Unit 3:
Methods and Arrays

What we will learn

- ✓ Single Dimensional arrays
- ✓ Copying arrays
- ✓ Passing and returning array from method
- ✓ Searching and sorting arrays and the Array class
- ✓ Two-Dimensional array and its processing
- ✓ Passing Two-dimensional Array to methods
- ✓ Multidimensional Arrays

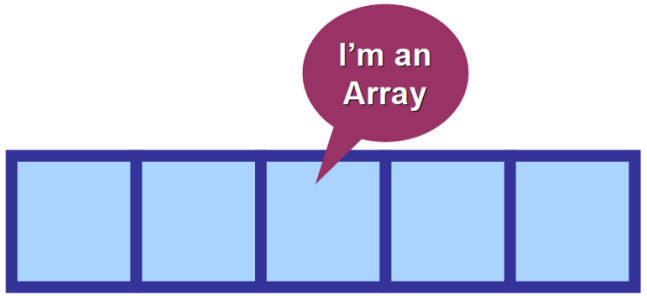
Unit-3

Introduction

Large Data Handling

Why Array?

- Very often we need to deal with relatively large set of data.
- E.g.
  - Percentage of all the students of the college. (May be in thousands)
  - Age of all the citizens of the city. (May be lakhs)
- We need to declare thousands or lakhs of the variable to store the data which is practically not possible.
- We need a solution to store more data in a single variable.
- **Array** is the most appropriate way to handle such data.
- As per English Dictionary, "***Array means collection or group or arrangement in a specific order.***"



Array

- **An array is a fixed size sequential collection of elements of same data type grouped under single variable name.**

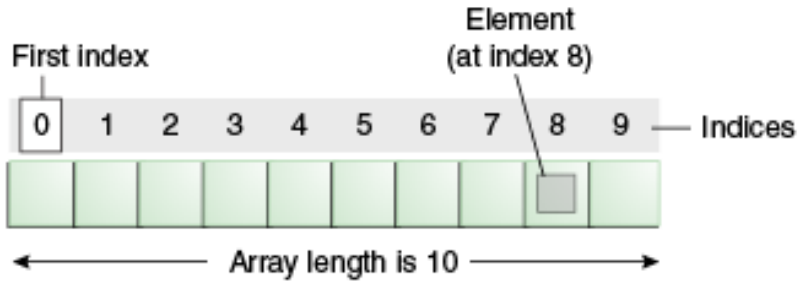
```
int percentage[10];
```

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

Fixed Size	Sequential	Same Data type	Single Variable Name
The size of an array is fixed at the time of declaration which cannot be changed later on. Here **array size** is **10**.	All the elements of an array are stored in a consecutive blocks in a memory. **10** (0 to 9)	Data type of all the elements of an array is same which is defined at the time of declaration. Here **data type** is **int**	All the elements of an array will be referred through common name. Here **array name** is **percentage**

Array declaration

- Normal Variable Declaration: int **a**;
- Array Variable Declaration: int **b**[10];
- Individual value or data stored in an array is known as an **element of an array**.
- Positioning / indexing of an elements in an array always **starts with 0 not 1**.
  - If 10 elements in an array then index is 0 to 9
  - If 100 elements in an array then index is 0 to 99
  - If 35 elements in an array then index is 0 to 34
- Variable **a** stores 1 integer number where as variable **b** stores 10 integer numbers which can be accessed as b[0], b[1], b[2], b[3], b[4], b[5], b[6], b[7], b[8] and b[9]



Array

- Important point about Java array.
  - An array is **derived** datatype.
  - An array is **dynamically** allocated.
  - The individual elements of an array is refereed by their **index/subscript** value.
  - The **subscript** for an array always begins with **0**.

35	13	28	106	35	42	5	83	97	14
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

One-Dimensional Array

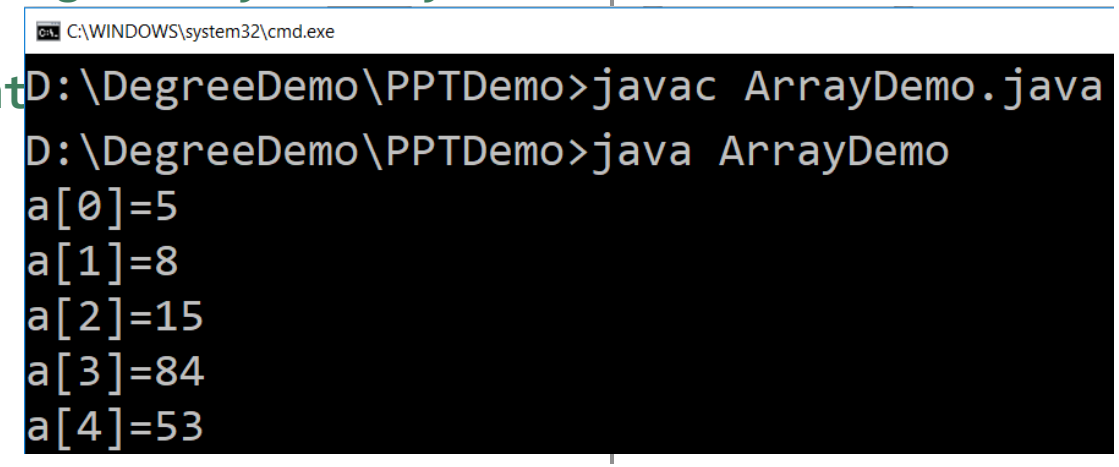
- An array using **one subscript** to represent the **list of elements** is called **one dimensional array**.
- A One-dimensional array is essentially a **list** of **like-typed variables.**
- Array declaration: `type var-name[];`

  Example: `int student_marks[];`
- Above example will represent array with no value (null).
- To link **student_marks** with actual array of integers, we must allocate one using ***new*** keyword.

  Example: `int student_marks[] = new int[20];`

Example (Array)

```
public class ArrayDemo{
    public static void main(String[] args) {
        int a[]; // or int[] a
        // till now it is null as it does not assigned any memory

        a = new int[5]; // here we actually creat
        a[0] = 5;
        a[1] = 8;
        a[2] = 15;
        a[3] = 84;
        a[4] = 53;

        /* in java we use length property to determine the length
         * of an array, unlike c where we used sizeof function */
        for (int i = 0; i < a.length; i++) {
                System.out.println("a["+i+"]="+a[i]);
        }
    }
}
```



WAP to store 5 numbers in an array and print them

```
1. import java.util.*;
2. class ArrayDemo1{
3. public static void main (String[] args){
4.       int i, n;
5.       int[] a=new int[5];
6.       Scanner sc = new Scanner(System.in);
7.       System.out.print("enter Array Length:");
8.       n = sc.nextInt();
9.       for(i=0; i<n; i++) {
10.           System.out.print("enter a["+i+"]:");
11.           a[i] = sc.nextInt();
12.      }
13.      for(i=0; i<n; i++)
14.          System.out.println(a[i]);
15.   }
16.}
```

Output:

```
enter Array
Length:5
enter a[0]:1
enter a[1]:2
enter a[2]:4
enter a[3]:5
enter a[4]:6
1
2
4
5
6
```

WAP to print elements of an array in reverse order

```
1. import java.util.*;
2. public class RevArray{
3. public static void  main(String[] args) {
4.    int i, n;
5.    int[] a;
6.    Scanner sc=new Scanner(System.in);
7.    System.out.print("Enter Size of an Array:");
8.    n=sc.nextInt();
9.    a=new int[n];
10.   for(i=0; i<n; i++){
11.       System.out.print("enter a["+i+"]:");
12.       a[i]=sc.nextInt();
13.   }
14.   System.out.println("Reverse Array");
15.   for(i=n-1; i>=0; i--)
16.       System.out.println(a[i]);
17.  }
18.}
```

Output:

```
Enter Size of an
Array:5
enter a[0]:1
enter a[1]:2
enter a[2]:3
enter a[3]:4
enter a[4]:5
Reverse Array
5
4
3
2
1
```

WAP to count positive number, negative number and zero from an array of n size

```
1.  import java.util.*;
2.  class ArrayDemo1{
3.  public static void main (String[] args){
4.        int n,pos=0,neg=0,z=0;
5.        int[] a=new int[5];
6.        Scanner sc = new Scanner(System.in);
7.        System.out.print("enter Array Length:");
8.        n = sc.nextInt();
9.        for(int i=0; i<n; i++) {
10.           System.out.print("enter a["+i+"]:");
11.           a[i] = sc.nextInt();
12.           if(a[i]>0)
13.             pos++;
14.           else if(a[i]<0)
15.             neg++;
16.           else
17.             z++;
18.       }
19.       System.out.println("Positive no="+pos);
20.       System.out.println("Negative no="+neg);
21.       System.out.println("Zero no="+z);
22. }}
```

Output:

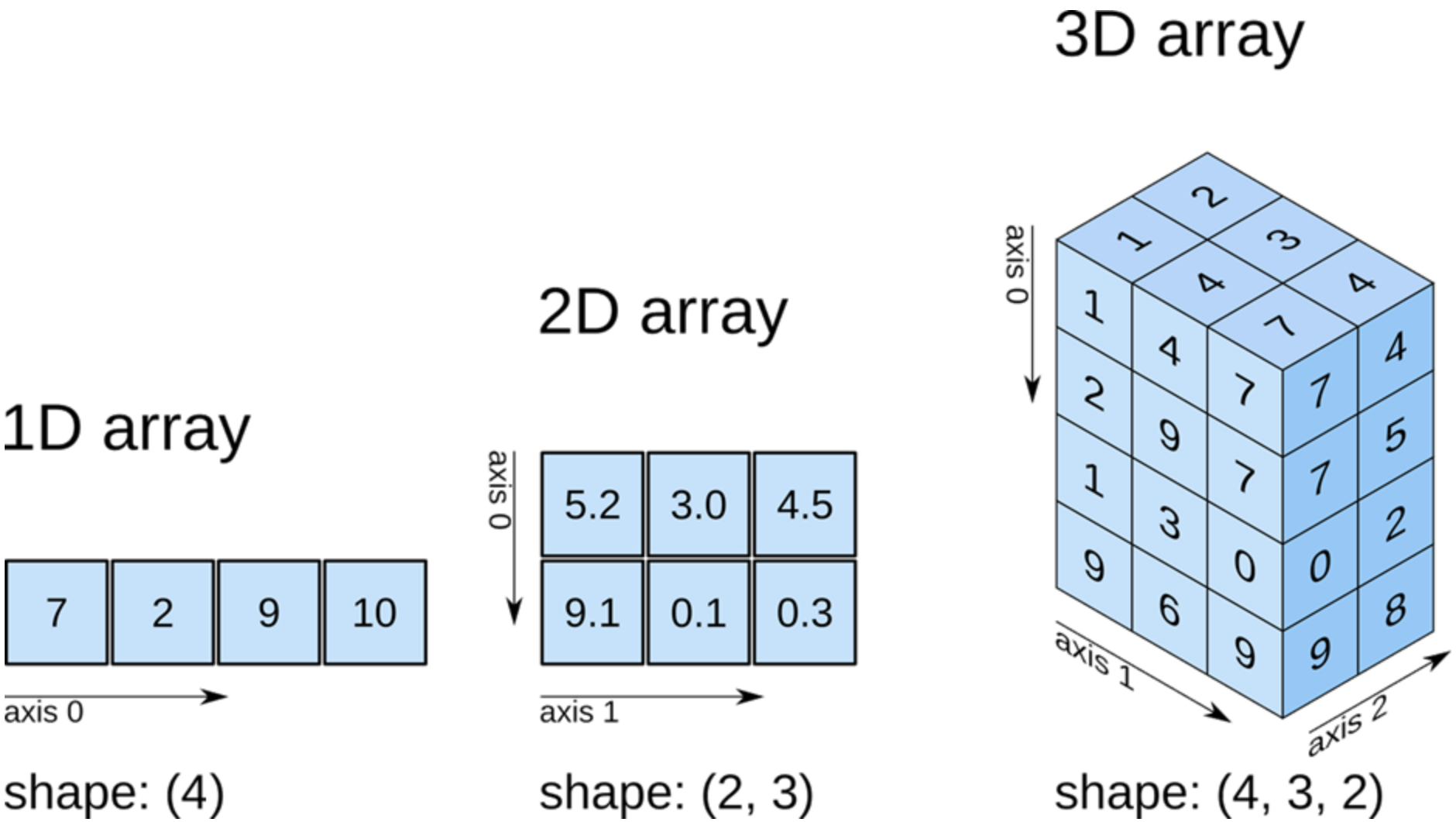
```
enter Array
Length:5
enter a[0]:-3
enter a[1]:5
enter a[2]:0
enter a[3]:-2
enter a[4]:00
Positive no=1
Negative no=2
Zero no=2
```

Exercise: Array

1. WAP to count odd and even elements of an array.
2. WAP to calculate sum and average of n numbers from an array.
3. WAP to find largest and smallest from an array.

Multidimensional Array

Multidimensional Array



WAP to read 3 x 3 elements in 2d array

```
1. import java.util.*;
2. class Array2Demo{
3. public static void main(String[] args) {
4.    int size;
5.    Scanner sc=new Scanner(System.in);
6.    System.out.print("Enter size of an array");
7.    size=sc.nextInt();
8.    int a[][]=new int[size][size];
9.    for(int i=0;i<a.length;i++){
10.     for(int j=0;j<a.length;j++){
11.             a[i][j]=sc.nextInt();
12.     }
13.   }

14.   for(int i=0;i<a.length;i++){
15.     for(int j=0;j<a.length;j++){
16.          System.out.print("a["+i+"]["+j+"]:"+a[i][j]+"
17.     }
18.     System.out.println();
19.   }
20. }
21.}
```

	Column-0	Column-1	Column-2
Row-0	**11**	**18**	**-7**
Row-1	**25**	**100**	**0**
Row-2	**-4**	**50**	**88**

Output:

```
11
12
13
14
15
16
17
18
19
a[0][0]:11      a[0][1]:12      a[0][2]:13
a[1][0]:14      a[1][1]:15      a[1][2]:16
a[2][0]:17      a[2][1]:18      a[2][2]:19
```

WAP to perform addition of two 3 x 3 matrices

```
1. import java.util.*;
2. class Array2Demo{
3. public static void main(String[] args) {
4. int size;
5. int a[][],b[][],c[][];
6. Scanner sc=new Scanner(System.in);
7.  System.out.print("Enter size of an
                                    array:");
8.  size=sc.nextInt();
9.  a=new int[size][size];
10. System.out.println("Enter array
                                  elements:");
11.for(int i=0;i<a.length;i++){
12.    for(int j=0;j<a.length;j++){
13.       System.out.print("Enter
                        a["+i+"]["+j+"]:");
14.       a[i][j]=sc.nextInt();
15.    }
16.}
```

```
1. b=new int[size][size];
2. for(int i=0;i<b.length;i++){
3.  for(int j=0;j<b.length;j++){
4.    System.out.print("Enter
                        b["+i+"]["+j+"]:");
5.    b[i][j]=sc.nextInt();
6.  }
7. }
8. c=new int[size][size];
9. for(int i=0;i<c.length;i++){
10. for(int j=0;j<c.length;j++){
11. System.out.print("c["+i+"]["+j+"]:“
                    +(a[i][j]+b[i][j])+"\t");
12.  }
13.  System.out.println();
14.  }
15. }//main()
16.}//class
```

WAP to perform addition of two 3 x 3 matrices

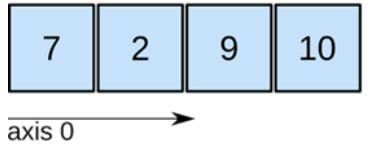
Output:

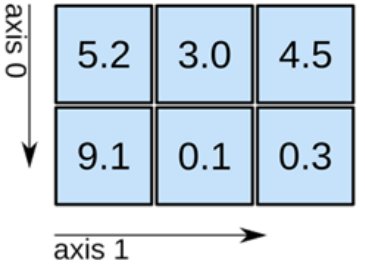
```
Enter size of an array:3
Enter array elements:
Enter a[0][0]:1
Enter a[0][1]:1
Enter a[0][2]:1
Enter a[1][0]:1
Enter a[1][1]:1
Enter a[1][2]:1
Enter a[2][0]:1
Enter a[2][1]:1
Enter a[2][2]:1
Enter b[0][0]:4
Enter b[0][1]:4
Enter b[0][2]:4
Enter b[1][0]:4
Enter b[1][1]:4
Enter b[1][2]:4
Enter b[2][0]:4
Enter b[2][1]:4
Enter b[2][2]:4
c[0][0]:5       c[0][1]:5       c[0][2]:5
c[1][0]:5       c[1][1]:5       c[1][2]:5
c[2][0]:5       c[2][1]:5       c[2][2]:5
```

Initialization of an array elements

1. One dimensional Array

1. int a[5] = { 7, 3, -5, 0, 11 }; // a[0]=7, a[1] = 3, a[2] = -5, a[3] = 0, a[4] = 11
2. int a[5] = { 7, 3 }; // a[0] = 7, a[1] = 3, a[2], a[3] and a[4] are 0
3. int a[5] = { 0 }; // all elements of an array are initialized to 0





2. Two dimensional Array

1. int a[2][4] = { { 7, 3, -5, 10 }, { 11, 13, -15, 2} }; // 1st row is 7, 3, -5, 10 & 2nd row is 11, 13, -15, 2
2. int a[2][4] = { 7, 3, -5, 10, 11, 13, -15, 2 }; // 1st row is 7, 3, -5, 10 & 2nd row is 11, 13, -15, 2
3. int a[2][4] = { { 7, 3 }, { 11} }; // 1st row is 7, 3, 0, 0 & 2nd row is 11, 0, 0, 0
4. int a[2][4] = { 7, 3 }; // 1st row is 7, 3, 0, 0 & 2nd row is 0, 0, 0, 0
5. int a[2][4] = { 0 }; // 1st row is 0, 0, 0, 0 & 2nd row is 0, 0, 0, 0

Multi-Dimensional Array

- In java, multidimensional array is actually **array of arrays.**
- **Example**: `int runPerOver[][] = new int[3][6];`

First Over (a[0])	**4** a[0][0]	**0** a[0][1]	**1** a[0][2]	**3** a[0][3]	**6** a[0][4]	**1** a[0][5]
Second Over (a[1])	**1** a[1][0]	**1** a[1][1]	**0** a[1][2]	**6** a[1][3]	**0** a[1][4]	**4** a[1][5]
Third Over (a[2])	**2** a[2][0]	**1** a[2][1]	**1** a[2][2]	**0** a[2][3]	**1** a[2][4]	**1** a[2][5]

- **`length` field:**
  - If we use length field with multidimensional array, it will return length of first dimension.
  - Here, if **runPerOver.length** is accessed it will return **3**
  - Also if **runPerOver[0].length** is accessed it will be **6**

Multi-Dimensional Array (Example)

```
Scanner s = new Scanner(System.in);
int runPerOver[][] = new int[3][6];
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 6; j++) {
        System.out.print("Enter Run taken" +
        " in Over numner " + (i + 1) +
        " and Ball number " + (j + 1) + " = ");
        runPerOver[i][j] = s.nextInt();
    }
}
int totalRun = 0;
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 6; j++) {
        totalRun += runPerOver[i][j];
    }
}
double average = totalRun / (double) runPerOver.length;
System.out.println("Total Run = " + totalRun);
System.out.println("Average per over = " + average);
```

```
C:\WINDOWS\system32\cmd.exe
Enter Run taken in Over numner 1 and Ball number 1 = 4
Enter Run taken in Over numner 1 and Ball number 2 = 0
Enter Run taken in Over numner 1 and Ball number 3 = 1
Enter Run taken in Over numner 1 and Ball number 4 = 3
Enter Run taken in Over numner 1 and Ball number 5 = 6
Enter Run taken in Over numner 1 and Ball number 6 = 1
Enter Run taken in Over numner 2 and Ball number 1 = 1
Enter Run taken in Over numner 2 and Ball number 2 = 1
Enter Run taken in Over numner 2 and Ball number 3 = 0
Enter Run taken in Over numner 2 and Ball number 4 = 6
Enter Run taken in Over numner 2 and Ball number 5 = 0
Enter Run taken in Over numner 2 and Ball number 6 = 4
Enter Run taken in Over numner 3 and Ball number 1 = 2
Enter Run taken in Over numner 3 and Ball number 2 = 1
Enter Run taken in Over numner 3 and Ball number 3 = 1
Enter Run taken in Over numner 3 and Ball number 4 = 0
Enter Run taken in Over numner 3 and Ball number 5 = 1
Enter Run taken in Over numner 3 and Ball number 6 = 1
Total Run = 33
Average per over = 11.0
```

Multi-Dimensional Array (Cont.)

- **manually** allocate **different** size:

```
int runPerOver[][] = new int[3][];
runPerOver[0] =  new int[6];
runPerOver[1]  =  new int[7];
runPerOver[2]  =  new int[6];
```

- **initialization**:

```
int runPerOver[][] = {
                        {0,4,2,1,0,6},
                        {1,-1,4,1,2,4,0},
                        {6,4,1,0,2,2},
                }
```

Note : here to specify extra runs (Wide, No Ball etc..) negative values are used

Searching in Array

Searching in Array

- Searching is the process of looking for a specific element in an array. for example, discovering whether a certain element is included in the array.
- Searching is a common task in computer programming. Many algorithms and data structures are devoted to searching.
- We will discuss two commonly used approaches,
  - **Linear Search:** The linear search approach compares the key element key sequentially with each element in the array. It continues to do so until the key matches an element in the array or the array is exhausted without a match being found.
  - **Binary Search:** The binary search first compares the key with the element in the middle of the array. Consider the following three cases:
    - If the key is less than the middle element, you need to continue to search for the key only in the first half of the array.
    - If the key is equal to the middle element, the search ends with a match.
    - If the key is greater than the middle element, you need to continue to search for the key only in the second half of the array.

  Note: Array should be sorted in ascending order if we want to use Binary Search.

Linear Search

```
1. import java.util.*;
2. class LinearSearchDemo{
3. public static void main(String[] args) {
4.     int size;
5.     int a[]={1,2,3,4,5,6,7,8,9};
6.     int search;
7.     boolean flag=false;
8.     Scanner sc=new Scanner(System.in);
9.     System.out.print("Enter element to search");
10.    search=sc.nextInt();
11.       for(int i=0;i<a.length;i++){
12.           if(a[i]==search){
13.           System.out.println("element found at "+i+"th index");
14.           flag=true;
15.           break;
16.           }
17.       }
18.    if(!flag)
19.           System.out.println("element NOT found!");
20.    }
21.}
```

Output:

```
Enter element to search 6
element found at 5th index
Enter element to search 35
element NOT found!
```

Binary Search (Animation)

Binary Search

```
1.  import java.util.*;
2.  class BinaryDemo{
3.  public  static  void  main(String[]
                            args){
4.  int size;
5.  int a[]={1,2,3,4,5,6,7,8,9};
6.  int search;
7.  boolean flag=false;
8.  Scanner sc=new Scanner(System.in);
9.  System.out.print("Enter element to
                            search:");
                    .nextInt();
```

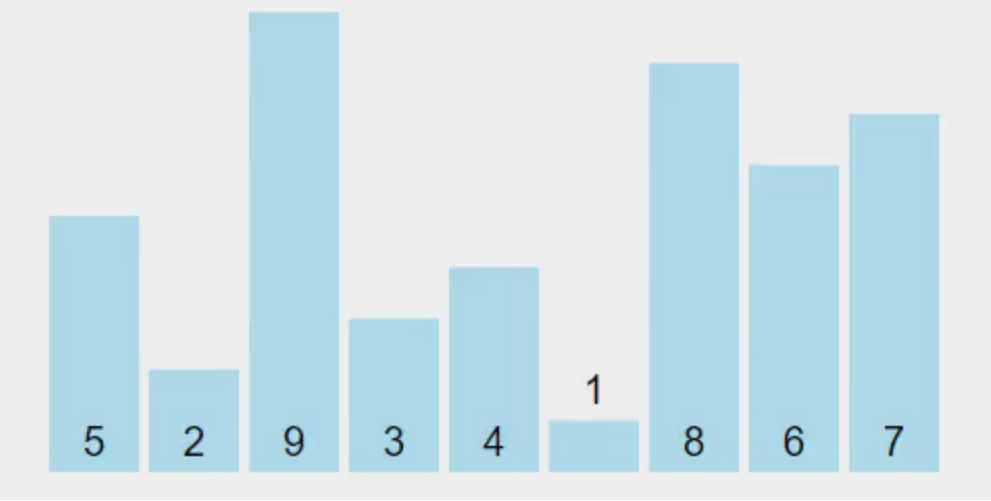
Output:

```
Enter element to search:5
element found at 4 index

Enter element to search:9
element found at 8 index

Enter element to search:56
element not found
```

```
13. while(high>=low){
14.       int mid=(high+low)/2;
15.       if(search==a[mid]){
16.         flag=true;
17.         System.out.println("element found
                              at "+mid+" index ");
18.         break;
19.       }
20.       else if(search<a[mid]){
21.             high=mid-1;
22.       }
23.       else if(search>a[mid]){
24.             low=mid+1;
25.       }
26. }//while
27. if(!flag)
28.  System.out.println("element not found");
29.   }
30. }
```

Sorting Array

- Sorting, like searching, is a common task in computer programming. Many different algorithms have been developed for sorting.
- There are many sorting techniques available, we are going to explore selection sort.
- Selection sort
  - finds the smallest number in the list and swaps it with the first element.
  - It then finds the smallest number remaining and swaps it with the second element, and so on, until only a single number remains.



Selection Sort (Example)

```
1. import java.util.*;
2. class SelectionSearchDemo{
3.  public static void main(String[] args)
{
4.  int a[]={ 5, 2, 9, 3, 4, 1, 8, 6, 7 };
```

```
5. for (int i = 0; i < a.length - 1; i++) {
6. // Find the minimum in the list[i..a.length-1]
7.   int min = a[i];
8.   int minIndex = i;
9.   for (int j = i + 1; j < a.length; j++) {
10.   if (min > a[j]) {
11.     min = a[j];
12.     minIndex = j;
13.   }
14.  }//inner for loop j
15.// Swap a[i] with a[minIndex]
16.  if (minIndex != i) {
17.    a[minIndex] = a[i];
18.    a[i] = min;
19.    }
20. }//outer for i
21. for(int temp: a) { // this is foreach loop
22.    System.out.print(temp + ", ");
23.   }
24.  }//main()
25.}//class
```

Output:

```
1, 2, 3, 4, 5, 6, 7, 8, 9,
```

What we will learn

- Defining and calling method
- Passing argument by values

Unit-3
Method

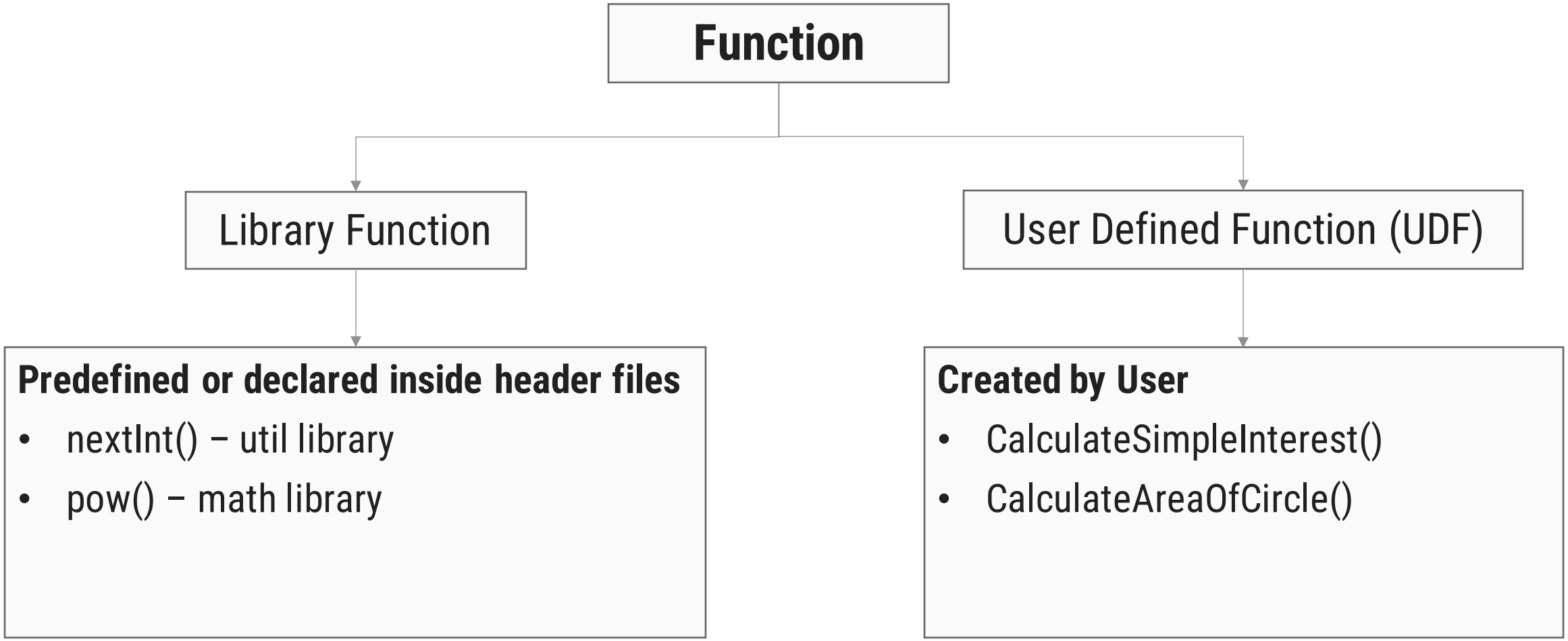
What is Method?

What is Method?

- A **method** is a group of statements that performs a specific task.
- A large program can be divided into the basic building blocks known as **method/function**.
- The function contains the set of programming statements enclosed by { }.
- Program execution in many programming language starts from the **main** function.
- main is also a **method/function**.

```
void main()
{
    // body part
}
```

Types of Function



Program Structure of Function

- User-defined function's program structure is divided into three parts as follows:

C:Function Structure

```
void func1();


void main()
{
   ....
   func1();
}

void func1()
{
   ....
   //function body
   ....
}
```

Function Prototype

Function call

Function definition

Java:Function for addition of 2 numbers

```
class MethodDemo{
 public static void main(String[] ags){
        int a=10,b=20,c;
        c=add(a,b);
        System.out.println("a+b="+c);
 }
static int add(int i, int j){
        return i+j;
 }
}
```

Method Definition

- A method definition defines the **method header** and **body**.
- A **method body** part defines method logic.
  - ↪ Method statements

Syntax

```
return-type method_name(datatyp1 arg1, datatype2 arg2, …)
{
        functions statements
}
```

Example

```
int addition(int a, int b);
{
        return a+b;
}
```

WAP to add two number using add(int, int) Function

```
1. class MethodDemo{
2.  public static void main(String[] args) {
3.   int a=10,b=20,c;
4.   MethodDemo md=new MethodDemo();
5.   c=md.add(a,b);
6.   System.out.println("a+b="+c);
7.  }//main()
8.  int add(int i, int j){
9.          return i+j;
10. }
11.}
```

Output:

```
a+b=30
```

Actual parameters v/s Formal parameters

- Values that are passed from the calling functions are known **actual parameters**.
- The variables declared in the function prototype or definition are known as **formal parameters**.
- Name of formal parameters can be same or different from actual parameters.
- Sequence of parameter is **important**, not name.

Actual Parameters

```
int a=10,b=20,c;
MethodDemo md=new MethodDemo();
c=md.add(a,b);
// a and b are the actual parameters.
```

Formal Parameters

```
int add(int i, int j)
{
                return i+j;
}
// i and j are the formal parameters.
```

Return Statement

- The function can **return only one value**.
- Function cannot return more than one value.
- If function is not returning any value then return type should be **`void`**.

Actual Parameters

```
int a=10,b=20,c;
MethodDemo md=new MethodDemo();
c=md.sub(a,b);
// a and b are the actual parameters.
```

Formal Parameters

```
int sub(int i, int j)
{
        return i - j;
}
// i and j are the formal parameters.
```

WAP to calculate the Power of a Number using method

```
1.  import java.util.*;
2.  public class PowerMethDemo1{
3.  public static void main(String[] args){
4.     int num, pow, res;
5.     Scanner sc=new Scanner(System.in);
6.     System.out.print("enter num:");
7.     num=sc.nextInt();
8.     System.out.print("enter pow:");
9.     pow=sc.nextInt();
10.    PowerMethDemo1 pmd=new
                              PowerMethDemo1();
11.    res = pmd.power(num, pow);
12.    System.out.print("ans="+res);
13.    } //main()
```

```
14.     int power(int a, int b){
15.      int i, r = 1;
16.      for(i=1; i<=b; i++)
17.      {
18.              r = r * a;
19.      }
20.      return r;
21.    }//power()
22. }//class
```

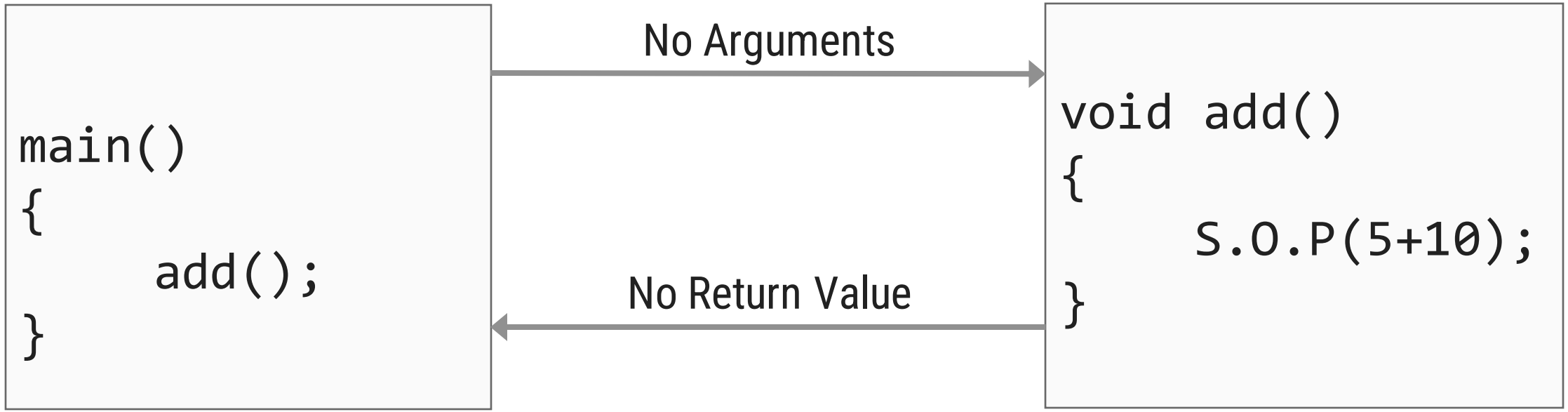
Output:

```
enter num:5
enter pow:3
ans=125
```

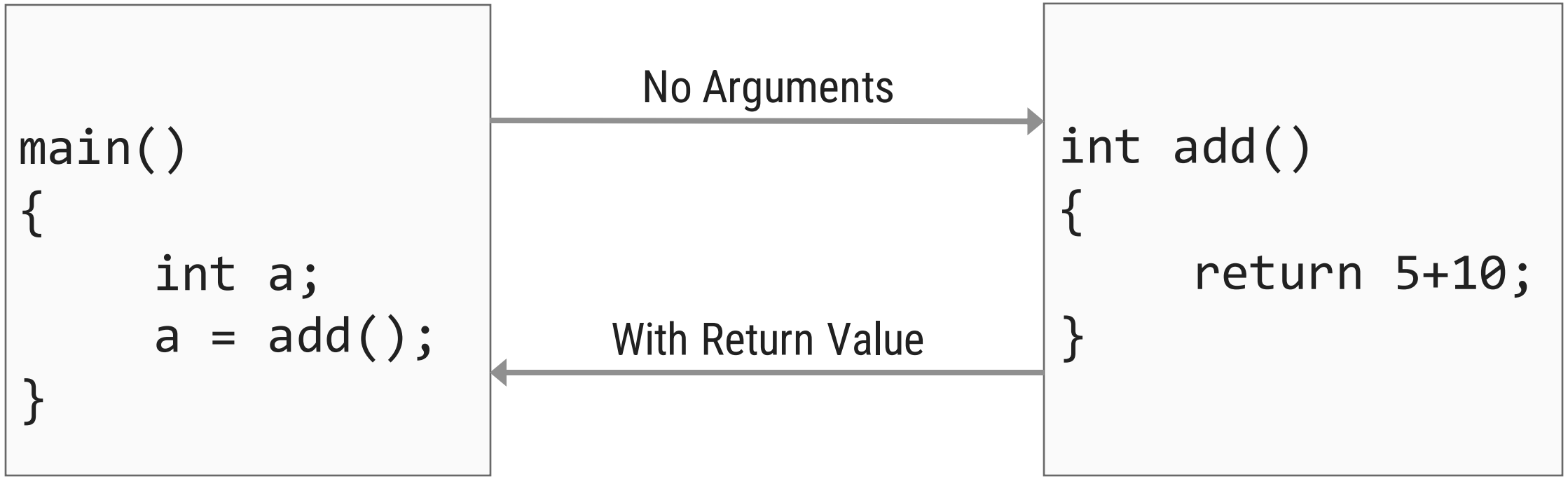
Types of Methods(Method Categories)

- Functions can be divided in 4 categories based on arguments and return value.
  1. Method without arguments and without return value `void add();`
  2. Method without arguments and with return value `int add();`
  3. Method with arguments and without return value `void add(int, int);`
  4. Method with arguments and with return value `int add(int, int);`

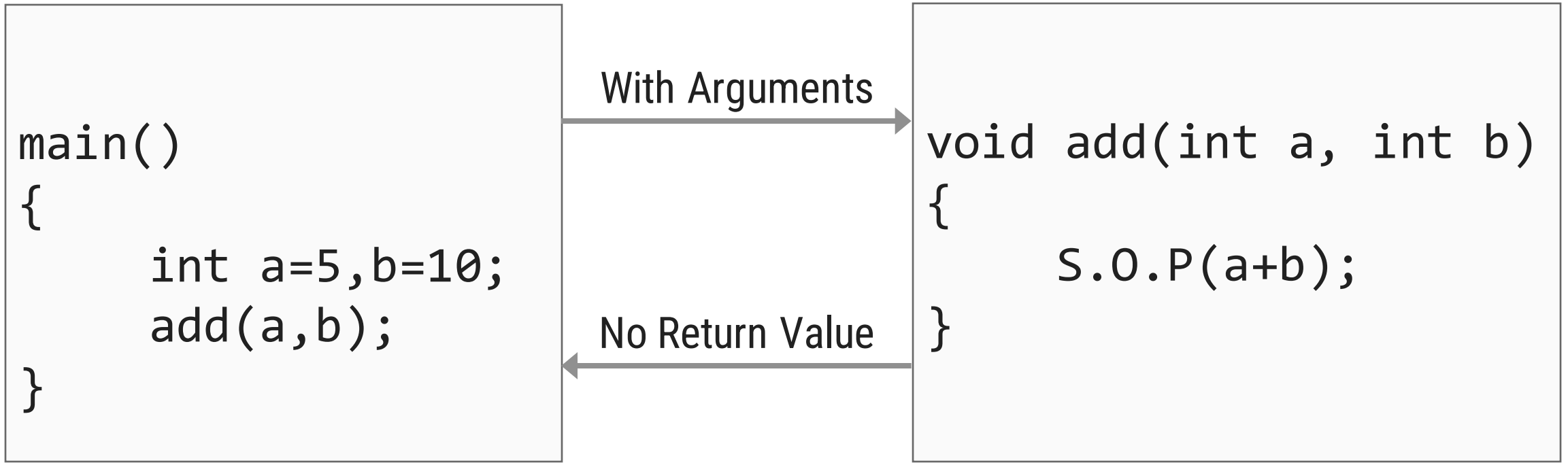
Method without arguments and without return value



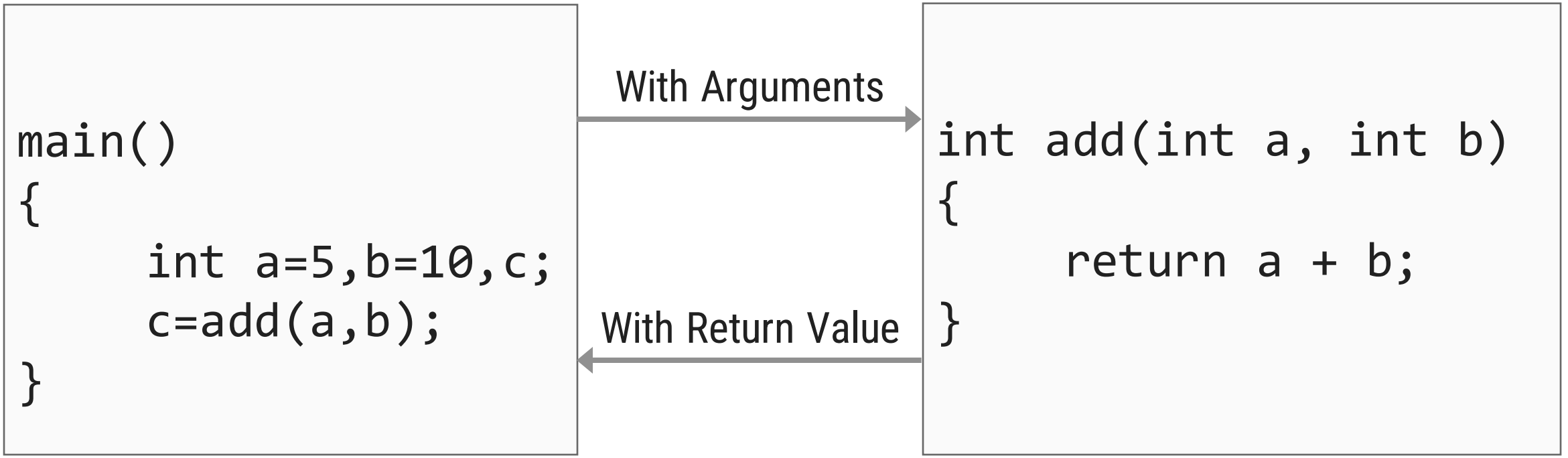
Method without arguments and with return value



Method with arguments and without return value



Method with arguments and with return value



Method Overloading

Method Overloading: Compile-time Polymorphism

- **Definition**: When two or more methods are implemented that share same name but different parameter(s), the methods are said to be ***overloaded***, and the process is referred to as ***method overloading***
- Method overloading is one of the ways that Java implements polymorphism.
- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.
  - E.g.

```
public void draw()
public void draw(int height, int width)
public void draw(int radius)
```

- Thus, overloaded methods must differ in the type and/or number of their parameters.
- While in overloaded methods with different return types and same name & parameter are not allowed ,as the return type alone is insufficient for the compiler to distinguish two versions of a method.

Method Overloading: Compile-time Polymorphism

```
1. class Addition{
2. int i,j,k;
3.   void add(int a){
4.     i=a;
5.     System.out.println("add i="+i);
6.   }
7.   void add(int a,int b){\\overloaded add()
8.     i=a;
9.     j=b;
10.    System.out.println("add i+j="+(i+j));
11.  }
12.  void add(int a,int b,int c){\\overloaded add()
13.    i=a;
14.    j=b;
15.    k=c;
16.    System.out.println("add i+j+k="+(i+j+k));
17.  }
18.}
```

```
19.class OverloadDemo{
20.public static void
       main(String[] args){
21.   Addition a1= new Addition();
22.   //call all versions of add()
23.     a1.add(20);
24.     a1.add(30,50);
25.     a1.add(10,30,60);
26.     }
27.}
```

Output

```
add i=20
add i+j=80
add i+j+k=100
```

Method Overloading: Compile-time Polymorphism

```
1. class Addition{
2. int i,j,k;
3.   void add(int a){
4.      i=a;
5.      System.out.println("add i="+i);
6.   }
7.   void add(int a,int b){\\overloaded add()
8.      i=a;
9.      j=b;
10.     System.out.println("add i+j="+(i+j));
11.  }
12.  void add(double a, double b){\\overloaded add()
13.     System.out.println("add a+b="+(a+b));
14.  }
15.  void add(int a,int b,int c){\\overloaded add()
16.     i=a;
17.     j=b;
18.     k=c;
19.     System.out.println("add i+j+k="+(i+j+k));
20.  }
21.}
```

```
22.class OverloadDemo{
23.public static void
       main(String[] args){
24.  Addition a1= new Addition();
25.  //call all versions of add()
26.    a1.add(20);
27.    a1.add(30,50);
28.    a1.add(10,30,60);
29.    a1.add(30.5,50.67);
30.    }
31.}
```

Output

```
add i=20
add i+j=80
add i+j+k=100
add a+b=81.17
```

Method Overloading: Points to remember

- Method overloading supports polymorphism because it is one way that Java implements the "one interface, multiple methods" paradigm.
- Overloading **increases** the readability of the program.
- There are two ways to overload the method in java
  1. By changing number of arguments
  2. By changing the data type
- In java, method overloading is not possible by changing the return type of the method only because of ambiguity.

Method Overloading: Points to remember

Can we overload java main() method?

- Yes, by method overloading. We 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.

Advantages of Method

- Reduced Code Redundancy
  - ↪ Rewriting the same logic or code again and again in a program can be avoided.
- Reusability of Code
  - ↪ Same function can be call from multiple times without rewriting code.
- Reduction in size of program
  - ↪ Instead of writing many lines, just function need to be called.
- Saves Development Time
  - ↪ Instead of changing code multiple times, code in a function need to be changed.
- More Traceability of Code
  - ↪ Large program can be easily understood or traced when it is divide into functions.
- Easy to Test & Debug
  - ↪ Testing and debugging of code for errors can be done easily in individual function.

Scope, Lifetime and Visibility of a Variable

Scope of a Variable

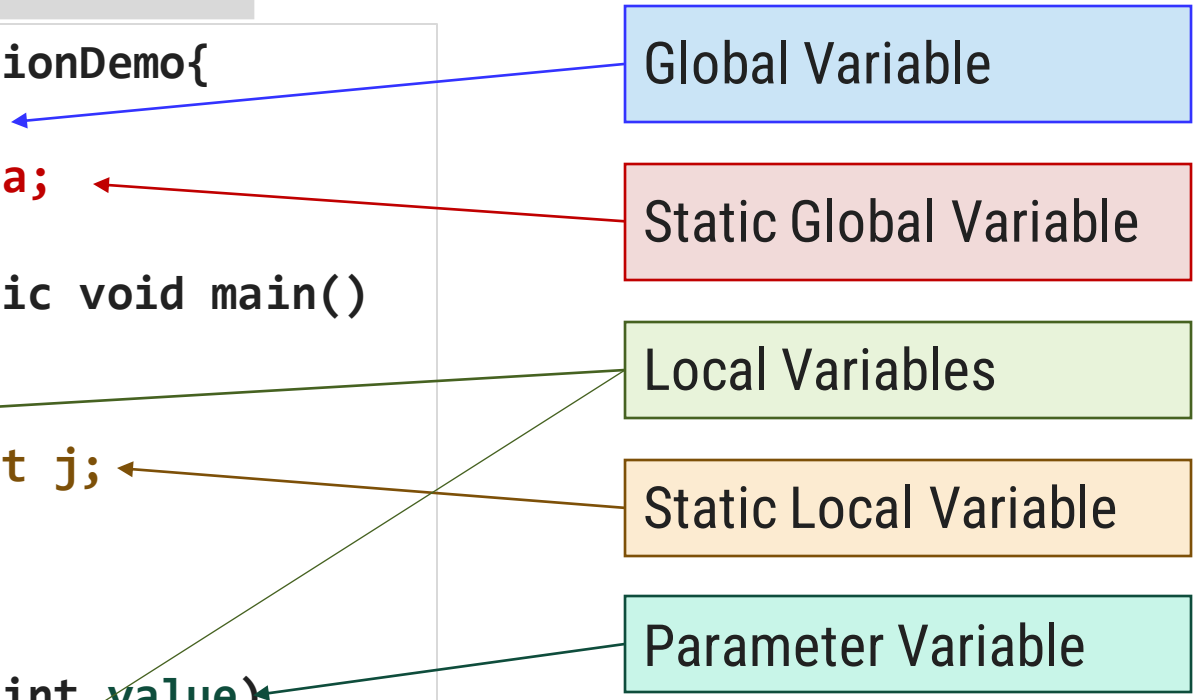
- Whenever we declare a variable, we also determine its scope, lifetime and visibility.

Scope	Scope is defined as the area in which the declared variable is 'accessible'. **There are five scopes**: program, file, function, block, and class. ***Scope*** is the region or section of code where a variable can be accessed. Scoping has to do with **when** a variable is accessible and used.
Lifetime	The lifetime of a variable is the period of time in which the variable is allocated a space (i.e., the period of time for which it "lives"). There are three lifetimes in C: static, automatic and dynamic. ***Lifetime* is the time duration where an object/variable is in a valid state.** Lifetime has to do with when a variable is created and destroyed
Visibility	Visibility is the "accessibility" of the variable declared. It is the result of hiding a variable in outer scopes.

Scope of a Variable

Function Structure

```
class FunctionDemo{
float f;
static int a;

public static void main()
{
  int i;
  static int j;
  func1(i);
}

void func1(int value)
{
  int x;
  //function body
  ....
}
}
```



Scope	Description
Local (block/ function)	"visible" within function or statement block from point of declaration until the end of the block.
Class	"seen" by class members.
File (program)	visible within current file.
Global	visible everywhere unless "hidden".

Lifetime of a variable

- The **lifetime** of a variable or object is the time period in which the variable/object has valid memory.
- Lifetime is also called "allocation method" or "storage duration".

	Lifetime	**Stored**
Static	Entire duration of the program's execution.	data segment
Automatic	Begins when program execution enters the function or statement block and ends when execution leaves the block.	function call stack
Dynamic	Begins when memory is allocated for the object (e.g., by a call to malloc() or using new) and ends when memory is deallocated (e.g., by a call to free() or using delete).	heap

Scope vs Lifetime of a variable

Variable Type	Scope of a Variable	Lifetime of a Variable
Instance Variable	Throughout the class except in static methods	Until object is available in the memory
Class Variable	Throughout the class	Until end of the Class
Local Variable	Throughout the block/function in which it is declared	Until control leaves the block

Exercise

1. Write a function to check whether given number is prime or not.
2. Write a function to search a given number from an array.

Thank You