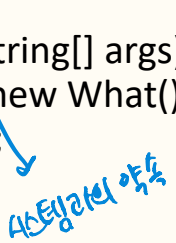
# Singleton Pattern

# Before start…

```
public class What {
        What() {
                System.out.println(“What is this?\n”);
        }
        public void doThis() {
                System.out.println(“Ok!, I will do this.\n”);
        }
        public static void main(String[] args) {
                What doWhat = new What();
                doWhat.doThis();
        }
}
```

# Before start…

```
public class What {
        private What() {
                System.out.println(“What is this?\n”);
        }
        public void doThis() {
                System.out.println(“Ok!, I will do this.\n”);
        }
        public static void main(String[] args) {
                What doWhat = new What();
                doWhat.doThis();
        }
}
```



# Find difference?

What.java

```
public class What {
        private What() {
                System.out.println("What is this?\n");
        }
        public void doThis() {
                System.out.println("Ok!, I will do this.\n");
        }
}
```

Main.java

```
public class Main {
        public static void main(String[] args) {
                What doWhat = new What();
                doWhat.doThis();
        }
}
```

# Yes!

```
> java What
What is this?

Ok!, I will do this.

> java Main
Main.java:3: What() has private access
in What
                What doWhat = new What();
                              ^
1 error
>
```

# Introducing getInstance()

```
public class What {
        private What() {
                System.out.println("What is this?\n");
        }
        public void doThis() {
                System.out.println("Ok!, I will do this.\n");
        }
        public static What getInstance() {
                What onlyWhat = new What();
                return onlyWhat;
        }
}
```

# Singleton Pattern

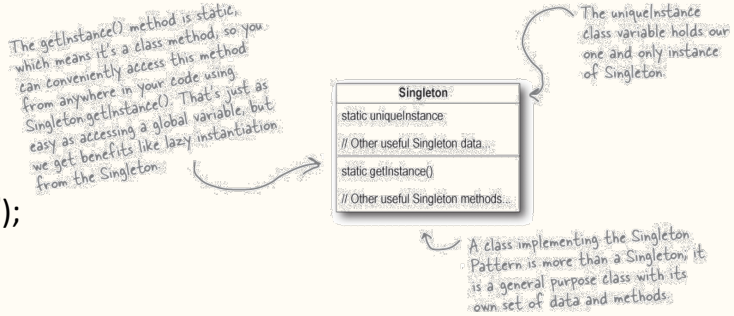
- Intention: Ensure a class only has one instance, and provide a global point of access to it
- Motivation: system objects that hold global data (like database, file system, printer spooler, or registry). It must be ensured that such objects are instantiated only once within a system and that their sole instance can be accessed easily from all parts of the system.
- Solution: We could create a special method that is used to instantiate the desired object. When this method is called, it checks to see whether the object has already been instantiated. If it has this method just returns a reference to the object. If not, the method instantiates new object and return it.

# Why Singleton?

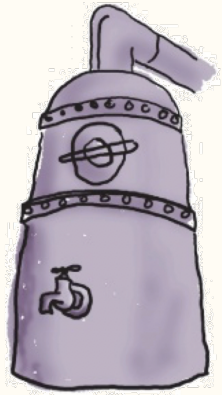
- When we have limited number of shared resources
- For performance reasons, just instantiate only one object and then use this again and again, if possible
- Static global variable can do the same job, but if we choose static global variable method, That resource will be locked until this application is terminated
- Singleton makes objects responsible for themselves

# First try! Works fine but…

```
public class What {
        private static What onlyWhat = null;
        private What() {
                System.out.println("What is this?\n");
        }
        public void doThis() {
                System.out.println("Ok!, I will do this.\n");
        }
        public static What getInstance() {
                if(onlyWhat == null) onlyWhat = new What();
                return onlyWhat;
        }
}
```



# Chocolate Factory



```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    public  ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // fill the boiler with a milk/chocolate mixture
        }
    }

    public void drain() {
        if (!isEmpty() && isBoiled()) {
            // drain the boiled milk and chocolate
            empty = true;
        }
    }

    public void boil() {
        if (!isEmpty() && !isBoiled()) {
            // bring the contents to a boil
            boiled = true;
        }
    }

    public boolean isEmpty() {
        return empty;
    }

    public boolean isBoiled() {
        return boiled;
    }
}
```

This code is only started when the boiler is empty!

To fill the boiler it must be empty, and, once it's full, we set the empty and boiled flags.

To drain the boiler, it must be full (non-empty) and also boiled. Once it is drained we set empty back to true.

To boil the mixture, the boiler has to be full and not already boiled. Once it's boiled we set the boiled flag to true.

# Threads and Executors

- The Concurrency API was first introduced with the release of Java 5

```
Runnable task = () -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);
};

task.run();

Thread thread = new Thread(task);
thread.start();

System.out.println("Done!");
```

Introducing lambda expression for thread

```
Runnable runnable = () -> {
    try {
        String name = Thread.currentThread().getName();
        System.out.println("Foo " + name);
        TimeUnit.SECONDS.sleep(1);
        System.out.println("Bar " + name);
    }
    catch (InterruptedException e) {
        e.printStackTrace();
    }
};

Thread thread = new Thread(runnable);
thread.start();
```

# Executors

- The Concurrency API introduces the concept of an ExecutorService as a higher level replacement for working with threads directly.
- Executors are capable of running asynchronous tasks and typically manage a pool of threads, so we don't have to create new threads manually.
- All threads of the internal pool will be reused under the hood for revenant tasks, so we can run as many concurrent tasks as we want throughout the life-cycle of our application with a single executor service.

# Executors and ExecutorService

- The class Executors provides convenient factory methods for creating different kinds of executor services.
- In this sample we use an executor with a thread pool of size one.
- The result looks similar to the above sample but when running the code you'll notice an important difference: the java process never stops! Executors have to be stopped explicitly - otherwise they keep listening for new tasks.
- An ExecutorService provides two methods for that purpose: shutdown() waits for currently running tasks to finish while shutdownNow() interrupts all running tasks and shut the executor down immediately.

```java
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.submit(() -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);
});

// => Hello pool-1-thread-1
```

```java
try {
    System.out.println("attempt to shutdown executor");
    executor.shutdown();
    executor.awaitTermination(5, TimeUnit.SECONDS);
}
catch (InterruptedException e) {
    System.err.println("tasks interrupted");
}
finally {
    if (!executor.isTerminated()) {
        System.err.println("cancel non-finished tasks");
    }
    executor.shutdownNow();
    System.out.println("shutdown finished");
}
```

# Callables and Futures

- Callables are functional interfaces just like runnables but instead of being void they return a value.
- Callables can be submitted to executor services just like runnables.
- But what about the callables result?
  - Since submit() doesn't wait until the task completes, the executor service cannot return the result of the callable directly.
  - Instead, the executor returns a special result of type Future which can be used to retrieve the actual result at a later point in time.

```
Callable<Integer> task = () -> {
    try {
        TimeUnit.SECONDS.sleep(1);
        return 123;
    }
    catch (InterruptedException e) {
        throw new IllegalStateException("task interrupted", e);
    }
};
```

```
ExecutorService executor = Executors.newFixedThreadPool(1);
Future<Integer> future = executor.submit(task);

System.out.println("future done? " + future.isDone());

Integer result = future.get();

System.out.println("future done? " + future.isDone());
System.out.print("result: " + result);
```

Calling the method get() blocks the current thread and waits until the callable completes before returning the actual result 123. Now the future is finally done, and we see the following result on the console:

newFixedThreadPool(1) to create an executor service backed by a thread-pool of size one. This is equivalent to newSingleThreadExecutor() but we could later increase the pool size by simply passing a value larger than one.

# Timeouts

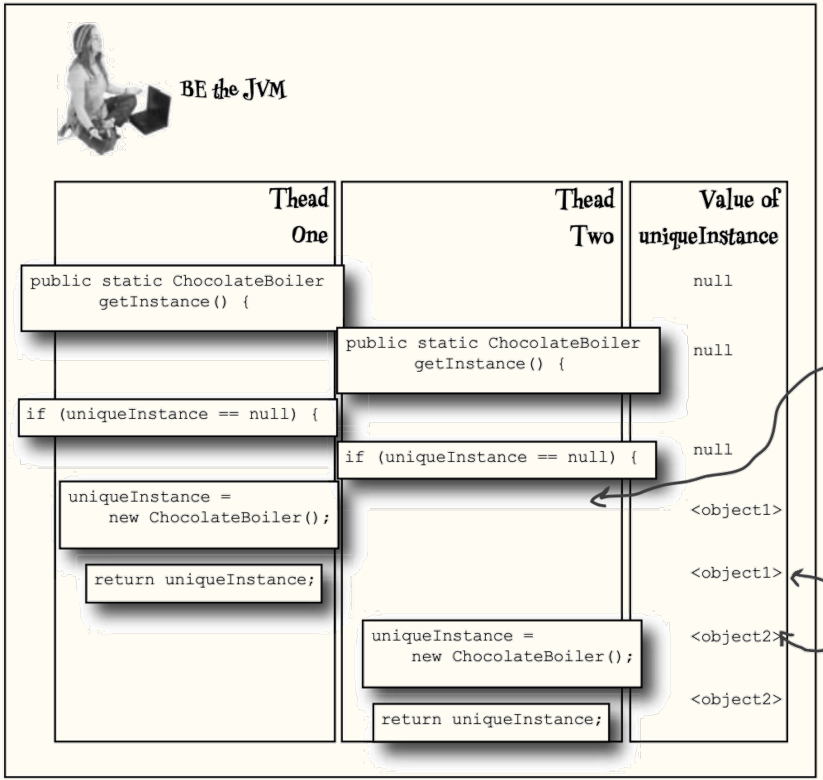
- Any call to future.get() will block and wait until the underlying callable has been terminated. In the worst case a callable runs forever - thus making your application unresponsive. You can simply counteract those scenarios by passing a timeout:

```
ExecutorService executor = Executors.newFixedThreadPool(1);

Future<Integer> future = executor.submit(() -> {
    try {
        TimeUnit.SECONDS.sleep(2);
        return 123;
    }
    catch (InterruptedException e) {
        throw new IllegalStateException("task interrupted", e);
    }
});

future.get(1, TimeUnit.SECONDS);
```

cf. http://winterbe.com/posts/2015/04/30/java8-concurrency-tutorial-synchronized-locks-examples/


BE the JVM
Thead One
Thead Two
Value of uniqueInstance
public static ChocolateBoiler getInstance() {
public static ChocolateBoiler getInstance() {
if (uniqueInstance == null) {
if (uniqueInstance == null) {
uniqueInstance = new ChocolateBoiler();
return uniqueInstance;
uniqueInstance = new ChocolateBoiler();
return uniqueInstance;
null
null
null
<object1>
<object1>
<object2>
<object2>
Uh oh, this doesn't look good!
Two different objects are returned! We have two ChocolateBoiler instances!!!

# Synchronized Method

```
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
}
```

By adding the synchronized keyword to getInstance(), we force every thread to wait its turn before it can enter the method. That is, no two threads may enter the method at the same time.

Too expensive! and slow...

# private static (final) Singleton

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

    private Singleton() {}

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

Go ahead and create an instance of Singleton in a static initializer. This code is guaranteed to be thread safe!

We've already got an instance, so just return it

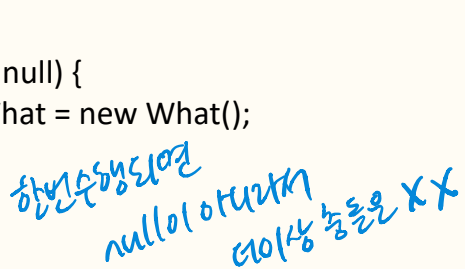
Not bad! But waste of memory.

# Double-checked locking

```
public class What {
        private volatile static What onlyWhat;
        private What() { }
        public void doThis() {
                System.out.println("Ok!, I will do this.\n");
        }
        public static What getInstance() {
                if(onlyWhat == null) {
                        synchronized(this) {
                                if(onlyWhat == null) {
                                        onlyWhat = new What();
                }       }       }
                return onlyWhat;
}       }
```

Need Java 1.5 + !!

cf. https://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html



# One step more

- Try to think about the Singleton Pattern in class hierarch. (i.e., Abstract class with the Singleton properties)
- Is it possible to inherit the Singleton properties to the subclass?
- What about polymorphism?

- https://stackoverflow.com/questions/8549825/singleton-with-subclassing-in-java