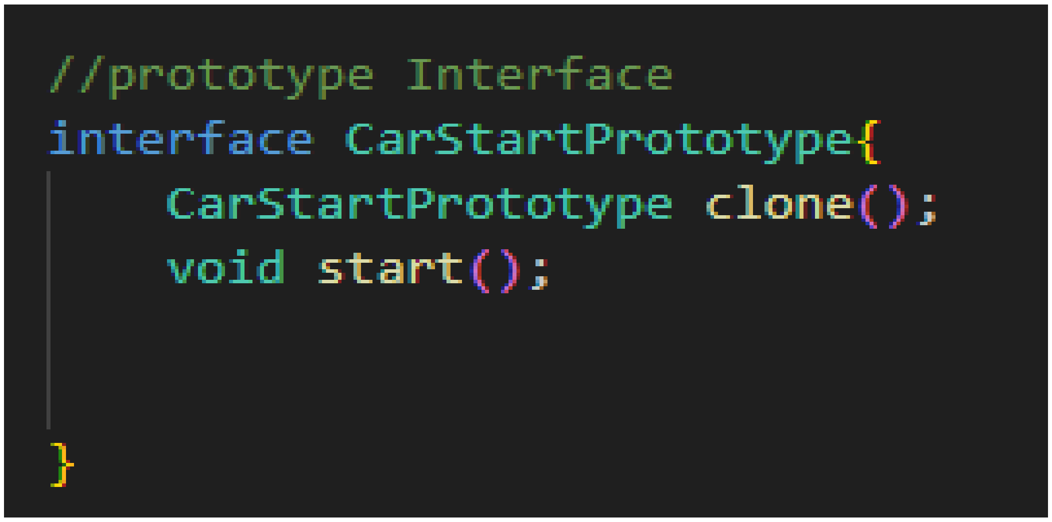
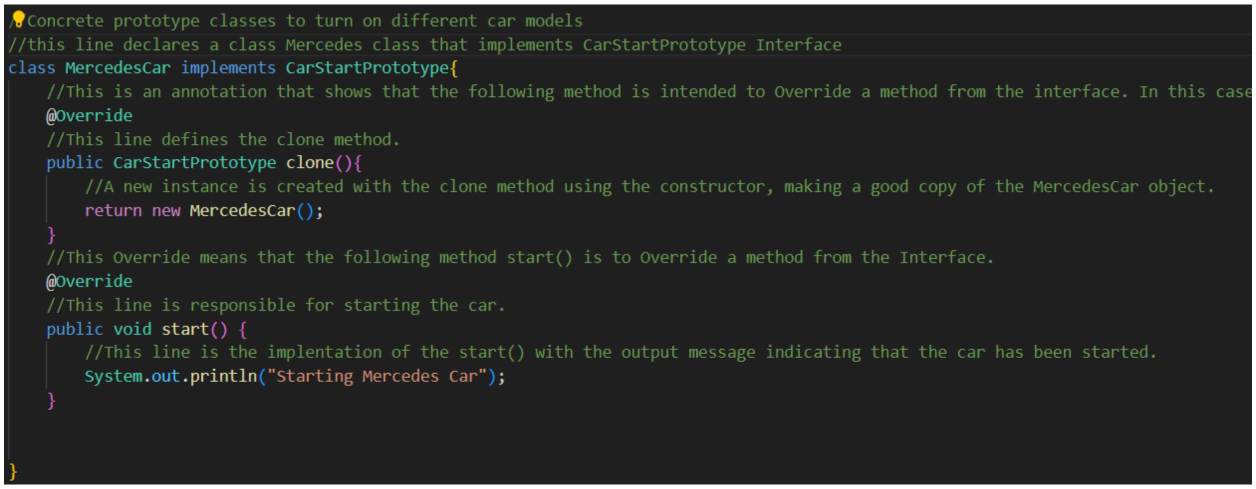
The prototype pattern is a pattern which is used for creating objects by copying and existing object, which is then known as prototype.

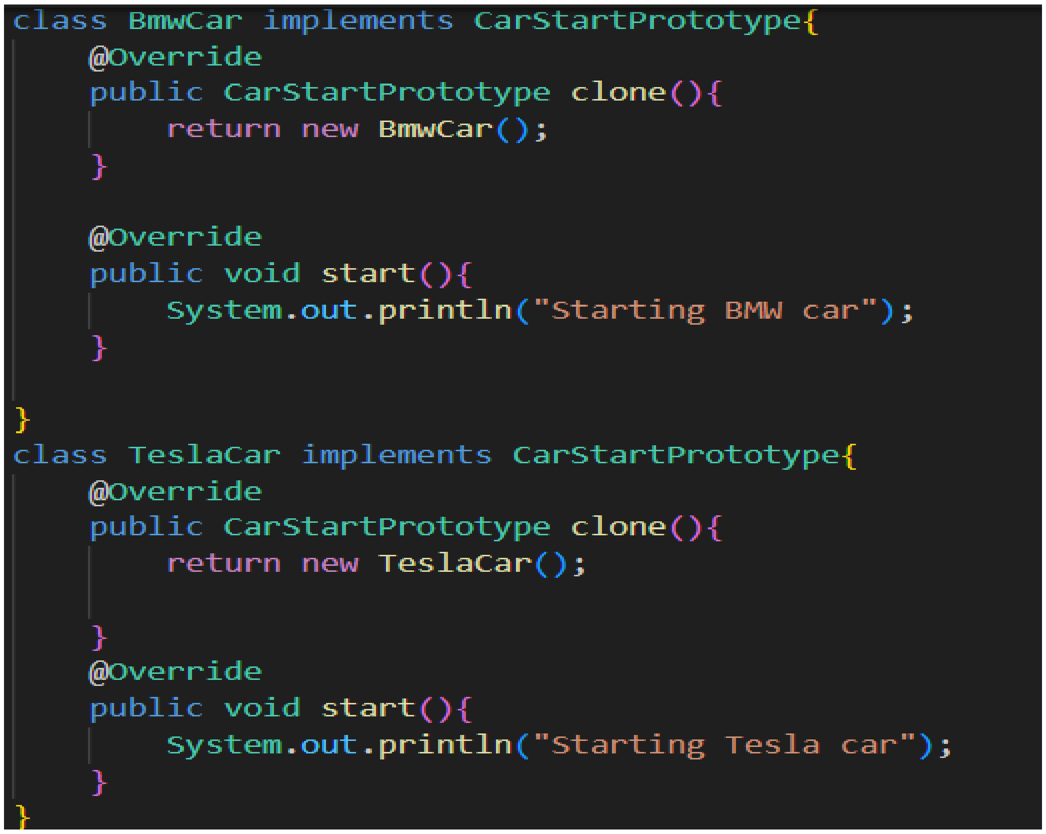


```
//prototype Interface
interface CarStartPrototype{
    CarStartPrototype clone();
    void start();


}
```

Starting with the prototype pattern we have a prototype interface. The interface which I have declares two methods: clone() and start().
The clone() method is a key method required by the Prototype pattern which is responsible for creating a copy of the current object. All concret prototype classes must provide their implementation of the clone().

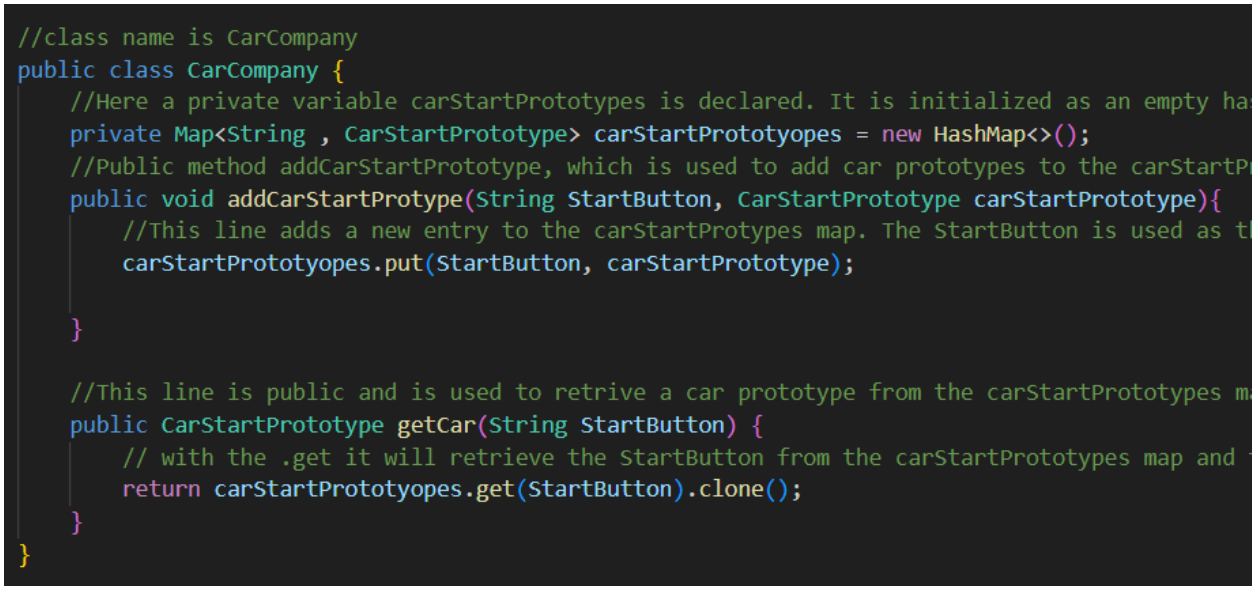


```
//Concrete prototype classes to turn on different car models
//this line declares a class Mercedes class that implements CarStartPrototype Interface
class MercedesCar implements CarStartPrototype{
    //This is an annotation that shows that the following method is intended to Override a method from the interface. In this case
    @Override
    //This line defines the clone method.
    public CarStartPrototype clone(){
        //A new instance is created with the clone method using the constructor, making a good copy of the MercedesCar object.
        return new MercedesCar();
    }
    //This Override means that the following method start() is to Override a method from the Interface.
    @Override
    //This line is responsible for starting the car.
    public void start() {
        //This line is the implentation of the start() with the output message indicating that the car has been started.
        System.out.println("Starting Mercedes Car");
    }


}
```



```
class BmwCar implements CarStartPrototype{
    @Override
    public CarStartPrototype clone(){
        return new BmwCar();
    }

    @Override
    public void start(){
        System.out.println("Starting BMW car");
    }

}
class TeslaCar implements CarStartPrototype{
    @Override
    public CarStartPrototype clone(){
        return new TeslaCar();

    }
    @Override
    public void start(){
        System.out.println("Starting Tesla car");
    }
}
```

The prototype also has concrete classes. I choose MercedesCar BmwCar and TeslaCar. These classes implement the CarStartPrototype interface. Each class has its own implementation of the clone() method which creates a new instance of the same class. This effectively copies this object. This also provides an implementation of the start() method which defines how each car will start. Each car model has its unique start behavior.



```
//class name is CarCompany
public class CarCompany {
    //Here a private variable carStartPrototypes is declared. It is initialized as an empty ha
    private Map<String , CarStartPrototype> carStartPrototyopes = new HashMap<>();
    //Public method addCarStartPrototype, which is used to add car prototypes to the carStartP
    public void addCarStartProtype(String StartButton, CarStartPrototype carStartPrototype){
        //This line adds a new entry to the carStartProtypes map. The StartButton is used as t
        carStartPrototyopes.put(StartButton, carStartPrototype);

    }

    //This line is public and is used to retrive a car prototype from the carStartPrototypes m
    public CarStartPrototype getCar(String StartButton) {
        // with the .get it will retrieve the StartButton from the carStartPrototypes map and
        return carStartPrototyopes.get(StartButton).clone();
    }
}
```

The CarCompany class acts as the client code which is responsible for managing car prototypes. It has a private map carStartPrototypes that stores car prototypes. In my case this map uses a string key (Mercedes,BMW,..) to associate with the corresponding prototype.
Here the addCarStartPrototype allows the client to add car prototypes to the map. This method takes a string key (Mercedes) and the car prototype object (MercedesCar object) and stores them in the map.
The getCar method retrieves a car prototype from the map based on the provided string key (Mercedes). It will first check for the prototype from the map and then it will call the clone() method to create a copy of the car prototype.

The Prototype benefits the code, because it allows me to create new car prototypes by copying existing ones, which is beneficial and time saving for me because I don't need to create new objects from scratch. Each car model has their own unique behavior defined in the start() method and this pattern ensures that these behaviors are preserved in the cloned objects.
The other thing I have a big benefit is that it provides me the flexibility, where I can remove or add prototypes without modifying the client code. This pattern is very good in saving time and getting the job done.
If I would choose not to use this pattern for this code creation, then I would need to create each car object from scratch whenever I want to produce a new car. This could lead to performance issues, in case the car object is hard to create. This would also introduce scalability issues which means that as my application would grow when I introduce a new car model then I would also need to extend the client code which would be a disadvantage if I need to maintain or find errors in the code.

To summarize, the Prototype Pattern simplifies object creation and management, resulting in more efficient, maintainable, and adaptable code to changing requirements.