OO-Concept: Interface and Polymorphism

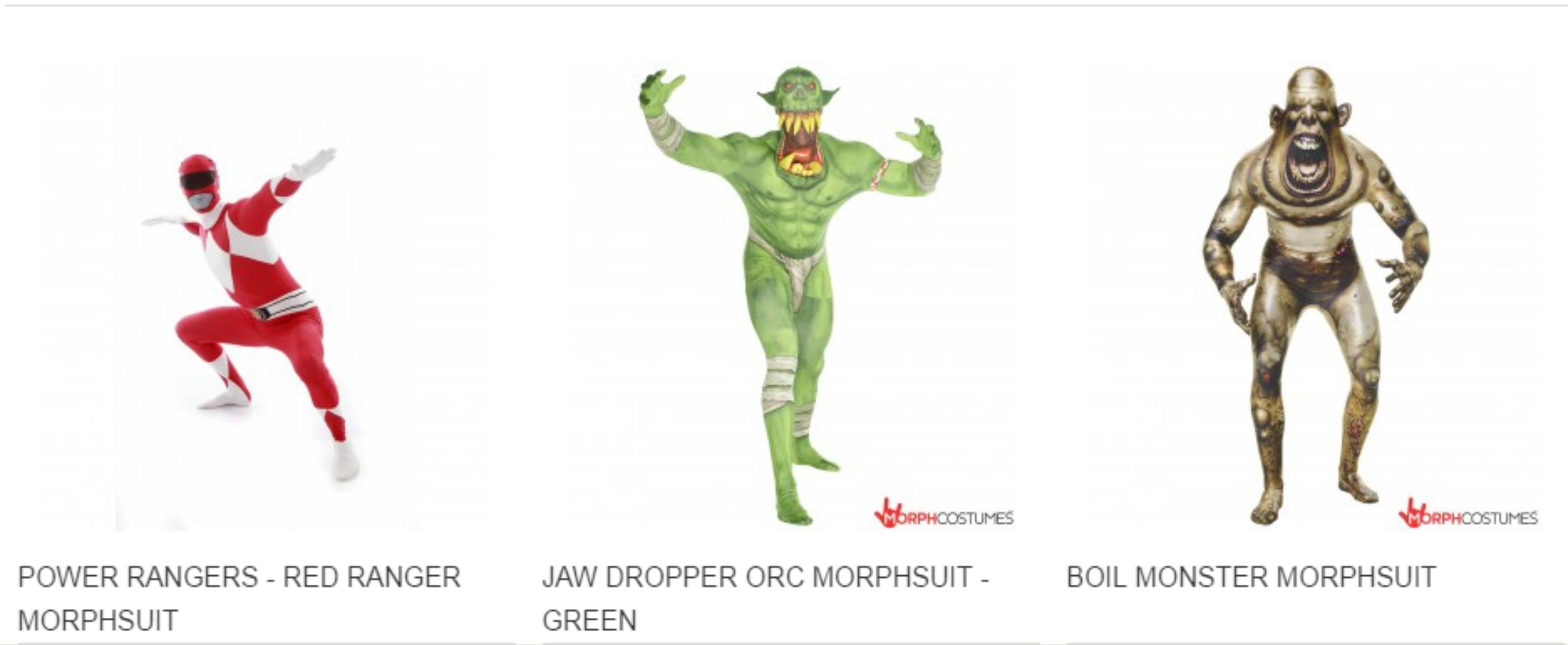
objective

- To be able to declare and use interface types
- To understand the concept of polymorphism
- To appreciate how interfaces can be used to decouple classes
- To learn how to implement helper classes as inner classes
- //Inner class for GUI event listeners will be covered later

Synopsis

- In order to increase programming productivity, we want to be able to reuse software components in multiple projects. However, some adaptations are often required to make reuse possible.
- In this chapter, you will learn an important strategy for separating the reusable part of a computation from the parts that vary in each reuse scenario.
- The reusable part invokes methods of an interface. It is combined with a class that implements the interface methods.
- To produce a different application, you simply plug in another class that implements the same interface.
- The program's behavior varies according to the implementation that is plugged in—this phenomenon is called polymorphism.

What is Polymorphism?



POWER RANGERS - RED RANGER MORPHSUIT

JAW DROPPER ORC MORPHSUIT - GREEN

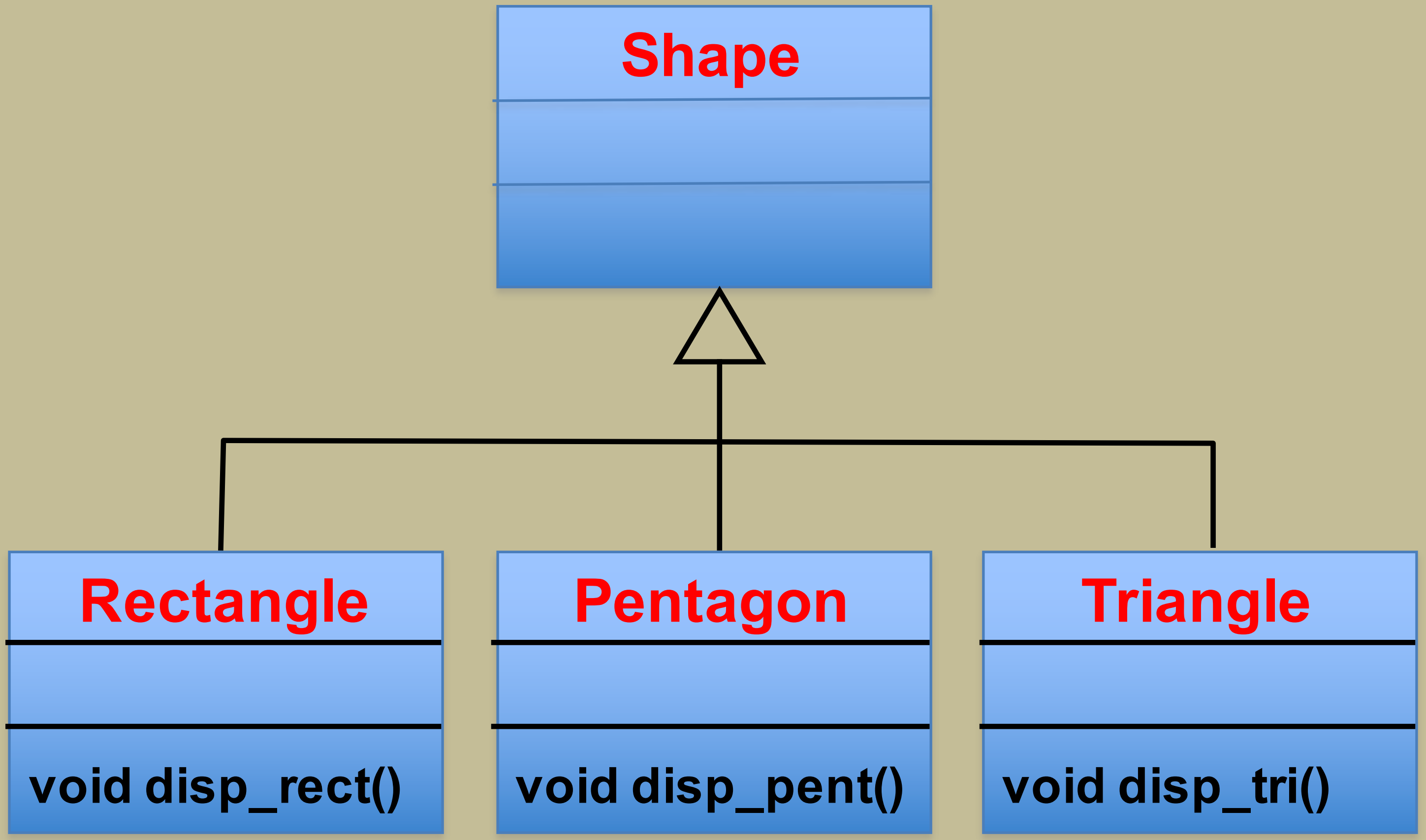
BOIL MONSTER MORPHSUIT

- Many (Poly) Shapes (Morphs)
- Calling method transform() may results in different appearance depending on which object (Red Ranger, JawDropper or BoilMonster) hence MorphSuit available at runtime
- JVM dynamic binding (or late@runtime binding) leads to what is known as Polymorphism whereby the same method call can lead to different behaviors depending on the **type of object** on which the method call is made

Using Interfaces for Algorithm Reuse

- In Java, there are 3 kinds of polymorphism :
  - Overriding an inherited method (in this chapter)
  - Implementing an abstract method
  - Implementing a Java interface (in this chapter)
- It is possible to make a service available to a wider set of inputs by focusing on the essential operations that the service requires
- ***Interface types*** are used to express these common operations.

Method Overriding



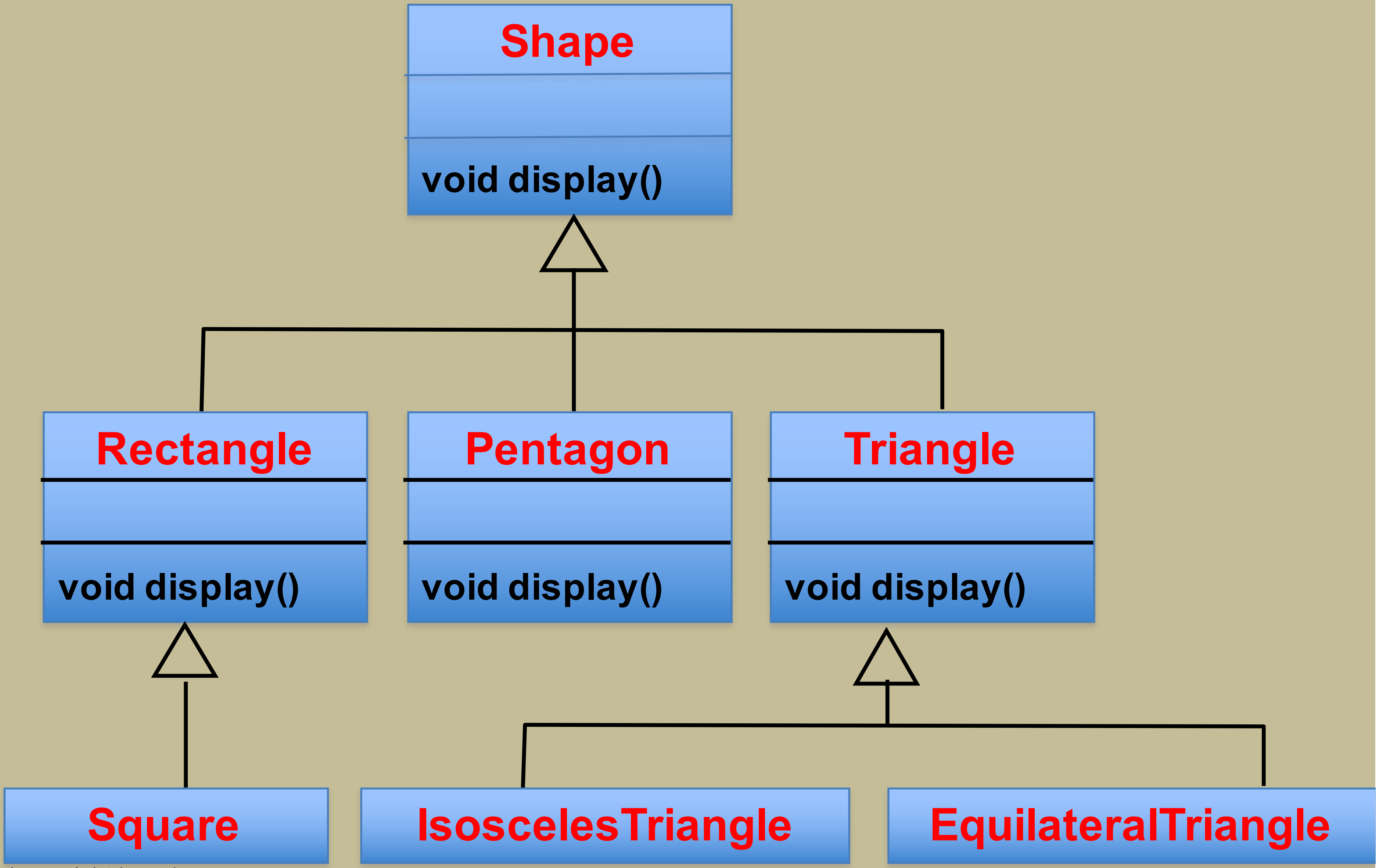
- The following code displays each element of a Shape array:

```
class Application {
    public static void main(String[ ] args) {
            Shape shapes[] = new Shape[50];
            shapes[0] = new Rectangle(...);
            shapes[1] = new Triangle(...);
            shapes[2] = new Pentagon(...);
            for (int i=0; i < 3; i++)
                    if (shapes[i] instanceof Rectangle)
                            shapes[i].disp_rect();
                    else if (shapes[i] instanceof Pentagon)
                            shapes[i].disp_pent();
                    else if (shapes[i] instanceof Triangle)
                            shapes[i].disp_tri();
    }

}
```

- Note the use of ***subtyping*** in the program: various kinds of shapes can be stored in a Shape array.
- Note also that the program uses the ***instanceof*** operator to test an object’s class. It returns true if its left operand is an instance of its right operand.
- The message to be sent to the current array element depends on the type of that element.

- What if we wish to extend the program to include hexagons?

- Now consider the following:



```
class Shape {
            …
      public void display( )  { }
      …
}
```

The Shape class defines a display() method.

```
class Rectangle extends Shape {
            …
      public void display( )  {
            …
      }
      …
}
```

Method display() is overridden in the Rectangle class.

```
class Square extends Rectangle {
            …
}
```

```
class Triangle extends Shape {
                …
      public void display( ) {
            …
      }
      …
}
```

Method display() is overridden in the Triangle class.

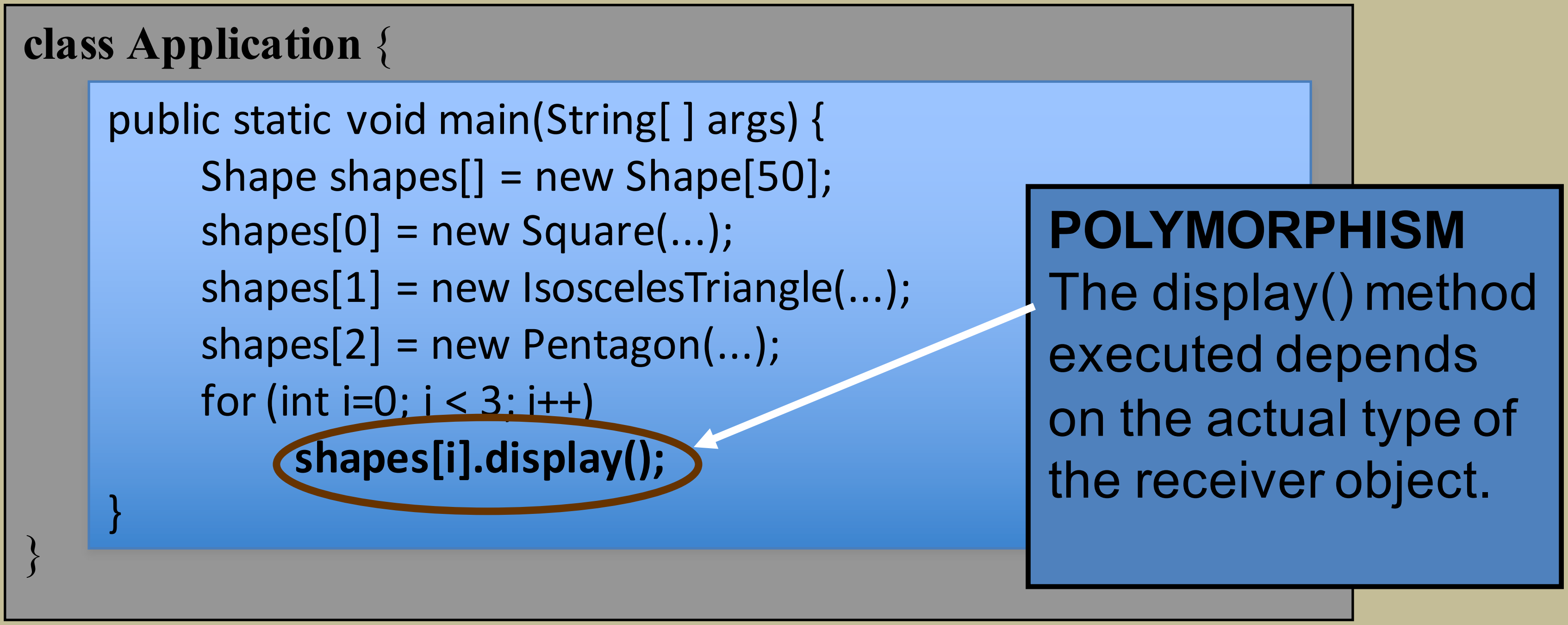
```
class EquilateralTriangle extends Triangle {
            …
}
```

```
class IsoscelesTriangle extends Triangle {
            …
}
```

```
class Pentagon extends Shape {
            …
      public void display( ) {
            …
      }
      …
}
```

Method display() is overridden in the Pentagon class.

- An example of applying polymorphism is shown below:



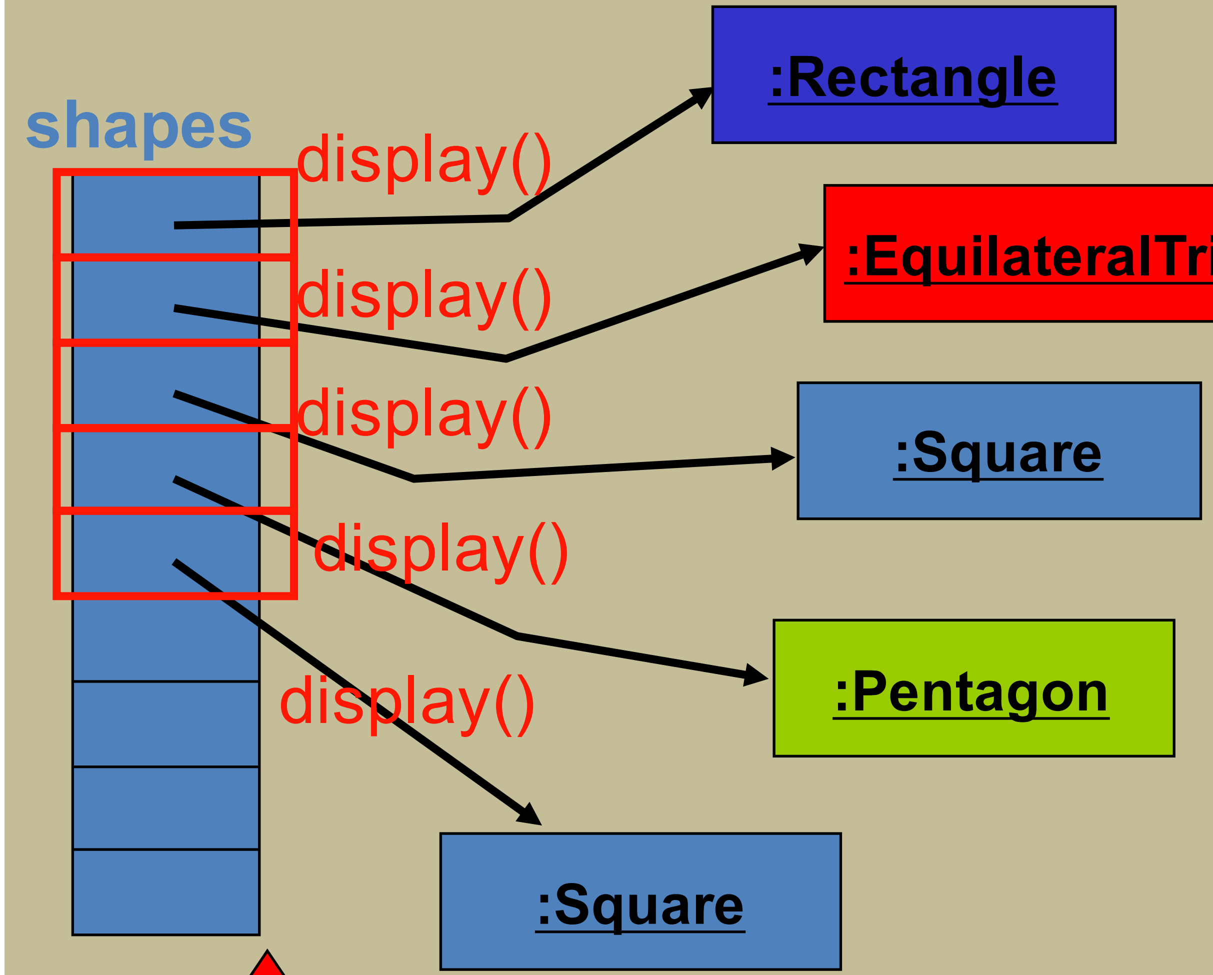
```
class Application {
    public static void main(String[ ] args) {
        Shape shapes[] = new Shape[50];
        shapes[0] = new Square(...);
        shapes[1] = new IsoscelesTriangle(...);
        shapes[2] = new Pentagon(...);
        for (int i=0; i < 3; i++)
            shapes[i].display();
    }
}
```

POLYMORPHISM

The display() method executed depends on the actual type of the receiver object.

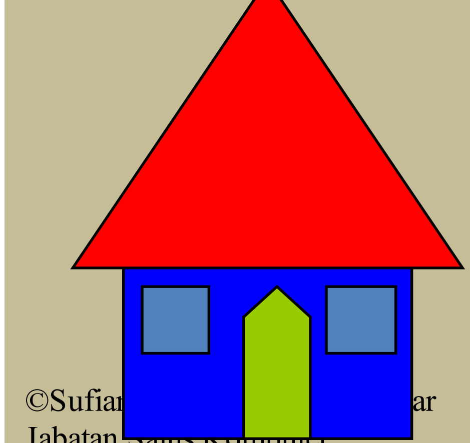
- Programmer need not test the actual type of the receiver object to determine the method to be executed.
- This results in simpler code.

- The *display( )* method that will be executed for *shapes[i].display()* depends on the actual type of the object referred to by *shapes[i]*
  - For example..



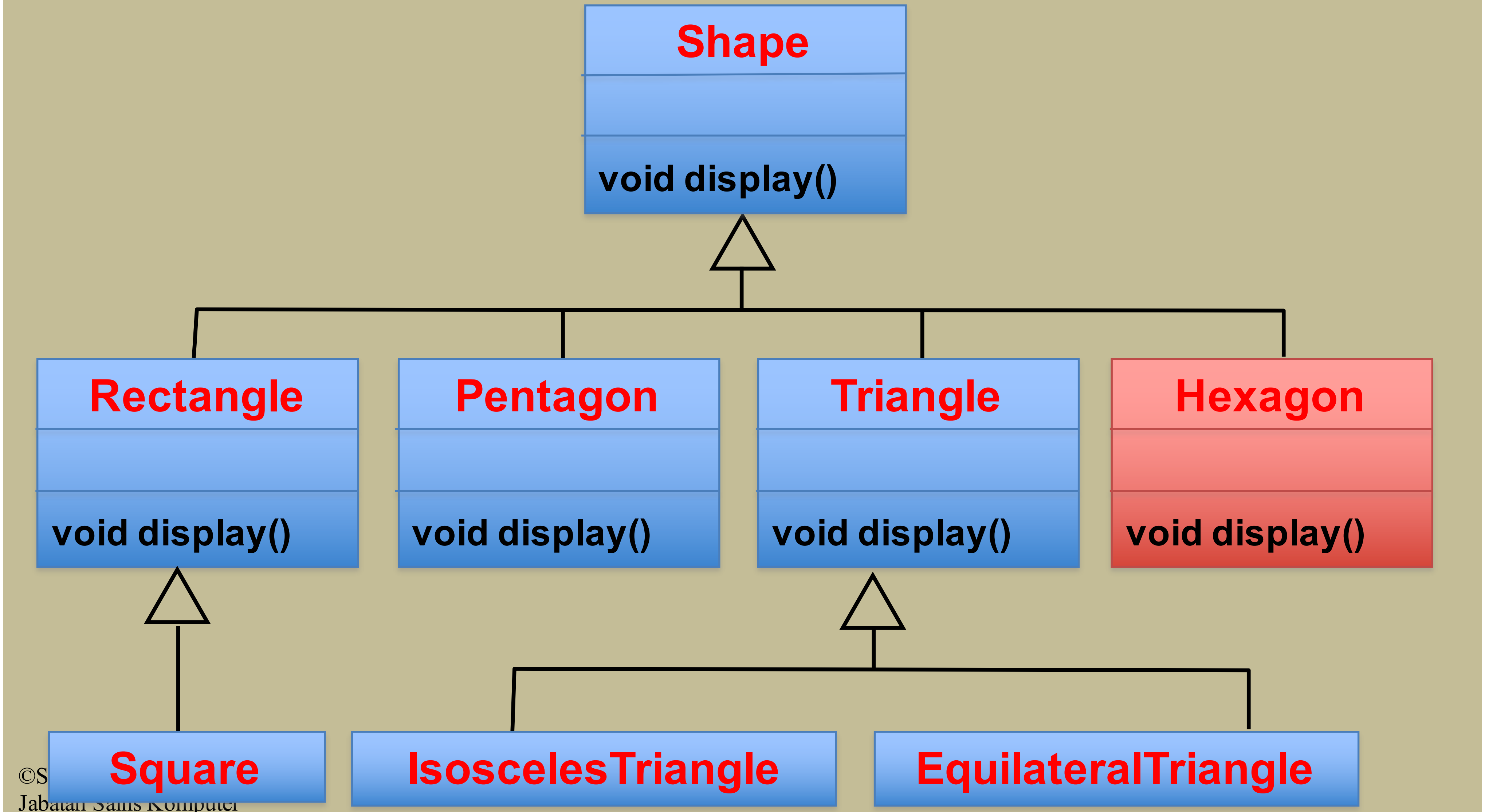
```
class Pentagon {
    …
    public void display( )
        …
    }
    …
}

class Rectangle {
    …
    public void display( )
        …
    }
    …
}
```

```
class Triangle {
    …
    public void display( ) {
        …
    }
    …
}
```



- What if we wish to extend the program to include hexagons?

- We just extend the inheritance hierarchy by adding a new subclass for hexagons.



Shape

void display()

Rectangle

void display()

Pentagon

void display()

Triangle

void display()

Hexagon

void display()

Square

IsoscelesTriangle

EquilateralTriangle

```
class Hexagon extends Shape {
        ...
    public void display( )  {
            ...
    }
    ...
}
```

- With Java’s support for polymorphism, the *for* loop in the *main()* method **DOES NOT NEED** to be modified.

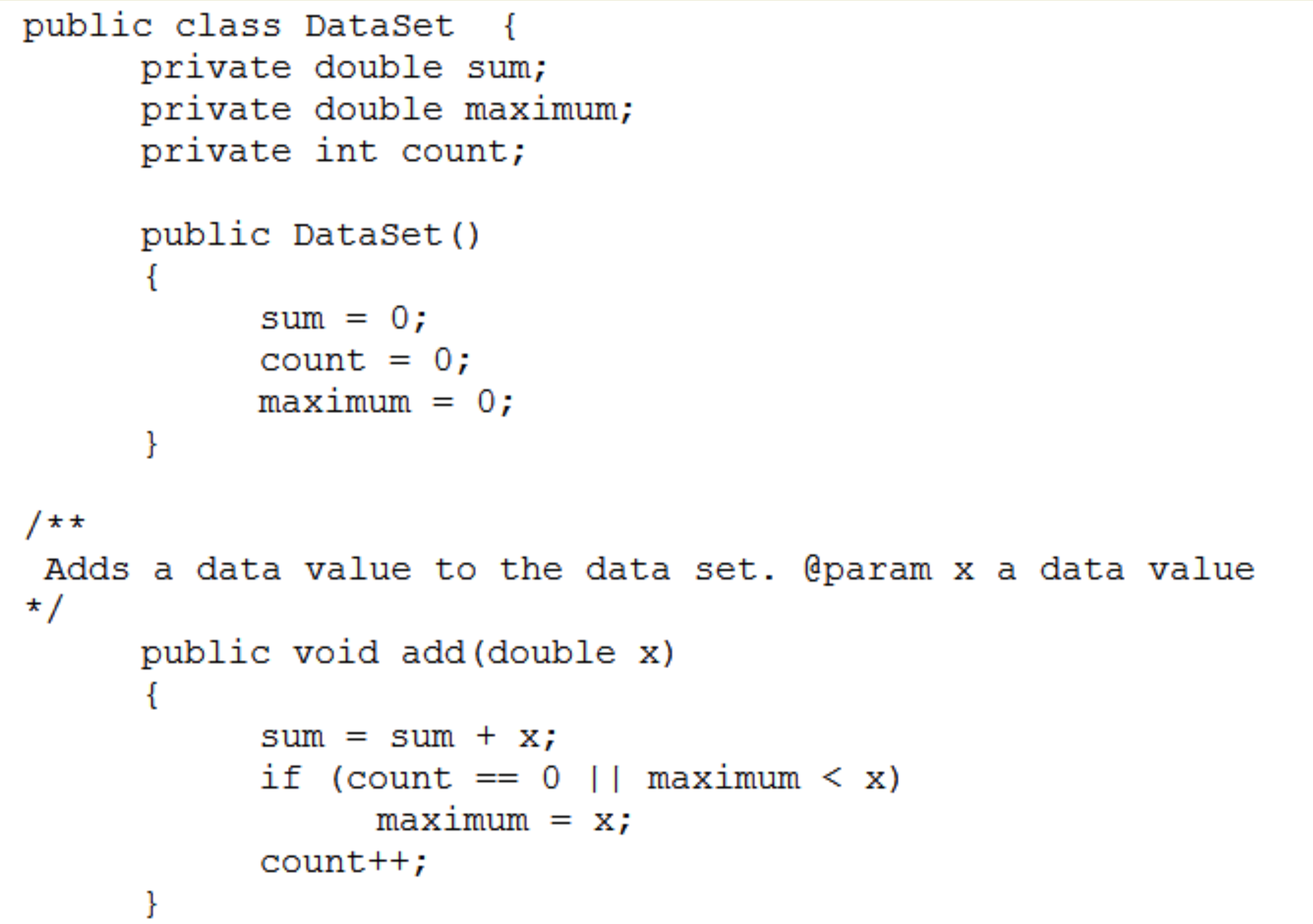
```
class Application {
    public static void main(String[ ] args) {
            Shape shapes[] = new Shape[50];
            shapes[0] = new Square(...);
            shapes[1] = new IsoscelesTriangle(...);
            shapes[2] = new Pentagon(...);
            for (int i=0; i < 3; i++)
                    shapes[i].display();
    }
}
```

Thank you, polymorphism!

There is no need to modify this statement in order to take into account Hexagon objects.

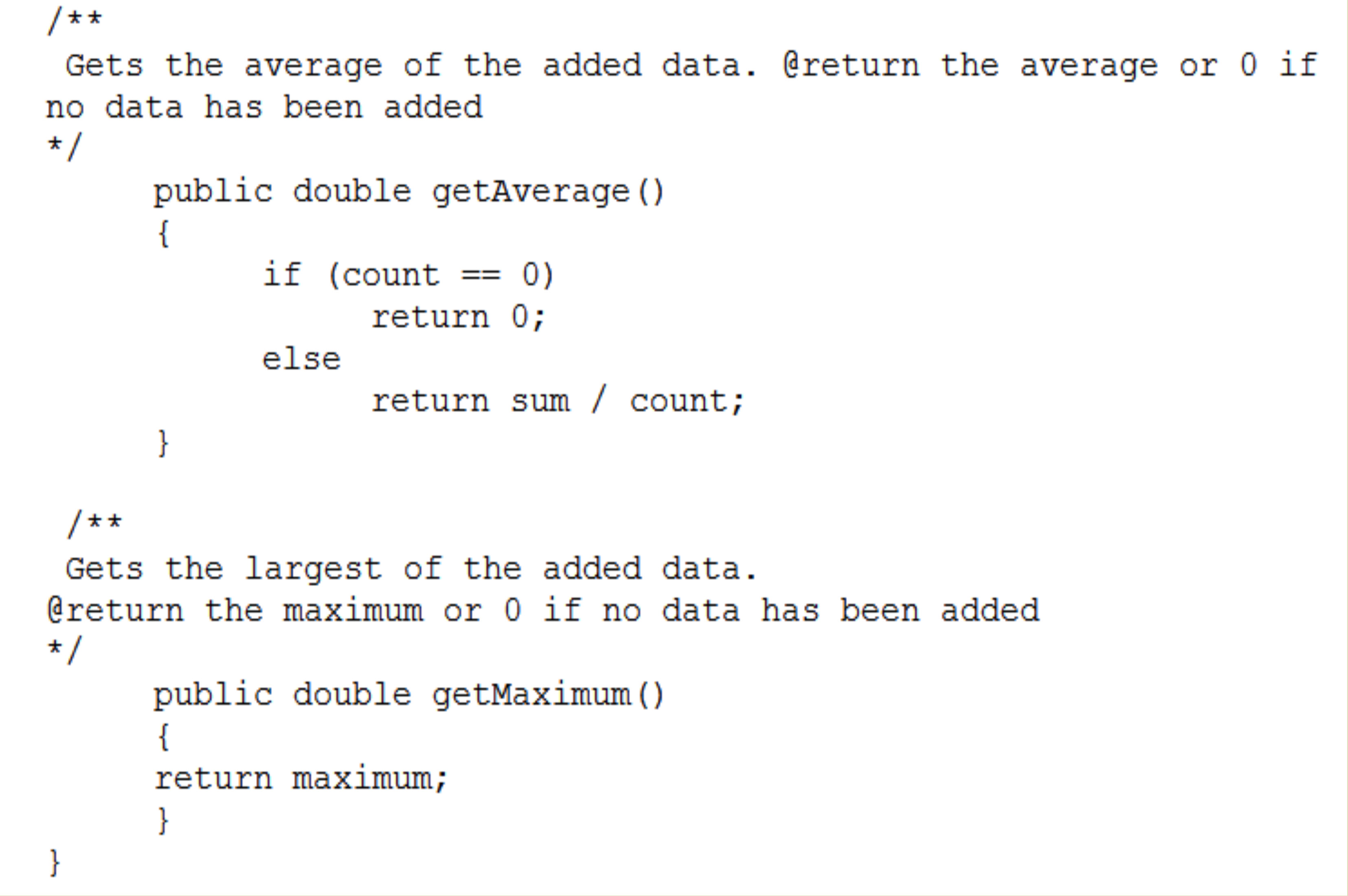
Using Interface For Polymorphism

DataSet example: provides a service to compute average and maximum of a set of input values



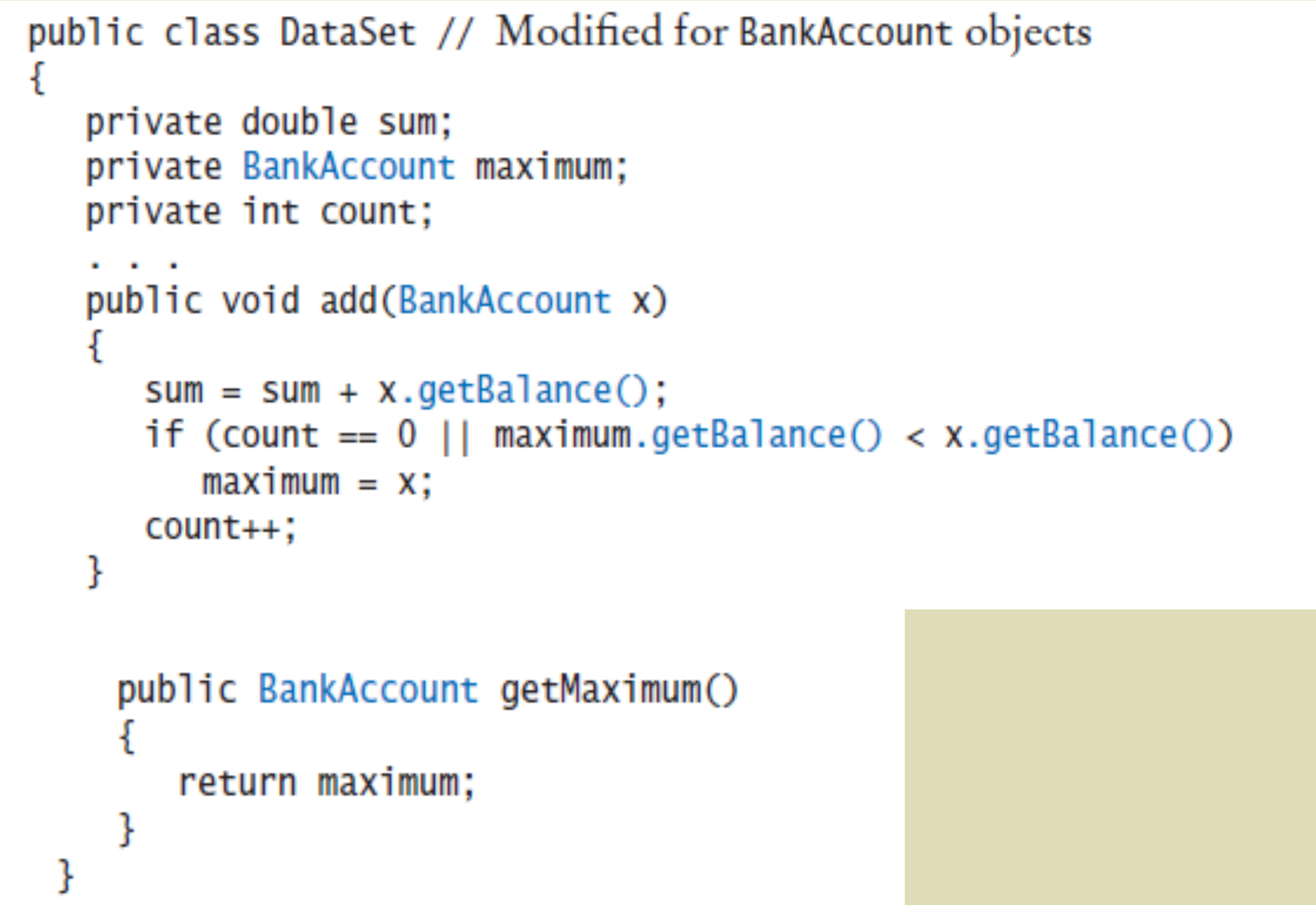
```
public class DataSet  {
      private double sum;
      private double maximum;
      private int count;

      public DataSet()
      {
            sum = 0;
            count = 0;
            maximum = 0;
      }

/**
 Adds a data value to the data set. @param x a data value
*/
      public void add(double x)
      {
            sum = sum + x;
            if (count == 0 || maximum < x)
                  maximum = x;
            count++;
      }
```

DataSet (cont)



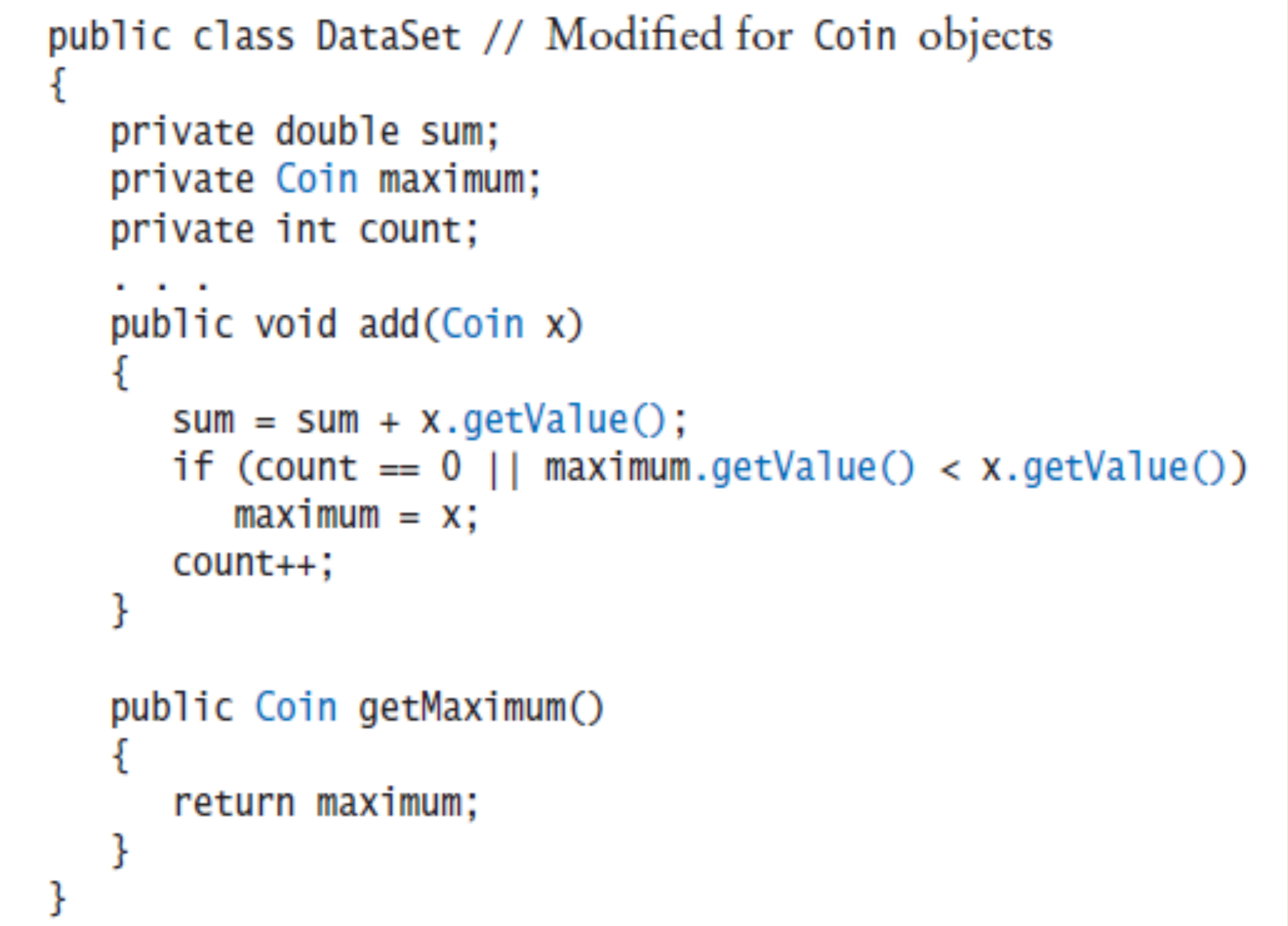
```
/**
 Gets the average of the added data. @return the average or 0 if
no data has been added
*/
     public double getAverage()
     {
           if (count == 0)
                return 0;
           else
                return sum / count;
     }

 /**
 Gets the largest of the added data.
@return the maximum or 0 if no data has been added
*/
     public double getMaximum()
     {
     return maximum;
     }
}
```

Modified DataSet : extend service for to find the bank account with the highest balance



```
public class DataSet // Modified for BankAccount objects
{
   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;
    }
 }
```

Modified DataSet: Extend service find the coin with the highest value among a set of coins



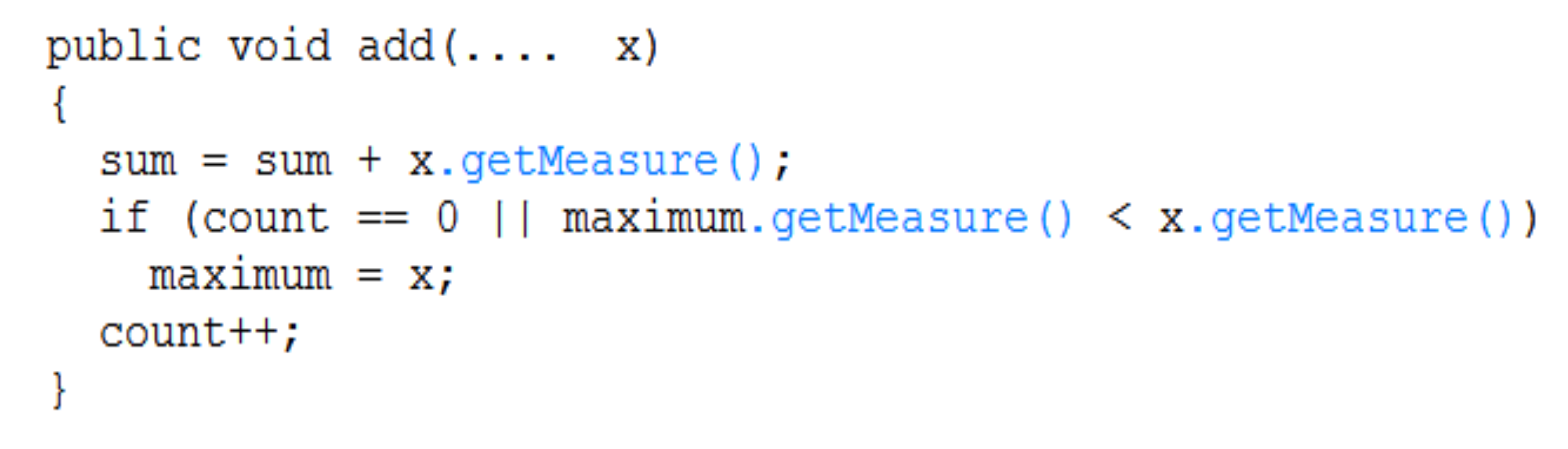
```
public class DataSet // Modified for Coin objects
{
   private double sum;
   private Coin maximum;
   private int count;
   . . .
   public void add(Coin x)
   {
      sum = sum + x.getValue();
      if (count == 0 || maximum.getValue() < x.getValue())
         maximum = x;
      count++;
   }

   public Coin getMaximum()
   {
      return maximum;
   }
}
```

Observation

- The algorithm for the data analysis service is the same in all cases, but the details of measurement differ.
- It is best to provide a *single* class that provides this service to any objects that can be measured.
- Suppose all classes agree to use method *getMeasure()*
- For bank accounts, *getMeasure* returns the *balance*.
- For coins, *getMeasure* returns the coin *value*, etc.

Modified DataSet

- DataSet class whose add method looks like this:

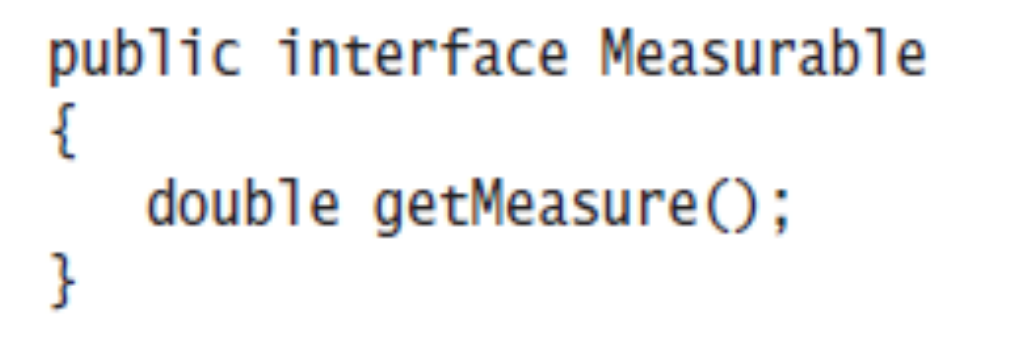


```
public void add(....  x)
{
  sum = sum + x.getMeasure();
  if (count == 0 || maximum.getMeasure() < x.getMeasure())
    maximum = x;
  count++;
}
```

- What type is variable x?
- x should refer to *any class* that has a *getMeasure* method.

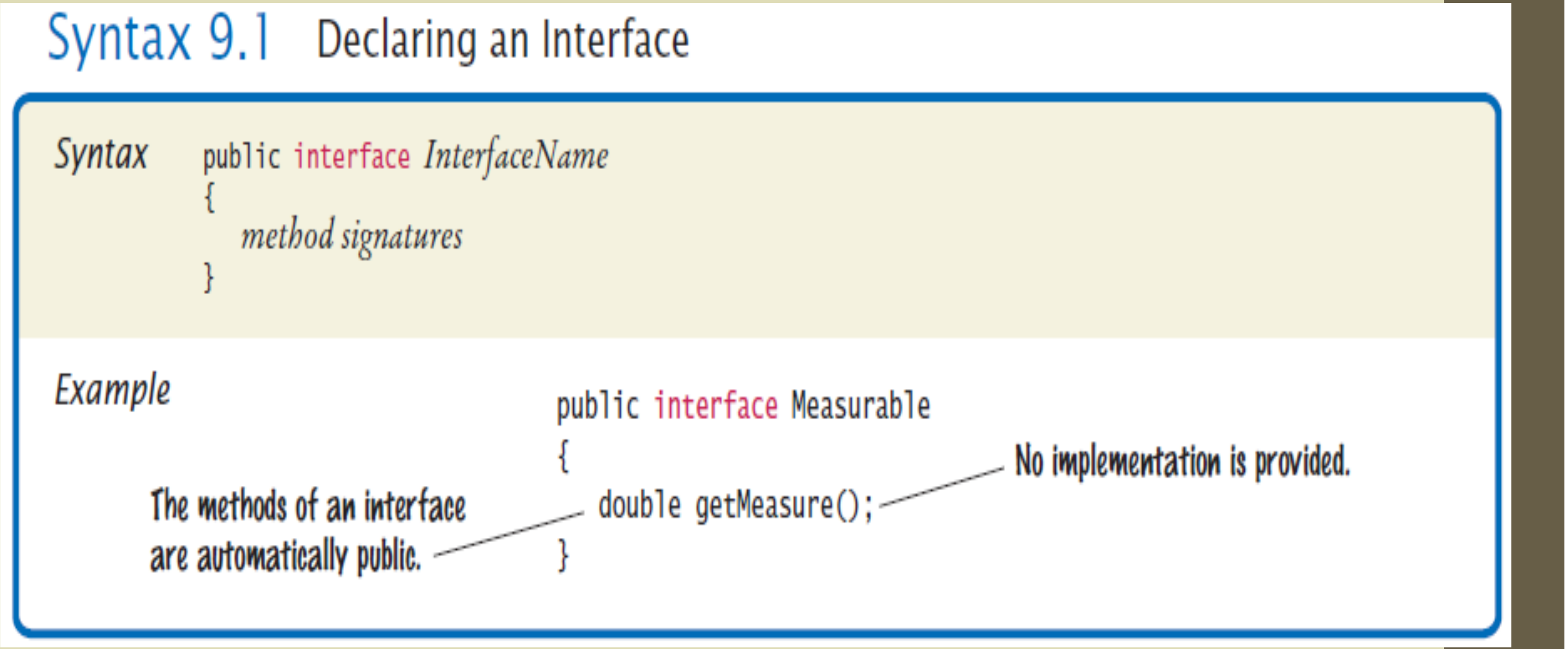
Measurable interface

- **interface type** is used to specify required operations. We will declare an interface type that we call *Measurable*:



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

- A Java interface type declares methods but does not provide their implementations.
- All methods declared in *interface type* are public



Syntax 9.1 Declaring an Interface

Syntax

```
public interface InterfaceName
{
   method signatures
}
```

Example

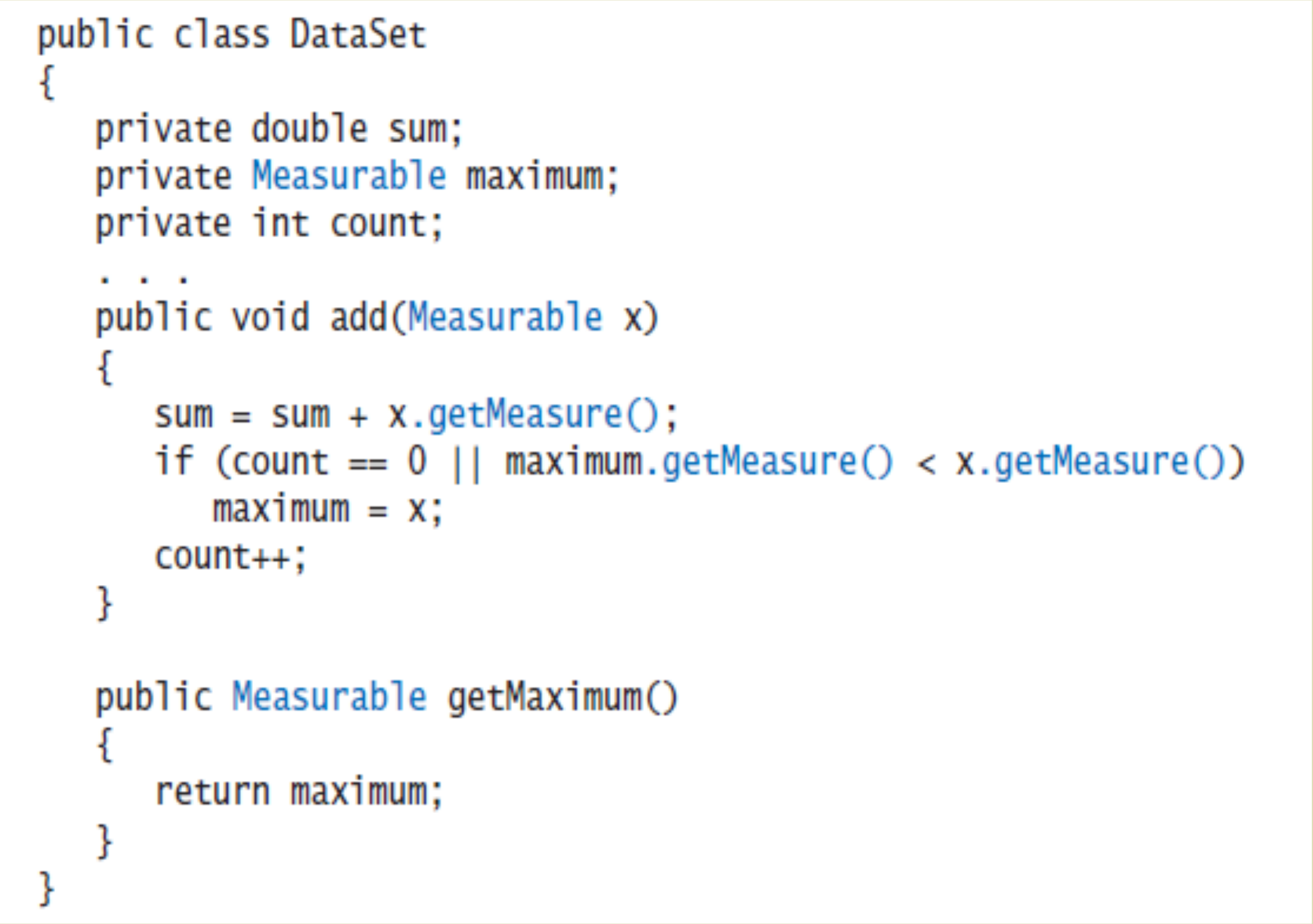
```
public interface Measurable
{
   double getMeasure();
}
```

The methods of an interface are automatically public.

No implementation is provided.

- An interface type is similar to a class, but there are several important differences:
- • All methods in an interface type are *abstract* ;
  - they have a name, parameters, and a return type, but they don’t have an implementation.
- • All methods in an interface type are automatically public.
- • An interface type does not have instance variables.

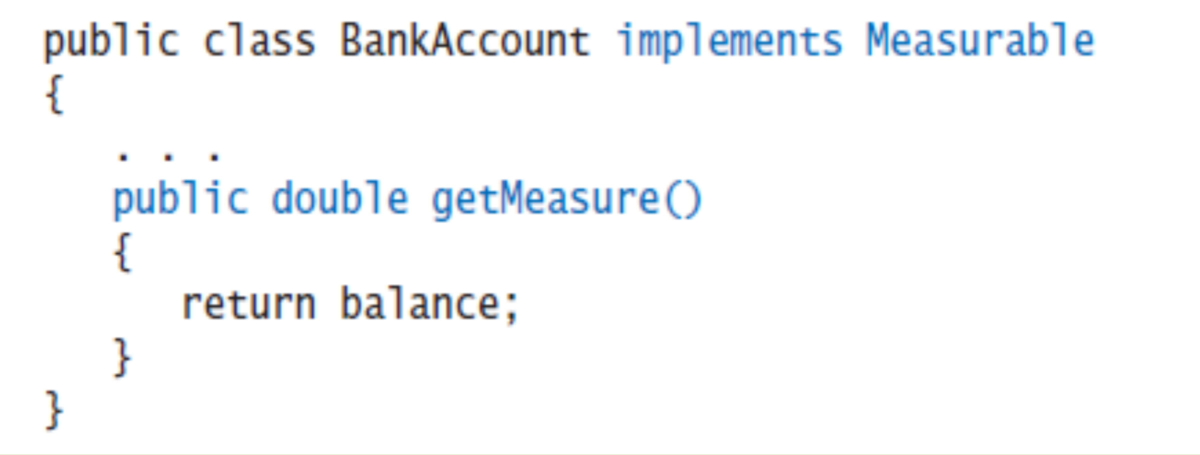
Modified Dataset: using *Measurable* *interface type* to declare the variables *x* and *maximum*



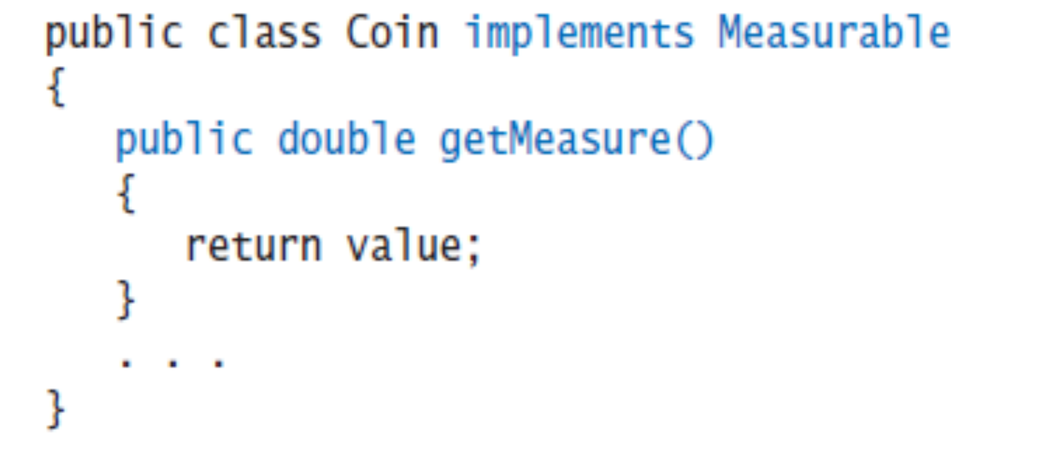
```
public class DataSet
{
   private double sum;
   private Measurable maximum;
   private int count;
   . . .
   public void add(Measurable x)
   {
      sum = sum + x.getMeasure();
      if (count == 0 || maximum.getMeasure() < x.getMeasure())
         maximum = x;
      count++;
   }

   public Measurable getMaximum()
   {
      return maximum;
   }
}
```

Class implements *interface type*

- A class **implements an interface** type if it declares the interface in an implements clause.
- It should then implement the method or methods that the interface requires.



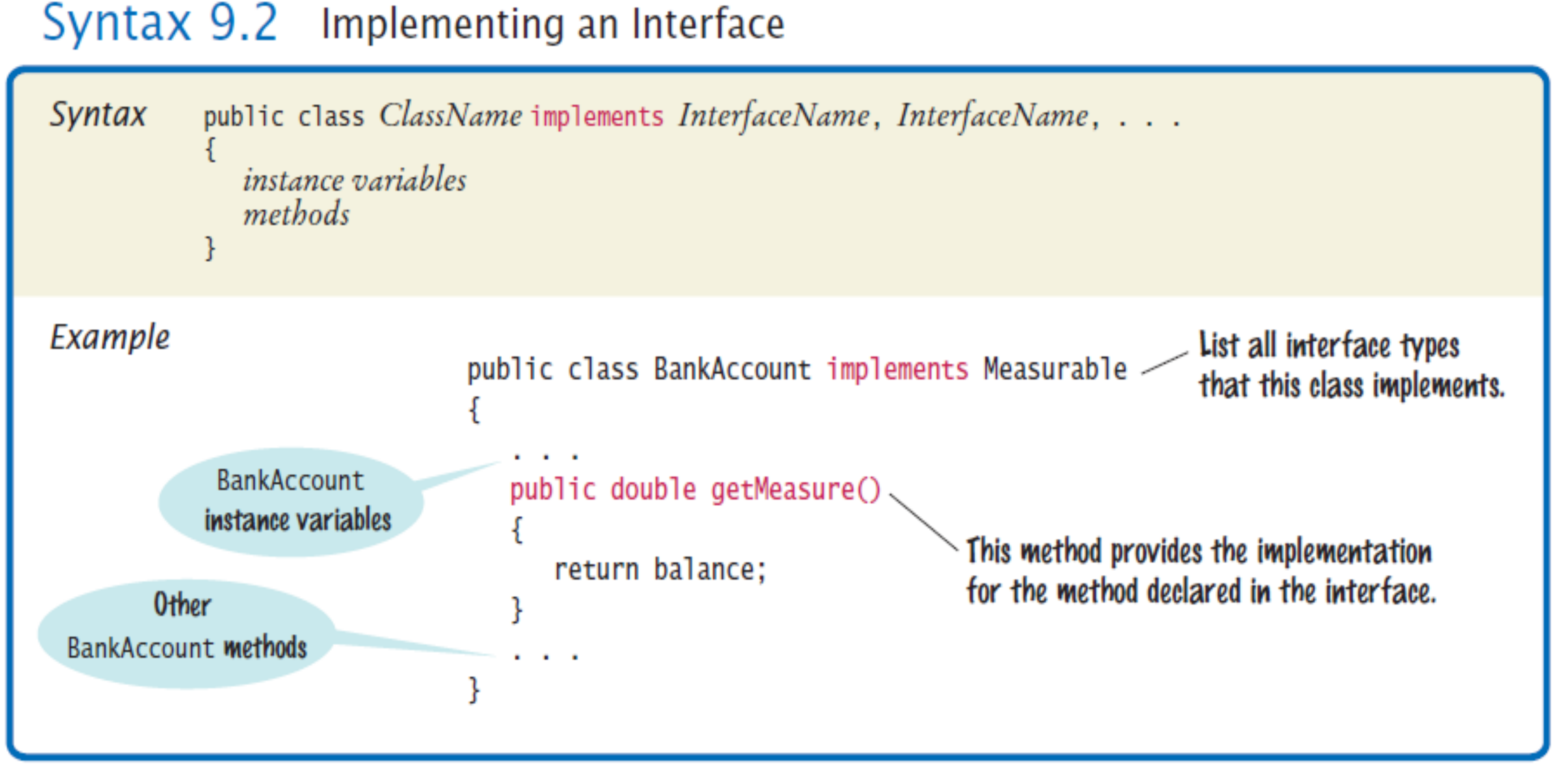
```
public class BankAccount implements Measurable
{
   . . .
   public double getMeasure()
   {
      return balance;
   }
}
```



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

Summary

- Note that the class must declare the method as public, whereas the interface need not—all methods in an interface are public.
- the *Measurable* interface expresses what all measurable objects have in common. This commonality makes the flexibility of the improved *DataSet* class possible.
- A data set can analyze objects of *any* class that implements the *Measurable* interface.
- Use interface types to make code more reusable.



Syntax 9.2 Implementing an Interface

Syntax

```
public class ClassName implements InterfaceName, InterfaceName, . . .
{
   instance variables
   methods
}
```

Example

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

List all interface types that this class implements.

BankAccount instance variables

This method provides the implementation for the method declared in the interface.

Other BankAccount methods

UML Diagram representation

- Note that the DataSet class depends only on the Measurable interface. It is decoupled from the BankAccount and Coin classes.

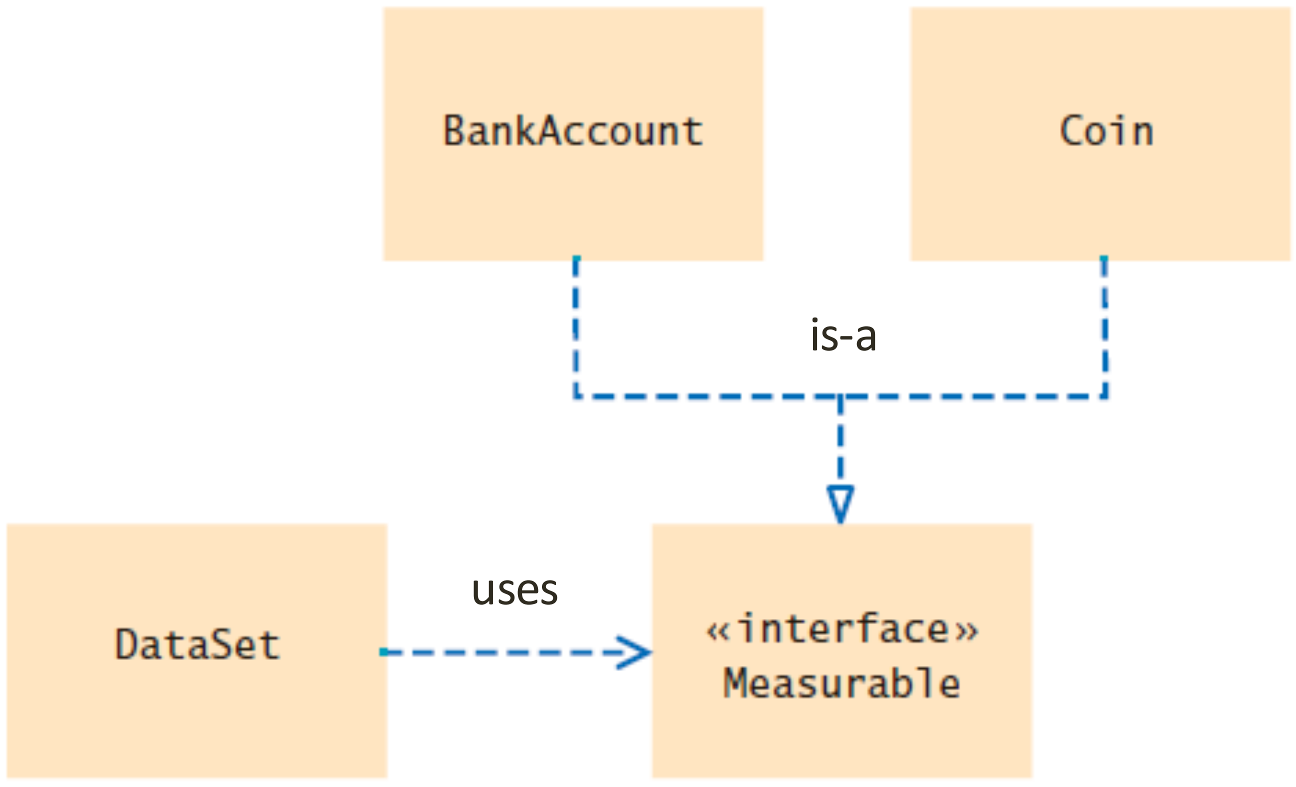
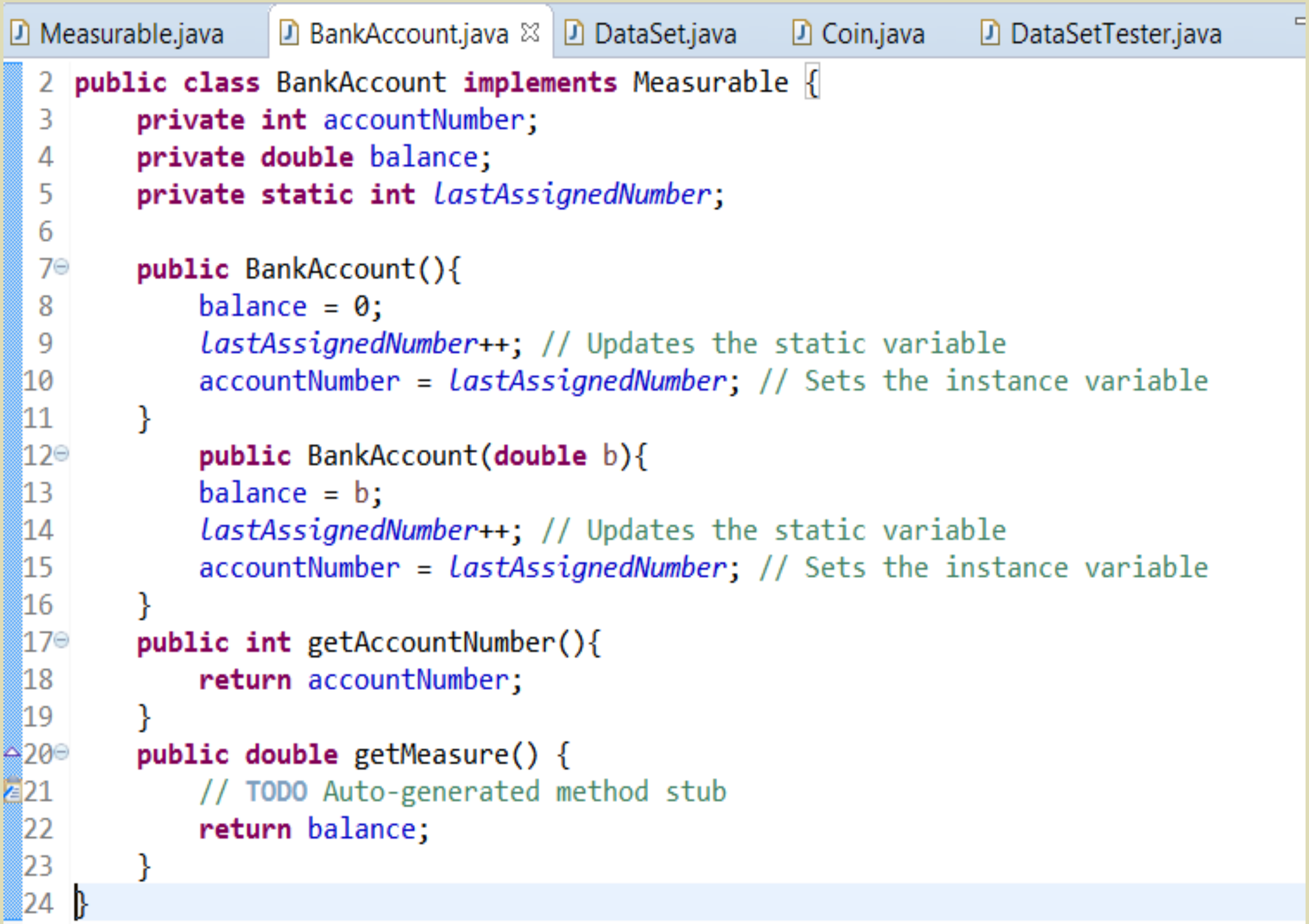


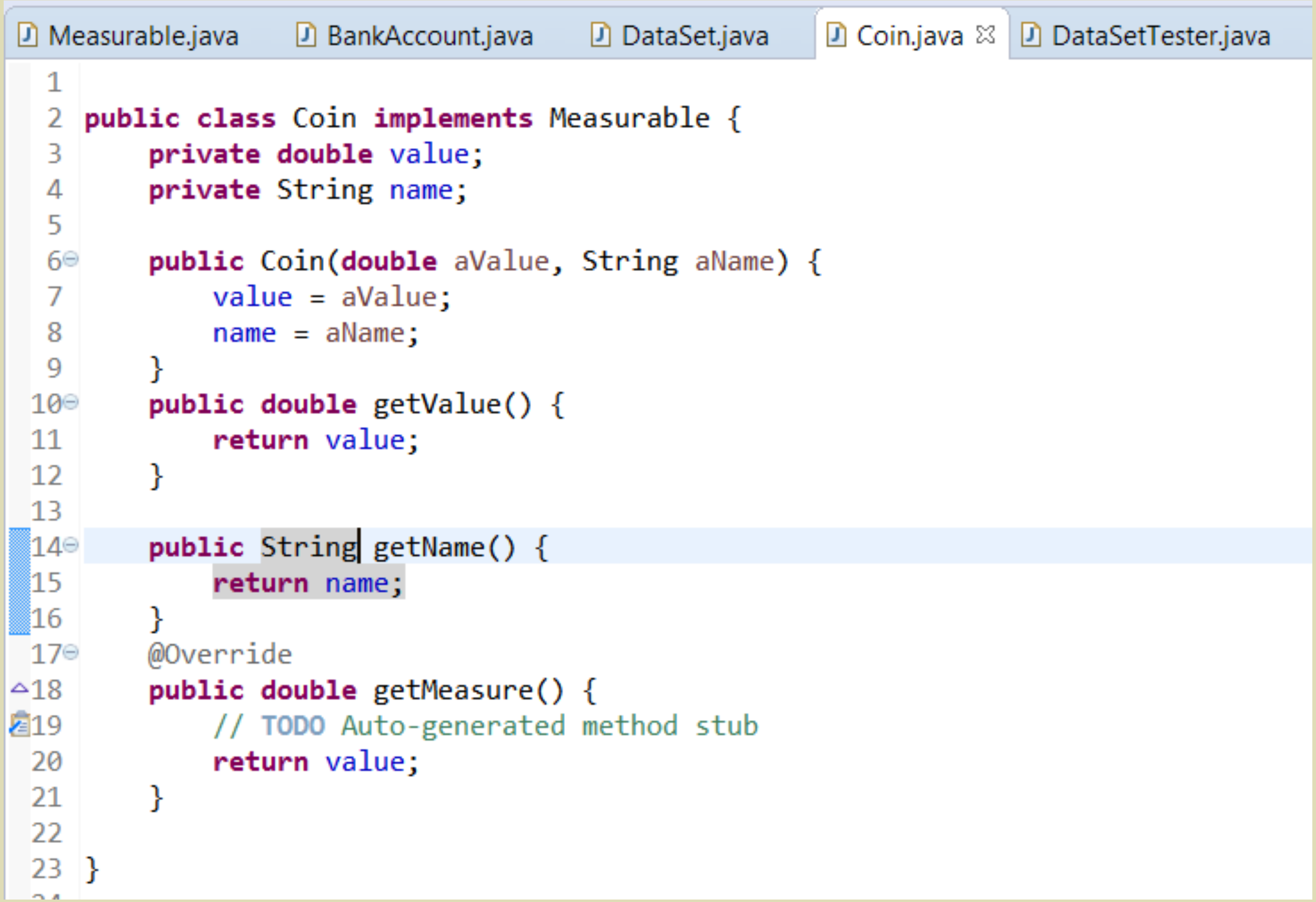
Figure 2 UML Diagram of the DataSet Class and the Classes that Implement the Measurable Interface

Class BankAccount



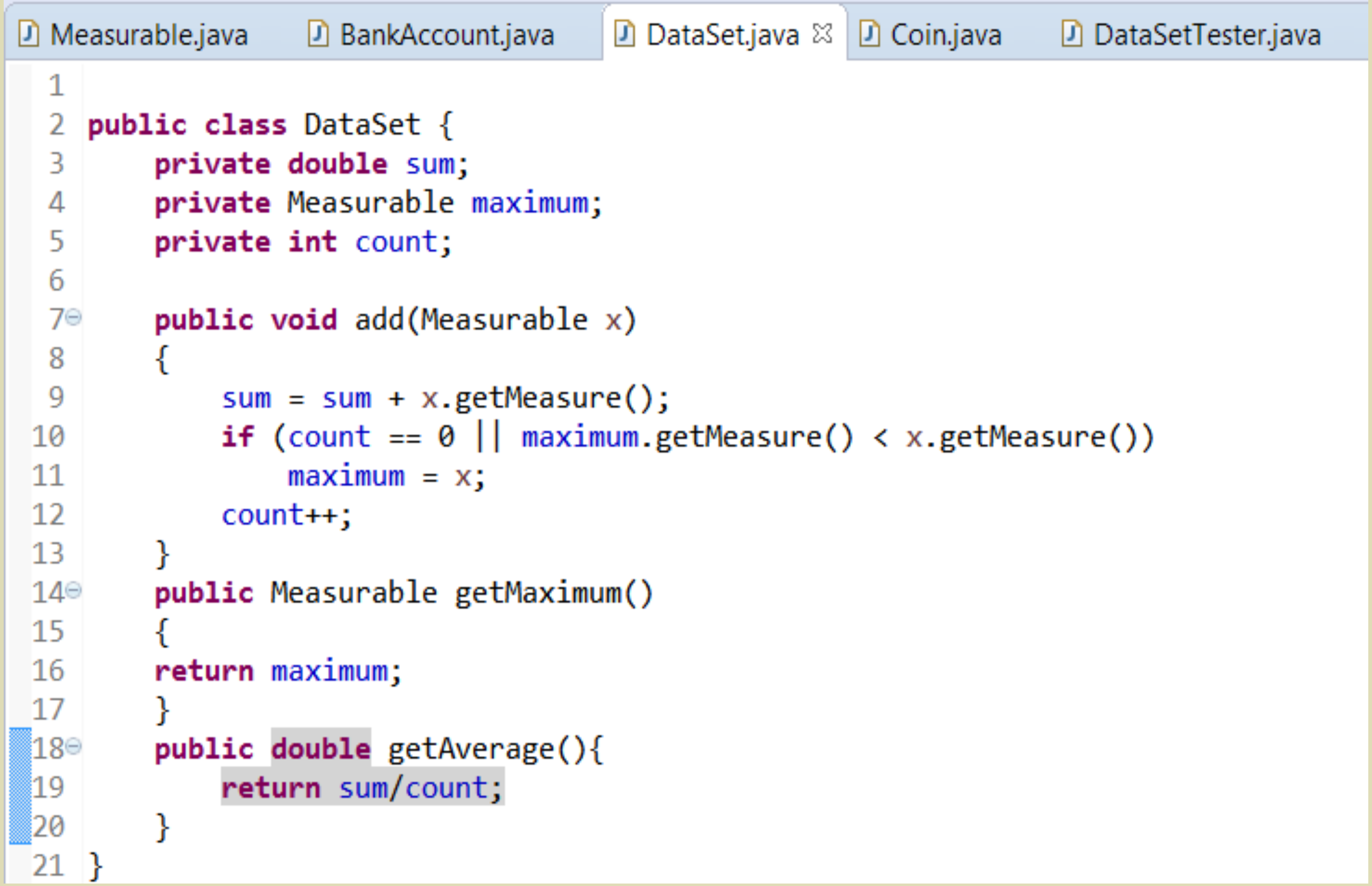
```java
public class BankAccount implements Measurable {
    private int accountNumber;
    private double balance;
    private static int lastAssignedNumber;

    public BankAccount(){
        balance = 0;
        lastAssignedNumber++; // Updates the static variable
        accountNumber = lastAssignedNumber; // Sets the instance variable
    }
        public BankAccount(double b){
        balance = b;
        lastAssignedNumber++; // Updates the static variable
        accountNumber = lastAssignedNumber; // Sets the instance variable
    }
    public int getAccountNumber(){
        return accountNumber;
    }
    public double getMeasure() {
        // TODO Auto-generated method stub
        return balance;
    }
}
```

Class Coin



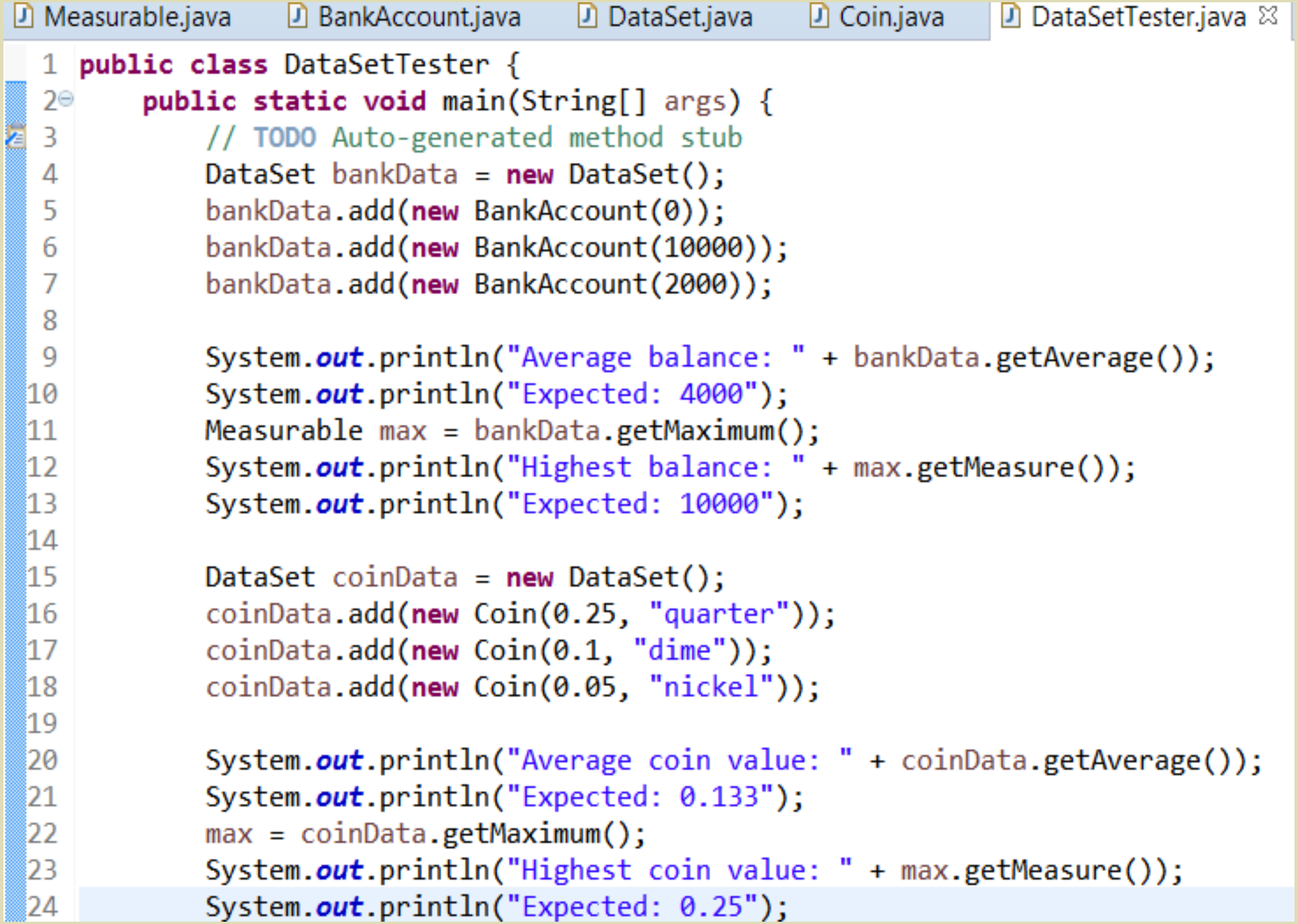
```java

public class Coin implements Measurable {
    private double value;
    private String name;

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

    public String getName() {
        return name;
    }
    @Override
    public double getMeasure() {
        // TODO Auto-generated method stub
        return value;
    }

}
```

Class DataSet



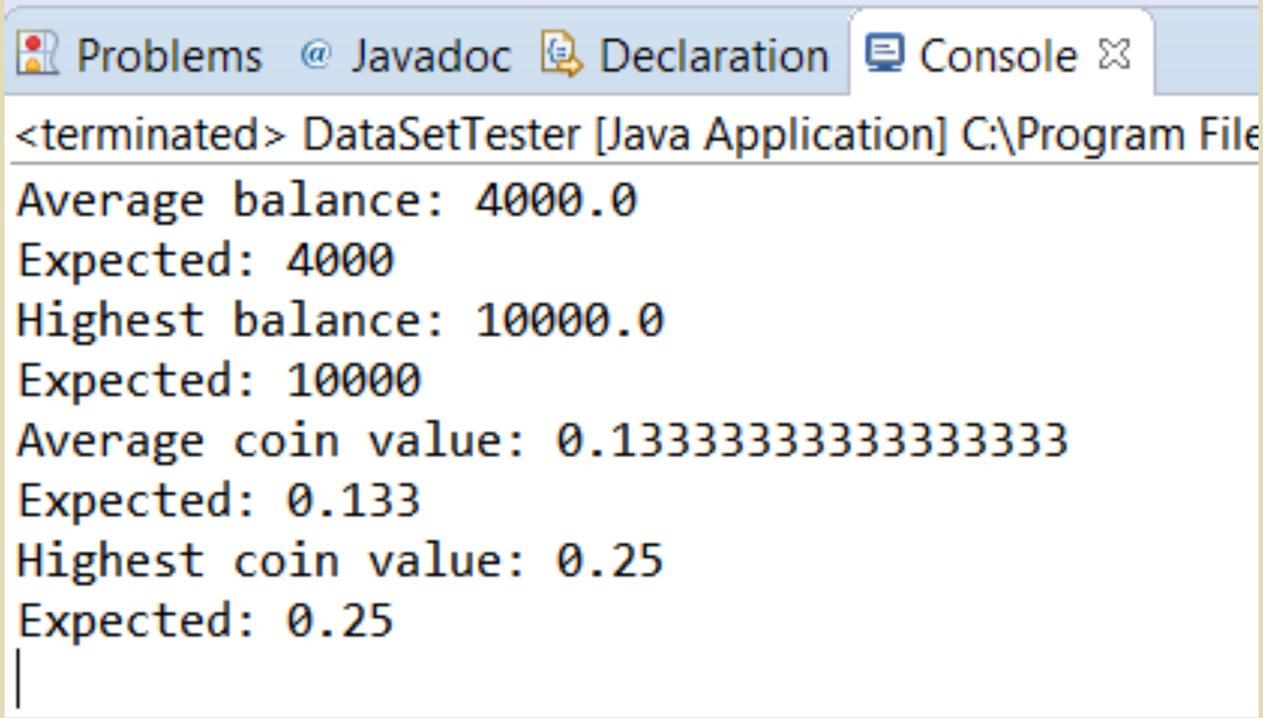
```java

public class DataSet {
    private double sum;
    private Measurable maximum;
    private int count;

    public void add(Measurable x)
    {
        sum = sum + x.getMeasure();
        if (count == 0 || maximum.getMeasure() < x.getMeasure())
            maximum = x;
        count++;
    }
    public Measurable getMaximum()
    {
    return maximum;
    }
    public double getAverage(){
        return sum/count;
    }
}
```

DataSetTester



```java
public class DataSetTester {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        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());
        System.out.println("Expected: 4000");
        Measurable max = bankData.getMaximum();
        System.out.println("Highest balance: " + max.getMeasure());
        System.out.println("Expected: 10000");

        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());
        System.out.println("Expected: 0.133");
        max = coinData.getMaximum();
        System.out.println("Highest coin value: " + max.getMeasure());
        System.out.println("Expected: 0.25");
```

Run DataSetTester



```
<terminated> DataSetTester [Java Application] C:\Program File
Average balance: 4000.0
Expected: 4000
Highest balance: 10000.0
Expected: 10000
Average coin value: 0.13333333333333333
Expected: 0.133
Highest coin value: 0.25
Expected: 0.25
```

Constants in Interfaces

- Interfaces cannot have instance variables, but it is legal to specify *constants*.
- All variables in an interface are automatically *public static final*

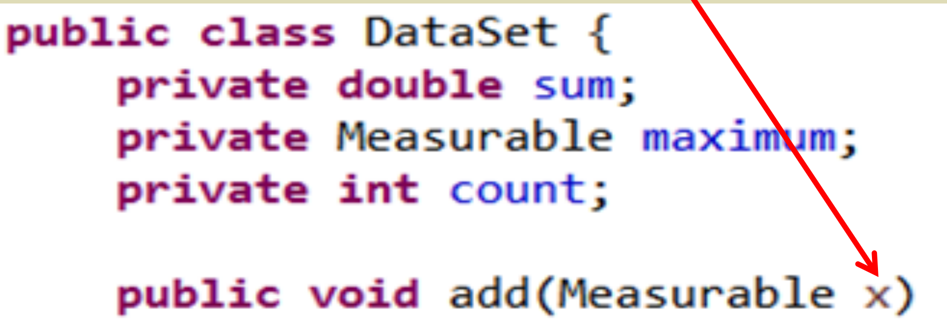
```
public interface SwingConstants
{
int NORTH = 1;
int NORTHEAST = 2;
int EAST = 3;
. . .
}
```

Converting Between Class and Interface Types

- Convert from a class type to an interface type is allowed,provided the class implements the interface.

bankData.add(new BankAccount(1000)); //similar to

BankAccount account = new BankAccount(1000);
Measurable meas = account; // OK



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

    public void add(Measurable x)
```

Coin dime = new Coin(0.1, "dime");

Measurable meas = dime; // Also OK coz Coin implements Measurable

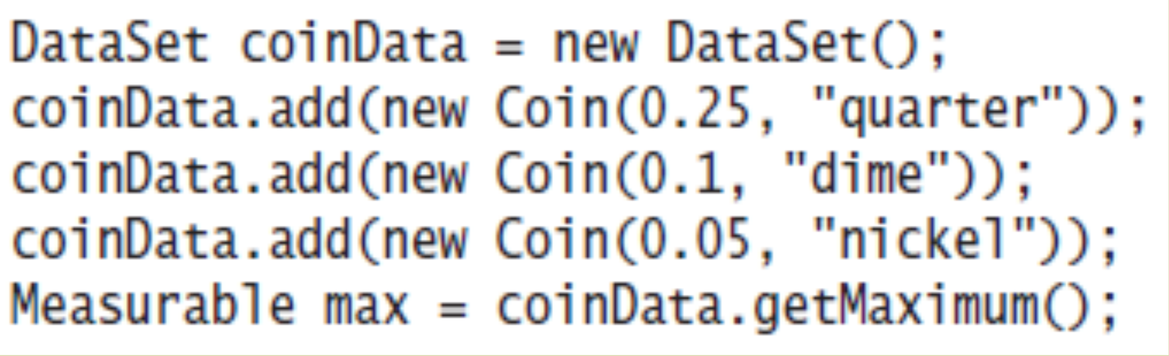
- meas can only call getMeasure() method

Measurable meas = new Rectangle(5, 10, 20, 30); // Error

- Class Rectangle from Java Standard library does not implements the Measurable interface

Converting Between Class and Interface Types

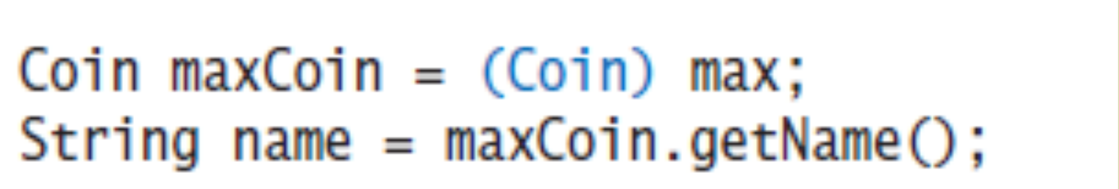
- Occasionally, it happens that you store an object in an interface reference and you need to convert its type back.
- This happens in the *getMaximum* method of the DataSet class. The DataSet stores the object with the largest measure, *as a* Measurable *reference*.



```
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"));
Measurable max = coinData.getMaximum();
```

max.getName(); //Error

- Knowing that var *max* is a refering to Coin object, we can use **Cast notation** to convert max to Coin type



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

Common Error

- You can declare variables whose type is an interface, for example:

```
        Measurable meas;
```

- However, you can *never* construct an object of an interface type:

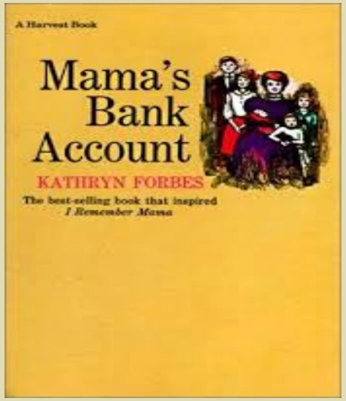
```
        Measurable meas = new Measurable(); // Error
```

Polymorphism

```
Measurable meas;
```

- Just remember that the object to which *meas* refers doesn't have type *Measurable*. Instead, the type of the object is some class that implements the *Measurable* interface.
- It can be object of class BankAccount @ Coin @ any other class with *getMeasure()* method

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



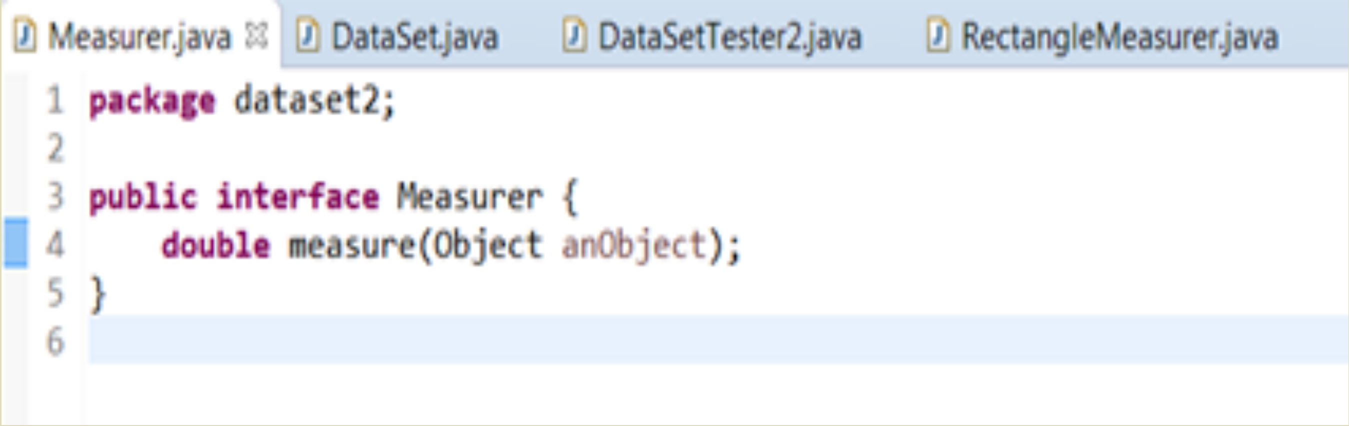
- Depends on the momentary contents of meas. This mechanism for locating the appropriate method is called *dynamic method lookup*

Using Interfaces for Callbacks

- A callback is a mechanism for specifying code that is executed at a later time.
- The data set needs to measure the objects that are added. When the objects are required to be of type Measurable, the responsibility of measuring lies with the added objects themselves
- It would be better if we could give a method for measuring objects to a data set.
- When collecting rectangles, we might give it a method for computing the area of a rectangle. When collecting savings accounts, we might give it a method for getting the account's interest rate.
- Such method is called Callback. Java is an object-oriented programming language. Therefore, we turn callbacks into objects.

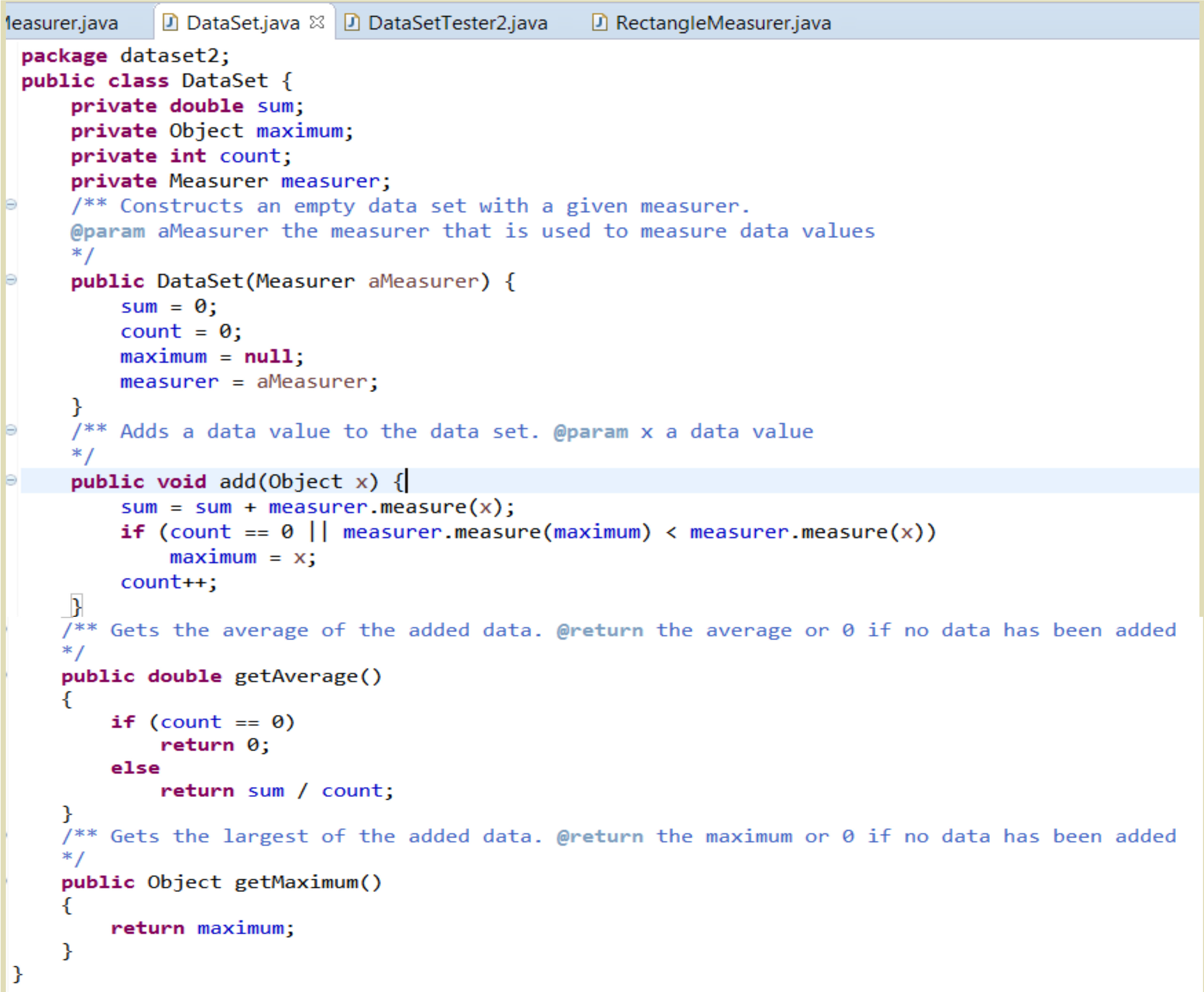
Callback Object

- This process starts by declaring an interface for the callback:

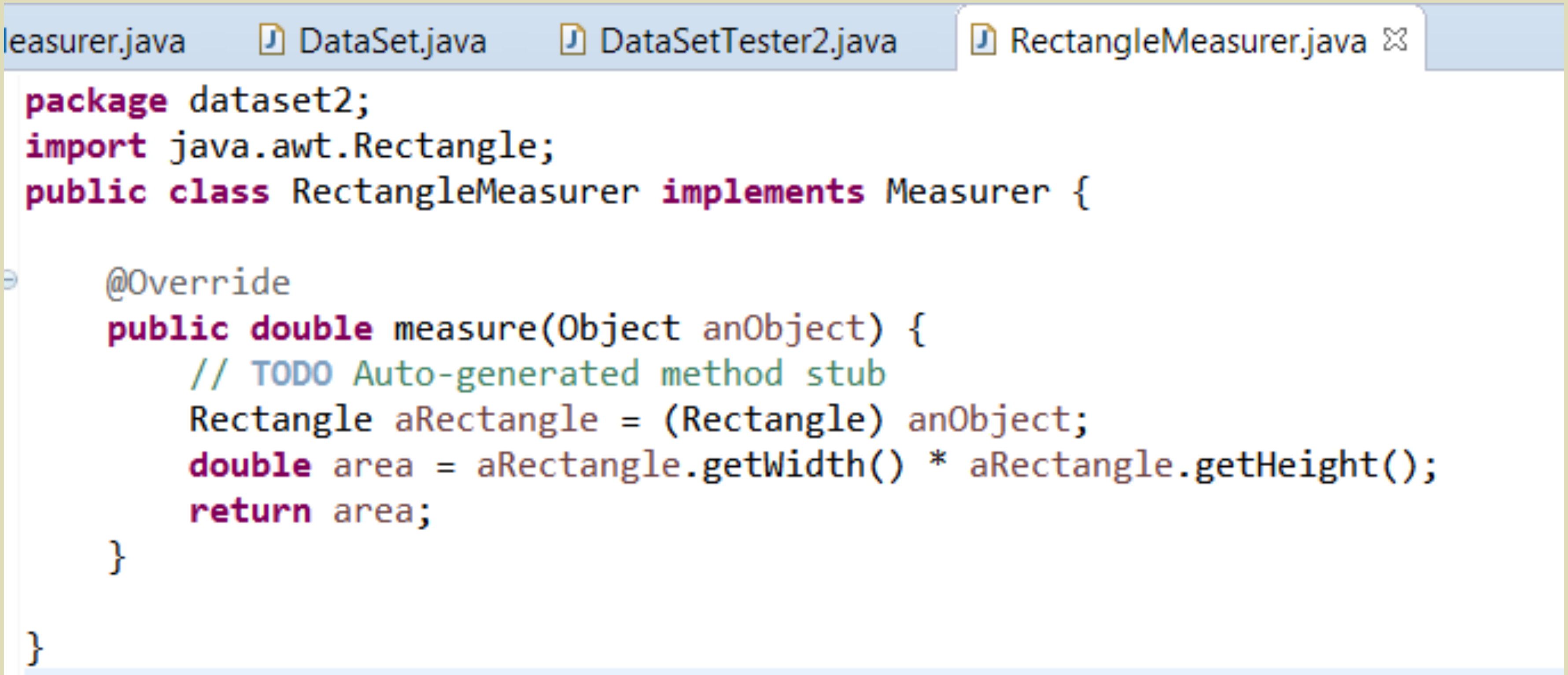


```
package dataset2;

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

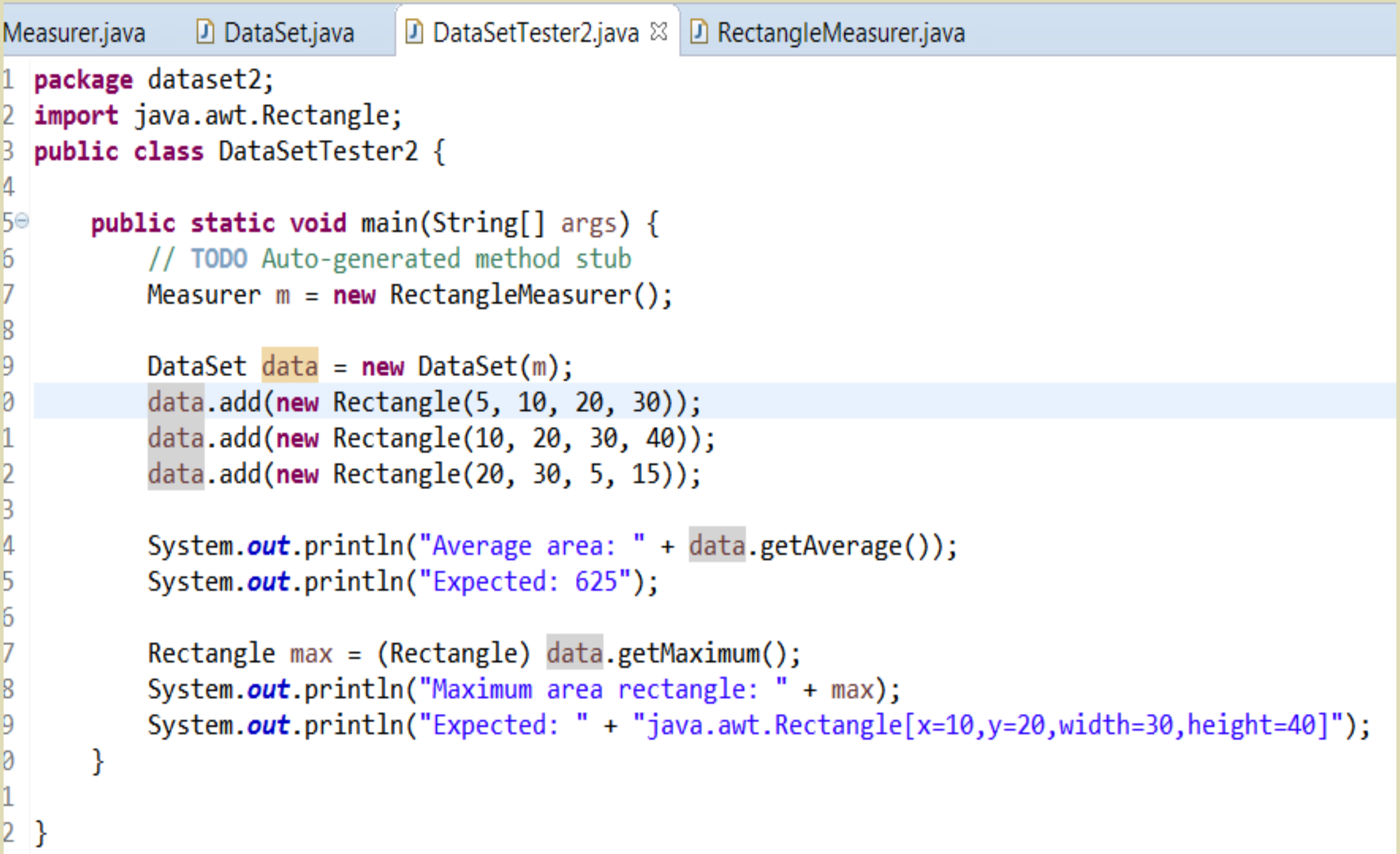
- The measure method measures an object and returns its measurement. Here we use the fact that all objects can be converted to the type Object



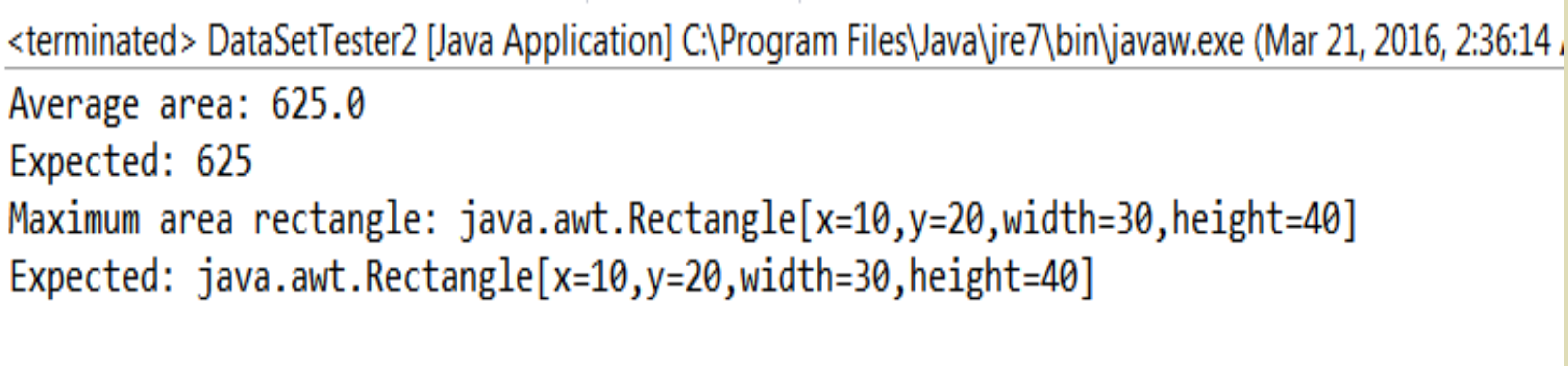
```java
package dataset2;
public class DataSet {
    private double sum;
    private Object maximum;
    private int count;
    private Measurer measurer;
    /** Constructs an empty data set with a given measurer.
    @param aMeasurer the measurer that is used to measure data values
    */
    public DataSet(Measurer aMeasurer) {
        sum = 0;
        count = 0;
        maximum = null;
        measurer = aMeasurer;
    }
    /** Adds a data value to the data set. @param x a data value
    */
    public void add(Object x) {
        sum = sum + measurer.measure(x);
        if (count == 0 || measurer.measure(maximum) < measurer.measure(x))
            maximum = x;
        count++;
    }
    /** Gets the average of the added data. @return the average or 0 if no data has been added
    */
    public double getAverage()
    {
        if (count == 0)
            return 0;
        else
            return sum / count;
    }
    /** Gets the largest of the added data. @return the maximum or 0 if no data has been added
    */
    public Object getMaximum()
    {
        return maximum;
    }
}
```



```java
package dataset2;
import java.awt.Rectangle;
public class RectangleMeasurer implements Measurer {

    @Override
    public double measure(Object anObject) {
        // TODO Auto-generated method stub
        Rectangle aRectangle = (Rectangle) anObject;
        double area = aRectangle.getWidth() * aRectangle.getHeight();
        return area;
    }

}
```



```java
package dataset2;
import java.awt.Rectangle;
public class DataSetTester2 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        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, 15));

        System.out.println("Average area: " + data.getAverage());
        System.out.println("Expected: 625");

        Rectangle max = (Rectangle) data.getMaximum();
        System.out.println("Maximum area rectangle: " + max);
        System.out.println("Expected: " + "java.awt.Rectangle[x=10,y=20,width=30,height=40]");
    }

}
```



```
<terminated> DataSetTester2 [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Mar 21, 2016, 2:36:14
Average area: 625.0
Expected: 625
Maximum area rectangle: java.awt.Rectangle[x=10,y=20,width=30,height=40]
Expected: java.awt.Rectangle[x=10,y=20,width=30,height=40]
```

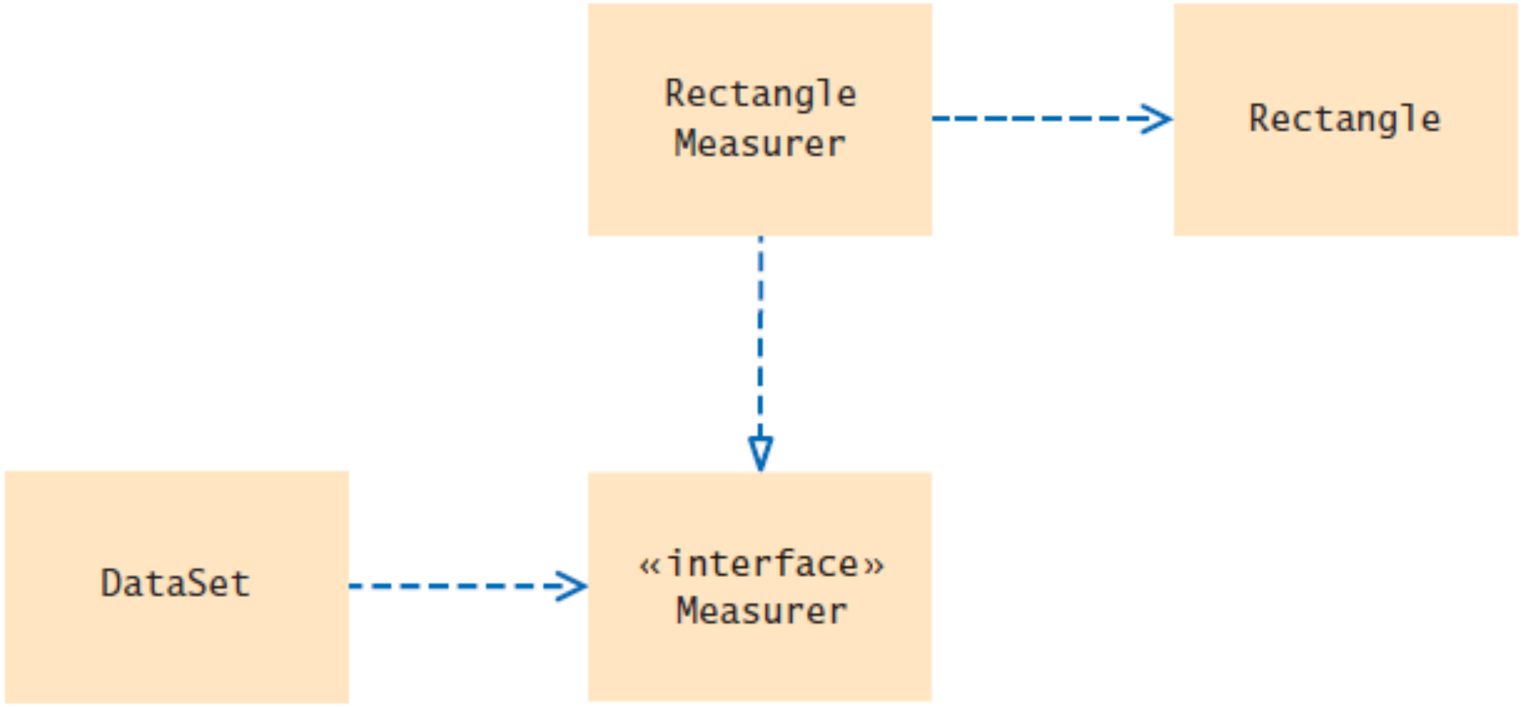
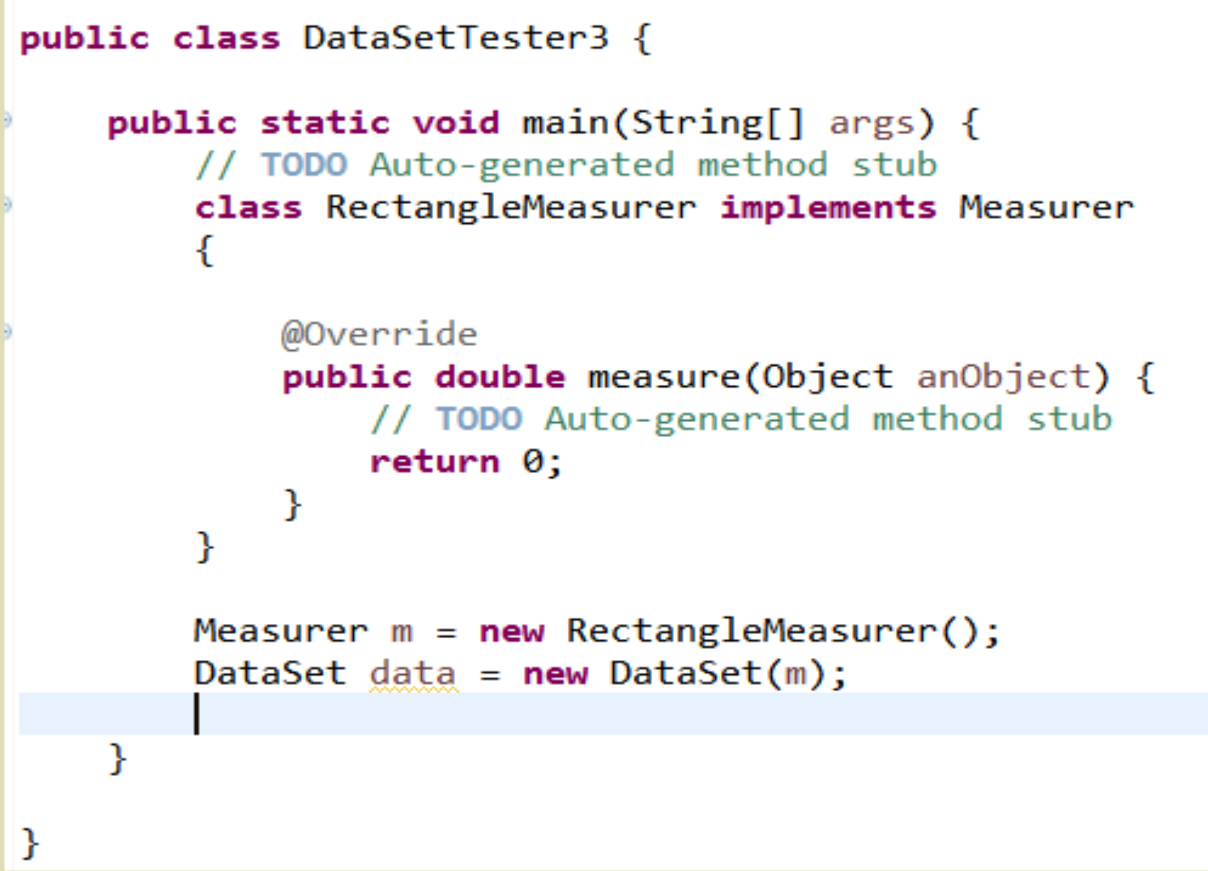


Figure 5 UML Diagram of the DataSet Class and the Measurer Interface

Inner Classes

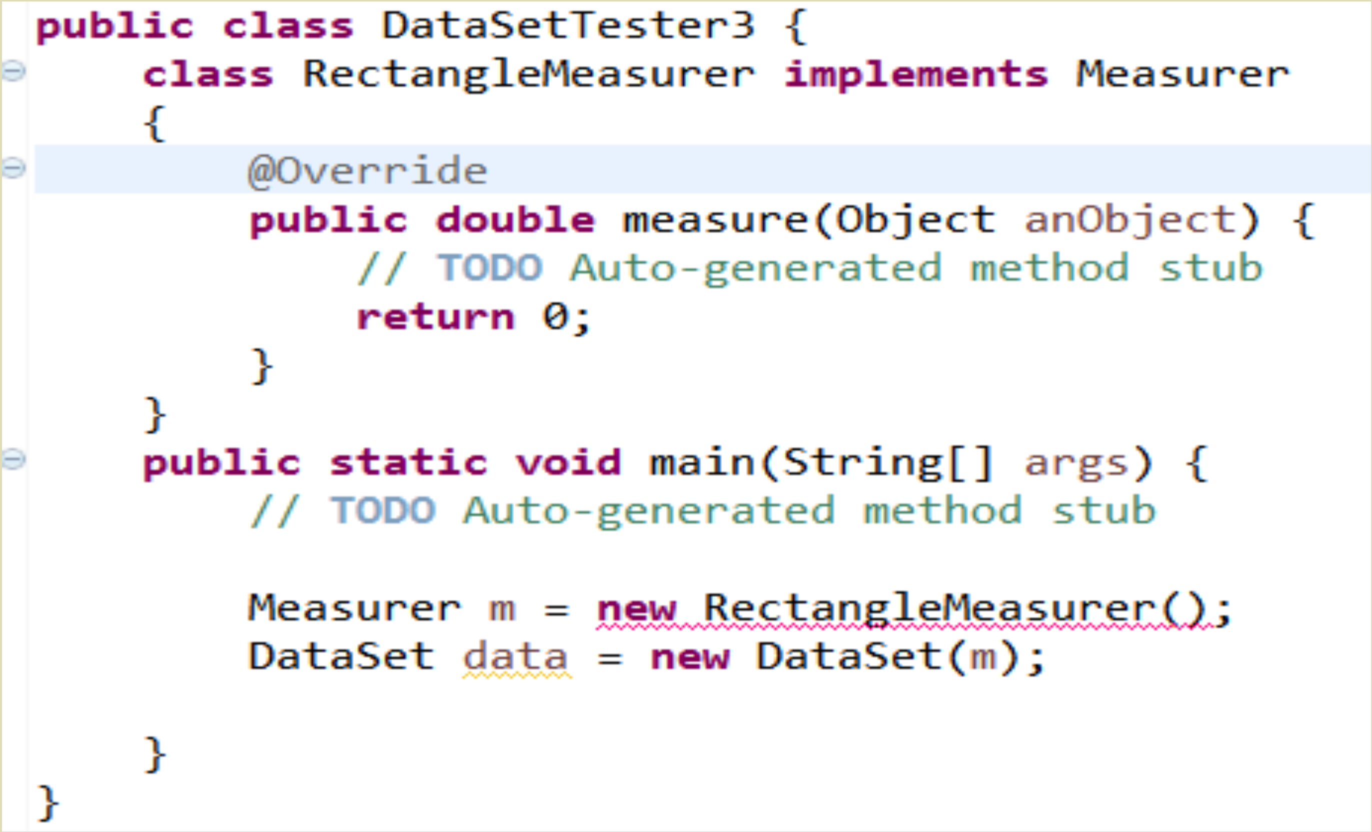
- The RectangleMeasurer class is a very trivial class. We need this class only because the DataSet class needs an object of some class that implements the Measurer interface.
- When you have a class that serves a very limited purpose, such as this one, you can declare the class ***inside*** the method that needs it:



```
public class DataSetTester3 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        class RectangleMeasurer implements Measurer
        {

            @Override
            public double measure(Object anObject) {
                // TODO Auto-generated method stub
                return 0;
            }
        }

        Measurer m = new RectangleMeasurer();
        DataSet data = new DataSet(m);

    }

}
```

Inner Class

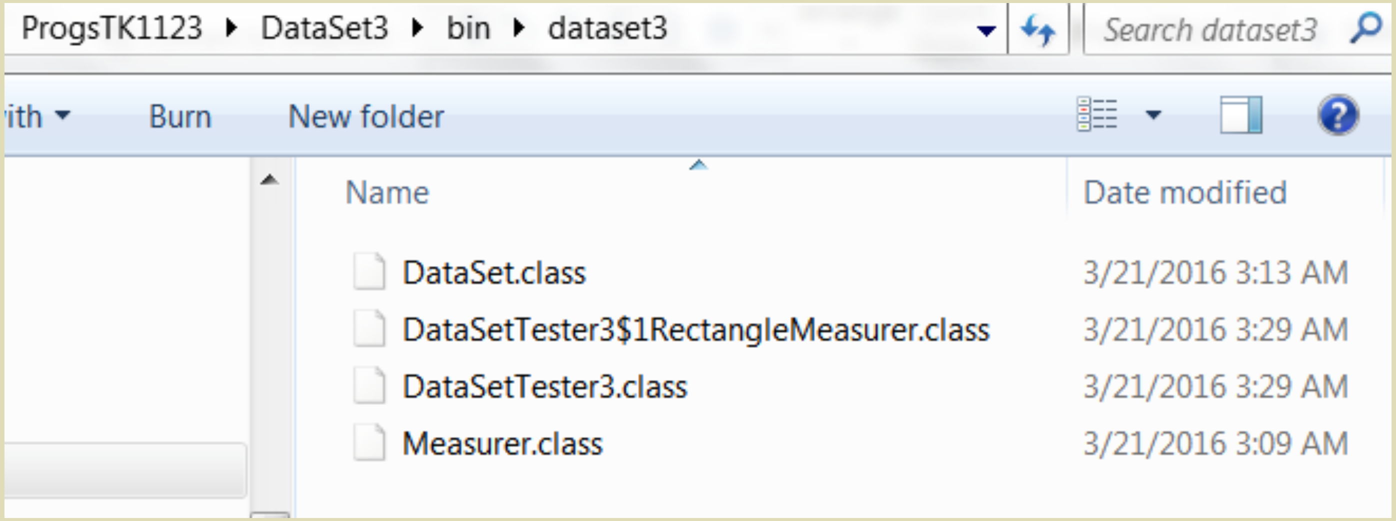
- You can also declare an inner class inside an enclosing class, but outside of its methods. Then the inner class is available to all methods of the enclosing class.



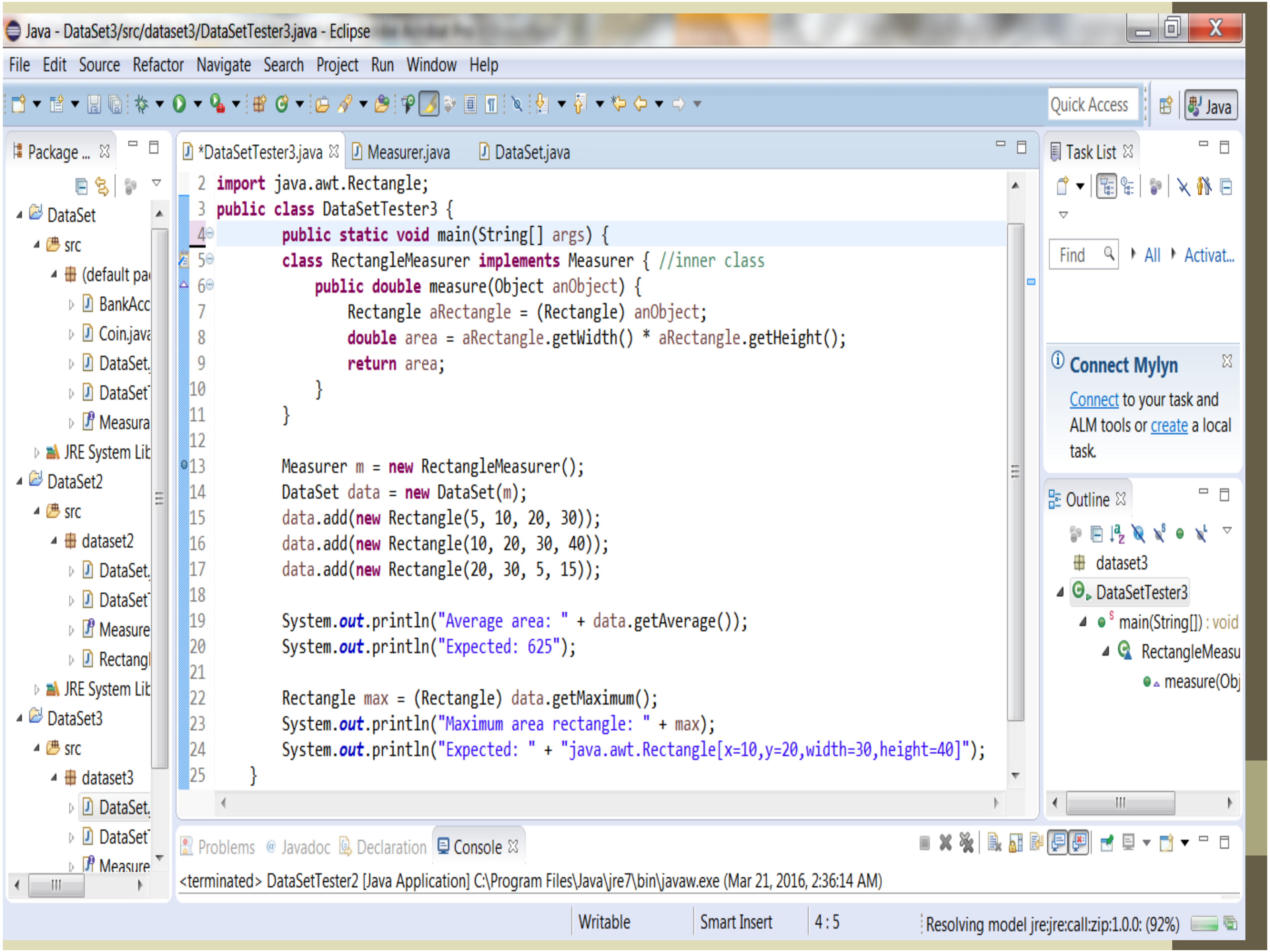
```
public class DataSetTester3 {
    class RectangleMeasurer implements Measurer
    {
        @Override
        public double measure(Object anObject) {
            // TODO Auto-generated method stub
            return 0;
        }
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub

        Measurer m = new RectangleMeasurer();
        DataSet data = new DataSet(m);

    }
}
```

Inner Class

- When you compile the source files for a program that uses inner classes, have a look at the class files in your program directory—you will find that the inner classes are stored in files with curious names, such as DataSetTester3$1RectangleMeasurer.class.
- The exact names aren’t important. The point is that the compiler turns an inner class into a regular class file.



Name	Date modified
DataSet.class	3/21/2016 3:13 AM
DataSetTester3$1RectangleMeasurer.class	3/21/2016 3:29 AM
DataSetTester3.class	3/21/2016 3:29 AM
Measurer.class	3/21/2016 3:09 AM



```
import java.awt.Rectangle;
public class DataSetTester3 {
        public static void main(String[] args) {
        class RectangleMeasurer implements Measurer { //inner class
            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));
        data.add(new Rectangle(20, 30, 5, 15));

        System.out.println("Average area: " + data.getAverage());
        System.out.println("Expected: 625");

        Rectangle max = (Rectangle) data.getMaximum();
        System.out.println("Maximum area rectangle: " + max);
        System.out.println("Expected: " + "java.awt.Rectangle[x=10,y=20,width=30,height=40]");
    }
```

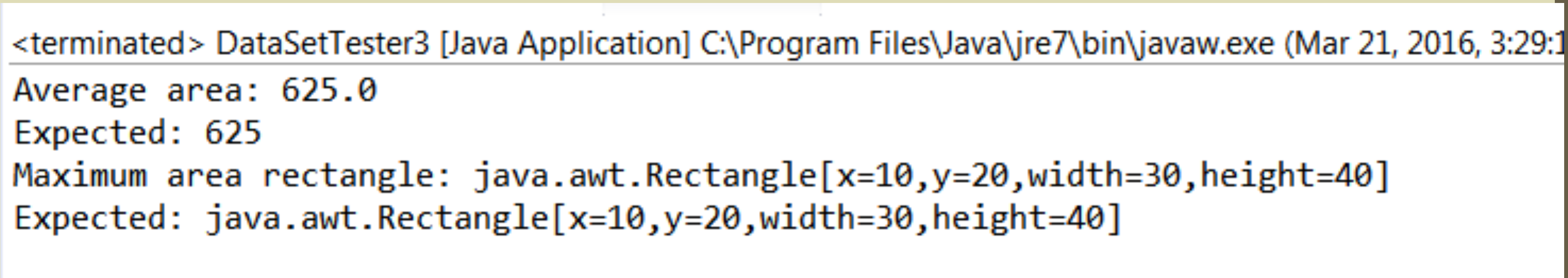
Connect Mylyn

Connect to your task and ALM tools or create a local task.

```
<terminated> DataSetTester2 [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Mar 21, 2016, 2:36:14 AM)
```

Writable | Smart Insert | 4 : 5 | Resolving model jre:jre:call:zip:1.0.0: (92%)

Run DataSetTester3



<terminated> DataSetTester3 [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Mar 21, 2016, 3:29:1

```
Average area: 625.0
Expected: 625
Maximum area rectangle: java.awt.Rectangle[x=10,y=20,width=30,height=40]
Expected: java.awt.Rectangle[x=10,y=20,width=30,height=40]
```