Module 11:
Polymorphism and virtual methods

In this module we will cover

- **Polymorphism**
- **virtual methods**

Aims of this module

A key feature of object oriented programming is the ability to use objects "polymorphically"
This is one of the most powerful featrues, and you cannot be said to be truly writing OO code unless you understand and can use polymorphism.

Polymorphism allows code to be written which operates upon a base class type, but which will also immediately (without modification) work on any other concrete classes which inherit from this base class, and which may be invented at any time in the future.

We explain this feature and describe a context in which it might apply.

11.1 Polymorphism

....and the words get bigger.....

This is all to do with being able to write applications which operate upon a base class type, but which will also immediately (without modification) work on any other concrete classes which inherit from this base class, and which may be invented at any time in the future.

This is one of the most powerful OO features, and you cannot be said to be truly writing OO unless you understand and can use polymorphism.

A normal example is to consider graphics objects - so we will ...

... but first consider in the abstract

Payroll program: Employee-> ProjectManager / Contractor / Programmer

BPhysicsAnalysis: TrackJet / ClusterJet

Suppose you were going to write a DISPLAY program to display graphics shape objects on the screen.

The problem is that at the point of writing you do not know which specific shape objects will be available.

Therefore you write a generic `Shape` class which has all of the methods you want for your program to work:

```
class Shape
{
  public:
   void draw( float x, float y ) ;   //draw object at x,y
   void setSize( float size ) ;      //set size of shape

  private:
    GraphicsService graphics ;

};
```

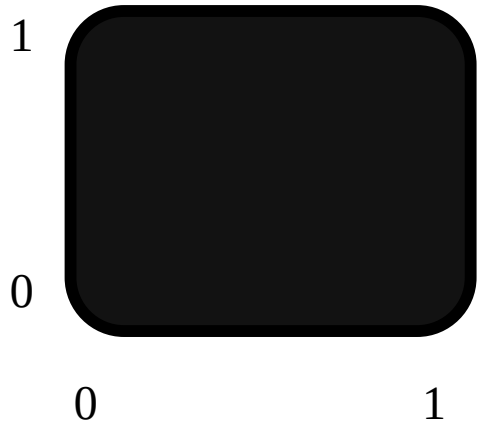
..and here is some generic code to implement the methods..

```
//.....................................
void Shape::draw( float x, float y )
{
  //For generic shape just make a dot
  // call graphics library function to draw a point
  graphics.point( x, y ) ;

}


//.....................................
void Shape::setSize( float size )
{
  // A dot has no size so do nothing
}
```

Now lets imagine a bit of the DISPLAY program which draws a border at the top of the screen, using a Shape

```
main()
{

  Shape x ;

  drawTopBorder( x ) ;

}
```

```
void drawTopBorder( Shape& s )
{
  // screen runs 0->1 in both directions
  // lets draw 10 shapes along top

  // Set size so that 10 fit in a line
  s.setSize( 0.1 ) ;

  // Top means y = 1.
  float y = 1. ;

  // Now draw 10 shapes along x
  for( float x=0; x < 1 ; x+=0.1 )
  {
    s.draw( x, y ) ;
  }
}
```

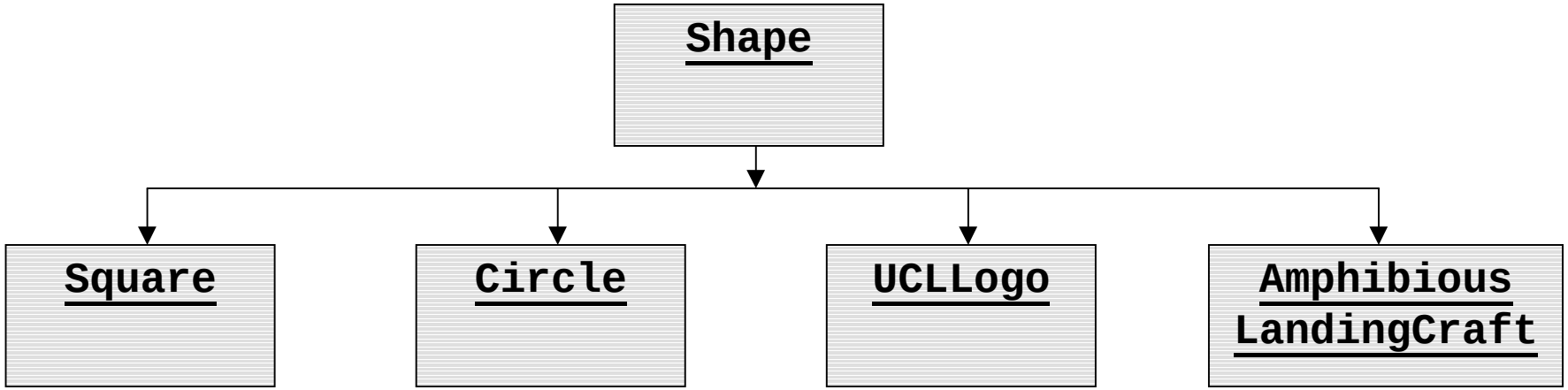


this code should draw something like this....

Dots are boring - no-one is buying our code !!!

We want to add the possibility of having lots of different shapes to draw.

⇒ **Lets make lots of shapes inherited from the Shape class !**



Here is the square class...

```
class Square: public Shape
{

  public:

   void draw( float x, float y ) ;
   void setSize( float size ) ;

  private:

   float length ;
};
```

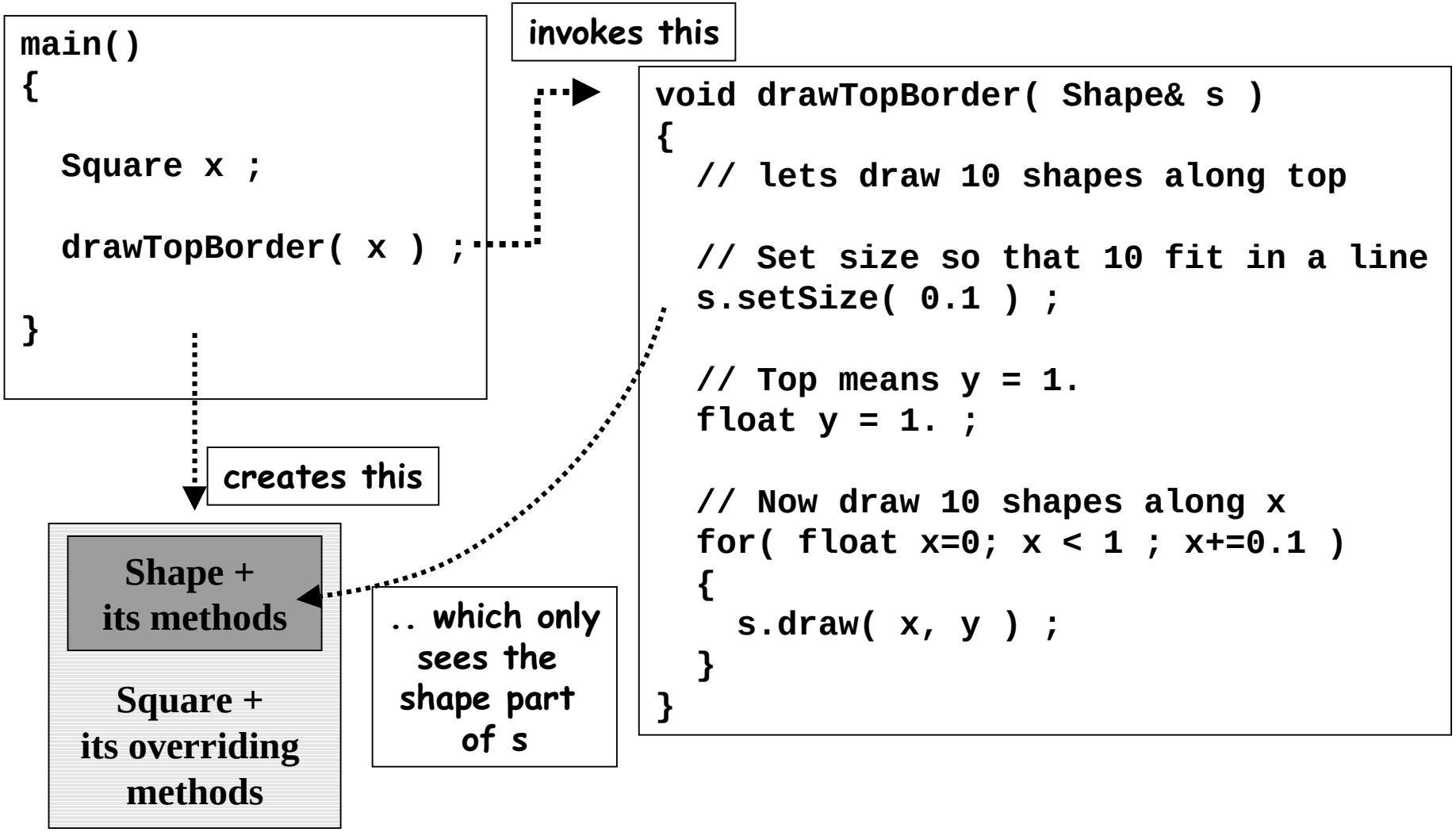
Note that we override the methods

we add a variable to characterise a square (lenght of side)

... and here are the overriding methods...

```
//.....................................
void Square::draw( float x, float y )
{
  // Use graphis library to draw 4 lines
 graphics.line( x, x + lengh, y, y ) ;
 graphics.line( x + length, x + length, y, y+length ) ;
 graphics.line( x, x , y, y + length ) ;
 graphics.line( x, x+length , y+length, y + length ) ;
}


//.....................................
void Square::setSize( float size )
{
  length = size ;
}
```

Now look at a modified main display program:



... so this code will still draw this as it only sees the default Shape methods....

This problem is solved by using:

`virtual methods`

These are a way of telling the system that:

- **an object may have its methods overridden**
- **therefore it should look to see if the object that has been passed is really a sub class**
- **if so then invoke the appropriate methods**

A method is made virtual by placing the `virtual` keyword before its signature

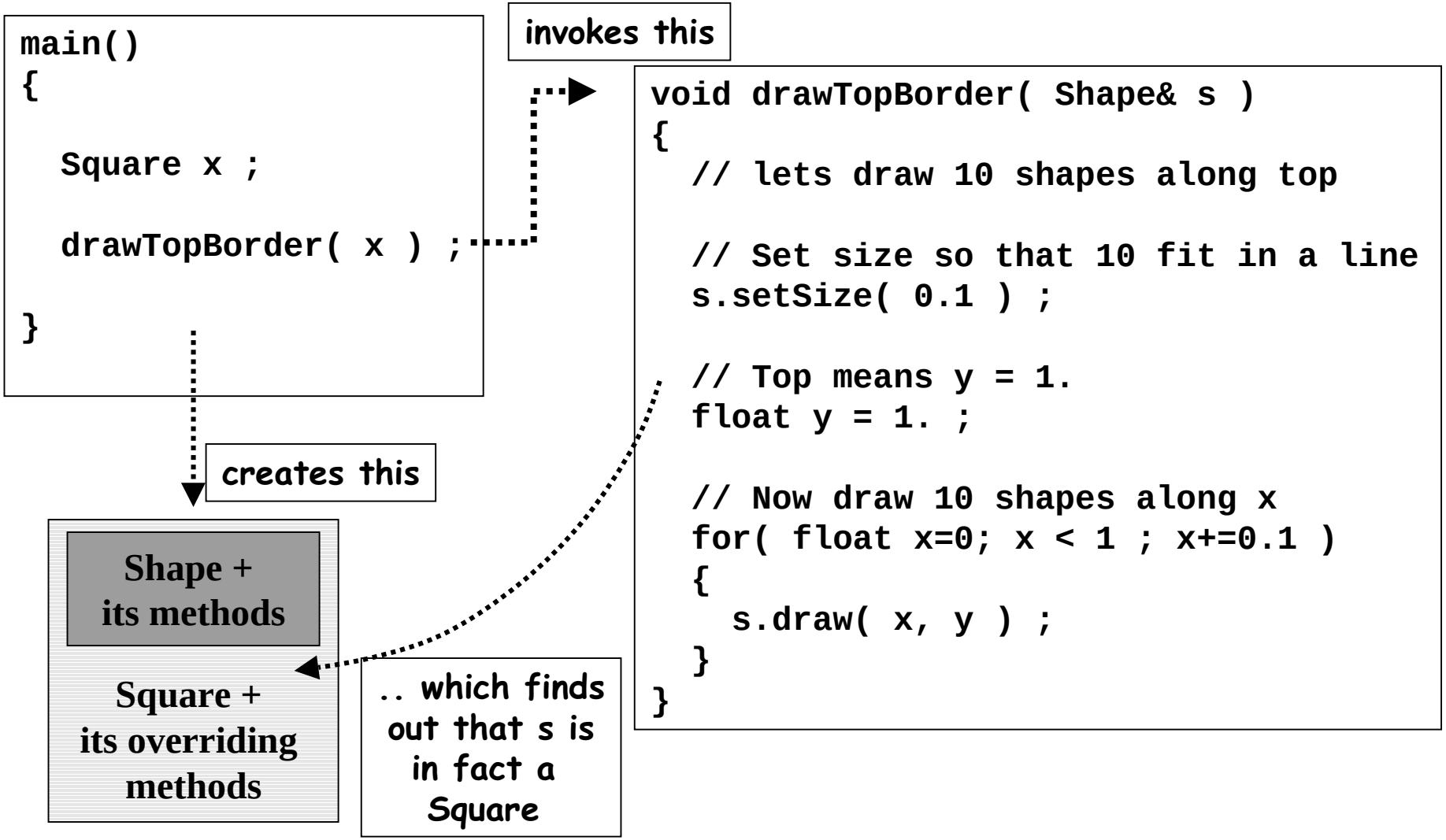
To do this you modify the Shape class like this...

```
class Shape
{
  public:
   virtual void draw( float x, float y ) ;
   virtual void setSize( float size ) ;
};
```

.... and similarly the methods...

```
//.................................
virtual void Shape::draw( float x, float y )
{
    .... same as before ...
}

//.....................................
virtual void Shape::setSize( float size )
{
    .... same as before ...
}
```

...now when you do this (this code is unchanged)...

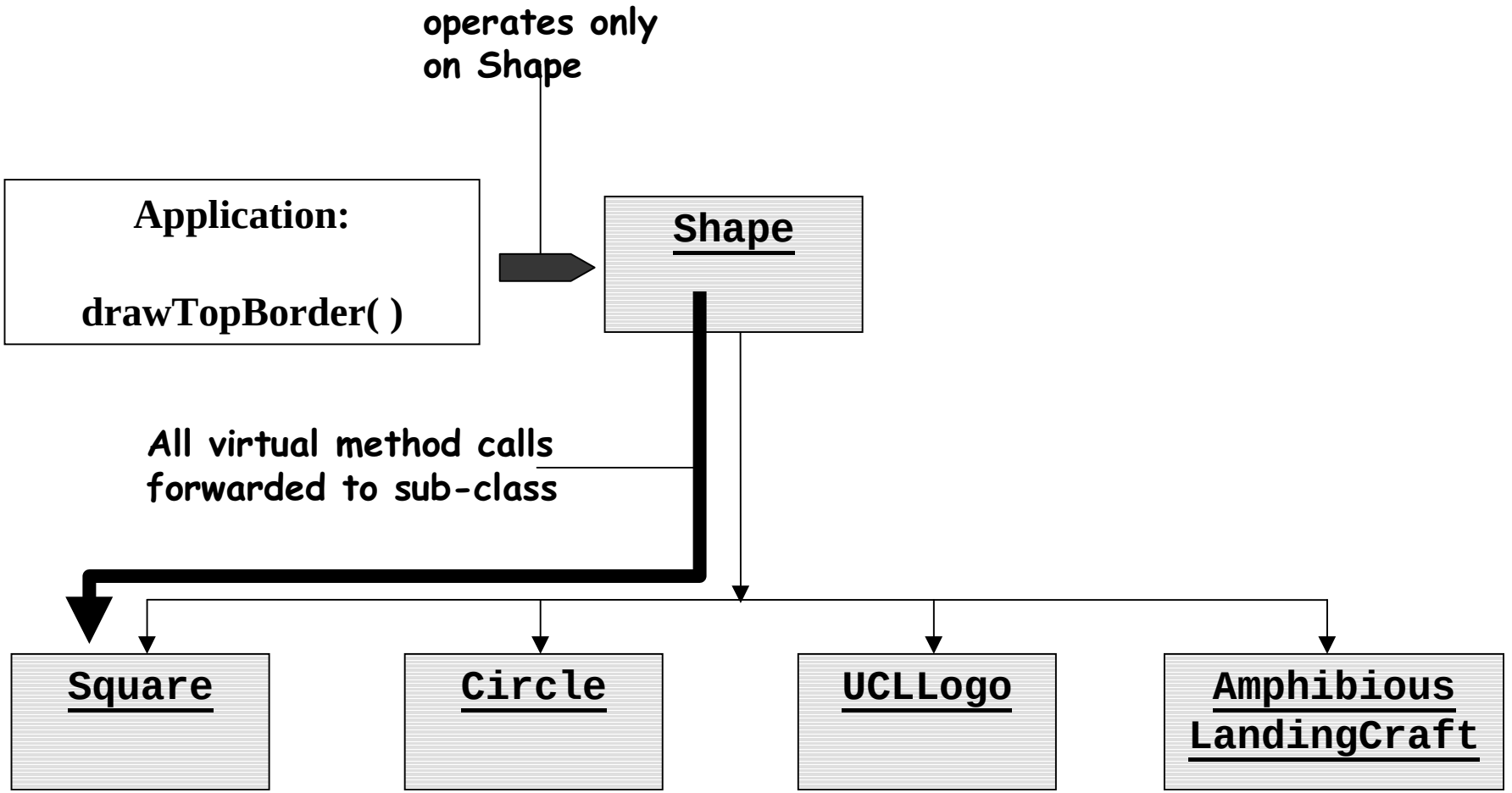


...and so finally ....

...and everyone buys our graphics program ....

This is called "Polymorphism"

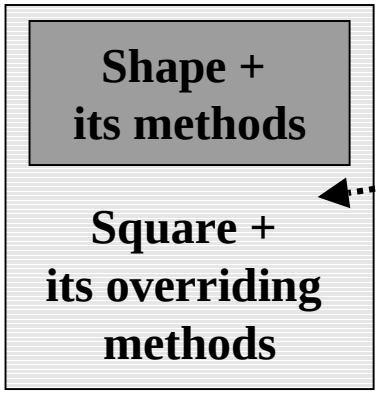
You are said to be using subclasses of Shape polymorphically



Technical point: You MUST have a "pointer" or a "reference" here !!!
Otherwise it just makes a local copy of the Shape part of Square

```
main()
{

  Square x ;

  drawTopBorder( x ) ;

}
```

```
void drawTopBorder( Shape& s )
{
  // lets draw 10 shapes along top

  // Set size so that 10 fit in a line
  s.setSize( 0.1 ) ;

  // Top means y = 1.
  float y = 1. ;

  // Now draw 10 shapes along x
  for( float x=0; x < 1 ; x+=0.1 )
  {
    s.draw( x, y ) ;
  }
}
```



Technical point: You MUST have a "pointer" or a "reference" here !!!
Otherwise it just makes a local copy of the Shape part of Square

```
main()
{

  Square x ;

  drawTopBorder( x ) ;

}
```

```
void drawTopBorder( Shape s )
{
  // lets draw 10 shapes along top

  // Set size so that 10 fit in a line
  s.setSize( 0.1 ) ;

  // Top means y = 1.
  float y = 1. ;

  // Now draw 10 shapes along x
  for( float x=0; x < 1 ; x+=0.1 )
  {
    s.draw( x, y ) ;
  }
}
```

Copy of Shape

Shape + its methods

Square + its overriding methods

Student exercise (easy option)

Write the classes to represent the inheritance structure shown below (for this example we use entities which are fairly trivially different)

Write and run a function which receives a `ThreeVector` reference and uses it to invoke a virtual method which prints the state of the actual sub-class entity passed in its argument

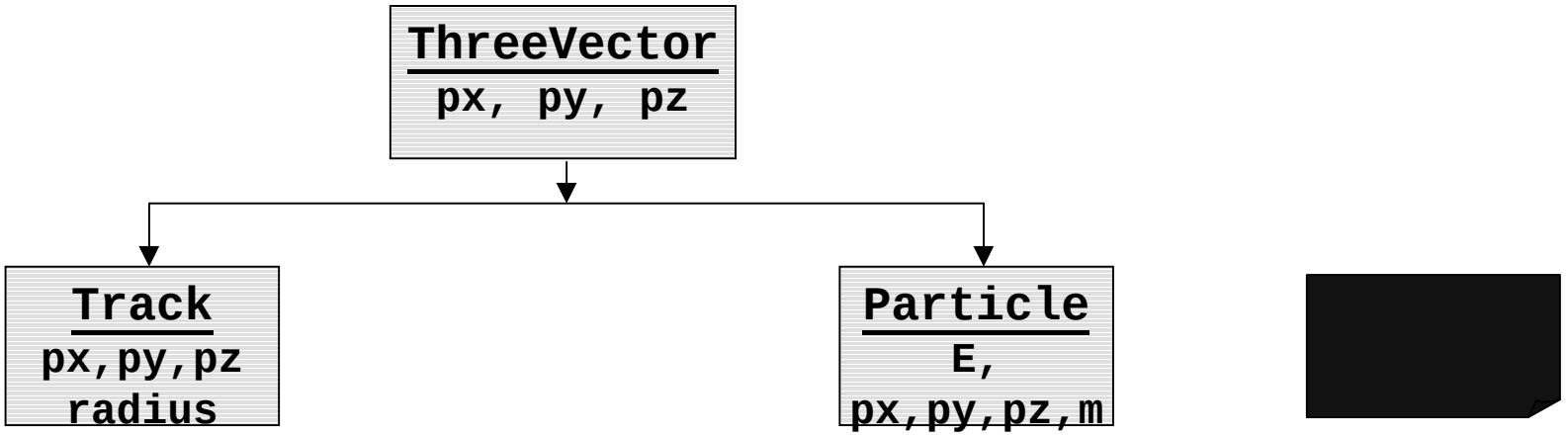
I.e. something like:

```
void printEntityState( ThreeVector& e ) {....}
```

You will have to make sure that you supply a

```
virtual void print( )
```

method in each of the base class and sub-classes



Student exercise (difficult option)

Modify the SuperHero code you wrote earlier so that methods are virtual

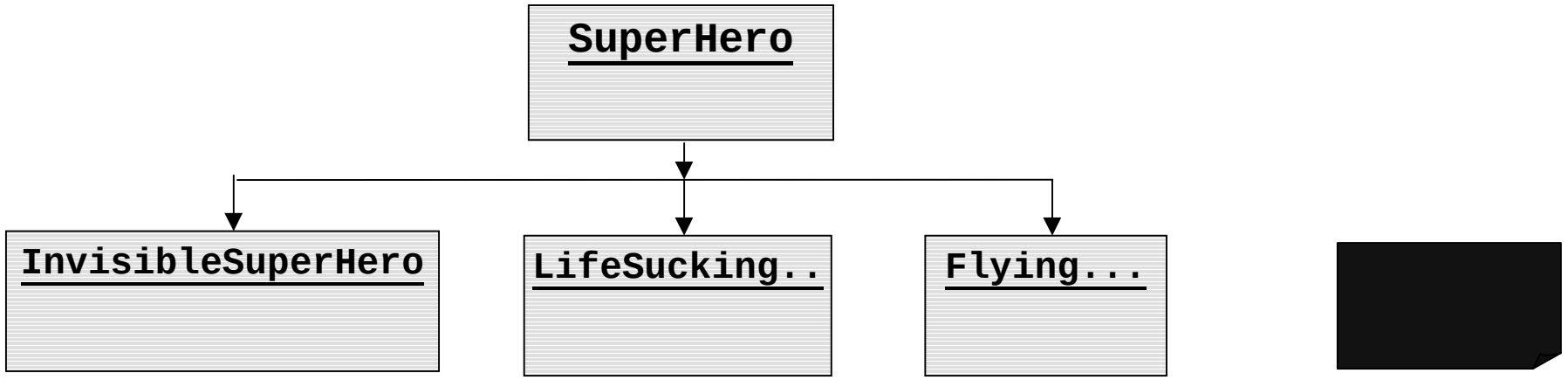
Modify the fight method such that its argument is always SuperHero&

Write a battle function

```
void battle( SuperHero& a, SuperHero& b )
```

Which takes two SuperHero's polymorphically. and makes them fight. It then tells the winner to print its name (you will need to add a method)

Write a main program to create some SuperHero's and then call the battle function, passing them as arguments in turn.



Summary of Module 11: Polymorphism & virtual methods

In this module we have covered the following topics.

- **Polymorphism**

  Use of sub-classes polymorphically via references or pointers to a base class

  Use of virtual methods and the virtual keyword