

Flow of Control

# CHAPTER 3

---

Slides prepared by Rose Williams, *Binghamton University*

# Flow of Control

- As in most programming languages, *flow of control* in Java refers to its *branching* and *looping* mechanisms
- Java has several branching mechanisms: **if-else**, **if**, and **switch** statements
- Java has three types of loop statements: the **while**, **do-while**, and **for** statements
- Most branching and looping statements are controlled by Boolean expressions
  - A Boolean expression evaluates to either **true** or **false**
  - The primitive type **boolean** may only take the values **true** or **false**

# Boolean Expressions

```
if (myScore > yourScore) {  
    System.out.println("I win!");  
}  
else {  
    System.out.println  
        ("I wish these were golf scores.");  
}
```

# Boolean Expressions

- A Boolean expression is an expression that is either **true** or **false**
- The simplest Boolean expressions compare the value of two expressions

**time < limit**

**yourScore == myScore**

- Note that Java uses two equal signs (**==**) to perform equality testing: A single equal sign (**=**) is used only for assignment
- A Boolean expression does not need to be enclosed in parentheses, unless it is used in an **if-else** statement

# Java Comparison Operators

**Display 3.3 Java Comparison Operators**

MATH NOTATION	NAME	JAVA NOTATION	JAVA EXAMPLES
=	Equal to	==	<code>x + 7 == 2*y</code> <code>answer == 'y'</code>
≠	Not equal to	!=	<code>score != 0</code> <code>answer != 'y'</code>
>	Greater than	>	<code>time &gt; limit</code>
≥	Greater than or equal to	>=	<code>age &gt;= 21</code>
<	Less than	<	<code>pressure &lt; max</code>
≤	Less than or equal to	<=	<code>time &lt;= limit</code>

# Boolean Expressions

- $1 \neq 2$
- $3 > 4$
- $5 \leq 6$
- $!(1 \neq 2 \ \&\& \ 3 > 4 \ || \ 5 \leq 6)$

# Building Boolean Expressions

- When two Boolean expressions are combined using the *"and"* (`&&`) operator, the entire expression is true provided both expressions are true
  - Otherwise the expression is false
- When two Boolean expressions are combined using the *"or"* (`||`) operator, the entire expression is true as long as one of the expressions is true
  - The expression is false only if both expressions are false
- Any Boolean expression can be negated using the `!` operator
  - Place the expression in parentheses and place the `!` operator in front of it
- Unlike mathematical notation, strings of inequalities must be joined by `&&`
  - Use `(min < result) && (result < max)` rather than `min < result < max`



# Evaluating Boolean Expressions

- Even though Boolean expressions are used to control branch and loop statements, Boolean expressions can exist independently as well
  - A Boolean variable can be given the value of a Boolean expression by using an assignment statement
- A Boolean expression can be evaluated in the same way that an arithmetic expression is evaluated
  - The only difference is that arithmetic expressions produce a number as a result, while Boolean expressions produce either **true** or **false** as their result

```
boolean madeIt = (time < limit) && (limit < max);
```



# Truth Tables

Display 3.5 Truth Tables

## AND

<i>Exp_1</i>	<i>Exp_2</i>	<i>Exp_1</i> && <i>Exp_2</i>
true	true	true
true	false	false
false	true	false
false	false	false

## OR

<i>Exp_1</i>	<i>Exp_2</i>	<i>Exp_1</i>    <i>Exp_2</i>
true	true	true
true	false	true
false	true	true
false	false	false

## NOT

<i>Exp</i>	!( <i>Exp</i> )
true	false
false	true

# Short-Circuit and Complete Evaluation

- Java can take a shortcut when the evaluation of the first part of a Boolean expression produces a result that evaluation of the second part cannot change
- This is called *short-circuit evaluation* or *lazy evaluation*
  - For example, when evaluating two Boolean subexpressions joined by `&&`, if the first subexpression evaluates to `false`, then the entire expression will evaluate to `false`, no matter the value of the second subexpression
  - In like manner, when evaluating two Boolean subexpressions joined by `||`, if the first subexpression evaluates to `true`, then the entire expression will evaluate to `true`

# Short-Circuit and Complete Evaluation

- There are times when using short-circuit evaluation can prevent a *runtime error*
  - In the following example, if the number of **kids** is equal to zero, then the second subexpression will not be evaluated, thus preventing a *divide by zero error*
  - Note that reversing the order of the subexpressions will not prevent this

```
if ((kids !=0) && ((toys/kids) >=2)) . . .
```


- Sometimes it is preferable to always evaluate both expressions, i.e., request complete evaluation
  - In this case, use the **&** and **|** operators instead of **&&** and **||**

# Precedence and Associativity Rules

- Boolean and arithmetic expressions need not be fully parenthesized
- If some or all of the parentheses are omitted, Java will follow *precedence* and *associativity* rules (summarized in the following table) to determine the order of operations
  - If one operator occurs higher in the table than another, it has *higher precedence*, and is grouped with its operands before the operator of lower precedence
  - If two operators have the same precedence, then *associativity rules* determine which is grouped first

# Precedence and Associativity Rules

Display 3.6 Precedence and Associativity Rules

<p>Highest Precedence (Grouped First)</p>  <p>Lowest Precedence (Grouped Last)</p>	PRECEDENCE	ASSOCIATIVITY
	From highest at top to lowest at bottom. Operators in the same group have equal precedence.	
	Dot operator, array indexing, and method invocation ., [ ], ( )	Left to right
	++ (postfix, as in x++), -- (postfix)	Right to left
	The unary operators: +, -, ++ (prefix, as in ++x), -- (prefix), and !	Right to left
	Type casts (Type)	Right to left
	The binary operators *, /, %	Left to right
	The binary operators +, -	Left to right
	The binary operators <, >, <=, >=	Left to right
	The binary operators ==, !=	Left to right
	The binary operator &	Left to right
	The binary operator	Left to right
	The binary operator &&	Left to right
	The binary operator	Left to right
	The ternary operator (conditional operator) ?:	Right to left
	The assignment operators: =, *=, /=, %=, +=, -=, &=,  =	Right to left

# Evaluating Expressions

- In general, parentheses in an expression help to document the programmer's intent
  - Instead of relying on precedence and associativity rules, it is best to include most parentheses, except where the intended meaning is obvious
- *Binding*: The association of operands with their operators
  - A fully parenthesized expression accomplishes binding for all the operators in an expression
- *Side Effects*: When, in addition to returning a value, an expression changes something, such as the value of a variable
  - The *assignment*, *increment*, and *decrement* operators all produce side effects
  - `j = i++;`
  - `i = keyboard.nextInt();`

# Rules for Evaluating Expressions

- Perform binding
  - Determine the equivalent fully parenthesized expression using the precedence and associativity rules
- Proceeding left to right, evaluate whatever subexpressions can be immediately evaluated
  - These subexpressions will be operands or method arguments, e.g., numeric constants or variables
- Evaluate each outer operation and method invocation as soon as all of its operands (i.e., arguments) have been evaluated



# Pitfall: Using == with Strings

- The equality comparison operator (==) can correctly test two values of a *primitive* type
- However, when applied to two *objects* such as objects of the **String** class, == tests to see if they are stored in the same memory location, not whether or not they have the same value
- In order to test two strings to see if they have equal values, use the method **equals**, or **equalsIgnoreCase**

```
string1.equals(string2)
```

```
string1.equalsIgnoreCase(string2)
```

– e.g. **StringComparisonDemo.java**

# Lexicographic and Alphabetical Order

- *Lexicographic* ordering is the same as *ASCII* ordering, and includes letters, numbers, and other characters
  - All uppercase letters are in alphabetic order, and all lowercase letters are in alphabetic order, but all uppercase letters come before lowercase letters
  - If **s1** and **s2** are two variables of type **String** that have been given **String** values, then **s1.compareTo(s2)** returns a negative number if **s1** comes before **s2** in lexicographic ordering, returns zero if the two strings are equal, and returns a positive number if **s2** comes before **s1**
- When performing an alphabetic comparison of strings (rather than a lexicographic comparison) that consist of a mix of lowercase and uppercase letters, use the **compareToIgnoreCase** method instead

# ASCII Table

Dec	Hex	Name	Char	Ctrl-char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	Null	NUL	CTRL-@	32	20	Space	64	40	@	96	60	`
1	1	Start of heading	SOH	CTRL-A	33	21	!	65	41	A	97	61	a
2	2	Start of text	STX	CTRL-B	34	22	"	66	42	B	98	62	b
3	3	End of text	ETX	CTRL-C	35	23	#	67	43	C	99	63	c
4	4	End of xmit	EOT	CTRL-D	36	24	\$	68	44	D	100	64	d
5	5	Enquiry	ENQ	CTRL-E	37	25	%	69	45	E	101	65	e
6	6	Acknowledge	ACK	CTRL-F	38	26	&	70	46	F	102	66	f
7	7	Bell	BEL	CTRL-G	39	27	'	71	47	G	103	67	g
8	8	Backspace	BS	CTRL-H	40	28	(	72	48	H	104	68	h
9	9	Horizontal tab	HT	CTRL-I	41	29	)	73	49	I	105	69	i
10	0A	Line feed	LF	CTRL-J	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	VT	CTRL-K	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	FF	CTRL-L	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage feed	CR	CTRL-M	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	SO	CTRL-N	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	SI	CTRL-O	47	2F	/	79	4F	O	111	6F	o
16	10	Data line escape	DLE	CTRL-P	48	30	0	80	50	P	112	70	p
17	11	Device control 1	DC1	CTRL-Q	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	DC2	CTRL-R	50	32	2	82	52	R	114	72	r
19	13	Device control 3	DC3	CTRL-S	51	33	3	83	53	S	115	73	s
20	14	Device control 4	DC4	CTRL-T	52	34	4	84	54	T	116	74	t
21	15	Neg acknowledge	NAK	CTRL-U	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	SYN	CTRL-V	54	36	6	86	56	V	118	76	v
23	17	End of xmit block	ETB	CTRL-W	55	37	7	87	57	W	119	77	w
24	18	Cancel	CAN	CTRL-X	56	38	8	88	58	X	120	78	x
25	19	End of medium	EM	CTRL-Y	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	SUB	CTRL-Z	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	ESC	CTRL-[	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	FS	CTRL-\	60	3C	<	92	5C	\	124	7C	
29	1D	Group separator	GS	CTRL-]	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	RS	CTRL-^	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	US	CTRL-`	63	3F	?	95	5F	_	127	7F	DEL

# Multiple Choice Exercises

- Go to <http://bit.ly/1gpSJCu>
- Bookmark it

# Multiple Choice Exercises

- Q1: How do you get to school today?
  - A: By foot
  - B: By bike
  - C: By car
  - D: By public transportation  
(train/tram/bus)
  - E: By helicopter

# Multiple Choice Exercises

- Q2: How much do you know about Java?
  - A: Experienced
  - B: Basic concepts
  - C: Knows another OO Language
  - D: Knows another non-OO Language
  - E: No programming background

# Multiple Choice Exercises

- Q3:  $-2 * -3 < 5 \ \&\& \ 6 \leq 7 \ || \ 0 \neq 9 / 10$   
     $==?$ 
  - A: true
  - B: false
  - C: 1
  - D: 0
  - E: Don't know



# Branching with an `if-else` Statement

- An `if-else` statement chooses between two alternative statements based on the value of a Boolean expression

```
if (Boolean_Expression)
    Yes_Statement
else
    No_Statement
```

- The `Boolean_Expression` must be enclosed in parentheses
- If the `Boolean_Expression` is `true`, then the `Yes_Statement` is executed
- If the `Boolean_Expression` is false, then the `No_Statement` is executed

# Compound Statements

- Each **Yes\_Statement** and **No\_Statement** branch of an **if-else** can be made up of a single statement or many statements
- *Compound Statement*: A branch statement that is made up of a list of statements
  - A compound statement must always be enclosed in a pair of braces (**{ }**)
  - A compound statement can be used anywhere that a single statement can be used

# Compound Statements

```
if (myScore > yourScore)
{
    System.out.println("I win!");
    wager = wager + 100;
}
else
{
    System.out.println
        ("I wish these were golf scores.");
    wager = 0;
}
```

# Omitting the `else` Part

- The `else` part may be omitted to obtain what is often called an `if` statement

```
if (Boolean_Expression)
    Action_Statement
```

- If the `Boolean_Expression` is true, then the `Action_Statement` is executed
- The `Action_Statement` can be a single or compound statement
- Otherwise, nothing happens, and the program goes on to the next statement

```
if (weight > ideal)
    calorieIntake = calorieIntake - 500;
```

# Nested Statements

- **if-else** statements and **if** statements both contain smaller statements within them
  - For example, single or compound statements
- In fact, any statement at all can be used as a subpart of an **if-else** or **if** statement, including another **if-else** or **if** statement
  - Each level of a nested **if-else** or **if** should be indented further than the previous level
  - Exception: *multiway if-else* statements

# Multiway **if-else** Statements

- The multiway **if-else** statement is simply a normal **if-else** statement that nests another **if-else** statement at every **else** branch
  - It is indented differently from other nested statements
  - All of the **Boolean\_Expressions** are aligned with one another, and their corresponding actions are also aligned with one another
  - The **Boolean\_Expressions** are evaluated in order until one that evaluates to **true** is found
  - The final **else** is optional

# Multiway if-else Statement

```
if (Boolean_Expression)
    Statement_1
else if (Boolean_Expression)
    Statement_2
    :
else if (Boolean_Expression_n)
    Statement_n
else
    Statement_For_All_Other_Possibilities
```

**e.g. IncomeTax.java**



# The `switch` Statement

- The `switch` statement is the only other kind of Java statement that implements *multiway* branching
  - When a `switch` statement is evaluated, one of a number of different branches is executed
  - The choice of which branch to execute is determined by a *controlling expression* enclosed in parentheses after the keyword `switch`
    - The controlling expression must evaluate to a `char`, `int`, `short`, `byte` or `String` (not supported until Java 7/1.7, i.e., not examable)

# The `switch` Statement

- Each branch statement in a `switch` statement starts with the reserved word `case`, followed by a *constant* called a *case label*, followed by a colon, and then a sequence of statements
  - Each case label must be of the same type as the controlling expression
  - Case labels need not be listed in order or span a complete interval, but each one may appear only once
  - Each sequence of statements may be followed by a `break` statement ( `break;` )

# The `switch` Statement

- There can also be a section labeled `default`:
  - The `default` section is optional, and is usually last
  - Even if the case labels cover all possible outcomes in a given `switch` statement, it is still a good practice to include a `default` section
    - It can be used to output an error message, for example
- When the controlling expression is evaluated, the code for the case label whose value matches the controlling expression is executed
  - If no case label matches, then the only statements executed are those following the `default` label (if there is one)

# The **switch** Statement

- The **switch** statement ends when it executes a **break** statement, or when the end of the **switch** statement is reached
  - When the computer executes the statements after a case label, it continues until a **break** statement is reached
  - If the **break** statement is omitted, then after executing the code for one case, the computer will go on to execute the code for the next case
  - If the **break** statement is omitted inadvertently, the compiler will not issue an error message

# The `switch` Statement

```
switch (Controlling_Expression)
{
    case Case_Label_1:
        Statement_Sequence_1
        break;
    case Case_Label_2:
        Statement_Sequence_2
        break;

    case Case_Label_n:
        Statement_Sequence_n
        break;
    default:
        Default_Statement_Sequence
        break;
}
```

**E.g. SwitchDemo.java**

# The Conditional Operator

- The *conditional operator* is a notational variant on certain forms of the **if-else** statement

- Also called the *ternary operator* or *arithmetic if*
- The following examples are equivalent:

```
if (n1 > n2)    max = n1;  
else           max = n2;
```

vs.

```
max = (n1 > n2) ? n1 : n2;
```

- The expression to the right of the assignment operator is a *conditional operator expression*
- If the Boolean expression is true, then the expression evaluates to the value of the first expression (**n1**), otherwise it evaluates to the value of the second expression (**n2**)

# Loops

- *Loops* in Java are similar to those in other high-level languages
- Java has three types of loop statements: the **while**, the **do-while**, and the **for** statements
  - The code that is repeated in a loop is called the *body* of the loop
  - Each repetition of the loop body is called an *iteration* of the loop



# while statement

- A **while** statement is used to repeat a portion of code (i.e., the loop body) based on the evaluation of a Boolean expression
  - The Boolean expression is checked *before* the loop body is executed
    - When false, the loop body is not executed at all
  - Before the execution of each following iteration of the loop body, the Boolean expression is checked again
    - If true, the loop body is executed again
    - If false, the loop statement ends
  - The loop body can consist of a single statement, or multiple statements enclosed in a pair of braces (**{ }**)

# while Syntax

```
while (Boolean_Expression)  
    Statement
```

Or

```
while (Boolean_Expression)  
{  
    Statement_1  
    Statement_2  
    :  
    Statement_Last  
}
```

E.g. WhileDemo.java, Averager.java

# do-while Statement

- A **do-while** statement is used to execute a portion of code (i.e., the loop body), and then repeat it based on the evaluation of a Boolean expression
  - The loop body is executed at least once
    - The Boolean expression is checked *after* the loop body is executed
  - The Boolean expression is checked after each iteration of the loop body
    - If true, the loop body is executed again
    - If false, the loop statement ends
    - Don't forget to put a semicolon after the Boolean expression
  - Like the while statement, the loop body can consist of a single statement, or multiple statements enclosed in a pair of braces (**{ }**)

# do-while Syntax

```
do
    Statement
while (Boolean_Expression);
```

Or

```
do
{
    Statement_1
    Statement_2
    :
    Statement_Last
} while (Boolean_Expression);
```

# Algorithms and Pseudocode

- The hard part of solving a problem with a computer program is not dealing with the syntax rules of a programming language
- Rather, coming up with the underlying solution method is the most difficult part
- An *algorithm* is a set of precise instructions that lead to a solution
  - An algorithm is normally written in *pseudocode*, which is a mixture of programming language and a human language, like English
  - Pseudocode must be precise and clear enough so that a good programmer can convert it to syntactically correct code
  - However, pseudocode is much less rigid than code: One needn't worry about the fine points of syntax or declaring variables, for example

# The `for` Statement

- The `for` statement is most commonly used to step through an integer variable in equal increments
- It begins with the keyword `for`, followed by three expressions in parentheses that describe what to do with one or more *controlling variables*
  - The first expression tells how the control variable or variables are *initialized* or *declared* and *initialized* before the first iteration
  - The second expression determines when the loop should *end*, based on the evaluation of a Boolean expression *before* each iteration
  - The third expression tells how the control variable or variables are *updated after* each iteration of the loop body

# The `for` Statement Syntax

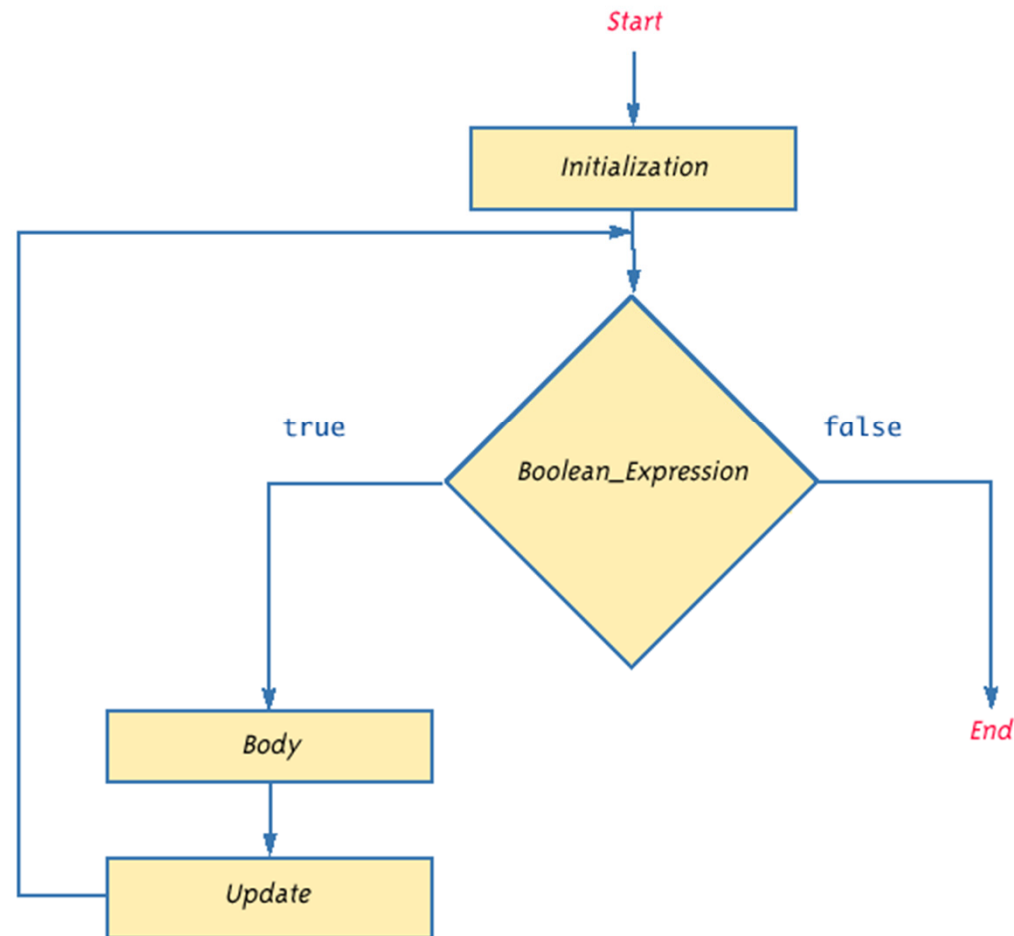
```
for (Initializing; Boolean_Expression; Update)  
    Body
```

- The **Body** may consist of a single statement or a list of statements enclosed in a pair of braces (`{ }`)
- Note that the three control expressions are separated by two, not three, semicolons
- Note that there is no semicolon after the closing parenthesis at the beginning of the loop

# Semantics of the `for` Statement

Display 3.9 Semantics of the `for` Statement

```
for (Initialization; Boolean_Expression; Update )  
    Body
```





# for Statement Syntax and Alternate Semantics

## Display 3.10 for Statement Syntax and Alternate Semantics (Part 1 of 2)

---

### for STATEMENT SYNTAX:

#### SYNTAX:

```
for (Initialization; Boolean_Expression; Update)  
    Body
```

#### EXAMPLE:

```
for (number = 100; number >= 0; number--)  
    System.out.println(number  
        + " bottles of beer on the shelf.");
```

# for Statement Syntax and Alternate Semantics

Display 3.10 for Statement Syntax and Alternate Semantics (Part 2 of 2)

---

## EQUIVALENT while LOOP:

### EQUIVALENT SYNTAX:

```
Initialization;  
while (Boolean_Expression)  
{  
    Body  
    Update;  
}
```

### EQUIVALENT EXAMPLE:

```
number = 100;  
while (number >= 0)  
{  
    System.out.println(number  
        + " bottles of beer on the shelf.");  
    number--;  
}
```

### SAMPLE DIALOGUE

```
100 bottles of beer on the shelf.  
99 bottles of beer on the shelf.  
.  
.  
.  
0 bottles of beer on the shelf.
```

# Coding Exercise

- Write a for loop that calculates the sum of the numbers from one to a hundred.

# The Comma in `for` Statements

- A `for` loop can contain multiple initialization actions separated with commas
  - Caution must be used when combining a declaration with multiple actions
  - It is illegal to combine multiple type declarations with multiple actions, for example
  - To avoid possible problems, it is best to declare all variables outside the `for` statement
- A `for` loop can contain multiple update actions, separated with commas, also
  - It is even possible to eliminate the loop body in this way
- However, a `for` loop can contain only one Boolean expression to test for ending the loop

# Infinite Loops

- A **while**, **do-while**, or **for** loop should be designed so that the value tested in the Boolean expression is changed in a way that eventually makes it false, and terminates the loop
- If the Boolean expression remains true, then the loop will run forever, resulting in an *infinite loop*
  - Loops that check for equality or inequality (**==** or **!=**) are especially prone to this error and should be avoided if possible

# Nested Loops

- Loops can be *nested*, just like other Java structures
  - When nested, the inner loop iterates from beginning to end for each single iteration of the outer loop

```
int rowNum, columnNum;
for (rowNum = 1; rowNum <=3; rowNum++)
{
    for (columnNum = 1; columnNum <=2;
        columnNum++)
        System.out.print(" row " + rowNum +
            " column " + columnNum);
    System.out.println();
}
```

# The **break** and **continue** Statements

- The **break** statement consists of the keyword **break** followed by a semicolon
  - When executed, the **break** statement ends the nearest enclosing switch or loop statement
- The **continue** statement consists of the keyword **continue** followed by a semicolon
  - When executed, the **continue** statement ends the current loop body iteration of the nearest enclosing loop statement
  - Note that in a **for** loop, the **continue** statement transfers control to the *update* expression
- When loop statements are nested, remember that any **break** or **continue** statement applies to the innermost, containing loop statement

# The Labeled **break** Statement

- There is a type of **break** statement that, when used in nested loops, can end any containing loop, not just the innermost loop
- If an enclosing loop statement is labeled with an *Identifier*, then the following version of the break statement will exit the labeled loop, even if it is not the innermost enclosing loop:  
`break someIdentifier;`
- To label a loop, simply precede it with an *Identifier* and a colon:  
`someIdentifier:`



# The `exit` Statement

- A `break` statement will end a loop or switch statement, but will not end the program
- The `exit` statement will immediately end the program as soon as it is invoked:  
`System.exit(0);`
- The `exit` statement takes one integer argument
  - By tradition, a zero argument is used to indicate a normal ending of the program

# Loop Bugs

- The two most common kinds of loop errors are unintended *infinite loops* and *off-by-one errors*
  - An off-by-one error is when a loop repeats the loop body one too many or one too few times
    - This usually results from a carelessly designed Boolean test expression
  - Use of **==** in the controlling Boolean expression can lead to an infinite loop or an off-by-one error
    - This sort of testing works only for characters and integers, and should never be used for floating-point

# Tracing Variables

- *Tracing variables* involves watching one or more variables change value while a program is running
- This can make it easier to discover errors in a program and debug them
- Many *IDEs* (*Integrated Development Environments*) have a built-in utility that allows variables to be traced without making any changes to the program
- Another way to trace variables is to simply insert temporary output statements in a program

```
System.out.println("n = " + n); // Tracing n
```

- When the error is found and corrected, the trace statements can simply be commented out

# Assertion Checks

- An *assertion* is a sentence that says (asserts) something about the state of a program
  - An assertion must be either true or false, and should be true if a program is working properly
  - Assertions can be placed in a program as comments
- Java has a statement that can check if an assertion is true

**assert Boolean\_Expression;**

- If assertion checking is turned on and the **Boolean\_Expression** evaluates to **false**, the program ends, and outputs an *assertion failed error message*
- Otherwise, the program finishes execution normally

# Assertion Checks

- A program or other class containing assertions is compiled in the usual way
- After compilation, a program can run with assertion checking turned on or turned off
  - Normally a program runs with assertion checking turned off
- In order to run a program with assertion checking turned on, use the following command (using the actual **ProgramName**):  
**java -enableassertions ProgramName**