Chapter 3: Control Statements (Part I)

Java™ How to Program, 8/e

Instructor: Yuqun Zhang

Objectives

- To learn and use basic problem-solving techniques
- To develop algorithms using pseudo codes
- To use `if` and `if…else` selection statements
- To use `while` repetition statement

Programming like a Professional

- Before writing a program, you should have a thorough understanding of the problem and a carefully planned approach to solving it.
- While writing a program, you should know the techniques available and use proven program construction techniques.

Algorithms

- Any computing problem can be solved by executing a series of actions in a specific order
- An algorithm is a procedure for solving a problem in terms of
  - the actions to execute and
  - the order in which these actions execute
- The "rise-and-shine algorithm" for an executive: (1) get out of bed; (2) take off pajamas; (3) take a shower; (4) get dressed; (5) eat breakfast; (6) carpool to work.
- Specifying the order in which statements (actions) execute in a program is called program control.

Pseudocode

- Pseudocode is an informal language for developing algorithms
- Similar to everyday English
- Helps you "think out" a program

```
Start Program
Enter two numbers, A, B
Add the numbers together
Print Sum
End Program
```

- Pseudocode normally describes only statements representing the actions, e.g., input, output or calculations.
- Carefully prepared pseudocode can be easily converted to a corresponding Java program

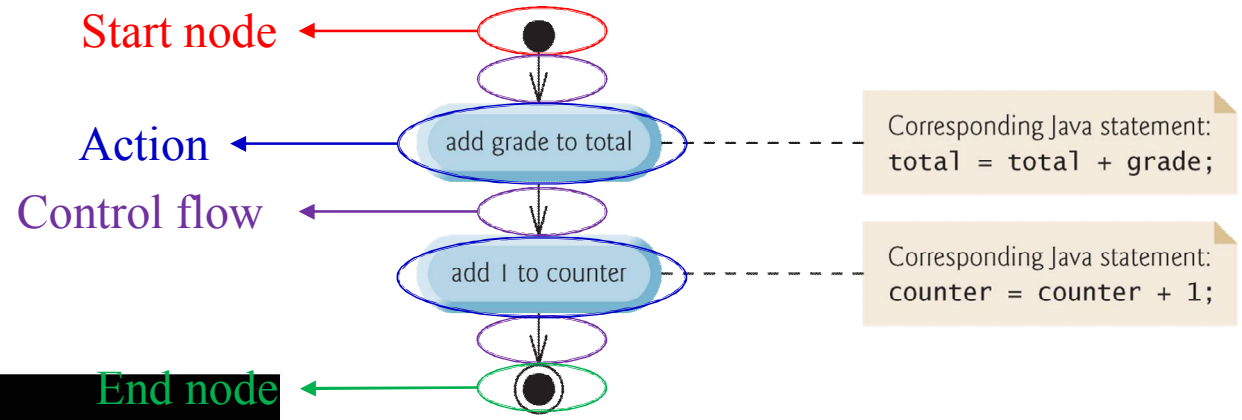
Control Structures

- Sequential execution: normally, statements in a program are executed one after the other in the order in which they are written.
- Transfer of control: various Java statements enable you to specify the next statement to execute, which is not necessarily the next one in sequence.
- All programs can be written in terms of only three control structures—the sequence structure, the selection structure and the repetition structure.

Sequence Structure

- Unless directed otherwise, computers execute Java statements one after the other in the order in which they're written.

- The activity diagram (a flowchart showing activities performed by a system) below illustrates a typical sequence structure in which two calculations are performed in order.



Selection Structure

- Three types of selection statements:
  - `if` statement
  - `if...else` statement
  - `switch` statement

Repetition Structure

- Three repetition statements (a.k.a., looping statements). Perform statements repeatedly while a loop-continuation condition remains true.
  - `while` statement
  - `for` statement
  - `do...while` statement

`if` Single-Selection Statement

- Pseudocode:

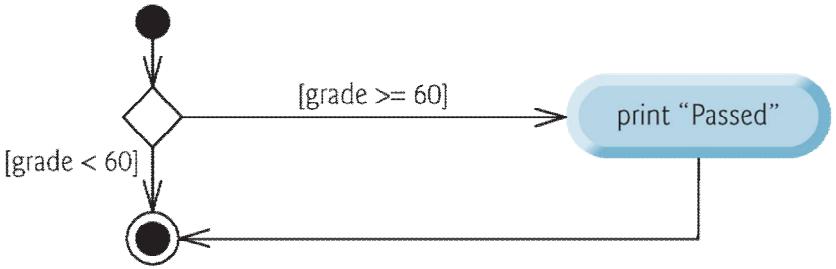
  If student's grade is greater than or equal to 60

  Print "Passed"

- Java code:

```
if ( studentGrade >= 60 )
   System.out.println( "Passed" );
```

if Single-Selection Statement



- Diamond, or decision symbol, indicates that a decision is to be made.
- Workflow continues along a path determined by the symbol's guard conditions, which can be true or false.
- Each transition arrow from a decision symbol has a guard condition.
- If a guard condition is true, the workflow enters the action state to which the transition arrow points .

`if…else` Double-Selection Statement

- Pseudocode:

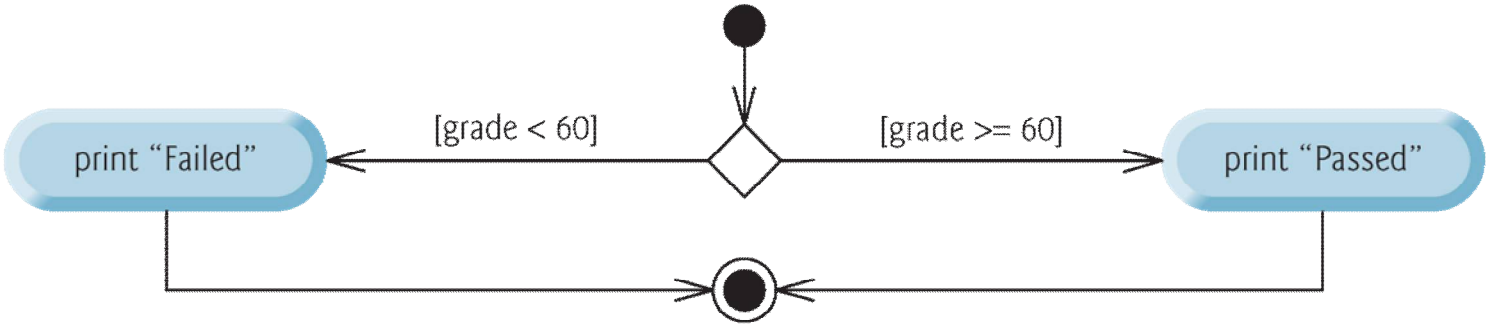
  If student's grade is greater than or equal to 60
  Print "Passed"
  Else
  Print "Failed"

- Java code:

```
if ( grade >= 60 )
   System.out.println( "Passed" );
else
   System.out.println( "Failed" );
```

`if…else` Double-Selection Statement

- The symbols in the UML activity diagram represent actions and decisions



Conditional operator ?:

```
String result = studentGrade >= 60 ? "Passed" : "Failed"
```

The operands and `?:` form a conditional expression.

Shorthand of `if...else`

Conditional operator ?:

```
String result = studentGrade >= 60 ? "Passed" : "Failed"
```

A `boolean` expression that evaluates to `true` or `false`

The conditional expression takes this value if the `boolean` expression evaluates to `true`

The conditional expression takes this value if the `boolean` expression evaluates to `false`

A More Complex Example

- Pseudocode:

If student's grade is greater than or equal to 90
*   Print "A"*
else
*   If student's grade is greater than or equal to 80*
*   Print "B"*
*   else*
*   If student's grade is greater than or equal to 70*
*   Print "C"*
*   else*
*   If student's grade is greater than or equal to 60*
*   Print "D"*
*   else*
*   Print "F"*

Nested `if`…`else` statements

A More Complex Example

- Translate the pseudocode to real Java code:

```
if ( studentGrade >= 90 )
   System.out.println( "A" );
else
   if ( studentGrade >= 80 )
      System.out.println( "B" );
   else
      if ( studentGrade >= 70 )
         System.out.println( "C" );
      else
         if ( studentGrade >= 60 )
            System.out.println( "D" );
         else
            System.out.println( "F" );
```

A More Elegant Version

- Most Java programmers prefer to write the preceding nested `if…else` statement as:

```
if ( studentGrade >= 90 )
   System.out.println( "A" );
else if ( studentGrade >= 80 )
   System.out.println( "B" );
else if ( studentGrade >= 70 )
   System.out.println( "C" );
else if ( studentGrade >= 60 )
   System.out.println( "D" );
else
   System.out.println( "F" );
```

If-else Matching Rule

- The Java compiler always associates an `else` with the immediately preceding `if` unless told to do otherwise by the placement of braces (`{` and `}`)

- The following code does not execute like what it appears:

```
if ( x > 5 )
   if ( y > 5 )
      System.out.println( "x and y are > 5" );
else
   System.out.println( "x is <= 5" );
```

If-else Matching Rule

- Recall that the extra spaces are irrelevant in Java. The compiler actually interprets the statement as

```
if ( x > 5 )
   if ( y > 5 )
      System.out.println( "x and y are > 5" );
   else
      System.out.println( "x is <= 5" );
```

If-else Matching Rule

What if you really want this effect?

```
if ( x > 5 )
     if ( y > 5 )
          System.out.println( "x and y are > 5" );
else
     System.out.println( "x is <= 5" );
```

Curly braces indicate that the 2nd `if` is the body of the 1st `if`

```
if ( x > 5 ) {
     if ( y > 5 )
          System.out.println( "x and y are > 5" );
} else
     System.out.println( "x is <= 5" );
```

Tip: always use {} to make the bodies of if and else clear.

Syntax and Logic Errors Revisited

- Syntax errors (e.g., when one brace in a block is left out of the program) are caught by the compiler

- A logic error (e.g., when both braces in a block are left out of the program) has its effect at execution time
  - A fatal logic error causes a program to fail and terminate prematurely
  - A nonfatal logic error allows a program to continue executing but causes it to produce incorrect results

Empty Statement

- Just as a block can be placed anywhere a single statement can be placed, it's also possible to have an empty statement

- The empty statement is represented by placing a semicolon (`;`) where a statement would normally be

```
if (x == 1) {
    ;
} else if (x == 2) {
    ;
} else {
    ;
}
```

This program is valid, although meaningless.

`while` Repetition Statement

- Repeat an action while a condition remains true
- Pseudocode

  While there are more items on my shopping list
  Purchase next item and cross it off my list

- The repetition statement's body may be a single statement or a block. Eventually, the condition should become false, and the repetition terminates, and the first statement after the repetition statement executes.

Example

- Example of Java's *while* repetition statement: find the first power of 3 larger than 100

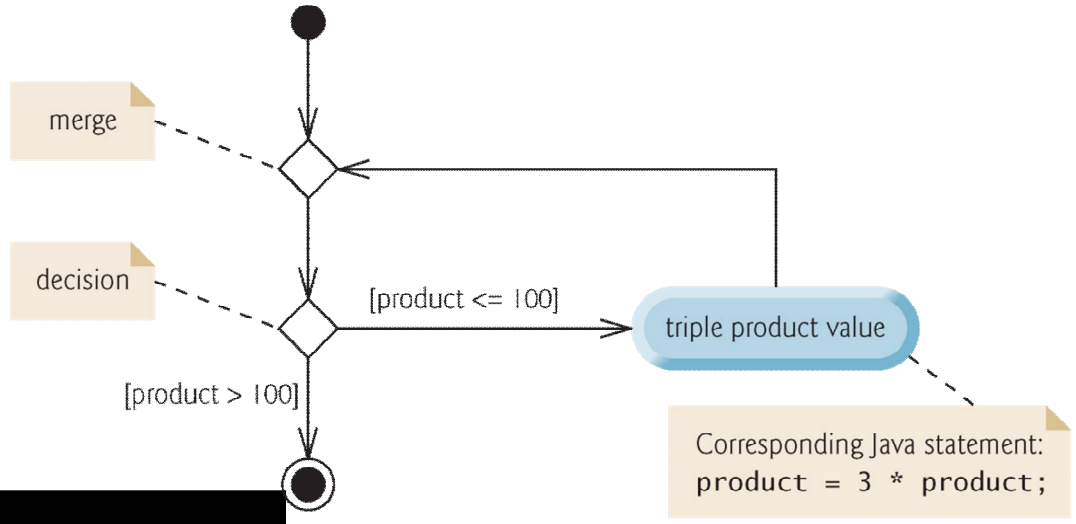
```
int product = 3;

while ( product <= 100 )
   product = 3 * product;
```

product value
3
9
27
81
243 ← Loop terminates

`while` Statement Activity Diagram

- The UML represents both the merge symbol and the decision symbol as diamonds

- The merge symbol joins two flows of activity into one



Formulating Algorithms: Counter-Controlled Repetition

- *Problem: A class of ten students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Determine the class average on the quiz*

- *Analysis:* the algorithm for solving this problem on a computer must input each grade, keep track of the total of all grades input, perform the averaging calculation and print the result

- Solution: Use counter-controlled repetition to input the grades one at a time. A variable called a counter (or control variable) controls the number of times a set of statements will execute.

Formulating Algorithms: Counter-Controlled Repetition

```
1   Set total to zero
2   Set grade counter to one
3
4   While grade counter is less than or equal to ten
5       Prompt the user to enter the next grade
6       Input the next grade
7       Add the grade into the total
8       Add one to the grade counter
9
10  Set the class average to the total divided by ten
11  Print the class average
```

total accumulates the sum of several values

grade counter counts the number of inputs

```
1  // Fig. 3.6: ClassAverage.java
2  // Counter-controlled repetition: Class-average problem.
3  import java.util.Scanner; // program uses class Scanner
4
5  public class ClassAverage
6  {
7     public static void main( String[] args )
8     {
9        // create Scanner to obtain input from command window
10       Scanner input = new Scanner( System.in );
11
12       int total; // sum of grades entered by user
13       int gradeCounter; // number of the grade to be entered next
14       int grade; // grade value entered by user
15       int average; // average of grades
16
17       // initialization phase
18       total = 0; // initialize total
19       gradeCounter = 1; // initialize loop counter
20
```

```
21          // processing phase
22          while ( gradeCounter <= 10 ) // loop 10 times
23          {
24             System.out.print( "Enter grade: " ); // prompt
25             grade = input.nextInt(); // input next grade
26             total = total + grade; // add grade to total
27             gradeCounter = gradeCounter + 1; // increment counter by 1
28          } // end while
29
30          // termination phase
31          average = total / 10; // integer division yields integer result
32
33          // display total and average of grades
34          System.out.printf( "\nTotal of all 10 grades is %d\n", total );
35          System.out.printf( "Class average is %d\n", average );
36       } // end main
37    } // end class ClassAverage
```

```
Enter grade: 67
Enter grade: 78
Enter grade: 89
Enter grade: 67
Enter grade: 87
Enter grade: 98
Enter grade: 93
Enter grade: 85
Enter grade: 82
Enter grade: 100

Total of all 10 grades is 846
Class average is 84
```

The Scope of Variables

- Variables declared in a method body are local variables and can be used only from the line of their declaration to the closing right brace of the method declaration.
- A local variable cannot be accessed outside the method in which it's declared.
- A local variable's declaration must appear before the variable is used in that method

Formulating Algorithms: Sentinel-Controlled Repetition

- ***A new problem:*** *Develop a class-averaging program that processes grades for an arbitrary number of students each time it is run.*

- **Analysis:** The number of grades was know earlier, but here how can the program determine when to stop the input of grades?

Formulating Algorithms: Sentinel-Controlled Repetition

We can use **a special value** called a sentinel value (a.k.a, signal value, dummy value or flag value) can be used to indicate "end of data entry".

Marking the end of inputs

92, 77, 68, 84, 35, 72, 95, 79, 88, 84, **-1**

Formulating Algorithms: Sentinel-Controlled Repetition

- Sentinel-controlled repetition is often called indefinite repetition because the number of repetitions is not known before the loop begins executing
- A sentinel value must be chosen that cannot be confused with an acceptable input value

One of the left items? Of course not…

```
1   Initialize total to zero
2   Initialize counter to zero
3
4   Prompt the user to enter the first grade
5   Input the first grade (possibly the sentinel)
6
7   While the user has not yet entered the sentinel
8       Add this grade into the running total
9       Add one to the grade counter
10      Prompt the user to enter the next grade
11      Input the next grade (possibly the sentinel)
12
13  If the counter is not equal to zero
14      Set the average to the total divided by the counter
15      Print the average
16  else
17      Print "No grades were entered"
```

total stores the sum of grades
counter stores the number grades

Try to take an input

If no sentinel value seen, repeat the process

Compute and print average (avoid division by 0)

```
1  // Fig. 3.8: ClassAverage.java
2  // Sentinel-controlled repetition: Class-average problem.
3  import java.util.Scanner; // program uses class Scanner
4
5  public class ClassAverage
6  {
7     public static void main( String[] args )
8     {
9        // create Scanner to obtain input from command window
10       Scanner input = new Scanner( System.in );
11
12       int total; // sum of grades
13       int gradeCounter; // number of grades entered
14       int grade; // grade value
15       double average; // number with decimal point for average
16
17       // initialization phase
18       total = 0; // initialize total
19       gradeCounter = 0; // initialize loop counter
20
21       // processing phase
22       // prompt for input and read grade from user
23       System.out.print( "Enter grade or -1 to quit: " );
24       grade = input.nextInt();
```

```
25
26          // loop until sentinel value read from user
27          while ( grade != -1 )
28          {
29             total = total + grade; // add grade to total
30             gradeCounter = gradeCounter + 1; // increment counter
31
32             // prompt for input and read next grade from user
33             System.out.print( "Enter grade or -1 to quit: " );
34             grade = input.nextInt();
35          } // end while
36
37          // termination phase
38          // if user entered at least one grade...
39          if ( gradeCounter != 0 )
40          {
41             // calculate average of all grades entered
42             average = (double) total / gradeCounter;
43
44             // display total and average (with two digits of precision)
45             System.out.printf( "\nTotal of the %d grades entered is %d\n",
46                gradeCounter, total );
47             System.out.printf( "Class average is %.2f\n", average );
48          } // end if
```

```
49          else // no grades were entered, so output appropriate message
50             System.out.println( "No grades were entered" );
51       } // end main
52    } // end class ClassAverage
```

```
Enter grade or -1 to quit: 97
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 72
Enter grade or -1 to quit: -1

Total of the 3 grades entered is 257
Class average is 85.67
```

Type Cast and Promotion

```
int total;
int gradeCounter;
double average;
```

```
average = (double) total / gradeCounter;
```

The unary cast operator creates a temporary floating-point copy of its operand

- Cast operator performs explicit conversion (or type cast).
- The value stored in the operand is unchanged.
- Java evaluates only arithmetic expressions in which the operands' types are identical.
- Promotion (or implicit conversion) performed on operands.
- In the above expression, the `int` value of gradeCounter is promoted to a `double` value for computation.

More on Cast Operators

- Cast operators are available for any type.
- Cast operator formed by placing parentheses around the name of a type.
- Cast operators associate from right to left.
- This precedence is one level higher than that of the multiplicative operators *, / and %.

Case Study: Nested Control Statements

- A college offers a course that prepares students for the state licensing exam for real estate brokers. Last year, ten of the students who completed this course took the exam. The college wants to know how well its students did on the exam.

- You've been asked to write a program to summarize the results. You've been given a list of these 10 students. Next to each name is written a 1 if the student passed the exam or a 2 if the student failed.

Case Study: Nested Control Statements

- Your program should analyze the exam results as follows:
  - Input each test result (i.e., a 1 or a 2). Display the message "Enter result" on the screen each time the program requests another test result.
  - Count the number of test results of each type (pass or fail).
  - Display a summary of the test results, indicating the number of students who passed and the number who failed.
  - If more than eight students passed the exam, print the message "Bonus to instructor!"

```
1  Initialize passes to zero
2  Initialize failures to zero
3  Initialize student counter to one
4
5  While student counter is less than or equal to 10
6      Prompt the user to enter the next exam result
7      Input the next exam result
8
9      If the student passed
10         Add one to passes
11     Else
12         Add one to failures
13
14     Add one to student counter
15
16 Print the number of passes
17 Print the number of failures
18
19 If more than eight students passed
20     Print "Bonus to instructor!"
```

Two variables defined: passes and failures

Counter-controlled repetition

`if…else` nested in `while`

```
1  // Fig. 3.10: Analysis.java
2  // Analysis of examination results.
3  import java.util.Scanner; // class uses class Scanner
4
5  public class Analysis
6  {
7     public static void main( String[] args )
8     {
9        // create Scanner to obtain input from command window
10       Scanner input = new Scanner( System.in );
11
12       // initializing variables in declarations
13       int passes = 0; // number of passes
14       int failures = 0; // number of failures
15       int studentCounter = 1; // student counter
16       int result; // one exam result (obtains value from user)
17
18       // process 10 students using counter-controlled loop
19       while ( studentCounter <= 10 )
20       {
21          // prompt user for input and obtain value from user
22          System.out.print( "Enter result (1 = pass, 2 = fail): " );
23          result = input.nextInt();
```

```
24
25             // if...else nested in while
26             if ( result == 1 )           // if result 1,
27                passes = passes + 1;      // increment passes;
28             else                         // else result is not 1, so
29                failures = failures + 1; // increment failures
30
31             // increment studentCounter so loop eventually terminates
32             studentCounter = studentCounter + 1;
33          } // end while
34
35          // termination phase; prepare and display results
36          System.out.printf( "Passed: %d\nFailed: %d\n", passes, failures );
37
38          // determine whether more than 8 students passed
39          if ( passes > 8 )
40             System.out.println( "Bonus to instructor!" );
41       } // end main
42    } // end class Analysis
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 9
Failed: 1
Bonus to instructor!
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 6
Failed: 4
```

Compound Assignment Operators

- Compound assignment operators simplify assignment expressions.

- *variable = variable operator expression;* where operator is one of +, -, *, / or % can be written in the form

  variable operator= expression;

- `c = c + 3;` can be written as `c += 3;`

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	10 to c
`-=`	`d -= 4`	`d = d - 4`	1 to d
`*=`	`e *= 5`	`e = e * 5`	20 to e
`/=`	`f /= 3`	`f = f / 3`	2 to f
`%=`	`g %= 9`	`g = g % 9`	3 to g

Fig. 3.11 | Arithmetic compound assignment operators.

Increment and Decrement Operators

- Unary increment operator, `++`, adds one to its operand
- Unary decrement operator, `--`, subtracts one from its operand
- An increment or decrement operator that is prefixed to (placed before) a variable is referred to as the prefix increment or prefix decrement operator, respectively.
- An increment or decrement operator that is postfixed to (placed after) a variable is referred to as the postfix increment or postfix decrement operator, respectively.

```
int a = 6;  int b = ++a;  int c = a--;
```

Preincrementing/Predecrementing

- Using the prefix increment (or decrement) operator to add (or subtract) 1 from a variable is known as preincrementing (or predecrementing) the variable.
- Preincrementing (or predecrementing) a variable causes the variable to be incremented (decremented) by 1; then the new value is used in the expression in which it appears.

```
int a = 6;
int b = ++a; // b gets the value 7
```

Postincrementing/Postdecrementing

- Using the postfix increment (or decrement) operator to add (or subtract) 1 from a variable is known as postincrementing (or postdecrementing) the variable.
- This causes the current value of the variable to be used in the expression in which it appears; then the variable's value is incremented (decremented) by 1.

```
int a = 6;
int b = a++; // b gets the value 6
```

Note the Difference

```
int a = 6;
int b = a++; // b gets the value 6
```

```
int a = 6;
int b = ++a; // b gets the value 7
```

In both cases, a becomes 7 after execution, but b gets different values. Be careful when programming.

The Operators Introduced So Far

Precedence

Operators						Associativity	Type
++	--					right to left	unary postfix
++	--	+	-	(*type*)		right to left	unary prefix
*	/	%				left to right	multiplicative
+	-					left to right	additive
<	<=	>	>=			left to right	relational
==	!=					left to right	equality
?:						right to left	conditional
=	+=	-=	*=	/=	%=	right to left	assignment

Fig. 3.14 | Precedence and associativity of the operators discussed so far.

Please practice each of the operators by yourself ☺