Interfaces and polymorphism

Chapter 9

Interfaces

- Used to express operations common to more than one purpose.
- Example:
  - You want to find the maximum gpa of a group of students.
  - You want to find the maximum balance of the bank accounts of a bank.
  - You use the same operation to find the maximum.
  - With what we know, we would have to rewrite the method for each.
  - There needs to be a better way.

Using Interfaces for Code Reuse

- Interface types makes code more reusable
- Interface type declares a set of methods and their signatures.
- An interface type is similar to a class
- Differences
  - All method in an interface type are abstract
    - Name
    - Parameter
    - Return type
    - **Don’t’ have an implementation**
  - All methods are automatically public
  - **Does not have instance fields**

Using Interfaces for Code Reuse

- In Chap. 6, we created a `DataSet` to find the average and maximum of a set of values (*numbers*)
- What if we want to find the average and maximum of a set of `BankAccount` values?

Using Interfaces for Code Reuse

```
public class DataSet // Modified for BankAccount objects
{
   . . .
   public void add(BankAccount x)
   {
      sum = sum + x.getBalance();
      if (count == 0 || maximum.getBalance() < x.getBalance())
         maximum = x;
      count++;
   }

   public BankAccount getMaximum()
   {
      return maximum;
   }
   private double sum;
   private BankAccount maximum;
   private int count;
}
```

Using Interfaces for Code Reuse

- Or suppose we wanted to find the coin with the highest value among a set of coins. We would need to modify the `DataSet` class again

Using Interfaces for Code Reuse

```
public class DataSet // Modified for Coin objects
{
   . . .
   public void add(Coin x)
   {
      sum = sum + x.getValue();
      if (count == 0 || maximum.getValue() < x.getValue())
         maximum = x;
      count++;
   }

   public Coin getMaximum()
   {
      return maximum;
   }
   private double sum;
   private Coin maximum;
   private int count;
}
```

Using Interfaces for Code Reuse

- The mechanics of analyzing the data is the same in all cases; details of measurement differ
- Classes could agree on a method `getMeasure` that obtains the measure(or the value) to be used in the analysis

Using Interfaces for Code Reuse

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

```
sum = sum + x.getMeasure();
if (count == 0 || maximum.getMeasure() < x.getMeasure())
   maximum = x;
count++;
```

- In this case x can be either a bank account or it can be a coin or a gpa.
- We need an interface.
  - We will call it Measureable
  - It will declare one method (getMeasure)

Example

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

Notice:

- Type is interface
- No instance fields
- No implementation

Use

- When we do this we can use the DataSet class for any class that *implements* the Measurable interface

Using Interfaces for Code Reuse

- What is the type of the variable x?
  `x` should refer to any class that has a `getMeasure` method

```
sum = sum + x.getMeasure();
if (count == 0 || maximum.getMeasure() < x.getMeasure())
   maximum = x;
count++;
```

Using Interfaces for Code Reuse

- An *interface type* is used to specify required operations

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

- When we use the interface, our class must have a method or methods that correspond to each method declared in the interface.
- Interface declaration lists all methods (and their signatures) that the interface type requires

How to *Implement*

- **Use implements keyword to indicate that a class implements an interface type**

```
public class BankAccount implements Measurable
{
   public double getMeasure()
   {
      return balance;
   }
   // Additional methods and fields
}
```

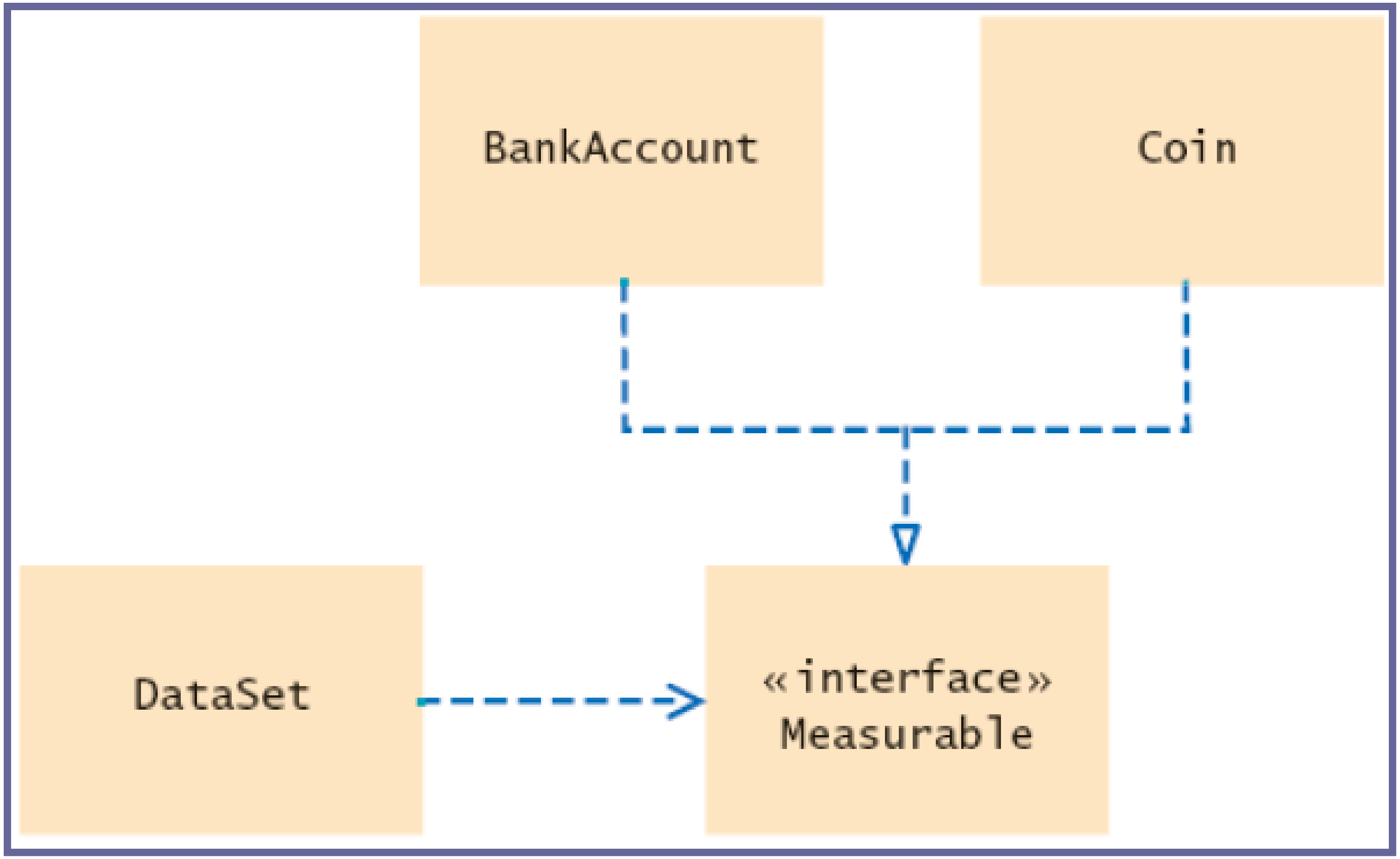
- **We must put the method in the program that implement the interface.**

- **A class can implement more than one interface type**
  - Class must define all the methods that are required by all the interfaces it implements

UML Diagram of Dataset and Related Classes

- Interfaces can reduce the coupling between classes
- UML notation:
  - Interfaces are tagged with a "stereotype" indicator «interface»
  - A dotted arrow with a triangular tip denotes the "is-a" relationship between a class and an interface
  - A dotted line with an open v-shaped arrow tip denotes the "uses" relationship or dependency
- Note that `DataSet` is *decoupled* from `BankAccount` and `Coin`

UML



Generic `DataSet` for Measureable Objects

```
public class DataSet
{
   . . .
   public void add(Measurable x)
   {
      sum = sum + x.getMeasure();
      if (count == 0 || maximum.getMeasure() < x.getMeasure())
         maximum = x;
      count++;
   }

   public Measurable getMaximum()
   {
      return maximum;
   }

   private double sum;
   private Measurable maximum;
   private int count;
}
```

File DataSetTester.java

```java
/**
   This program tests the DataSet class.
*/
public class DataSetTester
{
   public static void main(String[] args)
   {
      DataSet bankData = new DataSet();

      bankData.add(new BankAccount(0));
      bankData.add(new BankAccount(10000));
      bankData.add(new BankAccount(2000));

      System.out.println("Average balance = "
            + bankData.getAverage());
      Measurable max = bankData.getMaximum();
      System.out.println("Highest balance = "
         + max.getMeasure());
```

File DataSetTester.java

```java

      DataSet coinData = new DataSet();

      coinData.add(new Coin(0.25, "quarter"));
      coinData.add(new Coin(0.1, "dime"));
      coinData.add(new Coin(0.05, "nickel"));

      System.out.println("Average coin value = "
            + coinData.getAverage());
      max = coinData.getMaximum();
      System.out.println("Highest coin value = "
            + max.getMeasure());
   }
}
```

Output

```
Average balance = 4000.0
Highest balance = 10000.0
Average coin value = 0.13333333333333333
Highest coin value = 0.25
```

Converting Between Class and Interface Types

- Interfaces are used to express the commonality between classes
- You can convert from a class type to an interface type, provided the class implements the interface

```
BankAccount account = new BankAccount(10000);
Measurable x = account; // OK
```

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

Converting Between Class and Interface Types

- You can not convert between unrelated types

Measurable x = new Rectangle (5,10,20,30); // illegal

- Because `Rectangle` doesn't implement `Measurable`
- **`Rectangle can’t implement Measurable because it is a system class`**

Casts

- Add coin objects to `DataSet`

```
DataSet coinData = new DataSet();
coinData.add(new Coin(0.25, "quarter"));
coinData.add(new Coin(0.1, "dime"));
. . .
Measurable max = coinData.getMaximum(); // Get the largest coin
```

- What can you do with it? It's not of type `Coin`

```
String name = max.getName(); // ERROR
```

Continued…

Casts

- 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:

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

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

Casts

- Difference with casting numbers:
  - When casting number types you agree to the information loss
  - When casting object types you agree to that risk of causing an exception

Polymorphism

- Interface variable holds reference to object of a class that implements the interface

  ```
  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

Continued…

Polymorphism

- You can call any of the interface methods:

```
double m = x.getMeasure();
```

- Which method is called?

Polymorphism

- 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 (many shapes): Behavior can vary depending on the actual type of an object

Continued…

Polymorphism

- Called *late binding*: resolved at runtime
- Different from overloading; overloading is resolved by the compiler (*early binding*)
- Remember – overloading is when you have 2 methods with the same name. The explicit parameter determines which method will be used.

Using Interfaces for Callbacks

- Limitations of `Measurable` interface:
- Can add `Measurable` interface only to classes under your control
- Can measure an object in only one way E.g., cannot analyze a set of savings accounts both by bank balance and by interest rate
- Callback mechanism: allows a class to call back a specific method when it needs more information

Using Interfaces for Callbacks

- `Object` is the "lowest common denominator" of all classes
- In previous `DataSet` implementation, responsibility of measuring lies with the added objects themselves
- Alternative: Hand the object to be measured to a method:

```
public interface Measurer
{
   double measure(Object anObject);
}
```

Using Interfaces for Callbacks

- `add` method asks measurer (and not the added object) to do the measuring

```
public void add(Object x)
{
   sum = sum + measurer.measure(x);
   if (count == 0 || measurer.measure(maximum) < measurer.measure(x))
      maximum = x;
   count++;
}
```

Using Interfaces for Callbacks

- You can define measurers to take on any kind of measurement

```
public class RectangleMeasurer implements Measurer
{
   public double measure(Object anObject)
   {
      Rectangle aRectangle = (Rectangle) anObject;
      double area = aRectangle.getWidth() * aRectangle.getHeight();
      return area;
   }
}
```

Using Interfaces for Callbacks

- Must cast from `Object` to `Rectangle`

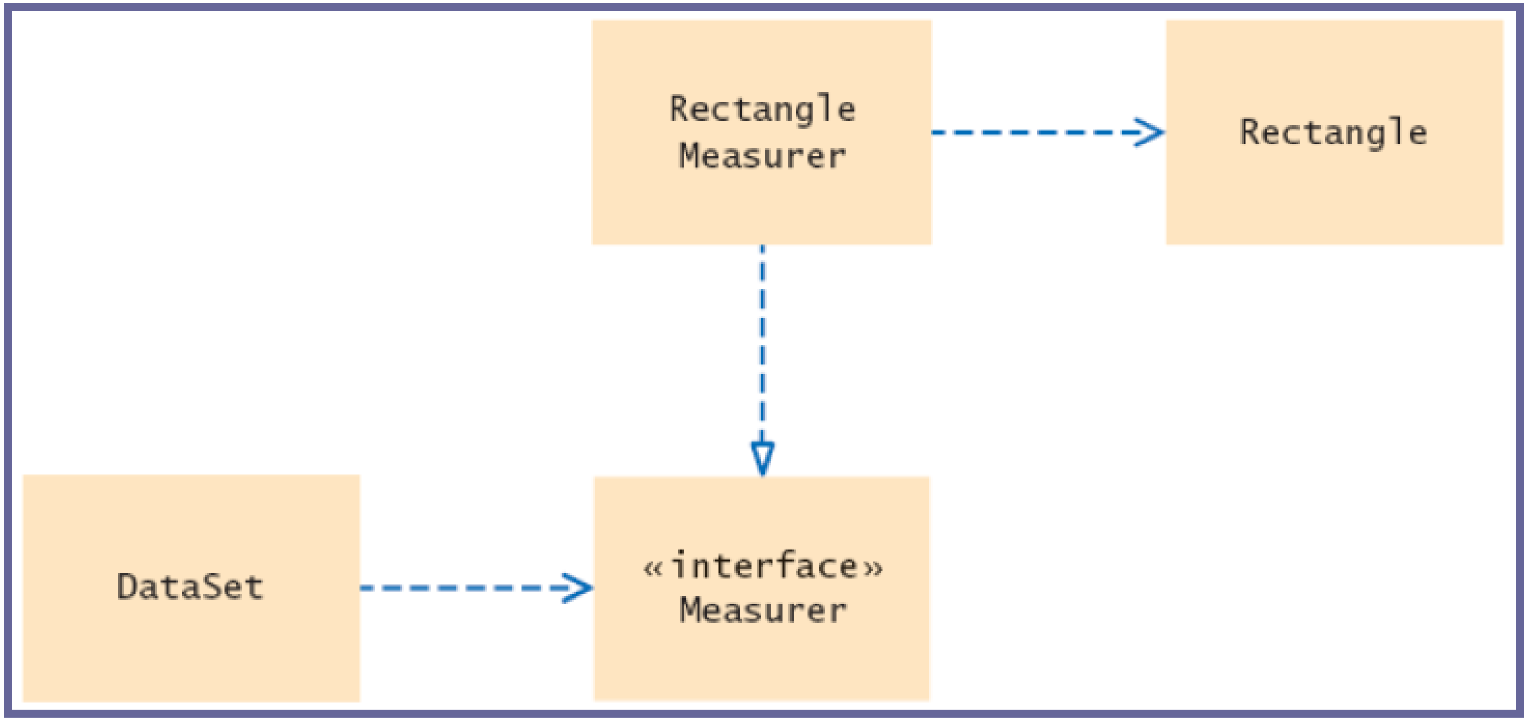
```
Rectangle aRectangle = (Rectangle) anObject;
```

- Pass measurer to data set constructor:

```
Measurer m = new RectangleMeasurer();
DataSet data = new DataSet(m);
data.add(new Rectangle(5, 10, 20, 30));
data.add(new Rectangle(10, 20, 30, 40));
. . .
```

UML

- Note that the `Rectangle` class is decoupled from the `Measurer` interface



Inner Classes

- Trivial class can be defined inside a method

```
public class DataSetTester3
{
   public static void main(String[] args)
   {
      class RectangleMeasurer implements Measurer
      {
         . . .
      }
      Measurer m = new RectangleMeasurer();
      DataSet data = new DataSet(m); . . .
   }
}
```

Continued…

Inner Classes

- If inner class is defined inside an enclosing class, but outside its methods, it is available to all methods of enclosing class
- Compiler turns an inner class into a regular class file:

```
DataSetTester$1$RectangleMeasurer.class
```

Syntax 11.3: Inner Classes

```
Declared inside a method

class OuterClassName
{
   method signature
   {
      . . .
      class InnerClassName
      {
         // methods
         // fields
      }
      . . .
   }
   . . .
}
```

```
Declared inside the class

class OuterClassName
   {
      // methods
      // fields
      accessSpecifier class
         InnerClassName
      {
         // methods
         // fields
      }
   . . .
}
```

Continued…

Syntax 11.3: Inner Classes

Example:

```
public class Tester
{
   public static void main(String[] args)
   {
      class RectangleMeasurer implements Measurer
      {
         . . .
      }
      . . .
   }
}
```

Purpose:

To define an inner class whose scope is restricted to a single method or the methods of a single class

File FileTester3.java

```java
import java.awt.Rectangle;

/**
   This program demonstrates the use of a Measurer.
*/
public class DataSetTester3
{
   public static void main(String[] args)
   {
      class RectangleMeasurer implements Measurer
      {
         public double measure(Object anObject)
         {
            Rectangle aRectangle = (Rectangle) anObject;
            double area
                  = aRectangle.getWidth()
                        * aRectangle.getHeight();
            return area;
```

Continued…

File FileTester3.java

```java
         }
      }

      Measurer m = new RectangleMeasurer();

      DataSet data = new DataSet(m);

      data.add(new Rectangle(5, 10, 20, 30));
      data.add(new Rectangle(10, 20, 30, 40));
      data.add(new Rectangle(20, 30, 5, 10));

      System.out.println("Average area = " + data.getAverage());
      Rectangle max = (Rectangle) data.getMaximum();
      System.out.println("Maximum area rectangle = " + max);
   }
}
```

Accessing Surrounding Variables

- Local variables that are accessed by an inner-class method must be declared as final
- Inner class can access fields of surrounding class that belong to the object that constructed the inner class object
- An inner class object created inside a static method can only access static surrounding fields