UCL

COMP1008 Inheritance

UCL

Outline

- Introduction to inheritance.
- How Java supports inheritance.

Inheritance is a key feature of object-oriented programming.

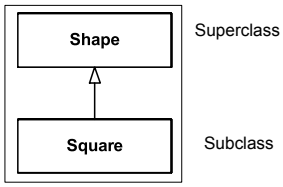
UCL

Inheritance

- Models the “kind-of” or “specialisation-of” or “extension-of” relationship between classes.
- Specifies that one class extends another class.
- For example:
  - A Square is a kind-of Shape.
  - A class Square can extend a class Shape.
  - A bus is a kind-of vehicle.
  - Integer is a specialisation of Number.
  - An EmailAddressString is an extension of String.
    - Email addresses have a specific format.

UCL

Subclass and Superclass

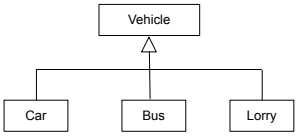


- A *subclass* inherits from a *superclass*.
- The subclass gains all the properties of the superclass, can specialise them and can add more.

UCL

Multiple Subclasses

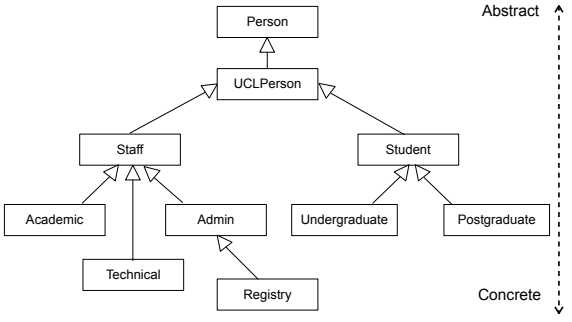
- Several subclasses can inherit from same superclass.



- Java supports single inheritance, one superclass only.
  - Some languages support multiple inheritance (MI), 2 or more superclasses, e.g., C++.
  - MI is complex and often seen as a bad idea.

UCL

Inheritance Hierarchy



Generalisation & Specialisation

- A superclass is a generalisation.
  - Shape defines the abstract properties of shapes in general.
  - Number defines the abstract behaviour of numbers.
  - Person defines common attributes (name, date of birth, etc.)
- A subclass is a specialisation.
  - Square represents a specific kind of concrete shape.
  - Integer, Double define specific kinds of number representation.
  - Undergraduate defines specific attributes (year, unit courses)

Abstract v. Concrete

- Abstract classes provide a partial or abstract description.
  - Not enough to create instance objects.
  - Define a common set of public methods that all subclass objects must have - common interface.
  - Variables/methods can be shared via inheritance.
    - Do not need to be duplicated in subclasses.
- Concrete classes provide a complete description.
  - Inherited + new attributes/methods.
  - Inherit shared interface.
  - Can create instance objects.

Shapes and Squares

- Assume all shapes:
  - have an x,y coordinate.
  - can be drawn.
  - can be moved to a new position.
  - Defined by class Shape.
- Class Square *extends* or *specialises* this basic behaviour for squares.
  - Allows squares to be drawn, moved, etc.
- Class Triangle and class Circle can do same for triangles and circles.

Shape v.1

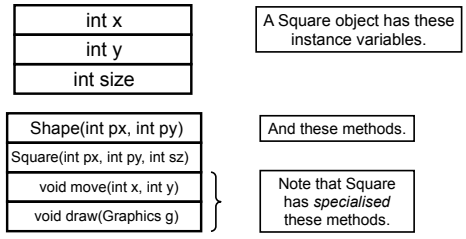
```
public class Shape
{
  private int x, y ;
  public Shape(int px, py)
  { ??? }
  public void draw(Graphics g)
  { ??? }
  public void move(int px, int py)
  { ??? }
}
```


Square v.1

```
public class Square extends Shape
{
  private int size ; // Need a size
  public Square(int px, int py, int sz)
  { ??? }
  public void draw(Graphics g)
  { ??? }
  public void move(int px, int py)
  { ??? }
}
```

New Keyword

Square objects



OK...

- Seen the basic idea but we have to fill in the details.
- And learn how to use inheritance correctly.

Health warning – inheritance is a powerful mechanism but easily misused.

Square Constructor

- Let's try:

```
public Square(int px, int py, int sz)
{
  x = px ;          // Uh Oh ...
  y = py ;
  size = sz ;
}
```

- x and y are inherited *but are private to Shape*.

Private and inheritance

- Private variables are inherited and are part of subclass objects.
- BUT they can be accessed by superclass methods *only*.
  - Encapsulation is respected.
- Subclass methods have no access.
- Problem?

protected

- Change Shape:

```
public class Shape
{
  protected int x, y;
  ...
}
```

- A *protected* variable can also be accessed from subclasses.

protected (2)

- Allows the selective weakening of strict encapsulation.
- But increases the *coupling* between super and sub classes.
  - Some believe this to be unacceptable.
  - Could use *getter* and *setter* methods instead (also called *accessor* methods).
  - int getX(), void setX(int), int getY(int), void setY(int)

Square Constructor (2)

- Can now write:

```
public Square(int px, int py, int sz)
{
  x = px ;          // OK
  y = py ;
  size = sz ;
}
```

- But we don't actually want to do this!!

Shape constructor

```
private int x, y ;
public Shape(int px, py)
{
   x = px ;
   y = py ;
}
```

- We actually want the Shape constructor to assign initial values to x and y.

Localise

- Shape declares x and y.
- Shape should initialise them.
- Don't want to scatter copies of the initialisation code around all the subclasses.

[Some would argue that x and y should only ever be accessed by Shape methods and, hence, must be private.]

Square Constructor (3)

```
public Square(int px, int py, int sz)
{
 // What about x and y?
   size = sz ;
}
```

- Now need a mechanism to call the Shape constructor.

Square Constructor (4)

```
public Square(int px, int py, int sz)
{
   super(px,py) ; // Another new keyword
   size = sz ;
}
```

- *super* is a reference to the superclass.
- When used in a constructor like this, it results in a call to the superclass constructor with the matching parameter list.

Super goes first

```
public Square(int px, int py, int sz)
{
   size = sz ;  // Error
   super(x,y) ;
}
```

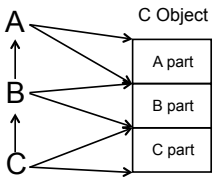
- Super must be *first* statement in the constructor body.

Creating Square Objects

```
Square sq = new Square(1,1,10) ;
```

- Turns out to be a multi-stage process:
  - Allocate memory for object.
  - Call Square constructor.
  - Call Shape constructor *before* anything else is done by the Square constructor.
  - Execute rest of Square constructor.
  - Return reference to newly created *and* initialised object.

Subclass object initialisation - the general case



- A constructor must be called for each inherited part of a C object.
- And the constructor bodies executed in the order A, B, C.

Guaranteed

- Superclass constructors must called and in the correct order.
- The language guarantees this will happen.
- Initialisation must be done!

It will happen...

```
public C()
{              // Look, no super.
   v = 10 ; // some instance variable
}
```

- The compiler adds a call to super (no parameters)

```
public C()
{
   super() ; // added during compilation
   v = 10 ;  // some instance variable
}
```


What if?

- Try:

```
public Square(int px, int py, int sz)
{
   size = sz ;
}
```

- Compiler does its bit to give:

```
public Square(int px, int py, int sz)
{
   super() ;
   size = sz ;
}
```


But...

```
public Square(int px, int py, int sz)
{
   super() ; // no way...
   size = sz ;
}
```

- This would call the Shape constructor that takes no arguments.
- Except there isn't one, so it can't be called, so you get a compilation error.

Super may have to be used

```
public Square(int px, int py, int sz)
{
   super(px,py) ;
   size = sz ;
}
```

- An explicit super must be used here, with the correct arguments.

Questions?

Back to class Shape

- This method was suggested:
  public void draw(Graphics g)
  { ??? }
- What goes in the method body?
- Well, nothing useful. A Shape is an abstract rather than concrete kind of thing.
- A Shape doesn't have a shape that can be drawn!

Option 1

- Simply leave the method body empty.

```
public void draw(Graphics g)
{
   // do nothing
}
```

- Default drawing is to draw nothing.

Option 2

- Delete the draw method altogether.
- But this would be a bad move.
  - The method must be present to be specialised by subclasses.
  - Want the method to be part of the public method interface of Shape, so it can be used with all types of shape.
  - Guarantee that all shapes have a draw method.

Option 3

- Declare the method *abstract*.

```
public abstract void draw(Graphics g) ;
```

- No method body is given.
  - Note where semi-colon is.
  - No braces.
- Put down a marker that the method must exist in subclasses.

Consequences

- A class containing an abstract method *cannot* have instance objects.
- It does not provide a complete description of instance objects.
- But that is OK - we don't want instances of class Shape.

Abstract class

- Declaring an abstract method actuallly forces the *class* to be declared abstract as well.

```
public abstract class Shape
{
  ...
}
```

- An abstract class can have no instances.
- It is a partial description that can be inherited.
- Declares shared instance variables/methods.

Move method

- Shape can provide a default or *shared* implementation:

```
public void move(int px, int py)
{
    x = px ;
    y = py ;
}
```

- Square may specialise this method, but it doesn't actually have to.
  - Just inherit unchanged.

Class Shape v.2

```
public abstract class Shape
{
  protected int x, y ;
  public Shape(int px, int py)
  { x = px ; y = py ; }
  public abstract void draw(Graphics g) ;
  public void move(int px, int py)
  { x = px ; y = py ; }
}
```


Square class ?

- Need to declare a draw method body to draw a square.
- Don't need to declare a move method at all.
  - The inherited version is good enough.
- This will create a complete class.
- Instance objects can be created.

Questions?

Using Shapes

- Create a Square object and use it:

```
Square sq = new Square(5,5,50) ;
sq.draw(g) ;       // g references a
                   // Graphics object
sq.move(25,25) ;
sq.draw(g) ;
// Shape shape = new Shape(5,5) ; NO!!
```

Shape is abstract.
No instances.

move

- If the move method was inherited and *not* specialised by Square.

```
sq.move(25,25) ;
```

- A call to move executes the move method declared in class Shape.

draw

- The requirement to provide a draw method was inherited by Square.
- Square specialised the method by re-declaring it with a complete method body.
- This is called *overriding*.
- Square *overrides* the draw method.
  - Don't confuse with overloading.
  - An overriding method must be declared in a subclass, have the same name, parameters and return type.

draw (2)

```
sq.draw(d) ;
```

- A call to draw executes the draw method declared in class Square.
- And draws a square.

Overriding move

- Suppose Square also overrides move?
- What does sq.move(20,20) do?

- Calls the move method defined by Square.

Why?

```
sq.move(25,25) ;
```

- When a method call is made, the method executed depends on the *class of the object it is called for*.
- The class is Square. If it provides move, then execute it.
- If not, then go to the superclass and see if it provides move.
- If neither class Square or Shape (or any other superclass) provides move then an error.
  - In fact, the compiler will not compile the code.
  - The compiler can check that a move method is defined somewhere in the inheritance chain.

Superclass references

- What about this?

```
Shape sh = new Square(10,10,40) ;
```

- This is legal!
- *A superclass reference to a subclass object.*
- Reference type is different from object type, but related by inheritance.

Public interface

Shape sh = new Square(10,10,40) ;

- Shape defined a set of public methods inherited by Square.
  - Either complete or abstract.
- Square must have the *same* public methods.
  - Or can override a method.
- Anything that can be done with a Shape can be done with a Square.
  - Any method that can be called using a Shape reference must be available on a Square object.
- Square will specialise what happens.

sh.draw

Shape sh = new Square(10,10,40) ;
sh.draw(g) ; //OK

- Class of object referenced by sh is Square, so find Square.draw and execute it.
- The type of the reference sh is Shape.
  - The code will compile as class Shape declares a draw method.
  - But the method called at runtime is determined by the class of the object referenced not the type of the reference.
    - Static v. Dynamic typing.

sh.move

Shape sh = new Square(10,10,40) ;
sh.move(20,20) ;

- Class of object referenced by sh is Square, so check for Square.move and execute it if it exists.
- Otherwise use Shape.move.

rotate?

- Suppose a rotate method is added to Square but not Shape?

Shape sh = new Square(10,10,40) ;
sh.rotate(50) ;

- Error!
- Shape does not define a rotate method.
- So can't be called via a Shape reference, even though the object has one.
  - Compiler will say that class Shape does not define a method called rotate.
- Can be called via a reference of type Square.

Dynamic binding

- *Binding* is the term used for the process of mapping a method call to a method body that can be executed.
- *Dynamic binding* means that the method body is determined at runtime by looking at the class of the object the method is called for.

Instance methods

- Instance methods are always dynamically bound.
- Look at the class of the object a method is called for.
  - If it provides a method body, execute it.
  - Otherwise go to superclass(es) and repeat.
  - If not found then report an error.
    - Note that the error is located and reported by the compiler, not at runtime.
    - Runtime is used to determine which method, but a method must exist.
    - In some languages *method lookup* is entirely dynamic and a program can fail at runtime when a method is not found.

Static binding

- Static methods are statically bound.
- This means the method body to be executed is always uniquely determined.
- And can be determined when the program is compiled.
- (The same can be done for instance methods if no overriding has taken place.)

Questions?

So…

- We have class Shape
- And subclasses like Square, Circle, Triangle
- Method binding.
- What use is this?

Remove duplication

- A superclass holds common variable and method declarations.
- Code does not have to be duplicated in subclasses.
- Implementation inheritance.

Everything is a Shape

- Can treat all subclass objects as Shapes.
- The ability to be a Shape is inherited and specialised.
- Code can use Shapes without knowing what specific kind of Shape.
- Interface inheritance.

Storing Shapes

```
// Array of Shape references
Shape[] shapes = new Shape[50];
// Can reference mixed collection of subclass objects
Shapes[0] = new Square(2,3,4);
Shapes[1] = new Triangle(3,4,5,6);
Shapes[2] = new Circle(5,6,8);
Etc…
Or an ArrayList:
    ArrayList<Shape> = new ArrayList<Shape>();
```

Drawing Shapes

```
for (Shape shape : shapes)
{
  shape.draw(g);
}
```

- All shape subclasses can draw but don't need to know which subclass is being used.
- Shape declares draw, inherited and specialised by all subclasses.
- Program using methods declared by class Shape but any subclass object can be used when code is run.

Generic Code

- Most application code can be written to use the superclass type (Shape).
  - Subclasses not named.
  - Any Shape subclass object will fit.
- Small section(s) of code must name subclasses and create objects.
- But rest of code independent of subclass details.

Big Advantage

- Most code uses superclass types and methods declared by superclasses.
- Most code doesn't have to change if subclasses change.
- The affects of changes to subclasses localised.

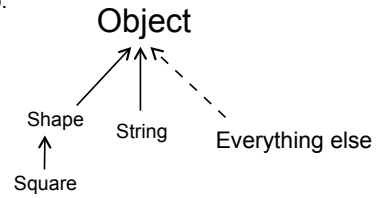
Questions?

Object

- What does class Shape inherit from?
- Nothing was specified but...
- All classes either directly or indirectly inherit from class Object.
- Including Shape, even though we didn't say so.

Java Inheritance Tree

- All Java classes you ever use or write yourself are in the *inheritance tree* with class Object at the top:



Everything is an Object

```
Object obj = new Square() ;
```

- OK
- But can only call methods declared by class Object.
- Of course, they may be overridden by subclasses.

Class Object?

- Provides a small set of methods that *all* classes inherit and can be called for *all* objects.
- For example, the toString() method.
- See the text book for more details.

Object array

```
Object[] elements = new Object[n];
```

- Array elements can reference *any* kind of object.
- ArrayList and other data structures depend on this.
  - void add(Object obj);
  - Object get(int n);

Is that all?

- No!
- There are yet more important details about inheritance not covered here.
- We'll briefly look at a few more details but refer to the course text book for more information.

Summary so far (not the end…)

- Inheritance allows one class to extend another.
- Rules enforce the behaviour of constructors.
- Dynamic binding determines what methods are executed.
- All objects are Objects!

Calls to self

- As we know, an object can call methods on itself (i.e., a method can can call another method of the same class for the same object).

```
public void f()   // instance method
{
   ...
   g() ;  // another instance method
}         // of the same class
```

Calls to self (2)

g() ;

- equivalent to:

this.g() ;

- In fact, g can be a superclass method, if the current class has not overridden g.

Calls to self (3)

super.g() ;

- An overridden (and otherwise hidden) superclass method can be called using super.
- Must be public or protected, though.

Super and variables

- A subclass can hide an inherited instance variable by declaring its own instance variable of the same name.
- Super can be used to access the hidden variable (if public or protected):

super.x = 10 ;

More on super?

See the text book!

Template method

- A *superclass* method can have the form:

```
public void doSomething()
{
  doThis() ;
  ... // Whatever
  doThat() ;
}
```

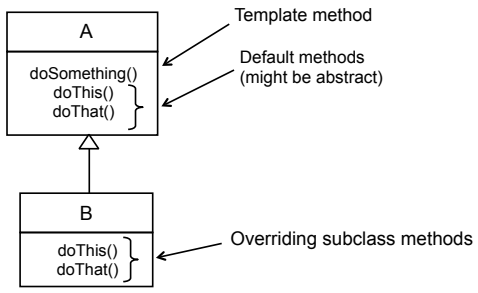

Template method (2)

- A *subclass* might override doThis and doThat methods but not doSomething.
- This allows doSomething to define an algorithm that can be partly specialised by a subclass.
- In other words doSomething acts as a template.
- Example:

```
while (!endOfGame())
{
  playturn();
  getscore();
}
```

High level control, specialised by subclass.

Template method (3)



final (again)

- The final keyword can be used to prevent inheritance.
- Declaring a class final:

  public final class X { }
- prevents the class from being subclassed.

final (yet more)

- Declaring a method final stops it being overridden:

```
public final void doSomething()
{
  doThis() ;
  ... // Whatever
  doThat() ;
}
```

- doThis and doThat can be overridden but not doSomething.

Why use final?

- Gives the class programmer control.
- Not all classes or methods are designed to be subclassed or overridden.
- Can explicitly enforce design decisions.
- But be wary of making classes/methods final as it can make testing a lot harder.

Good Practice

- Inheritance should only be used when a subclass is really an extension of a superclass.
- It should be possible to use a subclass object where the superclass has been specified.

The Contract

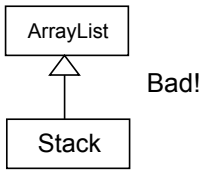
- A subclass extends and specialises but implements the *contract* specified by the superclass.
  - Object behaviour should be consistent.
  - Methods should behave consistently.

If in doubt…

- Don't use inheritance.
  - A class wants to use another class but is not an extension of the other class.
- Use association instead (i.e., an object reference).
  - A class uses another class.
  - Often called delegation.
  - Delegate to another class to provide a service.

Stacks (again!)

- A stack can be implemented using an ArrayList.
- So might inherit ArrayList and add push, pop, etc. methods…



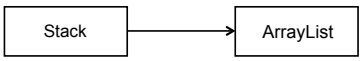
Bad!

But…

- Stack subclasses inherits all the ArrayList public methods, as well as the ability to store a collection of objects.
- Don't want the public methods – not part of the stack abstraction.
  - For example, inserting into middle of stack is not a stack operation.
- A stack is *not* an extension of an ArrayList.

Instead

- A Stack class should use an ArrayList by association.
  - Exploit the container properties, ignore the interface.



Stack has a private reference to ArrayList.
Users of Stack unaware of implementation.

Reminder - Why have inheritance?

- Allows classification hierarchies.
- Enables the use of common interfaces.
- Enables implementation sharing (by extension, not copy and edit).

Finally

- Covered a lot of ground.
- Introduced inheritance and its realisation in Java.
- Investigated some of the details.
- Considered good v. bad inheritance.