Chapter 3
Introduction to Classes, Objects Methods and Strings

Java™ How to Program, 9/e

3.1 Introduction

- Covered in this chapter
  - Classes
  - Objects
  - Methods
  - Parameters
  - `double` primitive type

3.2 Declaring a Class with a Method and Instantiating an Object of a Class

- Create a new class (`GradeBook`)
- Use it to create an object.
- Each class declaration that begins with keyword `public` must be stored in a file that has the same name as the class and ends with the `.java` file-name extension.
- Keyword `public` is an access modifier.
  - Indicates that the class is "available to the public"

3.2 Declaring a Class with a Method and Instantiating an Object of a Class (Cont.)

- The `main` method is called automatically by the Java Virtual Machine (JVM) when you execute an application.
- Normally, you must call methods explicitly to tell them to perform their tasks.
- A `public` is "available to the public"
  - It can be called from methods of other classes.
- The return type specifies the type of data the method returns after performing its task.
- Return type `void` indicates that a method will perform a task but will *not* return (i.e., give back) any information to its calling method when it completes its task.

3.2 Declaring a Class with a Method and Instantiating an Object of a Class (Cont.)

- Method name follows the return type.
- By convention, method names begin with a lowercase first letter and subsequent words in the name begin with a capital letter.
- Empty parentheses after the method name indicate that the method does not require additional information to perform its task.
- Together, everything in the first line of the method is typically called the Method header
- Every method's body is delimited by left and right braces.
- The method body contains one or more statements that perform the method's task.

```
1  // Fig. 3.1: GradeBook.java
2  // Class declaration with one method.
3
4  public class GradeBook
5  {
6     // display a welcome message to the GradeBook user
7     public void displayMessage()
8     {
9        System.out.println( "Welcome to the Grade Book!" );
10    } // end method displayMessage
11 } // end class GradeBook
```

Performs the task of displaying a message on the screen; method `displayMessage` must be called to perform this task

Fig. 3.1 | Class declaration with one method.

3.2 Declaring a Class with a Method and Instantiating an Object of a Class (Cont.)

- Use class `GradeBook` in an application.
- Class `GradeBook` is not an application because it does not contain `main`.
- Can't execute `GradeBook`; will receive an error message like:
  - `Exception in thread "main" java.lang.NoSuchMethodError: main`
- Must either declare a separate class that contains a `main` method or place a `main` method in class `GradeBook`.
- To help you prepare for the larger programs, use a separate class containing method `main` to test each new class.
- Some programmers refer to such a class as a driver class.

```
1  // Fig. 3.2: GradeBookTest.java
2  // Creating a GradeBook object and calling its displayMessage method.
3
4  public class GradeBookTest
5  {
6     // main method begins program execution
7     public static void main( String[] args )
8     {
9        // create a GradeBook object and assign it to myGradeBook
10       GradeBook myGradeBook = new GradeBook();
11
12       // call myGradeBook's displayMessage method
13       myGradeBook.displayMessage();
14    } // end main
15 } // end class GradeBookTest
```

Creates a `GradeBook` object and assigns it to variable `myGradeBook`

Invokes method `displayMessage` on the GradeBook object that was assigned to variable `myGradeBook`

```
Welcome to the Grade Book!
```

Fig. 3.2 | Creating a `GradeBook` object and calling its `displayMessage` method.

3.2 Declaring a Class with a Method and Instantiating an Object of a Class (Cont.)

- A `static` method (such as `main`) is special
  - It can be called without first creating an object of the class in which the method is declared.
- Typically, you cannot call a method that belongs to another class until you create an object of that class.
- Declare a variable of the class type.
  - Each new class you create becomes a new type that can be used to declare variables and create objects.
  - You can declare new class types as needed; this is one reason why Java is known as an extensible language.

3.2 Declaring a Class with a Method and Instantiating an Object of a Class (Cont.)

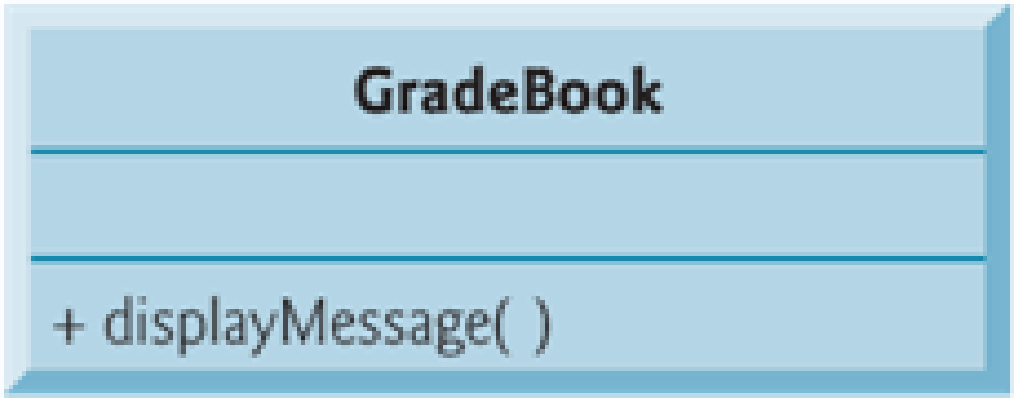
- Class instance creation expression
  - Keyword `new` creates a new object of the class specified to the right of the keyword.
  - Used to initialize a variable of a class type.
  - The parentheses to the right of the class name are required.
  - Parentheses in combination with a class name represent a call to a constructor, which is similar to a method but is used only at the time an object is created to initialize the object's data.

3.2 Declaring a Class with a Method and Instantiating an Object of a Class (Cont.)

- Call a method via the class-type variable
  - Variable name followed by a dot separator (.), the method name and parentheses.
  - Call causes the method to perform its task.
- Any class can contain a `main` method.
  - The JVM invokes the `main` method only in the class used to execute the application.
  - If multiple classes that contain `main`, then one that is invoked is the one in the class named in the `java` command.

3.3 Declaring a Class with a Method and Instantiating an Object of a Class (Cont.)

- UML class diagram for class `GradeBook`.
- Each class is modeled in a class diagram as a rectangle with three compartments.
  - Top: contains the class name centered horizontally in boldface type.
  - Middle: contains the class's attributes, which correspond to instance variables (Section 3.5).
  - Bottom: contains the class's operations, which correspond to methods.
- Operations are modeled by listing the operation name preceded by an access modifier (in this case +) and followed by a set of parentheses.
- The plus sign (+) corresponds to the keyword `public`.



3.3 Declaring a Method with a Parameter

- Car analogy
  - Pressing a car's gas pedal sends a message to the car to perform a task—make the car go faster.
  - The farther down you press the pedal, the faster the car accelerates.
  - Message to the car includes the task to perform and additional information that helps the car perform the task.
- Parameter: Additional information a method needs to perform its task.

3.3 Declaring a Method with a Parameter (Cont.)

- A method can require one or more parameters that represent additional information it needs to perform its task.
  - Defined in a comma-separated parameter list
  - Located in the parentheses that follow the method name
  - Each parameter must specify a type and an identifier.
- A method call supplies values—called arguments—for each of the method's parameters.

```
1  // Fig. 3.4: GradeBook.java
2  // Class declaration with a method that has a parameter.
3
4  public class GradeBook
5  {
6     // display a welcome message to the GradeBook user
7     public void displayMessage( String courseName )
8     {
9        System.out.printf( "Welcome to the grade book for\n%s!\n",
10          courseName );
11
12 } // end class GradeBook
```

Parameter `courseName` provides the additional information that the method requires to perform its task

Parameter `courseName`'s value is displayed as part of the output

Fig. 3.4 | Class declaration with one method that has a parameter.

```
1  // Fig. 3.5: GradeBookTest.java
2  // Create GradeBook object and pass a String to
3  // its displayMessage method.
4  import java.util.Scanner; // program uses Scanner
5
6  public class GradeBookTest
7  {
8     // main method begins program execution
9     public static void main( String[] args )
10    {
11       // create Scanner to obtain input from command window
12       Scanner input = new Scanner( System.in );
13
14       // create a GradeBook object and assign it to myGradeBook
15       GradeBook myGradeBook = new GradeBook();
16
17       // prompt for and input course name
18       System.out.println( "Please enter the course name:" );
19       String nameOfCourse = input.nextLine(); // read a line of text
20       System.out.println(); // outputs a blank line
21
```

Reads a `String` from the user

Fig. 3.5 | Creating a `GradeBook` object and passing a `String` to its `displayMessage` method. (Part 1 of 2.)

```
22          // call myGradeBook's displayMessage method
23          // and pass nameOfCourse as an argument
24          myGradeBook.displayMessage( nameOfCourse );
25       } // end main
26    } // end class GradeBookTest
```

Passes the value of `nameOfCourse` as the argument to method `displayMessage`

```
Please enter the course name:
CS101 Introduction to Java Programming

Welcome to the grade book for
CS101 Introduction to Java Programming!
```

Fig. 3.5 | Creating a `GradeBook` object and passing a `String` to its `displayMessage` method. (Part 2 of 2.)

3.3 Declaring a Method with a Parameter (Cont.)

- `Scanner` method `nextLine`
  - Reads characters typed by the user until the newline character is encountered
  - Returns a `String` containing the characters up to, but not including, the newline
  - Press *Enter* to submit the string to the program.
  - Pressing *Enter* inserts a newline character at the end of the characters the user typed.
  - The newline character is discarded by `nextLine`.
- `Scanner` method `next`
  - Reads individual words
  - Reads characters until a white-space character is encountered, then returns a `String` (the white-space character is discarded).
  - Information after the first white-space character can be read by other statements that call the `Scanner`'s methods later in the program-.

3.3 Declaring a Method with a Parameter (Cont.)

- More on Arguments and Parameters
  - The number of arguments in a method call must match the number of parameters in the parameter list of the method's declaration.
  - The argument types in the method call must be "consistent with" the types of the corresponding parameters in the method's declaration.

3.3 Declaring a Method with a Parameter (Cont.)

- The UML class diagram of Fig. 3.6 models class `GradeBook` of Fig. 3.4.
- The UML models a parameter by listing the parameter name, followed by a colon and the parameter type in the parentheses- following the operation name.
- The UML type `String` corresponds to the Java type `String`.

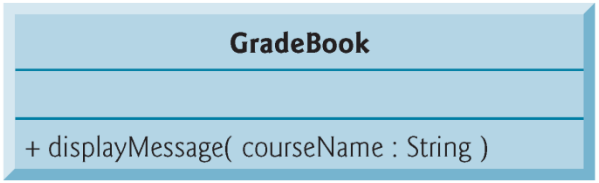


Fig. 3.6 | UML class diagram indicating that class `GradeBook` has a `displayMessage` operation with a `courseName` parameter of UML type `String`.

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

- Local variables
  - Variables declared in the body of a particular method.
  - When a method terminates, the values of its local variables are lost.
  - Recall from Section 3.2 that an object has attributes that are carried with the object as it's used in a program. Such attributes exist before a method is called on an object and after the method completes execution.

3.4 Instance Variables, *set* Methods and *get* Methods (Cont.)

- A class normally consists of one or more methods that manipulate the attributes that belong to a particular object of the class.
  - Attributes are represented as variables in a class declaration.
  - Called fields.
  - Declared inside a class declaration but outside the bodies of the class's method declarations.
- Instance variable
  - When each object of a class maintains its own copy of an attribute, the field is an instance variable
  - Each object (instance) of the class has a separate instance of the variable in memory.

```
1  // Fig. 3.7: GradeBook.java
2  // GradeBook class that contains a courseName instance variable
3  // and methods to set and get its value.
4
5  public class GradeBook
6  {
7     private String courseName; // course name for this GradeBook
8
9     // method to set the course name
10    public void setCourseName( String name )
11    {
12       courseName = name; // store the course name
13    } // end method setCourseName
14
15    // method to retrieve the course name
16    public String getCourseName()
17    {
18       return courseName;
19    } // end method getCourseName
20
```

Each GradeBook object maintains its own copy of instance variable courseName

Method allows client code to change the courseName

Method allows client code to obtain the courseName

Fig. 3.7 | GradeBook class that contains a courseName instance variable and methods to set and get its value. (Part 1 of 2.)

```
21      // display a welcome message to the GradeBook user
22      public void displayMessage()
23      {
24         // calls getCourseName to get the name of
25         // the course this GradeBook represents
26         System.out.printf( "Welcome to the grade book for\n%s!\n",
27            getCourseName() );
28      } // end method displayMessage
29   } // end class GradeBook
```

No parameter required; all methods in this class already know about instance variable `courseName` and the class's other methods

Good practice to access your instance variables via *set* or *get* methods

Fig. 3.7 | `GradeBook` class that contains a `courseName` instance variable and methods to set and get its value. (Part 2 of 2.)

3.4 Instance Variables, *set* Methods and *get* Methods (Cont.)

- Every instance (i.e., object) of a class contains one copy of each instance variable.
- Instance variables typically declared `private`.
  - `private` is an access modifier.
  - `private` variables and methods are accessible only to methods of the class in which they are declared.
- Declaring instance `private` is known as data hiding or information hiding.
- `private` variables are encapsulated (hidden) in the object and can be accessed only by methods of the object's class.
  - Prevents instance variables from being modified accidentally by a class in another part of the program.
  - *Set* and *get* methods used to access instance variables.

```
1  // Fig. 3.8: GradeBookTest.java
2  // Creating and manipulating a GradeBook object.
3  import java.util.Scanner; // program uses Scanner
4
5  public class GradeBookTest
6  {
7     // main method begins program execution
8     public static void main( String[] args )
9     {
10       // create Scanner to obtain input from command window
11       Scanner input = new Scanner( System.in );
12
13       // create a GradeBook object and assign it to myGradeBook
14       GradeBook myGradeBook = new GradeBook();
15
16       // display initial value of courseName
17       System.out.printf( "Initial course name is: %s\n\n",
18          myGradeBook.getCourseName() );
19
20       // prompt for and read course name
21       System.out.println( "Please enter the course name:" );
22       String theName = input.nextLine(); // read a line of text
23       myGradeBook.setCourseName( theName ); // set the course name
```

Gets the value of the `myGradeBook` object's `courseName` instance variable

Sets the value of the `courseName` instance variable

Fig. 3.8 | Creating and manipulating a `GradeBook` object. (Part 1 of 2.)

```
24         System.out.println(); // outputs a blank line
25
26         // display welcome message after specifying course name
27         myGradeBook.displayMessage();
28      } // end main
29   } // end class GradeBookTest
```

Displays the GradeBook's message, including the value of the `courseName` instance variable

```
Initial course name is: null

Please enter the course name:
CS101 Introduction to Java Programming

Welcome to the grade book for
CS101 Introduction to Java Programming!
```

Fig. 3.8 | Creating and manipulating a `GradeBook` object. (Part 2 of 2.)

3.4 Instance Variables, *set* Methods and *get* Methods (Cont.)

- *set* and *get* methods
  - A class's `private` fields can be manipulated only by the class's methods.
  - A client of an object calls the class's `public` methods to manipulate the `private` fields of an object of the class.
  - Classes often provide `public` methods to allow clients to *set* (i.e., assign values to) or *get* (i.e., obtain the values of) `private` instance variables.
  - The names of these methods need not begin with *set* or *get*, but this naming convention is recommended.

3.4 Instance Variables, *set* Methods and *get* Methods (Cont.)

- Figure 3.9 contains an updated UML class diagram for the version of class `GradeBook` in Fig. 3.7.
  - Models instance variable `courseName` as an attribute in the middle compartment of the class.
  - The UML represents instance variables as attributes by listing the attribute name, followed by a colon and the attribute type.
  - A minus sign (–) access modifier corresponds to access modifier `private`.

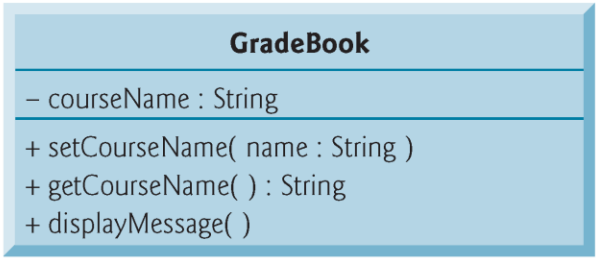


Fig. 3.9 | UML class diagram indicating that class `GradeBook` has a private `courseName` attribute of UML type `String` and three public operations—`setCourseName` (with a `name` parameter of UML type `String`), `getCourseName` (which returns UML type `String`) and `displayMessage`.

3.5 Primitive Types vs. Reference Types

- Types are divided into primitive types and reference types.
- The primitive types are `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` and `double`.
- All nonprimitive types 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, and variables of type `boolean` are initialized to `false`.
- You can specify your own initial value for a primitive-type variable by assigning the variable a value in its declaration.

3.5 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.
- Objects that are referenced may each contain many instance variables and methods.
- Reference-type instance variables are initialized by default to the value `null`
  - A reserved word that represents a "reference to nothing."
- 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.

3.6 Initializing Objects with Constructors (Cont.)

- By default, the compiler provides a default constructor with no parameters in any class that does not explicitly include a constructor.
  - Instance variables are initialized to their default values.
- Can provide your own constructor to specify custom initialization for objects of your class.
- 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 create a default constructor for that class.*

```
1  // Fig. 3.10: GradeBook.java
2  // GradeBook class with a constructor to initialize the course name.
3
4  public class GradeBook
5  {
6     private String courseName; // course name for this GradeBook
7
8     // constructor initializes courseName with String argument
9     public GradeBook( String name )
10    {
11       courseName = name; // initializes courseName
12    } // end constructor
13
14    // method to set the course name
15    public void setCourseName( String name )
16    {
17       courseName = name; // store the course name
18    } // end method setCourseName
19
```

Constructor that initializes `courseName` to the specified value

Fig. 3.10 | `GradeBook` class with a constructor to initialize the course name. (Part 1 of 2.)

```
20      // method to retrieve the course name
21      public String getCourseName()
22      {
23         return courseName;
24      } // end method getCourseName
25
26      // display a welcome message to the GradeBook user
27      public void displayMessage()
28      {
29         // this statement calls getCourseName to get the
30         // name of the course this GradeBook represents
31         System.out.printf( "Welcome to the grade book for\n%s!\n",
32            getCourseName() );
33      } // end method displayMessage
34   } // end class GradeBook
```

Fig. 3.10 | `GradeBook` class with a constructor to initialize the course name. (Part 2 of 2.)

```
1  // Fig. 3.11: GradeBookTest.java
2  // GradeBook constructor used to specify the course name at the
3  // time each GradeBook object is created.
4
5  public class GradeBookTest
6  {
7     // main method begins program execution
8     public static void main( String[] args )
9     {
10       // create GradeBook object
11       GradeBook gradeBook1 = new GradeBook(
12          "CS101 Introduction to Java Programming" );
13       GradeBook gradeBook2 = new GradeBook(
14          "CS102 Data Structures in Java" );
15
16       // display initial value of courseName for each GradeBook
17       System.out.printf( "gradeBook1 course name is: %s\n",
18          gradeBook1.getCourseName() );
19       System.out.printf( "gradeBook2 course name is: %s\n",
20          gradeBook2.getCourseName() );
21    } // end main
22 } // end class GradeBookTest
```

Class instance creation expression initializes the GradeBook and returns a reference that is assigned to variable gradeBook1

Class instance creation expression initializes the GradeBook and returns a reference that is assigned to variable gradeBook1

Fig. 3.11 | GradeBook constructor used to specify the course name at the time each GradeBook object is created. (Part 1 of 2.)

```
gradeBook1 course name is: CS101 Introduction to Java Programming
gradeBook2 course name is: CS102 Data Structures in Java
```

Fig. 3.11 | `GradeBook` constructor used to specify the course name at the time each `GradeBook` object is created. (Part 2 of 2.)

3.6 Initializing Objects with Constructors (Cont.)

- The UML class diagram of Fig. 3.12 models class `GradeBook` of Fig. 3.10, which has a constructor that has a `name` parameter of type `String`.
- Like operations, the UML models constructors in the third compartment of a class in a class diagram.
- To distinguish a constructor, the UML requires that the word "constructor" be placed between guillemets (« and ») before the constructor's name.
- List constructors before other operations in the third compartment.

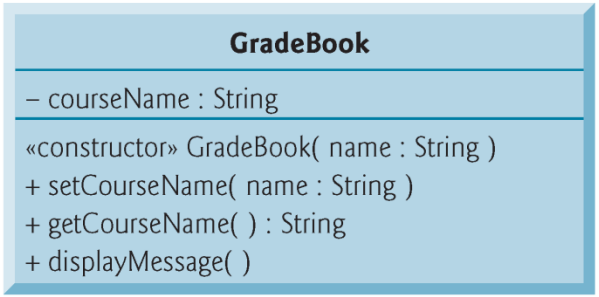


Fig. 3.12 | UML class diagram indicating that class `GradeBook` has a constructor that has a `name` parameter of UML type `String`.

3.7 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 seven 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.

```
1  // Fig. 3.13: Account.java
2  // Account class with a constructor to validate and
3  // initialize instance variable balance of type double.
4
5  public class Account
6  {
7     private double balance; // instance variable that stores the balance
8
9     // constructor
10    public Account( double initialBalance )
11    {
12       // validate that initialBalance is greater than 0.0;
13       // if it is not, balance is initialized to the default value 0.0
14       if ( initialBalance > 0.0 )
15          balance = initialBalance;
16    } // end Account constructor
17
18    // credit (add) an amount to the account
19    public void credit( double amount )
20    {
21       balance = balance + amount; // add amount to balance
22    } // end method credit
```

Floating-point number for the account balance

Parameter used to initialize the `balance` instance variable

Validating the parameter's value to ensure that it is greater than 0

Initializes `gradeCounter` to 1; indicates first grade about to be input

Fig. 3.13 | `Account` class with a constructor to validate and initialize instance variable `balance` of type `double`. (Part 1 of 2.)

```
23
24      // return the account balance
25      public double getBalance()
26      {
27         return balance; // gives the value of balance to the calling method
28      } // end method getBalance
29   } // end class Account
```

Returns the value of the `balance` instance variable as a `double`

Fig. 3.13 | `Account` class with a constructor to validate and initialize instance variable `balance` of type `double`. (Part 2 of 2.)

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

- `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.

```
1  // Fig. 3.14: AccountTest.java
2  // Inputting and outputting floating-point numbers with Account objects.
3  import java.util.Scanner;
4
5  public class AccountTest
6  {
7     // main method begins execution of Java application
8     public static void main( String[] args )
9     {
10       Account account1 = new Account( 50.00 ); // create Account object
11       Account account2 = new Account( -7.53 ); // create Account object
12
13       // display initial balance of each object
14       System.out.printf( "account1 balance: $%.2f\n",
15          account1.getBalance() );
16       System.out.printf( "account2 balance: $%.2f\n\n",
17          account2.getBalance() );
18
19       // create Scanner to obtain input from command window
20       Scanner input = new Scanner( System.in );
21       double depositAmount; // deposit amount read from user
```

Output floating-point values with two-digits of precision

Fig. 3.14 | Inputting and outputting floating-point numbers with `Account` objects. (Part 1 of 3.)

```
22
23          System.out.print( "Enter deposit amount for account1: " ); // prompt
24          depositAmount = input.nextDouble(); // obtain user input
25          System.out.printf( "\nadding %.2f to account1 balance\n\n",
26             depositAmount );
27          account1.credit( depositAmount ); // add to account1 balance
28
29          // display balances
30          System.out.printf( "account1 balance: $%.2f\n",
31             account1.getBalance() );
32          System.out.printf( "account2 balance: $%.2f\n\n",
33             account2.getBalance() );
34
35          System.out.print( "Enter deposit amount for account2: " ); // prompt
36          depositAmount = input.nextDouble(); // obtain user input
37          System.out.printf( "\nadding %.2f to account2 balance\n\n",
38             depositAmount );
39          account2.credit( depositAmount ); // add to account2 balance
40
41          // display balances
42          System.out.printf( "account1 balance: $%.2f\n",
43             account1.getBalance() );
```

Returns a `double` value typed by the user

Fig. 3.14 | Inputting and outputting floating-point numbers with `Account` objects. (Part 2 of 3.)

```
44         System.out.printf( "account2 balance: $%.2f\n",
45            account2.getBalance() );
46      } // end main
47   } // end class AccountTest
```

```
account1 balance: $50.00
account2 balance: $0.00

Enter deposit amount for account1: 25.53

adding 25.53 to account1 balance

account1 balance: $75.53
account2 balance: $0.00

Enter deposit amount for account2: 123.45

adding 123.45 to account2 balance

account1 balance: $75.53
account2 balance: $123.45
```

Fig. 3.14 | Inputting and outputting floating-point numbers with `Account` objects. (Part 3 of 3.)

3.7 Floating-Point Numbers and Type `double` (Cont.)

- The UML class diagram in Fig. 3.15 models class `Account` of Fig. 3.13.

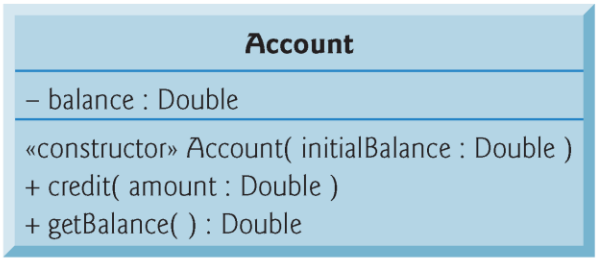


Fig. 3.15 | UML class diagram indicating that class `Account` has a `private` `balance` attribute of UML type `Double`, a constructor (with a parameter of UML type `Double`) and two `public` operations—`credit` (with an `amount` parameter of UML type `Double`) and `getBalance` (returns UML type `Double`).