Singleton Chapter 5

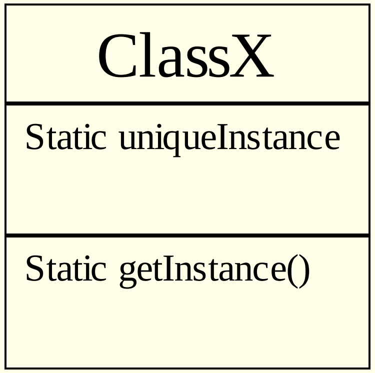
Slides for ACS-3913

2 to 5 and

12 to 16

Singleton Chapter 5

To guarantee that there is at most one instance of a class we can apply the singleton pattern.



uniqueInstance is a class variable that holds the <u>one</u> instance.

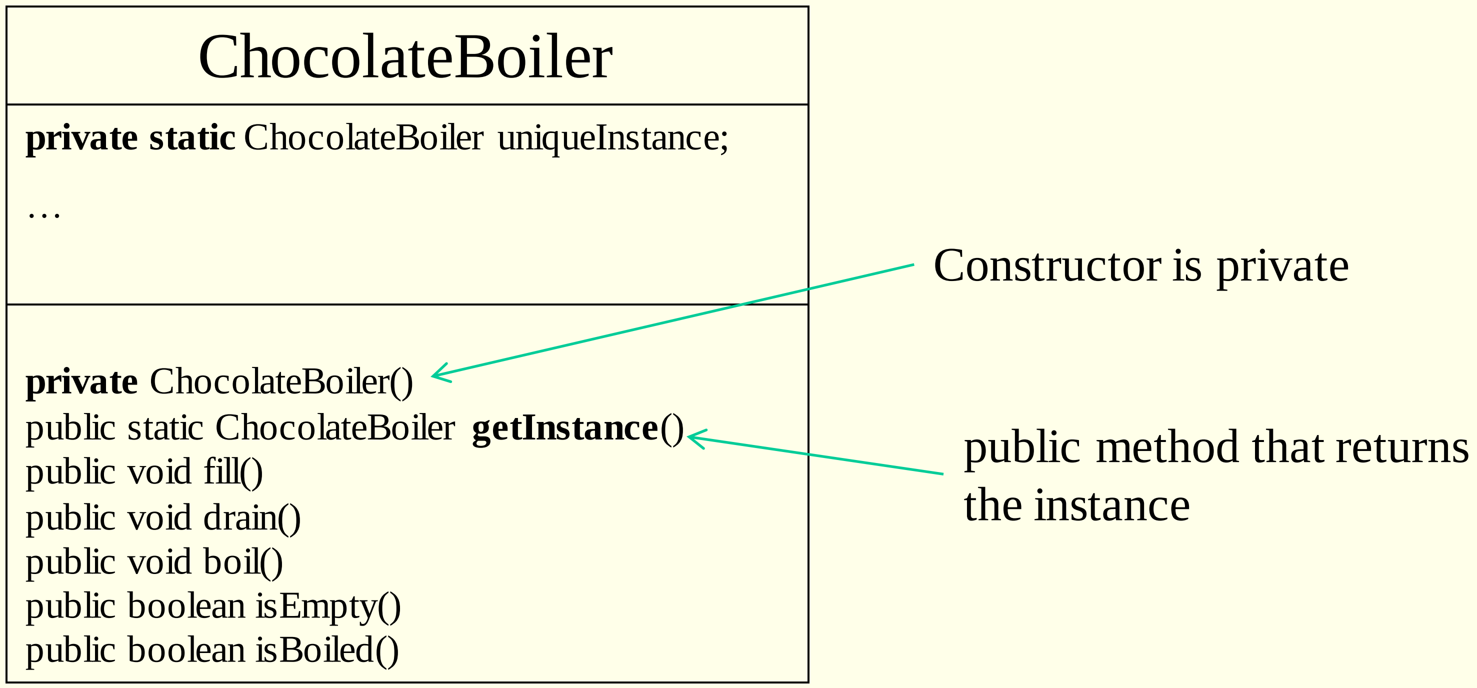
getInstance() is a class method we can access regardless of whether there is an instance or not.

getInstance() is just the recommended name for this method – you may encounter examples that use a different method name.

Constructor is private and called only by getInstance()

Singleton

The text has a ChocolateBoiler class for which they want to have at most one instance.

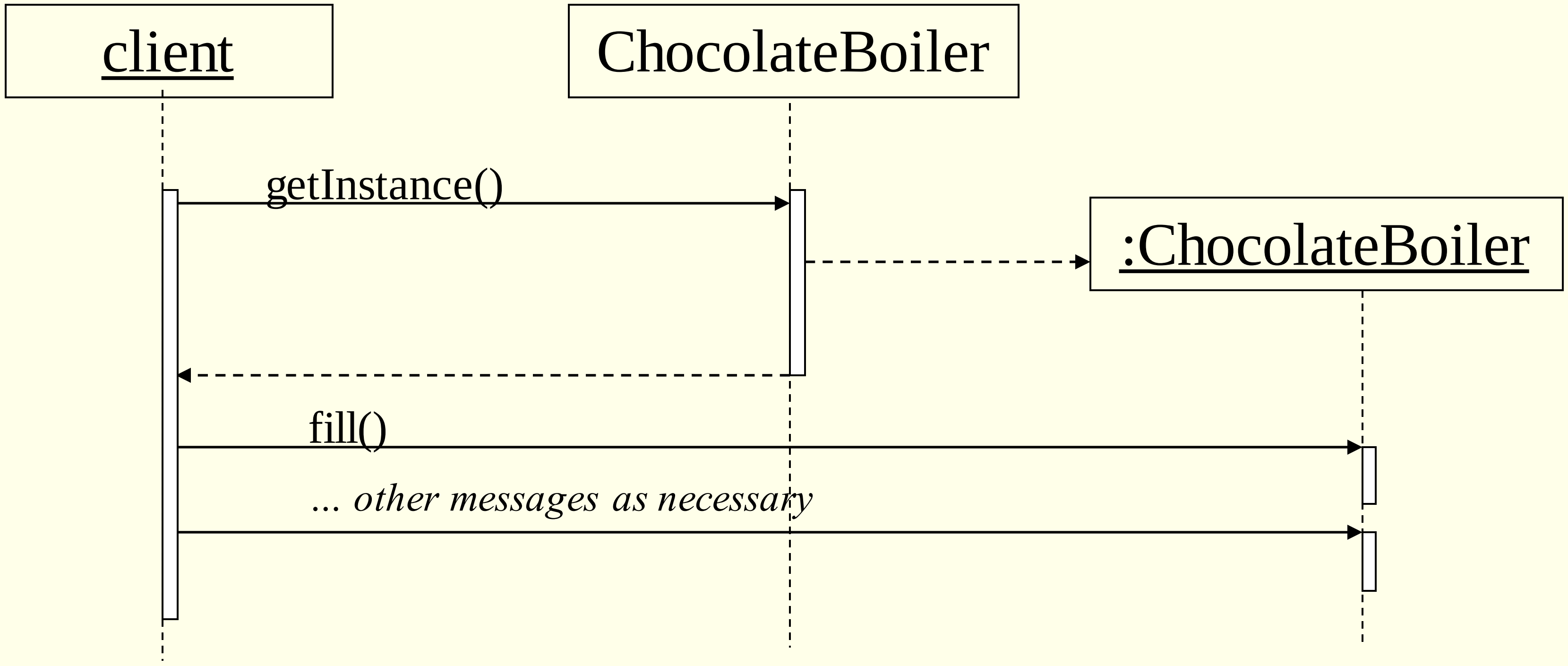

ChocolateBoiler
private static ChocolateBoiler uniqueInstance;
…
private ChocolateBoiler()
public static ChocolateBoiler **getInstance**()
public void fill()
public void drain()
public void boil()
public boolean isEmpty()
public boolean isBoiled()
Constructor is private
public method that returns the instance

Sequence Diagram for Singleton

In order to get a reference to the singleton object, the client object just sends the message:

ChocolateBoiler singleton = ChocolateBoiler.getInstance();

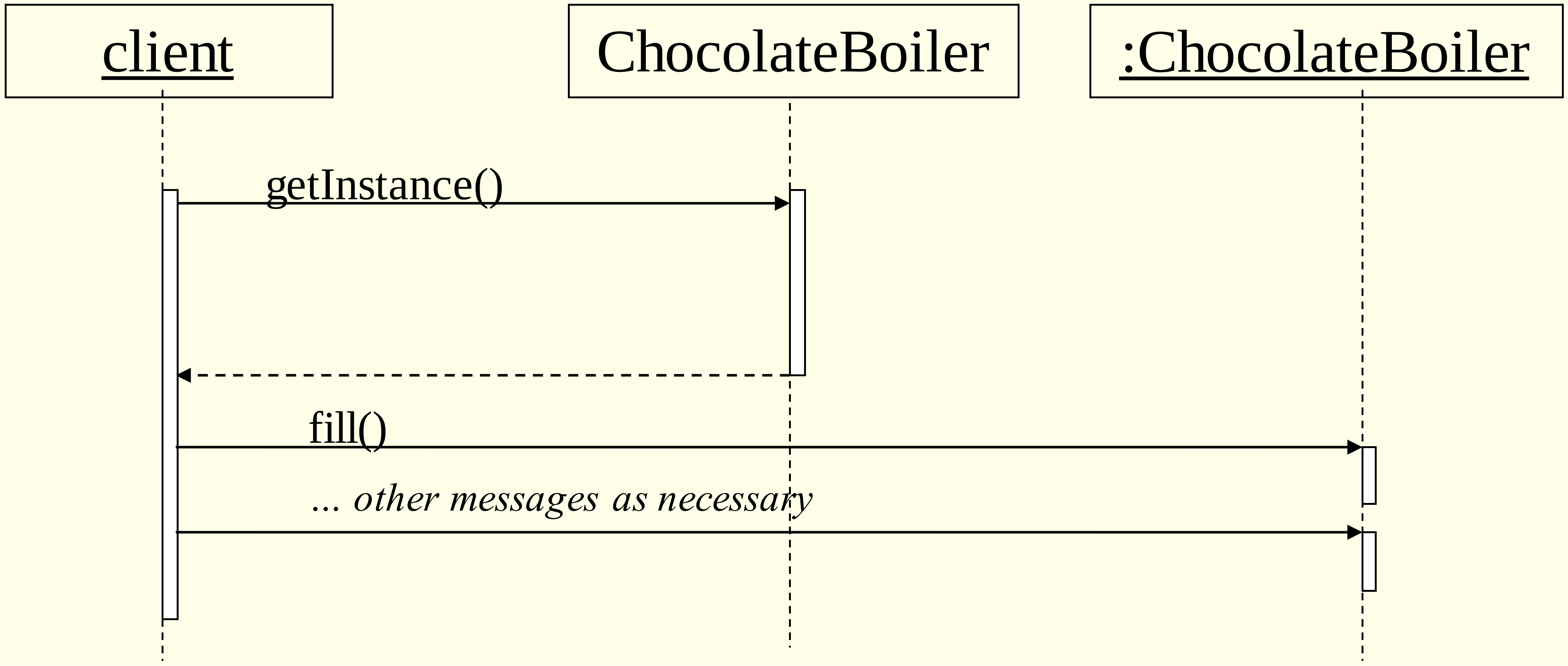
One scenario:



Sequence Diagram for Singleton

The other scenario:

(when is this realized?)



Singleton

If there's only ever one thread, the previous implementation will work.

However, if there's more than one thread we could end up with multiple "singletons" (see page 178)

Each thread has its own "copy" of variables. The value of a variable could be "out-of-sync" with the main copy.

We could have getInstance started by several threads. Each could test the value of uniqueInstance and proceed to instantiate the singleton.

To work correctly, we need a technique to synchronize the actions of the threads. With Java we could use synchronization of methods or variables.

Multi-threading and Singletons

The text presents synchronized static methods as one solution. This results in locking and unlocking of the class, and hence only one synchronized method will execute at a time.

```
public class Singleton {
        private static Singleton uniqueInstance;
        // other useful instance variables here

        private Singleton() {}

        public static synchronized Singleton getInstance() {
                if (uniqueInstance == null) {
                        uniqueInstance = new Singleton();
                }
                return uniqueInstance;
        }
        // other useful methods here
}
```

Synchronized Methods

When a static synchronized method is invoked, the thread must first acquire the intrinsic lock for the Class object associated with the class – this is done automatically for you.

Only one thread can hold a lock at one time; other threads that request a lock are placed in a wait queue.

Synchronized Methods

Synchronized methods introduce overhead and we can avoid them.

Some alternatives, use:

a) double-checked locking

This scheme uses a synchronized statement and a volatile variable.

b) eager instantiation

Synchronized Methods

If a variable is <u>not</u> declared as volatile (i.e. it is <u>non-volatile</u>) then thread A, when accessing the variable, may not see the most recent value that was written by some other thread, say thread B.

If a variable is declared as <u>volatile</u> then it is guaranteed that any thread which reads the field will see the most recently written value.

1st alternative: Double-checked locking

Check first to see if the instance exists or not. If not, then lock up a block of code.

```
// Danger!  This implementation of Singleton not
// guaranteed to 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) {
                                                            uniqueInstance = new Singleton();
                                                }  }  }
                        return uniqueInstance;
                                                }  }
```

A thread's copy of a volatile attribute is reconciled with the "master" copy each time it is referenced.

Note these two checks on uniqueInstance

Second check is necessary to verify uniqueInstance is still null

A synchronized block of code.

Only one thread at a time will execute this.

Results in very little overhead compared to synchronizing a whole method/class.

2nd alternative: Eager Instantiation

If your system always instantiates the singleton, then create it in advance – very simple – done by class loader prior to the class being used.

One instance as a class variable

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

        private Singleton() {}

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

When asked, just return the reference to the static variable

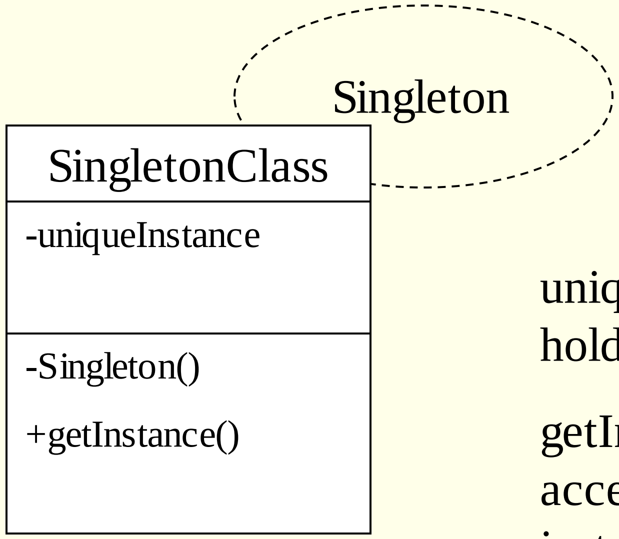
When the class is loaded, the instance is created and available. getInstance() is always realized the same way.

Singleton

How do we show that some class in a UML class diagram is a singleton class?

Singleton – class diagram

To guarantee that there is at most one instance of a class we can apply the singleton pattern.



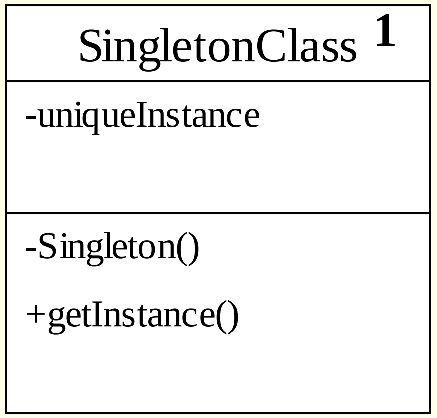
uniqueInstance is a class variable that holds the one instance.

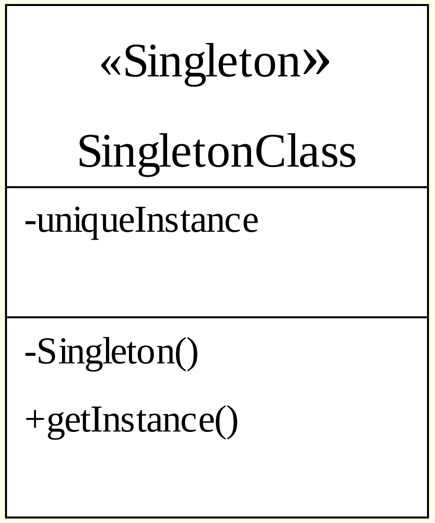
getInstance() is a class method we can access regardless of whether there is an instance or not.

Constructor is private.

Singleton – class diagram

Other notations:





Singleton

Some other examples

- Consider the *shapes* example in BlueJ

- Consider the wikipedia entry for Singleton Pattern