**Defining Java**

Ø  Java is a high-level programming language developed by Sun Microsystems.

Ø  Java can be used to write applets (small applications running inside other applications hence the diminutive suffix 'let') but is not the only language that can produce applets.

**History of Java**

Ø  Java was developed at Sun Microsystems, Inc. (Sun) by James Gosling.

Ø  The language needed to be small, fast, reliable, and portable (hardware independent).

Ø  With the goals it had achieved (smallness, speed, reliability, portability), Java was discovered to be ideal for web pages.

Ø  Java can also be used to write stand-alone applications.

**Java Technology**

Ø  A programming language

Ø  A development environment

Ø  An application environment

Ø  A deployment environment

**Features of Java**

Ø  Object-Oriented

o   Java uses classes to organize code into logical modules

Ø  Portable and Platform-Independent

o   Java is platform-neutral, which means that programs developed with Java can run on any computer system with no changes

Ø  Dynamically Linked

o   Java codes are linked at runtime

Ø  Multithreaded

o   Java programs can contain multiple threads of execution

Ø  Garbage Collected

o   Java does its own garbage collection, which means that programs are not required to delete objects allocated in memory

**Java Requirements**

ü  Pentium D processor or higher

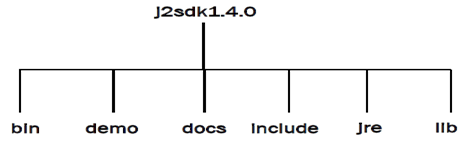
ü  At least 128MB of RAM

ü  Using an applet within a Web Browser with

ü  JDK plug-in

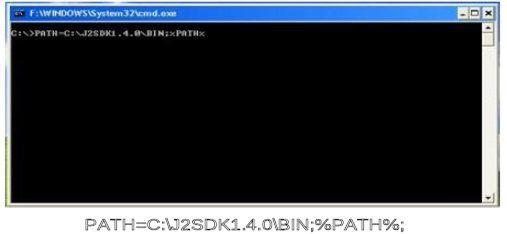
**Java Development Environment**

Ø  Installing JDK (Java Development Kit) produces several folders in your system. Each of these folders holds pertinent files that would be necessary in building your program. For instance, the bin folder contains executable files for compiling your program (javac.exe).



**Java Development Environment – Using Path**

Ø  Since javac.exe (the one that compiles your program) is located in the bin directory. It is necessary to execute this command, by doing so you can now compile on any directory your program may be.



**Java Development Environment– Using CLASSPATH**

Ø  The class path is used to tell Java where to look for classes. Set the environment variable CLASSPATH if there’s a need to look for classes in different folders.

**Java Development Environment**

Ø  The tool to be used for programming is called JCreator

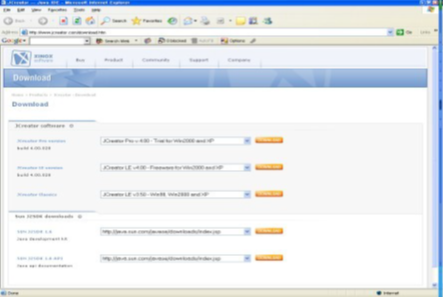
Ø  You can get a copy of JCreator at

o   [https://www.jcreator.com](https://www.jcreator.com/)

Ø  Download the free version (JCreator LE).

Ø  Learn more about JCreator by surfing their sites 





**Other Java programming environment:**

      Symantec Visual Café

      Microsoft J++

      Borland JBuilder

      IBM Visual Age for Java

      Sun Java Forte

      Java 2 SDK 1.4.2 FCS

**Applets and Applications**

Ø  Java can be used to create two types of programs:

§  **Applet**

·         special applications designed to run within the context of a Web browser

§  Application

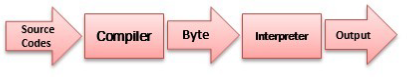
·         stand-alone program that does not need a browser to run

**Java Program Cycle**

Ø  Use any text editor to create source code — .java file

Ø  Source code compiled according to JVM to get byte code — .class file

Ø  Byte code executed using an interpreter



**Java Program**

      A minimum Java program has the following format:

**public class <*classname*>{**

**public static void main (String[] args) {**

**<*program statement*>}}**

**Java Coding Guidelines**

Ø  Java program files must

o   Have same name as public class

o   End with extension .java

Ø  Use comments for documentation and readability

Ø  White spaces are ignored

Ø  Indent for readability

**Java Statements**

Ø  A statement is one or more lines of code terminated by a semicolon.

Example:

                                    System.out.print(“Hello, World!”);

**Java Blocks**

Ø  A block is formed by enclosing statements by curly braces.

Ø  Block statements can be nested indefinitely.

Ø  Any amount of white space is allowed.

Example:

public static void main (String[]args) {

System.out.println(“Hello”);

System.out.print(“World!”);}

**Java Comments**

Ø  A **comment**is an optional statement used to describe what a program or a line of program is doing.

// This is a comment

/\* This is a comment \*/

/\*\* This is a special comment \*\*/

o   Comment lines are ignored by the compiler.

public class Hello {

/\*\*

\* My First Java Program

\*/

public static void main (String[] args) {

// print “Hello, World!” on screen

System.out.println(“Hello, World!”); }}

**Java Identifiers**

Ø  Identifiers used to label variables, methods, classes, etc.

Ø  Case-sensitive

Ø  May contain letters, digits, underscore and dollar sign ($)

Ø  May not start with a digit

Ø  May not use Java keywords

**Java Identifiers Rules and Guidelines**

**Rules:**

§  Identifiers can use alphabetic characters of either case (a–z and A–Z), numbers (0–9), underscores ( \_ ), and dollar signs ( $ ).

§  Identifiers cannot start with a number.

§  Keywords cannot be used as identifiers (for this reason keywords are sometimes called reserved words).

**Guidelines:**

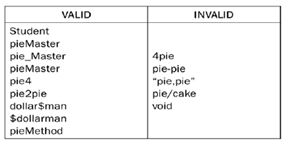
§  Name your identifiers close to its functionality.

§  Method and variable names start in lowercase while classes start in uppercase.

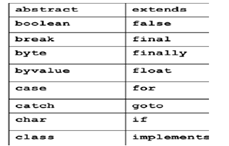
§  For multi-word identifiers, either use underscores to separate the words, or capitalize the start of each word.

§  Avoid starting the identifiers using the underscore.

**Java Identifier: Example**



**Java Keywords**



**Java Literals**

Ø  **Literals**are the representation of values.

o   Integers

o   Floating Point Numbers

o   Booleans (true or false)

o   Strings (enclosed in “ “)

o   Characters Java Literals (enclosed in ‘ ‘)

**Integer Literal**

Ø  can be expressed in three different bases:

o   Octal (base 8)

o   Decimal (base 10)

o   Hexadecimal (base 16)

**Floating Point Literal**

Ø  appears in several forms

Ø  typical form makes use of digits and a decimal point

Ø  note that digits may appear before or after or before and after the decimal point

**Boolean Literal**

Ø  only two *Boolean literals*exist: true and false, representing the Boolean concepts of true and false, respectively

**String Literal**

Ø  sequence of characters within double quotes

Ø  the characters can be escape sequences

**Character Literal**

Ø  Character literals come in two forms. They both use the single quote (‘ ’) as a delimiter. The first form places the literal character between single quotes. Examples include 'a', '+', and '$'.

Ø  Some characters, such as the newline character, don't have visible literal representations. For these, an escape sequence must be used, which consists

**Data Types**

There are two kinds of data types:

***Simple -***Built-in or primitive Java data types

***Composite***

Created by the programmer using simple types, arrays, classes, and interfaces

**Primitive Data types**

The Java programming language defines eight primitive data types:

Ø  boolean (for logical)

Ø  char (for textual)

Ø  byte

Ø  short

Ø  int

Ø  long (integral)

Ø  double

Ø  float (floating-point)

**Boolean Data Type**

Ø  one-bit wide and takes on the values true or false

Ø  default value: **false**

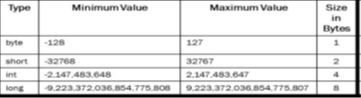
Ø  cannot be cast to a number or any other type

**Char Data Type**

Ø  represents a 16-bit Unicode character

Ø  must have its literal enclosed in single quotes (‘’)

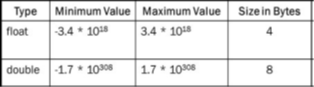
**Integer Data Type**



**Floating Point Data Type**

has types: **float**(32 bits single precision) and **double**(64-bit double precision)

**Java Variables**



Ø  A variable is an item of data used to store the state of objects.

Ø  A variable is composed of:

o   *Data type*– indicates the type of value that the variable can hold

o   *Name*– must follow rules for identifiers

**Declaring and Initializing Variables**

A variable is declared as follows:

Note:

Values enclosed in <> are required values, while those values in [] are option

**Guideline in Declaring and Initializing Variables**

1. Always initialize your variables as you declare them.

2. Use descriptive names for your variables.

3. Declare one variable per line of code.

**Java Variable Example**

public class VarSample {

public static void main(String[]args){

boolean result;

char option;

option = ‘C’;

double grade = 0.0;}}

**Displaying Variable Data**

Ø  To display the value of a certain variable, we use the following commands:

System.out.print()

System.out.println()

Example:

public class VarOutput {

public static void main(String[] args) {

int value = 10;

char x;

x = ‘A’;

System.out.println(value);

System.out.println(“The value of x = ” + x);}}

Ø  System.out.print()

o   Does not append newline at the end of the data output

Example:

System.out.print(“Hello”);

System.out.print(“World”);

Output:

HelloWorld

Ø  System.out.println()

o   Appends a newline at the end of the data output

Example:

System.out.println(“Hello”);

System.out.println(“World”);

Output:

Hello

World

**Types of Variables**

Ø  **Primitive Variables**

o   variables with primitive data types

o   stores data in the actual memory location where the variable is

Ø  **Reference Variables**

o   variables that stores the address in the memory location

o   points to another memory location where the actual data is

**Constants**

Ø  value never changes

Ø  Use the **final**type modifier in class definition

public class Variables {

public static void main(String[] args){

final double PI = 3.14;}}

**Java Expressions**

Ø  An expression produces a result and returns a value.

Ø  An expression can be any combination of variables, literals, and operators.

Ø  Purposes of expressions:

o    to compute values

o    to assign values to variables

o    to control the flow of execution

Examples:

x = 5;

y = x;

z = x \* y;

**Java Operators**

Ø  Operators are special symbols that perform specific operations on one, two, or three operands, and then return a result.

Operators used are:

o    Arithmetic

o    Increment and Decrement

o    Assignment

o    Relational

o    Logical

**Arithmetic Operators**

+ Addition

- Subtraction

\* Multiplication

/ Division

% Modulo

x = 6; // assign 6 to x

y = 4; // assign 4 to y

x = x + 2; // x is equal to 8

y = y - 3; // y is equal to 1

z = x \* y; // z is equal to 8

z = x / y; // z is equal to 8

z = x % y; // z is equal to 0

**Increment and Decrement Operators**

++ Pre/Post-Increment

- - Pre/Post-Decrement

if num = 5

z = num++ // z = 5

z = ++num // z = 6

**Assignment Operators**

= Assignment

+= Addition

-= Subtraction

\*= Multiplication

/= Division

%= Remainder

a = 8;

b = 12;

c = 3;

d = 15;

b += 1; // same as (b=b+1)

a -= 2; // same as (a=a-2)

c \*= 2; // same as (c=c\*2)

d /= 5; // same as (d=d/5)

**Relational Operators**

< Less than

> Greater than

<= Less than or equal to

>= Greater than or equal to

!= Not equal to

Expression                   Result               Expression                    Result

3+4 == 7                       true                               3+4 < 10           true

3+4 != 7                        false                             3+4 <=              10 true

3+4 != 2+6                    true                               3+4 == 4+4       false

**Logical Operators**

|| OR

^ Exclusive-OR

! NOT

 (3+2==5)                     &&                   (6+2==8) true

(4+3==9)                       &&                    (3+3==6) false

(3+6==2)                       ||                       (4+4==8) true

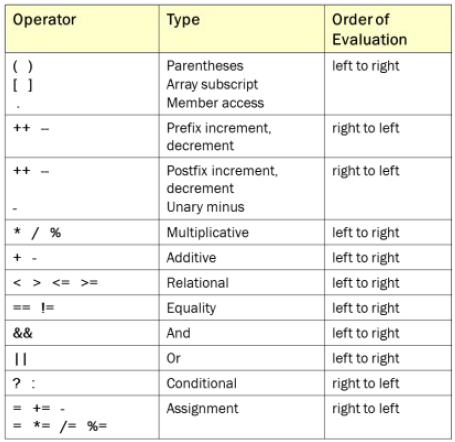
(5+7==12)                     ^                       (4+3==8) true

(5+7==12)                     ^                       (4+3==7) false

(4+3==5)                                                                false

!(4+3==5)                                                               true

**Operator Precedence**



**Exercises**

Answer the following:

Assume: m=10

a.       3 – 2 + 5 + 6 – 1 = \_\_\_\_\_\_\_

b.       4 \* (5 + 4) / 2 = \_\_\_\_\_\_\_

c.       5 % 2 + (2 \* 8) = \_\_\_\_\_\_\_

d.       !((((m-2)\*4)+15 / 5) == (15%3)\*7)) result = \_\_\_\_\_\_\_

e.       !(m == 10) && (m+3 == 13) result = \_\_\_\_\_\_\_

Assume: b=6 and num=10

f.        num-=2 num = \_\_\_\_\_\_\_

g.  num = b + 3; = \_\_\_\_\_\_\_\_

- -num ;             num = \_\_\_\_\_\_\_

                           num- -;              num = \_\_\_\_\_\_\_

h. num += 2            num = \_\_\_\_\_\_\_

                         i. num = 11 % 3      num = \_\_\_\_\_\_\_

**Java Statements**

Ø  In Java, a *statement*is one or more lines of code ending with a semi-colon (;).

Ø  It generally contains expressions (expressions have a value).

**Simple Statement**

o   Basic building blocks of a program

o   One or more lines terminated by a semicolon (;)

Example:

System.out.println (“Hello, World!”);

**Compound Statement or Block**

o   Used to organize simple statements into complex structures

o   One or more statements enclosed by braces { }

o   Recommended to indent blocks for readability

Example:

public static void main (String[] args) {

System.out.println (“Hello, ”);

System.out.println (“World!”);}

**Java Control Structure**

Ø  Control structures allow to change the ordering of how the statements in a program is executed

Types of control structures:

o    ***Decision control structures***

§  allow to select specific sections of code to be executed

o   ***Repetition control structures***

§  allow to execute specific sections of the code a number of times

**Decision Control Structures**

Ø  Java statements that allows a programmer to select and execute Specific blocks of code while skipping other sections.

Ø  **If Structure**

o   The *if*statement specifies that a statement (or block of code) will be executed if and only if a certain Boolean statement is true.

The general format of an *if*statement is:

if (*expression*)

*statement or block;*

or

if (*expression) statement;*

Ø  Where *expression*represents a relational, equality, or logical expression (*conditional expression*).

Ø  If there is a need to execute several statements (*compound statement*), then left and right {} braces can group these statements.

if (*expression*){

*statement1;*

*statement2;…*}

Every time Java encounters an *if*statement,

1.       It determines whether *expression*is true or false.

2.       If *expression*is true, then Java executes *statement*.

3.       If *expression*is false, then Java ignores or skips the remainder of the *if*statement and proceeds to the next statement.

Ø  Making a decision involves choosing between two alternative courses of action based on some value within a program.

Ø  If Structure: Example 1

o   Suppose that the passing grade on an examination is 75 (out of 100). Then the if statement may be written in Java as:

|  |  |
| --- | --- |
| |  | | --- | | Exercises  1.       Test if the value of the variable count is greater than 10. If the answer is yes, print, “Count is greater than 10”.  2.       Test if the value of the variable number is an odd number. If the answer is yes, print “This number is not an even number.”  3.       Test if the value of the variable age is greater than 18 and less than or equal to 40. If the answer is yes, print “You’re not getting old!” | |

if (grade >= 75)

System.out.println(“You

Passed!”);

Ø  If Structure: Example 2

                   int grade = 68;

                   if (grade > 60){

                   System.out.println(“Congratulation s!”);

                   System.out.println(“You Passed!”);

Ø  **If-Else Structure**

o    The *if-else*statement is used when you want to execute one statement if a condition is true, and another statement if the condition is false.

The general format of an *if-else*statement is:

if (*expression*)

*statement1*;

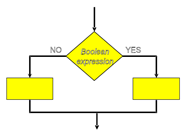
else

*statement2*;

or

if (*expression) statement1;*

else *statement2;*



Ø  As in the *if*statement, compound statements may also be used in *if-else*statements (provided they be grouped together by braces).

if (*expression*){

*statement1;*

*statement2;*

*…*}

|  |  |
| --- | --- |
| |  | | --- | | **Exercises**  1.       Test if the value of the variable count is greater than 10. If it is, print “Count is  greater than 10”, otherwise, print “Count is less than 10”.  2.       Test if the value of the variable number is an odd number. If it is, print “This number is not an even number” otherwise, print “This number is an odd number.”  3.       Test if the value of the variable age is greater than 18 and less than or equal to 40. If it is, print “You’re not getting old!” otherwise, print “You’re not getting younger.” | |

else{

*statement3;*

*statement4;*

*…*}

Ø  **If-Else Structure: Example**

            int grade = 68;

            if (grade > 60)

            System.out.println(“Congratulations!”);

            else

           System.out.println(“Sorry you failed!”);

Ø  **Nesting If and If-Else Structure**

o   The statement in the else-clause of an if-else block can be another if-else structure.

The general format of an *if-else-else if*statement is:

if (*expression 1*) {

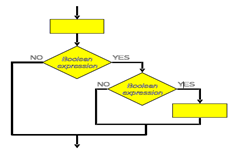
*statement1*;}

else if ( *expression 2*) {

*statement2*;}

else {

*statement 3;*}



Ø  **Nesting If and If-Else Structure: Example**

int grade = 68;

if(grade > 90) {

System.out.println(“Very Good”);}

else if(grade > 60) {

System.out.println(“Good”);}

else {

System.out.println(“Failed”);}

**Common Errors**

Ø  The condition inside the if-statement does not evaluate to a Boolean value

Ø  The variable number does not hold a Boolean value

Ø  Writing elseif instead of else if

Ø  Using = instead of == for comparison

**Switch Structure**

Ø  The *switch*statement enables a program to switch between different outcomes based on a given expression.

The general format of a *switch*statement is:

switch (*switch\_expression*){

*case 1:*

*statement1;*

*case 2:*

*statement2;*

*default:*}

Ø  Every time Java encounters a *switch*statement,

1.       It evaluates the switch expression, and jumps to the case whose selector matches the value of the expression.

2.       The program executes the statements in order from that point on until a break statement is encountered, skipping then to the first statement after the end of the switch structure.

3.       If none of the cases are satisfied, the default block is executed. However, note that the default part is optional.

Ø  Unlike with the if statement, the multiple statements are executed in the switch statement without needing the curly braces.

Ø  When a case in a switch statement has been matched, all the statements associated with that case are executed. Not only that, the statements associated with the succeeding cases are also executed.

Ø  To prevent the program from executing statements in the subsequent cases, use a break statement as the last statement.

|  |  |
| --- | --- |
| |  | | --- | | **Exercises**  Ø  Analyze what happens when the following code is executed with val equal to 10? 100? 1,000? Write a program to verify your answer.    switch (val){  case 10:  System.out.println("ten");  case 100:  System.out.println("hundred");  default:  System.out.println("thousand");} | |

**Switch Structure: Example**

switch (x){

case 1:

y = 1;

break;

case 2:

y = 2;

break;

case 3:

y = 3;

break;

default:

y = 0;}

Ø  **Repetition Control Structures**

o   Java statements that allows a programmer to execute specific blocks of code a number of times.

Ø  **While Structure**

o   The *while*loop is a statement or block of statements that is repeated as long as some condition is satisfied.

The general format of a *while*statement is:

while (*boolean\_expression*){

*statement1;*

*statement2*}

Ø  **While Structure: Examples**

int counter = 1;

while (counter <= 10) {

System.out.println("counter = " + counter);

counter += 1;}

int counter = 1;

while (counter\*counter < 1000) {

System.out.println(counter \* counter);

counter++;}

Ø  **Do-While Structure**

o   The *do-while*loop is similar to the while-loop. The statements inside a do-while loop are executed several times as long as the condition is satisfied.

The general format of a *do-while*statement is:

do{

*statement1;*

*statement2*

} while ( *boolean\_expression)*;

Ø  **Do-While Structure: Examples**

nt x = 0;

do {

System.out.println(“x = ” + x);

x++;}

 while(x < 10);

do {

System.out.println(“Hello!”);

} while (false);

Ø  **For Structure**

o   The *for*statement instructs a program to perform a block of code a specified number of times.

The general format of a *for*statement is:

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

*statement 1;*

*statement 2;*}

Ø  **For Structure: Examples**

int x;

for(x=1; x<=10; x++){

System.out.println(“x= ” + x);}

int i;

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

System.out.println(i);}

**Branching Statements**

Ø  These statements redirect the flow of program execution.

These include:

o   **Break Statement**

§  The *break*statement causes an exit from an enclosing loop.

§        In other words, encountering the *break*statement causes immediate termination of the loop.

§   Program control picks up at the code following the loop.

o   **Break Statement: Example**

§   This program segment will normally print all numbers between 1 to 100. But because of the *break*statement, only the numbers between 1 to 50 will be printed.

for (ctr = 1; ctr <= 100; ++ctr) {

System.out.print(“\nctr = “ + ctr);

if (crt = = 50)

break;}

o   **Continue Statement**

§  The *continue*statement works in a somewhat similar way to the *break*statement.

§  But instead of forcing loop termination, *continue*forces the next iteration for the loop to take place, skipping any code in between.

o   **Return Statement**

§   The *return*statement is used to exit from the current method.

§  Flow of control returns to the statement that follows the original method call.

**Arrays**

Ø  Suppose there are three variables of type intwith different identifiers for each variable.

int number1;

int number2;

int number 3;

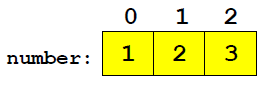
number1 = 1;

number2 = 2;

number3 = 3;

Ø  This seems like a tedious task in order to just initialize and use the variables especially if they are used for the same purpose.

Ø  An ***array***is a sequence of memory locations for storing data set out in such a way that any one of the individual locations can be accessed by quoting its *index*number.



Ø  The individual items within an array are called ***elements***.

**Declaring Java Arrays**

Ø  To declare an array, write the data type, followed by a set of square brackets **[]**, followed by the identifier name.

**ElementType[] arrayName;**

**Example:**

**int[] x;**

**int[] ages;**

Or equivalently

**int x[];**

**int ages[];**

**Initializing Array Variables**

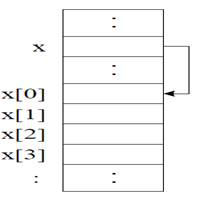
Ø  To initialize an array, you use the **new**operator to allocate memory for objects. The new operator is followed by the data type with the number of elements to allocate specified. The number of elements to be allocated is placed within the [] operator.

**Example:**

**name = new String[100];**

**age = new int[10];**

Ø  **Memory model for array storage**



Ø  Like other variables, an array variable can be initialized when it is declared.

**ElementType[] arrayName = new ElementType[sizeOfArray];**

or

**ElementType arrayName[] = new ElementType[sizeOfArray];**

**Example:**

**int[] x = new int[10];**

**int[] ages = new int[100];**

Ø   An array can be also created by directly initializing it with data.

**Example:**

**int[] arr = {1, 2, 3, 4, 5};**

Ø  This statement declares and creates an array of integers with five elements, and initializes this with the values 1, 2, 3, 4, and 5.

Ø  Creates an array of Boolean variables with identifier results.

Ø  This array contains 4 elements that are initialized to values {true, false, true, false}.

**boolean[] results = {true, false, true, false};**

Ø  Creates an array of 4 double variables initialized to the values {100, 90, 80, 75}.

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

Ø  Creates an array of strings with identifier days and initialized. This array contains 7 elements.

**String[] days = {“Mon”, “Tue”, “Wed”, “Thu”, “Fri”, “Sat”, “Sun”};**

**Exercise:**

Ø  Declare and create array variables for the following data:

o   An array of 15 doubles

o    An array of 20 strings

**Accessing Array Element**

**Example:**

//assigns 10 to the first element

in the array

**ages [0] = 10;**

//prints the last element in the

array

**System.out.print(ages[99]);**

Ø  To access an array element, or a part of the array, use a number called an *index*or a *subscript*.

Ø  **Index number or subscript**

o   assigned to each member of the array, to allow the program to access an individual member of the array

o   an integer beginning at zero and progresses sequentially by whole numbers to the end of the array

o   *Note:*index is from 0 to (sizeOfArray-1) Accessing an

Ø  The example below illustrates how to print all the elements in the array.

**public class ArraySample {**

**public static void main(String[] args) {**

**int[] ages = new int[100];**

**for(int i=0; i<100; i++) {**

**System.out.print(ages[i]);}}}**

**Coding Guidelines**

1.       It is better to create the array right away after you declare it.

**int[] arr = new int[100];**

2.       The elements of an n-element array have indexes from 0 to n-1. Note that there is no array element arr[n]! This will result in an arrayindex- out-of-bounds exception.

3.       You cannot resize an array. However, you can create a new array of a different size, copy the desired data into the new array, and assign the new array to the array variable.

**Array Length**

Ø  All array indices begin at 0. The number of elements in an array is stored as part of the array object in the ***length***attribute.

Ø  Use the ***length***attribute to iterate on an array as follows:

**int list[] = new int[10];**

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

**System.out.println(list[i]);}**

**Array Manipulation**

**/\*\* Array manipulation \*/**

**public class upavon{**

**public static void main(String[] args) {**

**int [] sizes = {31,28,31,30,31,30,31,31,30,31,30,31};**

**System.out.print("number ... ");**

**int month = (int) wellreader.read\_number();**

**if (month < 1 || month > 12){**

**System.out.println("Not a valid month number");**

**} else {**

**System.out.print("That month has ");**

**System.out.print(sizes[month-1]);**

**System.out.println(" days");}**

**System.out.print("Length of sizes array: ");**

**System.out.println(sizes.length);**

**int day = month;**

**sizes = new int[7];**

**for (int k=0;k<7;k++) sizes[k]=24;**

**if (day < 1 || day > 7){**

**System.out.println("Not a valid day number");**

**} else {**

**System.out.print("That day has ");**

**System.out.print(sizes[day-1]);**

**System.out.println(" hours");}**

**System.out.print("Length of sizes array: ");**

**System.out.println(sizes.length);**

**}}**

**Multidimensional Arrays**

Ø  Multidimensional arrays are implemented as arrays of arrays.

Ø  Multidimensional arrays are declared by appending the appropriate number of bracket pairs after the array name.

**Example:**

//integer array 512 x 128 elements

**int[][] twoD = new int[512][128];**

 //character array 8 x 16

**char[][] twoD = new char[8][16];**

//String array 4 rows x 2 columns

**String[][] dogs = {{“terry”, “brown”},**

**{“kristin”, “white”},**

**{“toby”, “gray”},**

**{“fido”, “black”}};**

Ø  Accessing an element in a multidimensional array is the same as accessing elements in a one dimensional array.

**Example:**

o   To access the first element in the first row of the array dogs, we write,

**System.out.print(dogs[0][0]);**

o   This will print the String “**terry**” on the screen.

**Example**

**int[][] matrix = {{0, 1, 2, 3},**

**{1, 0, 3, 2}, {2, 3, 0, 1}, {3,**

**2, 1, 0}};**

**int row, col;**

**for(row = 0; row < 4; row++) {**

**for(col = 0; col < 4; col++){**

**System.out.print(“ ” + matrix[row][col]);**

**System.out.println();**

**}}**