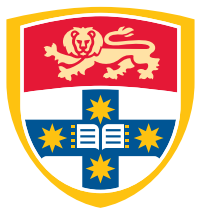
INFO1103: Introduction to Programming

School of Information Technologies, University of Sydney



Lecture 24: More inheritance abstractions

Abstract classes and Interfaces

Abstract classes

A class which cannot become an Object

Designing more Classes

Let's design classes based on geometrical shapes

What are attributes of a square?

What would be the data types?

Create the class to represent these attributes

Designing more Classes

What are attributes of a circle?

What would be the data types?

Create the class to represent these attributes

Generic Shape

Use inheritance to define Shape as a super class of Circle and Square

There are common attributes/methods identified among them:

- `String name`
- `double scale`, how big is it?
- is the shape convex or concave? `boolean` or `enum`?
- `_____ calculatePerimeter ( _____ )`
- `_____ calculateArea ( _____ )`

Implement the "calculate the area of a shape" methods in each classes `Circle`, `Square` and `Shape`

Create three new objects `Circle`, `Square` and `Shape`. Calculate the area

Abstract classes

This case illustrates a situation where it doesn't make sense to have a super class that can be *instantiated*.

We can make a `Shape` object, but why would it be useful?

Suppose you now have to "draw the shape", _____ `draw` (_____)

- trickier to know how to implement for Dodecagon or ellipse,
- what is it supposed to do if it is just a Shape?

Abstract classes

To prevent Shape from being *instantiated*, we make it an abstract class by using the abstract keyword appropriately

You can think of an abstract class as an "incomplete class": it might have some parts of it defined, but not enough to work

```
public String getName() { return name; }
public boolean isConvex() { ...  }
public abstract double calculatePerimieter();
```

Shape is an abstract class

So shape should be an `abstract class`:

```
1 public abstract class Shape {
2 ...
3 }
```

Shape is an abstract class

So shape should be an `abstract class`:

```
1 public abstract class Shape {
2 ...
3 }
```

Let's give it a method for illustration.

```
1 public abstract class Shape {
2     public abstract double calculateArea();
3 }
```

While it doesn't make sense for an *abstract* shape to be able to calculate its area, it should definitely be the case for *concrete* shapes to do so.

Properties of abstract classes

`abstract class`

is a class

cannot be instantiated

can have methods

can have variables

can have a constructor

`Shape.calculateArea()`

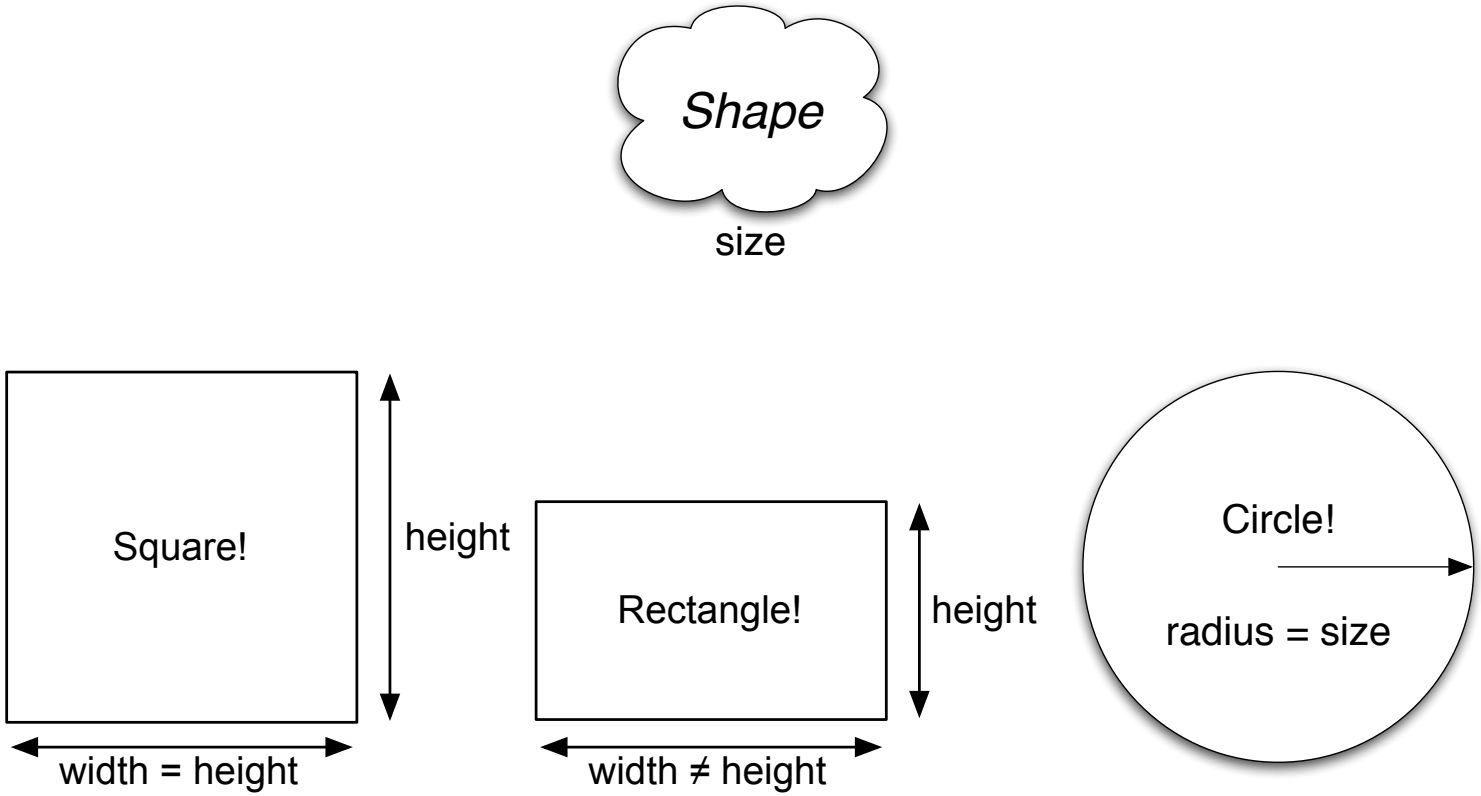
`calculateArea()` will return a `double` value of the area of the shape. It's easy to work out the area of many shapes: rectangle, square, circle, triangle for instance.

Shape		**Area**
square	h^2	$height^2$
rectangle	wh	$width \times height$
triangle	$bh/2$	half $base \times height$
circle	πr^2	$\pi \times radius^2$

A hierarchy of shapes

I will now go through a set of classes beginning with Shape, and them moving on to

Shapes!


Shape
size
Square!
height
width = height
Rectangle!
height
width ≠ height
Circle!
radius = size

Example of abstract classes: geometry

```
1  public abstract class Shape {
2
3      protected String name;
4
5      protected double size;          // all subtypes can access size
6
7      public Shape(double size) {     // constructor just sets size
8          this.size = size;
9      }
10
11     public Shape(double size, String str) {
12         this(size);          // calls Shape constructor with size arg.
13         name = str;
14     }
15
16     // subtypes MUST implement this:
17     public abstract double calculateArea();
```

Example of abstract classes: `geometry` (cont.)

```
18
19      public void resize(double factor) {
20          // this method is common to all Shapes
21          size *= factor;
22      }
23  }
```

This abstract class cannot be instantiated but it has some concrete members.

If any method is marked `abstract` then the whole class must be marked `abstract`, but it's ok to mark a class `abstract` even if *nothing* inside it is marked `abstract`.

Square.java

```java
public class Square extends Shape {

    public Square(double size) {
        super(size, "square");  // this constructor has to be here
    }

    @Override
    public double calculateArea() {
        return size * size;
    }

}
```

The square takes one argument and sets that as the size.

Rectangle.java

```java
public class Rectangle extends Shape  {

    private double aspectRatio;       // ratio height:width

    public Rectangle(double size) {
        super(size, "rectangle");     // calls super constructor
        aspectRatio = 1.0;
    }
    public Rectangle(double size, double ratio) {
        super(size, "rectangle");     // calls super constructor
        aspectRatio = ratio;          // sets aspect ratio
    }

    @Override
    public double calculateArea() {
        return size * size * aspectRatio;
    }

}
```

Circle.java

```java
public class Circle extends Shape {

    public Circle(double size) {
        super(size, "circle");          // set the size and name
    }

    @Override
    public double calculateArea() {
        return Math.PI * size * size; // treating size as radius
    }

}
```

The constructor uses the single argument as the radius of the circle.

Interfaces

More abstract than an abstract class, has no data

Interface

An interface is a type closely related to abstract class. It defines a *specification* of all the possible operations (methods) that are used. e.g. Shape.draw()

However, an interface does not describe any implementation of how those operations work. unlike Shape.getName()

Interfaces are a way to make a kind of contract between sections of code.

Interface: define a specification

Defining an interface is uses the keyword `interface` in Java:

Here's an interface now:

```
1 public interface MyInterface {
2     // ^^^^^^^^^^ different type here
3
4     public String getString(); // <- no method body
5     public void prime10();
6     public double getMax10();
7 }
```

- There is no constructor
- There is no dynamic data stored (only constant values i.e. static final)
- Indicates only method signatures (name, argument types, return type) but not method body

`implements`: Make a class use the interface

If we want a class to do all the things with the interface, we use `implements`

```
1 public interface MyInterface {
2     public String getString();
3     public void prime10();
4     public double getMax10();
5 }
```

```
1     public class MyClass implements MyInterface {
2       //                  ^^^^^^^^^^
3
4     }
```

Now we have to implement the methods as required by this interface (specification)

Make a class conform to the interface specification

The interface should be well documented to describe what is the expected behaviour when calling that function.

```java
public interface MyInterface {
    public String getString(); // returns the text "burger"
    public void prime10();  // print the first prime > 100
    public double getMax10();  // return max of 10 random double
}
```

```java
public class MyClass implements MyInterface
{
    public String getString() {
        return "burger"; // as specified
    }
    public void prime10() {
        // do as interface says...
    }
    public double getMax10() {
        // do as interface says...
    }
}
```

classes must implement all interface methods

Interfaces are a way to make a kind of contract between sections of code.

When a `class` `implements` an `interface`, then all the methods listed in `interface` must be defined in the `class`

Put otherwise, if `class` Student `implements` KungFu, then the Student class must have all the methods of KungFu, such as punch() and block() methods

By having all methods implemented, we are guaranteed that any two interacting classes using interfaces, they will match their specification.

Many things can implement an interface

Let's have a very simple interface as an example. Everything that implements this must have the method `print` as described below:

```
1 public interface Printable {
2     public void print(); // prints the object to the console
3 }
```

Implementing `Printable`

```java
public class Complex implements Printable {
    private double im, re; // this is allowed
    public Complex(double x, double y) {
        re = x; im = y; // so is this but don't do it in your work
    }
    @Override
    public void print() {
        System.out.print("(" + re + "," + im + "i)");
    }
}
```

```java
public class Message implements Printable {
    private String msg;
    public Message(String s) {
        msg = s;
    }
    @Override
    public void print() {
        System.out.print(msg);
    }
}
```

@Override?

The `@Override` keyword is an indicator to the compiler that I am *overriding* a method that was declared in another type, in this case, in the interface `Printable`.

The compiler also checks if you are trying to override a method that is not defined in a super type
This will give a compile error:

```
1    @Override
2    public void foo() {}
```

Using interfaces

As the interface is a type, we can declare variables with that type:

```
1    public static void main(String[] args) {
2        Printable message = new Message("Hello, World!");
3        Printable complex = new Complex(1.1, 0.4);
4        message.print();
5        System.out.println();
6        complex.print();
7        System.out.println();
8    }
```

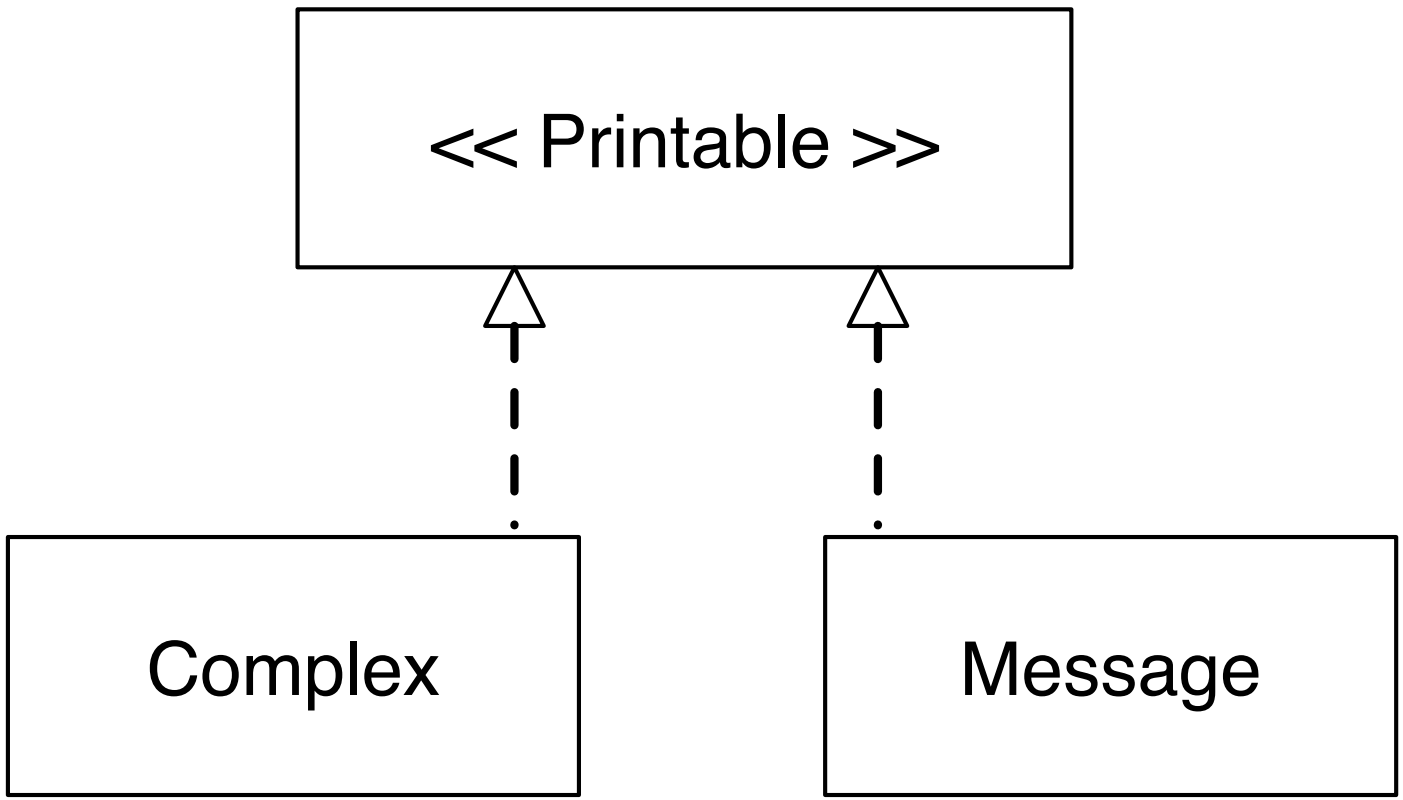
yielding

```
Hello, World!
(1.1,0.4i)
```

Note above that the type on the left is always Printable, but the types on the right are Message and Complex.
We shall see what this is all about very shortly...

Subtypes

If a class A implements an interface B then we say A is a *subtype* of B.
We often draw the relationships between types with Unified Modelling Language or UML:



Above, Printable is in italics or oblique shape because it's an interface.
Below Printable are two subtypes Complex and Message

Example: the List interface

```
public interface List<E> extends Collection<E>
```

What are the methods of the List interface?

Suppose we want to make our own kind of List.

What do we need to write to build a class SimpleList that implements the List interface?

Comparison of abstract classes and interface

Abstract classes are not the same as interfaces:

`abstract class`	`interface`
cannot be instantiated	cannot be instantiated
can have methods	*can only have method signature*
can have variables	*can only have variables marked* `static final`
can have a constructor	no constructor
is a class	is an interface