... continued

This lesson continues the discussion on implementing the Singleton pattern in Java.

Implementing a thread-safe Singleton class is a popular interview question with the double checked locking as the most debated implementation. For completeness, we present below all the various evolutions of the Singleton pattern in Java.

- The easiest way to create a singleton is to mark the constructor of the class private and create a private static instance of the class that is initialized inline. The instance is returned through a public getter method. The drawback of this approach is if the singleton object is never used then we have spent resources creating and retaining the object in memory. The static member is initialized when the class is loaded. Additionally, the singleton instance can be expensive to create and we may want to delay creating the object till it is actually required.

  Singleton eager initialization

  ```
  public class Superman {
      private static final Superman superman =  = new Superman();

      private Superman() {
      }

      public static Superman getInstance() {
          return superman;
      }
  }
  ```

  A variant of the same approach is to initialize the instance in a static block.

Singleton initialization in static block

```
public class Superman {
    private static Superman superman;

    static {
        try {
            superman = new Superman();
        } catch (Exception e) {
            // Handle exception here
        }
    }

    private Superman() {
    }

    public static Superman getInstance() {
        return superman;
    }
}
```

The above approach is known as **Eager Initialization** because the singleton is initialized irrespective of whether it is used or not. Also, note that we don't need any explicit thread synchronization because it is provided for free by the JVM when it loads the `Superman` class.

```
class Demonstration {
    public static void main( String args[] ) {
        Superman superman = Superman.getInstance();
        superman.fly();
    }
}

class Superman {
    private static Superman superman = new Superman();

    private Superman() {
    }

    public static Superman getInstance() {
        return superman;
    }

    public void fly() {
      System.out.println("I am flyyyyingggggg ...");
    }
```

```
}
}
```




- The next approach is to lazily create the singleton object. **Lazy intialization means delaying creating a resource till the time of its first use.** This saves precious resources if the singleton object is never used or is expensive to create. First, let's see how the pattern will be implemented in a single threaded environment.

  Singleton initialization in static block

  ```
  public class Superman {
      private static Superman superman;

      private Superman() {
      }

      public static Superman getInstance() {

          if (superman == null) {
              superman = new Superman();
          }

          return superman;
      }
  }
  ```

  With the above approach we are able to introduce lazy initialization, however, the class isn't thread-safe. Also, we needlessly check if the instance is null every time we invoke `getInstance()` method.

  To make the above code thread safe we synchronize the `getInstance()` method and get a thread-safe class.

  Thread safe

  ```
  public class Superman {
      private static Superman superman;

      private Superman() {
  ```

```
    }

    public synchronized static Superman getInstance() {

        if (superman == null) {
            superman = new Superman();
        }

        return superman;
    }
}
```

Note that the method is synchronized on the class object. The problem with the above approach is we are serializing access for threads even after the singleton object is safely initialized the first time. This slows down performance unnecessarily. The next evolution is to move the lock inside of the method.

```
class Demonstration {
    public static void main( String args[] ) {
        Superman superman = Superman.getInstance();
        superman.fly();

    }
}

class Superman {
    private static Superman superman;

    private Superman() {
    }

    public synchronized static Superman getInstance() {

        if (superman == null) {
            superman = new Superman();
        }

        return superman;
    }

    public void fly() {
      System.out.println("I am flyyyyingggggg ...");
    }
}
```

- The next evolution is the double checked locking that we have already discussed in the previous lesson.
- Another implementation of the singleton pattern is the **holder** or **Bill Pugh's** singleton. The idea is to create a private nested static class that holds the static instance. The nested class `Helper` isn't loaded when the outer class `Superman` is loaded. The inner static class `Helper` is loaded only when the method `getInstance()` is invoked. This saves us from eagerly initializing the singleton instance.

```
class Demonstration {
    public static void main( String args[] ) {
        Superman superman = Superman.getInstance();
        superman.fly();
    }
}

class Superman {

    private Superman() {
    }

    private static class Holder {
        private static final Superman superman = new Superman();
    }

    public static Superman getInstance() {
        return Holder.superman;
    }

    public void fly() {
        System.out.println("I am flyyyyingggggg ...");
    }
}
```