Topic 2: Storing Data

Programming Practice and Applications (4CCS1PPA)

Dr. Martin Chapman

Friday 30th September

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

Q: If you were programming a computer game, when would you need to store data?

STORING DATA: OBJECTIVES

To understand **why** and **how** to store data in a program.

To understand the different **types** of **simple** data we can store in a program.

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World");
  }
}
```

Must be inside a file called HelloWorld.java

A SLIGHTLY MORE COMPLEX PROGRAM

*We tend to use different classes to hold **distinct** pieces of functionality, like this.*

*Because class names summarise functionality, they are typically **nouns**.*

```java
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**.

A THIRD PROGRAM: SOME MORE BOILERPLATE

```java
public class TimePrinter {
  public static void main(String[] args) {
    System.out.println(System.currentTimeMillis());
  }
}
```

TimePrinter.java

The number of milliseconds that have elapsed since midnight, January 1, 1970.

Using the syntax shown in the last three slides, we are able to solve **new** straight forward problems (**try these in your lab**).

- Print a 3x3 text grid to the terminal (using symbols).
- Print the current time to the terminal three times.

What about slightly more **complex** problems?

- Because the statements in a program are executed line-by-line, it takes **time** for a program to run. In theory, the more lines (or if particular operations are performed within those lines), the longer the program takes to run.
- Print how long it takes for the computer to execute the print line statements from Slide 4 (only using `System.currentTimeMillis()`).

RUNNING TIME

```
public class MartinPrinter {

  public static void main(String[] args) {

    Record the current time

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

    System.out.println(  Compute the difference between the    );
                         time now and the recorded time.
  }

}
```

MartinPrinter.java

For solving problems like this in Java, it is clear that we need to **store** some data.

Calculating running time is quite a **specific** problem motivating the storage of data.

- But it avoids the introduction of new syntax at this stage.

There are a number of other more **intuitive** problems that require the storage of data, that might be too complex to discuss at this stage.

- Storing the current score in a game.
- Storing the result of user input.
- Counting the number of cars entering a car park.

When programming, **variables** provide us with a place in which to store data.

Here's an example of a **variable declaration**:

```
public class VariableTester {

   public static void main (String[] args) {

      int myFirstIntegerVariable;

   }

}
```

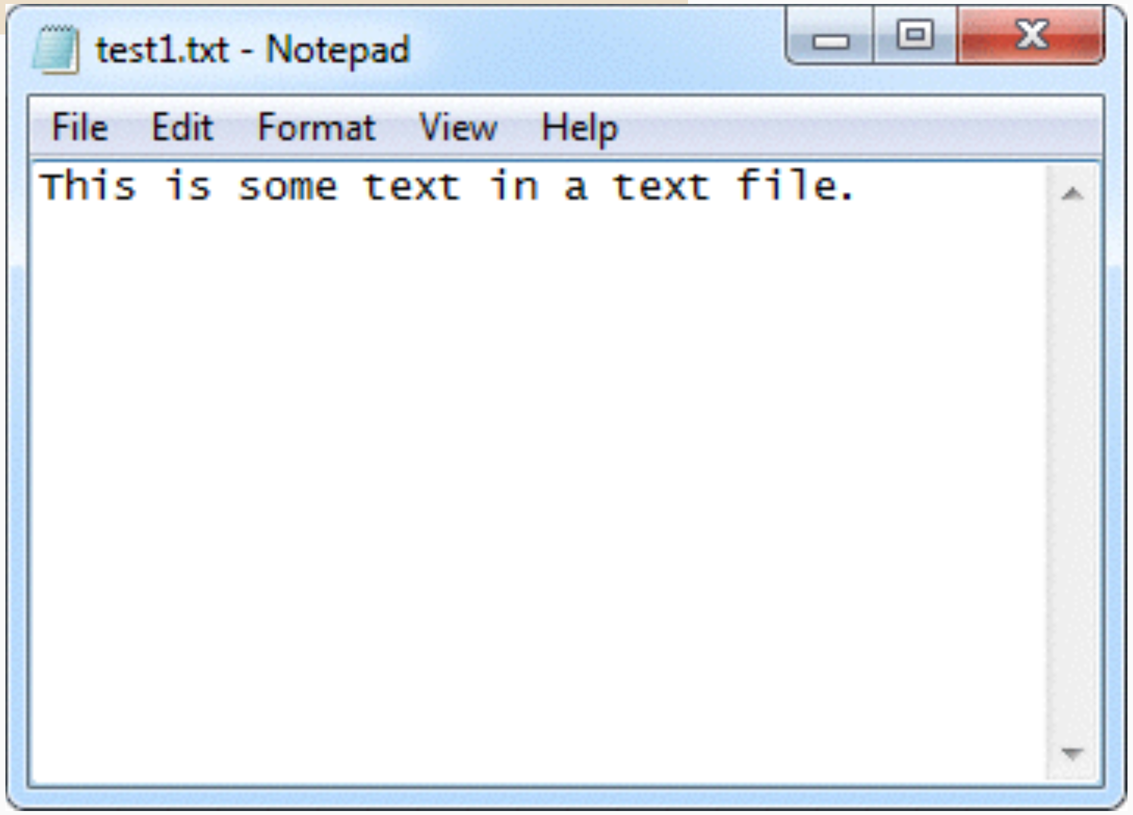
VariableTester.java

To help us conceptualise a variable, and what we can do with it, we have a choice of (useful) **metaphors**.

(USEFUL) METAPHORS GALORE

```java
public class VariableTester {

    public static void main (String[] args) {

        int myFirstIntegerVariable;

    }

}
```

A labelled box



A named file

A VARIABLE AS A BOX (METAPHOR 1) (1)

We give a variable a **name**.


```
int myFirstIntegerVariable;
```

There is a **label** on our box.

REMEMBER: NAMING RULES AND CONVENTIONS IN JAVA (1)

Not all the words in a Java program have a special meaning. Some are **selected by us**.

```
int myVerySpecialPlaceInWhichToStoreThings;
```

But there are rules for **human selected** text (intuitive)

- Cannot **begin with a number** (because we wouldn't be able to identify **values**).
- Cannot contain **spaces**.
- Cannot contain symbols **other** than $ and _.
- Cannot be **reserved Java keywords** (such as int).

REMEMBER: NAMING RULES AND CONVENTIONS IN JAVA (2)

camelCaseNotation is **typical**, but not enforced

- Variable names do **not** capitalise the first letter (**myFirstIntegerVariable**)

Making intelligible and presentable variable name choices is part of the **artistic** element of coding.

```java
int myAMZNGVARIABLE777776611
```

Using the same variable name twice will result in a **name conflict**.

- But any name selected will differ from the same name written in a **different case**.

We can **assign** values to the variable.

To run, this code would still have to be in a main method, in a class, I've just shortened it for readability.

*The first time we assign a value to a variable is called **initialisation**.*


```
int myFirstIntegerVariable;
myFirstIntegerVariable = 1;
```

or

```
int myFirstIntegerVariable = 1;
```

*We can **declare** and **initialise** a variable in the same line if we already know the **value** it will contain.*

We can open the box, and put things in.

We can **reference** the variable as many times as we like, using its name.


```
int myFirstIntegerVariable = 1;
System.out.println(myFirstIntegerVariable);
System.out.println(myFirstIntegerVariable);
```

We can open the box and look inside as many times as we like.

A VARIABLE AS A BOX (METAPHOR 1) (4)

We can **reassign** the variable as many times as we like.


```
int myFirstIntegerVariable = 1;
myFirstIntegerVariable = 3;
myFirstIntegerVariable = 20;
```

We can open the box, take something out, and put something else in.

A VARIABLE AS A BOX (METAPHOR 1) (5)

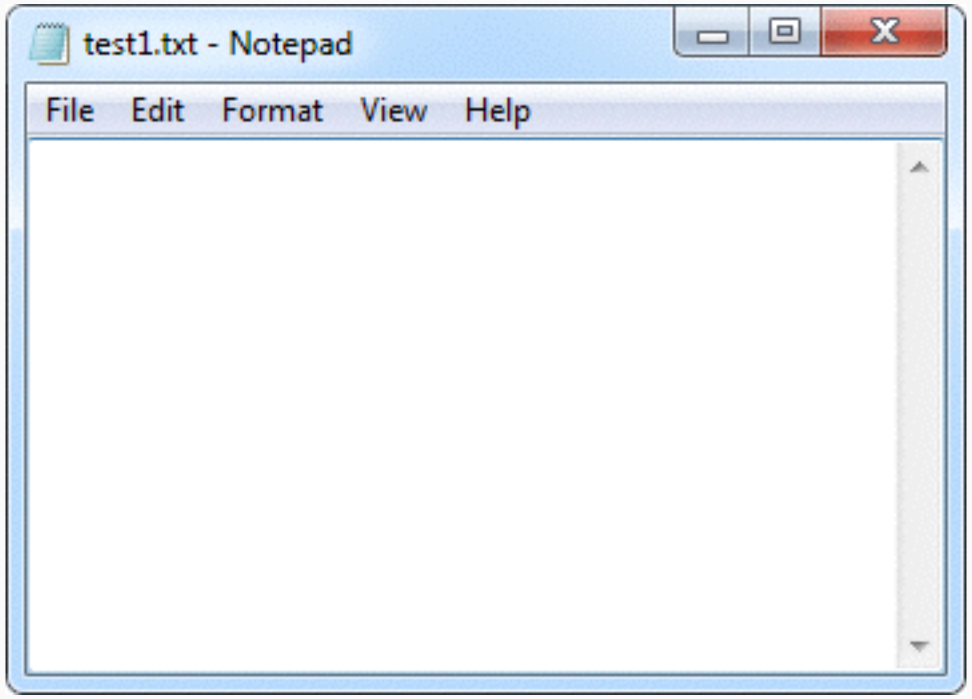
We **declare** that this variable is designed to store integers (whole numbers within a certain range).


```
int myFirstIntegerVariable;
```

We place a label on the box, so that it is **clear what we expect to find inside**.

A VARIABLE AS A FILE (METAPHOR 2) (1)

We give a variable a **name**.

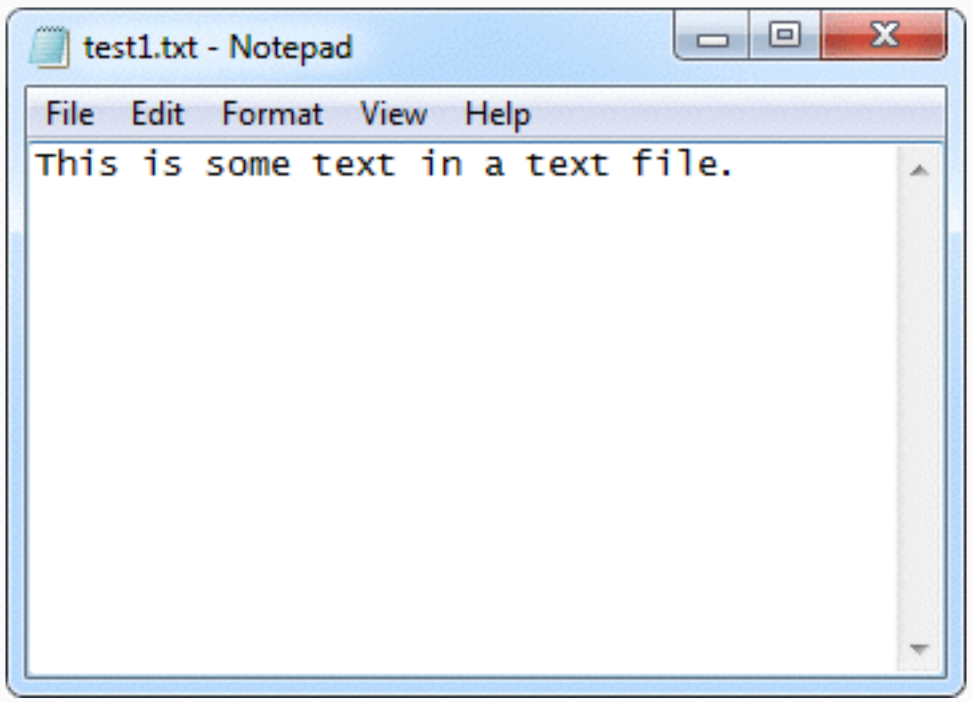



```
int myFirstIntegerVariable;
```

A file has a **name**.

A VARIABLE AS A FILE (METAPHOR 2) (2)

We can **assign** values to the variable.




```
int myFirstIntegerVariable;
myFirstIntegerVariable = 1;
```

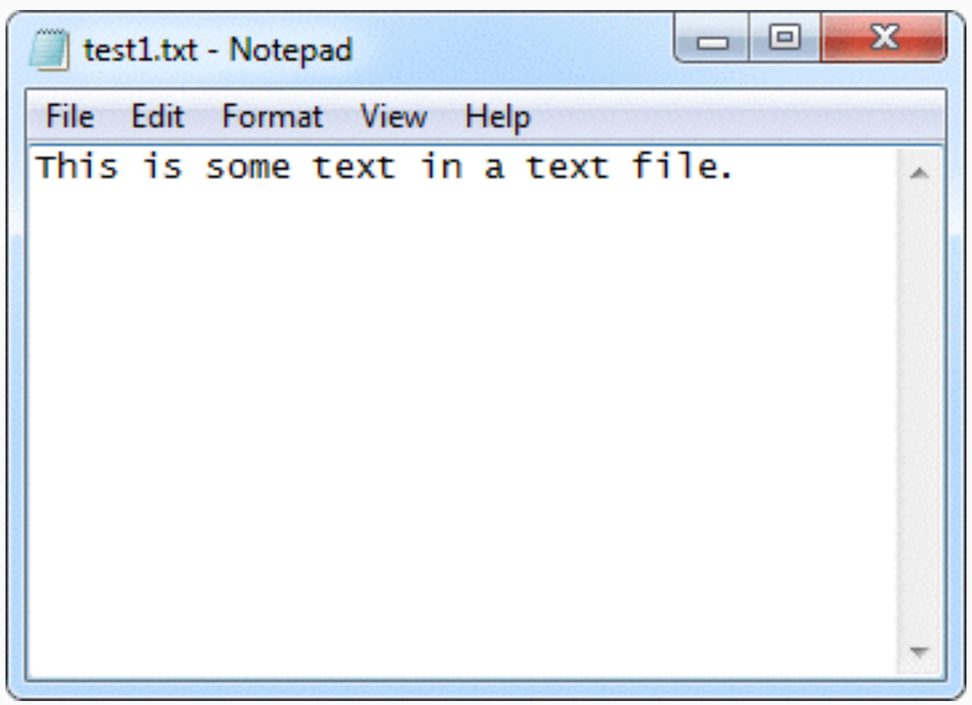
or

```
int myFirstIntegerVariable = 1;
```

We can open the file, and **enter** some information.

A VARIABLE AS A FILE (METAPHOR 2) (3)

We can **reference** the variable as many times as we like, using its name.

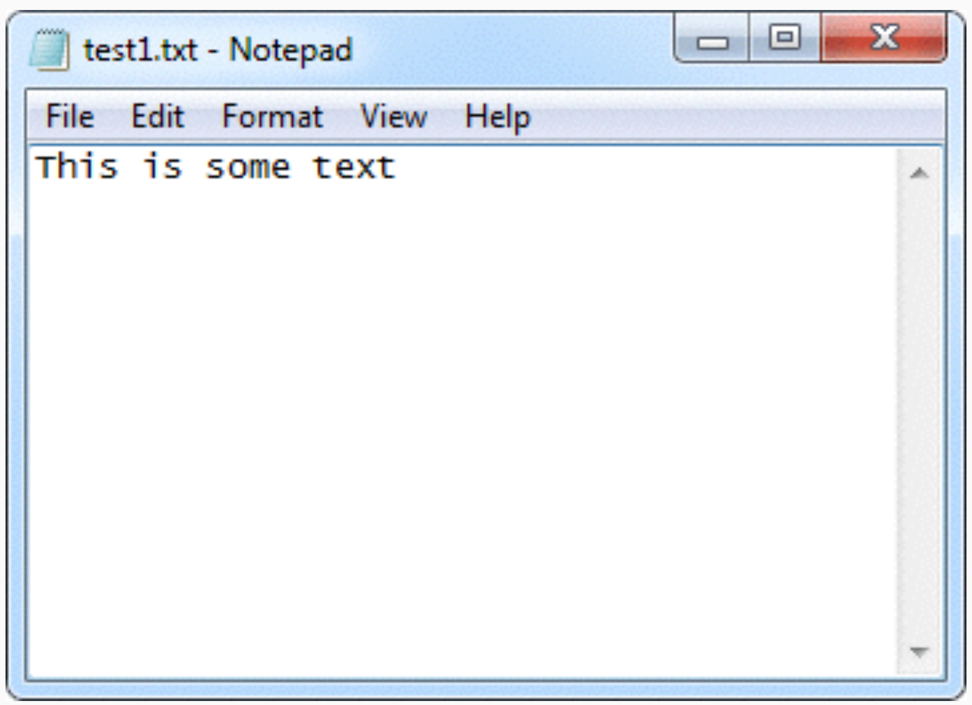



```java
int myFirstIntegerVariable = 1;
System.out.println(myFirstIntegerVariable);
System.out.println(myFirstIntegerVariable);
```

We can open the file and **extract** the information as often as we wish.

A VARIABLE AS A FILE (METAPHOR 2) (4)

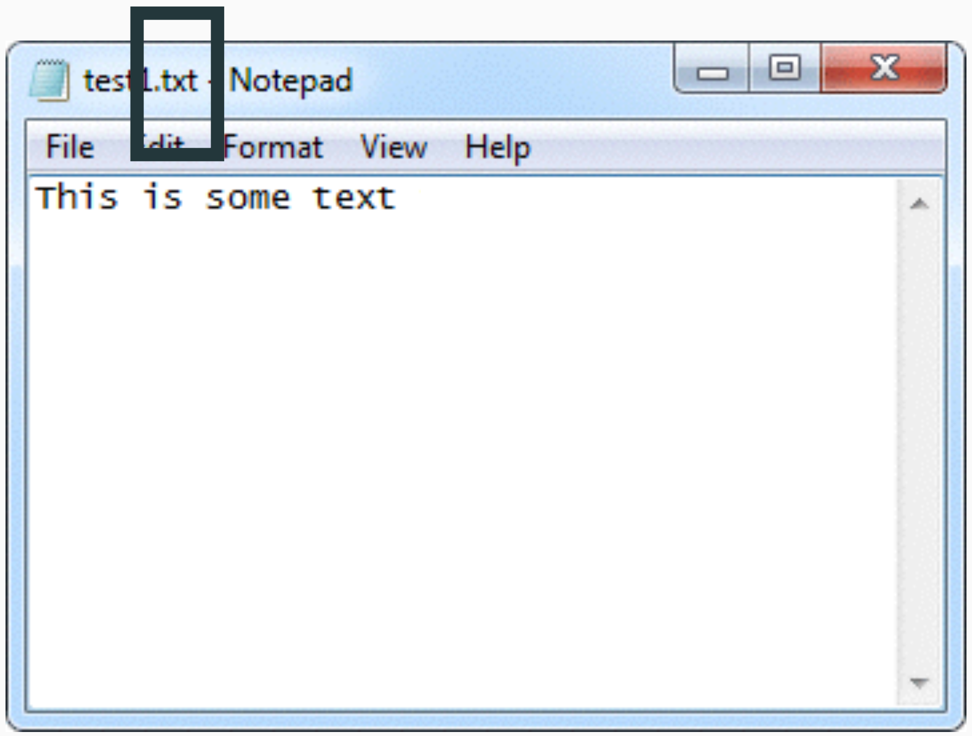
We can **reassign** the variable as many times as we like.




```
int myFirstIntegerVariable = 1;
myFirstIntegerVariable = 3;
myFirstIntegerVariable = 20;
```

We can **delete** the text in the file, and enter something new.

We declare that this variable is designed to store integers (whole numbers within a certain range).




```
int myFirstIntegerVariable;
```

A file demonstrates the type of data it can hold through its labelled **extension** (e.g. .txt means text content).

TYPED LANGUAGES

Like a labelled file extension, languages care, to different degrees, about you being **explicit** about the types of things you intend to store in a variable.

- Errors can be determined at **compile time**.
- Our code is **readable**.
- Some loss of flexibility.

Java **forces** us to explicitly state what we **intend to store** in a variable.

- Languages like **Python** or **Javascript** are less strict.

Of course we're already adhering to this rule in our example.

```java
public class VariableTester {

   public static void main (String[] args) {

      int myFirstIntegerVariable;

   }

}
```

VariableTester.java

Here, we've told Java that we've decided to **only** store integers in this variable.

The format of a variable declaration is thus **type** followed by **name**.

A good way to learn how to program is to try to **break** things.

For example, what happens if we **don't** specify a type?

```
public class VariableTester {

  public static void main (String[] args) {

    myFirstIntegerVariable;

  }

}
```

ERROR: NOT A STATEMENT

1. bash

```
Bob:topic2 Martin$ javac VariableTester.java
VariableTester.java:7: error: not a statement
                myFirstIntegerVariable;
                ^
1 error
Bob:topic2 Martin$
```

The role of the **Java compiler** is not just to transform our text file into a program, but also to tell us when our text file contains certain keywords that are **not valid Java syntax**.

Every time we do something wrong, like in the previous slide, we will receive an **error message** in the terminal, **rather** than any **output** from our **own** program.

These are called **compile-time** errors.

BREAKING THE RULES (2): NO VALUE

What happens if we try and **interact** with a variable (e.g. try and print it), without first **assigning a value to that variable**?

```
int myFirstIntegerVariable;
System.out.println(myFirstIntegerVariable);
```

ERROR: VARIABLE MIGHT NOT HAVE BEEN INITIALISED

```
Bob:topic2 Martin$ javac VariableTester.java
VariableTester.java:47: error: variable myFirstIntegerVariable might not have been initialized
                System.out.println(myFirstIntegerVariable);
                                   ^
1 error
Bob:topic2 Martin$
```

This occurs because variables, when they are used in this way, are **not** given default values.

Aside: this contrasts other languages, where the accessible content of uninitialised variables depends on the state of memory.

Later on, we will see variables used in a different way. Then, variables **will** have default values.

BREAKING THE RULES (3): WRONG TYPE

What happens if we assign a **floating point number** to an declared integer variable?

```
int myFirstIntegerVariable = 3.1415;
```

ERROR: POSSIBLE LOSSY CONVERSION

1. bash

```
Bob:topic2 Martin$ javac VariableTester.java
VariableTester.java:13: error: incompatible types: possible lossy conversion
from double to int
                int myFirstIntegerVariable = 3.1415;
                                             ^
1 error
Bob:topic2 Martin$
```

BREAKING THE RULES (3): WRONG TYPE

What happens if we assign a **floating point number** to an declared integer variable?

```
int myFirstIntegerVariable = 3.1415;
```

In the same way that you cannot (easily) place an image into a text file, you cannot place data of a **different type** into a typed variable (directly).

This is usually because it would result in us **losing some data** (more shortly).

Fortunately, variables can store different **types** of data, not just integers, so we simply declare the right type for our purpose...

Double is a type that allows us to represent floating point numbers.

```
double myFirstDoubleVariable = 3.1415;
```

Different data types have different **representations** in **memory**.

It is not strictly true that you cannot assign values of different types to typed variables.

If this were true programming in Java would be very **frustrating**.

In reality, values can be **converted** (or **cast**) as different types.

This will happen **automatically** (**implicit** casting) but only when there will be **no loss of precision**.

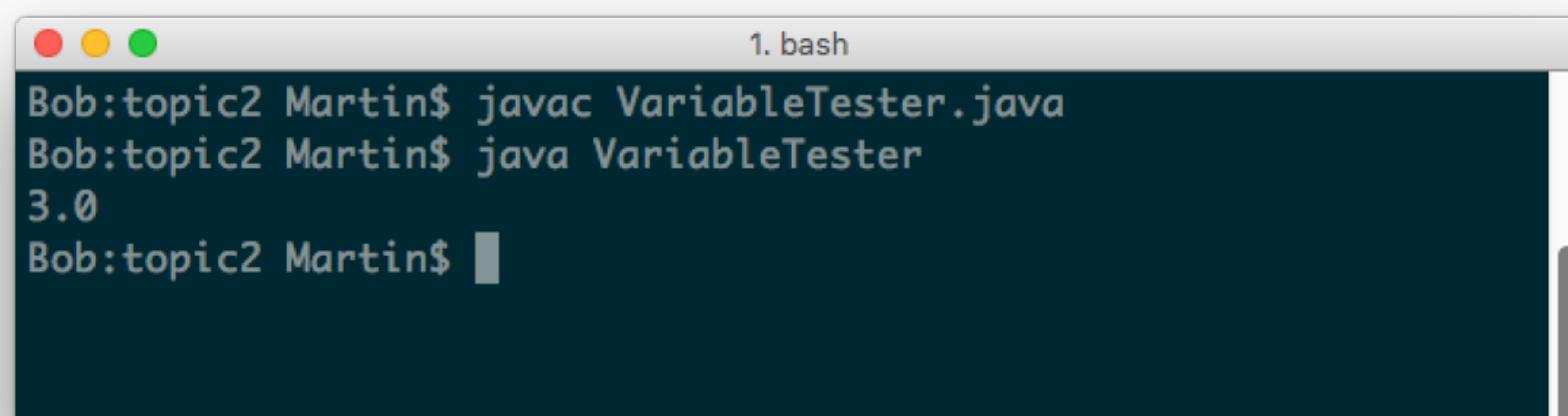
ASIDE: CASTING (IMPLICIT)

For instance, in our previous example, when we tried to assign a floating point number to an integer variable, Java would be **unable** to store the fractional part (given the **memory format** used to store an integer), so automatic casting does not take place, and we get an **error**.

```java
int myFirstIntegerVariable = 3.1415;
```

However, the other way round is fine, because nothing is **lost**. An integer is **implicitly** converted to a double with **extra** precision.

```java
double myFirstDoubleVariable = 3;
System.out.println(myFirstDoubleVariable);
```



ASIDE: CASTING (EXPLICIT)

We could, if we desperately wanted, **force** Java to omit the additional precision of a double, thus allowing us to store the double as an integer.

```java
int myFirstIntegerVariable = (int)3.1415;
System.out.println(myFirstIntegerVariable);
```

```
1. bash
Bob:topic2 Martin$ javac VariableTester.java
Bob:topic2 Martin$ java VariableTester
3
Bob:topic2 Martin$
```

This bracketed notation is called an **explicit cast**.

Explicit casts from double to integer are rare (in my experience), but later in the course we will see when explicit casts are more important.

ASIDE: CASTING

Why `casting'? Because the **same value** is **cast in different roles**.

What happens if we assign a **huge** number to an integer variable?

```
int myFirstIntegerVariable = 4294967296;
```

ERROR: INTEGER NUMBER TOO LARGE

1. bash

```
Bob:topic2 Martin$ javac VariableTester.java
VariableTester.java:9: error: integer number too large: 4294967296
                int myFirstIntegerVariable = 4294967296;
                                             ^
1 error
Bob:topic2 Martin$ 
```

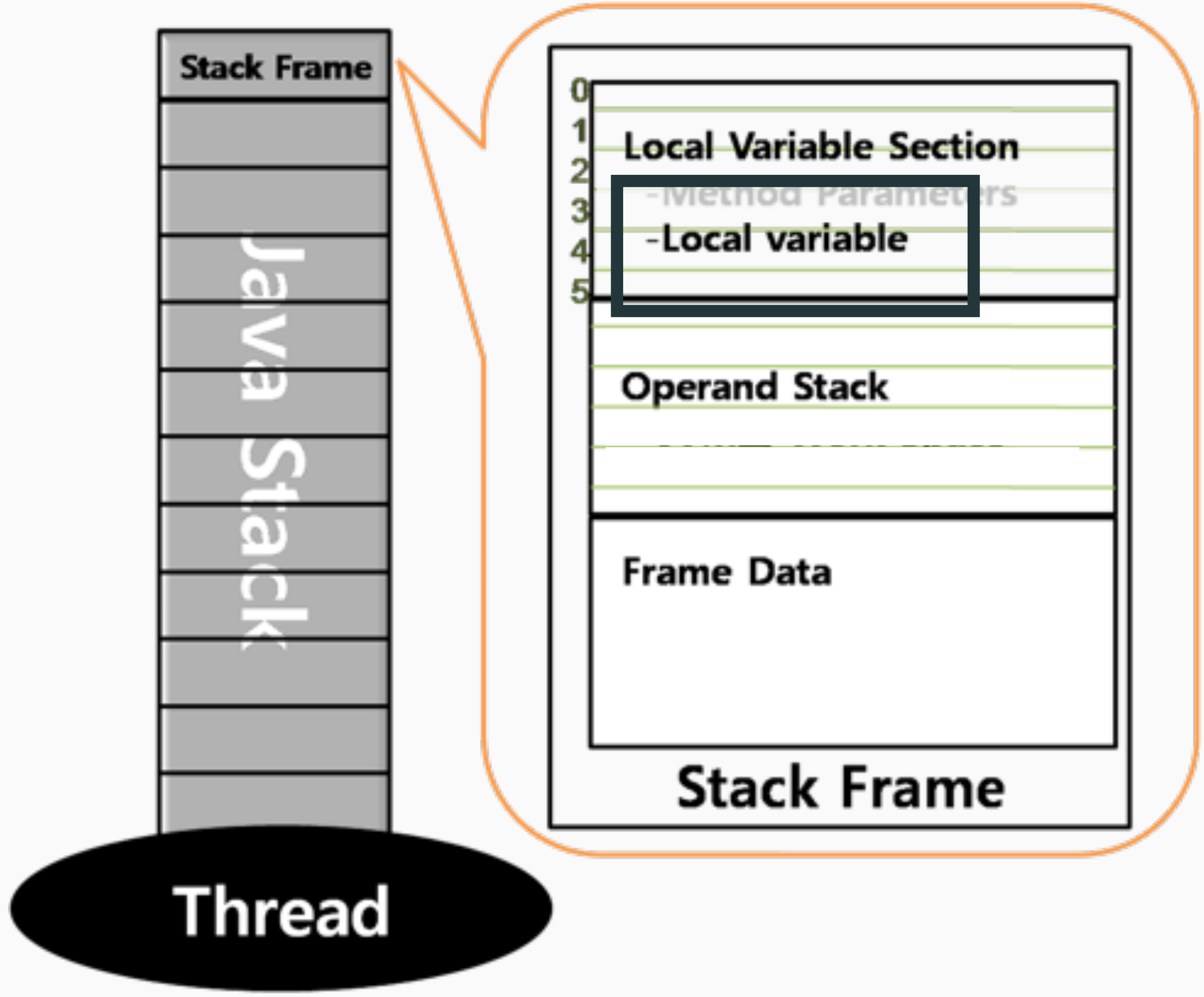
What happens if we assign a **huge** number to an integer variable?

```
int myFirstIntegerVariable = 4294967296;
```

Why is this? Variables are given a **maximum size** in **memory**. The sizes selected were designed to maximise **speed** and reduce **memory consumption**.

- More about this in **4CCS1CS1**
- Why have I chosen this number specifically?

ASIDE: MEMORY ALLOCATION OF VARIABLES



This doesn't mean we will be purposefully inefficient!

For the majority of the tasks in PPA, you will not have to be concerned about memory.

More generally, we will typically concern ourselves with what is going on at a **high level**, rather than at the machine level.

If we want to store a large number, we have to use a different datatype, a **long**; `long integer’

```
long myFirstLongVariable = 4294967296l;
```

What’s this?

So far, we have seen several **syntactic representations** of data types. We use these to express **values** in our program.

```
double myFirstDoubleVariable = 3.1415;
```

We call these **literals**.

Sometimes the type of a literal isn't always clear, so we need **additional syntax** to help us specify type.

```
long myFirstLongVariable = 4294967296l;
```

In the previous example it is isn't clear whether the literal is **supposed** to be an integer **or** a long, so we postfix an **`l'** to make this distinction.

ASIDE: BACK TO RUNNING TIME

We now have enough syntax to solve our running time problem.

```
public class MartinPrinter {

    public static void main(String[] args) {

        long currentTime = System.currentTimeMillis();

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

        System.out.println(System.currentTimeMillis() - currentTime);

    }

}
```

Anything that can be printed can also be stored

We can do simple arithmetic in Java, using standard notation (more later).

MartinPrinter.java

Open question: How do we know that time is returned in a long format?

ASIDE: COMMENTS

Now that we have code of reasonable **complexity**, it makes sense that we should **annotate** this code with a **description**, in order to remind us of what it does, should we wish to come back to it in the future.

Single line comments (ignored plaintext, which starts with a double slash notation) enable us to do this.

```
public class MartinPrinter {

    public static void main(String[] args) {

        // Record the current time.
        long currentTime = System.currentTimeMillis();

        // Perform the operation we wish to measure.
        System.out.println("+------+");
        System.out.println("|Martin|");
        System.out.println("+------+");
```

Annotating any code you find in these slides with comments, as you write it out, is ***excellent for revision***.

Remember: If we want to store a large number, we have to use a different datatype, a **long**; `long integer'

```
long myFirstLongVariable = 4294967296l;
```

The opposite is true of **double**. We could represent a **smaller** value using the type **float**. (Think double = double size).

```
float myFirstFloatVaraible = 3.1415f;
```

Note the intuitive literal

COMPARISON OF TYPES

FLOAT OR DOUBLE? LONG OR INT?

Somewhat a **style** choice.

I rarely use **longs**, opting for **integers** (I have no need, usually, for the additional space).

- We might have to use longs if the **values given to us** are of the long type (as we saw with our running time example).

I rarely use **floats**, because **doubles** offer more precision, and the memory overhead is **negligible.**

OTHER DATA TYPES (1)

The types we have seen so far — int, long, float, double — are **simple**, so we refer to them as **primitive types**.

For some of these primitive types, we have answered the following:

- What kind of **literal values** can they store? How do we **differentiate literals** (i.e. do we need to postfix a particular character)?
- What happens if you try and assign values of a **different** type to variables of **this** type?
- Can other type values be **implicitly cast** to this type?
- What happens if other type values are **explicitly cast** to this type?

Complete this analysis for **all** types. There are **8** primitive types in total, so you must **research** the **4** types we have not talked about.

OTHER DATA TYPES (2)

Keep an exhaustive list of any tests you do as code, and note the result using **comments**.

- These comments don't necessarily have to describe what the code does (as we saw previously), but instead what you have **learnt**.
- Comments like this are good for your own **understanding** but you **shouldn't** generally use them in code you submit for assessment.

```
// Integer variables accept integer literals.
int myFirstIntegerVariable = 1;

// Integer variables do not accept double literals.
int mySecondIntegerVariable = 1.0;
```

Keep this **for reference** and for **revision**.

ASIDE: IMPERATIVE VS. FUNCTIONAL LANGUAGES (1)

Java is traditionally an **imperative** language.

- There's an expectation that we solve problems by issuing **instructions** that change a program's **state** (held in **variables**).
- We describe **how** things are done in a series of **steps**.

ASIDE: IMPERATIVE VS. FUNCTIONAL LANGUAGES (2)

Other languages (e.g. Haskel, Prolog) are **declarative**.

- Often considered an **abstraction** over imperative languages, where we state **what** to do, **not** how to do it.
- Popularly defined by the sub-approach **functional programming**, where state cannot be changed (i.e. variables cannot be changed), but instead problems are solved using a set of functions that correlate **input to output**.
- So we could solve some of the problems stated in these slides **without storing** any results.

In the latest version of Java, the introduction of certain syntactic elements (e.g. Lambda expressions) suggests more of a synthesis with functional programming.

Which is better? Hotly debated. But imperative is currently more widely used.

🤔

ASIDE: IMPERATIVE VS. FUNCTIONAL LANGUAGES (3)

```
function double (arr) {
  let results = []
  for (let i = 0; i < arr.length; i++){
    results.push(arr[i] * 2)
  }
  return results
}
```

```
function double (arr) {
  return arr.map((item) => item * 2)
}
```

Topic 2: Storing Data

Programming Practice and Applications (4CCS1PPA)

Dr. Martin Chapman

Friday 30th September

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.