

Week 9

- **Abstract Classes**
- **Interfaces**
- **Arrays (Assigning , Passing, Returning)**
- **Multi-dimensional Arrays**

1

Abstract Classes

- Suppose we have derived **Shape** and **Circle** subclasses from the superclass **Shape**.
- We may decide computing area is an operation all subclasses of **Shape** should provide.
- By making it a superclass operation, we can compute the area of all such objects in a *polymorphic* way as in:

```
Shape [] s = new Shape[5];
double[] areas = new double[5];
s[0] = new Circle(...);
s[1] = new Square(...);
...
for (int i=0; i<5; i++)
    areas[i] = s[i].computeArea();
```

2

Abstract method

- However, we cannot compute the area of shape, as shape is an abstract entity.
- We can make the `computeArea()` return a default value of 0.0. However, the subclass may forget to override it resulting in a wrong value
- To force the subclass to override we can declare `computeArea()` as an abstract method as in:

```
public abstract void computeArea();
```

3

Abstract vs Concrete class

- Note an *abstract* method has no implementation.
- You cannot construct objects of classes with *abstract* methods - called an *abstract class*.
- In Java, you must declare all abstract classes explicitly with the keyword *abstract*.
- If a subclass of `Shape` such as `Square` overrides this method providing an implementation, and has no other *abstract* methods then `Square` will be considered a *concrete class*.
- Then we can create instances of `Square` class.

4

```

abstract class Shape {
    ...
    public abstract double computeArea();
    public abstract void print();
    String color;
}
class Square extends Shape{ // concrete class
    public Square(double xt, double yt, double size) {
        ...
    }
    public double computeArea() {
        return Math.PI * radius * radius;
    }
    ... // other methods and instance variables
}

Shape aShape;           // OK
aShape = new Shape();   // Error. Cannot instantiate
aShape = new Square(...); // OK
aShape = null;          // OK

```

5

Abstract Class: Some points

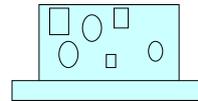
- Possible to declare a class abstract even it has no *abstract* methods - preventing users instantiating
- Note that unlike *interfaces*, which we will study next, *abstract classes* can have instance variables.
- Shape may have an instance variable for foreground colour.

6

A possible scenario

Suppose we have created classes for modelling a square and a circle shape as shown below. Assume these classes are fully tested and used in some existing applications.

```
abstract class Shape {
    public abstract double computeArea();
}
class Circle extends Shape {
    ...
}
class Square extends Shape{
    ...
}
```



Now an applet based application for children requires **Circle** and **Square** objects but with the ability to draw these objects on a **Graphics** object, through a method such as **draw(Graphics g)**.

7

Design possibilities ...

- we may add a **draw(Graphics g)** method in the **Circle** and **Square** classes.
- As **draw()** cannot be used in console mode, adding this method makes the class less generic.
- Besides as **Circle** and **Square** classes are fully tested it will not be wise to alter these classes.

A better solution is to create an interface (say **Drawable**) with one method **draw(Graphics g)** which can be implemented by the **Circle** and **Square** derived classes.

Wait a minute ... what is an interface ?

8

What is an interface?

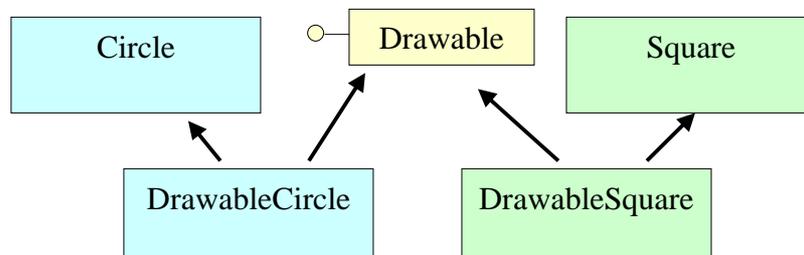
- A useful construct in Java is an *interface*, which defines a number of *abstract* operations.
- Interface is similar to an abstract class but it is not allowed to have any *instance variables*.
- The interface methods must be defined in the classes implementing the interface.
- In our problem we provide an *interface* **Drawable** with only one method, as in:

```
interface Drawable{  
    public void draw(Graphics g);  
}
```

9

Design using an interface

Now to provide **Circle** and **Square** classes which can be drawn on a **Graphics** object we extend the existing classes making them implement the **Drawable** interface.



10

```

class DrawableCircle extends Circle implements Drawable{
    public DrawableCircle(double x, double y, double rad) {
        super(x,y,rad);
    }
    public void draw(Graphics g) {
        g.drawOval((int)(getXCentre()-getRadius() ),(int)(getYCentre()-getRadius()),
            2*(int)getRadius(), 2*(int)getRadius());
    }
}
class DrawableSquare extends Square implements Drawable{
    public DrawableSquare(double xt, double yt, double size) {
        super(xt,yt,size);
    }
    public void draw(Graphics g) {
        g.drawRect((int)getXTop(), (int) getYTop(),
            (int)getSize(),(int)getSize());
    }
}

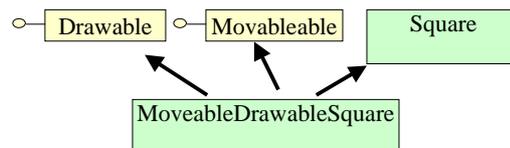
```

11

Java and Multiple Inheritance

- Java does not allow us to extend more than one class.
- If we have a class for student (**Student**) and employee (**Employee**) we will not be allowed to create a new class for a student who is both working and studying as in:

```
class EmployedStudent extends Student, Employee {
```
- However we can overcome this to some extent as we can extend one class and implement as many *interfaces* as we need as shown below.



12

Using an array of interfaces

The `paint()` below constructs `DrawableCircle` and `DrawableSquare` objects storing them in an array of superclass references before computing their areas.

Finally it calls `drawAllObjects()` passing the array of shape references, number of elements and `Graphics` object.

```
public void paint(Graphics g)    {
    Shape[] s = new Shape[5];
    s[0] = new DrawableCircle(50,100,100);
    s[1] = new DrawableSquare(75,150,100);
    s[2] = new DrawableCircle(90,150,250);
    s[3] = new DrawableSquare(40,200,100);

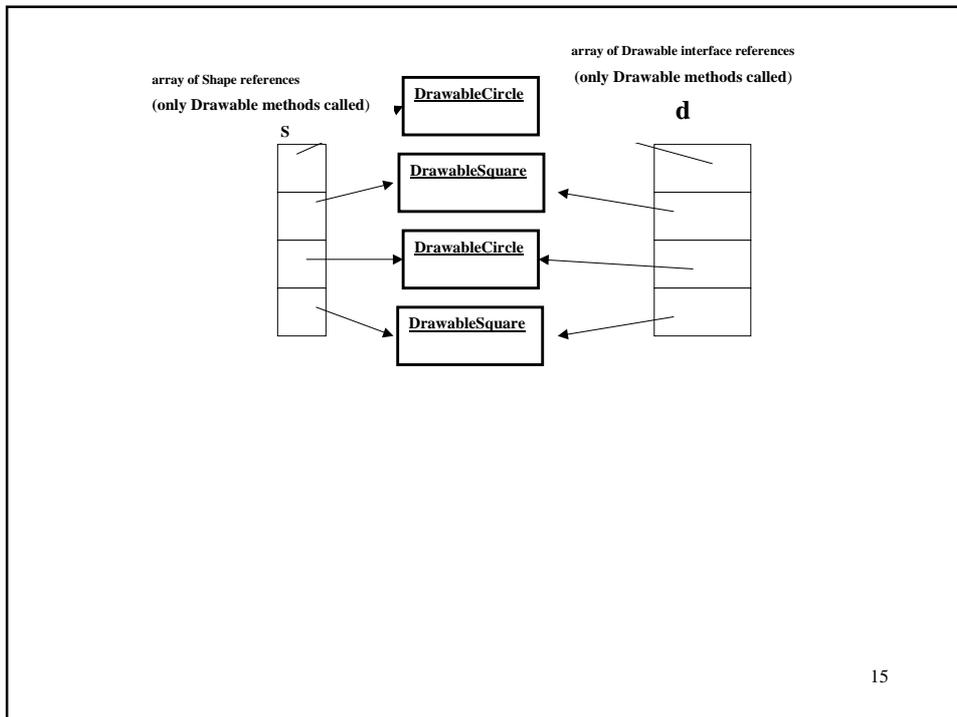
    for (int i=0; i<4; i++)
        g.drawString("Area of shape " + i + " is "
                    + s[i].computeArea(),100,100+50*i);
    drawAllObjects(s,4,g);
}
```

13

- The method `drawAllObjects()` below first creates an array of `Drawable` interface references and sets them to refer to the `DrawableSquare` and `DrawableCircle` objects.
- Note that an explicit cast is needed as `s` is an array of `Shape` references (though they are referring to `Drawable` objects).

```
void drawAllObjects(Shape[] s, int n, Graphics g) {
    Drawable[] d = new Drawable[n];
    for(int i=0; i<n; i++)
        d[i] = (Drawable) s[i];
    for (int i=0; i<4; i++)
        d[i].draw(g);
}
```

14



How do we get user input in an applet ?

How do we get user input in an applet?

Using Buttons, TextFields, ...

How do I add these to my applet?

Override the `init()` method in the Applet derived class. Create instances of TextField, Button, ... and add them to the applet.

What happens when I press a button?

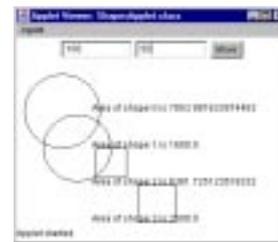
An event is generated and passed to the event handler for processing

Who can handle Button events ?

An Objects whose class that implements the interface ActionListener which has one method `actionPerformed()`. This method will process the event (take some action).

How do I register an event handler for button ?

Using the method `addActionListener()`. In this example the same object handles the events - hence - `moveButton.addActionListener(this);`



**Wait till
next
semester for
details**

An applet class which takes user input - x and y values

```
import java.applet.*;
import java.awt.event.*;
import java.awt.*;
public class ShapesApplet extends Applet implements ActionListener {
    TextField text1, text2;
    Button moveButton;
    int x, y;
    public void init() {
        text1 = new TextField(10);
        text2 = new TextField(10);
        moveButton = new Button("Move");
        moveButton.addActionListener(this);
        add(text1); add(text2); add(moveButton);
    }
    public void actionPerformed(ActionEvent e) {
        x = Integer.parseInt(text1.getText());
        y = Integer.parseInt(text2.getText());
        repaint();
    }
}
```

17

Arrays

```
int studentMarks[] = new int[256];
```

Creates a 256 element array of **int** (initialised to 0).

The size of an array can be specified as above, or at the declaration as follows:

```
int assignmentTotals = {6, 5, 8, 4, 5};
```

Note that **new** is not required. Defining and initialising an array can be done at declaration time only.

The array elements can be any type, both primitive and reference. For example, the following is legal:

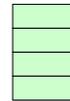
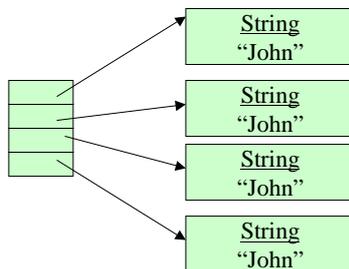
18

An array of references

```
String fabFour[] = new String[4];
```

But, of course, these declarations only allocate memory for the references. It is possible to do the declaration and instantiation at the same time:

```
String fabFour[] = {"John", "Paul", "George", "Ringo"};
```



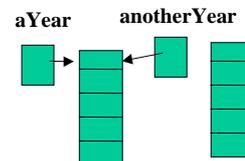
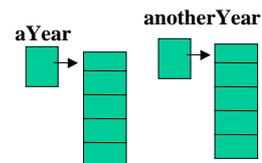
19

Assigning arrays

As arrays are reference objects, when assigning one to another the reference is copied (not the values).

```
int aYear[] = new int[365];  
int anotherYear[] = new int[365];  
  
// setting 2nd array to 1st  
anotherYear = aYear;
```

The last statement creates an alias rather than copying the values of **aYear** onto **anotherYear**.



If a copy is needed, use: `java.lang.System.arraycopy()`

20

Passing ints - What is the output ?

```
public class PassingInts{
    public static void main (String[] args){
        int n1 = 10;
        int n2 = 20;
        int n3 = 30;
        addOne(n1,n2,n3);
        System.out.println("Values are "+ n1 +
            ", "+n2+", "+n3);
    }
    public static void addOne(int x,int y,int z){
        x++;
        y++;
        z++;
    }
}
```

n1 n2 n3

10 20 30

x y z

21

Passing Array - What is the output ?

```
public class PassingIntArray{
    public static void main (String[] args){
        int n[] = {10,20,30};
        addOne(n);
        System.out.println("Values are "+
            n[0] + ", "+n[1]+", "+n[2]);
    }
    public static void addOne(int x[]) {
        x[0]++;
        x[1]++;
        x[2]++;
    }
}
```

n

10 20 30

22

Returning arrays

Since arrays are reference types, it is also possible to return arrays.

What is returned is a reference to the array, so we can store the reference in an appropriate array variable.

For example `readMarks()` method below sets up an array inside the method and returns the array to the `main()` method.

```
public static void main(String[] args) {
    int nums[];
    nums = readMarks(5);
    ...
}
int[] static readMarks(int numberOfMarks)
    throws IOException {
    ...
    int[] buffer = new int[numberOfMarks]; //
    ... // reading the values into the array
    return buffer;
}
```

nums

buffer

23

Multidimensional arrays

Often, a matrix-like data type is needed to solve a problem.

Two dimensional array is an array of arrays.

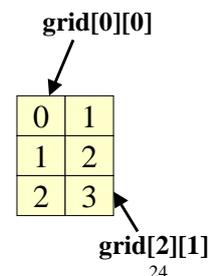
To describe an 8×8 grid we can declare `int grid[][]`; or even better `int[] grid[]`; or even `int[][] grid`;

To create an array with 3 rows and 2 columns use

```
int [][] grid = new int[3][2];
```

To set these elements you may use nested loops as in

```
for (int i=0; i<3; i++)
    for (int j=0; j<2; j++)
        grid[i][j] = i+j;
```



Quiz

Write the code to create the 2D arrays and to initialise them as shown using for loops

0	1	2
0	1	2
0	1	2

1	2	2
2	1	2
2	2	1