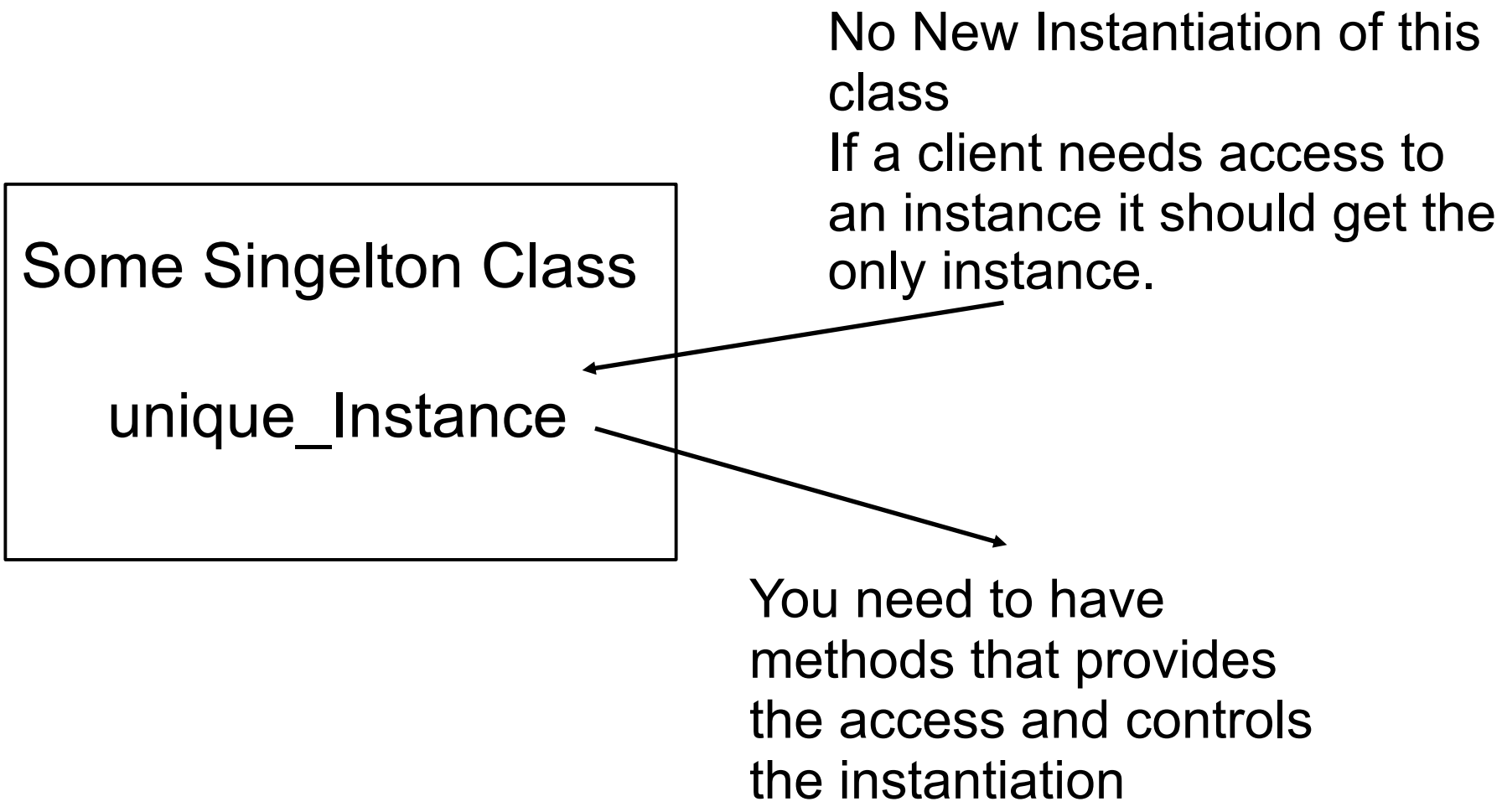
Singleton Pattern

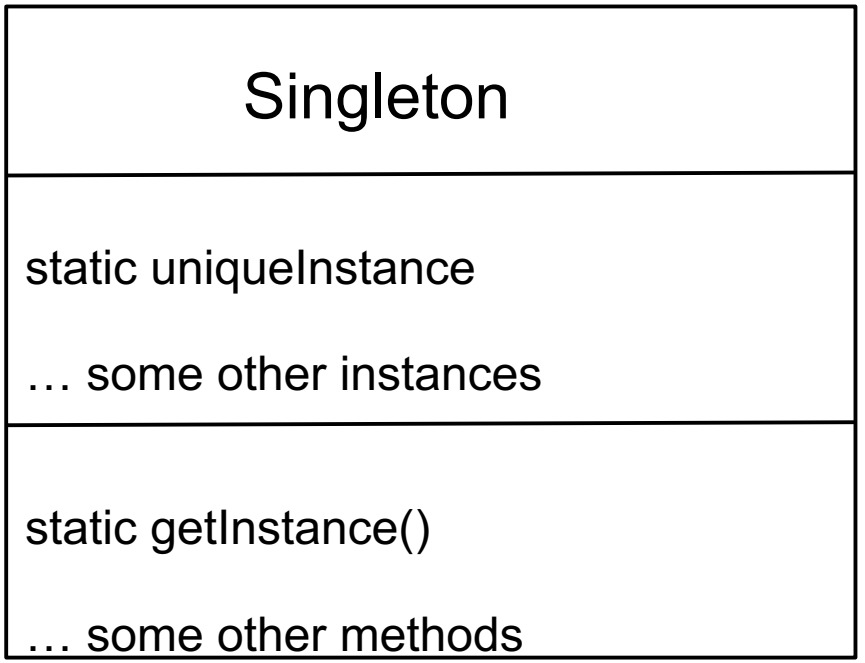
Problem – Single Instance of a Class

- You want to have only one instance of class.
- You want to restrict object instantiation of a specific class.



Definition of Singleton Pattern

- “The Singleton Pattern ensures you have at most one instance of a class in your application.”



Implementation

- We make constructor private
- We use static method

```
public class Singleton {
        private static Singleton uniqueInstance;

        private Singleton() {}

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

        // other methods here ...


        public String getDescription() {
                return "I did a classic Singleton Implementation!";
        }
}
```

Using Static variables and static initialization

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

        private Singleton() {}

        public static Singleton getInstance() {
                return uniqueInstance;
        }

        // other methods here . . .
        public String getDescription() {
                return "Using statical initialization for Singleton!";
        }
}
```

Participants

Only one participant

- **Singleton**
  - It defines an Instance operation that lets clients (main or other classes) access its unique instance (getInstance() method) . Instance is a class operation, in Java it is a static class.
  - It may be responsible for creating its own unique instance.

When use/Not Use the Singleton Pattern

- You can use Singleton pattern when
  - there **must be exactly one instance** of a class, and it must be accessible to clients from a well-known access point.
  - Or you want to have n number of instances only.
  - when the single instance should be extensible by sub-classing, and clients should be able to use an extended instance without modifying their code.

Consequences

- **Controlled access to sole instance.**
  - Singleton class encapsulates its only single instance, it can have strict control over how and when clients access it.
- **Avoids polluting the code with global variables.**
  - The Singleton pattern is an improvement over global variables. It avoids polluting the name space with global variables that store sole instances.
- **Permits refinement of operations and representation.**
  - The Singleton class may be subclassed, and it's easy to configure an application with an instance of this extended class. You can configure the application with an instance of the class you need at run-time (combination with strategy pattern).

Consequences

- **Permits a variable number of instances.**
  - It is possible to change your mind in future and allow more than one instance of the Singleton class.
  - You can control the number of instances that the application uses. Only the operation that grants access to the Singleton instance needs to change.
- **More flexible than class operations.**
  - Another way to package a singleton's functionality is to use class operations, like static methods/functions and static variables.
- In general, not always a good idea to restrict object instantiation.

Threads – Refreshing …

- Threads?
  - Threads (or thread execution) are a sequence of computation tasks that can be run on a computer operating system.
  - A program can be divided into multiple subsequences that can be run simultaneously to achieve parallelism of computations and so better performance of running tasks.

Threads – Refreshing …

- In Java:

```
class MyThread extends Thread { …
    public void run() {  // here define your computation … }  }
    MyThread p = new MyThread();
                p.start();
```

- Or you can implement the Runnable interface

```
class MyClassRun implements Runnable { … }
    MyClassRun p = new MyClassRun();
        new Thread(p).start();
```

Thread-Safe Implementation

- In Java you can use the synchronized keyword

```
public class Singleton {
        private static Singleton uniqueInstance;
        // other instance variables here ...
        private Singleton() {}

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

        // other methods here . . .
        public String getDescription() {
                return "I'm a thread safe Singleton!";
        }
}
```

Thread-Safe Implementation

```
// Note: this implementation might work prior to Java 5

public class Singleton {
        private volatile static Singleton uniqueInstance;

        private Singleton() {}

        public static Singleton getInstance() {
                if (uniqueInstance == null) {
                        synchronized (Singleton.class) {
                                if (uniqueInstance == null) { // check again – Why?
                                        uniqueInstance = new Singleton();
                                }
                        }
                }
                return uniqueInstance;
        } }
```

- The **volatile** keyword ensures that multiple threads handle the uniqueInstance variable correctly. Java volatile keyword guarantees visibility of changes to variables across threads.
- We only synchronize the first time when the sole instance is not instantiated
- The static method is not synchronized.

Multiton Pattern

- Singleton Pattern but with more than one single object
- Restrict the object creation to maximum **n** different number of objects
- With N=1 , Multition becomes Singleton

Summary of Singleton Pattern

- The Singleton pattern is the simplest in terms of its class diagram.
- Implementation of thread-safe singleton pattern requires synchronized access to the shared sole object.

- Singleton pattern is one of the "**Creational Patterns".**
- Creational Patterns that we learned so far:
  - Factory Method
  - Abstract Factory
  - Singleton