

Object Initialization

Objects, unlike the primitive types, must be constructed. An object is constructed when it is instantiated by a call to the builtin `new` operator.

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

Each class will usually have one or more constructors defined. Constructors define the different ways in which an object can be constructed. If no destructor is defined, then the Java compiler generates a default constructor for the object. A constructor is easily identified, because it has the same name as the class. Constructors can be *overloaded*, so there can be several constructor methods with the same name, but their type signatures must be unique

```
public class SomeObject {
    private int len; // len defaults to zero
    String s; // s defaults to null

    SomeObject() { this.len = 0; } // default constructor

    SomeObject (int x) { len = x; } // overloaded constructor

    SomeObject (String s1, String s2) { this(s1.length()+s2.length()); s = s1 + s2; }
}
```

```
SomeObject a = new SomeObject();
SomeObject b = new SomeObject(5);
SomeObject c = new SomeObject("hello", " world");
```

Note that in Java, unlike C++, a constructor can call another constructor. Constructors also do not return any value.

When an object is created, all data local to that object is automatically initialized before any constructor is called. So, you are guaranteed that all primitive types and references have a well-defined value. A constructor can redefine the value as soon as it executes. The order of initialization of local data is the order in which it is declared.

Overloaded Method Names

Note that in order for an overloaded constructor name to be distinct, the type, number, and order of the arguments declared for the constructor have to be unique. So, all of the following are unique overloaded constructor names:

```
MyString ()  
MyString (char c)  
MyString (char a, char b)  
MyString (char[] c_array)  
MyString (char c, byte b)  
MyString (byte[] b_array)  
MyString (byte[] b_array, int len)  
MyString (int len, byte[] b_array)
```

However, it is not possible to distinguish a method on its return type alone, for example:

```
float f();  
int f();
```

This is because ambiguity can arise when you don't expect it to. If you are explicit in your usage, then the compiler could deduce which method you mean:

```
int x = f(); // call the f() that returns an int
```

However, if you have the following expression

```
int x = abs(y) + f();
```

Which f() should be called? The one that returns a float or an int? What about this case:

```
f(); // which one should be called?
```

There are no default arguments in Java

In C++, it is possible to define default arguments to constructors and methods, like so:

```
class Complex {
    double real, imag;
public:
    Complex () { real = 0.0; imag = 0.0 }
    Complex (double r = 0.0) { real = r; imag = 0.0; }
    Complex (double r = 0.0, double i = 0.0) { real = r; imag = i; }
}
```

The problem with default arguments, is that ambiguity can occur frequently. What happens in C++ in the following case?

```
Complex c = new Complex();
```

Which constructor is called?

Java avoids this type of complexity by not allowing default arguments. However, the way that you can achieve a similar effect, is to take advantage of default initialization and the ability for one constructor call another constructor:

```
public class Complex {
    private double real;
    private double imag;

    Complex(double r, double i) { real = r; image = i; }
    Complex(double r) { this(r, 0.0); }
    Complex() { this(0.0, 0.0); }
}
```

The “this” Reference

Every object in Java is uniquely identified by its “this” reference. The this reference is the “handle” to the object that is used to invoke methods or explicitly reference instance variables.

When we create an object by calling `new`, a reference variable is initialized with the value of the object’s location in memory. When we use a reference variable to invoke an operation on an object, we are really invoking a method that is associated with some class, and we dynamically associate the state of some particular object with the method.

```
String s = new String("This is a string");  
int x = s.length(); // compute length of string
```

The compiler turns the invocation of `s.length` into the following function call:

```
int x = String.length(this=s);
```

So, the `length` method of the `String` class is invoked, and it will have a `this` reference that is initialized to the object pointed to by `s`.

Conceptually, what you have to keep in mind in an object oriented language like Java or C++, is that the actual objects are really just a chunk of memory somewhere in the heap that contain the data elements of the object, along with some bookkeeping information generated by the compiler for the object. At run-time, the object consists of data in the heap, and a single copy of the methods for all objects of a particular class are separate. When you invoke a method on an object, the method and the object’s state are dynamically “tied together” for the duration of the method’s execution. The “this” reference is the mechanism that is used to tie them together at run-time.

So, you should think of the `this` reference as a hidden first argument to every non-static class method. Static methods do not have a `this` reference passed to them, as static methods are in fact “Class Methods”, that is to say, methods defined for the class, not on objects. In Java, static methods are very often called “Factory Methods” as they are used to manufacture objects by invoking a method on a class, not on an object.

Using the “this” Reference in a Class

Most of the time, you will not explicitly refer to the “this” reference, unless you are invoking a constructor from another constructor, as shown previously. However, some people prefer a programming style that requires the use of the this reference to avoid ambiguity:

```
class Complex {
    double real, imag;

    Complex(double real, double imag) { this.real = real; this.imag = imag; }
    ...
}
```

In this case, the programmer wants to convey to the user of the class as much information as possible about the arguments expected by the constructor, so the same names of the local variables are used. However, this requires that the this reference be used to disambiguate the local real and imag variables from those used as formal parameters to the constructor. So, you sometimes see this explicit use of the this pointer when referring to local instance variables.

It is not possible to refer to the this variable in a static method of a class.

```
class Foo {
    int x;
    ...
    static int f() { return this.x; } // ERROR
}

int x = Foo.f();
```

Since f() is static, it is called on the class Foo, not an object of type Foo. So, it is NOT the case that f() is passed an implicit first argument to which the “this” reference is initialized. So, within the body of f(), the “this” reference is uninitialized, and therefore unusable.

Static Methods

The most common static method you will write is the main method. Every Java program has to have at least one main method defined in some class.

```
import java.net.*; // need to import InetAddress class

public class Host {
    static String host;
    static String user;
    static final int err = -1; // same as const int err = -1 in C++

    public static void main (String[] args)
    {
        try {
            InetAddress addr = InetAddress.getLocalHost();
            host = addr.getHostName();
            user = System.getProperty("user.name");
            System.out.println(user + "@" + host);
        }
        catch (Exception e) { System.err.println(e); System.exit(err); }
    }
}
```

Notice that the call to `InetAddress.getLocalHost()` is a static method call, which returns an `InetAddress` object. This is an example of a static “Factory Method”. The method `getLocalHost` manufactures an instance of the `InetAddress` object for the local machine on which the program is running. An `InetAddress` object has the Internet address of the local machine. Using that `InetAddress` object, I can then obtain the string `hostname` by calling the non-static `getHostName` method on the `InetAddress` object ‘`addr`’. Similarly, the `System.getProperty` method is a static method defined as part of the `java.lang.System` class, which returns back a string object containing the value of system property “`user.name`”. So, we would expect the output in this case to be “`user@host.cs.utexas.edu`”

Note that the static ‘`err`’ variable is declared **final**, which means immutable, which is like a constant in C++.

Static Data

The `java.lang.System` class is one that you should familiarize yourself with, as it consists of mostly static methods and static data. Three static data members are very important:

```
java.lang.System.in  
java.lang.System.out  
java.lang.System.err
```

These are the Standard Input, Output, and Error objects for doing input and output.

```
import java.io.*  
...  
public static void main(String[] args) {  
  
    String input = null; // need to initialize any variables declared local to a method  
    BufferedReader rdr = new BufferedReader(new InputStreamReader(System.in));  
  
    // read line buffered input from System.in until EOF  
    try {  
        do {  
            input = rdr.readLine();  
            if (input != null)  
                System.out.println(input);  
        } while (input != null);  
    }  
    catch (IOException e) { System.err.println(e); System.exit(-1); }  
}
```

`System.in` is actually a `InputStream` object, which is not line buffered, so you have to create an `InputStreamReader` object and use it to initialize a `BufferedReader` object. In Java, you will find that you have several different types of Input/Output stream objects, and you need to be able to convert them for different uses.

Static Data Initialization

Static data is usually initialized at the point of declaration within a class:

```
class Program {
    static String s = "hello, world";
    static int x = -1;
    ...
}
```

Alternatively, you can use a **static block** to do the initialization.

```
class Program {
    static String s;
    static int x;

    static {
        s = new String("hello, world");
        x = -1;
    }
    ...
}
```

A static block is NOT the same thing as a constructor. Static variables are initialized when either a static method is called on the class that contains the static data, or an object of the class is first instantiated.

Object Cleanup

As already mentioned, Java has a garbage collector that looks for objects that are no longer referenced, and it collects the storage associated with those objects. Before the garbage collector collects an object, it first arranges for the **finalize** method to be executed.

A Java finalize method is NOT the same thing as a destructor in C++. In Java, the finalize method is called at the time of garbage collection, and normally, you have no idea when that will happen. The purpose of the finalize method (which by-the-way is one of the methods inherited from the class Object), is to give an object a chance to do some cleanup before the storage for the object is reclaimed.

The type of cleanup that can be done is limited, but might include closing a file, writing some log messages, or releasing some network resource.

```
import java.net.*

public class NetworkConnection {
    Socket s;

    NetworkConnection(String hostname, int port) {
        try {
            s = new Socket(hostname, port); // open connection to hostname+port
        }
        catch (Exception e) { /* do something useful */ }
    }
    ...
    void finalize() { s.close(); } // make sure socket gets closed when we die
}
```

Forcing a Garbage Collection Cycle

It is possible, although not generally required, to force the garbage collector to execute. Since you don't always know when there is sufficient garbage to warrant a garbage collection cycle when your program is running, if you call the garbage collector explicitly, you are probably just wasting CPU cycles.

However, the way you do it is by calling the `java.lang.System.gc()` method, which is really just a call to the `Runtime.getRuntime` Factory method to get the VM runtime object, and invoke the `gc()` method on the runtime object. In fact, many of the `System` class methods are just calls to the Java VM runtime. Note by the way that it is impossible to instantiate an object of the `System` class because its constructor is private!

```
public final
class System {

    /** Don't let anyone instantiate this class */
    private System() {}

    * Runs the garbage collector.
    * <p>
    * Calling the <code>gc</code> method suggests that the Java Virtual
    * Machine expend effort toward recycling unused objects in order to
    * make the memory they currently occupy available for quick reuse.
    * When control returns from the method call, the Java Virtual
    * Machine has made a best effort to reclaim space from all unused
    * objects.
    *
    * @see    java.lang.Runtime#gc()
    * @since  JDK1.0
    */
    public static void gc() { Runtime.getRuntime().gc(); }
```

Array Initialization

Arrays are not objects, and are initialized in a different manner. Arrays of primitive data elements are initialized in a manner similar to arrays in C and C++

```
int a[] = { 1, 2, 3, 4, 5 }; // looks just like C/C++
int[] a = { 1, 2, 3, 4, 5 }; // preferred declaration in Java
```

If you declare an array of objects, you can initialize that array in line as well, as shown in the following example:

```
public class Editor {
    String editor;

    Editor (String name) { editor = name; }

    int doEdit(String filename)
    {
        String[] cmd = new String[] { "xterm", "-e", editor, filename };
        Runtime r = Runtime.getRuntime(); // Factory method to get the VM's runtime object

        try {
            Process p = r.exec(cmd); // run editor as a subprocess of the Java VM
            return p.waitFor(); // wait for editor subprocess to complete
        }
        catch (Exception e) { System.err.println(e); return -1; }
    }
}
```

Array Bounds

Note that in Java, unlike C++, all array accesses are checked to ensure that there is not out-of-bounds access. As with C/C++, arrays start with index zero.

When you initialize an array, the array has a public final variable that is initialized, called the **length** field. Hence, you can always use this builtin array variable to ensure that you do not go over the bounds of the array.

```
int[] a = { 1, 2, 3, 4, 5 };
...
for (int i = 0; i < a.length; i++)
    a[i]++;
}
```

What if you wrote:

```
a[10] = 0;
```

Since you are attempting to access outside of the bounds of the array, an exception, called the **ArrayIndexOutOfBoundsException** is generated automatically. As long as you always make sure that for any given array `a`, you always access `a[0]` through `a[a.length - 1]`, you should not encounter this exception. A common place where you will encounter this exception, is if you expect arguments to `main`, and they are not passed

```
public class Main {
    public static void main (String[] args)
    {
        String[] s = new String[] { args[0], args[1] };
        ...
    }
}
% java Main
java.lang.ArrayIndexOutOfBoundsException: 0
    at Main.main(Main.java:4)
```