Marwadi
education foundation

Unit – 3
Methods and Arrays

Prepared By
Prof. Ravikumar Natarajan
Assistant Professor, CE Dept.

Method in Java

A Java method is a collection of statements that are grouped together to perform an operation. When you call the System.out.println() method, for example, the system actually executes several statements in order to display a message on the console.

Now we will learn how to create your own methods with or without return values, invoke a method with or without parameters, and apply method abstraction in the program design.

Creating method

Considering the following example to explain the syntax of a method –

Syntax:

```
modifier returnType nameOfMethod (Parameter List) {
// method body
}
```

Methods in Java

The syntax shown above includes –

- **modifier** – It defines the access type of the method and it is optional to use.
- **returnType** – Method may return a value.
- **nameOfMethod** – This is the method name. The method signature consists of the method name and the parameter list.
- **Parameter List** – The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
- **method body** – The method body defines what the method does with the statements.

Example

```
public static int minFunction(int n1, int n2) {
int min;
  if (n1 > n2)
      min = n2;
  else
      min = n1;
  return min;
}
```

Method Calling

For using a method, it should be called. There are two ways in which a method is called i.e., method returns a value or returning nothing (no return value).

The process of method calling is simple. When a program invokes a method, the program control gets transferred to the called method. This called method then returns control to the caller in two conditions, when –

✓ the return statement is executed.
✓ it reaches the method ending closing brace.

The methods returning void is considered as call to a statement. Lets consider an example − System.out.println("This is method!");

The method returning value can be understood by the following example –

int result = sum(6, 9);

Examples

Check Programs → FunctionWithReturn.java ,
FunctionWithoutReturn.java

FunctionWithReturn

```
//A method with a return value
public class Main {
 static int myMethod(int x) {
   return 5 + x;
 }
 public static void main(String[] args) {
   System.out.println(myMethod(3));
 }
}
```

```
public class ReturnTypeTest1 {
  public int add() { // without arguments
    int x = 30;      int y = 70;      int z = x+y;
    return z;
  }
  public static void main(String args[]) {
    ReturnTypeTest1 test = new ReturnTypeTest1();
    int add = test.add();
    System.out.println("The sum of x and y is: " + add);
  }
}
```

```
public class ReturnTypeTest2 {
  public int add(int x, int y) { // with arguments
    int z = x+y;
    return z;
  }
  public static void main(String args[]) {
    ReturnTypeTest2 test = new ReturnTypeTest2();
    int add = test.add(10, 20);
    System.out.println("The sum of x and y is: " + add);
  }
}
```

FunctionWithoutReturn

```
//A method without any return values:
public class Main {
  static void myMethod() {
    System.out.println("I just got executed!");
  }

  public static void main(String[] args) {
    myMethod();
  }
}
```

```
// Java Program to Illustrate a Method
// with 2 Parameters and without Return Type
import java.util.*;
public class Main {
          public static void main(String args[])
          {
                    int a = 4;
                    int b = 5;

                    // Calling the function with 2 parameters
                    calc(a, b);
          }
          public static void calc(int x, int y)
          {
                    int sum = x + y;
                    // Displaying the sum
                    System.out.print("Sum of two numbers is :" + sum);
          }
}
```

Parameter Passing in Java

There are different ways in which parameter data can be passed into and out of methods and functions. Let us assume that a function B() is called from another function A(). In this case A is called the "caller function" and B is called the "called function or callee function". Also, the arguments which A sends to B are called actual arguments and the parameters of B are called formal arguments.

Types of parameters:

Formal Parameter : A variable and its type as they appear in the prototype of the function or method.

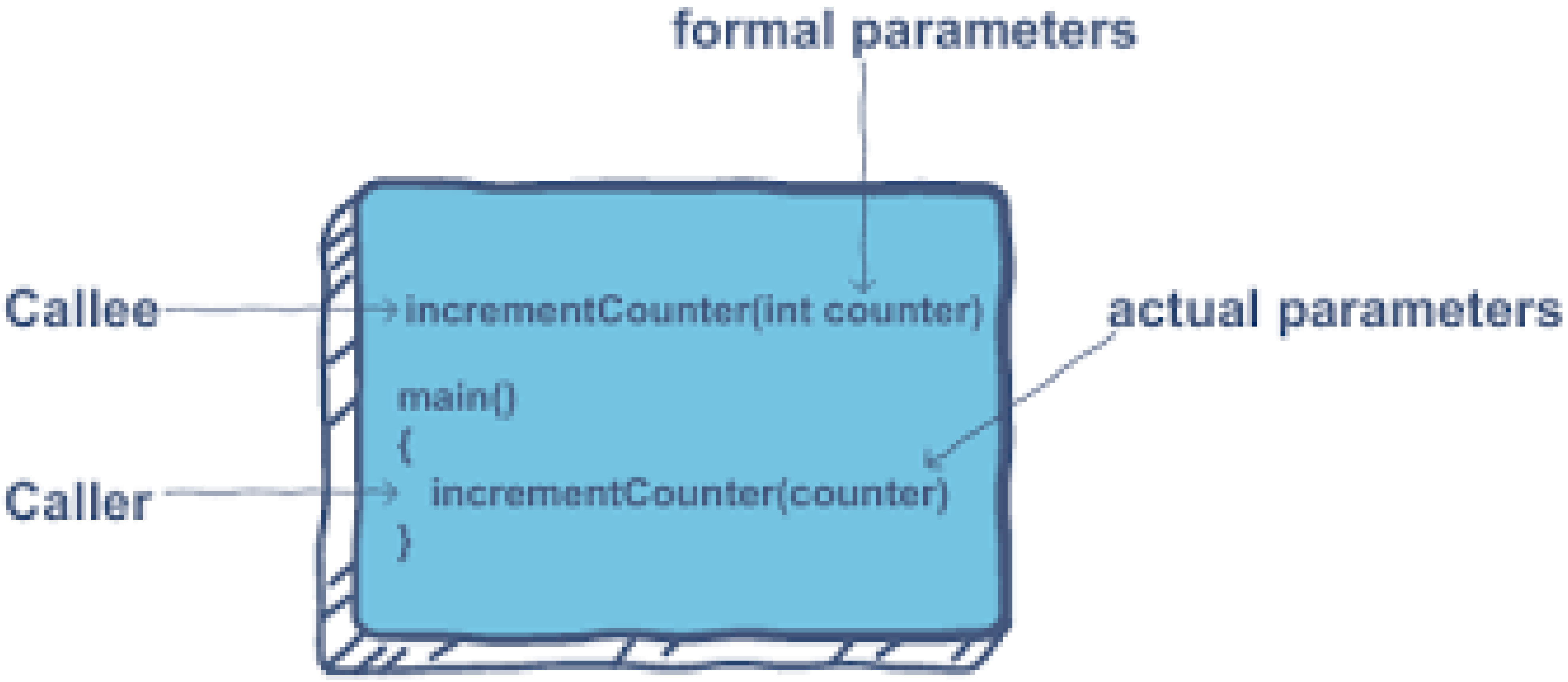
Syntax:

```
function_name(datatype variable_name)
```

Actual Parameter : The variable or expression corresponding to a formal parameter that appears in the function or method call in the calling environment.

Syntax:

```
func_name(variable name(s));
```

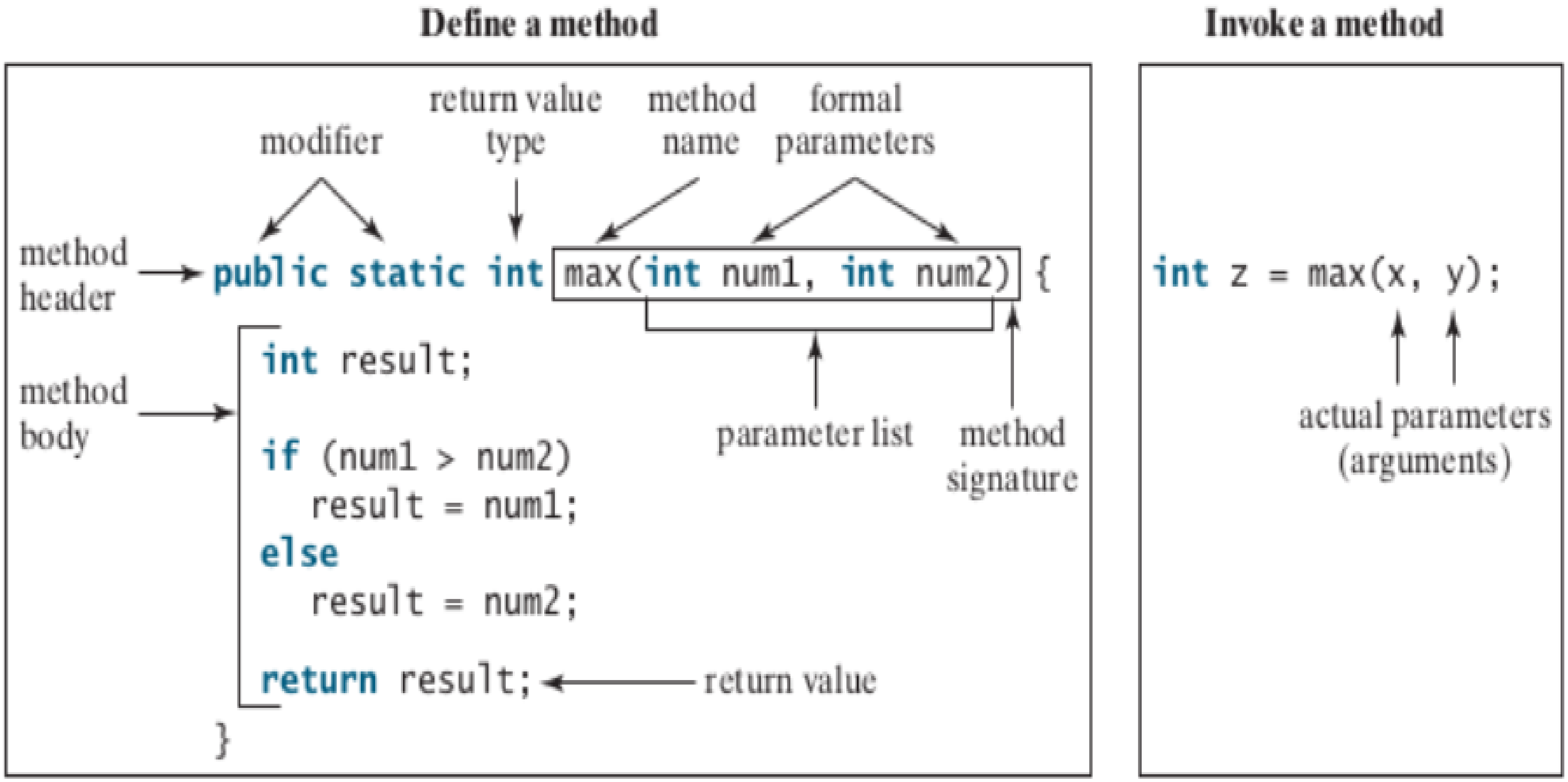

formal parameters
Callee
incrementCounter(int counter)
actual parameters
main()
{
Caller
incrementCounter(counter)
}

```
class Summation
{
        int x,y,z;
        void sum(int a, int b)
        {
                x=a;
                y=b;
                z=x+y;
                System.out.println (z);
        }
        public static void main(String args[])
        {
                Summation ob=new
Summation();
                Int m,n;
                m=20;
                n=30;
                ob.sum(m,n);
        }
}
```

Formal Parameter

Actual Parameter

Define and Invoke Method in Java



```
public static int max(int num1, int num2) {
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

```
int z = max(x, y);
```

Parameter Passing in Java

parameter passing in Java is always Pass-by-Value. However, the context changes depending upon whether we're dealing with Primitives or Objects:

- ✓ For Primitive types, parameters are pass-by-value
- ✓ For Object types, the object reference is pass-by-value

In case of primitives, the value is simply copied inside stack memory which is then passed to the callee method.

In case of non-primitives, a reference in stack memory points to the actual data which resides in the heap. When we pass an object, the reference in stack memory is copied and the new reference is passed to the method.

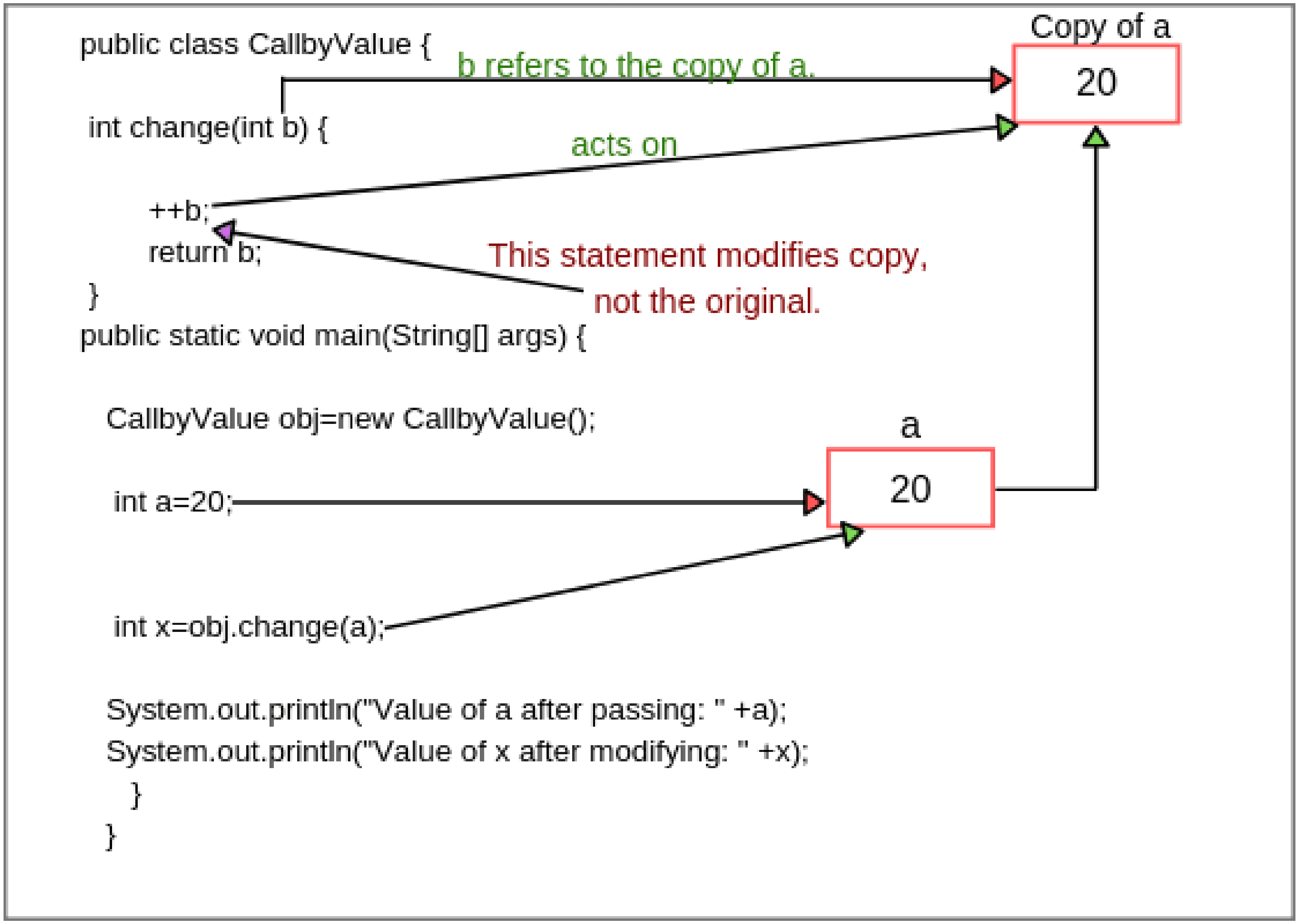
Check Example : Unit 3 → CallByValue.java , CallByReference.java

Pass By Value / Call by Value

- "Call by value" in Java means that argument's value is copied and is passed to the parameter list of a method.
- That is, when we call a method with passing argument values to the parameter list, these argument values are copied into the small portion of memory and a copy of each value is passed to the parameters of the called method.
- When these values are used inside the method either for "read or write operations", we are actually using the copy of these values, not the original argument values which are unaffected by the operation inside the method.
- That is, the values of the parameters can be modified only inside the scope of the method but such modification inside the method doesn't affect the original passing argument.
- When the method returns, the parameters are gone and any changes to them are lost. This whole mechanism is called call by value or pass by value.

Check Example : Unit 3 → CallByValue.java

```
//CallByValue
public class Main
{
  int change(int b)
  {
    ++b; // Changes will be in the local variable only.
    return b;
  }
public static void main(String[] args)
{
// Create an object of class.
    Main obj = new Main();
     int a = 20;
     int x = obj.change(a);
     System.out.println("Value of a after passing: " +a);
     System.out.println("Value of x after modifying: " +x);
 }
}
```



```
public class CallbyValue {

 int change(int b) {

       ++b;
       return b;
 }
public static void main(String[] args) {

   CallbyValue obj=new CallbyValue();

    int a=20;


    int x=obj.change(a);

   System.out.println("Value of a after passing: " +a);
   System.out.println("Value of x after modifying: " +x);
     }
  }
```

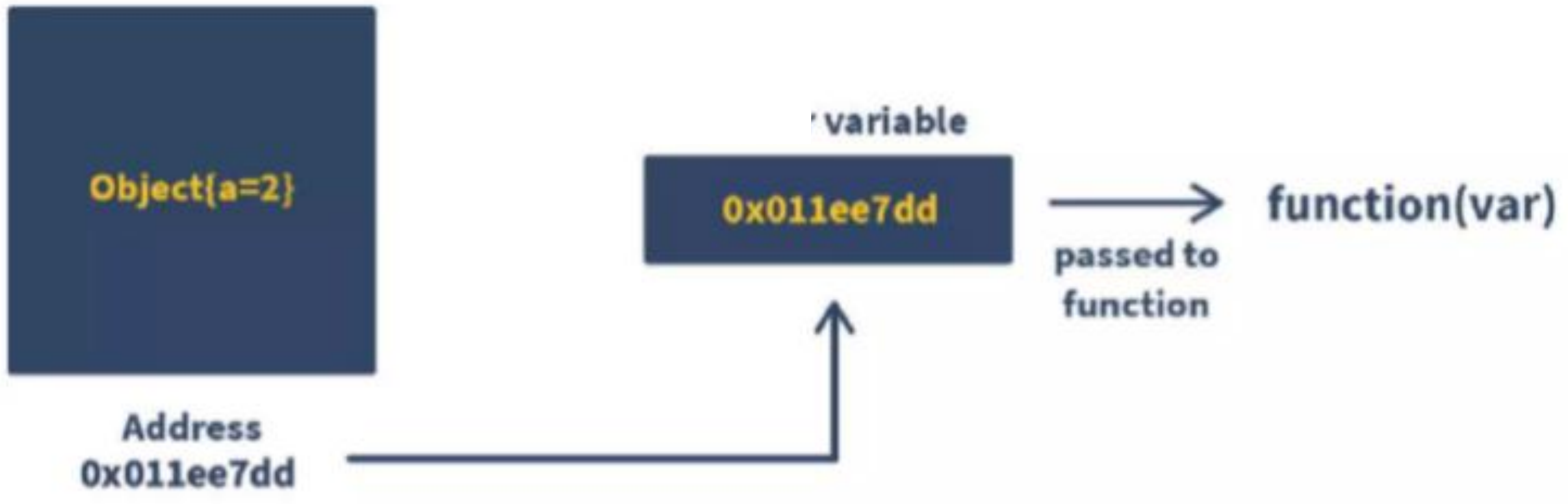
Copy of a
20
b refers to the copy of a.
acts on
This statement modifies copy, not the original.
a
20

Call by reference(aliasing)

- Call by Reference means calling a method with a parameter as a reference. Through this, the argument reference is passed to the parameter.
- In call by value, the modification done to the parameter passed does not reflect in the caller's scope while in the call by reference, the modification done to the parameter passed are persistent and changes are reflected in the caller's scope.

Check Example : Unit 3 → CallByReference.java

Call by reference(aliasing)



Call By Value vs Call By Reference

```
public class Main{
//call-by-value
   int a = 10;
   void call(int a) {

            a = a+10;
   }
    public static void main(String[] args) {
       Main eg = new Main();
      S.o.p("Before call-by-value: " + eg.a);
       eg.call(50510);
      S.o.p("After call-by-value: " + eg.a);
   }
}
```

```
public class Main{
   int a = 10;
   void call(Main e) {
      e.a = e.a+10;
   }
   public static void main(String[] args) {

      Main e = new Main();
     S.o.p("Before call-by-reference: " + e.a);
     e.call(e);
     S.o.p("After call-by-reference: " + e.a);



   }
}
```

Overloading methods

- In Java, two or more methods can have same name if they differ in parameters (different number of parameters, different types of parameters, or both). These methods are called overloaded methods and this feature is called method overloading. For example:

- void func() { ... }
- void func(int a   { ... }
- float func(double a) { ... } float func(int a, float b) { ... }

- Here, the func() method is overloaded. These methods have the same name but accept different arguments.
- Notice that, the return type of these methods is not the same. Overloaded methods may or may not have different return types, but they must differ in parameters they accept.

Why method overloading ?

- Suppose, you have to perform the addition of given numbers but there can be any number of arguments (let’s say either 2 or 3 arguments for simplicity).

- In order to accomplish the task, you can create two methods : sum2num(int, int) and
- sum3num(int, int, int) for two and three parameters respectively.

- However, other programmers, as well as you in the future may get confused as the behavior of both methods are the same but they differ by name.

- The better way to accomplish this task is by overloading methods. And, depending upon the argument passed, one of the overloaded methods is called. This helps to increase the readability of the program.

Method overloading

Marwadi education foundation

Two or more methods can have same name inside the same class if they accept different arguments. This feature is known as method overloading.

Method overloading is achieved by either:

- ✓ changing the number of arguments.
  (**Check Program:** Unit – 3 → MethodOverloading1.java)

- ✓ changing the datatype of arguments.
- ✓ (**Check Program:** Unit – 3 → MethodOverloading2.java)

Method overloading is not possible by changing the return type of methods.

Method overloading

```
//Overloading by changing the number of parameters
class MethodOverloading {
   private static void display(int a){
      System.out.println("Arguments: " + a);
   }

   private static void display(int a, int b){
      System.out.println("Arguments: " + a + " and " + b);
   }

   public static void main(String[] args) {
      display(1);
      display(1, 4);
   }
}
```

```
//changing the data type of parameters
class MethodOverloading2 {

   // this method accepts int
   private static void display(int a){
      System.out.println("Got Integer data.");
   }

   // this method  accepts String object
   private static void display(String a){
      System.out.println("Got String object.");
   }

   public static void main(String[] args) {
      display(1);
      display("Hello");
   }
}
```

Can we overload java main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But JVM calls main() method which receives string array as arguments only. Let's see the simple example:

```
class TestOverloading4{
 public static void main(String[] args)
        {System.out.println("main with String[]");}
 public static void main(String args)
        {System.out.println("main with String");}
 public static void main()
        {System.out.println("main without args");}
}
```

Output:

```
main with String[]
```

Method Overloading

Why Method Overloading is not possible by changing the return type of method only?

In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

```
class Adder{
static int add(int a,int b){return a+b;}
static double add(int a,int b){return a+b;}
}
class TestOverloading3{
public static void main(String[] args){
System.out.println(Adder.add(11,11));//ambiguity
}}
```

OUTPUT :

```
Compile Time Error: method add(int,int) is already defined in class Adder
```

Scope of Variables

- The scope of a variable is the part of the program where the variable can be referenced.
- A variable defined inside a method is referred to as a local variable. The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable.

```
public class Main{
public static void main(String[] args) {
 for(int i=0; i<10; i++){
System.out.println(i);              }
 System.out.println(i); //Out of scope
}}
```

Method Abstraction and Stepwise Refinement

- Method abstraction is achieved by separating the use of a method from its implementation. The client can use a method without knowing how it is implemented.
- The details of the implementation are encapsulated in the method and hidden from the client who invokes the method. This is also known as information hiding or encapsulation.
- Ex. Println(), max() all in-built methods ,you only know how to invoke this method, but you are not required to know how they are implemented.
- Method abstraction can be applied to the process of developing programs.
- When writing a large program, you can use the divide-and-conquer strategy, also known as stepwise refinement, to decompose it into sub-problems. The sub-problems can be further decomposed into smaller, more manageable problems.

Method Abstraction and Stepwise Refinement

- Top-Down Design:
    - Problem is broken into sub-problems.
  - For Ex. To calculate area of circle, first read the input, then implement the actual logic and then print the answer.

  Top-Down Vs Bottom-up Implementation:

  - Top down approach begins with high level design and ends with low level design or development. Whereas, bottom up approach begins with low level design or development and ends with high level design.
  - In top down approach, main() function is written first and all sub functions are called from main function. Then, sub functions are written based on the requirement. Whereas, in bottom up approach, code is developed for modules and then these modules are integrated with main() function.

Method Abstraction and Stepwise Refinement

- OOPs programming language is Top-Down or Bottom-Up?
- OOPs follow Bottom-Up approach

Benefits of stepwise refinement

- ► Each sub-problem can be implemented using a method. This approach makes the program easier to write, reuse, debug, test, modify, and maintain.

1. Simpler program:
2. Reusing Methods:
3. Easier Developing, Debugging and Testing:
4. Better facilitating teamwork:

Single Dimensional Array

- A single array variable can reference a large collection of data.

- Once an array is created, its size is fixed. An array reference variable is used to access the elements in an array using an index.

- Instead of declaring individual variables, such as number0, number1, . . . , and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], . . . , and numbers[99] to represent individual variables.

Declaring 1-D Array Variables

- Syntax:

  datatype[ ] arr_ref_var;

  OR

  datatype arr_ref_var[ ];

- Ex. double myList[ ];
- Unlike declarations for primitive data type variables, the declaration of an array variable does not allocate any space in memory for the array. It creates only a storage location for the reference to an array.
- you can create an array by using the new operator and assign its reference to the variable.

  Syntax:

  Arr_ref_var = new datatype [size];

Declaring 1-D Array Variables

- Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement as:
  - datatype[ ] arr_ref_var = new datatype [Size];

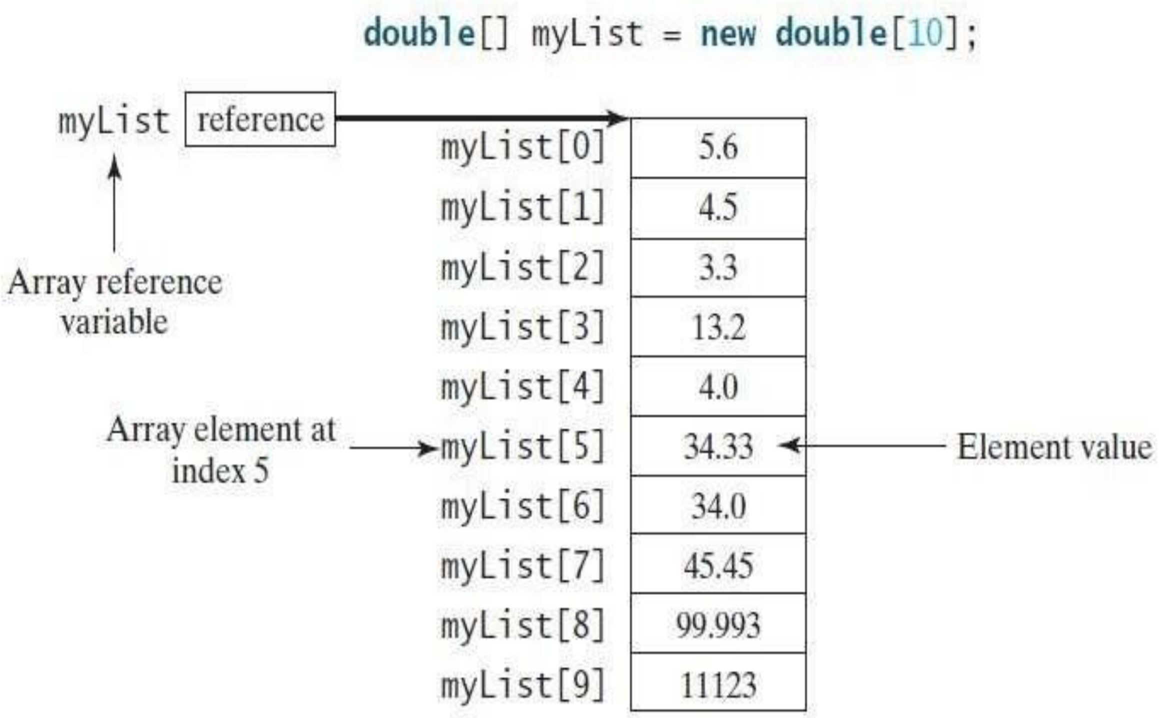
OR

  - datatype arr_ref_var[ ] = new datatype [Size];

- Ex. **double**[] myList = **new double**[**10**];

Assign values to Array

- Ex.
  - myList[**0**] = **5.6**;
  - myList[**1**] = **4.5**;
  - myList[**2**] = **3.3**;
  - myList[**3**] = **13.2**;
  - myList[**4**] = **4.0**;
  - myList[**5**] = **34.33**;
  - myList[**6**] = **34.0**;
  - myList[**7**] = **45.45**;
  - myList[**8**] = **99.993**;
  - myList[**9**] = **11123**;



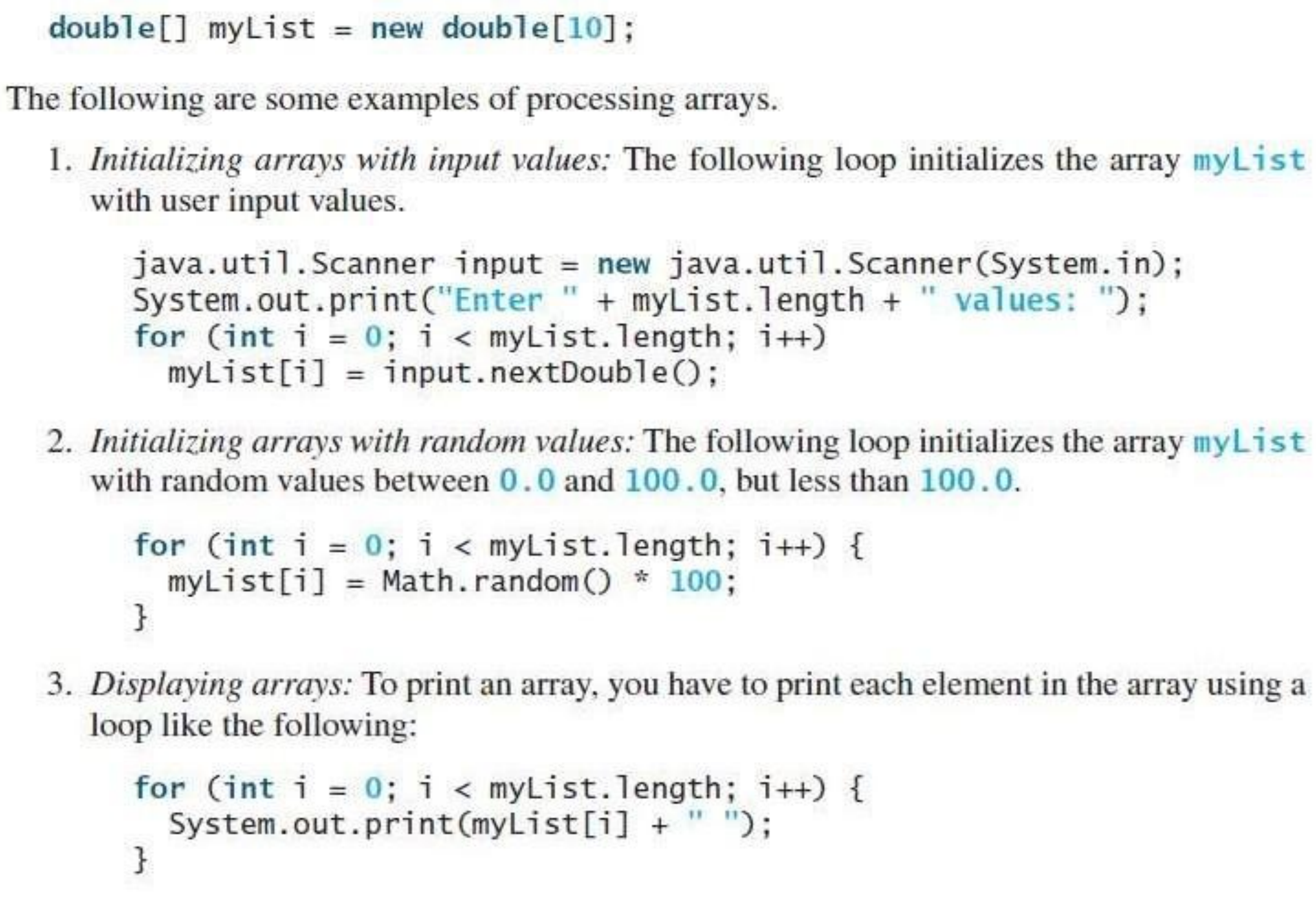
Array size and default values

- The size of an array cannot be changed after the array is created.
- Size can be obtained using arr_ref_var**.length**.
- For example, myList.length is 10.
- myList holds 10 double values, and the indices are from **0 to 9**.
- For example, myList[9] represents the **last** element in the array myList.
- Array Initializer:

  Syntax: elementType[] arrayRefVar = {value0, value1, ...,

  value*k*}; Example: double[] myList = {1.9, 2.9, 3.4, 3.5};

Processing Array



```
double[] myList = new double[10];
```

The following are some examples of processing arrays.

1. *Initializing arrays with input values:* The following loop initializes the array `myList` with user input values.

```
java.util.Scanner input = new java.util.Scanner(System.in);
System.out.print("Enter " + myList.length + " values: ");
for (int i = 0; i < myList.length; i++)
  myList[i] = input.nextDouble();
```

2. *Initializing arrays with random values:* The following loop initializes the array `myList` with random values between `0.0` and `100.0`, but less than `100.0`.

```
for (int i = 0; i < myList.length; i++) {
  myList[i] = Math.random() * 100;
}
```

3. *Displaying arrays:* To print an array, you have to print each element in the array using a loop like the following:

```
for (int i = 0; i < myList.length; i++) {
  System.out.print(myList[i] + " ");
}
```

1D – Array Example

```
public class Array1D{
     public static void main(String args[])
       {
               double[ ] mylist = new double[10]; // Array declaration
                java.util.Scanner sc = new java.util.Scanner(System.in);

               System.out.println("Enter  " + mylist.length + " values ");
               //intialization array with input values
               for(int i=0;i<mylist.length;i++)
                     mylist[i] = sc.nextInt();

               //Display Array
               for(int i=0;i<mylist.length;i++)
                     System.out.print((int)mylist[i] + " ");



       }
}
```

For each Loop

- **For** Java supports a convenient for loop, known as a foreach loop, which enables you to traverse the array sequentially without using an index variable.
- Syntax:

```
for (elementType element: arrayRefVar)
{
    // Process the element
}
```

- You can read the code as "for each element e in myList, do the following." If you wish to traverse in different order then you need index variable.

Ex.

```
for(double e : myList)
{
    System.out.println(e);
}
```

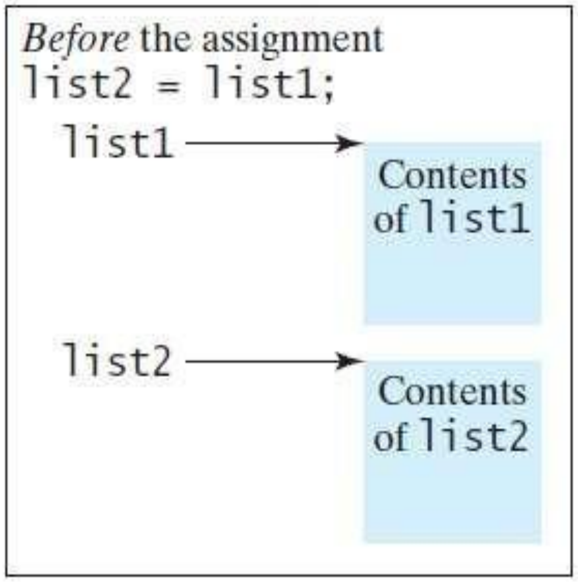
Copying Array

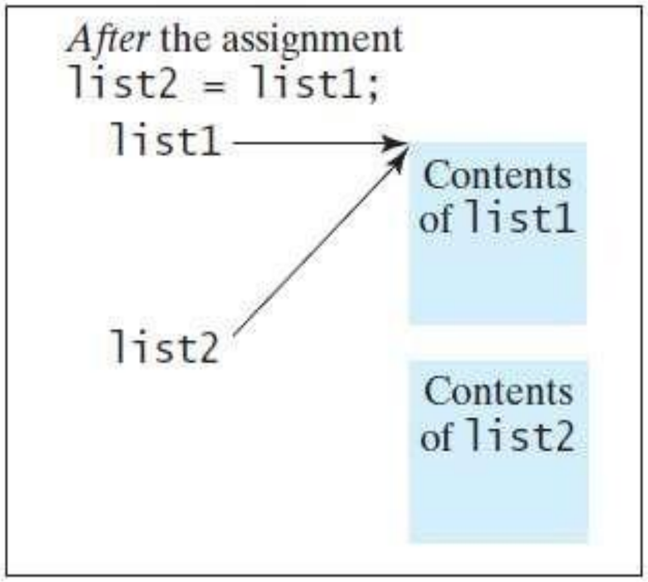
- To copy the contents of one array into another, you have to copy the array's individual elements into the other array.

Method : 1

list2 = list1;

- This statement does not copy the contents of the array referenced by list1 to list2, but instead it copies the reference value from **list1** to **list2**.





Copying Array

- The other 3 ways to copy Array:

  Method 2: Use a loop to copy individual elements one by one.

  Method 3: Use the static arraycopy method in the System class.

  Method 4: Use the clone method to copy arrays

Method 2: Loop to copy individual element

```
int[] sourceArray = {2, 3, 1, 5, 10};
int[] targetArray = new int[sourceArray . length];
for (int i = 0; i < sourceArray.length; i++)
{
        targetArray[i] = sourceArray[i];
}
```

Copying Array – Method 1

```
class Main {
   public static void main(String[] args) {

      int [] numbers = {1, 2, 3, 4, 5, 6};
      int [] positiveNumbers = numbers;    // copying arrays

      // change value of first array
      numbers[0] = -1;

      // printing the second array
      for (int number: positiveNumbers) {
         System.out.print(number + ", ");
      }
   }
}
```

Copying Array – Method 2

```
import java.util.Arrays;

class Main {
    public static void main(String[] args) {

        int [] source = {1, 2, 3, 4, 5, 6};
        int [] destination = new int[6];

        // iterate and copy elements from source to destination
        for (int i = 0; i < source.length; ++i) {
            destination[i] = source[i];
        }

         // converting array to string
        System.out.println(Arrays.toString(destination));
    }
}
```

Copying Array – Method 3

```
//arraycopy
import java.util.Arrays;

class Main {
    public static void main(String[] args) {
        int[] n1 = {2, 3, 12, 4, 12, -2};

        // Creating n2 array of having length of n1 array
        int[] n2 = new int[n1.length];

        // copying entire n1 array to n2
        System.arraycopy(n1, 0, n2, 0, n1.length);
        System.out.println("n2 = " + Arrays.toString(n2));

    }
}
```

Copying Array

Method 3: Another approach is to use the **arraycopy** method in the **java.lang.System** class to copy arrays instead of using a loop.

Syntax: arraycopy(sourceArray, srcPos, targetArray, tarPos, length);

Ex: System.arraycopy(sourceArray, **0**, targetArray, **0**, sourceArray.length);

Method 4: Another approach is to use the **clone** method in the

Copying Array – Method 4

```
class Main {
    public static void main(String args[])     {
    int num_Array[] = {5,10,15,20,25,30};
     int clone_Array[] = num_Array.clone();

   System.out.println("Original num_Array:");
   for (int i = 0; i <num_Array.length; i++) {
          System.out.print(num_Array[i]+" ");
      }
      System.out.println();

     System.out.println("Cloned num_Array:");
     for (int i = 0; i <clone_Array.length; i++) {
          System.out.print(clone_Array[i]+" ");
      }
    }
}
```

Two-Dimensional Array

- Data in a table or a matrix can be represented using a two- dimensional array.
- An element in a two-dimensional array is accessed through a row and column index.
- Syntax:

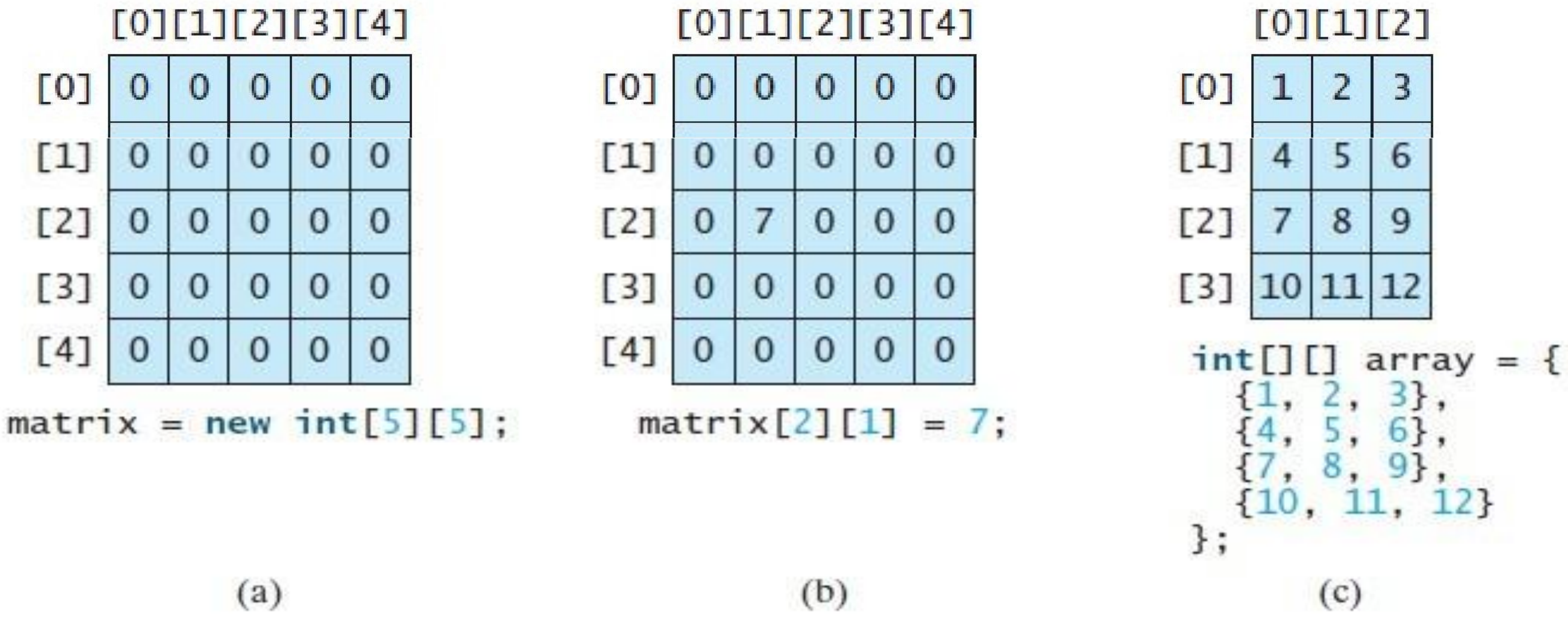
```
elementType[ ][ ] arrayRefVar;
                OR
elementType arrayRefVar[ ][ ];
```

- Ex:

```
int[ ][ ] matrix;
        OR
int matrix[ ][ ];
```

Two-Dimensional Array

matrix = new int[5][5];



```
matrix = new int[5][5];
```

(a)

```
matrix[2][1] = 7;
```

(b)

```
int[][] array = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9},
  {10, 11, 12}
};
```

(c)

	[0]	[1]	[2]	[3]	[4]
[0]	0	0	0	0	0
[1]	0	0	0	0	0
[2]	0	0	0	0	0
[3]	0	0	0	0	0
[4]	0	0	0	0	0

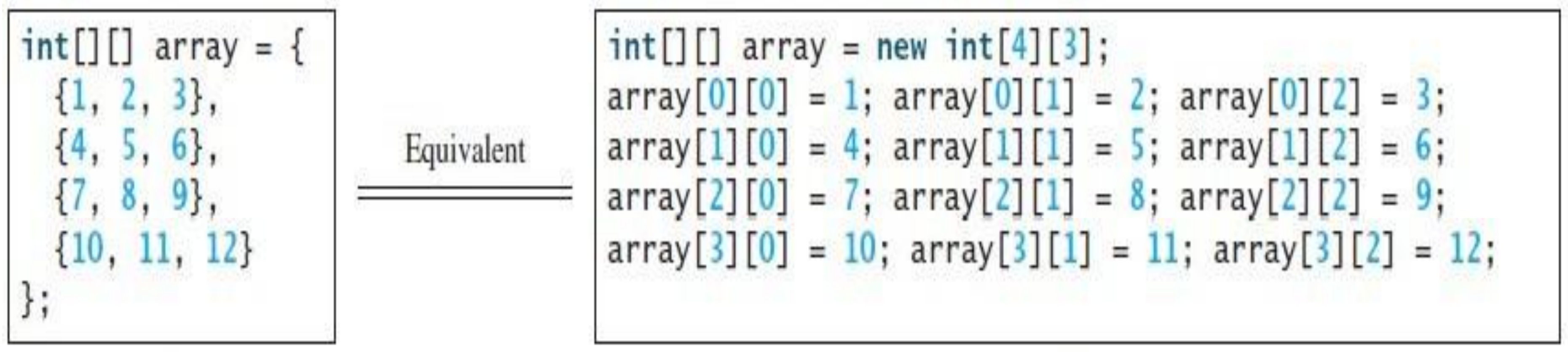
	[0]	[1]	[2]	[3]	[4]
[0]	0	0	0	0	0
[1]	0	0	0	0	0
[2]	0	7	0	0	0
[3]	0	0	0	0	0
[4]	0	0	0	0	0

	[0]	[1]	[2]
[0]	1	2	3
[1]	4	5	6
[2]	7	8	9
[3]	10	11	12

Two-Dimensional Array

```
class Main
{
        public static void main(String args[])
        {
        int[][] a={{10,20},{30,40}};//declaration and initialization
        System.out.println("Two dimensional array elements are");
        System.out.println(a[0][0]);
        System.out.println(a[0][1]);
        System.out.println(a[1][0]);
        System.out.println(a[1][1]);
        }
}
```

Two-Dimensional Array



```
int[][] array = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9},
  {10, 11, 12}
};
```

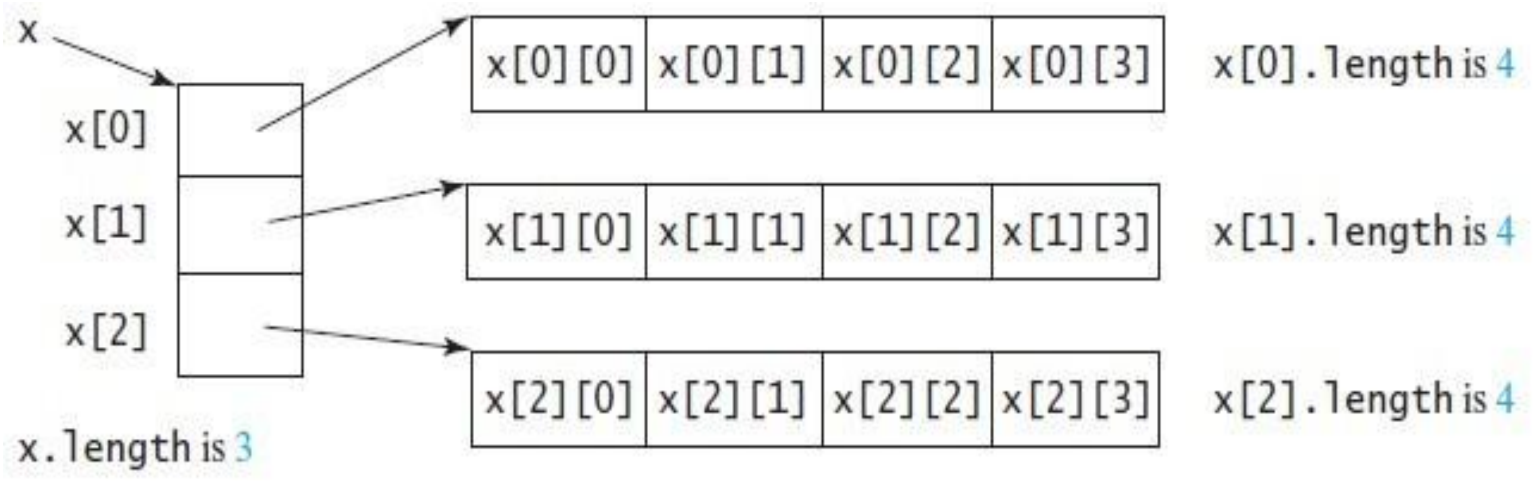
Equivalent

```
int[][] array = new int[4][3];
array[0][0] = 1; array[0][1] = 2; array[0][2] = 3;
array[1][0] = 4; array[1][1] = 5; array[1][2] = 6;
array[2][0] = 7; array[2][1] = 8; array[2][2] = 9;
array[3][0] = 10; array[3][1] = 11; array[3][2] = 12;
```

(a) (b)

Obtaining length of 2-D Array

Example:
x = new int[3][4], x[0], x[1], and x[2] are one-dimensional arrays and each contains four elements. So, x.length is 3, and x[0].length, x[1].length, and x[2].length are 4.


x
x[0]
x[1]
x[2]
x.length is 3
x[0][0] x[0][1] x[0][2] x[0][3]
x[1][0] x[1][1] x[1][2] x[1][3]
x[2][0] x[2][1] x[2][2] x[2][3]
x[0].length is 4
x[1].length is 4
x[2].length is 4

Processing 2-D Arrays

```
int[][] matrix = new int[10][10];
```

The following are some examples of processing two-dimensional arrays.

1. *Initializing arrays with input values.* The following loop initializes the array with user input values:

```
java.util.Scanner input = new Scanner(System.in);
System.out.println("Enter " + matrix.length + " rows and " +
  matrix[0].length + " columns: ");
for (int row = 0; row < matrix.length; row++) {
  for (int column = 0; column < matrix[row].length; column++) {
    matrix[row][column] = input.nextInt();
  }
}
```

2. *Initializing arrays with random values.* The following loop initializes the array with random values between 0 and 99:

```
for (int row = 0; row < matrix.length; row++) {
  for (int column = 0; column < matrix[row].length; column++) {
    matrix[row][column] = (int)(Math.random() * 100);
  }
}
```

Processing 2-D Arrays

3. *Printing arrays.* To print a two-dimensional array, you have to print each element in the array using a loop like the following:

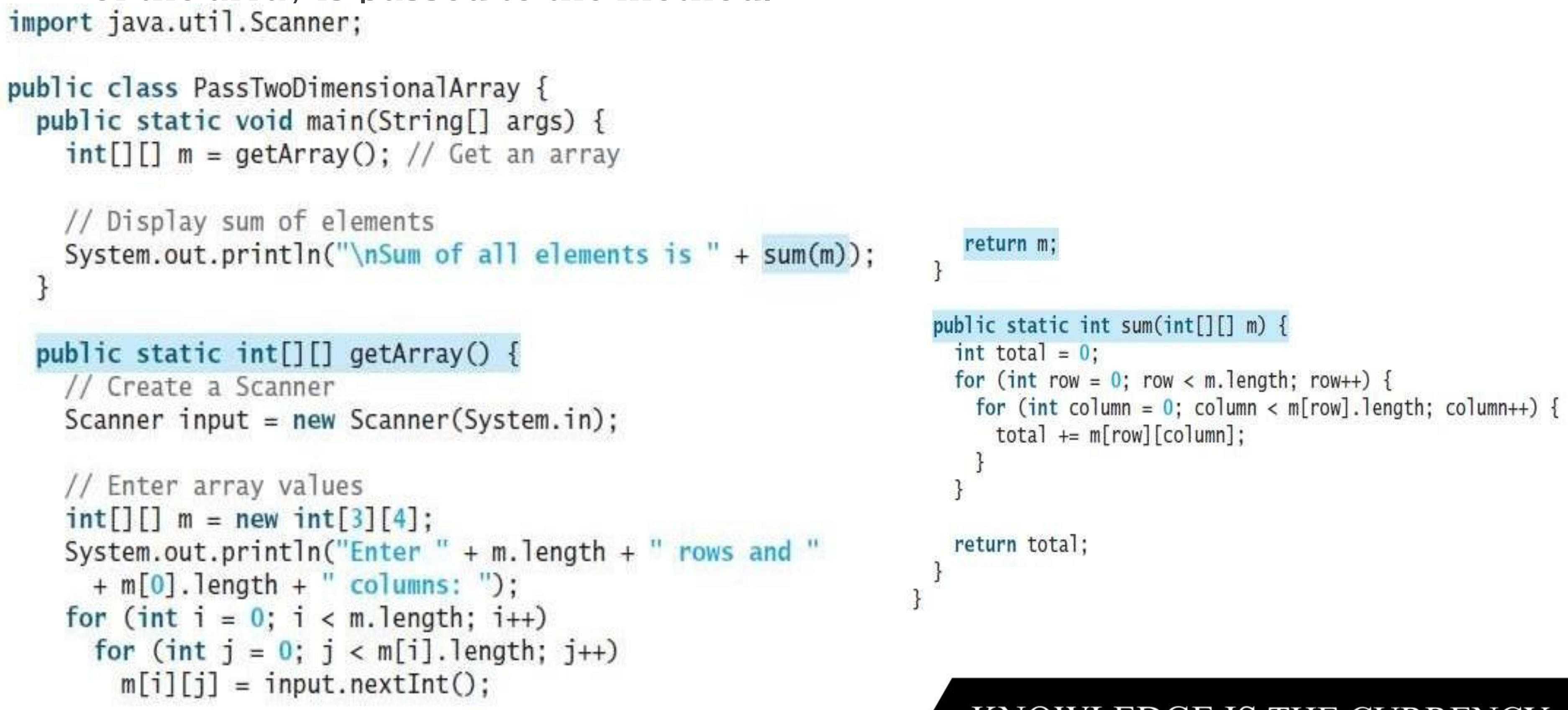
```
for (int row = 0; row < matrix.length; row++) {
  for (int column = 0; column < matrix[row].length; column++) {
    System.out.print(matrix[row][column] + " ");
  }

  System.out.println();
}
```

4. *Summing all elements.* Use a variable named `total` to store the sum. Initially `total` is `0`. Add each element in the array to `total` using a loop like this:

```
int total = 0;
for (int row = 0; row < matrix.length; row++) {
  for (int column = 0; column < matrix[row].length; column++) {
    total += matrix[row][column];
  }
}
```

Passing 2-D Array to methods

When passing a two-dimensional array to a method, the reference of the array is passed to the method.



```
import java.util.Scanner;

public class PassTwoDimensionalArray {
  public static void main(String[] args) {
    int[][] m = getArray(); // Get an array

    // Display sum of elements
    System.out.println("\nSum of all elements is " + sum(m));
  }

  public static int[][] getArray() {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Enter array values
    int[][] m = new int[3][4];
    System.out.println("Enter " + m.length + " rows and "
      + m[0].length + " columns: ");
    for (int i = 0; i < m.length; i++)
      for (int j = 0; j < m[i].length; j++)
        m[i][j] = input.nextInt();

    return m;
  }

  public static int sum(int[][] m) {
    int total = 0;
    for (int row = 0; row < m.length; row++) {
      for (int column = 0; column < m[row].length; column++) {
        total += m[row][column];
      }
    }

    return total;
  }
}
```

2-D Array Example

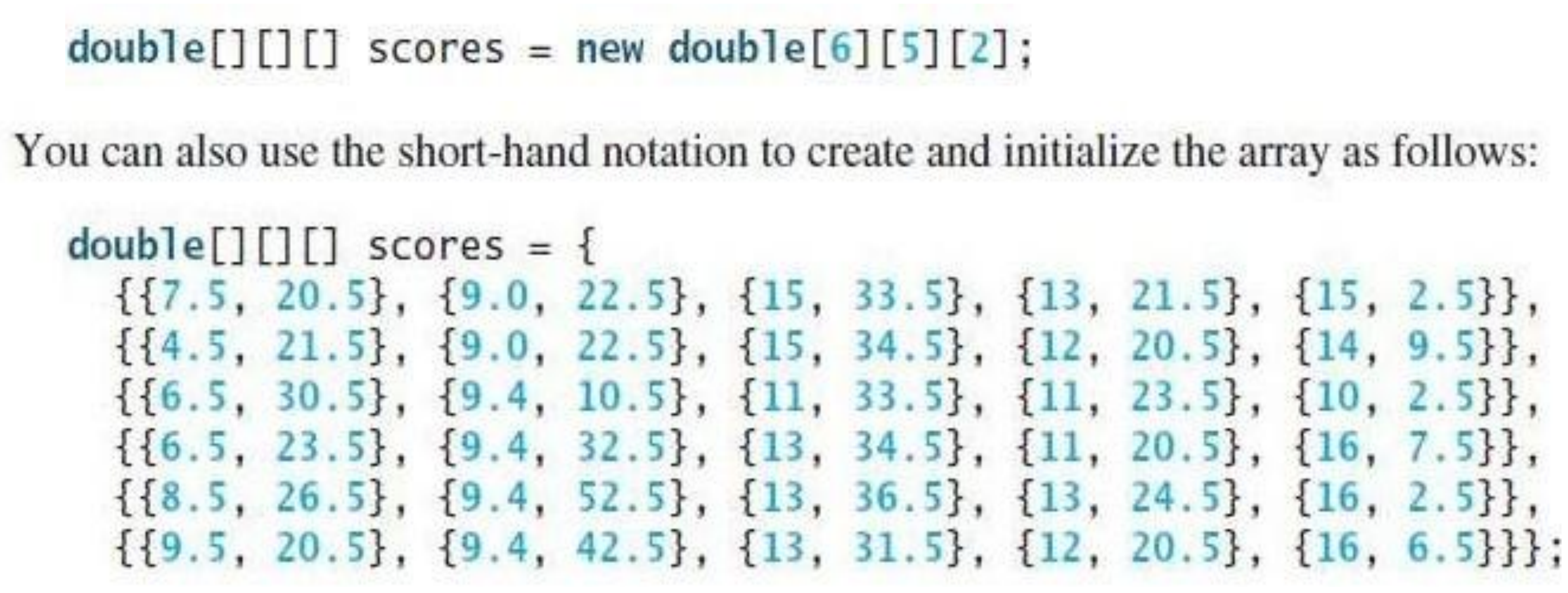
```
class Main
{
            public static void main(String args[])
            {
            int[][] a={{10,20},{30,40},{50,60}};//declaration and initialization
            System.out.println("Two dimensional array elements are");
            for (int i = 0; i < 3; i++)
            {
        for (int j = 0; j < 2; j++)
               {
           System.out.println(a[i][j]);
             }
            }
            }
}
```

2-D Array to Method Example

```
public class Main {
        public static void main(String[] args) {
                int[][] a = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
                arr2method(a); // pass it to the method
        }

        public static void arr2method(int[][] a) {
                System.out.println("Elements are :");

                for (int i = 0; i < a.length; i++) {
                        for (int j = 0; j < a[i].length; j++) {
                                System.out.print(a[i][j] + "\t");
                        }
                        System.out.println("");
                }
        }
}
```

Multi Dimensional Array

A two-dimensional array consists of an array of one-dimensional arrays and a three dimensional array consists of an array of two- dimensional arrays.



```
double[][][] scores = new double[6][5][2];
```

You can also use the short-hand notation to create and initialize the array as follows:

```
double[][][] scores = {
  {{7.5, 20.5}, {9.0, 22.5}, {15, 33.5}, {13, 21.5}, {15, 2.5}},
  {{4.5, 21.5}, {9.0, 22.5}, {15, 34.5}, {12, 20.5}, {14, 9.5}},
  {{6.5, 30.5}, {9.4, 10.5}, {11, 33.5}, {11, 23.5}, {10, 2.5}},
  {{6.5, 23.5}, {9.4, 32.5}, {13, 34.5}, {11, 20.5}, {16, 7.5}},
  {{8.5, 26.5}, {9.4, 52.5}, {13, 36.5}, {13, 24.5}, {16, 2.5}},
  {{9.5, 20.5}, {9.4, 42.5}, {13, 31.5}, {12, 20.5}, {16, 6.5}}};
```

Multi Dimensional Array

```
Data_Type[][][] Name = new int[Tables][Row_Size][Column_Size];

Int [] [] [] sum = new int [2] [3] [4]
```

Tables: Total number of tables it can accept. 2D Array is always a single table with rows and columns. In contrast, Multi Dimensional array in Java is more than one table with rows and columns.

Row_Size: Number of Row elements. For example, Row_Size = 3, then the 3D array holds 3 rows.

Column_Size: Column elements it can store. Column_Size = 4, then the 3D array holds 4 Columns.

Multi Dimensional Array

```
public class Main
{
  public static void main(String[] args) {

    //initialize 3-d array
    int[][][] intArray = { { { 1, 2, 3}, { 4, 5, 6 } ,  { 7, 8, 9 } } };
    System.out.println ("3-d array is given below :");
    //print the elements of array
    for (int i = 0; i < 1; i++)
      for (int j = 0; j < 3; j++)
      for (int z = 0; z < 3; z++)
        System.out.println ("intArray [" + i
      + "][" + j + "][" + z + "] = " + intArray [i][j][z]);
  }
}
```

Multi Dimensional Array

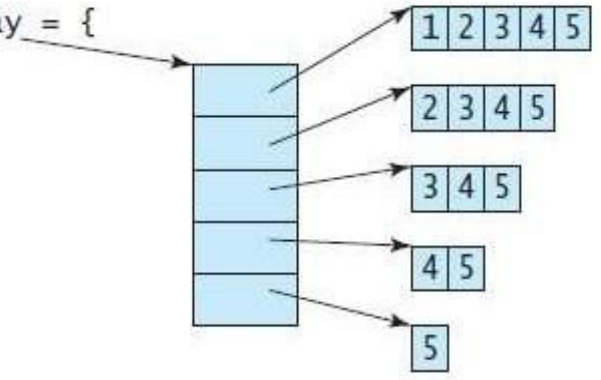
```
public class Main {
    public static void main(String[] args) {
        //initialize 3-d array
        int[][][] myArray = { { { 1, 2, 3 }, { 4, 5, 6 } },  { { 1, 4, 9 }, { 16, 25, 36 } },
                { { 1, 8, 27 }, { 64, 125, 216 } } };
        System.out.println("3x2x3 array is given below:");
        //print the 3-d array
        for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 2; j++) {
        for (int k = 0; k < 3; k++) {
        System.out.print(myArray[i][j][k] + "\t");
        }
        System.out.println();
                    }
        System.out.println();
        }     }   }
```

Jagged Array

- Each row in a two-dimensional array is itself an array. Thus, the rows can have different lengths. An array of this kind is known as a **jagged array**.
- If you don't know the values in a jagged array in advance, but do know the sizes.

```
int[][] triangleArray = new int[5][];
triangleArray[0] = new int[5];
triangleArray[1] = new int[4];
triangleArray[2] = new int[3];
triangleArray[3] = new int[2];
triangleArray[4] = new int[1];

triangleArray[0][3] = 50;
triangleArray[4][0] = 45;
```

```
int[][] triangleArray = {
  {1, 2, 3, 4, 5},
  {2, 3, 4, 5},
  {3, 4, 5},
  {4, 5},
  {5}
};
```



Jagged Array Example

```
//Java Program to illustrate the jagged array
class JaggedArrayDemo{
  public static void main(String[] args){
    //declaring a 2D array with odd columns
    int arr[][] = new int[3][];
    arr[0] = new int[3];
    arr[1] = new int[4];
    arr[2] = new int[2];
    //initializing a jagged array
    int count = 0;
    for (int i=0; i<arr.length; i++)
      for(int j=0; j<arr[i].length; j++)
        arr[i][j] = count++;

    //printing the data of a jagged array
    for (int i=0; i<arr.length; i++){
      for (int j=0; j<arr[i].length; j++){
        System.out.print(arr[i][j]+" ");
      }
      System.out.println();//new line
    }
  }
}
```

Jagged Array Example

```
class Main
{
   public static void main(String[] args)
   {
       // Declare a 2-D array with 3 rows
      int myarray[][] = new int[3][];

      // define and initialize jagged array

      myarray[0] = new int[]{1,2,3};
      myarray[1] = new int[]{4,5};
      myarray[2] = new int[]{6,7,8,9,10};

      // display the jagged array
      System.out.println("Two dimensional Jagged Array:");
      for (int i=0; i<myarray.length; i++)
      {
         for (int j=0; j<myarray[i].length; j++)
            System.out.print(myarray[i][j] + " ");
         System.out.println();
       }
   }
}
```

Command-Line Argument

- Can a main method receive arguments?
- **Yes**
- The main method can receive string arguments from the command line.
- Write a program to make calculator using Command Line argument.

```
class Main{
public static void main(String args[]){
System.out.println("Your first argument is: "+args[0]);
}
}
```

END OF UNIT - 3