Chapter 3
Introduction to Classes, Objects Methods and Strings

Objectives

- In this chapter you'll learn:
  - How to declare a class and use it to create an object.
  - How to implement a class's behaviors as methods.
  - How to implement a class's attributes as instance variables and properties.
  - How to call an object's methods to make them perform their tasks.
  - What local variables of a method are and how they differ from instance variables.
  - What primitive types and reference types are.
  - How to use a constructor to initialize an object's data.
  - How to represent and use numbers containing decimal points.

3.1 Introduction

- Covered in this chapter
  - Classes
  - Objects
  - Methods
  - Parameters
  - Floating-point numbers

3.2 Instance Variables, set Methods and get Methods

```
1  // Fig. 3.1: Account.java
2  // Account class that contains a name instance variable
3  // and methods to set and get its value.
4
5  public class Account
6  {
7     private String name; // instance variable
8
9     // method to set the name in the object
10    public void setName(String name)
11    {
12       this.name = name; // store the name
13    }
14
15    // method to retrieve the name from the object
16    public String getName()
17    {
18       return name; // return value of name to caller
19    }
20 } // end class Account
```

Fig. 3.1 | Account class that contains a name instance variable and methods to *set* and *get* its value.

3.2 Instance Variables, set Methods and get Methods

- The *class declaration* begins in line 5:

```
public class Account
```

- Instance Variable name:
  - declared *inside* a class declaration but *outside* the bodies of the class's methods

```
private String name; // instance variable
```

Good Programming Practice 3.1

- We prefer to list a class's instance variables first in the class's body, so that you see the names and types of the variables before they're used in the class's methods. You can list the class's instance variables anywhere in the class outside its method declarations, but scattering the instance variables can lead to hard-to-read code.

Controlling Access to Members

- Member access modifiers
  - public
    - Variables and methods accessible to clients of the class and its subclass
  - private
    - Variables and methods not accessible to clients of the class
    - Only accessible in methods of that superclass
    - Declaring instance variables private is known as data hiding
  - protected
    - Can accessible by methods of the superclass, by methods of subclasses and by methods of other class in the same package
  - default
    - Control access to class's variables and methods
    - A member that is not declared public, protected, or private is said to have default access and may be accessed from, and only from, anywhere in the package in which it is declared.

- *argument*
  - Method setName receives *parameter* name of type String—which represents the name that will be passed to the method as an *argument*.
  - Parameters are local variables.
- *this*
  - The method's body can use the keyword ***this*** to refer to the shadowed instance variable explicitly.
- *return*
  - The method's **return type** (which appears before the method name) specifies the type of data the method returns to its *caller* after performing its task.

Driver Class AccountTest

```
1  // Fig. 3.2: AccountTest.java
2  // Creating and manipulating an Account object.
3  import java.util.Scanner;
4
5  public class AccountTest
6  {
7     public static void main(String[] args)
8     {
9        // create a Scanner object to obtain input from the command window
10       Scanner input = new Scanner(System.in);
11
12       // create an Account object and assign it to myAccount
13       Account myAccount = new Account();
14
15       // display initial value of name (null)
16       System.out.printf("Initial name is: %s%n%n", myAccount.getName());
17
```

```
18          // prompt for and read name
19          System.out.println("Please enter the name:");
20          String theName = input.nextLine(); // read a line of text
21          myAccount.setName(theName); // put theName in myAccount
22          System.out.println(); // outputs a blank line
23
24          // display the name stored in object myAccount
25          System.out.printf("Name in object myAccount is:%n%s%n",
26             myAccount.getName());
27       }
28    } // end class AccountTest
```

```
Initial name is: null

Please enter the name:
Jane Green

Name in object myAccount is:
Jane Green
```

Fig. 3.2 | Creating and manipulating an Account object. (Part 2 of 2.)

- ***new***
  - Keyword **new** creates a new object of the specified class
- ***Calling Class Account's getName Method***
  - object followed by a **dot separator** (**.**)
  - The number of arguments in a method call must match the number of parameters in the method declaration's parameter list
- ***null—the Default Initial Value for String Variables***
  - Unlike local variables, which are not automatically initialized, every instance variable has a default initial value

Compiling and Executing an App with Multiple Classes

javac Account.java AccountTest.java

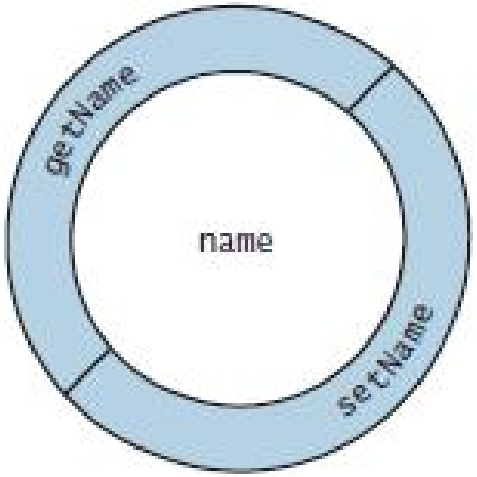
Javac *.java

Notes on import Declarations

- Classes System and String are in package java.lang
  - Implicitly imported into every Java program
  - Can use the java.lang classes without explicitly importing them
  - Most classes you'll use in Java programs must be imported explicitly.
- Classes that are compiled in the same directory on disk are in the same package—known as the default package.
- Classes in the same package are implicitly imported into the source-code files of other classes in the same package.
- An import declaration is not required if you always refer to a class via its fully qualified class name
  - owed by a dot (.) and the class name.

Software Engineering with private Instance Variables and public *set* and *get* Methods

- ***Conceptual View of an Account Object with Encapsulated Data***
  - Any client code that needs to interact with the Account object can do
- so *only* by calling the public methods of the protective outer layer.



3.3 Primitive Types vs. Reference Types

- Types in Java
  - Primitive
    - boolean, byte, char, short, int, long, float, double
  - Reference (sometimes called nonprimitive types)
    - All non primitive types are reference types.
    - Objects
    - Default value of null
    - Used to invoke an object’ s methods

Example: e.java

Error-Prevention Tip 3.1

An attempt to use an uninitialized local variable causes a compilation error.

3.4 Primitive Types vs. Reference Types

- Programs use variables of reference types (normally called references) to store the locations of objects in the computer's memory.
  - Such a variable is said to refer to an object in the program.
- When using an object of another class, a reference to the object is required to invoke (i.e., call) its methods.
  - Also known as sending messages to an object.

3.4 Initializing Objects with Constructors

- When an object of a class is created, its instance variables are initialized by default.
- Java requires a constructor call for *every* object that is created.
- Keyword new
- A constructor *must* have the same name as the class.

3.4 Initializing Objects with Constructors (Cont.)

- By default, the compiler provides a default constructor with no parameters
- A constructor's parameter list specifies the data it requires to perform its task.
- Constructors cannot return values, so they cannot specify a return type.
- Normally, constructors are declared public.
- *If you declare any constructors for a class, the Java compiler will not cre onstructor for that class.*

```
1  // Fig. 3.5: Account.java
2  // Account class with a constructor that initializes the name.
3
4  public class Account
5  {
6     private String name; // instance variable
7
8     // constructor initializes name with parameter name
9     public Account(String name) // constructor name is class name
10    {
11       this.name = name;
12    }
13
14    // method to set the name
15    public void setName(String name)
16    {
17       this.name = name;
18    }
19
20    // method to retrieve the name
21    public String getName()
22    {
23       return name;
24    }
25 } // end class Account
```

Fig. 3.5 | Account class with a constructor that initializes the name.

```
1  // Fig. 3.6: AccountTest.java
2  // Using the Account constructor to initialize the name instance
3  // variable at the time each Account object is created.
4
5  public class AccountTest
6  {
7     public static void main(String[] args)
8     {
9        // create two Account objects
10       Account account1 = new Account("Jane Green");
11       Account account2 = new Account("John Blue");
12
13       // display initial value of name for each Account
14       System.out.printf("account1 name is: %s%n", account1.getName());
15       System.out.printf("account2 name is: %s%n", account2.getName());
16    }
17 } // end class AccountTest
```

```
account1 name is: Jane Green
account2 name is: John Blue
```

Fig. 3.6 | Using the `Account` constructor to initialize the `name` instance variable at the time each `Account` object is created.

3.5 Floating-Point Numbers and Type double

- Floating-point number
  - A number with a decimal point, such as 7.33, 0.0975 or 1000.12345).
  - float and double primitive types
  - double variables can store numbers with larger magnitude and finer detail than float variables.
- float represents single-precision floating-point numbers up to 7 significant digits.
- double represents double-precision floating-point numbers that require twice as much memory as float and provide 15 significant digits—approximately double the precision of float variables.

3.5 Floating-Point Numbers and Type double (Cont.)

- Java treats all floating-point literals (such as 7.33 and 0.0975) as double values by default.
- Appendix D, Primitive Types shows the ranges of values for floats and doubles.

```
1  // Fig. 3.8: Account.java
2  // Account class with a double instance variable balance and a constructor
3  // and deposit method that perform validation.
4
5  public class Account
6  {
7     private String name; // instance variable
8     private double balance; // instance variable
9
10    // Account constructor that receives two parameters
11    public Account(String name, double balance)
12    {
13       this.name = name; // assign name to instance variable name
14
15       // validate that the balance is greater than 0.0; if it's not,
16       // instance variable balance keeps its default initial value of 0.0
17       if (balance > 0.0) // if the balance is valid
18          this.balance = balance; // assign it to instance variable balance
19    }
20
21    // method that deposits (adds) only a valid amount to the balance
22    public void deposit(double depositAmount)
23    {
24       if (depositAmount > 0.0) // if the depositAmount is valid
25          balance = balance + depositAmount; // add it to the balance
26    }
27
```

```
28      // method returns the account balance
29      public double getBalance()
30      {
31         return balance;
32      }
33
34      // method that sets the name
35      public void setName(String name)
36      {
37         this.name = name;
38      }
39
40      // method that returns the name
41      public String getName()
42      {
```

Fig. 3.8 | `Account` class with a `double` instance variable `balance` and a constructor and `deposit` method that perform validation. (Part 1 of 2.)

```
1  // Fig. 3.9: AccountTest.java
2  // Inputting and outputting floating-point numbers with Account objects.
3  import java.util.Scanner;
4
5  public class AccountTest
6  {
7     public static void main(String[] args)
8     {
9        Account account1 = new Account("Jane Green", 50.00);
10       Account account2 = new Account("John Blue", -7.53);
11
12       // display initial balance of each object
13       System.out.printf("%s balance: $%.2f%n",
14          account1.getName(), account1.getBalance());
15       System.out.printf("%s balance: $%.2f%n%n",
16          account2.getName(), account2.getBalance());
17
18       // create a Scanner to obtain input from the command window
19       Scanner input = new Scanner(System.in);
20
21       System.out.print("Enter deposit amount for account1: "); // prompt
22       double depositAmount = input.nextDouble(); // obtain user input
23       System.out.printf("%nadding %.2f to account1 balance%n%n",
24          depositAmount);
25       account1.deposit(depositAmount); // add to account1's balance
26
```

```
27          // display balances
28          System.out.printf("%s balance: $%.2f%n",
29             account1.getName(), account1.getBalance());
30          System.out.printf("%s balance: $%.2f%n%n",
31             account2.getName(), account2.getBalance());
32
33          System.out.print("Enter deposit amount for account2: "); // prompt
34          depositAmount = input.nextDouble(); // obtain user input
35          System.out.printf("%nadding %.2f to account2 balance%n%n",
36             depositAmount);
37          account2.deposit(depositAmount); // add to account2 balance
38
39          // display balances
40          System.out.printf("%s balance: $%.2f%n",
41             account1.getName(), account1.getBalance());
42          System.out.printf("%s balance: $%.2f%n%n",
43             account2.getName(), account2.getBalance());
44       } // end main
45    } // end class AccountTest
```

```
Jane Green balance: $50.00
John Blue balance: $0.00

Enter deposit amount for account1: 25.53

adding 25.53 to account1 balance

Jane Green balance: $75.53
John Blue balance: $0.00
```

```
Enter deposit amount for account2: 123.45

adding 123.45 to account2 balance

Jane Green balance: $75.53
John Blue balance: $123.45
```

- `System.out.printf`
  - Format specifier `%.2f`
  - `%f` is used to output values of type `float` or `double`.
  - `.2` represents the number of decimal places (2) to output to the right of the decimal point—known as the number's precision.
  - Any floating-point value output with `%.2f` will be rounded to the hundredths position.
- `Scanner` method `nextDouble` returns a `double` value entered by the user.

BigDecimal class

- Immutable, arbitrary-precision signed decimal numbers.
- Provides operations for arithmetic, scale manipulation, rounding, comparison, hashing, and format conversion.

Example: Arith.java

3.6 (Optional) GUI and Graphics Case Study: A Simple GUI

- A graphical user interface (GUI) presents a user-friendly mechanism for interacting with an app.
- A GUI(pronounced “GOO-ee”) gives an app a distinctive “look-and-feel”.
- JavaFX—Java’s GUI, graphics and multimedia technology of the future.

3.6 (Optional) GUI and Graphics Case Study: A Simple GUI (Cont.)

- GUIs are built from GUI components – JavaFX refers to these as controls.
- Controls are GUI components, such as Label objects that display text.
- Programs can respond to these events – known as event handling.

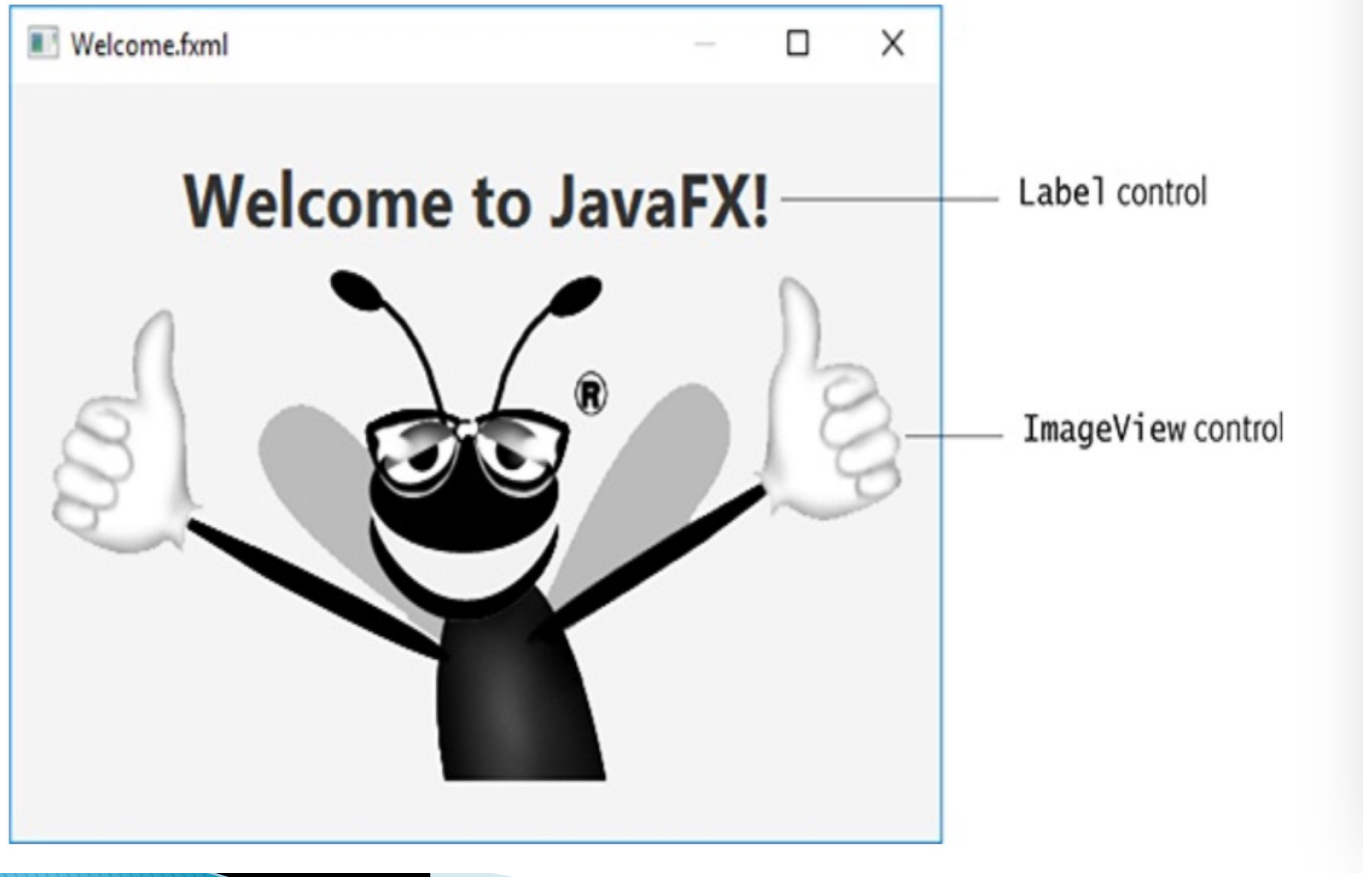
3.6 (Optional) GUI and Graphics Case Study: A Simple GUI (Cont.)

- JavaFX Scene Builder and FXML
  - Scene Builder is a tool that enables you to create GUIs simply by dragging and dropping pre-built GUI components.
  - FXML(FX Markup Language) is a language for defining and arranging JavaFX GUI controls without writing any Java code.
  - You can download Scene Builder from:

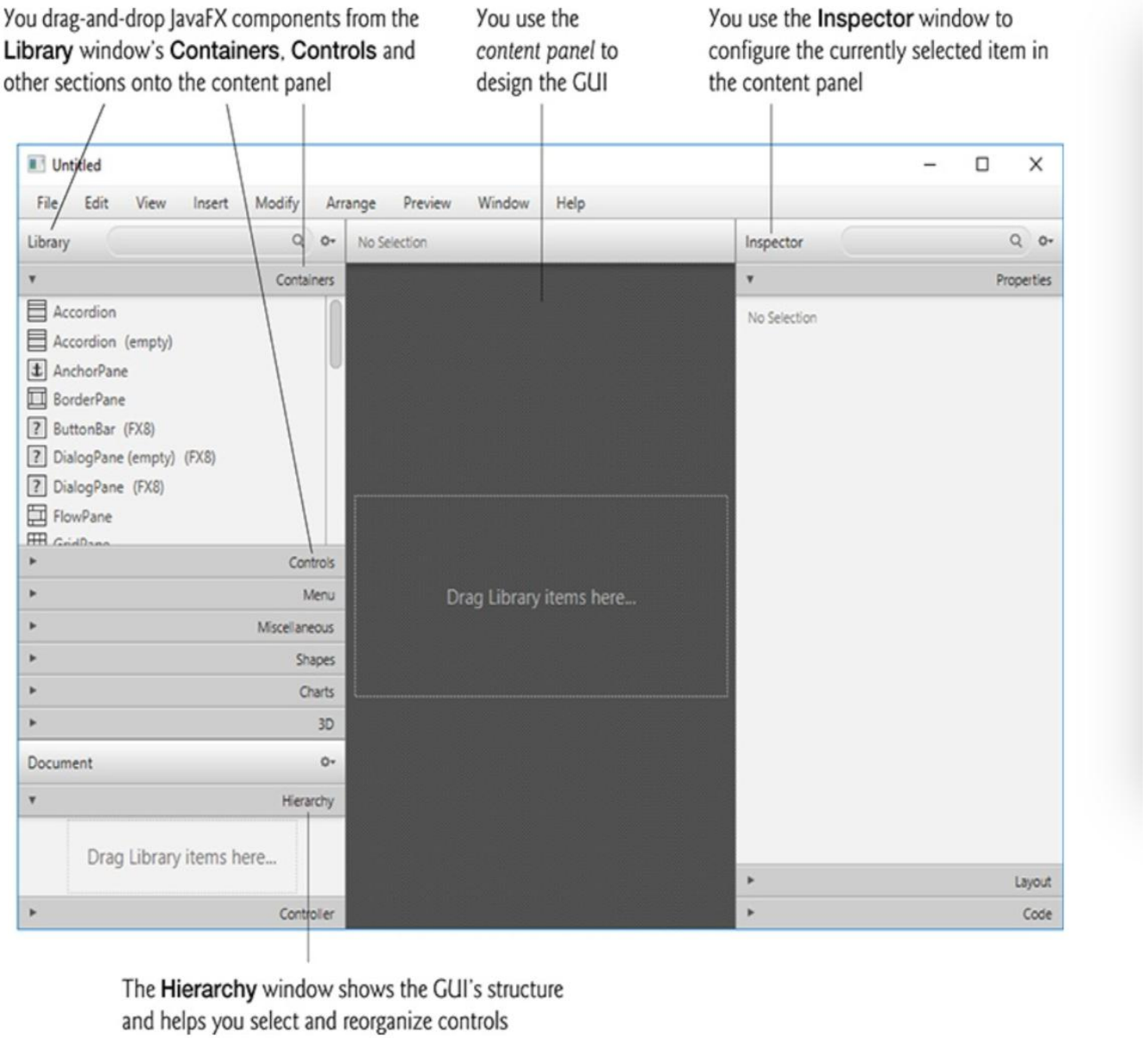
```
http://gluonhq.com/labs/scene-builder/
```

3.6 (Optional) GUI and Graphics Case Study: A Simple GUI (Cont.)

- Welcome App – Displaying Text and an Image



3.6 (Optional) GUI and Graphics Case Study: A Simple GUI (Cont.)



3.6 (Optional) GUI and Graphics Case Study: A Simple GUI (Cont.)

- JavaFX layout container help you arrange and size controls.
- For the VBox, set the following properties:
  - Alignment—center
  - PrefWidth, PrefHeight—450， 300
- For the Label, set the following properties:
  - text — Welcome to JavaFX!
  - font — Style to Bold , Size to 30.
- For the ImageView, set the following properties:
  - Image
  - Fit Width and Fit Height ——Reset to Default
- Preview -> Show Preview in Window
- File >