CMPE 101
Object Oriented Programming

İstanbul
Bilgi University

Dr. Emel Küpçü

Department of Computer Engineering

İstanbul Bilgi University

Methods, Constructors, Parameters, Primitive Data Types and Encapsulation

2.7 Arithmetic Operator

Java operation	Operator	Algebraic expression	Java expression
Addition	+	$f + 7$	`f + 7`
Subtraction	-	$p - c$	`p - c`
Multiplication	*	bm	`b * m`
Division	/	x / y or $\frac{x}{y}$ or $x \div y$	`x / y`
Remainder	%	$r \bmod s$	`r % s`

Fig. 2.11 | Arithmetic operators.

2.7 Arithmetic Operator (cont.)

- The asterisk (*) indicates multiplication
- The percent sign (%) is the remainder operator
- The remainder operator, %, yields the remainder after division.
- Parentheses are used to group terms in expressions in the same manner as in algebraic expressions.
- If an expression contains nested parentheses, the expression in the innermost set of parentheses is evaluated first.
- As in algebra, it's acceptable to place *redundant parentheses* (unnecessary parentheses) in an ex-pression to make the expression clearer.

2.7 Arithmetic Operator (cont.)

- Rules of operator precedence
  - Multiplication, division and remainder operations are applied first.
  - If an expression contains several such operations, they are applied from left to right.
  - Multiplication, division and remainder operators have the same level of precedence.
  - Addition and subtraction operations are applied next.
  - If an expression contains several such operations, the operators are applied from left to right.
  - Addition and subtraction operators have the same level of precedence.

2.7 Arithmetic Operator (cont.)

Operator(s)	Operation(s)	Order of evaluation (precedence)
*	Multiplication	Evaluated first. If there are several operators of this type, they're evaluated from *left to right*.
/	Division	
%	Remainder	
+	Addition	Evaluated next. If there are several operators of this type, they're evaluated from *left to right*.
-	Subtraction	
=	Assignment	Evaluated last.

Fig. 2.12 | Precedence of arithmetic operators.

Step 1. `y = 2 * 5 * 5 + 3 * 5 + 7;` *(Leftmost multiplication)*

Fig. 2.13 | Order in which a second-degree polynomial is evaluated.

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

```
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.)

Each class you create becomes a new type that can be used to declare variables and create objects.

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.

```
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 {
```

Identifiers and Camel Case Naming

- Class, method and variable names are identifiers.
- By convention all use camel case names.
  - **Camel case** is a naming style where multiple words are joined without spaces, and each word (except the first in lower camel case) starts with a capital letter, like myVariableName or MyClassName.
- Class names begin with an uppercase letter, and method and variable names begin with a lowercase letter.

```
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      }
```

3.2.1 Instance Variable

- 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.
  - Instance variables exist before, during, and after method execution in an object.
- 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' method declarations.
- Each object (instance) of the class has its own copy of each of the class's instance variables.

```
public class Account {
   private String name; // (Instance variable) Stores the account holder's name.

   // Sets the account holder's name
   public void setName(String name) {
      this.name = name; // Assigns the given name to the instance variable
   }
   // Retrieves the account holder's name
   public String getName() {
      return name; // Returns the stored name
   }
}
```

Access Modifiers

- **`public`** → The member is accessible from **anywhere** in the program.
- **`protected`** → The member is accessible **within the same package** and **in subclasses**.
- **`private`** → The member is accessible **only within the same class** and **nowhere else**.
  - Variables or methods declared with access modifier **`private`** are accessible only to methods of the class in which they're declared.
- Most instance-variable declarations are preceded with the keyword **`private`**, which is an access modifier.

```
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) {
```

Encapsulation

- Encapsulation is a core idea in object-oriented programming (OOP).
- It groups the data (variables) and the methods (functions) that work on that data into one unit, called a class.
- It limits direct access to some parts of an object.
- It allows access only through specific public methods (getters and setters).
- **Key Benefits of Encapsulation:**

1. **Data Protection**: By hiding the internal state of an object, encapsulation ensures that data can only be accessed or modified through controlled methods, reducing the risk of unintended or harmful changes.
2. **Code Maintenance**: Makes it easier to change or update the internal implementation of a class without affecting other parts of the program.
3. **Flexibility**: Allows the class to control how its data is accessed or modified, providing flexibility to add validation logic.

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

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

```
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  }
```

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

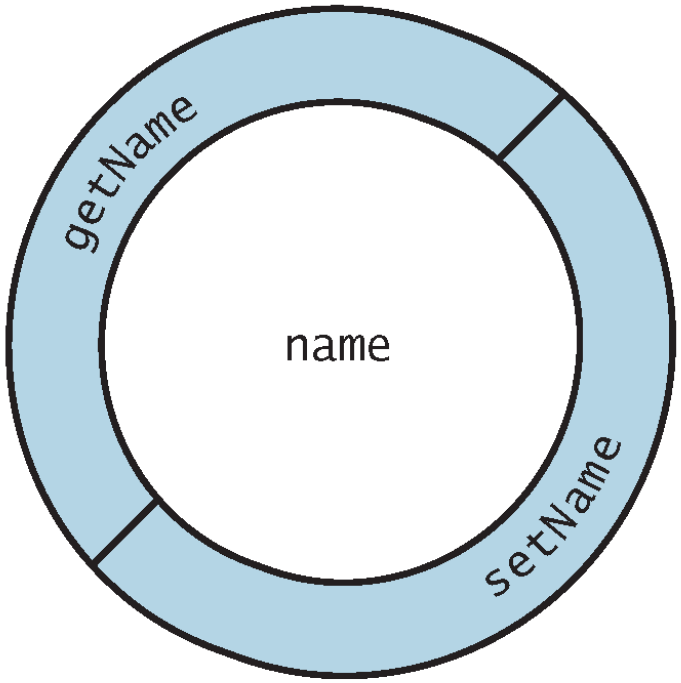

Fig. 3.4 | Conceptual view of an `Account` object with its encapsulated `private` instance variable `name` and protective layer of `public` methods.

Parameters of a Method

- 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.

Example:

```
public void displayInfo(String name, int age, double height) {
   System.out.println("Name: " + name);
   System.out.println("Age: " + age);
   System.out.println("Height: " + height + " meters");
}
```

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).

```
 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       }
```

Return Type of a Method

- 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.
- A method with a return type must send a result back to the caller when it finishes.

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

- The return statement sends a value from a method back to the caller.
- Classes often provide `public` methods to allow the class' 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.

```
public class Account {
   private String name; // Stores the account holder's name

   // Sets the account holder's name
   public void setName(String name) {
      // Assigns the given name to the instance variable
      this.name = name;
   }
   // Retrieves the account holder's name
   public String getName() {
      return name; // Returns the stored name
   }
}
```

This keyword

this is a reference variable that refers to the **current object** of a class.

```
public class Account {
    private String name; // Stores the account holder's name (Instance variable)

    // Sets the account holder's name
    public void setName(String name) {// Parameter name matches instance variable
        this.name = name; // 'this' differentiates instance variable from parameter
    }

    // Retrieves the account holder's name
    public String getName() {
        return name; // Returns the stored name
    }
}
```

name = name;

?X

This keyword (cont.)

- We could have avoided using **`this`** by giving the parameter a different name in line 9. However, using this in line 10 is a common practice because it keeps the code simple and prevents unnecessary variable names.

```java
public class Account {
    private String name; // Stores the account holder's name (Instance variable)

    // Sets the account holder's name
    public void setName(String name) {// Parameter name matches instance variable
        this.name = name; // 'this' differentiates instance variable from parameter
    }

    // Retrieves the account holder's name
    public String getName() {
        return name; // Returns the stored name
    }
}
```

```java
public class Account {
    private String name; // Instance variable

    // Sets the account holder's name
    public void setName(String newname) {// Different parameter name
        name = newname; // No need for 'this'
    }

    // Retrieves the account holder's name
    public String getName() {
        return name; // Returns the stored name
    }
}
```

Account Class

```
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.

3.2.2 AccountTest 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
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 }
```

Fig. 3.2

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`.

```
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    }
```

Instantiating an Object—Keyword new

- A class instance creation expression begins with keyword **new** and creates a new object.

```
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
```

Calling Class's 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.

```
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
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 }
```

Common Programming Error 3.1

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

```
int myVari
able = 10;
```

```
int myVariable = 10;
```

```
String message = "Hello, this is a very long
string.";
```

```
String message = "Hello, this is a very long " +
                 "string.";
```

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.

```
public class LocalVariableExampleClass {
    public void displayMessage(String message) { // 'message' is a local variable (parameter)
        int length = message.length(); // 'length' is a local variable declared inside the method
        System.out.println("Message: " + message);
        System.out.println("Message Length: " + length);
    } // Both 'message' and 'length' are lost after method execution
    public static void main(String[] args) {
        LocalVariableExample obj = new LocalVariableExample();
        obj.displayMessage("Hello, Java!"); // Passing argument to local variable 'message'
    }
}
```

The Default Value of Instance Variable

- **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`
  - **`int`** is `0`
  - **`double`** is `0.0`
  - **`boolean`** is `false`
  - …

- **Local variables are not automatically initialized.**
  - If you declare a local variable without initializing it, **it will not have a default value** like instance variables do.
  - Instead, trying to use an uninitialized local variable will result in a **compilation error**.

3.2.5 Notes on import Declarations on Class AccountTest

- 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.
- An `import` declaration is not required when one class in a package uses another in the same package.
- 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.

```
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
```

Not in the same package with AccountTest class

In the same package with AccountTest class

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.

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.
- ***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.

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.

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

```
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.

Primitive Data Types

Data Type	Size (in bytes)	Range	Example
byte	1	Stores whole numbers from -128 to 127	byte b = 100;
short	2	Stores whole numbers from -32,768 to 32,767	short s = 1000;
int	4	Stores whole numbers from -2,147,483,648 to 2,147,483,647	int x = 25;
long	8	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	long l = 50000L;
float	4	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits	float f = 3.14f;
double	8	Stores fractional numbers. Sufficient for storing 15 decimal digits	double d = 5.67;
char	2	Stores a single character/letter or ASCII values	char c = 'A';
boolean	1	Stores true or false values	boolean isTrue = true;

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`**, **`short`**, **`int`**, **`long`**, **`float`** and **`double`** are initialized to **`0`**.

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.
- **Reference-type variables** refer to **objects** in the program.
- **Reference-type instance variables** are initialized by default to the value **`null`**.
- You need a reference to an object to call its methods.
- An object can have many instance variables and methods.
- A primitive-type variable doesn't refer to an object, so it cannot be used to call methods.

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.
- **`double`** variables require **twice as much memory as `float` variables and provide 15 significant digits**—approximately double the precision of `float` variables.

Type	Assigned Value	Stored Value(approx.)	Digit Preserved
Float	1234567.1234567f	1234567.1(rounded)	~7
Double	1234567.1234567	1234567.1234567(exact).	~15

Fig 3.8 Account Class with a double instance variable balance and a constructor and deposit method that perform validation

```
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    }
```

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

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.)

- 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.

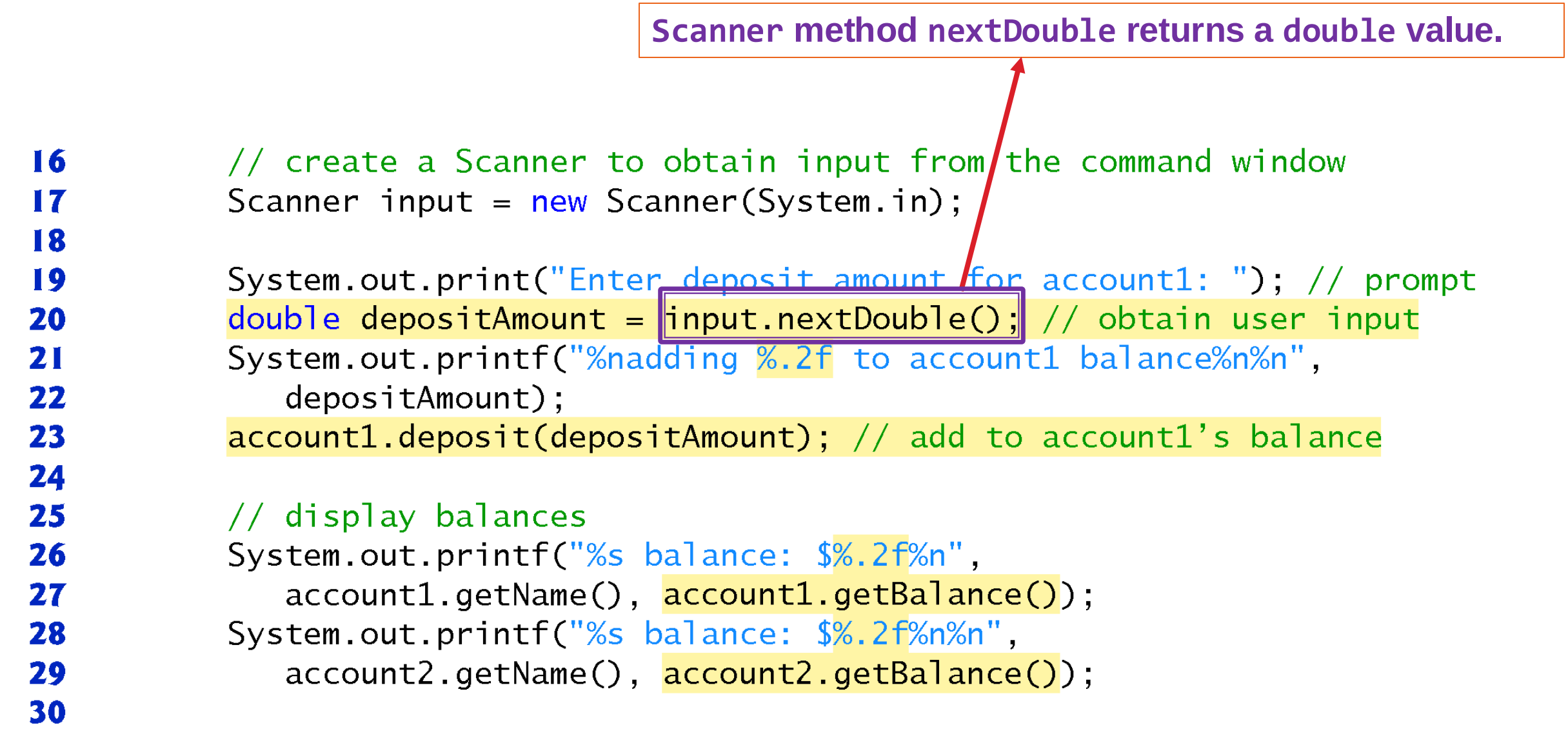

Fig. 3.9 | Inputting and outputting floating-point numbers with `Account` objects. (Part 2 of 4.)

```
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.)

Group Discussion Question Part-1

```
import java.util.Scanner; // Import Scanner class to take input
class Member {
    // Step 1: Create private attributes: name(string), age(int), sport(string), height (float), weight (double)
```

```
    // Step 2: Setter methods to set the values of the attributes
```

```
    // Step 3: Getter methods to retrieve the values of the attributes
```

```
    // Step 4: Method (displayMemberInfo) to display the member's details
```

```
}
```

Group Discussion Question Part-2

```
public class SportsClub {
    public static void main(String[] args) {
        // Step 5: Take user input and set the member's details using setter methods and
        // Scanner class' methods: nextLine(), nextInt(), nextFloat(), nextDouble()
```

```
        // Step 6: Display the member's details using the method displayMemberInfo()
```

```
    }
}
```

Questions?