Topic 7: Arrays

Programming Practice and Applications (4CCS1PPA)

Dr. Martin Chapman
Thursday 17th November

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

Q: What is the non-technical definition of an array?

ARRAYS: OBJECTIVES

To understand how to employ the **array** data structure in our programs.

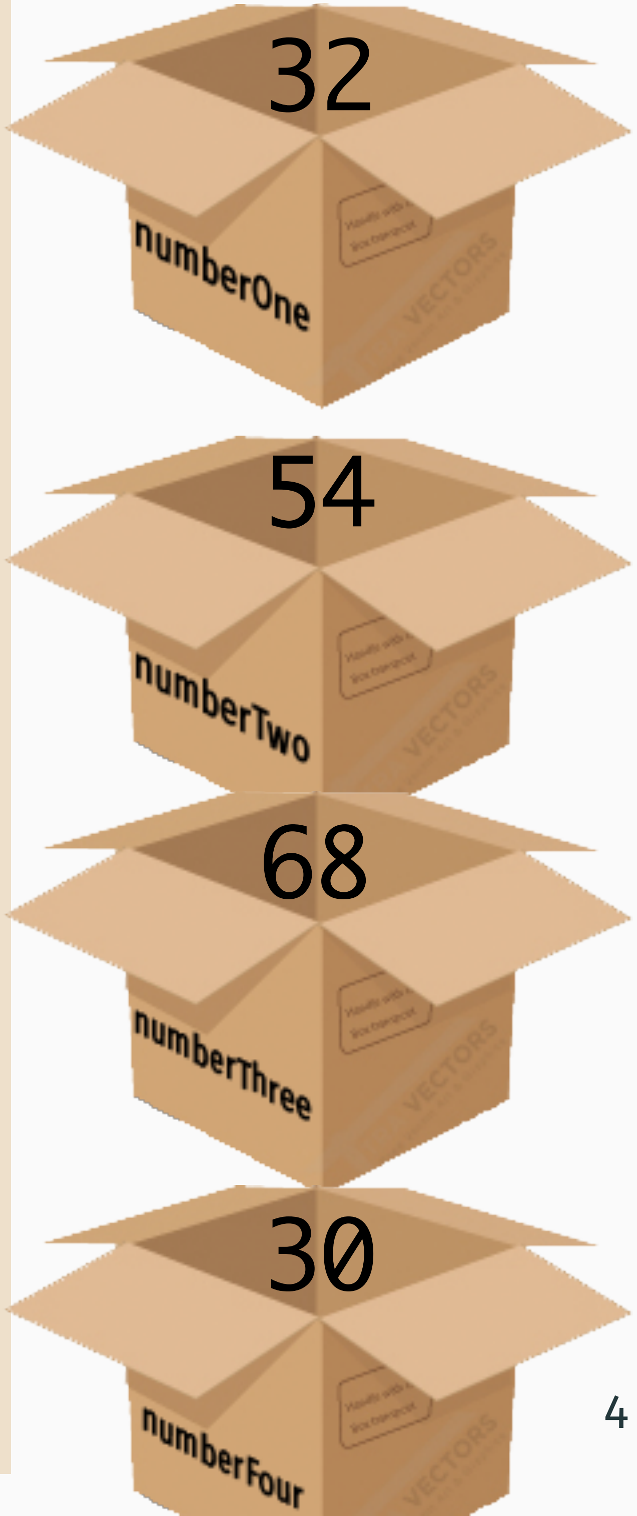
- To build a working representation of the game of Noughts and Crosses.

EXERCISE: HANDLING SETS OF NUMBERS (1)

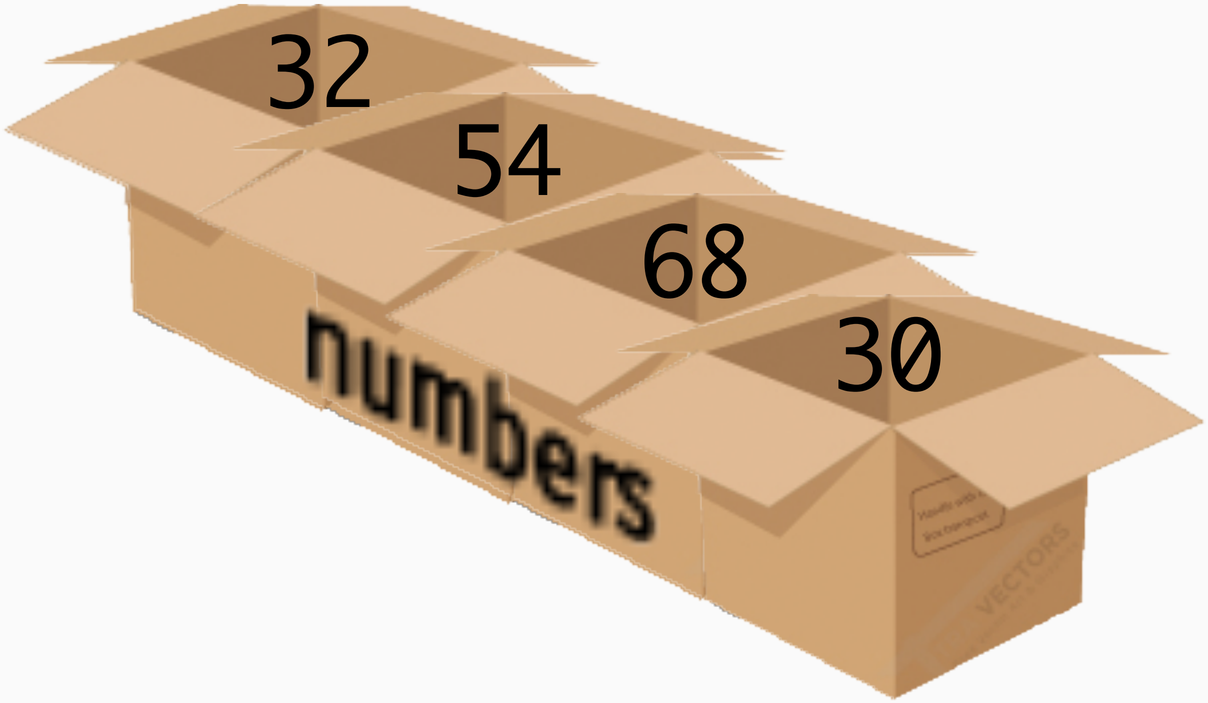
Given this set of numbers — 32, 54, 68 and 30 — write a program that stores these numbers, and then examines each of them and prints the word YES if the number is **greater** than the **average (46)**.

EXERCISE: HANDLING SETS OF NUMBERS (2)

```
int numberOne, numberTwo, numberThree, numberFour;

numberOne = 32;

if ( numberOne > 46 ) {

    System.out.println("YES");

}

numberTwo = 54;

if ( numberTwo > 46 ) System.out.println("YES");

numberThree = 68;

if ( numberThree > 46 ) System.out.println("YES");

numberFour = 30;

if ( numberFour > 46 ) System.out.println("YES");
```



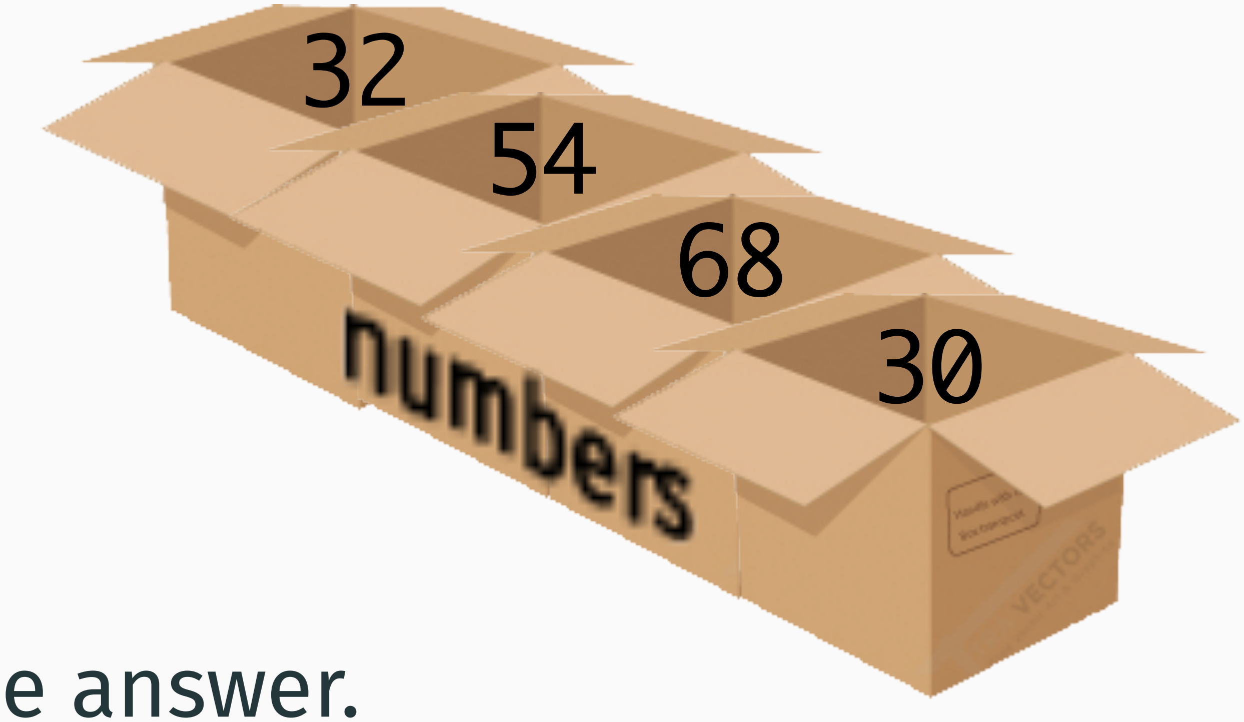
EXERCISE: HANDLING SETS OF NUMBERS (3)



Ideally, rather than having to interact with each stored integer **individually**, we want to interact with a **single set** of **connected** integers.

This necessitates a **single** label for all the stored values.

ARRAYS (1)



An **array** is the answer.

- Dictionary definition: ‘**To arrange a group of things in a particular order**’.

An array is a **type** of variable with **multiple slots**. Each slot can hold a different value.

The values in an array must all be of the **same type**.

ARRAYS (2)

`A array is a **type** of variable’

```
int numberOne, numberTwo, numberThree, numberFour;
```

```
int[] numbers
```

We place square brackets after a type to indicate that we want to store a sequence of values of this type.

`with multiple **slots**.’

```
int[] numbers = new int[4];
```

`My numbers variable has 4 slots’

`Each slot can hold a different value.’

```
numberOne = 32;

numberTwo = 54;

numberThree = 68;

numberFour = 30;
```

```
numbers[0] = 32;

numbers[1] = 54;

numbers[2] = 68;

numbers[3] = 30;
```

ARRAYS (3): CREATE AND STORE (SUMMARY)

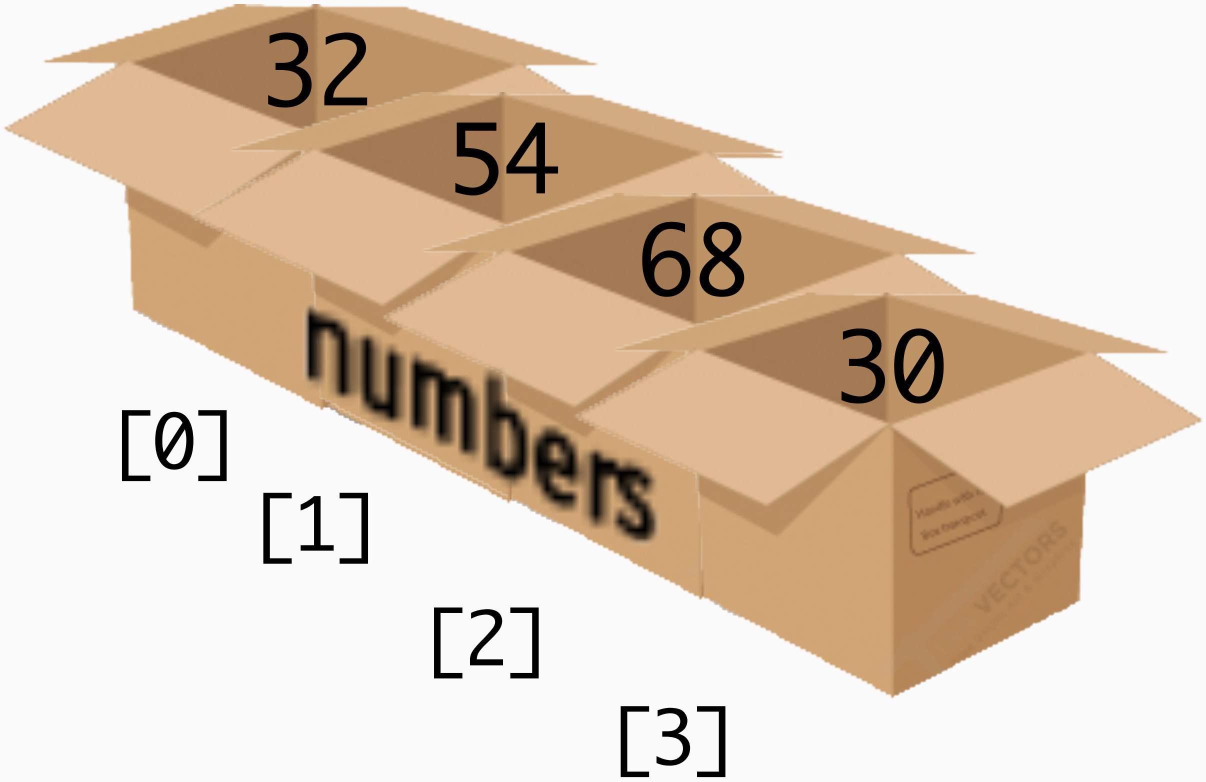
Create an array...

```
int[] numbers = new int[4];
```

The length of the array differs from the last index.

...and then store things in that array.

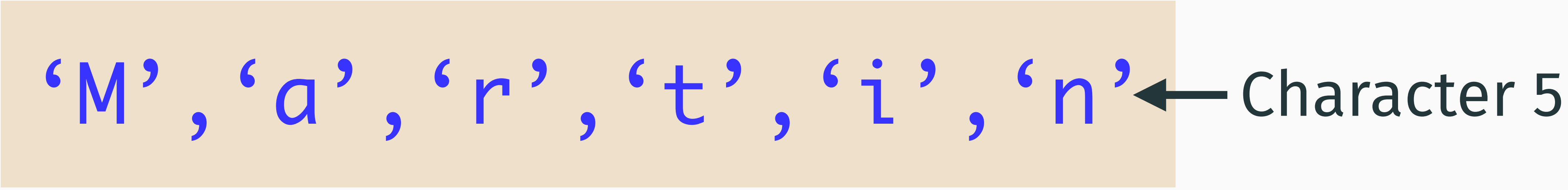
```
numbers[0] = 32;

numbers[1] = 54;

numbers[2] = 68;

numbers[3] = 30;
```



REMEMBER: INDEX VS. LENGTH

Because of this breakdown, we can imagine each character in a string as having an **index**.

- This means that there is a distinct difference between the **length** of a string and the **last index**.



```
‘M’,‘a’,‘r’,‘t’,‘i’,‘n’
```

Character 5

Length = 6.

We can leverage this idea in order to approach our current problem.

ARRAYS (3): CREATE AND STORE (SINGLE LINE)

```
int[] numbers = { 32, 54, 68, 30 };
```

We could **combine** the steps we saw previously in order to **declare** and **initialise** an array in a **single line**.

- A similar principle as a single line declaration and initialisation for a **normal variable**.
- Thus, we would need to know the values we want in that array **immediately**.

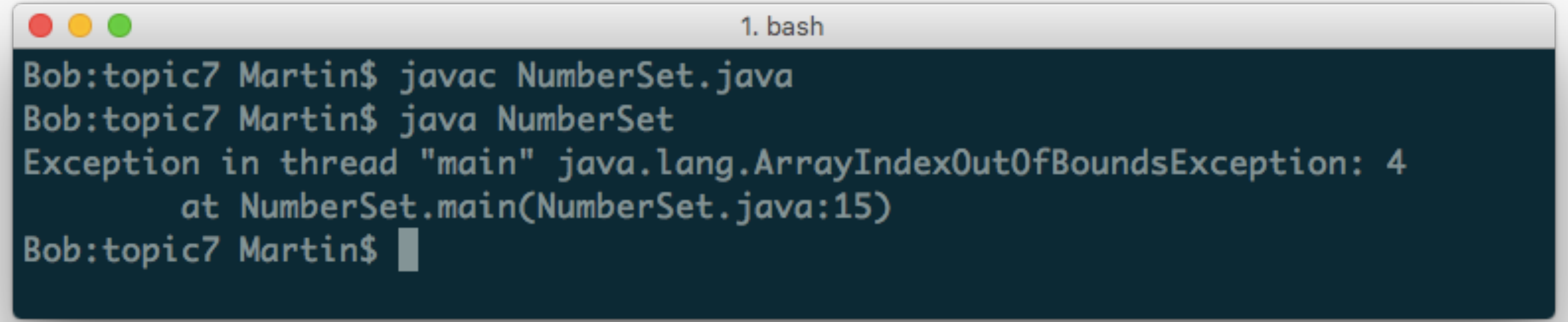
This would give us an array of a size **equal** to the number of values.

FIXED ARRAY SIZE

```
int[] numbers = new int[4];

numbers[0] = 32;

numbers[1] = 54;

numbers[2] = 68;

numbers[3] = 30;

numbers[4] = 52;
```

When we specify the initial size of an array, this size is **fixed**, such that if we try and write to a non-existent index, the following occurs:

ERROR: INDEX OUT OF BOUNDS EXCEPTION



```
Bob:topic7 Martin$ javac NumberSet.java
Bob:topic7 Martin$ java NumberSet
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
        at NumberSet.main(NumberSet.java:15)
Bob:topic7 Martin$ 
```

Note that this error doesn't occur at **compile time** (after the compiler is run), but instead at **run time** (after the virtual machine is run).

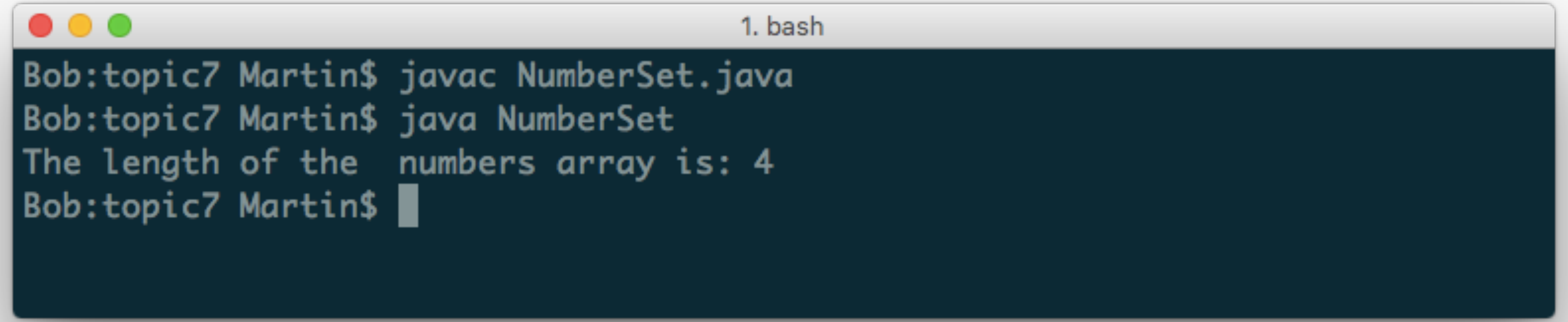
- The compiler is (necessarily) too simplistic to catch this issue at compile time.
- It also isn't always possible to identify this type of error at compile time. For example, access to, or the size of, an array may occur as the result of **user input**.
  - We also saw an **InputMismatchException** runtime error in Topic 6.
- We'll return to these ideas.

HOW DO WE AVOID A NON-EXISTENT INDEX IN AN ARRAY BEING WRITTEN TO? (1)

We first need to work out how to determine the **length** of an array.

If we want to split a string over two lines, we need to concatenate.

```
int[] numbers = new int[4];

System.out.println("The length of the  "
        + "numbers array is: " +  numbers.length);
```



1. bash

```
Bob:topic7 Martin$ javac NumberSet.java
Bob:topic7 Martin$ java NumberSet
The length of the  numbers array is: 4
Bob:topic7 Martin$ 
```

HOW DO WE AVOID A NON-EXISTENT INDEX IN AN ARRAY BEING WRITTEN TO? (2)

We can then combine this information with a **conditional statement**, to ensure that a non-existent index in an array is not written to.

```
int i = 4, value = 52;

int[] numbers = new int[4];

if ( 0 <= i && i < numbers.length ) {

    numbers[i] = value;

}
```

This test would become particularly important if we weren't (directly) in ***control*** *of the value in i.*

ASIDE: ARRAY LENGTH VS. STRING LENGTH

We’ve seen the word length used in two places now.

- To determine the length of a string.

  ```
  password.length()
  ```

  - This is a **method call** (brackets).

- To determine the length of an array.

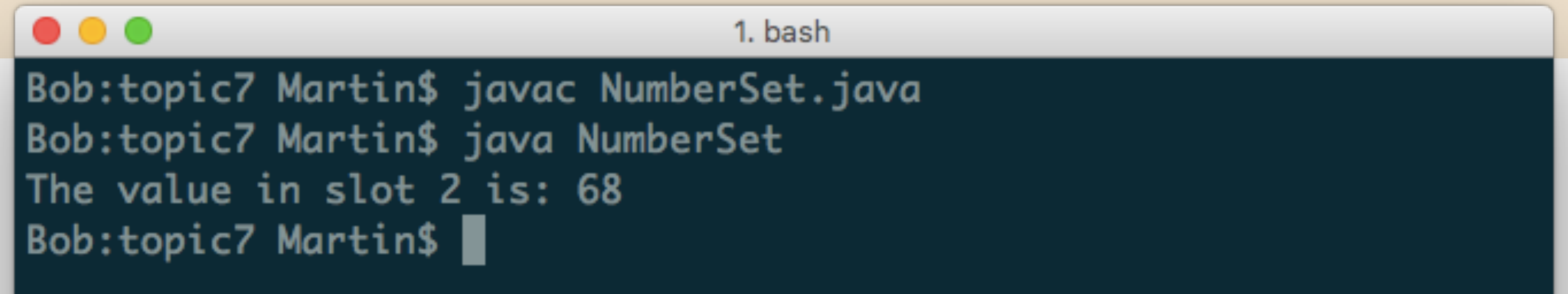
  ```
  numbers.length
  ```

- This resembles a reference to a **field** (no brackets).
  - More shortly.

HOW DO WE GET DATA BACK FROM AN ARRAY?

In order to get data back from an array, we simply reference the **index** of the data that we want.

```
int[] numbers = new int[4];

numbers[0] = 32;

numbers[1] = 54;

numbers[2] = 68;

numbers[3] = 30;

System.out.println("The value in slot 2 is: " +
                                               numbers[2]);
```



```
Bob:topic7 Martin$ javac NumberSet.java
Bob:topic7 Martin$ java NumberSet
The value in slot 2 is: 68
Bob:topic7 Martin$
```

EXERCISE: BACK TO HANDLING SETS OF NUMBERS (1)

Given this set of numbers — 32, 54, 68 and 30 — write a program that stores these numbers, and then examines each of them and prints the word YES if the number is **greater** than the **average (46)**.

Hint: Think about how we might extend the idea of using index and length, as seen in Slide 14.

REMEMBER: LEVERAGING THE LOOP INDEX (3)

```java
public class NumberPrinter {

   public void printNumber(int num) {

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

   }

}
```

```java
public class Driver {

   public static void main(String[] args) {

      NumberPrinter numberPrinter = new NumberPrinter();

      for ( int i = 1; i < 4; i++ ) {

         numberPrinter.printNumber(i);

      }

   }

}
```

*As the loop index is just a variable declaration, it can be referenced **inside** the loop.*

EXERCISE: BACK TO HANDLING SETS OF NUMBERS (2)

```
int[] numbers = new int[4];

numbers[0] = 32;

numbers[1] = 54;

numbers[2] = 68;

numbers[3] = 30;

for ( int i = 0; i < numbers.length; i++ ) {

   if ( numbers[i] > 46 ) System.out.println("YES");

}
```

We could shorten this to a single line initialisation and declaration.

We use our knowledge of loop bounds to carefully control how we access each index.

CAPTURING AN UNKNOWN AMOUNT OF USER INPUT (1)

Let's imagine we want to extend our number checker to instead **store** an **arbitrary** amount of numbers from a user, and then check which are over the average.

REMEMBER: TRACKING A USER’S CALORIES (3)

```
import java.util.Scanner;

public class CalorieTracker {

  public static void main(String[] args) {

    Scanner in = new Scanner(System.in);

    Person person = new Person();

    System.out.println("Enter the calories in your starter:");

    Dish starter = new Dish();
    starter.setCalories(in.nextInt())
```

Every time we invoke the nextInt method, our program will stop and wait for ***another*** *token of user input.*

CAPTURING AN UNKNOWN AMOUNT OF USER INPUT (2)

Let’s fill in as much of the solution as we can:

```
import java.util.Scanner;

Scanner in = new Scanner(System.in);

int[] numbers = new int[ ? ];

for ( int i = 0; i <       ?       ; i++ ) {

    numbers[i] = in.nextInt();

}

in.close();
```

Familiar scanner syntax.

Familiar array syntax.

What are our options here?

OPTION 1: LET THE USER SPECIFY A MAXIMUM

```java
import java.util.Scanner;

Scanner in = new Scanner(System.in);

int[] numbers = new int[ in.nextInt() ];

for ( int i = 0; i < numbers.length; i++ ) {

    numbers[i] = in.nextInt();

}

in.close();
```

What if the user themselves is unsure?

OPTION 2: SPECIFY A MAXIMUM FOR THE USER (1)

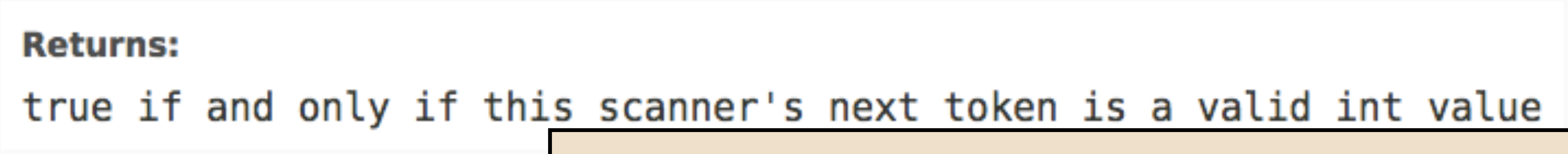
```java
import java.util.Scanner;
Scanner in = new Scanner(System.in);
int[] numbers = new int[100];
for ( int i = 0; i < numbers.length; i++ ) {
    numbers[i] = in.nextInt();
}
in.close();
```

OPTION 2: SPECIFY A MAXIMUM FOR THE USER (2)

```
import java.util.Scanner;

Scanner in = new Scanner(System.in);

int[] numbers = new int[100];

for ( int i = 0; i < numbers.length; i++ ) {

    numbers[i] = in.nextInt();

}

in.close();
```

What if someone wants to enter **less** than 100 numbers?

OPTION 2: SPECIFY A MAXIMUM FOR THE USER (3)



Returns:

`true if and only if this scanner's next token is a valid int value`

```
System.out.println("Enter N numbers." +
                   "Type 'done' to finish.");
```

```
import jav

Scanner in = new Scanner(System.in);

int[] numbers = new int[100];

while ( in.hasNextInt() ) {

      numbers[ ] = in.nextInt();

}

in.close();
```

We could keep looping while there are still integers to read.

How do we know where to place the next number?

What if someone wants to enter **less** than 100 numbers?

PARTIALLY FILLED ARRAYS (1)

This problem stems from the fact that arrays (initially) have to be a **fixed size**.

Therefore, if a programmer wants to account for an **unknown** number of input values (such as input coming from a user), they have to specify a large fixed size (like we have done), and prepare for this array to be **partially filled** with values.

Storing values in a partially filled array requires an extra step...

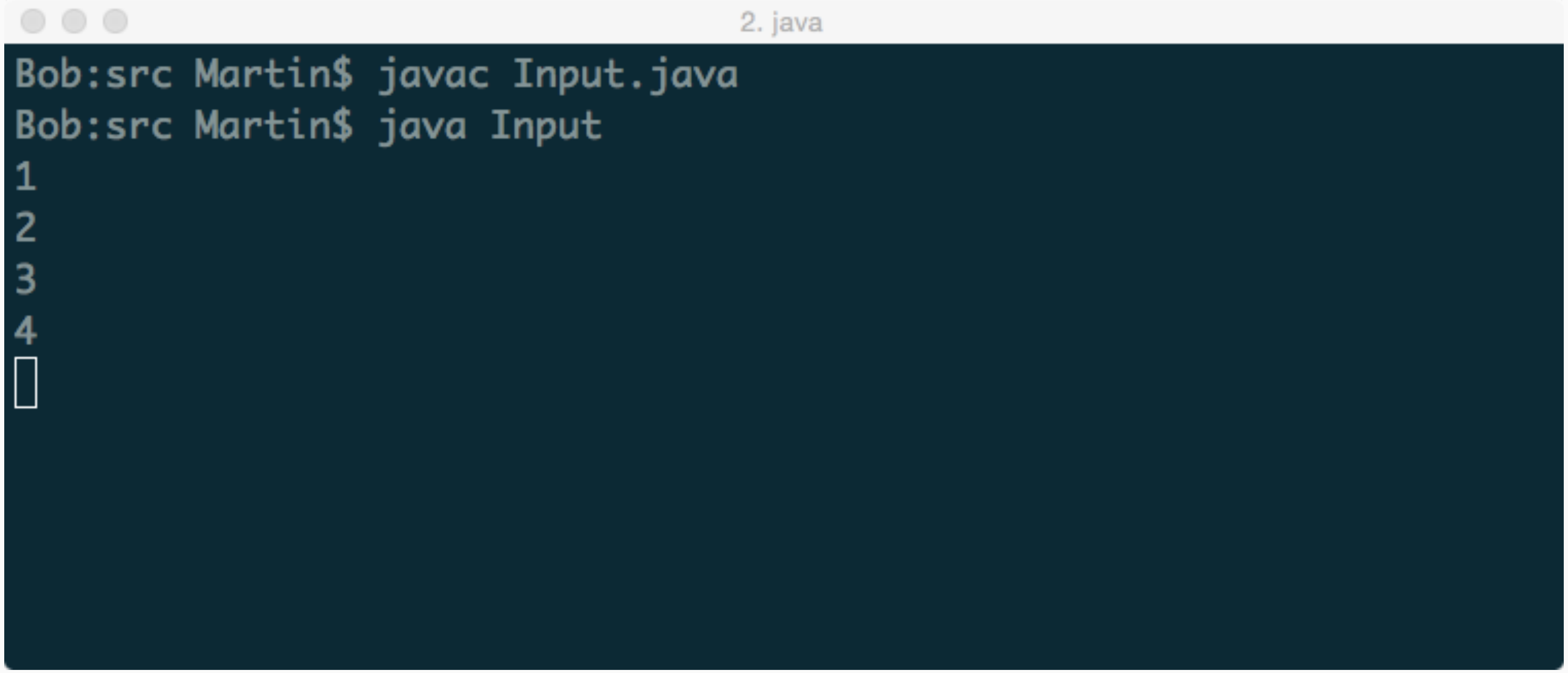
PARTIALLY FILLED ARRAYS (2)

```
import java.util.Scanner;

Scanner in = new Scanner(System.in);

int[] numbers = new int[100];

int elements = 0;

while ( in.hasNextInt() ) {

    numbers[ elements++ ] = in.nextInt();

}

in.close();
```

We define an additional variable to track the ***next free slot*** *in the array.*

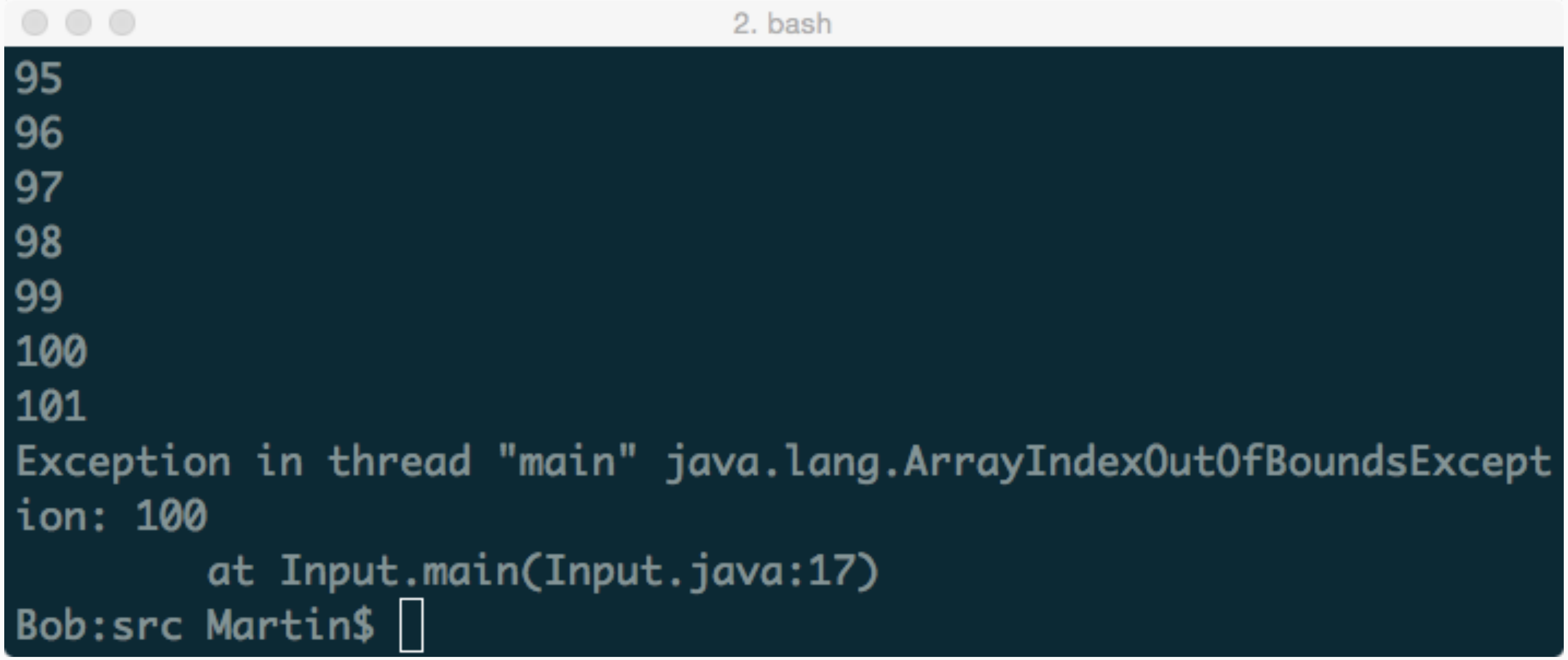
Every time we add an item to the array, we ***increment*** *this variable.*

PARTIALLY FILLED ARRAYS (3)



```
Bob:src Martin$ javac Input.java
Bob:src Martin$ java Input
1
2
3
4
```

If our user intends to enter a large amount of numbers...



```
95
96
97
98
99
100
101
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsExcept
ion: 100
        at Input.main(Input.java:17)
Bob:src Martin$
```

...then eventually the elements variable may cause problems.

PARTIALLY FILLED ARRAYS (4)

```
import java.util.Scanner;

Scanner in = new Scanner(System.in);

int[] numbers = new int[100];

int elements = 0;

while ( in.hasNextInt() && elements < numbers.length ) {

    numbers[ elements++ ] = in.nextInt();

}

in.close();
```

We can also use our element index variable to check whether it's possible to ***continue adding items*** *to the array.*

ACCESSING DATA IN A PARTIALLY FILLED ARRAY (1)

Once a user has finished inputting their numbers, we can examine what they have entered by again using our elements variable.

```
for ( int i = 0; i < elements; i++) {

    total += numbers[i];

}
```

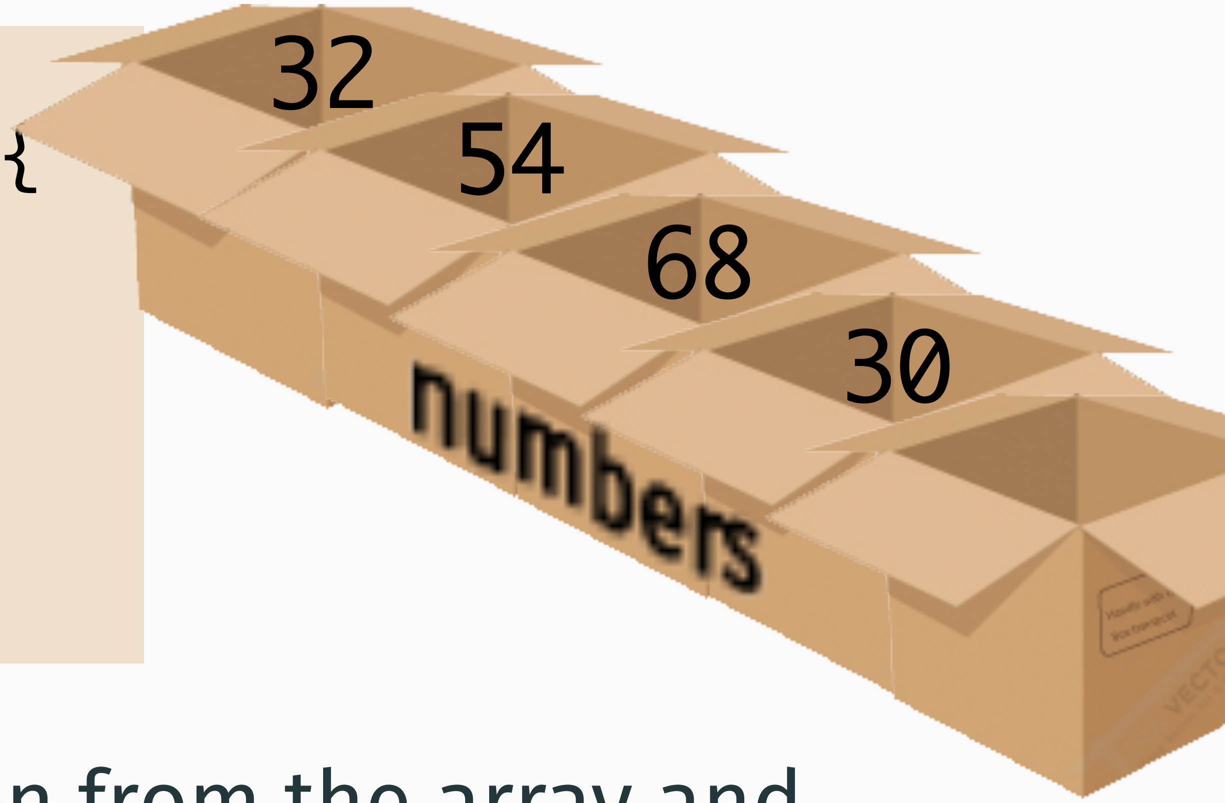
Length is no longer useful because it refers to the length of the array, not the number of items in it.

```
for ( int i = 0; i < elements; i++) {

    if ( numbers[i] > total / (double) elements )
                                 System.out.println("YES");

}
```

ACCESSING DATA IN A PARTIALLY FILLED ARRAY (2): FOR EACH (1)

There is an alternative here, something called the **for each** loop (originally mentioned in Topic 5):

```
for ( int number : numbers ) {

    total += number;

}
```



Each number is **automatically** taken from the array and placed into a temporary variable, which can be referenced in the loop. **Changes to this variable do not affect the array.**

The loop ends when all the items in the array have been examined.

ACCESSING DATA IN A PARTIALLY FILLED ARRAY (3)

```
for ( int number : numbers ) {

    total += number;

}
```

```
for ( int number : numbers ) {

    if ( number > total / (double) elements )
            System.out.println("YES");

}
```

I almost always use the for each loop.

- A relatively **new** construct, hence the existence of the more traditional index approach.

- Does limit **flexibility** slightly, given the **lack** of an index value.

ACCESSING DATA IN A PARTIALLY FILLED ARRAY (4)

```
int[] numbers = new int[100];
for ( int i = 0; i < 50; i++) {
    numbers[ i ] = i;
}
System.out.println(numbers[50]);
```

If we were to fill partially fill an array with a certain number of values, but then accidentally access an index that was **within** the size of the array, but not manually filled, what would happen?

- We would **not** be given an error, but instead a **zero** would be printed.
- Arrays of a **primitive** type contain **default values**.

ARRAYS VS. OBJECTS (1)

Recall that I also described objects as **multiple slot variables** in Topic 4, as I have with arrays, suggesting some **connection** between the two.

- The fact that the slots in an integer array are given default values, also suggests some connection to the **fields** of an object.
- As does the fact that the default values in an array, when an array is of a primitive type, are the **same** as those in the fields of an object, when those fields are of the same primitive type.

Arrays are indeed objects (albeit a special type).

ARRAYS VS. OBJECTS (2)

The array types themselves (e.g. int[], boolean[]) aren't **physical classes**, but instead **simulated** runtime classes provided to us by the JVM, that have certain properties that are similar to those demonstrated by actual classes.

- The use of the **new** command.

  ```
  int[] numbers = new int[4];
  ```

  - New copies to store **different** data.
- **Default field values** (mentioned).
- **Public** (immutable) **fields**.

  ```
  numbers.length
  ```

ARRAYS OF OBJECTS (1)

Recall that in Topic 4 we determined that primitive types can be **replaced** with class types.

This idea doesn't stop with arrays.

```
int[] numbers = new int[4];
```

```
private Dish[] starterMainDessert = new Dish[3];
```

We can, as we might expect, connect a set of objects in an array.

Indeed, we've had an example of this right under our noses for quite some time:

```
String[] args
```

REMEMBER: BACK TO MAIN

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

We now know a **even 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.
- We know that main has to be **visible** from outside the class, so that it can be run by the JVM (Topic 4).
- We know that main can be accessed without having to make an object of our driver class, which makes things **easier** for the JVM.

REMEMBER: BACK TO MAIN

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

We now know **everything** about **main**.

- We know that main can **never return** anything (where would it go anyway?)
- This is a **String array**, which is passed to the main method.
- We know that main has to be **visible** from outside the class, so that it can be run by the JVM (Topic 4).
- We know that main can be accessed without having to make an object of our driver class, which makes things **easier** for the JVM (Topic 6).

REMEMBER: USING THE COMPILER AND THE JAVA VIRTUAL MACHINE

When we use software such as a video converter to convert files, or we use a word processor, we **input our source files** (.mp3 file, .docx file) into the software using the **GUI**.

- We might right-click on the .docx file and select `open with Microsoft Word'.

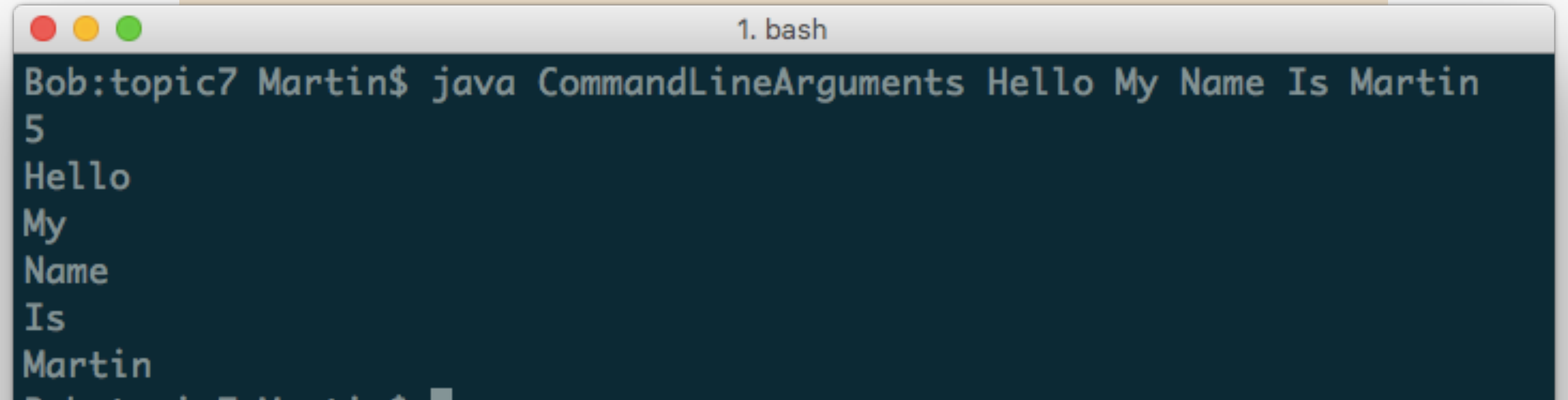
When we use software such as the Java compiler and the Java virtual machine to compile and run programs, we input our source files using a **terminal** (a **command line interface**).

- It is possible to run compiled (Java) programs from the GUI, but it requires a more complex compilation configuration.

COMMAND LINE ARGUMENTS (1)

Each **token** we write **after** the name of our compiled program when we pass it to the Java virtual machine, will be passed, token-by-token, to each **index** of the **args array**.

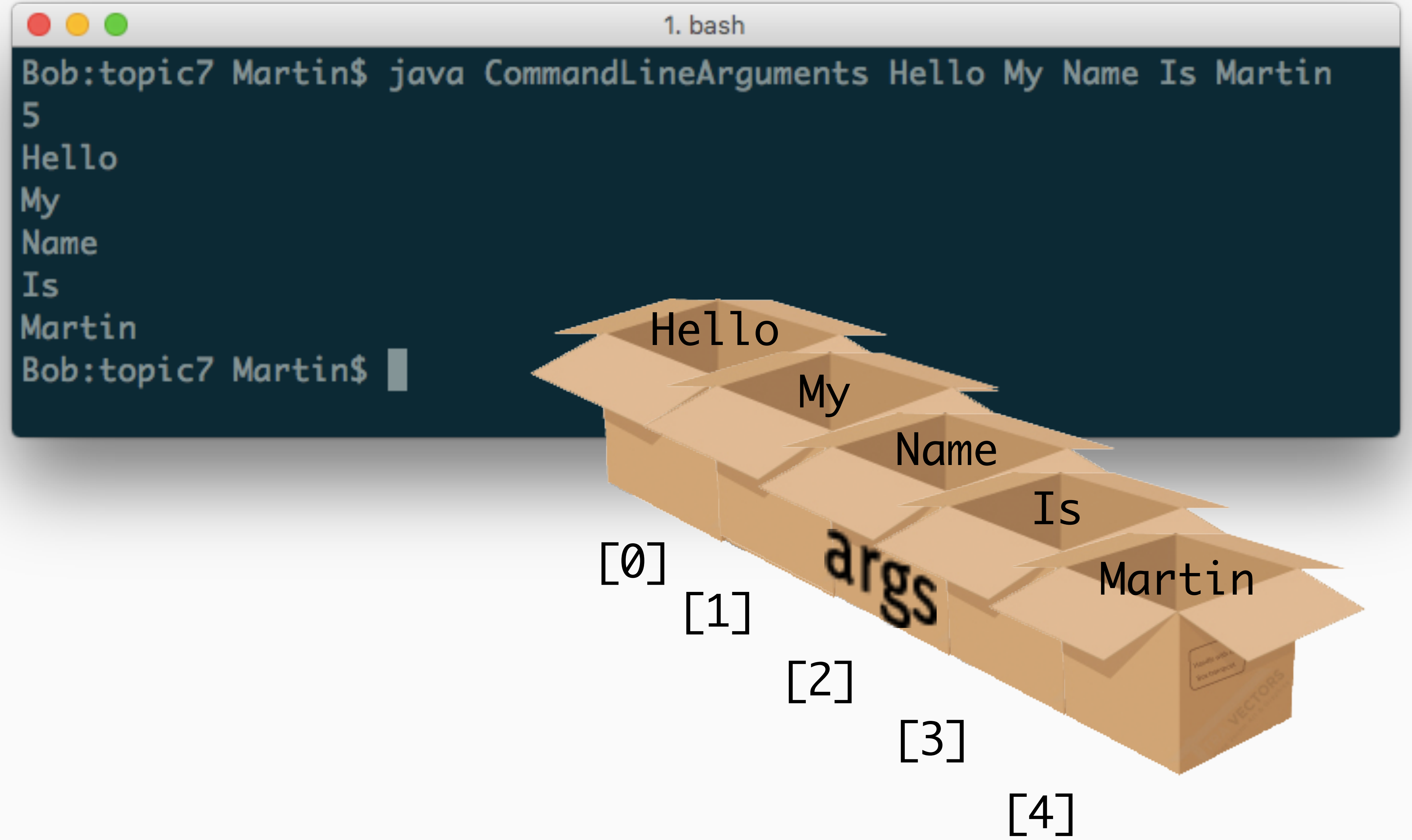
```java
public class CommandLineArguments {

  public static void main(String[] args) {

    System.out.println(args.length);

    for ( String argument: args ) {

      System.out.println(argument);
```



```
Bob:topic7 Martin$ java CommandLineArguments Hello My Name Is Martin
5
Hello
My
Name
Is
Martin
Bob:topic7 Martin$ 
```

COMMAND LINE ARGUMENTS (2)

Each **token** we write **after** the name of our compiled program when we pass it to the Java virtual machine, will be passed, token-by-token, to each **index** of the **args array**.



```
Bob:topic7 Martin$ java CommandLineArguments Hello My Name Is Martin
5
Hello
My
Name
Is
Martin
Bob:topic7 Martin$ 
```

COMMAND LINE ARGUMENTS (3)

A piece of data that we pass to a program via the command line is referred to as an **argument**.

- Hence the abbreviation **args** (although this can be changed).
- An argument, in the literal sense, is a piece of information from which another action may be inferred.
- We also pass arguments (data) to parameters when we call **methods**.

SCANNER VS. ARGS (1)

Command line arguments give us another way to take **user input.**

As we can only do this **once**, this style of input only really applies to simple programs, or to setting **flags** that affect the **overall operation** of the program.

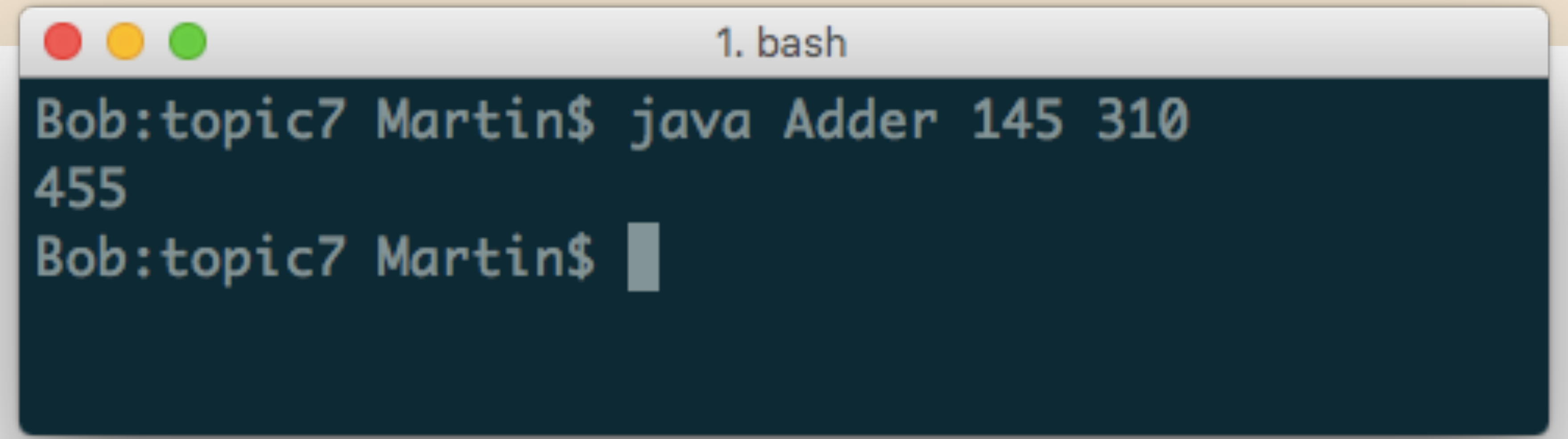
To take input more than once, and in an arguably more **user friendly** manner (with the inclusion of print statements), we need to use the Scanner class, as we saw in Topic 6.

SCANNER VS. ARGS (2): SIMPLE PROGRAMS

```
public class Adder {

  public static void main(String[] args) {

    if ( args.length == 2 ) {

      System.out.println(Integer.parseInt(args[0]) +
                                Integer.parseInt(args[1]));

    }

  }

}
```

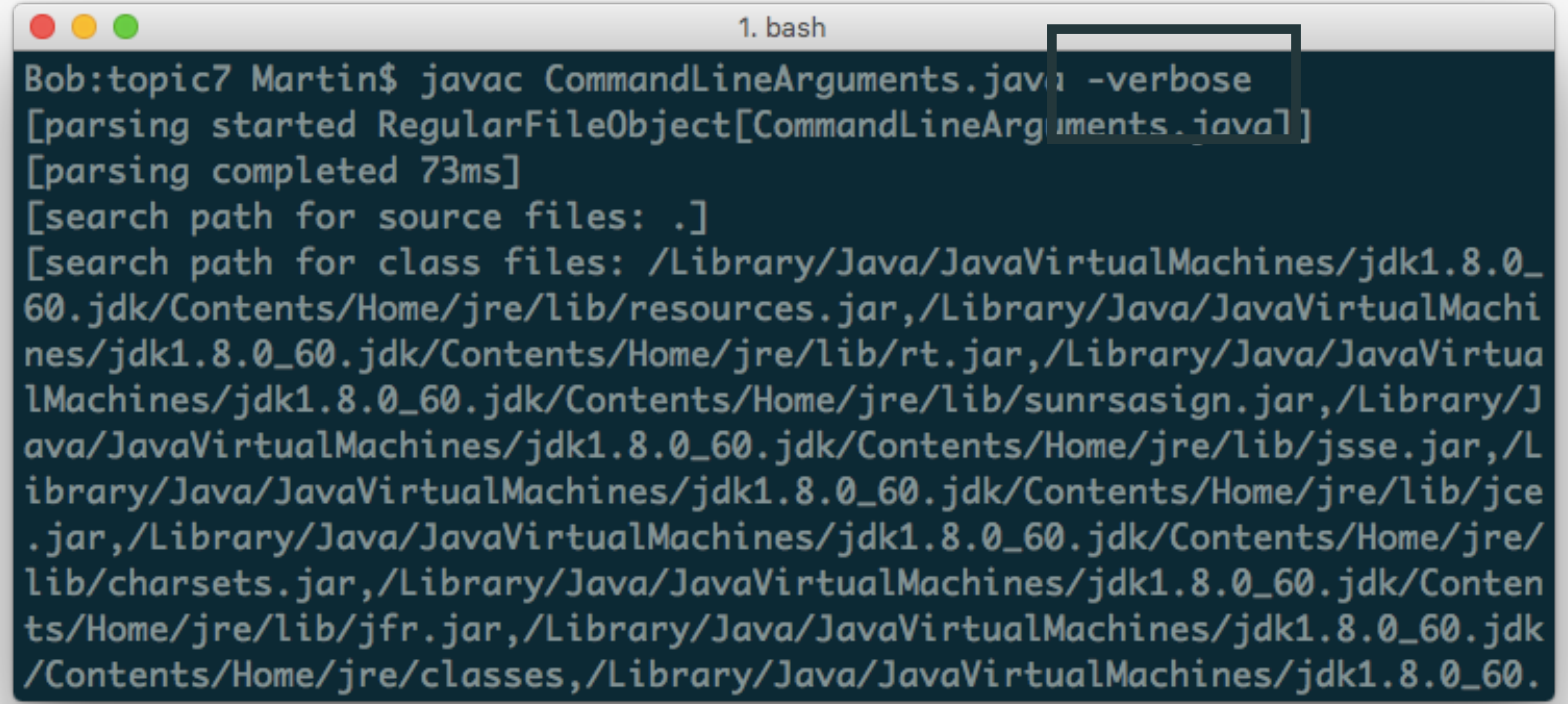
Even if we enter numbers into the command line, each token is always treated as a string.

Another Library class enabling the conversion of strings to integers. We will return to look at this class and this idea.



```
Bob:topic7 Martin$ java Adder 145 310
455
Bob:topic7 Martin$
```

SCANNER VS. ARGS (3): SETTING FLAGS



```
Bob:topic7 Martin$ javac CommandLineArguments.java -verbose
[parsing started RegularFileObject[CommandLineArguments.java]]
[parsing completed 73ms]
[search path for source files: .]
[search path for class files: /Library/Java/JavaVirtualMachines/jdk1.8.0_
60.jdk/Contents/Home/jre/lib/resources.jar,/Library/Java/JavaVirtualMachi
nes/jdk1.8.0_60.jdk/Contents/Home/jre/lib/rt.jar,/Library/Java/JavaVirtua
lMachines/jdk1.8.0_60.jdk/Contents/Home/jre/lib/sunrsasign.jar,/Library/J
ava/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/jre/lib/jsse.jar,/L
ibrary/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/jre/lib/jce
.jar,/Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/jre/
lib/charsets.jar,/Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Conten
ts/Home/jre/lib/jfr.jar,/Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk
/Contents/Home/jre/classes,/Library/Java/JavaVirtualMachines/jdk1.8.0_60.
```

REMEMBER: LECTURE EXERCISE: BLOCKCHAIN (4) - MINER AND BLOCKCHAIN

```
public class Miner {

   public static void main(String[] args) {

      Block myBlock = new Block();

      Blockchain chain = new Blockchain();

      chain.addBlock(myBlock);

   }

}
```

Back to arrays of objects...

```
public class Blockchain {

   Keep a list of blocks

   public void addBlock(Block block) {

      Add the block to a list

   }

}
```

How do we do this?

On to Topic 7!

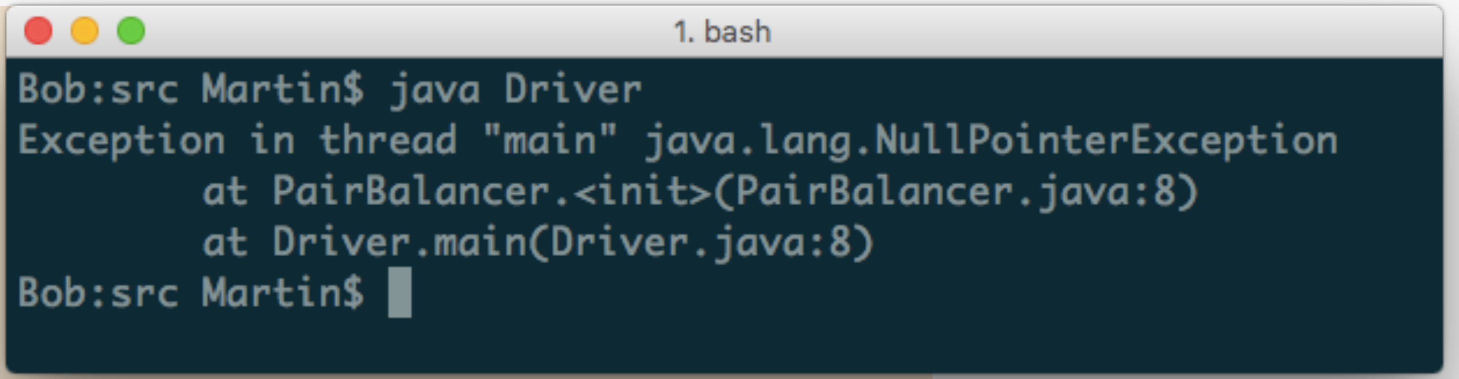
REMEMBER: LECTURE EXERCISE: BLOCKCHAIN (4) - MINER AND BLOCKCHAIN

```
public class Miner {

   public static void m

      Block myBlock = n

      Blockchain chain 

      chain.addBlock(my

   }

}
```

```
public class Blockchain {

   private Block[] chain;
   private int blocks;

   public Blockchain(int chainLength) {

      chain = new Block[chainLength];

   }

   public void addBlock(Block block) {

      chain[blocks++] = block;

   }

}
```

NULL (1)

```
public class PairBalancer

   private Pair toBalance;

   public PairBalancer(int valueA, int valueB) {

      toBalance.setValueA(valueA);
      toBalance.setValueB(valueB);

   }
```



```
Bob:src Martin$ java Driver
Exception in thread "main" java.lang.NullPointerException
        at PairBalancer.<init>(PairBalancer.java:8)
        at Driver.main(Driver.java:8)
Bob:src Martin$ 
```

Like accessing a variable of a class type when it is a field before it has had a copy of a class assigned to it, accessing an empty index in an array of a class type will result in a **NullPointerException** at **runtime**.

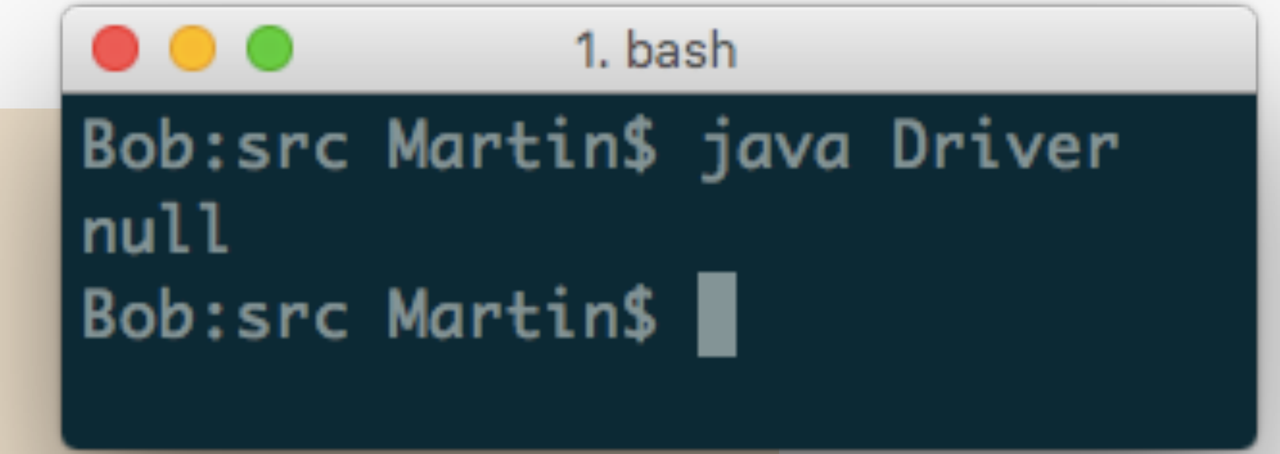
- We've already seen **InputMismatchExceptions** and **ArrayIndexOutOfBoundsExceptions**. We will formalise these observations in **Topic 10**.

NULL (2)

This error may seem to contradict the idea that fields (and the indices in an array) are given **default** values.

But in reality, there is a default value here, added in automatically for us:

```
public class PairBalancer {

   private Pair toBalance;

   public PairBalancer(int valueA, int valueB) {

      System.out.println(toBalance);

   }
```



```
Bob:src Martin$ java Driver
null
Bob:src Martin$
```

NULL (3)

The value **null** can be seen as a special **literal**, placed inside variables of a class type before they are assigned a copy of a class, and thus become objects (under our current understanding).

- Again, the **true** role of null will become clearer when we look at objects in memory.

We can use this literal to **protect against** errors:

```
if ( toBalance != null ) {

  toBalance.setValueA(valueA);
  toBalance.setValueB(valueB);

}
```

ASIDE: THE ARRAYS UTILITY CLASS

In the Java library, you’ll find a class called **Arrays**.

```
import java.util.Arrays;
```

The class has a number of useful static methods:

```
int[] hulk = new int[10];
int elements = 0;

for ( int i = 0; i < 100; i++ ) {

   if ( elements == hulk.length ) {

      hulk = Arrays.copyOf(hulk, hulk.length * 2);

   }

   hulk[elements++] = i;

}

System.out.println(Arrays.toString(hulk));
```

Double the size of the array as necessary (not particularly neat).

Print the content of the array. ***Open question****: what happens if we print the array directly?*

LECTURE EXERCISE: LARGEST NUMBER (1)

Given the array of numbers we had earlier:

```
int[] numbers = { 32, 54, 68, 30 };
```

Write a program that identifies the **largest** number in the list.

LECTURE EXERCISE: LARGEST NUMBER (2)

```
int largest = numbers[0];

for ( int i = 1; i < numbers.length; i++ ) {

     if ( numbers[i] > largest ) {

          largest = numbers[i];

     }

}

System.out.println("Largest: " + largest);
```

ARRAY EXERCISES

Some exercises involving arrays, for you to try in the laboratories:

- Write a method that sums all the values in an array.
- Fill an array, of size 50, with **random** numbers (between 1 and 100). Print out the contents of this array, 10 numbers per line, separating each number with a comma. Finish the whole sequence will a full stop.
- With the same array, replace the number in the second index with the number `1337’, and **shift** every other number up by one place.

Noughts And Crosses

IDENTIFYING CLASSES

We’ve already discussed, and seen lots of examples of, how code can be used to **model real world phenomena** (i.e. a set of objects).

Before we model something, we have to make a **decision** about the **classes** we want to **include** in our model.

- Typically these are the **nouns** pertinent to the problem.
- We will look more formally at this process **next semester**.

We then build these objects **individually**, before **combining** them to complete our model.

Let’s look at this process for the game of Noughts and Crosses.

CLASSES IN NOUGHTS AND CROSSES

For the game of noughts and crosses, I would propose that we look at the following classes:

Game (or Board)

Piece

Move

And a Main class.

FOLLOWING REQUIREMENTS (1)

Following **requirements**, or the **design decisions** made by someone else, is something you will now be familiar with from the **assignment** briefs.

We have a **solution in mind**, and are using these requirements to **guide** you towards it.

- Only way to structure the **collective** production of large programs (be this for an assignment, or as part of a lecture).
- Some connection with **industry**, where you will be building to a **specification**.
- Asks you to suspend the **why** slightly.

FOLLOWING REQUIREMENTS (2): VERSIONS

When we follow requirements during a lecture, to produce a solution together, we will do so with the aid of **offline versions**.

You will find these on KEATS.

```
/**
 * A class that represents A, B and C.
 *
 * Version 1.
 *
 * New in this version:
 *
 * 1. field D
 * 2. method E
 *
 * @author Martin
 *
 */
public class F {
```

- Show you **incremental snapshots** of the solution.
- Important if you are **lost** or **fall** behind (pickup from the **last version**).
- Only **new additions** are commented.

Versions 1 And 2: Piece

PIECE: REQUIREMENTS SUMMARY

```
public class Piece {
```

V1.1 Store whether a piece is a nought (if it isn't we know it's a cross).

V1.2 When I make a piece (when it's a new turn, and a piece appears), I want to know whether the last piece played was a cross. If it was, I know that I want this piece to be a nought.

V1.3 When I print a piece, I want to see whether it's a nought or a cross.

V2.1 When I compare one piece to another, I want to know whether it's a nought.

PIECE: REQUIREMENT V1.1

Store whether a piece is a nought (if it isn't we know it's a cross).

```java
public class Piece {

   private boolean isNought;

   public Piece(boolean isNought) {

      this.isNought = isNought;

   }

}
```

PIECE: REQUIREMENT V1.2

When I make a piece (when it’s a new turn, and a piece appears), I want to know whether the last piece played was a cross. If it was, I know that I want this piece to be a nought.

```
public Piece( Piece lastPiecePlayed ) {

    isNought = !lastPiecePlayed.isNought;

}
```

We effectively `flip’ the piece.

Version 1, Piece.java, Second constructor

REMEMBER: MODELLING A BANK ACCOUNT (7): TRANSFER

```java
public class BankAccount {

    private double balance;

    public void deposit(double amount) {

        balance = balance + amount;

    }

    public void printBalance() { … }

    public void withdraw(double amount) {

        balance = balance - amount;

    }

    public void transfer(BankAccount otherAccount, double amount) {

        withdraw(amount);
        otherAccount.deposit(amount);

    }

}
```

The notion of a method of a class accepting objects of that class itself should be a familiar one.

FLEXIBLE ENCAPSULATION

```
public Piece( Piece lastPiecePlayed ) {

    isNought = !lastPiecePlayed.isNought;

}
```

This is also why we can call private static fields with a class prefix from inside that class.

Looking at the way we've interacted with the isNought field here, we can see that we've called it directly through a Piece object, despite that field being **private.**

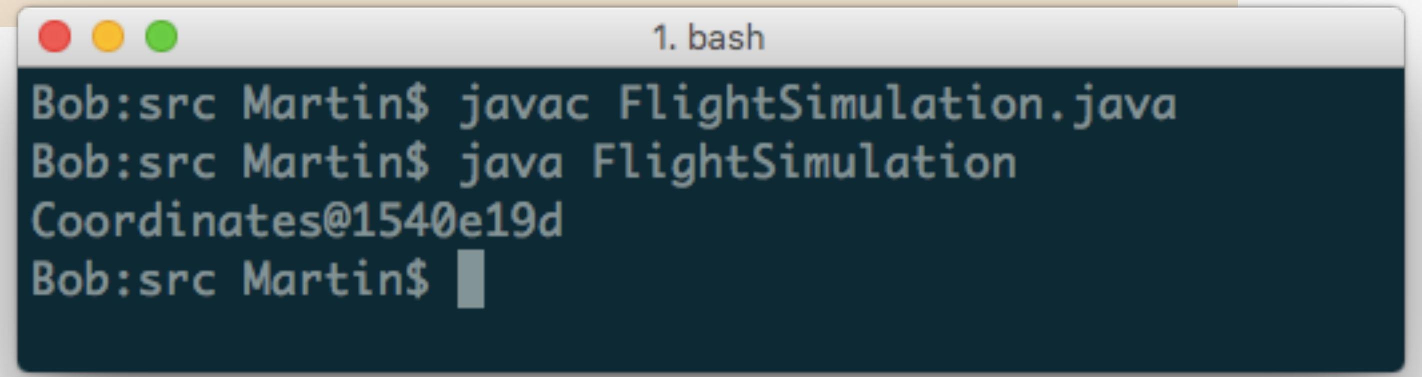
This may seem strange, given what we know about private fields, but is permitted by the compiler **when a class attempts to access a field of an object of itself**.

Because this interaction happens **inside the class itself**, this class is still in control of how the values in this field are manipulated, so **encapsulation** is **not** compromised.

PRINTING OBJECTS (1)

Some of you will have already experimented with what occurs if an object is **printed**.

```
Coordinates planeCoordinates = new Coordinates(10, 30);
System.out.println(planeCoordinates);
```



```
Bob:src Martin$ javac FlightSimulation.java
Bob:src Martin$ java FlightSimulation
Coordinates@1540e19d
Bob:src Martin$ 
```

If you print an object (including an **array**) directly, Java will print the **name** of that object's class, and a code derived from the object's **memory address** (more next semester).

However, it is quite **intuitive** to want to print objects, especially if they represent entities that **are themselves values** (such as a coordinate).

THE TOSTRING() METHOD (1)

Because it is intuitive to want to be able to print an object, and to get some information back from it by doing so, Java provides us with a way to **make our objects printable**.

That is, when you pass an object to a print statement, the JVM will **look for** a method called **toString** in that object, and will **call** it, effectively transforming the object into whatever is **returned** by the toString method.

THE TOSTRING() METHOD (2)

In the coordinates class:

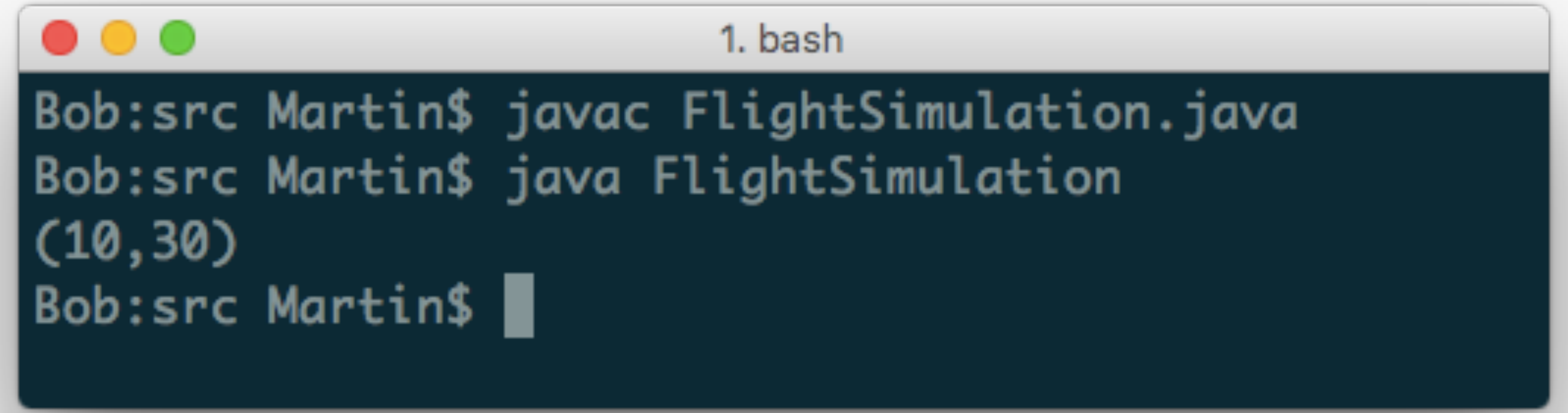
You **must** return a String.

You **cannot** supply any parameters.

```
public String toString() {

    return "(" + x + "," + y + ")";

}
```

In a driving class:

```
Coordinates planeCoordinates = new Coordinates(10, 30);
System.out.println(planeCoordinates);
```



```
Bob:src Martin$ javac FlightSimulation.java
Bob:src Martin$ java FlightSimulation
(10,30)
Bob:src Martin$ 
```

THE TOSTRING() METHOD (3)

Lots of open questions:

- How does the JVM know to look for a method called **toString**?
- Why are details of memory printed if we don’t add a toString method **ourselves**?
- What happens if we **don’t** return a String?
- What happens if we **add** parameters?

PIECE: REQUIREMENT V1.3

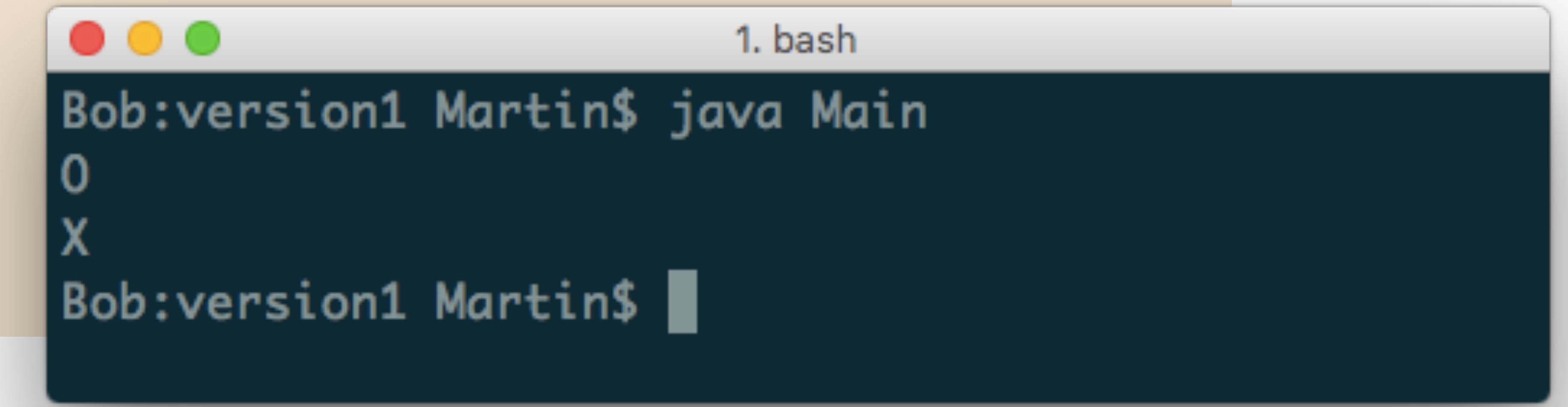
When I print a piece, I want to see whether it’s a nought or a cross.

```
public String toString() {
    if (isNought) {
        return "O";
    } else {
        return "X";
    }
}
```

Version 1, Piece.java, toString

VERSION 1: TESTING

```java
public class Main {

  public static void main(String[] args) {

    Piece pieceOne = new Piece(true);

    System.out.println(pieceOne);

    Piece pieceTwo = new Piece(pieceOne);

    System.out.println(pieceTwo);

  }

}
```



```
Bob:version1 Martin$ java Main
O
X
Bob:version1 Martin$ 
```

Version 1, Main.java, main method

PIECE: REQUIREMENT V2.1

When I compare one piece to another, I want to know whether it’s a nought.

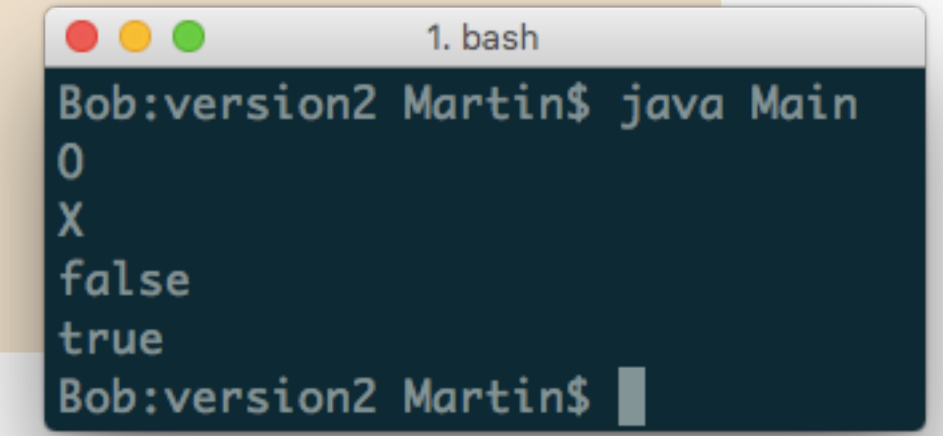
```
public boolean matches( Piece otherPiece ) {
    if ( otherPiece == null ) return false;
    return otherPiece.isNought == isNought;
}
```

If the other piece is null, we already know they can’t be equal.

Version 2, Piece.java, matches

VERSION 2: TESTING

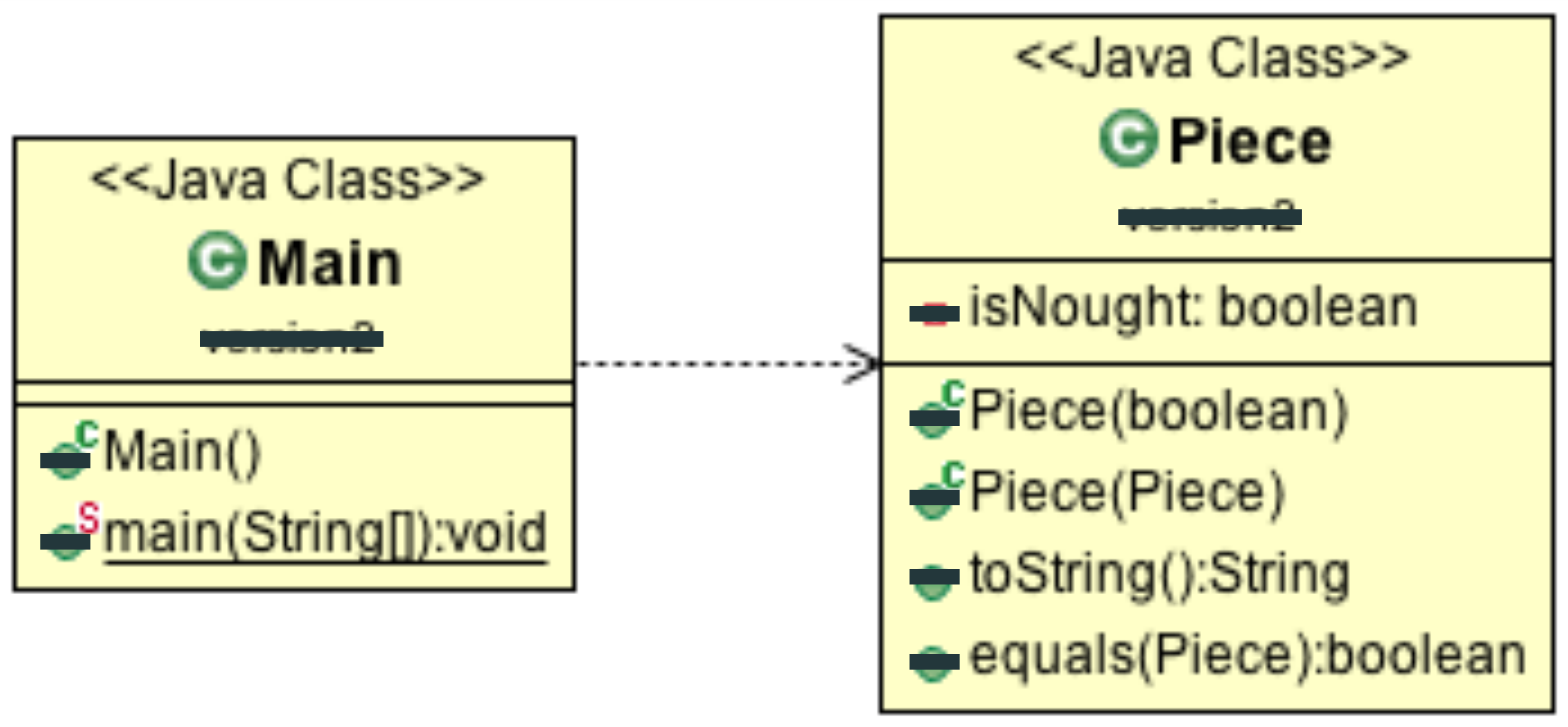
```java
public class Main {

   public static void main(String[] args) {

      Piece pieceOne = new Piece(true);

      System.out.println(pieceOne);

      Piece pieceTwo = new Piece(pieceOne);

      System.out.println(pieceTwo);

      System.out.println(pieceOne.matches(pieceTwo));

      Piece pieceThree = new Piece(true);

      System.out.println(pieceOne.matches(pieceThree));

   }

}
```



```
Bob:version2 Martin$ java Main
O
X
false
true
Bob:version2 Martin$
```

Version 2, Main.java, main method

PIECE, OVERVIEW.



<<Java Class>>

Main

[redacted]

Main()

main(String[]):void

<<Java Class>>

Piece

[redacted]

isNought: boolean

Piece(boolean)

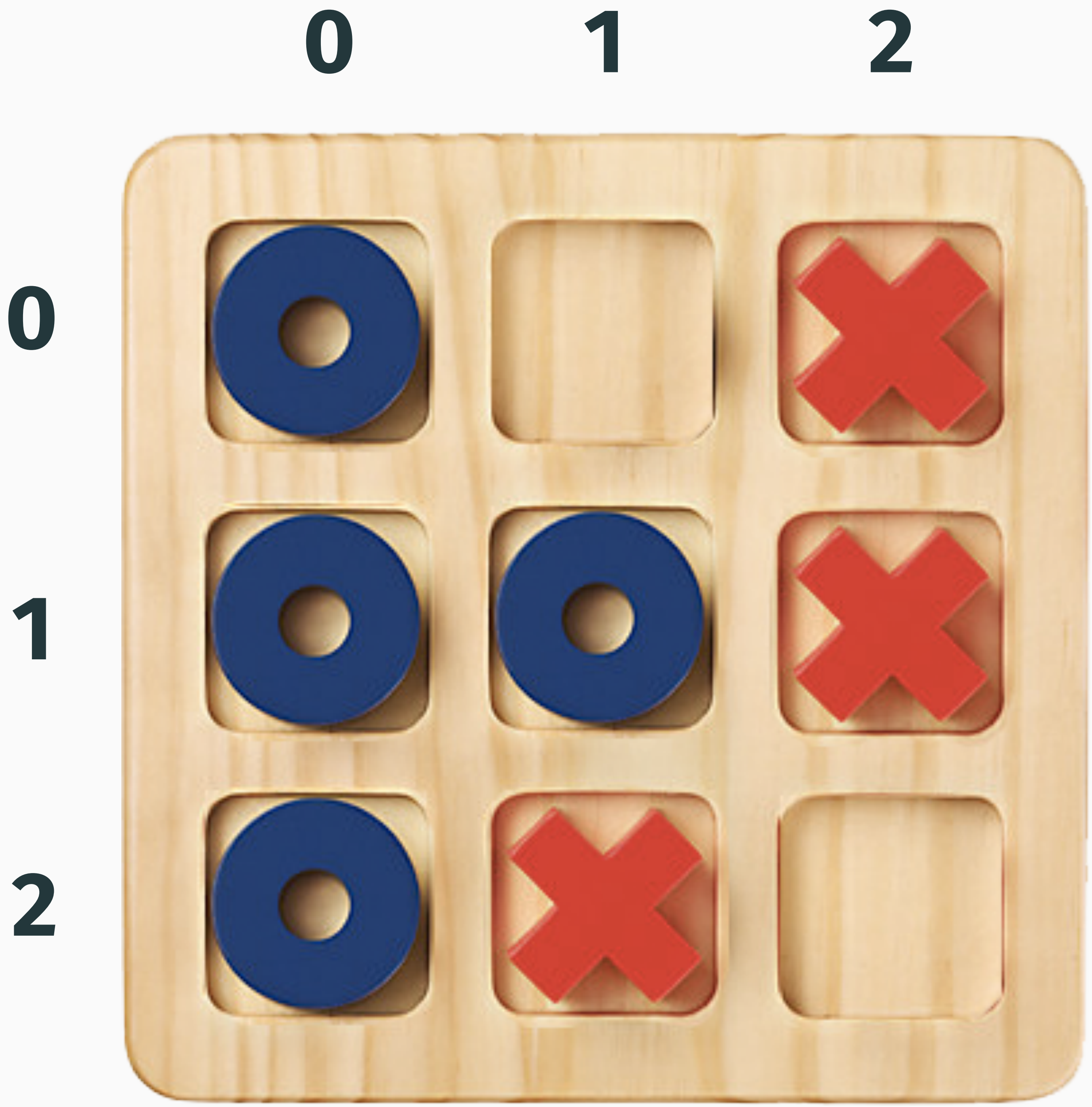
Piece(Piece)

toString():String

equals(Piece):boolean

Version 3: Piece And Board

PREFACE: MODELLING A 2D SPACE (1)



```
Piece[] row0 =
{ new Piece(true), null,
  new Piece(false) };
```

```
Piece[] row1 =
{ new Piece(true),
  new Piece(true),
  new Piece(false) };
```

```
Piece[] row2 =
{ new Piece(true),
  new Piece(false), null };
```

REMEMBER: INTERACTING WITH A CLASS COPY WITHOUT A VARIABLE

There are several other phenomena that show that we're simplifying things at this stage.

```
copyOfMartinPrinter.printMartin();
```

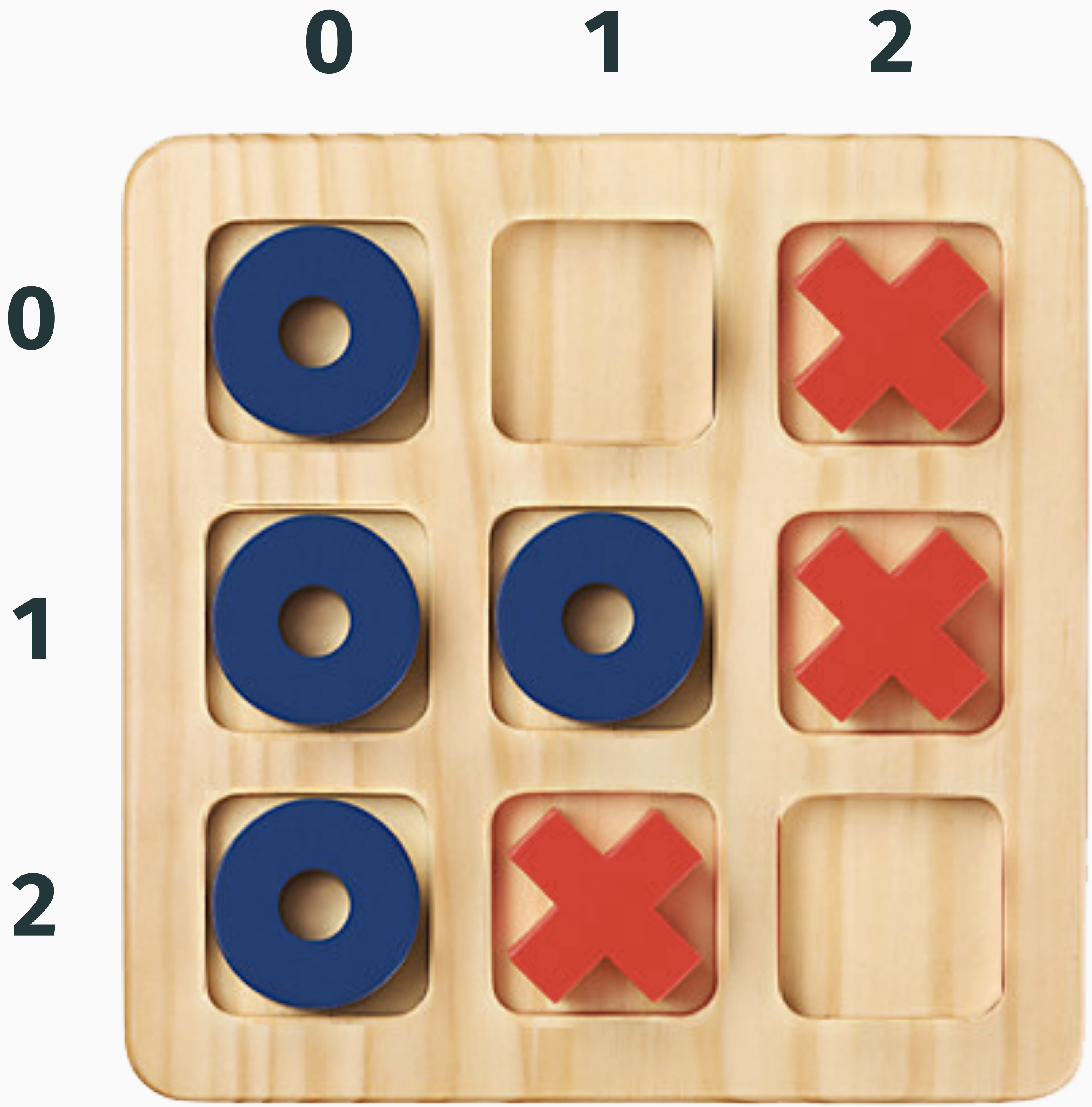
```
password.length()
```

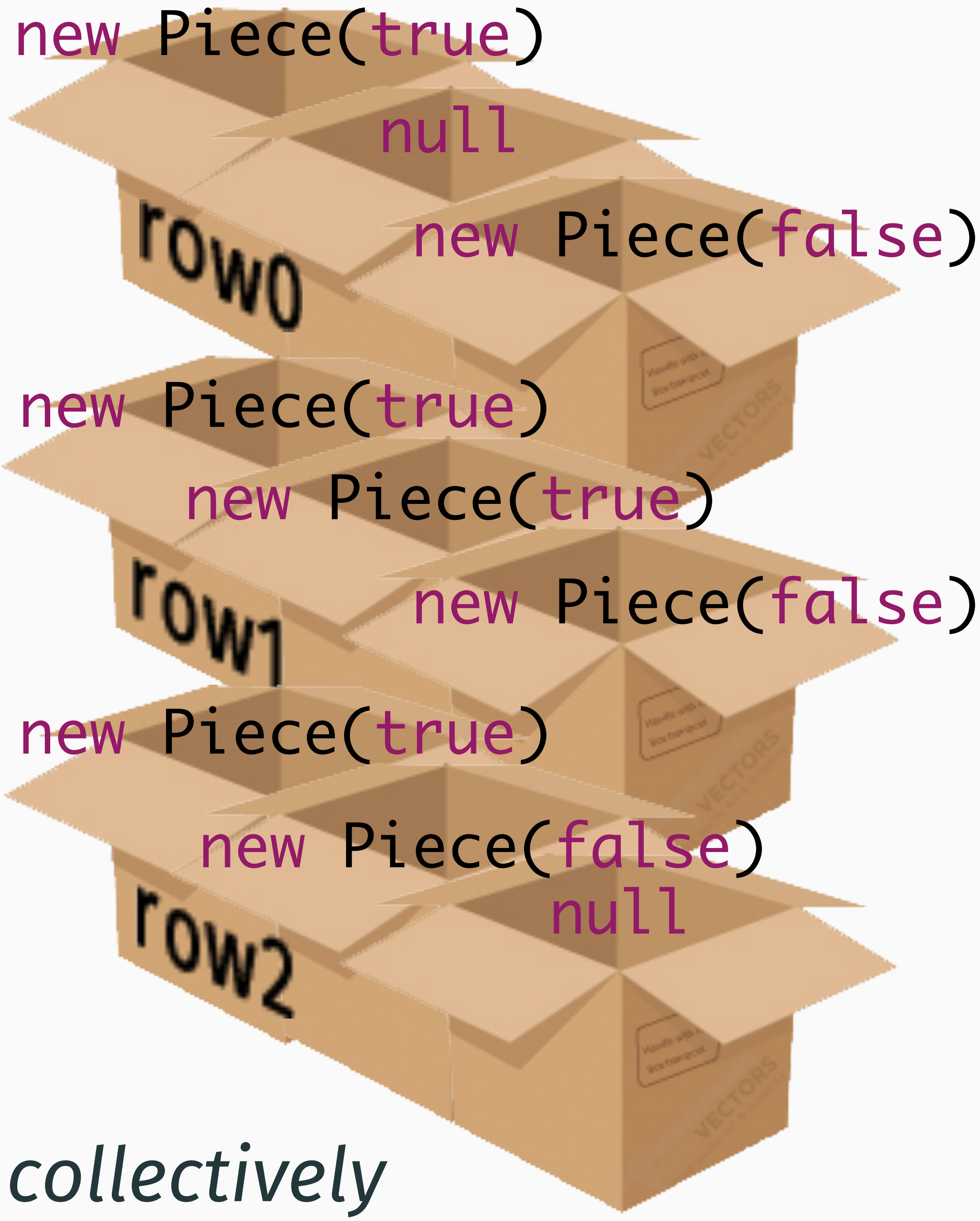
```
new MartinPrinter().printMartin();
```

```
"mypassword".length();
```

For example, we can interact with copies of a class without placing that copy into a variable.

PREFACE: MODELLING A 2D SPACE (2)

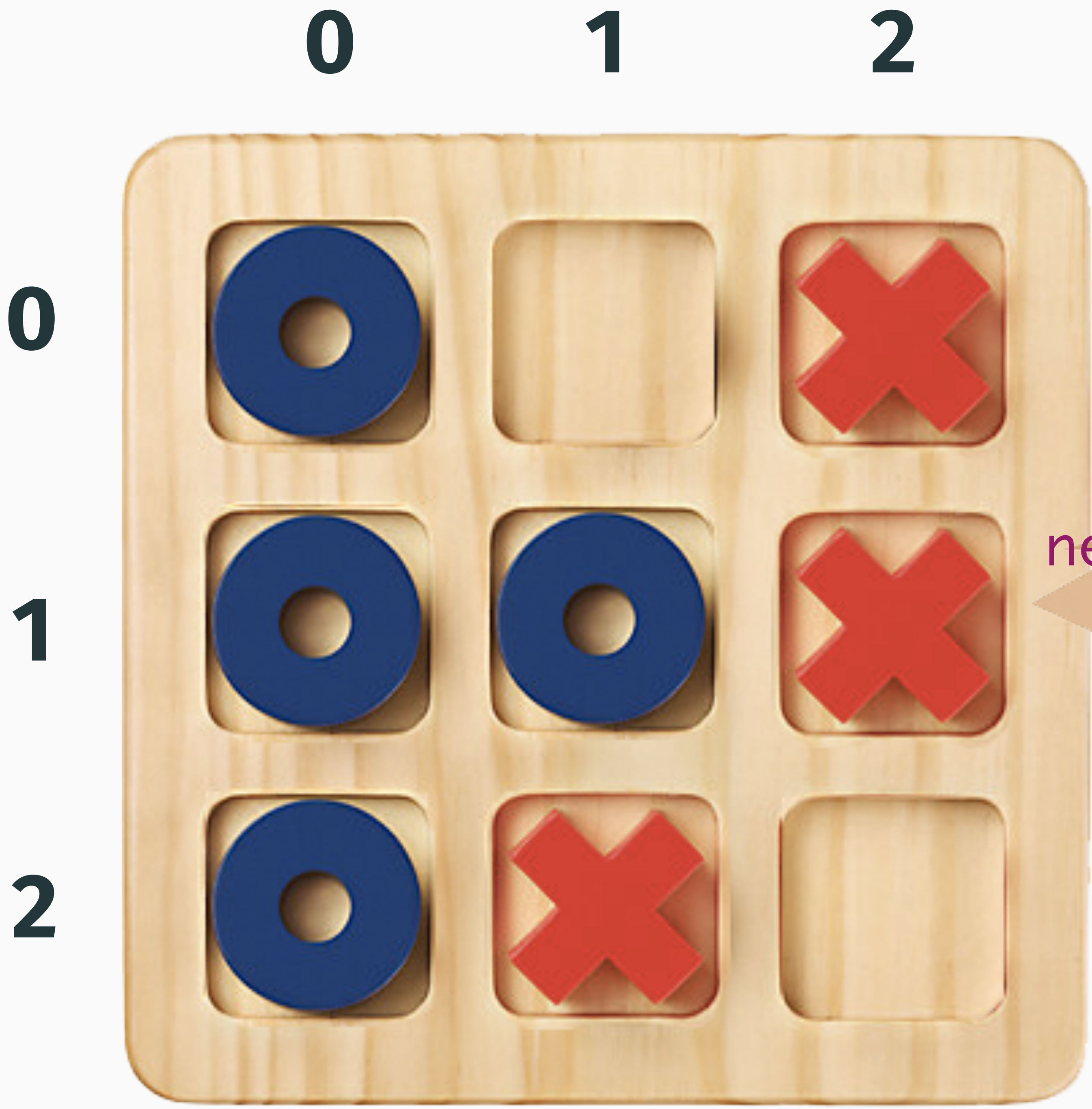


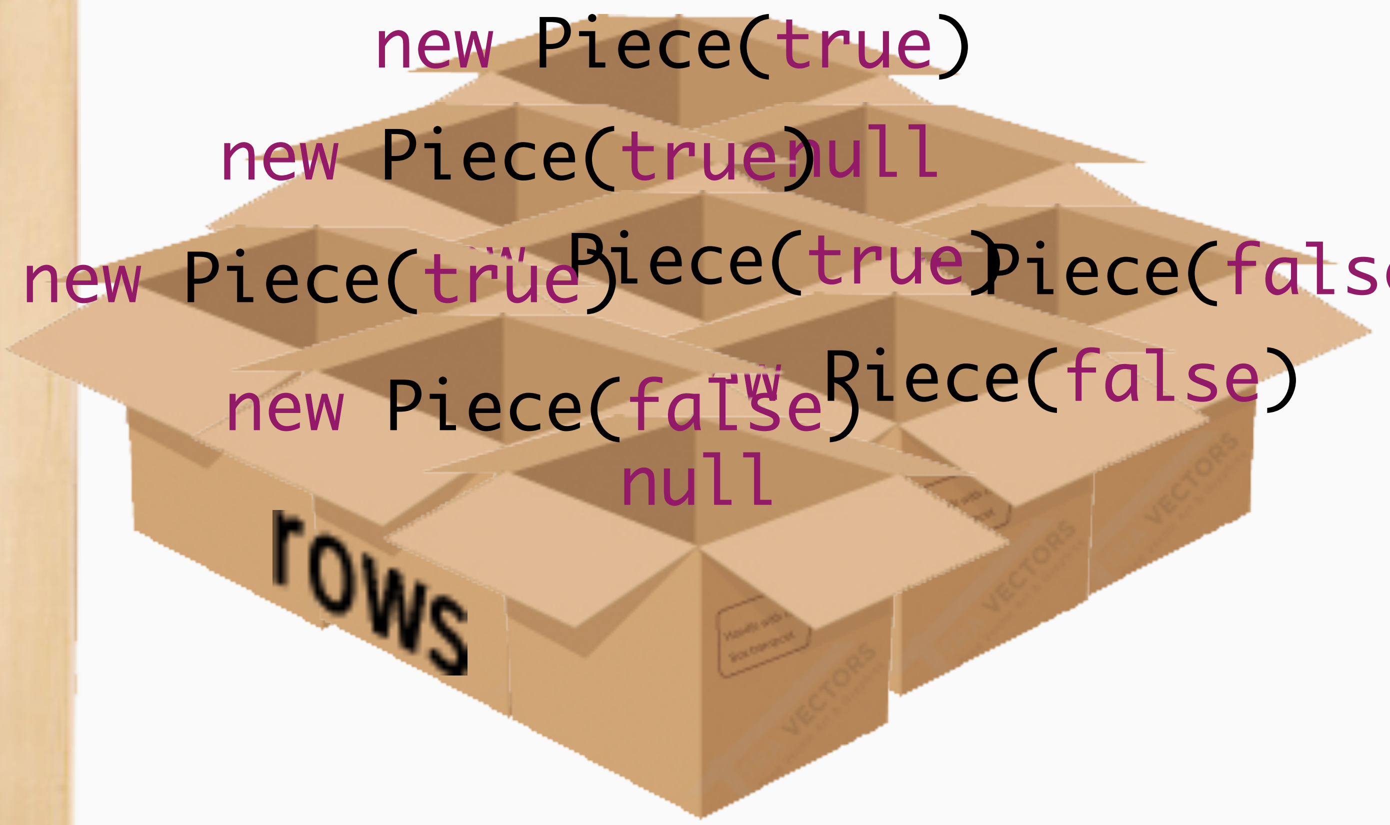


```
new Piece(true)
null
new Piece(false)
new Piece(true)
new Piece(true)
new Piece(false)
new Piece(true)
new Piece(false)
null
```

Values we want to deal with collectively are once again separated.

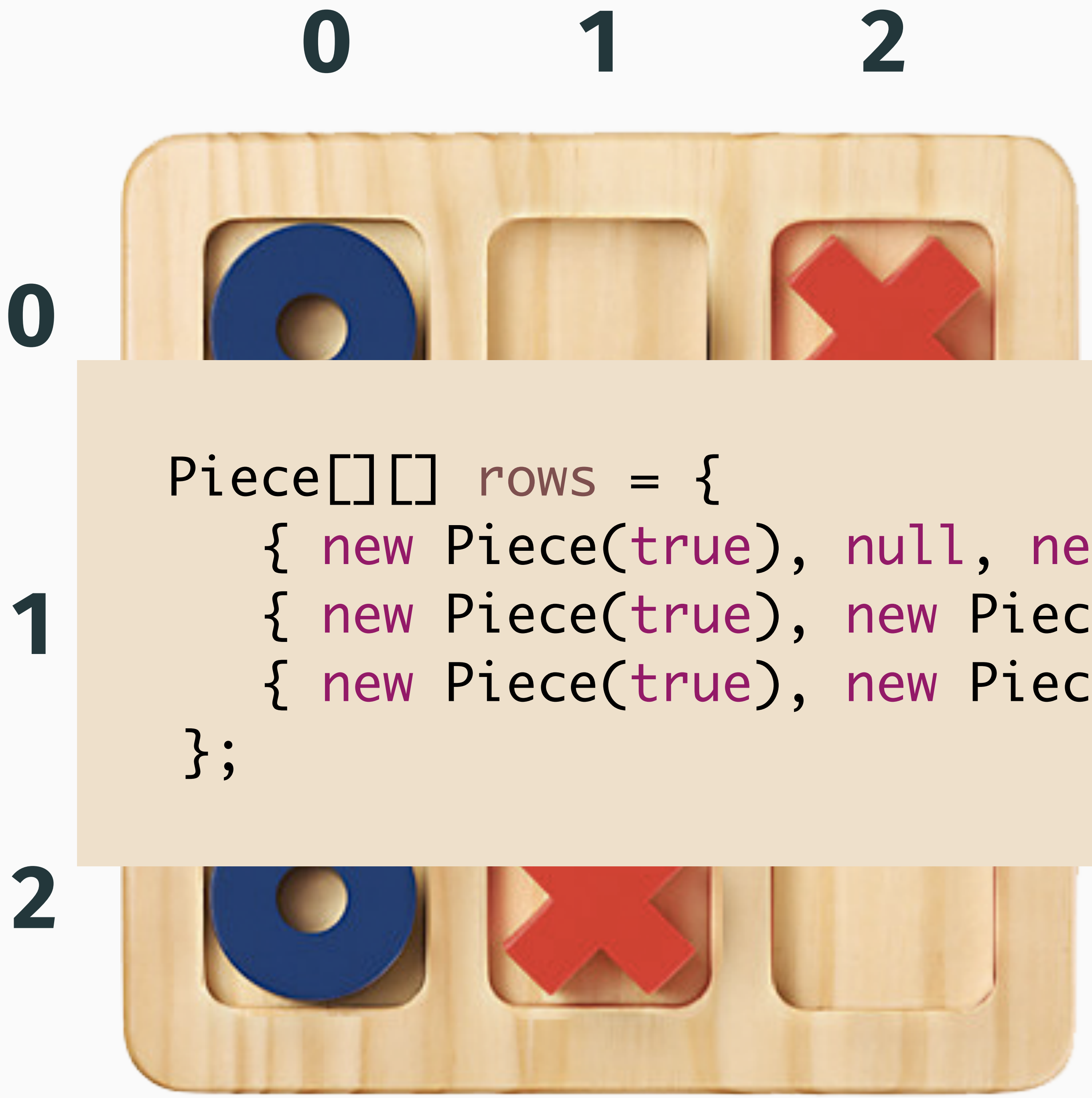
PREFACE: MODELLING A 2D SPACE (3)





Grouping things together among two dimensions.

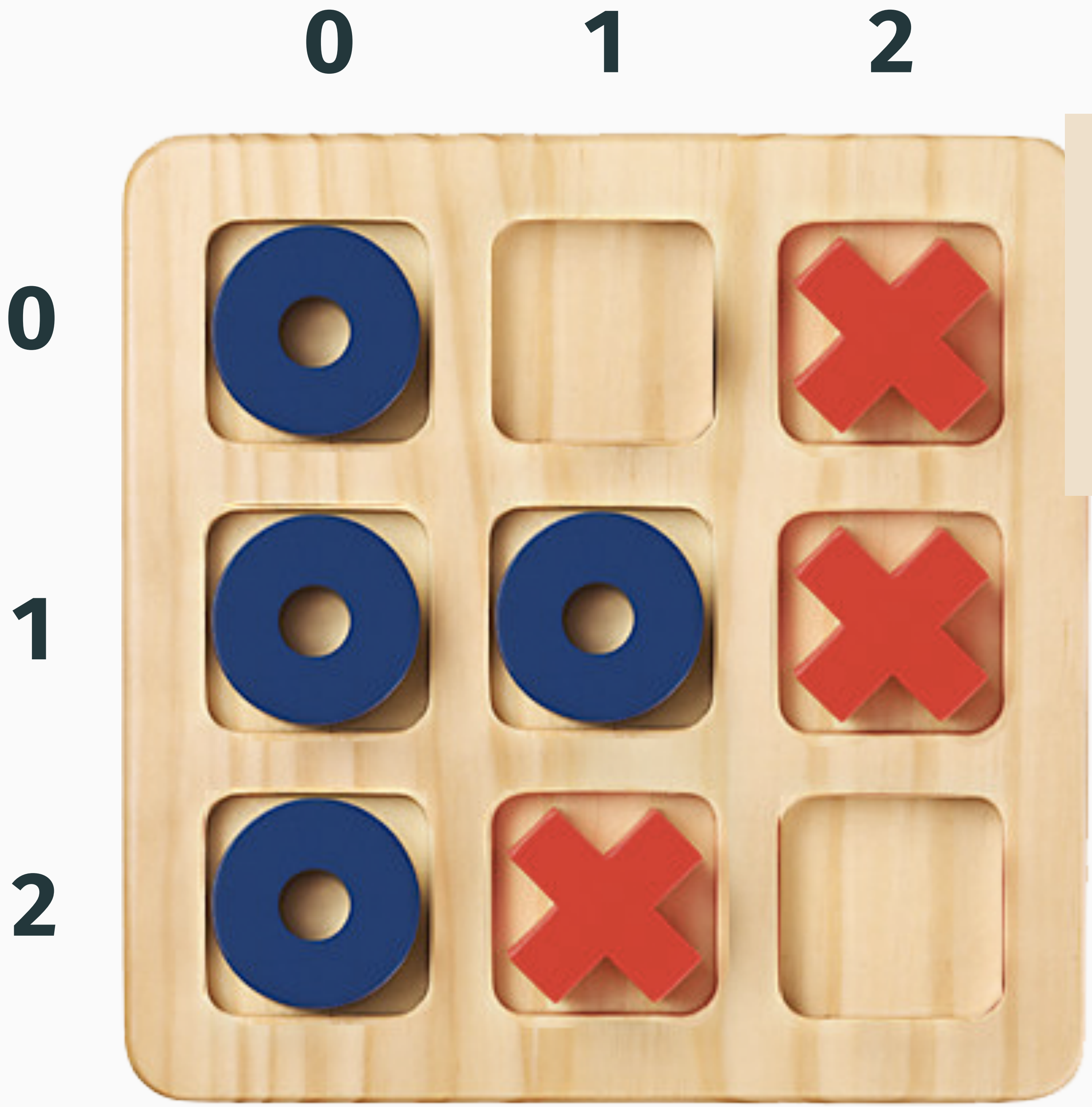
PREFACE: MODELLING A 2D SPACE (4)



```
Piece[][] rows = {
    { new Piece(true), null, new Piece(false) },
    { new Piece(true), new Piece(true), new Piece(false) },
    { new Piece(true), new Piece(false), null },
 };
```

One outer array, three inner arrays; each index of the outer array is itself an array; there arrays in an array with three positions.

PREFACE: MODELLING A 2D SPACE (5)



```
Piece[][] rows = new Piece[3][3];

rows[0][0] = new Piece(true);
rows[0][1] = null;
rows[0][2] = new Piece(false);
```

Row

Column

BOARD: REQUIREMENTS SUMMARY

```
public class Game {
```

V3.1 Create an empty board

V3.2 If I want to start a new game, the board should be cleared.

BOARD: REQUIREMENT V3.1

V3.1 Create an empty board.

```
public class Game {
  private Piece[][] board;
  public Game() {
    board = new Piece[3][3];
  }
```

Version 3, Game.java, constructor

BOARD: REQUIREMENT V3.2

V3.2 If I want to start a new game, the board should be cleared.

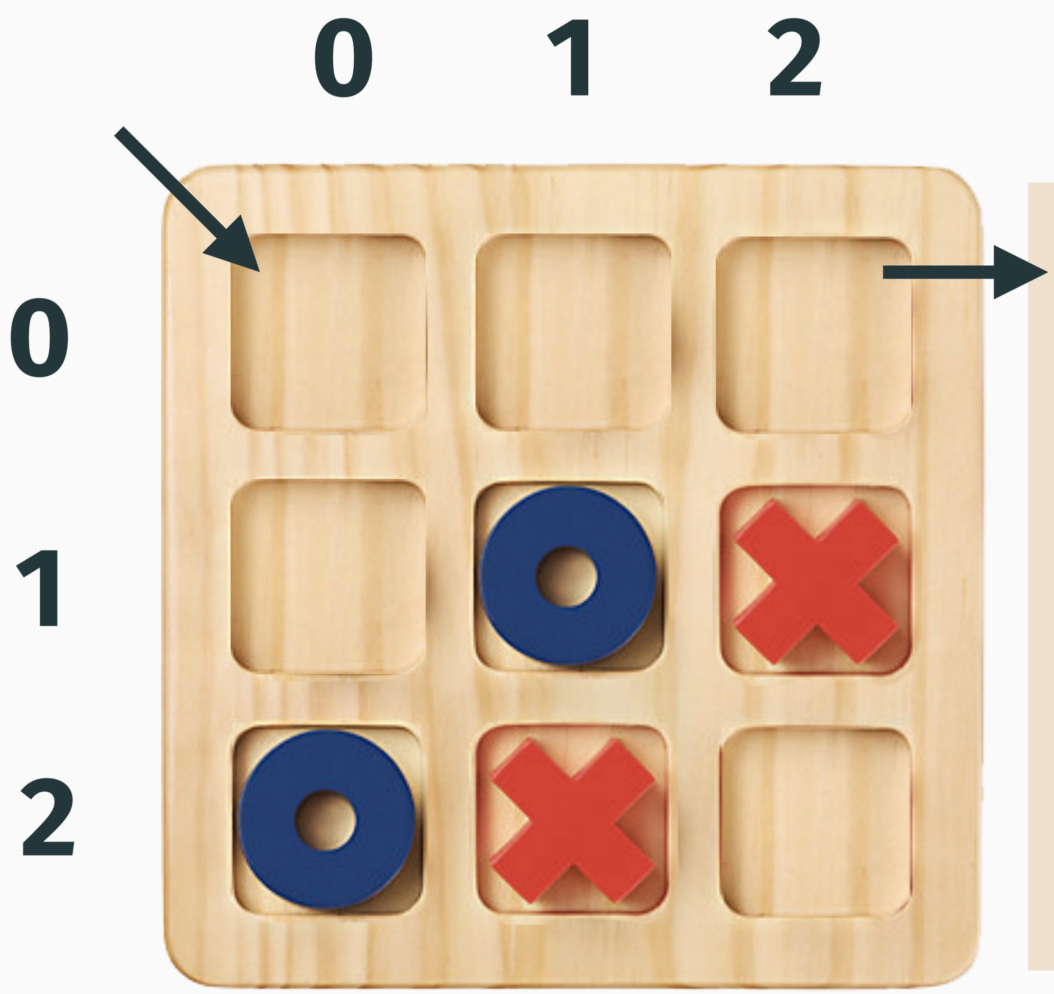
```
public void newGame() {

    for ( int row = 0; row < board.length; row++ ) {

        for ( int column = 0; column < board[0].length; column++ ) {

            board[row][column] = null;

        }

    }
}
```

The number of columns in the first row.

We could simply overwrite the board array with a new array, but this feels a little brute force.

Version 3, Game.java, newGame

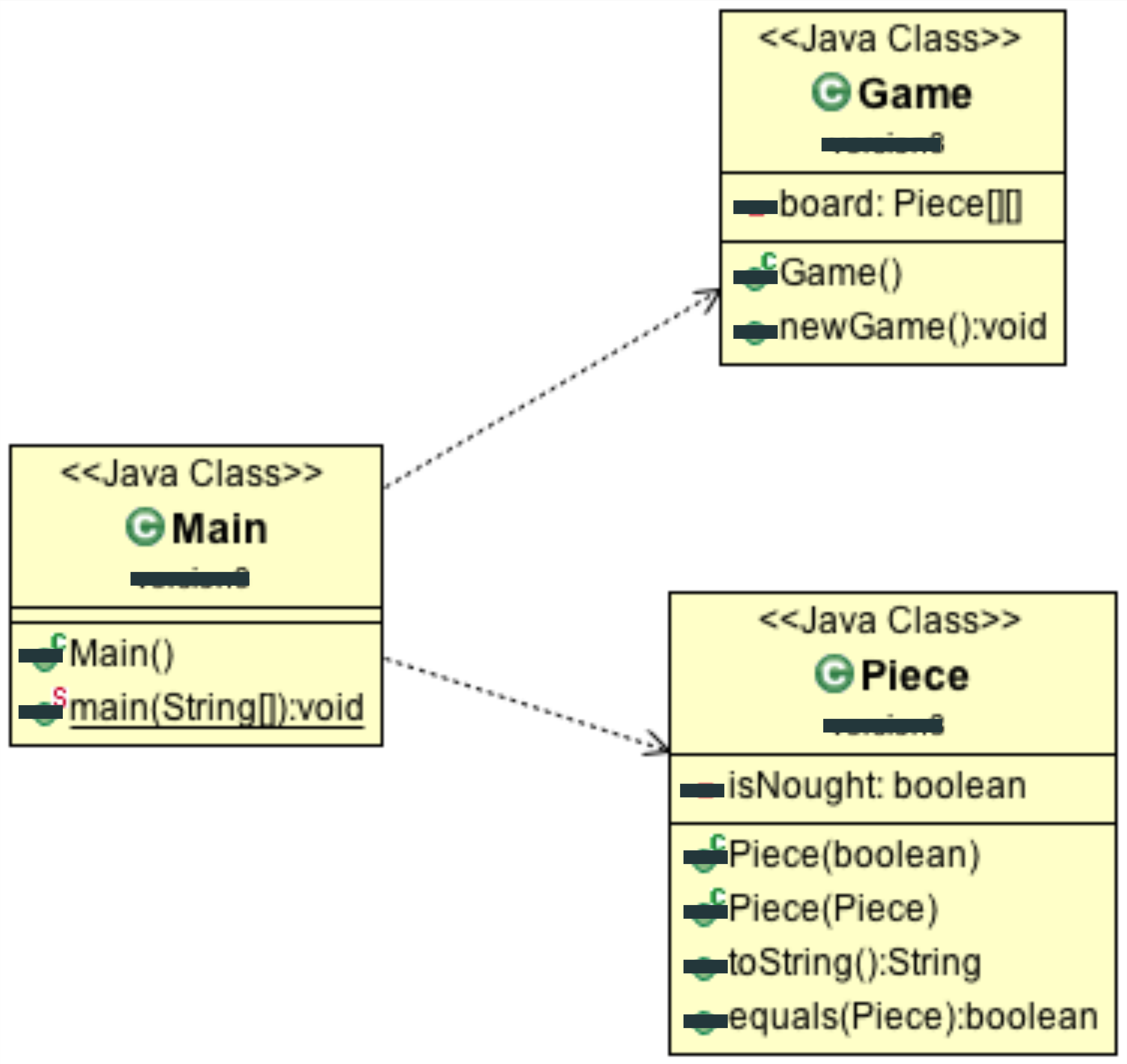
INTERACTING WITH A 2D ARRAY: NESTED LOOPS



```
for ( int row = 0; row < board.length; row++ ) {

    for ( int column = 0; column < board[0].length;
                                   column++ ) {

        board[row][column] = null;

    }

}
```

Value of row	Value of column	board[row][column]
0	0	board[0][0]
0 (Stays fixed)	1	board[0][1]
0 (Stays fixed)	2	board[0][2]
1 (Increments)	0 (Restarts)	board[1][0]

PIECE AND GAME, OVERVIEW.



<<Java Class>>
Game
[redacted]

[redacted] board: Piece[][]

[redacted] Game()
[redacted] newGame():void

<<Java Class>>
Main
[redacted]

[redacted] Main()
[redacted] main(String[]):void

<<Java Class>>
Piece
[redacted]

[redacted] isNought: boolean

[redacted] Piece(boolean)
[redacted] Piece(Piece)
[redacted] toString():String
[redacted] equals(Piece):boolean

THE PROCESS OF PROBLEM SOLVING

We do not tackle the modelling problem **directly**.

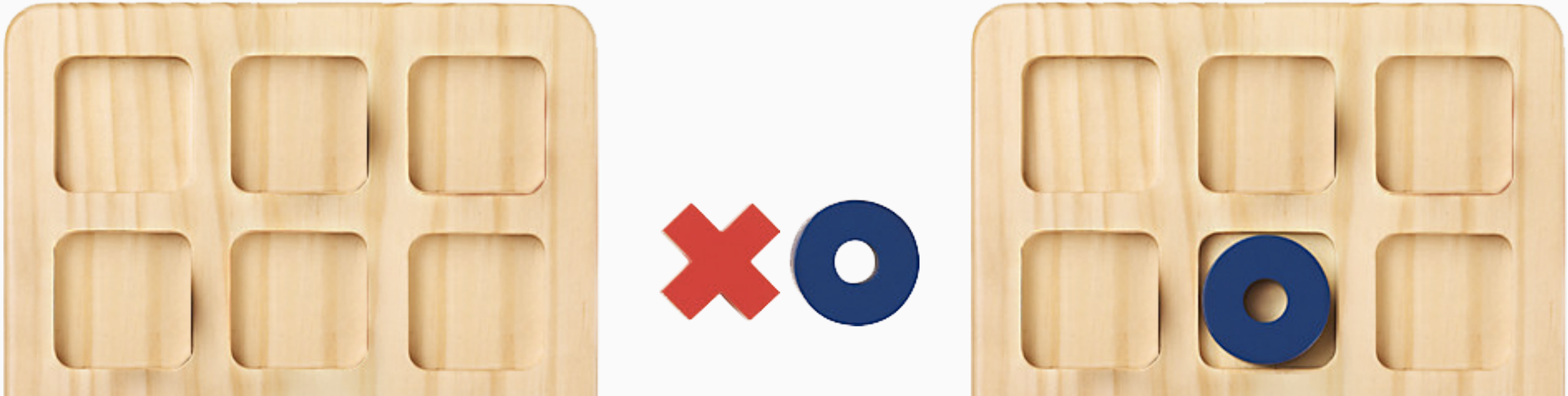
We identify **individual objects first**, and then use them as the **building blocks** for our **complete program**.

We solve **what we can at first** while **ignoring** other parts, and then **return later**. We keep **iterating** and **refining** the solution.

This process cannot be taught. It is learnt by **practise**.

There is no content in the remaining requirements that will be directly assessed. But, you are encouraged to follow the remainder of the Noughts and Crosses exercise, to indirectly prepare yourself for future coursework tasks, and the examination.

Version 4: Move



PIECE: REQUIREMENTS SUMMARY

```
public class Move {
```

V4.1 Store the column and the row in which the move took place.

V4.2 Provide the ability to return the row and return the column.

V4.3 Check whether the row and column in the move are valid (i.e. on the board).

PIECE: REQUIREMENT 4.1

Store the column and the row in which the move took place.

```java
public class Move {

  private int row;
  private int column;

  public Move(int row, int column) {

    this.row = row;
    this.column = column;

  }
```

Version 4, Move.java, constructor

PIECE: REQUIREMENT 4.2

Provide the ability to return the row and return the column.

```
public int getRow() {
    return row;
}
public int getColumn() {
    return column;
}
```

Version 4, Move.java, getRow and getColumn

PIECE: REQUIREMENT 4.3

Check whether the row and column in the move are valid (i.e. on the board).

```
public boolean isValid() {

   return row >= 0 && row < 3 && column >=0 && column < 3;

 }
```

The position of the piece has to be on the board in order for this move to be valid.

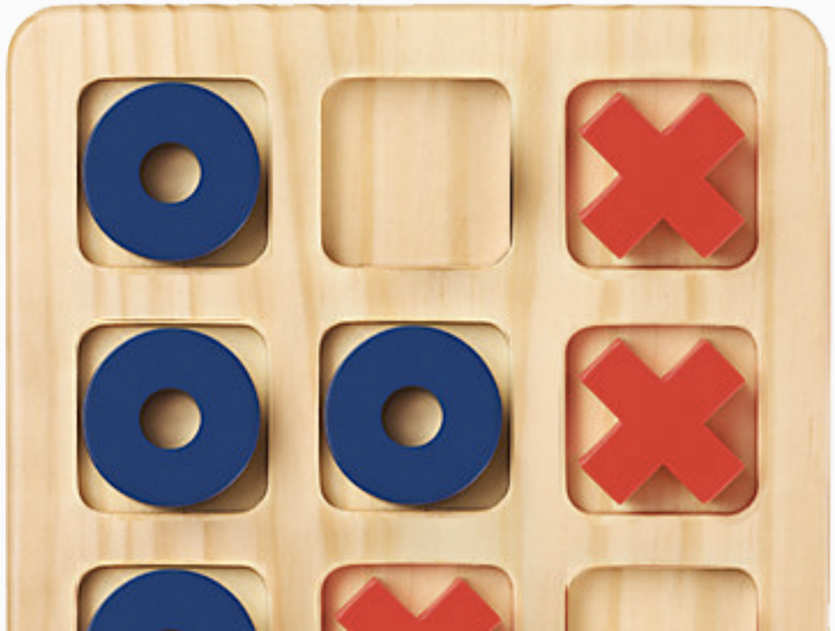
Version 4, Move.java, isValid

VERSION 4: TESTING

```java
public static void main(String[] args) {
    Move move1 = new Move(1, 1);
    System.out.println(move1.isValid());
    Move move2 = new Move(2, 1);
    System.out.println(move2.isValid());
    Move move3 = new Move(100, 100);
    System.out.println(move3.isValid());
}
```

Version 4, Main.java, main

Version 5: Playing The Game



PLAYING THE GAME: REQUIREMENTS

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

    Game g = new Game();

    Piece piece = new Piece(true);

    Move firstMove = new Move(1, 1);

}
```

If I want to play a piece in the middle square of the board, this is the **expressivity** I have to do so, currently.

Really though I want to be able to **add** a piece to the board (**V5.1**).

Version 5, Main.java, main

PLAYING THE GAME: REQUIREMENT 5.1

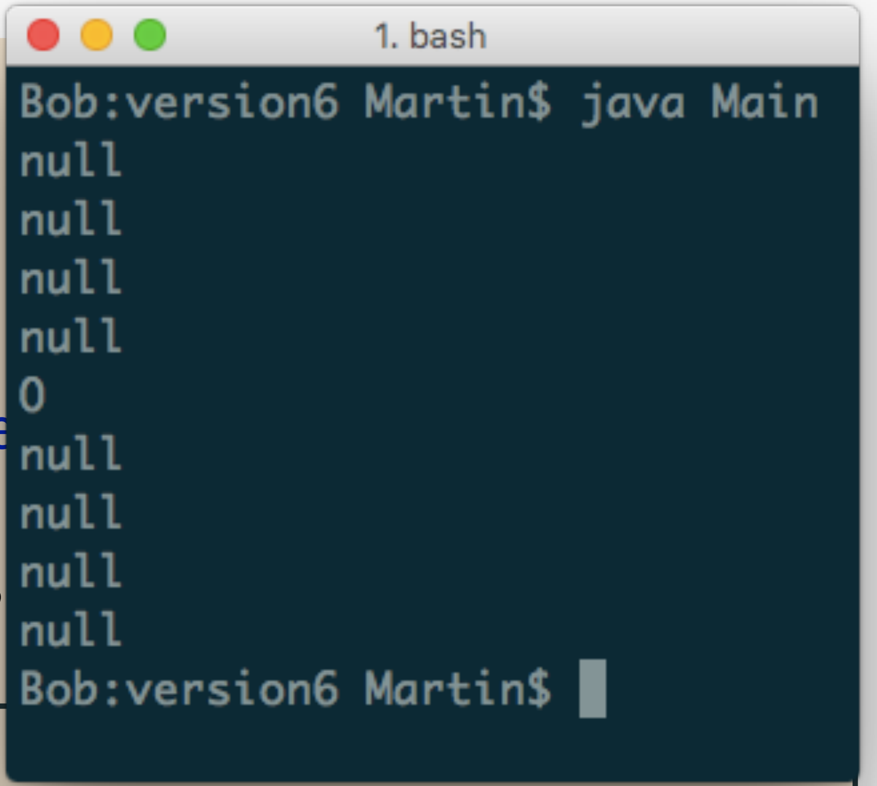
Supply a method that, given a move and a piece, makes that move by adding the piece to the board.

```
public void play(Piece piece, Move move) {

    board[move.getRow()][move.getColumn()] = piece;

}
```

Version 5, Game.java, play

VERSION 5: TESTING

```
public void printBoard() {

  for ( int row = 0; row < board.length; row++

    for ( int column = 0; column < board[0].le

      System.out.println(board[row][column]);

    }

  }

}
```

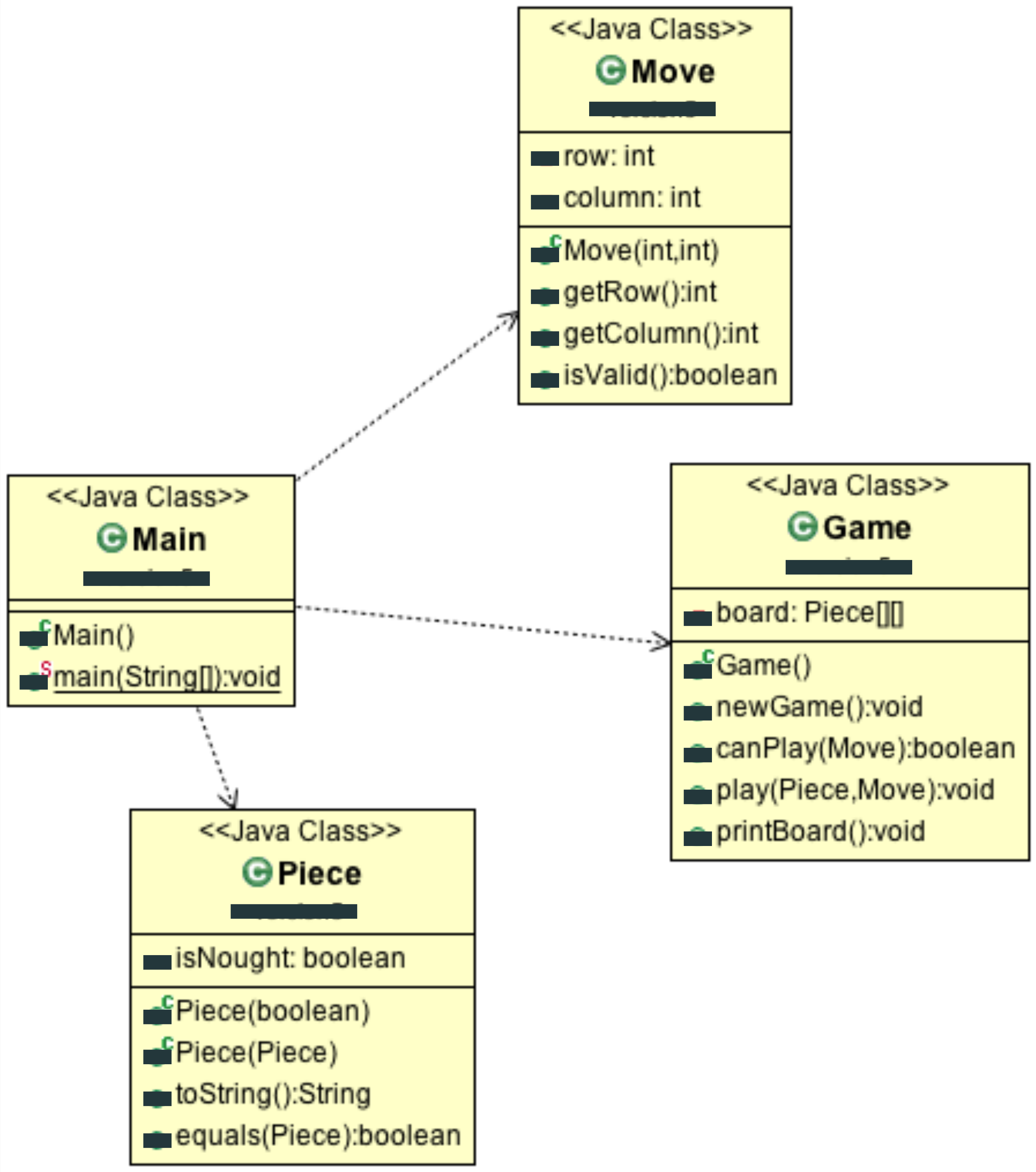


```
Bob:version6 Martin$ java Main
null
null
null
null
0
null
null
null
null
Bob:version6 Martin$ 
```

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

    Game game = new Game();

    Piece piece = new Piece(true);

    Move firstMove = new Move(1, 1);

    game.play(piece, firstMove);

}
```

Version 5, Game.java printBoard and Main.java, main

PIECE, GAME AND MOVE, OVERVIEW.



<<Java Class>>
Move
[redacted]

row: int
column: int

Move(int,int)
getRow():int
getColumn():int
isValid():boolean

<<Java Class>>
Main
[redacted]

Main()
main(String[]):void

<<Java Class>>
Game
[redacted]

board: Piece[][]

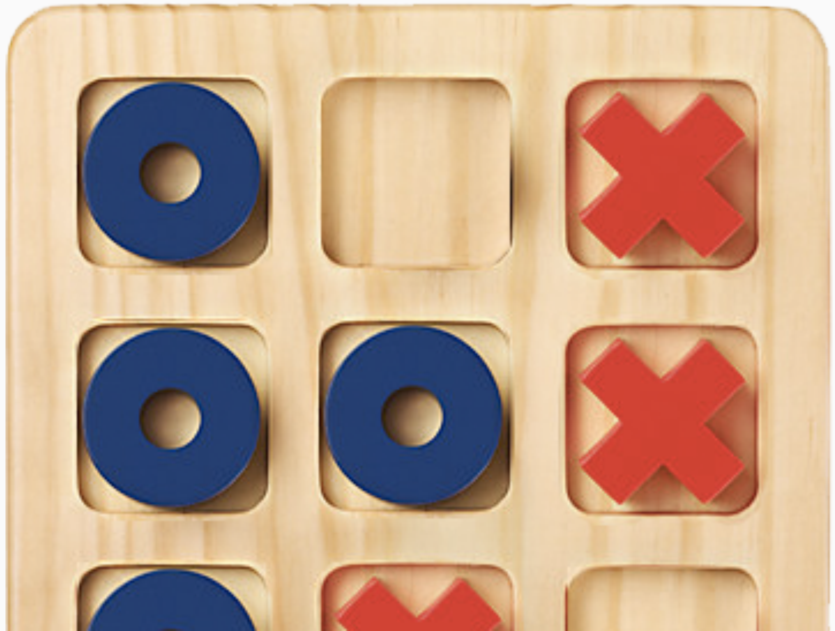
Game()
newGame():void
canPlay(Move):boolean
play(Piece,Move):void
printBoard():void

<<Java Class>>
Piece
[redacted]

isNought: boolean

Piece(boolean)
Piece(Piece)
toString():String
equals(Piece):boolean

Version 6: Rules Of The Game



RULES OF THE GAME: REQUIREMENTS

We don’t want players to be able to add pieces to the board as they wish.

Instead, this should be subject to **restriction**.

I should only be able to play a piece if the position I want to fill is **on the board** (Done: **V4.3**) and if the position I want to fill is **empty** (New: **V6.1**).

RULES OF THE GAME: REQUIREMENT 6.1 (1)

Check whether a position on the board is empty.

```
public boolean canPlay(Move m) {

  return board[m.getRow()][m.getColumn()] == null;

}
```

Given a move, this method will tell me whether the row and column in that move refer to an empty position on the board.

Version 6, Game.java, canPlay

PLAYING THE GAME: REQUIREMENT 6.1 (2)

I should only be able to play a piece if the position I want to fill is **on the board** and if the position I want to fill is **empty**.

```
public boolean play(Piece piece, Move move) {

   if ( move.isValid() && canPlay(move) ) {

      board[move.getRow()][move.getColumn()] = piece;

      return true;

   } else {

      return false;

   }

}
```

Version 6, Game.java, play

VERSION 6: TESTING

```java
public class Main {

    public static void main(String[] args) {

        Game game = new Game();

        Piece piece = new Piece(true);

        Move firstMove = new Move(1, 1);

        System.out.println(game.play(piece, firstMove));

    }

}
```

Version 6, Main.java, main

Version 7: Involving The User

INVOLVING THE USER: CURRENT TIMELINE

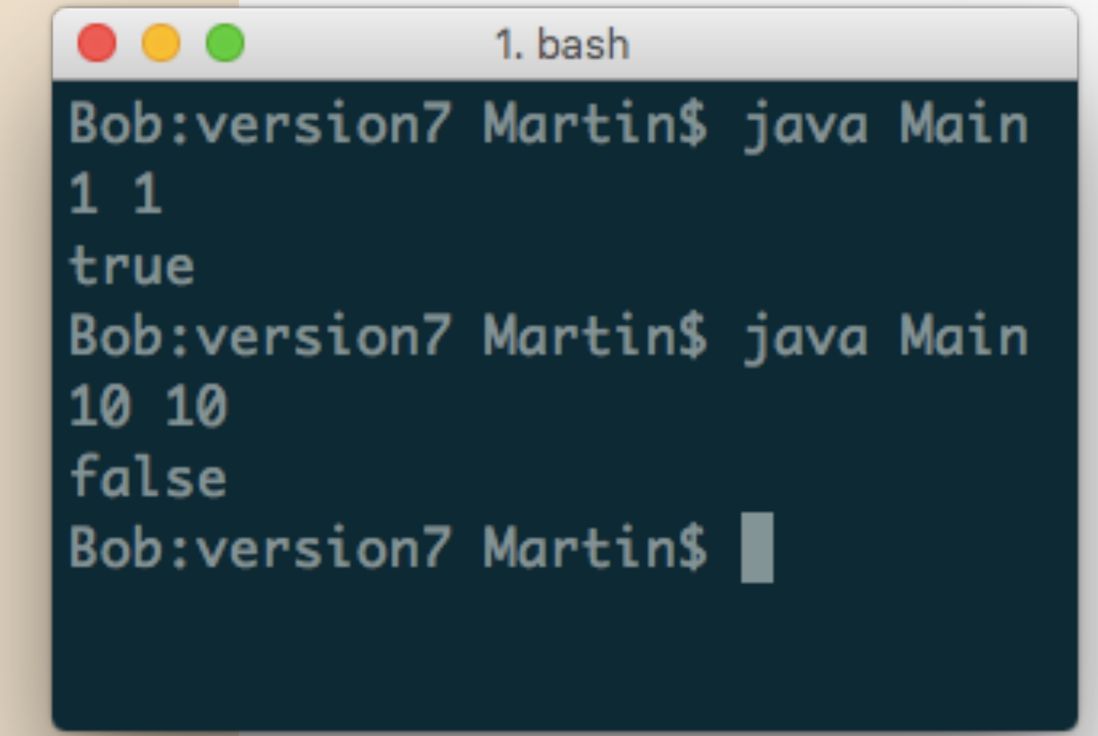
```
Game game = new Game();

Piece piece = new Piece(true);

Move firstMove = new Move(100, 100);

System.out.println(game.play(piece, firstMove));

END
```

We want to evolve our game, such that a user is asked for **input**, and if their move is not valid, they are asked for their input **again (V7.1)**.

INVOLVING THE USER: REQUIREMENT 7.1 (FIRST PASS)

We want to evolve our game, such that a user is asked for **input**, and if their move is not valid, they are asked for their input **again**.

```
Scanner in = new Scanner(System.in);

Game game = new Game();

Piece piece = new Piece(true);

boolean valid = false;

int row = in.nextInt();

int column = in .nextInt();

Move move = new Move(row, column);

valid = game.play(piece, move);

System.out.println(valid);

in.close();
```



```
Bob:version7 Martin$ java Main
1 1
true
Bob:version7 Martin$ java Main
10 10
false
Bob:version7 Martin$ 
```

Version 7, Main.java, main

INVOLVING THE USER: REQUIREMENT 7.1

```
Scanner in = new Scanner(System.in);

Game game = new Game();

Piece piece = new Piece(true);

boolean valid = false;

while (!valid) {

   int row = in.nextInt();

   int column = in .nextInt();

   Move move = new Move(row, column);

   valid = game.play(piece, move);

   System.out.println(valid);

}
```

We keep asking a user for input while their suggested move is not valid.

Version 7,
Main.java,
main

Version 8: Game Timeline

GAME TIMELINE: CURRENT TIMELINE

```
Scanner in = new Scanner(System.in);

Game game = new Game();

Piece piece = new Piece(true);

boolean valid = false;

while (!valid) {

   int row = in.nextInt();

   int column = in .nextInt();

   Move move = new Move(row, column);

   valid = game.play(piece, move);

   System.out.println(valid);

}
```

Our current timeline only allows for **one** valid move.

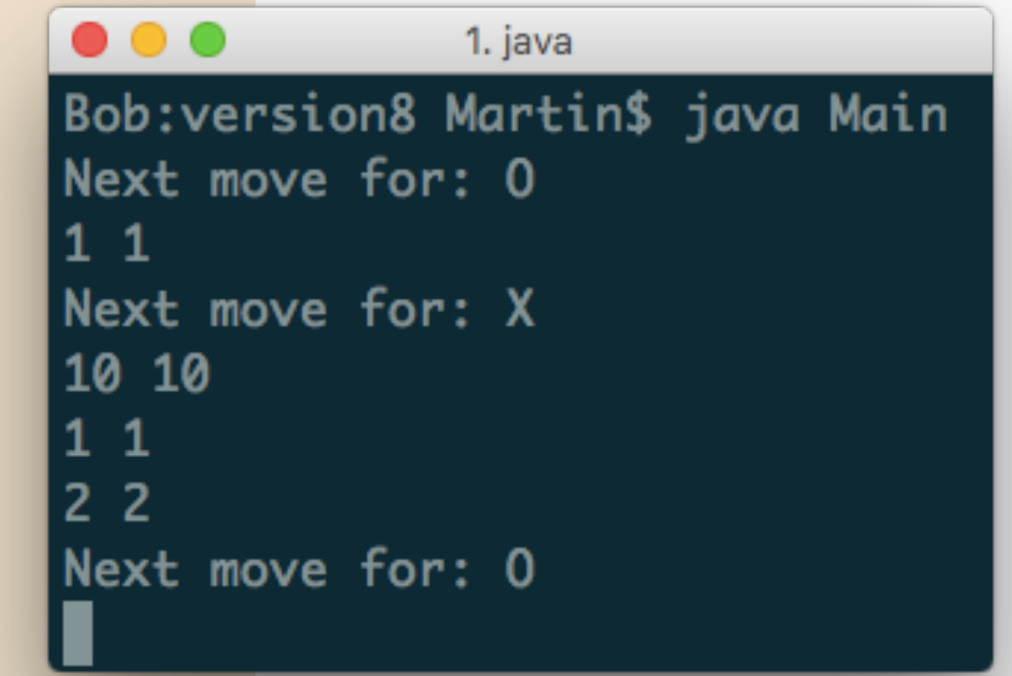
We obviously want to allow for **more than one** move.

For simplicity, let’s first imagine an **infinite game**.

GAME TIMELINE: REQUIREMENTS

```java
Piece piece = new Piece(true);

while (true) {

   System.out.println("Next move for: " + piece);

   boolean valid = false;

   while (!valid) {

      int row = in.nextInt();

      int column = in .nextInt();

      Move move = new Move(row, column);

      valid = game.play(piece, move);


   }

   piece = new Piece(piece);

}
```

V8.1 Add the capability for continuous moves, with alternating pieces.



```
Bob:version8 Martin$ java Main
Next move for: O
1 1
Next move for: X
10 10
1 1
2 2
Next move for: O
```

Version 8, Main.java, main

Version 9: Ending The Game (Part 1)

ENDING A GAME: OVERVIEW

We don't want to keep asking for pieces forever, which is the functionality we have currently.

Instead, we want to end a game in the following circumstances:

- (Part 1) When every cell is filled (i.e. after **9 moves**).
- (Part 2) When someone wins, with the following winning conditions: **full column, full row, full forward diagonal** or **full backward diagonal**

ENDING A GAME, PART 1: REQUIREMENTS

V9.1 Add a counter to a game that counts the number of moves made, and is able to report that a game is finished when that counter exceeds 9.

V9.2 A game should only continue while it is not finished.

ENDING A GAME, PART 1: REQUIREMENT V9.1 (1)

```
private int plays;
private boolean finished;
public boolean play(Piece piece, Move move) {
   if ( move.isValid() && canPlay(move) ) {
      board[move.getRow()][move.getColumn()] = piece;
      plays++;
      if ( plays == 9 ) finished = true;
      return true;
   } else {
      return false;
   }
}
```

Version 9, Game.java, play

V9.1 Add a counter to a game that counts the number of moves made, and is able to report that a game is finished when that counter exceeds 9.

ENDING A GAME, PART 1: REQUIREMENT V9.1 (2)

```
public boolean gameOver() {
    return finished;
}
```

```
plays = 0;
finished = false;
```

V9.1 Add a counter to a game that counts the number of moves made, and is able to report that a game is finished when that counter exceeds 9.

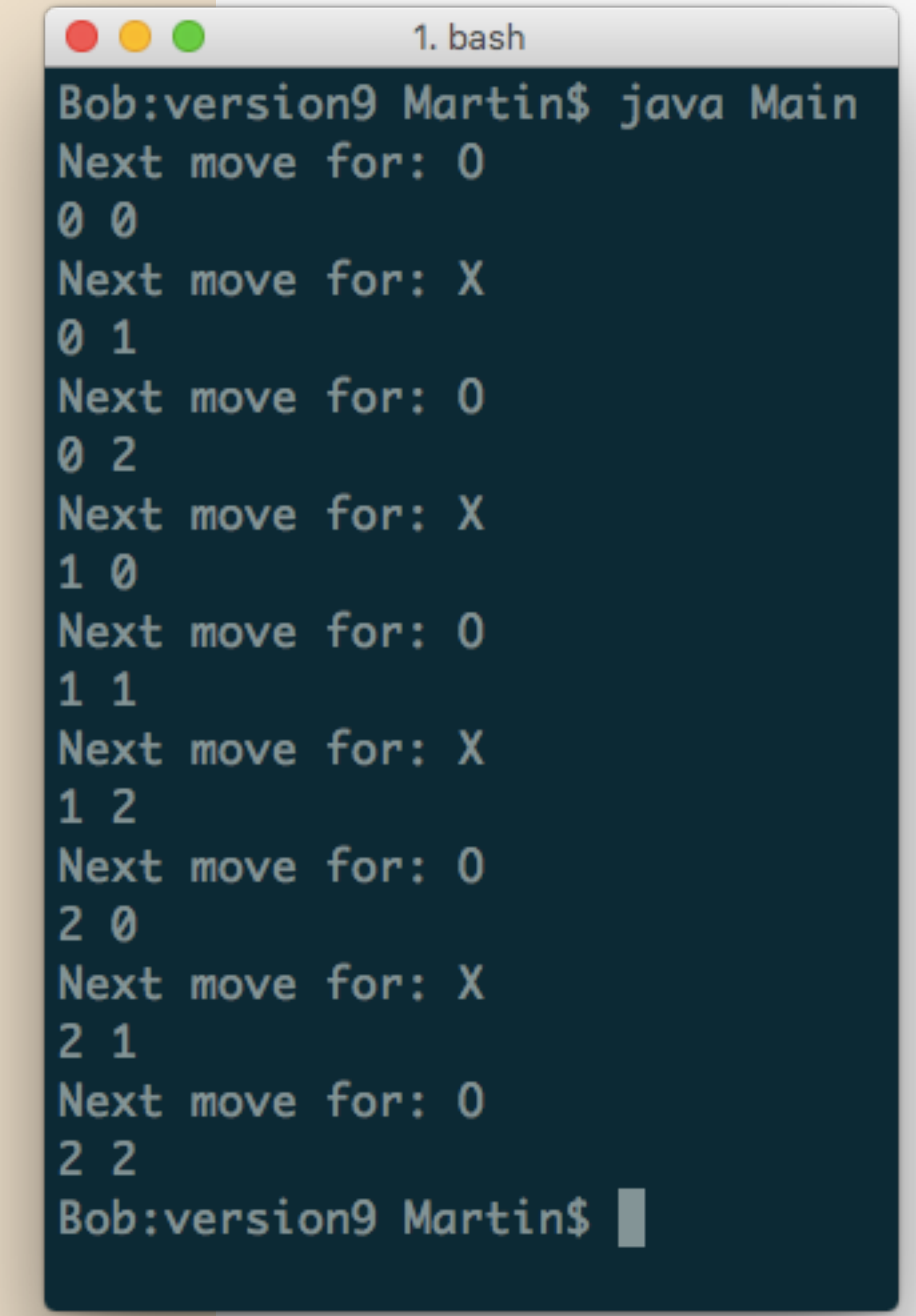
Version 9, Game.java, gameOver and newGame

ENDING A GAME, PART 1: REQUIREMENT V9.2

```
while ( !game.gameOver() ) {
    System.out.println("Next move for: " + piece);
    boolean valid = false;
    while (!valid) {
        int row = in.nextInt();
        int column = in .nextInt();
        Move move = new Move(row, column);
        valid = game.play(piece, move);
    }
    piece = new Piece(piece);
}
```

V9.2 A game should only continue while it is not finished.

Version 9, Main.java, main



```
Bob:version9 Martin$ java Main
Next move for: O
0 0
Next move for: X
0 1
Next move for: O
0 2
Next move for: X
1 0
Next move for: O
1 1
Next move for: X
1 2
Next move for: O
2 0
Next move for: X
2 1
Next move for: O
2 2
Bob:version9 Martin$
```

Version 10: Ending The Game (Part 2)

ENDING A GAME: OVERVIEW

We don’t want to keep asking for pieces forever, which is the functionality we have currently.

Instead, we want to end a game in the following circumstances:

- (Part 1) When every cell is filled (i.e. after **9 moves**).
- (Part 2) When someone wins, with the following winning conditions: **full column, full row, full forward diagonal** or **full backward diagonal**

ENDING A GAME, PART 2: SCAFFOLDING

```
 public boolean play(Piece piece, Move move) {
   if ( move.isValid() && canPlay(move) ) {
      board[move.getRow()][move.getColumn()] = piece;
      plays++;
      checkWinner();
      if ( plays == 9 ) finished = true;
      return true;
   } else {
      return false;
   }
}
```

We know that we want to check for a winner after each play.

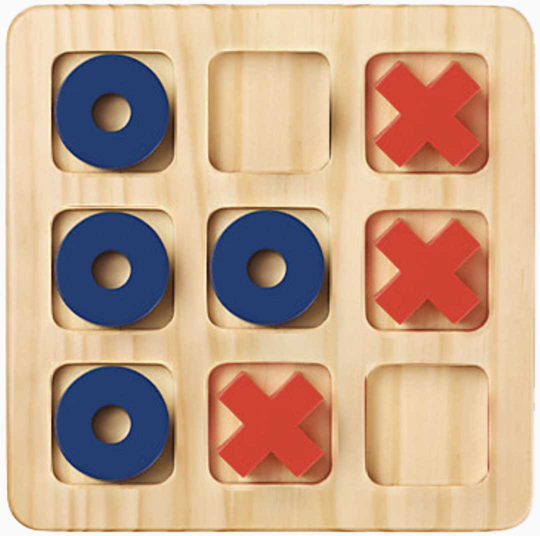
```
public void checkWinner() {

}
```

Methods provide us with the ability to scaffold for future functionality.

Version 10, Game.java, play

ENDING A GAME, PART 2: ONE FULL COLUMN (AND MORE SCAFFOLDING)

Let’s first imagine that the only way to win a game of noughts and crosses is to fill the first column (**column 0**, as shown).



We could add another method to our code to facilitate a check for this winning condition:

```java
public void checkWinner() {

    checkColumn();

}
```

```java
public void checkColumn() {


}
```

Version 10, Game.java, checkWinner and checkColumn

ENDING A GAME, PART 2: REQUIREMENTS

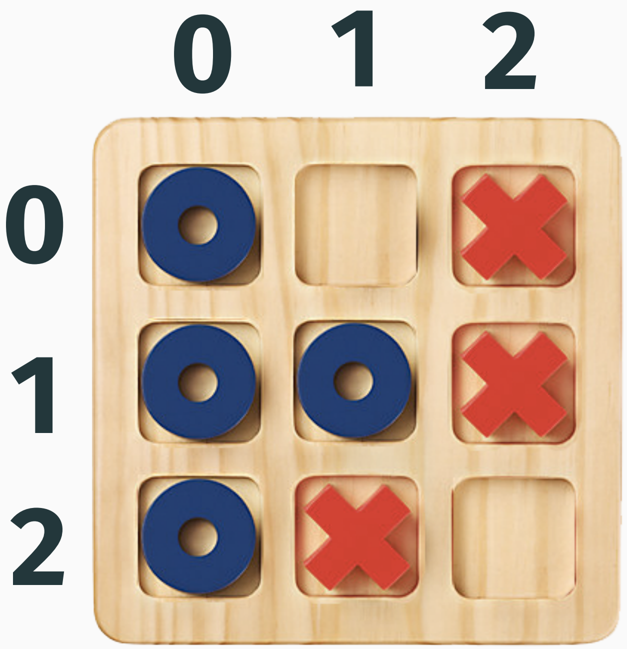
V10.1 Complete the functionality in checkColumn to discern whether the first column contains a winning move by either player, and set a variable to indicate which player this is, should a winning move occur.

V10.2 Change this method to determine whether **any** column contains a winning move by a player, and thus our notion of how a game is won.

V10.3 When a game is over, if nobody has won (because all the moves have been exhausted), then print this, otherwise, print the winner.

ENDING A GAME, PART 2: REQUIREMENT V10.1

```
private Piece winner;

public void checkColumn() {

  Piece extractedFirstPiece = board[0][0];

  if ( extractedFirstPiece != null &&
       extractedFirstPiece.matches(board[1][0]) &&
       extractedFirstPiece.matches(board[2][0])) {

    finished = true;

    winner = extractedFirstPiece;

  }

}
```



We use null here and short-circuit evaluation to protect against null references.

V10.1 Write a method that checks whether the first column contains a winning move by either player, and sets a variable to indicate which player this is, should this occur.

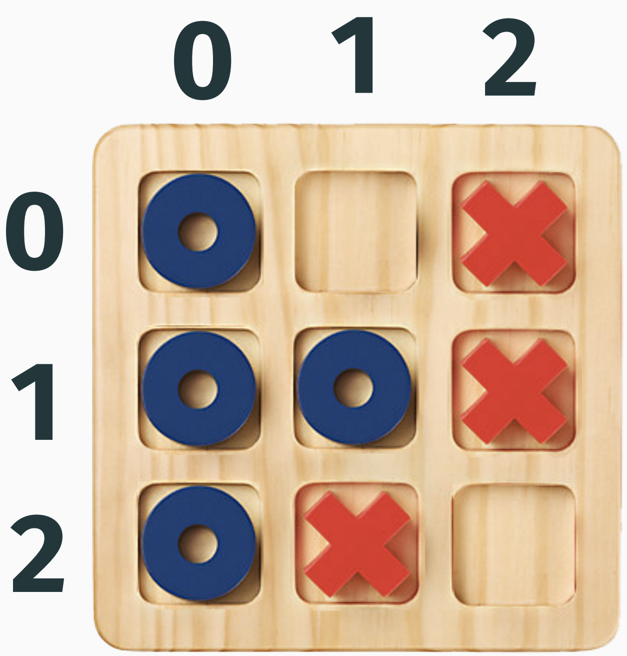
Version 10, Game.java, checkColumn

ENDING A GAME, PART 2: REQUIREMENT V10.2

```java
public void checkColumn(int column) {

    Piece extractedFirstPiece = board[0][column];

    if ( extractedFirstPiece != null &&
         extractedFirstPiece.matches(board[1][column]) &&
         extractedFirstPiece.matches(board[2][column])) {

        finished = true;

        winner = extractedFirstPiece;

    }

}
```

```java
public void checkWinner() {

    for ( int column = 0; column < 3; column++ ) {

        checkColumn(column);

    }

}
```

```java
public Piece getResult() {

    return winner;

}
```



V10.2 Extend this method to determine whether **any** column contains a winning move by a player, and thus our notion of how a game is won.

Version 10, Game.java, checkColumn, checkWinner and getResult

ENDING A GAME, PART 2: REQUIREMENT V10.3

V10.3 When a game is over, if nobody has won (because all the moves have been exhausted), then print this, otherwise, print the winner.

```java
System.out.println( game + "\n Game Over.");
if ( game.getResult() == null ) {
  System.out.println("Nobody won :-(");
} else {
  System.out.println(game.getResult() + " won :-)");
}
```

Version 10, Main.java, main

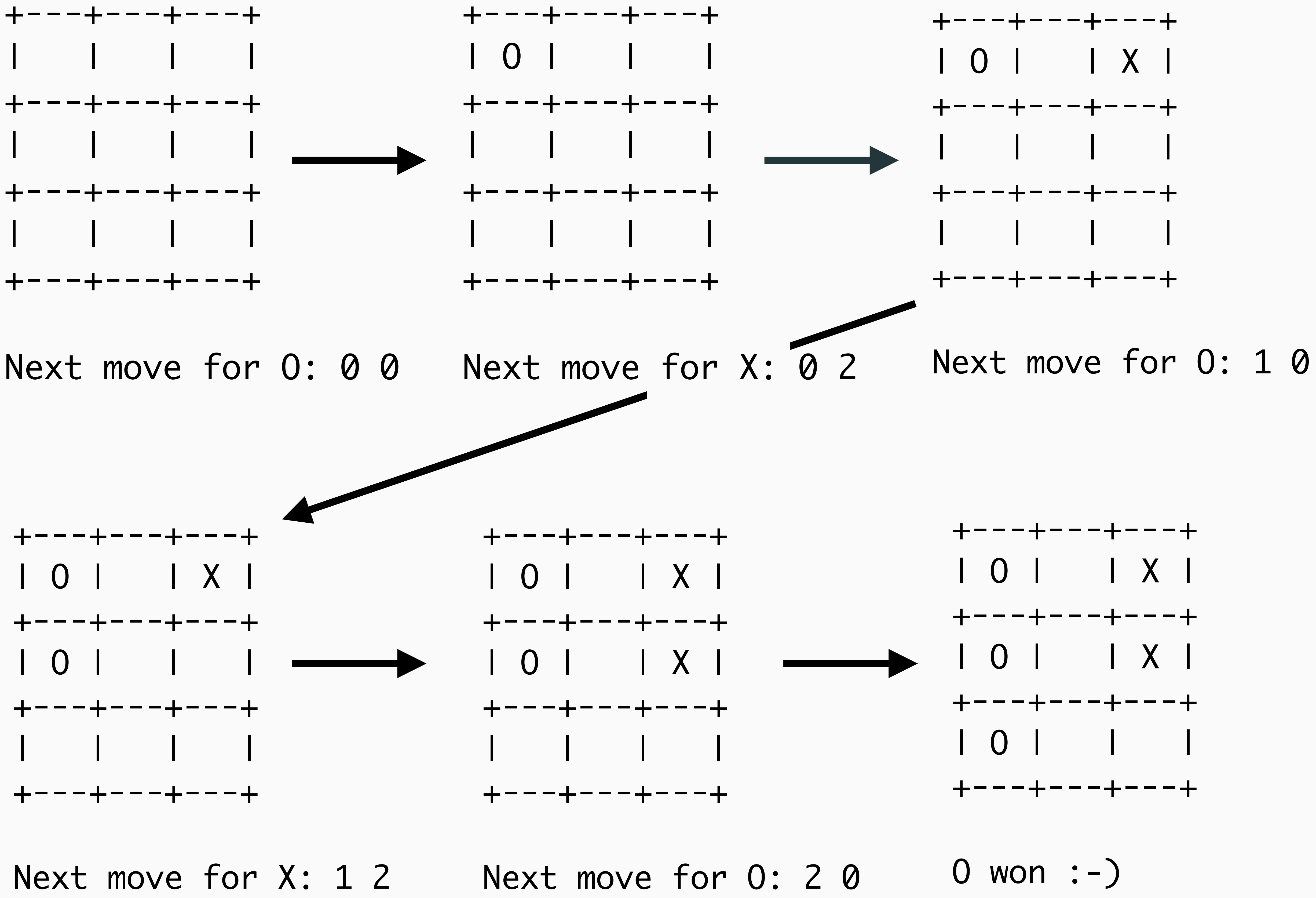
Playing The Game...

MAKING THINGS PRETTY

```
public String toString() {
        String output = "+---+---+---+\n";
        for ( int row = 0; row < 3; row++ ) {
          output = output + "| ";
            for( int column = 0; column < 3; column++ ) {
                if ( board[row][column] == null ) {
                  output = output + "  | ";
                } else {
                  output = output + board[row][column] + " | ";
                }
            }
            output = output + "\n+---+---+---+\n";
        }
        return output;
    }
```

Version 10, Game.java, toString

PLAYING A GAME



```
+---+---+---+
|   |   |   |
+---+---+---+
|   |   |   |
+---+---+---+
|   |   |   |
+---+---+---+

Next move for O: 0 0
```

```
+---+---+---+
| O |   |   |
+---+---+---+
|   |   |   |
+---+---+---+
|   |   |   |
+---+---+---+

Next move for X: 0 2
```

```
+---+---+---+
| O |   | X |
+---+---+---+
|   |   |   |
+---+---+---+
|   |   |   |
+---+---+---+

Next move for O: 1 0
```

```
+---+---+---+
| O |   | X |
+---+---+---+
| O |   |   |
+---+---+---+
|   |   |   |
+---+---+---+

Next move for X: 1 2
```

```
+---+---+---+
| O |   | X |
+---+---+---+
| O |   | X |
+---+---+---+
|   |   |   |
+---+---+---+

Next move for O: 2 0
```

```
+---+---+---+
| O |   | X |
+---+---+---+
| O |   | X |
+---+---+---+
| O |   |   |
+---+---+---+

O won :-)
```

COMPLETING THE GAME LOGIC

The game showed in the previous slide is still only based on detecting whether either player completes an entire column.

In the laboratory, complete the functionality, so that the remaining winning conditions can be detected:

- A row is completed.
- A forward diagonal is completed.
- A backward diagonal is completed.

You should add to the checkWinner method, and try and consider the efficiency of your solution: when do we no longer need to check for a winner?

Topic 7: Arrays

Programming Practice and Applications (4CCS1PPA)

Dr. Martin Chapman
Thursday 17th November

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.