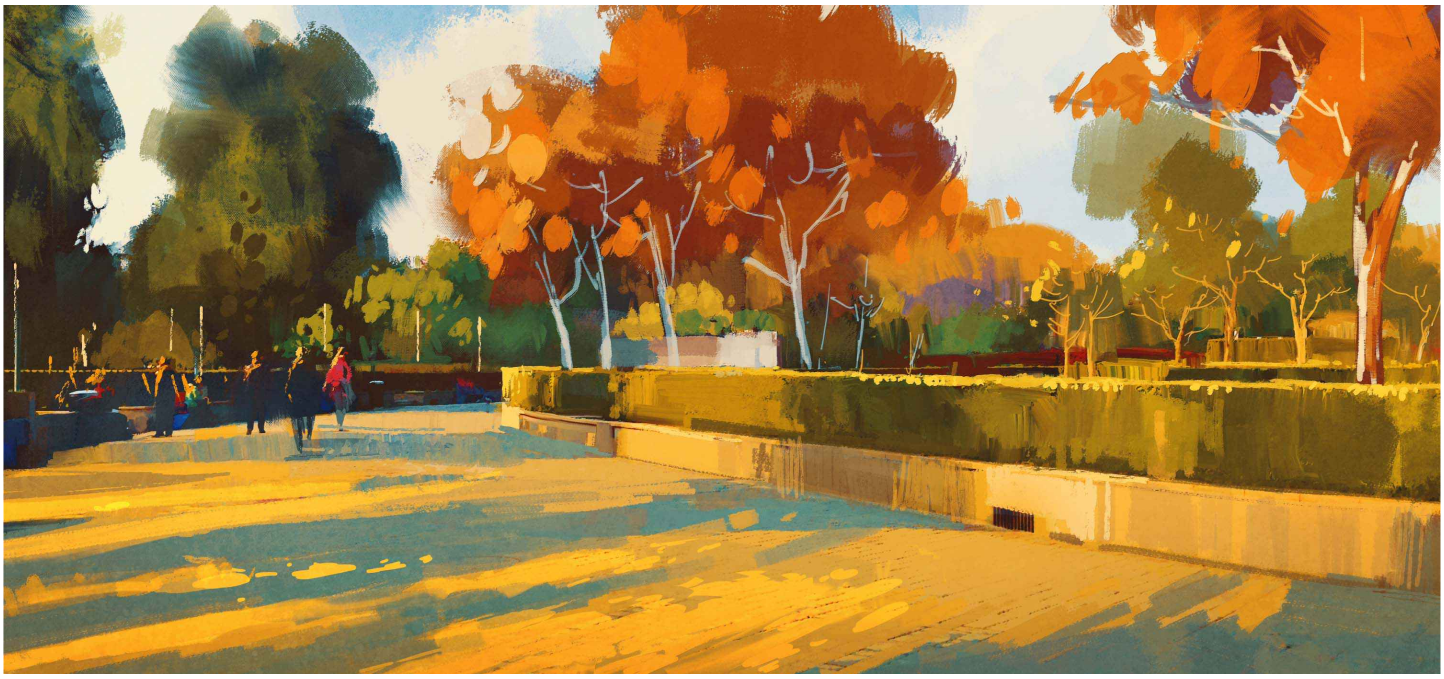


12-Static Variables, Classes & Blocks

Author: Vincent Lau

Learning Objectives

> Use the static keyword for methods and variables
>
> Understand the difference between instance and static variables/methods
>
> Declare constant variables
>
> Understand static initialization blocks and initializer blocks
>
> Understand order of execution of initialization blocks and result of initialization

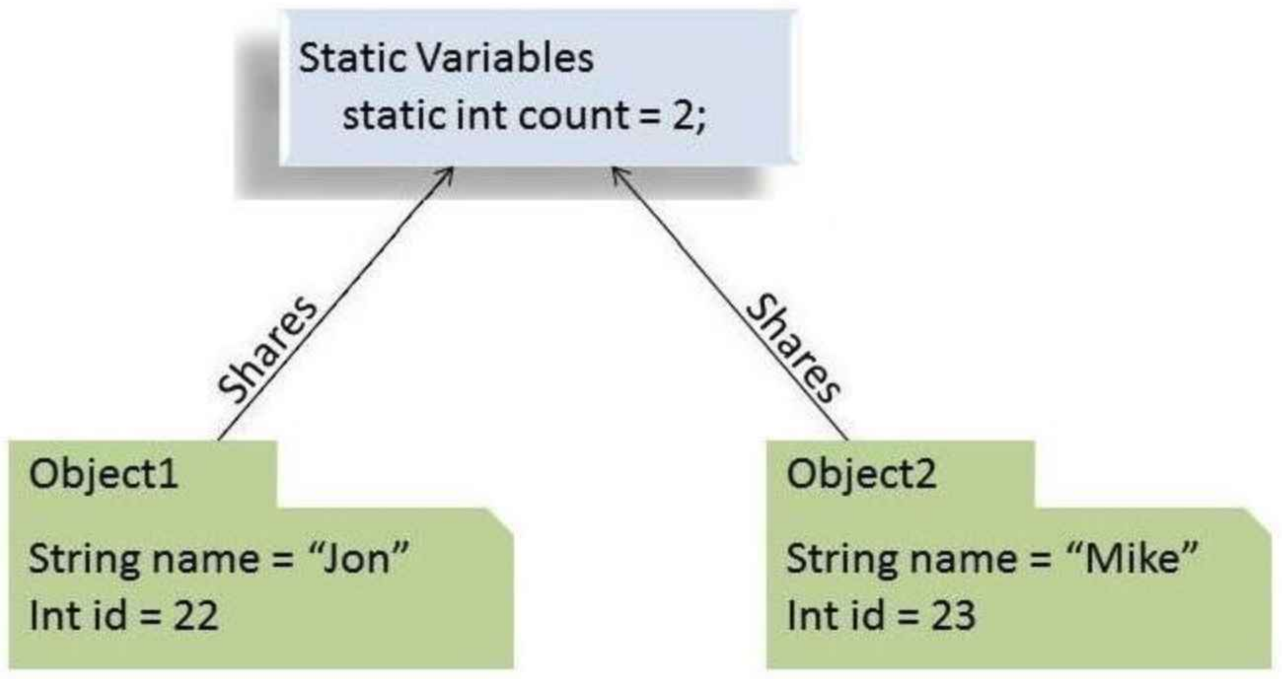
Static Keyword

- Sometimes, you want to have variables or methods that are common to all objects. This is where the *static* modifier comes in. Fields or methods that have the *static* modifier in their declaration are called ***static* *fields/methods*** or ***class* *fields/methods***.

- Static fields/methods are associated with the class, rather than with any object. Every instance of the class shares the static variable/method, which is in one fixed location in memory (i.e. OOP: Creating too many unnecessary static variables can incur non-releasable memory overhead).
- Any object can change the value of a static variable, but static variables can also be manipulated without creating an instance of the class.
- Static methods should be invoked with the class name to make it clear they are static methods.

```
Person person1 = new Person();

// Recommended -- Invoking static method with class name
Person.someStaticMethod();

// Not Recommended -- Invoking static method with object reference
person1.someStaticMethod();
```

- Static methods cannot access instance variables or instance methods directly - they must use an object reference.
- Static **cannot** use ***this*** keyword as there is no instance for *this* to refer to.

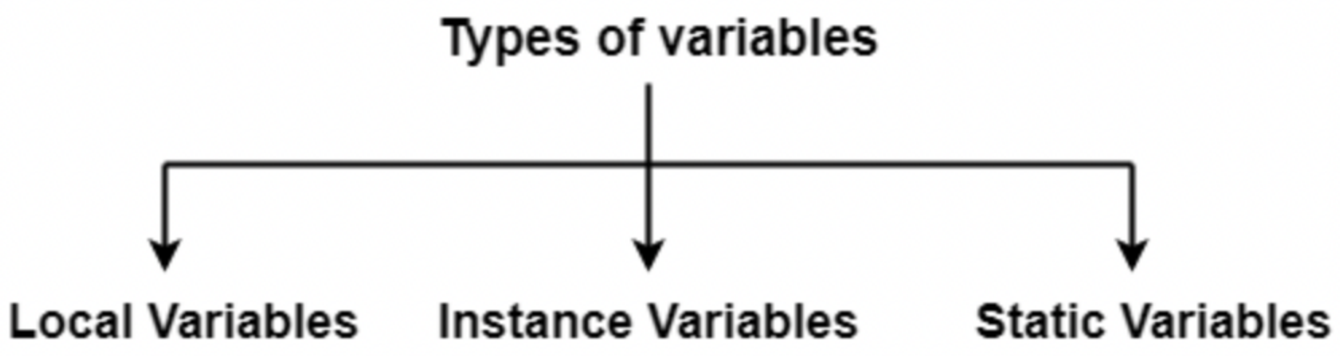


Static Variable

```java
public class Person {
    private int id;
    private int age;
    private static int counter = 0;

    public Person() {
        this.id = ++counter;
    }

    public int getId() {
        return id;
    }

    // getter and setter for age ...

    public static int getCounter() {
        return counter;
    }

    public static void setCounter(int counter) {
        Person.counter = counter;
    }
}

public class StaticDemo {
    public static void main(String[] args) {
        Person person1 = new Person();
        Person person2 = new Person();
        Person person3 = new Person();
        Person person4 = new Person();
        Person person5 = new Person();

        System.out.println("Person1's id : " + person1.getId());  // prints 1
        System.out.println("Person2's id : " + person2.getId());  // prints 2
        System.out.println("Person3's id : " + person3.getId());  // prints 3
        System.out.println("Person4's id : " + person4.getId());  // prints 4
        System.out.println("Person5's id : " + person5.getId());  // prints 5
        System.out.println("Person's Counter : " + Person.getCounter());  // prints 5

        Person.setCounter(100);
        Person person6 = new Person();
        System.out.println("Person6's id : " + person6.getId());  // prints 101
    }
}
```

Types of Variables

Let's recap all the types of variables that we have learned these weeks.



- `Instance variables (Non-static fields)` - State of an individual object which can be different between different objects of the same class, such as a person's age **(Chapter 7 - Instance Variables)**
- `Class variables (Static fields)` - A class-level variable shared by all objects of that class **(This Chapter)**
- `Local variables` - Variables declared within a method or code block **(Chapter 5 - Local Variable)**
- `Parameters` - The list of variables in a method declaration

```java
class ClassName {
    int count = 100; //instance variable
    static int a = 5; //class variable

    static int addMethod(int value) { // parameter
        int b = 77; //local variable
        b = b + value;
        return b;
    }
}
```

Initializing Fields by Blocks

- Sometimes when the values of some variables are known before creating the object, we can declare and initialize them in one line for simplicity's sake.

- If the logic of initialization is more complicated or if the values are not known ahead of time, we can leave the initialization to the constructor, where error handling and validation can be applied.
- It is not necessary to initialize every field at the beginning of the class definition.

```java
public class ShoeFactory {
    private static int numOfWorkers;
    private boolean running = false;

    static {
        // perform initialization here
        numOfWorkers = 102;
    }
}
```

Static Initialization Blocks

- A *static initialization block* is a normal block of code enclosed in braces `{}` , and preceded by the *static* keyword. For example:

```java
static {
    // perform initialization here
    numOfWorkers = 102;
}
```

- A class can have any number of static initialization blocks, and they can appear anywhere in the class body. **The runtime system guarantees that static initialization blocks are invoked in the order they appear in the source code. They are called only once over the lifetime of a program**.
- Here is an alternative to static blocks:

```java
class SomeClass {
    private static String someVar = initializeStaticVariable();

    // private static method
    private static String initializeStaticVariable() {
        // initialization code goes here

    }
}
```

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

Initializer Blocks

- Initializer blocks look just like static initialization blocks, but they are for instance variables, and do not have the *static* keyword.

```
{
    // perform initialization here
}
```

- **The compiler copies initializer blocks into every constructor**. Therefore, this approach can be used to share a block of code between multiple constructors.
- Same for initializer blocks, they are **invoked in the order they appear in the source code**

Order of Execution by Example

- Variables are declared as *public* for demonstration purpose (Don't do this in reality!).
- Notice how **static blocks and methods are called in the order they appear in the code**, before initializer blocks and constructors.
- Note that if *ClassB* extends *ClassA*, when creating an object of *ClassB (but not Class A)*, **ONLY** initializer blocks & constructor of *ClassA* will be invoked (**static blocks of Class A won't be invoked**).

```
public class ClassA {
    // 1
    public static final int SOME_NUMBER = initSomeNum();
    public static int someOtherNumber = 99;
    // 2
    public final String someString = initSomeString();

    // 1.2
    static {
        System.out.println("[Class A] Calling static initialization block");
//          someOtherNumber = 100;
    }
    // 2.2
    {
        System.out.println("[Class A] Calling initialization block");
```

```java
    }
    // 1.1
    private static int initSomeNum() {
        System.out.println("[Class A] Calling private static method");
        return 0;
    }
    // 2.1
    private String initSomeString() {
        System.out.println("[Class A] Calling private method");
        return "SomeStringA";
    }
    // 3
    public ClassA() {
        System.out.println("[Class A] Calling constructor");
    }
}

public class InitDemo {
    public static void main(String[] args) {
        System.out.println("[Class A] Beginning of object creation");
        ClassA objectA = new ClassA();
        System.out.println("[Class A] End of object creation");
    }
}

/*
------ Output ------
[Class A] Beginning of object creation
[Class A] Calling private static method
[Class A] Calling static initialization block
[Class A] Calling private method
[Class A] Calling initialization block
[Class A] Calling constructor
[Class A] End of object creation
*/
```

Initialization Result

```java
public class InitializationResult {
    public static void main(String[] args) {
        ClassA objectA = new ClassA();
        ClassB objectB = new ClassB();

        System.out.println("objectA.someString = " + objectA.someString);
        System.out.println("objectB.someString = " + objectB.someString);
        System.out.println("ClassA.SOME_NUMBER  = " + ClassA.SOME_NUMBER);
        System.out.println("ClassB.SOME_NUMBER  = " + ClassB.SOME_NUMBER);

        ClassB.someOtherNumber += 1;
        System.out.println("ClassA.someOtherNumber  = " +
ClassA.someOtherNumber);
    }
}

/*
------ Output ------
objectA.someString = SomeStringA
objectB.someString = SomeStringB
ClassA.SOME_NUMBER  = 0
ClassB.SOME_NUMBER  = 1
ClassA.someOtherNumber  = 100
*/
```

Questions

- What is the difference between instance and static variables/methods?
- What is the order of execution for static blocks, initializer blocks and constructors?