Oak Language Specification

FirstPerson, Inc.
100 Hamilton Avenue
Palo Alto, CA 94301
U.S.A.

100 Hamilton Avenue, Palo Alto California 94301 U.S.A.

Please
Recycle

Contents

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.

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 19)
- import statements (see "Packages" on page 19)
- class declarations (see "Classes" on page 10)
- interface declarations (see "Interfaces" on page 18)

This restriction is not yet enforced by the compiler, although it's necessary for efficient package importation (which is documented in "Packages" on page 19).

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 10).

When Oak source code is compiled, the result is Oak bytecode. Oak bytecode consists of machine-independent instructions that can be interpreted quickly by the Oak runtime system.

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

1.1 Starting Oak Programs

When an Oak program is executed, the interpreter must determine which method to execute first. It does so by executing the class method `main()` in the class that's specified when the interpreter is invoked. The `main()` method must have the following definition:

In the UNIX implementation, the classname is specified as follows:
`oaki ClassName args`

```
public static void main(String arguments[]) {
    /* startup code goes here */
}
```

At least one class per program must implement the `main()` method.

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, and line feeds are not tokens, but they often are used to separate tokens.

Unicode source files aren't allowed yet because there's no editor/development environment to generate them. Instead, ASCII input is accepted.

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.

See the `oak`(1) man page for information on automatically generating documentation.

2.2 Identifiers

Unicode identifiers aren't implemented yet. Instead, identifiers are ASCII and follow the C rules.

Identifiers must start with a letter, underscore ("_"), or dollar sign ("$"); subsequent characters can also contain digits. For the part of Unicode that overlaps ISO-Latin-1, letters are the characters "A" through "Z", "a" through "z", and all the accented letters. 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.

2.3 Keywords

The **ushort**, **Cstring**, **string**, and **unsynchronized** keywords are obsolete.

The **protect** and **unprotect** keywords are subject to change.

enum isn't implemented yet.

instanceof might become a method instead of a keyword.

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

boolean	Cstring	goto	protected	throw
break	default	if	public	transient
byte	do	import	return	try
case	double	instanceof	short	unprotect
catch	else	int	static	unsynchronized
char	enum	interface	string	ushort
class	final	long	super	void
clone	finally	new	switch	volatile
const	float	private	synchronized	while
continue	for	protect	this	

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 zero (`0`). If an integer literal begins with `0x`, it is interpreted as a hexadecimal integer. If a nonzero literal begins with `0`, it is interpreted as an octal integer. Hexadecimal integers can include digits (0-9) and the letters a-f and A-F. Octal integers can include only the digits 0-7.

Type determination is not implemented yet. Forcing literals to be **long** is not implemented yet.

The type of an integer literal is the narrowest integer type that it fits in (see "Integer Types" on page 8). A literal can be forced to be **long** by appending an `L` or `l` to its value.

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`).

Double precision, NaN, Inf, and the type suffixes are not implemented yet.

As described in "Floating Point Types" on page 9, 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**

Specifying too many significant digits for a single precision literal is an error.

2.4.3 *Boolean Literals*

The **boolean** type has two literal values: `true` and `false`. See "Boolean Types" on page 9 for more information on boolean values.

2.4.4 *Character Literals*

Character literals are currently implemented much like in C. When Unicode support is implemented, escape sequences will change.

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 10).

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).

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 documentation of each operator, see "Operators" on page 22.

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 10 and "Interfaces" on page 18).

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

Currently, all integer types act like 32-bit, 2's complement, signed integers.

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 four integer types have widths of 8, 16, 32, and 64 bits, and are signed unless prefixed by the **unsigned** modifier.

unsigned isn't implemented yet; it might never be.

Width	Name	Comments
8	byte	The Oak **byte** type is what C programmers are used to thinking of as the **char** type. But in the Oak language, characters are 16 bits wide. Having a separate **byte** type removes the confusion in C between the interpretation of **char** as an 8 bit integer and as a character.
16	short	In C, the width of **short** is generally 16 bits, but the C specification says it can be larger. In Oak, **short** is always 16 bits wide.
32	int	An **int** in the Oak language is always 32 bits wide. In C, the width of **int** is implementation defined and is most often 32 bits, but is sometimes 16 bits, and has been other values (such as 60).
64	long	The Oak language's definition of **long** is a break from the C tradition that specifies that **long** is 32 bits and **long long** is 64 bits. With the standardization of **int** to mean 32 bits, it is redundant to have two types with the same meaning and unnecessary to have such an odd type name for 64 bits.

A variable's type does not directly affect its storage allocation. Type only determines the 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.

Value reduction is not implemented yet.

3.2 Floating Point Types

double is not implemented yet.

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 29 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 relational operators such as `>`.

Boolean values are not numbers and can't be converted into numbers by casting.

3.4 Character Types

Unicode is not implemented yet. Characters currently have ASCII values, although they are stored in 16 bits.

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

The Oak language includes support for *arrays*—sets of ordered data items. Arrays are referred to and passed by reference.

Subscripts are checked to make sure they're valid:

```
int a[10];
a[5] = 1;
a[11] = 2;/* ERROR */
```

Array dimensions can be integer expressions:

```
void doIt(int n) {
    float arr[n];
    ...
}
```

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

The **print** operator is one of a group of operators whose functionality might be moved into classes.

```
int a[10][3];
print(a.length + ", " + a[0].length + "\n");

10, 3
```

Arrays are allocated either where they're declared (by specifying the dimensions of the array when it is declared, as shown above) or dynamically with the **new** keyword:

Although arrays can be created with **new**, just as instances of classes are created, arrays are not currently objects.

```
int a[];
a = new int[10];
Raster foo[];
foo = new Raster[10];        //creates an array, but not the
                             //Raster objects in the array.
foo[1] = new Raster("blah.jpg");
```

4 *Classes*

Classes represent the classical object oriented programming model. They support data abstraction and implementations tied to data.

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.

If B is a subclass of A, then an instance of B can be used as an instance of A. In fact, if there is no ambiguity, then no explicit cast is needed. 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*. Casts from a class to a subclass are always checked to make sure that the object is actually an instance of the subclass (or one of its subclasses).

The Oak language supports 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 18). Instances of classes are stored in a garbage collected heap (see "Garbage Collection" on page 28); local variables are references to objects in the heap.

The immediate superclass of a class and the interfaces that the class implements (if any) are indicated in the class declaration by the keywords **extends** and **implements**, respectively:

```
public class Classname extends Superclassname
        implements Interface1, Interface2 {
    /* . . . */
}
```

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

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

is the same as

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

Classes are either private (the default) or public. Private classes are invisible outside of the package in which they're declared. Public classes can be used outside of their package. Public classes can't be derived from private classes. To declare a class public (or private), use the **public** (or **private**) keyword.

4.1 Instance Variables

Instance variables are declared just like local variables. They can be of any type and can have initializers. These initializers are executed when the instance is initialized. (If an instance variable does not have an initializer, it is initialized to zero or, for **boolean** variables, to `false`.) An example of an initializer for an instance variable named **j** follows.

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

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:

```
void aMethod() {
    /* . . . */
     otherObject.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;
    /* . . . */
    print(a + "\n");      // a == "this.a"
    /* . . . */
}
```

Instance variables can't be hidden by being redeclared in subclasses. Specifically, if a class declares a public or protected instance variable, that variable cannot be redeclared in any subclass, although it can be used by any subclass. (See "Access to Variables and Methods" on page 15 for information on declaring variables public, protected, and private.) Private instance variables can be redeclared, since they aren't visible to subclasses.

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. (Note: All user-defined operations in Oak are implemented with methods; Oak has no functions.)

A method declaration in a class has the following form:

[accessSpecifiers] returnType methodName `(` *parameterList* `) {`
    [methodBody]
`}`

A method declaration in an interface has the following form:

[accessSpecifiers] returnType methodName `(` *parameterList* `) = 0;`

Methods:

- Have a return type unless they're constructors, in which case they must 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 parameters. The parameter list should be empty if the method has no parameters.

In the Oak 0.2 release, local variables and parameters can't hide instance variable names. In the future, this restriction will be eased; the compiler will generate a warning instead of an error.

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

4.3 Overriding and Overloading Methods

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.

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.

To overload a method, a class declares a method that has the same name and return type as another method (which has been declared in the class or in one of its superclasses), but a different parameter list. The Oak runtime system resolves which method to call by matching the *actual parameter list* (the parameter list passed to the method) against the *formal parameter lists* of all methods with the same name.

```
class A {
    void Thermostat(Foo f) {}
}
class B extends A {
    void Thermostat(Foo f) {}    // override
    void Thermostat() {}         // overload
    int Thermostat() {}          // ERROR: Duplicate method
}
```

When deciding which method to invoke, the runtime system computes the number of conversions required to change the actual parameter list into the types declared in each method's formal parameter list. The method that requires the fewest conversions is chosen. If there is a tie, the method call is ambiguous and a compilation error occurs.

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

4.4 Used before Set

Methods are rigorously checked to be sure that all *local variables* (variables declared inside a method) are set before they are referenced. Used-before-set is a fatal compilation error.

4.5 Class Variables and 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 class variables and class 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;                       // Class variable
    void seti(int I) { i = I; }         // Instance method
    static void setj(int J) { j = J; } // Class method
};

Ahem a = new Ahem();
Ahem.j = 2;     /* valid; class var via class */
a.j = 3;        /* valid; class var via instance */
Ahem.setj(2);   /* valid; class method via class */
a.setj(3);      /* valid; class 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 class variable exists only once per address space, no matter how many instances of the class exist in that address space. For distributed applications that run in multiple address spaces, each address space has one occurrence of the class variable. When you refer to a class variable relative to some object (for example, `obj.aVar`) the class variable `aVar` is fetched from the address space where `obj` resides. See “Synchronized Methods and Blocks” on page 15 for information on achieving synchronized access to class variables.

Class 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 class variables are initialized, as shown in the following example.

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

Class methods cannot refer to instance variables; they can only use class variables.

4.6 Constants

Instance and class variables can be marked **const** to indicate that, once initialized, their value never changes.

4.7 Volatile Variables

Instance and class variables can be marked **volatile** so that the compiler treats them specially during optimization. The values of volatile variables are never cached in registers and are always re-read when referenced. Variables should be marked **volatile** when they might be changed by means undetectable by the compiler, such as by another thread or device.

4.8 Transient Variables

Variables marked **transient** are treated specially when instances of the class are written out as persistent objects. Specifically, the values of transient variables are not written out. When the persistent object is reconstituted, transient variables are initialized to zero.

4.9 Final Classes and Methods

The **final** keyword is an access specifier that marks a class as never having subclasses, or a method as never being overridden. Using **final** lets the compiler perform a variety of optimizations. One such optimization is inline expansion of method bodies, which is done for small, final methods (where the meaning of *small* is implementation dependent).

4.10 Access to Variables and Methods

Each variable or method declared in a class has one of the following types of access: **public**, **protected**, or **private**. These access types affect whether the variable or method can be used by other classes.

The rules for access to classes and interfaces in the same package might change.

Note: All classes in a particular package can use all variables and methods declared in the classes in that package, regardless of **public**, **protected**, and **private** declarations (see "Packages" on page 19).

By default all variables and methods in a class (including constructors) are **private**. Private variables and methods can be accessed only by methods declared in the class, and not by its subclasses or any other classes (except for classes in the same package). Public variables and methods—those declared with the **public** type modifier—can be accessed by anyone. The **protected** type modifier makes a variable or method accessible to subclasses, but not to any other classes (except those in the same package).

The following example shows how to specify access.

```
class Stuff {
    int i;                    /* private by default */
    public int j;             /* visible to everyone */
    protected int k;          /* subclasses see this */
    void method1() { }        /* private by default */
    public void method2() { } /* public */
    protected static void method3() { } /* protected */
};
```

4.11 Synchronized Methods and Blocks

The **synchronized** keyword is an access specifier that marks a method or block of code as being required to acquire a lock, so that it does not run at the same time as other code that needs access to the same resource. (The other code must also be marked **synchronized**.) Each object has exactly one lock associated with it; each class also has exactly one lock.

Synchronized methods are declared as follows:

```
synchronized [other access specifiers]* <return type>
        <method name> (<parameter list>) {
    /* implementation */
}
```

Currently, you must use **synchronize** (no "d") instead of **synchronized** when declaring synchronized blocks. Synchronized blocks based on classes (as opposed to objects) currently don't work.

Synchronized blocks are declared as follows:

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

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

Synchronized blocks of code behave similarly, except 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 **synchronize** statement.

For more information on blocks of code running simultaneously, see the Thread class documentation in the *FirstPerson Programming Interface*.

4.12 Constructors

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

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; }
}

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);
```

Before the constructor is called, storage for an instance is atomically allocated and initialized to be a copy of the prototype for the class.

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. Calling a constructor must be the first thing in the method body; calling a constructor later is illegal.

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

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

Invoking a constructor in the current class is done as shown in the following code example.

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

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

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

```
ClassName() {
    super();
}
```

4.13 Order of Declarations

The order of declaration of classes and the methods and instance variables within them is irrelevant. Methods are free to make forward references to other methods and instance variables. The following works:

```
class A {
    void a() { f.set(42); }
    B f;
}

class B {
    void set(long n) { N = n; }
    long N;
}
```

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. (The exception is that if the implementing class is *abstract*—never instantiated—it can leave the implementation of some or all 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 often 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.

In the future, the "=0" part of declaring methods in interfaces may go away.

```
public interface Storing {
    void freezeDry(Stream s) = 0;
    void reconstitute(Stream s) = 0;
}

public class Raster 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. As for classes, the **public** and **private** keywords specify whether an interface is public or private.

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*. 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 just to guarantee that many classes implement the same methods.

```
class StorageManager {
    Stream stream;
    ...
    void pickle(Storing obj) {
        obj.freezeDry(stream);
    }
}
```

5.2 Constants in Interfaces

Besides methods, interfaces can also declare constants. The value of the constant must be set in the interface. For example:

```
interface InterfaceName {
    const int aConstant = 42;
    . . .
}
```

Code can refer to the interface constants as if they were declared as class (**static**) variables in the implementing class. When Oak detects a class implementing two or more interfaces that declare constants with the same name, an error results. One way to avoid the possibility of this error is to specify the interface name before every use of the constant, whether in the implementing class or in clients:

```
InterfaceName.aConstant
o.aConstant        // where o is declared as "InterfaceName o;"
```

5.3 Combining Interfaces

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

```
interface DoesItAll extends Storing, Painting {
    void doesSomethingElse() = 0;
}
```

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. (See “Classes” on page 10 and “Access to Variables and Methods” on page 15 for information on how packages affect the namespace.) By convention, package names consist of period-separated words, with the first name representing the organization that developed the package.

The oak.lang package contains classes and interfaces that are integral to the Oak language. All Oak programs automatically import the contents of the oak.lang package. (Importing is discussed in “Using Classes and Interfaces from Other

Packages" on page 20.) The oak.lang package is documented in the *FirstPerson Programming Interface* in the chapter "The Language (oak.lang) Package."

Implementation Note: In the current implementation of Oak, packages are closely tied to the file system. Everything that's in a particular package is in one directory, and only one package can be in a particular directory. The name of a package indicates its directory. For example, a package named `oak.lang` would be in a directory `lang` under a directory named `oak` . The `oak/lang` directory could be anywhere in the file system. The Oak runtime system looks for it in the directories specified by the CLASSPATH environment variable.

6.1 Specifying a Compilation Unit's Package

The package that a compilation unit is in is specified by a **package** statement. 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.

A compilation unit automatically imports every class and interface in its own package.

6.2 Using Classes and Interfaces from Other Packages

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
myCo.aGroup.AClass o=new myCo.aGroup.AClass();
```

- 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.

```
// importing just a class
import myCo.aGroup.AClass;
AClass o=new AClass();

// importing an entire package
import myCo.aGroup.*;
AClass o=new AClass();
```

The syntax for importing packages is likely to change.

7 *Assertions*

Assertions aren't implemented yet.

The Oak language has a set of facilities that allow assertions to be made about the behavior of programs. These allow extensive checking and a corresponding increase in the reliability of programs. A failed assertion results in an AssertionFailedException (see "Exceptions" on page 26).

7.1 Constraints on Instance Variables and Methods

The **assert** keyword can be used to declare a set of constraints on instance variables and methods. This enables concise documentation of a class designer's intentions. The annotations also serve as a binding contract between a class designer and a class maintainer.

While objects are not required to obey the legality constraints within methods, the constraints are enforced at the entry and exit of every public and protected method. In classes that use **assert** for all variables, all public and protected methods can expect to operate on a coherent object and have the responsibility of restoring coherence before finishing. The following example shows how to use **assert** to constrain the values of two instance variables.

```
class Calender {
    static int lastDay[12]=
            {31,29,31,30,31,30,31,31,30,31,30,31};
    int month assert(month >=1 && month <=12);
    int date assert(date>=1 && date<=lastDay[month]);
}
```

7.2 Preconditions and Postconditions

The syntax described in this section might change when preconditions and postconditions are implemented.

The behavior of a method can be specified by a set of preconditions that must hold before the method begins and a set of postconditions that must hold after it finishes.

```
class Stack {
    int length;
    Element element[];
    boolean full() {        /* . . . */ };
    boolean empty() { return length==0; }

    Element pop() {
        precondition: !empty();
        /* . . . */
        postcondition: !full();
    }

    void push(Element x) {
        precondition: !full();
        /* . . . */
        postcondition: !empty();
    }
}
```

Preconditions and postconditions are inherited by subclasses: methods overridden by a subclass must obey the preconditions and postconditions of their superclass. Inherited preconditions and postconditions cannot be restricted or redefined.

8 *Expressions*

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

8.1 Operators

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

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

8.1.1 *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** or **short**. 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: *op* **integer ⟹ integer**

Operator	Operation
–	unary negation
~	bitwise complement

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.

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

The integer relational operators **<**, **>**, **<=**, **>=**, **==**, and **!=** produce **boolean** results. They cannot overflow.

8.1.2 *Operators on Boolean Values*

The &, |, and ^ operators aren't implemented yet for **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 conditional operator ?: works much as it does in C.

8.1.3 *Operators on Floating Point Values*

Double precision and special mathematical values are not implemented yet.

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. 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. Floating point operations cannot cause exceptions, but they can produce the special values of Infinity or Not-a-Number.

The usual relational operators are also available, and produce **boolean** results: `>`, `<`, `>=`, `<=`, `==`, `!=`. Because of the properties of Not-a-Number, 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 29 for details on the Oak language's floating point implementation.

8.1.4 *Operators on Strings*

The operator + concatenates Strings, automatically converting operands into Strings if necessary.

```
float a = 1.0;
print("The value of a is " + a + "\n");
print("" + 1.01 + 2 + "\n");
print(1.01 + 2 + "\n");    // = (1.01 + 2) + "\n"

The value of a is 1
1.012
3.01
```

The += and ++ operators also work on Strings. They and their rough equivalents are shown below. Note, however, that the left hand side (`s1` in the following examples) is evaluated only once.

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

s1++;    //s1 = s1 + "1"
```

print() and **println()** aren't really operators; they're Oak-defined global methods. Such global methods are expected to be replaced by class-based methods.

The **print()** operator prints the argument specified in its parentheses, coercing it to be a String if necessary. The **println()** operator is the same as **print()**, except that it adds `"\n"` (the newline character) to the end of the specified String.

8.1.5 *Operators on Objects*

The unary operator **clone** is applied to an object. It atomically allocates space for a new instance of the same class and copies the contents of the existing object into it, making the new object an exact, shallow copy of the old one. For example, if the existing object refers to another object, the clone refers to the same object—the referred-to object is not cloned. The **clone** operator is normally used inside **new** to clone the prototype of some class, before applying the initializers (constructors).

instanceof might be replaced by an operator that is more dynamic. It might become a method.

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

```
(thermostat instanceof MeasuringDevice)
```

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

8.2 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. 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 InvalidClassCastException.

9 *Statements*

9.1 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++) . . .
}
```

9.2 Expressions

As in C, expressions are statements:

```
a = 3;
print(23);
```

9.3 Control Flow

Except for the **for** statement, which can contain declarations (as described in “Declarations” on page 25), this is just like C:

```
if(boolean) statement
else statement

switch(e1) {
    case e2: statements
    default: statements
}

break;

goto label;
```

```
continue;

return e1;

for(e1; e2; e3) statement

while(boolean) statement

do statement
while(boolean);
```

9.4 Exceptions[1]

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. By default, exceptions result in the thread terminating with 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 GenericException 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 GenericException and throw an exception.

```
class MyException extends GenericException {};

if (/* no error occurred */)
    /* do something */
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** clauses—one per exception class that the program can handle at that point. In each **catch** clause is exception handling code. For example:

```
try {
    p.a = 10;
} catch (NullPointerException e) {
    print("p was null\n");
} catch (GenericException e) {
    print("other error occurred\n");
}
```

A **catch** clause is like a method definition with exactly one parameter and no return type. When an exception occurs, the runtime system searches the nested **try**/**catch** clauses. The first one with a parameter type that is the same class or a superclass of the thrown object has its **catch** clause executed. After the **catch** clause executes, execution resumes after the **try**/**catch** statement. It is not possible

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

for an exception handler to resume execution at the point that the exception occurred.

For example, this code fragment:

```
class Foo {};

print("now ");
try {
    print("is ");
    throw new Foo();
    print("a ");
} catch(Foo p) {
    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, and 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 without specifying a GenericException instance. Note that the method that rethrows the exception stops executing after the **throw** statement; it never returns.

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

9.4.1 *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;
}
/* clean up after it */
```

The **finally** statement is executed even if the **try** block contains a **goto**, **return**, **break**, **continue**, or **throw** statement. For example, the following code example always results in "finally" being printed, but "after try" is never printed.

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

9.4.2 *Asynchronous Exceptions*

Generally, exceptions are synchronous—they are thrown by code executed sequentially by an Oak program. However, in programs that have multiple threads of execution, one thread can throw an exception (using Thread's `postException()` instance method) to another thread. The second thread can't predict exactly when it will be thrown an exception, so the exception is *asynchronous*.

Implementation Note: As of Oak 0.2, no FirstPerson-supplied code throws asynchronous exceptions, so you don't need to worry about them unless you use them in your own code.

The default will probably be changed to *not* allow asynchronous exceptions except in explicitly *unprotected* sections of code.

By default, asynchronous exceptions can happen at any time. To prevent asynchronous exceptions from occuring in a critical section of code, you can mark the code with the **protect** keyword, as shown below:

```
protect {
    /* critical section goes here */
}
```

To allow asynchronous exceptions to occur in an otherwise protected section of code, use the **unprotect** keyword, as follows:

```
unprotect {
    /* code that can afford asynchronous exceptions  */
}
```

10 *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.

The garbage collector also does compaction: it copies all objects to the beginning of the heap, coalescing free space in one large chunk at the end. This eliminates the loss of free space due to fragmentation.

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" on page 30). 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.

Glossary

abstract class
A class that should never be instantiated; only its subclasses should be instantiated. Abstract classes are defined so that other classes can inherit from them.

actual parameter list
The arguments specified in a particular method call. See also *formal parameter list*.

argument
A data item specified in a method call. An argument can be a literal value, a variable, or an expression.

array
A collection of data items, all of the same type, in which each item's position is uniquely designated by an integer.

ASCII
American Standard Code for Information Interchange. A standard assignment of 7-bit numeric codes to characters. See also *Unicode*.

atomic
Refers to an operation that is never interrupted or left in an incomplete state under any circumstance.

binary operator
An operator that has two arguments.

bit
The smallest unit of information in a computer, with a value of either 0 or 1.

bitwise operator
An operator that manipulates bit-oriented data, such as by performing the logical AND operation such that each bit that's 1 in either operand is 1 in the result.

block
In the Oak language, any code between matching braces ({ and }).

boolean
Refers to an expression or variable that can have only a true or false value. The Oak language provides the **boolean** type and the literal values **true** and **false**.

byte
A sequence of eight bits. The Oak language provides a corresponding **byte** type.

bytecode
Machine-independent code generated by the Oak compiler and executed by the Oak interpreter.

casting
Explicit conversion from one data type to another.

class
In the Oak language, a type that defines the implementation of a particular kind of object. A class definition defines instance and class variables and methods, as well as specifying the interfaces the class implements and the immediate superclass of the class.

class method
Any method that can be invoked using the name of a particular class. Class methods affect the class as a whole, not a particular instance of the class. Class methods are defined in class definitions. See also *instance method*.

class variable
A data item associated with a particular class as a whole—not with particular instances of the class. Class variables are defined in class definitions. See also *instance variable*.

comment
In a program, explanatory text that is ignored by the compiler. In Oak programs, comments are delimited using // or /*...*/.

compilation unit
The smallest unit of Oak code that can be compiled. In the current Oak implementation, the compilation unit is a file.

compiler
A program to translate source code into code to be executed by a computer. The Oak compiler translates Oak source code into Oak bytecode. See also *interpreter*.

constructor
A method that creates an object. In the Oak language, constructors are instance methods with the same name as their class. Oak constructors are invoked using the **new** keyword.

critical section
A segment of code in which a thread uses resources (such as certain instance variables) that can be used by other threads, but that must not be used by them at the same time.

declaration
A statement that establishes an identifier and associates attributes with it, without necessarily reserving its storage (for data) or providing the implementation (for methods). See also *definition*.

definition
A declaration that reserves storage (for data) or provides implementation (for methods).

derived from
Describes a class that inherits properties of another class. See also *subclass*, *superclass*.

distributed
Running in more than one address space.

double precision
In the Oak language specification, describes a floating point number that holds 64 bits of data. See also *single precision*.

encapsulation
The localization of knowledge within a module. Because objects encapsulate data and implementation, the user of an object can view the object as a black box that provides services. Instance variables and methods can be added, deleted, or changed, but as long as the services provided by the object remain the same, code that uses the object can continue to use it without being rewritten.

exception
An event during program execution that prevents the program from continuing normally; generally, an error. The Oak language supports exceptions with the **try**, **catch**, and **throw** keywords. See also *exception handler*.

exception handler
A block of code that reacts to a specific type of exception. If the exception is for an error that the program can recover from, the program can resume executing after the exception handler has executed. See also *exception*.

formal parameter list
The parameters specified in the definition of a particular method. See also *actual parameter list*.

garbage collection
The automatic detection and freeing of memory that is no longer in use. The Oak runtime system performs garbage collection so that programmers never explicitly free objects and other data.

hexadecimal
The numbering system that uses 16 as its base. The marks 0-9 and a-f (or equivalently A-F) represent the digits 0 through 15. In Oak programs, hexadecimal numbers must be preceded with **0x**. See also *octal*.

hierarchy
A classification of relationships in which each item except the top one (known as the *root*) is a specialized form of the item above it. Each item can have one or more items below it in the hierarchy. In the Oak class hierarchy, the root is the Object class.

identifier
The name of an item in an Oak program.

inheritance
The concept of classes automatically containing the variables and methods defined in their superclasses.

instance
An object of a particular class. In Oak programs, an instance of a class is created using the **new** operator followed by the class name.

instance method
Any method that can be invoked using an instance of a class, but not using the class name. Instance methods are defined in class definitions. See also *class method*.

instance variable
Any item of data that's associated with a particular object. Each instance of a class has its own copy of the instance variables defined in the class. See also *class variable*.

interface
In the Oak language, a group of methods that can be implemented by several classes, regardless of where the classes are in the class hierarchy.

interpreter
A module that alternately decodes and executes every statement in some body of code. The Oak interpreter decodes and executes Oak bytecode. See also *compiler, runtime system*.

lexical
Pertaining to how the characters in source code are translated into tokens that the compiler can understand.

linker
A module that builds an executable, complete program from component machine code modules. The Oak linker creates a runnable program from compiled classes. See also *compiler, interpreter, runtime system*.

literal
The basic representation of any integer, floating point, or character value. For example, **3.0** is a single-precision floating point literal, and **'a'** is a character literal.

local variable
A data item known within a block, but inaccessible to code outside the block. For example, any variable defined within an Oak method is a local variable and can't be used outside the method.

method
A function defined in a class. See also *instance method*, *class method*.

multithreaded
Describes a program that is designed to have parts of its code execute concurrently. See also *thread*.

object
The principle building blocks of object-oriented programs. Each object is a programming unit consisting of data (instance variables) and functionality (instance methods). See also *class*.

object oriented design
A software design method that models the characteristics of abstract or real objects using classes and objects.

octal
The numbering system using 8 as its base, using the numerals 0-7 as its digits. In Oak programs, octal numbers must be preceded with **0**. See also *hexadecimal*.

overloading
Using one identifier to refer to multiple items in the same scope. In the Oak language, you can overload methods but not variables or operators.

overriding
Providing a different implementation of a method in a subclass of the class that originally defined the method.

package
In the Oak language, a group of classes. Packages are declared with the **package** keyword.

pixel
The smallest addressable picture element on a display screen or printed page.

pointer
A data element whose value is an address.

process
A virtual address space containing one or more threads.

root
In a hierarchy of items, the one item from which all other items are descended. The root item has nothing above it in the hierarchy. See also *hierarchy*, *class*, *package*.

scope
A characteristic of an identifier that determines where the identifier can be used. Most identifiers in the Oak language have either class or local scope. Instance and class variables and methods have class scope; they can be used outside the class and its subclasses only by prefixing them with an instance of the class or (for class variables and methods) with the class name. All other variables are declared within methods and have local scope; they can be used only within the enclosing block.

single precision
In the Oak language specification, describes a floating point number with 32 bits of data. See also *double precision*.

subclass
A class that is derived from a particular class, perhaps with one or more classes in between. See also *superclass*.

superclass
A class from which a particular class is derived, perhaps with one or more classes in between. See also *subclass*.

thread
The basic unit of program execution. A process can have several threads running concurrently, each performing a different job, such as waiting for events or

performing a time-consuming job that the program doesn't need to complete before going on. When a thread has finished its job, the thread is suspended or destroyed. See also *process*.

Unicode
A 16-bit character set defined by ISO 10646.

variable
An item of data named by an identifier. Each variable has a type, such as **int** or Object, and a scope. See also *class variable*, *instance variable*, *local variable*.

Index