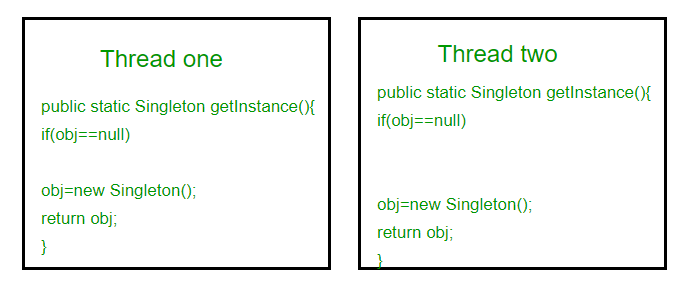
**Singleton Design Pattern | Implementation**

The singleton pattern is one of the simplest design patterns. 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.*   
Let’s see various design options for implementing such a class. If you have a good handle on static class variables and access modifiers this should not be a difficult task.  
   
**Method 1: Classic Implementation** 

* Java

|  |
| --- |
| // Classical Java implementation of singleton  // design pattern  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. Consider the following execution sequence.



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

# Prototype Design Pattern

1. [Prototype Design Pattern](https://www.javatpoint.com/prototype-design-pattern)
2. [Advantage of Prototype DP](https://www.javatpoint.com/prototype-design-pattern" \l "adv)
3. [Usage of Prototype DP](https://www.javatpoint.com/prototype-design-pattern" \l "usage)
4. [UML of Prototype DP](https://www.javatpoint.com/prototype-design-pattern" \l "uml)
5. [Example of Prototype DP](https://www.javatpoint.com/prototype-design-pattern" \l "ex)

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.

#### **Advantage of Prototype Pattern**

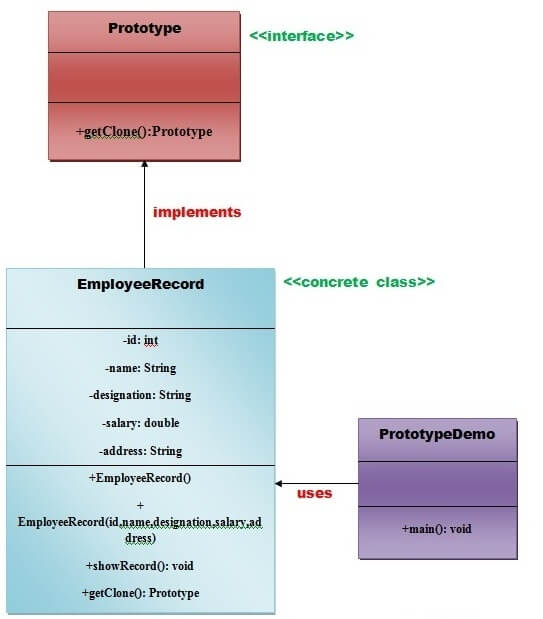
The main advantages of prototype pattern are as follows:

* It reduces the need of sub-classing.
* It hides complexities of creating objects.
* The clients can get new objects without knowing which type of object it will be.
* It lets you add or remove objects at runtime.

#### **Usage of Prototype Pattern**

* When the classes are instantiated at runtime.
* When the cost of creating an object is expensive or complicated.
* When you want to keep the number of classes in an application minimum.
* When the client application needs to be unaware of object creation and representation.

#### **UML for Prototype Pattern**



* We are going to create **an interface Prototype** that contains a method **getClone()** of **Prototype type.**
* Then, we create **a concrete class EmployeeRecord** which implements **Prototype interface** that does the cloning of EmployeeRecord object.
* **PrototypeDemo class** will uses this concrete class **EmployeeRecord.**

#### **Example of Prototype Design Pattern**

Let's see the example of prototype design pattern.

Play Videox

*File: Prototype.java*

1. **interface** Prototype {
3. **public** Prototype getClone();
5. }//End of Prototype interface.

*File: EmployeeRecord.java*

1. **class** EmployeeRecord **implements** Prototype{
3. **private** **int** id;
4. **private** String name, designation;
5. **private** **double** salary;
6. **private** String address;
8. **public** EmployeeRecord(){
9. System.out.println("   Employee Records of Oracle Corporation ");
10. System.out.println("---------------------------------------------");
11. System.out.println("Eid"+"\t"+"Ename"+"\t"+"Edesignation"+"\t"+"Esalary"+"\t\t"+"Eaddress");
13. }
15. **public**  EmployeeRecord(**int** id, String name, String designation, **double** salary, String address) {
17. **this**();
18. **this**.id = id;
19. **this**.name = name;
20. **this**.designation = designation;
21. **this**.salary = salary;
22. **this**.address = address;
23. }
25. **public** **void** showRecord(){
27. System.out.println(id+"\t"+name+"\t"+designation+"\t"+salary+"\t"+address);
28. }
30. @Override
31. **public** Prototype getClone() {
33. **return** **new** EmployeeRecord(id,name,designation,salary,address);
34. }
35. }//End of EmployeeRecord class.

*File: PrototypeDemo.java*

1. **import** java.io.BufferedReader;
2. **import** java.io.IOException;
3. **import** java.io.InputStreamReader;
5. **class** PrototypeDemo{
6. **public** **static** **void** main(String[] args) **throws** IOException {
8. BufferedReader br =**new** BufferedReader(**new** InputStreamReader(System.in));
9. System.out.print("Enter Employee Id: ");
10. **int** eid=Integer.parseInt(br.readLine());
11. System.out.print("\n");
13. System.out.print("Enter Employee Name: ");
14. String ename=br.readLine();
15. System.out.print("\n");
17. System.out.print("Enter Employee Designation: ");
18. String edesignation=br.readLine();
19. System.out.print("\n");
21. System.out.print("Enter Employee Address: ");
22. String eaddress=br.readLine();
23. System.out.print("\n");
25. System.out.print("Enter Employee Salary: ");
26. **double** esalary= Double.parseDouble(br.readLine());
27. System.out.print("\n");
29. EmployeeRecord e1=**new** EmployeeRecord(eid,ename,edesignation,esalary,eaddress);
31. e1.showRecord();
32. System.out.println("\n");
33. EmployeeRecord e2=(EmployeeRecord) e1.getClone();
34. e2.showRecord();
35. }
36. }//End of the ProtoypeDemo class.

# Builder Design Pattern

1. [Builder Design Pattern](https://www.javatpoint.com/builder-design-pattern)
2. [Advantage of Builder DP](https://www.javatpoint.com/builder-design-pattern" \l "adv)
3. [Usage of Builder DP](https://www.javatpoint.com/builder-design-pattern" \l "usage)
4. [UML of Builder DP](https://www.javatpoint.com/builder-design-pattern" \l "uml)
5. [Example of Builder DP](https://www.javatpoint.com/builder-design-pattern" \l "ex)

Builder Pattern says that **"construct a complex object from simple objects using step-by-step approach"**

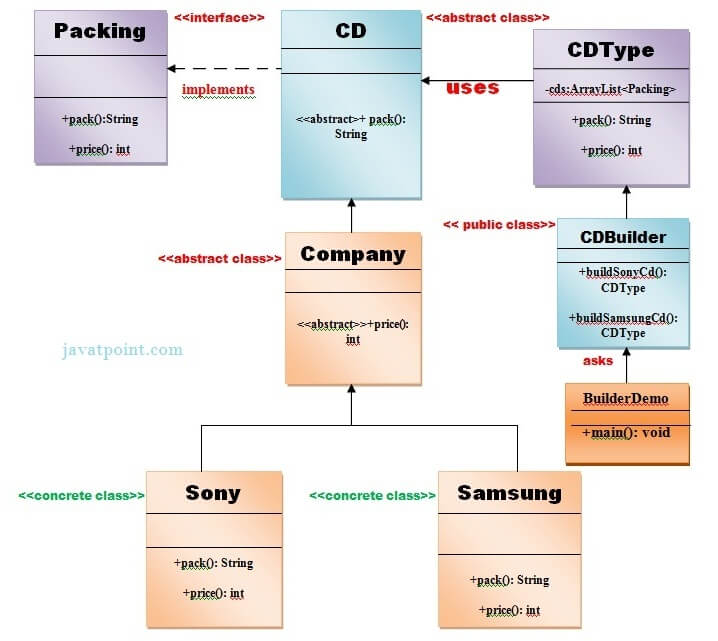
It is mostly used when object can't be created in single step like in the de-serialization of a complex object.

#### **Advantage of Builder Design Pattern**

The main advantages of Builder Pattern are as follows:

* It provides clear separation between the construction and representation of an object.
* It provides better control over construction process.
* It supports to change the internal representation of objects.

#### **UML for Builder Pattern Example**



**Factory method for designing pattern**

The factory method is a [creational design pattern](https://www.geeksforgeeks.org/design-patterns-set-1-introduction/), i.e., related to object creation. In the Factory pattern, we create objects without exposing the creation logic to the client and the client uses the same common interface to create a new type of object.

The idea is to use a static member-function (static factory method) that creates & returns instances, hiding the details of class modules from the user.  
A factory pattern is one of the core design principles to create an object, allowing clients to create objects of a library(explained below) in a way such that it doesn’t have a tight coupling with the class hierarchy of the library.

***What is meant when we talk about*libraries*and clients?***   
A library is something that is provided by some third party that exposes some public APIs and clients make calls to those public APIs to complete their tasks. A very simple example can be different kinds of Views provided by Android OS.   
   
***Why factory pattern?***   
Let us understand it with an example:

* C++

|  |
| --- |
| // A design without factory pattern  #include <iostream>  using namespace std;    // Library classes  class Vehicle {  public:      virtual void printVehicle() = 0;  };  class TwoWheeler : public Vehicle {  public:      void printVehicle()      {          cout << "I am two wheeler" << endl;      }  };  class FourWheeler : public Vehicle {  public:      void printVehicle()      {          cout << "I am four wheeler" << endl;      }  };    // Client (or user) class  class Client {  public:      Client(int type)      {            // Client explicitly creates classes according to          // type          if (type == 1)              pVehicle = new TwoWheeler();          else if (type == 2)              pVehicle = new FourWheeler();          else              pVehicle = NULL;      }        ~Client()      {          if (pVehicle) {              delete pVehicle;              pVehicle = NULL;          }      }        Vehicle\* getVehicle() { return pVehicle; }    private:      Vehicle\* pVehicle;  };    // Driver program  int main()  {      Client\* pClient = new Client(1);      Vehicle\* pVehicle = pClient->getVehicle();      pVehicle->printVehicle();      return 0;  } |

**Output**

I am two wheeler

***What*are*the problems with*the *above design?***   
As you must have observed in the above example, the Client creates objects of either TwoWheeler or FourWheeler based on some input during constructing of its object.   
Say, the library introduces a new class ThreeWheeler to incorporate three-wheeler vehicles also. What would happen? The client will end up chaining a new else if in the conditional ladder to create objects of ThreeWheeler. Which in turn will need the Client to be recompiled. So, each time a new change is made at the library side, the Client would need to make some corresponding changes at its end and recompile the code. Sounds bad? This is a very bad practice of design.

**How do avoid the problem?**   
The answer is, to create a static (or factory) method. Let us see the below code.

* C++

|  |
| --- |
| // C++ program to demonstrate factory method design pattern  #include <iostream>  using namespace std;    enum VehicleType {      VT\_TwoWheeler,    VT\_ThreeWheeler,    VT\_FourWheeler  };    // Library classes  class Vehicle {  public:      virtual void printVehicle() = 0;      static Vehicle\* Create(VehicleType type);      virtual ~Vehicle(){}  };  class TwoWheeler : public Vehicle {  public:      void printVehicle() {          cout << "I am two wheeler" << endl;      }  };  class ThreeWheeler : public Vehicle {  public:      void printVehicle() {          cout << "I am three wheeler" << endl;      }  };  class FourWheeler : public Vehicle {      public:      void printVehicle() {          cout << "I am four wheeler" << endl;      }  };    // Factory method to create objects of different types.  // Change is required only in this function to create a new object type  Vehicle\* Vehicle::Create(VehicleType type) {      if (type == VT\_TwoWheeler)          return new TwoWheeler();      else if (type == VT\_ThreeWheeler)          return new ThreeWheeler();      else if (type == VT\_FourWheeler)          return new FourWheeler();      else return NULL;  }    // Client class  class Client {  public:        // Client doesn't explicitly create objects      // but passes type to factory method "Create()"      Client()      {          VehicleType type = VT\_ThreeWheeler;          pVehicle = Vehicle::Create(type);      }      ~Client() {          if (pVehicle) {              delete pVehicle;              pVehicle = NULL;          }      }      Vehicle\* getVehicle()  {          return pVehicle;      }    private:      Vehicle \*pVehicle;  };    // Driver program  int main() {      Client \*pClient = new Client();      Vehicle \* pVehicle = pClient->getVehicle();      pVehicle->printVehicle();        delete pClient;      return 0;  } |

**Output**

I am three wheeler

In the above example, we have totally decoupled the selection of types for object creation from the Client. The library is now responsible to decide which object type to create based on an input. The client just needs to make calls to the library’s factory Create method and pass the type it wants without worrying about the actual implementation of the creation of objects.

**Other examples of the Factory Method:**

1. Say, in a ‘Drawing’ system, depending on the user’s input, different pictures like squares, rectangles, the circle can be drawn. Here we can use the factory method to create instances depending on the user’s input. For adding a new type of shape, no need to change the client’s code.
2. Another example: On the travel site, we can book train tickets as well as bus tickets and flight tickets. In this case, the user can give his travel type as ‘bus’, ‘train’, or ‘flight’.   
   Here we have an abstract class ‘AnyTravel’ with a static member function ‘GetObject’ which depending on the user’s travel type, will create & return an object of ‘BusTravel’ or ‘ TrainTravel’. ‘BusTravel’ or ‘ TrainTravel’ have common functions like passenger name, Origin, and destination parameters.