05 Inheritance

Inheriting Classes

Object differences via properties – Objects from the same class have the *same behaviour* but *different properties*:

```
Button ok = new Button();
Button cancel = new Button();
ok.text="OK";
cancel.text="Cancel";
```

A "blob" (see the example!) has properties fill colour, stroke color, center, and radius. A blob has methods jitter and draw. By setting different properties, quite different blobs can be generated even though their class is the same, e.g.

```
Blob blob1, blob2, blob3, blob4;

void setup(){
  blob1 = new Blob(200,200,100);
  blob2 = new Blob(400,200,50);
  blob2.fill=#5649EA;
  blob3 = new Blob(200,400,75);
  blob3.fill=#E81CCD;
  blob3.stroke=#1811F5;
  blob4 = new Blob(400,400,100);
  blob4.fill=#D1F739;
}
```

All objects of the same class have the same behaviour, but the properties may be different.

Objects with totally different behaviours – If two objects have completely different behaviours, then they belong to completely different classes (e.g. Blob vs PFont).

We can imagine a spectrum:

- objects from the same class at one end, differing only in their property values but sharing all property names, types and methods (e.g. two different `PVectors`)
- objects from different classes at the other end, in which all the properties and methods are completely different (e.g. a `PVector` and a `PImage`)

Sometimes we want something in between, i.e. two classes that share all properties and methods except for a few, or perhaps one class should have an extra few properties or methods that the other does not. For example imagine a `PImage3D` class. Like a `PImage`, it has `width`, `height`, and `pixels` properties.

But since it represents a 3D image, it also needs a `depth` property.

Inheritance

Inheritance is used whenever you want multiple classes to share *some* behaviour but *not all* the behaviour.

For example, the Blob class can jitter and draw itself but it cannot slowly rotate. A RotatingBlob class is nearly the same as a Blob class, but it has an additional property (rotation speed) and a slightly different drawing method (i.e. the points forming the blob's outline must be rotated first)

	Blob	RotatingBlob
Fields	fill stroke center radius	fill stroke center radius **rotationSpeed**
Methods	draw jitter	**draw** jitter

Blob and RotatingBlob are so similar that it would be wasteful to create two completely different classes.

Inheritance – lets two classes share properties and methods.

`extends` keyword – is the main way to link up two classes via inheritance:

```
class Blob {
     /*…properties and methods…*/
}

class RotatingBlob extends Blob {
     /*…properties and methods…*/
}
```

RotatingBlob will now inherit properties and methods defined inside the Blob class.

Inheritance is unidirectional (one way) – RotatingBlob gets all the properties and methods that Blob has, but the opposite is not true.

Superclass/Subclass – Blob is called a superclass or parent class, RotatingBlob a subclass or derived class or child class.

Generalization/Specialization – Blob is a more general class, RotatingBlob is a more specialized class.

How to control what is inherited – mark fields/methods as either public, private or protected:

`public`	Inherited and accessible outside of class
`protected`	Inherited but not accessible outside of class
`private`	Not inherited and not accessible outside of class

A sketch of the blob class –

```
class Blob {
     // Properties
     public color fill, stroke;
     protected PVector center;
     public float radius;
     protected ArrayList<PVector> outline;

     // Constructor
     public Blob(float x, float y, float radius) {
       /* …code… */
     }

     // Draw method
     public void draw() {
       /* …code… */
     }

     // Method to jitter the blob
     public void jitter(float howMuch) {
       /* …code… */
     }
}
```

A sketch of the Rotating blob class –

```
class RotatingBlob extends Blob {

     // Property
     public float rotationSpeed;

     // Constructor
     public RotatingBlob(float x, float y,float radius) {
          /* …code… */
     }

     // Overridden draw method
     public void draw() {
          /* …code… */
     }
}
```

Advantages of inheritance – if the subclass is very similar to the superclass, then the subclass does not need much code.

For example, there is no fill property or jitter() method defined for RotatingBlob but the RotatingBlob class still has them due to the inheritance relationship:

```
RotatingBlob blob = new RotatingBlob(400,400,100);
blob.fill=#D1F739;
blob.rotationSpeed=TWO_PI/500;
/* …code… */
blob.draw();
blob.jitter(0.01);
```

Likewise, inside the RotatingBlob there is no outline ArrayList defined, but the inherited version of outline can be accessed:

```
for (PVector point: outline)
       point.rotate(rotationSpeed);
```

Overriding methods – if a subclass wants to modify a method belonging to the superclass, it can override it by defining the same method again, e.g. see the draw() method in the RotatingBlob class.

`super` keyword – used inside a subclass to access methods and properties belonging to the superclass. There are two general cases where this is needed: (i) inside the constructor, when you want the objects of both classes to be set up the same and (ii) inside a method, when you want to reuse the original version of a method you have overridden.

Example of a constructor with `super`:

```
class RotatingBlob extends Blob {
     /*…code…*/

     public RotatingBlob(float x, float y,float radius) {
          super(x, y, radius);
     }

     /*…code…*/
}
```

Example of an overridden method with `super`:

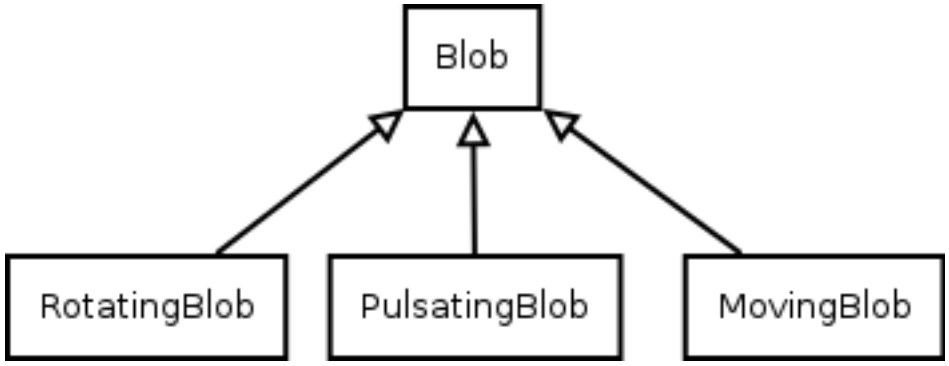
```
class RotatingBlob extends Blob {
     /*…code…*/
     public void draw() {
          // Rotate all the points in the outline
          for (PVector point: outline)
               point.rotate(rotationSpeed);
          // Draw the blob
          super.draw();
     }
     /*…code…*/
}
```

Inheritance Hierarchies

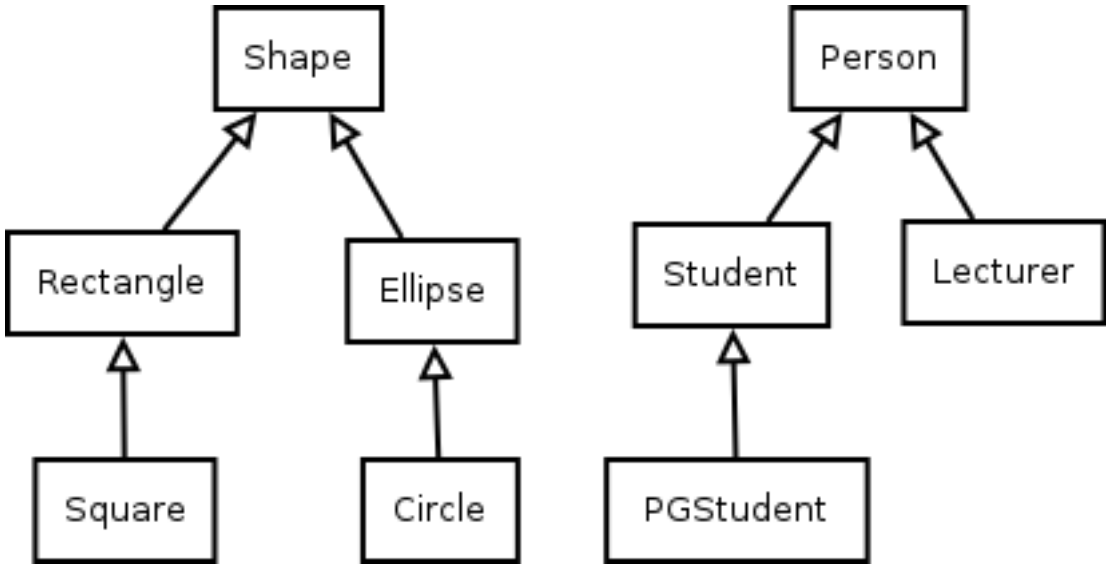
Many classes can be related by inheritance. For example, besides a rotating blob, we may also be interested in defining classes for pulsating blobs, moving blobs etc:

```
class Blob {}
class RotatingBlob extends Blob {
}
class PulsatingBlob extends Blob {
}
class MovingBlob extends Blob {
}
```

Inheritance diagram – can be used to visualize the relationship:



Inheritance hierarchy – a group of two or more inheriting classes, e.g.:



Abstract Classes

The highest levels of an inheritance hierarchy are usually so generalized that it does not make sense to use them to create objects. However, they do have properties that should be inherited, for example a shape is too abstract to draw but it has a position and a size:

```
class Shape {
  protected PVector position,size; // common properties

  public Shape(float x, float y, float w, float h) {
     position=new PVector(x,y);
     size=new PVector(w,h);
  }

  public void draw() { // how to draw a shape?
  }
}
```

Given the above class, some people may accidentally/incorrectly create objects of type shape, e.g.

```
Shape someShape = new Shape(30,40,100,100);
```

In fact, what was intended was for only subclasses to be used to create shapes, e.g.

```
Rectangle rect=new Rectangle(30,40,100,100);
Circle circle=new Circle(500,500,23);
```

`abstract` keyword – the solution is to mark classes as abstract if they are not meant to be used to create objects, i.e.

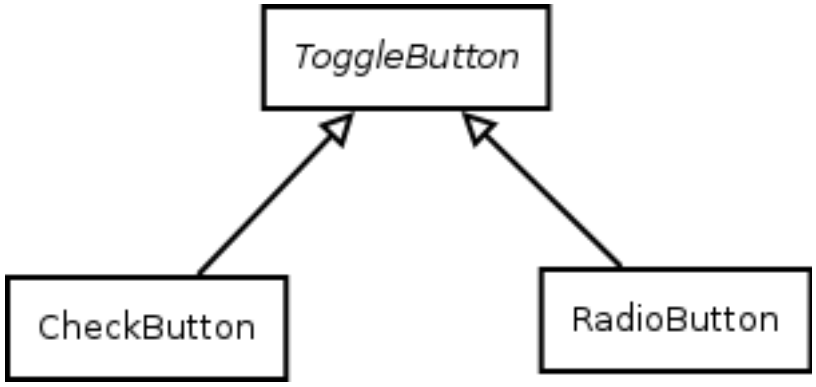
```
abstract class Shape {
     /*…class definition…*/
}
```

Abstract methods

If it does not make sense to implement a method in a superclass, but all subclasses should implement it, then mark it abstract as well.

```
abstract class Shape {
 /*…methods & properties…*/

 abstract public void draw(); //subclasses MUST have draw

}
```

Toggle button example – a toggle button is a button that stays on or off after you click it. GUIs usually have two different types of toggle buttons: radio buttons and check buttons. They are drawn differently and have different shapes (square vs. circle). However nearly all other behaviours are the same.



Inheritance diagrams with abstract classes – use italics or a different style to distinguish abstract from concrete classes.

Sketch of the ToggleButton class with abstract methods bolded:

```
abstract public class ToggleButton {
  protected PVector position;
  protected float size;
  protected boolean state;
  protected String label;

  public ToggleButton() {/*…code…*/}

  public boolean mouseClicked() {
    if (ptInRegion(mouseX, mouseY)) {
      state=!state;
      return true;
    }
    return false;
  }

  abstract public boolean ptInRegion(float x, float y);
  abstract public void drawButton();

  public void drawLabel() {/*…code…*/}

  public void draw() {
    drawButton();
    if (label!=null) drawLabel();
  }
}
```

Note how implemented methods such as mouseClicked() can call abstract methods such as ptInRegion()!

Implementation of abstract methods – must be done in the subclasses, e.g.

```
public class RadioButton extends ToggleButton {

 /*…fields…*/

  public RadioButton() {/*…code…*/}

  public boolean pointInButtonRegion(float x, float y) {
    return PVector.dist(center,
                        new PVector(x,y))<=size/2;
  }

  public void drawButton() {
    noFill();
    stroke(255);
    strokeWeight(4);
    ellipseMode(CORNER);
    ellipse(position.x, position.y, size, size);
    if (state) {
      fill(255);
      ellipse(position.x+size/4, position.y+size/4,
              size/2, size/2);
    }
  }

 /*…other methods…*/
}
```

Creating objects – create objects from inheriting classes in the normal way:

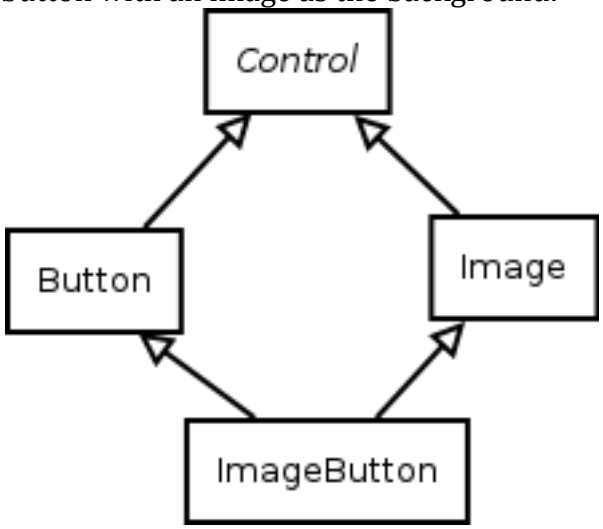
```
CheckBox cb1, cb2;
RadioButton rb1, rb2;

void setup(){
  cb1=new CheckBox(100,100,30, null);
  cb2=new CheckBox(300,100,30, "pizza");
  cb2.set();
  rb1=new RadioButton(100,200,30, null);
  rb1.set();
  rb2=new RadioButton(300,200,30, "milk");
}
```

Note that the set method is inherited in both cases!

Single vs Multiple Inheritance

Single inheritance – if every class inherits from at most one other class, then inheritance is single.

Multiple inheritance – occurs when one class can inherit from more than one class. For example all GUI controls have a position on the screen and can draw themselves. A button is a control and an image is a control. A image button is a button with an image as the background:



The main problem with multiple inheritances is working out what gets inherited:

```
abstract class Control {
     PVector position;
     abstract public void draw();
}
class Button extends Control {
     public void draw(){ /*…draw the button…*/ }
}
class Image extends Control {
     public void draw(){ /*…draw the image…*/ }
}
class ImageButton extends Button, Image  {
     /* Which draw method is inherited?? */ }
}
```

This makes the language messy and therefore Processing only allows single inheritance.

Multiple to Single Inheritance – multiple inheritance hierarchies can be converted into single inheritance hierarchies with some thought, e.g. we could make an image a property of Button and dispose of the ImageButton class:

```
abstract class Control {
     PVector position;
     abstract public void draw();
}
class Button extends Control {
     public Image image=null;
     public void draw(){
         // draw the image
         if (image!=null) image.draw();
          /*…draw the rest of the button…*/
     }
}
class Image extends Control {
     public void draw(){ /*…draw the image…*/ }
}
```