### Iterative (Looping) Control Structures

One of a computer’s most important attributes is the ability to perform repeatedly certain actions quickly, and loops are at the very heart of that. A loop is a repetition of a certain code segment/s in the program while the condition being evaluated remains to be true.

Loops usually have the following components:

1. ***Initialization***of a variable or of several variables
2. ***Condition***(that would evaluate to either true or false); the condition check is usually made on the current value of the variable initialized in (1) above.
3. ***Body of the loop***which maybe a single statement or a group of statements
4. A ***change of state***which is usually a statement inside the body of the loop that changes the contents of the variable(s)

Example: Printing the word “HELLO” fifty times.

Initialization: counter = 1

Condition: While counter <= 50 Body of the loop: Print “HELLO” Change of state: counter = counter + 1

There are three available loop control structures in Java. These are:

1. while Loop
2. for Loop
3. do-while Loop

### while Loop

The syntax of for the while loop is as follows:

<initialization> while(<condition>)

{

<statement1>;

. . .

<statementN>;

<change of state>;

}

The body of the loop or statements under the while-loop will be repeatedly executed while the condition is TRUE. Otherwise, the loop will terminate and the next statement after the end of the while loop will be executed next.

Example: Considering the example given above, we have to write a program that will print the word “HELLO” fifty times.

//Printing HELLO 50 times public class PrintHello{

public static void main(String[] args){ int counter;

counter = 1; // Initialization

while(counter <= 50){ // Condition

System.out.println(“HELLO”); // Body counter = counter + 1; // Change of State

}

}

}

The program above can be modified easily to print the word “HELLO” any number of times. It is simply a matter of changing the value inside the condition.

Exercises: Evaluating the above program answer the following questions:

* + 1. What will happen if we remove the initialization counter = 1 in the program? This means that the value of **counter** is undefined or garbage.
    2. What will happen if we remove the curly brackets enclosing the body of the loop?
    3. What will happen if the programmer committed a typographical error, such that instead of pressing the less than symbol, the greater than symbol was pressed, i.e. the condition becomes counter >= 50?
    4. What will happen if the programmer forgot changing the value of

counter, i.e., counter = counter + 1 was omitted.

* + 1. Write a new program using a while loop where it will print the numbers 10 down to 1 in vertical way.

Note that the change of state need not always be an increment or decrement by 1. It can be of any value (also known as the step value) that is appropriate in the problem. Consider the next example:

Example: Program that will compute the sum of all numbers that are divisible by 5 from 0 to 100.

// by 5’s program using while loop

public class ByFive{

public static void main(String[] args){ int num, sum;

sum = 0;

num = 0;

while(num <= 100){ sum = sum + num; num = num + 5;

}

System.out.println(“sum = ” + sum);

}

}

Note that the change of state in the above program is an increment of 5. Once the value of num reaches 105, the body of the loop will terminate and the statement below it will be executed next, which is System.out.println(“sum = “ sum);.

Variable used for the change of state are not limited to integers only and need not always be incremental. Consider the following example:

// Step down using while loop public class StepDown{

public static void main(String[] args){ double n = 10;

while(n >= 1){

System.out.println(n); n = n – 0.5;

}

}

}

Try simulating the output of the step down program written above.

### for Loop

The syntax for a for-loop is as follows:

for([initialization]; [condition]; [change of state]){

<statement1>;

. . .

<statementN>;

}

The for-loop is actually a “more compact” form of the while loop. The loop is executed as follows:

* + 1. Perform the initialization
    2. Check the condition
    3. If the condition results to a TRUE value, the statement/s inside the for- loop (body) will be executed. Else proceed to #6
    4. Change of state
    5. Goes back to #2.
    6. Exit the loop and the statement after the end of the for-loop will be executed.

Example: Print the word “HELLO” fifty times using for-loop:

//HELLO 50 times using for-loop example

public class PrintHello{

public static void main(String[] args){ int counter;

for(counter = 1; counter <= 50; counter++){

System.out.println(“HELLO”); // Body

}

}

}

Note that the initialization, condition, and the change of state are only found in one line. The body of the for-loop like the while-loop is enclosed in a pair of curly braces. If there is only one statement representing the body of the loop, then the pair of curly brackets can be optional.

counter++ is also the same as counter = counter + 1

Example: Print all EVEN numbers from 100 down to 0

// Even from 100 down to 0 using for-loop public class EvenNumbers{

public static void main(String[] args){

int i;

for(i = 100; i >= 0; i = i - 2) System.out.println(i);

}

}

Example: A program that display the Fibonacci series.

// Fibonacci series using for-loops import java.io.\*;

public class FibonacciNumbers{

public static void main(String[] args){ int prev = 0, next = 1, ans = 0;

int i, num = 0; String input = “ “;

BufferedReader in = new BufferedReader(new

InputStreamReader(System.in));

System.out.print(“Input a number: “);

try{

input = in.readLine();

}catch(IOException e){

System.out.println(“Error!”);

}

num = Integer.parseInt(input);

System.out.println(“Fibonacci series is: ” + num);

for(i = num; i >=0; i--){ System.out.print(ans); prev = next;

next = ans;

ans = prev + next;

}

}

}

Sample Output: The Fibonacci of 10

Input a number: 10

The Fibonacci series of 10 is:

0 1 1 2 3 5 8 13 21 34 55 \_

### do-while Loop

The syntax for a Do-While-loop is as follows:

<initialization>;

do{

<statement1>;

. . .

<statementN>;

<change of state>;

}while(<condition>);

The Do-While is executed as follows:

* + 1. Performs the initialization.
    2. Performs the body of the loop
    3. Change of state
    4. Checks the condition. If the condition results to a TRUE value, the statement/s inside the do-while Loop (body) will be executed until condition will be FALSE.
    5. If condition is FALSE, exit the loop and the statement after the end of the do-while Loop will be executed.

Example: Print the word “HELLO” fifty times using do-while-Loop:

//HELLO x 50 using do-while-loop example public class PrintHello{

public static void main(String[] args){ int counter;

counter = 1; // Initialization do{

System.out.println(“HELLO”); // Body counter = counter + 1; // Change of State

}while(counter <= 50); // Condition

}

}

Note that in a do-while Loop, the body is executed at least once before the condition is checked. Unlike in the while and for loops, the condition is first checked before the body of the loop can be executed.

Example: Print all EVEN numbers from 100 down to 0

// Even from 100 down to 0 using do-while-loop

public class EvenNumbers{

public static void main(String[] args){ int i;

i = 100;

do{

System.out.println(i); i = i – 2;

}while(i >= 0);

}

}

### Other implementation of loops

Loops are not confined on processing numeric data, the program will depend on the problem that is being solved. Let us consider the problem below.

// Interactive program using loops import java.io.\*;

public class InteractiveProgram{

public static void main(String[] args){ int a, b = 0, sum;

char again;

String input = “ “;

BufferedReader in = new BufferedReader(new

InputStreamReader(System.in));

do{

sum = 0;

for(a = 0; a < 10; a++){

System.out.println(“Input number “ + a + “ of 9: ”);

try{

input = in.readLine();

}catch(IOException e){

System.out.println(“Error!”);

}

b = Integer.parseInt(input); sum = sum + b;

}

System.out.println(“The sum = ” + sum); System.out.println(“Do you want to try again? (Y/N)”); try{

input = in.readLine();

}catch(IOException e){} again = input.charAt(0);

}while(again == ‘Y’ || again == ‘y’); System.out.println(“Have a nice day!”);

}

}

### Counters

A counter is a variable that is used to keep track of the count (frequency) of a certain group of items. Usually,

* Its data is integer
* It is initialized to a value of 0
* Incremented by 1 inside a loop

Example: Write a program that will ask the user to input 10 integers. The program should output how many of the data are positive. Assume that zero is a positive integer.

// Counter program import java.io.\*;

public class CounterProgram

public static void main(String[] args){ int i=0, num, ctr\_positive;

String input = “ “;

ctr\_positive = 0;

BufferedReader in = new BufferedReader(new

InputStreamReader(System.in));

for(i = 0; i < 10; i++){

System.out.println(“Input number “ + i + “ of 9:”, i); try{

input = in.readLine();

}catch(IOException e){

System.out.println(“Error !”);

}

num = Integer.parseInt(input); if(num >= 0){

ctr\_positive = ctr\_positive + 1;

}

}

System.out.println(“Positive integers= ” + ctr\_positive);

}

}

Exercise:

* + 1. Modify the previous program such that it will also count and print the number of negative integers that were inputted by the user.
    2. Create a new program that will ask the user to input **n** integer values then print the number of ODD and EVEN numbers.
    3. Create a program the will display all numbers that are divisible by 5 and count how many of them are there in the range of a signed integer data type.

### Accumulators

An **accumulator** is a variable that is used to keep track of the accumulated value of a certain group of items. An accumulator

* May have a data type of int, float or double
* It is usually initialized to a value of 0
* Changes by assuming the sum of the current value of the accumulator and the value of another variable

Example: Write a program that will ask the user to input **n** integers. The program should output the sum of all the input data.

// Accumulator Sample import java.io.\*;

public class AccumulatorSample{

public static void main(String[] args){ int n, i, num, sum;

String input = “ “;

sum = 0;

BufferedReader in = new BufferedReader(new

InputStreamReader(System.in)); System.out.println(“How many numbers to process? ”); try{

input = in.readLine();

}catch(IOException e){

System.out.println(“Error!”);

}

n = Integer.parseInt(input); for(i = 1; i <= n; i++)

{

System.out.println(“Input number ” + i + “ of ” + n);

try{

input = in.readLine();

}catch(IOException e){

System.out.println(“Error!”);

}

num = Integer.parseInt(input); sum = sum + num;

}

System.out.println(“Sum of all input data is = ” + sum);

}

}

Exercises:

* + 1. Modify the previous program such that it will compute and output the sum of positive numbers and the sum of negative numbers.
    2. Create a new program that will ask the user to input how many students are there in a class. Thereafter, the user will be asked to input the final grade for each student. The program should determine the average of all the grades of the student in that class. How many of them passed, and how many of them failed.

Assume that the passing grades are from 75 to 99, and failing grades are from 70 to 74.

# UNIT 3

## Arrays and Introduction to Methods

### Java Arrays

An array stores multiple data items of the same type, in a contiguous block of memory, divided into a number of slots. Furthermore, **array** is a capability of programming languages wherein one variable can store a list of data and manipulate these data more efficiently.

In simple words:

* An array is a group of elements with the same data type (homogeneous).
* An array is characterized by its name, dimension, size, and element data type.
* An array has a dimension:
  + If dimension is 1, we say that the array is a one-dimensional; Also known as a list.
  + If dimension is 2, we say that the array is a two-dimensional; Also known as a table.
  + Array having more than one dimension is also known as multi- dimensional array.

In our course, we will just discuss one and two dimensional arrays. But in the Java language, you can have more than two dimensions for an array.

### Declaring a One-Dimensional Array

The syntax for a one-dimensional array:

<data\_type> <array\_name>[];

or

<data\_type>[] <array\_name>;

After declaring, you must create the array and specify its length using a **constructor.** The process of using a **constructor** to create an array is called **instantiation**.

Sample Declaration:

//declaration int number[];

//instantiation number = new int[10];

It can also be written as:

//declare and instantiate int number[] = new int[10];

The declaration tells the Java Compiler that the identifier number will be used as the name of the array containing integers and creates or instantiate a new array containing 10 elements.

Examples:

char name[] = new char[35]; int grades[] = new int[50]; float area[] = new float[5]; double data[] = new double[10];

Note: array elements not explicitly initialized will have garbage values. To reference an element in a one-dimensional array:

<array\_name>[<index>]

Key Points when referencing elements in an array in Java:

* The range of values that you can use as array index is from 0 to

<size-1>

* The first element is always at index 0
* The last element is always at index <size -1>
* Index cannot be negative
* Index cannot be a float or double (real) number, only integer values Example: A list of 10 scores in a quiz:

int scores[] = new int[10];

* + The range of index values is from 0 to 9
  + The first element is at scores[0]
  + The last element is at scores[9]

|  |  |  |
| --- | --- | --- |
| scores[0] | = | 7 |
| scores[1] | = | 3 |
| scores[2] | = | 8 |
| scores[3] | = | 4 |
| scores[4] | = | 6 |
| scores[5] | = | 3 |
| scores[6] | = | 6 |
| scores[7] | = | 9 |
| scores[8] | = | 7 |
| scores[9] | = | 5 |

index 

scores:

Graphical representation of scores array having 10 as its size:

0 1 2 3 4 5 6 7 8 9

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **7** | **3** | **8** | **4** | **6** | **3** | **6** | **9** | **7** | **5** |

values

You can also initialize values of an array during declaration. Examples are:

/\*creates an array of boolean variable with identifier values initialized to True and False \*/

boolean values[] = {true, false};

/\*creates an array of 4 double values (100, 90, 80, 75) with identifier name grades\*/

double[] grades = {100, 90, 80, 75};

//creates an array of Strings with identifier days String days[] = {“Mon”, ”Tues”, ”Wed”, ”Thurs”, ”Fri”,

“Sat”, ”Sun”};

Example: Create a program that will declare a one-dimensional array with a size of 10. Initialize the array and let the user input 10 quiz scores as given above, thereafter compute the average of these 10 scores. Assume that the perfect score per quiz is 10 points.

// 10 quiz scores using 1-D array import java.io.\*;

public class TenQuizScores{

public static void main(String[] args){ int scores[] = new int[10];

int i, sum = 0, num = 0; double ave = 0;

String input = “ “;

BufferedReader in = new BufferedReader(new

InputStreamReader(System.in));

// Initialize list to zero for(i = 0; i < 10; i++){

scores[i] = 0;

}

// Input 10 scores

for(i = 0; i < 10; i++){ System.out.print(“Input score “ + i + “ = ”); try{

input = in.readLine();

}catch(IOException e){

System.out.println(“Error!”);

}

num = Integer.parseInt(input); scores[i] = num;

}

// Print inputted scores for(i = 0; i < 10; i++){

System.out.println(“score “ + i + “ = ”+ scores[i]);

}

// Compute sum of all 10 scores for(i = 0; i < 10; i++){

sum = sum + scores[i];

}

// Compute the average

ave = (double)sum / 10; // type casts (double)sum

System.out.println(“Sum of all scores is = “ + sum

+ “ over 100”); System.out.println(“The average is = ” + ave);

}

}

Note: ave = (double)sum/10. If we remove the cast (double) before the variable sum, the result will be an integer division, since sum is an integer and 10 is also an integer. The fraction part will be dropped. In order to retain the fraction part for the ave variable, we need to preface the variable with a new type in parenthesis. This is called type casting in Java.

### Declaring Two-Dimensional Array

The syntax in declaring a two-dimensional array is as follows:

<data\_type> <array\_name>[<rowsize>][<colsize>]; Or

<data\_type>[<rowsize>][<colsize>] <array\_name>;

The **instantiation** of a two-dimensional array is the same as one- dimensional array except that the size would contain two value, size for the row and size for the column.

Sample Declaration:

//declaration int number[][];

//instantiation

number = new int[5][10];

It can also be written as:

//declare and instantiate

int number[][] = new int[5][10];

Example:

int table[][] = new int[3][4];

Graphical representation of table[][] of int[3][4]:

**Col 0 Col 1 Col 2 Col 3**

**Row 0**

|  |  |  |  |
| --- | --- | --- | --- |
| **[0][0]** | **[0][1]** | **[0][2]** | **[0][3]** |
| **[1][0]** | **[1][1]** | **[1][2]** | **[1][3]** |
| **[2][0]** | **[2][1]** | **[2][2]** | **[2][3]** |

**Row 1**

**Row 2**

Graphical representation of table[][] of int[3][4] using row major implementation:

|  |
| --- |
| **[0][0]** |
| **[0][1]** |
| **[0][2]** |
| **[0][3]** |
| **[1][0]** |
| **[1][1]** |
| **[1][2]** |
| **[1][3]** |
| **[2][0]** |
| **[2][1]** |
| **[2][2]** |
| **[2][3]** |

**Row 0**

**Row 1**

**Row 2**

To reference an element in a two-dimensional array:

<array\_name>[<row\_index>][<column\_index>]

Key Points when referencing elements in a two-dimensional array in Java:

* The range of row index is from 0 to <rowsize> - 1
* The range of the column index is from 0 to <columnsize> - 1
* The first element is always at index <array\_name>[0][0]
* The last element is at index <array\_name>[rowsize-1][colsize-1]
* Index cannot be negative
* Index cannot be a float or double (real) number, only integer values
* Normally, a double loop is used as control structure for processing two-dimensional array.
* Normally, the processing is done row-by-row, and column-by-column within the same row.

Example: A table 3 x 4 of integer data :

int table[][] = new int[3][4]; table[0][0] = 2;

table[0][1] = 4;

table[0][2] = 5;

table[0][3] = 7;

table[1][0] = 1;

table[1][1] = 6;

table[1][2] = 9;

table[1][3] = 3;

table[2][0] = 4;

table[2][1] = 6;

table[2][2] = 2;

table[2][3] = 8;

Graphical representation of table[][] of int[3][4]:

**Col 0 Col 1 Col 2 Col 3**

**Row 0**

|  |  |  |  |
| --- | --- | --- | --- |
| **2** | **4** | **5** | **7** |
| **1** | **6** | **9** | **3** |
| **4** | **6** | **2** | **8** |

**Row 1**

**Row 2**

//Table 3 x 4 program public class Table3x4{

public static void main(String[] args){ int table[][] = new int[3][4];

int i, j; table[0][0] = 2;

table[0][1] = 4;

table[0][2] = 5;

table[0][3] = 7;

table[1][0] = 1;

table[1][1] = 6;

table[1][2] = 9;

table[1][3] = 3;

table[2][0] = 4;

table[2][1] = 6;

table[2][2] = 2;

table[2][3] = 8;

// print table in matrix format for(i = 0; i < 3; i++){

for(j = 0; j < 4; j++){

System.out.print(“ “ + table[i][j]);

}

System.out.println();

}

}

}

Using the same table, create a program that will initialize the table to zero and ask the user to input values. Thereafter, the program will print the inputted values in matrix form. The program should also display the sum of all the values inputted.

Exercise:

//Table 3 x 4 modified program import java.io.\*;

public class Table3x4Modified{

public static void main(String[] args){ int table[][] = new int[3][4];

int i, j, sum = 0, num; String input = “ “;

BufferedReader in = new BufferedReader(new

InputStreamReader(System.in));

// Initialize table to zero for(i = 0; i < 3; i++){

for(j = 0; j < 4; j++){ table[i][j] = 0;

}

}

// Input values

for(i = 0; i < 3; i++){ for(j = 0; j < 4; j++) {

System.out.print("Input table [" + i + "]["+

j + "] = ");

try{

input = in.readLine();

}catch(IOException e){

System.out.println(“Error!”);

}

num = Integer.parseInt(input); table[i][j] = num;

}

}

// computing the sum for(i = 0; i < 3; i++){

for(j = 0; j < 4; j++){

sum = sum + table[i][j]);

}

}

// print table in matrix format for(i = 0; i < 3; i++){

for(j = 0; j < 4; j++){

System.out.print(“ “ + table[i][j]);

}

System.out.println();

}

System.out.println(“Sum = ” + sum);

}

}

Exercises:

Write a program that will create a 2-dimensional array with a row size = 10 and a column size = 10 also. Or simply a 10 x 10 table, thereafter the program should:

* + 1. Initialize the array to zero.
    2. Ask the user to input integer values.
    3. Display the table in matrix form.
    4. Display the sum of all elements.
    5. Display the average of all elements.
    6. Display the number of negative elements in the table.
    7. Print only the elements on the main diagonals of the table.
    8. Modify the program and display the sum of the elements on a specified row.
    9. Modify the program and display the sum of the elements on a specified column.
    10. Assume three matrices of the same size, say matrix A, B and C. Write a program that will add the two matrices A and B and store the sum to

C. Matrix addition is done component-wise, i.e., C[i][j] = A[i][j] + B[i][j] where i,j are the row and column index respectively.

### 1.3 Array Lengths

In order to get the number of elements in an array, the length field of an array can be used. The length field returns the size of the array and is written as follows:

arrayName.length

Sample Program: Using length field

public class ArrayLength{

public static void main(String[] args){ int[] scores = new int[50];

int i;

for(i = 0; i < scores.length; i++) System.out.print(scores);

}

}

Note: It is better to use the length field as a condition statement of a loop to allow the loop to adjust automatically to different - sized arrays.

### Java Methods

To know more of what a method is, let us first consider the following scenario. Let us assume that you were a programmer and you were asked to write a program that will be used to handle Automated Teller Machine transactions. Such transactions includes: (1) balance inquiry, (2) deposit and (3) withdraw. Aside from these, the user should be presented a menu showing there transaction as possible options. For a non-trivial program like these, it is advise to write the codes under one method which the **main()**method.

It is better to subdivide the programs into smaller portions – which we call methods. Each method should be able to solve part of the bigger problem. In the scenario above, one method will handle the generation of the menu, another method will be in-charge of handling the balance inquiry, another method will be used for deposit transaction, and a method that will handle withdrawal transactions. Notice that each method does a specified job of other methods.

In a more concrete term, a method is basically a program by itself – it has inputs, output, and will perform some kind of processing steps.

In computer programming, methods are also called subprograms. An aspiring computer programmer should be able to determine based from the problem statements, what methods are needed and how to relate these methods in order to solve the problem.

Characteristics of Methods

* It can return one or no values
* It may accept as many parameters it needs or no parameter at all
  + **Parameters** are also called function arguments
* After the method has finished execution, it goes back to the method that called it

### Declaring Static Methods

**Static methods** are methods that can be invoked without instantiating a class. These methods belong to the class as a whole and not to a certain instance of a class. Static method is defined by the keyword static and can only be invoked inside another static method.

To declare methods, we follow the syntax:

<access\_modifier> static <return\_type>

<method name>([parameters]){ method body

}

Where,

* **access\_modifier** can either be public, private, protected (if no access is present, the default modifier is in effect).
* **static** is a keyword used to indicate that the method is a static method
* **return\_type** can be the different data types.
* **method\_name** is any name given by the programmer to identify the method.
* **parameters** are arguments that is passed from one-method to another (optional if no arguments are to be passed).

Sample Method Declarations:

public static int withParameter(int n1, int n2){

//body of the method

}

public static void withOutParameters(){

//body of method

}

### Calling Static Methods

To call a static method, the syntax to follow is,

ClassName.staticMethodName(params);

Or

staticMethodName(params);

Sample Program: Program that print the text “Hello World!” where “Hello” is to be printed by one method and “World!” is to be printed by another method.

//Declaring and Calling Methods public class UsingMethods{

public static void printHello(){

System.out.print(“Hello “);

}

public static void printWorld(){

System.out.println(“World!”);

}

public static void main(String[] args){ printHello();//same as UsingMethods.printHello printWorld();//same as UsingMethods.printWorld

}

}

The previous program is interpreted as,

* //Declaring and Calling Methods
  + A program comment
* public class UsingMethods{
  + The name of the class (and the file)
* public static void printHello(){
  + A user – defined static method without any return value whose method name is printHello without any parameters
* System.out.print(“Hello “);
  + Prints the text Hello on screen
* }
  + Ends the method body
* public static void printWorld(){
  + A user – defined static method without any return value whose method name is printWorld without any parameters
* System.out.println(“World!“);
  + Prints the text World! on screen
* }
  + Ends the method body
* public static void main(String[] args){
  + the main method (pre-defined method) with 1 parameter
* printHello();
  + Calling the method printHello and performing statements found therein
* printWorld();
  + Calling the method printWorld and performing statement found therein
* }
  + Ends the main method body
* }
  + Ends the class

Example: Calling **hello()** function three times inside **main()**

// calling hello() three times in main() public class Hello3x{

public static void hello(){

System.out.println(“Hello World!”);

}

public static void main(String[] args){ hello(); // 1st method call hello(); // 2nd method call hello(); // 3rd method call

}

}

Multiple invocation of the same method must be placed inside a loop.

// calling hello() 1000 times in main()

public class Hello1000x{

public stati void hello(){

System.out.println(“Hello World!”);

}

public static void main(String[] args){ int i;

for(i = 0; i < 1000; i++){ hello();

}

}

}

Exercise: What will happen if we write the **hello()** function after the **main()** function? Will the program behave as what is expected? If not, how will you remedy the situation where functions are written after the **main()** method?

### Calling a Method Inside another Method

Example:

A method can be invoked inside another method aside from

main().

// function call beside main() public class FunctionCallBesideMain{

public static void printHello(){

System.out.print(“Hello”);

}

public static void printSpace(){

System.out.print(“ “);

}

public static void printWorld(){

System.out.println(“world!”);

}

public static void allTogetherNow(){ printHello();

printSpace(); printWorld();

}

public static void main(String[] args){ allTogetherNow();

}

}

### Passing Variables in Methods

There are two ways of passing data to methods, one is to pass-by- value and the other is to pass-by-reference.

### Pass-by-value

When pass-by-value occurs, the method makes a copy of the value of the variable passed to the method.

Example: Passing a value to a method

//Pass by value

public class PassByValue{

//Method with parameter

public static void methodWithParam(int j){

//print value of j

System.out.println(“Value of j = “ + j);

}

public static void main(String[] args){ int i;

Pass i as a parameter which is copied to j

Return to the

caller

and perform the

i = 50;

statement after //call methodWithParam passing value of i

the method call methodWithParam(i);

//print value of i System.out.println(i);

}

}

The output of the above program would be:

Value of j = 50; Value of i = 50;

Supposed that you add the statement j = 100; before the System.out.println(“Value of j = “ + j); inside methodWithParam(int j), the output would be:

Value of j = 100; Value of i = 50;

Note: The method cannot modify the original value of the variable being passed. By default, all primitive data types when passed to a method are pass-by-value.

Sample program passing two parameters.

//Pass by value

public class PassByValue{

//Method with parameter

public static void methodWithParam(int j, int k){

//print value of j System.out.println(“Value of j = “ + j); System.out.println(“Value ok k = “ + k);

}

public static void main(String[] args){ int i;

i = 50;

/\*call methodWithParam passing value of i copied to both parameters of methodWithParam\*/

methodWithParam(i, i);

//print value of i System.out.println(i);

}

}

The output of the above program would be

Value of j = 50 Value of k = 50 Value of i = 50

### Pass-by-reference

When pass-by-reference occurs, the reference to an object is passed to the calling method. But unlike pass-by-value, the method can modify the actual object that the reference is pointing to since the location of the data pointed to is the same.

Example: Passing reference to a method

public class PassByReference{

public static void methodWithParam(int[] arr){

//change values of array

for(int i = 0; i<arr.length; i++) arr[i] = i + 50;

}

Pass numbers as parameter which whose memory reference is pointed to by

public static void main(String[] args){

//create array of integers int numbers[] = {10, 20, 30};

//print array values

for(int i = 0; i < numbers.length; i++) System.out.println(numbers[i]);

variable arr

Return to the caller

/\*call methodWithParam ans array \*/

pass reference to

and perform the methodWithParam(numbers);

statement after

the

method call

}

//print array values again

for(int i = 0; i < numbers.length; i++) System.out.println(numbers[i]);

}

The output of the above program is:

10

20

30

60

70

80

Note: Instead of copying the values passed by the method, the pointer to the memory reference is the one being passes since it does not involve a primitive data type.

### Methods with return types

So far, example given regarding methods makes use of **void** as a return type. The type **void** should be used as the return type only when the method does not return anything. However, there are cases

wherein the method will have to return a character, an integer, a float, or a double data type depending on the problem being solved.

Example: A program that will return integer type.

// method with return type int public class MethodWithReturnType{

//declaring method with int as return type public static int computeSum(int x, int y){

int z;

z = x + y; return z;

}

public static void main(String[] args){ int a, b, c;

/\*call method computeSum passing 8 and 13 as parameters\*/

System.out.println(“Sum = “ + computeSum(8, 13));

/\*calling method computeSum passing 100 and 200

as parameters and storing the result to variable c\*/

c = computeSum(100, 200);

System.out.println(“Sum = “ + c);

a = 22;

b = 14;

/\*call method computeSum passing values of variables A and b as parameters\*/

System.out.println(“Sum = ” + computeSum(a, b));

}

}

The output of the above program would be:

Sum = 21

Sum = 300

Sum = 36

Exercise:

1. What will happen if the statement **return z** was omitted?
2. Write a method named **Difference()** that accepts two integer parameters named **x** and **y**. The method should compute and return the value of the difference of **x** and **y**. Call this method inside **main()**.
3. Write a method named **Product()** that accepts two floating point parameters named **x** and **y**. The method should compute and return the value of the product for **x** and **y**. Use **double** as the return type. Call this method inside **main()**.

Example: Write a method that will accept an integer variable as parameters, thereafter, the method should return 1 if the integer is positive, otherwise it should return 0.

// function to return 1 if positive otherwise return 0

public static int numberPositive(int n){ int result;

if (n > = 0)

result = 1;

else

result = 0;

return result;

}

Another way to implement this method is as follows:

// Another implementation

public static int numberPositive(int n)

{

if (n >= 0)

return 1;

else

return 0;

}

### 2.5 Important things to remember about methods

1. If the method will not return anything, use **void** as the return type.
2. If the method will return a value, the appropriate data type should be specified.
3. If the method will return a value, don’t forget to use the return statement inside the method.
4. The data type of the value that appears after the return statement must be compatible with the return data type.
5. You can use any name for the method as long as you follow the naming convention and do not use Java keywords as method names.
6. A method may have zero, one or more parameters.
7. If the method does not have any parameter, leave the area in between the parentheses blank.
8. If the method has parameters, specify their data types and names.
9. Parameters should be separated by comma.
10. The name of the parameters does not matter. Always remember that the n*umber of parameters, their respective data types, and their sequence* are the things that actually matter!
11. When calling a method, constants, expressions and variables maybe used as actual parameters.
12. A method can return at the most only one value at any given point in time.

### Scope of Variable

The **scope** determines where in the program the variable is accessible. The scope also determines the lifetime or how long the variable can exist in memory. The placement of the variable, or where the variable is declared, determines the scope for that variable.

For example:

public class ScopeExample{

public static void main(String[] args){ int i = 0;

int j = 0;

### A

for(int k = 0 ; k<10; k++){

//some code

**B**

}

}

}

The sample code snippet written above represents two scopes indicated by the lines and letter representing the scope.

**Scope A** – the scope of variables i and j is true to the whole main(), meaning, both variables i and j can be used inside the main() block

**Scope B** - variable k can only be used inside the for loop block. Given a complete sample program with method,

public class PassByReference{

public static void main(String[] args){

//create an array of integers int numbers[] = {10, 20, 30};

**B**

**A**

**C**

}

//print array values

for(int i = 0; i < number.length; i++){ System.out.println(numbers[i]);

}

//call method and pass reference to array methodWithParam(numbers);

//print array values again

for(int i = 0; i < numbers.length; i++){ System.out.println(numbers[i]);

}

public static methodWithParam(int[] arr){

//change value of array

for(int i = 0; i< arr.length; i++){

**E** arr[i] = i + 50; **D**

}

}

}

Inside the main() method: numbers[] – scope A i in B – scope B

i in C – scope C

Inside the methodWithParam() method:

arr[] – scope D

i in E – scope E

# REFERENCES

### Books

Cadenhead, R., Lemay, L. (2007), **Sams Teach yourself Java 6 in 21 Days, 5th Edition**, Sams Publishing.

Deitel, P.J., Deitel, H.M. (2007), **Java: How to Program, 7th Edition**, Prentice Hall.

Bates, B., Sierra, K. (2005), **Head First Java, 2nd Edition,** O’ Reilly.

Schildt, H. (2005), **Java: The Complete Reference, 5th Edition** , McGraw-Hill.

Williams, A. (2002), **Java 2, Core Language, Little Black Book**, Paraglyph.

### Websites

**The Java Tutorial: A Practical Guide for Programmers**. Available at <http://java.sun.com/docs/books/tutorial>

### Java in a Nutshell. Chapter 4: The Java Platform.

Available at <http://www.unix.org.ua/orelly/java-ent/jnut/ch04_10.htm>

### Diagnosing Java Code: Java Generics Without the Pain, Part 1 by Eric Allen.

Available at

<http://www-106.ibm.com/developerworks/java/library/j-djc02113.html>

### Java Programming (with Passion!)

Available at [http://www.javapassion.com/portal/index.php?option=com\_content&view=](http://www.javapassion.com/portal/index.php?option=com_content&view=article&id=66) [article&id=66](http://www.javapassion.com/portal/index.php?option=com_content&view=article&id=66)