The Essence of OOP using Java, Instance Initializers

Baldwin explains the use of instance initializers, static initializers in conjunction with constructors, and the initialization of ordinary instance variables. He explains and demonstrates the order in which they are executed.

Published: August 19, 2003
By Richard G. Baldwin

Java Programming Notes # 1634

Preface

This series of lessons is designed to teach you about the essence of Object-Oriented Programming *(OOP)* using Java.

The first lesson in the series was entitled The Essence of OOP Using Java, Objects, and Encapsulation. The previous lesson was entitled The Essence of OOP using Java, Static Initializer Blocks.

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different figures and listings while you are reading about them.

For further reading, see my extensive collection of online Java tutorials at Gamelan.com. A consolidated index is available at www.DickBaldwin.com.

Preview

Proper initialization is important

As I mentioned in the previous lesson in this series, proper initialization of variables is an important aspect of programming. Unlike other programming languages, it is not possible to

write a Java program in which the variables are accidentally initialized with the garbage left over in memory from the programs that previously ran in the computer.

Automatic initialization to default values

Instance variables and class *(static)* variables are automatically initialized to standard default values if you fail to purposely initialize them. Although local variables are not automatically initialized, you cannot compile a program that fails to either initialize a local variable or assign a value to that local variable before it is used.

Thus, Java programmers are prevented from committing the cardinal sin of allowing their variables to be initialized with random garbage through programming negligence.

Initialization during declaration

You should already know that you can initialize instance variables and class variables when you declare them, by including an initialization expression in the variable declaration statement. Figure 1 shows an example of a primitive instance variable named **simpleInitTime** that is purposely initialized to a **long** value obtained by invoking a static method of the **Init02** class named **relTime**. *(You will learn more about this method later.)*

```
  long simpleInitTime =
Init02.relTime();
```

Figure 1

While the expression in Figure 1 is a little complex, it is still just an expression and is perfectly suitable for use in initializing an instance variable when it is declared.

Constructor

What if your initialization requirements are more complex than can be satisfied with a single initialization expression? You should already know that you can write one or more overloaded constructors to purposely initialize all instance variables when an object is instantiated from the class. The code in a constructor can be as complex as you need it to be.

Not all classes allow constructors

What you may not know, however, is that you cannot always write a constructor for a class when you define it. For example, anonymous classes, which we will study in a subsequent lesson, do not allow the definition of a constructor.

However, even anonymous classes allow you to write *instance initializer blocks* when you define the class. The code in an instance initializer block, which can also be quite complex, is executed when an object is instantiated from the class.

Not as powerful as a constructor

You can write any number of instance initializer blocks in your class definition. However, unlike constructors, instance initializer blocks do not receive parameters. Therefore, they are less powerful than constructors. In terms of power, instance initializer blocks fall between simple initialization expressions *(such as that shown in Figure 1)* and constructors. Because they can execute complex code, they are more powerful than simple initialization expressions. Because they cannot receive parameters, they are less powerful than constructors.

Similar to noarg constructors

An instance initializer block is similar to a constructor that doesn't receive any parameters, except that you can write any number of instance initializer blocks into your class definition, and you can only write one *noarg* constructor in your class definition.

The order of execution

The code in an instance initializer block is executed after the constructor for the superclass is executed, and before the constructor for the class to which the initializer belongs is executed.

If the class definition contains a combination of instance initializer blocks in combination with the declaration of instance variables with initialization expressions, the code that comprises those items is executed in the order in which it appears in the class definition.

The order of execution of instance initializers in combination with instance variable initializations, constructors, and static initializer blocks will be illustrated in the sample program that I will discuss later in this lesson.

What does Flanagan have to say?

Here is how one of my favorite authors, David Flanagan of Java in a Nutshell fame, summarizes the situation:

> *"An instance initializer is simply a block of code inside curly braces that is embedded in a class definition, where a field or method definition normally appears. A class (any class -- not just anonymous classes) can have any number of instance initializers. The instance initializers and any variable initializers that appear in field definitions for the class are executed in the order they appear in the Java source code. These initializers are automatically run after the superclass constructor has returned but before the constructor, if any, of the current class runs."*

Why do we need instance initializers?

Flanagan goes on to explain the value of instance initializers, not only for anonymous classes, but also for non-anonymous classes. According to Flanagan,

"Instance initializers allow you to initialize an object's fields near the definition of those fields, rather than deferring that initialization to a constructor defined further away in the class. Used in this way, they can sometimes improve code readability."

The sample program that I will discuss in the next section will illustrate many aspects of instance initializers in combination with static initializer blocks, constructors, and simple instance variable initializations.

Discussion and Sample Code

I will discuss and explain a Java program named **Init02** in this lesson. *(A complete listing of the program is provided in Listing 15 near the end of the lesson.)* As mentioned above, this program illustrates many aspects of instance initializers in combination with static initializer blocks, constructors, and simple instance variable initializations.

Description of the program

Instance initializers behave much like *noarg* constructors. They are particularly useful for anonymous classes, which are not allowed to define any constructors, even those that take no arguments. However, the syntax for anonymous classes, even in the absence of instance initializers, is very cryptic. Therefore, I decided to explain and illustrate instance initializers in the context of ordinary top-level classes rather than to combine that explanation with the explanation of anonymous classes in a subsequent lesson.

The class hierarchy

This program defines a class named **B** that extends a class named **A**. Parameterized constructors are used in both **A** and **B** to instantiate an object of the class named **B**.

The base time is recorded

The controlling class defines and initializes a static variable containing the time that the program starts running in milliseconds relative to 00:00:00 GMT, Jan 1, 1970. This value is used as the base for computing time intervals later as the execution of the program progresses.

The times that are computed and displayed later are in milliseconds relative to the time at which the program started running.

The class loading process

Static initializers are defined in both **A** and **B** to display the time that the two classes are loaded and the order in which they are loaded. You will see that both classes are loaded when an attempt is made to instantiate an object of the subclass **B**. You will also see that the superclass is loaded before the subclass is loaded, and both are loaded before the object is instantiated.

An initialized instance variable

An instance variable is defined in the class named **B** and is initialized *(using a simple initialization expression)* with the time in milliseconds that the variable is initialized. In addition, two separate instance initializers are defined in the class named **B** that perform initialization after the constructor for **A** completes execution and before the constructor for **B** begins execution.

In terms of physical location, the instance variable mentioned above follows the first of the two instance initializers and appears before the second instance initializer in the class definition.

The order of execution

The first of the two instance initializers executes before the instance variable is initialized. The second of the two initializers executes after the instance variable is initialized, demonstrating that initialization based on simple initialization expressions and instance initializers occurs in the order that the code appears in the class definition.

Initialization time is displayed

The two constructors *(for classes **A** and **B**)* and the two initializers each display time information when they are executed to show the order in which the constructors and the initializers are executed.

Two separate objects

Two separate instances *(objects)* of the class named **B** are created, showing not only the order in which the instance initializers and the constructors are executed, but also showing that the static initializers are executed *one time only* when the classes are first loaded.

Display values of instance variables

Each time an object of the class named **B** is instantiated, an instance method of the class is invoked to display the values of the instance variables initialized during the process of instantiating the object.

One-hundred millisecond delays

Several one-hundred millisecond time delays are purposely inserted at strategic points within the program to force the time intervals between the different steps in the program to be measurable. Otherwise, the time intervals between steps would be so small that it would not be possible to distinguish between them on the basis of time recorded in milliseconds.

Will discuss in fragments

I will discuss the program code in fragments. In discussing the fragments, I will present much of the code in the order that it is executed, which is not necessarily the same order that the code appears in the program.

As mentioned earlier, a complete listing of the program can be viewed in Listing 15 near the end of the lesson.

Two utility methods

I will begin by presenting two **static** utility methods that are used to simplify the code in the body of the program. Both of these methods are defined in the controlling class named **Init02**.

Relative time in milliseconds

The utility method named **relTime**, shown in Listing 1, is used to compute and return the current time in milliseconds relative to a time value stored in a **static** variable of the controlling class named **baseTime**. As you will see later, **baseTime** contains the time that the program started running. Thus, each time this method is called, it returns the current time relative to the time that the program started running.

```
  static long relTime(){
    return ((new Date().getTime()) -
baseTime);
  }//end printTime

Listing 1
```

I relegated this code to a utility method simply due to the length and complexity of the expression, and the large number of times that the relative time is needed throughout the program.

Insert a delay

The utility method shown in Listing 2 causes the current thread to sleep for one-hundred milliseconds. Thus, each time this method is called, it inserts a one-hundred millisecond delay in the execution of the program.

```
  static void delay(){
    try{

Thread.currentThread().sleep(100);
    }catch(Exception
e){e.printStackTrace();}
  }//end delay

Listing 2
```

This method is also called numerous times throughout the program. Once again, therefore, I relegated this code to a utility method to simplify the code in the body of the program.

Establish the start time

Listing 3 shows the beginning of the controlling class named **Init02**, including the declaration and initialization of the class variable named **baseTime**.

```
public class Init02{
  static long baseTime = new
Date().getTime();

Listing 3
```

According to the Sun documentation, the **getTime** method of the **Date** class in Listing 3 *"Returns the number of milliseconds since January 1, 1970, 00:00:00 GMT represented by this Date object."*
Thus, this variable will contains the time in milliseconds that the class named **Init02** is loaded, which is also the time that the program starts running. This time will serve as the base time against which various time intervals will be computed during the running of the program.

The main method

The code in Listing 4 shows the beginning of the **main** method, which displays the current time relative to the start time, and instantiates a new object of the class named **B**. In addition, the code in Listing 4 invokes the method named **showData** on the new object. The **showData** method displays the values stored in several instance values as various initialization steps are completed during the instantiation of the object.

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

    System.out.println("Instantiate
first obj: "
                                  +
relTime());

    new B("Construct 1").showData();

Listing 4
```

The output

As you have probably already guessed, the print statement in Listing 4 produces the output shown in Figure 2. In other words, there was less than one millisecond of elapsed time between the initialization of the **static** variable named **baseTime** and the invocation of the **relTime** method in Listing 4.

```
Instantiate first obj: 0
Figure 2
```

Load classes A and B

The instantiation of the new object in Listing 4 triggers a whole chain of events. I will discuss those events in the sequence in which they occur in the paragraphs that follow. The first pair of interesting events is the loading of the classes named **A** and **B**.

What does it mean to say that a class is loaded?

As I explained in a previous lesson, I can't provide a description of exactly what happens from a technical viewpoint. However, I can tell you what seems to happen from a functional viewpoint.

Functionally, an object of the class whose name is **Class** is created and saved in memory. This object represents the class that is being loaded *(in this case, two classes named **A** and **B** are loaded, so two separate **Class** objects are created).* From that point forward, all static members of the class are available to the program by referring to the name of the class and the name of the member. Other information about the class is also available by invoking methods, such as the method named **getSuperclass**, on a reference to the **Class** object.

As you will see, when an attempt is made to instantiate an object of the subclass named **B**, that class and its superclass named **A** are both loaded. Furthermore, the superclass named **A** is loaded first. The loading of both classes takes place before the other steps required to instantiate the object take place.

A static initializer in the class named A

Listing 5 shows the beginning of the class named **A**. The code in Listing 5 declares an instance variable named **aXstrTime**, which will be used later to record the time that the constructor for the class named **A** is executed. More important for this part of the discussion, however, is the static initializer shown in Listing 5.

```
class A{
  long aXstrTime;

  static{//This is a static
initializer.
    System.out.println("Class A
loaded: " +

Init02.relTime());
  }//End static initializer

Listing 5
```

You will recall from the previous lesson on static initializers that they execute *one time only* when the class is loaded. The static initializer in Listing 5 prints a message showing the time that the class named **A** is loaded.

Class A load time

On my machine the code in Listing 5 produced the output shown in Figure 3. Presumably, the ten-millisecond delay between the start of the program and the point in time that the class named

A was loaded was due primarily to the time required for the program to find the class file on the disk and to load it into memory. Your system may produce a different result depending on the speed of your computer.

```
Class A loaded: 10
```

Figure 3

A static initializer in the class named B

Listing 6 shows the beginning of the class named **B**. The code in Listing 6 declares three instance variables, which will be used later to record the time that the constructor and the instance initializers are executed. More important for this part of the discussion, however, is the static initializer shown in Listing 6.

```
class B extends A{
  long bXstrTime;
  long init1Time;
  long init2Time;

  static{//This is a static
initializer.
    Init02.delay();

    System.out.println("Class B
loaded: " +

Init02.relTime() + "\n");
  }//End static initializer

Listing 6
```

The output

The static initializer in Listing 6 purposely inserts a one-hundred millisecond delay and then prints the time that it finishes executing. This is the time that the class named **B** finishes loading.

```
Class B loaded: 110
```

Figure 4

Figure 4 shows that the class named **B** was loaded immediately following the loading of the class named **A**.

Create the new object of the class named B

After the classes are loaded, the system proceeds to create the new instance of the class named **B**. You should recall however that objects are actually created beginning with the contribution from the class named **Object**, and proceeding down the inheritance hierarchy to the class from which the object is actually being instantiated. Thus, the next identifiable significant event is the execution of the constructor for the class named **A**, which is the superclass of the class named **B**. *(Nothing in the program makes it possible for us to identify the construction of that portion of the object attributable to the superclass named **Object**.)*

The constructor for the class named A

Listing 7 shows the constructor for the class named **A**. The code in the constructor purposely inserts a one-hundred millisecond delay, and then gets and saves the time that the constructor is executed. *(The time is saved in the instance variable named **aXstrTime**, which was declared in Listing 5 earlier.)*

```
  A(String str){//constructor

    Init02.delay();

    aXstrTime = Init02.relTime();
    System.out.println(str + "A:  " +
aXstrTime);
  }//end constructor for A
```

Listing 7

Then the code in the constructor prints that time, producing the output shown in Figure 5.

```
Construct 1A:  210
```

Figure 5

The important point here is that the constructor for the superclass is executed after the classes named **A** and **B** are loaded, but before the instance initializers for the subclass named **B** are executed.

Instance initializers in the subclass named B

If you examine Listing 15 near the end of the lesson you will see that the class named **B** contains two separate instance initializers, which are physically separated by an ordinary instance variable declaration *(with initialization)* and a constructor. As you will see in the discussion that follows, the execution of the first instance initializer follows the execution of the constructor for the superclass named **A** shown in Listing 7.

Then the initialization of the ordinary instance variable takes place, following the execution of the first instance initializer. This is followed by the execution of the second instance initializer.

Despite their physical placement in the code, the execution of both instance initializers and the initialization of the ordinary instance variable all take place before the constructor for the class named **B** is executed.

The first instance initializer

The first instance initializer is shown in Listing 8. Once again, as described by David Flanagan, "*An instance initializer is simply a block of code inside curly braces that is embedded in a class definition, where a field or method definition normally appears. ... The instance initializers and any variable initializers that appear in field definitions for the class are executed in the order*

they appear in the Java source code. These initializers are automatically run after the superclass constructor has returned but before the constructor, if any, of the current class runs."

```
  {//This is an instance initializer
    Init02.delay();

    init1Time = Init02.relTime();

    System.out.println("Initializer-1:
" +

init1Time);

    Init02.delay();
  }//end instance initializer
```

Listing 8

Insert a delay

The code in the instance initializer in Listing 8 begins by inserting a one-hundred millisecond delay to force the time interval between the execution of the constructor for the class named **A** and the execution of the instance initializer to be distinguishable.

Get, save, and display the relative time

After sleeping for one-hundred milliseconds, the code in the initializer gets and saves the current time relative to the start of the program.

Then the code in the initializer displays that time, producing the output shown in Figure 6.

```
Initializer-1: 310
```

Figure 6

If you compare Figure 6 with Figure 5 showing the time that the constructor for the class named **A** was executed, you will see that the printout produced by the initializer followed the printout produced by the constructor by the one-hundred millisecond delay introduced at the beginning of the initializer. This confirms that the first initializer in the class named **B** was executed following the execution of the constructor for the superclass named **A**.

Insert another time delay

Finally the instance initializer shown in Listing 8 inserts an additional one-hundred millisecond delay. This makes it possible to distinguish the time that the ordinary instance variable *(to be discussed next)* was initialized from the time that the print statement in the first instance initializer was executed.

An ordinary instance variable

Listing 9 shows the declaration and initialization of an ordinary instance variable named

simpleInitTime. Recall that the physical location of this variable declaration is after the first instance initializer and before the constructor and the second instance initializer.

```
  long simpleInitTime =
Init02.relTime();

Listing 9
```

Although the code in Listing 9 doesn't display the time of initialization, code later in the program causes the value stored in the variable named **simpleInitTime** to be displayed, producing the output shown in Figure 7.

```
class B simple init: 410
Figure 7
```

As you can see from the relative time shown in Figure 7, the instance variable was initialized immediately following completion of execution of the first instance initializer shown in Listing 8 *(compare Figure 7 with Figure 6).*

The second instance initializer

An examination of Listing 15 near the end of the lesson shows that the class named **B** contains a second instance initializer, separated from the first instance initializer by an ordinary instance variable declaration and a constructor. The second instance initializer is shown in Listing 10.

```
  {//This is another instance
initializer

    Init02.delay();

    init2Time = Init02.relTime();

    System.out.println("Initializer-2:
" +

init2Time);
  }//end instance initializer

Listing 10
```

Insert a delay

This initializer begins by inserting a one-hundred millisecond delay. Then it gets, saves, and displays the time relative to the start time for the program as shown in Figure 8.

```
Initializer-2: 510
Figure 8
```

A comparison of the relative time shown in Figure 8 with Figure 7 confirms that the second instance initializer was executed after the initialization of the ordinary instance variable shown in Listing 9. This confirms that the instance initializers and variable initializers are executed in the order they appear in the Java source code.

The constructor for the class named B

The constructor for the class named **B** physically separates the two instance initializers in the class definition. The code for the constructor begins in Listing 11.

```
  B(String str){
    super(str);

Listing 11
```

The constructor begins by using the **super** keyword to invoke a parameterized constructor on the superclass named **A**.
*(If you are unfamiliar with this use of the **super** keyword, see Lesson 1628 entitled The Essence of OOP using Java, The **this** and **super** Keywords at www.DickBaldwin.com.)*

Insert a time delay

The remaining code in the constructor, as shown in Listing 12, inserts a one-hundred millisecond time delay.

```
    Init02.delay();

    bXstrTime = Init02.relTime();
    System.out.println(str + "B:  " +
bXstrTime);
  }//end constructor for B

Listing 12
```

Get and display the time

Then the constructor gets, saves, and displays the time, producing the output shown in Figure 9. A comparison of the relative time shown in Figure 9 with the previous figures confirms that the initializers are run after the superclass constructor has returned but before the constructor of the current class runs.

```
Construct 1B:  611
Figure 9
```

The showData method

The class named **B** contains a method named **showData.** This method is shown in its entirety in Listing 13. The purpose of this method is to summarize the order of initialization by displaying the values stored in the various instance variables as the object was being instantiated.

```
  void showData(){
    System.out.println(
                    "\nInitialization
values:");
    System.out.println("class A xstr:
" +
```

```
aXstrTime);
    System.out.println("class B init-
1: " +

init1Time);
    System.out.println("class B simple
init: " +

simpleInitTime);
    System.out.println("class B init-
2: " +

init2Time);
    System.out.println("class B xstr:
" +

bXstrTime);
    System.out.println();//blank line
  }//end showData
```

Listing 13

The output of the showData method

The values stored in the instance variables are displayed in the order that the initialization steps took place during the instantiation of the object. Recall that the **showData** method was invoked on the object when it was instantiated in the **main** method in Listing 4, producing the output shown in Figure 10.

```
Initialization values:
class A xstr: 210
class B init-1: 310
class B simple init: 410
class B init-2: 510
class B xstr: 611
```

Figure 10

An examination of Figure 10 makes it clear that the superclass constructor was executed first, at a relative time of 210 milliseconds. This was followed by execution of the first instance initializer at 310 milliseconds, initialization of the ordinary instance variable at 410 milliseconds, and execution of the second instance initializer at 510 milliseconds. Finally the constructor for the class named **B** was executed at a relative time of 611 milliseconds.

Instantiate another object

Listing 14 shows the remaining code in the **main** method, which was not previously discussed.

```
    delay();

    System.out.println("Instantiate
second obj: "
                                  +
relTime());
```

```
    new B("Construct 2").showData();
  }//end main

Listing 14
```

The remaining code in the **main** method inserts another one-hundred millisecond delay, and then instantiates another object of the class named **B**. As before, this triggers a whole series of events, many of which produce output on the screen.

The screen output

The output produced by instantiating another object is shown in its entirety in Figure 11.

```
Instantiate second obj: 711
Construct 2A:  811
Initializer-1: 911
Initializer-2: 1111
Construct 2B:  1211

Initialization values:
class A xstr: 811
class B init-1: 911
class B simple init: 1011
class B init-2: 1111
class B xstr: 1211
Figure 11
```

Except for the differences in the relative time values, the output shown in Figure 11 matches that shown in Figure 2 and Figures 5 through 10. Note, however, that there is nothing in Figure 11 corresponding to the output previously shown in Figures 3 and 4.

Classes are not reloaded

Figures 3 and 4 show the output produced by the execution of the static initializers in the classes named **A** and **B**. The process of instantiating another object from a set of previously loaded classes does not cause those classes to be reloaded. Since static initializers are executed one time only when the class is first loaded, those initializers are not executed when the second object is instantiated from the class named **B**.

Constructors and instance initializers are executed for each object

However, constructors and instance initializers are executed, and ordinary instance variables are initialized each time a new object is instantiated. Therefore, the output shown in Figure 11 contains messages and time tags corresponding to the execution of both constructors, the execution of both instance initializers, and the initialization of an ordinary instance variable, all in the proper order.

Run the Program

At this point, you may find it useful to compile and run the program shown in Listing 15 near the end of the lesson.

Summary

Static initializer blocks

A static initializer is a block of code surrounded by curly braces that is embedded in a class definition and is qualified by the keyword **static**.

You can include any number of static initializer blocks within your class definition. They can be separated by other code such as method definitions and constructors. The static initializer blocks will be executed in the order that they appear in the code, regardless of the other code that may separate them.

The static initializers belonging to a class are executed *one time only* when the class is loaded.

Instance initializers

An instance initializer is a block of code surrounded by curly braces that is embedded in a class definition. *(It is not qualified by the keyword **static**.)* You can include any number of instance initializers in your class definition, and the initializers may be physically separated by other items, such as constructors, method definitions, variable declarations, etc.

Instance initializers and variable initializers, along with constructors, are executed each time a new object of the class is instantiated. The instance initializers and variable initializers are executed in the order that they appear in the Java source code. They are executed after the constructor for the superclass is executed, and before the constructor for the current class is executed.

Instance initializers are especially useful in anonymous classes *(to be explained in a future lesson).* However, they can be included in any class definition, and may make code more readable by initializing instance variables near the declaration of those variables rather than deferring that initialization to a constructor that is located further away in the class definition.

Instance initializers are very similar to *noarg* constructors, except that a class can define only one *noarg* constructor, but can define any number of instance initializers. A class definition can contain instance initializers in addition to a *noarg* constructor, in which case, the instance initializers will be executed before the *noarg* constructor is executed.

What's Next?

The next lesson in this series will explain and discuss inner classes, with special emphasis on *member* classes. Subsequent lessons will explain *local* classes, *anonymous* classes, and *top-level nested* classes.

Complete Program Listing

A complete listing of the program discussed in this lesson is show in Listing 15 below.

```
/*File Init02.java
Copyright 2003 R.G.Baldwin

Illustrates the use of instance initializers.

Instance initializers behave much like noarg
constructors, and are particularly useful for
anonymous classes, which are not allowed to
define any constructors, even those that take no
arguments.

This program defines a class named B that extends
a class named A.  Parameterized constructors are
used in both A and B to instantiate an object of
the class named B.

The controlling class defines and initializes
a static variable containing the time that the
program starts running in msec.  This value is
used as the base for computing time intervals
later as the execution of the program progresses.
The times that are computed and displayed later
are in msec relative to the time at which the
program starts running.

Static initializers are defined in both A and B
to display the time that the two classes are
loaded and the order in which they are loaded.

An instance variable is defined in the class
named B and is initialized (using a simple
initialization expression) with the time in msec
that the variable is initialized.  The physical
location of the instance variable follows the
first of two instance initializer in the class
definition.

In addition, two separate instance initializers
are defined in the class named B that perform
initialization after the constructor for A
completes execution and before the constructor
for B begins execution.  The first of these
initializers executes before the instance
variable mentioned above is initialized.  The
second of these initializers executes after the
instance variable is initialized, demonstrating
that initialization based on simple
initialization expressions and instance
initializers occurs in the order that the code'
appears in the class definition.

The two constructors and the two initializers
```

each get and print time information when they are executed to show the order in which the constructors and the initializers are executed.

Two separate instances of the class named B are created, showing not only the order in which the instance initializers and the constructors are executed, but also showing that the static initializers are executed one time only when the classes are loaded.

Each time an object of the class named B is instantiated, an instance method of the class is invoked to display the values of the instance variables initialized during the process of instantiating the object of the class named B.

100-msec time delays are purposely inserted at strategic points within the program in order to force the time intervals between the occurrence of the different steps in the program to be measurable.

The output for one run is shown below. Your results may be different depending on the speed of your computer.

```
Instantiate first obj: 0
Class A loaded: 10
Class B loaded: 110

Construct 1A:  210
Initializer-1: 310
Initializer-2: 510
Construct 1B:  611

Initialization values:
class A xstr: 210
class B init-1: 310
class B simple init: 410
class B init-2: 510
class B xstr: 611

Instantiate second obj: 711
Construct 2A:  811
Initializer-1: 911
Initializer-2: 1111
Construct 2B:  1211

Initialization values:
class A xstr: 811
class B init-1: 911
class B simple init: 1011
class B init-2: 1111
class B xstr: 1211
```

```
Note the 100-msec elapsed time intervals
between the various steps in the execution of the
program.  Also note the order in which the class
loading operations and the initialization steps
occur.

Tested using SDK 1.4.1 under WinXP
**********************************************/
import java.util.Date;

public class Init02{
  //Establish the base time in msec.
  static long baseTime = new Date().getTime();


  //This is a utility method used to insert a
  // 100-millisecod delay.
  static void delay(){
    try{
      Thread.currentThread().sleep(100);
    }catch(Exception e){e.printStackTrace();}
  }//end delay
  //----------------------------------------//

  //This is a utility method used to compute the
  // current time relative to the value stored
  // in the static variable named baseTime.
  static long relTime(){
    return ((new Date().getTime()) - baseTime);
  }//end printTime

  //----------------------------------------//

  public static void main(String[] args){
    //Invoke a parameterized constructor for the
    // class named B, which is a subclass of A.
    // Also invoke the showData method on that
    // object to display the values of the
    // instance variables that were initialized
    // during the construction of the object.
    System.out.println("Instantiate first obj: "
                                    + relTime());
    new B("Construct 1").showData();
    //Sleep 100 msec and then instantiate another
    // object.
    delay();
    System.out.println("Instantiate second obj: "
                                     + relTime());
    new B("Construct 2").showData();
  }//end main
  //----------------------------------------//
}//end class Init02
//========================================//

class A{
  long aXstrTime;
```

```
  static{//This is a static initializer, which is
  // run one time only when the class is loaded.
    //Print a message showing the time that the
    // class finishes loading.
    System.out.println("Class A loaded: " +
                                 Init02.relTime());
  }//End static initializer
  //-----------------------------------------//

  A(String str){//constructor
    //Sleep for 100 msec before completing this
    // construction
    Init02.delay();

    //Record the time of construction and print
    // a message showing the construction time.
    aXstrTime = Init02.relTime();
    System.out.println(str + "A:  " + aXstrTime);
  }//end constructor for A

}//end class A
//=========================================//

class B extends A{
  long bXstrTime;
  long init1Time;
  long init2Time;

  static{//This is a static initializer, which is
  // run one time only when the class is loaded.
    //Sleep for 100 msec to show the order
    // that the classes named A and B are loaded.
    Init02.delay();
    //Print a message showing the time that the
    // class finishes loading.
    System.out.println("Class B loaded: " +
                        Init02.relTime() + "\n");
  }//End static initializer
  //-----------------------------------------//

  {//This is an instance initializer
    //Sleep for 100 msec before doing this
    // initialization.
    Init02.delay();
    //Record the time and print a message showing
    // the time that this instance initializer
    // was executed.
    init1Time = Init02.relTime();
    System.out.println("Initializer-1: " +
                                     init1Time);

    //Sleep for 100 msec after doing this
    // initialization to separate this
    // initialization from the initialization of
    // the instance variable that follows.
```

```
    Init02.delay();
  }//end instance initializer
  //-----------------------------------------//

  //Note that this initialized instance variable
  // is located after the first instance
  // initializer and before the second instance
  // initializer..
  long simpleInitTime = Init02.relTime();

  //-----------------------------------------//

  //Note that this constructor is physically
  // located between the two instance initializer
  // blocks.  Both initializer blocks are
  // executed before the constructor for this
  // class is executed, but after the constructor
  // for the superclass is executed.
  B(String str){
    //Invoke a parameterized constructor on the
    // superclass.
    super(str);
    //Sleep for 100 msec before constructing
    // this part of the object.
    Init02.delay();
    //Record the time and print a message showing
    // the construction time for this part of
    // the object.
    bXstrTime = Init02.relTime();
    System.out.println(str + "B:  " + bXstrTime);
  }//end constructor for B
  //-----------------------------------------//

  {//This is another instance initializer
    //Sleep for 100 msec before doing this
    // initialization.
    Init02.delay();
    //Record the time and print a message showing
    // the time that this instance initializer
    // was executed.
    init2Time = Init02.relTime();
    System.out.println("Initializer-2: " +
                                      init2Time);
  }//end instance initializer
  //-----------------------------------------//

  void showData(){
    //This method displays the values that were
    // saved in the instance variables during the
    // five initialization steps, one of which
    // was execution of the superclass
    // constructor.  The values are displayed
    // in the order that the initialization steps
    // occurred.
    System.out.println(
                     "\nInitialization values:");
```

```
    System.out.println("class A xstr: " +
                                    aXstrTime);
    System.out.println("class B init-1: " +
                                    init1Time);
    System.out.println("class B simple init: " +
                                     simpleInitTime);
    System.out.println("class B init-2: " +
                                    init2Time);
    System.out.println("class B xstr: " +
                                    bXstrTime);
    System.out.println();//blank line
  }//end showData
}//end class B
```

Listing 15

About the author

Richard Baldwin *is a college professor (at Austin Community College in Austin, Texas) and private consultant whose primary focus is a combination of Java, C#, and XML. In addition to the many platform and/or language independent benefits of Java and C# applications, he believes that a combination of Java, C#, and XML will become the primary driving force in the delivery of structured information on the Web.*

Richard has participated in numerous consulting projects, and he frequently provides onsite training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Programming Tutorials, which has gained a worldwide following among experienced and aspiring programmers. He has also published articles in JavaPro magazine.

Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.

baldwin@DickBaldwin.com

-end-