What's a design pattern?

A design patterns are well-proved solution for solving the specific problem/task.

Classification of patterns

1. **Creational patterns**

   Creational patterns provide object creation mechanisms that increase flexibility and reuse of existing code. Below are following creational design patterns:

   1. **Factory Method**
   2. **Abstract factory**
   3. **Builder**
   4. **Prototype**
   5. **Singleton**

2. **Structural patterns**

   Structural patterns explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.

   1. **Adapter**
   2. **Bridge**
   3. **Composite**
   4. **Decorator**
   5. **Façade**
   6. **Flyweight**
   7. **Proxy**

3. **Behavioral patterns**

   Behavioral patterns take care of effective communication and the assignment of responsibilities between objects.

   1. **Chain of Responsibility**
   2. **Command**
   3. **Iterator**
   4. **Mediator**
   5. **Memento**
   6. **Observer**
   7. **State**
   8. **Strategy**
   9. **Template Method**
   10. **Visitor**

Factory Method

```
interface Currency {
	String getSymbol();
}
// Concrete Rupee Class code
class Rupee implements Currency {
	@Override
	public String getSymbol() {
		return "Rs";
	}
}

// Concrete SGD class Code
class SGDDollar implements Currency {
```

```
	@Override
	public String getSymbol() {
		return "SGD";
	}
}

// Concrete US Dollar code
class USDollar implements Currency {
	@Override
	public String getSymbol() {
		return "USD";
	}
}
// Factroy Class code
class CurrencyFactory {

	public static Currency createCurrency (String country) {
	if (country. equalsIgnoreCase ("India")){
		return new Rupee();
	}else if(country. equalsIgnoreCase ("Singapore")){
		return new SGDDollar();
	}else if(country. equalsIgnoreCase ("US")){
		return new USDollar();
	 }
	throw new IllegalArgumentException("No such currency");
	}
}

public class Factory {
	public static void main(String args[]) {
		String country = args[0];
		Currency rupee = CurrencyFactory.createCurrency(country);
		System.out.println(rupee.getSymbol());
	}
}
```

Read more: https://javarevisited.blogspot.com/2011/12/factory-design-pattern-java-example.html#ixzz5wAjLmlDG

Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. In another word A Factory Pattern or Factory Method Pattern says that just define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate. i.e subclasses are responsible to create the instance of the class. It provides loose coupling and high cohesion.

Real Time Example:

Here I am going to take a real-life example to demonstrate it very clearly. Let suppose there is company which manufactures two types of car, First one is “Suzuki” and other one is “Honda”.

I need to know the details of each car based on their name. I will only pass the name and it will give us all the required details of the particular car, so it can be achieved to create two simple classes.

Real world examples of Factory Method pattern

A class implementing factory design pattern works as bridge between multiple classes. Consider an example of using multiple database servers like SQL Server and Oracle. If you are

developing an application using SQL Server database as backend, but in future need to change backend database to oracle, you will need to modify all your code, if you haven't written your code following factory design pattern.
In factory design pattern you need to do very little work to achieve this. A class implementing factory design pattern takes care for you and lessen your burden. Switching from database server won't bother you at all. You just need to make some small changes in your configuration file.

```
interface DbTable
{
    public function create(String[] DbInfo);
}

class MySqlTable implements DbTable
{
    public function create(String[] DbInfo)
    {
        // add a record to a table on mysql database
    }
}

class OracleTable implements DbTable
{
    public function create(String[] DbInfo)
    {
        // add a record to a table on oracle database
    }
}
// this code in php convert it into my java
class TableFactory
{
    private $dbTypeConfig = 'mysql';

    public function createTable()
    {
        if ($this->dbTypeConfig == 'mysql') {
            return new MySqlTable();
        } elseif ($this->dbTypeConfig == 'oracle') {
            return new OracleTable();
        }
        return null;
    }
}
```

Abstract Factory

Abstract Factory design pattern adds another layer of abstraction over Factory Pattern and Abstract Factory implementation itself e.g. Abstract Factory will allow us to choose a particular Factory implementation based upon need which will then produce different kinds of products.
In short
1) Abstract Factory design pattern creates Factory
2) Factory design pattern creates ProductsRead
more: https://javarevisited.blogspot.com/2013/01/difference-between-factory-and-abstract-factory-design-pattern-java.html#ixzz5wosr6DVG

[What design patterns are used in Spring framework? [closed]]

Proxy (AOP and remoting), Factory (For creating bean by Beanfactory container and application context container), Singleton (Beans defined in spring config files are singletons by default.), Model View Controller (MVC), Template method - used extensively to deal with boilerplate repeated code (such as closing connections cleanly, etc..). For example **JdbcTemplate**, **JmsTemplate**, **JpaTemplate**. Note: Any place where you need a run-time value to construct a particular dependency, **Abstract Factory** is the solution.

Example of Factory design pattern:

```
interface Bank{
       String getBankName();
}
```

```
class HDFC implements Bank{
        private final String BNAME;
        public HDFC(){
              BNAME="HDFC BANK";
       }
       public String getBankName() {
                return BNAME;
       }
}
```

```
class ICICI implements Bank{
      private final String BNAME;
      ICICI(){
              BNAME="ICICI BANK";
       }
       public String getBankName() {
                return BNAME;
      }
}
```

```
class SBI implements Bank{
     private final String BNAME;
     public SBI(){
              BNAME="SBI BANK";
       }
      public String getBankName(){
                return BNAME;
      }
}
```

```
abstract class Loan{
   protected double rate;
   abstract void getInterestRate(double rate);
```

```
  public void calculateLoanPayment(double loanamount, int years)
  {
      /*
           to calculate the monthly loan payment i.e. EMI

           rate=annual interest rate/12*100;
           n=number of monthly installments;
           1year=12 months.
           so, n=years*12;

         */

        double EMI;
        int n;

        n=years*12;
        rate=rate/1200;
        EMI=((rate*Math.pow((1+rate),n))/((Math.pow((1+rate),n))-1))*loanamount;

System.out.println("your monthly EMI is "+ EMI +" for the amount"+loanamount+" you have borrowed");
 }
}// end of the Loan abstract class.

class HomeLoan extends Loan{
    public void getInterestRate(double r){
        rate=r;
   }
}//End of the HomeLoan class.
```

```
class BussinessLoan extends Loan{
   public void getInterestRate(double r){
        rate=r;
    }

}//End of the BusssinessLoan class.
```

```
class EducationLoan extends Loan{
    public void getInterestRate(double r){
      rate=r;
 }
}//End of the EducationLoan class.
```

```
abstract class AbstractFactory{
```

```
  public abstract Bank getBank(String bank);
  public abstract Loan getLoan(String loan);
}
```

```
class BankFactory extends AbstractFactory{
   public Bank getBank(String bank){
      if(bank == null){
         return null;
      }
      if(bank.equalsIgnoreCase("HDFC")){
         return new HDFC();
      } else if(bank.equalsIgnoreCase("ICICI")){
         return new ICICI();
      } else if(bank.equalsIgnoreCase("SBI")){
         return new SBI();
      }
      return null;
   }
  public Loan getLoan(String loan) {
      return null;
   }
}//End of the BankFactory class.
```

```
class LoanFactory extends AbstractFactory{
          public Bank getBank(String bank){
               return null;
          }
       public Loan getLoan(String loan){
      if(loan == null){
         return null;
      }
      if(loan.equalsIgnoreCase("Home")){
         return new HomeLoan();
      } else if(loan.equalsIgnoreCase("Business")){
         return new BussinessLoan();
      } else if(loan.equalsIgnoreCase("Education")){
         return new EducationLoan();
      }
      return null;
   } }
class FactoryCreator {
     public static AbstractFactory getFactory(String choice){
      if(choice.equalsIgnoreCase("Bank")){
```

```
      return new BankFactory();
    } else if(choice.equalsIgnoreCase("Loan")){
      return new LoanFactory();
    }
    return null;
  }
}//End of the FactoryCreator.
```

```
import java.io.*;
class AbstractFactoryPatternExample {
    public static void main(String args[])throws IOException {

    BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

    System.out.print("Enter the name of Bank from where you want to take loan amount: ");
    String bankName=br.readLine();

System.out.print("\n");
System.out.print("Enter the type of loan e.g. home loan or business loan or education loan : ");

String loanName=br.readLine();
AbstractFactory bankFactory = FactoryCreator.getFactory("Bank");
Bank b=bankFactory.getBank(bankName);

System.out.print("\n");
System.out.print("Enter the interest rate for "+b.getBankName()+ ": ");

double rate=Double.parseDouble(br.readLine());
System.out.print("\n");
System.out.print("Enter the loan amount you want to take: ");

double loanAmount=Double.parseDouble(br.readLine());
System.out.print("\n");
System.out.print("Enter the number of years to pay your entire loan amount: ");
int years=Integer.parseInt(br.readLine());

System.out.print("\n");
System.out.println("you are taking the loan from "+ b.getBankName());

AbstractFactory loanFactory = FactoryCreator.getFactory("Loan");
        Loan l=loanFactory.getLoan(loanName);
        l.getInterestRate(rate);
```

```
        l.calculateLoanPayment(loanAmount,years);
  }
}//End of the  AbstractFactoryPatternExample
```

Builder design pattern

Builder Pattern says that "construct a complex object from simple objects using step-by-step approach". It is mostly used when object can't be created in single step like in the de-serialization of a complex object.
Constructors in Java are used to create object and can take parameters required to create object. Problem starts when an Object can be created with lot of parameters, some of them may be mandatory and others may be optional.
Read more: https://javarevisited.blogspot.com/2012/06/builder-design-pattern-in-java-example.html#ixzz5wNaKDpw2
If we are going to have **overloaded constructor** for different kind of cake then there will be many constructors and even worst they will accept many parameters.
Builder design pattern not only improves readability but also reduces chance of error by adding ingredients explicitly and making object available once fully constructed.

When to use Builder Design pattern in Java

Builder Design pattern is a creational pattern and should be used when number of parameter required in constructor is more than manageable usually 4 or at most 5.
Don't confuse with Builder and Factory pattern there is an obvious difference between Builder and Factory pattern, **as Factory can be used to create different implementation of same interface but Builder is tied up with its Container class and only returns object of Outer class.**
Builder pattern is probably one of the easiest (except singleton) to implement and to use it. It's very convenient to use, when an entity has a lot of constructors and becomes very hard for external user of entity to decide which one to use.
Builder pattern helps to avoid such confusion, because the user can decide for which fields set values and for which not.

Example of Builder Design pattern in Java

```
public class BuilderPatternExample {

   public static void main(String args[]) {

      //Creating object using Builder pattern in java

      Cake whiteCake =
newCake.Builder().sugar(1).butter(0.5).eggs(2).vanila(2).flour(1.5).bakingpowder(0.75).milk(0.5).build();

//Cake is ready to eat :)

System.out.println(whiteCake);

   }

}
```

```
class Cake {

   private final double sugar;   //cup
   private final double butter;  //cup
   private final int eggs;
   private final int vanila;     //spoon
   private final double flour;   //cup
   private final double bakingpowder; //spoon
   private final double milk;  //cup
   private final int cherry;

   public static class Builder {

      private double sugar;   //cup
      private double butter;  //cup
      private int eggs;
      private int vanila;     //spoon
      private double flour;   //cup
      private double bakingpowder; //spoon
      private double milk;  //cup
      private int cherry;

      //builder methods for setting property
      public Builder sugar(double cup){this.sugar = cup; return this; }
      public Builder butter(double cup){this.butter = cup; return this; }
      public Builder eggs(int number){this.eggs = number; return this; }
      public Builder vanila(int spoon){this.vanila = spoon; return this; }
      public Builder flour(double cup){this.flour = cup; return this; }
      public Builder bakingpowder(double spoon){this.sugar = spoon; return this; }
      public Builder milk(double cup){this.milk = cup; return this; }
      public Builder cherry(int number){this.cherry = number; return this; }


      //return fully build object
      public Cake build() {
         return new Cake(this);
      }
   }

   //private constructor to enforce object creation through builder
   private Cake(Builder builder) {
      this.sugar = builder.sugar;
      this.butter = builder.butter;
      this.eggs = builder.eggs;
      this.vanila = builder.vanila;
      this.flour = builder.flour;
      this.bakingpowder = builder.bakingpowder;
      this.milk = builder.milk;
      this.cherry = builder.cherry;
   }
```

```
    @Override
    public String toString() {
        return "Cake{" + "sugar=" + sugar + ", butter=" + butter + ", eggs=" + eggs + ", vanila=" + vanila + ", flour=" + flour + ", bakingpowder=" + bakingpowder + ", milk=" + milk + ", cherry=" + cherry + '}';

    }

}
```

Output:
Cake{sugar=0.75, butter=0.5, eggs=2, vanila=2, flour=1.5, bakingpowder=0.0, milk=0.5, cherry=0}

Another example:
I like to use the Builder pattern when I have an entity which consists of a lot of fields and I need to test some service, but before that I need to set some information to an entity. Let's say we have a Person entity:

```
public class Person {
        private String firstName;
        private String lastName;
        private LocalDate birthDate;
        private String addressOne;
        private String addressTwo;
        private String sex;
        private boolean driverLicence;
        private boolean married;
// Getters/Setters
```

And I have a service, which needs to be tested, but before that, I need to set some information to an entity. I could create an entity with particular information in my test as follows:

```
private Person createPersonForTesting() {
        Person person = new Person();
        person.setFirstName("FirstName");
        person.setLastName("LastName");
        person.setAddressOne("Address1");
        ...
        return person;
    }
```

But at this point it doesn't look good, it takes a little bit more space and it takes more time to write this method (and speed to us developers is very important, isn't it?). Also, there is a risk to miss some field. To avoid all these mentioned problems, better is to use a Builder design pattern:

```
private Person createPersonForTesting() {
        return Person.builder()
                .firstName("FirstName")
                .lastName("LastName")
                .addressOne("AddressOne")
                .addressTwo("AddressTwo")
                .birthDate(LocalDate.of(1995, Month.APRIL, 13))
                .sex("male")
                .driverLicence(true)
                .married(true)
                .build();
    }
```

It looks much nicer, isn't it?
And This is Builder pattern:

```
public class Person {
        private String firstName;
        private String lastName;
        private LocalDate birthDate;
        private String addressOne;
        private String addressTwo;
        private String sex;
        private boolean driverLicence;
        private boolean married;

        public String getFirstName() {
            return firstName;
        }
        public void setFirstName(String firstName) {
            this.firstName = firstName;
        }
        public String getLastName() {
            return lastName;
        }
        public void setLastName(String lastName) {
            this.lastName = lastName;
        }
        public LocalDate getBirthDate() {
            return birthDate;
        }
        public void setBirthDate(LocalDate birthDate) {
            this.birthDate = birthDate;
        }
        public String getAddressOne() {
            return addressOne;
        }
        public void setAddressOne(String addressOne) {
            this.addressOne = addressOne;
        }
        public String getAddressTwo() {
            return addressTwo;
        }
        public void setAddressTwo(String addressTwo) {
            this.addressTwo = addressTwo;
        }
        public String getSex() {
            return sex;
        }
        public void setSex(String sex) {
            this.sex = sex;
        }
        public boolean isDriverLicence() {
            return driverLicence;
        }
        public void setDriverLicence(boolean driverLicence) {
            this.driverLicence = driverLicence;
        }
        public boolean isMarried() {
            return married;
        }
        public void setMarried(boolean married) {
            this.married = married;
        }
```

```
    public static PersonBuilder builder() {
        return new PersonBuilder();
    }
    public static class PersonBuilder {
        private String firstName;
        private String lastName;
        private LocalDate birthDate;
        private String addressOne;
        private String addressTwo;
        private String sex;
        private boolean driverLicence;
        private boolean married;
        public PersonBuilder firstName(String firstName) {
            this.firstName = firstName;
            return this;
        }
        public PersonBuilder lastName(String lastName) {
            this.lastName = lastName;
            return this;
        }
        public PersonBuilder birthDate(LocalDate birthDate) {
            this.birthDate = birthDate;
            return this;
        }
        public PersonBuilder addressOne(String addressOne) {
            this.addressOne = addressOne;
            return this;
        }
        public PersonBuilder addressTwo(String addressTwo) {
            this.addressTwo = addressTwo;
            return this;
        }
        public PersonBuilder sex(String sex) {
            this.sex = sex;
            return this;
        }
        public PersonBuilder driverLicence(boolean driverLicence) {
            this.driverLicence = driverLicence;
            return this;
        }
        public PersonBuilder married(boolean married) {
            this.married = married;
            return this;
        }
        public Person build() {
            Person person = new Person();
            person.firstName = this.firstName;
            person.lastName = this.lastName;
            person.addressOne = this.addressOne;
            person.addressTwo = this.addressTwo;
            person.birthDate = this.birthDate;
            person.sex = this.sex;
            person.driverLicence = this.driverLicence;
            person.married = this.married;
            return person;
        }
    }
}
```

Note we can use private constructor as well like previous example.

Prototype Design pattern

Prototype Pattern says that cloning of an existing object instead of creating new one and can also be customized as per the requirement.
This pattern should be followed, if the cost of creating a new object is expensive and resource intensive.
This is same as java cloning and real time example is ChargeVO in TEMS project.
Another example of prototype is chess game for every new game we will not create new object. We will just make the prototype of object and return the new copy of game.

Singleton Design pattern

Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.
Example: Database connection class should be **Singleton. i.e a single database object shared by different parts of the program.**
Note : Singleton design pattern vs Singleton beans in Spring container
Singleton scope in spring means single instance in a Spring context.
Spring container merely returns the same instance again and again for subsequent calls to get the bean.
And spring doesn't bother if the class of the bean is coded as singleton or not , in fact if the class is coded as singleton whose constructor as private, Spring use BeanUtils.instantiateClass (javadoc here) to set the constructor to accessible and invoke it.
Alternatively, we can use a factory-method attribute in bean definition like this

```
<bean id="exampleBean" class="example.Singleton"  factory-method="getInstance"/>
```

Typically singletons are used for global configuration. The simplest example would be LogManager - there's a static LogManager.getLogManager() method, and a single global instance is used.

Adapter Design pattern

Adapter design pattern is a structural design pattern also known as the Wrapper pattern that allows objects with incompatible interfaces to collaborate and which helps to bridge the gap between two classes in Java.
Imagine that you're creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user.
At some point, you decide to improve the app by integrating a smart 3rd-party analytics library. But there's a catch: the analytics library only works with data in JSON format.
An Adapter Pattern says that just "**converts the interface of a class into another interface that a client wants".**
Eg: If you have got an onsite opportunity and had visited US, UK or any other European or North American country, you might have seen different kinds of electrical socket than in India e.g. USA has a rectangular socket, as compared to cylindrical one from India. Basically, when we talk Electrical adapter, we talked about adapter which changes the voltage or the one's which allows using a plug designed for one country in a Socket of another country.
You can neither change the Socket of visiting country, neither can change the plug of your laptop, So you introduce an Adapter, which makes things working without changing any party. Similarly, **Adapter design pattern makes incompatible interfaces work together, without changing them.** Only new code which is inserted is in the form of Adapter or Wrapper class.
There are two ways to implement Adapter design pattern in Java, one using Inheritance also known as Class Adapter pattern and other is using Composition, better known as Object Adapter pattern.
Adapter just converts one interface to another, without adding additional functionalities, while Decorator adds new functionality into an interface.
This design pattern is used to provide interface between two or more incompatible objects.