Key Object Oriented Language Constructs

COMSM0103

Dr Simon Lock & Dr Sion Hannuna

The aim of this session is to further explore some
of the key OO concepts introduced previously

WARNING:
There is some overlap with videos in the workbook
Mainly because we're still just covering fundamentals
This is a chance for clarification if things are unclear

There will be more divergence in future weeks
Topics become diverse (and workbooks more complex)

Classes

A class is a "module" of source code in Java
To start with, think of it like a "fancy" struct from C
Typically a 1-1 mapping between "Class" and "file"
Name of a class should match the name of the file

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

A Class describes a particular type of Object
Instances of that type are then created at run time

Java Types

The _majority_ of data types in Java are Classes
The exception to this are "Primitive" types
(int, char, boolean, float etc. - note lower case !)

Primitives are just simple data (the same as in C)
Everything else in Java is a Class/Object !

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) the class

Importantly NOT (usually) global to whole program

Remember the notion of "Encapsulation" ?

For example:

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

Methods

Classes have a bunch of functions called "Methods"

```
class Counter
{
   int count;

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

Such Methods are "attached" to that Class

They are called "on" a specific instance of that Class

```
Counter carCounter;
carCounter.increment();
```

Difference Between Functions & Methods

Methods are tied to a particular object
Functions are just "floating around"

In C you just call a function in isolation:

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

In Java you call a method ON a particular Object:

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

Standard Naming Conventions

Classes: camel case, starting with a capital

`String, RallyCar, FlyingRobot, WarmBloodedAnimal`

Objects: 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 create instances of a Class
These "constructors" have same name as the Class

```
class Counter
{
   int count;

   public Counter() {
      count = 0;
   }
}

Counter myCounter = new Counter();
```

Notice: no return type defined for a constructor !

Multiple Constructors

We can have simple constructor with default values:

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

Or complex ones, where we pass in some values:

```
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
A bunch of methods to "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 to lower case version
- substring: splits off a chunk of the string

All packaged up into a nice neat bundle - a Class !

Example Objects

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

```
String unitCode = new String("COMSM0103");
String unitCode = new String("COMSM0204");
```

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

```
String unitCode = "COMSM0305";
```

Provided because creating Strings is so common

Inheritance

Allows us 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 can "extend" the basic String Class,
making use of all the existing methods,
but also adding in some of our own...

StyledString Class

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

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

Overall Outcome

We have only added two new methods:

- `setUnderlined`
- `setItalics`

But because we extended 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

This is called...

Overriding

Method Chaining

If we are really clever...
We can *reuse* the method of super (parent) class
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;
}
```

Using StyledString

```
public static void main(String args[])
{
    StyledString message = new StyledString("Hello");
    System.out.println(message.toString());
    message.setUnderlined(true);
    message.setItalics(true);
    System.out.println(message.toString());
}
```

StyledString
run-demo

Polymorphism

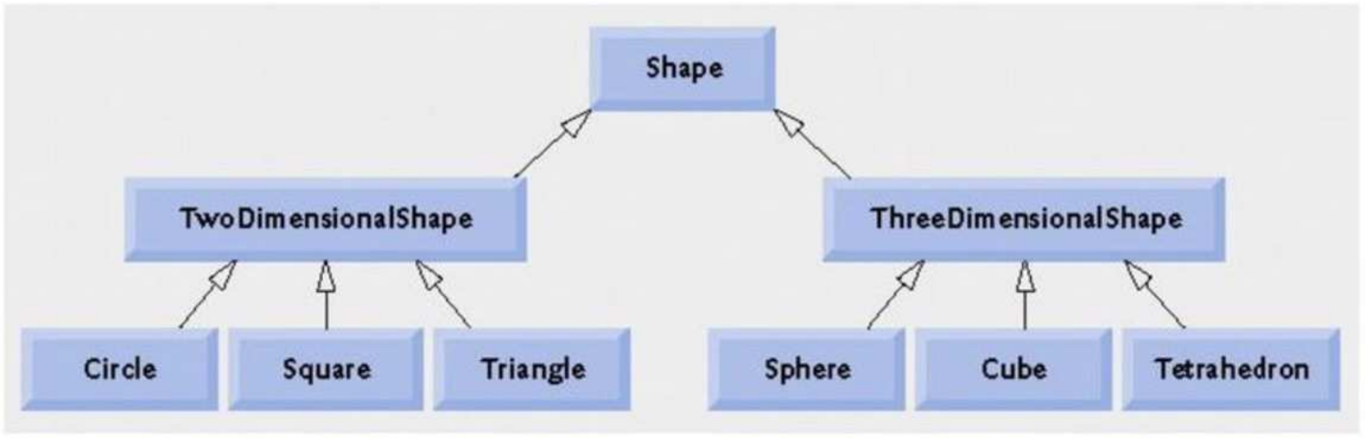
It is possible to "do things" to families of classes

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, tetrahedron



Polymorphism in Action

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

It might be a plain String or maybe a StyledString

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 void spellCheck(String text) ...
```

And this will be able to operate on ANY kind of String

(Anything that is a String or sub-class 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 !
If we don't know how the object uses them...
How can we know it is safe to change them ?

DVD Collection Example

Imagine you had a collection of DVDs

Your friends can come and ask to borrow one

You can keep track of who has which disk

But what would happen if people could just walk in,

and take them without asking !

There would be chaos !!!

You won'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) ...
boolean getUnderlined() ...
boolean isUnderlined() ...
```

Controlling Access

We can define attributes and methods to be:

- Private: Only inside class in which they are defined
- Public: Access from any location (avoid variables)
- 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" (lets not go there)
Just use public or private (as appropriate) for now

Warning: Don't try this at home !

The concept of a Styled String is just an example
A simple and understandable concept for this lecture

Text styling tags are not universal to all platforms
Java is supposed to be platform independent

There is also a technical reason why we can't do it
But we haven't learnt enough yet to understand why

This week's workbook

https://github.com/drslock/JAVA2022

The first 6 tasks are all fairly straight-forward
They just require you to grasp some key concepts...
Then write some little bits of code to apply them

Task 7 will however take more time
It involves performing some fiddly computations
Also requires you to make use of a test framework

TriangleTests
TriangleTestViewer