Topic 4: Why Object-Orientation?

Programming Practice and Applications (4CCS1PPA)

Dr. Martin Chapman

Thursday 13th October

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

Q: What do you think constitutes a real world object?

To understand why object-orientation is an important **paradigm** for code development.

Reason #1: Code Organisation (Review)

REMEMBER: 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.

```java
public class Driver {

  public static void main(String[] args) {

     MartinPrinter copyOfMartinPrinter = new MartinPrinter();

  }

}
```

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

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.*

```java
public class MartinPrinter {

  public           void printMartin() {

  }

}
```

```java
public class NumberPrinter {

  public        void printNumber(int num) {

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

😴

LECTURE EXERCISE: PRETTY TIME (1)

Remember `System.currentTimeMillis()` ? The native output from this statement isn't particularly clear (it's just a number).

Make a class called `PrettyTime` with a method called `printTime`. In this method we should print "The Unix time is: " followed by the (Unix) time.

You should then make a `Driver` class, which makes an object of the `PrettyTime` class, and uses this object to call the `printTime` method.

LECTURE EXERCISE: PRETTY TIME (2)

```
public class PrettyTime {

   public void printTime() {

      long currentTime = System.currentTimeMillis();
      System.out.println("The Unix time is: " + currentTime);

   }

}
```

*printTime here is **not** static, and the methods in your coursework should not be either.*

```
public class Driver {

   public static void main(String[] args) {

      PrettyTime prettyTime = new PrettyTime();
      prettyTime.printTime();

   }

}
```

*We **could** leave this methods static, and reference it using the class, but as we haven't yet **fully explored the implications of doing this**, we will not.*

LECTURE EXERCISE: PRETTY TIME (3)

1. bash

```
Bob:topic4 Martin$ javac Driver.java
Bob:topic4 Martin$ java Driver
The Unix time is: 1474816475617
Bob:topic4 Martin$
```

ASIDE: MAKING THINGS MORE EFFICIENT (1)

Often, when first learning to program, people do things in a fairly **verbose** way.

This is fine, as its important to start somewhere.

But often, when reviewing code, we can find small enhancements to make.

For example, these two lines:

```
long currentTime = System.currentTimeMillis();
System.out.println("The Unix time is: " + currentTime);
```

Could be reduced to this **single** line:

```
System.out.println("The Unix time is: " + System.currentTimeMillis());
```

ASIDE: MAKING THINGS MORE EFFICIENT (2)

Investigating your code and trying to **reduce the number of lines** you have used is an important skill to learn.

- Less lines are **neater** and **easier to read**
- Code is often more **efficient**. In this case, for example, we're not using memory to store the time **unnecessarily** (although remember we won't concern ourselves with memory **too** much).

In the laboratories, when you are **copying the code in these slides** in order to try it out, **check** if anything can be done more efficiently.

- Have I done certain things inefficiently? **Probably**! Catch me out, and let me know.
- **Tackling verbosity** does not always necessitate the introduce of new syntax.

WHY OBJECT-ORIENTATION: SO FAR WE KNOW...

Classes and objects provide us with a way to **organise** our code (Topic 3).

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**.

Reason #2: Reusability

REUSABILITY (1)

In the previous exercise, we were again able to use objects and classes to **organise** our code nicely, such that methods like `printTime` can be called **along with** methods from other classes, while still retaining a separation of functionality (as we did in Topic 3).

```
public class Driver {

    public static void main(String[] args) {

        MartinPrinter copyOfMartinPrinter = new MartinPrinter();
        copyOfPrintMartin.printMartin();

        NumberPrinter copyOfNumberPrinter = new NumberPrinter();
        copyOfNumberPrinter.printNumber(191);

        PrettyTime prettyTime = new PrettyTime();
        prettyTime.printTime();

    }

}
```

The **implicit** benefit of this is **reusability**. We could call `printTime` in one program, and then use it in **another** program, simply by making an object of the class **again**.

And of course, we can do the same thing inside any of our classes, even those **without** a main method.

```
public class NumberPrinter {

    public void printNumber(int num) {

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

        PrettyTime prettyTime = new PrettyTime();
        prettyTime.printTime();

    }

}
```

```
public class Dri                                      

    public static                                     g[] args) {

        PrettyTime                                    w PrettyTime();
        prettyTime

    }

}
```

WHY OBJECT-ORIENTATION: SO FAR WE KNOW...

In increasing order of importance (to keep the purists happy):

Classes and objects provide us with a way to **organise** our code (Topic 3).

Classes and objects provide us with a way in which to **reuse** our code.

REMEMBER RUNNING TIME?

A way to calculate how long it takes for a piece of code to execute.

```
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);

   }

}
```

Can we wrap this code in a class in order to make this **functionality reusable** across different programs? Is it as **simple** as before? 😴

DESIRED DRIVER...

Sometimes it’s helpful to look at what we want the end product to look like when creating a class:

```
public class Driver {

    public static void main(String[] args) {

        RunningTimeCalculator copyOfRunningTimeCalculator = new RunningTimeCalculator();

        copyOfRunningTimeCalculator.recordCurrentTime();

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

        copyOfRunningTimeCalculator.printRunningTime();

    }

}
```

This is the first time we’ve seen the ***calling of multiple methods*** *in practice, but hopefully it is intuitive that this can be done.*

In this case, this is the functionality that should be provided to us by our reusable running time calculator.

THE LOCATION OF VARIABLES INSIDE CLASSES (1)

So far we have only seen variable declarations **inside** methods.

So we might approach writing our reusable running time calculator as follows:

```
public class RunningTimeCalculator {

   public void recordCurrentTime() {

      long currentTime = System.currentTimeMillis();

   }

}
```

We record the current time. We then want to reference this value at a later point in time, via a different method call.

Because these variables are **close** to the method, so to speak, we call them **local variables**.

LOCAL VARIABLES: VISIBILITY

Local variables can **only** be **referenced inside** the method in which they are declared.

```
public class RunningTimeCalculator {

   public void recordCurrentTime() {

      long currentTime = System.currentTimeMillis();

   }

   public void printRunningTime() {

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

   }

}
```

So we can reference the current time variable within this method.

But we cannot reference it within this one.

*(This code will **not** compile.)*

Therefore, local variables are typically used to store **temporary values** not values that need to be **used again**, like in this example.

ASIDE: SCOPE

The formal way to refer to the visibility of a variable is that variable's **scope**. When a variable is **in scope** it is in our **sight** and visible.

LOCAL VARIABLES: LIFETIME

As a local variable can only be referenced within a method, it is **only associated with that method**.

Therefore, more fundamentally, local variables only **exist** while the method in which they are defined is being **executed**. **What can we do?**

```
public class RunningTimeCalculator {

    public void recordCurrentTime() {

        long currentTime = System.currentTimeMillis();

    }

    public void printRunningTime() {

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

    }

}
```

Values assigned to this variable will disappear after the method finishes executing.

THE LOCATION OF VARIABLES INSIDE CLASSES (2)

We can, if we want to, **declare this variable inside the class itself**.

```
public class RunningTimeCalculator {

   private long currentTime;

   public void recordCurrentTime() {

      currentTime = System.currentTimeMillis();

   }

}
```

*By default, we keep fields **private** to the class. We will return to this idea.*

We call these variables **fields** (they correspond, somewhat, to **global variables**).

Why would we do this?

FIELDS: VISIBILITY

Fields can be referenced **anywhere inside** a class.

So, if we declare a current time field, we can **still assign that field** when we set the current time, **and** then also reference it in order to print the running time.

```
public class RunningTimeCalculator {

    private long currentTime;

    public void recordCurrentTime() {

        currentTime = System.currentTimeMillis();

    }

    public void printRunningTime() {

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

    }

}
```

*Indentation helps to confirm field **visibility**.*

*As with method calls, **where** a field appears in a class does not affect its visibility. The current time field could appear below the print time method, for example.*

FIELDS: LIFETIME (1)

As fields can be referenced anywhere within a class, they are associated with **the whole class** (rather than just a single method).

Therefore fields **exist** while the **class is being `executed'** (rather than while a single method is being executed).

What is the equivalent of a class being executed? When it is **copied into an object**.

Ergo, **fields exist while an object exists**.

```
public class RunningTimeCalculator {

    private long currentTime;

    public void recordCurrentTime() {

        currentTime = System.currentTimeMillis();

    }

    public void printRunningTime() {
```

*Values assigned to this field **won't** disappear after the method that assigns the value has finished executing.*

This is tough, let’s see this in practice with our current example…

Let's see how the lifetime of a field allows our run time calculator to operate as intended.

```
public class Driver {

    public static void main(String[] args) {

        RunningTimeCalculator copyOfRunningTimeCalculator = new RunningTimeCalculator();

        copyOfRunningTimeCalculator.recordCurrentTime();

    }

}
```

```
public class RunningTimeCalculator {

    private long currentTime;

    public void recordCurrentTime() {

        currentTime = System.currentTimeMillis();

    }

    public void printRunningTime() {

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

Let's see how the lifetime of a field allows our run time calculator to operate as intended.

```
public class Driver {

    public static void main(String[] args) {

        RunningTimeCalculator copyOfRunningTimeCalculator = new RunningTimeCalculator();

        copyOfRunningTimeCalculator.recordCurrentTime();

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

        copyOfRunningTimeCalculator.printRunningTime();

    }

}
```

The life of the object, and thus the fields it contains, and their stored values.

FIELDS: LIFETIME (4)

Let's see how the lifetime of a field allows our run time calculator to operate as intended.

```
public class Driver {

    public static void main(String[] args) {

        RunningTimeCalculator copyOfRunningTimeCalculator = new RunningTimeCalculator();

        copyOfRunningTimeCalculator.recordCurrentTime();

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

```
c class RunningTimeCalculator {

rivate long currentTime;

ublic void recordCurrentTime() {

  currentTime = 1234567890



ublic void printRunningTime() {
```

Le … te

```
public class RunningTimeCalculator {

    private long 1234567890 ;

    public void recordCurrentTime() {

        currentTime = System.currentTimeMillis();

    }

    public void printRunningTime() {

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

    }

}
```

*We’ve moved to a new place in the program, and the field still **retains** the value assigned previously.*

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

       copyOfRunningTimeCalculator.printRunningTime();

    }

}
```

OUTPUT: REUSABLE RUNNING TIME CALCULATOR

1. bash

```
Bob:topic4 Martin$ javac Driver.java
Bob:topic4 Martin$ java Driver
+------+
|Martin|
+------+
1
Bob:topic4 Martin$
```

ASIDE: DO WE EVER NEED LOCAL VARIABLES, THEN?

Can’t we just store everything in **fields** to be sure?

We could, there would be nothing wrong with that from a compilation perspective.

But stylistically this may be questionable.

- Does it make sense to have a variable **visible** in every method?
- Does it keep our class **neat** and **readable**?
- What about **temporary results** (e.g. from calculations) as mentioned previously?

OPEN QUESTION 1: WHAT IF WE WE USE BOTH VARIABLES?

```
public class RunningTimeCalculator {

    private long currentTime;

    public void recordCurrentTime() {

        long currentTime = System.currentTimeMillis();
```

What if we — intentionally, accidentally or out of intrigue — decided to use a local variable **as well as** a field, with the **same name**, what would happen? **Would this code compile?**

This code **does compile** because Java has **specific rules in this case to avoid name conflict**.

We will return to this idea.

OPEN QUESTION 2: RETURNING VARIABLES FROM A METHOD (1)

It is probably intuitively clear that we can **return variables** (or the result of manipulating variables) from a method, rather that just **literal** values, but we have not yet seen this in practice.

We have seen this process for method calls (see Topic 2, Slide 43).

```
printNumber(1);
```

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

As such, we could modify the `printRunningTime` method we saw in the previous example to return the running time rather than just printing it.

```
public long getRunningTime() {

    return System.currentTimeMillis() - currentTime;

}
```

OPEN QUESTION 2: RETURNING VARIABLES FROM A METHOD (2)

Whether you return a value (or a variable) from a method or simply printed it is, again (you guessed it), **a style choice**.

Personally I always try and return values from methods rather than printing them, and do as much printing in my Driver class as possible.

- When a method gives you a value back, rather than just printing it, you have the **option to do something else with that value**, so the class becomes more **reusable**.
- So far I've been opting for printing within a method, because we don't yet have all the syntactic tools we need to always return information from methods.

We will return to this idea.

OPEN QUESTION 3: WHAT IF WE ADD FIELDS TO OUR DRIVER CLASS?

Although we're using our Driver class simply to house our main method, it is still **just a class**.

Therefore, a valid question is, what do fields look like in this class, and how are they referenced?

The answer is pretty much in the same way, only remember our rule: **anything that is referenced from main must also be static**.

Thus, fields in the Driver class must be static.

```
public class Driver {

    private static int fieldInDriver;

    public static void main(String[] args) {

        fieldInDriver = 1;

    }

}
```

This might not be ideal, for (you've guessed it) reasons we will return to. 😴

Reason #3: Storing Complex Data

OBJECTS AND VARIABLES

Remember: fields are defined as part of a class (not a particular method) and therefore **exist while any objects of that class exists**.

Because of this, in their simplest form, objects are simply **multi-slot variables**.

We can use the fields inside an object to **store multiple pieces** of data, rather than a **single** piece of data, which is what we would do with a primitive variable.


```java
public class RunningTimeCalculator {

    private long currentTime;
```

Moreover, recall that a data type describes the **format** of what a variable will contain.

```
int myFirstIntegerVariable;
```

Because classes tell us the format of the fields inside an object it is sensible to view our own classes as **custom data types**.

This idea is reinforced by the **style** of an object declaration, which is very similar to that of a **primitive** variable declaration, as we have seen...

RELATIONSHIP: TYPES AND CLASSES; VARIABLES AND OBJECTS

Type	Variable	Value
`int`	`myFirstIntegerVariable`	`= 1;`

Describes the format and structure of

holds

Class	Object	Class Copy
`RunningTime Calculator`	`copyOfRunningTime Calculator`	`= new RunningTime Calculator();`

Describes the format and structure of

holds

DEFINING OUR OWN DATA TYPE (1)

Let's think about **why** we might want to define our own type, and why it might be useful.

When programming, it is often useful to store **pairs** of integers (e.g. a coordinate). But there's no **simple** data type that allows us to do this.

Let's define our own data type (a class) that allows us to store pairs of integer.

DEFINING OUR OWN DATA TYPE (2)

What will the **fields** of a Pair class look like?

Which **methods** will a Pair class need?

```java
public class Pair {

    private int valueA;
    private int valueB;

}
```

DEFINING OUR OWN DATA TYPE (4) - METHODS

When we want to store data in the different fields of an object, and when we want to get that data back again, we need to do so **via methods**.

```
public class Pair {

    private int valueA;
    private int valueB;

    public void setPair(int passedValueA, int passedValueB) {

        valueA = passedValueA;
        valueB = passedValueB;

    }

    public int getValueA() {

        return valueA;

    }

    public int getValueB() {

        return valueB;

    }

}
```

DEFINING OUR OWN DATA TYPE (5) - DRIVER

Not that I've dropped the `copyOf' object name prefix.

```
ic class Driver {

ublic static void main(String[] args)

  Pair pair = new Pair();
  pair.setPair(1, 2);
  System.out.println(pair.getValueA()
```

```
public class Pair {

    private int valueA;
    private int valueB;

    public void setPair(int passedValueA, int passedVal

        valueA = passedValueA;
        valueB = passedValueB;

    }

    public int getValueA() {

        return valueA;

    }

    public int getValueB() {

        return valueB;
```

1. bash

```
Bob:topic4 Martin$ javac Driver.java
Bob:topic4 Martin$ java Driver
1
Bob:topic4 Martin$ 
```

In this example, our methods are predominantly what we call **accessors** and **mutators**.

Accessors give us the value from a field **back**; mutators **change** the value in that field to something else.

These are the **simplest** type of methods.

But it's important to remember that methods can also perform **computation** based upon data, as in our first example.

```
public void printRunningTime() {
    System.out.println(System.currentTimeMillis() - currentTime);
}
```

ASIDE: A POTENTIAL OPTIMISATION? (1)

These variable names (parameters) are **informative**, but they aren't particularly **readable**.

It would actually be quite nice to be able to use **consistent names**.

Earlier, we discussed that this will compile (parameters are just **another form of local variable**), but naming our parameters in this way necessitates the following update.

How do we then determine **which** variable we want to assign to which?

```java
public class Pair {

    private int valueA;
    private int valueB;

    public void setPair(int valueA, int valueB) {

        valueA = valueA;
        valueB = valueB;

    }

    public int getValueA() {

        return valueA;

    }

    public int getValueB() {

        return valueB;

    }

}
```

ASIDE: A POTENTIAL OPTIMISATION? (2)

If we want to reference fields, but we have a (permitted) **naming conflict**, we can use the prefix **this**.

When we write the keyword **this**, we say to Java `**get *this* object**', and when we write a dot after this, we are also saying `**and look for the following:**'.

So, here we are saying, `get this object, and find the field valueA' (remember fields are part of the object).

This syntax allows us to differentiate between local variables and fields with the **same name**.

```java
public class Pair {

    private int valueA;
    private int valueB;

    public void setPair(int valueA, int valueB) {

        this.valueA = valueA;
        this.valueB = valueB;

    }

    public int getValueA() {

        return valueA;

    }

    public int getValueB() {

        return valueB;

    }

}
```

Because there's no conflict here, there is no need for the this notation.

*But some people choose to write this before **all** field references, as a style choice (I don't).*

In the laboratories, write your own data type. It doesn't have to be too complex, but make sure it has the following...

- The appropriate fields
- The appropriate methods

...also make sure you test your data type using a Driver class.

The idea of a class as a data type is **exciting**, because it suggests that we can have **multiple objects of the same class**, in the **same** program, in very much the same way that we can have **multiple variables of the same type**, in the same program.

In other words, our classes are not just reusable across different programs, they're reusable within the **same program**.

This is probably intuitively clear, but let's see what that looks like anyway...

MULTIPLE INSTANCES OF THE SAME CLASS

I can have as many Pairs as I like in my program.

```
public class Driver {

    public static void main(String[] args) {

        Pair pairA = new Pair();
        Pair pairB = new Pair();

    }

}
```

We call each object a different **instance** of the class.

CLASSES AS TYPES (2): MULTIPLE OBJECTS WITH DIFFERENT VALUES

More exciting still, if **variables of the same type can hold different values**, in the **same** program, then (the fields of) **objects of the same class can hold different values**, in the same program.

In other words, not only can we reuse classes in the same program, we can **manipulate the data in the objects of those classes** without affecting **other instances**.

MANIPULATING INDIVIDUAL INSTANCES (OBJECTS) OF A CLASS (1)

I can manipulate the values in one Pair instance, **without** affecting the value in another.

```
public class Driver {

   public static void main(String[] args) {

      Pair pairA = new Pair();
      Pair pairB = new Pair();

      pairA.setPair(1, 2);
      pairB.setPair(3, 4);

      System.out.println(pairA.getValueA());
      System.out.println(pairB.getValueA());

   }

}
```

MANIPULATING INDIVIDUAL INSTANCES (OBJECTS) OF A CLASS (2)

I can manipulate the values in one Pair instance, **without** affecting the value in another.

1. bash

```
Bob:topic4 Martin$ javac Driver.java
Bob:topic4 Martin$ java Driver
1
3
Bob:topic4 Martin$
```

MANIPULATING INDIVIDUAL INSTANCES (OBJECTS) OF A CLASS (3)

This can often be hard to wrap your head around, because we have only **physically written** one Pair class, and thus **one set** of the `valueA` and `valueB` fields.

But because we make a copy of a class when we use the new keyword, we also have **virtual** copies of the fields, which can be manipulated separately.

MANIPULATING INDIVIDUAL INSTANCES (OBJECTS) OF A CLASS (4)

```java
public class Driver {

   public static void main(String[] args) {

      Pair pairA = new Pair();
      Pair pairB = new Pair();

      pairA.setPair(1, 2 );
      pairB.setPair(3, 4 );

      System.out.println(pairA.getValueA());
      System.out.println(pairB.getValueA());

   }

}
```

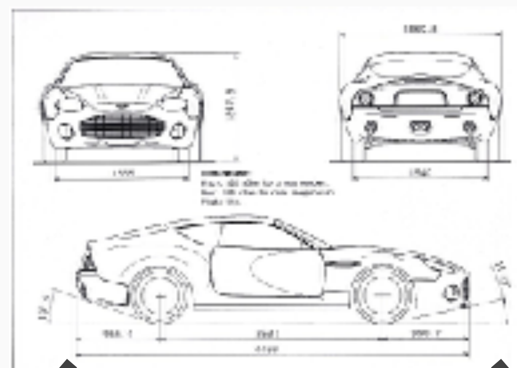
```java
public class Pair {

   private int valueA;
   private int valueB;

   public void setPair(int pass

      valueA = passedValueA;
      valueB = passedValueB;

   }
```

```java
public class Pair {

   private int valueA;
   private int valueB;

   public void setPair(int pass

      valueA = passedValueA;
      valueB = passedValueB;

   }
```

One pair class; **two** copies.

ANOTHER WAY TO THINK ABOUT CLASSES AND OBJECTS...

The ability for there to be multiple instances (objects) of the same class, each with **unique** values, is like having a **template** or a **blueprint** that can be **configured** in some way once the object it represents is **created**.



`Pair`

```java
Pair pairA = new Pair();
pairA.setPair(1, 2);
```

```java
Pair pairB = new Pair();
pairB.setPair(3, 4);
```

DEFINE YOUR OWN TYPE (2)

In the laboratories, play with the new type you've just made:

- Make **different instances** of this type.
- Change the **values** in this type.

ASIDE: A LANGUAGE WITHOUT OBJECTS AND CLASSES

How would we go about defining a Pair if we **didn't** have the expressivity of objects and classes?

```
public class Driver {

    public static void main(String[] args) {

        int valueAInPair1 = 1;
        int valueBInPair1 = 2;

        int valueAInPair2 = 3;
        int valueBInPair2 = 4;

    }

}
```

With no neat way to **collectively associate** these values, things become **messy**.

This necessitates **complex data types** like objects.

CLASSES AS TYPES (3): CUSTOM TYPES IN METHODS (1)

Because classes are types, we **can replace any references to primitive types, with references to classes**.

Remember our printNumber method?

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

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

}
```

We can replace a simple integer parameter with a parameter of Pair type, and extract an element from the pair in order to print it out.

```
/**
 * Prints the first value of the supplied pair
 * surrounded by a box.
 */
public static void printNumber(Pair pair) {

   System.out.println("+-------+");
   System.out.println("|" + pair.getValueA() + "|");
   System.out.println("+-------+");

}
```

CLASSES AS TYPES (3): CUSTOM TYPES IN METHODS (3)

We can also return objects of our class from a method, by altering the return type to match our custom type.

```
public class Driver {

    public static void main(String[] args) {

        System.out.println(getMeAPair().getValueA());

    }

    public static Pair getMeAPair() {

        Pair pairA = new Pair();
        pairA.setPair(3, 4);
        return pairA;

    }

}
```

This is one way we can ***return multiple values*** *from a method while still only returning one* ***thing****.*

DEFINE YOUR OWN TYPE (1)

In the laboratories, play **even more** with the new type you've just made:

- Create methods that accept this type as a **parameter**.
- Create methods that **return** objects of this type.

CLASSES AS TYPES (4): CUSTOM TYPES AS FIELDS (1)

Now that we've opened ourselves up to the prospect of **replacing any primitive type** with one of our own **class types**, it's natural that we can have methods inside a class that accept **types of other classes**, and **fields** that are able to **store objects** of this type.

```
public class TwoPairs {

    private Pair firstPair;
    private Pair secondPair;

    public void setFirstPair(Pair firstPair) {

        this.firstPair = firstPair;

    }

    public void setSecondPair(Pair secondPair) {

        this.secondPair = secondPair;

    }

    public Pair getFirstPair() {

        return firstPair;

    }

    public Pair getSecondPair() {

        return secondPair;

    }

}
```

Moreover, this now means that we can have methods inside a class that accept **types of that class *itself***.

```
public class Pair {

  private int valueA;
  private int valueB;

  public void setPair(Pair pair) {

    valueA = pair.getValueA();
    valueB = pair.getValueB();

  }
```

We'll return to this idea later in the topic.

WHY OBJECT-ORIENTATION: SO FAR WE KNOW...

In increasing order of importance (to keep the purists happy):

Classes and objects provide us with a way to **organise** our code (Topic 3).

Classes and object provide us with a way in which to **reuse** our code.

Objects provide us with an place in which to **store complex data**; classes define what that data looks like.

Reason #4: Expressivity

THE EXPRESSIVITY OF OBJECT-ORIENTED PROGRAMMING (1)

So far we have used classes to **model** very **technical** things: e.g. a running time calculator, a pair data type.

- We did this for **practical** purposes such as **organisation**, **reuse** and **data storage**.

This is important, as it shows us very early on how object-oriented programming might **actually be used**.

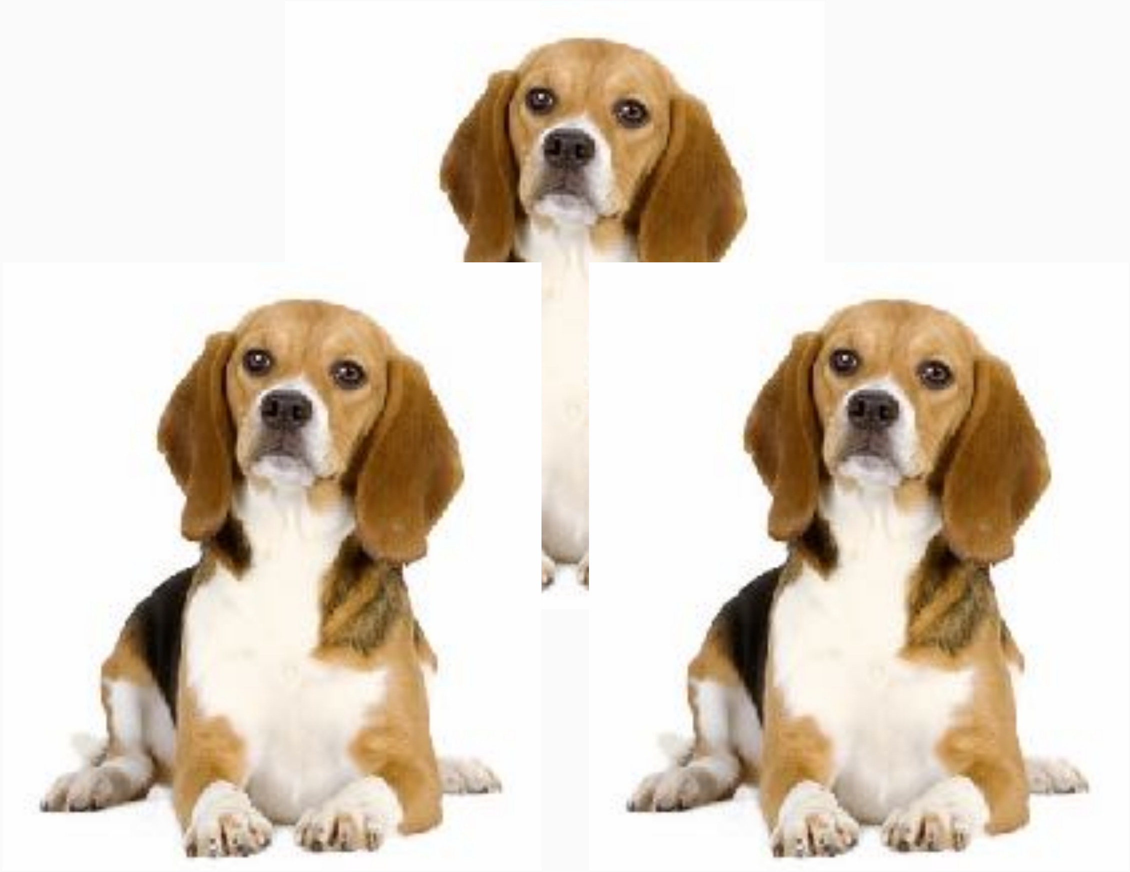
But it's also important to learn that object-orientation is **designed** to be a much **larger** and **expressive tool**.

In fact, we can write code that **models almost anything in the world** with object-orientation. Why?

Because object-orientation **comes from the real world**.

We all have classes in our mind; **shared concepts of things in the world**.

We see objects of these classes every day; **real instances of these concepts**.



The idea of a dog doesn't just exist in our minds, they exist all around us.

The fields of these classes are clear; **common features that we can all identify as a part of the shared concept**.

But fields often hold different data; things in the world are in different **states**.

CLASSES AND OBJECTS IN THE REAL WORLD (4): BEHAVIOUR

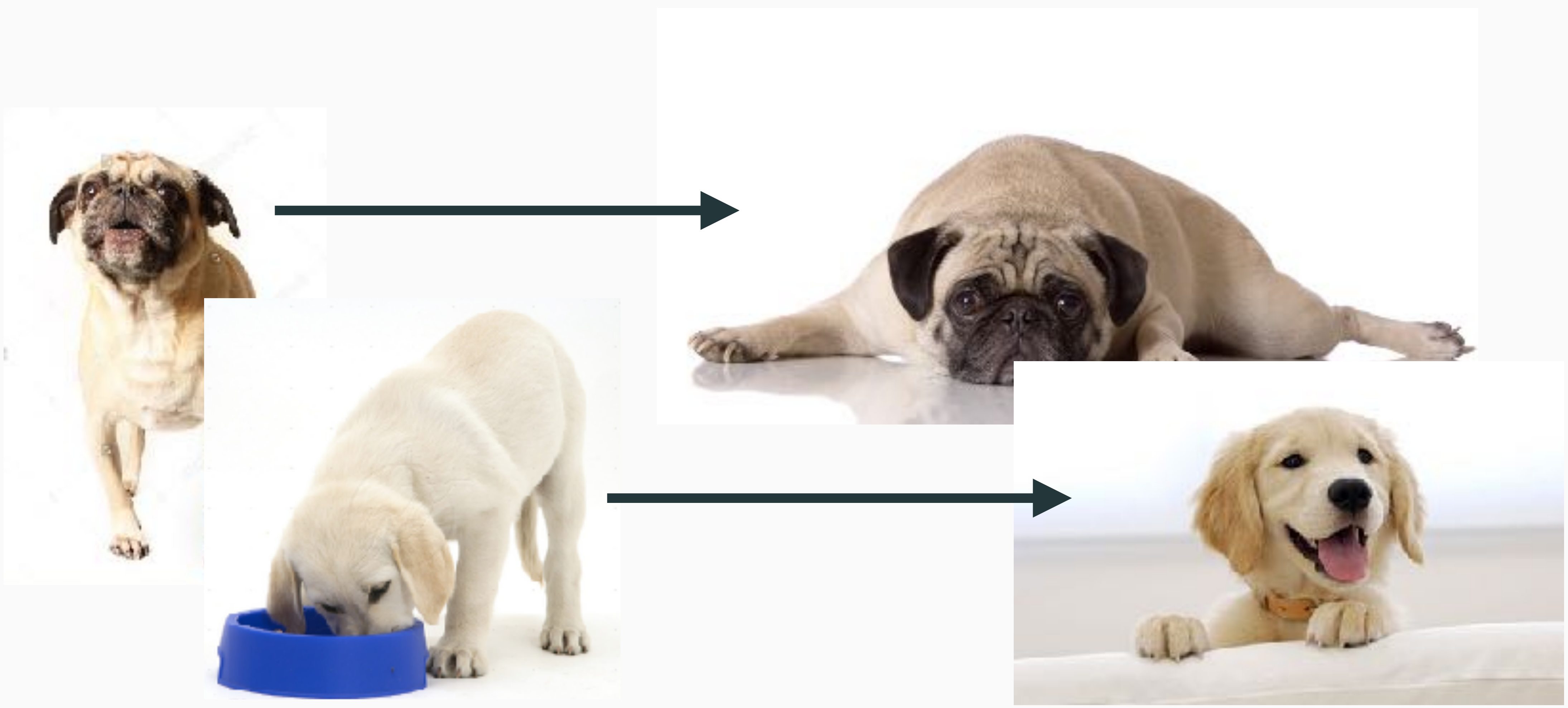
The methods of these classes are clear; **common actions that we can all identify as being exhibited by the shared concept**.

Things in the world exhibit **certain behaviours**.

CLASSES AND OBJECTS IN THE REAL WORLD (4): STATE AND BEHAVIOUR

It’s clear to see methods updating fields; **actions when performed have some effect on the object.**

Behaviours **affect** state.



This understanding shows us that not only can we model technical things (like pairs and run time counters), but we can **also** model **real world objects**. Why would we do this?

- **Simulation**
- **Games**
- **Learning**. Certain concepts are easier to understand when we aim to model real objects.

Let's try it out...


In order to revisit the mind bending notion of a class accepting parameters of itself (Slide 64), let's aim to model a **bank account**.

This is a good example, because it's a concept from the real world, but we can also imagine our code supporting a real banking system.

MODELLING A BANK ACCOUNT (1)

What's the **state** (fields) of a bank account?

Let's just think about a **balance** for now.

Which **behaviours** (methods) can a bank account exhibit?

Let's think about **deposit**, **withdraw** and **transfer**.

MODELLING A BANK ACCOUNT (2): FIELDS

```
public class BankAccount {

  private double balance;

}
```

We should definitely use a floating point format here as we'll be dealing with currency.

MODELLING A BANK ACCOUNT (3): DEPOSIT

*It is **not enough** to simply add a value to the amount, we must also **reassign the new value**, or the result will be **lost**.*

```
public class BankAccount {

  private double balance;

  public void deposit(double amount) {

    balance = balance + amount;

  }

  public void printBalance() {

    System.out.println(balance);

  }

}
```

Some more basic arithmetic, which we will return to, along with the idea of reassignment.

ASIDE: ADDING OR CONCATENATION?

We’ve now seen the **plus** (+) symbol used for two different things:

- Concatenation (placing things side-by-side in output)

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

- Adding numbers together.

```
balance = balance + amount;
```

How does Java know which to do? When the operands either side of the symbol are **numeric** (and the **only things present**), Java will perform arithmetic such as addition.

(Another) **Open question**: How does Java know when to concatenate? 😴

ASIDE: DEFAULT VARIABLE VALUES

```
public class BankAccount {

    private double balance;

    public void deposit(double amount) {

        balance = balance + amount;

    }
```

We first discussed this in Topic 2.

You'll notice here that, the first time we deposit an amount, it looks like we're adding this amount to nothing, as we haven't explicitly specified the **initial amount** in the balance field.

In reality this isn't strictly true, as primitive types **when they are used as fields** are given **default values**.

In this case, the default value given to a double value is **0.0**, so our code works as intended. In your laboratory session, investigate the default values prescribed to other primitive types when they are used as fields.

MODELLING A BANK ACCOUNT (4): TESTING DEPOSIT

```java
public class Driver {

   public static void main(String[] args) {

      BankAccount accountA = new BankAccount();

      accountA.deposit(10.0);

      accountA.printBalance();

   }

}
```

We'll investigate further exactly what is going on here shortly.

MODELLING A BANK ACCOUNT (5): WITHDRAW

```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;

   }

}
```

*I'm going to **shorten** this method down, because we know what it does (I may do this from time-to-time to focus attention on other methods).*

MODELLING A BANK ACCOUNT (6): TESTING WITHDRAW

```
public class Driver {

   public static void main(String[] args) {

      BankAccount accountA = new BankAccount();

      accountA.deposit(10.0);

      accountA.printBalance();

      accountA.withdraw(10.0);

      accountA.printBalance();

   }

}
```

Again, we'll examine what is going on here shortly.

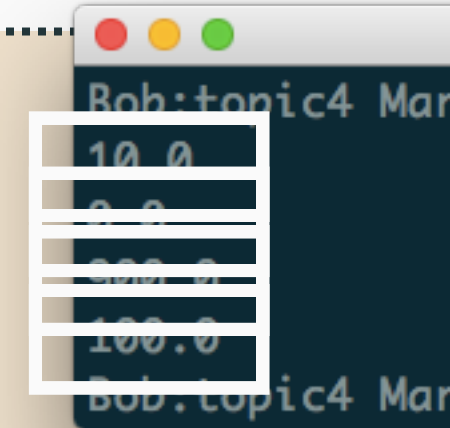
```java
public class BankAccount {

    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);

    }

}
```

We could just interact with the balance field directly, but why not call the method we already have?

MODELLING A BANK ACCOUNT (8): TESTING TRANSFER

```
s Driver {

ublic static void main(String[] args) {

  BankAccount accountA = new BankAccount(

  accountA.deposit(10.0);

  accountA.printBalance();

  accountA.withdraw(10.0);

  accountA.printBalance();

  accountA.deposit(1000.0);
                                          100.0    100.0
  BankAccount accountB = new BankAccount(

  accountA.transfer(accountB, 100);

  accountA.printBalance();

  accountB.printBalance();
                                          100.0    100.0
```

```
public class  accountA  {

    private double 900.0 ;

    public void deposit(double amount) {

        900.0 = 900.0 + amount;

    }

accountB lic void printBalance() { … }

  100.0 ic void withdraw(double amount) {

        900.0   900.0 - amount;

    }

    public void transfer(BankAccount otherAccount, d 100.e

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




```
public class  accountB  {

    private double   0.0

    public void deposit(double amount) {
```

The previous slide had **72** animations in it (!).

I’ve placed a video of these animations on **KEATS**.

In the lab, go through the video **animation by animation**, and see if you can write a simple sentence that describes what is going on each time something moves.

Keep this description for **revision**.

You should also **write out all the bank account code** and annotate it with **comments** (although you should be doing that anyway).

ASIDE: BACK TO PASS-BY-REFERENCE AND PASS-BY-VALUE (1)

In the previous topic, you should have discerned that Java is always **pass by value**.

```
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);

  }

}
```

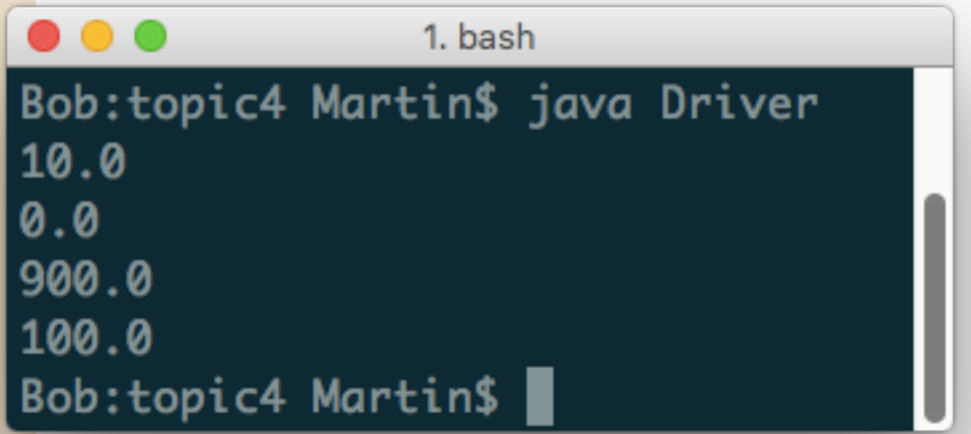
This will print '1'.

When a variable is passed to a method, it is effectively copied, such that any interactions with that variable have ***no effect on the original variable.***

ASIDE: BACK TO PASS-BY-REFERENCE AND PASS-BY-VALUE (2)

So, then, why is it the case that `accountB` retains the £100 transferred to it, if only a copy of `accountB` is passed to `accountA`'s transfer method?

```
        BankAccount accountA = new BankAccount();

        accountA.deposit(10.0);

        accountA.printBalance();

        accountA.withdraw(10.0);

        accountA.printBalance();

        accountA.deposit(1000.0);

        BankAccount accountB = new BankAccount();

        accountA.transfer(accountB, 100);

        accountA.printBalance();

        accountB.printBalance();

    }

}
```



ASIDE: OBJECTS IN MEMORY (2)

That's because, **at our current level of abstraction**, the rule is slightly different for objects.

When an object is passed to a method, any interactions with that object **will alter the original object.**

- **Reassigning the variable holding the class copy**, however, will **not** alter the **object**.

Next semester, when we look at objects in memory (i.e. **a lower level of abstraction**), the reasoning behind this should become clearer.

MODEL YOUR OWN REAL WORLD OBJECT

In a laboratory, go back and model the real world dog object we showed earlier as a class, or **even better**, pick your own object to model.

- This class should have appropriate fields
- This class should have appropriate methods.
- Methods should update the values in fields, where appropriate.
- You should test this class with a Driver class.

In increasing order of importance (to keep the purists happy):

Classes and objects provide us with a way to **organise** our code (Topic 3).

Classes and object provide us with a way in which to **reuse** our code.

Objects provide us with an place in which to **store complex data**; classes define what that data looks like.

Objects and classes provide a **natural** way to **conceptualise the world**.

Reason #5: Control

Once you start writing code on a more permanent basis, it's inevitable that your code will be **used by other people**.

This is particularly true if you write **reusable code**, like we have been doing so far.

The unfortunate thing about people is that they **often do things wrong**, break your code and then **complain**.

So, it's important to **control how your code is used**.

The object-oriented paradigm gives us several ways to control how our code is used.

CONTROL 1: DOCUMENTING YOUR CODE (1)

We use documentation to help us understand our **own** code, but remember it's also a useful tool in helping **other** people to understand our code.

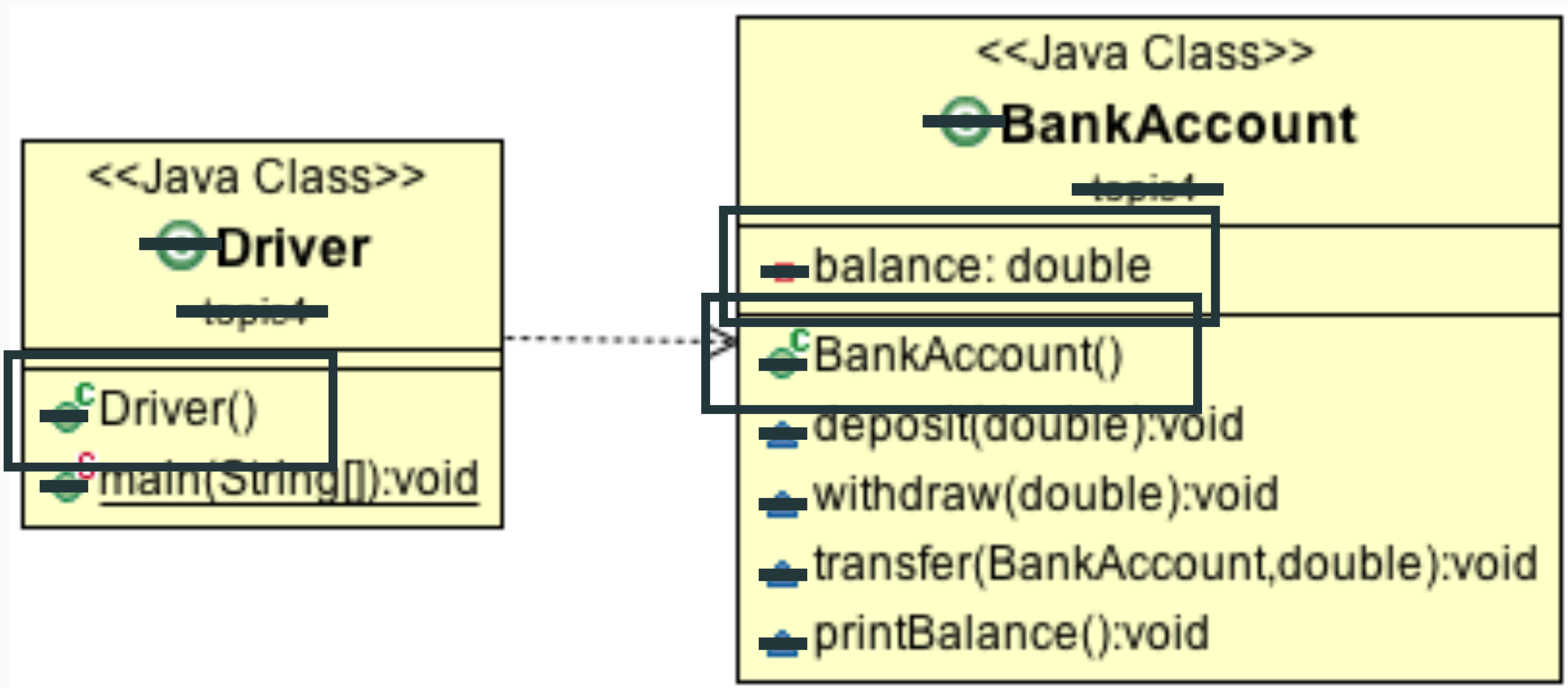
```java
/**
 * Prints the supplied number surrounded by a box.
 */
public static void printNumber(int num) {

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

}
```

Accessible code documentation is something that's **indirectly** supported by object-orientation (more later).

This doesn't stop people using your code **incorrectly**, but reduces the risk of it happening.

CONTROL 1: DOCUMENTING YOUR CODE (2): CLASS DIAGRAMS

We can also use the expressivity of our class diagram to show fields.



What are these?

BRACKETS WHEN INITIALISING AN OBJECT

```
public class Driver {

   public static void main(String[] args) {

      MartinPrinter copyOfMartinPrinter = new MartinPrinter();

   }

}
```

In Topic 3, I promised I would discuss what the brackets, **appearing after the name of the class** you wish to copy are for (we also saw these in the previous diagram). Now is this time.

This format looks very much like a **method call**.

That's because it is! Every class has a **hidden method** called a **constructor** which is called every time you write the **new** command.

```
public class Driver {

   public static void main(String[] args) {

      BankAccount accountA = new BankAccount();

   }

}
```

```
public class BankAccount {

   private double balance;

   public BankAccount() {


   }

   void deposit(double amount) {
```

The method labels match.

CONSTRUCTORS

These hidden methods are known as implicit **constructors** because they are called when you **construct** an object (make a copy of the code in a class).

Behind the scenes, they help setup the class for **use** (more later).

```
public class BankAccount {

  private double balance;

  public BankAccount() {


  }

  void deposit(double amount) {
```

Constructors don't have to be hidden, however, we can **actually write them into our classes**, and place code into them like a normal method.

Any code inside a constructor will therefore be called when the class is first **copied**.

Note how we differentiate a constructor from a **normal** method:

1. No return type

2. The same **name as the class**

```
public class BankAccount {

   private double balance;

   public BankAccount() {

      System.out.println("Oooh I'm being constructed");

   }

   public void deposit(double amount) {
```

PARAMETERS IN CONSTRUCTORS (1)

Apart from having **no return type** and having to have **the same name as the class** — to ensure they are called when the class is first constructed — constructors are **just normal methods**.

This means we can add **parameters to a constructor**, if we wish to.

```
public class BankAccount {

  private double balance;

  public BankAccount(int initialDeposit) {

    System.out.println("Oooh I'm being constructed");


  }
```

PARAMETERS IN CONSTRUCTORS (2)

What is the **effect** of this?

Remember that in order to alter Java's **execution order**, you typically have to **match** a certain pattern (i.e. match the **signature** of a method).

Thus, the current way in which we construct an object, in our bank account example, will **no longer work**, because we **do not use the correct pattern to match the constructor and thus create an object of the class**.

As such, we cannot make copies of or use the class **without** giving an initial deposit.

PARAMETERS IN CONSTRUCTORS (3)

```java
public class Driver {

  public static void main(String[] args) {

    BankAccount accountA = new BankAccount();

  }

}
```

```java
public class BankAccount {

  private double balance;

  public BankAccount(int initialDeposit) {

    System.out.println("Oooh I'm being constructed");

  }
```

ERROR: CONSTRUCTOR CANNOT BE APPLIED TO GIVEN TYPES

```
1. bash
Bob:topic4 Martin$ clear
Bob:topic4 Martin$ javac Driver.java
Driver.java:5: error: constructor BankAccount in class BankAccount cannot be applied to given types;
               BankAccount accountA = new BankAccount();
                                      ^
  required: int
  found: no arguments
  reason: actual and formal argument lists differ in length
1 error
Bob:topic4 Martin$
```

```java
public class Driver {

    public static void main(String[] args) {

        BankAccount accountA = new BankAccount(100);

    }

}
```

```java
public class BankAccount {

    private double balance;

    public BankAccount(int initialDeposit) {

        System.out.println("Oooh I'm being constructed");

    }
```

CONTROL 2: CONSTRUCTORS (1)

Therefore, when we place a constructor with parameters into our code, we are able to control how our class is used by **forcing** a user to provide values when an object is made of that class.

Why is this beneficial?

Because the code in a constructor is the first thing to execute when an object is made of a class, we can take the data supplied to the constructor and store it in one or more **fields**, to ensure that any methods that rely on data being in these fields do not cause **unexpected** or **erroneous** behaviour.

CONTROL 2: CONSTRUCTORS (2)

```
public class BankAccount {

    private double balance;

    public BankAccount(int initialDeposit) {

        balance = initialDeposit;

    }

    public void withdraw(double amount) {

        balance = balance - amount;

    }
```

Here, the assignment of the initial deposit will always be the first thing that happens in our class.

So when a withdrawal is made, we know that it is being taken ***after*** *an initial deposit is made.*

So, in theory, one cannot take money from an account before putting something in

In this scenario, of course, we assume that the user makes a sensible deposit (i.e. a positive number).

Open question: How would we discern whether a suitable deposit had been made?

CONTROL 2: CONSTRUCTORS (3)

As a more practical example, we could consider adding two parameters to a constructor in our Pair class.

```
public Pair(int valueA, int valueB) {
```

This would avoid a user calling an accessor without first specifying what the pair contains, and receiving a zero value.

```
Pair pair = new Pair();
pair.getValueA();
```

It also enables more efficient use of the class, as the setting of the values is effectively combined with the construction of the class.

```
Pair pair = new Pair(1, 2);
pair.getValueA();
```

There's also an element of **style** here.

The intelligibility of a class is increased if it's clear **what data that class needs to operate** by **just looking at the constructor**.

The constructor is also a good place to initialise fields, and stops us having to initialise **on the field itself**.

- Less chance of inefficient reassignment when programming.
- Stylistically nicer.

So, if we wanted to give everyone with a bank account £100 (rather than asking for an initial deposit), we would do it like this...

```
private double balance;

public BankAccount() {

   balance = 100;

}
```

*In general, **at this stage**, I would not do **anything** outside a method, except declare fields.*

...instead of this...

```
private double balance = 100;
```

(This is a debated topic.)

ASIDE: RELINQUISHING CONTROL FOR FLEXIBILITY (1)

There might be a case in which we want to offer the user the ability to pass information to an object when it is constructed, but not **force them to**.

For example, we might want to give the user the option to specify an initial balance when they create an account **or** to create an account without specifying this value.

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

    BankAccount account = new BankAccount();
    BankAccount secondAccount = new BankAccount(100);

}
```

ASIDE: RELINQUISHING CONTROL FOR FLEXIBILITY (2)

If we want to do this we can specify **multiple constructors**, each of which accepts different parameters, thus giving a user the option to construct an object of our class in **different ways**.

Each of our constructors has a different pattern, and thus there are different ways to match and create objects of the class.

*Specifically, we use an **empty** constructor. We can see this as **replacing** the **default** constructor we had previously, to remove any restrictions from the creation of objects of our class.*

```
private double balance;

   public BankAccount() {}

   public BankAccount(double deposit) {

      balance = deposit;

   }
```

Open question: Why is it ok for us to break our `methods with unique names' (Topic 3, Slide 14) rule here?

You've been patient, and now it's time to talk about public and private.

The meaning of these **access modifiers** should be intuitively clear, given that we've been **calling code in other classes** by creating objects:

- Anything that is public in a class can be **referenced** from **any other class that is used as part of your program**.
    - When we look at **packaging up** classes next semester, we will see how this rule changes.
- Anything that is private **can only be referenced within the class itself**.

Given this definition, it makes sense for classes to be **public** as we want to be able to use our classes in as many different places as possible.

But it's less clear why we want our fields to be private, especially when I show you the following piece of **bad but oh so tempting syntax**:

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

    Pair pair = new Pair();
    pair.valueA = 1;

}
```

We can avoid writing accessor and mutator methods if we make our fields public.

```
public class Pair {

    public int valueA;
```

Remember our dot syntax? As this allows us to reference any public identifiers, we can **also** access public variables.

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

    Pair pair = new Pair();
    pair.

}
```

```
public class Pair {

    public int valueA;
```

WHAT'S WRONG WITH PUBLIC FIELDS? (1)

It's not always the case that we want users to be able to set the values in fields **arbitrarily**.

For example, let's imagine that we implement a **pin** system into our bank account, such that a user must supply the right pin before being allowed to make a withdrawal:

WHAT'S WRONG WITH PUBLIC FIELDS? (2)

```
public class BankAccount {

   public  double balance;

   public void withdraw(double amount, int pin) {

      balance = balance - amount;

   }
```

(This method is incomplete)

If our balance field is public, users can simply interact with it directly, circumventing the pin, and do what they wish.

```
BankAccount account = new BankAccount();
account.balance = -10000;
```

Instead, we want to **force all requests to alter the values in fields to come through the parameters of methods.** We do this by making fields private.

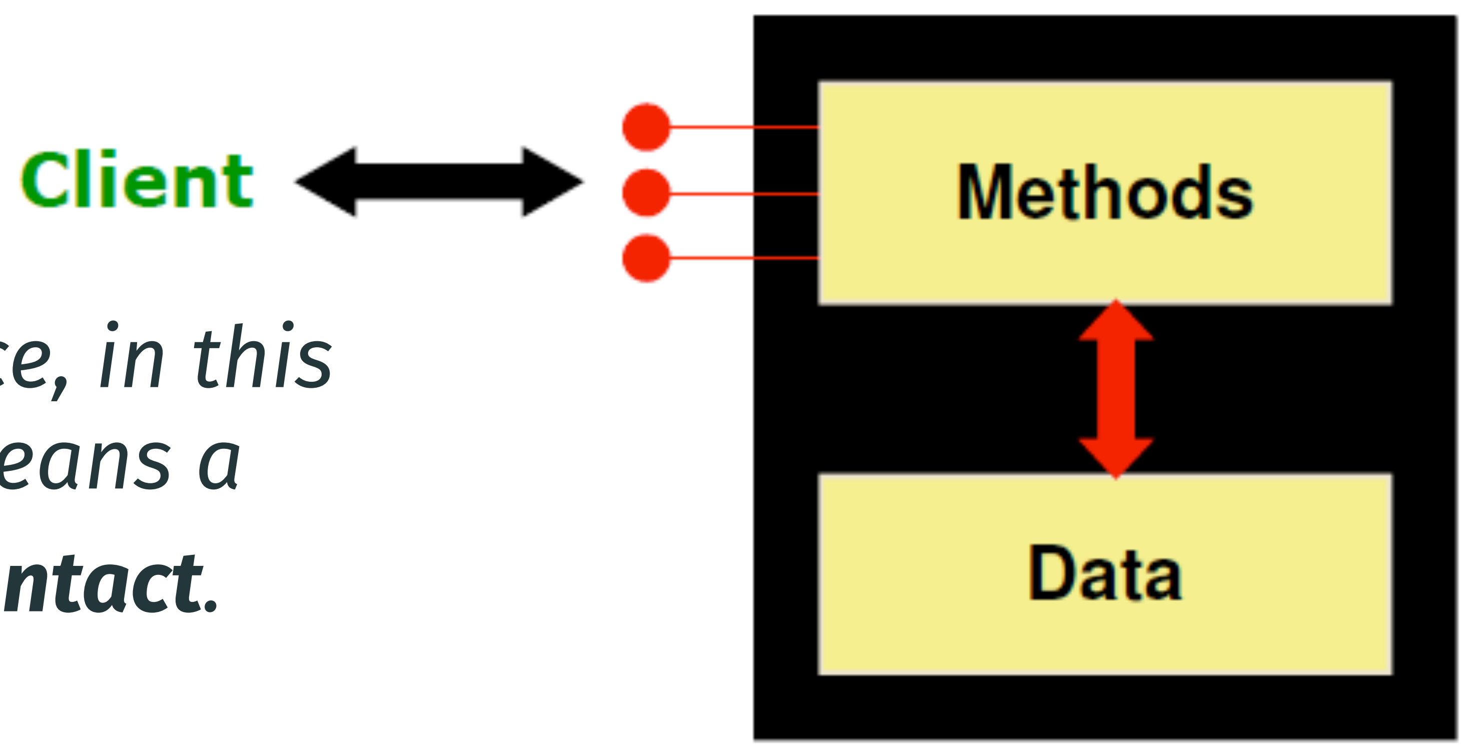
In this way, we can validate that a request is **sensible** (e.g. the correct pin is supplied) and doesn't adversely affect the intended operation of our class, **inside the method itself**.

VALIDATING PARAMETERS INSIDE A METHOD


```
public class BankAccount {

   private double balance;

   public void withdraw(double amount, int pin) {

      Ensure pin is correct before:
           balance = balance - amount;
      If not, print:
           System.out.println("Ah ah ah, you didn't say the magic word.");

   }
```

But we’re currently missing the logic to do this.

When we ensure users can only change the data (state) of an object (of a class) through methods, we say that our class has a **well defined interface**.



*An interface, in this context, means a **point of contact**.*

The process of **hiding** data from our user, and carefully selecting which methods are available to manipulate that data, is known as **encapsulation**.

ASIDE: OPEN QUESTIONS ABOUT ACCESS MODIFIERS?

What about private in unexpected places?

- A private **class**
- A private **constructor**

Are these the **only** access modifiers available to us?

What happens if we **don’t** write an access modifier?

We will return to these questions, mostly in the second semester.

WHY OBJECT-ORIENTATION?

In increasing order of importance (to keep the purists happy):

Classes and objects provide us with a way to **organise** our code (Topic 3).

Classes and object provide us with a way in which to **reuse** our code.

Objects provide us with an place in which to **store complex data**; classes define what that data looks like.

Objects and classes provide a **natural** way to **conceptualise the world**.

Object-orientation allows us to **control how our code is used**.

Topic 4: Why Object-Orientation?

Programming Practice and Applications (4CCS1PPA)

Dr. Martin Chapman

Thursday 13th 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.