



Statics

AUGUST 8, 2015

Static is one of the **non-access modifier** which has a huge role in Java. It can be used with both variables and methods. It is used when we want some variable or method not to be a part of any specific object of a class rather common for all objects of the class.

For example, keeping count of all the instances created for a particular class we can make the count variable `static` or when we have a method which returns the same result irrespective of the instance invoking it like `getRandom()` returning any random number or `getMax(int x, int y)` returning the larger number between `x` and `y` we can make the methods `static`.

Ways to access static variables/methods

As static variable/methods doesn't belong to any particular object of a class, you need not create any object to access static variable/method. You can access directly with the help of classname and dot `'.'` operator but you can also access normally like any other instance variable/method i.e, via object reference and dot `'.'` operator. See below:

```
1  class Frog {
2
3      static int frogCount = 0; // Declare and initialize
4                                // static variable
5      int frogSize = 0;
6
7      public Frog() {
8          frogCount += 1; // Modify the value in the constructor
9      }
10
11     public static int getFrogCount() {
12         return frogCount;
13     }
14
15     public int getFrogSize() {
16         return frogSize;
17     }
18
19     public static void main(String[] args) {
20         Frog f = new Frog();
21         System.out.println(f.getFrogSize()); // Access instance
22                                                // method via object reference f
23
24         System.out.println(Frog.frogCount); // Access static
25                                                // variable via class
26
27         System.out.println(f.frogCount); // Access static
28                                                // variable via object reference f
29
30         System.out.println(Frog.getFrogCount()); // Access static
31                                                // method via class
32
33         System.out.println(f.getFrogCount()); // Access static
34                                                // method via object reference f
35     }
36 }
```

Some points to remember here

- Static method **cannot access** a instance(non-static) variable directly.

```
public static int getFrogSize() {  
    return frogSize; // compiler error  
}
```

But they can access through a object reference.

```
public static int getFrogSize() {  
    Frog obj = new Frog();  
    return obj.frogSize; // ok  
}
```

- Static method **cannot access** a instance method directly.

```
public static int getFrogSizeSum() {  
    int size = getFrogSize(); // compiler error  
    return size * frogCount;  
}
```

But they can access through a object reference.

```
public static int getFrogSizeSum() {  
    Frog obj = new Frog();  
    int size = obj.getFrogSize(); // ok  
    return size * frogCount;  
}
```

-

Static method **can access** a static variable or static method.

-

Static methods **cannot be overridden**. This is an important point. Consider the below program to understand:

```

1  class Animal {
2      static void doStuff() {
3          System.out.print("a ");
4      }
5  }
6
7  class Dog extends Animal {
8
9      // it's a redefinition,
10     // not an override
11     static void doStuff() {
12         System.out.print("d ");
13     }
14
15     public static void main(String[] args) {
16         Animal[] a = {new Animal(), new Dog(), new Animal()};
17         for (int x = 0; x < a.length; x++)
18             a[x].doStuff();
19     }
20 }

```

The output of the above program is 'a a a'. If the `doStuff()` method had been overridden then the output would have been 'a d a'. Because in case of **method overriding** polymorphism comes to play, according to which the actual object type determines which method is to be invoked but in this case the object reference determines which method is to be invoked (`doStuff()` of `Animal` class is invoked all the time as the object reference is of `Animal` type) i.e., no overriding and no polymorphism.

In other words, it perfectly makes sense as only inherited methods are overridden and `static` methods are never inherited as they don't belong to any object, so they can't be overridden.

- Static methods **can be overloaded** in the same class as well in the child classes. Read **method overloading** to be more clear.

```
1  public class Foo {
2      public static void doStuff(int x) { }
3
4      public static void doStuff(int x, int y) { } // Valid overload as DIFFERENT ARGUMENT L1
5                                                    // (and methods can be overloaded
6                                                    // in the same class or in subclass)
7  }
8
9  class Bar extends Foo {
10
11      public static void doStuff(String x) { } // Valid overload as DIFFERENT ARGUMENT LIST
12                                              // (and methods can be overloaded
13                                              // in the same class or in subclass)
14  }
```

Constructors

[« Previous](#)

Nested Classes

[Next »](#)

0 Comments **Java Notes**

 **Login** ▾

 **Recommend**

 **Share**

Sort by Best ▾



Start the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 

Name

Be the first to comment.

ALSO ON JAVA NOTES

Nested Classes | Java Notes



2 comments • 3 years ago

Amit Satpathy — Ram, it's really great to find your short and precise writeups on tech. Keep up the passion.

Overloading | Java Concepts

1 comment • 3 years ago

Anagh Hegde — `public class MyTest { public static void main(String[] args) { MyTest test = new MyTest(); int i = 9; test.TestOverLoad(i); } ...`

 **Subscribe**  **Add Disqus to your site**[Add DisqusAdd](#)  **Disqus' Privacy Policy**[Privacy Policy](#)[Privacy Policy](#)[Privacy Policy](#)

Carefully curated by [Ram swaroop](#). Powered by [Jekyll](#) with [Type Theme](#).