The Essence of OOP using Java, Static Initializer Blocks

Baldwin explains and illustrates the use of static initializer blocks.

Published: July 15, 2003
By Richard G. Baldwin

Java Programming Notes # 1632

- Preface
- Preview
- Discussion and Sample Code
- Run the Program

- Summary
- What's Next
- Complete Program Listing

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, Exception Handling.

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

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 initialized with the garbage left over in memory from the programs that previously ran in the computer.

Automatic initialization to default values

Instance 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.

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 class variable named **var1** that is purposely initialized to the value 6, along with a reference variable of type **Date** that is allowed to be automatically initialized to the default value null.

```
  static int var1 = 6;

  Date date;
```

Figure 1

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 a constructor 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.

Static initializer blocks

What you may not know, however, is that you can also write *static initializer blocks* to initialize static variables when the class is loaded. The code in a static initializer block can also be quite complex.

A static initializer block resembles a method with no name, no arguments, and no return type. It doesn't need a name, because there is no need to refer to it from outside the class definition. The code in a static initializer block is executed by the virtual machine when the class is loaded.

Like a constructor, a static initializer block cannot contain a return statement. Therefore, it doesn't need to specify a return type.

Because it is executed automatically when the class is loaded, parameters don't make any sense, so a static initializer block doesn't have an argument list.

Syntax

So, what does this leave as the syntax of a static initializer block? All that is left is the keyword **static** and a pair of matching curly braces containing the code that is to be executed when the class is loaded.

Multiple static initializer blocks

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

According to one of my favorite authors, David Flanagan, author of Java in a Nutshell,

> *"What the compiler actually does is to internally produce a single class initialization routine that combines all the static variable initializers and all of the static initializer blocks of code, in the order that they appear in the class declaration. This single initialization procedure is run automatically, one time only, when the class is first loaded."*

The sample program that I will discuss in the next section will illustrate many aspects of static initializer blocks.

Discussion and Sample Code

In this lesson, I will discuss and explain a Java program named **Init01**, which illustrates static initializer blocks. As is often the case, I will discuss the program in fragments. A complete listing of the program can be viewed in Listing 14 near the end of the lesson.

This program illustrates the use of static initializer blocks to execute initialization code that is too complex to be contained in a simple variable initializer expression.

The code demonstrates that multiple initializer blocks are allowed, and that other code can be inserted between the static initializer blocks.

Finally, the code demonstrates that static initializer blocks are combined in the order in which they appear in the class definition, and executed once only.

The program was tested using SDK 1.4.1 under WinXP.

Order of execution

Because the compiler combines multiple static initializer blocks into a single initialization procedure, the order of execution of program code will not necessarily be the same as the order in which the code appears in the program. *(Multiple static initializer blocks can be separated by non-static code.)* This causes the program to be a little difficult to explain. In some cases, I will

present the code in a more meaningful order than the order in which it appears in the program. Time tags are displayed at various points in the program to help make sense of the order of execution.

Because the time tags are based on the system clock, the output produced by the program will be different each time the program is run *(at least the times will be different).* I will show you the output produce by a single running of the program. *(Note that in some cases, I manually inserted line breaks in the program output to force the material to fit in this narrow publication format.)*

The main method

The code in Listing 1 shows the beginning of the controlling class and the beginning of the **main** method. The code in the **main** method in Listing 1 displays the time that the program starts running, which is also the time at which the class named **A** is caused to start loading *(as you will see in Listing 2).*

```
public class Init01{
  public static void main(String[]
args){
    System.out.println("Start load: "
+

new Date());
```

Listing 1

Figure 2 shows the time at which the class named **A** started loading.

```
Start load: Fri May 30 15:28:49 CDT
2003
```

Figure 2

Load class A

The code in Listing 2 causes the class named **A** to start loading. Flanagan refers to the term ***A.class*** in Listing 2 as a class literal. *(I will show you an alternative way to accomplish the same thing using a method call in Listing 3).*

```
    Class aClass = A.class;
```

Listing 2

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

While I can't provide a description of exactly what happens from a technical viewpoint, 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, the class named **A**).* 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 on a reference to the **Class** object.
*(The code in Listing 2 causes the **Class** object's reference to be saved in the reference variable named **aClass**. However, the reference isn't used for any purpose elsewhere in the program. The purpose of the statement in Listing 2 is to cause the class to load, and is not to get and save a reference to the **Class** object.)*

An alternative approach

The comments in Listing 3 show an alternative way to force the class named **A** to be loaded, and to cause the **Class** object's reference to be saved in a reference variable named **aClass**. If activated, this code would cause the class to be loaded by invoking the static **forName** method of the class named **Class**, passing the name of the class to be loaded to the method as a **String** parameter.

```
    //try{
      //Class aClass =
Class.forName("A");
    //}catch(ClassNotFoundException
e){
    //
e.printStackTrace();}
```

Listing 3

If you find the code *(shown as comments)* in Listing 3 to be confusing, just ignore it. Understanding that code isn't critical to understanding static initializer blocks.

Display end of load time

Continuing in the **main** method, the code in Listing 4 causes the time to be displayed when the loading of the class named **A** is complete.

```
    System.out.println("End load: " +

new Date());
```

Listing 4

The code in Listing 4 causes the date and time shown in Figure 3 to be displayed on the screen.

```
End load: Fri May 30 15:28:54 CDT 2003
```

Figure 3

(You can view all of the screen output in the order in which it appears in the comments in Listing 14 near the end of the lesson.)

Five seconds have elapsed

If you compare the time shown in Figure 3 with the time shown in Figure 2, you will note that five seconds have elapsed during the time that the class was being loaded. The reason for this will become obvious as we examine the static initializer blocks in the definition of class **A**.

Now discuss the class named A

At this point, I am going to defer the remaining discussion of the **main** method until later and discuss the static initialization defined in the class named **A**.

Listing 5 shows an abbreviated listing of the class named **A**, showing only the code involved in the static initialization of the class when it is loaded. A complete listing of the class definition is shown in Listing 14 near the end of the lesson.

```
class A{
  //Declare six static variables
  static int var1 = 6;
  static int var2 = 9;
  static int var3;//originally
initialized to 0
  static long var4;//originally
initialized to 0

  static Date date1;//initialized to
null
  static Date date2;//initialized to
null

  //instance var declaration omitted
for brevity

  static{
    //First static initializer block
    date1 = new Date();
    for(int cnt = 0; cnt < var2;
cnt++){
      var3 += var1;
    }//end for loop
    System.out.println("End first
static init: "
                                +
new Date());
  }//end first static initializer
block

  //Constructor and instance method
omitted
  // for brevity.

  static{
    //Second static initializer block
    try{
      //Sleep for five seconds

Thread.currentThread().sleep(5000);
    }catch(Exception
e){System.out.println(e);}
    date2 = new Date();
    var4 = date2.getTime() -
```

```
date1.getTime();
    System.out.println("End second
static init: "
                                   +
new Date());
  }//end second static initializer
block
}//end class A

Listing 5
```

Six ordinary static variables

A careful examination of Listing 5 shows that it includes the declaration of six ordinary static variables with two of them being initialized to integer values when they are declared. The remaining four are automatically initialized to their default values when they are declared.

Two static initializer blocks

Listing 5 also shows two static initializer blocks, physically separated by the class constructor and an instance method. According to Flanagan, the code in Listing 5 is all combined by the compiler into a single static initialization procedure, which is executed one time only when the class is loaded.

I will separate the code in Listing 5 into several fragments and discuss those fragments in the paragraphs that follow.

Ordinary static variables

Listing 6 shows the declaration of six ordinary static variables, and the purposeful initialization of two of them. The remaining four are automatically initialized to their default values, but code in the static initializer blocks to follow will also provide non-default initial values for some of them.

```
class A{
  static int var1 = 6;
  static int var2 = 9;
  static int var3;//initialized to 0
  static long var4;//initialized to 0

  static Date date1;//initialized to
null
  static Date date2;//initialized to
null

Listing 6
```

First static initializer block

Listing 7 shows the first static initializer block.

```
  static{
    date1 = new Date();

    for(int cnt = 0; cnt < var2;
cnt++){
      var3 += var1;
    }//end for loop

    System.out.println("End first
static init: "
                                  +
new Date());
  }//end first static initializer
block
```

Listing 7

The code in this static initializer block records the date and time in one of the static variables, **date1**, declared earlier.

Then it executes a **for** loop to compute an initial value for one of the other static variables, **var3**, also declared earlier. Note that the computation of the initial value for **var3** is based on the initial value given to **var1** when it was declared.
(The contents of date1 and var3 will be displayed later.)

Complex code

Note also that the code in this initializer block is far too complex to be included in a simple initialization expression when a static variable is declared. Thus, this initializer block demonstrates the primary purpose of static initializer blocks - to execute code that cannot be included in an initialization expression that is part of a static variable declaration.

Display the date and time

Finally, the code in Listing 7 displays the date and time that the code in the initializer block completes execution. This date and time is shown in Figure 4.

```
End first static init:
                      Fri May 30
15:28:49 CDT 2003
```

Figure 4

If you compare the time in Figure 4 with the time in Figure 2, you will see that the two times are indistinguishable. In other words, the time required to execute the code in the first static initializer block was so short that the granularity of the time-display mechanism was too large to show actual the time difference.

Second static initializer block

If you refer back to Listing 5, or refer to Listing 14 near the end of the lesson, you will see that the static initializer block shown in Listing 7 is followed by the definition of the class constructor

and an instance method of the class. This is followed by a second static initializer block, as shown in Listing 8.

```
  static{
    try{

Thread.currentThread().sleep(5000);
    }catch(Exception
e){System.out.println(e);}

    date2 = new Date();
    var4 = date2.getTime() -
date1.getTime();

    System.out.println("End second
static init: "
                                    +
new Date());
  }//end second static initializer
block
}//end class A
```

Listing 8

The code in the second static initializer block purposely inserts a five-second time delay by putting the thread to sleep for five seconds. The static variable named **date2** is then initialized with a reference to a new **Date** object, which reflects the date and time following the five-second delay.

The time difference

Following this, the code in Listing 8 computes a new initial value for the previously declared static variable named **var4** based on values saved during previous initialization operations.

The variable named **var4** is initialized with a value that represents the time difference in milliseconds between the time recorded during the execution of the first static initializer block, and the time following the five-second delay in the second initializer block. Later, when the values stored in the static variables are displayed, we will see that the time difference was 5008 milliseconds.

Display the date and time

Finally, the code in the second initializer block shown in Listing 8 displays the date and time that the execution of the second initializer block is complete. This date and time is shown in Figure 5.

```
End second static init:
                      Fri May 30
15:28:54 CDT 2003
```

Figure 5

As you should expect, this date and time matches the date and time displayed by the **main**

method in Figure 3 as the time that the loading operation for the class named **A** was completed.

Now back to the main method

After the class named **A** is loaded, the main method purposely causes the main thread to sleep for five seconds as shown in Listing 9.

```
//Back in discussion of the main
method
    try{

Thread.currentThread().sleep(5000);
    }catch(Exception
e){System.out.println(e);}
```

Listing 9

A new object of the class named A

After the main thread has been allowed to sleep for five seconds, the code in the main method instantiates a new object of the class named **A** and invokes the **showData** method on that object. This code, which is shown in Listing 10, causes the values previously stored in the static variables to be displayed.

```
    new A().showData();
  }//end main
}//end class Init01
```

Listing 10

When the **showData** method returns, the program terminates.

Resume discussion of class A definition

Some of the code that I skipped in my earlier discussion of the definition of class **A** was the declaration of an instance variable named **date3**, as shown in Listing 11.

```
//Resume discussion of class A

  Date date3;
```

Listing 11

When does the initialization occur?

It is very important to understand that the initialization of static variables occurs when the class is originally loaded, while the initialization of instance variables occurs when an object of the class is instantiated. That characteristic of OOP is demonstrated in this program.

Record date and time of object instantiation

The class constructor, shown in Listing 12, causes the instance variable named **date3**, declared in Listing 11, to contain the date and time that the object is actually instantiated.

```
  A(){//constructor
    //Record the time in an instance
variable.
    date3 = new Date();
  }//end constructor

Listing 12
```

Display data in the variables

The method named **showData** is shown in Listing 13. The code in this method displays the times that the variables were initialized, along with the values stored in those variables.

```
  void showData(){//an instance method
    System.out.println("var3
initialized: "

+ date1);
    System.out.println("var3 = " +
var3);
    System.out.println("var4
initialized: "

+ date2);
    System.out.println("var4 = " +
var4

+ " msec");
    System.out.println("Obj
instantiated: "

+ date3);
  }//end showData

Listing 13
```

The output produced by the showData method

The **showData** method is invoked as soon as the new object is instantiated by the code in the **main** method, producing the screen output shown in Figure 6.

```
var3 initialized: Fri May 30 15:28:49
CDT 2003
var3 = 54
var4 initialized: Fri May 30 15:28:54
CDT 2003
var4 = 5008 msec
Obj instantiated: Fri May 30 15:28:59
CDT 2003
Figure 6
```

Five-second intervals

First, you should note the five-second intervals that separate the two initialization operations and the object instantiation.

Recall that the first five-second delay, that separates the two static initialization operations, was caused when the second static initializer block put the thread that was loading the class to sleep for five seconds.

Recall also that the second five-second delay was caused by the **main** method, which put the main thread to sleep for five seconds after the class named **A** was loaded, and before an object of the class named **A**, was instantiated.

Result of the loop computation

Also note the value of 54 stored in the static variable named **var3**. Recall that this value was computed by a **for** loop in the first static initializer block.

The time difference in milliseconds

Finally, note the value of 5008 milliseconds stored in the static variable named **var4**. This value was computed by code in the second static initializer block, after sleeping for 5000 milliseconds. This value represents the time difference between the execution of the first static initializer block and the completion of the second static initializer block following a five-second delay.

Run the Program

At this point, you may find it useful to compile and run the program shown in Listing 14 near the end of the lesson.

Summary

No random garbage allowed

Unlike other programming languages, it is not possible to write a Java program in which the variables are initialized with the random garbage left over in memory from the programs that previously ran in the computer.

Instance and static variables are automatically initialized to standard default values if you fail to purposely initialize them.

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.

Different approaches to variable initialization

You can initialize instance variables and class variables when you declare them, by including an initialization expression in the variable declaration statement.

If your initialization needs are more complex than can be satisfied with a single initialization expression, you can write a constructor to purposely initialize all instance variables when the object is instantiated. The code in a constructor can be as complex as needed.

You can also write a *static initializer block* to initialize static variables when the class is loaded. The code in a static initializer block, which is executed by the virtual machine when the class is loaded, can also be quite complex.

Static initializer blocks

A static initializer block resembles a method with no name, no arguments, and no return type. When you remove these items from the syntax, all that is left is the keyword **static** and a pair of matching curly braces containing the code that is to be executed when the class is loaded.

You may 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 in which they appear in the code, regardless of the other code that may separate them.

Instance initializers

Although not covered in this lesson, it is also possible to define an instance initializer block, which can be used to initialize instance variables in the absence of, or in addition to a constructor. As you will learn in a future lesson on anonymous classes, it is not always possible to define a constructor for a class, but it is always possible to define an instance initializer block.

What's Next?

The next lesson will explain and discuss instance initializer blocks that can be used in the absence of, or in addition to class constructors.

Complete Program Listing

A complete listing of the program discussed in this lesson is show in Listing 14 below.

```
/*File Init01.java
Copyright 2003 R.G.Baldwin

Illustrates the use of static initializer blocks
to execute code that is too complex to be
contained in a simple variable initializer.

Demonstrates that static initializer blocks are
executed in the order in which they appear in the
class definition.
```

```
Demonstrates that other code can be inserted
between static initializer blocks in a class
definition.

The output will change each time this program is
run.  The output for one run is shown below. Line
breaks were manually inserted to force the
material to fit in this narrow publication
format.

Start load: Fri May 30 15:28:49 CDT 2003
End first static init:
                     Fri May 30 15:28:49 CDT 2003
End second static init:
                     Fri May 30 15:28:54 CDT 2003
End load: Fri May 30 15:28:54 CDT 2003
var3 initialized: Fri May 30 15:28:49 CDT 2003
var3 = 54
var4 initialized: Fri May 30 15:28:54 CDT 2003
var4 = 5008 msec
Obj instantiated: Fri May 30 15:28:59 CDT 2003

Note the five-second time intervals that separate
the two initializations and the object
instantiation in the above output.

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

public class Init01{
  public static void main(String[] args){
        //Display start load time
    System.out.println("Start load: " +
                                      new Date());
    //Force the class named A to load using a
    // class literal.
    Class aClass = A.class;

    //Alternative way to cause the class named A
    // to load.
    //try{
      //Class aClass = Class.forName("A");
    //}catch(ClassNotFoundException e){
    //                     e.printStackTrace();}
    //Display end load time
    System.out.println("End load: " +
                                      new Date());

    //Sleep for five seconds after the class
    // loads
    try{
      Thread.currentThread().sleep(5000);
    }catch(Exception e){System.out.println(e);}
```

```
    //Instantiate a new object of the class named
    // A and display the data stored in the
    // variables.
    new A().showData();
  }//end main
}//end class Init01
//===========================================//

class A{
  //Declare six static variables and initialize
  // some of them when they are declared.  The
  // others will be automatically initialized to
  // either zero or null, but this may change
  // later due to the code in static initializer
  // blocks.
  static int var1 = 6;
  static int var2 = 9;
  static int var3;//originally initialized to 0
  static long var4;//originally initialized to 0

  static Date date1;//initialized to null
  static Date date2;//initialized to null

  //Declare an instance variable which is
  // originally initialized to null.
  Date date3;

  static{
    //First static initializer block records the
    // time and then executes a loop to
    // compute a new initial value for var3.
    date1 = new Date();
    for(int cnt = 0; cnt < var2; cnt++){
      var3 += var1;
    }//end for loop
    System.out.println("End first static init: "
                                    + new Date());
  }//end first static initializer block
  //-----------------------------------------//

  //Note that the constructor and an instance
  // method physically separate the two static
  // initializer blocks.
  A(){//constructor
    //Record the time in an instance variable.
    date3 = new Date();
  }//end constructor
  //-----------------------------------------//

  void showData(){//an instance method
    //Display the times that the variables were
    // initialized along with the values stored
    // in those variables.
    System.out.println("var3 initialized: "
                                        + date1);
    System.out.println("var3 = " + var3);
```

```
    System.out.println("var4 initialized: "
                                        + date2);
    System.out.println("var4 = " + var4
                                      + " msec");
    System.out.println("Obj instantiated: "
                                        + date3);
  }//end showData
  //----------------------------------------//

  static{
    //Second static initializer block sleeps for
    // five seconds, records the time, and then
    // computes a new intial value for var4 based
    // on values recorded during previous
    // initialization operations.
    try{
      //Sleep for five seconds
      Thread.currentThread().sleep(5000);
    }catch(Exception e){System.out.println(e);}
    date2 = new Date();
    var4 = date2.getTime() - date1.getTime();
    System.out.println("End second static init: "
                                    + new Date());
  }//end second static initializer block
}//end class A
```

Listing 14

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-