Chapter 4
Writing Classes

Java Software Solutions
Foundations of Program Design
Seventh Edition

John Lewis
William Loftus
Writing Classes

• We've been using predefined classes from the Java API. Now we will learn to write our own classes.

• Chapter 4 focuses on:
  – class definitions
  – instance data
  – encapsulation and Java modifiers
  – method declaration and parameter passing
  – constructors
  – graphical objects
  – events and listeners
  – buttons and text fields
Outline

Anatomy of a Class
Encapsulation
Anatomy of a Method
Graphical Objects
Graphical User Interfaces
Buttons and Text Fields
Writing Classes

• The programs we’ve written in previous examples have used classes **defined** in the Java standard class library

• Now we will begin to design programs that rely on classes that **we write ourselves**

• The class that contains the **main** method is just the starting point of a program

• True object-oriented programming is based on defining **classes** that represent **objects** with well-defined characteristics and functionality
# Examples of Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Attributes</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student</td>
<td>Name</td>
<td>Set address</td>
</tr>
<tr>
<td></td>
<td>Address</td>
<td>Set major</td>
</tr>
<tr>
<td></td>
<td>Major</td>
<td>Compute grade point average</td>
</tr>
<tr>
<td></td>
<td>Grade point average</td>
<td></td>
</tr>
<tr>
<td>Rectangle</td>
<td>Length</td>
<td>Set length</td>
</tr>
<tr>
<td></td>
<td>Width</td>
<td>Set width</td>
</tr>
<tr>
<td></td>
<td>Color</td>
<td>Set color</td>
</tr>
<tr>
<td>Aquarium</td>
<td>Material</td>
<td>Set material</td>
</tr>
<tr>
<td></td>
<td>Length</td>
<td>Set length</td>
</tr>
<tr>
<td></td>
<td>Width</td>
<td>Set width</td>
</tr>
<tr>
<td></td>
<td>Height</td>
<td>Set height</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute volume</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute filled weight</td>
</tr>
<tr>
<td>Flight</td>
<td>Airline</td>
<td>Set airline</td>
</tr>
<tr>
<td></td>
<td>Flight number</td>
<td>Set flight number</td>
</tr>
<tr>
<td></td>
<td>Origin city</td>
<td>Determine status</td>
</tr>
<tr>
<td></td>
<td>Destination city</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Current status</td>
<td></td>
</tr>
<tr>
<td>Employee</td>
<td>Name</td>
<td>Set department</td>
</tr>
<tr>
<td></td>
<td>Department</td>
<td>Set title</td>
</tr>
<tr>
<td></td>
<td>Title</td>
<td>Set salary</td>
</tr>
<tr>
<td></td>
<td>Salary</td>
<td>Compute wages</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute bonus</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute taxes</td>
</tr>
</tbody>
</table>
Classes and Objects

• Recall from our overview of objects in Chapter 1 that an object has state and behavior

• Consider a six-sided die (singular of dice)
  – It’s state can be defined as which face is showing
  – It’s primary behavior is that it can be rolled

• We represent a die by designing a class called Die that models this state and behavior
  – The class serves as the blueprint for a die object

• We can then instantiate as many die objects as we need for any particular program
Classes

- A class can contain data declarations and method declarations

```java
int size, weight;
char category;
```

Data declarations

Method declarations
Quick Check

• List some attributes and operations that might be defined for a class called PictureFrame that represents a picture frame.
Quick Check

• List some attributes and operations that might be defined for a class called PictureFrame that represents a picture frame.

• Attributes could include: `height`, `width`, `isEmpty` (boolean)
• Operations could include: `getHeight`, `setHeight`, `get_width`, `setWidth`, `isEmpty`, `fill`
Classes

• The **values** of the **data** define the state of an object created from the class

• For our **Die** class, we might declare an integer called **faceValue** that represents the current value showing on the face

• The **functionality** of the **methods** define the behaviors of the object

• One of the methods would “roll” the die by setting **faceValue** to a random number between one and six
Classes

• We’ll want to design the Die class so that it is a versatile and reusable resource

• Any given program will probably not use all operations of a given class

• See RollingDice.java
• See Die.java
public class RollingDice
{
    //-------------------------------------------------
    // Creates two Die objects and rolls them several times.
    //-------------------------------------------------
    public static void main (String[] args)
    {
        Die die1, die2;
        int sum;

        die1 = new Die();
        die2 = new Die();

        die1.roll();
        die2.roll();
        System.out.println("Die One: " + die1 + ", Die Two: " + die2);
    }
}
continue

die1.roll();
die2.setFaceValue(4);
System.out.println("Die One: " + die1 + ", Die Two: " + die2);

sum = die1.getFaceValue() + die2.getFaceValue();
System.out.println("Sum: " + sum);

sum = die1.roll() + die2.roll();
System.out.println("Die One: " + die1 + ", Die Two: " + die2);
System.out.println("New sum: " + sum);
}

```java
continue

die1.roll();
die2.setFaceValue(4);
System.out.println("Die One: "+die1+", Die Two: "+die2);
sum = die1.getFaceValue()+die2.getFaceValue();
System.out.println("Sum: "+sum);
sum = die1.roll()+die2.roll();
System.out.println("Die One: "+die1+", Die Two: "+die2);
System.out.println("New sum: "+sum);
```
public class Die
{
    private final int MAX = 6; // maximum face value

    private int faceValue; // current value showing on the die

    public Die()
    {
        faceValue = 1;
    }

    continue
// Rolls the die and returns the result.
public int roll()
{
    faceValue = (int)(Math.random() * MAX) + 1;
    return faceValue;
}

// Face value mutator.
public void setFaceValue (int value)
{
    faceValue = value;
}

// Face value accessor.
public int getFaceValue()
{
    return faceValue;
}
// Returns a string representation of this die.
//
public String toString()
{
    String result = Integer.toString(faceValue);
    return result;
}
The Die Class

• The Die class contains two data values
  – a constant MAX that represents the maximum face value
  – an integer faceValue that represents the current face value

• The roll method uses the random method of the Math class to determine a new face value

• There are also methods to explicitly set and retrieve the current face value at any time (accessors)
The toString Method

• It's good practice to define a `toString` method for a class

• The `toString` method must return a character string that represents the object in some way

• It is called automatically when an object is concatenated to a string or when it is passed to the `println` method

• It's also convenient for debugging problems
Quiz

Which of the following reserved words in Java is used to create an instance of a class?
A) class
B) public
C) public or private, either could be used
D) import
E) new
Quiz

Which of the following reserved words in Java is used to create an instance of a class?

A) class
B) public
C) public or private, either could be used
D) import
E) new

The reserved word "new" is used to instantiate an object, that is, to create an instance of a class. The statement new is followed by the name of the class. This calls the class' constructor. Example: Car x = new Car(); will create a new instance of a Car and set the variable x to it.
Exercise

• Write another implementation of the roll() method that uses a random number generator object of the class Random
public int roll() {
    Random rGen = new Random();
    faceValue = rGen.nextInt(6) + 1;
    return faceValue;
}
Quiz

• Write a method called cube that accepts one integer parameter and returns that value raised to the third power.
Quiz

• Write a method called cube that accepts one integer parameter and returns that value raised to the third power.

```java
public int cube (int num) {
    return (int) Math.pow (num, 3);
}
```
Quick Check

• For each of the following pairs, which represents a class and which represents an object of that class?
  • **Superhero, Superman**
  • **Justin, Person**
  • **Rover, Pet**
  • **Magazine, Time**
  • **Christmas, Holiday**
Quick Check

• For each of the following pairs, which represents a class and which represents an object of that class?

• **Superhero, Superman**
  - *Class: Superhero, Object: Superman*

• **Justin, Person**
  - *Class: Person, Object: Justin*

• **Rover, Pet**
  - *Class: Pet, Object: Rover*

• **Magazine, Time**
  - *Class: Magazine, Object: Time*

• **Christmas, Holiday**
  - *Class: Holiday, Object: Christmas*
Constructors

• As mentioned previously, a constructor is used to set up an object when it is initially created

• A constructor has the same name as the class

• The Die constructor is used to set the initial face value of each new die object to one

• We examine constructors in more detail later in this chapter
Quiz

```java
public class Student {
    private String name, major;
    private double gpa;
    private int hours;
    public Student(String newName, String newMajor, double newGPA, int newHours) {
        name = newName;
        major = newMajor;
        gpa = newGPA;
        hours = newHours;
    }
}
```

Which of the following could be used to instantiate a new Student s1?

A) `Student s1 = new Student( );`  
B) `s1 = new Student( );`  
C) `Student s1 = new Student("Jane Doe", "Computer Science", 3.333, 33);`  
D) `new Student s1 = ("Jane Doe", "Computer Science", 3.333, 33);`  
E) `new Student(s1);`
public class Student {
    private String name;
    private String major;
    private double gpa;
    private int hours;
    public Student(String newName, String newMajor, double newGPA, int newHours) {
        name = newName;
        major = newMajor;
        gpa = newGPA;
        hours = newHours;
    }
}

Which of the following could be used to instantiate a new Student s1?
A) Student s1 = new Student( );
B) s1 = new Student( );
C) Student s1 = new Student("Jane Doe", "Computer Science", 3.333, 33);
D) new Student s1 = ("Jane Doe", "Computer Science", 3.333, 33);
E) new Student(s1);
Data Scope

• The **scope** of data is the area in a program in which that data can be referenced (used):
  – Data **declared at the class level** can be referenced by all methods in that class
  – Data declared **within a method** can be used only in that method - it is called *local data*

• In the Die class, the variable **result** is declared inside the **toString** method -- it is local to that method and cannot be referenced anywhere else
Instance Data

• A variable declared at the **class level** (such as `faceValue`) is called **instance data**

• Each instance (object) has its own instance variable

• A class declares the type of the data, but it does not reserve memory space for it

• Each time a `Die` object is created, a new `faceValue` variable is created as well

• *The objects of a class share the method definitions, but each object has its own data space*

• That's the only way two objects can have different states
Instance Data

• We can depict the two Die objects from the RollingDice program as follows:

![Diagram showing two Die objects with their face values](image.png)

Each object maintains its own faceValue variable, and thus its own state.
Quiz

• Which of the methods defined for the Die class can change the state of a Die object?
Quiz

• Which of the methods defined for the Die class can change the state of a Die object?

• Roll and setFaceValue
Quiz

Write a method of the Die class called cube that reads the current face value of the die and set it to the third power of the current value, modulo 6, plus 1.
Write a method of the Die class called cube that reads the current face value of the die and set it to the third power of the current value, modulo 6, plus 1.

```java
public void cube ()
{
    setFaceValue((int) Math.pow (getFaceValue(), 3) % MAX + 1);
}
```
Quiz

Instance data for a Java class
A) are limited to primitive types (e.g., int, float, char)
B) are limited to Strings
C) are limited to objects (e.g., Strings, classes defined by other programmers)
D) may be primitive types or objects, but objects must be defined to be private
E) may be primitive types or objects
Quiz

Instance data for a Java class
A) are limited to primitive types (e.g., int, float, char)
B) are limited to Strings
C) are limited to objects (e.g., Strings, classes defined by other programmers)
D) may be primitive types or objects, but objects must be defined to be private
E) may be primitive types or objects

The instance data are the entities that make up the class and may be any type available whether primitive or object, and may be public or private. By using objects as instance data, it permits the class to be built upon other classes.
Quiz

Data members of the Die class are defined as:

```java
public class Die {
    private final int MAX = 6;
    private int faceValue;
    ...
}
```

Could you have defined them as follows?

```java
public class Die {
    private static final int MAX = 6;
    private int faceValue;
    ...
}
```
Quiz

Data members of the Die class are defined as:

```java
public class Die {
    private final int MAX = 6;
    private int faceValue;
    ...
}
```

Could you have defined them as follows?

```java
public class Die {
    private static final int MAX = 6;
    private int faceValue;
    ...
}
```

Yes, and normally the second form is preferred. If an instance data member cannot be modified (final) then there is only one value for this field and this could be stored in the class (static).
UML Diagrams

• UML stands for the *Unified Modeling Language*

• *UML diagrams* show relationships among classes and objects

• A UML *class diagram* consists of one or more classes, each with sections for the class name, attributes (data), and operations (methods)

• Lines between classes represent *associations*

• A dotted arrow shows that one class *uses* the other (calls its methods)
UML Class Diagrams

- A UML class diagram for the `RollingDice` program:

```
RollingDice
   main (args : String[]) : void

Die
   faceValue : int
   roll() : int
   setFaceValue (int value) : void
   getFaceValue() : int
   toString() : String
```

**Dependency**: is a relationship which indicates that one class depends on another because it uses it at some point of time. One class depends on another if the latter is a local variable or parameter variable of a method of the former.
Quick Check

What is the relationship between a class and an object?
Quick Check

What is the relationship between a class and an object?

A class is the definition/pattern/blueprint of an object. It defines the data that will be managed by an object but doesn't reserve memory space for it. Multiple objects can be created from a class, and each object has its own copy of the instance data.
Quick Check

Where is instance data declared?

What is the scope of instance data?

What is local data?
Quick Check

Where is **instance data** declared?

At the class level.

What is the **scope** of instance data?

It can be referenced in any method of the class.

What is **local data**?

Local data is declared within a method, and is only accessible in that method.
Outline

Anatomy of a Class
Encapsulation
Anatomy of a Method
Graphical Objects
Graphical User Interfaces
Buttons and Text Fields

Copyright © 2012 Pearson Education, Inc.
Encapsulation

• We can take one of two views of an object:
  
  – **internal** - the details of the variables and methods of the class that defines it
  
  – **external** - the services that an object provides and how the object interacts with the rest of the system

• From the external view, an object is an *encapsulated* entity, providing a set of specific services

• These services define the *interface* to the object
A Software Object
Encapsulation

- An encapsulated object can be thought of as a black box -- its inner workings are hidden from the client.
- The client invokes the interface methods and they manage the instance data.
Encapsulation

- One object (called the *client*) may use another object for the *services* it provides.

- The client of an object may request its services (call its methods), but it **should not** have to be aware of **how those services are accomplished**.

- Any *changes* to the object's state (its variables) should be made by that object's *methods*.

- We should make it difficult, if not **impossible**, for a *client* to access an object’s *variables* directly.

- That is, an object should be **self-governing**.
Visibility Modifiers

• In Java, we accomplish encapsulation through the appropriate use of *visibility modifiers*

• A *modifier* is a Java reserved word that specifies particular characteristics of a method or data

• We've used the *final* modifier to define constants

• Java has three visibility modifiers: *public*, *protected*, and *private*

• The *protected* modifier involves inheritance, which we will discuss later
Visibility Modifiers

• Members of a class that are declared with \textit{public} visibility can be referenced anywhere.

• Members of a class that are declared with \textit{private} visibility can be referenced only within that class.

• Members declared without a visibility modifier have \textit{default} visibility and can be referenced by any class in the same package.

• An overview of all Java modifiers is presented in Appendix E.
Visibility Modifiers

• **Public variables violate encapsulation** because they allow the client to modify the values directly.

• Therefore **instance variables** should **not** be declared with **public** visibility.

• It is acceptable to give a **constant public** visibility, which allows it to be used outside of the class.

• Public constants do not violate encapsulation because, although the client can access it, its value cannot be changed.
Visibility Modifiers

- **Methods** that provide the object's services are declared with **public** visibility so that they can be invoked by clients.

- **Public methods** are also called **service methods**.

- A method created simply to assist a service method is called a **support method**.

- Since a **support** method is not intended to be called by a client, it should not be declared with **public** visibility.
# Visibility Modifiers

<table>
<thead>
<tr>
<th>Variables</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>public</strong></td>
<td><strong>Provide services to clients</strong></td>
</tr>
<tr>
<td>Violate encapsulation</td>
<td><strong>Support other methods in the class</strong></td>
</tr>
<tr>
<td><strong>private</strong></td>
<td><strong>Enforce encapsulation</strong></td>
</tr>
</tbody>
</table>

Copyright © 2012 Pearson Education, Inc.
Accessors and Mutators

• If instance data is private, a class should provide services to access and modify data values:
  – An **accessor method** returns the current value of a variable
  – A **mutator method** changes the value of a variable

• The names of accessor and mutator methods take the form `getX` and `setX`, respectively, where `X` is the name of the data

• They are sometimes called “getters” and “setters”
Mutator Restrictions

• The use of mutators gives the class designer the ability to restrict a client’s options to modify an object’s state

• A mutator is often designed so that the values of variables can be set only within particular limits

• For example, the setFaceValue mutator of the Die class should restrict the value to the valid range (1 to MAX)

• We’ll see in Chapter 5 how such restrictions can be implemented
Quick Check

Why was the `faceValue` variable declared as `private` in the `Die` class?

Why is it ok to declare `MAX` as `public` in the `Die` class?
Quick Check

Why was the `faceValue` variable declared as `private` in the `Die` class?

By making it private, each `Die` object controls its own data and allows it to be modified only by the well-defined operations it provides.

Why is it ok to declare `MAX` as `public` in the `Die` class?

`MAX` is a constant. Its value cannot be changed. Therefore, there is no violation of encapsulation.
Quiz

• To define a class that will represent a car, which of the following definitions is most appropriate?

A) private class car
B) public class car
C) public class Car
D) public class CAR
E) private class Car
Quiz

• To define a class that will represent a car, which of the following definitions is most appropriate?

A) private class car
B) public class car
C) public class Car
D) public class CAR
E) private class Car

• Classes should be defined to be public so that they can be accessed by other classes. Class names should start with a capital letter and be lower case except for the beginning of each new word.
Quiz

Visibility modifiers include
A) public, private
B) public, private, protected
C) public, private, protected, final
D) public, protected, final, static
E) public, private, protected, static
Quiz

Visibility modifiers include
A) public, private
B) public, private, protected
C) public, private, protected, final
D) public, protected, final, static
E) public, private, protected, static

Public, private, protected control the visibility of variables and methods. Final controls whether a variable, method, or class can be further changed or overridden, not visibility. Static controls whether a variable or method is associated with instances of a class or the class itself.
Quiz

• Define the class Rectangle, with length, width and color instance data. Choose an appropriate type for this data.

• Define getter and setters for the class.
import java.awt.Color;

public class Rectangle {
    private int length = 0;
    private int width = 0;
    private Color color = new Color(0,0,0);

    public int getLength() {
        return length;
    }

    public void setLength(int lengthIn) {
        length = lengthIn;
    }

    public int getWidth() {
        return width;
    }

    public void setWidth(int widthIn) {
        width = widthIn;
    }

    public Color getColor() {
        return color;
    }

    public void setColor(Color colorIn) {
        color = colorIn;
    }
}

Outline

Anatomy of a Class
Encapsulation
Anatomy of a Method
Graphical Objects
Graphical User Interfaces
Buttons and Text Fields
Method Declarations

• A *method declaration* specifies the code that will be executed when the method is invoked (called)

• When a method is **invoked**, the flow of control **jumps** to the **method** and executes its code

• When **complete**, the flow **returns** to the place **where the method was called** and **continues**

• The invocation may or may not return a **value**, depending on how the method is defined
Method Control Flow

- If the called method is in the same class, only the method name is needed
Method Control Flow

- The called method is often part of another class or object
Exercise

• Modify the roll() method of the Die class so that after having randomly changed the faceValue of the Die object it uses the method toString of the object to print the new value of the Die.
Exercise

• Modify the roll() method of the Die class so that after having randomly changed the faceValue of the Die object it uses the method toString of the object to print the new value of the Die.

```java
public int roll() {
    faceValue = (int) (Math.random() * MAX) + 1;
    System.out.println(toString());
    return faceValue;
}
```
Quiz

- Consider a sequence of method invocations as follows: main calls m1, m1 calls m2, m2 calls m3 and then m2 calls m4. If m4 has just terminated, what method will resume execution?

A) m1
B) m2
C) m3
D) main
Quiz

• Consider a sequence of method invocations as follows: main calls m1, m1 calls m2, m2 calls m3 and then m2 calls m4. If m4 has just terminated, what method will resume execution?

A) m1  
B) m2  
C) m3  
D) main

Once a method terminates, control resumes with the method that called that method. In this case, m2 calls m4, so that when m4 terminates, m2 is resumed.
Method Header

A method declaration begins with a *method header*

```
char calc (int num1, int num2, String message)
```

- **Method Name**: The name of the method.
- **Parameter List**: Specifies the type and name of each parameter.
- **Return Type**: The type of value the method returns.

The parameter list specifies the type and name of each parameter.

The name of a parameter in the method declaration is called a *formal parameter*.
Method Body

• The method header is followed by the *method body*

```java
char calc (int num1, int num2, String message)
{
    int sum = num1 + num2;
    char result = message.charAt (sum);

    return result;
}
```

The return expression must be consistent with the return type

sum and result are local data

They are created each time the method is called, and are destroyed when it finishes executing.
The return Statement

• The *return type* of a method indicates the type of value that the method sends back to the calling location

• A method that does not return a value has a *void* return type

• A *return statement* specifies *when* and *what value* will be returned

  ```java
  return expression;
  ```

• *expression* must conform to the return type
Quiz

• Write a method called randomColor that creates and returns a Color object that represents a random color. Recall that a Color object can be defined (constructed) by (passing) three integer values between 0 and 255 (to a constructor method) representing the contributions of red, green, and blue (its RGB value).
Quiz

• Write a method called randomColor that creates and returns a Color object that represents a random color. Recall that a Color object can be defined by three integer values between 0 and 255 representing the contributions of red, green, and blue (its RGB value).

```java
final int MAX = 256;
public Color randomColor () {
    Random generator = new Random();
    int randRed = generator.nextInt(MAX);
    int randGreen = generator.nextInt(MAX);
    int randBlue = generator.nextInt(MAX);
    return new Color(randRed, randGreen, randBlue);
}
```
Quiz

• Suppose you have a class called Child with an instance data value called age. Write a getter method and a setter method for age.
Quiz

• Suppose you have a class called Child with an instance data value called age. Write a getter method and a setter method for age.

```java
public int getAge() {
    return age;
}

public void setAge(int newAge) {
    age = newAge;
}
```
Parameters

- When a method is called, the *actual parameters* in the invocation are copied into the *formal parameters* in the method header.

```java
char calc (int num1, int num2, String message)
{
    int sum = num1 + num2;
    char result = message.charAt (sum);
    return result;
}
```

```java
ch = obj.calc (25, count, "Hello");
```
Local Data

• As we’ve seen, **local variables** can be declared inside a method

• The **formal parameters** of a method create **automatic local variables** when the method is invoked

• When the method finishes, all local variables are destroyed (including the formal parameters)

• Keep in mind that instance variables, declared at the class level, exists as long as the object exists
• Consider a method defined with the header: 
  \texttt{public void foo(int a, int b). Which of the following method calls is legal?}
  
  A) \texttt{foo(0, 0.1);} 
  B) \texttt{foo(0 / 1, 2 * 3);} 
  C) \texttt{foo(0);} 
  D) \texttt{foo( );} 
  E) \texttt{foo(1 + 2, 3 * 0.1);}
Quiz

Consider a method defined with the header:

```java
public void foo(int a, int b);
```

Which of the following method calls is legal?

A) `foo(0, 0.1);`
B) `foo(0 / 1, 2 * 3);`
C) `foo(0);`
D) `foo( );`
E) `foo(1 + 2, 3 * 0.1);`

The only legal method call is one that passes two int parameters. In the case of answer B, 0 / 1 is an int division (equal to 0) and 2 * 3 is an int multiplication. So this is legal. The answers A and E contain two parameters, but the second of each is a double. The answers for C and D have the wrong number of parameters.
Quiz

public class Student {
    private String name, major;
    private double gpa;
    private int hours;
    public Student(String newName, String newMajor, double newGPA, int newHours) {
        name = newName;
        major = newMajor;
        gpa = newGPA;
        hours = newHours;
    }
}

Write a method that will updates the Student's number of credit hours. This method will receive a number of credit hours and add these to the Student's current hours.
public class Student {
    private String name, major;
    private double gpa;
    private int hours;
    public Student(String newName, String newMajor,
                   double newGPA, int newHours) {
        name = newName;
        major = newMajor;
        gpa = newGPA;
        hours = newHours;
    }

    Write a method that will updates the Student's number of credit hours. This method will receive a number of credit hours and add these to the Student's current hours.

    public void updateHours (int moreHours) {
        hours += moreHours;
    }
}
Bank Account Example

• Let’s look at another example that demonstrates the implementation details of classes and methods

• We’ll represent a bank account by a class named Account

• Its state can include the account number, the current balance, and the name of the owner

• An account’s behaviors (or services) include deposits and withdrawals, and adding interest
Driver Programs

• A *driver program* drives the use of other, more interesting parts of a program

• Driver programs are often used to test other parts of the software

• The `Transactions` class contains a `main method` that drives the use of the `Account` class, exercising its services

• See `Transactions.java`

• See `Account.java`
public class Transactions {
    //------------------------------------------------------------------
    //  Creates some bank accounts and requests various services.
    //------------------------------------------------------------------
    public static void main (String[] args) {
        Account acct1 = new Account("Ted Murphy", 72354, 102.56);
        Account acct2 = new Account("Jane Smith", 69713, 40.00);
        Account acct3 = new Account("Edward Demsey", 93757, 759.32);

        acct1.deposit (25.85);

        double smithBalance = acct2.deposit (500.00);
        System.out.println("Smith balance after deposit: " +
                          smithBalance);
        continue
    }
}
continue

    System.out.println ("Smith balance after withdrawal: " + 
            acct2.withdraw (430.75, 1.50));

    acct1.addInterest();
    acct2.addInterest();
    acct3.addInterest();

    System.out.println ();
    System.out.println (acct1);
    System.out.println (acct2);
    System.out.println (acct3);
}
System.out.println("Smith balance after deposit: 540.0");
System.out.println("Smith balance after withdrawal: 107.75");
acct2.addInterest();
acct1.addInterest();
acct3.addInterest();
System.out.println();
System.out.println(acct1);
System.out.println(acct2);
System.out.println(acct3);
continue

72354   Ted Murphy      $132.90
69713   Jane Smith      $111.52
93757   Edward Demsey   $785.90
import java.text.NumberFormat;

public class Account {
    private final double RATE = 0.035;  // interest rate of 3.5%

    private long acctNumber;
    private double balance;
    private String name;

    public Account (String owner, long account, double initial) {
        name = owner;
        acctNumber = account;
        balance = initial;
    }
}

continue
public double deposit (double amount)
{
    balance = balance + amount;
    return balance;
}

public double withdraw (double amount, double fee)
{
    balance = balance - amount - fee;
    return balance;
}
public double addInterest ()
{
    balance += (balance * RATE);
    return balance;
}

public double getBalance ()
{
    return balance;
}

public String toString ()
{
    NumberFormat fmt = NumberFormat.getCurrencyInstance();
    return (acctNumber + "\t" + name + "\t" + fmt.format(balance));
}
Quiz

• Can the instance variables accNumber and name be final?
Quiz

• Can the instance variables accNumber and name be final?

• YES, because after the object is instantiated with a constructor these two variables are never changed.
Bank Account Example

acct1

acctNumber 72354
balance 102.56
name "Ted Murphy"

acct2

acctNumber 69713
balance 40.00
name "Jane Smith"
Bank Account Example

- There are some improvements that can be made to the `Account` class
- Formal getters and setters could have been defined for all data
- The design of some methods could also be more robust, such as verifying that the `amount` parameter to the `withdraw` method is positive
Constructors Revisited

• Note that a constructor has no return type specified in the method header, not even `void`.

• A common error is to put a return type on a constructor, which makes it a “regular” method that happens to have the same name as the class.

• The programmer does not have to define a constructor for a class.

• Each class has a `default constructor` that accepts no parameters.
  – if no other constructors were defined!
Quiz

• A variable whose scope is restricted to the method where it was declared is known as a

A) parameter
B) global variable
C) local variable
D) public instance data
E) private instance data
Quiz

• A variable whose scope is restricted to the method where it was declared is known as a
  A) parameter
  B) global variable
  C) local variable
  D) public instance data
  E) private instance data

Local variables are those that are "local" to the method in which they have been declared, that is, they are accessible only inside that method. Global variables are those that are accessible from anywhere, while parameters are the variables passed into a method. Instance data can be thought of as global variables for an entire object.
Quick Check

How do we express which Account object's balance is updated when a deposit is made?
Quick Check

How do we express which Account object's balance is updated when a deposit is made?

Each account is referenced by an object reference variable:

```
Account myAcct = new Account(...);
```

and when a method is called, you call it through a particular object:

```
myAcct.deposit(50);
```
Quiz

• All methods are invoked through (or on) a particular object? What is the exception to that rule?
Quiz

• All methods are invoked through (or on) a particular object? What is the exception to that rule?

• No – the exception is the invocation of a static method on a class (e.g. Math.pow(2,5))
Outline

Anatomy of a Class
Encapsulation
Anatomy of a Method
Graphical Objects
Graphical User Interfaces
Buttons and Text Fields
Graphical Objects

- Some objects contain information that determines how the object should be represented visually.
- Most GUI components are graphical objects.
- We can have some effect on how components get drawn.
- We did this in Chapter 2 when we defined the `paint` method of an applet.
- Let's look at some other examples of graphical objects.
Smiling Face Example

• The SmilingFace program draws a face by defining the paintComponent method of a panel

• See SmilingFace.java
• See SmilingFacePanel.java

• The main method of the SmilingFace class instantiates a SmilingFacePanel and displays it

• The SmilingFacePanel class is derived from the JPanel class using inheritance
import javax.swing.JFrame;

public class SmilingFace {

    public static void main (String[] args) {
        JFrame frame = new JFrame ("Smiling Face");
        frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

        SmilingFacePanel panel = new SmilingFacePanel();

        frame.getContentPane().add(panel);
        frame.pack();
        frame.setVisible(true);
    }
}
import javax.swing.JFrame;

public class SmilingFace {

    //-------------------------------
    // Creates the main frame of the program.
    //-------------------------------
    public static void main(String[] args) {
        JFrame frame = new JFrame("Smiling Face");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        SmilingFacePanel panel = new SmilingFacePanel();

        frame.getContentPane().add(panel);

        frame.pack();
        frame.setVisible(true);
    }
}
import javax.swing.JPanel;
import java.awt.*;

public class SmilingFacePanel extends JPanel
{
    private final int BASEX = 120, BASEY = 60; // base point for head

    public SmilingFacePanel ()
    {
        setBackground (Color.blue);
        setPreferredSize (new Dimension(320, 200));
        setFont (new Font("Arial", Font.BOLD, 16));
    }

    continue
public void paintComponent (Graphics page)
{
    super.paintComponent (page);

    page.setColor (Color.yellow);
    page.fillOval (BASEX, BASEY, 80, 80); // head
    page.fillOval (BASEX-5, BASEY+20, 90, 40); // ears

    page.setColor (Color.black);
    page.drawOval (BASEX+20, BASEY+30, 15, 7); // eyes
    page.drawOval (BASEX+45, BASEY+30, 15, 7);

    page.fillOval (BASEX+25, BASEY+31, 5, 5); // pupils
    page.fillOval (BASEX+50, BASEY+31, 5, 5);

    page.drawArc (BASEX+20, BASEY+25, 15, 7, 0, 180); // eyebrows
    page.drawArc (BASEX+45, BASEY+25, 15, 7, 0, 180);

    page.drawArc (BASEX+35, BASEY+40, 15, 10, 180, 180); // nose
    page.drawArc (BASEX+20, BASEY+50, 40, 15, 180, 180); // mouth
}

continue
continue

    page.setColor (Color.white);
    page.drawString ("Always remember that you are unique!",
                        BASEX-105, BASEY-15);
    page.drawString ("Just like everyone else.", BASEX-45, BASEY+105);
}
Smiling Face Example

• Every Swing component has a `paintComponent` method

• The `paintComponent` method accepts a `Graphics` object that represents the graphics context for the panel

• We define the `paintComponent` method to draw the face with appropriate calls to the `Graphics` methods

• Note the difference between drawing on a panel and adding other GUI components to a panel
Splat Example

• The Splat example is structured a bit differently

• It draws a set of colored circles on a panel, but each circle is represented as a separate object that maintains its own graphical information

• The `paintComponent` method of the panel "asks" each circle to draw itself

• See `Splat.java`
• See `SplatPanel.java`
• See `Circle.java`
public class Splat {
    // Presents a collection of circles.
    public static void main (String[] args) {
        JFrame frame = new JFrame("Splat");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(new SplatPanel());
        frame.pack();
        frame.setVisible(true);
    }
}
//**************************
// Splat.java       Author: Lewis/Loftus
//
// Demonstrates the use of graphical objects.
//**************************
import javax.swing.*;
import java.awt.*;

public class Splat{
    //----------------------
    // Presents a collection of circles.
    //----------------------
    public static void main (String[] args){
        JFrame frame = new JFrame("Splat");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.getContentPane().add(new SplatPanel());

        frame.pack();
        frame.setVisible(true);
    }
}
import javax.swing.*;
import java.awt.*;

public class SplatPanel extends JPanel {
    private Circle circle1, circle2, circle3, circle4, circle5;

    // Constructor: Creates five Circle objects.
    public SplatPanel() {
        circle1 = new Circle (30, Color.red, 70, 35);
        circle2 = new Circle (50, Color.green, 30, 20);
        circle3 = new Circle (100, Color.cyan, 60, 85);
        circle4 = new Circle (45, Color.yellow, 170, 30);
        circle5 = new Circle (60, Color.blue, 200, 60);

        setPreferredSize (new Dimension(300, 200));
        setBackground (Color.black);
    }
}

continue
continue

// Draws this panel by requesting that each circle draw itself.

public void paintComponent (Graphics page)
{
    super.paintComponent(page);

circle1.draw(page);
circle2.draw(page);
circle3.draw(page);
circle4.draw(page);
circle5.draw(page);
}

import java.awt.*;

public class Circle
{
    private int diameter, x, y;
    private Color color;

    public Circle (int size, Color shade, int upperX, int upperY)
    {
        diameter = size;
        color = shade;
        x = upperX;
        y = upperY;
    }
}

continue
public void draw (Graphics page)
{
    page.setColor (color);
    page.fillOval (x, y, diameter, diameter);
}

public void setDiameter (int size)
{
    diameter = size;
}

public void setColor (Color shade)
{
    color = shade;
}
public void setX (int upperX)
{
    x = upperX;
}

public void setY (int upperY)
{
    y = upperY;
}

public int getDiameter ()
{
    return diameter;
}
//-----------------------------------------------------------------
//  Color accessor.
//-----------------------------------------------------------------
public Color getColor ()
{
    return color;
}

//-----------------------------------------------------------------
//  X accessor.
//-----------------------------------------------------------------
public int getX ()
{
    return x;
}

//-----------------------------------------------------------------
//  Y accessor.
//-----------------------------------------------------------------
public int getY ()
{
    return y;
}
Outline

Anatomy of a Class
Encapsulation
Anatomy of a Method
Graphical Objects
Graphical User Interfaces
Buttons and Text Fields
Graphical User Interfaces

- A Graphical User Interface (GUI) in Java is created with at least three kinds of objects:
  - components, events, and listeners

- We've previously discussed components, which are objects that represent screen elements:
  - labels, buttons, text fields, menus, etc.

- Some components are containers that hold and organize other components:
  - frames, panels, applets, dialog boxes
Events

• An *event* is an **object** that represents some activity to which we may want to respond

• For example, we may want our program to perform some action when the following occurs:
  – the mouse is moved
  – the mouse is dragged
  – a mouse button is clicked
  – a graphical button is pressed
  – a keyboard key is pressed
  – a timer expires
Events and Listeners

• The Java API contains several classes that represent typical events

• Components, such as a graphical button, generate (or fire) an event when it occurs

• We set up a listener object to respond to an event when it occurs

• We can design listener objects to take whatever actions are appropriate when an event occurs
Events and Listeners

A component object generates an event

When the event occurs, the component calls the appropriate method of the listener, passing an object that describes the event
GUI Development

• To create a Java program that uses a GUI we must:
  – instantiate and set up the necessary components
  – implement listener classes for any events we care about
  – establish the relationship between listeners and the components that generate the corresponding events

• Let's now explore some new components and see how this all comes together
Outline

Anatomy of a Class
Encapsulation
Anatomy of a Method
Graphical Objects
Graphical User Interfaces
Buttons and Text Fields
Buttons

• A *push button* is defined by the `JButton` class

• It generates an *action event*

• The `PushCounter` example displays a push button that increments a counter each time it is pushed

• See `PushCounter.java`

• See `PushCounterPanel.java`
import javax.swing.JFrame;

public class PushCounter {

    public static void main (String[] args) {
        JFrame frame = new JFrame("Push Counter");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.getContentPane().add(new PushCounterPanel());

        frame.pack();
        frame.setVisible(true);
    }
}
import javax.swing.JFrame;

public class PushCounter {
    // Demonstrates a graphical user interface and an event listener.
    // PushCounter.java       Authors: Lewis/Loftus
    //                      //
    //                   //
    // Creates the main program frame.
    //                   //
    public static void main (String[] args) {
        JFrame frame = new JFrame("Push Counter");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.getContentPane().add(new PushCounterPanel());

        frame.pack();
        frame.setVisible(true);
    }
}
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class PushCounterPanel extends JPanel
{
    private int count;
    private JButton push;
    private JLabel label;

    // Constructor: Sets up the GUI.
    public PushCounterPanel()
    {
        count = 0;

        push = new JButton("Push Me!");
        push.addActionListener(new ButtonListener());
    }

    continue
continue

label = new JLabel("Pushes: " + count);

add (push);
add (label);

setPreferredSize (new Dimension(300, 40));
setBackground (Color.cyan);
}

//*****************************************************************
// Represents a listener for button push (action) events.
//*****************************************************************
private class ButtonListener implements ActionListener
{
    //.....................................................................
    // Updates the counter and label when the button is pushed.
    //.....................................................................
    public void actionPerformed (ActionEvent event)
    {
        count++;
        label.setText("Pushes: " + count);
    }
}
Push Counter Example

- The components of the GUI are the button, a label to display the counter, a panel to organize the components, and the main frame.

- The `PushCounterPanel` class represents the panel used to display the button and label.

- The `PushCounterPanel` class is derived from `JPanel` using inheritance.

- The constructor of `PushCounterPanel` sets up the elements of the GUI and initializes the counter to zero.
Push Counter Example

• The `ButtonListener` class is the listener for the action event generated by the button.

• It is implemented as an `inner class`, which means it is defined within the body of another class.

• That facilitates the communication between the listener and the GUI components (can access count instance variable of `PushCounterPanel`).

• Inner classes should only be used in situations where there is an intimate relationship between the two classes and the inner class is not needed in any other context.
Push Counter Example

- Listener classes are written by implementing a listener interface

- The ButtonListener class implements the ActionListener interface

- An interface is a list of methods that the implementing class must define

- The only method in the ActionListener interface is the actionPerformed method

- The Java API contains interfaces for many types of events

- We discuss interfaces in more detail in Chapter 6
Push Counter Example

• The PushCounterPanel constructor:
  – instantiates the ButtonListener object
  – establishes the relationship between the button and the listener by the call to addActionListener

• When the user presses the button, the button component creates an ActionEvent object and calls the actionPerformed method of the listener

• The actionPerformed method increments the counter and resets the text of the label
Quick Check

Which object in the Push Counter example generated the event?

What did it do then?
Which object in the Push Counter example generated the event?

The button component generated the event.

What did it do then?

It called the `actionPerformed` method of the listener object that had been registered with it.
Quiz

Consider the "Push Me!" program shown before. As given the push counter starts at zero and is incremented each time the "Push Me!" button is pressed. Let's say that you want to add a class called ResetListener that resets the count to zero. Write the ResetListener class.
Quiz

Consider the "Push Me!" program shown before. As given the push counter starts at zero and is incremented each time the "Push Me!" button is pressed. Let's say that you want to add a class called ResetListener that resets the count to zero. Write the ResetListener class.

```java
private class ResetListener implements ActionListener {
    public void actionPerformed (ActionEvent event) {
        count = 0;
    }
}
```
Text Fields

- Let's look at another GUI example that uses another type of component.

- A *text field* allows the user to enter one line of input.

- If the cursor is in the text field, the text field object generates an action event when the enter key is pressed.

- See `Fahrenheit.java`
- See `FahrenheitPanel.java`
import javax.swing.JFrame;

public class Fahrenheit
{
    //------------------------------------------------------------------------------
    //  Creates and displays the temperature converter GUI.
    //------------------------------------------------------------------------------
    public static void main (String[] args)
    {
        JFrame frame = new JFrame ("Fahrenheit");
        frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

        FahrenheitPanel panel = new FahrenheitPanel();

        frame.getContentPane().add(panel);
        frame.pack();
        frame.setVisible(true);
    }
}
import javax.swing.JFrame;

public class Fahrenheit {
    //-----------------------------------------------------------------
    // Creates and displays the temperature converter GUI.
    //-----------------------------------------------------------------
    public static void main (String[] args) {
        JFrame frame = new JFrame ("Fahrenheit");
        frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

        FahrenheitPanel panel = new FahrenheitPanel();

        frame.getContentPane().add(panel);
        frame.pack();
        frame.setVisible(true);
    }
}
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class FahrenheitPanel extends JPanel
{
    private JLabel inputLabel, outputLabel, resultLabel;
    private JTextField fahrenheit;

    // Constructor: Sets up the main GUI components.
    public FahrenheitPanel()
    {
        inputLabel = new JLabel("Enter Fahrenheit temperature:");
        outputLabel = new JLabel("Temperature in Celsius: ");
        resultLabel = new JLabel("---");

        fahrenheit = new JTextField (5);
        fahrenheit.addActionListener (new TempListener());
    }

    // Method to convert Fahrenheit to Celsius
    public double convertFahrenheitToCelsius()
    {...}

    // Method to display Celsius temperature
    public void displayCelsius()
    {...}
}

class TempListener extends ActionListener
{
    @Override
    public void actionPerformed(ActionEvent e)
    {...}
}
null

```java
continue

add (inputLabel);
add (fahrenheit);
add (outputLabel);
add (resultLabel);

setPreferredSize (new Dimension(300, 75));
setBackground (Color.yellow);
}
//*****************************************************************
// Represents an action listener for the temperature input field.
//*****************************************************************
private class TempListener implements ActionListener
{
    //--------------------------------------------------------------------------
    // Performs the conversion when the enter key is pressed in
    // the text field.
    //--------------------------------------------------------------------------
    public void actionPerformed (ActionEvent event)
    {
        int fahrenheitTemp, celsiusTemp;

        String text = fahrenheit.getText();
```
continue

    fahrenheitTemp = Integer.parseInt (text);
    celsiusTemp = (fahrenheitTemp - 32) * 5/9;

    resultLabel.setText (Integer.toString (celsiusTemp));
  }
}
}
Fahrenheit Example

• Like the PushCounter example, the GUI is set up in a separate panel class

• The TempListener inner class defines the listener for the action event generated by the text field

• The FahrenheitPanel constructor instantiates the listener and adds it to the text field

• When the user types a temperature and presses enter, the text field generates the action event and calls the actionPerformed method of the listener
Summary

- Chapter 4 focused on:
  - class definitions
  - instance data
  - encapsulation and Java modifiers
  - method declaration and parameter passing
  - constructors
  - graphical objects
  - events and listeners
  - buttons and text fields
Exercise

• Write a constructor of the class Rectangle that we have defined previously in the lecture
Exercise

• Write a constructor of the class Rectangle that we have defined previously in the lecture

```java
public Rectangle(int lengthIn, int widthIn,
                 Color colorIn) {
    length = lengthIn;
    width = widthIn;
    color = colorIn;
}
```
Exercise

• Write two methods of the class Rectangle that compute the area and the perimeter
Exercise

• Write two methods of the class Rectangle that compute the area and the perimeter

```java
public int perimeter() {
    return 2 * (length + width);
}

public int area() {
    return length * width;
}
```
Exercise

• Write a new method of the Account class that executes a withdraw from the account: reduces the balance of the account by an amount plus a tax, which is a percentage of the amount.
• The method has therefore two parameters: amount and tax
• Choose the appropriate types for the parameters
• The method returns the new balance.
Exercise

public double withdraw2 (double amount, double fee)
{
    balance -= (1 + fee) * amount;
    return balance;
}
Exercise

• For each of the following pairs, which represents a class and which represents an object of that class?
  – Superhero, Superman
  – Justin, Person
  – Rover, Pet
  – Magazine, Time
  – Christmas, Holiday
Exercise

• List some attributes and operations that might be defined for a class called Meeting that represents a business meeting.

• Suppose you have a class called Movie. Write a constructor for the class that initializes the title and director instance variables based on parameters passed to the constructor.

• Suppose you have a class called Child with an instance data value called age. Write a getter method and a setter method for age.
Exercise

• List some attributes and operations that might be defined for a class called Meeting that represents a business meeting.
  – Attributes could include: date, time, location, purpose, attendees (list of People)
  – Operations could include: setters and getters for all attributes, announce, cancel
Exercise

• Suppose you have a class called Movie. Write a constructor for the class that initializes the title and director instance variables based on parameters passed to the constructor.

    public Movie(String theTitle, String theDirector)
    {
        title = theTitle;
        director = theDirector;
    }
Exercise

• Suppose you have a class called Child with an instance data value called age. Write a getter method and a setter method for age.

```java
public int getAge()
{
    return age;
}

public void setAge(int newAge)
{
    age = newAge;
}
```
Exercise

• Write a Flight class that describes a flight of an airline, for a particular type of airplane, from an origin to a destination and a distance (int)

• Write an appropriate constructor (sets all the instance data)

• Write a method that compute the time of the flight in hours:mins (String) of the flight depending on the distance and the airplane type (Fast, 1000km/h Slow 600 km/h)

• Write a FlightTest class that creates three flights and print their flight times (implement a toString method for the class Flight).
public class Flight {
    private String airline, origin, destination, type;
    private int flightNumber, distance, speed;

    public Flight(String airlineInit, String originInit, String destinationInit, int flightNumberInit, String typeInit, int distanceInit) {
        airline = airlineInit;
        origin = originInit;
        destination = destinationInit;
        flightNumber = flightNumberInit;
        distance = distanceInit;
        type = typeInit;
        if (type.equals("fast"))
            speed = 1000;
        else if (type.equals("slow"))
            speed = 600;
        else
            speed = 800;
    }

    public String timeFlight() {
        return "" + distance / speed + ":" + (int)((double)(distance % speed) / speed) * 60;
    }
}
public class FlightTest {

    public static void main(String[] args) {
        Flight f2 = new Flight("Delta", "Philadelphia", "London", 212, "slow", 5000);
        Flight f3 = new Flight("Continental", "Atlanta", "Chicago", 822, "average", 750);

        System.out.println(f1);
        System.out.println(f2);
        System.out.println(f3);
    }
}