# Programming in Java

## Lecture 4: Control Structures

# SELECTION STATEMENTS

- Java supports two selection statements: **if** and **switch.**

## if statement

```
if (condition) statement1;
else statement2;
```

- Each statement may be a single statement or a compound statement enclosed in curly braces (block).
- The condition is any expression that returns a **boolean value.**
- **The else** clause is optional.
- If the condition is true, then statement1 is executed. Otherwise, statement2 (if it exists) is executed.
- In no case will both statements be executed.

## Nested ifs

- A nested if is an if statement that is the target of another if or else.
- In nested ifs an 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.

```
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 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 statements are executed from the top to down.

## switch

- The switch statement is Java's multi-way branch statement.
- provides an easy way to dispatch execution to different parts of your code based on the value of an expression.
- provides a better alternative than a large series of **if-else-if statements.**

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

## Nested switch Statements

- You 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: // ...
```

## ITERATION STATEMENTS (LOOPS)

# Iteration Statements

- In Java, iteration statements (loops) are:
  - for
  - while, and
  - do-while

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

## While Loop

- While loop repeats a statement or block while its controlling expression is true.
- 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 following the loop.

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

```
class While
    {
        public static void main(String args[]) {
        int n = 10;
        char a = 'G';
        while(n > 0)
                 {
                  System.out.print(a);
                  n--;
                  a++;
                 }
          }
    }
```

- The body of the loop will not execute even once if the condition is false.

- 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.

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

- Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression.
- If this expression is true, the loop will repeat. Otherwise, the loop terminates.

## for Loop

```
for (initialization; condition; iteration)
    {
      // body
    }
```

- Initialization portion sets the value of loop control variable.
- Initialization expression is only executed once.
- Condition must be a Boolean expression. It usually tests the loop control variable against a target value.
- Iteration is an expression that increments or decrements the loop control variable.

The for loop operates as follows.

- When the loop first starts, the initialization portion of the loop is executed.

- Next, condition is evaluated. 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.

```
class ForTable
        {
                public static void main(String args[])
                 {
                        int n;
                        int x=5;
                        for(n=1; n<=10; n++)
                          {
                          int p = x*n;
                          System.out.println(x+"*"+n +"="+ p);
                          }
                }
        }
```

## Declaring loop control variable inside loop

- We can declare the variable inside the initialization portion of the for.

```
for ( int i=0; i<10; i++)
        {
            System.out.println(i);
        }
```

- Note: The scope of this variable i is limited to the for loop and ends with the for statement.

## Using multiple variables in a for loop

- More than one statement in the initialization and iteration portions of the for loop can be used.

Example 1:

```
class var2 {
        public static void main(String arr[]) {
                int a, b;
                b = 5;
                for(a=0; a<b; a++) {
                        System.out.println("a = " + a);
                        System.out.println("b = " + b);
                        b--;
                }
        }
}
```

- Comma (separator) is used while initializing multiple loop control variables.

Example 2:

```
class var21
        {
                public static void main(String arr[]) {
                        int x, y;
                        for(x=0, y=5; x<=y; x++, y--) {
                                System.out.println("x=  " +   x);
                                System.out.println(“y =  " +  y);
                        }
                }
        }
```

- Initialization and iteration can be moved out from for loop.

# For-Each Version of the for Loop

- Beginning with JDK 5, a second form of for was defined that implements a “for-each” style loop.

- For-each is also referred to as the **enhanced for loop.**

- Designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish.

## for (type itr-var : collection) statement-block

- *type* specifies the type.
- *itr-var* specifies the name of an iteration variable that will receive the elements from a collection, one at a time, from beginning to end.
- With each iteration of the loop, the next element in the collection is retrieved and stored in itr-var.
- The loop repeats until all elements in the collection have been obtained.

```
class ForEach
    {
      public static void main(String arr[])
            {
                  int num[] = { 1, 2, 3, 4, 5 };
                  int sum = 0;
                  for(int i : num)
                     {
                        System.out.println("Value is: " + i);
                        sum += i;
                     }
                  System.out.println("Sum is:  " + sum);
            }
    }
```

# Iterating Over Multidimensional Arrays

```
class ForEachMArray {
        public static void main(String args[]) {
                int sum = 0, c=1;
                int num[][] = new int[3][5];
        // give num some values
                for(int i = 0; i < 3; i++)
                        for(int j=0; j < 5; j++)
                                num[i][j] = c++;
        // use for-each for to display and sum the values
                for(int x[] : num) {
                        for(int y : x) {
                                System.out.println("Value is: " + y);
                                sum += y;
                                  }
                               }
                System.out.println("Summation: " + sum);
                        }
                }
```

## Nested Loops

```
class NestedLoop
   {
         public static void main(String arr[])
                 {
                         int i, j;
                         for(i=0; i<10; i++)
                                 {
                                         for(j=i; j<10; j++)
                                             System.out.print(“* ”);
                                 System.out.println(  );
                                 }
                 }
   }
```

# Lets Do It

- WAP to display following types pattern:

(a)

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

(b)

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

# Lets Do It

- WAP to display following types pattern:
- C)
```
   *
  ***
 *****
*******
```
- D)
```
          *
        + * +
      * * * * *
    + * * * * * +
  * * * * * * * * *
  + + + + + + + +
```

## Jump Statements

- Java supports three jump statements:
  - break
  - continue
  - return
- These statements transfer control to another part of the program.

- Break: break statement has three uses.
  - terminates a statement sequence in a switch statement
  - used to exit a loop
  - used as a “civilized” form of goto

Note: Java does not have goto statement.

Example 1:

```
class BreakLoop
    {
        public static void main(String arr[])
                {
                        for(int i=0; i<100; i++)
                                {
                                  if(i == 10) break;
    // terminate loop if i is 10
                                  System.out.println("i: " + i);
                                }
                        System.out.println("Loop complete.");
                }
     }
```

- Java defines an expanded form of the break statement.
- By using this form of break, we can break out of one or more blocks of code.
- These blocks need not be part of a loop or a switch. They can be  any block.
- Further, we can specify precisely where execution will resume, because this form of **break works with a label.**

break *label;*

- When this form of break executes, control is transferred out of the named block. The labeled block must enclose the break statement.

```
class BreakLoop
   {
         public static void main(String arr[])
                 {
                 outer: for(int i=0; i<3; i++)
                         {
                          System.out.print("Pass " + i + ": ");
                          for(int j=0; j<100; j++)
                                  {
                                  if(j == 10) break outer; // exit both loops
                                          System.out.print(j + " ");
                                  }
                         System.out.println("This will not print");
                         }
                 System.out.println("Loops complete.");
                 }
    }
```

# Continue

- 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 is bypassed.

- Example:

```
class Continue {
public static void main(String args[]){
        for(int i=0; i<5; i++) {
                System.out.println(i + " ");
                if (i%2 == 0) continue;
                System.out.println("No Continue");
               }
        }
}
```

- Similar to break statement, Continue may specify a label to describe which enclosing loop to continue.

Example:

```
class ContinueLabel {
    public static void main(String args[]) {
            outer: for (int i=0; i<10; i++) {
                        for(int j=0; j<10; j++) {
                            if(j > i) {
                                System.out.println();
                                continue outer;
                                }
                            System.out.print(" " + (i * j));
                            }
                        }
                System.out.println();
                }
            }
```

## return

- The return statement is used to explicitly return from a method.
- It causes program control to transfer back to the caller of the method.

Example:

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

# Let’s Do It

- Write a Java program that prints the numbers from 1 to 50. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print “FizzBuzz”.