGIN:")
buffer.put (nick + ":")
buffer.put (password + "\n")
buffer -> serverStream

// get the message of the day
buffer.clear()
buffer.put ("MOTD:x:x\n")
buffer -> serverStream

// function to join a room
function join (room) {
    buffer.clear()
    buffer.put ("JOIN:")
    buffer.put (room + ":xxx\n")
    buffer -> serverStream
}

//  join all the rooms we set up in props file
if (typeof (rooms) == "vector") {
    foreach room rooms {
        join (room)
    }
} else {
    join (rooms)                    // only one room
}

// get a list of all the users
```

```
buffer.clear()
buffer.put ("USERS:x:x\n")
buffer -> serverStream
```

In the code we performed the following tasks:

1. Started the client thread
2. Sent a LOGIN command to server
3. Sent a MOTD command to server to get the message of the day
4. Joined all the rooms we specified in the properties file
5. Sent a USERS command to get a list of all the users who are logged in

16.4.5. Getting input from user

The main purpose of a chat client is to read input from a user and communicate with the chat server. The user types messages on the keyboard and those messages are either interpreted as a command to control the client or are sent to the server as a text message to be sent to the other users. Let's write the code:

```
function clientCommand (command) {
   if (sizeof (command) == 0) {              // ignore empty string
      return
   }

   buffer.clear()                            // buffer cleared and ready for use
   if (command[0] == '/') {                  // client-side command?
      // process client command
   } else {                                  // regular message
      buffer.put ("MESSAGE:")                // form MESSAGE command
      buffer.put (nick + ":")
      buffer.put (command + "\n")
      buffer -> serverStream                 // send it
   }
}
```

The things typed by the user are either a client command or a regular message. A client command is signified by the initial character being a slash (/). Anything else is a regular message.

Now we need to be able to read from the keyboard and call this function for everything typed:

```
for (;;) {
   var msg = ""
   stdin -> msg
   try {
      clientCommand (msg)
   } catch (e) {
      // print error to screen
   }
}
```

This is the main loop of the chat client.

16.4.6. Screen input and output

A commercial chat client will use a graphical user interface for its input and output. It will have 2 windows, one for input and one for output. That is too complex for this example, to we will use a regular text based terminal for our client.

All modern terminals (windows on a screen usually) support cursor motion commands through the use of escape codes sent to the terminal. We will make use of the cursor motion command to divide our screen into 2 regions: one for input and one for output.

It is beyond the scope of this example to describe the implementation of such a screen control package but let's assume the existence of a package called TTY (old abbreviation for Teletype). This package contains the following facilities:

Function	Parameters	Purpose
print	string	print to the current position on the screen
printAt	row, column, string	Print the string at the given position
close		close the terminal control session
goto	row, column	move cursor to given row and column
clearLine		clear the current line on the screen
clearScreen		clear the whole screen
getline	top, bottom, left, right	read a line from the keyboard and limit it echo to the rectangle specified by the parameters

The TTY package also provides 2 variables: *rows* and *cols*. These are the number of rows and columns on the screen respectively.

Further, let's define a simple windowing abstraction on top of the TTY package. This will allow us to divide the screen into the regions we need. Let's call the class "Windows" (I'm sure I've heard that name somewhere before, but just can't quite place it).

```
class Windows (tty) {
   import tty                                          // out little TTY package

   class Window (protected top, protected bottom,
                 protected left, protected right) {
   public:
      function clear() {                               // clear the window
         for (var l = top ; l <= bottom ; l++) {
            tty.goto (l, 0)
            tty.clearLine()
         }
      }
   }


   // an output window represents a region that is used for output.  The region scrolls
   // if necessary to fit the output
   class OutputWindow (t, b, l ,r) extends Window (t, b, l, r) {
      function printLine (line) {
         // print a line to the screen, scrolling if necessary
      }

      public operator -> (data, isout) {
         if (!isout) {
            throw "Output window cannot be used for input"
```

```
            } else {
                switch (typeof (data)) {
                case "vector":
                    var str = ""
                    foreach item data {
                        str += item
                    }
                    printLine (str)
                    break
                default:
                    printLine (data)
                }
            }
        }

        // an input window is where the user types.  It is limited to a region on the screen
        class InputWindow (t, b, l, r) extends Window (t, b, l, r) {
            public operator -> (stream, isout) {
                if (isout) {
                    throw "Cannot use input window as output"
                } else {
                    tty.getline (top, bottom, left, right) -> stream
                }
            }
        }


        function close() {
            tty.close()
        }

    }
```

Both the *OutputWindow* and *InputWindow* classes provide an instance of the stream operator. This allows to stream data to and from the windows like any other stream device.

We can now write the code to use the screen package. We decide to split the screen into 3 regions:

1. A title window at the top of the screen showing information about the current chat session
2. An output window
3. A 2 line input window at the bottom of the screen

```
var title = null
var out = null
var in = null
var tty = new TTY()
var windows = new Windows (tty)

try {
    tty.clearScreen()
    title = new Windows.OutputWindow (0, 1, 0, tty.cols)
    out = new Windows.OutputWindow (2, tty.rows – 4, 0, tty.cols)
    in = new Windows.InputWindow (tty.rows – 2, tty.rows – 1, 0, tty.cols)
} catch (e) {
    windows.close()
    throw e
```

```
}
```

Now we can write to the output window by:

```
line -> out
```

and read from the input window:

```
in -> msg
```

Remembering to trap any exceptions and close the TTY. If we fail to close the TTY before exiting the program the terminal will probably be broken as the TTY handler will set it in raw mode.

16.4.7. Complete client program

Here is a complete implementation of a client program. It contains slightly more functionality than the simple client just described. This program is actually in use as I type this. The client presented also includes a lot of features such as color handling and window resizing. It is left as an exercise to the reader to follow the code for these.

First, the Windows class:

```
class Windows (tty) {
public:

   class Line (public text, public color = 0) {
      public operator sizeof() {
         return sizeof (text)
      }
      public operator[] (i, j = -1) {
         if (j >= 0) {
            return text[i:j]
         } else {
            return text[i]
         }
      }
   }

   class Window (protected top, protected bottom, protected left, protected right) {
   public:
      function clear() {
         tty.grabCursor()
         for (var l = top ; l <= bottom ; l++) {
            tty.goto (l, 0)
            tty.clearLine()
         }
         tty.releaseCursor(true)
      }

      function resize (t, b, l, r) {
         top = t
         bottom = b
         left = l
         right = r
      }
```

```
}


class OutputWindow (t, b, l, r) extends Window (t, b, l, r) {
   var nlines = 0                          // number of filled lines
   var height = bottom - top
   var width = right - left
   var col = 0                     // current column

   const MAXLINES = 1000
   var lines = []                          // all lines output to window
   var scrolledlines = 0                   // distance we have scrolled
   var topline = 0                         // index of line at top of window
   var bottomline = topline + height - 1           // index of line at bottom of window

public:
   function scroll (direction, nl) {
      if (nlines < height) {
         return
      }
      switch (direction) {
      case 0:
         if (topline == 0) {
            return
         }
         if (nl > topline) {
            nl = topline
         }
         topline -= nl
         bottomline -= nl
         tty.scroll (tty.DOWN, top, bottom, nl)
         scrolledlines += nl
         foreach i nl {
            tty.grabCursor()
            tty.goto (top + i, left)
            tty.clearLine()
            tty.releaseCursor(true)
            var line = lines[topline + i]
            tty.printAt (top + i, left, line.text, line.color)
         }
         break
      case 1:
         var disttobottom = sizeof (lines) - bottomline - 1
         if (disttobottom == 0) {
            return
         }
         if (nl > disttobottom) {
            nl = disttobottom
         }
         topline += nl
         bottomline += nl
         tty.scroll (tty.UP, top, bottom, nl)
         scrolledlines -= nl
         foreach i nl {
            tty.grabCursor()
            tty.goto (bottom - i - 1, left)
```

```
                tty.clearLine()
                tty.releaseCursor(true)
                var line = lines[bottomline - i]
                tty.printAt (bottom - i - 1, left, line.text, line.color)
            }
            break
        }
    }

    function printLine (line : Line) {
        var len = sizeof (line)
        var ln = top + nlines
        var lenlines = len == 0 ? 1 : (len - 1) / width + 1
        var remaining = len

        function printLines (line) {
            if (len == 0) {
                new Line (" ") -> lines
                return
            }
            foreach n lenlines {
                var left = n * width
                var right = width
                if (right > remaining) {
                    right = remaining
                }
                var index = left + right - 1
                if (index >= len) {
                    return
                }
                var l = line[left : index]
                if (sizeof (lines) == MAXLINES) {
                    delete lines[0]
                }
                new Line (l, line.color) -> lines                    // append to
end of lines
                tty.printAt (ln + n, col, l, line.color)
                remaining -= right
                col = 0
            }
        }

        // if we have scrolled, scroll back to bottom
        if (scrolledlines != 0) {
            scroll (tty.DOWN, scrolledlines)
        }
        var linesleft = height - nlines
        if (linesleft < lenlines) {              // any room for the lines
            tty.scroll (tty.UP, top, bottom, lenlines)
            topline += lenlines
            bottomline += lenlines
            ln = bottom - lenlines
            printLines (line)
        } else {
            printLines (line)
            nlines += lenlines
```

```
        }
    }

    function refresh() {
        for (var i = 0 ; i < nlines ; i++) {
            tty.printAt (top + i, 0, lines[topline + i].text, lines[topline + i].color) ;
        }
    }

    // print a short string on the screen in bold and leave the cursor at the end
    function printBold (text : string) {
        tty.setColor (3)
        var ln = top + nlines
        var linesleft = height - nlines
        if (linesleft < 1) {            // any room for the lines
            tty.scroll (tty.UP, top, bottom, 1)
            ln = bottom - 1
            tty.printAt (ln, col, text)
        } else {
            tty.printAt (ln, col, text)
        }
        col += sizeof (text)
        tty.setColor (0)
    }

    operator -> (data, isout) {
        if (!isout) {
            throw "Cannot use OutputWindow as input"
        } else {
            switch (typeof (data)) {
            case "vector":
                var str = ""
                foreach item data {
                    str += item
                }
                printLine (new Line (str))
                break
            default:
                printLine (data)
            }
        }
    }

    function resize (t,b,l,r) extends resize (t,b,l,r) {
        var vdiff = (bottom - top) - height
        var hdiff = (right - left) - width
        if (vdiff < 0) {                        // shorter window?
            nlines += vdiff
        }
        height = bottom - top
        width = right - left
    }

}

class InputWindow (t,b,l,r) extends Window (t,b,l,r) {
```

```
        public operator -> (stream, isout) {
            //tty.grabCursor()
            //tty.goto (t, l)
            //tty.print ("")
            //tty.releaseCursor(false) ;
            if (isout) {
                throw "Cannot use InputWindow as output"
            } else {
                tty.getline (top,bottom,left,right) -> stream
            }
        }

        public function resize (t,b,l,r) extends resize (t,b,l,r) {
            tty.resizeInput (t, b, l, r)
        }
    }


    function newOutputWindow (t, b, l, r) {
        return new OutputWindow (t,b,l,r)
    }

    function newInputWindow (t, b, l, r) {
        return new InputWindow (t,b,l,r)
    }

    function newLine (text, color = 0) {
        return new Line (text, color)
    }

    function close() {
        tty.close()
    }

    function getRows() {
        return tty.rows
    }

    function getCols() {
        return tty.cols
    }
}
```

Now the client code itself. Note that the client calls itself ‘pchat’.

```
// this class is a client to the chat server
class ChatClient (nickname) {
    import net                      // network
    import streambuffer             // buffering for network
    import properties               // properties object
    import security
    import tty
    import windows
```

```
   const version = "1.54"

   var home = System.getenv ("HOME")                    // get home directory

   // read in the properties file for the user
   var userprops = new Properties()
   var userfile = System.openin (home + "/pchat.props")  // open props file
   userfile -> userprops                                 // read props file
   System.close (userfile)

   // get the variables from the properties
   var nick = nickname == "" ? userprops["nick"] : nickname
   var me = nick

   function getPassword {
      const prompt = "Enter your chat password: "
      if (nickname != "") {
         return Security.getpassword (prompt)
      }
      var password = ""
      try {
         password = userprops["password"]                // in the clear, be sure of file
protections
      } catch (e) {
         password = Security.getpassword (prompt)
      }
      return password
   }

   var password = getPassword()

   password = Security.encrypt (password, "da")

   var server = userprops["server"]
   var port = cast<int>(userprops["port"])

   var attn = ""
   generic rooms = []
   var inactivity = 60 * 1000000
   var incolor = true
   var audio = true

   if (nickname == "") {
      try {
         attn = userprops["attn"]
      } catch (e) {
         attn = ""
      }

   // see if there are any default rooms to join
      try {
         rooms = userprops["rooms"]
      } catch (e) {                                    // may not be any rooms, ignore the
error
      }
```

```
    try {
        inactivity = cast<int>(userprops["inactivity"])
        inactivity *= 1000000                    // convert to microseconds
    } catch (e) {
    }

    try {
        var c = userprops["color"]
        if (c == "off") {
            incolor = false
        }
    } catch (e) {
    }

    try {
        var a = userprops["audio"]
        if (a == "off") {
            audio = false
        }
    } catch (e) {
    }
}

// open the network connection to the server
var serverStream = Network.open (server, port, Network.TCP)

// create a utility buffer
var buffer = new Streambuffer()

var logfile = stdout
var logging = false

// set up the terminal with 3 windows
var tty = new TTY()
var windows = new Windows (tty)

var out = null                  // output window
var in = null                   // input window
var title = null                // title window

var titlestart = 0

function drawTitle {
    // make a centered title string for the title window
    var titlestring = "PCHAT Version " + version
    var padstring = ""
    foreach i ((tty.cols - sizeof (titlestring)) / 2) {
        padstring += ' '
    }
    titlestart = sizeof (padstring)
    [padstring, titlestring] -> title
}

function drawMargins {
    try {
        tty.grabCursor()
```

```
        tty.goto (1, 0)
        foreach i tty.cols {
                tty.print ('=', false)
        }
        tty.goto (tty.rows - 3, 0)
        foreach i tty.cols {
                tty.print ('-', false)
        }
        tty.releaseCursor (true)
    } catch (e) {
        windows.close ()
        throw e
    }
}



// setup and initialize the windows
try {
    tty.clearScreen()

    title = windows.newOutputWindow (0, 1, 0, tty.cols)
    out = windows.newOutputWindow (2, tty.rows - 4, 0, tty.cols)
    in = windows.newInputWindow (tty.rows - 2, tty.rows -1, 0, tty.cols)

    drawMargins ()
    drawTitle()
} catch (e) {
    windows.close ()
    throw e
}

// resize of screen detected
function resize (r, c) {
    title.resize (0, 1, 0, c)
    out.resize (2, r - 4, 0, c)
    in.resize (r - 2, r - 1, 0, c)

    var oldr = tty.rows

    tty.resize()                    // tell tty handler to resize

    // delete the bottom margin
    tty.goto (oldr - 3, 0)
    tty.clearLine()

    tty.goto (0, 0)
    tty.clearLine()

    drawMargins()
    title.refresh()
    out.refresh()
}

// default colors
const BLACK = 0
```

```
const RED = 1
const GREEN = 2
const BEIGE = 3
const BLUE = 4
const MAGENTA = 5
const CYAN = 6
const GREY = 7

var nextColor = 0                       // color next user gets
var colormap = {}                       // map of user versus color
var ncolors = 8                         // number of colors

var colors = new [8]
try {
   var usercolors = userprops["colors"]                 // get colors property
   if (typeof (usercolors) == "vector") {           // more than one?
      ncolors = sizeof (usercolors)
      var i = 0
      foreach c usercolors {
         colors[i++] = cast<int>(c)
      }
   } else {                                         // only one color
      ncolors = 1
      colors[0] = cast<int>(usercolors)
   }
} catch (e) {
   foreach i 8 {
      colors[i] = i                                 // default colors
   }
}

// read any new color assignments and do the assignment
foreach i 8 {
   try {
      var c = userprops["color" + i]
      tty.mapColor (i, c)
   } catch (e) {
   }
}

// read the user color assignments
try {
   function setUserColor (usercolor) {
      var ex = usercolor["([^=]+)=(.+)"]
      if (sizeof (ex) == 3) {
         var user = usercolor[ex[1].start:ex[1].end]            // get user name
         var col = usercolor[ex[2].start:ex[2].end]             // get color as string
         var color = cast<int>(col)                             // get color as number
         colormap[user] = color                                 // set colormap
      }
   }
   var usercolors = userprops["usercolors"]
   if (typeof (usercolors) == "vector") {           // more than one?
      foreach c usercolors {
         setUserColor (c)
      }
```

```
    } else {                                    // only one color
        setUserColor (usercolors)
    }
} catch (e) {
}

function getColorForUser (nick) {
    var color = BLACK
    if (!incolor) {
        return color
    }
    if (typeof (colormap[nick]) == "none") {
        color = colors[nextColor]
        colormap[nick] = color
        nextColor = (nextColor + 1) % ncolors
    } else {
        color = colormap[nick]
    }
    return color
}

var callstring = "I want to talk to you"

try {
    callstring = userprops["call"]
} catch (e) {
}

var currentRoom = ""                    // the room I am talking in

function showCurrentRoom() {
    var c = tty.cols / 5                // one fifth of the way across
    var text = currentRoom
    var pad = titlestart - c - sizeof (currentRoom)                    // pad to start of
title
    if (pad > 0) {
        foreach i pad {
            text += " "
        }
    }
    tty.printAt (0, c, text, BLUE)
}

var aliases = {}                // map of alias name versus definition


function addAlias (name, defn) {
    aliases += {name = defn}
}

function showAliases() {
    foreach alias aliases {
        windows.newLine (alias.first + "\t\t" + alias.second) -> out
    }
}
```

```
   // print a list of supported commands
   function help() {
      function print (line) {
         try {
            windows.newLine (line) -> out
         } catch (e) {
            windows.close()
            throw e
         }
      }

      print ("Supported commands:")
      print ("      /join <room>              join the given room (room is created if it doesn't
exist)")
      print ("      /leave <room>             leave the given room")
      print ("      /talk [room]              talk only in the given room")
      print ("      /broadcast <message>    send to all users")
      print ("      /call <user>          call the specified user")
      print ("      /users                    show all logged in users")
      print ("      /rooms                    show all available rooms")
      print ("      /log                      toggle logging (currently " + (logging?"on":"off")
+ ")")
      print ("      /showlog [room [n]]       show last n server log entries for a given room")
      print ("      /invite <user email> <room>                invite user to chat room via
email")
      print ("      /kill <id>                kill a user id")
      print ("      /setcolor <c#> <name>   set a color number to the given color name")
      print ("      /alias [name text]      create a command alias (no paras = list aliases")
      if (nick == "root") {
         print ("   /kick <user> <reason>    kick a user off")
         print ("   /shutdown <time> [reason]         shutdown the server")
         print ("   /clearlog                 clear the server log")
      }
   }

   var soundoff = false

   function beep() {
      if (audio && !soundoff) {
         '\a' -> stdout
      }
   }

   var lastTimeReceived = System.time()                    // last time a message was
received

   thread clock() {
      for (;;) {
         var now = System.date()
         var timestring = "       " + (now.mon+1) + "/" + now.mday + "/" + (now.year + 1900)
         timestring += " " + now.hour + ":" + now.min + ":" + now.sec
         var timelen = sizeof (timestring)
         var timecol = tty.cols - timelen
         tty.printAt (0, timecol, timestring)

         // print current inactivity timer
```

```
        var time = System.time() - lastTimeReceived
        timestring = cast<string>(time/1000000) + "    "
        var color = BLACK
        if (incolor && time > inactivity) {
            color = RED
        }
        tty.printAt (0, 0, timestring, color)

        sleep (1000000)             // 1 second
    }
}

function showStats (numUsers, numRooms) {
    tty.printAt (0, 10, "" + numUsers + "/" + numRooms)
}

// thread for receiving messages from the server
thread client() {

    // print to the output window and the log file (if logging is enabled)
    function print (text, color = BLACK) {
        try {
            if (!incolor) {
                color = BLACK
            }
            windows.newLine (text, color) -> out                    // to output window
        } catch (e) {
            windows.close()
            throw e
        }
        if (logging) {
            [text, "\n"] -> logfile                 // to log file
            System.flush (logfile)
        }
    }

    // process an incoming command from the server
    function serverCommand (command) {
        var commandexpr = command["([^:]+):([^:]+):(.*)"]
        if (sizeof (commandexpr) > 2) {
            var cmd = command[commandexpr[1].start : commandexpr[1].end]
            var nick = command[commandexpr[2].start : commandexpr[2].end]
            var text = command[commandexpr[3].start : commandexpr[3].end]
            switch (cmd) {
            case "JOIN":
                if (nick == me) {                       // always talk in most recent room
                    currentRoom = text
                    showCurrentRoom()
                }
                beep()
                print ("** " + nick + " has joined " + text, GREEN)
                break
            case "LEAVE":
                beep()
                print ("** " + nick + " has left " + text, BLUE)
                break
```

```
case "MESSAGE":
   var color = getColorForUser (nick)
   var who = nick + " says: "
   if (attn != "") {
      if (sizeof (text[attn]) != 0) {
         beep() ; beep()
      }
   }
   var now = System.time()
   if ((now - lastTimeReceived) > inactivity) {
      var date = System.date()
      var prefix = "at " + date.hour + ":" + date.min + ":" + date.sec + ", "
      who = prefix + who
      beep()
   }
   lastTimeReceived = now
   print (who + text, color)
   break
case "ERROR":
   beep()
   print ("** Error: " + nick, RED)
   break
case "KICK":
   beep()
   beep()
   beep()
   print ("** You have been kicked off by " + nick + " because: " + text, RED)
   break
case "KILL":
   beep()
   beep()
   beep()
   print ("** You have been killed by " + nick, RED)
   break
case "LOGIN":
   print (nick + " logged in")
   break
case "LOGOUT":
   print (nick + " logged out")
   break
case "SHUTDOWN":
   beep()
   beep()
   if (nick == "now") {
      print ("*** server shutting down immediately: " + text)
   } else {
      print ("*** server shutting down in " + nick + " seconds: " + text)
   }
   break
case "ENDLOG":
   print ("--- END OF LOG ---", MAGENTA)
   soundoff = false
   break
case "STATS":
   showStats (nick, text)
   break
```

```
            case "TALKING":
                if (nick == "*") {
                    windows.newLine ("** you are now talking all rooms") -> out
                    currentRoom = "*all*"
                } else {
                    windows.newLine ("** you are now talking in " + nick) -> out
                    currentRoom = nick
                }
                showCurrentRoom()
                break
            default:
                print (command)
            }
        } else {
            print (command)
        }
    }

    // main thread loop, read and process messages from the server
    var message = ""
    while (!System.eof (serverStream)) {
        serverStream -> message
        serverCommand (message)
    }
    print ("***** server has closed connection, exiting\r\n", RED)
    windows.close()
    System.exit (0)
}

thread sizer() {
    var prevrow = -1
    var prevcol = -1
    var row = 0
    var col = 0
    var resizeimminent = false
    try {
        for (;;) {
            sleep (500000)                          // half a second
            if (resizeimminent) {
                if (prevrow == row && prevcol == col) {          // stabalized?
                    resize (row, col)
                    resizeimminent = false
                }
            }
            if (tty.getScreenSize (row, col)) {
                prevrow = row
                prevcol = col
                resizeimminent = true
            }
        }
    } catch (e) {
        windows.close()
        throw e
    }
}
```

```
client()                        // start the client thread
clock()                         // start the clock thread
sizer()                         // screen size monitor

// log in to the server
buffer.put ("LOGIN:")
buffer.put (nick + ":")
buffer.put (password + "\n")
buffer -> serverStream

// get the Message of the Day
buffer.clear()
buffer.put ("MOTD:x:x\n")
buffer -> serverStream

// function to join a room
function join (room) {
   buffer.clear()
   buffer.put ("JOIN:")
   buffer.put (room + ":xxx\n")
   buffer -> serverStream
}

// any rooms to join now?
if (typeof (rooms) == "vector") {                   // multiple rooms?
   foreach room rooms {
      join (room)
   }
} else {
   join (rooms)
}

// get a list of users who are logged on
buffer.clear()
buffer.put ("USERS:x:x\n")
buffer -> serverStream

// process a command typed in by the user
function clientCommand (command) {
   if (sizeof (command) == 0) {                          // blank line?
      return
   }

   var curindex = 0
   function skipspaces() {
      while (curindex < sizeof (command) && command[curindex] == ' ') {
         ++curindex
      }
   }

   // get a string from the input line
   function getString(stopatspace = true) {
      var str = ""
      skipspaces()
      while (curindex < sizeof (command)) {
         if (stopatspace && command[curindex] == ' ') {
```

```
            break
          }
          str += command[curindex++]
        }
        return str
      }

      // given a command and a set of arguments, look to see if it is an alias.  If
      // so, replace the command with the (resursive) alias defintion and replace the
      // arguments ($1, $2) etc with the actual parameters.

      function expandCommand (ci) {
        var cmd = getString()                    // get first word in command
        var alias = aliases[cmd]
        if (typeof (alias) != "none") {          // alias?
          var args = []
          var arg0 = ""
          for (var i = curindex ; i < sizeof (command) ; i++) {    // make arg0
            arg0 += command[i]
          }
          arg0 -> args
          while (curindex < sizeof (command)) {          // collect the args
            getString() -> args
          }
          var newcmd = ""                                          // new command is
placed here
          for (var i = 0 ; i < sizeof (alias) ; i++) {
            if (alias[i] == '$') {
              i++
              switch (alias[i]) {
              case '$':
                newcmd += '$' ;
                break ;
              default:
                if (alias[i] < '0' || alias[i] > '9') {
                  newcmd += alias[i]
                } else {
                  newcmd += args[alias[i] - '0']
                }
              }
            } else {
              newcmd += alias[i]
            }
          }
          command = newcmd                       // commit new command
          curindex = 0
          var ncmd = getString()                 // get first word (to prevent infinite
recursion)
          curindex = 0
          if (ncmd != cmd) {                     // not the same?
            expandCommand(0)                     // expand aliases in new command
          }
        } else {
          curindex = ci
        }
      }
```

```
buffer.clear()
// system commands start with slash
if (command[0] == '/') {
   curindex = 1
   expandCommand(1)                          // expand any alias in command

   var cmd = getString()
   switch (cmd) {
   case "alias":
      var name = getString()
      if (name == "") {
         showAliases()
      } else {
         var defn = getString (false)
         addAlias (name, defn)
      }
      break

   case "log":
      if (!logging) {
         var logfilename = home + "/pchat.log"
         logfile = System.openout (logfilename)
         windows.newLine ("Logging output to "+ logfilename) -> out
         logging = true
      } else {
         System.close (logfile)
         windows.newLine ("Logging off") -> out
         logging = false
      }
      break
   case "join":
      buffer.put ("JOIN:")
      var room = getString(false)
      if (room == "") {
         throw "Must supply name of room to join"
      }
      buffer.put (room)
      buffer.put (":xxx\n")
      windows.newLine ("joining room " + room) -> out
      buffer -> serverStream
      break
   case "leave":
      buffer.put ("LEAVE:")
      var room = getString(false)
      if (room == "") {
         throw "Must supply name of room to leave"
      }
      buffer.put (room)
      buffer.put (":xxx\n")
      buffer -> serverStream
      break

   case "talk":
      buffer.put ("TALK:")
```

```
    var room = getString(false)
    if (room == "") {
        room = "*"
    }
    buffer.put (room)
    buffer.put (":xxx\n")
    buffer -> serverStream
    break

case "broadcast":
    buffer.put ("BROADCAST:" + nick +":")
    var text = getString (false)
    if (text == "") {
        break
    }
    buffer.put (text)
    buffer.put ("\n")
    buffer -> serverStream
    break

case "call":
    buffer.put ("BROADCAST:" + nick +":")
    var who = getString()
    if (who == "") {
        break
    }
    buffer.put ("Calling " + who+ ": " + callstring)
    buffer.put ("\n")
    buffer -> serverStream
    break

case "setcolor":
    var colornum = getString()
    if (colornum == "") {
        throw "Specify color number"
    }
    var colorname = getString()
    if (colorname == "") {
        throw "specify color name"
    }
    tty.mapColor (cast<int>(colornum), colorname)
    break

case "help":
    help() ;
    break

case "kick":
    if (nick == "root") {
        buffer.put ("KICK:")
        var person = getString()
        if (person == "") {
            throw "Must supply name of person to kick"
        }
        var reason = getString (false)
        if (reason == "") {
```

```
                reason = "Just felt like it"
            }
            buffer.put (person + ":" + reason + "\n")
            buffer -> serverStream
        }
        break
    case "kill":
        buffer.put ("KILL:")
        buffer.put (nick + ":")
        var id = getString()
        if (id == "") {
            throw "Must supply id of user to kick"
        }
        buffer.put (id + "\n")
        buffer -> serverStream
        break
    case "users":
        buffer.put ("USERS:x:x\n")
        buffer -> serverStream
        break
    case "rooms":
        buffer.put ("ROOMS:x:x\n")
        buffer -> serverStream
        break
    case "invite":
        var who = getString()
        if (who == "") {
            throw "Invite who?"
        }
        var room = getString (false)
        if (room == "") {
            throw "Invite " + who + " to what room?"
        }
        System.system ("/usr/ucb/Mail -s \"Please join me in pchat room " + room + "\" "
+ who + " < /dev/null")
        windows.newLine ("Invitation sent to " + who + " to join you in " + room) -> out
        break
    case "showlog":
        buffer.put ("SHOWLOG:")
        var room = getString()
        if (room == "") {
          room = "*"
        }
        var lines = getString()
        if (lines == "") {
            lines = "50"
        }
        buffer.put (room + ":" + lines + "\n")
        buffer -> serverStream
        windows.newLine ("---- SERVER LOG ----") -> out
        soundoff = true
        break
    case "shutdown":
        if (nick == "root") {
            buffer.put ("SHUTDOWN:")
            var time = getString()
```

```
            if (time == "") {
                throw "Must supply shutdown time"
            }
            var reason = getString (false)
            if (reason == "") {
                reason = "Restarting"
            }
            buffer.put (time + ":" + reason + "\n")
            buffer -> serverStream
          }
          break
        case "clearlog":
          if (nick == "root") {
            buffer.put ("CLEARLOG:x:x\n")
            buffer -> serverStream
          }
          break
        default:
          throw "No such command"
        }
    } else {                                    // regular text, send to server
        buffer.put ("MESSAGE:")
        buffer.put (nick + ":")
        buffer.put (command + "\n")
        buffer -> serverStream
        lastTimeReceived = System.time()                // reset inactivity timer
    }
}

// process a scroll command
function scroll(cmd) {
    try {
        switch (cmd) {
        case 1:                 // up arrow
            out.scroll (tty.UP, 1)
            break
        case 2:                 // down arrow
            out.scroll (tty.DOWN, 1)
            break
        }
    } catch (e) {
        windows.close()
        throw e
    }
}

function refresh {
    try {
        tty.clearScreen()
        drawMargins ()
        title.refresh()
        //drawTitle()
        out.refresh()
    } catch (e) {
        windows.close()
        throw e
```

```
        }
    }

    // set the window title
    tty.setTitle ("PCHAT Version " + version)


    // read from the pchatrc file and execute the commands
    try {
        var rclines = System.readfile (home + "/.pchatrc")
        foreach line rclines {
            buffer.clear()
            try {
                if (sizeof (line) > 1 && line[0] != '#') {
                    clientCommand (line[0:sizeof(line) - 2])                    // remove newline from
end
                }
            } catch (error) {
                windows.newLine ("Error: "+  error) -> out
            }
        }
    } catch (e) {
    }


    // main client loop, read from keyboard and send to server
    for (;;) {
        buffer.clear()
        var msg = ""
        try {
            in.clear()                          // clear the input window
            in -> msg                           // read the text from input window
        } catch (e) {
            if (typeof (e) == "integer") {              // special output
                if (e == 0) {                           // EOF?
                    break
                } elif (e == 1 || e == 2) {                     // cursor motions
                    scroll(e)
                    continue
                } elif (e == 3) {                       // refresh
                    refresh()
                    continue
                }
                break
            } else {
                windows.close()                         // other exception, so put the terminal
back
                throw e                                 // and rethrow the exception
            }
        }
        try {
            clientCommand (msg)                         // process input
        } catch (error) {
            windows.newLine ("Error: "+  error) -> out
        }
    }
```

```
    windows.close()                    // user hit EOF, so reset terminal
}

// process optional single parameter (alternative username)

var nick = ""
if (sizeof (args) == 1) {
    nick = args[0]
} elif (sizeof (args) > 1) {
    throw "usage: pchat [nick]"
}

// run the client
ChatClient (nick)
```

Chapter 17. Aikido Debugger

Aikido provides a simple command-line debugger that can be used to help debug a program going awry. Those familiar with the Sun debugger (dbx) will find no difficulty getting to grips with the Aikido debugger as its command set is very similar to dbx.

The Aikido debugger is invoked by passing the option "-debug" to the aikido command:

% aikido –debug myprog.aikido

The Aikido interpreter has the code for the debugger built into it so there is no separate command needed to access the debugger. When passed the "-debug" flag, the interpreter will record information about all the symbols used in the program and present you with a command line prompt:

% aikido –debug myprog.aikido
aikido>

From this command line you can type debugger commands to control the program. A very useful command is *help*. The output from the help command is:

```
Aikido debugger
(C) Copyright 2000-2002 Sun Microsystems Inc.
David Allison, Sun Microsystems Laboratories

      quit                    exit the debugger
      history                 show command history (use with history
substitution)
      alias                   list all aliases
      alias name              print the definition of the named alias
      alias name defn         define the named alias
      unalias name            delete the named alias
      stop in <name> [if <expr>]     stop in named block
      stop at <line> [if <expr>]     stop at given line
      stop throw on|off       switch on or off break on exception
handling
      cont                    continue execution
      run                     run the program
      clear                   clear all breakpoints
      clear <bpnum>           clear the given breakpoint
      step                    single step one line
      step up                 continue until return
      nexti                   single step one instruction (over calls)
      stepi                   single step one instruction
      next                    single step one line (over calls)
      status                  show debugger status
      print <expr>            print value of expression
      set <expr>              execute expression
      call <expr>             call expression
      up                      move up stack one frame
      down                    move down one stack frame
      where                   show stack trace
      list [s] [e]            list lines from s to e
```

```
dis [n]                 print the next n (10) instructions
file <name>             select current file
files                   list all available files
threads                 list all threads
thread [n]              select thread n (or show current thread)
disable <bpnum>         disable breakpoint
enable <bpnum>          enable breakpoint
show <what>             show debugger things:
      blocks            show all non-system blocks
      allblocks         show all blocks
      breaks            show all breakpoints
      files             show all available files
      stack             show stack trace
      threads           show all threads
      vars              show all variables
```

This shows the list of commands that are available and a summary of the command's function.

17.1. Running the program

When the interpreter is first invoked with the "-debug" flag the program is not yet running. It has been parsed and found to be correct syntactically. To get the program running, use the ***run***, ***step*** or ***next*** commands. The ***run*** command starts the program running and it will not stop until it hits a breakpoint of the program completes. The ***step*** and ***next*** commands run the program until the first user-supplied statement is reached.

If the program is currently running and a new 'run' command is issued, the program will be restarted immediately,

17.2. Breakpoints

Breakpoints allow you to stop the program at designated points. Once the program has stopped you get a command prompt and can issue other debugger commands. You can set a breakpoint at the following locations:

1. The first statement of any block (function, class, package, etc)
2. Any line containing a statement
3. At any exception throw

Breakpoints are set using the ***stop*** command. There are 2 variants of it:

stop in <block name>
stop at <line number>

The first variant allows the program to be stopped at the first statement or a named block. The second sets a breakpoint at a particular line.

To stop when an exception is thrown, you can use the command 'stop throw on'

```
aikido> stop throw on
```

The program will stop when an exception is thrown. You can then examine variables and continue as normal.

Breakpoints may be disabled and enabled. When first set, a breakpoint is enabled and is shown in the 'status' listing as a number in parentheses. When a breakpoint is disabled it is shown in square brackets in

the ‘status’ listing. Disabling a breakpoint means that it will not trigger. The commands to enable and disable breakpoints are:

```
enable <bpnum>
disable <bpnum>
```

Where <bpnum> is the number of the breakpoint in the ‘status’ listing.

17.2.1. Stopping in a block

Setting a breakpoint in a block stops the program whenever the first statement of the block is about to be executed. You specify the block by naming it in the ***stop in*** command variant. A block name is the name of a function, class, monitor, package or thread. If you don’t know the name of the block you want to stop in, you can issue the ***show blocks*** (section xxx) command to see all the blocks defined in the program.

If the name of the block you specify in the ***stop in*** command variant is ambiguous the debugger will present you with a list of all the blocks that match the name. You can then choose one of them. For example, if you type the following command:

```
aikido> stop in toString
```

The debugger will respond with:

```
Ambiguous block name, choose one of the following:
 0) cancel
 1) main.System.Date.toString
 2) main.System.Exception.toString
 3) main.System.FileException.toString
 4) main.System.ParameterException.toString
>
```

You can then type the number corresponding to the block you choose. Typing 0 will cancel the stop command.

17.2.2. Stopping at a line

Another option for setting a breakpoint is to stop at a particular line in the program. In order to stop at a line, there must be executable code on that line. To set a breakpoint at a line use the ***stop at*** command variant:

```
aikido> stop at 356
```

But how do we specify which file to use? To do this you need to set the *current file* using the ***file*** command:

```
aikido> file genconst.aikido
```

Will set the current file to “genconst.aikido”. This file must be part of the program being debugged. In order to see what the valid files are you can use the ***show files*** command variant.

17.2.3. Conditional breakpoints

It is sometimes very useful to be able to set a conditional breakpoint. This is a breakpoint that is controlled by an expression. Each time the breakpoint is activated (a thread of control passes over it), the controlling expression is evaluated and if the result is non-zero the program stops. If the result of the expression is zero then the execution continues.

The expression can be any valid expression for the point of the program in which the breakpoint sits. You can use any variables that are in scope at that point.

The syntax for conditional breakpoints is simply to append the condition to the end of the ***stop*** command separated by an ***if*** clause:

```
aikido> stop at 156 if indentLevel > 0
```

This stops at line 156 if the variable *indentLevel* (in scope at line 156 of the program) has a value greater than 0.

A conditional breakpoint can be set for either the ***stop in*** or ***stop at*** command variants.

Setting a conditional breakpoint will cause the program to run slower if the breakpoint is activated frequently as the interpreter needs to evaluate an expression each time it is activated.

17.2.4. Clearing breakpoints

When a breakpoint is set, it is assigned a unique number. This number identifies the breakpoint for the purpose of displaying and clearing them. In order to clear a breakpoint you use the ***clear*** command.

This command can take 2 forms:

```
clear
clear <bpnum>
```

The first form (with no arguments) clears all the breakpoints set in the program. The second form clears a particular breakpoint specified by the breakpoint number. You can use the ***status*** or ***show breaks*** commands to see what breakpoints have been set.

17.3. Controlling execution

The main purpose of a debugger is to allow to control the execution of the program. The ***run*** (section 17.1) command allows a program to be run and *breakpoints* (section 17.2) allow it to be stopped. When you have a command line prompt you have access to other commands that are helpful in the debugging task.

17.3.1. Where am I?

When a program is stopped it is useful to be able to see where you are in the program. When running, the program will call functions, create objects, etc. A breakpoint may cause the program to stop at any point. By issuing the ***where*** or ***show stack*** command you can see the path the program took to get to the current location. For example:

```
aikido> where
=>[1] parseWordDef at line 127 in file genconst.aikido
  [2] parseLine at line 315 in file genconst.aikido
  [3] parseFile at line 334 in file genconst.aikido
```

```
   [4] main at line 353 in file genconst.aikido
```

This shows a stack trace after stopping at a breakpoint. The first line listed is the top of the stack. The marker (=>) shows the current location and may be changed using the ***up*** and ***down*** commands. Moving "up' the stack means moving down the list. If you issue the ***up*** command twice from the current location you will move up to the parseFile and move the marker to that point.

17.3.2. Threads

When you are stopped in the debugger, you are always in the context of a thread known as the 'current thread'. There may be other threads running in the program,, and all these are stopped when the debugger prompt is on the terminal To see what threads exist, the command 'threads' may be used. Each thread is given a number that can be used to refer to the thread. To switch threads, use the 'thread' command, passing the thread number to it.

When you switch threads, the stack is switched to the new current thread and all commands will work as expected.

The 'threads' command will display a list of all the threads that are currently executing in the program:

```
aikido> threads
   0        main(t.aikido:12)
   1        println (system.aikido:78)
 * 2        a(t.aikido:2)
   3        t(t.aikido:9)
```

The line in the display marked by an asterisk is the current thread (thread 2). All the commands that refer to the current state work in the context of the current thread. When the debugger is running at the prompt, all the threads in the program are halted.

To switch the current thread, use the 'thread' command:

```
aikido> thread 3
9:}
aikido> threads
   0        main(t.aikido:12)
   1        println (system.aikido:78)
   2        a(t.aikido:2)
 * 3        t(t.aikido:9)
```

17.3.3. Single stepping the program

Once stopped at a breakpoint you can continue execution using the ***cont*** command. This will cause the program to continue until it is stopped again at another breakpoint or it terminates. Another option of resuming execution is to issue a ***step*** or ***next*** command. These commands continue the execution of the program for one line. The difference between the ***step*** and ***next*** commands is that ***step*** will step into a called block whereas ***next*** will step over a called block. For example, if the current line is a function call statement then the ***step*** command will stop at the first line of the function being called and the ***next*** command will execute the whole function and stop at the line after the call is done.

Another option for single stepping is to step one *instruction* at a time. This can be achieved using the ***stepi*** and ***nexti*** commands. An *instruction* in the context of Aikido is one of the *Virtual Machine* instructions executed by the interpreter. This is really only useful for very low level debugging and requires an

understanding of the Aikido Virtual Machine instruction set (see Chapter 18 for details of the instruction set)

The ***stepi*** and ***nexti*** commands show the instruction that will be executed next.

17.3.4. Executing an expression

It is sometimes useful to be able to set the value of a variable or to call a function while inside the debugger. The commands available to do this are the ***set*** and ***call*** commands.

The ***set*** command takes an expression as its argument. Although any expression is allowed here, it is only useful to use an assignment expression. The expression is evaluated in the current context. That means that the current location is used to determine which variables are accessible for use in the expression. Moving around the stack by using the ***up*** and ***down*** commands changes this. For example, to set the value of a variable:

```
aikido> set connected = true
```

The ***set*** command may also be used to create new variables in the current scope. If the variable being set does not exist then it is created for you. This is very useful for assigning an alias for a complex expression to save on typing.

The ***call*** command is very similar to the ***set*** command in that it takes an expression as an argument. The intention is that the ***call*** command be used to call functions and other blocks in the program but this is not necessarily required. An example of the use of the ***call*** command:

```
aikido> call System.println (connected)
```

This calls the system printer function to print the value of a variable. Like the ***set*** command, the current location determines what is available to be called. Any expression can be executed in the call command.

17.4. Displaying information

17.4.1. Printing expressions

It is useful to be able to show the value of variables in the program when stopped at a breakpoint. We've seen how to do this with the ***call*** command in section 17.3.4, but it would a tad inconvenient to have to type in a call to System.println for every expression we want to print. Also, System.println function cannot print the contents of an object.

The ***print*** command is a powerful command that evaluates an expression and prints its value in a meaningful way. The command may be used to print any value including instances of objects. If you print an object it will list all the members of the object and their associated value. It also prints a vector as a whole with each element indexed and numbered. Any expression can be printed:

```
aikido> print x
x = 123
Aikido> print vec[5:3]
vec[5:3] = [0] 4
[1] 5
[2] 6
Aikido> print 123 * 456
```

```
123 * 456 = 56088
```

The variables that may be used in the expression are those in scope in the current location.

17.4.2. Listing file contents

The contents of the current file may be listed to the terminal by use of the ***list*** command. Without any arguments, the ***list*** command lists from the current line forward by 10 lines or the end of file. With one argument, it lists from the specified line forward by 10 lines. With 2 arguments it lists the lines between the two lines specified.

17.5. Other commands

17.5.1. Aliases

Aliases allow commands to be defined in terms of other commands. This is useful for abbreviating frequently used commands. The alias command has 3 forms:

```
alias
alias name
alias name definition
unalias name
```

The first form lists all the aliases that have been set. The second form shows the definition (if any) for the named alias. The third form defines (or redefines) the named alias with the given definition.

The ‘unalias’ command deletes an alias.

17.5.2. History

The debugger maintains a history like the UNIX® shells. The bang character (!) is used to select a history line to be repeated. To see all the history lines, the command ‘history’ may be used.

```
aikido> history
0     stop in a
1     run
2     threads
3     cont
4     threads
5     history
```

To select a particular command to be run again, use the bang character (!) and append either the command number or the first couple of characters of the command. For example, the following are the same:

```
aikido> !3
aikido> !c
aikido> !co
```

The history substitution may be used anywhere in the command.

17.5.3. Show command

The ‘show’ command is a way to get the debugger to list things. There are a number of options:

```
show  blocks            show all non-system blocks
show  allblocks         show all blocks
show  breaks            show all breakpoints
show  files             show all available files
show  stack             show stack trace
show  threads           show all threads
show  vars              show all variables
```

The one to note is the ‘show vars’ command. This lists the names and values of all variables in the current stack frame and all parent frames. It is sometimes useful to simply list all the variables rather than using individual ‘print’ commands.

The ‘show allblocks’ command differs from the ‘show blocks’ command by listing all the blocks known to the system rather than just the user-defined blocks.

17.5.4. Disassembly

The ***dis*** command allows the internal virtual machine instructions to be disassembled. This is only of use to advanced users who have an understanding of the instructions. Please see Chapter 1 for details of the instruction set. The ***dis*** command prints the next ‘n’ instructions to the screen. The disassembly listing starts at the current instruction. The default value for ‘n’ is 10.

17.5.5. Quitting

The ***quit*** command exits the debugger and returns you to the operating system’s command prompt.

Chapter 18. Virtual Machine Instruction Set

The Aikido Virtual Machine instruction set is a simple code that is used internally by the Aikido interpreter. Each thread running in the interpreter has an instance of a Virtual Machine (VM). The VM is the engine that interprets the instructions in order to execute the program.

The instruction set is register based. This means that all instructions use a set of registers to get their data upon which they operate. Most instructions write a result of the operation to a destination. The destination may be another register or may be one of the variables used in the program.

18.1. Instruction set summary

Instruction	Arguments	Operation
MOV	dest, src	Move src to dest (check for type and constant assignment)
MOVC	dest, src	Move constant src to dest (no check for constant overwrite)
MOVF	dest, src	Move Forced src to dest (no checks done)
MOVO	dest, src	Move Override src to dest. Used for virtual function override
LD	dest, src	Load indirect via register. Src contains address
ST	dest, src, value	Store indirect via register and also place in dest
COPY	dest, src	Make a copy of src and place in dest (used for strings)
MKVECTOR	dest, nelems	Make a vector of nelems length. Elements are on stack
MKMAP	dest, nelems	Make a map of nelems length. Elements are in pairs on stack
ADD	dest, src1, src2	dest = src1 + src2
SUB	dest, src1, src2	dest = src1 – src2
MUL	dest, src1, src2	dest = src1 * src2
DIV	dest, src1, src2	dest = src1 / src2
MOD	dest, src1, src2	dest = src1 % src2
SLL	dest, src1, src2	dest = src2 << src2
SRL	dest, src1, src2	dest = src1 >> src2 (unsigned shift right)
SRA	dest, src1, src2	dest = src1 >> src2 (signed shift right)
OR	dest, src1, src2	dest = src1 | src2 (bitwise OR)
XOR	dest, src1, src2	dest = src1 ^ src2 (exclusive OR)
AND	dest, src1, src2	dest = src1 & src2 (bitwise AND)
COMP	dest, src	dest = ~src (ones complement)
NOT	dest, src	dest = !src (unary NOT)
UMINUS	dest, src	dest = -src (unary minus)
SIZEOF	dest, src	dest = sizeof (src)
TYPEOF	dest, src	dest = typeof (src)

CAST	dest, src1, src2 [,pnum]	Cast src2 to type of src1 and store in dest. Pnum is optional parameter number for information only
MUX	dest, cond, src1, src2	dest = cond ? src1 : src2 (conditional expression)
CMPEQ	dest, src1, src2	dest = src1 == src2
CMPNE	dest, src1, src2	dest = src1 != src2
CMPLT	dest, src1, src2	dest = src1 < src2
CMPLE	dest, src1, src2	dest = src1 <= src2
CMPGT	dest, src1, src2	dest = src1 > src2
CMPGE	dest, src1, src2	dest = src1 >= src2
B	address	pc = address (address is constant)
BT	src, address	if (src) pc = address (address is integer type)
BF	src, address	if (!src) pc = address (address is integer type)
CALL	dest, nargs, block	Call the given block. Arguments are on stack and there are 'nargs' of them. Result of call is placed in dest
RET		Return from call
RETVAL	src	Return from call with value src
SUPERCALL	block	Call superblock constructor
TRY	catchlabel, endlabel	Start of 'try' exception block. Catchlabel is address of catch clause, endlabel is address at end of catch clause
CATCH	var	Execute start of catch clause. Place caught exception in var.
THROW	src	Throw the value in src as an exception
NOP		No operation
NEW	dest, nargs, block	Create an instance of the given block. There are nargs arguments on the stack
NEWVECTOR	dest, ndims, ctstart, ctend	Create a vector of 'ndims' dimensions. Size of dimensions are on stack. If ctstart and ctend are not zero, they are the addresses of the start and end of the construction code for each element of the vector
DELETE	src	Delete the operand
STREAM	dest, src1, src2	Stream src1 to src2, place result in dest
MACRO	macro, endlabel	Execute instance of macro. PC is set to endlabel after macro executes
ENUM	block	Assign enumeration constant values
FINDV	dest, src1, src2	Find the variable src2 in the value src1, place the value of the variable in dest
FINDA	dest, src1, src2	Find the variable src2 in the value src1. Place the address of the variable in dest
STV	dest, src1, src2, value	First find the value src2 in the variable src2 then store the value 'value' in it.
ADDR	dest, src	Get the address of src and place in dest
ADDRSUB	dest, base, s1 [,s2]	Get the address of base subscripted by s1. If s2 is present then the subscript it [s1:s2]
SETSUB	dest, src, base, s1 [,s2]	Set the value of base subscripted by s1 (:s2 if present) to the value of src. Store result dest also.

GETSUB	dest, base, s1 [,s2]	Get the value of base subscipted by s1 (:s2 is present).
DELSUB	base, s1 [,s2]	Delete base[s1] or base[s1:s2]
GETOBJ	dest, src	Get the instance of an object in src. If src is a package, get the automatic instance of it
PACKAGEASSIGN	dest, src	Store the automatic instance of a package
GETTHIS	dest, src	Get the "this" pointer from the src
PUSHSCOPE	scope	Push the scope onto the scope stack
POPSCOPE	ns, nt, nf	Pop ns (scopes), nt (trys), nf (foreach) off the stack
FOREACH	var, src, endlabel	Start of a foreach loop. Var is the control variable, src is the expression to iterate through. Endlabel is the address of the end of the loop
NEXT		Next iteration in current foreach loop
PUSHADDR	src	Push the address of src onto the stack
PUSH	src	Push the value of src onto the stack
POP	n	Pop n values off the stack
FOREIGN	src	Pass stream in src to foreign code handler
INLINE	dest, endaddr	Execute inline block. Code follows instruction. Result placed in dest. End of code is enaddr.
ADD.I	dest, src1, src2	Add integer
SUB.I	dest, src1, src2	Subtract integer
MUL.I	dest, src1, src2	Multiply integer
DIV.I	dest, src1, src2	Divide integer
MOD.I	dest, src1, src2	Modulus of integers
SLL.I	dest, src1, src2	Shift integer left
SRL.I	dest, src1, src2	Shift integer right logically
SRA.I	dest, src1, src2	Shift integer right arithmetically
OR.I	dest, src1, src2	Bitwise or integer
XOR.I	dest, src1, src2	Bitwise xor integer
AND.I	dest, src1, src2	Bitwise and integer
COMP.I	dest, src	Complement integer
NOT.I	dest, src1	Not of integer
UMINUS.I	dest, src	unary minus of integer
RETVAL.I	src	Return integer
CMPEQ.I	dest, src1, src2	Compare integers EQ
CMPNE.I	dest, src1, src2	Compare integers NE
CMPLT.I	dest, src1, src2	Compare integers LT
CMPLE.I	dest, src1, src2	Compare integers LE
CMPGT.I	dest, src1, src2	Compare integers GT
CMPGE.I	dest, src1, src2	Compare integers GE
ADD.R	dest, src1, src2	Add real
SUB.R	dest, src1, src2	Subtract real
MUL.R	dest, src1, src2	Multiply real
DIV.R	dest, src1, src2	Divide real
MOD.R	dest, src1, src2	Modulus of reals
UMINUS.R	dest, src	unary minus of real
RETVAL.R	src	Return real
CMPEQ.R	dest, src1, src2	Compare reals EQ
CMPNE.R	dest, src1, src2	Compare reals NE

CMPLT.R	dest, src1, src2	Compare reals LT
CMPLE.R	dest, src1, src2	Compare reals LE
CMPGT.R	dest, src1, src2	Compare reals GT
CMPGE.R	dest, src1, src2	Compare reals GE
ADD.S	dest, src1, src2	Add strings
SLL.S	dest, src1, src2	Shift string left
SRL.S	dest, src1, src2	Shift string right
CMPEQ.S	dest, src1, src2	Compare strings EQ
CMPNE.S	dest, src1, src2	Compare strings NE
CMPLT.S	dest, src1, src2	Compare strings LT
CMPLE.S	dest, src1, src2	Compare strings LE
CMPGT.S	dest, src1, src2	Compare strings GT
CMPGE.S	dest, src1, src2	Compare strings GE
RETVAL.S	src	Return string

18.2. Operands

Each of the instructions takes a number of operands. An operand can be one of:

- A register number
- A variable
- A constant value

There are an unlimited number of registers available. A register is named by 'Rx', where x is a number. A variable can be local (in the current static scope) or may be a number of levels up the static chain. If the variable name is suffixed with a number in parentheses then it is non-local and the number gives the number of frames up the static chain.

If the operand is a constant then is appears in the disassembly listing preceded by a hash (or pound sign).

The instruction suffixed with .I, .R or .S are optimizations and are used only where the types of the operands are known at compile time.

For example:

ADD R6, x(1), #1
destination = register 6
src1 = variable x (1 level up static chain)
src2 = constant 1

FINDV result, R4, print
destination = local variable "result"
src1 = register 4
src2 = identifier print

18.3. Notes

Some of the instructions use a stack. A value is pushed onto the stack by a **PUSH** or **PUSHADDR** instruction. The values are popped off automatically by the instruction making use of the stack.

The **PUSHSCOPE** and **POPSCOPE** instructions allow tracking of the current scope. In addition, the POPSCOPE instruction deals with nested 'try' blocks and 'foreach' statements.

The **CALL** instruction can be used to call any block, not just functions. The **CALL** instruction always has a "***this***" pointer as its first argument (on the stack). This may be ignored for static functions and those without a "this" argument. The **NEW** instruction is identical to a **CALL**.

When the block being called is invoked, the virtual machine allocates a complete new set of registers for that block so there is never a need to save the registers from the caller.

The arguments for a **CALL** are pushed from right to left onto the stack. The call mechanism takes the arguments off the stack and inserts them into the variables or native parameter list of the called function. The called block assigns the default parameter values and casts to the formal parameter types. The **CALL** instruction deals with the varargs case (insertion of additional args into the args vector). Arguments that are not constant are passed as addresses. This is to allow for reference parameters.

When the callee is invoked the arguments will have the values passed from the caller.

The call instruction pops all the arguments off the stack

Example:

```
x = func (1,2,3)

        PUSH 3
        PUSH 2
        PUSH 1
        CALL x, 3, func
```

The **NEW** instruction is similar to call. The arguments are pushed onto the stack in reverse order and the call is made. The object returned is placed in the destination location.

```
var x = new Tree (1,2)

        PUSH 2
        PUSH 1
        NEW x, 2, Tree
```

The **SUPERCALL** instruction calls the constructor for the superblock of an object during the construction of the object. Like a regular call, the arguments are pushed onto the stack and popped by the instruction.

The **NEWVECTOR** instruction allows a vector to be created, it is pretty complex. There can be any number of dimensions to the vector and each dimension will have a specified size. The last dimension can specify an object to be constructed.

The sizes of the dimensions are pushed onto the stack starting with the last dimension. For example

```
new [100][10][5]

        PUSH 5
        PUSH 10
        PUSH 100
        NEWVECTOR dest, 3, 0, 0

new Class [10][5] (1,2)
```

```
        PUSH 5
        PUSH 10
        NEWVECTOR dest, 2, I1, I2
I1:
        PUSH 2
        PUSH 1
        NEW dest, 2, Class
        END
I2:
```

The sizes of the dimensions are popped by the **NEWVECTOR** instruction

Inline code blocks are signified by the **INLINE** instruction and are coded as follows:

```
INLINE dest, I1
        code for block
I1:
```

The value from the inline block is returned using a RETVAL instruction.

Chapter 19. Regular Expressions


```
DIFFERENCES FROM PERL
     The differences described here are with respect to Perl
     5.005.

     1. By default, a whitespace character is any character that
     the C library function isspace() recognizes, though it is
     possible to compile PCRE with alternative character type
     tables. Normally isspace() matches space, formfeed, newline,
     carriage return, horizontal tab, and vertical tab. Perl 5 no
     longer includes vertical tab in its set of whitespace char-
     acters. The \v escape that was in the Perl documentation for
     a long time was never in fact recognized. However, the char-
     acter itself was treated as whitespace at least up to 5.002.
     In 5.004 and 5.005 it does not match \s.

     2. PCRE does not allow repeat quantifiers on lookahead
     assertions. Perl permits them, but they do not mean what you
     might think. For example, (?!a){3} does not assert that the
     next three characters are not "a". It just asserts that the
     next character is not "a" three times.

     3. Capturing subpatterns that occur inside negative looka-
     head assertions are counted, but their entries in the
     offsets vector are never set. Perl sets its numerical vari-
     ables from any such patterns that are matched before the
     assertion fails to match something (thereby succeeding), but
     only if the negative lookahead assertion contains just one
     branch.

     4. Though binary zero characters are supported in the sub-
     ject string, they are not allowed in a pattern string
     because it is passed as a normal C string, terminated by
     zero. The escape sequence "\0" can be used in the pattern to
     represent a binary zero.

     5. The following Perl escape sequences are not supported:
     \l, \u, \L, \U, \E, \Q. In fact these are implemented by
     Perl's general string-handling and are not part of its pat-
     tern matching engine.

     6. The Perl \G assertion is not supported as it is not
     relevant to single pattern matches.

     7. Fairly obviously, PCRE does not support the (?{code}) and
     (?p{code}) constructions. However, there is some experimen-
     tal support for recursive patterns using the non-Perl item
     (?R).
```

8. There are at the time of writing some oddities in Perl 5.005_02 concerned with the settings of captured strings when part of a pattern is repeated. For example, matching "aba" against the pattern /^(a(b)?)+$/ sets $2 to the value "b", but matching "aabbaa" against /^(aa(bb)?)+$/ leaves $2 unset. However, if the pattern is changed to /^(aa(b(b))?)+$/ then $2 (and $3) are set.

In Perl 5.004 $2 is set in both cases, and that is also true of PCRE. If in the future Perl changes to a consistent state that is different, PCRE may change to follow.

9. Another as yet unresolved discrepancy is that in Perl 5.005_02 the pattern /^(a)?(?(1)a|b)+$/ matches the string "a", whereas in PCRE it does not. However, in both Perl and PCRE /^(a)?a/ matched against "a" leaves $1 unset.

10. PCRE provides some extensions to the Perl regular expression facilities:

(a) Although lookbehind assertions must match fixed length strings, each alternative branch of a lookbehind assertion can match a different length of string. Perl 5.005 requires them all to have the same length.

(b) If PCRE_DOLLAR_ENDONLY is set and PCRE_MULTILINE is not set, the $ meta- character matches only at the very end of the string.

(c) If PCRE_EXTRA is set, a backslash followed by a letter with no special meaning is faulted.

(d) If PCRE_UNGREEDY is set, the greediness of the repetition quantifiers is inverted, that is, by default they are not greedy, but if followed by a question mark they are.

(e) PCRE_ANCHORED can be used to force a pattern to be tried only at the start of the subject.

(f) The PCRE_NOTBOL, PCRE_NOTEOL, and PCRE_NOTEMPTY options for pcre_exec() have no Perl equivalents.

(g) The (?R) construct allows for recursive pattern matching (Perl 5.6 can do this using the (?p{code}) construct, which PCRE cannot of course support.)

REGULAR EXPRESSION DETAILS

The syntax and semantics of the regular expressions supported by PCRE are described below. Regular expressions are also described in the Perl documentation and in a number of other books, some of which have copious examples. Jeffrey Friedl's "Mastering Regular Expressions", published by O'Reilly (ISBN 1-56592-257), covers them in great detail.

The description here is intended as reference documentation.

The basic operation of PCRE is on strings of bytes. However, there is the beginnings of some support for UTF-8 character strings. To use this support you must configure PCRE to include it, and then call pcre_compile() with the PCRE_UTF8 option. How this affects the pattern matching is described in the final section of this document.

A regular expression is a pattern that is matched against a subject string from left to right. Most characters stand for themselves in a pattern, and match the corresponding characters in the subject. As a trivial example, the pattern

  The quick brown fox

matches a portion of a subject string that is identical to itself. The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the pattern by the use of meta-characters, which do not stand for themselves but instead are interpreted in some special way.

There are two different sets of meta-characters: those that are recognized anywhere in the pattern except within square brackets, and those that are recognized in square brackets. Outside square brackets, the meta-characters are as follows:

```
  \      general escape character with several uses
  ^      assert start of  subject  (or  line,  in  multiline
mode)
  $      assert end of subject (or line, in multiline mode)
  .      match any character except newline (by default)
  [      start character class definition
  |      start of alternative branch
  (      start subpattern
  )      end subpattern
  ?      extends the meaning of (
         also 0 or 1 quantifier
         also quantifier minimizer
  *      0 or more quantifier
  +      1 or more quantifier
  {      start min/max quantifier
```

Part of a pattern that is in square brackets is called a "character class". In a character class the only meta-characters are:

```
  \      general escape character
  ^      negate the class, but only if the first character
  -      indicates character range
  ]      terminates the character class
```

The following sections describe the use of each of the meta-characters.

BACKSLASH

The backslash character has several uses. Firstly, if it is followed by a non-alphameric character, it takes away any special meaning that character may have. This use of backslash as an escape character applies both inside and outside character classes.

For example, if you want to match a "*" character, you write "*" in the pattern. This applies whether or not the following character would otherwise be interpreted as a meta-character, so it is always safe to precede a non-alphameric with "\" to specify that it stands for itself. In particular, if you want to match a backslash, you write "\\".

If a pattern is compiled with the PCRE_EXTENDED option, whitespace in the pattern (other than in a character class) and characters between a "#" outside a character class and the next newline character are ignored. An escaping backslash can be used to include a whitespace or "#" character as part of the pattern.

A second use of backslash provides a way of encoding non-printing characters in patterns in a visible manner. There is no restriction on the appearance of non-printing characters, apart from the binary zero that terminates a pattern, but when a pattern is being prepared by text editing, it is usually easier to use one of the following escape sequences than the binary character it represents:

```
  \a     alarm, that is, the BEL character (hex 07)
  \cx    "control-x", where x is any character
  \e     escape (hex 1B)
  \f     formfeed (hex 0C)
  \n     newline (hex 0A)
  \r     carriage return (hex 0D)
  \t     tab (hex 09)
  \xhh   character with hex code hh
  \ddd   character with octal code ddd, or backreference
```

The precise effect of "\cx" is as follows: if "x" is a lower case letter, it is converted to upper case. Then bit 6 of the character (hex 40) is inverted. Thus "\cz" becomes hex 1A, but "\c{" becomes hex 3B, while "\c;" becomes hex 7B.

After "\x", up to two hexadecimal digits are read (letters can be in upper or lower case).

After "\0" up to two further octal digits are read. In both cases, if there are fewer than two digits, just those that are present are used. Thus the sequence "\0\x\07" specifies two binary zeros followed by a BEL character. Make sure you supply two digits after the initial zero if the character that follows is itself an octal digit.

The handling of a backslash followed by a digit other than 0 is complicated. Outside a character class, PCRE reads it and any following digits as a decimal number. If the number

is less than 10, or if there have been at least that many previous capturing left parentheses in the expression, the entire sequence is taken as a back reference. A description of how this works is given later, following the discussion of parenthesized subpatterns.

Inside a character class, or if the decimal number is greater than 9 and there have not been that many capturing subpatterns, PCRE re-reads up to three octal digits following the backslash, and generates a single byte from the least significant 8 bits of the value. Any subsequent digits stand for themselves. For example:

```
  \040   is another way of writing a space
  \40    is the same, provided there are fewer than 40
            previous capturing subpatterns
  \7     is always a back reference
  \11    might be a back reference, or another way of
            writing a tab
  \011   is always a tab
  \0113  is a tab followed by the character "3"
  \113   is the character with octal code 113 (since there
            can be no more than 99 back references)
  \377   is a byte consisting entirely of 1 bits
  \81    is either a back reference, or a binary zero
            followed by the two characters "8" and "1"
```

Note that octal values of 100 or greater must not be introduced by a leading zero, because no more than three octal digits are ever read.

All the sequences that define a single byte value can be used both inside and outside character classes. In addition, inside a character class, the sequence "\b" is interpreted as the backspace character (hex 08). Outside a character class it has a different meaning (see below).

The third use of backslash is for specifying generic character types:

```
  \d     any decimal digit
  \D     any character that is not a decimal digit
  \s     any whitespace character
  \S     any character that is not a whitespace character
  \w     any "word" character
  \W     any "non-word" character
```

Each pair of escape sequences partitions the complete set of characters into two disjoint sets. Any given character matches one, and only one, of each pair.

A "word" character is any letter or digit or the underscore character, that is, any character which can be part of a Perl "word". The definition of letters and digits is controlled by PCRE's character tables, and may vary if locale-specific matching is taking place (see "Locale support" above). For example, in the "fr" (French) locale, some char-

acter codes greater than 128 are used for accented letters, and these are matched by \w.

These character type sequences can appear both inside and outside character classes. They each match one character of the appropriate type. If the current matching point is at the end of the subject string, all of them fail, since there is no character to match.

The fourth use of backslash is for certain simple assertions. An assertion specifies a condition that has to be met at a particular point in a match, without consuming any characters from the subject string. The use of subpatterns for more complicated assertions is described below. The backslashed assertions are

```
  \b     word boundary
  \B     not a word boundary
  \A     start of subject (independent of multiline mode)
  \Z     end of subject or newline at end (independent of
multiline mode)
  \z     end of subject (independent of multiline mode)
```

These assertions may not appear in character classes (but note that "\b" has a different meaning, namely the backspace character, inside a character class).

A word boundary is a position in the subject string where the current character and the previous character do not both match \w or \W (i.e. one matches \w and the other matches \W), or the start or end of the string if the first or last character matches \w, respectively.

The \A, \Z, and \z assertions differ from the traditional circumflex and dollar (described below) in that they only ever match at the very start and end of the subject string, whatever options are set. They are not affected by the PCRE_NOTBOL or PCRE_NOTEOL options. If the startoffset argument of pcre_exec() is non-zero, \A can never match. The difference between \Z and \z is that \Z matches before a newline that is the last character of the string as well as at the end of the string, whereas \z matches only at the end.

CIRCUMFLEX AND DOLLAR

Outside a character class, in the default matching mode, the circumflex character is an assertion which is true only if the current matching point is at the start of the subject string. If the startoffset argument of pcre_exec() is non-zero, circumflex can never match. Inside a character class, circumflex has an entirely different meaning (see below).

Circumflex need not be the first character of the pattern if a number of alternatives are involved, but it should be the first thing in each alternative in which it appears if the

pattern is ever to match that branch. If all possible alternatives start with a circumflex, that is, if the pattern is constrained to match only at the start of the subject, it is said to be an "anchored" pattern. (There are also other constructs that can cause a pattern to be anchored.)

A dollar character is an assertion which is true only if the current matching point is at the end of the subject string, or immediately before a newline character that is the last character in the string (by default). Dollar need not be the last character of the pattern if a number of alternatives are involved, but it should be the last item in any branch in which it appears. Dollar has no special meaning in a character class.

The meaning of dollar can be changed so that it matches only at the very end of the string, by setting the PCRE_DOLLAR_ENDONLY option at compile or matching time. This does not affect the \Z assertion.

The meanings of the circumflex and dollar characters are changed if the PCRE_MULTILINE option is set. When this is the case, they match immediately after and immediately before an internal "\n" character, respectively, in addition to matching at the start and end of the subject string. For example, the pattern /^abc$/ matches the subject string "def\nabc" in multiline mode, but not otherwise. Consequently, patterns that are anchored in single line mode because all branches start with "^" are not anchored in multiline mode, and a match for circumflex is possible when the startoffset argument of pcre_exec() is non-zero. The PCRE_DOLLAR_ENDONLY option is ignored if PCRE_MULTILINE is set.

Note that the sequences \A, \Z, and \z can be used to match the start and end of the subject in both modes, and if all branches of a pattern start with \A it is always anchored, whether PCRE_MULTILINE is set or not.

FULL STOP (PERIOD, DOT)

Outside a character class, a dot in the pattern matches any one character in the subject, including a non-printing character, but not (by default) newline. If the PCRE_DOTALL option is set, dots match newlines as well. The handling of dot is entirely independent of the handling of circumflex and dollar, the only relationship being that they both involve newline characters. Dot has no special meaning in a character class.

SQUARE BRACKETS

An opening square bracket introduces a character class, terminated by a closing square bracket. A closing square bracket on its own is not special. If a closing square

bracket is required as a member of the class, it should be the first data character in the class (after an initial circumflex, if present) or escaped with a backslash.

A character class matches a single character in the subject; the character must be in the set of characters defined by the class, unless the first character in the class is a circumflex, in which case the subject character must not be in the set defined by the class. If a circumflex is actually required as a member of the class, ensure it is not the first character, or escape it with a backslash.

For example, the character class [aeiou] matches any lower case vowel, while [^aeiou] matches any character that is not a lower case vowel. Note that a circumflex is just a convenient notation for specifying the characters which are in the class by enumerating those that are not. It is not an assertion: it still consumes a character from the subject string, and fails if the current pointer is at the end of the string.

When caseless matching is set, any letters in a class represent both their upper case and lower case versions, so for example, a caseless [aeiou] matches "A" as well as "a", and a caseless [^aeiou] does not match "A", whereas a caseful version would.

The newline character is never treated in any special way in character classes, whatever the setting of the PCRE_DOTALL or PCRE_MULTILINE options is. A class such as [^a] will always match a newline.

The minus (hyphen) character can be used to specify a range of characters in a character class. For example, [d-m] matches any letter between d and m, inclusive. If a minus character is required in a class, it must be escaped with a backslash or appear in a position where it cannot be interpreted as indicating a range, typically as the first or last character in the class.

It is not possible to have the literal character "]" as the end character of a range. A pattern such as [W-]46] is interpreted as a class of two characters ("W" and "-") followed by a literal string "46]", so it would match "W46]" or "-46]". However, if the "]" is escaped with a backslash it is interpreted as the end of range, so [W-\]46] is interpreted as a single class containing a range followed by two separate characters. The octal or hexadecimal representation of "]" can also be used to end a range.

Ranges operate in ASCII collating sequence. They can also be used for characters specified numerically, for example [\000-\037]. If a range that includes letters is used when caseless matching is set, it matches the letters in either case. For example, [W-c] is equivalent to [][\^_`wxyzabc], matched caselessly, and if character tables for the "fr" locale are in use, [\xc8-\xcb] matches accented E characters

in both cases.

The character types \d, \D, \s, \S, \w, and \W may also appear in a character class, and add the characters that they match to the class. For example, [\dABCDEF] matches any hexadecimal digit. A circumflex can conveniently be used with the upper case character types to specify a more restricted set of characters than the matching lower case type. For example, the class [^\W_] matches any letter or digit, but not underscore.

All non-alphameric characters other than \, -, ^ (at the start) and the terminating] are non-special in character classes, but it does no harm if they are escaped.

POSIX CHARACTER CLASSES

Perl 5.6 (not yet released at the time of writing) is going to support the POSIX notation for character classes, which uses names enclosed by [: and :] within the enclosing square brackets. PCRE supports this notation. For example,

```
[01[:alpha:]%]
```

matches "0", "1", any alphabetic character, or "%". The supported class names are

```
alnum    letters and digits
alpha    letters
ascii    character codes 0 - 127
cntrl    control characters
digit    decimal digits (same as \d)
graph    printing characters, excluding space
lower    lower case letters
print    printing characters, including space
punct    printing characters, excluding letters and digits
space    white space (same as \s)
upper    upper case letters
word     "word" characters (same as \w)
xdigit   hexadecimal digits
```

The names "ascii" and "word" are Perl extensions. Another Perl extension is negation, which is indicated by a ^ character after the colon. For example,

```
[12[:^digit:]]
```

matches "1", "2", or any non-digit. PCRE (and Perl) also recognize the POSIX syntax [.ch.] and [=ch=] where "ch" is a "collating element", but these are not supported, and an error is given if they are encountered.

VERTICAL BAR

Vertical bar characters are used to separate alternative

patterns. For example, the pattern

```
gilbert|sullivan
```

matches either "gilbert" or "sullivan". Any number of alternatives may appear, and an empty alternative is permitted (matching the empty string). The matching process tries each alternative in turn, from left to right, and the first one that succeeds is used. If the alternatives are within a subpattern (defined below), "succeeds" means matching the rest of the main pattern as well as the alternative in the subpattern.

INTERNAL OPTION SETTING

The settings of PCRE_CASELESS, PCRE_MULTILINE, PCRE_DOTALL, and PCRE_EXTENDED can be changed from within the pattern by a sequence of Perl option letters enclosed between "(?" and ")". The option letters are

```
i  for PCRE_CASELESS
m  for PCRE_MULTILINE
s  for PCRE_DOTALL
x  for PCRE_EXTENDED
```

For example, (?im) sets caseless, multiline matching. It is also possible to unset these options by preceding the letter with a hyphen, and a combined setting and unsetting such as (?im-sx), which sets PCRE_CASELESS and PCRE_MULTILINE while unsetting PCRE_DOTALL and PCRE_EXTENDED, is also permitted. If a letter appears both before and after the hyphen, the option is unset.

The scope of these option changes depends on where in the pattern the setting occurs. For settings that are outside any subpattern (defined below), the effect is the same as if the options were set or unset at the start of matching. The following patterns all behave in exactly the same way:

```
(?i)abc
a(?i)bc
ab(?i)c
abc(?i)
```

which in turn is the same as compiling the pattern abc with PCRE_CASELESS set. In other words, such "top level" settings apply to the whole pattern (unless there are other changes inside subpatterns). If there is more than one setting of the same option at top level, the rightmost setting is used.

If an option change occurs inside a subpattern, the effect is different. This is a change of behaviour in Perl 5.005. An option change inside a subpattern affects only that part of the subpattern that follows it, so

```
  (a(?i)b)c
```

matches abc and aBc and no other strings (assuming PCRE_CASELESS is not used). By this means, options can be made to have different settings in different parts of the pattern. Any changes made in one alternative do carry on into subsequent branches within the same subpattern. For example,

```
  (a(?i)b|c)
```

matches "ab", "aB", "c", and "C", even though when matching "C" the first branch is abandoned before the option setting. This is because the effects of option settings happen at compile time. There would be some very weird behaviour otherwise.

The PCRE-specific options PCRE_UNGREEDY and PCRE_EXTRA can be changed in the same way as the Perl-compatible options by using the characters U and X respectively. The (?X) flag setting is special in that it must always occur earlier in the pattern than any of the additional features it turns on, even when it is at top level. It is best put at the start.

SUBPATTERNS

Subpatterns are delimited by parentheses (round brackets), which can be nested. Marking part of a pattern as a subpattern does two things:

1. It localizes a set of alternatives. For example, the pattern

```
  cat(aract|erpillar|)
```

matches one of the words "cat", "cataract", or "caterpillar". Without the parentheses, it would match "cataract", "erpillar" or the empty string.

2. It sets up the subpattern as a capturing subpattern (as defined above). When the whole pattern matches, that portion of the subject string that matched the subpattern is passed back to the caller via the ovector argument of pcre_exec(). Opening parentheses are counted from left to right (starting from 1) to obtain the numbers of the capturing subpatterns.

For example, if the string "the red king" is matched against the pattern

```
  the ((red|white) (king|queen))
```

the captured substrings are "red king", "red", and "king", and are numbered 1, 2, and 3, respectively.

The fact that plain parentheses fulfil two functions is not

always helpful. There are often times when a grouping subpattern is required without a capturing requirement. If an opening parenthesis is followed by "?:", the subpattern does not do any capturing, and is not counted when computing the number of any subsequent capturing subpatterns. For example, if the string "the white queen" is matched against the pattern

```
the ((?:red|white) (king|queen))
```

the captured substrings are "white queen" and "queen", and are numbered 1 and 2. The maximum number of captured substrings is 99, and the maximum number of all subpatterns, both capturing and non-capturing, is 200.

As a convenient shorthand, if any option settings are required at the start of a non-capturing subpattern, the option letters may appear between the "?" and the ":". Thus the two patterns

```
(?i:saturday|sunday)
(?:(?i)saturday|sunday)
```

match exactly the same set of strings. Because alternative branches are tried from left to right, and options are not reset until the end of the subpattern is reached, an option setting in one branch does affect subsequent branches, so the above patterns match "SUNDAY" as well as "Saturday".

REPETITION

Repetition is specified by quantifiers, which can follow any of the following items:

```
a single character, possibly escaped
the . metacharacter
a character class
a back reference (see next section)
a parenthesized subpattern (unless it is an assertion - see below)
```

The general repetition quantifier specifies a minimum and maximum number of permitted matches, by giving the two numbers in curly brackets (braces), separated by a comma. The numbers must be less than 65536, and the first must be less than or equal to the second. For example:

```
z{2,4}
```

matches "zz", "zzz", or "zzzz". A closing brace on its own is not a special character. If the second number is omitted, but the comma is present, there is no upper limit; if the second number and the comma are both omitted, the quantifier specifies an exact number of required matches. Thus

```
[aeiou]{3,}
```

matches at least 3 successive vowels, but may match many more, while

```
\d{8}
```

matches exactly 8 digits. An opening curly bracket that appears in a position where a quantifier is not allowed, or one that does not match the syntax of a quantifier, is taken as a literal character. For example, {,6} is not a quantifier, but a literal string of four characters.
The quantifier {0} is permitted, causing the expression to behave as if the previous item and the quantifier were not present.

For convenience (and historical compatibility) the three most common quantifiers have single-character abbreviations:

```
*    is equivalent to {0,}
+    is equivalent to {1,}
?    is equivalent to {0,1}
```

It is possible to construct infinite loops by following a subpattern that can match no characters with a quantifier that has no upper limit, for example:

```
(a?)*
```

Earlier versions of Perl and PCRE used to give an error at compile time for such patterns. However, because there are cases where this can be useful, such patterns are now accepted, but if any repetition of the subpattern does in fact match no characters, the loop is forcibly broken.

By default, the quantifiers are "greedy", that is, they match as much as possible (up to the maximum number of permitted times), without causing the rest of the pattern to fail. The classic example of where this gives problems is in trying to match comments in C programs. These appear between the sequences /* and */ and within the sequence, individual * and / characters may appear. An attempt to match C comments by applying the pattern

```
/\*.*\*/
```

to the string

```
/* first command */  not comment  /* second comment */
```

fails, because it matches the entire string owing to the greediness of the .* item.

However, if a quantifier is followed by a question mark, it ceases to be greedy, and instead matches the minimum number of times possible, so the pattern

```
/\*.*?\*/
```

does the right thing with the C comments. The meaning of the various quantifiers is not otherwise changed, just the preferred number of matches. Do not confuse this use of question mark with its use as a quantifier in its own right. Because it has two uses, it can sometimes appear doubled, as in

```
\d??\d
```

which matches one digit by preference, but can match two if that is the only way the rest of the pattern matches.

If the PCRE_UNGREEDY option is set (an option which is not available in Perl), the quantifiers are not greedy by default, but individual ones can be made greedy by following them with a question mark. In other words, it inverts the default behaviour.

When a parenthesized subpattern is quantified with a minimum repeat count that is greater than 1 or with a limited maximum, more store is required for the compiled pattern, in proportion to the size of the minimum or maximum.

If a pattern starts with .* or .{0,} and the PCRE_DOTALL option (equivalent to Perl's /s) is set, thus allowing the . to match newlines, the pattern is implicitly anchored, because whatever follows will be tried against every character position in the subject string, so there is no point in retrying the overall match at any position after the first. PCRE treats such a pattern as though it were preceded by \A. In cases where it is known that the subject string contains no newlines, it is worth setting PCRE_DOTALL when the pattern begins with .* in order to obtain this optimization, or alternatively using ^ to indicate anchoring explicitly.

When a capturing subpattern is repeated, the value captured is the substring that matched the final iteration. For example, after

```
(tweedle[dume]{3}\s*)+
```

has matched "tweedledum tweedledee" the value of the captured substring is "tweedledee". However, if there are nested capturing subpatterns, the corresponding captured values may have been set in previous iterations. For example, after

```
/(a|(b))+/
```

matches "aba" the value of the second captured substring is "b".

BACK REFERENCES

Outside a character class, a backslash followed by a digit

greater than 0 (and possibly further digits) is a back

reference to a capturing subpattern earlier (i.e. to its left) in the pattern, provided there have been that many previous capturing left parentheses.

However, if the decimal number following the backslash is less than 10, it is always taken as a back reference, and causes an error only if there are not that many capturing left parentheses in the entire pattern. In other words, the parentheses that are referenced need not be to the left of the reference for numbers less than 10. See the section entitled "Backslash" above for further details of the handling of digits following a backslash.

A back reference matches whatever actually matched the capturing subpattern in the current subject string, rather than anything matching the subpattern itself. So the pattern

```
(sens|respons)e and \1ibility
```

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If caseful matching is in force at the time of the back reference, the case of letters is relevant. For example,

```
((?i)rah)\s+\1
```

matches "rah rah" and "RAH RAH", but not "RAH rah", even though the original capturing subpattern is matched caselessly.

There may be more than one back reference to the same subpattern. If a subpattern has not actually been used in a particular match, any back references to it always fail. For example, the pattern

```
(a|(bc))\2
```

always fails if it starts to match "a" rather than "bc". Because there may be up to 99 back references, all digits following the backslash are taken as part of a potential back reference number. If the pattern continues with a digit character, some delimiter must be used to terminate the back reference. If the PCRE_EXTENDED option is set, this can be whitespace. Otherwise an empty comment can be used.

A back reference that occurs inside the parentheses to which it refers fails when the subpattern is first used, so, for example, (a\1) never matches. However, such references can

be useful inside repeated subpatterns. For example, the pattern

```
(a|b\1)+
```

matches any number of "a"s and also "aba", "ababbaa" etc. At each iteration of the subpattern, the back reference matches the character string corresponding to the previous iteration. In order for this to work, the pattern must be such that the first iteration does not need to match the back reference. This can be done using alternation, as in the example above, or by a quantifier with a minimum of zero.

ASSERTIONS

An assertion is a test on the characters following or preceding the current matching point that does not actually consume any characters. The simple assertions coded as \b, \B, \A, \Z, \z, ^ and $ are described above. More complicated assertions are coded as subpatterns. There are two kinds: those that look ahead of the current position in the subject string, and those that look behind it.

An assertion subpattern is matched in the normal way, except that it does not cause the current matching position to be changed. Lookahead assertions start with (?= for positive assertions and (?! for negative assertions. For example,

```
\w+(?=;)
```

matches a word followed by a semicolon, but does not include the semicolon in the match, and

```
foo(?!bar)
```

matches any occurrence of "foo" that is not followed by "bar". Note that the apparently similar pattern

```
(?!foo)bar
```

does not find an occurrence of "bar" that is preceded by something other than "foo"; it finds any occurrence of "bar" whatsoever, because the assertion (?!foo) is always true when the next three characters are "bar". A lookbehind assertion is needed to achieve this effect.

Lookbehind assertions start with (?<= for positive assertions and (?<! for negative assertions. For example,

```
(?<!foo)bar
```

does find an occurrence of "bar" that is not preceded by "foo". The contents of a lookbehind assertion are restricted such that all the strings it matches must have a fixed length. However, if there are several alternatives, they do not all have to have the same fixed length. Thus

```
  (?<=bullock|donkey)
```

is permitted, but

```
  (?<!dogs?|cats?)
```

causes an error at compile time. Branches that match different length strings are permitted only at the top level of a lookbehind assertion. This is an extension compared with Perl 5.005, which requires all branches to match the same length of string. An assertion such as

```
  (?<=ab(c|de))
```

is not permitted, because its single top-level branch can match two different lengths, but it is acceptable if rewritten to use two top-level branches:

```
  (?<=abc|abde)
```

The implementation of lookbehind assertions is, for each alternative, to temporarily move the current position back by the fixed width and then try to match. If there are insufficient characters before the current position, the match is deemed to fail. Lookbehinds in conjunction with once-only subpatterns can be particularly useful for matching at the ends of strings; an example is given at the end of the section on once-only subpatterns.

Several assertions (of any sort) may occur in succession. For example,

```
  (?<=\d{3})(?<!999)foo
```

matches "foo" preceded by three digits that are not "999". Notice that each of the assertions is applied independently at the same point in the subject string. First there is a check that the previous three characters are all digits, and then there is a check that the same three characters are not "999". This pattern does not match "foo" preceded by six characters, the first of which are digits and the last three of which are not "999". For example, it doesn't match "123abcfoo". A pattern to do that is

```
  (?<=\d{3}...)(?<!999)foo
```

This time the first assertion looks at the preceding six characters, checking that the first three are digits, and then the second assertion checks that the preceding three characters are not "999".

Assertions can be nested in any combination. For example,

```
  (?<=(?<!foo)bar)baz
```

matches an occurrence of "baz" that is preceded by "bar"

which in turn is not preceded by "foo", while

```
(?<=\d{3}(?!999)...)foo
```

is another pattern which matches "foo" preceded by three digits and any three characters that are not "999".

Assertion subpatterns are not capturing subpatterns, and may not be repeated, because it makes no sense to assert the same thing several times. If any kind of assertion contains capturing subpatterns within it, these are counted for the purposes of numbering the capturing subpatterns in the whole pattern. However, substring capturing is carried out only for positive assertions, because it does not make sense for negative assertions.

Assertions count towards the maximum of 200 parenthesized subpatterns.

ONCE-ONLY SUBPATTERNS

With both maximizing and minimizing repetition, failure of what follows normally causes the repeated item to be re-evaluated to see if a different number of repeats allows the rest of the pattern to match. Sometimes it is useful to prevent this, either to change the nature of the match, or to cause it fail earlier than it otherwise might, when the author of the pattern knows there is no point in carrying on.

Consider, for example, the pattern \d+foo when applied to the subject line

```
123456bar
```

After matching all 6 digits and then failing to match "foo", the normal action of the matcher is to try again with only 5 digits matching the \d+ item, and then with 4, and so on, before ultimately failing. Once-only subpatterns provide the means for specifying that once a portion of the pattern has matched, it is not to be re-evaluated in this way, so the matcher would give up immediately on failing to match "foo" the first time. The notation is another kind of special parenthesis, starting with (?> as in this example:

```
(?>\d+)bar
```

This kind of parenthesis "locks up" the part of the pattern it contains once it has matched, and a failure further into the pattern is prevented from backtracking into it. Backtracking past it to previous items, however, works as normal.

An alternative description is that a subpattern of this type matches the string of characters that an identical standalone pattern would match, if anchored at the current point

in the subject string.

Once-only subpatterns are not capturing subpatterns. Simple cases such as the above example can be thought of as a maximizing repeat that must swallow everything it can. So, while both \d+ and \d+? are prepared to adjust the number of digits they match in order to make the rest of the pattern match, (?>\d+) can only match an entire sequence of digits.

This construction can of course contain arbitrarily complicated subpatterns, and it can be nested.

Once-only subpatterns can be used in conjunction with lookbehind assertions to specify efficient matching at the end of the subject string. Consider a simple pattern such as

```
abcd$
```

when applied to a long string which does not match. Because matching proceeds from left to right, PCRE will look for each "a" in the subject and then see if what follows matches the rest of the pattern. If the pattern is specified as

```
^.*abcd$
```

the initial .* matches the entire string at first, but when this fails (because there is no following "a"), it backtracks to match all but the last character, then all but the last two characters, and so on. Once again the search for "a" covers the entire string, from right to left, so we are no better off. However, if the pattern is written as

```
^(?>.*)(?<=abcd)
```

there can be no backtracking for the .* item; it can match only the entire string. The subsequent lookbehind assertion does a single test on the last four characters. If it fails, the match fails immediately. For long strings, this approach makes a significant difference to the processing time.

When a pattern contains an unlimited repeat inside a subpattern that can itself be repeated an unlimited number of times, the use of a once-only subpattern is the only way to avoid some failing matches taking a very long time indeed. The pattern

```
(\D+|<\d+>)*[!?]
```

matches an unlimited number of substrings that either consist of non-digits, or digits enclosed in <>, followed by either ! or ?. When it matches, it runs quickly. However, if it is applied to

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

it takes a long time before reporting failure. This is because the string can be divided between the two repeats in

a large number of ways, and all have to be tried. (The example used [!?] rather than a single character at the end, because both PCRE and Perl have an optimization that allows for fast failure when a single character is used. They remember the last single character that is required for a match, and fail early if it is not present in the string.) If the pattern is changed to

```
((?>\D+)|<\d+>)*[!?]
```

sequences of non-digits cannot be broken, and failure happens quickly.

CONDITIONAL SUBPATTERNS

It is possible to cause the matching process to obey a subpattern conditionally or to choose between two alternative subpatterns, depending on the result of an assertion, or whether a previous capturing subpattern matched or not. The two possible forms of conditional subpattern are

```
(?(condition)yes-pattern)
(?(condition)yes-pattern|no-pattern)
```

If the condition is satisfied, the yes-pattern is used; otherwise the no-pattern (if present) is used. If there are more than two alternatives in the subpattern, a compile-time error occurs.

There are two kinds of condition. If the text between the parentheses consists of a sequence of digits, the condition is satisfied if the capturing subpattern of that number has previously matched. The number must be greater than zero. Consider the following pattern, which contains non-significant white space to make it more readable (assume the PCRE_EXTENDED option) and to divide it into three parts for ease of discussion:

```
( \( )?    [^()]+    (?(1) \) )
```

The first part matches an optional opening parenthesis, and if that character is present, sets it as the first captured substring. The second part matches one or more characters that are not parentheses. The third part is a conditional subpattern that tests whether the first set of parentheses matched or not. If they did, that is, if subject started with an opening parenthesis, the condition is true, and so the yes-pattern is executed and a closing parenthesis is required. Otherwise, since no-pattern is not present, the subpattern matches nothing. In other words, this pattern matches a sequence of non-parentheses, optionally enclosed in parentheses.

If the condition is not a sequence of digits, it must be an assertion. This may be a positive or negative lookahead or lookbehind assertion. Consider this pattern, again contain-

ing non-significant white space, and with the two alternatives on the second line:

```
(?(?=[^a-z]*[a-z])
\d{2}-[a-z]{3}-\d{2}  |  \d{2}-\d{2}-\d{2} )
```

The condition is a positive lookahead assertion that matches an optional sequence of non-letters followed by a letter. In other words, it tests for the presence of at least one letter in the subject. If a letter is found, the subject is matched against the first alternative; otherwise it is matched against the second. This pattern matches strings in one of the two forms dd-aaa-dd or dd-dd-dd, where aaa are letters and dd are digits.

COMMENTS

The sequence (?# marks the start of a comment which continues up to the next closing parenthesis. Nested parentheses are not permitted. The characters that make up a comment play no part in the pattern matching at all.

If the PCRE_EXTENDED option is set, an unescaped # character outside a character class introduces a comment that continues up to the next newline character in the pattern.

RECURSIVE PATTERNS

Consider the problem of matching a string in parentheses, allowing for unlimited nested parentheses. Without the use of recursion, the best that can be done is to use a pattern that matches up to some fixed depth of nesting. It is not possible to handle an arbitrary nesting depth. Perl 5.6 has provided an experimental facility that allows regular expressions to recurse (amongst other things). It does this by interpolating Perl code in the expression at run time, and the code can refer to the expression itself. A Perl pattern to solve the parentheses problem can be created like this:

```
$re = qr{\( (?: (?>[^()]+) | (?p{$re}) )* \)}x;
```

The (?p{...}) item interpolates Perl code at run time, and in this case refers recursively to the pattern in which it appears. Obviously, PCRE cannot support the interpolation of Perl code. Instead, the special item (?R) is provided for the specific case of recursion. This PCRE pattern solves the parentheses problem (assume the PCRE_EXTENDED option is set so that white space is ignored):

```
\( ( (?>[^()]+) | (?R) )* \)
```

First it matches an opening parenthesis. Then it matches any number of substrings which can either be a sequence of non-parentheses, or a recursive match of the pattern itself

(i.e. a correctly parenthesized substring). Finally there is a closing parenthesis.

This particular example pattern contains nested unlimited repeats, and so the use of a once-only subpattern for matching strings of non-parentheses is important when applying the pattern to strings that do not match. For example, when it is applied to

```
(aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa()
```

it yields "no match" quickly. However, if a once-only subpattern is not used, the match runs for a very long time indeed because there are so many different ways the + and * repeats can carve up the subject, and all have to be tested before failure can be reported.

The values set for any capturing subpatterns are those from the outermost level of the recursion at which the subpattern value is set. If the pattern above is matched against

```
(ab(cd)ef)
```

the value for the capturing parentheses is "ef", which is the last value taken on at the top level. If additional parentheses are added, giving

```
\( ( ( (?>[^()]+) | (?R) )* ) \)
   ^                        ^
   ^                        ^ the string they capture is
```

"ab(cd)ef", the contents of the top level parentheses. If there are more than 15 capturing parentheses in a pattern, PCRE has to obtain extra memory to store data during a recursion, which it does by using pcre_malloc, freeing it via pcre_free afterwards. If no memory can be obtained, it saves data for the first 15 capturing parentheses only, as there is no way to give an out-of-memory error from within a recursion.

Chapter 20. Grammar definition

This chapter gives a description of the formal grammar of the language. It is not meant as a precise definition of the language. The Aikido grammar is not a regular grammar. The Aikido parser is tries to be clever and make sense of the user's input. In particular, the user can omit the semicolon characters from statements if the end of the statement is obvious. This feature makes it very difficult (if not impossible) to describe the grammar in a way that could be used as input to a regular grammar checker (such as *yacc*).

20.1. Program structure

A program consists of a sequence of statements

program:
statement-sequence

statement-sequence:
statement
statement-sequence statement

statement:
macro-instantiation
null-statement
compound-statement
expression
variable-declaration
static-declaration
block-declaration
block-extension
if-statement
break-statement
continue-statement
return-statement
while-statement
do-statement
for-statement
foreach-statement
switch-statement
macro-statement
inner-statement
try-statement
throw-statement
import-statement
access-control-statement
using-statement
delete-statement
synchronized-statement

static-declaration:
static *variable-declaration*
static *block-declaration*

20.2. Declarations

A declaration is used to tell the parser about the existence of a named thing. This 'thing' can be a variable or a block.

20.2.1. Variables

A variable is introduced by the keyword **const**, **var** or **generic**. Constants and regular variables must be initialized with an expression. Generic variables need not be initialized.

variable-declaration:
    constant-declaration
    regular-var-declaration
    generic-var-declaration

constant-declaration:
    const *var-decl-sequence*

regular-var-declaration:
    var *var-decl-sequence*

generic-var-declaration:
    generic *gen-var-decl-sequence*

var-decl-sequence:
    var-decl
    var-decl-sequence , var-decl

var-decl:
    identifier = expression

gen-var-decl-sequence:
    gen-var-decl
    gen-var-decl-sequence , gen-var-decl

gen-var-decl:
    identifier
    identifier = expression

20.2.2. Blocks

A block is a package, class, function, thread, operator or monitor. In this section we also include enumerations which, although are not blocks in the same sense as a package, are similar in construction.

block-declaration:
    package-declaration
    class-declaration
    function-declaration
    thread-declaration
    monitor-declaration

enum-declaration
operator-declaration
interface-declaration

package-declaration:
package *package-name block-definition*

class-declaration:
class *identifier block-definition*

function-declaration:
function *identifier block-definition*
native function$_{opt}$ *identifier native-parameters*

thread-declaration:
thread *identifier block-definition*

monitor-declaration:
monitor *identifier block-definition*

enum-declaration:
enum *identifier super-enum*$_{opt}$ *enum-body*

operator-declaration:
operator *operator-name block-definition*

interface-declaration:
interface *identifier interface-definition*

Enumeration structure. Enumerations can have a super-enum (be derived from another enumeration). Their body consists of a brace-enclosed set of identifiers.

super-enum:
extends *identifier*

enum-body:
{ enum-ident-seq$_{opt}$ *}*

enum-ident-seq:
enum-identifier
enum-ident-seq , enum-identifier

enum-identifier:
identifier enum-id-value$_{opt}$

enum-id-value:
= constant-expression

Operator and package names are not simple identifiers. An operator is one of a set of built-in operator tokens. A package name is a set of identifiers separated by dots.

operator-name: one of

*	+	-	/	%	~	^	!	&
|	<<	>>	>>>	<	>	<=	>=	==

!= -> sizeof typeof foreach cast [] ( ) in

package-name:
identifier
identifier . identifier

A block definition consists of a set of parameters, a super-block definition and the block body. Alternatively it may be simply an ellipsis (...) meaning that it is a forward declaration of the block.

block-definition:
block-parameters$_{opt}$ function-result-type$_{opt}$ superblock-definition$_{opt}$ block-body
ellipsis

interface-definition:
superblock-definition$_{opt}$ interface-body
ellipsis

block-parameters:
(parameter-decl-sequence)

parameter-decl-sequence:
parmeter-decl
parameter-decl-sequence , parameter-decl

parameter-decl:
parameter-decl-specifier$_{opt}$ access-mode$_{opt}$ identifier para-type$_{opt}$ para-default$_{opt}$
ellipsis

parameter-decl-specifier:
var
const

access-mode:
private
protected
public

para-type:
: postfix-expression

para-default:
= expression

function-result-type:
: postfix-expression

superblock-definition:
extends *identifier implements-clause$_{opt}$ superblock-parameters$_{opt}$*

implements-clause:
implements *interface-sequence*

interface-sequence:
identifier

interface-sequence , identifier

superblock-parameters:
expression
superblock-parameters , expression

block-body:
{ statement-sequence$_{opt}$ }

interface-body:
{ interface-member-sequence$_{opt}$ }

interface-member-sequence:
interface-member
interface-member-sequence interface-member

interface-member:
interface-member-type identifier block-parameters
***operator** operator-name block-parameters*

native-parameters:
(** native-para-list **)

native-para-list:
native-para
*native-para-list **,** native-para*

native-para:
identifier
***var** identifier*
ellipsis

Block extension allows us to extend an existing block. The syntax does not specify the block type as this is known from the block name. The block being extended must be in scope (its name is a simple identifier). The block may be a regular block or an enumeration.

block-extension:
***extend** identifier externsion-body*

extension-body:
extension-parameters$_{opt}$ block-body
enum-body

extension-parameters:
(ext-para-decl-sequence)

ext-para-decl-sequence:
ext-parm-decl
ext-para-decl-sequence , ext-para-decl

ext-para-decl:
parameter-decl-specifier$_{opt}$ access-mode$_{opt}$ identifier para-type$_{opt}$ para-default

20.3. Statements

Statements are the meat of the program, providing the code to be executed. An expression and a declaration are also statements. Note that there is no semicolon terminator on the statements themselves. Generally, semicolons are ignored when they are found except in certain circumstances. See section 7.2 for details of the semantics of this.

compound-statement:
{ statement-sequence$_{opt}$ }

null-statement:
;

if-statement:
***if** (expression) statement*
***if** (expression) statement **else** statement*
***if** (expression) statement elif-clause-seq*
***if** (expression) statement elif-clause-seq **else** statement*

elif-clause-seq:
elif-clause
elif-clause-seq elif-clause

elif-clause:
***elif** (expression) statement*

break-statement:
break

continue-statement:
continue

return-statement:
***return** expression$_{opt}$*

while-statement:
***while** (expression) statement*

do-statement:
***do** statement **while** (expression)*

for-statement:
***for** (for-init$_{opt}$; expression$_{opt}$; expression$_{opt}$) statement*

for-init:
expression
***var** identifier = expression*

foreach-statement:
***foreach** identifier **in**$_{opt}$ expression statement*
***foreach** identifier **in**$_{opt}$ range-expression statement*

switch-statement:
***switch** (expression) switch-body*

switch-body:
{ switch-clause-seq }

switch-clause-seq:
switch-clause
switch-clause-seq switch-clause

switch-clause:
***case** expression : statement*
***default** : statement*

try-statement:
***try** statement catch-clause*

catch-clause:
***catch** (identifier) statement*

throw-statement:
***throw** expression*

import-statement:
***import** import-ident-list*
***import** string-literal*

import-ident-list:
wildcarded-identifier
import-ident-list . wildcarded-identifier

access-control-statement:
global-access-control
local-access-control

global-access-control:
access-mode :

local-access-control:
access-mode statement

using-statement:
***using** using-ident-list*

using-ident-list:
using-ident
using-ident-list , using-ident

using-ident:
identifier
using-ident . identifier

delete-statement:
***delete** expression*

synchronized-statement:
***synchronized** (expression) statement*

synchronized *statement*

Macros are slightly different in structure than a regular statement. They are textual entities that buffer up a block of text for later when they are instantiated. They don't care what is in the block until the instantiation is done.

macro-statement:
macro *identifier macro-args*$_{opt}$ *super-macro*$_{opt}$ *macro-body*

macro-args:
macro-arg
macro-args , macro-arg

macro-arg:
identifier macro-arg-default$_{opt}$

macro-arg-default:
= string-literal

super-macro:
extends *identifier super-macro-args*$_{opt}$

super-macro-args:
rest of characters up to line-feed

macro-body:
{ ***line-feed*** *macro-body-parts }*

macro-body-parts:
any character sequence

inner-statement:
ellipsis

macro-instantiation:
macro-name macro-actual-parameters$_{opt}$ *inner-block*$_{opt}$

macro-name:
identifier

macro-actual-parameters:
macro-act-para-seq

macro-act-para-seq:
macro-act-para$_{opt}$
macro-act-para-seq , macro-act-para$_{opt}$

macro-act-para:
character sequence of any character except comma and open brace

inner-block:
{ ***line-feed*** *statement-sequence }*

20.4. Expressions

Expressions are presented in their usual recursive descent form. The lowest priority expressions are listed first.

expression:
assignment-expression

constant-expression:
const-or-expression

assignment-expression:
stream-expression
assignment-expression assignment-op stream-expression

assignment-op: one of
= += -= *= /= %= &= |= ^=
<<= >>= >>>=

stream-expression:
conditional-expression
stream-expression -> conditional-expression

conditional-expression:
logical-or-expression
logical-or-expression ? expression : conditional-expression

logical-or-expression:
logical-and-expression
logical-or-expression || logical-and-expression

logical-and-expression:
or-expression
logical-and-expression && or-expression

or-expression:
xor-expression
or-expression | xor-expression

xor-expression:
and-expression
xor-expression ^ and-expression

and-expression:
equality-expression
and-expression & equality-expression

equality-expression:
relational-expression
equality-expression == relational-expression
equality-expression != relational-expression

relational-expresion:
shift-expression
relational-expression < shift-expression

relational-expression <= shift-expression
relational-expression > shift-expression
relational-expression >= shift-expression
relational-expression ***instanceof*** *shift-expression*
relational-expression ***in*** *range-expression*

range-expression:
shift-expression
shift-expression ellipsis shift-expression

shift-expression:
additive-expression
shift-expression << additive-expression
shift-expression >> additive-expression
shift-expression >>> additive-expression

additive-expression:
mult-expression
additive-expression + mult-expression
additive-expression - mult-expression

mult-expression:
unary-expression
*mult-expression * unary-expression*
mult-expression /unary-expression
mult-expression %unary-expression

unary-expression:
+ unary-expression
- unary-expression
! unary-expression
~ unary-expression
++ unary-expression
-- unary-expression
sizeof *expression*
typeof *expression*
cast-expression
postfix-expression

cast-expression:
cast *< postfix-expression > (expression)*

The postfix expression set cause the most problems with the “Natural End” of a statement rules. Any operator used as a postfix expression operator cannot be preceded by a linefeed character. The text **{no linefeed}** is not part of the syntax, but signifies that no linefeed can occur at that point.

postfix-expression:
primary-expression
postfix-expression **{no linefeed}**++
postfix-expression **{no linefeed}** --
subscript-expression
member-access-expression
call-expression

subscript-expression:
postfix-expression **{no linefeed}** *[expression]*

member-access-expression:
postfix-expression . operator operator-name
postfix-expression . identifier

call-expression:
postfix-expression **{no linefeed}** *(expression-list)*

expression-list:
expression
expression-list , expression

primary-expression:
(expression)
identifier
integer-number
real-number
character-literal
string-literal
true
false
null
vector-literal
map-literal
new-expression
direct-operator-expression
inline-block-expression
anonymous-block-expression

vector-literal:
[expression-list$_{opt}$]

map-literal:
{ map-entry-seq$_{opt}$ }

map-entry-seq:
map-entry
map-entry-seq , map-entry

map-entry:
conditional-expression = conditional-expression

new-expression:
***new** new-vector-specifier*
***new** new-identifier new-vector-specifier$_{opt}$ call-expression*

new-vector-specifier:
new-vector
new-vector-specifier new-vector

new-vector:
[expression]

new-identifier:
identifier
new-identifier . identifier

direct-operator-expression:
operator *operator-name*

inline-block-expression:
` *statement-sequence* `

anonymous-block-expression:
anon-function-expression
anon-thread-expression
anon-class-expression
anon-monitor-expression
anon-package-expression

anon-function-expression:
function *anon-block-definition*

anon-thread-expression:
thread *anon-block-definition*

anon-class-expression:
class *anon-block-definition*

anon-monitor-expression:
monitor *anon-block-definition*

anon-package-expression:
package *anon-block-definition*

anon-block-definition:
$block\text{-}parameters_{opt}$ $superblock\text{-}definition_{opt}$ *block-body*

Constant expressions are integer-only expressions that can be evaluated at parse time. They can contain identifiers but these must be an existing enumeration constant.

const-or-expression:
const-xor-expression
const-or-expression | *const-xor-expression*

const-xor-expression:
const-and-expression
const-xor-expression ^ *const-and-expression*

const-and-expression:
const-equality-expression
const-and-expression & *const-equality-expression*

const-equality-expression:
const-relational-expression
const-equality-expression == *const-relational-expression*
const-equality-expression != *const-relational-expression*

const-relational-expresion:

const-shift-expression
const-relational-expression < const-shift-expression
const-relational-expression <= const-shift-expression
const-relational-expression > const-shift-expression
const-relational-expression >= const-shift-expression

const-shift-expression:
const-additive-expression
const-shift-expression << const-additive-expression
const-shift-expression >> const-additive-expression
const-shift-expression >>> const-additive-expression

const-additive-expression:
const-mult-expression
const-additive-expression + const-mult-expression
const-additive-expression - const-mult-expression

const-mult-expression:
const-unary-expression
*const-mult-expression * const-unary-expression*
const-mult-expression /const-unary-expression
const-mult-expression %const-unary-expression

const-unary-expression:
+ const-unary-expression
- const-unary-expression
! const-unary-expression
~ const-unary-expression
const-primary-expression

const-primary-expression:
(constant-expression)
identifier
integer-number
character-literal
true
false

20.5. Lexical conventions

These are the low level constructs of the language.

integer-number
hex-number
decimal-number
octal-number
binary-number

hex-number:
0x *hex-digit-sequence*

decimal-number:

digit-sequence

octal-number:
0 *octal-digit-sequence*

binary-number:
0b *binary-digit-sequence*

hex-digit-sequence:
hex-digit
hex-digit-sequence hex-digit

digit-sequence:
digit
digit-sequence digit

octal-digit-sequence:
octal-digit
octal-digit-sequence octal-digit

binary-digit-sequence:
binary-digit
binary-digit-sequence binary-digit

hex-digit: one of
0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

decimal-digit: one of
0 1 2 3 4 5 6 7 8 9

octal-digit: one of
0 1 2 3 4 5 6 7

binary-digit: one of
0 1

real-number:
decimal-digit-sequence$_{opt}$ *. decimal-digit-sequence exponent*$_{opt}$
decimal-digit-sequence exponent

exponent:
e sign decimal-digit-sequence
E sign decimal-digit-sequence

sign: one of
+ -

identifier:
letter-or-underscore identifier-character-sequence

identifier-character-sequence:
identifier-character
identifier-character-sequence identifier-character

letter-or-underscore: one of
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
_

identifier-character: one of
decimal-digit letter-or-underscore

wildcarded-identifier:
wildcard-character
wildcarded-identifier wildcard-character

wildcard-character:
any character except .

string-literal:
“ string-character-sequence “

character-literal:
‘ character ‘

string-character-sequence:
string-character
string-character-sequence string-character

string-character:
any character except \, line-feed or “
escape-sequence

character:
any character except \, line-feed or ‘
escape-sequence

escape-sequence:
simple-escape
octal-escape
hex-escape

simple-escape: one of
\a \b \f \n \r \t \v
\\ \’ \”

octal-escape:
\ octal-digit
\ octal-digit octal-digit
\ octal-digit octal-digit octal-digit

hex-escape:
\x hex-digit-sequence

ellipsis:
...
..

Index