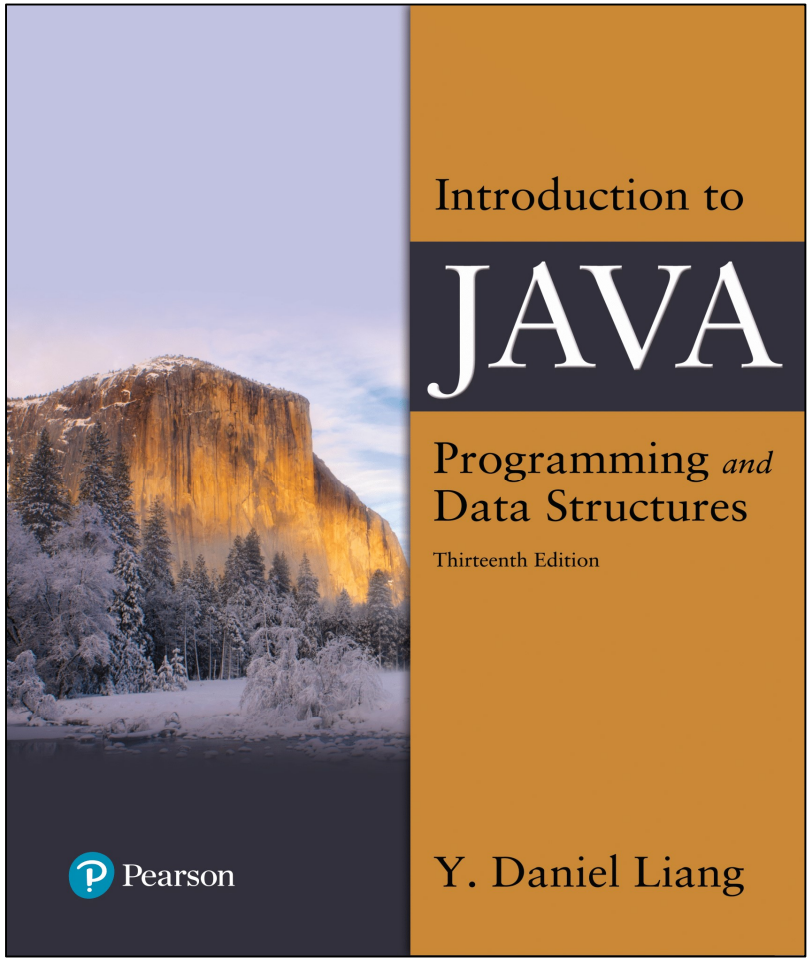
Introduction to Java Programming and Data Structures

Thirteenth Edition



Chapter 3

Selections

Motivations

If you assigned a negative value for **`radius`** in program, `CircleArea.java`, the program would print an invalid result. If the radius is negative, you don't want the program to compute the area. How can you deal with this situation?

Review

- Four types of program control
  - Sequence
  - Selection
  - Iteration
  - Subprograms
- Sequence example
  - Given hours worked and pay rate, calculate gross pay.

Preview

- Selection Program Control
  - Make decisions and execute alternative statements based on the result of your decision.
  - Example
    - Given hours worked and pay rate, calculate gross pay. Overtime (over 40 hours is paid 1.5)

The `boolean` Type and Operators

Often in a program you need to compare two values, such as whether i is greater than j. Java provides six comparison operators (also known as relational operators) that can be used to compare two values. The result of the comparison is a Boolean value: true or false.

```
boolean b = (1 > 2);
```

Relational Operators

Java Operator	Mathematics Symbol	Name	Example (radius is 5)	Result
<	<	less than	radius < 0	false
<=	≤	less than or equal to	radius <= 0	false
>	>	greater than	radius > 0	true
>=	≥	greater than or equal to	radius >= 0	true
==	=	equal to	radius == 0	false
!=	≠	not equal to	radius != 0	true

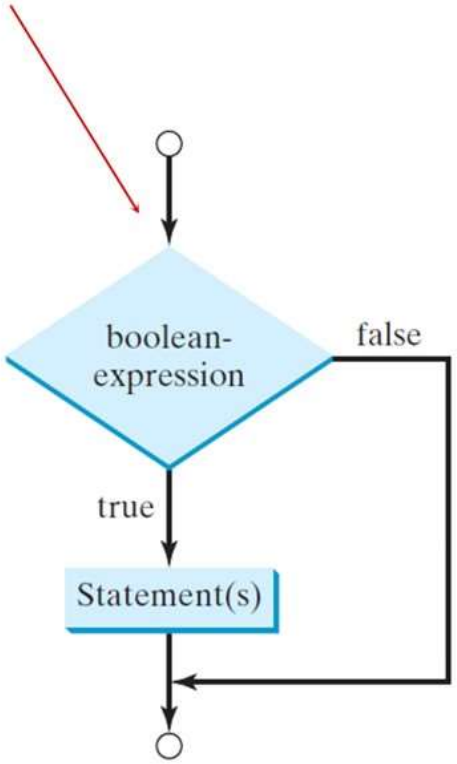
Problem: A Simple Math Learning Tool

This example creates a program to let a first grader practice additions. The program randomly generates two single-digit integers number1 and number2 and displays a question such as “What is 7 + 9?” to the student. After the student types the answer, the program displays a message to indicate whether the answer is true or false.

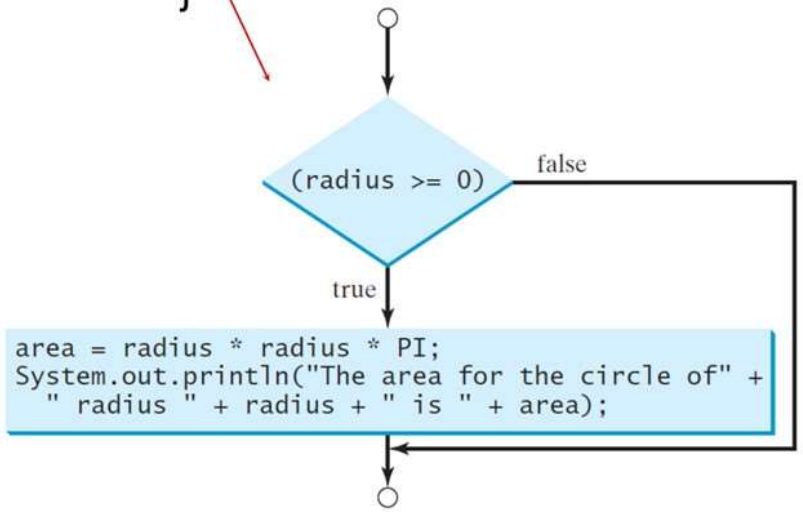
AdditionQuiz

One-Way `if` Statements

```
if (boolean-expression) {
  statement(s);
}
```




```
if (radius >= 0) {
  area = radius * radius * PI;
  System.out.println("The area"
    + " for the circle of radius "
    + radius + " is " + area);
}
```



Pearson

Note

```
if i > 0 {
  System.out.println("i is positive");
}
```

(a) Wrong

```
if (i > 0) {
  System.out.println("i is positive");
}
```

(b) Correct

```
if (i > 0) {
  System.out.println("i is positive");
}
```

(a)

Equivalent

```
if (i > 0)
  System.out.println("i is positive");
```

(b)

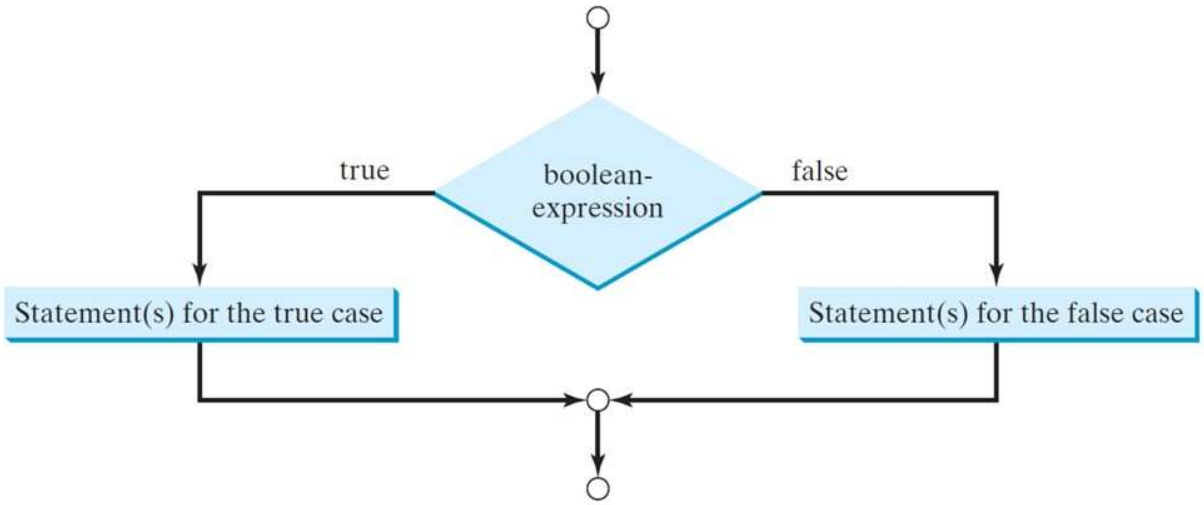
Simple `if` Demo

Write a program that prompts the user to enter an integer. If the number is a multiple of 5, print HiFive. If the number is divisible by 2, print HiEven.

SimpleIfDemo

The Two-Way `if` Statement

```
if (boolean-expression) {
   statement(s)-for-the-true-case;
}
else {
   statement(s)-for-the-false-case;
}
```

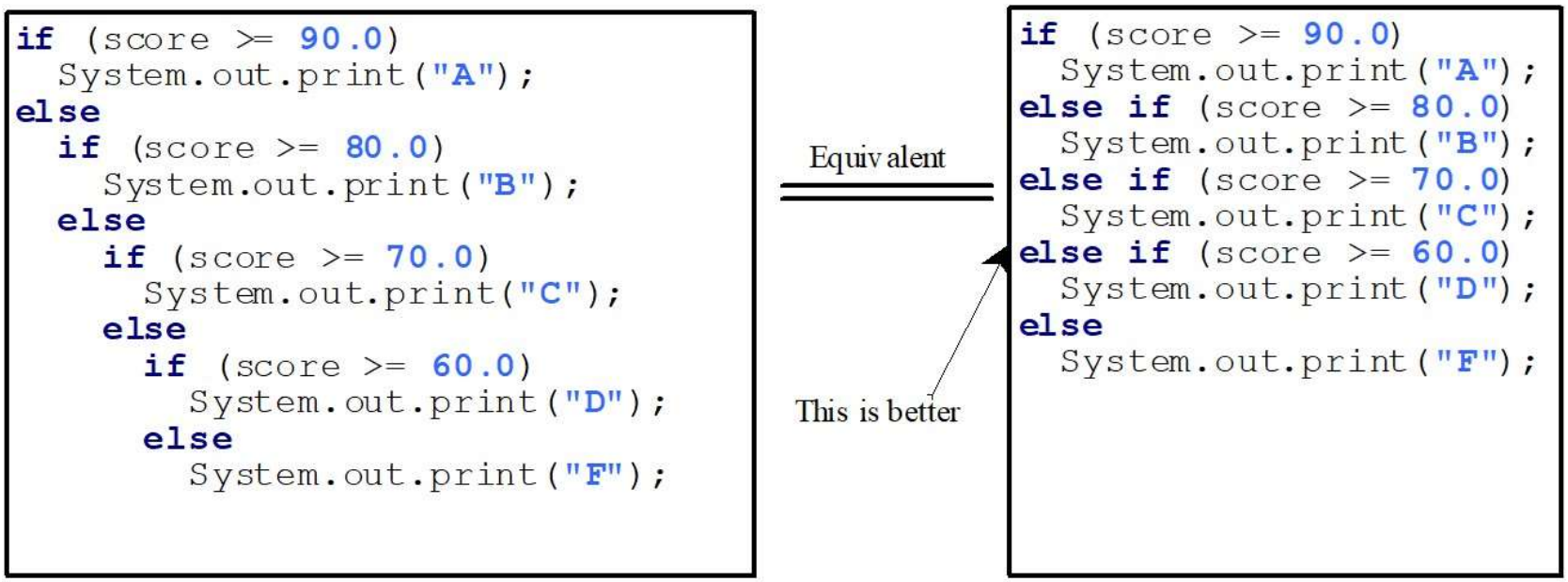


if-else Example

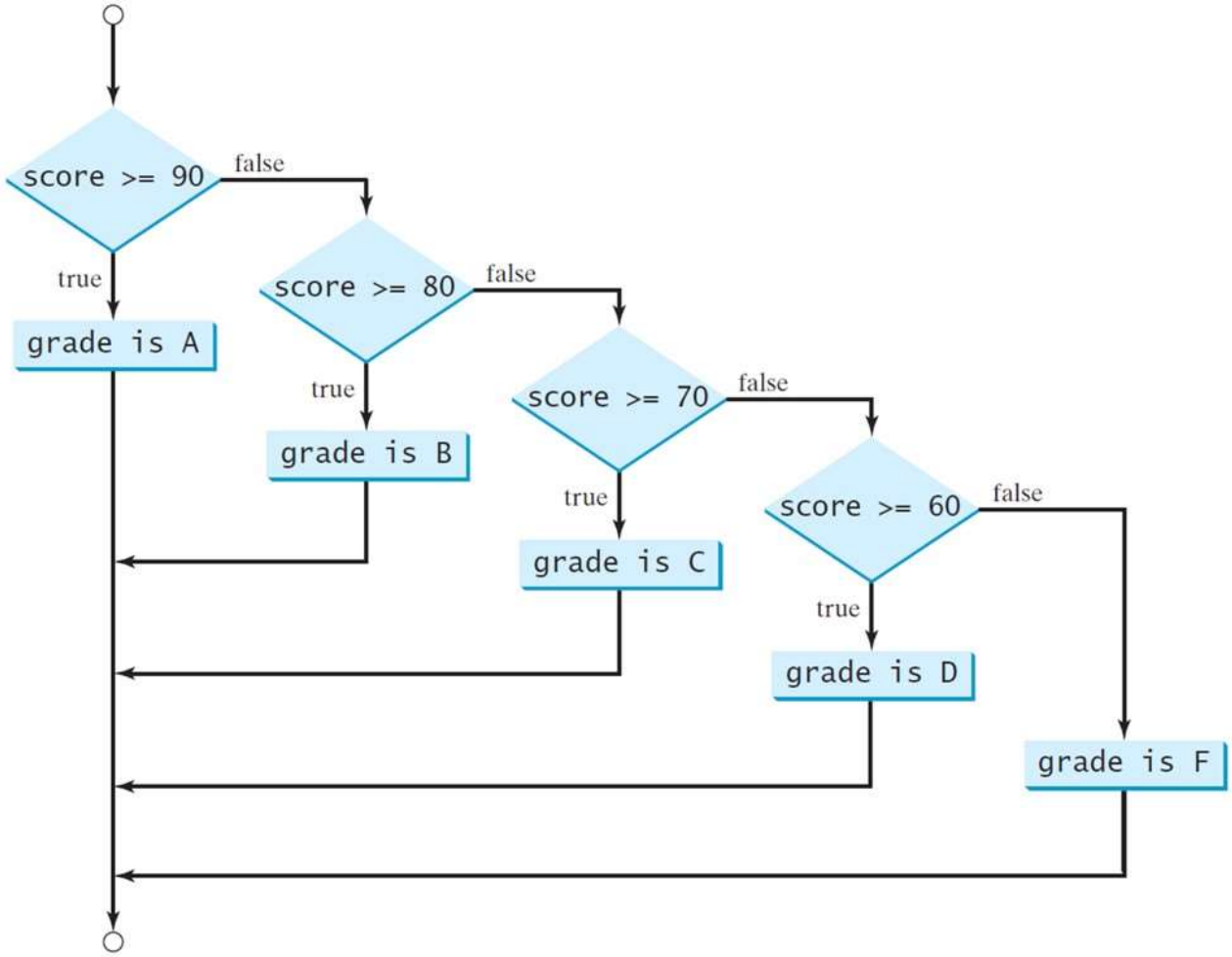
```
if (radius >= 0) {
  area = radius * radius * 3.14159;

    System out.println("The area for the “
    + “circle of radius " + radius +
    " is " + area);
}
else {
  System.out.println("Negative input");
}
```

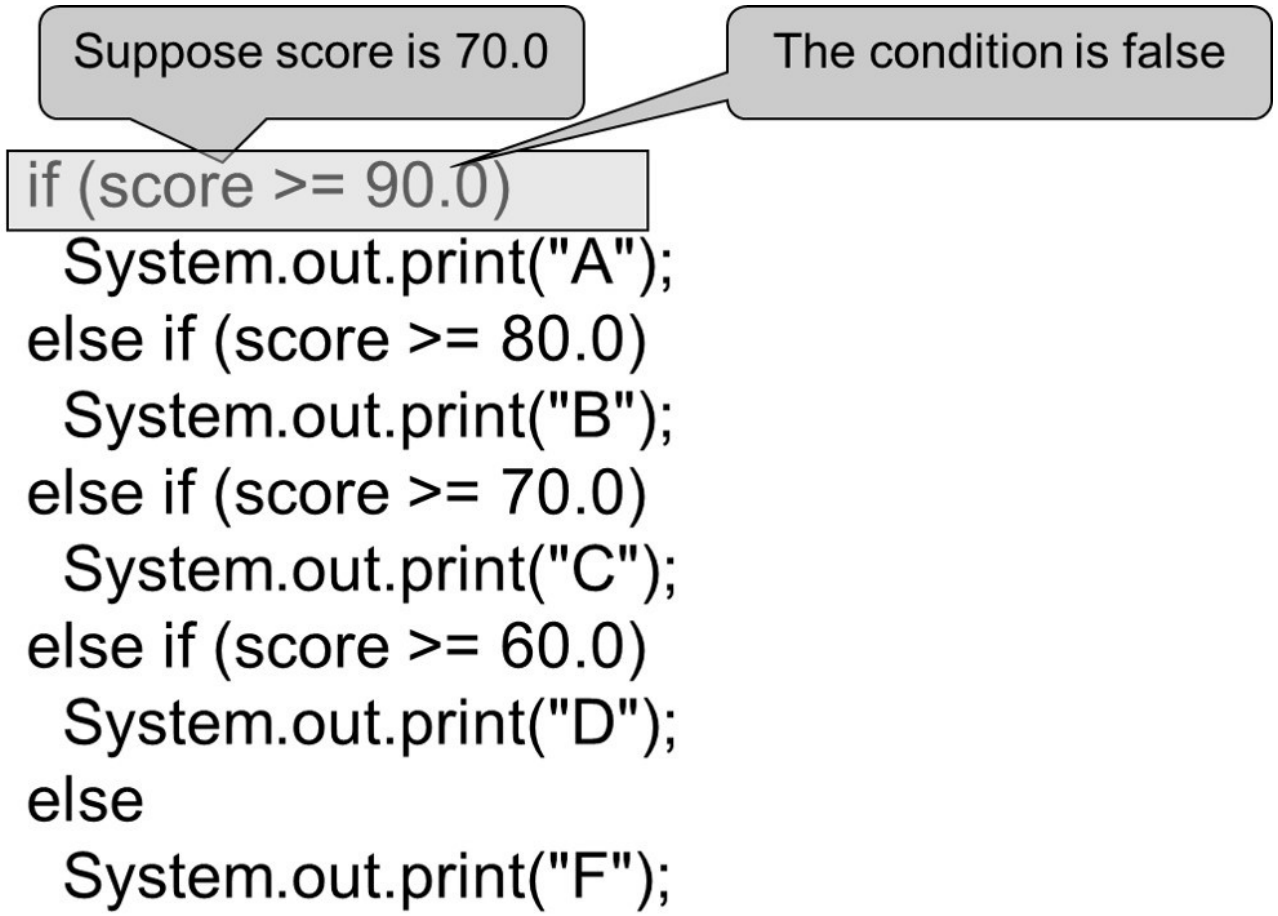
Multiple Alternative `if` Statements



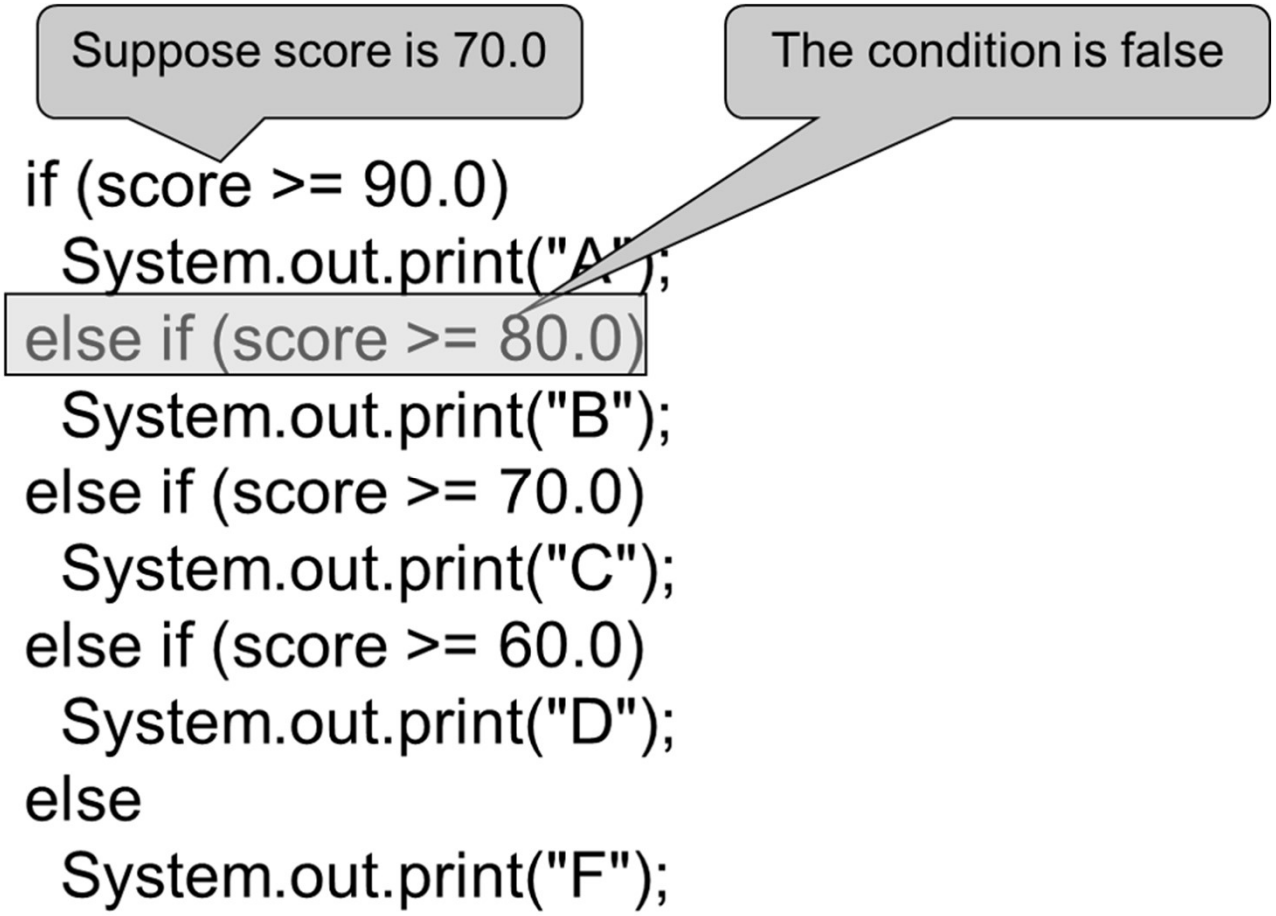
Multi-Way `if-else` Statements



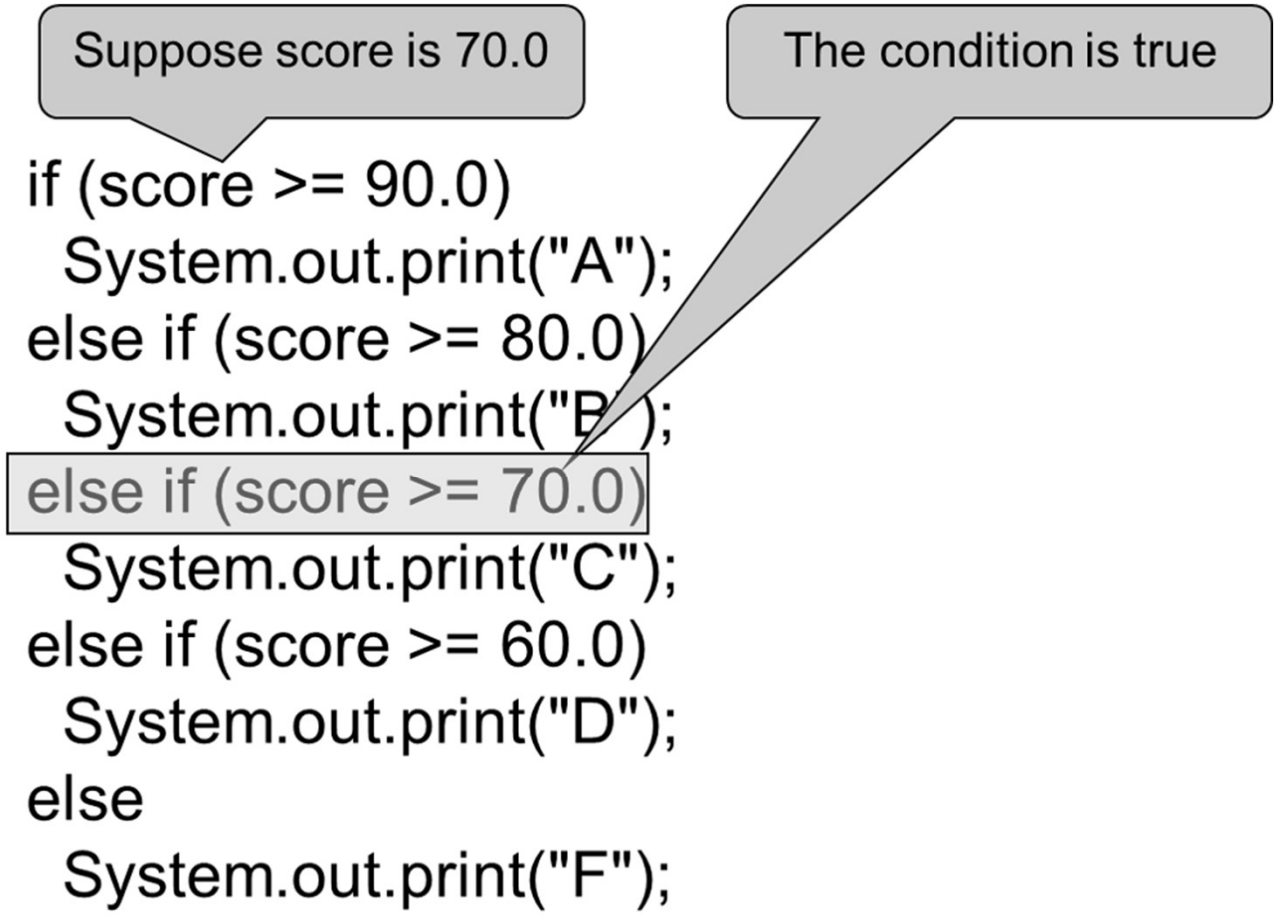
Trace `if-else` Statement (1 of 5)



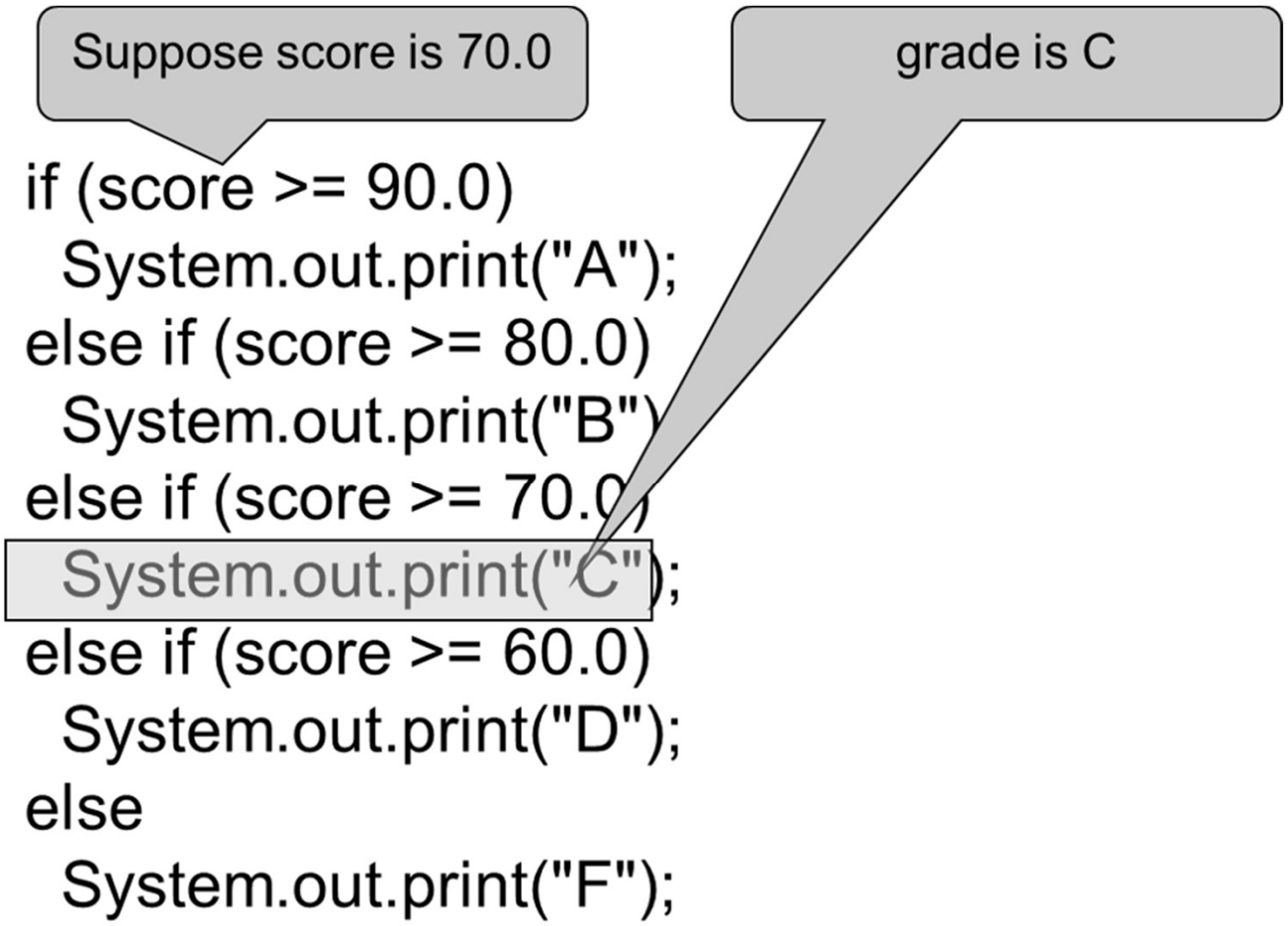
Trace `if-else` Statement (2 of 5)



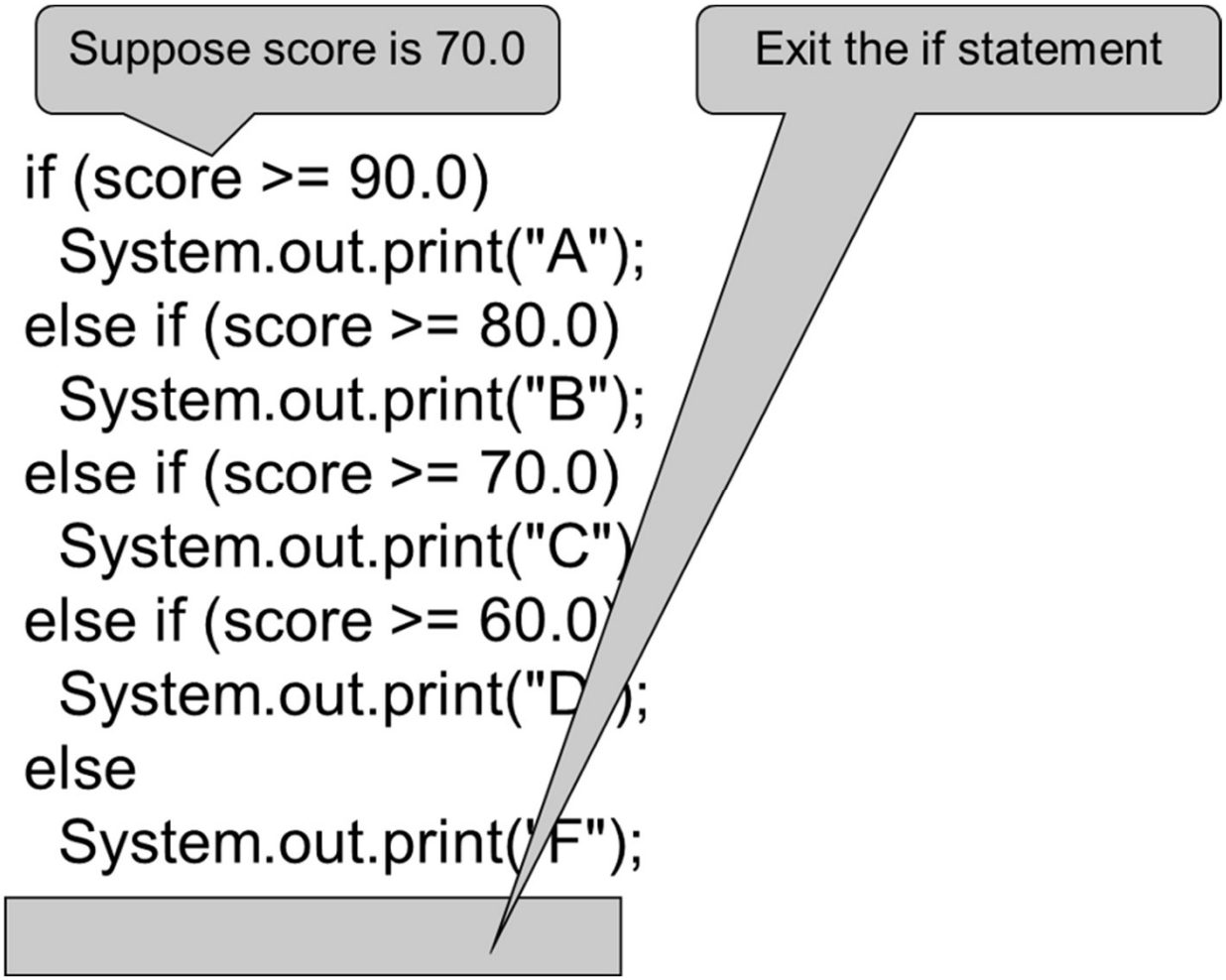
Trace `if-else` Statement (3 of 5)



Trace `if-else` Statement (4 of 5)

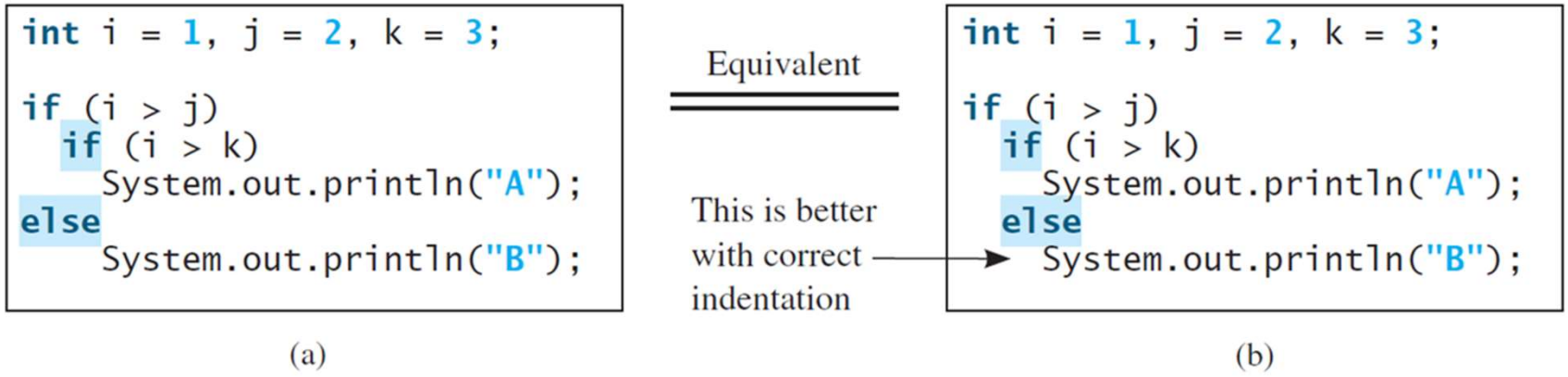


Trace `if-else` Statement (5 of 5)



Note (1 of 2)

The **`else`** clause matches the most recent **`if`** clause in the same block.



Note (2 of 2)

Nothing is printed from the preceding statement. To force the **`else`** clause to match the first if clause, you must add a pair of braces:

```
int i = 1;
int j = 2;
int k = 3;
if (i > j) {
   if (i > k)
    System.out.println("A");
}
else
   System.out.println("B");
```

This statement prints B.

Common Errors

Adding a semicolon at the end of an **`if`** clause is a common mistake.

```
if (radius >= 0);
{
  area = radius*radius*PI;
  System.out.println(
    "The area for the circle of radius " +
    radius + " is " + area);
}
```

Wrong

This mistake is hard to find, because it is not a compilation error or a runtime error, it is a logic error.

This error often occurs when you use the next-line block style.

TIP

```
if (number % 2 == 0)
  even = true;
else
  even = false;
```

(a)

Equivalent

```
boolean even
  = number % 2 == 0;
```

(b)

Caution

```
if (even == true)
  System.out.println(
    "It is even.");
```

(a)

Equivalent

```
if (even)
  System.out.println(
    "It is even.");
```

(b)

Problem: An Improved Math Learning Tool

This example creates a program to teach a first grade child how to learn subtractions. The program randomly generates two single-digit integers **`number1`** and **`number2`** with **`number1 >= number2`** and displays a question such as "What is 9 – 2?" to the student. After the student types the answer, the program displays whether the answer is correct.

SubtractionQuiz

Logical Operators

Operator	Name	Description
!	not	logical negation
&&	and	logical conjunction
||	or	logical disjunction
^	exclusive or	logical exclusion

Truth Table for Operator !

p	!p	Example (assume age = 24, weight = 140)
true	false	!(age > 18) is false, because (age > 18) is true.
false	true	!(weight == 150) is true, because (weight == 150) is false.

Truth Table for Operator &&

p_1	p_2	p_1 && p_2	Example (assume age = 24, weight = 140)
false	false	false	(age <= 18) && (weight < 140) is false, because both conditions are both false.
false	true	false	
true	false	false	(age > 18) && (weight > 140) is false, because (weight > 140) is false.
true	true	true	(age > 18) && (weight >= 140) is true, because both (age > 18) and (weight >= 140) are true.

Truth Table for Operator ||

p_1	p_2	$p_1 \| p_2$	Example (assume age = 24, weight = 140)
false	false	false	
false	true	true	(age > 34) || (weight <= 140) is true, because (age > 34) is false, but (weight <= 140) is true.
true	false	true	(age > 14) || (weight >= 150) is false, because (age > 14) is true.
true	true	true	

Truth Table for Operator ^

p_1	p_2	p_1 ^ p_2	Example (assume age = 24, weight = 140)
false	false	false	(age > 34) ^ (weight > 140) is true, because (age > 34) is false and (weight > 140) is false.
false	true	true	(age > 34) ^ (weight >= 140) is true, because (age > 34) is false but (weight >= 140) is true.
true	false	true	(age > 14) ^ (weight > 140) is true, because (age > 14) is true and (weight > 140) is false.
true	true	false	

Examples (1 of 2)

Here is a program that checks whether a number is divisible by **2** and **3**, whether a number is divisible by **2** or **3**, and whether a number is divisible by **2** or **3** but not both:

TestBooleanOperators

Examples (2 of 2)

```
System.out.println("Is " + number + " divisible by 2 and 3? " +
  ((number % 2 == 0) && (number % 3 == 0)));

System.out.println("Is " + number + " divisible by 2 or 3? " +
  ((number % 2 == 0) || (number % 3 == 0)));

System.out.println("Is " + number +
" divisible by 2 or 3, but not both? " +
  ((number % 2 == 0) ^ (number % 3 == 0)));
```

TestBooleanOperators

Problem: Determining Leap Year?

This program first prompts the user to enter a year as an int value and checks if it is a leap year.

A year is a leap year if it is divisible by 4 but not by 100, or it is divisible by 400.

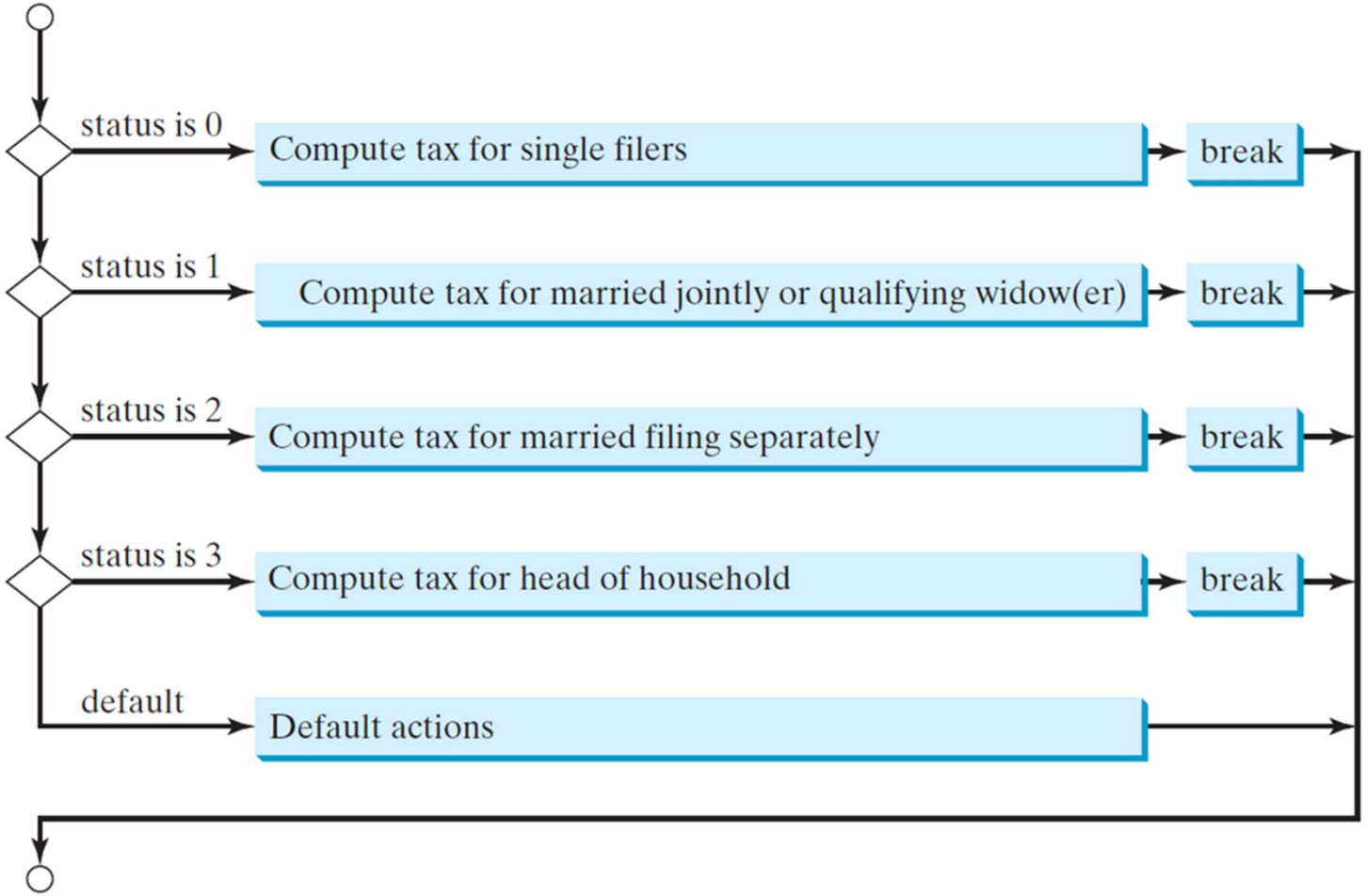
```
(year % 4 == 0 && year % 100 != 0) || (year % 400 == 0)
```

LeapYear

`switch` Statements

```
switch (status) {
   case 0: compute taxes for single filers;
           break;
   case 1: compute taxes for married file jointly;
           break;
   case 2: compute taxes for married file separately;
           break;
   case 3: compute taxes for head of household;
           break;
   default: System.out.println("Errors: invalid status");
           System.exit(1);
}
```

`switch` Statement Flow Chart



`switch` Statement Rules (1 of 2)

The switch-expression must yield a value of char, byte, short, or int type and must always be enclosed in parentheses.

The value1, ..., and valueN must have the same data type as the value of the switch-expression. The resulting statements in the case statement are executed when the value in the case statement matches the value of the switch-expression. Note that value1, ..., and valueN are constant expressions, meaning that they cannot contain variables in the expression, such as 1 + x.

```
switch (switch-expression) {
  case value1:  statement(s)1;
                break;
  case value2: statement(s)2;
                break;
  ...
  case valueN: statement(s)N;
                break;
  default: statement(s)-for-default;
}
```

`switch` Statement Rules (2 of 2)

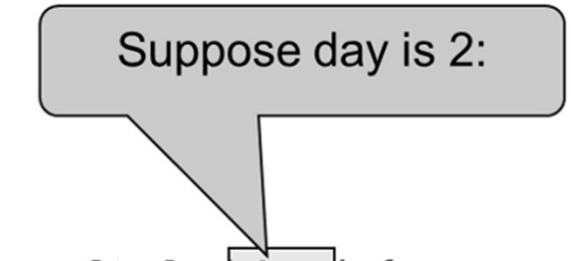
The keyword break is optional, but it should be used at the end of each case in order to terminate the remainder of the switch statement. If the break statement is not present, the next case statement will be executed.

The default case, which is optional, can be used to perform actions when none of the specified cases matches the switch-expression.

```
switch (switch-expression) {
  case value1:  statement(s)1;
        break;
  case value2: statement(s)2;
        break;
  …
  case valueN: statement(s)N;
        break;
  default: statement(s)-for-default;
}
```

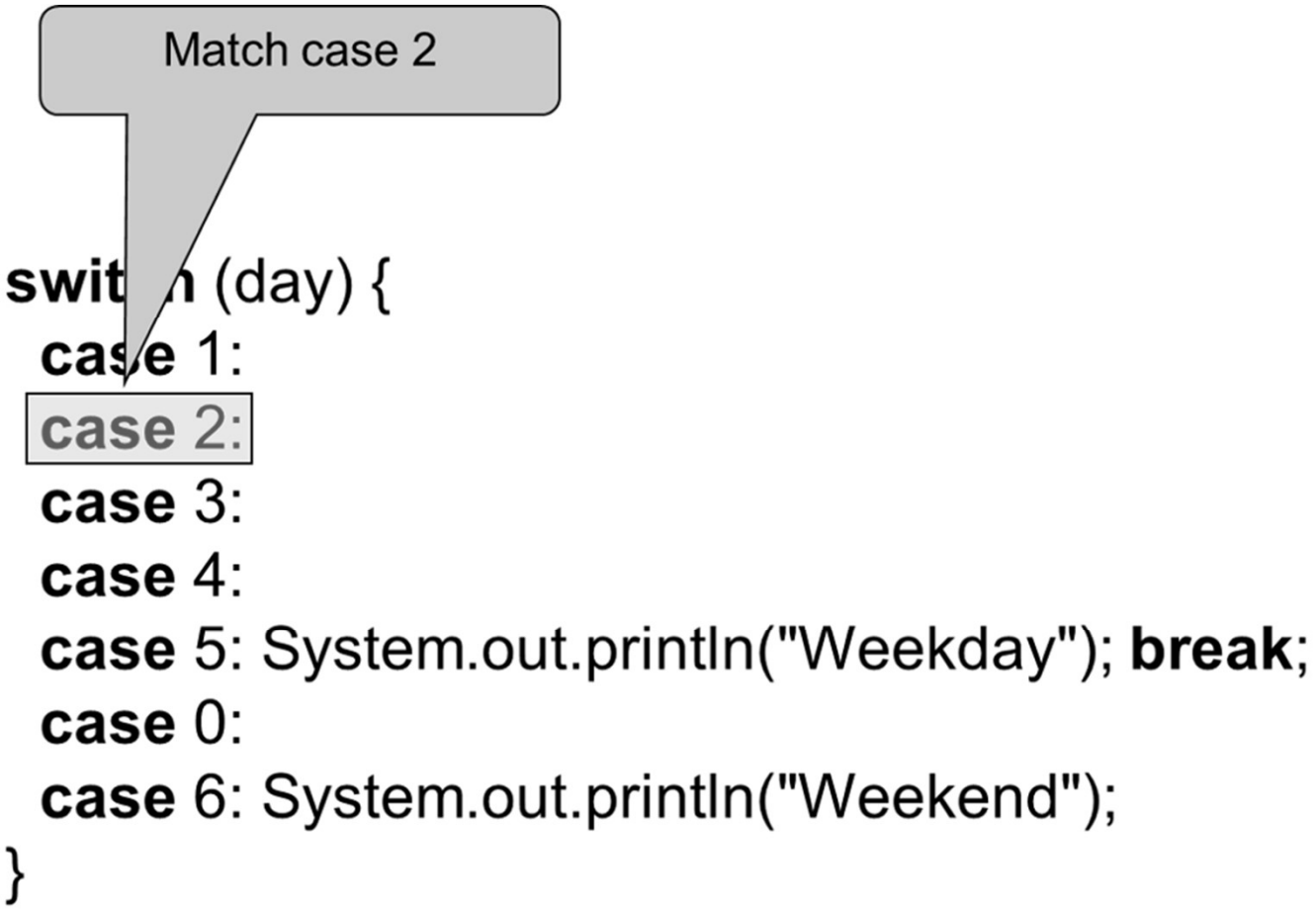
When the value in a **case** statement matches the value of the **switch-expression**, the statements *starting from this case* are executed until either a **break** statement or the end of the **switch** statement is reached.

Trace `switch` Statement (1 of 7)

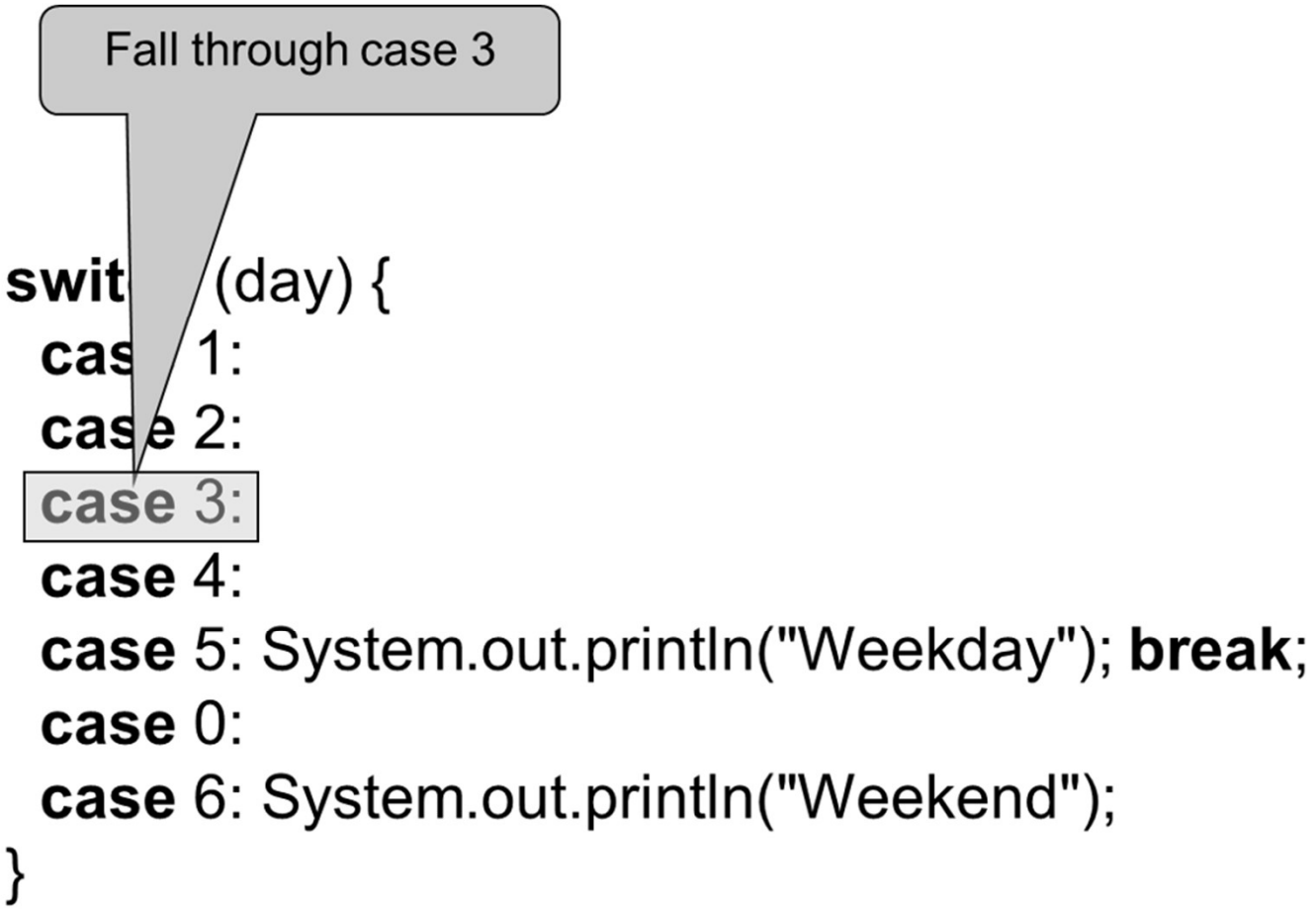



```
switch (day) {
  case 1:
  case 2:
  case 3:
  case 4:
  case 5: System.out.println("Weekday"); break;
  case 0:
  case 6: System.out.println("Weekend");
}
```

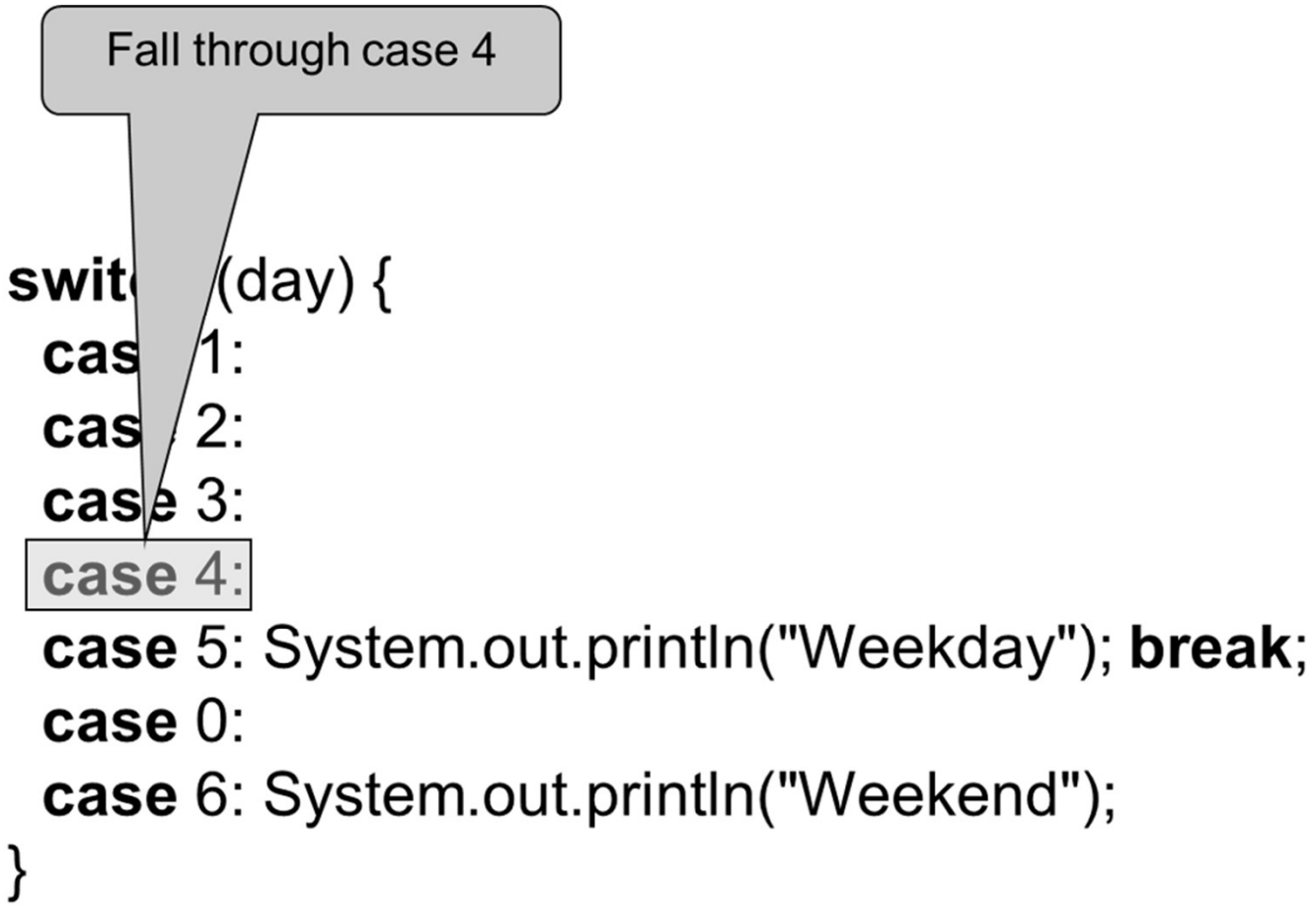
Trace `switch` Statement (2 of 7)



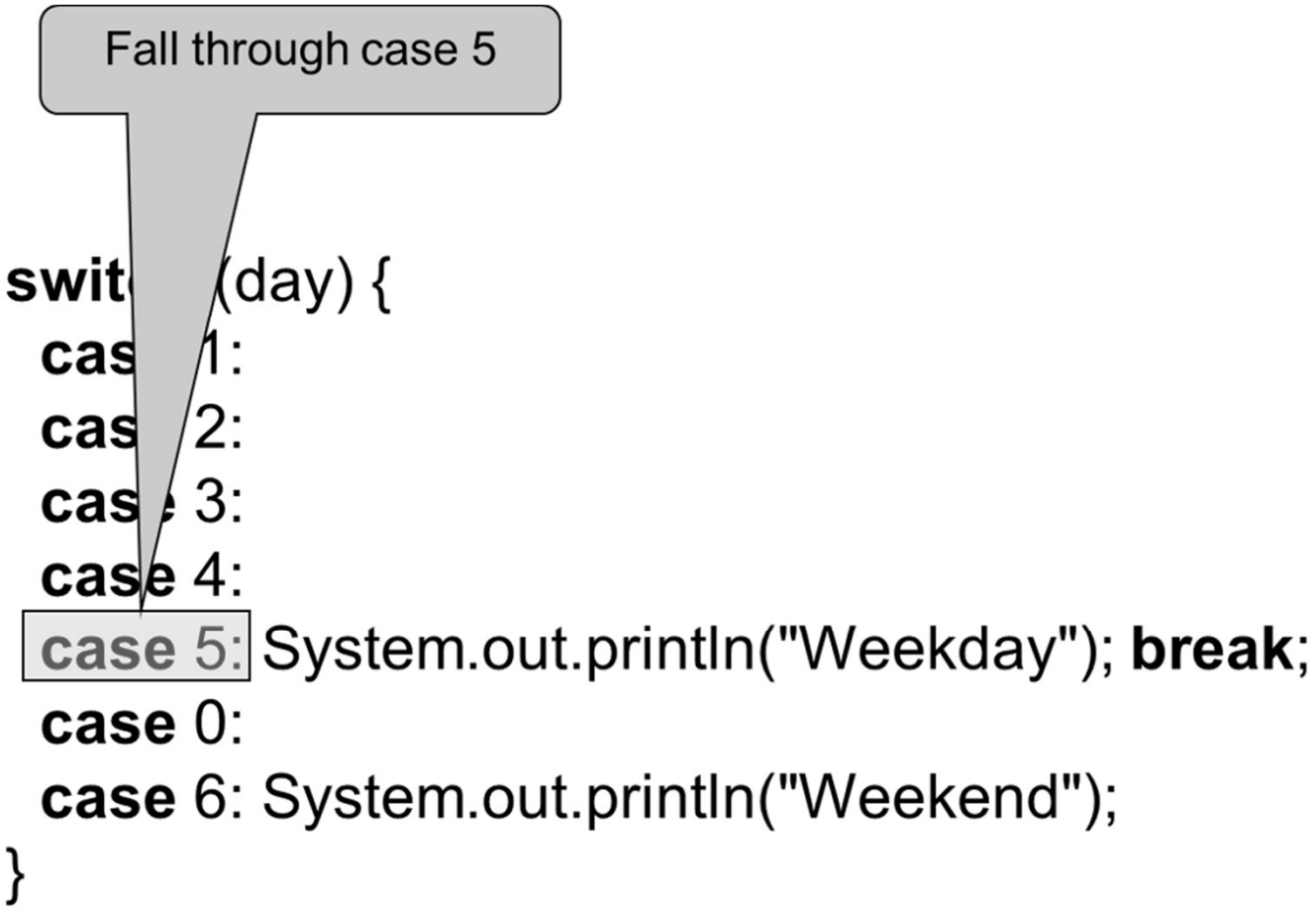
Trace `switch` Statement (3 of 7)



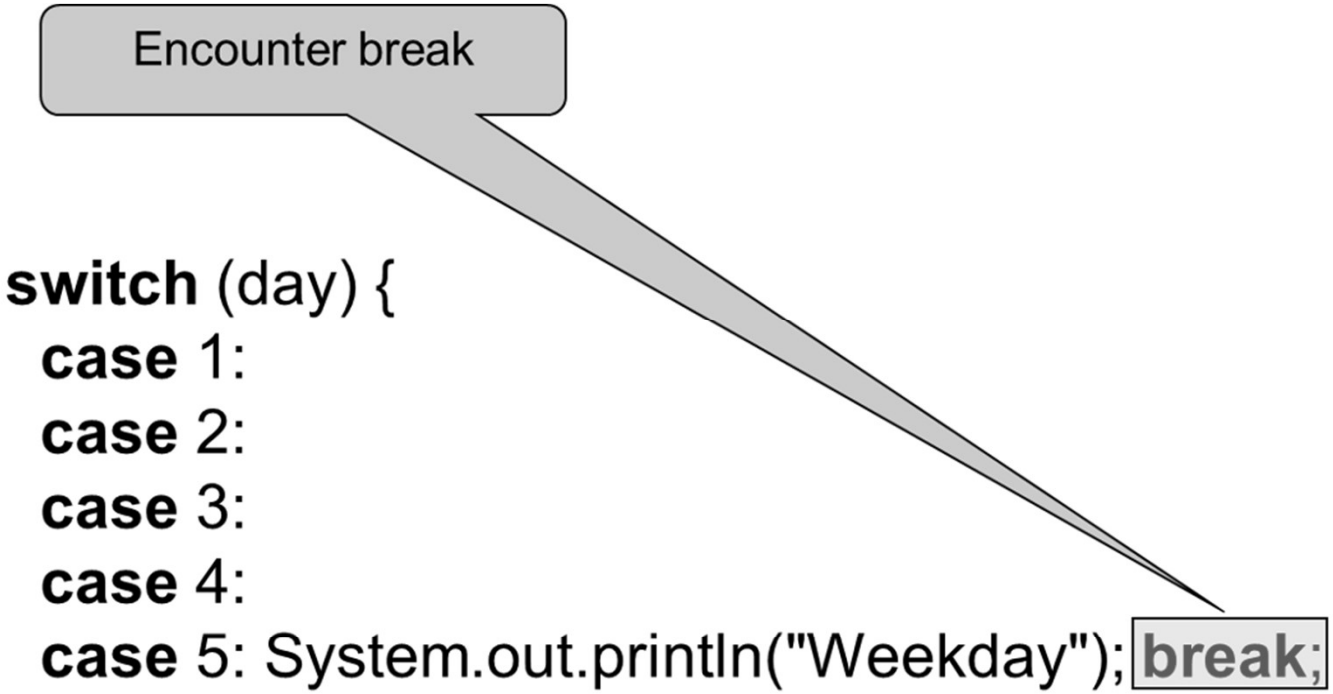
Trace `switch` Statement (4 of 7)



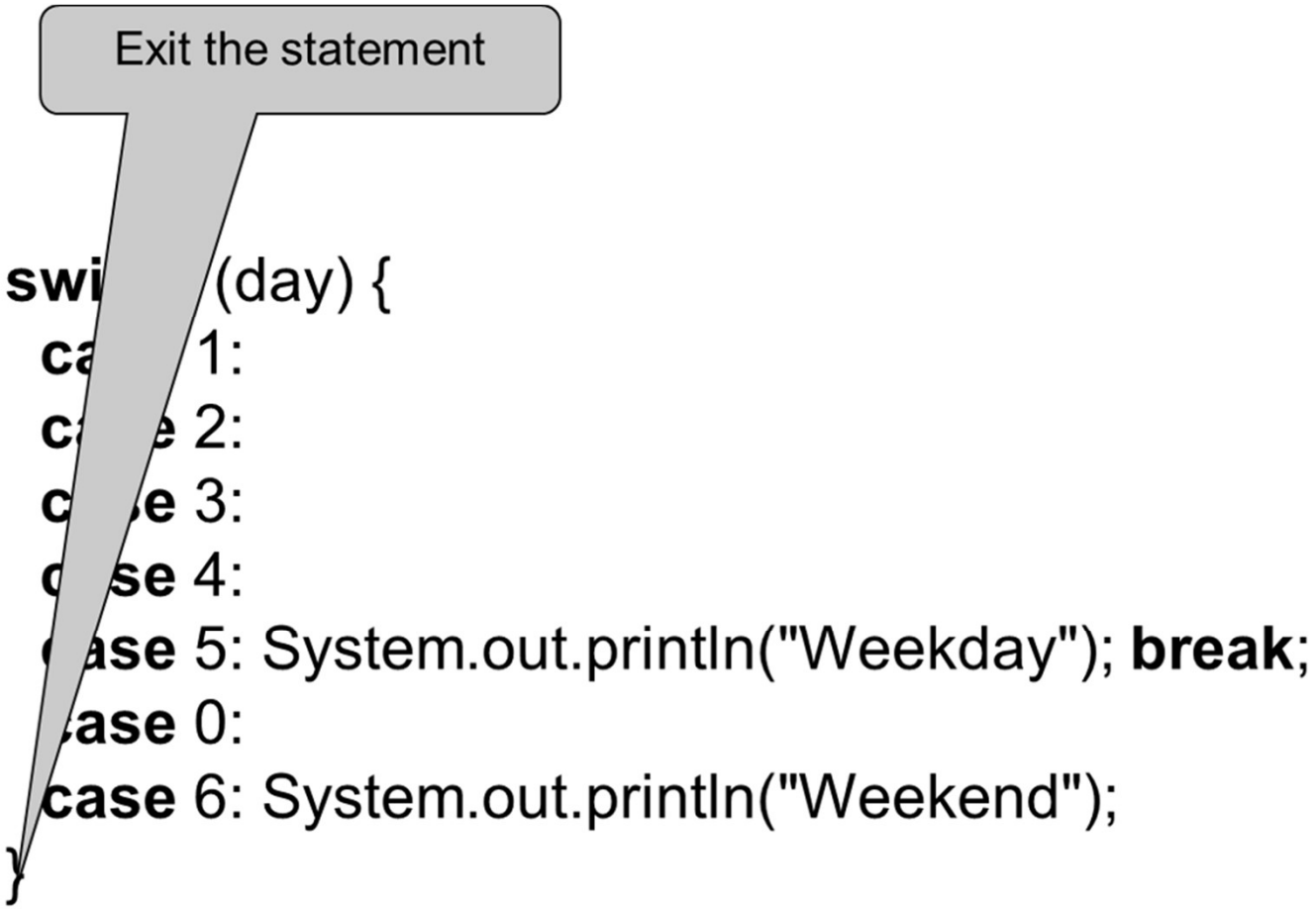
Trace `switch` Statement (5 of 7)



Trace `switch` Statement (6 of 7)



Trace `switch` Statement (7 of 7)



JDK 14 Arrow Operator (->)

Forgetting a break statement when it is needed is a common error. To avoid this type of errors, JDK 14 introduced a new arrow operator (->). You can use the arrow operator to replace the colon operator (:). With the arrow operator, there is no need to use the break statement. When a case is matched, the matching statement(s) are executed, and the switch statement is finished.

```
int day = 1;
switch (day) {
  case 1 -> System.out.print(1 + " ");
  case 2 -> System.out.print(2 + " ");
  case 3 -> System.out.print(3 + " ");
}
```

JDK 14 `switch` Expression

Java 14 also introduced switch expressions. A switch expression returns a value. Here is an example:

```
int day = 1;
    System.out.println(
      switch (day) {
        case 1 -> 1 + " ";
        case 2 -> 2 + " ";
        case 3 -> 3 + " ";
        default -> " ";
      }
);
```

The switch expression in this example returns a string. A switch expression must cover all cases, while a switch statement does not have to cover all cases. In the preceding example, the default clause is to required to cover the integers not listed in the cases.

JDK 14 `switch` Combining Cases

In Java 14, the cases can be combined. For example, the preceding code can be simplified as follows:

```
int day = 1;
System.out.println(
  switch (day) {
    case 1, 2, 3 -> day + " ";
    default -> " ";
  }
);
```

case 1, 2, 3 means case 1, case 2, or case 3.

The `yield` Keyword (1 of 2)

In Java 14, the cases can be combined. For example, the preceding code can be simplified as follows:

```
int day = 1;
System.out.println(
  switch (day) {
    case 1, 2, 3 -> day + " ";
    default -> " ";
  }
);
```

case 1, 2, 3 means case 1, case 2, or case 3.

The `yield` Keyword (2 of 2)

If the result for a matching case in a switch expression is not a simple value, you need to use the yield keyword to return the value. Here is an example,

```
int year = 2000;
   int month = 2;
   System.out.println(
     switch (month) {
       case 2 -> {
         if (isLeapYear(year))
           yield "29 days";
         else
           yield "28 days";
       }
       default -> " ";
     }
   );
```

Conditional Operators

```
if (x > 0)
  y = 1
else
   y = -1;
```

is equivalent to

```
y = (x > 0) ? 1 : -1;
```

```
(boolean-expression) ? expression1 : expression2
```

Ternary operator

Binary operator

Unary operator

Conditional Operator (1 of 2)

```
if (num % 2 == 0)
  System.out.println(num + "is even");
else
  System.out.println(num + "is odd");


System.out.println(
  (num % 2 == 0)? num + "is even" :
  num + "is odd");
```

Conditional Operator (2 of 2)

```
boolean-expression ? exp1 : exp2
```

Operator Precedence

- `var++, var--`
- `+, - (Unary plus and minus), ++var,--var`
- `(type) Casting`
- `! (Not)`
- `*, /, % (Multiplication, division, and remainder)`
- `+, - (Binary addition and subtraction)`
- `<, <=, >, >= (Relational operators)`
- `==, !=; (Equality)`
- `^ (Exclusive OR)`
- `&& (Conditional AND) Short-circuit AND`
- `|| (Conditional OR) Short-circuit OR`
- `=, +=, -=, *=, /=, %= (Assignment operator)`

Operator Precedence and Associativity

The expression in the parentheses is evaluated first. (Parentheses can be nested, in which case the expression in the inner parentheses is executed first.) When evaluating an expression without parentheses, the operators are applied according to the precedence rule and the associativity rule.

If operators with the same precedence are next to each other, their associativity determines the order of evaluation. All binary operators except assignment operators are left-associative.

Operator Associativity

When two operators with the same precedence are evaluated, the **associativity** of the operators determines the order of evaluation. All binary operators except assignment operators are **left-associative**.

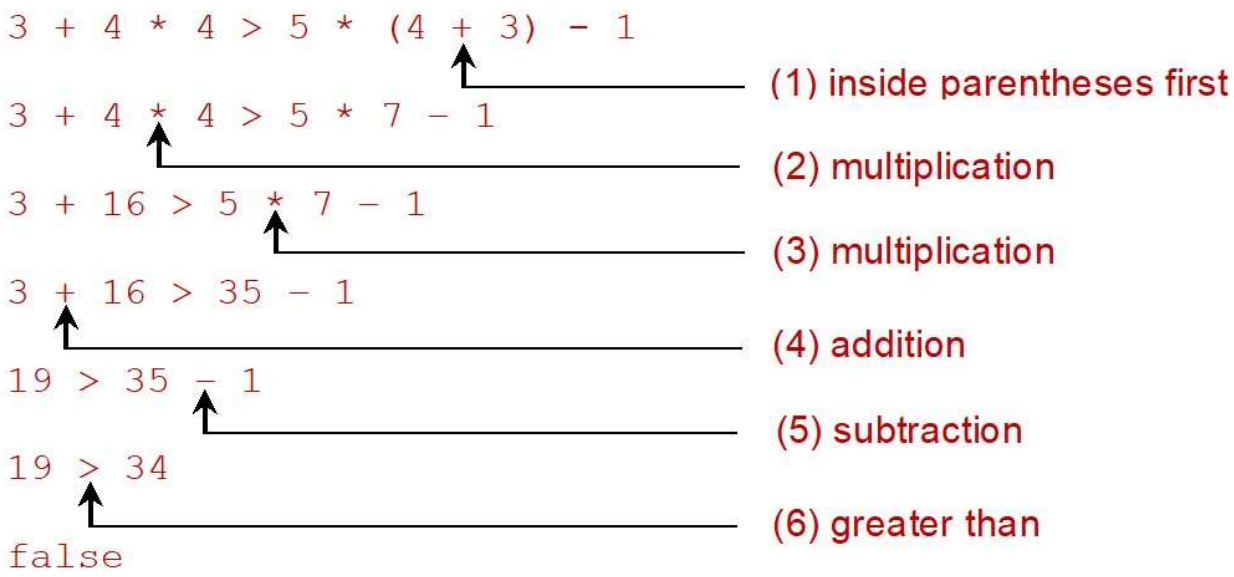
$a - b + c - d$ is equivalent to $((a - b) + c) - d$

Assignment operators are **right-associative**. Therefore, the expression

$a = b += c = 5$ is equivalent to $a = (b += (c = 5))$

Example

Applying the operator precedence and associativity rule, the expression $3 + 4 * 4 > 5 * (4 + 3) - 1$ is evaluated as follows:



Debugging

Logic errors are called **bugs**. The process of finding and correcting errors is called debugging. A common approach to debugging is to use a combination of methods to narrow down to the part of the program where the bug is located. You can hand-trace the program (i.e., catch errors by reading the program), or you can insert print statements in order to show the values of the variables or the execution flow of the program. This approach might work for a short, simple program. But for a large, complex program, the most effective approach for debugging is to use a debugger utility.

Debugger

Debugger is a program that facilitates debugging. You can use a debugger to

- Execute a single statement at a time.
- Trace into or stepping over a method.
- Set breakpoints.
- Display variables.
- Display call stack.
- Modify variables.