Computer Science 2

Lecture 2

Interfaces, Polymorphism, and Generics in Java

Overview

- Interfaces
- Interface Implementations
- Converting between Interfaces and Classes
- Polymorphism
- Strategy Pattern
- Processing Time Events
- Java Generics

Class `DataSet` for BankAccounts

- Assume that we have to write a class **`DataSet`** for **`BankAccount`** objects.
- A **DataSet** object has to add **`BankAccount`** objects one by one, and to output the reference to a **`BankAccount`** object with max balance.
- We will keep track of the sum total of all accounts, just for an example

Class DataSet for BankAccounts

```
public class DataSet
{ ...
   public void add(BankAccount x)
 { sum = sum + x.getBalance();
   if (count == 0 ||
       max.getBalance() < x.getBalance())
     max = x;
   count++;
 }
 public BankAccount getMaximum()
 { return max;
 }
 private double sum;
 private BankAccount max;
 private int count;
}
```

Class `DataSet` for Coins

- That was handy
- Let us try the same thing for coins
- First we will need a class that represents coins
- Coins have two properties
- Name (penny, dime, …)
- Value (1 cent, 10 cents, …)
- They get these properties when they are created
- And they never change
- For now, all we care about is their value

Coin Class

```
public class Coin
{         public Coin(double aValue,String aName)
        {    value = aValue;
             name = aName;
        }
        public double getValue()
        {    return value;
        }
        public String getName( ) {
             return name;
}
        private double value;
        private String name;
}
```

Class `DataSet` for Coins

- Now we can build a class that will do for coins what we did earlier for bank accounts
- Keep a running total
- Be able to return the Coin with the maximum value
- This may look familiar

Another DataSet for Coins

```
public class DataSet
{ ...
   public void add(Coin x)
 { sum = sum + x.getValue();
   if (count == 0 ||
       max.getValue() < x.getValue())
     max = x;
   count++;
 }
 public Coin getMaximum()
 { return max;
 }
 private double sum;
 private Coin max;
 private int count;
}
```

Class `DataSet` for Coins

- The DataSet for bank accounts and the DataSet for coins is almost identical
- The word BankAccount has been swapped out for Coin
- The getBalance method has been replaced by the getValue method
- That sounds good – you can just copy-paste, change a few words, and you are good to go
- It is actually bad
- If the logic for DataSet ever changes (and it will), now you have to change code in two different places
- And the update will go wrong, guaranteed

Class `DataSet` for Coins

- You only want one DataSet class, that can operate on both bank accounts and coins
- This way any change only has to be made in one place
- This is an example of the DRY principle
- Don’t Repeat Yourself

How to make only one `DataSet` class?

- Can we make **`DataSet`** independent of the data types?
- Yes, if **`BankAccount`**, and **`Coin`** agree on a single method **`getMeasure()`**.
- Then we can implement a single reusable **`DataSet`** class with `add` method implemented as follows:

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

- But, what is the type of the variable **`x`**?

Interfaces

- There is no direct way in Java to say a variable can be of either class A *or* class B
- And DataSet really does not care in any case
- All DataSet cares about is that the variable has a getMeasure( ) method
- What we need is a way to say “this variable belongs to a class that has the getMeasure( ) method”
- This is what *interfaces* are for

Interfaces

- The variable **`x`** will "belong to" a new interface data type
- This data type is defined below:

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

- An **`interface`** is a named collection of:
- (1) abstract methods (methods without implementations), and
- (2) constant declarations
- Note that getMeasure( ) is defined, but not implemented

Interfaces

- An interface is not a class
- You cannot create variables of an interface
- Instead, a class *implements* an interface
- When a class implements an interface, it does two things
- Promises to provide bodies for all the functions that interface defines
- Gets the variables described by the interface for free
- We will look at a few examples to see how this is done

Interface Implementation

- To implement an interface, the class must say so when the class is declared
- Then class must implement all the methods that the interface requires.
- Note that a class can implement more than one interface

```
class BankAccount implements Measurable
{ public double getMeasure()
   { return balance;
   }
   ...
}
```

```
class Coin implements Measurable
{ public double getMeasure()
  { return value;
  }
...
}
```

Interface Implementation

- Because BankAccount and Coin both implement Measurable, whenever we see something that wants a Measurable, we can pass it either a BankAccount or a Coin
- Now we can modify DataSet so that it works on Measurable objects
- Which solves our original problem – getting DataSet to work on both Coins and BankAccounts

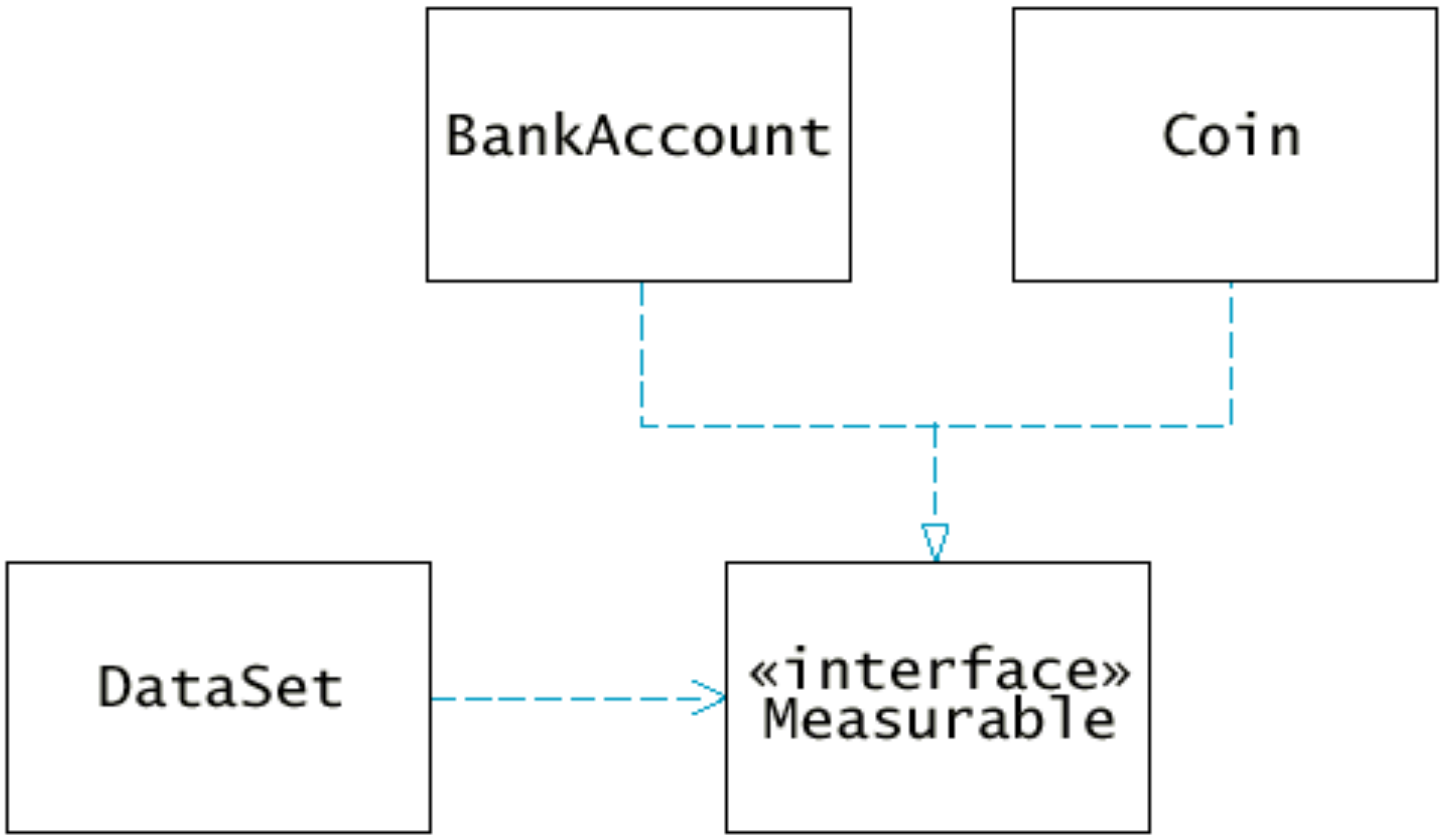
A new DataSet for Measurables

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

      max = x;
    count++;
  }
  public Measurable getMaximum()
  { return max;
  }
  private double sum;
  private Measurable max;
  private int count;
}
```

New DataSet Usage

```
public static void main(String[] args)
{ DataSet bankData = new DataSet();
  bankData.add(new BankAccount(10000));
  bankData.add(new BankAccount(2000));
  Measurable max = bankData.getMaximum();
  System.out.println("Highest balance = "
   + max.getMeasure());

  DataSet coinData = new DataSet();
  coinData.add(new Coin(0.25, "quarter"));
  coinData.add(new Coin(0.05, "nickel"));
  max = coinData.getMaximum();
  System.out.println("Highest coin value = "
   + max.getMeasure());
 }
```

UML Diagram



Note that: interfaces can reduce coupling between classes!

Interfaces

- Note that in an **`interface`**:
  - All the methods are abstract; that is, they have a name, parameters, and a return type, but they don't have an implementation;
  - All the methods are implicitly **`public`**;
  - All data declarations are implicitly constant declarations; i.e., **`public static final`**. This implies that instance fields cannot be declared in **`interfaces`**;
- **`interfaces`** are not classes; i.e., they cannot be instantiated. Thus, the line

```
Measurable x = new Measurable();
```

is a mistake!

Type Converting: Class Interface

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

```
BankAccount account = new
BankAccount(1000);
Measurable x = account;
```

- However, you cannot convert between unrelated types:

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

- This is an error since the class **`Rectangle`** doesn't implement the interface **`Measurable`**

Type Converting: Interface Class

- You need a cast to convert from an interface type to a class type:

```
DataSet coinData = new DataSet();
coinData.add(new Coin(0.25, "quarter"));
coinData.add(new Coin(0.05, "nickel"));
Measurable max = coinData.getMaximum();
String name = max.getName(); //ERROR
```

- Substitute the red line with:

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

Type Converting: Interface Class

- Why was that an error?
- After all,
- You put only Coins in CoinData
- Coin has a getName( ) method
- So anything you pull out of CoinData will be a Coin and have a getName( ) method
- So why doesn't Java just call max's getName( ) ?
- Because Java does not know max has one
- Why not?
- Because all Java knows is that max is a Measurable, and Measurable does not have getName( )

Type Converting: Interface Class

- After all, even though it is named CoinData, it accepts any Measurable
- You could have added a BankAccount in there
- After all, CoinData accepts any Measurable
- BankAccounts do not have getName( )
- So Java plays it safe and throws the error
- We will see how to avoid this later

Different Castings

- When casting number types, you lose information and you tell the compiler that you agree to the information loss:

```
int x = (int) 2.4;
```

- When casting object types, you take a risk of causing an exception, and you tell the compiler that you agree to that risk:

```
Coin maxCoin = (Coin)max;
```

Different Castings

- It is easy to get errors when casting
- For example, both of these seem like they might work:

```
Coin maxCoin = (Coin)max;
BankAccount ba = (BankAccount)max;
```

- But only one would
- Depending on what type max was before it was added to CoinData
- Get it wrong and Java will throw an error
- How can one get it right?
- You need a way to determine what class a variable is

How to Control Casting?

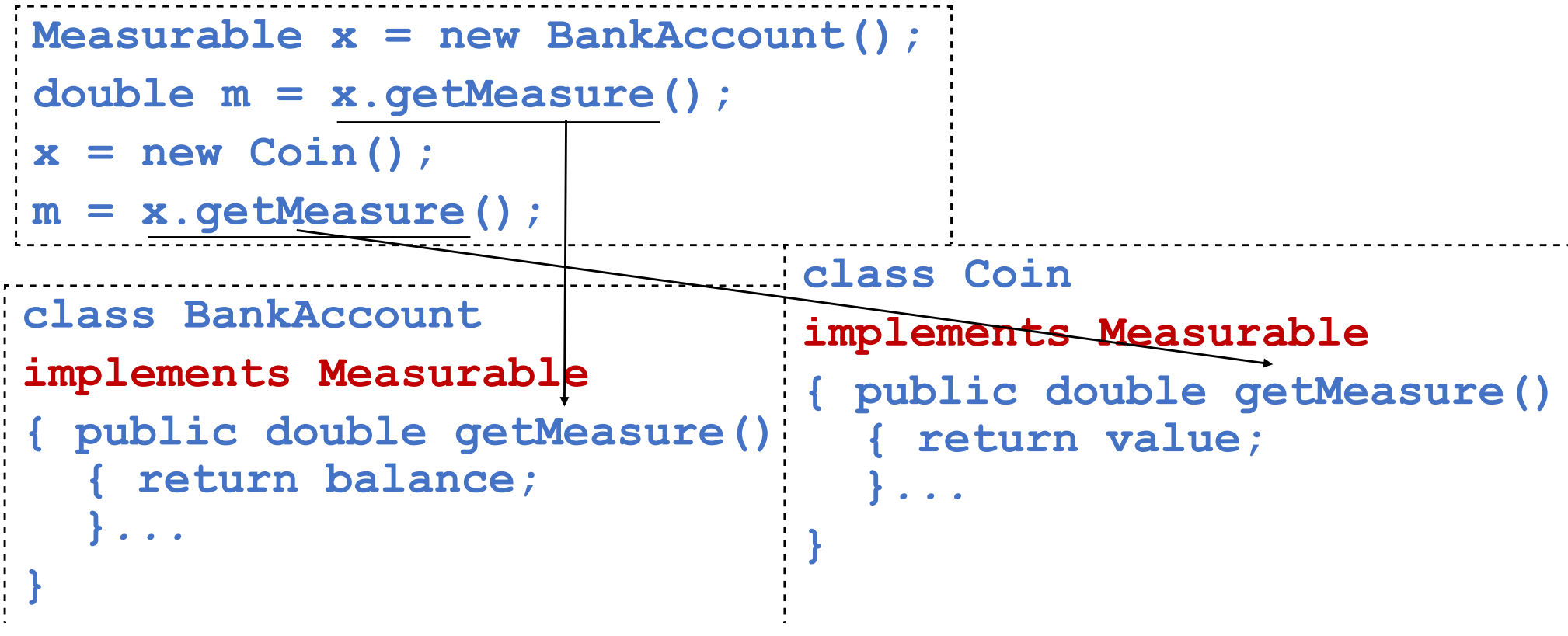
- Use the **`instanceof`** operator. It tests whether an object belongs to a particular type:

```
if (x instanceof Coin)
{ Coin c = (Coin)x;
  ...
}
```

- The **`instanceof`** operator returns **`true`** if the object is an instance of the class. Otherwise, it returns **`false`**.

Polymorphism

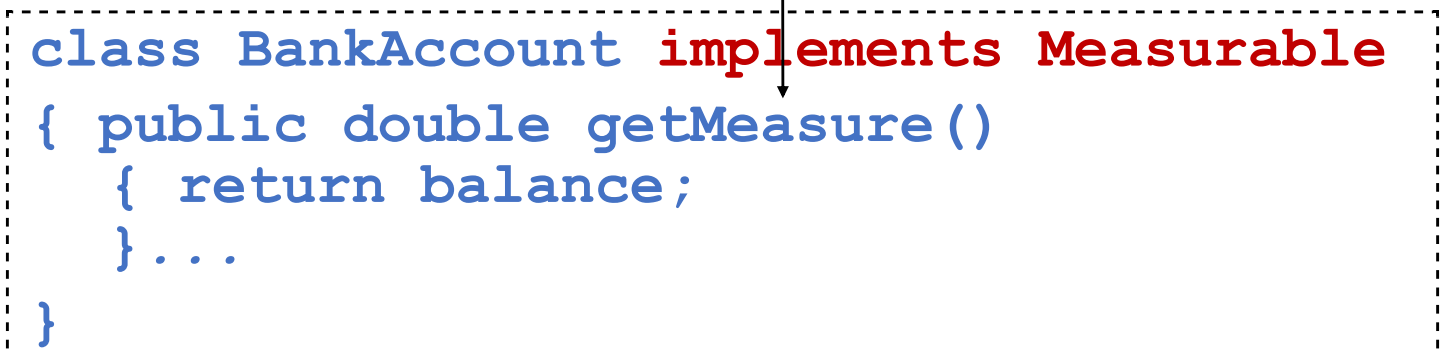
- Polymorphism denotes the principle that behavior can vary depending on the type of an object.
- Polymorphism is implemented in Java so that the actual type of an object determines the method called.



Polymorphism and Overloading

- Polymorphism is like overloading, but different
- An overloaded method is chosen during compilation (**Early binding**);
- A polymorphic method is chosen when the program is running (**Late binding**).

```
BankAccount y = new BankAccount();
Measurable x = new BankAccount(10);
double m = x.getMeasure();
```



Interfaces Summary

- An interface describes (part of) the interface of a class
- The number of methods, their names, parameters, and return types
- A class that implements an interface defines the implementation of the interface
- How the methods actually work
- If class C implements interface I, then a variable of class C can be used wherever the code wants an object of interface I
- For example, if a parameter wants something of interface I, an object of class C can be passed in

Interfaces Summary

- Let us say you have a parameter of type I, and classes B and C that implement it
- An object of either class B or class C can be passed in
- When using the parameter, you may only call the methods defined in I
- Even if other methods are defined in B and C
- Java remembers what class the parameter is
- So it can call the correct method in either B or C
- If you want a variable of the original class (B or C) back, you can cast the parameter from I to either B or C
- Use instanceof to determine which class is the right one

The Strategy Pattern

- There is a shortcoming to interfaces, however
- Suppose you are writing a graphical application
- You would like to know
- The total area covered by java.awt.Rectangle objects
- The biggest Rectangle
- This sounds like a job for DataSet
- But DataSet will not work
- It wants a Measurable, and Rectangle is not one
- And you cannot cast Rectangle to be a Measurable
- This leads us to the problem

The Strategy Pattern

- Disadvantages of the **`Measurable`** interface:
- You can't force classes that aren't under your control to implement the interface (for example class `Rectangle`);
- You can measure an object in only one way.
- Solution: let another object carry out the measurements. The class of this object has to implement a new interface:

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

The Strategy Pattern

- An example of class implementing the `Measurer` interface:

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

- `Measurer` object has to be provided to `DataSet` constructor:

```
Measurer m = new RectangleAreaMeasurer();
DataSet data = new DataSet(m);
```

The Strategy Pattern

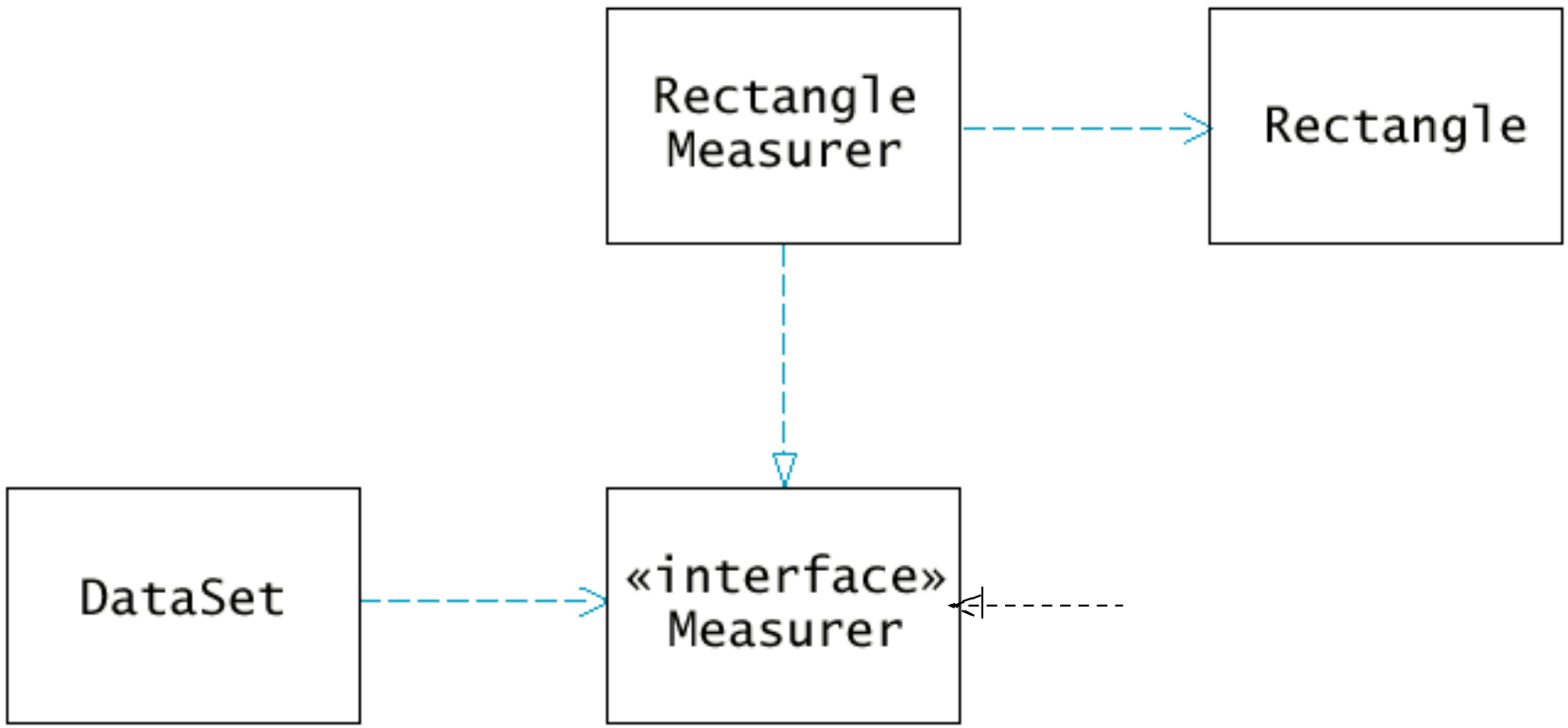
- The new class **`DataSet`** is constructed with a **`Measurer`** object (an object of a class that implements the **`Measurer`** interface);
- The method `add` is as follows:

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

```
public class DataSet
{  public DataSet(Measurer aMeasurer)
   {  sum = 0;
      count = 0;
      maximum = null;
      measurer = aMeasurer;
   }
   public void add(Object x)
   {  sum = sum + measurer.measure(x);
      if (count == 0 || measurer.measure(maximum) <
               measurer.measure(x))
         maximum = x;
      count++;
   }
   public double getAverage()
   {  if (count == 0) return 0;
      else return sum / count;
   }
   public Object getMaximum()
   { return maximum; }
   private double sum;
   private Object maximum;
   private int count;
   private Measurer measurer;
}
```

```
public class DataSetTest
{  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;
         }
      }
      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));
      System.out.println("AverArea = " + data.getAverage());
      Rectangle max = (Rectangle)data.getMaximum();
      System.out.println("Maximum area = " + max);
   }
}

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

UML Diagram


Rectangle Measurer
Rectangle
DataSet
«interface» Measurer

Processing Timer Events

- Let us look at a practical example of implementing an interface from a Java library
- `javax.swing.Timer` objects generate timer "events" at fixed intervals;
- When a timer event occurs, the `Timer` object needs to notify some object called an *event listener*;
- The class of the event-listener object must implement the `ActionListener` interface:

```
public interface ActionListener
 { void actionPerformed(ActionEvent event);
}
```

Processing Timer Events

- Here is an outline of a class that implements ActionListener:

```
class MyListener implements ActionListener
{    void actionPerformed(ActionEvent event)
       { // place your actions here
  }
}
```

- Which is great, but we also need to hook up a MyListener object to a Timer so that the MyListener object actually gets notifications

Processing Timer Events

- Pass the reference of the listener to the **`Timer`** constructor and then start:

```
  MyListener listener = new MyListener();
Timer t = new Timer(interval, listener);
t.start();
```

- Timer `t` calls the **`actionPerformed`** method of the **`listener`** object every **`interval`** milliseconds!

Processing Timer Events

```
public class TimerTest
{  public static void main(String[] args)
   {  class CountDown implements ActionListener
      {  public CountDown(int initialCount)
         { count = initialCount; }
         public void actionPerformed(ActionEvent event)
         {  if (count >= 0)
               System.out.println(count);
            if (count == 0)
               System.out.println("Liftoff!");
            count--;
         }
         private int count;
      }
      CountDown listener = new CountDown(10);
      final int DELAY = 1000;
      Timer t = new Timer(DELAY, listener);
      t.start();
   }
}
```

Generics in Java

- Suppose you have a DataSet that you only use for Coins
- It would be nice if
- It would accept only Coins as a parameter to the add method
- It would return a Coin from the getMaximum method
- It would also be nice if you could do the same thing for a DataSet that held only BankAccount objects
- But you do not want to have two separate DataSet classes
- It would be handy if there were a way to work with Java
- I will put only Coins in this DataSet object
- Java will return Coins (and only Coins) from this DataSet object
- Otherwise, DataSet will work like one would expect
- This would be a little involved for DataSet, so we will look at simpler examples

Generics in Java

- **Generics** are a facility of generic programming that were added to the Java programming language in 2004 within version J2SE 5.0.
- **Generics** were designed to extend Java's type system to allow *"a type or method to operate on objects of various types while providing compile-time type safety".*

Motivation for Generics in Java

- The code below is compiled without error. However, it throws a **runtime exception** (`java.lang.ClassCastException`). This type of logic error *cannot* be detected during compile time.

```
ArrayList v = new ArrayList();
v.add("test");
Integer i = (Integer)v.get(0); // Run time error
```

- The above code fragment can be rewritten using generics as follows:

```
  ArrayList<String> v = new ArrayList<String>();
v.add("test");
Integer i = v.get(0); // Compilation-time error
```

Motivation for Generics in Java

- What we get is "***compile-time type safety***"; i.e. the errors appear during program development, not when they are used.
- Why is it better to get an error while developing the program, rather than when the program is used?
- If you are writing code for an airplane, the time to find the bugs is during development, not when the plane is flying

Generics in Java

- Generics come in two flavors
- Generic classes (and interfaces)
- Generic methods
- Java generic classes enable programmers to specify a set of related types with a single class.
- Java generic methods enable programmers to specify a set of related methods with a single method.
- This is achieved by using **type variables.**
- A **type variable** is an unqualified identifier; i.e. a variable whose value is a type.

Generic Classes

- A **class** is *generic* if it declares one or more type variables. These type variables are known as the ***type*** parameters of the class.

```
public class A<T>
  {  private T t;
     public T get()
    { return this.t;}
    public void set(T t)
    { this.t = t; }
     public static void main(String args[])
    { A<String> a = new A<String>();
       a.set("Computer Science 2"); //valid
     }
  }
```

Generic Interfaces

- An interface is *generic* if it declares one or more type variables. These type variables are known as its formal ***type*** parameters.

```
public interface myInterface<T>
{ void myMethod(T t);
}
public class MyClass implements myInterface<Integer>
{    public void myMethod(Integer t)
   { System.out.println(t); }
}
```

Generic Methods

- A **method** is *generic* if it declares one or more type variables. These type variables are known as the formal ***type*** parameters of the method.

```
public static <T> boolean isEqual(A<T> g1, A<T> g2)
{ return g1.get().equals(g2.get());
}
public static void main(String args[])
{  A<String> g1 = new A<String>();
   g1.set("Computer Science 2");
   A<String> g2 = new A<String>();
   g2.set("Computer Science 2");
  boolean equal = isEqual(g1, g2);
}
```

What we have learned

- Interfaces
- Interface Implementations
- Converting between Interfaces and Classes
- Polymorphism
- Strategy Pattern
- Nested Classes
- Processing Time Events
- Generics in Java