Supplement: Initialization Block
For Introduction to Java Programming
By Y. Daniel Liang

Initialization blocks can be used to initialize objects along with the constructors. An *initialization block* is a block of statements enclosed inside a pair of braces. An initialization block appears within the class declaration, but not inside methods or constructors. It is executed as if it were placed at the beginning of every constructor in the class.

Initialization blocks can simplify the classes if you have multiple constructors sharing a common code and none of them can invoke other constructors. The common code can be placed in an initialization block, as shown in the example in Figure 1a. In this example, none of the constructors can invoke any of the others using the syntax **this(...)**. When an instance is created using a constructor of the **Book** class, the initialization block is executed to increase the object count by **1**. The program is equivalent to Figure 1b:

```
public class Book {
  private static int numOfObjects;
  private String title
  private int id;

  public Book(String title) {
    this.title = title;
  }

  public Book(int id) {
    this.id = id;
  }

  {
    numOfObjects++;
  }
}
```

Equivalent

```
public class Book {
  private static int numOfObjects;
  private String title;
  private int id;

  public Book(String title) {
    numOfObjects++;
    this.title = title;
  }

  public Book(int id) {
    numOfObjects++;
    this.id = id;
  }
}
```

(a) A class with initialization blocks **(b) An equivalent class**

Figure 1
An initialization block can simplify coding for constructors.

> **NOTE**
> A class may have multiple initialization blocks. In such cases, the blocks are executed in the order they appear in the class.

The initialization block in Figure 10.7(a) is referred to as an *instance initialization block* because it is executed whenever an instance of the class is created. A *static*

initialization block is much like an instance initialization block except that it is declared **static**, can only refer to static members of the class, and is executed when the class is loaded. The JVM loads the class dynamically when it is needed. A superclass is loaded before its subclasses. The order of the execution can be summarized as follows:

1. When a class is used for the first time, it needs to be loaded. Loading involves two phases:

   1.1. Load superclasses. Before loading any class, its superclass must be loaded if it is not already loaded. This is a recursive process until a superclass along the inheritance chain is already loaded.
   1.2. After a class is loaded to the memory, its static data fields and static initialization block are executed in the order they appear in the class.

2. Invoking a constructor of the class involves three phases:

   2.1. Invoke a constructor of the superclass. This is a recursive process until the superclass is **java.lang.Object**.
   2.2. Initialize instance data fields and execute initialization blocks in the order they appear in the class.
   2.3. Execute the body of the constructor.

Listing 1 demonstrates the execution order of initialization blocks.

Listing 1 InitializationDemo.java

```
public class InitializationDemo {
  public static void main(String[] args) {
    new InitializationDemo();
  }

  public InitializationDemo() {
    new M();
  }

  {
    System.out.println("(2) InitializationDemo's instance block");
  }

  static {
    System.out.println("(1) InitializationDemo's static block");
  }
}
```

```
class M extends N {
  M() {
    System.out.println("(8) M's constructor body");
  }

  {
    System.out.println("(7) M's instance initialization block");
  }

  static {
    System.out.println("(4) M's static initialization block");
  }
}

class N {
  N() {
    System.out.println("(6) N's constructor body");
  }

  {
    System.out.println("(5) N's instance initialization block");
  }

  static {
    System.out.println("(3) N's static initialization block");
  }
}
```

Sample Output

```
(1) InitializationDemo's static block
(2) InitializationDemo's instance block
(3) N's static initialization block
(4) M's static initialization block
(5) N's instance initialization block
(6) N's constructor body
(7) M's instance initialization block
(8) M's constructor body
```

The program is executed in the following order:

(1) The superclass of **InitializationDemo**, **java.lang.Object**, is loaded first. Then class **InitializationDemo** is loaded, so **InitializationDemo**'s static initialization block is executed.

(2) **InitializationDemo**'s constructor is invoked (line 3), so **InitializationDemo**'s instance initialization block is executed.

(3) When executing **new M()** (line 7), class **M** needs to be loaded, which causes class **M**'s superclass (i.e., **N**) to be loaded first. So **N**'s static initialization block is executed. (Note that **N**'s superclass **java.lang.Object** has already been loaded in (1)).

(4) Class **M** is now loaded. So **M**'s static initialization block is executed.
(5) When invoking **M**'s constructor, the no-arg constructor of **M**'s superclass is invoked first; therefore, **N**'s instance initialization block is executed.
(6) The regular code in **N**'s no-arg constructor is invoked after **N**'s instance initialization block is executed.
(7) After **N**'s no-arg constructor is invoked, **M**'s no-arg constructor is invoked, which causes **M**'s instance initialization block to be executed first.
(8) The regular code in **M**'s no-arg constructor is invoked after **M**'s instance initialization block is executed.

NOTE

If an instance variable is declared with an initial value (e.g., **double radius = 5**), the variable is initialized just as in an initialization block. That is, it is initialized when the constructor of the class is executed. If a static variable is declared with an initial value (e.g., **static double radius = 5**), the variable is initialized just as in a static initialization block. That is, it is initialized when the class is loaded.

Q: Why the code in (a) is correct, but the code in (b) is wrong?

```
public class A {
  private int[] grid = new int[SIZE];
  static int SIZE = 10;
}
```

(a)

```
public class A {
  private int[] grid = new int[SIZE];
  int SIZE = 10;
}
```

(b)

A: The static field SIZE is initialized before the instance field grid. So, (a) is correct. In (b) grid is initialized, but SIZE has not been defined yet.