

# Java interface

## Lecture 15

Waterford Institute of Technology

April 3, 2016

John Fitzgerald

# Presentation outline

Estimated duration presentation

Questions at end presentation

Topics discussed:

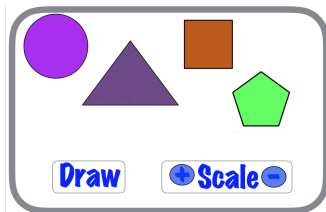
- Java 7 interface type
- Comparison with class
- Polymorphism
- Application in algorithm reuse

# Device interface

## Simple example

Consider interface design for toy shown

- Draw button: draw shapes.
- Scale button:
  - (+) increase size.
  - (-) decrease size.



# Device interface

## Outline implementations for toy example

### Circle

```
public class Circle {  
    private int radius;  
    public Circle (int radius) {  
        this.radius = radius;  
    }  
    public void draw () {  
        // Draw circle  
    }  
    public void scale(int factor) {  
        radius *= factor;  
    }  
}
```

### Triangle

```
public class Triangle {  
    private int height;  
    private int base;  
    public Triangle(int height, int base) {  
        this.height = height;  
        this.base = base;  
    }  
    public void draw () {  
        // Draw triangle  
    }  
    public void scale(int factor) {  
        base *= factor;  
        height *= factor;  
    }  
}
```

# Device interface

## Outline implementations for toy example

### Square

```
public class Square {  
    private int side;  
    public Square(int side) {  
        this.side = side;  
    }  
    public void draw () {  
        // Draw square  
    }  
    public void scale(int factor) {  
        side *= factor;  
    }  
}
```

### Pentagon

```
public class Pentagon {  
    private int radius;  
    public Pentagon(int radius) {  
        this.radius = radius;  
    }  
    public void draw () {  
        // Draw pentagon  
    }  
    public void scale(int factor) {  
        radius *= factor;  
    }  
}
```

# Device interface

## Outline implementations for toy example

### Instantiate shapes

```
Circle circle = new Circle (100);  
Triangle triangle = new Triangle (100, 100);  
Square square = new Square (100);  
Pentagon pentagon = new Pentagon (100);
```

### Draw Shapes

```
circle.draw();  
triangle.draw();  
square.draw();  
pentagon.draw();
```

# Device interface

Outline implementations for toy example using Java interface type

## Define an interface

```
public interface Drawable
{
    void draw();
    void scale(int factor);
}
```

Each shape implements Drawable, example:

```
public class Circle implements Drawable
{
    ...
    void draw();
    void scale(int factor);
}
```

## Create list Drawable shapes and draw them

```
ArrayList<Drawable> shapes = new ArrayList<>();
shapes.push(new Circle (100));
shapes.push(new Triangle (100, 100));
shapes.push(new Square (100));
shapes.push(new Pentagon (100));

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

# Device interface

Outline implementations for toy example - comparison different designs

## Instantiate shapes

```
Circle circle = new Circle (100);  
Triangle triangle = new Triangle (100, 100);  
Square square = new Square (100);  
Pentagon pentagon = new Pentagon (100);
```

## Draw Shapes

```
circle.draw();  
triangle.draw();  
square.draw();  
pentagon.draw();
```

## Create list Drawable shapes

```
ArrayList<Drawable> shapes;  
shapes = new ArrayList<>();  
  
shapes.push(new Circle (100));  
shapes.push(new Triangle (100, 100));  
shapes.push(new Square (100));  
shapes.push(new Pentagon (100));
```

## Draw shapes

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



# Java *interface*

## Description

*interface* is a Java type that may contain only

- Method signatures
- Constant declarations

Note that

- *interface* defines interfaces
- *class* defines classes
- Methods implemented in class that implements interface

```
public interface Drawable
{
    public void draw();
    public void scale(int x, int y);
}
```

access modifier **public** optional

# Java *interface*

Compare with *class*

Java *interface* different from *class*

- *interface* specifies behaviour only
- Cannot create objects of an *interface*
- Create objects of classes that implement interfaces

```
public class Tree implements Drawable
{
    public void draw() {
        ...
    }
    public void scale(int x, int y) {
        ...
    }
}
```

implementation here

# Java *interface*

## Implementation

A class may:

- Provide additional methods unrelated to interface
- Is obliged to implement all methods in interface
- May, optionally, provide `@Override` annotation to implemented methods

```
public class Triangle implements Drawable
{
    @Override
    public void draw() {...} //must implement draw
    @Override
    public void scale(int x, int y) {...} //must implement scale
    public int getArea(){...} //may include additional methods
}
```

# Java *interface*

## Implementation

Many classes may implement particular interface

- Class states that it implements particular interface

```
public class Triangle implements Drawable { ... }
```

- Class provides suitable implementation of interface methods

```
public class Triangle implements Drawable
{
    @Override
    public void draw() {...}
}
```

```
public class House implements Drawable
{
    @Override
    public void draw() {...}
}
```

# Java *interface*

## Converting to class

Object of class implementing interface may be stored in variable whose type is the interface

- Tree implements Drawable
- Tree object reference can be stored in Drawable variable
- Facilitates unifying behaviour

```
Drawable element = new Tree(...);
```

```
//create array of Drawable variables  
Drawable[] elements = new Drawable[2];  
//Assign different objects to elements in array  
Drawable elements[0] = new House(...);  
Drawable elements[1] = new Triangle(...);
```

# Java *interface*

## Working without Java *interfaces*

```
ArrayList<House> houses = new ArrayList<>();
```

```
houses.add(new House(100, 200));
```

```
houses.add(new House(150, 250));
```

```
for(House house : houses) {  
    house.draw();  
}
```

```
ArrayList<Tree> trees = new ArrayList<>();
```

```
trees.add(new Tree(100, 200, 400));
```

```
trees.add(new Tree(500, 150, 250));
```

```
for(Tree tree : trees) {  
    tree.draw();  
}
```

# Java *interface*

## Working with Java *interfaces*

```
ArrayList<Drawable> elements = new ArrayList<>();

elements.add(new House(100, 200));
elements.add(new House(150, 250));

elements.add(new Tree(100, 200, 400));
elements.add(new Tree(500, 150, 250));

for(Drawable element : elements) {
    element.draw();
}
```

House and Tree class must both implement Drawable interface.

# Java *interface*

## Polymorphism

*Polymorphism*: derived from Greek:

- *poly* : many
- *morph* : able to change form



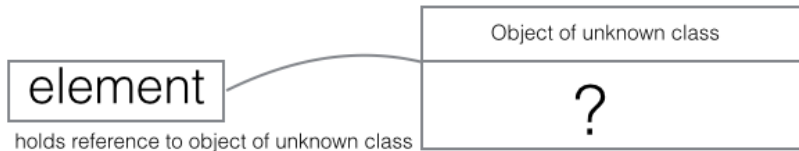


# Java *interface*

## Polymorphism

- Here *element* a reference to Drawable variable
- No way to know what class type referenced
- Only know object has method *draw()*

```
Array<Drawable> elements;//elements contains Houses, Trees, Triangles,...  
Drawable element = elements.get(i);//specific member of elements index i
```



# Java *interface*

## An example of polymorphism

*draw()* method can draw different shapes depending on how implemented in each class

- As for-each loop traverses elements in list
  - *element.draw()*; may call different methods
    - House draw method
    - Triangle draw method
    - Tree draw method
- Class whose *draw()* method invoked must implement *Drawable*

```
ArrayList<Drawable> elements;//elements contains Houses, Trees, Triangles,...  
  
for (Drawable element : elements)  
{  
    element.draw();  
}
```

# Java *interface*

## Polymorphism in action

### Facilitates system expansion

```
public class Circle implements Drawable {  
    ...  
    public void draw(){...}  
}
```

```
ArrayList<Drawable> elements = new ArrayList<>();  
elements.add(new House(100, 200));  
elements.add(new House(150, 250));  
elements.add(new Tree(100, 200, 400));  
elements.add(new Tree(500, 150, 250));  
/*add the circle object to existing list Drawable types*/  
elements.add(new Circle(200, 400, 150));  
  
for(Drawable element : elements) {  
    element.draw();  
}
```

# Java *interface*

## Class implementing multiple *interfaces*

Class may implement any number interfaces

- Each class must implement all interface methods

```
public interface Moveable {void moveTo(int x, int y);}
public interface Drawable {void draw();}

public class Circle implements Moveable, Drawable
{
    public void moveTo(int x, int y) {
        ...
    }
    public void draw() {
        ...
    }
}
```

# Java *interface*

What you may and may not do

```
public class Circle() implements Moveable, Drawable {  
    public void moveTo(...) {...}  
    public void draw() {...}  
    public scale(int x, int y) {...}  
}
```

```
Drawable drawable = new Circle();  
drawable.draw(); // ok  
drawable.scale(); // error: scale not in Drawable  
drawable.moveTo(); // error: moveTo not in Drawable
```

```
Moveable moveable = new Circle();  
moveable.draw(); // error: draw not in Moveable  
moveable.moveTo(); // ok
```

# Java *interface*

## Class implementing multiple *interfaces*

### *instanceof* test

- Drawable list references House, Tree and Circle objects
- These 3 classes implement Drawable
- Only Circle and Tree implement Moveable
- How to use existing ArrayList Drawable?

```
for(Drawable element : elements)
{
    if(element instanceof Moveable)
    {
        Moveable m = (Moveable)element; //Cast element to Moveable
        m.moveTo(10, 10);
    }
}
```

Cast Drawable to Moveable

# Java *interface*

## Class implementing multiple *interfaces*

### Casting

- *Moveable m = (Moveable)element;*
  - Casts the object to *Moveable* type
- *moveTo* cannot be invoked on *element*
  - *element* is *Drawable* so does not have *moveTo* method

```
for(Drawable element : elements)
{
    if(element instanceof Moveable)
    {
        ((Moveable)element).moveTo(10, 10);
    }
}
```

# Casting

## Verbose and compact

### Verbose

```
if(element instanceof Moveable)
{
    Moveable m = (Moveable)element;
    m.moveTo(10, 10);
}
```

### Compact

```
if(element instanceof Moveable)
{
    ((Moveable)element).moveTo(10, 10);
}
```



# Java *interface*

## Algorithm reuse

Algorithm: obtain maximum size rectangle in array Rectangle objects

```
{
    public static double maximum(Rectangle[] rects)
    {
        //Error check should be included to ensure array has values
        double max = rects[0].getArea();
        for(int i = 1; i < rects.length; i += 1)
        {
            if(rects[i].getArea() > max) {
                max = rects[i].getArea();
            }
        }
        return max;
    }
}
```

# Java *interface*

## Algorithm reuse

Algorithm: obtain maximum volume sphere in array Sphere objects

```
public static double maximum(Sphere[] spheres)
{
    //Error check should be included to ensure array has values
    double max = spheres[0].getArea();
    for(int i = 1; i < spheres.length; i += 1)
    {
        if(spheres[i].getArea() > max) {
            max = spheres[i].getArea();
        }
    }
    return max;
}
```

# Java *interface*

## Algorithm reuse

We may require such algorithms for several types  
Here's how to use *interfaces* to unify behaviour:

- Create a Measurable interface
- Refactor Rectangle and Sphere as follows
  - Have classes implement Measurable interface
  - Implement the *getMeasure()* methods in each class
- Develop Data class to
  - Traverse array Measureable objects
  - Discover object generating maximum value
- Develop a TestData class to test the system

```
//Create Measurable interface  
public interface Measurable  
{  
    double getMeasure();  
}
```

# Java *interface*

## Algorithm reuse

### Refactored Rectangle class implements Measurable

```
class Rectangle implements Measurable
{
    private double length;
    private double width;
    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }
    /**
     * @return returns area rectangle
     */
    @Override
    public double getMeasure() {
        return length*width;
    }
}
```

# Java *interface*

## Algorithm reuse

### Refactored Sphere class implements Measurable

```
class Sphere implements Measurable
{
    private double radius;
    public Sphere(double radius) {
        this.radius = radius;
    }
    /**
     * @return returns volume sphere
     */
    @Override
    public double getMeasure() {
        return 4*Math.PI*radius*radius*radius/3;
    }
}
```

# Java *interface*

## Algorithm reuse

Data class to calculate maximum measured quantity

```
public class Data
{
    public static Measurable maximum(Measurable[] objects)
    {
        if (objects.length == 0) { return null;}
        Measurable max = objects[0];
        for (int i = 1; i < objects.length; i += 1)
        {
            if(objects[i].getMeasure() > max.getMeasure())
            {
                max = objects[i];
            }
        }
        return max;
    }
}
```

# Java *interface*

## Algorithm reuse

TestData class to demo system

```
public class TestData
{
    public void testData() {
        Sphere[] spheres = {
            new Sphere(100),
            new Sphere(200),
            new Sphere(250),
            new Sphere(300)
        };
        Measurable largest = Data.maximum(spheres);
        System.out.println("Largest: "+largest);
    }
}
```

# Java *interface*

## Polymorphism in action

### *toString* implementations Rectangle and Sphere

```
@Override
public String toString()
{
    return "Sphere [radius=" + radius + " volume=" + getMeasure() + "];"
}
```

```
@Override
public String toString()
{
    return "Rectangle
    [length =" + length + "width =" + width + " area=" + getMeasure() + "];"
}
```



# Summary

- Java interface an abstract class type
- Instantiation disallowed
- Instance of class (object) assignable to interface variable
- Facilitates unifying common behaviour
- Facilitates algorithm reuse