2.1 A Simple Java Program

- The following program displays the message `Java rules!` on the screen.

```java
// Displays the message "Java rules!"

public class JavaRules {
    public static void main(String[] args) {
        System.out.println("Java rules!");
    }
}
```
Java Programs in General

• Building blocks of a Java program:
  – **Classes.** A class is a collection of related variables and/or methods (usually both). A Java program consists of one or more classes.
  – **Methods.** A method is a series of statements. Each class may contain any number of methods.
  – **Statements.** A statement is a single command. Each method may contain any number of statements.
2.2 Executing a Java Program

• Steps involved in executing a Java program:
  – Enter the program
  – Compile the program
  – Run the program
Chapter 2: Writing Java Programs

Diagram:

1. Editor
   - Java source code
2. Compiler
   - Class file
3. Interpreter for Java Virtual Machine
   - Output of program
Integrated Development Environments

• An *integrated development environment* (IDE) is an integrated collection of software tools for developing and testing programs.

• A typical IDE includes at least an editor, a compiler, and a debugger.

• A programmer can write a program, compile it, and execute it, all without leaving the IDE.

• We will use Eclipse IDE.
2.3 Program Layout

• The JavaRules program raises a couple of issues:
  – Why do we put comments into programs, and what are the rules for doing so?
  – How should we “lay out” a program? Does it matter where we put spaces and blank lines? Where should the curly braces go?
Comments

• *Comments* are an important part of every program.
• They provide information that’s useful for anyone who will need to read the program in the future.
• Typical uses of comments:
  – To document who wrote the program, when it was written, what changes have been made to it, and so on.
  – To describe the behavior or purpose of a particular part of the program, such as a variable or method.
  – To describe how a particular task was accomplished, which algorithms were used, or what tricks were employed to get the program to work.
Types of Comments

• Single-line comments:
  // Comment style 1

• Multiline comments:
  /* Comment style 2 */

• “Doc” comments:
  /** Comment style 3 */

• Doc comments are designed to be extracted by a special program, javadoc.
Problems with Multiline Comments

- Forgetting to terminate a multiline comment may cause the compiler to ignore part of a program:

  ```java
  System.out.print("My ");   /* forgot to close this comment...  
  System.out.print("cat ");
  System.out.print("has ");   /* so it ends here */
  System.out.println("fleas");
  ```
Single-line Comments

• Many programmers prefer `//` comments to `/* ... */` comments, for several reasons:
  – Ease of use
  – Safety
  – Program readability
  – Ability to “comment out” portions of a program
Tokens

- A Java compiler groups the characters in a program into *tokens*.
- The compiler then puts the tokens into larger groups (such as statements, methods, and classes).
- Tokens in the `JavaRules` program:

```java
public class JavaRules { public static void main ( String [ ] args ) { System . out . println ( "Java rules!" ) ; } }
```
Avoiding Problems with Tokens

• Always leave at least one space between tokens that would otherwise merge together:
  ```java
  public class JavaRules {
  ```

• Don’t put part of a token on one line and the other part on the next line:
  ```java
  public class JavaRules {
  ```
Chapter 2: Writing Java Programs

Indentation

• Programmers use indentation to indicate nesting.
• An increase in the amount of indentation indicates an additional level of nesting.
• The JavaRules program consists of a statement nested inside a method nested inside a class:

```java
public class JavaRules {
    public static void main(String[] args) {
        System.out.println("Java rules!");
    }
}
```
How Much Indentation?

• Common amounts of indentation:
  – 2 spaces: the bare minimum
  – 3 spaces: the optimal amount
  – 4 spaces: what many programmers use
  – 8 spaces: probably too much

• The JavaRules program with an indentation of four spaces:

```java
public class JavaRules {
    public static void main(String[] args) {
        System.out.println("Java rules!");
    }
}
```
Brace Placement

• Brace placement is another important issue.
• One technique is to put each left curly brace at the end of a line. The matching right curly brace is lined up with the first character on that line:

```java
public class JavaRules {
    public static void main(String[] args) {
        System.out.println("Java rules!");
    }
}
```
Brace Placement

• Some programmers prefer to put left curly braces on separate lines:

```java
public class JavaRules {
    public static void main(String[] args) {
        System.out.println("Java rules!");
    }
}
```

• This makes it easier to verify that left and right braces match up properly. However, program files become longer because of the additional lines.
2.4 Using Variables

- In Java, every variable must be declared before it can be used.
- Declaring a variable means informing the compiler of the variable’s name and its properties, including its type.
- int is one of Java’s types. Variables of type int can store integers (whole numbers).
Declaring Variables

• Form of a *variable declaration*:
  – The type of the variable
  – The name of the variable
  – A semicolon

• Example:
  ```java
  int i;  // Declares i to be an int variable
  ```

• Several variables can be declared at a time:
  ```java
  int i, j, k;
  ```

• It’s often better to declare variables individually.
Initializing Variables

• A variable is given a value by using =, the assignment operator:
  \[ i = 0; \]

• Initializing a variable means to assign a value to the variable for the first time.

• Variables always need to be initialized before the first time their value is used.

• The Java compiler checks that variables declared in methods are initialized prior to their first use.
Initializers

• Variables can be initialized at the time they’re declared:
  
  int i = 0;
  
  0 is said to be the initializer for i.

• If several variables are declared at the same time, each variable can have its own initializer:
  
  int i = 0, j, k = 1;
Changing the Value of a Variable

• The assignment operator can be used both to initialize a variable and to change the value of the variable later in the program:

```java
i = 1;  // Value of i is now 1
...
i = 2;  // Value of i is now 2
```
Program: Printing a Lottery Number

```
Lottery.java

// Displays the winning lottery number

public class Lottery {
    public static void main(String[] args) {
        int winningNumber = 973;
        System.out.print("The winning number ");
        System.out.print("in today's lottery is ");
        System.out.println(winningNumber);
    }
}
```
2.5 Types

• A partial list of Java types:
  int — An integer
  double — A floating-point number
  boolean — Either true or false
  char — A character

• Declarations of double, boolean, and char variables:
  double x, y;
  boolean b;
  char ch;
Literals

• A literal is a token that represents a particular number or other value.

• Examples of int literals:
  0  297  30303

• Examples of double literals:
  48.0  48.  4.8e1  4.8e+1  .48e2  480e-1

• The only boolean literals are true and false.

• char literals are enclosed within single quotes:
  'a'  'z'  'A'  'Z'  '0'  '9'  '%'.  '..'  ' '
Using Literals as Initializers

- Literals are often used as initializers:

```java
double x = 0.0, y = 1.0;
boolean b = true;
char ch = 'f';
```
2.6 Identifiers

- **Identifiers** (names chosen by the programmer) are subject to the following rules:
  - Identifiers may contain letters (both uppercase and lowercase), digits, and underscores (_).
  - Identifiers begin with a letter or underscore.
  - There’s no limit on the length of an identifier.
  - Lowercase letters are not equivalent to uppercase letters. (A language in which the case of letters matters is said to be *case-sensitive.*)
Multiword Identifiers

- When an identifier consists of multiple words, it’s important to mark the boundaries between words.
- One way to break up long identifiers is to use underscores between words:
  ```java
  last_index_of
  ```
- Another technique is to capitalize the first letter of each word after the first:
  ```java
  lastIndexOf
  ```

This technique is the one commonly used in Java.
Conventions

• A rule that we agree to follow, even though it’s not required by the language, is said to be a convention.

• A common Java convention is beginning a class name with an uppercase letter:
  
  Color
  FontMetrics
  String

• Names of variables and methods, by convention, never start with an uppercase letter.
Keywords

- The following **keywords** can’t be used as identifiers because Java has already given them a meaning:

<table>
<thead>
<tr>
<th>abstract</th>
<th>double</th>
<th>int</th>
<th>super</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>else</td>
<td>interface</td>
<td>switch</td>
</tr>
<tr>
<td>break</td>
<td>extends</td>
<td>long</td>
<td>synchronized</td>
</tr>
<tr>
<td>byte</td>
<td>final</td>
<td>native</td>
<td>this</td>
</tr>
<tr>
<td>case</td>
<td>finally</td>
<td>new</td>
<td>throw</td>
</tr>
<tr>
<td>catch</td>
<td>float</td>
<td>package</td>
<td>throws</td>
</tr>
<tr>
<td>char</td>
<td>for</td>
<td>private</td>
<td>transient</td>
</tr>
<tr>
<td>class</td>
<td>goto</td>
<td>protected</td>
<td>try</td>
</tr>
<tr>
<td>const</td>
<td>if</td>
<td>public</td>
<td>void</td>
</tr>
<tr>
<td>continue</td>
<td>implements</td>
<td>return</td>
<td>volatile</td>
</tr>
<tr>
<td>default</td>
<td>import</td>
<td>short</td>
<td>while</td>
</tr>
<tr>
<td>do</td>
<td>instanceof</td>
<td>static</td>
<td></td>
</tr>
</tbody>
</table>

- `null`, `true`, and `false` are also reserved.
2.7 Performing Calculations

• In general, the right side of an assignment can be an expression.
• A literal is an expression, and so is a variable.
• More complicated expressions are built out of operators and operands.
• In the expression $5 / 9$, the operands are 5 and 9, and the operator is /.
• The operands in an expression can be variables, literals, or other expressions.
Operators

• Java’s arithmetic operators:
  +   Addition
  −   Subtraction
  *   Multiplication
  /   Division
  %   Remainder

• Examples:
  6 + 2  ⇒  8
  6 − 2  ⇒  4
  6 * 2  ⇒  12
  6 / 2  ⇒  3
Integer Division

• If the result of dividing two integers has a fractional part, Java throws it away (we say that it truncates the result).

• Examples:
  
  \[
  1 \div 2 \Rightarrow 0 \\
  5 \div 3 \Rightarrow 1
  \]
double Operands

• +, −, *, and / accept double operands:
  6.1 + 2.5 ⇒ 8.6
  6.1 − 2.5 ⇒ 3.6
  6.1 * 2.5 ⇒ 15.25
  6.1 / 2.5 ⇒ 2.44

• int and double operands can be mixed:
  6.1 + 2 ⇒ 8.1
  6.1 − 2 ⇒ 4.1
  6.1 * 2 ⇒ 12.2
  6.1 / 2 ⇒ 3.05
Binary Operators

• The +, −, *, and / operators are said to be binary operators, because they require two operands.
• There’s one other binary arithmetic operator: % (remainder).
• The % operator produces the remainder when the left operand is divided by the right operand:
  13 % 3 ⇒ 1
• % is normally used with integer operands.
• It’s a good idea to put a space before and after each binary operator.
Unary Operators

• Java also has two *unary* arithmetic operators:
  +  Plus
  –  Minus

• Unary operators require just one operand.

• The unary + and – operators are often used in conjunction with literals (−3, for example).
Round-Off Errors

• Calculations involving floating-point numbers can sometimes produce surprising results.

• If $d$ is declared as follows, its value will be 0.09999999999999987 rather than 0.1:

  ```java
double d = 1.2 - 1.1;
```

• *Round-off errors* such as this occur because some numbers (1.2 and 1.1, for example) can’t be stored in *double* form with complete accuracy.
Operator Precedence

• What’s the value of $6 + 2 \times 3$?
  – $(6 + 2) \times 3$, which yields 24?
  – $6 + (2 \times 3)$, which yields 12?

• Operator precedence resolves issues such as this.

• $\times$, $/$, and $\%$ take precedence over $+$ and $-$.

• Examples:
  
  
  
  \[
  5 + 2 \div 2 \Rightarrow 5 + (2 \div 2) \Rightarrow 6
  \]
  
  \[
  8 \times 3 - 5 \Rightarrow (8 \times 3) - 5 \Rightarrow 19
  \]
  
  \[
  6 - 1 \times 7 \Rightarrow 6 - (1 \times 7) \Rightarrow -1
  \]
  
  \[
  9 \div 4 + 6 \Rightarrow (9 \div 4) + 6 \Rightarrow 8
  \]
  
  \[
  6 + 2 \% 3 \Rightarrow 6 + (2 \% 3) \Rightarrow 8
  \]
Associativity

- Precedence rules are of no help when it comes to determining the value of 1 - 2 - 3.
- Associativity rules come into play when precedence rules alone aren’t enough.
- The binary +, -, *, /, and % operators are all left associative:
  
  \[
  2 + 3 - 4 \Rightarrow (2 + 3) - 4 \Rightarrow 1 \\
  2 * 3 / 4 \Rightarrow (2 * 3) / 4 \Rightarrow 1
  \]
Parentheses in Expressions

- Parentheses can be used to override normal precedence and associativity rules.
- Parentheses in the expression \((6 + 2) * 3\) force the addition to occur before the multiplication.
- It’s often a good idea to use parentheses even when they’re not strictly necessary:
  \[(x * x) + (2 * x) - 1\]
- However, don’t use too many parentheses:
  \[((x) * (x)) + ((2) * (x)) - (1)\]
Assignment Operators

• The assignment operator (\(=\)) is used to save the result of a calculation in a variable:

\[ \text{area} = \text{height} \times \text{width}; \]

• The type of the expression on the right side of an assignment must be appropriate for the type of the variable on the left side of the assignment.

• Assigning a `double` value to an `int` variable is not legal. Assigning an `int` value to a `double` variable is OK, however.
Using Assignment to Modify a Variable

• Assignments often use the old value of a variable as part of the expression that computes the new value.

• The following statement adds 1 to the variable \(i\): 

\[ i = i + 1; \]
Chapter 2: Writing Java Programs

Compound Assignment Operators

• The *compound assignment operators* make it easier to modify the value of a variable.

• A partial list of compound assignment operators:

  += Combines addition and assignment
  -= Combines subtraction and assignment
  *= Combines multiplication and assignment
  /= Combines division and assignment
  %= Combines remainder and assignment
Compound Assignment Operators

- **Examples:**
  
  ```java
  i += 2;  // Same as i = i + 2;
  i -= 2;  // Same as i = i - 2;
  i *= 2;  // Same as i = i * 2;
  i /= 2;  // Same as i = i / 2;
  i %= 2;  // Same as i = i % 2;
  ```
Program: Converting from Fahrenheit to Celsius

```
// Converts a Fahrenheit temperature to Celsius

public class FtoC {
    public static void main(String[] args) {
        double fahrenheit = 98.6;
        double celsius = (fahrenheit - 32.0) * (5.0 / 9.0);
        System.out.print("Celsius equivalent: ");
        System.out.println(celsius);
    }
}
```

FtoC.java
2.8 Constants

• A constant is a value that doesn’t change during the execution of a program.

• Constants can be named by assigning them to variables:

```java
double freezingPoint = 32.0;
double degreeRatio = 5.0 / 9.0;
```

• To prevent a constant from being changed, the word final can be added to its declaration:

```java
final double freezingPoint = 32.0;
final double degreeRatio = 5.0 / 9.0;
```
Naming Constants

• The names of constants are often written entirely in uppercase letters, with underscores used to indicate boundaries between words:

```java
final double FREEZING_POINT = 32.0;
final double DEGREE_RATIO = 5.0 / 9.0;
```
Adding Constants to the FtoC Program

FtoC2.java

// Converts a Fahrenheit temperature to Celsius

public class FtoC2 {
    public static void main(String[] args) {
        final double FREEZING_POINT = 32.0;
        final double DEGREE_RATIO = 5.0 / 9.0;
        double fahrenheit = 98.6;
        double celsius =
            (fahrenheit - FREEZING_POINT) * DEGREE_RATIO;
        System.out.print("Celsius equivalent: ");
        System.out.println(celsius);
    }
}

Copyright © 2000 W. W. Norton & Company. All rights reserved.
Advantages of Naming Constants

• Advantages of naming constants:
  – Programs are easier to read. The alternative is a program full of “magic numbers.”
  – Programs are easier to modify.
  – Inconsistencies and typographical errors are less likely.

• Always create meaningful names for constants. There’s no point in defining a constant whose name signifies its value:

```java
final int TWELVE = 12;
```
2.9 Methods

- A *method* is a series of statements that can be executed as a unit.
- A method does nothing until it is activated, or *called*.
- To call a method, we write the name of the method, followed by a pair of parentheses.
- The method’s *arguments* (if any) go inside the parentheses.
- A call of the `println` method:
  ```java
  System.out.println("Java rules!");
  ```
Methods in the Math Class

- The Math class contains a number of methods for performing mathematical calculations.
- These methods are called by writing Math.name, where name is the name of the method.
- The methods in the Math class return a value when they have completed execution.
The **pow** and **sqrt** Methods

- **The pow method** raises a number to a power:
  
  Math.pow(2.0, 3.0) \Rightarrow 8.0  
  Math.pow(-2.0, 3.0) \Rightarrow -8.0  
  Math.pow(2.0, -1.0) \Rightarrow 0.5  

- **The sqrt method** computes the square root of a number:
  
  Math.sqrt(2.0) \Rightarrow 1.4142135623730951  
  Math.sqrt(4.0) \Rightarrow 2.0  

- **Both pow and sqrt return values of type double.**
The *abs* and *max* Methods

- The *abs* method computes the absolute value of a number:
  
  
  \[
  \begin{align*}
  \text{Math.abs}(2.0) & \Rightarrow 2.0 \\
  \text{Math.abs}(-2.0) & \Rightarrow 2.0 \\
  \text{Math.abs}(2) & \Rightarrow 2 \\
  \text{Math.abs}(-2) & \Rightarrow 2
  \end{align*}
  \]

- The *max* method finds the larger of two numbers:
  
  \[
  \begin{align*}
  \text{Math.max}(3.0, 5.5) & \Rightarrow 5.5 \\
  \text{Math.max}(10.0, -2.0) & \Rightarrow 10.0 \\
  \text{Math.max}(12, -23) & \Rightarrow 12 \\
  \text{Math.max}(-5, -2) & \Rightarrow -2
  \end{align*}
  \]
The min Method

• The min method finds the smaller of two numbers:
  Math.min(3.0, 5.5) ⇒ 3.0
  Math.min(10.0, -2.0) ⇒ -2.0
  Math.min(12, -23) ⇒ -23
  Math.min(-5, -2) ⇒ -5

• The value returned by abs, max, and min depends on the type of the argument:
  – If the argument is an int, the methods return an int.
  – If the argument is a double, the methods return a double.
The round Method

• The `round` method rounds a double value to the nearest integer:
  Math.round(4.1) ⇒ 4
  Math.round(4.5) ⇒ 5
  Math.round(4.9) ⇒ 5
  Math.round(5.5) ⇒ 6
  Math.round(-4.1) ⇒ -4
  Math.round(-4.5) ⇒ -4
  Math.round(-4.9) ⇒ -5
  Math.round(-5.5) ⇒ -5

• `round` returns a long value rather than an int value.
Using the Result of a Method Call

• The value returned by a method can be saved in a variable for later use:

```java
double y = Math.abs(x);
```

• Another option is to use the result returned by a method directly, without first saving it in a variable. For example, the statements

```java
double y = Math.abs(x);
double z = Math.sqrt(y);
```

can be combined into a single statement:

```java
double z = Math.sqrt(Math.abs(x));
```
Using the Result of a Method Call

• Values returned by methods can also be used as operands in expressions.

• Example (finding the roots of a quadratic equation):
  
  ```java
  double root1 =
      (-b + Math.sqrt(b * b - 4 * a * c)) / (2 * a);
  double root2 =
      (-b - Math.sqrt(b * b - 4 * a * c)) / (2 * a);
  ```

• Because the square root of $b^2 - 4ac$ is used twice, it would be more efficient to save it in a variable:
  
  ```java
  double discriminant = Math.sqrt(b * b - 4 * a * c);
  double root1 = (-b + discriminant) / (2 * a);
  double root2 = (-b - discriminant) / (2 * a);
  ```
Using the Result of a Method Call

- The value returned by a method can be printed without first being saved in a variable:

```java
System.out.println(Math.sqrt(2.0));
```
2.10 Input and Output

• Most programs require both input and output.
• Input is any information fed into the program from an outside source.
• Output is any data produced by the program and made available outside the program.
Displaying Output on the Screen

- Properties of `System.out.print` and `System.out.println`:
  - Can display any single value, regardless of type.
  - The argument can be any expression, including a variable, literal, or value returned by a method.
  - `println` always advances to the next line after displaying its argument; `print` does not.
Displaying a Blank Line

• One way to display a blank line is to leave the parentheses empty when calling `println`:
  
  ```java
  System.out.println("Hey Joe");
  System.out.println();  // Write a blank line
  ```

• The other is to insert `\n` into a string that’s being displayed by `print` or `println`:
  
  ```java
  System.out.println("A hop,\n  a skip,\n  \n  and a jump");
  ```

  Each occurrence of `\n` causes the output to begin on a new line.
Escape Sequences

• The backslash character combines with the character after it to form an escape sequence: a combination of characters that represents a single character.

• The backslash character followed by \n forms \n, the new-line character.
Escape Sequences

• Another common escape sequence is \", which represents " (double quote):
  System.out.println("He yelled \"Stop!\" and we stopped.");

• In order to print a backslash character as part of a string, the string will need to contain two backslash characters:
  System.out.println("APL\\360");
Printing Multiple Items

• The + operator can be used to combine multiple items into a single string for printing purposes:

```java
System.out.println("Celsius equivalent: " + celsius);
```

• At least one of the two operands for the + operator must be a string.
import java.util.Scanner;

public class ScanProgram1 {
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = sc.nextLine();
        System.out.println("Hello, your name is"+name);
    }
}

import java.util.Scanner;

public class ScanProgram2 {
    public static void main(String args[]) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int num1 = sc.nextInt();
        System.out.print("Enter another number: ");
        int num2 = sc.nextInt();
        int result = num1 + num2;
        System.out.println("The sum of "+num1+
            " and "+num2+" is "+result);
    }
}

Chapter 2: Writing Java Programs

Obtaining Input from the User
Program: Converting from Fahrenheit to Celsius (Revisited)

`FtoC3.java`

// Converts a Fahrenheit temperature entered by the user to Celsius

import java.util.Scanner;

public class FtoC3 {
    public static void main(String[] args) {
        final double FREEZING_POINT = 32.0;
        final double DEGREE_RATIO = 5.0 / 9.0;

        Scanner sc = new Scanner(System.in);
        System.out.print("Enter Fahrenheit temperature: ");
        double fahrenheit = sc.nextDouble();
        double celsius =
            (fahrenheit - FREEZING_POINT) * DEGREE_RATIO;
        System.out.println("Celsius equivalent: " + celsius);
    }
}
2.11 Case Study: Computing a Course Average

• The CourseAverage program will calculate a class average, using the following percentages:
  Programs  30%
  Quizzes   10%
  Test 1    15%
  Test 2    15%
  Final exam 30%

• The user will enter the grade for each program (0–20), the score on each quiz (0–10), and the grades on the two tests and the final (0–100). There will be eight programs and five quizzes.
Output of the CourseAverage Program

Welcome to the CSc 2310 average calculation program.

Enter Program 1 score: 20
Enter Program 2 score: 19
Enter Program 3 score: 15
Enter Program 4 score: 18.5
Enter Program 5 score: 20
Enter Program 6 score: 20
Enter Program 7 score: 18
Enter Program 8 score: 20

Enter Quiz 1 score: 9
Enter Quiz 2 score: 10
Enter Quiz 3 score: 5.5
Enter Quiz 4 score: 8
Enter Quiz 5 score: 9.5

Enter Test 1 score: 78
Enter Test 2 score: 92
Enter Final Exam score: 85

Course average: 88
Design of the CourseAverage Program

1. Print the introductory message ("Welcome to the CSc 2310 average calculation program").
2. Prompt the user to enter eight program scores.
3. Compute the program average from the eight scores.
4. Prompt the user to enter five quiz scores.
5. Compute the quiz average from the five scores.
6. Prompt the user to enter scores on the tests and final exam.
7. Compute the course average from the program average, quiz average, test scores, and final exam score.
8. Round the course average to the nearest integer and display it.
Design of the CourseAverage Program

- **double** variables can be used to store scores and averages.
- Computing the course average involves scaling the program average and quiz average so that they lie between 0 and 100:

  \[
  \text{courseAverage} = \frac{1}{5} \times 0.3 \times \text{programAverage} + \frac{1}{10} \times 0.1 \times \text{quizAverage} + 0.15 \times \text{test1} + 0.15 \times \text{test2} + 0.3 \times \text{finalExam}
  \]

- **Math.round** can be used to round the course average to the nearest integer.
CourseAverage.java

// Program name: CourseAverage
// Author: K. N. King
// Written: 1998-04-05
// Modified: 1999-01-27

// Prompts the user to enter eight program scores (0-20), five quiz scores (0-10), two test scores (0-100), and a final exam score (0-100). Scores may contain digits after the decimal point. Input is not checked for validity. Displays the course average, computed using the following formula:

// Programs 30%
// Quizzes 10%
// Test 1 15%
// Test 2 15%
// Final exam 30%

// The course average is rounded to the nearest integer.
import java.util.Scanner;

public class CourseAverage {
    public static void main(String[] args) {
        // Print the introductory message
        System.out.println("Welcome to the CSc 2310 average " + "calculation program.\n");

        Scanner sc = new Scanner(System.in);
        // Prompt the user to enter eight program scores
        System.out.print("Enter Program 1 score: ");
        double program1 = sc.nextDouble();

        System.out.print("Enter Program 2 score: ");
        double program2 = sc.nextDouble();

        System.out.print("Enter Program 3 score: ");
        double program3 = sc.nextDouble();
    }
}
System.out.print("Enter Program 4 score: ");
double program4 = sc.nextDouble();

System.out.print("Enter Program 5 score: ");
double program5 = sc.nextDouble();

System.out.print("Enter Program 6 score: ");
double program6 = sc.nextDouble();

System.out.print("Enter Program 7 score: ");
double program7 = sc.nextDouble();

System.out.print("Enter Program 8 score: ");
double program8 = sc.nextDouble();
// Compute the program average from the eight scores
double programAverage =
  (program1 + program2 + program3 + program4 +
   program5 + program6 + program7 + program8) / 8;

// Prompt the user to enter five quiz scores
System.out.print("Enter Quiz 1 score: ");
double quiz1 = sc.nextDouble();

System.out.print("Enter Quiz 2 score: ");
double quiz2 = sc.nextDouble();

System.out.print("Enter Quiz 3 score: ");
double quiz3 = sc.nextDouble();
System.out.print("Enter Quiz 4 score: ");
double quiz4 = sc.nextDouble();

System.out.print("Enter Quiz 5 score: ");
double quiz5 = sc.nextDouble();

// Compute the quiz average from the five scores
double quizAverage =
    (quiz1 + quiz2 + quiz3 + quiz4 + quiz5) / 5;

// Prompt the user to enter scores on the tests and final
// exam
System.out.print("\nEnter Test 1 score: ");
double test1 = sc.nextDouble();
Chapter 2: Writing Java Programs

System.out.print("Enter Test 2 score: ");
double test2 = sc.nextDouble();

System.out.print("Enter Final Exam score: ");
double finalExam = sc.nextDouble();

// Compute the course average from the program average, 
// quiz average, test scores, and final exam score. 
// The program average (0-20) is multiplied by 5 to put 
// it on a scale of 0 to 100. The quiz average (0-10) is 
// multiplied by 10 for the same reason.
double courseAverage = .30 * programAverage * 5 + 
    .10 * quizAverage * 10 + 
    .15 * test1 + 
    .15 * test2 + 
    .30 * finalExam;
// Round the course average to the nearest integer and display it
System.out.println("\nCourse average: " + 
        Math.round(courseAverage));
}
Style Issues

- Style issues raised by the `CourseAverage` program:
  - Comment blocks
  - Blank lines
  - Short comments
  - Long lines
Improving the Program

The number of variables in `CourseAverage` can be reduced by keeping a “running total” of all scores entered so far, rather than storing each score in a separate variable:

```java
// Prompt the user to enter eight program scores.
System.out.print("Enter Program 1 score: ");
double programTotal = sc.nextDouble();

System.out.print("Enter Program 2 score: ");
programTotal += sc.nextDouble();
...

// Compute the program average from the eight scores.
double programAverage = programTotal / 8;
```
2.12 Debugging

• **Debugging** is the process of finding bugs in a program and fixing them.

• Types of errors:
  – Compile-time errors
  – Run-time errors (called *exceptions* in Java)
  – Incorrect behavior
Fixing Compile-Time Errors

• Strategies for fixing compile-time errors:
  – Read error messages carefully. Example:

    Buggy.java:8: Undefined variable: i
    System.out.println(i);
    ^

    Buggy.java:10: Variable j may not have been initialized
    System.out.println(j);
    ^

  – Pay attention to line numbers.
  – Fix the first error.
Fixing Compile-Time Errors

- Don’t trust the compiler (completely). The error isn’t always on the line reported by the compiler. Also, the error reported by the compiler may not accurately indicate the nature of the error. Example:

  System.out.print("Value of i: ")
  System.out.println(i);

  A semicolon is missing at the end of the first statement, but the compiler reports a different error:

  Buggy.java:8: Invalid type expression.
       System.out.print("Value of i: ")
   ^

  Buggy.java:9: Invalid declaration.
       System.out.println(i);
   ^
Fixing Run-Time Errors

• When a run-time error occurs, a message will be displayed on the screen. Example:

  Exception in thread "main"
  java.lang.NumberFormatException: foo
  at java.lang.Integer.parseInt(Compiled Code)
  at java.lang.Integer.parseInt(Integer.java:458)
  at Buggy.main(Buggy.java:11)

• Once we know what the nature of the error is and where the error occurred, we can work backwards to determine what caused the error.
Fixing Behavioral Errors

• Errors of behavior are the hardest problems to fix, because the problem probably lies either in the original algorithm or in the translation of the algorithm into a Java program.

• Other than simply checking and rechecking the algorithm and the program, there are two approaches to locating the source of a behavioral problem, depending on whether a debugger is available.
Using a Debugger

• A **debugger** doesn’t actually locate and fix bugs. Instead, it allows the programmer to see inside a program as it executes.

• Things to look for while debugging:
  – Order of statement execution
  – Values of variables

• Key features of a debugger:
  – Step
  – Breakpoint
  – Watch
Debugging Without a Debugger

• The JDK includes a debugger, named jdb.
• A debugger isn’t always necessary, however.
• If a run-time error occurs in a Java program, the message displayed by the Java interpreter may be enough to identify the bug.
• Also, `System.out.println` can be used to print the values of variables for the purpose of debugging:

```java
System.out.println("Value of a: "+a+
                      " Value of b: "+b);
```
Choosing Test Data

• Testing a program usually requires running it more than once, using different input each time.
• One strategy, known as boundary-value testing, involves entering input at the extremes of what the program considers to be legal.
• Boundary-value testing is both easy to do and surprisingly good at revealing bugs.