OAK *and* WEBRUNNER: *What and Why*

James Gosling, May 1994

Introduction

OAK is a programming language and environment that was designed to solve a number of problems in modern programming practice. It started as a part of a larger project to develop advanced software for consumer electronics. These are small reliable portable distributed real-time embedded systems. When we started the project, we intended to use C++, but we encountered a number of problems. Initially these were just compiler technology problems, but as time passed we encountered a set of problems that were best solved by changing the language.

WEBRUNNER is an application built on top of OAK and is an environment that makes the Internet "come alive". It builds on the network browsing techniques established by Mosaic and expands them by adding dynamic behavior that transforms static documents into living applications. Current documents in Mosaic are limited to text, illustrations, limited sounds and videos. WEBRUNNER adds to this the ability to add arbitrary behavior: from interactive science experiments in educational material, to games, specialized shopping applications, interactive advertising, customized newspapers, ... It provides a way for users to access applications in a whole new way. Software transparently migrates across the network. There's no such thing as "installing" software. It just comes when you need it (after, perhaps, asking you to pay for it...). "Content" developers for the World Wide Web don't have to worry about whether or not some special piece of software is installed in the viewers system, it just gets there automatically. This frees them from the boundaries of the fixed media types like images and text & lets them do whatever they'd like.

This document contains a lot of technical words and acronyms that may be unfamiliar. You may want to look at the glossary on page 11.

OAK

> OAK: A simple, object oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high performance, multithreaded, dynamic language.

One way to characterize a system is with a set of buzzwords. We use a standard set of them in describing OAK. The rest of this section is an explanation of what we mean by those buzzwords and the problems that we were trying to solve.

> ❧ *Archimedes Inc. is a fictitious software company that produces software to teach about basic physics.This software is designed to interact with the user, providing not only text and illustrations in the manner of a traditional textbook, but also providing a set of software lab benches on which experiments can be set up and their behavior simulated. For example, the most basic one allows students to put together levers and pulleys and see how they act. A narrative of their trials and tribulations is used to provide examples of the concepts presented.*

Simple

We wanted to build a system that could be programmed easily without a lot of esoteric training and which leveraged today's standard practice. Most programmers working these days use C, and most doing object-oriented programming use C++. So even though we found that C++ was unsuitable, we tried to stick as close as possible to C++ in order to make the system more comprehensible.

OAK omits many rarely used, poorly understood, confusing features of C++ that in our experience bring more grief than benefit. This primarily consists of operator overloading (although it does have method overloading), multiple inheritance, and extensive automatic coercions.

Paradoxically, we were able to simplify the programming task by making the system somewhat more complicated. A good example of a common source of complexity in many C and C++ applications is storage management: the allocation and freeing of memory. OAK does automatic garbage collection — this not only makes the programming task easier, it also dramatically cuts down on bugs.

> ❧ *The folks at Archimedes wanted to spend their time thinking about levers and pulleys, but instead spent a lot of time on mundane programming tasks. Their central expertise was teaching, not programming. One of the most complicated of these programming tasks was figuring out where memory was being wasted across their 20K lines of code.*

Another aspect of simple is small. One of the goals of OAK is to enable the construction of software that can run stand-alone in small machines. The size of the basic interpreter and class support is about 30K bytes, adding the basic standard libraries and thread support (essentially a self-contained microkernel) brings it up to about 120K.

Object-Oriented

This is, unfortunately, one of the most overused buzzwords in the industry. But object-oriented design is still very powerful since it facilitates the clean definition of interfaces and makes it possible to provide reusable "software ICs".

A simple definition of object oriented design is that it is a technique that focuses design on the data (=objects) and on the interfaces to it. To make an analogy with carpentry, an "object oriented" carpenter would be mostly concerned with the chair he was building, and secondarily with the tools used to make it; a "non-OO" carpenter would think primarily of his tools. This is also the mechanism for defining how modules "plug&play".

The object-oriented facilities of OAK are essentially those of C++, with extensions for more dynamic method resolution that came from Objective C.

The folks at Archimedes had lots of kinds of things in their simulation. Among them, ropes and elastic bands. In their initial C version of the product, they ended up with a pretty big system because they had to write separate software for describing ropes versus elastic bands. When they re-wrote their application in an object oriented style, they found they could define one basic object that represented the common aspects of ropes and elastic bands, and then ropes and elastic bands were defined as variations (subclasses) of the basic type. When it came time to add chains, it was a snap because they could build on what had been written before, rather than writing a whole new object simulation.

Distributed

Disabled in release 0.1, incomplete in 0.2, will eventually be superceeded by the use of IDL..

Networking is integrated into the language and runtime system and is hence (almost) transparent. Objects can be remote: when an application has a pointer to an object, that object may exist on the same machine, or some other machine on the network. Method invocations on remote objects are turned into RPCs (Remote Procedure Calls).

A distributed application looks very much like a non-distributed one. Both cases use an essentially similar programming model. The distributed case does, however, require that applications pay some attention to the consequences of network failures. The system deals with much of it automatically, but some of it does need to be dealt with on a case-by-case basis.

The RPC layer of the network protocol is implicitly defined by the OO interface. It can layer on most datagram protocols. We currently use IP, the Internet datagram Protocol.

The folks at Archimedes initially built their stuff for CD ROM. But they had some ideas for interactive learning games that they'd like to try out for their next product. For example, they wanted to allow students on different computers to cooperate in building a machine to be simulated. But all the networking systems they'd seen were complicated and required esoteric software specialists. So they gave up.

Robust

OAK is intended for writing programs that need to be reliable in a variety of ways. There is a lot of emphasis on early checking for possible problems, later dynamic (runtime) checking, and on eliminating situations which are error prone.

One of the advantages of a strongly typed language (like C++) is that it allows extensive compile-time checking so bugs can be found early. Unfortunately, C++ inherits a number of loopholes in this checking from C, which was relatively lax (the major issue is method/procedure declarations). In OAK, we require declarations and do not support C style implicit declarations.

The linker understands the type system and repeats many of the type checks done by the compiler to guard against version mismatch problems.

A mentioned before, automatic garbage collection avoids storage allocation bugs.

The single biggest difference between OAK and C/C++ is that OAK has a pointer model that eliminates the possibility of overwriting memory and corrupting data. Rather than having pointer arithmetic, OAK has true arrays. This allows subscript checking to be performed. And it is not possible to turn an arbitrary integer into a pointer by casting.

> ❧ *The folks at Archimedes had their application basically working in C pretty quickly. But their schedule kept slipping because of all the small bugs that kept slipping through. They had lots of trouble with memory corruption, versions out-of-sync and interface mismatches. What they gained because C let them pull strange tricks in their code, they paid for in Quality Assurance time. They also had to reissue their software after the first release because of all the bugs that slipped through.*

While OAK doesn't pretend to make the QA problem go away, it does make it significantly easier.

Very dynamic languages like Lisp, TCL and Smalltalk are often used for prototyping. One of the reasons for their success at this is that they are very robust: you don't have to worry about freeing or corrupting memory. Programmers can be relatively fearless about dealing with memory because they don't have to worry about it getting messed up. OAK has this property and it has been found to be very liberating. Another reason given for these languages being good for prototyping is that they don't require you to pin down decisions early on. OAK has exactly the opposite property: it forces you to make choices explicitly. Along with these choices come a lot of assistance: you can write method invocations and if you get something wrong, you get told about it early, without waiting until you're deep into executing the program. You can also get a lot of flexibility by using interfaces instead of classes.

Secure

OAK is intended to be used in networked/distributed situations. Toward that end a lot of emphasis has been placed on security. OAK enables the construction of virus-free, tamper-free systems. The authentication techniques are based on public-key encryption.

There is a strong interplay between “robust” and “secure”. For example, the changes to the semantics of pointers make it impossible for applications to forge access to data structures or to access private data in objects that they do have access to. This closes the door on most activities of viruses.

Not included in release 0.1 or 0.2.

There is a mechanism for defining approval seals for software modules and interface access. For example, it is possible for a system built on OAK to say “only software with a certain seal of approval is allowed to be loaded” and it is possible for individual modules to say “only software with a certain seal of approval is allowed to access my interface”. These approval seals cannot be forged since they are based on public-key encryption.

> ❧ *Someone wrote an interesting “patch” to the PC version of the Archimedes system. They posted this patch to one of the major bulletin boards. Since it was easily available and added some interesting features to the system, lots of people downloaded it. It hadn’t been checked out by the folks at Archimedes, but it seemed to work. Until the next April first when thousands of folks discovered rude pictures popping up in their children’s lessons. Needless to say, even though they were in no way responsible for the incident, the folks at Archimedes still had a lot of damage to control.*

Architecture Neutral

OAK was designed to support applications on networks. In general, networks are composed of a variety of systems with a variety of CPU and operating system architectures. In order for an OAK application to be able to execute anywhere on the network, the compiler generates an architecture neutral object file format — the compiled code is executable on many processors, given the presence of the OAK runtime.

This is useful not only for networks but also for single system software distribution. In the present personal computer market, application writers have to produce versions of their application that are compatible with the IBM PC and with the Apple Macintosh. With the PC market (through Windows/NT) diversifying into many CPU architectures, and Apple moving off the 68000 towards the PowerPC, this makes the production of software that runs on all platforms almost impossible. With OAK, the same version of the application runs on all platforms.

The OAK compiler does this by generating bytecode instructions which have nothing to do with a particular computer architecture. Rather, they are designed to be both easy to interpret on any machine and easily translated into native machine code on the fly.

> ❧ *Archimedes is a small company. They started out producing their software for the PC since that was the largest market. After a while, they were a large enough company that they could afford to do a port to the Macintosh, but it was a pretty big effort and didn’t really pay off. They couldn’t afford to port to the PowerPC Mac or MIPS NT machine. They couldn’t “catch the new wave” as it was happening, and a competitor jumped in...*

Portable

Being architecture neutral is a big chunk of being portable, but there's more to it than just that. Unlike C and C++ there are no "implementation dependent" aspects of the specification. The sizes of the primitive data types are specified, as is the behaviour of arithmetic on them. For example, "int" always means a signed twos complement 32 bit integer, and "float" always means a 32 bit IEEE 754 floating point number. Making these choices is feasable in this day and age because essentially all interesting CPUs share these characteristics.

The libraries that are a part of the system define portable interfaces. For example, there is an abstract Window class and implementations of it for Unix, Windows and the Mac[†].

The OAK system itself is quite portable. The new compiler is written in OAK and the runtime is written in ANSI C with a clean portability boundary. The portability boundary is essentially POSIX.

Interpreted

The OAK interpreter can execute OAK bytecodes directly on any machine to which the interpreter has been ported. And since linking is a more incremental & lightweight process, the development process can be much more rapid and exploratory.

As a part of the bytecode stream, more compile-time information is carried over and available at runtime. This is what the linker's type checks are based on, and what the RPC protocol derivation is based on. It also makes programs more amenable to debugging.

❧ *The programmers at Archimedes spent a lot of time waiting for programs to compile and link. They also spent a lot of time tracking down senseless bugs because some changed source files didn't get compiled (despite using a fancy "make" facility), which caused version mismatches; and they had to track down procedures that were declared inconsistently in various parts of their programs. Another couple of months lost in the schedule.*

High Performance

While the performance of interpreted bytecodes is usually more than adequate, there are situations where higher performance is required. The byte code can be translated on the fly (at runtime) into machine code for the particular CPU the application is running on. For those used to the normal design of a compiler and dynamic loader, this is somewhat like putting the final machine code generator in the dynamic loader.

The byte code format was designed with this in mind, so the actual process of generating machine code is generally simple. Reasonably good code is produced: it does automatic register allocation and the compiler does some optimization when it produces the bytecode.

In interpreted code we're getting about 300,000 method calls per second on an SS10. The performance of bytecodes converted to machine code is almost indistinguishable from native C or C++.

[†] The Windows and Mac versions aren't complete yet.

When Archimedes was starting up, they did a prototype in SmallTalk. This impressed the investors enough that they got funded, but it didn't really help them produce their product: in order to make their simulations fast enough and the system small enough, it had to be rewritten in C.

Multithreaded

There are many things going on at the same time in the world around us. Multithreading is a way of building applications that are built out of multiple threads[†]. Unfortunately, writing programs that deal with many things happening at once can be much more difficult that writing in the conventional single-threaded C and C++ style.

OAK has a sophisticated set of synchronization primitives that are based on the widely used monitor and condition variable paradigm that was introduced by C.A.R.Hoare[‡]. By integrating these concepts into the language they become much more easy to use and robust. Much of the style of this integration came from Xerox's Cedar/Mesa system.

Other benefits are better interactive responsiveness and realtime behaviour. This is limited, however, by the underlying platform: stand-alone OAK runtime environemnts have good realtime behaviour. Running on top of other systems like Unix, Windows, the Mac or NT limits the realtime responsivness to that of the underlying system.

Lots of things were going on at once in their simulations. Ropes were being pulled, wheels were turning, levers were rocking, and input from the user was being tracked. Because they had to write this all in a single threaded form, all the things that happen at the same time, even though they had nothing to do with each other, had to be manually intermixed. Using an "event loop" made things a little cleaner, but it was still a mess. The system became fragile and hard to understand.

They were pulling in data from all over the net. But originally they were doing it one chunk at a time. This serialized network communication was very slow. When they converted to a multithreaded style, it was trivial to overlap all of their network communication.

Dynamic

In a number of ways, OAK is a more dynamic language than C or C++. It was designed to adapt to an evolving environment.

For example, one of the big problems with using C++ in a production environment is a side-effect of the way that it is always implemented. If company A produces a class library (a library of plug&play components) and company B buys it and uses it in their product, then if A changes it's library and distributes a new release then B will almost certainly have to recompile and redistribute their software. In an environment where the end user gets A and B's software independently (say A is an OS vendor and B is an application

[†] Threads are sometimes also called lightweight processes or execution contexts.

[‡] 1974. Hoare, C.A.R. *Monitors: An Operating System Structuring Concept*, Comm. ACM **17**, 10:549-557 (October)

vendor) then if A distributes an upgrade to its libraries then all of the users software from B will break. It is possible to avoid this problem in C++, but it is extraordinarily difficult and it effectively means not using any of the language's OO features directly.

> ❦ *Archimedes built their product using the object oriented graphics library from 3DPC Inc. 3DPC released a new version of the graphics library which several computer manufacturers bundled with their new machines. Customers of Archimedes that bought these new machines discovered to their dismay that their old software no longer worked. [In real life, this only happens on Unix systems. In the PC world, 3DPC would never have released such a library: their ability to change their product and use C++'s object oriented features is severely hindered]*

By making these interconnections between modules later, OAK completely avoids these problems and makes the use of the OO paradigm much more straightforward. Libraries can freely add new methods and instance variables without any effect on their clients.

OAK understands the concept of an *interface*. An interface is a concept borrowed from Objective C and is similar to a class. An interface is simply a specification of a set of methods that an object responds to. It does not include any instance variables or implementation. Interfaces can be multiply-inherited (unlike classes) and they can be used in a more flexible way than the usual rigid class inheritance structure.

Classes have a runtime representation: there is a class named *Class*, instances of which contain runtime class definitions. From an object you can find out what class it belongs to. If, in a C or C++ program, you have a pointer to an object but you don't know what type of object it is, there is no way to find out. In OAK, finding out based on the runtime type information is straightforward. For example, type conversions (casts) are checked at runtime in OAK, whereas in C and C++, the compiler just trusts that you're doing the right thing.

It is also possible to lookup the definition of a class given a string containing its name. This means that you can compute a data type name and have it trivially dynamically linked into the running system.

> ❦ *To expand their revenue stream, the folks at Archimedes wanted to architect their product so that new aftermarket plug-in modules could be added to extend the system. This was possible on the PC, but just barely. They had to hire a couple of new programmers because it was so complicated. This also added terrible problems when debugging.*

WEBRUNNER

The internet is a vast sea of data represented in many formats and stored on many hosts. Mosaic is a browser for that data. It allows people to easily look at data on one machine, then follow links to data on other machines. It integrates the function of fetching the data with figuring out what it is and displaying it.

One of the most important file types it understands is HTML, the hypertext markup language. HTML allows text data object to imbed simple formatting information and references to other objects.

Mosaic has sparked a revolution in the way that people look at the internet. In a sense, Mosaic brings no new function to the internet, it just makes transparent the things that you could do awkwardly before. It conceptually weaves together all of these systems into a web that has a unified appearance to those who use it.

There is also a new twist happening here in the concept of client/server computing. As most people think of it, client server computing consists of a big centralized server that clients connect to for a long time and access data and applications on that server. It is roughly a star with a big server in the middle and clients arrayed around it. The new model is a wide-spread web with short-lived connections between clients and many servers. The controlling intelligence shifts from the server to the client. The answer to "who's in charge?" shifts from the server to the client.

But the objects browsed by Mosaic are static and the set of them that it understands is fixed. There is, however, a simple extension mechanism: there is a method for defining patterns which are looked for in the data, and the names of applications to launch when the patterns are found. The application must already exist on the machine, and there is no mechanism for viewing such objects embedded in larger objects, like a hypertext page.

> ❧ *The folks at Archimedes had a successful CD ROM business. They heard all the fuss being made about the NII and the Internet and wanted to get involved. While part of their system was roughly an electronic book, a lot of it was a variety of simulations for teaching various concepts. To make their product worthwhile, they needed to get these simulations across, but there were no facilities for this in HTML. They thought about porting their software to internet hosts, but there are so many kinds of hosts that the task is impossible.*

WEBRUNNER is a system built on top of OAK that brings to the concept of hypertext access to the Internet pioneered by Mosaic a new dimension of dynamic behavior. It brings the net to life. OAK allows code to be dynamically linked into a running application in a manner that is independent of the CPU architecture of the system it is running on, and providing strong guarantees about the security of the system — eliminating the possibility of viruses.

For example, someone could write an OAK program following the WEBRUNNER API that implemented an interactive chemistry simulation. Then someone browsing the net with the WEBRUNNER browser could easily get this simulation and interact with it, rather than just having a static picture with some text. They can do this and be assured that the code which brings the chemistry experiment to life, and doesn't also contain malicious code that damages is system. Code which attempts to be malicious or which has bugs essentially can't get out of the tight walls placed around it by the security/robustness features of OAK.

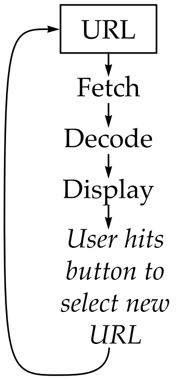
Using these dynamic facilities, service providers can define new types of data and behavior that suit their situation, rather than being bound by a fixed set of objects.

Being multithreaded, WebRunner allows these live objects to exhibit very animated and responsive behavior.

> *What kids really liked about Archimedes product is that it got them involved. They could learn all the stuff and then they could apply it. The section on Newtonian mechanics not only taught Newtons laws, but it demonstrated them in action and let the kids pilot a spaceship to the moon. Early internet experiments didn't work since the material was dry, like a book — while it was indexed and linked together, the kids weren't drawn in. But later versions that used* WebRunner *on the Internet brought the material back to life.*

Implementation details

The basic structure of the WebRunner browser is instructive. It is easiest understood from the operation of Mosaic:



Mosaic starts with a URL and fetches the object using the specified protocol. The *host* and *localinfo* fields are passed to the protocol handler. The result of this is a bag of bytes that contains the object that has been fetched. These bytes are inspected to determine the type of the data (for example, HTML document or JPEG image). From this type information, code to manipulate and view this data is invoked.

That's all there is to Mosaic. It's essentially very simple. But despite this, it is actually huge since it contains handlers for all of these data types.

In contrast, the WebRunner browser will be very small since all of the protocol and data handlers are brought in from the outside. It exploits the achilles heel of Mosaic: all the protocol handlers and most of the object viewers are built in. For example, when it calls the protocol handler, it has a table that has a fixed list of protocols that it understands. the major twist in WebRunner is that instead of using this static table, WebRunner uses this type string to derive a class name. The protocol handler for this type is dynamically linked in if it is missing. They can be linked in from the local system, or they can be linked in from definitions stored on the host where the URL was found, or anywhere else on the net that the browser suspects might be a good place to look. Similarly, the code to handle different types of data objects and different ways of viewing them.

This may sound like a simple twist, but it's really explosive. It provides a way for users to access applications in a whole new way. Software transparently migrates across the network. There's no such thing as "installing" software. It just comes when you need it (after, perhaps, asking you to pay for it...). If you're a "content" developer for the World Wide Web you don't have to worry about whether or not some special piece of software is installed in the viewers system, it just gets there automatically. This frees them from the boundaries of the fixed media types like images and text & lets them do whatever they'd like.

Glossary

This paper is filled with all sorts of words and TLAs that my be hard to decipher. Here are the definitions of a few:

API

Application Programmer Interface. The specification of how a programming writing an application accesses the facilities of some object. Interfaces can be specified in OAK and C++ using classes. OAK also has a special interface syntax that allows interfaces that are more flexible than classes.

FTP

The basic internet File Transfer Protocol. It enables the fetching and storing of files between hosts on the internet. It is based on TCP/IP.

HTML

HyperText Markup Language. This is a file format, based on SGML, for hypertext documents on the internet. It is very simple and allows for the imbedding of images, sounds, video streams, form fields and simple text formatting. References to other objects are imbedded using URLs.

HTTP

Hypertext Transfer Protocol. This is the internet protocol used to fetch hypertext objects from remote hosts. It is based on TCP/IP.

Internet

An enormous network consisting of literally millions of hosts from many organizations and countries around the world. It is physically put together from many smaller networks and is held together by a common set of protocols.

IP

Internet Protocol. The basic protocol of the internet. It enables the unreliable delivery of individual packets from one host to another. It makes no guarantees about whether or not the packet will be delivered, how long it will take, or if multiple packets will arrive in the order they were sent. Protocols built on top of this add the notions of connection and reliability.

JPEG

Joint Photographic Experts Group. An image file compression standard established by this group. It achieves tremendous compression at the cost of introducing distortions into the image which are almost always imperceptible.

Mosaic

A program that provides a simple GUI that enables easy access to the data stored on the internet. These may be simple files, or hypertext documents.

RPC

Remote Procedure Call. Executing what looks like a normal procedure call (or method invocation) by sending network packets to some remote host.

SGML Standardized, Generalized Markup Language. An ISO/ANSI/ECMA standard that specifies a way to annotate text documents with information about types of sections of a document. For example "this is a paragraph" or "this is a title".

TCP/IP Transmission Control Protocol based on IP. This is an internet protocol that provides for the reliable delivery of streams of data from one host to another.

TLA Three Letter Acronym.

URL Uniform Resource Locator. A standard for writing a textual reference to an arbitrary piece of data in the WWW. A URL looks like "*protocol*://*host*/*localinfo*" where *protocol* specifies a protocol to use to fetch the object (like HTTP or FTP), *host* specifies the internet name of the host on which to find it, and *localinfo* is a string (often a file name) passed to the protocol handler on the remote host.

WWW World Wide Web. The web of systems and the data in them that is the internet.

Oak Language Specification

This document is a preliminary specification of the Oak language. Both the specification and the language are subject to change. When a feature that exists in both Oak and ANSI C isn't explained fully in this specification, the feature should be assumed to work as it does in ANSI C. Send comments on the Oak Language and specification to oak-comments@firstperson.COM

1 *Program Structure*

The source code for an Oak program consists of one or more *compilation units*. Each compilation unit can contain only the following (in addition to white space and comments):

- a package statement (see "Packages" on page 32)
- import statements (see "Packages" on page 32)
- class declarations (see "Classes" on page 19)
- interface declarations (see "Interfaces" on page 31)

Although each Oak compilation unit can contain multiple classes or interfaces, at most one class or interface per compilation unit can be public (see "Classes" on page 19).

When Oak source code is compiled, the result is Oak bytecode. Oak bytecode consists of machine-independent instructions that can be interpreted efficiently by the Oak runtime system. For information on the Oak virtual machine see *The Oak Virtual Machine Specification*.

Implementation Note: In the current Oak implementation, each compilation unit is a file with a ".oak" suffix.

2 *Lexical Issues*

During compilation, the characters in Oak source code are reduced to a series of tokens. The Oak compiler recognizes five kinds of tokens: identifiers, keywords, literals, operators, and miscellaneous separators. Comments and *white space* such as blanks, tabs, line feeds, and formfeeds are not tokens, but they often are used to separate tokens.

Oak programs are written using the Unicode character set, or some character set that is converted to Unicode before being compiled.

2.1 Comments

The Oak language has four kinds of comments:

`// text` All characters from // to the end of the line are ignored.

`/* text */` All characters from /* to */ are ignored.

`/** text */` Like /*...*/, except that these comments are treated specially when they occur immediately before any declaration or when they occur on the same line as a declaration (even if after it). These comments indicate that the enclosed text should be included in automatically generated documentation as a description of the declared item.

`//* text` Like //, except that these comments, like /**...*/, indicate text to be included in automatically generated documentation. Any subsequent // comments with no intervening code are included in the automatically generated documentation for the declared item.

2.2 Identifiers

Identifiers must start with a letter, underscore ("_"), or dollar sign ("$"); subsequent characters can also contain digits (0-9). Oak uses the Unicode character set. For the purposes of determining what is a legal identifier the following are considered "letters:"

- The characters "A" through "Z"
- The characters "a" through "z"
- All Unicode characters with a character number above hex 00C0

Other characters valid after the first letter of an identifier include every character except those in the segment of Unicode reserved for special characters.

Thus, "garçon" and "Mjølner" are legal identifiers, but strings containing characters such as "¶" are not.

For more information on the Unicode standard see *The Unicode Standard, Worldwide Character Encoding*, Version 1.0, Volumes 1&2. The ftp address for Unicode, Inc. (formerly the Unicode Consortium) is unicode.org.

2.3 Keywords

The following identifiers are reserved for use as keywords. They can not be used in any other way.

boolean
break
byte
byvalue[a]
case
catch
char
class
const[b]
continue
default
do
double
else
extends
false
final
finally
float
for
goto
if
implements
import
instanceof
int
interface
long
new
null
package
private
public
return
short
static
super
switch
synchronized
this
threadsafe
throw
transient
true
try
void
while

[a] Reserved but currently unused.
[b] Reserved but currently unused.

2.4 Literals

Literals are the basic representation of any integer, floating point, boolean, character, or string value.

2.4.1 *Integer Literals*

Integers can be expressed in decimal (base 10), hexadecimal (base 16), or octal (base 8) format. A decimal integer literal consists of a sequence of digits (optionally suffixed as described below) *without* a leading 0 (zero). An integer can be expressed in octal or hexadecimal rather than decimal. A leading 0 (zero) on an integer literal means it is in octal; a leading 0x (or 0X) means hexadecimal. Hexadecimal integers can include digits (0-9) and the letters a-f and A-F. Octal integers can include only the digits 0-7.

Integer literals are of type **int** unless they are larger than 32-bits, in which case they are of type **long** (see “Integer Types” on page 17). A literal can be forced to be **long** by appending an `L` or `l` to its value.

The following are all legal integer literals:

```
2, 2L, 0777, 0xDeadBeef
```

2.4.2 *Floating Point Literals*

A floating point literal can have the following parts: a decimal integer, a decimal point (“.”), a fraction (another decimal number), an exponent, and a type suffix. The exponent part is an `e` or `E` followed by an integer, which can be signed. A floating point literal must have at least one digit, plus either a decimal point or `e` (or `E`). Some examples of floating point literals are:

```
3.1415   3.1E12   .1e12    2E12
```

As described in “Floating Point Types” on page 17, the Oak language has two floating point types: **float** (IEEE 754 single precision) and **double** (IEEE 754 double precision). You specify the type of a floating point literal as follows:

`2.0d` or `2.0D`	**double**
`2.0f` or `2.0F` or `2.0`	**float**

2.4.3 *Boolean Literals*

The **boolean** type has two literal values: `true` and `false`. See “Boolean Types” on page 18 for more information on boolean values.

2.4.4 *Character Literals*

A character literal is a character (or group of characters representing a single character) enclosed in single quotes. Characters have type **char** and are drawn from the Unicode character set (see “Character Types” on page 18). The following escape sequences allow for the representation of some non-graphic characters as well as the single quote, ‘ and the backslash \, in Oak code:

continuation	<newline>	\
new-line	NL (LF)	\n
horizontal tab	HT	\t
back space	BS	\b
carriage return	CR	\r
form feed	FF	\f
backslash	\	\\
single quote	‘	\’
double quote	"	\"
octal bit pattern	0ddd	\ddd
hex bit pattern	0xdd	\xdd
unicode char	0xdddd	\udddd

2.4.5 *String Literals*

A string literal is zero or more characters enclosed in double quotes. Each string literal is implemented as a String object (*not* as an array of characters). For example, “abc” creates an new instance of class String. The following are all legal string literals:

```
""      \\ the empty string
"\""
"This is a string"
"This is a \
 two-line string"
```

2.5 Operators and Miscellaneous Separators

The following characters are used in Oak source code as operators or separators:

+ – ! % ^ & * | ~ / > < ( ) { } [ ] ; ? : , . =

In addition, the following character combinations are used as operators:

++ — == <= >= != << >> >>> += –= *= /= &= |= ^=

%= <<= >>= >>>= || &&

For more information see “Operators” on page 33.

3 *Types*

Every variable and every expression has a type. Type determines the allowable range of values a variable can hold, allowable operations on those values, and the meanings of the operations. A number of built-in types are provided by the Oak language. Programmers can compose new types using the *class* and *interface* mechanisms (see “Classes” on page 19 and “Interfaces” on page 31).

The Oak language has two kinds of types: simple and composite. Simple types are those that cannot be broken down; they are atomic. The integer, floating point, boolean, and character types are all simple types. Composite types are built on simple types. The Oak language has three kinds of composite types—arrays, classes, and interfaces. Simple types and arrays are discussed in this section.

3.1 Integer Types

Integers in the Oak language are similar to those in C and C++, with two exceptions: all integer types are machine independent, and some of the traditional definitions have been changed to reflect changes in the world since C was introduced. The five integer types have widths of 8, 16, 32, and 64 bits and are signed.

Width	**Name**
8	byte
16	short
32	int
64	long

A variable’s type does not directly affect its storage allocation. Type only determines a variable’s arithmetic properties and legal range of values. If a value is assigned to a variable that is outside the legal range of the variable, the value is reduced modulo the range.

3.2 Floating Point Types

The **float** keyword denotes single precision (32 bit); **double** denotes double precision (64 bit). The result of a binary operator on two **float** operands is a **float**. If either operand is a **double**, the result is a **double**.

Floating point arithmetic and data formats are defined by IEEE 754. See “Appendix: Floating Point” on page 41 for details on the Oak language’s floating point implementation.

3.3 Boolean Types

The **boolean** type is used for variables that can be either **true** or **false**, and for methods that return **true** and **false** values. It's also the type that is returned by the relational operators (e.g., >=).

Boolean values are not numbers and cannot be converted into numbers by casting.

3.4 Character Types

The Oak language uses the Unicode character set throughout. Consequently the **char** data type is defined as a 16-bit unsigned integer.

3.5 Arrays

Arrays in Oak are first class objects. They replace pointer arithmetic. All objects (including arrays) are referred to by pointers that cannot be damaged by being manipulated as numbers. Arrays are created using the **new** operator:

```
char s[] = new char[30];
```

The first element of an array is at index 0 (zero). Specifying dimensions in the declarations is not allowed. Every allocation of an array must be explicit—use **new** every time:

```
int i[] = new int[3];
```

Oak does not support multi-dimensional arrays. Instead, programmers can create arrays of arrays:

```
int i[][] = new int[3][4];
```

At least one dimension must be specified but other dimensions can be explicitly allocated by a program at a later time. For example:

```
int i[][] = new int[3][];
```

is a legal declaration.

Subscripts are checked to make sure they're valid:

```
int a[] = new int[10];
a[5] = 1;
a[1] = a[0] + a[2];
a[-1] = 4;    // Raises an exception at runtime
a[10] = 2;    // Raises an exception at runtime
```

Array dimensions must be integer expressions:

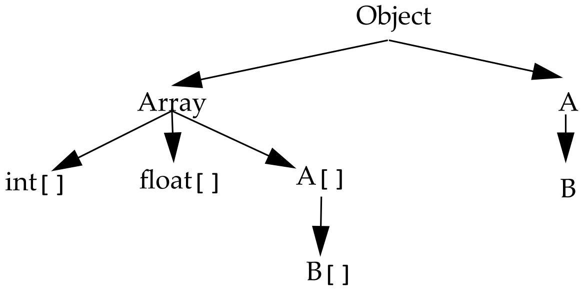
```
int n;
...
float arr[] = new float[n + 1];
```

The length of any array can be found by using **.length**:

```
int a[][] = new int[10][3];
println(a.length);      // prints 10
println(a[0].length);   // prints 3
```

3.5.1 *Array Detail*

Arrays are instances of subclasses of class Object. In the class hierarchy there is a class named Array, which has one instance variable, "length". For each primitive type there is a corresponding subclass of Array. Similarly, for all classes a corresponding subclass of Array implicitly exists. For example, "new Thread`[n]`" creates an instance of Thread`[]`. If class A is a superclass of class B (i.e., B extends A) then A`[]` is a superclass of B`[]` (see the diagram below).



Hence, you can assign an array to an Object:

```
Object o;
int a[] = new int[10];
o = a;
```

and you can cast an Object to an array:

```
a = (int[])o;
```

Array classes cannot be explicitly subclassed.

4 *Classes*

Oak classes represent the classical object oriented programming model. They support data abstraction and implementations tied to data. In Oak, each new class creates a new type.

To make a new class, the programmer must base it on an existing class. The new class is said to be *derived* from the existing class. The derived class is also called a *subclass* of the other, which is known as a *superclass*. Class derivation is transitive: if B is a subclass of A, and C is a subclass of B, then C is a subclass of A.

The immediate superclass of a class and the interfaces (see “Interfaces” on page 31) that the class implements (if any) are indicated in the class declaration by the keywords **extends** and **implements**, respectively:

[Doc comment] [Modifiers] **class** *Classname*
    *[***extends** *Superclassname]*
    *[***implements** *Interface{, Interface}] {*
        ClassBody
}

For example:

```
/** 2 dimensional point */
public class Point {
   float x, y;
   ...
}

/** Printable point */
class PrintablePoint extends Points implements Printable {
   ...
   public void print() {
       ...
   }
}
```

All Oak classes are derived from a single root class: Object. Every class except Object has exactly one immediate superclass. If a class is declared without specifying an immediate superclass, Object is assumed. For example, the following

```
class Point {
    float x, y;
}
```

is the same as

```
class Point extends Object {
    float x, y;
}
```

The Oak language supports only single inheritance. Through a feature known as *interfaces*, it supports some features that in other languages are supported through multiple inheritance (see “Interfaces” on page 31).

4.1 Casting Between Class Types

Oak supports casting between types and because each Oak class is a new type, Oak supports casting between class types. If B is a subclass of A, then an instance of B can be used as an instance of A. No explicit cast is required but an explicit cast is legal—this is called *widening*. If an instance of A needs to be used as if it were an instance of B, the programmer can write a type conversion or *cast*—this is called *narrowing*. Casts from a class to a subclass are always checked at runtime to make sure that the object is actually an instance of the subclass (or one of its subclasses). Casting between sibling classes is a compile-time error. The syntax of a class cast is:

```
(classname)ref
```

where `(classname)` is the object being cast to and `ref` is the object being cast.

Casting affects only the reference to the object, not the object itself. However, access to instance variables is affected by the type of the object reference. Casting an object from one type to another may result in a different instance variable being accessed even though the same variable name is used.

```
class ClassA {
   String name= "ClassA";
}

class ClassB extends ClassA {// ClassB is a subclass of ClassA
   String name= “ClassB”;
}
```

```
class AccessTest {
   void test() {
      ClassB b = new ClassB();
      println(b.name);       // print: ClassB

      ClassA a;
      a = (ClassA)b;
      println(a.name);       // print: ClassA
   }
}
```

4.2 Methods

Methods are the operations that can be performed on an object or class. They can be declared in either classes or interfaces, but they can be implemented only in classes. (All user-defined operations in Oak are implemented with methods.)

A method declaration in a class has the following form (native and abstract methods have no method body):

[Doc comment] [Modifiers] returnType methodName (parameterList) {
[methodBody]
}

Methods:

- Have a return type unless they're constructors, in which case they have no return type. If a non-constructor method does not return any value, it must have a **void** return type.
- Have a parameter list consisting of comma-separated pairs of types and parameter names. The parameter list should be empty if the method has no parameters.

Variables declared in methods (*local variables*) can't hide other local variables or parameters in the same method. For example, if a method is implemented with a parameter named `i`, it's a compile-time error for the method to declare a local variable named `i`. In the following example:

```
class Rectangle {
   void vertex(int i, int j) {
      for (int i = 0; i <= 100; i++) {// ERROR
         ...
      }
   }
}
```

the declaration of "i" in the for loop of the method body of "vertex" is a compile-time error.

The Oak language allows *polymorphic* method naming—declaring a method with a name that has already been used in the class or its superclass—for overriding and overloading methods. *Overriding* means providing a different implementation of an inherited method. *Overloading* means declaring a method that has the same name as another method, but a different parameter list.

Note: Return types are not used to distinguish methods. Within a class scope, methods that have the same name and parameter list *must* return the same type.

4.2.1 *Instance Variables*

All variables in a class declared outside the scope of a method and not marked static (see “Static Variables and Static Methods” on page 26) are instance variables. (Variables declared inside the scope of a method are considered local variables.) Instance variables can have modifiers (see “Modifiers” on page 29).

Instance variables can be of any type and can have initializers. (If an instance variable does not have an initializer, it is initialized to zero, **boolean** variables are initialized to **false** and objects are initialized to **null**. An example of an initializer for an instance variable named **j** is:

```
class A {
    int j = 23;
    ...
}
```

4.2.2 *The this and super Variables*

Inside the scope of an instance of a class, the name **this** represents the current object. For example, an object may need to pass itself as an argument to another object’s method:

```
class MyClass {
    void aMethod(OtherClass obj) {
        ...
        obj.Method(this);
        ...
    }
}
```

Any time a method refers to its own instance variables or methods an implicit “`this.`” is in front of each reference:

```
class Foo {
    int a, b, c;
    ...
    void myPrint(){
        print(a + "\n");     // a == "this.a"
    }
    ...
}
```

The **super** variable is very similar to the **this** variable. The **this** variable contains a reference to the current object; its type is the class containing the currently executing method. The **super** variable contains a reference which has the type of the superclass.

4.2.3 *Setting Local Variables*

Methods are rigorously checked to be sure that all *local variables* (variables declared inside a method) are set before they are referenced. Using a local variable before it is initialized is a compile-time error.

4.3 **Overriding Methods**

To override a method, a subclass of the class that originally declared the method must declare a method with the same name, return type (or a subclass) and parameter list. When the method is invoked on an instance of the subclass, the new method is called rather than the original method. The overridden method can be invoked using the **super** variable such that:

```
setThermostat(...)         // refers to the overriding method
super.setThermostat(...)   // refers to the overridden method
```

4.4 Overload Resolution

Overloaded methods have the same name as an existing method but differ in the number and/or the types of arguments. Overload resolution involves determining which overloaded method to invoke. The return type is not considered when resolving overloaded methods. Methods may be overloaded within the same class. The order of method declaration within a class is not significant.

Methods may be overloaded by varying both the number and the type of arguments. Oak attempts to determine which matching method has the lowest type conversion cost. Only methods with the same name and number of arguments are considered for matching. The cost of matching a method is the maximum cost of converting any one of its arguments. There are two types of arguments to consider, object types and base types.

The cost of converting among object types is the number of links in the class tree between the actual parameter's class and the prototype parameter's class. Only widening conversions are considered. (See "Casting Between Class Types" on page 20 for more information on object conversion.) No conversion is necessary for argument types that match exactly, making their cost 0.

The cost of converting base types is calculated from the table below. Exact matches cost 0.

					To			
		byte	short	char	int	long	float	double
From	byte	0	1	2	3	4	6	7
	short	10	0	10	1	2	4	5
	char	11	10	0	1	2	4	5
	int	12	11	11	0	1	5	4
	long	12	11	11	10	0	6	5
	float	15	14	13	12	11	0	1
	double	16	15	14	13	12	10	0

Note: Cost >= 10 causes data loss.

Once a conversion cost is assigned to each matching method, Oak chooses the method which has the lowest conversion cost. If there is more than one potential method with the same lowest cost the match is ambiguous and a compile-time error occurs.

For example,

```
class A {
   int method(Object o, Thread t);
   int method(Thread t, Object o);

   void g(Object o, Thread t) {
      method(o, t);    // calls the first method.
      method(t, o);    // calls the second method.
      method(o, o);    // ambiguous - compile-time error
   }
}
```

Note: The names of parameters are not significant. Only the number, type, and order are.

4.5 Constructors

Constructors are special methods provided for initialization. They are distinguished by having the same name as their class and by not having any return type. Constructors are automatically called upon the creation of an object. They cannot be called explicitly through an object.

Constructors can be overloaded by varying the number and types of parameters, just as any other method can be overloaded.

```
class Foo {
   int x;
   float y;
   Foo() {
      x = 0;
      y = 0.0;
   }
   Foo(int a) {
      x = a;
      y = 0.0;
   }
   Foo(float a) {
      x = 0;
      y = a;
   }
   Foo(int a, float b) {
      x = a;
      y = b;
   }
   static void myFoo() {
      Foo obj1 = new Foo();      //calls Foo();
      Foo obj2 = new Foo(4);     //calls Foo(int a);
      Foo obj3 = new Foo(4.0);   //calls Foo(float a);
      Foo obj4 = new Foo(4, 4.0);//calls Foo(int a, float b);
   }

}
```

The instance variables of superclasses are initialized by calling either a constructor for the immediate superclass or a constructor for the current class. If neither is specified in the code, the superclass constructor that has no parameters is invoked. If a constructor calls another constructor in this class or a constructor in the immediate super class, that call must be the first thing in the constructor body. The variables **this** and **super** are not defined until the constructor is called. Thus, it is illegal to make any implicit or explicit references to **this** or **super** in the constructor’s parameter list.

Invoking a constructor of the immediate superclass is done as follows:

```
class MyClass extends OtherClass {
    MyClass(someParameters) {
        /* Call immediate superclass constructor */
        super(otherParameters);
        ...
    }
    ...
}
```

Invoking a constructor in the current class is done as follows:

```
class MyClass extends OtherClass {
    MyClass(someParameters) {
        ...
    }
    MyClass(otherParameters) {
        /* Call the constructor in this class that has the
            specified parameter list. */
        this(someParameters);
        ...
    }
    ...
}
```

The Foo and FooSub methods below are examples of constructors.

```
class Foo extends Bar {
    int a;
    Foo(int a) {
        // implicit call to Bar()
        this.a = a;
    }
    Foo() {
        this(42);       // calls Foo(42) instead of Bar()
    }
}

class FooSub extends Foo {
    int b;
    FooSub(int b) {
        super(13);      // calls Foo(13); without this line,
                        // would have called Foo()
        this.b = b;
    }
}
```

If a class declares no constructors, the compiler automatically generates one of the following form:

```
class MyClass extends OtherClass {
    MyClass() {         // automatically generated
        super();
    }
}
```

4.6 Object Creation—the new Operator

A class is a template used to define the state and behavior of an object. An *object* is an *instance* of a class. All instances of Oak classes are allocated in a garbage collected heap. Declaring a reference to an object in Oak does not allocate any storage for that object. The programmer must explicitly allocate the storage for objects, but no explicit deallocation is required; the garbage collector automatically reclaims the memory when it is no longer needed.

To allocate storage for an object in Oak, use the **new** operator. In addition to allocating storage, **new**, initializes the instance variables and then calls the instance's constructor. The constructor is a method that initializes an object (see "Constructors" on page 24). The following syntax allocates and initializes a new instance of a class named ClassA:

```
a = new ClassA();
```

4.6.1 *Garbage Collection*

The Oak garbage collector makes most aspects of storage management simple and robust. Oak programs never need to explicitly free storage: it is done for them automatically. The garbage collector never frees pieces of memory that are still referenced, and it always frees pieces that are not. This makes both dangling pointer bugs and storage leaks impossible. It also frees designers from having to figure out which parts of a system have to be responsible for managing storage.

4.6.2 *The* **null** *Reference*

The keyword **null** is a predefined constant that represents "no instance." **null** can be used anywhere an instance is expected and can be cast to any class type.

4.7 Static Variables and Static Methods

Variables and methods declared in a class can be declared **static**, which makes them apply to the class itself, rather than to an instance of the class. As shown in the following code example, both static variables and static methods are accessed using the class name. For convenience, they can also be accessed using an instance of the class.

```
class Ahem {
    int i;                          // Instance variable
    static int j;                   // Static variable
    void seti(int i) {              // Instance method
        this.i = i;
    }
    static void setj(int j) {       // Static method
        this.j = j;
    }
    static void clearThroat() {
        Ahem a = new Ahem();
        Ahem.j = 2;      // valid; class var via class
        a.j = 3;         // valid; class var via instance
        Ahem.setj(2);    // valid; static method via class
        a.setj(3);       // valid; static method via instance
        a.i = 4;         // valid; instance var via instance
        Ahem.i = 5;      // ERROR; instance var via class
        a.seti(4);       // valid; instance method via instance
        Ahem.seti(5);    // ERROR; instance method via class
    }
}
```

A static variable exists only once per class, no matter how many instances of the class exist.

Static variables can have initializers, just as instance variables can. These initializers are executed just before the first runtime use of the class, before any instances are created. You can also add a code fragment to be executed at the same time the static variables are initialized, as shown in the following example.

```
class A {
   static int arr[] = new int[12];
   static {        // code fragment: initialize the array
      for (int i = 0; i < arr.length; i++) {
         arr[i] = i;
      }
   }
}
```

4.7.1 *Order of Declarations*

The order of declaration of classes and the methods and instance variables within them is irrelevant. However, it is possible for cycles to exist during initialization. For information on cycles during initialization see “Order of Initialization” on page 27. Methods are free to make forward references to other methods and instance variables. The following is legal:

```
class A {
   void a() {
      f.set(42);
   }
   B f;
}
class B {
    void set(long n) {
      this.n = n; }
   long n;
}
```

4.7.2 *Order of Initialization*

When a class is loaded, its static initializer is executed. If this static initializer depends on some other unloaded class, that class is loaded and its static initializer is executed first.

For example, if ClassA is loaded, its static initializer is executed. However, ClassA’s static initializer can have a reference to another unloaded class, for example, ClassB. In that case, ClassB is loaded and its static initializers are executed before ClassA’s. Then, ClassA’s static initializers are executed. A cycle is created if ClassB has a reference to ClassA in its static initializer. A cycle causes an exception to be thrown.

Static variable initializations are executed in lexical order in the static constructor, then instance variable initializations are executed in lexical order in the regular constructor.

It is an compile-time error for regular or static variable initializations to have a forward dependency. For example the following code:

```
int i = j + 2;
int j = 4;
```

results in a compile-time error.

An instance variable’s initialization can have an apparent forward dependency on a static variable. For example in the following code fragment:

```
int i = j + 2;      // Instance variable
static int j = 4;   // Static variable
```

it appears that `i` has a forward dependency on `j`. However, `i` is initialized to 6 and `j` is initialized to 4.

This initialization occurs because `j` is a static variable and is initialized in the static initializer. Static initializers are executed before any other initializers get executed. Thus, `j` is initialized to 4 before `i` is initialized.

Static methods cannot refer to instance variables; they can only use static variables and static methods.

4.8 Access Specifiers

Access specifiers are modifiers that allow programmers to control access to methods and variables. The keywords used to control access are **public** and **private**. Methods marked as **private** can be accessed only from within the class in which they are declared. Since private methods are not visible outside the class, they are effectively **final** and cannot be overridden (see "Final Classes, Methods and Variables" on page 29 for more information). Moreover, you cannot override a non-private method and give it private access.

Public access can be applied to classes, methods, and variables. Classes, methods, and variables marked as **public** can be accessed from anywhere by any other class or method. The access of a public method cannot be changed by overriding it.

Classes, methods, and variables that do not have either private or public access specified can be accessed only from within the package where they are declared (see "Packages" on page 32).

4.9 Variable Scoping Rules

Within a package, when a class is defined as a subclass of another, declarations made in the superclass are visible in the subclass. When a variable is referenced inside a method definition Oak uses the following scoping rules:

- The current block is searched first, and then all enclosing blocks, up to and including the current method. This is considered the local scope.

After the local scope, the search continues in the class scope:

- The variables of the current class are searched.
- If the variable is not found, variables of all superclasses are searched, starting with the immediate superclass, and continuing up through class Object until the variable is found. If the variable is not found, imported classes and package names are searched. If it is not found, it is a compile-time error.

Multiple variables with the same name within the same class are not allowed and result in a compile-time error.

4.10 Modifiers

4.10.1 *Threadsafe Variables*

An instance or static variable can be marked **threadsafe** to indicate that the variable will never be changed by some other thread while one thread is using it, i.e., the variable never changes asynchronously. The purpose of marking a variable as threadsafe is to allow the compiler to perform some optimizations that may mask the occurrence of asynchronous changes. The primary optimization enabled by the use of **threadsafe** is the caching of instance variables in registers.

4.10.2 *Transient Variables*

Transient is a flag available to the interpreter and is intended to be used for persistent objects. Variables marked **transient** are treated specially when instances of the class are written out as persistent objects.

4.10.3 *Final Classes, Methods and Variables*

The **final** keyword is a modifier that marks a class as never having subclasses, a method as never being overridden, or a variable as having a constant value. It is a compile-time error to override a final method, subclass a final class, or change the value of a final variable. Variables marked as **final** behave like constants.

Using **final** lets the compiler perform a variety of optimizations. One such optimization is inline expansion of method bodies, which may be done for small, final methods (where the meaning of *small* is implementation dependent).

Examples of the various final declarations are:

```
class Foo {
    final int value = 3;                // final variable
    final int foo(int a, int b) {       // final method
        ...
    }
}
```

4.10.4 *Native Methods*

Methods marked as native are implemented in a platform-dependent language, e.g., C, not Oak. Native methods do not have a method body, instead the declaration is terminated with a semi-colon. Constructors cannot be marked as native. Though implemented in a platform-dependent language, native methods behave exactly as non-native methods do, for example, it is possible to override them. An example of a native method declaration is:

```
native long timeOfDay();
```

4.10.5 *Abstract Methods*

Methods marked as abstract must be defined in a subclass of the class in which they are declared. An abstract method does not have a method body, instead the declaration is terminated with a semi-colon.

The following rules apply to the use of the **abstract** keyword:

- Constructors cannot be marked as abstract.
- Static methods cannot be abstract.
- Private methods cannot be abstract.
- Abstract methods must be defined in some subclass of the class in which they are declared.
- A method that overrides a superclass method cannot be abstract.
- A class that contains abstract methods and a class that inherits abstract methods without overriding them are considered abstract classes.
- It is an compile-time error to instantiate an abstract class or attempt to call an abstract method directly.

4.10.6 *Synchronized Methods and Blocks*

The **synchronized** keyword is a modifier that marks a method or block of code as being required to acquire a lock. The lock is necessary so that the synchronized code does not run at the same time as other code that needs access to the same resource. Each object has exactly one lock associated with it; each class also has exactly one lock. Synchronized methods are reentrant.

When a synchronized method is invoked, it waits until it can acquire the lock for the current instance (or class, if it's a static method). After acquiring the lock, it executes its code and then releases the lock.

Synchronized blocks of code behave similarly to synchronized methods. The difference is that instead of using the lock for the current instance or class, they use the lock associated with the object or class specified in the block's **synchronized** statement.

Synchronized blocks are declared as follows:

```
/* ...preceding code in the method... */
synchronized(<object or class name>) {   //sync. block
    /* code that requires synchronized access */
}
/* ...remaining code in the method... */
```

An example of the declaration of a synchronized method is:

```
class Point {
    float x, y;
    synchronized void scale(float f) {
        x *= f;
        y *= f;
    }
}
```

An example of a synchronized block is:

```
class Rectangle {
    Point topLeft;
    ...
    void print() {
        synchronized (topLeft) {
            println("topLeft.x = " + topLeft.x);
            println("topLeft.y = " + topLeft.y);
        }
        ...
    }
}
```

5 *Interfaces*

An interface specifies a collection of methods without implementing their bodies. Interfaces provide encapsulation of method protocols without restricting the implementation to one inheritance tree. When a class implements an interface, it generally must implement the bodies of all the methods described in the interface. (If the implementing class is abstract—never implemented—it can leave the implementation of some or all of the interface methods to its subclasses.)

Interfaces solve some of the same problems that multiple inheritance does without as much overhead at runtime. However, because interfaces involve dynamic method binding, there is a small performance penalty to using them.

Using interfaces allows several classes to share a programming interface without having to be fully aware of each other's implementation. The following example shows an interface declaration (with the **interface** keyword) and a class that implements the interface.

```
public interface Storing {
    void freezeDry(Stream s);
    void reconstitute(Stream s);
}
public class Image implements Storing, Painting {
    ...
    void freezeDry(Stream s) {
        // JPEG compress image before storing
        ...
    }
    void reconstitute (Stream s) {
        // JPEG decompress image before reading
        ...
    }
}
```

Like classes, interfaces are either private (the default) or public. The scope of public and private interfaces is the same as that of public and private classes, respectively. Methods in an interface are always **public**. Variables are **public**, **static** and **final**.

5.1 Interfaces as Types

The declaration syntax *interfaceName variableName* declares a variable or parameter to be an instance of some class that implements *interfaceName*. Interfaces behave exactly as classes when used as a type. This lets the programmer specify that an object must implement a given interface, without having to know the exact type or inheritance of that object. Using interfaces makes it unnecessary to force related classes to share a common abstract superclass or to add methods to Object.

The following pseudocode illustrates the *interfaceName variableName* syntax.

```
class StorageManager {
    Stream stream;
    ...
    // Storing is the interface name
    void pickle(Storing obj) {
        obj.freezeDry(stream);
    }
}
```

5.2 Methods in Interfaces

Methods in interfaces are declared as follows:

returnType methodName (parameterList)`;`

The declaration contains no modifiers. All methods specified in an interface are public and abstract and no other modifiers may be applied.

See “Abstract Methods” on page 29 for more information on abstract methods.

5.3 Variables in Interfaces

Variables declared in interfaces are **final**, **public** and **static**. No modifiers can be applied. Variables in interfaces must be initialized.

5.4 Combining Interfaces

Interfaces can incorporate one or more other interfaces, using the **extends** keyword as follows:

```
interface DoesItAll extends Storing, Painting {
   void doesSomethingElse();
}
```

6 *Packages*

Packages are groups of classes and interfaces. They are a tool for managing a large namespace and avoiding conflicts. Every class and interface name is contained in some package. By convention, package names consist of period-separated words, with the first name representing the organization that developed the package.

6.1 Specifying a Compilation Unit’s Package

The package that a compilation unit is in is specified by a **package** statement. When this statement is present, it must be the first non-comment, non-white space line in the compilation unit. It has the following format:

```
package packageName;
```

When a compilation unit has no **package** statement, the unit is placed in a default package, which has no name.

6.2 Using Classes and Interfaces from Other Packages

The Oak language provides a mechanism for making the definitions and implementations of classes and interfaces available across packages. The **import** keyword is used to mark classes as being imported into the current package. A compilation unit automatically imports every class and interface in its own package.

Code in one package can specify classes or interfaces from another package in one of two ways:

- By prefacing each reference to the class or interface name with the name of its package:

  ```
  // prefacing with a package
  acme.project.FooBar obj = new acme.project.FooBar();
  ```

- By importing the class or interface or the package that contains it, using an **import** statement. Importing a class or interface makes the name of the class or interface available in the current namespace. Importing a package makes the names of all of its public classes and interfaces available. The construct:

  ```
  // import all classes from acme.project
  import acme.project.*;
  ```

  means that every public class from acme.project is imported.

  ```
  // import FooBar from acme.project
  import acme.project.FooBar;
  FooBar obj = new FooBar();
  ```

It is illegal to specify an ambiguous class name and doing so always generates a compile-time error. Class names may be disambiguated through the use of a fully qualified class name, i.e., one that includes the name of the class's package.

7 *Expressions*

Expressions in the Oak language are much like expressions in C.

7.1 Operators

The Oak operators, from highest to lowest precedence, are:

```
. [] ()
++ -- ! ~ instanceof
* / %
+ -
<< >> >>>
< > <= >=
== !=
&
^
|
&&
||
?:
= op=
,
```

7.1.1 *Operators on Integers*

For operators with integer results, if any operand is **long**, the result type is **long**. Otherwise the result type is **int**—never **byte**, **short** or **char**. Thus, if a variable `i` is declared a **short** or a **byte**, `i+1` would be an **int**. When a result outside an operator's range would be produced, the result is reduced modulo the range of the result type.

Table 1. Unary Integer Operators:

Operator	Operation
–	unary negation
~	bitwise complement
++	Increment
––	Decrement

The ++ operator is used to express incrementing directly. Incrementing can also be expressed indirectly using addition and assignment. ++lvalue means *lvalue*+=`1`. ++*lvalue* also means *lvalue*=*lvalue*+`1` (as long as lvalue has no side effects). The -- operator is used to express decrementing. The `++` and `--` operators can be used as both prefix and postfix operators.

Table 2. Binary Integer Operators: integer *op* integer ⇒ integer

Operator	Operation
+	addition
–	subtraction
*	multiplication
/	division
%	modulus
&	bitwise AND
|	bitwise OR
^	bitwise XOR
<<	left shift
>>	sign-propagating right shift
>>>	zero-fill right shift

Integer division rounds toward zero. Division and modulus obey the identity `(a/b)*b + (a%b) == a`. Although it may not be obvious that % could overflow, it does for a zero divisor.

The only exception Oak has for integer arithmetic is the "divide by zero" exception. Underflow generates zero. Overflow leads to wrap-around, i.e., adding 1 to the maximum integer wraps around to the minimum integer.

An *op*= assignment operator corresponds to each of the binary operators in the above table.

The integer relational operators <, >, <=, >=, ==, and != produce **boolean** results.

The operators `abs(x)`, `max(x,y)` and `min(x,y)` work for integers as they do for all numbers.

7.1.2 *Operators on Boolean Values*

Variables or expressions that are **boolean** can be combined to yield other **boolean** values. The unary operator **!** is boolean negation. The binary operators **&**, **|**, and **^** are the logical AND, OR, and XOR operators; they force evaluation of both operands. To avoid evaluation of right-hand operands, you can use the short-cut evaluation operators **&&** and **||**. You can also use == and !=. The assignment operators also work: **&=**, **|=**, **^=**. The ternary conditional operator ?: works much as it does in C.

7.1.3 *Operators on Floating Point Values*

Floating point values can be combined using the usual operators: unary **–**; binary +, –, *, and /; and the assignment operators +=, –=, *=, and /=. The ++ and -- operators also work on floating point values (they add or subtract 1.0). In addition, % and %= work on floating point values, i.e.,

```
a % b
```

is the same as

```
a - ((int)(a / b) * b)
```

Which means that a%b is the floating point equivalent of the remainder after division.

The operators `abs(x)`, `max(x,y)` and `min(x,y)` work for floats as they do for all numbers.

Operators that work on integers but that aren't listed in this section work on floating point values by first converting the floating point values into integers.

Floating point expressions involving only single-precision operands are evaluated using single-precision operations and produce single-precision results. Floating point expressions that involve at least one double-precision operand are evaluated using double-precision operations and produce double-precision results.

Oak has no arithmetic exceptions for floating point arithmetic. Following the IEEE 754 floating point specification, the distinguished values Inf and NaN are used instead. Overflow generates Inf. Underflow generates 0. Divide by zero generates Inf.

The usual relational operators are also available, and produce **boolean** results: >, <, >=, <=, ==, !=. Because of the properties of NaN, floating point values are not fully ordered, so care must be taken in comparison. For instance, if `a<b` is not true, it does not follow that `a>=b`. Likewise, `a!=b` does not imply that `a>b || a<b`. In fact, there may no ordering at all.

Floating point arithmetic and data formats are defined by IEEE 754, "Standard for Floating Point Arithmetic." See "Appendix: Floating Point" on page 41 for details on the Oak language's floating point implementation.

7.1.4 *Operators on Arrays*

```
<expression>[<expression>]
```

Gets the value of an element of an array. Legal ranges for the expression are from 0 to the length of the array minus 1. The range is checked only at runtime.

7.1.5 *Operators on Strings*

Oak strings are implemented as String objects (see “String Literals” on page 16 for more information). The operator + concatenates Strings, automatically converting operands into Strings if necessary. If the operand is an object it can define a method call to String() that returns a String in the class of the object.

```
float a = 1.0;
print("The value of a is " + a + "\n");
String s = "a = " + a;
```

The += operator works on Strings. Note, that the left hand side (`s1` in the following example) is evaluated only once.

```
s1 += a; //s1 = s1 + a; a is converted to String if necessary
```

7.1.6 *Operators on Objects*

The binary operator **instanceof** tests whether the specified object is an instance of the specified class or one of its subclasses. For example,

```
if (thermostat instanceof MeasuringDevice) {
    MeasuringDevice dev = (MeasuringDevice)thermostat;
    ...
}
```

determines whether thermostat is a MeasuringDevice object (an instance of MeasuringDevice or one of its subclasses).

7.2 **Casts and Conversions**

The Oak language and runtime system restrict casts and conversions to help prevent the possibility of corrupting the system. Integers and floating point numbers can be cast back and forth, but integers cannot be cast to arrays or objects. Objects cannot be cast to base types. An instance can be cast to a superclass with no penalty, but casting to a subclass generates a runtime check. If the object being cast to a subclass is not an instance of the subclass (or one of its subclasses), the runtime system throws an exception.

8 *Statements*

8.1 **Declarations**

Declarations can appear anywhere that a statement is allowed. The scope of the declaration ends at the end of the enclosing block.

In addition, declarations are allowed at the head of **for** statements, as shown below:

```
for (int i = 0; i < 10; i++) {
    ...
}
```

Items declared in this way are valid only within the scope of the **for** statement. For example, the preceding code sample is equivalent to the following:

```
{
    int i = 0;
    for (; i < 10; i++) {
       ...
    }
}
```

8.2 Expressions

Expressions are statements:

```
a = 3;
print(23);
foo.bar();
```

8.3 Control Flow

The following is a summary of control flow in Oak:

```
if(boolean) statement
else statement
switch(e1) {
    case e2: statements
    default: statements
}
break [label];
continue [label];
return e1;
for([e1]; [e2]; [e3]) statement
while(boolean) statement
do statement
while(boolean);
label:statement
```

Oak supports labeled loops and labeled breaks, for example:

```
outer:                            // the label
    for (int i = 0; i < 10; i++) {
        for (int j= 0; j< 10; j++) {
            if (...) {
                break outer;
            }
            if (...) {
            }
        }
    }
```

The use of labels in loops and breaks has the following rules:

- Any statement can have a label.
- If a break statement has a label it must be the label of an enclosing loop or switch statement.
- If a continue statement has a label it must be the label of an enclosing loop.

8.4 Exceptions

When an error occurs in an Oak program—for example, when an argument has an invalid value—the code that detects the error can *throw* an exception[1]. By default, exceptions result in the thread terminating after printing an error message. However, programs can have *exception handlers* that *catch* the exception and recover from the error.

Some exceptions are thrown by the Oak runtime system. However, any class can define its own exceptions and cause them to occur using **throw** statements. A **throw** statement consists of the **throw** keyword followed by an object. By convention, the object should be an instance of Exception or one of its subclasses. The **throw** statement causes execution to switch to the appropriate exception handler. When a **throw** statement is executed, any code following it is not executed, and no value is returned by its enclosing method. The following example shows how to create a subclass of Exception and throw an exception.

```
class MyException extends Exception {
}

class MyClass {
    void oops() {
        if (/* no error occurred */) {
            ...
        } else { /* error occurred */
            throw new MyException();
        }
    }
}
```

To define an exception handler, the program must first surround the code that can cause the exception with a **try** statement. After the **try** statement come one or more **catch** statements—one per exception class that the program can handle at that point. In each **catch** statement is exception handling code. For example:

```
try {
    p.a = 10;
} catch (NullPointerException e) {
    println("p was null");
} catch (Exception e) {
    println("other error occurred");
} catch (Object obj) {
    println("Who threw that object?");
}
```

A **catch** statement is like a method definition with exactly one parameter and no return type. The parameter can be either a class or an interface. When an exception occurs, the nested **try/catch** statements are searched for a parameter that matches the exception class. The parameter is said to match the exception if it:

- is the same class as the exception; or
- is a superclass of the exception; or
- if the parameter is an interface, the exception class implements the interface.

[1] Oak exception handling closely follows the proposal in the second edition of *The C++ Programming Language*, by Bjarne Stroustrup.

The first **try/catch** statement that has a parameter that matches the exception has its **catch** statement executed. After the **catch** statement executes, execution resumes after the **try**/**catch** statement. It is not possible for an exception handler to resume execution at the point that the exception occurred. For example, this code fragment:

```
print("now ");
try {
   print("is ");
   throw new MyException();
   print("a ");
} catch(MyException e) {
   print("the ");
}
print("time\n");
```

prints “now is the time”. As this example shows, exceptions don’t have to be used only for error handling, but any other use is likely to result in code that’s hard to understand.

Exception handlers can be nested, allowing exception handling to happen in more than one place. Nested exception handling is often used when the first handler can’t recover completely from the error, yet needs to execute some cleanup code (as shown in the following code example). To pass exception handling up to the next higher handler, use the **throw** keyword using the same object that was caught. Note that the method that rethrows the exception stops executing after the **throw** statement; it never returns.

```
try {
   f.open();
} catch(Exception e) {
   f.close();
   throw e;
}
```

8.4.1 *The* **finally** *Statement*

The following example shows the use of a **finally** statement that is useful for guaranteeing that some code gets executed whether or not an exception occurs. You can use either a **catch** statement or a **finally** statement within a particular **try** block, but not both. For example, the following code example:

```
try {
   // do something
} finally {
   // clean up after it
}
```

is similar to:

```
try {
   // do something
} catch(Object e){
   // clean up after it
   throw e;
}
// clean up after it
```

The **finally** statement is executed even if the **try** block contains a **return**, **break**, **continue**, or **throw** statement. For example, the following code example always results in “finally” being printed, but “after try” is printed only if a != 10 or when an exception occurs in the try block.

```
try {
   if (a == 10) {
      return;
   }
} finally {
   print("finally\n");
}
print("after try\n");
```

9 *Future Directions*

The following issues are still under consideration:

- Destructors
- Unicode sequences
- The interaction between method matching and interfaces

A Appendix: Floating Point

This appendix discusses properties of Oak floating point arithmetic: general precision notes and special values, binary format conversion, ordering. At the end is a section summarizing the differences between Oak arithmetic and the IEEE 754 standard. For more information on the IEEE 754 standard, see "IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std. 754-1985."

Operations involving only single-precision **float** and integer values are performed using at least single-precision arithmetic and produce a single-precision result. Other operations are performed in double precision and produce a double precision result. Oak floating-point arithmetic produces no exceptions.

Underflow is gradual.

A.1 Special Values

There is both a positive zero and a negative zero. The latter can be produced in a number of special circumstances: the total underflow of a * or / of terms of different sign; the addition of -0 to itself or subtraction of positive zero from it; the square root of -0. Converting -0 to a string results in a leading '-'. Apart from this, the two zeros are indistinguishable.

Calculations which would produce a value beyond the range of the arithmetic being used deliver a signed infinite result. An infinity (`Inf`) has a larger magnitude than any value with the same sign. Infinities of the same sign cannot be distinguished. Thus, for instance `(1./0.) + (1./0.) == (1./0.).` Division of a finite value by infinity yields a 0 result.

Calculations which cannot produce any meaningful numeric result deliver a distinguished result called Not A Number (`NaN`). Any operation having a NaN as an operand produces a NaN as the result. NaN is not signed and not ordered (see "Ordering"). Division of infinity by infinity yields NaN, as does subtraction of one infinity from another of the same sign.

A.2 Binary Format Conversion

Converting a floating-point value to an integer format results in a value with the same sign as the argument value and having the largest magnitude less than or equal to that of the argument. In other words, conversion rounds towards zero. Converting infinity or any value beyond the range of the target integer type gives a result having the same sign as the argument and the maximum magnitude of that sign. Converting NaN results in 0.

Converting an integer to a floating format results in the closest possible value in the target format. Ties are broken in favor of the most even value (having 0 as the least-significant bit).

A.3 Ordering

The usual relational operators can be applied to floating-point values. With the exception of `NaN`, all floating values are ordered, with `-Inf` < all finite values < `Inf`.

`-Inf == -Inf, +Inf == +Inf, -0. == 0`. The ordering relations are transitive. Equality and inequality are reflexive.

`NaN` is unordered. Thus the result of any order relation between NaN and any other value is false and produces 0. The one exception is that “`NaN !=` anything” is true.

Note that, because NaN is unordered, Oak’s logical inversion operator, !, does not distribute over floating point relationals as it can over integers.

A.4 Summary of IEEE-754 Differences

Oak arithmetic is a subset of the IEEE-754 standard. Here is a summary of the key differences.

- Nonstop Arithmetic—The Oak system will not throw exceptions, traps, or otherwise signal the IEEE exceptional conditions: invalid operation, division by zero, overflow, underflow, or inexact. Oak has no signaling NaN.
- Rounding—Oak rounds inexact results to the nearest representable value, with ties going to the value with a 0 least-significant bit. This is the IEEE default mode. But, Oak rounds towards zero when converting a floating value to an integer. Oak does not provide the user-selectable rounding modes for floating-point computations: up, down, or towards zero.
- Relational set—Oak has no relational predicates which include the unordered condition, except for !=. However, all cases but one can be constructed by the programmer, using the existing relations and logical inversion. The exception case is ordered but unequal. There is no specific IEEE requirement here.
- Extended formats—Oak does not support any extended formats, except that double will serve as single-extended. Other extended formats are not a requirement of the standard.

INDEX