1. What is dispatcher-servlet in spring? Why is it used for? Shortly explain about how spring-boot lifecycle works.

   -> Dispatcher servlet is the front controller in spring web application. The dispatcher servlet receives all the requests and delegates them to the appropriate controller. It is used to create web applications and rest services in spring mvc.

2. Write a few differences between “git merge” and “git rebase”. When will you choose git merge and when will you choose git rebase?

   git merge:
   1. Keeps the original history intact.
   2. Fully traceable history.
   3. Maintain the context of the branch.

   git rebase:
   1. History is flat and easy to read.
   2. Avoids merge commit “noise” in busy repos with busy branches.
   3. Cleans intermediate commits by making them a single commit.

   If the remote branch is very busy, better to use git merge. In local, rebase is best the choice.

3. Suppose you made a wrong commit and pushed the code into your remote branch. Now you feel the push needs to be reverted / undoed or the previous head should be back into the remote branch. What would you like to do?

   In that case we can find commit commit log from the following command,

   ```
   git log --online
   ```

   Then delete the recent commit by following command,

   ```
   git reset HEAD~
   ```

   Then forcefully push the code into the remote branch by following command,

   ```
   git push origin <branch-name> --force
   ```

4. What do you understand about J2EE? What are the main J2EE components? Briefly explain EJB

   J2EE:

   J2EE stands for Java 2 Enterprise Edition. The functionality of J2EE is developing and deploying multi-tier web based enterprise applications.

   J2EE Components:

   1. Enterprise JavaBeans(EJB)
   2. Java Servlets
   3. JavaServer Pages(JSP)
   4. Java Database Connectivity(JDBC)
   5. Java Message Service (JMS)
   6. Java Transaction API (JTA)
   7. JavaMail

   EJB:

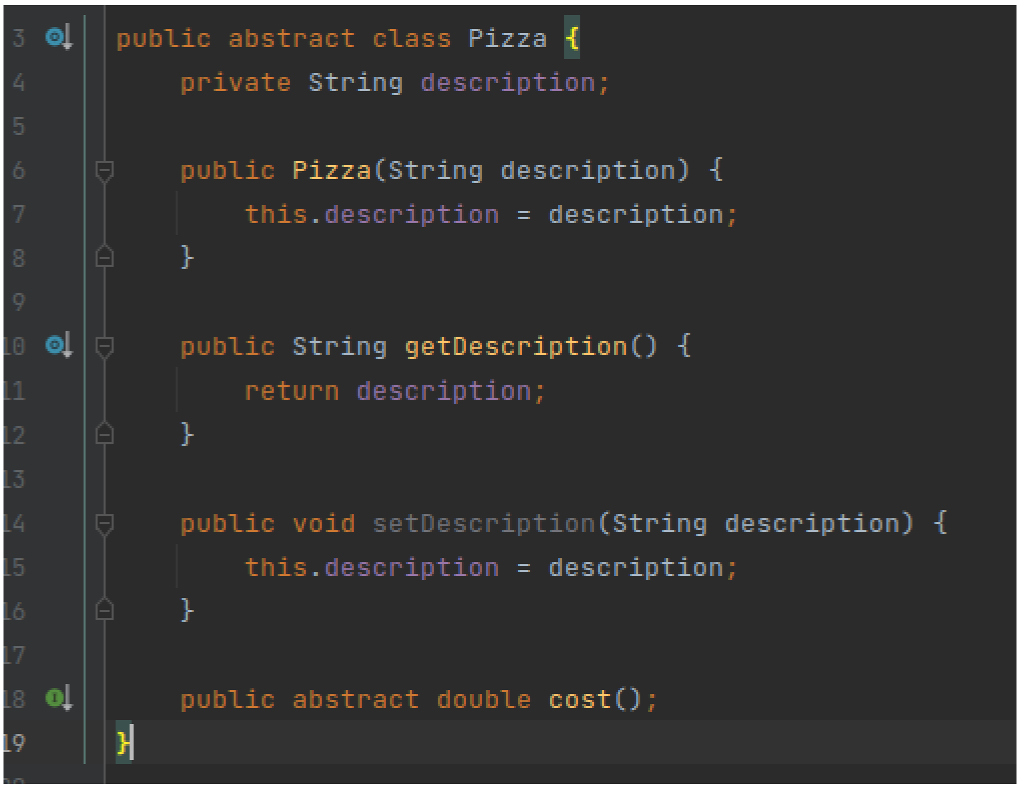
   EJB is a component technology that helps developers create business objects in the middle tier. These business objects (enterprise beans) consist of fields and methods that implement business logic. EJBs are the building blocks of enterprise systems. They perform specific tasks by themselves, or forward operations to other enterprise beans. EJBs are under control of the J2EE application server.

5. Suppose you started a pizza restaurant with two main types, margarita and classic pizzas. Once customers start coming in, they demand add-ons like mushrooms, onions etc. To save the day, you created subclasses for mushroom and onions. But shortly after, a competitor opens a new restaurant nearby with subclasses for corn, olives, etc. Now considering new competitors, a number of subclasses you will need to create for an effective billing system can go overboard. Which design pattern do you think will be fit most to solve this problem and why? Please briefly explain.

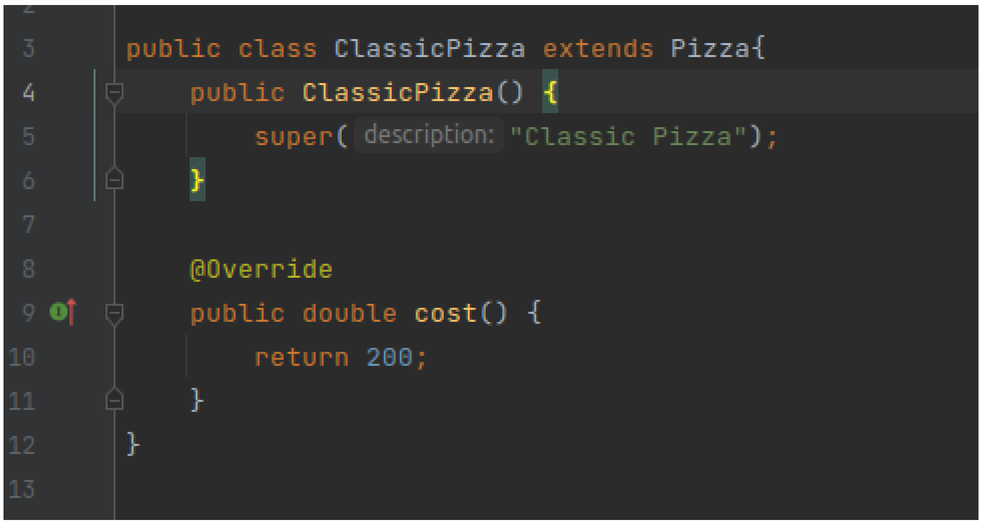
   * We can use The Decorator Design Pattern here.

   * Decorator design pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

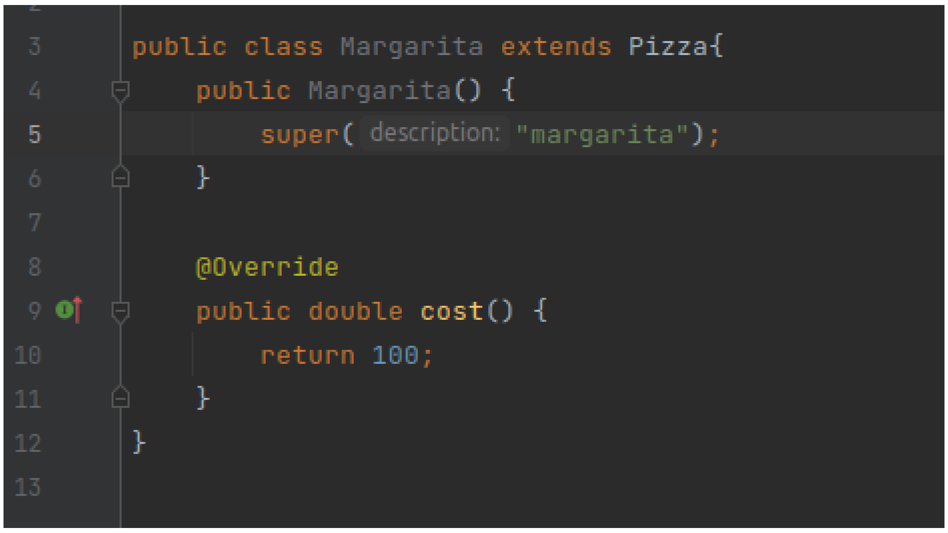
   The code is below:



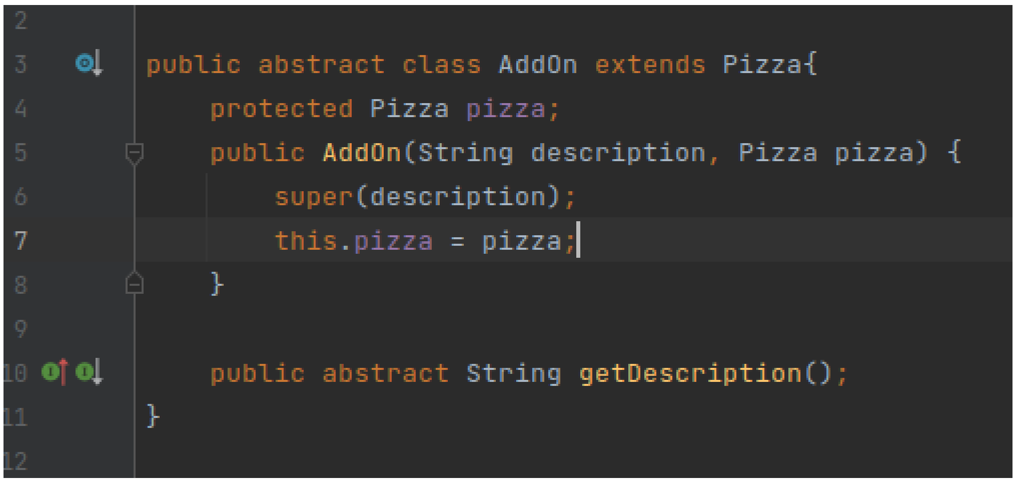
```java
public abstract class Pizza {
    private String description;

    public Pizza(String description) {
        this.description = description;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public abstract double cost();
}
```



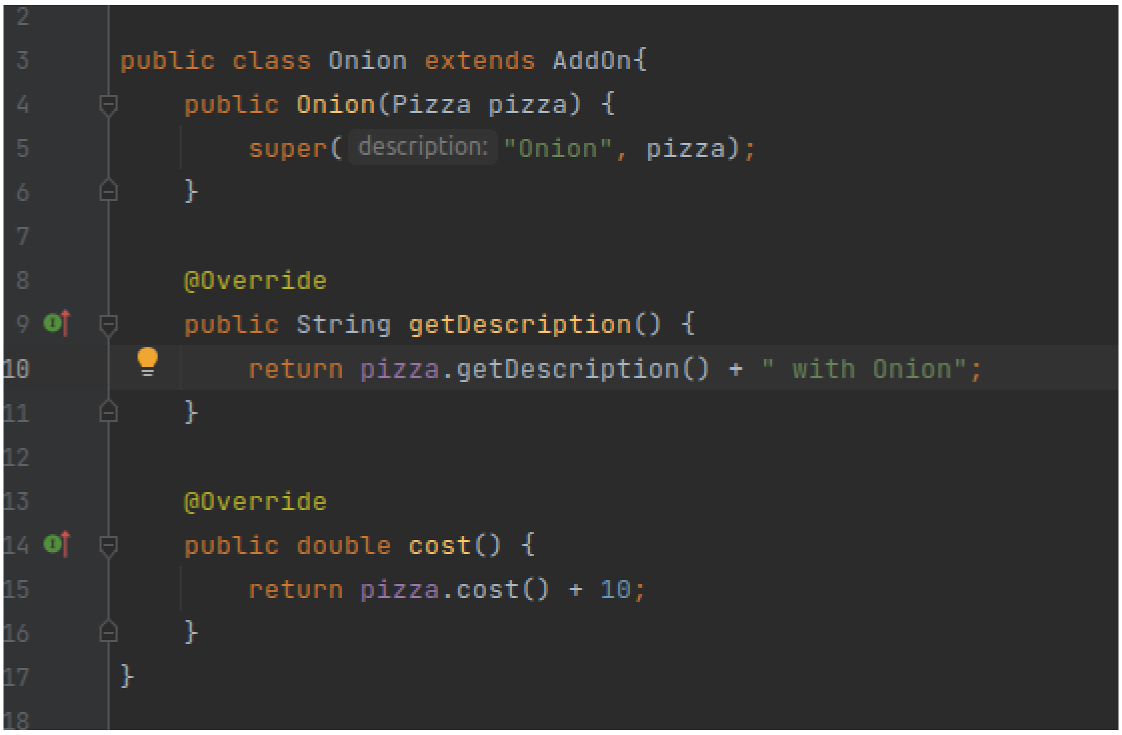
```java
public class ClassicPizza extends Pizza{
    public ClassicPizza() {
        super( description: "Classic Pizza");
    }

    @Override
    public double cost() {
        return 200;
    }
}
```



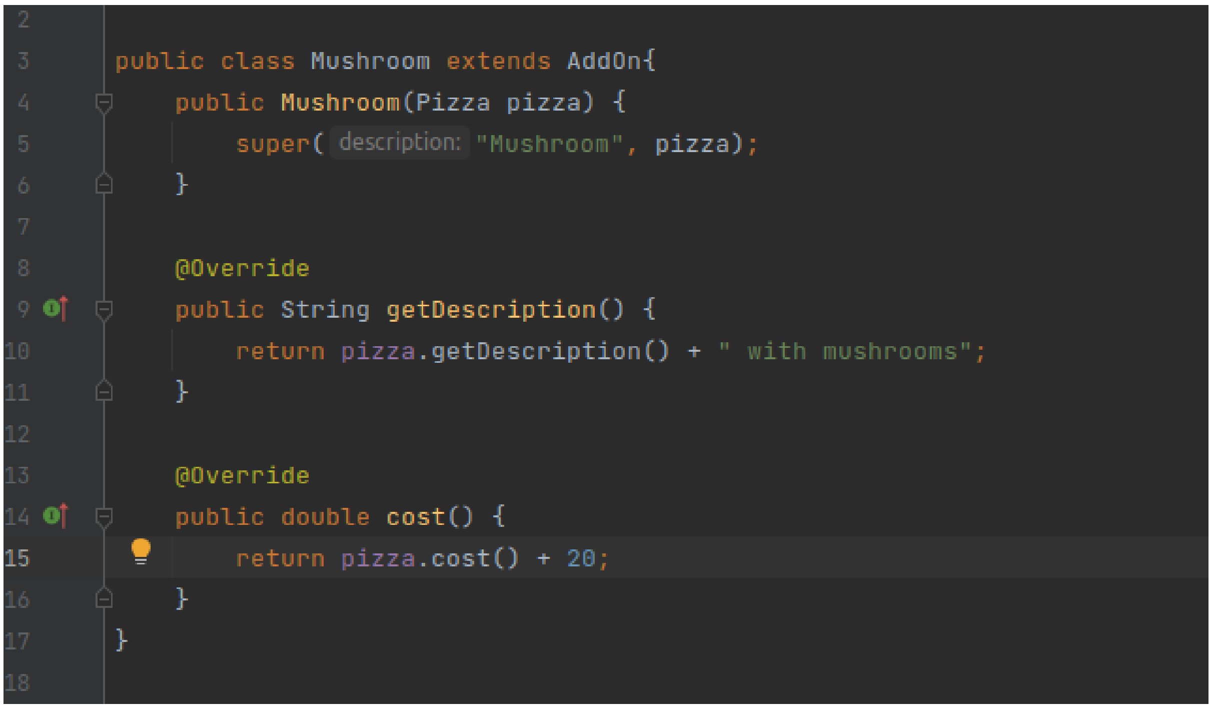
```java
public class Margarita extends Pizza{
    public Margarita() {
        super( description: "margarita");
    }

    @Override
    public double cost() {
        return 100;
    }
}
```



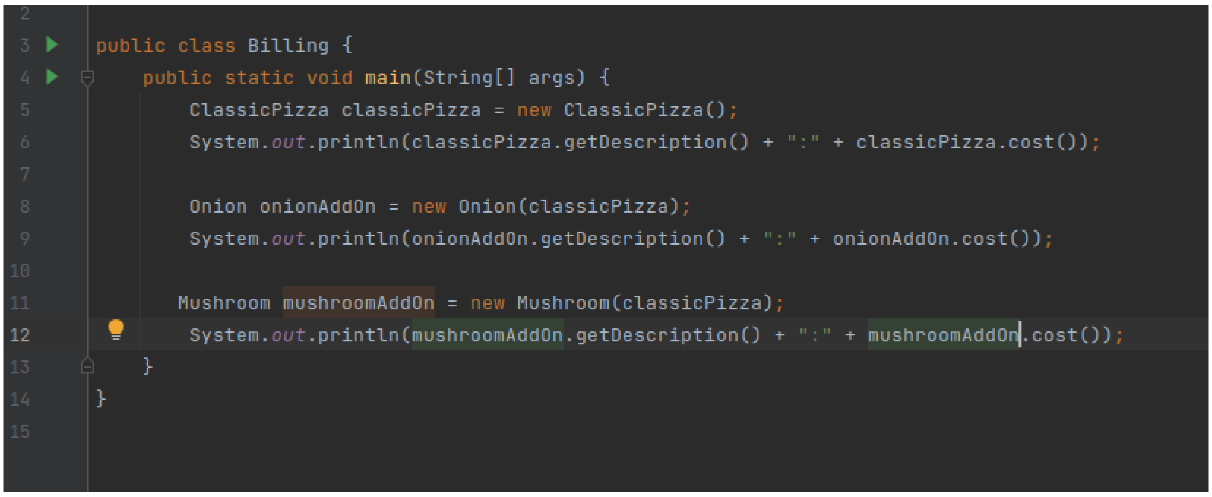
```java

public abstract class AddOn extends Pizza{
    protected Pizza pizza;
    public AddOn(String description, Pizza pizza) {
        super(description);
        this.pizza = pizza;
    }

    public abstract String getDescription();
}
```



```java

public class Onion extends AddOn{
    public Onion(Pizza pizza) {
        super( description: "Onion", pizza);
    }

    @Override
    public String getDescription() {
        return pizza.getDescription() + " with Onion";
    }

    @Override
    public double cost() {
        return pizza.cost() + 10;
    }
}

```

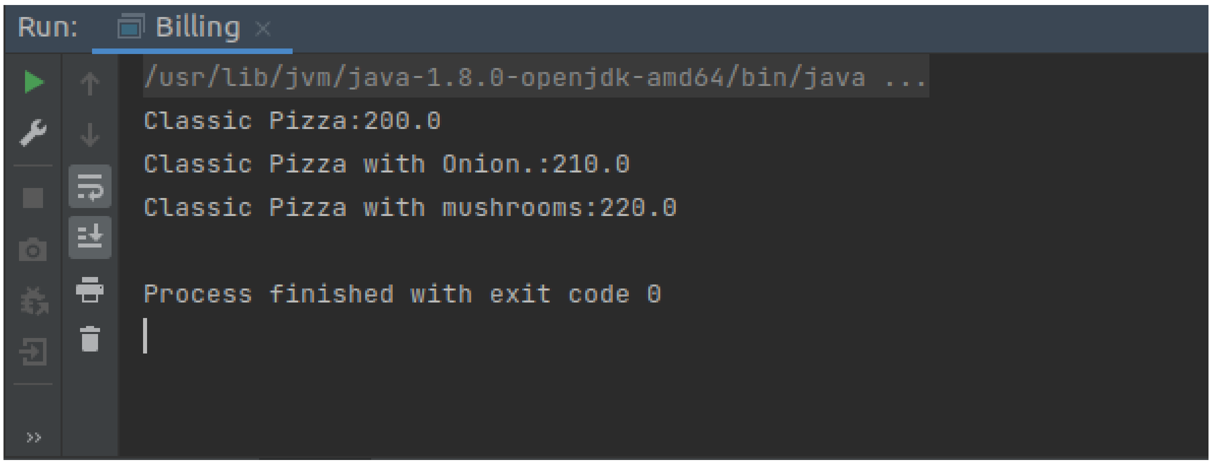


```java

public class Mushroom extends AddOn{
    public Mushroom(Pizza pizza) {
        super( description: "Mushroom", pizza);
    }

    @Override
    public String getDescription() {
        return pizza.getDescription() + " with mushrooms";
    }

    @Override
    public double cost() {
        return pizza.cost() + 20;
    }
}

```



```java
public class Billing {
    public static void main(String[] args) {
        ClassicPizza classicPizza = new ClassicPizza();
        System.out.println(classicPizza.getDescription() + ":" + classicPizza.cost());

        Onion onionAddOn = new Onion(classicPizza);
        System.out.println(onionAddOn.getDescription() + ":" + onionAddOn.cost());

       Mushroom mushroomAddOn = new Mushroom(classicPizza);
        System.out.println(mushroomAddOn.getDescription() + ":" + mushroomAddOn.cost());
    }
}
```

Output:



```
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java ...
Classic Pizza:200.0
Classic Pizza with Onion.:210.0
Classic Pizza with mushrooms:220.0

Process finished with exit code 0
```