Singleton, Factory, Adaptor

Imagine a 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.

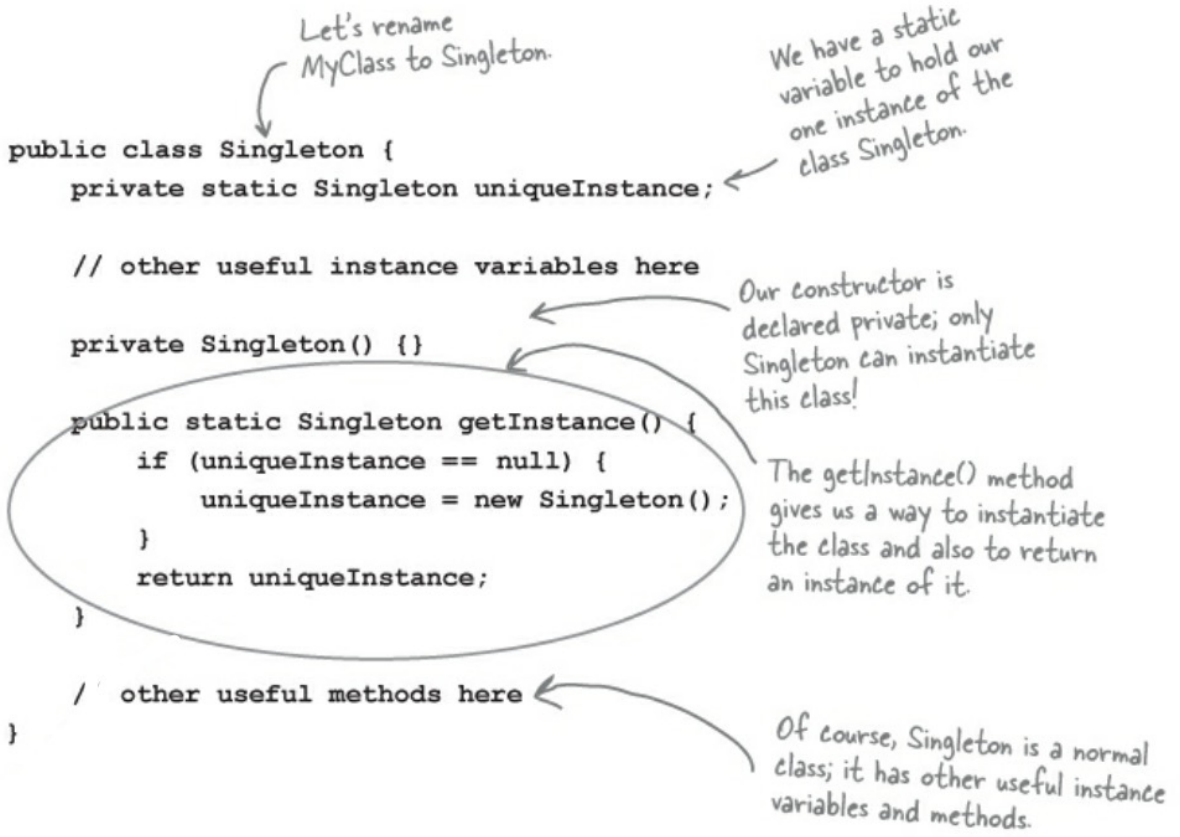
Questions to ask

- Given that there is only one physical boiler, is it wise to give its control to more than one instance of boiler?
- How would you solve it?
  - Make sure that there is only one instance of boiler

Singleton Pattern

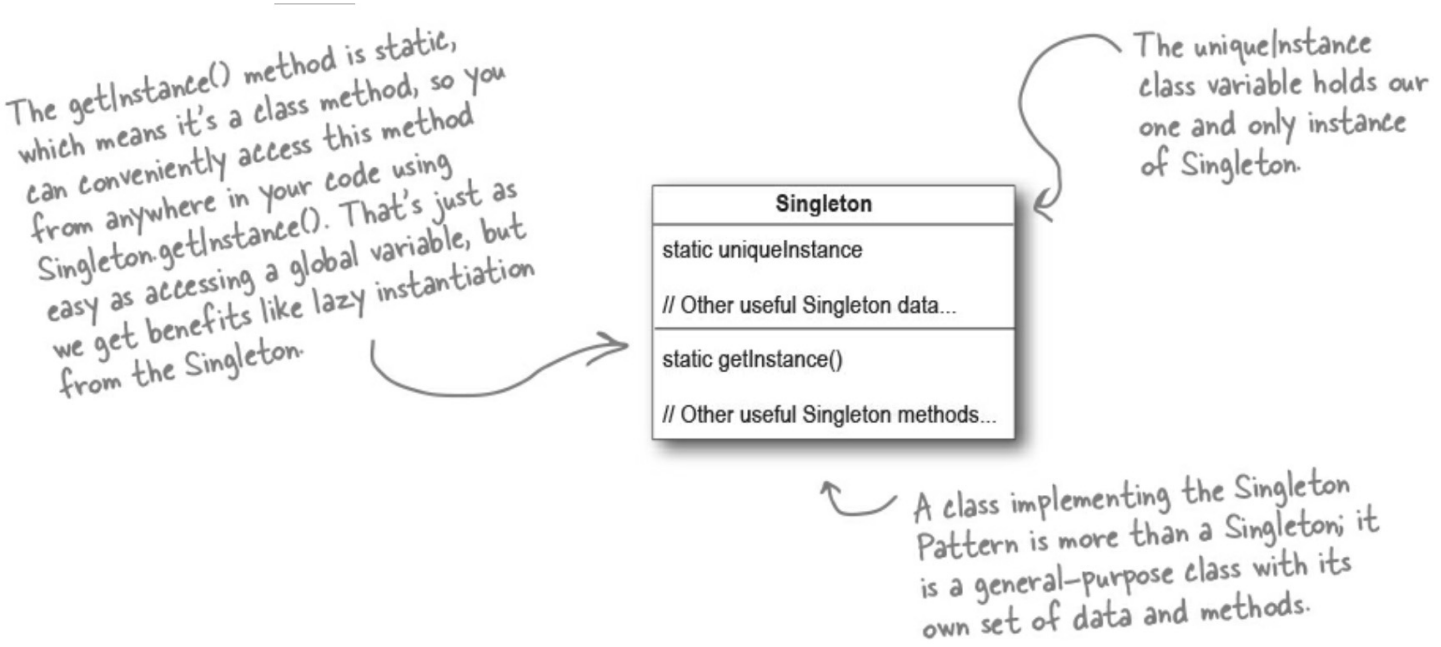
- There are many objects we only need one of: thread pools, caches, dialog boxes, objects that handle preferences and registry settings, objects used for logging, and objects that act as device drivers to devices like printers and graphics cards.

The Classic Implementation



Singleton Pattern

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



Class Exercise

- Modify the Chocolate Boiler code to use singleton
- Write driver code to show that only one instance is available
- Time: 15 minutes

Singleton for rescue

- Would this solve the problem?
- How about in cases of multithreading?

Could the addition of threads have caused this? Isn't it the case that once we've set the uniqueInstance variable to the sole instance of ChocolateBoiler, all calls to getInstance() should return the same instance? Right?

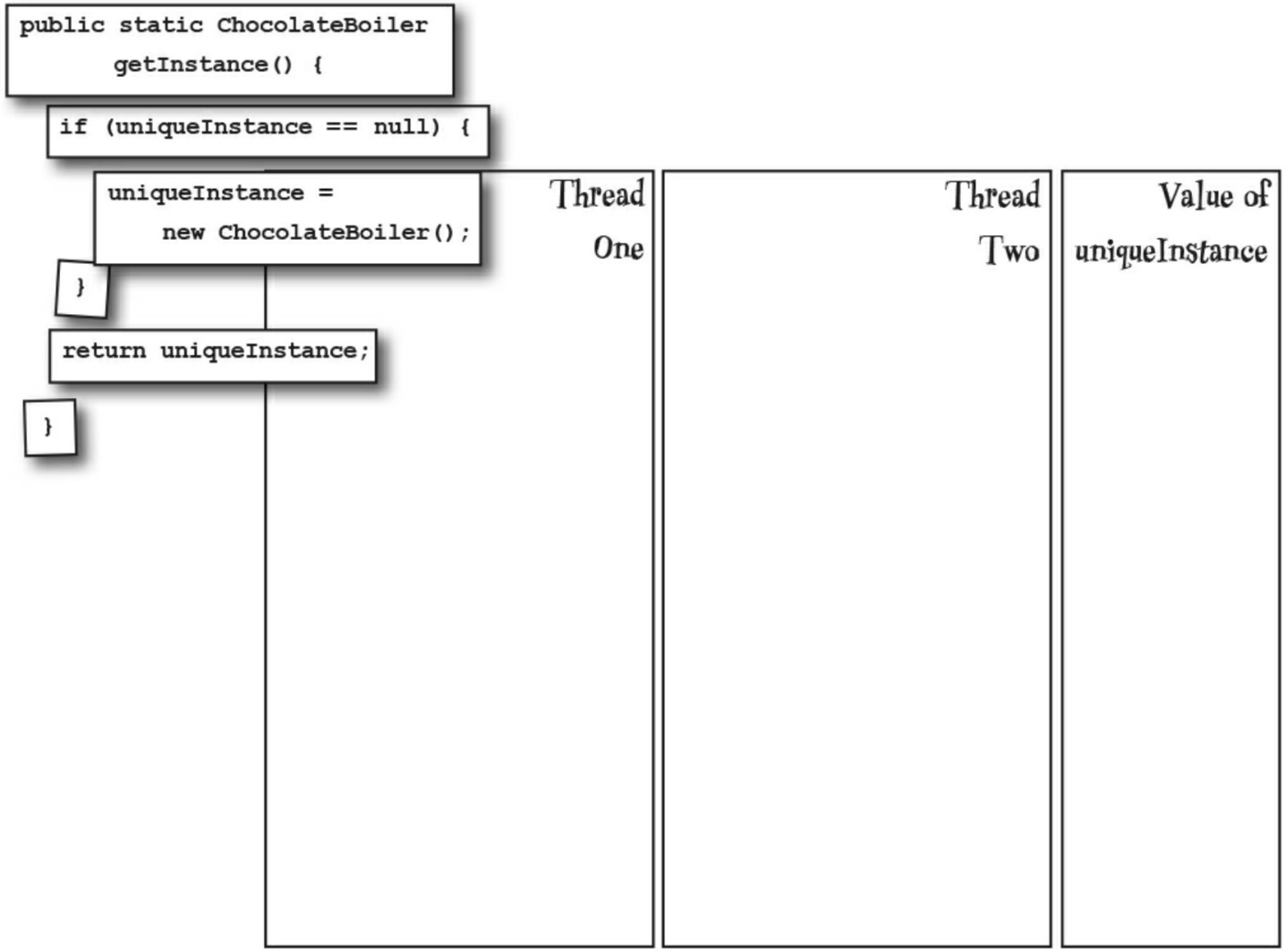
Singleton & Multi-threading

What can happen in this code with two threads?

```
ChocolateBoiler boiler =
        ChocolateBoiler.getInstance();
boiler.fill();
boiler.boil();
boiler.drain();
```

Singleton & Multithreading

- Show how it can go wrong?
- How will you fix it?



Singleton & Multi-threading

Use locks in Python or C++

```
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.

Cost of Synchronization

- Do nothing if the performance of getInstance() isn't critical to your application

- Move to an eagerly created instance rather than a lazily created one
  - If your application always creates and uses an instance of the Singleton, or the overhead of creation and runtime aspects of the Singleton isn't onerous

- Use "double-checked locking" to reduce the use of synchronization in getInstance()
  - With double-checked locking, we first check to see if an instance is created, and if not, THEN we synchronize. This way, we only synchronize the first time through, just what we want.

Double-checked locking

- If performance is an issue in your use of the getInstance() method, then this method of implementing the Singleton can drastically reduce the overhead.
- Python does not have volatile
  - Does not need it

```
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!

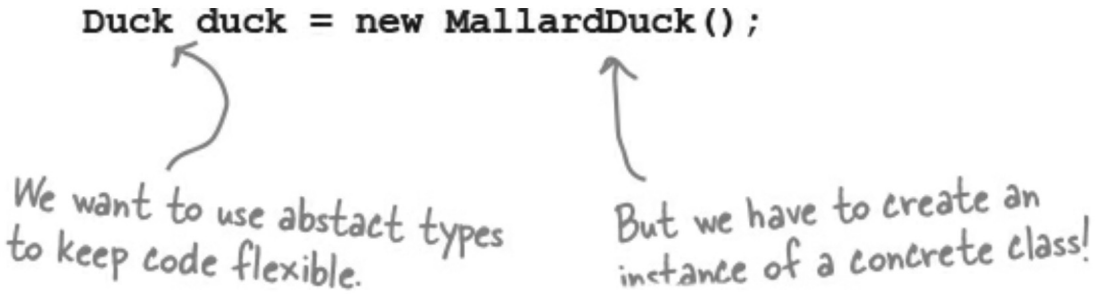
Once in the block, check again and if it's still null, create an instance.

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

The Factory Pattern

Loosely-coupled OO Design

- Minimize Dependencies
  - new == concrete implementation



Imagine you started your own Pizza Shop

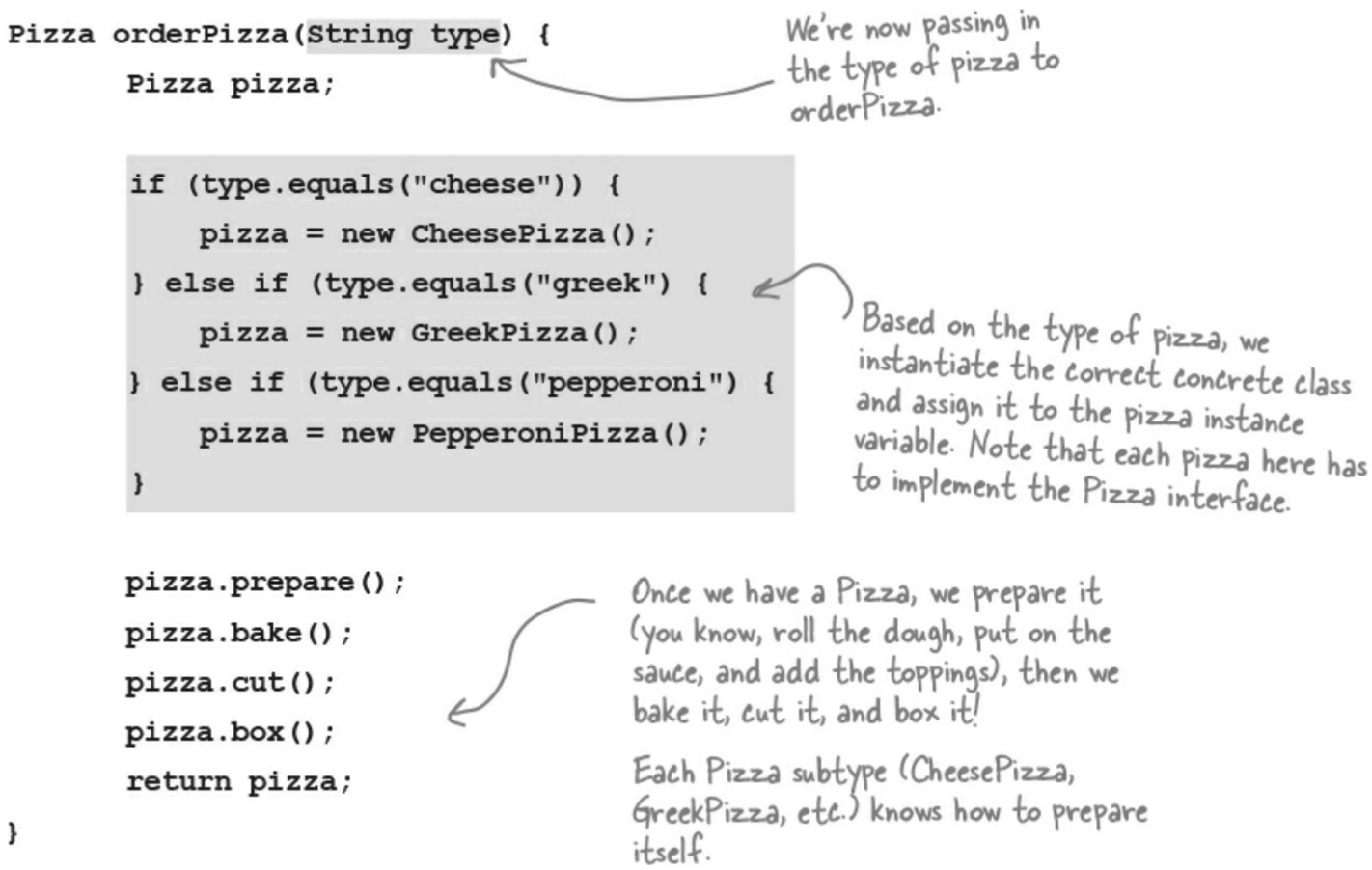
- But you need more than one type of pizza
  - Think of your coffee shop

```
Pizza orderPizza() {
        Pizza pizza = new Pizza();

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
}
```

For flexibility, we really want this to be an abstract class or interface, but unfortunately we can't directly instantiate either of those.

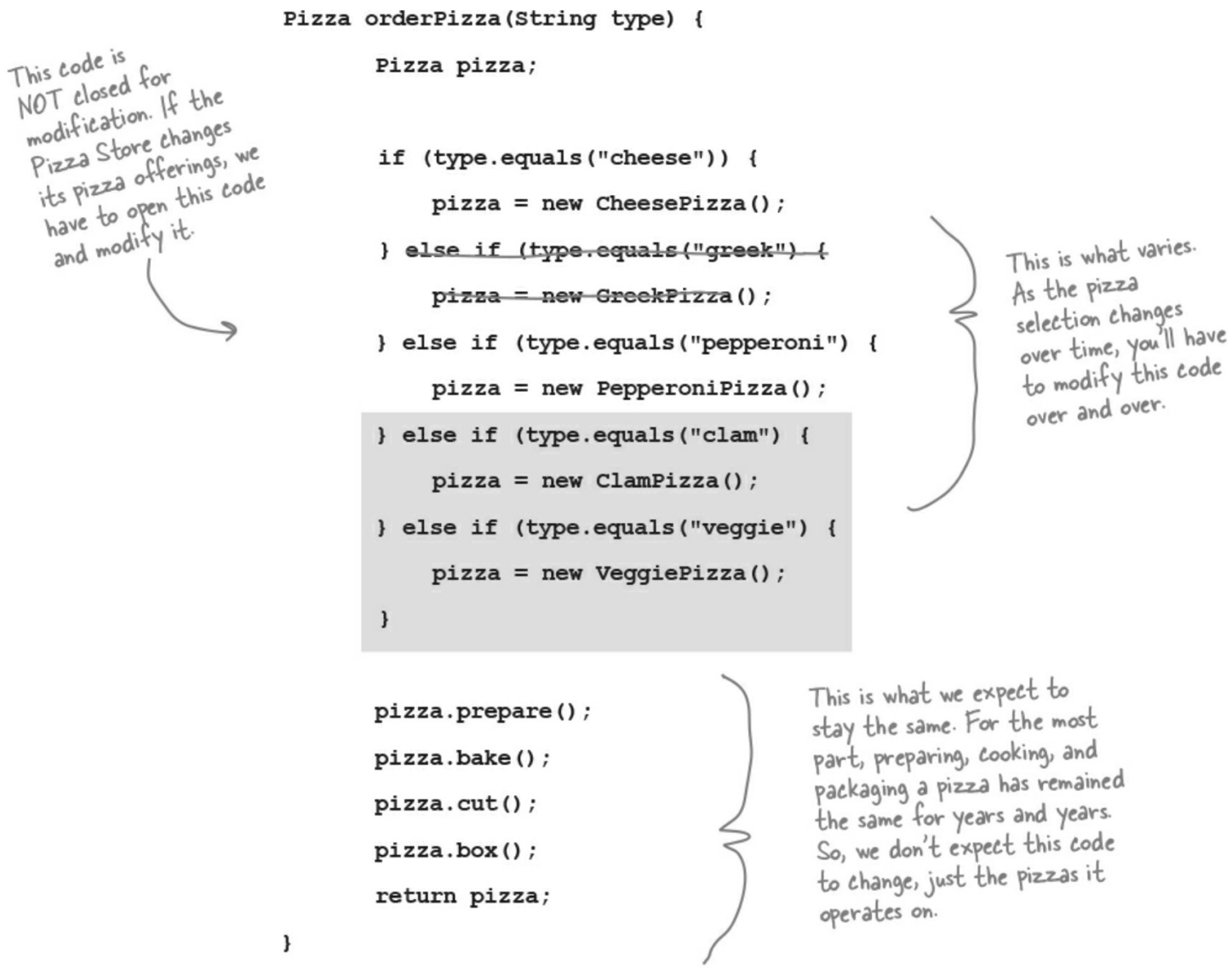
Add more types

- Add more types of Pizza…
- You may also find that some pizza types are not selling so need to stop making those

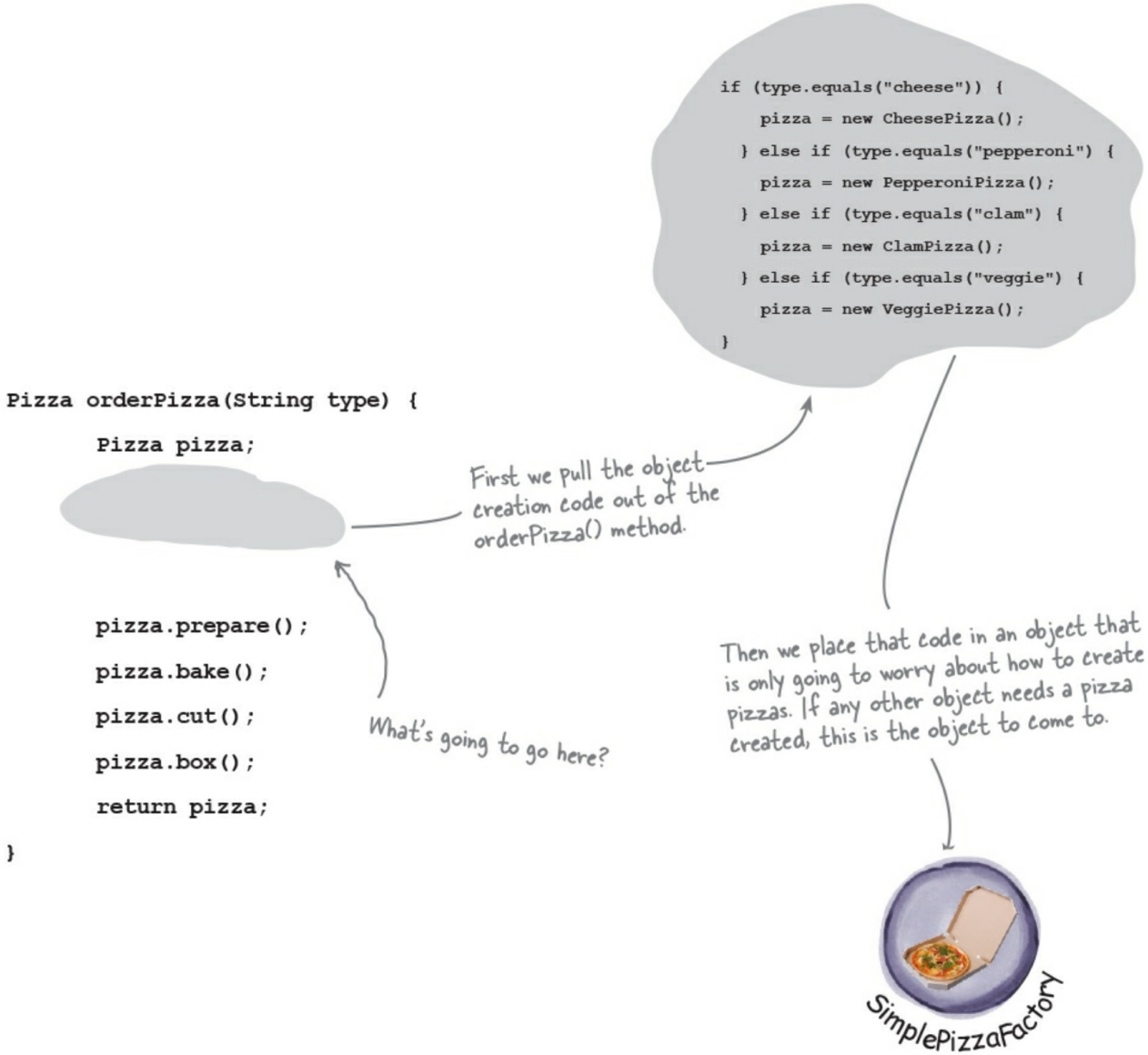


Change the menu

- You are modifying a running code
- Solution
  - Encapsulating object creation



Encapsulating object creation



A Simple Factory

First we give PizzaStore a reference to a SimplePizzaFactory.

PizzaStore gets the factory passed to it in the constructor.

And the orderPizza() method uses the factory to create its pizzas by simply passing on the type of the order.

Notice that we've replaced the new operator with a createPizza method in the factory object. No more concrete instantiations here!

```
public class PizzaStore {
    SimplePizzaFactory factory;

    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = factory.createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }

    // other methods here
}
```

Here's our new class, the SimplePizzaFactory. It has one job in life: creating pizzas for its clients.

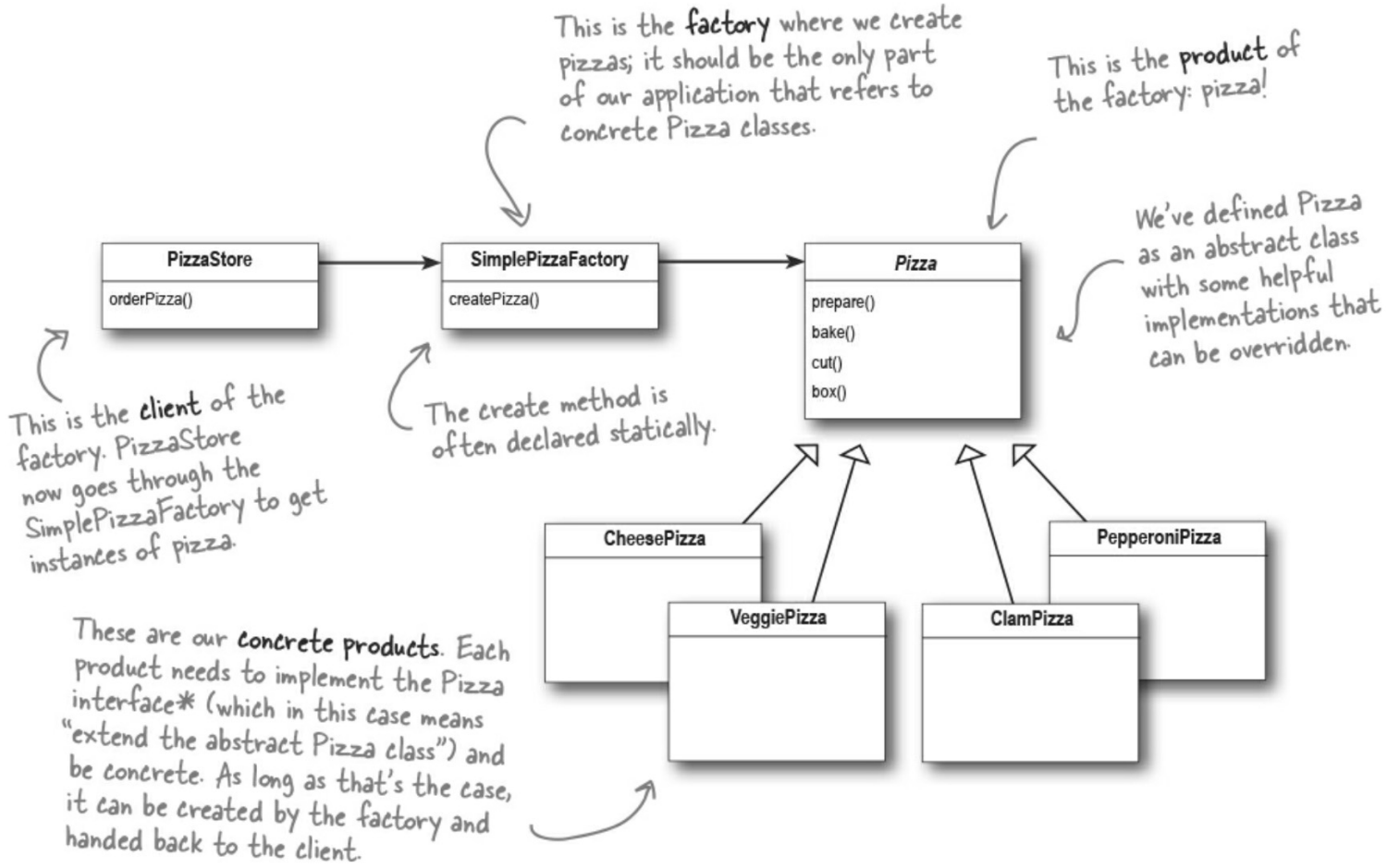
First we define a createPizza() method in the factory. This is the method all clients will use to instantiate new objects.

Here's the code we plucked out of the orderPizza() method.

This code is still parameterized by the type of the pizza, just like our original orderPizza() method was.

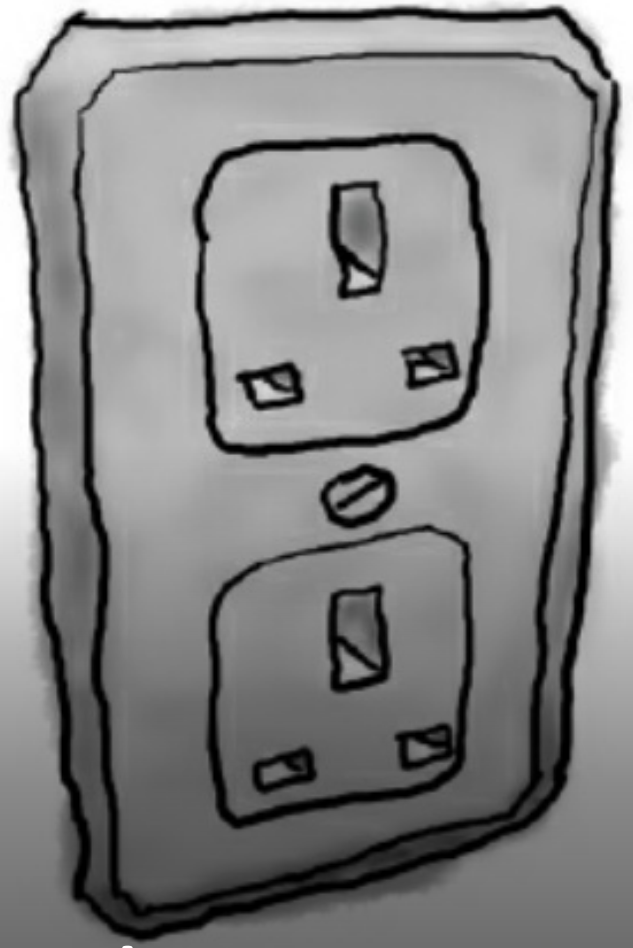
```
public class SimplePizzaFactory {

    public Pizza createPizza(String type) {
        Pizza pizza = null;

        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("clam")) {
            pizza = new ClamPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }
        return pizza;
    }
}
```

A Simple Factory



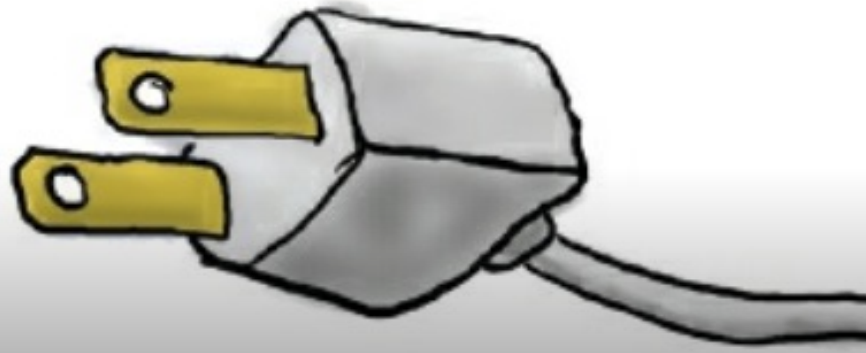
Things to remember

- No class should derive from a concrete class.
- No method should override an implemented method of any of its base classes.



AC Power Adapter

US Standard AC Plug

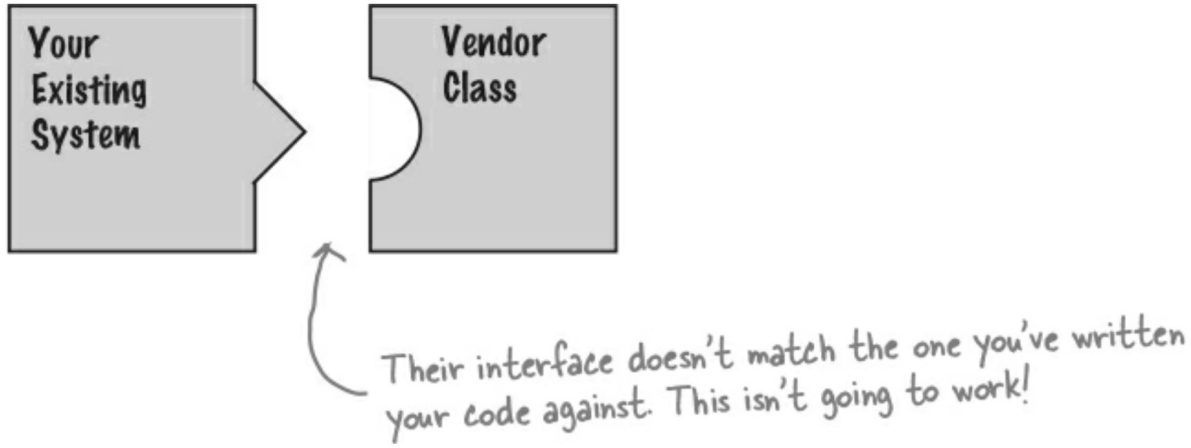


The US laptop expects another interface.

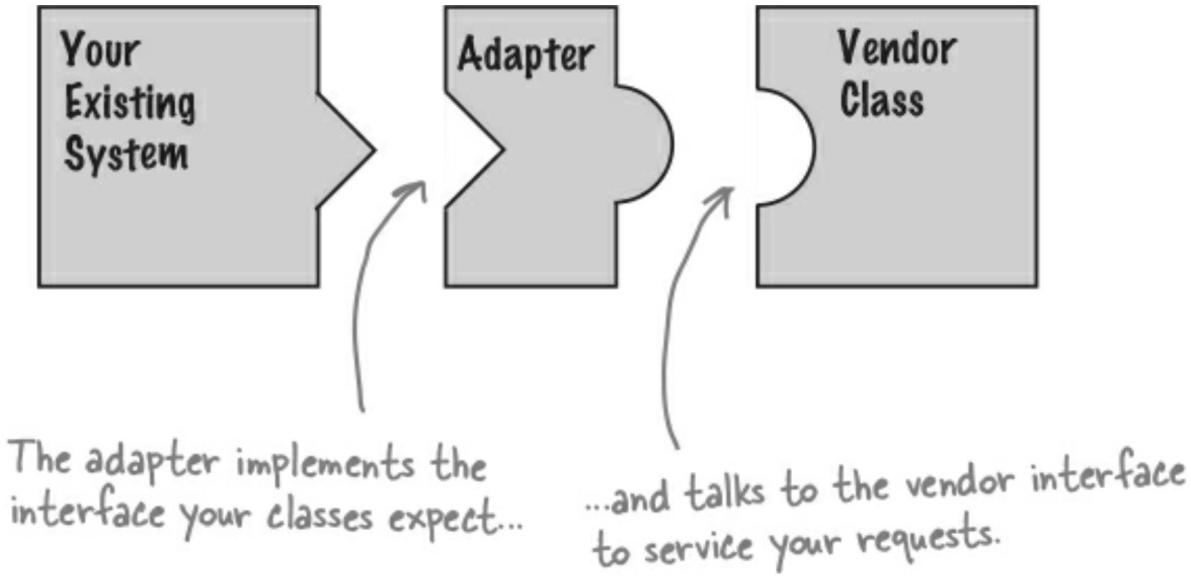
Adapter Pattern

Adapter Pattern

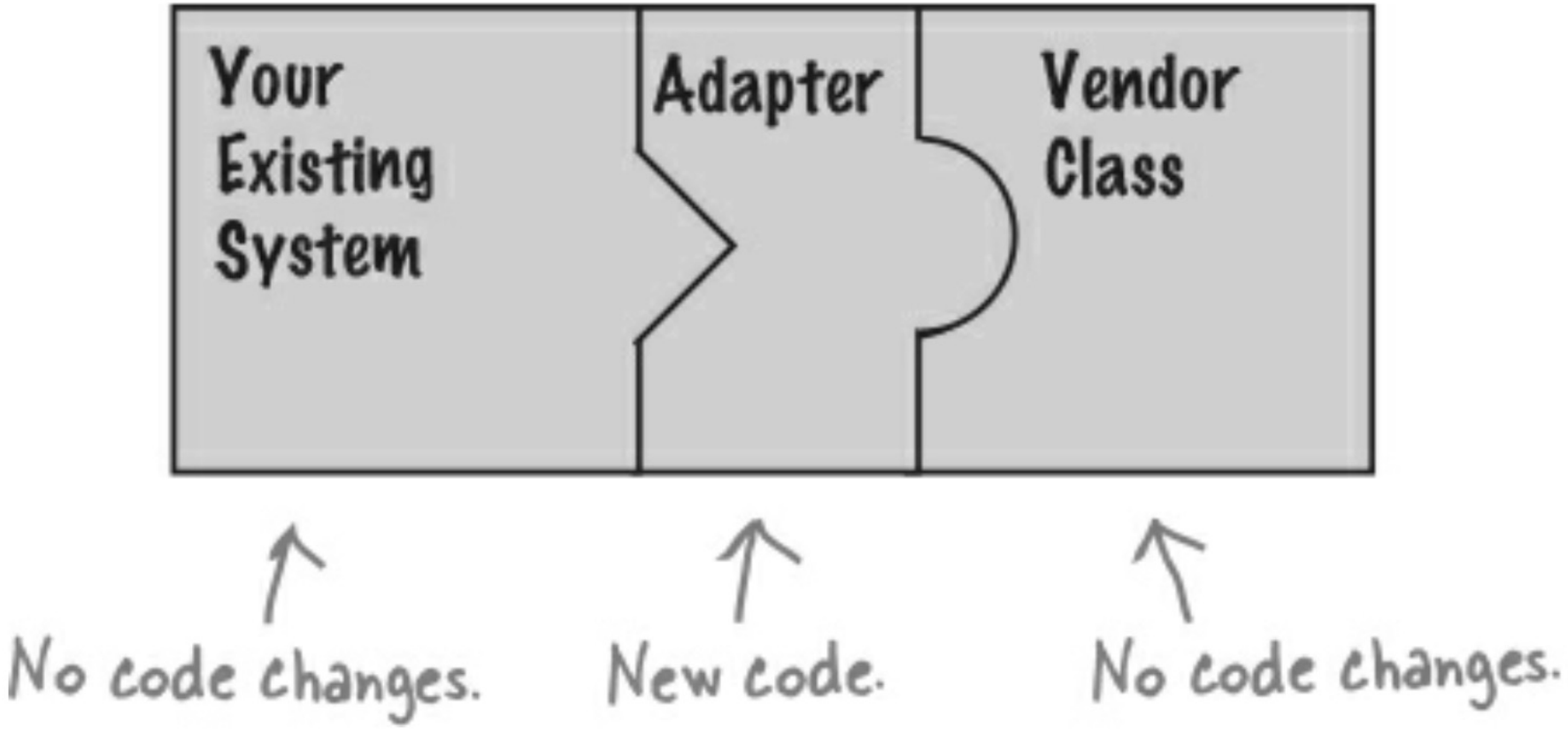
- You've got an existing software system that you need to work a new vendor class library into
  - But the new vendor designed their interfaces differently than the last vendor



Okay, you don't want to solve the problem by changing your existing code (and you can't change the vendor's code). So what do you do? Well, you can write a class that adapts the new vendor interface into the one you're expecting.



Adapter Pattern



Adapter Pattern

- If it walks like a duck and quacks like a duck, then it must might be a duck turkey wrapped with a duck adapter...

```
public interface Duck {
    public void quack();
    public void fly();
}
```

This time around, our ducks implement a Duck interface that allows Ducks to quack and fly.

Adapter Pattern

```
public class MallardDuck implements Duck {
    public void quack() {
        System.out.println("Quack");
    }

    public void fly() {
        System.out.println("I'm flying");
    }
}
```

Simple implementations: MallardDuck just prints out what it is doing.

Now it's time to meet the newest fowl on the block:

```
public interface Turkey {
    public void gobble();
    public void fly();
}
```

Turkeys don't quack, they gobble.

Turkeys can fly, although they can only fly short distances.

```
public class WildTurkey implements Turkey {
    public void gobble() {
        System.out.println("Gobble gobble");
    }

    public void fly() {
        System.out.println("I'm flying a short distance");
    }
}
```

Here's a concrete implementation of Turkey; like MallardDuck, it just prints out its actions.

Writing an Adapter

First, you need to implement the interface of the type you're adapting to. This is the interface your client expects to see.

```
public class TurkeyAdapter implements Duck {
    Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    public void quack() {
        turkey.gobble();
    }

    public void fly() {
        for(int i=0; i < 5; i++) {
            turkey.fly();
        }
    }
}
```

Next, we need to get a reference to the object that we are adapting; here we do that through the constructor.

Now we need to implement all the methods in the interface; the quack() translation between classes is easy: just call the gobble() method.

Even though both interfaces have a fly() method, Turkeys fly in short spurts—they can't do long-distance flying like ducks. To map between a Duck's fly() method and a Turkey's, we need to call the Turkey's fly() method five times to make up for it.

Testing an Adapter

```
public class DuckTestDrive {
    public static void main(String[] args) {
        Duck duck = new MallardDuck();

        Turkey turkey = new WildTurkey();
        Duck turkeyAdapter = new TurkeyAdapter(turkey);

        System.out.println("The Turkey says...");
        turkey.gobble();
        turkey.fly();

        System.out.println("\nThe Duck says...");
        testDuck(duck);

        System.out.println("\nThe TurkeyAdapter says...");
        testDuck(turkeyAdapter);
    }

    static void testDuck(Duck duck) {
        duck.quack();
        duck.fly();
    }
}
```

Let's create a Duck...

...and a Turkey.

And then wrap the turkey in a TurkeyAdapter, which makes it look like a Duck.

Then, let's test the Turkey: make it gobble, make it fly.

Now let's test the duck by calling the testDuck() method, which expects a Duck object.

Now the big test: we try to pass off the turkey as a duck...

Here's our testDuck() method; it gets a duck and calls its quack() and fly() methods.

Test run

```
File  Edit  Window  Help  Don'tForgetToDuck
%java DuckTestDrive
The Turkey says...
Gobble gobble
I'm flying a short distance

The Duck says...
Quack
I'm flying

The TurkeyAdapter says...
Gobble gobble
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
```

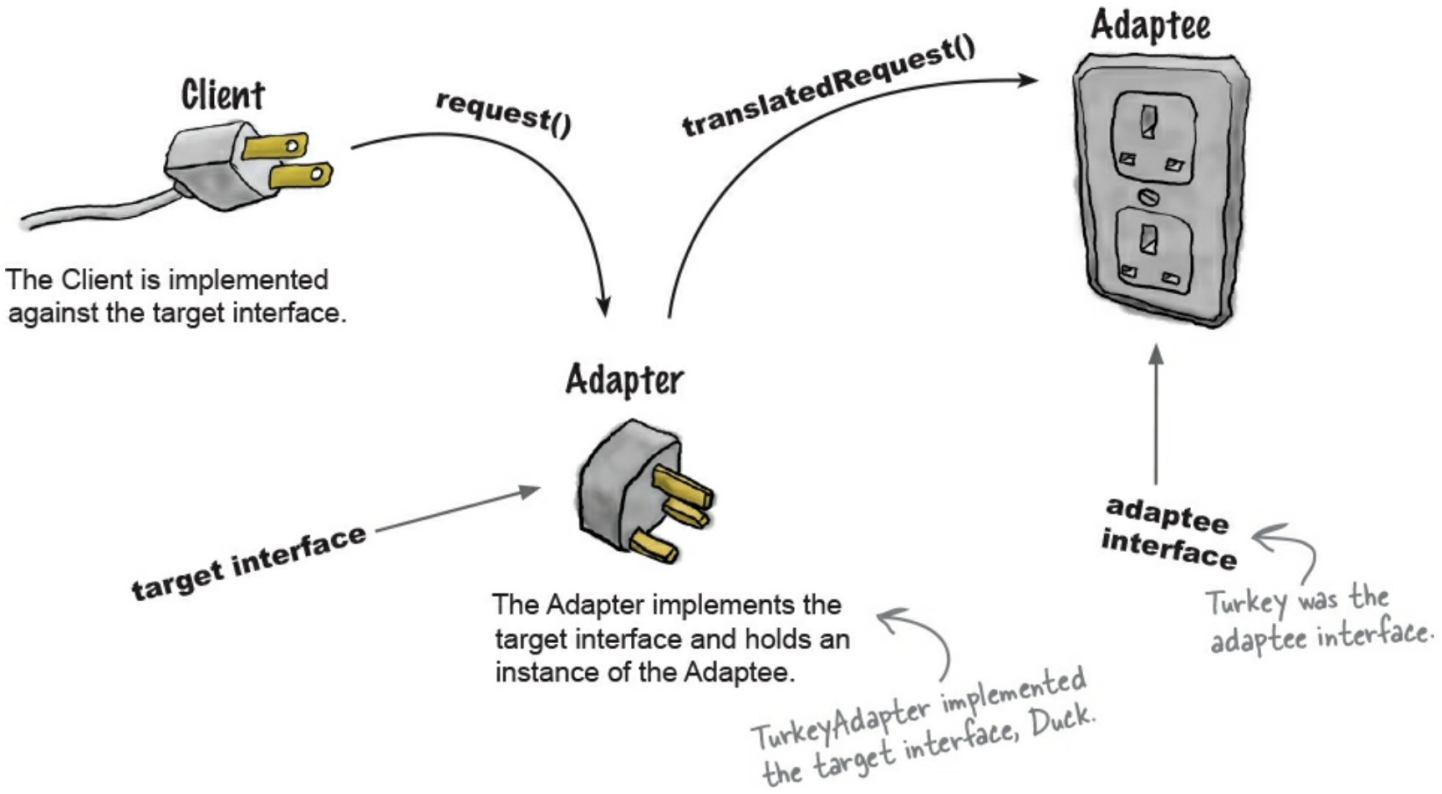
The Turkey gobbles and flies a short distance.

The Duck quacks and flies just like you'd expect.

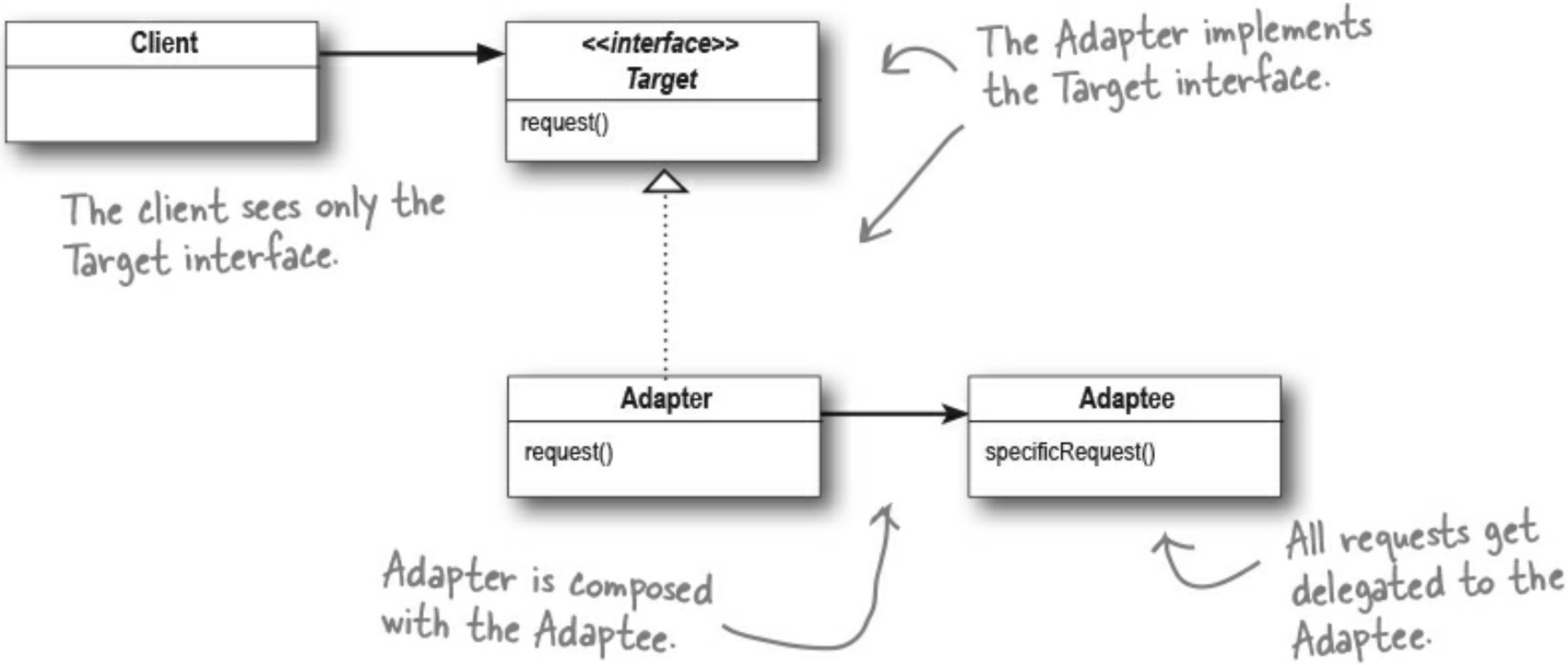
And the adapter gobbles when quack() is called and flies a few times when fly() is called. The testDuck() method never knows it has a turkey disguised as a duck!

Adapter Pattern

- The client makes a request to the adapter by calling a method on it using the target interface.
- The adapter translates the request into one or more calls on the adaptee using the adaptee interface.
- The client receives the results of the call and never knows there is an adapter doing the translation.



Adapter Pattern



Class Exercise

- Add a chicken adapter
  - Cluck sound
  - Flies 10 times to implement duck fly