Key Object Oriented Language Constructs

COMSM0086

Dr Simon Lock & Dr Sion Hannuna

Today's Aim

The aim of today's session is to further explore some of the key OO concepts mentioned previously

Note: for your convenience, fragments of this lecture will appear embedded within the workbook tasks
Allows you to refresh memory when attempting tasks

There are also additional slides/video in the workbooks
Be sure to view these: you'll not have seen some before

Warning

We are about to see a lot of terminology !

You won't be expected to remember it immediately
It will become more familiar over the next few weeks

Any terms in 'single quotes' are official terminology
Anything in "double quotes" are my own informal words
See the next slide for some examples...

Classes

A 'Class' is a module of source code in Java
For time being, think of it as a "fancy" struct from C

```
class Counter
{
    // Code for class goes here !
}
```

We advise a 1-1 mapping between 'Class' and 'file'
Name of a class should match the name of the file
So the above would be in a file called Counter.java

Objects and Instances

'Classes' are static source code that you've written

A 'Class' describes a particular type of 'Object'

'Instances' of a class are created at run time

These live dynamic things are the 'Objects' in OOP

Process of creating live 'Objects' from 'Classes' is:

'Instantiation'

Java Types

Java's 'Primitive' data types will be relatively familiar
(int, char, boolean, float etc. - note lower case !)
These primitives are just simple data (just like C)

We can however ALSO use Classes as data types !
These are more sophisticated: Data AND Behaviour

Java provides the concept of an 'array' (just like C)
These can contain either 'Primitives' or 'Objects'
Arrays are homogenous - all elements of same type
(well, kinda ;o)

Attributes

A Class has a number of data fields or 'Attributes'

These are global to (accessible within) that class

Importantly NOT (usually) global to whole program

This is that notion of 'Encapsulation'

For example:

```
class Counter
{
   int count = 0;
}
```

Methods

Classes have a bunch of "functions" called 'Methods'

```
class Counter
{
   int count = 0;

   void increment() {
      count++;
   }
}
```

Such methods are "attached to" the Class

Methods are called ON a specific instance of the Class

```
Counter clubCounter;
clubCounter.increment();
```

Difference Between Functions & Methods

'Methods' are "tied to" a particular Class/Object
'Functions' are just "floating around" in namespace

In C you just call a function in isolation:

```
printf("Hello");
```

In Java you call a method ON a particular Object:

```
out.printf("Hello");
file.printf("Hello");
serial.printf("Hello");
```

Standard Naming Conventions

Classes: camel case, starting with a capital

`String, RallyCar, FlyingRobot, WarmBloodedAnimal`

Objects/Instances: camel case, starting with lower case

`colour, carCounter, termTimeAddress, fluffyBunny`

Methods: camel case, starting with lower case

`draw, incrementCounter, getAddress, strokeBunny`

Don't use underscores (this isn't Python)

Constructor Methods

Special methods exist to initialise instances of Class

These 'constructors' have same name as the Class

```
class Counter
{
   int count = 0;

   public Counter() {
      count = 0;
   }
}

Counter myCounter = new Counter();
```

Notice: no return type is defined for a constructor !

'public' so that it can be called from anywhere

Multiple Constructors

We can write a simple constructor (default values):

```
public Counter() {
    count = 0;
}
```

Or complex ones, where we pass in some value(s):

```
public Counter(int startValue) {
    count = startValue;
}
```

Providing more than one method with the same name in this way is referred to as 'Overloading'

Example Class

Java has a String class to store & manipulate text
Inside there's some kind of array to store characters
(That's Abstraction and Encapsulation in action !)

Also a bunch of methods that "do things to" the text:
- length: gives the number of characters in the text
- charAt: gives you the char at a particular position
- contains: tells you if string contains a sequence
- toLowerCase: converts string into lower case text
- substring: splits off a chunk of the string

Example Objects

We can create Objects (instances) of the String class
Each with a different character sequence inside:

```
String firstUnit = new String("COMSM0103");
String secondUnit = new String("COMSM0204");
```

There is (just for the String class) a shorthand:

```
String thirdUnit = "COMSM0305";
```

Provided because creating Strings is soooo common

Inheritance

Powerful feature to reuse & extend existing code

Take the String class for example...
Currently it is just plain text
But what if we wanted to add text styling ?
(Bold, Italics, Underline etc.)

We could 'extend' the basic String Class,
making use of all of the existing methods,
but also adding in some extra ones of our own...

StyledString Class

```
class StyledString extends String
{
    void setUnderlined(boolean underline) {
        // Some code goes in here !
    }

    void setItalics(boolean italic) {
        // Some code goes in here !
    }
}
```

Overall Outcome

We've only added two new methods to StyledString:

- `setUnderlined`
- `setItalics`

But, because we are 'extending' String, we also get:

- `length`
- `charAt`
- `contains`
- `toLowerCase`
- `etc.`

All for free !

Overriding

Adding features to a 'child' (the extending class)

Is often achieved by *replacing* an existing method

With *a new one* containing additional features

We write a new method, with a duplicate name

This replaces original method of the 'parent' class

This is called 'Overriding'

'Method in child class overrides the method of parent'

Method 'Chaining'

If we are REALLY clever...
We can REUSE the method of 'superclass' (parent)
And then "tack on" the extra features at the end:

```
public String toString()
{
  String text = super.toString();
  if(isUnderlined) text = "\033[4m" + text + "\033[0m";
  if(isItalics) text = "\033[3m" + text + "\033[0m";
  return text;
}
```

Tags are a bit like: text = "" + text + "";

Polymorphism

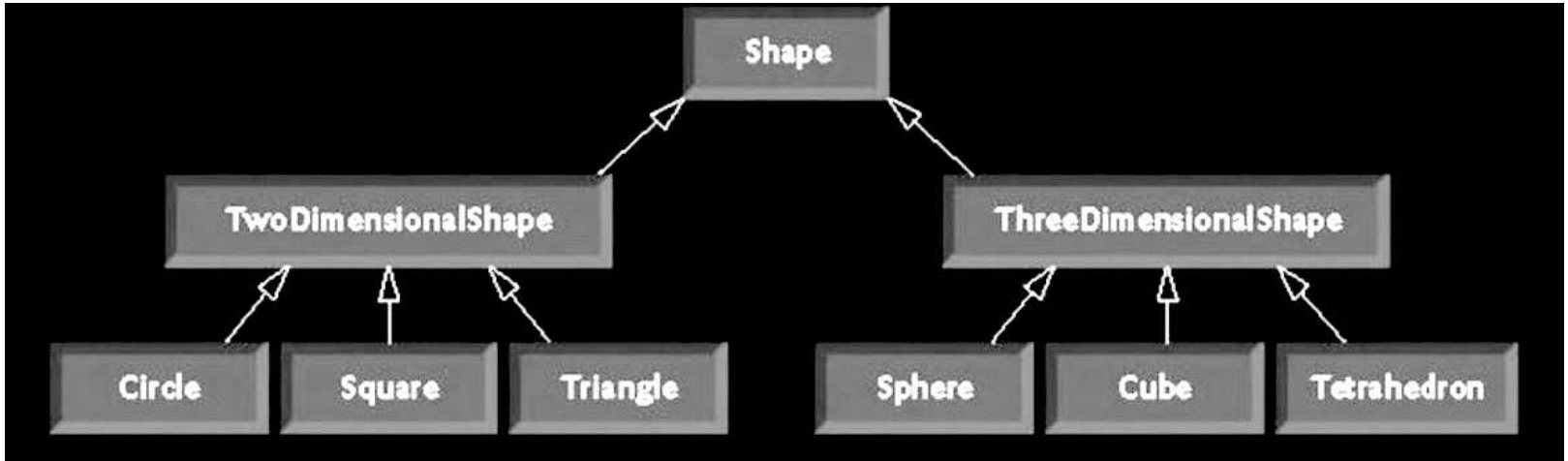
It is possible to "do things to" families of classes

Without caring exactly which class we are doing it to

For example, we can "do things to" ALL 3D shapes

(Such as getting volume, rotating in 3 axes etc.)

Without caring if it is a Sphere, Cube or Tetrahedron



Polymorphism in Action

What if we don't know what kind of String to expect ?

It might be a String, StyledString, ColourString...
Don't want to write a different method for each type

Luckily we don't have to !
We can just write a "general purpose" method:

```
public String correctSpelling(String text) ...
```

And this will be able to operate on ANY kind of String
(Anything that is a String or any 'subclass' of String)

Encapsulation

If this were C, we could just "reach in"
and set the style boolean attributes like so:

```
myString.isUnderlined = true;
myString.isItalics = true;
```

But this is very dangerous !
We don't know how the object uses these attributes
How can we be sure that it is safe to change them ?

DVD Collection Example

Imagine that you had a collection of DVDs
Your friends can come and ask to borrow one
You can keep track of who has which disc
(You might even record loans in a relational DB ;o)

But what would happen if people could just walk in,
and take them without even asking !
There would be chaos !!!
You wouldn't know where anything was :o(

Accessor and Mutator methods

We may wish external objects to access internal data
But must ensure this is done in a controlled manner

Can be achieved by providing additional methods...
'Accessors' ("Getters") and 'Mutators' ("Setters"):

```
boolean isUnderlined = false;

void setUnderlined(boolean underline)... // Setter
boolean getUnderlined()...               // Getter
boolean isUnderlined()...                // Alternative
```

Controlling Access

We can define attributes and methods to be:

- 'public' access from any location (avoid variables)
- 'private' only inside class in which they're defined
- 'protected' inside the class OR any subclass
  (also from any class in the same package/folder)

If you don't specify any of the above, you will get:
"semi-protected" (let's NOT even go there just now)
Just use 'public' or 'private' (as appropriate) for now

This probably doesn't make much sense at the moment !
Things *should* become clearer during the workbook

Review video fragments of this lecture as needed
They are embedded in the workbook for this purpose !

At some point you'll be able to talk fluent "java-speak"
(and will wonder how you ever managed before ;o)