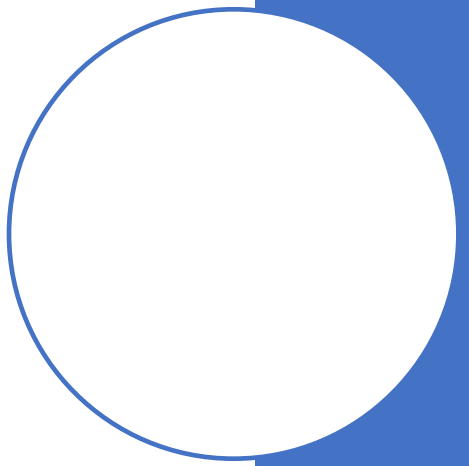
Singleton Pattern



Singleton

This pattern can be used for ensuring one and only one object is instantiated for a given class.

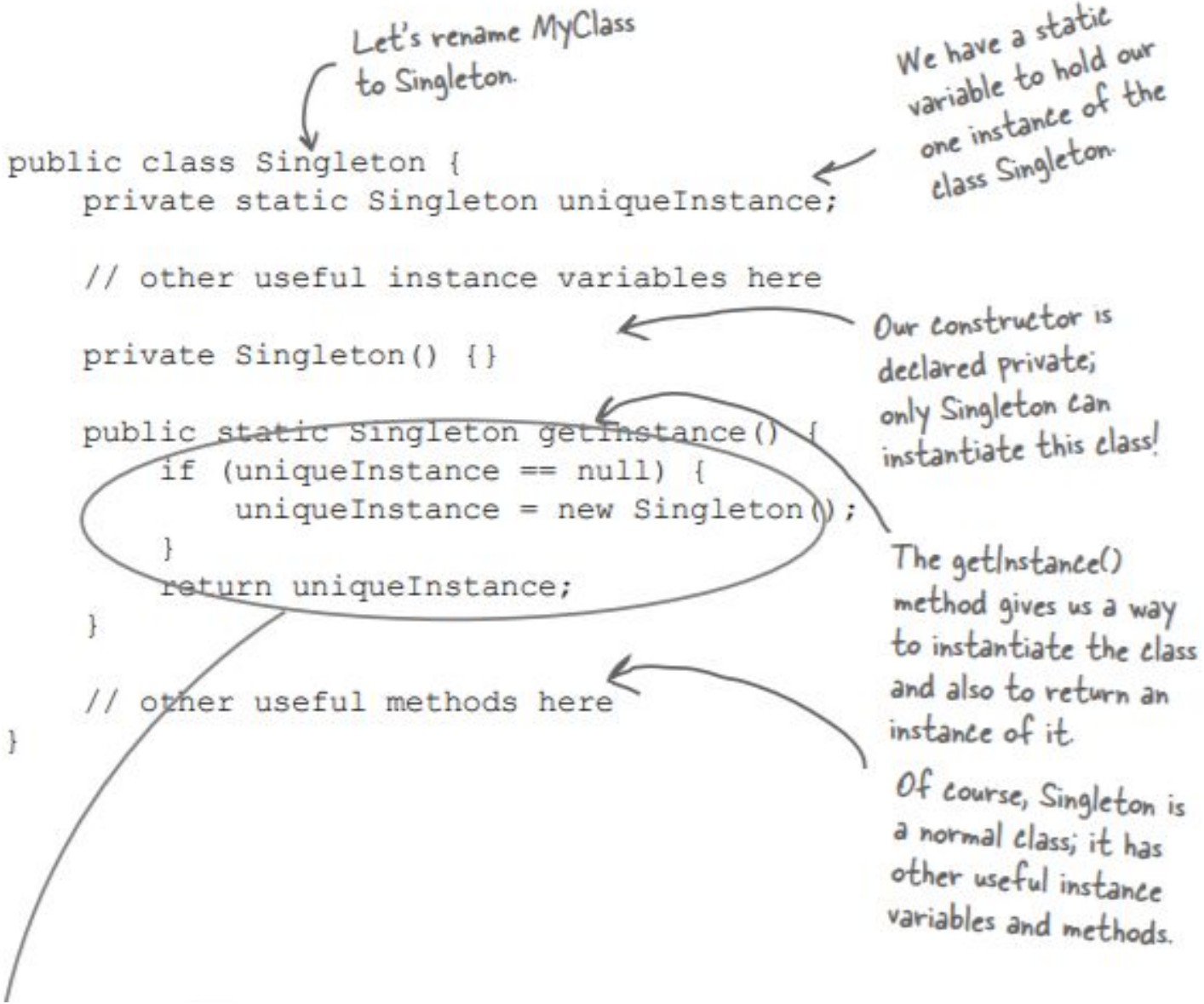
Examples: a single object for dialog boxes, objects that handle registry settings, objects used for logging, and objects that act as device drivers to devices like printers and graphics cards.

The Singleton Pattern also gives us a **global point of access**, just like a global variable, **but without the downsides**.

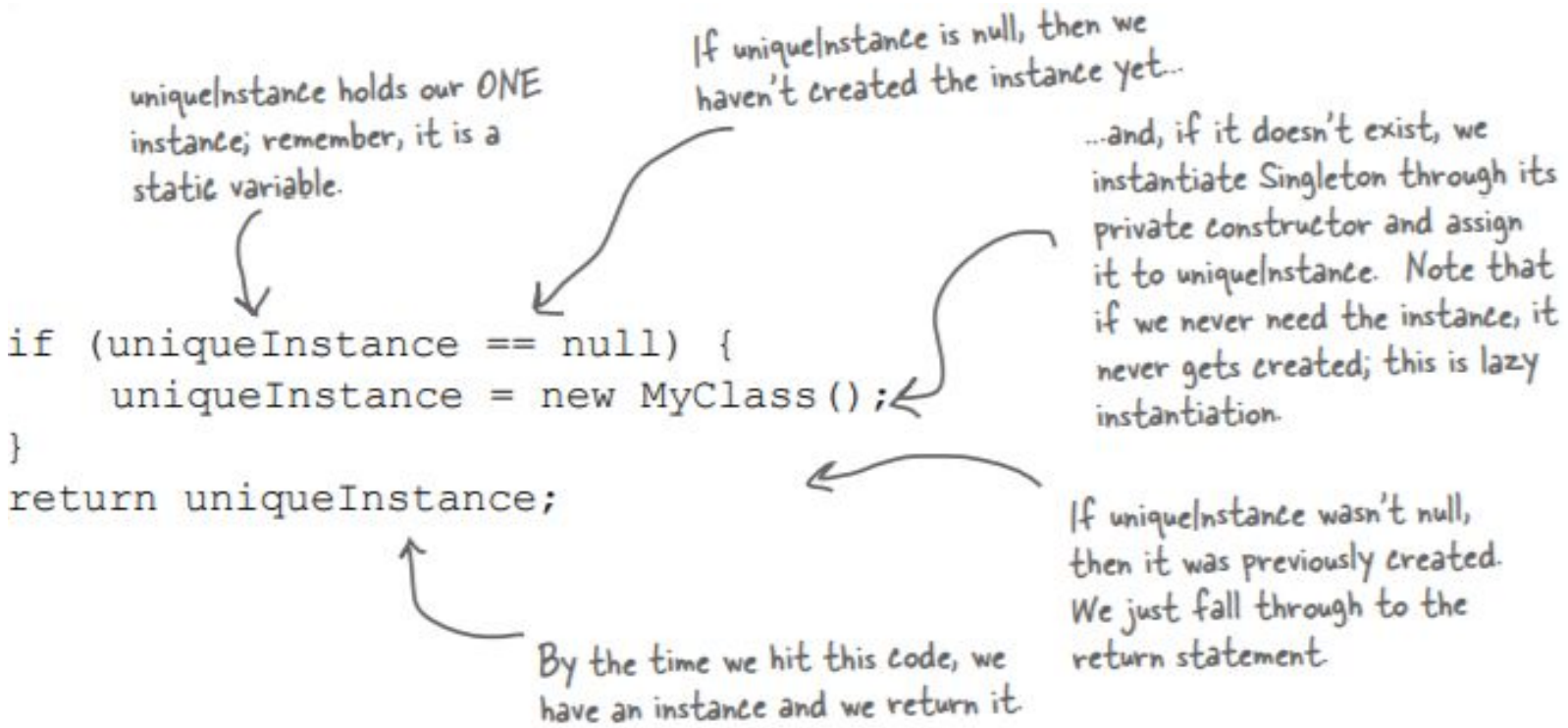
What downsides?

Well, here's one example: if you assign an object to a global variable, then that object might be created when your application begins. Right? What if this object is resource intensive and your application never ends up using it? As you will see, with the Singleton Pattern, we can create our objects only when they are needed.

Example: Implementation



Code up close



Chocolate Factory Example

Everyone knows that all modern chocolate factories have computer-controlled chocolate boilers.

The job of the boiler is to take in chocolate and milk, bring them to a boil, and then pass them on to the next phase of making chocolate bars.

In the next slide, we have the controller class for Choc-O-Holic, Inc.'s industrial strength Chocolate Boiler.

Check out the code; you'll notice they've tried to be very careful to ensure that bad things don't happen, like draining 500 gallons of un-boiled mixture, or filling the boiler when it's already full, or boiling an empty boiler!

```
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;
    }
}
```

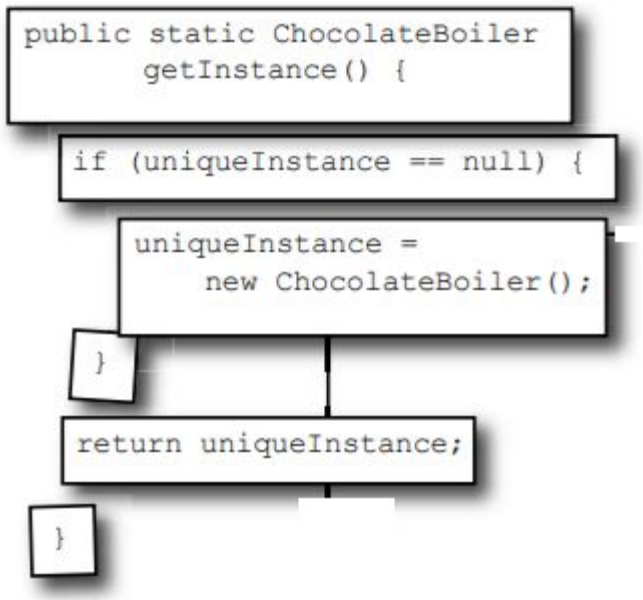
This code is only started when the boiler is empty!

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

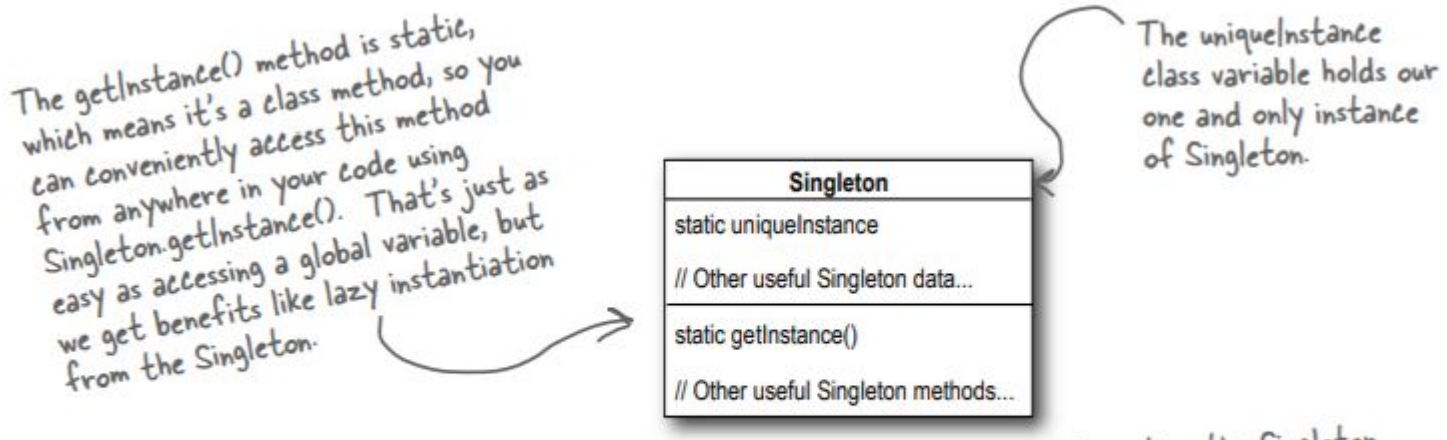
To drain the boiler, it must be full (non empty) and also boiled. Once it is drained we set empty back to true.

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.

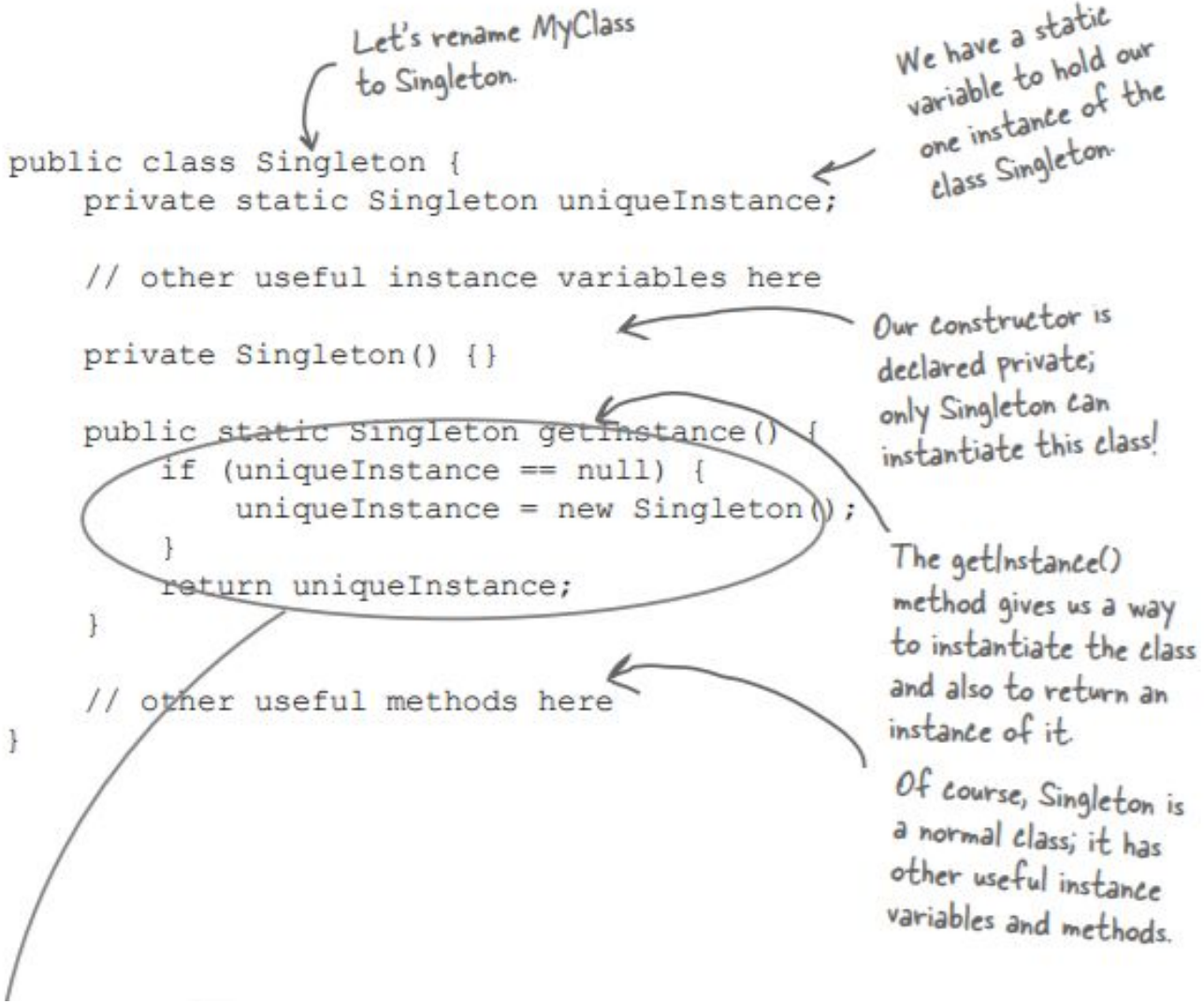
Using Singleton Pattern



Class Diagram



What if we have multiple threads? Problem?



Dealing with Multithreading Using a 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.

Use “Double-checked locking”

```
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;
    }
}
```

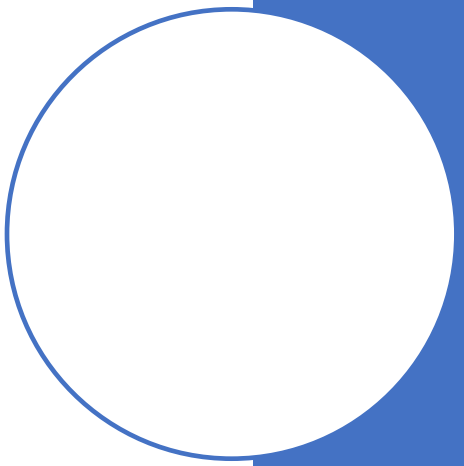
Check for an instance and if there isn't one, enter a synchronized block.

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.

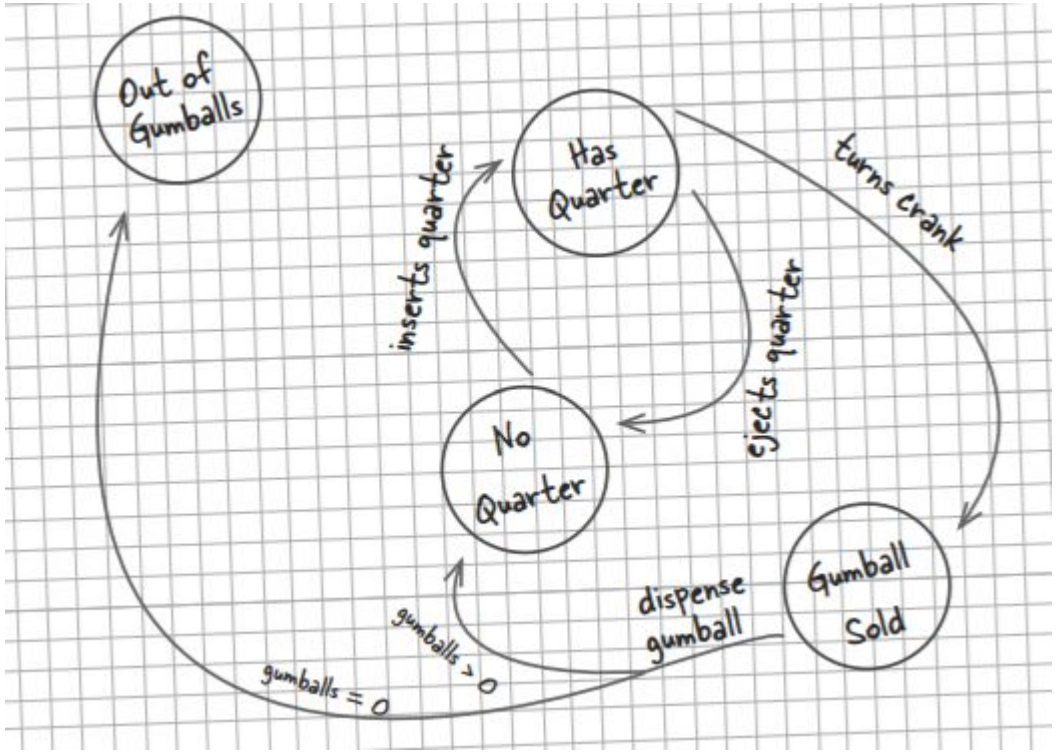
State Pattern



Gumball Machine

All states are just different configurations of the machine that behave in a certain way and need some action to take them to another state.

State, State Transitions?



Initial Thoughts (Common Implementation)

1. First, gather up your states:

2. Next, create an instance variable to hold the current state, and define values for each of the states:

Let's just call "Out of Gumballs" "Sold Out" for short.

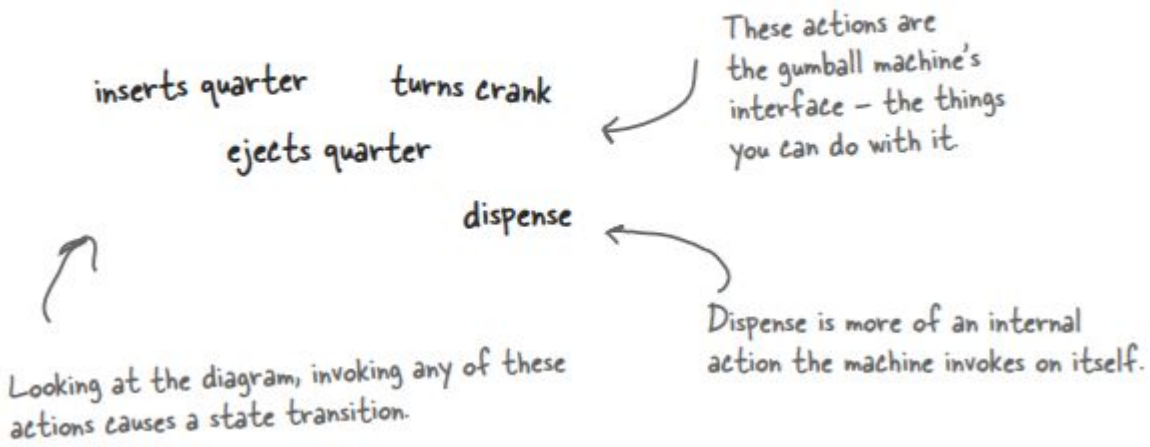
```
final static int SOLD_OUT = 0;
final static int NO_QUARTER = 1;
final static int HAS_QUARTER = 2;
final static int SOLD = 3;

int state = SOLD_OUT;
```

Here's each state represented as a unique integer...

...and here's an instance variable that holds the current state. We'll go ahead and set it to "Sold Out" since the machine will be unfilled when it's first taken out of its box and turned on.

Initial Thoughts (Common Implementation)

3 Now we gather up all the actions that can happen in the system:



Initial Thoughts (Common Implementation)

```
public void insertQuarter() {

    if (state == HAS_QUARTER) {

        System.out.println("You can't insert another quarter");

    } else if (state == SOLD_OUT) {

        System.out.println("You can't insert a quarter, the machine is sold out");

    } else if (state == SOLD) {

        System.out.println("Please wait, we're already giving you a gumball");

    } else if (state == NO_QUARTER) {

        state = HAS_QUARTER;
        System.out.println("You inserted a quarter");

    }
}
```

Each possible state is checked with a conditional statement...

...and exhibits the appropriate behavior for each possible state...

...but can also transition to other states, just as depicted in the diagram.

Similar to the above, we will be writing the implementation for remaining functions i.e., ejects quarter, turn crank, and dispense.

Problem?

What if we have 100 functions? Would it be easy to write if else conditions for each state in all those 100 functions?

Would this system be easy to maintain (incase a change request is made by the client)?

Solution: Let see how State Pattern solve this problem?

State Pattern

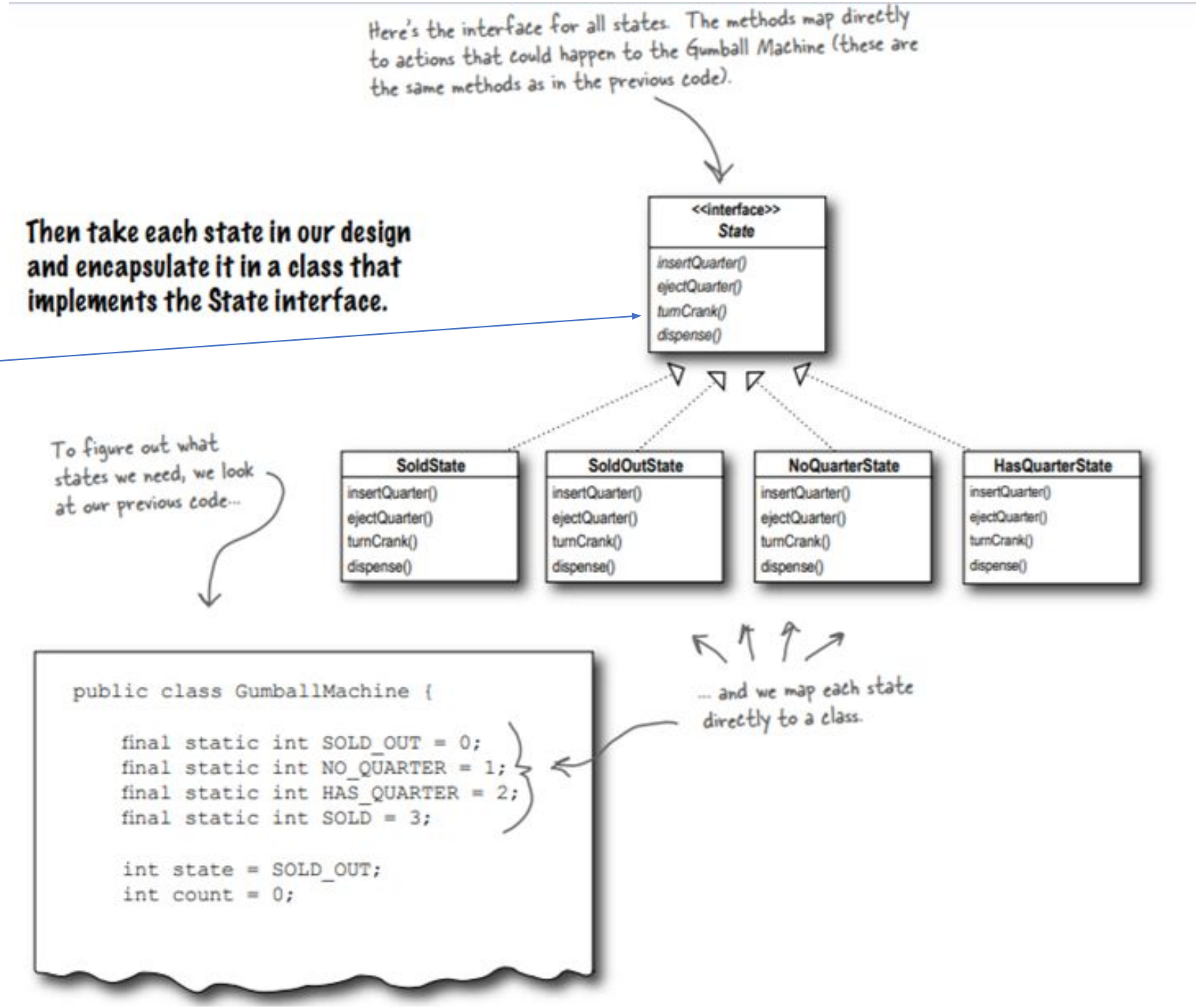
We’re going to encapsulate state objects in their own classes and then delegate to the current state when an action occurs.

1. **First, we’re going to define a State interface that contains a method for every action in the Gumball Machine.**
2. **Then we’re going to implement a State class for every state of the machine. These classes will be responsible for the behavior of the machine when it is in the corresponding state.**
3. **Finally, we’re going to get rid of all of our conditional code and instead delegate to the state class to do the work for us.**

GumballMachine

- SoldOutState: State
- SoldState: State
- NoQuarterState: State
- HasQuarterState: State

- state: State



Let focus on NoQuarterState Class

First we need to implement the State interface.

We get passed a reference to the Gumball Machine through the constructor. We're just going to stash this in an instance variable.

```
public class NoQuarterState implements State {
    GumballMachine gumballMachine;

    public NoQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You inserted a quarter");
        gumballMachine.setState(gumballMachine.getHasQuarterState());
    }

    public void ejectQuarter() {
        System.out.println("You haven't inserted a quarter");
    }

    public void turnCrank() {
        System.out.println("You turned, but there's no quarter");
     }

    public void dispense() {
        System.out.println("You need to pay first");
    }
}
```

If someone inserts a quarter, we print a message saying the quarter was accepted and then change the machine's state to the HasQuarterState.

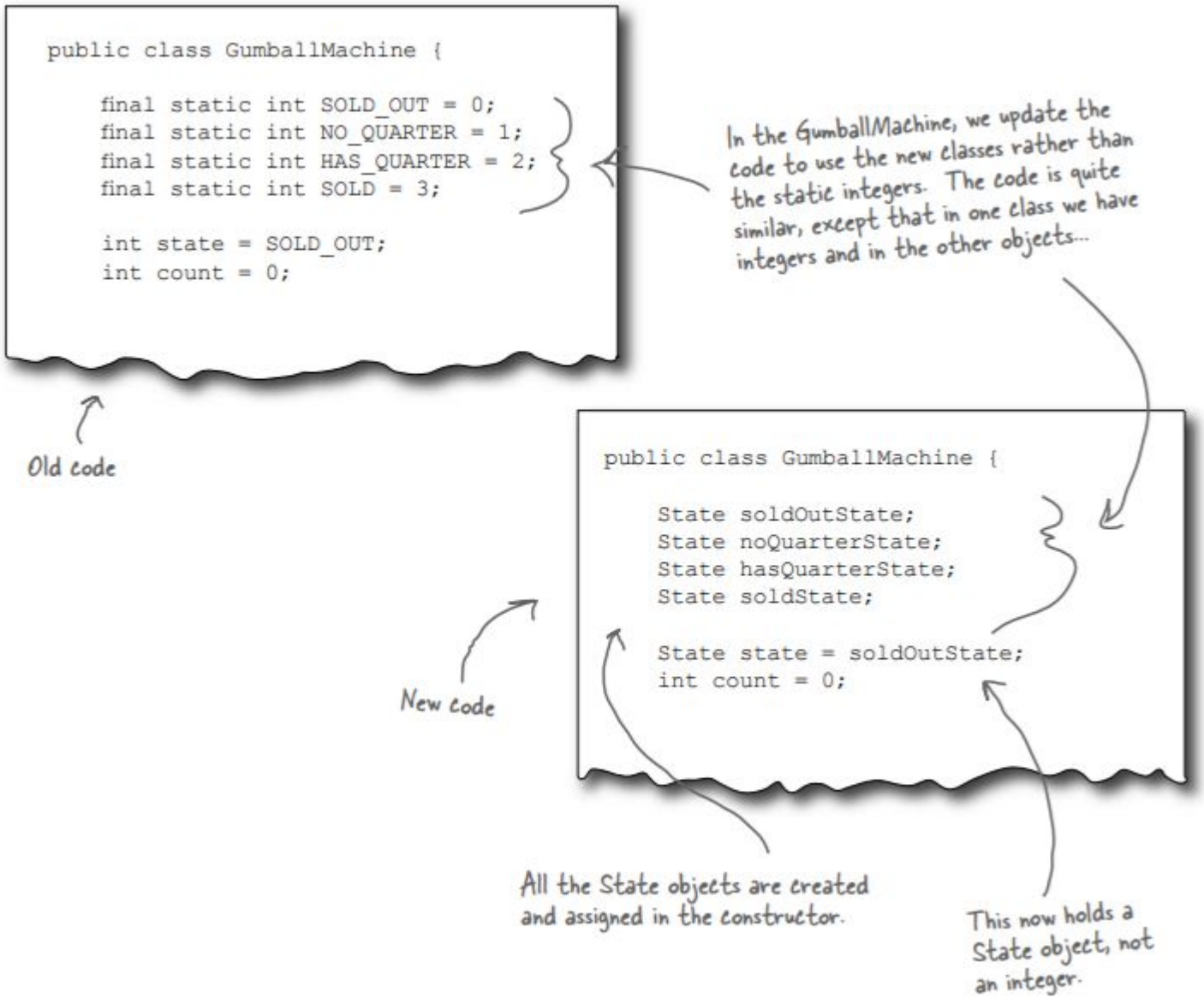
You'll see how these work in just a sec...

You can't get money back if you never gave it to us!

And, you can't get a gumball if you don't pay us.

We can't be dispensing gumballs without payment

Reworking the Gumball Machine



GumballMachine class Code

```
public class GumballMachine {

    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;

    State state = soldOutState;
    int count = 0;

    public GumballMachine(int numberGumballs) {
        soldOutState = new SoldOutState(this);
        noQuarterState = new NoQuarterState(this);
        hasQuarterState = new HasQuarterState(this);
        soldState = new SoldState(this);
        this.count = numberGumballs;
        if (numberGumballs > 0) {
            state = noQuarterState;
        }
    }

    public void insertQuarter() {
        state.insertQuarter();
    }

    public void ejectQuarter() {
        state.ejectQuarter();
    }

    public void turnCrank() {
        state.turnCrank();
        state.dispense();
    }

    void setState(State state) {
        this.state = state;
    }

    void releaseBall() {
        System.out.println("A gumball comes rolling out the slot...");
        if (count != 0) {
            count = count - 1;
        }
    }

    // More methods here including getters for each State...
}
```

Here are all the States again...

...and the State instance variable.

The count instance variable holds the count of gumballs – initially the machine is empty.

Our constructor takes the initial number of gumballs and stores it in an instance variable.

It also creates the State instances, one of each.

If there are more than 0 gumballs we set the state to the NoQuarterState.

Now for the actions. These are VERY EASY to implement now. We just delegate to the current state.

Note that we don't need an action method for dispense() in GumballMachine because it's just an internal action; a user can't ask the machine to dispense directly. But we do call dispense() on the State object from the turnCrank() method.

This method allows other objects (like our State objects) to transition the machine to a different state.

The machine supports a releaseBall() helper method that releases the ball and decrements the count instance variable.

Driver Class

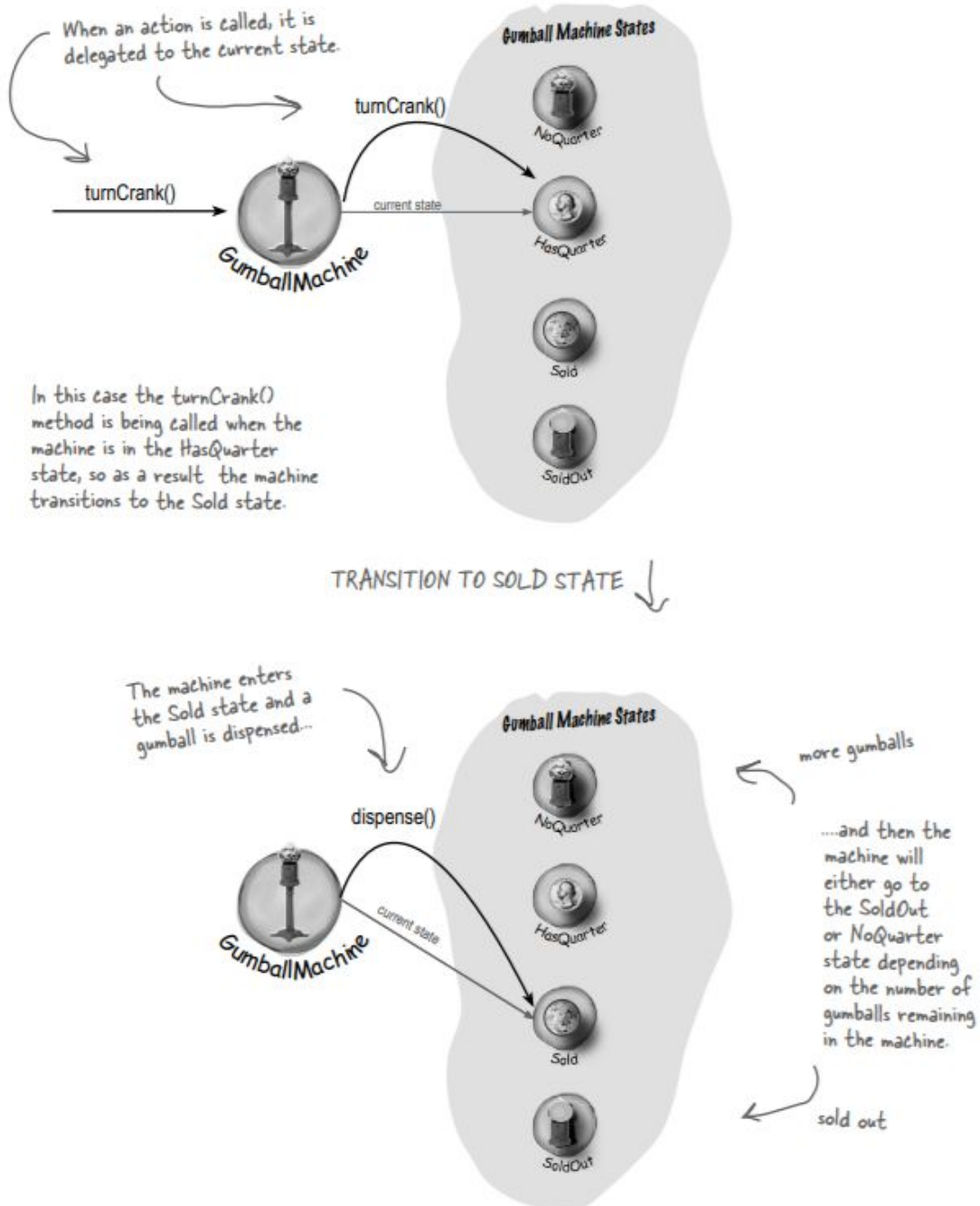
This code really hasn't changed at all; we just shortened it a bit.

Once, again, start with a gumball machine with 5 gumballs.

```
public class GumballMachineTestDrive {
    public static void main(String[] args) {
        GumballMachine gumballMachine = new GumballMachine(5);

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);
    }
}
```

We want to get a winning state, so we just keep pumping in those quarters and turning the crank. We print out the state of the gumball machine every so often...

Let’s visualize what we have done


When an action is called, it is delegated to the current state.
turnCrank()
turnCrank()
current state
GumballMachine
Gumball Machine States
NoQuarter
HasQuarter
Sold
SoldOut
In this case the turnCrank() method is being called when the machine is in the HasQuarter state, so as a result the machine transitions to the Sold state.
TRANSITION TO SOLD STATE
The machine enters the Sold state and a gumball is dispensed...
dispense()
current state
GumballMachine
Gumball Machine States
NoQuarter
HasQuarter
Sold
SoldOut
more gumballs
....and then the machine will either go to the SoldOut or NoQuarter state depending on the number of gumballs remaining in the machine.
sold out