JAVA

UNIT 2: DATA TYPES, VARIABLES, AND ARRAYS

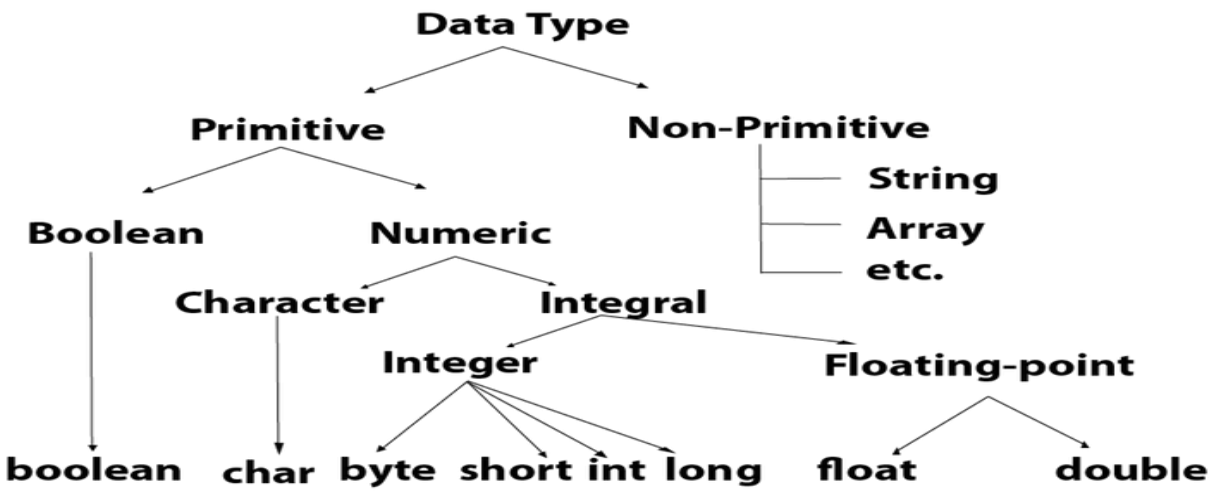
Data types in Java:

- Data types specify the different sizes and values that can be stored in the variable.
- There are two types of data types in Java:
    1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
    2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

1. Primitive data types:

- **primitive data types** are the building blocks of data manipulation.
- These are the most basic data types available in **Java language**.
- There are 8 types of **primitive data types**:

    1. boolean data type
    2. byte data type
    3. char data type
    4. short data type
    5. int data type
    6. long data type
    7. float data type
    8. double data type



Variables in Java:

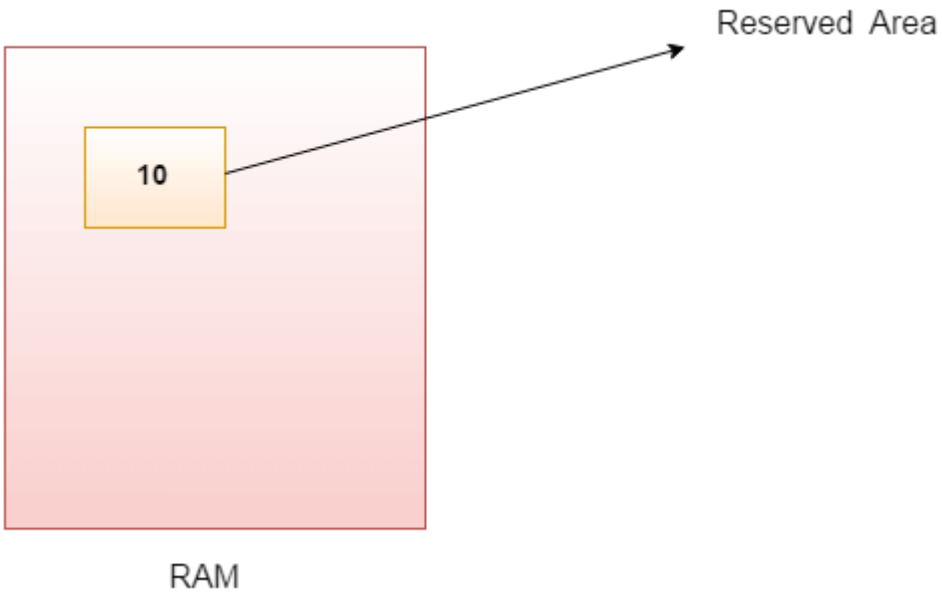
- A variable is a container which holds the value while the Java program is executed.
- A **variable** is assigned with a **data type**.
- **Variable** is a name of **memory location**.
- **Variable** is name of **reserved area allocated** in **memory**.
- In other words, it is a name of memory location. It is a combination of "**vary + able**" that means its **value can be changed**.
- There are **three types** of **variables** in java: **local, instance** and **static**.

Syntax:

Data_type variable_name = values;

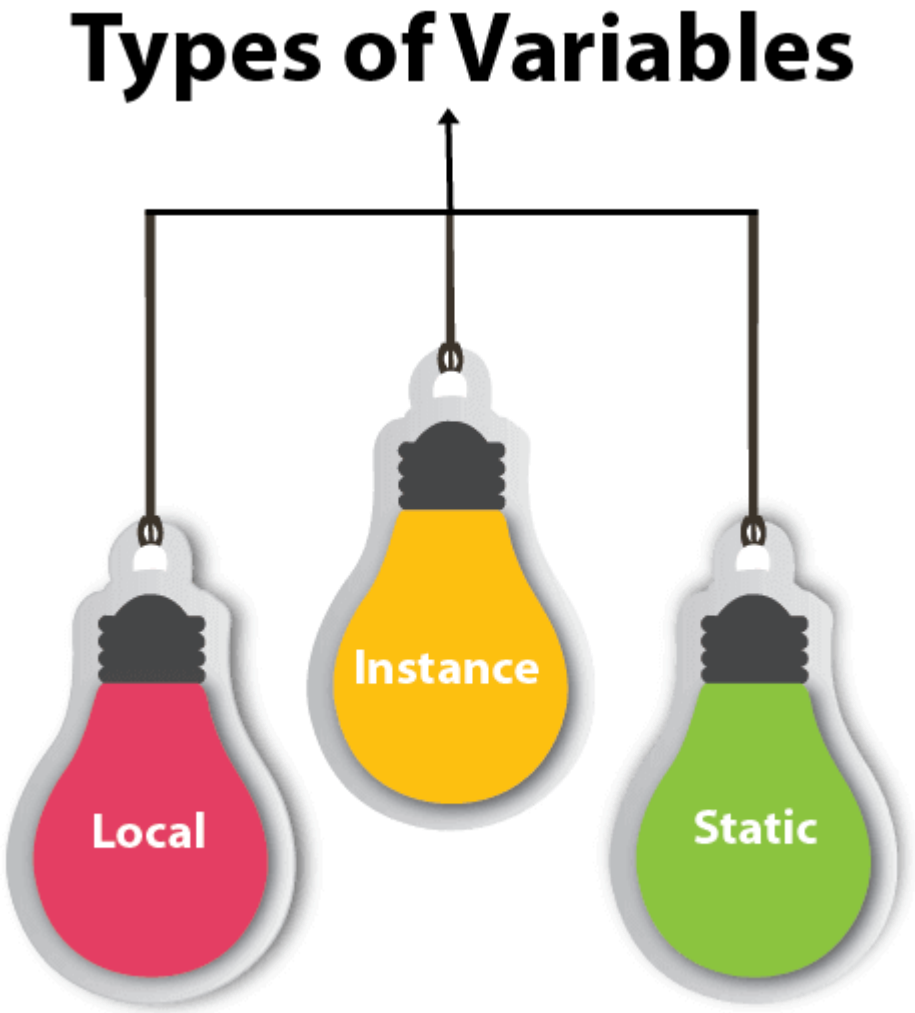
Example:

int number = 10;



int data=10; //Here **data** is variable.

```
int a, b, c;          // Declares three int, a, b, and c.
int a = 10, b = 10;   // Example of initialization
byte B = 22;          // initializes a byte type variable B.
double pi = 3.14159; // declares and assigns a value of PI.
char a = 'a';         // the char variable a is initialized with
value 'a'
```



Local Variable:

- A variable declared inside the body of the method is called local variable.
- You can use this variable only within that method.
- Local variables are declared in methods, constructors, or blocks.
- Access modifiers(private, public, protected) cannot be used for local variables.
- A local variable cannot be defined with "static" keyword.

Example:

Here, **age** is a local variable.

```
public class Test {
   public void pupAge() {
      int age = 0;
      age = age + 7;
      System.out.println("Puppy age is : " + age);
   }

   public static void main(String args[]) {
      Test test = new Test();
      test.pupAge();
   }
}
```

Output:

Puppy age is: 7

Instance Variable:

- A variable declared inside the class but outside the body of the method, is called instance variable.
- It is not declared as static.
- It is called instance variable because its value is instance specific and is not shared among instances.
- Access modifiers can be given for instance variables.

Example:

```
public class Employee {

   // this instance variable is visible for any child class.
   public String name;

   // salary  variable is visible in Employee class only.
   private double salary;

   // The name variable is assigned in the constructor.
   public Employee (String empName) {
      name = empName;
   }

   // The salary variable is assigned a value.
   public void setSalary(double empSal) {
      salary = empSal;
   }

   // This method prints the employee details.
   public void printEmp() {
      System.out.println("name  : " + name );
      System.out.println("salary :" + salary);
   }

   public static void main(String args[]) {
      Employee empOne = new Employee("Ransika");
      empOne.setSalary(1000);
      empOne.printEmp();
   }
}
```

Output:

Name: Ransika

Salary: 1000.0

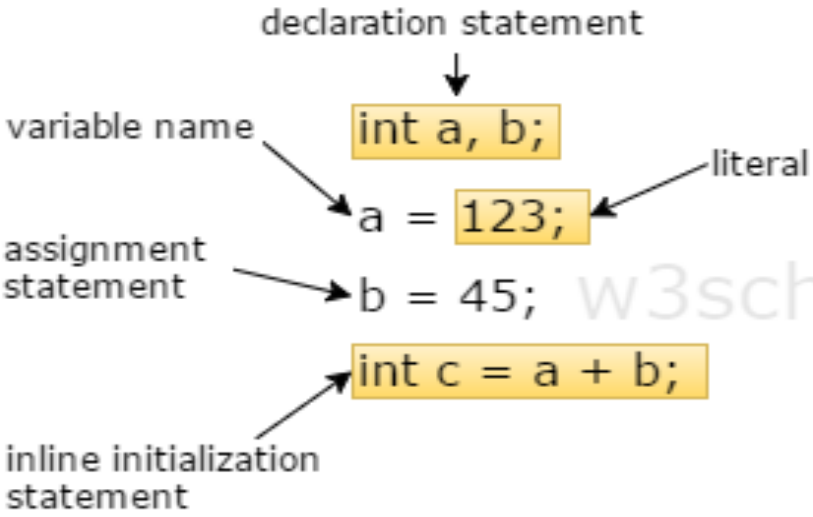
static Variable:

- Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor.
- You can create a single copy of static variable and share among all the instances of the class.
- Memory allocation for static variable happens only once when the class is loaded in the memory.

```
class A{
int data=50;//instance variable
static int m=100;//static variable
void method(){
int n=90;//local variable
}
}//end of class
```

Declaring Variable:

- To create a variable, you must specify the type and assign it a value:



Declaration and assignment statements

Syntax

type variable_name = values;

Example

Create a variable called **name** of type `String` and assign it the value "**John**":

```
String name = "John";
System.out.println(name);
```

Dynamic initialization of variable:

- If any variable is not assigned with value at compile-time and assigned at run time is called dynamic initialization of a variable.

Example:

```
class demo
{
    Public static void main(String args[ ])
    {
        int radius;
        define pi = 3.14*radius*radius;
        System.out.println("Area:", + radius);
    }
}
```

SCOPE AND LIFETIME OF A VARIABLE

Variable Type	Scope	Lifetime
Instance Variable	All through the class except in static methods.	Until the object is available in the memory.
Class Variable	All through the class.	Until the end of the program.
Local Variable	Within the block it is declared.	Until the control leaves the block.

Java Class Libraries:

- The Java Class Library is a set of dynamically loadable libraries that Java Virtual Machine languages can call at run time.

List of Library Classes in Java:

Library classes	Purpose of the class
Java.io	Use for input and output functions.
Java.lang	Use for character and string operation.
Java.awt	Use for windows interface.
Java.util	Use for develop utility programming.
Java.applet	Use for applet.
Java.net	Used for network communication.
Java.math	Used for various mathematical calculations like power, square root etc.

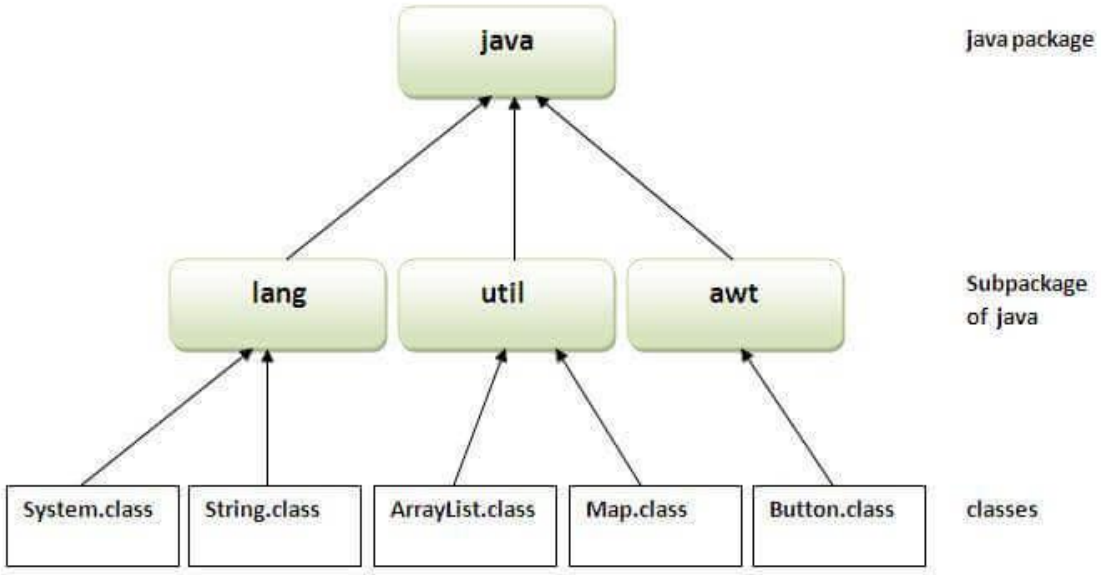
Java library

```
import java.io.*;
public class Employee {

   // salary  variable is a private static variable
   private static double salary;

   // DEPARTMENT is a constant
   public static final String DEPARTMENT = "Development ";

   public static void main(String args[]) {
      salary = 1000;
      System.out.println(DEPARTMENT + "average salary:" + salary);
   }
}
```



Java Token:

- In Java, the program contains classes and methods. Further, the methods contain the expressions and statements required to perform a specific operation.
- These statements and expressions are made up of **tokens**.

- we can say that the expression and statement is a set of **tokens**.
- The tokens are the small building blocks of a Java program that are meaningful to the Java compiler.
- The Java compiler breaks the line of code into text (words) is called **Java tokens**.
- These are the smallest element of the Java program.

For example, consider the following code.

```
public class Demo
{
public static void main(String args[])
{
System.out.println("javatpoint");
}
}
```

- In the above code **public, class, Demo, {, static, void, main, (, String, args, [,],), System, ., out, println, javatpoint**, etc. are the Java tokens.
- The Java compiler translates these tokens into Java bytecode. Further, these bytecodes are executed inside the interpreted Java environment.

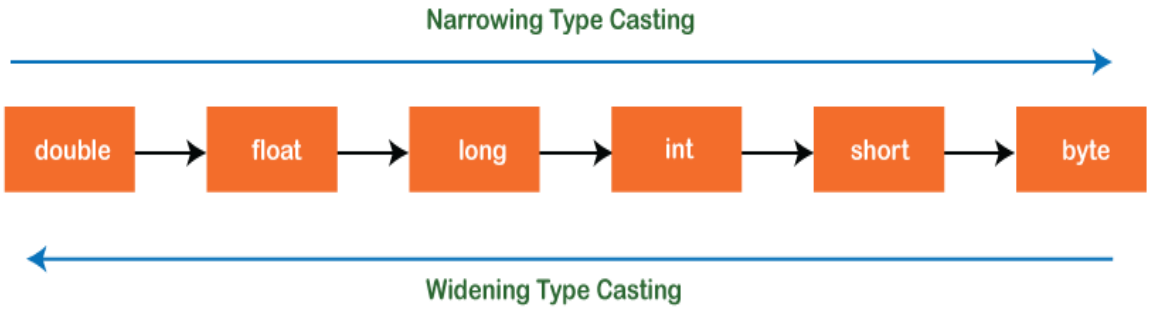
Keywords:

- These are the **pre-defined** reserved words of any programming language.
- It is always written in lower case.
- Java provides the following keywords:

01. abstract	02. boolean	03. byte	04. break	05. class
06. case	07. catch	08. char	09. continue	10. default
11. do	12. double	13. else	14. extends	15. final
16. finally	17. float	18. for	19. if	20. implements
21. import	22. instanceof	23. int	24. interface	25. long
26. native	27. new	28. package	29. private	30. protected
31. public	32. return	33. short	34. static	35. super
36. switch	37. synchronized	38. this	39. thro	40. throws
41. transient	42. try	43. void	44. volatile	45. while
46. assert	47. const	48. enum	49. goto	50. strictfp

Type conversion and Casting:

- In Java, **type casting** is a method or process that **converts** a **data type** into **another data type** in both ways **manually** and **automatically**.
- The **automatic conversion** is done by the **compiler** and **manual conversion** performed by the **programmer**.



Type Casting in Java

Types of Type Casting:

There are two types of type casting:

1. Widening Type Casting
2. Narrowing Type Casting

1. Widening Type Casting:

- Converting a lower data type into a higher one is called **widening** type casting.
- It is also known as **implicit conversion** or **casting down**.
- It is done **automatically**.
- It is safe because there is no chance to lose data.
- Both data types must be compatible with each other.
- The target type must be larger than the source type.

---

byte -> **short** -> **char** -> **int** -> **long** -> **float** -> **double**

Example:

WideningTypeCastingExample.java

```
public class WideningTypeCastingExample
{
public static void main(String[] args)
{
int x = 7;
//automatically converts the integer type into long type
long y = x;
//automatically converts the long type into float type
float z = y;
System.out.println("Before conversion, int value "+x);
System.out.println("After conversion, long value "+y);
System.out.println("After conversion, float value "+z);
}
}
```

Output

```
Before conversion, the value is: 7
After conversion, the long value is: 7
After conversion, the float value is: 7.0
```

- In the above example, we have taken a variable x and converted it into a long type. After that, the long type is converted into the float type.

2. Narrowing Type Casting:

- Converting a higher data type into a lower one is called **narrowing** type casting.
- It is also known as **explicit conversion** or **casting up**.
- It is done **manually** by the **programmer**.

- If we do not perform casting then the compiler reports a compile-time error.

---

double -> float -> long -> int -> char -> short -> byte

Example:

NarrowingTypeCastingExample.java

```
public class NarrowingTypeCastingExample
{
public static void main(String args[])
{
double d = 166.66;
//converting double data type into long data type
long l = (long)d;
//converting long data type into int data type
int i = (int)l;
System.out.println("Before conversion: "+d);
//fractional part lost
System.out.println("After conversion into long type: "+l);
//fractional part lost
System.out.println("After conversion into int type: "+i);
}
}
```

Output

```
Before conversion: 166.66
After conversion into long type: 166
After conversion into int type: 166
```

- we have performed the **narrowing type casting** two times.
- First, we have converted the double type into long data type after that long data type is converted into int type.

Access Modifiers in Java:

- The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class.
- We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

access modifiers: There are **four types** of **Java access modifiers**:

1. **Private**:
    - The access level of a private modifier is only within the class.
    - It cannot be accessed from outside the class.
2. **Default**:
    - The access level of a default modifier is only within the package.
    - It cannot be accessed from outside the package.
    - If you do not specify any access level, it will be the default.
3. **Protected**:
    - The access level of a protected modifier is within the package and outside the package through child class.
    - If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**:
    - The access level of a public modifier is everywhere.
    - It can be accessed from within the class, outside the class, within the package and outside the package.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

- There are many **non-access modifiers**, such as **static**, **abstract**, **synchronized**, **native**, **volatile**, **transient**, etc.

Example:

1. **Private:** The private access modifier is accessible only within the class.

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}

public class Simple{
 public static void main(String args[]){
   A obj=new A();
   System.out.println(obj.data);//Compile Time Error
   obj.msg();//Compile Time Error
   }
}
```

- In this example, we have created two **classes A** and **Simple**.
- A class contains **private data member** and **private method**.
- We are accessing these **private members from outside the class**, so there is a **compile-time error**.

2. Default:

- If you don't use any modifier, it is treated as **default** by default.
- The **default modifier** is accessible **only within package**.
- It **cannot** be accessed from **outside the package**.
- It provides more accessibility than private. but, it is more restrictive than protected, and public.

```
//save by A.java
package pack;
class A{
 void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;
class B{
 public static void main(String args[]){
  A obj = new A();//Compile Time Error
  obj.msg();//Compile Time Error
 }
}
```

- In this example, we have created two packages **pack** and **mypack**.
- We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.
- the scope of **class A** and its **method msg()** is default so it cannot be accessed from outside the package.

3. protected:

- The **protected access modifier** is accessible within package and outside the package but through inheritance only.
- The protected access modifier can be applied on the data member, method and constructor.
- It **can't** be applied on the **class**.
- It provides more accessibility than the **default modifier**.

```
//save by A.java
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;

class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
  }
}
```

```
Output:Hello
```

- In this example, we have created the two packages **pack** and **mypack**.
- The A class of pack package is public, so can be accessed from outside the package.
- But msg() method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

4. **public:** The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

```
//save by A.java

package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java

package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

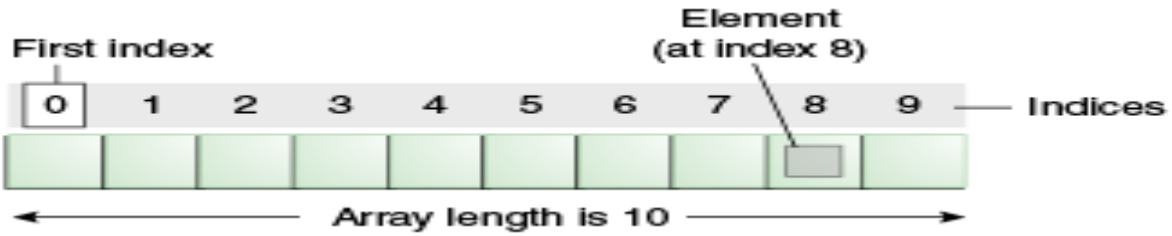
```
Output:Hello
```

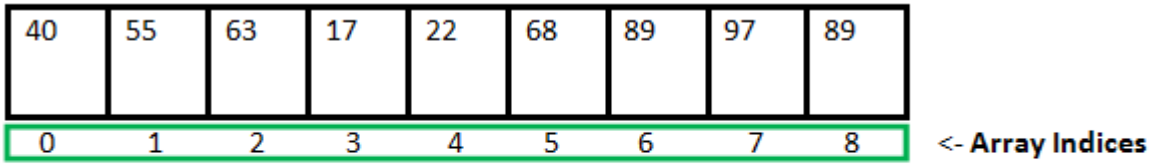
JAVA

UNIT 2: DATA TYPES, VARIABLES, AND ARRAYS

Array:

- ✓ An **array** is a collection of **similar type (Homogeneous)** of elements which has **contiguous memory location.**
- ✓ **Java array** is an **object** which contains elements of a **similar data type**.
- ✓ The elements of an **array** are stored in a **contiguous memory location**.
- ✓ It is a **data structure** where we store **similar elements**.
- ✓ We can store only a fixed set of elements in a Java array.
- ✓ **Array** is **index-based**, the **first element** of the array is stored at the **0th index**, **2nd element** is stored on **1st index**, **last element** is stored on **(n)th-1**, where **n** is **number** of **elements** and so on.
- ✓ we can also create **single dimensional** or **multidimensional** arrays in Java.





Array Length = 9
First Index = 0
Last Index = 8

Advantages:

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
- **Random access:** We can get any data located at an index position.

Disadvantages:

- **Size Limit:** We can store only the fixed size of elements in the array.

- It doesn't grow its size at runtime.
- To solve this problem, collection framework is used in Java which grows automatically.

Declaring Array Variables:

- To use an array in a program, you must declare a **variable** to reference the **array**.
- Here is the **syntax** for **declaring an array variable** –

Syntax:

```
dataType[ ] arrayVariable;
```

or

```
dataType arrayVariable[ ];
```

Creating Arrays:

- You can create an array by using the **new operator** with the following syntax−

Syntax:

```
arrayVariable = new dataType [arraySize];
```

The above statement does two things –

- It creates an array using new dataType[arraySize].
- It assigns the reference of the newly created array to the variable arrayRefVar.

Declaring an array variable, **creating an array**, and **assigning** the reference of the array to the variable can be combined in one statement, as shown below−

```
dataType[ ] arrayVariable = new dataType[arraySize];
```

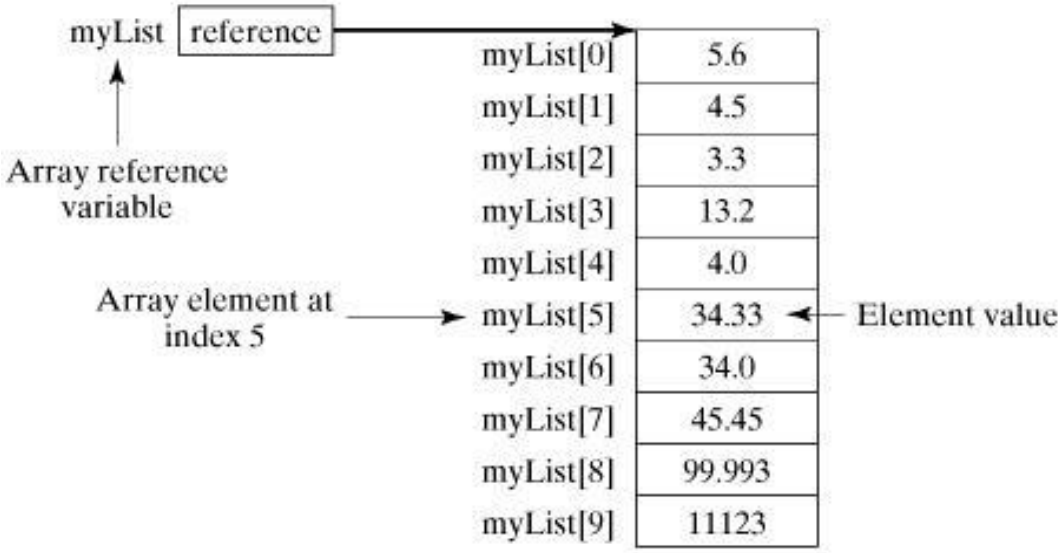
Alternatively, you can create arrays as follows –

```
dataType[ ] arrayVariable = {value1, value2, value3, ….., valueN};
```

The array elements are accessed through the **index**.

Example:

- Following statement declares an **array variable**, **myList**, creates an array of 10 elements of **double type** and assigns its reference to **myList** –
- Following picture represents array **myList**. Here, **myList** holds 10 double values and the **index** are **from 0 to 9**.



Types of Array in java:

There are two types of array.

1. Single Dimensional Array
2. Multidimensional Array

1. Single Dimensional Array in Java

Syntax to Declare an Array in Java

```
dataType[] arr; (or)
dataType []arr; (or)
dataType arr[];
```

Instantiation of an Array in Java

```
arrayRefVar=new datatype[size];
```

Example: Program

- Let's see the simple example of java array, where we are going to **declare, instantiate, initialize** and **traverse an array.**

```
//Java Program to illustrate how to declare, instantiate, initialize
//and traverse the Java array.
class Testarray{
public static void main(String args[]){
int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
//traversing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

Output:

10

20

70

40

50

Declaration, Instantiation and Initialization of Java Array:

- We can declare, instantiate and initialize the java array together by:
- `int a[ ]={33,3,4,5};//declaration, instantiation and initialization`

Example:

```
/*Java Program to illustrate the use of declaration, instantiation and initializat
ion of Java array in a single line */
class Testarray
{
        public static void main(String args[])
        {
        int a[ ]={33,3,4,5}; //declaration, instantiation and initialization
        for(int i=0; i<a.length; i++)//length is the property of array
        System.out.println(a[i]);   //printing array
        }
}
```

Output:

```
33
3
4
5
```

2. Multidimensional Array in Java

- In such case, data is stored in **row** and **column** based index (also known as matrix form).

Syntax to Declare Multidimensional Array in Java:

```
dataType[][] arrayRefVar; (or)
dataType [][]arrayRefVar; (or)
dataType arrayRefVar[][]; (or)
dataType []arrayRefVar[];
```

Example to instantiate Multidimensional Array in Java:

```
int[][] arr=new int[3][3]; // 3 row and 3 column
```

Example to initialize Multidimensional Array in Java:

```
arr[0][0] = 1;
arr[0][1] = 2;
arr[0][2] = 3;
arr[1][0] = 4;
arr[1][1] = 5;
arr[1][2] = 6;
arr[2][0] = 7;
arr[2][1] = 8;
arr[2][2] = 9;
```

Example of Multidimensional Java Array:

- Let's see the simple example to **declare, instantiate, initialize** and **print the 2Dimensional array.**

```
// Multidimensional Array
class Testarray
{
        public static void main(String args[])
        {
        //declaring and initializing 2D array
        int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
        //printing 2D array
        for(int i=0;i<3;i++)
        {
                for(int j=0;j<3;j++)
                {
                 System.out.print(arr[i][j]+" ");
                }
                 System.out.println();
        }
        }
}
```

Output:

```
1  2  3
2  4  5
4  4  5
```