Class and object initialization

Source http://www.javaworld.com/javaworld/jw-11-2001/jw-1102-java101.html

Initialization prepares classes and objects for use during a program's execution. Although we tend to think of initialization in terms of assigning values to variables, initialization is so much more. For example, initialization might involve opening a file and reading its contents into a memory buffer, registering a database driver, preparing a memory buffer to hold an image's contents, acquiring the resources necessary for playing a video, and so on. Think of anything that prepares a class or an object for use in a program as initialization.

Java supports initialization via language features collectively known as initializers. Because Java handles class initialization differently from object initialization, and because classes initialize before objects, we first explore class initialization and class-oriented initializers. Later, we explore object initialization and object-oriented initializers.

Class initialization

A program consists of classes. Before a Java application runs, Java's class loader loads its starting class -- the class with a public static void main(String [] args) method -- and Java's byte code verifier verifies the class. Then that class initializes. The simplest kind of class initialization is automatic initialization of class fields to default values. Listing 1 demonstrates that initialization:

Listing 1. ClassInitializationDemo1.java

```
                // ClassInitializationDemo1.java
class ClassInitializationDemo1
{
  static boolean b;
  static byte by;
  static char c;
  static double d;
  static float f;
  static int i;
  static long l;
  static short s;
  static String st;
  public static void main (String [] args)
  {
    System.out.println ("b = " + b);
    System.out.println ("by = " + by);
    System.out.println ("c = " + c);
    System.out.println ("d = " + d);
    System.out.println ("f = " + f);
    System.out.println ("i = " + i);
    System.out.println ("l = " + l);
    System.out.println ("s = " + s);
    System.out.println ("st = " + st);
  }
}
```

ClassInitializationDemo1's static keyword introduces a variety of class fields. As you can see, no explicit values assign to any of those fields. And yet, when you run ClassInitializationDemo1, you see the following output:

```
b = false
by = 0
c =
d = 0.0
f = 0.0
i = 0
l = 0
s = 0
st = null
```

The false, 0, 0.0, and null values are the type-oriented representations of default values. They represent the result of all bits automatically set to zero in each class field. And what automatically set those bits to zero? The JVM, after a class is verified. (Note: In the preceding output, you do not see a value beside c = because the JVM interprets c's default value as the nondisplayable null value.)

Class field initializers

After automatic initialization, the next simplest kind of class initialization is the explicit initialization of class fields to values. Each class field explicitly initializes to a value via a class field initializer.

*~~~ End of Article ~~~*