A clever person solves a problem. A wise person avoids it.  
A. Einstein
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
  – arcs and images
  – events and event handlers
  – buttons and text fields
Outline

Anatomy of a Class
Encapsulation
Anatomy of a Method
Arcs
Images
Graphical User Interfaces
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, Address, Major, Grade point average</td>
<td>Set address, Set major, Compute grade point average</td>
</tr>
<tr>
<td>Rectangle</td>
<td>Length, Width, Color</td>
<td>Set length, Set width, Set color</td>
</tr>
<tr>
<td>Aquarium</td>
<td>Material, Length, Width, Height</td>
<td>Set material, Set length, Set width, Set height, Compute volume, Compute filled weight</td>
</tr>
<tr>
<td>Flight</td>
<td>Airline, Flight number, Origin city, Destination city, Current status</td>
<td>Set airline, Set flight number, Determine status</td>
</tr>
<tr>
<td>Employee</td>
<td>Name, Department, Title, Salary</td>
<td>Set department, Set title, Set salary, Compute wages, Compute bonus, 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;
cchar 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, empty (boolean)
  • Operations could include: getHeight, setHeight, getWidth, 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
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);
}
}
Sample Run

Die One: 5, Die Two: 2
Sum: 5

Die One: 1, Die Two: 4

Die One: 4, Die Two: 2
New sum: 6

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

    // Constructor: Sets the initial face value.
    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 (see Chapter 03 for examples)

Algorithm

– Declaration of the method (signature)
– Creation of the Random object
– Assignment of a random value to the instance data
– Returning the new value of the instance data
public int roll() {
    Random rGen = new Random();
    faceValue = rGen.nextInt(6) + 1;
    return faceValue;
}
Quiz

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

• Write a method of the class Die 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
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:

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 sets to the faceValue and returns the value computed as follows: a) read the current face value of the object, b) compute the third power of that value, c) take the remainder of the division by 6, and finally add 1.
Write a method of the Die class called cube that sets to the faceValue and returns the value computed as follows: a) read the current face value of the object, b) compute the third power of that value, c) take the remainder of the division by 6, and finally add 1.

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

class Die {
    private final int MAX = 6;
    private int faceValue;
}

Could you have defined them as follows?

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
- - - -->
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?
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.
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>public</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td>Violate encapsulation</td>
<td></td>
<td>Enforce encapsulation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
<th>public</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provide services to clients</td>
<td></td>
<td>Support other methods in the class</td>
</tr>
</tbody>
</table>
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.
  – Hint: for the color variable use the class javafx.scene.paint.Color
  – A new color with given rgb values is created with the Color.rgb(int r, int g, int b) (static) method
import javafx.scene.paint.Color;
public class Rectangle {
    private int length = 0;
    private int width = 0;
    private Color color = Color.rgb(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
Arcs
Images
Graphical User Interfaces
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

```
obj.doIt();
main
```

```
doIt
  helpMe();
```

```
helpMe()
```

```
obj
```
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;
}
```
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*

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

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

    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 constructed by (passing) three integer values between 0 and 255 to the Color.rgb() method. They represent the contributions of red, green, and blue (its RGB value). Use the java.util.Random class.
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 constructed by (passing) three integer values between 0 and 255 to the `Color.rgb()` method. They represent the contributions of red, green, and blue (its RGB value). Use the `java.util.Random` class.

```java
public Color randomColor () {
    final int MAX = 256;
    Random generator = new Random();
    int randRed = generator.nextInt(MAX);
    int randGreen = generator.nextInt(MAX);
    int randBlue = generator.nextInt(MAX);
    return Color.rgb(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 ch = obj.calc(25, count, "Hello");

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

• Consider a method defined with the header:
  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);
Quiz

• Consider a method defined with the header:
  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.
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 named updateHours 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;
    }
    public void updateHours (int moreHours) {
        hours += moreHours;
    }
}

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*
transactions.java       Author: Lewis/Loftus
Demonstrates the creation and use of multiple Account objects.

public class Transactions
{
    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);
    }
}
continue

acct1.addInterest();
acct2.addInterest();
acct3.addInterest();
System.out.println();
System.out.println(acct1);
System.out.println(acct2);
System.out.println(acct3);

Output

Smith balance after deposit: 540.0
Smith balance after withdrawal: 107.75

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;

    // Sets up the account by defining its owner, account number, and initial balance.
    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
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:

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

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

```java
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))
Arcs

• In Chapter 3 we explored basic shapes: lines, rectangles, circles, and ellipses

• In JavaFX, an arc is defined as a portion of an ellipse

• Like an ellipse, the first four parameters to the Arc constructor specify the center point (x and y) as well as the radii along the horizontal and vertical

• Two additional parameters specify the portion of the ellipse that define the arc
Arcs

- The Arc constructor:

  \[ \text{Arc}(\text{centerX}, \text{centerY}, \text{radiusX}, \text{radiusY}, \text{startAngle}, \text{arcLength}) \]

- The \textit{start angle} is where the arc begins relative to the horizontal

- The \textit{arc length} is the angle that defines how big the arc is

- Both angles are specified in degrees
Arcs

- An arc whose underlying ellipse is centered at (150, 100), a horizontal radius of 70 and a vertical radius of 30, a start angle of 45 and a arc length of 90:

\[ \text{Arc } \text{myArc} = \text{new Arc}(150, 100, 70, 30, 45, 90); \]
Arcs

• An arc also has an arc type:

<table>
<thead>
<tr>
<th>ArcType</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArcType.OPEN</td>
<td>The curve along the ellipse edge</td>
</tr>
<tr>
<td>ArcType.CHORD</td>
<td>End points are connected by a straight line</td>
</tr>
<tr>
<td>ArcType.ROUND</td>
<td>End points are connected to the center point of the ellipse, forming a rounded “pie” piece</td>
</tr>
</tbody>
</table>

• See ArcDisplay.java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.Group;
import javafx.scene.paint.Color;
import javafx.scene.shape.Arc;
import javafx.scene.shape.ArcType;
import javafx.scene.shape.Ellipse;
import javafx.stage.Stage;

//************************************************************************
// ArcDisplay.java  Author: Lewis/Loftus
//
// Demonstrates the use of the JavaFX Arc class.
//************************************************************************

public class ArcDisplay extends Application
{
    //--------------------------------------
    //  Draws three arcs based on the same underlying ellipse.
    //--------------------------------------
    public void start(Stage primaryStage)
    {
        Ellipse backgroundEllipse = new Ellipse(250, 150, 170, 100);
        backgroundEllipse.setFill(null);
        backgroundEllipse.setStroke(Color.GRAY);
        backgroundEllipse.setStrokeDashArray().addAll(5.0, 5.0);
        continue
Arc arc1 = new Arc(250, 150, 170, 100, 90, 90);
arcl.setType(ArcType.OPEN);
arcl.setFill(null);

Arc arc2 = new Arc(250, 150, 170, 100, 20, 50);
arcl2.setType(ArcType.ROUND);
arcl2.setFill(Color.GREEN);

Arc arc3 = new Arc(250, 150, 170, 100, 230, 130);
arcl3.setType(ArcType.CHORD);
arcl3.setFill(null);

Group root = new Group(backgroundEllipse, arc1, arc2, arc3);
Window.setScene(scene);
Window.show();
Arc arc1 = new Arc(250, 150, 170, 100, 90, 90);
arc1.setType(ArcType.OPEN);
arc1.setStroke(Color.RED);
arc1.setFill(null);

Arc arc2 = new Arc(250, 150, 170, 100, 20, 50);
arc2.setType(ArcType.ROUND);
arc2.setStroke(Color.GREEN);
arc2.setFill(Color.GREEN);

Arc arc3 = new Arc(250, 150, 170, 100, 230, 130);
arc3.setType(ArcType.CHORD);
arc3.setStroke(Color.BLUE);
arc3.setFill(null);

Group root = new Group(backgroundEllipse, arc1, arc2, arc3);

Scene scene = new Scene(root, 500, 300, Color.LIGHTYELLOW);

primaryStage.setTitle("Arc Display");
primaryStage.setScene(scene);
primaryStage.show();
}
}
Images

• The JavaFX Image class is used to load an image from a file or URL

• Supported formats: jpeg, gif, and png

• To display an image, use an ImageView object

• An Image object cannot be added to a container directly

• See ImageDisplay.java
import javafx.application.Application;
import javafx.geometry.Rectangle2D;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

//************************************************************************
// ImageDisplay.java  Author: Lewis/Loftus
//
// Demonstrates a the use of Image and ImageView objects.
//****************************************************************************

class ImageDisplay extends Application {  
    
    public void start(Stage primaryStage) {  
        Image img = new Image("gull.jpg");
        ImageView imgView = new ImageView(img);

        StackPane pane = new StackPane(imgView);
        pane.setStyle("-fx-background-color: cornsilk");
    }
}
continue
continue

Scene scene = new Scene(pane, 500, 350);

primaryStage.setTitle("Image Display");
primaryStage.setScene(scene);
primaryStage.show();

}
Scene scene = new Scene(pane, 500, 350);
primaryStage.setTitle("Image Display");
primaryStage.setScene(scene);
primaryStage.show();
Layout Panes

- This example uses a `StackPane` instead of a `Group` as the root node of the scene.

- A stack pane is a JavaFX *layout pane* (one of several), which governs how its contents are presented.

- A stack pane stacks its nodes on top of each other.

- Since the image view is the only node in the pane, the stack pane simply serves to keep the image centered in the window.
Layout Panes

• The background color of a layout pane is set using a call to the `setStyle` method

• The `setStyle` method accepts a string that can specify various style properties

• The notation used for JavaFX style properties are similar to cascading style sheets (CSS), used to specify the look of HTML elements on a Web page

• JavaFX style property names begin with the prefix “-fx-”
Images

• The parameter to the Image constructor can include a pathname:

```java
Image logo = new Image("myPix/smallLogo.png");
```

• It can also be a URL:

```java
Image logo = new Image("http://example.com/images/bio.jpg");
```
Viewports

• A viewport is a rectangular area that restricts the pixels displayed in an `ImageView`.

• It is defined by a `Rectangle2D` object:

```java
imgView.setViewport(new Rectangle2D(200, 80, 70, 60));
```
Outline

Anatomy of a Class
Encapsulation
Anatomy of a Method
Arcs
Images
Graphical User Interfaces
Text Fields
Graphical User Interfaces

• A Graphical User Interface (GUI) in Java is created with at least three kinds of objects:
  – controls, events, and event handlers

• A control is a screen element that displays information or allows the user to interact with the program:
  – labels, buttons, text fields, sliders, etc.
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
Graphical User Interfaces

• The Java API contains several classes that represent typical events

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

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

• We design event handlers to take whatever actions are appropriate when an event occurs
Graphical User Interfaces

A control object generates an event

A corresponding event handler is designed to respond to the event

When the event occurs, the control calls the appropriate method of the listener, passing an object that describes the event
Graphical User Interfaces

• A JavaFX *button* is defined by the *Button* class

• It generates an *action event*

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

• See *PushCounter.java*
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.text.Text;
import javafx.scene.layout.FlowPane;
import javafx.stage.Stage;

//************************************************************************
// PushCounter.java       Author: Lewis/Loftus
//************************************************************************

public class PushCounter extends Application
{
    private int count;
    private Text countText;

    continue
// Presents a GUI containing a button and text that displays how many times the button is pushed.

public void start(Stage primaryStage)
{
    count = 0;
    countText = new Text("Pushes: 0");

    Button push = new Button("Push Me!");
    push.setOnAction(this::processButtonPress);

    FlowPane pane = new FlowPane(push, countText);
    pane.setAlignment(Pos.CENTER);
    pane.setHgap(20);
    pane.setStyle("-fx-background-color: cyan");

    Scene scene = new Scene(pane, 300, 100);

    primaryStage.setTitle("Push Counter");
    primaryStage.setScene(scene);
    primaryStage.show();
}
// Updates the counter and text when the button is pushed.

public void processButtonPress(ActionEvent event)
{
    count++;
    countText.setText("Pushes: " + count);
}

public void processButtonPress(ActionEvent event) {
    count++;
    countText.setText("Pushes: " + count);
}
Graphical User Interfaces

• A call to the `setOnAction` method sets up the relationship between the button that generates the event and the event handler that responds to it.

• This example uses a `method reference` (using the `::` operator) to specify the event handler method.

• The `this` reference indicates that the event handler method is in the same class.

• So the `PushCounter` class also represents the event handler for this program.
Graphical User Interfaces

• The event handler method can be called whatever you want, but must accept an ActionEvent object as a parameter

• In this example, the event handler method increments the counter and updates the text object

• The counter and Text object are declared at the class level so that both methods can use them
Graphical User Interfaces

• In this example, a `FlowPane` is used as the root node of the scene

• A flow pane is another layout pane, which displays its contents horizontally in rows or vertically in columns

• A gap of 20 pixels is established between elements on a row using the `setHGap` method
Alternate Event Handlers

• Instead of using a method reference, the event handler could be specified using a separate class that implements the `EventHandler` interface:

```java
public class ButtonHandler implements EventHandler<ActionEvent>
{
    public void handle(ActionEvent event)
    {
        count++;
        countText.setText("Pushes: "+ count);
    }
}
```
Alternate Event Handlers

• The event handler class could be defined as public in a separate file or as a private inner class in the same file

• Either way, the call to the `setOnAction` method would specify a new event handler object:

```java
push.setOnAction(new ButtonHandler());
```
Alternate Event Handlers

• Another approach would be to define the event handler using a *lambda expression* in the call to `setOnAction`:

```java
push.setOnAction((event) -> {
    count++;
    countText.setText("Pushes: " + count);
});
```

• A lambda expression is defined by a set of parameters, the `->` operator and an expression
Alternate Event Handlers

- A lambda expression can be used whenever an object of a functional interface is required
- A functional interface contains a single method
- The EventHandler interface is a functional interface
- The method reference approach is equivalent to a lambda expression
Quick Check

Which object in the Push Counter example generated the event?

What did it do then?
Quick Check

Which object in the Push Counter example generated the event?

The Button component generated the event.

What did it do then?

It called the `ProcessButtonPress` method or the `Handle` method of the event handler object that had been registered with it.
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 ButtonHandlerReset with a handle method that resets the count to zero. Write that class.
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 ButtonHandlerReset with a handle method that resets the count to zero. Write that class.

```java
private class ButtonHandlerReset implements EventHandler<ActionEvent> {
    public void handle(ActionEvent event) {
        count = 0;
        countText.setText("Pushes: "+ count);
    }
}
```
Quiz – cont.

• In the original code you must add the following statements if you want to display the new button and set the proper handler

```java
Button reset = new Button("Reset");
reset.setOnAction(new ButtonHandlerReset());
FlowPane pane = new FlowPane(push, countText, reset);
```
Outline

Anatomy of a Class
Encapsulation
Anatomy of a Method
Arcs
Images
Graphical User Interfaces
Text Fields
Text Fields

• Let's look at a GUI example that uses another type of control

• 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 FahrenheitConverter.java
• See FahrenheitPane.java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

//****************************************************************************
// FahrenheitConverter.java       Author: Lewis/Loftus
//
// Demonstrates the use of a TextField and a GridPane.
//****************************************************************************

public class FahrenheitConverter extends Application
{
    //------------------------------------------------------------------------------
    // Launches the temperature converter application.
    //------------------------------------------------------------------------------
    public void start(Stage primaryStage)
    {
        Scene scene = new Scene(new FahrenheitPane(), 300, 150);

        primaryStage.setTitle("Fahrenheit Converter");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

//************************************************************************
// FahrenheitConverter.java
// Author: Lewis / Loftus
// Demonstrates the use of a TextField and a GridPane.
//************************************************************************

class FahrenheitConverter extends Application {
    // Launches the temperature converter application.
    public void start(Stage primaryStage) {
        Scene scene = new Scene(new FahrenheitPane(), 300, 150);

        primaryStage.setTitle("Fahrenheit Converter");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}

Text Fields

- The details of the user interface are set up in a separate class that extends `GridPane`.

- `GridPane` is a JavaFX layout pane that displays nodes in a rectangular grid.

- The GUI elements are set up in the constructor of `FahrenheitPane`.

- The event handler method is also defined in `FahrenheitPane`.
import javafx.event.ActionEvent;
import javafx.geometry.HPos;
import javafx.geometry.Pos;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.scene.text.Font;

//************************************************************************
// FahrenheitPane.java       Author: Lewis/Loftus
//************************************************************************

public class FahrenheitPane extends GridPane
{
    private Label result;
    private TextField fahrenheit;

    continue
Sets up a GUI containing a labeled text field for converting temperatures in Fahrenheit to Celsius.

```java
public FahrenheitPane()
{
    Font font = new Font(18);

    Label inputLabel = new Label("Fahrenheit:'");
    inputLabel.setFont(font);
    GridPane.setAlignment(inputLabel, HPos.RIGHT);

    Label outputLabel = new Label("Celsius:'");
    outputLabel.setFont(font);
    GridPane.setAlignment(outputLabel, HPos.RIGHT);

    result = new Label("---");
    result.setFont(font);
    GridPane.setAlignment(result, HPos.CENTER);
}
```
fahrenheit = new TextField();
fahrenheit.setFont(font);
fahrenheit.setPrefWidth(50);
fahrenheit.setAlignment(Pos.CENTER);
fahrenheit.setOnAction(this::processReturn);

setAlignment(Pos.CENTER);
setHgap(20);
setVgap(10);
setStyle("-fx-background-color: yellow");

add(inputLabel, 0, 0);
add(fahrenheit, 1, 0);
add(outputLabel, 0, 1);
add(result, 1, 1);
continue

// Computes and displays the converted temperature when the user
// presses the return key while in the text field.
//
public void processReturn(ActionEvent event)
{
    int fahrenheitTemp = Integer.parseInt(fahrenheit.getText());
    int celsiusTemp = (fahrenheitTemp - 32) * 5 / 9;
    result.setText(celsiusTemp + "");
}
}
Text Fields

• Through inheritance, a FahrenheitPane is a GridPane and inherits the add method

• The parameters to add specify the grid cell to which to add the node

• Row and column numbering in a grid pane start at 0

• When the user presses return, the event handler method is called, which converts the value and updates the text result
Summary

• Chapter 4 focused on:
  – class definitions
  – instance data
  – encapsulation and Java modifiers
  – method declaration and parameter passing
  – constructors
  – arcs and images
  – events and event handlers
  – buttons and text fields
Exercise

• Write a constructor of the class Rectangle that we have defined previously in the lecture (slide 67)
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.
public double withdraw2 (double amount, double fee)
{
    balance -= (1 + fee) * amount;
    return balance;
}
Exercise

• List some attributes and operations that might be defined for a class called Meeting that represents a business meeting.
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 (of type String) based on parameters passed to the constructor.
Exercise

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

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