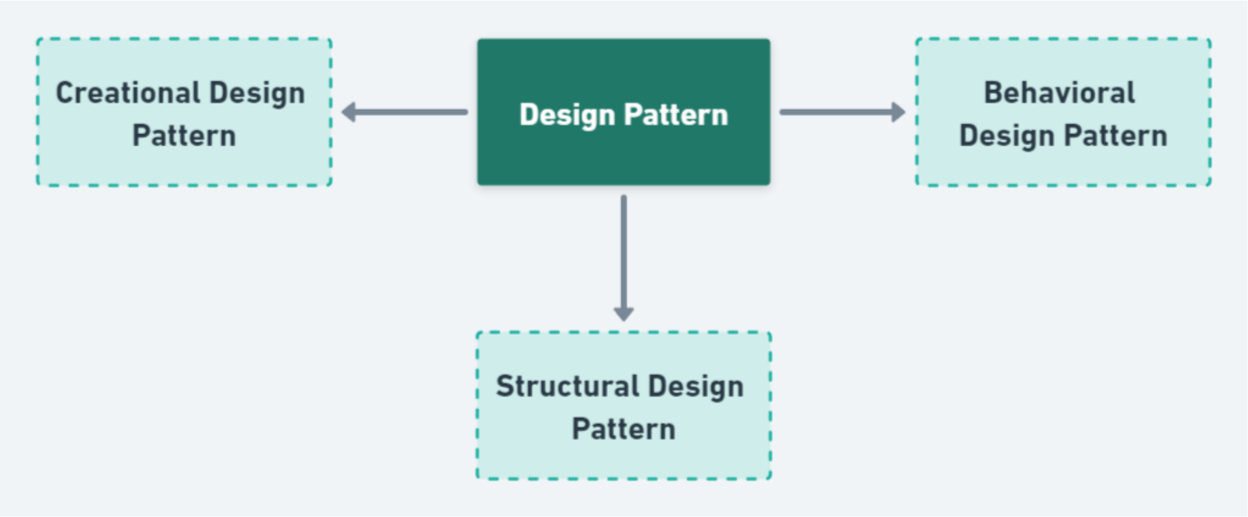
DESIGN PATTERN IN JAVA :

By reading the above heading the first question which comes to our mind is –

1). What is Design Pattern? Why is it important in software industry?

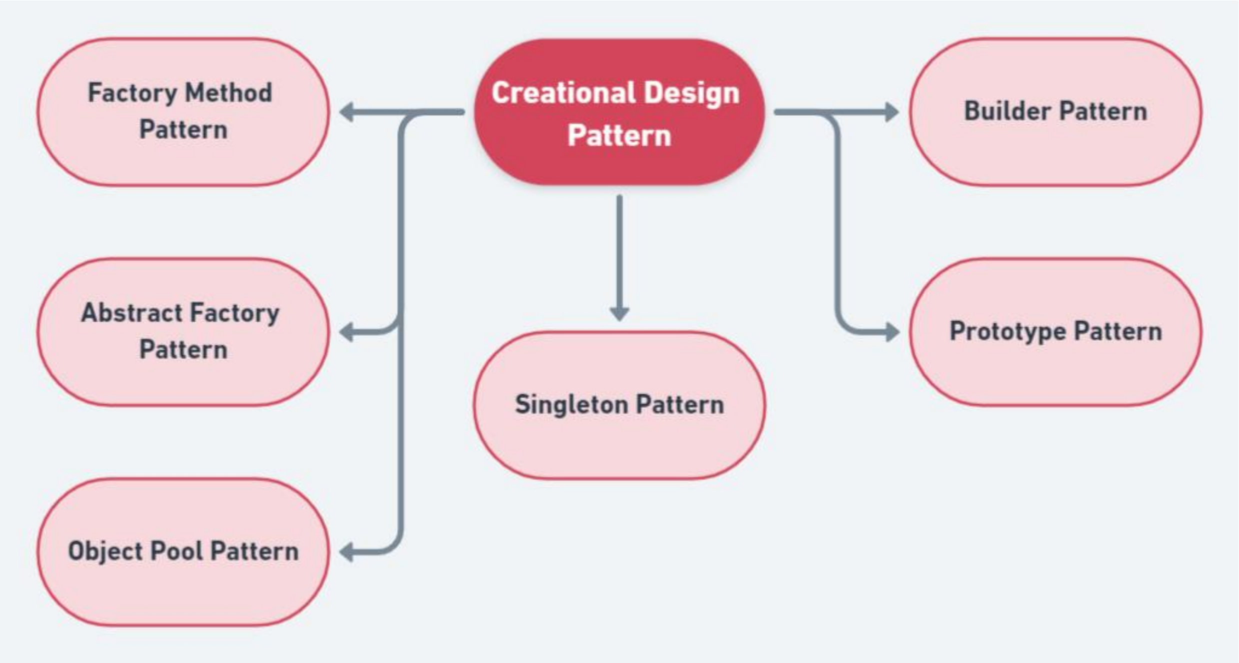
- ➢ Design Pattern is an Industry Approach to solve the more complex and recurring problem.
- ➢ It promotes reusability which leads to robust and maintainable code.
- ➢ It makes our code easy to understand and can be easily debugged.

• Types of Design Pattern in Java :


Creational Design Pattern
Design Pattern
Behavioral Design Pattern
Structural Design Pattern

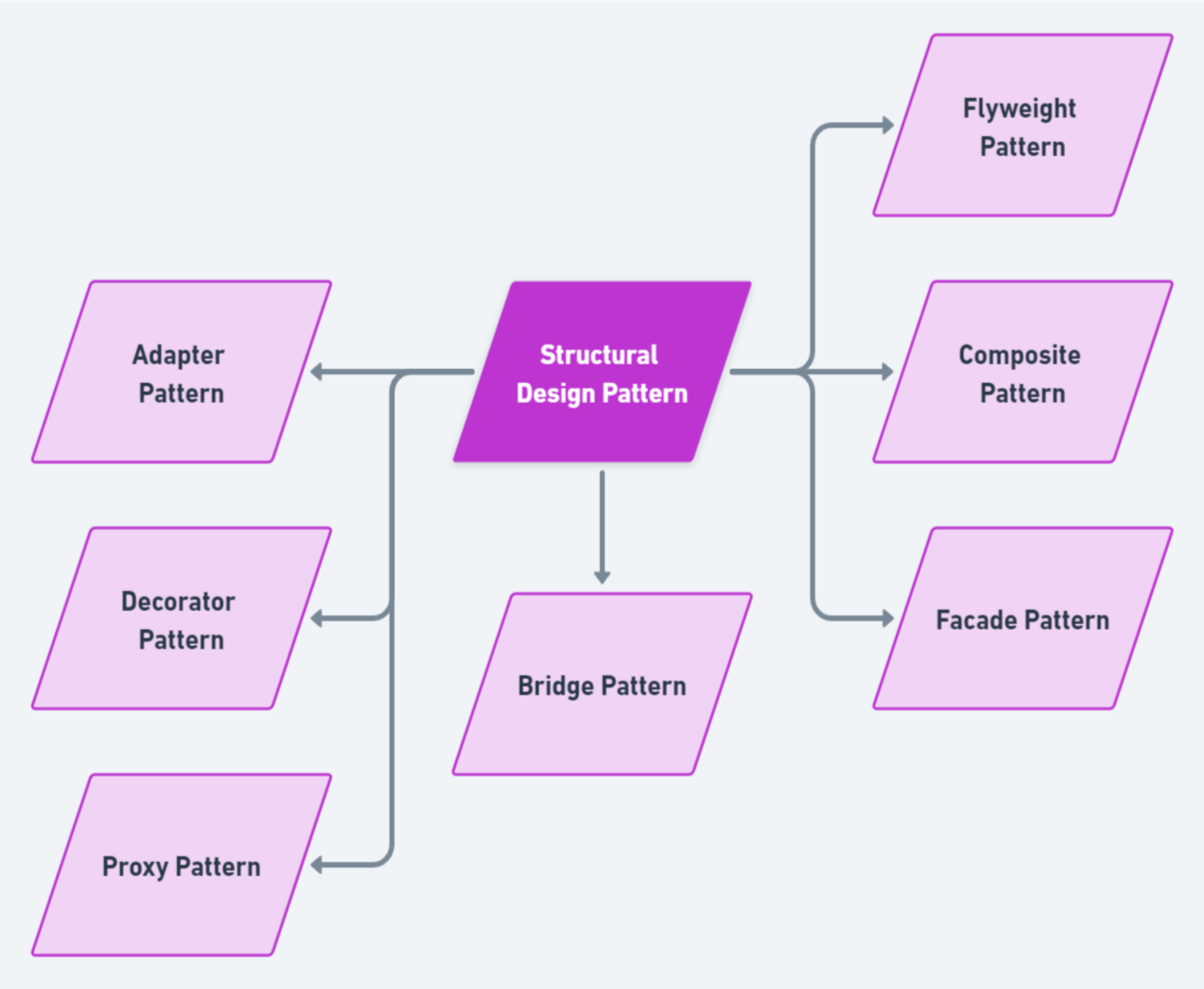
- **Creational Design Pattern :**

  - Creational Design Patterns allows us to create the object in best possible way.



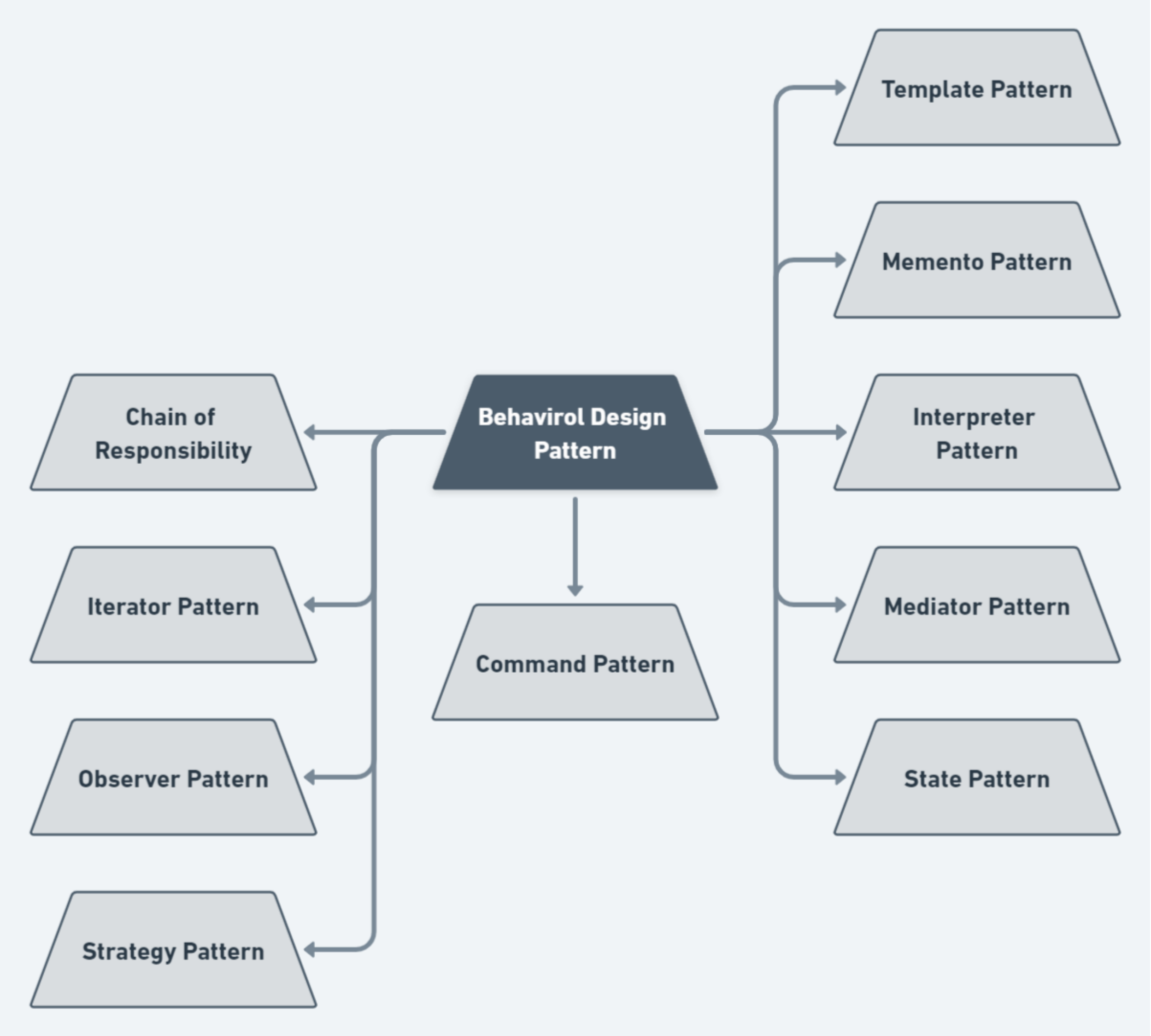
- **Structural Design Pattern :**

  - Structural design pattern describes how objects and classes can be combined to form the largest structure.



- **Behavioral Design Pattern :**

➢ Behavioral Design Pattern describes how one class communicate with other class in a loose couple manner.



Singleton Design Pattern :

- ✓ It allows us to create only one instance of the class.
- ✓ We must make sure that singleton class must provide the global access point to get the instance
- ✓ i.e. we must have at least one public method in a class which returns instance of a class.

Different approaches to implement singleton design pattern :

- Eager initialization
- Static Block initialization
- Lazy initialization
- Thread safe singleton
- Bill pugh singleton implementation
- Using Reflection to destroy singleton pattern
- Enum singleton
- Using clone to destroy/prevent singleton
- Using serialization destroy/prevent singleton
- Example of singleton within JDK

Steps to create Singleton Pattern :

1. Always create your constructor private so that multiple objects cannot be created from other classes.
2. Create private static variable so that it is the only instance of the class.
3. Create public static method to return the instance of a class and it remains the global access for the instance of the singleton class.

❖ Eager initialization :

- ✓ In eager initialization the instance/object of the class is created at the time of class loading.
- ✓ The drawback for this is the instance of class is created whether the client is using it or not.
- ✓ Exception handling cannot be done.

Step 1:

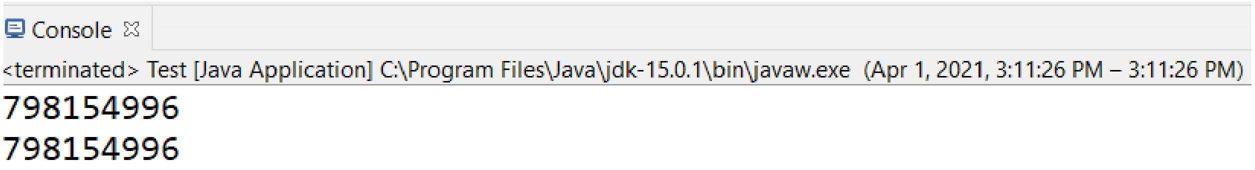
```
package pack_Singleton;

public class Singleton
{
	private static final Singleton instance = new
Singleton();

	private Singleton()
	{

	}

	public static Singleton getinstance()
	{
		return instance;
	}

}
```

Step 2:

```
package pack_Singleton;

public class Test {

	public static void main(String[] args)
	{
		Singleton obj1 = Singleton.getinstance();
		Singleton obj2 = Singleton.getinstance();
```

```
        System.out.println(obj1.hashCode());
        System.out.println(obj2.hashCode());

    }

}
```

Output :



```
Console ☒
<terminated> Test [Java Application] C:\Program Files\Java\jdk-15.0.1\bin\javaw.exe  (Apr 1, 2021, 3:11:26 PM – 3:11:26 PM)
798154996
798154996
```

- The above output show that the unique code for both the objects are same that means the object is created only once and is shared while calling the method.

❖ Static Block Initialization :

- ✓ In static block initialization we need to create one static block and inside that block we give memory to the instance of a class.
- ✓ We can handle exception inside the static block.

Step 1 :

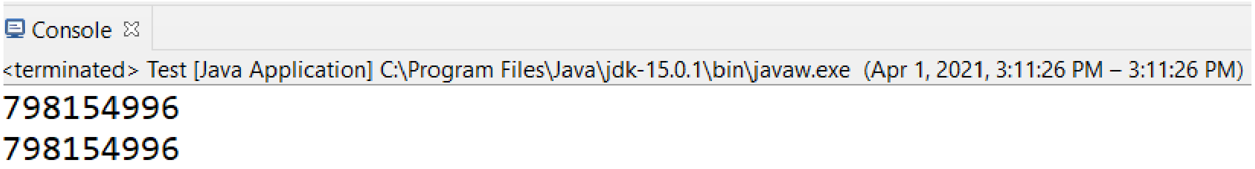
```
package pack_Singleton;

public class Singleton
{
    private static Singleton instance = null;

    private Singleton()
    {

    }

    static
    {
        try
        {
            if(instance == null)
            {
                instance = new Singleton();
            }
        } catch (Exception e)
        {
            e.printStackTrace();
        }
    }

    public static Singleton getinstance()
    {
        return instance;
    }

}
```

Step 2 :

```
package pack_Singleton;
```

```java
public class Test {

	public static void main(String[] args)
	{
		Singleton obj1 = Singleton.getinstance();
		Singleton obj2 = Singleton.getinstance();

		System.out.println(obj1.hashCode());
		System.out.println(obj2.hashCode());

	}

}
```

Output :



```
<terminated> Test [Java Application] C:\Program Files\Java\jdk-15.0.1\bin\javaw.exe  (Apr 1, 2021, 3:11:26 PM – 3:11:26 PM)
798154996
798154996
```

- **The above output show that the unique code for both the objects are same that means the object is created only once and is shared while calling the method.**

❖ Lazy Initialization :

- ✓ In lazy initialization the instance is given memory inside the getinstance method.
- ✓ This approach is good for single threaded environment for multithreaded it creates problem as more than one thread can come inside the if block.
- ✓ So it is not thread safe.

Step 1 :

```
package pack_Singleton;

public class Singleton
{
    private static Singleton instance = null;

    private Singleton()
    {

    }

    public static Singleton getinstance()
    {
        if(instance == null)
        {
            instance = new Singleton();
        }
        return instance;
    }

}
```

Step 2 and output will be same as that of the above 2 approaches.

❖ Thread Safe Singleton :

- ✓ Always used for multithreaded environment.
- ✓ Always make use of synchronized method or double locking to prevent breakage of singleton pattern.

Step 1 :

First create one class with main method to create and maintain a reusable pool of threads.

```
package pack_Singleton;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Test {

    public static void main(String[] args)
    {
        ExecutorService executorService = null;
        MyThread mythread = new MyThread();

        try
        {
            executorService =
Executors.newFixedThreadPool(3);
            executorService.execute(mythread);
            executorService.execute(mythread);
            executorService.execute(mythread);
        } catch (Exception e)
        {
            e.printStackTrace();
        }
        finally
        {
            if(executorService != null)
            {
                executorService.shutdown();
            }
        }
    }

}
```

Step 2:

Create a class to implement runnable interface

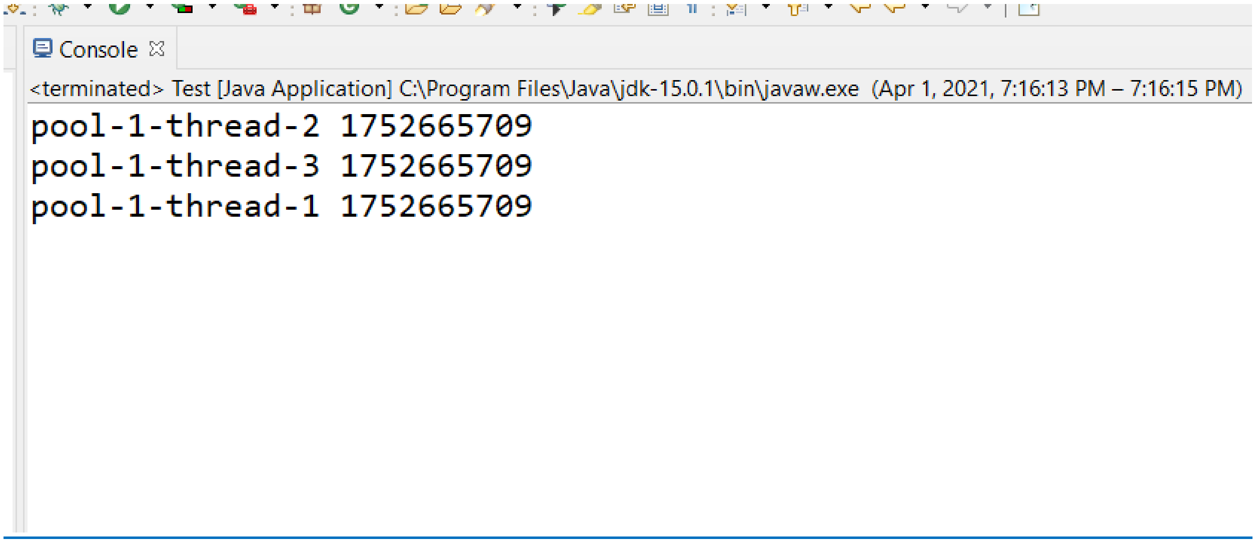
```
package pack_Singleton;

public class MyThread implements Runnable
{
    @Override
    public void run()
    {
        Singleton obj1 = Singleton.getinstance();


    System.out.println(Thread.currentThread().getName()+"
"+obj1.hashCode());
    }
}
```

Step 3:

Create a class to implement singleton pattern

```java
package pack_Singleton;

public class Singleton
{
    private static Singleton instance = null;

    private Singleton()
    {

    }

    public static synchronized Singleton getinstance()
    {
        if(instance == null)
        {
            try
            {
                Thread.sleep(2000);
            } catch (Exception e)
            {
                e.printStackTrace();
            }
            instance = new Singleton();
        }
        return instance;
    }

}
```

Output:



```
<terminated> Test [Java Application] C:\Program Files\Java\jdk-15.0.1\bin\javaw.exe  (Apr 1, 2021, 7:16:13 PM – 7:16:15 PM)
pool-1-thread-2 1752665709
pool-1-thread-3 1752665709
pool-1-thread-1 1752665709
```

❖ Bill pugh singleton implementation :

- ✓ To prevent the issue of memory modelling in java bill pugh is used.
- ✓ When any client call the getinstance method then only the instance is created.
- ✓ When we use this with multithreaded environment we don't need to synchronized the class, therefore it is thread safe.

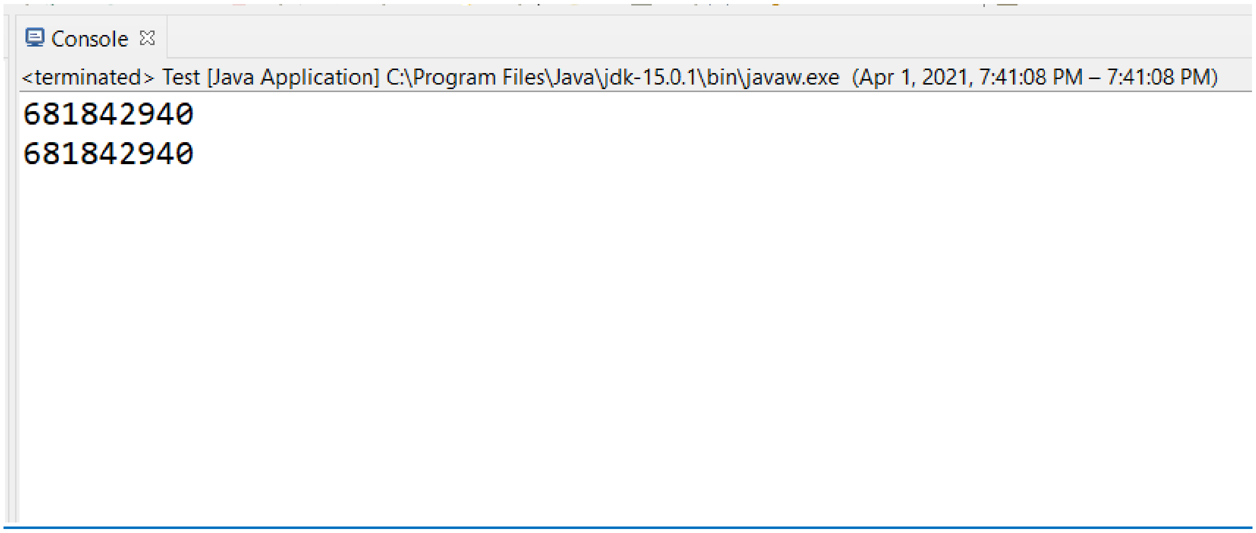
Step 1:

```
package pack_Singleton;

public class Singleton
{
	private Singleton()
	{
    }

    public static class SingletonFolder
    {
        public static final Singleton instance = new
Singleton();
    }

    public static Singleton getinstance()
    {
        return SingletonFolder.instance;
    }

}
```

Step 2:

```
package pack_Singleton;

public class Test {

    public static void main(String[] args)
    {
        Singleton obj1 = Singleton.getinstance();
        Singleton obj2 = Singleton.getinstance();

        System.out.println(obj1.hashCode());
        System.out.println(obj2.hashCode());
    }

}
```

Output:



```
<terminated> Test [Java Application] C:\Program Files\Java\jdk-15.0.1\bin\javaw.exe  (Apr 1, 2021, 7:41:08 PM – 7:41:08 PM)
681842940
681842940
```

❖ Using Reflection to destroy singleton pattern :

- ✓ Using reflection we can break singleton pattern created by above all the approaches.
- ✓ So if we want that our singleton shouldn't be broked by reflection we should use enum.

Step 1:

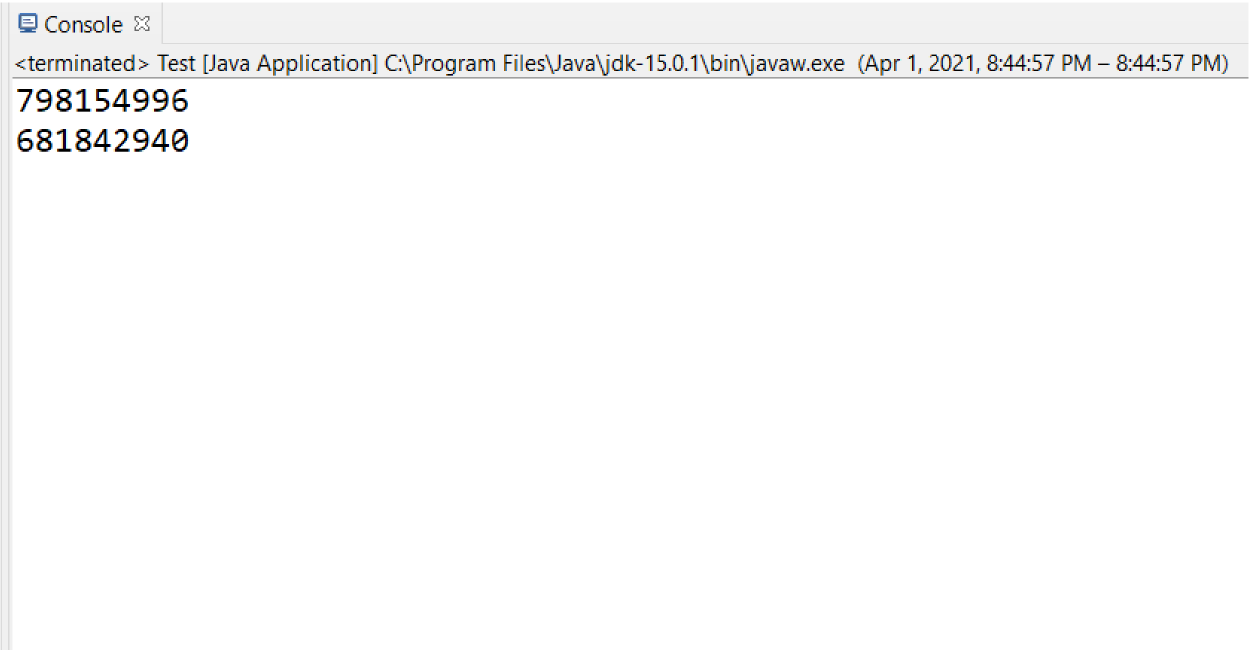
```
package pack_Singleton;

public class Singleton
{
    public static Singleton instance = null;
    private Singleton()
    {

    }

    public static Singleton getinstance()
    {
```

```java
        if(instance == null)
        {
            instance = new Singleton();
        }
        return instance;
    }

}
```

Step 2:

```java
package pack_Singleton;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

public class Test {

    public static void main(String[] args) throws
InstantiationException, IllegalAccessException,
IllegalArgumentException, InvocationTargetException
    {
        Singleton obj1 = Singleton.getinstance();
        Singleton obj2 = null;

        Constructor<?>[] constructors =
Singleton.class.getDeclaredConstructors();
        for (Constructor<?> constructor : constructors)
        {
            constructor.setAccessible(true);

            Object object = constructor.newInstance();
            obj2 = (Singleton)object;

            break;

        }
        System.out.println(obj1.hashCode());
        System.out.println(obj2.hashCode());
        }

}
```

Output :



```
<terminated> Test [Java Application] C:\Program Files\Java\jdk-15.0.1\bin\javaw.exe  (Apr 1, 2021, 8:44:57 PM – 8:44:57 PM)
798154996
681842940
```

❖ Singleton using enum :

- ✓ Using enum we can avoid breakage of singleton using reflection .
- ✓ In enum, java allow instantiation only once because it has global access.
- ✓ Drawback of enum is that it doesn’t allow Lazy initialization.

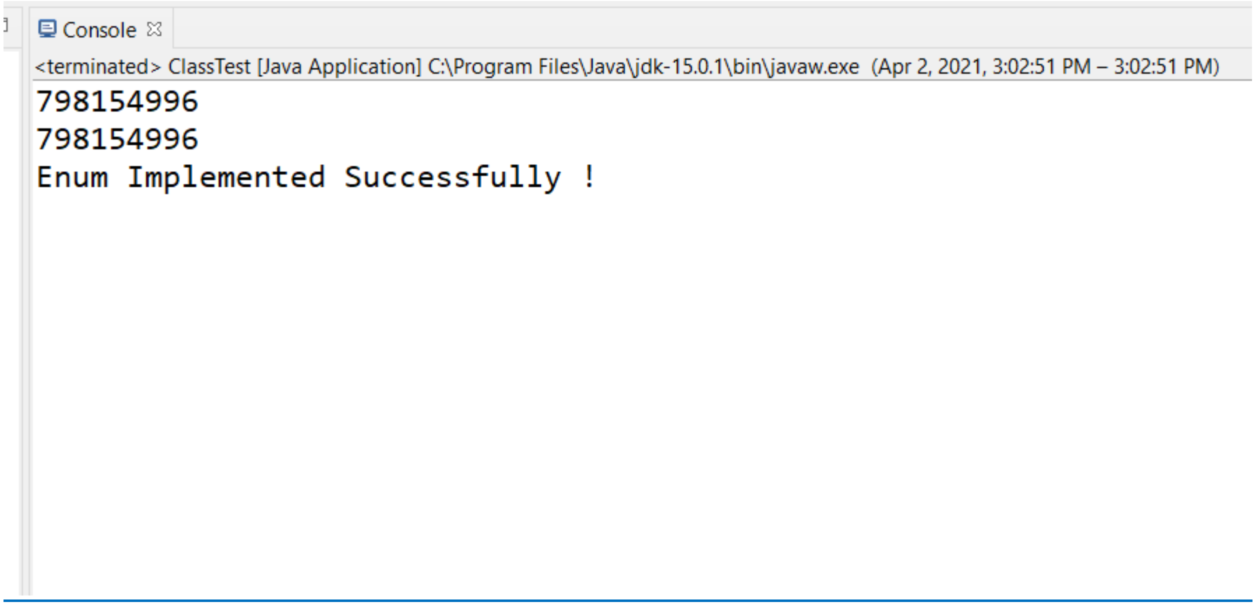
Step 1 :

```
package pack_enumDesign;

public enum Singleton
{
    Getinstance;

    public String printMessage()
    {
        return "Enum Implemented Successfully !" ;
    }

}
```

Step 2 :

```
package pack_enumDesign;

public class ClassTest
{
    public static void main(String[] args)
    {
        Singleton obj1 = Singleton.Getinstance;
        Singleton obj2 = Singleton.Getinstance;

        System.out.println(obj1.hashCode());
        System.out.println(obj2.hashCode());

        System.out.println(obj1.printMessage());
    }


}
```

Output :



```
<terminated> ClassTest [Java Application] C:\Program Files\Java\jdk-15.0.1\bin\javaw.exe  (Apr 2, 2021, 3:02:51 PM – 3:02:51 PM)
798154996
798154996
Enum Implemented Successfully !
```

❖ How to prevent cloning to break a Singleton Class Pattern :

- ✓ While creating a clone there is the chance of breakage of singleton pattern .
- ✓ For that we just need to throw exception of clone not supported from inside the overriden method clone.

Step 1:

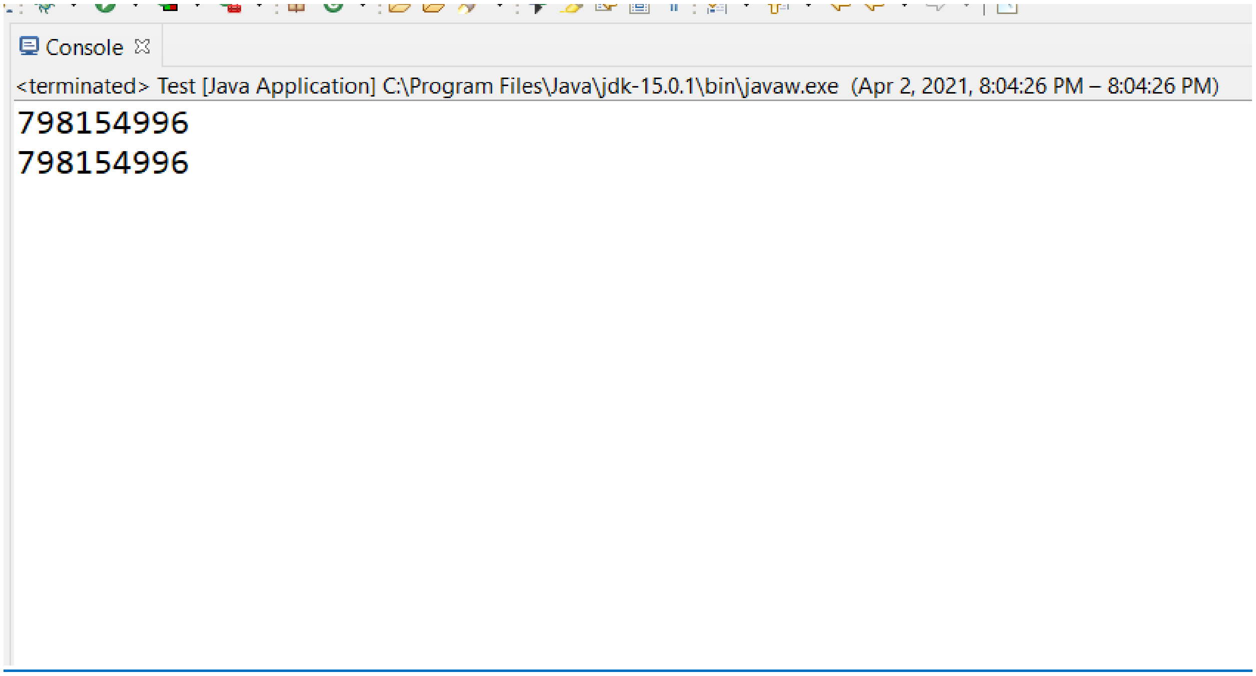
```
package pack_Singleton;

public class Singleton implements Cloneable
{
	public static Singleton instance = null;
	private Singleton()
```

```
    {

    }

    public static Singleton getinstance()
    {
        if(instance == null)
        {
            instance = new Singleton();
        }
        return instance;
    }
    @Override
    protected Object clone() throws
CloneNotSupportedException
    {
        throw new CloneNotSupportedException("You cannot
create clone of singleton");
        //return super.clone();
    }

}
```

Step 2:

```
package pack_Singleton;

import java.io.ObjectOutput;

public class Test {

    public static void main(String[] args)
    {
        Singleton obj1 = Singleton.getinstance();
        //Singleton obj2 = (Singleton)obj1.clone();
        Singleton obj2 = Singleton.getinstance();

        System.out.println(obj1.hashCode());
        System.out.println(obj2.hashCode());

    }

}
```

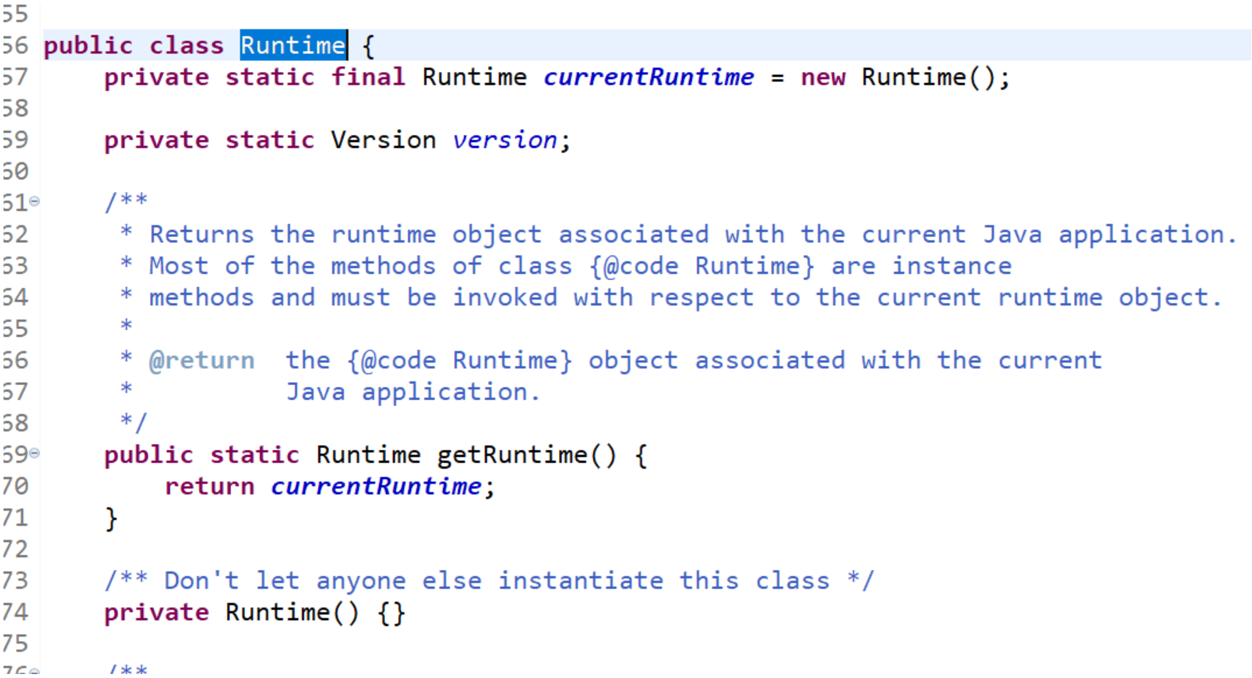
Output :



```
<terminated> Test [Java Application] C:\Program Files\Java\jdk-15.0.1\bin\javaw.exe  (Apr 2, 2021, 8:04:26 PM – 8:04:26 PM)
798154996
798154996
```

❖ Examples of Singleton Design Pattern in JDK :

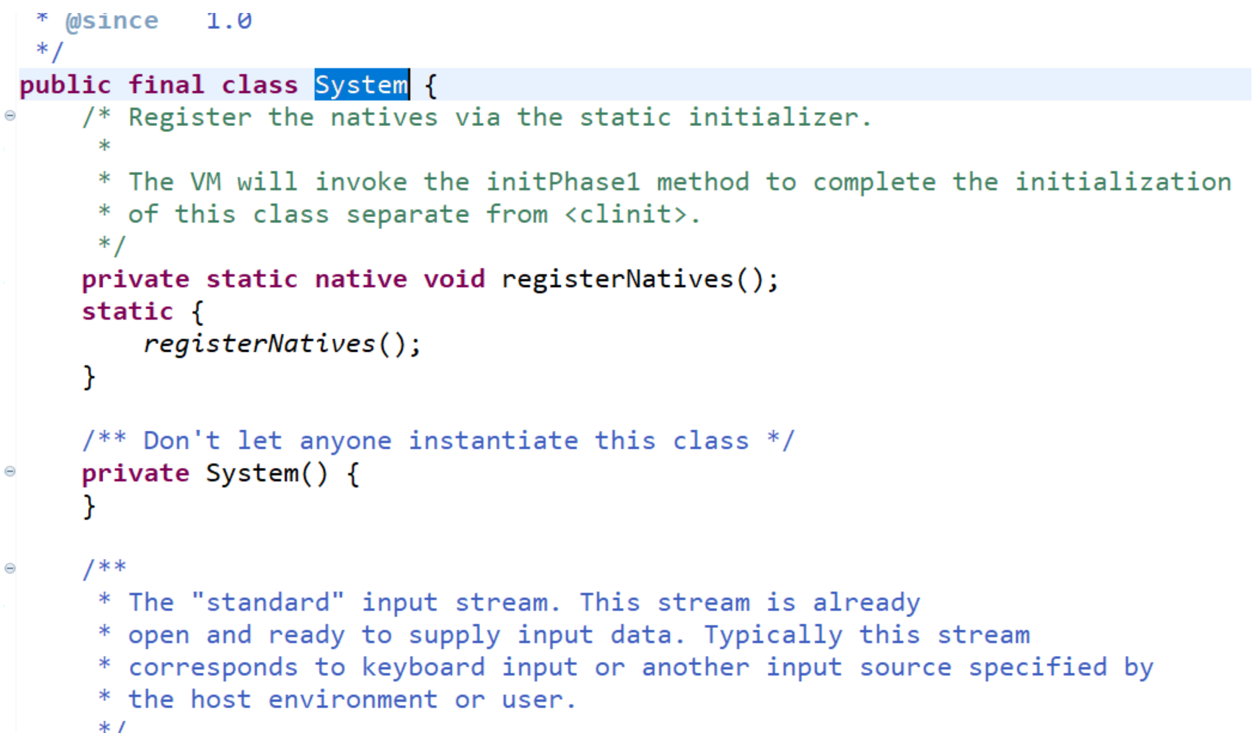
1). Runtime class



```java

public class Runtime {
    private static final Runtime currentRuntime = new Runtime();

    private static Version version;

    /**
     * Returns the runtime object associated with the current Java application.
     * Most of the methods of class {@code Runtime} are instance
     * methods and must be invoked with respect to the current runtime object.
     *
     * @return  the {@code Runtime} object associated with the current
     *          Java application.
     */
    public static Runtime getRuntime() {
        return currentRuntime;
    }

    /** Don't let anyone else instantiate this class */
    private Runtime() {}

    /**
```

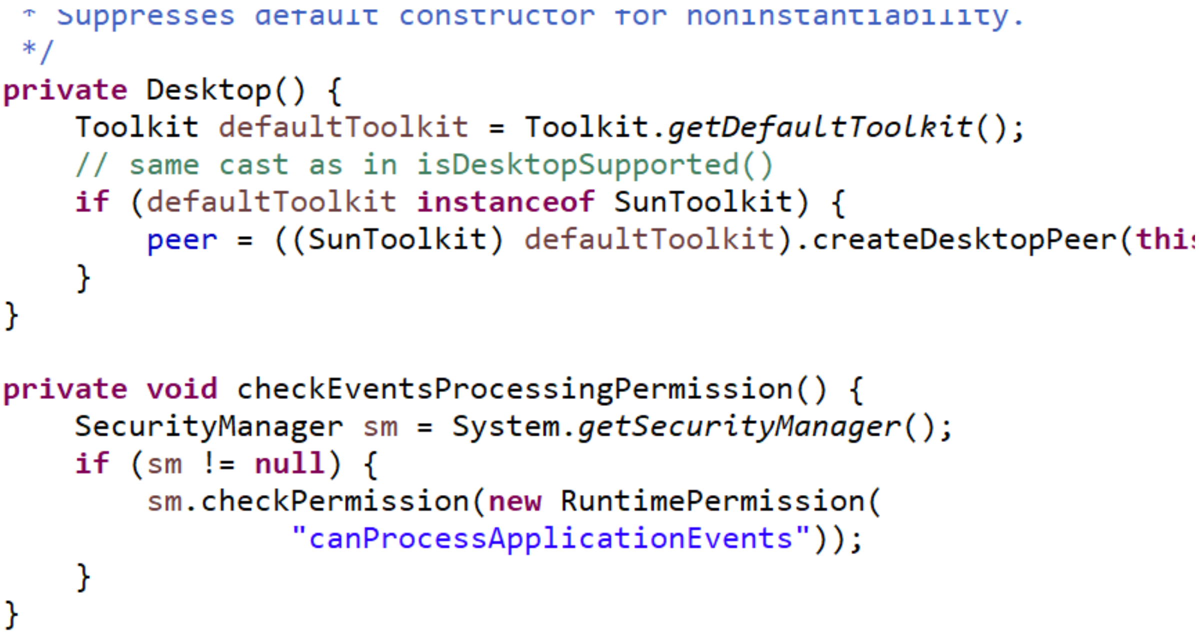
- **As you can see Runtime class is implementing singleton design pattern, instance of class is created while loading of class and it has private constructor.**

2). System class



```
 * @since   1.0
 */
public final class System {
    /* Register the natives via the static initializer.
     *
     * The VM will invoke the initPhase1 method to complete the initialization
     * of this class separate from <clinit>.
     */
    private static native void registerNatives();
    static {
        registerNatives();
    }

    /** Don't let anyone instantiate this class */
    private System() {
    }

    /**
     * The "standard" input stream. This stream is already
     * open and ready to supply input data. Typically this stream
     * corresponds to keyboard input or another input source specified by
     * the host environment or user.
     */
```

3). Desktop



```
     * Suppresses default constructor for noninstantiability.
     */
    private Desktop() {
        Toolkit defaultToolkit = Toolkit.getDefaultToolkit();
        // same cast as in isDesktopSupported()
        if (defaultToolkit instanceof SunToolkit) {
            peer = ((SunToolkit) defaultToolkit).createDesktopPeer(thi
        }
    }

    private void checkEventsProcessingPermission() {
        SecurityManager sm = System.getSecurityManager();
        if (sm != null) {
            sm.checkPermission(new RuntimePermission(
                    "canProcessApplicationEvents"));
        }
    }
```