Singleton

Goal
Ensure a class has only one instance, and provide a global point of access to it.

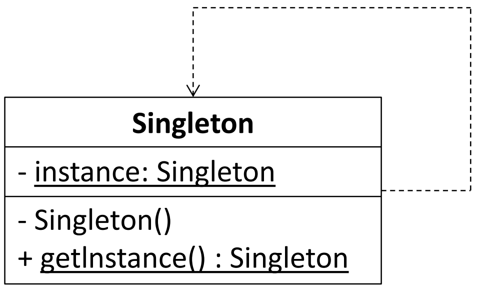
Motivation
- Cache-implementation
- Objects which manage registry configurations or preferences
- Thread pool
- Class that can be used to play MP3 files. If a new file can be played while a file is being played, the behavior is unpredictable.
  => Solution: only one instance that coordinates the play of files.
- Driver (for a printer or a database).
  When implementing drivers, global invariants must be ensured, e.g. that a database is not accessed at the same time from several threads.
- Communication across computer boundaries:
  Data which is sent over a socket must be synchronized

First Approach:
Declare a class with only static variables and static methods. A class exists only once per class loader.

Question:
What are the disadvantages of this approach?
Hint: If a complex initialization were necessary, where and how would this be done?
There are other disadvantages beyond initialization!

Let us look at a solution that uses a single real object:

Structure



Implementation

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

    public static Singleton getInstance() {
         return instance;
    }

    private Singleton() {}
}
```

Examples:

`java.lang.Runtime`	(1 instance, represents the system on which the JVM is running)
`java.lang.Class`	(n instances, n = number of loaded classes)

Questions:

- The constructor is declared as `private`.
  What are the problems when it were declared `protected`?

  Remark:
  If the constructor is declared `private`, then the class could also be declared as `final`.

- Specify how to implement methods `equals`, `hashCode` and `clone` in a singleton class.

Remark: If the object is serializable, then copies can be created by reading new instances.
→ `readResolve`! More information in http://www.javalobby.org/java/forums/t17491.html

Lazy initialization and synchronization

A Singleton implementation may look like this:

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

Access to the Singleton instance must be synchronized, otherwise several objects could be created. Create a sequence of statements where two threads "at the same time" try to create a singleton and act so unhappily that two instances are created at the end. Just look at method `getInstance()`.

Thread 1	*Thread 2*	*Value von instance*	*Time*
`public static Singleton getInstance() {`		`null`	↓
	`public static Singleton getInstance() {`	`null`	

How does a "thread-safe" solution of the lazy initialization variant look like?