There are two methods present in Reflection API which we can use to create objects

1- Class.newInstance() → Inside java.lang package

2- Constructor.newInstance() → Inside java.lang.reflect package

However, there are total 5 ways create objects in Java

1- Class.newInstance() :-

Below code demonstrates how we can create objects using `Class.newInstance()`

```
Employee emp = Employee.class.newInstance();
```

OR

```
Employee emp = (Employee) Class.forName("org.programming.mitra.exercises.Employee").newInstance();
```

2- Constructor.newInstance():-

In order to use Constructor.newInstance() method we first need to get constructor object for that class and then we can call newInstance() on it to create objects as shown below

```
Employee emp= null;
try        {
Constructor [] constructors = Employee.class.getDeclaredConstructors();
            for (Constructor constructor : constructors)             {
                // Below code will destroy the singleton pattern
//But this is also a way of creating object using reflection which will destroy the singleton pattern
              constructor.setAccessible(true);
emp = (Employee) constructor.newInstance();                 break;
            }
        } catch (Exception e)
         {
            e.printStackTrace();
        }
```

Difference between Class.newInstance() and Constructor.newInstance()

1. Class.newInstance() can only invoke the no-arg constructor,
Constructor.newInstance() can invoke any constructor, regardless of the number of parameters.

2. Class.newInstance() requires that the constructor should be visible,
Constructor.newInstance() can also invoke private constructors under certain circumstances.

3. Class.newInstance() throws any exception (checked or unchecked) thrown by the constructor,
Constructor.newInstance() always wraps the thrown exception with an InvocationTargetException.

Important: -

readResolve():-

For Serializable and Externalizable classes, the **readResolve** method allows a class to replace/resolve the object read from the stream before it is returned to the caller.

By implementing the readResolve method, a class can directly control the types and instances of its own instances being deserialized.

How to prevent Singleton Pattern from Reflection, Serialization and Cloning?

1-Reflection: - Reflection can be caused to destroy singleton property of singleton class

Example: -

Let's see how reflection can break the singleton pattern: -

```
import java.lang.reflect.Constructor;
class Singleton {
public static Singleton singleton;
private Singleton()     {    }
public static Singleton getInstance(){
If(singleton == null){
singleton = new  Singleton();
}
return singleton;
}
}
public class ReflectionDemo {
public static void main(String[] args)    {
Singleton object1 = Singleton.singleton;
Singleton object2 = null;
try        {
Constructor[] constructors = Singleton.class.getDeclaredConstructors();
for (Constructor constructor : constructors)
{
// Below code will destroy the singleton pattern
constructor.setAccessible(true);
object2 = (Singleton) constructor.newInstance();
break;
}
```

```
}catch (Exception e)
{
e.printStackTrace();
}
System.out.println("object1 .hashCode():- " + object1.hashCode());
System.out.println("object2.hashCode():- " + object2.hashCode());
}
}
```

Output: -

object1.hashCode(): - 636712542

object2.hashCode(): - 122163700

Explanation: -

As you can see that 2 different hashcodes are created for singleton class. Hence singleton pattern has been destroyed using Reflection.

Solution: -

To overcome above issue, we can throw a runtime exception in private constructor if singleton class is already initialized. Check below piece of code, inside Singleton class: -

```
class Singleton
{
   public static Singleton singleton;
   private Singleton ()
   {
        if (singleton! = null) {
           throw new RuntimeException("Do not try to break the Singleton class");
        }
   }
   public static Singleton getInstance(){
      If (singleton == null) {
         singleton = new Singleton ();
      }
   return singleton;
   }
}
```

2-Serialization: - Serialization can also cause breakage of singleton property of singleton classes. Serialization is used to convert an object of byte stream and save in a file or send over a network. Suppose you serialize an object of a singleton class. Then if you de-serialize that object it will create a new instance and hence break the singleton pattern.

// Java code to explain effect of Serilization on singleton classes

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInput;
import java.io.ObjectInputStream;
import java.io.ObjectOutput;
import java.io.ObjectOutputStream;
import java.io.Serializable;

class Singleton implements Serializable
{
   // public instance initialized when loading the class
   public static Singleton instance = new Singleton ();
    private Singleton ()
   {
      // private constructor
   }
// implement readResolve method (This is the solution of the issue just we need to add this method)
   protected Object readResolve()
   {
           return instance;
   }
}
 public class GFG {
   public static void main (String [] args)
   {
      try {
         Singleton instance1 = Singleton.instance;
```

```
        ObjectOutput out = new ObjectOutputStream(new FileOutputStream("file.text"));
        out.writeObject(instance1);
        out.close();
        // deserailize from file to object
        ObjectInput in = new ObjectInputStream(new FileInputStream("file.text"));
        Singleton instance2 = (Singleton) in.readObject();
        in.close();
        System.out.println("instance1 hashCode:- "+ instance1.hashCode());
        System.out.println("instance2 hashCode:- "+ instance2.hashCode());
    }
        catch (Exception e) {e.printStackTrace(); }
  }
}
```

Output: -

instance1 hashCode:- 1550089733

instance2 hashCode:- 865113938

As you can see, hashCode of both instances is different, hence there are 2 objects of a singleton class. Thus, the class is no more singleton.

Overcome serialization issue: - To overcome this issue, we have to implement method readResolve() method.

// Just need to add this method implementation. I have added this same in above program only.

```
// implement readResolve method
    protected Object readResolve()
    {
        return instance;
    }
```

3- Cloning: - Cloning is a concept to create duplicate objects. Using clone, we can create copy of object. Suppose, we create clone of a singleton object, then it will create a copy that is there are two instances of a singleton class, hence the class is no more singleton.

```
// JAVA code to explain cloning
// issue with singleton
class SuperClass implements Cloneable
{
  int i = 10;
  @Override
```

```
  protected Object clone () throws CloneNotSupportedException
  {
    return super.clone();
  }
}

// Singleton class
class Singleton extends SuperClass
{
  // public instance initialized when loading the class
  public static Singleton instance = new Singleton ();
  private Singleton ()
  {
    // private constructor
  }
}

public class GFG
{
  public static void main (String [] args) throws CloneNotSupportedException
  {
    Singleton instance1 = Singleton.instance;
    Singleton instance2 = (Singleton) instance1.clone();
    System.out.println("instance1 hashCode:- "+ instance1.hashCode());
    System.out.println("instance2 hashCode:- "+ instance2.hashCode());
  }
}
```

Output: -

```
instance1 hashCode:- 366712642
instance2 hashCode:- 1829164700
```

Two different hashCode means there are 2 different objects of singleton class.

Overcome Cloning issue: - To overcome this issue, override clone () method and throw an exception from clone method that is CloneNotSupportedException. Now whenever user will try to create clone of singleton object, it will throw exception and hence our class remains singleton.

```
@Override
```

```
protected Object clone () throws CloneNotSupportedException
{
  throw new CloneNotSupportedException();
}
```