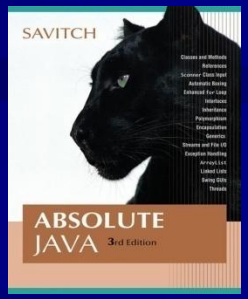
Comp 248
Introduction to Programming
Chapter 3 – *Flow of Control*

Dr. Aiman Hanna
Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada

These slides has been extracted, modified and updated from original slides of Absolute Java 3rd Edition by Savitch; which has originally been prepared by Rose Williams of Binghamton University. Absolute Java is published by Pearson Education / Addison-Wesley.



Flow of Control

- *flow of control* in Java refers to its *branching* and *looping*
- Several branching mechanisms: **`if-else`**, **`if`**, and **`switch`** statements
- Three types of loops: **`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 `if` Statement

The *condition* must be a boolean expression.
It must evaluate to either true or false.

if is a Java reserved word

```
if ( condition )
   statement;
```

If the *condition* is true, the *statement* is executed.
If it is false, the *statement* is skipped.

The `if` Statement

Example:

```
if (x > 10)
    System.out.println("Hello");
```

Compound Statements

- *Compound Statement*: If the statement under if is made up of more than one statement, they must be enclosed in curly braces ({ })

Example:

```
if (amount < balance)
{
   System.out.println("Thank you. Withdrawal will take place");
   balance = balance – amount;
}
```

- Statements1.java (MS-Word file)
- Statements2.java (MS-Word file)

`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
```

Example:

```
if (x > 10)
        System.out.println("Hello");
else
        System.out.println("Hi");
```

Compound Statements

- *Compound Statement*: Same rule; multiple statements must be enclosed in curly braces (**{ }**)

Example:

```
if (amount < balance)
{
   System.out.println("Thank you. Withdrawal will take place");
   balance = balance – amount;
}
else {
   System.out.println("Sorry. You do not have enough fund.");
   System.out.println("Transaction will be cancelled!.");
}
```

- Statements3.java (MS-Word file)

Nested Statements

- Statements within **`if-else`** or **`if`** statements can themselves be another **`if-else`** or **`if`** statements
  - For clarity, each level of a nested **`if-else`** or **`if`** should be indented further than the previous level

- Statements4.java (MS-Word file)

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
  - The **`Boolean_Expressions`** are evaluated in order until one that evaluates to **`true`** is found
  - The final **`else`** is optional

- Statements5.java (MS-Word file)
- Statements6.java (MS-Word file)

The `switch` Statement

- The general syntax of a switch statement is:

```
switch ( expression )
{
    case value1 :
       statement-list1
       break;
    case value2 :
       statement-list2
       break;
    case value3 :
       statement-list3
       break;
    case  ...

    default:
       default-statement
}
```

Can be: char, byte, short, int, or String

Note: can also be an *enumerated* type or other special classes that will be discussed later

switch case and Break are reserved words

If *expression* matches *value2*, control jumps to here

Optional

The `switch` Statement

- Statements7.java (MS-Word file)
- Statements7B.java (MS-Word file)

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;
```

- ConditionalOperator1.java (MS-Word file)

Java Comparison Operators

Display 3.3 Java Comparison Operators

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

Pitfall: Using == with Strings

- The equality comparison operator (**==**) can correctly test two values of a *primitive* type

- In order to test two strings to see if they have equal values, use the method **equals**, or **equalsIgnoreCase**

```
E.g.:
if (s1.equals(s2))
if (s1.equalsIgnoreCase(s2))
```

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

Building Boolean Expressions

!	Logical NOT
&&	Logical AND
||	Logical OR

x	!x
true	false
false	true

Truth Tables

x	y	x && y
true	true	true
true	false	false
false	true	false
false	false	false

x	y	x || y
true	true	true
true	false	true
false	true	true
false	false	false

Evaluating Boolean Expressions

- Boolean expressions can exist independently as well

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

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*

Example:

```
int x = 10, y = 15;

if (x < 4 && y == 15)          // y == 15 will NOT be evaluated
{
  …..
}
```

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

Precedence and Associativity Rules

Display 3.6 Precedence and Associativity Rules

Highest Precedence (Grouped First)

Lowest Precedence (Grouped Last)

Precedence From highest at top to lowest at bottom. Operators in the same group have equal precedence.	Associativity
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

Loops

- Java has three types of loop statements:
  - the **`while`** statements
  - the **`do-while`** statements
  - the **`for`** statement

`while` statement

- The *while statement* has the following syntax:

`while` is a reserved word

```
while ( condition )
    statement;
```

If the *`condition`* is true, the *`statement`* is executed. Then the *`condition`* is evaluated again.

The *`statement`* is executed repeatedly until the *`condition`* becomes false.

`while` Syntax

```
while (Boolean_Expression)
    Statement
```

Or, in the case where there are multiple statements

```
while (Boolean_Expression)
{
    Statement_1
    Statement_2
    …
}
```

- Statements8.java (MS-Word file)
- Statements9.java (MS-Word file)

`do-while` Statement

- The *do-while statement* has the following syntax:

`do` and `while` are reserved words

```
do
{
    statement;
}
while ( condition )
```

The ***`statement`*** is executed once initially, and then the ***`condition`*** is evaluated

The ***`statement`*** is executed repeatedly until the ***`condition`*** becomes false

- Statements10.java (MS-Word file)

The `for` Statement

- The *for statement* has the following syntax:

Reserved word

The ***initialization*** is executed once before the loop begins

The ***statement*** is executed until the ***condition*** becomes false

```
for ( initialization ; condition ; increment )
   statement;
```

The ***increment*** portion is executed at the end of each iteration

The ***condition-statement-increment*** cycle is executed repeatedly

`for` Statement Syntax and Alternate Semantics

Examples:

```
for (i=0; i <= 10; i++)
      System.out.println("Hello");
```

Or, in the case where there are multiple statements

```
for (num=100; num > 0; num = num - 20)
{
  System.out.println("looping");
  System.out.println("num is :" + num);
}
```

- Statements11.java (MS-Word file)
- Statements12.java (MS-Word file)

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

- Statements13.java (MS-Word file)

- Notice that variables declared inside for statement are local to this statement; i.e. they cannot be see outside of the statement

- Statements14.java (MS-Word file)

- Statements15.java (MS-Word file)

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

- Statements16.java (MS-Word file)

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

General Debugging Techniques

- Examine the system as a whole – don't assume the bug occurs in one particular place
- Try different test cases and check the input values
- Comment out blocks of code to narrow down the offending code
- Check common pitfalls
- Take a break and come back later
- DO NOT make random changes just hoping that the change will fix the problem!

Debugging Example (1 of 9)

- The following code is supposed to present a menu and get user input until either ‘a’ or ‘b’ is entered.

```
String s = "";
char c = ' ';
Scanner keyboard = new Scanner(System.in);

do
{
  System.out.println("Enter 'A' for option A or 'B' for option B.");
  s = keyboard.next();
  s.toLowerCase();
  c = s.substring(0,1);
}
while ((c != 'a') || (c != 'b'));
```

Debugging Example (2 of 9)

Result: Syntax error:

```
c = s.substring(0,1);     : incompatible types
found:  java.lang.String
required: char
```

- Using the "random change" debugging technique we might try to change the data type of `c` to `String`, to make the types match
- This results in more errors since the rest of the code treats `c` like a `char`

Debugging Example (3 of 9)

- First problem: substring returns a String, use charAt to get the first character:

```
String s = "";
char c = ' ';
Scanner keyboard = new Scanner(System.in);

do
{
  System.out.println("Enter 'A' for option A or 'B' for option B.");
  s = keyboard.next();
  s.toLowerCase();
  c = s.charAt(0);
}
while ((c != 'a') || (c != 'b'));
```

Now the program compiles, but it is stuck in an infinite loop. Employ tracing:

Debugging Example (4 of 9)

```
do
{
   System.out.println("Enter 'A' for option A or 'B' for option B.");
   s = keyboard.next();
   System.out.println("String s = " + s);
   s.toLowerCase();
   System.out.println("Lowercase s = " + s);
   c = s.charAt(0);
   System.out.println("c = " + c);
}
while ((c != 'a') || (c != 'b'));
```

Sample output:

```
Enter 'A' for option A or 'B' for option B.
A
String s = A
Lowercase s = A
c = A
Enter 'A' for option A or 'B' for option B.
```

From tracing we can see that the string is never changed to lowercase. Reassign the lowercase string back to `s`.

Debugging Example (5 of 9)

- The following code is supposed to present a menu and get user input until either ‘a’ or ‘b’ is entered.

```
do
{
  System.out.println("Enter 'A' for option A or 'B' for option B.");
  s = keyboard.next();
  s = s.toLowerCase();
  c = s.charAt(0);
}
while ((c != 'a') || (c != 'b'));
```

However, it’s still stuck in an infinite loop. What to try next?

Debugging Example (6 of 9)

- Could try the following "patch"

```
do
{
   System.out.println("Enter 'A' for option A or 'B' for option B.");
   s = keyboard.next();
   s = s.toLowerCase();
   c = s.charAt(0);
   if ( c == 'a')
        break;
   if (c == 'b')
        break;
}
while ((c != 'a') || (c != 'b'));
```

This works, but it is ugly! Considered a coding atrocity, it doesn't fix the underlying problem. The boolean condition after the while loop has also become meaningless. Try more tracing:

Debugging Example (7 of 9)

```
do
{
   System.out.println("Enter 'A' for option A or 'B' for option B.");
   s = keyboard.next();
   s = s.toLowerCase();
   c = s.charAt(0);
   System.out.println("c != 'a' is " + (c != 'a'));
   System.out.println("c != 'b' is " + (c != 'b'));
   System.out.println("(c != 'a') || (c != 'b')) is "
                + ((c != 'a') || (c != 'b')));
}
while ((c != 'a') || (c != 'b'));
```

Sample output:

```
Enter 'A' for option A or 'B' for option B.
A
c != 'a' is false
c != 'b' is true
(c != 'a') || (c != 'b')) is true
```

From the trace we can see that the loop's boolean expression is true because `c` cannot be not equal to `'a'` and not equal to `'b'` at the same time.

Debugging Example (8 of 9)

- Fix: We use && instead of ||

```
do
{
  System.out.println("Enter 'A' for option A or 'B' for option B.");
  s = keyboard.next();
  s = s.toLowerCase();
  c = s.charAt(0);
}
while ((c != 'a') && (c != 'b'));
```

Debugging Example (9 of 9)

- Alternative Solution: Declare a boolean variable to control the do-while loop. This makes it clear when the loop exits if we pick a meaningful variable name.

```
boolean invalidKey;
do
{
   System.out.println("Enter 'A' for option A or 'B' for option B.");
   s = keyboard.next();
   s = s.toLowerCase();
   c = s.charAt(0);
   if (c == 'a')
         invalidKey = false;
   else if (c == 'b')
         invalidKey = false;
   else
         invalidKey = true;
}
while (invalidKey);
```

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
```

Preventive Coding

- Incremental Development
  - Write a little bit of code at a time and test it before moving on
- Code Review
  - Have others look at your code
- Pair Programming
  - Programming in a team, one typing while the other watches, and periodically switch roles