Singleton Pattern

Shin-Jie Lee (李信杰)
Associate Professor
Computer and Network Center
Department of CSIE
National Cheng Kung University

Design Aspect of Facade

the sole instance of a class

Outline

- Chocolate Boiler Requirements Statements
- Initial Design
- Recurrent Problems
- Intent
- Singleton Pattern Structure
- Singleton with Multi-threading Issues

Chocolate Boiler

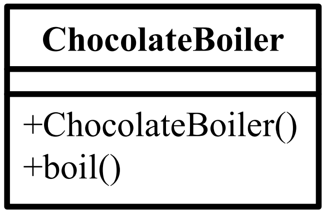
Shin-Jie Lee (李信杰)
Assistant Professor
Computer and Network Center
Department of CSIE
National Cheng Kung University

Requirements Statements

- ❑ A chocolate boiler is used to boil chocolate.
- ❑ Before boiling chocolate with the boiler, you have to make sure that the boiler is now empty and then fill chocolate in. Besides, you can’t boil chocolate again while the chocolate has already been boiled.
- ❑ After boiling, it is time to drain out the boiled chocolate and make the boiler empty again.
- ❑ In order to prevent some unexpected situation, it is not allowed to have multiple instances of the chocolate boiler in the system.

Requirements Statements$_1$

- A chocolate boiler is used to boil chocolate.

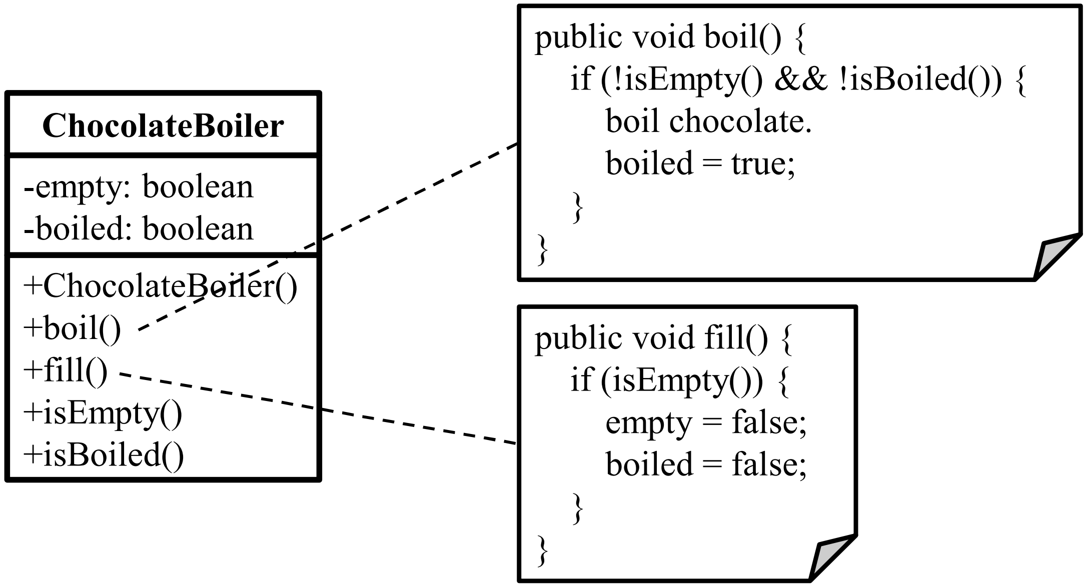


ChocolateBoiler

+ChocolateBoiler()
+boil()

Requirements Statements$_2$

- Before boiling chocolate with the boiler, you have to make sure that the boiler is now empty and then fill chocolate in. Besides, you can't boil chocolate again while the chocolate has already been boiled.



ChocolateBoiler

-empty: boolean
-boiled: boolean

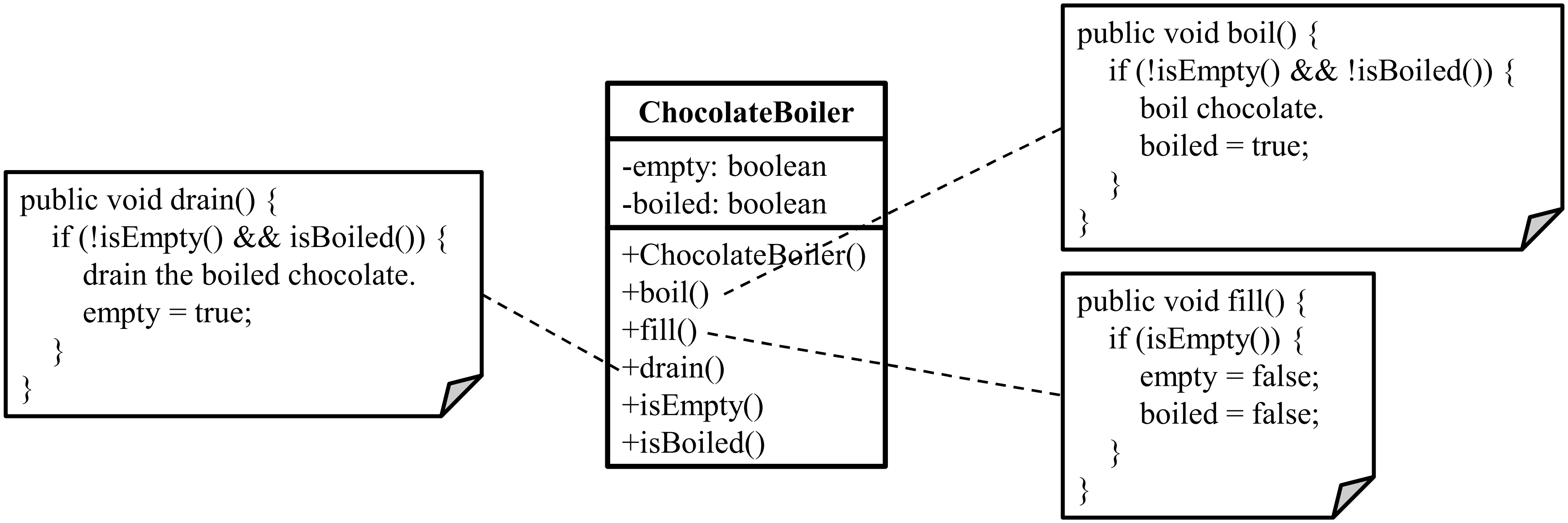
+ChocolateBoiler()
+boil()
+fill()
+isEmpty()
+isBoiled()

```
public void boil() {
   if (!isEmpty() && !isBoiled()) {
      boil chocolate.
      boiled = true;
   }
}
```

```
public void fill() {
   if (isEmpty()) {
      empty = false;
      boiled = false;
   }
}
```

Requirements Statements$_3$

- After boiling, it is time to drain out the boiled chocolate and make the boiler empty again.



ChocolateBoiler

-empty: boolean
-boiled: boolean

+ChocolateBoiler()
+boil()
+fill()
+drain()
+isEmpty()
+isBoiled()

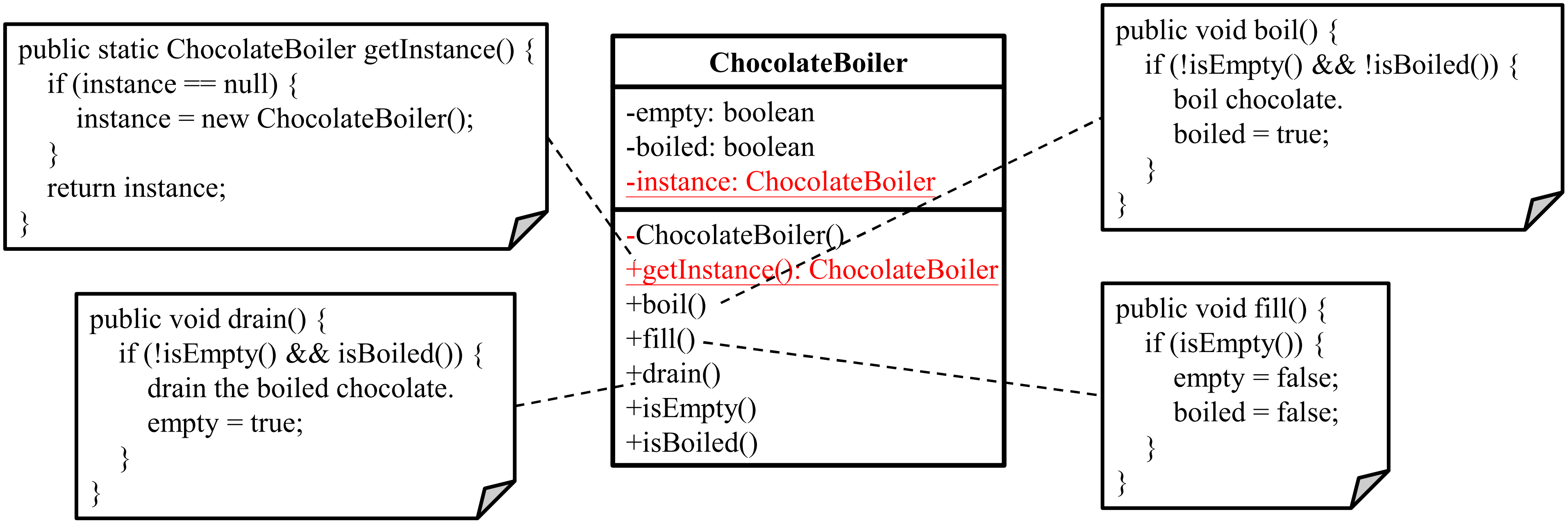
```
public void boil() {
    if (!isEmpty() && !isBoiled()) {
        boil chocolate.
        boiled = true;
    }
}
```

```
public void fill() {
    if (isEmpty()) {
        empty = false;
        boiled = false;
    }
}
```

```
public void drain() {
    if (!isEmpty() && isBoiled()) {
        drain the boiled chocolate.
        empty = true;
    }
}
```

Requirements Statements$_4$

- In order to prevent some unexpected situation, it is not allowed to have multiple instances of the chocolate boiler in the system.



```
public static ChocolateBoiler getInstance() {
    if (instance == null) {
        instance = new ChocolateBoiler();
    }
    return instance;
}
```

ChocolateBoiler

-empty: boolean
-boiled: boolean
-instance: ChocolateBoiler

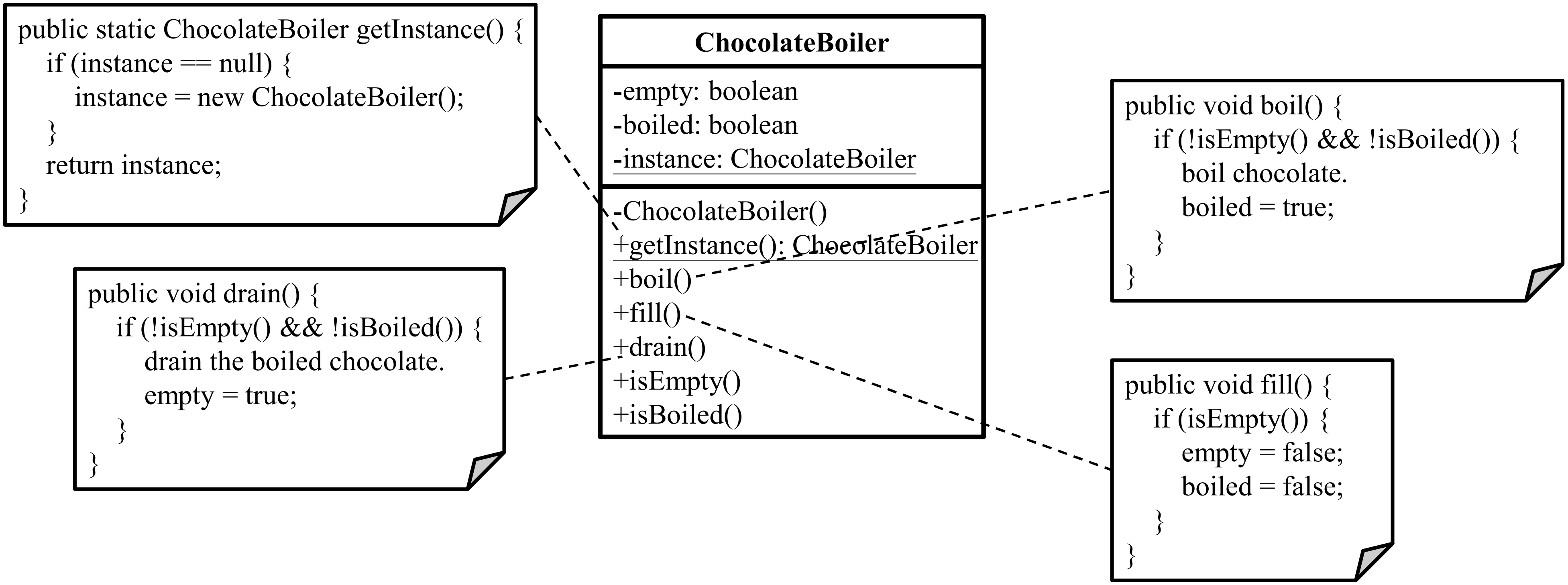
-ChocolateBoiler()
+getInstance(): ChocolateBoiler
+boil()
+fill()
+drain()
+isEmpty()
+isBoiled()

```
public void boil() {
    if (!isEmpty() && !isBoiled()) {
        boil chocolate.
        boiled = true;
    }
}
```

```
public void drain() {
    if (!isEmpty() && isBoiled()) {
        drain the boiled chocolate.
        empty = true;
    }
}
```

```
public void fill() {
    if (isEmpty()) {
        empty = false;
        boiled = false;
    }
}
```

Initial Design



```
public static ChocolateBoiler getInstance() {
   if (instance == null) {
      instance = new ChocolateBoiler();
   }
   return instance;
}
```

```
public void drain() {
   if (!isEmpty() && !isBoiled()) {
      drain the boiled chocolate.
      empty = true;
   }
}
```

ChocolateBoiler

```
-empty: boolean
-boiled: boolean
-instance: ChocolateBoiler

-ChocolateBoiler()
+getInstance(): ChocolateBoiler
+boil()
+fill()
+drain()
+isEmpty()
+isBoiled()
```

```
public void boil() {
   if (!isEmpty() && !isBoiled()) {
      boil chocolate.
      boiled = true;
   }
}
```

```
public void fill() {
   if (isEmpty()) {
      empty = false;
      boiled = false;
   }
}
```

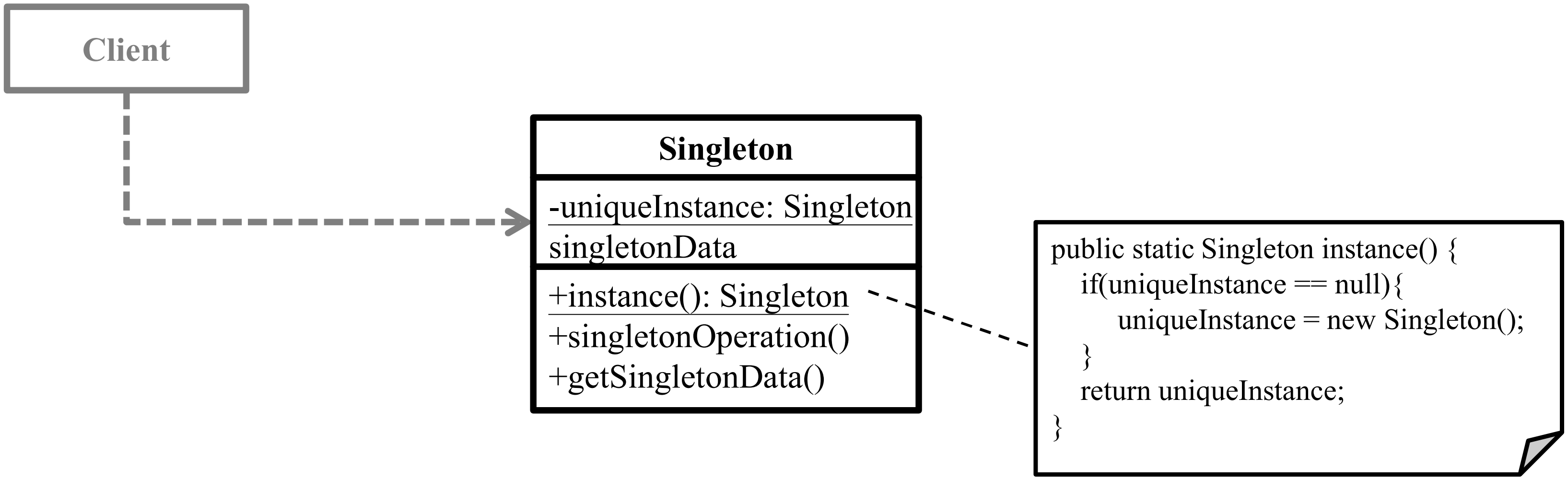
Recurrent Problem

- It's important for some classes to have exactly one instance and ensure that the instance is easily accessible.
- A global variable makes an object accessible, but it doesn't keep you from instantiating multiple objects.

Intent

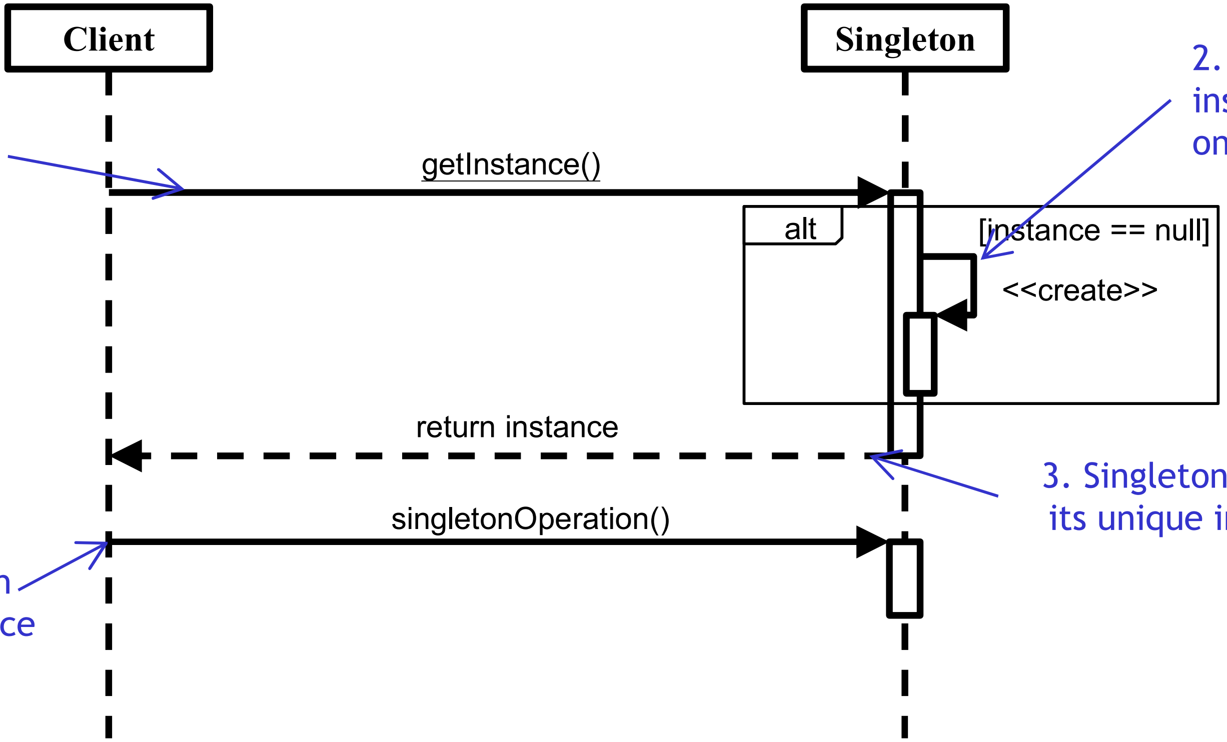
- ❑ Ensure a class only has one instance, and provide a global point of access to it.

Singleton Structure$_1$



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

Singleton Structure$_2$



1. Obtain Singleton instance through getInstance()

2. If the instance is not instantiated yet, create one

3. Singleton returns its unique instance

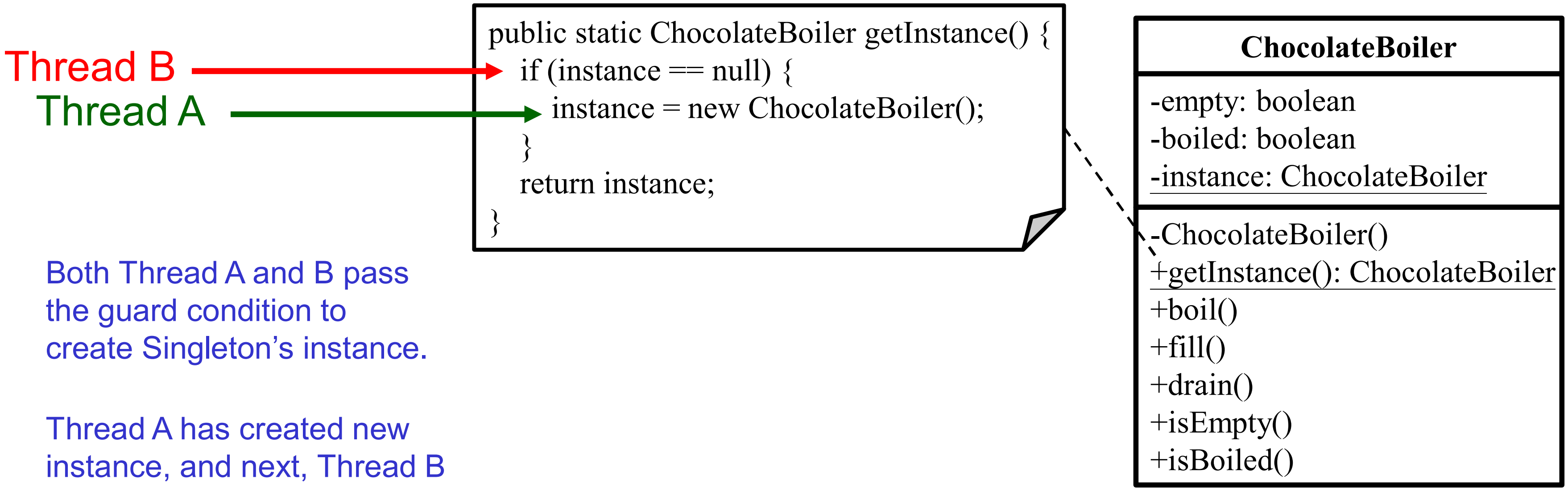
4. Client performs singletonOperation() with Singleton's unique instance

Singleton Structure$_3$

	Instantiation	Use	Termination
Singleton	Singleton creates itself and make sure there is only one instance.	Client gets the unique instance of Singleton from Singleton, and performs its operation(s).	Don’t Care

Singleton with Multi-threading Issues$_1$

- **Issue:** In a multi-threading situation, if more than one thread request Singleton’s instance, it may result in multiple instances which violates Singleton’s intent.



```
public static ChocolateBoiler getInstance() {
   if (instance == null) {
      instance = new ChocolateBoiler();
   }
   return instance;
}
```

ChocolateBoiler

-empty: boolean
-boiled: boolean
-instance: ChocolateBoiler

-ChocolateBoiler()
+getInstance(): ChocolateBoiler
+boil()
+fill()
+drain()
+isEmpty()
+isBoiled()

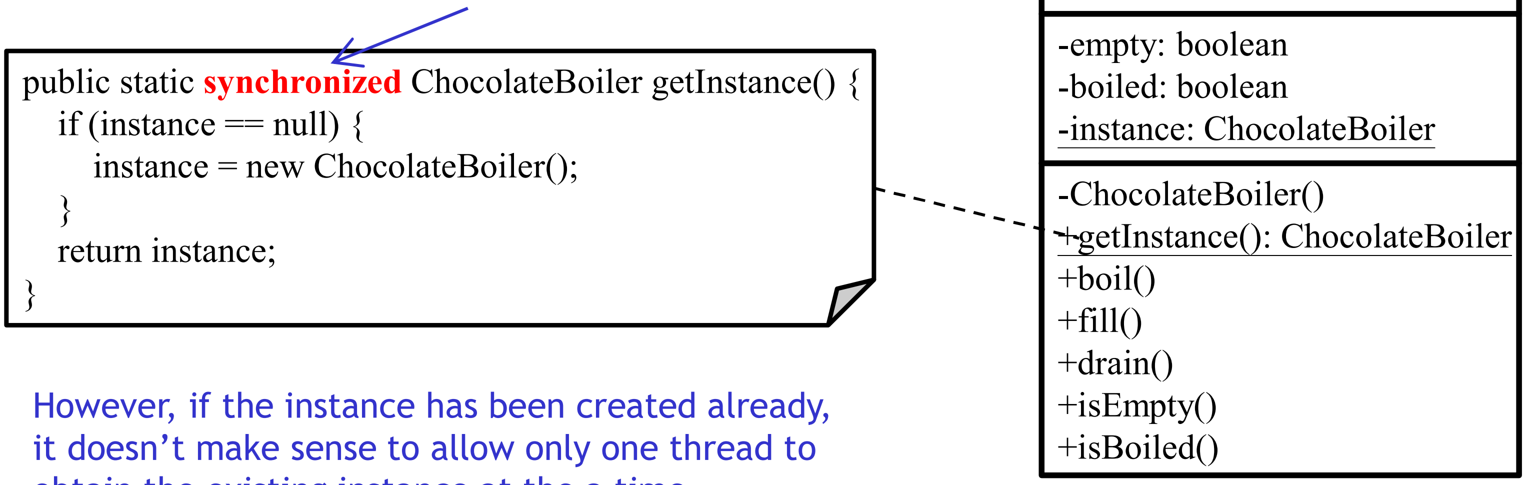
Both Thread A and B pass the guard condition to create Singleton’s instance.

Thread A has created new instance, and next, Thread B will also create another instance.

Singleton with Multi-threading Issues$_2$

- **Solution 1:** Synchronizing the getInstance() method

This keyword in Java permits only one thread to enter the method at a time.

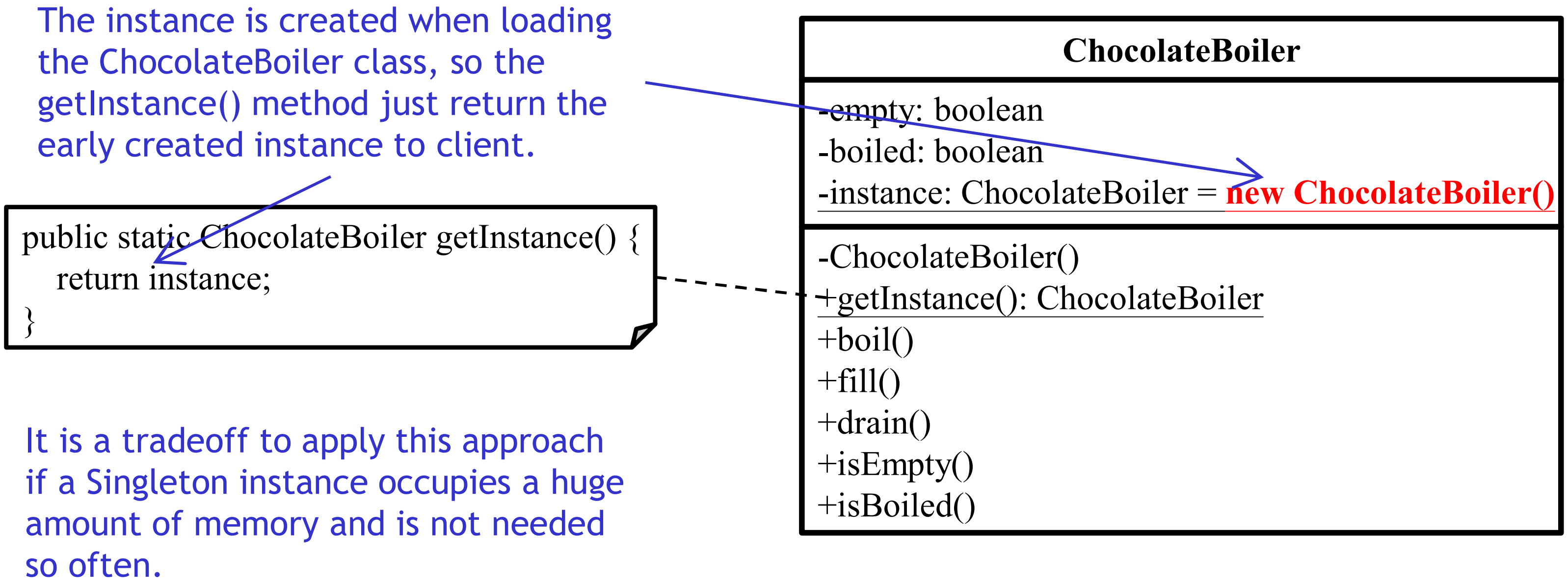


```
public static synchronized ChocolateBoiler getInstance() {
    if (instance == null) {
        instance = new ChocolateBoiler();
    }
    return instance;
}
```

However, if the instance has been created already, it doesn’t make sense to allow only one thread to obtain the existing instance at the a time.

Singleton with Multi-threading Issues$_3$

❑ **Solution 2:** Eager creation



The instance is created when loading the ChocolateBoiler class, so the getInstance() method just return the early created instance to client.

```
public static ChocolateBoiler getInstance() {
    return instance;
}
```

ChocolateBoiler

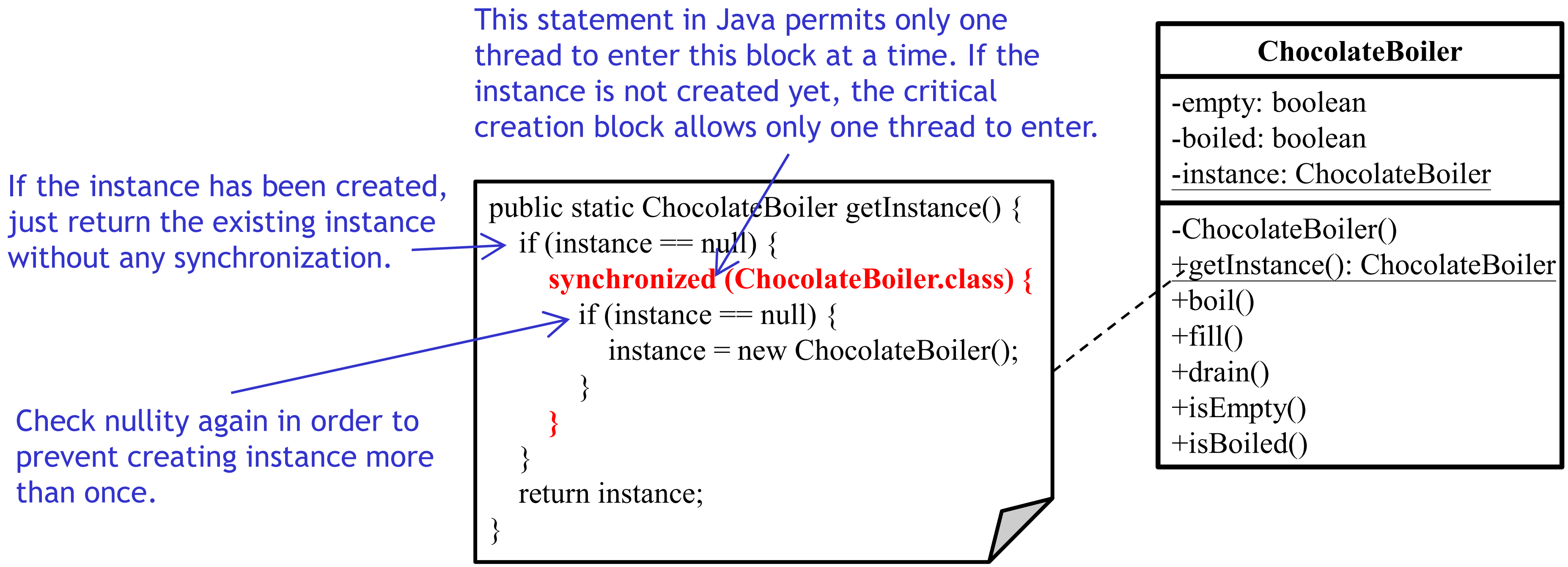
-empty: boolean
-boiled: boolean
-instance: ChocolateBoiler = **new ChocolateBoiler()**

-ChocolateBoiler()
+getInstance(): ChocolateBoiler
+boil()
+fill()
+drain()
+isEmpty()
+isBoiled()

It is a tradeoff to apply this approach if a Singleton instance occupies a huge amount of memory and is not needed so often.

Singleton with Multi-threading Issues$_4$

- **Solution 3:** Synchronizing the getInstance() method with double-checked locking



This statement in Java permits only one thread to enter this block at a time. If the instance is not created yet, the critical creation block allows only one thread to enter.

If the instance has been created, just return the existing instance without any synchronization.

```
public static ChocolateBoiler getInstance() {
  if (instance == null) {
    synchronized (ChocolateBoiler.class) {
      if (instance == null) {
        instance = new ChocolateBoiler();
      }
    }
  }
  return instance;
}
```

Check nullity again in order to prevent creating instance more than once.

ChocolateBoiler

-empty: boolean
-boiled: boolean
-instance: ChocolateBoiler

-ChocolateBoiler()
+getInstance(): ChocolateBoiler
+boil()
+fill()
+drain()
+isEmpty()
+isBoiled()