Flyweight Design Pattern

Flyweight pattern is one of the [structural design patterns](https://www.geeksforgeeks.org/design-patterns-set-1-introduction/) as this pattern provides ways to decrease object count thus improving application required objects structure. Flyweight pattern is used when we need to create a large number of similar objects (say 105). One important feature of flyweight objects is that they are **immutable**. This means that they cannot be modified once they have been constructed.

**Why do we care for number of objects in our program?**

* Less number of objects reduces the memory usage, and it manages to keep us away from errors related to memory like [java.lang.OutOfMemoryError.](https://docs.oracle.com/javase/7/docs/api/java/lang/OutOfMemoryError.html)
* Although creating an object in Java is really fast, we can still reduce the execution time of our program by sharing objects.

In Flyweight pattern we use a [HashMap](https://www.geeksforgeeks.org/hashmap-treemap-java/) that stores reference to the object which have already been created, every object is associated with a key. Now when a client wants to create an object, he simply has to pass a key associated with it and if the object has already been created we simply get the reference to that object else it creates a new object and then returns it reference to the client.

**Intrinsic and Extrinsic States**

To understand Intrinsic and Extrinsic state, let us consider an example.

Suppose in a text editor when we enter a character, an object of Character class is created, the attributes of the Character class are {name, font, size}. We do not need to create an object every time client enters a character since letter ‘B’ is no different from another ‘B’ . If client again types a ‘B’ we simply return the object which we have already created before. Now all these are intrinsic states (name, font, size), since they can be shared among the different objects as they are similar to each other.

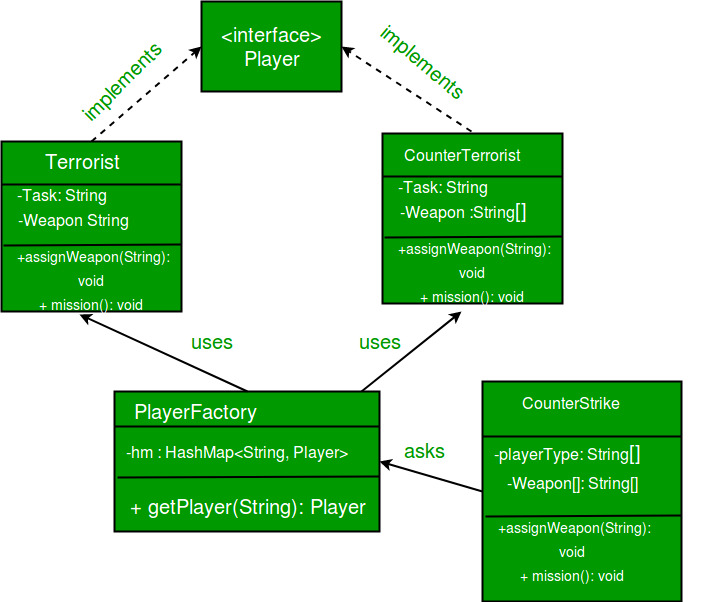
Now we add to more attributes to the Character class, they are row and column. They specify the position of a character in the document. Now these attributes will not be similar even for same characters, since no two characters will have the same position in a document, these states are termed as extrinsic states, and they can’t be shared among objects.

**Implementation :**We implement the creation of Terrorists and Counter Terrorists In the game of [Counter Strike](https://en.wikipedia.org/wiki/Counter-Strike). So we have 2 classes one for **T**errorist(**T**) and other for **C**ounter **T**errorist(**CT**). Whenever a player asks for a weapon we assign him the asked weapon. In the mission, terrorist’s task is to plant a bomb while the counter terrorists have to diffuse the bomb.

**Why to use Flyweight Design Pattern in this example?**Here we use the Fly Weight design pattern, since here we need to reduce the object count for players. Now we have n number of players playing CS 1.6, if we do not follow the Fly Weight Design Pattern then we will have to create n number of objects, one for each player. But now we will only have to create 2 objects one for terrorists and other for counter terrorists, we will reuse then again and again whenever required.

**Intrinsic State :**Here ‘task’ is an intrinsic state for both types of players, since this is always same for T’s/CT’s. We can have some other states like their color or any other properties which are similar for all the Terrorists/Counter Terrorists in their respective Terrorists/Counter Terrorists class.

**Extrinsic State :**Weapon is an extrinsic state since each player can carry any weapon of his/her choice. Weapon need to be passed as a parameter by the client itself.

Class Diagram :[](https://media.geeksforgeeks.org/wp-content/uploads/flyweight.jpg)

|  |
| --- |
| // A Java program to demonstrate working of  // FlyWeight Pattern with example of Counter  // Strike Game  import java.util.Random;  import java.util.HashMap;    // A common interface for all players  interface Player  {      public void assignWeapon(String weapon);      public void mission();  }    // Terrorist must have weapon and mission  class Terrorist implements Player  {      // Intrinsic Attribute      private final String TASK;        // Extrinsic Attribute      private String weapon;        public Terrorist()      {          TASK = "PLANT A BOMB";      }      public void assignWeapon(String weapon)      {          // Assign a weapon          this.weapon = weapon;      }      public void mission()      {          //Work on the Mission          System.out.println("Terrorist with weapon "                             + weapon + "|" + " Task is " + TASK);      }  }    // CounterTerrorist must have weapon and mission  class CounterTerrorist implements Player  {      // Intrinsic Attribute      private final String TASK;        // Extrinsic Attribute      private String weapon;        public CounterTerrorist()      {          TASK = "DIFFUSE BOMB";      }      public void assignWeapon(String weapon)      {          this.weapon = weapon;      }      public void mission()      {          System.out.println("Counter Terrorist with weapon "                             + weapon + "|" + " Task is " + TASK);      }  }    // Class used to get a player using HashMap (Returns  // an existing player if a player of given type exists.  // Else creates a new player and returns it.  class PlayerFactory  {      /\* HashMap stores the reference to the object         of Terrorist(TS) or CounterTerrorist(CT).  \*/      private static HashMap <String, Player> hm =                           new HashMap<String, Player>();        // Method to get a player      public static Player getPlayer(String type)      {          Player p = null;            /\* If an object for TS or CT has already been             created simply return its reference \*/          if (hm.containsKey(type))                  p = hm.get(type);          else          {              /\* create an object of TS/CT  \*/              switch(type)              {              case "Terrorist":                  System.out.println("Terrorist Created");                  p = new Terrorist();                  break;              case "CounterTerrorist":                  System.out.println("Counter Terrorist Created");                  p = new CounterTerrorist();                  break;              default :                  System.out.println("Unreachable code!");              }                // Once created insert it into the HashMap              hm.put(type, p);          }          return p;      }  }    // Driver class  public class CounterStrike  {      // All player types and weapon (used by getRandPlayerType()      // and getRandWeapon()      private static String[] playerType =                      {"Terrorist", "CounterTerrorist"};      private static String[] weapons =        {"AK-47", "Maverick", "Gut Knife", "Desert Eagle"};          // Driver code      public static void main(String args[])      {          /\* Assume that we have a total of 10 players             in the game. \*/          for (int i = 0; i < 10; i++)          {              /\* getPlayer() is called simply using the class                 name since the method is a static one \*/              Player p = PlayerFactory.getPlayer(getRandPlayerType());                /\* Assign a weapon chosen randomly uniformly                 from the weapon array  \*/              p.assignWeapon(getRandWeapon());                // Send this player on a mission              p.mission();          }      }        // Utility methods to get a random player type and      // weapon      public static String getRandPlayerType()      {          Random r = new Random();            // Will return an integer between [0,2)          int randInt = r.nextInt(playerType.length);            // return the player stored at index 'randInt'          return playerType[randInt];      }      public static String getRandWeapon()      {          Random r = new Random();            // Will return an integer between [0,5)          int randInt = r.nextInt(weapons.length);            // Return the weapon stored at index 'randInt'          return weapons[randInt];      }  } |

**Output:**

Counter Terrorist Created

Counter Terrorist with weapon Gut Knife| Task is DIFFUSE BOMB

Counter Terrorist with weapon Desert Eagle| Task is DIFFUSE BOMB

Terrorist Created

Terrorist with weapon AK-47| Task is PLANT A BOMB

Terrorist with weapon Gut Knife| Task is PLANT A BOMB

Terrorist with weapon Gut Knife| Task is PLANT A BOMB

Terrorist with weapon Desert Eagle| Task is PLANT A BOMB

Terrorist with weapon AK-47| Task is PLANT A BOMB

Counter Terrorist with weapon Desert Eagle| Task is DIFFUSE BOMB

Counter Terrorist with weapon Gut Knife| Task is DIFFUSE BOMB

Counter Terrorist with weapon Desert Eagle| Task is DIFFUSE BOMB

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.**

## **2. 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

Let's see how to implement each component.

### **2.1. Vehicle Interface**

To begin with, we'll create a Vehicle interface. Since this interface will be the return type of the factory method we need to make sure to expose all the relevant methods:

|  |  |
| --- | --- |
| 1  2  3 | public void start();  public void stop();  public Color getColor(); |

### **2.2. Concrete Vehicle**

Next up, let's make a Car class as a concrete Vehicle. Our car will implement all the methods of the vehicle interface. As for its state, it'll have an engine and a color field:

|  |  |
| --- | --- |
| 1  2 | private Engine engine;  private Color color; |

### **2.3. Vehicle Factory**

Last but not least, we'll create the VehicleFactory. Building a new vehicle is a very expensive operation so the factory will only create one vehicle per color.

In order to do that, we keep track of the created vehicles using a map as a simple cache:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | private static Map<Color, Vehicle> vehiclesCache    = new HashMap<>();    public static Vehicle createVehicle(Color color) {      Vehicle newVehicle = vehiclesCache.computeIfAbsent(color, newColor -> {          Engine newEngine = new Engine();          return new Car(newEngine, newColor);      });      return newVehicle;  } |

Notice how the client code can only affect the extrinsic state of the object (the color of our vehicle) passing it as an argument to the createVehicle method.

## **3. Use Cases**

### **3.1. 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.

### **3.2. 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.