Tecniche di Progettazione: Design Patterns

GoF: Singleton

Singleton pattern

- Intent
  - Ensure a class only has one instance
  - Provide a global point of access to it
- Motivation
  - Sometimes we want just a single instance of a class to exist in the system;
    - For example, we want just one window manager. Or just one factory for a family of products.
  - We need to have that one instance easily accessible
  - And we want to ensure that additional instances of the class can not be created

Overview

- Recognizing Singleton
- Singleton Mechanics
- Lazy initialization
- Singletons and Threads
- Subclassing Singletons

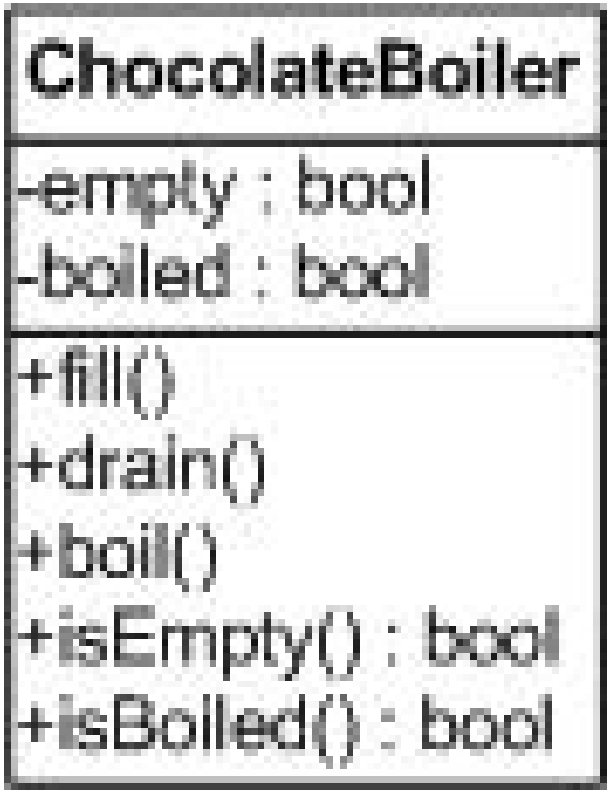
Recognizing Singleton

- Unique objects are not uncommon
- Most objects in an application bear a unique responsibility
- Yet singleton classes are relatively rare
- Fact that an object/class is unique doesn't mean that the Singleton pattern is at work

Chocolate Factory Case Study

- Choc-O-Holic Inc's industrial strength Chocolate Boiler mixes ingredients and milk at a high temperature to make liquid chocolate
- The `ChocolateBoiler` class also has two boolean attributes `empty` and `boiled`
- The `ChocolateBoiler` class contains five methods `fill()`, `drain()`, `boil()`, `isEmpty()` and `isBoiled()`
- **Model this class**

ChocolateFactory



Problems…

- The Chocolate Boiler has overflowed! It added more milk to the mix even though it was full!!
- What happened?
- **Hint:** What happens if more than two instances of `ChocolateBoiler` **are created?**

- **The problem** is with **two instances** controlling the same **phisycal** boiler

Prevent multiple instances

- How can you prevent other developers from constructing new instances of your class?
  - Create a single constructor with `private` access

```
private static ChocolateBoiler
_chocolateboiler = new
ChocolateBoiler()
```

  - Make the unique instance available through a public static `GetChocolateBoiler()` method

Lazy Initialization

- Rather than creating a singleton instance ahead of time – wait until instance is first needed

```
public static ChocolateBoiler GetChocolateBoiler()
{
  if (_chocolateboiler == null)
  {
            _chocolateboiler = new ChocolateBoiler();
            // ...
  }
  return _chocolateboiler
}
```

Why use Lazy Initialization?

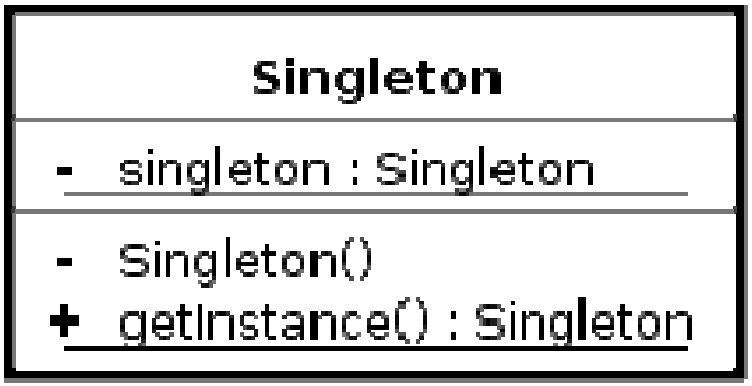
1. Might not have enough information to instantiate a singleton at static initialization time
   - **Example**: a `ChocolateBoiler` singleton may have to wait for the real factory's machines to establish communication channels

2. If the singleton is resource intensive and may not be required
   - **Example:** a program that has an optional query function that requires a database connection

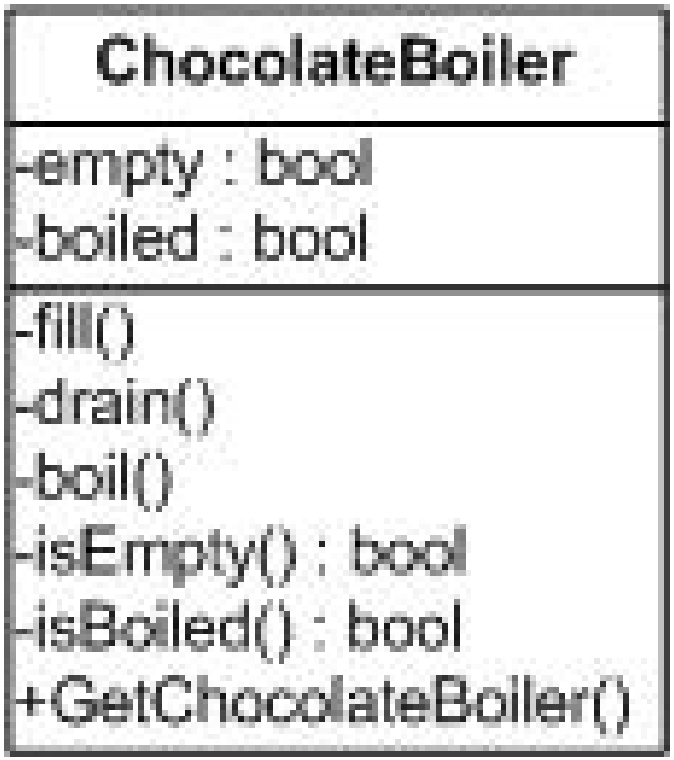
Full Picture

```
public class ChocolateBoiler {
  private static ChocolateBoiler _chocolateboiler;
  private ChocolateBoiler () {};
  public static ChocolateBoiler GetChocolateBoiler()
  {
    if (_chocolateboiler == null)
    {
       _chocolateboiler = new ChocolateBoiler();
             // ...
    }
    return _chocolateboiler
  }
}
```

UML Class Diagram



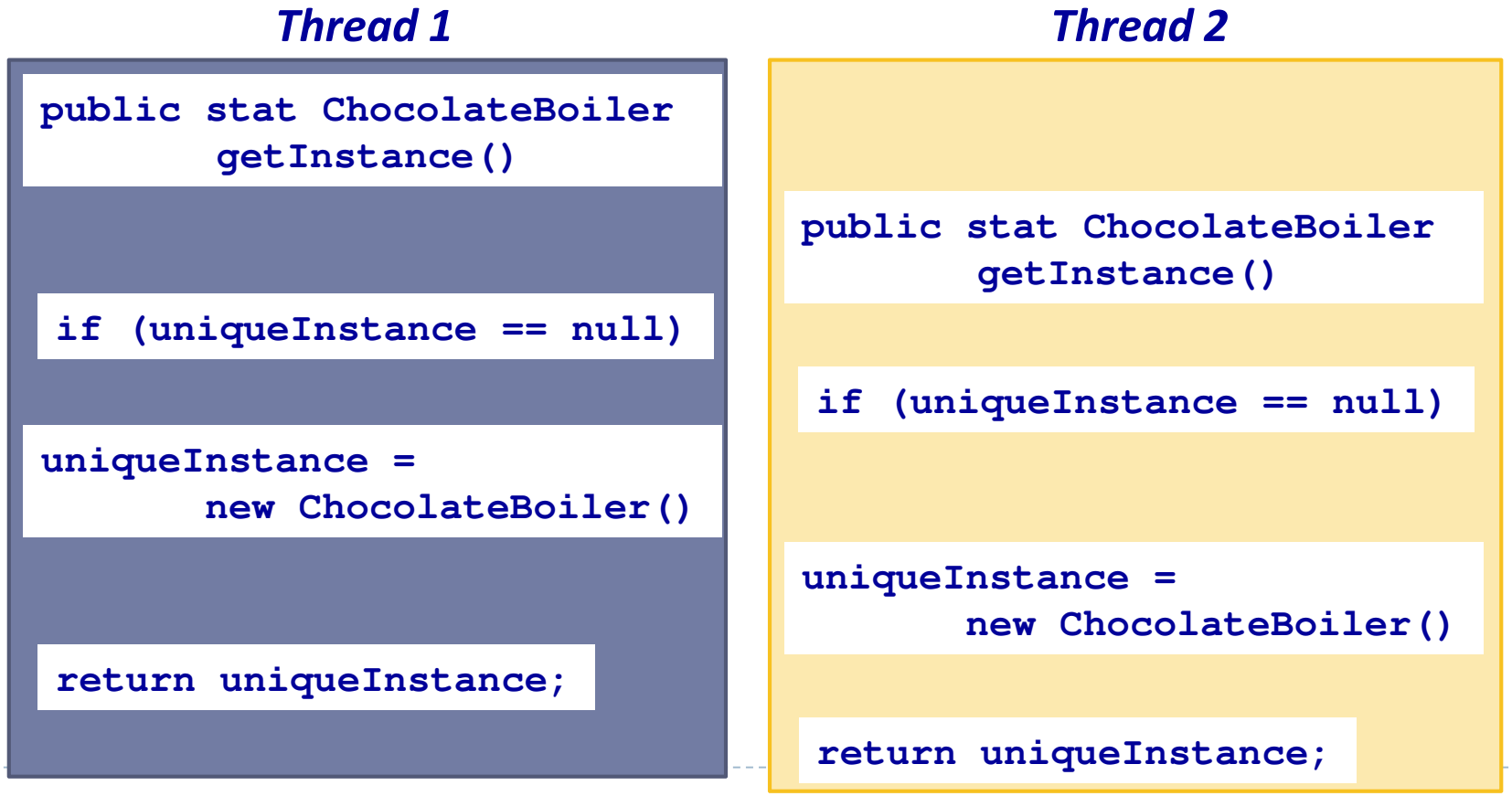
Our class so far...



- **as it is problems with threads …**

Thread Example

- If the program is run in a multi-threaded environment it is possible for two threads to initialize two singletons at roughly the same time



Problems with Multithreading

- In the case of multithreading with more than one processor the getInstance() method could be called at more or less the same time resulting in to more than one instance being created.
- Possible solutions:
  - Synchronize the getInstance() method
  - Move to an eagerly created instance rather than a lazily created one.
  - Use double—checked—locking

Synchronizing the getInstance() Method

Code

```
public static synchronized Singleton getInstance()
  {...
   }
```

- Disadvantage – synchronizing can decrease system performance by a factor of 100.
- Synchronization is expensive, however, and is really only needed the first time the unique instance is created.
- Use only if the performance of the getInstance() method is not critical to the application.

Use an Eagerly Created Instance Rather than a Lazy One

- Code:

```
//Data elements
private static Singleton uniqueInstance = new
  Singleton()

private Singleton() {}

public static Singleton getInstance() {
  return uniqueInstance
}
```

- Disadvantage – Memory may be allocated and not used.

Use double—checked—locking

- In an effort to make this method more efficient, an idiom called double-checked locking was created. The idea is to avoid the costly synchronization for all invocations of the method except the first. The cost of synchronization differs from JVM to JVM. In the early days, the cost could be quite high. As more advanced JVMs have emerged, the cost of synchronization has decreased, but there is still a performance penalty for entering and leaving a synchronized method or block. Regardless of the advancements in JVM technology, programmers never want to waste processing time unnecessarily.

Use double-checked-locking

- Code:

```
private volatile static Singleton uniqueInstance
private Singleton() {}
public static Singleton getInstance() {
  if (uniqueInstance == null)
      synchronized (Singleton.class) {
         if (uniqueInstance == null) {
            uniqueInstance  = new Singleton()
  return uniqueInstance
}
```

- If a variable is declared as volatile then is guaranteed that any thread which reads the field will see the most recently written value.

Singleton With Subclassing

- What if we want to be able to subclass Singleton and have the single instance be a subclass instance?
- For example, suppose MazeFactory had subclasses EnchantedMazeFactory and AgentMazeFactory. We want to instantiate just one of them.
- How could we do this?
  1. Have the static instance() method of MazeFactory determine the particular subclass instance to instantiate. This could be done via an argument or environment variable. The constructors of the subclasses can not be private in this case, and thus clients *could instantiate other instances of the* subclasses.
  2. Have each subclass provide a static instance() method. Now the subclass constructors can be private.

Singleton With Subclassing Method 1

- Method 1: Have the MazeFactory instance() method determine the subclass to instantiate

```
        // Class MazeFactory only allows one instantiation of a subclass.
public abstract class MazeFactory {
        // The private reference to the one and only instance.
    private static MazeFactory uniqueInstance = null;
        // The MazeFactory constructor.
        // If you have a default constructor, it can not be private here!
    protected MazeFactory() {}
```

Singleton With Subclassing Method 1 (Continued)

```
        // Return instance. If instance not yet created, create "enchanted" as default.
public static MazeFactory instance() {
        if (uniqueInstance == null) return instance("enchanted");
                else return uniqueInstance;
}
        // Create the instance using the specified String name.
public static MazeFactory instance(String name) {
        if(uniqueInstance == null)
                if (name.equals("enchanted"))
                        uniqueInstance = new EnchantedMazeFactory();
                else if (name.equals("agent"))
                        uniqueInstance = new AgentMazeFactory();
return uniqueInstance;}}
```

Singleton With Subclassing Method 1 (Continued)

- Client code to access the factory or create it:

```
MazeFactory factory = MazeFactory.instance();
MazeFactory factory = MazeFactory.instance(".....");
```

- (code smell…..)

- The constructors of EnchantedMazeFactory and AgentMazeFactory can not be private, since MazeFactory must be able to instantiate them. Thus, clients could potentially instantiate other instances of these subclasses.

Singleton With Subclassing Method 1 (Continued)

- The instance(String name) methods violates the Open-Closed Principle, since it must be modified for each new MazeFactory subclass
- We could use Java class names as the argument to the instance(String) method, yielding simpler code:

```
public static MazeFactory instance(String name) {
   if (uniqueInstance == null)
         uniqueInstance = Class.forName(name).newInstance();
   return uniqueInstance;
}
```

Singleton With Subclassing Method 2

Have each subclass provide a static instance method()

```
/**
* The subclasses provide an implementation of a static instance() method.
*/
    public abstract class MazeFactory {
                    // The protected reference to the one and only instance.
            protected static MazeFactory uniqueInstance = null;
                    // The MazeFactory constructor. If you have a default
                    //constructor, it can not be private here!
            protected MazeFactory() {}
                    // Returns a reference to the single instance.
            public static MazeFactory instance() {return uniqueInstance;}
    }
```

Singleton With Subclassing Method 2 (Continued)

```
public class EnchantedMazeFactory extends MazeFactory {
                    // Return a reference to the single instance.
        public static MazeFactory instance() {
                    if(uniqueInstance == null)
                                uniqueInstance = new EnchantedMazeFactory();
                    return uniqueInstance;
    }
        // Private subclass constructor!!
    private EnchantedMazeFactory() {}
}
```

Singleton With Subclassing Method 2 (Continued)

- Client code to create factory the first time:

```
MazeFactory factory = EnchantedMazeFactory.instance();
```

- Client code to access the factory:

```
MazeFactory factory = MazeFactory.instance();
```

- Note that now the constructors of the subclasses are private. Only one subclass instance can be created!
- Also note that the client can get a null reference if it invokes MazeFactory.instance() before the unique subclass instance is first created
- Finally, note that uniqueInstance is now protected!

Static Attributes in a Class

- Each object of a class has its own copy of all the instance variables of that class.
- However, in certain cases all class objects should share only one copy of a particular variable.
  - Such variables are called static variables. A program contains only one copy of each of a class's static variables in memory, no matter how many objects of the class have been instantiated.
- A static variable represents class-wide information. All class objects share the same static data item.
- The public static attributes of a class can be accessed through the class name and dot operator (e.g. Math.PI). Private static attributes can only be accessed through methods and properties of that class.

Better using singletons or static classes?

- With a singleton you can pass the object as a parameter to another method;
- With a singleton you can implement interfaces or derive a base class;
- With a singleton you can use a factory pattern to build up your instance (and/or choose which class to instantiate).

- In both cases care with multithreading.

Homework

- Try refactor the Christmas tree example using Singleton for the star constraint.