What is a design pattern?

- Wikipedia:
  Un patron de conception [design pattern] est un arrangement caractéristique de modules, reconnu comme bonne pratique en réponse à un problème de conception d'un logiciel. Il décrit une solution standard, utilisable dans la conception de différents logiciels
- In this week, we will see:
  - The Singleton design pattern
  - The Factory Method design pattern
  - The Observer/Observable design pattern
  - The Visitor design pattern

The Singleton Design Pattern

An example

- This program gets information about the available memory from the JVM:

```
public class TestRunTime {
    public static void main(String[] args) {
        Runtime rt=Runtime.getRuntime();
        long m=rt.totalMemory();
        System.out.println(String.format("We have %d bytes available",m));
    }
}
```

- Why not this?

```
public static void main(String[] args) {
    Runtime rt=new Runtime();
    long m=rt.totalMemory();
    System.out.println(String.format("We have %d bytes available",m));
}
```

Does not compile!

An example (2)

- This is how the class Runtime is defined in the JDK:

```
public class Runtime {
    private static final Runtime currentRuntime = new Runtime();

    /** Don't let anyone else instantiate this class */
    private Runtime() {}

    public static Runtime getRuntime() {
        return currentRuntime;
    }

    public long totalMemory() {
        ...
    }
}
```

Singleton

- The authors of the Runtime class wanted to be sure that there is **only one instance** of that class
- Such a class is called a *Singleton*
- The Singleton is like a global variable. There is only **one** Runtime object in the entire program.
- Singletons are quite popular in Java. Some examples:

  java.lang.Runtime

  java.awt.Desktop

  ...
- Singletons make sense if you have certain resources (runtime, desktop,...) that only exist once
- Don't use Singletons if not absolutely necessary!

How to not use Singletons

```
class MySingleton {
    public String username;
    private static final MySingleton single = new MySingleton();
    private MySingleton() {}
    public static MySingleton getSingleton() { return single; }
}
class MyClass1 {
    void method1() {
        MySingleton.getSingleton().username="Alice";
    }
}
class MyClass2 {
    void method2() {
        System.out.println(MySingleton.getSingleton().username);
    }
}
```

Singleton misused to create a global variable "username"

- Do not use Singletons to emulate global variables in Java
- This program is <u>very hard</u> to maintain because the Singleton creates a "magic connection" between MyClass1 and MyClass2. You cannot see from outside that MyClass1 and MyClass2 depend on each other.

“Lazy initialization”

- Lazy initialization can be used if you want to create a Singleton object only when it is really needed
- Here is a possible version of the Runtime class with lazy initialization:

```
public class Runtime {
    // don’t create the object at the beginning
    private static final Runtime currentRuntime = null;

    private Runtime() {}

    public static Runtime getRuntime() {
        if(currentRuntime==null) {              // already created?
            currentRuntime=new Runtime();       // create now
        }
        return currentRuntime;
    }

    public long totalMemory() { ... }
}
```