11__14-08-2022

Sunday, 14 August 2022 8:07 PM

singleton design pattern

↳ one object or instance for a given class.

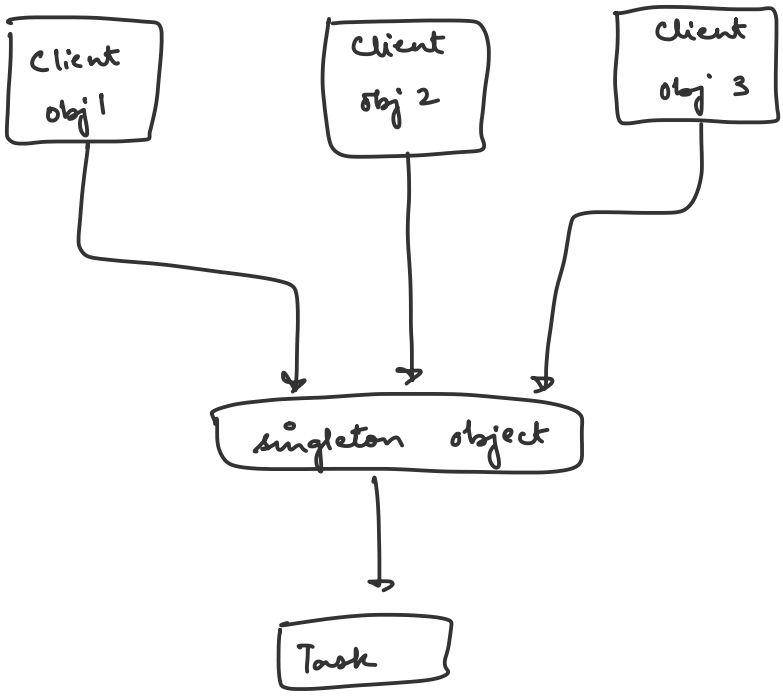
App^{n} – C_1 C_2 C_3 - - - - C_{100}

↓

logging

↓

app.log



To create a singleton class —

1. Create private constructor
2. Create a private static attribute of the class type that refers to the single object.
3. Create a public static method that allows us to create and access the object we created.

Singleton class can be realized in one of the following ways —

(1) Early loading
(2) Lazy loading
(3) Lazy loading - thread safe
(4) Enum type.

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

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

public class Program {
   private static void printFirstMessage()
   {
      SingletonExample obj1 = SingletonExample.getInstance();
      obj1.print("Message from obj 1");
   }

   private static void printSecondMessage()
   {
      SingletonExample obj3 = SingletonExample.getInstance();
      obj3.print("Message from obj 2");
   }

   private static void printThirdMessage()
   {
```

```
        SingletonExample obj1 = SingletonExample.getInstance();
        obj1.print("Message from obj 3");
    }

    public static void main(String[] args) {
        ExecutorService executorService = Executors.newCachedThreadPool();
        Runnable createFirstObj = () -> printFirstMessage();
        executorService.execute(createFirstObj);

        Runnable createSecondObj = () -> printSecondMessage();
        executorService.execute(createSecondObj);

        //inline
        executorService.execute(() -> printThirdMessage());

        executorService.shutdown();
    }
}
```

Output:
Private constructor
Private constructor
Message : Message from obj 1
Message : Message from obj 2
Message : Message from obj 3

The lazy loading approach works fine in the single threaded environment but in multi threaded environment it violates singleton concept.

Thread safety in singleton class

(1) Create the instance variable at the time of class loading.

↓

Early loading.

Pros

→ Thread safety without synchronization.

→ Easy to implement.

Cons -

→ Early creation of resources that might not be used in the appⁿ.

→ The client appⁿ cannot pass any argument, so we cannot reuse it.

(2) Synchronize the getInstance() method.

Pros –

→ Thread safety is guaranteed.

→ Lazy loading achieved.

→ client app^n can pass parameters.

Cons –

→ slow performance because of locking overhead.

→ unnecessary synchronization that is not required once the instance variable is initialized.

(3) Use synchronized block inside the if- loop and volatile variables.

Pros -

→ Thread safety is guaranteed.

→ Lazy loading achieved

→ Client app^n can pass parameters, so we can reuse the code.

→ Synchronization overhead is minimal and applicable only for the first few threads when the variable is null.

Cons -

→ Extra if condition.

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

   private static volatile SingletonExample instance;
   private static Object object = new Object();

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

      return singletonExample;
   }

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

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

public class Program {
   private static void printFirstMessage()
   {
      SingletonExample obj1 = SingletonExample.getInstance();
      obj1.print("Message from obj 1");
   }

   private static void printSecondMessage()
   {
```

```
        SingletonExample obj3 = SingletonExample.getInstance();
        obj3.print("Message from obj 3");
    }

    private static void printThirdMessage()
    {
        SingletonExample obj1 = SingletonExample.getInstance();
        obj1.print("Message from obj 1");
    }

    public static void main(String[] args) {
        ExecutorService executorService = Executors.newCachedThreadPool();
        Runnable createFirstObj = () -> printFirstMessage();
        executorService.execute(createFirstObj);

        Runnable createSecondObj = () -> printSecondMessage();
        executorService.execute(createSecondObj);

        //inline
        executorService.execute(() -> printThirdMessage());

        executorService.shutdown();
    }
}
```

Output:
Private constructor
Message : Message from obj 3
Message : Message from obj 2
Message : Message from obj 1

Note -

"singleton Example" instance is there to improve the performance of the code. In cases

where the instance is already initialized which will be happening most of the times then the volatile field is only accessed once because of the statement -

Either — return singletonExample;

or — return instance;

Singleton class using Enum type

To create instance use ENUM.

```
public enum SingletonEnum
{
    Instance;
}
```

Note -

Enums are public static final fields.

This is the behavior of the class instances required by the design of singleton. This is thread safe.

```
public enum SingletonEnum {
   Instance;

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

```
public class Program {
   public static void main(String[] args) {
      SingletonEnum.Instance.print("Message from obj 1");
      SingletonEnum.Instance.print("Message from obj 2");
      SingletonEnum.Instance.print("Message from obj 3");
   }
}
```

Output:
Message : Message from obj 1
Message : Message from obj 2
Message : Message from obj 3