starting out with >>>

JAVA™ EARLY OBJECTS

FIFTH EDITION

CHAPTER 3

A First Look at Classes and Objects

TONY GADDIS

Addison-Wesley
is an imprint of
PEARSON

Topics

- Classes
- More about Passing Arguments
- Instance Fields and Methods
- Constructors
- A `BankAccount` Class
- Classes, Variables, and Scope
- Packages and `import` Statements
- Focus on Object Oriented Design: Finding the Classes and their Responsibilities

Classes

- **In an object-oriented programming language, like Java, you create programs that are made of objects.**
- **In software, an object has two capabilities:**
  - An object can store data.
  - An object can perform operations.
- **The data stored in an object are commonly called *attributes* or *fields*.**
- **The operations that an object can perform are called *methods*.**

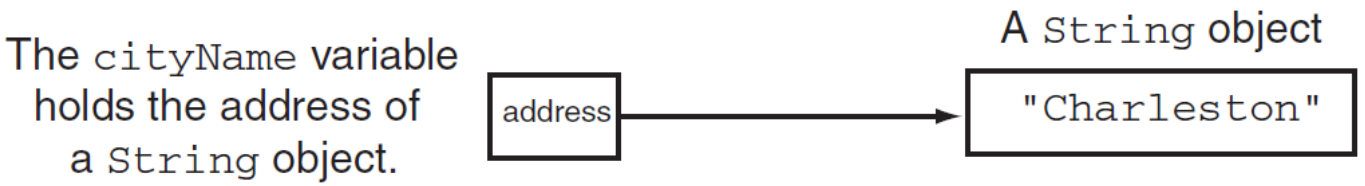
Strings as Objects

- **A primitive data type can only store data, as it has no other built-in capabilities.**
- **An object can store data and perform operations on that data.**
  - In addition to storing strings, `String` objects have numerous methods that perform operations on the strings they hold.

Strings as Objects (cont'd)

- From chapter 2, we learned that a reference variable contains the address of an object.

Figure 3-1 The `cityName` variable references a `String` object



Strings as Objects (cont'd)

- The `length()` method of the `String` class returns and integer value that is equal to the length of the string.

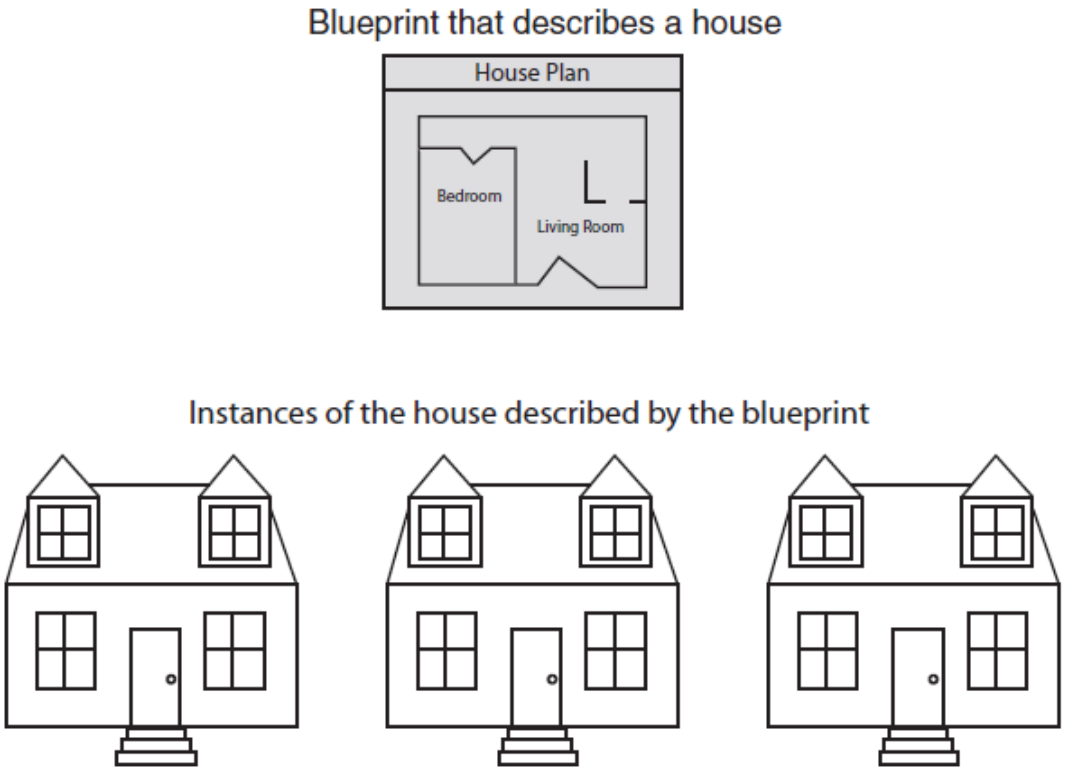
```
int stringLength = cityName.length();
```

- The variable `stringLength` will contain 10 after this statement since the string "Charleston" has 10 characters.
- Primitives can not have methods that can be run whereas objects can.

Classes and Instances

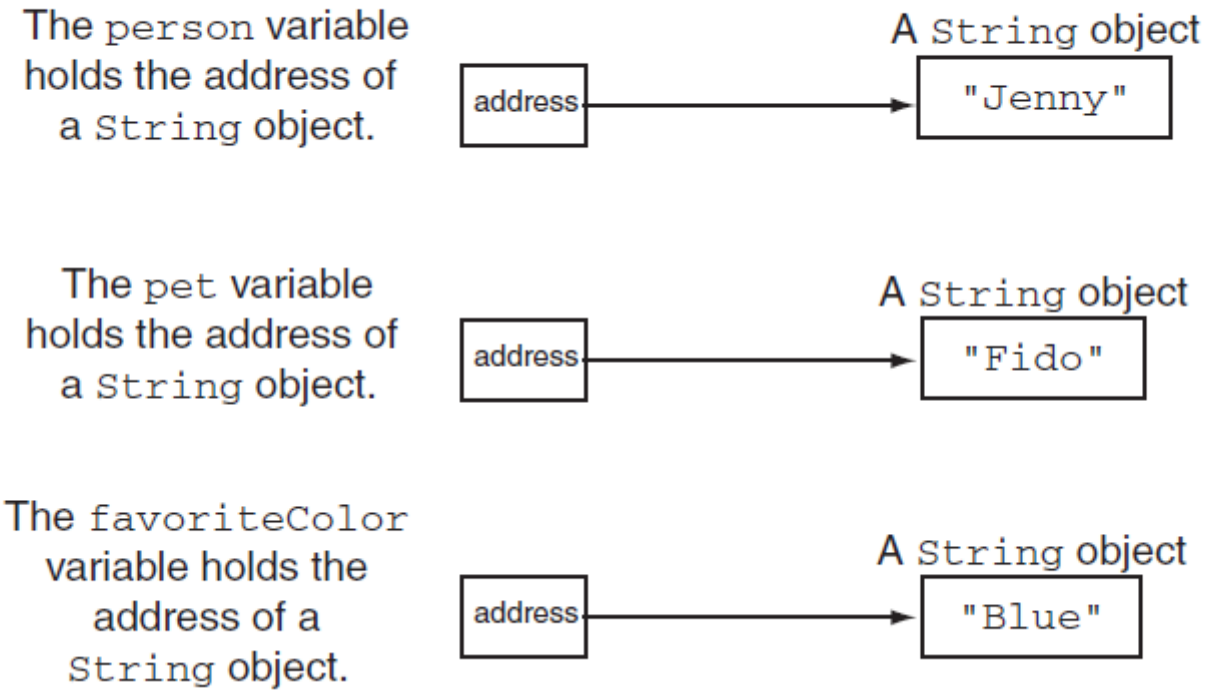
- **Many objects can be created from a class.**
- **Each object is independent of the others.**

Figure 3-2 A blueprint and houses built from the blueprint



Classes and Instances (cont'd)

Figure 3-3 Three variables referencing three `String` objects



Classes and Instances (cont'd)

- Each instance of the `String` class contains different data.
- The instances are all share the same design.
- Each instance has all of the attributes and methods that were defined in the `String` class.
- Classes are defined to represent a single concept or service.

Access Modifiers

- **An access modifier is a Java key word that indicates how a field or method can be accessed.**
- **There are three Java access modifiers:**
  - `public`
  - `private`
  - `protected`

Access Modifiers

- ***public*: This access modifier states that any other class can access the resource.**
- ***private*: This access modifier indicates that only data within this class can access the resource.**
- ***protected*: This modifier indicates that only classes in the current package or a class lower in the class hierarchy can access this resource.**
- **These will be explained in greater detail later.**

Access Modifiers

- **Classes that need to be used by other classes are typically made public.**
- **If there is more than one class in a file, only one may be public and it must match the file name.**
- **Class headers have a format:**

```
AccessModifier class ClassName
{
  Class Members
}
```

Encapsulation

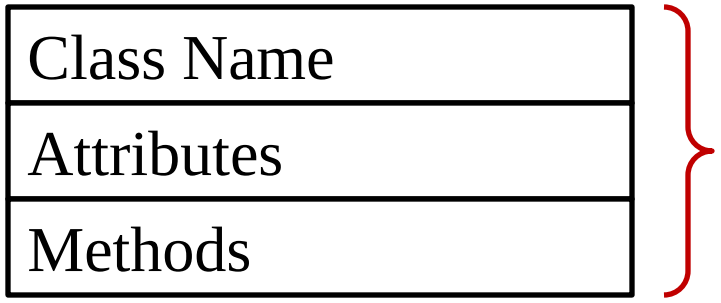
- **Classes should be as limited in scope as needed to accomplish the goal.**
- **Each class should contain all that is needed for it to operate.**
- **Enclosing the proper attributes and methods inside a single class is called *encapsulation*.**
- **Encapsulation ensures that the class is *self-contained*.**

Designing a Class

- **When designing a class, decisions about the following must be made.**
  - what data must be accounted for
  - what actions need to be performed
  - what data can be modified
  - what data needs to be accessible
  - any rules as to how data should be modified
- **Class design typically is done with the aid of a Unified Modeling Language (UML) diagram.**

UML Class Diagram

- **A UML class diagram is a graphical tool that can aid in the design of a class.**
- **The diagram has three main sections.**



UML diagrams are easily converted to Java class files. There will be more about UML diagrams a little later.

- The class name should concisely reflect what the class represents.

Attributes

- **The data elements of a class define the object to be instantiated from the class.**
- **The attributes must be specific to the class and define it completely.**
- **Example: A rectangle is defined by**
  - length
  - width
- **The attributes are then accessed by methods within the class.**

Data Hiding

- Another aspect of encapsulation is the concept of *data hiding*.
- Classes should not only be self-contained but they should be *self-governing* as well.
- Classes use the *private* access modifier on fields to hide them from other classes.
- Classes need methods to allow access and modification of the class' data.

Methods

- **The class' methods define the actions that an instance of the class can perform**
- **Methods headers have a format:**

```
AccessModifier ReturnType
  MethodName(Parameters)
{
   //Method body.
}
```

- **Methods that need to be used by other classes should be made public.**

Methods

- **The attributes of a class might need to be:**
  - changed
  - accessed
  - calculated
- **The methods that change and access attributes are called *accessors* and *mutators*.**

Accessors and Mutators

- **Because of the concept of data hiding, fields in a class are private.**
- **The methods that retrieve the data of fields are called *accessors*.**
- **The methods that modify the data of fields are called *mutators*.**
- **Each field that the programmer wishes to be viewed by other classes needs an accessor.**
- **Each field that the programmer wishes to be modified by other classes needs a mutator.**

Accessors and Mutators

- **For the `Rectangle` example, the accessors and mutators are:**
  - `setLength` : Sets the value of the `length` field.
    ```
    public void setLength(double len) ...
    ```
  - `setWidth` : Sets the value of the `width` field.
    ```
    public void setLength(double w) ...
    ```
  - `getLength` : Returns the value of the `length` field.
    ```
    public double getLength() ...
    ```
  - `getWidth` : Returns the value of the `width` field.
    ```
    public double getWidth() ...
    ```
- **Other names for these methods are *getters* and *setters*.**

Stale Data

- **Some data is the result of a calculation.**
- **Consider the area of a rectangle.**
  - `length` times `width`
- **It would be impractical to use an `area` variable here.**
- **Data that requires the calculation of various factors has the potential to become *stale*.**
- **To avoid stale data, it is best to calculate the value of that data within a method rather than store it in a variable.**

Stale Data

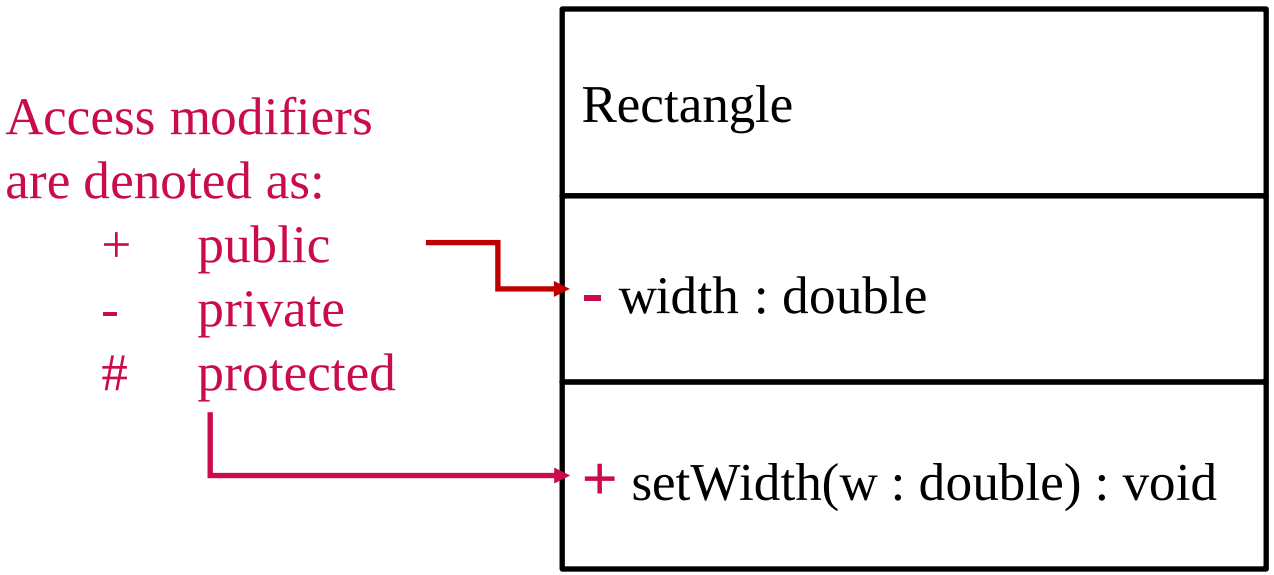
- **Rather than use an `area` variable in a rectangle class:**

```
public double getArea()
{
  return length * width;
}
```

- **This dynamically calculates the value of the rectangle's area when the method is called.**
- **Now, any change to the `length` or `width` variables will not leave the area of the rectangle stale.**

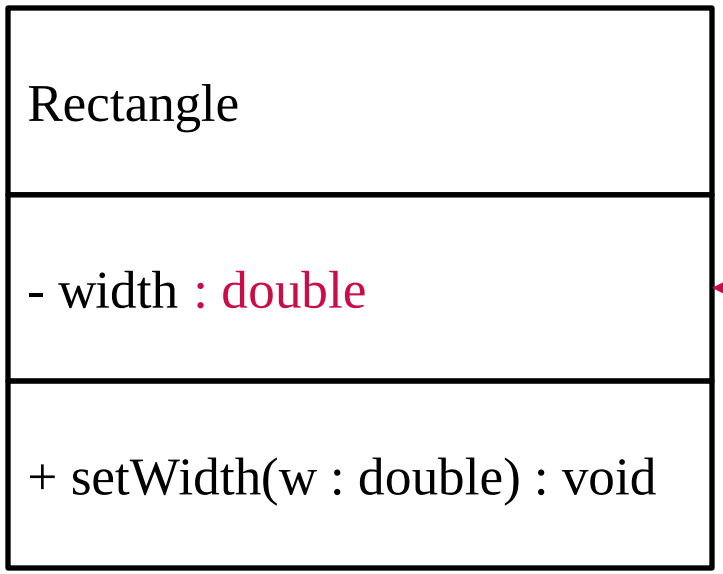
UML Data Type and Parameter Notation

- **UML diagrams are language independent.**
- **UML diagrams use an independent notation to show return types, access modifiers, etc.**



UML Data Type and Parameter Notation

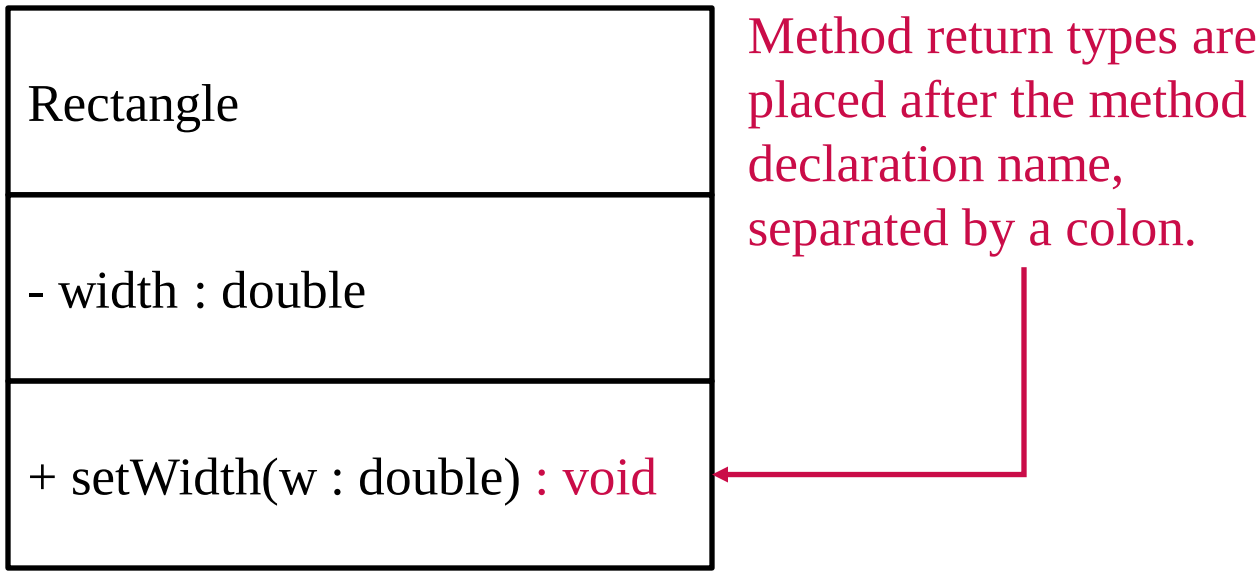
- **UML diagrams are language independent.**
- **UML diagrams use an independent notation to show return types, access modifiers, etc.**



Variable types are placed after the variable name, separated by a colon.

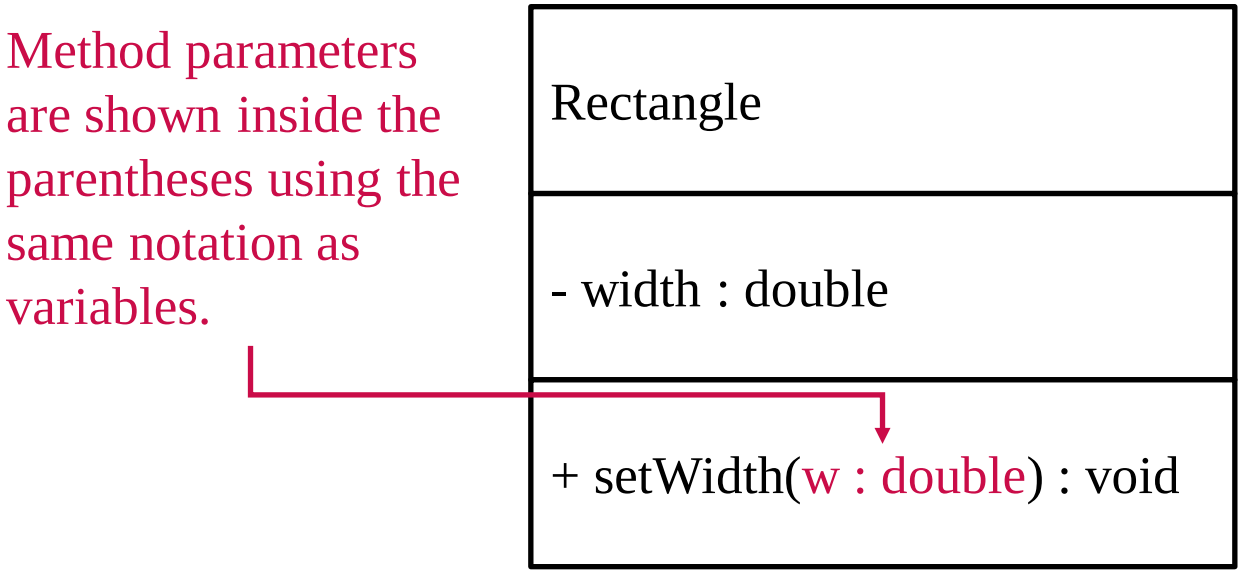
UML Data Type and Parameter Notation

- **UML diagrams are language independent.**
- **UML diagrams use an independent notation to show return types, access modifiers, etc.**



UML Data Type and Parameter Notation

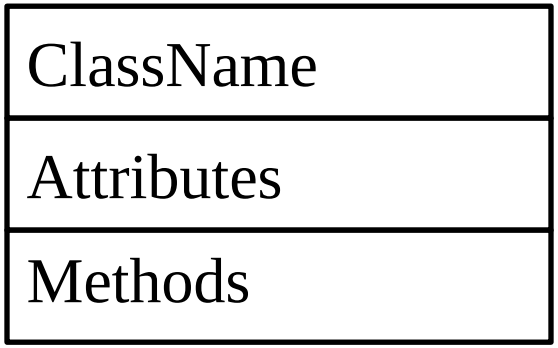
- **UML diagrams are language independent.**
- **UML diagrams use an independent notation to show return types, access modifiers, etc.**



Converting the UML Diagram to Code

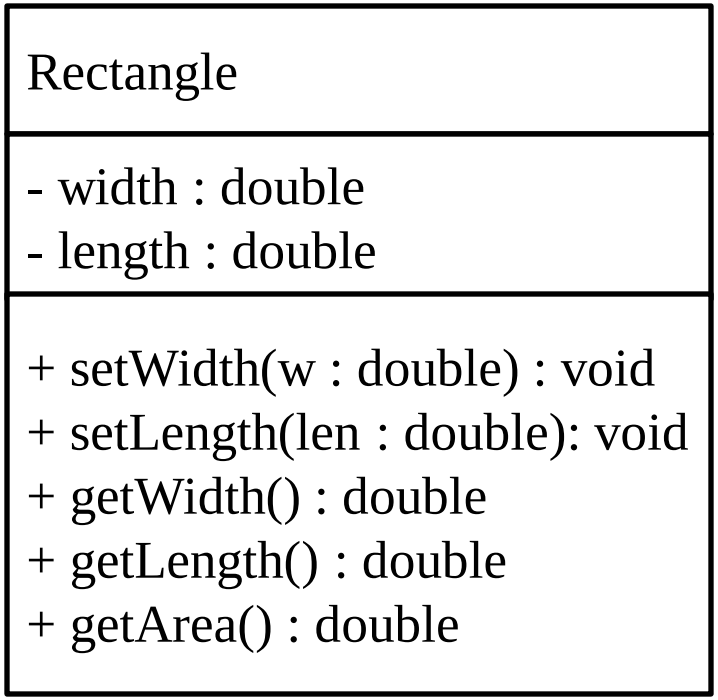
- **Putting all of this information together, a Java class file can be built easily using the UML diagram.**
- **The UML diagram parts match the Java class file structure.**

```
class header
{
  Attributes
  Methods
}
```



Converting the UML Diagram to Code

The structure of the class can be compiled and tested without having bodies for the methods. Just be sure to put in dummy return values for methods that have a return type other than void.

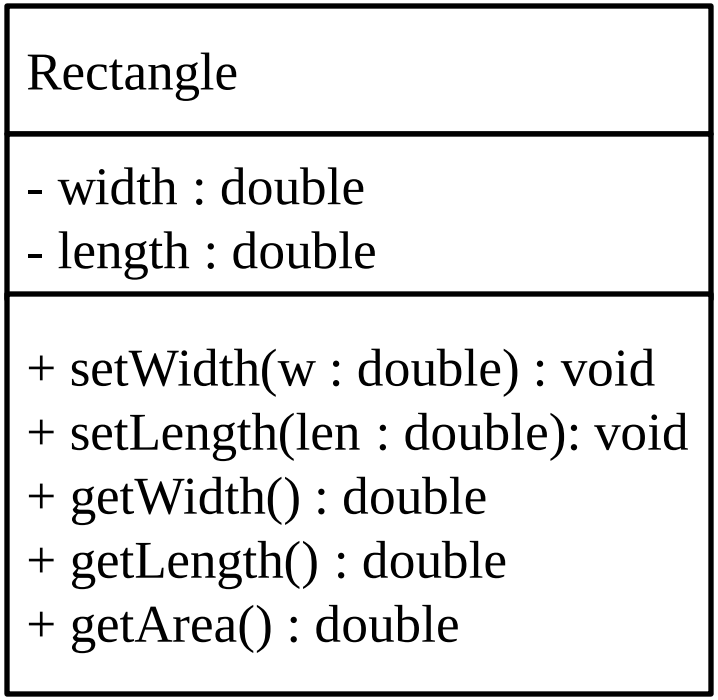



```
public class Rectangle
{
   private double width;
   private double length;

   public void setWidth(double w)
   {
   }
   public void setLength(double len)
   {
   }
   public double getWidth()
   {    return 0.0;
   }
   public double getLength()
   {    return 0.0;
   }
   public double getArea()
   {    return 0.0;
   }
}
```

Converting the UML Diagram to Code

Once the class structure has been tested, the method bodies can be written and tested.




```
public class Rectangle
{
   private double width;
   private double length;

   public void setWidth(double w)
   {    width = w;
   }
   public void setLength(double len)
   {    length = len;
   }
   public double getWidth()
   {    return width;
   }
   public double getLength()
   {    return length;
   }
   public double getArea()
   {    return length * width;
   }
}
```

Class Layout Conventions

- **The layout of a source code file can vary by employer or instructor.**
- **Generally the layout is:**
  - Attributes are typically listed first
  - Methods are typically listed second
    - The main method is sometimes first, sometimes last.
    - Accessors and mutators are typically grouped.

A Driver Program

- An application in Java is a collection of classes that interact.
- The class that starts the application must have a `main` method.
- This class can be used as a *driver* to test the capabilities of other classes.
- In the `Rectangle` class example, notice that there was no `main` method.

A Driver Program

```
public class RectangleDemo
{
  public static void main(String[] args)
  {
    Rectangle r = new Rectangle();
    r.setWidth(10);
    r.setLength(10);
    System.out.println("Width = "
                    + r.getWidth());
     System.out.println("Length = "
                    + r.getLength());
     System.out.println("Area = "
                    + r.getArea());
  }
}
```

This `RectangleDemo` class is a Java application that uses the Rectangle class.

```
public class Rectangle
{
    private double width;
    private double length;

    public void setWidth(double w)
    {     width = w;
    }
    public void setLength(double len)
    {     length = len;
    }
    public double getWidth()
    {     return width;
    }
    public double getLength()
    {     return length;
    }
    public double getArea()
    {     return length * width;
    }
}
```

Multiple Arguments

- **Methods can have multiple parameters.**
- **The format for a multiple parameter method is:**

```
AccessModifier ReturnType MethodName(ParamType ParamName,
                                     ParamType ParamName,
                                     etc)
{
}
```

- **Parameters in methods are treated as local variables within the method.**
- **Example: MultipleArgs.java**

Arguments Passed By Value

- In Java, all arguments to a method are passed “by value”.
- If the argument is a reference to an object, it is the reference that is passed to the method.
- If the argument is a primitive, a copy of the value is passed to the method.

Instance Fields and Methods

- **Fields and methods that are declared as previously shown are called *instance fields* and *instance methods*.**
- **Objects created from a class each have their own copy of instance fields.**
- **Instance methods are methods that are not declared with a special keyword, *static*.**

Instance Fields and Methods

- **Instance fields and instance methods require an object to be created in order to be used.**
- **Example: RoomAreas.java**
- **Note that each room represented in this example can have different dimensions.**

```
Rectangle kitchen = new Rectangle();
Rectangle bedroom = new Rectangle();
Rectangle den = new Rectangle();
```

Constructors

- Classes can have special methods called *constructors*.
- Constructors are used to perform operations at the time an object is created.
- Constructors typically initialize instance fields and perform other object initialization tasks.

Constructors

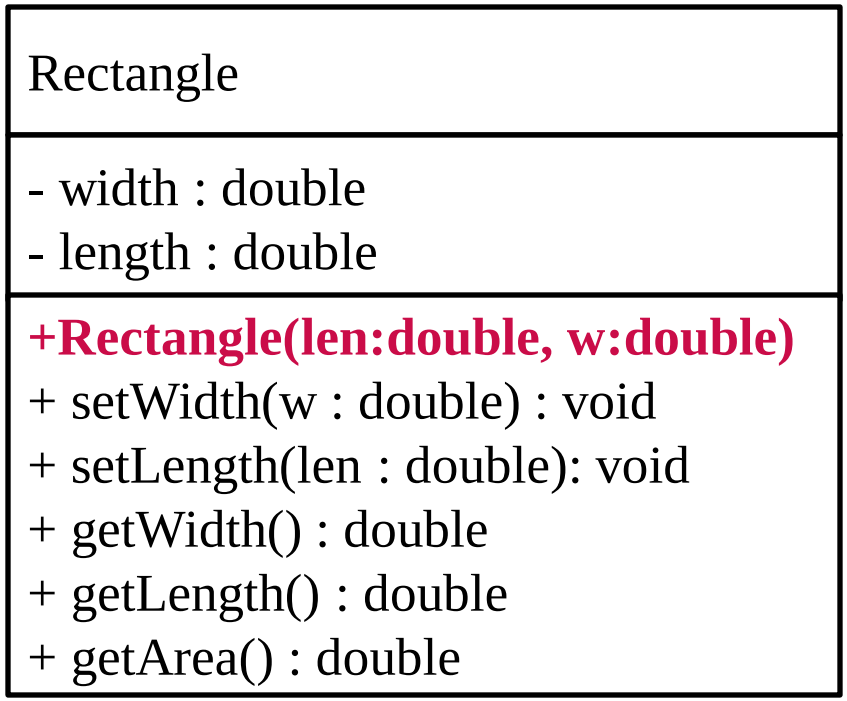
- **Constructors have a few special properties that set them apart from normal methods.**
  - Constructors have the same name as the class.
  - Constructors have no return type (not even void).
  - Constructors may not return any values.
  - Constructors are typically public.
- **Example: ConstructorDemo.java**
- **Example: RoomConstructor.java**

The Default Constructor

- **If a constructor is not defined, Java provides a default constructor.**
  - It sets all of the class' numeric fields to 0.
  - It sets all of the class' boolean fields to false.
  - It sets all of the class' reference variables, the default constructor sets them to the special value *null*.
- **The default constructor is a constructor with no parameters.**
- **Default constructors are used to initialize an object in a default configuration.**

Constructors in UML

- **In UML, the most common way constructors are defined is:**



Notice there is no return type listed for constructors.

The `String` Class Constructor

- **One of the `String` class constructors accepts a string literal as an argument.**
- **This string literal is used to initialize a `String` object.**
- **For instance:**

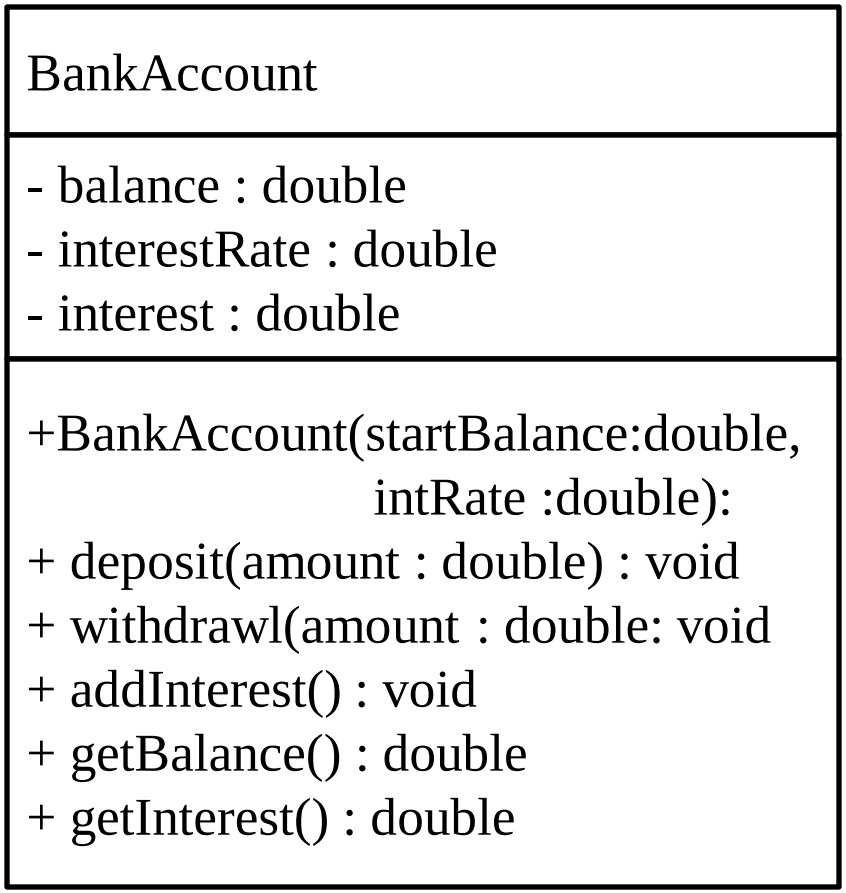
```
String name = new String("Michael Long");
```

The `String` Class Constructor

- **This creates a new reference variable *name* that points to a `String` object that represents the name "Michael Long"**
- **Because they are used so often, Strings can be created with a shorthand:**

```
String name = "Michael Long";
```

The `BankAccount` Example



Classes, Variables and Scope

- **The list below shows the scope of a variable depending on where it is declared.**
  - Inside a method:
    - Visible only within that method.
    - Called a local variable.
  - In a method parameter:
    - Called a parameter variable.
    - Same as a local variable
    - Visible only within that method.
  - Inside the class but not in a method:
    - Visible to all methods of the class.
    - Called an instance field.

Shadowing

- A parameter variable is, in effect, a local variable.
- Within a method, variable names must be unique.
- A method may have a local variable with the same name as an instance field.
- This is called shadowing.
- The local variable will *hide* the value of the instance field.
- Shadowing is discouraged and local variable names should not be the same as instance field names.

Packages and `import` Statements

- **A package is a group of related classes.**
- **The classes in the Java API are organized into packages.**
  - For example, the `Scanner` class is in the `java.util` package.
- **Many of the API classes must be imported before they can be used. For example, the following statement is required to import the `Scanner` class:**

```
import java.util.Scanner;
```

This statement appears at the top of the program's source code.

Packages and `import` Statements

- **Explicit `import` Statements**
- **An explicit `import` statement specifies a single class:**

```
import java.util.Scanner;
```

- **Wildcard `import` Statements**
- **A wildcard `import` statement imports all of the classes in a package:**

```
import java.util.*;
```

Packages and `import` Statements

- **The `java.lang` package**
  - Automatically imported into every Java program.
  - Contains general classes such as `String` and `System`.
  - You do not have to write an `import` statement for the `java.lang` package.

Packages and `import` Statements

- **You will use other packages as you learn more about Java.**
- **Table 3-2 lists a few examples.**

Package	Description
`java.applet`	Provides the classes necessary to create an applet.
`java.awt`	Provides classes for the Abstract Windowing Toolkit. These classes are used in drawing images and creating graphical user interfaces.
`java.io`	Provides classes that perform various types of input and output.
`java.lang`	Provides general classes for the Java language. This package is automatically imported.
`java.net`	Provides classes for network communications.
`java.security`	Provides classes that implement security features.
`java.sql`	Provides classes for accessing databases using structured query language.
`java.text`	Provides various classes for formatting text.
`java.util`	Provides various utility classes.
`javax.swing`	Provides classes for creating graphical user interfaces.

Object Oriented Design
Finding Classes and Their Responsibilities

- **Finding the classes**
  - Get written description of the problem domain
  - Identify all nouns, each is a potential class
  - Refine list to include only classes relevant to the problem
- **Identify the responsibilities**
  - Things a class is responsible for knowing
  - Things a class is responsible for doing
  - Refine list to include only classes relevant to the problem

Object Oriented Design
Finding Classes and Their Responsibilities

- Identify the responsibilities
  - Things a class is responsible for knowing
  - Things a class is responsible for doing
  - Refine list to include only classes relevant to the problem