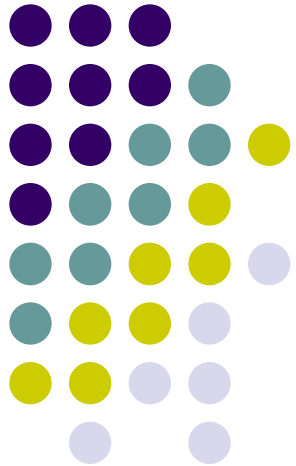
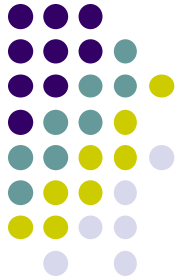
Singleton Pattern

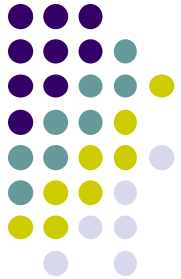


What use is that?



- There are many objects we only need one of: thread pools, caches, dialog boxes, object that handle preferences and registry setting, objects used for logging, and objects that act as device drivers to devices like printers and graphics cards. In fact, for many of these types of objects, if we were to instantiate more than one we'd run into all sorts of problems like incorrect program behavior, overuse of resources, or inconsistent results.

Dialog (Guru-G, Developer-D)



G: How would you create a single object?

D: new MyObject();

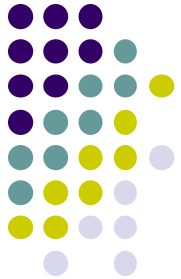
G: what if another object wanted to create a MyObject? Could it call new on MyObject again?

D: Yes, of course.

G: So as long as we have a class, can we always instantiate it one or more times?

D: Yes. Well, only if it's a public class.

Cont'd

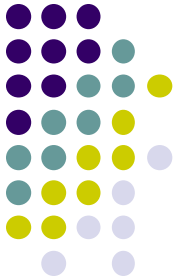


- G: And if not?
- D: Well, if it's not a public class, only classes in the same package can instantiate it. But they can still instantiate it more than once.
- G: Did you know you could do this?

```
public MyClass {
    private MyClass() {}
}
```

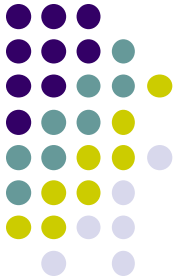
- D: No, I'd never thought of it, but I guess it makes sense because it is a legal definition.

Cont'd



- G: What does it mean?
- D: I suppose it is a class that can't be instantiated because it has a private constructor.
- G: Well, is there ANY object that could use the private constructor?
- D: I think the code in MyClass is the only code that could call it. But that doesn't make much sense.

Cont'd

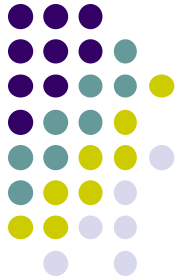


- G: Why not?
- D: Because I'd have to have an instance of the class to call it, but I can't have an instance because no other class can instantiate it. It's a chicken and egg problem: I can use the constructor from an object of type MyClass, but I can never instantiate that object because no other object can use "new MyClass()".
- G: OK. It was just a thought. What does this mean?

```
public MyClass {
    public static MyClass getInstance() { }
}
```

- D: MyClass is a class with a static method. We can call the static method like this: MyClass.getInstance();

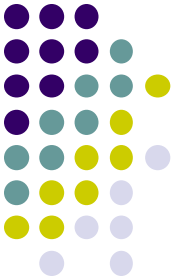
Cont'd



- G: Why did you use MyClass, instead of some object name?
- D: Well, getInstance() is a static method; in other words, it is a CLASS method. You need to use the class name to reference a static method.
- G: Very interesting. What if we put things together. Now can I instantiate a MyClass?

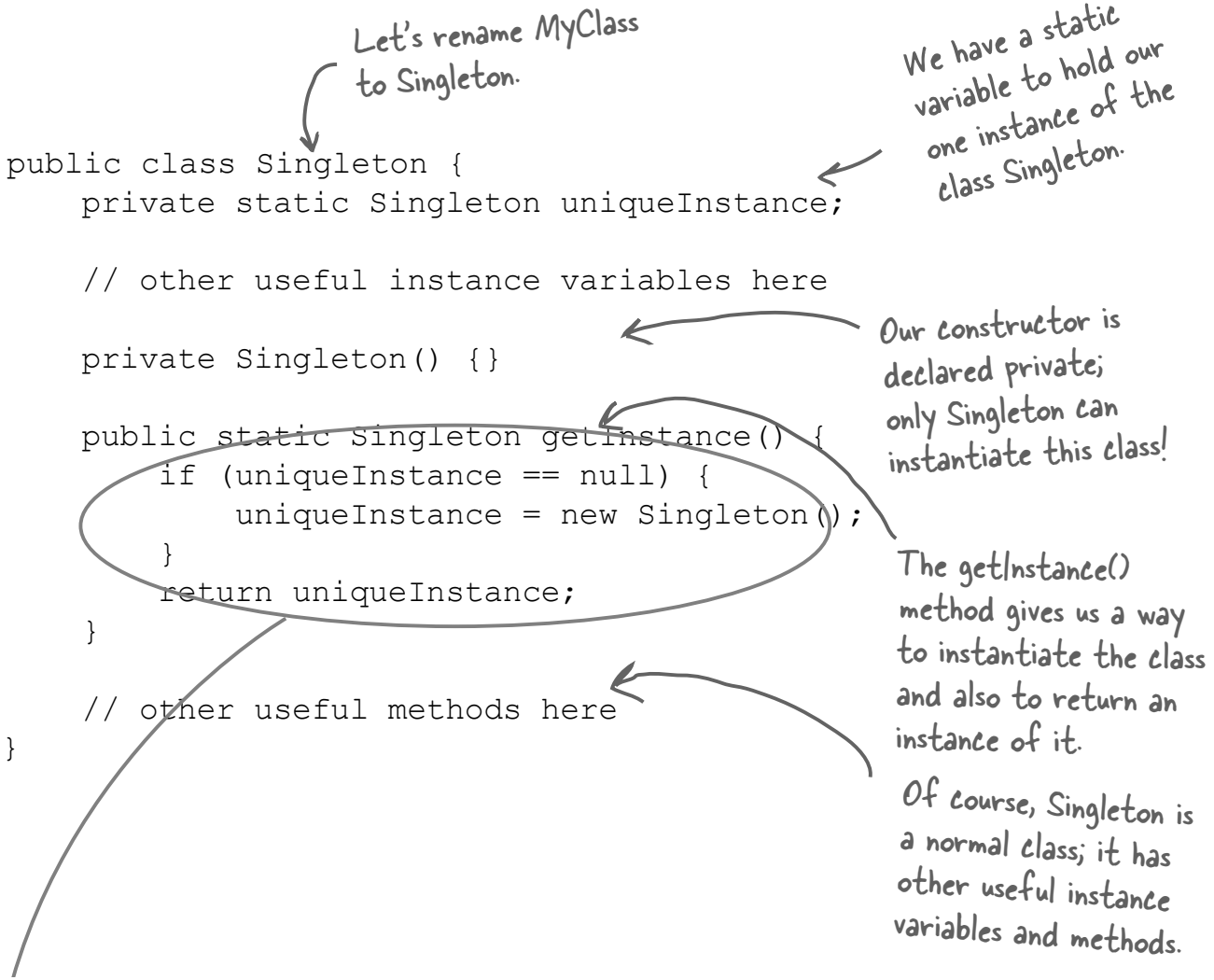
```
public MyClass {
        private MyClass() {}
        public static MyClass getInstance() {
                return new MyClass();
        }
}
```

Cont'd



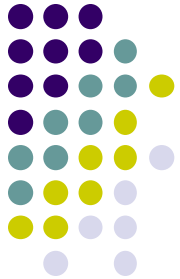
- D: Wow, you sure can.
- G: So, now can you think of a second way to instantiate an object?
- D: MyClass.getInstance();

Code




```
public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables here

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // other useful methods here
}
```

Code up close



uniqueInstance holds our ONE instance; remember, it is a static variable.

If uniqueInstance is null, then we haven't created the instance yet...

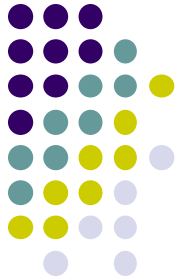
...and, if it doesn't exist, we instantiate Singleton through its private constructor and assign it to uniqueInstance. Note that if we never need the instance, it never gets created; this is lazy instantiation.

If uniqueInstance wasn't null, then it was previously created. We just fall through to the return statement.

By the time we hit this code, we have an instance and we return it.

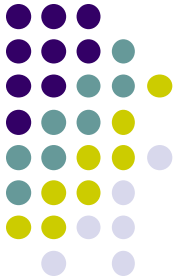
```
if (uniqueInstance == null) {
    uniqueInstance = new MyClass();
}
return uniqueInstance;
```

Singleton Pattern

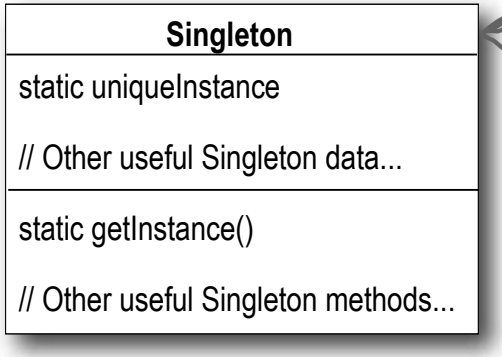


- The Singleton Pattern ensures a class has only one instance, and provides a global point of access to it.

The class diagram

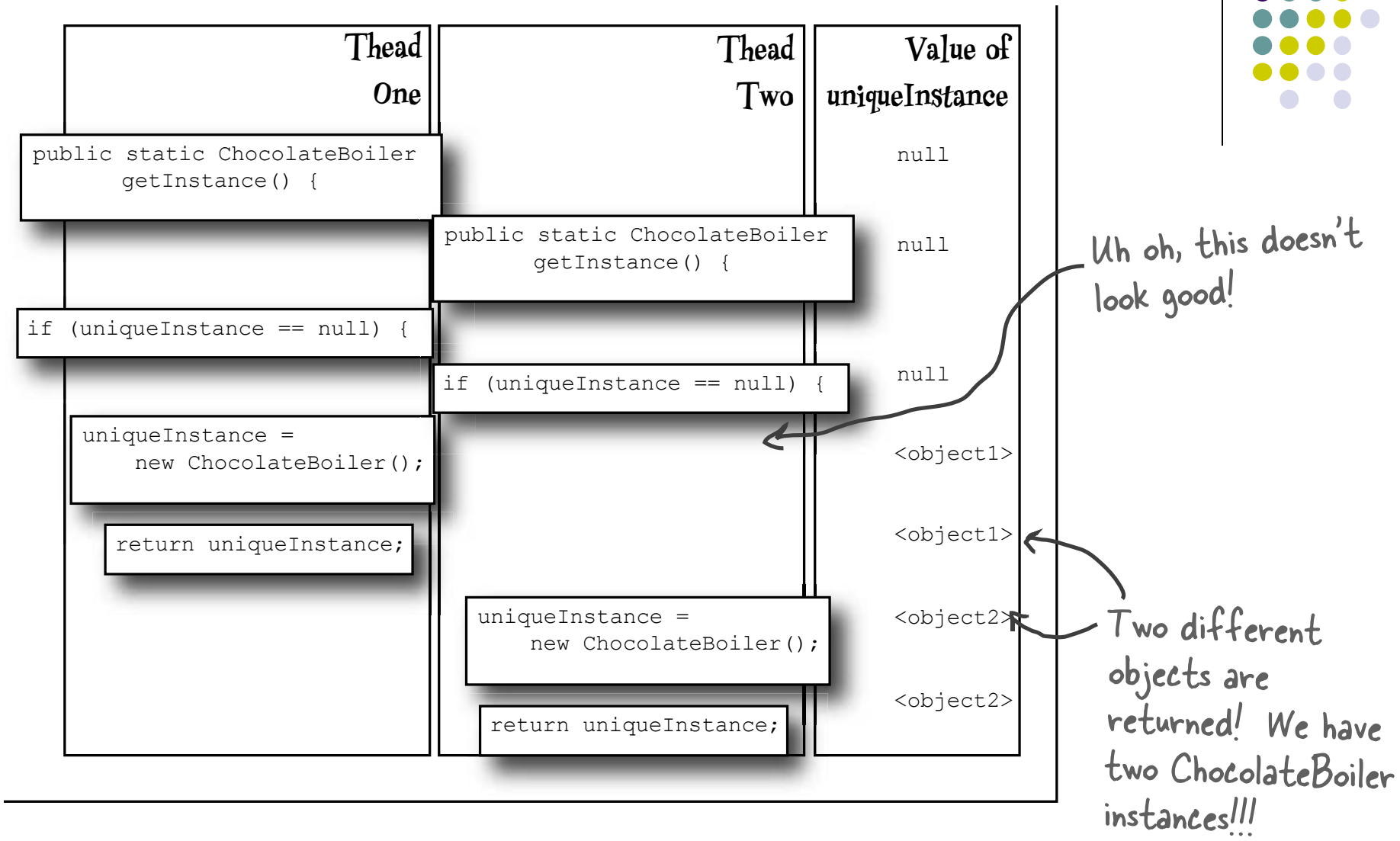


The getInstance() method is static, which means it's a class method, so you can conveniently access this method from anywhere in your code using Singleton.getInstance(). That's just as easy as accessing a global variable, but we get benefits like lazy instantiation from the Singleton.

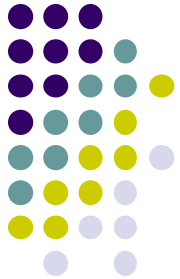


The uniqueInstance class variable holds our one and only instance of Singleton.

A class implementing the Singleton Pattern is more than a Singleton; it is a general purpose class with its own set of data and methods.


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!!!

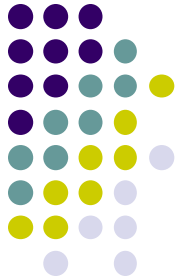
Dealing with multithreading



```
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.

Improve multithreading



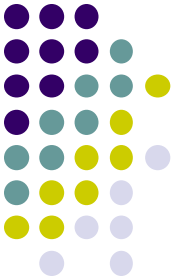
- Do nothing if the performance of getInstance() is not critical to your application.
- Move to an eagerly created instance rather than a lazily created on.

```
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.

Improve multithreading



- Use “double-checked locking” to reduce the use of synchronization in getInstance()

```
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;
    }
}
```

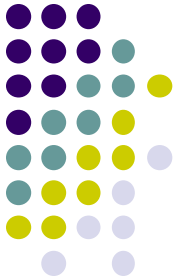
Check for an instance and if there isn't one, enter a synchronized block.

Note we only synchronize the first time through!

Once in the block, check again and if still null, create an instance.

* The volatile keyword ensures that multiple threads handle the uniqueInstance variable correctly when it is being initialized to the Singleton instance.

public static ChocolateBoiler getInstance()



For each solution, describe its applicability to the problem of fixing the Chocolate Boiler code:

Synchronize the getInstance() method:

_____

_____

Use eager instantiation:

_____

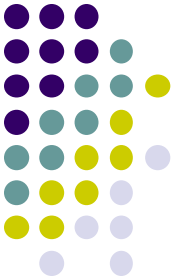
_____

Double-checked locking:

_____

_____

Reviews



- The Singleton Pattern ensures you have at most one instance of a class in your application.
- The Singleton Pattern also provides a global access point to that instance.
- Java's implementation of the Singleton Pattern makes use of a private constructor, a static method combined with a static variable.
- Examine your performance and resource constraints and carefully choose an appropriate Singleton implementation for multithreaded applications (and we should consider all applications multithreaded).