Basics of Java Programming

1

Programmer I Exam Objectives		
[1.2]	Define the structure of a Java class ❍ *See also §3.1, p. 48.*	*§1.2, p. 2*
[1.3]	Create executable Java applications with a main method; run a Java program from the command line; including console output ❍ *See also §4.3, p. 107.*	*§1.10, p. 16*
[1.5]	Compare and contrast the features and components of Java such as: platform independence, object orientation, encapsulation, etc.	*§1.12, p. 21*
[2.3]	Know how to read or write to object fields	*§1.3, p. 4*
Supplementary Objectives		
•	Introduce the basic terminology and concepts in object-oriented programming: classes, objects, references, fields, methods, members, inheritance, and associations	*Chapter 1*
•	Format and print values to the terminal window	*§1.11, p. 18*

1.1 Introduction

Before embarking on the road to Java programmer certification, it is important to understand the basic terminology and concepts in object-oriented programming (OOP). In this chapter, the emphasis is on providing an introduction to OOP, rather than exhaustive coverage. In-depth coverage of the concepts follows in subsequent chapters of the book.

Java supports the writing of many different kinds of executables: applications, applets, and servlets. The basic elements of a Java application are introduced in this chapter. The old adage that practice makes perfect is certainly true when learning a programming language. To encourage programming on the computer, the mechanics of compiling and running a Java application are outlined.

1.2 Classes

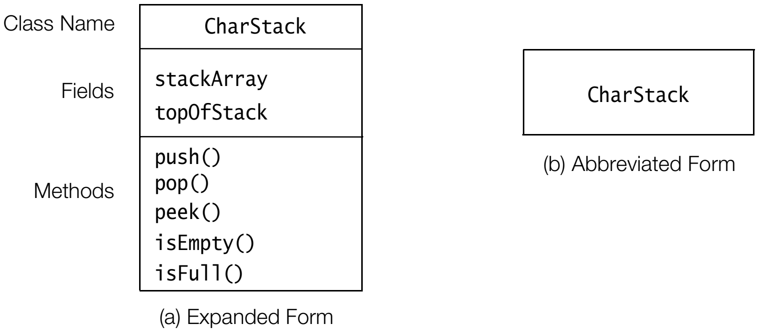
One of the fundamental ways in which we handle complexity is by using *abstractions*. An abstraction denotes the essential properties and behaviors of an object that differentiate it from other objects. The essence of OOP is modeling abstractions, using classes and objects. The hard part of this endeavor is finding the right abstraction.

A *class* denotes a category of objects, and acts as a blueprint for creating objects. A class models an abstraction by defining the properties and behaviors for the objects representing the abstraction. An *object* exhibits the properties and behaviors defined by its class. The properties of an object of a class are also called *attributes*, and are defined by fields in Java. A *field* in a class is a variable that can store a value that represents a particular property of an object. The behaviors of an object of a class are also known as *operations*, and are defined using *methods* in Java. Fields and methods in a class declaration are collectively called *members*.

An important distinction is made between the *contract* and the *implementation* that a class provides for its objects. The contract defines *which* services are provided, and the implementation defines *how* these services are provided by the class. Clients (i.e., other objects) need to know only the contract of an object, and not its implementation, to avail themselves of the object's services.

As an example, we will implement different versions of a class that models the abstraction of a stack that can push and pop characters. The stack will use an array of characters to store the characters, and a field to indicate the top element in the stack. Using Unified Modeling Language (UML) notation, a class called `CharStack` is graphically depicted in Figure 1.1, which models the abstraction. Both fields and method names are shown in Figure 1.1a.

Figure 1.1 *UML Notation for Classes*



(a) Expanded Form

(b) Abbreviated Form

Declaring Members: Fields and Methods

Example 1.1 shows the declaration of the class `CharStack` depicted in Figure 1.1. Its intention is to illustrate the salient features of a class declaration in Java, rather than an effective implementation of stacks. The character sequence `//` in the code indicates the start of a *single-line comment* that can be used to document the code. All characters after this sequence and to the end of the line are ignored by the compiler.

A class declaration contains member declarations that define the fields and the methods of the objects the class represents. In the case of the class `CharStack`, it has two fields declared at (1):

- `stackArray`, which is an array to hold the elements of the stack (in this case, characters)
- `topOfStack`, which denotes the top element of the stack (i.e., the index of the last character stored in the array)

The class `CharStack` has five methods, declared at (3), that implement the essential operations on a stack:

- `push()` pushes a character on to the stack.
- `pop()` removes and returns the top element of the stack.
- `peek()` returns the top element of the stack for inspection.
- `isEmpty()` determines whether the stack is empty.
- `isFull()` determines whether the stack is full.

The class declaration also has a method-like declaration at (2) with the same name as the class. Such declarations are called *constructors*. As we shall see, a constructor is executed when an object is created from the class. However, the implementation details in the example are not important for the present discussion.

Example 1.1 *Basic Elements of a Class Declaration*

```
// File: CharStack.java
public class CharStack {        // Class name
  // Class Declarations:

  // Fields:                                                   (1)
  private char[] stackArray;     // The array implementing the stack
  private int    topOfStack;     // The top of the stack

  // Constructor:                                              (2)
  public CharStack(int capacity) {
    stackArray = new char[capacity];
    topOfStack = -1;
  }

  // Methods:                                                  (3)
  public void push(char element) { stackArray[++topOfStack] = element; }
  public char pop()              { return stackArray[topOfStack--]; }
  public char peek()             { return stackArray[topOfStack]; }
  public boolean isEmpty()       { return topOfStack == -1; }
  public boolean isFull()        { return topOfStack == stackArray.length - 1; }
}
```

1.3 Objects

Class Instantiation, Reference Values, and References

The process of creating objects from a class is called *instantiation*. An *object* is an instance of a class. The object is constructed using the class as a blueprint and is a concrete instance of the abstraction that the class represents. An object must be created before it can be used in a program.

A *reference value* is returned when an object is created. A reference value denotes a particular object. A *variable* denotes a location in memory where a value can be stored. An *object reference* (or simply *reference*) is a variable that can store a reference value. Thus a reference provides a handle to an object, as it can indirectly denote an object whose reference value it holds. In Java, an object can be manipulated only via its reference value, or equivalently by a reference that holds its reference value.

This setup for manipulating objects requires that a reference be declared, a class be instantiated to create an object, and the reference value of the object created be stored in the reference. These steps are accomplished by a *declaration statement*.

```
CharStack stack1 = new CharStack(10); // Stack length: 10 chars
```

In the preceding declaration statement, the left-hand side of the `=` operator declares that `stack1` is a reference of class `CharStack`. The reference `stack1`, therefore, can refer to objects of class `CharStack`.

The right-hand side of the `=` operator creates an object of class `CharStack`. This step involves using the `new` operator in conjunction with a call to a constructor of the class (`new CharStack(10)`). The `new` operator creates an instance of the `CharStack` class and returns the reference value of this instance. The `=` operator (called the *assignment operator*) stores the reference value in the reference `stack1` declared on the left-hand side of the assignment operator. The reference `stack1` can now be used to manipulate the object whose reference value is stored in it.

Analogously, the following declaration statement declares the reference `stack2` to be of class `CharStack`, creates an object of class `CharStack`, and assigns its reference value to the reference `stack2`:

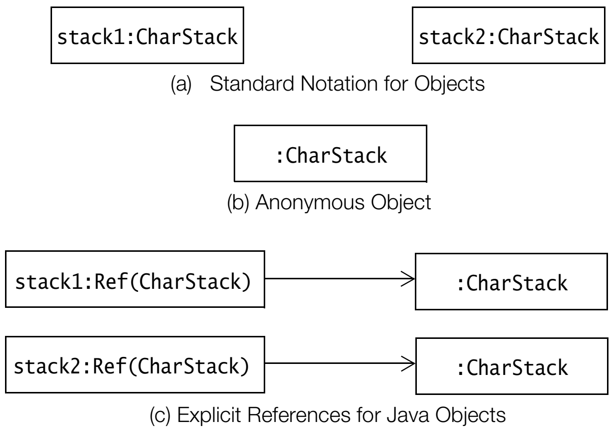
```
CharStack stack2 = new CharStack(5);  // Stack length: 5 chars
```

Each object that is created has its own copy of the fields declared in the class declaration in Example 1.1. That is, the two stack objects, referenced by `stack1` and `stack2`, will have their own `stackArray` and `topOfStack` fields.

The purpose of the constructor call on the right-hand side of the `new` operator is to initialize the newly created object. In this particular case, for each new `CharStack` object created using the `new` operator, the constructor at (2) in Example 1.1 creates an array of characters. The length of this array is given by the value of the argument to the constructor. The constructor also initializes the `topOfStack` field.

Figure 1.2 shows the UML notation for objects. The graphical representation of an object is very similar to that of a class. Figure 1.2 shows the canonical notation, where the name of the reference denoting the object is prefixed to the class name with a colon (`:`). If the name of the reference is omitted, as in Figure 1.2b, this denotes an anonymous object. Since objects in Java do not have names, but rather are denoted by references, a more elaborate notation is shown in Figure 1.2c, where references of the `CharStack` class explicitly refer to `CharStack` objects. In most cases, the more compact notation will suffice.

Figure 1.2 *UML Notation for Objects*



(a) Standard Notation for Objects

(b) Anonymous Object

(c) Explicit References for Java Objects

Object Aliases

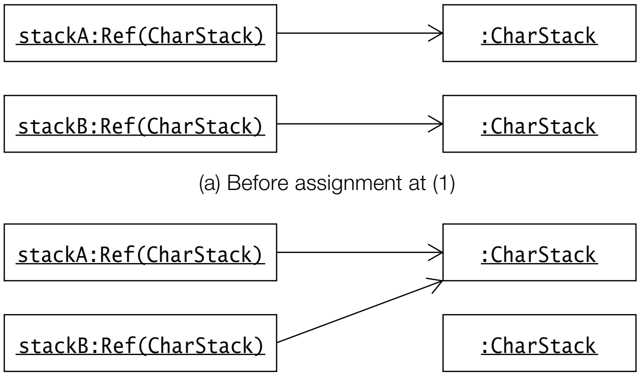
Several references can refer to the same object, meaning that they store the reference value of the same object. Such references are called *aliases*. The object can be manipulated via any one of its aliases, as each one refers to the same object.

```
// Create two distinct stacks of chars.
CharStack stackA = new CharStack(12); // Stack length: 12 chars
CharStack stackB = new CharStack(6);  // Stack length: 6 chars

stackB = stackA;                      // (1) aliases after assignment
// The stack previously referenced by stackB can now be garbage collected.
```

Two stack objects are created in the preceding code. Before the assignment at (1), the situation is as depicted in Figure 1.3a. After the assignment at (1), the references `stackA` and `stackB` will denote the same stack, as depicted in Figure 1.3b. The *reference value* in `stackA` is assigned to `stackB`. The references `stackA` and `stackB` are aliases after the assignment, as they refer to the same object. What happens to the stack object that was denoted by the reference `stackB` before the assignment? When objects are no longer in use, their memory is, if necessary, reclaimed and reallocated for other objects. This process is called *automatic garbage collection*. Garbage collection in Java is taken care of by the runtime environment.

Figure 1.3 *Aliases*



(a) Before assignment at (1)

(b) After assignment at (1)

1.4 Instance Members

Each object created will have its own copies of the fields defined in its class. The fields of an object are called *instance variables*. The values of the instance variables in an object constitute its *state*. Two distinct objects can have the same state if their instance variables have the same values. The methods of an object define its behavior; such methods are called *instance methods*. It is important to note that these methods pertain to each object of the class. In contrast, the *implementation* of the methods is shared by all instances of the class. Instance variables and instance methods, which belong to objects, are collectively called *instance members*, to distinguish them from *static members*, which belong to the class only. Static members are discussed in §1.5.

Invoking Methods

Objects communicate by message passing. As a consequence, an object can be made to exhibit a particular behavior by sending the appropriate message to the object. In Java, this is done by *calling* a method on the object using the binary dot (`.`) operator. A *method call* spells out the complete message: the object that is the receiver of the message, the method to be invoked, and the arguments to be passed to the method, if any. The method invoked on the receiver can also send information back to the sender, via a single return value. The method called must be one that is defined for the object; otherwise, the compiler reports an error.

```
CharStack stack = new CharStack(5);      // Create a stack
stack.push('J');            // (1) Character 'J' pushed
char c = stack.pop();       // (2) One character popped and returned: 'J'
stack.printStackElements(); // (3) Compile-time error: No such method in CharStack
```

The sample code given here invokes methods on the object denoted by the reference `stack`. The method call at (1) pushes one character on the stack, and the method call at (2) pops one character off the stack. Both the `push()` and `pop()` methods are defined in the class `CharStack`. The `push()` method does not return any value, but the `pop()` method returns the character popped. Trying to invoke a method named `printStackElements` on the stack results in a compile-time error, as no such method is defined in the class `CharStack`.

The dot (`.`) notation can also be used with a reference to access the fields of an object. Use of the dot notation is governed by the *accessibility* of the member. The fields in the class `CharStack` have `private` accessibility, indicating that they are not accessible from outside the class. Thus the following code in a client of the `CharStack` class will not compile:

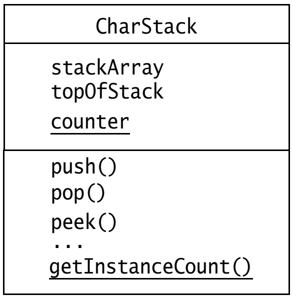
```
stack.topOfStack++;     // Compile-time error: topOfStack is not visible.
```

1.5 Static Members

In some cases, certain members should belong only to the class; that is, they should not be part of any instance of the class. As an example, suppose a class wants to keep track of how many objects of the class have been created. Defining a counter as an instance variable in the class declaration for tracking the number of objects created does not solve the problem. Each object created will have its own counter field. Which counter should then be updated? The solution is to declare the counter field as being `static`. Such a field is called a *static variable*. It belongs to the class, rather than to any specific object of the class. A static variable is initialized when the class is loaded at runtime. Similarly, a class can have *static methods* that belong to the class, rather than to any specific objects of the class. Static variables and static methods are collectively known as *static members*, and are declared with the keyword `static`.

Figure 1.4 shows the class diagram for the class `CharStack`. It has been augmented by two static members, whose names are underlined. The augmented definition of the `CharStack` class is given in Example 1.2. The field `counter` is a static variable declared at (1). It will be allocated and initialized to the default value `0` when the class is loaded. Each time an object of the `CharStack` class is created, the constructor at (2) is executed. The constructor explicitly increments the counter in the class. The method `getInstanceCount()` at (3) is a static method belonging to the class. It returns the counter value when called.

Figure 1.4 *Class Diagram Showing Static Members of a Class*



Example 1.2 *Static Members in Class Declaration*

```
// File: CharStack.java
public class CharStack {
  // Instance variables:
  private char[] stackArray;     // The array implementing the stack
  private int    topOfStack;     // The top of the stack

  // Static variable
  private static int counter;                                      // (1)

  // Constructor now increments the counter for each object created.
  public CharStack(int capacity) {                                 // (2)
    stackArray = new char[capacity];
    topOfStack = -1;
    counter++;
  }

  // Instance methods:
  public void push(char element) { stackArray[++topOfStack] = element; }
  public char pop()              { return stackArray[topOfStack--]; }
  public char peek()             { return stackArray[topOfStack]; }
  public boolean isEmpty()       { return topOfStack == -1; }
  public boolean isFull()        { return topOfStack == stackArray.length - 1; }

  // Static method                                                    (3)
  public static int getInstanceCount() { return counter; }
}
```

Figure 1.5 shows the classification of the members in the class `CharStack`, using the terminology we have introduced so far. Table 1.1 provides a summary of the terminology used in defining members of a class.

Clients can access static members in the class by using the class name. The following code invokes the `getInstanceCount()` method in the class `CharStack`:

```
int count = CharStack.getInstanceCount(); // Class name to invoke static method
```

Static members can also be accessed via object references, although doing so is considered bad style:

```
CharStack myStack = new CharStack(20);
int count = myStack.getInstanceCount();   // Reference invokes static method
```

Static members in a class can be accessed both by the class name and via object references, but instance members can be accessed only by object references.

Figure 1.5 *Members of a Class*

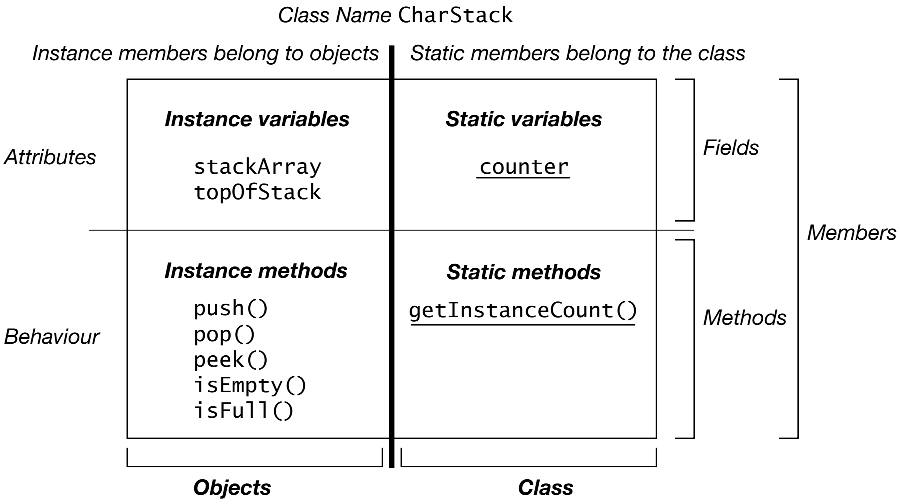


Table 1.1 *Terminology for Class Members*

Instance members	The instance variables and instance methods of an object. They can be accessed or invoked only through an object reference.
Instance variable	A field that is allocated when the class is instantiated (i.e., when an object of the class is created). Also called a *non-static field* or just a *field* when the context is obvious.
Instance method	A method that belongs to an instance of the class. Objects of the same class share its implementation.

Continues

Table 1.1 *Terminology for Class Members (Continued)*

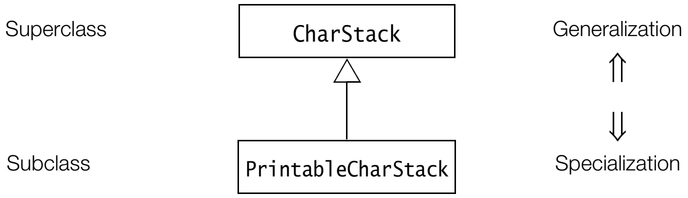
Static members	The static variables and static methods of a class. They can be accessed or invoked either by using the class name or through an object reference.
Static variable	A field that is allocated when the class is loaded. It belongs to the class, and not to any specific object of the class. Also called a *static field* or a *class variable*.
Static method	A method that belongs to the class, and not to any object of the class. Also called a *class method*.

1.6 Inheritance

There are two fundamental mechanisms for building new classes from existing ones: *inheritance* and *aggregation*. It makes sense to *inherit* from an existing class `Vehicle` to define a class `Car`, since a car is a vehicle. The class `Vehicle` has several *parts*; therefore, it makes sense to define a *composite object* of the class `Vehicle` that has *constituent objects* of such classes as `Engine`, `Axle`, and `GearBox`, which make up a vehicle.

Inheritance is illustrated here by an example that implements a stack of characters that can print its elements on the terminal. This new stack has all the properties and behaviors of the `CharStack` class, along with the additional capability of printing its elements. Given that this printable stack is a stack of characters, it can be derived from the `CharStack` class. This relationship is shown in Figure 1.6. The class `PrintableCharStack` is called the *subclass*, and the class `CharStack` is called the *superclass*. The `CharStack` class is a *generalization* for all stacks of characters, whereas the class `PrintableCharStack` is a *specialization* of stacks of characters that can also print their elements.

Figure 1.6 *Class Diagram Depicting Inheritance Relationship*



In Java, deriving a new class from an existing class requires the use of the `extends` clause in the subclass declaration. A subclass can *extend* only one superclass. The subclass can inherit members of the superclass. The following code fragment implements the `PrintableCharStack` class:

```
class PrintableCharStack extends CharStack {                    // (1)
  // Instance method
  public void printStackElements() {                            // (2)
    // ... implementation of the method...
  }
  // The constructor calls the constructor of the superclass explicitly.
  public PrintableCharStack(int capacity) { super(capacity); }      // (3)
}
```

The `PrintableCharStack` class extends the `CharStack` class at (1). Implementing the `printStackElements()` method in the `PrintableCharStack` class requires access to the field `stackArray` from the superclass `CharStack`. However, this field is *private* and, therefore, not accessible in the subclass. The subclass can access these fields if the accessibility of the fields is changed to *protected* in the `CharStack` class. Example 1.3 uses a version of the class `CharStack`, which has been modified to support this access. Implementation of the `printStackElements()` method is shown at (2). The constructor of the `PrintableCharStack` class at (3) calls the constructor of the superclass `CharStack` to initialize the stack properly.

Example 1.3 *Defining a Subclass*

```
// File: CharStack.java
public class CharStack {
  // Instance variables
  protected char[] stackArray;  // The array that implements the stack
  protected int    topOfStack;  // The top of the stack

  // The rest of the definition is the same as in Example 1.2.
}
```

```
// File: PrintableCharStack.java
public class PrintableCharStack extends CharStack {                // (1)

  // Instance method
  public void printStackElements() {                               // (2)
    for (int i = 0; i <= topOfStack; i++)
      System.out.print(stackArray[i]); // Print each char on terminal
    System.out.println();
  }

  // Constructor calls the constructor of the superclass explicitly.
  PrintableCharStack(int capacity) { super(capacity); }            // (3)
}
```

Objects of the `PrintableCharStack` class will respond just like the objects of the `CharStack` class, but they also have the additional functionality defined in the subclass:

```
PrintableCharStack pcStack = new PrintableCharStack(3);
pcStack.push('H');
pcStack.push('i');
pcStack.push('!');
pcStack.printStackElements();    // Prints "Hi!" on the terminal
```

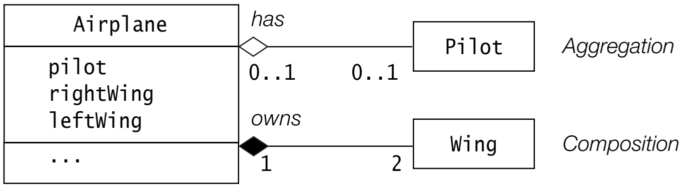
1.7 Associations: Aggregation and Composition

An *association* defines a static relationship between objects of two classes. One such association, called *aggregation*, expresses how an object uses other objects. Java supports aggregation of objects by reference, since objects cannot contain other objects explicitly. The aggregate object usually has fields that denote its constituent objects. A constituent object can be *shared* with other aggregate objects.

For example, an object of class `Airplane` might have a field that denotes an object of class `Pilot`. This `Pilot` object of an `Airplane` object might be shared among other aggregate objects (not necessarily `Airplane` objects) once the pilot has finished duty on one airplane. In fact, the `Pilot` object can still be used even when its `Airplane` object no longer exists. This *aggregation* relationship is depicted by the UML diagram in Figure 1.7 (empty diamond), showing that each object of the `Airplane` class *has* zero or one object of class `Pilot` associated with it.

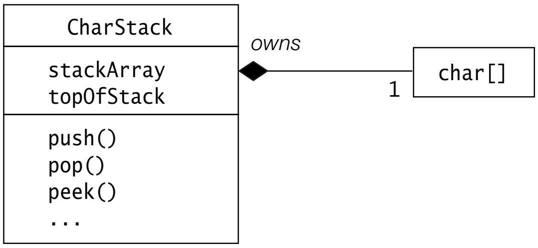
The aggregate association can be made stronger if the constituent objects cannot be shared with other aggregate objects—for example, an `Airplane` object with two `Wing` objects. The `Wing` objects cannot be shared and can exist only with their `Airplane` object; that is, the `Airplane` object has *ownership* of its `Wing` objects. Conversely, the `Wing` objects are a *part of* their `Airplane` object. This stronger aggregation association is called *composition* and is depicted by the UML diagram in Figure 1.7 (filled diamond), showing that each object of the `Airplane` class *owns* two objects of class `Wing`.

Figure 1.7 *Class Diagram Depicting Associations*



In the case of the `CharStack` class used in the earlier examples, each object of this class has a field to store the reference value of an array object that holds the characters. It would not be a good idea to share this array with other stack objects. The stack owns the array of characters. The relationship between the stack object and its constituent array object can be expressed by composition (Figure 1.8), showing that each object of the `CharStack` class will own one array object of type `char` associated with it.

Figure 1.8 *Class Diagram Depicting Composition*



1.8 Tenets of Java

- Code in Java must be encapsulated in classes.
- There are two kinds of values in Java: objects that are instances of classes or arrays, and *atomic* values of primitive data types.
- References store reference values that denote objects, and are used to manipulate objects.
- Objects in Java cannot contain other objects; they can only have references to other objects.
- During execution, reclamation of objects that are no longer in use is managed by the runtime environment.

Review Questions

1.1 Which statement is true about methods?

Select the one correct answer.

(a) A method is an implementation of an abstraction.
(b) A method is an attribute defining the property of a particular abstraction.
(c) A method is a category of objects.
(d) A method is an operation defining the behavior for a particular abstraction.
(e) A method is a blueprint for making operations.

1.2 Which statement is true about objects?

Select the one correct answer.

(a) An object is what classes are instantiated from.
(b) An object is an instance of a class.
(c) An object is a blueprint for creating concrete realization of abstractions.
(d) An object is a reference.
(e) An object is a variable.

1.3 Which is the first line of a constructor declaration in the following code?

```
public class Counter {                                          // (1)
  int current, step;
  public Counter(int startValue, int stepValue) {               // (2)
    setCurrent(startValue);                                     // (3)
    setStep(stepValue);
  }
  public int  getCurrent()          { return current; }         // (4)
  public void setCurrent(int value)  { current = value; }        // (5)
  public void setStep(int stepValue) { step = stepValue; }       // (6)
}
```

Select the one correct answer.

(a) (1)
(b) (2)
(c) (3)
(d) (4)
(e) (5)
(f) (6)

1.4 Given that `Thing` is a class, how many objects and how many references are created by the following code?

```
Thing item, stuff;
item = new Thing();
Thing entity = new Thing();
```

Select the two correct answers.

(a) One object is created.
(b) Two objects are created.
(c) Three objects are created.
(d) One reference is created.
(e) Two references are created.
(f) Three references are created.

1.5 Which statement is true about instance members?

Select the one correct answer.

(a) An instance member is also called a static member.
(b) An instance member is always a field.
(c) An instance member is never a method.
(d) An instance member belongs to an instance, not to the class as a whole.
(e) An instance member always represents an operation.

1.6 How do objects communicate in Java?

Select the one correct answer.

(a) They communicate by modifying each other's fields.
(b) They communicate by modifying the static variables of each other's classes.
(c) They communicate by calling each other's instance methods.
(d) They communicate by calling static methods of each other's classes.

1.7 Given the following code, which statements are true?

```
class A {
  protected int value1;
}

class B extends A {
  int value2;
}
```

Select the two correct answers.

(a) Class `A` extends class `B`.
(b) Class `B` is the superclass of class `A`.
(c) Class `A` inherits from class `B`.
(d) Class `B` is a subclass of class `A`.
(e) Objects of class `A` have a field named `value2`.
(f) Objects of class `B` have a field named `value1`.

1.8 Given the following code, which statements express the most accurate association?

```
class Carriage { }

class TrainDriver { }

class Train {
  private Carriage[] carriages;
  private TrainDriver driver;
  Train(TrainDriver trainDriver, int noOfCarriages) {
    carriages = new Carriage[noOfCarriages];
    driver = trainDriver;
  }
  void insertCarriage(Carriage newCarriage) { /* ... */ }
}
```

Select the three correct answers.

(a) A `Train` object *has* an array of `Carriage` objects.
(b) A `Train` object *owns an* array of `Carriage` objects.
(c) A `Train` object *owns* `Carriage` objects.
(d) A `Train` object *has a* `TrainDriver` object.
(e) A `Train` object *owns a* `TrainDriver` object.
(f) A `TrainDriver` object is *part of* a `Train` object.
(g) An array of `Carriage` objects is *part of* a `Train` object.
(h) `Carriage` objects are *part of* a `Train` object.

1.9 Java Programs

A Java *source file* can contain more than one class declaration. Each source file name has the extension `.java`. The JDK (Java Development Kit) enforces the rule that any class in the source file that has `public` accessibility must be declared in its own file, meaning that such a public class must be declared in a source file whose file name comprises the name of this public class with `.java` as its extension. This rule implies that a source file can contain at most one public class. If the source file contains a public class, the file naming rule is enforced by the JDK.

Each class declaration in a source file is compiled into a separate *class file*, containing *Java bytecode*. The name of this file comprises the name of the class with `.class` as its extension. The JDK provides tools for compiling and running programs, as explained in the next section. The classes in the Java SE platform API are already compiled, and the JDK tools know where to find them.

1.10 Sample Java Application

The term *application* is just a synonym for a *program*, referring to source code that is compiled and directly executed. To create an application in Java, the program must have a class that defines a method named `main`, which is the starting point for the execution of any application.

Essential Elements of a Java Application

Example 1.4 is an example of an application in which a client uses the `CharStack` class to reverse a string of characters.

Example 1.4 *An Application*

```
// File: CharStack.java
public class CharStack {
  // Same as in Example 1.2.
}
```

```
// File: Client.java
public class Client {

  public static void main(String[] args) {

    // Create a stack.
    CharStack stack = new CharStack(40);

    // Create a string to push on the stack:
    String str = "!no tis ot nuf era skcatS";
    System.out.println("Original string: " + str);            // (1)
    int length = str.length();

    // Push the string char by char onto the stack:
    for (int i = 0; i < length; i++) {
      stack.push(str.charAt(i));
    }
    System.out.print("Reversed string: ");                    // (2)
    // Pop and print each char from the stack:
    while (!stack.isEmpty()) { // Check if the stack is not empty.
      System.out.print(stack.pop());
    }
    System.out.println();                                     // (3)
  }
}
```

Output from the program:

```
Original string: !no tis ot nuf era skcatS
Reversed string: Stacks are fun to sit on!
```

The public class `Client` defines a method with the name `main`. To start the application, the `main()` method in this public class is invoked by the Java interpreter, also called the Java Virtual Machine (JVM). The *method header* of this `main()` method must be declared as shown in the following method stub:

```
public static void main(String[] args)   // Method header
{ /* Implementation */ }
```

The `main()` method has `public` accessibility—that is, it is accessible from any class. The keyword `static` means the method belongs to the class. The keyword `void` indicates that the method does not return any value. The parameter `args` is an array of strings that can be used to pass information to the `main()` method when execution starts.

Compiling and Running an Application

Java source files can be compiled using the Java compiler tool `javac`, which is part of the JDK.

The source file `Client.java` contains the declaration of the `Client` class. This source file can be compiled by giving the following command at the command line (the character `>` is the command prompt):

```
>javac Client.java
```

This command creates the class file `Client.class` containing the Java bytecode for the `Client` class. The `Client` class uses the `CharStack` class, and if the file `CharStack.class` does not already exist, the compiler will also compile the source file `CharStack.java`.

Compiled classes can be executed by the Java interpreter `java`, which is also part of the JDK. To run Example 1.4, give the following command on the command line:

```
>java Client
Original string: !no tis ot nuf era skcatS
Reversed string: Stacks are fun to sit on!
```

Note that only the name of the class is specified, resulting in the execution starting in the `main()` method of the specified class. The application in Example 1.4 terminates when the execution of the `main()` method is completed.

1.11 Program Output

Data produced by a program is called *output*. This output can be sent to different devices. The examples presented in this book send their output to a terminal window, where the output is printed as line of characters with a cursor that advances as characters are printed. A Java program can send its output to the terminal window using an object called *standard out*. This object, which can be accessed using the public static final field `out` in the `System` class, is an object of the class `java.io.PrintStream` that provides methods for printing values. These methods convert values to their string representation and print the resulting string.

Example 1.4 illustrates the process of printing values to the terminal window. The argument in the call to the `println()` method at (1) is first evaluated, and the resulting string is printed to the terminal window. This method always terminates the current line, which results in the cursor being moved to the beginning of the next line:

```
System.out.println("Original string: " + str);            // (1)
```

The `print()` method at (2) prints its argument to the terminal window, but it does not terminate the current line:

```
System.out.print("Reversed string: ");                     // (2)
```

To terminate a line, without printing any values, we can use the no-argument `println()` method:

```
System.out.println();                                      // (3)
```

Formatted Output

To have more control over how the values are printed, we can create formatted output. The following method of the `java.io.PrintStream` class can be used for this purpose:

> `PrintStream printf(String format, Object... args)`
>
> The `String` parameter `format` specifies how formatting will be done. It contains *format specifications* that determine how each subsequent value in the parameter `args` will be formatted and printed. The parameter declaration `Object... args` represents an array of zero or more arguments to be formatted and printed. The resulting string from the formatting will be printed to the *destination stream*. (`System.out` will print to the *standard out* object.)
>
> Any error in the format string will result in a runtime exception.

The following call to the `printf()` method on the standard out object formats and prints three values:

```
System.out.printf("Formatted values|%5d|%8.3f|%5s|%n", // Format string
                  2016, Math.PI, "Hi");               // Values to format
```

At runtime, the following line is printed in the terminal window:

```
Formatted values| 2016|   3.142|   Hi|
```

The format string is the first argument in the method call. It contains four *format specifiers*. The first three are `%5d`, `%8.3f`, and `%5s`, which specify how the three arguments should be processed. The letter in the format specifier indicates the type of value to format. Their location in the format string specifies where the textual representation of the arguments should be inserted. The fourth format specifier, `%n`, is a platform-specific line separator. Its occurrence causes the current line to be terminated, with the cursor moving to the start of the next line. All other text in the format string is fixed, including any other spaces or punctuation, and is printed verbatim.

In the preceding example, the first value is formatted according to the first format specifier, the second value is formatted according to the second format specifier, and so on. The `|` character has been used in the format string to show how many character positions are taken up by the text representation of each value. The output shows that the `int` value was written right-justified, spanning five character positions using the format specifier `%5d`; the `double` value of `Math.PI` took up eight character positions and was rounded to three decimal places using the format specifier `%8.3f`; and the `String` value was written right-justified, spanning five character positions using the format specifier `%5s`. The format specifier `%n` terminates the current line. All other characters in the format string are printed verbatim.

Table 1.2 shows examples of some selected format specifiers that can be used to format values. Their usage is illustrated in Example 1.5, which prints a simple invoice.

At the top of the invoice printed by Example 1.5, the company name is printed at (1) with a format string that contains only fixed text. The date and time of day are printed on the same line, with leading zeros at (2). A header is then printed at (3). The column names `Item`, `Price`, `Quantity`, and `Amount` are positioned appropriately with the format specifications `%-20s`, `%7s`, `%9s`, and `%8s`, respectively.

Beneath the heading, the items purchased are printed at (5), (6), and (7) using the same field widths as the column headings. The format for each item is defined by the format string at (4). The item name is printed with the format string `"%-20s"`, resulting in a 20-character-wide string, left-justified. The item price and the total amount for each type of item are printed as floating-point values using the format specifications `%7.2f` and `%8.2f`, respectively. The quantity is printed as an integer using the format specification `%9d`. The strings are left-justified, while all numbers are right-justified. The character `s` is the conversion code for objects, while floating-point and integer values are printed using the codes `f` and `d`, respectively.

Table 1.2 *Format Specifier Examples*

Parameter value	Format spec	Example value	String printed	Description
Integer value	`"%d"`	`123`	`"123"`	Occupies as many character positions as needed.
	`"%6d"`	`123`	`"   123"`	Occupies six character positions and is right-justified. The printed string is padded with leading spaces, if necessary.
	`"%06d"`	`123`	`"000123"`	Occupies six character positions and is right-justified. The printed string is padded with leading zeros, if necessary.
Floating-point value	`"%f"`	`4.567`	`"4.567000"`	Occupies as many character positions as needed, but always includes six decimal places.
	`"%.2f"`	`4.567`	`"4.57"`	Occupies as many character positions as needed, but includes only two decimal places. The value is rounded in the output, if necessary.
	`"%6.2f"`	`4.567`	`"  4.57"`	Occupies six character positions, including the decimal point, and uses two decimal places. The value is rounded in the output, if necessary.
Any object	`"%s"`	`"Hi!"`	`"Hi!"`	The string representation of the object occupies as many character positions as needed.
	`"%6s"`	`"Hi!"`	`"   Hi!"`	The string representation of the object occupies six character positions and is right-justified.
	`"%-6s"`	`"Hi!"`	`"Hi!   "`	The string representation of the object occupies six character positions and is left-justified.

At (8), the total cost of all items is printed using the format specification `%8.2f`. To position this value correctly under the column `Amount`, we print the string `"Total:"` using the format `%-36s`. The width of 36 characters is found by adding the width of the first three columns of the invoice.

Example 1.5 *Formatted Output*

```
// File: Invoice.java
public class Invoice {
  public static void main(String[] args) {
    System.out.printf("Secure Data Inc.       ");                     // (1)
    System.out.printf("%02d/%02d/%04d, %02d:%02d%n%n",                // (2)
                       2, 13, 2016, 11, 5);
    System.out.printf("%-20s%7s%9s%8s%n",                             // (3)
                      "Item", "Price", "Quantity", "Amount");

    int quantity = 4;
    double price = 120.25, amount = quantity*price, total = amount;
    String itemFormat = "%-20s%7.2f%9d%8.2f%n";                       // (4)
    System.out.printf(itemFormat,
                      "FlashDrive, 250GB", price, quantity, amount);  // (5)
    quantity = 2;
    price = 455.0; amount = quantity*price; total = total + amount;
    System.out.printf(itemFormat,
                      "Ultra HD, 4TB", price, quantity, amount);      // (6)
    quantity = 1;
    price = 8.50; amount = quantity*price; total = total + amount;
    System.out.printf(itemFormat,
                      "USB 3.0 cable", price, quantity, amount);      // (7)

    System.out.printf("%-36s%8.2f%n", "Total:", total);               // (8)
  }
}
```

Output from the program:

```
Secure Data Inc.       02/13/2016, 11:05

Item                  Price Quantity  Amount
FlashDrive, 250GB    120.25        4  481.00
Ultra HD, 4TB        455.00        2  910.00
USB 3.0 cable          8.50        1    8.50
Total:                               1399.50
```

1.12 The Java Ecosystem

Since its initial release as Java Development Kit 1.0 (JDK 1.0) in 1996, the name Java has become synonymous with a thriving ecosystem that provides the components and the tools necessary for developing systems for today's multicore world. Its diverse community, comprising a multitude of volunteers, organizations, and corporations, continues to fuel its evolution and grow with its success. Many free open-source technologies now exist that are well proven, mature, and supported, making their adoption less daunting. These tools and frameworks provide support for all phases of the software development life cycle and beyond.

There are three major Java Platforms for the Java programming language:

- Java SE (Standard Edition)
- Java EE (Enterprise Edition)
- Java ME (Micro Edition)

Each platform provides a hardware/operating system–specific JVM and an API (*application programming interface*) to develop applications for that platform. The Java SE platform provides the core functionality of the language. The Java EE platform is a superset of the Java SE platform and, as the most extensive of the three platforms, targets enterprise application development. The Java ME platform is a subset of the Java SE platform, having the smallest footprint, and is suitable for developing mobile and embedded applications. The upshot of this classification is that a Java program developed for one Java platform will not necessary run under the JVM of another Java platform. The JVM must be compatible with the Java platform that was used to develop the program.

The API and the tools for developing and running Java applications are bundled together as JDK. Just the JVM and the runtime libraries are also bundled separately as JRE (Java Runtime Environment).

The subject of this book is Java SE 8. We recommend installing the appropriate JDK for Java SE 8 (or a newer version) depending on the hardware and operating system.

The rest of this section summarizes some of the factors that have contributed to the evolution of Java from an object-oriented programming language to a full-fledged ecosystem for developing all sorts of systems, including large-scale business systems and embedded systems for portable computing devices. A lot of jargon is used in this section, and might be difficult to understand at the first reading, but we recommend coming back after working through the book to appreciate the factors that have contributed to the success of Java.

Object-Oriented Paradigm

The Java programming language supports the object-oriented paradigm, in which the properties of an object and its behavior are encapsulated in the object. The properties and the behavior are represented by the fields and the methods of the object, respectively. The objects communicate through method calls in a procedural manner. Encapsulation ensures that objects are immune to tampering except when manipulated through their public interface. Encapsulation exposes only *what* an object does and not *how* it does it, so that its implementation can be changed with minimum impact on its clients. Some basic concepts of object-oriented programming, such as inheritance and aggregation, were introduced earlier in this chapter, and subsequent chapters will expand on this topic.

Above all, object-oriented system development promotes code reuse where existing objects can be reused to implement new objects. It also facilitates implementation of large systems, allowing their decomposition into manageable subsystems.

Interpreted: The JVM

Java programs are compiled to bytecode that is interpreted by the JVM. Various optimization technologies (e.g., just-in-time [JIT] delivery) have led to the JVM becoming a lean and mean virtual machine with regard to performance, stability, and security. Many other languages, such as Scala, Groovy, and Clojure, now compile to bytecode and seamlessly execute on the JVM. The JVM has thus evolved into an ecosystem in its own right.

Architecture-Neutral and Portable Bytecode

The often-cited slogan “Write once, run everywhere” is true only if a compatible JVM is available for the hardware and software platform. In other words, to run Java SE applications under Windows 10 on a 64-bit hardware architecture, the right JVM must be installed. Fortunately, the JVM has been ported to run under most platforms and operative systems that exist today, including hardware devices such as smart cards, mobile devices, and home appliances.

The specification of the bytecode is architecture neutral, meaning it is independent of any hardware architecture. It is executed by a readily available hardware and operating system–specific JVM. The portability of the Java bytecode thus eases the burden of cross-platform system development.

Simplicity

Language design of Java has been driven by a desire to simplify the programming process. Although Java borrows heavily from the C++ programming language, certain features that were deemed problematic were not incorporated into its design. For example, Java does not have a preprocessor, and it does not allow pointer handling, user-defined operator overloading, or multiple class inheritance.

Java opted for automatic garbage collection, which frees the programmer from dealing with many issues related to memory management, such as memory leaks.

However, the jury is still out on whether the syntax of nested classes or introduction of wild cards for generics can be considered simple.

Dynamic and Distributed

The JVM can dynamically load class libraries from the local file system as well as from machines on the network, when those libraries are needed at runtime. This feature facilitates linking the code as and when necessary during the execution of a program. It is also possible to query programmatically a class or an object at runtime about its meta-information, such as its methods and fields.

Java provides extensive support for networking to build distributed systems, where objects are able to communicate across networks using various communication protocols and technologies, such as Remote Method Invocation (RMI) and socket connections.

Robust and Secure

Java promotes the development of reliable, robust, and secure systems. It is a strong statically typed language: The compiler guarantees runtime execution if the code compiles without errors. Elimination of pointers, runtime index checks for arrays and strings, and automatic garbage collection are some of the features of Java that promote reliability. The exception handling feature of Java is without doubt the main factor that facilitates the development of robust systems.

Java provides multilevel protection from malicious code. The language does not allow direct access to memory. A bytecode verifier determines whether any untrusted code loaded in the JVM is safe. The sandbox model is used to confine and execute any untrusted code, limiting the damage that such code can cause. These features, among others, are provided by a comprehensive Java security model to ensure that application code executes securely in the JVM.

High Performance and Multithreaded

The performance of Java programs has improved significantly with various optimizations that are applied to the bytecode at runtime by the JVM. The JIT feature monitors the program at runtime to identify performance-critical bytecode (called *hotspots*) that can be optimized. Such code is usually translated to machine code to boost performance. The performance achieved by the JVM is a balance between native code execution and interpretation of fully scripted languages, which fortunately is adequate for many applications.

Java has always provided high-level support for multithreading, allowing multiple threads of execution to perform different tasks concurrently in an application. It has risen to the new challenges that have emerged in recent years to harness the increased computing power made available by multicore architectures. Functional programming, in which computation is treated as side-effects–free evaluation of functions, is seen as a boon to meet these challenges. Java 8 brings elements of functional-style programming into the language, providing language constructs (lambda expressions and functional interfaces) and API support (through its Fork & Join Framework and Stream API) to efficiently utilize the many cores to process large amounts of data in parallel.

Review Questions

1.9 Which command from the JDK should be used to compile the following source code contained in a file named `SmallProg.java`?

```
public class SmallProg {
  public static void main(String[] args) { System.out.println("Good luck!"); }
}
```

Select the one correct answer.

(a) `java SmallProg`
(b) `javac SmallProg`
(c) `java SmallProg.java`
(d) `javac SmallProg.java`
(e) `java SmallProg main`

1.10 Which command from the JDK should be used to execute the `main()` method of a class named `SmallProg`?

Select the one correct answer.

(a) `java SmallProg`
(b) `javac SmallProg`
(c) `java SmallProg.java`
(d) `java SmallProg.class`
(e) `java SmallProg.main()`

1.11 Which statement is true about Java?

Select the one correct answer.

(a) A Java program can be executed by any JVM.
(b) Java bytecode cannot be translated to machine code.
(c) Only Java programs can be executed by a JVM.
(d) A Java program can create and destroy objects.
(e) None of the above

Chapter Summary

The following topics were covered in this chapter:

- Essential elements of a Java application
- Accessing object fields and calling methods
- Compiling and running Java applications
- Formatting and printing values to the terminal window

- Basic terminology and concepts in OOP, and how these concepts are supported in Java
- Factors and features of the Java ecosystem that have contributed to its evolution and success

Programming Exercise

1.1 Modify the `Client` class from Example 1.4 to use the `PrintableCharStack` class, rather than the `CharStack` class from Example 1.2. Utilize the `printStackElements()` method from the `PrintableCharStack` class. Is the new program behavior-wise any different from Example 1.4?