Java interface

Waterford Institute of Technology

May 11, 2014

John Fitzgerald

Java Operator

Ternary

Conditional operator ?:

- Also known as *ternary* operator
- Can be thought of as *if-then-else* operator
- If condition true assign value1 else value2

```
int value1 = 1;
int value2 = 2;
int result;
boolean someCondition = true;
result = someCondition ? value1 : value2;
```

Java *interface*

Description

interface is a Java type that may contain only

- Method signatures
- Constant delarations

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();
}
```

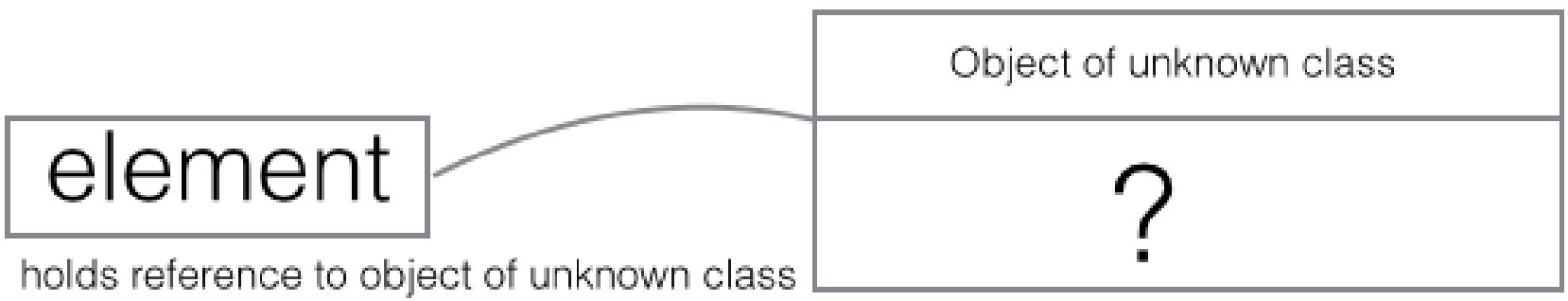
House and Tree class must both implement Drawable interface.

Java *interface*

Polymorphism

- 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
    - Methods yet to be added to application
- Class whose *draw()* method invoked must implement *Drawable*

```
ArrayList<Drawable> elements;//elements contains Houses, Trees, Triangles,...

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

Java *interface*

Importance of polymorphism

Term *polymorphism* used generally in Java where

- Method invoked depends on invoking object
- *object.method();*

Why important?

- Allows building of expandable systems
- New types can be added without changing program logic
- Example
  - Create new class, *Circle implements Drawable*
  - Add new Circle object to ArrayList of existing Drawable elements

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*

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 *interface*s 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()+"]";
}
```