CS205 Object Oriented Programming in Java

Module 2 - **Core Java Fundamentals (Part 4)**

Prepared by Renetha J.B.

Topics

- Core Java Fundamentals**:**
- ✓ **Control Statements**
  - ✓ Selection Statements,
  - ✓ Iteration Statements
  - ✓ Jump Statements.

Control statements

- A programming language uses control statements to cause the **flow of execution** to **advance** and **branch** based on changes to the state of a program
- Categories of control statements
  - ✓ Selection Statements,
  - ✓ Iteration Statements
  - ✓ Jump Statements.

Control statements(contd.)

- **Selection statements** allow the program
  - to choose different paths of execution based on condition (outcome of an expression or the state of a variable).
- **Iteration statements** enable program execution
  - to *repeat one or more statements* (that is, iteration statements form loops).
- **Jump statements** allow your program
  - to execute in a nonlinear fashion.

Java's Selection Statements

- Also called **decision making statements.**
- Selection statements **control the flow of program's execution** based upon **conditions** *known only during run time.* It helps to choose different paths of execution based on condition.
- Java supports two selection statements:
  - ✓ **if**
  - ✓ **switch**

if statement

- **if** statement is Java's **conditional branch statement.**
  - It can be used to route program execution through different paths.

- Syntax of **simple if** statement

```
if (condition)
{
 // block of code to be executed if the condition is true
……….
}
```

Simple if E.g.

```
class Sample{
public static void main(String args[])
{
    int a=5;
    if(a>0)
    {
       System.out.println(" a is a positive number”);
    }
}
```

If-else statement

- General form of the **if statement:**

if (***condition***) *statement1;*

else *statement2;*

- Statement may be **a single statement** or a **compound statement enclosed in curly braces** (that is, a block).
- The ***condition*** is any expression that returns a **boolean value.**
- The **else** clause is optional.

Working of if-else

if (***condition***) *statement1;*

else *statement2;*

- If the condition is true, then *statement1* is executed.
- Otherwise, *statement2* (if it exists) is executed.
- Both statements will not be executed at the same time.

If-else E.g

```
class Sample{
public static void main(String args[]) {
int a=5, b=3;
if(a < b) a = 0;
else b = 0;
System.out.println(" a=" + a);
System.out.println(" b=" + b);
}
}
```

OUTPUT
a=5
b=0

If statement(contd.0

- If statement can be controlled using a boolean variable.
- E.g.

```
…
boolean dataAvailable;
// ...
if (dataAvailable)              //if dataAvailable is true
    ProcessData();              //call this function
else
    waitForMoreData();          //call this function
..
```

If statement(contd.)

- Only one statement can appear directly after the **if** or the **else.**

if(condition)

Statement1;

else

Statement

- If we want to include **more statements** inside **if** statement or **else** , we have to create a block (start with { and end with}

if(condition)

{

Statement1;

Statement1;

.....

}

Nested ifs

- A **nested if** is an if statement that is the inside (target of) another **if** or **else**.
- The **else statement** always refers to the
  - nearest if statement that is within the **same block** as the else and that is *not already associated with an else*.

Nested if E.g

```
if(i == 10)
    {
        if(j < 20) a = b;
                if(k > 100) c = d;      // this if is
                else a = c;             // associated with this else
}
else a = d; // this else refers to if(i == 10)
```

The if-else-if Ladder

- A common programming construct that is based upon a **sequence of nested ifs** is the **if-else-if ladder.**

if(*condition)*

statement;

else if(*condition)*

statement;

else if(*condition)*

statement;

...

else

statement;

The if-else-if Ladder(contd.)

- The if statements are executed from the **top down**.
- As soon as one of the conditions controlling the **if is true**, the statement associated with that if is executed, and the rest of the ladder is bypassed.
- If **NONE of the conditions is true**, then the final else statement will be executed.
- The last **else** acts as a default condition; that is, if all other conditional tests fail, then the last else statement is performed.

```
class IfElse {
public static void main(String args[]) {
int month = 4; // April
String season;
if(month == 12 || month == 1 || month == 2)
season = "Winter";
else if(month == 3 || month == 4 || month == 5)
season = "Spring";
else if(month == 6 || month == 7 || month == 8)
season = "Summer";
else if(month == 9 || month == 10 || month == 11)
season = "Autumn";
else
season = "Bogus Month";
System.out.println("April is in the " + season + ".");}}
```

```
class IfElse {
public static void main(String args[]) {
int month = 4; // April
String season;
if(month == 12 || month == 1 || month == 2)
season = "Winter";
else if(month == 3 || month == 4 || month == 5)
season = "Spring";
else if(month == 6 || month == 7 || month == 8)
season = "Summer";
else if(month == 9 || month == 10 || month == 11)
season = "Autumn";
else
season = "Bogus Month";
System.out.println("April is in the " + season + ".");}}
```

switch statement

- The switch statement is Java's **multiway branch** statement.
- It is an better alternative than a large series of **if-else-if** statements.

Syntax of switch


```
switch (expression)
{
    case value1:
       // statement sequence
       break;
    case value2:
       // statement sequence
    break;
    ...
    case valueN:
       // statement sequence
       break;
    default:
       // default statement sequence
}
```

Switch(contd.)

switch(***expression***){…..}

- The ***expression* inside switch** must be of type byte, short, int, or char;
  - each of the values specified in the case statements must be of a type compatible with the expression. (An enumeration value can also be used to control a switch statement)

Working of switch

- The value of the expression inside switch is compared with each of the literal values in the case statements.
  - If a match is found, the code sequence following that case statement is executed.
  - If none of the constants in the case matches the value of the expression, then the default statement is executed.
  - default statement is optional.
  - If no case matches and no default is present, then no further action is taken.

Working of switch(cond.)

- The break statement is used inside the switch to terminate a statement sequence.
- When a break statement is encountered, execution branches to the first line of code **after the entire switch** statement.
- This has the effect of "jumping out" of the switch.

```
class SampleSwitch {
public static void main(String args[]) {
for(int i=0; i<5; i++)
    switch(i)
    {
    case 0:
        System.out.println("i is zero.");
        break;
    case 1:
        System.out.println("i is one.");
        break;
    case 2:
        System.out.println("i is two.");
        break;
    default:
       System.out.println("i is greater than 2.");
    } } }
```

OUTPUT

```
i is zero.
i is one.
i is two.
i is greater than 2.
i is greater than 2.
```

```
class Switcheg {
public static void main(String args[]) {
for(int i=0; i<4; i++)
switch(i)
{
case 0:
    System.out.println("i is zero.");
case 1:
    System.out.println("i is one.");
case 2:
    System.out.println("i is two.");
    break;
default:
    System.out.println("i is greater than 2.");
} } }
```

OUTPUT

```
i is zero.
i is one.
i is two.
i is one.
i is two.
i is two.
i is greater than 2.
```

```
import java.util.Scanner;
class Switchvow {
public static void main(String args[]) {
Scanner sc=new Scanner(System.in);
System.out.println("Enter a letter:");
char c=sc.next().charAt(0); ;
switch(c)
{
case 'a':
case 'e':
case 'i':
case 'o':
case 'u':
case 'A':
case 'E':
case 'I':
case 'O':
case 'U':
System.out.println("vowel");
break;
default:  System.out.println("Not vowel-may be consonent");  } } }
```

OUTPUT
Enter a letter:
A
vowel

Nested switch Statements

- We can use a switch as part of the statement **sequence of an outer switch**. This is called a *nested switch*

```
switch(count) {
    case 1:
        switch(target) { // nested switch
                    case 0:
                            System.out.println("target is zero");
                            break;
                    case 1:  // no conflicts with outer switch
                            System.out.println("target is one");
                            break;
                    }
        break;
    case 2: // ...   }
```

Features of the switch statement

- The switch differs from the if in that **switch can only test for equality**, whereas if can evaluate any type of Boolean expression.
  - switch looks only for a match between the value of the expression inside **switch** and one of its **case** constants.
- No two case constants in the same switch can have identical values.
  - But a switch statement and an enclosing outer switch can have case constants in common.
- A switch statement is usually **more efficient** than a set of nested ifs.

Swich(features)

- When **Java compiler** compiles a switch statement, it will inspect each of the case constants and create a "jump table" that it will use for selecting the path of execution depending on the value of the expression.
- So a switch statement will run much faster than the equivalent logic coded using a sequence of if-elses.
- The compiler can do this because it knows that the case constants are all the same type and simply must be **compared for equality** with the switch expression.
- The compiler has no such knowledge of a long list of if expressions

Iteration Statements

- A iteration statements or loop repeatedly executes the same set of instructions until a termination condition is met.

- Java's iteration statements (**looping** statements) are
  - ✓ **for**
  - ✓ **while**
  - ✓ **do-while**

while

- The while loop is Java's most fundamental loop statement. It is **ENTRY CONTROLLED** loop.
  - The statements inside the body of **while** is executed only if the *condition inside while is **true***.
- It repeats a statement or block while its controlling expression is true.
- General form:

```
while(condition)
{
// body of loop
}
```

Working of while

```
while(condition)
{
// body of loop
}
```

- The **condition** can be any Boolean expression.
  - The body of the loop will be executed as long as the conditional expression is **true**.
  - When condition becomes **false**, control passes to the next line of code immediately after the loop.

While(contd.)

- The curly braces are not needed if only a single statement is being repeated.

```
while(condition)
    Statement;
```


```
// Demonstrate the while loop.
class Whileeg {
public static void main(String args[]) {
int n = 10;
while(n > 0) {
System.out.println("tick " + n);
n--;
}
}
}
```

OUTPUT

```
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1
```

While(contd.)

- The body of the while (or any other of Java's loops) can be empty.
  - This is because a **null statement** (one that consists only of a **semicolon**) is syntactically valid in Java.

while(condition) ;

Here if condition is true no statement is executed as part of while

```
class Whileeg {
public static void main(String args[])
{
    int n = 10;
    while(n > 0)
     {
    System.out.println("tick " + n);
    }
}
}
```

OUTPUT

```
tick 10
tick 10
tick 10
.....

.


.
```

INFINITE LOOP

do-while

- The **do-while loop always executes its body at least** once, because its conditional expression is at the bottom of the loop.

```
do
{
   //statements
}
while(condition);
```

Working of do-while

do-while is EXIT CONTROLLED loop.

```
do
{
   //statements
}
while(condition);
```

1. Initially the **statements** inside the do-while loop is executed
2. then only the **condition** inside while is checked.
3. Then the loop is executed only if that condition is true.
   - That is condition is checked only during exit from do-while loop.

```
class Menu {
public static void main(String args[]) throws java.io.IOException
    {
…
char choice = (char) System.in.read();
…
}
```

Here **System.in.read( )** is used here to obtain the value of choice from user. So main() function throws java.io.IOException

```
class Whileeg {
public static void main(String args[])
{
    do
     {
    System.out.println("Hello");
    }while(true);
}
}
```

OUTPUT

Hello
Hello
Hello
Hello
....
...
..

.

INFINITE LOOP

Difference between while and do-while

BASIS FOR COMPARISON	WHILE	DO-WHILE

General Form	`while ( condition) {`
	`statements; //body of loop`
	`}`
		`do{`
		`.`
		`statements; // body of loop.`
		`.`
		`} while( Condition );`

Controlling Condition	In 'while' loop the controlling condition appears at the start of the loop.	In 'do-while' loop the controlling condition appears at the end of the loop.
Iterations	The iterations do not occur if, the condition at the first iteration, appears false.	The iteration occurs at least once even if the condition is false at the first iteration.

for

It is an iteration statement(looping)

for(***initialization***; *condition;* ***iteration***) *{*

// body

}

Working of for loop

- When the loop first starts, the **initialization** portion of the loop is executed. It acts as a loop control variable (counter).
  - the initialization expression is only executed once.
- Next, condition is evaluated.(Boolean expression)
  - It usually tests the loop control variable against a target value.
  - If this expression is **true**, then the body of the loop is executed.
  - If it is **false**, the loop terminates.
- Next, the iteration portion of the loop is executed.
  - increments or decrements the loop control variable.
- Next, condition is evaluated.
- And the process continues until condition becomes **false**

```
for( ; ; )
{
// ...
}
```

INFINITE LOOP

The For-Each Version of the for Loop

for(*type var :* collection) statement-block;

- Here, *type* specifies the type and *var* specifies the name of an iteration variable that will receive the elements from a collection, one at a time, from beginning to end.
- The collection being cycled through is specified by collection

```
class Foreacheg {
public static void main(String args[])
{
 int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
for(int x: nums)
    System.out.println(x);
}
}
```

```
OUTPUT
1
2
3
4
5
6
7
8
9
10
```

- During each pass through the loop, x is automatically given a value equal to the next element in nums.
  - Thus, on the first iteration, x contains 1;
  - on the second iteration, x contains 2; and so on.
- Not only is the syntax streamlined, but it also prevents boundary errors.

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int x: nums)
    System.out.println(x);
```

```
1
2
3
4
5
6
7
8
9
10
```

- With each pass through the loop, x is automatically given a value equal to the next element in nums.
  - Thus, on the first iteration, x contains 1;
  - on the second iteration, x contains 2; and so on.
- Not only is the syntax streamlined, but it also prevents boundary errors.

Nested loops


```
// Loops may be nested.
class Foreg2{
public static void main(String args[]) {
int i, j;
for(i=0; i<4; i++) {
for(j=0; j<3; j++)
System.out.print("i="+i+" j="+j + "\t\t");
System.out.println();
}
}
}
```

OUTPUT

i=0 j=0	i=0 j=1	i=0 j=2
i=1 j=0	i=1 j=1	i=1 j=2
i=2 j=0	i=2 j=1	i=2 j=2
i=3 j=0	i=3 j=1	i=3 j=2

```
// Loops may be nested.
class Foreg2{
public static void main(String args[]) {
int i, j;
for(i=0; i<4; i++) {
for(j=0; j<3; j++)
System.out.print(i +"\t\t");
System.out.println();
}
}
}
```

OUTPUT

0	0	0
1	1	1
2	2	2
3	3	3

```
// Loops may be nested.
class Foreg2{
public static void main(String args[]) {
int i, j;
for(i=0; i<4; i++) {
for(j=0; j<3; j++)
System.out.print(j+"\t\t");
System.out.println();
}
}
}
```

OUTPUT

```
0       1       2
0       1       2
0       1       2
0       1       2
```

Nested loops

```
// Loops may be nested.
class Nested {
public static void main(String args[]) {
int i, j;
for(i=0; i<4; i++) {
for(j=0; j<2; j++)
System.out.print(“*");
System.out.println();
}
}
}
```

```
**
**
**
**
.
```

Nested loops

```
// Loops may be nested.
class Nested {
public static void main(String args[]) {
int i, j;
for(i=0; i<4; i++) {
for(j=i; j<4; j++)
System.out.print(“*");
System.out.println();
}
}
}
```

```
****
***
**
*


.
```

Jump Statements

- Java supports three jump statements:
  - ✓ **break**
  - ✓ **continue**
  - ✓ **return**

break statement

- Three uses.

✓ First it **terminates** a statement sequence in a switch statement.

✓ Second, it can be used to **exit** a loop.

✓ Third, it can be used as a "civilized" form of goto.

break E.g

```
// Using break to exit a loop.
class BreakLoop {
public static void main(String args[]) {
for(int i=0; i<6; i++)
{
if(i == 3)
    break; // terminate loop if i is 3
System.out.println("i: " + i);
}
System.out.println("Loop complete.");
}
}
```

OUTPUT

```
i: 0
i: 1
i: 2
Loop complete.
```

Using break as a Form of Goto

- By using this form of **break, you can, for example, break out of** one or more blocks of code.
- The general form of the labeled break statement is :

  break *label;*

```
// Using break as a civilized form of goto.
class Breakeg {
public static void main(String args[]) {
boolean t = true;
first: {
        second: {
                third: {
                        System.out.println("Before the break.");
                        if(t) break second; //break of second block
                        System.out.println("This won't execute");
                        }
                System.out.println("This won't execute");
                }
    System.out.println("After second block.");
    }
}
}
```

OUTPUT

Before the break.
After second block..

continue statement

- In **while** and **do-while** loops, a **continue** statement causes **control to be transferred directly to the conditional expression** that controls the loop.
- In a **for** loop, **control goes first to the iteration portion of the for statement and then to the conditional expression**.
- For all three loops, **any intermediate code after continue is bypassed(skipped).**

continue E.g

```
// Using break to exit a loop.
class continueeg{
public static void main(String args[]) {
for(int i=0; i<6; i++)
{
if(i == 3)
    continue; // skip remaining stmts if i is 3
        // continue loop.control goes to iteration
System.out.println("i: " + i);
}
System.out.println("Loop complete.");
}
}
```

OUTPUT

```
i: 0
i: 1
i: 2
i: 4
i: 5
Loop complete.
```

```
// Demonstrate continue.
class Continueeg {
public static void main(String args[]) {
for(int i=0; i<10; i++) {
System.out.print(i + " ");
if (i%2 == 0) continue;
System.out.println("");
}
}
}
```

OUTPUT

```
0 1
2 3
4 5
6 7
8 9
```

return statement

- The **return** statement is used to explicitly return from a method.
  - The **return** causes program control to transfer back to the caller of the method.
- When **return** statement is executed the method terminates.
- The r**eturn** causes execution to return to the Java run-time system
- Methods that have a return type other than void **return a value** to the calling method(function)

return *value;*

- Here, *value is the value is returned to the calling function*

```
// Demonstrate return.
class Return {
public static void main(String args[]) {
boolean t = true;
System.out.println("Before the return.");
if(t) return;
System.out.println("This won't execute.");
}
}
```

OUTPUT
Before the return

Reference

- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.