Theory: Initialization blocks

15 minutes 0 / 5 problems solved

Skip this topic Start practicing

2515 users solved this topic. Latest completion was about 19 hours ago.

§1. Static initialization block

A **static initialization block** is a block of code enclosed in braces `{}` and preceded by the `static` keyword:

```
static {
    // code
}
```

It's used to initialize **static fields** and **constants**, just like constructors help to initialize instance fields. We can create objects and invoke static methods in a static block.

Here is an example.

```
import java.util.Date;

public class StaticInitBlockExample {

    private static String stringField;
    private static Date dateField;

    private static final String A_STRING_CONSTANT;


    static {

        stringField = getEmptyString();

        dateField = new Date();

        A_STRING_CONSTANT = "unknown";

    }



    private static String getEmptyString() {

        return "empty";

    }

}
```

A class can have multiple static blocks which will be executed in the order that they appear in the source code. The values initialized in the first block are overwritten by the following blocks.

But the question is, what has performed earlier — the direct assignment to static fields or the static block?

See the following example.

```
public class StaticInitOrderExample {

    static int field = 30; // the first assignment

    static {
        field = 50; // the second assignment
    }
}
```

First, the direct assignment to the static field is performed. After that, the static block is executed. If you print the value of `field`, it will be equal to 50.

Note, it's impossible to access instance fields and methods in a **static block**.

A static initialization block is executed once for the whole class, not for each instance of the class.

Current topic:

Initialization blocks

Topic depends on:

✓ Static members

Table of contents:

§2. Instance initialization block

There is also an **instance initialization block**. It's used to initialize instance data members. It is run each time an object of the class is created. An instance initialization block is code enclosed in braces `{}`.

```
class InstanceInitBlockExample {

    private int field;

    {
        field = 40;
    }
}
```

Of course, we can also directly assign values to fields:

```
private int field = 40;
```

But if we need to perform more complex logic before a constructor is invoked, it's convenient to write an instance initialization block. For example, an instance initialization block is useful when we need to fill an array:

```
class ArrayInitExample {

    private int[] array;

    {
        System.out.println("Before the constructor");

        array = new int[10];
        for (int i = 0; i < array.length; i++) {

            array[i] = i * i;

        }

    }



    public void print() {

        for (int num : array) {

            System.out.printf("%d ", num);

        }

    }

}
```

The **instance initialization block** is executed before any constructor of a class (but after the superclass constructors). The java compiler invokes the block as the first statement in the constructor, before other statements.

All instances of this class will be initialized during creation. There is an example:

```
public class UsingArrayExample {
    public static void main(String args[]) {
        ArrayInitExample obj = new ArrayInitExample();
        obj.print();
    }
}
```

This code outputs:

```
Before the constructor
0 1 4 9 16 25 36 49 64 81
```

You can write as many initialization blocks as you need. They will be performed in the order in which they appear in your code.

Note, static class members can be accessed in an **instance initialization block**.

Report a typo

222 users liked this theory. 8 didn't like it. **What about you?**

😍 🙂 😐 🙁 😡

Start practicing

Comments (6) Hints (1) Useful links (3) Show discussion