Java Basics

Topics in this section include:

- What makes Java programs portable, secure, and robust
- The structure of Java applets and applications
- How Java applications are executed
- How applets are invoked and executed
- The Java Language, Part I
- Comments
- Declarations
- Expressions
- Statements
- Garbage collection
- Java Semantics

Portability

Java programs are portable across operating systems and hardware environments. Portability is to your advantage because:

- You need only one version of your software to serve a broad market.
- The Internet, in effect, becomes one giant, dynamic library.
- You are no longer limited by your particular computer platform.

Three features make Java `String` programs portable:

1. **The language**. The Java language is completely specified; all data-type sizes and formats are defined as part of the language. By contrast, C/C++ leaves these "details" up to the compiler implementor, and many C/C++ programs therefore

are not portable.

2. **The library**. The Java class library is available on any machine with a Java runtime system, because a portable program is of no use if you cannot use the same class library on every platform. Window-manager function calls in a Mac application written in C/C++, for example, do not port well to a PC.

3. **The byte code**. The Java runtime system does not compile your source code directly into machine language, an inflexible and nonportable representation of your program. Instead, Java programs are translated into machine-independent byte code. The byte code is easily interpreted and therefore can be executed on any platform having a Java runtime system. (The latest versions of the Netscape Navigator browser, for example, can run applets on virtually any platform).

Security

The Java language is secure in that it is very difficult to write incorrect code or viruses that can corrupt/steal your data, or harm hardware such as hard disks. There are two main lines of defense:

- Interpreter level:
- No pointer arithmetic
- Garbage collection
- Array bounds checking
- No illegal data conversions
- Browser level (applies to applets only):
- No local file I/O
- Sockets back to host only
- No calls to native methods

Robustness

The Java language is robust. It has several features designed to avoid crashes during program execution, including:

- No pointer arithmetic

- Garbage collection--no bad addresses
- Array and string bounds checking
- No jumping to bad method addresses
- Interfaces and exceptions

Java Program Structure

A file containing Java source code is considered a compilation unit. Such a compilation unit contains a set of classes and, optionally, a package definition to group related classes together. Classes contain data and method members that specify the state and behavior of the objects in your program.

Java programs come in two flavors:

- Standalone applications that have no initial context such as a pre-existing main window
- Applets for WWW programming

The major differences between applications and applets are:

- Applets are not allowed to use file I/O and sockets (other than to the host platform). Applications do not have these restrictions.
- An applet must be a subclass of the Java `Applet` class. Aplications do not need to subclass any particular class.
- Unlike applets, applications can have menus.
- Unlike applications, applets need to respond to predefined lifecycle messages from the WWW browser in which they're running.

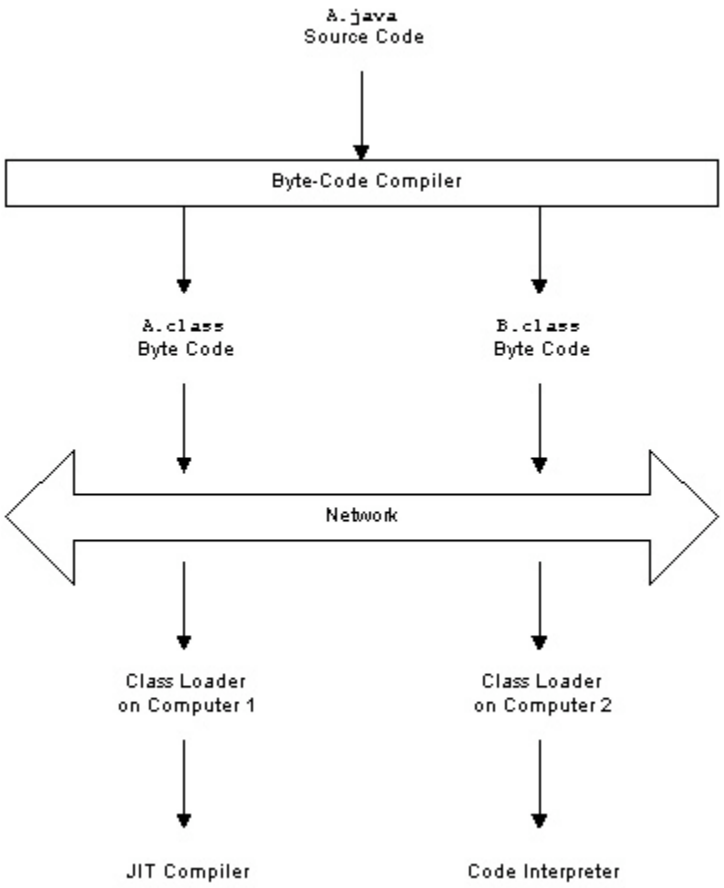
Java Program Execution

The Java byte-code compiler translates a Java source file into machine-independent byte code. The byte code for each publicly visible class is placed in a separate file, so that the Java runtime system can easily find it. If your program instantiates an object of class `A`, for example, the class loader searches the directories listed in your `CLASSPATH` environment variable for a file called `A.class` that contains the class definition and byte code for class `A`.

There is no link phase for Java programs; all linking is done dynamically at

runtime.

The following diagram shows an example of the Java compilation and execution sequence for a source file named `A.java` containing public class `A` and non-public class `B`:



Java programs are, in effect, distributed applications. You may think of them as a collection of DLLs (dynamically loadable libraries) that are linked on demand at runtime. When you write your own Java applications, you will often integrate your program with already-existing portions of code that reside on other machines.

A Simple Application

Consider the following trivial application that prints `"hi there"` to standard output:

```
public class TrivialApplication {
  // args[0] is first argument
  // args[1] the second
  public static void main(String args[]) {
    System.out.println("hi there");
  }
}
```

The command `java TrivialApplication` tells the Java runtime system to begin with the class file `TrivialApplication.class` and to look in that file for a method with the signature:

```
public static void main(String args[]);
```

The `main()` method will always reside in one of your class files. The Java language does not allow methods outside of class definitions. The class, in effect, creates scoped symbol `StartingClassName.main` for your `main()` method.

Applet Execution

An applet is a Java program that runs within a Java-compatible WWW browser or in an appletviewer. To execute your applet, the browser:

- Creates an instance of your applet
- Sends messages to your applet to automatically invoke predefined lifecycle methods

The predefined methods automatically invoked by the runtime system are:

- `init()`. This method takes the place of the `Applet` constructor and is only called once during applet creation. Instance variables should be initialized in this method. GUI components such as buttons and scrollbars should be added to the GUI in this method.
- `start()`. This method is called once after `init()` and whenever your applet is revisited by your browser, or when you deiconify your browser. This method should be used to start animations and other threads.
- `paint(Graphics g)`. This method is called when the applet drawing area needs to be redrawn. Anything not drawn by contained components must be drawn in this method. Bitmaps, for example, are drawn here, but buttons are not because they handle their own painting.
- `stop()`. This method is called when you leave an applet or when you iconify your browser. The method should be used to suspend animations and other

threads so they do not burden system resources unnecessarily. It is guaranteed to be called before `destroy()`.

- `destroy()`. This method is called when an applet terminates, for example, when quitting the browser. Final clean-up operations such as freeing up system resources with `dispose()` should be done here. The `dispose()` method of `Frame` removes the menu bar. Therefore, do not forget to call `super.dispose()` if you override the default behavior.

  The basic structure of an applet that uses each of these predefined methods is:

```
import java.applet.Applet;
// include all AWT class definitions
import java.awt.*;

public class AppletTemplate extends Applet {
  public void init() {
    // create GUI, initialize applet
  }
  public void start() {
    // start threads, animations etc...
  }
  public void paint(Graphics g) {
    // draw things in g
  }
  public void stop() {
    // suspend threads, stop animations etc...
  }
  public void destroy() {
    // free up system resources, stop threads
  }
}
```

  All you have to do is fill in the appropriate methods to bring your applet to life. If you don't need to use one or more of these predefined methods, simply leave them out of your applet. The applet will ignore messages from the browser attempting to invoke any of these methods that you don't use.

A Simple Applet

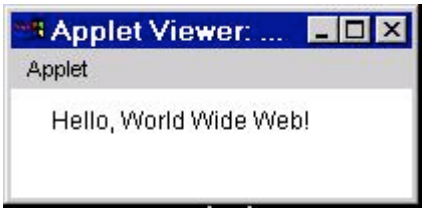
The following complete applet displays `"Hello, World Wide Web!"` in your browser window:

```
import java.applet.Applet;
import java.awt.Graphics;

public class TrivialApplet extends Applet {
  public void paint(Graphics g) {
    // display a string at 20,20
```

```
    // where 0,0 is the upper-left corner
    g.drawString("Hello, World Wide Web!", 20, 20);
  }
}
```

An appletviewer may be used instead of a WWW browser to test applets. For example, the output of `TrivialApplet` on an appletviewer looks like:



HTML/Applet Interface

The HTML `applet` tag is similar to the HTML `img` tag, and has the form:

```
<applet code=AppletName.class width=w height=h>
[parameters]
</applet>
```

where the optional parameters are a list of parameter definitions of the form:

```
<param name=n value=v>
```

An example tag with parameter definitions is:

```
<applet code=AppletName.class width=300 height=200>
<param name=p1 value=34>
<param name=p2 value="test">
</applet>
```

where `p1` and `p2` are user-defined parameters.

The `code`, `width`, and `height` parameters are mandatory. The parameters `codebase`, `alt`, `archives`, `align`, `vspace`, and `hspace` are optional within the <applet> tag itself. Your applet can access any of these parameters by calling:

```
Applet.getParameter("p")
```

which returns the `String` value of the parameter. For example, the applet:

```
import java.applet.Applet;

public class ParamTest extends Applet {
  public void init() {
    System.out.println("width is " + getParameter("width"));
    System.out.println("p1 is "    + getParameter("p1"));
```

```
    System.out.println("p2 is "    + getParameter("p2"));
  }
}
```

prints the following to standard output:

```
width is 300
p1 is 34
p2 is test
```

Comments

Java comments are the same as C++ comments, i.e.,

```
/* C-style block comments */
```

where all text between the opening `/*` and closing `*/` is ignored, and

```
// C++ style single-line comments
```

where all text from the opening `//` to the end of the line is ignored.

Note that these two comments can make a very useful combination. C-style comments (`/* ... */`) cannot be nested, but *can* contain C++ style comments. This leads to the interesting observation that if you always use C++-style comments (`// ...`), you can easily comment out a section of code by surrounding it with C-style comments. So try to use C++ style comments for your "normal" code commentary, and reserve C-style comments for commenting out sections of code.

The Java language also has a document comment:

```
/** document comment */
```

These comments are processed by the `javadoc` program to generate documentation from your source code. For example,

```
/** This class does blah blah blah */
class Blah {
  /** This method does nothing
  /**
   * This is a multiple line comment.
   * The leading * is not placed in documentation.
   */
  public void nothing() {;}
}
```

Declarations

A Java variable may refer to an object, an array, or an item of primitive type. Variables are defined using the following simple syntax:

```
TypeName variableName;
```

For example,

```
int a;     // defines an integer
int[] b;   // defines a reference to array of ints
Vector v;  // reference to a Vector object
```

Primitive Types

The Java language has the following primitive types:

Primitive Types

Primitive Type	*Description*
`boolean`	`true/false`
`byte`	8 bits
`char`	16 bits (UNICODE)
`short`	16 bits
`int`	32 bits
`long`	64 bits
`float`	32 bits IEEE 754-1985
`double`	64 bits IEEE 754-1985

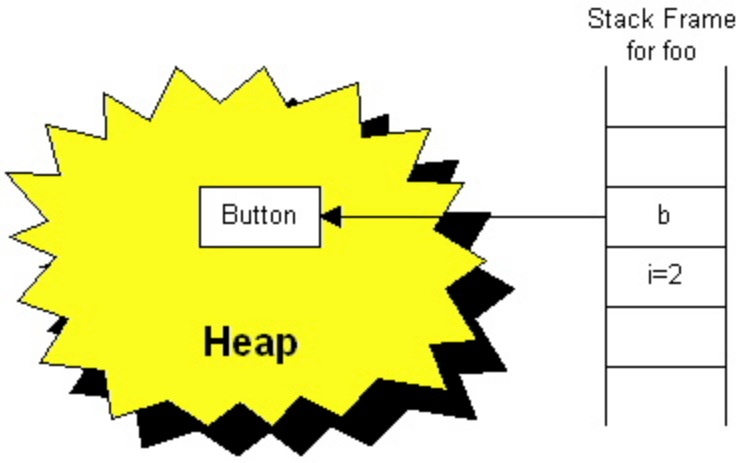
Java `int` types may **not** be used as `boolean` types and are always signed.

Objects

A simple C++ object or C struct definition such as `"Button b;"` allocates memory on the stack for a `Button` object and makes `b` refer to it. By contrast, you must specifically instantiate Java objects with the `new` operator. For example,

```
// Java code

void foo() {
  // define a reference to a Button; init to null
  Button b;
  // allocate space for a Button, b points to it
  b = new Button("OK");
  int i = 2;
}
```



As the accompanying figure shows, this code places a reference `b` to the `Button` object on the stack and allocates memory for the new object on the heap.

The equivalent C++ and C statements that would allocate memory on the heap would be:

```
// C++ code
Button *b = NULL;       // declare a new Button pointer
b = new Button("OK");   // point it to a new Button

/* C code */
Button *b = NULL;               /* declare a new Button pointer */
b = calloc(1, sizeof(Button)); /* allocate space for a Button */
init(b, "OK");                  /* something like this to init b */
```

All Java objects reside on the heap; there are no objects stored on the stack. Storing objects on the heap does not cause potential memory leakage problems because of garbage collection.

Each Java primitive type has an equivalent object type, e.g., `Integer`, `Byte`, `Float`, `Double`. These primitive types are provided in addition to object types purely for efficiency. An `int` is much more efficient than an `Integer`.

Strings

Java string literals look the same as those in C/C++, but Java strings are real objects, not pointers to memory. Java strings may or may not be `null`-terminated. Every string literal such as

```
"a string literal"
```

is interpreted by the Java compiler as

```
new String("a string literal")
```

Java strings are constant in length and content. For variable-length strings, use `StringBuffer` objects.

Strings may be concatenated by using the plus operator:

```
String s = "one" + "two"; // s == "onetwo"
```

You may concatenate any object to a string. You use the `toString()` method to convert objects to a `String`, and primitive types are converted by the compiler. For example,

```
String s = "1+1=" + 2; // s == "1+1=2"
```

The length of a string may be obtained with String method `length()`; e.g., `"abc".length()` has the value 3.

To convert an `int` to a `String`, use:

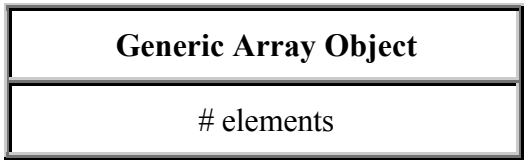
```
String s = String.valueOf(4);
```

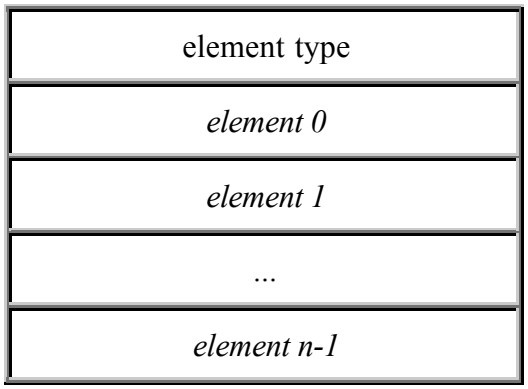
To convert a `String` to an `int`, use:

```
int a = Integer.parseInt("4");
```

Array Objects

In C and C++, arrays are pointers to data in memory. Java arrays are objects that know the number and type of their elements. The first element is index `0`, as in C/C++.





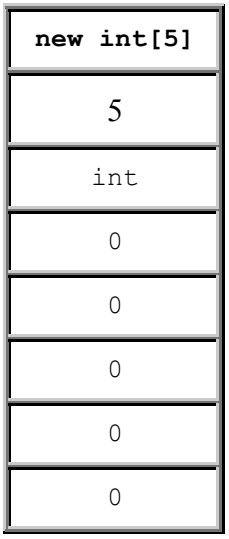
The syntax for creating an array object is:

```
TypeName[] variableName;
```

This declaration defines the array object--it does *not* allocate memory for the array object *nor* does it allocate the elements of the array In addition, you may not specify a size within the square brackets.

To allocate an array, use the `new` operator:

```
int[] a = new int[5];  // Java code: make array of 5 ints
```



In C or C++, by contrast, you would write either

```
/* C/C++ code: make array of 5 ints on the stack */
int a[5];
```

or

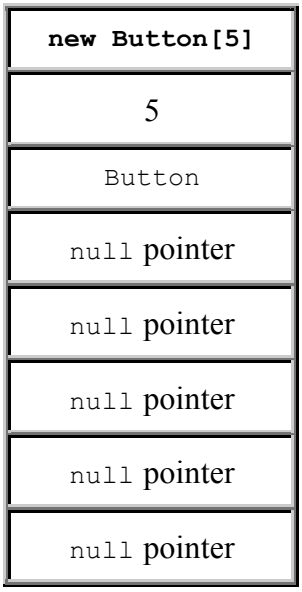
```
/* C/C++ code: make array of 5 ints on the heap */
```

```
int *a = new int[5];
```

An array of Java objects such as

```
// Java code: make array of 5 references to Buttons
Button[] a = new Button[5];
```

creates the array object itself, but not the elements:



You must use the `new` operator to create the elements:

```
a[0] = new Button("OK");
a[3] = new Button("QUIT");
```

In C++, to make an array of pointers to objects you would write:

```
// C++: make an array of 5 pointers to Buttons
Button **a = new Button *[5]; // Create the array
a[0] = new Button("OK");      // create two new buttons
a[3] = new Button("QUIT");
```

In C, code for the same task would look like:

```
/* C: make an array of 5 pointers to structs */
/* Allocate the array */
Button **a = calloc(5, sizeof(Button *));
/* Allocate one button */
a[0] = calloc(1, sizeof(Button));
/* Init the first button */
setTitle(a[0], "OK");
/* Allocate another button */
a[3] = calloc(1, sizeof(Button));
/* Init the second button */
```

```
setTitle(a[3], "QUIT");
```

Multi-dimensional Java arrays are created by making arrays of arrays, just as in C/C++. For example,

```
T[][] t = new T[10][5];
```

makes a five-element array of ten arrays of references to objects of type `T`. This statement does not allocate memory for any `T` objects.

Accessing an undefined array element causes a runtime exception called `ArrayIndexOutOfBoundsException`.

Accessing a defined array element that has not yet been assigned to an object results in a runtime `NullPointerException`.

Initializers

Variables may be initialized as follows:

- Primitive types

```
int i = 3;
boolean g = true;
```

- Objects

```
Button b = null;
Employee e = new Employee();
```

- Arrays

```
int[] i = {1, 2, 3, 4};
```

or in Java 1.1

```
int[] i;
i = new int[] {1, 2, 3, 4};
```

Constants

Variables modified by the `static final` keywords are constants (equivalent to the `const` keyword in C++; no equivalent in C). For example,

```
// same as "const int version=1;" in C++
static final int version = 1;

static final String Owner = "Terence";
```

Expressions

Most Java expressions are similar to those in C/C++.

Constant Expressions

Item	Examples or Description
id	`i, nameList`
qualified-id	`Integer.MAX_VALUE, obj.member, npackage.class, package.obj`
`id[e][f]...[g]`	`a[i], b[3][4]`
`String` literal	`"Jim"`, delimited by ""
`char` literal	`'a', '\t'`, delimited by '
Unicode character constant	`\u00ae`
`boolean` literal	`true`, `false` (*not* an int)
`int` constant	`4`
`float` constant	`3.14f, 2.7e6F`, f or F suffix
`double` constant	`3.14, 2.7e6D`, (default) / d or D suffix
hexadecimal constant	0x123
octal constant	077
`null`	the `null` object (note lowercase!)
`this`	the current object
`super`	the superclass view of this object

General Expressions

Item	Examples or Description

id	`i, nameList`
`obj.method(args)`	instance method call
`class.method(args)`	class method call
`( expr )`	(3+4)*7
`new T(`*`constructor-args`*`)`	instantiates a new object or class T
`new T[e][f]...[g]`	allocates an array object

Operators

The Java language has added the >>> zero-extend right-shift operator to the set of C++ operators. (C++ operators include `instanceof` and `new`, which are not present in C. Note that `sizeof` has been removed, as memory allocation is handled for you.) The operators, in order of highest to lowest priority, are:

- `new`
- `.`
- `-- ++ + - ~ ! (TypeName)`
- `* / %`
- `+ -`
- `<< >> >>>`
- `< > <= >= instanceof`
- `== !=`
- `&`
- `^`
- `|`
- `&&`
- `||`
- `?:`
- `= *= /= %= += -= <<= >>= >>>= &= ^= |=`

Note that the precedence of the `new` operator and the `'.'` operator bind

differently than in C++. A proper Java statement is:

```
// Java code
new T().method();
```

In C++, you would use:

```
// C++ code

(new T)->method();
```

Statements

Java statements are similar to those in C/C++ as the following table shows.

Forms of Common Statements

Statement	*Examples*
`if`	`if (`*`boolean-expr`*`)` *`stat1`* `if (`*`boolean-expr`*`)` *`stat1`* `else` *`stat2`*

`switch`	`switch (`*`int-expr`*`) {`
	`case` *`int-const-expr`* `:` *`stat1`*
	`case` *`int-const-expr`* `:` *`stat2`*
	`default :` *`stat3`*
	`}`

`for`	`for (int i=0; i<10; i++) stat`
`while`	`while (`*`boolean-expr`*`)` *`stat`*
`do-while`	`do {` *`stats`* `} while (`*`boolean-expr`*`)`
`return`	`return` *`expr;`*

The Java `break` and `continue` statements may have labels. These labels refer to the specific loop that the `break` or `continue` apply to. (Each loop can be preceded by a label.)

Java Semantics

We say that the Java language has "reference semantics" and C/C++ have "copy semantics." This means that Java objects are passed to methods by reference in Java, while objects are passed by value in C/C++.

Java primitive types, however, are not treated in the same way as Java objects. Primitive types are assigned, compared, and passed as arguments using copy

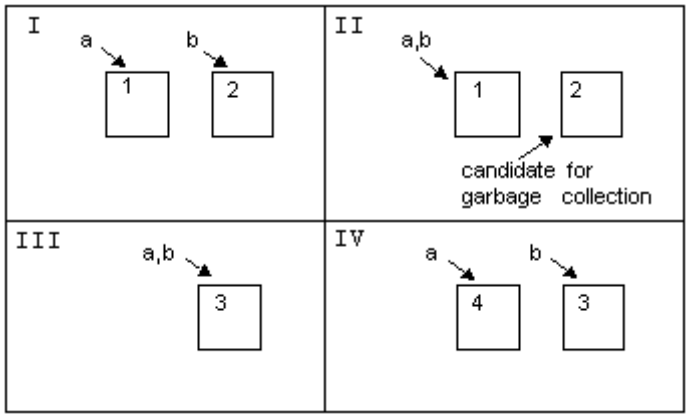
semantics, just as in C/C++. For example, `i = j` for two `int` variables `i` and `j` performs a 32-bit integer copy.

Assignment of Objects

Assignment makes two variables refer to the same object. For example,

```
class Data {
  public int data = 0;
  public Data(int d) { data = d; }
}


I       Data a = new Data(1); // a.data is 1
I       Data b = new Data(2); // b.data is 2
II      b = a;                // b.data and a.data are 1
III     a.data = 3;           // b.data and a.data are 3
IV      a = new Data(4);      // b.data is 3, a.data is 4
```



To copy objects, define and use `clone()`:

```
class Data implements Cloneable {
  public int data = 0;
  public Data(int d) { data = d; }
  public Object clone() {
    Data d = (Data) super.clone();
    d.data = data;
    return d;
  }
}

...

Data a = new Data(1); // a.data is 1
Data b = new Data(2); // b.data is 2
b = a.clone();        // b.data and a.data are 1
a.data = 3;           // b.data is 1, a.data is 3
```

Note: The above class definition requires exception handling code. We, however, have not yet discussed exception handling. For now, pretend that it is not necessary.

Method Parameters and Return Values

Arguments and return values for primitive types are passed by value to and from all Java methods because they are implied assignments, as in C/C++. However, all Java objects are passed by reference. For example, the C/C++ code:

```
// C++ code

int foo(int j) { return j + 34;}
Button *bfoo(Button *b) {
  if ( b != NULL ) return b;
  else return new Button();
}
```

or, in C

```
/* C code */

int foo(int j) { return j + 34;}
Button *bfoo(Button *b) {
  if ( b != NULL ) return b;
  else return calloc(sizeof(Button));
}
```

would be written in the Java language:

```
// Java code

int foo(int j) { return j + 34;}
Button bfoo(Button b) {
  if ( b != null ) return b;
  else return new Button("OK");
}
```

Equality

Two Java primitive types are *equal* (using the == operator) when they have the same value (e.g., `"3 == 3"`). However, two object variables are equal if and only if they refer to the same instantiated object--a "shallow" comparison. For example,

```
void test() {
  Data a = new Data(1);
  Data b = new Data(2);
  Data c = new Data(1);
  // a == b is FALSE
  // a == c is FALSE (in C++, this'd be TRUE)
```

```
  Data d = a;
  Data e = a;
  // d == e is TRUE,
  // d,e are referring to same object
}
```

To perform a "deep" comparison, the convention is to define a method called `equals()`. You would rewrite `Data` as:

```
class Data {
  public int data = 0;
  public Data(int d) { data = d; }
  boolean equals(Data d) {
    return data == d.data;
  }
}

...

Data a = new Data(1);
Data b = new Data(1);
// a.equals(b) is true!!!!
```

No Pointers!

The Java language does not have pointer types nor address arithmetic. Java variables are either primitive types or references to objects. To illustrate the difference between C/C++ and Java semantics, consider the following equivalent code fragments.

```
// C++ code (C code would be similar)

Stack *s = new Stack; // point to a new Stack
s->push(...);

// dereference and access method push()
```

The equivalent Java code is:

```
// Java code

// internally, consider s to be a (Stack *)
Stack s = new Stack();
// dereference s automatically
s.push(...);
```

Garbage Collection

An automatic garbage collector deallocates memory for objects that are no longer needed by your program, thereby relieving you from the tedious and error-prone

task of deallocating your own memory.

As a consequence of automatic garbage collection and lack of pointers, a Java object is either `null` or valid--there is no way to refer to an invalid or stale object (one that has been deallocated).

To illustrate the effect of a garbage collector, consider the following C++ function that allocates 1000 objects on the heap via the `new` operator (a similar C function would allocate memory using calloc/malloc):

```
// C++ code

void f() {
  T *t;
  for (int i = 1; i <= 1000; i++) {
    t = new T; // ack!!!!!
  }
}
```

Every time the loop body is executed, a new instance of class `T` is instantiated, and `t` is pointed to it. But what happens to the instance that `t` used to point to? It's still allocated, but nothing points to it and therefore it's inaccessible. Memory in this state is referred to as "leaked" memory.

In the Java language, memory leaks are not an issue. The following Java method causes no ill effects:

```
// Java code

void f() {
  T t;
  for (int i = 1; i <= 1000; i++) {
    t = new T();
  }
}
```

In Java, each time `t` is assigned a new reference, the old reference is now available for garbage collection. Note that it isn't immediately freed; it remains allocated until the garbage collector thread is next executed and notices that it can be freed.

Put simply, automatic garbage collection reduces programming effort, programming errors, and program complexity.

[*MML: 0.995a*]
[*Version: $Id: //depot/main/src/edu/modules/JavaBasics/javaBasics.mml#3 $*]

This page intentionally left blank