Chapter 11 – Interfaces and Polymorphism

Chapter Goals
- Learn about interfaces
- Convert between class and interface references
- Understand the concept of polymorphism
- Understand the purpose of interfaces to decouple classes

Interfaces and Polymorphism
- Interfaces are important for developing reusable software components
- Polymorphism is the principal at the heart of this process – a key component of object oriented programming

Concept #1: Interfaces
- If you think of a class as a sphere – the interface is the surface
  - How will code behave (what methods will it define)
  - Ex. Javadocs are an interface to the underlying class – defines how the class behaves

Interfaces
- Java uses interfaces to define a common set of behaviors that varying objects can share
  - Define an interface that only specifies methods that must be used (not how to use them)
  - Create a class that implements this interface – it is signing a contract that it will define all of the methods the interface specifies
  - This contract insures that we can make assumptions about what methods are available (without looking at a Javadoc)

Interfaces
- An interface is much like a class in terms of how it is defined except
  - Not instantiable
  - No fields (except constants)
  - No body to the methods, only signatures
  - Methods are automatically public
11.1 Using Interfaces for Code Reuse

- Use interface types to make code more general
- Identify common/essential operations

Let's say there is a class `DataSet` that keeps track of a running total of real numbers.

```java
public class DataSet{
    private double sum, maximum;
    private int count;
    
    public void add(double x){
        sum = sum + x;
        if (count == 0 || maximum < x)
            maximum = x;
        count++;
    }
    public double getMaximum(){
        return maximum;
    }
    public int getCount(){
        return count;
    }
}
```

Problem: Only works for numbers
- What if we wanted to keep track of BankAccounts?

```java
public class DataSet{
    private double sum;
    private BankAccount maximum;
    private int count;
    
    public void add(BankAccount x) {
        sum = sum + x.getBalance();
        if (count == 0 || maximum.getBalance() < x.getBalance())
            maximum = x;
        count++;
    }
    public BankAccount getMaximum(){
        return maximum;
    }
    public int getCount(){
        return count;
    }
}
```
What if we want to do the same for Coins?

The mechanics of analyzing the data is the same in all cases; details of measurement differ. We have three classes doing three very similar tasks, but they all contain redundant code. Classes could agree on a method `getMeasure()` that obtains the measure to be used in the analysis.

We can then implement a single reusable `DataSet` class whose add method looks like this:

```java
public class DataSet{
    private double sum;
    private Coin maximum;
    private int count;

    public void add(Coin x) {
        sum = sum + x.getValue();
        if (count == 0 || maximum.Value() < x.getValue())
            maximum = x;
        count++;
    }

    public Coin getMaximum(){
        return maximum;
    }

    public int getCount(){
        return count;
    }
}
```

Interfaces

What type is `x`? We want `x` to be an type of object that has a `getMeasure()` method. Interfaces allow us to ensure that this is the case.

An interface type is used to specify required operations for a class.

```java
public interface Measurable {
    double getMeasure();
}
```
public class DataSet{
    private double sum;
    private double sum;
    private
    private
    Measurable
    Measurable
    maximum;
    maximum;
    private int count;
    private int count;
    public void
    public void
    add(Measurable x) {
        sum = sum + x.getMeasure();
        if (count == 0
            || maximum.
            || maximum.
                getMeasure()
                getMeasure()
                < x.
                < x.
                    getMeasure())
                    < x.
                    getMeasure())
                        maximum = x;
                        maximum = x;
        count++;
    }
    public Measurable getMaximum(){
        return maximum;
    }
    public int getCount(){
        return count;
    }
}

Interfaces
■ Now DataSet can be used for any class that implements the Measurable interface
■ To implement an interface, use implements reserved word and implement all methods specified in the interface

Defining interfaces
public interface InterfaceName
{
    // method signatures
}

Implements
public class ClassName implements Measurable
{
    public double getMeasure()
    {
        Implementation
    }
    // Additional methods and fields
}
■ Note that interface names often end in –able
■ Describe an “ability” of the class
■ Comparable, Readable, Appendable

Implementing interfaces
public class ClassName implements
    InterfaceName, InterfaceName, ...
{
    // methods
    // instance variables
}
■ Can implement multiple interfaces

public class BankAccount implements Measurable
{
    public double getMeasure()
    {
        return balance;
    }
    // Additional methods and fields
}
11.2 Converting between classes and interfaces

- You can convert from a class type to an interface type, provided the class implements the interface.

```java
BankAccount account = new BankAccount(10000);
Measurable x = account; // OK
Coin dime = new Coin(0.1, "dime");
Measurable x = dime; // Also OK
```

- Cannot convert between unrelated types
  ```java
  Measurable x = new Rectangle(5, 10, 20, 30); // ERROR
  ```
  Because `Rectangle` doesn't implement `Measurable`.

We know that since `Coin` implements `Measurable`, we can assign a `Coin` object to a `Measurable` reference variable.

- But what about the other way? Could we always assign a `Measurable` object to a `Coin` reference variable?

You need a cast to convert from an interface type to a class type.

You know it's a coin, but the compiler doesn't. Apply a cast:

```java
Coin maxCoin = (Coin) max;
String name = maxCoin.getName();
```

- If you are wrong and `max` isn't a coin, the compiler throws an exception.

Compare to casting numbers:

- When casting number types you agree to the information loss.
- When casting object types you agree to the risk of causing an exception.

Type casting

- Add coin objects to `DataSet`
  ```java
  DataSet coinData = new DataSet();
  coinData.add(new Coin(0.25, "quarter"));
  coinData.add(new Coin(0.1, "dime"));
  . . .
  Measurable max = coinData.getMaximum();
  ```

- What can you do with it? It's not of type `Coin`
  ```java
  String name = max.getName(); // ERROR
  ```
### 11.3 Polymorphism

- Interface variable holds reference to object of a class that implements the interface
  
  ```java
  Measurable x;
  x = new BankAccount(10000);
  x = new Coin(0.1, "dime");
  ```

- Note that the object to which x refers doesn't have type Measurable;
  - The type of the object is some class that implements the Measurable interface

#### Purpose of Polymorphism

- You can call any of the interface methods:
  ```java
  double m = x.getMeasure();
  ```

- Which method is called?
  - Depends on the actual object.
    - If x refers to a bank account, calls `BankAccount.getMeasure()`
    - If x refers to a coin, calls `Coin.getMeasure()`

### Polymorphism

- Polymorphism (many shapes): Behavior can vary depending on the actual type of an object
  - The property the we can call `x.getMeasure()` with multiple contexts is an instance of polymorphism

- Called *late binding*: resolved at runtime

- Different from overloading; overloading is resolved by the compiler (*early binding*)