CHAPTER 3 - FLOW OF CONTROL

CSC111

DR. SULTAN ALFARHOOD

sultanf@ksu.edu.sa

OUTLINE

- Basic `if-else` Statement
- Boolean Expressions
- Comparing Strings
- Nested `if-else` Statements
- Multibranch `if-else` Statements
- The `exit` Method
- The `switch` Statement

FLOW OF CONTROL

- *Flow of control* is the order in which a program performs actions.
  - Up to this point, the order has been sequential.
- A *branching statement* chooses between two or more possible actions.
- A *loop statement* repeats an action until a stopping condition occurs.

THE `IF-ELSE` STATEMENT

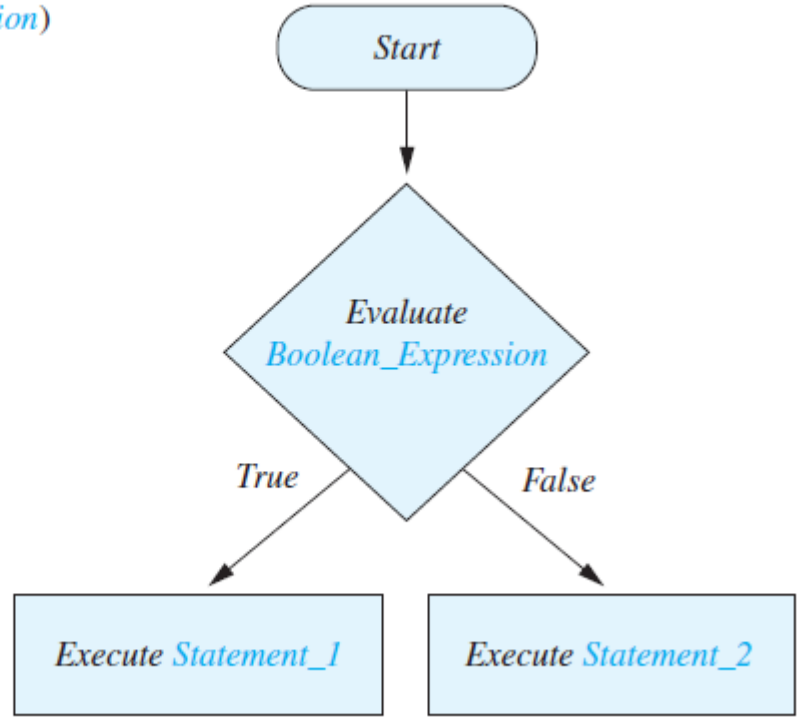
- A branching statement that chooses between two possible actions.
- Syntax

```
if (Boolean_Expression)
  Statement_1
else
  Statement_2
```

SEMANTICS OF THE `IF-ELSE` STATEMENT

FIGURE 3.2 The Semantics of the `if-else` Statement

```
if (Boolean_Expression)
    Statement_1
else
    Statement_2
```



EXAMPLE

```
import java.util.Scanner;

public class PassFail {
    public static void main(String args[]) {

        double score=0;
        Scanner input= new Scanner(System.in);

        System.out.print("Enter your score: ");
        score= input.nextDouble();

        if( score >= 60 )
            System.out.println("Congratulation! you have passed :)");
        else
            System.out.println("Sorry, you have failed.\nWork hard next semester.");
    }
}
```

EXAMPLE

```
Enter your score: 98
Congratulation! you have passed :)
```

```
Enter your score: 50
Sorry, you have failed.
Work hard next semester.
```

COMPOUND STATEMENTS

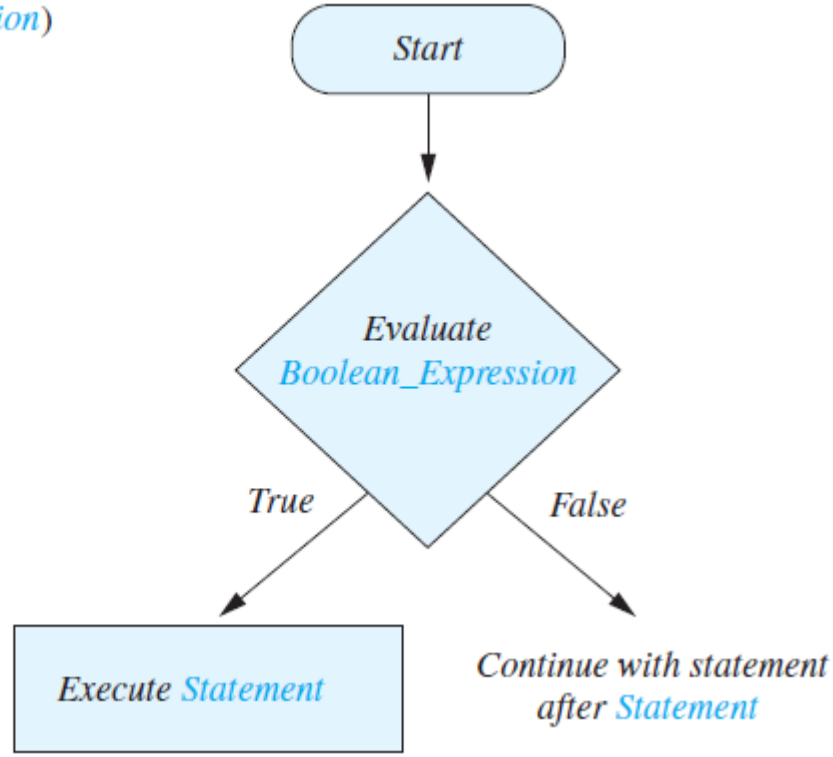
- To include multiple statements in a branch, enclose the statements in braces.

```
if (count < 3)
{
    total = 0;
    count = 0;
}
```

OMITTING THE `ELSE` PART

- The Semantics of an `if` Statement without an `else`

```
if (Boolean_Expression)
    Statement
```



INTRODUCTION TO BOOLEAN EXPRESSIONS

- The value of a *boolean expression* is either `true` or `false`.
- Examples

```
time < limit
balance <= 0
```

JAVA COMPARISON OPERATORS

- Java Comparison Operators

Math Notation	Name	Java Notation	Java Examples
=	Equal to	==	`balance == 0` `answer == 'y'`
≠	Not equal to	!=	`income != tax` `answer != 'y'`
>	Greater than	>	`expenses > income`
≥	Greater than or equal to	>=	`points >= 60`
<	Less than	<	`pressure < max`
≤	Less than or equal to	<=	`expenses <= income`

COMPOUND BOOLEAN EXPRESSIONS

- Boolean expressions can be combined using the "and" `(&&)` operator.
- Example

```
if ((score > 0) && (score <= 100))
...
```

- Not allowed

```
if (0 < score <= 100)
...
```

COMPOUND BOOLEAN EXPRESSIONS

- Syntax

```
(Sub_Expression_1) && (Sub_Expression_2)
```

- Parentheses often are used to enhance readability.
- The larger expression is true only when both of the smaller expressions are true.

COMPOUND BOOLEAN EXPRESSIONS

- Boolean expressions can be combined using the "or" `(||)` operator.
- Example

```
if ((quantity > 5) || (cost < 10))
...
```

- Syntax

```
(Sub_Expression_1) || (Sub_Expression_2)
```

COMPOUND BOOLEAN EXPRESSIONS

- The larger expression is true
  - When either of the smaller expressions is true
  - When both of the smaller expressions are true.
- The Java version of "or" is the *inclusive or* which allows either or both to be true.
- The *exclusive or* allows one or the other, but not both to be true.

NEGATING A BOOLEAN EXPRESSION

- A boolean expression can be negated using the "not" (!) operator.
- Syntax

```
!(Boolean_Expression)
```

- Example

```
(a || b) && !(a && b)
```

which is the *exclusive or*

NEGATING A BOOLEAN EXPRESSION

- Avoiding the Negation Operator

! (*A Op B*) Is Equivalent to (*A Op B*)	
<	>=
<=	>
>	<=
>=	<
==	!=
!=	==

JAVA LOGICAL OPERATORS

Name	Java Notation	Java Examples
Logical *and*	&&	`(sum > min) && (sum < max)`
Logical *or*	||	`(answer == 'y') || (answer == 'Y')`
Logical *not*	!	`!(number < 0)`

BOOLEAN OPERATORS

- The Effect of the Boolean Operators **&&** (and), **||** (or), and **!** (not) on Boolean values

Value of *A*	Value of *B*	Value of *A* && *B*	Value of *A* || *B*	Value of ! (*A*)
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

USING ==

- == is appropriate for determining if two integers or characters have the same value.

```
if (a == 3)
```

where a is an integer type

- == is **not** appropriate for determining if two floating points values are equal. Use < and some appropriate tolerance instead.

```
if (abs(b - c) < epsilon)
```

where b, c, and epsilon are floating point types

USING ==

- == is not appropriate for determining if two objects have the same value.
  - `if (s1 == s2)`, where `s1` and `s2` refer to strings, determines only if s1 and s2 refer the a common memory location.
  - If `s1` and `s2` refer to strings with identical sequences of characters, but stored in different memory locations, `(s1 == s2)` is false.

USING ==

- To test the equality of objects of class String, use method **`equals`**.

```
s1.equals(s2)
```

or

```
s2.equals(s1)
```

- To test for equality ignoring case, use method **`equalsIgnoreCase`**.

```
("Hello".equalsIgnoreCase("hello"))
```

EQUALS AND EQUALSIGNORECASE

- Syntax

```
String.equals(Other_String)
String.equalsIgnoreCase(Other_String)
```

TESTING STRINGS FOR EQUALITY EXAMPLE

LISTING 3.2 Testing Strings for Equality *(part 1 of 2)*

```
import java.util.Scanner;
public class StringEqualityDemo
{
    public static void main(String[] args)
    {
        String s1, s2;
        System.out.println("Enter two lines of text:");
        Scanner keyboard = new Scanner(System.in);
        s1 = keyboard.nextLine();
        s2 = keyboard.nextLine();

        if (s1.equals(s2))
           System.out.println("The two lines are equal.");
        else
           System.out.println("The two lines are not equal.");

        if (s2.equals(s1))
           System.out.println("The two lines are equal.");
        else
           System.out.println("The two lines are not equal.");

        if (s1.equalsIgnoreCase(s2))
           System.out.println(
                      "But the lines are equal, ignoring case.");
        else
           System.out.println(
                      "Lines are not equal, even ignoring case.");
    }
}
```

These two invocations of the method `equals` *are equivalent.*

TESTING STRINGS FOR EQUALITY

- `class StringEqualityDemo`

Sample Screen Output

```
Enter two lines of text:
Java is not coffee.
Java is NOT COFFEE.
The two lines are not equal.
The two lines are not equal.
But the lines are equal, ignoring case.
```

LEXICOGRAPHIC ORDER

- Lexicographic order is similar to alphabetical order, but is it based on the order of the characters in the ASCII (and Unicode) character set.
  - All the digits come before all the letters.
  - All the uppercase letters come before all the lower case letters.

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	*[NULL]*
1	1	1	1	*[START OF HEADING]*
2	2	10	2	*[START OF TEXT]*
3	3	11	3	*[END OF TEXT]*
4	4	100	4	*[END OF TRANSMISSION]*
5	5	101	5	*[ENQUIRY]*
6	6	110	6	*[ACKNOWLEDGE]*
7	7	111	7	*[BELL]*
8	8	1000	10	*[BACKSPACE]*
9	9	1001	11	*[HORIZONTAL TAB]*
10	A	1010	12	*[LINE FEED]*
11	B	1011	13	*[VERTICAL TAB]*
12	C	1100	14	*[FORM FEED]*
13	D	1101	15	*[CARRIAGE RETURN]*
14	E	1110	16	*[SHIFT OUT]*
15	F	1111	17	*[SHIFT IN]*
16	10	10000	20	*[DATA LINK ESCAPE]*
17	11	10001	21	*[DEVICE CONTROL 1]*
18	12	10010	22	*[DEVICE CONTROL 2]*
19	13	10011	23	*[DEVICE CONTROL 3]*
20	14	10100	24	*[DEVICE CONTROL 4]*
21	15	10101	25	*[NEGATIVE ACKNOWLEDGE]*
22	16	10110	26	*[SYNCHRONOUS IDLE]*
23	17	10111	27	*[ENG OF TRANS. BLOCK]*
24	18	11000	30	*[CANCEL]*
25	19	11001	31	*[END OF MEDIUM]*
26	1A	11010	32	*[SUBSTITUTE]*
27	1B	11011	33	*[ESCAPE]*
28	1C	11100	34	*[FILE SEPARATOR]*
29	1D	11101	35	*[GROUP SEPARATOR]*
30	1E	11110	36	*[RECORD SEPARATOR]*
31	1F	11111	37	*[UNIT SEPARATOR]*
32	20	100000	40	*[SPACE]*
33	21	100001	41	!
34	22	100010	42	"
35	23	100011	43	#
36	24	100100	44	$
37	25	100101	45	%
38	26	100110	46	&
39	27	100111	47	'
40	28	101000	50	(
41	29	101001	51	)
42	2A	101010	52	*
43	2B	101011	53	+
44	2C	101100	54	,
45	2D	101101	55	-
46	2E	101110	56	.
47	2F	101111	57	/
48	30	110000	60	0
49	31	110001	61	1
50	32	110010	62	2
51	33	110011	63	3
52	34	110100	64	4
53	35	110101	65	5
54	36	110110	66	6
55	37	110111	67	7
56	38	111000	70	8
57	39	111001	71	9
58	3A	111010	72	:
59	3B	111011	73	;
60	3C	111100	74	<
61	3D	111101	75	=
62	3E	111110	76	>
63	3F	111111	77	?
64	40	1000000	100	@
65	41	1000001	101	A
66	42	1000010	102	B
67	43	1000011	103	C
68	44	1000100	104	D
69	45	1000101	105	E
70	46	1000110	106	F
71	47	1000111	107	G
72	48	1001000	110	H
73	49	1001001	111	I
74	4A	1001010	112	J
75	4B	1001011	113	K
76	4C	1001100	114	L
77	4D	1001101	115	M
78	4E	1001110	116	N
79	4F	1001111	117	O
80	50	1010000	120	P
81	51	1010001	121	Q
82	52	1010010	122	R
83	53	1010011	123	S
84	54	1010100	124	T
85	55	1010101	125	U
86	56	1010110	126	V
87	57	1010111	127	W
88	58	1011000	130	X
89	59	1011001	131	Y
90	5A	1011010	132	Z
91	5B	1011011	133	[
92	5C	1011100	134	\
93	5D	1011101	135	]
94	5E	1011110	136	^
95	5F	1011111	137	_
96	60	1100000	140	`
97	61	1100001	141	a
98	62	1100010	142	b
99	63	1100011	143	c
100	64	1100100	144	d
101	65	1100101	145	e
102	66	1100110	146	f
103	67	1100111	147	g
104	68	1101000	150	h
105	69	1101001	151	i
106	6A	1101010	152	j
107	6B	1101011	153	k
108	6C	1101100	154	l
109	6D	1101101	155	m
110	6E	1101110	156	n
111	6F	1101111	157	o
112	70	1110000	160	p
113	71	1110001	161	q
114	72	1110010	162	r
115	73	1110011	163	s
116	74	1110100	164	t
117	75	1110101	165	u
118	76	1110110	166	v
119	77	1110111	167	w
120	78	1111000	170	x
121	79	1111001	171	y
122	7A	1111010	172	z
123	7B	1111011	173	{
124	7C	1111100	174	|
125	7D	1111101	175	}
126	7E	1111110	176	~
127	7F	1111111	177	*[DEL]*

LEXICOGRAPHIC ORDER

- Strings consisting of alphabetical characters can be compared using method `compareTo` and method `toUpperCase` or method `toLowerCase`.

```
String s1 = "Hello";
String lowerS1 = s1.toLowerCase();
String s2 = "hello";
if (s1.compareTo(s2) == 0)
    System.out.println("Equal!");
```

METHOD COMPARETO

- Syntax

```
String_1.compareTo(String_2)
```

- Method `compareTo` returns
  - a negative number if `String_1` precedes `String_2`
  - zero if the two strings are equal
  - a positive number of `String_2` precedes `String_1`.

NESTED `IF-ELSE` STATEMENTS

- An `if-else` statement can contain any sort of statement within it.
- In particular, it can contain another `if-else` statement.
  - An `if-else` may be nested within the "if" part.
  - An `if-else` may be nested within the "else" part.
  - An `if-else` may be nested within both parts.

NESTED STATEMENTS

- Syntax

```
if (Boolean_Expression_1)
    if (Boolean_Expression_2)
        Statement_1
    else
        Statement_2
else
   if (Boolean_Expression_3)
        Statement_3
    else
        Statement_4
```

NESTED STATEMENTS

- Each `else` is paired with the nearest unmatched `if`.
- **If used properly**, indentation communicates which `if` goes with which `else`.
- Braces can be used like parentheses to group statements.

NESTED STATEMENTS

- Subtly different forms

First Form

```
if (a > b)
{
    if (c > d)
        e = f;
}
    else
        g = h;
```

Second Form

```
if (a > b)
    if (c > d)
        e = f;
    else
        g = h;
// oops
```

COMPOUND STATEMENTS

- When a list of statements is enclosed in braces ({ }), they form a single *compound statement.*
- Syntax

```
{
    Statement_1;
    Statement_2;
  …
}
```

COMPOUND STATEMENTS

- A compound statement can be used wherever a statement can be used.
- Example

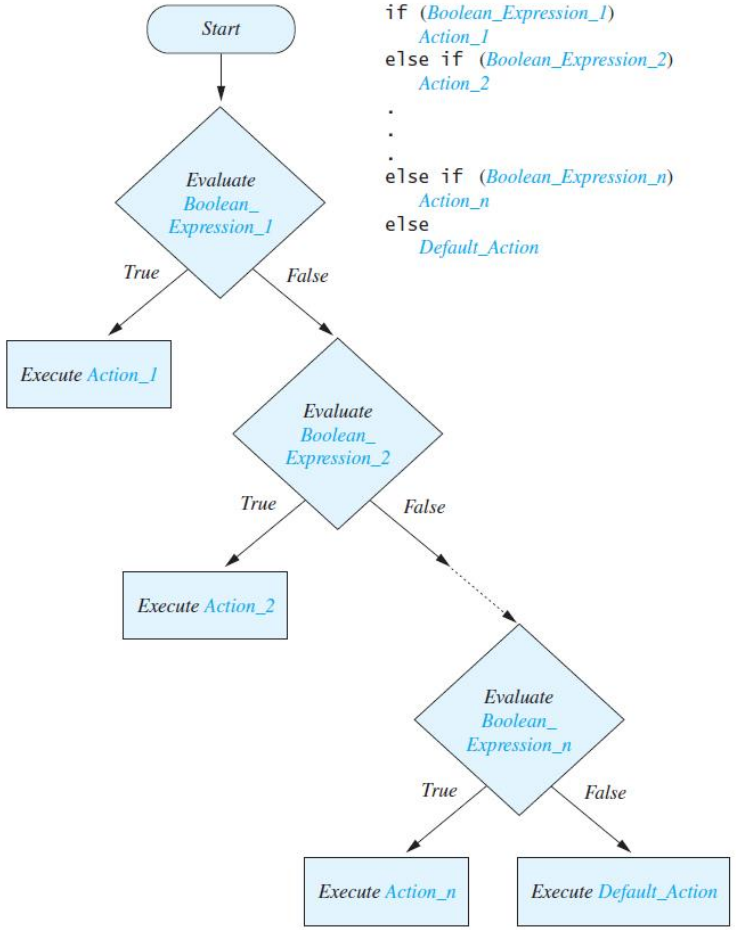
```
if (total > 10)
{
    sum = sum + total;
    total = 0;
}
```

MULTIBRANCH `IF-ELSE` STATEMENTS

- Syntax

```
if (Boolean_Expression_1)
    Statement_1
else if (Boolean_Expression_2)
    Statement_2
else if (Boolean_Expression_3)
    Statement_3
else if …
else
    Default_Statement
```

MULTIBRANCH `IF-ELSE` STATEMENTS



MULTIBRANCH IF-ELSE STATEMENTS EXAMPLE

LISTING 3.3 **Assigning Letter Grades Using a Multibranch `if-else` Statement** *(part 1 of 2)*

```
import java.util.Scanner;
public class Grader
{
    public static void main(String[] args)
    {
        int score;
        char grade;

        System.out.println("Enter your score: ");
        Scanner keyboard = new Scanner(System.in);
        score = keyboard.nextInt();

        if (score >= 90)
            grade = 'A';
        else if (score >= 80)
            grade = 'B';
        else if (score >= 70)
            grade = 'C';
        else if (score >= 60)
            grade = 'D';
        else
            grade = 'F';
        System.out.println("Score = " + score);
        System.out.println("Grade = " + grade);
    }
}
```

MULTIBRANCH `IF-ELSE` STATEMENTS

- `class Grader`

```
Enter your score:
85
Score = 85
Grade = B
```

Sample screen output

MULTIBRANCH IF-ELSE STATEMENTS

- Equivalent code

```
if (score >= 90)
    grade = 'A';
else if ((score >= 80) && (score < 90))
    grade = 'B';
else if ((score >= 70) && (score < 80))
    grade = 'C';
else if ((score >= 60) && (score < 70))
    grade = 'D';
else
    grade = 'F';
```

CHALLENGE

```
//What does this program print?
public class Challenge1 {
    public static void main(String args[]) {
        int x=1,y=2;
        int z=y=x;
        if(++x < x++)
            x=y;
        else if( (x=y) <= ++z)
            x*=2;
        x*= y*z/2;
        System.out.println("x= " + x);
    }
}
```

CASE STUDY – BODY MASS INDEX

- Body Mass Index (BMI) is used to estimate the risk of weight-related problems
- BMI = mass / height^2
  - Mass in kilograms, height in meters
- Health assessment if:
  - BMI < 18.5 Underweight
  - 18.5 ≤ BMI < 25 Normal weight
  - 25 ≤ BMI < 30 Overweight
  - 30 ≤ BMI Obese

CASE STUDY – BODY MASS INDEX

- Algorithm
  - Input height in feet & inches, weight in pounds
  - Convert to meters and kilograms
    - 1 lb = 2.2 kg
    - 1 inch = 0.254 meters
  - Compute BMI
  - Output health risk using if statements

BODY MASS INDEX EXAMPLE

LISTING 3.4 A Body Mass Index Calculation Program *(part 1 of 2)*

```
import java.util.Scanner;
public class BMI
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        int pounds, feet, inches;
        double heightMeters, mass, BMI;
        System.out.println("Enter your weight in pounds.");
        pounds = keyboard.nextInt();
        System.out.println("Enter your height in feet" +
           "followed by a space" +
           "then additional inches.");
        feet = keyboard.nextInt();
        inches = keyboard.nextInt();
        heightMeters = ((feet * 12) + inches) * 0.0254;
        mass = (pounds / 2.2);
        BMI = mass / (heightMeters * heightMeters);
        System.out.println("Your BMI is " + BMI);
        System.out.print("Your risk category is ");
        if (BMI < 18.5)
           System.out.println("Underweight.");
        else if (BMI < 25)
           System.out.println("Normal weight.");
```

BODY MASS INDEX EXAMPLE

LISTING 3.4 A Body Mass Index Calculation Program *(part 2 of 2)*

```
            else if (BMI < 30)
               System.out.println("Overweight.");
            else
               System.out.println("Obese.");
        }
    }
```

Sample Screen Output

```
Enter your weight in pounds.
150
Enter your height in feet followed
by a space then additional inches.
5 5
Your BMI is 25.013498117367398
Your risk category is Overweight.
```

THE `EXIT` METHOD

- Sometimes a situation arises that makes continuing the program pointless.
- A program can be terminated normally by

```
System.exit(0);
```

THE EXIT METHOD

- Example

```
if (numberOfWinners == 0)
{
    System.out.println ("Error: Dividing by zero.");
    System.exit (0);
}
else
{
    oneShare = payoff / numberOfWinners;
    System.out.println ("Each winner will receive $" + oneShare);
}
```

THE TYPE BOOLEAN

- The type `boolean` is a primitive type with only two values: `true` and `false`.
- Boolean variables can make programs more readable.

```
if (systemsAreOK)
```

instead of

```
if((temperature <= 100) && (thrust >= 12000) &&
  (cabinPressure > 30) && …)
```

BOOLEAN EXPRESSIONS AND VARIABLES

- Variables, constants, and expressions of type **`boolean`** all evaluate to either **`true`** or **`false`**.
- A boolean variable can be given the value of a boolean expression by using an assignment operator.

```
boolean isPositive = (number > 0);
...
if (isPositive) ...
```

NAMING BOOLEAN VARIABLES

- Choose names such as `isPositive` or `systemsAreOk`.
- Avoid names such as `numberSign` or `systemStatus`.

PRECEDENCE RULES

- Parentheses should be used to indicate the order of operations.
- When parentheses are omitted, the order of operation is determined by *precedence rules.*

PRECEDENCE RULES

- Operations with *higher precedence* are performed before operations with *lower precedence.*
- Operations with *equal precedence* are done left-to-right (except for unary operations which are done right-to-left).

PRECEDENCE RULES

Highest Precedence

First: the unary operators +, −, ++, −−, and !

Second: the binary arithmetic operators *, /, %

Third: the binary arithmetic operators +, −

Fourth: the boolean operators <, >, <=, >=

Fifth: the boolean operators ==, !=

Sixth: the boolean operator &

Seventh: the boolean operator |

Eighth: the boolean operator &&

Ninth: the boolean operator ||

Lowest Precedence

PRECEDENCE RULES

- In what order are the operations performed?

```
score < min/2 - 10 || score > 90
score < (min/2) - 10 || score > 90
score < ((min/2) - 10) || score > 90
(score < ((min/2) - 10)) || score > 90
(score < ((min/2) - 10)) || (score > 90)
```

SHORT-CIRCUIT EVALUATION

- Sometimes only part of a boolean expression needs to be evaluated to determine the value of the entire expression.
  - If the first operand associated with an `||` is `true`, the expression is `true`.
  - If the first operand associated with an `&&` is `false`, the expression is `false`.
- This is called *short-circuit* or *lazy* evaluation.

SHORT-CIRCUIT EVALUATION

- Short-circuit evaluation is not only efficient, sometimes it is essential!
- A run-time error can result, for example, from an attempt to divide by zero.

```
if ((number != 0) && (sum/number > 5))
```

- *Complete evaluation* can be achieved by substituting `&` for `&&` or `|` for `||`.

EXAMPLE

```
public class ShortCircuitExample
{
    public static void main(String args[])
    {
        int x=10,y=20;

        if (x>15 && y++<50)
            x=0;
        System.out.println("x= " + x +" ; y= "+y);

        if (x>15 & y++<50)
            x=0;
        System.out.println("x= " + x +" ; y= "+y);
    }
}
```

```
x= 10 ; y= 20
x= 10 ; y= 21
```

INPUT AND OUTPUT OF BOOLEAN VALUES

- Example

```
boolean booleanVar = false;
System.out.println(booleanVar);

Scanner keyboard = new Scanner(System.in);
System.out.println("Enter a boolean value:");
booleanVar = keyboard.nextBoolean();
System.out.println("You entered " + booleanVar);
```

INPUT AND OUTPUT OF BOOLEAN VALUES

- Dialog:

```
false
Enter a boolean value:
true
You entered true
```

INPUT VALIDATION

- You should check your input to ensure that it is within a valid or reasonable range. For example, consider a program that converts feet to inches. You might write the following:

```
int feet = keyboard.nextInt();
int inches = feet * 12;
```

- What if:
  - The user types a negative number for feet?
  - The user enters an unreasonable value like 100? Or a number larger than can be stored in an int? (2,147,483,647)

INPUT VALIDATION

- Address these problems by ensuring that the entered values are reasonable:

```
int feet = keyboard.nextInt();
if ((feet >= 0) && (feet < 10))
{
    int inches = feet * 12;
    ...
}
```

THE `SWITCH` STATEMENT

- The `switch` statement is a mutltiway branch that makes a decision based on an *integral* (integer or character) expression.
  - Java 7 allows String expressions
- The `switch` statement begins with the keyword `switch` followed by an integral expression in parentheses and called the *controlling expression.*

THE `SWITCH` STATEMENT

- A list of cases follows, enclosed in braces.
- Each case consists of the keyword `case` followed by
  - A constant called the *case label*
  - A colon
  - A list of statements.
- The list is searched for a case label matching the controlling expression.

THE `SWITCH` STATEMENT

- The action associated with a matching case label is executed.
- If no match is found, the case labeled `default` is executed.
  - The `default` case is optional, but recommended, even if it simply prints a message.
- Repeated case labels are not allowed.

THE SWITCH STATEMENT

- Syntax

```
switch (Controlling_Expression)
{
    case Case_Label:
          Statement(s);
          break;
    case Case_Label:
    …
    default:
    …
}
```

LISTING 3.5 A switch Statement *(part 1 of 2)*

```
import java.util.Scanner;
public class MultipleBirths
{
    public static void main(String[] args)
    {
        int numberOfBabies;
        System.out.print("Enter number of babies: ");
        Scanner keyboard = new Scanner(System.in);
        numberOfBabies = keyboard.nextInt();

        switch (numberOfBabies)
        {
           case 1:
               System.out.println("Congratulations.");
               break;
           case 2:
               System.out.println("Wow. Twins.");
               break;
           case 3:
               System.out.println("Wow. Triplets.");
               break;
           case 4:
           case 5:
               System.out.print("Unbelievable; ");
               System.out.println(numberOfBabies +
                                  " babies.");
               break;
           default:
               System.out.println("I don't believe you.");
               break;
        }
    }
}
```

Controlling expression

Case label

break *statement*

Case with no break

THE SWITCH STATEMENT

Sample Screen Output 1

```
Enter number of babies: 1
Congratulations.
```

Sample Screen Output 2

```
Enter number of babies: 3
Wow. Triplets.
```

Sample Screen Output 3

```
Enter number of babies: 4
Unbelievable; 4 babies.
```

Sample Screen Output 4

```
Enter number of babies: 6
I don't believe you.
```

THE `SWITCH` STATEMENT

- The action for each case typically ends with the word **`break`**.
- The optional **`break`** statement prevents the consideration of other cases.
- The controlling expression can be anything that evaluates to an integral type.