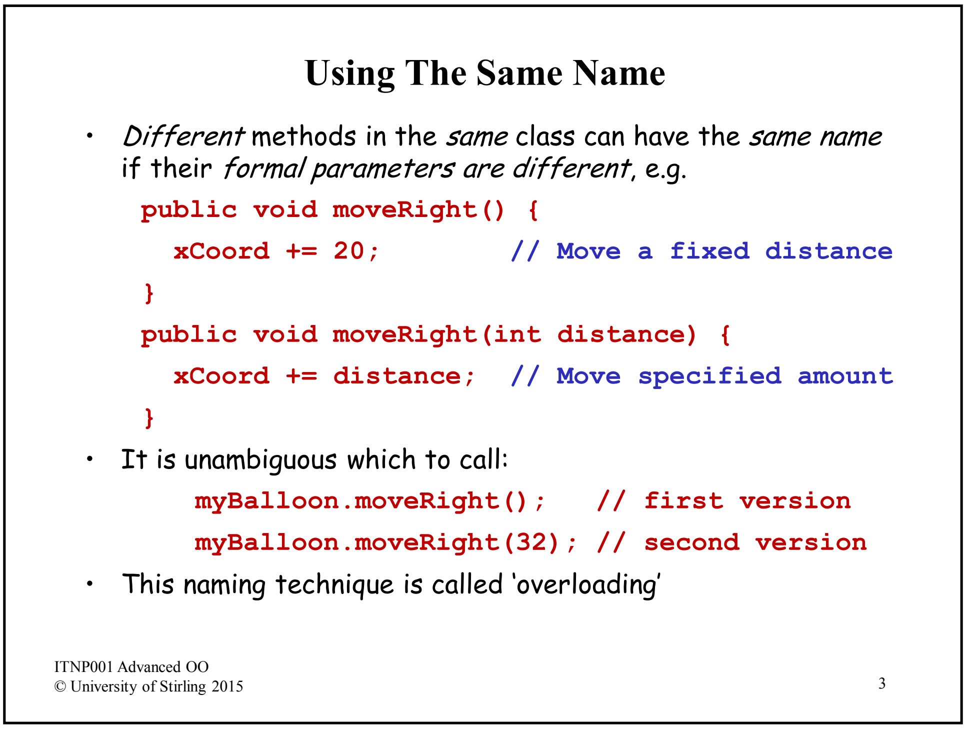
Advanced Object Orientation

- In this section:
  - Overloading
  - Inheritance

Method Overloading (JFS, chap 5)

- Consider how methods are named:
  - Within a class, methods normally have different names to avoid ambiguity
  - Methods in different classes may share names because the class or object name is given when called:

    ```
    rainTable.display(...);
    myBalloon.display(...);
    ```
  - Methods in different classes might well have different parameters
- In fact, the different names restriction can be relaxed a little
  - This can help by allowing consistent naming
  - We don't always have to dream up a new name!



Using The Same Name

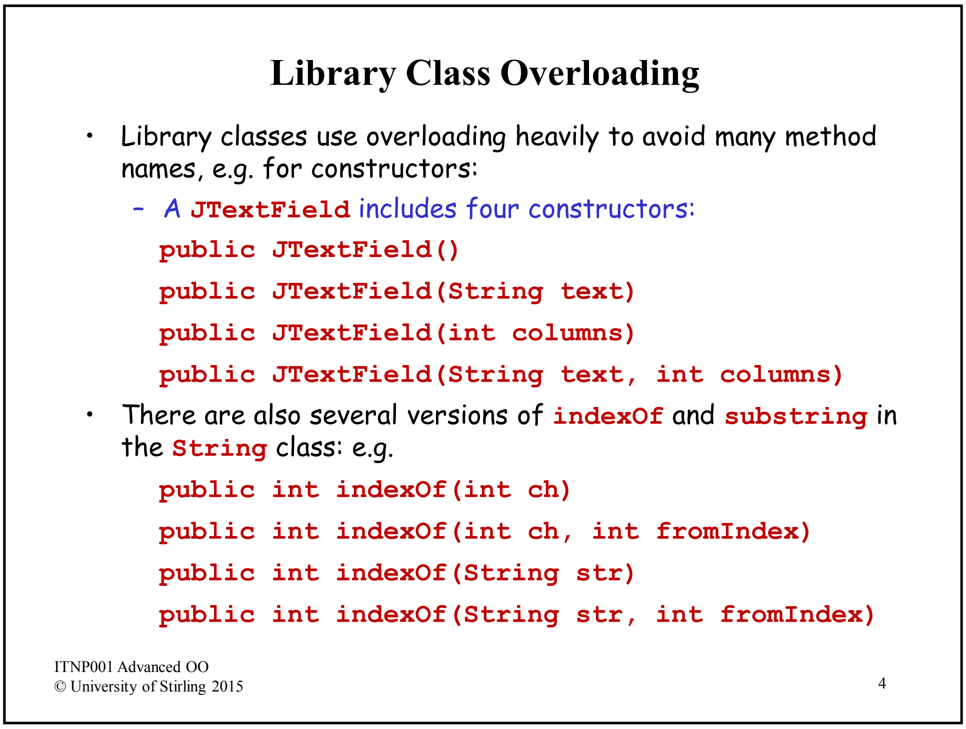
- *Different* methods in the *same* class can have the *same name* if their *formal parameters are different*, e.g.

```
public void moveRight() {
  xCoord += 20;         // Move a fixed distance
}
public void moveRight(int distance) {
  xCoord += distance;   // Move specified amount
}
```

- It is unambiguous which to call:

```
myBalloon.moveRight();    // first version
myBalloon.moveRight(32); // second version
```

- This naming technique is called 'overloading'



Library Class Overloading

- Library classes use overloading heavily to avoid many method names, e.g. for constructors:
  - A `JTextField` includes four constructors:

```
public JTextField()
public JTextField(String text)
public JTextField(int columns)
public JTextField(String text, int columns)
```

- There are also several versions of `indexOf` and `substring` in the `String` class: e.g.

```
public int indexOf(int ch)
public int indexOf(int ch, int fromIndex)
public int indexOf(String str)
public int indexOf(String str, int fromIndex)
```

Inheritance (JFS, chap 10)

- Inheritance has been used from the start, e.g.:

  ```
  public class Greeting extends JFrame
  ```

- This introduces a class called `Greeting` which extends a known class, namely `JFrame`
- We use `extends` to create a new class:
  - It contains ('inherits') *everything* the original class has (constants, variables, methods)
  - Extra items can be defined specifically for it
  - Items from the original class can be 'overridden' by new definitions
- Terminology:
  - `JFrame` is called the *superclass*
  - `Greeting` is called the *subclass*

Example: Extending JFrame

- See the Java documentation for methods in `JFrame` which are inherited by subclasses, e.g.:

  ```
  setDefaultCloseOperation()
  getContentPane()
  setJMenuBar()
  ```

  - We have called these from our code
- A `JFrame` is typically extended by adding:
  - Instance variables (data and widgets)
  - Methods such as `main`, `createGUI`, `actionPerformed`
- A `JFrame` extends the classes `Frame`, `Window`, `Container`, `Component`, `Object` in turn
  - See the Java documentation here
  - Which shows all inherited variables and methods

http://docs.oracle.com/javase/7/docs/api/javax/swing/JFrame.html

Inheritance in Java

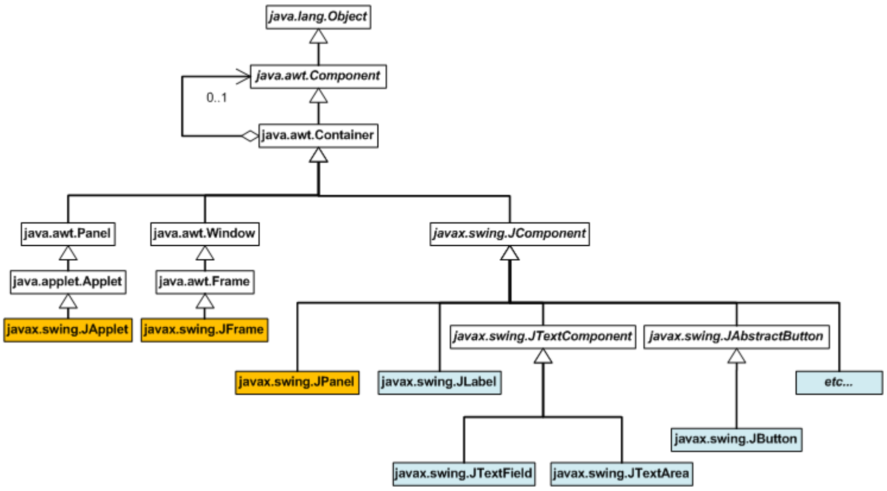
- So, programs have facilities defined in classes **`JFrame`**, **`Frame`**, **`Window`**, ... e.g.:

  `setVisible, setTitle`

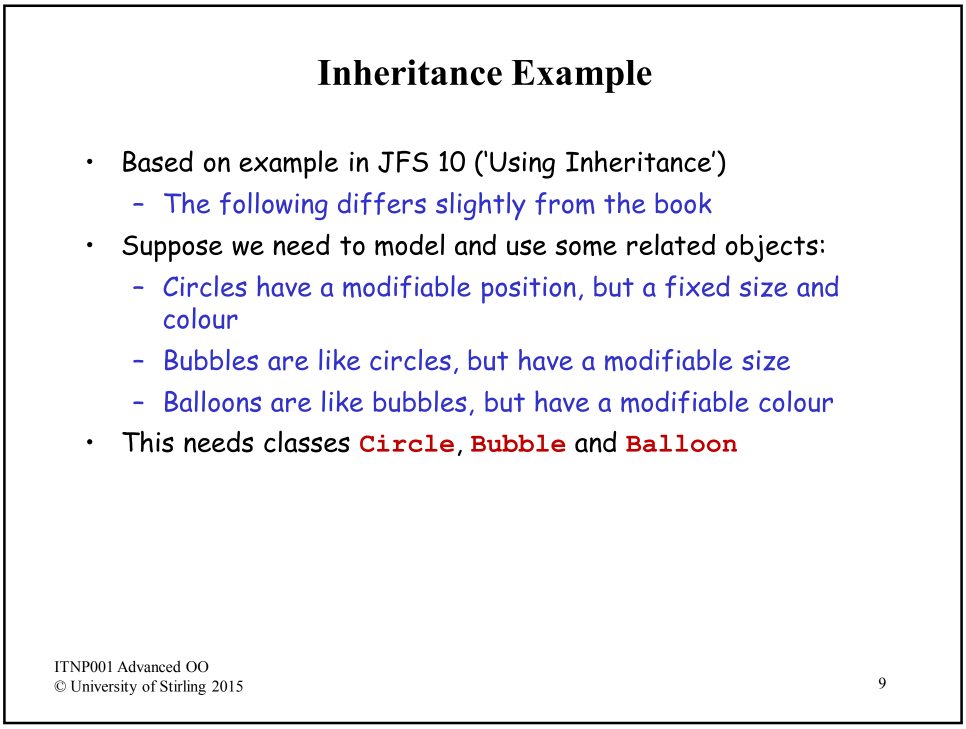
  `setBackground, getWidth`

- All Java is like this:
  - The libraries form a *hierarchical structure* - see next slide
  - This helps to compartmentalise functions
  - It organises class relationships, e.g. **`Window`** and **`JPanel`** extend **`Container`** in slightly different ways
- Our programs can use inheritance too
  - Again, helps to compartmentalise functions

- Part of the Java libraries class hierarchy:

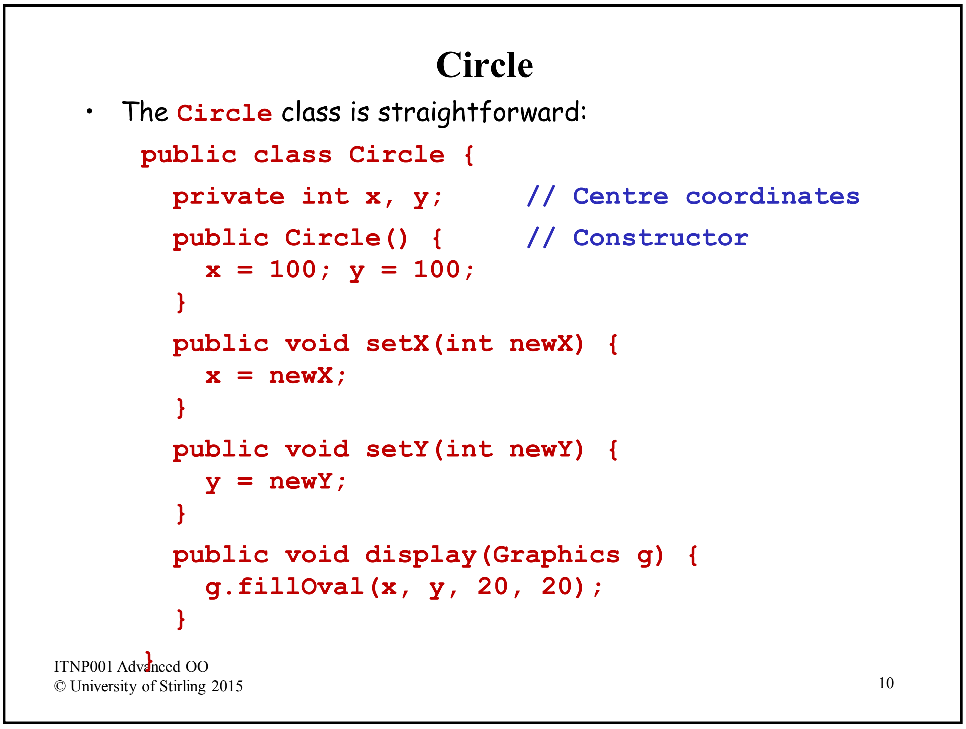


From: Stephen Wong https://www.clear.rice.edu/comp310/JavaResources/GUI/



Inheritance Example

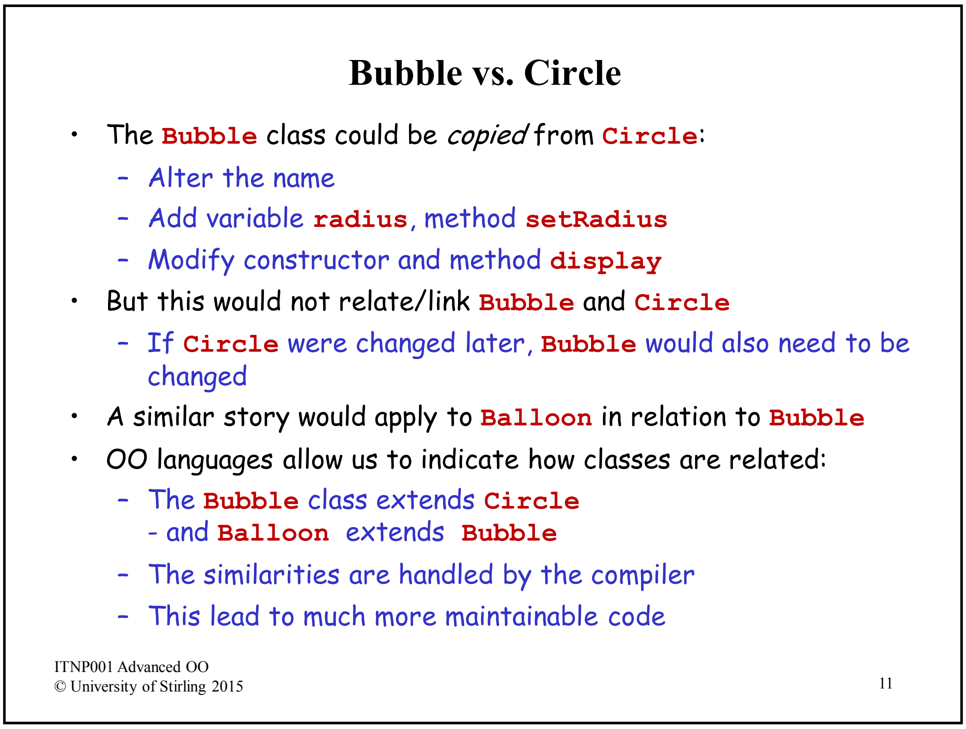
- Based on example in JFS 10 ('Using Inheritance')
  - The following differs slightly from the book
- Suppose we need to model and use some related objects:
  - Circles have a modifiable position, but a fixed size and colour
  - Bubbles are like circles, but have a modifiable size
  - Balloons are like bubbles, but have a modifiable colour
- This needs classes `Circle`, `Bubble` and `Balloon`



Circle

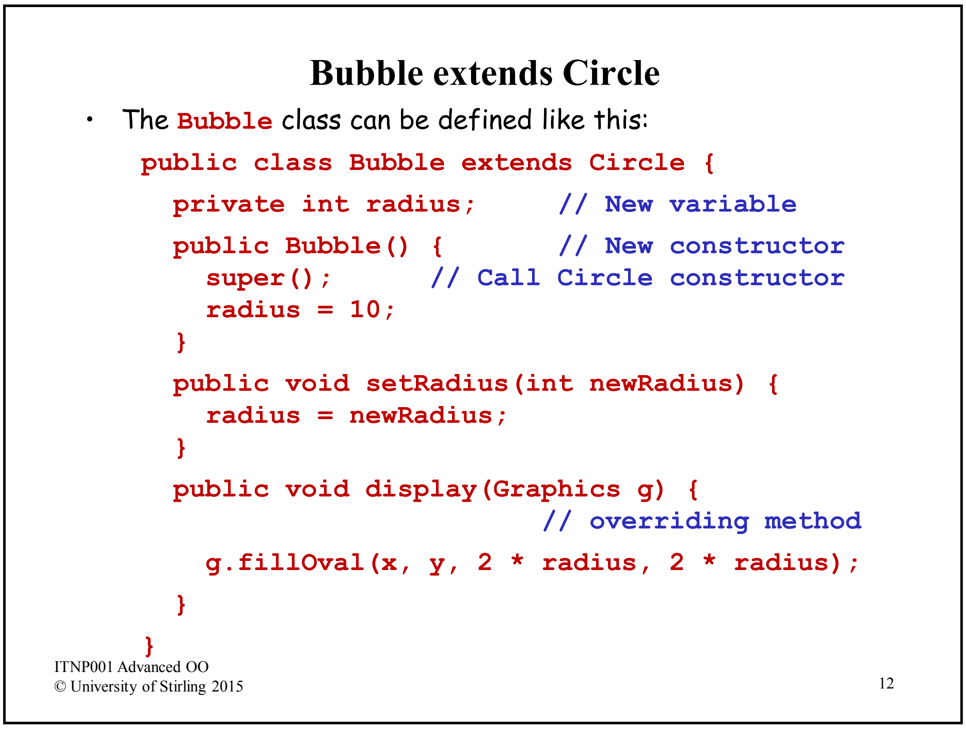
- The `Circle` class is straightforward:

```
public class Circle {
  private int x, y;      // Centre coordinates
  public Circle() {      // Constructor
    x = 100; y = 100;
  }
  public void setX(int newX) {
    x = newX;
  }
  public void setY(int newY) {
    y = newY;
  }
  public void display(Graphics g) {
    g.fillOval(x, y, 20, 20);
  }
}
```



Bubble vs. Circle

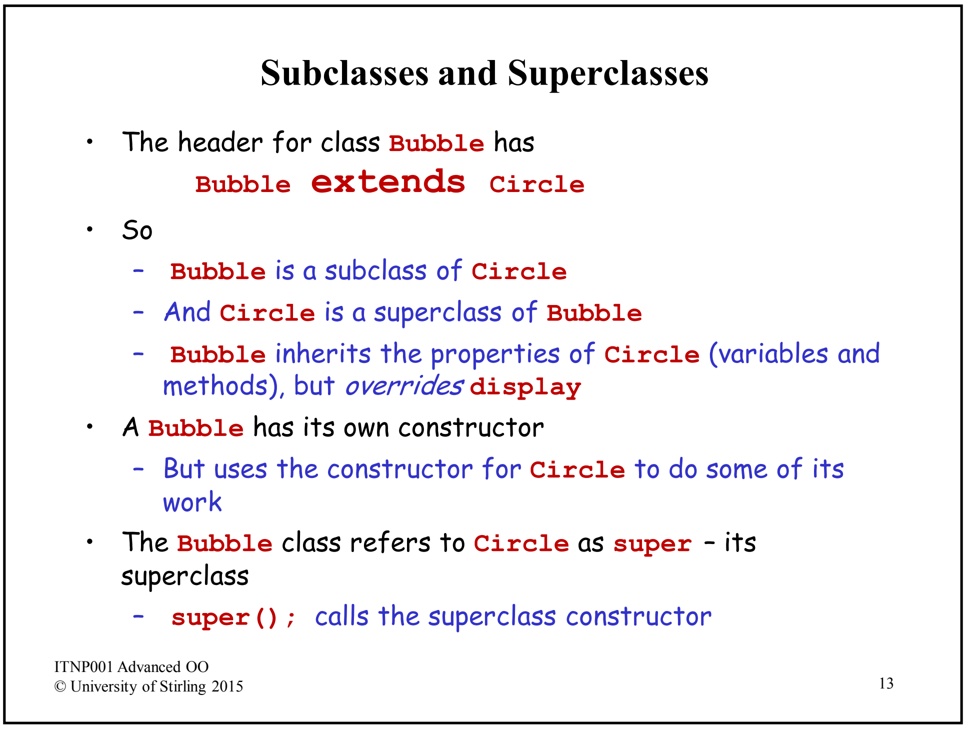
- The **Bubble** class could be *copied* from **Circle**:
  - Alter the name
  - Add variable **radius**, method **setRadius**
  - Modify constructor and method **display**
- But this would not relate/link **Bubble** and **Circle**
  - If **Circle** were changed later, **Bubble** would also need to be changed
- A similar story would apply to **Balloon** in relation to **Bubble**
- OO languages allow us to indicate how classes are related:
  - The **Bubble** class extends **Circle**
    - and **Balloon** extends **Bubble**
  - The similarities are handled by the compiler
  - This lead to much more maintainable code



Bubble extends Circle

- The **Bubble** class can be defined like this:

```
public class Bubble extends Circle {
  private int radius;     // New variable
  public Bubble() {       // New constructor
    super();      // Call Circle constructor
    radius = 10;
  }
  public void setRadius(int newRadius) {
    radius = newRadius;
  }
  public void display(Graphics g) {
                         // overriding method
    g.fillOval(x, y, 2 * radius, 2 * radius);
  }
}
```

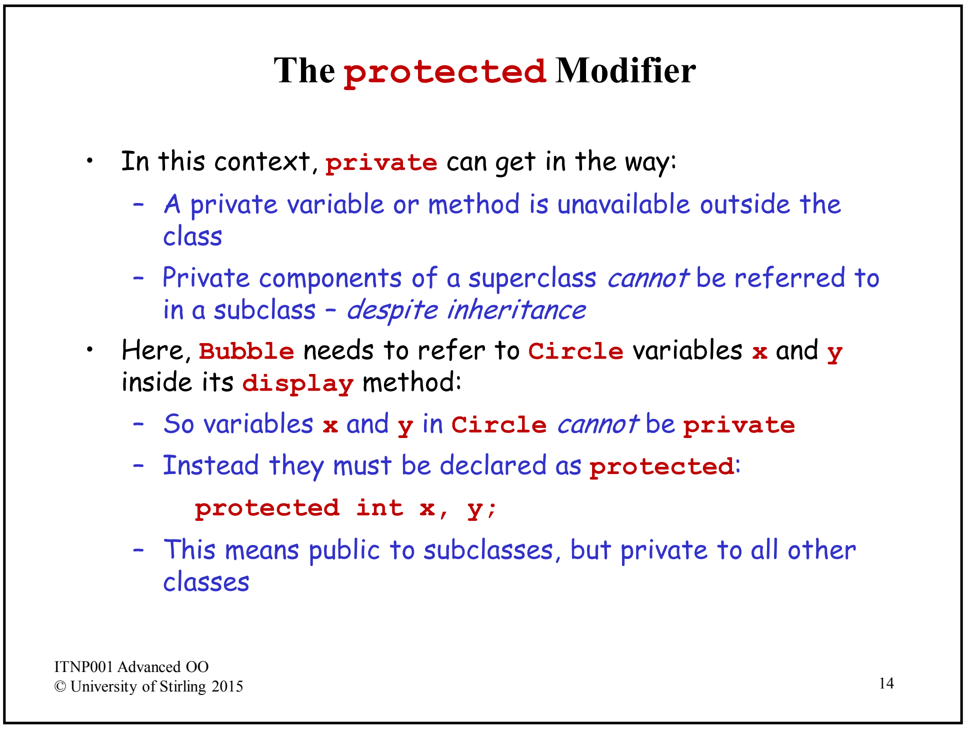


Subclasses and Superclasses

- The header for class `Bubble` has

  `Bubble extends Circle`

- So
  - `Bubble` is a subclass of `Circle`
  - And `Circle` is a superclass of `Bubble`
  - `Bubble` inherits the properties of `Circle` (variables and methods), but *overrides* `display`
- A `Bubble` has its own constructor
  - But uses the constructor for `Circle` to do some of its work
- The `Bubble` class refers to `Circle` as `super` - its superclass
  - `super();` calls the superclass constructor

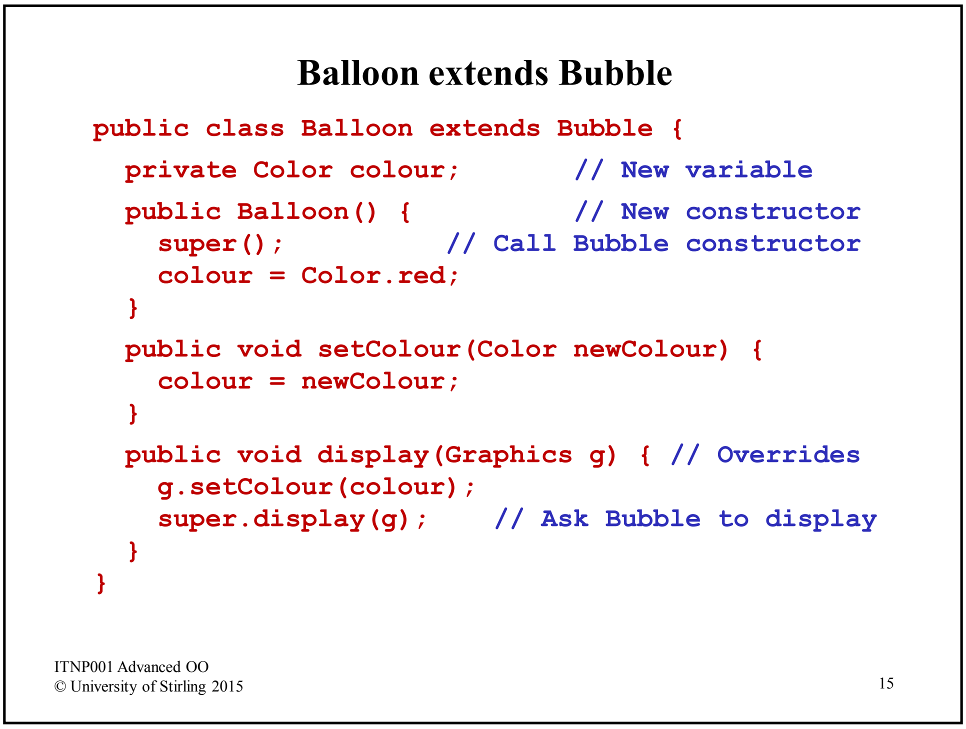


The `protected` Modifier

- In this context, `private` can get in the way:
  - A private variable or method is unavailable outside the class
  - Private components of a superclass *cannot* be referred to in a subclass - *despite inheritance*
- Here, `Bubble` needs to refer to `Circle` variables `x` and `y` inside its `display` method:
  - So variables `x` and `y` in `Circle` *cannot* be `private`
  - Instead they must be declared as `protected`:

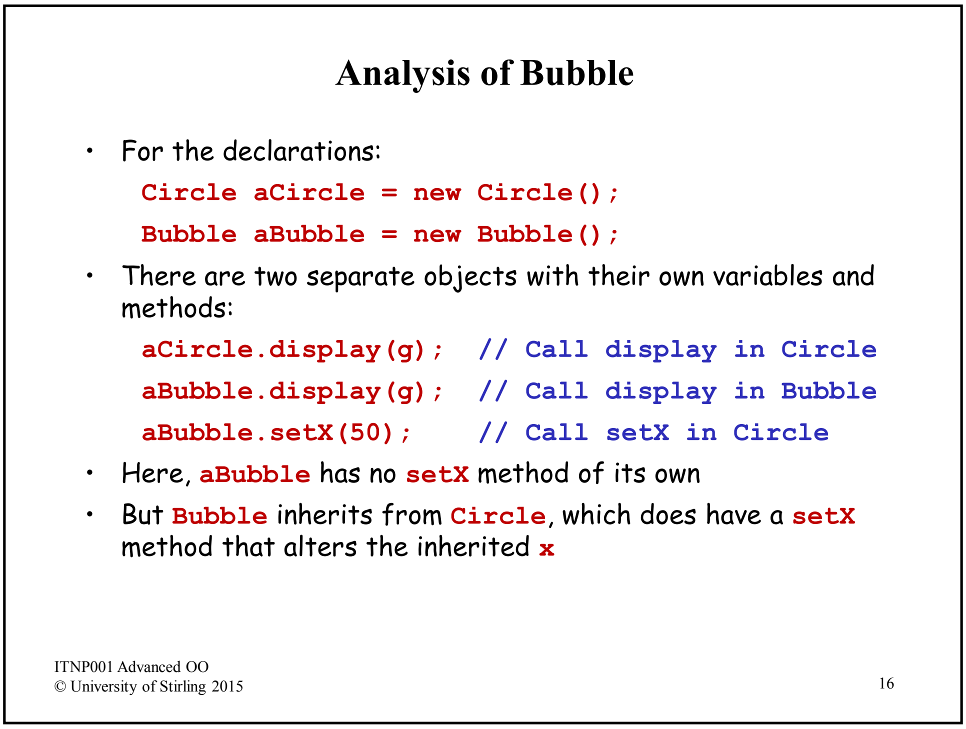
    `protected int x, y;`

  - This means public to subclasses, but private to all other classes



Balloon extends Bubble

```
public class Balloon extends Bubble {
  private Color colour;        // New variable
  public Balloon() {           // New constructor
    super();           // Call Bubble constructor
    colour = Color.red;
  }
  public void setColour(Color newColour) {
    colour = newColour;
  }
  public void display(Graphics g) { // Overrides
    g.setColour(colour);
    super.display(g);     // Ask Bubble to display
  }
}
```



Analysis of Bubble

- For the declarations:

```
Circle aCircle = new Circle();
Bubble aBubble = new Bubble();
```

- There are two separate objects with their own variables and methods:

```
aCircle.display(g);  // Call display in Circle
aBubble.display(g);  // Call display in Bubble
aBubble.setX(50);    // Call setX in Circle
```

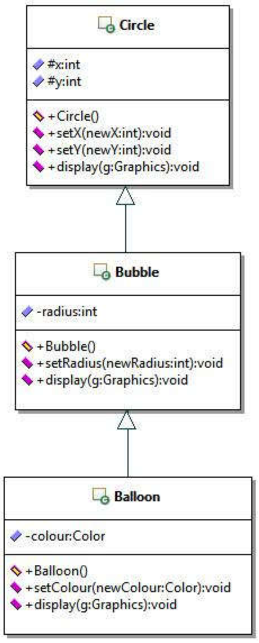
- Here, `aBubble` has no `setX` method of its own
- But `Bubble` inherits from `Circle`, which does have a `setX` method that alters the inherited `x`

Analysis of Balloon

- Class Balloon inherits from Bubble and so, implicitly, also from Circle:
  - Its setX comes from Circle
  - Its setRadius comes from Bubble
  - Its display overrides the one in Bubble
- Note that super.display calls the (overridden) display method from the Bubble superclass
- Note that the radius variable from Bubble is not referred to by Balloon
- This could remain private in Balloon, and does not have to be protected

Class diagrams

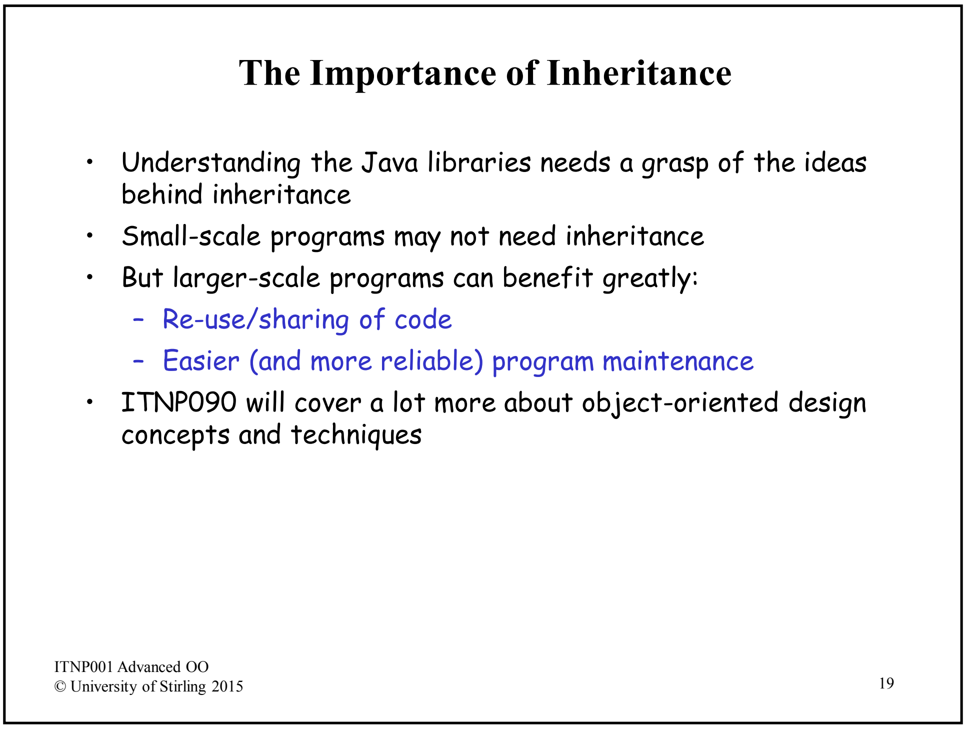
- OO program structures are often represented visually using *UML class diagrams*
- This diagram was drawn by a tool available in our labs called Borland Together
- More about this next semester in ITNP090



Circle
#x:int
#y:int
+Circle()
+setX(newX:int):void
+setY(newY:int):void
+display(g:Graphics):void

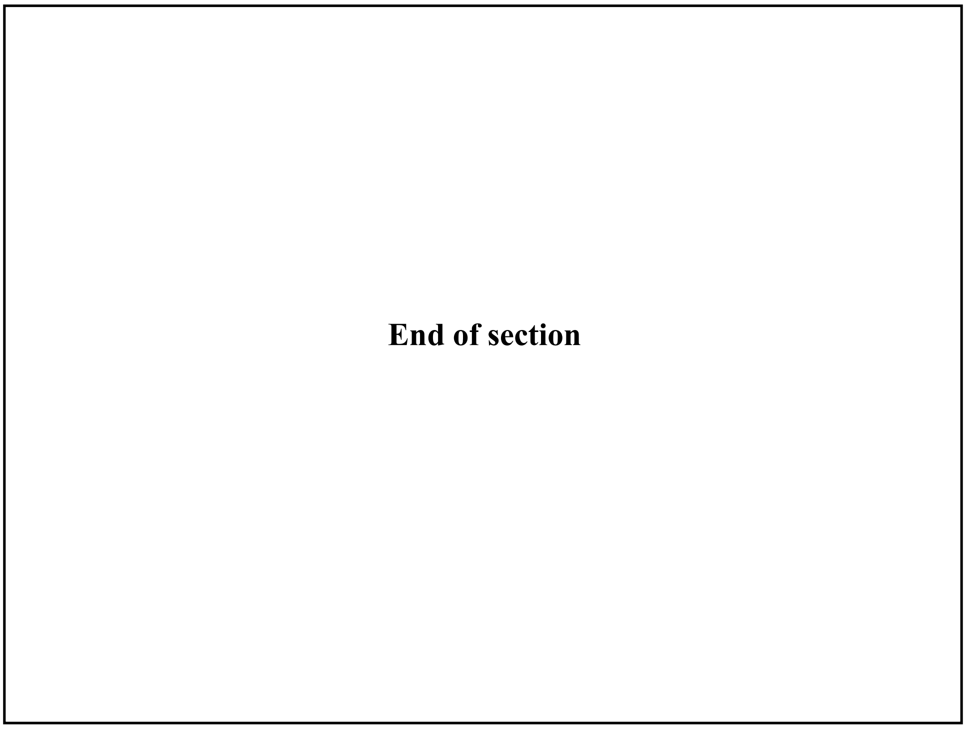
Bubble
-radius:int
+Bubble()
+setRadius(newRadius:int):void
+display(g:Graphics):void

Balloon
-colour:Color
+Balloon()
+setColour(newColour:Color):void
+display(g:Graphics):void



The Importance of Inheritance

- Understanding the Java libraries needs a grasp of the ideas behind inheritance
- Small-scale programs may not need inheritance
- But larger-scale programs can benefit greatly:
  - Re-use/sharing of code
  - Easier (and more reliable) program maintenance
- ITNP090 will cover a lot more about object-oriented design concepts and techniques



End of section