Singleton

A creational pattern

Learning goals

1. Learn the idea, structure, and Java implementation of the Singleton design pattern.
2. Learn to apply the Singleton DP in your own programming.

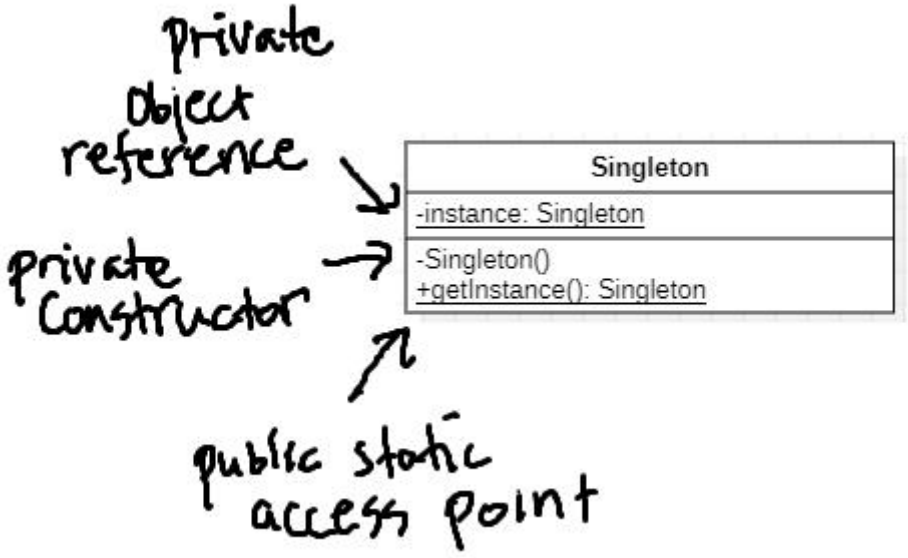
Idea of Singleton

- The Singleton design pattern guarantees that an object is created once and only once in the lifespan of the application.
- The usage of the **new** operator is prohibited by deliberately hiding the constructor.
- A dedicated method takes care of the object construction.
  - The method verifies that the object does not pre-exist.
  - If it does, then a reference to the existing object is returned.

Why only one object?

1. Resource management and performance
   - For example, establishing a database connection is a slow process. Each connection should be established only once.
2. Globally accessible single point of control
   - Provides a single, uniform way to always get access to an object.
3. Consistency
   - Prevents accidentally creating many objects that who might have conflicting states.
     - E.g. two clocks or timers
4. Thread safety
   - Provides a way to guarantee only one object is created even if the application is run in many threads.
5. Deferring object creation
   - Makes it possible to create an object only at the point when it is really needed.

General structure



- In addition, the Singleton class contains the business operations that it was designed for.
- The private constructor overrides the default public constructor.
  - Prevents **new** from outside the Singleton class

Java implementation

```
public class Singleton {
    private static Singleton instance;
    private Singleton() {
    }

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

    // business logic for this class
}
```

Object creation by client

```
public class Main {

    public static void main(String[] args) {

        // right
        Singleton s = Singleton.getInstance();

        // wrong
        Singleton s2 = new Singleton();

    }
}
```

fails as no public constructor available

Roles

- **Singleton**: defines a **getInstance()** method that lets the client access the reference to the object.
  - If necessary, the **getInstance()** method will create a new instance of the class.

Drawbacks

- Singleton is one of the more controversial GoF design patterns.
- Reasons for criticism:
  1. Global Context
     - Adds global state to the application. Less encapsulation.
  2. Tight Coupling
     - An object can be accessed from anywhere.
       - less modularity
       - harder to maintain.
  3. Testing Challenges
     - Harder to write unit tests, as there is a single shared instance.
  4. Potential Misuse
     - Sloppy use may lead to complex dependencies and difficulties in maintaining code consistency.

Thread-safety

- The Singleton DP provides a relatively easy way to make sure that only one object is created, irrespective of the number of threads.
  - Add the **synchronized** keyword to the **getInstance()** method:

```
public static synchronized Singleton getInstance() {
   if (instance == null) {
      instance = new Singleton();
   }
   return instance;
}
```

Thread-safety (another option)

- In Java, an alternative approach is to generate the instance as the class is loaded.
- This is guaranteed to happen once per application.
  - In the beginning of the execution, or
  - When the class is first used.

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

   public static synchronized Singleton getInstance() {
      return instance;
   }

   // business logic for this class
}
```

The enum ’trick’

- In Java, olso the enumerated type (Enum) can be cleverly used to implement the Singleton design pattern.
- It is guaranteed that each enum is initialized only once.

```
public enum Singleton {
   INSTANCE;

   // Business logic:
   public void performOperation() {
      System.out.println("SingletonEnum performing operation.");
   }
}

public class Main {
   public static void main(String[] args) {
      Singleton.INSTANCE.performOperation();
   }
}
```