The Java Language:

A White Paper Overview

Harry H. Porter III
Portland State University
May 5, 2002

harry@cs.pdx.edu

Table of Contents

Abstract

This document provides a quick, yet fairly complete overview of the Java language. It does not discuss the principles behind object-oriented programming or how to create good Java programs; instead it focuses only on describing the language.

Introduction

Java is a programming language developed by Sun Microsystems. It is spreading quickly due to a number of good decisions in its design. Java grew out of several languages and can be viewed as a "cleaning up" of C and C++. The syntax of Java is similar to C/C++ syntax.

Charater Set

Almost all computer systems and languages use the ASCII character encoding. The ASCII code represents each character using 8 bits (that is, one byte) and there are 256 different characters available. Several of these are "control characters."

Java, however, uses 16 bits (that is, 2 bytes) for each character and uses an encoding called Unicode. The first 256 characters in the Unicode character set correspond to the traditional ASCII character set, but the Unicode character set also includes many unusual characters and symbols from several different languages.

Typically, a new Java program is written and placed in a standard ASCII file. Each byte is converted into the corresponding Unicode character by the Java compiler as it is read in. When an executing Java program reads (or writes) character data, the characters are translated from (or to) ASCII. Unless you specifically use Unicode characters, this difference with traditional languages should be transparent.

To specify a Unicode character, use the escape sequence `\u`*XXXX* where each *X* is a hex digit. (You may use either uppercase A-F or lowercase a-f.)

Non-ASCII Unicode characters may appear in character strings or in identifiers, although this is probably not a good idea. It may introduce portability problems with operating systems that do not support Unicode fonts. The Unicode characters are categorized into classes such as "letters," "digits," and so forth.

Comments

There are three styles of comments.

```
// This is a comment
/* This is a comment */
/** This is a comment */
```

The first and second styles are the same as in C++. The first style goes through the end of the line, while the second and third styles may span several lines.

The second and third styles do not nest. In other words, attempting to comment out large sections of code will not work, since the comment will be ended prematurely by the inner comment:

```
/* Ignore this code...
   i = 3;
   j = 4;   /* This is a comment */
   k = 5;
*/
```

The third comment style is used in conjunction with the JavaDoc tool and is called a JavaDoc comment. The JavaDoc tool scans the Java source file and produces a documentation summary in HTML format. JavaDoc comments contain embedded formatting information, which is interpreted by the JavaDoc tool. Each JavaDoc comment must appear directly before a class declaration, a class member, or a constructor. The comment is interpreted to apply to the item following it.

We do not discuss JavaDoc comments any further in this paper, except to say that they are not free-form text like other comments. Instead, they are written in a structured form that the JavaDoc tool understands.

Identifiers

An identifier is a sequence of letters and digits and must start with a letter. The definition of letters and digits for the Unicode character set is extended to include letters and digits from other alphabets. For the purposes of the definition of identifiers, "letters" also includes the dollar (`$`) and underscore (`_`) characters. Identifiers may be any length.

A number of identifiers are reserved as keywords, and may not be used as identifiers (see the section on Reserved Words).

Reserved Words (Keywords)

Here are the keywords. Those marked *** are unused.

```
abstract    default     if          private     this
boolean     do          implements  protected   throw
break       double      import      public      throws
byte        else        instanceof  return      transient
case        extends     int         short       try
catch       final       interface   static      void
char        finally     long        strictfp    volatile
class       float       native      super       while
const ***   for         new         switch
continue    goto ***    package     synchronized
```

In this document, keywords will be underlined, like `this`.

The following identifiers are not keywords. Technically, they are literals.

```
null
true
false
```

Primitive Data Types

The following are the basic types:

`boolean`	
`char`	*16-bit Unicode character*
`byte`	*8-bit integer*
`short`	*16-bit integer*
`int`	*32-bit integer*
`long`	*64-bit integer*
`float`	*32-bit floating point*
`double`	*64-bit floating point*

All integers are represented in two's complement. All integer values are therefore signed. Floating point numbers are represented using the IEEE 754-1985 floating point standard. All `char` values are distinct from `int` values, but characters and integers can be cast back and forth.

(Note that the basic type names begin with lowercase letters; there are similar class names for "wrapper classes.")

Useful constants include:

```
Byte.MIN_VALUE
Byte.MAX_VALUE

Short.MIN_VALUE
Short.MAX_VALUE

Integer.MIN_VALUE
Integer.MAX_VALUE

Long.MIN_VALUE
Long.MAX_VALUE

Float.MIN_VALUE
Float.MAX_VALUE
Float.Nan
Float.NEGATIVE_INFINITY
Float.POSITIVE_INFINITY

Double.MIN_VALUE
Double.MAX_VALUE
Double.Nan
Double.NEGATIVE_INFINITY
```

```
Double.POSITIVE_INFINITY
```

Boolean

There are two literals of type `boolean`: `true` and `false`. The following operators operate on `boolean` values:

`!`	Logical negation
`==  !=`	Equal, not-equal
`&	^`	Logical "and," "or," and "exclusive-or" (both operands evaluated)
`&&  ||`	Logical "and" and "or" (short-circuit evaluation)
`?:`	Ternary conditional operator
`=`	Assignment
`&=  |=  ^=`	The operation, followed by assignment

The assignment operator "=" can be applied to many types and is listed here since it can be used for `boolean` values. The type of the result of the ternary conditional operator "`?:`" is the more general of the types of its second and third operands. All the rest of these operators yield a `boolean` result.

Integers

Integer literals may be specified in several ways:

```
123             Decimal notation
0x7b            Hexadecimal notation
0X7B            Hexadecimal notation (case is insignificant)
0173            Leading zero indicates octal notation
```

There are four integer data types:

```
byte            8-bits
short           16-bits
int             32-bits
long            64-bits
```

Literal constants are assumed to be of type `int`; an integer literal may be suffixed with "`L`" to indicate a `long` value, for example `123L`. (You may also use lowercase "`l`", but don't since it looks like the digit "`1`.")

Floating-Point

Floating-point literals may be written in several ways:

```
34.
3.4e1
.34E2
```

There are two floating-point types:

```
float    32-bits
double   64-bits
```

By default, floating-point literals are of type `double`, unless followed by a trailing "`F`" or "`f`" to indicate a 32-bit value. You may also put a trailing "`D`" or "`d`" after a floating-point literal to indicate that it is of type `double`.

```
12.34f
12.34F
12.34d
12.34D
```

There is a positive zero (`0.0` or `+0.0`) and a negative zero (`-0.0`). The two zeros are considered equal by the == operator, but can produce different results in some calculations.

Numerical Operations

Here are the operations for numeric values:

expr++ *expr*--	Post-increment, post-decrement
++*expr* --*expr*	Pre-increment, pre-decrement
-*expr* +*expr*	Unary negation, unary positive
+ - *	Addition, subtraction, multiplication
/	Division
%	Remainder
<< >> >>>	Shift-left, shift-right-arithmetic, shift-right-logical
< > <= >=	Relational
== !=	Equal, not-equal
=	Simple assignment
+= -+ *= /= %= <<= >>= >>>=	The operation, followed by assignment

The << operator shifts bits left, filling with zeros on the right. The >> operator shifts right, with sign extension on the left. The >>> operator shifts right, filling with zeros on the left.

Character and String Literals

Character literals use single quotes. For example:

```
'a'
'\n'
```

The following escape sequences may be used in both character and string literals:

`\n`	newline
`\t`	tab
`\b`	backspace
`\r`	return
`\f`	form-feed
`\\`	
`\'`	
`\"`	
`\`*DDD*	octal specification of a character (`\000` through `\377` only)
`\u`*XXXX*	hexadecimal specification of a Unicode character

String constants may not span multiple lines. In other words, string literals may not contain the newline character directly. If you want a string literal with a newline character in it, you must use the `\n` escape sequence.

Implicit Type Conversion and Explicit Casting

A type conversion occurs when a value of one type is copied to a variable with a different type. In certain cases, the programmer does not need to say anything special; this is called an "implicit type conversion" and the data is transformed from one representation to another without fanfare or warning. In other cases, the programmer must say something special or else the compiler will complain that the two types in an assignment are incompatible; this is called an "explicit cast" and the syntax of "C" is used:

```
x = (int) y;
```

Implicit Type Conversions The general rule is that no explicit cast is needed when going from a type with a smaller range to a type with a larger range. Thus, no explicit cast is needed in the following cases:

`char` → `short`
`byte` → `short`
`short` → `int`
`int` → `long`
`long` → `float`
`float` → `double`

When an integer value is converted to larger size representation, the value is sign-extended to the larger size.

Note that an implicit conversion from `long` to `float` will involve a loss of precision in the least significant bits.

All integer arithmetic (for `byte`, `char`, and `short` values) is done in 32-bits.

Consider the following code:

```
byte x, y, z;
...
x = y + z;        // Will not compile
```

In this example, "y" and "z" are first converted to 32-bit quantities and then added. The result will be a 32-bit value. A cast must be used to copy the result to "x":

```
x = (byte) (y + z);
```

It may be the case that the result of the addition is to large to be represented in 8 bits; in such a case, the value copied into x will be mathematically incorrect. For example, the following code will move the value -2 into "x."

```
y=127;
z=127;
x = (byte) (y + z);
```

The next example will cause an overflow during the addition operation itself, since the result is not representable in 32 bits. No indication of the overflow will be signaled; instead this code will quietly set "x" to -2.

```
int x, y, z;
y=2147483647;
z=2147483647;
x = y + z;
```

When one operand of the "+" operator is a `String` and the other is not, the `String` concatenation method will be invoked, not the addition operator. In this case, an implicit conversion will be inserted automatically for the non-string operand, by applying the `toString` method to it first. This is the only case where method invocations are silently inserted. This makes the printing of non-string values convenient, as in the following example:

```
int i = ...;
System.out.println ("The value is " + i);
```

This would be interpreted as if the following had been written:

```
System.out.println ("The value is " + i.toString() );
```

Explicit Casts When there is a possible loss of data, you must cast. For example:

```
anInt = (int) aLong;
```

A `boolean` cannot be cast to a numeric value, or vice-versa.

When floating-point values are cast into integer values, they are rounded toward zero. When integer types are cast into a smaller representation (as in the above example of casting), they are shortened by chopping off the most significant bits, which may change value and even the sign. (However, such a mutation of the value will never occur if the original value is within the range

of the newer, smaller integer type.) When characters are cast to numeric values, either the most significant bits are chopped off, or they are filled with zeros.

Pointers are Strongly-Typed

In the following examples in this document, we will assume that the programmer has defined a class called "`Person`."

Consider the following variable declaration:

```
Person p;
```

This means that variable `p` will either be `null` or will point to an object that is an instance of class `Person` or one of its subclasses. This is a key invariant of the Java type system; whatever happens at runtime, `p` will always either (1) be `null`, (2) point to an instance of `Person`, or (3) point to an instance of one of `Person`'s subclasses.

We say that p is a "`Person` reference." Assume that class `Person` has two subclasses called `Student` and `Employee`. Variable `p` may point to an instance of `Student`, or `p` may also point to an instance of some other subclass of `Person`, such as `Employee`, which is not a `Student`.

Java has strong, static type checking. The compiler will assure that variable `p` never violates this invariant. In languages like C++, the programmer can force `p` to point to something that is not a `Person`; in Java this is impossible.

A class reference may be explicitly cast into a reference to another class. Assume that `Student` is a subclass of `Person`.

```
Person p;
Student s;
...
p = s;            // No cast necessary.
...
s = (Student) p;  // Explicit cast is necessary
```

The first assignment

```
p = s;
```

involves an implicit conversion. No additional code will be inserted by the compiler. The pointer will simply be copied. The invariant about variable `p` cannot be violated by this assignment, since we know that `s` must either (1) be `null`, (2) point to an instance of `Student`, or (3) point to an instance of one of `Student`'s subclasses, which would necessarily be one of `Person`'s subclasses.

The second assignment

```
s = (Student) p;
```

is a cast from a superclass reference down to a subclass reference. This must be checked at runtime, and the compiler will insert code that performs a check. For example, assume that `Employee` is a subclass of `Person`; then `p` could legitimately point to an `Employee` at runtime before we execute this assignment, without violating the invariant about `p`'s type. But if the pointer is blindly copied into variable `s`, we would violate the invariant about variable `s`, since it would cause `s` to point to something that is not a subtype of `Student`.

The compiler will guard against the above disaster by quietly inserting a "dynamic check" (i.e., "runtime check") before the code to copy the pointer. If `p` points to an object that is not a `Student` (or one of `Student`'s subclasses), then the system will throw a `ClassCastException`.

It is as if the compiler translates

```
s = (Student) p;
```

into the following:

```
if (p instanceof Student) {
  s = p;
} else {
  throw new ClassCastException ();
}
```

Assignment and Equality Operators

The assignment operator is "=". For example:

```
x = 123;
```

The assignment operator may be used as an expression, just as in "C":

```
if (x = 0) ...;
```

The equality operators "==" and "!=" test whether two primitive data values are equal or not. When applied to operands with object types, the "==" and "!=" operators test for "pointer identity." In other words, they test to see if the two operands refer to the same object, not whether they refer to two objects that are distinct but "equal" in some deeper sense.

```
Person p, q;
 ...
if (p == q) ...;
```

The "==" operation is often referred to as "identity" (instead of "equality") to make this distinction. Two `String` objects may be equal but not identical. For example:

```
String s, t;
s = "abc" + "xyz";
if (s == "abcxyz") ...;
if (s.equals ("abcxyz")) ...;
```

The first test will fail. The second test will succeed.

The "Not-A-Number" floating-point value is never identical with anything. Even the following test will be false:

```
if (double.Nan == double.Nan) ...;
```

Instanceof

The keyword `instanceof` may be used to determine whether the type of an object is a certain type. For example:

```
Person p = ...;
 ...
if (p instanceof Student) ...;
```

The type of the first operand (`p`) is determined at runtime. We assume that class `Student` is a subclass of `Person`. Consequently, it is possible that `p` may point to a `Student` object at runtime. If so, the test will succeed.

The second operand of `instanceof` should be a type (either a class or an interface).

If `instanceof` is applied to `null` (that is, if `p` is `null`), the result is always `false`.

Pointers in Java (References)

Pointers in "C" are explicit. They are simply integer memory addresses. The data they point to can be retrieved from memory and the memory they point to can be stored into. Here is an example in "C". Note that a special operation (`*`) is used to "dereference" the pointer.

```
struct MyType { ... };      // "C/C++" language
MyTpye *p, *q;
...
(*p).field = (*q).field;    // Get from memory & store into memory
...
p = q;                      // Copy the pointer
...
if (p == q) ...             // Compare pointers
...
if (*p == *q) ...           // Compare two structs
```

The "C++" language did not go beyond "C" in this aspect.

In contrast, pointers in modern OOP languages are implicit. To enforce this distinction, we usually call them "references," not "pointers", although they are still implemented as integer memory addresses. Just as in "C," the data they point to can be retrieved from memory and the memory they point to can be stored into. However, the dereferencing is always implicit.

```
class MyType { ... };      // Java language
MyTpye p, q;
...
p.field = q.field;         // Get from memory & store into memory
...
p = q;                     // Copy the pointer
...
if (p == q) ...            // Compare pointers
...
if (p.equals(q)) ...       // Compare two objects
```

One important difference is that in "C/C++" the programmer can explicitly manipulate addresses, as in this example:

```
p = (MyType *) 0x0034abcd; // "C/C++" language
(*p).field = ...;          // Move into arbitrary memory location
```

This sort of thing is impossible in Java. You cannot cast references back and forth with integers. One benefit is that the language can verify that memory is never corrupted randomly and that each variable in memory contains only the type of data it is supposed to contain.

Another benefit of the OOP approach to references is that the runtime system can identify all pointers and can even move objects from one location to another in memory while a program is running, without upsetting the program. (In fact, the garbage collector does this from time-to-time while the program is running.) When an object is moved, all references can be readjusted and the program will never by able to detect that some of its pointers have been changed to point to different memory addresses.

Operator Syntax

Here is a list of all the operators, in order of parsing precedence. All operators listed on one line have the same precedence. Operators with higher precedence bind more tightly.

highest	`[]` `.` (*params*) *expr*`++` *expr*`--`
	`++`*expr* `--`*expr* `+`*expr* `-`*expr* `~` `!`
	`new` (*type*) *expr*
	`*` `/` `%`
	`+` `-`
	`<<` `>>` `>>>`
	`<` `>` `<=` `>=` `instanceof`
	`==` `!=`

	&											
	^											
	|											
	&&											
	||											
	?:											
lowest	=	+=	-=	*=	/=	%=	<<=	>>=	>>>=	&=	^=	|=

All operators are left-associative except for assignment. Thus

```
a = b = c;
```

is parsed as:

```
a = (b = c);
```

Here are some comments about the operators:

`==` `!=`	Identity testing (i.e., pointer comparison)
`/`	Integer division: truncates toward zero `-7/2 == -3`

`%`	Remainder after /
	`(x/y)*y + x%y == x`
	`-7%2 == -1`

`[]`	Array indexing
`.`	Member accessing
(*params*)	Message sending
`&` `|` `^` `!`	Logical AND, OR, XOR, and NOT (valid on `boolean` values)
`&` `|` `^` `~`	Bitwise AND, OR, XOR, and NOT (valid on integer and `char` values)
`<<` `>>` `>>>`	Shift bits (SLL, SRA, SRL)
`&&` `||`	Boolean only, will evaluate second operand only if necessary
`?:`	Boolean-expr ? then-value : else-value
(*type*) *expr*	Explicit type cast
`+`	Numeric addition and `String` concatenation

Expressions as Statements

Just as in "C", every expression can be used as a statement. You simply put a semicolon after it. Several sorts of expressions occur commonly and are often thought of as statements in their own right, although technically they are just examples of expressions occurring at the statement level.

Assignment Statement The assignment operator may be used at the statement level.

```
x = y + 5;
a = b = c = -1;    // Multiple assignment is ok
```

Message-Sending Statements Message-sending expressions may be used at the statement level:

```
p.addDependent (a,b,c);
```

A method may be non-void or void. That is, it may either return a result or not. If a method returns a result and the method is invoked at the statement level, the result will be discarded.

Increment and Decrement Statements Another sort of expression that commonly occurs at the statement level is given in these examples:

```
i++;
j--;
++i;   // same as i++;
--j;   // same as j--;
```

Object Creation Statements The `new` expression may be used at statement level, as in the following. In this case, the object is created and its constructor is executed. The `new` expression returns a reference to the newly constructed object, but this reference is then discarded.

```
new Person ("Thomas", "Green");
```

Flow of Control Statements

The `while` loop is the same as in "C/C++":

```
while (boolean-condition) statement;
```

The `for` loop is the same as in "C/C++." Here is an example:

```
for (i=0,j=100; i<5; i++,j--) statement;
```

The `do` statement has the following form:

```
do statement while (boolean-condition);
```

Braces "{" and "}" may be used to group statements into blocks, just as in "C/C++". Here are some examples combining flow of control statements with blocks.

```
while (boolean-condition) {
  statement;
  statement;
  statement;
}
for (i=0; i<5; i++) {
  statement;
  statement;
  statement;
}
```

```
do {
   statement;
   statement;
   statement;
} while (boolean-condition);
```

In the switch statement, the expression is evaluated to determine which case to execute. The switching expression must have an integer type. For example:

```
switch (integer-expression) {
   case 23:
         statement;
         statement;
         break;
   case 45:
         statement;
         statement;
         break;
   case 51:
   case 52:
   case 53:
         statement;
         statement;
         break;
   default:
         statement;
         statement;
         break;
}
```

The break statement is optional; if missing, control will fall through into the following case. The default clause is also optional.

Loops and blocks may also be labeled. This label may be used in break and continue statements. For example:

```
              while (condition-1) {
my_loop:         while (condition-2) {
                    while (condition-3) {
                       ...
                       break my_loop;
                       ...
                       continue my_loop;
                       ...
                    }
                 }
              }
```

In the break and continue statements, the label is optional. If missing, the break or continue will apply to the innermost do, for, while, or switch.

The return statement has two forms:

```
return;
return expression;
```

The first form is used to return from void methods while the second form is used to return from methods that yield a result. If a non-void method expects a result, then the expression supplied must be a sub-type of (or equal to) the type expected. For example, if the method returns a `double`, then the `return` statement may provide an `int`, since an `int` may be implicitly coerced into a `double`. If the method returns an object of type `Person`, the `return` statement may provide a `Student` object, since `Student` is a subtype of `Person`.

The `if` statement has the same form as in "C/C++":

```
if (boolean-expression)
    statement-1;
else
    statement-2;
```

The `else` clause is optional. Typically the two statements are blocks, giving a form like this:

```
if (boolean-expression) {
    statement;
    statement;
    statement;
} else {
    statement;
    statement;
    statement;
}
```

Arrays

Arrays may be declared as in this example:

```
Person [] p;
```

To create an array object with 10 elements, use this form of the `new` expression:

```
new Person [10]
```

For example:

```
Person [] p = new Person [10];
```

This creates an array with the following elements:

```
p[0], p[1], p[2], ... , p[9]
```

The older "C/C++" syntax, shown next, has the "p" and the "[]" reversed. This syntax can also be used but it is discouraged:

```
Person p [] = new Person [10];
```

In either case, the array object is created and all 10 elements are initialized to null. You may wish to initialize the elements of the array, as in this example:

```
Person [] p = new Person [10];
for (int i = 0; i < p.length; i++) {
   p[i] = new Person(...);
}
```

In this example, we used "new Person(...)" to create a Person object. The new expression is discussed later. It returns a pointer, which is subsequently copied into one of the array elements.

The size of an array can be obtained by applying the "length" attribute to an array object, as in:

```
p.length
```

Reading and writing array elements can be done using traditional [] notation:

```
p[i] = p[j];
```

If an attempt is made to read or write an array element that does not exist, the index-out-of-bounds error will be signaled. (This exception is called "ArrayIndexOutOfBoundsException." Exception handling will be discussed later.)

Arrays may be initialized using syntax like this example:

```
int [] [] a = { {1, 2}, {4, 5, 6}, {3}};
```

An extra comma is allowed after the last item in these lists, as in:

```
int [] a = {1, 2, 3, 4, 5, 6, };   // extra comma at end is optional
```

The extra comma is convenient when editing a Java source code file. Often you may want to delete the last element or copy lines, as in the following example.

```
int [] [] a = {
                {1, 1, 4, 1, 1, 1},
                {1, 1, 5, 1, 1, 1},
                {1, 1, 6, 1, 1, 1},
              };
```

Strings

The `String` class is included in one of the Java packages. You may declare a `String` variable and give it an initial value like this:

```
String x = "hello";
```

`String`s may be printed as follows:

```
System.out.print (x);
System.out.println (x);   // Follows it with a newline
```

`String`s may be concatenated with the `+` operator. For example:

```
x = x + " there";
System.out.println (x + " everyone!");
```

The + operator is treated somewhat specially. When one operand is a `String` and the other is something else (like an "`int`"), the second thing will be converted to a `String` and then concatenated with the first thing. This makes it very convenient to print out data values, as shown in the next example:

```
int i = ...;

System.out.println ("The value is " + i);
```

This implicit conversion occurs whenever non-`String`s are used in the concatenation operation and is done by invoking the `toString()` method. Therefore, this previous example is equivalent to:

```
System.out.println ("The value is " + (i.toString()));
```

The characters in a `String` are numbered 0..(length-1). To find the length of a `String`, use:

```
x.length
```

There are many useful `String` methods. Here are a few:

```
x.length () ➔ int
x.charAt (int) ➔ char
x.indexOf (char) ➔ int
x.equals (String) ➔ boolean
x.equalsIgnoreCase (String) ➔ boolean
x.startsWith (prefixString) ➔ boolean
x.endsWith (suffixString) ➔ boolean
x.compareTo (String) ➔ -1,0,+1
x.substring (startPos,endPos) ➔ String
x.toLowerCase () ➔ String
x.toUpperCase () ➔ String
x + y ➔ String
x.toCharArray () ➔ char[]
```

`String`s are immutable. That is, they may not be modified once created.

To determine if two `String`s have the same sequence of characters, do not use the == identity operator. Instead, use the "`equals()`" method. This is discussed in the next few paragraphs.

There are two ways to test the equality of objects. The first is to test whether the objects are the same object. That is, given two pointers to two objects, you can test whether the pointers are equal (using ==) as shown in the next example. The == test is called "object identity."

```
String s1 = ...;
String s2 = ...;
...
if (s1 == s2) {
  // both point to the same object
}
```

The second way is to ask whether two objects contain the same data. This test is called "object equality" and is performed with the `equals()` method. The `equals()` message is understood by all `Object`s. For `String`s, this method will test whether the `String`s contain the same characters in the same order.

```
if (s1.equals(s2)) {
  // both contain the same character sequence
}
```

Note that "equal" `String`s will not necessarily be "identical." You may have two different instances of `String` that happen to contain the same character sequences. On the other hand, if two `String`s are "identical," they will necessarily be "equal."

[In an earlier version of Java (Java 1.0.1) it was the case that all "equal" `String`s were also "identical," but this is not true of more recent versions. In other words, there was only one copy of every distinct `String`. The earlier approach made it quick to compare two `String`s (you merely had to compare pointers) but it required that a table of all known `String`s be maintained behind the scenes and whenever a new `String` was created, this table had to be searched to see if we already had such a `String` in existence.]

The class `StringBuffer` has many of the same methods as `String` but adds the functionality to allow you to modify the characters. Here are some additional methods for that class:

```
StringBuffer (String) ➔ StringBuffer
StringBuffer (initCapacity) ➔ StringBuffer
StringBuffer () ➔ StringBuffer

StringBuffer x, y;
x.append (y) ➔ StringBuffer
x.setCharAt (int,char) ➔ void
x.setLength (int) ➔ void
```

Classes

The following example illustrates how a class is defined.

```
public class Person {
   String first;
   String last;
   int age;
   static int total = 0;
   Person (String f, String l, int a) {
      first = f;
      last = l;
      age = a;
      total++;
   }
   String getName () {
      return last + ", " + first;
   }
   void setName (String f, String l) {
      first = f;
      last = l;
   }
   static int getCount () {
      return total;
   }
}
```

The keyword `class` is followed by the name of the class being defined. In this example, we are defining a class called `Person`. Class names are capitalized.

A number of modifiers may be associated with a class and they are listed before the keyword `class`. In this example, the class is declared to be `public`. Modifiers of a class include the following keywords:

```
public
abstract
final
strictfp
```

A class definition contains a number of "members." The first three members are "data members." Data members are sometimes called "fields" or "instance variables." The syntax for data members is similar to variable declarations. Data members may also have modifiers, although none are shown in this example.

Each `Person` object will have three fields, called `first`, `last`, and `age`. The fourth member (named "`total`" in this example) has the modifier `static`, which means that there will only be a single copy of the `total` variable regardless of how many `Person` objects have been created. "Static fields" are sometimes called "class variables;" the concept is the same.

The next member is a "constructor" method. A constructor looks like a normal method, except the name of the method (`Person`) is exactly the same as the class name. A constructor provides code to create a new object of this class. Whenever a new `Person` object is to be created, this constructor method will be invoked. Constructor methods may access the fields of the newly created object. This example is typical in that the constructor initializes the fields of the new object based on arguments passed to it. In this example, the constructor also modifies the static field "`total`" to keep a count of the total number of `Person` objects created.

A constructor always has the same name as the class. In this example, the class is `Person` and the constructor is `Person()`. There will often be several constructors, differing only in the number or types of arguments each expects. Whenever an object is created, one of the constructors must be invoked; the compiler will determine this by the number and types of arguments provided in the `new` expression.

The next two members are methods called `getName()` and `setName()`. There will typically be many methods differing in names and argument types. Methods may return a value (as `getName`() does) or may return no value (as `setName`() does).

Methods will usually have modifiers, although in this example none are provided for `getName()` and `setName()`.

The last method `getCount()` has the modifier `static`, and is called a "static method." Static methods are sometimes called "class methods." They may not access the fields (i.e., data members) of the object.

To invoke a normal (i.e., non-static) method, an object is provided; this object is called the "receiver" and the method is invoked on that object. In the case of a static method, there is no receiver object. Within a static method, you may not access the non-static fields because there is no `this` object when a static method is executed. However, within a static method you may access the static fields, since static fields are shared across the class.

In this example, the static method simply returns the value of the static variable `total`.

Object Creation

In the following example, a variable of type `Person` is declared and a new `Person` object is allocated:

```
Person p;
p = new Person ("Harry", "Porter", 50);
```

Variable declarations may have initializing expressions (just as in "C/C++") so these two lines could also be written as:

```
Person p = new Person ("Harry", "Porter", 50);
```

This causes a new object of class `Person` to be allocated on the heap and a reference to that object to be stored in the variable `p`.

Our example `Person` class did not have a "no-argunment" constructor, but we may modify it as follows:

```
public class Person {
    ...
   Person () {
      first = "John";
      last = "Doe";
      age = 0;
      total ++;
   }
   ...
}
```

Now we may create a new `Person` with:

```
Person p = new Person ();
```

We may also create objects in other contexts besides variable initialization. For example, we may create a `Person` object and use it immediately as an argument to some other method:

```
grp5.addToGroup (new Person ("Susan", "Brown", 20), "President");
```

When an object creation is attempted, but there is insufficient memory available, then the following exception will be thrown:

```
OutOfMemoryError
```

Interfaces

Java includes support for separating "specification" from "implementation." We use a "class" to provide the implementation details for objects. The class tells how the object will be represented (fields) and what code it will execute when messages are sent to it (methods). We use an "interface" to provide a specification for objects. The interface tells what messages will be understood by an object, but gives no details on how the objects will be implemented.

An interface is specified with syntax as illustrated by this example:

```
interface MyInter extends OtherInterA, OtherInterB, OtherInterC {
  int foo (...);
  int bar (...);
  ...
  int x = 123;
  double pi = 3.1415;
}
```

Every interface has a name (in this example, the interface is named `MyInter`), which must begin with an uppercase letter. The `extends` clause is optional and can list one or more other interfaces. In this example, the interface being defined will extend the interfaces named `OtherInterA`, `OtherInterB`, and `OtherInterC`. The body of the interface lists a number of messages and constants. In this example, we see two messages, named `foo` and `bar`, and two constants, called `x` and `pi`.

An interface may also include nested classes and interfaces, but these are not shown here. The messages, constants, nested classes, and nested interfaces may be listed in any order.

Note the distinction between "messages" and "methods." The syntax of these messages is just like the syntax of methods, except that there is no body of code. Interfaces contain messages and classes contain methods.

```
void myFunct (int a, char ch);                              // message
...
void myFunct (int a, char ch) { ... statements ... }       // method
```

An interface tells what messages an object must be understood, while a class tells how the object will implement the message. The class provides methods to implement the messages: the class will provide one method for each message in the interface.

To take an example, let us define an interface for "taxable entities." A taxable entity must be able to respond to certain messages, for example, to query its name and tax ID number and to compute and return the tax due.

```
interface TaxableEntity {
  String getName ();
  int getID ();
  int computeTax (int year);
}
```

Several classes may implement (or "follow" or "respect") the `TaxableEntity` interface. For example, the classes `Person` and `Corporation` may be two classes that follow this interface.

```
class Person implements TaxableEntity {
  ...
  String getName () { ... }
  int getID () { ... }
  int computeTax (int year) { ... }
  ...
}
class Corporation implements TaxableEntity {
  ...
  String getName () { ... }
  int getID () { ... }
  int computeTax (int year) { ... }
  ...
}
```

As you can see, the syntax for class definitions allows an optional `implements` clause which indicates the relationship between a class and one (or more) interfaces. In addition to fields and other methods that the class provides, the class must provide methods for the messages in the interface, and the compiler will check this.

An interface may have no messages at all. In such a case, the interface is known as a "marker interface." Marker interfaces are used as a form of documentation and program organization. Since the marker interface adds no additional messages, its presence has little impact to the compiler. Instead, such an interface acts as a signal to the users that the objects have some special behavior or property. An example marker interface is `Cloneable`, which is discussed elsewhere in this document.

Declarations

Variables are declared in declarations. Here are some examples:

```
int i = 1;
int j = 2;
int k = 3;
```

There is a variant syntax, just as in the "C" language:

```
int i = 1, j = 2, k = 3;
```

Variables may also be declared without assigning them initial values, as in:

```
int i, j, k;
```

The above syntax can be used for

- local variables in methods
- normal fields in classes
- static fields in classes (class variables)
- final fields (constants in classes and interfaces)

The variables in a method are called "local variables". If no initial value is given, then local variables are not initialized to default values; instead the compiler will check that no undefined variables are ever used. In other words, the compiler will perform a data-flow analysis of the method to make sure that the variable is always assigned before it is used, and complain if it appears that an uninitialized variable is being used.

Fields may be given initial values. If no initial value is given, the field will be given a default value, based on its type.

```
class MyClass {
  int x;            // initialized to default of 0
  int y = 47;
  ...
```

```
}
```

The default values for field initialization are:

`boolean`	`false`	
`char`	`\u0000`	*16-bit Unicode character*
`byte`	`0`	*8-bit integer*
`short`	`0`	*16-bit integer*
`int`	`0`	*32-bit integer*
`long`	`0`	*64-bit integer*
`float`	`0.0`	*32-bit floating point*
`double`	`0.0`	*64-bit floating point*
`<object reference>`	`null`	*pointer to an object*

The initialization expression may be rather complex. In fact, it may contain method invocations (and therefore result in arbitrary user code being executed).

```
int x = 2 * a.foo (b, c);
```

The initializing expression in a field declaration may not throw any checked exceptions, since there is no "surrounding code" to catch the exception.

The following access modifiers may appear in declarations:

- `public`
- `private`
- `protected`

  These control the "access visibility" of fields, and are discussed elsewhere in this document.
- `static`

  A static field is a "class variable." There is only one copy of the variable and it is shared by all instances of the object.
- `final`

  A `final` variable is a constant. Once given a value, it may not be reassigned. Local variables may be marked `final`; normally they would have an initializing expression and any assignments would be disallowed. You may also leave out the initializing expression, in which case the compiler will check that the method has exactly one assignment to the variable. A `final` field would normally have an initializing expression. If this is left off, then the field must be initialized within an initialization block or within the constructor. Parameters may also be marked as `final`, in which case they may not be assigned to within the method.
- `volatile`

  In the presence of multiple threads and shared memory, a field's value may be changed asynchronously by another thread. Normally the compiler / virtual machine assumes that fields do not get changed by other threads and the virtual machine may cache the field's value in registers and so on. This modifier prevents such cacheing and forces the virtual machine to reload a field from memory every time it is used.

A variable declaration may specify arrays with brackets. Here is an example creating three 2-dimensional arrays of Persons:

```
public static volatile Person [] [] x, y, z = ... ;
```

Types: Basic Types, Classes, and Interfaces

Variables are declared in declarations, which have the following general form. (We are ignoring some details of the syntax here, since this section is discussing types, not declarations.)

```
<type> <var> = <expr>;
```

For example:

```
int i = 0;
```

The <type> can be either:

- primitive types (like `int`)
- classes (like `Person`)
- interfaces (like `TaxableEntity`)

Throughout this document, when we say "type" we mean "class, interface, or primitive type."

Consider a variable that has, as its type, an interface:

```
TaxableEntity t;
```

This means that `t` will always point to a class that implements `TaxableEntity`. So we could do this:

```
t = new Person (...);
```

since `Person` implements the `TaxableEntity` interface.

Assume that `Student` is a subclass of `Person`. Then `Student` will also have the required methods (`getName`, `getID`, and `computeTax`). This is true because either `Student` will inherit them from `Person`, or `Student` will override them. But in either case `Student` must have methods for these messages. Therefore, it is safe to allow `t` to point to a `Student` object. There can never be a "Message Not Understood" error at runtime.

More on Interfaces

Members of an interface may not have any modifiers. For example:

```
interface TaxableEntity {
```

```
  public static final void foo (...);
  ^^^^^^ ^^^^^^ ^^^^^                    // Syntax error here
  ...
  private float = 3.1415;
  ^^^^^^^                                 // Syntax error here
}
```

(Here is why modifiers are not allowed. First, the `public` keyword is not used for the messages and constants in the interface. All the members of an interface are public by default. Making them private or protected makes no sense since it would mean that the member is only accessible from code in the interface or in its sub-interfaces, but interfaces do not contain any code at all. A `final` method may not be overridden; the messages in interfaces have no implementation and they must be overridden to be used. Therefore messages are never `final`. On the other hand, all variables in an interfaces are constants, so they are always `final` by default. Other attributes (`native`, `strictfp`, `synchronized`) constrain the implementation in some way; these are not allowed since the interface is a specification, and does not contain implementation details.)

A class may implement several interfaces. Here is an example:

```
class Person implements TaxableEntity, AnimateThing, GraphicObj {
  ...
}
```

Any field appearing in an interface is implicitly `static` and `final`. Thus, any field in an interface is defining a constant. The constant must have an initializing expression. We can use the constant like we use any static field. The normal syntax is to list the name of the interface and the field, separated by a period, as in this example:

```
... a + (5 * TaxableEntity.pi) + b ...
```

We can also refer to the constant through the use of a variable whose compile-time type is the interface, but this is not recommended since it is a little unclear. Here is an example:

```
TaxableEntity t;
...
... a + (5 * t.pi) + b ...     // Not recommended
```

Garbage Collection

Java has a built-in "garbage collector" which will automatically reclaim unreachable objects. Thus, there is no explicit object deletion operator. If an object can be reached via pointers from other objects, then its space will not be reclaimed; if the object is not reachable, then the garbage collector will eventually collect its memory space and reuse it for other objects.

Object Deletion and Finalize

When the garbage collector identifies an object as unreachable and therefore as collectible, it gives the programmer one last chance to look at the object before it is reclaimed. If the object has a method called `finalize()`, it is invoked before the object is collected.

More particularly, if the class of the object has this method:

```
public void finalize () throws Throwable {
  ...
}
```

then it will be invoked. Since there may also be a `finalize()` method implemented in the superclass, it is good practice to invoke

```
super.finalize();
```

within the method.

The `finalize()` method may contain arbitrary Java code. In particular, it may modify pointers to make the object in question once again reachable, and therefore no longer collectible. If so, the garbage collector will not collect the object. (Making the object reachable again is called "resurrecting" the object. It is a dubious programming practice.) At some later time, the object may once again become unreachable and identified as collectable; however, the `finalize()` method will not be invoked a second time: the `finalize()` method is invoked at most once per object. The second time the object is identified as unreachable, it will simply be collected.

Note that the garbage collector runs at somewhat unpredictable intervals. The exact timing of when an object is identified as unreachable, or when the `finalize()` method is invoked is therefore somewhat random.

Accessing Fields

The fields of an object may be accessed from code outside the code of the class by using the "dot" operator:

```
x = p.first;
p.last = "Smith";
```

The fields of an object may be accessed from code inside the class by simply naming them:

```
class Person {
  ...
  String getName () {
    return last + ", " + first;
  }
}
```

In this case, it is understood that the receiver is meant; it is the fields of the receiver object that are being referred to. This can be made explicit by specifying the receiver with the `this` keyword:

```
class MyClass {
  ...
  void myMethod () {
    ...
    this.field = ...;
    ...
    ... this.field ...
    ...
  }
}
```

In the above example, the `this` is optional; normally such code would be written as follows:

```
class MyClass {
  ...
  void myMethod () {
    ...
    field = ...;
    ...
    ... field ...
    ...
  }
}
```

Subclasses

Classes are related to one another by the subclass / superclass relationship. The class hierarchy is tree-shaped: every class has exactly one superclass except the class named "`Object`", which is the root of the class tree. In Java terminology, we say that a subclass "extends" its superclass.

In the following example, the `Person` class is extended by `Student`. This is specified with the `extends` keyword:

```
class Student extends Person {
  ...
}
```

Here is another class called `Employee` which extends class `Person`:

```
class Employee extends Person {
  ...
}
```

The subclass will inherit all the fields in the superclass, and may add additional fields if desired. The new fields may have the same names as fields in the superclass. If a field in the subclass has

the same name, the field in the superclass is hidden. To get at the hidden field, you may do it two ways.

First, within a method in the subclass, you may use `super`:

```
...super.field...
```

Second, you may access the field by using a reference to the superclass instead of a reference to the subclass:

```
Student s = ...;
...s.field...       // gets the new Student field

Person p = s;
...p.field...       // gets the hidden Person field
```

Note that when you access a field, the declared class of the variable is used, whereas when you send a message, the runtime class of the object is used to determine which method is executed.

Access Control / Member Visibility

Each member of a class has an access visibility. Each member has either:

- public visibility
- protected visibility
- package visibility (the default)
- private visibility

These are shown in order: "public visibility" specifies no access control and "private visibility" specifies the greatest restriction. Each visibility is more restrictive than the previously listed visibility.

By "member" we mean either a "field" or a "method."

Public Visibility A `public` member may be accessed from any code anywhere. That is, a `public` member may be accessed from code in other classes and in other packages in the same program. By "accessed," we mean a field may be read or modified or a method may be invoked.

Protected Visibility A `protected` member may be accessed from any code in the same package as the class containing the field. In addition, a `protected` member of class P may be accessed from code in class S (assuming S is a subclass of P) using an object reference whose type is class S or one of S's subclasses.

Consider an example with a superclass `Person` and a subclass `Student`, which extends `Person`. Assume that `Person` has a protected field called "`field`." Also assume that we have another class `Employee` which is a subclass of `Person`, but which is not related to `Student`.

```
class Person {
  protected int field;
  ...
}
class Student extends Person {
  ...
}
class Employee extends Person {
  ...
}
```

When is the following access allowed by the compiler?

```
x.field
```

It depends both on the declared type of the variable "x" and where the access occurs.

Every `Student` and every `Employee` will have a field called "`field`". In code within class `Student`, access to `field` is legal only when "x" has a type of `Student`, or one of its subclasses. So a method that is invoked on a `Student` may access "`field`". Within such a method, the receiver (`this`) has type `Student`. If the method also deals with other `Students` besides the receiver, it may access "`field`" in the other objects as well.

However, imagine a `Student` method which deals with `Persons` that are not `Students`. (Perhaps there is a method called `assignAdvisor()` which is passed an `Employee` object.) Within such a method, the code may not access the protected parts of the `Employee` object. Code within `Student` may not access the `Employee`'s `field`. So even if "x" has type `Person` or `Employee`, the access would not be allowed from code in `Student`.

The rule about `protected` visibility seems complex but the intuition is straightforward: When a subclass (like `Student`) extends a superclass (like `Person`), we want to give `Student` access to the protected parts of `Persons`. So when an extender of `Person` is working with a `Student` (or one of its subclasses), that code may access the protected parts of the object. There may be other extensions to `Person` (like `Employee`) but the implementation details of those classes will remain hidden within `Student`. If the implementation details of `Employee` change later, it should not affect `Student` or its subclasses.

Exception #1: Note that "protected" members are also accessible throughout the package, so if `Person` or `Employee` were defined in the same package as the access in question, then the access would be allowed.

Exception #2: Static members can be accessed from code in any subclass. In this example, if "`field`" were also declared to be static, then the access would be allowed from code in class `Student`, even if "x" had a type of `Employee`. This applies to static fields, static members, and even static classes and static interfaces.

For example, if `totalPopulationCount` is a `protected static` field in `Person`, it may be accessed from methods in `Person`, `Student`, `Employee`.

Package Visibility A field with "package visibility" may be accessed from any code in classes in this package but not from code in other packages. "Package visibiliy" is the default if no visibility access is specified, and it is never specified explicitly. (There is a `public` keyword, but it is used for a different purpose.) The assumption is that a package will be created as a unit in isolation. For example, a package would be created as a single programming act by a single programmer. Therefore, all parts of the package should trust all other parts.

Private Visibility A `private` field may be accessed from code in only the class containing the field, but not in its subclasses. For example, if `ssNumber` is a private field in `Person`, it may be accessed only from methods in class `Person`, and not from `Student` or `Employee`.

Sending Messages

To invoke a method, we must send a message. Whenever a message is sent, a method will be invoked. A message is sent to an object, called the "receiver." For example, if `p` is a `Person` object, we can send a message as follows:

```
p.setName ("Thomas", "Green");
```

Even when no arguments are supplied in a message-send, the parentheses are still required:

```
x.foo (a, b, c);
x.bar ();
```

Because parentheses are required, you can easily tell whether a message is being sent or whether a field is being accessed:

```
y = x.myMeth ();
y = x.myField;
```

The receiver may be a sub-expression and the arguments may also be sub-expressions. Parentheses can be used to group sub-expressions.

```
(x.bar()) . setName ( "Sue"+"Ann", y.computeLast(a,b,c) );
```

Message sending expressions are parsed as left-associative, so the previous example could also be written as follows, with no change in meaning:

```
x.bar().setName("Sue"+"Ann", y.computeLast(a,b,c));
```

Consider a message sending expression:

```
x.foo (a, b, c);
```

At compile-time, the types of all expressions and sub-expressions will be determined. Assume that we have this definition of `x`:

```
Person x;
```

The compile-time type of the receiver is therefore `Person`.

Next, assume that there is a subclass of `Person` called `Student`. At runtime, the variable `x` may refer to a `Student` object. The following assignment is legal and would create this situation:

```
Student s =  ...;
x = s;
```

Now, go back to the message-sending expression in question:

```
x.foo (a, b, c);
```

The compile-time type of the receiver is `Person`, but the run-time class of the receiver is `Student`. Assume that `Person` includes a `foo()` method and that `Student` overrides the `foo()` method.

```
class Person {
   void foo (...) { ...}
     ...
}
class Student extends Person {
   void foo (...) { ...}
     ...
}
```

Any subclass will either inherit a method or override it. Therefore, any subclass of `Person` must have a `foo()` method. The compiler can conclude that the message-sending expression is legal, since `Person` provides a `foo()` method. Whatever kind of object `x` points to at run-time, it must be a subclass of (or equal to) `Person` and any subclass extending `Person` must either inherit or override the `foo()` method.

There is no "Message Not Understood" error in Java. In this way Java differs from untyped languages like Smalltalk.

At run-time, the class of the receiver is determined. (In this case, the receiver is a `Student` object.) Then, the `foo()` method from that class is invoked and executed. The `foo()` method from `Person` will not be invoked, at least in this example. Which method will be invoked can only be determined at run-time. This is called "dynamic binding" since it must be done at run-time.

Within a method, a message may be sent to the receiver by using the `this` keyword:

```
class Person {
  void foo (...) {
    ...
    this.bar (...);
    ...
  }
}
```

In this case, the `this` keyword can be omitted. If no receiver of a message is specified, `this` is assumed. So, the above is equivalent to:

```
class Person {
  void foo (...) {
    ...
    bar (...);
    ...
  }
}
```

Arguments are Passed by Value

All arguments to methods are passed by value, as in "C". The argument value is copied to the parameter. (Parameters are sometimes called "formal variables.") Within the method, the parameter may be updated, but the definition of "pass by value" is that the original argument will not be modified.

```
x.foo (i);
...
void foo (int a) {
  a = 5;                // OK.  Will not modify i.
}
```

References to objects may be passed and they are passed "by value," too.

```
Person p = ...;
x.foo2 (p);
...
void foo2 (Person a) {
  a.setName ("Thomas");
}
```

During message sending, the objects themselves are never copied; instead object references are copied and passed. Of course, the method may follow the reference and modify the object.

"this" and "super"

Within a method, you may refer to the receiver object with the keyword `this`. For example, you can send another message to the receiver:

```
class Person {
  ...
```

```
  void foo () {
    ...
    this.anotherMessage ();
    ...
  }
  ...
}
```

When sending a message to the receiver, you may leave `this` out; by default, the message will be sent to the current receiver:

```
this.foo (a,y,z)
foo (x,y,z)            // equivalent
```

If you wish to invoke an overridden method, you must send the message to `super` instead of `this`. Using `super` is commonly done to invoke the overridden method from the method that does the overriding, but we can also invoke other methods (as is shown with `bar()` below) from the superclass, without getting the version of bar() in this class.

```
class Student extends Person {
  ...
  void foo () {         // overrides the inherited version
    ...
    foo ();             // invoke this method, recursively
    ...
    super.foo ();       // invoke the overridden version
    ...
    super.bar ();
    ...
  }
  ...
}
```

Often the first action of a method is to invoke the method it is overriding. This might be done when the subclass method is adding functionality to the superclass method.

```
void foo () {
  super.foo ();
  ...
}
```

Invoking Static Methods

The invocation of static methods is different from the invocation of normal methods, since a receiver object is not required. A static method may be invoked in either of two ways.

In the first way, the class is named directly. The syntax suggests that the message is being sent to the class itself. For example, assume that `getCount()` is a static method in class `Person`; we can write:

```
int i = Person.getCount();
```

In the second syntax, a receiver object is specified, but the receiver is not used.

```
Person p =  ...;
int i = p.getCount();
```

Here, the object referenced to by `p` is not available within the static method. It is used only to indicate that a static method from `Person` is to be invoked. The variable `p` may in fact be `null` at the time the static method is invoked; this is fine and `getCount()` will be called anyway.

On the other hand, the variable `p` may in fact refer to an object whose class is a subclass of `Person`. For example, `p` may refer to a `Student` object. For static methods, Java uses "static binding." In other words, it ignores the run-time class of the receiver object and uses only the compile-time type of the receiver expression in determining the method to be invoked. Even if the `Student` class also has a static `getCount()` method (which is different from the other `getCount()` method), the `getCount()` method from `Person` will always be invoked here.

For this reason, we say that static methods are never overridden. Since the receiver object is not relevant for static methods, I prefer to invoke static methods using this syntax:

```
Person.getCount();
```

instead of this syntax:

```
p.getCount();
```

Method Overloading

The terms "overloading" and "overriding" sound similar but mean different things. This section discusses overloading and the next section discusses overriding.

If two methods have the same name but have a different number of parameters or have different types on their parameters, then we say that this method is overloaded. In this example, "foo" is overloaded.

```
class Person {
  ...
  void foo () {
    ...
  }
  void foo (int i) {
    ...
  }
  void foo (String s) {
    ...
  }
  ...
}
```

In some sense, overloaded methods are different methods altogether. Yet, in another sense, these methods are all related. (In other languages like Smalltalk, the name is all that matters: "Same name, same method. Different name, different method." Since overloading can make Java programs difficult to understand, I feel it should be avoided.)

Differences in only the number of thrown exceptions or in the type of the returned value are not sufficient to distinguish between two methods and cause overloading. You may have several methods with the same name in a single class as long as they differ in the number or types of parameters. The compiler will complain if two methods in the same class have the same name, the same number of parameters, and the same types for their parameters.

When the compiler processes a message-sending expression involving overloaded method names, it must determine which method to invoke and it does this based only on the method name and the number and types of the arguments in the sending expression. It ignores the returned types and throw clauses.

When method overloading is present, here is how the compiler determines which method is meant in a message-sending expression. First, the compiler finds all the methods that have (1) have the same name, (2) have the same number of arguments, and (3) have parameters types that are assignment-compatible with the arguments in the message-sending expression. Second, if there is a set of more than one method selected in this way, then the compiler determines if any method in this set has more restrictive types on its arguments than another method in the set and, if so, we eliminate the method with the more restrictive parameter types. Finally, the remaining set must have a single method or the compiler will complain that the message-sending expression is ambiguous.

Method Overriding

When a subclass provides a method with the same name and the same number and types of parameters as a method in a superclass, the superclass's method is overridden. In other words, to override a method, the new method must have the same name, the same number of parameters and the exact same types on each of the parameters.

When overriding a method, the returned types must be identical. If they are not, the compiler will complain.

Also, the exceptions listed in the `throws` clause of the subclass's method header must be "narrower" than the superclass's method. In other words, the overriding method may throw fewer exceptions, but not more, than the overridden method. This means that every exception listed in the subclass method must be the same as (or a subtype of) an exception listed in the superclass's method header.

When a method is overridden in a subclass, the subclass method may specify the same or a different level of access visibility. If different, the subclass method must specify a greater access visibility, not a lesser visibility.

The general principle behind the above rules is that the overriding method must continue to respect the overridden method's contract. In other words, the overriding method must be useable in any context that the overridden method was useable. The new (overriding) method cannot throw some new, unexpected exception. It cannot suddenly become invisible (and hence un-useable) where it was okay to use the overridden method.

Overriding Fields in Subclasses

A subclass may contain a field with the same name as a field in a superclass. In such case, the superclass field is hidden by the subclass field.

The type of the superclass field does not need to be related to the type of the field in the subclass. In other words, the subclass field need not have as its type a subtype or supertype of the field in the superclass.

For example, assume that a field called `name` is declared in class `Person` and re-declared in subclass `Student`. Thus, every `Student` object will have two `name` fields. Within `Student`'s code, you may access the hidden `name` field of the receiver in either of two ways.

```
super.name
Person.name
```

You may also access the hidden `name` field of other objects as long as the access visibility of the hidden field permits it.

```
x.name
```

In such an example, will this access the `Student name` field or the hidden `Person name` field? The static type of the object expression (which the compiler knows) determines whether you get the field from the superclass or the field from the subclass. The dynamic class (i.e., the class of the object pointed to at run-time, which the compiler cannot determine) is not used!

```
Person p = ...;
Student s = ...;
...
p.name   // Gets the field from the superclass, regardless
         //      of what p points to at run-time.
s.name   // Gets the field from the subclass.
```

It is probably a good programming practice to completely avoid hiding (i.e., overriding) inherited fields.

Final Methods and Final Classes

A method may be marked `final`, in which case it may not be overridden in a subclass. If a subclass attempts to override the method, the compiler will complain.

```
final void foo (...) {
  ...
}
```

A class may be marked `final`, in which case it may not be extended. In other words, a `final` class may not have subclasses. All methods in such a class are implicitly `final`.

```
final class MyClass {
  ...
}
```

Here are some reasons for marking a method `final`:

- For security reasons. Normally a class's behavior can be changed in a subclass by overriding methods, but you may not want the behavior of a critical class to be changed by other programmers.
- If a method has been marked `final`, the compiler may be able to make certain optimizations. Normally, dynamic-binding must be used since the compiler cannot (in general) determine which method will be invoked by a message-sending expression. But when a method has been marked `final`, there will be no other implementations. This may allow the compiler to replace the dynamic method lookup and binding by a simple "procedure call," which will execute faster.
- Classes with `final` methods may be more understandable. Object-oriented programs are sometimes difficult to read. When you see a traditional "call statement" you know which routine will be invoked, so you can understand the expression that does the call by looking up and understanding the routine in question. However in an object-oriented language, an expression that sends the "foo" message may invoke one of many "foo" methods. Simply locating and understanding a "foo" method may not be sufficient since there may be another "foo" method that you are unaware of. However, seeing that the "foo" method was marked as `final` guarantees that there will be no additional "foo" methods overriding it.

On the other hand, the power of object-oriented programming is that code in a pre-existing class can be re-used by extending the class. If the pre-existing class doesn't do exactly what you want, you may create a subclass and override several of the methods to change their behavior, while leaving other methods as is. Marking a class or a method `final` makes this sort of code re-use impossible.

Anonymous Classes

A normal class has a name that is given in its class definition:

```
class MyClass implements MyInterface {
  ...
}
```

An instance of the class may be created with a new expression:

```
MyInterface x;
x = new MyClass (...);
```

With "anonymous classes," the entire class definition occurs at the point the new instance is created:

```
x = new MyInterface { ... fields and methods ...};
```

In this form the name of an interface follows the keyword `new`. This means we are defining a class that will implement the interface called `MyInterface`.

In the second form the name of a class follows the keyword `new`. This means we are creating a class that will extend the class called `MySuperClass`.

```
x = new MySuperClass { ... fields and methods ...};
```

In either case, `MyInterface` or `MySuperClass` will define the new object's interface. That is, instances of the anonymous class will understand the messages given by `MyInterface` or `MySuperClass`, whichever was used.

Since an anonymous class has no name, it will have no constructors of its own. However, the `new` expression may invoke the constructors from the superclass. Assuming that `MySuperClass` has a three-argument constructor, this expression will invoke it:

```
x = new MySuperClass (a,b,c) { ... fields and methods ...};
```

The Java book from Sun says "You should probably avoid anonymous classes longer than about six lines" and [anonymous classes can] "create impenetrable inscrutability." [quoted from "The Java Programming Language"] Since it is easy to make up new class names, one wonders whether anonymous classes should ever be used or whether they should even be in the language at all.

The "main" Method

A Java program should contain a "main" method. When the virtual machine is invoked on some class, the `main` method from that class will be executed. This method should be

```
public static void
```

and should take a single parameter with type `String[]` as shown below:

```
class Echo {
  public static void main (String[] args) {
    System.out.println("Welcome!");
    for (int i = 0; i<args.length; i++) {
      System.out.print (args[i] + " ");
    }
```

```
        System.out.println();
    }
}
```

This program will print out its arguments, not including the program name. It may be run with the following command line sequence ("%" is a Unix prompt and user input is in boldface). The "java" command invokes the bytecode interpreter, which is often called the "virtual machine."

```
% java Echo hello there folks
Welcome!
hello there folks
% ...
```

A program may consist of several classes, each with a main method. When the virtual machine is invoked, it is invoked on a particular class, which determines which main method is executed.

Methods in Class "Object"

Object identity (often called "object equality") is tested with the following two relational operators.

```
if (x == y) ...;
if (x != y) ...;
```

They simply compare pointers to determine if the object referred to by "x" is the same object as the object referred to by "y," and return either true or false. We can summarize these methods using the following shorthand:

```
x == y ➔ boolean
x != y ➔ boolean
```

Below are several other important methods that are understood by all objects. We summarize their interfaces (i.e., their parameters, their return values, and their exceptions) as follows:

```
x.equals (Object obj) ➔ boolean
x.hashCode ()➔ int
x.clone ()➔ Object (throws CloneNotSupportedException)
x.getClass ()➔ Class
x.finalize ()➔ void (throws Throwable)
x.toString ()➔ String
```

The default implementation of equals() simply tests for object identity (==); you may override it in subclasses, when two distinct objects in some sense represent the same "thing". For example, you may have multiple representations of polynomials; you might override equals() with a method that will perform algebraic manipulations to determine if they represent the same function.

Each object must be able to compute a "hash-code" so that it can be stored in various data structures. The method hashCode() may be sent to any object; it will return a pseudo-random

integer which can be used as an index into a hash-based look-up table. The `hashCode()` function should be overridden whenever `equals()` is overridden so that two "equal" objects always have the same hash-code. The default implementation may be used if `equals()` is not overridden.

The default implementation of the `clone()` method makes a shallow copy of the object and returns it. (A "shallow" copy involves allocating a single new object of the same size as the receiver and copying the fields of the receiver into the new object. Pointers are copied, but the objects they point to are not themselves copied.)

You may override the default implementation of `clone()` to (perhaps) recursively copy the objects that are pointed to. On the other hand, if an object should never be copied, you may override `clone()` to throw `CloneNotSupportedException`. (There is also an interface called `Cloneable`, which has zero methods; it is a "marker" interface. Your interfaces and classes may implement or not implement the `Cloneable` interface.) The method `Object.clone()` is declared to be `protected`; when you override it, you may make it `public` so that the object can be cloned from outside the class. One possible implementation is:

```
class MyClass ... implements Cloneable {
  ...
  public Object clone() throws CloneNotSupportedException {
    return super.clone();
  }
  ...
}
```

Note that `clone()` returns an `Object`; therefore its result must be cast.

```
MyClass x, y ...;
...
y = (MyClass) x.clone();
```

Variables of Type Object

Note that when used as a type, `Object` means that the variable may point to any kind of object, including any kind of array. However, a variable of type `Object` may not contain a primitive data value (e.g., an `int`, `float`, `boolean`, etc.)

Casting Object References

Here is an example of a casting expression. Assume that `MySupertype` and `MySubtype` are related in the type hierarchy.

```
MySupertype x;
MySubtype y;
...
y = (MySubtype) x;
```

Casting an object reference will insert code to perform a runtime type check.

Because of x's declaration in this example, we (and the compiler) know that, at runtime, x must point to an object of type MySupertype or one of its subtypes, such as MySubtype. In this example, the programmer "believes" that directly before the assignment to y, x will point to an object of type MySubtype (or one of its subtypes). In order to use it as a MySubtype, he must first cast it down to that type.

If, at runtime, the programmer turns out to be wrong and "x" does not point to an object of type MySubtype (or one of its subtypes), then the ClassCastException will be thrown.

The "null" Pointer

The keyword null refers to a pointer to no object. "null" can be stored in variables, fields, array elements, parameters, and so on. Whenever a message is sent, a field is accessed, or an array element is accessed,

```
x.foo()
x.field
x[i]
```

it is possible that the object reference x is null. If so, the NullPointerException is thrown. (However, if a static method is invoked using the "x.foo()" syntax, then it is okay for the object reference x to be null since the binding is done statically and x's value is never used.)

In some OOP languages like Smalltalk, "null" or "nil" refers to a special object. In Java, null does not point to any object. It is sufficient to imagine that null is implemented with a zero pointer and that no object is ever stored at address 0.

"Static Final" Constants

A constant may be defined and named as in the following example:

```
class MyClass {
  static final double pi = 3.14159;
  ...
}
```

The keyword final means that the value of the field being defined will not be changed once set. The keyword static means that field being defined is not member of an instance, but is related to the class as a whole.

Constants may also be placed in interfaces, in which case the keywords static and final are not used. Any and all fields appearing in interfaces are assumed to be constants.

```
interface MyInterface {
 double pi = 3.14159;
  ...
}
```

A constant defined in a class may be used within that class by simply naming it; anywhere else, the constant may be referenced by prefixing its name with the class or interface it is a part of.

```
area = dia * pi;            // within MyClass
area = dia * MyClass.pi;    // anywhere else
```

Abstract Methods and Classes

Normally, each method in a class will include a body of code. However, you may avoid providing an implementation for some methods in the class. In this case, the method is said to be "abstract" and the `abstract` keyword is used.

If a class contains any abstract methods, then the class heading must also contain the `abstract` keyword. Here is an example:

```
abstract class MyAbstractClass {
  ...
  public void foo () { ... }   // Normal method
  abstract void bar ();        // Abstract method
  ...
}
```

An abstract class is incomplete since it doesn't have an implementation for some of its methods. Therefore, you may not create instances of an abstract class and the compiler will complain if you try to. Consider this example:

```
MyAbstractClass x;
x = new MyAbstractClass ();    // Compiler error here
...
x.foo ();                      //   ...this method doesn't exist.
```

Normally, an abstract class will have subclasses, which will provide normal methods for any abstract methods that were inherited. The `abstract` keyword indicates that we expect to find a subclass that completes the implementation. The abstract class provides a partial implementation; the subclass completes it.

For example, assume we have a subclass which provides an implementation for the abstract methods and inherits other non-abstract methods. The subclass would not be marked as abstract.

```
class MySub extends MyAbstractClass {
  ...
  // foo is inherited
  void bar () { ... }          // An implementation is given for bar
  ...
}
```

Note that we can have variables whose type is the abstract class, as in this code:

```
MyAbstractClass x;
...
x = new MySub ();
...
x.foo ();                        // An implementation will exist
```

A class may be marked `abstract`, even though none of its methods are marked `abstract`. This might be used for a class that is intended to provide code for superclasses, but which has no intrinsic meaning on its own.

A subclass that extends some superclass may override a normal method, making it abstract. Perhaps the superclass was not abstract; in any case, the subclass will be abstract since it now contains unimplemented methods.

```
class MySuper {
  ...
  public void bar () { ... }
  ...
}

abstract class MySub extends MySuper {
  ...
  abstract void bar ();
  ...
}
```

We assume that there will be a further subclass of `MySub`, providing method implementations for the abstract methods it inherits (like `bar`).

Throwing Exceptions

Errors may arise at runtime. Errors that are handled by the Java language are called "exceptions." Here are some pre-defined exceptions, which were mentioned earlier:

```
ArrayIndexOutOfBoundsException
NullPointerException
ClassCastException
```

When an exception occurs, it is said to be "thrown." Exceptions may be thrown implicitly, by executing code that does something bad. For example, this code might throw the `ArrayIndexOutOfBoundsException`:

```
x = a[i+1];
```

Exceptions may also be thrown explicitly by using the `throw` statement. In this example, the programmer has defined a new kind of error called `MyExcept`.

```
throw new MyExcept ();
```

Exceptions are modeled with classes and objects. There is a class for every sort of exception. There are pre-existing classes with names like:

```
ArrayIndexOutOfBoundsException
NullPointerException
...
```

If you wish to create a new sort of exception like `MyExcept`, you must create a class called `MyExcept`. It should extend the predefined class called `Exception`.

```
class MyExcept extends Exception {
  ...
}
```

You may write code that catches any exceptions that occur, using the `try` statement.

```
try {
  statements
} catch (SomeException e) {
  statements
} catch (MyExcept e) {
  statements
} catch (YetAnotherException e) {
  statements
...
} finally {
  statements
}
```

The `try` statement has a block of statements which follows the `try` keyword. (We will call these the "body statements.") Then, the `try` statement has zero or more `catch` clauses, followed by an (optional) `finally` clause, with its statements.

The body statements are executed first. If no exceptions are thrown during their execution, or during the execution of any methods they invoke, then none of the `catch` clauses are executed. If an exception is thrown during the execution of the body statements, then the corresponding `catch` clause will be executed.

In more detail, the process of throwing and catching an exception works like this. First, an instance of the exception class is created. If the exception was caused explicitly by a `throw` statement, then the `throw` statement will create the new object. The `throw` statement is followed by an expression, which would normally be a `new` clause, which invokes a constructor for the exception class. In our example,

```
throw new MyExcept ();
```

we create a new instance of `MyExcept`, using the "no argument" constructor.

If the exception arises implicitly, then the virtual machine constructs the object.

Next, the virtual machine looks for a matching `catch` clause. The clauses are searched in order and the first matching `catch` clause will be selected. The `catch` clause has a single parameter. In this example, it is "e". This parameter is set to the exception object and the corresponding statements are then executed.

```
} catch (MyExcept e) {
  statements...           // may use "e" here
}
```

We can pass information from the `throw` statement to the corresponding `catch` statements by passing it in the exception object. Imagine that we wish to pass a `String` to indicate the severity of the problem. We can add a field called `severity` to `MyExcept` and then invoke a constructor which will set that field.

```
class MyExcept extends Exception {
  String severity;
  MyExcept (String s) { severity = s; }
  ...
}


try {
  ...
  throw new MyExcept ("Mission-Critical");
  ...
} catch (MyExcept e) {
  ... use e.severity here ...
}
```

Exception classes can be related to one another in the subclass / superclass hierarchy. For example, we might have made `MyExcept` a subclass of some other exception class. The type in the `catch` clause does not need to exactly match the class of the thrown exception. The `catch` clauses are tried in order they appear and the first matching `catch` is selected.

(If one `catch` clause in the `try` statement is a supertype of some subsequent `catch` clause in the same `try` statement, then the first `catch` clause will always be selected before the second `catch` clause. This is considered an error and the compiler will catch it: No `catch` clause may have a type that is a subtype of a `catch` clause listed earlier in the same `try` statement.)

`Try` statements may be nested. If there is no matching `catch` clause in the innermost `try` statement, the next most inner `try` statement will be searched.

```
try {
  ...
    try {
```

```
        ...
        throw ...;
        ...
      } catch (...) {
        statements
      } catch (...) {
        statements
      ...
      }
    ...
  } catch (...) {
    statements
  } catch (...) {
    statements
  ...
  }
```

Various `try` statements are "entered" and "exited from" as the program runs. Thus, the nesting we just discussed is actually dynamic, not lexical. Here is another example, in which method `bar()` calls method `foo()` from within a `try` statement.

```
bar () {
  ...
    try {
      ...
      x.foo ();
      ...
    } catch (...) {
      statements
    } catch (...) {
      statements
    ...
    }
  ...
}
```

Suppose method `foo()` contains another `try` statement whose body may throw an exception.

```
foo () {
  ...
    try {
      ...
      throw ...;
      ...
    } catch (...) {
      statements
    } catch (...) {
      statements
    ...
    }
  ...
}
```

When the `throw` statement is executed, the virtual machine will first look for a matching `catch` clause in the `try` statement in `foo()`. If none is found, it will then search the `catch` clauses in the `bar()` method. So which `catch` clauses are searched depends on the order of execution. If method `foo()` had been called from some other method instead of `bar()`, then `bar()`'s `catch` clauses would not be searched.

The body of a `try` statement will be exited as the result of either:

- normal completion (i.e., execution falls out the bottom of the "body statements")
- a `return` statement is executed
- an exception is thrown

If an exception is thrown during the body and the `try` statement catches it, then the corresponding `catch` statements will be executed. The `catch` statements will then end as a result of either:

- normal completion (i.e., execution falls out the bottom of the `catch` statements)
- a `return` statement is executed
- another exception is thrown within the `catch` statements

If an exception is thrown during the execution of a `catch` clause, then this new exception will be propagated outside the `try` statement. In other words, the `try` statement will be exited so the same set of `catch` clauses will not be searched for the new exception.

A `try` statement may contain an optional `finally` clause, which includes several statements, which we will call the "`finally` statements." The `finally` statements will always be executed before execution leaves the `try` statement, regardless of whether exceptions were thrown and regardless of whether a matching `catch` clause was found and executed.

The `finally` statements will be entered after either:

- normal completion of the body or a `catch` clause
- the execution of a `return` statement in the body or a `catch` clause
- the result of an uncaught exception in the body
- the result of a exception getting thrown in some `catch` clause

The virtual machine remembers which. If there is an uncaught exception, it is said to be "pending." If there was a `return`, then the `return` is pending. Otherwise, a normal completion is pending.

The `finally` statements may end with either:

- normal completion (i.e., execution falls out the bottom of the `finally` statements)
- a `return` statement is executed in the `finally` statements

- another exception is thrown within the `finally` statements

If the `finally` statements execute a `return` statement or throw some new exception, then that will take precedence over whatever sort of exit was pending. In other words, if the `finally` statements were entered with a `return` pending or an uncaught exception pending, then it will be ignored and forgotten.

Otherwise, if the `finally` statements complete normally, whatever sort of exit is pending will then happen. In particular, if there is an uncaught exception pending (from the body or one of the `catch` clauses), that exception will be propagated to the code surrounding the `try` statement. If there is a pending `return`, then the `return` will occur. If there is a normal completion pending, then execution will continue after the `try` statement.

All exception classes are arranged in the subclass-superclass hierarchy and each is a subclasses of the class named `Throwable`. Here is the class hierarchy, showing the predefined exception classes. User-defined exceptions would be made subclasses of the class named `Exception`.

```
Object
  ...
  Throwable
    Exception
      <User Defined Exception #1>
      <User Defined Exception #2>
      <User Defined Exception #3>
      ...
      RuntimeException
        ArithmeticException
        ArrayStoreException
        ClassCastException
        ConcurrentModificationException
        EmptyStackException
        IllegalArgumentException
          IllegalThreadStateException
          NumberFormatException
        IllegalMonitorStateException
        IllegalStateException
        IndexOutOfBoundsException
          ArrayIndexOutOfBoundsException
          StringIndexOutOfBoundsException
        MissingResourceException
        NegativeArraySizeException
        NoSuchElementException
        NullPointerException
        SecurityException
        UndeclaredThrowableException
        UnsupportedOperationException
    Error
      LinkageError
        ClassCircularityError
        ClassFormatError
          UnsupportedClassVersionError
        ExceptionInitializerError
```

```
    IncompatibleClassChangeError
      AbstractMethodError
      IllegalAccesError
      InstantiationError
      NoSuchFieldError
      NoSuchMethodError
    NoClassDefFoundError
    UnsatisfiedLinkError
    VerifyError
  ThreadDeath
  VirtualMachineError
    InternalError
    OutOfMemoryError
    StackOverflowError
    UnknownError
```

Contracts and Exceptions

When a programmer writes a method, he should be aware of the "contract" the method has with the users of the method. This contract is the interface between the user code (i.e., the caller) and the method. This contract will include aspects that are checked by the compiler (like the number and types of the arguments and return values) and semantic, logical aspects that are not checked by the compiler (like the fact that "Argument n will never be negative" or that "The returned value will be such-and-such"). Each method has a contract, which should be clearly documented.

For example, consider a method (called "`search`") that searches some data structure for a specified element, such as a dictionary mapping strings to integers. Assume this method is passed the key to search for. For example, the method might be passed a `String`. The contract states that the value returned will be the `int` that was previously entered into the data structure with the given key.

What happens if the search element does not exist -- this can be considered an error, of sorts. Is it the responsibility of the user to ensure that the method will never be called to find an element that doesn't exist, or is it the responsibility of the method to deal correctly when asked to search for something that does not exist? The contract documentation should discuss how this error is handled. An unclear decision about who is responsible for handling extreme and unusual cases is the source of many subtle program bugs.

In Java, exceptions can be used to deal with errors and other extreme conditions, as in the above example. Sometimes, it makes sense to return a special value to indicate an unusual result. For example, we might choose to return –1 to indicate the search key was not found. But in other cases, it might be better for the method to throw an exception. In our example, a reasonable design is to code the `search` method so that it will throw an exception if asked to find an element that does not exist. The `search` method itself will not catch the exception; instead it will be passed up to the calling code.

The fact that the method may throw an exception is clearly part of its contract. The communication between the calling code and the method consists of more than just arguments

and returned values. A method may also indicate its result (or complete its computation) by throwing an exception.

In most languages, the programmer must specify what types of values are passed to a method and what types of values will be returned. In Java, the programmer must also specify which exceptions may be thrown by the method.

When we say "an exception is thrown by method `foo`," we mean that some code in `foo` will throw an exception, but that the exception will not be caught within `foo`. Instead, it will be propagated up to the caller's code. It could also be that case that the exception is thrown in some method `foo` calls, but neither the called method nor `foo` catches the exception. Again, it propagates up to the code that called `foo`.

If a method like `foo` may throw an exception but not catch it, then the method header for `foo` must specify that `foo` may throw the exception. The syntax for method `foo` makes explicit which exceptions may be thrown in `foo` and propagated to the calling code:

```
public void foo (...)
       throws MyExcept, AnotherExcept, YetAnotherException
  {
    ...
    throw new myExcept (...);
    ...
    anotherMethod (...);
    ...
  }
```

By looking at the method header, the caller of `foo` will see that `foo` may throw any one of the three exceptions listed in the header. Therefore, the calling code must either catch these exceptions or propagate them even further. Assume a method called "`bar`" calls `foo`, as shown below. The `bar` method handles two of the exceptions (`MyExcept` and `AnotherExcept`), but does not handle the third exception, which it propagates. Therefore, `bar` must list the third exception in its header.

```
public void bar (...)
       throws YetAnotherException
  {
    ...
    try {
      ... foo (...) ...
    } catch (MyExcept e) {
      ...
    } catch (AnotherExcept e) {
      ...
    }
  }
```

The Java compiler checks all this: it ensures that a method cannot throw exceptions that are not listed in is header. If a method may throw (without catching) any exception from any subclass of `Exception`, then the exception must be listed in the method header.

Notice that the hierarchy of exceptions begins with `Throwable`. One major subdivision is `Exception`. Many of them are predefined and these are grouped under `RuntimeException`. `RuntimeExceptions` relate to rather common, run-of-the-mill problems, such as:

```
ArrayIndexOutOfBoundsException
NullPointerException
NoSuchElementException
```

User-defined exceptions would also be added under `Exception`.

The other major subdivision of `Throwable` is `Error`. These are more serious and indicate system failure. Examples are:

```
InternalError
OutOfMemoryError
StackOverflowError
```

There are a number of predefined errors and exceptions, which are quite common and can occur in virtually any code. The burden of listing them all in every method header would be unbearable. Therefore, you do not need to list the predefined exceptions in method headers. More precisely, any subclass of `RuntimeException` or `Error` does not need to be listed in method headers; these are implicitly assumed. Instead, you only need to list those exceptions outside of these classes. This allows beginning Java programmers to pretty much ignore the exception handling statements (`throw` and `try-catch-finally`) altogether, if they choose.

Initialization Blocks

A class may contain an "initialization block." If so, it will be executed as if it were placed at the beginning of every constructor in the class. Thus, it is executed once every time an object is created.

```
class MyClass {
  ...
  foo () { ... statements ... }
  ...
  { ... statements... }
  ...
}
```

Below is an example that keeps track of how many instances are created. There are two constructors. We could put the code to increment `personCount` in each constructor, but instead we factor it into an initialization block.

```
class Person {
  static int personCount = 0;
  Person () { ... statements ... }
  Person (String name) { ... statements ... }
  { personCount++; }
  ...
}
```

There may be multiple initialization blocks. If so, they are executed in the order they are listed in the class.

Static initialization blocks

A class may contain a "static initialization block," which is syntactically similar to an "initialization block," except the `static` keyword is used:

```
class MyClass {
  ...
  static { ... statements... }
  ...
}
```

The static initialization block is executed when the class is first loaded. This may be at program startup time or may be at some later time, when the class is first needed.

A class may have several static initialization blocks.

When a class is loaded, its static fields are set to their default values. Then the static fields and the static initialization blocks are executed in the order they appear in the source code. In other words, consider this example:

```
class MyClass {
  ...
  static { ... statements-1... }
  ...
  static int x = ...;
  ...
  static { ... statements-2... }
  ...
  static int y = ...;
  ...
  static { ... statements-3... }
  ...
}
```

The execution order would be:

```
statements-1
x's initialization expression
statements-2
y's initialization expression
statements-3
```

The static initialization block may only refer to static class fields and may not throw any uncaught exceptions.

Wrapper Classes

Each primitive class (like `int`, `boolean`, `double`, etc.) has a corresponding class, called a "wrapper class." The wrapper classes are:

```
Number
  Byte
  Short
  Integer
  Long
  Float
  Double
Boolean
Character
Void
```

Wrapper classes serve two functions.

First, they act as a place to put constants and methods related to the primitive type. Examples include a method to test a `double` value against infinity, and a method to return the largest `long` integer value.

Second, wrapper classes can be used to encapsulate values of a primitive type. Imagine that you have a `Set` class that can hold sets of Objects. The `Set` class is somewhat general, since it can hold any type of `Object`. But it could not hold an `int` value, since that is a primitive value, not an `Object`. To create a set of `int`s, you can first put each `int` into a wrapper object.

To create a wrapper object, you must supply a primitive value.

```
Integer i = new Integer (123);
...
Object x = i;
```

To get the value out of a wrapper object you may use methods like this:

```
int j = i.intValue ();
```

Of course there are similar methods for the other primitive types.

```
new Byte (byte) ➔ Byte
new Short (short) ➔ Short
new Integer (int) ➔ Integer
new Long (long) ➔ Long
new Float (float) ➔ Float
new Double (double) ➔ Double
new Boolean (boolean) ➔ Boolean
```

```
new Character (char) ➔ Character

x.byteValue (Byte) ➔ byte
x.shortValue (Short) ➔ short
x.intValue (Integer) ➔ int
x.longValue (Long) ➔ long
x.floatValue (Float) ➔ float
x.doubleValue (Double) ➔ double
x.booleanValue (Boolean) ➔ boolean
x.charValue (Character) ➔ char
```

Note the inconsistency in the spellings:

```
char   Character
int    Integer
```

There is also a `Void` wrapper class, but it has no instances.

Each wrapper object is immutable. Once created, it may not be changed.

Packages

Classes and interfaces are grouped together into "packages." Every class and interface will belong to some package. Typically, the parts of a package will be related. They will be developed together and will be released and delivered as a unit.

As of this writing, the Sun "Java 2 Standard Edition, v 1.4" appears to contain about 137 packages, comprising 2738 classes and interfaces. A typical package might contain perhaps 5 interfaces and 15 classes.

Some important packages are:

```
java.lang    Essential classes;  always imported automatically
java.io      Basic I/O (files and character streams)
java.util    Data structure classes
java.awt     "Abstract Windowing Toolkit" (user interface classes)
java.net     Sockets, TCP/IP, and URL classes
java.applet  Running Java programs over WWW and internet browsers
```

Packages can be nested in a tree-shaped hierarchy. Each source file begins with a line like this:

```
package x.y.z;
```

The `package` statement must be the first line in a file. It will indicate which package the classes or interfaces in this file belong to. For example:

```
package com.sun.games;
```

If the `package` statement is missing, an "unnamed" package is meant. This is useful for small, single-file Java programs.

Although the dot notation suggests nesting or a hierarchy or some other relationship between packages, it is purely naming. The package named "`x.y`" cannot reference stuff in the package named "`x`" or in the package named "`x.y.z`" any differently than it can reference stuff in a package named "`a.b`". As the Sun documentation says, "Nesting can group related packages and help programmers find classes in a logical hierarchy, but it confers no other benefits." [The Java Programming Language: Third Edition.]

Any package (which we can call the "current package" for convenience), may make use of things in some other package, but we must specify the thing's name in full. We prefix the name of the item with the name of the package containing it.

The package named `java.util` contains a class called `Date`, so we could write:

```
java.util.Date d = new java.util.Date ();
```

This gets tedious. The `import` statement may be used in the current package, to make full qualifications unnecessary.

```
import java.util.Date;  // to import a single class or interface
import java.util.*;     // to import everything in the package
```

Now we can write:

```
Date d = new Date ();
```

Threads

A "thread of control" is lightweight process. They are used in multi-programmed applications. There is a class called `Thread`, which is used in creating and manipulating threads in Java.

To create a thread, you begin by creating a subclass of the `Thread` class. You should override the `run()` method. (The inherited `run()` method does nothing and returns immediately.) The method you provide will become the "main" method of that thread.

```
public class MyThread extends Thread {
  public void run () {
    ... statements ...
  }
  ...
}
```

Each running thread is represented with a `Thread` object, so you must first create a `Thread` object:

```
Thread t = new Thread ();
```

When the `Thread` object is first created, it is not initially running; you must send it the `start()` method to cause it to begin executing.

```
t.start ();
```

To get a pointer to the `Thread` object for the currently running thread, use this method:

```
Thread.currentThread () ➔ Thread
```

To put a thread to sleep for a specific duration, send this method to a thread:

```
aThread.sleep (long milliseconds) ➔ void
```

To yield the processor to any other runnable threads, use this method:

```
Thread.yield () ➔ void
```

To "join" two threads, one thread can send this message to another thread. This will wait for `aThread` to die and will then return.

```
aThread.join () ➔ void
```

Each thread has a name and a priority. These can be changed or queried with these methods:

```
aThread.setName (String name) ➔ void
aThread.setPriority (int pri) ➔ void
aThread.getName () ➔ String
aThread.getPriority () ➔ int
```

You may send the `interrupt()` message to a running thread.

Locking Objects and Classes

Java supports "monitor-style" concurrency control with locks and the `synchronized` keyword.

Every object has a lock associated with it. This lock can be used to control concurrent access to the object. Methods and statements may be marked with the `synchronized` keyword.

If some thread attempts to execute a `synchronized` method on some receiver object, the lock of that object will first be acquired, then the method will execute, then the lock will be released. If some other thread attempts to execute a `synchronized` method on the same object (either the same method or a different method) while the first thread still holds the lock, the second thread will wait until the lock is released

If a thread that has already acquired the object's lock attempts to execute another `synchronized` method on the same object, it is not a problem: there will be no waiting. Since

the thread already owns the lock, the method invocation will occur without delay. The object's lock will be released when the `synchronized` method that acquired the lock returns (or exits by throwing an uncaught exception).

A subclass method may override a `synchronized` method. The new method may be `synchronized` or not. The superclass method will still be `synchronized`, whenever it is invoked.

Each class also has a lock associated with it. If a class method (i.e., static method) is marked `synchronized`, then the class lock will be acquired for the duration of the class method.

The class lock is unrelated to the locks of the instances of the class. In other words, a (non-static) `synchronized` method does not acquire the class lock before proceeding; the class lock may be held by some other thread while the first thread executes its `synchronized` method. This allows a synchronized static method to proceed in parallel with a synchronized non-static method.

You may also acquire the lock on an object with the `synchronized` statement:

```
synchronized (expression) {
  ... statements ...
}
```

This works the same way. The expression is evaluated to yield an object. The lock for that object is acquired, waiting if necessary. Then the statements are executed and the lock is released.

All objects understand the following messages, which can be used to further control concurrency:

```
x.wait () ➔ void
x.notify () ➔ void
x.notifyAll () ➔ void
```

The `wait()` method will suspend the thread which invokes it, until some other thread executes a `notify()` or `notifyAll()` operation. The `wait()` method will both suspend the thread and release the lock (atomically). Later (after a `notify()` or `notifyAll()` has been executed), the lock will be reacquired and the waiting thread will be reawakened.

[I am not sure whether the Java specification requires (1) "Hoare semantics", (2) "Mesa semantics", or (3) does not specify either.]

A typical programming paradigm might involve two methods that work on a single object:

```
synchronized void doWhenCondition () {
  while (!condition) {
    wait ();
  }
  ... Do what must be done when condition is true ...
```

```
}

synchronized void changeCondition () {
  ... Change some value that may be used in the condition test ...
  notifyAll ();
}
```

Note that the invocation of `wait()` is in a loop. This is because some other thread may sneak in between the invocation of `notifyAll()` and the end of the `wait()`, and change the object. The waiting thread will generally need to re-check the condition before proceeding.

Strict Floating-Point Evaluations

When an expression contains floating-point arithmetic, its result can be dependent on the order in which the operations are evaluated. For example, with "real number" arithmetic, multiplication is associative, but with floating-point arithmetic (which is an approximation to real arithmetic), the result may not be the same.

```
(x * y) * z   ≠   x * (y * z)
```

Normally, the compiler may re-write certain expressions to gain efficiency, but this code may generate results that are different. In some calculations the programmer may wish to control the exact order in which operations are performed. The programmer can prevent the compiler from re-writing expressions by using the `strictfp` keyword.

The `strictfp` keyword can be applied to classes, methods, and interfaces. (Interfaces may contain expressions in the initialization of constants.)

Online Web Resources

The current version of Java is called J2SE ("Java 2 Standard Edition"), version 1.4. Sun's website is fairly large; the following websites are of interest:

`java.sun.com/j2se/1.4/docs`

Contains links to many good Java documents.

`java.sun.com/j2se/1.4/docs/api`

The Java 2 class library. This site brings up a multi-frame window for convenient browsing.

`java.sun.com/j2se/1.4/docs/api/overview-summary.html`

The same as above, but without the browsing frames. (This page may be faster and easier to use.)

`www.cs.pdx.edu/~harry/musings/JavaLang.html`

An HTML version of the document you are reading.

`www.cs.pdx.edu/~harry/musings/JavaLanguage.doc`

An MS-Word version of the document you are reading.

Please email any corrections to the author at:

`harry@cs.pdx.edu`

I really appreciate your feedback on this document and I thank you in advance for any typos, mistakes, errors, confusing wordings, ambiguous phrasings, sloppy writing, imprecise thinking, or just plain bad writing, that you might bring to my attention!