

Lesson 3

Control Structures

Absolute Java

Walter Savitch

Compound Statement

- A list of statements enclosed in a pair of braces is called a compound statement.
- A compound statement is treated as a single statement by Java and may be used anywhere that a single statement may be used.

Relational/Comparison Operators

- Relational/Comparison operators are used to relate two values/variables, and form a boolean expression. The relational operators present in Java are:

| Operator | Usage | Example | Explanation |
|----------|--------------------------|---------|---------------------------------|
| < | Less than | A<B | A is less than B |
| > | Greater than | A>B | A is greater than B |
| <= | Less than or equal to | A<=B | A is less than or equal to B |
| >= | Greater than or equal to | A>=B | A is greater than or equal to B |
| == | Equal to | A==B | A is equal to B |
| != | Not equal to | A!=B | A is not equal to B |

Relational/Comparison Operators

- When testing strings for equality, do not use `==`. Instead, use either `equals` or `equalsIgnoreCase`.
- Although `==` correctly tests two values of a primitive type, such as two numbers, to see whether they are equal, it has a different meaning when applied to objects, such as objects of the class `String`.
- Recall that an object is something whose type is a class, such as a string. All strings are in the class `String` (that is, are of type `String`), so **`==` applied to two strings does not test to see whether the strings are equal. Instead, it tests whether two strings refer to the same object.**

Logical Operators

- Two conditions/boolean expressions can be logically combined with the help of logical operators. The logical operators present in Java are:

| Operator | | Usage | Example |
|----------|-----|--|------------------|
| && | AND | The compound condition evaluates to true, if both the conditions in the compound condition evaluate to true. | ((a>b) && (a>c)) |
| | OR | The compound condition evaluates to true, if any or both the conditions in the compound condition evaluate to true. | ((a>b) (a>c)) |
| ! | NOT | It negates the condition. That is: <ul style="list-style-type: none">•If the condition evaluates to true, it makes it false.•If the condition evaluates to false, it makes it true. | !(a>b) |

Truth-Table for AND Operator

| A | B | Result |
|-------|-------|--------|
| False | False | False |
| False | True | False |
| True | False | False |
| True | True | True |

Truth-Table for OR Operator

| A | B | Result |
|-------|-------|--------|
| False | False | False |
| False | True | True |
| True | False | True |
| True | True | True |

Truth-Table for Not Operator

| A | Result |
|-------|--------|
| False | True |
| True | False |

Conditional Constructs

- They help us to execute a set of statements based on a condition/ Boolean expression.
- There are two conditional constructs present in Java.
- **if...else construct:** An if-else statement chooses between two alternative statements based on the value of a Boolean expression.
- **switch...case construct**
- **If there is only one statement associated with a case, then the curly braces are optional.**
- ***else* is always optional with a particular *if*.**

if construct

| Syntax | Example |
|--|---|
| <pre>if(<condition>) statement;</pre> <pre>if(<condition>) { statement 1; statement 2; . . . statement n; }</pre> | <pre>if(a>b) System.out.println(a);</pre> <pre>if(a>b) { System.out.println("diff=" + (a-b)); System.out.println("quotient=" + (a/b)); }</pre> |

if construct

```
if(<condition>)  
    statement;  
else  
    statement;
```

```
if(<condition>)  
{  
    statement 1;  
    statement 2;  
    .  
    .  
    .  
    statement n;  
}  
else  
{  
    statement 1;  
    statement 2;  
    .  
    .  
    statement n;  
}
```

```
if(a>b)  
    System.out.println(a);  
else  
    System.out.println(b);
```

```
if(a>b)  
{  
    System.out.println("diff=" + (a-b));  
    System.out.println("quotient=" + (a/b));  
}  
else  
{  
    System.out.println("diff=" + (b-a));  
    System.out.println("quotient=" + (b/a));  
}
```

if construct

```
if(<condition1>)  
{  
    statement 1;  
    .  
    .  
}  
else if(<condition2>)  
{  
    statement 1;  
    .  
    .  
}  
.  
.  
else  
{  
    statement 1;  
    .  
}
```

```
if((a>=b)&& (a>=c))  
    System.out.println("largest=" + a);  
else if((b>=a)&& (b>=c))  
    System.out.println("largest=" + b);  
else  
    System.out.println("largest=" + c);
```

Switch...case construct

- In a switch...case construct the value of the controlling expression is compared against a set of constant values.
- Wherever, a match is found, the statements associated with that particular case are executed.
- If the value of the switch variable does not match any of the case constants, then the default case is executed.
- The choice of which branch to execute is determined by a controlling expression given in parentheses after the keyword switch.
- Following this are a number of occurrences of the reserved word case followed by a constant and a colon.
- These constants are called case labels.
- The controlling expression for a switch statement must be one of the types **char, int, short, byte, or String**.
- The String data type is allowed only in Java 7 or higher.
- The case labels must all be of the same type as the controlling expression.
- No case label can occur more than once, because that would be an ambiguous instruction.
- There may also be a section labeled default:, which is usually last.

Switch...case construct

| Syntax | Example |
|--|--|
| <pre>switch(<switch variable>) { case <constant1>: statement1; . . statement n; case <constant2>: statement1; . . statement n; . . case <constant n>: statement1; . . statement n; . . <u>default</u> : statement1; . . statement n; }</pre> <p><i>The underlined is optional.</i></p> | <pre>switch(color) { case 3: System.out.println("Red"); case 2: System.out.println("Orange"); case 7: System.out.println("Black"); }</pre> |

Switch...case construct

Example

```
switch (choice)
{
case 1: System.out.println("Sum=") ;
        System.out.println(a+b) ;
        break;
case 2: System.out.println("Diff=") ;
        System.out.println(a-b) ;
        break;
case 3: System.out.println("Product=") ;
        System.out.println(a*b) ;
        break;
case 4: System.out.println("Quot=") ;
        System.out.println(a/b) ;
        break;
case 5: System.out.println("Rem=") ;
        System.out.println(a%b) ;
        break;
default: System.out.println("wrong input") ;
}
```

Switch...case construct

- The switch statement ends when either a break statement is executed or the end of the switch statement is reached.
- A break statement consists of the keyword break followed by a semicolon.
- When the computer executes the statements after a case label, it continues until it reaches a break statement.
- When the computer encounters a break statement, the switch statement ends.
- If you omit the break statements, then after executing the code for one case, the computer will go on to execute the code for the next case.

Switch...case construct

- You can have two case labels for the same section of code, as in the following portion of a switch statement:
- **case 'A':**
case 'a': `System.out.println("Excellent. You need not take the final.");`
`break;`
- If no case label has a constant that matches the value of the controlling expression, then the statements following the default label are executed.
- You need not have a default section. If there is no default section and no match is found for the value of the controlling expression, then nothing happens when the switch statement is executed.
- However, it is safest to always have a default section. If you think your case labels list all possible outcomes, you can put an error message in the default section.

true and false Are Not Numbers

- Many programming languages traditionally use 1 and 0 for true and false.
- **In Java, the values true and false are not numbers, nor can they be type cast to any numeric type. Similarly, values of type int cannot be type cast to boolean values.**

Find the output

```
char letter = 'B';  
switch (letter)  
{  
    case 'A':  
    case 'a':  
        System.out.println("Some kind of A.");  
    case 'B':  
    case 'b':  
        System.out.println("Some kind of B.");  
        break;  
    default:  
        System.out.println("Something else.");  
        break;  
}
```

Find the output

```
int key = 1;
switch (key + 1)
{
    case 1:
        System.out.println("Apples");
        break;
    case 2:
        System.out.println("Oranges");
        break;
    case 3:
        System.out.println("Peaches");
    case 4:
        System.out.println("Plums");
        break;
    default:
        System.out.println("Fruitless");
}
```

Precedence and Associativity Rules

- Boolean expressions (and arithmetic expressions) need not be fully parenthesized.
- **If you omit parentheses, Java follows precedence rules and associativity rules in place of the missing parentheses.**
- The computer uses precedence rules to decide where to insert parentheses, but the precedence rules do not differentiate between two operators at the same precedence level, in which case the computer uses the associativity rules to “break the tie.”

Precedence and Associativity Rules

1. If one operator occurs higher on the list than another in the precedence table , the higher one is said to have higher precedence. **If one operator has higher precedence than another, the operator of higher precedence is grouped with its operands (its arguments) before the operator of lower precedence.**

Example:

balance * rate + bonus

Execution:

- * has a higher precedence than +, so computer first groups * and its operands, as follows:
 - (balance * rate) + bonus
 - Next, it groups + with its operands to obtain the fully parenthesized expression
 - ((balance * rate) + bonus)

Precedence and Associativity Rules

2. Sometimes two operators have the same precedence, in which case the parentheses are added using the associativity rules.

Example:

bonus + balance * rate / correctionFactor - penalty

The operators * and / have higher precedence than either + or -, so * and / are grouped first. But * and / have equal precedence, so the computer consults the associativity rule for * and /, which says they associate from left to right.

This means that the *, which is the leftmost of * and /, is grouped first. So the computer interprets the expression as

- **bonus + (balance * rate) / correctionFactor - penalty**
- which in turn is interpreted as

bonus + ((balance * rate) / correctionFactor) - penalty

- because / has higher precedence than either + or -.

Precedence and Associativity Rules

- This expression is still not fully parenthesized, however. The computer still must choose to group + first or - first.
- + and - have equal precedence. So the computer must use the associativity rules, which say that + and - are associated left to right. So, it interprets the expression as

$(\text{bonus} + ((\text{balance} * \text{rate}) / \text{correctionFactor})) - \text{penalty}$

- which in turn is interpreted as the following fully parenthesized

$((\text{bonus} + ((\text{balance} * \text{rate}) / \text{correctionFactor})) - \text{penalty})$

Precedence and Associativity Rules

- The assignment operators associate from right to left.

Example 1

`number1 = number2 = number3`

means

`number1 = (number2 = number3)`

which in turn is interpreted as the following fully parenthesized expression:

`(number1 = (number2 = number3))`

Precedence and Associativity Rules

Example 2

number1 = number2 = number3 + 7 * factor

- The operator of highest precedence is *, and the operator of next-highest precedence is +, so this expression is equivalent to

number1 = number2 = (number3 + (7 * factor))

- which leaves only the assignment operators to group. They associate right to left, so the fully parenthesized equivalent version of our expression is

(number1 = (number2 = (number3 + (7 * factor))))

Binding

- The association of operands with operators is called binding.
- For example, when parentheses determine which two expressions (two operands) are being added by a particular + sign, that is called binding the two operands to the + sign.
- A fully parenthesized expression accomplishes binding for all the operators in an expression.

Binding

- To evaluate an expression, Java uses the following three rules:
 - Java first does binding; that is, it first fully parenthesizes the expression using precedence and associativity rules, just as we have outlined.
 - Then it simply evaluates expressions left to right.
 - If an operator is waiting for its two (or one or three) operands to be evaluated, then that operator is evaluated as soon as its operands have been evaluated.

Binding

Example:

$(\text{number} + 1) > 2 \mid\mid (\text{number} + 5) < -3$

- This expression is equivalent to

$((\text{number} + 1) > 2) \mid\mid ((\text{number} + 5) < (-3))$

- because $>$ and $<$ have higher precedence than $\mid\mid$.
- So, in
 $((\text{number} + 1) > 2) \mid\mid ((\text{number} + 5) < (-3))$
- first the expressions $(\text{number} + 1)$, $(\text{number} + 5)$, and (-3) are evaluated (in any order),
- then the $>$ and $<$ are evaluated,
- and then the $\mid\mid$ is applied.

Check your understanding

Determine the value, true or false, of each of the following Boolean expressions, assuming that the value of the variable count is 0 and the value of the variable limit is 10. (Give your answer as one of the values true or false.)

1. `!(((count < 10) || (x < y)) && (count >= 0))`
2. `((limit/count) > 7) || (limit < 20)`

Check your understanding

2. Does the following sequence produce a division by zero?

```
int j = -1;  
if ((j > 0) && (1/(j+1) > 10))  
    System.out.println(i);
```

3. Convert the following expression to an equivalent fully parenthesized expression:

bonus + day * rate / correctionFactor * newGuy – penalty

Arithmetic Assignment Operators

Arithmetic assignment operator, is one, which combines arithmetic operator and an assignment operator.

The arithmetic assignment operators present in Java are:

`+=`

`-=`

`*=`

`/=`

`%=`

- **Example:** `a+=b; //` is same as `a=a+b;`

Increment/Decrement Unary Operators

- **Increment (++) Operator:** It is used to increment the value of variable by 1. It can be used in the following two ways:

| Pre Increment | Post Increment |
|--|--|
| In this, the operator precedes the variable. Example: ++a | In this, the operator follows the variable. Example: a++ |
| b=++a ; This statement is equivalent to the following two statements:- a=a+1 ; b=a ; So, in pre increment the value of the variable is incremented, before it is used in any other operation (Assignment in this case) | b=a++ ; This statement is equivalent to the following two statements:- b=a ; a=a+1 ; So, in post increment the value of the variable is incremented, after it is used in any other operation (Assignment in this case) |

Increment/Decrement Unary Operators

- **Decrement (--) Operator:** It is used to decrement the value of variable by 1. It can be used in the following two ways:

| Pre Decrement | Post Decrement |
|---|--|
| In this, the operator precedes the variable. Example: <code>--a</code> | In this, the operator follows the variable. Example: <code>a--</code> |
| <code>b= --a ;</code> This statement is equivalent to the following two statements:- <code>a=a-1 ;</code> <code>b=a ;</code> So, in pre decrement the value of the variable is decremented, before it is used in any other operation (Assignment in this case) | <code>b=a-- ;</code> This statement is equivalent to the following two statements:- <code>b=a ;</code> <code>a=a-1 ;</code> So, in post decrement the value of the variable is decremented, after it is used in any other operation (Assignment in this case) |


Loops

- A loop causes a program section to be executed a number of times depending on a specified condition. The repetition continues till a specified condition is true and terminates as soon as the specified condition turns false. At the termination of the loop, the control of the execution is passed on to the statement following the loop.
- There are three types of loops in Java
 - **for**
 - **while**
 - **do...while**

for Loop

- The *for* loop is used to execute a program section a fixed number of times. The condition is always checked first before the loop is executed.
- **Syntax:**

```
for(<initialization>; <condition>; <update expression>)  
{  
    statement 1;  
    statement 2;  
    statement 3;  
    .  
    .  
}
```



loop body

for Loop

- **Initialization:** It is used to initialize the loop variable and is executed only once at the start of the loop. As shown in the example **1 (a)** - loop variable **a** is initialized to **1**.
- **Condition/Boolean Expression:** It is evaluated every time before the execution of the statements in the loop body. The loop is executed if the condition evaluates to true, if evaluated to false the control of execution is passed to the statement immediately after the loop.
- **Update(Increment/Decrement) expression:** This statement is executed after each execution of the statements in the loop body and therefore makes the loop variable attain the final value, after which the condition turns false and the loop execution stops.

for Loop

Examples:

| 1 (a) | 1 (b) |
|---|---|
| <pre>for(a=1;a<=10;a++) System.out.print(a + " ");</pre> | <pre>for(a=10;a>=1;a--) System.out.print(a + " ");</pre> |
| Output | |
| 1 2 3 4 5 6 7 8 9 10 | 10 9 8 7 6 5 4 3 2 1 |

for Loop

| 2 (a) | 2 (b) |
|---|--|
| <pre>for(i=1,j=8,sum=0;i<=4;i++,j-=2) { System.out.println(i + " " + j); sum+=j; } System.out.println("sum is: " + sum);</pre> | <pre>for(i=1,j=2,k=5;i<=5;i++,j+=2,k+=5) System.out.println(i + ":" + j + ":" + k);</pre> |
| Output | |
| <pre>1 8 2 6 3 4 4 2 sum is: 20</pre> | <pre>1:2:5 2:4:10 3:6:15 4:8:20 5:10:25</pre> |

for Loop

| 3 (a) loop without any statements | 3 (b) loop without initialization and step value |
|--|--|
| <pre>for (a=1;a<=10;a++); System.out.println(a); //Outside the loop /* The value of a is incremented from 1 to 11 and loop is terminated as the condition turns false*/</pre> | <pre>int l=1,sum=0; for(; l<=5;) { sum+=l; l++; } System.out.println(l + "\t" + sum);</pre> |
| Output | |
| 11 | 6 15 |

while Loop

- The *while* loop executes a section of the program till a specified condition is true.
- *while* loop is generally used when the number of times the loop is to be executed is not known.
- The condition is always checked first before the loop is executed.

Syntax:

```
while(<condition>)
```

```
{
```

```
    statement 1;
```

```
    statement 2;
```

```
    .
```

```
    .
```

```
    .
```

```
    statement n;
```

```
}
```

} Loop Body

while Loop

- The *while* loop executes a section of the program till a specified condition is true.
- *while* loop is generally used when the number of times the loop is to be executed is not known.
- The condition is always checked first before the loop is executed.

Syntax:

```
while(<condition>)
```

```
{
```

```
    statement 1;
```

```
    statement 2;
```

```
    .
```

```
    .
```

```
    .
```

```
    statement n;
```

```
}
```

} Loop Body

while Loop

Examples:

1. **while-loop** with a single statement

```
int a=1;
while(a<=4)
    System.out.println(a++);
System.out.println("The final value of a:" + a);
```

Output

1
2
3
4

The final value of a: 5

2. The execution of the following loop depends on the value of **Ans**, therefore the number of times the loop is executed is not known.

```
int Ans;
System.out.println("Want to find the square
of a number (1/0)?");
Ans=input.nextInt();
int N;
while(Ans==1)
{
    System.out.println("Enter a no: ");
    N=input.nextInt();
    System.out.println("Square=" + N*N);
    System.out.println("Repeat(1/0):");
    Ans=input.nextInt();
}
```

do...while Loop

- The ***do..while*** loop executes a section of the program till the specified condition is true.
- ***do..while*** loop is generally used when the number of times the loop is to be executed is not known.
- The body of the ***do...while*** loop is executed at least once, since the condition is placed at the end of the loop.
- Moreover, **the statements written in this loop have to be always enclosed in a pair of curly braces, even if there is only a single statement in the loop. Do not forget the final semicolon at the end of the loop.**

do...while Loop

Syntax:

do
{

statement 1;
statement 2;
.
.
statement n;



Loop Body

} while(<condition>); //Final semicolon

do...while Loop

Examples:

| 1. | 2. |
|--|--|
| <pre>int a=1; do { System.out.println(a++); } while (a<=4);</pre> | <pre>int Ans; int N; do { System.out.println("Enter a no: "); N=input.nextInt(); System.out.println("Square=" + (N*N)); System.out.println("Repeat(1/0):"); Ans=input.nextInt(); } while (Ans==1);</pre> |
| Output | |
| 1 | |
| 2 | |
| 3 | |
| 4 | |

Entry and Exit controlled Loops

Entry Controlled Loops: These loops have a test expression/condition in the beginning of the loop, the test expression (condition) is therefore evaluated first, the loop is executed if the condition holds true. The control then moves back to test the condition for the next time and the process continues as long as the test expression evaluates to be true. If the condition is false right in the beginning the loop will not be executed even once. Examples: ***for*** and ***while*** loops.

Exit Controlled Loops: These loops have a test expression/condition at the end of the loop, thus having a control for exiting from the loop. The loop is executed at least once before the test expression is evaluated. If the condition holds true, the loop is executed again, otherwise, it exits the loop. Example ***do..while*** .

In Entry controlled loops ***for*** and ***while***, the body of the loop may not execute even once if the test expression evaluates to be false the first time, whereas in ***do..while***, the loop is executed at least once whether the condition holds true the first time or not.

Entry and Exit controlled Loops

Example:

1. In the following example after the initialization of **x** with value **1**, the control moves to test expression **(x>=10)**, the result evaluates to be false, and hence the body of the loop will not execute even for once.

```
for (x = 1; x>=10; x++)  
    System.out.println(x);
```

2. In the following example, first the body of the loop will execute with the value of **x** taken as **1**, and then only the test expression **(x>=10)** shall be evaluated to be false, so that the control does not go back for the second iteration.

```
int x = 1;  
do  
{  
    System.out.println(x++);  
}  
while (x >=10);
```


Nested Loops

When a ***for***, ***while*** or ***do..while*** loop is used inside another looping construct, the concept is called a nested loop. This concept is used whenever for each repetition of a process many repetitions of another process are required.

Some sample skeletal structures of
Nested Loops

| | | | |
|---|---|---|--|
| <pre>for (..) { for(..) <statement> }</pre> | <pre>while() { while(..) <statement> }</pre> | <pre>do { while(..) <statement> } while (..);</pre> | <pre>do { for (..) <statement> } while (..);</pre> |
|---|---|---|--|

Nested Loops

Example:

```
for (p = 1; p<=3; p++) //Outer Loop
{
    for (q = 10; q <=20; q+=10) //Inner Loop
        System.out.print(q + " ");
    System.out.println(); //Statement belongs to outer loop
}
```

Working

| p | q |
|---------------------|----------------|
| 1 | 10 |
| | 20 |
| | 30- loop stops |
| 2 | 10 |
| | 20 |
| | 30- loop stops |
| 3 | 10 |
| | 20 |
| | 30- loop stops |
| 4- outer loop stops | |

Output:

| | |
|----|----|
| 10 | 20 |
| 10 | 20 |
| 10 | 20 |

Check your understanding

1. What is the output of the following?

```
for (int count = 1; count < 5; count++)  
    System.out.print((2 * count) + " ");
```

2. What is the output of the following?

```
for (int n = 10; n > 0; n = n - 2)  
    System.out.println("Hello " + n);
```

3. What is the output of the following?

```
for (double sample = 2; sample > 0; sample = sample - 0.5)  
    System.out.print(sample + " ");
```