Basics of Java Programming 1

Chapter Topics

- Factors and features of the Java ecosystem that have contributed to its evolution and success
- Basic terminology and concepts in Java, and how the language supports object-oriented programming (OOP)
- Understand the distinction between a class and an object
- Basics of how to create objects, access their fields and call methods on them
- Essential elements of a Java program
- Compiling and running Java programs
- Executing single-file source-code programs
- Brief introduction to the Java Shell tool (`jshell`)
- Formatting and printing values to the terminal window

Java SE 17 Developer Exam Objectives	
[3.2] Create classes and records, and define and use instance and static fields and methods, constructors, and instance and static initializers ❍ *Basic terminology for declaring and using classes and class members is introduced in this chapter.* ❍ *For details on classes, fields, methods, and constructors, see Chapter 3, p. 97.* ❍ *For record classes, see §5.14, p. 305.* ❍ *For instance and static initializers, see §10.5, p. 554.*	*§1.2, p. 7* *§1.3, p. 9* *§1.4, p. 10* *§1.5, p. 11*

Java SE 11 Developer Exam Objectives	
[3.2] Define and use fields and methods, including instance, static and overloaded methods ❍ *Basic terminology for declaring and using classes and class members is introduced in this chapter.* ❍ *For details on classes, fields, methods, and constructors, see Chapter 3, p. 97.*	*§1.2, p. 7* *§1.3, p. 9* *§1.4, p. 10* *§1.5, p. 11*

Before embarking on the road to Java certification, it is important to understand the basic terminology and concepts in Java. No particular exam objective is covered in this chapter. The emphasis is on providing an introduction to Java and core concepts in object-oriented programming (OOP). In-depth coverage of the concepts introduced will follow in due course in subsequent chapters.

The basic elements of a Java program are introduced in this chapter. The old adage that practice makes perfect is certainly true when learning a programming language. We highly encourage programming on the computer. The mechanics of compiling and running a Java program are provided in this chapter. We begin with an overview of factors that make Java the platform of choice for enterprises and developers.

1.1 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 and 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 lifecycle and beyond.

There are different Java platforms, each targeting different application domains:

- Java SE (Standard Edition): designed for developing desktop and server environments
- Java EE, also known as Jakarta EE (Enterprise Edition): designed for developing enterprise applications
- Java ME (Micro Edition): designed for embedded systems, such as mobile devices and set-top boxes
- Java Card: designed for tiny memory footprint devices, such as smart cards

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 a small footprint, and is suitable for developing mobile and embedded applications. The Java Card platform allows development of embedded applications that have a very tiny memory footprint, targeting devices like smart cards. The upshot of this classification is that a Java program developed for one Java platform will not necessarily run under the JVM of another Java platform. The JVM must be compatible with the Java platform that was used to develop the application.

The API and the tools for developing and running Java applications are bundled together as the JDK. Starting with Java 11, JRE (Java Runtime Environment) is no longer available as a stand-alone bundle providing runtime support for execution of Java programs, but it continues to be a subset of the now modular JDK. As before, one needs to install the JDK to both develop and run Java programs. However, to deploy Java programs, the JDK tool `jlink` can be used to create a runtime image that includes the program code and the necessary runtime support to run the program—a topic that we will get to when we discuss modules.

We highly recommend installing the JDK for Java SE 17 depending on the hardware and operating system. Although newer versions of Java are released periodically, Java SE 17 is readily available as an LTS (*long-term support*) release, and is the subject of this book.

As of Java SE 17, Oracle is making the Oracle JDK available for free under the *Oracle No-Fee Terms and Conditions* (NFTC) license. Although subject to the conditions, it permits free use for *all* users.

Key Features of Java

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 it might be difficult to understand at the first reading, so we recommend coming back after working through the book to appreciate the factors that have contributed to the success of Java.

Multi-paradigm Programming

The Java programming language supports the *object-oriented programming 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—in other words, Java also incorporates the *procedural programming paradigm.* 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 minimal impact on its clients. The later sections in this chapter provide an overview of basic concepts of object-oriented programming, such as inheritance and aggregation, and subsequent chapters will expand on this topic.

Java has also evolved to support the *functional-style programming paradigm* with the introduction of lambda expressions and their implementation using functional interfaces. This topic will be thoroughly explored in this book.

Above all, object-oriented system development promotes code reuse where existing classes can be reused to implement new classes. Its module facility facilitates implementation of large systems, allowing their decomposition into manageable subsystems, as we will see when we discuss modules.

Bytecode Interpreted by 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 operating 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

The 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.

The introduction of functional-style features has enhanced Java's appeal, and the potential of its module system is yet to be seen.

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 programmatically query 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. Runtime index checks for arrays and strings, automatic garbage collection, and elimination of pointers are some of the features of Java that promote reliability. The exception handling feature of Java is without a doubt the main factor that facilitates the development of robust systems. And the module system further enhances encapsulation and configuration.

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 brings elements of functional-style programming into the language, providing language constructs (lambda expressions and functional interfaces) and API support (through its Concurrent and Stream APIs) to efficiently utilize the many cores to process large amounts of data in parallel.

1.2 Classes

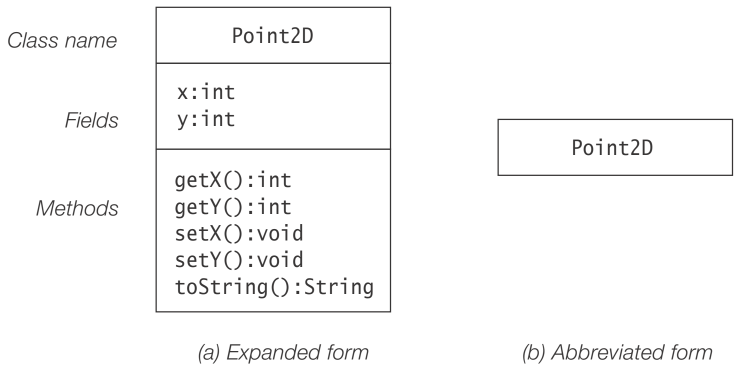
One of the fundamental ways in which we handle complexity is by using *abstractions*. An abstraction denotes the essential properties and behaviors of an object that differentiate it from other objects. The essence of OOP is modeling abstractions, using classes and objects. The hardest part of this endeavor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

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 of 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 only know the contract of an object, and not its implementation, to avail themselves of the object's services.

As an example, we will implement a class that models the abstraction of a point as (x, y)-coordinates in a two-dimensional plane. The class `Point2D` will use two `int` fields `x` and `y` to store the coordinates. Using simplified Unified Modeling Language (UML) notation, the class `Point2D` is graphically depicted in Figure 1.1, which models the abstraction. Both fields, with their type and method names and their return value type, are shown in Figure 1.1a.

Figure 1.1 *UML Notation for Classes*



Declaring Members: Fields and Methods

Example 1.1 shows the declaration of the class `Point2D` depicted in Figure 1.1. Its intention is to illustrate the salient features of a class declaration in Java, rather than an industrial-strength implementation. We will come back to the nitty-gritty of the Java syntax in subsequent chapters.

In Example 1.1, 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 can contain member declarations that define the fields and the methods of the objects the class represents. In the case of the class `Point2D`, it has the following two fields declared at (1):

- `x`, which is the x-coordinate of a point
- `y`, which is the y-coordinate of a point

The class `Point2D` has five methods, declared at (3), that implement the essential operations provided by a point:

- `getX()` returns the x-coordinate of the point.
- `getY()` returns the y-coordinate of the point.
- `setX()` sets the x-coordinate to the value passed to the method.
- `setY()` sets the y-coordinate to the value passed to the method.
- `toString()` returns a string with the coordinate values formatted as `"(x,y)"`.

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: Point2D.java
public class Point2D {            // Class name
  // Class Member Declarations

  // Fields:                                                    (1)
  private int x;     // The x-coordinate
  private int y;     // The y-coordinate

  // Constructor:                                               (2)
  public Point2D(int xCoord, int yCoord) {
    x = xCoord;
    y = yCoord;
  }

  // Methods:                                                   (3)
  public int  getX()          { return x; }
  public int  getY()          { return y; }
```

```
    public void setX(int xCoord) { x = xCoord; }
    public void setY(int yCoord) { y = yCoord; }
    public String toString() { return "(" + x + "," + y + ")"; } // Format: (x,y)
  }
```

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 uniquely 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 only be manipulated by a reference that holds its reference value. Direct access to the reference value is not permitted.

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*:

```
Point2D p1 = new Point2D(10, 20);  // A point with coordinates (10,20)
```

In the preceding declaration statement, the left-hand side of the *assignment operator* (=) declares that p1 is a reference of class Point2D. The reference p1, therefore, can refer to objects of class Point2D.

The right-hand side of the assignment operator creates an object of class Point2D. This step involves using the new operator in conjunction with a call to a constructor of the class (new Point2D(10, 20)). The new operator creates an instance of the Point2D class and returns the reference value of this instance. The arguments passed in the constructor call are used to initialize the x and the y fields, respectively. The assignment operator stores the reference value in the reference p1 declared on the left-hand side of the assignment operator. The reference p1 can now be used to manipulate the object whose reference value it holds.

Analogously, the following declaration statement declares the reference p2 to be of class Point2D, creates an object of class Point2D, and assigns its reference value to the reference p2:

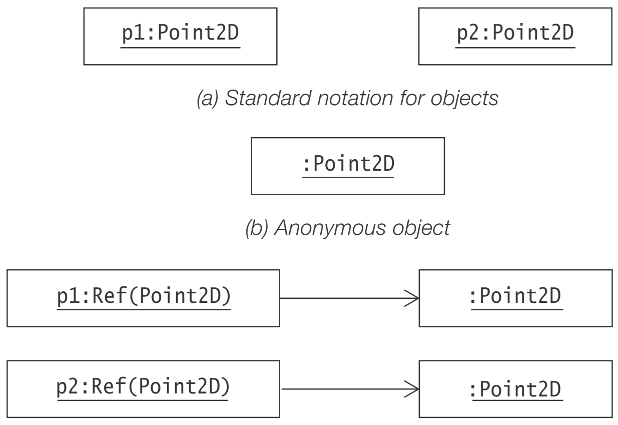
```
Point2D p2 = new Point2D(5, 15);  // A point with coordinates (5,15)
```

Each object that is created has its own copy of the fields declared in the class declaration. That is, the two point objects, referenced by p1 and p2, will have their own x and y fields. The fields of an object are also 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 purpose of the constructor call on the right-hand side of the new operator is to initialize the fields of the newly created object. In this particular case, for each new Point2D object created using the new operator, the constructor at (2) in Example 1.1 creates the x and y fields and initializes them with the arguments passed.

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 Point2D class explicitly refer to Point2D objects. In most cases, the more compact notation will suffice.

Figure 1.2 *UML Notation for Objects*



(c) Explicit references for Java objects

1.4 Instance Members

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 to the instance variables, 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* (p. 11), which only belong to the class.

Invoking Methods

Objects communicate by calling methods on each other. As a consequence, an object can be made to exhibit a particular behavior by calling the appropriate method on the object. This is achieved by a *method call* whose basic form is the following: a reference that refers to the object, the binary dot (.) operator, and the name of method to be invoked, together with a list of any arguments required by the method.

reference.*methodName*(*listOfArguments*)

The method invoked on the object can also return results back to its caller, via a single return value. The method called must be one that is defined for the object; otherwise, the compiler reports an error.

```
Point2D point = new Point2D(-1, -4);   // Creates a point with coordinates (-1,-4)
point.setX(-2);                        // (1) The x field is set to the value -2
int yCoord = point.getY();             // (2) Returns the value -4 of the y field
System.out.println(point.toString());  // (3) Prints: (-2,-4)
point.distanceFromOrigin();            // (4) Compile-time error: No such method.
```

The sample code above invokes methods on the object denoted by the reference `point`. The method call at (1) sets the value of the `x` field of `point`, and the method call at (2) returns the value of the `y` field of `point`. At (3), the call to the `toString()` method returns the string `"(-2,-4)"` which is printed. The `setX()`, `getY()`, and `toString()` methods are all defined in the class `Point2D`. The `setX()` method does not return any value, but the `getY()` and `toString()` methods do. Trying to invoke a method named `distanceFromOrigin()` at (4) on `point` results in a compile-time error, as no such method is defined in the class `Point2D`.

The dot (.) notation can also be used with a reference to access the fields of an object. The basic form for field access is as follows:

reference.*fieldName*

Use of the dot notation is governed by the *accessibility* of the member. The methods of the `Point2D` class are `public` and can thus be called by the clients of the class. However, the fields in the class `Point2D` have `private` access, indicating that they are not accessible from outside the class. Thus the code below at (1) in a client of the `Point2D` class will not compile. Typically, a class provides `public` methods to access values in its `private` fields, as class `Point2D` does.

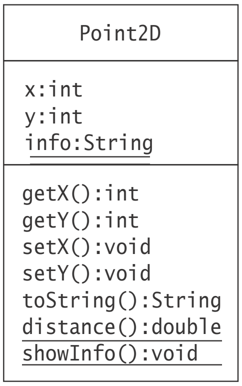
```
System.out.println(point.x);        // (1) Compile-time error: x is not accessible.
System.out.println(point.getX());  // OK.
```

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. Such members are called *static members*. Fields and methods that are static members are easily distinguishable in a class declaration as they must always be declared with the keyword `static`.

Figure 1.3 shows the class diagram for the class `Point2D`. It has been augmented by three static members, whose names are underlined to distinguish them from instance members. The augmented declaration of the `Point2D` class is given in Example 1.2.

Figure 1.3 *Class Diagram Showing Static Members of a Class*



In Example 1.2, the field `info` at (1) is declared as a *static variable*. This field has information about the purpose of the class that the class can share with its clients. A static variable belongs to the class, rather than to any specific object of the class. It will be allocated in the class and initialized to the string specified in its declaration when the class is loaded. Declaring the `info` field as `static` makes sense, as it is unnecessary that every object of the class `Point2D` should have a copy of this information.

```
private static String info = "A point represented by (x,y)-coordinates.";
```

In Example 1.2, the two methods `distance()` and `showInfo()` at (5) are *static methods* belonging to the class. Both are declared with the keyword `static`. The static method `distance()` calculates and returns the distance between two points passed as arguments to the method. The static method `showInfo()` prints the string with the information referenced by the static variable `info`. These methods belong to the class, rather than to any specific objects of the class.

Clients can access static members in the class by using the class name. The following code invokes the static method `distance()` in the class `Point2D`:

```
double d = Point2D.distance(p1, p2); // Class name to invoke static method
```

Static members can also be accessed via object references, although doing so is not encouraged:

```
p1.showInfo();                        // 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.

Example 1.2 *Static Members in Class Declaration*

```
// File: Point2D.java
public class Point2D {             // Class name
  // Class Member Declarations

  // Static variable:                                              (1)
  private static String info = "A 2D point represented by (x,y)-coordinates.";

  // Instance variables:                                           (2)
  private int x;
  private int y;

  // Constructor:                                                  (3)
  public Point2D(int xCoord, int yCoord) {
    x = xCoord;
    y = yCoord;
  }

  // Instance methods:                                             (4)
  public int  getX()          { return x; }
  public int  getY()          { return y; }
  public void setX(int xCoord) { x = xCoord; }
  public void setY(int yCoord) { y = yCoord; }
  public String toString() { return "(" + x + "," + y + ")"; } // Format: (x,y)

  // Static methods:                                               (5)
  public static double distance(Point2D p1, Point2D p2) {
    int xDiff = p1.x - p2.x;
    int yDiff = p1.y - p2.y;
    return Math.sqrt(xDiff*xDiff + yDiff*yDiff);
  }
  public static void showInfo() { System.out.println(info); }
}
```

Figure 1.4 shows the classification of the members in the class `Point2D`, using the terminology we have introduced so far. Table 1.1 provides a summary of the terminology used in defining members of a class.

Figure 1.4 *Members of a Class*

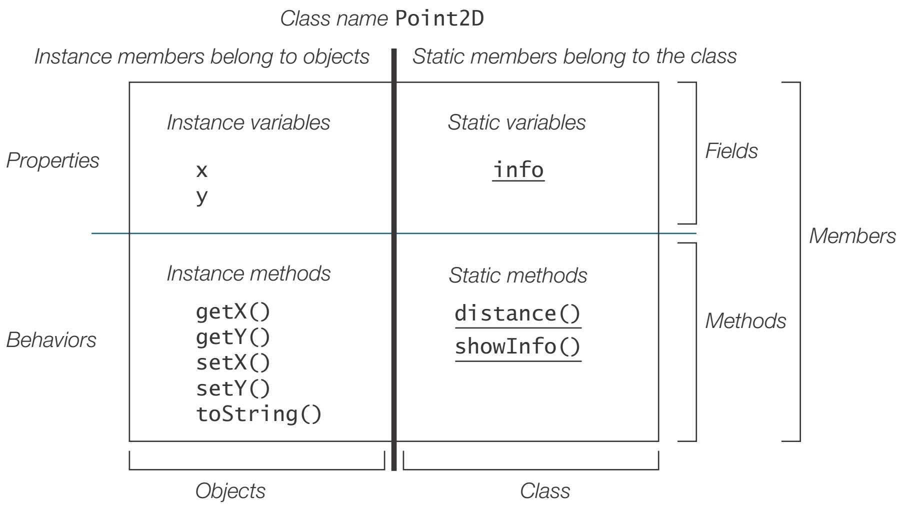


Table 1.1 *Terminology for Class Members*

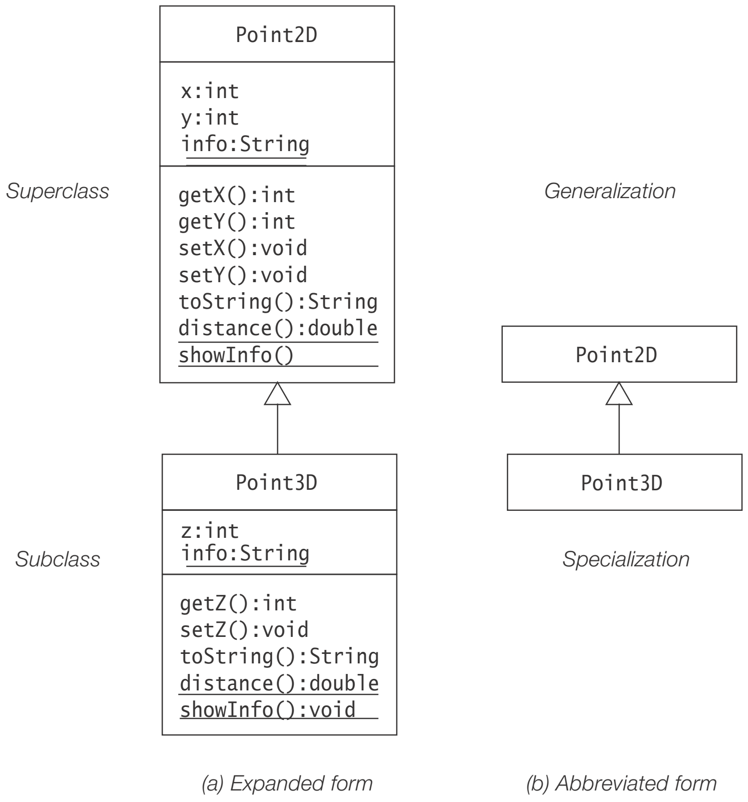
Instance members	The instance variables and instance methods of an object. They can be accessed or invoked only through an object reference.
Instance variable	A field that is allocated when the class is instantiated (i.e., when an object of the class is created). Also called a *non-static field* or just 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.
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* (p. 17). 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 point in three-dimensional space—that is, a 3D point represented by (x, y, z)-coordinates. We can derive the 3D point from the `Point2D` class. This 3D point will have all the properties and behaviors of the `Point2D` class, along with the additional third dimension. This relationship is shown in Figure 1.5 and implemented in Example 1.3. The class `Point3D` is called the *subclass*, and the class `Point2D` is called the *superclass*. The `Point2D` class is a *generalization* for points, whereas the class `Point3D` is a *specialization* of points that have three coordinates.

Figure 1.5 *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 Point3D extends the Point2D class, shown at (1).

```
public class Point3D extends Point2D {             // (1) Uses extends clause
  // ...
}
```

The Point3D class only declares the z-coordinate at (3), as every object of the subclass will have the x and y fields that are specified in its superclass Point2D. Note that these fields are declared private in the superclass Point2D, but they are accessible to a Point3D object indirectly through the public get and set methods for the x- and the y-coordinates in the superclass Point2D. These methods are *inherited* by the Point3D class.

The constructor of the Point3D class at (4) takes three arguments corresponding to the x-, y-, and z-coordinates. The call to super() at (5) results in the constructor of the superclass Point2D being called to initialize the x- and y-coordinates.

It addition, the Point3D class declares methods at (6) to get and set the z-coordinate. It provides its own version of the toString() method to format a point that has three coordinates.

Since calculating the distance is also different in three-dimensional space from that in a two-dimensional plane, the Point3D class provides its own distance() static method at (7). As its objects represent 3D points, it declares its own static field info and provides its own static method showInfo() to print this information.

Example 1.3 *Defining a Subclass*

```
// File: Point2D.java
public class Point2D {
  // Same as in Example 1.2.
}
```

```
// File: Point3D.java
public class Point3D extends Point2D {            // (1) Uses extends clause

  // Static variable:                                               (2)
  private static String info = "A 3D point represented by (x,y,z)-coordinates.";

  // Instance variable:                                             (3)
  private int z;

  // Constructor:                                                   (4)
  public Point3D(int xCoord, int yCoord, int zCoord) {
    super(xCoord, yCoord);                                       // (5)
    z = zCoord;
  }

  // Instance methods:                                              (6)
  public int  getZ()          { return z; }
```

```
    public void setZ(int zCoord) { z = zCoord; }
    @Override
    public String toString() {
      return "(" + getX() + "," + getY() + "," + z + ")"; // Format: (x,y,z)
    }

    // Static methods:                                                    (7)
    public static double distance(Point3D p1, Point3D p2) {
      int xDiff = p1.getX() - p2.getX();
      int yDiff = p1.getY() - p2.getY();
      int zDiff = p1.getZ() - p2.getZ();
      return Math.sqrt(xDiff*xDiff + yDiff*yDiff + zDiff*zDiff);
    }
    public static void showInfo() { System.out.println(info); }
  }
```

Objects of the `Point3D` class will respond just like objects of the `Point2D` class, but they also have the additional functionality defined in the subclass. References of the class `Point3D` are used in the code below. The comments indicate in which class a method is invoked. Note that the subclass reference can invoke the inherited get and set methods in the superclass.

```
Point3D p3A = new Point3D(10, 20, 30);
System.out.println(p3A.toString());             // (10,20,30)       (Point3D)
System.out.println("x: " + p3A.getX());         // x: 10            (Point2D)
System.out.println("y: " + p3A.getY());         // y: 20            (Point2D)
System.out.println("z: " + p3A.getZ());         // z: 30            (Point3D)
p3A.setX(-10); p3A.setY(-20); p3A.setZ(-30);
System.out.println(p3A.toString());             // (-10,-20,-30)    (Point3D)

Point3D p3B = new Point3D(30, 20, 10);
System.out.println(p3B.toString());             // (30,20,10)       (Point3D)
System.out.println(Point3D.distance(p3A, p3B));   // 69.2820323027551 (Point3D)
Point3D.showInfo(); // A 3D point represented by (x,y,z)-coordinates. (Point3D)
```

1.7 Aggregation

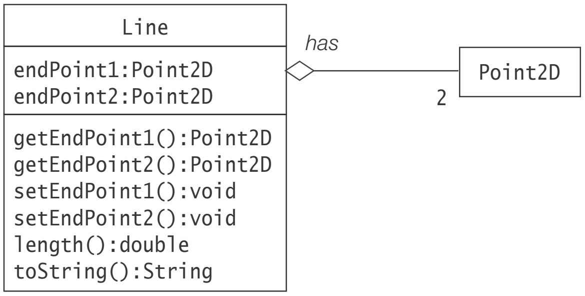
An *association* defines a static relationship between objects of two classes. One such association, called *aggregation* (also known as *composition*), expresses how an object uses other objects. Java supports aggregation of objects by reference, since objects cannot contain other objects explicitly. The aggregate object usually has fields that denote its constituent objects. By default, Java uses aggregation when fields denoting objects are declared in a class declaration. Typically, an aggregate object delegates its tasks to its constituent objects.

We illustrate aggregation by implementing a finite-length straight line that has two end points in a two-dimensional plane. We would like to use the class `Point2D` to implement such a line. A class `Line` could be implemented by having fields for two `Point2D` objects that would represent the end points of a line. This aggregate relationship is depicted in Figure 1.6, which shows that a `Line` object has two `Point2D` objects, indicated by the diamond notation. The complete declaration of the `Line` class is shown in Example 1.4. The two fields `endPoint1` and `endPoint2` declared at (1) represent the two end points. In particular, note the `length()` method at (2) which delegates the computation of the length to the `Point2D.distance()` method.

The following code shows how a `Line` object can be manipulated:

```
Line line1 = new Line(new Point2D(5,6), new Point2D(7,8));
System.out.println(line1.toString());              // Line[(5,6),(7,8)]
line1.setEndPoint1(new Point2D(11, 12));
line1.setEndPoint2(new Point2D(13, 14));
System.out.println(line1.toString());              // Line[(11,12),(13,14)]
System.out.println("Length: " + line1.length());    // Length: 2.8284271247461903
```

Figure 1.6 *Class Diagram Depicting Aggregation*



Example 1.4 *Using Aggregation*

```
// File: Point2D.java
public class Point2D {
  // Same as in Example 1.2.
}
```

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

  // Instance variables:                                                     (1)
  private Point2D endPoint1;
  private Point2D endPoint2;

  // Constructor:
  public Line(Point2D p1, Point2D p2) {
    endPoint1 = p1;
    endPoint2 = p2;
  }

  // Methods:
  public Point2D getEndPoint1() { return endPoint1; }
  public Point2D getEndPoint2() { return endPoint2; }
```

```
    public void setEndPoint1(Point2D p1) { endPoint1 = p1; }
    public void setEndPoint2(Point2D p2) { endPoint2 = p2; }
    public double length() {                                          // (2)
      return Point2D.distance(endPoint1, endPoint2);
    }
    public String toString()  {
      return "Line[" + endPoint1 + "," + endPoint2 + "]";
    }
  }
```

Review Questions

1.1 Which statement is true about methods?
Select the one correct answer.

(a) A method is an attribute defining a particular property of an abstraction.
(b) A method is a category of objects.
(c) A method is an operation defining a particular behavior of an abstraction.
(d) A method is a blueprint for defining 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 are created and how many references are declared 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 declared.
(e) Two references are declared.
(f) Three references are declared.

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 is always a part of an instance.
(e) An instance member always represents an operation.

1.6 How do objects communicate with each other 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 of the following 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 Sample Java Program

The term *program* refers to source code that is compiled and directly executed. The terms *program* and *application* are often used synonymously, and are so used in this book. To create a program in Java, the program must have a class that defines a method named `main`, which is invoked at runtime to start the execution of the program. The class with this `main()` method is known as the *entry point of the program*.

Essential Elements of a Java Program

Example 1.5 comprises three classes: `Point2D`, `Point3D`, and `TestPoint3D`. The public class `TestPoint3D` in the file `TestPoint3D.java` is the entry point of the program. It defines a method with the name `main`. 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` access—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.

Example 1.5 *A Sample Program*

```
// File: Point2D.java
public class Point2D {
  // Same as in Example 1.2.
}
```

```
// File: Point3D.java
public class Point3D extends Point2D {
  // Same as in Example 1.3.
}
```

```
// File: TestPoint3D.java
public class TestPoint3D {
  public static void main(String[] args) {
    Point3D p3A = new Point3D(10, 20, 30);
    System.out.println("p3A: " + p3A.toString());
    System.out.println("x: " + p3A.getX());
    System.out.println("y: " + p3A.getY());
    System.out.println("z: " + p3A.getZ());
    p3A.setX(-10); p3A.setY(-20); p3A.setZ(-30);
    System.out.println("p3A: " + p3A.toString());

    Point3D p3B = new Point3D(30, 20, 10);
    System.out.println("p3B: " + p3B.toString());
    System.out.println("Distance between p3A and p3B: " +
                       Point3D.distance(p3A, p3B));
    Point3D.showInfo();
  }
}
```

Output from the program:

```
p3A: (10,20,30)
x: 10
y: 20
z: 30
p3A: (-10,-20,-30)
p3B: (30,20,10)
Distance between p3A and p3B: 69.2820323027551
A 3D point represented by (x,y,z)-coordinates.
```

Compiling a Program

The JDK provides tools for compiling and running programs. The classes in the Java SE Platform API are already compiled, and the JDK tools know where to find them.

Java source files can be compiled using the *Java Language Compiler*, `javac`, which is part of the JDK. Each source file name has the extension `.java`. Each class declaration in a source file is compiled into a separate *class file*, containing its *Java bytecode*. The name of this file comprises the name of the class with `.class` as its extension.

The source files `Point2D.java`, `Point3D.java`, and `TestPoint3D.java` contain the declarations of the `Point2D`, `Point3D`, and `TestPoint3D` classes, respectively. The respective source files are in the same directory. The source files can be compiled by giving the following `javac` command on the command line (the character > is the command prompt and we will use bold type for anything typed on the command line):

```
>javac Point2D.java Point3D.java TestPoint3D.java
```

This `javac` command creates the class files `Point2D.class`, `Point3D.class`, and `TestPoint3D.class` containing the Java bytecode for the `Point2D`, `Point3D`, and `TestPoint3D` classes, respectively. The command creates the class files in the same directory as the source files.

Although a Java source file can contain more than one class declaration, the Java compiler enforces the rule that there can only be *at the most* one class in the source file that has `public` access. If there is a public class in the source file, the name of the source file must be the name of the public class with `.java` as its extension. In the absence of a public class in the source file, the name of the file can be arbitrary, but still with the `.java` extension. Regardless, each class declaration in a source file is compiled into a separate `.class` file.

Running a Program

It is the bytecode in the class files that is executed when a Java program is run—the source code is immaterial in this regard. A Java program is run by the *Java Application Launcher*, `java`, which is also part of the JDK. The `java` command creates an instance of the JVM that executes the bytecode.

The following `java` command on the command line will launch the program in Example 1.5:

```
>java TestPoint3D
p3A: (10,20,30)
...
```

Note that only the name of the class that is the entry point of the program is specified, resulting in the execution starting in the `main()` method of the specified class. This `main()` method is found in the *class* file of the `TestPoint3D` class. The program in Example 1.5 terminates when the execution of the `main()` method is completed.

Running a Single-File Source-Code Program

Typically, Java source code is first compiled by the `javac` command to Java bytecode in class files and then the bytecode in the class files is executed by the `java` command. The compilation step can be omitted if the complete source code of the program is contained in a *single* source file, meaning that all class declarations that comprise the program are declared in one source file.

In Example 1.5, the program is composed of three source files: `Point2D.java`, `Point3D.java`, and `TestPoint3D.java`, containing the declarations of the `Point2D`, `Point3D`, and `TestPoint3D` classes, respectively. In Example 1.6, the class declarations are now contained in the `Demo-App.java` file; in other words, the complete source code of the program is now in a single source file. We can run the program with the following `java` command, without compiling the source code first:

```
>java Demo-App.java
p3A: (10,20,30)
...
```

The full name of the single source file is specified in the command line. Full program output is shown in Example 1.6.

Note that no class files are created. The source code is compiled fully in memory and executed.

In order to run a single-file source-code program, the following conditions must be met:

- The single source file must contain *all* source code for the program.
- Although there can be several class declarations in the source file, the *first* class declaration in the source file must provide the `main()` method; that is, it must be the entry point of the program.

- There must not exist *class* files corresponding to the class declarations in the single source file that are accessible by the `java` command.

Unlike the `javac` command, the name of the single source file (e.g., `Demo-App.java`) need not be a valid class name, but it must have the `.java` extension. Also unlike the `javac` command, the `java` command allows several public classes in the single source file (only public classes in the `Demo-App.java` file).

Examples of single-file source-code programs can be found throughout the book.

Example 1.6 *A Single-File Source-Code Program*

```
// File: Demo-App.java
public class TestPoint3D {
  // Same as in Example 1.5.
  // Provides the main() method and is the first class declaration in the file.
}

public class Point2D {
  // Same as in Example 1.2.
}

public class Point3D extends Point2D {
  // Same as in Example 1.3.
}
```

Running the program:

```
>java Demo-App.java
p3A: (10,20,30)
x: 10
y: 20
z: 30
p3A: (-10,-20,-30)
p3B: (30,20,10)
Distance between p3A and p3B: 69.2820323027551
A 3D point represented by (x,y,z)-coordinates.
```

The Java Shell Tool (`jshell`)

This subsection is not on any Java Developer Exam. Its sole purpose is to introduce a JDK tool that is an excellent aid in learning Java programming.

The interactive command-line tool `jshell` is excellent when it comes to learning the Java programming language. It is a *Read-Evaluate-Print Loop* (REPL) tool, meaning that it continuously reads what is typed at the terminal, evaluates the input, and prints the results. It evaluates such language constructs as declarations, statements, and expressions as they are entered at the terminal, and shows the results immediately. It provides access to the Java SE Platform API. Pressing the TAB key results in auto-completion of the snippet, and if that fails, suggests possible options. It is an ideal tool for quickly testing code snippets. We also encourage the reader to consult the documentation for the `jshell` JDK tool.

The following is an example of a session with the jshell tool:

```
>jshell
|  Welcome to JShell -- Version 17.0.2
|  For an introduction type: /help intro

jshell> int i = 20
i ==> 20

jshell> Math.sqrt(i)
$6 ==> 4.47213595499958

jshell> 3 + 4 * 5
$7 ==> 23

jshell> (3 + 4) * 5
$8 ==> 35

jshell> /exit
|  Goodbye
>
```

1.9 Program Output

Data produced by a program is called *output*. This output can be sent to different devices. The examples presented in this book usually send their output to a terminal window, where the output is printed as a line of characters with a cursor that advances as the 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. This class provides methods for printing values. These methods convert values to their text representation and print the resulting string.

The print methods convert a primitive value to a string that represents its literal value, and then print the resulting string.

```
System.out.println(2022);                               // 2022
```

An object is first converted to its text representation by calling its toString() method implicitly, if it is not already called explicitly on the object. The print statements below will print the same text representation of the Point2D object denoted by the reference origin:

```
Point2D origin = new Point2D(0, 0);
System.out.println(origin.toString());                  // (0,0)
System.out.println(origin);                             // (0,0)
```

The toString() method called on a String object returns the String object itself. As string literals are String objects, the following statements will print the same result:

```
System.out.println("Stranger Strings".toString());      // Stranger Strings
System.out.println("Stranger Strings");                 // Stranger Strings
```

The `println()` method always terminates the current line, which results in the cursor being moved to the beginning of the next line. The `print()` method prints its argument to the terminal window, but it does not terminate the current line:

```
System.out.print("Don't terminate this line!");
```

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

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

Formatted Output

This subsection is not on any Java Developer Exam. It is solely included because many examples in this book format their output to aid in understanding the computed results.

For more control over how the values are printed, we can format the 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.

This method returns the `PrintStream` on which the method is invoked, and can be ignored, as in the examples here.

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 text 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.

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.
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 text representation of the object occupies as many character positions as needed.
	`"%6s"`	`"Hi!"`	`"   Hi!"`	The text representation of the object occupies six character positions and is right-justified.
	`"%-6s"`	`"Hi!"`	`"Hi!   "`	The text representation of the object occupies six character positions and is left-justified.

Review Questions

1.8 Which command from the JDK will create a class file with the bytecode of 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`

1.9 Which command from the JDK should be used to execute the `main()` method of a class named `SmallProg` that has been compiled?
Select the one correct answer.

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

1.10 Which of the following statements are true about a single-file source-code program?
Select the two correct answers.

(a) It can be composed of multiple class declarations in the source file, where the first class declaration must provide the `main()` method.
(b) It can access previously compiled user-defined classes.
(c) It can be composed of multiple source files.
(d) It can accept program arguments on the command line.

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