POWERING POTENTIAL

Inheritance – Towards Polymorphism

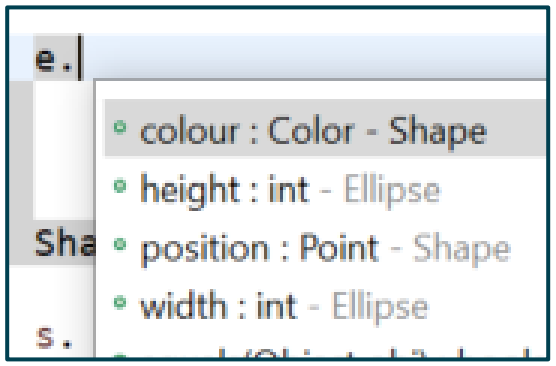
CONTENTS

- **Objectives**
  - To understand and use polymorphism
- **Contents**
  - Constructors – how they are affected
  - Overriding of methods
  - Substitutability
  - Runtime method version look up - polymorphism
- **Hands-on labs**

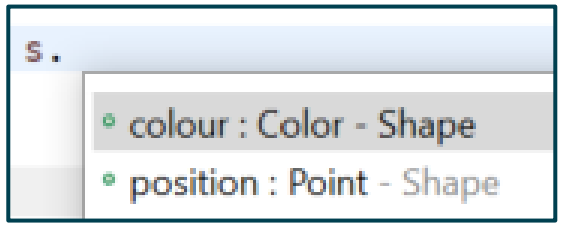
The principle of substitutability

- **Object of derived type exhibits all behavior of base type**
  - A derived object is a ‘kind of’ base object

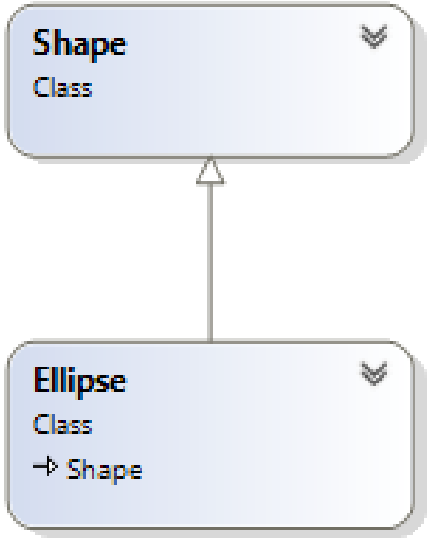
```
Ellipse e = new Ellipse();
Shape s = e;
```



But why would you do it?



Missing all the Ellipse's stuff



Why use substitution of references?

```
private static void drawShape(Shape s) {
        // code to draw
}

public static void main(String[] args) {
        Ellipse ellipse = new Ellipse(new Point(10,5));
        drawShape(ellipse);
```

1- Pass a parameter of base class type

```
public static Shape makeShape(int picNo) {
  if (picNo == 1)
        return new Ellipse(new Point(5,5));
}

public static void main(String[] args) {
        Shape s = makeShape(1);
```

2- Returning parameters of the base class type

Why use substitution of references?

```
Shape[] shapes = {
        myEllipse,
        yourTriangle,
        ourCircle
};

for (Shape s : shapes) {
        drawShape(s);
}
```

3- Collections and arrays

```
foreach (Shape s in shapes) {
        drawShape(s);
}
```

C#

```
public void drawShape(Shape shape) {
    // code for drawing a shape
}
```

Towards polymorphism..

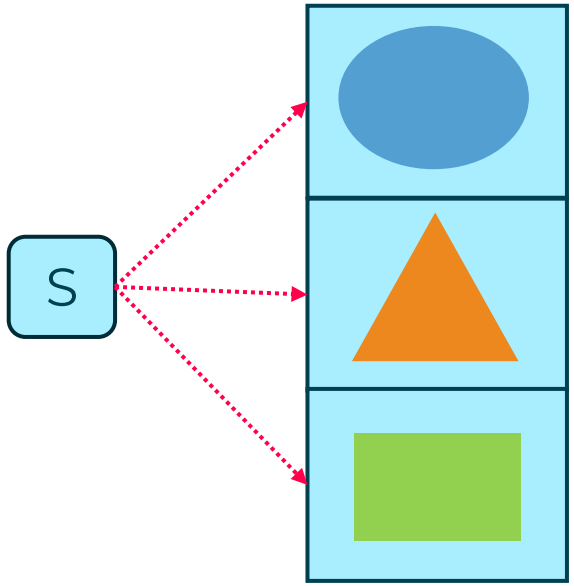
- Morphing into many shapes

```
Shape[] shapes = {
        myEllipse,
        yourTriangle,
        ourRectangle
};

for (Shape s : shapes) {
        drawShape(s);
}
```

```
foreach (Shape s in shapes) {
        drawShape(s);
}
```

C #



Java – Overriding base class methods

```
Rectangle rec = new Rectangle();

System.out.println(rec.getArea());
```

```
class Shape {
    public int getArea() {
        return 0;
    }
}

class Rectangle extends Shape {
    public int getArea() {
        return 100;
    }
}
```

which method *is invoked*?

Shape getArea() or Rectangle getArea()

100

Java – Overriding base class methods…

```
Shape rec = new Rectangle();

System.out.println(rec.getArea());
```

```
class Shape {
    public int getArea() {
        return 0;
    }
}

class Rectangle extends Shape {
    public int getArea() {
        return 100;
    }
}
```

which method *is invoked*?

Shape getArea() or Rectangle getArea()

100

Java: Enabling overriding

- **A derived class might want to alter implementation**
- **Best use the @Override annotation**
  - Compiler checks the method and its parameters
  - Good indication to the other developers

```
public class Shape {
  public Point position;
  public Color colour;

  public int getArea() {
        return 0;
  }
```

```
public class Rectangle extends Shape {
  public int width, height;

 @Override
  public int getArea() {
        return width * height;
  }
```

Good practice

Polymorphism – Lists and Arrays

```
public class Shape {
  public Point position;
  public Color colour;

  public int getArea() {
        return 0;
  }
```

```
public class Rectangle extends Shape {
  public int width, height;

  public int getArea() {
        return width * height;
  }
```

```
Shape myShape = new Shape();
Rectangle myRectangle = new Rectangle();
Shape[] shapes = {myShape, myRectangle};

for (Shape s : shapes)
        print(s.getArea());
```

Which of the getArea() methods are invoked? Shape or Rectangle?

Basics of casting – downcasting

- The data type of a reference that controls what is ‘visible’

```
class Person {
 private String name;
 public String getName() {
        return name;
 }
}
```

```
class Student extends Person {
 private String subject;
 public String getSubject() {
        return subject;
 }
}
```

```
Person[] people = {new Person(), new Student()};
for(Person p : people) {          // C#: foreach (var p in people)
  print(p.getName());             // Every Person has a name
  print(p.getSubject());          // Person -no subject

  Student s = (Student)p;         // new reference has new type
  print(s.getSubject());          // Student has ‘Subject’
}
```

Compiles but will it crash at runtime?

Safe downcasting

A downcast could fail at runtime with ‘`ClassCastException`’

- Test whether cast is safe via the **`instanceof`** keyword

```
Person[] people = { new Person(), new Student() };

for(Person p : people) {
  System.out.print(p.getName());          // Every Person has a name

 if (p instanceof Student) {
    Student s = (Student) p;              // cast to Student type
    System.out.println(s.getSubject()); // Student has Subject
}
```

Invoking base class functionality

- **A derived class can access base class member**

Calls first method with matching signature up the inheritance hierarchy

```
public class Student extends Person {
  private String subject;
  public String getDetails() {

    String data = super.getDetails(); // call base class
    //…… code
    return data + “\t” + subject;
  }
}
```

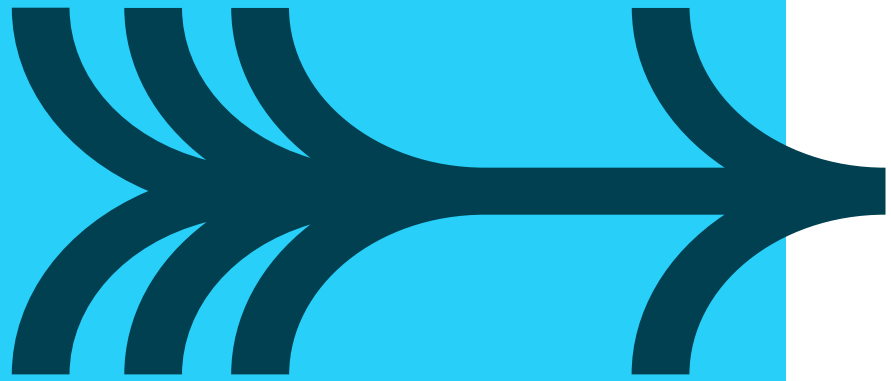
Non extendible classes and methods

- **A class can be written with the `final` modifier to prevent extension**

```
public final class String {
   ...
}
```

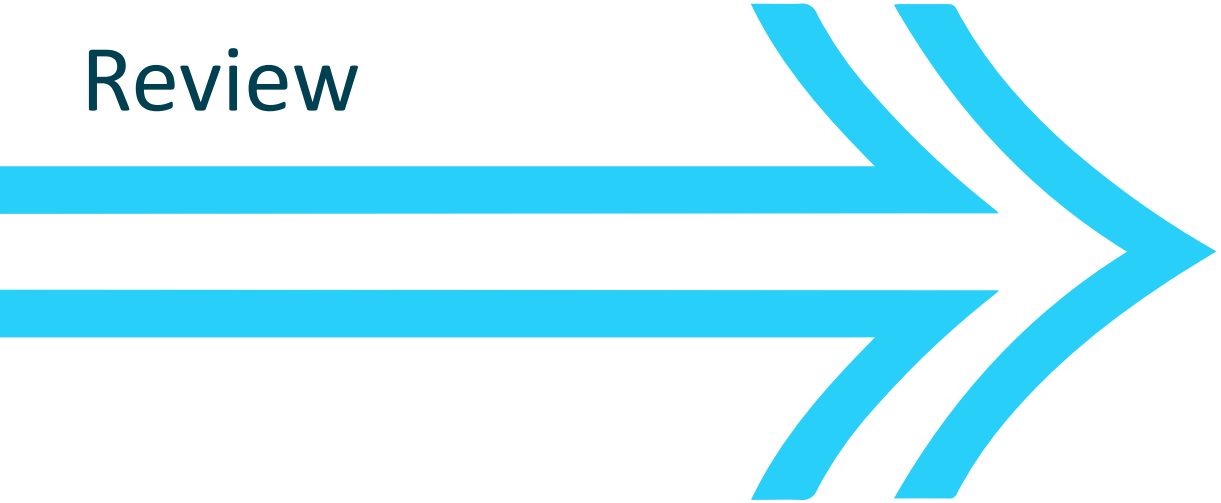
  - A method can be marked `final` to stop it being overridden

BEST PRACTICE



- **Use inheritance only for genuine "is a" relationships**
  - Logical to substitute object of derived class for object of base class
  - All methods in base class should make sense in derived class
- **Ad-hoc inheritance for short-term convenience tends to lead to future problems and surprises!**
- **Java only supports single inheritance**
  - Choosing a base class is thus significant in lots of ways

Review



- **Why do we do Inheritance?**
  - So we can upcast refs to a common base type to effect polymorphism
  - Maybe a bit of code reuse as well

- **Derived class inherits and can override and add**
- **Method calls automatically polymorphic**
- **Started to look at (down)casting**

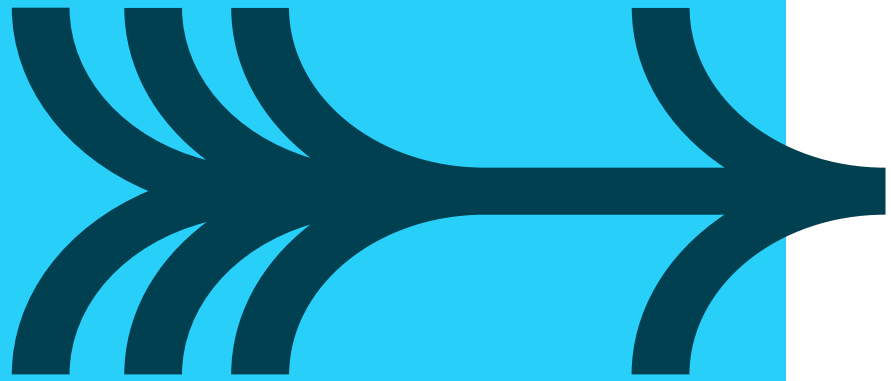
Hands-on labs

Working with inheritance:

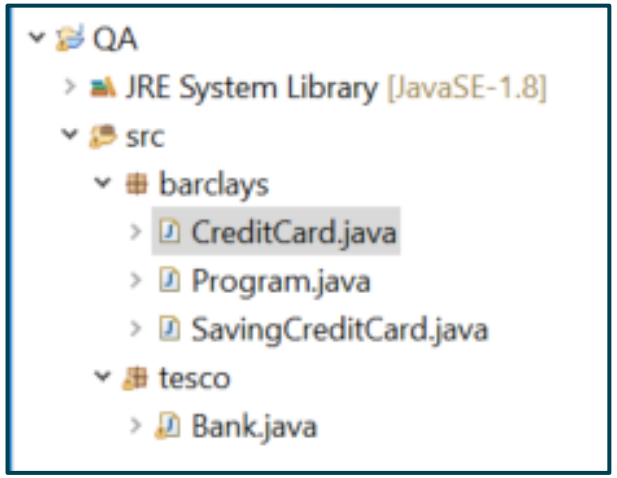
- Racing Cars and Employee Hierarchy

PROTECTED

- **Modifier that allows access to deriving types only**
  - Used to restrict access to methods
    - Fields should always be private, remember
- **Let's view an example...**



Protected example



Can be accessed by a class in the same package

```java
package barclays;

public class  CreditCard {
   protected int pin;

   public CreditCard(int pin) {
      this.pin = pin;
   }
}
```

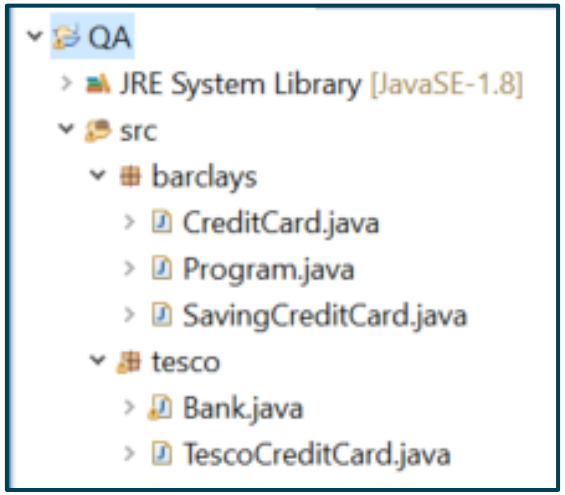
```java
package barclays;

public class Program {
   public static void main(String[] args) {
        CreditCard cc1 = new CreditCard(111);
        System.out.println(cc1.pin);
   }
}
```

```java
package tesco;
import barclays.*;
public class Bank {
    public static void main(String[] args) {
          CreditCard cc = new CreditCard(333);
          System.out.println(cc1.pin);
   }
}
```

But not by a class outside of the package

Protected example …




```java
package barclays;

public class CreditCard {
   protected int pin;

   public CreditCard(int pin) {
      this.pin = pin;
   }
}
```

```java
package tesco;
import barclays.CreditCard;

public class TescoCreditCard extends CreditCard {
   public TescoCreditCard(int pin) {
         super(pin);
   }
   public void changePin(int newPin) {
         this.pin = newPin;
   }
}
```

Can be accessed by a class outside of the package which extends the class

C#: protected internal

```
public class Account {
  protected internal double balance;
}

public class SavingAccount : Account {
    public double GetInterest() {
        return balance * 0.02;
    }
}
```

classes in the inheritance hierarchy have access to balance

```
public class Bank {
    public void SomeMethod() {
        Account acc = new Account();
        acc.balance *= 0.03;
    }
}
```

And so has any other class in the same assembly (only)

```
public class CompanyAccount : Account {
    public double GetInterest() {
        return balance * 0.05;
    }
}
```

Even when defined in a separate assembly