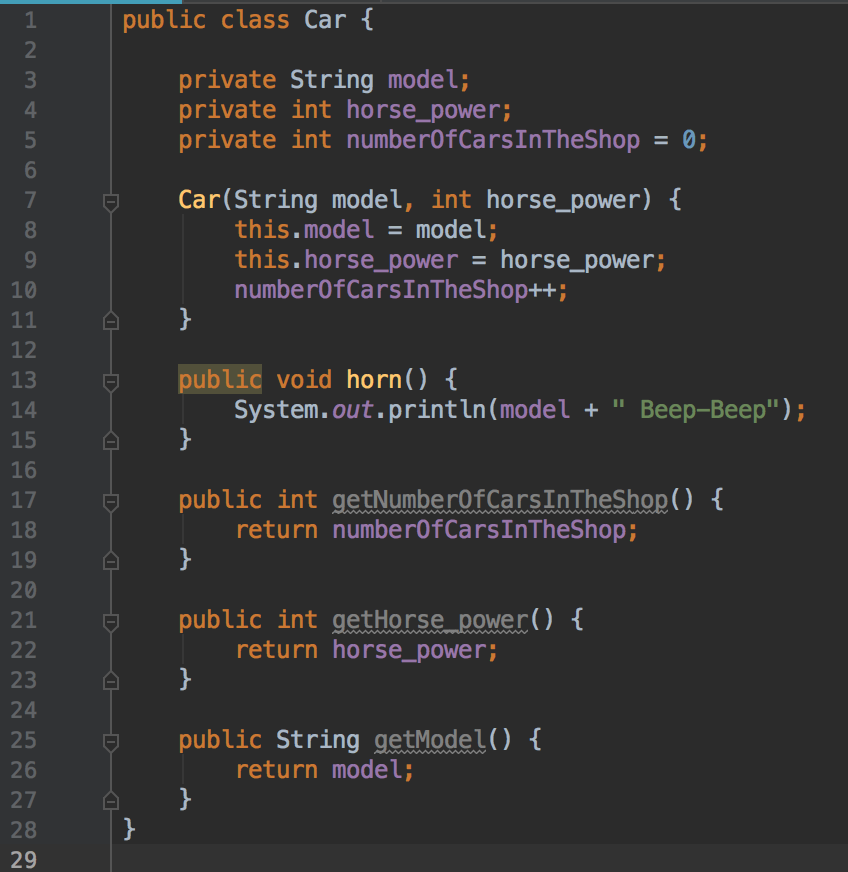
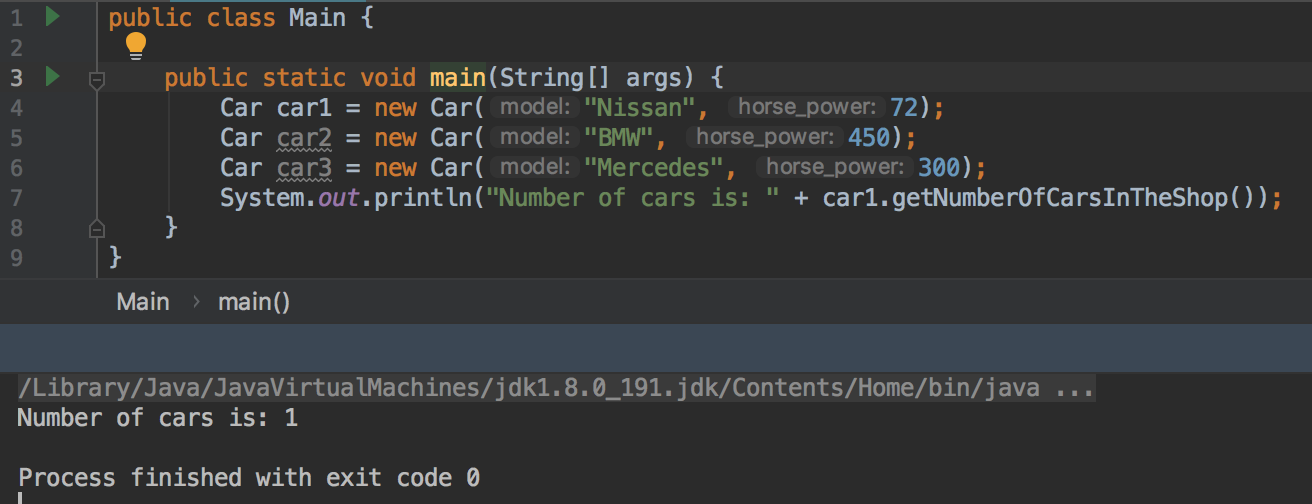
Java "wants" us to think Object-oriented, in other words to think about our program as a collection of objects, because every class in Java implicitly or explicitly extends from Object.class. When you create a class you are describing how instances of that class - objects should behave. You can not use variable and methods of your class, until you instantiate it with **new** keyword, and at that point **JVM** allocates some memory on the **Heap**, stores the address of you instance on **Stack** and methods become available(You can read about Stack and Heap in more details in my [**previous article**](https://www.linkedin.com/pulse/what-where-stack-heap-maxim-malisciuc/). Ok, that's clear. But...

Let's think a little. Imagine the situation: You have to make an app, for a Car Shop, to help them to keep track of all the cars. So when a new model comes, you need to increment the counter and when the car is sold to decrement. First think you would do is creating "Car.class" as a model.

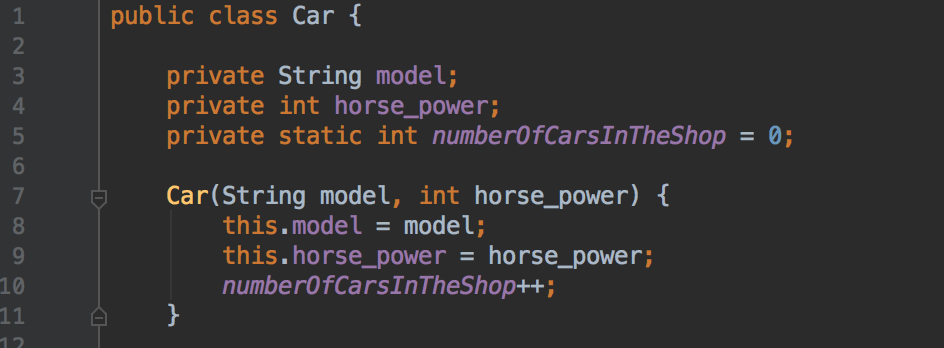


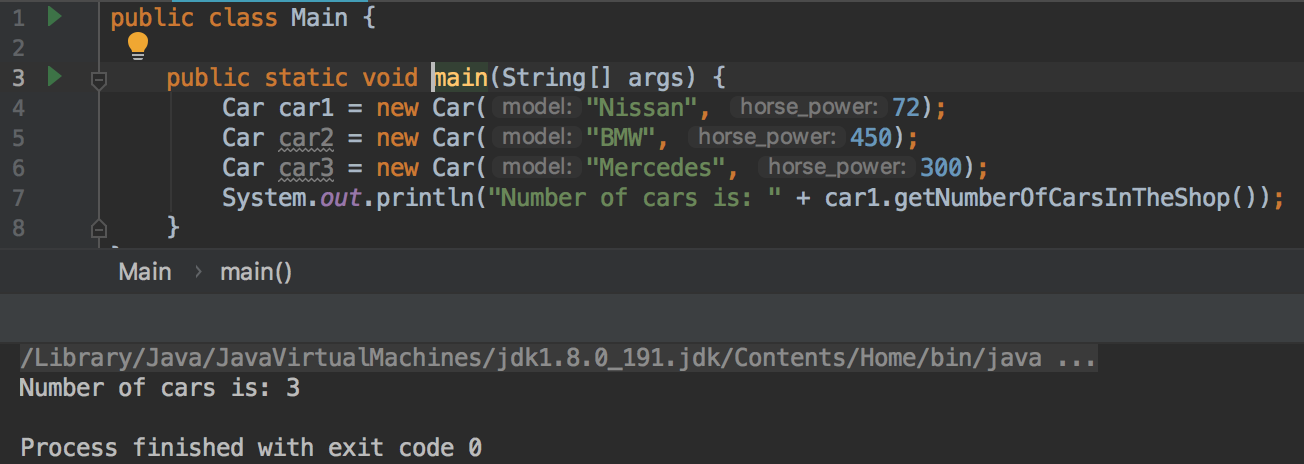
And for the example purposes, I would create a few objects. And will try to count the cars.



Off, Houston, we have a problem. If you know what Stack and Heap, then you would probably know, what is going on. The problem is - we can not count our cars this way, because every time we create an object, a new Heap Space allocates for each object and for each of the variables inside of the class. So we will end up with 3 copies of *numberOfCarsInTheShop,* being not interconnected between them.

When you say something is **static**, it means, that data is not tied to any particular object instance of the class. With ordinary non-static method, you have to create an instance of the class and use it to access the method. Of course, since static methods does not need an instance to be called, they cannot directly access non-static methods or members(because the instance is not create therefore does not have any address, so the method would not know what to call). When we create a static variable or method it is stored in the special area on heap: **PermGen(Permanent Generation)**, where it lays down with all the data applying to classes(non-instance data). Starting from Java 8 the PermGen became **- Metaspace**. The difference is that Metaspace is auto-growing space, while PermGen has a fixed Max size, and this space is shared among all of the instances. Plus the Metaspace is a part of a Native Memory and not JVM Memory. From this point I will use "Metaspace" definition. Let's see, what is going to happen if we change our variable to static.



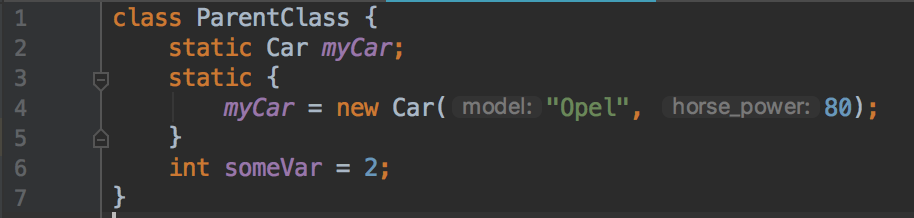


We can see it works, that is because we end up, having space allocated just once for *numberOfCarsInTheShop* in Metaspace and we can create as many instances as we want - the variable will refer to the same piece of memory, the same applies to the static methods.

A **static** variable is initialised only, when the class is first loaded into the JVM, which happens the first time it is referenced in code, **not** when the instance is created.

**Static object references and static methods.**

**Static object reference:**



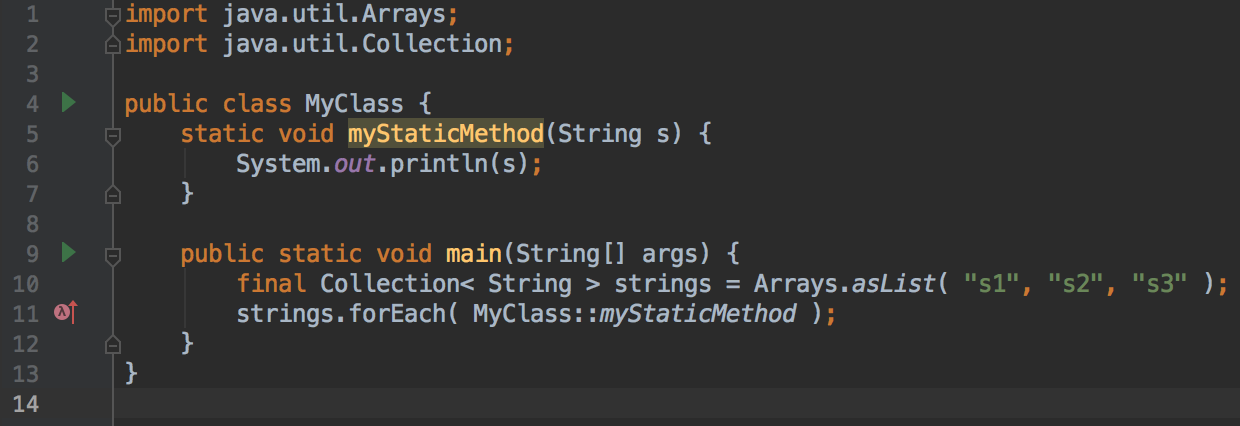
This means, that *myCar* will be created just once and shared among all the instances of the Parent class, whether we create them or not. The car-object-Opel will be stored in the Heap section for instances, while the variable *myCar* which stores the address of the instance will be stored in **MetaSpace**. In this image you can see a **static block**, where *myCar* variable is initialised. We will come back to it later.

**Static methods** are stored in **Metaspace** space of native heap as they are associated to the class in which they reside not to the objects of that class. But their local variables and the passed arguments are stored in the stack. Since static methods belong to the class - they can be called without creating a object of the class. The use-case of static methods is the same as with static variable. Also one of the frequent use-cases you may see is **Utils** and **Helpers** classes, which often become containers for all kind of non-related methods which have not found other place to be put in but should be shared somehow and used by other classes, but such design decisions **should be avoided** in most cases: it is always possible to find another way to reuse the required functionality, keeping the code clean and concise. **Java 8** made introduced some functional concepts into the Java language. The foundation of that is treating the methods as data, a concept which was not supported by the language before (however, since **Java 7**, the **JVM** and the Java standard library have already some features to make it possible). Thankfully, with method references, it is now possible. You can call static method like this:

1) *MyClass.myStaticMethod()*via class name.

2) *myClassInstance.myStaticMethod()* via instance name.

3) *MyClass::myStaticMethod*starting from Java 8.



All of us had seen the convention in Java "*public static void main(String[] args);*" Why is like that?

**public** - for JVM to able to call the method from anywhere.

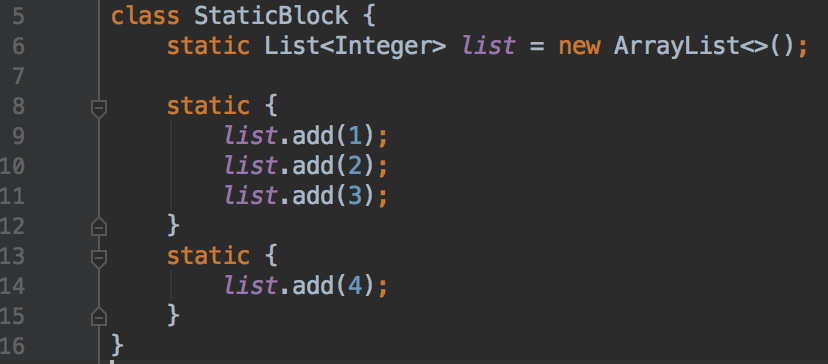
**static** - so JVM can be easily called by the runtime without instantiating the class, since main() is the 1st method to be invoked by JVM and thus we have a true starting point for any program.

**Instance method vs Static method**

* Instance method can access both instance and static methods and variables.
* Static methods can access only static variables and static methods.
* Static methods can’t access instance methods and instance variables directly. They must use reference to object. And static method can’t use "**this**" keyword as there is no instance for "this" to refer to.

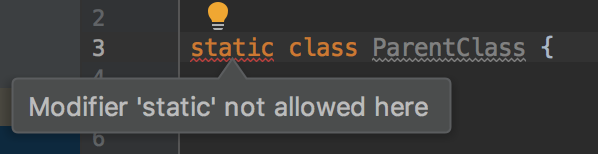
**A static block**

Earlier you have seen the **static block** in the example with **static object reference**. A static block is the perfect place to initialise all of your static fields. It is also called, when the class is first time referenced in the code. You could force this method to be invoked by explicitly calling *ParentClass.init(),* or to use reflection API like this *Class.forName("ParentClass")* but usually it is better to let JVM do it. Let's see an use-case of static block. Because static initialization block can be triggered from multiple parallel threads (when the loading of the class happens in the first time), Java runtime guarantees that it will be executed only once and in **thread-safe manner** + when we have more than 1 static block - it **guarantees the** **sequential execution** of the blocks, so we won't end up of having a list {4, 1, 2, 3}, it will always be {1, 2, 3, 4}



**A static class**

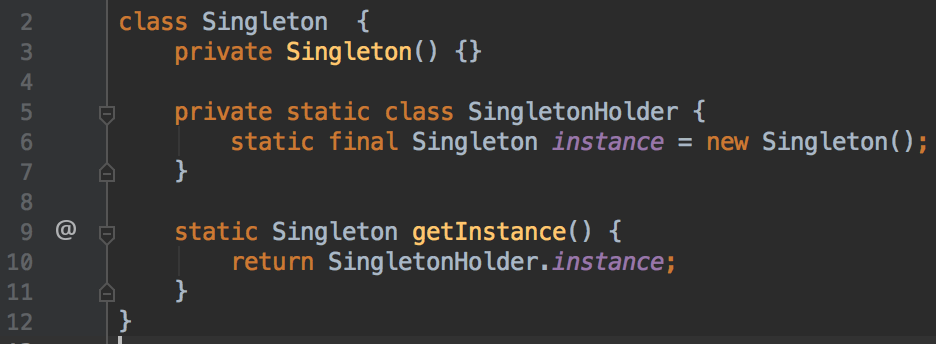
**Can a class be static in Java ?** Yes, it can. We have already seen: static variables, static methods and static blocks. Let's have a look on static class and why do we need it.



Oops, not cool, it seems, that Java does not let us create a static class. Yes, we can not mark our "top-level" class as static, because it is useless, think about it, why do we need to mark our class as static?? We can refer to it, without it being static :) But... Java allows us to define a class within another class. Such a class is called a **Nested class**. The class which enclosed nested class is known as **Outer class**. In Java, we can’t make Top level class static. *Only nested classes can be static*. In Java we have [**Nested**](https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html) and [**Inner**](https://docs.oracle.com/javase/tutorial/java/javaOO/innerclasses.html) classes. The differences are that **"Nested"** is static and "**Inner**" is not. The Inner classes have access to all member of the enclosing class (including private), whereas the *static*nested classes only have access to static members of the outer class. We will talk only about *Nested classes*, as the topic is static keyword. Nested classes:

* *Are considered "****top-level****".*
* *Do not require an instance of the outer class to be constructed.*
* *May not reference the containing class members without an explicit reference.*
* *Have their own lifetime.*

**Nested** classes have a lot of use-cases. You may see it as an extension for the local class, so it if your requirements are similar to those of a local class, you want to make the type more widely available, and you don't require access to local variables or method parameters - use it. One of the is the most widely used approach of creating [**singleton objects**,](https://www.tutorialspoint.com/design_pattern/singleton_pattern.htm) because it is easy learn and implement.



**What about static in Kotlin?**