DESIGN PATTERNS

Design Patterns are very popular among software developers. A design pattern is a well described solution to a common software problem.

Benefits of using design patterns are:

1. Design Patterns are already defined and provides **industry standard approach** to solve a recurring problem, so it saves time if we sensibly use the design pattern. There are many java design patterns that we can use in our java based projects.
2. Using design patterns promotes **reusability** that leads to more **robust** and highly maintainable code. It helps in reducing total cost of ownership (TCO) of the software product.
3. Since design patterns are already defined, it makes our code easy to understand and debug. It leads to faster development and new members of team understand it easily.

Java Design Patterns are divided into three categories – **creational**, **structural**, and **behavioral** design patterns. This post serves as an index for all the java design patterns articles I have written so far.

I. Creational Design Patterns

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

II. Structural Design Patterns

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

III. Behavioral Design Patterns

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

II. Structural Design Patterns:

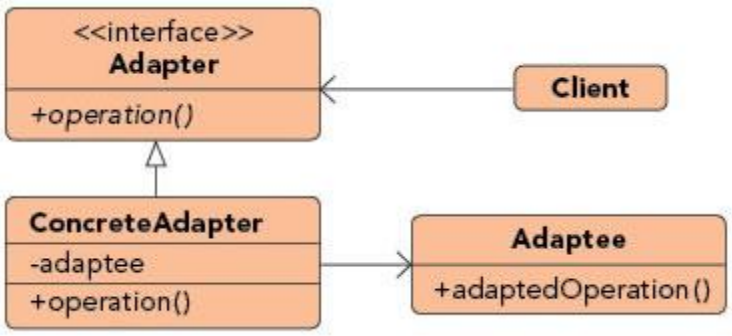
- Structural Patterns provide different ways to create a class structure, for example using inheritance and composition to create a large object from small objects.
- These patterns deal with the composition of classes and objects. They provide different ways of using object composition and inheritance to create some abstraction.

1. Adapter Pattern:

- **Purpose**: Permits classes with disparate (dissimilar) interfaces to work together by creating a common object by which they may communicate and interact.
- Adapter pattern works as a bridge between two incompatible interfaces.
- An Adapter pattern acts as a connector between two incompatible interfaces that otherwise cannot be connected directly.
- This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces.
- The object that joins these unrelated interfaces is called Adapter.
- An Adapter wraps an existing class with a new interface so that it becomes compatible with client's interface.
- The main motive behind using this pattern is to convert an existing interface into another interface that client expects. It's usually implemented once the application is designed.

Adapter Design Pattern Example in JDK

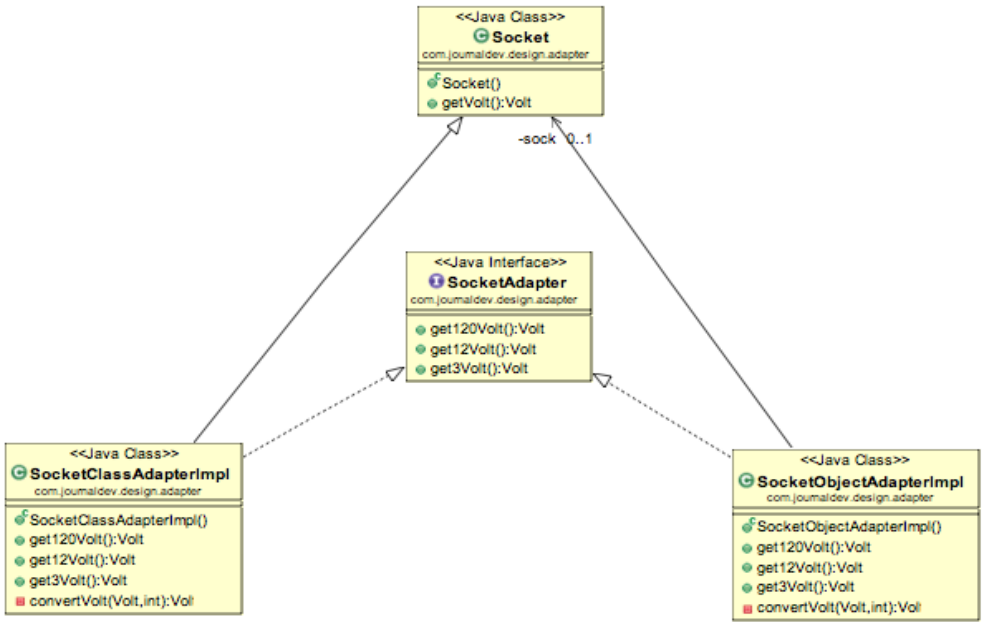
- java.util.Arrays#asList()
- java.io.InputStreamReader(InputStream) (returns a Reader)
- java.io.OutputStreamWriter(OutputStream) (returns a Writer)



Real Life Example:

A billing application needs to interface with an HR application in order to exchange employee data, however each has its own interface and implementation for the Employee object. In addition, the SSN is stored in different formats by each system. By creating an adapter we can create a common interface between the two applications that allows them to communicate using their native objects and is able to transform the SSN format in the process.

Example:
As a real life example, we can think of a mobile charger as an adapter because mobile battery needs 3 Volts to charge but the normal socket produces either 120V (US) or 240V (India).
So the mobile charger works as an adapter between mobile charging socket and the wall socket.



We'll have two classes: Volt – (to measure volts) and Socket (producing constant volts of 120V):

```
public class Volt {

    private int volts;

    public Volt(int v) {
          this.volts=v;
    }

    public int getVolts() {
          return volts;
    }

    public void setVolts(int volts) {
          this.volts = volts;
    }
}


public class Socket {

    public Volt getVolt() {
          return new Volt(120);
    }
}
```

Now we want to build an adapter that can produce 3 volts, 12 volts and default 120 volts. So first of all we will create an adapter interface with these methods.

```
public interface SocketAdapter {
    public Volt get120Volt();
    public Volt get12Volt();
    public Volt get3Volt();
}
```

Two Way Adapter Pattern:

While implementing Adapter pattern, there are two approaches – class adapter and object adapter – however both these approaches produce same result.

1. **Class Adapter** – This form uses **java inheritance** and extends the source interface, in our case Socket class.
2. **Object Adapter** – This form uses **java composition** and adapter contains the source object.

Adapter Design Pattern – Class Adapter

Here is the **class adapter** approach implementation of our adapter.

```
//Using inheritance for adapter pattern
public class SocketClassAdapterImpl extends Socket implements
                                                        SocketAdapter{

    @Override
    public Volt get120Volt() {
          return getVolt ();
    }

    @Override
    public Volt get12Volt() {
          Volt v = getVolt();
          return convertVolt(v,10);
    }

    @Override
    public Volt get3Volt() {
          Volt v = getVolt();
          return convertVolt(v,40);
    }

    private Volt convertVolt(Volt v, int i) {
          return new Volt(v.getVolts()/i);
    }
}
```

Adapter Design Pattern – Object Adapter Implementation
Here is the **Object adapter** implementation of our adapter.

```
public class SocketObjectAdapterImpl implements SocketAdapter {

   //Using Composition for adapter pattern
   private Socket sock = new Socket();

   @Override
   public Volt get120Volt() {
         return sock.getVolt();
   }

   @Override
   public Volt get12Volt() {
         Volt v = sock.getVolt();
         return convertVolt(v,10);
   }

   @Override
   public Volt get3Volt() {
         Volt v = sock.getVolt();
         return convertVolt(v,40);
   }

   private Volt convertVolt(Volt v, int i) {
         return new Volt(v.getVolts()/i);
   }
}
```

Notice that both the adapter implementations are almost same and they implement the SocketAdapter interface. The adapter interface can also be an **abstract class**.

Program to consume our adapter design pattern implementation:

```
public class AdapterPattern {

   public static void main(String[] args) {

         testClassAdapter();
         testObjectAdapter();
   }

   private static void testObjectAdapter() {
         SocketAdapter sockAdapter = new SocketObjectAdapterImpl();
         Volt v3 = getVolt(sockAdapter,3);
         Volt v12 = getVolt(sockAdapter,12);
         Volt v120 = getVolt(sockAdapter,120);

         System.out.println("v3 volts using Object Adapter = " +
                     v3.getVolts());
         System.out.println("v12 volts using Object Adapter = " +
                     v12.getVolts());
         System.out.println("v120 volts using Object Adapter = " +
                     v120.getVolts());
   }
```

```
    private static void testClassAdapter() {
          SocketAdapter sockAdapter = new SocketClassAdapterImpl();

          Volt v3 = getVolt(sockAdapter,3);
          Volt v12 = getVolt(sockAdapter,12);
          Volt v120 = getVolt(sockAdapter,120);
          System.out.println("v3 volts using Class Adapter = " +
                       v3.getVolts());
          System.out.println("v12 volts using Class Adapter = " +
                       v12.getVolts());
          System.out.println("v120 volts using Class Adapter = " +
                       v120.getVolts());
    }

    private static Volt getVolt(SocketAdapter sockAdapter, int i) {
          switch (i) {

          case 3: return sockAdapter.get3Volt();
          case 12: return sockAdapter.get12Volt();
          case 120: return sockAdapter.get120Volt();
          default: return sockAdapter.get120Volt();
          }
    }
}
```

When to use Adapter Pattern:

- A class to be used doesn't meet interface requirements.
- Complex conditions tie object behavior to its state.
- Transitions between states need to be explicit.
- Composition, creation, and representation of objects should be decoupled from a system.
- Classes to be created are specified at runtime.
- A limited number of state combinations exist in an object.
- Objects or object structures are required that are identical or closely resemble other existing objects or object structures.
- The initial creation of each object is an expensive operation.

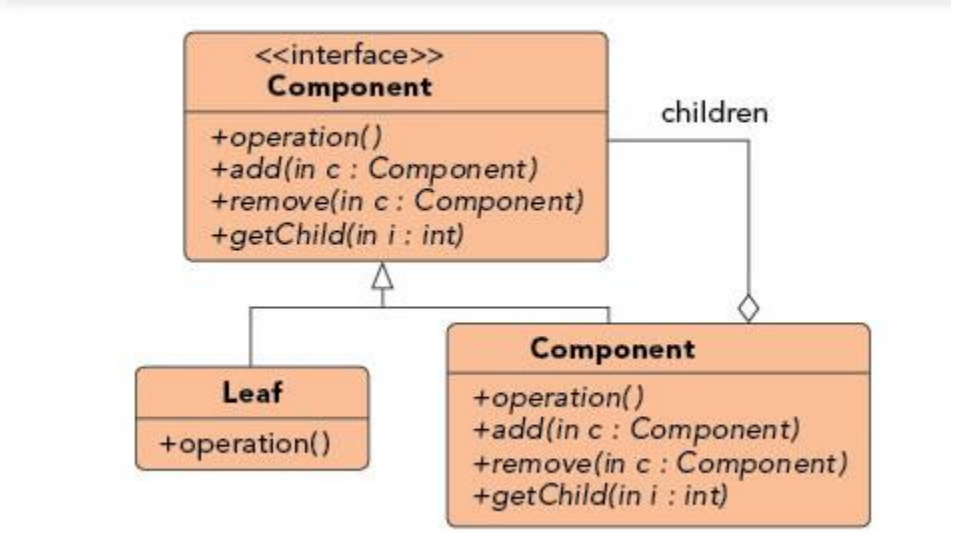
2. Composite Pattern:

- **Purpose:** Facilitates the creation of object hierarchies where each object can be treated independently or as a set of nested objects through the same interface.
- Composite pattern is used when we have to represent a part-whole hierarchy.
- Composite pattern is used where we need to treat a group of objects in similar way as a single object.
- Composite pattern composes objects in term of a tree structure to represent part as well as whole hierarchy of objects.
- This pattern creates a class that contains group of its own objects. This class provides ways to modify its group of same objects.

  We can break the pattern down into:
  - component – is the base interface for all the objects in the composition. It should be either an interface or an abstract class with the common methods to manage the child composites.
  - leaf – implements the default behavior of the base component. It doesn't contain a reference to the other objects.
  - composite – has leaf elements. It implements the base component methods and defines the child-related operations.
  - client – has access to the composition elements by using the base component object.

Composite Design Pattern Example in JDK:

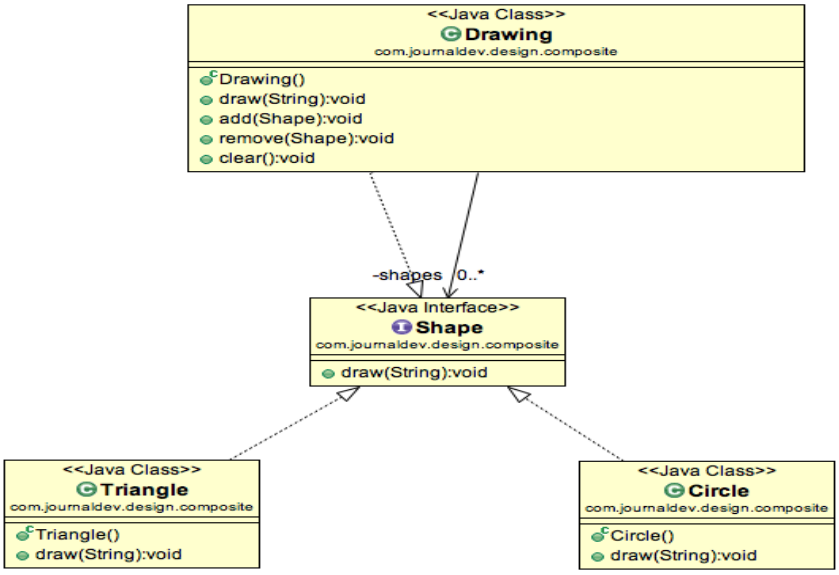
- java.awt.Container#add(Component) is a great example of Composite pattern in java and used a lot in Swing.



Real Life Example:

Sometimes the information displayed in a shopping cart is the product of a single item while other times it is an aggregation of multiple items. By implementing items as composites we can treat the aggregates and the items in the same way, allowing us to simply iterate over the tree and invoke functionality on each item. By calling the getCost() method on any given node we would get the cost of that item plus the cost of all child items, allowing items to be uniformly treated whether they were single items or groups of items.

Example:



The Base Component:

Composite pattern base component defines the common methods for leaf and composites. We can create a class **Shape** with a method draw(String fillColor) to draw the shape with given color.

```
public interface Shape {

	public void draw(String fillColor);
}
```

Leafs:

These are the building block for the composite. We can create multiple leaf objects such as Triangle, Circle etc.

```
public class Triangle implements Shape {

	@Override
	public void draw(String fillColor) {
		System.out.println("Drawing Triangle with color " + fillColor);
	}

}


public class Circle implements Shape {

	@Override
	public void draw(String fillColor) {
		System.out.println("Drawing Circle with color " + fillColor);
	}
}
```

Composite object:
A composite object contains group of leaf objects and we should provide some helper methods to add or delete leafs from the group. We can also provide a method to remove all the elements from the group.

```
public class Drawing implements Shape {

	//collection of Shapes
	private List<Shape> shapes = new ArrayList<>();

	@Override
	public void draw(String fillColor) {
		shapes.forEach(shape -> shape.draw(fillColor));
	}

	//adding shape to drawing
	public void add(Shape shape) {
		this.shapes.add(shape);
	}

	//removing shape from drawing
	public void remove(Shape shape) {
		shapes.remove(shape);
	}

	//removing all the shapes
	public void clear() {
		System.out.println("Clearing all the shapes from drawing");
		this.shapes.clear();
	}
}
```

Note: Composite also implements component and behaves similar to leaf except that it can contain group of leaf elements.

Composite Design Pattern Program:

```
public class TestComposite {

	public static void main(String[] args) {
		Shape tri = new Triangle();
		Shape tri1 = new Triangle();
		Shape cir = new Circle();

		Drawing drawing = new Drawing();
		drawing.add(tri1);
		drawing.add(tri1);
		drawing.add(cir);

		drawing.draw("Red");
		drawing.clear();
```

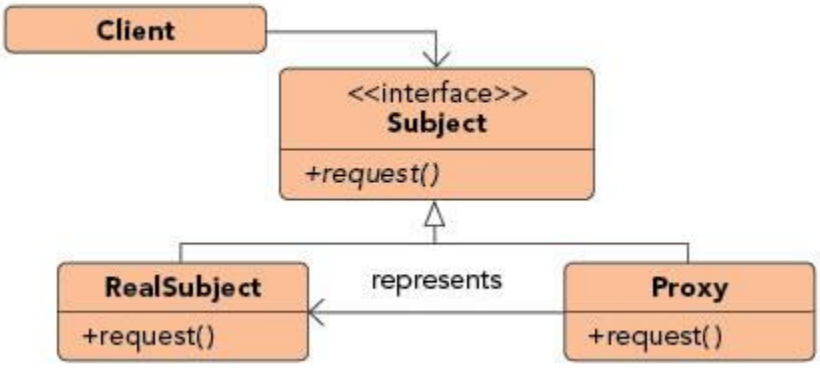
```
            drawing.add(tri);
            drawing.add(cir);
            drawing.draw("Green");
        }
}
```

When to use Composite Pattern:

- Composite pattern should be applied only when the group of objects should behave as the single object.
- Composite design pattern can be used to create a tree like structure.
- Hierarchical representations of objects are needed.
- Objects and compositions of objects should be treated uniformly.

3. Proxy Pattern:

- **Purpose:** Allows for object level access control by acting as a pass through entity or a placeholder object.
- In proxy pattern, a class represents functionality of another class.
- Proxy pattern intent is to "Provide a surrogate or placeholder for another object to control access to it".
- Proxy pattern is used when we want to provide controlled access of a functionality.
- With this pattern, we create an intermediary that acts as an interface to another resource, **e.g., a file, a connection.** This secondary access provides a surrogate for the real component and protects it from the underlying complexity.

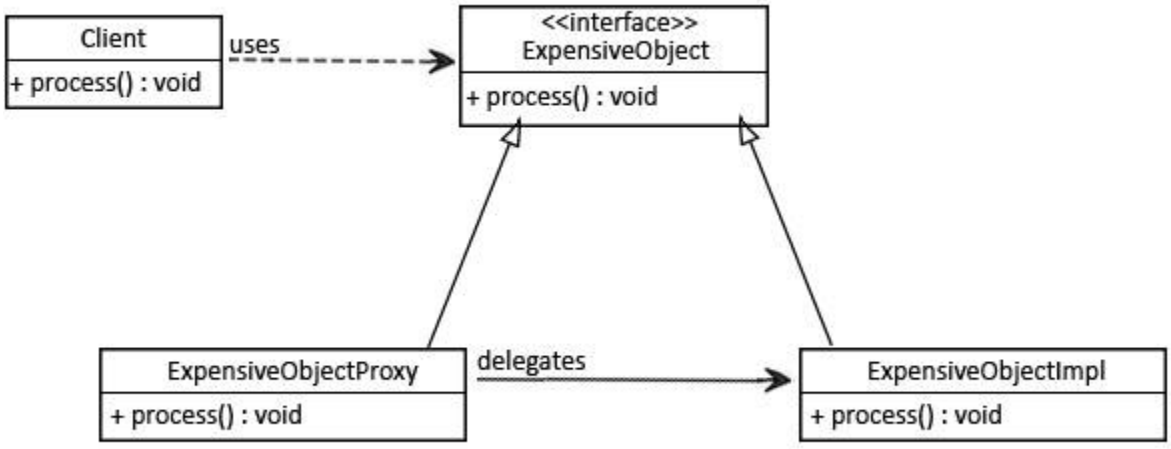


Real Life Example:

Ledger applications often provide a way for users to reconcile their bank statements with their ledger data on demand, automating much of the process. The actual operation of communicating with a third party is a relatively expensive operation that should be limited. By using a proxy to represent the communications object we can limit the number of times or the intervals the communication is invoked. In addition, we can wrap the complex instantiation of the communication object inside the proxy class, decoupling calling code from the implementation details.

Another example can be a cheque or credit card is a proxy for what is in our bank account. It can be used in place of cash, and provides a means of accessing that cash when required. And that's exactly what the Proxy pattern does – "**Controls and manage access to the object they are protecting**".

Example:
Consider a heavy Java object (like a JDBC connection or a *SessionFactory*) that requires some initial configuration.
We only want such objects to be initialized on demand, and once they are, we'd want to reuse them for all calls:



Create a simple interface and the configuration for this object:

```
public interface ExpensiveObject {
    void process();
}
```

The implementation of this interface with a large initial configuration:

```
public class ExpensiveObjectImpl implements ExpensiveObject {

    public ExpensiveObjectImpl() {
          heavyInitialConfiguration();
    }

    @Override
    public void process() {
          System.out.println("processing complete.");
    }

    private void heavyInitialConfiguration() {
          System.out.println("Loading initial configuration...");
    }
}
```

Utilize the Proxy pattern and initialize our object on demand:

```
public class ExpensiveObjectProxy implements ExpensiveObject {
    private static ExpensiveObject object;
```

```
    @Override
    public void process() {
          if (object == null) {
                object = new ExpensiveObjectImpl();
          }
          object.process();
    }
}
```

Whenever our client calls the *process()* method, they'll just get to see the processing and the initial configuration will always remain hidden:

```
public class TestProxy {
    public static void main(String[] args) {
          ExpensiveObject object = new ExpensiveObjectProxy();
          object.process();
          object.process();
    }
}
Output:
Loading initial configuration...
processing complete.
processing complete.
```

Note that we're calling the *process()* method twice. Behind the scenes, the settings part will occur only once – when the object is first initialized.

For every other subsequent call, this pattern will skip the initial configuration, and only processing will occur.

When to Use Proxy Pattern:

Proxy pattern is used when we need to create a wrapper to cover the main object's complexity from the client.

- **When we want a simplified version of a complex or heavy object.**
  In this case, we may represent it with a skeleton object which loads the original object on demand, also called as lazy initialization. This is known as the **Virtual Proxy**.

- **When the original object is present in different address space, and we want to represent it locally**.
  We can create a proxy which does all the necessary boilerplate stuff like creating and maintaining the connection, encoding, decoding, etc., while the client accesses it as it was present in their local address space. This is called the **Remote Proxy.**

- **When we want to add a layer of security to the original underlying object to provide controlled access based on access rights of the client**. This is called **Protection Proxy**.
- A **smart proxy** provides additional layer of security by interposing specific actions when the object is accessed. 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.

4. Flyweight Pattern:

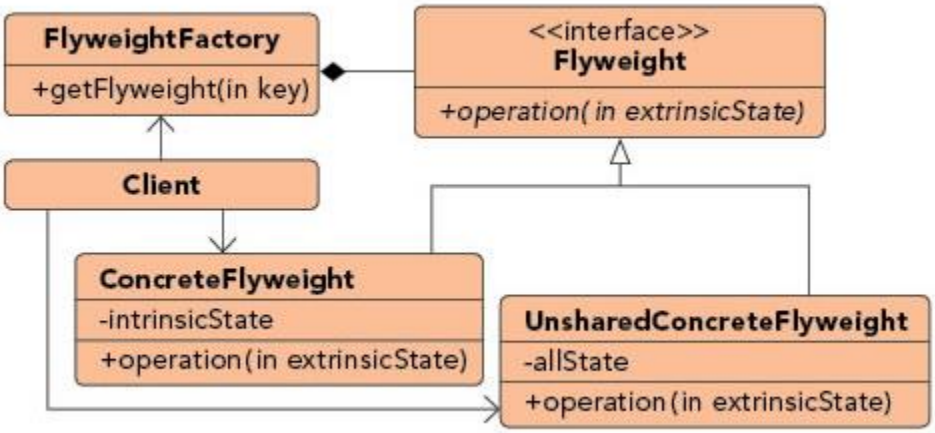
- **Purpose:** Facilitates the reuse of many fine grained objects, making the utilization of large numbers of objects more efficient.
- Flyweight pattern is primarily used to reduce the number of objects created and to decrease memory footprint and increase performance. This type of design pattern comes under structural pattern as this pattern provides ways to decrease object count thus improving the object structure of application.
- Flyweight pattern tries to reuse already existing similar kind objects by storing them and creates new object when no matching object is found.
- Flyweight design pattern is used when we need to create a lot of Objects of a class. Since every object consumes memory space that can be crucial for low memory devices, such as mobile devices or embedded systems, flyweight design pattern can be applied to reduce the load on memory by sharing objects.
- This pattern is used to reduce the memory footprint. It can also improve performance in applications where object instantiation is expensive.
- Simply put, the flyweight pattern is based on a factory which recycles created objects by storing them after creation. Each time an object is requested, the factory looks up the object in order to check if it's already been created. If it has, the existing object is returned – otherwise, a new one is created, stored and then returned.
- The flyweight object's state is made up of an invariant component shared with other similar objects (intrinsic) and a variant component which can be manipulated by the client code (extrinsic).
- It's very important that the flyweight objects are immutable: any operation on the state must be performed by the factory.

Flyweight Design Pattern Example in JDK:

- All the wrapper classes **valueOf()** method uses cached objects showing use of Flyweight design pattern.
- The best example is Java String class String Pool implementation.

Real Life Example:

Systems that allow users to define their own application flows and layouts often have a need to keep track of large numbers of fields, pages, and other items that are almost identical to each other. By making these items into flyweights all instances of each object can share the intrinsic state while keeping the extrinsic state separate. The intrinsic state would store the shared properties, such as how a textbox looks, how much data it can hold, and what events it exposes. The extrinsic state would store the unshared properties, such as where the item belongs, how to react to a user click, and how to handle events.



- Before we apply flyweight design pattern, we need to consider following factors:

    1. The number of Objects to be created by application should be huge.
    2. The object creation is heavy on memory and it can be time consuming too.
    3. The object properties can be divided into intrinsic and extrinsic properties, extrinsic properties of an Object should be defined by the client program.

- To apply flyweight pattern, we need to divide Object property into **intrinsic** and **extrinsic** properties. Intrinsic properties make the Object unique whereas extrinsic properties are set by client code and used to perform different operations. For example, an Object Circle can have extrinsic properties such as color and width.

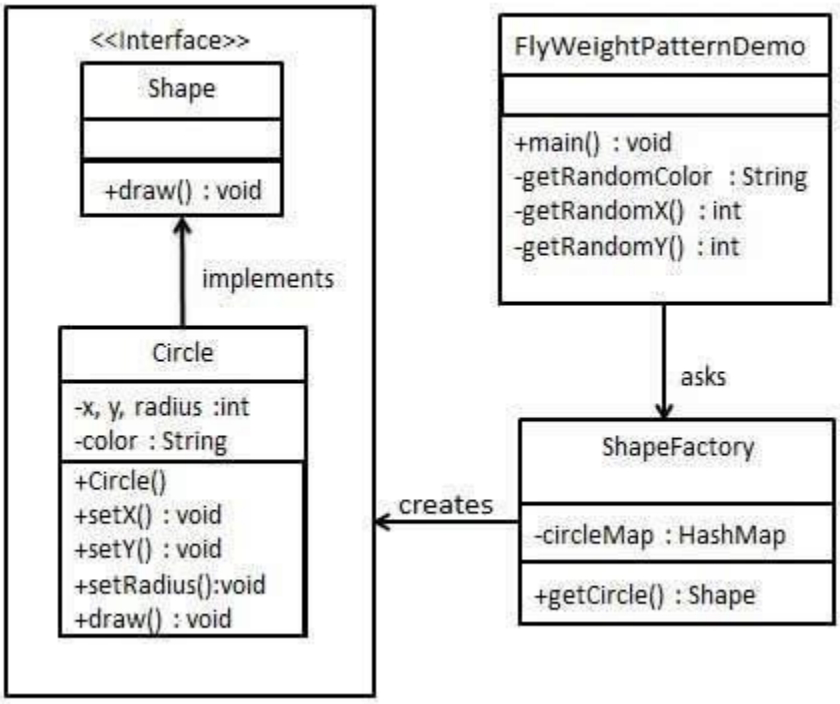
Implementation:
The main elements of the pattern are:

- an interface which defines the operations that the client code can perform on the flyweight object
- one or more concrete implementations of our interface
- a factory to handle objects instantiation and caching

Example:

For applying flyweight pattern, we need to create a **Flyweight factory** that returns the shared objects.

Create a Shape interface and concrete class Circle implementing the Shape interface. Since this interface will be the return type of the factory method we need to make sure to expose all the relevant methods:



Create a *Shape* interface and concrete class *Circle* implementing the *Shape* interface.

```
public interface Shape {
      void draw();
}
```

Create concrete class implementing the same interface.

```
public class Circle implements Shape {

      private String color;
      private int x;
      private int y;
      private int radius;

      public Circle(String color){
            this.color = color;
      }

      public void setX(int x) {
            this.x = x;
      }
```

```
	public void setY(int y) {
		this.y = y;
	}

	public void setRadius(int radius) {
		this.radius = radius;
	}

	@Override
	public void draw() {
		System.out.println(
		"Circle: Draw() [color=" + color + ", x=" + x + ", y=" + y + ",
		radius=" + radius + "]");
	}
}
```

Create a factory to generate object of concrete class based on given information.

```
public class ShapeFactory {

	private static final HashMap<Object, Object> circleMap = new
	HashMap<>();

	public static Shape getCircle(String color) {
		Circle circle = (Circle)circleMap.get(color);

		if(circle == null) {
			circle = new Circle(color);
			circleMap.put(color, circle);
			System.out.println("Creating circle of color : " + color);
		}
		return circle;
	}
}
```

Use the factory to get object of concrete class by passing an information such as color.

```
public class TestFlyweight {

	private static final String[] colors = { "Red", "Green", "Blue",
	"White", "Black" };
	public static void main(String[] args) {

		for(int i = 0; i < 5; ++i) {
			Circle circle =
(Circle)ShapeFactory.getCircle(getRandomColor());
			circle.setX(getRandomX());
			circle.setY(getRandomY());
			circle.setRadius(100);
			circle.draw();
		}
	}

	private static String getRandomColor() {
		return colors[(int)(Math.random()*colors.length)];
	}
```

```
        private static int getRandomX() {
              return new Random().nextInt();
        }

        private static int getRandomY() {
              return new Random().nextInt();
        }
  }
```

Use Cases:

1. Data Compression

- The goal of the flyweight pattern is to reduce memory usage by sharing as much data as possible, hence, it's a good basis for lossless compression algorithms. In this case, each flyweight object acts as a pointer with its extrinsic state being the context-dependent information.
- A classic example of this usage is in a word processor. Here, each character is a flyweight object which shares the data needed for the rendering. As a result, only the position of the character inside the document takes up additional memory.

2. Data Caching

- Many modern applications use caches to improve response time. The flyweight pattern is similar to the core concept of a cache and can fit this purpose well.
- Of course, there are a few key differences in complexity and implementation between this pattern and a typical, general-purpose cache.

When to use Flyweight Pattern:

- Many like objects are used and storage cost is high.
- The majority of each object's state can be made extrinsic.
- A few shared objects can replace many unshared ones.
- The identity of each object does not matter.

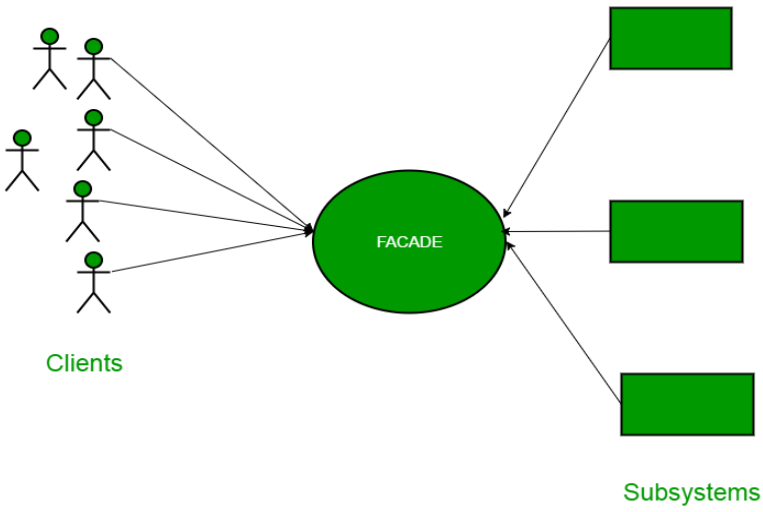
5. Facade Pattern:

- **Purpose**: Supplies a single interface to a set of interfaces within a system.
- A facade encapsulates a complex subsystem behind a simple interface. **It hides much of the complexity and makes the subsystem easy to use.**
- **Provide a unified interface to a set of interfaces in a subsystem. Facade Pattern defines a higher-level interface that makes the subsystem easier to use**
- Besides a much simpler interface, there's one more benefit of using this design pattern. **It decouples a client implementation from the complex subsystem.** Thanks to this, we can make changes to the existing subsystem and don't affect a client.
- Facade Pattern is used to help client applications to easily interact with the system.

Real Life Example:

Suppose we have an application with set of interfaces to use MySQL/Oracle database and to generate different types of reports, such as HTML report, PDF report etc. So we will have different set of interfaces to work with different types of database. Now a client application can use these interfaces to get the required database connection and generate reports. But when the complexity increases or the interface behavior names are confusing, client application will find it difficult to manage it. So we can apply Facade pattern here and provide a wrapper interface on top of the existing interface to help client application.

Another example, By exposing a set of functionalities through a web service the client code needs to only worry about the simple interface being exposed to them and not the complex relationships that may or may not exist behind the web service layer. A single web service call to update a system with new data may actually involve communication with a number of databases and systems, however this detail is hidden due to the implementation of the façade pattern.



Example:
Let's consider a hotel. This hotel has a hotel keeper. There are a lot of restaurants inside hotel e.g. Veg restaurants, Non-Veg restaurants and Veg/Non both restaurants. You, as client want access to different menus of different restaurants. You do not know what the different menus they have are. You just have access to hotel keeper who knows his hotel well. Whichever menu you want, you tell the hotel keeper and he takes it out of from the respective restaurants and hands it over to you. Here, the hotel keeper acts as the **facade**, as he hides the complexities of the system hotel.

```
public interface Hotel {
	public Menus getMenus();
}


public class Menus {
}


class NonVegMenu extends Menus {
	String nonVeg;
	public NonVegMenu() {
		nonVeg = "Chicken Handi";
	}

	@Override
	public String toString() {
		return "NonVegMenu [nonVeg=" + nonVeg + "]";
	}
}


class VegMenu extends Menus {

	String veg;
	public VegMenu() {
		veg = "Palak Paneer";
	}

	@Override
	public String toString() {
		return "VegMenu [veg=" + veg + "]";
	}
}
```

The hotel interface only returns Menus. Similarly, the Restaurant are of two types and can implement the hotel interface. Let's have a look at the code for one of the Restaurants.

```
public class VegRestaurant implements Hotel {

    public Menus getMenus() {
      return new VegMenu();
    }
}
```

```
public class NonVegRestaurant implements Hotel {

      public Menus getMenus() {
            return new NonVegMenu();
      }
}
```

Now let's consider the facade,

```
public class HotelKeeper {
    public VegMenu getVegMenu() {
        VegRestaurant v = new VegRestaurant();
        VegMenu vegMenu = (VegMenu)v.getMenus();
        return vegMenu;
    }

    public NonVegMenu getNonVegMenu() {
        NonVegRestaurant v = new NonVegRestaurant();
        NonVegMenu nonVegMenu = (NonVegMenu)v.getMenus();
        return nonVegMenu;
    }
}
```

From this, it is clear that the complex implementation will be done by HotelKeeper himself. The client will just access the HotelKeeper and ask for either Veg or NonVeg menu.

How will the client program access this façade?

```
public class Client {
    public static void main (String[] args) {
        HotelKeeper keeper = new HotelKeeper();

        VegMenu v = keeper.getVegMenu();
        NonVegMenu nv = keeper.getNonVegMenu();

        System.out.println(v.toString());
        System.out.println(nv.toString());
    }
}
```

In this way the implementation is sent to the façade. The client is given just one interface and can access only that. This hides all the complexities.

Drawbacks:

- The facade pattern doesn't force us to unwanted tradeoffs, because it only adds additional layers of abstraction.
- Sometimes the pattern can be overused in simple scenarios, which will lead to redundant implementations.

Important points:

- Facade design pattern is more like a helper for client applications, it doesn't hide subsystem interfaces from the client. Whether to use Facade or not is completely dependent on client code.
- Facade design pattern can be applied at any point of development, usually when the number of interfaces grow and system gets complex.
- Subsystem interfaces are not aware of Facade and they shouldn't have any reference of the Facade interface.
- Facade design pattern should be applied for similar kind of interfaces, its purpose is to provide a single interface rather than multiple interfaces that does the similar kind of jobs.
- We can use Factory pattern with Facade to provide better interface to client systems.

When to use Façade Pattern:

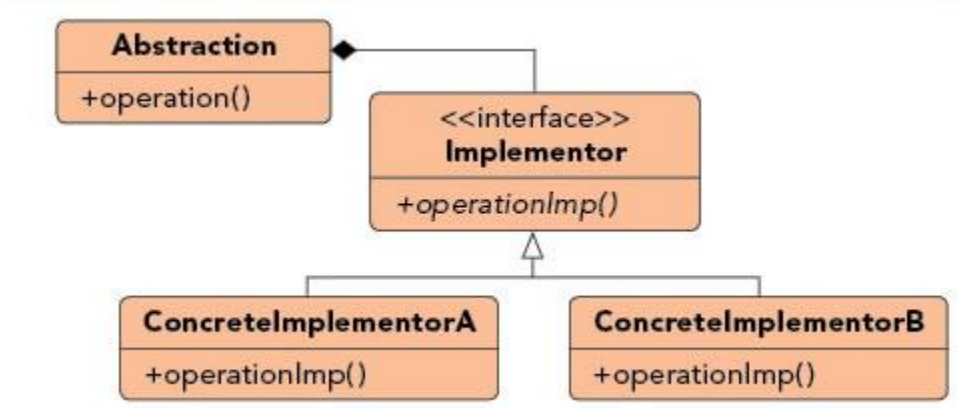
- When a simple interface is needed to provide access to a complex system.
- When there are many dependencies between system implementations and clients.
- When systems and subsystems should be layered.
- The facade pattern is appropriate when you have a **complex system** that you want to expose to clients in a simplified way, or you want to make an external communication layer over an existing system which is incompatible with the system. Facade deals with interfaces, not implementation.
- Its purpose is to hide internal complexity behind a single interface that appears simple on the outside.

6. Bridge Pattern:

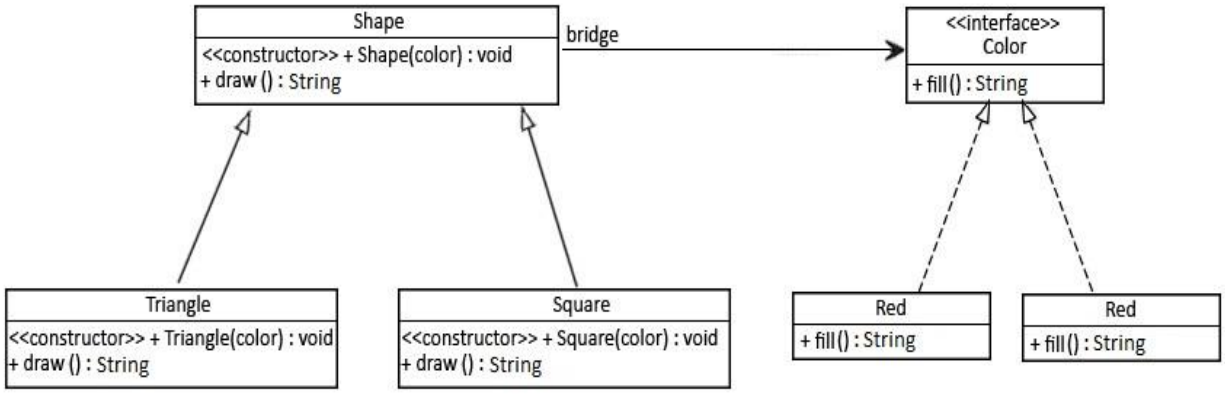
- **Purpose:** Defines an abstract object structure independently of the implementation object structure in order to limit coupling.
- Design pattern is: Decouple an abstraction from its implementation so that the two can vary independently.
- The implementation of bridge design pattern follows the notion to prefer Composition over Inheritance.
- Bridge is used when we need to decouple an abstraction from its implementation so that the two can vary independently.
- This pattern decouples implementation class and abstract class by providing a bridge structure between them.
- This pattern involves an interface which acts as a bridge which makes the functionality of concrete class's independent from interface implementer classes. Both types of classes can be altered structurally without affecting each other.

Implementation in JDK:

The Java Virtual Machine (JVM) has its own native set of functions that abstract the use of windowing, system logging, and byte code execution but the actual implementation of these functions is delegated to the operating system the JVM is running on. When an application instructs the JVM to render a window it delegates the rendering call to the concrete implementation of the JVM that knows how to communicate with the operating system in order to render the window.



Example:
For the Bridge pattern, we'll consider two layers of abstraction; one is the geometric shape (like triangle and square) which is filled with different colors (our second abstraction layer):



Define a color interface:

```
public interface Color {
    String fill();
}
```

Create a concrete class for this interface:

```
public class Red implements Color {

    @Override
    public String fill() {
           return "Color is Red";
    }
}
```

Create a similar concrete class Blue for this interface: Color

Create an abstract *Shape* class which consists a reference (bridge) to the *Color* object:

```
public abstract class Shape {
    protected Color color;

    public Shape() {}

    public Shape(Color color) {
          this.color = color;
    }

    public abstract String draw();
}
```

Create a concrete class of *Shape* interface which will utilize method from *Color* interface as well:

```
public class Square extends Shape {
    public Square(Color color) {
          super(color);
    }

    @Override
    public String draw() {
          return "Square drawn. " + color.fill();
    }
}
```

Create a similar concrete class Triangle which will utilize method from *Color* interface as well:

Using Bridge pattern and passing the desired color object.

```
public class BridgePattern {
    public static void main(String[] args) {
          //a square with red color
          Shape square = new Square(new Red());
          System.out.println(square.draw());

          Triangle triangle = new Triangle(new Blue());
          System.out.println(triangle.draw());
    }
}
```

Output:
Square drawn. Color is Red
Triangle drawn. Color is Blue

As we can note in the output, the shape gets draws with the desired color.

When to Use Bridge Design Pattern:

- When we want a parent abstract class to define the set of basic rules, and the concrete classes to add additional rules.
- When we have an abstract class that has a reference to the objects, and it has abstract methods that will be defined in each of the concrete classes.
- Bridge design pattern can be used when both abstraction and implementation can have different hierarchies independently and we want to hide the implementation from the client application.
- Abstractions and implementations should not be bound at compile time.
- Abstractions and implementations should be independently extensible.
- Changes in the implementation of an abstraction should have no impact on clients.
- Implementation details should be hidden from the client.

Important Points:

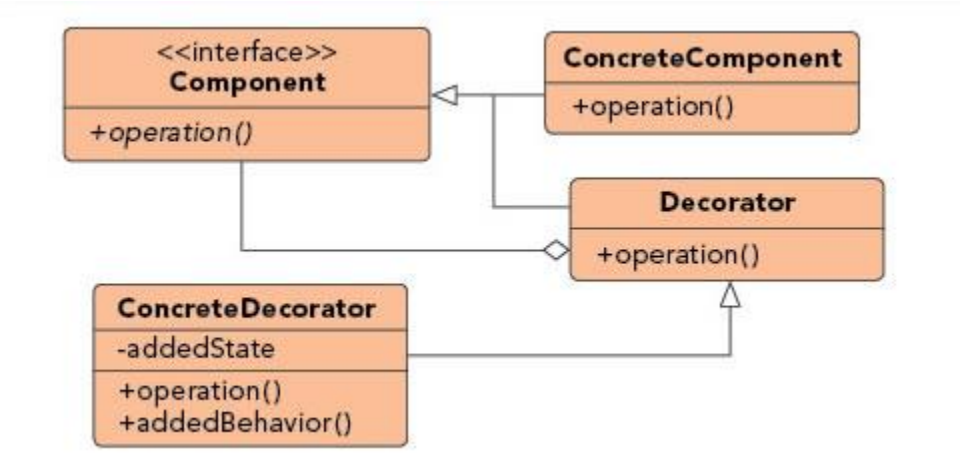
- A Bridge pattern can only be implemented before the application is designed.
- Allows an abstraction and implementation to change independently whereas an Adapter pattern makes it possible for incompatible classes to work together.

7. Decorator Pattern:

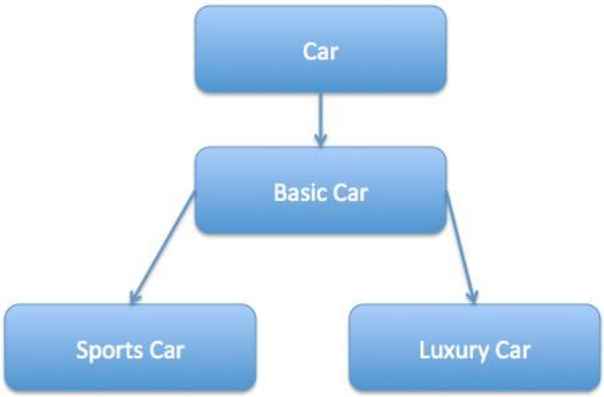
- **Purpose**: Allows for the dynamic wrapping of objects in order to modify their existing responsibilities and behaviors.
- **Decorator** design pattern is used to modify the functionality of an object at runtime. At the same time other instances of the same class will not be affected by this, so individual object gets the modified behavior.
- Decorator pattern allows a user to add new functionality to an existing object without altering its structure.
- This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.
- We use inheritance or composition to extend the behavior of an object but this is done at compile time and it's applicable to all the instances of the class. We can't add any new functionality of remove any existing behavior at runtime – this is when Decorator pattern comes into picture.
- A Decorator pattern can be used to attach additional responsibilities to an object either statically or dynamically. A Decorator provides an enhanced interface to the original object.
- In the implementation of this pattern, we prefer composition over an inheritance – so that we can reduce the overhead of sub classing again and again for each decorating element. The recursion involved with this design can be used to decorate our object as many times as we require.
- **Decorator pattern is used a lot in Java IO classes, such as FileReader, BufferedReader etc**.

Real-life Example:

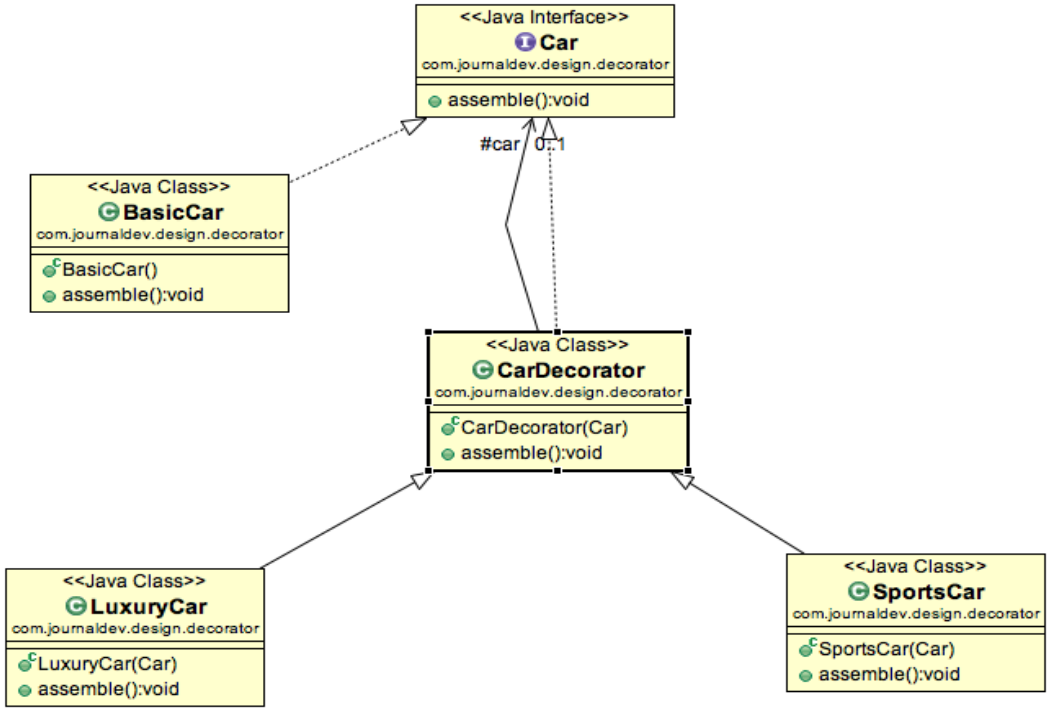
Many businesses set up their mail systems to take advantage of decorators. When messages are sent from someone in the company to an external address the mail server decorates the original message with copyright and confidentiality information. As long as the message remains internal the information is not attached. This decoration allows the message itself to remain unchanged until a runtime decision is made to wrap the message with additional information.



Example:
Suppose we want to implement different kinds of cars – we can create interface Car to define the assemble method and then we can have a Basic car, further more we can extend it to Sports car and Luxury Car. The implementation hierarchy will look like below image.



But if we want to get a car at runtime that has both the features of sports car and luxury car, then the implementation gets complex and if further more we want to specify which features should be added first, it gets even more complex. Now imagine if we have ten different kind of cars, the implementation logic using inheritance and composition will be impossible to manage. To solve this kind of programming situation, we apply decorator pattern in java.



Component Interface – The interface or abstract class defining the methods that will be implemented. In our case Car will be the component interface.

```
public interface Car {
    public void assemble();
}
```

Component Implementation – The basic implementation of the component interface. We can have BasicCar class as our component implementation.

```
public class BasicCar implements Car {

    @Override
    public void assemble() {
          System.out.print("Basic Car.");
    }

}
```

Decorator – Decorator class implements the component interface and it has a HAS-A relationship with the component interface. The component variable should be accessible to the child decorator classes, so we will make this variable protected.

```
public class CarDecorator implements Car {

    protected Car car;

    public CarDecorator(Car car){
          this.car = car;
    }

    @Override
    public void assemble() {
          this.car.assemble();
    }
}
```

Concrete Decorators – Extending the base decorator functionality and modifying the component behavior accordingly. We can have concrete decorator classes as LuxuryCar and SportsCar.

```
public class LuxuryCar extends CarDecorator {

    public LuxuryCar(Car car) {
          super(car);
    }

    @Override
    public void assemble(){
          super.assemble();
          System.out.print("Adding features of Luxury Car.");
    }
}
```

Create a similar class SportsCar.

Create client program that can create different kinds of Object at runtime and they can specify the order of execution too.

```
public class DecoratorPattern {

    public static void main(String[] args) {
          Car sportsCar = new SportsCar(new BasicCar());
          sportsCar.assemble();
          System.out.println("\n*****");

          Car sportsLuxuryCar = new SportsCar(new LuxuryCar(new
                                                      BasicCar()));
          sportsLuxuryCar.assemble();
    }
}
```

Output:
Basic Car.Adding features of Sports Car.
*****
Basic Car.Adding features of Luxury Car.Adding features of Sports Car.

When to Use Decorator Pattern:

- When we wish to add, enhance or even remove the behavior or state of objects.
- When we just want to modify the functionality of a single object of class and leave others unchanged
- When Object responsibilities and behaviors should be dynamically modifiable.
- When Concrete implementations should be decoupled from responsibilities and behaviors.
- When Sub classing to achieve modification is impractical or impossible.
- When Specific functionality should not reside high in the object hierarchy.
- When a lot of little objects surrounding a concrete implementation is acceptable.

Important Points:

- Decorator design pattern is helpful in providing runtime modification abilities and hence more flexible. It's easy to maintain and extend when the number of choices are more.
- The disadvantage of decorator design pattern is that it uses a lot of similar kind of objects (decorators).
- Although Proxy and Decorator patterns have similar structures, they differ in intention; while Proxy's prime purpose is to facilitate ease of use or controlled access, a Decorator attaches additional responsibilities
- Both Proxy and Adapter patterns hold reference to the original object.
- All the decorators from this pattern can be used recursively, infinite number of times, which is neither possible with other models.