Core Java Design Patterns

In core java, there are mainly three types of design patterns, which are further divided into their sub-parts:

1.Creational Design Pattern

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

2. Structural Design Pattern

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

3. Behavioral Design Pattern

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

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).

But everyone knows an object is created by using new keyword in java. For example:

1. StudentRecord s1=**new** StudentRecord();

Hard-Coded code is not the good programming approach. Here, we are creating the instance by using the new keyword. Sometimes, the nature of the object must be changed according to the nature of the program. In such cases, we must get the help of creational design patterns to provide more general and flexible approach.

Types of creational design patterns

There are following 6 types of creational design patterns.

1. [Factory Method Pattern](https://www.javatpoint.com/factory-method-design-pattern)
2. [Abstract Factory Pattern](https://www.javatpoint.com/abstract-factory-pattern)
3. [Singleton Pattern](https://www.javatpoint.com/singleton-design-pattern-in-java)
4. [Prototype Pattern](https://www.javatpoint.com/prototype-design-pattern)
5. [Builder Pattern](https://www.javatpoint.com/builder-design-pattern)
6. [Object Pool Pattern](https://www.javatpoint.com/object-pool-pattern)

# Factory Method Pattern

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.** In other words, subclasses are responsible to create the instance of the class.

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.

#### UML for Factory Method Pattern

* We are going to create a Plan abstract class and concrete classes that extends the Plan abstract class. A factory class GetPlanFactory is defined as a next step.
* GenerateBill class will use GetPlanFactory to get a Plan object. It will pass information (DOMESTICPLAN / COMMERCIALPLAN / INSTITUTIONALPLAN) to GetPalnFactory to get the type of object it needs.

### Calculate Electricity Bill : A Real World Example of Factory Method

**Step 1:**Create a Plan abstract class.

1. **import** java.io.\*;
2. **abstract** **class** Plan{
3. **protected** **double** rate;
4. **abstract** **void** getRate();
6. **public** **void** calculateBill(**int** units){
7. System.out.println(units\*rate);
8. }
9. }//end of Plan class.

**Step 2:**Create the concrete classes that extends Plan abstract class.

1. **class**  DomesticPlan **extends** Plan{
2. //@override
3. **public** **void** getRate(){
4. rate=3.50;
5. }
6. }//end of DomesticPlan class.
7. **class**  CommercialPlan **extends** Plan{
8. //@override
9. **public** **void** getRate(){
10. rate=7.50;
11. }
12. /end of CommercialPlan **class**.
13. **class**  InstitutionalPlan **extends** Plan{
14. //@override
15. **public** **void** getRate(){
16. rate=5.50;
17. }
18. /end of InstitutionalPlan **class**.

**Step 3:**Create a GetPlanFactory to generate object of concrete classes based on given information..

1. **class** GetPlanFactory{
3. //use getPlan method to get object of type Plan
4. **public** Plan getPlan(String planType){
5. **if**(planType == **null**){
6. **return** **null**;
7. }
8. **if**(planType.equalsIgnoreCase("DOMESTICPLAN")) {
9. **return** **new** DomesticPlan();
10. }
11. **else** **if**(planType.equalsIgnoreCase("COMMERCIALPLAN")){
12. **return** **new** CommercialPlan();
13. }
14. **else** **if**(planType.equalsIgnoreCase("INSTITUTIONALPLAN")) {
15. **return** **new** InstitutionalPlan();
16. }
17. **return** **null**;
18. }
19. }//end of GetPlanFactory class.

**Step 4:**Generate Bill by using the GetPlanFactory to get the object of concrete classes by passing an information such as type of plan DOMESTICPLAN or COMMERCIALPLAN or INSTITUTIONALPLAN.

1. **import** java.io.\*;
2. **class** GenerateBill{
3. **public** **static** **void** main(String args[])**throws** IOException{
4. GetPlanFactory planFactory = **new** GetPlanFactory();
6. System.out.print("Enter the name of plan for which the bill will be generated: ");
7. BufferedReader br=**new** BufferedReader(**new** InputStreamReader(System.in));
9. String planName=br.readLine();
10. System.out.print("Enter the number of units for bill will be calculated: ");
11. **int** units=Integer.parseInt(br.readLine());
13. Plan p = planFactory.getPlan(planName);
14. //call getRate() method and calculateBill()method of DomesticPaln.
16. System.out.print("Bill amount for "+planName+" of  "+units+" units is: ");
17. p.getRate();
18. p.calculateBill(units);
19. }
20. }//end of GenerateBill class.

# Singleton design pattern in Java

1. [Singleton design pattern in Java](https://www.javatpoint.com/singleton-design-pattern-in-java)
2. [Advantage of Singleton Pattern](https://www.javatpoint.com/singleton-design-pattern-in-java#advantage)
3. [Usage of Singleton Pattern](https://www.javatpoint.com/singleton-design-pattern-in-java#usage)
4. [Example of Singleton Pattern](https://www.javatpoint.com/singleton-design-pattern-in-java#example)

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 single object can be used by all other classes.

There are two forms of singleton design pattern

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

#### 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.

#### Uml of Singleton design pattern

#### How to create Singleton design pattern?

To create the singleton class, we need to have static member of class, private constructor and static factory method.

* **Static member:** It gets memory only once because of static, itcontains the instance of the Singleton class.
* **Private constructor:** It will prevent to instantiate the Singleton class from outside the class.
* **Static factory method:** This provides the global point of access to the Singleton object and returns the instance to the caller.

### Understanding early Instantiation of Singleton Pattern

In such case, we create the instance of the class at the time of declaring the static data member, so instance of the class is created at the time of classloading.

Let's see the example of singleton design pattern using early instantiation.

*File: A.java*

1. **class** A{
2. **private** **static** A obj=**new** A();//Early, instance will be created at load time
3. **private** A(){}
5. **public** **static** A getA(){
6. **return** obj;
7. }
9. **public** **void** doSomething(){
10. //write your code
11. }
12. }

### Understanding lazy Instantiation of Singleton Pattern

In such case, we create the instance of the class in synchronized method or synchronized block, so instance of the class is created when required.

Let's see the simple example of singleton design pattern using lazy instantiation.

*File: A.java*

1. **class** A{
2. **private** **static** A obj;
3. **private** A(){}
5. **public** **static** A getA(){
6. **if** (obj == **null**){
7. **synchronized**(Singleton.**class**){
8. **if** (obj == **null**){
9. obj = **new** Singleton();//instance will be created at request time
10. }
11. }
12. }
13. **return** obj;
14. }
16. **public** **void** doSomething(){
17. //write your code
18. }
19. }

### Significance of Classloader in Singleton Pattern

#### If singleton class is loaded by two classloaders, two instance of singleton class will be created, one for each classloader.

### Significance of Serialization in Singleton Pattern

If singleton class is Serializable, you can serialize the singleton instance. Once it is serialized, you can deserialize it but it will not return the singleton object.

To resolve this issue, you need to override the **readResolve() method** that enforces the singleton. It is called just after the object is deserialized. It returns the singleton object.

1. **public** **class** A **implements** Serializable {
2. //your code of singleton
3. **protected** Object readResolve() {
4. **return** getA();
5. }
7. }

### Understanding Real Example of Singleton Pattern

* We are going to create a JDBCSingleton class. This JDBCSingleton class contains its constructor as private and a private static instance jdbc of itself.
* JDBCSingleton class provides a static method to get its static instance to the outside world. Now, JDBCSingletonDemo class will use JDBCSingleton class to get the JDBCSingleton object.

**Assumption:** you have created a table userdata that has three fields uid, uname and upassword in mysql database. Database name is ashwinirajput, username is root, password is ashwini.

*File: JDBCSingleton.java*

1. **import** java.io.BufferedReader;
2. **import** java.io.IOException;
3. **import** java.io.InputStreamReader;
4. **import** java.sql.Connection;
5. **import** java.sql.DriverManager;
6. **import** java.sql.PreparedStatement;
7. **import** java.sql.ResultSet;
8. **import** java.sql.SQLException;
10. **class** JDBCSingleton {
11. //Step 1
12. // create a JDBCSingleton class.
13. //static member holds only one instance of the JDBCSingleton class.
15. **private** **static** JDBCSingleton jdbc;
17. //JDBCSingleton prevents the instantiation from any other class.
18. **private** JDBCSingleton() {  }
20. //Now we are providing gloabal point of access.
21. **public** **static** JDBCSingleton getInstance() {
22. **if** (jdbc==**null**)
23. {
24. jdbc=**new**  JDBCSingleton();
25. }
26. **return** jdbc;
27. }
29. // to get the connection from methods like insert, view etc.
30. **private** **static** Connection getConnection()**throws** ClassNotFoundException, SQLException
31. {
33. Connection con=**null**;
34. Class.forName("com.mysql.jdbc.Driver");
35. con= DriverManager.getConnection("jdbc:mysql://localhost:3306/ashwanirajput", "root", "ashwani");
36. **return** con;
38. }
40. //to insert the record into the database
41. **public** **int** insert(String name, String pass) **throws** SQLException
42. {
43. Connection c=**null**;
45. PreparedStatement ps=**null**;
47. **int** recordCounter=0;
49. **try** {
51. c=**this**.getConnection();
52. ps=c.prepareStatement("insert into userdata(uname,upassword)values(?,?)");
53. ps.setString(1, name);
54. ps.setString(2, pass);
55. recordCounter=ps.executeUpdate();
57. } **catch** (Exception e) { e.printStackTrace(); } **finally**{
58. **if** (ps!=**null**){
59. ps.close();
60. }**if**(c!=**null**){
61. c.close();
62. }
63. }
64. **return** recordCounter;
65. }
67. //to view the data from the database
68. **public**  **void** view(String name) **throws** SQLException
69. {
70. Connection con = **null**;
71. PreparedStatement ps = **null**;
72. ResultSet rs = **null**;
74. **try** {
76. con=**this**.getConnection();
77. ps=con.prepareStatement("select \* from userdata where uname=?");
78. ps.setString(1, name);
79. rs=ps.executeQuery();
80. **while** (rs.next()) {
81. System.out.println("Name= "+rs.getString(2)+"\t"+"Paasword= "+rs.getString(3));
83. }
85. } **catch** (Exception e) { System.out.println(e);}
86. **finally**{
87. **if**(rs!=**null**){
88. rs.close();
89. }**if** (ps!=**null**){
90. ps.close();
91. }**if**(con!=**null**){
92. con.close();
93. }
94. }
95. }
97. // to update the password for the given username
98. **public** **int** update(String name, String password) **throws** SQLException  {
99. Connection c=**null**;
100. PreparedStatement ps=**null**;
102. **int** recordCounter=0;
103. **try** {
104. c=**this**.getConnection();
105. ps=c.prepareStatement(" update userdata set upassword=? where uname='"+name+"' ");
106. ps.setString(1, password);
107. recordCounter=ps.executeUpdate();
108. } **catch** (Exception e) {  e.printStackTrace(); } **finally**{
110. **if** (ps!=**null**){
111. ps.close();
112. }**if**(c!=**null**){
113. c.close();
114. }
115. }
116. **return** recordCounter;
117. }
119. // to delete the data from the database
120. **public** **int** delete(**int** userid) **throws** SQLException{
121. Connection c=**null**;
122. PreparedStatement ps=**null**;
123. **int** recordCounter=0;
124. **try** {
125. c=**this**.getConnection();
126. ps=c.prepareStatement(" delete from userdata where uid='"+userid+"' ");
127. recordCounter=ps.executeUpdate();
128. } **catch** (Exception e) { e.printStackTrace(); }
129. **finally**{
130. **if** (ps!=**null**){
131. ps.close();
132. }**if**(c!=**null**){
133. c.close();
134. }
135. }
136. **return** recordCounter;
137. }
138. }// End of JDBCSingleton class

*File: JDBCSingletonDemo.java*

1. **import** java.io.BufferedReader;
2. **import** java.io.IOException;
3. **import** java.io.InputStreamReader;
4. **import** java.sql.Connection;
5. **import** java.sql.DriverManager;
6. **import** java.sql.PreparedStatement;
7. **import** java.sql.ResultSet;
8. **import** java.sql.SQLException;
9. **class** JDBCSingletonDemo{
10. **static** **int** count=1;
11. **static** **int**  choice;
12. **public** **static** **void** main(String[] args) **throws** IOException {
14. JDBCSingleton jdbc= JDBCSingleton.getInstance();

17. BufferedReader br=**new** BufferedReader(**new** InputStreamReader(System.in));
18. **do**{
19. System.out.println("DATABASE OPERATIONS");
20. System.out.println(" --------------------- ");
21. System.out.println(" 1. Insertion ");
22. System.out.println(" 2. View      ");
23. System.out.println(" 3. Delete    ");
24. System.out.println(" 4. Update    ");
25. System.out.println(" 5. Exit      ");
27. System.out.print("\n");
28. System.out.print("Please enter the choice what you want to perform in the database: ");
30. choice=Integer.parseInt(br.readLine());
31. **switch**(choice) {
33. **case** 1:{
34. System.out.print("Enter the username you want to insert data into the database: ");
35. String username=br.readLine();
36. System.out.print("Enter the password you want to insert data into the database: ");
37. String password=br.readLine();
39. **try** {
40. **int** i= jdbc.insert(username, password);
41. **if** (i>0) {
42. System.out.println((count++) + " Data has been inserted successfully");
43. }**else**{
44. System.out.println("Data has not been inserted ");
45. }
47. } **catch** (Exception e) {
48. System.out.println(e);
49. }
51. System.out.println("Press Enter key to continue...");
52. System.in.read();
54. }//End of case 1
55. **break**;
56. **case** 2:{
57. System.out.print("Enter the username : ");
58. String username=br.readLine();
60. **try**  {
61. jdbc.view(username);
62. } **catch** (Exception e) {
63. System.out.println(e);
64. }
65. System.out.println("Press Enter key to continue...");
66. System.in.read();
68. }//End of case 2
69. **break**;
70. **case** 3:{
71. System.out.print("Enter the userid,  you want to delete: ");
72. **int** userid=Integer.parseInt(br.readLine());
74. **try** {
75. **int** i= jdbc.delete(userid);
76. **if** (i>0) {
77. System.out.println((count++) + " Data has been deleted successfully");
78. }**else**{
79. System.out.println("Data has not been deleted");
80. }
82. } **catch** (Exception e) {
83. System.out.println(e);
84. }
85. System.out.println("Press Enter key to continue...");
86. System.in.read();
88. }//End of case 3
89. **break**;
90. **case** 4:{
91. System.out.print("Enter the username,  you want to update: ");
92. String username=br.readLine();
93. System.out.print("Enter the new password ");
94. String password=br.readLine();
96. **try** {
97. **int** i= jdbc.update(username, password);
98. **if** (i>0) {
99. System.out.println((count++) + " Data has been updated successfully");
100. }
102. } **catch** (Exception e) {
103. System.out.println(e);
104. }
105. System.out.println("Press Enter key to continue...");
106. System.in.read();
108. }// end of case 4
109. **break**;
111. **default**:
112. **return**;
113. }
115. } **while** (choice!=4);
116. }
117. }

# Abstract Factory Pattern

Abstract Factory Pattern says that just **define an interface or abstract class for creating families of related (or dependent) objects but without specifying their concrete sub-classes.**That means Abstract Factory lets a class returns a factory of classes. So, this is the reason that Abstract Factory Pattern is one level higher than the Factory Pattern.

An Abstract Factory Pattern is also known as **Kit.**

#### Advantage of Abstract Factory Pattern

* Abstract Factory Pattern isolates the client code from concrete (implementation) classes.
* It eases the exchanging of object families.
* It promotes consistency among objects.

#### Usage of Abstract Factory Pattern

* When the system needs to be independent of how its object are created, composed, and represented.
* When the family of related objects has to be used together, then this constraint needs to be enforced.
* When you want to provide a library of objects that does not show implementations and only reveals interfaces.
* When the system needs to be configured with one of a multiple family of objects.

### UML for Abstract Factory Pattern

* We are going to create a **Bank interface** and a **Loan abstract class** as well as their sub-classes.
* Then we will create **AbstractFactory** class as next step.
* Then after we will create concrete classes, **BankFactory,** and **LoanFactory** that will extends **AbstractFactory class**
* After that, **AbstractFactoryPatternExample** class uses the **FactoryCreator** to get an object of **AbstractFactory** class.
* See the diagram carefully which is given below:

### Example of Abstract Factory Pattern

Here, we are calculating the loan payment for different banks like HDFC, ICICI, SBI etc.

**Step 1:** Create a Bank interface

1. **import** java.io.\*;
2. **interface** Bank{
3. String getBankName();
4. }

**Step 2:** Create concrete classes that implement the Bank interface.

1. **class** HDFC **implements** Bank{
2. **private** **final** String BNAME;
3. **public** HDFC(){
4. BNAME="HDFC BANK";
5. }
6. **public** String getBankName() {
7. **return** BNAME;
8. }
9. }
10. **class** ICICI **implements** Bank{
11. **private** **final** String BNAME;
12. ICICI(){
13. BNAME="ICICI BANK";
14. }
15. **public** String getBankName() {
16. **return** BNAME;
17. }
18. }
19. **class** SBI **implements** Bank{
20. **private** **final** String BNAME;
21. **public** SBI(){
22. BNAME="SBI BANK";
23. }
24. **public** String getBankName(){
25. **return** BNAME;
26. }
27. }

**Step 3:** Create the Loan abstract class.

1. **abstract** **class** Loan{
2. **protected** **double** rate;
3. **abstract** **void** getInterestRate(**double** rate);
4. **public** **void** calculateLoanPayment(**double** loanamount, **int** years)
5. {
6. /\*
7. to calculate the monthly loan payment i.e. EMI
9. rate=annual interest rate/12\*100;
10. n=number of monthly installments;
11. 1year=12 months.
12. so, n=years\*12;
14. \*/
16. **double** EMI;
17. **int** n;
19. n=years\*12;
20. rate=rate/1200;
21. EMI=((rate\*Math.pow((1+rate),n))/((Math.pow((1+rate),n))-1))\*loanamount;
23. System.out.println("your monthly EMI is "+ EMI +" for the amount"+loanamount+" you have borrowed");
24. }
25. }// end of the Loan abstract class.

**Step 4:** Create concrete classes that extend the Loan abstract class..

1. **class** HomeLoan **extends** Loan{
2. **public** **void** getInterestRate(**double** r){
3. rate=r;
4. }
5. }//End of the HomeLoan class.
6. **class** BussinessLoan **extends** Loan{
7. **public** **void** getInterestRate(**double** r){
8. rate=r;
9. }
11. }//End of the BusssinessLoan class.
12. **class** EducationLoan **extends** Loan{
13. **public** **void** getInterestRate(**double** r){
14. rate=r;
15. }
16. }//End of the EducationLoan class.

**Step 5:** Create an abstract class (i.e AbstractFactory) to get the factories for Bank and Loan Objects.

1. **abstract** **class** AbstractFactory{
2. **public** **abstract** Bank getBank(String bank);
3. **public** **abstract** Loan getLoan(String loan);
4. }

**Step 6:** Create the factory classes that inherit AbstractFactory class to generate the object of concrete class based on given information.

1. **class** BankFactory **extends** AbstractFactory{
2. **public** Bank getBank(String bank){
3. **if**(bank == **null**){
4. **return** **null**;
5. }
6. **if**(bank.equalsIgnoreCase("HDFC")){
7. **return** **new** HDFC();
8. } **else** **if**(bank.equalsIgnoreCase("ICICI")){
9. **return** **new** ICICI();
10. } **else** **if**(bank.equalsIgnoreCase("SBI")){
11. **return** **new** SBI();
12. }
13. **return** **null**;
14. }
15. **public** Loan getLoan(String loan) {
16. **return** **null**;
17. }
18. }//End of the BankFactory class.
19. **class** LoanFactory **extends** AbstractFactory{
20. **public** Bank getBank(String bank){
21. **return** **null**;
22. }
24. **public** Loan getLoan(String loan){
25. **if**(loan == **null**){
26. **return** **null**;
27. }
28. **if**(loan.equalsIgnoreCase("Home")){
29. **return** **new** HomeLoan();
30. } **else** **if**(loan.equalsIgnoreCase("Business")){
31. **return** **new** BussinessLoan();
32. } **else** **if**(loan.equalsIgnoreCase("Education")){
33. **return** **new** EducationLoan();
34. }
35. **return** **null**;
36. }
38. }

**Step 7:** Create a FactoryCreator class to get the factories by passing an information such as Bank or Loan.

1. **class** FactoryCreator {
2. **public** **static** AbstractFactory getFactory(String choice){
3. **if**(choice.equalsIgnoreCase("Bank")){
4. **return** **new** BankFactory();
5. } **else** **if**(choice.equalsIgnoreCase("Loan")){
6. **return** **new** LoanFactory();
7. }
8. **return** **null**;
9. }
10. }//End of the FactoryCreator.

**Step 8:** Use the FactoryCreator to get AbstractFactory in order to get factories of concrete classes by passing an information such as type.

1. **import** java.io.\*;
2. **class** AbstractFactoryPatternExample {
3. **public** **static** **void** main(String args[])**throws** IOException {
5. BufferedReader br=**new** BufferedReader(**new** InputStreamReader(System.in));
7. System.out.print("Enter the name of Bank from where you want to take loan amount: ");
8. String bankName=br.readLine();
10. System.out.print("\n");
11. System.out.print("Enter the type of loan e.g. home loan or business loan or education loan : ");
13. String loanName=br.readLine();
14. AbstractFactory bankFactory = FactoryCreator.getFactory("Bank");
15. Bank b=bankFactory.getBank(bankName);
17. System.out.print("\n");
18. System.out.print("Enter the interest rate for "+b.getBankName()+ ": ");
20. **double** rate=Double.parseDouble(br.readLine());
21. System.out.print("\n");
22. System.out.print("Enter the loan amount you want to take: ");
24. **double** loanAmount=Double.parseDouble(br.readLine());
25. System.out.print("\n");
26. System.out.print("Enter the number of years to pay your entire loan amount: ");
27. **int** years=Integer.parseInt(br.readLine());
29. System.out.print("\n");
30. System.out.println("you are taking the loan from "+ b.getBankName());
32. AbstractFactory loanFactory = FactoryCreator.getFactory("Loan");
33. Loan l=loanFactory.getLoan(loanName);
34. l.getInterestRate(rate);
35. l.calculateLoanPayment(loanAmount,years);
36. }
37. }//End of the  AbstractFactoryPatternExample

# 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#adv)
3. [Usage of Prototype DP](https://www.javatpoint.com/prototype-design-pattern#usage)
4. [UML of Prototype DP](https://www.javatpoint.com/prototype-design-pattern#uml)
5. [Example of Prototype DP](https://www.javatpoint.com/prototype-design-pattern#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.

*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#adv)
3. [Usage of Builder DP](https://www.javatpoint.com/builder-design-pattern#usage)
4. [UML of Builder DP](https://www.javatpoint.com/builder-design-pattern#uml)
5. [Example of Builder DP](https://www.javatpoint.com/builder-design-pattern#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

### Example of Builder Design Pattern

To create simple example of builder design pattern, you need to follow 6 following steps.

1. Create Packing interface
2. Create 2 abstract classes CD and Company
3. Create 2 implementation classes of Company: Sony and Samsung
4. Create the CDType class
5. Create the CDBuilder class
6. Create the BuilderDemo class

#### 1) Create Packing interface

*File: Packing.java*

1. **public** **interface** Packing {
2. **public** String pack();
3. **public** **int** price();
4. }

#### 2) Create 2 abstract classes CD and Company

Create an abstract class CD which will implement Packing interface.

*File: CD.java*

1. **public** **abstract** **class** CD **implements** Packing{
2. **public** **abstract** String pack();
3. }

*File: Company.java*

1. **public** **abstract** **class** Company **extends** CD{
2. **public** **abstract** **int** price();
3. }

#### 3) Create 2 implementation classes of Company: Sony and Samsung

*File: Sony.java*

1. **public** **class** Sony **extends** Company{
2. @Override
3. **public** **int** price(){
4. **return** 20;
5. }
6. @Override
7. **public** String pack(){
8. **return** "Sony CD";
9. }
10. }//End of the Sony class.

*File: Samsung.java*

1. **public** **class** Samsung **extends** Company {
2. @Override
3. **public** **int** price(){
4. **return** 15;
5. }
6. @Override
7. **public** String pack(){
8. **return** "Samsung CD";
9. }
10. }//End of the Samsung class.

#### 4) Create the CDType class

*File: CDType.java*

1. **import** java.util.ArrayList;
2. **import** java.util.List;
3. **public** **class** CDType {
4. **private** List<Packing> items=**new** ArrayList<Packing>();
5. **public** **void** addItem(Packing packs) {
6. items.add(packs);
7. }
8. **public** **void** getCost(){
9. **for** (Packing packs : items) {
10. packs.price();
11. }
12. }
13. **public** **void** showItems(){
14. **for** (Packing packing : items){
15. System.out.print("CD name : "+packing.pack());
16. System.out.println(", Price : "+packing.price());
17. }
18. }
19. }//End of the CDType class.

#### 5) Create the CDBuilder class

*File: CDBuilder.java*

1. **public** **class** CDBuilder {
2. **public** CDType buildSonyCD(){
3. CDType cds=**new** CDType();
4. cds.addItem(**new** Sony());
5. **return** cds;
6. }
7. **public** CDType buildSamsungCD(){
8. CDType cds=**new** CDType();
9. cds.addItem(**new** Samsung());
10. **return** cds;
11. }
12. }// End of the CDBuilder class.

#### 6) Create the BuilderDemo class

*File: BuilderDemo.java*

1. **public** **class** BuilderDemo{
2. **public** **static** **void** main(String args[]){
3. CDBuilder cdBuilder=**new** CDBuilder();
4. CDType cdType1=cdBuilder.buildSonyCD();
5. cdType1.showItems();
7. CDType cdType2=cdBuilder.buildSamsungCD();
8. cdType2.showItems();
9. }
10. }