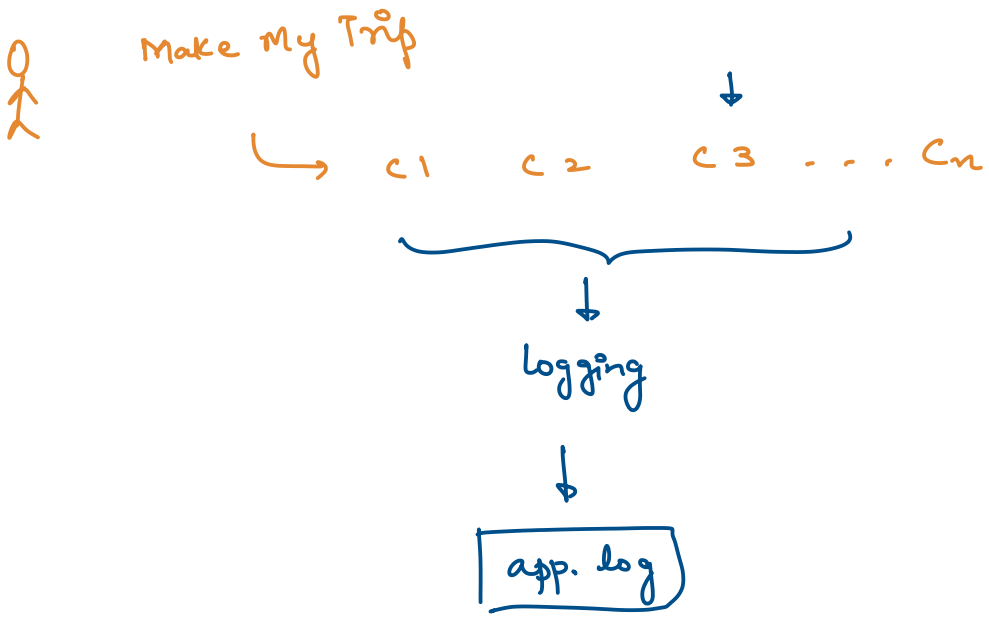
10__31-07-2022

Sunday, 31 July 2022 8:03 PM

Singleton design pattern

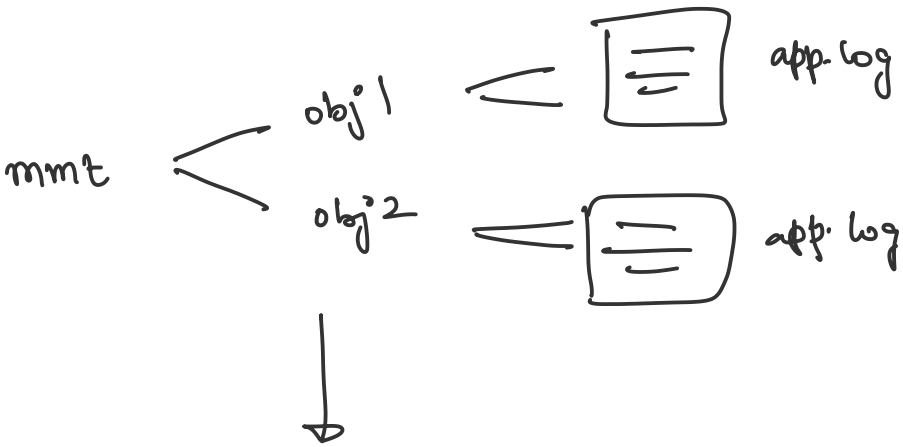
↳ Majorly used when you want a single instance of a given class.

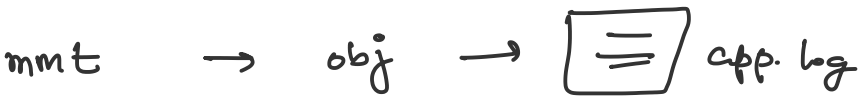


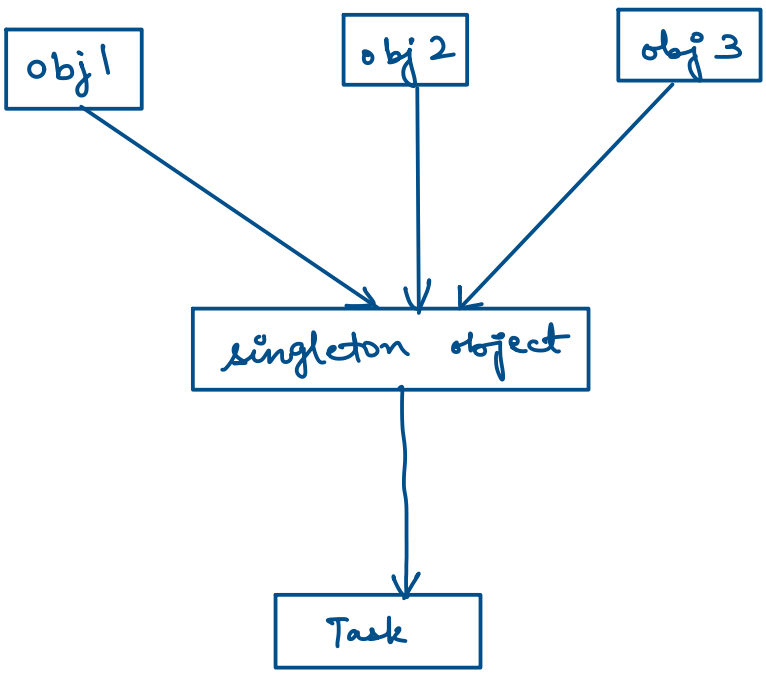
↳ It belongs to creational type pattern.

↳ Dealing with object creation mechanism.

↳ Used when we want to have only one object of a particular class is instantiated.







This single instance of the object is

responsible for invoking all the methods or events.

How to create singleton?

↳ Make the constructor private

↳ Helps in restricting object creation outside of the class.

```
public class A
{
    public A()
    {
        ==
    }
}
```

```
A obj = new A();
```

Private constructor

```
public class A
{
    private A()        → private constructor
    {
        ==
    }
}
```

Note –

We cannot inherit a class with private constructor. However we can have nested classes.

```
public class A
{
    private A()
    {   =
    }
}
```

```
public class B extends A
```

```
public class B extends A
{

}
```

This is not allowed,

this will give error.

In nested classes —

```
public class A
{
    private A()
    {
      ==
    }

    public class B
    {
      ==
    }
}
```

This is allowed.

↳ Responsibility of creation of the object should be taken by the class

itself.

This can achieved by creating a private attribute of the class type that refers to the single object.

```
public class A
{
    private A()
    {
        =
    }

    private static A instance = null;
}
```

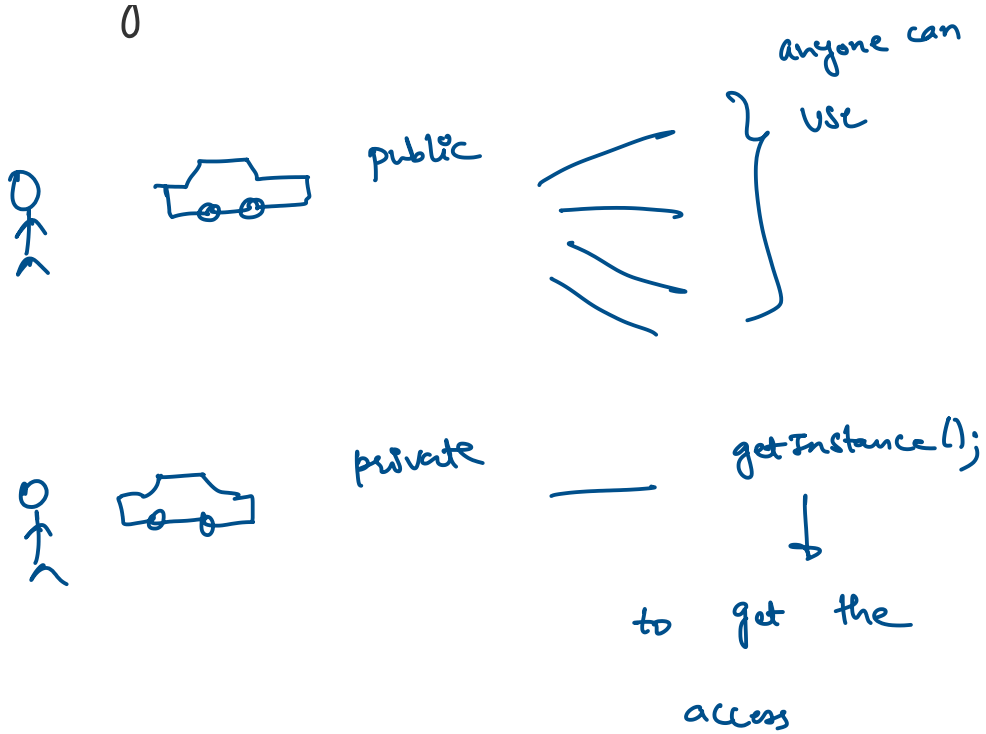
↳ Create a static method that allows us to create and access the object we created.

```
public static A getInstance()
{
        return instance;
}
```

For a non static method, in order to access it outside the class —

```
{ A obj = new A(); }
  obj.method();
```

But here we cannot instantiate the class because of private constructor, so we cannot call it using the above approach, thats why we creating a static method.




```
public class SingletonExample {
   private SingletonExample()
   {
      System.out.println("Private Constructor");
   }

   private static SingletonExample instance = new SingletonExample();

   public static SingletonExample getInstance()
   {
      return instance;
   }

   public void printMessage(String message)
   {
      System.out.println("Message : " + message);
   }
}
```

```
public class Program {
   public static void main(String[] args) {
      SingletonExample se = SingletonExample.getInstance();
      se.printMessage("Hello world");
   }
```

```
}
```

Output:

```
Private Constructor
Message : Hello world
```

Another example:

```
public class Demo {
   public Demo()
   {
      System.out.println("Demo : constructor");
   }

   public void printDemo()
   {
      System.out.println("Print demo");
   }
}
```

```
public class SingletonExample {
   private SingletonExample()
   {
      System.out.println("Private Constructor");
   }

   private static SingletonExample instance = new SingletonExample();

   public static SingletonExample getInstance()
   {
      return instance;
   }
```

```
    public void printMessage(String message)
    {
        System.out.println("Message : " + message);
    }

    public static void printStaticMessage(String message)
    {
        System.out.println("Static Message : " + message);
    }
}
```

```
public class Program {
    public static void main(String[] args) {
        //multiple instances of Demo class
        Demo obj1 = new Demo();
        Demo obj2 = new Demo();
        Demo obj3 = new Demo();
        obj1.printDemo();
        System.out.println();

        //single instance of SingletonExample class
        SingletonExample se1 = SingletonExample.getInstance();
        SingletonExample se2 = SingletonExample.getInstance();
        SingletonExample se3 = SingletonExample.getInstance();
        se1.printMessage("Sample message for a non static method");
        SingletonExample.printStaticMessage("Static message
example");
    }
}
```

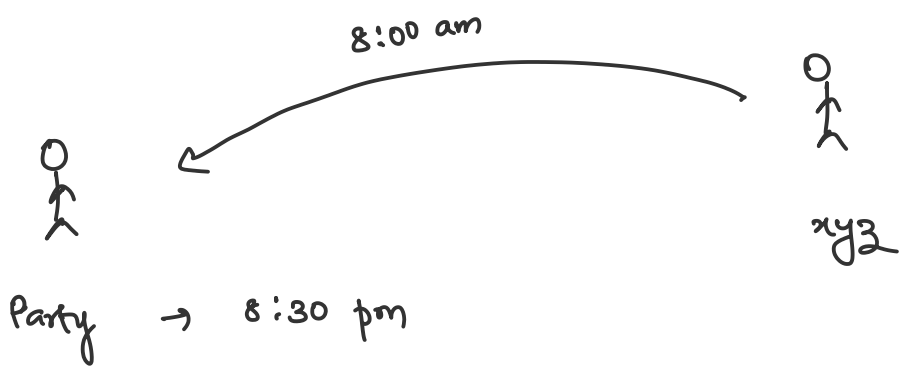
Output:

```
Demo : constructor
Demo : constructor
Demo : constructor
Print demo

Private Constructor
Message : Sample message for a non static method
Static Message : Static message example
```

Singleton class in java could be realized in 4 ways—

1. Early Loading (Eager loading)
2. Lazy Loading (On demand loading)
3. Lazy loading - thread safe
4. Enum type.

Early loading



Eager (Early) loading helps you to load

all your needed entities at once.

Example for same — Above example

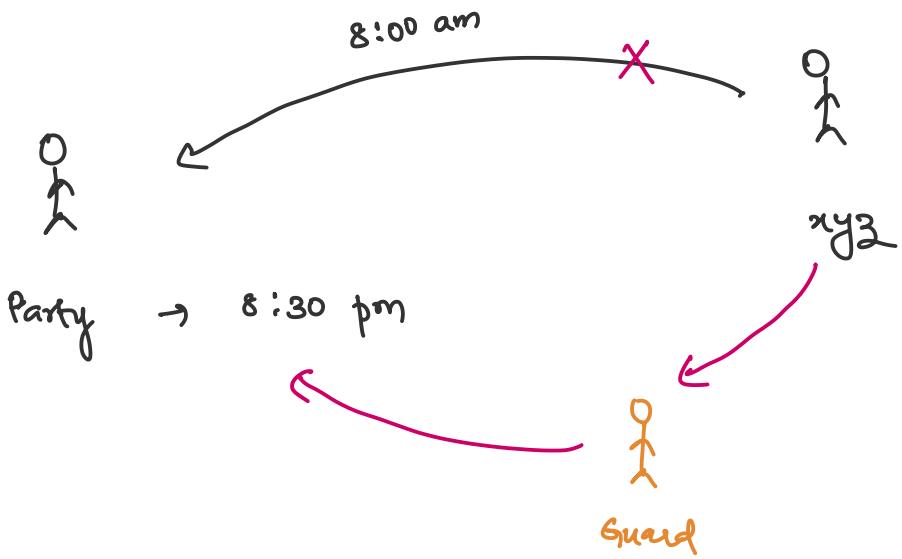
Eager loading is nothing but to initialize the required object before it is being accessed, which means we instantiate the object and keep it ready and use it when needed it.

This type of initialization is used in lower memory footprints.

In early loading, the common language runtime takes care of the variable initialization and its thread safety, so we don't need to write

any explicit coding for thread safety.

Lazy or on demand initialization



Here we will be delaying instance creation till the getInstance () method is invoked, this delayed instance creation is called as lazy initialization.

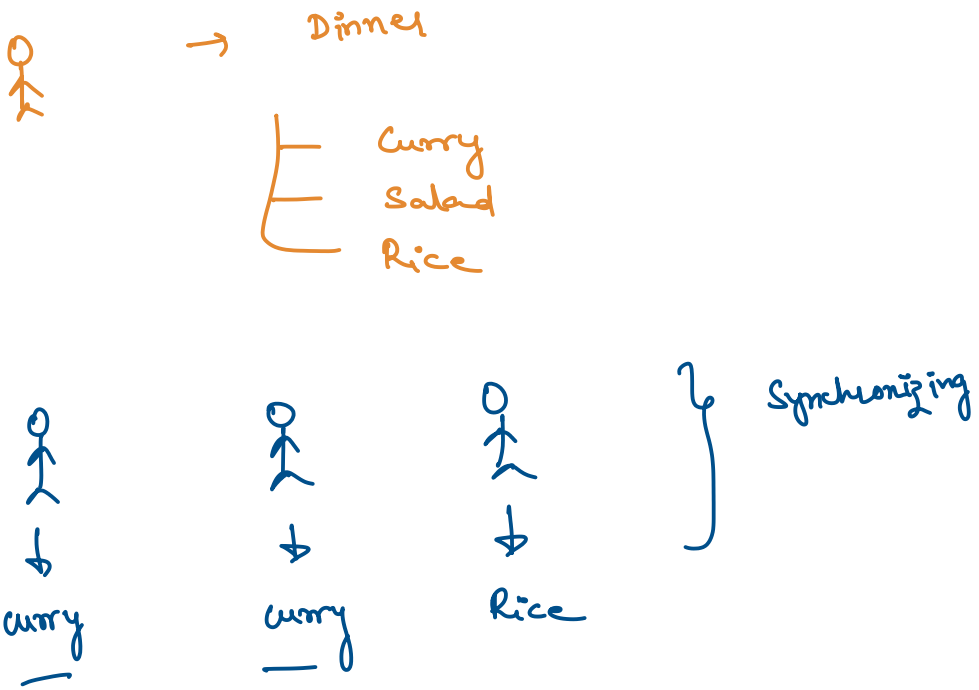
```
public class SingletonExample {
   private SingletonExample()
   {
      System.out.println("Private Constructor");
   }

   private static SingletonExample instance = null;

   public static SingletonExample getInstance()
   {
      if(instance == null)
      {
         instance = new SingletonExample();
      }

      return instance;
   }

   public void printMessage(String message)
   {
      System.out.println("Message : " + message);
   }

   public static void printStaticMessage(String message)
   {
      System.out.println("Static Message : " + message);
   }
}
```

```
public class Program {
   public static void main(String[] args) {
      SingletonExample se1 = SingletonExample.getInstance();
      SingletonExample se2 = SingletonExample.getInstance();
      SingletonExample se3 = SingletonExample.getInstance();
      se1.printMessage("Sample message for a non static method");
      SingletonExample.printStaticMessage("Static message example");
   }
}
```

Output:
Private Constructor
Message : Sample message for a non static method
Static Message : Static message example

Java — multi-threaded language
↳ multiple threads

↳ By default it is single thread
↳ works in main method.



The previous example (lazy loading) works perfectly fine in single threaded environment. But when you have the

situation of multiple threads then there is a chance that we can create multiple instances of this singleton object.