Initialization Block

PREVIEW / RECAP

We know that there exists a concept of Constructor and Methods in classes and objects. However in this topic we will learn about a concept known as initialization block.

Initializing Fields

As you have seen, you can often provide an initial value for a field in its declaration:

```
public class BedAndBreakfast {

    // initialize to 10
    public static int capacity = 10;

    // initialize to false
    private boolean full = false;
}
```

This works well when the initialization value is available and the initialization can be put on one line. However, this form of initialization has limitations because of its simplicity.
If initialization requires some logic (for example, error handling or a for loop to fill a complex array), simple assignment is inadequate.

Static Initialization Blocks

Instance variables can be initialized in constructors, where error handling or other logic can be used. To provide the same capability for class variables, the Java programming language includes static initialization blocks.

A static initialization block is a normal block of code enclosed in braces, { }, and preceded by the static keyword. Here is an example:

```
static {
    // whatever code is needed for initialization goes here
}
```

There is an alternative to static blocks — you can write a private static method:

```
class Whatever {
    public static varType myVar = initializeClassVariable();

    private static varType initializeClassVariable() {

        // initialization code goes here
    }
}
```

However, the static initialization blocks can only initialize the static instance variables. These blocks are only executed once when the class is loaded. There can be multiple static initialization blocks in a class that is called in the order they appear in the program.

```java
public class Demo {
   static int[] numArray = new int[10];
   static {
      System.out.println("Running static initialization block.");
      for (int i = 0; i < numArray.length; i++) {
         numArray[i] = (int) (100.0 * Math.random());
      }
   }
   void printArray() {
      System.out.println("The initialized values are:");
      for (int i = 0; i < numArray.length; i++) {
         System.out.print(numArray[i] + " ");
      }
      System.out.println();
   }
   public static void main(String[] args) {
      Demo obj1 = new Demo();
      System.out.println("For obj1:");
      obj1.printArray();
      Demo obj2 = new Demo();
      System.out.println("\nFor obj2:");
      obj2.printArray();
   }
}
```

Output

Running static initialization block.

For obj1:

The initialized values are:

40 75 88 51 44 50 34 79 22 21

For obj2:

The initialized values are:

40 75 88 51 44 50 34 79 22 21

The advantage of private static methods is that they can be reused later if you need to reinitialize the class variable.

Instance Initialization Block

This is similar to a static initialization block. But the main difference is that static blocks get called only once **at the time of program compiling**, whereas instance block gets called **every time the instance is created**.

Now you might have a question that whatever is given above related to Instance Initialization Block is the same as what a constructor does. Then why is there a need for this instance initialization block?
Because, at the time of compiling the instance block gets copied into constructors and at the time of execution it gets executed. So assume that there are more than one constructors in a class (having 90% the same code). Then we can just write a single Instance Initialization Block (which will get copied into each constructor at run time).

Okay! Let's get back to the Instance Initialization Concept.

There are two alternatives to using a constructor to initialize instance variables: initializer blocks and final methods.
Initializer blocks for instance variables look just like static initializer blocks, but without the static keyword:

```
{
    // whatever code is needed for initialization goes here
}
```

As already discussed above, the Java compiler copies initializer blocks into every constructor. Therefore, this approach can be used to share a block of code between multiple constructors.
A final method cannot be overridden in a subclass. This is discussed in the lesson on interfaces and inheritance. Here is an example of using a final method for initializing an instance variable:

```
class Whatever {
    private varType myVar = initializeInstanceVariable();

    protected final varType initializeInstanceVariable() {

        // initialization code goes here
    }
}
```

This is especially useful if subclasses might want to reuse the initialization method. The method is final because calling non-final methods during instance initialization can cause problems.

Great!!! Now you must have understood the initialization bloc concept. Let's have a look at one more program for better understanding:

```java
public class Test {

    static int staticVariable;
    int nonStaticVariable;

    // Static initialization block:
    // Runs once (when the class is initialized)
    static {
        System.out.println("Static initalization.");
        staticVariable = 5;
    }

    // Instance initialization block:
    // Runs each time you instantiate an object
    {
        System.out.println("Instance initialization.");
        nonStaticVariable = 7;
    }

    //The constructor
    public Test() {
        System.out.println("Constructor.");
    }

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

Output

Static initalization.
Instance initialization.
Constructor.
Instance initialization.
Constructor.

Explanation

As we can see even though we created 2 instances of Test Class. THe static Block was called only once (at compile time), whereas the instance block got copied into the constructor (at compile time for each instance) and the constructor got executed at runtime.