# Creational Patterns

In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. Creational design patterns are composed of two dominant ideas. One is encapsulating knowledge about which concrete classes the system uses. Another is hiding how instances of these concrete classes are created and combined.

The creational patterns aim to separate a system from how its objects are created, composed, and represented. They increase the system’s flexibility in terms of the what, who, how, and when of object creation.

Consider applying creational patterns when:

A system should be independent of how its objects and products are created.  
A set of related objects is designed to be used together.  
Hiding the implementations of a class library or product, revealing only their interfaces.  
Constructing different representation of independent complex objects.  
A class wants its subclass to implement the object it creates.  
The class instantiations are specified at run-time.  
There must be a single instance and client can access this instance at all times.  
Instance should be extensible without being modified.

1. [Builder Design Pattern in Java](https://howtodoinjava.com/design-patterns/creational/builder-pattern-in-java/)
2. [Prototype design pattern in Java](https://howtodoinjava.com/design-patterns/creational/prototype-design-pattern-in-java/)
3. [Abstract Factory Pattern Explained](https://howtodoinjava.com/design-patterns/creational/abstract-factory-pattern-in-java/)
4. [Java Factory Pattern Explained](https://howtodoinjava.com/design-patterns/creational/implementing-factory-design-pattern-in-java/)
5. [Java Singleton Pattern Explained](https://howtodoinjava.com/design-patterns/creational/singleton-design-pattern-in-java/)

# Prototype design pattern

Please ensure that you want to [deep clone or shallow clone](https://howtodoinjava.com/java/cloning/a-guide-to-object-cloning-in-java/) your prototype because both will have different behavior on runtime. If deep copy is needed, you can use a good technique given here [using in memory serialization](https://howtodoinjava.com/java/serialization/how-to-do-deep-cloning-using-in-memory-serialization-in-java/).

## Prototype pattern – Participants

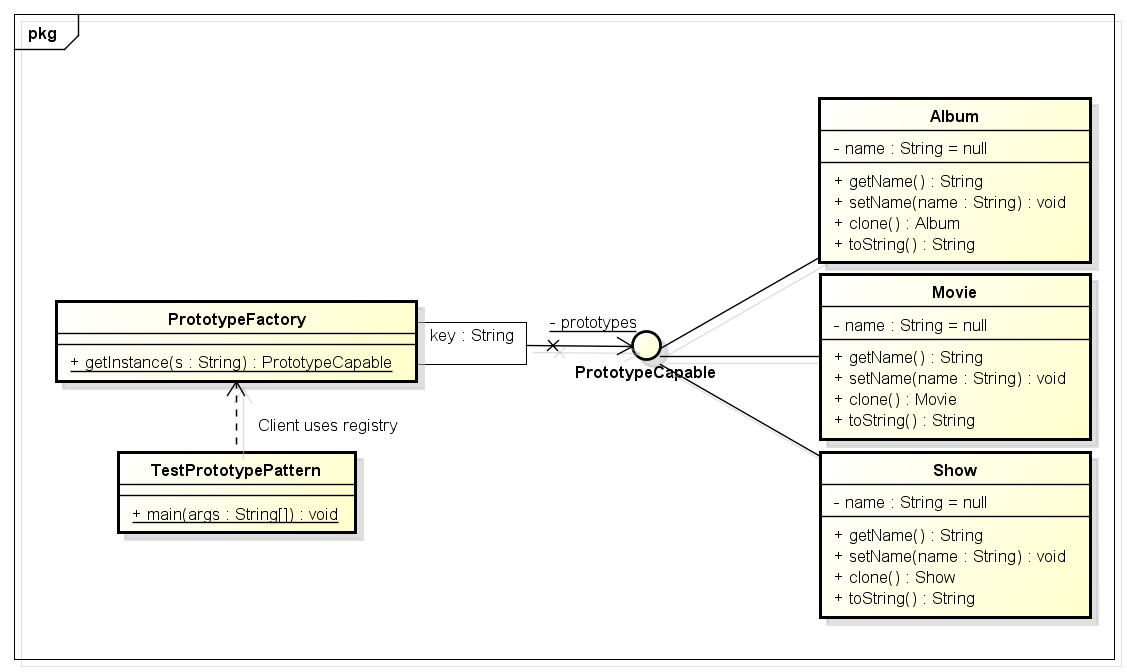
* **Prototype** : This is the prototype of actual object as discussed above.
* **Prototype registry** : This is used as registry service to have all prototypes accessible using simple string parameters.
* **Client** : Client will be responsible for using registry service to access prototype instances.

## Problem Statement

Lets understand this pattern using an example. I am creating an entertainment application that will require instances of Movie, Album and Show classes very frequently. I do not want to create their instances everytime as it is costly. So, I will create their prototype instances, and everytime when i will need a new instance, I will just clone the prototype.

## Prototype pattern example – Implementation

Lets start by creating class diagram.

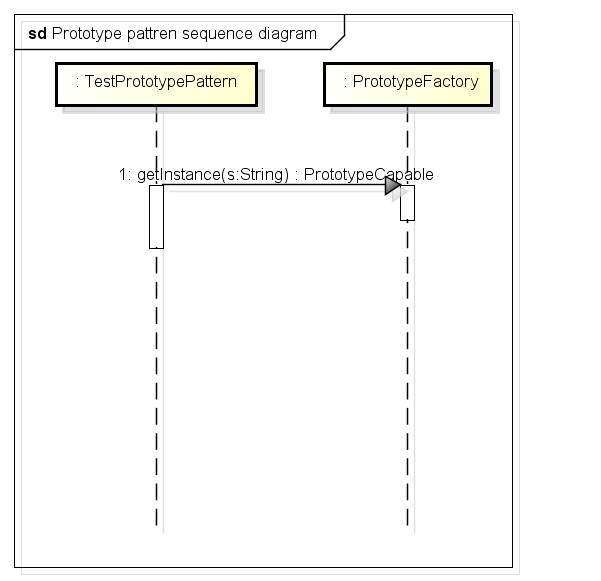


Above class diagram explains the necessary classes and their relationship.

Only one interface, “**PrototypeCapable**” is new addition in solution. The reason to use this interface is [broken behavior of Cloneable interface](https://howtodoinjava.com/java/cloning/cloneable-interface-is-broken-in-java/). This interface helps in achieving following goals:

* Ability to clone prototypes without knowing their actual types
* Provides a type reference to be used in registry

Their workflow will look like this:



Lets hit the keyboard and compose these classes.

**PrototypeCapable.java**

|  |
| --- |
| package com.howtodoinjava.prototypeDemo.contract;    public interface PrototypeCapable extends Cloneable  {      public PrototypeCapable clone() throws CloneNotSupportedException;  } |

**Movie.java, Album.java and Show.java**

|  |
| --- |
| package com.howtodoinjava.prototypeDemo.model;    import com.howtodoinjava.prototypeDemo.contract.PrototypeCapable;    public class Movie implements PrototypeCapable  {      private String name = null;      public String getName() {          return name;      }      public void setName(String name) {          this.name = name;      }      @Override      public Movie clone() throws CloneNotSupportedException {          System.out.println("Cloning Movie object..");          return (Movie) super.clone();      }      @Override      public String toString() {          return "Movie";      }  }    public class Album implements PrototypeCapable  {      private String name = null;      public String getName() {          return name;      }      public void setName(String name) {          this.name = name;      }      @Override      public Album clone() throws CloneNotSupportedException {          System.out.println("Cloning Album object..");          return (Album) super.clone();      }      @Override      public String toString() {          return "Album";      }  }    public class Show implements PrototypeCapable  {      private String name = null;      public String getName() {          return name;      }      public void setName(String name) {          this.name = name;      }      @Override      public Show clone() throws CloneNotSupportedException {          System.out.println("Cloning Show object..");          return (Show) super.clone();      }      @Override      public String toString() {          return "Show";      }  } |

**PrototypeFactory.java**

|  |
| --- |
| package com.howtodoinjava.prototypeDemo.factory;    import com.howtodoinjava.prototypeDemo.contract.PrototypeCapable;  import com.howtodoinjava.prototypeDemo.model.Album;  import com.howtodoinjava.prototypeDemo.model.Movie;  import com.howtodoinjava.prototypeDemo.model.Show;    public class PrototypeFactory  {      public static class ModelType      {          public static final String MOVIE = "movie";          public static final String ALBUM = "album";          public static final String SHOW = "show";      }      private static java.util.Map<String , PrototypeCapable> prototypes = new java.util.HashMap<String , PrototypeCapable>();        static      {          prototypes.put(ModelType.MOVIE, new Movie());          prototypes.put(ModelType.ALBUM, new Album());          prototypes.put(ModelType.SHOW, new Show());      }        public static PrototypeCapable getInstance(final String s) throws CloneNotSupportedException {          return ((PrototypeCapable) prototypes.get(s)).clone();      }  } |

**TestPrototypePattern**

|  |
| --- |
| package com.howtodoinjava.prototypeDemo.client;    import com.howtodoinjava.prototypeDemo.factory.PrototypeFactory;  import com.howtodoinjava.prototypeDemo.factory.PrototypeFactory.ModelType;    public class TestPrototypePattern  {      public static void main(String[] args)      {          try          {              String moviePrototype  = PrototypeFactory.getInstance(ModelType.MOVIE).toString();              System.out.println(moviePrototype);                String albumPrototype  = PrototypeFactory.getInstance(ModelType.ALBUM).toString();              System.out.println(albumPrototype);                String showPrototype  = PrototypeFactory.getInstance(ModelType.SHOW).toString();              System.out.println(showPrototype);            }          catch (CloneNotSupportedException e)          {              e.printStackTrace();          }      }  } |

When you run the client code, following is the output.

Cloning Movie object..

Movie

Cloning Album object..

Album

Cloning Show object..

Show

I hope, you liked this post on **Java prototype pattern example**. If any question, drop a comment.

# Explain a use case for the Builder Design Pattern.

Builder pattern aims to “Separate the construction of a complex object from its representation so that the same construction process can create different representations.”

## Where we need Builder Pattern

We already know the benefits of [**immutability**](https://howtodoinjava.com/java/basics/how-to-make-a-java-class-immutable/) and immutable instances in application. If you have any question over it, the please let me remind you of String class in java. And as I already said, builder pattern helps us in creating immutable classes with large set of state attributes.

Let’s discuss a common problem in our application. In any user management module, primary entity is User, let’s say. Ideally and practically as well, once a user object is fully created, you will not want to change it’s state. It simply does not make sense, right? Now, let’s assume, our User object has following 5 attributes i.e. firstName, lastName, age, phone and address.

In normal practice, if you want to make a immutable User class, then you must pass all five information as parameters to constructor. It will look like this:

|  |
| --- |
| public User (String firstName, String lastName, int age, String phone, String address){      this.firstName = firstName;      this.lastName = lastName;      this.age = age;      this.phone = phone;      this.address = address;  } |

Very good. Now what if only firstName and lastName are **mandatory** and rest 3 fields are optional. Problem !! We need more constructors.

|  |
| --- |
| public User (String firstName, String lastName, int age, String phone){ ... }  public User (String firstName, String lastName, String phone, String address){ ...  }  public User (String firstName, String lastName, int age){ ...   }  public User (String firstName, String lastName){ ...    } |

We will need some more like above. Still can manage? Now let’s introduce our sixth attribute i.e. salary. Now it is problem.

One way it to create more constructors, and another is to loose the immutability and introduce setter methods. You choose any of both options, you loose something, right?

Here, builder pattern will help you to consume additional attributes while retaining the immutability of Use class.

## A sample implementation using Builder Pattern

Below is the coded solution of problem we discussed above. This uses a additional class UserBuilder which helps us in building desired User object with all mandatory attributes and combination of optional attributes, without loosing the immutability.

|  |
| --- |
| public class User  {      //All final attributes      private final String firstName; // required      private final String lastName; // required      private final int age; // optional      private final String phone; // optional      private final String address; // optional        private User(UserBuilder builder) {          this.firstName = builder.firstName;          this.lastName = builder.lastName;          this.age = builder.age;          this.phone = builder.phone;          this.address = builder.address;      }        //All getter, and NO setter to provde immutability      public String getFirstName() {          return firstName;      }      public String getLastName() {          return lastName;      }      public int getAge() {          return age;      }      public String getPhone() {          return phone;      }      public String getAddress() {          return address;      }        @Override      public String toString() {          return "User: "+this.firstName+", "+this.lastName+", "+this.age+", "+this.phone+", "+this.address;      }        public static class UserBuilder      {          private final String firstName;          private final String lastName;          private int age;          private String phone;          private String address;            public UserBuilder(String firstName, String lastName) {              this.firstName = firstName;              this.lastName = lastName;          }          public UserBuilder age(int age) {              this.age = age;              return this;          }          public UserBuilder phone(String phone) {              this.phone = phone;              return this;          }          public UserBuilder address(String address) {              this.address = address;              return this;          }          //Return the finally consrcuted User object          public User build() {              User user =  new User(this);              validateUserObject(user);              return user;          }          private void validateUserObject(User user) {              //Do some basic validations to check              //if user object does not break any assumption of system          }      }  } |

And below is the way, we will use the UserBuilder in our code:

|  |
| --- |
| public static void main(String[] args) {      User user1 = new User.UserBuilder("Lokesh", "Gupta")      .age(30)      .phone("1234567")      .address("Fake address 1234")      .build();        System.out.println(user1);        User user2 = new User.UserBuilder("Jack", "Reacher")      .age(40)      .phone("5655")      //no address      .build();        System.out.println(user2);        User user3 = new User.UserBuilder("Super", "Man")      //No age      //No phone      //no address      .build();        System.out.println(user3);  }    Output:    User: Lokesh, Gupta, 30, 1234567, Fake address 1234  User: Jack, Reacher, 40, 5655, null  User: Super, Man, 0, null, null |

Please note that above created user object **does not have any setter method**, so it’s state can not be changed once it has been built. This provides the desired immutability.

Sometimes developers may forget to add support for a new attribute to the builder when they add that attribute to the User class. To minimize this, we should enclose the builders inside the class (as in above example) that they build so that it’s more obvious to the developer that there is a relevant builder that needs to be updated too.

Sometimes I think there should be a **Destroyer pattern** (opposite to builder) which should tear down certain attributes from a complex object in systematic manner. What do you think?

## Existing implementations in JDK

All implementations of [java.lang.Appendable](https://docs.oracle.com/javase/7/docs/api/java/lang/Appendable.html" \o "Appendable) are infact good example of use of Builder pattern in java. e.g.

[java.lang.StringBuilder#append()](https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html#append%28java.lang.CharSequence%29) [Unsynchronized class]

[java.lang.StringBuffer#append()](https://docs.oracle.com/javase/1.5.0/docs/api/java/lang/StringBuffer.html#append%28java.lang.CharSequence%29) [Synchronized class]

[java.nio.ByteBuffer#put()](https://docs.oracle.com/javase/6/docs/api/java/nio/ByteBuffer.html#put%28java.nio.ByteBuffer%29) (also on CharBuffer, ShortBuffer, IntBuffer, LongBuffer, FloatBuffer and DoubleBuffer)

Another use can be found in [javax.swing.GroupLayout.Group#addComponent()](https://docs.oracle.com/javase/6/docs/api/javax/swing/GroupLayout.Group.html" \l "addComponent%28java.awt.Component%29" \o "GroupLayout).

Look how similar these implementations look like what we discussed above.

|  |
| --- |
| StringBuilder builder = new StringBuilder("Temp");  String data = builder.append(1)                  .append(true)                  .append("friend")                  .toString();  System.out.println(data);  Output:  Temp1truefriend |

## Benefits and Advantages of Builder Pattern

Undoubtedly, the **number of lines of code increase** at least to double in builder pattern, but the effort pays off in terms of **design flexibility** and much more **readable code**. The **parameters to the constructor are reduced** and are provided in **highly readable method calls**.

Builder pattern also helps minimizing the number of parameters in constructor and thus there is **no need to pass in null for optional parameters** to the constructor. It’s really attracts me.

Another advantage is that Object is always instantiated in a complete state rather than sitting in an incomplete state until the developer calls (if ever calls) the appropriate “setter” method to set additional fields.

And I finally I can build **immutable objects** without much complex logic in object building process.

## Costs and Disadvantages of Builder Pattern

Though Builder pattern reduce some line of code buy eliminating the need of setter methods, still in **double up total lines** by introducing the Builder object. Furthermore, although client code is more readable, the client code is also more verbose. Though for me, readability weighs more than lines of code.

That’s only disadvantage I can think of.

# Explain a use case for the Singleton Design Pattern.

**Singleton pattern** is a design solution where an application wants to have one and only one instance of any class, in all possible scenarios without any exceptional condition. It has been debated long enough in java community regarding possible approaches to make any class singleton. Still, you will find people not satisfied with any solution you give. They cannot be overruled either. In this post, we will discuss some good approaches and will work towards our best possible effort.

[1. Singleton with eager initialization](https://howtodoinjava.com/design-patterns/creational/singleton-design-pattern-in-java/#eager_initialization)

[2. Singleton with lazy initialization](https://howtodoinjava.com/design-patterns/creational/singleton-design-pattern-in-java/#lazy_initialization)

[3. Singleton with static block initialization](https://howtodoinjava.com/design-patterns/creational/singleton-design-pattern-in-java/#static_block_initialization)

[4. Singleton with bill pugh solution](https://howtodoinjava.com/design-patterns/creational/singleton-design-pattern-in-java/#bill_pugh_singleton)

[5. Singleton using Enum](https://howtodoinjava.com/design-patterns/creational/singleton-design-pattern-in-java/#enum_singleton)

[6. Add readResolve() to singleton objects](https://howtodoinjava.com/design-patterns/creational/singleton-design-pattern-in-java/#readresolve_in_singleton)

[7. Add serialVersionUID to singleton objects](https://howtodoinjava.com/design-patterns/creational/singleton-design-pattern-in-java/#serial_version_id_in_singleton)

[8. Conclusion](https://howtodoinjava.com/design-patterns/creational/singleton-design-pattern-in-java/#conclusion)

## 1. Singleton with eager initialization

This is a design pattern where an instance of a class is created much before it is actually required. Mostly it is done on system startup. In an eager initialization singleton pattern, the singleton instance is created irrespective of whether any other class actually asked for its instance or not.

|  |
| --- |
| public class EagerSingleton {      private static volatile EagerSingleton instance = new EagerSingleton();        // private constructor      private EagerSingleton() {      }        public static EagerSingleton getInstance() {          return instance;      }  } |

The above method works fine, but it has one drawback. The instance is created irrespective of it is required in runtime or not. If this instance is not a big object and you can live with it being unused, this is the best approach.

Let’s solve the above problem in the next method.

## 2. Singleton with lazy initialization

In computer programming, [lazy initialization](https://en.wikipedia.org/wiki/Lazy_initialization) is the tactic of delaying the creation of an object, the calculation of a value, or some other expensive process, until the first time it is needed. In a singleton pattern, it restricts the creation of the instance until it is requested for first time. Lets see this in code:

|  |
| --- |
| public final class LazySingleton {      private static volatile LazySingleton instance = null;        // private constructor      private LazySingleton() {      }        public static LazySingleton getInstance() {          if (instance == null) {              synchronized (LazySingleton.class) {                  instance = new LazySingleton();              }          }          return instance;      }  } |

On the first invocation, the above method will check if the instance is already created using the instance variable. If there is no instance i.e. the instance is null, it will create an instance and will return its reference. If the instance is already created, it will simply return the reference of the instance.

But, this method also has its own drawbacks. Let’s see how. Suppose there are two threads T1 and T2. Both come to create the instance and check if “instance==null”. Now both threads have identified instance variable as null thus they both assume they must create an instance. They sequentially go into a synchronized block and create the instances. In the end, we have two instances in our application.

This error can be solved using [double-checked locking](https://en.wikipedia.org/wiki/Double_checked_locking_pattern#Usage_in_Java). This principle tells us to recheck the instance variable again in a synchronized block as given below:

|  |
| --- |
| public class LazySingleton {      private static volatile LazySingleton instance = null;        // private constructor      private LazySingleton() {      }        public static LazySingleton getInstance() {          if (instance == null) {              synchronized (LazySingleton.class) {                  // Double check                  if (instance == null) {                      instance = new LazySingleton();                  }              }          }          return instance;      }  } |

Above code is the correct implementation of the singleton pattern.

Please be sure to use “[**volatile**](https://en.wikipedia.org/wiki/Volatile_variable#In_Java)” keyword with instance variable otherwise you can run into an out of order write error scenario, where reference of an instance is returned before actually the object is constructed i.e. JVM has only allocated the memory and constructor code is still not executed. In this case, your other thread, which refers to the uninitialized object may throw null pointer exception and can even crash the whole application.

## 3. Singleton with static block initialization

If you have an idea of the class loading sequence, you can use the fact that static blocks are executed during the loading of a class, even before the constructor is called. We can use this feature in our singleton pattern like this:

|  |
| --- |
| public class StaticBlockSingleton {      private static final StaticBlockSingleton INSTANCE;        static {          try {              INSTANCE = new StaticBlockSingleton();          } catch (Exception e) {              throw new RuntimeException("Uffff, i was not expecting this!", e);          }      }        public static StaticBlockSingleton getInstance() {          return INSTANCE;      }        private StaticBlockSingleton() {          // ...      }  } |

The above code has one drawback. Suppose there are 5 static fields in a class and the application code needs to access only 2 or 3, for which instance creation is not required at all. So, if we use this static initialization, we will have one instance created though it is required or not.

The next section will overcome this problem.

## 4. Singleton with bill pugh solution

Bill Pugh was main force behind the [java memory model](https://en.wikipedia.org/wiki/Java_Memory_Model) changes. His principle “[Initialization-on-demand holder idiom](https://en.wikipedia.org/wiki/Initialization_on_demand_holder_idiom)” also uses the static block idea, but in a different way. It suggest to use static inner class.

|  |
| --- |
| public class BillPughSingleton {      private BillPughSingleton() {      }        private static class LazyHolder {          private static final BillPughSingleton INSTANCE = new BillPughSingleton();      }        public static BillPughSingleton getInstance() {          return LazyHolder.INSTANCE;      }  } |

As you can see, until we need an instance, the LazyHolder class will not be initialized until required and you can still use other static members of BillPughSingleton class. ***This is the solution, i will recommend to use. I have used it in my all projects.***

## 5. Singleton using Enum

This type of implementation employs the use of enum. [Enum](https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html" \o "enum in java), as written in the java docs, provided implicit support for thread safety and only one instance is guaranteed. **Java enum singleton** is also a good way to have singleton with minimal effort.

|  |
| --- |
| public enum EnumSingleton {      INSTANCE;      public void someMethod(String param) {          // some class member      }  } |

## 6. Add readResolve() to Singleton Objects

By now you must have made your decision about how you would like to implement your singleton. Now let’s see other problems that may arise even in job interviews.

Let’s say your application is distributed and it frequently serializes objects into the file system, only to read them later when required. Please note that de-serialization always creates a new instance. Let’s understand using an example:

Our singleton class is:

|  |
| --- |
| public class DemoSingleton implements Serializable {      private volatile static DemoSingleton instance = null;        public static DemoSingleton getInstance() {          if (instance == null) {              instance = new DemoSingleton();          }          return instance;      }        private int i = 10;        public int getI() {          return i;      }        public void setI(int i) {          this.i = i;      }  } |

Let’s serialize this class and de-serialize it after making some changes:

|  |
| --- |
| public class SerializationTest {      static DemoSingleton instanceOne = DemoSingleton.getInstance();        public static void main(String[] args) {          try {              // Serialize to a file              ObjectOutput out = new ObjectOutputStream(new FileOutputStream(                      "filename.ser"));              out.writeObject(instanceOne);              out.close();                instanceOne.setI(20);                // Serialize to a file              ObjectInput in = new ObjectInputStream(new FileInputStream(                      "filename.ser"));              DemoSingleton instanceTwo = (DemoSingleton) in.readObject();              in.close();                System.out.println(instanceOne.getI());              System.out.println(instanceTwo.getI());            } catch (IOException e) {              e.printStackTrace();          } catch (ClassNotFoundException e) {              e.printStackTrace();          }      }  }    Output:  20  10 |

Unfortunately, both variables have different values of the variable “i”. Clearly, there are two instances of our class. So, again we are in the same problem of multiple instances in our application.

To solve this issue, we need to include a readResolve() method in our DemoSingleton class. This method will be invoked when you will de-serialize the object. Inside of this method, you must return the existing instance to ensure a single instance application wide.

|  |
| --- |
| public class DemoSingleton implements Serializable {      private volatile static DemoSingleton instance = null;        public static DemoSingleton getInstance() {          if (instance == null) {              instance = new DemoSingleton();          }          return instance;      }        protected Object readResolve() {          return instance;      }        private int i = 10;        public int getI() {          return i;      }        public void setI(int i) {          this.i = i;      }  } |

Now when you execute the class SerializationTest, it will give you correct output.

|  |
| --- |
| 20  20 |

## 7. Add serialVersionUId to singleton objects

So far so good. Untill now, we have solved both of the problems of synchronization and serialization. Now, we are just one step away from a correct and complete implementation. The only missing part is a serial version id.

This is required in cases where your class structure changes between serialization and deserialization. A changed class structure will cause the JVM to give an exception in the de-serializing process.

|  |
| --- |
| java.io.InvalidClassException: singleton.DemoSingleton; local class incompatible: stream classdesc serialVersionUID = 5026910492258526905, local class serialVersionUID = 3597984220566440782  at java.io.ObjectStreamClass.initNonProxy(Unknown Source)  at java.io.ObjectInputStream.readNonProxyDesc(Unknown Source)  at java.io.ObjectInputStream.readClassDesc(Unknown Source)  at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)  at java.io.ObjectInputStream.readObject0(Unknown Source)  at java.io.ObjectInputStream.readObject(Unknown Source)  at singleton.SerializationTest.main(SerializationTest.java:24) |

This problem can be solved only by adding a unique serial version id to the class. It will prevent the compiler from throwing the exception by telling it that both classes are same, and will load the available instance variables only.

## 8. Conclusion

After having discussed so many possible approaches and other possible error cases, I will recommend to you the code template below, to design your singleton class which shall ensure only one instance of a class in the whole application in all above discussed scenarios.

|  |
| --- |
| public class DemoSingleton implements Serializable {      private static final long serialVersionUID = 1L;        private DemoSingleton() {          // private constructor      }        private static class DemoSingletonHolder {          public static final DemoSingleton INSTANCE = new DemoSingleton();      }        public static DemoSingleton getInstance() {          return DemoSingletonHolder.INSTANCE;      }        protected Object readResolve() {          return getInstance();      }  } |

I hope this post has enough information to help make you understand the most common approaches for the **singleton pattern** and **singleton best practices**. Let me know your thoughts.

Happy Learning !!

**Realtime Singleton Examples** – I just thought to add some examples which can be referred for further study and mention in interviews:

* [java.awt.Desktop#getDesktop()](https://docs.oracle.com/javase/6/docs/api/java/awt/Desktop.html#getDesktop%28%29)
* [java.lang.Runtime#getRuntime()](https://docs.oracle.com/javase/6/docs/api/java/lang/Runtime.html#getRuntime%28%29)

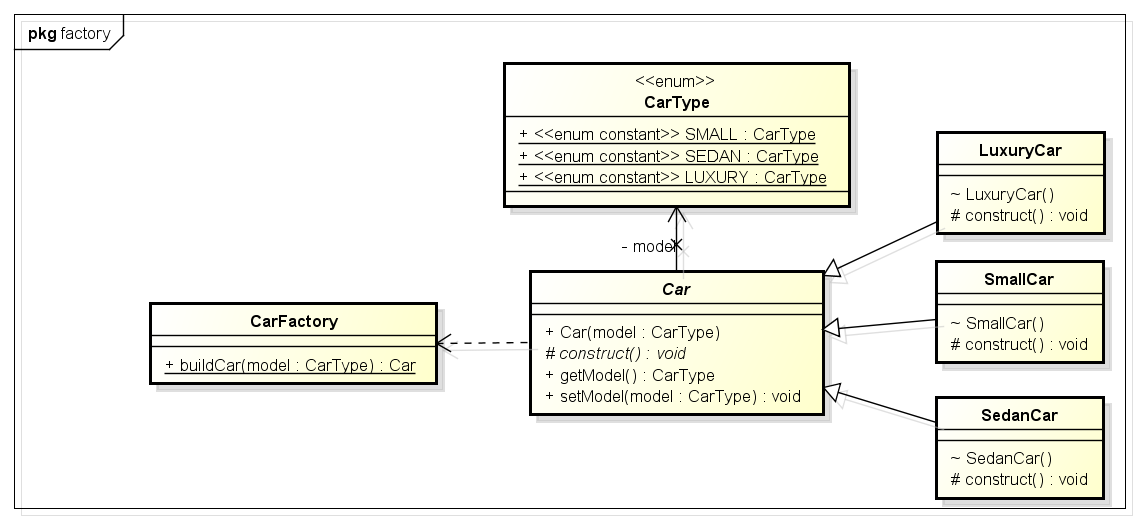
# Explain a use case for the Factory Design Pattern.

## [Factory pattern](https://en.wikipedia.org/wiki/Factory_method_pattern), for creating instances for your classes. Factory, as the name suggests, is a place to create some different products which are somehow similar in features yet divided into categories.

## 1. When to use factory pattern?

Factory pattern introduces **loose coupling between classes** which is the most important principle one should consider and apply while designing the application architecture. Loose coupling can be introduced in application architecture by programming against abstract entities rather than concrete implementations. This not only makes our architecture more flexible but also less fragile.

A picture is worth a thousand words. Let’s see how a factory implementation will look like.



Above class-diagram depicts a common scenario using an example of a car factory which is able to build 3 types of cars i.e. small, sedan and luxury. Building a car requires many steps from allocating accessories to final makeup. These steps can be written in programming as methods and should be called while creating an instance of a specific car type.

## 2. Java Factory Pattern Example

So far we have design the classes need to be designed for making a **CarFactory**. Let’s create them now.

#### 2.1. Object types

CarType will hold the types of car and will provide car types to all other classes.

|  |
| --- |
| CarType.java |
| package designPatterns.creational.factory;    public enum CarType {      SMALL, SEDAN, LUXURY  } |

#### 2.2. Object implementations

Car is parent class of all car instances and it will also contain the common logic applicable in car making of all types.

|  |
| --- |
| Car.java |
| package designPatterns.creational.factory;    public abstract class Car {        public Car(CarType model) {          this.model = model;          arrangeParts();      }        private void arrangeParts() {          // Do one time processing here      }        // Do subclass level processing in this method      protected abstract void construct();        private CarType model = null;        public CarType getModel() {          return model;      }        public void setModel(CarType model) {          this.model = model;      }  } |

LuxuryCar is concrete implementation of car type LUXURY.

|  |
| --- |
| LuxuryCar.java |
| package designPatterns.creational.factory;    public class LuxuryCar extends Car {        LuxuryCar() {          super(CarType.LUXURY);          construct();      }        @Override      protected void construct() {          System.out.println(&quot;Building luxury car&quot;);          // add accessories      }  } |

SmallCar is concrete implementation of car type SMALL.

|  |
| --- |
| SmallCar.java |
| package designPatterns.creational.factory;    public class SmallCar extends Car {        SmallCar() {          super(CarType.SMALL);          construct();      }        @Override      protected void construct() {          System.out.println(&quot;Building small car&quot;);          // add accessories      }  } |

SedanCar is concrete implementation of car type SEDAN.

|  |
| --- |
| SedanCar.java |
| package designPatterns.creational.factory;    public class SedanCar extends Car {        SedanCar() {          super(CarType.SEDAN);          construct();      }        @Override      protected void construct() {          System.out.println(&quot;Building sedan car&quot;);          // add accessories      }  } |

#### 2.3. Factory to create objects

CarFactory.java is our main class implemented using factory pattern. It instantiates a car instance only after determining its type.

|  |
| --- |
| CarFactory.java |
| package designPatterns.creational.factory;    public class CarFactory {      public static Car buildCar(CarType model) {          Car car = null;          switch (model) {          case SMALL:              car = new SmallCar();              break;            case SEDAN:              car = new SedanCar();              break;            case LUXURY:              car = new LuxuryCar();              break;            default:              // throw some exception              break;          }          return car;      }  } |

#### 2.4. Test factory pattern

In TestFactoryPattern, we will test our factory code. Lets run this class.

|  |
| --- |
| CarFactory.java |
| package designPatterns.creational.factory;    public class TestFactoryPattern {      public static void main(String[] args) {          System.out.println(CarFactory.buildCar(CarType.SMALL));          System.out.println(CarFactory.buildCar(CarType.SEDAN));          System.out.println(CarFactory.buildCar(CarType.LUXURY));      }  } |

Program Output.

|  |
| --- |
| Console |
| Building small car  designPatterns.creational.factory.SmallCar@7c230be4  Building sedan car  designPatterns.creational.factory.SedanCar@60e1e567  Building luxury car  designPatterns.creational.factory.LuxuryCar@e9bfee2 |

As you can see, the factory is able to return any type of car instance it is requested for. It will help us in making any kind of changes in car making process without even touching the composing classes i.e. classes using CarFactory.

## 3. Benefits of factory pattern

By now, you should be able to count the main advantages of using the factory pattern. Let’s note down:

1. The creation of an object precludes its reuse without significant duplication of code.
2. The creation of an object requires access to information or resources that should not be contained within the composing class.
3. The lifetime management of the generated objects must be centralized to ensure a consistent behavior within the application.

## 4. Final notes

**Factory pattern is most suitable where there is some complex object creation steps are involved**. To ensure that these steps are centralized and not exposed to composing classes, factory pattern should be used. We can see many realtime examples of factory pattern in JDK itself e.g.

* [java.sql.DriverManager#getConnection()](https://docs.oracle.com/javase/10/docs/api/java/sql/DriverManager.html#getConnection%28java.lang.String%29)
* [java.net.URL#openConnection()](https://docs.oracle.com/javase/10/docs/api/java/net/URL.html#openConnection%28%29)
* [java.lang.Class#newInstance()](https://docs.oracle.com/javase/10/docs/api/java/lang/Class.html#newInstance%28%29)
* [java.lang.Class#forName()](https://docs.oracle.com/javase/10/docs/api/java/lang/Class.html#forName%28java.lang.String%29)

I hope, I have included enough information in this **Java factory pattern example** to make this post informative.

# Explain a use case for the Factory of Factory Design Pattern.

**Abstract factory pattern** is yet another [creational design pattern](https://howtodoinjava.com/design-patterns/creational/) and is considered as another layer of abstraction over [factory pattern](https://howtodoinjava.com/design-patterns/creational/implementing-factory-design-pattern-in-java/). In this tutorial, we will expand the scope of car factory problem discussed in [factory pattern](https://howtodoinjava.com/design-patterns/creational/implementing-factory-design-pattern-in-java/).

## 1. Design global car factory using abstract factory pattern

In “[factory design pattern](https://howtodoinjava.com/design-patterns/creational/implementing-factory-design-pattern-in-java/)“, we discussed how to abstract the car making process for various car model types and their additional logic included in car making process. Now, imagine if our car maker decides to go global.

To **support global operations**, we will require to enhance the system to support different car making styles for different countries. For example, in some countries we see steering wheel on left side, and in some countries it is on right side. There can be many more such differences in different part of cars and their making processes.

To describable the abstract factory pattern, we will consider 3 kind of makes – the *USA*, *Asia*, and the *default* for all other countries. Supporting multiple locations will need critical design changes.

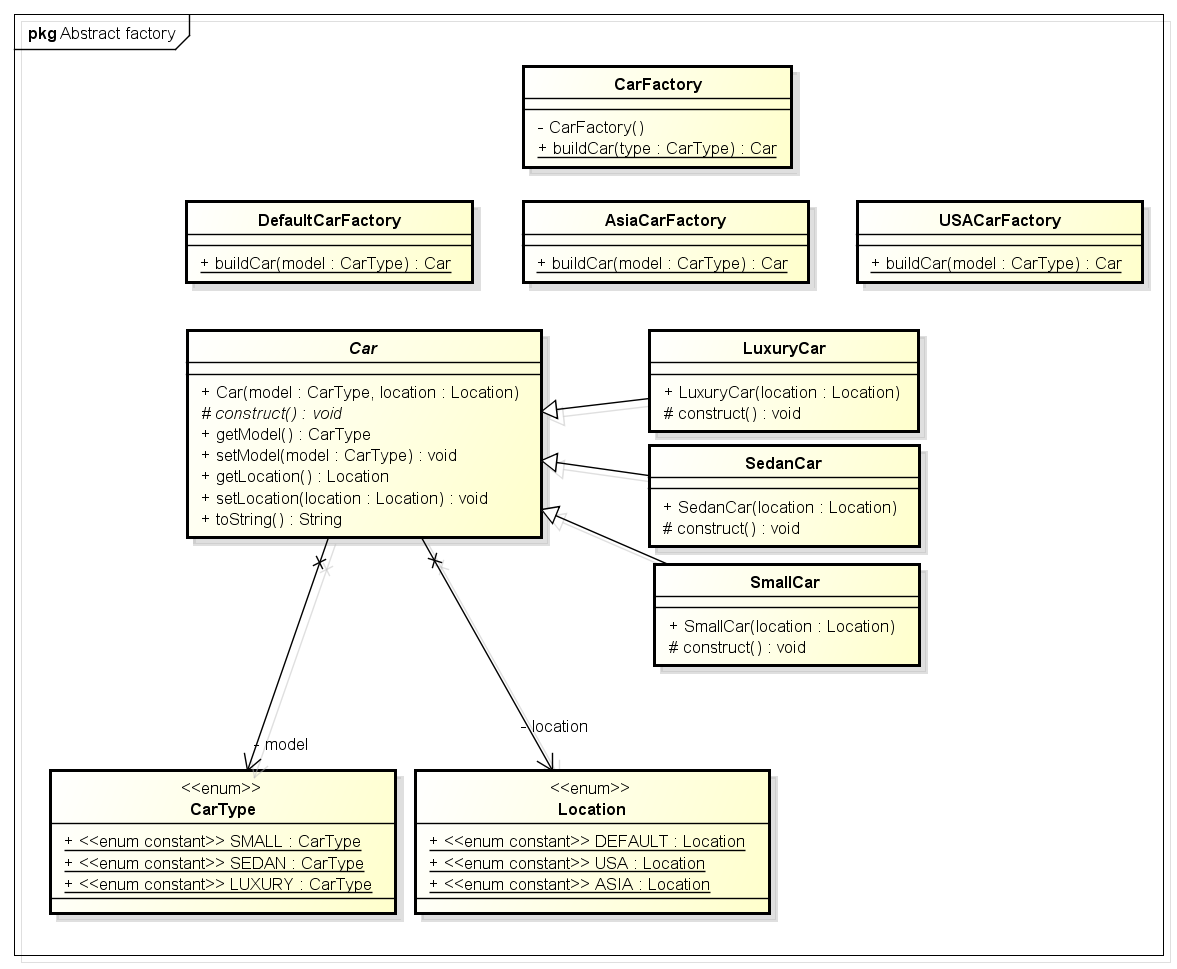
First of all, we need car factories in each **location** specified in the problem statement. i.e. *USACarFactory*, *AsiaCarFactory* and *DefaultCarFactory*. Now, our application should be smart enough to identify the location where is being used, so we should be able to use appropriate car factory without even knowing which car factory implementation will be used internally. This also saves us from someone calling the wrong factory for a particular location.

So basically, we need another layer of abstraction which will identify the location and internally use correct car factory implementation without even giving a single hint to the user. This is exactly the problem, which abstract factory pattern is used to solve.

## 2. Abstract factory pattern based solution

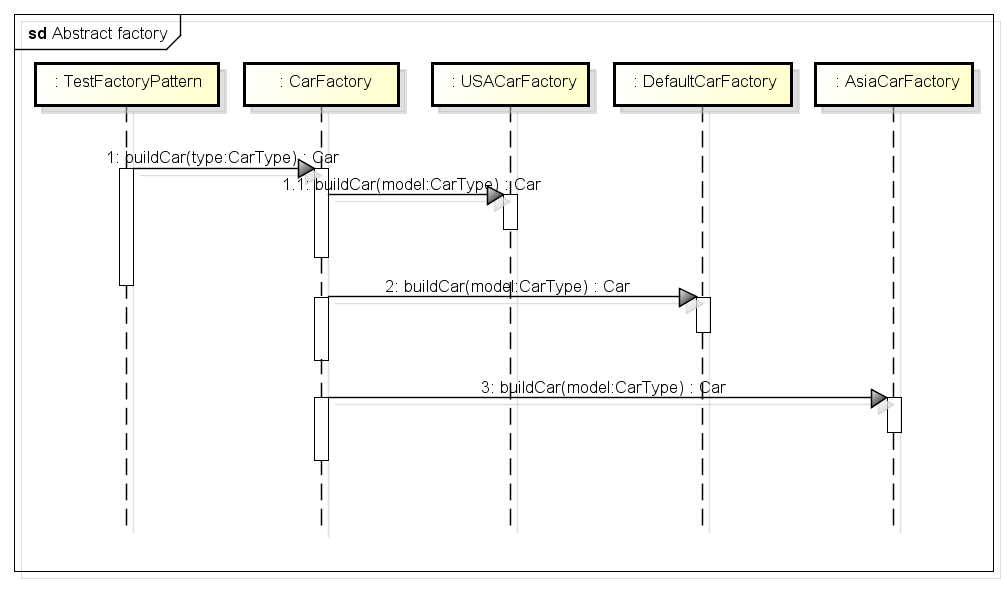
#### 2.1. Package Diagram

Class diagram for participating classes in design of global car factory using abstract factory pattern.



#### 2.2. Sequence Diagram

This diagram shows the interaction between classes and abstraction behind CarFactory factory class.



Please note that I have designed the solution to completely hide the location detail with end user. So, I have not exposed any location specific factory directly.  
  
In alternate solution, we can first get the location specific factory based on location argument and then use it’s buildCar() method on abstract reference to build the actual car instance.

## 3. Abstract factory pattern implementation

Java classes implementing abstract factory pattern for global car factory.

First, wee have to write all separate car factories for different locations. To support, location specific features, begin with modifying our Car.java class with another attribute – location.

|  |
| --- |
| **Car.java** |
| **public abstract class Car {**    **public Car(CarType model, Location location){**  **this.model = model;**  **this.location = location;**  **}**    **protected abstract void construct();**    **private CarType model = null;**  **private Location location = null;**    **//getters and setters**    **@Override**  **public String toString() {**  **return "Model- "+model + " built in "+location;**  **}**  **}** |

This adds extra work of creating another enum for storing different locations.

|  |
| --- |
| **Location.java** |
| **public enum Location {**  **DEFAULT, USA, ASIA**  **}** |

All car types will also have additional location property. We are writing only for the luxury car. Same follows for small and sedan also.

|  |
| --- |
| **LuxuryCar.java** |
| **public class LuxuryCar extends Car**  **{**  **public LuxuryCar(Location location)**  **{**  **super(CarType.LUXURY, location);**  **construct();**  **}**    **@Override**  **protected void construct() {**  **System.out.println("Building luxury car");**  **//add accessories**  **}**  **}** |

So far we have created basic classes. Now let’s have different car factories which is the core idea behind abstract factory pattern.

|  |
| --- |
| **AsiaCarFactory.java** |
| **public class AsiaCarFactory**  **{**  **public static Car buildCar(CarType model)**  **{**  **Car car = null;**  **switch (model)**  **{**  **case SMALL:**  **car = new SmallCar(Location.ASIA);**  **break;**    **case SEDAN:**  **car = new SedanCar(Location.ASIA);**  **break;**    **case LUXURY:**  **car = new LuxuryCar(Location.ASIA);**  **break;**    **default:**  **//throw some exception**  **break;**  **}**  **return car;**  **}**  **}** |
| **DefaultCarFactory.java** |
| **public class DefaultCarFactory**  **{**  **public static Car buildCar(CarType model)**  **{**  **Car car = null;**  **switch (model)**  **{**  **case SMALL:**  **car = new SmallCar(Location.DEFAULT);**  **break;**    **case SEDAN:**  **car = new SedanCar(Location.DEFAULT);**  **break;**    **case LUXURY:**  **car = new LuxuryCar(Location.DEFAULT);**  **break;**    **default:**  **//throw some exception**  **break;**  **}**  **return car;**  **}**  **}** |

|  |
| --- |
| **USACarFactory.java** |
| **public class USACarFactory**  **{**  **public static Car buildCar(CarType model)**  **{**  **Car car = null;**  **switch (model)**  **{**  **case SMALL:**  **car = new SmallCar(Location.USA);**  **break;**    **case SEDAN:**  **car = new SedanCar(Location.USA);**  **break;**    **case LUXURY:**  **car = new LuxuryCar(Location.USA);**  **break;**    **default:**  **//throw some exception**  **break;**  **}**  **return car;**  **}**  **}** |

Well, now we have all 3 different Car factories. Now, we have to abstract the way these factories are accessed.

|  |
| --- |
| **CarFactory.java** |
| **public class CarFactory**  **{**  **private CarFactory() {**  **//Prevent instantiation**  **}**    **public static Car buildCar(CarType type)**  **{**  **Car car = null;**  **Location location = Location.ASIA; //Read location property somewhere from configuration**  **//Use location specific car factory**  **switch(location)**  **{**  **case USA:**  **car = USACarFactory.buildCar(type);**  **break;**  **case ASIA:**  **car = AsiaCarFactory.buildCar(type);**  **break;**  **default:**  **car = DefaultCarFactory.buildCar(type);**  **}**  **return car;**  **}**  **}** |

We are done with writing code. Now, let’s test the factories and cars.

|  |
| --- |
| **TestFactoryPattern.java** |
| **public class TestFactoryPattern**  **{**  **public static void main(String[] args)**  **{**  **System.out.println(CarFactory.buildCar(CarType.SMALL));**  **System.out.println(CarFactory.buildCar(CarType.SEDAN));**  **System.out.println(CarFactory.buildCar(CarType.LUXURY));**  **}**  **}** |

Program output:

|  |
| --- |
| **Console** |
| **Output: (Default location is Asia)**    **Building small car**  **Model- SMALL built in ASIA**    **Building sedan car**  **Model- SEDAN built in ASIA**    **Building luxury car**  **Model- LUXURY built in ASIA** |

## 4. Summary

We already have seen the use case scenarios of [Factory pattern](https://howtodoinjava.com/design-patterns/creational/implementing-factory-design-pattern-in-java/) so whenever you need **another level of abstraction over a group of factories**, you should consider using the *abstract factory pattern*. It is probably only **difference between factory pattern vs abstract factory pattern**.

You can already look deeper into different *real time examples of abstract factory* in JDK distribution:

* [DocumentBuilderFactory#newInstance()](https://docs.oracle.com/javase/6/docs/api/javax/xml/parsers/DocumentBuilderFactory.html#newInstance%28%29)
* [TransformerFactory#newInstance()](https://docs.oracle.com/javase/6/docs/api/javax/xml/transform/TransformerFactory.html#newInstance%28%29)

There are other similar examples but the need is to have the feel of the abstract factory design pattern, which you must have got till now.

# Structural Patterns

Structural design patterns show you how to glue different pieces of a system together in a flexible and extensible fashion. They help you guarantee that when one of the parts changes, the entire structure does not need to change.

These patterns focus on, how the classes inherit from each other and how they are composed from other classes. Structural patterns use inheritance to compose interface or implementations. Structural object patterns describe ways to compose objects to realize new functionality

1. [Proxy Design Pattern](https://howtodoinjava.com/design-patterns/structural/proxy-design-pattern/)
2. [Flyweight Design Pattern](https://howtodoinjava.com/design-patterns/structural/flyweight-design-pattern/)
3. [Facade Design Pattern](https://howtodoinjava.com/design-patterns/structural/facade-design-pattern/)
4. [Composite Design Pattern](https://howtodoinjava.com/design-patterns/structural/composite-design-pattern/)
5. [Bridge Design Pattern](https://howtodoinjava.com/design-patterns/structural/bridge-design-pattern/)
6. [Adapter Design Pattern in Java](https://howtodoinjava.com/design-patterns/structural/adapter-design-pattern-in-java/)
7. [Decorator Design Pattern in Java](https://howtodoinjava.com/design-patterns/structural/decorator-design-pattern/)

# Proxy Design Pattern

According to GoF definition of **proxy design pattern**, a proxy object provide **a surrogate or placeholder** for another object to control access to it. A proxy is basically a substitute for an intended object which we create due to many reasons e.g. security reasons or cost associated with creating fully initialized original object.

## 1. When to use proxy design pattern

A proxy object hides the original object and control access to it. We can use proxy when we may want to use a class that can perform as an interface to something else.

Proxy is heavily used to implement lazy loading related usecases where we do not want to create full object until it is actually needed.

A proxy can be used to add an additional security layer around the original object as well.

## 2. Real world example of proxy pattern

* In [hibernate](https://howtodoinjava.com/hibernate-tutorials/), we write the code to fetch entities from the database. Hibernate returns an object which a proxy (by dynamically constructed by Hibernate by extending the domain class) to the underlying entity class. The client code is able to read the data whatever it needs to read with the proxy.

These proxy entity classes help in implementing lazy loading scenarios where associated entities are fetched only when they are requested explicitly. It helps in improving performance of DAO operations.

* In corporate networks, internet access is guarded behind a network proxy. All network requests goes through proxy which first check the requests for allowed websites and posted data to network. If request looks suspicious, proxy block the request – else request pass through.
* In aspect oriented programming (AOP), an object created by the AOP framework in order to implement the aspect contracts (advise method executions and so on). For example, in the [Spring AOP](https://howtodoinjava.com/spring-aop/spring-aop-aspectj-example-tutorial-using-annotation-config/), an AOP proxy will be a JDK dynamic proxy or a CGLIB proxy.

#### Design participants

* **Subject** – is an interface which expose the functionality available to be used by the clients.
* **Real Subject** – is a class implementing Subject and it is concrete implementation which needs to be hidden behind a proxy.
* **Proxy** – hides the real object by extending it and clients communicate to real object via this proxy object. Usually frameworks create this proxy object when client request for real object.

## 4. Proxy design pattern example

In given example, we have a RealObject which client need to access to do something. It will ask the framework to provide an instance of RealObject. But as the access to this object needs to be guarded, framework returns the reference to RealObjectProxy.

Any call to proxy object is used for additional requirements and the call is passed to real object.

|  |
| --- |
| **RealObject.java** |
| **public interface RealObject**  **{**  **public void doSomething();**  **}** |
| **RealObjectImpl.java** |
| **public class RealObjectImpl implements RealObject {**    **@Override**  **public void doSomething() {**  **System.out.println("Performing work in real object");**  **}**    **}** |

|  |
| --- |
| **RealObjectProxy.java** |
| **public class RealObjectProxy extends RealObjectImpl**  **{**  **@Override**  **public void doSomething()**  **{**  **//Perform additional logic and security**  **//Even we can block the operation execution**  **System.out.println("Delegating work on real object");**  **super.doSomething();**  **}**  **}** |
| **Client.java** |
| **public class Client**  **{**  **public static void main(String[] args)**  **{**  **RealObject proxy = new RealObjectProxy();**  **proxy.doSomething();**  **}**  **}** |

Program output.

|  |
| --- |
| **Console** |
| **Delegating work on real object**  **Performing work in real object** |

## 5. FAQs

#### 5.1. what are different types of proxies

Proxies are generally divided into four types –

1. **Remote proxy** – represent a remotely lactated object. To talk with remote objects, the client need to do additional work on communication over network. A proxy object does this communication on behalf of original object and client focuses on real talk to do.
2. **Virtual proxy** – delay the creation and initialization of expensive objects until needed, where the objects are created on demand. Hibernate created proxy entities are example of virtual proxies.
3. **Protection proxy** – help to implement security over original object. They may check for access rights before method invocations and allow or deny access based on the conclusion.
4. **Smart Proxy** – performs additional housekeeping work when an object is accessed by a client. An example can be to check if the real object is locked before it is accessed to ensure that no other object can change it.

#### 5.2. Proxy pattern vs decorator pattern

The primary difference between both patterns are responsibilities they bear. [Decorators](https://howtodoinjava.com/design-patterns/structural/decorator-design-pattern/) focus on adding responsibilities, but proxies focus on controlling the access to an object.

Drop me your questions related to **proxy pattern** in comments.

# Flyweight Design Pattern

As per GoF definition, **flyweight design pattern** enables use sharing of objects to support large numbers of fine-grained objects efficiently. A flyweight is a **shared object** that can be used in multiple contexts simultaneously. The flyweight acts as an independent object in each context.

## 1. When to use flyweight design pattern

We can use flyweight pattern in following scenarios:

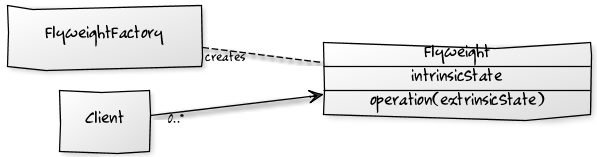
* When we need a large number of similar objects that are unique in terms of only a few parameters and most of the stuffs are common in general.
* We need to control the memory consumption by large number of objects – by creating fewer objects and sharing them across.

## 2. Extrinsic and intrinsic attributes

A flyweight objects essentially has two kind of attributes – intrinsic and extrinsic.

An **intrinsic** state attribute is stored/shared in the flyweight object, and it is independent of flyweight’s context. As the best practice, we should make intrinsic states [**immutable**](https://howtodoinjava.com/java/basics/how-to-make-a-java-class-immutable/).

An **extrinsic** state varies with flyweight’s context, which is why they cannot be shared. Client objects maintain the extrinsic state, and they need to pass this to a flyweight object during object creation.

Flyweight Pattern

## 3. Real world example of flyweight pattern

* Suppose we have a **pen** which can exist with/without **refill**. A refill can be of any color thus a pen can be used to create drawings having N number of colors.

Here Pen can be flyweight object with refill as extrinsic attribute. All other attributes such as pen body, pointer etc. can be intrinsic attributes which will be common to all pens. A pen will be distinguished by its refill color only, nothing else.

All application modules which need to access a red pen – can use the same instance of red pen (shared object). Only when a different color pen is needed, application module will ask for another pen from flyweight factory.

* In programming, we can see **java.lang.String** constants as flyweight objects. All strings are stored in string pool and if we need a string with certain content then runtime return the reference to already existing string constant from the pool – if available.
* In browsers, we can use an image in multiple places in a webpage. Browsers will load the image only one time, and for other times browsers will reuse the image from cache. Now image is same but used in multiple places. It’s URL is intrinsic attribute because it’s fixed and shareable. Images position coordinates, height and width are extrinsic attributes which vary according to place (context) where they have to be rendered.

## 4. Flyweight design pattern example

In given example, we are building a Paint Brush application where client can use brushes on three types – THICK, THIN and MEDIUM. All the thick (thin or medium) brush will draw the content in exact similar fashion – only the content color will be different.

|  |
| --- |
| **Pen.java** |
| **public interface Pen**  **{**  **public void setColor(String color);**  **public void draw(String content);**  **}** |
| **BrushSize.java** |
| **public enum BrushSize {**  **THIN, MEDIUM, THICK**  **}** |

|  |
| --- |
| **ThickPen.java** |
| **public class ThickPen implements Pen {**    **final BrushSize brushSize = BrushSize.THICK;    //intrinsic state - shareable**  **private String color = null;                    //extrinsic state - supplied by client**    **public void setColor(String color) {**  **this.color = color;**  **}**    **@Override**  **public void draw(String content) {**  **System.out.println("Drawing THICK content in color : " + color);**  **}**  **}** |
| **ThinPen.java** |
| **public class ThinPen implements Pen {**    **final BrushSize brushSize = BrushSize.THIN;**  **private String color = null;**    **public void setColor(String color) {**  **this.color = color;**  **}**    **@Override**  **public void draw(String content) {**  **System.out.println("Drawing THIN content in color : " + color);**  **}**  **}** |

|  |
| --- |
| **MediumPen.java** |
| **public class MediumPen implements Pen {**    **final BrushSize brushSize = BrushSize.MEDIUM;**  **private String color = null;**    **public void setColor(String color) {**  **this.color = color;**  **}**    **@Override**  **public void draw(String content) {**  **System.out.println("Drawing MEDIUM content in color : " + color);**  **}**  **}** |

Here brush color is extrinsic attribute which will be supplied by client, else everything will remain same for the Pen. So essentially, we will create a pen of certain size only when the color is different. Once another client or context need that pen size and color, we will reuse it.

|  |
| --- |
| **PenFactory.java** |
| **import java.util.HashMap;**    **public class PenFactory**  **{**  **private static final HashMap<String, Pen> pensMap = new HashMap<>();**    **public static Pen getThickPen(String color)**  **{**  **String key = color + "-THICK";**    **Pen pen = pensMap.get(key);**    **if(pen != null) {**  **return pen;**  **} else {**  **pen = new ThickPen();**  **pen.setColor(color);**  **pensMap.put(key, pen);**  **}**    **return pen;**  **}**    **public static Pen getThinPen(String color)**  **{**  **String key = color + "-THIN";**    **Pen pen = pensMap.get(key);**    **if(pen != null) {**  **return pen;**  **} else {**  **pen = new ThinPen();**  **pen.setColor(color);**  **pensMap.put(key, pen);**  **}**    **return pen;**  **}**    **public static Pen getMediumPen(String color)**  **{**  **String key = color + "-MEDIUM";**    **Pen pen = pensMap.get(key);**    **if(pen != null) {**  **return pen;**  **} else {**  **pen = new MediumPen();**  **pen.setColor(color);**  **pensMap.put(key, pen);**  **}**    **return pen;**  **}**  **}** |

Let’s test the flyweight pen objects using a client. The client here creates three THIN pens, but in runtime their is only one pen object of thin type and it’s shared with all three invocations.

|  |
| --- |
| **PaintBrushClient.java** |
| **public class PaintBrushClient**  **{**  **public static void main(String[] args)**  **{**  **Pen yellowThinPen1 = PenFactory.getThickPen("YELLOW");  //created new pen**  **yellowThinPen1.draw("Hello World !!");**    **Pen yellowThinPen2 = PenFactory.getThickPen("YELLOW");  //pen is shared**  **yellowThinPen2.draw("Hello World !!");**    **Pen blueThinPen = PenFactory.getThickPen("BLUE");       //created new pen**  **blueThinPen.draw("Hello World !!");**    **System.out.println(yellowThinPen1.hashCode());**  **System.out.println(yellowThinPen2.hashCode());**    **System.out.println(blueThinPen.hashCode());**  **}**  **}** |

Program output.

|  |
| --- |
| **Console** |
| **Drawing THICK content in color : YELLOW**  **Drawing THICK content in color : YELLOW**  **Drawing THICK content in color : BLUE**    **2018699554      //same object**  **2018699554      //same object**  **1311053135** |

## 5. FAQs

#### 5.1. Difference between singleton pattern and flyweight pattern

The singleton pattern helps we maintain only one object in the system. In other words, once the required object is created, we cannot create more. We need to reuse the existing object in all parts of the application.

The flyweight pattern is used when we have to create large number of similar objects which are different based on client provided extrinsic attribute.

#### 5.2. Effect of concurrency on flyweights

Similar to singleton pattern, if we create flyweight objects in concurrent environment, we may end up having multiple instances of same flyweight object which is not desirable.

To fix this, we need to use **double checked locking** as used in [singleton pattern](https://howtodoinjava.com/design-patterns/creational/singleton-design-pattern-in-java/) while creating flyweights.

#### 5.3. Benefits of flyweight design pattern

Using flyweights, we can –

* reduce memory consumption of heavy objects that can be controlled identically.
* reduce the total number of “complete but similar objects” in the system.
* provide a centralized mechanism to control the states of many “virtual” objects.

#### 5.4. Is intrinsic and extrinsic data shareable?

The intrinsic data is shareable as it is common to all contexts. The extrinsic data is not shared. Client need to pass the information (states) to the flyweights which is unique to it’s context.

#### 5.5. Challenges of flyweight pattern

* We need to take the time to configure these flyweights. The design time and skills can be overhead, initially.
* To create flyweights, we extract a common template class from the existing objects. This additional layer of programming can be tricky and sometimes hard to debug and maintain.
* The flyweight pattern is often combined with singleton factory implementation and to guard the singularity, additional cost is required.

# Composite Design Pattern

**Composite design pattern** is a **structural pattern** which modifies the structure of an object. This pattern is most suitable in cases where you need to work with **objects which form a tree like hierarchy**. In that tree, each node/object (except root node) is either composite or leaf node. Implementing the composite pattern lets clients treat individual objects and compositions uniformly.

## When to use composite design pattern

Composite design pattern compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

* When application has hierarchical structure and needs generic functionality across the structure.
* When application needs to aggregate data across a hierarchy.
* When application wants to treat composite and individual objects uniformly.

**Real life example usage of composite design pattern** may be:

1. Building consolidate view of a customer’s account in bank (i.e. customer’s portfolio)
2. Building general ledgers
3. Computer/network monitoring applications
4. Retail and inventory applications
5. Directory structure in file system implementations
6. Menu items in GUI screens

## Final notes

1. The composite pattern defines class hierarchies consisting of individual objects and composite objects.
2. Clients treat primitive and composite objects uniformly through a component interface which makes client code simple.
3. Adding new components can be easy and client code does not need to be changed since client deals with the new components through the component interface.
4. Composite hierarchy can be traversed with Iterator design pattern.
5. Visitor design pattern can apply an operation over a Composite.
6. Flyweight design pattern is often combined with Composite to implement shared leaf nodes.

# What are SOLID Design Pattern in Java?

First, we'll start by **exploring the reasons they came about and why we should consider them** when designing software. Then we'll outline each principle alongside some example code.

## **2. The Reason for SOLID Principles**

The SOLID principles were introduced by Robert C. Martin in his 2000 paper [“Design Principles and Design Patterns.”](https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf) These concepts were later built upon by Michael Feathers, who introduced us to the SOLID acronym. And in the last 20 years, these five principles have revolutionized the world of object-oriented programming, changing the way that we write software.

So, what is SOLID and how does it help us write better code? Simply put, Martin and Feathers' **design principles encourage us to create more maintainable, understandable, and flexible software.** Consequently,**as our applications grow in size, we can reduce their complexity** and save ourselves a lot of headaches further down the road!

The following five concepts make up our SOLID principles:

1. **S**ingle Responsibility
2. **O**pen/Closed
3. **L**iskov Substitution
4. **I**nterface Segregation
5. **D**ependency Inversion

While these concepts may seem daunting, they can be easily understood with some simple code examples. In the following sections, we'll take a deep dive into these principles, with a quick Java example to illustrate each one.

## **3. Single Responsibility**

Let's begin with the single responsibility principle. As we might expect, this principle states that **a class should only have one responsibility. Furthermore, it should only have one reason to change.**

**How does this principle help us to build better software?** Let's see a few of its benefits:

1. **Testing** – A class with one responsibility will have far fewer test cases.
2. **Lower coupling** – Less functionality in a single class will have fewer dependencies.
3. **Organization** – Smaller, well-organized classes are easier to search than monolithic ones.

For example, let's look at a class to represent a simple book:

**public** **class** **Book** {

**private** String name;

**private** String author;

**private** String text;

//constructor, getters and setters

}

In this code, we store the name, author and text associated with an instance of a Book.

Let's now add a couple of methods to query the text:

**public** **class** **Book** {

**private** String name;

**private** String author;

**private** String text;

//constructor, getters and setters

// methods that directly relate to the book properties

**public** String **replaceWordInText**(String word){

**return** text.replaceAll(word, text);

}

**public** **boolean** **isWordInText**(String word){

**return** text.contains(word);

}

}

Now our Book class works well, and we can store as many books as we like in our application.

But what good is storing the information if we can't output the text to our console and read it?

Let's throw caution to the wind and add a print method:

**public** **class** **Book** {

//...

**void** **printTextToConsole**(){

// our code for formatting and printing the text

}

}

However, this code violates the single responsibility principle we outlined earlier.

To fix our mess, we should implement a separate class that deals only with printing our texts:

**public** **class** **BookPrinter** {

// methods for outputting text

**void** **printTextToConsole**(String text){

//our code for formatting and printing the text

}

**void** **printTextToAnotherMedium**(String text){

// code for writing to any other location..

}

}

Awesome. Not only have we developed a class that relieves the Book of its printing duties, but we can also leverage our BookPrinter class to send our text to other media.

Whether it's email, logging, or anything else, we have a separate class dedicated to this one concern.

## **4. Open for Extension, Closed for Modification**

It's now time for the O in SOLID, known as the **open-closed principle.** Simply put, **classes should be open for extension but closed for modification.** **In doing so, we** **stop ourselves from modifying existing code and causing potential new bugs** in an otherwise happy application.

Of course, the **one exception to the rule is when fixing bugs in existing code.**

Let's explore the concept with a quick code example. As part of a new project, imagine we've implemented a Guitar class.

It's fully fledged and even has a volume knob:

**public** **class** **Guitar** {

**private** String make;

**private** String model;

**private** **int** volume;

//Constructors, getters & setters

}

We launch the application, and everyone loves it. But after a few months, we decide the Guitar is a little boring and could use a cool flame pattern to make it look more rock and roll.

At this point, it might be tempting to just open up the Guitar class and add a flame pattern — but who knows what errors that might throw up in our application.

Instead, let's **stick to the open-closed principle and simply extend our Guitar class**:

**public** **class** **SuperCoolGuitarWithFlames** **extends** **Guitar** {

**private** String flameColor;

//constructor, getters + setters

}

By extending the Guitar class, we can be sure that our existing application won't be affected.

## **5. Liskov Substitution**

Next on our list is [Liskov substitution](https://www.baeldung.com/cs/liskov-substitution-principle), which is arguably the most complex of the five principles. Simply put, **if class A is a subtype of class B, we should be able to replace B with A without disrupting the behavior of our program.**

Let's jump straight to the code to help us understand this concept:

**public** **interface** **Car** {

**void** **turnOnEngine**();

**void** **accelerate**();

}

Above, we define a simple Car interface with a couple of methods that all cars should be able to fulfill: turning on the engine and accelerating forward.

Let's implement our interface and provide some code for the methods:

**public** **class** **MotorCar** **implements** **Car** {

**private** Engine engine;

//Constructors, getters + setters

**public** **void** **turnOnEngine**() {

//turn on the engine!

engine.on();

}

**public** **void** **accelerate**() {

//move forward!

engine.powerOn(1000);

}

}

As our code describes, we have an engine that we can turn on, and we can increase the power.

But wait — we are now living in the era of electric cars:

**public** **class** **ElectricCar** **implements** **Car** {

**public** **void** **turnOnEngine**() {

**throw** **new** AssertionError("I don't have an engine!");

}

**public** **void** **accelerate**() {

//this acceleration is crazy!

}

}

By throwing a car without an engine into the mix, we are inherently changing the behavior of our program. This is**a blatant violation of Liskov substitution and is a bit harder to fix than our previous two principles.**

One possible solution would be to rework our model into interfaces that take into account the engine-less state of our Car.

## **6. Interface Segregation**

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.**

For this example, we're going to try our hands as zookeepers. And more specifically, we'll be working in the bear enclosure.

Let's start with an interface that outlines our roles as a bear keeper:

**public** **interface** **BearKeeper** {

**void** **washTheBear**();

**void** **feedTheBear**();

**void** **petTheBear**();

}

As avid zookeepers, we're more than happy to wash and feed our beloved bears. But we're all too aware of the dangers of petting them. Unfortunately, our interface is rather large, and we have no choice but to implement the code to pet the bear.

Let's **fix this by splitting our large interface into three separate ones**:

**public** **interface** **BearCleaner** {

**void** **washTheBear**();

}

**public** **interface** **BearFeeder** {

**void** **feedTheBear**();

}

**public** **interface** **BearPetter** {

**void** **petTheBear**();

}

Now, thanks to interface segregation, we're free to implement only the methods that matter to us:

**public** **class** **BearCarer** **implements** **BearCleaner**, **BearFeeder** {

**public** **void** **washTheBear**() {

//I think we missed a spot...

}

**public** **void** **feedTheBear**() {

//Tuna Tuesdays...

}

}

And finally, we can leave the dangerous stuff to the reckless people:

**public** **class** **CrazyPerson** **implements** **BearPetter** {

**public** **void** **petTheBear**() {

//Good luck with that!

}

}

Going further, we could even split our *[BookPrinter](https://www.baeldung.com/solid-principles" \l "s)* class from our example earlier to use interface segregation in the same way. By implementing a Printer interface with a single print method, we could instantiate separate ConsoleBookPrinter and OtherMediaBookPrinter classes.

## **7. Dependency Inversion**

**The principle of dependency inversion refers to the decoupling of software modules. This way, instead of high-level modules depending on low-level modules, both will depend on abstractions.**

To demonstrate this, let's go old-school and bring to life a Windows 98 computer with code:

**public** **class** **Windows98Machine** {}

But what good is a computer without a monitor and keyboard? Let's add one of each to our constructor so that every Windows98Computer we instantiate comes prepacked with a Monitor and a StandardKeyboard:

**public** **class** **Windows98Machine** {

**private** **final** StandardKeyboard keyboard;

**private** **final** Monitor monitor;

**public** **Windows98Machine**() {

monitor = **new** Monitor();

keyboard = **new** StandardKeyboard();

}

}

This code will work, and we'll be able to use the StandardKeyboard and Monitor freely within our Windows98Computer class.

Problem solved? Not quite. **By declaring the StandardKeyboard and Monitor with the new keyword, we've tightly coupled these three classes together.**

Not only does this make our Windows98Computer hard to test, but we've also lost the ability to switch out our StandardKeyboard class with a different one should the need arise. And we're stuck with our Monitor class too.

Let's decouple our machine from the StandardKeyboard by adding a more general Keyboard interface and using this in our class:

**public** **interface** **Keyboard** { }

**public** **class** **Windows98Machine**{

**private** **final** Keyboard keyboard;

**private** **final** Monitor monitor;

**public** **Windows98Machine**(Keyboard keyboard, Monitor monitor) {

**this**.keyboard = keyboard;

**this**.monitor = monitor;

}

}

Here, we're using the dependency injection pattern to facilitate adding the Keyboard dependency into the Windows98Machine class.

Let's also modify our StandardKeyboard class to implement the Keyboard interface so that it's suitable for injecting into the Windows98Machine class:

**public** **class** **StandardKeyboard** **implements** **Keyboard** { }

Now our classes are decoupled and communicate through the Keyboard abstraction. If we want, we can easily switch out the type of keyboard in our machine with a different implementation of the interface. We can follow the same principle for the Monitor class.

Excellent! We've decoupled the dependencies and are free to test our Windows98Machine with whichever testing framework we choose.

## **8. Conclusion**

In this article, we've taken a **deep dive into the SOLID principles of object-oriented design.**

We **started with a quick bit of SOLID history and the reasons these principles exist.**

Letter by letter, we've **broken down the meaning of each principle with a quick code example that violates it. We then saw how to fix our code**and make it adhere to the SOLID principles.