Object Oriented Programming

Interfaces and Polymorphism

Dr. Seán Russell
sean.russell@ucd.ie

School of Computer Science,
University College Dublin

September XX, 2019

Table of Contents

Learning outcomes

After this lecture and the related practical students should...

- be able to declare and use interface types
- understand the concept of polymorphism
- be able to convert between interface types and classes

Table of Contents

Cost Of Programming

- Developing software is expensive
- Reusing software components makes it cheaper
- Changes required to make reuse possible

Interfaces

- **Interfaces** can be used for software reuse
- An interface is a list of the actions an object can do
- An interface is a list of all methods that an object **must** have in order to behave like an "X"
- E.g. anything that "behaves like" a light, should have the methods `turnOn` and `turnOff`

Table of Contents

Common Code

- Focus on the important operations
- Describe operations with interface
- For an example we will design a class to calculate the average and maximum of a set of numbers
- This class would require the following methods
  - A method to add new numbers to the set
  - A method to get the average value
  - A method to get the maximum value

```java
public class DataSet {
   private double sum;
   private double max;
   private int count;
   public void add(double x) {
      sum += x;
      if (count == 0 || x > max) {
         max = x;
      }
      count++;
   }
   public double getAverage() {
      if (count == 0) {
         return 0;
      } else {
         return sum / count;
      }
   }
   public double getMaximum() {
      return max;
   }
}
```

Common Code Problems

- This class works for a set of doubles
- But it cannot be used for anything else
- Changing the code so it works for bank accounts or coins means it will no longer work for doubles

```
1  public class DataSet {
2     private double sum;
3     private BankAccount max;
4     private int count;
5     public void add(BankAccount x){
6        sum += x.getBalance();
7        if(count == 0 || x.getAmount()>max.getAmount()){
8           max = x;
9        }
10       count++;
11    }
12    public double getAverage() {
13       if (count == 0) {
14          return 0;
15       } else {
16          return sum / count;
17       }
18    }
19    public BankAccount getMaximum(){
20       return max;
21    }
22 }
```

```
1  public class DataSet {
2     private double sum;
3     private Coin max;
4     private int count;
5     public void add(Coin x) {
6        sum += x.getValue();
7        if(count == 0 || x.getValue()>max.getValue()) {
8           max = x;
9        }
10       count++;
11    }
12    public double getAverage() {
13       if (count == 0) {
14          return 0;
15       } else {
16          return sum / count;
17       }
18    }
19    public Coin getMaximum() {
20       return max;
21    }
22 }
```

Common Algorithm

- Types changed
- But the structure is the same
- Only the **measurements** are different
  - For doubles we used the value of the parameter
  - For bank accounts we used the method `getBalance`
  - For coins used the method `getValue`

Common Algorithm

- Define code that works for any object that can be measured
- Everyone agrees on a single method to use
- The method `getMeasure` will be used
- We can use the same algorithm

```
1 sum += x.getMeasure();
2 if (count == 0 || x.getMeasure() > max) {
3   max = x;
4 }
5 count++;
```

What type?

The code shows a variable named `x` that we call the method `getMeasure` on

- But what is the type of the variable `x`?
- Ideally, the type would be any class that contains the method `getMeasure`

Interfaces

- We can use **Interfaces** to specify required operations

```
1 public interface Measurable {
2   double getMeasure();
3 }
```

Interface

Interface Syntax

```
1 public interface InterfaceName {
2     method signatures
3 }
```

Example

```
1 public interface Bicycle {
2     void changeGear(int newValue);
3     void speedUp(int increment);
4     void applyBrakes(int decrement);
5 }
```

Differences between Interface and Class

- All methods in an interface are **abstract**
  - No implementation
- All methods are `public`

- No **instance variables**
  - You cannot create an instance of an interface
- Using the Measurable interface we can rewrite the `DataSet` class

```
1  public class DataSet {
2     private double sum;
3     private Measurable max;
4     private int count;
5     public void add(Measurable x) {
6        sum += x.getMeasure();
7        if(count==0||x.getMeasure()>max.getMeasure()){
8           max = x;
9        }
10       count++;
11    }
12    public double getAverage() {
13       if (count == 0) {
14          return 0;
15       } else {
16          return sum / count;
17       }
18    }
19    public Measurable getMaximum() {
20       return max;
21    }
22 }
```

DataSet Class Using Interface

Works for any class that **implements** the `Measurable` interface

Important Code

- The declaration of the `max` variable

```
3   private Measurable max;
```

- The parameter and functionality of the add method:

```
5   public void add(Measurable x) {
6     sum += x.getMeasure();
7
     if(count==0||x.getMeasure()>max.getMeasure()){
```

- The return value of the `getMaximum` method:

```
19  public Measurable getMaximum() {
```

Table of Contents

Implementing an Interface

- The parameter of the add method is Measurable
- How does Java know if an object "behaves like" measurable?
- Class must have the method `getMeasure`
- Class must define relationship

Syntax of implementing an Interface

```
1 public class ClassName implements InterfaceName {
```

Example

```
1 public class BankAccount implements Measurable {
```

Implementing an Interface is a Contract

- Implementing an interface is a contract
- We must provide an implementation for each method in the interface
- Or the code **will not compile**
- If `BankAccount` defines that it "behaves like" `Measurable` then it **must** have a method named `getMeasure`
- The signature must match

Implementing the Methods

- When implementing an interface, all methods must be implemented
  - The methods must be **public**
  - The return type, name and parameters must match exactly

```
1 public class Coin implements Measurable{
2   ...
3   public double getMeasure(){
4     return value;
5   }
```

Table of Contents

Implementing multiple interfaces

- An object can "behave like" many things
- A bank account can "behave like" something measurable and also taxable
- A ship can "behave like" something that can be drawn on the screen, and something that can move and something that can be destroyed
- To do this we define a **list** of interfaces separated by commas

Implementing multiple interfaces

Syntax

```
1 public class ClassName implements IfName1,
      IfName2,... {
2     ...
3 }
```

Example

```
1 public class BankAccount implements
      Measurable, Taxable {
2     ...
3 }
```

Common Code expressed as Interface

- The `Measurable` interface shows what measurable objects have in **common**
- This makes the `DataSet` class more useful
- This is normally how interfaces are used in programming
  - I have often used interfaces defined by people I have never met

Interfaces Example

- A good example of the reuse of code is the `Collections` class
- `Collections` contains many methods, such as:
  `public static void sort(List l)`
- `List` is an interface
- This method only needed to be written once, and can be used for many classes

Table of Contents

Interfaces As Types

- No instances of interfaces
- How can the a `Measurable` object be a parameter?
- Only objects that implements the `Measurable` interface
- Don't know type of object
- What methods can we call?
- We can **only** use methods in the `Measurable` interface

Converting objects to Interface Types

```
5    public void add(Measurable x) {
```

- If parameter is a `BankAccount`, it is converted to `Measurable`
- We can manually convert a `BankAccount` object to `Measurable`

```
1 BankAccount ac = new BankAccount(1234);
2 Measurable meas = ac;
```

Converting objects to Interface Types

- After conversion only use some methods

```
1 BankAccount ac = new BankAccount(1234);
2 Measurable meas = ac;
3
4 double money1 = meas.getMeasure();
5 double money2 = meas.getBalance();
```

- `getBalance` in the `BankAccount` class can only be called using a `BankAccount` object
- Here `meas.getBalance()` does not compile

Converting objects to Interface Types

- To convert an object, the class must implement the interface

Example

```
1 Measureable m = new BankAccount(1000);
2 Measureable m2 = new String();
```

- The first line will compile
- The second line will not compile

Converting Interface Types to objects

- We can also convert in the opposite direction, converting an object that is stored as an interface type into a class type
  - If the type is not correct we will crash our program
  - This conversion is called **type casting**

Type casting

If we have an object that has a type, we can convert the type of the variable

```
1 Measureable m = dataset.getMaximum();
2 BankAccount b = (BankAccount) m;
```

Type casting

- Type casting can be used in a lot of circumstances
- It changes the type that Java thinks an **object reference** points to
- The object is never changed

Syntax

The basic syntax is `(newType) variableName`

E.g. `newType newVar = (newType)oldVar;`

- There are now **two** references to the same object

Safer Type Casting

- What if I typecast a `String` to a `BankAccount`?
- We can check it is **safe** to typecast
- `instanceof` lets us check if a typecast is allowed

instanceof

If we X and we want to see if it is the type Y, we can use the following if statement

```
if(X instanceof Y){
   Y a = (Y)X;
}
```

Instanceof

- The instanceof operator is a boolean operator
- It returns true if the object on the left is the type on the right

Example

```
1 Measurable m = new BankAccount(1000);
2 ...
3 BankAccount b;
4 if(m instanceof BankAccount){
5     b = (BankAccount)m;
6 }
```

- Works for interfaces and classes

Table of Contents

Shape interface

- An interface defining common code about different shapes
- Shapes can be moved, have an area and an outline

Shape Interface

```
1 public interface Shape {
2   void move(int x, int y);
3   double getArea();
4   double getOutline();
5 }
```

```
1  public class Square implements Shape {
2     int x, y, width;
3     public Square(int x1, int y1, int w) {
4        x = x1; y = y1;
5        width = w;
6     }
7     public void move(int x1, int y1) {
8        x += x1;
9        y += y1;
10    }
11    public double getArea() {
12       return width * width;
13    }
14    public double getOutline() { return 4 *
       width; }
15 }
```

```
1  public class Circle implements Shape {
2    private int x, y, radius;
3    public Circle(int x1, int y1, int r) {
4      x = x1;  y = y1;
5      radius = r;
6    }
7    public void move(int x1, int y1) {
8      x += x1; y += y1;
9    }
10   public double getArea() {
11     return Math.PI * radius * radius;
12   }
13   public double getOutline() { return 2 *
      Math.PI * radius; }
14   public void grow(int r) { radius += r; }
15 }
```

Polymorphism

- Classes can implement the methods differently
  - `Square` and `Circle` have different implementations for `getArea` and `getOutline`
- Which version will be executed?
  - There are no `Shape` objects
  - The object is either `Square`, `Circle` or some other class
  - The JVM chooses the correct method

Dynamic lookup

- Choosing the correct method is called **dynamic method lookup**
- Dynamic method lookup allows a programming technique called **polymorphism**

Polymorphism

In programming this means that we can store object references using an interface type, and when executing methods each object is treated the same but different objects may execute different versions of the same method

Polymorphism Example

Using Polymorphism

```
1 public static void main(String[] args) {
2   Shape s1 = new Square(1, 1, 5);
3   Shape s2 = new Circle(2, 2, 5);
4
5   double a1 = s1.getArea();
6   double a2 = s2.getArea();
7 }
```

- When the `getArea` method is called on `s1`, it uses the code defined in the `Square` class
- When the `getArea` method is called on `s2`, it uses the code defined in the `Circle` class

Table of Contents

Updated Interfaces

- Interfaces have been updated
- Static methods
- Default methods

Table of Contents

Static Methods

- Can put some utility code in interfaces
- No Instance variables
- No non-final static variables
- Similar to static methods in classes

```
1  public interface Shape {
2    void move(int x, int y);
3    double getArea();
4    double getOutline();
5    static Shape getLargest(Shape s1, Shape
      s2) {
6      if (s1.getArea() > s2.getArea()) {
7        return s1;
8      } else {
9        return s2;
10     }
11   }
12 }
```

Static Methods

- Compares two parameters
- Can only call methods

```
1  public static void main(String[] args) {
2     Shape[] shapes = new Shape[3];
3     shapes[0] = new Circle(5,5,5);
4     shapes[1] = new Square(5,5,5);
5     shapes[2] = new Rectangle(5,5,5,10);
6
7     Shape temp = Shape.getLargest(shapes[0],
       shapes[1]);
8     Shape largest = Shape.getLargest(temp,
       shapes[2]);
9     System.out.println("Largest shape has area
       of " +largest.getArea());
10 }
```

Table of Contents

Default Methods

- Similar to defining methods for classes
- No instance variables
- Can only call methods

```java
public interface Shape {
  void move(int x, int y);
  double getArea();
  double getOutline();
  static Shape getLargest(Shape s1, Shape
   s2) {...}
  default void printArea() {
    System.out.println("My area is " +
   getArea());
  }
}
```

Default Methods

- The methods only prints out a message
- This can be called on any object that implements Shape

```
1 public static void main(String[] args) {
2    Circle c = new Circle(5, 5, 5);
3    Square s = new Square(5, 5, 5);
4    Rectangle r = new Rectangle(5, 5, 5, 10);
5    c.printArea();
6    s.printArea();
7    r.printArea();
8 }
```