

Packages, Classes, Fields and Methods

Java requires that for every class, instance variable, and method you define, that you declare an access qualifier as part of the definition, or accept the default access (which is not private access) Compare this to what you must write in C++. A public Java class must correspond to a file of the same name. So, the Hello class must be defined in a Hello.java file, which is part of the 'test' package. Technically, the main method is called:

```
edu.utexas.cs.test.Hello.main() // a fully qualified method name
```

NOTE: Unlike C++ which uses the '::' operator for scope resolution, Java uses the '.' operator for both scope resolution and class member access.

A *fully qualified name* is of the form <package>.<class>.<field or method>. You might organize your packages into subpackages, in which case you write <package>.<subpackage>.<class>.<field or method>. A fully qualified name corresponds to a directory path in the filesystem, which is how Java is able to guarantee unique naming. So the name test.Hello might correspond to a file:

```
/home/user/edu/utexas/cs/test/Hello.java
```

A single .java file may contain multiple class definitions for a package, but only one may be public. The name of the public class must correspond to the name of the file. The java compiler will compile a .java file with multiple classes into multiple .class files: one for each class.

The CLASSPATH environment variable is used to tell the java interpreter where to look for packages and classes. Rather than always using a fully qualified name all the time, you can "open" a package using the **import** statement. When you specify an import statement, it opens the package, and allows you to omit the package name qualifiers. A package name may be a compound name: For example, the following input/output (io) subpackage is part of the standard Java SDK package:

```
import java.io.*;
```

The import statement causes the java.applet package to be opened, and the "*" says to import all classes from that package.

References and Objects

The non-primitive data types in Java are **objects** and **arrays**, which are called reference types because objects and arrays are accessed using reference variables, which contain a reference to the actual object or array. When you declare a variable for a reference type, it is initialized to the special value **null**, which is a reserved word in java.

There are **NO pointers in Java**. That implies that there is also no address-of operator '&', no dereference operator '*', and no pointer access operator '>', as there are in C/C++.

```
Button p = null;
Button q = null;
p = new Button();
q = p;
p.setLabel("OK");
String s = q.getLabel(); // gets the value "OK" from button
```

Strings have somewhat special treatment in Java because they are so often used. You can write the following:

```
String s = "Hello, world";
```

instead of

```
String s = new String("Hello, world");
```

The Java compiler turns all string literals, like "Hello, world", into heap allocated, garbage collected, String objects automatically.

Unlike C++, user-defined operator overloading is not allowed in Java. However, Java does provide all the standard C operators, and includes a few builtin overloadings. For example, the '+' operator is overloaded for String objects, so that you can perform concatenation.

```
System.out.print("hello," + " world");
```

Java Data Types

Java has three categories of data types: **primitive types**, **arrays**, and **reference types**.

Because java programs execute on a virtual machine, the primitive types are defined to be the same across all real machine architectures. Not that this is different than in C/C++, where the real machine architecture has an impact of the precision. Variables defined as a primitive boolean type default to false on declaration within a class:

boolean	true or false
---------	---------------

Variables defined as one of the following primitive integral types default to a value of zero on declaration with a class:

byte	-128..127
short	-32768..32767
int	-2147483648..2147483647
long	-922337203685477508..922337203685477507
char	0..65535

Characters in java are Unicode characters (16 bits). The Java String class implements a Unicode character string object, so strings can contain non-ASCII character for international languages. This is a very important feature today, since software should be written to permit internalization of character values and strings. For example, so that multi-lingual error message catalogs can be used.

Java's floating point types implement IEEE-754 single-precision (32-bit) and double precision (64-bit) values. They default to 0.0f or 0.0d when declared in a class but not initialized:

```
float
double
```

Objects

All objects are by default inherited from a base object class called **java.lang.Object**. Java objects are allocated from the heap and a garbage collector takes care of deleting the objects from the heap when there are no more references to the object. When you create a new object, the result of the new operator is a reference to the heap allocated object that is assigned to an object reference variable. You use this object reference variable to access the public fields and methods of an object.

The Object class provides some high-level methods that can be called on an object of any type. For example, the object class defines a clone method that makes a copy of an object:

You can copy objects using a special method called `java.lang.Object.clone()`, which is defined for all objects since all objects inherit from the `java.lang.Object` class.

```
Vector v = new Vector();
Vector z = v;
Vector c = z.clone();
```

The `java.lang.Object.clone()` method does a field by field copy of the object.

Equality between objects can be tricky. You have to distinguish between equivalent references and equivalent objects. Two references are equal if they refer to the same object.

```
v == z is true
c == z is false
```

To check to see if two distinct objects are equivalent, you call the equals() method for those objects. The equals method can be overridden by a class to implement a type specific comparison. Every object inherits the `java.lang.Object.equals()` method, which just checks to see if two references point at the same object. The statement:

```
x.equals(null)
```

for some reference type variable x should always yield false. Why?

The Object Class

```
package java.lang;
/**
 * Class <code>Object</code> is the root of the class hierarchy.
 * Every class has <code>Object</code> as a superclass. All objects,
 * including arrays, implement the methods of this class.
 *
 * @author unascribed
 * @version 1.39, 01/20/97
 * @see java.lang.Class
 * @since JDK1.0
 */
public class Object {
    public final native Class getClass();
    public native int hashCode();
    public String toString() {
        return getClass().getName() + "@" + Integer.toHexString(hashCode());
    }
    // object equality and copying (cloning) methods
    public boolean equals(Object obj) { return (this == obj); }
    protected native Object clone() throws CloneNotSupportedException;
    // object synchronization methods
    public final native void notify();
    public final native void notifyAll();
    public final native void wait(long timeout) throws InterruptedException;
    public final void wait(long timeout, int nanos) throws InterruptedException { ... }
    public final void wait() throws InterruptedException { wait(0); }
    // object cleanup (garbage collection) methods
    protected void finalize() throws Throwable { }
}
```

The Object.equals Method

```
/**
 * Compares two Objects for equality.
 * <p>
 * The equals method for class <code>Object</code> implements the most
 * discriminating possible equivalence relation on objects; that is,
 * for any reference values <code>x</code> and <code>y</code>, this
 * method returns <code>>true</code> if and only if <code>x</code> and
 * <code>y</code> refer to the same object (<code>x==y</code> has the
 * value <code>>true</code>).
 *
 * @param obj the reference object with which to compare.
 * @return <code>>true</code> if this object is the same as the obj
 *         argument; <code>>false</code> otherwise.
 * @see java.lang.Boolean#hashCode()
 * @see java.util.Hashtable
 * @since JDK1.0
 */
public boolean equals(Object obj) {
    return (this == obj);
}
```

The Object.toString Method

```
/**
 * Returns a string representation of the object. In general, the
 * <code>toString</code> method returns a string that
 * "textually represents" this object. The result should
 * be a concise but informative representation that is easy for a
 * person to read.
 * It is recommended that all subclasses override this method.
 * <p>
 * The <code>toString</code> method for class <code>Object</code>
 * returns a string consisting of the name of the class of which the
 * object is an instance, the at-sign character '<code>@</code>', and
 * the unsigned hexadecimal representation of the hash code of the
 * object.
 *
 * @return a string representation of the object.
 * @since JDK1.0
 */
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

The Object.clone Method

```
/**
 * Creates a new object of the same class as this object. It then
 * initializes each of the new object's fields by assigning it the
 * same value as the corresponding field in this object. No
 * constructor is called.
 * <p>
 * The <code>clone</code> method of class <code>Object</code> will
 * only clone an object whose class indicates that it is willing for
 * its instances to be cloned. A class indicates that its instances
 * can be cloned by declaring that it implements the
 * <code>Cloneable</code> interface.
 *
 * @return a clone of this instance.
 * @exception CloneNotSupportedException if the object's class does not
 *         support the <code>Cloneable</code> interface. Subclasses
 *         that override the <code>clone</code> method can also
 *         throw this exception to indicate that an instance cannot
 *         be cloned.
 * @exception OutOfMemoryError if there is not enough memory.
 * @see java.lang.Cloneable
 * @since JDK1.0
 */
protected native Object clone() throws CloneNotSupportedException;
```

The Object.finalize Method

```
/**
 * Called by the garbage collector on an object when garbage collection
 * determines that there are no more references to the object.
 * A subclass overrides the <code>finalize</code> method to dispose of
 * system resources or to perform other cleanup.
 * <p>
 * Any exception thrown by the <code>finalize</code> method causes
 * the finalization of this object to be halted, but is otherwise
 * ignored.
 * <p>
 * The <code>finalize</code> method in <code>Object</code> does
 * nothing.
 *
 * @exception java.lang.Throwable [Need description!]
 * @since JDK1.0
 */
protected void finalize() throws Throwable { }
```

Example Java Class: java.lang.String

```
public final class String {
    private char[] value; // The value is used for character storage.
    private int offset; // The offset is the first index of the storage that is used.
    private int count; //The count is the number of characters in the String. */

    public String() { value = new char[0]; }

    public String(String value) {
        count = value.length();
        this.value = new char[count];
        value.getChars(0, count, this.value, 0);
    }

    public String(char[] value) {
        this.count = value.length;
        this.value = new char[count];
        System.arraycopy(value, 0, this.value, 0, count);
    }

    ... // lots of other stuff
};
```

A “final” class means that the class cannot be extended by a subclass. Several of the `java.lang` package classes are declared as final.

Note the `System.arraycopy` method. It copies from `value` beginning at the first position, to `this.value` beginning at the first position, for `count` elements.

Arrays

Arrays are created by calling `new`, which returns a reference to the array. Arrays are automatically garbage collected.

```
byte octet_buffer[] = new byte[1024];
```

Each byte in the `octet_buffer` array is initialized to its default value, which is zero for integral types.

```
Button button[] = new Button[10];
```

An array of objects results in an array of `Button` references, all of which are null. This is different from C++, where an array of objects causes the default constructor to be executed for each array element. So, you have to initialize each array reference with a new object.

```
for (int i = 0; i < button.length; i++)
    button[i] = new Button();
```

Every array as a “const” length field, which is a read-only variable that returns the length of the array as an integer. You also can initialize an array with a static initializer:

```
String commands[] = { "File", "Edit", "Format", "Help" };
```

Which is equivalent to:

```
String[] commands = { new String("File"), new String("Edit"),
    new String("Format"), new String("Help") };
```

Note that in Java, unlike C/C++, you can place the `[]` indicating an array, next to the typename `T` instead of the variable name. This is more intuitive since what you are reference to an array of some type `T`. For example:

```
String[] s; // declare a reference to an array of Strings
s = new String[100];
s = new String[10]; // reuse the array reference variable, previous array is garbage
```

String Equality

The `java.lang.Object.equals()` method is overridden by the `String` class to perform an equality check. The type signature of the `equals` method takes a single `Object` reference, which must be cast down to a `String` object, but only if the object is an “instance of” the `String` class. The `instanceof` operator returns true if the object on the LHS is an instance of the class on the RHS, or implements the interface of the class on the RHS. It returns false if the object on the LHS is not an instance of the class, or if the object is null.

Note that the argument of type `Object` must be downcast to a string and a run-time “instance of” check is performed. If you downcast from `Object` to a incorrect type, then a run-time “invalid type cast” exception is thrown

```
public boolean equals(Object obj) {
    if ((obj != null) && (obj instanceof String)) {
        String anotherString = (String)obj; // safe downcast since instanceof true
        int n = count;
        if (n == anotherString.count) {
            char v1[] = value;
            char v2[] = anotherString.value;
            int i = offset;
            int j = anotherString.offset;
            while (n-- != 0) {
                if (v1[i++] != v2[j++]) {
                    return false;
                }
            }
            return true;
        }
    }
    return false;
}
```