SHARE

Technology • Connections • Results

# 8359
# Object-oriented Programming with Java, Part 2

Stephen Pipes

IBM Hursley Park Labs, United Kingdom

Dallas 2003

# Intro to Java recap

- Classes are like user-defined types
- Objects are like variables of those types
- We send messages which invoke methods
- Classes have a class object

# Agenda

SHARE
Technology • Connections • Results

- <u>Inheritance and relationships</u>
- Abstraction and interfaces
- Polymorphism
- Overloading

# The Circle class

SHARE
Technology • Connections • Results

```
class Circle {

  // Data encapsulated by the class
  private SimplePoint center;
  private int radius;

  // Methods that form external interface
  public double circumference() { ... }
  public double area() { ... }
  public SimplePoint getCenter() { return center; }
  public int getRadius() { return radius; }
}
```

# The GraphicCircle class #1

```
class GraphicCircle {

  // Data encapsulated by the class
  private SimplePoint center;
  private int radius;

  // Methods that form external interface
  public double circumference() { ... }
  public double area() { ... }
  public SimplePoint getCenter() { return center; }
  public int getRadius() { return radius; }

  public void draw(Graphics g) { ... }
}
```

# GraphicCircle #1

- It's a cut-and-paste job
- Error prone
- Two copies of the "Circle" code
- Harder to maintain

# The GraphicCircle class #2



```
class GraphicCircle {

  // Data encapsulated by the class
  private Circle circle;

  // Methods that form external interface
  public double circumference() { ... }
  public double area() { ... }
  public SimplePoint getCenter() { return
circle.getCenter(); }
  public int getRadius() { return circle.getRadius(); }

  public void draw(Graphics g) { ... }
}
```

# GraphicCircle #2

- Uses existing code and logic for base circle
- But, needs lots of annoying wrapper methods!

# The GraphicCircle class #3

```
class GraphicCircle extends Circle {
  public void draw(Graphics g) { ... }
}
```

# GraphicCircle #3

- GraphicCircle is defined as an extension of the Circle class
- We call it a subclass
- GraphicCircle has all of the functionality of Circle, plus its own additional methods (and data)
- We say that GraphicCircle inherits the functionality of Circle
- We call the act of extending a class inheritance

# Inheritance

SHARE
Technology • Connections • Results

- GraphicCircle is a Circle
- You can use it anywhere a Circle is required
  - public aMethod(Circle c)
- You can treat it just like a Circle when you use it
  - graphicCircle.getRadius()
- Use inheritance when you have an "is a" relationship

# Other key relationships

SHARE
Technology • Connections • Results

- "is a" -> inheritance
- "has a" -> data member (e.g. Circle has a SimplePoint, its center) - containment
- "uses"  class A uses class B if:
  - a method of A sends a message to an object of class B
  - a method of A creates, receives or returns objects of class B
  - try to minimize the number of classes that use each other

# Agenda

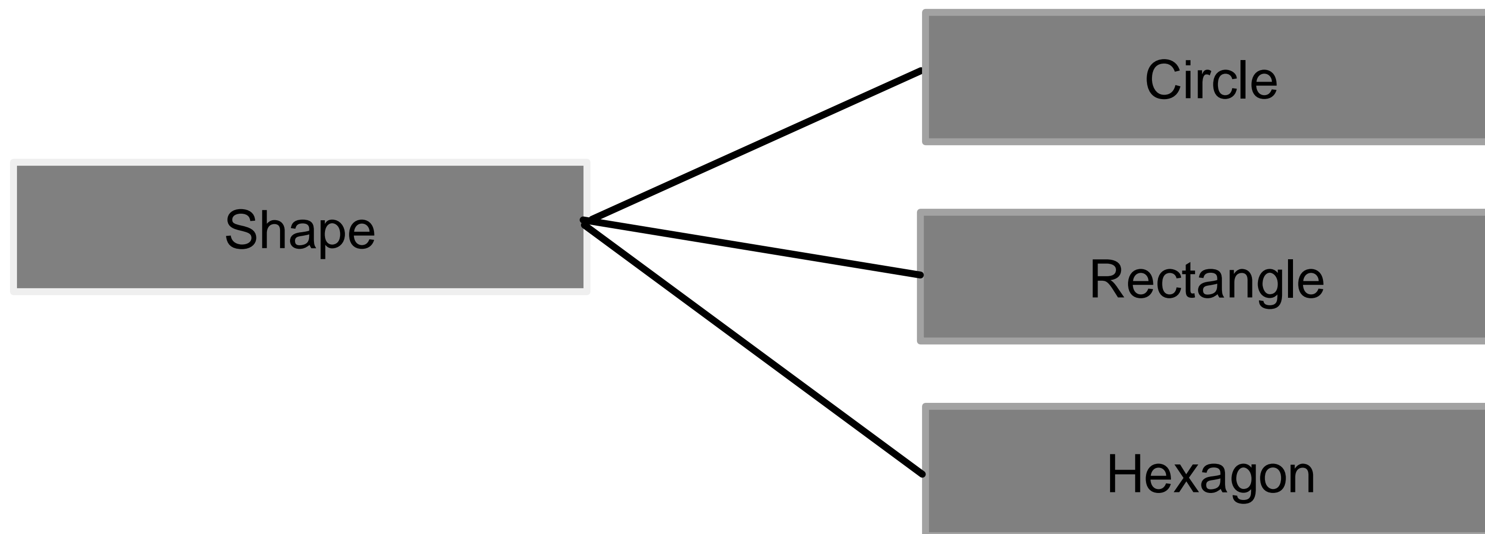
SHARE
Technology • Connections • Results

- Inheritance and relationships
- <u>Abstraction and interfaces</u>
- Polymorphism
- Overloading

# Abstraction and Interfaces

SHARE
Technology • Connections • Results

- We extend our super class to create more specific and useful sub classes




*general*
*super class*

*specifc*
*sub class*

# Abstraction and Interfaces



- The Circle class can calculate the area and circumference of a circle only

```
class Circle extends Shape {
    protected double r;
    protected static final double PI=3.14159265358979323846;

    public Circle() { r = 1.0; }
    public Circle(double r) { this.r = r; }

    public double area() { return PI * r * r; }
    public double circumference() { return 2 * PI * r; }
    public double getRadius() { return r; }
}
```

# Abstraction and Interfaces

SHARE
Technology • Connections • Results

- On the other hand, we want the Shape class (super-class) to encapsulate whatever features all our shapes have in common, such as area and circumference
- Since the Shape class is generic to all shapes, it cannot implement these features, so we use abstract methods as placeholders in our Shape class

# Abstraction and Interfaces

SHARE
Technology • Connections • Results

- Abstract classes do not implement functionality; note the semicolon immediately after the method definition. For this reason, they cannot be instantiated, since they contain no code

```
public abstract class Shape {
    public abstract double area();
    public abstract double circumference();
}
```

# Abstraction and Interfaces

- Abstraction allows us to group types of class and deal with them in a standard way
- We can group any classes that extend and implement the abstract Shape class

# Abstraction and Interfaces

- What is this code doing?

```
Shape[] shapes = new Shape[2]; // create an array of Shape
shapes[0] = new Circle(2.0);  // Circle is element 0
shapes[1] = new Rectangle(1.0, 2.0); // Rectangle is element 1

double total_area = 0;
for (int i = 0; i < shapes.length; i++) total_area += shapes[i].area();  //
compute the total area
```

# Abstraction and Interfaces

SHARE
Technology • Connections • Results

- Defines a Shape array and populates with different types of Shape. We do not need to cast the Circle and Rectangle classes onto the Shape array, since Circle and Rectangle inherit from Shape
- Calculates the total area of all Shapes in the array by calling the abstract methods of the Shape class. The abstract Shape class provides pointers to the correct methods in the appropriate sub-classes

# Abstraction and Interfaces

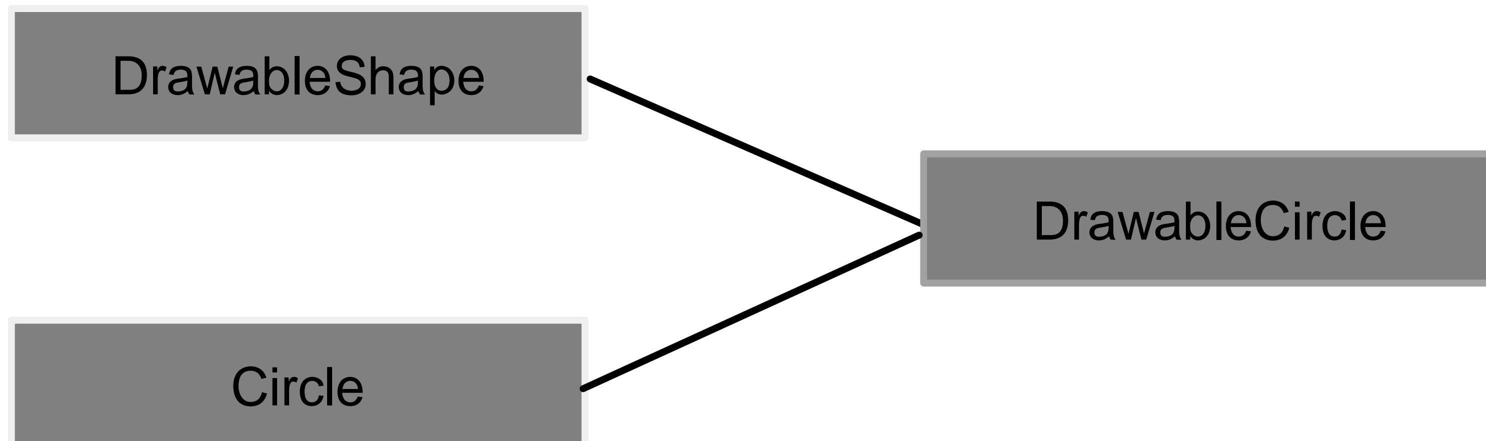
SHARE
Technology • Connections • Results

- We can draw our shapes by defining an abstract class called DrawableShape, which provides the abstract methods to draw the shape
- To draw a circle, we define DrawableCircle which provides the functionality to draw a circle, pointed to by the abstract DrawableShape class

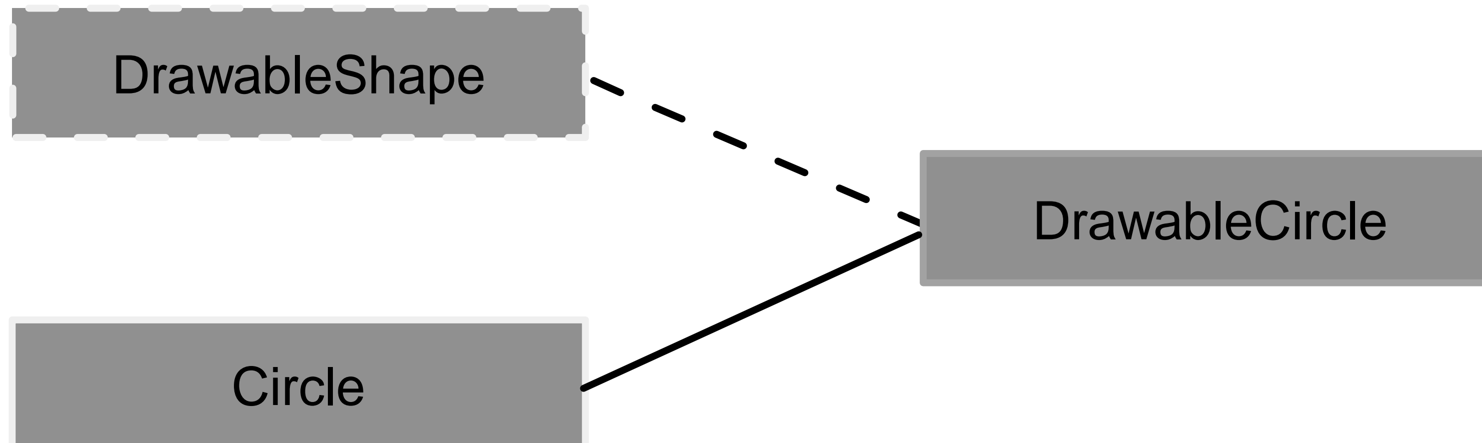
# Abstraction and Interfaces

SHARE
Technology • Connections • Results

- We need to calculate a circle before drawing it, so we extend our Circle class
- However, Java does not allow more than <u>one super class</u>, so we define DrawableShape as an interface



# Abstraction and Interfaces




```
public interface DrawableShape {
    public void setColor(Color c);
    public void setPosition(double x, double y);
    public void draw(DrawWindow dw);
}
```

# Abstraction and Interfaces



- Our DrawableCircle class will implement the DrawableShape interface

```
public class DrawableCircle extends Circle implements
DrawableShape {
    private Color c;
    private double x, y;

   // Provide a constructor that sets the super-class…
   public DrawableCircle(double r) { super(r); }
   // Now we implement those methods defined in DrawableShape…
   public void setColor(Color c) { this.c = c; }
   public void setPosition(double x, double y) { this.x = x; this.y = y; }
   public void draw(DrawWindow dw) { dw.drawCircle(x, y, r, c); }
}
```

# Abstraction and Interfaces

```
Shape[] shapes = new Shape[2]; // create an array of Shape
DrawableShape[] drawables = new DrawableShape[2];  // create an array of drawable shapes

// now we create some drawable shapes…
DrawableCircle dc = new DrawableCircle(1.1);
DrawableRectangle dr = new DrawableRectangle(…);

// since all drawable shapes extend Shape and implement DrawableShape, we can assign the
above drawable classes to both arrays…
shapes[0] = dc;
shapes[1] = dr;
drawables[0] = dc;
drawables[1] = dr;

double total_area = 0;
for (int i = 0; i < shapes.length; i++) {
    total_area += shapes[i].area();  // compute the total area
    drawables[i].setPosition(i*10.0, i*10.0);
    drawables[i].draw(draw_window);
}
```

# Abstraction and Interfaces



- Abstract classes may contain both abstract and implemented methods
- Interfaces contain only abstract methods. There can be no method code in a interface
- We may implement many interfaces in Java. This way, it is possible to implement a larger set of functionality into a single class.
- Constants may be defined in interfaces. These constants will be available to any class that implements the interface

# Abstraction and Interfaces

SHARE
Technology • Connections • Results

- Interfaces can have super-interfaces in the same way that classes can have super-classes
- The one difference is that interfaces can inherit from more than one super-interface
- Should a class implement such an interface, it must implement the abstract methods defined in the interface and all its super-interfaces
- java.lang.Runnable implemented by Thread

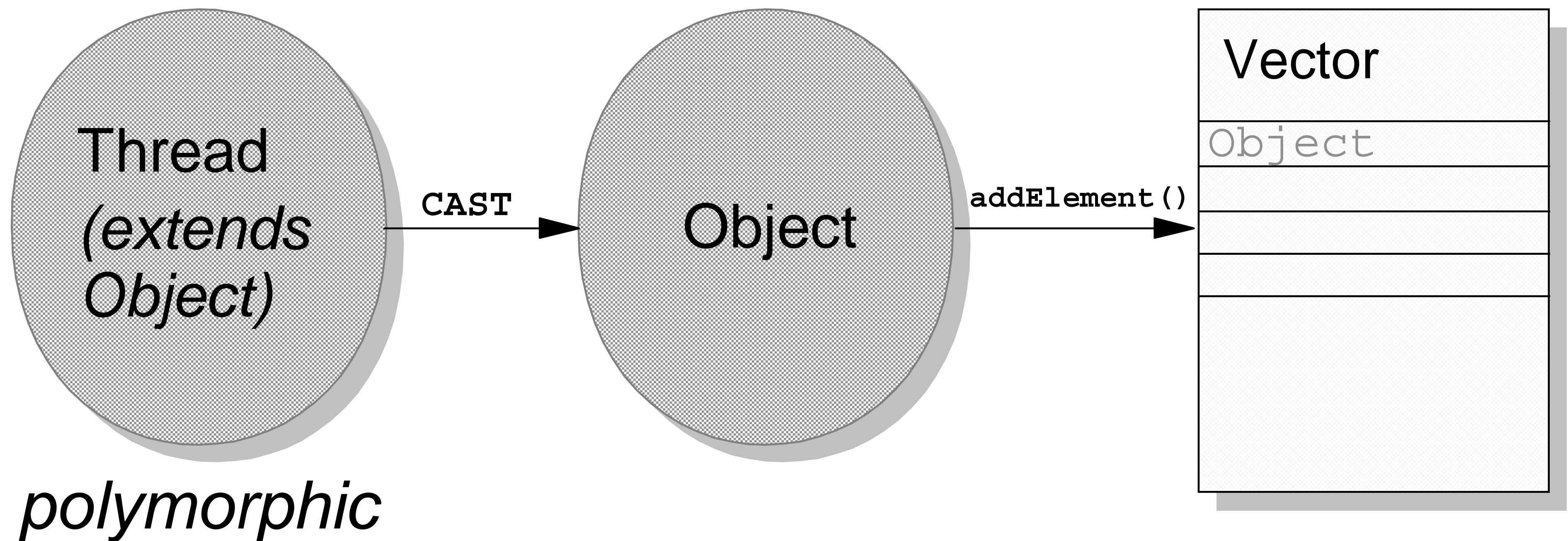
# Agenda

- Inheritance and relationships
- Abstraction and interfaces
- Polymorphism
- Overloading

# Polymorphism

SHARE
Technology • Connections • Results

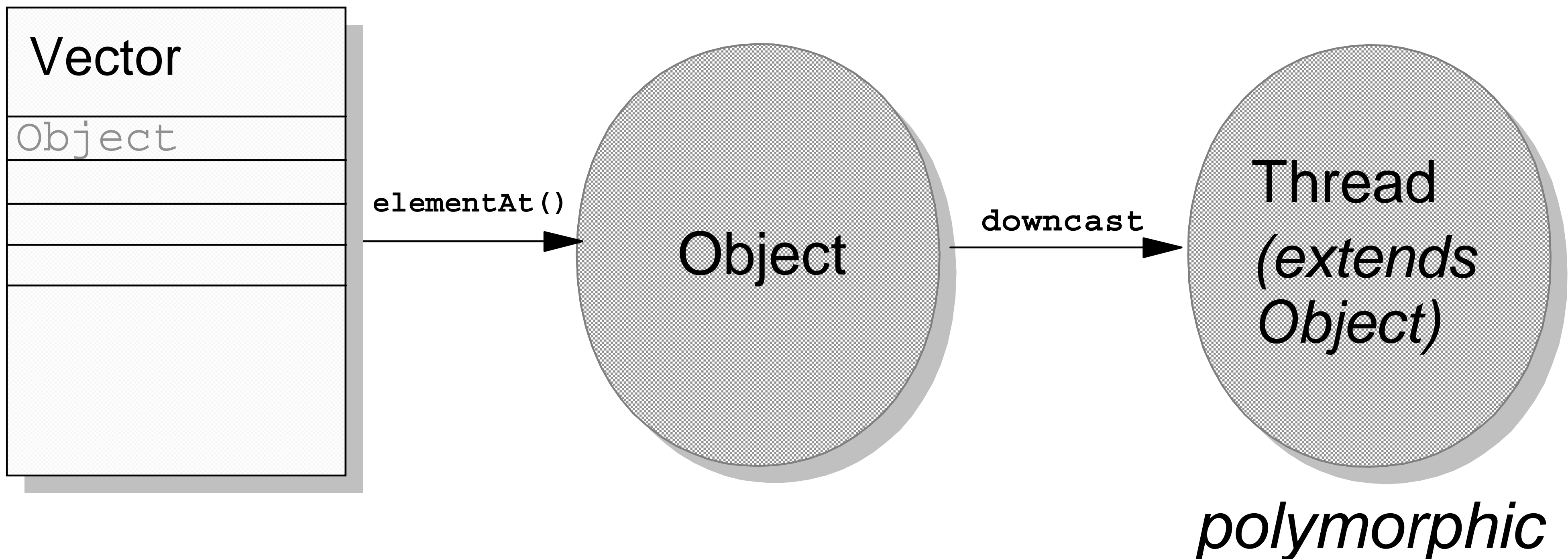
- Something that is polymorphic may appear in different guises. A polymorphic object may be <u>cast appropriately</u> to suit



# Polymorphism



- Down-casting casts an object to one of its descendents



# Polymorphism

- We can retrieve our object from the Vector using the elementAt method. It will be returned as type Object since it was cast to type Object before we passed it to the Vector
- If we call our object's toString method, it will be downcast to its original type Thread. Why is this?

# Agenda

SHARE
Technology • Connections • Results

- Inheritance and relationships
- Abstraction and interfaces
- Polymorphism
- <u>Overloading</u>

# Overloading

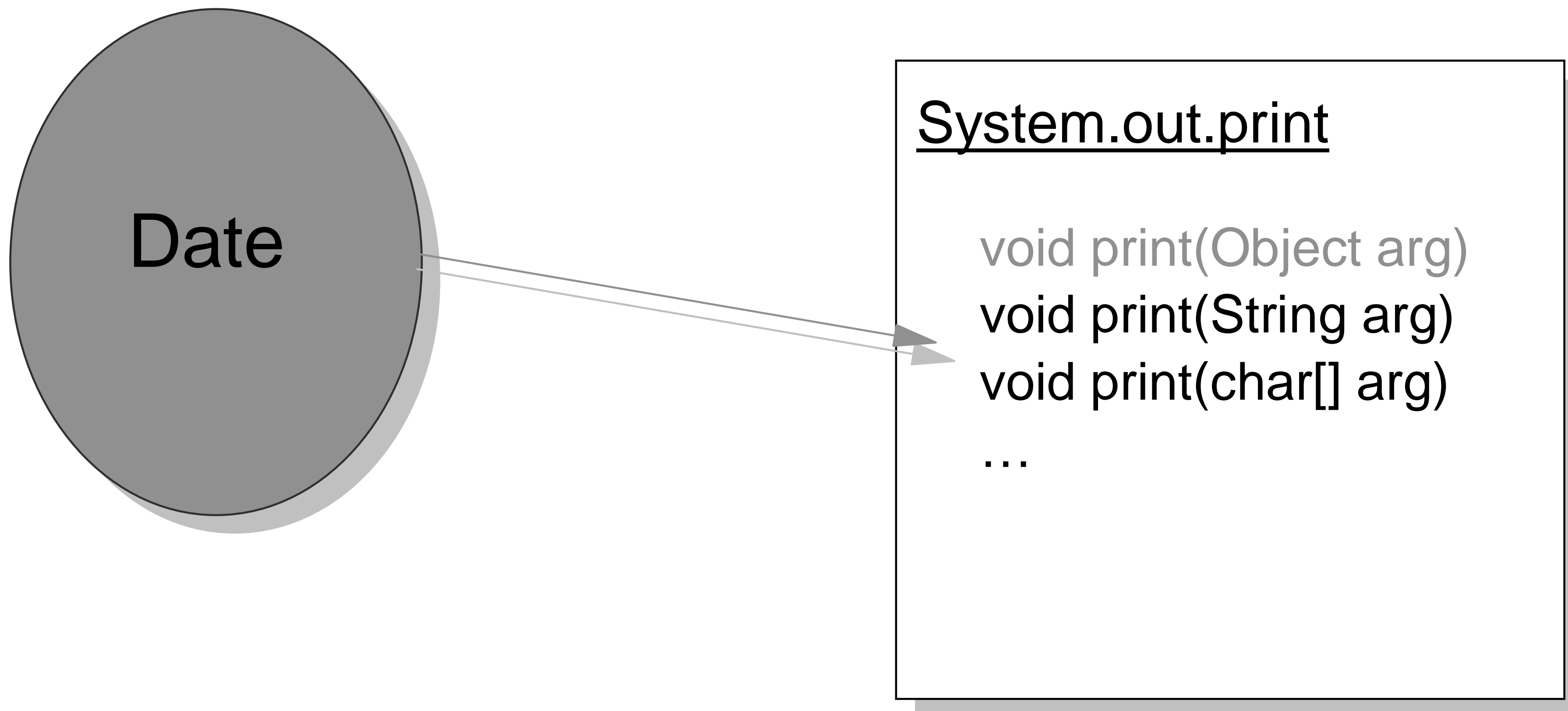
- Method overloading is a useful form of polymorphism, which allows objects to behave in exactly the same way irrespective of the information passed it
- In order to overload methods, we define more than one method of the same name in a class, but with different types or numbers of args.
- System.out.print method...

```
void print(Object arg) { … }
void print(String arg) { … }
void print(char[] arg) { … }
…
```

# Overloading

SHARE
Technology • Connections • Results

- If the print method is invoked with an argument of unknown type then the compiler will select the closest match



# Overloading

SHARE
Technology • Connections • Results

- A simple addition class. We may pass two or three integer arguments in to the Addition object. The same operation will be performed in either case.

```
class Addition {
    int calc(int a, int b) {
        return (a + b);
    }
    int calc(int a, int b, int c) {
        return (a + b + c);
    }
}
```

# Overloading



- Constructors may also be overloaded

```
public class Shape {
    public Shape(int xPos, int yPos) { … }
    public Shape(int xPos, int yPos, int width, int height) { … }
}
```

We may initialise this class in one of two ways.

```
new Shape(x,y);  // provide x and y positions
new Shape(x,y,w,h);  // provide x, y positions & width, height data
new Shape();  // default constructor  <--NO!
```

# What do we know?

SHARE
Technology • Connections • Results

- Abstract classes provide a common interface to all sub-classes. They cannot be instantiated
- Abstract methods provide no implementation but are placeholders for methods in subclasses
- Classes may have only one super-class. Interfaces allow us to extend the functionality of a class by providing abstract methods. Interfaces may have many super-interfaces
- Polymorphism allows down-casting and