

# Oak

## *Specification*



**first** **person**

**Confidential**



---

## *Contents*

Oak is.....	5
Program Structure .....	7
Lexical Issues.....	8
Comments	8
Identifiers	8
Keywords	9
Literals	9
Integer Literals	9
Floating Point Literals	9
Boolean Literals	9
Character Literals	10
Operators and Miscellaneous Separators	10
Types .....	10
Integer Types	10
Floating Point Types	11
Boolean Types	11
Character Types	12
Arrays	12
Types Created with the <b>typedef</b> Keyword	12
Classes.....	13
Instance Variables	14
Methods	15
Overriding and Overloading Methods	15
Used before Set	16
Static Variables and Methods	16
Volatile Variables	17
Transient Variables	17
Final Classes and Methods	17
Properties of Variables	17

---

Access to Variables and Methods	18
Variables with Public-Key Seals	18
Synchronized Methods	19
Constructors	19
Order of Declarations	20
Interfaces . . . . .	21
Packages . . . . .	22
Assertions . . . . .	22
Constraints on Class Variables	22
Preconditions and Postconditions	23
Expressions . . . . .	23
Operators	23
Operators on Integers	24
Operators on Boolean Values	25
Operators on Floating Point Values	25
Operators on Character Arrays	25
Operators on Objects	25
Casts and Conversions	26
Statements . . . . .	26
Declarations	26
Expressions	26
Control flow	26
Exceptions	27
Garbage Collection . . . . .	29
Appendix: Floating Point . . . . .	31
Special Values	31
Binary Format Conversion	31
Ordering	32
Precision	32
Summary of IEEE-754 Differences	32



# Language Specification

Notes about the current im-<sup>1</sup>  
plementation are off to the  
side.  
Future Oak development en-  
vironments might use a dif-  
ferent compilation unit than  
files.

## Program Structure

---

The source code for an Oak program consists of one or more files with the “.oak” suffix. Each file can contain only the following (in addition to white space and comments):

- import statements
- class definitions (see “Classes” on page 13)
- interface definitions (see “Interfaces” on page 21)
- main code

*Main code* is Oak code that isn’t associated with an object. Generally, main code simply serves to get a program underway, by creating one or more objects and starting their execution. Oak main code is analogous to the C `main()` routine. For example, variables defined in main code are local to the main code; they cannot be used by classes defined in the same file. Exactly one file per program can contain main code.

When an Oak source file is compiled, several files containing Oak bytecode can be created. Specifically, the compiler creates one bytecode file per class, with each file named after its class with an additional “.class” suffix. The compiler also creates one bytecode file for the main code; by default, this file is named after the Oak source file containing the main code, without the “.oak” suffix.

When Oak bytecode is interpreted, the Oak runtime system checks a predetermined class path for each class used in the program. When the runtime system finds the file containing the class bytecode, it loads the class definition. Each class in a program is searched for at most once each time the program runs; subsequent references to the class are immediately resolved. See the `oak(1)` man page for more information on class paths and the Oak compiler and interpreter.

## 2 *Lexical Issues*

---

During compilation, the characters in an Oak source file are reduced to a series of tokens. Oak has 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.

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

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

### 1 **Comments**

Oak has three kinds of comments:

<code>// text</code>	All characters from <code>//</code> to the end of the line are ignored.
<code>/* text */</code>	All characters from <code>/*</code> to <code>*/</code> are ignored.
<code>** text */</code>	Like <code>/*...*/</code> , except that these comments are treated specially when they occur immediately before a declaration. These comments indicate that the enclosed text should be included in automatically generated documentation as a description of the following declaration. See <code>&lt;&lt;wherever the oakc -doc option is documented&gt;&gt;</code> 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 and can contain letters, digits, and underscores (“\_”). Since Unicode is a large character set that is hard to characterize, the definition of a letter is difficult. For the part of Unicode that overlaps ISO-Latin-1, letters are the characters “A” through “Z”, “a” through “z”, and all the accented characters. 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 “Mjølnér” are legal identifiers, but strings containing characters such as “¶” are not.

## 2.3 Keywords

Not every keyword has been completely implemented yet.

**enum** might go away.

**instanceof** will probably become a method instead of a keyword.

**unsigned** isn't implemented yet.

**void** might go away.

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

break	continue	for	private	throw
byte	default	goto	protected	transient
case	do	if	public	try
catch	double	instanceof	return	typedef
char	else	int	short	unsigned
class	enum	interface	static	void
clone	final	long	switch	volatile
const	float	new	synchronized	while

## 2.4 Literals

Literals are the basic representation of any integer, floating point, boolean, or character 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. Hexidecimal 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 10). 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 consists of a decimal integer, a decimal point, a fraction part (another decimal number), an exponent part, and an optional type suffix. The exponent part is an e or E followed by an integer, which can be signed.

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

As described in “Floating Point Types” on page 11, Oak has two floating point types: **float** (32 bits, also known as *single precision*) and **double** (64 bits, known as *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 11 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 12).

An array of characters can be represented by a sequence of characters between double quotes. For example, `"hello world\n"` is a literal array of characters.

Unicode support is not implemented yet. Some or all strings might become objects.

#### **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:

`++ -- == <= >= != += - = *= /= &= |= ^= %= <<= >>= >>>= &= &= &= || && << >> >>>`

## 3 *Types*

---

Every variable and every expression has a type. Type determines the allowable range of values a variable can take on, 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* mechanism (see “Classes” on page 13) and, in a limited way, using **typedef**.

Oak has two kinds of types: simple and composite. Simple types are those that cannot be broken down; they are atomic. The simple types are all integer, floating point, boolean, or character types. Composite types are built on simple types. Oak has three kinds of composite types—arrays, types created with **typedef**, and classes. Simple types, arrays, and **typedef** are discussed in this section. Classes are discussed in “Classes” on page 13.

### 3.1 **Integer Types**

Not implemented yet.

Integers in Oak 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 are signed unless prefixed by the **unsigned** modifier and have widths of 8, 16, 32, and 64 bits.

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

Value reduction is not implemented yet.

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, then the value is reduced modulo the range.

### 3.2 Floating Point Types

**double** 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 **floats** 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 31 for details on Oak'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 `>`.

Although boolean values are ordered, with `false < true`, booleans aren't numbers. They can't be converted into numbers by casting.

### 3.4 Character Types

Unicode not implemented yet. Characters currently have ASCII values.

Oak uses the Unicode character set throughout. Consequently the **char** data type is defined as a 16 bit, unsigned integer (**unsigned short**). Character strings are implemented with the **string** type (see “Arrays” on page 12).

### 3.5 Arrays

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

Oak checks subscripts 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**:

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

10, 3
```

The **prints** operator is one of a group of function- and constant-like operators whose functionality will eventually be moved into classes. This API is documented <<where?>>.

Arrays are allocated either where they’re declared (by specifying the dimensions of the array when it is declared) 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");
```

### 3.6 Types Created with the typedef Keyword

The **typedef** keyword and functionality might go away.

Programmers can define new, nonclass types using the **typedef** keyword. Types defined with **typedef** are not globally defined; instead, they are defined only within the scope of the class that contains the **typedef** statement. To use such a type in another class that is not a subclass, the programmer specifies the class and the type name together. For example, assume a class named AClass defines a type aType. A class that is not a subclass of AClass would define a variable named x with type aType as follows:

```
AClass.aType x;
```

Types defined with **typedef** can be thought of as synonyms for simple types, arrays, or existing classes. In particular, Oak doesn't support the C **struct** and **union** keywords, so the only way to create a type with multiple types of data fields is to create a class.

## 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 or *base* class. The derived class is also called a *subclass* of the other, which is sometimes 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 a class B is a subclass of A, then a value of B can be used as a value of A. In fact, if there is no ambiguity, then no explicit cast is needed. If a value of class A needs to be used as if it were of class B, the programmer can write a type conversion or *cast*. Conversions from a base to a derived class are always checked.

Oak supports only single inheritance, but some advantages of multiple inheritance are supported through the use of interfaces (see "Interfaces" on page 21). Instances of classes are stored in a garbage collected heap (see "Garbage Collection" on page 29), and the actual stack variable is an object reference.

Interfaces not implemented yet.

The base class of a class and the interfaces that the class implements (if any) are indicated in the class definition by the keywords **extends** and **implements**:

```
class classname    extends superclassname
                  implements interface1, interface2{
    /* . . . */
}
```

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

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

is the same as

Not implemented yet.

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

To limit the scope of a class definition, classes can be defined inside of classes.

```
class MyClass {
    class Helper {
        /* . . . */
    }
    Helper helper;
}
```

#### 4.1 Instance Variables

Instance variables are declared just like any other variable. They can be of any type and can have initializers. These initializers are executed when the instance is initialized. 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:

```
class Foo {
    /* . . . */
    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;
    /* . . . */
    prints(a + "\n"); // a == "this.a"
}
```

Overloading variable access with method access is not implemented yet.

Instance variable accesses can be overloaded with methods based on the name of the variable, as follows:

```
class Foo {
    int a;

    // Get...
    int a() { return a; }

    // Set...
    void a(int v) { this.a = v; }
}
Foo p;
p.a = 3; // equivalent to p.a(3)
i = p.a; // equivalent to i = p.a();
```

Overloading variable access allows classes to evolve by hiding the distinction between instance variables and accessor methods. For example, a point data type could be implemented as either polar or cartesian coordinates, without the client being aware of the distinction, and while retaining the convenient instance variable access syntax.

Changing variable access to method access does not require separately compiled modules to be recompiled. For example, assume you have a class A defined in file **A.oak** and a class B defined in **B.oak**, and B references a variable in an A object. If A is changed so that the referenced variable changes from being an instance variable to being a pair of accessor methods, **B.oak** does not need to be recompiled. The runtime system handles the change transparently.

## 4.2 Methods

A method is a function defined within a class. A method definition must follow this form:

```
[returntype] methodname ( <parameter list> ) {
    <method body>
}
```

The above form means that methods must:

- Have a return type if the method returns a value. If the method does not return any value, it must either have a **void** return type or no return type.
- Have a parameter list—the parameter list should be empty if the method has no parameters.
- Have a body—the body can be null.

You must currently specify a return type, even if the method doesn't return a value.

## 4.3 Overriding and Overloading Methods

Oak allows *polymorphic* method naming—defining 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 defining a method that has the same name and return type as another method, but a different parameter list.

**Note:** Return types are not used to distinguish methods. In each 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 defined the method must define a method with the same name, return type (or a subtype), 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 defines a method that has the same name and return type as another method (which has been defined in the class or in one of its superclasses), but a different parameter list. Oak 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
}

```

Not implemented yet.

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 an exception occurs.

#### 4.4 Used before Set

Not completely implemented.

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

#### 4.5 Static Variables and Methods

Variables and methods defined in a class can be declared **static**, which makes the same implementation apply to every object in the class and to the class itself. Unlike instance variables, static variables have the same value, no matter what instance of the class is used to obtain them. As shown in the following code example, both static variables and static methods can also be accessed using the class name, instead of using an instance of the class.

```

class Ahem {
    int i; // Instance variable
    static int j; // Static variable
    void seti(int I) { i = I; } // Instance method
    static void setj(int J) { j = J; } // Static method
};

Ahem a = new Ahem();
a.j = 2; /* valid; static var via instance */
Ahem.j = 3; /* valid; static var via class */
a.setj(2); /* valid; static method via instance */
Ahem.setj(3); /* valid; static method via class */
a.i = 4; /* valid; instance var via instance */
Ahem.i = 5; /* INVALID; instance var via class */
a.seti(4); /* valid; instance method via instance */
Ahem.seti(5); /* INVALID; instance method via class */

```

A static variable exists only once per Oak application, no matter how many objects of the class exist in the application. For distributed applications that run in multiple address spaces, each application has one occurrence of the static variable. When you refer to a static variable relative to some object (for example, `obj.aVar`) the static variable `aVar` is fetched from the address space where `obj` resides. To achieve synchronized access to static variables, you have to use a static method (see "Synchronized Methods" on page 19).

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

Not implemented yet.

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

Not implemented yet.

## 6 Volatile Variables

Variables marked **volatile** are treated specially by the optimizer. 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.

## 4.7 Transient Variables

Variables marked **transient** are treated specially when instances of the class are written out to the file system. Specifically, the values of transient variables are not written out.

## 4.8 Final Classes and Methods

The **final** keyword should be used for classes that cannot be superclasses and for methods that cannot be 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 and final methods (where the meaning of *small* is implementation dependent).

## 4.9 Properties of Variables

Not implemented; experimental.

Instance and static variables can be assigned user defined properties. The syntax is

```
prop(connection) Raster v;
```

which declares *v* to be an instance of class *Raster* that has the *connection* property. Properties are used by a variety of different runtime packages. For example, the development environment uses the *connection* property to label variables that are connections to other objects that the editor can hook up to other objects.

#### 4.10 Access to Variables and Methods

Not implemented; experimental. The public key stuff is especially subject to change.

Each variable or method defined 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.

**Note:** All classes in the same package can use all variables and methods defined in the classes in that package, regardless of **public**, **protected**, and **private** declarations (see “Packages” on page 22).

By default all variables and methods in a class are **protected**. Protected variables and methods can be accessed only by methods defined in the class and its subclasses, and not by 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 **private** type modifier makes a variable or method inaccessible even to subclasses (except those in the same package).

The following example shows how to specify access.

```
class Stuff {
    int i;           /* protected by default */
    public int j;    /* visible to everyone */
    private int k;   /* not even subclasses
                     see this */
    void method1() { } /* protected by default */
    void public method2() { } //public
    static private void method3() { } //private
    private static void method4() { } //private; alternate
    word order
};
```

Access to variables can also be controlled with public-key seals.

#### 4.11 Variables with Public-Key Seals

Access to an instance or static variable can be enforced by the use of a public-key seal. A seal is like an unforgeable<sup>1</sup> stamp that can be placed on an object module. These seals have a variety of meanings:

- upgrade** If a new module arrives that is to be installed as an upgrade to the current one, the new module must have been sealed with the same private key.
- requires** States that this module (or a particular instance or static variable) requires that client modules have a corresponding **has** seal.
- has** States that this module has permission to use interfaces that require this seal.

When a sealed class or a class with sealed variables is compiled, the private key specified by the name following the word **seal** is used to place a **requires** seal on the object module.

1. Made unforgeable through the use of RSA public key encryption and MD5 message digests.



#### 4.12 Synchronized Methods

The **synchronized** access specifier marks a method as being required to run in the monitor, so that it does not run at the same time as another method that needs access to the same resource. (The other method must also be declared **synchronized**.) For more information on **synchronized**, see the section on class Thread in the class documentation.

#### 4.13 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 value.

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; }
}
```

Method discrimination based on **int** versus **float** doesn't work.

```
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.

Instance variable initializations (see “Instance Variables” on page 14) are effectively turned into code that is prepended to all constructors. (Static variable initializations are executed at a different time, as described in “Static Variables and Methods” on page 16.)

```
class a {
    long n = 42;
    static long k = 99;
}

class Foo {
    int a;
    Foo(float a, int b) { this.a = a * b; }
    Foo(int a) { this.a = a; }
}
Foo z = new Foo(.4, 100);
```

The instance variables of superclasses are initialized by calling either the constructor for the base class or a constructor for the current class. If neither is specified in the code, “`super()`” is assumed. Calling a constructor must be the first thing in the method body; calling a constructor later is illegal.

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

```
super(parameters); /* Call base class constructor */
```

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

```
this(parameters); /* Call constructor from this
class. Normally this is only
done when there are multiple
constructors, all having
different parameter lists */
```

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); // no implicit call to Bar()
    }
}

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

#### 4.14 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*

---

Interfaces not implemented yet.

An interface specifies a collection of methods without defining their bodies. Interfaces provide encapsulation of method protocols without chaining the implementation to one inheritance tree. When a class implements an interface, it must implement the bodies of all the methods described in the interface. (If the implementing class is *abstract*—never instantiated—it can leave the implementation of some or all methods to its subclasses.) 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 definition (with the **interface** keyword) and a class that implements the interface.

```
interface Storing {
    void FreezeDry(Stream);
    void Reconstitute(Stream);
}

class Raster implements Storing, Painting {
    ...
    void FreezeDry(Stream s) {
        /* JPEG compress image before storing */
        ...
    }

    void Reconstitute (stream s) {
        /* JPEG decompress image before reading */
    }
}
```

The declaration syntax *classname*<*interfacename*> declares a variable to be an instance of some subclass of *classname* 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(Object<Storage> obj) {
        obj.FreezeDry(stream);
    }
}
```

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

## 6 *Packages*

---

Packages are groups of classes. They are a tool for managing a large namespace of classes and avoiding conflicts. Every class name is contained in some package. A package may be nested in some other package, or it may be a *root package*. Root package names generally correspond to organizations (corporations, schools, or governments) developing software. Subpackages generally correspond to suborganizations, projects, or products.

The **package** declaration defines the name of the package that all subsequent class definitions should be placed in:

```
package fp.os;
class Thread {
    ...
}
```

Public/private key protection is not implemented yet.

Each file is required to have a **package** statement at the beginning, and no other **package** statements. A package name can be associated with a public/private key pair used to control who is allowed to write classes that become part of a package.

Within a package, classes are free to access elements of other classes without regard to **public**, **protected**, and **private** declarations.

The **import** declaration is used to make the name of a class from another package available in the current compilation:

```
import fp.os.Thread;
Thread p = new Scheduler();
```

## 7 *Assertions*

---

Assertions aren't implemented yet.

Oak 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 throws an `AssertionFailedException` (see "Exceptions" on page 27).

Oak automatically converts assertions into exception traps and ensures that such checks are consistently and completely incorporated.

### 7.1 **Constraints on Class Variables**

The **assert** keyword can be used to declare a set of constraints on class variables. 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 method. All public 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 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[];
    int sizeof() {}
    int full();
    int 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.

# 8 *Expressions*

---

Expressions in Oak are much like expressions in C.

## 8.1 Operators

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

```
. [] ()
++ -- ! ~ instanceof new clone
* / %
+ -
<< >> >>>
< > <= >=
== !=
& (binary)
^
```

```

|
&&
| |
?:
= op=
,

```

### 8.1.1 Operators on Integers

If any operand is **long**, then the result type is **long**. Otherwise the result type is **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:  $op \text{ integer} \Rightarrow \text{integer}$**

Operator	Operation
-	unary negation
~	bitwise complement

**Table 2. Binary Integer Operators:  $\text{integer } op \text{ integer} \Rightarrow \text{integer}$**

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*

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 (not bitwise) 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 **!=**. Since **boolean** values are ordered, you can use the relational operators **<**, **>**, **<=**, and **>=**. The assignment operators also work: **&=**, **|=**, **^=**.

### 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 **/=**. 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 (non-signaling) Not-a-Number. Special mathematical values are documented *<<nowhere yet>>*.

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 paradoxically 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.

### 8.1.4 *Operators on Character Arrays*

The operator **+** concatenates arrays of characters, automatically converting operands if necessary.

In the current implementation, the left hand side of an *op=* operator gets re-evaluated. This is a bug

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

```
The value of a is 1
1.012
3.01
```

### 8.1.5 *Operators on Objects*

The unary operator **clone** is applied to an object instance. It atomically allocates space for a new object of the same type and copies the contents of the existing object into it, making the new object a perfect copy of the old one. This is normally used inside **new** to clone the prototype of some class, before applying the initializers (constructors).

This may be replaced by an operator that is more dynamic. And it may 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

Oak protects against doing anything illegal. Integers and floating point numbers can be cast back and forth, but integers cannot be cast to pointers. An object reference can be cast to a superclass with no penalty. Casting to a subclass generates a runtime check, which raises the `InvalidClassCastException` if the object is not, in fact, an instance of some subclass. Use the **instanceof** operator to determine whether such an exception would be raised for a cast.

Conversions between the arithmetic types might be checked and exceptions might be raised for range violations. C programmers should be especially aware that casting between **int** and **unsigned** is not a simple re-interpretation of a bit pattern.

## 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.

Not implemented yet.

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

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

### 9.2 Expressions

As in C, expressions are statements:

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

### 9.3 Control flow

Again, 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<sup>1</sup>

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 program terminating. However, programs can define *exception handlers* that *catch* the exception and recover from the error.

Some exceptions are thrown by the Oak runtime. However, any class can define its own exceptions and cause them to occur using **throw** statement. 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. Any code following the **throw** statement is not executed, and no value is returned by its enclosing method.

```

class GenericException.MyException {};

if (/* no error occurred */)
  /* do something */
else
  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) {
  prints("p was null\n");
}

```

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

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 that has a parameter type that matches the type 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 for an exception handler to resume execution at the point that the exception occurred.

For example, this code fragment:

```
class Foo {};

prints("now ");
try {
    prints("is ");
    throw new Foo();
    prints("a ");
} catch(Foo p) {
    prints("the ");
}
prints("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 bound to cause confusion.

Exception handlers can be nested, allowing exception handling to happen in more than one place. 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.

**finally** isn’t implemented yet.

The following example shows the use of a **finally** statement that is useful for guaranteeing that some cleanup code gets executed:

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

is the same as:

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

## 10 *Garbage Collection*

---

The Oak garbage collector makes most aspects of storage management simple and robust. Applications 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.

The algorithm used is a fairly conventional mark-and-sweep <<*refer to some old lisp book, like the Lisp 1.5 manual*>> with modifications for compaction and asynchronous operation.

