Interfaces

Interfaces

Interfaces may remind you of abstract classes, but lack any kind of implementation.

- **Abstract classes**
  are when we have some kind of logic in the class and want to extend the behaviour.
- **Interfaces**
  contain member declarations only. They are useful when we want a certain set of classes to conform to the same form.

Interfaces

Why not just use abstract classes?

1. Interfaces define a contract, abstract classes a behaviour.
2. You can implement *several* interfaces, but can only extend *one* superclass.
3. Interfaces apply on disparate objects, inheritance requires commonality.

Interfaces

Syntax

```
public interface IMyInterface {
    void method();
    int anotherMethod();
    String yetAnotherMethod();
}
```

Only method signatures

- No implementations*
- No access modifiers

*) In Java 8 interfaces can include **default implementation** code, but it's an advanced topic

Shapes example

Interfaces, example (1)

We are working on a drawing program that will support various shape objects:

Each shape **should** implement the following methods:

```
public double area() {
    //implementation
}

public double perimeter() {
    //implementation
}
```

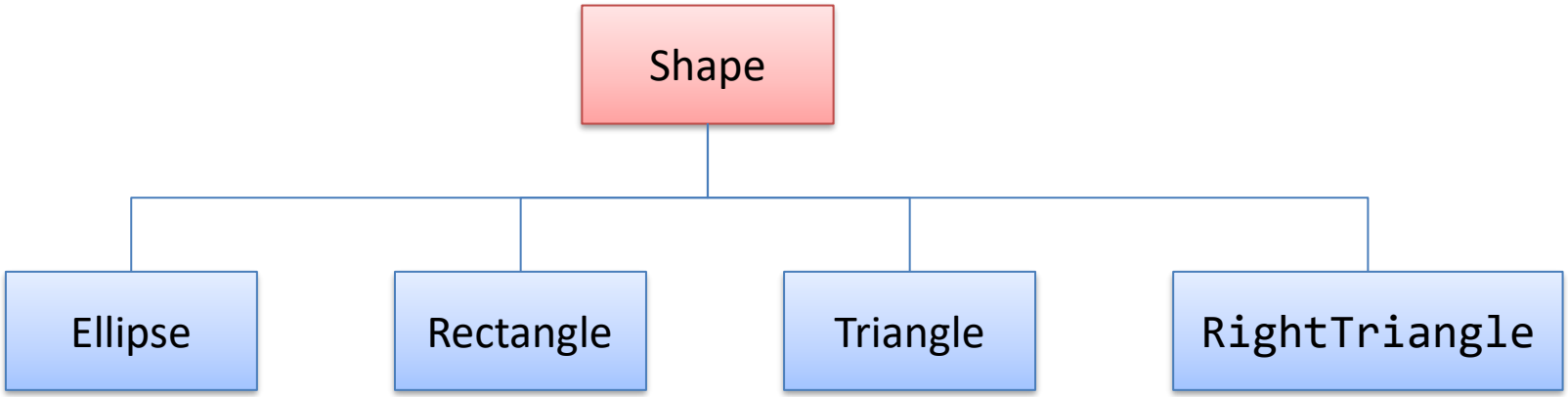
We can let them implement an interface to make sure they have the same methods

Interfaces, example (2)

We create a **Shape** interface:

```
public interface Shape {
    double area();
    double perimeter();
}
```

The Shape contains **definitions** which every shape has to **implement**.



Interfaces, example (3)

The syntax for **implementing** an interface is similar to **inheritance**, except we use the **implements** keyword.

```
public class Circle implements Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override public double area() {
        return Math.PI * radius * radius;
    }

    @Override public double perimeter() {
        return Math.PI * 2 * radius;
    }
}
```

Interfaces, example (4)

Rectangle also **implements** Shape, but with a different constructor and calculations.

Notice the different constructor compared to Circle

```
public class Rectangle implements Shape {
    private double width;
    private double height;

    public Rectangle(double width, double height) {
        this.height = height;
        this.width = width;
    }

    @Override public double area() {
        return width * height;
    }

    @Override public double perimeter() {
        return 2 * width + 2 * height;
    }
}
```

The constructor is never part of the interface

Interfaces, example (5)

RightTriangle has a similar implementation to Rectangle, but the calculations are slightly different.

The constructor definition is never part of the interface

```
public class RightTriangle implements Shape {
    private double width;
    private double height;

    public RightTriangle(double width, double height) {
        this.height = height;
        this.width = width;
    }

    @Override public double area() {
        return width * height / 2;
    }

    @Override public double perimeter() {
        double hypotenuse = Math.sqrt(width * width + height * height);
        return width + height + hypotenuse;
    }
}
```

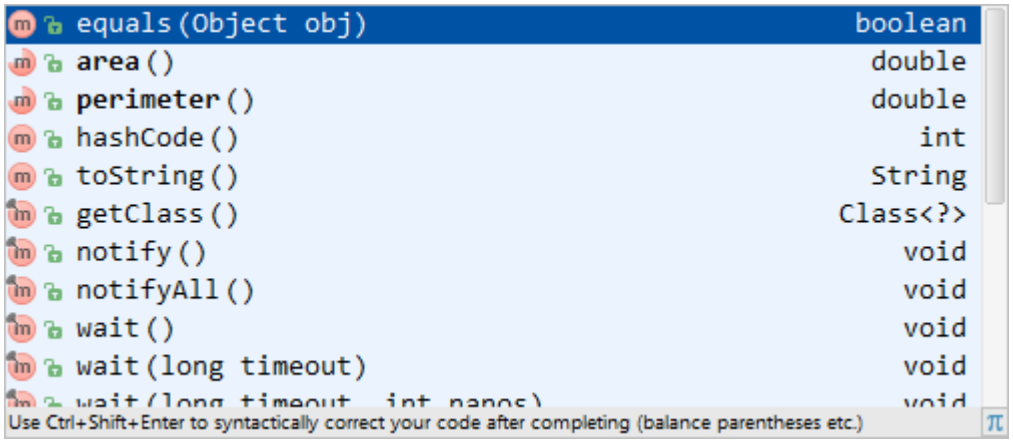
Interfaces

We can declare **interface** variables and instantiate with different **implementations**.

```
public static void main(String[] args) {
    Shape rect = new Rectangle(10, 10);
    Shape circle = new Circle(30);
    Shape circle = new RightTriangle(30, 10);

    rect.
}
```

Notice the **Shape** is used here as a type



Part of the interface

Interfaces

Where could this be useful?

We can pass different **implementations** as method arguments without the receiving method having any implementation details knowledge.

```
public class Canvas {
    public void printShapeInfo(Shape shape) {
        // This method has no knowledge of the type of
        // shape it is dealing with!

        System.out.println("Area: " + shape.area());
        System.out.println("Perimeter: " + shape.perimeter());
    }
}
```

Interfaces

We use **printShapeInfo** without giving the **Canvas** class any details more than **interface** specification.

```
public static void main(String[] args) {
    Shape rect = new Rectangle(4.0, 4.5);
    Shape circle = new Circle(2.0);
    Shape triangle = new RightTriangle(3.0, 4.0);

    Canvas c = new Canvas();

    c.printShapeInfo(rect);
    c.printShapeInfo(circle);
    c.printShapeInfo(triangle);
}
```

```
Area: 18.0
Perimeter: 17.0
Area: 12.566370614359172
Perimeter: 12.566370614359172
Area: 6.0
Perimeter: 12.0
```

Exercise 20

Lets do exercise 20