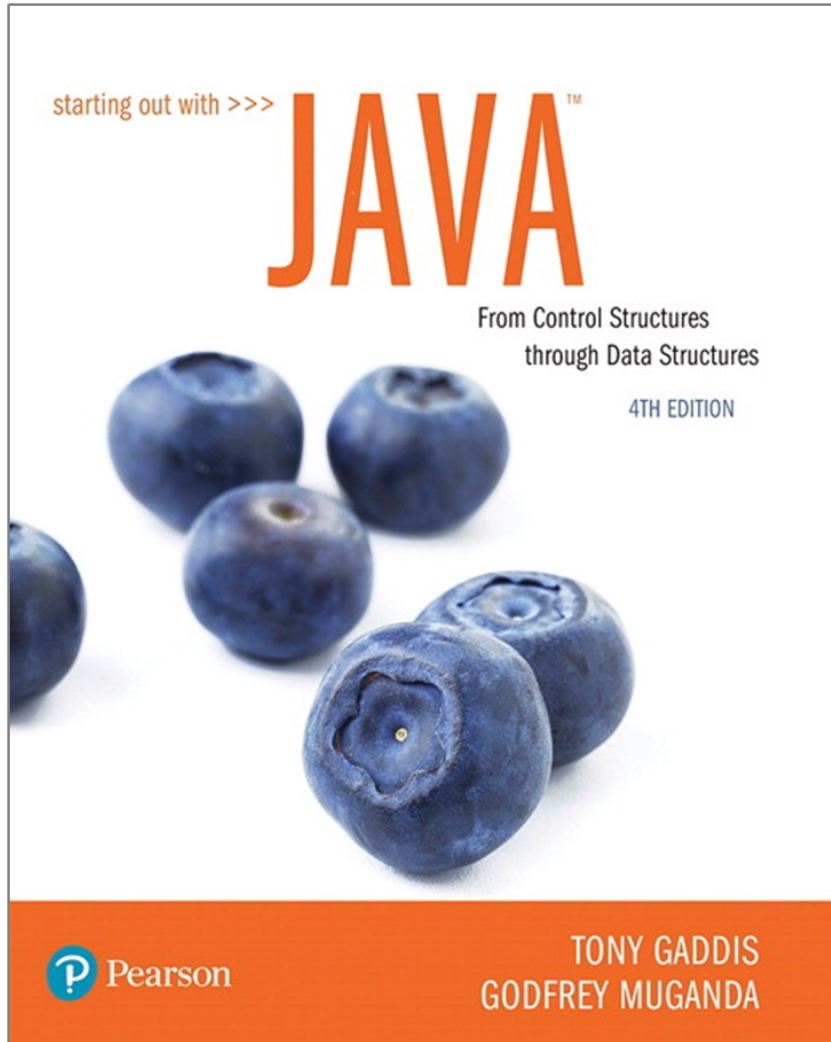


STARTING OUT WITH JAVA™

4th Edition



Chapter 6

A First Look at Classes

Chapter Topics

Chapter 6 discusses the following main topics:

- Objects and Classes
- Writing a Simple Class, Step by Step
- Instance Fields and Methods
- Constructors
- Passing Objects as Arguments
- Overloading Methods and Constructors
- Scope of Instance Fields
- Packages and `import` Statements

Objects and Classes (1 of 8)

- An object exists in memory, and performs a specific task.
- Objects have two general capabilities:
 - Objects can store data. The pieces of data stored in an object are known as *fields*.
 - Objects can perform operations. The operations that an object can perform are known as *methods*.

Objects and Classes (2 of 8)

- You have already used the following objects:
 - `Scanner` objects, for reading input
 - `Random` objects, for generating random numbers
 - `PrintWriter` objects, for writing data to files
- When a program needs the services of a particular type of object, it creates that object in memory, and then calls that object's methods as necessary.

Objects and Classes (3 of 8)

- Classes: Where Objects Come From
 - A *class* is code that describes a particular type of object. It specifies the data that an object can hold (the object's fields), and the actions that an object can perform (the object's methods).
 - You can think of a class as a code "blueprint" that can be used to create a particular type of object.

Objects and Classes (4 of 8)

- When a program is running, it can use the class to create, in memory, as many objects of a specific type as needed.
- Each object that is created from a class is called an *instance* of the class.

Objects and Classes (5 of 8)

Example:

This expression creates a Scanner object in memory.

```
Scanner keyboard = new Scanner(System.in);
```

The object's memory address
is assigned to the keyboard
variable.

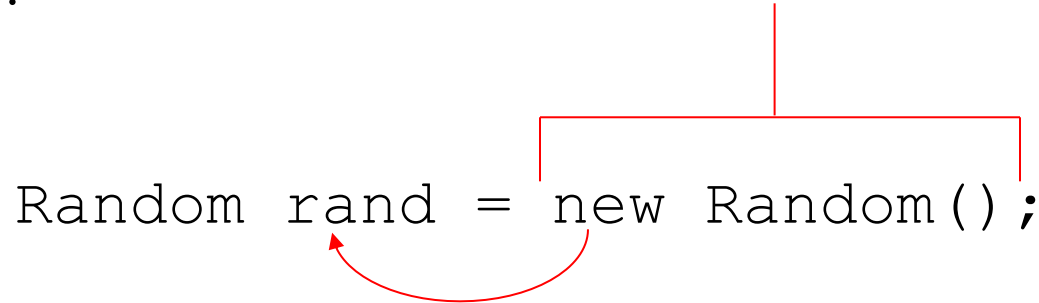


Objects and Classes (6 of 8)

Example:

This expression creates a
Random object in memory.

```
Random rand = new Random();
```



The object's memory address is
assigned to the `rand` variable.



Objects and Classes (7 of 8)

Example:

This expression creates a
PrintWriter object in memory.

```
PrintWriter outputFile = new PrintWriter("numbers.txt");
```

The object's memory address is assigned to
the `outputFile` variable.



Objects and Classes (8 of 8)

- The Java API provides many classes
 - So far, the classes that you have created objects from are provided by the Java API.
 - Examples:
 - `Scanner`
 - `Random`
 - `PrintWriter`
- See [ObjectDemo.java](#)

Writing a Class, Step by Step (1 of 2)

- A `Rectangle` object will have the following fields:
 - `length`. The length field will hold the rectangle's length.
 - `width`. The width field will hold the rectangle's width.

Writing a Class, Step by Step (2 of 2)

- The `Rectangle` class will also have the following methods:
 - **`setLength`**. The `setLength` method will store a value in an object's `length` field.
 - **`setWidth`**. The `setWidth` method will store a value in an object's `width` field.
 - **`getLength`**. The `getLength` method will return the value in an object's `length` field.
 - **`getWidth`**. The `getWidth` method will return the value in an object's `width` field.
 - **`getArea`**. The `getArea` method will return the area of the rectangle, which is the result of the object's `length` multiplied by its `width`.

UML Diagram

- Unified Modeling Language (UML) provides a set of standard diagrams for graphically depicting object-oriented systems.

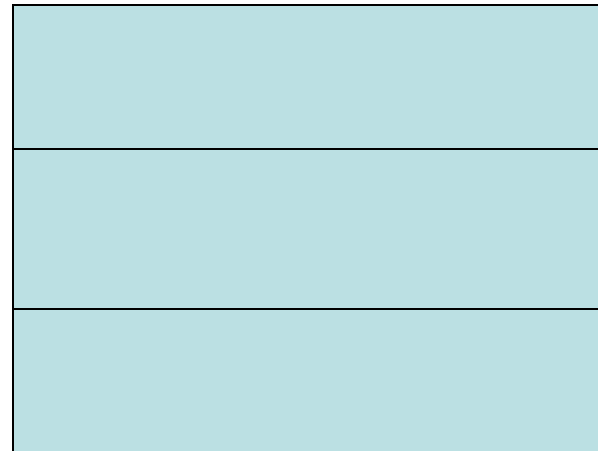
Class name goes here



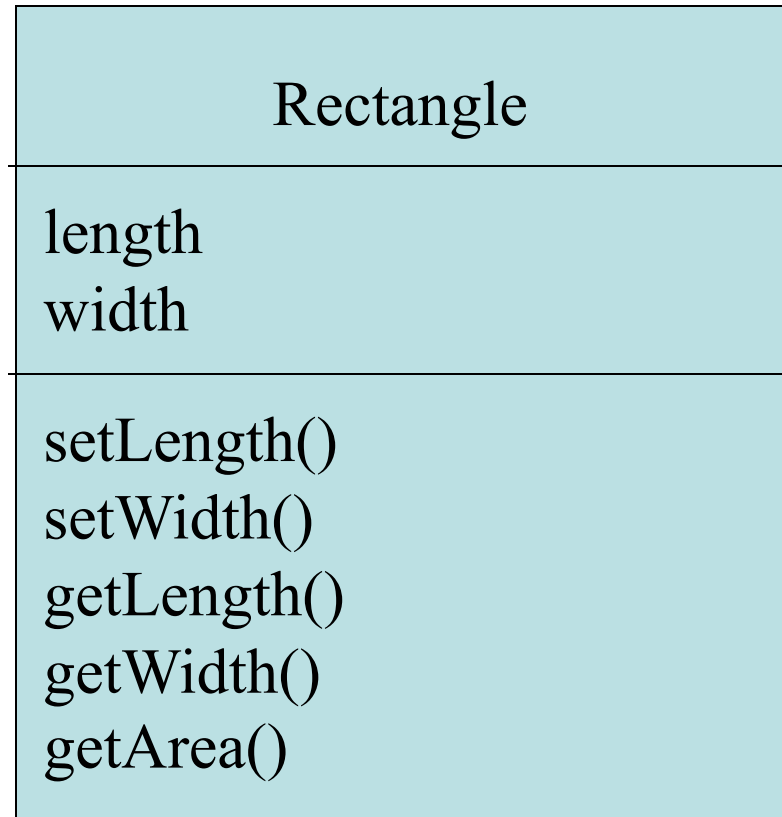
Fields are listed here



Methods are listed here



UML Diagram for Rectangle class



Writing the Code for the Class Fields

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

Access Specifiers

- An access specifier is a Java keyword that indicates how a field or method can be accessed.
- `public`
 - When the `public` access specifier is applied to a class member, the member can be accessed by code inside the class or outside.
- `private`
 - When the `private` access specifier is applied to a class member, the member cannot be accessed by code outside the class. The member can be accessed only by methods that are members of the same class.

Header for the `setLength` Method

Access specifier
↓
public

Return Type
↓
void

Method Name
↓
setLength

(double len)

Notice the word **static** does not appear in the method header designed to work on an instance of a class (*instance method*).

Parameter variable declaration



Writing and Demonstrating the `setLength` Method

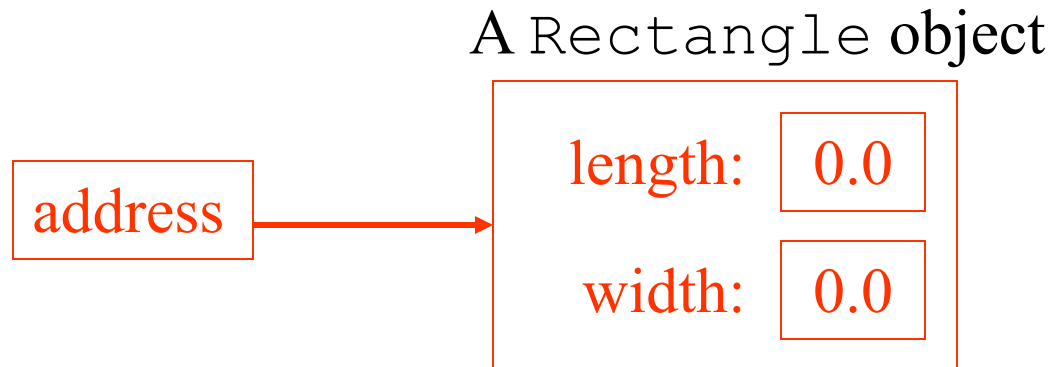
```
/**  
    The setLength method stores a value in the  
    length field.  
    @param len The value to store in length.  
*/  
public void setLength(double len)  
{  
    length = len;  
}
```

Examples: [Rectangle.java](#), [LengthDemo.java](#)

Creating a Rectangle object

```
Rectangle box = new Rectangle ();
```

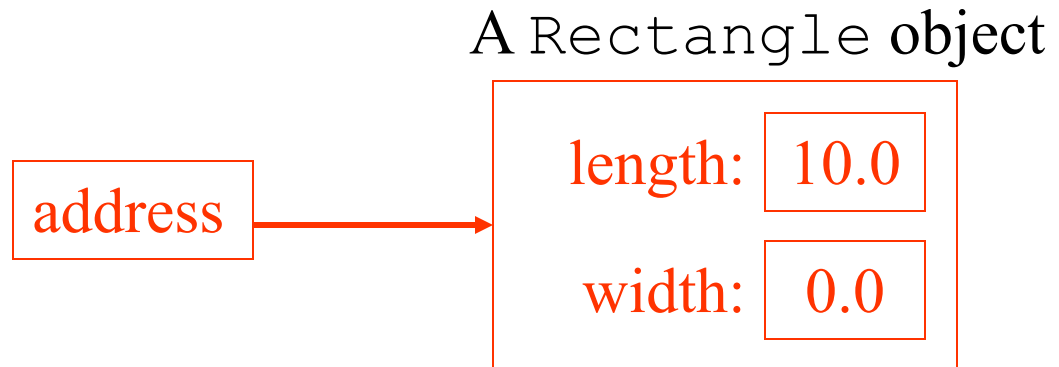
The box
variable holds
the address of
the Rectangle
object.



Calling the `setLength` Method

```
box.setLength(10.0);
```

The `box` variable holds the address of the `Rectangle` object.



This is the state of the `box` object after the `setLength` method executes.

Writing the `getLength` Method

```
/**  
    The getLength method returns a Rectangle  
    object's length.  
    @return The value in the length field.  
*/  
public double getLength()  
{  
    return length;  
}
```

Similarly, the `setWidth` and `getWidth` methods can be created.

Examples: [Rectangle.java](#), [LengthWidthDemo.java](#)

Writing and Demonstrating the getArea Method

```
/**  
    The getArea method returns a Rectangle  
    object's area.  
    @return The product of length times width.  
*/  
  
public double getArea()  
{  
    return length * width;  
}
```

Examples: [Rectangle.java](#), [RectangleDemo.java](#)

Accessor and Mutator Methods

- 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*.

Data Hiding (1 of 2)

- An object hides its internal, private fields from code that is outside the class that the object is an instance of.
- Only the class's methods may directly access and make changes to the object's internal data.
- Code outside the class must use the class's public methods to operate on an object's private fields.

Data Hiding (2 of 2)

- Data hiding is important because classes are typically used as components in large software systems, involving a team of programmers.
- Data hiding helps enforce the integrity of an object's internal data.

Stale Data (1 of 2)

- 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 (2 of 2)

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

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

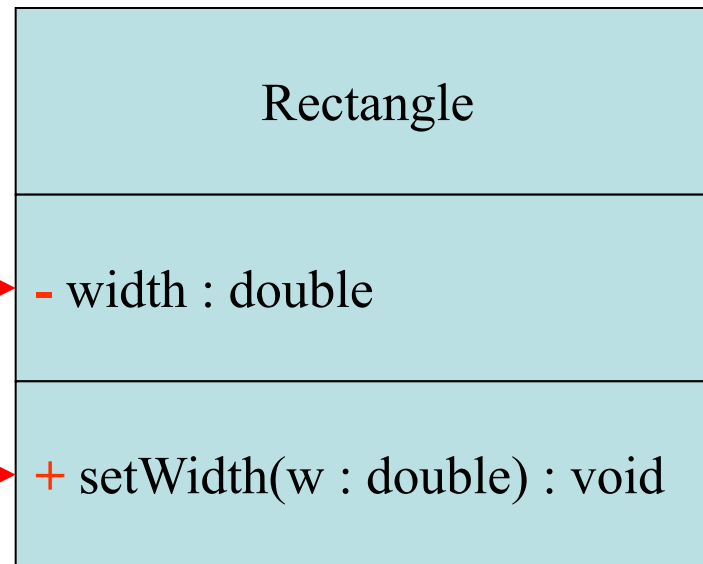
- This dynamically calculates the value of the rectangle's area when the method is called.
- Now, any change to the `length` or `width` variables will not leave the area of the rectangle stale.

UML Data Type and Parameter Notation (1 of 4)

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

Access modifiers
are denoted as:

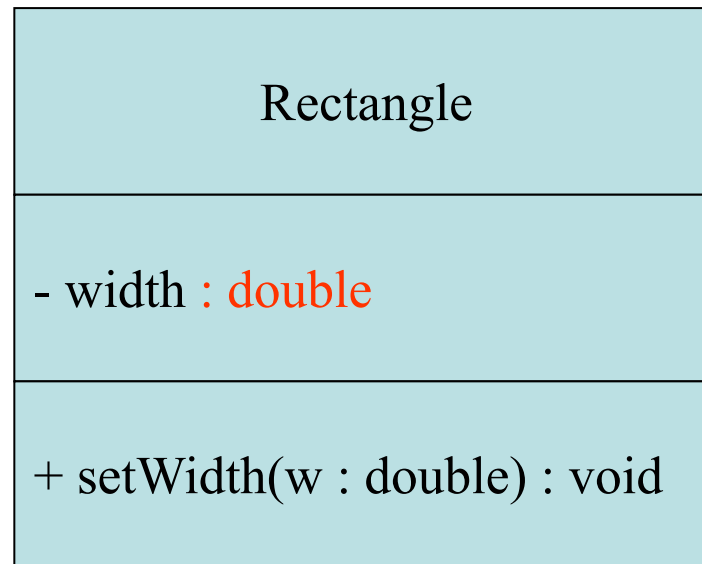
+ public
- private



UML Data Type and Parameter Notation

(2 of 4)

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

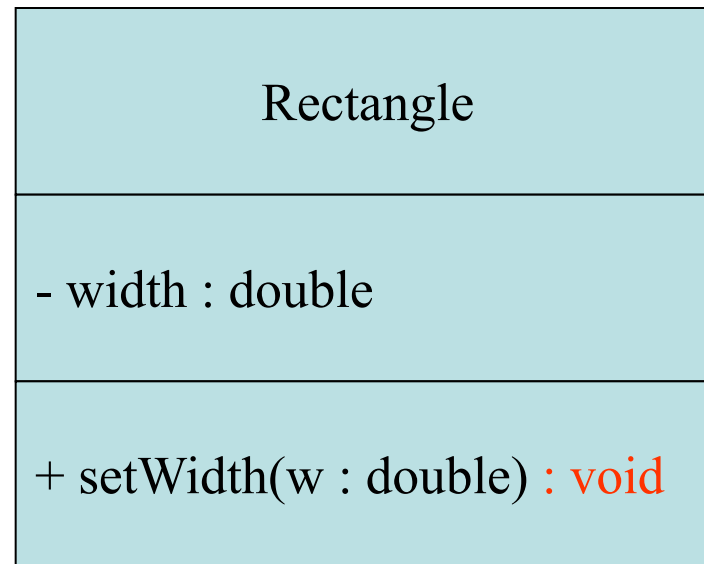


Variable types are placed after the variable name, separated by a colon.

UML Data Type and Parameter Notation

(3 of 4)

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



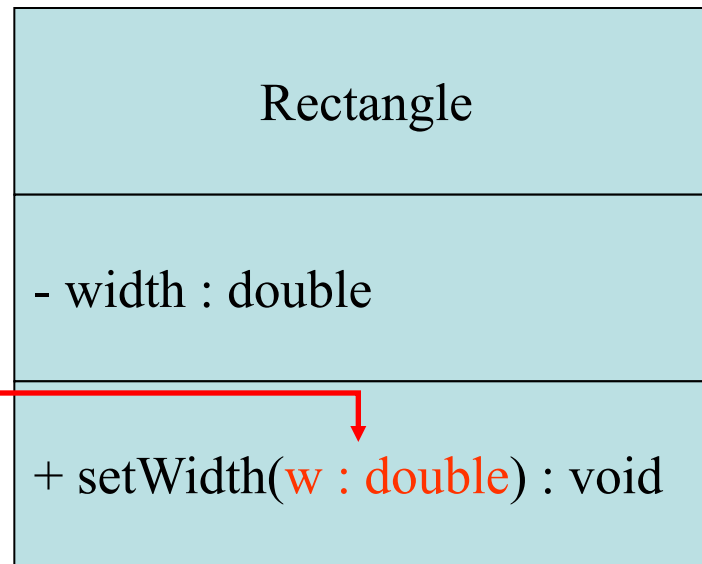
Method return types are placed after the method declaration name, separated by a colon.

UML Data Type and Parameter Notation

(4 of 4)

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

Method parameters are shown inside the parentheses using the same notation as variables.



Converting the UML Diagram to Code

(1 of 3)

- 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.

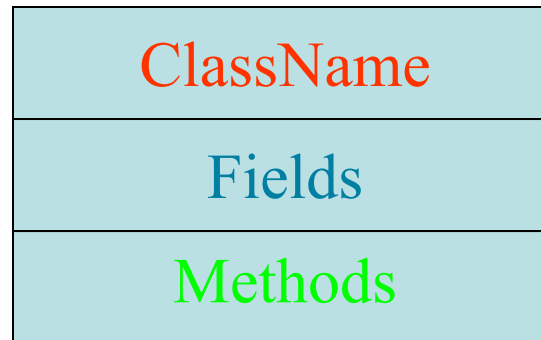
class header

{

Fields

Methods

}



Converting the UML Diagram to Code

(2 of 3)

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.

Rectangle
- width : double - length : double
+ setWidth(w : double) : void + setLength(len : double): void + getWidth() : double + getLength() : double + getArea() : double

```
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

(3 of 3)

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

Rectangle
- width : double - length : double
+ setWidth(w : double) : void + setLength(len : double): void + getWidth() : double + getLength() : double + getArea() : double

```
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.
- A common layout is:
 - Fields listed first
 - Methods listed second
 - Accessors and mutators are typically grouped.
- There are tools that can help in formatting layout to specific standards.

Instance Fields and Methods (1 of 2)

- 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 (2 of 2)

- Instance fields and instance methods require an object to be created in order to be used.
- See 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();
```

States of Three Different Rectangle Objects

The kitchen variable holds the address of a Rectangle Object.

address

length: 10.0

width: 14.0

The bedroom variable holds the address of a Rectangle Object.

address

length: 15.0

width: 12.0

The den variable holds the address of a Rectangle Object.

address

length: 20.0

width: 30.0

Constructors (1 of 2)

- Classes can have special methods called *constructors*.
- A constructor is a method that is automatically called when an object is created.
- 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 (2 of 2)

- 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.

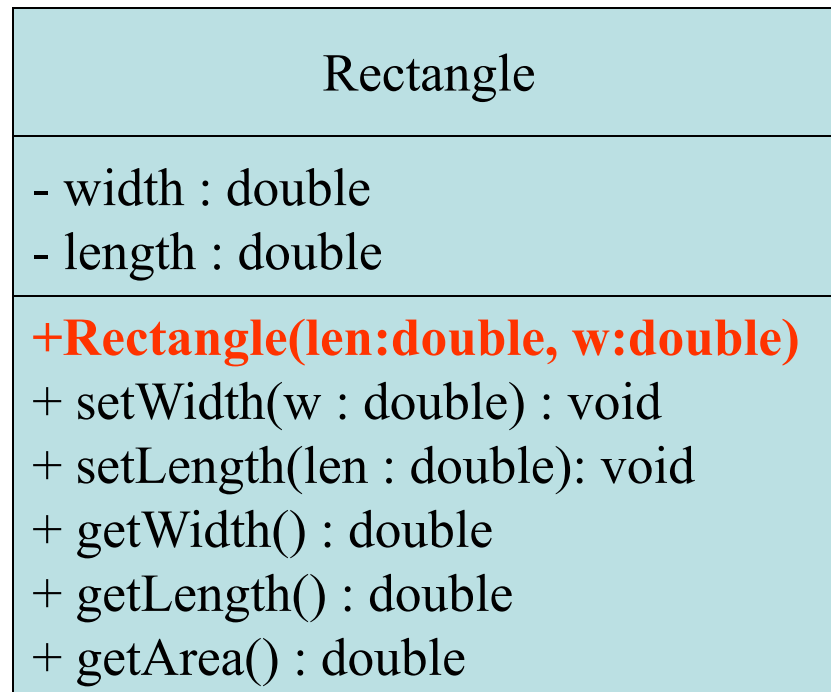
Constructor for Rectangle Class

```
/**  
    Constructor  
    @param len The length of the rectangle.  
    @param w The width of the rectangle.  
*/  
public Rectangle(double len, double w)  
{  
    length = len;  
    width = w;  
}
```

Examples: [Rectangle.java](#), [ConstructorDemo.java](#)

Constructors in UML

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



Notice there is no return type listed for constructors.

Uninitialized Local Reference Variables

- Reference variables can be declared without being initialized.

```
Rectangle box;
```

- This statement does not create a `Rectangle` object, so it is an uninitialized local reference variable.
- A local reference variable must reference an object before it can be used, otherwise a compiler error will occur.

```
box = new Rectangle(7.0, 14.0);
```

- `box` will now reference a `Rectangle` object of length 7.0 and width 14.0.

The Default Constructor (1 of 2)

- When an object is created, its constructor is always called.
- If you do not write a constructor, Java provides one when the class is compiled. The constructor that Java provides is known as the *default constructor*.
 - It sets all of the object's numeric fields to 0.
 - It sets all of the object's `boolean` fields to `false`.
 - It sets all of the object's reference variables to the special value *null*.

The Default Constructor (2 of 2)

- The default constructor is a constructor with no parameters, used to initialize an object in a default configuration.
- The only time that Java provides a default constructor is when you do not write any constructor for a class.
 - See example: First version of [Rectangle.java](#)
- A default constructor is not provided by Java if a constructor is already written.
 - See example: [Rectangle.java](#) with Constructor

Writing Your Own No-Arg Constructor

- A constructor that does not accept arguments is known as a *no-arg constructor*.
- The default constructor (provided by Java) is a no-arg constructor.
- We can write our own no-arg constructor

```
public Rectangle()  
{  
    length = 1.0;  
    width = 1.0;  
}
```

The `String` Class Constructor (1 of 2)

- 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 (2 of 2)

- 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, `String` objects can be created with a shorthand:

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

Passing Objects as Arguments

- When you pass an object as an argument, the thing that is passed into the parameter variable is the object's memory address.
- As a result, parameter variable references the object, and the receiving method has access to the object.
- See [DieArgument.java](#)

Overloading Methods and Constructors

- Two or more methods in a class may have the same name as long as their parameter lists are different.
- When this occurs, it is called *method overloading*. This also applies to constructors.
- Method overloading is important because sometimes you need several different ways to perform the same operation.

Overloaded Method add

```
public int add(int num1, int num2)
{
    int sum = num1 + num2;
    return sum;
}
```

```
public String add (String str1, String str2)
{
    String combined = str1 + str2;
    return combined;
}
```

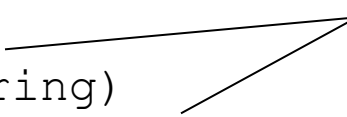
Method Signature and Binding

- A method signature consists of the method's name and the data types of the method's parameters, in the order that they appear. The return type is not part of the signature.

`add(int, int)`

`add(String, String)`

*Signatures of the
add methods of
previous slide*



- The process of matching a method call with the correct method is known as *binding*. The compiler uses the method signature to determine which version of the overloaded method to bind the call to.

Binding (1 of 3)

- "binding" refers to the process of associating a method call with the actual method implementation that will be executed at runtime
- There are two main types of binding:
- compile-time binding and
(also known as early binding or static binding)
- runtime binding
(also known as late binding or dynamic binding)

Binding (2 of 3)

Compile-Time (Static) Binding:

- Occurs during the compilation phase
- Compiler determines which method to call based on the static type of the object reference and the method signature
- In other words, the decision about which method to call is made at compile-time based on the type information available at that time
- Common with method overloading: multiple methods, same name, different parameter lists
- Appropriate method decided by the compiler based on the arguments passed and their types

Binding (3 of 3)

Runtime (Dynamic) Binding

- Occurs during runtime
- The decision about which method to call is made based on the actual type of the object at runtime

Rectangle Class Constructor Overload

(1 of 2)

If we were to add the no-arg constructor we wrote previously to our `Rectangle` class in addition to the original constructor we wrote, what would happen when we execute the following calls?

```
Rectangle box1 = new Rectangle();
```

```
Rectangle box2 = new Rectangle(5.0, 10.0);
```

Rectangle Class Constructor Overload

(2 of 2)

If we were to add the no-arg constructor we wrote previously to our `Rectangle` class in addition to the original constructor we wrote, what would happen when we execute the following calls?

```
Rectangle box1 = new Rectangle();  
Rectangle box2 = new Rectangle(5.0, 10.0);
```

The first call would use the no-arg constructor and `box1` would have a length of 1.0 and width of 1.0.

The second call would use the original constructor and `box2` would have a length of 5.0 and a width of 10.0.

The BankAccount Example

BankAccount.java

AccountTest.java

Overloaded Constructors

Overloaded deposit methods

Overloaded withdraw methods

Overloaded setBalance methods

BankAccount	
-balance:double	
+BankAccount()	
+BankAccount(startBalance:double)	
+BankAccount(str:String):	
+deposit(amount:double):void	
+deposit(str:String):void	
+withdraw(amount:double):void	
+withdraw(str:String):void	
+setBalance(b:double):void	
+setBalance(str:String):void	
+getBalance():double	

Scope of Instance Fields

- Variables declared as instance fields in a class can be accessed by any instance method in the same class as the field.
- If an instance field is declared with the `public` access specifier, it can also be accessed by code outside the class, as long as an instance of the class exists.

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

- Classes in the Java API are organized into *packages*.
- Explicit and Wildcard `import` statements
 - Explicit imports name a specific class
 - `import java.util.Scanner;`
 - Wildcard imports name a package, followed by an `*`
 - `import java.util.*;`
- The `java.lang` package is automatically made available to any Java class.

Some Java Standard Packages

Table 6-2 A few of the standard Java packages

Package	Description
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.

Object Oriented Design

Finding Classes and Their Responsibilities (1 of 2)

- 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 (2 of 2)

- 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

Copyright



This work is protected by United States copyright laws and is provided solely for the use of instructions in teaching their courses and assessing student learning. dissemination or sale of any part of this work (including on the World Wide Web) will destroy the integrity of the work and is not permitted. The work and materials from it should never be made available to students except by instructors using the accompanying text in their classes. All recipients of this work are expected to abide by these restrictions and to honor the intended pedagogical purposes and the needs of other instructors who rely on these materials.