Controller in MVC

Dr. Youna Jung

Northeastern University

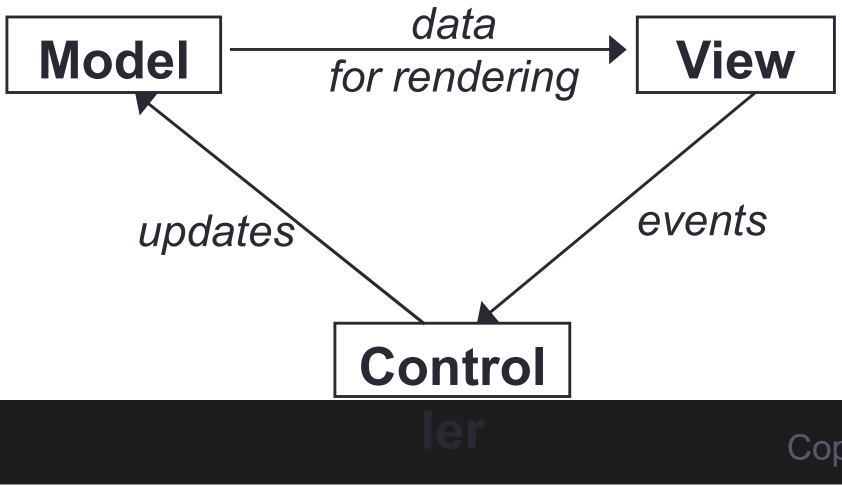
yo.jung@northeastern.edu

LVX
VERITAS
VIRTVS

MVC PATTERNS

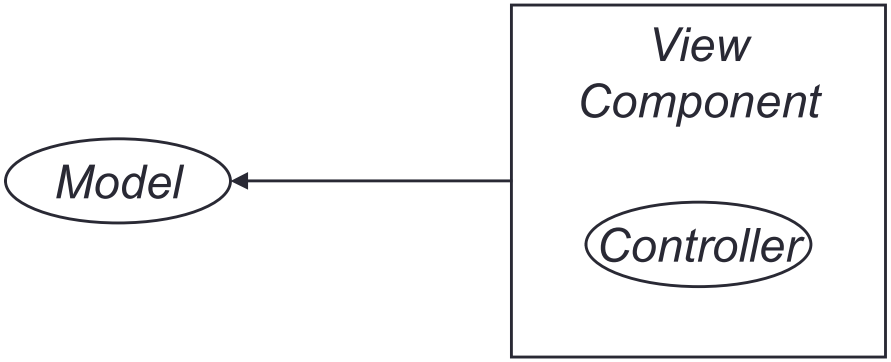
Model-View-Controller Pattern

- **Model**
  - ✓ **Classes** in your system that are related to the internal representation of **data** and **state** of the **system**
    - often part of the model is connected to **file**(s) or **database**(s)
    - Ex) **Card** game - **Card**, **Deck**, **Player**
    - EX) **Bank** system - **Account**, **User**, **UserList**
  - ✓ What it does
    - implements all the **functionality**
  - ✓ Does not do
    - does not care about **which functionality** is used **when**, **how results** are **shown** to the user



MVC Pattern

- ❑ **Controller**
  - ✓ Classes that connect **model** and **view**
    - defines how user interface reacts to user input (events)
    - receives messages from **view** (where events come from)
    - sends messages to **model** (tells what data to display)
  - ✓ What it does
    - **Takes** user inputs, tells **model** what to **do** and **view** what to **display**
  - ✓ Does not do
    - does **not care how model implements** functionality, screen layout to **display results**



MVC Pattern

- ❑ **View**
  - ✓ Classes in your system that **display** the state of the model to the user
    - generally, this is your **GUI** (could also be a **text UI**)
    - should not contain crucial application data
    - **Different views** can represent the same data in different ways
      - ➢Ex) Bar chart vs. pie chart
  - ✓ What it does
    - **display** results to user
  - ✓ Does not do
    - does not care **how** the **results** were **produced**, **when** to **respond** to user action

Advantages of MVC

- **Separating Model** (Data **Representation**) from **View** (Data **Presentation**)
  - ✓ easy to add **multiple data presentations** for the **same data**,
  - ✓ facilitates **adding new types** of data presentation as technology develops.
  - ✓ Model and View components can vary independently enhancing maintainability, extensibility, and testability.

- **Separating Controller** (Application **Behavior**) from **View** (Application **Presentation**)
  - ✓ permits **run-time selection** of appropriate **views**
    - – Views based on workflow, user preferences, or Model state.

Advantages of MVC

- **Separating Controller** (Application **Behavior**) from **Model** (Application **Representation**)
  - ✓ allows **configurable mapping** of **user actions** on the Controller to application **functions** on the Model.

CONTROLLER

MVC Pattern

Controller

- ❑ **Control how** and **when** the **model** and **view** are **used**
- ❑ **Handle** user and external **inputs** and **outputs** to/from the program
  - ✓ *How to handle diverse types of inputs and output?*

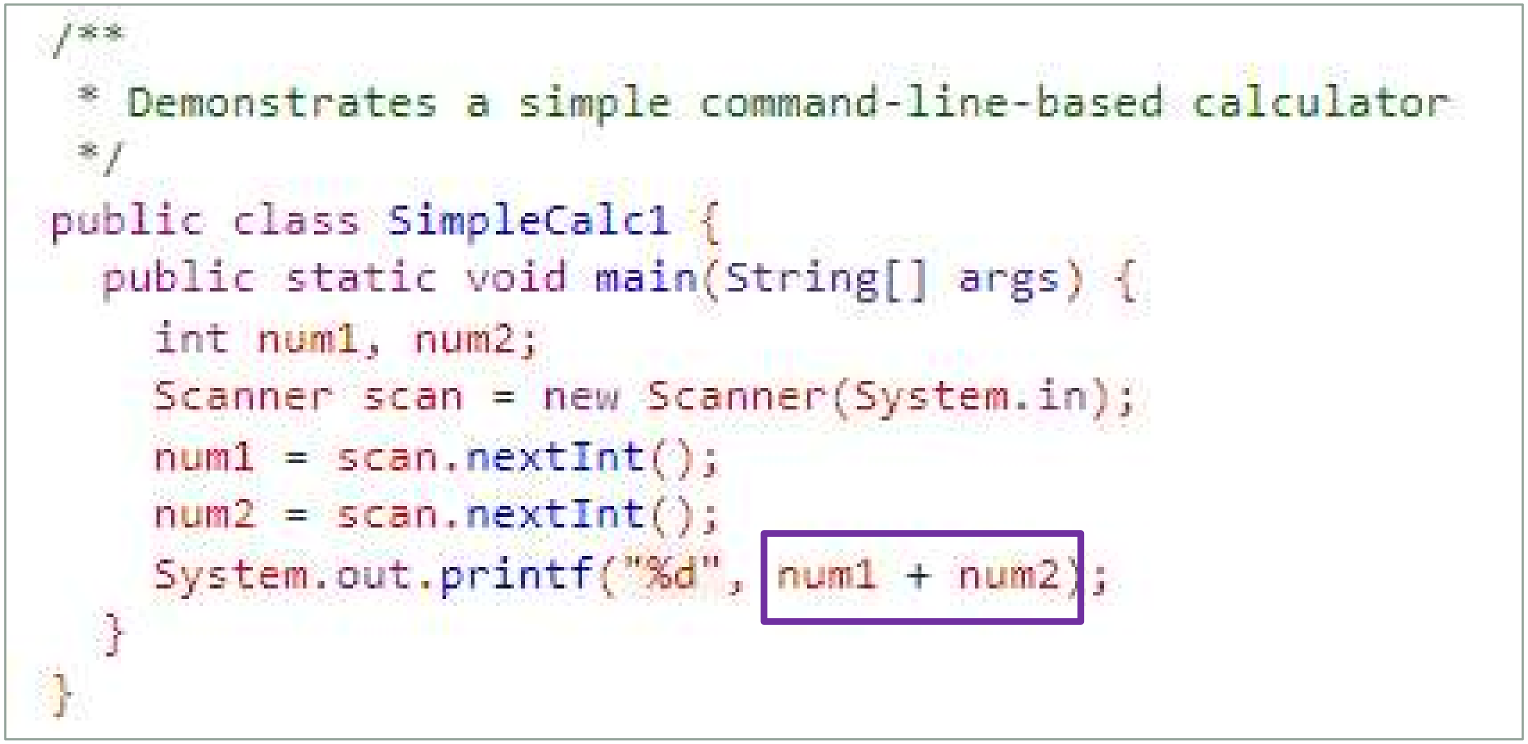
- ❑ **2 Type** of Controller
  - ✓ **Synchronous controller** ***Batch-processing*** *programs*
    - – implement such a **pre-defined sequence**
    - – Typically, **a method** that goes through this sequence in a loop
  - ✓ **Asynchronous controller** *Most of* ***GUI programs***
    - – Executes **depending** on user **input**
    - – usually divided into **several methods**
      - ➢ **Each** method is called in **response to** specific user **action**.

TTT Controller

1. Ask the **user** to **input** the next move
2. Tell the **model** to make the move as specified by the user
3. Get the current board from the **model**
4. Tell the **view** to show the current board
5. **IF** the game is over, go to **STEP 6**, **Else** go to **STEP 1**
6. Ask the **model** for the winner
7. Tell the **view** to show the current winner, **OR** a suitable **message** if there is no winner

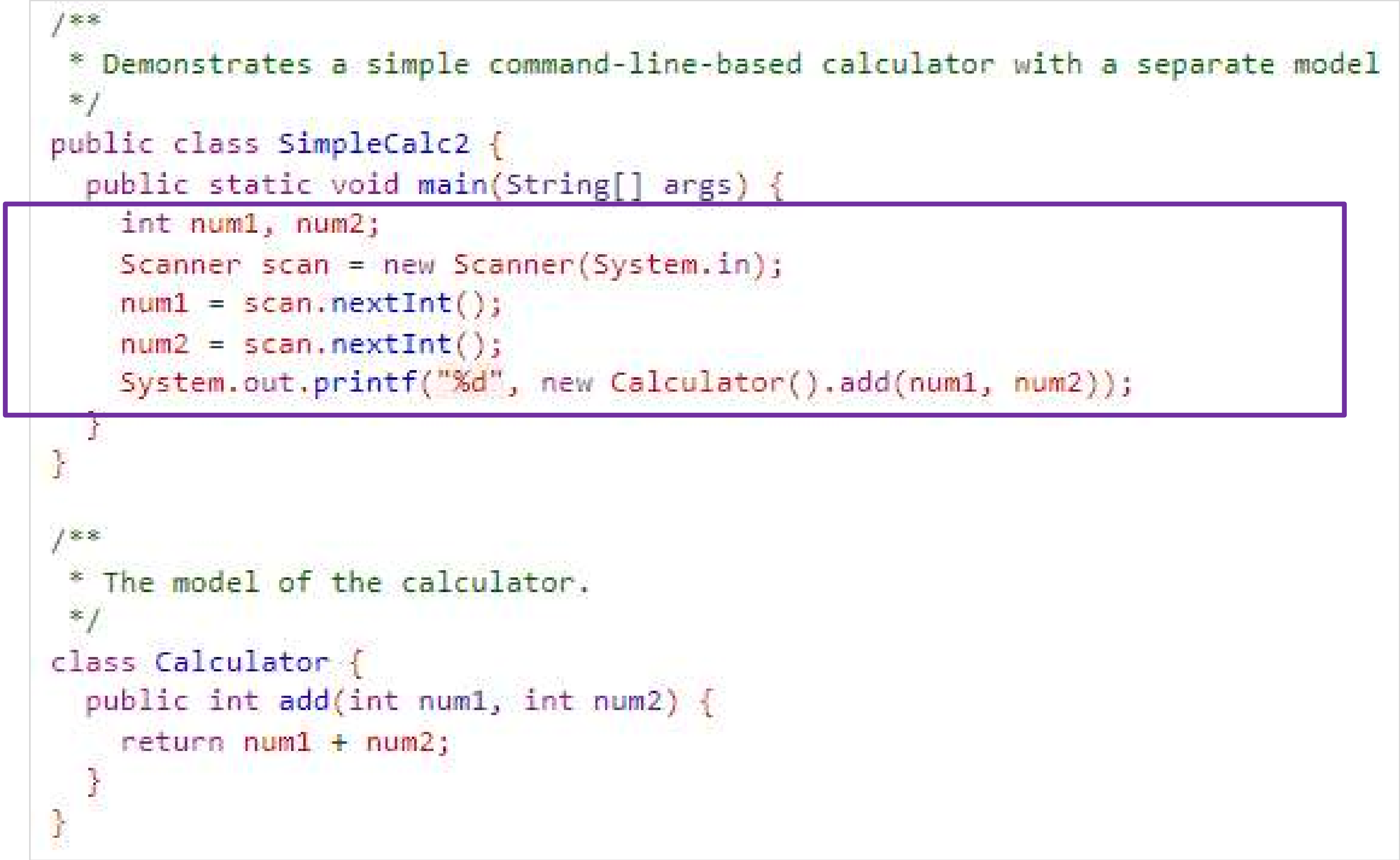
Example: Calculator

- ❑ A monolithic design, with `main()` doing all the work



```
/**
 * Demonstrates a simple command-line-based calculator
 */
public class SimpleCalc1 {
  public static void main(String[] args) {
    int num1, num2;
    Scanner scan = new Scanner(System.in);
    num1 = scan.nextInt();
    num2 = scan.nextInt();
    System.out.printf("%d", num1 + num2);
  }
}
```

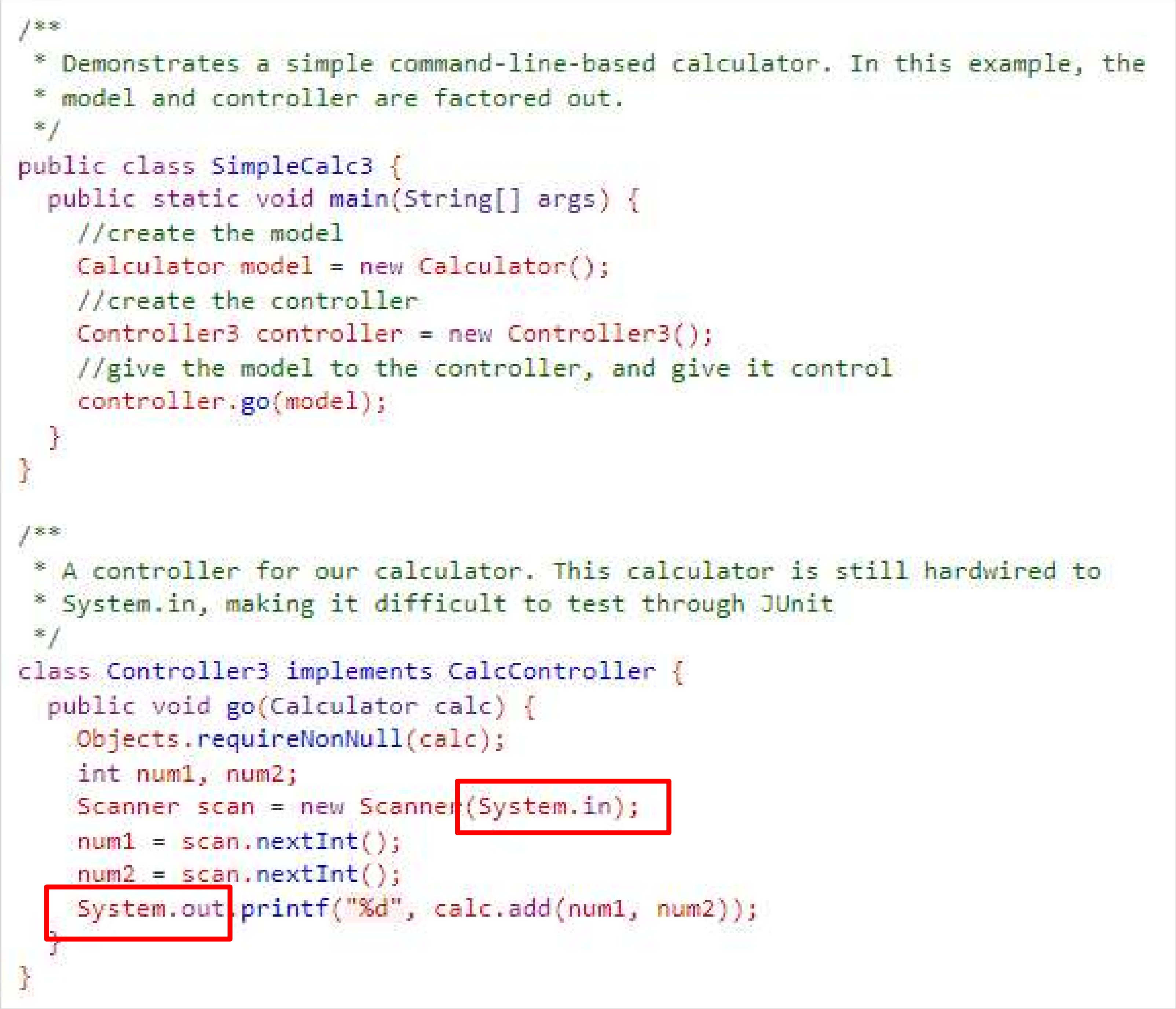
➔ **Factoring out the model**



```java
/**
 * Demonstrates a simple command-line-based calculator with a separate model
 */
public class SimpleCalc2 {
  public static void main(String[] args) {
    int num1, num2;
    Scanner scan = new Scanner(System.in);
    num1 = scan.nextInt();
    num2 = scan.nextInt();
    System.out.printf("%d", new Calculator().add(num1, num2));
  }
}

/**
 * The model of the calculator.
 */
class Calculator {
  public int add(int num1, int num2) {
    return num1 + num2;
  }
}
```

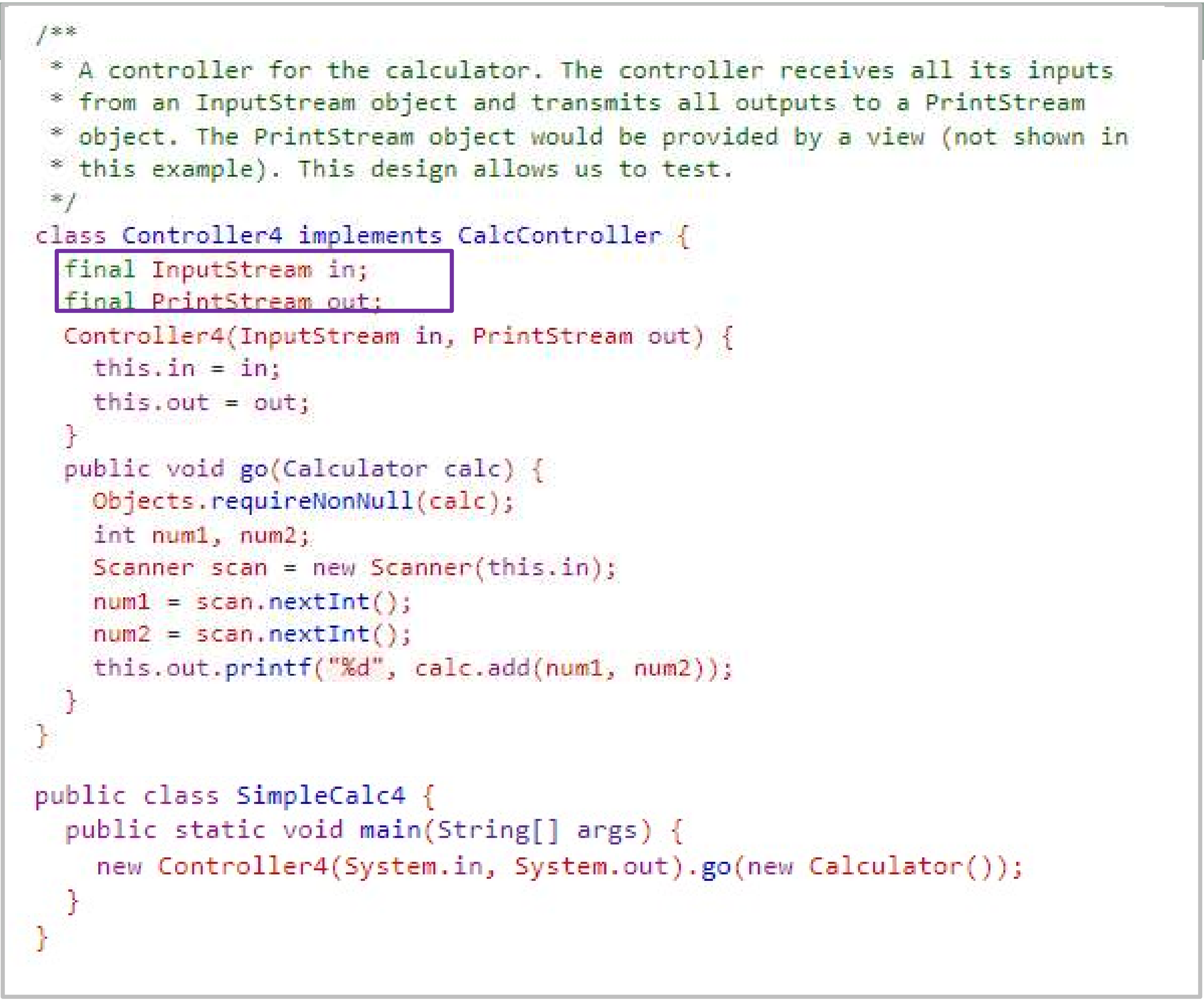
- Factoring out the **controller**



```
/**
 * Demonstrates a simple command-line-based calculator. In this example, the
 * model and controller are factored out.
 */
public class SimpleCalc3 {
  public static void main(String[] args) {
    //create the model
    Calculator model = new Calculator();
    //create the controller
    Controller3 controller = new Controller3();
    //give the model to the controller, and give it control
    controller.go(model);
  }
}

/**
 * A controller for our calculator. This calculator is still hardwired to
 * System.in, making it difficult to test through JUnit
 */
class Controller3 implements CalcController {
  public void go(Calculator calc) {
    Objects.requireNonNull(calc);
    int num1, num2;
    Scanner scan = new Scanner(System.in);
    num1 = scan.nextInt();
    num2 = scan.nextInt();
    System.out.printf("%d", calc.add(num1, num2));
  }
}
```

❑ Hardcoded with **`System.in`**(input) and **`System.out`** (output)

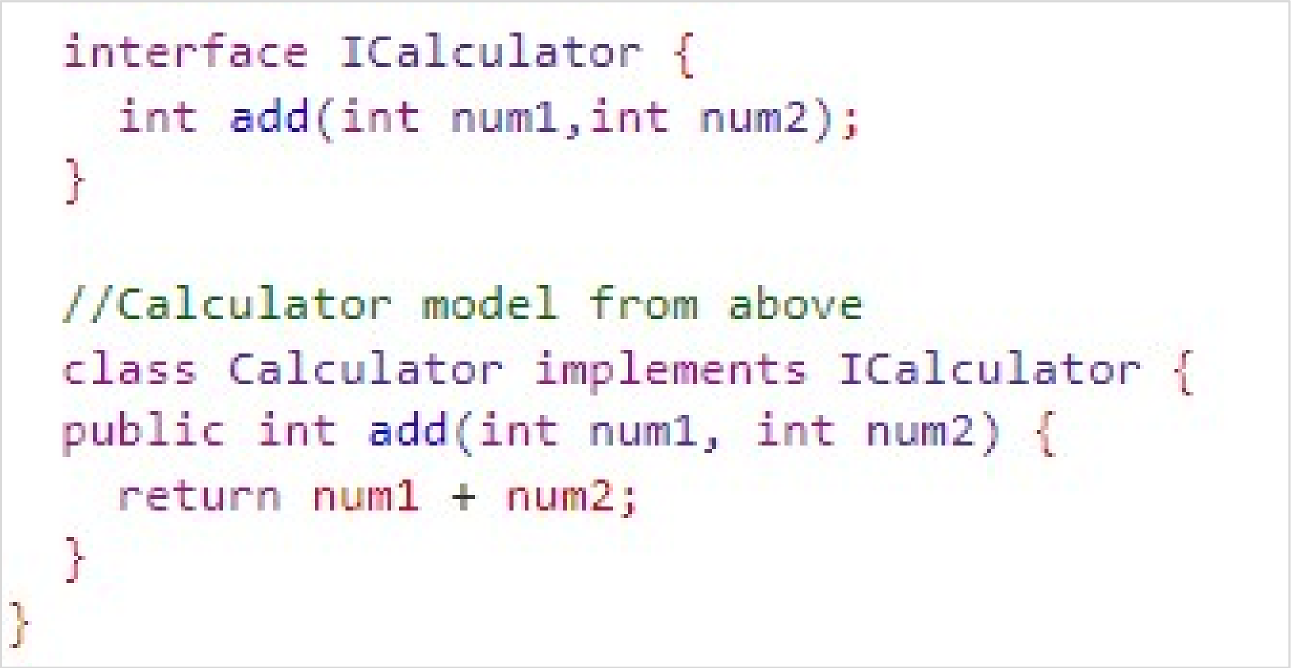


```
/**
 * A controller for the calculator. The controller receives all its inputs
 * from an InputStream object and transmits all outputs to a PrintStream
 * object. The PrintStream object would be provided by a view (not shown in
 * this example). This design allows us to test.
 */
class Controller4 implements CalcController {
  final InputStream in;
  final PrintStream out;
  Controller4(InputStream in, PrintStream out) {
    this.in = in;
    this.out = out;
  }
  public void go(Calculator calc) {
    Objects.requireNonNull(calc);
    int num1, num2;
    Scanner scan = new Scanner(this.in);
    num1 = scan.nextInt();
    num2 = scan.nextInt();
    this.out.printf("%d", calc.add(num1, num2));
  }
}

public class SimpleCalc4 {
  public static void main(String[] args) {
    new Controller4(System.in, System.out).go(new Calculator());
  }
}
```

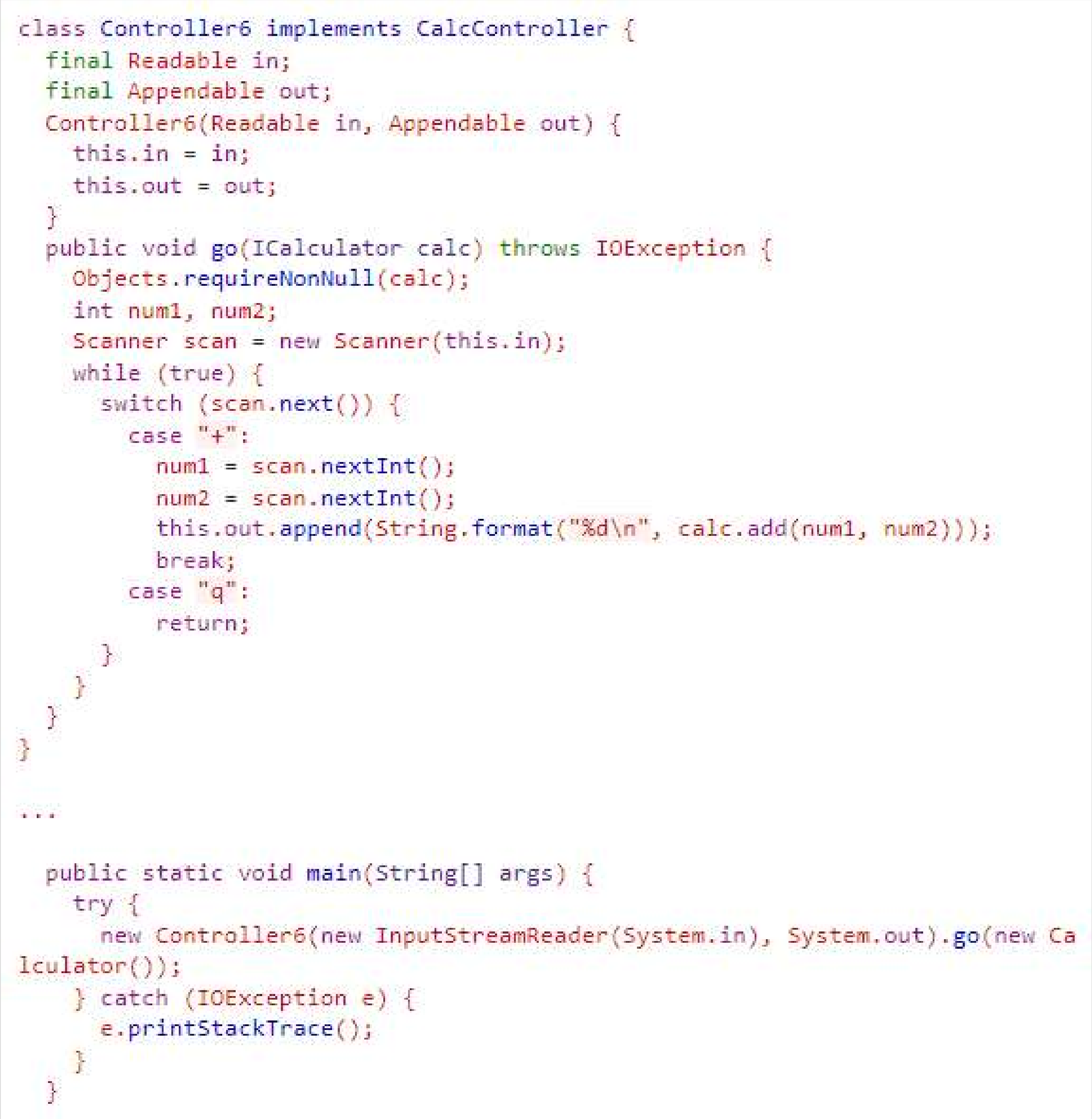
Testing the controller in isolation

- ❑ To **isolate** the controller, need a **mock** model
  - ✓ looks like the real one but is simpler
  - ✓ → Create an explicit interface for a mock model → Make our mock model implement this interface



```
 interface ICalculator {
  int add(int num1,int num2);
 }

 //Calculator model from above
 class Calculator implements ICalculator {
 public int add(int num1, int num2) {
  return num1 + num2;
 }
}
```

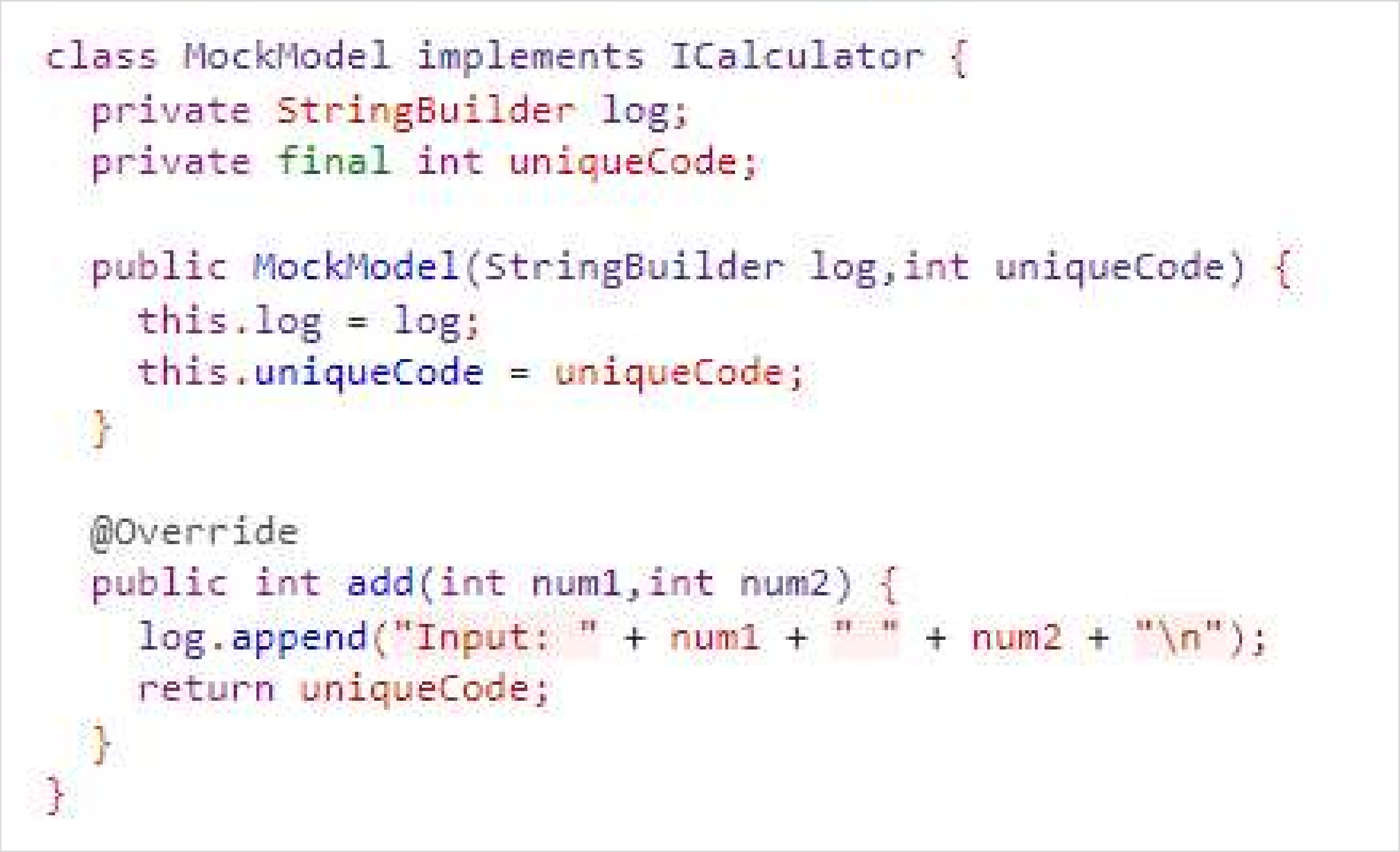


```
class Controller6 implements CalcController {
  final Readable in;
  final Appendable out;
  Controller6(Readable in, Appendable out) {
    this.in = in;
    this.out = out;
  }
  public void go(ICalculator calc) throws IOException {
    Objects.requireNonNull(calc);
    int num1, num2;
    Scanner scan = new Scanner(this.in);
    while (true) {
      switch (scan.next()) {
        case "+":
          num1 = scan.nextInt();
          num2 = scan.nextInt();
          this.out.append(String.format("%d\n", calc.add(num1, num2)));
          break;
        case "q":
          return;
      }
    }
  }
}

...

  public static void main(String[] args) {
    try {
      new Controller6(new InputStreamReader(System.in), System.out).go(new Ca
lculator());
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
```

Mock model for Isolation

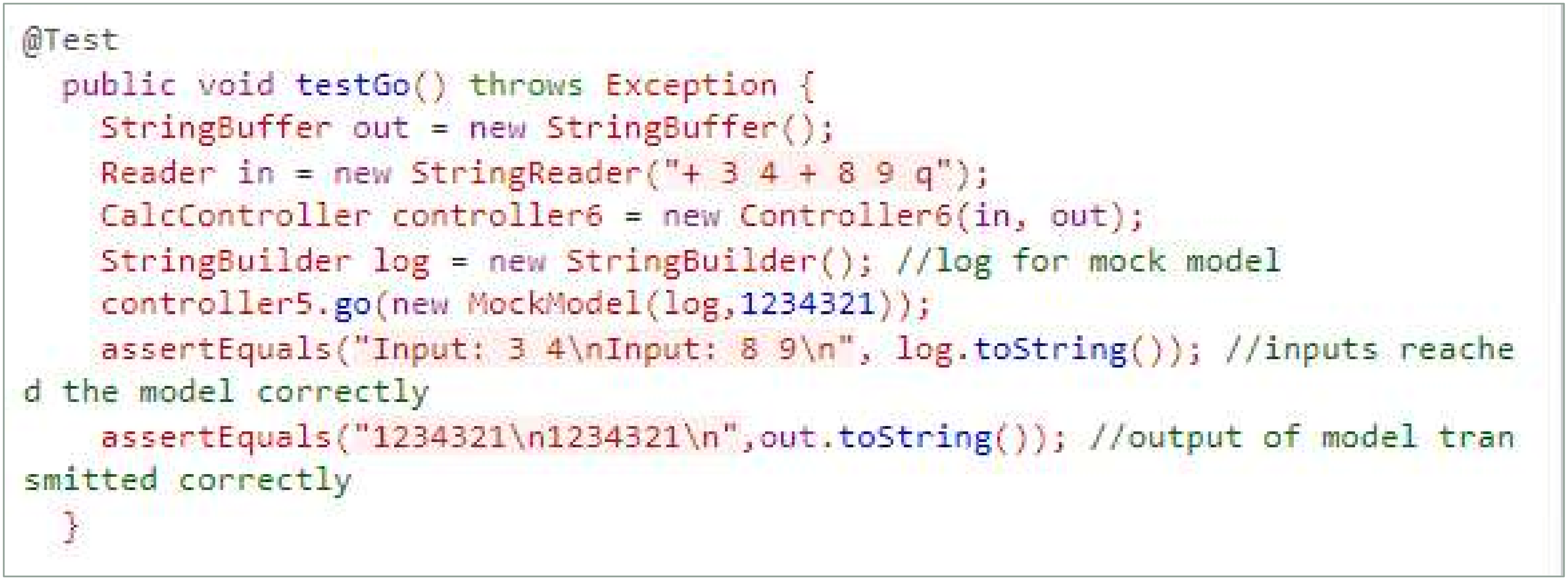


```
class MockModel implements ICalculator {
  private StringBuilder log;
  private final int uniqueCode;

  public MockModel(StringBuilder log,int uniqueCode) {
    this.log = log;
    this.uniqueCode = uniqueCode;
  }

  @Override
  public int add(int num1,int num2) {
    log.append("Input: " + num1 + " " + num2 + "\n");
    return uniqueCode;
  }
}
```

- ❑ This mock model not actually add numbers: it merely **logs** the **inputs** provided to it, and returns a unique number provided to it at creation



```
@Test
  public void testGo() throws Exception {
    StringBuffer out = new StringBuffer();
    Reader in = new StringReader("+ 3 4 + 8 9 q");
    CalcController controller6 = new Controller6(in, out);
    StringBuilder log = new StringBuilder(); //log for mock model
    controller5.go(new MockModel(log,1234321));
    assertEquals("Input: 3 4\nInput: 8 9\n", log.toString()); //inputs reache
d the model correctly
    assertEquals("1234321\n1234321\n",out.toString()); //output of model tran
smitted correctly
  }
```

❑It tests whether

- the inputs provided to the controller were correctly transmitted to the model
- the results from the model were correctly transmitted to the Appendable object by the controller
- It does not test whether the controller-model combination produced the correct answer.