Strategy Design Pattern

Notes By Bhavuk Jain

Design patterns allow us to write *reusable, scalable, and maintainable* code. One such pattern in System Design is **Strategy Design Pattern**

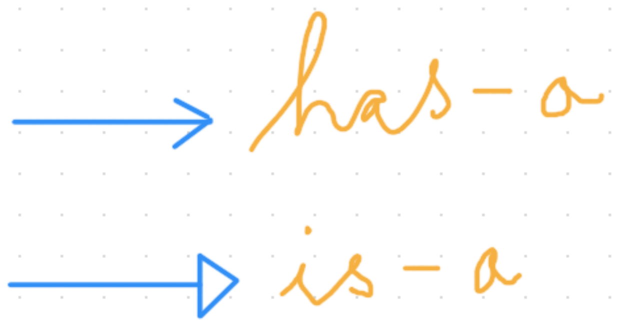
I will be covering the problem and, the core idea behind this pattern along with it's code implementation.

(Tip: Remember to read through the code comments as well for better perspective)

What & Why?

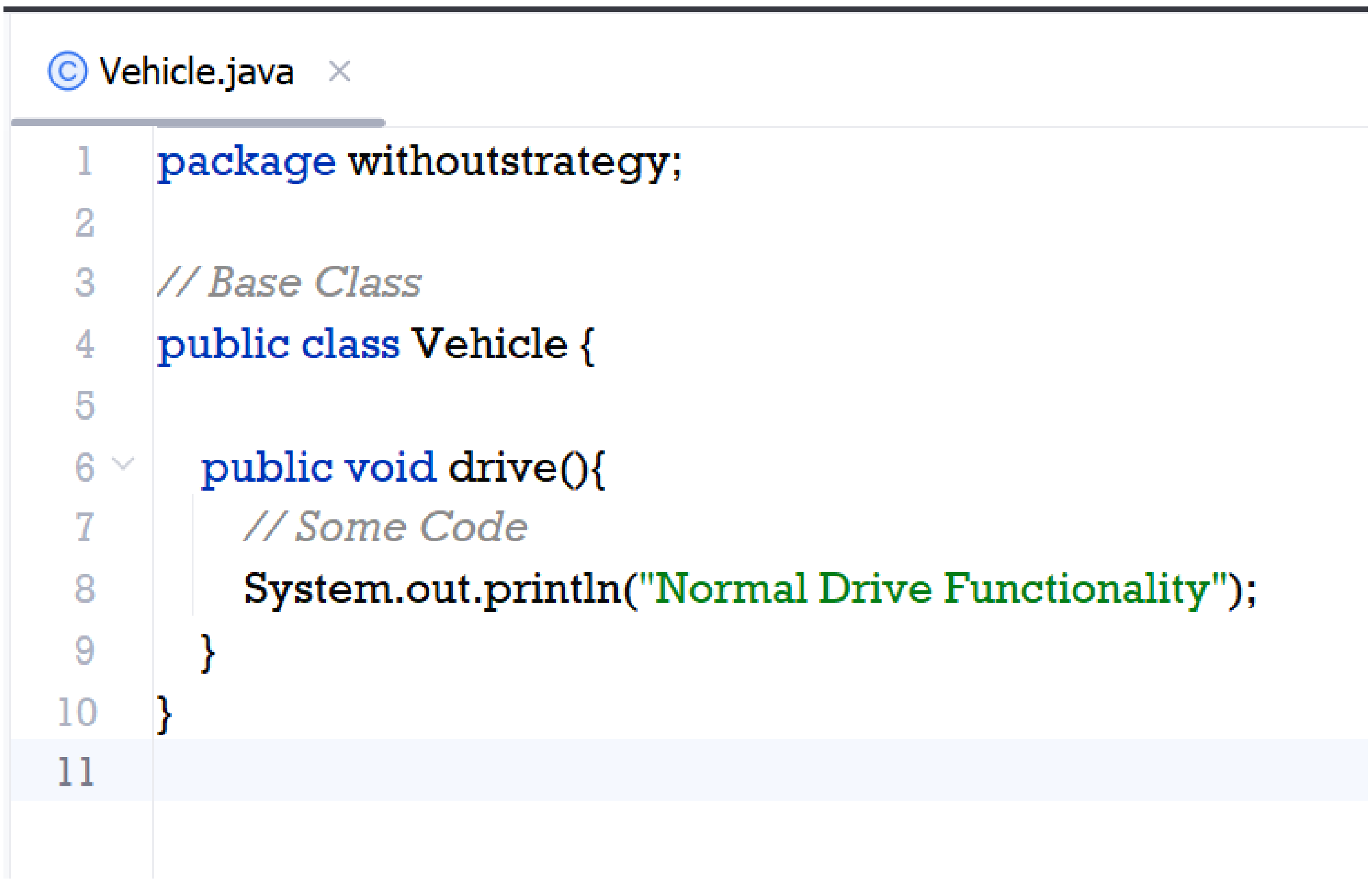
Strategy Design Pattern focuses on **DRY** (Don't Repeat Yourself) principle. It aims to *reduce code duplication* or *redundancy,* primarily when there are multiple sub-classes that inherit some features from a base class and also want to provide additional features on their own which are not already present in the base class.
Assuming, you are familiar with the basic OOPs concepts, we are good to go to dive deep into this design pattern.

In OOPS, ***has-a*** relationship is based on composition, whereas,***is-a*** relationship is based on inheritance.
These relationships are denoted as follows in Block diagrams:



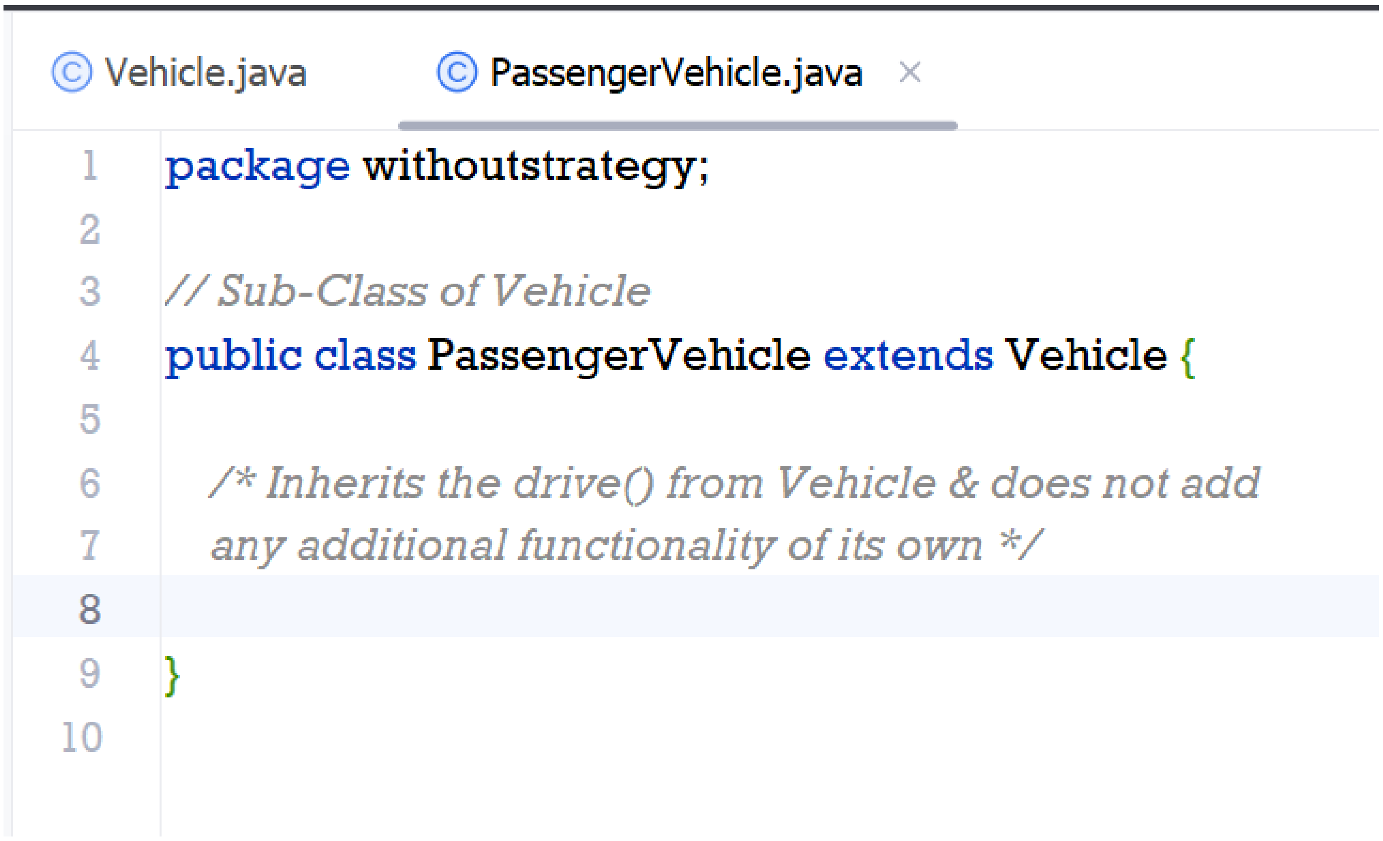
Problem Without Strategy Pattern

Let's say, we have a Base Class **Vehicle**, which has a Normal Drive functionality implemented using **drive()** method.



```java
package withoutstrategy;

// Base Class
public class Vehicle {

    public void drive(){
        // Some Code
        System.out.println("Normal Drive Functionality");
    }
}

```

We have another class called **PassengerVehicle** which is inherited from **Vehicle** (thus forming an **'is-a'** relationship) because PassengerVehicle is a Vehicle. It can use the **drive()** method of **Vehicle** because that would also be inherited.

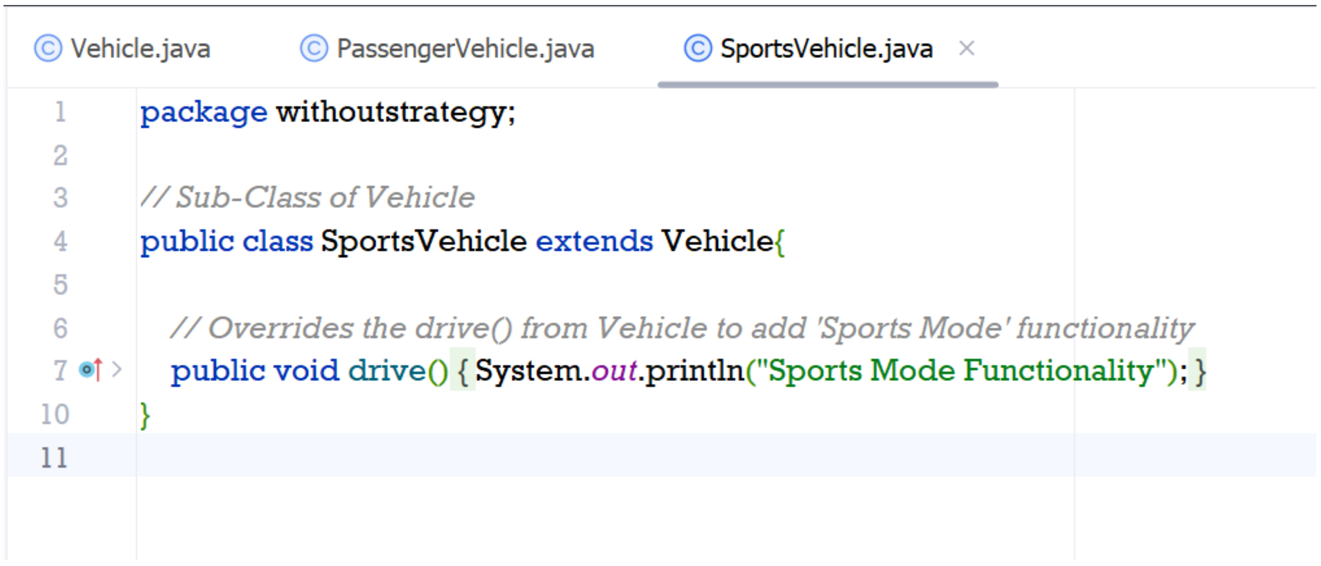


Vehicle.java   PassengerVehicle.java

```java
package withoutstrategy;

// Sub-Class of Vehicle
public class PassengerVehicle extends Vehicle {

   /* Inherits the drive() from Vehicle & does not add
   any additional functionality of its own */

}

```

So, if do something like:
PassengerVehicle **obj** = new PassengerVehicle();
obj.drive(); // It will print "Normal Drive Functionality" as expected.

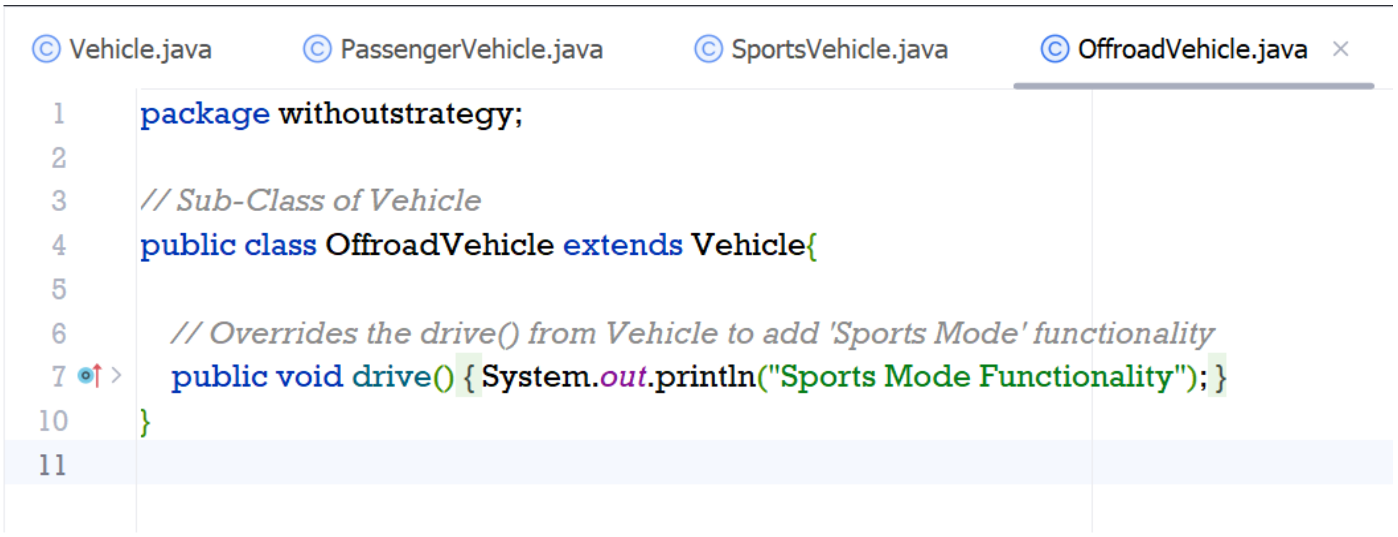
Now, we have another class called **SportsVehicle** which is also inherited from **Vehicle**. It is overriding the **drive()** method from **Vehicle** to provide it's own implementation to it. In this case, it wants to provide it "Sports Mode Functionality".



```java
package withoutstrategy;

// Sub-Class of Vehicle
public class SportsVehicle extends Vehicle{

    // Overrides the drive() from Vehicle to add 'Sports Mode' functionality
    public void drive() { System.out.println("Sports Mode Functionality"); }
}

```

So, if do something like:
SportsVehicle **obj** = new SportsVehicle();
obj.drive(); // It will print "Sports Mode Functionality" as expected.

Similarly, we have another class called **OffroadVehicle** which also inherits from **Vehicle** and overrides it's **drive()** method to provide it's own functionality which is again the "Sports Mode Functionality".



```java
package withoutstrategy;

// Sub-Class of Vehicle
public class OffroadVehicle extends Vehicle{

    // Overrides the drive() from Vehicle to add 'Sports Mode' functionality
    public void drive() { System.out.println("Sports Mode Functionality"); }
}

```

So, if do something like:
OffroadVehicle **obj** = new OffroadVehicle ();

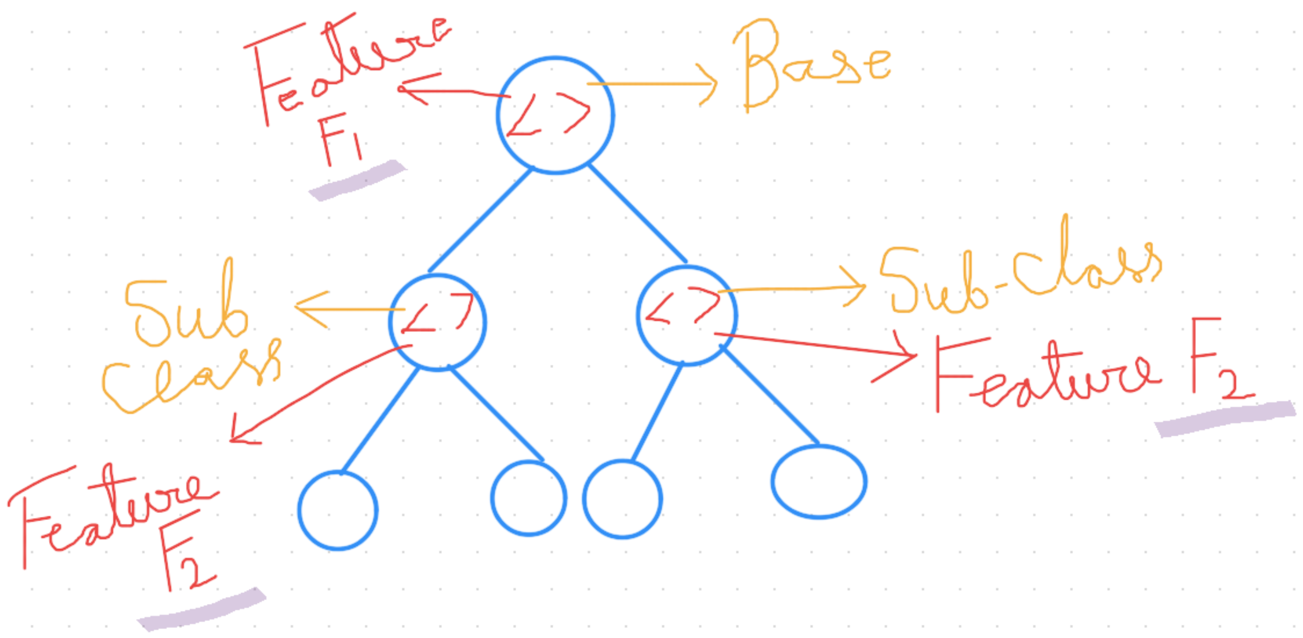
obj.drive(); // It will also print "Sports Mode Functionality" as expected.

Now, as you can notice, **SportsVehicle** and **OffroadVehicle** both want to implement the same functionality i.e., **Sports Mode Functionality**. But, as the base class **Vehicle** does not provide the implementation of **drive()** with Sports Mode Functionality, these two sub classes have to provide this functionality on their own.

So what's the actual problem here?

The problem lies in the fact that we are repeating the code to implement **Sports Mode Functionality** in both the sub-classes. **SportsVehicle** and **OffroadVehicle** both want to override the **drive()** method of base class **Vehicle** to add the same feature that is the "Sports Mode Functionality".
At first, this might not seem to be an issue to worry about as currently we only have a single **System.out.println** statement inside the **drive()** method but as soon as we start adding more classes to our system that require this "Sports Mode Functionality", things would really go out of hand as we would be repeating a lot of code.

You can imagine this scenario in this way:-



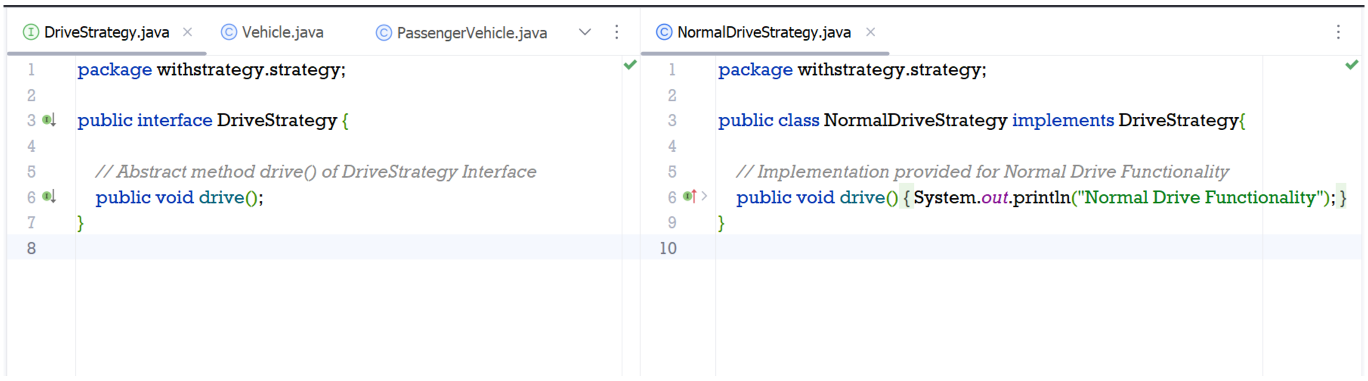
We have a **Base Class** which has a **Feature F1**. And, we have two sub-classes that inherit from Base Class and override Base Class' implementation of Feature **F1** with Feature **F2**.
Notice that, both the sub-classes want to have a Feature F2. In this case, we are **duplicating the code for Feature F2 in both the sub-classes,** just because **F2** was not provided by base class **Vehicle**.

How can we solve this problem?

Now, that we have understood the problem, we can go on to find the solution for the same.
The solution is very straight-forward where we would separate out or move out the logic for **drive()** method from Vehicle Class.
To do that, we would be creating one interface called **DriveStrategy** and an abstract method **drive()** inside it. Also, we would create a few **strategy classes** that would provide their own implementation to the **drive()** abstract method of **DriveStrategy** interface.

Code Snippets:

1) On the left, we create an **interface** called **DriveStrategy** which will have an abstract method **drive()** in it.
2) On the right, we create a class called **NormalDriveStrategy** which implements **DriveStrategy** interface and provides the implementation to its **drive()** method.



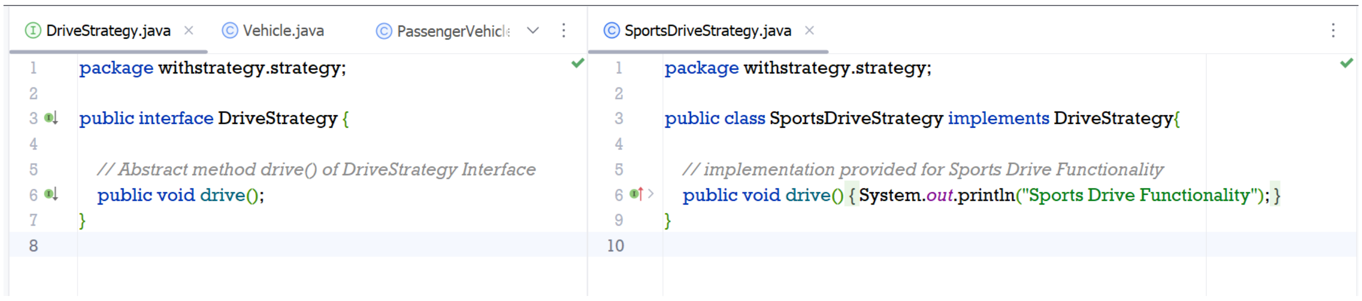
DriveStrategy.java

```java
package withstrategy.strategy;

public interface DriveStrategy {

    // Abstract method drive() of DriveStrategy Interface
    public void drive();
}
```

NormalDriveStrategy.java

```java
package withstrategy.strategy;

public class NormalDriveStrategy implements DriveStrategy{

    // Implementation provided for Normal Drive Functionality
    public void drive() { System.out.println("Normal Drive Functionality"); }
}
```

3) On the right, we again create a class called **SportsDriveStrategy** which implements **DriveStrategy** interface and provides the implementation to its **drive()** method.



DriveStrategy.java

```java
package withstrategy.strategy;

public interface DriveStrategy {

    // Abstract method drive() of DriveStrategy Interface
    public void drive();
}
```

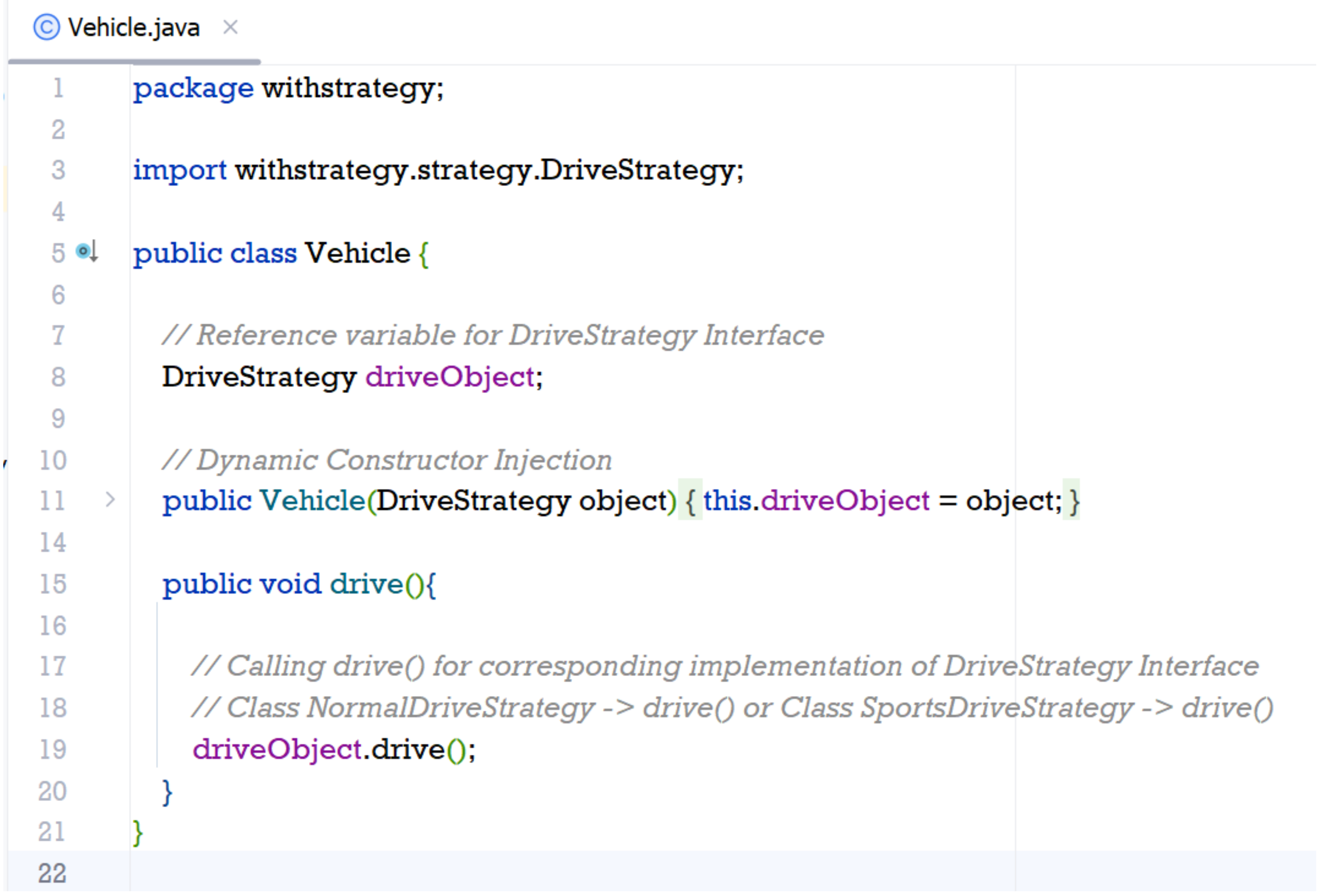
SportsDriveStrategy.java

```java
package withstrategy.strategy;

public class SportsDriveStrategy implements DriveStrategy{

    // implementation provided for Sports Drive Functionality
    public void drive() { System.out.println("Sports Drive Functionality"); }
}
```

Till here, we have created one interface and two classes that provide their own implementation to the abstract method of that interface. Now, that our strategies are ready, we shall see how to use them.

1.) We created a Vehicle Base Class that has the following:

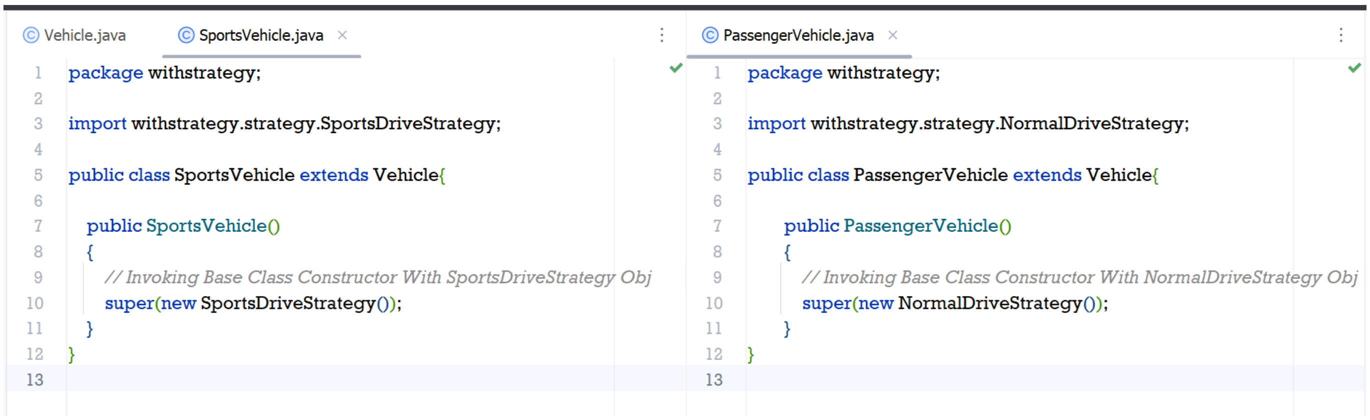
a) A *reference variable* of type **DriveStrategy**. Notice, we haven't assigned any object/instance to this reference variable because we want to keep it dynamic.
b.) A parameterized constructor which would accept an object/instance from the strategy classes we created. Either it can be **NormalDriveStrategy**or **SportsDriveStrategy**. After receiving the object, we would then assign it to the reference variable **driveObject**.
c.) A **drive()** method that would call the **drive()** method of the respective strategy class whose instance we have received (either NormalDriveStrategy or SportsDriveStrategy).


Vehicle.java

```java
package withstrategy;

import withstrategy.strategy.DriveStrategy;

public class Vehicle {

    // Reference variable for DriveStrategy Interface
    DriveStrategy driveObject;

    // Dynamic Constructor Injection
    public Vehicle(DriveStrategy object) { this.driveObject = object; }

    public void drive(){

        // Calling drive() for corresponding implementation of DriveStrategy Interface
        // Class NormalDriveStrategy -> drive() or Class SportsDriveStrategy -> drive()
        driveObject.drive();
    }
}

```

2.) On the left, we have created a **SportsVehicle** Class that extends/inherits the **Vehicle** Base Class and on the right also, we

created a **PassengerVehicle** Class that extends/inherits the **Vehicle** Base Class.
Notice, how in their respective constructors, we have invoked the parent class constructor using **super** keyword by passing the objects of **SportsDriveStrategy** and the **NormalDriveStrategy** classes.



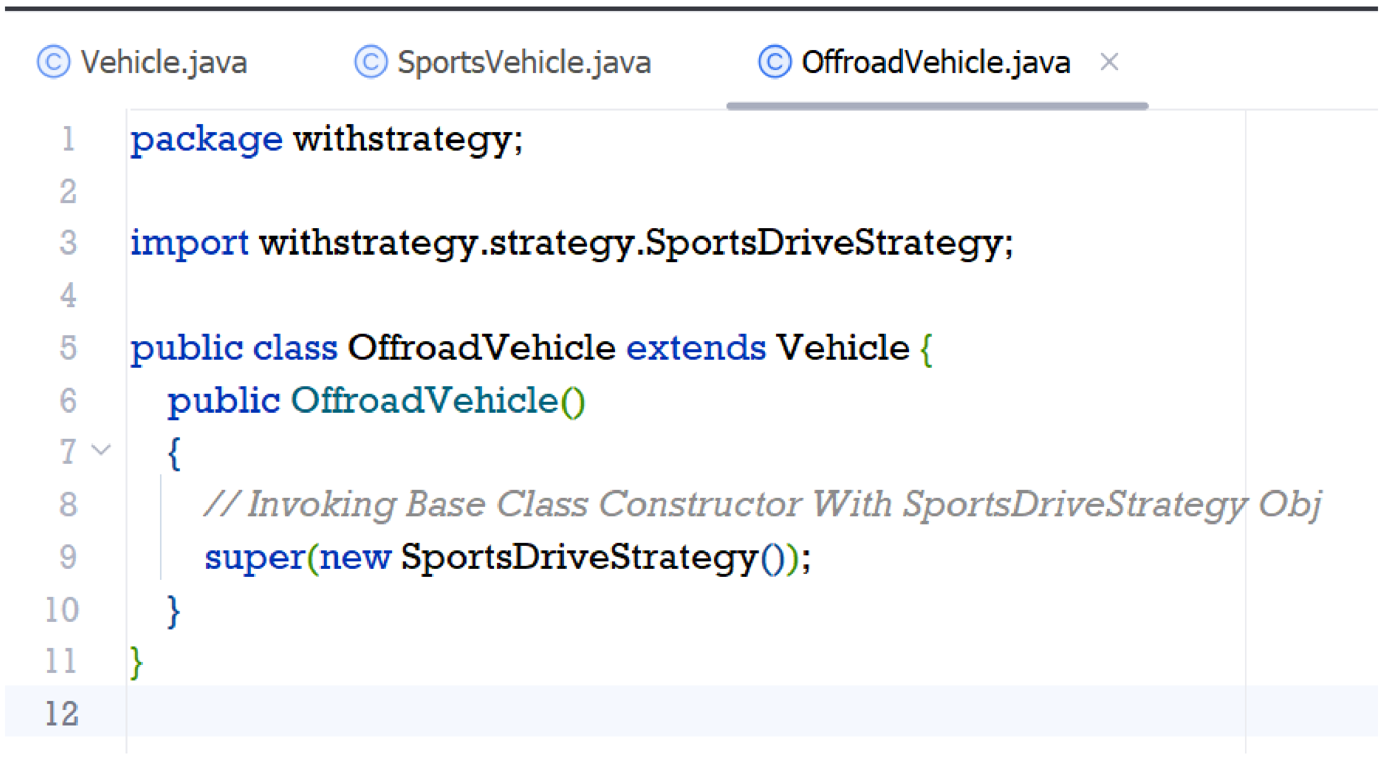
```java
package withstrategy;

import withstrategy.strategy.SportsDriveStrategy;

public class SportsVehicle extends Vehicle{

    public SportsVehicle()
    {
        // Invoking Base Class Constructor With SportsDriveStrategy Obj
        super(new SportsDriveStrategy());
    }
}
```

```java
package withstrategy;

import withstrategy.strategy.NormalDriveStrategy;

public class PassengerVehicle extends Vehicle{

    public PassengerVehicle()
    {
        // Invoking Base Class Constructor With NormalDriveStrategy Obj
        super(new NormalDriveStrategy());
    }
}
```

What are we actually doing here?

The sub-class **SportsVehicle** is inherited from **Vehicle**. Also, it has a default constructor inside which we are again calling the constructor of its parent class using **super** keyword.
To put things into perspective, just scroll above a little, and you can see the **Vehicle** Class implementation in which there is a **parameterized constructor** which takes an object.
Now, in the **SportsVehicle** and **PassengerVehicle** sub-classes what we are doing is called **Constructor Chaining**, wherein we can call a constructor of the Base Class from its Sub-Class.

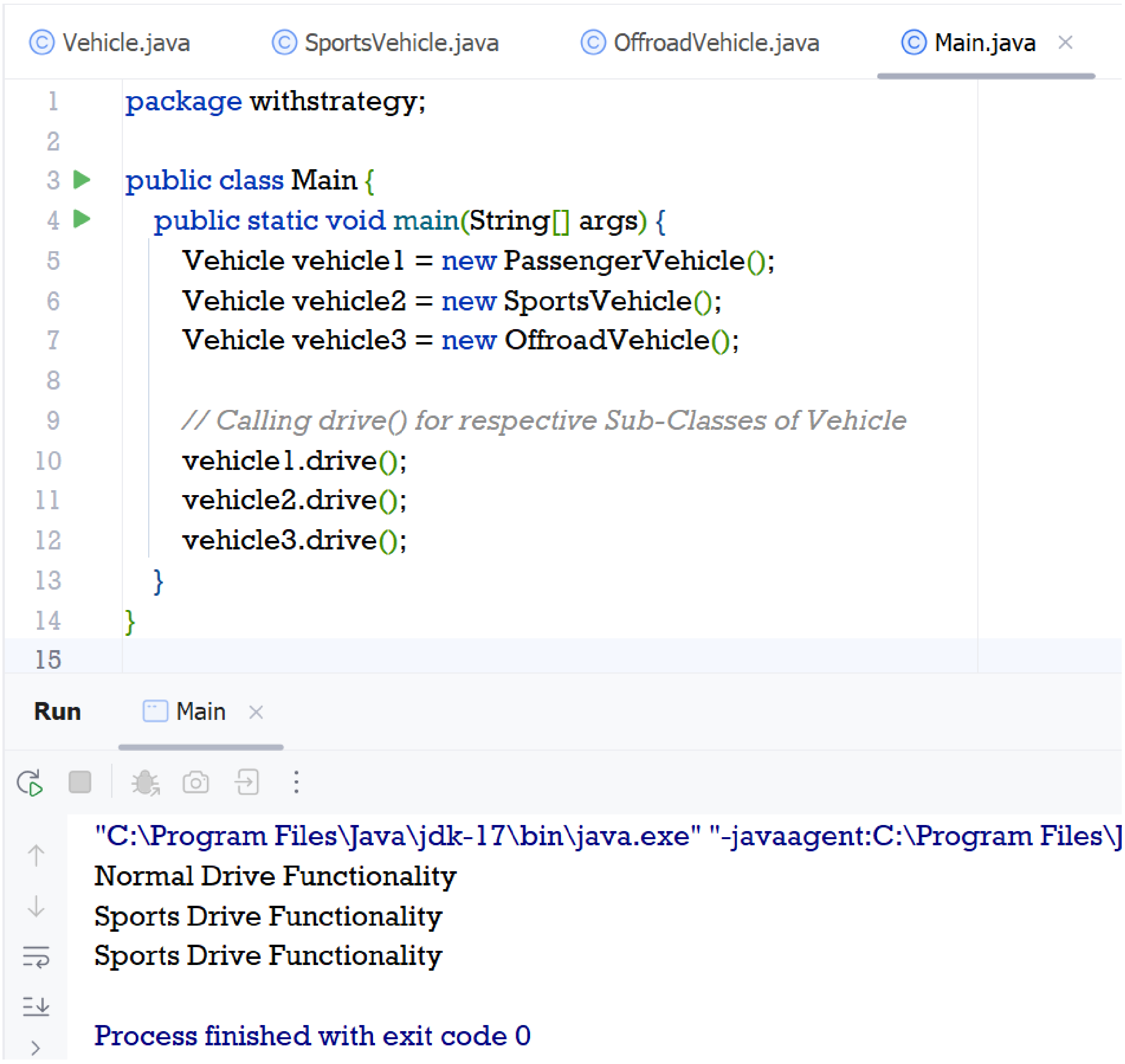
Let's say we have another category of **Vehicle** called **OffroadVehicle** which also wants to have the *Sports Mode Functionality*, so we do it like this:-


Vehicle.java   SportsVehicle.java   OffroadVehicle.java

```java
package withstrategy;

import withstrategy.strategy.SportsDriveStrategy;

public class OffroadVehicle extends Vehicle {
    public OffroadVehicle()
    {
        // Invoking Base Class Constructor With SportsDriveStrategy Obj
        super(new SportsDriveStrategy());
    }
}

```

Finally, it's time to test our implementation:-

We create 3 reference variables of Vehicle: vehicle1 | vehicle2 | vehicle3 And, we have assigned them the objects of **PassengerVehicle**, **SportsVehicle** and, **OffroadVehicle**.
Upon creating the object of **PassengerVehicle**, it would invoke the **Vehicle** Class constructor with **new NormalDriveStrategy()** object. Which, in turn allows us to call the **drive()** method of **NormalDriveStrategy** class from the **Vehicle** class after dynamically assigning the object.
Similarly for **SportsVehicle** and **OffroadVehicle** as well, we are calling the parent class Vehicle's constructor with **new SportsDriveStrategy()** object to achieve the desired functionality.



```java
package withstrategy;

public class Main {
   public static void main(String[] args) {
      Vehicle vehicle1 = new PassengerVehicle();
      Vehicle vehicle2 = new SportsVehicle();
      Vehicle vehicle3 = new OffroadVehicle();

      // Calling drive() for respective Sub-Classes of Vehicle
      vehicle1.drive();
      vehicle2.drive();
      vehicle3.drive();
   }
}

```

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\J
Normal Drive Functionality
Sports Drive Functionality
Sports Drive Functionality

Process finished with exit code 0
```

What have we achieved by doing all this?

Certainly, by separating out the core logic from our base class, we have made our design more scalable in the sense that if, in future, we want to add multiple new strategies that would be shared among multiple classes, we can easily do that. Also, the code would be written once for one strategy and multiple classes can use the same code instead of writing it again and again.

Hope you found this useful. Thanks a lot for reading!

Signing off,

Bhavuk Jain