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
– 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
– 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
– 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
– 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

Example:

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

This expression creates a Scanner object in memory.

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

Objects and Classes

Example:

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

This expression creates a Random object in memory.

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

Objects and Classes

Example:

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

This expression creates a PrintWriter object in memory.

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

Objects and Classes

• 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

• 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

• 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.

**UML Diagram for Rectangle class**

```
Rectangle
+ length
+ width
+ setLength()
+ getWidth()
+ getLength()
+ getArea()
```

**Writing the Code for the Class Fields**

```java
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**

```
public void setLength (double len)
```

**Writing and Demonstrating the setLength Method**

```java
/**
 * 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

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

The box variable holds the address of the Rectangle object.

A Rectangle object

```
x = 0.0
y = 0.0
length = 0.0
width = 0.0
```

Calling the setLength Method

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

The box variable holds the address of the Rectangle object.

A Rectangle object

```
x = 0.0
y = 0.0
length = 10.0
width = 0.0
```

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

Writing the getLength Method

```java
/**
 * 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

```java
/**
 * 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.
    ```java
    public void setLength(double len) ...
    ```
  - setWidth : Sets the value of the width field.
    ```java
    public void setWidth(double w) ...
    ```
  - getLength : Returns the value of the length field.
    ```java
    public double getLength() ...
    ```
  - getWidth : Returns the value of the width field.
    ```java
    public double getWidth() ...
    ```
- Other names for these methods are getters and setters.
Data Hiding

• 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

• 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

• Some data is the result of a calculation.
• Consider the area of a rectangle.
  – length × 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.
UML Data Type and Parameter Notation

- 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.

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

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.

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

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

- 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.

Objects created from a class each have their own copy of instance fields.

Instance Fields and Methods

- 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

<table>
<thead>
<tr>
<th>Variable</th>
<th>Address</th>
<th>Length</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>kitchen</td>
<td></td>
<td>10.0</td>
<td>14.0</td>
</tr>
<tr>
<td>bedroom</td>
<td></td>
<td>15.0</td>
<td>12.0</td>
</tr>
<tr>
<td>den</td>
<td></td>
<td>20.0</td>
<td>30.0</td>
</tr>
</tbody>
</table>

The kitchen variable holds the address of a Rectangle Object.
The bedroom variable holds the address of a Rectangle Object.
The den variable holds the address of a Rectangle Object.

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.

```
/**
 * 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:

\[
\text{Rectangle} \\
\quad width : \text{double} \\
\quad length : \text{double} \\
\quad \text{+Rectangle}(\text{len:double}, \text{w:double}) \\
\quad \text{+setWidth(} \text{w : double}) : \text{void} \\
\quad \text{+setLength(} \text{len : double}) : \text{void} \\
\quad \text{+getWidth(} : \text{double} \\
\quad \text{+getLength(} : \text{double} \\
\quad \text{+getArea(} : \text{double}
\]

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.

  \[
  \text{box = new Rectangle(7.0, 14.0);}
  \]

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

The Default Constructor

• 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 String Class Constructor

• One of the String class constructors accepts a string literal as an argument.

  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, String objects can be created with a shorthand:

  ```java
  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

```java
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.

- 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.

```
add(int, int)   add(String, String)
Signatures of the add methods of previous slide
```

Rectangle Class Constructor Overload

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?

```java
Rectangle box1 = new Rectangle();
Rectangle box2 = new Rectangle(5.0, 10.0);
```
Rectangle Class Constructor Overload

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.

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.

Packanges 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.

The BankAccount Example

BankAccount.java
AccountTest.java

Overloaded Constructors
- BankAccount()
- BankAccount(double)
- BankAccount(double, double)

Overloaded deposit methods
- deposit(double)
- deposit(double, String)

Overloaded withdraw methods
- withdraw(double)
- withdraw(double, String)

Overloaded setBalance methods
- setBalance(double)
- setBalance(double, String)

Scope of Instance Fields

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