Chapter 2

Java Fundamentals

Goals

- Introduce the Java syntax necessary to write programs
- Be able to write a program with user input and console output
- Evaluate and write arithmetic expressions
- Use a few of Java's types such as `int` and `double`

2.1 Elements of Java Programming

The essential building block of Java programs is the **class**. In essence, a Java class is a sequence of characters (text) stored as a file, whose name always ends with `.java`. Each class is comprised of several elements, such as a **class heading** (`public class` *`class-name`*) and **methods**—a collection of statements grouped together to provide a service. Below is the general form for a Java class that has one method: `main`. Any class with a `main` method, including those with only a `main` method, can be run as a program.

***General Form:* A simple Java program (only one class)**

```
// Comments: any text that follows // on the same line
import package-name.class-name;

public class class-name {

  public static void main(String[] args) {
    variable declarations and initializations
    messages and operations such as assignments
  }
}
```

General forms describe the **syntax** necessary to write code that compiles. The general forms in this textbook use the following conventions:

- Boldface elements must be written exactly as shown. This includes words such as **`public static void main`** and symbols such as **`[`**, **`]`**, **`(`**, and).
- Italicized items are defined somewhere else or must be supplied by the programmer.

A Java Class with One Method Named `main`

```
// Read a number and display that input value squared
import java.util.Scanner;

public class ReadItAndSquareIt {

  public static void main(String[] args) {
    // Allow user input from the keyboard
    Scanner keyboard = new Scanner(System.in);

    // I)nput Prompt user for a number and get it from the keyboard
    System.out.print("Enter an integer: ");
    int number = keyboard.nextInt();

    // P)rocess
    int result = number * number;

    // O)utput
    System.out.println(number + " squared = " + result);
  }
}
```

Dialog

```
Enter an integer: -12
-12 squared = 144
```

The first line in the program shown above is a comment indicating what the program will do. Comments in Java are always preceded by the // symbol, and are "ignored" by the program. The next line contains the word `import`, which allows a program to use classes stored in other files. This program above has access to a class named `Scanner` for reading user input. If you omit the import statement, you will get this error:

```
Scanner keyboard = new Scanner(System.in);
    Scanner cannot be resolved to a type
```

Java classes, also known as types, are organized into over seventy packages. Each package contains a set of related classes. For example, `java.net` has classes related to networking, and `java.io` has a collection of classes for performing input and output. To use these classes, you could simply use the import statement. Otherwise you would have to precede the class name with the correct package name, like this:

```
java.util.Scanner keyboard = new java.util.Scanner(System.in);
```

The next line in the sample program is a class heading. A class is a collection of methods and variables (both discussed later) enclosed within a set of matching curly braces. You may use any valid class name after `public class`; however, the class name must match the file name. Therefore, the preceding program must be stored in a file named `ReadItAndSquareIt.java`.

The file-naming convention

class-name`.java`

The next line in the program is a method heading that, for now, is best retyped exactly as shown (an explanation–intentionally skipped here–is required to have a program):

```
public static void main(String[] args)     // Method heading
```

The opening curly brace begins the body of the `main` method, which is a collection of executable statements and variables. This `main` method body above contains a variable declaration, variable initializations, and four messages, all of which are described later in this chapter. When run as a program,

the first statement in main will be the first statement executed. The body of the method ends with a closing curly brace.
This Java source code represents input to the Java compiler. A compiler is a program that translates source code into a language that is closer to what the computer hardware understands. Along the way, the compiler generates error messages if it detects a violation of any Java syntax rules in your source code. Unless you are perfect, you will see the compiler generate errors as the program scans your source code.

Tokens — The Smallest Pieces of a Program

As the Java compiler reads the source code, it identifies individual **tokens**, which are the smallest recognizable components of a program. Tokens fall into four categories:

Token	Examples
Special symbols	`;  ()  ,  .  {  }`
Identifiers	`main args credits courseGrade String List`
Reserved identifiers	`public static void class double int`
Literals (constant values)	`"Hello World!" 0 -2.1 'C' true`

Tokens make up more complex pieces of a program. Knowing the types of tokens in Java should help you to:

- More easily write syntactically correct code.
- Better understand how to fix syntax errors detected by the compiler.
- Understand general forms.
- Complete programs more quickly and easily.

Special Symbols

A special symbol is a sequence of one or two characters, with one or possibly many specific meanings. Some special symbols separate other tokens, for example: `{`, `;`, and `,`. Other special symbols represent operators in expressions, such as: `+`, `-`, and `/`. Here is a partial list of single-character and double-character special symbols frequently seen in Java programs:

```
()   .   +   -   /   *   <=   >=   //   { }   ==   ;
```

Identifiers

Java **identifiers** are words that represent a variety of things. `String`, for example is the name of a class for storing a string of characters. Here are some other identifiers that Java has already given meaning to:

```
sqrt  String  get  println  readLine  System  equals  Double
```

Programmers must often create their own identifiers. For example, `test1`, `finalExam`, `main`, and `courseGrade` are identifiers defined by programmers. All identifiers follow these rules.

- Identifiers begin with upper- or lowercase letters `a` through `z` (or `A` through `Z`), the dollar sign `$`, or the underscore character `_`.
- The first character may be followed by a number of upper- and lowercase letters, digits (`0` through `9`), dollar signs, and underscore characters.
- Identifiers are case sensitive; `Ident`, `ident`, and `iDENT` are three different identifiers.

Valid Identifiers

```
main     ArrayList   incomeTax       MAX_SIZE          $Money$
Maine    URL         employeeName    all_4_one         _balance
miSpel    String     A1               world_in_motion   balance
```

Invalid Identifiers

```
1A           // Begins with a digit
miles/Hour   // The / is not allowed
first Name   // The blank space not allowed
pre-shrunk   // The operator - means subtraction
```

Java is case sensitive. For example, to run a class as a program, you must have the identifier `main`. `MAIN` or `Main` won't do. The convention employed by Java programmers is to use the "camelBack" style for variables. The first letter is always lowercase, and each subsequent new word begins with an uppercase letter. For example, you will see `letterGrade` rather than `lettergrade`, `LetterGrade`, or `letter_grade`. Class names use the same convention, except the first letter is also in uppercase. You will see `String` rather than `string`.

Reserved Identifiers

Reserved identifiers in Java are identifiers that have been set aside for a specific purpose. Their meanings are fixed by the standard language definition, such as `double` and `int`. They follow the same rules as regular identifiers, but they cannot be used for any other purpose. Here is a partial list of Java reserved identifiers, which are also known as keywords.

Java Keywords

boolean	**default**	**for**	**new**
break	**do**	**if**	**private**
case	**double**	**import**	**public**
catch	**else**	**instanceof**	**return**
char	**extends**	**int**	**void**
class	**float**	**long**	**while**

The case sensitivity of Java applies to keywords. For example, there is a difference between `double` (a keyword) and `Double` (an identifier, not a keyword). All Java keywords are written in lowercase letters.

Literals

A literal value such as 123 or -94.02 is one that cannot be changed. Java recognizes these numeric literals and several others, including `String` literals that have zero or more characters enclosed within a pair of double quotation marks.

```
"Double quotes are used to delimit String literals."
"Hello, World!"
```

Integer literals are written as numbers without decimal points. Floating-point literals are written as numbers with decimal points (or in exponential notation: 5e3 = 5 * 10^3 = 5000.0 and 1.23e-4 = 1.23 x 10^{-4} = 0.0001234). Here are a few examples of integer, floating-point, string, and character literals in Java, along with both Boolean literals (`true` and `false`) and the `null` literal value.

The Six Types of Java Literals

Integer	Floating Point	String	Character	Boolean	Null
-2147483648	-1.0	"A"	'a'	**true**	**null**
-1	0.0	"Hello World"	'0'	**false**	
0	39.95	"\n new line"	'?'		
1	1.23e09	"1.23"	' '		
2147483647	-1e6	"The answer is: "	'7'		

Note: Other literals are possible such as `12345678901L` for integers > 2,147,483,647.

Comments

Comments are portions of text that annotate a program, and fulfill any or all of the following expectations:

- Provide internal documentation to help one programmer read and understand another's program.
- Explain the purpose of a method.
- Describe what a method expects of the input arguments (n must be > 0, for example).
- Describe a wide variety of program elements.

Comments may be added anywhere within a program. They may begin with the two-character special symbol `/*` when closed with the corresponding symbol `*/`.

```
/*
   A comment may extend over many lines
   when using slash start at the beginning
   and ending the comment with a star slash.
*/
```

An alternate form for comments is to use `//` before a line of text. Such a comment may appear at the beginning of a line, in which case the entire line is "ignored" by the program, or at the end of a line, in which case all code prior to the special symbol will be executed.

```
// This Java program displays "hello, world to the console.
public class ShowHello {

  public static void main(String[] args) {
    System.out.println("hello, world");
  }
}
```

Comments can help clarify and document the purpose of code. Using intention-revealing identifiers and writing code that is easy to understand, however, can also do this.

Self-Check

2-1 List each of the following as a valid identifier or explain why it is not valid.

-a	`abc`	-i	`H.P.`
-b	`123`	-j	`double`
-c	`ABC`	-k	`55_mph`
-d	`_.$`	-l	`sales Tax`
-e	`my Age`	-m	`$$$$`
-f	`identifier`	-n	`_____`
-g	`(identifier)`	-o	`Mile/Hour`
-h	`mispellted`	-p	`Scanner`

2-2 Which of the following are valid Java comments?

-a `// Is this a comment?`
-b `/ / Is this a comment?`
-c `/* Is this a comment?`
-d `/* Is this a comment? */`

2.2 Java Types

Java has two types of variables: primitive types and reference types. Reference variables store information necessary to locate complex values such as strings and arrays. On the other hand, Primitive variables store a single value in a fixed amount of computer memory. The eight "primitive" (simple) types are closely related to computer hardware. For example, an `int` value is stored in 32 bits (4 bytes) of memory. Those 32 bits represent a simple positive or negative integer value. Here is summary of all types in Java along with the range of values for the primitive types:

The Java Primitive Types

integers:
`byte` (8 bits) -128 .. 128
`short` (16 bits) -32,768 .. 32,767
`int` (32 bits) -2,147,483,648 .. 2,147,483,647
`long` (64 bits) -9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807

real numbers:
`float` (32 bits), ±1.40129846432481707e-45 .. ±3.40282346638528860e+38
`double` (64 bits), ±4.94065645841246544e-324 .. ±1.79769313486231570e+308

other primitive types:
`char` `'A'`, `'@'`, or `'z'` for example
`boolean` has only two literal values **`false`** and **`true`**

The Java Reference Types

classes Chapter 5
arrays Chapters 8-11
interfaces Chapter 12

Declaring a primitive variable provides the program with a named data value that can change while the program is running. An initialization allows the programmer to set the original value of a variable. This value can be accessed or changed later in the program by using the variable name.

General Form: **Initializing (declaring a primitive variable and giving it a value)**

```
type identifier;                 // Declare one variable
type identifier = initial-value; // For primitive types like int and double
```

Example: The following code declares one `int` and two `double` primitive variables while it initializes `grade`.

```
int credits;
double grade = 4.0;
double GPA;
```

The following table summarizes the initial value of these variables:

Variable Name	Value
`credits`	`?` `// Unknown`
`grade`	`4.0` `// This was initialized above`
`GPA`	`?` `// Unknown`

If you do not initialize a variable, it cannot be used unless it is changed with an assignment statement. The Java compiler would report this as an error.

Assignment

An **assignment** gives a value to a variable. The value of the expression to the right of the assignment operator (=) replaces the value of the variable to the left of =.

***General Form:* Assignment**

variable-name **=** *expression* **;**

The *expression* must be a value that can be stored by the type of variable to the left of the assignment operator (=). For example, an expression that results in a floating-point value can be stored in a `double` variable, and likewise an integer value can be stored in an `int` variable.

```
int credits = 4;
double grade = 3.0;
double GPA = (credits * grade) / credits; // * and / evaluate before =
```

The assignment operator = has a very low priority, it assigns after all other operators evaluate. For example, `(credits * grade) / credits` evaluates to 3.0 before 3.0 is assigned to `GPA`. These three assignments change the value of all three variables. The values can now be shown like this:

Variable	Value
`credits`	`4`
`grade`	`3.0`
`GPA`	`3.0`

In an assignment, the Java compiler will check to make sure you are assigning the correct type of value to the variable. For example, a string literal cannot be assigned to a numeric variable. A floating-point number cannot be stored in an `int`.

```
grade = "Noooooo, you can't do that"; // Cannot store string in a double
credits = 16.5; // Cannot store a floating-point number in an int
```

Self-Check

2-3 Which of the following are valid attempts at assignment, given these two declarations?

```
double aDouble = 0.0;
int anInt = 0;
```

-a `anInt = 1;`
-b `anInt = 1.5;`
-c `anInt = "1.5";`
-d `anInt = anInt + 1;`
-e `aDouble = 1;`
-f `aDouble = 1.5;`
-g `aDouble = "1.5";`
-h `aDouble = aDouble + 1.5;`

Input and Output (I/O)

Programs communicate with users. Such communication is provided through—but is not limited to—keyboard **input** and screen **output**. In Java, this two-way communication is made possible by sending messages, which provide a way to transfer control to another method that performs some well-defined responsibility. You may have written that method, or it may very likely be a method you cannot see in one of the existing Java classes. Some messages perform particular actions. Two such methods are the `print` and `println` messages sent to `System.out`:

***General Form:* Output with print and println**

`System.out.print(`*expression***`);`**
`System.out.println(`*expression***`);`**

`System.out` is an existing reference variable that represents the console—the place on the computer screen where text is displayed (not actually printed). The expression between the parentheses is known as the **argument**. In a print or println message, the value of the expression will be displayed on the computer screen. With `print` and `println`, the arguments can be any of the types mentioned so far (`int`, `double`, `char`, `boolean`), plus others. The semicolon (`;`) terminates messages. The only difference between `print` and `println` is that `println` generates a new line. Subsequent output begins at the beginning of a new line. Here are some valid output messages:

```
System.out.print("Enter credits: ");  // Use print to prompt the user
System.out.println();                 // Print a blank line
```

Input

To make programs more applicable to general groups of data—for example, to compute the GPA for any student—variables are often assigned values through keyboard input. This allows the program to accept data which is specific to the user. There are several options for obtaining user input from the keyboard. Perhaps the simplest option is to use the `Scanner` class from the `java.util` package. This class has methods that allow for easy input of numbers and other types of data, such as strings.

Before you can use `Scanner` messages such as `nextDouble` or `nextInt`, your code must create a reference variable to which messages can be sent. The following code initializes a reference variable named `keyboard` that will allow the keyboard to be a source of input. (`System.in` is an existing reference variable that allows characters to be read from the keyboard.)

Creating an Instance of `Scanner` to Read Numeric Input

```
// Store a reference variable named keyboard to read input from the user.
// System.in is a reference variable already associated with the keyboard
Scanner keyboard = new Scanner(System.in);
```

In general, a reference variable is initialized with the keyword `new` followed by *class-name* and *(initial-values)*.

***General Form:* Initializing reference variables with `new`**

class-name reference-variable-name **`=`** **`new`** *class-name* **`();`**
class-name reference-variable-name **`=`** **`new`** *class-name* **`(`***initial-value(s)***`);`**

The expression to the right of = evaluates to a reference value, which is then stored in the reference variable to the left of =. That reference value is used later for sending messages. Messages sent to `keyboard` can obtain textual input from the keyboard and can convert that text (for example, 3.45 and 99) into numbers. Here are two messages that allow users to input numbers into a program:

Numeric Input

```
keyboard.nextInt();     // Pause until user enters an integer
keyboard.nextDouble(); // Pause until user enters a floating-point number
```

In general, use this form to send a message to a reference variable that will, in turn, cause some operation to execute:

***General Form:* Sending messages**

reference-variable-name **`.`***message-name* **`(`***argument-list***`)`**

When a `nextInt` or `nextDouble` message is sent to `keyboard`, the method waits until the user enters some type of input and then presses the Enter key. If the user enters the number correctly, the text will be converted into the proper machine representation of the number. If the user enters a letter when `keyboard` is expecting a number, the program may terminate with an error message.

These two methods are examples of expressions that evaluate to some value. Whereas a `nextInt` message evaluates to a primitive `int` value, a `nextDouble` message evaluates to a primitive floating-point value. Because `nextInt` and `nextDouble` return numeric values, they are often seen on the right-hand side of assignment statements. These messages will be seen in text-based input and output programs (ones that have no graphical user interface).

For example, the following code prompts the user to enter two numbers using `print`, `nextInt`, and `nextDouble` messages.

```
System.out.print("Enter credits: ");     // Prompt the user
credits = keyboard.nextInt();            // Read and assign an integer
System.out.print("Enter grade: ");       // Prompt the user
qualityPoints = keyboard.nextDouble();   // Read and assign a double
```

Dialog

```
Enter credits: 4
Enter grade: 3.0
```

In the last line of code above—the fourth message—the `nextDouble` message causes a pause in program execution until the user enters a number. When the user types a number and presses the enter key, the `nextDouble` method converts the text user into a floating-point number. That value is then assigned to the variable `qualityPoints`. All of this happens in one line of code.

Prompt and Input

The output and input operations are often used together to obtain values from the user of the program. The program informs the user what must be entered with an output message and then sends an input message to get values for the variables. This happens so often that this activity can be considered to be a pattern. The **Prompt and Input** pattern has two activities:

1. Request the user to enter a value (prompt).
2. Obtain the value for the variable (input).

Algorithmic Pattern: Prompt and Input

Pattern:	Prompt and Input
Problem:	The user must enter something.
Outline:	1. Prompt the user for input. 2. Input the data
Code Example:	`System.out.println("Enter credits: ");` `int credits = keyboard.nextInt();`

Strange things may happen if the prompt is left out. The user will not know what must be entered. Whenever you require user input, make sure you prompt for it first. Write the code that tells the user precisely what you want. First output the prompt and then obtain the user input. Here is another instance of the Prompt and Input pattern:

```
System.out.println("Enter test #1: ");
double test1 = keyboard.nextDouble(); // Initialize test1 with input
System.out.println("You entered " + test1);
```

Dialogue

```
Enter test #1: 97.5
You entered 97.5
```

In general, tell the user what value is needed, then input a value into that variable with an input message such as `keyboard.nextDouble();`.

***General Form:* Prompt and Input**

```
System.out.println("prompt user for input : " );
input = keyboard.nextDouble();   // or keyboard.nextInt();
```

Arithmetic Expressions

Arithmetic expressions are made up of two components: operators and operands. An arithmetic operator is one of the Java special symbols `+`, `-`, `/`, or `*`. The operands of an arithmetic expression may be numeric variable names, such as `credits`, and numeric literals, such as `5` and `0.25`.

An Arithmetic Expression may be	Example
numeric variable	`double aDouble`
numeric literal	`100` or `99.5`
expression + expression	`aDouble + 100`
expression – expression	`aDouble - 100`
expression * expression	`aDouble * 100`
expression / expression	`aDouble / 99.5`
(expression)	`(aDouble + 2.0)`

The last definition of "expression" suggests that we can write more complex expressions.

```
1.5 * ((aDouble - 99.5) * 1.0 / aDouble)
```

Since arithmetic expressions may be written with many literals, numeric variable names, and operators, rules are put into force to allow a consistent evaluation of expressions. The following table lists four Java arithmetic operators and the order in which they are applied to numeric variables.

Most Arithmetic Operators

* / %	In the absence of parentheses, multiplication and division evaluate before addition and subtraction. In other words, *, /, and % have precedence over + and -. If more than one of these operators appears in an expression, the leftmost operator evaluates first.
+ –	In the absence of parentheses, + and - evaluate after all of the *, /, and % operators, with the leftmost evaluating first. Parentheses may override these precedence rules.

The operators of the following expression are applied to operands in this order: `/`, `+`, `-`.

```
2.0 + 5.0 - 8.0 / 4.0        // Evaluates to 5.0
```

Parentheses may alter the order in which arithmetic operators are applied to their operands.

```
(2.0 + 5.0 - 8.0) / 4.0      // Evaluates to -0.25
```

With parentheses, the `/` operator evaluates last, rather than first. The same set of oper-ators and operands, with parentheses added, has a different result (`-0.25` rather than `5.0`).

These precedence rules apply to binary operators only. A binary operator is one that requires one operand to the left and one operand to the right. A unary operator requires one operand on the right. Consider this expression, which has the binary multiplication operator * and the unary minus operator -.

```
3.5 * -2.0  // Evaluates to -7.0
```

The unary operator evaluates before the binary * operator: 3.5 times negative 2.0 results in negative 7.0. Two examples of arithmetic expressions are shown in the following complete program that computes the GPA for two courses.

```
// This program calculates the grade point average (GPA) for two courses.
import java.util.Scanner;

public class TwoCourseGPA {

  public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);

    // Prompt and Input the credits and grades for two courses
    System.out.print("Enter credits for first course: ");
    double credits1 = keyboard.nextDouble();
    System.out.print("Enter grade for first course: ");
    double grade1 = keyboard.nextDouble ();
    System.out.print("Enter credits for second course: ");
    double credits2 = keyboard.nextDouble ();
    System.out.print("Enter grade for second course: ");
    double grade2 = keyboard.nextDouble();

    // Compute the GPA
    double qualityPoints = (credits1 * grade1) + (credits2 * grade2);
    double GPA = qualityPoints / (credits1 + credits2);

    // Show the result
    System.out.println();
    System.out.println("GPA for these two courses: ");
    System.out.println(GPA);
  }
}
```

Output

```
Enter credits for first course: 3.0
Enter grade for first course: 4.0
Enter credits for second course: 2.0
Enter grade for second course: 3.0

GPA for these two courses
3.6
```

Self-Check

2-4. Write a complete Java program that prompts for a number from 0.0 to 1.0 and echos (prints) the user's input. The dialog generated by your program should look like this:

```
Enter relativeError [0.0 through 1.0]: 0.341
You entered: 0.341
```

2-5. Write the output generated by the following program:

```
public class Arithmetic {
  public static void main(String[] args) {
    double x = 1.2;
    double y = 3.4;
    System.out.println(x + y);
    System.out.println(x - y);
    System.out.println(x * y);
  }
}
```

2-6. Write the complete dialog (program output and user input) generated by the following program when the user enters each of these input values for `sale`:

a. ***10.00*** b. ***12.34*** c. ***100.00***

```
import java.util.Scanner;

public class InputProcessOutput  {

  public static void main(String[] args)  {
    double sale = 0.0;
    double tax = 0.0;
    double total = 0.0;
    double TAX_RATE = 0.07;
    Scanner keyboard = new Scanner(System.in);
    // I)nput
    System.out.print("Enter sale: ");
    sale = keyboard.nextDouble();  // User enters 10.00, 12.34, or 100.00
    // P)rocess
    tax = sale * TAX_RATE;
    total = sale + tax;

    // O)utput
    System.out.println("Sale: " + sale);
    System.out.println("Tax: " + tax);
    System.out.println("Total: " + total);
  }
}
```

2-7 Evaluate the following arithmetic expressions:

```
double x = 2.5;
double y = 3.0;
```

-a `x * y + 3.0`
-b `0.5 + x / 2.0`
-c `1.0 + x * 3.0 / y`
-d `1.5 * ( x - y )`
-e `y + -x`
-f `( x - 2.0 ) * ( y - 1.0 )`

`int` Arithmetic

A variable declared as `int` can store a limited range of whole numbers (numbers without fractions). Java `int` variables store integers in the range of -2,147,483,648 through 2,147,483,647 inclusive. All `int` variables have operations similar to `double` (`+`, `*`, `-`, `=`), but some differences do exist, and there are times when `int` is the correct choice over `double`. For example, a fractional remainder cannot be stored in an `int`. In fact, you cannot assign a floating-point literal or `double` variable to an `int` variable. If you do, the compiler complains with an error.

```
int anInt = 1.999;        // ERROR
int anotherInt = 0.0;     // ERROR
```

The / operator has different meanings for `int` and `double` operands. Whereas the result of 3 / 4 is 0, the result of 3.0 / 4.0 is 0.75. Two integer operands with the / operator have an integer result—not a floating-point result, as in the latter example. When writing programs, remember to choose an `int` or `double` data type correctly, in order to appropriately reflect the type of value you would like to store.

The remainder operation—symbolized with the % `(modulus)` operator—is also available for both `int` and `double operands`. For example, the result of 18 % 4 is the integer remainder after dividing 18 by 4, which is 2. Integer arithmetic is illustrated in the following code, which shows % and / operating on integer expressions, and / operating on floating-point operands. In this example, the integer results describe whole hours and whole minutes rather than the fractional equivalent.

```
public class DivMod  {
  public static void main(String[] args)  {
    int totalMinutes = 254;
    int hours = totalMinutes / 60;
    int minutes = totalMinutes % 60;
    System.out.println(totalMinutes + " minutes can be rewritten as ");
    System.out.println(hours + " hours and " + minutes + " minutes");
  }
}
```

Output

```
254 minutes can be rewritten as
4 hours and 14 minutes
```

The preceding program indicates that even though `int`s and `double`s are similar, there are times when `double` is the more appropriate type than `int`, and vice versa. The `double` type should be specified when you need a numeric variable with a fractional component. If you need a whole number, select `int`.

Mixing Integer and Floating-Point Operands

Whenever integer and floating-point values are on opposite sides of an arithmetic operator, the integer operand is **promoted** to its floating-point equivalent. The integer 6, for example, becomes 6.0, in the case of 6 / 3.0. The resulting expression is then a floating-point number, 2.0. The same rule applies when one operand is an `int` variable and the other a `double` variable. Here are a few examples of expression with the operands are a mix of `int` and `double`.

```
public class MixedOperands {

  public static void main(String[] args)  {
    int number = 9;
    double sum = 567.9;
    System.out.println(sum / number);  // Divide a double by an int
    System.out.println(number / 2);    // Divide an int by an int
    System.out.println(number / 2.0);  // Divide an int by a double
    System.out.println(2.0 * number);  // Result is a double: 18.0 not 18
  }
}
```

Output

```
63.099999999999994
4
4.5
18.0
```

Expressions with more than two operands will also evaluate to floating-point values if one of the operands is floating-point—for example, (8.8/4+3) = (2.2 + 3) = 5.2. Operator precedence rules also come into play—for example, (3 / 4 + 8.8) = (0 + 8.8) = 8.8.

Self-Check

2-8 Evaluate the following expressions.

- **-a** `5 / 9`
- **-b** `5.0 / 9`
- **-c** `5 / 9.0`
- **-d** 2 + 4 * 6 / 3
- **-e** (2 + 4) * 6 / 3
- **-f** `5 / 2`
- **-g** `7 / 2.5 * 3 / 4`
- **-h** `1 / 2.0 * 3`
- **-i** 5 / 9 * (50.0 - 32.0)
- **-j** 5 / 9.0 * (50 - 32)

The **boolean** Type

Java has a primitive `boolean` data type to store one of two `boolean` literals: `true` and `false`. Whereas arithmetic expressions evaluate to a number, `boolean` expressions, such as `credits > 60.0`, evaluate to one of these `boolean` values. A `boolean` expression often contains one of these relational operators:

Operator	Meaning
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

When a relational operator is applied to two operands that can be compared to one another, the result is one of two possible values: true or false. The next table shows some examples of simple `boolean` expressions and their resulting values.

Boolean Expression	Result
`double x = 4.0;`	
`x < 5.0`	`true`
`x > 5.0`	`false`
`x <= 5.0`	`true`
`5.0 == x`	`false`
`x != 5.0`	`true`

Like primitive numeric variables, `boolean` variables can be declared, initialized, and assigned a value. The assigned expression must be a `boolean` expression—thus, the result of the expression must also evaluate to true or false. This is shown in the initializations, assignments, and output of three boolean variables in the following code:

```
// Initialize three boolean variables to false
boolean ready = false;
boolean willing = false;
boolean able = false;
double credits = 28.5;
double hours = 9.5;
// Assign true or false to all three boolean variables
ready = hours >= 8.0;
willing = credits > 20.0;
able = credits <= 32.0;

System.out.println("ready: " + ready);
System.out.println("willing: " + willing);
System.out.println("able: " + able);
```

Output

```
ready: true
willing: true
able: true
```

Self-Check

2-9 Evaluate the following expressions to their correct value.

```
int j = 4;
int k = 7;
```

a. `(j + 4) == k`
b. `j == 0`
c. `j >= k`
d. `j != k`
e. `j < k`
f. `j == 4`
g. `j == (j + k - j)`
h. `(k - 5) <= (j + 2)`

Boolean Operators

Java has three boolean operators `!` to represent logical not, `||` to represent logical or, and `&&` to represent logical and. These three Boolean operators allow us to write more complex boolean expressions to express our intentions. For example, this boolean expression shows the boolean "and" operator (&&) applied to two boolean operands to determine if test is in the range of 0 through 100 inclusive.

```
(test >= 0) && (test <= 100)
```

Used in assertions, this Boolean expression evaluates to true when test is 97 and false when test is 977:

When `test` is `97`

```
(test >= 0) && (test <= 100)
( 97  >= 0) && ( 97  <= 100)
     true      &&       true
               true
```

When `test` is `977`

```
(test >= 0) && (test <= 100)
(977  >= 0) && (977  <= 100)
     true      &&      false
               false
```

Since there are only two Boolean values, true and false, the following table shows every possible combination of Boolean values and operators `!`, `||`, and `&&`:

`!` boolean "not" operator

Expression	Result
`! false`	`true`
`! true`	`false`

`||` boolean "or" operator

Expression	Result
`true  || true`	`true`
`true  || false`	`true`
`false || true`	`true`
`false || false`	`false`

`&&` boolean "and" operator

Expression	Result
`true && true`	`true`
`true && false`	`false`
`false && true`	`false`
`false && false`	`false`

Precedence Rules

Programming languages have **operator precedence** rules governing the order in which operators are applied to operands. For example, in the absence of parentheses, the relational operators >= and <= are evaluated before the `&&` operator. Most operators are grouped (evaluated) in a left-to-right order: `a/b/c/d` is equivalent to `(((a/b)/c)/d)`.

Table 6.1 lists some (though not all) of the Java operators in order of precedence. The dot `.` and `()` operators are evaluated first (have the highest precedence), and the assignment operator = is evaluated last. This table shows all of the operators used in this textbook (however, there are more).

Precedence rules for operators *some levels of priorities are not shown*

Precedence	Operator	Description	Associativity
1	`.`	Member reference	Left to right
	`()`	Method call	
2	`!`	Unary logical complement (”not”)	Right to left
	`+`	Unary plus	
	`-`	Unary minus	
3	`new`	Constructor of objects	
4	`*`	Multiplication	Left to right
	`/`	Division	
	`%`	Remainder	
5	`+`	Addition (for `int` and `double`)	Left to right
	`+`	String concatenation	
	`-`	Subtraction	
7	`<`	Less than	Left to right
	`<=`	Less than or equal to	
	`>`	Greater than	
	`>=`	Greater than or equal to	
8	`==`	Equal	Left to right
	`!=`	Not equal	
12	`&&`	Boolean “and”	Left to right
13	`||`	Boolean “or”	Left to right
15	`=`	Assignment	Right to left

These elaborate precedence rules are difficult to remember. If you are unsure, use parentheses to clarify these precedence rules. Using parentheses makes the code more readable and therefore more understandable that is more easily debugged and maintained.

Self-Check

2-10 Evaluate the following expressions to `true` or `false`:

a. `false || true`
b. `true && false`
c. `(1 * 3 == 4 || 2 != 2)`
d. `false || true && false`
e `3 < 4 && 3 != 4`
f. `!false && !true`
g. `(5 + 2 > 3 * 4) && (11 < 12)`
h. `! ((false && true) || false)`

2-11 Write an expression that is true only when an `int` variable named `score` is in the range of 1 through 10 inclusive.

Errors

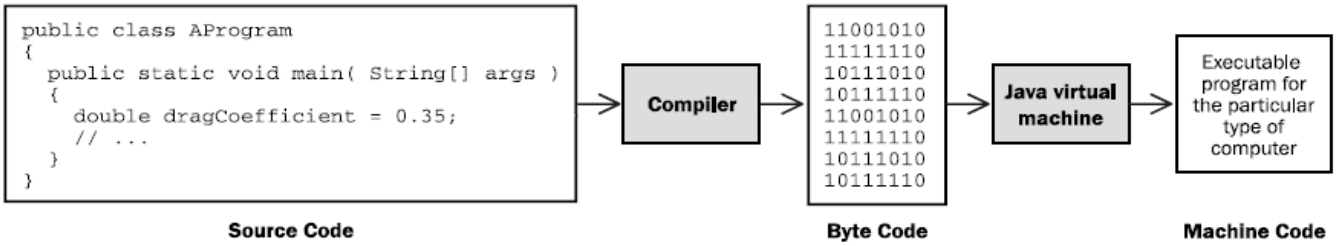
There are several categories of errors encountered when programming:

- **syntax errors**—errors that occur when compiling source code into byte code
- **intent errors**—the program does what you typed, not what you intended
- **exception**—errors that occur as the program executes

When programming, you will be writing source code using the syntax for the Java programming language. This source code is translated into byte code by the compiler, and is then stored in a `.class` file. The byte code is the same for each computer system.
For this byte code to execute, another program, called the Java virtual machine (JVM), translates the Java byte code into instructions understood by that computer. This extra step is necessary for one of the main advantages of Java: the same program can run in any computing environment! A computer might be

running Windows, MacOS, Solaris, Unix, or Linux—each computer system has its own Java virtual machine program. Having a particular Java virtual machine for each computer system also allows the same Java `.class` file to be transported around the Internet. The following figure shows the levels of translation needed in order to get executable programs to run on most computers.

From Source Code to a Program that Runs on Many Computers



1. The programmer translates algorithms into Java source code.
2. The compiler translates the source code into byte code.
3. The Java virtual machine translates byte code into the instructions understood by the computer system (Solaris, Unix, Linux, Mac OS, or Windows).

Syntax Errors Detected at Compile Time

When you are compiling source code or running your program on a computer, errors may crop up. The easiest errors to detect and fix are the errors generated by the **compiler**. These are **syntax errors** that occur during **compile time**, the time at which the compiler is examining your source code to detect and report errors, and/or to attempt to generate executable byte code from error-free source code.

A programming language requires strict adherence to its own set of formal syntax rules. It is not difficult for programmers to violate these syntax rules! All it takes is one missing `{` or `;` to foul things up. As you are writing your source code, you will often use the compiler to check the syntax of the code you wrote. While the Java compiler is translating source code into byte code so that it can run on a computer, it is also locating and reporting as many errors as possible. If you have any syntax errors, the byte code will not be generated—the program simply cannot run. If you are using the Eclipse integrated development, you will see compile time errors as you type, sometimes because you haven't finished what you were doing. To get a properly running program, you need to first correct ALL of your syntax errors.

Compilers generate many error messages. However, it is your source code that is the origin of these errors. Small typographical (and human) mistakes can be responsible for much larger roadblocks, from the compiler's perspective. Whenever your compiler appears to be nagging you, remember that the compiler is there to help you correct your errors!

The following program attempts to show several errors that the compiler should detect and report. Because error messages generated by compilers vary among systems, the reasons for the errors below are indexed with numbers to explanations that follow. Your system will certainly generate quite different error messages.

```
// This program attempts to convert pounds to UK notation.
// Several compile time errors have been intentionally retained.
public class CompileTimeErrors {

  public static void main(String[] args)  {
    System.out.println("Enter weight in pounds: ") ❶
    int pounds = keyboard.nextInt()❷;
    System.out.print("In the U.K. you weigh❸);
    System.out.print(❹Pounds / 14 + " stone, "❺pounds % 14);
  }
}
```

❶ A semicolon (`;`) is missing
❷ `keyboard` was not declared
❸ A double quote (") is missing
❹ `pounds` was written as `Pounds`
❺ The extra expressions require a missing concatenation symbol (+)

Syntax errors take some time to get used to, so try to be patient and observe the location where the syntax error occurred. The error is usually near the line where the error was detected, although you may have to fix preceding lines. Always remember to fix the first error first. An error that was reported on line 10 might be the result of a semicolon that was forgotten on line 5. The corrected source code, without error, is given next, followed by an interactive dialog (user input and computer output):

```
// This program converts pounds to the UK weight measurement.
import java.util.Scanner;

public class ErrorFree {

  public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);

    System.out.print("Enter weight in pounds: ");
    int pounds = keyboard.nextInt();
    System.out.print("In the U.K. you weigh ");
    System.out.println((pounds / 14) + " stone, " + (pounds % 14));
  }
}
```

Dialog

```
Enter weight in pounds: 146
In the U.K. you weigh 10 stone, 6
```

A different type of error occurs when `String[] args` is omitted from the `main` method:

```
  public static void main()
```

When the program tries to run, it looks for a method named `main` with `(String[] identifier)`. If you forget to write `String[] args`, you would get the error below shown after the program begins. The same error occurs if main has an uppercase M.

```
  Exception in thread "main" java.lang.NoSuchMethodError: main
```

This type of error, which occurs while the program is running, is known as an **exception**.

Exceptions

After your program compiles with no syntax errors, you will get a .class file containing the byte code that can be run on the Java virtual machine. The virtual machine can be invoked by issuing a Java command with the .class file name. For example, entering the command `java ErrorFree` at your operating system prompt will run the above program, assuming that you have a Java runtime environment (jre) installed on your computer and that the file ErrorFree.class exists.

However, when a program runs, errors may still occur. If the user enters a string that is supposed to be a number, what is the program to do? If the user enters "1oo" instead of "100" for example, is the program supposed to assume that the user meant 100? What should happen when the user enters "Kim" instead of a number? What should happen when an arithmetic expression results in division by zero? Or when there is an attempt to read from a file on a disk, but there is no disk in the drive, or the file name is wrong? Such events that occur while the program is running are known as exceptions.
One exception was shown above. The `main` method was valid, so the code compiled. However, when the program ran, Java's runtime environment was unable to locate a `main` method with `String[] args`. The error could not be discovered until the user ran the program, at which time Java began attempted to locate

the beginning of the program. If Java cannot find a method with the following line of code, a runtime exception occurs and the program terminates prematurely.

```
  public static void main(String[] args)
```

Now consider another example of an exception that occurs while the program is running. The output for the following code indicates that Java does not allow integer division by zero. The compiler does a lot of things, but it does not check the values of variables. If, at runtime, the denominator in a division happens to be 0, an `ArithmeticException` occurs.

```
public class AnArithmeticException {

  public static void main(String[] args) {
    // Integer division by zero throws an ArithmeticException
    int numerator = 5;
    int denominator = 0;
    int quotient = numerator / denominator; // A runtime error
    System.out.println("This message will not execute.");
  }
}
```

Output

```
   Exception in thread "main" java.lang.ArithmeticException: / by zero
      at A.main(A.java:8)
```

When you encounter one of these exceptions, consider the line number (`7`) where the error occurred. The reason for the exception (`/ by zero`) and the name of the exception (`ArithmeticException`) are two other clues to help you figure out what went wrong.

Intent Errors (Logic Errors)

Even when no syntax errors are found and no runtime errors occur, the program still may not execute properly. A program may run and terminate normally, but it may not be correct. Consider the following program:

```
// This program finds the average given the sum and the size
import java.util.Scanner;

public class IntentError {

  public static void main(String[] args) {
    double sum = 0.0;
    double average = 0.0;
    int number = 0;
    Scanner keyboard = new Scanner(System.in);
    // Input:
    System.out.print("Enter sum: ");
    sum = keyboard.nextDouble();
    System.out.print("Enter number: ");
    number = keyboard.nextInt();
    // Process
    average = number / sum;
    // Output
    System.out.println("Average: " + average);
  }
}
```

Dialog

```
Enter sum: 291
Enter number: 3
Average: 0.010309278350515464
```

Such intent errors occur when the program does what was typed, not what was intended. The compiler cannot detect such intent errors. The expression `number / sum` is syntactically correct—the compiler just has no way of knowing that this programmer intended to write `sum / number` instead.
Intent errors, also known as logic errors, are the most insidious and usually the most difficult errors to correct. They also may be difficult to detect—the user, tester, or programmer may not even know they exist! Consider the program controlling the Therac 3 cancer radiation therapy machine. Patients received massive overdoses of radiation resulting in serious injuries and death, while the indicator displayed everything as normal. Another infamous intent error involved a program controlling a probe that was supposed to go to Venus. Simply because a comma was missing in the Fortran source code, an American Viking Venus probe burnt up in the sun. Both programs had compiled successfully and were running at the time of the accidents. However, they did what the programmers had written—obviously not what was intended.

Answers to Self-Check Questions

2-1
- -a VALID
- -b can't start an identifier with digit 1
- -c VALID
- -d . is a special symbol.
- -e A space is not allowed.
- -f VALID
- -g ( ) are not allowed.
- -h VALID
- -i Periods (.) are not allowed.
- -j VALID
- -k Can't start identifiers with a digit.
- -l A space is not allowed.
- -m VALID but not very clear
- -n VALID but not very clear
- -o / is not allowed.
- -p VALID (but don't use it, Java already does)

2-2 Which of the following are valid Java comments?

-a	`// Is this a comment?`	Yes
-b	`/ / Is this a comment?`	No, there is a space between the slashes
-c	`/* Is this a comment?`	No, the closing `*/` is missing
-d	/* Is this a comment? */	Yes

2-3
- a VALID
- b attempts to assign a floating-point to an `int`.
- c attempts to assign a string to an `int`
- d VALID
- e VALID
- f valid
- g attempts to assign a string to a `double`.
- h VALID

2-4
```
import java.util.Scanner;
public class RelativeError {   // Your class name may vary
  public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
    System.out.print("Enter relativeError [0.0 through 1.0]: ");
    double relativeError = keyboard.nextDouble();
    System.out.print("You entered: " + relativeError);
  }
}
```

2-5
```
 4.6
-2.2
 4.08
```

2-6 a. ***10.00***
```
Enter sale: 10.00
Sale: 10.0
Tax: 0.7
Total: 10.7
```
b. ***12.34***
```
Enter sale: 12.34
Sale: 12.34
Tax: 0.8638
Total: 13.2038
```
c. ***100.00***
```
Enter sale: 100.00
Sale: 100.0
Tax: 7.0
Total: 107.0
```

2.7
- -a 10.5
- -b 1.75
- -c 3.5
- -d -0.75
- -e 0.5
- -f 1.0

2-8
- -a 0
- -b 0.55556
- -c 0.55556
- -d 10
- -e 12
- -f 2
- -g 2.1
- -h 1.5
- -i 0.0 5/9 is 0, 0*18.0 is 0.0
- -j 10.0

2-9
- -a false
- -b false
- -c false
- -d true
- -e true
- -f true
- -g false
- -h true

2-10
- a. true
- b. false
- c. false
- d. false
- e true
- f. false
- g. false
- h. true

2-11 (score >= 1) && (score <= 10)