**Static and Non Static methods in Java**

Before I explain what static and non-static in Java means, let me ask you a question. Have you ever seen the word Math or Calendar used in java? If so you might have a rough idea of what I’m about to tell you. You already know that when you want to call a method, you just use a reference variable.

Ex:

**Dog d = new Dog ();**

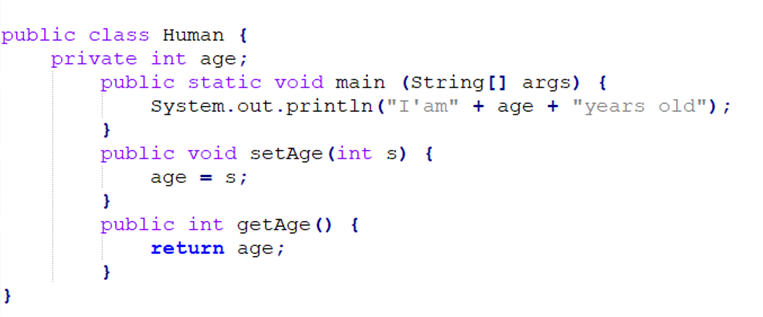
**d.bark();**

But a static method doesn’t use any kind of variables. If you want to call a static method, just use the class name.

Ex:

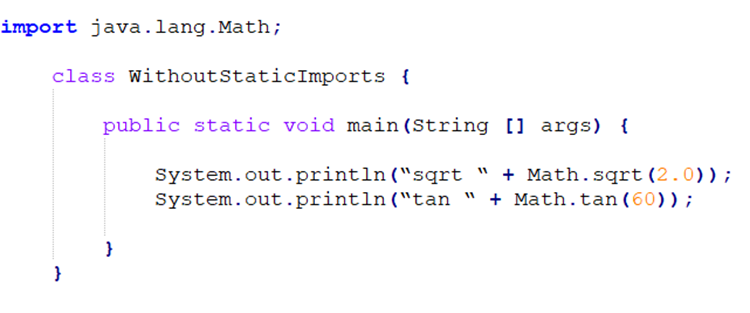
**Math.min(50,100);**

Which means a static method can run without any instance of that particular class. However, if you try to create an instance, it will throw an error. I know you must be having multiple questions, but I’ll try to cover as much as possible. Let's consider the following example.

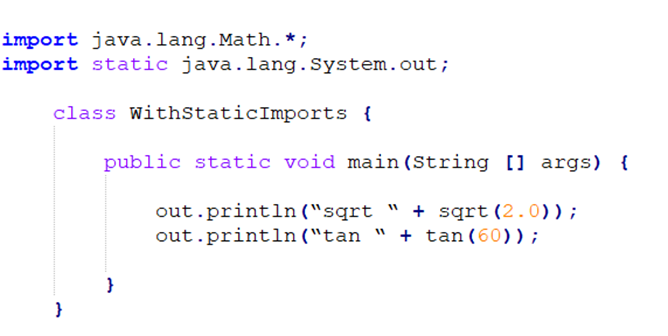


Will this work? No, it WON’T. The compiler cannot figure out the value of the instance variable age. There can be multiple objects on the heap. What specific instance is this referencing? This means static methods can’t use instance variables. Then what about methods. Can a static method use a non-static method? A non-static method usually uses non-static variables right! So, it’s the same answer. It doesn’t know the value of a non-static variable inside a non-static method.

Additionally, static methods could be imported. Which means we can simply use the class name where it is necessary as shown below



You just import it once on top of the source code.



This might be useful to some people but some might find it confusing. At the end of the day, you get to decide which way is the easiest.

Ok so now we know about static methods. What about static variables. A static variable is used throughout the entire class NOT per instance. To mark a variable static just use the keyword static in front of the variable name.

Ex:

**private static int age = 0;**

Usually, a static variable is initialized when the class is loaded. Static variables in a class are initialized before any object of that class can be created or any static method of the class runs. A variable marked ‘final’ means the value will never change once initialized. Which means a ‘static final’ variable will never change as long as the class is loaded. Let’s consider ‘PI’ in the Math class.

Ex:

**public static final double PI = 3.141592653589793;**

· As you already know any variable marked **public** can be accessed outside the class

· The variable is marked **static** because you don’t need to create an instance

· And at last, the keyword **final** is used because the value of PI never changes.

Variables marked as static final are called constants.

As shown above a value of a variable named final can never change. Moreover, a method named final can never be overridden and a class named final can never be extended.

Before I move on to some numbering conventions, lets move on to an entirely different topic. ‘Wrapping a primitive’

I know this sound out of the blue but I’ll try to keep as simple as possible. We already know that an Arraylist is an upgraded version of a usual array that allows us to enter and remove data or rather elements easily. But at times we insert primitive data types to an Arraylist without any issue.

Ex:

**int x = 10;**

**ArrayList list = new ArrayList();**

**List.add(10);**

If you compile this using a version greater than Java 5.0, it will work successfully. But does the add() method of an Arraylist take primitives? No it doesn’t. It only takes object references. Then how did this work? This is because of a feature called autoboxing used in Java versions greater than Java 5.0. Autoboxing is a mechanism which takes a primitive and wraps it or unwraps it according to the method. This allows the primitive to act as an object. Usually wrapping a primitive is kind of a hassle.

Ex:

**Int x = 10;**

**Integer wrapVal = new Integer(x);**

As you may have noticed I used the keyword ‘Integer’ to wrap an int value. Likewise all primitives have wrappers that has the same name but in different naming conventions

**double -> Double**

**float -> Float**

**int -> Integer**

**long -> Long**

**bool -> Boolean**

You can also unwrap a primitive as shown below.

Ex:

**int unWrapVal = wrapVal.intValue();**

Autoboxing can also be used in method arguments, return values, Boolean expressions, operations in numbers and assignments. Besides these functionalities, autoboxing is used to convert a string to a primitive.

Ex:

**String s = “5”;**

**int x = Integer.parseInt(s);**

**double d = Double.parseDouble(“125.48”);**

**boolean b = Boolean.parseBoolean(“True”);**

But what about this?

**String s = “Five”;**

**int x = Integer.parseInt(s);**

Will it work? NOPE. The parseInt() method or any other method regarding numbers can only parse numeric values. It can also do this vice versa. Which means it can turn a primitive number into a string.

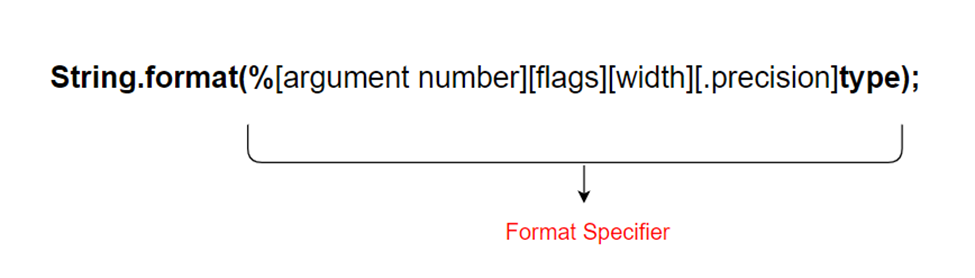
Ex:

**int x = 5;**

**String s = Integer.toString(x);**

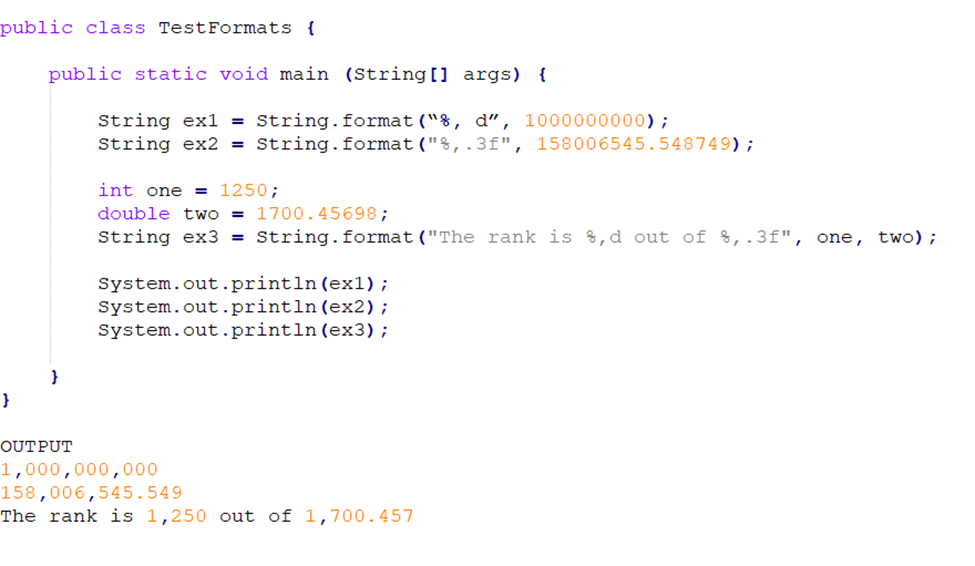
Now let’s move on to formatting numbers and dates in Java.

Some of you may come across an issue where you had to use commas in a long number to make it easier to read. In those types of scenarios, Java provides us with a method that allows us to format a number by adding commas, decimals etc. The formatting is done inside a ‘format()’ method using a format specifier.



“A format specifier can have up to five different parts (not including the “%”). Everything in brackets [ ] below is optional, so only the percent (%) and the type are required. But the order is also mandatory, so any parts you DO use must go in this order.” (Sierra & Bates, 2005)

Ex:



**%, d** — insert commas and format the number as a decimal integer.

**%.2f** — format the number as a floating point with a precision of two decimal places

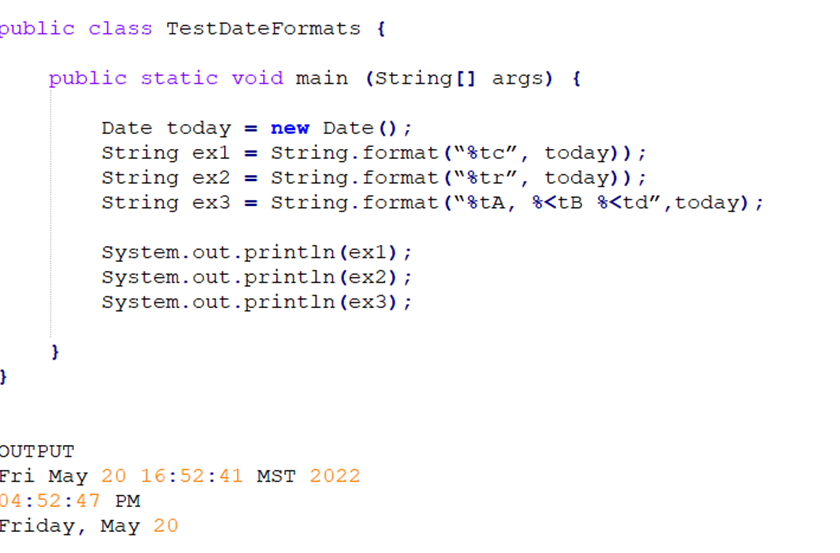
**%,.2f** — insert commas and format the number as a floating point with a precision of two decimal places

Talking about date formatting, we’ll look into 2 classes we can use.

1. **java.util.Date**

Ø This class is mainly used to format the current date and time. Consider the following example.

Ex:



The variable ‘ex1’ is used to display the complete date and time while ‘ex2’ is used to display only the current time. ‘ex3’ has 2 arguments. One indicating the current day, month and date respectively, and the other the date object. The angle bracket ‘<’ is used to get the same object on all of the specifiers instead of passing 3 objects for the 3 specifiers.

**Duplicating the same argument**

**String ex3 = String.format(“%tA, %tB %td”,today, today, today);**

**Without duplicating the same argument**

**String ex3 = String.format(“%tA, %<tB %<td”,today);**

1. **java.util.Calendar**

Ø To start off, Calendar is an abstract class. Which means you can’t create an instance of that class but you can get an instance of a concrete subclass. You can call static methods on Calendar since static methods are called using a particular class rather than on a particular instance. One of the static methods of the Calendar class is the getInstance() method which gives an instance of a concrete subclass as mentioned before. There are many advantages when using the Calendar class.

1. Can be used to set and get a Calendar’s year or month

2. Can be used to either increment or decrement a certain period of time from a particular date

3. Date and time can be represented in milliseconds.

Ex:

