

Topic 3: Organising Your Code

Programming Practice and Applications (4CCS1PPA)

Dr. Martin Chapman
Thursday 6th October

programming@kcl.ac.uk
martinchapman.co.uk/teaching

To understand how to organise code

- **In the same file**
- **Into different files**

REMEMBER: CLASSES AND FILES

Each file can only contain **one** public class.

The name of this class **must match the name of the file** in which the class resides.

Therefore, at this stage, we will consider classes to be **roughly** equivalent to files.

- One file for each class.
- Later on in the course, we'll see that this isn't strictly true.

Organising Your Code Within The Same File

REMEMBER: A SLIGHTLY MORE COMPLEX PROGRAM

```
public class MartinPrinter {  
  
    public static void main(String[] args) {  
  
        System.out.println("+-----+");  
        System.out.println("|Martin|");  
        System.out.println("+-----+");  
  
    }  
  
}
```

MartinPrinter.java

When we have more than one line in a program, the program is executed **line-by-line**.

CODE REPETITION (1)

Being self-centred, I want to print my name **three times** to the terminal.

Cool.

```
public class MartinPrinter {  
  
    public static void main(String[] args) {  
  
        System.out.println("+-----+");  
        System.out.println("|Martin|");  
        System.out.println("+-----+");  
  
        System.out.println("+-----+");  
        System.out.println("|Martin|");  
        System.out.println("+-----+");  
  
        System.out.println("+-----+");  
        System.out.println("|Martin|");  
        System.out.println("+-----+");  
  
    }  
}
```

MartinPrinter.java

CODE REPETITION (2)

Being **very** self-centred, I want to print my name **fifty times** to the terminal.

What. This will take **time**, and I'm likely to make **mistakes**.

[illegible]

LABELLING CODE (1)

It would be useful if we could somehow **label** our print statement code, it **wouldn't execute** until we **specifically said to**, and we could **call it as many times as we like** (3 times, for example).

`Print Martin':

```
System.out.println("+-----+");  
System.out.println("|Martin|");  
System.out.println("+-----+");
```

Go to `Print Martin'

Go to `Print Martin'

Go to `Print Martin'

*We would still have to reference the label (later, we'll see how we can address this issue), but the reference is simpler, it would reduce the lines we need by a third, and we would only have to write the print code **once**.*

These steps are not yet in a language that the computer can understand.

We're **expressing** or **designing** the solution we want using written English.

We call this **pseudocode**.

There is no particular **standard pseudocode syntax** (although efforts have been made to construct a standard language), but **personal consistency** is important.

You will be expected to write pseudocode for your assignment documentation (see KEATS).

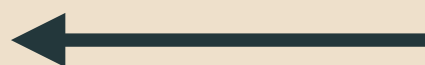
EXECUTION ORDER

If we were to be able to do the below, we would **no longer** be executing our code **line-by-line**. We would start at the arrow, and then:

`Print Martin':

```
System.out.println("+-----+");  
System.out.println("|Martin|");  
System.out.println("+-----+");
```

Go to `Print Martin'



Go to `Print Martin'

Go to `Print Martin'

*The arrow returns to the next `go to' statement after each Martin is printed because the JVM must execute all of our **requested** lines of code.*



METHODS

Fortunately, we can do just this in Java by creating something called a **method**.

```
public class MartinPrinter {
```

```
    public static void printMartin() {
```

```
        System.out.println("+-----+");
```

```
        System.out.println("|Martin|");
```

```
        System.out.println("+-----+");
```

```
    }
```

```
    public static void main(String[] args) {
```

```
        printMartin();
```

```
        printMartin();
```

```
        printMartin();
```

```
    }
```

```
}
```

*This is now one of our **own** methods and we can have as **many** as we like.*

*We've already seen a **main** method.*

MartinPrinter.java

METHODS: AN EXAMPLE (1) (LABELS)

Methods provide us with the labels we saw in the previous example in the form of **named sections of code**.

```
public class MartinPrinter {  
  
    `Print Martin':    public static void printMartin() {  
  
        System.out.println("+-----+");  
        System.out.println("|Martin|");  
        System.out.println("+-----+");  
  
    }  
  
    public static void main(String[] args) {  
  
        printMartin();  
        printMartin();  
        printMartin();  
  
    }  
  
}
```

Go to `Print Martin'
Go to `Print Martin'
Go to `Print Martin'

*The methods in a class can be called by any **other** method in that class (usually).*

MartinPrinter.java

METHODS: AN EXAMPLE (1) (BLOCKED OFF CODE)

We block off our code accordingly, so it is clear **where** these sections are.

*Everything is still within the **same class** (the same **file**).*

```
public class MartinPrinter {  
    public static void printMartin() {  
        System.out.println("+-----+");  
        System.out.println("|Martin|");  
        System.out.println("+-----+");  
    }  
    public static void main(String[] args) {  
        printMartin();  
        printMartin();  
        printMartin();  
    }  
}
```

*Note the merit of **indentation** once multiple methods are introduced.*

MartinPrinter.java



METHODS: AN EXAMPLE (1) (CALLING A METHOD)

When we want to **run the code within a method**, we write the name of the method (including the brackets) and Java **matches** the correct place for **execution** to **move** to.

The **location** of the target method in the class (before or after) is **unimportant**.

We'll discuss the function of the **brackets** after the name of the method shortly.

```
public class MartinPrinter {  
    public static void printMartin() {  
        System.out.println("+-----+");  
        System.out.println("|Martin|");  
        System.out.println("+-----+");  
    }  
  
    public static void main(String[] args) {  
        printMartin();  
        printMartin();  
        printMartin();  
    }  
}
```

Unlike class names, which are typically nouns, methods names are typically **verbs** because they **do something**.

Like matching method names, much of Java's **control structure** is about matching labels.

As such, **for now**, we will implement methods with **unique names** to avoid match **conflicts**.

MartinPrinter.java



METHODS: AN EXAMPLE (1) (EXECUTION ORDER)

Thus, the program runs as seen previously:

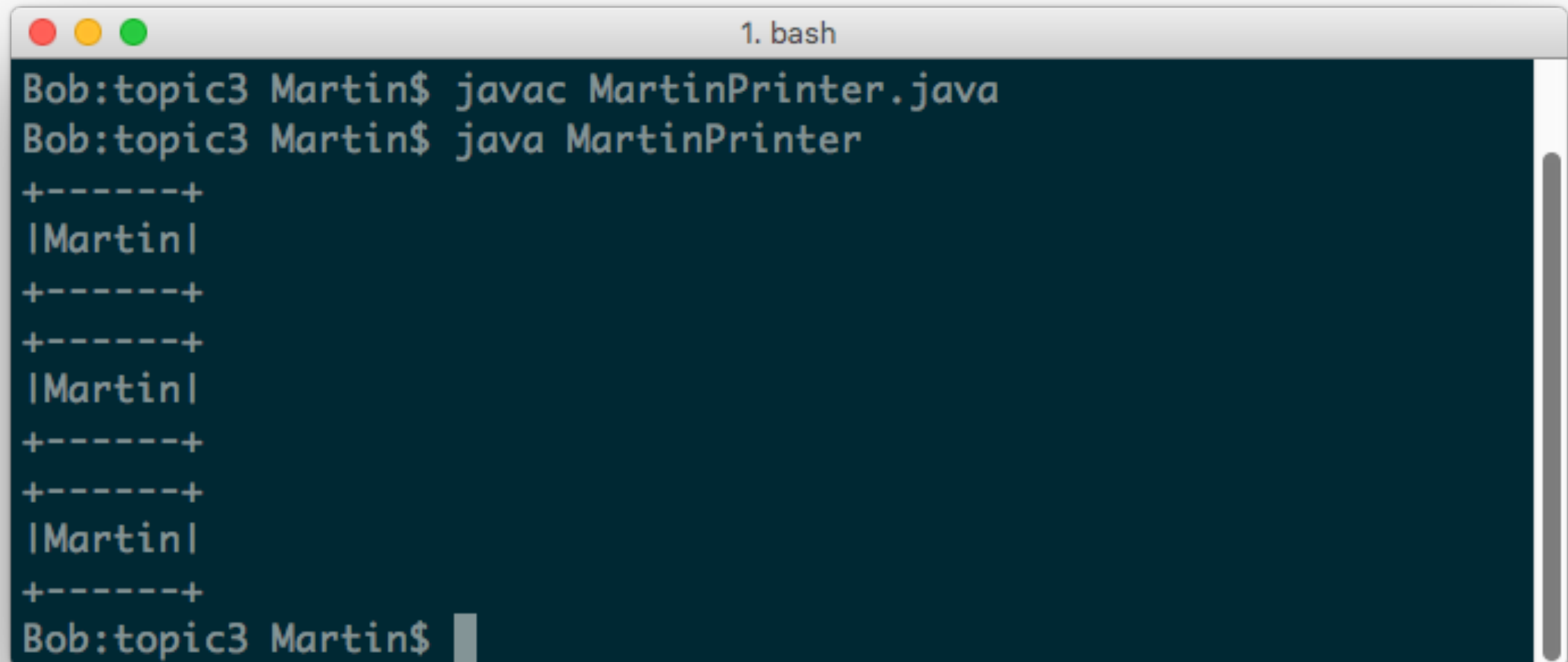
```
public class MartinPrinter {  
  
    public static void printMartin() {  
  
        System.out.println("+-----+");  
        System.out.println("|Martin|");  
        System.out.println("+-----+");  
  
    }  
  
    public static void main(String[] args) {  
  
        printMartin();  
        printMartin();  
        printMartin();  
  
    }  
  
}
```

MartinPrinter.java

Even though the JVM is asked to move to a different place in the program to execute code, it cannot just **ignore** other statements that have been entered into our main method. Thus, it **moves back** to execute them.



METHODS: AN EXAMPLE (1) (OUTPUT)



```
1. bash
Bob:topic3 Martin$ javac MartinPrinter.java
Bob:topic3 Martin$ java MartinPrinter
+-----+
|Martin|
+-----+
+-----+
|Martin|
+-----+
+-----+
|Martin|
+-----+
Bob:topic3 Martin$
```

A terminal window titled "1. bash" with a dark blue background and light gray text. It shows the compilation and execution of a Java program. The prompt is "Bob:topic3 Martin\$". The first command is "javac MartinPrinter.java". The second command is "java MartinPrinter". The output consists of three lines, each showing a box with the name "Martin" inside. Each box is formed by a top and bottom row of dashes and vertical bars on the sides. The prompt "Bob:topic3 Martin\$" is shown at the bottom with a cursor.

METHODS AS A TOOL FOR CODE ORGANISATION (1)

It's true that by labelling our code, we are able to facilitate the alteration of the **execution order** of our program.

- Currently this allows us to call the **same** code **multiple times** within the same class.
- I always aim to show you what certain syntax gives us in **practice**.

But also, more abstractly, when we label our code, we split our code into **distinct** sections.

- This allows us to **separate functionality** in our class.
- Each method should perform a **specific task**.
- Thus, methods may **only be used once**.

We will see examples of this when we look at larger pieces of code.



LECTURE EXERCISE: LARGE PRIMES (1)

Let's write a class called `LargePrimes`, which contains a method `subtractPrimes` that prints the result of the second millionth prime number (32452843) minus the first millionth prime number (15485863).

Let's call our method twice.

The syntax for subtraction is available in Topic 2.

LECTURE EXERCISE: LARGE PRIMES (2)

```
public class LargePrimes {  
    public static void subtractPrimes() {  
        System.out.println(32452843 - 15485863);  
    }  
  
    public static void main(String[] args) {  
        subtractPrimes();  
        subtractPrimes();  
    }  
}
```

METHODS: AN EXAMPLE (1) (PUBLIC AND STATIC)

We still aren't really in a position to talk about **public** or **static**.

```
public class MartinPrinter {  
    public static void printMartin() {  
        System.out.println("+-----+");  
        System.out.println("|Martin|");  
        System.out.println("+-----+");  
    }  
    public static void main(String[] args) {  
        printMartin();  
        printMartin();  
        printMartin();  
    }  
}
```

*But, for now, it's important to know that **any method called from a static method must also be static.***

MartinPrinter.java



METHODS: AN EXAMPLE (1) (RETURN TYPES)

But we can talk about **void**.

```
public class MartinPrinter {  
    public static void printMartin() {  
        System.out.println("+-----+");  
        System.out.println("|Martin|");  
        System.out.println("+-----+");  
    }  
  
    public static void main(String[] args) {  
        printMartin();  
        printMartin();  
        printMartin();  
    }  
}
```

MartinPrinter.java

METHOD RETURN VALUES (SUBSTITUTION) (1)

I mentioned earlier that much of Java's control structure works with **matching labels**.

When something can be **matched** it can also be **substituted** (if possible).

- e.g. **$Y = X, X = 7$**
- We can match X , substitute it for 7, and end up with **$Y = 7$** .
- Let's code this...

METHOD RETURN VALUES (SUBSTITUTION) (2)

Y = X, X = 7 so Y = 7

```
public class Substitutor {  
    public static void X() {  
        return 7;        ?  
    }  
  
    public static void main(String[] args) {  
        int y = X();  
        System.out.println(y);    ?  
    }  
}
```

(This will **not** compile)



METHOD RETURN VALUES (THE RETURN KEYWORD)

When we use the keyword **return** in a method, any **matched references** to that method will be **substituted by the value given after the return statement.**

The method **becomes** the value it returns.

```
public class Substitutor {  
    public static void X() {  
        return 7;  
    }  
  
    public static void main(String[] args) {  
        int y = 7;  
        System.out.println(y);  
    }  
}
```

*Return almost always has to be the **last** action in a method.*

*(This will **not** compile)*



METHOD RETURN VALUES (RETURN TYPES) (1)

We noted previously that Java **forces** us to **declare** the type of variables.

```
int myFirstIntegerVariable
```

This allows **assignment** errors to be caught at **compile time**, and increases the **readability** of our code.

The same is true of methods: we must specify which **type** of value will be substituted if we call that method.

- This allows Java to check that we are doing sensible things with the returned values, like not assigning them to **unreasonable** types.
- From the previous example:

```
return 7.0;
```

```
int y = 7.0;
```

METHOD RETURN VALUES (RETURN TYPES) (2)

So in reality, this code will not work if we continue to use **void**.

Void means **we do not intend to return anything**, which is clearly not true in this case.

```
public class Substitutor {  
    public static void X() {  
        return 7;  
    }  
  
    public static void main(String[] args) {  
        int y = X();  
        System.out.println(y);  
    }  
}
```

(This will **not** compile)

METHOD RETURN VALUES (RETURN TYPES) (3)

Instead we need to change void to the **type of data we intend to return**.

In this case **int**.

```
public class Substitutor {  
    public static int X() {  
        return 7;  
    }  
  
    public static void main(String[] args) {  
        int y = X();  
        System.out.println(y);  
    }  
}
```

(This **will** compile; Substitutor.java)



It might be useful to think about the return value following a **path** back to the place in which it was called.

```
public class Substitutor {  
    public static int X() {  
        return 7;  
    }  
  
    public static void main(String[] args) {  
        int y = X();  
        System.out.println(y);  
    }  
}
```

Substitutor.java



SUBSTITUTION DOESN'T ALWAYS OCCUR

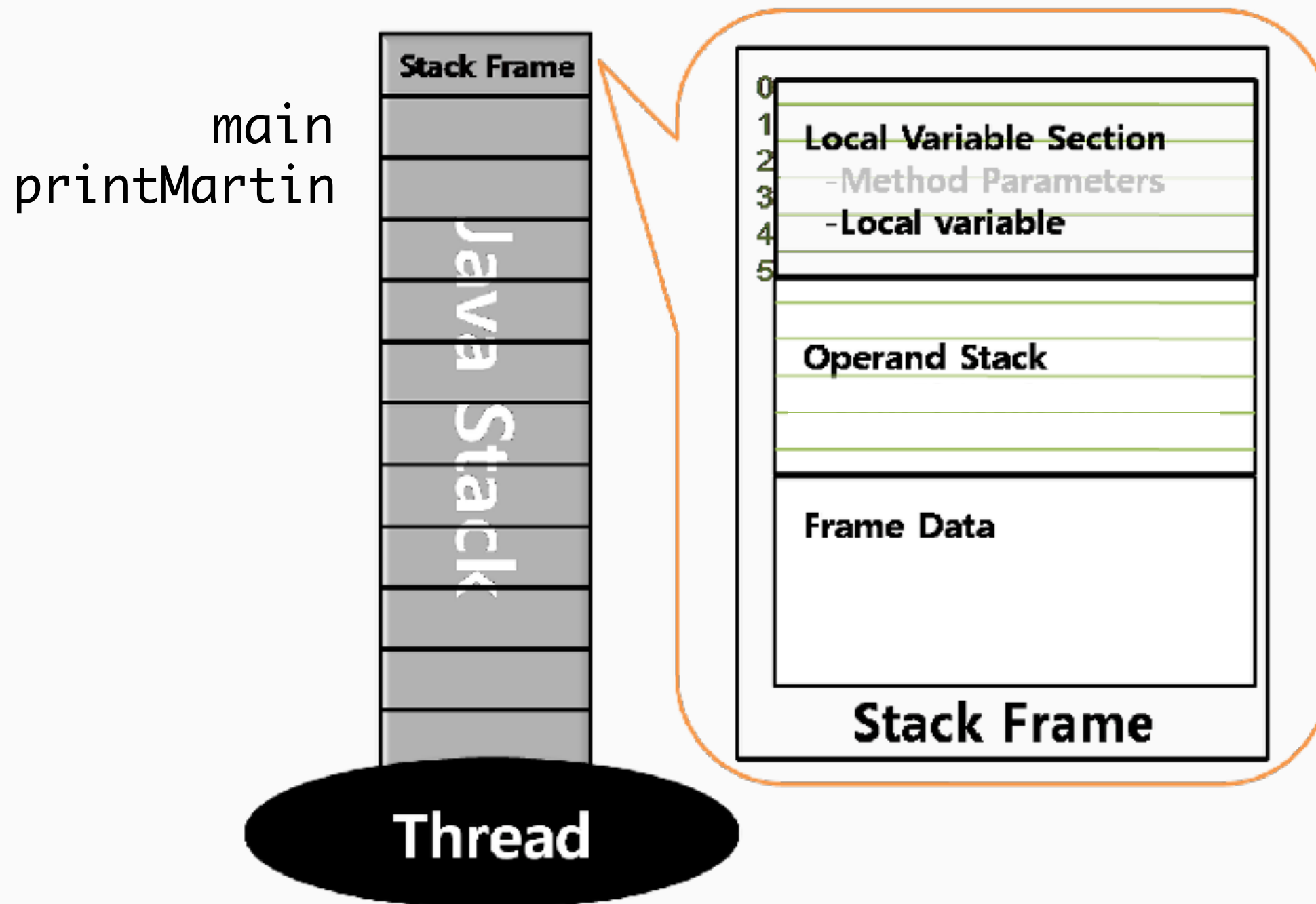
Back in MartinPrinter however, it is perfectly acceptable to write void as we do **not** have a return statement.

```
public class MartinPrinter {  
  
    public static void printMartin() {  
  
        System.out.println("+-----+");  
        System.out.println("|Martin|");  
        System.out.println("+-----+");  
  
    }  
  
    public static void main(String[] args) {  
  
        printMartin();  
        printMartin();  
        printMartin();  
  
    }  
  
}
```

Moreover, the method's function does not necessitate a return value.

Thus, there is no real substitution in this instance.

ASIDE: WHAT HAPPENS WHEN YOU CALL A METHOD



All the storage elements pertinent to a method are pushed onto an abstract **stack** structure, and then popped off once the method finishes, and we return to where we started.



LECTURE EXERCISE: LARGE PRIMES (3)

Let's alter the operation of `subtractPrimes` so that it instead returns the result of subtracting two prime numbers, rather than simply printing the result.

We can then print inside the main method (twice).

LECTURE EXERCISE: LARGE PRIMES (4)

```
public class LargePrimes {  
    public static int subtractPrimes() {  
        return 67867967 - 49979687;  
    }  
  
    public static void main(String[] args) {  
        System.out.println(subtractPrimes());  
        System.out.println(subtractPrimes());  
    }  
}
```


WE'RE STARTING TO BUILD UP A TEMPLATE FOR METHODS

Methods, so far, have to be public and static.

We specify the return type

We give our method a name

```
public static int x() {  
    return 7;  
}
```

What about these brackets?



So far we have considered what methods **output (return)**.

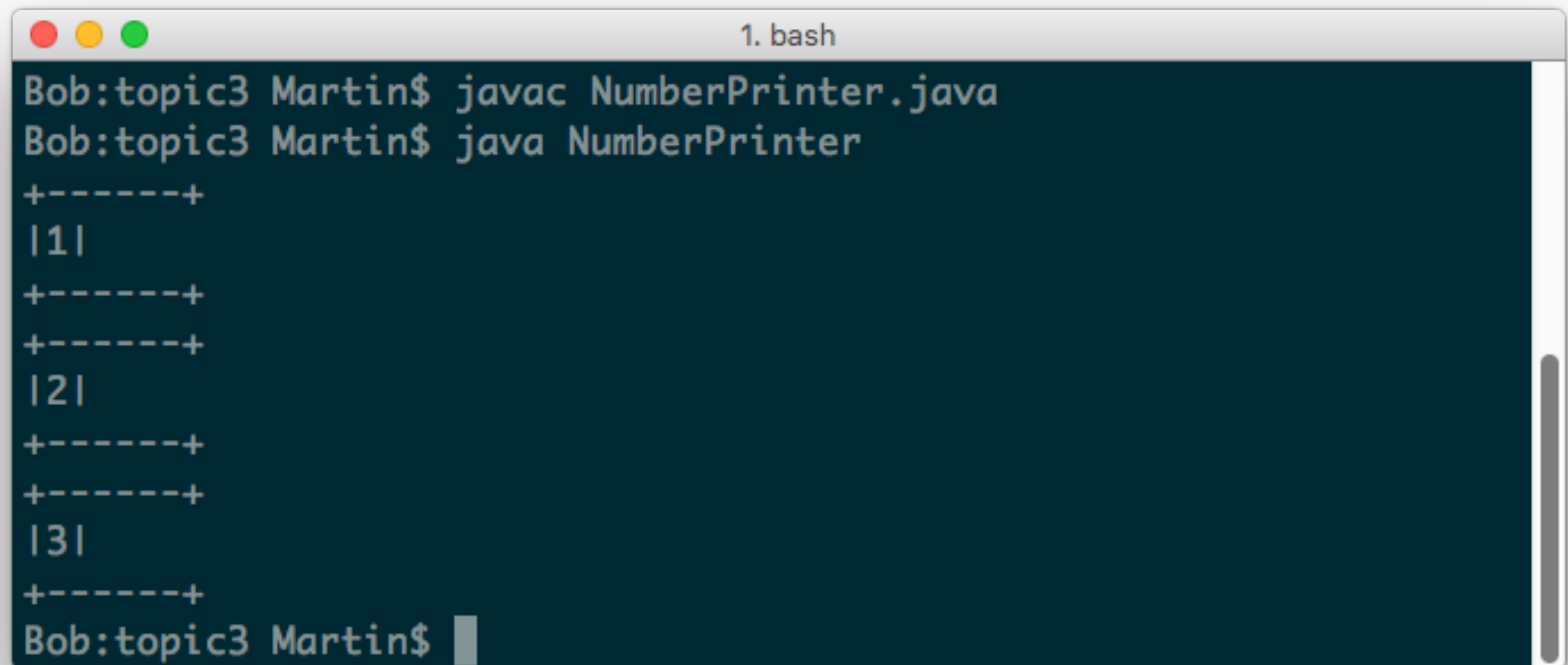
```
return 7;
```

What about method **input**?

What if I didn't just want to print `Martin' in a box to the screen, but any **number**, an arbitrary number of times?

```
System.out.println("+-----+");  
System.out.println("|    ?    |");  
System.out.println("+-----+");
```

AIM: BOXED NUMBERS

A terminal window titled "1. bash" with a dark blue background and light gray text. It shows the execution of a Java program. The prompt is "Bob:topic3 Martin\$". The first command is "javac NumberPrinter.java". The second command is "java NumberPrinter". The output consists of three lines, each preceded by a separator line "+-----+". The first line is "|1|", the second is "|2|", and the third is "|3|". The prompt "Bob:topic3 Martin\$" is visible at the bottom with a cursor.

```
Bob:topic3 Martin$ javac NumberPrinter.java
Bob:topic3 Martin$ java NumberPrinter
+-----+
|1|
+-----+
+-----+
|2|
+-----+
+-----+
|3|
+-----+
Bob:topic3 Martin$
```

I don't just want to **reuse** the same code, I want to **alter** its operation, in some small way.

LABELLING CODE (2)

It would be useful if we could not **only** label the code, but also provide **placeholders** for information to be filled in later.

`Print number`:

```
System.out.println("+-----+");  
System.out.println("|    3    |");  
System.out.println("+-----+");
```

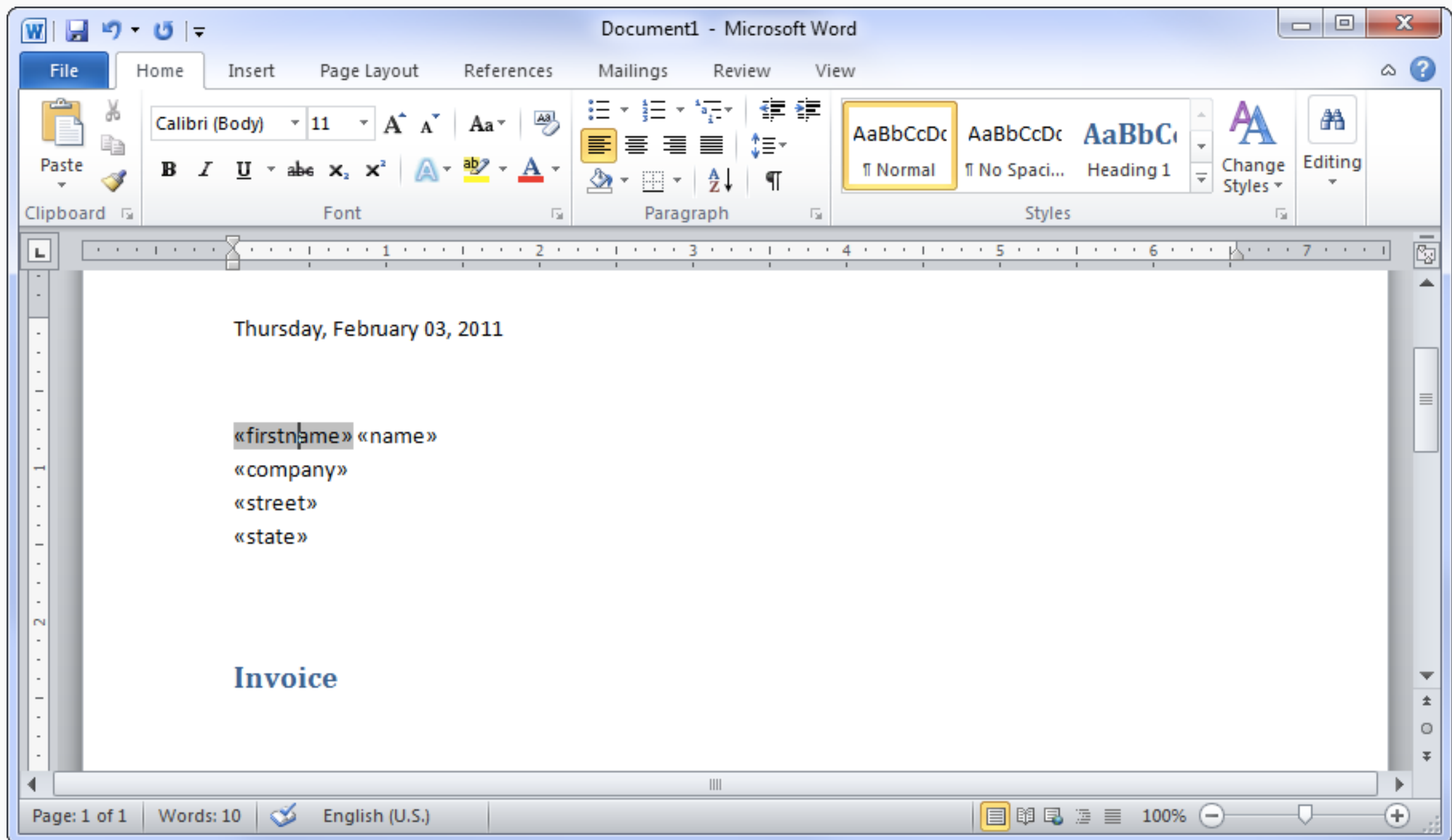
*Go to `Print number` set **NUM = 1*** ←

*Go to `Print number` set **NUM = 2***

*Go to `Print number` set **NUM = 3***



ASIDE: MAIL MERGE PLACEHOLDERS



METHODS: AN EXAMPLE (2)

Fortunately, we can do just this in Java.

```
public class NumberPrinter {
```

`Print number': `public static void printNumber(int num) {`

```
    System.out.println("+-----+");
```

```
    System.out.println("|" + num + "|"); <NUM>
```

```
    System.out.println("+-----+");
```

```
}
```

```
public static void main(String[] args) {
```

Go to `Print number' `printNumber(1); set NUM = 1`

Go to `Print number' `printNumber(2); set NUM = 2`

Go to `Print number' `printNumber(3); set NUM = 3`

```
}
```

```
}
```

NumberPrinter.java

ASIDE: CONCATENATION (1)

In this example, you will see the following syntax:

```
System.out.println("|" + num + "|");
```

Whenever a piece of text (e.g. "|") is **added**, using the plus symbol, to a variable (or another piece of text), and then **printed** to the terminal, they will appear **side-by-side**.

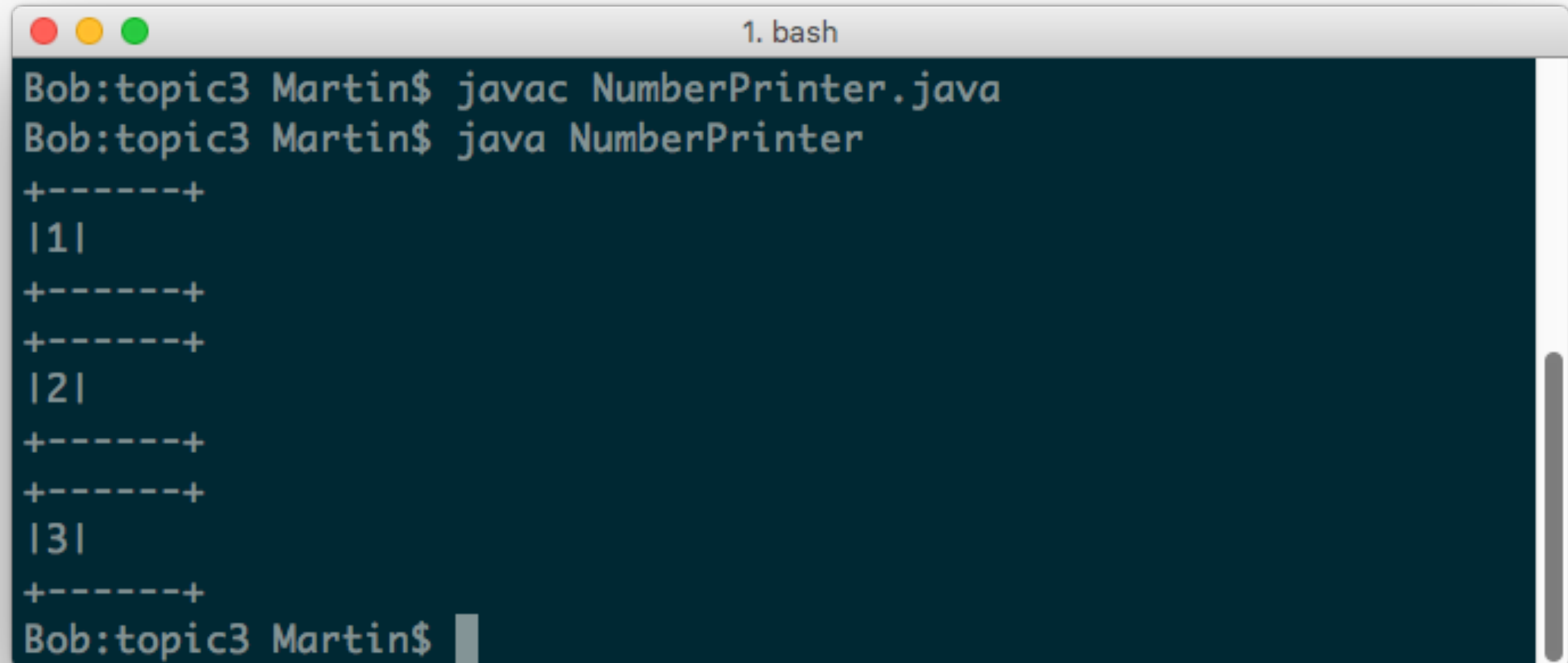
This is known as **concatenation** (loosely).

Open questions:

- Where else does concatenation occur?
- Is this the only way in which the '+' symbol is used?
- How is a variable converted to text?



ASIDE: CONCATENATION (2)



```
1. bash
Bob:topic3 Martin$ javac NumberPrinter.java
Bob:topic3 Martin$ java NumberPrinter
+-----+
|1|
+-----+
+-----+
|2|
+-----+
+-----+
|3|
+-----+
Bob:topic3 Martin$
```


METHODS: AN EXAMPLE (2) (BRACKETS)

Let's talk about the brackets.

```
public class NumberPrinter {  
    public static void printNumber(int num) {  
        System.out.println("+-----+");  
        System.out.println("|" + num + "|");  
        System.out.println("+-----+");  
    }  
  
    public static void main(String[] args) {  
        printNumber(1);  
        printNumber(2);  
        printNumber(3);  
    }  
}
```

NumberPrinter.java

PARAMETERS

We use the brackets at the end of a method to define our **placeholders** (e.g. **<NUM>**).

- These are **variable declarations**, of a form you should be familiar with from Topic 2.

```
er(int num) {
```

- We call these placeholder variable declarations **parameters**.

Once defined, these parameters (placeholders) can be **referenced anywhere inside** the method.

If we want to replace our parameters with values (like replacing **<NUM>** with the values 1, 2 and 3 previously) we have to **assign values** to these parameters.

METHODS: AN EXAMPLE (2) (PASSING DATA) (1)

To assign a parameter, we have to **call the method with a value written in brackets next to it**.

```
public class NumberPrinter {  
  
    public static void printNumber(int num) {  
  
        System.out.println("+-----+");  
        System.out.println("|" + num + "|");  
        System.out.println("+-----+");  
  
    }  
  
    public static void main(String[] args) {  
  
        printNumber(1);  
        printNumber(2);  
        printNumber(3);  
  
    }  
  
}
```

NumberPrinter.java

MATCHING LABELS

Remember that changing Java execution order, for simple method calls, relies on **matching** labels; we have to match the **method name** (and the brackets) in order to call the method and alter the flow of the program.

```
public static void printName() {  
    printName();  
}
```

This same is true for methods with parameters: we have to match the name, the **existence** of a parameter **and the type of the parameter** in order to call the method.

```
public static void printNumber(int number) {  
    printNumber(1);  
}
```

*The format of a method is often referred to as its **signature**.*



METHODS: AN EXAMPLE (2) (PASSING DATA) (2)

When this line is executed, the method will be matched (name, brackets, value and type), and any data in the brackets will be **pushed** to the waiting parameters of the method.

```
public class NumberPrinter {  
    public static void printNumber(int num) {  
        System.out.println("+-----+");  
        System.out.println("|" + num + "|");  
        System.out.println("+-----+");  
    }  
  
    public static void main(String[] args) {  
        printNumber(1);  
        printNumber(2);  
        printNumber(3);  
    }  
}
```

NumberPrinter.java

METHODS: AN EXAMPLE (2) (PASSING DATA) (3)

Thus we effectively transform the parameter into an **assigned variable**.

```
public class NumberPrinter {  
    public static void printNumber(int num = 1) {  
        System.out.println("+-----+");  
        System.out.println("|" + num + "|");  
        System.out.println("+-----+");  
    }  
  
    public static void main(String[] args) {  
        printNumber(1);  
        printNumber(2);  
        printNumber(3);  
    }  
}
```

(Not real Java code.)

METHODS: AN EXAMPLE (2) (PASSING DATA) (4)

As such, we effectively **transform** all **references** to this parameter into the value that is pushed.

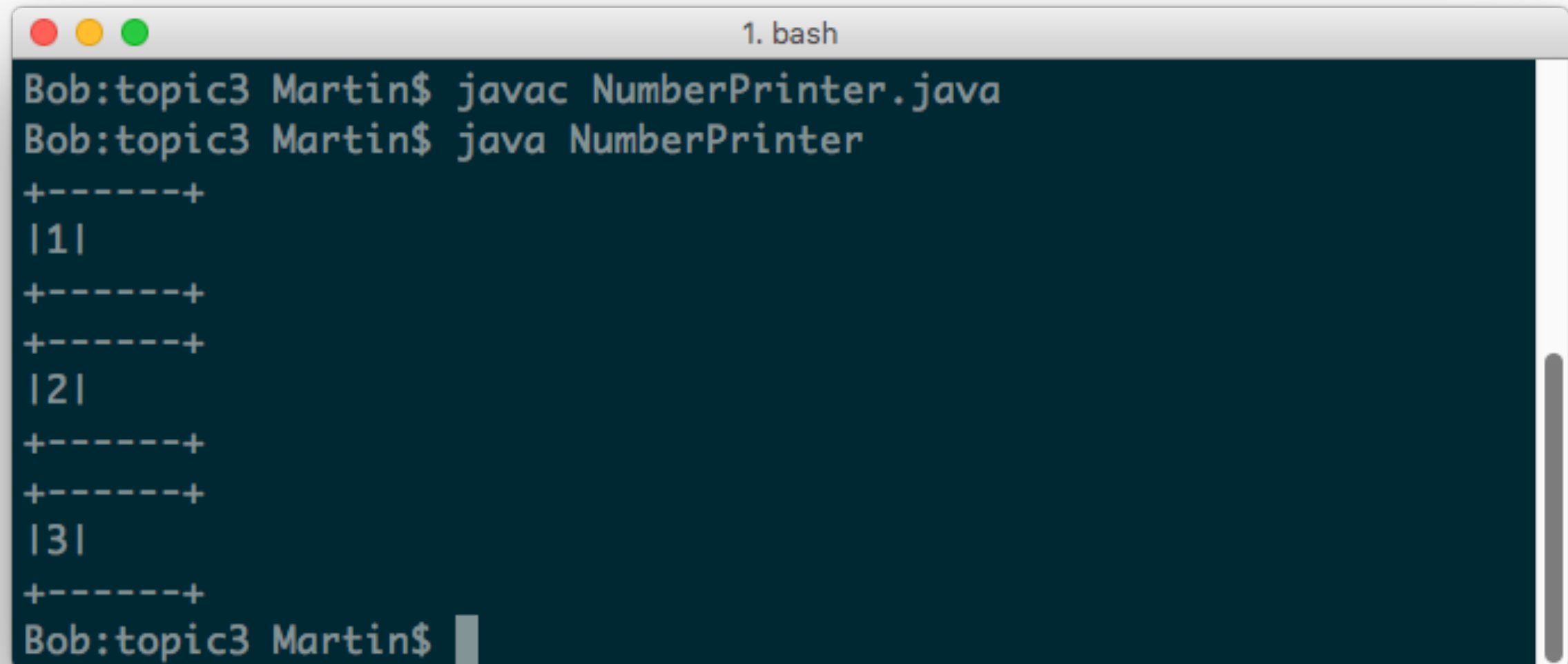
```
public class NumberPrinter {  
  
    public static void printNumber(1) {  
  
        System.out.println("+-----+");  
        System.out.println("|" + 1 + "|");  
        System.out.println("+-----+");  
  
    }  
  
    public static void main(String[] args) {  
  
        printNumber(1);  
        printNumber(2);  
        printNumber(3);  
  
    }  
  
}
```

We then repeat for the remaining calls, and receive the desired output...

(Not real Java code.)



OUTPUT: BOXED NUMBERS



```
1. bash
Bob:topic3 Martin$ javac NumberPrinter.java
Bob:topic3 Martin$ java NumberPrinter
+-----+
|1|
+-----+
+-----+
|2|
+-----+
+-----+
|3|
+-----+
Bob:topic3 Martin$
```


METHODS AND PARAMETERS: FLOW OF DATA

Again, It can be helpful to think about data **flowing through** the program.

```
public class NumberPrinter {  
  
    public static void printNumber(int num) {  
  
        System.out.println("+-----+");  
        System.out.println("|" + num + "|");  
        System.out.println("+-----+");  
  
    }  
  
    public static void main(String[] args) {  
  
        printNumber(1);  
        printNumber(2);  
        printNumber(3);  
  
    }  
  
}
```

*We will show data
flow like this again.*

NumberPrinter.java



So we now know that methods offer us (within the same class) the ability to:

- Use the same code **multiple times**.
- **Separate** functionality.
- Use the same code in **different ways**.

MULTIPLE PARAMETERS

The parameter syntax gives us a lot of freedom.

We can write a method with as many parameters as we like.

```
public static void printNumber(int num1, double num2) {
```

A further matching constraint: **the number of parameters must be the same** and the **types must be in the same order**.

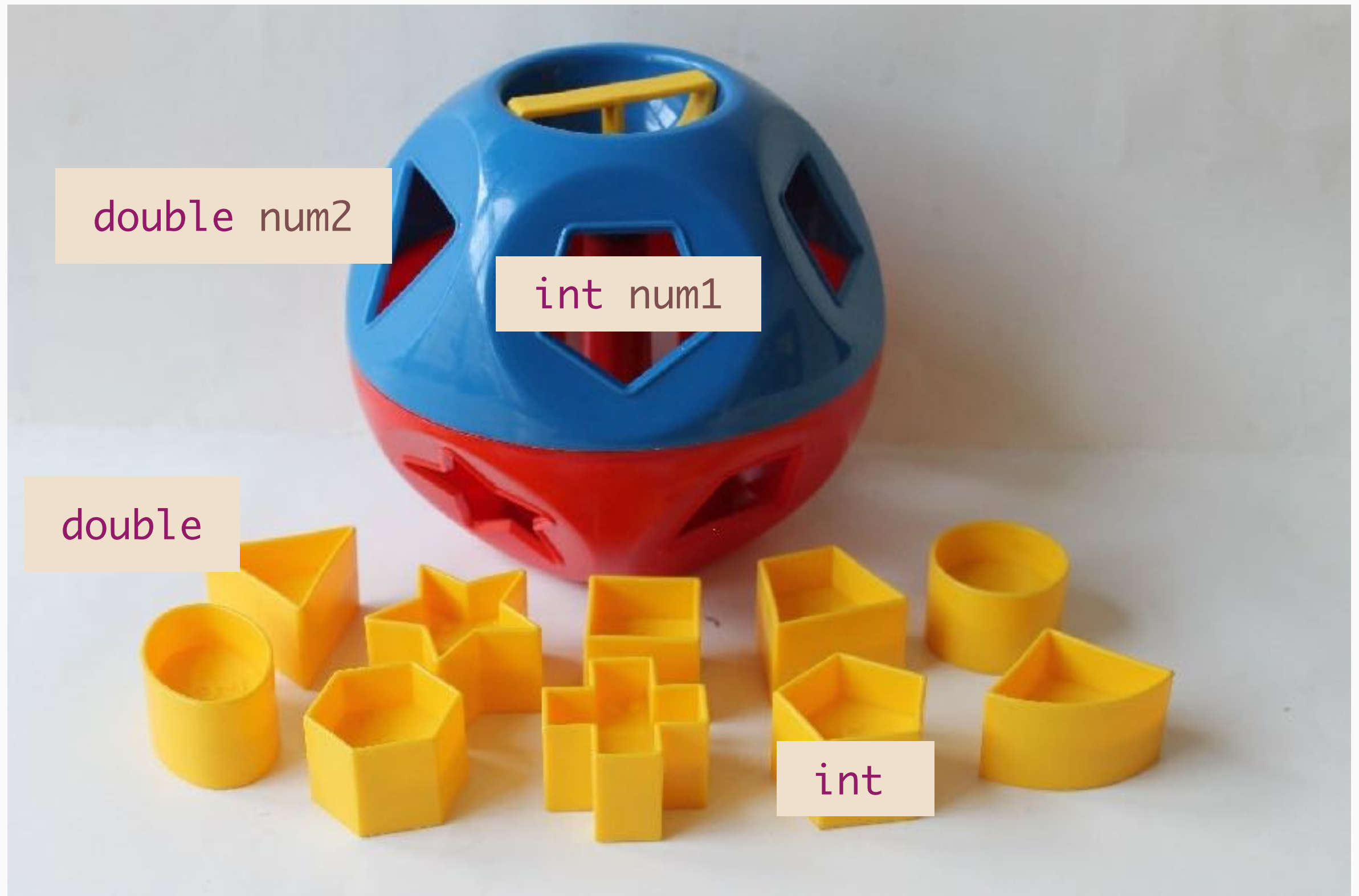
```
printNumber(1, 1.0);
```

But we can only ever return one **thing** (not equivalent to value).

- Otherwise, which returned entity would we **substitute** the call to the method with?



FITTING THE RIGHT PARAMETERS



LECTURE EXERCISE: SUBTRACTION (1)

Let's evolve the functionality from our `LargePrimes` class into a new class `Subtractor` that contains a method which allows us to subtract **any** two numbers from each other.

Then, let's subtract 15485863 from 32452843 again, and then go ahead and subtract 49979687 from 67867967.

LECTURE EXERCISE: SUBTRACTION (2)

```
public class Subtractor {  
    public static void subtract(int firstOperand, int secondOperand) {  
        System.out.println(firstOperand - secondOperand);  
    }  
  
    public static void main(String[] args) {  
        subtract(32452843, 15485863);  
        subtract(67867967, 49979687);  
    }  
}
```

Open question: Should we print the result, or return it for printing?

Anytime we now want to perform a subtraction in our class, we can simply call the subtract method (because we're lazy).

PASS BY REFERENCE VS. PASS BY VALUE (1)

So far, we've only passed **literal values** to our methods, but there's no reason we can't pass **variables** instead, as the **matching** rules relating to type are still maintained.

```
printNumber(1);
```

```
int numberOne = 1;  
printNumber(numberOne);
```

When we do this, a natural question to ask is, **if we change the passed variable inside the method, does the original value change?**



PASS BY REFERENCE VS. PASS BY VALUE (2)

```
public class NumberChanger {  
    public static void changeNumber(int changeMe) {  
        changeMe = 2;  
    }  
  
    public static void main(String[] args) {  
        int numberOne = 1;  
        changeNumber(numberOne);  
        System.out.println(numberOne);  
    }  
}
```

NumberChanger.java

Will the print statement print **1** or **2**?

Does Java pass **the variable itself** (the **reference**) or a **copy of the value** held in the variable?

Try this out in the lab.

Caveat: This is quite a weak example, more to come.


```
public static void main(String[] args) {
```

We now know a little bit more about **main**.

- We know that main can **never return** anything (where would it go anyway?)
- This is a **parameter**, so main must **accept** some information.
- But unfortunately, we **still** aren't in a position to discuss what is passed to the main method.
- And of course poor old public and static are neglected again.

METHOD ANNOTATION

Type out all the method syntax we have seen (**which you should be doing anyway**), and **annotate each method** with a brief description of what that method does:

```
/**
 * Prints the supplied number surrounded by a box.
 */
public static void printNumber(int num) {

    System.out.println("+-----+");
    System.out.println("|" + num + "|");
    System.out.println("+-----+");

}
```

ASIDE: COMMENT TYPES

```
/**
 * For methods, we should use special documentation comments,
 * which start with a double star after a slash.
 */
public static void printNumber(int num) {

    // Previously we use single line methods to annotate our code
    System.out.println("+-----+");
    System.out.println("|" + num + "|");
    System.out.println("+-----+");

    /* We can also use multi-line comments, if we like,
     * which start with a single star after a slash.
     */
}
```

BREAKING THE RULES, AGAIN



Once you've typed out all the methods and added documentation comments you should try and **break** things.

- Remove parameters, remove names, change names, change return types. **Call** the method in **different ways**.

```
// We can't call methods with a non-castable type  
changeNumber(1.0);
```

- Use single line comments to make notes **for yourself** again.

Organising Your Code Into Different Files

MARTIN PRINTER AND NUMBER PRINTER (1)

So far, we've introduced two key methods, one for printing Martin...

```
public class MartinPrinter {  
  
    public static void printMartin() {  
  
        System.out.println("+-----+");  
        System.out.println("|Martin|");  
        System.out.println("+-----+");  
  
    }  
  
    public static void main(String[] args) {  
  
        printMartin();  
        printMartin();  
        printMartin();  
  
    }  
  
}
```

MartinPrinter.java

MARTIN PRINTER AND NUMBER PRINTER (2)

...and one for printing numbers.

```
public class NumberPrinter {  
  
    public static void printNumber(int num) {  
  
        System.out.println("+-----+");  
        System.out.println("|" + num + "|");  
        System.out.println("+-----+");  
  
    }  
  
    public static void main(String[] args) {  
  
        printNumber(1);  
        printNumber(2);  
        printNumber(3);  
  
    }  
  
}
```

NumberPrinter.java

MARTIN PRINTER AND NUMBER PRINTER (3)

I kept these methods in **distinct classes** (in different files) to enable you to run most of the examples you find **independently**, in the laboratory sessions.

But what if we wanted to call **both** of these methods from the same **main method**?

- We want to print **`Martin'** followed by his **IQ (191)**.

```
printMartin();  
printNumber(191);
```


SEPARATING FUNCTIONALITY

Because we can have as many methods as we wish in a class, we could just move these methods into the **same class** and call them from the **same main method**.

Issues with this? What would we **call** this class?

```
public class MartinPrinterAndNumberPrinter {  
  
    public static void printMartin() {  
  
        System.out.println("+-----+");  
        System.out.println("|Martin|");  
        System.out.println("+-----+");  
  
    }  
  
    public static void printNumber(int num) {  
  
        System.out.println("+-----+");  
        System.out.println("|" + num + "|");  
    }  
}
```

It was actually quite nice having **separate functionality** in **different classes**, while using the **class name as a label** for that functionality.

There's also the risk of **saturating** a single class with too many methods (remember we can have as many methods in a class as we wish).

What's the **solution**?

SEPARATING OUT THE MAIN METHOD INTO A DRIVER CLASS (1)

The solution is to have have **neither** `printMartin()` or `printNumber(int num)` in the same class as the main method!

Instead, we create a **new class** that will **only hold the main method**.

We are going to call this class **Driver** because it is going to **drive** the rest of the code in our program, and serve as a **hub** for making things happen.

This class will **communicate** with both the `NumberPrinter` and `MartinPrinter` classes from which we will **remove** the main method.

- If we're trying to run multiple pieces of code in different files, typically we can only have **one** main method.



SEPARATING OUT THE MAIN METHOD INTO A DRIVER CLASS (2)

```
public class MartinPrinter {  
  
    public static void printMartin() {  
  
        System.out.println("+-----+");  
        System.out.println("|Martin|");  
        System.out.println("+-----+");  
  
    }  
  
}
```

Stored in a file called MartinPrinter.java

SEPARATING OUT THE MAIN METHOD INTO A DRIVER CLASS (3)

```
public class NumberPrinter {  
  
    public static void printNumber(int num) {  
  
        System.out.println("+-----+");  
        System.out.println("|" + num + "|");  
        System.out.println("+-----+");  
  
    }  
  
}
```

Stored in a file called NumberPrinter.java

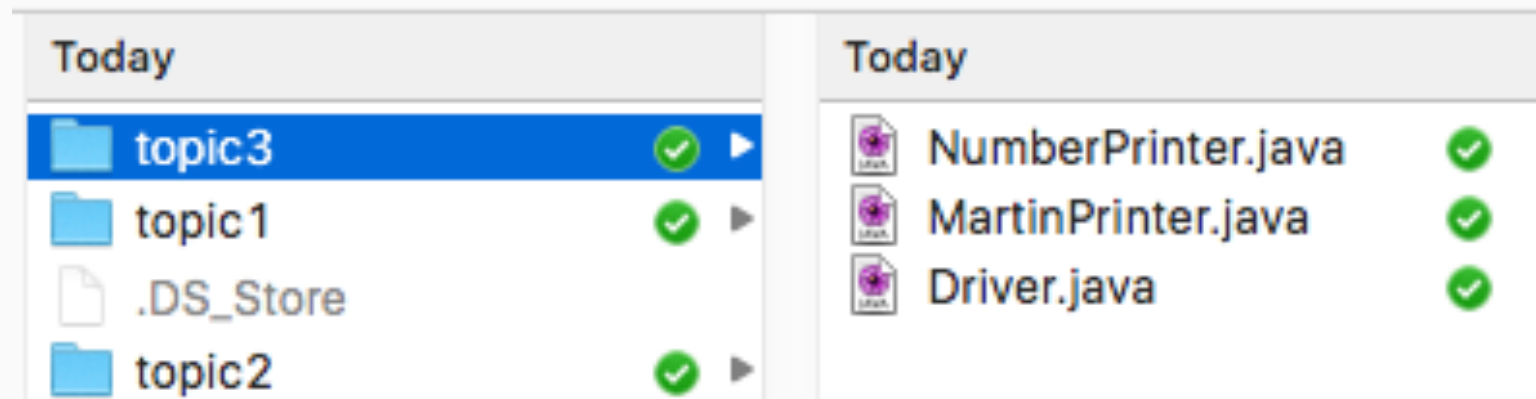
SEPARATING OUT THE MAIN METHOD INTO A DRIVER CLASS (4)

[illegible]

Stored in a file called Driver.java

ASIDE: FILES, CLASSES AND FOLDERS

If you wish to use the code from one class in another class without any additional code, those classes must be in the **same folder**.



```
1. bash
Bob:topic3 Martin$ ls
Driver.java          MartinPrinter.java  NumberPrinter.java
Bob:topic3 Martin$
```

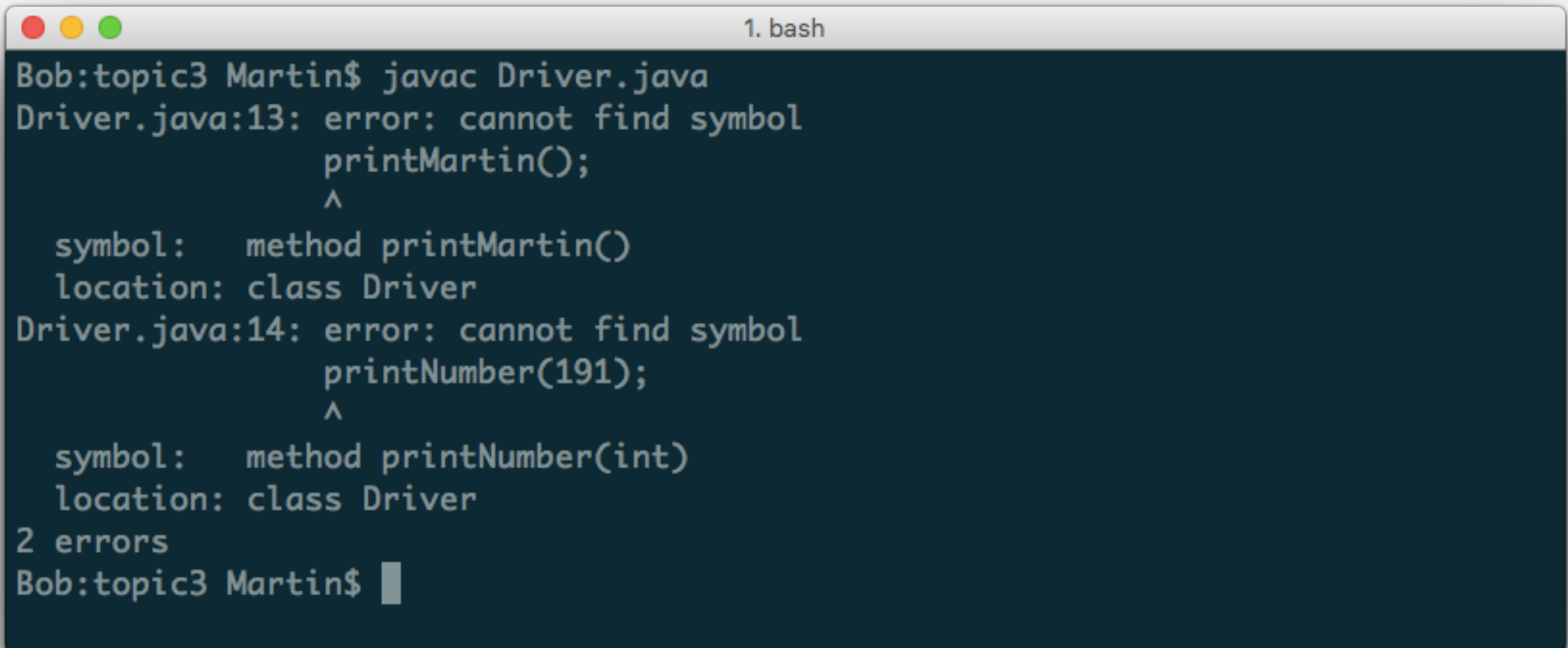
SEPARATING OUT THE MAIN METHOD INTO A DRIVER CLASS (4)

What now?

```
public class Driver {  
    public static void main(String[] args) {  
        printMartin();  
        printNumber(191);  
    }  
}
```

Stored in a file called Driver.java

ERROR: CANNOT FIND SYMBOL

A terminal window titled "1. bash" with a dark blue background and light gray text. It shows the output of a Java compilation command. The prompt is "Bob:topic3 Martin\$". The command "javac Driver.java" has been executed. The output shows two errors. The first error is on line 13: "error: cannot find symbol", followed by "printMartin();" and a caret (^) pointing to the method name. Below this, it says "symbol: method printMartin()" and "location: class Driver". The second error is on line 14: "error: cannot find symbol", followed by "printNumber(191);" and a caret (^) pointing to the method name. Below this, it says "symbol: method printNumber(int)" and "location: class Driver". The output ends with "2 errors" and the prompt "Bob:topic3 Martin\$" followed by a cursor.

```
Bob:topic3 Martin$ javac Driver.java
Driver.java:13: error: cannot find symbol
    printMartin();
      ^
  symbol:   method printMartin()
  location: class Driver
Driver.java:14: error: cannot find symbol
    printNumber(191);
      ^
  symbol:   method printNumber(int)
  location: class Driver
2 errors
Bob:topic3 Martin$
```

Not that easy.

ACCESSING A METHOD IN ANOTHER CLASS (FILE)

If we want to access a method in another class (file), we have to go through a **very specific process**.

This process involves making a **copy** of all the code in that class and **storing it inside a variable (!)**.

We can then **interact with the copied code through the variable** in order to call methods **within** that code.



LET'S IMAGINE WHAT THIS MIGHT LOOK LIKE...

```
public class Driver {  
    public static void main(String[] args) {  
        Type copyOfMartinPrinter  
    }  
}
```

This is our special variable declaration, which is still a type followed by a name (we have chosen).

Driver.java



THE VARIABLE TYPE USED TO STORE A COPY OF A CLASS (1)

Remember that for something to be a variable it must have a **type** followed by a **name**.

Given that we have a variable, storing a copy of a class, what should the type of this variable be?

```
public class Driver {  
    public static void main(String[] args) {  
        Type copyOfMartinPrinter  
    }  
}
```

Driver.java



THE VARIABLE TYPE USED TO STORE A COPY OF A CLASS (2)

The answer is the **name of the class we're copying**.

This makes sense because a variable type **describes** (the format of) **what the variable will contain** (in this case a copy of a particular class).

```
public class Driver {  
    public static void main(String[] args) {  
        MartinPrinter copyOfMartinPrinter  
    }  
}
```

Driver.java



MAKING A COPY OF A CLASS

How do we do this?

We need to **request** a **new** copy of the class. We use the **labelled name** of the class so that Java is able to **match** the class we want.

Then we need to **put** this copy inside the variable, using an **assignment**, in the same way that we put **values** inside variables.

```
public class Driver {  
    public static void main(String[] args) {  
        MartinPrinter copyOfMartinPrinter = new MartinPrinter();  
    }  
}
```

(We'll come back to what the brackets mean).

Driver.java

We will call a variable that contains a **copy** of a class an **object**.



This process is called *making an object of a class*, and forms the foundation of object-oriented programming.

ACCESSING A METHOD IN ANOTHER CLASS (FILE)

If we want to access a method in another class (file), we have to go through a **very specific process**.

This process involves making a **copy** of all the code in that class and **storing it inside a variable (!)**.

We can then **interact with the copied code through the variable** in order to call methods **within** that code.



INTERACTING WITH COPIES OF CLASSES (OBJECTS) (1)

In order to interact with the code stored in a variable (object), and thus call methods, we need to **open the variable (object) up and look inside.**

To do this we write our variable (object) name and then a **dot.**

```
public class Driver {  
    public static void main(String[] args) {  
        MartinPrinter copyOfMartinPrinter = new MartinPrinter();  
        copyOfMartinPrinter.  
    }  
}
```

Driver.java

INTERACTING WITH COPIES OF CLASSES (OBJECTS) (2)

When we write a dot after an object (a variable containing a copy of a class), we can see **inside** the object, into the copied class, and have the option to reference **any of the (public) identifiers within it**.

```
public class Driver {  
    public static void main(String[] args) {  
        MartinPrinter copyOfMartinPrinter = new MartinPrinter();  
        copyOfMartinPrinter.printMartin();  
    }  
}
```

In other words, we can call any of the methods within the copy.

Driver.java

INTERACTING WITH COPIES OF CLASSES (OBJECTS) (3)

To reiterate: The dot is an **entry** point through the object into the class copy through which we **reference** items in that copy of the class.

copyOfMartinPrinter.

```
public class MartinPrinter {  
    public static void printMartin()  
    {  
        System.out.println("+-----+");  
        System.out.println("|MartinPrinter|");  
        System.out.println("+-----+");  
    }  
}
```

ACHIEVING OUR GOAL

- We want to print **Martin** followed by his **IQ** (191).

```
public class Driver {  
    I'm using blank lines here more liberally than  
    usual, in order to make my code clearer.  
    public static void main(String[] args) {  
  
        MartinPrinter copyOfMartinPrinter = new MartinPrinter();  
        copyOfMartinPrinter.printMartin();  
  
        NumberPrinter copyOfNumberPrinter = new NumberPrinter();  
        copyOfNumberPrinter.printNumber(191);  
  
    }  
}
```

*When accessing a method through an object,
we call methods with parameters in exactly
the same way.*

Driver.java

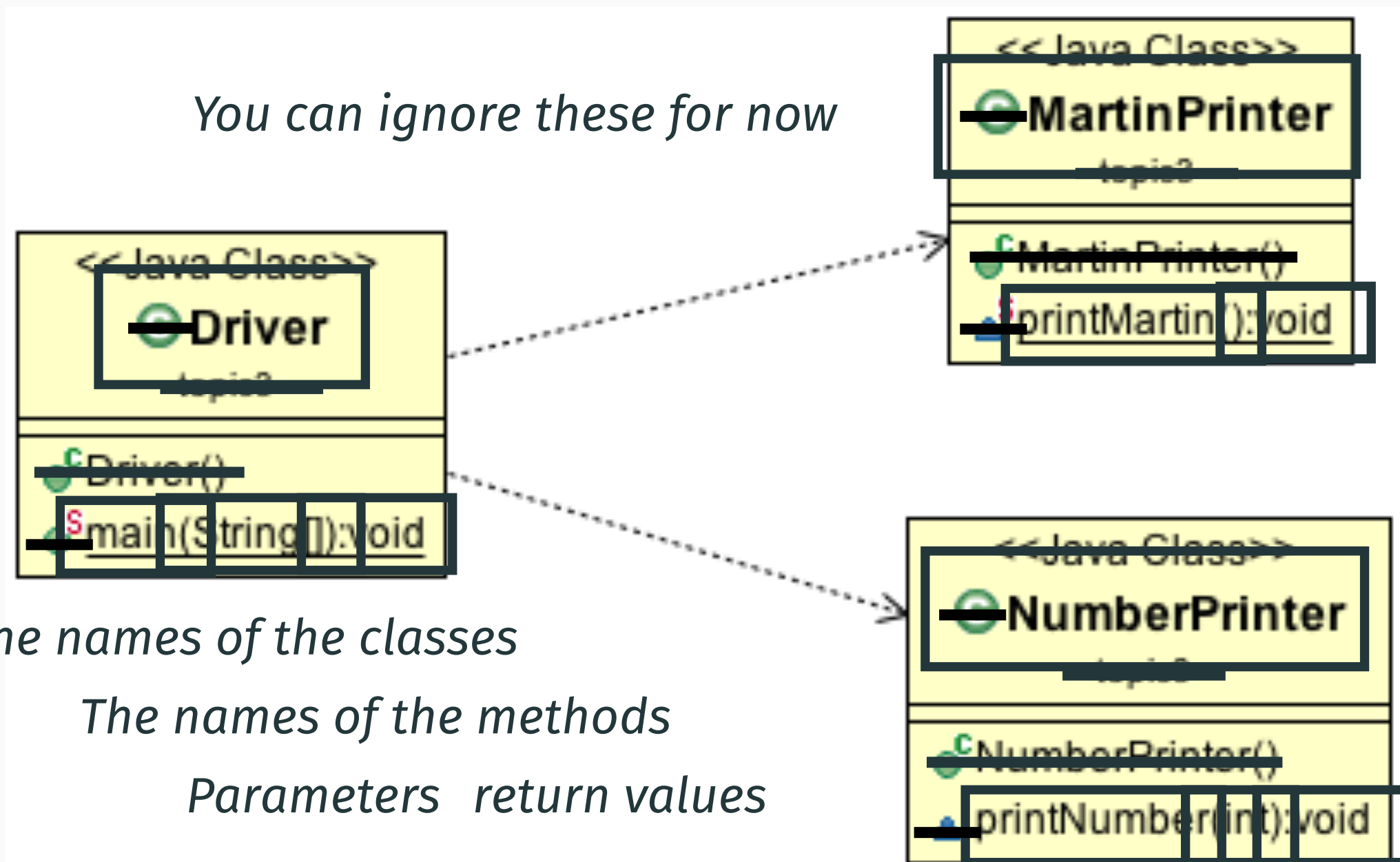
THE OUTPUT

```
1. bash
Bob:topic3 Martin$ ls
Driver.java          MartinPrinter.java    NumberPrinter.java
Bob:topic3 Martin$ javac Driver.java
Bob:topic3 Martin$ ls
Driver.class         MartinPrinter.class    NumberPrinter.class
Driver.java          MartinPrinter.java      NumberPrinter.java
Bob:topic3 Martin$ java Driver
+-----+
|Martin|
+-----+
+-----+
|191|
+-----+
Bob:topic3 Martin$
```

Note that we only need to compile the driver class directly, the compiler will automatically compile any other referenced classes.

ASIDE: DOCUMENTING OUR CODE WITH CLASS DIAGRAMS

Now that we have a program of reasonable structural complexity, it's useful to have a documentation technique to visualise our class structure.



You will need to produce these diagrams (i.e. draw them yourselves) as part of your documentation for each assignment.



ONE FINAL THING: REMEMBER OUR RULE ABOUT STATIC

Methods have to be static if they are called from a method that is also static.

*Because these methods are now being called **through an object** we can **drop** the static keyword.*

```
public class MartinPrinter {  
  
    public void printMartin() {
```

```
    }  
}  
  
public class NumberPrinter {  
  
    public void printNumber(int num) {  
  
        System.out.println("+-----+");  
        System.out.println("|" + num + "|");  
        System.out.println("+-----+");  
    }  
}
```



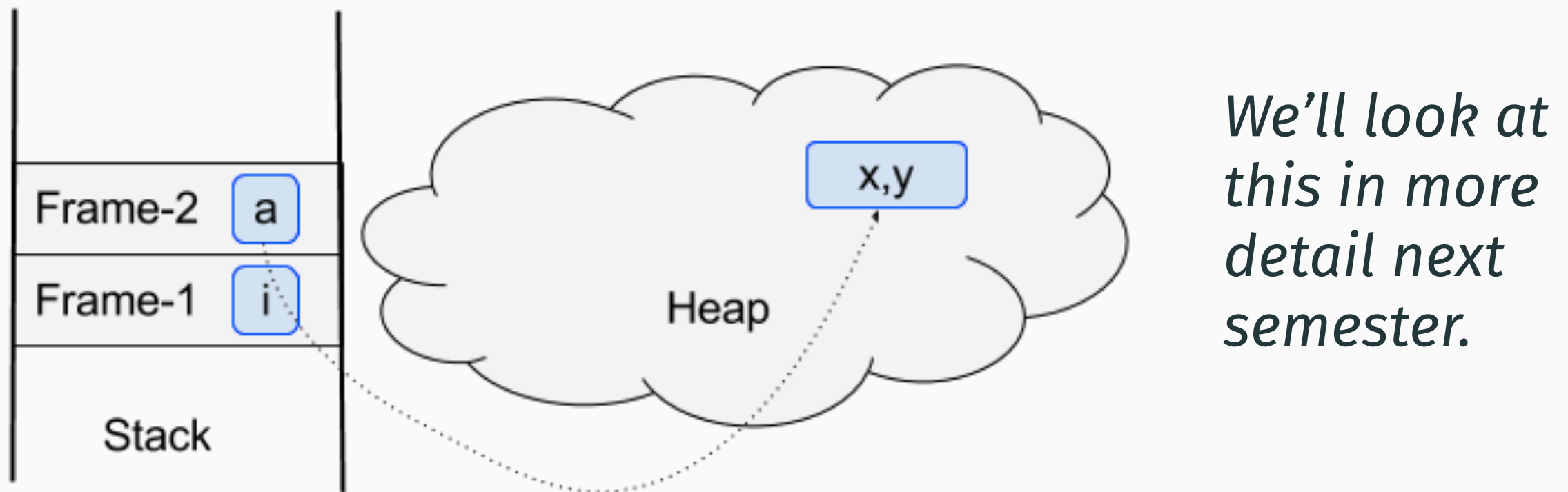
Object-oriented **purists** would be **angry** that I'm only selling classes (and their associated objects) as a **way to organise code**, because the idea is much more **powerful**.

- But I think this is a good **initial** way to understand things.
- We will gradually see more important reasons for using classes and objects going forward.

If any of the further information on objects and classes confuses you, just come back to this idea that an object is just a **copy of the code in a class**.

ASIDE: OBJECTS IN MEMORY

We will use the definition of an object quite loosely in the first semester of PPA, in order to **simplify** things, and help your initial understanding.



In reality, unlike primitive values, objects are stored in another area of the JVM's memory known as the **heap**. The variable containing the copy of the class is actually a **memory reference** from a variable on the stack to an object on the heap.

LECTURE EXERCISE: HELLO WORLD (1)

Let's wrap the `Hello World` functionality from Topic 1 (i.e. a single print statement printing `Hello World`) in a method called `printHelloWorld`, within a class called `HelloWorld`, and then make an object of this class in a `Driver` class in order to call the `printHelloWorld` method.

LECTURE EXERCISE: HELLO WORLD (2)

```
public class HelloWorld {  
    public void printHelloWorld() {  
        System.out.println("Hello World");  
    }  
}
```

```
public class Driver {  
    public static void main(String[] args) {  
        HelloWorld copyOfHelloWorld = new HelloWorld();  
        copyOfHelloWorld.printHelloWorld();  
    }  
}
```

Topic 3: Organising Your Code

Programming Practice and Applications (4CCS1PPA)

Dr. Martin Chapman
Thursday 6th October

programming@kcl.ac.uk
martinchapman.co.uk/teaching

These slides will be available on KEATS, but will be subject to ongoing amendments. Therefore, please always download a new version of these slides when approaching an assessed piece of work, or when preparing for a written assessment.