DESIGN PATTERNS

Design Patterns are very popular among software developers. A design pattern is a well described solution to a common software problem.

Benefits of using design patterns are:

1. Design Patterns are already defined and provides **industry standard approach** to solve a recurring problem, so it saves time if we sensibly use the design pattern. There are many java design patterns that we can use in our java based projects.
2. Using design patterns promotes **reusability** that leads to more **robust** and highly maintainable code. It helps in reducing total cost of ownership (TCO) of the software product.
3. Since design patterns are already defined, it makes our code easy to understand and debug. It leads to faster development and new members of team understand it easily.

Java Design Patterns are divided into three categories – **creational**, **structural**, and **behavioral** design patterns. This post serves as an index for all the java design patterns articles I have written so far.

I. Creational Design Patterns

1. Singleton Pattern
2. Factory Pattern
3. Abstract Factory Pattern
4. Builder Pattern
5. Prototype Pattern

II. Structural Design Patterns

1. Adapter Pattern
2. Composite Pattern
3. Proxy Pattern
4. Flyweight Pattern
5. Facade Pattern
6. Bridge Pattern
7. Decorator Pattern

III. Behavioral Design Patterns

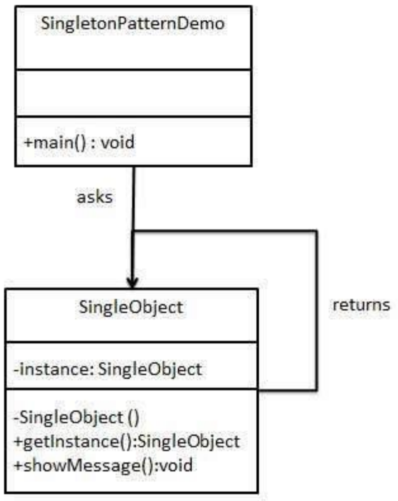
1. Template Method Pattern
2. Mediator Pattern
3. Chain of Responsibility Pattern
4. Observer Pattern
5. Strategy Pattern
6. Command Pattern
7. State Pattern
8. Visitor Pattern
9. Interpreter Pattern
10. Iterator Pattern
11. Memento Pattern

I. Creational Design Patterns

- Creational design patterns provide solution to instantiate an object in the best possible way for specific situations.
- The basic form of object creation could result in design problems or add unwanted complexity to the design.
- Creational design patterns solve this problem by **controlling the object creation** by different ways

1. Singleton Pattern:

- One Class, one Instance.
- Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the Java virtual machine.
- **The singleton class must provide a global access point to get the instance of the class**.



Uses of Singleton Pattern:

- Singleton pattern is used for logging, driver objects, caching and thread pool.
- Singleton design pattern is also used in other design patterns like **Abstract Factory**, **Builder**, **Prototype**, **Facade** etc.
- Singleton design pattern is used in core Java classes also. For example, **java.lang.Runtime** , **java.awt.Desktop**.

To implement Singleton pattern, there are many approaches but all of them have following common concepts:

- A **private constructor** to avoid instantiation of the class,
- A **private static variable** from the same class that's the only instance of the class.
- A **public static method that returns the instance of the class**, this is the global access point for the outer world to get the instance of the class.

Example:
```
public class Singleton  {
   private Singleton() {}

   private static class SingletonHolder {
         public static final Singleton instance = new Singleton();
   }

   public static Singleton getInstance() {
         return SingletonHolder.instance;
   }
}
```

Here, we've created a *static* inner class that holds the instance of the *Singleton* class. It creates the instance only when someone calls the *getInstance()* method and not when the outer class is loaded.

This is a widely used approach for a Singleton class as it **doesn't require synchronization**, is thread safe, enforces lazy initialization and has comparatively less boilerplate.

Note:

- The constructor has the *private* access modifier.
- **This is a requirement for creating a Singleton since a *public* constructor would mean anyone could access it and start creating new instances.**

To overcome above problem, we implement Thread-Safe implementation for Singleton

Example:
```
   class SingletonThreadSafeInstance {

         //private static variable
         private static SingletonThreadSafeInstance instance;

         //private constructor
         private SingletonThreadSafeInstance() {}

         //public static method that returns the instance of the class
         public static synchronized SingletonThreadSafeInstance
   getInstance(){
               if(instance == null) {
                     instance = new SingletonThreadSafeInstance();
               }

               return  instance;
         }
   }


   public class SingletonThreadSafe {
         public static void main(String[] args) {

               SingletonThreadSafeInstance singletonThread =
   SingletonThreadSafeInstance.getInstance();
```

```
                System.out.println(singletonThread);

                // Now let's instanciate another class
                SingletonThreadSafeInstance singletonThread1 =
SingletonThreadSafeInstance.getInstance();

                System.out.println(singletonThread1);

//Now check out your console, the two instances have the same reference
        }
}
```

In above example, every time a new instance is created, it will have same reference.

Explanation: In object-oriented programming, a singleton class is a class that can have only one object (an instance of the class) at a time.
After first time, if we try to instantiate the Singleton class, the new variable also points to the first instance created.

So whatever modifications we do to any variable inside the class through any instance, it affects the variable of the single instance created and is visible if we access that variable through any variable of that class type defined.

Normal class vs Singleton class: Difference in normal and singleton class in terms of instantiation is that, For normal class we use constructor, whereas for singleton class we use getInstance() method In general, to avoid confusion we may also use the class name as method name while defining this method.

When to Use Singleton Design Pattern:

- For resources that are expensive to create. (like database connection objects)
- It's good practice to keep all loggers as Singletons which increases performance.
- Classes which provide access to configuration settings for the application.
- Classes that contain resources that are accessed in shared mode.

Different approaches of Singleton pattern implementation and design concerns with the implementation.

i. Eager initialization
ii. Static block initialization
iii. Lazy Initialization
iv. Thread Safe Singleton
v. Bill Pugh Singleton Implementation
vi. Using Reflection to destroy Singleton Pattern
vii. Enum Singleton
viii. Serialization and Singleton

i. Eager Initialization:

- In eager initialization, the instance of Singleton Class is created at the time of class loading.
- This is the easiest method to create a singleton class.
- **Drawback**: instance is created even though client application might not be using it.

Implementation:

```
public class EagerInitializedSingleton {

    private static final EagerInitializedSingleton INSTANCE = new
EagerInitializedSingleton();

    //private constructor to avoid client applications to use constructor
    private EagerInitializedSingleton() {}
    public static EagerInitializedSingleton getInstance() {
        return INSTANCE;
    }
}
```

When to use:

- If your singleton class is not using a lot of resources, this is the approach to use.
- But in most of the scenarios, Singleton classes are created for resources such as File System, Database connections, etc. and we should avoid the instantiation until unless client calls the getInstance() method.
- Also this method doesn't provide any options for exception handling.

ii. Static Block Initialization:

- Static block initialization implementation is similar to eager initialization, except that instance of class is created in the static block that provides option for exception handling.

Implementation:

```
public class StaticBlockSingleton {

   private static StaticBlockSingleton instance;

   private StaticBlockSingleton(){}

   //static block initialization for exception handling
   static{
         try{
               instance = new StaticBlockSingleton();
         } catch(Exception e){
               throw new RuntimeException("Exception occured in creating
singleton instance");
         }
   }

   public static StaticBlockSingleton getInstance(){
         return instance;
   }

}
```

Note:
Both eager initialization and static block initialization creates the instance even before it's being used and that is not the best practice to use.

iii.Lazy Initialization(Class-based singleton):

- Lazy initialization method to implement Singleton pattern creates the instance in the global access method.
- The most popular approach is to implement a Singleton by creating a regular class and making sure it has:
    - A private constructor.
    - A static field containing its only instance
    - A static factory method for obtaining the instance

Implementation:

```
public class LazyInitializedSingleton {

	private static LazyInitializedSingleton instance;

	private LazyInitializedSingleton() {}

	public static LazyInitializedSingleton getInstance(){
		if(instance == null){
			instance = new LazyInitializedSingleton();
		}
		return instance;
	}

}
```

Note:

- The above implementation works fine in case of single threaded environment.
- But when it comes to multithreaded systems, it can cause issues if multiple threads are inside the if-loop at the same time. It will destroy the singleton pattern and both threads will get the different instances of singleton class.

iv. Thread Safe Singleton:

- The easier way to create a thread-safe singleton class is to make the global access method synchronized, so that only one thread can execute this method at a time.

  Implementation:

```
public class ThreadSafeSingleton {

  private static ThreadSafeSingleton instance;
  private ThreadSafeSingleton() {}

  public static synchronized ThreadSafeSingleton getInstance() {
        if(instance == null) {
              instance = new ThreadSafeSingleton();
        }
        return instance;
  }


}
```

- Above implementation works fine and provides thread-safety but it reduces the performance because of cost associated with the synchronized method, although we need it only for the first few threads who might create the separate instances.
- To avoid this extra overhead every time, **double checked locking** principle is used. In this approach, the synchronized block is used inside the if-condition with an additional check to ensure that only one instance of singleton class is created.

  Implementation:

```
public static ThreadSafeSingleton getInstanceUsingDoubleLocking() {
        if(instance == null) {
              synchronized (ThreadSafeSingleton.class) {
                    if(instance == null){
                          instance = new ThreadSafeSingleton();
                    }
              }
        }
        return instance;

}
```

v. Bill Pugh Singleton Implementation:

- Prior to Java 5, java memory model had a lot of issues and above approaches used to fail in certain scenarios where too many threads try to get the instance of the Singleton class simultaneously.
- So Bill Pugh came up with a different approach to create the Singleton class using an inner static helper class.

  Implementation:

```
public class BillPughSingleton {

      private BillPughSingleton() {}

      private static class SingletonHelper {
            private static final BillPughSingleton INSTANCE = new
                                          BillPughSingleton();
      }

      public static BillPughSingleton getInstance() {
            return SingletonHelper.INSTANCE;
            }

        }
```

Note:

- The **private inner static class** that contains the instance of the singleton class.
- When the singleton class is loaded, SingletonHelper class is not loaded into memory and only when someone calls the *getInstance* method, this class gets loaded and creates the Singleton class instance.
- This is the most widely used approach for Singleton class as it doesn't require synchronization.

vi. Using Reflection To Destroy Singleton Pattern:

- Reflection can be used to destroy all the above singleton implementation approaches.

  Implementation:

```
public class ReflectionSingleton {

    public static void main(String[] args) {
        EagerInitializedSingleton instanceOne =
EagerInitializedSingleton.getInstance();
        EagerInitializedSingleton instanceTwo = null;

        try {
            Constructor[] constructors =
EagerInitializedSingleton.class.getDeclaredConstructors();
            for (Constructor constructor : constructors) {
                //Below code will destroy the singleton pattern
                constructor.setAccessible(true);
                instanceTwo = (EagerInitializedSingleton)
constructor.newInstance();
                break;
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println(instanceOne.hashCode());
        System.out.println(instanceTwo.hashCode());
    }


}
```

- When you run the above test class, you will notice that hashCode of both the instances are not same that destroys the singleton pattern. Reflection is very powerful and used in a lot of frameworks like Spring and Hibernate.

vii. Enum Singleton:

- To overcome the situation with Reflection(to destroy singleton implementation), Joshua Bloch suggests the use of Enum to implement Singleton design pattern as Java ensures that any enum value is instantiated only once in a Java program.
- Since Java Enum values are globally accessible, so is the singleton.
- **Advantage**: This approach has serialization and thread-safety guaranteed by the enum implementation itself, which ensures internally that only the single instance is available, correcting the problems pointed out in the class-based implementation.

- **Drawback**: The enum type is somewhat inflexible; for example, it does not allow lazy initialization.

Implementation:

```
public enum EnumSingleton {

    INSTANCE;

    public EnumSingleton getInstance() {
            return INSTANCE;
    }

 }
```

viii. Serialization and Singleton:

- Sometimes in distributed systems, we need to implement Serializable interface in Singleton class so that we can store its state in file system and retrieve it at later point of time.

Implementation:

```
public class SerializedSingleton implements Serializable {

    private static final long serialVersionUID = 707713214475084978L;

    private SerializedSingleton() {}

    private static class SingletonHelper {
            private static final SerializedSingleton instance = new
SerializedSingleton();
    }

    public static SerializedSingleton getInstance() {
            return SingletonHelper.instance;
    }
}
```

The problem with above serialized singleton class is that whenever we deserialize it, it will create a new instance of the class.

```
public class SingletonSerializedTest {

   public static void main(String[] args) throws
FileNotFoundException, IOException, ClassNotFoundException {

         SerializedSingleton instanceOne =
SerializedSingleton.getInstance();
         ObjectOutput out = new ObjectOutputStream(new
FileOutputStream(
                     "filename.ser"));
         out.writeObject(instanceOne);
         out.close();

         //deserailize from file to object
         ObjectInput in = new ObjectInputStream(new FileInputStream(
                     "filename.ser"));
         SerializedSingleton instanceTwo = (SerializedSingleton)
in.readObject();
         in.close();

         System.out.println("instanceOne
hashCode="+instanceOne.hashCode());
         System.out.println("instanceTwo
hashCode="+instanceTwo.hashCode());

   }

}
```

Output for above program return different hashcode. So it destroys the singleton pattern, to overcome this scenario all we need to do it provide the implementation of readResolve() method.

```
protected Object readResolve() {
            return getInstance();

}
```

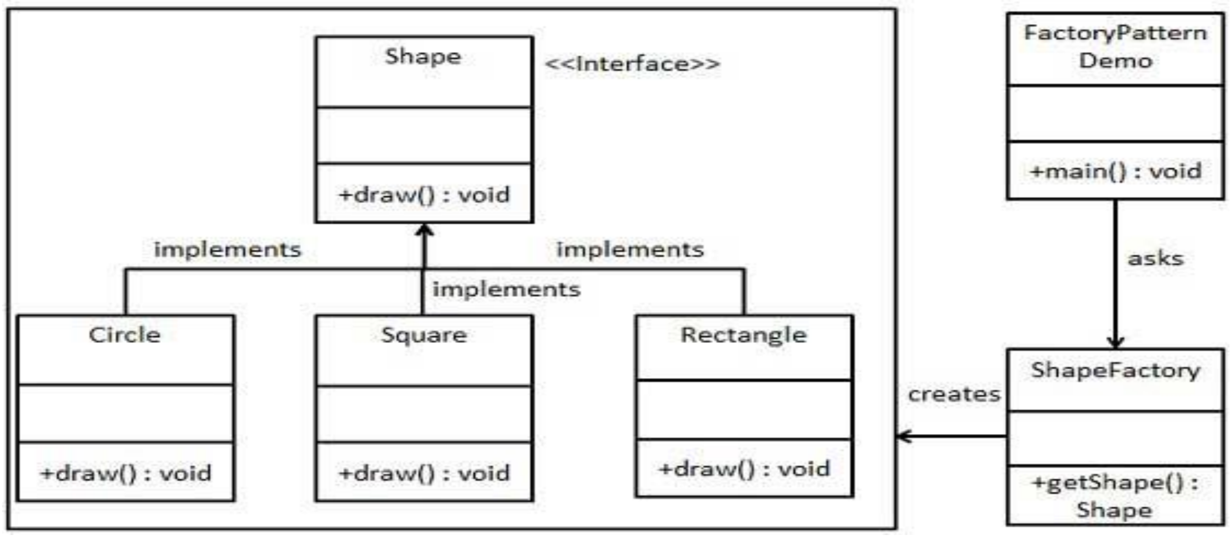
After this you will notice that hashCode of both the instances are same in test program.

2. Factory Pattern:

- Factory design pattern is used when we have a super class with multiple sub-classes and based on input, we need to return one of the sub-class.
- **Creates objects of several related classes without specifying the exact object to be created.**
- This pattern take out the responsibility of instantiation of a class from client program to the factory class.
- This pattern delegates the responsibility of initializing a class from the client to a particular factory class by creating a type of virtual constructor.
- To achieve this, we rely on a factory which provides us with the objects, hiding the actual implementation details. The created objects are accessed using a common interface.
- Super Class: Super class in factory pattern can be an interface, abstract class or a normal Java class.
- Sub-classes: Sub-classes extending or implanting the super class.

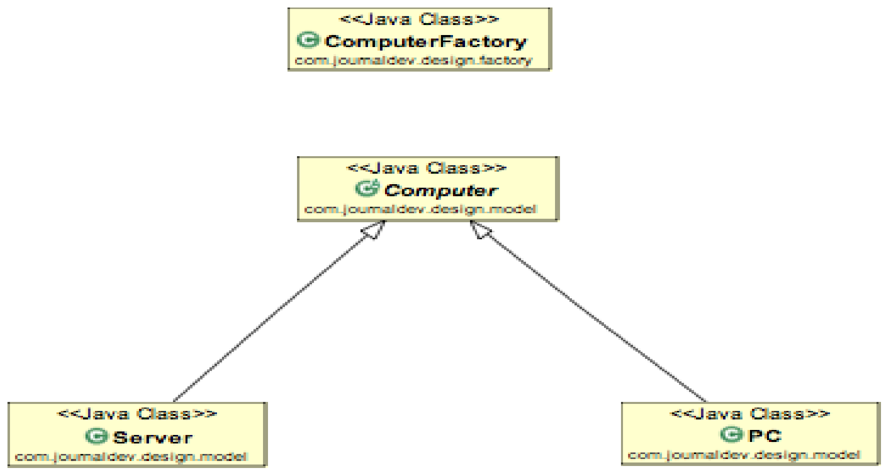
Steps to create Factory Pattern:

1. Create an interface.
2. Create concrete classes implementing the same interface.
3. Create a Factory to generate object of concrete class based on given information.
4. Use the Factory to get object of concrete class by passing an information such as type.



Example:

We have super class as abstract class with overridden toString() method for testing purpose. Let’s say we have two sub-classes PC and Server with implementation.



<<Java Class>>
ComputerFactory
com.journaldev.design.factory
<<Java Class>>
Computer
com.journaldev.design.model
<<Java Class>>
Server
com.journaldev.design.model
<<Java Class>>
PC
com.journaldev.design.model

Super Classes:

```
public abstract class Computer {

   public abstract int getRam();
   public abstract double getCpu();
   public abstract int getHdd();

   @Override

   public String toString(){

               return "Config :: RAM = " + this.getRam() + " CPU = " +
               this.getCpu() + " HDD = " + this.getHdd();
   }
}
```

Sub-Classes:

```
public class PC extends Computer {

    private int ram;
    private double cpu;
    private int hdd;


    public PC(int ram,int hdd, double cpu){
        this.ram = ram;
        this.hdd= hdd;
        this.cpu = cpu;
    }

    public int getRam(){
```

```java
        return this.ram;
    }
    public double getCpu(){
        return this.cpu;
    }

    public int getHdd(){
        return this.hdd;
    }
}


public class Server extends Computer {

   private int  ram;
   private double cpu;
   private int  hdd;

   public Server(int ram,int hdd,double cpu) {
         this.ram = ram;
         this.hdd= hdd;
         this.cpu = cpu;
   }

   public int getRam() {
         return this.ram;
   }

   public double getCpu() {
         return this.cpu;
   }

   public int getHdd() {
         return this.hdd;
   }
}


public class FactoryClass {

   public static Computer getComputer(String c,int ram,int hdd,double dd)
{

         //PC and Server classes extend from Computer.
         if("PC".equalsIgnoreCase(c))
               return new PC(ram, hdd, dd);
         else if("Server".equalsIgnoreCase(c))
               return new Server(ram, hdd, dd);

         return null;
   }

   public static void main(String[] args) {

       Computer comp1 = FactoryClass.getComputer("PC", 16, 499, 4.3);
       System.out.println(comp1);
```

```
        Computer comp2 = FactoryClass.getComputer("Server", 30, 900,9);
        System.out.println(comp2);

      //Now you can see the output in your console.
    }
}

/*Output:
Config :: RAM = 16 CPU = 4.3 HDD = 499
Config :: RAM = 30 CPU = 9.0 HDD = 900*/
```

Advantages of Factory Pattern:

1. Factory design pattern provides approach to code for interface rather than implementation.
2. Factory pattern removes the instantiation of actual implementation classes from client code.
3. Factory pattern makes our code more robust, less coupled and easy to extend. For example, we can easily change PC class implementation because client program is unaware of this.
4. Factory pattern provides abstraction between implementation and client classes through inheritance.

Factory Design Pattern Examples in JDK:

1. **java.util.Calendar**, **ResourceBundle** and **NumberFormat getInstance()** method uses Factory pattern.
2. **valueOf()** method in wrapper classes like Boolean, Integer etc.

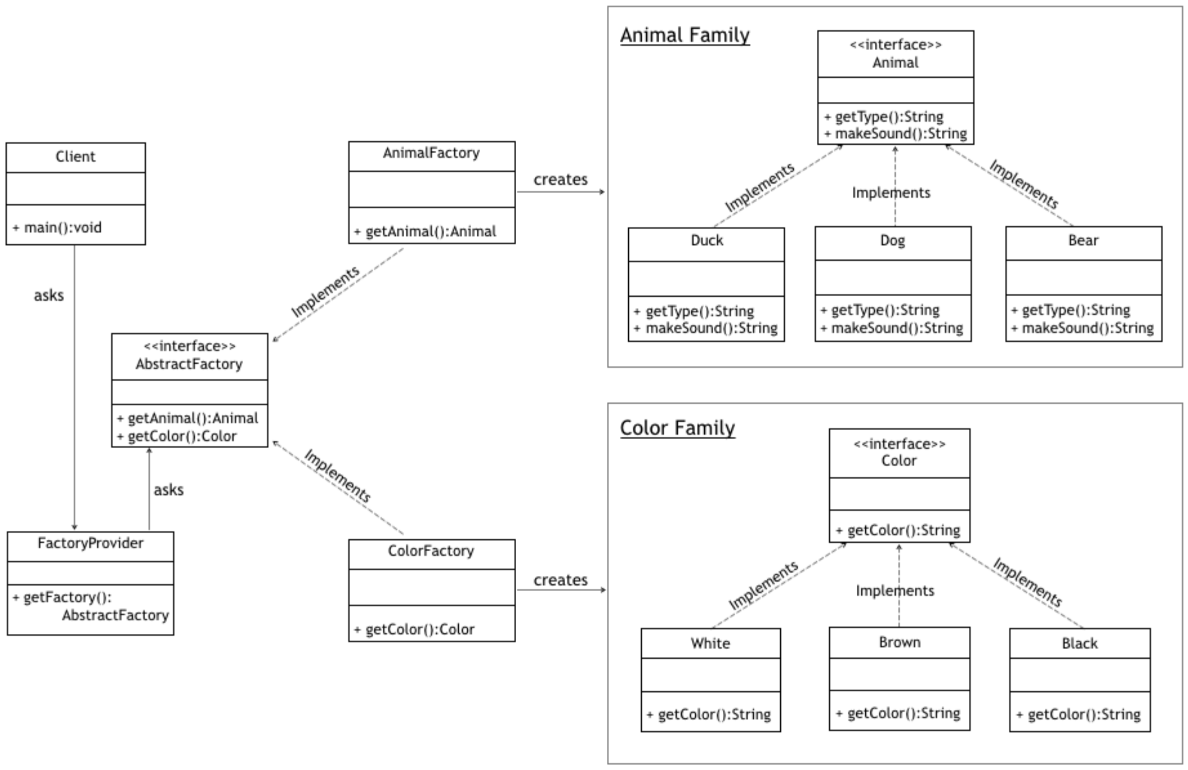
When to Use Factory Method Design Pattern:

- When the implementation of an interface or an abstract class is expected to change frequently
- When the current implementation cannot comfortably accommodate new change.
- When the initialization process is relatively simple, and the constructor only requires a handful of parameters.

3. Abstract Factory Pattern:

- It is almost similar to Factory Pattern except the fact that it's most like Factory of factories.
- By contrast, the Abstract Factory Design Pattern is used to create families of related or dependent objects. Also sometimes called a factory of factories.
- The GoF definition states that an Abstract Factory **"provides an interface for creating families of related or dependent objects without specifying their concrete classes".**

Example:



- Create a family of *Animal* class and later use it in our Abstract Factory

Interface Animal:

```
public interface Animal {

    String getAnimal();
    String makeSound();
}
```

- And a concrete implementation *Duck*:

```
public class Duck implements Animal {

    @Override
    public String getAnimal() {
          return "Duck";
    }

    @Override
    public String makeSound() {
          return "Squeeks";
    }
 }
```

The Abstract Factory deals with families of dependent objects. With that in mind, we're going to introduce one more family *Color* as an interface with a few implementations.
(Code skipped for Color interface and it's implementing classes)

- Create an *AbstractFactory* interface for Animal and Color:

```
public interface AbstractFactory {

    Animal getAnimal(String animalType) ;
    Color getColor(String colorType);
}
```

- Implement an *AnimalFactory* using the Factory Method design pattern

```
public class AnimalFactory implements AbstractFactory {

  @Override
  public Animal getAnimal(String animalType) {
        if ("Dog".equalsIgnoreCase(animalType)) {
              return new Dog();
        } else if ("Duck".equalsIgnoreCase(animalType)) {
              return new Duck();
        }

        return null;
  }

  @Override
  public Color getColor(String color) {
        throw new UnsupportedOperationException();
  }


}
```

Similarly, we can implement a factory for the *Color* interface using the same design pattern.

When all this is set, we'll create a *FactoryProvider* class that will provide us with an implementation of *AnimalFactory*or, *ColorFactory* depending on the argument supplied to the *getFactory()* method:

```
public class FactoryProvider {
  public static AbstractFactory getFactory(String choice){

        if("Animal".equalsIgnoreCase(choice)) {
              return new AnimalFactory();
        }
```

```
            else if("Color".equalsIgnoreCase(choice)) {
                  return new ColorFactory();
            }

            return null;
      }

    }
```

When to Use Abstract Factory Pattern:

- The client should be independent of how the products are created and composed in the system.
- The system consists of multiple families of products, and these families are designed to be used together.
- We need a run-time value to construct a particular dependency.

4. Builder Pattern:

- Builder pattern builds a complex object using simple objects and using a step by step approach.
- A Builder class builds the final object step by step. This builder is independent of other objects.
- This pattern was introduced to solve some of the problems with Factory and Abstract Factory patterns when the Object contains a lot of attributes.
- This pattern deals with a static nested class and then copy all the arguments from the outer class to the Builder class.
- Builder pattern solves the issue with large number of optional parameters and inconsistent state by providing a way to build the object step-by-step and provide a method that will actually return the final Object

There are three major issues with Factory and Abstract Factory design patterns when the Object contains a lot of attributes.

1. Too Many arguments to pass from client program to the Factory class that can be error prone because most of the time, the type of arguments are same and from client side it is hard to maintain the order of the argument.
2. Some of the parameters might be optional but in Factory pattern, we are forced to send all the parameters and optional parameters need to send as NULL.
3. If the object is heavy and its creation is complex, then all that complexity will be part of Factory classes that is confusing.

Solution:

- We can solve the issues with large number of parameters by providing a constructor with required parameters and then different setter methods to set the optional parameters.
- The problem with this approach is that the Object state will be **inconsistent** until unless all the attributes are set explicitly.
- Builder pattern solves the issue with large number of optional parameters and inconsistent state by providing a way to build the object step-by-step and provide a method that will actually return the final Object.

Implementation steps of builder design pattern in java:

1. First of all you need to create a static nested class and then copy all the arguments from the outer class to the Builder class.
   We should follow the naming convention (e.g. if the class name is BankAccount then builder class should be named as BankAccountBuilder)
2. Java Builder class should have a public constructor with all the required attributes as parameters
3. Java Builder class should have methods to set the optional parameters and it should return the same Builder object after setting the optional attribute.
4. The final step is to provide a method in the builder class that will return the Object needed by client program. For this we need to have a private constructor in the Class with Builder class as argument.

There are really various implementations of this pattern in JDK:

- java.lang.StringBuilder#append() (unsynchronized)
- java.lang.StringBuffer#append() (synchronized) .

Example:

BankAccount which contains a builder as a *static* inner class:

```
public class BankAccount {

      private String name;
      private String accountNumber;
      private String email;
      private boolean newsletter;

      //Note the private visibility of the constructor, and the builder
      variable type.
      private BankAccount(BankAccountBuilder builder) {
            this.name = builder.name;
            this.accountNumber = builder.accountNumber;
            this.email = builder.email;
            this.newsletter = builder.newsletter;
      }
      public static class BankAccountBuilder {
            // builder code
      }

}
```

Note:

All the access modifiers on the fields are declared ***private*** since so that no outer objects can access them directly.

The **constructor is also *private*** so that only the Builder assigned to this class can access it. All of the properties set in the constructor are extracted from the builder object which we supply as an argument

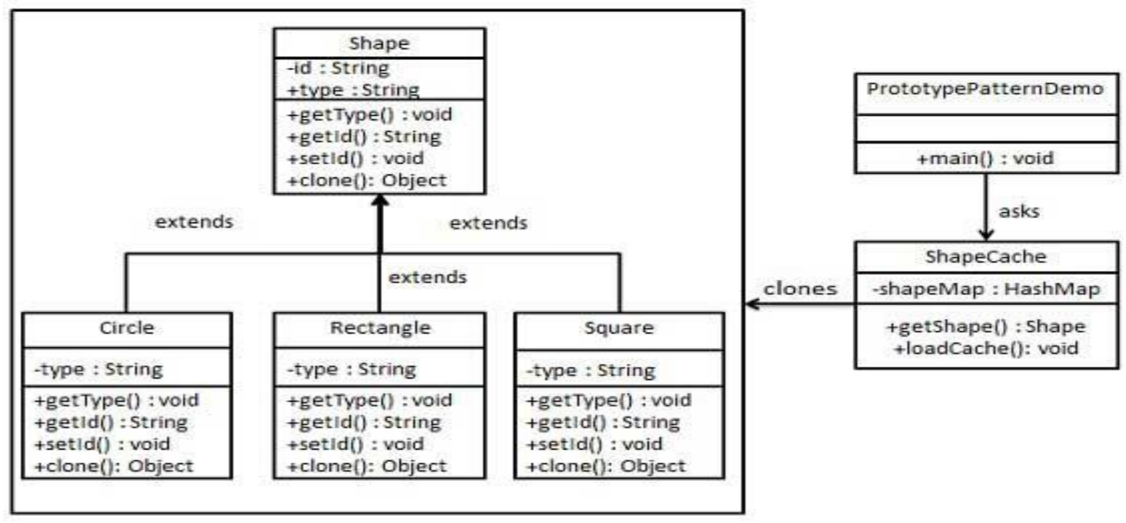
```
public class BankAccount {

	//required params.
	private String name;
	private String accountNumber;

	//optionals params
	private String email;
	private boolean newsletter;

	// Note the private visibility of the constructor, and the builder
	variable type.
	private BankAccount(BankAccountBuilder builder) {
		this.name = builder.name;
		this.accountNumber = builder.accountNumber;
		this.email = builder.email;
		this.newsletter = builder.newsletter;
	}

	//the builder inner class
	public static class BankAccountBuilder {

		//required params.
		private String name;
		private String accountNumber;

		//optionals params
		private String email;
		private boolean newsletter;

		public BankAccountBuilder(String name, String accountNumber) {
			this.name = name;
			this.accountNumber = accountNumber;
		}

		public BankAccountBuilder withEmail(String email) {
			this.email = email;
			return this;
		}
		public BankAccountBuilder wantNewsletter(boolean newsletter) {
			this.newsletter = newsletter;
			return this;
		}

		// The build function.
		public BankAccount build() {
			return new BankAccount(this);
		}
	}
	public static void main(String[] args) {
		BankAccount newAccount = new BankAccount
					.BankAccountBuilder("Jon", "22738022275")
					.withEmail("jon@example.com")
					.wantNewsletter(true)
					.build();

		System.out.println(newAccount);
	}
}
```

When to Use Builder Pattern:

1. When the process involved in creating an object is extremely complex, with lots of mandatory and optional parameters.
2. When an increase in the number of constructor parameters leads to a large list of constructors.
3. When client expects different representations for the object that's constructed.

5. Prototype Pattern:

- Prototype pattern refers to creating duplicate object while keeping performance in mind. This is one of the best ways to create an object.
- This pattern involves implementing a prototype interface which tells to create a clone of the current object.
- This pattern provides a mechanism to copy the original object to a new object and then modify it according to our needs. This pattern uses Java cloning to copy the object.
- Prototype design pattern mandates that the Object which you are copying should provide the copying feature. It should not be done by any other class. However whether to use shallow or deep copy of the Object properties depends on the requirements and it's a design decision.



Example:

Suppose we have an Object that loads data from database. Now we need to modify this data in our program multiple times, so it’s not a good idea to create the Object using new keyword and load all the data again from database.

```
public class Employees implements Cloneable {
	private List<String> empList;

	public Employees(){
		empList = new ArrayList<String>();
	}

	public Employees(List<String> list){
		this.empList=list;
	}
	public void loadData(){
		//read all employees from database and put into the list
		empList.add("Pankaj");
		empList.add("Raj");
		empList.add("David");
		empList.add("Lisa");
	}

	public List<String> getEmpList() {
		return empList;
	}

	//Notice that the clone method is overridden to provide a deep copy of
	the employees list.
	@Override
	public Object clone() throws CloneNotSupportedException{
		List<String> temp = new ArrayList<String>();
		for(String s : this.getEmpList()){
			temp.add(s);
		}
		return new Employees(temp);
	}
}

public class PrototypePattern {

	public static void main(String[] args) throws
	                                CloneNotSupportedException {

		Employees emps = new Employees();
		emps.loadData();

		//Use the clone method to get the Employee object
		Employees newEmp1 = (Employees) emps.clone();
		Employees newEmp2 = (Employees) emps.clone();

		List<String> list1 = newEmp1.getEmpList();
		list1.add("John");

		List<String> list2 = newEmp2.getEmpList();
		list2.remove("Pankaj");
```

```
			System.out.println("emps List: " + emps.getEmpList());
			System.out.println("empsNew List: " + list1);
			System.out.println("empsNew1 List: " + list2);
		}
}
```

Output:

```
emps List: [Pankaj, Raj, David, Lisa]
empsNew List: [Pankaj, Raj, David, Lisa, John]
empsNew1 List: [Raj, David, Lisa]
```

If the object cloning was not provided, we will have to make database call to fetch the employee list every time. Then do the manipulations that would have been resource and time consuming.

When to Use Prototype Pattern:

- Prototype design pattern is used when the Object creation is a costly affair and requires a lot of time and resources and you have a similar object already existing.