**Types of Design Patterns**

**Creational design patterns**

Creational design patterns are concerned with**the way of creating objects.** These design patterns are used when a decision must be made at the time of instantiation of a class (i.e. creating an object of a class).

**Singleton Design Patterns:**

Singleton Pattern says that just "define a class that has only one instance and provides a global point of access to it".

In other words, a class must ensure that only single instance should be created and all other classes can use single object.

6+5 for every object may be costly. Similarly, there can be a single configuration manager or error manager in an application that handles all problems instead of creating multiple managers.

**Important Points:**

* The singleton class must provide a global access point to get the instance of the class.
* Singleton pattern is used for [logging](https://www.journaldev.com/977/logger-in-java-logging-example), drivers objects, caching and [thread pool](https://www.journaldev.com/1069/threadpoolexecutor-java-thread-pool-example-executorservice).
* Singleton design pattern is also used in other design patterns like [Abstract Factory](https://www.journaldev.com/1418/abstract-factory-design-pattern-in-java), [Builder](https://www.journaldev.com/1425/builder-design-pattern-in-java), [Prototype](https://www.journaldev.com/1440/prototype-design-pattern-in-java), [Facade](https://www.journaldev.com/1557/facade-design-pattern-in-java) etc.
* Singleton design pattern is used in core java classes also, for example java.lang.Runtime, java.awt.Desktop.

**Advantage of Singleton design pattern**

* Saves memory because object is not created at each request. Only single instance is reused again and again.

**Usage of Singleton design pattern**

* Singleton pattern is mostly used in multi-threaded and database applications. It is used in logging, caching, thread pools, configuration settings etc.

**How to create Singleton design pattern?**

To implement Singleton pattern, we have different approaches but all of them have following common concepts.

* Private constructor to restrict instantiation of the class from other classes.
* Private static variable of the same class that is the only instance of the class.
* Public static method that returns the instance of the class, this is the global access point for outer world to get the instance of the singleton class.

There are two forms of singleton design pattern

* **Early Instantiation:** creation of instance at load time.
* **Lazy Instantiation:** creation of instance when required.

**Early Instantiation:**

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 but it has a drawback that instance is created even though client application might not be using it.

|  |
| --- |
| **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***;  }  } |

**Lazy Instantiation:**

|  |
| --- |
| **public** **class** LazyInitializedSingleton {  **private** **static** LazyInitializedSingleton *instance*;    **private** LazyInitializedSingleton(){}    **public** **static** LazyInitializedSingleton getInstance(){  **if**(*instance* == **null**){  *instance* = **new** LazyInitializedSingleton();  }  **return** *instance*;  }  } |

### **Thread Safe Singleton**

The easier way to create a thread-safe singleton class is to make the global access method [synchronized](https://www.journaldev.com/1061/thread-safety-in-java), so that only one thread can execute this method at a time. General implementation of this approach is like the below class.

|  |
| --- |
| **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 (Read: [Java Synchronization](https://www.journaldev.com/1061/thread-safety-in-java)). 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.

Below code snippet provides the double checked locking implementation.

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

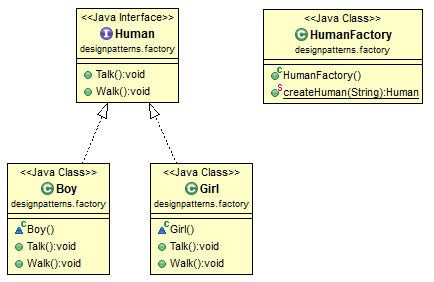
**Factory Method Pattern:**

Factory design pattern is used for creating an object based on different parameters.

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. This pattern take out the responsibility of instantiation of a class from client program to the factory class.

The example below is about creating human in a factory. If we ask the factory for a boy, the factory will produce a boy; if we ask for a girl, the factory will produce a girl. Based on different parameters, the factory produce different stuff.

**Factory pattern class diagram:**



**Example:**

|  |
| --- |
| **public** **interface** Human {  **public** **void** talk();  **public** **void** walk();  }  **public** **class** Girl **implements** Human {  @Override  **public** **void** talk() {  System.***out***.println("Girl is talking");  }  @Override  **public** **void** walk() {  System.***out***.println("Girl is walking");  }  }  **public** **class** Boy **implements** Human {  @Override  **public** **void** talk() {  System.***out***.println("boy is talking");  }  @Override  **public** **void** walk() {  System.***out***.println("boy is walking");  }  }  **class** HumanFactory {  **public** **static** Human CreateHuman(String obj) {  **if**(obj == **null**) {  **return** **null**;  }  **if**(obj.equalsIgnoreCase("boy")) {  **return** **new** Boy();  } **else** **if**(obj.equalsIgnoreCase("girl")){  **return** **new** Girl();  }  **return** **null**;  }  }  **public** **class** FactoryDesignPattern {  **public** **static** **void** main(String[] args) {  Human obj1 = HumanFactory.*CreateHuman*("boy");  obj1.talk();  obj1.walk();    Human obj2 = HumanFactory.*CreateHuman*("girl");  obj2.talk();  obj2.walk();    Human obj3 = HumanFactory.*CreateHuman*("kk");  System.***out***.println(obj3);    }  }  **Output:**  boy is talking  boy is walking  Girl is talking  Girl is walking  null |

The Factory Method Pattern is also known as Virtual Constructor.

#### Advantage of Factory Design Pattern

* Factory Method Pattern allows the sub-classes to choose the type of objects to create.
* It promotes the loose-coupling by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.

#### Usage of Factory Design Pattern

* When a class doesn't know what sub-classes will be required to create
* When a class wants that its sub-classes specify the objects to be created.
* When the parent classes choose the creation of objects to its sub-classes.

#### Factory Design Pattern Examples in JDK

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

The pattern is present in core Java libraries:

* [**java.util.Calendar#getInstance()**](http://docs.oracle.com/javase/8/docs/api/java/util/Calendar.html#getInstance--)
* [**java.util.ResourceBundle#getBundle()**](http://docs.oracle.com/javase/8/docs/api/java/util/ResourceBundle.html#getBundle-java.lang.String-)
* [**java.text.NumberFormat#getInstance()**](http://docs.oracle.com/javase/8/docs/api/java/text/NumberFormat.html#getInstance--)
* [**java.nio.charset.Charset#forName()**](http://docs.oracle.com/javase/8/docs/api/java/nio/charset/Charset.html#forName-java.lang.String-)
* [**java.net.URLStreamHandlerFactory#createURLStreamHandler(String)**](http://docs.oracle.com/javase/8/docs/api/java/net/URLStreamHandlerFactory.html)**(Returns different singleton objects, depending on a protocol)**
* [**java.util.EnumSet#of()**](https://docs.oracle.com/javase/8/docs/api/java/util/EnumSet.html#of(E))
* [**javax.xml.bind.JAXBContext#createMarshaller()**](https://docs.oracle.com/javase/8/docs/api/javax/xml/bind/JAXBContext.html#createMarshaller--)**and other similar methods.**

**Some important points about Factory Design Pattern method are;**

1. We can keep Factory class [Singleton](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples) or we can keep the method that returns the subclass as [static](https://www.journaldev.com/1365/static-keyword-in-java).
2. Notice that based on the input parameter, different subclass is created and returned. getComputer is the factory method.

**Builder Design Pattern**

Builder pattern is used to construct a complex object step by step and the final step will return the object.

Bilder pattern is a creational design pattern it means its solves problem related to object creation. 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. Consider a class which is used to create Cake, now you need number of item like egg, milk, flour to create cake. many of them are mandatory and some  of them are optional like cherry, fruits etc. If we are going to have [overloaded constructor](http://javarevisited.blogspot.sg/2012/01/what-is-constructor-overloading-in-java.html) for different kind of cake then there will be many constructor and even worst they will accept many parameter.

**When to use Builder Pattern:**

When a class has too many variables and many of them have same data type, creating object confusion for client program.

When some of the class variables are optional, so client program will have to pass NULL if factory pattern is used,

When class instance creation is heavy and complex, so all those complexity will become part of factory too.

**Problems:**

1) too many constructors to maintain.

2) error prone because many fields has same type e.g. sugar and and butter are in cups so instead of 2 cup sugar if you pass 2 cup butter, your compiler will not complain but will get a buttery cake with almost no sugar with high cost of wasting butter.

**Builder design pattern in Java – Pros and Cons:**

**Advantages:**

1. more maintainable if number of fields required to create object is more than 4 or 5.
2. less error-prone as user will know what they are passing because of explicit method call.
3. more robust as only fully constructed object will be available to client.

**Disadvantages:**

1. verbose and code duplication as Builder needs to copy all fields from Original or Item class.

### **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.

**We can implement builder design pattern in java.**

1. First of all you need to create a [static nested class](https://www.journaldev.com/996/java-inner-class) and then copy all the arguments from the outer class to the Builder class. We should follow the naming convention and if the class name is Computerthen builder class should be named as ComputerBuilder.
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 build() 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.

**Example:**

|  |
| --- |
| **package** com.mng.design.patterns;  **public** **class** Computer {  // required parameters  **private** String HDD;  **private** String RAM;  // optional parameters  **private** **boolean** isGraphicsCardEnabled;  **private** **boolean** isBluetoothEnabled;  **public** String getHDD() {  **return** HDD;  }  **public** String getRAM() {  **return** RAM;  }  **public** **boolean** isGraphicsCardEnabled() {  **return** isGraphicsCardEnabled;  }  **public** **boolean** isBluetoothEnabled() {  **return** isBluetoothEnabled;  }  **private** Computer(ComputerBuilder builder) {  **this**.HDD = builder.HDD;  **this**.RAM = builder.RAM;  **this**.isGraphicsCardEnabled = builder.isGraphicsCardEnable;  **this**.isBluetoothEnabled = builder.isBluetoothEnabled;  }  @Override  **public** String toString() {  **return** "Computer [HDD=" + HDD + ", RAM=" + RAM + ", isGraphicsCardEnabled=" + isGraphicsCardEnabled  + ", isBluetoothEnabled=" + isBluetoothEnabled + "]";  }  **public** **static** **class** ComputerBuilder {  // required parameters  **private** String HDD;  **private** String RAM;  // optional parameters  **private** **boolean** isGraphicsCardEnable;  **private** **boolean** isBluetoothEnabled;  **public** ComputerBuilder setGraphicsCardEnable(**boolean** isGraphicsCardEnable) {  **this**.isGraphicsCardEnable = isGraphicsCardEnable;  **return** **this**;  }  **public** ComputerBuilder setBluetoothEnabled(**boolean** isBluetoothEnabled) {  **this**.isBluetoothEnabled = isBluetoothEnabled;  **return** **this**;  }  **public** ComputerBuilder(String hDD, String rAM) {  **this**.HDD = hDD;  **this**.RAM = rAM;  }  **public** Computer build() {  **return** **new** Computer(**this**);  }  }  }  **public** **class** ComputerBuilderDesign {  **public** **static** **void** main(String[] args) {  Computer obj1 = **new** Computer.ComputerBuilder("segate", "ram").build();  System.***out***.println(obj1);  Computer obj2 = **new** Computer.ComputerBuilder("segate", "ram").setBluetoothEnabled(**true**).setGraphicsCardEnable(**true**).build();  System.***out***.println(obj2);  }  }  **OutPut:**  Computer [HDD=segate, RAM=ram, isGraphicsCardEnabled=false, isBluetoothEnabled=false]  Computer [HDD=segate, RAM=ram, isGraphicsCardEnabled=true, isBluetoothEnabled=true]  **Example 2:**  **public** **class** ComputerBuilder {  // required parameters  **private** String HDD;  **private** String RAM;  // optional parameters  **private** **boolean** isGraphicsCardEnable;  **private** **boolean** isBluetoothEnabled;    **public** ComputerBuilder setHDD(String hDD) {  HDD = hDD;  **return** **this**;  }  **public** ComputerBuilder setRAM(String rAM) {  RAM = rAM;  **return** **this**;  }  **public** ComputerBuilder setGraphicsCardEnable(**boolean** isGraphicsCardEnable) {  **this**.isGraphicsCardEnable = isGraphicsCardEnable;  **return** **this**;  }    **public** ComputerBuilder setBluetoothEnabled(**boolean** isBluetoothEnabled) {  **this**.isBluetoothEnabled = isBluetoothEnabled;  **return** **this**;  }    **public** Computer build() {  **return** **new** Computer(HDD, RAM, isGraphicsCardEnable, isBluetoothEnabled);    }  @Override  **public** String toString() {  **return** "ComputerBuilder [HDD=" + HDD + ", RAM=" + RAM + ", isGraphicsCardEnable=" + isGraphicsCardEnable  + ", isBluetoothEnabled=" + isBluetoothEnabled + "]";  }    }  **public** **class** Computer {  // required parameters  **private** String HDD;  **private** String RAM;  // optional parameters  **private** **boolean** isGraphicsCardEnabled;  **private** **boolean** isBluetoothEnabled;    **public** Computer(String hDD, String rAM, **boolean** isGraphicsCardEnabled, **boolean** isBluetoothEnabled) {  **super**();  HDD = hDD;  RAM = rAM;  **this**.isGraphicsCardEnabled = isGraphicsCardEnabled;  **this**.isBluetoothEnabled = isBluetoothEnabled;  }    **public** String getHDD() {  **return** HDD;  }  **public** String getRAM() {  **return** RAM;  }  **public** **boolean** isGraphicsCardEnabled() {  **return** isGraphicsCardEnabled;  }  **public** **boolean** isBluetoothEnabled() {  **return** isBluetoothEnabled;  }  }  **public** **class** ComputerBuilderDesign {  **public** **static** **void** main(String[] args) {  Computer obj1 = **new** ComputerBuilder().setHDD("segate").setRAM("transend").setBluetoothEnabled(**true**)  .setGraphicsCardEnable(**true**).build();  System.***out***.println(obj1);    Computer obj2 = **new** ComputerBuilder().setBluetoothEnabled(**true**)  .setGraphicsCardEnable(**true**).build();  System.***out***.println(obj2);  }  }  **Output:**  ComputerBuilder [HDD=segate, RAM=transend, isGraphicsCardEnable=true, isBluetoothEnabled=true]  ComputerBuilder [HDD=null, RAM=null, isGraphicsCardEnable=true, isBluetoothEnabled=true] |

**Q1) What is an immutable class?**

**Ans)** Immutable class is a class which once created, it’s contents can not be changed. Immutable objects are the objects whose state can not be changed once constructed. e.g. String class

**Q2) How to create an immutable class?**

**Ans)** To create an immutable class following steps should be followed:

1. Create a final class.
2. Set the values of properties using constructor only.
3. Make the properties of the class final and private
4. Do not provide any setters for these properties.
5. If the instance fields include references to mutable objects, don't allow those objects to be changed:
   1. Don't provide methods that modify the mutable objects.
   2. Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

public final class FinalPersonClass {

private final String name;

private final int age;

public FinalPersonClass(final String name, final int age) {

this.name = name;

this.age = age;

}

public int getAge() {

return age;

}

public String getName() {

return name;

}

}

**Q3) Immutable objects are automatically thread-safe –true/false?**

**Ans)** True. Since the state of the immutable objects can not be changed once they are created they are automatically synchronized/thread-safe.

**Q4) Which classes in java are immutable?**

**Ans)** All wrapper classes in java.lang are immutable –   
String, Integer, Boolean, Character, Byte, Short, Long, Float, Double, BigDecimal, BigInteger

**Q5) What are the advantages of immutability?**

Ans)

* Immutable objects are automatically thread-safe, the overhead caused due to use of synchronisation is avoided.
* Once created the state of the immutable object can not be changed so there is no possibility of them getting into an inconsistent state.
* The references to the immutable objects can be easily shared or cached without having to copy or clone them as there state can not be changed ever after construction.
* The best use of the immutable objects is as the keys of a map.
* Since String is immutable, it is safe for [multithreading](https://www.journaldev.com/1079/multithreading-in-java). A single String instance can be shared across different threads. This avoids the use of synchronization for thread safety. Strings are implicitly thread-safe.
* Immutable objects are thread-safe so you will not have any synchronization issues.
* Immutability makes it easier to parallelize program as there are no conflicts among objects.
* Thread safety: Immutable objects do not need synchronization and are unharmed even during concurrent access by multiple threads.

**Solid design pattern**

**Single responsibility principle** - A class should have only one reason to change.

class should only have one responsibility. Furthermore, it should only have one reason to change.

**Example:**

we have two classes Person and Account. Both have single responsibility to store their specific information. If we want to change state of Person then we do not need to modify the class Account and vice-versa.

|  |
| --- |
| public class Person  {      private Long personId;      private String firstName;      private String lastName;      private String age;      private List<Account> accounts;  }  public class Account  {      private Long guid;      private String accountNumber;      private String accountName;      private String status;      private String type;  } |

# **Open Closed Principle:**

You should not modify existing behavior but if you want, you can extend it according to your needs.” i.e. **Code is open for extension but closed for modification.**

**Vehicle** class to make it bike specific.So here comes the open closed principle in action. You can ask the mechanic to extend existing **Vehicle** class and override service method to write bike specific code in it. In this manner **Vehicle** class code is not modified and mechanic creates **Bike** class extending **Vehicle** to add bike servicing specific code.

|  |
| --- |
| **public** **class** Vehicle {  **public** **void** service() {  System.***out***.println("General procedures being done on your vehicle");  }  }  **public** **class** Bike **extends** Vehicle {  @Override  **public** **void** service() {  System.***out***.println("Now doing bike specific servicing");  }  }  **public** **class** Ocp {  **public** **static** **void** main(String[] args) {  Bike b = **new** Bike();  b.service();  }  } |

# **Liskov Substitution Principle:**

Derived classes must be substitutable(similar, compatible) for their base classes.

if you have a class, Animal, with a MakeNoise() method, then any subclass of Animal should reasonably implement MakeNoise(). Cats should meow, dogs should bark, etc. What you wouldn't do is define a MuteMouse class that throws IDontActuallyMakeNoiseException.

This is an extension of open/close principle. Derived classes should not change the behavior of the base class (behavior of inherited methods)

**Interface Segregation (**separation**):**

A Client should not be forced to implement an interface that it doesn’t use.

This rule means that we should break our interfaces in many smaller ones, so they better satisfy the exact needs of our clients.

The ‘I ‘ in SOLID stands for interface segregation, and it simply means that **larger interfaces should be split into smaller ones. By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.**

#### Dependency Inversion

In spring framework, all modules are provided as separate components which can work together by simply injected dependencies in other module. This dependency is managed externally in XML files.