

# 1B1b Inheritance

1 Copyright © 2004, Graham Roberts Department of Computer Science

## Agenda

- Introduction to inheritance.
- How Java supports inheritance.

Inheritance is a key feature of object-oriented programming.

2 Copyright © 2004, Graham Roberts Department of Computer Science

## 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.

3 Copyright © 2004, Graham Roberts Department of Computer Science

## Subclass and Superclass

```
classDiagram
    class Shape
    class Square
    Shape <|-- Square
```

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

4 Copyright © 2004, Graham Roberts Department of Computer Science

## Generalisation & Specialisation

- A superclass is a generalisation.
  - Shape defines the *abstract* properties of shapes in general.
- A subclass is a specialisation.
  - Square represents a specific kind of *concrete* shape.

5 Copyright © 2004, Graham Roberts Department of Computer Science

## Shapes and Squares

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

6 Copyright © 2004, Graham Roberts Department of Computer Science

### 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)
    { ??? }
}
```

7 Copyright © 2004, Graham Roberts Department of Computer Science

### Square v.1

New Keyword

```
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)
    { ??? }
}
```

8 Copyright © 2004, Graham Roberts Department of Computer Science

### Square objects

int x
int y
int size

A Square object has these instance variables.

Shape(int px, int py)
Square(int px, int py, int sz)
void move(int x, int y)
void draw(Graphics g)

And these methods.

Note that Square has *specialised* these methods.

9 Copyright © 2004, Graham Roberts Department of Computer Science

### 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.

10 Copyright © 2004, Graham Roberts Department of Computer Science

### 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*.

11 Copyright © 2004, Graham Roberts Department of Computer Science

### Private and inheritance

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

12 Copyright © 2004, Graham Roberts Department of Computer Science

## protected

- Change Shape:

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

- A protected variable can also be accessed from subclasses.

13 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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)`

14 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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!!

15 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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.

16 Copyright © 2004, Graham Roberts

Department of Computer Science



## Localise

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

[As we have seen, some would argue that x and y should only ever be accessed by Shape methods and, hence, be private.]

17 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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.

18 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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.

19 Copyright © 2004, Graham Roberts

Department of Computer Science



## Creating Square Objects

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

- Turns out to be a multi-stage process:
  - Allocate space 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.

20 Copyright © 2004, Graham Roberts

Department of Computer Science



## Super goes first

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

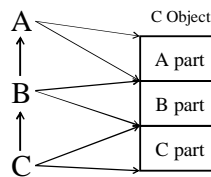
- Super *must* be first thing in the constructor body.

21 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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.

22 Copyright © 2004, Graham Roberts

Department of Computer Science



## Guaranteed

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

23 Copyright © 2004, Graham Roberts

Department of Computer Science



## It will happen...

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

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

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

24 Copyright © 2004, Graham Roberts

Department of Computer Science



## What if?

- Try:
 

```
public Square(int px, int py, int sz)
{
    size = sz ;
}
```
- Compiler does its bit:
 

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

25 Copyright © 2004, Graham Roberts

Department of Computer Science



## Wait...

- ```
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.

26 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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, with the correct arguments.

27 Copyright © 2004, Graham Roberts

Department of Computer Science



## Questions?

28 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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!

29 Copyright © 2004, Graham Roberts

Department of Computer Science



## Option 1

- Simply leave the method body empty.
 

```
public void draw(Graphics g)
{
    // do nothing
}
```
- Default drawing is to draw nothing.

30 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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.

31 Copyright © 2004, Graham Roberts

Department of Computer Science



## Option 3

- Declare the method *abstract*.
 

```
public abstract void draw(Graphics g) ;
```
- No method body is given.
- Put down a marker that the method must exist in subclasses.

32 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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`.

33 Copyright © 2004, Graham Roberts

Department of Computer Science



## Abstract class

- Declaring an abstract method 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.

34 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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.

35 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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 ; }
}
```

36 Copyright © 2004, Graham Roberts

Department of Computer Science



## Square class ?

- Only need to write 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.

37 Copyright © 2004, Graham Roberts

Department of Computer Science



## Questions?

38 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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!!
```

39 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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.

40 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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.

41 Copyright © 2004, Graham Roberts

Department of Computer Science



## draw (2)

- ```
sq.draw(d) ;
```
- A call to draw executes the draw method declared in class Square.
  - And draws a square.

42 Copyright © 2004, Graham Roberts

Department of Computer Science



## Overriding move

- Suppose Square also overrides move?
- What does `sq.move(20,20)` do?
- Calls the move method defined by Square.

43 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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.

44 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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.

45 Copyright © 2004, Graham Roberts

Department of Computer Science



## Public interface

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

- Shape defined a set of public methods inherited by Square.
- Square has the *same* public methods.
- Anything that can be done with a Shape can be done with a Square!
- Square will specialise what happens.

46 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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.

47 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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`.

48 Copyright © 2004, Graham Roberts

Department of Computer Science



## rotate?

- Suppose a rotate method is added to Square?  

```
Shape sh = new Square(10,10,40);  
sh.rotate(50);
```
- Error!
- Shape doesn't define a rotate method.
- So can't be called, even though the object has one.

49 Copyright © 2004, Graham Roberts

Department of Computer Science



## Reference type matters

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

- Only public methods declared in Shape can be called, regardless of class of object.
- So the reference type defines what can be called but not exactly which method gets executed.

50 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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.

51 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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.

52 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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.)

53 Copyright © 2004, Graham Roberts

Department of Computer Science



## Questions?

54 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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.

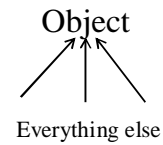
55 Copyright © 2004, Graham Roberts

Department of Computer Science



## Inheritance Tree

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



56 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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.

57 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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.

58 Copyright © 2004, Graham Roberts

Department of Computer Science



## Why have inheritance?

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

59 Copyright © 2004, Graham Roberts

Department of Computer Science



## Questions?

60 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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!

61 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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.

62 Copyright © 2004, Graham Roberts

Department of Computer Science



## Calls to self

- An object can call methods on itself (i.e., a method 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
```

63 Copyright © 2004, Graham Roberts

Department of Computer Science



## Calls to self (2)

- equivalent to:
 

```
g();
this.g();
```
- In fact, g can be a superclass method, if the current class has not overridden g.

64 Copyright © 2004, Graham Roberts

Department of Computer Science



## Recursion

- A method can call itself - *recursion*!

```
int factorial(int i)
{
    if (i > 0)
        return i * factorial(i-1);
    else
        return 1;
}
// See text book p199
```

65 Copyright © 2004, Graham Roberts

Department of Computer Science



## Recursion & iteration

- Recursion provides another way of doing iteration.
- Most use of recursion can be re-written using loops and vice-versa.
- Some algorithms are much easier to express using recursion.

66 Copyright © 2004, Graham Roberts

Department of Computer Science



## Another example

```
int sum(final int n)
{
    if (n == 0)
    {
        return 0 ;
    }
    else
    {
        return sum(n-1) + n ;
    }
}
```

Sum the numbers from 1 to n.

67 Copyright © 2004, Graham Roberts

Department of Computer Science



## Recursive calls made by sum

```
call sum(5)
sum(5): n != 0, call sum(4)
sum(4): n != 0, call sum(3)
sum(3): n != 0, call sum(2)
sum(2): n != 0, call sum(1)
sum(1): n != 0, call sum(0)
sum(0): n == 0, return 0
sum(1): return 0 + 1 (1)
sum(2): return 1 + 2 (3)
sum(3): return 3 + 3 (6)
sum(4): return 6 + 4 (10)
sum(5): return 10 + 5 (15)
```

68 Copyright © 2004, Graham Roberts

Department of Computer Science



## Cost of recursion

- Each recursive call has the 'cost' of a method call.
  - Recursion is potentially slower than iteration.
  - But a recursive algorithm may be more efficient.
- Each recursive call uses up a bit of memory.
- Too many recursive calls will use up all the stack space and the program will fail.

69 Copyright © 2004, Graham Roberts

Department of Computer Science



## Calls to self (3)

```
super.g() ;
```

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

70 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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 ;
```

71 Copyright © 2004, Graham Roberts

Department of Computer Science



## More on super?

See the text book!

72 Copyright © 2004, Graham Roberts

Department of Computer Science



## Template method

- A *superclass* method can have the form:

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

73 Copyright © 2004, Graham Roberts

Department of Computer Science



## Template method (2)

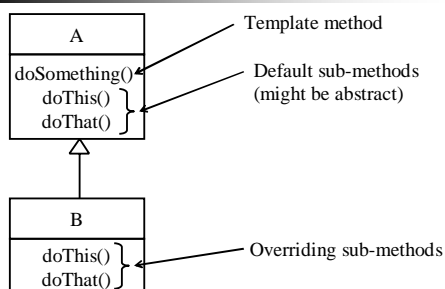
- A *subclass* might override doThis and doThat methods but not doSomething.
- This allows doSomething to define an algorithm that can be partly specialised by a subclass.
- In other words doSomething acts as a template.

74 Copyright © 2004, Graham Roberts

Department of Computer Science



## Template method (3)



75 Copyright © 2004, Graham Roberts

Department of Computer Science



## final

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

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

76 Copyright © 2004, Graham Roberts

Department of Computer Science



## final (2)

- Declaring a method final stops it being overridden:

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

- doThis and doThat can be overridden but not doSomething.

77 Copyright © 2004, Graham Roberts

Department of Computer Science



## Why final?

- Gives the class programmer control.
- Not all classes or methods are designed to be subclassed or overridden.
- Can explicitly enforce design decisions.

78 Copyright © 2004, Graham Roberts

Department of Computer Science



## Questions?

79 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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.

80 Copyright © 2004, Graham Roberts

Department of Computer Science



## The Contract

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

81 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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.

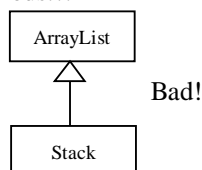
82 Copyright © 2004, Graham Roberts

Department of Computer Science



## Stacks (again!)

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



83 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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.

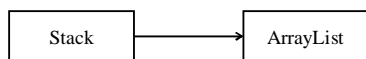
84 Copyright © 2004, Graham Roberts

Department of Computer Science



## 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.

85 Copyright © 2004, Graham Roberts

Department of Computer Science



## Finally

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

86 Copyright © 2004, Graham Roberts

Department of Computer Science

