Chapter 3
Introduction to Classes, Objects, Methods and Strings

Java How to Program, 11/e

Questions? E-mail paul.deitel@deitel.com

OBJECTIVES

In this chapter you'll:

- Declare a class and use it to create an object.
- Implement a class's behaviors as methods.
- Implement a class's attributes as instance variables.
- Call an object's methods to make them perform their tasks.

OBJECTIVES (cont.)

- Learn what local variables of a method are and how they differ from instance variables.
- Learn what primitive types and reference types are.
- Use a constructor to initialize an object's data.
- Represent and use numbers containing decimal points.
- Learn why classes are a natural way to model real-world things and abstract entities.

OUTLINE

OUTLINE (cont.)

OUTLINE (cont.)

OUTLINE (cont.)

3.2 Instance Variables, *set* Methods and *get* Methods

- Each class you create becomes a new type that can be used to declare variables and create objects.
- You can declare new classes as needed; this is one reason Java is known as an extensible language.

```
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     private String name; // instance variable
7
8     // method to set the name in the object
9     public void setName(String name) {
10       this.name = name; // store the name
11    }
12
13    // method to retrieve the name from the object
14    public String getName() {
15       return name; // return value of name to caller
16    }
17 }
```

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

3.2.1 `Account` Class with an Instance Variable, a *set* Method and a *get* Method (Cont.)

Class Declaration

- Each class declaration that begins with the access modifier `public` must be stored in a file that has the same name as the class and ends with the `.java` filename extension.
- Every class declaration contains keyword `class` followed immediately by the class's name.

3.2.1 Account Class with an Instance Variable, a *set* Method and a *get* Method (Cont.)

Identifiers and Camel Case Naming

- Class, method and variable names are identifiers.
- By convention all use camel case names.
- Class names begin with an uppercase letter, and method and variable names begin with a lowercase letter.

3.2.1 Account Class with an Instance Variable, a *set* Method and a *get* Method (Cont.)

Instance Variable name

- An object has attributes that are implemented as instance variables and carried with it throughout its lifetime.
- Instance variables exist before methods are called on an object, while the methods are executing and after the methods complete execution.
- A class normally contains one or more methods that manipulate the instance variables that belong to particular objects of the class.
- Instance variables are declared inside a class declaration but outside the bodies of the class's method declarations.
- Each object (instance) of the class has its own copy of each of the class's instance variables.

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.

3.2.1 `Account` Class with an Instance Variable, a *set* Method and a *get* Method (Cont.)

Access Modifiers `public` and `private`

- Most instance-variable declarations are preceded with the keyword `private`, which is an access modifier.
- Variables or methods declared with access modifier `private` are accessible only to methods of the class in which they're declared.

3.1.1 Account Class with an Instance Variable, a *set* Method and a *get* Method (Cont.)

***setName* Method of Class `Account`**

- Parameters are declared in a comma-separated parameter list, which is located inside the parentheses that follow the method name in the method declaration.
- Multiple parameters are separated by commas.
- Each parameter must specify a type followed by a variable name.

3.1.1 Account Class with an Instance Variable, a *set* Method and a *get* Method (Cont.)

Parameters Are Local Variables

- Variables declared in the body of a particular method are local variables and can be used only in that method.
- When a method terminates, the values of its local variables are lost.
- A method's parameters are local variables of the method.

3.1.1 Account Class with an Instance Variable, a *set* Method and a *get* Method (Cont.)

`setName` Method Body

- Every method's body is delimited by left and right braces ({ and }).
- Each method's body contains one or more statements that perform the method's task(s).

Good Programming Practice 3.2

We could have avoided the need for keyword `this` here by choosing a different name for the parameter in line 9, but using the `this` keyword as shown in line 10 is a widely accepted practice to minimize the proliferation of identifier names.

3.2.1 `Account` Class with an Instance Variable, a *set* Method and a *get* Method (Cont.)

*`getName` **Method of Class** `Account`*

- The method's return type specifies the type of data returned to a method's caller.
- Keyword `void` indicates that a method will perform a task but will not return any information.
- Empty parentheses following a method name indicate that the method does not require any parameters to perform its task.
- When a method that specifies a return type other than `void` is called and completes its task, the method must return a result to its calling method.

3.2.1 `Account` Class with an Instance Variable, a *set* Method and a *get* Method (Cont.)

- The `return` statement passes a value from a called method back to its caller.
- Classes often provide `public` methods to allow the class's clients to *set* or *get* `private` instance variables.
- The names of these methods need not begin with *set* or *get*, but this naming convention is recommended.

3.2.2 `AccountTest` Class That Creates and Uses an Object of Class `Account`

Driver Class `AccountTest`

- A class that creates an object of another class, then calls the object's methods, is a driver class.

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

Fig. 3.2 | Creating and manipulating an `Account` object. (Part 1 of 2.)

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

```
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.)

3.2.2 `AccountTest` Class That Creates and Uses an Object of Class `Account` (Cont.)

`Scanner` Object for Receiving Input from the User

- `Scanner` method `nextLine` reads characters until a newline character is encountered, then returns the characters as a `String`.
- `Scanner` method `next` reads characters until any white-space character is encountered, then returns the characters as a `String`.

3.2.2 `AccountTest` Class That Creates and Uses an Object of Class `Account` (Cont.)

Instantiating an Object—Keyword `new` and Constructors

- A class instance creation expression begins with keyword `new` and creates a new object.
- A constructor is similar to a method but is called implicitly by the `new` operator to initialize an object's instance variables at the time the object is created.

3.2.2 `AccountTest` Class That Creates and Uses an Object of Class `Account` (Cont.)

Calling Class `Account`'s `getName` Method

- To call a method of an object, follow the object name with a dot separator, the method name and a set of parentheses containing the method's arguments.

Error-Prevention Tip 3.1

Never use as a format-control a string that was input from the user. When method `System.out.printf` evaluates the format-control string in its first argument, the method performs tasks based on the conversion specifier(s) in that string. If the format-control string were obtained from the user, a malicious user could supply conversion specifiers that would be executed by `System.out.printf`, possibly causing a security breach.

Common Programming Error 3.1

Splitting a statement in the middle of an identifier or a string is a syntax error.

3.2.2 `AccountTest` Class That Creates and Uses an Object of Class `Account` (Cont.)

`null`—the Default Initial Value for `String` Variables

- Local variables are not automatically initialized.
- Every instance variable has a default initial value—a value provided by Java when you do not specify the instance variable's initial value.
- The default value for an instance variable of type `String` is `null`.

3.2.2 `AccountTest` Class That Creates and Uses an Object of Class `Account` (Cont.)

Calling Class `Account`'s `setName` Method

- A method call supplies values—known as arguments—for each of the method's parameters.
- Each argument's value is assigned to the corresponding parameter in the method header.
- The number of arguments in a method call must match the number of parameters in the method declaration's parameter list.
- The argument types in the method call must be consistent with the types of the corresponding parameters in the method's declaration.

3.2.3 Compiling and Executing an App with Multiple Classes

- The `javac` command can compile multiple classes at once.
- Simply list the source-code filenames after the command with each filename separated by a space from the next.
- If the directory containing the app includes only one app's files, you can compile all of its classes with the command `javac *.java`.
- The asterisk (*) in `*.java` indicates that all files in the current directory ending with the filename extension "`.java`" should be compiled.

3.2.4 Account UML Class Diagram

Top Compartment

- In the UML, each class is modeled in a class diagram as a rectangle with three compartments. The top one contains the class's name centered horizontally in boldface.

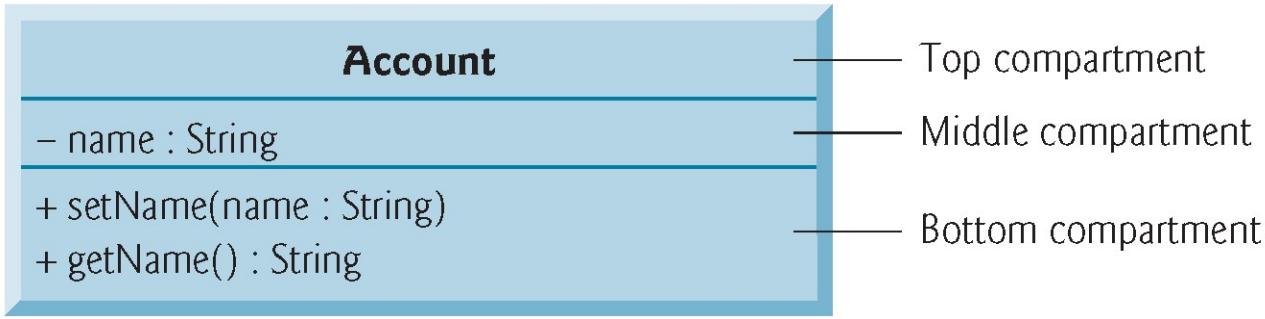

Fig. 3.3 | UML class diagram for class `Account` of Fig. 3.1.

3.2.4 Account UML Class Diagram

Middle Compartment

- The middle compartment contains the class's attributes, which correspond to instance variables in Java.

3.2.4 Account UML Class Diagram

Bottom Compartment

- The bottom compartment contains the class's operations, which correspond to methods and constructors in Java.
- The UML represents instance variables as an attribute name, followed by a colon and the type.
- Private attributes are preceded by a minus sign (–) in the UML.
- The UML models operations by listing the operation name followed by a set of parentheses.
- A plus sign (+) in front of the operation name indicates that the operation is a public one in the UML (i.e., a `public` method in Java).

3.2.4 Account UML Class Diagram

Return Types

- The UML indicates an operation's return type by placing a colon and the return type after the parentheses following the operation name.
- UML class diagrams do not specify return types for operations that do not return values.
- Declaring instance variables `private` is known as data hiding or information hiding.

3.2.4 Account UML Class Diagram

Parameters

- The UML models a parameter of an operation by listing the parameter name, followed by a colon and the parameter type between the parentheses after the operation name

3.2.5 Additional Notes on Class AccountTest

`static` Method `main`

- You must call most methods other than `main` explicitly to tell them to perform their tasks.
- A key part of enabling the JVM to locate and call method `main` to begin the app's execution is the `static` keyword, which indicates that `main` is a `static` method that can be called without first creating an object of the class in which the method is declared.

3.2.5 Additional Notes on Class `AccountTest` (Cont.)

Notes on `import` Declarations

- Most classes you'll use in Java programs must be imported explicitly.
- There's a special relationship between classes that are compiled in the same directory.
- By default, such classes are considered to be 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 that package.

3.2.5 Additional Notes on Class `AccountTest` (Cont.)

- An `import` declaration is not required when one class in a package uses another in the same package.
- An `import-` declaration is not required if you always refer to a class with its fully qualified class name, which includes its package name and class name.

Software Engineering Observation 3.1

The Java compiler does not require `import` declarations in a Java source-code file if the fully qualified class name is specified every time a class name is used. Most Java programmers prefer the more concise programming style enabled by `import` declarations.

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

- Declaring instance variables `private` is known as data hiding or information hiding.

Software Engineering Observation 3.2

Precede each instance variable and method declaration with an access modifier. Generally, instance variables should be declared `private` and methods `public`. Later in the book, we'll discuss why you might want to declare a method `private`.

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

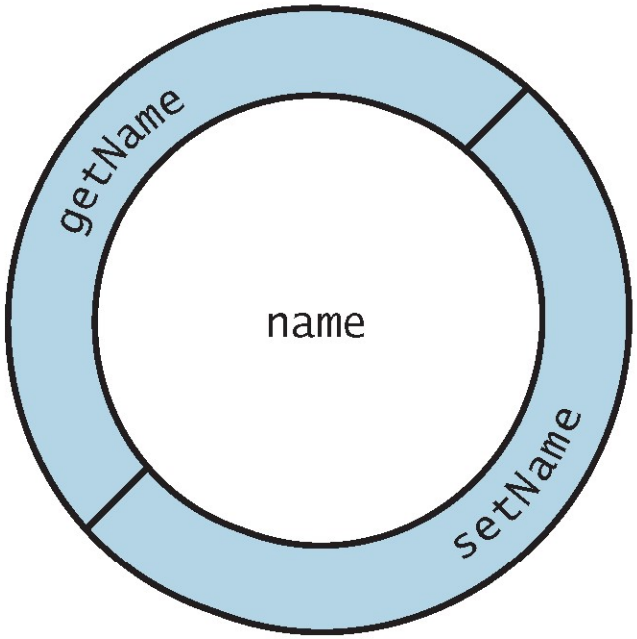

Fig. 3.4 | Conceptual view of an `Account` object with its encapsulated `private` instance variable `name` and protective layer of `public` methods.

3.3 Account Class: Initializing Objects with Constructors

- Each class you declare can optionally provide a constructor with parameters that can be used to initialize an object of a class when the object is created.
- Java *requires* a constructor call for *every* object that's created.

3.3.1 Declaring an Account Constructor for Custom Object Initialization

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

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

Error-Prevention Tip 3.2

Even though it's possible to do so, do not call methods from constructors. We'll explain this in Chapter 10, Object-Oriented Programming: Polymorphism and Interfaces.

3.3.2 Class AccountTest: Initializing Account Objects When They're Created

Constructors Cannot Return Values

- Constructors can specify parameters but not return types.

Default Constructor

- If a class does not define constructors, the compiler provides a default constructor with no parameters, and the class's instance variables are initialized to their default values.

There's No Default Constructor in a Class That Declares a Constructor

- If you declare a constructor for a class, the compiler will *not* create a *default constructor* for that class.

```
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     public static void main(String[] args) {
7        // create two Account objects
8        Account account1 = new Account("Jane Green");
9        Account account2 = new Account("John Blue");
10
11       // display initial value of name for each Account
12       System.out.printf("account1 name is: %s%n", account1.getName());
13       System.out.printf("account2 name is: %s%n", account2.getName());
14    }
15 }
```

```
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.

Software Engineering Observation 3.3

Unless default initialization of your class's instance variables is acceptable, provide a custom constructor to ensure that your instance variables are properly initialized with meaningful values when each new object of your class is created.

3.3.2 Class `AccountTest`: Initializing Account Objects When They're Created (Cont.)

Adding the Contructor to Class `Account`'s UML Class Diagram

- The UML models constructors in the third compartment of a class diagram.
- To distinguish a constructor from a class's operations, the UML places the word "constructor" between guillemets (« and ») before the constructor's name.

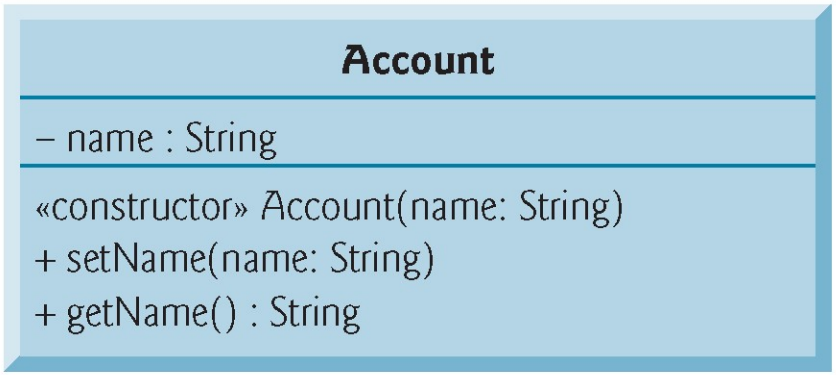

Fig. 3.7 | UML class diagram for `Account` class of Fig. 3.5.

3.4 Account Class with a Balance; Floating-Point Numbers

- A floating-point number is a number with a decimal point.
- Java provides two primitive types for storing floating-point numbers in memory—`float` and `double`.
- Variables of type `float` represent single-precision floating-point numbers and have seven significant digits.
- Variables of type `double` represent double-precision floating-point numbers.
- These require twice as much memory as `float` variables and provide 15 significant digits—approximately double the precision of `float` variables.
- Floating-point literals are of type `double` by default.

```
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     private String name; // instance variable
7     private double balance; // instance variable
8
9     // Account constructor that receives two parameters
10    public Account(String name, double balance) {
11       this.name = name; // assign name to instance variable name
12
13       // validate that the balance is greater than 0.0; if it's not,
14       // instance variable balance keeps its default initial value of 0.0
15       if (balance > 0.0) { // if the balance is valid
16          this.balance = balance; // assign it to instance variable balance
17       }
18    }
```

Fig. 3.8 | `Account` class with a `double` instance variable `balance` and a constructor and `deposit` method that perform validation. (Part 1 of 3.)

```
19
20      // method that deposits (adds) only a valid amount to the balance
21      public void deposit(double depositAmount) {
22         if (depositAmount > 0.0) { // if the depositAmount is valid
23            balance = balance + depositAmount; // add it to the balance
24         }
25      }
26
27      // method returns the account balance
28      public double getBalance() {
29         return balance;
30      }
31
32      // method that sets the name
33      public void setName(String name) {
34         this.name = name;
35      }
```

Fig. 3.8 | `Account` class with a `double` instance variable `balance` and a constructor and `deposit` method that perform validation. (Part 2 of 3.)

```
36
37      // method that returns the name
38      public String getName() {
39         return name;
40      }
41   }
```

Fig. 3.8 | `Account` class with a `double` instance variable `balance` and a constructor and `deposit` method that perform validation. (Part 3 of 3.)

3.4.2 AccountTest Class to Use Class Account

```
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     public static void main(String[] args) {
7        Account account1 = new Account("Jane Green", 50.00);
8        Account account2 = new Account("John Blue", -7.53);
9
10       // display initial balance of each object
11       System.out.printf("%s balance: $%.2f%n",
12          account1.getName(), account1.getBalance());
13       System.out.printf("%s balance: $%.2f%n%n",
14          account2.getName(), account2.getBalance());
15
```

Fig. 3.9 | Inputting and outputting floating-point numbers with `Account` objects. (Part 1 of 4.)

```
16          // create a Scanner to obtain input from the command window
17          Scanner input = new Scanner(System.in);
18
19          System.out.print("Enter deposit amount for account1: "); // prompt
20          double depositAmount = input.nextDouble(); // obtain user input
21          System.out.printf("%nadding %.2f to account1 balance%n%n",
22             depositAmount);
23          account1.deposit(depositAmount); // add to account1's balance
24
25          // display balances
26          System.out.printf("%s balance: $%.2f%n",
27             account1.getName(), account1.getBalance());
28          System.out.printf("%s balance: $%.2f%n%n",
29             account2.getName(), account2.getBalance());
30
```

Fig. 3.9 | Inputting and outputting floating-point numbers with `Account` objects. (Part 2 of 4.)

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

Fig. 3.9 | Inputting and outputting floating-point numbers with `Account` objects. (Part 3 of 4.)

```
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
```

Fig. 3.9 | Inputting and outputting floating-point numbers with `Account` objects. (Part 4 of 4.)

3.4.2 `AccountTest` Class to Use Class `Account` (Cont.)

- `Scanner` method `nextDouble` returns a `double` value.

3.4.2 `AccountTest` Class to Use Class `Account` (Cont.)

Formatting Floating-Point Numbers for Display

- The format specifier `%f` is used to output values of type `float` or `double`.
- The format specifier `%.2f` specifies that two digits of precision should be output to the right of the decimal point in the floating-point number.

Error-Prevention Tip 3.3

The Java compiler issues a compilation error if you attempt to use the value of an uninitialized local variable. This helps you avoid dangerous execution-time logic errors. It's always better to get the errors out of your programs at compilation time rather than execution time.

3.4.2 `AccountTest` Class to Use Class `Account` (Cont.)

- The default value for an instance variable of type `double` is `0.0`, and the default value for an instance variable of type `int` is `0`.

Software Engineering Observation 3.4

Replacing duplicated code with calls to a method that contains one copy of that code can reduce the size of your program and improve its maintainability.

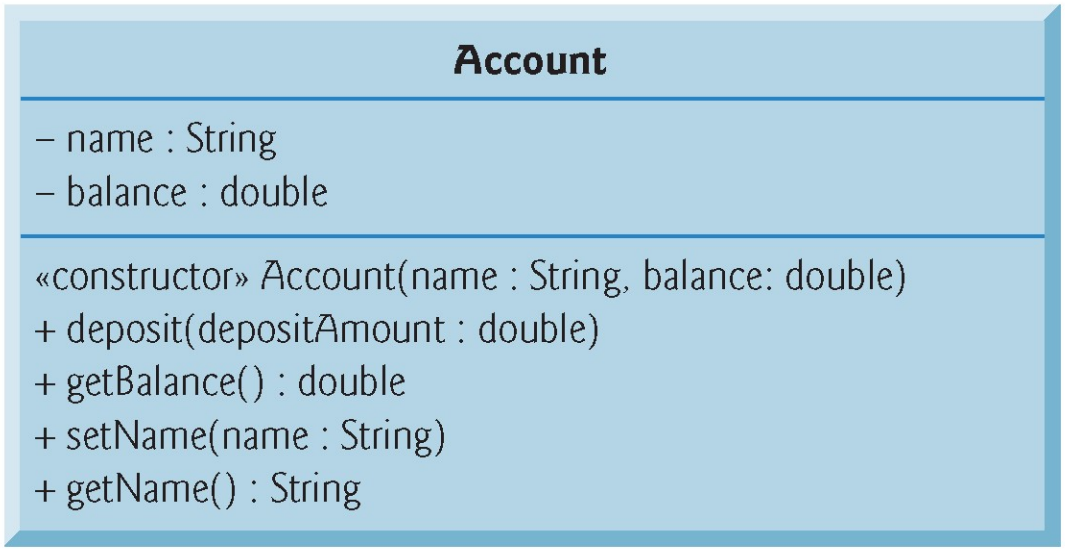

Fig. 3.10 | UML class diagram for `Account` class of Fig. 3.8.

3.5 Primitive Types vs. Reference Types

- Types in Java are divided into two categories—primitive types and reference types.
- The primitive types are `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` and `double`.
- All other types are reference types, so classes, which specify the types of objects, are reference types.
- A primitive-type variable can store exactly one value of its declared type at a time.
- Primitive-type instance variables are initialized by default.
- Variables of types `byte`, `char`, `short`, `int`, `long`, `float` and `double` are initialized to `0`.

3.5 Primitive Types vs. Reference Types (Cont.)

- Variables of type `boolean` are initialized to `false`.
- Reference-type variables (called references) store the location of an object in the computer's memory.
- Such variables refer to objects in the program.
- The object that's referenced may contain many instance variables and methods.
- Reference-type instance variables are initialized by default to the value `null`.
- A reference to an object is required to invoke an object's methods.
- A primitive-type variable does not refer to an object and therefore cannot be used to invoke a method.

3.6 (Optional) GUI and Graphics Case Study: Using Dialog Boxes

Section or Exercise	What you'll do
Section 3.6: A Simple GUI	Display text and an image.
Section 4.15: Event Handling; Drawing Lines	In response to a `Button` click, draw lines using JavaFX graphics capabilities.
Section 5.11: Drawing Rectangles and Ovals	Draw rectangles and ovals.
Section 6.13: Colors and Filled Shapes	Draw filled shapes in multiple colors.
Section 7.17: Drawing Arcs	Draw a rainbow with arcs.
Section 8.16: Using Objects with Graphics	Store shapes as objects.
Section 10.14: Drawing with Polymorphism	Identify the similarities between shape classes and create a shape class hierarchy.
Exercise 13.9: Interactive Polymorphic Drawing Application	A capstone exercise in which you'll enable users to select each shape to draw, configure its properties (such as color and fill) and drag the mouse to size the shape.

Fig. 3.11 | Summary of the GUI and Graphics Case Study in each chapter.

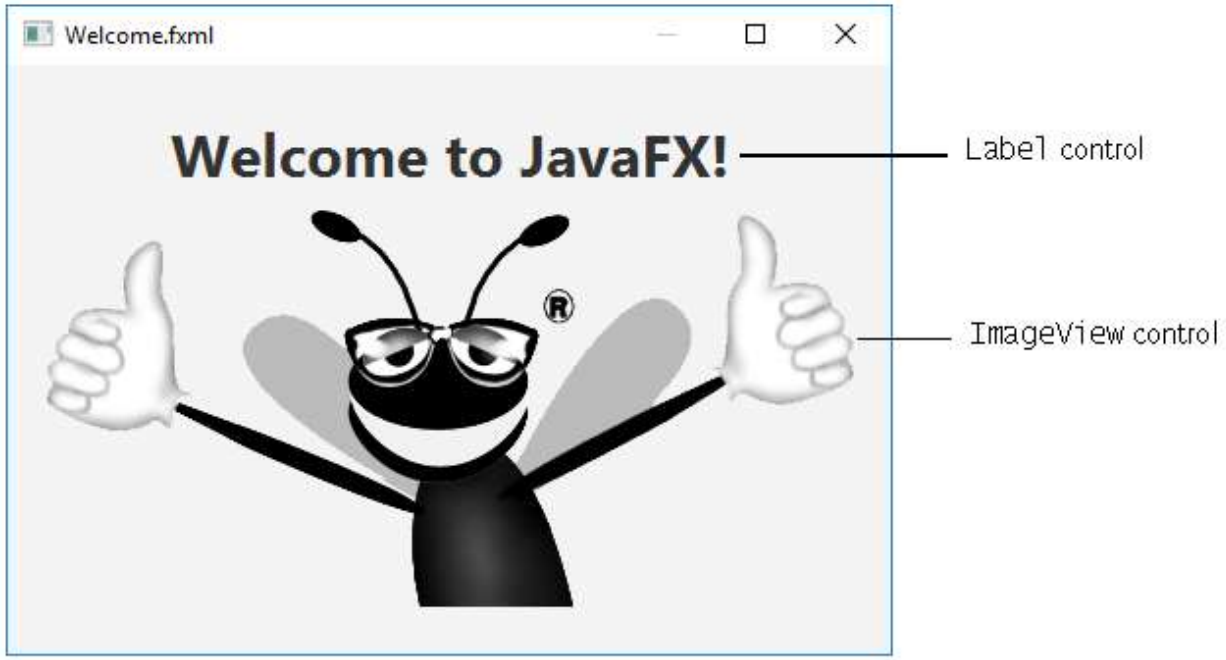

Fig. 3.12 | Final **Welcome** GUI in a preview window on Microsoft Windows 10.

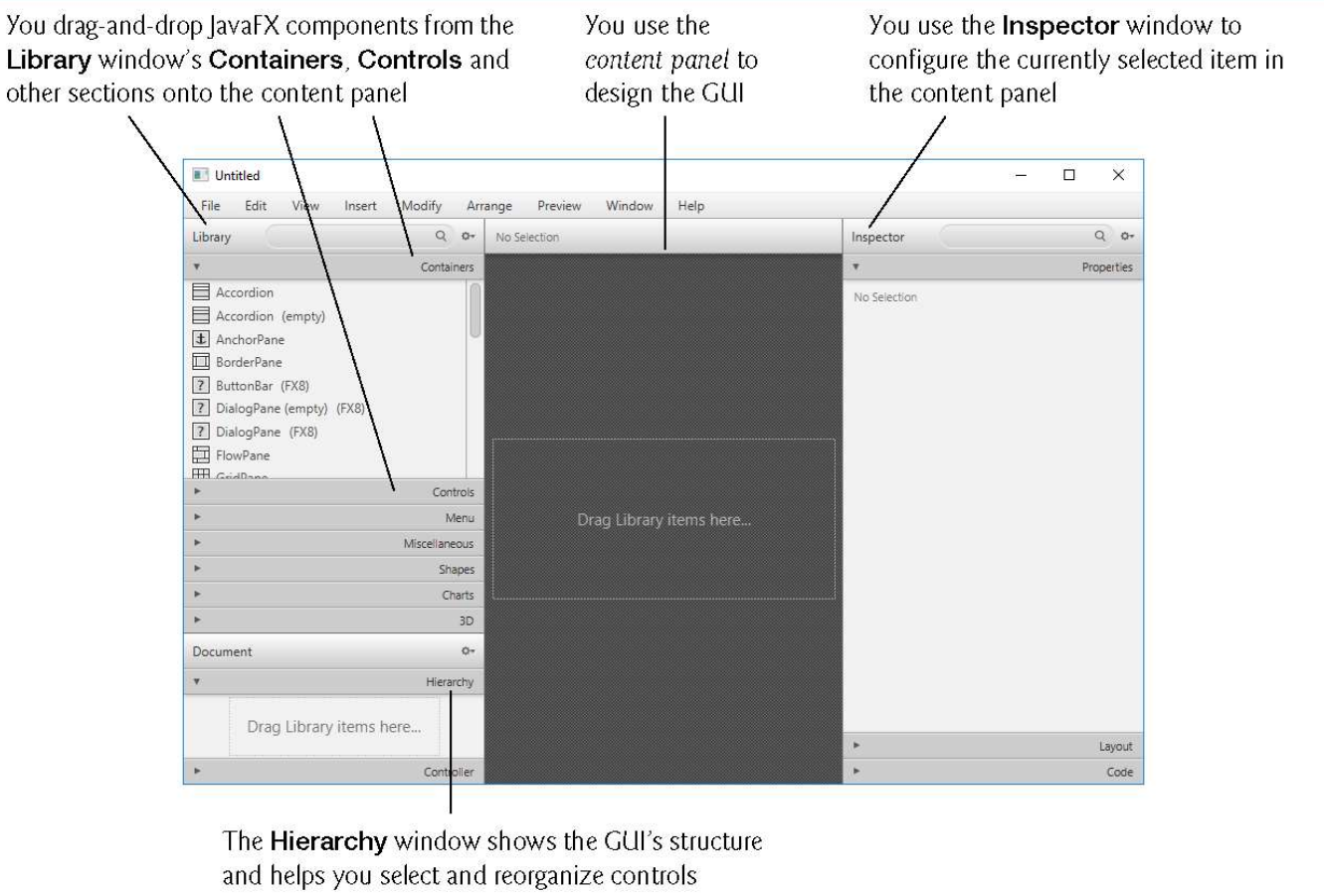

Fig. 3.13 | JavaFX Scene Builder when you first open it.

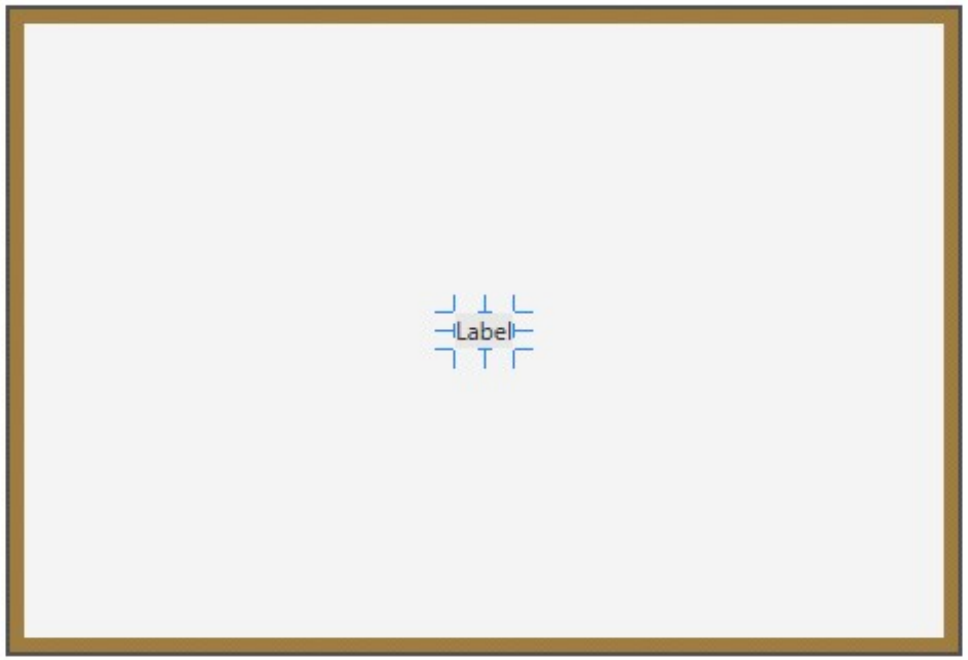

Fig. 3.14 | `Label` centered in a `VBox`.

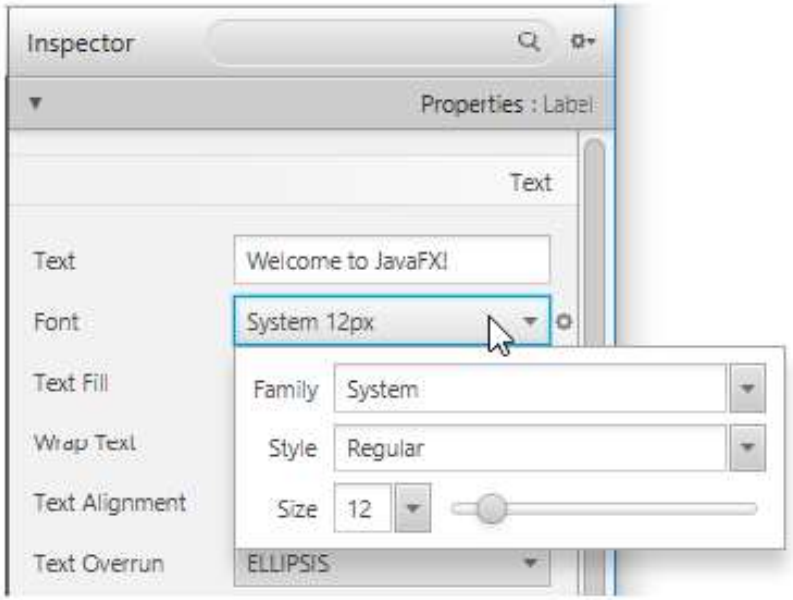

Fig. 3.15 | Setting the **Label**'s **Font** property.

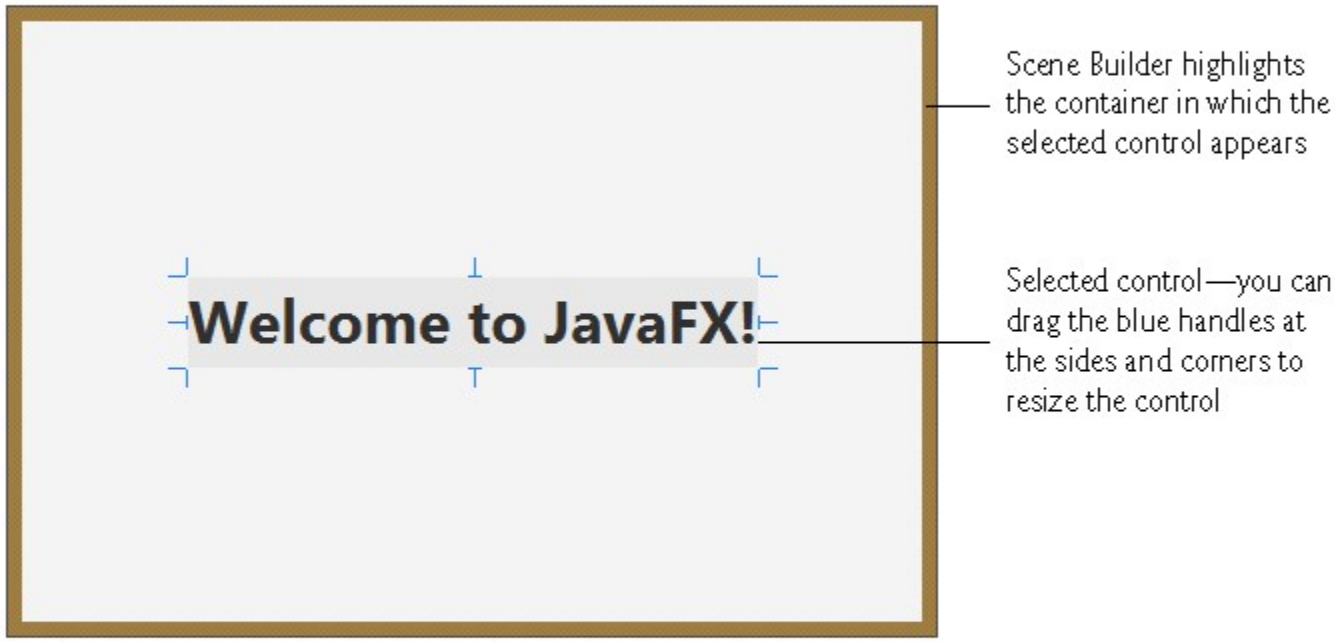

Fig. 3.16 | **Welcome** GUI's design after adding and configuring a **Label**.

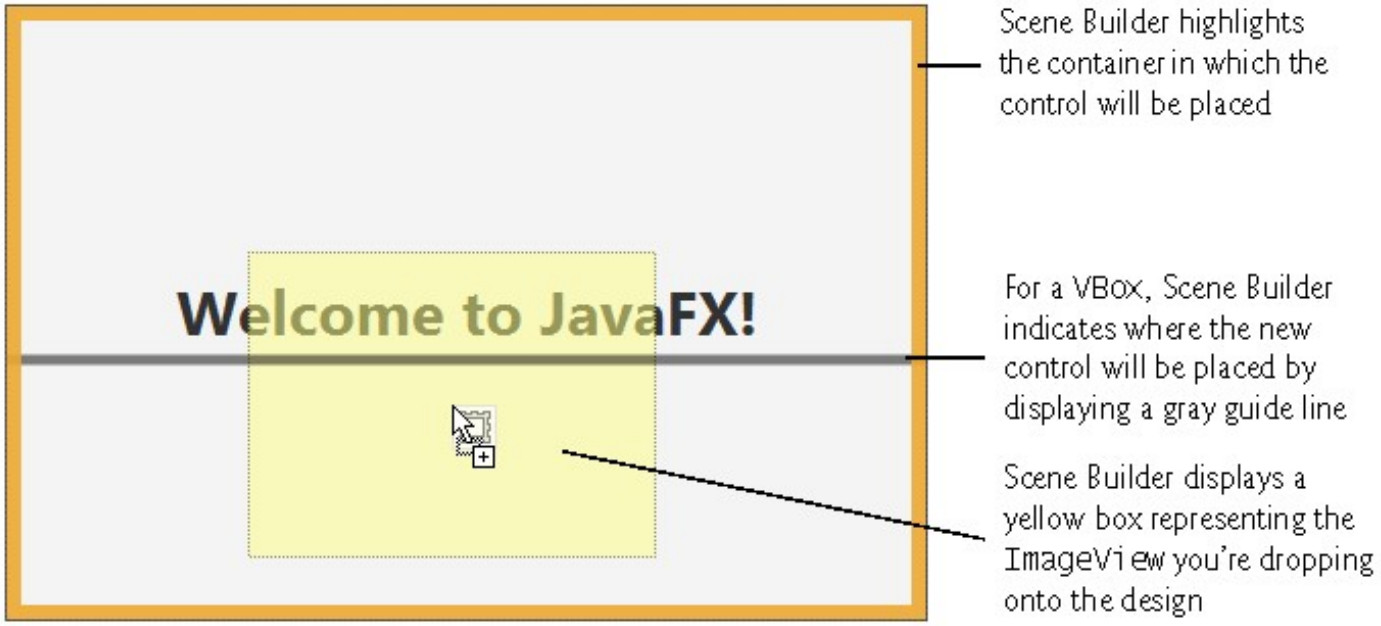

Fig. 3.17 | Dragging and dropping the **`ImageView`** below the **`Label`**.

Fig. 3.18 | Completed **Welcome** GUI in Scene Builder's content panel.

Fig. 3.19 | Previewing the **Welcome** GUI on Microsoft Windows 10—only the window borders will differ on Linux, macOS and earlier Windows versions.