Factory Design Pattern

a) When there is superclass and multiple subclasses and we want to get the object of subclass based on the input and requirement
b) Then we create the factory class which takes the responsibility of creating object of class based on the input

Steps to implement the factory Design pattern

1. Let us suppose we have an interface as Employee which has abstract method as

```
public interface Employee {

int salary();


}
```

2. Let us suppose we have 2 subclasses as AdroidDeveloper and WebDeveloper which implements the Employee interface

```
public class AndroidDeveloper implements Employee {
@Override
public int salary() {
System.out.println("Getting Android Developer Salary");
return 40000;
}
}
```

```
public class WebDeveloper implements Employee {
@Override
public int salary() {
System.out.println("Getting WebDeveloper Salary");
return 50000;
}
}
```

3. Now we have main class if we will create the object in main class then we need to create the object manually and also it will become the tightly coupled
4. To avoid this, we use Factory Design pattern

5. **We will create a new class EmployeeFactory which will store/create all the objects related information**

```
public class EmployeeFactory {

public static Employee getEmployee(String emp) {
if(emp.trim().equalsIgnoreCase("Android Developer")) {
return new AndroidDeveloper();
} else if(emp.trim().equalsIgnoreCase("Web Developer")) {
return new WebDeveloper();
} else {
return null;
}

}
}
```

Main class

```
public class ExecuteClass {

public static void main(String[] args) {

Employee emp1 = EmployeeFactory.getEmployee("Android Developer");
System.out.println(emp1.salary());

Employee emp2 = EmployeeFactory.getEmployee("Web Developer");
System.out.println(emp2.salary());

Employee emp3 = EmployeeFactory.getEmployee("Test");
System.out.println(emp3.salary());
}
}
```

Advantages of Factory Design pattern

a) **Focus on creating object for interface rather than implementation**
b) **Loosely coupled**
c) **More robust code**