JAVA ADVANCE

(BEST PRACTICE WHILE CODING)

**USE STATIC FACTORY METHOD**

A static factory method is a public static method on the object that returns a new instance of the object.

eg:

1. public final class Arithmetic {
2. public static Arithmetic add(int a, int b)
3. {
   1. return new Arithmetic (a, b);
4. }
5. public static Arithmetic multi(int a, int b)
6. {
   1. return new Arithmetic (a, b);
7. }
8. public static void main(String args[])
9. {
10. Arithmetic sum = Arithmetic.add(10,20);
11. Arithmetic product = Arithmetic.multi(2,5);
12. System.out.println(“Sum = “+(sum.a+sum.b);
13. System.out.println(“Multiplication = “+(product.a\* product.b);
14. }
15. }

Instead of using **Static factory method** we can use **Constructor**. But problem is that for the above code we can use only one constructor. Here are some of the problem we face while using constructor and how we can avoid such problems by using static factory method

|  |  |
| --- | --- |
| While Using Constructor | While Using Static Factory Method |
| The constructor doesn’t have a meaningful name, so they always restricted to the standard naming convention  Constructors can’t have any return type not even void.  Inside the constructor, we can only perform the initialization of objects.   |  |  | | --- | --- | | Constructor always creates a new object inside the heap, so it is not possible to return a cached instance of the class from a constructor. |  | | The first line inside every constructor should be either super() or this() |  | | The static factory method can have a meaningful name hence we can explicitly convey what this method does  Static factory methods can return the same type that implements the method, a subtype, and also primitives   |  |  | | --- | --- | |  | Inside static factory method other than initialization if we want to perform any activity for every object creation like increasing count value for every object creation we can do this in the static factory method.  But Factory methods can take advantage of caching i.e we can return the same instance of Immutable class from the factory method instead of always creating a new object.  But inside the factory method, it is not necessary that the first line must be either super() or this() | |

So Use Static factory methods

Avoid multiple constructor and replace it with static factory method and use proper name for each one based on their functionality

Avoid creation unnecessary duplicate object by calling constructor while static factory methods are not required to create a new object each time they’re invoked

**CREATE IMMUTABLE CLASS**

Immutable class in java means that once an object is created, we cannot change its content. In Java, all the [wrapper classes](https://www.geeksforgeeks.org/wrapper-classes-java/) (like Integer, Boolean, Byte, Short) and String class is immutable

To create immutable class, we need to follow these requirements

 The **class** must be declared as **final** so that child classes can’t be created.

 **Data members** in the class must be declared **private** so that direct access is not allowed.

 **Data members** in the class must be declared as so that we can’t change the value of it after object creation.

 A **parameterized constructor** should initialize all the fields performing a **deep copy** so that data members can’t be modified with an object reference.

 **Deep Copy** of objects should be performed in the **getter methods** to return a copy rather than returning the actual object reference

final class Student {

    private final String name;

    private final int regNo;

    private final Map<String, String> metadata;

// Constructor of immutable class  Parameterized

  public Student(String name, int regNo,

                   Map<String, String> metadata)

    {

        this.name = name;

        this.regNo = regNo;

//Creating Map object with reference to HashMap of String type

        Map<String, String> tempMap = new HashMap<>();

        for (Map.Entry<String, String> entry :

             metadata.entrySet()) {

            tempMap.put(entry.getKey(), entry.getValue());

        }

        this.metadata = tempMap;

    }

    public String getName() { return name; }

    public int getRegNo() { return regNo; }

    public Map<String, String> getMetadata()

    {

        Map<String, String> tempMap = new HashMap<>();

        for (Map.Entry<String, String> entry :

             this.metadata.entrySet()) {

            tempMap.put(entry.getKey(), entry.getValue());

        }

        return tempMap;

    }

}

eg:

class StudentMain {

    // Main driver method

    public static void main(String[] args)

    {

        // Creating Map object with reference to HashMap

        Map<String, String> map = new HashMap<>();

        // Adding elements to Map object

        // using put() method

        map.put("1", "first");

        map.put("2", "second");

        Student s = new Student("ABC", 101, map);

        // Calling the above methods 1,2,3 of class1

        // inside main() method in class2 and

        // executing the print statement over them

        System.out.println(s.getName());

        System.out.println(s.getRegNo());

        System.out.println(s.getMetadata());

        // Uncommenting below line causes error

        // s.regNo = 102;

        map.put("3", "third");

        // Remains unchanged due to deep copy in constructor

        System.out.println(s.getMetadata());

        s.getMetadata().put("4", "fourth");

        // Remains unchanged due to deep copy in getter

        System.out.println(s.getMetadata());

    }

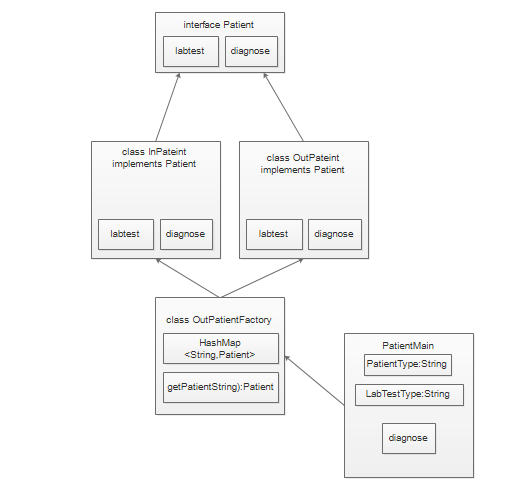
}

An **immutable** object remains in exactly one state, the state in which it was created. Therefore, immutable object is **thread-safe** so there is no synchronization issue. They **cannot be corrupted** by **multiple threads** accessing them concurrently. This is far and away the easiest approach to achieving thread safety. Hence the internal state of program will be consistent even if you have **exceptions**.

**USE FLYWEIGHT DESIGN PATTERN**

**Flyweight** is a **structural design pattern** that lets you fit more objects into the available amount of **RAM** by sharing common parts of state between multiple objects instead of keeping all of the data in each object

The bellow diagram is a simple representation of how flyweight pattern is used



<https://github.com/jaya47sankar/AdvanceJavaTutorials/blob/main/AdvanceJavaTutorials/src/jss/day1/flyweight/PatientMain.java>

check the above url for reference code