JAVA Programming 2014-2
Java Programming Part-II

1. Introduction to Object Oriented Programming

1.1 Introduction to Object Technology

- Objects, or more precisely, the classes objects come from, are essentially reusable software components.
  - There are date objects, time objects, audio objects, video objects, automobile objects, people objects, etc.
  - Almost any noun can be reasonably represented as a software object in terms of attributes (e.g., name, color and size) and behaviors (e.g., calculating, moving and communicating).
- Using a modular, object-oriented design and implementation approach can make software-development groups much more productive than was possible with earlier popular techniques like "structured programming"—object-oriented programs are often eas-ier to understand, correct and modify.
- The Automobile as an Object
  - Let's begin with a simple analogy.
  - Suppose you want to *drive a car and make it go faster by pressing its accelerator pedal.*
  - Before you can drive a car, someone has to *design it.*
  - A car typically begins as engineering drawings, similar to the *blueprints that describe the design of a house.*
  - Drawings include the design for an accelerator pedal.
  - Pedal *hides from the driver the complex mechanisms that actually make the car go faster, just as the brake pedal hides the mechanisms that slow the car, and the steering wheel "hides" the mechanisms that turn the car.*
  - Enables people with little or no knowledge of how engines, braking and steering mechanisms work to drive a car easily.
  - Before you can drive a car, it must be *built* from the engineering drawings that describe it.
  - A completed car has an *actual* accelerator pedal to make the car go faster, but even that's not enough—the car won't accelerate on its own (hopefully!), so the driver must press the pedal to accelerate the car.
- Methods and Classes
  - Performing a task in a program requires a method.
  - The method houses the program statements that actually perform its tasks.
  - Hides these statements from its user, just as the accelerator pedal of a car hides from the driver the mechanisms of making the car go faster.
  - In Java, we create a program unit called a class to house the set of methods that perform the class's tasks.
  - A class is similar in concept to a car's engineering drawings, which house the design of an accelerator pedal, steering wheel, and so on.
- Instantiation

- Just as someone has to build a car from its engineering drawings before you can actually drive a car, you must build an object of a class before a program can perform the tasks that the class's methods define.
- An object is then referred to as an instance of its class.

- Reuse
  - Just as a car's engineering drawings can be *reused* many times to build many cars, you can reuse a class many times to build many objects.
  - Reuse of existing classes when building new classes and programs saves time and effort.
  - Reuse also helps you build more reliable and effective systems, because existing classes and components often have gone through extensive *testing, debugging and performance tuning.*
  - Just as the notion of *interchangeable parts* was crucial to the Industrial Revolution, reusable classes are crucial to the software revolution that has been spurred by object technology.
- Messages and Methods Calls
  - When you drive a car, pressing its gas pedal sends a *message* to the car to perform a task—that is, to go faster.
  - Similarly, you send messages to an object.
  - Each message is implemented as a method call that tells a method of the object to perform its task.
- Attributes and Instance Variables
  - A car has *attributes*
  - Color, its number of doors, the amount of gas in its tank, its current speed and its record of total miles driven (i.e., its odometer reading).
  - The car's attributes are represented as part of its design in its engineering diagrams.
  - Every car maintains its *own attributes.*
  - Each car knows how much gas is in its own gas tank, but *not* how much is in the tanks of other cars.
  - An object, has attributes that it carries along as it's used in a program.
  - Specified as part of the object's class.
  - A bank account object has a *balance* attribute that represents the amount of money in the account.
  - Each bank account object knows the balance in the account it represents, but *not* the balances of the other accounts in the bank.
  - Attributes are specified by the class's instance variables.
- Encapsulation
  - Classes encapsulate (i.e., wrap) attributes and methods into objects—an object's attributes and methods are intimately related.
  - Objects may communicate with one another, but they're normally not allowed to know how other objects are implemented—implementation details are *hidden* within the objects themselves.
  - Information hiding, as we'll see, is crucial to good software engineering.
- Inheritance
  - A new class of objects can be created quickly and conveniently by inheritance—the new class absorbs the characteristics of an existing class, possibly customizing them and adding unique characteristics of its own.
  - In our car analogy, an object of class "convertible" certainly *is an* object of the more general class "automobile," but more specifically, the roof can be raised or lowered.
- Object-Oriented Analysis and Design (OOAD)

- How will you create the code (i.e., the program instructions) for your programs?
- Follow a detailed analysis process for determining your project's requirements (i.e., defining *what* the system is supposed to do)
- Develop a design that satisfies them (i.e., deciding *how* the system should do it).
- Carefully review the design (and have your design reviewed by other software professionals) before writing any code.
- Analyzing and designing your system from an object-oriented point of view is called an object-oriented analysis and design (OOAD) process.
- Languages like Java are object oriented.
- Object-oriented programming (OOP) allows you to implement an object-oriented design as a working system.

- The UML (Unified Modeling Language)
  - The Unified Modeling Language (UML) is the most widely used graphical scheme for modeling object-oriented systems.

Diagram	Description
Activity Diagram	Depicts high-level business processes, including data flow, or to model the logic of complex logic within a system.
Class Diagram	Shows a collection of static model elements such as classes and types, their contents, and their relationships.
Communication Diagram	Shows instances of classes, their interrelationships, and the message flow between them. Communication diagrams typically focus on the structural organization of objects that send and receive messages. Formerly called a Collaboration Diagram
Component Diagram	Depicts the components that compose an application, system, or enterprise. The components, their interrelationships, interactions, and their public interfaces are depicted.
Composite Structure Diagram	Depicts the internal structure of a classifier (such as a class, component, or use case), including the interaction points of the classifier to other parts of the system.
Deployment Diagram	Shows the execution architecture of systems. This includes nodes, either hardware or software execution environments, as well as the middleware connecting them.
Interaction Overview Diagram	A variant of an activity diagram which overviews the control flow within a system or business process. Each node/activity within the diagram can represent another interaction diagram.
Object Diagram	Depicts objects and their relationships at a point in time, typically a special case of either a class diagram or a communication diagram.
Package Diagram	Shows how model elements are organized into packages as well as the dependencies between packages.
Sequence Diagram	Models the sequential logic, in effect the time ordering of messages between classifiers.
State Machine Diagram	Describes the states an object or interaction may be in, as well as the transitions between states. Formerly referred to as a state diagram, state chart diagram, or a state-transition diagram.
Timing Diagram	Depicts the change in state or condition of a classifier instance or role over time. Typically used to show the change in state of an object over time in response to external events.
Use Case Diagram	Shows use cases, actors, and their interrelationships.

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

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

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

Fig. 3.1 | Class declaration with one method.

- Create a new class (GradeBook)
- Use it to create an object.
- Each class declaration that begins with keyword public must be stored in a file that has the same name as the class and ends with the .java file-name extension.
- Keyword public is an access modifier Indicates that the class is “available to the public”
- The main method is called automatically by the Java Virtual Machine (JVM) when you execute an application.
- Normally, you must call methods explicitly to tell them to perform their tasks.
- A public is “available to the public”
- It can be called from methods of other classes.
- The return type specifies the type of data the method returns after performing its task.
- Return type void indicates that a method will perform a task but will *not* return (i.e., give back) any information to its calling method when it completes its task.
- Method name follows the return type.
- By convention, method names begin with a lowercase first letter and subsequent words in the name begin with a capital letter.
- Empty parentheses after the method name indicate that the method does not require additional information to perform its task.
- Together, everything in the first line of the method is typically called the Method header
- Every method’s body is delimited by left and right braces.
- The method body contains one or more statements that perform the method’s task.
- Use class GradeBook in an application.
- Class GradeBook is not an application because it does not contain main.
- Can’t execute GradeBook; will receive an error message like:
- Exception in thread "main" java.lang.NoSuchMethodError: main
- Must either declare a separate class that contains a main method or place a main method in class GradeBook.
- To help you prepare for the larger programs, use a separate class containing method main to test each new class.
- Some programmers refer to such a class as a driver class.

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

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

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

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

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

- A static method (such as main) is special
- It can be called without first creating an object of the class in which the method is declared.
- Typically, you cannot call a method that belongs to another class until you create an object of that class.
- Declare a variable of the class type.
- Each new class you create becomes a new type that can be used to declare variables and create objects.
- You can declare new class types as needed; this is one reason why Java is known as an extensible language.
- Class instance creation expression
  - Keyword new creates a new object of the class specified to the right of the keyword.
  - Used to initialize a variable of a class type.
  - The parentheses to the right of the class name are required.
  - Parentheses in combination with a class name represent a call to a constructor, which is similar to a method but is used only at the time an object is created to initialize the object's data.
- Call a method via the class-type variable
  - Variable name followed by a dot separator (.), the method name and parentheses.
  - Call causes the method to perform its task.
  - Any class can contain a main method.
  - The JVM invokes the main method only in the class used to execute the application.
  - If multiple classes that contain main, then one that is invoked is the one in the class named in the java command.
- Figure 3.3: UML class diagram for class GradeBook.
- Each class is modeled in a class diagram as a rectangle with three compartments.
  - Top: contains the class name centered horizontally in boldface type.
  - Middle: contains the class's attributes, which correspond to instance variables (Section 3.5).
  - Bottom: contains the class's operations, which correspond to methods.
- Operations are modeled by listing the operation name preceded by an access modifier (in this case +) and followed by a set of parentheses.

  The plus sign (+) corresponds to the keyword public

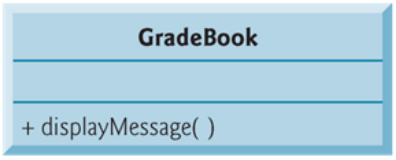

Fig. 3.3 | UML class diagram indicating that class `GradeBook` has a `public` `displayMessage` operation.

1.3 Declaring a Method with a Parameter

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

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

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

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

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

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

Reads a `String` from the user

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

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

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

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

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

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

- Scanner method nextLine
  - Reads characters typed by the user until the newline character is encountered
  - Returns a String containing the characters up to, but not including, the newline
  - Press *Enter* to submit the string to the program.
  - Pressing *Enter* inserts a newline character at the end of the characters the user typed.
  - The newline character is discarded by nextLine.
- Scanner method next
  - Reads individual words
  - Reads characters until a white-space character is encountered, then returns a String (the white-space character is discarded).
  - Information after the first white-space character can be read by other statements that call the Scanner's methods later in the program-.
- The number of arguments in a method call must match the number of parameters in the parameter list of the method's declaration.
- The argument types in the method call must be "consistent with" the types of the corresponding parameters in the method's declaration.
- The UML class diagram of Fig. 3.6 models class GradeBook of Fig. 3.4.
- The UML models a parameter by listing the parameter name, followed by a colon and the parameter type in the parentheses- following the operation name.
- The UML type String corresponds to the Java type String.

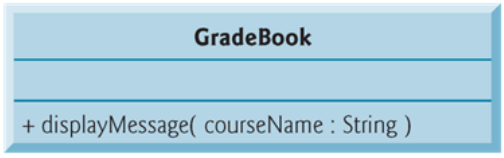

Fig. 3.6 | UML class diagram indicating that class `GradeBook` has a `displayMessage` operation with a `courseName` parameter of UML type `String`.

-

1.4 Instance Variables, set Methods and get Methods

- Local variables

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

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

  Each object (instance) of the class has a separate instance of the variable in memory.

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

Each GradeBook object maintains its own copy of instance variable courseName

Method allows client code to change the courseName

Method allows client code to obtain the courseName

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

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

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

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

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

- Every instance (i.e., object) of a class contains one copy of each instance variable.
- Instance variables typically declared private.
  - private is an access modifier.
  - private variables and methods are accessible only to methods of the class in which they are declared.

- Declaring instance private is known as data hiding or information hiding.
- private variables are encapsulated (hidden) in the object and can be accessed only by methods of the object's class.
  - Prevents instance variables from being modified accidentally by a class in another part of the program.
  - *Set* and *get* methods used to access instance variables.
- When a method that specifies a return type other than void completes its task, the method returns a result to its calling method.
- Method setCourseName and getCourseName each use variable courseName even though it was not declared in any of the methods.
  - Can use an instance variable of the class in each of the classes methods.
  - Exception to this is static methods (Chapter 8)
- The order in which methods are declared in a class does not determine when they are called at execution time.
- One method of a class can call another method of the same class by using just the method name.
- Unlike local variables, which are not automatically initialized, every field has a default initial value—a value provided by Java when you do not specify the field's initial value.
- Fields are not required to be explicitly initialized before they are used in a program—unless they must be initialized to values other than their default values.
- The default value for a field of type String is null

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

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

Sets the value of the courseName instance variable

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

- *set* and *get* methods
  - A class's private fields can be manipulated only by the class's methods.
  - A client of an object calls the class's public methods to manipulate the private fields of an object of the class.
  - Classes often provide public methods to allow clients to *set* (i.e., assign values to) or *get* (i.e., obtain the values of) private instance variables.
  - The names of these methods need not begin with *set* or *get*, but this naming convention is recommended.
- Figure 3.9 contains an updated UML class diagram for the version of class GradeBook in Fig. 3.7.

- Models instance variable courseName as an attribute in the middle compartment of the class.
- The UML represents instance variables as attributes by listing the attribute name, followed by a colon and the attribute type.
- A minus sign (–) access modifier corresponds to access modifier private.

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

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

```
Initial course name is: null

Please enter the course name:
CS101 Introduction to Java Programming

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

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

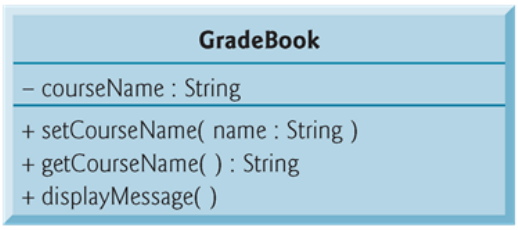

Fig. 3.9 | UML class diagram indicating that class `GradeBook` has a private `courseName` attribute of UML type `String` and three public operations—`setCourseName` (with a `name` parameter of UML type `String`), `getCourseName` (which returns UML type `String`) and `displayMessage`.

1.5 Primitive Types vs. Reference Types

- Types are divided into primitive types and reference types.
- The primitive types are boolean, byte, char, short, int, long, float and double.
- All nonprimitive types are reference types.
- A primitive-type variable can store exactly one value of its declared type at a time.
- Primitive-type instance variables are initialized by default—variables of types byte, char, short, int, long, float and double are initialized to 0, and variables of type boolean are initialized to false.
- You can specify your own initial value for a primitive-type variable by assigning the variable a value in its declaration.
- Programs use variables of reference types (normally called references) to store the locations of objects in the computer's memory. Such a variable is said to refer to an object in the program.
- Objects that are referenced may each contain many instance variables and methods.
- Reference-type instance variables are initialized by default to the value null

- A reserved word that represents a “reference to nothing.”
- When using an object of another class, a reference to the object is required to invoke (i.e., call) its methods. Also known as sending messages to an object.

1.6 Initializing Objects with Constructors

- When an object of a class is created, its instance variables are initialized by default.
- Each class can provide a constructor that initializes an object of a class when the object is created.
- Java requires a constructor call for *every* object that is created.
- Keyword new requests memory from the system to store an object, then calls the corresponding class’s constructor to initialize the object.
- A constructor *must* have the same name as the class.
- By default, the compiler provides a default constructor with no parameters in any class that does not explicitly include a constructor.
- Instance variables are initialized to their default values.
- Can provide your own constructor to specify custom initialization for objects of your class.
- A constructor’s parameter list specifies the data it requires to perform its task.
- Constructors cannot return values, so they cannot specify a return type.
- Normally, constructors are declared public.
- *If you declare any constructors for a class, the Java compiler will not create a default constructor for that class.*
- The UML class diagram of Fig. 3.12 models class GradeBook of Fig. 3.10, which has a constructor that has a name parameter of type String.
- Like operations, the UML models constructors in the third compartment of a class in a class diagram.
- To distinguish a constructor, the UML requires that the word “constructor” be placed between guillemets (« and ») before the constructor’s name.
- List constructors before other operations in the third compartment.
-

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

Constructor that initializes courseName to the specified value

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

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

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

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

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

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

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

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

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

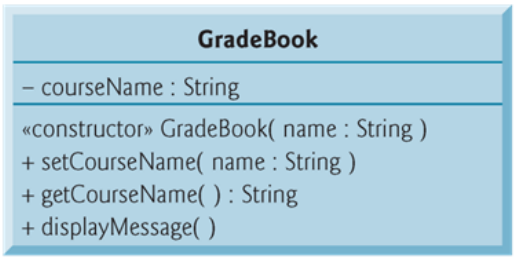

Fig. 3.12 | UML class diagram indicating that class `GradeBook` has a constructor that has a `name` parameter of UML type `String`.

Notes on import Declarations

- Classes System and String are in package java.lang
- Implicitly imported into every Java program
- Can use the java.lang classes without explicitly importing them
- Most classes you'll use in Java programs must be imported explicitly.
- Classes that are compiled in the same directory on disk are in the same package—known as the default package.
- Classes in the same package are implicitly imported into the source-code files of other classes in the same package.
- An import declaration is not required if you always refer to a class via its fully qualified class name
- Package name followed by a dot (.) and the class name.

Java API Packages

- Java contains many predefined classes that are grouped into categories of related classes called packages.
- A great strength of Java is the Java API's thousands of classes.
- Some key Java API packages are described in Fig. 6.5.

Package	Description
`java.applet`	The **Java Applet Package** contains a class and several interfaces required to create Java applets—programs that execute in web browsers. Applets are discussed in Chapter 23, Applets and Java Web Start; interfaces are discussed in Chapter 10, Object-Oriented Programming: Polymorphism.)
`java.awt`	The **Java Abstract Window Toolkit Package** contains the classes and interfaces required to create and manipulate GUIs in early versions of Java. In current versions, the Swing GUI components of the `javax.swing` packages are typically used instead. (Some elements of the `java.awt` package are discussed in Chapter 14, GUI Components: Part 1, Chapter 15, Graphics and Java 2D, and Chapter 25, GUI Components: Part 2.)
`java.awt.event`	The **Java Abstract Window Toolkit Event Package** contains classes and interfaces that enable event handling for GUI components in both the `java.awt` and `javax.swing` packages. (See Chapter 14, GUI Components: Part 1, and Chapter 25, GUI Components: Part 2.)

Fig. 6.5 | Java API packages (a subset). (Part 1 of 3.)

Package	Description
`java.awt.geom`	The **Java 2D Shapes Package** contains classes and interfaces for working with Java's advanced two-dimensional graphics capabilities. (See Chapter 15, Graphics and Java 2D.)
`java.io`	The **Java Input/Output Package** contains classes and interfaces that enable programs to input and output data. (See Chapter 17, Files, Streams and Object Serialization.)
`java.lang`	The **Java Language Package** contains classes and interfaces (discussed bookwide) that are required by many Java programs. This package is imported by the compiler into all programs.
`java.net`	The **Java Networking Package** contains classes and interfaces that enable programs to communicate via computer networks like the Internet. (See Chapter 27, Networking.)
`java.sql`	The **JDBC Package** contains classes and interfaces for working with databases. (See Chapter 28, Accessing Databases with JDBC.)
`java.util`	The **Java Utilities Package** contains utility classes and interfaces that enable such actions as date and time manipulations, random-number processing (class `Random`) and the storing and processing of large amounts of data. (See Chapter 20, Generic Collections.)

Fig. 6.5 | Java API packages (a subset). (Part 2 of 3.)

Package	Description
`java.util.concurrent`	The **Java Concurrency Package** contains utility classes and interfaces for implementing programs that can perform multiple tasks in parallel. (See Chapter 26, Multithreading.)
`javax.media`	The **Java Media Framework Package** contains classes and interfaces for working with Java's multimedia capabilities. (See Chapter 24, Multimedia: Applets and Applications.)
`javax.swing`	The **Java Swing GUI Components Package** contains classes and interfaces for Java's Swing GUI components that provide support for portable GUIs. (See Chapter 14, GUI Components: Part 1, and Chapter 25, GUI Components: Part 2.)
`javax.swing.event`	The **Java Swing Event Package** contains classes and interfaces that enable event handling (e.g., responding to button clicks) for GUI components in package `javax.swing`. (See Chapter 14, GUI Components: Part 1, and Chapter 25, GUI Components: Part 2.)
`javax.xml.ws`	The **JAX-WS Package** contains classes and interfaces for working with web services in Java. (See Chapter 31, Web Services.)

Fig. 6.5 | Java API packages (a subset). (Part 3 of 3.)

2. Methods – A deeper Look

2.2 static Methods, static Fields and Class Math

- Sometimes a method performs a task that does not depend on the contents of any object.
  - Applies to the class in which it's declared as a whole
  - Known as a static method or a class method
- It's common for classes to contain convenient static methods to perform common tasks.
- To declare a method as static, place the keyword static before the return type in the method's declaration.
- Calling a static method

  ClassName.methodName(arguments)
- Class Math provides a collection of static methods that enable you to perform common mathematical calculations.
- Method arguments may be constants, variables or expressions.

Method	Description	Example
`abs( x )`	absolute value of *x*	`abs( 23.7 )` is 23.7 `abs( 0.0 )` is 0.0 `abs( -23.7 )` is 23.7
`ceil( x )`	rounds *x* to the smallest integer not less than *x*	`ceil( 9.2 )` is 10.0 `ceil( -9.8 )` is -9.0
`cos( x )`	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 )` is 1.0
`exp( x )`	exponential method e^x	`exp( 1.0 )` is 2.71828 `exp( 2.0 )` is 7.38906
`floor( x )`	rounds *x* to the largest integer not greater than *x*	`floor( 9.2 )` is 9.0 `floor( -9.8 )` is -10.0
`log( x )`	natural logarithm of *x* (base *e*)	`log( Math.E )` is 1.0 `log( Math.E * Math.E )` is 2.0
`max( x, y )`	larger value of *x* and *y*	`max( 2.3, 12.7 )` is 12.7 `max( -2.3, -12.7 )` is -2.3
`min( x, y )`	smaller value of *x* and *y*	`min( 2.3, 12.7 )` is 2.3 `min( -2.3, -12.7 )` is -12.7

Fig. 6.2 | `Math` class methods. (Part 1 of 2.)

Method	Description	Example
`pow( x, y )`	*x* raised to the power *y* (i.e., x^y)	`pow( 2.0, 7.0 )` is 128.0 `pow( 9.0, 0.5 )` is 3.0
`sin( x )`	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 )` is 0.0
`sqrt( x )`	square root of *x*	`sqrt( 900.0 )` is 30.0
`tan( x )`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 )` is 0.0

Fig. 6.2 | `Math` class methods. (Part 2 of 2.)

- Math fields for common mathematical constants
  - Math.PI (3.141592653589793)
  - Math.E (2.718281828459045)
- Declared in class Math with the modifiers public, final and static
  - public allows you to use these fields in your own classes.
  - A field declared with keyword final is constant—its value cannot change after the field is initialized.
  - PI and E are declared final because their values never change.
- A field that represents an attribute is also known as an instance variable—each object (instance) of the class has a separate instance of the variable in memory.
- Fields for which each object of a class does not have a separate instance of the field are declared static and are also known as class variables.
- All objects of a class containing static fields share one copy of those fields.
- Together the class variables (i.e., static variables) and instance variables represent the fields of a class.
- Why is method **main** declared **static**?
  - The JVM attempts to invoke the main method of the class you specify—when no objects of the class have been created.
  - Declaring main as static allows the JVM to invoke main without creating an instance of the class.

2.3 Declaring Methods with Multiple Parameters

- Multiple parameters are specified as a comma-separated list.
- There must be one argument in the method call for each parameter (sometimes called a formal parameter) in the method declaration.
- Each argument must be consistent with the type of the corresponding parameter.
- Implementing method maximum by reusing method Math.max
  - Two calls to Math.max, as follows: return Math.max(x, Math.max(y, z));
  - The first specifies arguments x and Math.max(y, z).
  - Before any method can be called, its arguments must be evaluated to determine their values.
  - If an argument is a method call, the method call must be performed to determine its return value.
  - The result of the first call is passed as the second argument to the other call, which returns the larger of its two arguments.
- String concatenation
  - Assemble String objects into larger strings with operators + or +=.
  - When both operands of operator + are Strings, operator + creates a new String object
  - characters of the right operand are placed at the end of those in the left operand
  - Every primitive value and object in Java has a String representation.
  - When one of the + operator's operands is a String, the other is converted to a String, then the two are concatenated.
  - If a boolean is concatenated with a String, the boolean is converted to the String "true" or "false".
  - All objects have a toString method that returns a String representation of the object

```
1  // Fig. 6.3: MaximumFinder.java
2  // Programmer-declared method maximum with three double parameters.
3  import java.util.Scanner;
4
5  public class MaximumFinder
6  {
7     // obtain three floating-point values and locate the maximum value
8     public static void main( String[] args )
9     {
10       // create Scanner for input from command window
11       Scanner input = new Scanner( System.in );
12
13       // prompt for and input three floating-point values
14       System.out.print(
15          "Enter three floating-point values separated by spaces: " );
16       double number1 = input.nextDouble(); // read first double
17       double number2 = input.nextDouble(); // read second double
18       double number3 = input.nextDouble(); // read third double
19
20       // determine the maximum value
21       double result = maximum( number1, number2, number3 );
22
```

Fig. 6.3 | Programmer-declared method `maximum` with three `double` parameters. (Part 1 of 3.)

```
23       // display maximum value
24       System.out.println( "Maximum is: " + result );
25    } // end main
26
27    // returns the maximum of its three double parameters
28    public static double maximum( double x, double y, double z )
29    {
30       double maximumValue = x; // assume x is the largest to start
31
32       // determine whether y is greater than maximumValue
33       if ( y > maximumValue )
34          maximumValue = y;
35
36       // determine whether z is greater than maximumValue
37       if ( z > maximumValue )
38          maximumValue = z;
39
40       return maximumValue;
41    } // end method maximum
42 } // end class MaximumFinder
```

Fig. 6.3 | Programmer-declared method `maximum` with three `double` parameters. (Part 2 of 3.)

```
Enter three floating-point values separated by spaces: 9.35 2.74 5.1
Maximum is: 9.35
```

```
Enter three floating-point values separated by spaces: 5.8 12.45 8.32
Maximum is: 12.45
```

```
Enter three floating-point values separated by spaces: 6.46 4.12 10.54
Maximum is: 10.54
```

Fig. 6.3 | Programmer-declared method `maximum` with three `double` parameters. (Part 3 of 3.)

2.4 Notes on Declaring and Using Methods

- Three ways to call a method:
  - Using a method name by itself to call another method of the same class
  - Using a variable that contains a reference to an object, followed by a dot (.) and the method name to call a method of the referenced object
  - Using the class name and a dot (.) to call a static method of a class
- A non-static method can call any method of the same class directly and can manipulate any of the class's fields directly.
- A static method can call *only other static methods* of the same class directly and can manipulate *only static fields* in the same class directly.
- To access the class's non-static members, a static method must use a reference to an object of the class.
- A non-static method can call any method of the same class directly and can manipulate any of the class's fields directly.
- A static method can call *only other static methods* of the same class directly and can manipulate *only static fields* in the same class directly.
- To access the class's non-static members, a static method must use a reference to an object of the class.
- Three ways to return control to the statement that calls a method:
  - When the program flow reaches the method-ending right brace
  - When the following statement executes
    `return;`
  - When the method returns a result with a statement like
    `return` *expression*`;`

2.5 Method-Call Stack and Activation Records

- Stack data structure
  - Analogous to a pile of dishes
  - A dish is placed on the pile at the top (referred to as pushing the dish onto the stack).
  - A dish is removed from the pile from the top (referred to as popping the dish off the stack).

- Last-in, first-out (LIFO) data structures
  - The last item pushed (inserted) on the stack is the first item popped (removed) from the stack.
- When a program calls a method, the called method must know how to return to its caller
  - The return address of the calling method is pushed onto the program-execution (or method-call) stack.
- If a series of method calls occurs, the successive return addresses are pushed onto the stack in last-in, first-out order.
- The program-execution stack also contains the memory for the local variables used in each invocation of a method during a program's execution.
  - Stored as a portion of the program-execution stack known as the activation record or stack frame of the method call.
- When a method call is made, the activation record for that method call is pushed onto the program-execution stack.
- When the method returns to its caller, the method's activation record is popped off the stack and those local variables are no longer known to the program.
- If more method calls occur than can have their activation records stored on the program-execution stack, an error known as a stack overflow occurs.

2.6 Argument Promotion and Casting

- Argument promotion
  - Converting an argument's value, if possible, to the type that the method expects to receive in its corresponding parameter.
- Conversions may lead to compilation errors if Java's promotion rules are not satisfied.
- Promotion rules
  - specify which conversions are allowed.
  - apply to expressions containing values of two or more primitive types and to primitive-type values passed as arguments to methods.
- Each value is promoted to the "highest" type in the expression.
- Figure 6.4 lists the primitive types and the types to which each can be promoted.

Type	Valid promotions
`double`	None
`float`	`double`
`long`	`float` or `double`
`int`	`long`, `float` or `double`
`char`	`int`, `long`, `float` or `double`
`short`	`int`, `long`, `float` or `double` (but not `char`)
`byte`	`short`, `int`, `long`, `float` or `double` (but not `char`)
`boolean`	None (`boolean` values are not considered to be numbers in Java)

Fig. 6.4 | Promotions allowed for primitive types.

- Converting values to types lower in the table of Fig. 6.4 will result in different values if the lower type cannot represent the value of the higher type

- In cases where information may be lost due to conversion, the Java compiler requires you to use a cast operator to explicitly force the conversion to occur—otherwise a compilation error occurs.

2.7 Case Study: Random-Number Generation

- Simulation and game playing
  - element of chance
  - Class Random (package java.util)
  - static method random of class Math.
- Objects of class Random can produce random boolean, byte, float, double, int, long and Gaussian values
- Math method random can produce only double values in the range 0.0 £ *x < 1.0.*
- Class Random produces pseudorandom numbers
- A sequence of values produced by a complex mathematical calculation.
- The calculation uses the current time of day to seed the random-number generator.
- The range of values produced directly by Random method nextInt often differs from the range of values required in a particular Java application.
- Random method nextInt that receives an int argument returns a value from 0 up to, but not including, the argument's value.
- Rolling a Six-Sided Die

  face = 1 + randomNumbers.nextInt(6);
- The argument 6—called the scaling factor—represents the number of unique values that nextInt should produce (0–5)
- This is called scaling the range of values
- A six-sided die has the numbers 1–6 on its faces, not 0–5.
- We shift the range of numbers produced by adding a shifting value—in this case 1—to our previous result, as in
- The shifting value (1) specifies the first value in the desired range of random integers.
- Generalize the scaling and shifting of random numbers:

  number = *shiftingValue* + randomNumbers.nextInt(*scalingFactor*);

  where *shiftingValue* specifies the first number in the desired range of consecutive integers and *scalingFactor* specifies how many numbers are in the range.
- It's also possible to choose integers at random from sets of values other than ranges of consecutive integers:

  number = *shiftingValue* + *differenceBetweenValues* *
  randomNumbers.nextInt(*scalingFactor*);

  where *shiftingValue* specifies the first number in the desired range of values, *differenceBetweenValues* represents the constant difference between consecutive numbers in the sequence and *scalingFactor* specifies how many numbers are in the range.
- When debugging an application, it's sometimes useful to repeat the exact same sequence of pseudorandom numbers.
- To do so, create a Random object as follows:

```
Random randomNumbers =
  new Random( seedValue );
```

  seedValue (of type long) seeds the random-number calculation.
- You can set a Random object's seed at any time during program execution by calling the object's set method.

```java
// Fig. 6.6: RandomIntegers.java
// Shifted and scaled random integers.
import java.util.Random; // program uses class Random

public class RandomIntegers
{
   public static void main( String[] args )
   {
      Random randomNumbers = new Random(); // random number generator
      int face; // stores each random integer generated

      // loop 20 times
      for ( int counter = 1; counter <= 20; counter++ )
      {
         // pick random integer from 1 to 6
         face = 1 + randomNumbers.nextInt( 6 );

         System.out.printf( "%d  ", face ); // display generated value

         // if counter is divisible by 5, start a new line of output
         if ( counter % 5 == 0 )
            System.out.println();
      } // end for
   } // end main
} // end class RandomIntegers
```

Fig. 6.6 | Shifted and scaled random integers. (Part 1 of 2.)

```
1  5  3  6  2
5  2  6  5  2
4  4  4  2  6
3  1  6  2  2
```

```
6  5  4  2  6
1  2  5  1  3
6  3  2  2  1
6  4  2  6  4
```

Fig. 6.6 | Shifted and scaled random integers. (Part 2 of 2.)

- Fig 6.7: Rolling a Six-Sided Die 6,000,000 Times

```
1  // Fig. 6.7: RollDie.java
2  // Roll a six-sided die 6,000,000 times.
3  import java.util.Random;
4
5  public class RollDie
6  {
7     public static void main( String[] args )
8     {
9        Random randomNumbers = new Random(); // random number generator
10
11       int frequency1 = 0; // maintains count of 1s rolled
12       int frequency2 = 0; // count of 2s rolled
13       int frequency3 = 0; // count of 3s rolled
14       int frequency4 = 0; // count of 4s rolled
15       int frequency5 = 0; // count of 5s rolled
16       int frequency6 = 0; // count of 6s rolled
17
18       int face; // most recently rolled value
19
20       // tally counts for 6,000,000 rolls of a die
21       for ( int roll = 1; roll <= 6000000; roll++ )
22       {
23          face = 1 + randomNumbers.nextInt( 6 ); // number from 1 to 6
24
```

Fig. 6.7 | Roll a six-sided die 6,000,000 times. (Part 1 of 3.)

```
25          // determine roll value 1-6 and increment appropriate counter
26          switch ( face )
27          {
28             case 1:
29                ++frequency1; // increment the 1s counter
30                break;
31             case 2:
32                ++frequency2; // increment the 2s counter
33                break;
34             case 3:
35                ++frequency3; // increment the 3s counter
36                break;
37             case 4:
38                ++frequency4; // increment the 4s counter
39                break;
40             case 5:
41                ++frequency5; // increment the 5s counter
42                break;
43             case 6:
44                ++frequency6; // increment the 6s counter
45                break; // optional at end of switch
46          } // end switch
47       } // end for
48
```

Fig. 6.7 | Roll a six-sided die 6,000,000 times. (Part 2 of 3.)

```
49          System.out.println( "Face\tFrequency" ); // output headers
50          System.out.printf( "1\t%d\n2\t%d\n3\t%d\n4\t%d\n5\t%d\n6\t%d\n",
51             frequency1, frequency2, frequency3, frequency4,
52             frequency5, frequency6 );
53       } // end main
54    } // end class RollDie
```

```
Face    Frequency
1       999501
2       1000412
3       998262
4       1000820
5       1002245
6       998760
```

```
Face    Frequency
1       999647
2       999557
3       999571
4       1000376
5       1000701
6       1000148
```

Fig. 6.7 | Roll a six-sided die 6,000,000 times. (Part 3 of 3.)

2.9 Case Study: A Game of Chance; Introducing Enumerations

- Basic rules for the dice game Craps:
  - *You roll two dice. Each die has six faces, which contain one, two, three, four, five and six spots, respectively. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, you win. If the sum is 2, 3 or 12 on the first throw (called "craps"), you lose (i.e., the "house" wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first throw, that sum becomes your "point." To win, you must continue rolling the dice until you "make your point" (i.e., roll that same point value). You lose by rolling a 7 before making your point.*

Notes:

- myPoint is initialized to 0 to ensure that the application will compile.
- If you do not initialize myPoint, the compiler issues an error, because myPoint is not assigned a value in every case of the switch statement, and thus the program could try to use myPoint before it is assigned a value.
- gameStatus is assigned a value in every case of the switch statement—thus, it's guaranteed to be initialized before it's used and does not need to be initialized.
- **enum type Status**
  - An enumeration in its simplest form declares a set of constants represented by identifiers.
  - Special kind of class that is introduced by the keyword enum and a type name.
  - Braces delimit an enum declaration's body.
  - Inside the braces is a comma-separated list of enumeration constants, each representing a unique value.

- The identifiers in an enum must be unique.
- Variables of an enum type can be assigned only the constants declared in the enumeration.

- Why Some Constants Are Not Defined as enum Constants?
  - Java does not allow an int to be compared to an enumeration constant.
  - Java does not provide an easy way to convert an int value to a particular enum constant.
  - Translating an int into an enum constant could be done with a separate switch statement.
  - This would be cumbersome and not improve the readability of the program (thus defeating the purpose of using an enum).

```
1  // Fig. 6.8: Craps.java
2  // Craps class simulates the dice game craps.
3  import java.util.Random;
4
5  public class Craps
6  {
7     // create random number generator for use in method rollDice
8     private static final Random randomNumbers = new Random();
9
10    // enumeration with constants that represent the game status
11    private enum Status { CONTINUE, WON, LOST };
12
13    // constants that represent common rolls of the dice
14    private static final int SNAKE_EYES = 2;
15    private static final int TREY = 3;
16    private static final int SEVEN = 7;
17    private static final int YO_LEVEN = 11;
18    private static final int BOX_CARS = 12;
19
```

Fig. 6.8 | Craps class simulates the dice game craps. (Part 1 of 5.)

```
20      // plays one game of craps
21      public static void main( String[] args )
22      {
23         int myPoint = 0; // point if no win or loss on first roll
24         Status gameStatus; // can contain CONTINUE, WON or LOST
25
26         int sumOfDice = rollDice(); // first roll of the dice
27
28         // determine game status and point based on first roll
29         switch ( sumOfDice )
30         {
31            case SEVEN: // win with 7 on first roll
32            case YO_LEVEN: // win with 11 on first roll
33               gameStatus = Status.WON;
34               break;
35            case SNAKE_EYES: // lose with 2 on first roll
36            case TREY: // lose with 3 on first roll
37            case BOX_CARS: // lose with 12 on first roll
38               gameStatus = Status.LOST;
39               break;
```

Fig. 6.8 | `Craps` class simulates the dice game craps. (Part 2 of 5.)

```
40            default: // did not win or lose, so remember point
41               gameStatus = Status.CONTINUE; // game is not over
42               myPoint = sumOfDice; // remember the point
43               System.out.printf( "Point is %d\n", myPoint );
44               break; // optional at end of switch
45         } // end switch
46
47         // while game is not complete
48         while ( gameStatus == Status.CONTINUE ) // not WON or LOST
49         {
50            sumOfDice = rollDice(); // roll dice again
51
52            // determine game status
53            if ( sumOfDice == myPoint ) // win by making point
54               gameStatus = Status.WON;
55            else
56               if ( sumOfDice == SEVEN ) // lose by rolling 7 before point
57                  gameStatus = Status.LOST;
58         } // end while
59
```

Fig. 6.8 | `Craps` class simulates the dice game craps. (Part 3 of 5.)

```
60          // display won or lost message
61          if ( gameStatus == Status.WON )
62             System.out.println( "Player wins" );
63          else
64             System.out.println( "Player loses" );
65       } // end main
66
67       // roll dice, calculate sum and display results
68       public static int rollDice()
69       {
70          // pick random die values
71          int die1 = 1 + randomNumbers.nextInt( 6 ); // first die roll
72          int die2 = 1 + randomNumbers.nextInt( 6 ); // second die roll
73
74          int sum = die1 + die2; // sum of die values
75
76          // display results of this roll
77          System.out.printf( "Player rolled %d + %d = %d\n",
78             die1, die2, sum );
79
80          return sum; // return sum of dice
81       } // end method rollDice
82    } // end class Craps
```

Fig. 6.8 | `Craps` class simulates the dice game craps. (Part 4 of 5.)

```
Player rolled 5 + 6 = 11
Player wins
```

```
Player rolled 5 + 4 = 9
Point is 9
Player rolled 4 + 2 = 6
Player rolled 3 + 6 = 9
Player wins
```

```
Player rolled 1 + 2 = 3
Player loses
```

```
Player rolled 2 + 6 = 8
Point is 8
Player rolled 5 + 1 = 6
Player rolled 2 + 1 = 3
Player rolled 1 + 6 = 7
Player loses
```

Fig. 6.8 | `Craps` class simulates the dice game craps. (Part 5 of 5.)

2.10 Scope of Declarations

- Declarations introduce names that can be used to refer to such Java entities.
- The scope of a declaration is the portion of the program that can refer to the declared entity by its name.
  - Such an entity is said to be “in scope” for that portion of the program.
- Basic scope rules:
  - The scope of a parameter declaration is the body of the method in which the declaration appears.
  - The scope of a local-variable declaration is from the point at which the declaration appears to the end of that block.
  - The scope of a local-variable declaration that appears in the initialization section of a for statement’s header is the body of the for statement and the other expressions in the header.
  - A method or field’s scope is the entire body of the class.
- Any block may contain variable declarations.
- If a local variable or parameter in a method has the same name as a field of the class, the field is “hidden” until the block terminates execution—this is called shadowing.

```
1  // Fig. 6.9: Scope.java
2  // Scope class demonstrates field and local variable scopes.
3
4  public class Scope
5  {
6     // field that is accessible to all methods of this class
7     private static int x = 1;
8
9     // method main creates and initializes local variable x
10    // and calls methods useLocalVariable and useField
11    public static void main( String[] args )
12    {
13       int x = 5; // method's local variable x shadows field x
14
15       System.out.printf( "local x in main is %d\n", x );
16
17       useLocalVariable(); // useLocalVariable has local x
18       useField(); // useField uses class Scope's field x
19       useLocalVariable(); // useLocalVariable reinitializes local x
20       useField(); // class Scope's field x retains its value
21
22       System.out.printf( "\nlocal x in main is %d\n", x );
23    } // end main
```

Fig. 6.9 | Scope class demonstrates field and local variable scopes. (Part 1 of 3.)

```
24
25    // create and initialize local variable x during each call
26    public static void useLocalVariable()
27    {
28       int x = 25; // initialized each time useLocalVariable is called
29
30       System.out.printf(
31          "\nlocal x on entering method useLocalVariable is %d\n", x );
32       ++x; // modifies this method's local variable x
33       System.out.printf(
34          "local x before exiting method useLocalVariable is %d\n", x );
35    } // end method useLocalVariable
36
37    // modify class Scope's field x during each call
38    public static void useField()
39    {
40       System.out.printf(
41          "\nfield x on entering method useField is %d\n", x );
42       x *= 10; // modifies class Scope's field x
43       System.out.printf(
44          "field x before exiting method useField is %d\n", x );
45    } // end method useField
46 } // end class Scope
```

Fig. 6.9 | Scope class demonstrates field and local variable scopes. (Part 2 of 3.)

```
local x in main is 5

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 1
field x before exiting method useField is 10

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 10
field x before exiting method useField is 100

local x in main is 5
```

Fig. 6.9 | Scope class demonstrates field and local variable scopes. (Part 3 of 3.)

2.11 Method Overloading

- Method overloading
  - Methods of the same name declared in the same class
  - Must have different sets of parameters
- Compiler selects the appropriate method to call by examining the number, types and order of the arguments in the call.
- Used to create several methods with the same name that perform the same or similar tasks, but on different types or different numbers of arguments.
- Literal integer values are treated as type int, so the method call in line 9 invokes the version of square that specifies an int parameter.
- Literal floating-point values are treated as type double, so the method call in line 10 invokes the version of square that specifies a double parameter.

```
1  // Fig. 6.10: MethodOverload.java
2  // Overloaded method declarations.
3
4  public class MethodOverload
5  {
6     // test overloaded square methods
7     public static void main( String[] args )
8     {
9        System.out.printf( "Square of integer 7 is %d\n", square( 7 ) );
10       System.out.printf( "Square of double 7.5 is %f\n", square( 7.5 ) );
11    } // end main
12
13    // square method with int argument
14    public static int square( int intValue )
15    {
16       System.out.printf( "\nCalled square with int argument: %d\n",
17          intValue );
18       return intValue * intValue;
19    } // end method square with int argument
20
```

Fig. 6.10 | Overloaded method declarations. (Part 1 of 2.)

```
21    // square method with double argument
22    public static double square( double doubleValue )
23    {
24       System.out.printf( "\nCalled square with double argument: %f\n",
25          doubleValue );
26       return doubleValue * doubleValue;
27    } // end method square with double argument
28 } // end class MethodOverload
```

```
Called square with int argument: 7
Square of integer 7 is 49

Called square with double argument: 7.500000
Square of double 7.5 is 56.250000
```

Fig. 6.10 | Overloaded method declarations. (Part 2 of 2.)

3. Arrays

- Array
  - Group of variables (called elements) containing values of the same type.
  - Arrays are objects so they are reference types.
  - Elements can be either primitive or reference types.
- Refer to a particular element in an array
  - Use the element's index.

- Array-access expression—the name of the array followed by the index of the particular element in square brackets, [].

- The first element in every array has index zero.
- The highest index in an array is one less than the number of elements in the array.
- Array names follow the same conventions as other variable names.
- An index must be a nonnegative integer.
- Can use an expression as an index.
- An indexed array name is an array-access expression.
- Can be used on the left side of an assignment to place a new value into an array element.
- Every array object knows its own length and stores it in a length instance variable.
- Length cannot be changed because it's a final variable.

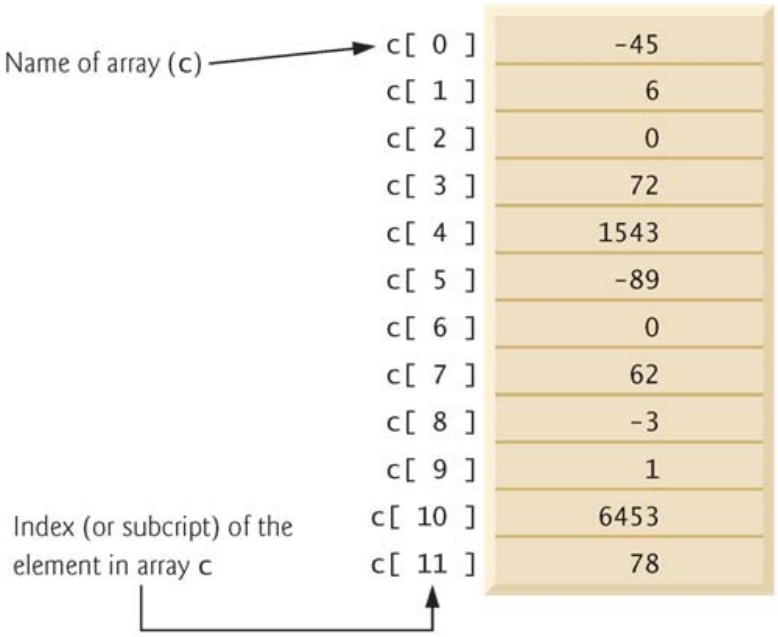

Fig. 7.1 | A 12-element array.

3.1 Declaring and Creating Arrays

- Array objects
  - Created with keyword new.
  - You specify the element type and the number of elements in an array-creation expression, which returns a reference that can be stored in an array variable.
- Declaration and array-creation expression for an array of 12 int elements
  ```
  int[] c = new int[ 12 ];
  ```
- Can be performed in two steps as follows:
  ```
  int[] c; // declare the array variable
  c = new int[ 12 ]; // creates the array
  ```

- In a declaration, square brackets following a type indicate that a variable will refer to an array (i.e., store an array reference).
- When an array is created, each element of the array receives a default value
  - Zero for the numeric primitive-type elements, false for boolean elements and null for references.
- When the element type and the square brackets are combined at the beginning of the declaration, all the identifiers in the declaration are array variables. For readability, declare only one variable per declaration.
- Every element of a primitive-type array contains a value of the array's declared element type.
  - Every element of an int array is an int value.
- Every element of a reference-type array is a reference to an object of the array's declared element type.
  - Every element of a String array is a reference to a String object.
- Fig. 7.2 uses keyword new to create an array of 10 int elements, which are initially zero (the default for int variables).

```
1  // Fig. 7.2: InitArray.java
2  // Initializing the elements of an array to default values of zero.
3
4  public class InitArray
5  {
6     public static void main( String[] args )
7     {
8        int[] array; // declare array named array
9
10       array = new int[ 10 ]; // create the array object
11
12       System.out.printf( "%s%8s\n", "Index", "Value" ); // column headings
13
14       // output each array element's value
15       for ( int counter = 0; counter < array.length; counter++ )
16          System.out.printf( "%5d%8d\n", counter, array[ counter ] );
17    } // end main
18 } // end class InitArray
```

Variable `array` will refer to an array of `int` values.

Creates an array of `10` `int` elements, each with the value `0` by default

`for` statement iterates while `counter` is less than the array's `length`

Array-access expression gets the value at the index represented by `counter`

Fig. 7.2 | Initializing the elements of an array to default values of zero. (Part 1 of 2.)

```
Index   Value
    0       0
    1       0
    2       0
    3       0
    4       0
    5       0
    6       0
    7       0
    8       0
    9       0
```

Fig. 7.2 | Initializing the elements of an array to default values of zero. (Part 2 of 2.)

3.1.1 Examples Using Arrays

- Array initializer
  - A comma-separated list of expressions (called an initializer list) enclosed in braces.
  - Used to create an array and initialize its elements.
  - Array length is determined by the number of elements in the initializer list.

int[] n = { 10, 20, 30, 40, 50 };

- Creates a five-element array with index values 0–4.

- Compiler counts the number of initializers in the list to determine the size of the array
  - Sets up the appropriate new operation "behind the scenes."

```
1  // Fig. 7.3: InitArray.java
2  // Initializing the elements of an array with an array initializer.
3
4  public class InitArray
5  {
6     public static void main( String[] args )
7     {
8        // initializer list specifies the value for each element
9        int[] array = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
10
11       System.out.printf( "%s%8s\n", "Index", "Value" ); // column headings
12
13       // output each array element's value
14       for ( int counter = 0; counter < array.length; counter++ )
15          System.out.printf( "%5d%8d\n", counter, array[ counter ] );
16    } // end main
17 } // end class InitArray
```

Array initializer list for a 10-element `int` array

Fig. 7.3 | Initializing the elements of an array with an array initializer. (Part 1 of 2.)

```
Index   Value
    0      32
    1      27
    2      64
    3      18
    4      95
    5      14
    6      90
    7      70
    8      60
    9      37
```

Fig. 7.3 | Initializing the elements of an array with an array initializer. (Part 2 of 2.)

- The application in Fig. 7.4 creates a 10-element array and assigns to each element one of the even integers from 2 to 20 (2, 4, 6, …, 20).
- final variables must be initialized before they are used and cannot be modified thereafter.
- An attempt to modify a final variable after it's initialized causes a compilation error
  - cannot assign a value to final variable *variableName*
- An attempt to access the value of a final variable before it's initialized causes a compilation error
  - variable *variableName* might not have been initialized

```
1  // Fig. 7.4: InitArray.java
2  // Calculating the values to be placed into the elements of an array.
3
4  public class InitArray
5  {
6     public static void main( String[] args )
7     {
8        final int ARRAY_LENGTH = 10; // declare constant
9        int[] array = new int[ ARRAY_LENGTH ]; // create array
10
11       // calculate value for each array element
12       for ( int counter = 0; counter < array.length; counter++ )
13          array[ counter ] = 2 + 2 * counter;
14
15       System.out.printf( "%s%8s\n", "Index", "Value" ); // column headings
16
17       // output each array element's value
18       for ( int counter = 0; counter < array.length; counter++ )
19          System.out.printf( "%5d%8d\n", counter, array[ counter ] );
20    } // end main
21 } // end class InitArray
```

Fig. 7.4 | Calculating the values to be placed into the elements of an array. (Part 1 of 2.)

```
Index   Value
    0       2
    1       4
    2       6
    3       8
    4      10
    5      12
    6      14
    7      16
    8      18
    9      20
```

Fig. 7.4 | Calculating the values to be placed into the elements of an array. (Part 2 of 2.)

- Figure 7.5 sums the values contained in a 10-element integer array.
- Often, the elements of an array represent a series of values to be used in a calculation.

```
1  // Fig. 7.5: SumArray.java
2  // Computing the sum of the elements of an array.
3
4  public class SumArray
5  {
6     public static void main( String[] args )
7     {
8        int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
9        int total = 0;
10
11       // add each element's value to total
12       for ( int counter = 0; counter < array.length; counter++ )
13          total += array[ counter ];
14
15       System.out.printf( "Total of array elements: %d\n", total );
16    } // end main
17 } // end class SumArray
```

Adds each value in `array` to `total`, which is displayed when the loop terminates

```
Total of array elements: 849
```

Fig. 7.5 | Computing the sum of the elements of an array.

- Many programs present data to users in a graphical manner.
- Numeric values are often displayed as bars in a bar chart.
  - Longer bars represent proportionally larger numeric values.
- A simple way to display numeric data is with a bar chart that shows each numeric value as a bar of asterisks (*).
- Format specifier %02d indicates that an int value should be formatted as a field of two digits.
  - The 0 flag displays a leading 0 for values with fewer digits than the field width (2).

```
1  // Fig. 7.6: BarChart.java
2  // Bar chart printing program.
3
4  public class BarChart
5  {
6     public static void main( String[] args )
7     {
8        int[] array = { 0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 };
9
10       System.out.println( "Grade distribution:" );
11
12       // for each array element, output a bar of the chart
13       for ( int counter = 0; counter < array.length; counter++ )
14       {
15          // output bar label ( "00-09: ", ..., "90-99: ", "100: " )
16          if ( counter == 10 )
17             System.out.printf( "%5d: ", 100 );
18          else
19             System.out.printf( "%02d-%02d: ",
20                counter * 10, counter * 10 + 9  );
21
```

Fig. 7.6 | Bar chart printing program. (Part 1 of 2.)

```
22          // print bar of asterisks
23          for ( int stars = 0; stars < array[ counter ]; stars++ )
24             System.out.print( "*" );
25
26          System.out.println(); // start a new line of output
27       } // end outer for
28    } // end main
29 } // end class BarChart
```

Nested `for` loop uses the outer `for` loop's `counter` variable to determine which element of the array to access, then displays the appropriate number of asterisks

```
Grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

Fig. 7.6 | Bar chart printing program. (Part 2 of 2.)

- Sometimes, programs use counter variables to summarize data, such as the results of a survey.
- Fig. 6.7 used separate counters in a die-rolling program to track the number of occurrences of each side of a six-sided die as the program rolled the die 6,000,000 times.
- Fig. 7.7 shows an array version of this application.
  - Line 14 of this program replaces lines 23–46 of Fig. 6.7.
- Array frequency must be large enough to store six counters.
  - We use a seven-element array in which we ignore frequency[0]
  - More logical to have the face value 1 increment frequency[1] than frequency[0].

```
1  // Fig. 7.7: RollDie.java
2  // Die-rolling program using arrays instead of switch.
3  import java.util.Random;
4
5  public class RollDie
6  {
7     public static void main( String[] args )
8     {
9        Random randomNumbers = new Random(); // random number generator
10       int[] frequency = new int[ 7 ]; // array of frequency counters
11
12       // roll die 6,000,000 times; use die value as frequency index
13       for ( int roll = 1; roll <= 6000000; roll++ )
14          ++frequency[ 1 + randomNumbers.nextInt( 6 ) ];
15
16       System.out.printf( "%s%10s\n", "Face", "Frequency" );
17
18       // output each array element's value
19       for ( int face = 1; face < frequency.length; face++ )
20          System.out.printf( "%4d%10d\n", face, frequency[ face ] );
21    } // end main
22 } // end class RollDie
```

Fig. 7.7 | Die-rolling program using arrays instead of `switch`. (Part 1 of 2.)

```
Face Frequency
   1    999690
   2    999512
   3   1000575
   4    999815
   5    999781
   6   1000627
```

Fig. 7.7 | Die-rolling program using arrays instead of `switch`. (Part 2 of 2.)

- Figure 7.8 uses arrays to summarize the results of data collected in a survey:
  - *Twenty students were asked to rate on a scale of 1 to 5 the quality of the food in the student cafeteria, with 1 being "awful" and 5 being "excellent." Place the 20 responses in an integer array and determine the frequency of each rating.*
- Array responses is a 20-element int array of the survey responses.
- 6-element array frequency counts the number of occurrences of each response (1 to 5).
  - Each element is initialized to zero by default.
  - We ignore frequency[0].

```
1  // Fig. 7.8: StudentPoll.java
2  // Poll analysis program.
3
4  public class StudentPoll
5  {
6     public static void main( String[] args )
7     {
8        // student response array (more typically, input at runtime)
9        int[] responses = { 1, 2, 5, 4, 3, 5, 2, 1, 3, 3, 1, 4, 3, 3, 3,
10          2, 3, 3, 2, 14 };
11       int[] frequency = new int[ 6 ]; // array of frequency counters
12
13       // for each answer, select responses element and use that value
14       // as frequency index to determine element to increment
15       for ( int answer = 0; answer < responses.length; answer++ )
16       {
17          try
18          {
19             ++frequency[ responses[ answer ] ];
20          } // end try
21          catch ( ArrayIndexOutOfBoundsException e )
```

Fig. 7.8 | Poll analysis program. (Part 1 of 3.)

```
22          {
23             System.out.println( e );
24             System.out.printf( "   responses[%d] = %d\n\n",
25                answer, responses[ answer ] );
26          } // end catch
27       } // end for
28
29       System.out.printf( "%s%10s\n", "Rating", "Frequency" );
30
31       // output each array element's value
32       for ( int rating = 1; rating < frequency.length; rating++ )
33          System.out.printf( "%6d%10d\n", rating, frequency[ rating ] );
34    } // end main
35 } // end class StudentPoll
```

Fig. 7.8 | Poll analysis program. (Part 2 of 3.)

```
java.lang.ArrayIndexOutOfBoundsException: 14
   responses[19] = 14

Rating Frequency
     1         3
     2         4
     3         8
     4         2
     5         2
```

Fig. 7.8 | Poll analysis program. (Part 3 of 3.)

- If a piece of data in the responses array is an invalid value, such as 14, the program attempts to add 1 to frequency[14], which is outside the bounds of the array.
  - Java doesn't allow this.
  - JVM checks array indices to ensure that they are greater than or equal to 0 and less than the length of the array—this is called bounds checking.
  - If a program uses an invalid index, Java generates a so-called exception to indicate that an error occurred in the program at execution time.
- An exception indicates a problem that occurs while a program executes.
- The name "exception" suggests that the problem occurs infrequently—if the "rule" is that a statement normally executes correctly, then the problem represents the "exception to the rule."
- Exception handling enables you to create fault-tolerant programs that can resolve (or handle) exceptions.
- When the JVM or a method detects a problem, such as an invalid array index or an invalid method argument, it throws an exception—that is, an exception occurs.
- **The try Statement**
  - To handle an exception, place any code that might throw an exception in a try statement.
  - The try block contains the code that might throw an exception.
  - The catch block contains the code that handles the exception if one occurs. You can have many catch blocks to handle different types of exceptions that might be thrown in the corresponding try block.
- **Executing the catch Block**

- When the program encounters the value 14 in the responses array, it attempts to add 1 to frequency[14], which is outside the bounds of the array—the frequency array has only six elements.
- Because array bounds checking is performed at execution time, the JVM generates an exception—specifically line 19 throws an ArrayIndexOutOfBoundsException to notify the program of this problem.
- At this point the try block terminates and the catch block begins executing—if you declared any variables in the try block, they're now out of scope.

- The catch block declares a type and an exception parameter, and can handle exceptions of the specified type.
- Inside the catch block, you can use the parameter's identifier to interact with a caught exception object.
- The exception object's toString method returns the error message that is stored in the exception object.
- The exception is considered handled when program control reaches the closing right brace of the catch block.

3.1.2 Case Study: Card Shuffling and Dealing Simulation

- Examples thus far used arrays containing elements of primitive types.
- Elements of an array can be either primitive types or reference types.
- Next example uses an array of reference-type elements—objects representing playing cards—to develop a class that simulates card shuffling and dealing.
- Class Card (Fig. 7.9) contains two String instance variables—face and suit—that are used to store references to the face and suit names for a specific Card.
- Method toString creates a String consisting of the face of the card, " of " and the suit of the card.
  - Can invoke explicitly to obtain a string representation of a Card.
  - Called implicitly when the object is used where a String is expected.

```
1  // Fig. 7.9: Card.java
2  // Card class represents a playing card.
3
4  public class Card
5  {
6     private String face; // face of card ("Ace", "Deuce", ...)
7     private String suit; // suit of card ("Hearts", "Diamonds", ...)
8
9     // two-argument constructor initializes card's face and suit
10    public Card( String cardFace, String cardSuit )
11    {
12       face = cardFace; // initialize face of card
13       suit = cardSuit; // initialize suit of card
14    } // end two-argument Card constructor
15
16    // return String representation of Card
17    public String toString()
18    {
19       return face + " of " + suit;
20    } // end method toString
21 } // end class Card
```

Must be declared with this first line it is to be called implicitly to convert `Card` objects to `String` representations

Fig. 7.9 | `Card` class represents a playing card.

- Class DeckOfCards (Fig. 7.10) declares as an instance variable a Card array named deck.
- Deck's elements are null by default
  - Constructor fills the deck array with Card objects.

- Method shuffle shuffles the Cards in the deck.
  - Loops through all 52 Cards (array indices 0 to 51).
  - Each Card swapped with a randomly chosen other card in the deck.
- Method dealCard deals one Card in the array.
  - currentCard indicates the index of the next Card to be dealt
- Returns null if there are no more cards to deal

```
1  // Fig. 7.10: DeckOfCards.java
2  // DeckOfCards class represents a deck of playing cards.
3  import java.util.Random;
4
5  public class DeckOfCards
6  {
7     private Card[] deck; // array of Card objects
8     private int currentCard; // index of next Card to be dealt (0-51)
9     private static final int NUMBER_OF_CARDS = 52; // constant # of Cards
10    // random number generator
11    private static final Random randomNumbers = new Random();
12
13    // constructor fills deck of Cards
14    public DeckOfCards()
15    {
16       String[] faces = { "Ace", "Deuce", "Three", "Four", "Five", "Six",
17          "Seven", "Eight", "Nine", "Ten", "Jack", "Queen", "King" };
18       String[] suits = { "Hearts", "Diamonds", "Clubs", "Spades" };
19
20       deck = new Card[ NUMBER_OF_CARDS ]; // create array of Card objects
21       currentCard = 0; // set currentCard so first Card dealt is deck[ 0 ]
22
```

Fig. 7.10 | DeckOfCards class represents a deck of playing cards. (Part 1 of 3.)

```
23       // populate deck with Card objects
24       for ( int count = 0; count < deck.length; count++ )
25          deck[ count ] =
26             new Card( faces[ count % 13 ], suits[ count / 13 ] );
27    } // end DeckOfCards constructor
28
29    // shuffle deck of Cards with one-pass algorithm
30    public void shuffle()
31    {
32       // after shuffling, dealing should start at deck[ 0 ] again
33       currentCard = 0; // reinitialize currentCard
34
35       // for each Card, pick another random Card (0-51) and swap them
36       for ( int first = 0; first < deck.length; first++ )
37       {
38          // select a random number between 0 and 51
39          int second = randomNumbers.nextInt( NUMBER_OF_CARDS );
40
41          // swap current Card with randomly selected Card
42          Card temp = deck[ first ];
43          deck[ first ] = deck[ second ];
44          deck[ second ] = temp;
45       } // end for
46    } // end method shuffle
```

Fig. 7.10 | DeckOfCards class represents a deck of playing cards. (Part 2 of 3.)

```
47
48     // deal one Card
49     public Card dealCard()
50     {
51        // determine whether Cards remain to be dealt
52        if ( currentCard < deck.length )
53           return deck[ currentCard++ ]; // return current Card in array
54        else
55           return null; // return null to indicate that all Cards were dealt
56     } // end method dealCard
57  } // end class DeckOfCards
```

Fig. 7.10 | `DeckOfCards` class represents a deck of playing cards. (Part 3 of 3.)

- Figure 7.11 demonstrates class DeckOfCards (Fig. 7.10).
- When a Card is output as a String, the Card's toString method is implicitly invoked.

```
1  // Fig. 7.11: DeckOfCardsTest.java
2  // Card shuffling and dealing.
3
4  public class DeckOfCardsTest
5  {
6     // execute application
7     public static void main( String[] args )
8     {
9        DeckOfCards myDeckOfCards = new DeckOfCards();
10       myDeckOfCards.shuffle(); // place Cards in random order
11
12       // print all 52 Cards in the order in which they are dealt
13       for ( int i = 1; i <= 52; i++ )
14       {
15          // deal and display a Card
16          System.out.printf( "%-19s", myDeckOfCards.dealCard() );
17
18          if ( i % 4 == 0 ) // output a newline after every fourth card
19             System.out.println();
20       } // end for
21    } // end main
22 } // end class DeckOfCardsTest
```

Fig. 7.11 | Card shuffling and dealing. (Part 1 of 2.)

```
Six of Spades       Eight of Spades     Six of Clubs        Nine of Hearts
Queen of Hearts     Seven of Clubs      Nine of Spades      King of Hearts
Three of Diamonds   Deuce of Clubs      Ace of Hearts       Ten of Spades
Four of Spades      Ace of Clubs        Seven of Diamonds   Four of Hearts
Three of Clubs      Deuce of Hearts     Five of Spades      Jack of Diamonds
King of Clubs       Ten of Hearts       Three of Hearts     Six of Diamonds
Queen of Clubs      Eight of Diamonds   Deuce of Diamonds   Ten of Diamonds
Three of Spades     King of Diamonds    Nine of Clubs       Six of Hearts
Ace of Spades       Four of Diamonds    Seven of Hearts     Eight of Clubs
Deuce of Spades     Eight of Hearts     Five of Hearts      Queen of Spades
Jack of Hearts      Seven of Spades     Four of Clubs       Nine of Diamonds
Ace of Diamonds     Queen of Diamonds   Five of Clubs       King of Spades
Five of Diamonds    Ten of Clubs        Jack of Spades      Jack of Clubs
```

Fig. 7.11 | Card shuffling and dealing. (Part 2 of 2.)

3.2 Enhanced for Statement

- Enhanced for statement
  - Iterates through the elements of an array without using a counter.
  - Avoids the possibility of "stepping outside" the array.
  - Also works with the Java API's prebuilt collections (see Section 7.14).
- Syntax:
  for (*parameter : arrayName*)
  statement
  where *parameter* has a type and an identifier and *arrayName* is the array through which to iterate.
- Parameter type must be consistent with the array's element type.
- The enhanced for statement simplifies the code for iterating through an array.
- The enhanced for statement can be used only to obtain array elements
- It cannot be used to modify elements.
- To modify elements, use the traditional counter-controlled for statement.
- Can be used in place of the counter-controlled for statement if you don't need to access the index of the element.
-

```
1  // Fig. 7.12: EnhancedForTest.java
2  // Using the enhanced for statement to total integers in an array.
3
4  public class EnhancedForTest
5  {
6     public static void main( String[] args )
7     {
8        int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
9        int total = 0;
10
11       // add each element's value to total
12       for ( int number : array )
13          total += number;
14
15       System.out.printf( "Total of array elements: %d\n", total );
16    } // end main
17 } // end class EnhancedForTest
```

```
Total of array elements: 849
```

Fig. 7.12 | Using the enhanced for statement to total integers in an array.

3.4 Passing Arrays to Methods

- To pass an array argument to a method, specify the name of the array without any brackets.
  - Since every array object "knows" its own length, we need not pass the array length as an additional argument.
- To receive an array, the method's parameter list must specify an array parameter.
- When an argument to a method is an entire array or an individual array element of a reference type, the called method receives a copy of the reference.

- When an argument to a method is an individual array element of a primitive type, the called method receives a copy of the element's value.
  - Such primitive values are called scalars or scalar quantities.

```
1  // Fig. 7.13: PassArray.java
2  // Passing arrays and individual array elements to methods.
3
4  public class PassArray
5  {
6     // main creates array and calls modifyArray and modifyElement
7     public static void main( String[] args )
8     {
9        int[] array = { 1, 2, 3, 4, 5 };
10
11       System.out.println(
12          "Effects of passing reference to entire array:\n" +
13          "The values of the original array are:" );
14
15       // output original array elements
16       for ( int value : array )
17          System.out.printf( "   %d", value );
18
19       modifyArray( array ); // pass array reference
20       System.out.println( "\n\nThe values of the modified array are:" );
21
22       // output modified array elements
23       for ( int value : array )
24          System.out.printf( "   %d", value );
```

Passes the reference to `array` into method `modifyArray`

Fig. 7.13 | Passing arrays and individual array elements to methods. (Part 1 of 3.)

```
25
26       System.out.printf(
27          "\n\nEffects of passing array element value:\n" +
28          "array[3] before modifyElement: %d\n", array[ 3 ] );
29
30       modifyElement( array[ 3 ] ); // attempt to modify array[ 3 ]
31       System.out.printf(
32          "array[3] after modifyElement: %d\n", array[ 3 ] );
33    } // end main
34
35    // multiply each element of an array by 2
36    public static void modifyArray( int[] array2 )
37    {
38       for ( int counter = 0; counter < array2.length; counter++ )
39          array2[ counter ] *= 2;
40    } // end method modifyArray
41
42    // multiply argument by 2
43    public static void modifyElement( int element )
44    {
45       element *= 2;
46       System.out.printf(
47          "Value of element in modifyElement: %d\n", element );
48    } // end method modifyElement
49 } // end class PassArray
```

Passes a copy of `array[3]`'s `int` value into `modifyElement`

Method receives copy of an array's reference, which gives the method direct access to the original array in memory

Method receives copy of an `int` value; the method cannot modify the original `int` value in `main`

Fig. 7.13 | Passing arrays and individual array elements to methods. (Part 2 of 3.)

```
Effects of passing reference to entire array:
The values of the original array are:
   1   2   3   4   5

The values of the modified array are:
   2   4   6   8   10

Effects of passing array element value:
array[3] before modifyElement: 8
Value of element in modifyElement: 16
array[3] after modifyElement: 8
```

Fig. 7.13 | Passing arrays and individual array elements to methods. (Part 3 of 3.)

- Pass-by-value (also called call-by-value)
  - A copy of the argument's *value is passed to the called method.*
  - The called method works exclusively with the copy.
  - Changes to the called method's copy do not affect the original variable's value in the caller.
- Pass-by-reference (also called call-by-reference)
  - The called method can access the argument's value in the caller directly and modify that data, if necessary.
  - Improves performance by eliminating the need to copy possibly large amounts of data.
- All arguments in Java are passed by value.
- A method call can pass two types of values to a method
  - Copies of primitive values
  - Copies of references to objects
- Objects cannot be passed to methods.
- If a method modifies a reference-type parameter so that it refers to another object, only the parameter refers to the new object
  - The reference stored in the caller's variable still refers to the original object.
- Although an object's reference is passed by value, a method can still interact with the referenced object by calling its public methods using the copy of the object's reference.
  - The parameter in the called method and the argument in the calling method refer to the same object in memory.

3.4.1 Case Study: Class GradeBook Using an Array to Store Grades

- Previous versions of class GradeBook process a set of grades entered by the user, but do not maintain the individual grade values in instance variables of the class.
  - Repeat calculations require the user to reenter the same grades.
  - We solve this problem by storing grades in an array.
- The grades array's size is determined by the length of the array that is passed to the constructor.
  - So a GradeBook object can process a variable number of grades.

```
1  // Fig. 7.14: GradeBook.java
2  // GradeBook class using an array to store test grades.
3
4  public class GradeBook
5  {
6     private String courseName; // name of course this GradeBook represents
7     private int[] grades; // array of student grades
8
9     // two-argument constructor initializes courseName and grades array
10    public GradeBook( String name, int[] gradesArray )
11    {
12       courseName = name; // initialize courseName
13       grades = gradesArray; // store grades
14    } // end two-argument GradeBook constructor
15
16    // method to set the course name
17    public void setCourseName( String name )
18    {
19       courseName = name; // store the course name
20    } // end method setCourseName
21
```

Will refer to an array passed by the creator of the GradeBook

Receives the array from the GradeBook creator

Initializes the grades instance variable

Fig. 7.14 | GradeBook class using an array to store test grades. (Part 1 of 7.)

```
22    // method to retrieve the course name
23    public String getCourseName()
24    {
25       return courseName;
26    } // end method getCourseName
27
28    // display a welcome message to the GradeBook user
29    public void displayMessage()
30    {
31       // getCourseName gets the name of the course
32       System.out.printf( "Welcome to the grade book for\n%s!\n\n",
33          getCourseName() );
34    } // end method displayMessage
35
36    // perform various operations on the data
37    public void processGrades()
38    {
39       // output grades array
40       outputGrades();
41
42       // call method getAverage to calculate the average grade
43       System.out.printf( "\nClass average is %.2f\n", getAverage() );
44
```

Fig. 7.14 | GradeBook class using an array to store test grades. (Part 2 of 7.)

```
45          // call methods getMinimum and getMaximum
46          System.out.printf( "Lowest grade is %d\nHighest grade is %d\n\n",
47             getMinimum(), getMaximum() );
48
49          // call outputBarChart to print grade distribution chart
50          outputBarChart();
51       } // end method processGrades
52
53       // find minimum grade
54       public int getMinimum()
55       {
56          int lowGrade = grades[ 0 ]; // assume grades[ 0 ] is smallest
57
58          // loop through grades array
59          for ( int grade : grades )
60          {
61             // if grade lower than lowGrade, assign it to lowGrade
62             if ( grade < lowGrade )
63                lowGrade = grade; // new lowest grade
64          } // end for
65
66          return lowGrade; // return lowest grade
67       } // end method getMinimum
68
```

Fig. 7.14 | GradeBook class using an array to store test grades. (Part 3 of 7.)

```
69       // find maximum grade
70       public int getMaximum()
71       {
72          int highGrade = grades[ 0 ]; // assume grades[ 0 ] is largest
73
74          // loop through grades array
75          for ( int grade : grades )
76          {
77             // if grade greater than highGrade, assign it to highGrade
78             if ( grade > highGrade )
79                highGrade = grade; // new highest grade
80          } // end for
81
82          return highGrade; // return highest grade
83       } // end method getMaximum
84
```

Fig. 7.14 | GradeBook class using an array to store test grades. (Part 4 of 7.)

```
85     // determine average grade for test
86     public double getAverage()
87     {
88        int total = 0; // initialize total
89
90        // sum grades for one student
91        for ( int grade : grades )
92           total += grade;
93
94        // return average of grades
95        return (double) total / grades.length;
96     } // end method getAverage
97
98     // output bar chart displaying grade distribution
99     public void outputBarChart()
100    {
101       System.out.println( "Grade distribution:" );
102
103       // stores frequency of grades in each range of 10 grades
104       int[] frequency = new int[ 11 ];
105
106       // for each grade, increment the appropriate frequency
107       for ( int grade : grades )
108          ++frequency[ grade / 10 ];
```

Calculation is based on the length of the array used to initialize the GradeBook

Fig. 7.14 | GradeBook class using an array to store test grades. (Part 5 of 7.)

```
109
110       // for each grade frequency, print bar in chart
111       for ( int count = 0; count < frequency.length; count++ )
112       {
113          // output bar label ( "00-09: ", ..., "90-99: ", "100: " )
114          if ( count == 10 )
115             System.out.printf( "%5d: ", 100 );
116          else
117             System.out.printf( "%02d-%02d: ",
118                count * 10, count * 10 + 9  );
119
120          // print bar of asterisks
121          for ( int stars = 0; stars < frequency[ count ]; stars++ )
122             System.out.print( "*" );
123
124          System.out.println(); // start a new line of output
125       } // end outer for
126    } // end method outputBarChart
127
```

Fig. 7.14 | GradeBook class using an array to store test grades. (Part 6 of 7.)

- The application of Fig. 7.15 creates an object of class GradeBook (Fig. 7.14) using the int array grades-Array.
- Lines 12–13 pass a course name and gradesArray to the GradeBook constructor.

```
1  // Fig. 7.15: GradeBookTest.java
2  // GradeBookTest creates a GradeBook object using an array of grades,
3  // then invokes method processGrades to analyze them.
4  public class GradeBookTest
5  {
6     // main method begins program execution
7     public static void main( String[] args )
8     {
9        // array of student grades
10       int[] gradesArray = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
11
12       GradeBook myGradeBook = new GradeBook(
13          "CS101 Introduction to Java Programming", gradesArray );
14       myGradeBook.displayMessage();
15       myGradeBook.processGrades();
16    } // end main
17 } // end class GradeBookTest
```

Fig. 7.15 | `GradeBookTest` creates a `GradeBook` object using an array of grades, then invokes method `processGrades` to analyze them. (Part 1 of 3.)

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

The grades are:

Student  1:  87
Student  2:  68
Student  3:  94
Student  4: 100
Student  5:  83
Student  6:  78
Student  7:  85
Student  8:  91
Student  9:  76
Student 10:  87

Class average is 84.90
Lowest grade is 68
Highest grade is 100
```

Fig. 7.15 | `GradeBookTest` creates a `GradeBook` object using an array of grades, then invokes method `processGrades` to analyze them. (Part 2 of 3.)

```
Grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

Fig. 7.15 | `GradeBookTest` creates a `GradeBook` object using an array of grades, then invokes method `processGrades` to analyze them. (Part 3 of 3.)

3.5 Multidimensional Arrays

- Two-dimensional arrays are often used to represent tables of values consisting of information arranged in rows and columns.
- Identify a particular table element with two indices.
  - By convention, the first identifies the element's row and the second its column.
- Multidimensional arrays can have more than two dimensions.
- Java does not support multidimensional arrays directly
  - Allows you to specify one-dimensional arrays whose elements are also one-dimensional arrays, thus achieving the same effect.
- In general, an array with *m* rows and *n* columns is called an *m*-by-*n* array.

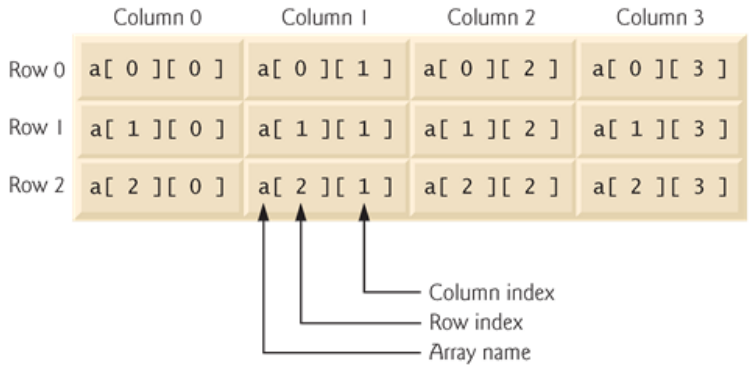

Fig. 7.16 | Two-dimensional array with three rows and four columns.

- Multidimensional arrays can be initialized with array initializers in declarations.
- A two-dimensional array b with two rows and two columns could be declared and initialized with nested array initializers as follows:
  `int[][] b = { { 1, 2 }, { 3, 4 } };`
  - The initial values are grouped by row in braces.
  - The number of nested array initializers (represented by sets of braces within the outer braces) determines the number of rows.

- The number of initializer values in the nested array initializer for a row determines the number of columns in that row.
- Rows can have different lengths.
- The lengths of the rows in a two-dimensional array are not required to be the same:
  int[][] b = { { 1, 2 }, { 3, 4, 5 } };
- Each element of b is a reference to a one-dimensional array of int variables.
- The int array for row 0 is a one-dimensional array with two elements (1 and 2).
- The int array for row 1 is a one-dimensional array with three elements (3, 4 and 5).
- Figure 7.17 demonstrates initializing two-dimensional arrays with array initializers and using nested for loops to traverse the arrays.

```
1  // Fig. 7.17: InitArray.java
2  // Initializing two-dimensional arrays.
3
4  public class InitArray
5  {
6     // create and output two-dimensional arrays
7     public static void main( String[] args )
8     {
9        int[][] array1 = { { 1, 2, 3 }, { 4, 5, 6 } };
10       int[][] array2 = { { 1, 2 }, { 3 }, { 4, 5, 6 } };
11
12       System.out.println( "Values in array1 by row are" );
13       outputArray( array1 ); // displays array1 by row
14
15       System.out.println( "\nValues in array2 by row are" );
16       outputArray( array2 ); // displays array2 by row
17    } // end main
18
19    // output rows and columns of a two-dimensional array
20    public static void outputArray( int[][] array )
21    {
22       // loop through array's rows
23       for ( int row = 0; row < array.length; row++ )
24       {
```

Fig. 7.17 | Initializing two-dimensional arrays. (Part 1 of 2.)

```
25          // loop through columns of current row
26          for ( int column = 0; column < array[ row ].length; column++ )
27             System.out.printf( "%d  ", array[ row ][ column ] );
28
29          System.out.println(); // start new line of output
30       } // end outer for
31    } // end method outputArray
32 } // end class InitArray
```

```
Values in array1 by row are
1  2  3
4  5  6

Values in array2 by row are
1  2
3
4  5  6
```

Fig. 7.17 | Initializing two-dimensional arrays. (Part 2 of 2.)

3.5.1 Case Study: Class GradeBook Using a Two-Dimensional Array

- In most semesters, students take several exams.
- Figure 7.18 contains a version of class GradeBook that uses a two-dimensional array grades to store the grades of a number of students on multiple exams.
  - Each row represents a student's grades for the entire course.
  - Each column represents the grades of all the students who took a particular exam.
- In this example, we use a ten-by-three array containing ten students' grades on three exams.

```
1  // Fig. 7.18: GradeBook.java
2  // GradeBook class using a two-dimensional array to store grades.
3
4  public class GradeBook
5  {
6     private String courseName; // name of course this grade book represents
7     private int[][] grades; // two-dimensional array of student grades
8
9     // two-argument constructor initializes courseName and grades array
10    public GradeBook( String name, int[][] gradesArray )
11    {
12       courseName = name; // initialize courseName
13       grades = gradesArray; // store grades
14    } // end two-argument GradeBook constructor
15
16    // method to set the course name
17    public void setCourseName( String name )
18    {
19       courseName = name; // store the course name
20    } // end method setCourseName
21
```

Fig. 7.18 | GradeBook class using a two-dimensional array to store grades. (Part 1 of 9.)

```
22     // method to retrieve the course name
23     public String getCourseName()
24     {
25        return courseName;
26     } // end method getCourseName
27
28     // display a welcome message to the GradeBook user
29     public void displayMessage()
30     {
31        // getCourseName gets the name of the course
32        System.out.printf( "Welcome to the grade book for\n%s!\n\n",
33           getCourseName() );
34     } // end method displayMessage
35
```

Fig. 7.18 | GradeBook class using a two-dimensional array to store grades. (Part 2 of 9.)

```
36     // perform various operations on the data
37     public void processGrades()
38     {
39        // output grades array
40        outputGrades();
41
42        // call methods getMinimum and getMaximum
43        System.out.printf( "\n%s %d\n%s %d\n\n",
44           "Lowest grade in the grade book is", getMinimum(),
45           "Highest grade in the grade book is", getMaximum() );
46
47        // output grade distribution chart of all grades on all tests
48        outputBarChart();
49     } // end method processGrades
50
```

Fig. 7.18 | GradeBook class using a two-dimensional array to store grades. (Part 3 of 9.)

```
51     // find minimum grade
52     public int getMinimum()
53     {
54        // assume first element of grades array is smallest
55        int lowGrade = grades[ 0 ][ 0 ];
56
57        // loop through rows of grades array
58        for ( int[] studentGrades : grades )
59        {
60           // loop through columns of current row
61           for ( int grade : studentGrades )
62           {
63              // if grade less than lowGrade, assign it to lowGrade
64              if ( grade < lowGrade )
65                 lowGrade = grade;
66           } // end inner for
67        } // end outer for
68
69        return lowGrade; // return lowest grade
70     } // end method getMinimum
71
```

Fig. 7.18 | GradeBook class using a two-dimensional array to store grades. (Part 4 of 9.)

```
72      // find maximum grade
73      public int getMaximum()
74      {
75         // assume first element of grades array is largest
76         int highGrade = grades[ 0 ][ 0 ];
77
78         // loop through rows of grades array
79         for ( int[] studentGrades : grades )
80         {
81            // loop through columns of current row
82            for ( int grade : studentGrades )
83            {
84               // if grade greater than highGrade, assign it to highGrade
85               if ( grade > highGrade )
86                  highGrade = grade;
87            } // end inner for
88         } // end outer for
89
90         return highGrade; // return highest grade
91      } // end method getMaximum
92
```

Fig. 7.18 | GradeBook class using a two-dimensional array to store grades. (Part 5 of 9.)

```
93      // determine average grade for particular set of grades
94      public double getAverage( int[] setOfGrades )
95      {
96         int total = 0; // initialize total
97
98         // sum grades for one student
99         for ( int grade : setOfGrades )
100           total += grade;
101
102        // return average of grades
103        return (double) total / setOfGrades.length;
104     } // end method getAverage
105
106     // output bar chart displaying overall grade distribution
107     public void outputBarChart()
108     {
109        System.out.println( "Overall grade distribution:" );
110
111        // stores frequency of grades in each range of 10 grades
112        int[] frequency = new int[ 11 ];
113
```

Fig. 7.18 | GradeBook class using a two-dimensional array to store grades. (Part 6 of 9.)

```
114         // for each grade in GradeBook, increment the appropriate frequency
115         for ( int[] studentGrades : grades )
116         {
117            for ( int grade : studentGrades )
118               ++frequency[ grade / 10 ];
119         } // end outer for
120
121         // for each grade frequency, print bar in chart
122         for ( int count = 0; count < frequency.length; count++ )
123         {
124            // output bar label ( "00-09: ", ..., "90-99: ", "100: " )
125            if ( count == 10 )
126               System.out.printf( "%5d: ", 100 );
127            else
128               System.out.printf( "%02d-%02d: ",
129                  count * 10, count * 10 + 9  );
130
131            // print bar of asterisks
132            for ( int stars = 0; stars < frequency[ count ]; stars++ )
133               System.out.print( "*" );
134
135            System.out.println(); // start a new line of output
136         } // end outer for
137      } // end method outputBarChart
```

Fig. 7.18 | GradeBook class using a two-dimensional array to store grades. (Part 7 of 9.)

```
138
139      // output the contents of the grades array
140      public void outputGrades()
141      {
142         System.out.println( "The grades are:\n" );
143         System.out.print( "            " ); // align column heads
144
145         // create a column heading for each of the tests
146         for ( int test = 0; test < grades[ 0 ].length; test++ )
147            System.out.printf( "Test %d  ", test + 1 );
148
149         System.out.println( "Average" ); // student average column heading
150
151         // create rows/columns of text representing array grades
152         for ( int student = 0; student < grades.length; student++ )
153         {
154            System.out.printf( "Student %2d", student + 1 );
155
156            for ( int test : grades[ student ] ) // output student's grades
157               System.out.printf( "%8d", test );
158
```

Fig. 7.18 | GradeBook class using a two-dimensional array to store grades. (Part 8 of 9.)

```
159          // call method getAverage to calculate student's average grade;
160          // pass row of grades as the argument to getAverage
161          double average = getAverage( grades[ student ] );
162          System.out.printf( "%9.2f\n", average );
163       } // end outer for
164    } // end method outputGrades
165 } // end class GradeBook
```

Fig. 7.18 | GradeBook class using a two-dimensional array to store grades. (Part 9 of 9.)

```
1  // Fig. 7.19: GradeBookTest.java
2  // GradeBookTest creates GradeBook object using a two-dimensional array
3  // of grades, then invokes method processGrades to analyze them.
4  public class GradeBookTest
5  {
6     // main method begins program execution
7     public static void main( String[] args )
8     {
9        // two-dimensional array of student grades
10       int[][] gradesArray = { { 87, 96, 70 },
11                               { 68, 87, 90 },
12                               { 94, 100, 90 },
13                               { 100, 81, 82 },
14                               { 83, 65, 85 },
15                               { 78, 87, 65 },
16                               { 85, 75, 83 },
17                               { 91, 94, 100 },
18                               { 76, 72, 84 },
19                               { 87, 93, 73 } };
```

Fig. 7.19 | GradeBookTest creates GradeBook object using a two-dimensional array of grades, then invokes method processGrades to analyze them. (Part 1 of 4.)

```
20       GradeBook myGradeBook = new GradeBook(
21          "CS101 Introduction to Java Programming", gradesArray );
22       myGradeBook.displayMessage();
23       myGradeBook.processGrades();
24    } // end main
25 } // end class GradeBookTest
```

Fig. 7.19 | GradeBookTest creates GradeBook object using a two-dimensional array of grades, then invokes method processGrades to analyze them. (Part 2 of 4.)

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

The grades are:

            Test 1  Test 2  Test 3  Average
Student  1      87      96      70    84.33
Student  2      68      87      90    81.67
Student  3      94     100      90    94.67
Student  4     100      81      82    87.67
Student  5      83      65      85    77.67
Student  6      78      87      65    76.67
Student  7      85      75      83    81.00
Student  8      91      94     100    95.00
Student  9      76      72      84    77.33
Student 10      87      93      73    84.33

Lowest grade in the grade book is 65
Highest grade in the grade book is 100
```

Fig. 7.19 | `GradeBookTest` creates `GradeBook` object using a two-dimensional array of grades, then invokes method `processGrades` to analyze them. (Part 3 of 4.)

```
Overall grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: ***
70-79: ******
80-89: ***********
90-99: *******
  100: ***
```

Fig. 7.19 | `GradeBookTest` creates `GradeBook` object using a two-dimensional array of grades, then invokes method `processGrades` to analyze them. (Part 4 of 4.)

3.6 Variable-Length Argument Lists

- Variable-length argument lists
  - Can be used to create methods that receive an unspecified number of arguments.
  - Parameter type followed by an ellipsis (...) indicates that the method receives a variable num-ber of arguments of that particular type.
  - The ellipsis can occur only once at the end of a parameter list.

```
1  // Fig. 7.20: VarargsTest.java
2  // Using variable-length argument lists.
3
4  public class VarargsTest
5  {
6     // calculate average
7     public static double average( double... numbers )
8     {
9        double total = 0.0; // initialize total
10
11       // calculate total using the enhanced for statement
12       for ( double d : numbers )
13          total += d;
14
15       return total / numbers.length;
16    } // end method average
17
18    public static void main( String[] args )
19    {
20       double d1 = 10.0;
21       double d2 = 20.0;
22       double d3 = 30.0;
23       double d4 = 40.0;
24
```

Fig. 7.20 | Using variable-length argument lists. (Part 1 of 2.)

```
25       System.out.printf( "d1 = %.1f\nd2 = %.1f\nd3 = %.1f\nd4 = %.1f\n\n",
26          d1, d2, d3, d4 );
27
28       System.out.printf( "Average of d1 and d2 is %.1f\n",
29          average( d1, d2 ) );
30       System.out.printf( "Average of d1, d2 and d3 is %.1f\n",
31          average( d1, d2, d3 ) );
32       System.out.printf( "Average of d1, d2, d3 and d4 is %.1f\n",
33          average( d1, d2, d3, d4 ) );
34    } // end main
35 } // end class VarargsTest
```

```
d1 = 10.0
d2 = 20.0
d3 = 30.0
d4 = 40.0

Average of d1 and d2 is 15.0
Average of d1, d2 and d3 is 20.0
Average of d1, d2, d3 and d4 is 25.0
```

Fig. 7.20 | Using variable-length argument lists. (Part 2 of 2.)

3.7 Using Command-Line Arguments

- Command-line arguments

- Can pass arguments from the command line to an application.
- Arguments that appear after the class name in the java command are received by main in the String array args.
- The number of command-line arguments is obtained by accessing the array's length attribute.
- Command-line arguments are separated by white space, not commas.

```
1  // Fig. 7.21: InitArray.java
2  // Initializing an array using command-line arguments.
3
4  public class InitArray
5  {
6     public static void main( String[] args )
7     {
8        // check number of command-line arguments
9        if ( args.length != 3 )
10          System.out.println(
11             "Error: Please re-enter the entire command, including\n" +
12             "an array size, initial value and increment." );
13       else
14       {
15          // get array size from first command-line argument
16          int arrayLength = Integer.parseInt( args[ 0 ] );
17          int[] array = new int[ arrayLength ]; // create array
18
19          // get initial value and increment from command-line arguments
20          int initialValue = Integer.parseInt( args[ 1 ] );
21          int increment = Integer.parseInt( args[ 2 ] );
22
```

Fig. 7.21 | Initializing an array using command-line arguments. (Part 1 of 3.)

```
23          // calculate value for each array element
24          for ( int counter = 0; counter < array.length; counter++ )
25             array[ counter ] = initialValue + increment * counter;
26
27          System.out.printf( "%s%8s\n", "Index", "Value" );
28
29          // display array index and value
30          for ( int counter = 0; counter < array.length; counter++ )
31             System.out.printf( "%5d%8d\n", counter, array[ counter ] );
32       } // end else
33    } // end main
34 } // end class InitArray
```

```
java InitArray
Error: Please re-enter the entire command, including
an array size, initial value and increment.
```

Fig. 7.21 | Initializing an array using command-line arguments. (Part 2 of 3.)

```
java InitArray 5 0 4
Index   Value
    0       0
    1       4
    2       8
    3      12
    4      16
```

```
java InitArray 8 1 2
Index   Value
    0       1
    1       3
    2       5
    3       7
    4       9
    5      11
    6      13
    7      15
```

Fig. 7.21 | Initializing an array using command-line arguments. (Part 3 of 3.)

3.8 Class Arrays

- Arrays class
  - Provides static methods for common array manipulations.
- Methods include
  - sort for sorting an array (ascending order by default)
  - binarySearch for searching a sorted array
  - equals for comparing arrays
  - fill for placing values into an array.
- Methods are overloaded for primitive-type arrays and for arrays of objects.
- System class static arraycopy method
  - Copies contents of one array into another.

```
1  // Fig. 7.22: ArrayManipulations.java
2  // Arrays class methods and System.arraycopy.
3  import java.util.Arrays;
4
5  public class ArrayManipulations
6  {
7     public static void main( String[] args )
8     {
9        // sort doubleArray into ascending order
10       double[] doubleArray = { 8.4, 9.3, 0.2, 7.9, 3.4 };
11       Arrays.sort( doubleArray );
12       System.out.printf( "\ndoubleArray: " );
13
14       for ( double value : doubleArray )
15          System.out.printf( "%.1f ", value );
16
17       // fill 10-element array with 7s
18       int[] filledIntArray = new int[ 10 ];
19       Arrays.fill( filledIntArray, 7 );
20       displayArray( filledIntArray, "filledIntArray" );
21
```

Fig. 7.22 | Arrays class methods. (Part 1 of 4.)

```
22       // copy array intArray into array intArrayCopy
23       int[] intArray = { 1, 2, 3, 4, 5, 6 };
24       int[] intArrayCopy = new int[ intArray.length ];
25       System.arraycopy( intArray, 0, intArrayCopy, 0, intArray.length );
26       displayArray( intArray, "intArray" );
27       displayArray( intArrayCopy, "intArrayCopy" );
28
29       // compare intArray and intArrayCopy for equality
30       boolean b = Arrays.equals( intArray, intArrayCopy );
31       System.out.printf( "\n\nintArray %s intArrayCopy\n",
32          ( b ? "==" : "!=" ) );
33
34       // compare intArray and filledIntArray for equality
35       b = Arrays.equals( intArray, filledIntArray );
36       System.out.printf( "intArray %s filledIntArray\n",
37          ( b ? "==" : "!=" ) );
38
39       // search intArray for the value 5
40       int location = Arrays.binarySearch( intArray, 5 );
41
42       if ( location >= 0 )
43          System.out.printf(
44             "Found 5 at element %d in intArray\n", location );
```

Fig. 7.22 | Arrays class methods. (Part 2 of 4.)

```
45         else
46            System.out.println( "5 not found in intArray" );
47
48         // search intArray for the value 8763
49         location = Arrays.binarySearch( intArray, 8763 );
50
51         if ( location >= 0 )
52            System.out.printf(
53               "Found 8763 at element %d in intArray\n", location );
54         else
55            System.out.println( "8763 not found in intArray" );
56      } // end main
57
58      // output values in each array
59      public static void displayArray( int[] array, String description )
60      {
61         System.out.printf( "\n%s: ", description );
62
63         for ( int value : array )
64            System.out.printf( "%d ", value );
65      } // end method displayArray
66   } // end class ArrayManipulations
```

Fig. 7.22 | Arrays class methods. (Part 3 of 4.)

```
doubleArray: 0.2 3.4 7.9 8.4 9.3
filledIntArray: 7 7 7 7 7 7 7 7 7 7
intArray: 1 2 3 4 5 6
intArrayCopy: 1 2 3 4 5 6

intArray == intArrayCopy
intArray != filledIntArray
Found 5 at element 4 in intArray
8763 not found in intArray
```

Fig. 7.22 | Arrays class methods. (Part 4 of 4.)

3.9 Introduction to Collections and Class ArrayList

- Java API provides several predefined data structures, called collections, used to store groups of related objects.
  - Each provides efficient methods that organize, store and retrieve your data without requiring knowledge of how the data is being stored.
  - Reduce application-development time.
- Arrays do not automatically change their size at execution time to accommodate additional elements.
- ArrayList<T> (package java.util) can dynamically change its size to accommodate more elements.
  - T is a placeholder for the type of element stored in the collection.
  - This is similar to specifying the type when declaring an array, except that only nonprimitive types can be used with these collection classes.
- Classes with this kind of placeholder that can be used with any type are called generic classes.

Method	Description
`add`	Adds an element to the end of the `ArrayList`.
`clear`	Removes all the elements from the `ArrayList`.
`contains`	Returns `true` if the `ArrayList` contains the specified element; otherwise, returns `false`.
`get`	Returns the element at the specified index.
`indexOf`	Returns the index of the first occurrence of the specified element in the `ArrayList`.
`remove`	Overloaded. Removes the first occurrence of the specified value or the element at the specified index.
`size`	Returns the number of elements stored in the `ArrayList`.
`trimToSize`	Trims the capacity of the `ArrayList` to current number of elements.

Fig. 7.23 | Some methods and properties of class `ArrayList<T>`.

- Figure 7.24 demonstrates some common ArrayList capabilities.
- An ArrayList's capacity indicates how many items it can hold without growing.
- When the ArrayList grows, it must create a larger internal array and copy each element to the new array.
  - This is a time-consuming operation. It would be inefficient for the ArrayList to grow each time an element is added.
  - An ArrayList grows only when an element is added and the number of elements is equal to the capacity—i.e., there is no space for the new element.
- Method add adds elements to the ArrayList.
  - One-argument version appends its argument to the end of the ArrayList.
  - Two-argument version inserts a new element at the specified position.
  - Collection indices start at zero.
- Method size returns the number of elements in the ArrayList.
- Method get obtains the element at a specified index.
- Method remove deletes an element with a specific value.
  - An overloaded version of the method removes the element at the specified index.
- Method contains determines if an item is in the ArrayList.

```
1  // Fig. 7.24: ArrayListCollection.java
2  // Generic ArrayList<T> collection demonstration.
3  import java.util.ArrayList;
4
5  public class ArrayListCollection
6  {
7     public static void main( String[] args )
8     {
9        // create a new ArrayList of Strings with an initial capacity of 10
10       ArrayList< String > items = new ArrayList< String >();
11
12       items.add( "red" ); // append an item to the list
13       items.add( 0, "yellow" ); // insert the value at index 0
14
15       // header
16       System.out.print(
17          "Display list contents with counter-controlled loop:" );
18
19       // display the colors in the list
20       for ( int i = 0; i < items.size(); i++ )
21          System.out.printf( " %s", items.get( i ) );
22
```

Fig. 7.24 | Generic `ArrayList<T>` collection demonstration. (Part 1 of 3.)

```
23       // display colors using foreach in the display method
24       display( items,
25          "\nDisplay list contents with enhanced for statement:" );
26
27       items.add( "green" ); // add "green" to the end of the list
28       items.add( "yellow" ); // add "yellow" to the end of the list
29       display( items, "List with two new elements:" );
30
31       items.remove( "yellow" ); // remove the first "yellow"
32       display( items, "Remove first instance of yellow:" );
33
34       items.remove( 1 ); // remove item at index 1
35       display( items, "Remove second list element (green):" );
36
37       // check if a value is in the List
38       System.out.printf( "\"red\" is %sin the list\n",
39          items.contains( "red" ) ? "": "not " );
40
41       // display number of elements in the List
42       System.out.printf( "Size: %s\n", items.size() );
43    } // end main
44
```

Fig. 7.24 | Generic `ArrayList<T>` collection demonstration. (Part 2 of 3.)

```
45      // display the ArrayList's elements on the console
46      public static void display( ArrayList< String > items, String header )
47      {
48         System.out.print( header ); // display header
49
50         // display each element in items
51         for ( String item : items )
52            System.out.printf( " %s", item );
53
54         System.out.println(); // display end of line
55      } // end method display
56   } // end class ArrayListCollection
```

```
Display list contents with counter-controlled loop: yellow red
Display list contents with enhanced for statement: yellow red
List with two new elements: yellow red green yellow
Remove first instance of yellow: red green yellow
Remove second list element (green): red yellow
"red" is in the list
Size: 2
```

Fig. 7.24 | Generic `ArrayList<T>` collection demonstration. (Part 3 of 3.)

4. Classes and Objects: A deeper Look

4.1 Time Class Case Study

```
1  // Fig. 8.1: Time1.java
2  // Time1 class declaration maintains the time in 24-hour format.
3
4  public class Time1
5  {
6     private int hour; // 0 - 23
7     private int minute; // 0 - 59
8     private int second; // 0 - 59
9
10    // set a new time value using universal time; throw an
11    // exception if the hour, minute or second is invalid
12    public void setTime( int h, int m, int s )
13    {
14       // validate hour, minute and second
15       if ( ( h >= 0 && h < 24 ) && ( m >= 0 && m < 60 ) &&
16          ( s >= 0 && s < 60 ) )
17       {
18          hour = h;
19          minute = m;
20          second = s;
21       } // end if
```

Fig. 8.1 | `Time1` class declaration maintains the time in 24-hour format. (Part 1 of 2.)

```
22       else
23          throw new IllegalArgumentException(
24             "hour, minute and/or second was out of range" );
25    } // end method setTime
26
27    // convert to String in universal-time format (HH:MM:SS)
28    public String toUniversalString()
29    {
30       return String.format( "%02d:%02d:%02d", hour, minute, second );
31    } // end method toUniversalString
32
33    // convert to String in standard-time format (H:MM:SS AM or PM)
34    public String toString()
35    {
36       return String.format( "%d:%02d:%02d %s",
37          ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
38          minute, second, ( hour < 12 ? "AM" : "PM" ) );
39    } // end method toString
40 } // end class Time1
```

Fig. 8.1 | `Time1` class declaration maintains the time in 24-hour format. (Part 2 of 2.)

```
1  // Fig. 8.2: Time1Test.java
2  // Time1 object used in an application.
3
4  public class Time1Test
5  {
6     public static void main( String[] args )
7     {
8        // create and initialize a Time1 object
9        Time1 time = new Time1(); // invokes Time1 constructor
10
11       // output string representations of the time
12       System.out.print( "The initial universal time is: " );
13       System.out.println( time.toUniversalString() );
14       System.out.print( "The initial standard time is: " );
15       System.out.println( time.toString() );
16       System.out.println(); // output a blank line
17
18       // change time and output updated time
19       time.setTime( 13, 27, 6 );
20       System.out.print( "Universal time after setTime is: " );
21       System.out.println( time.toUniversalString() );
22       System.out.print( "Standard time after setTime is: " );
23       System.out.println( time.toString() );
24       System.out.println(); // output a blank line
```

Fig. 8.2 | `Time1` object used in an application. (Part 1 of 3.)

```
25
26       // attempt to set time with invalid values
27       try
28       {
29          time.setTime( 99, 99, 99 ); // all values out of range
30       } // end try
31       catch ( IllegalArgumentException e )
32       {
33          System.out.printf( "Exception: %s\n\n", e.getMessage() );
34       } // end catch
35
36       // display time after attempt to set invalid values
37       System.out.println( "After attempting invalid settings:" );
38       System.out.print( "Universal time: " );
39       System.out.println( time.toUniversalString() );
40       System.out.print( "Standard time: " );
41       System.out.println( time.toString() );
42    } // end main
43 } // end class Time1Test
```

Fig. 8.2 | `Time1` object used in an application. (Part 2 of 3.)

```
The initial universal time is: 00:00:00
The initial standard time is: 12:00:00 AM

Universal time after setTime is: 13:27:06
Standard time after setTime is: 1:27:06 PM
```

```
Exception: hour, minute and/or second was out of range

After attempting invalid settings:
Universal time: 13:27:06
Standard time: 1:27:06 PM
```

Fig. 8.2 | `Time1` object used in an application. (Part 3 of 3.)

Discussion

- Class Time1 represents the time of day.
- private int instance variables hour, minute and second represent the time in universal-time format (24-hour clock format in which hours are in the range 0–23).
- public methods setTime, toUniversalString and toString are Clled the public services or the public interface that the class provides to its clients.
- Class Time1 does not declare a constructor, so the class has a default constructor that is supplied by the compiler.
- Each instance variable implicitly receives the default value 0 for an int.
- Instance variables also can be initialized when they are declared in the class body, using the same initialization syntax as with a local variable.
- Method setTime and Throwing Exceptions
  - Method setTime (lines 12–25) declares three int parameters and uses them to set the time.
  - Lines 15–16 test each argument to determine whether the value is in the proper range, and, if so, lines 18–20 assign the values to the hour, minute and second instance variables.
  - For incorrect values, setTime throws an exception of type IllegalArgumentException (lines 23–24)
    - Notifies the client code that an invalid argument was passed to the method.
    - Can use try...catch to catch exceptions and attempt to recover from them.
    - The throw statement (line 23) creates a new object of type IllegalArgumentException. In this case, we call the constructor that allows us to specify a custom error message.
    - After the exception object is created, the throw statement immediately terminates method setTime and the exception is returned to the code that attempted to set the time.
- The instance variables hour, minute and second are each declared private.
- The actual data representation used within the class is of no concern to the class's clients.
- Reasonable for Time1 to represent the time internally as the number of sec-onds since midnight or the number of minutes and seconds since midnight.
- Clients could use the same public methods and get the same results without being aware of this.
- Access modifiers public and private control access to a class's variables and methods.
  - Chapter 9 introduces access modifier protected.

- public methods present to the class's clients a view of the services the class provides (the class's public interface).
- Clients need not be concerned with how the class accomplishes its tasks.
  - For this reason, the class's private variables and private methods (i.e., its implementation details) are not accessible to its clients.
- private class members are not accessible outside the class.

```
1  // Fig. 8.3: MemberAccessTest.java
2  // Private members of class Time1 are not accessible.
3  public class MemberAccessTest
4  {
5     public static void main( String[] args )
6     {
7        Time1 time = new Time1(); // create and initialize Time1 object
8
9        time.hour = 7; // error: hour has private access in Time1
10       time.minute = 15; // error: minute has private access in Time1
11       time.second = 30; // error: second has private access in Time1
12    } // end main
13 } // end class MemberAccessTest
```

Each of these statements attempts to access data that is `private` to class `Time1`

Fig. 8.3 | Private members of class `Time1` are not accessible. (Part 1 of 2.)

```
MemberAccessTest.java:9: hour has private access in Time1
      time.hour = 7; // error: hour has private access in Time1
          ^
MemberAccessTest.java:10: minute has private access in Time1
      time.minute = 15; // error: minute has private access in Time1
          ^
MemberAccessTest.java:11: second has private access in Time1
      time.second = 30; // error: second has private access in Time1
          ^
3 errors
```

Fig. 8.3 | Private members of class `Time1` are not accessible. (Part 2 of 2.)

4.2 Referring to the Current Object's Members with the *this* Reference

- Every object can access a reference to itself with keyword this.
- When a non-static method is called for a particular object, the method's body implicitly uses keyword this to refer to the object's instance variables and other methods.
  - Enables the class's code to know which object should be manipulated.
  - Can also use keyword this explicitly in a non-static method's body.
- Can use the this reference implicitly and explicitly.
- When you compile a .java file containing more than one class, the compiler produces a separate class file with the .class extension for every compiled class.
- When one source-code (.java) file contains multiple class declarations, the compiler places both class files for those classes in the same directory.
- A source-code file can contain only one public class—otherwise, a compilation error occurs.
- Non-public classes can be used only by other classes in the same package.

```
1  // Fig. 8.4: ThisTest.java
2  // this used implicitly and explicitly to refer to members of an object.
3
4  public class ThisTest
5  {
6     public static void main( String[] args )
7     {
8        SimpleTime time = new SimpleTime( 15, 30, 19 );
9        System.out.println( time.buildString() );
10    } // end main
11 } // end class ThisTest
12
13 // class SimpleTime demonstrates the "this" reference
14 class SimpleTime
15 {
16    private int hour; // 0-23
17    private int minute; // 0-59
18    private int second; // 0-59
19
```

Fig. 8.4 | `this` used implicitly and explicitly to refer to members of an object. (Part 1 of 3.)

```
20    // if the constructor uses parameter names identical to
21    // instance variable names the "this" reference is
22    // required to distinguish between the names
23    public SimpleTime( int hour, int minute, int second )
24    {
25       this.hour = hour; // set "this" object's hour
26       this.minute = minute; // set "this" object's minute
27       this.second = second; // set "this" object's second
28    } // end SimpleTime constructor
29
30    // use explicit and implicit "this" to call toUniversalString
31    public String buildString()
32    {
33       return String.format( "%24s: %s\n%24s: %s",
34          "this.toUniversalString()", this.toUniversalString(),
35          "toUniversalString()", toUniversalString() );
36    } // end method buildString
37
```

Fig. 8.4 | `this` used implicitly and explicitly to refer to members of an object. (Part 2 of 3.)

```
38      // convert to String in universal-time format (HH:MM:SS)
39      public String toUniversalString()
40      {
41         // "this" is not required here to access instance variables,
42         // because method does not have local variables with same
43         // names as instance variables
44         return String.format( "%02d:%02d:%02d",
45            this.hour, this.minute, this.second );
46      } // end method toUniversalString
47   } // end class SimpleTime
```

```
this.toUniversalString(): 15:30:19
     toUniversalString(): 15:30:19
```

Fig. 8.4 | `this` used implicitly and explicitly to refer to members of an object. (Part 3 of 3.)

- SimpleTime declares three private instance variables—hour, minute and second.
- If parameter names for the constructor that are identical to the class's instance-variable names.
  - We don't recommend this practice
  - Use it here to shadow (hide) the corresponding instance
  - Illustrates a case in which explicit use of the this reference is required.
- If a method contains a local variable with the same name as a field, that method uses the local variable rather than the field.
  - The local variable *shadows* the field in the method's scope.
- A method can use the this reference to refer to the shadowed field explicitly.

4.3 Time Class Case Study: Overloaded Constructors

- Overloaded constructors enable objects of a class to be initialized in different ways.
- To overload constructors, simply provide multiple constructor declarations with different signatures.
- Recall that the compiler differentiates signatures by the *number* of parameters, the *types* of the parameters and the *order* of the parameter types in each signature.
- Class Time2 (Fig. 8.5) contains five overloaded constructors that provide convenient ways to initialize objects of the new class Time2.
- The compiler invokes the appropriate constructor by matching the number, types and order of the types of the arguments specified in the constructor call with the number, types and order of the types of the parameters specified in each constructor declaration.

```
1  // Fig. 8.5: Time2.java
2  // Time2 class declaration with overloaded constructors.
3
4  public class Time2
5  {
6     private int hour; // 0 - 23
7     private int minute; // 0 - 59
8     private int second; // 0 - 59
9
10    // Time2 no-argument constructor:
11    // initializes each instance variable to zero
12    public Time2()
13    {
14       this( 0, 0, 0 ); // invoke Time2 constructor with three arguments
15    } // end Time2 no-argument constructor
16
17    // Time2 constructor: hour supplied, minute and second defaulted to 0
18    public Time2( int h )
19    {
20       this( h, 0, 0 ); // invoke Time2 constructor with three arguments
21    } // end Time2 one-argument constructor
22
```

Fig. 8.5 | Time2 class with overloaded constructors. (Part 1 of 6.)

```
23    // Time2 constructor: hour and minute supplied, second defaulted to 0
24    public Time2( int h, int m )
25    {
26       this( h, m, 0 ); // invoke Time2 constructor with three arguments
27    } // end Time2 two-argument constructor
28
29    // Time2 constructor: hour, minute and second supplied
30    public Time2( int h, int m, int s )
31    {
32       setTime( h, m, s ); // invoke setTime to validate time
33    } // end Time2 three-argument constructor
34
35    // Time2 constructor: another Time2 object supplied
36    public Time2( Time2 time )
37    {
38       // invoke Time2 three-argument constructor
39       this( time.getHour(), time.getMinute(), time.getSecond() );
40    } // end Time2 constructor with a Time2 object argument
41
```

Fig. 8.5 | Time2 class with overloaded constructors. (Part 2 of 6.)

```
42      // Set Methods
43      // set a new time value using universal time;
44      // validate the data
45      public void setTime( int h, int m, int s )
46      {
47         setHour( h ); // set the hour
48         setMinute( m ); // set the minute
49         setSecond( s ); // set the second
50      } // end method setTime
51
52      // validate and set hour
53      public void setHour( int h )
54      {
55         if ( h >= 0 && h < 24 )
56            hour = h;
57         else
58            throw new IllegalArgumentException( "hour must be 0-23" );
59      } // end method setHour
60
```

Fig. 8.5 | `Time2` class with overloaded constructors. (Part 3 of 6.)

```
61      // validate and set minute
62      public void setMinute( int m )
63      {
64         if ( m >= 0 && m < 60 )
65            minute = m;
66         else
67            throw new IllegalArgumentException( "minute must be 0-59" );
68      } // end method setMinute
69
70      // validate and set second
71      public void setSecond( int s )
72      {
73         if ( s >= 0 && s < 60 )
74            second = ( ( s >= 0 && s < 60 ) ? s : 0 );
75         else
76            throw new IllegalArgumentException( "second must be 0-59" );
77      } // end method setSecond
78
79      // Get Methods
80      // get hour value
81      public int getHour()
82      {
83         return hour;
84      } // end method getHour
```

Fig. 8.5 | `Time2` class with overloaded constructors. (Part 4 of 6.)

```
85
86      // get minute value
87      public int getMinute()
88      {
89         return minute;
90      } // end method getMinute
91
92      // get second value
93      public int getSecond()
94      {
95         return second;
96      } // end method getSecond
97
98      // convert to String in universal-time format (HH:MM:SS)
99      public String toUniversalString()
100     {
101        return String.format(
102           "%02d:%02d:%02d", getHour(), getMinute(), getSecond() );
103     } // end method toUniversalString
104
```

Fig. 8.5 | `Time2` class with overloaded constructors. (Part 5 of 6.)

```
105     // convert to String in standard-time format (H:MM:SS AM or PM)
106     public String toString()
107     {
108        return String.format( "%d:%02d:%02d %s",
109           ( (getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12 ),
110           getMinute(), getSecond(), ( getHour() < 12 ? "AM" : "PM" ) );
111     } // end method toString
112  } // end class Time2
```

Fig. 8.5 | `Time2` class with overloaded constructors. (Part 6 of 6.)

```
1  // Fig. 8.6: Time2Test.java
2  // Overloaded constructors used to initialize Time2 objects.
3
4  public class Time2Test
5  {
6     public static void main( String[] args )
7     {
8        Time2 t1 = new Time2(); // 00:00:00
9        Time2 t2 = new Time2( 2 ); // 02:00:00
10       Time2 t3 = new Time2( 21, 34 ); // 21:34:00
11       Time2 t4 = new Time2( 12, 25, 42 ); // 12:25:42
12       Time2 t5 = new Time2( t4 ); // 12:25:42
13
14       System.out.println( "Constructed with:" );
15       System.out.println( "t1: all arguments defaulted" );
16       System.out.printf( "   %s\n", t1.toUniversalString() );
17       System.out.printf( "   %s\n", t1.toString() );
18
19       System.out.println(
20          "t2: hour specified; minute and second defaulted" );
21       System.out.printf( "   %s\n", t2.toUniversalString() );
22       System.out.printf( "   %s\n", t2.toString() );
23
```

Fig. 8.6 | Overloaded constructors used to initialize `Time2` objects. (Part 1 of 4.)

```
24       System.out.println(
25          "t3: hour and minute specified; second defaulted" );
26       System.out.printf( "   %s\n", t3.toUniversalString() );
27       System.out.printf( "   %s\n", t3.toString() );
28
29       System.out.println( "t4: hour, minute and second specified" );
30       System.out.printf( "   %s\n", t4.toUniversalString() );
31       System.out.printf( "   %s\n", t4.toString() );
32
33       System.out.println( "t5: Time2 object t4 specified" );
34       System.out.printf( "   %s\n", t5.toUniversalString() );
35       System.out.printf( "   %s\n", t5.toString() );
36
37       // attempt to initialize t6 with invalid values
38       try
39       {
40          Time2 t6 = new Time2( 27, 74, 99 ); // invalid values
41       } // end try
```

Fig. 8.6 | Overloaded constructors used to initialize `Time2` objects. (Part 2 of 4.)

```
42         catch ( IllegalArgumentException e )
43         {
44            System.out.printf( "\nException while initializing t6: %s\n",
45               e.getMessage() );
46         } // end catch
47      } // end main
48   } // end class Time2Test
```

```
Constructed with:
t1: all arguments defaulted
   00:00:00
   12:00:00 AM
t2: hour specified; minute and second defaulted
   02:00:00
   2:00:00 AM
```

Fig. 8.6 | Overloaded constructors used to initialize `Time2` objects. (Part 3 of 4.)

```
t3: hour and minute specified; second defaulted
   21:34:00
   9:34:00 PM
t4: hour, minute and second specified
   12:25:42
   12:25:42 PM
t5: Time2 object t4 specified
   12:25:42
   12:25:42 PM

Exception while initializing t6: hour must be 0-23
```

Fig. 8.6 | Overloaded constructors used to initialize `Time2` objects. (Part 4 of 4.)

- A program can declare a so-called no-argument constructor that is invoked without arguments.
- Such a constructor simply initializes the object as specified in the constructor's body.
- Using this in method-call syntax as the first statement in a constructor's body invokes another constructor of the same class.
  - Popular way to reuse initialization code provided by another of the class's constructors rather than defining similar code in the no-argument constructor's body.
- Once you declare any constructors in a class, the compiler will not provide a default constructor.
- Methods can access a class's private data directly without calling the *set* and *get* methods.
- However, consider changing the representation of the time from three int values (requiring 12 bytes of memory) to a single int value representing the total number of seconds that have elapsed since midnight (requiring only 4 bytes of memory).
- If we made such a change, only the bodies of the methods that access the private data directly would need to change—in particular, the individual *set* and *get* methods for the hour, minute and second.
- There would be no need to modify the bodies of methods setTime, toUniversalString or toString because they do not access the data directly.

- Designing the class in this manner reduces the likelihood of programming errors when altering the class's implementation.
- Similarly, each Time2 constructor could be written to include a copy of the appropriate statements from methods setHour, setMinute and setSecond.
- Doing so may be slightly more efficient, because the extra constructor call and call to setTime are eliminated.
- However, duplicating statements in multiple methods or constructors makes changing the class's internal data representation more difficult.
- Having the Time2 constructors call the constructor with three arguments (or even call setTime directly) requires any changes to the implementation of setTime to be made only once.

Default and No-Argument Constructors

- Every class must have at least one constructor.
- If you do not provide any constructors in a class's declaration, the compiler creates a default constructor that takes no arguments when it's invoked.
- The default constructor initializes the instance variables to the initial values specified in their declarations or to their default values (zero for primitive numeric types, false for boolean values and null for references).
- If your class declares constructors, the compiler will not create a default constructor.
- In this case, you must declare a no-argument constructor if default initialization is required.
- Like a default constructor, a no-argument constructor is invoked with empty parentheses.

Notes on Set and Get Methods

- Classes often provide public methods to allow clients of the class to *set* (i.e., assign values to) or *get* (i.e., obtain the values of) private instance variables.
- *Set* methods are also commonly called mutator methods, because they typically change an object's state—i.e., modify the values of instance variables.
- *Get* methods are also commonly called accessor methods or query methods.
- It would seem that providing *set* and *get* capabilities is essentially the same as making the instance variables public.
    - A public instance variable can be read or written by any method that has a reference to an object that contains that variable.
    - If an instance variable is declared private, a public *get* method certainly allows other methods to access it, but the *get* method can control how the client can access it.
    - A public *set* method can—and should—carefully scrutinize at-tempts to modify the variable's value to ensure valid values.
- Although *set* and *get* methods provide access to private data, it is restricted by the implementation of the methods.
- ***Validity Checking in Set Methods***
- The benefits of data integrity do not follow automatically simply because instance variables are declared private—you must provide validity checking.
- ***Predicate Methods***
- Another common use for accessor methods is to test whether a condition is true or false—such methods are often called predicate methods.
  Example: ArrayList's isEmpty method, which returns true if the ArrayList is empty.

4.4 Composition

- A class can have references to objects of other classes as members.
- This is called composition and is sometimes referred to as a has-a relationship.
- Example: An AlarmClock object needs to know the current time and the time when it's supposed to sound its alarm, so it's reasonable to include two references to Time objects in an AlarmClock object.

```
1  // Fig. 8.7: Date.java
2  // Date class declaration.
3
4  public class Date
5  {
6     private int month; // 1-12
7     private int day; // 1-31 based on month
8     private int year; // any year
9
10    private static final int[] daysPerMonth = // days in each month
11       { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
12
13    // constructor: call checkMonth to confirm proper value for month;
14    // call checkDay to confirm proper value for day
15    public Date( int theMonth, int theDay, int theYear )
16    {
17       month = checkMonth( theMonth ); // validate month
18       year = theYear; // could validate year
19       day = checkDay( theDay ); // validate day
20
```

Fig. 8.7 | `Date` class declaration. (Part 1 of 3.)

```
21       System.out.printf(
22          "Date object constructor for date %s\n", this );
23    } // end Date constructor
24
25    // utility method to confirm proper month value
26    private int checkMonth( int testMonth )
27    {
28       if ( testMonth > 0 && testMonth <= 12 ) // validate month
29          return testMonth;
30       else // month is invalid
31          throw new IllegalArgumentException( "month must be 1-12" );
32    } // end method checkMonth
33
34    // utility method to confirm proper day value based on month and year
35    private int checkDay( int testDay )
36    {
37       // check if day in range for month
38       if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
39          return testDay;
40
```

Fig. 8.7 | `Date` class declaration. (Part 2 of 3.)

```
41         // check for leap year
42         if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
43              ( year % 4 == 0 && year % 100 != 0 ) ) )
44            return testDay;
45
46         throw new IllegalArgumentException(
47            "day out-of-range for the specified month and year" );
48      } // end method checkDay
49
50      // return a String of the form month/day/year
51      public String toString()
52      {
53         return String.format( "%d/%d/%d", month, day, year );
54      } // end method toString
55   } // end class Date
```

Fig. 8.7 | `Date` class declaration. (Part 3 of 3.)

```
1  // Fig. 8.8: Employee.java
2  // Employee class with references to other objects.
3
4  public class Employee
5  {
6     private String firstName;
7     private String lastName;
8     private Date birthDate;
9     private Date hireDate;
10
11    // constructor to initialize name, birth date and hire date
12    public Employee( String first, String last, Date dateOfBirth,
13       Date dateOfHire )
14    {
15       firstName = first;
16       lastName = last;
17       birthDate = dateOfBirth;
18       hireDate = dateOfHire;
19    } // end Employee constructor
20
```

Fig. 8.8 | `Employee` class with references to other objects. (Part 1 of 2.)

```
21    // convert Employee to String format
22    public String toString()
23    {
24       return String.format( "%s, %s  Hired: %s  Birthday: %s",
25          lastName, firstName, hireDate, birthDate );
26    } // end method toString
27 } // end class Employee
```

Fig. 8.8 | `Employee` class with references to other objects. (Part 2 of 2.)

```
1  // Fig. 8.9: EmployeeTest.java
2  // Composition demonstration.
3
4  public class EmployeeTest
5  {
6     public static void main( String[] args )
7     {
8        Date birth = new Date( 7, 24, 1949 );
9        Date hire = new Date( 3, 12, 1988 );
10       Employee employee = new Employee( "Bob", "Blue", birth, hire );
11
12       System.out.println( employee );
13    } // end main
14 } // end class EmployeeTest
```

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Blue, Bob  Hired: 3/12/1988  Birthday: 7/24/1949
```

Fig. 8.9 | Composition demonstration.

4.5 Enumerations

- The basic enum type defines a set of constants represented as unique identifiers.
- Like classes, all enum types are reference types.
- An enum type is declared with an enum declaration, which is a comma-separated list of enum constants
- The declaration may optionally include other components of traditional classes, such as constructors, fields and methods.
- Each enum declaration declares an enum class with the following restrictions:
  - enum constants are implicitly final, because they declare constants that shouldn't be modified.
  - enum constants are implicitly static.
  - Any attempt to create an object of an enum type with operator new results in a compilation error.
  - enum constants can be used anywhere constants can be used, such as in the case labels of switch statements and to control enhanced for statements.
  - enum declarations contain two parts—the enum constants and the other members of the enum type.
  - An enum constructor can specify any number of parameters and can be overloaded.
- For every enum, the compiler generates the static method values that returns an array of the enum's constants.
- When an enum constant is converted to a String, the constant's identifier is used as the String representation.

```
1  // Fig. 8.10: Book.java
2  // Declaring an enum type with constructor and explicit instance fields
3  // and accessors for these fields
4
5  public enum Book
6  {
7     // declare constants of enum type
8     JHTP( "Java How to Program", "2012" ),
9     CHTP( "C How to Program", "2007" ),
10    IW3HTP( "Internet & World Wide Web How to Program", "2008" ),
11    CPPHTP( "C++ How to Program", "2012" ),
12    VBHTP( "Visual Basic 2010 How to Program", "2011" ),
13    CSHARPHTP( "Visual C# 2010 How to Program", "2011" );
14
15    // instance fields
16    private final String title; // book title
17    private final String copyrightYear; // copyright year
18
```

Fig. 8.10 | Declaring an `enum` type with constructor and explicit instance fields and accessors for these fields. (Part 1 of 2.)

```
19    // enum constructor
20    Book( String bookTitle, String year )
21    {
22       title = bookTitle;
23       copyrightYear = year;
24    } // end enum Book constructor
25
26    // accessor for field title
27    public String getTitle()
28    {
29       return title;
30    } // end method getTitle
31
32    // accessor for field copyrightYear
33    public String getCopyrightYear()
34    {
35       return copyrightYear;
36    } // end method getCopyrightYear
37 } // end enum Book
```

Fig. 8.10 | Declaring an `enum` type with constructor and explicit instance fields and accessors for these fields. (Part 2 of 2.)

```
1  // Fig. 8.11: EnumTest.java
2  // Testing enum type Book.
3  import java.util.EnumSet;
4
5  public class EnumTest
6  {
7     public static void main( String[] args )
8     {
9        System.out.println( "All books:\n" );
10
11       // print all books in enum Book
12       for ( Book book : Book.values() )
13          System.out.printf( "%-10s%-45s%s\n", book,
14             book.getTitle(), book.getCopyrightYear() );
15
16       System.out.println( "\nDisplay a range of enum constants:\n" );
17
18       // print first four books
19       for ( Book book : EnumSet.range( Book.JHTP, Book.CPPHTP ) )
20          System.out.printf( "%-10s%-45s%s\n", book,
21             book.getTitle(), book.getCopyrightYear() );
22    } // end main
23 } // end class EnumTest
```

Fig. 8.11 | Testing an enum type. (Part 1 of 2.)

```
All books:

JHTP       Java How to Program                           2012
CHTP       C How to Program                              2007
IW3HTP     Internet & World Wide Web How to Program      2008
CPPHTP     C++ How to Program                            2012
VBHTP      Visual Basic 2010 How to Program              2011
CSHARPHTP  Visual C# 2010 How to Program                 2011

Display a range of enum constants:

JHTP       Java How to Program                           2012
CHTP       C How to Program                              2007
IW3HTP     Internet & World Wide Web How to Program      2008
CPPHTP     C++ How to Program                            2012
```

Fig. 8.11 | Testing an enum type. (Part 2 of 2.)

- Use the static method range of class EnumSet (declared in package java.util) to access a range of an enum's constants.
  - Method range takes two parameters—the first and the last enum constants in the range
  - Returns an EnumSet that contains all the constants between these two constants, inclusive.
- The enhanced for statement can be used with an EnumSet just as it can with an array.
- Class EnumSet provides several other static methods.

4.6 static Class Members

- In certain cases, only one copy of a particular variable should be shared by all objects of a class.
  - A static field—called a class variable—is used in such cases.
- A static variable represents classwide information—all objects of the class share the same piece of data.
  - The declaration of a static variable begins with the keyword static.
- Static variables have class scope.
- Can access a class's public static members through a reference to any object of the class, or by qualifying the member name with the class name and a dot (.), as in Math.random().
- private static class members can be accessed by client code only through methods of the class.
- static class members are available as soon as the class is loaded into memory at execution time.
- To access a public static member when no objects of the class exist (and even when they do), prefix the class name and a dot (.) to the static member, as in Math.PI.
- To access a private static member when no objects of the class exist, provide a public static method and call it by qualifying its name with the class name and a dot.
- A static method cannot access non-static class members, because a static method can be called even when no objects of the class have been instantiated.
  - For the same reason, the this reference cannot be used in a static method.
  - The this reference must refer to a specific object of the class, and when a static method is called, there might not be any objects of its class in memory.
- If a static variable is not initialized, the compiler assigns it a default value—in this case 0, the default value for type int.

```
1  // Fig. 8.12: Employee.java
2  // Static variable used to maintain a count of the number of
3  // Employee objects in memory.
4
5  public class Employee
6  {
7     private String firstName;
8     private String lastName;
9     private static int count = 0; // number of Employees created
10
11    // initialize Employee, add 1 to static count and
12    // output String indicating that constructor was called
13    public Employee( String first, String last )
14    {
15       firstName = first;
16       lastName = last;
17
18       ++count;  // increment static count of employees
19       System.out.printf( "Employee constructor: %s %s; count = %d\n",
20          firstName, lastName, count );
21    } // end Employee constructor
22
```

Fig. 8.12 | `static` variable used to maintain a count of the number of `Employee` objects in memory. (Part 1 of 2.)

```
23      // get first name
24      public String getFirstName()
25      {
26         return firstName;
27      } // end method getFirstName
28
29      // get last name
30      public String getLastName()
31      {
32         return lastName;
33      } // end method getLastName
34
35      // static method to get static count value
36      public static int getCount()
37      {
38         return count;
39      } // end method getCount
40   } // end class Employee
```

Fig. 8.12 | `static` variable used to maintain a count of the number of `Employee` objects in memory. (Part 2 of 2.)

- String objects in Java are immutable—they cannot be modified after they are created.
  - Therefore, it's safe to have many references to one String object.
  - This is not normally the case for objects of most other classes in Java.
- If String objects are immutable, you might wonder why are we able to use operators + and += to concatenate String objects.
- String-concatenation operations actually result in a new String object containing the concatenated values—the original String objects are not modified.

```
1  // Fig. 8.13: EmployeeTest.java
2  // static member demonstration.
3
4  public class EmployeeTest
5  {
6     public static void main( String[] args )
7     {
8        // show that count is 0 before creating Employees
9        System.out.printf( "Employees before instantiation: %d\n",
10          Employee.getCount() );
11
12       // create two Employees; count should be 2
13       Employee e1 = new Employee( "Susan", "Baker" );
14       Employee e2 = new Employee( "Bob", "Blue" );
15
16       // show that count is 2 after creating two Employees
17       System.out.println( "\nEmployees after instantiation: " );
18       System.out.printf( "via e1.getCount(): %d\n", e1.getCount() );
19       System.out.printf( "via e2.getCount(): %d\n", e2.getCount() );
20       System.out.printf( "via Employee.getCount(): %d\n",
21          Employee.getCount() );
22
```

Fig. 8.13 | `static` member demonstration. (Part 1 of 3.)

```
23         // get names of Employees
24         System.out.printf( "\nEmployee 1: %s %s\nEmployee 2: %s %s\n",
25            e1.getFirstName(), e1.getLastName(),
26            e2.getFirstName(), e2.getLastName() );
27
28         // in this example, there is only one reference to each Employee,
29         // so the following two statements indicate that these objects
30         // are eligible for garbage collection
31         e1 = null;
32         e2 = null;
33      } // end main
34   } // end class EmployeeTest
```

Fig. 8.13 | `static` member demonstration. (Part 2 of 3.)

```
Employees before instantiation: 0
Employee constructor: Susan Baker; count = 1
Employee constructor: Bob Blue; count = 2

Employees after instantiation:
via e1.getCount(): 2
via e2.getCount(): 2
via Employee.getCount(): 2

Employee 1: Susan Baker
Employee 2: Bob Blue
```

Fig. 8.13 | `static` member demonstration. (Part 3 of 3.)

- Objects become "eligible for garbage collection" when there are no more references to them in the program.
- Eventually, the garbage collector might reclaim the memory for these objects (or the operating system will reclaim the memory when the program terminates).
- The JVM does not guarantee when, or even whether, the garbage collector will execute.
- When the garbage collector does execute, it's possible that no objects or only a subset of the eligible objects will be collected.

static Import

- A static import declaration enables you to import the static members of a class or interface so you can access them via their unqualified names in your class—the class name and a dot (.) are not required to use an imported static member.
- Two forms
  - One that imports a particular static member (which is known as single static import)
  - One that imports all static members of a class (which is known as static import on demand)
- The following syntax imports a particular static member:
  import static *packageName.ClassName.staticMemberName;*
- where *packageName* is the package of the class, ClassName is the name of the class and *staticMemberName* is the name of the static field or method.
- The following syntax imports all static members of a class:
  import static *packageName.ClassName.*;*

- where *packageName* is the package of the class and *ClassName* is the name of the class.
- * indicates that *all* static members of the specified class should be available for use in the class(es) declared in the file.
- static import declarations import only static class members.
- Regular import statements should be used to specify the classes used in a program.

```
1  // Fig. 8.14: StaticImportTest.java
2  // Static import of Math class methods.
3  import static java.lang.Math.*;
4
5  public class StaticImportTest
6  {
7     public static void main( String[] args )
8     {
9        System.out.printf( "sqrt( 900.0 ) = %.1f\n", sqrt( 900.0 ) );
10       System.out.printf( "ceil( -9.8 ) = %.1f\n", ceil( -9.8 ) );
11       System.out.printf( "E = %f\n", E );
12       System.out.printf( "PI = %f\n", PI );
13    } // end main
14 } // end class StaticImportTest
```

```
sqrt( 900.0 ) = 30.0
ceil( -9.8 ) = -9.0
log( E ) = 1.0
cos( 0.0 ) = 1.0
```

Fig. 8.14 | Static import of `Math` class methods.

final Instance Variables

- The principle of least privilege is fundamental to good software engineering.
  - Code should be granted only the amount of privilege and access that it needs to accomplish its designated task, but no more.
  - Makes your programs more robust by preventing code from accidentally (or maliciously) modifying variable values and calling methods that should not be accessible.
- Keyword final specifies that a variable is not modifiable (i.e., it's a constant) and any attempt to modify it is an error.

  private final int INCREMENT;
  - Declares a final (constant) instance variable INCREMENT of type int.
- final variables can be initialized when they are declared or by each of the class's constructors so that each object of the class has a different value.
- If a class provides multiple constructors, every one would be required to initialize each final variable.
- A final variable cannot be modified by assignment after it's initialized.
- If a final variable is not initialized, a compilation error occurs.

4.7 Time Class Case Study: Creating Packages

- Each class in the Java API belongs to a package that contains a group of related classes.
- Packages are defined once, but can be imported into many programs.
- Packages help programmers manage the complexity of application components.

- Packages facilitate software reuse by enabling programs to import classes from other packages, rather than copying the classes into each program that uses them.
- Packages provide a convention for unique class names, which helps prevent class-name conflicts.
- The steps for creating a reusable class:
- Declare a public class; otherwise, it can be used only by other classes in the same package.
- Choose a unique package name and add a package declaration to the source-code file for the reusable class declaration.
- In each Java source-code file there can be only one package declaration, and it must precede all other declarations and statements.
- Compile the class so that it's placed in the appropriate package directory.
- Import the reusable class into a program and use the class.
- Placing a package declaration at the beginning of a Java source file indicates that the class declared in the file is part of the specified package.
- Only package declarations, import declarations and comments can appear outside the braces of a class declaration.
- A Java source-code file must have the following order:
  - a package declaration (if any),
  - import declarations (if any), then
  - class declarations.
- Only one of the class declarations in a particular file can be public.
- Other classes in the file are placed in the package and can be used only by the other classes in the package.
- Non-public classes are in a package to support the reusable classes in the package.

```
1  // Fig. 8.15: Time1.java
2  // Time1 class declaration maintains the time in 24-hour format.
3  package com.deitel.jhtp.ch08;
4
5  public class Time1
6  {
7     private int hour; // 0 - 23
8     private int minute; // 0 - 59
9     private int second; // 0 - 59
10
11    // set a new time value using universal time; throw an
12    // exception if the hour, minute or second is invalid
13    public void setTime( int h, int m, int s )
14    {
15       // validate hour, minute and second
16       if ( ( h >= 0 && h < 24 ) && ( m >= 0 && m < 60 ) &&
17          ( s >= 0 && s < 60 ) )
18       {
19          hour = h;
20          minute = m;
21          second = s;
22       } // end if
```

Fig. 8.15 | Packaging class `Time1` for reuse. (Part 1 of 2.)

```
23         else
24            throw new IllegalArgumentException(
25               "hour, minute and/or second was out of range" );
26      } // end method setTime
27
28      // convert to String in universal-time format (HH:MM:SS)
29      public String toUniversalString()
30      {
31         return String.format( "%02d:%02d:%02d", hour, minute, second );
32      } // end method toUniversalString
33
34      // convert to String in standard-time format (H:MM:SS AM or PM)
35      public String toString()
36      {
37         return String.format( "%d:%02d:%02d %s",
38            ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
39            minute, second, ( hour < 12 ? "AM" : "PM" ) );
40      } // end method toString
41   } // end class Time1
```

Fig. 8.15 | Packaging class `Time1` for reuse. (Part 2 of 2.)

- Every package name should start with your Internet domain name in reverse order.
  - For example, our domain name is deitel.com, so our package names begin with com.deitel.
  - For the domain name *yourcollege.edu, the package name should begin with edu.yourcollege.*
- After the domain name is reversed, you can choose any other names you want for your package.
  - We chose to use jhtp as the next name in our package name to indicate that this class is from *Java How to Program.*
  - The last name in our package name specifies that this package is for Chapter 8 (ch08).
- Compile the class so that it's stored in the appropriate package.
- When a Java file containing a package declaration is compiled, the resulting class file is placed in the directory specified by the declaration.
- The package declaration

package com.deitel.jhtp.ch08;

- indicates that class Time1 should be placed in the directory

com
deitel
jhtp
ch08

- The directory names in the package declaration specify the exact location of the classes in the package.
- javac command-line option -d causes the javac compiler to create appropriate directories based on the class's package declaration.
- The option also specifies where the directories should be stored.
- Example:

javac -d . Time1.java

- specifies that the first directory in our package name should be placed in the current directory (.).
- The compiled classes are placed into the directory that is named last in the package statement.
- The package name is part of the fully qualified class name.
- Class Time1's name is actually com.deitel.jhtp.ch08.Time1

- Can use the fully qualified name in programs, or import the class and use its simple name (the class name by itself).
- If another package contains a class by the same name, the fully qualified class names can be used to distinguish between the classes in the program and prevent a name conflict (also called a name collision).
- Fig. 8.15, line 3 is a single-type-import declaration
- It specifies one class to import.
- When your program uses multiple classes from the same package, you can import those classes with a type-import-on-demand declaration.
- Example:

  import java.util.*; // import java.util classes
- uses an asterisk (*) at the end of the import declaration to inform the compiler that all public classes from the java.util package are available for use in the program.
- Only the classes from package java-.util that are used in the program are loaded by the JVM.

```
1  // Fig. 8.16: Time1PackageTest.java
2  // Time1 object used in an application.
3  import com.deitel.jhtp.ch08.Time1; // import class Time1
4
5  public class Time1PackageTest
6  {
7     public static void main( String[] args )
8     {
9        // create and initialize a Time1 object
10       Time1 time = new Time1(); // invokes Time1 constructor
11
12       // output string representations of the time
13       System.out.print( "The initial universal time is: " );
14       System.out.println( time.toUniversalString() );
15       System.out.print( "The initial standard time is: " );
16       System.out.println( time.toString() );
17       System.out.println(); // output a blank line
18
```

Fig. 8.16 | Time1 object used in an application. (Part 1 of 3.)

```
19         // change time and output updated time
20         time.setTime( 13, 27, 6 );
21         System.out.print( "Universal time after setTime is: " );
22         System.out.println( time.toUniversalString() );
23         System.out.print( "Standard time after setTime is: " );
24         System.out.println( time.toString() );
25         System.out.println(); // output a blank line
26
27         // attempt to set time with invalid values
28         try
29         {
30            time.setTime( 99, 99, 99 ); // all values out of range
31         } // end try
32         catch ( IllegalArgumentException e )
33         {
34            System.out.printf( "Exception: %s\n\n", e.getMessage() );
35         } // end catch
36
```

Fig. 8.16 | `Time1` object used in an application. (Part 2 of 3.)

```
37         // display time after attempt to set invalid values
38         System.out.println( "After attempting invalid settings:" );
39         System.out.print( "Universal time: " );
40         System.out.println( time.toUniversalString() );
41         System.out.print( "Standard time: " );
42         System.out.println( time.toString() );
43      } // end main
44   } // end class Time1PackageTest
```

```
The initial universal time is: 00:00:00
The initial standard time is: 12:00:00 AM

Universal time after setTime is: 13:27:06
Standard time after setTime is: 1:27:06 PM

Exception: hour, minute and/or second was out of range

After attempting invalid settings:
Universal time: 13:27:06
Standard time: 1:27:06 PM
```

Fig. 8.16 | `Time1` object used in an application. (Part 3 of 3.)

- Specifying the Classpath During Compilation
- When compiling a class that uses classes from other packages, javac must locate the .class files for all other classes being used.
- The compiler uses a special object called a class loader to locate the classes it needs.
  - The class loader begins by searching the standard Java classes that are bundled with the JDK.
  - Then it searches for optional packages.
  - If the class is not found in the standard Java classes or in the extension classes, the class loader searches the classpath, which contains a list of locations in which classes are stored.

- The classpath consists of a list of directories or archive files, each separated by a directory separator
- Semicolon (;) on Windows or a colon (:) on UNIX/Linux/Mac OS X.
- Archive files are individual files that contain directories of other files, typically in a compressed format.
- Archive files normally end with the .jar or .zip file-name extensions.
- The directories and archive files specified in the classpath contain the classes you wish to make available to the Java compiler and the JVM.
  -
- Specifying the Classpath When Executing an Application
- When you execute an application, the JVM must be able to locate the .class files of the classes used in that application.
- Like the compiler, the java command uses a class loader that searches the standard classes and extension classes first, then searches the classpath (the current directory by default).
- The classpath can be specified explicitly by using either of the techniques discussed for the compiler.
- As with the compiler, it's better to specify an individual program's classpath via command-line JVM options.
  - If classes must be loaded from the current directory, be sure to include a dot (.) in the classpath to specify the current directory.

4.8 Package Access

- If no access modifier is specified for a method or variable when it's declared in a class, the method or variable is considered to have package access.
- In a program uses multiple classes from the same package, these classes can access each other's package-access members directly through references to objects of the appropriate classes, or in the case of static members through the class name.
- Package access is rarely used.

```
1  // Fig. 8.17: PackageDataTest.java
2  // Package-access members of a class are accessible by other classes
3  // in the same package.
4
5  public class PackageDataTest
6  {
7     public static void main( String[] args )
8     {
9        PackageData packageData = new PackageData();
10
11       // output String representation of packageData
12       System.out.printf( "After instantiation:\n%s\n", packageData );
13
14       // change package access data in packageData object
15       packageData.number = 77;
16       packageData.string = "Goodbye";
17
18       // output String representation of packageData
19       System.out.printf( "\nAfter changing values:\n%s\n", packageData );
20    } // end main
21 } // end class PackageDataTest
22
```

Fig. 8.17 | Package-access members of a class are accessible by other classes in the same package. (Part 1 of 3.)

```
23  // class with package access instance variables
24  class PackageData
25  {
26     int number; // package-access instance variable
27     String string; // package-access instance variable
28
29     // constructor
30     public PackageData()
31     {
32        number = 0;
33        string = "Hello";
34     } // end PackageData constructor
35
36     // return PackageData object String representation
37     public String toString()
38     {
39        return String.format( "number: %d; string: %s", number, string );
40     } // end method toString
41  } // end class PackageData
```

Fig. 8.17 | Package-access members of a class are accessible by other classes in the same package. (Part 2 of 3.)

```
After instantiation:
number: 0; string: Hello

After changing values:
number: 77; string: Goodbye
```

Fig. 8.17 | Package-access members of a class are accessible by other classes in the same package. (Part 3 of 3.)