**Design Patterns : A design pattern** provides a general reusable solution for the common problems that occur in software design.The pattern shows relationships and interactions between classes or objects.

The idea is to speed up the development process by providing well tested, proven development/design paradigm.Design patterns are programming language independent strategies for solving a common problem.

That means a design pattern represents an idea, not a particular implementation. By using the design patterns you can make your code more flexible, reusable, and maintainable.

It’s not mandatory to implement design patterns in your project always. Design patterns are not meant for project development. Design patterns are meant for common problem-solving. Whenever there is a need, you have to implement a suitable pattern to avoid such problems in the future. To find out which pattern to use. You just have to try to understand the design patterns and their purposes.

**Goal:**  
Understand the purpose and usage of each design pattern. So, you will be able to pick and implement the correct pattern as needed.

**Example:**  
For **example**, in many real-world situations, we want to create only one instance of a class.

For example, there can be only one active president of the country at a time regardless of personal identity. This pattern is called a Singleton pattern.

Other software examples could be a single DB connection shared by multiple objects as creating a separate DB connection for every object will 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.

**Types of Design Patterns :**  
There are mainly three types of design patterns:

**1.Creational :**These design patterns are all about class instantiation or object creation. These patterns can be further categorized into Class-creational patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

Creational design patterns are the**Factory Method, Abstract Factory, Builder, Singleton, Object Pool, and Prototype.**

**Use case of creational design pattern:-**1) Suppose a developer wants to create a simple DBConnection class to connect to a database and wants to access the database at multiple locations from code, generally what developer will do is create an instance of DBConnection class and use it for doing database operations wherever required. Which results in creating multiple connections from the database as each instance of DBConnection class will have a separate connection to the database. In order to deal with it, we create DBConnection class as a singleton class, so that only one instance of DBConnection is created and a single connection is established. Because we can manage DB Connection via one instance so we can control load balance, unnecessary connections, etc.

2) Suppose you want to create multiple instances of similar kind and want to achieve loose [coupling](https://www.geeksforgeeks.org/coupling-in-java/) then you can go for Factory pattern. A class implementing factory design pattern works as a 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 back end, but in future need to change database to oracle, you will need to modify all your code, so as factory design patterns maintain loose coupling and easy implementation we should go for factory for achieving loose coupling and creation of similar kind of object.

**2.Structural :**These design patterns are about organizing different classes and objects to form larger structures and provide new functionality.

Structural design patterns are **Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Private Class Data, and Proxy.**

**Use Case Of Structural Design Pattern : -**

1) When 2 interfaces are not compatible with each other and want to establish a relationship between them through an adapter it’s called an adapter design pattern. Adapter pattern converts the interface of a class into another interface or class that the client expects, i.e adapter lets classes works together that could not otherwise because of incompatibility. so in these type of incompatible scenarios, we can go for the adapter pattern.

**3. Behavioral :**Behavioral patterns are about identifying common communication patterns between objects and realize these patterns.

Behavioral patterns are **Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, State, Strategy, Template method, Visitor**

**Use Case of Behavioral Design Pattern :-**

1) Template pattern defines the skeleton of an algorithm in an operation deferring some steps to sub-classes, Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm structure. For example in your project you want the behavior of the module to be able to extend, such that we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications. However, no one is allowed to make source code changes to it, i.e you can add but can’t modify the structure in those scenarios a developer can approach template design pattern.

**Singleton Design Pattern :** Sometimes we need to have only one instance of our class for example a single DB connection shared by multiple objects as creating a separate DB connection 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.  
**Definition:**The singleton pattern is a design pattern that restricts the instantiation of a class to one object.

**Method 1: Classic Implementation :**

**class** Singleton {

**private** **static** Singleton obj;

    // private constructor to force use of

    // getInstance() to create Singleton object

**private** Singleton() {}

**public** **static** Singleton getInstance()    {

**if** (obj==**null**)

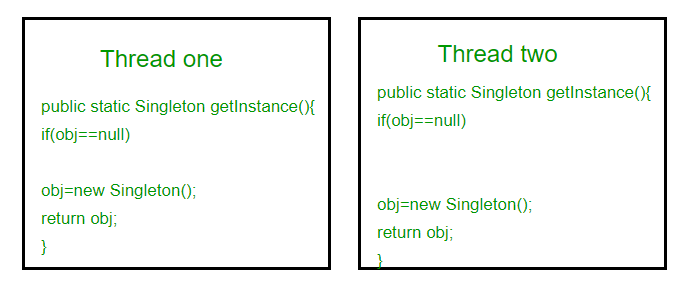
            obj = **new** Singleton();

**return** obj;

    }

}

Here we have declared getInstance() static so that we can call it without instantiating the class. The first time getInstance() is called it creates a new singleton object and after that it just returns the same object. Note that Singleton obj is not created until we need it and call getInstance() method. This is called **lazy instantiation.**The main problem with above method is that it is not thread safe.



This execution sequence creates two objects for singleton. Therefore this classic implementation is not thread safe.

**Method 2: make getInstance() synchronized**

**class** Singleton{

**private** **static** Singleton obj;

**private** Singleton() {}

    // Only one thread can execute this at a time

**public** **static** **synchronized** Singleton getInstance(){

**if** (obj==**null**)

            obj = **new** Singleton();

**return** obj;

    }

}

Here using synchronized makes sure that only one thread at a time can execute getInstance().   
The main **disadvantage** of this is method is that using synchronized every time while creating the singleton object is expensive and may decrease the performance of your program. However if performance of getInstance() is not critical for your application this method provides a clean and simple solution.

**Method 3: Eager Instantiation**

**class** Singleton{

**private** **static** Singleton obj = **new** Singleton();

**private** Singleton() {}

**public** **static** Singleton getInstance()  {

**return** obj;

    }

}

Here we have created instance of singleton in static initializer. JVM executes static initializer when the class is loaded and hence this is guaranteed to be thread safe. Use this method only when your singleton class is light and is used throughout the execution of your program.

**Method 4 (Best): Use “[Double Checked Locking](https://en.wikipedia.org/wiki/Double-checked_locking" \t "https://www.geeksforgeeks.org/singleton-design-pattern/_blank)”**  
If you notice carefully once an object is created synchronization is no longer useful because now obj will not be null and any sequence of operations will lead to consistent results.   
So we will only acquire lock on the getInstance() once, when the obj is null. This way we only synchronize the first way through, just what we want.

**class** Singleton{

**private** **static** **volatile** Singleton obj  = **null**;

**private** Singleton() {}

**public** **static** Singleton getInstance(){

**if** (obj == **null**) {

            // To make thread safe

**synchronized** (Singleton.**class**) {

                // check again as multiple threads

                // can reach above step

**if** (obj==**null**)

                    obj = **new** Singleton();

            }

        }

**return** obj;

    }

}

We have declared the obj **[volatile](https://www.geeksforgeeks.org/volatile-keyword-in-java/)**which ensures that multiple threads offer the obj variable correctly when it is being initialized to Singleton instance. This method drastically reduces the overhead of calling the synchronized method every time.

**Singleton Class in Java :**

In object-oriented programming, a singleton class is a class that can have only one object (an instance of the class) at a time.

**To design a singleton class:**

1. Make constructor as private.
2. Write a static method that has return type object of this singleton class. Here, the concept of [Lazy initialization](https://en.wikipedia.org/wiki/Lazy_initialization) is used to write this static method.

**class** Singleton {

    // static variable single\_instance of type Singleton

**private** **static** Singleton single\_instance = **null**;

    // variable of type String

**public** String s;

    // private constructor restricted to this class itself

**private** Singleton()     {

        s = "Hello I am a string part of Singleton class";

    }

      // static method to create instance of Singleton class

**public** **static** Singleton getInstance() {

**if** (single\_instance == **null**)

            single\_instance = **new** Singleton();

**return** single\_instance;

    }

}

  // Driver Class

**class** Main {

**public** **static** **void** main(String args[]) {

        // instantiating Singleton class with variable x

        Singleton x = Singleton.getInstance();

        // instantiating Singleton class with variable y

        Singleton y = Singleton.getInstance();

        // instantiating Singleton class with variable z

        Singleton z = Singleton.getInstance();

        // changing variable of instance x

        x.s = (x.s).toUpperCase();

        System.out.println("String from x is " + x.s);

        System.out.println("String from y is " + y.s);

        System.out.println("String from z is " + z.s);

        System.out.println("\n");

        // changing variable of instance z

        z.s = (z.s).toLowerCase();

        System.out.println("String from x is " + x.s);

        System.out.println("String from y is " + y.s);

        System.out.println("String from z is " + z.s);

    }

}

Singleton is a part of Gang of Four design pattern and it is categorized under creational design patterns.

An implementation of singleton class should have following properties:

1. **It should have only one instance :**This is done by providing an instance of the class from within the class. Outer classes or subclasses should be prevented to create the instance. This is done by making the constructor private in java so that no class can access the constructor and hence cannot instantiate it.
2. **Instance should be globally accessible :** Instance of singleton class should be globally accessible so that each class can use it. In Java, it is done by making the access-specifier of instance public.

**Initialization Types of Singleton: Singleton class can be instantiated by two methods:**

**1.Early initialization :** In this method, class is initialized whether it is to be used or not. The main advantage of this method is its simplicity. You initiate the class at the time of class loading. Its drawback is that class is always initialized whether it is being used or not.

**2.Lazy initialization :** In this method, class in initialized only when it is required. It can save you from instantiating the class when you don’t need it. Generally, lazy initialization is used when we create a singleton class.

**Examples of Singleton class**

**java.lang.Runtime :** Java provides a class Runtime in its lang package which is singleton in nature.

**java.awt.Desktop :**The Desktop class allows a Java application to launch associated applications registered on the native desktop to handle a URI or a file.

**Applications of Singleton classes**

There is a lot of applications of singleton pattern like cache-memory, database connection, drivers, logging.

**Important points**

> Singleton classes can have only one instance and that instance should be globally accessible.

> java.lang.Runtime and java.awt.Desktop are 2 singleton classes provided by JVM.

> Singleton Design pattern is a type of creational design pattern.

> Outer classes should be prevented to create instance of singleton class.

**how we can implement singletons in Java.**

> create a private constructor that restricts to create an object outside of the class

> create a private attribute that refers to the singleton object.

> create a public static method that allows us to create and access the object we created. Inside the method, we will create a condition that restricts us from creating more than one object.

<https://codepumpkin.com/preventing-cloning-in-singleton-design-pattern/>

**Template Method Design Pattern :** Template method design pattern is to define an algorithm as a skeleton of operations and leave the details to be implemented by the child classes. The overall structure and sequence of the algorithm is preserved by the parent class.

Template means Preset format like HTML templates which has a fixed preset format. Similarly in the template method pattern, we have a preset structure method called template method which consists of steps. This steps can be an abstract method which will be implemented by its subclasses.

This pattern comes under **behavior pattern category**. This design pattern is used popularly in framework development. This helps to avoid code duplication also.

* **AbstractClass**contains the templateMethod() which should be made final so that it cannot be overridden. This template method makes use of other operations available in order to run the algorithm but is decoupled for the actual implementation of these methods. All operations used by this template method are made abstract, so their implementation is deferred to subclasses.
* **ConcreteClass**implements all the operations required by the templateMethod that were defined as abstract in the parent class. There can be many different ConcreteClasses.



We are going to create a Game abstract class defining operations with a template method set to be final so that it cannot be overridden. Cricket and Football are concrete classes that extend Game and override its methods.

TemplatePatternDemo, our demo class, will use Game to demonstrate use of template pattern.

## **Step 1 :** Create an abstract class with a template method being final.

**Game.java**

public abstract class Game {

abstract void initialize();

abstract void startPlay();

abstract void endPlay();

**//template method**

public final void play(){

//initialize the game

initialize();

//start game

startPlay();

//end game

endPlay();

}

}

**Step 2 :**Create concrete classes extending the above class.

**Cricket.java**

public class Cricket extends Game {

@Override

void endPlay() {

System.out.println("Cricket Game Finished!");

}

@Override

void initialize() {

System.out.println("Cricket Game Initialized! Start playing.");

}

@Override

void startPlay() {

System.out.println("Cricket Game Started. Enjoy the game!");

}

}

**Football.java**

public class Football extends Game {

@Override

void endPlay() {

System.out.println("Football Game Finished!");

}

@Override

void initialize() {

System.out.println("Football Game Initialized! Start playing.");

}

@Override

void startPlay() {

System.out.println("Football Game Started. Enjoy the game!");

}

}

**Step 3 :**Use the Game's template method play() to demonstrate a defined way of playing game.

**TemplatePatternDemo.java**

public class TemplatePatternDemo {

public static void main(String[] args) {

Game game = new Cricket();

game.play();

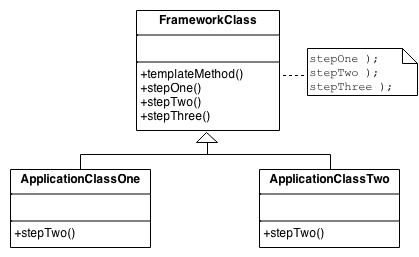
System.out.println();

game = new Football();

game.play();

}

}



**When to use template method :**The template method is used in frameworks, where each implements the invariant parts of a domain’s architecture, leaving “placeholders” for customization options.

The template method is used for the following reasons :

* Let subclasses implement varying behavior (through method overriding)
* Avoid duplication in the code, the general workflow structure is implemented once in the abstract class’s algorithm, and necessary variations are implemented in the subclasses.
* Control at what points subclassing is allowed. As opposed to a simple polymorphic override, where the base method would be entirely rewritten allowing radical change to the workflow, only the specific details of the workflow are allowed to change.