4: Initialization and cleanup

As the computer revolution progresses, “unsafe” programming has become one of the major culprits that makes programming expensive.

Two of these safety issues are initialization and cleanup. Many C bugs occur when the programmer forgets to initialize a variable. This is especially true with libraries when users don’t know how to initialize a library component, or even that they must. Cleanup is a special problem because it’s easy to forget about an element when you’re done with it, since it no longer concerns you. Thus, the resources used by that element are retained and you can easily end up running out of resources (most notably memory).

C++ introduced the concept of a constructor, a special method automatically called when an object is created. Java also adopted the constructor, and in addition has a garbage collector that automatically releases memory resources when they’re no longer being used. This chapter examines the issues of initialization and cleanup and their support in Java.

Guaranteed initialization with the constructor

You can imagine creating a method called initialize() for every class you write. The name is a hint that it should be called before using the object. Unfortunately, this means the user must remember to call the method. In Java, the class designer can guarantee initialization of every object by providing a special method called a constructor. If a class has a constructor, Java automatically calls that constructor when an object is created, before users can even get their hands on it. So initialization is guaranteed.

The next challenge is what to name this method. There are two issues. The first is that any name you use could clash with a name you might like to use as a member in the class. The second is that because the compiler is responsible for calling the constructor, it must always know which method to call. The C++ solution seems the easiest and most logical, so it’s also used in Java: The name of the constructor is the same as the name of the class. It makes sense that such a method will be called automatically on initialization.

Here’s a simple class with a constructor:

```java
//: SimpleConstructor.java
// Demonstration of a simple constructor
package c04;
```
class Rock {
    Rock() { // This is the constructor
        System.out.println("Creating Rock");
    }
}

public class SimpleConstructor {
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++)
            new Rock();
    }
} ///:~

Now, when an object is created:

new Rock();

storage is allocated and the constructor is called. It is guaranteed that the object will
be properly initialized before you can get your hands on it.

Note that the coding style of making the first letter of all methods lower case does not
apply to constructors, since the name of the constructor must match the name of the
class exactly.

Like any method, the constructor can have arguments to allow you to specify how an
object is created. The above example can easily be changed so the constructor takes
an argument:

class Rock {
    Rock(int i) {
        System.out.println("Creating Rock number " + i);
    }
}

public class SimpleConstructor {
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++)
            new Rock(i);
    }
}

Constructor arguments provide you with a way to provide parameters for the
initialization of an object. For example, if the class Tree has a constructor that takes
a single integer argument denoting the height of the tree, you would create a Tree
object like this:

Tree t = new Tree(12); // 12-foot tree

If Tree(int) is your only constructor, then the compiler won’t let you create a Tree
object any other way.

Constructors eliminate a large class of problems and make the code easier to read. In
the preceding code fragment, for example, you don’t see an explicit call to some
initialize( ) method that is conceptually separate from definition. In Java, definition and initialization are unified concepts – you can’t have one without the other.

The constructor is an unusual type of method because it has no return value. This is distinctly different from a void return value, in which the method returns nothing but you still have the option to make it return something else. Constructors return nothing and you don’t have an option. If there were a return value, and if you could select your own, the compiler would somehow need to know what to do with that return value.

Method overloading

One of the important features in any programming language is the use of names. When you create an object, you give a name to a region of storage. A method is a name for an action. By using names to describe your system, you create a program that is easier for people to understand and change. It’s a lot like writing prose – the goal is to communicate with your readers.

You refer to all objects and methods by using names. Well-chosen names make it easier for you and others to understand your code.

A problem arises when mapping the concept of nuance in human language onto a programming language. Often, the same word expresses a number of different meanings – it’s overloaded. This is useful, especially when it comes to trivial differences. You say “wash the shirt,” “wash the car,” and “wash the dog.” It would be silly to be forced to say, “shirtWash the shirt,” “carWash the car,” and “dogWash the dog” just so the listener doesn’t need to make any distinction about the action performed. Most human languages are redundant, so even if you miss a few words, you can still determine the meaning. We don’t need unique identifiers – we can deduce meaning from context.

Most programming languages (C in particular) require you to have a unique identifier for each function. So you could not have one function called print( ) for printing integers and another called print( ) for printing floats – each function requires a unique name.

In Java, another factor forces the overloading of method names: the constructor. Because the constructor’s name is predetermined by the name of the class, there can be only one constructor name. But what if you want to create an object in more than one way? For example, suppose you build a class that can initialize itself in a standard way and by reading information from a file. You need two constructors, one that takes no arguments (the default constructor), and one that takes a String as an argument, which is the name of the file from which to initialize the object. Both are constructors, so they must have the same name – the name of the class. Thus method overloading is essential to allow the same method name to be used with different argument types. And although method overloading is a must for constructors, it’s a general convenience and can be used with any method.
Here’s an example that shows both overloaded constructors and overloaded ordinary methods:

```java
//: Overloading.java
// Demonstration of both constructor
// and ordinary method overloading.
import java.util.*;

class Tree {
    int height;
    Tree() {
        prt("Planting a seedling");
        height = 0;
    }
    Tree(int i) {
        prt("Creating new Tree that is "+ i + " feet tall");
        height = i;
    }
    void info() {
        prt("Tree is " + height + " feet tall");
    }
    void info(String s) {
        prt(s + ": Tree is "+ height + " feet tall");
    }
    static void prt(String s) {
        System.out.println(s);
    }
}

public class Overloading {
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++) {
            Tree t = new Tree(i);
            t.info();
            t.info("overloaded method");
        }
        // Overloaded constructor:
        new Tree();
    }
}
```

A Tree object can be created either as a seedling, with no argument, or as a plant grown in a nursery, with an existing height. To support this, there are two constructors, one that takes no arguments (we call constructors that take no arguments default constructors) and one that takes the existing height.

You might also want to call the info() method in more than one way. For example, with a String argument if you have an extra message you want printed, and without if you have nothing more to say. It would seem strange to give two separate names to what is obviously the same concept. Fortunately, method overloading allows you to use the same name for both.

**Distinguishing overloaded methods**
If the methods have the same name, how can Java know which method you mean? There’s a simple rule: Each overloaded method must take a unique list of argument types.

If you think about this for a second, it makes sense: how else could a programmer tell the difference between two methods that have the same name, other than by the types of their arguments?

Even differences in the ordering of arguments is sufficient to distinguish two methods: (Although you don’t normally want to take this approach, as it produces difficult-to-maintain code.)

```java
//: OverloadingOrder.java
// Overloading based on the order of
// the arguments.

public class OverloadingOrder {
    static void print(String s, int i) {
        System.out.println(
            "String: " + s + ", int: " + i);
    }
    static void print(int i, String s) {
        System.out.println(
            "int: " + i + ", String: " + s);
    }
    public static void main(String[] args) {
        print("String first", 11);
        print(99, "Int first");
    }
} ///:~
```

The two `print()` methods have identical arguments, but the order is different, and that’s what makes them distinct.

**Overloading with primitives**

Primitives can be automatically promoted from a smaller type to a larger one and this can be slightly confusing in combination with overloading. The following example demonstrates what happens when a primitive is handed to an overloaded method:

```java
//: PrimitiveOverloading.java
// Promotion of primitives and overloading

public class PrimitiveOverloading {
    // boolean can't be automatically converted
    static void prt(String s) {
        System.out.println(s);
    }
    void f1(char x) { prt("f1(char)"); }
    void f1(byte x) { prt("f1(byte)"); }
    void f1(short x) { prt("f1(short)"); }
    void f1(int x) { prt("f1(int)"); }
} 5
```
void f1(long x) { prt("f1(long)"); }
void f1(float x) { prt("f1(float)"); }
void f1(double x) { prt("f1(double)"); }
void f2(byte x) { prt("f2(byte)"); }
void f2(short x) { prt("f2(short)"); }
void f2(int x) { prt("f2(int)"); }
void f2(long x) { prt("f2(long)"); }
void f2(float x) { prt("f2(float)"); }
void f2(double x) { prt("f2(double)"); }
void f3(short x) { prt("f3(short)"); }
void f3(int x) { prt("f3(int)"); }
void f3(long x) { prt("f3(long)"); }
void f3(float x) { prt("f3(float)"); }
void f3(double x) { prt("f3(double)"); }
void f4(int x) { prt("f4(int)"); }
void f4(long x) { prt("f4(long)"); }
void f4(float x) { prt("f4(float)"); }
void f4(double x) { prt("f4(double)"); }
void f5(long x) { prt("f5(long)"); }
void f5(float x) { prt("f5(float)"); }
void f5(double x) { prt("f5(double)"); }
void f6(float x) { prt("f6(float)"); }
void f6(double x) { prt("f6(double)"); }
void f7(double x) { prt("f7(double)"); }

void testConstVal() {
    prt("Testing with 5");
    f1(5); f2(5); f3(5); f4(5); f5(5); f6(5); f7(5);
}

void testChar() {
    char x = 'x';
    prt("char argument:");
    f1(x); f2(x); f3(x); f4(x); f5(x); f6(x); f7(x);
}

void testByte() {
    byte x = 0;
    prt("byte argument:");
    f1(x); f2(x); f3(x); f4(x); f5(x); f6(x); f7(x);
}

void testShort() {
    short x = 0;
    prt("short argument:");
    f1(x); f2(x); f3(x); f4(x); f5(x); f6(x); f7(x);
}

void testInt() {
    int x = 0;
    prt("int argument:");
    f1(x); f2(x); f3(x); f4(x); f5(x); f6(x); f7(x);
}

void testLong() {
    long x = 0;
    prt("long argument:");
    f1(x); f2(x); f3(x); f4(x); f5(x); f6(x); f7(x);
}

void testFloat() {
float x = 0;
prt("float argument:");
f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);

void testDouble() {
double x = 0;
prt("double argument:");
f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}

public static void main(String[] args) {
    PrimitiveOverloading p =
        new PrimitiveOverloading();
p.testConstVal();
p.testChar();
p.testByte();
p.testShort();
p.testInt();
p.testLong();
p.testFloat();
p.testDouble();
}
} ///:~

If you view the output of this program, you’ll see that the constant value 5 is treated as an int, so if an overloaded method is available that takes an int it is used. In all other cases, if you have a data type that is smaller than the argument in the method, that data type is promoted. char produces a slightly different effect, since if it doesn’t find an exact char match, it is promoted to int.

What happens if your argument is bigger than the argument expected by the overloaded method? A modification of the above program gives the answer:

//: Demotion.java
// Demotion of primitives and overloading

public class Demotion {
    static void prt(String s) {
        System.out.println(s);
    }

    void f1(char x) { prt("f1(char)"); }
    void f1(byte x) { prt("f1(byte)"); }
    void f1(short x) { prt("f1(short)"); }
    void f1(int x) { prt("f1(int)"); }
    void f1(long x) { prt("f1(long)"); }
    void f1(float x) { prt("f1(float)"); }
    void f1(double x) { prt("f1(double)"); }

    void f2(char x) { prt("f2(char)"); }
    void f2(byte x) { prt("f2(byte)"); }
    void f2(short x) { prt("f2(short)"); }
    void f2(int x) { prt("f2(int)"); }
    void f2(long x) { prt("f2(long)"); }
    void f2(float x) { prt("f2(float)"); }

    void f3(char x) { prt("f3(char)"); }
    void f3(byte x) { prt("f3(byte)"); }
    void f3(short x) { prt("f3(short)"); }
    void f3(int x) { prt("f3(int)"); }
}
void f3(long x) { prt("f3(long)"); }
void f4(char x) { prt("f4(char)"); }
void f4(byte x) { prt("f4(byte)"); }
void f4(short x) { prt("f4(short)"); }
void f4(int x) { prt("f4(int)"); }
void f5(char x) { prt("f5(char)"); }
void f5(byte x) { prt("f5(byte)"); }
void f5(short x) { prt("f5(short)"); }
void f6(char x) { prt("f6(char)"); }
void f6(byte x) { prt("f6(byte)"); }
void f7(char x) { prt("f7(char)"); }
void testDouble() {
    double x = 0;
    f1(x); f2((float)x); f3((long)x); f4((int)x);
    f5((short)x); f6((byte)x); f7((char)x);
}
public static void main(String[] args) {
    Demotion p = new Demotion();
    p.testDouble();
}
} ///:~

Here, the methods take narrower primitive values. If your argument is wider then you must cast to the necessary type using the type name in parentheses. If you don’t do this, the compiler will issue an error message.

You should be aware that this is a narrowing conversion, which means you might lose information during the cast. This is why the compiler forces you to do it – to flag the narrowing conversion.

**Overloading on return values**

It is common to wonder “Why only class names and method argument lists? Why not distinguish between methods based on their return values?” For example, these two methods, which have the same name and arguments, are easily distinguished from each other:

void f() {}
int f() {}

This works fine when the compiler can unequivocally determine the meaning from the context, as in `int x = f()`. However, you can call a method and ignore the return value; this is often referred to as calling a method for its side effect since you don’t care about the return value but instead want the other effects of the method call. So if you call the method this way:

f();
how can Java determine which `f()` should be called? And how could someone reading the code see it? Because of this sort of problem, you cannot use return value types to distinguish overloaded methods.

**Default constructors**

As mentioned previously, a default constructor is one without arguments, used to create a “vanilla object.” If you create a class that has no constructors, the compiler will automatically create a default constructor for you. For example:

```java
//: DefaultConstructor.java
class Bird {
    int i;
}

class DefaultConstructor {
    public static void main(String[] args) {
        Bird nc = new Bird(); // default!
    }
} ///:~
```

The line `new Bird();` creates a new object and calls the default constructor, even though one was not explicitly defined. Without it we would have no method to call to build our object. However, if you define any constructors (with or without arguments), the compiler will **not** synthesize one for you:

```java
class Bush {
    Bush(int i) {}  
    Bush(double d) {}
}
```

Now if you say:

```java
new Bush();
```

the compiler will complain that it cannot find a constructor that matches. It’s as if when you don’t put in any constructors, the compiler says “You are bound to need some constructor, so let me make one for you.” But if you write a constructor, the compiler says “You’ve written a constructor so you know what you’re doing; if you didn’t put in a default it’s because you meant to leave it out.”

**The this keyword**

If you have two objects of the same type called `a` and `b`, you might wonder how it is that you can call a method `f()` for both those objects:

```java
class Banana {  
    void f(int i) { /* ... */  }
}
Banana a = new Banana(), b = new Banana();
```
a.f(1);
b.f(2);

If there’s only one method called \texttt{f()}, how can that method know whether it’s being called for the object \texttt{a} or \texttt{b}?

To allow you to write the code in a convenient object-oriented syntax in which you “send a message to an object,” the compiler does some undercover work for you. There’s a secret first argument passed to the method \texttt{f()}, and that argument is the handle to the object that’s being manipulated. So the two method calls above become something like:

\begin{verbatim}
Banana.f(a, 1);
Banana.f(b, 2);
\end{verbatim}

This is internal and you can’t write these expressions and get the compiler to accept them, but it gives you an idea of what’s happening.

Suppose you’re inside a method and you’d like to get the handle to the current object. Since that handle is passed \textit{secretly} by the compiler, there’s no identifier for it. However, for this purpose there’s a keyword: \texttt{this}. The \texttt{this} keyword – which can be used only inside a method – produces the handle to the object the method has been called for. You can treat this handle just like any other object handle. Keep in mind that if you’re calling a method of your class from within another method of your class, you don’t need to use \texttt{this}; you simply call the method. The current \texttt{this} handle is automatically used for the other method. Thus you can say:

\begin{verbatim}
class Apricot {
  void pick() { /* ... */ }
  void pit() { pick(); /* ... */ }
}
\end{verbatim}

Inside \texttt{pit()}, you could say \texttt{this.pick()} but there’s no need to. The compiler does it for you automatically. The \texttt{this} keyword is used only for those special cases in which you need to explicitly use the handle to the current object. For example, it’s often used in \texttt{return} statements when you want to return the handle to the current object:

\begin{verbatim}
//: Leaf.java
// Simple use of the "this" keyword

public class Leaf {
  private int i = 0;
  Leaf increment() {
    i++;
    return this;
  }
  void print() {
    System.out.println("i = " + i);
  }
  public static void main(String[] args) {
    Leaf x = new Leaf();
    x.increment().increment().increment().print();
  }
  } //://~
\end{verbatim}
Because `increment()` returns the handle to the current object via the `this` keyword, multiple operations can easily be performed on the same object.

**Calling constructors from constructors**

When you write several constructors for a class, there are times when you’d like to call one constructor from another to avoid duplicating code. You can do this using the `this` keyword.

Normally, when you say `this`, it is in the sense of “this object” or “the current object,” and by itself it produces the handle to the current object. In a constructor, the `this` keyword takes on a different meaning when you give it an argument list: it makes an explicit call to the constructor that matches that argument list. Thus you have a straightforward way to call other constructors:

```java
//: Flower.java
// Calling constructors with "this"

public class Flower {
  private int petalCount = 0;
  private String s = new String("null");
  Flower(int petals) {
    petalCount = petals;
    System.out.println(
      "Constructor w/ int arg only, petalCount= "
      + petalCount);
  }
  Flower(String ss) {
    System.out.println(
      "Constructor w/ String arg only, s=" + ss);
    s = ss;
  }
  Flower(String s, int petals) {
    this(petals);
    // Can't call two!
    this.s = s; // Another use of "this"
    System.out.println("String & int args");
  }
  Flower() {
    this("hi", 47);
    System.out.println(
      "default constructor (no args)");
  }
  void print() {
    this(11); // Not inside non-constructor!
    System.out.println(
      "petalCount = " + petalCount + " s = "+ s);
  }
  public static void main(String[] args) {
    Flower x = new Flower();
    x.print();
  }
} ///:~
```

The constructor `Flower(String s, int petals)` shows that, while you can call one constructor using `this`, you cannot call two. In addition, the constructor call must be the first thing you do or you’ll get a compiler error message.
This example also shows another way you’ll see this used. Since the name of the argument s and the name of the member data s are the same, there’s an ambiguity. You can resolve it by saying this.s to refer to the member data. You’ll often see this form used in Java code, and it’s used in numerous places in this book.

In print( ) you can see that the compiler won’t let you call a constructor from inside any method other than a constructor.

The meaning of static

With the this keyword in mind, you can more fully understand what it means to make a method static. It means that there is no this for that particular method. You cannot call non-static methods from inside static methods (although the reverse is possible), and you can call a static method for the class itself, without any object. In fact, that’s primarily what a static method is for. It’s as if you’re creating the equivalent of a global function (from C). Except global functions are not permitted in Java, and putting the static method inside a class allows it access to other static methods and to static fields.

Some people argue that static methods are not object-oriented since they do have the semantics of a global function; with a static method you don’t send a message to an object, since there’s no this. This is probably a fair argument, and if you find yourself using a lot of static methods you should probably rethink your strategy. However, statics are pragmatic and there are times when you genuinely need them, so whether or not they are “proper OOP” should be left to the theoreticians. Indeed, even Smalltalk has the equivalent in its “class methods.”

Cleanup: finalization and garbage collection

Programmers know about the importance of initialization, but often forget the importance of cleanup. After all, who needs to clean up an int? But with libraries, simply “letting go” of an object once you’re done with it is not always safe. Of course, Java has the garbage collector to reclaim the memory of objects that are no longer used. Now consider a very special and unusual case. Suppose your object allocates “special” memory without using new. The garbage collector knows only how to release memory allocated with new, so it won’t know how to release the object’s “special” memory. To handle this case, Java provides a method called finalize( ) that you can define for your class. Here’s how it’s supposed to work. When the garbage collector is ready to release the storage used for your object, it will first call finalize( ), and only on the next garbage-collection pass will it reclaim the object’s memory. So if you choose to use finalize( ), it gives you the ability to perform some important cleanup at the time of garbage collection.

This is a potential programming pitfall because some programmers, especially C++ programmers, might initially mistake finalize( ) for the destructor in C++, which is a function that is always called when an object is destroyed. But it is important to distinguish between C++ and Java here, because in C++ objects always get
destroyed (in a bug-free program), whereas in Java objects do not always get
garbage-collected. Or, put another way:

Garbage collection is not destruction.

If you remember this, you will stay out of trouble. What it means is that if there is
some activity that must be performed before you no longer need an object, you must
perform that activity yourself. Java has no destructor or similar concept, so you must
create an ordinary method to perform this cleanup. For example, suppose in the
process of creating your object it draws itself on the screen. If you don’t explicitly
erase its image from the screen, it might never get cleaned up. If you put some kind
of erasing functionality inside finalize(), then if an object is garbage-collected, the
image will first be removed from the screen, but if it isn’t, the image will remain. So a
second point to remember is:

Your objects might not get garbage collected.

You might find that the storage for an object never gets released because your
program never nears the point of running out of storage. If your program completes
and the garbage collector never gets around to releasing the storage for any of your
objects, that storage will be returned to the operating system en masse as the
program exits. This is a good thing, because garbage collection has some overhead,
and if you never do it you never incur that expense.

What is finalize() for?

You might believe at this point that you should not use finalize() as a general-
purpose cleanup method. What good is it?

A third point to remember is:

Garbage collection is only about memory.

That is, the sole reason for the existence of the garbage collector is to recover
memory that your program is no longer using. So any activity that is associated with
garbage collection, most notably your finalize() method, must also be only about
memory and its deallocation.

Does this mean that if your object contains other objects finalize() should explicitly
release those objects? Well, no – the garbage collector takes care of the release of all
object memory regardless of how the object is created. It turns out that the need for
finalize() is limited to special cases, in which your object can allocate some storage
in some way other than creating an object. But, you might observe, everything in
Java is an object so how can this be?

It would seem that finalize() is in place because of the possibility that you’ll do
something C-like by allocating memory using a mechanism other than the normal
one in Java. This can happen primarily through native methods, which are a way to
call non-Java code from Java. (Native methods are discussed in Appendix A.) C and
C++ are the only languages currently supported by native methods, but since they
can call subprograms in other languages, you can effectively call anything. Inside the non-Java code, C’s malloc() family of functions might be called to allocate storage, and unless you call free() that storage will not be released, causing a memory leak. Of course, free() is a C and C++ function, so you’d need call it in a native method inside your finalize().

After reading this, you probably get the idea that you won’t use finalize() much. You’re correct; it is not the appropriate place for normal cleanup to occur. So where should normal cleanup be performed?

You must perform cleanup

To clean up an object, the user of that object must call a cleanup method at the point the cleanup is desired. This sounds pretty straightforward, but it collides a bit with the C++ concept of the destructor. In C++, all objects are destroyed. Or rather, all objects should be destroyed. If the C++ object is created as a local, i.e. on the stack (not possible in Java), then the destruction happens at the closing curly brace of the scope in which the object was created. If the object was created using new (like in Java) the destructor is called when the programmer calls the C++ operator delete (which doesn’t exist in Java). If the programmer forgets, the destructor is never called and you have a memory leak, plus the other parts of the object never get cleaned up.

In contrast, Java doesn’t allow you to create local objects – you must always use new. But in Java, there’s no “delete” to call for releasing the object since the garbage collector releases the storage for you. So from a simplistic standpoint you could say that because of garbage collection, Java has no destructor. You’ll see as this book progresses, however, that the presence of a garbage collector does not remove the need or utility of destructors. (And you should never call finalize() directly, so that’s not an appropriate avenue for a solution.) If you want some kind of cleanup performed other than storage release you must still call a method in Java, which is the equivalent of a C++ destructor without the convenience.

One of the things finalize() can be useful for is observing the process of garbage collection. The following example shows you what’s going on and summarizes the previous descriptions of garbage collection:

```java
//: Garbage.java
// Demonstration of the garbage
// collector and finalization

class Chair {
    static boolean gcrun = false;
    static boolean f = false;
    static int created = 0;
    static int finalized = 0;
    int i;
    Chair() {
        i = ++created;
        if(created == 47)
            System.out.println("Created 47");
    }
    protected void finalize() {
        gcrun = true;
        f = true;
        System.out.println("Finalized");
        ++finalized;
    }
}
```
if(!gcrun) {
    gcrun = true;
    System.out.println(
        "Beginning to finalize after " +
        created + " Chairs have been created");
}
if(i == 47) {
    System.out.println(
        "Finalizing Chair #47, " +
        "Setting flag to stop Chair creation");
    f = true;
}
finalized++;
if(finalized >= created)
    System.out.println(
        "All " + finalized + " finalized");
}

public class Garbage {
    public static void main(String[] args) {
        if(args.length == 0) {
            System.err.println("Usage: 
" +
            "java Garbage before\n or:java Garbage after");
            return;
        }
        while(!Chair.f) {
            new Chair();
            new String("To take up space");
        }
        System.out.println(
            "After all Chairs have been created:
" +
            "total created = " + Chair.created +
            ", total finalized = " + Chair.finalized);
        if(args[0].equals("before")) {
            System.out.println("gc():");
            System.gc();
            System.out.println("runFinalization():");
            System.runFinalization();
        }
        System.out.println("bye!");
        if(args[0].equals("after"))
            System.runFinalizersOnExit(true);
    }
} ///:~

The above program creates many Chair objects, and at some point after the garbage collector begins running, the program stops creating Chairs. Since the garbage collector can run at any time, you don’t know exactly when it will start up, so there’s a flag called gcrun to indicate whether the garbage collector has started running yet. A second flag f is a way for Chair to tell the main() loop that it should stop making objects. Both of these flags are set within finalize(), which is called during garbage collection.

Two other static variables, created and finalized, keep track of the number of objs created versus the number that get finalized by the garbage collector. Finally, each Chair has its own (non-static) int i so it can keep track of what number it is.
When Chair number 47 is finalized, the flag is set to true to bring the process of Chair creation to a stop.

All this happens in main(), in the loop

```
while(!Chair.f) {
    new Chair();
    new String("To take up space");
}
```

You might wonder how this loop could ever finish, since there’s nothing inside that changes the value of Chair.f. However, the finalize() process will, eventually, when it finalizes number 47.

The creation of a String object during each iteration is simply extra garbage being created to encourage the garbage collector to kick in, which it will do when it starts to get nervous about the amount of memory available.

When you run the program, you provide a command-line argument of “before” or “after.” The “before” argument will call the System.gc() method (to force execution of the garbage collector) along with the System.runFinalization() method to run the finalizers. These methods were available in Java 1.0, but the runFinalizersOnExit() method that is invoked by using the “after” argument is available only in Java 1.1 and beyond. (Note you can call this method any time during program execution, and the execution of the finalizers is independent of whether the garbage collector runs).

The preceding program shows that, in Java 1.1, the promise that finalizers will always be run holds true, but only if you explicitly force it to happen yourself. If you use an argument that isn’t “before” or “after” (such as “none”), then neither finalization process will occur, and you’ll get an output like this:

```
Created 47
Beginning to finalize after 8694 Chairs have been created
Finalizing Chair #47, Setting flag to stop Chair creation
After all Chairs have been created:
total created = 9834, total finalized = 108
bye!
```

Thus, not all finalizers get called by the time the program completes. To force finalization to happen, you can call System.gc() followed by System.runFinalization(). This will destroy all the objects that are no longer in use up to that point. The odd thing about this is that you call gc() before you call runFinalization(), which seems to contradict the Sun documentation, which claims that finalizers are run first, and then the storage is released. However, if you call runFinalization() first, and then gc(), the finalizers will not be executed.

One reason that Java 1.1 might default to skipping finalization for all objects is because it seems to be expensive. When you use either of the approaches that force garbage collection you might notice longer delays than you would without the extra finalization.
Member initialization

Java goes out of its way to guarantee that any variable is properly initialized before it is used. In the case of variables that are defined locally to a method, this guarantee comes in the form of a compile-time error. So if you say:

```java
void f() {
    int i;
    i++;
}
```

You’ll get an error message that says that i might not have been initialized. Of course, the compiler could have given i a default value, but it’s more likely that this is a programmer error and a default value would have covered that up. Forcing the programmer to provide an initialization value is more likely to catch a bug.

If a primitive is a data member of a class, however, things are a bit different. Since any method can initialize or use that data, it might not be practical to force the user to initialize it to its appropriate value before the data is used. However, it’s unsafe to leave it with a garbage value, so each primitive data member of a class is guaranteed to get an initial value. Those values can be seen here:

```java
//: InitialValues.java
// Shows default initial values

class Measurement {
    boolean t;
    char c;
    byte b;
    short s;
    int i;
    long l;
    float f;
    double d;
    void print() {
        System.out.println("Data type Initial value\n" + 
            "boolean " + t + "\n" + 
            "char " + c + "\n" + 
            "byte " + b + "\n" + 
            "short " + s + "\n" + 
            "int " + i + "\n" + 
            "long " + l + "\n" + 
            "float " + f + "\n" + 
            "double " + d);
    }
}

public class InitialValues {
    public static void main(String[] args) {
        Measurement d = new Measurement();
        d.print();
        /* In this case you could also say:
        new Measurement().print();
        */
    }
} ///:~
```
The output of this program is:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>false</td>
</tr>
<tr>
<td>char</td>
<td></td>
</tr>
<tr>
<td>byte</td>
<td>0</td>
</tr>
<tr>
<td>short</td>
<td>0</td>
</tr>
<tr>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>long</td>
<td>0</td>
</tr>
<tr>
<td>float</td>
<td>0.0</td>
</tr>
<tr>
<td>double</td>
<td>0.0</td>
</tr>
</tbody>
</table>

The **char** value is a null, which doesn’t print.

You’ll see later that when you define an object handle inside a class without initializing it to a new object, that handle is given a value of null.

You can see that even though the values are not specified, they automatically get initialized. So at least there’s no threat of working with uninitialized variables.

**Specifying initialization**

What happens if you want to give a variable an initial value? One direct way to do this is simply to assign the value at the point you define the variable in the class. (Notice you cannot do this in C++, although C++ novices always try.) Here the field definitions in class **Measurement** are changed to provide initial values:

```java
class Measurement {
    boolean b = true;
    char c = 'x';
    byte B = 47;
    short s = 0xff;
    int i = 999;
    long l = 1;
    float f = 3.14f;
    double d = 3.14159;
    // . . .
}
```

You can also initialize non-primitive objects in this same way. If **Depth** is a class, you can insert a variable and initialize it like so:

```java
class Measurement {
    Depth o = new Depth();
    boolean b = true;
    // . . .
}
```

If you haven’t given **o** an initial value and you go ahead and try to use it anyway, you’ll get a run-time error called an exception (covered in Chapter 9).

You can even call a method to provide an initialization value:

```java
class CInit {
    int i = f();
    // . . .
}
```
This method can have arguments, of course, but those arguments cannot be other class members that haven’t been initialized yet. Thus, you can do this:

```java
class CInit {
    int i = f();
    int j = g(i);
    //...
}
```

But you cannot do this:

```java
class CInit {
    int j = g(i);
    int i = f();
    //...
}
```

This is one place in which the compiler, appropriately, does complain about forward referencing, since this has to do with the order of initialization and not the way the program is compiled.

This approach to initialization is simple and straightforward. It has the limitation that every object of type Measurement will get these same initialization values. Sometimes this is exactly what you need, but at other times you need more flexibility.

**Constructor initialization**

The constructor can be used to perform initialization, and this gives you greater flexibility in your programming since you can call methods and perform actions at run time to determine the initial values. There’s one thing to keep in mind, however: you aren’t precluding the automatic initialization, which happens before the constructor is entered. So, for example, if you say:

```java
class Counter {
    int i;
    Counter() { i = 7; }
    // . . .
}
```

then i will first be initialized to zero, then to 7. This is true with all the primitive types and with object handles, including those that are given explicit initialization at the point of definition. For this reason, the compiler doesn’t try to force you to initialize elements in the constructor at any particular place, or before they are used – initialization is already guaranteed.

**Order of initialization**

Within a class, the order of initialization is determined by the order that the variables are defined within the class. Even if the variable definitions are scattered throughout in between method definitions, the variables are initialized before any methods can be called – even the constructor. For example:

```java
//: OrderOfInitialization.java
// Demonstrates initialization order.
```
// When the constructor is called, to create a
// Tag object, you'll see a message:
class Tag {
    Tag(int marker) {
        System.out.println("Tag( + marker + ")");
    }
}
class Card {
    Tag t1 = new Tag(1); // Before constructor
    Card() {
        // Indicate we're in the constructor:
        System.out.println("Card()");
        t3 = new Tag(33); // Re-initialize t3
    }
    Tag t2 = new Tag(2); // After constructor
    void f() {
        System.out.println("f()");
    }
    Tag t3 = new Tag(3); // At end
}
public class OrderOfInitialization {
    public static void main(String[] args) {
        Card t = new Card();
        t.f(); // Shows that construction is done
    }
} ///:~

In Card, the definitions of the Tag objects are intentionally scattered about to prove that they'll all get initialized before the constructor is entered or anything else can happen. In addition, t3 is re-initialized inside the constructor. The output is:

Tag(1)
Tag(2)
Tag(3)
Card()
f()

Thus, the t3 handle gets initialized twice, once before and once during the constructor call. (The first object is dropped, so it can be garbage-collected later.) This might not seem efficient at first, but it guarantees proper initialization – what would happen if an overloaded constructor were defined that did not initialize t3 and there wasn’t a “default” initialization for t3 in its definition?

**Static data initialization**

When the data is static the same thing happens; if it’s a primitive and you don’t initialize it, it gets the standard primitive initial values. If it’s a handle to an object, it’s null unless you create a new object and attach your handle to it.

If you want to place initialization at the point of definition, it looks the same as for non-statics. But since there’s only a single piece of storage for a static, regardless of
how many objects are created the question of when that storage gets initialized arises. An example makes this question clear:

```java
//: StaticInitialization.java
// Specifying initial values in a
// class definition.

class Bowl {
    Bowl(int marker) {
        System.out.println("Bowl(" + marker + ")");
    }
    void f(int marker) {
        System.out.println("f(" + marker + ")");
    }
}

class Table {
    static Bowl b1 = new Bowl(1);
    Table() {
        System.out.println("Table()");
        b2.f(1);
    }
    void f2(int marker) {
        System.out.println("f2(" + marker + ")");
    }
    static Bowl b2 = new Bowl(2);
}

class Cupboard {
    Bowl b3 = new Bowl(3);
    static Bowl b4 = new Bowl(4);
    Cupboard() {
        System.out.println("Cupboard()");
        b4.f(2);
    }
    void f3(int marker) {
        System.out.println("f3(" + marker + ")");
    }
    static Bowl b5 = new Bowl(5);
}

public class StaticInitialization {
    public static void main(String[] args) {
        System.out.println("Creating new Cupboard() in main");
        new Cupboard();
        System.out.println("Creating new Cupboard() in main");
        new Cupboard();
        t2.f2(1);
        t3.f3(1);
    }
    static Table t2 = new Table();
    static Cupboard t3 = new Cupboard();
} //:~
```

Bowl allows you to view the creation of a class, and Table and Cupboard create static members of Bowl scattered through their class definitions. Note that
**Cupboard** creates a non-static **Bowl b3** prior to the static definitions. The output shows what happens:

```java
Bowl(1)
Bowl(2)
Table()
f(1)
Bowl(4)
Bowl(5)
Bowl(3)
Cupboard()
f(2)
Creating new Cupboard() in main
Bowl(3)
Cupboard()
f(2)
Creating new Cupboard() in main
Bowl(3)
Cupboard()
f(2)
f2(1)
f3(1)
```

The static initialization occurs only if it's necessary. If you don't create a **Table** object and you never refer to **Table.b1** or **Table.b2**, the static **Bowl b1** and **b2** will never be created. However, they are created only when the first **Table** object is created (or the first static access occurs). After that, the static object is not re-initialized.

The order of initialization is **statics** first, if they haven't already been initialized by a previous object creation, and then the non-static objects. You can see the evidence of this in the output.

It's helpful to summarize the process of creating an object. Consider a class called **Dog**:

1. The first time an object of type **Dog** is created, or the first time a static method or static field of class **Dog** is accessed, the Java interpreter must locate **Dog.class**, which it does by searching through the classpath.
2. As **Dog.class** is loaded (which creates a **Class** object, which you'll learn about later), all of its static initializers are run. Thus, static initialization takes place only once, as the **Class** object is loaded for the first time.
3. When you create a new **Dog()**, the construction process for a **Dog** object first allocates enough storage for a **Dog** object on the heap.
4. This storage is wiped to zero, automatically setting all the primitives in **Dog** to their default values (zero for numbers and the equivalent for **boolean** and **char**).
5. Any initializations that occur at the point of field definition are executed.
6. Constructors are executed. As you shall see in Chapter 6, this might actually involve a fair amount of activity, especially when inheritance is involved.

**Explicit static initialization**

Java allows you to group other static initializations inside a special “static construction clause” (sometimes called a static block) in a class. It looks like this:
class Spoon {
    static int i;
    static {
        i = 47;
    }
    // . . .
}

So it looks like a method, but it's just the static keyword followed by a method body. This code, like the other static initialization, is executed only once, the first time you make an object of that class or you access a static member of that class (even if you never make an object of that class). For example:

//: ExplicitStatic.java
// Explicit static initialization
// with the "static" clause.

class Cup {
    Cup(int marker) {
        System.out.println("Cup(" + marker + ")");
    }
    void f(int marker) {
        System.out.println("f(" + marker + ")");
    }
}

class Cups {
    static Cup c1;
    static Cup c2;
    static {
        c1 = new Cup(1);
        c2 = new Cup(2);
    }
    Cups() {
        System.out.println("Cups()");
    }
}

public class ExplicitStatic {
    public static void main(String[] args) {
        System.out.println("Inside main()");
        Cups.c1.f(99);  // (1)
    }
    static Cups x = new Cups();  // (2)
    static Cups y = new Cups();  // (2)
} ///:~

The static initializers for Cups will be run when either the access of the static object c1 occurs on the line marked (1), or if line (1) is commented out and the lines marked (2) are uncommented. If both (1) and (2) are commented out, the static initialization for Cups never occurs.

Non-static instance initialization

Java 1.1 provides a similar syntax for initializing non-static variables for each object. Here's an example:

//: Mugs.java

class Mugs {
    static int marker;  // (1)
    static {
        marker = 99;
    }
    Mugs(int marker) {
        System.out.println("Mugs(int marker) "+ marker);
    }
    void foo() {
        System.out.println("foo()");
    }
}

public class Mugs {
    public static void main(String[] args) {
        Mugs m1 = new Mugs(99);  // (2)
        m1.foo();  // (2)
    }
    static Mugs x = new Mugs();  // (2)
    static Mugs y = new Mugs();  // (2)
} ///:~

The static initializers for Mugs will be run when either the access of the static object marker occurs on the line marked (1), or if line (1) is commented out and the lines marked (2) are uncommented. If both (1) and (2) are commented out, the static initialization for Mugs never occurs.
// Java 1.1 "Instance Initialization"

class Mug {
    Mug(int marker) {
        System.out.println("Mug(" + marker + ")");
    }
    void f(int marker) {
        System.out.println("f(" + marker + ")");
    }
}

class Mugs {
    Mug c1;
    Mug c2;
    {
        c1 = new Mug(1);
        c2 = new Mug(2);
        System.out.println("c1 & c2 initialized");
    }
    Mugs() {
        System.out.println("Mugs()");
    }
    public static void main(String[] args) {
        System.out.println("Inside main()");
        Mugs x = new Mugs();
    }
} ///:~

You can see that the instance initialization clause:

```java
{
    c1 = new Mug(1);
    c2 = new Mug(2);
    System.out.println("c1 & c2 initialized");
}
```

looks exactly like the static initialization clause except for the missing `static` keyword. This syntax is necessary to support the initialization of anonymous inner classes (see Chapter 7).

**Array initialization**

Initializing arrays in C is error-prone and tedious. C++ uses aggregate initialization to make it much safer. Java has no “aggregates” like C++, since everything is an object in Java. It does have arrays, and these are supported with array initialization.

An array is simply a sequence of either objects or primitives, all the same type and packaged together under one identifier name. Arrays are defined and used with the square-brackets `indexing operator` `[]`. To define an array you simply follow your type name with empty square brackets:

```java
int[] a1;
```

You can also put the square brackets after the identifier to produce exactly the same meaning:
int a1[];

This conforms to expectations from C and C++ programmers. The former style, however, is probably a more sensible syntax, since it says that the type is “an int array.” That style will be used in this book.

The compiler doesn’t allow you to tell it how big the array is. This brings us back to that issue of “handles.” All that you have at this point is a handle to an array, and there’s been no space allocated for the array. To create storage for the array you must write an initialization expression. For arrays, initialization can appear anywhere in your code, but you can also use a special kind of initialization expression that must occur at the point where the array is created. This special initialization is a set of values surrounded by curly braces. The storage allocation (the equivalent of using new) is taken care of by the compiler in this case. For example:

int[] a1 = { 1, 2, 3, 4, 5 };

So why would you ever define an array handle without an array?

int[] a2;

Well, it’s possible to assign one array to another in Java, so you can say:

a2 = a1;

What you’re really doing is copying a handle, as demonstrated here:

//: Arrays.java
//: Arrays of primitives.

public class Arrays {
    public static void main(String[] args) {
        int[] a1 = { 1, 2, 3, 4, 5 };
        int[] a2;
        a2 = a1;
        for(int i = 0; i < a2.length; i++)
            a2[i]++;
        for(int i = 0; i < a1.length; i++)
            System.out.println("a1[" + i + "] = " + a1[i]);
    }
    static void prt(String s) {
        System.out.println(s);
    }
} ///:~

You can see that a1 is given an initialization value while a2 is not; a2 is assigned later – in this case, to another array.

There’s something new here: all arrays have an intrinsic member (whether they’re arrays of objects or arrays of primitives) that you can query – but not change – to tell you how many elements there are in the array. This member is length. Since arrays in Java, like C and C++, start counting from element zero, the largest element you can index is length - 1. If you go out of bounds, C and C++ quietly accept this and allow you to stomp all over your memory, which is the source of many infamous
bugs. However, Java protects you against such problems by causing a run-time error (an exception, the subject of Chapter 9) if you step out of bounds. Of course, checking every array access costs time and code and there’s no way to turn it off, which means that array accesses might be a source of inefficiency in your program if they occur at a critical juncture. For Internet security and programmer productivity, the Java designers thought that this was a worthwhile tradeoff.

What if you don’t know how many elements you’re going to need in your array while you’re writing the program? You simply use new to create the elements in the array. Here, new works even though it’s creating an array of primitives (new won’t create a non-array primitive):

```
//: ArrayNew.java
// Creating arrays with new.
import java.util.*;

class ArrayNew {
    static Random rand = new Random();
    static int pRand(int mod) {
        return Math.abs(rand.nextInt()) % mod + 1;
    }
    public static void main(String[] args) {
        int[] a;
        a = new int[pRand(20)];
        prt("length of a = " + a.length);
        for(int i = 0; i < a.length; i++)
            prt("a[" + i + "] = " + a[i]);
    }
    static void prt(String s) {
        System.out.println(s);
    }
}
``` //:~

Since the size of the array is chosen at random (using the pRand( ) method defined earlier), it’s clear that array creation is actually happening at run-time. In addition, you’ll see from the output of this program that array elements of primitive types are automatically initialized to “empty” values. (For numerics, this is zero, for char, it’s null, and for boolean, it’s false.)

Of course, the array could also have been defined and initialized in the same statement:

```
int[] a = new int[pRand(20)];
``` 

If you’re dealing with an array of non-primitive objects, you must always use new. Here, the handle issue comes up again because what you create is an array of handles. Consider the wrapper type Integer, which is a class and not a primitive:

```
//: ArrayClassObj.java
// Creating an array of non-primitive objects.
import java.util.*;

class ArrayClassObj {
    static Random rand = new Random();
    static int pRand(int mod) {
```
Here, even after `new` is called to create the array:

```java
Integer[] a = new Integer[pRand(20)];
```

it's only an array of handles, and not until the handle itself is initialized by creating a new `Integer` object is the initialization complete:

```java
a[i] = new Integer(pRand(500));
```

If you forget to create the object, however, you'll get an exception at run-time when you try to read the empty array location.

Take a look at the formation of the `String` object inside the print statements. You can see that the handle to the `Integer` object is automatically converted to produce a `String` representing the value inside the object.

It's also possible to initialize arrays of objects using the curly-brace-enclosed list. There are two forms, the first of which is the only one allowed in Java 1.0. The second (equivalent) form is allowed starting with Java 1.1:

```java
//: ArrayInit.java
// Array initialization

public class ArrayInit {
    public static ArrayInit {
        public static void main(String[] args) {
            Integer[] a = {
                new Integer(1),
                new Integer(2),
                new Integer(3),
            };

            // Java 1.1 only:
            Integer[] b = new Integer[] {
                new Integer(1),
                new Integer(2),
                new Integer(3),
            };
        }
    } ///:~
```
This is useful at times, but it’s more limited since the size of the array is determined at compile time. The final comma in the list of initializers is optional. (This feature makes for easier maintenance of long lists.)

The second form of array initialization, added in Java 1.1, provides a convenient syntax to create and call methods that can produce the same effect as C’s variable argument lists (known as “varargs” in C). These included, if you choose, unknown quantity of arguments as well as unknown type. Since all classes are ultimately inherited from the common root class `Object`, you can create a method that takes an array of `Object` and call it like this:

```java
//: VarArgs.java
// Using the Java 1.1 array syntax to create
// variable argument lists

class A { int i; }

public class VarArgs {
    static void f(Object[] x) {
        for(int i = 0; i < x.length; i++)
            System.out.println(x[i]);
    }
    public static void main(String[] args) {
        f(new Object[] {
            new Integer(47), new VarArgs(),
            new Float(3.14), new Double(11.11) });
        f(new Object[] {"one", "two", "three" });
        f(new Object[] {new A(), new A(), new A()});
    }
} ///:~
```

At this point, there’s not much you can do with these unknown objects, and this program uses the automatic `String` conversion to do something useful with each `Object`. In Chapter 11 (run-time type identification or RTTI) you’ll learn how to discover the exact type of such objects so that you can do something more interesting with them.

### Multidimensional arrays

Java allows you to easily create multidimensional arrays:

```java
//: MultiDimArray.java
// Creating multidimensional arrays.
import java.util.*;

public class MultiDimArray {
    static Random rand = new Random();
    static int pRand(int mod) {
        return Math.abs(rand.nextInt()) % mod + 1;
    }
    public static void main(String[] args) {
        int[][] a1 = {
            { 1, 2, 3, },
            { 4, 5, 6, },
        };
        for(int i = 0; i < a1.length; i++)
```
for(int j = 0; j < a1[i].length; j++)
    prt("a1[" + i + "]" + j + "] = " + a1[i][j]);
// 3-D array with fixed length:
int[][][] a2 = new int[2][2][4];
for(int i = 0; i < a2.length; i++)
    for(int j = 0; j < a2[i].length; j++)
        for(int k = 0; k < a2[i][j].length; k++)
            prt("a2[" + i + "]" + j + "]" + k + "] = " + a2[i][j][k]);
// 3-D array with varied-length vectors:
int[][][] a3 = new int[pRand(7)][5][4];
for(int i = 0; i < a3.length; i++) {
    a3[i] = new int[pRand(5)][4];
    for(int j = 0; j < a3[i].length; j++)
        for(int k = 0; k < a3[i][j].length; k++)
            prt("a3[" + i + "]" + j + "]" + k + "] = " + a3[i][j][k]);
}
// Array of non-primitive objects:
Integer[][] a4 = {
    { new Integer(1), new Integer(2) },
    { new Integer(3), new Integer(4) },
    { new Integer(5), new Integer(6) },
};
for(int i = 0; i < a4.length; i++)
    for(int j = 0; j < a4[i].length; j++)
        prt("a4[" + i + "]" + j + "] = " + a4[i][j]);
Integer[][] a5;
a5 = new Integer[3][5];
for(int i = 0; i < a5.length; i++) {
    a5[i] = new Integer(3);    
    for(int j = 0; j < a5[i].length; j++)
        a5[i][j] = new Integer(i*j);
}
for(int i = 0; i < a5.length; i++)
    for(int j = 0; j < a5[i].length; j++)
        prt("a5[" + i + "]" + j + "] = " + a5[i][j]);
}
static void prt(String s) {
    System.out.println(s);
}
} ///:~

The code used for printing uses length so that it doesn’t depend on fixed array sizes.

The first example shows a multidimensional array of primitives. You delimit each vector in the array with curly braces:
Each set of square brackets moves you into the next level of the array.

The second example shows a three-dimensional array allocated with `new`. Here, the whole array is allocated at once:

```java
int[][][] a2 = new int[2][2][4];
```

But the third example shows that each vector in the arrays that make up the matrix can be of any length:

```java
int[][][] a3 = new int[pRand(7)][()][];
for(int i = 0; i < a3.length; i++) {
    a3[i] = new int[pRand(5)][];
    for(int j = 0; j < a3[i].length; j++)
        a3[i][j] = new int[pRand(5)];
}
```

The first `new` creates an array with a random-length first element and the rest undetermined. The second `new` inside the for loop fills out the elements but leaves the third index undetermined until you hit the third `new`.

You will see from the output that array values are automatically initialized to zero if you don’t give them an explicit initialization value.

You can deal with arrays of non-primitive objects in a similar fashion, which is shown in the fourth example, demonstrating the ability to collect many `new` expressions with curly braces:

```java
Integer[][] a4 = {
    { new Integer(1), new Integer(2) },
    { new Integer(3), new Integer(4) },
    { new Integer(5), new Integer(6) },
};
```

The fifth example shows how an array of non-primitive objects can be built up piece by piece:

```java
Integer[][] a5;
a5 = new Integer[3][];
for(int i = 0; i < a5.length; i++) {
    a5[i] = new Integer[3];
    for(int j = 0; j < a5[i].length; j++)
        a5[i][j] = new Integer(i*j);
}
```

The `i*j` is just to put an interesting value into the `Integer`.

**Summary**
The seemingly elaborate mechanism for initialization, the constructor, should give you a strong hint about the critical importance placed on initialization in the language. As Stroustrup was designing C++, one of the first observations he made about productivity in C was that improper initialization of variables causes a significant portion of programming problems. These kinds of bugs are hard to find, and similar issues apply to improper cleanup. Because constructors allow you to guarantee proper initialization and cleanup (the compiler will not allow an object to be created without the proper constructor calls), you get complete control and safety.

In C++, destruction is quite important because objects created with new must be explicitly destroyed. In Java, the garbage collector automatically releases the memory for all objects, so the equivalent cleanup method in Java isn’t necessary much of the time. In cases where you don’t need destructor-like behavior, Java’s garbage collector greatly simplifies programming, and adds much-needed safety in managing memory. Some garbage collectors are even cleaning up other resources like graphics and file handles. However, the garbage collector does add a run-time cost, the expense of which is difficult to put into perspective because of the overall slowness of Java interpreters at this writing. As this changes, we’ll be able to discover if the overhead of the garbage collector will preclude the use of Java for certain types of programs. (One of the issues is the unpredictability of the garbage collector.)

Because of the guarantee that all objects will be constructed, there’s actually more to the constructor than what is shown here. In particular, when you create new classes using either composition or inheritance the guarantee of construction also holds, and some additional syntax is necessary to support this. You’ll learn about composition, inheritance and how they affect constructors in future chapters.

**Exercises**

1. Create a class with a default constructor (one that takes no arguments) that prints a message. Create an object of this class.
2. Add an overloaded constructor to Exercise 1 that takes a String argument and prints it along with your message.
3. Create an array of object handles of the class you created in Exercise 2, but don’t actually create objects to assign into the array. When you run the program, notice whether the initialization messages from the constructor calls are printed.
4. Complete Exercise 3 by creating objects to attach to the array of handles.
5. Experiment with Garbage.java by running the program using the arguments “before,” “after” and “none.” Repeat the process and see if you detect any patterns in the output. Change the code so that System.runFinalization() is called before System.gc() and observe the results.