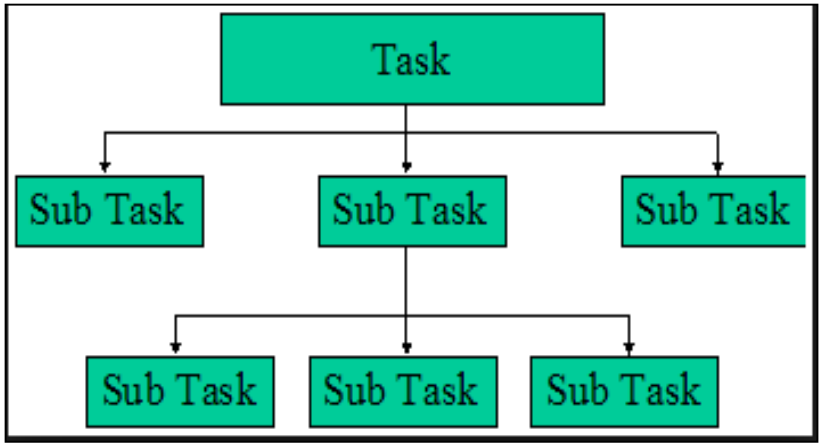
Top-down programming design

- **Top-down design** is a *programming style*, the mainstay of traditional procedural languages, in which design begins by specifying complex pieces and then dividing them into successively smaller pieces.
- **Top-down design** starts with an description of the overall system and usually consists of a *hierarchical structure* which contains more detailed descriptions of the system at each lower level.
- This method involves a *hierarchical* or tree-like *structure* for a system as illustrated by the following diagram:



Top-down programming design

- The technique for writing a program using **top-down design** is to write a main program (main method) that calls all the other methods it will need.

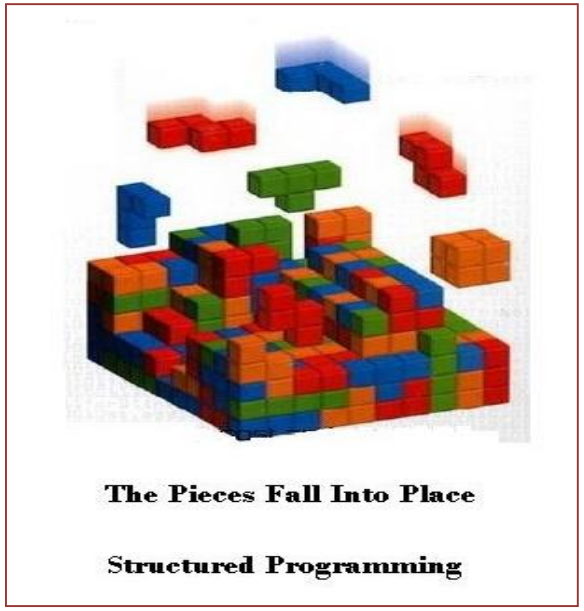
- If you have a big problem to solve, then a very effective method of working towards a solution is to break it down into smaller, more manageable problems.

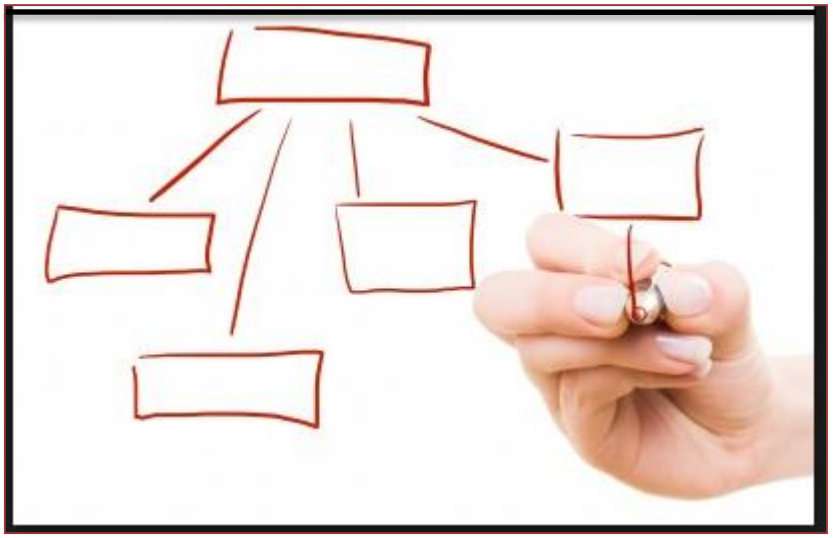
- You can ***invoke*** the same method ***repeatedly***. In fact, it is quite common and useful to do so.

- Methods can simplify a program by hiding a complex computation behind a single command.

Structured programming

- The use of methods will be our first step in the direction of **structured programming** (**התכנות המבני**) .





- Structured programming associated with a **top-down** approach to **design** where an overview of the system is first formulated, specifying but not detailing any first-level subsystems.
- Each subsystem is then refined in yet greater detail *until the entire specification is reduced to base elements.*

Methods - opening problem

Find the sum of integers from **1** to **10**, from **20** to **30**, and from **35** to **45** respectively.

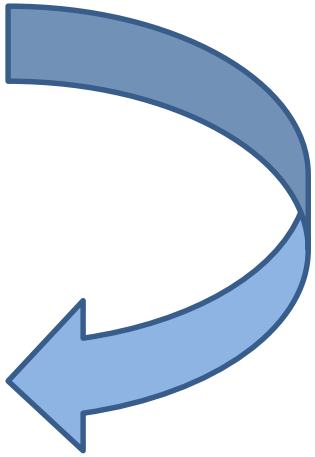
```
int sum = 0;
for (int i = 1; i <= 10; i++)
    sum + = i;
System.out.println("Sum from 1 to 10 is " + sum);
```

```
int sum = 0;
for (int i = 20; i <= 30; i++)
    sum + = i;
System.out.println("Sum from 20 to 30 is " + sum);
```

```
int sum = 0;
for (int i = 35; i <= 45; i++)
    sum + = i;
System.out.println("Sum from 35 to 45 is " + sum);
```

Cod reuse (שימוש חוזר)

- **Code reuse**, also called **software reuse**, is the use of existing program code, or software knowledge, to build new software (program code). The reuse of programming code is a common technique which attempts to ***save time and energy*** by reducing redundant work.
- Reusable components are simply pre-built pieces of programming code designed to perform a specific function.
- Programmers have always reused sections of code, methods, functions, and procedures. The **Java Math library** is a good example of code reuse.
- One of the most compelling **features** about **Java** is **code reuse**.



Solution - Method (פעולה/שיטה)

```
public static int sum(int i1, int i2)
{
    int sum = 0;
    for ( int i = i1; i <= i2; i++)
        sum += i;
    return sum;
} // sum
```

Think of a method as a ***subprogram*** that acts on data and often returns a value.

Methods are *time savers*, in that they allow for the repetition of sections of code without retyping the code.

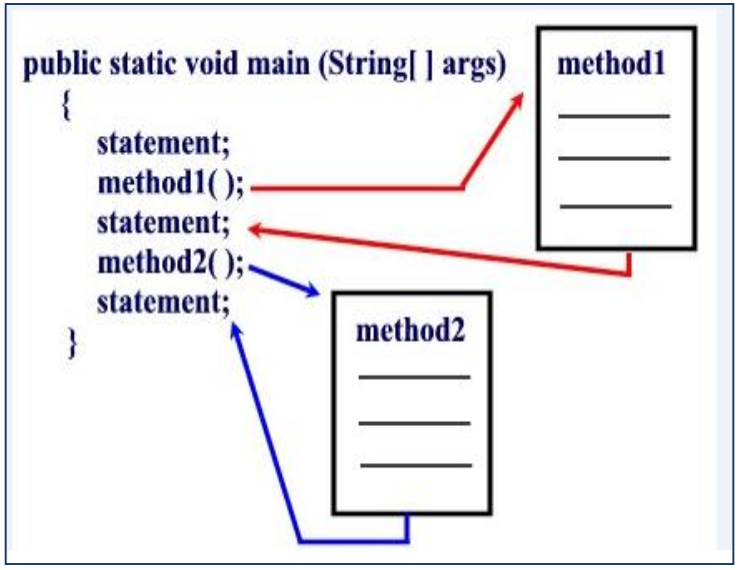
In certain other languages (C,Pascal,VB) methods are referred to as *procedures* and *functions*.

```
public static void main(String[ ] args)
{
   System.out.println("Sum from 1 to 10 is " + sum(1, 10));
   System.out.println("Sum from 20 to 30 is " + sum(20, 30));
   System.out.println("Sum from 35 to 45 is " + sum(35, 45));
} // main
```

What are Methods?

- Each method has its own name.
- When that name is encountered in a program, the execution of the program branches to the body of that method.
- When the method is finished, execution returns to the area of the program code from which it was called, and the program continues *on to the next line of code*.

methods can be saved and utilized again and again in newly developed programs.



The use of methods will be our first step in the direction of ***structured programming***.

Defining Methods

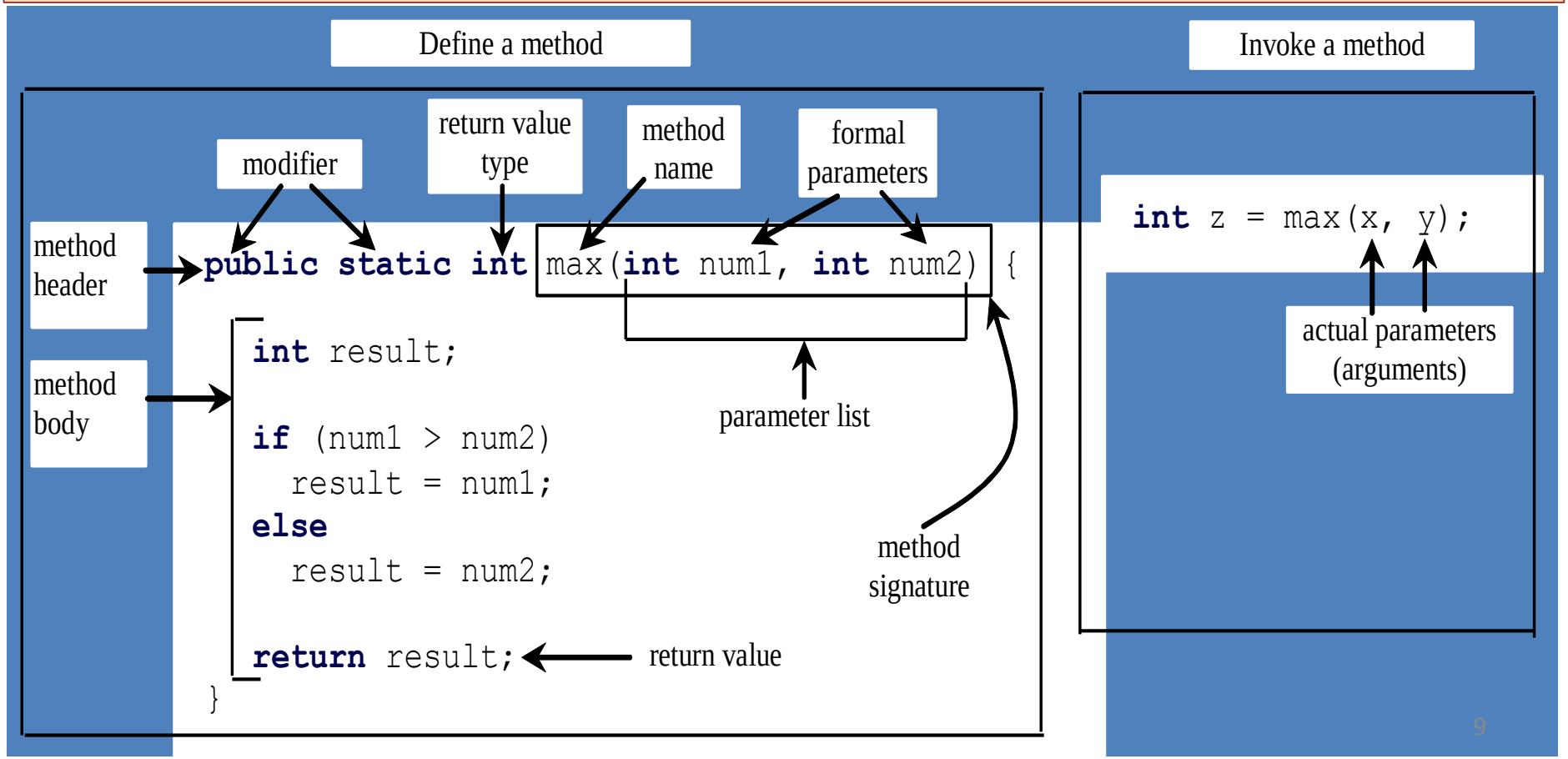
- **Modifiers:** The modifier, **which is optional**, tells the compiler how to call the method. This defines the access type of the method.(*סוג גישה*)
- **Return Type:** A method may return a value. The returnValueType is the data type of the value the method returns. Some methods perform the desired operations without returning a value. In this case, the returnValueType is the keyword **void** .
- **Method Name:** This is the actual name of the method. The method name and the parameter list together constitute the method signature(חתימה).
- **Parameters:** A parameter is like a placeholder. When a method is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.
- **Method Body:** The method body contains a collection of statements that define what the method does.

Defining Methods

In general, a method has the following syntax:

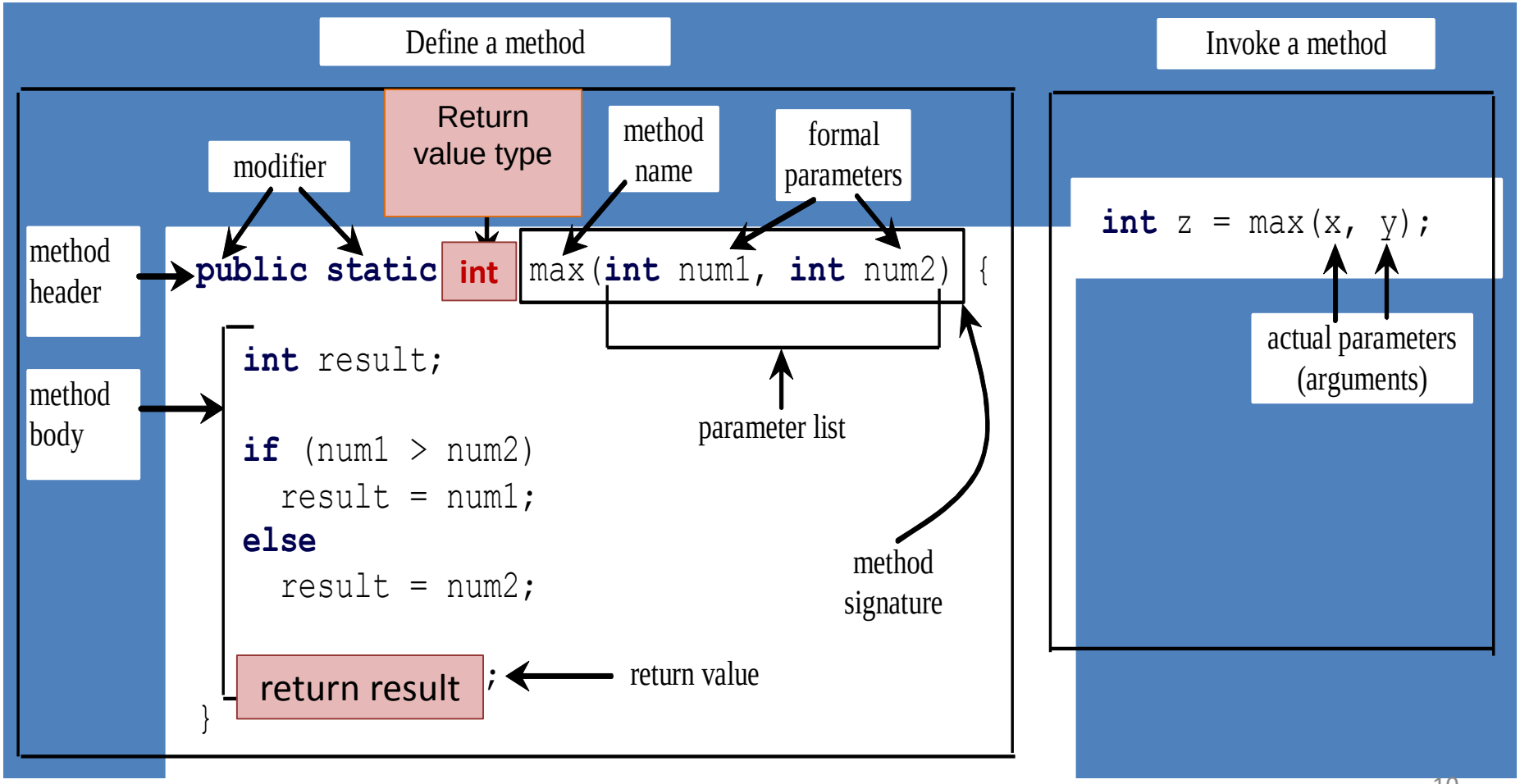
modifier returnValueType methodName (list of parameters)

{ Method body }

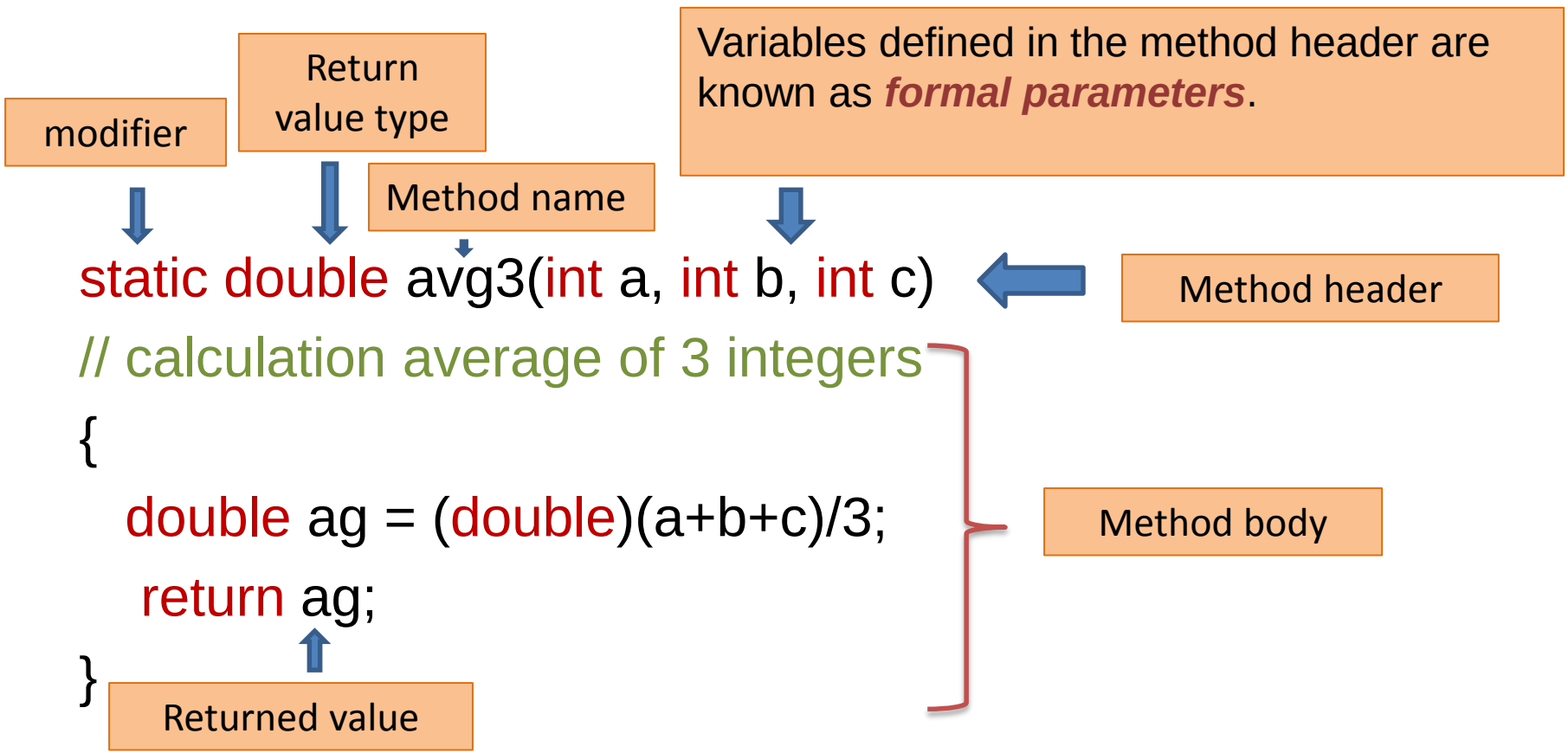


Return Value Type

A method may return a value. The ***returnValueType*** is the data type of the value the method returns. If the method does not return a value, the returnValueType is the keyword ***void***.



Example 1 - defining method



This method takes three parameters : a ,b and c and returns their average ag.

Example 1 - calling a method

```
int grade1,grade2,grade3 ; //  student's grades
double avGrade; // average grade
System.out.println("Enter 3 grades ");
grade1 = reader.nextInt();
grade2 = reader.nextInt();
grade3 = reader.nextInt();
```

When a method is invoked, you pass a value to the parameter.
This value is referred to as ***actual parameter*** *or argument*.

```
avGrade = avg3(grade1,grade2,grade3);
System.out.println( "The avarage grade is :" + avGrade);
```

Choice 1

```
System.out.println( "The avarage grade is :" + avg3(grade1,grade2,grade3));
```

Choice2

Example 2 - defining method

```
static boolean test(String str)
{
    boolean flag = true; // help variable
    int first = 0; // first letter
    int last = str.length( ) - 1; // last letter
    while (first<last  &&  flag) //goes to the middle letter
    {
         if (str.charAt(first) != str.charAt(last))
              flag = false;
        else
         {
              first++;
              last--;
         }
    }  //while
    return flag;
} // test
```

This method takes the string **str** as parameter and check it :
If **str** is **palindrome** the method returns **true**, **false** otherwise.

Example 2 - calling a method

```
public static void main(String[ ] args)
{
    int sum = 0; // number of palindromes
    for( int i = 0;i < 10;i++)
    {
        System.out.println("Enter the string : ");
        String   s = reader.next();
        if ( test(s) )
            sum++;
    }  //for
    System.out.println("The number of palindromes is :" + sum);
 } //main
```

This ***main method*** reads 10 strings and finds the number of palindromes.

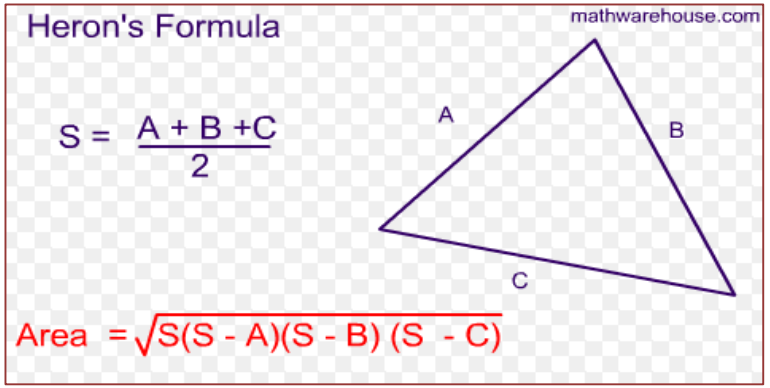
Calling the method **test**.

Example 3 - triangle area

This program calculates triangle area using Heron's formula.

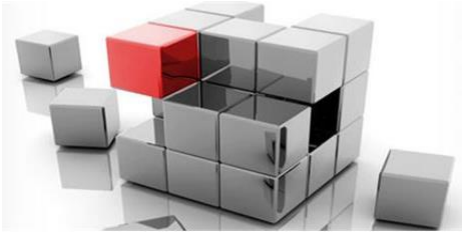
```
public static double distance(double x1, double y1, double x2, double y2)
{
        return Math.sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
} //distance

public static double heron(double a, double b, double c)
{
        double p = (a+b+c)/2;
        return Math.sqrt(p*(p-a)*(p-b)*(p-c));
} // heron
```



Example 3 - calling a methods

```
public static void main(String[ ] args)
{
    System.out.print("Input point 1 as x y: ");
    double x1 = reader.nextDouble();
    double y1 = reader.nextDouble();
    System.out.print("Input point 2 as x y: ");
    double x2 = reader.nextDouble();
    double y2 = reader.nextDouble();
    System.out.print("Input point 3 as x y: ");
    double x3 = reader.nextDouble();
    double y3 = reader.nextDouble();
    double a = distance(x1, y1, x2, y2);
    double b = distance(x1, y1, x3, y3);
    double c = distance(x3, y3, x2, y2);
    System.out.println("The area of the triangle is " + heron(a, b, c));
} // main
```



```
Input point 1 as x y: 2 1
Input point 2 as x y: 3 7
Input point 3 as x y: 0 4
The area of the triangle is 7.5
```

VOID Methods

- There are two ways to call a method; the choice is based on whether the method returns a value or not.
- **VOID** type of method does not return any value.

Example:

This method prints values of two integers before and after swap.

```
static void swap(int n1, int n2)
{
    System.out.println( "\t Inside the swap method" );
    System.out.println( "\t\t Before swapping n1 is " + n1 + " n2 is " + n2);
     int temp = n1; // help variable
     n1 = n2;
     n2 = temp;
     System.out.println( "\t\t After swapping n1 is " + n1 + " n2 is " + n2);
} // swap
```

Calling a void method

- A call to a void method **must be a statement**.
- This statement is like any Java statement terminated with a semicolon.

```
static void printGrade(int score)
{
        if (score >= 90)
                System.out.println('A');
        else
                if (score >= 80)
                        System.out.println('B');
                else
                        System.out.println('C');
 } // printGrade
public static void main(String[ ] args)
{
        printGrade(78); // call a void method
} //main
```

This would produce following result:

C

Calling a void method - example

```
static void revIntNum(int num)  {
    int reversNum = 0;
    do {
          int lastDigit = num%10;
          reversNum = (reversNum*10 )+ lastDigit;
          num = num/10;
    } while (num > 0);
    System.out.println(" That reversed number is : "+reversNum);
} //revIntNum
public static void main(String[ ] args)
{
      for( int i = 0 ;i < 10; i++ )  {
          System.out.print ("Enter an integer ");
          int x = reader.nextInt();
          revIntNum(x); // call a void method
      }  // for
} // main
```

This program reads 10 integers, reverses its digits mathematically and prints it.

Nesting of methods

A method can call more than one method in the same class .
If a method in Java calls a method in the same class it is called nesting of methods. First method can call the second method the second method can call a third method and so on.

```
public static void main(String[] args)
{
  .
  .
  . A( parameters)
  .
  .
} // main
```

```
A( )
{
  .
  .
  .
  B (parameters)
  .
  .
  .
  C (parameters)
  .
  .
  .
} // A method
```

```
B( )
{
  .
  .
  .
} // B method
```

```
C( )
{
  .
  .
  .
} // C method
```

When a method is called ,the flow of control transfers to that method.

When the method is done, control returns to the location where the call was made (next Java statement).

Nesting of methods - example

```
public class NestMethTest  {
    static void display(int x,int y) {
        System.out.println( "Value of X= " + x);
        System.out.println( "Value of Y= " + y);
    } //display
    static void add2(int a,int b) {
        a + = 2;
        b + = 2;
        display(a,b); //call display method
    } // add2
    static Scanner reader = new Scanner(System.in);
    public static void main(String[] args) {
        System.out.print( "enter first number->  " );
        int num1 = reader.nextInt(); // 4
        System.out.print( "enter second number->  " );
        int num2 = reader.nextInt(); // 7
        add2(num1,num2);
    }   //main
}  // class
```

Output will be displayed as:

```
enter first number -> 4
enter second number -> 7
Value of X= 6
Value of Y= 9
```

Passing Parameters by Values

- When calling a method, you need to provide arguments, which must be given in the *same order* and in the *same type* as their respective parameters in the method specification.

  This is known as *parameter order association*.
- When you invoke a method with a parameter, the value of the argument is passed to the parameter. This is referred to *as pass-by-value*. (העברה לפי ערך)The variable is not affected, regardless of the changes made to the parameter inside the method.
- For simplicity, Java programmers often say passing an argument x to a parameter y, which actually means *passing the value of x to y*.

Example - Passing Parameters by Values

Following program section demonstrates the effect of *passing by value*.
The **swap** method is invoked by passing two arguments.
The values of the arguments are not changed after the method is invoked.

```
int num1 = 1;
int num2 = 2;
System.out.println("Before swap method, num1 is " + num1 + " and num2 is " + num2);

swap(num1, num2);

System.out.println("After swap method, num1 is " + num1 + " and num2 is " + num2);
```

A call to a void method must be a statement.

```
Before swap method, num1  is  1 and num2  is  2
    Inside the swap method
            Before swapping  n1  is 1 n2  is  2
            After swapping    n1  is 2  n2  is  1
After swap method, num1  is  1 and num2 is  2
```

This would produce following result.

The Scope of Variables

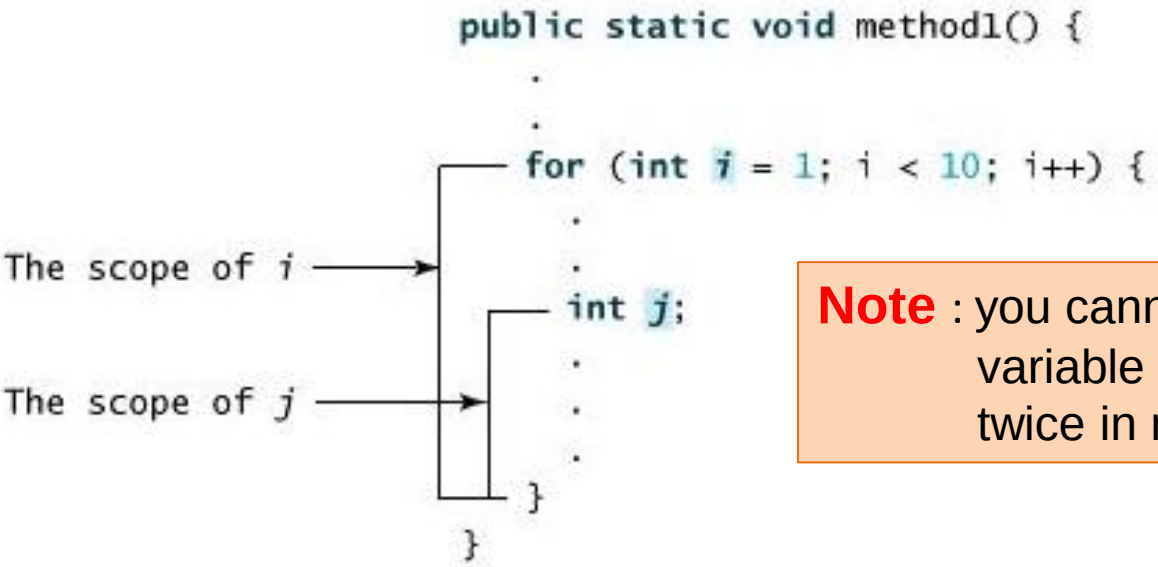
- The ***scope*** (*טווח הכרה*) of a variable is the part of the program where the variable can be referenced. A variable defined inside a method is referred to as a *local variable* (*משתנים מקומים*).
- The ***scope*** of a local variable starts from its declaration and continues to the end of the block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

  A local variable must be declared before it can be used.

- A parameter is actually a local variable. The ***scope*** of a method parameter covers the *entire method*.
- You can declare a local variable with the same name multiple times in different non-nesting blocks in a method, but you cannot declare a local variable twice in nested blocks.

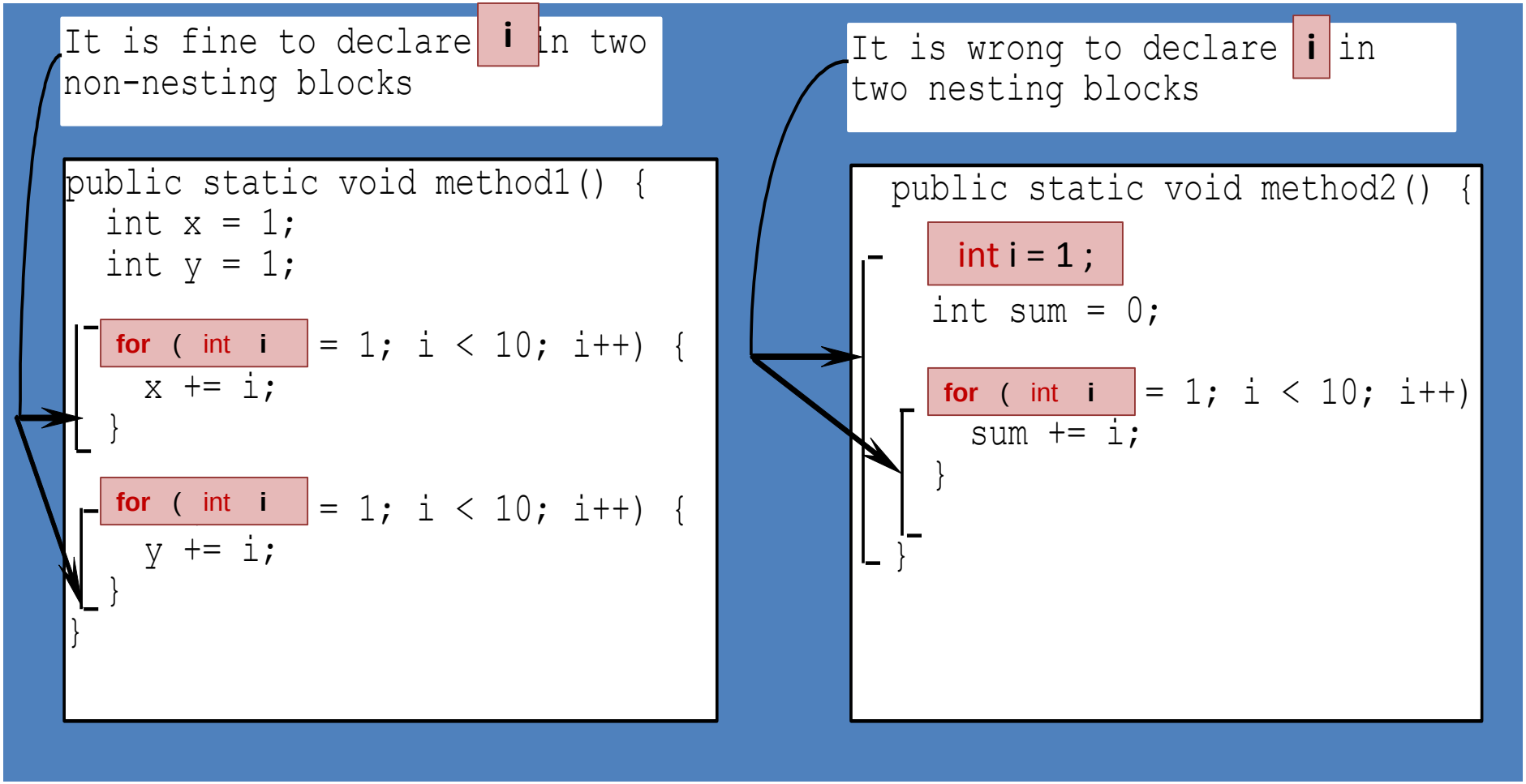
The Scope of Variables

- A variable declared in the initial action part of a for loop header has its scope in the entire loop.
- But a variable declared inside a for loop body has its scope limited in the loop body *from its declaration to the end of the block* that contains the variable as shown below:



Note : you cannot declare a local variable with the same name twice in nested blocks.

Scope of Local Variables, cont.



Passing Arrays to methods

- Copies of argument values are sent to the method, where the copy is manipulated and in certain cases, one value may be returned. While the copied values may change in the method, the original values in main method **did not change**.

- The situation, when working with arrays, *is somewhat different*. If we were to make copies of arrays to be sent to methods, we could potentially be copying very large amounts of data. Not very efficient!

- Arrays are ***passed-by-reference.*** (***העברה לפי כתובתה***)

  Passing-by-reference means that when an array is passed as an argument, its **memory address location** is actually passed, referred to as its "reference". In this way, the contents of an array **can be changed** inside of a method, since we are dealing directly with the actual array and not with a copy of the array.

Passing Arrays to methods

```
public static void printArray(int[ ] array)
 {
  for (int i = 0; i < array.length; i++)
    System.out.print(array[i] + " ");
 }
```

Invoke the method printArry:

```
int [ ] list = { 3, 1, 2, 6, 4, 2 };
printArray(list);
```

Choice 1

Invoke the method printArray

```
printArray(new int[ ] { 3, 1, 2, 6, 4, 2 });
```

Choice 2

Anonymous array

Anonymous Array

The statement

```
printArray( new int[ ] { 3, 1, 2, 6, 4, 2 } );
```

creates an array using the following syntax:

```
new dataType[ ] { literal0, literal1, ..., literalk } ;
```

There is no explicit reference variable name for the array.

Such array is called an **anonymous array**.

Example

```
public static void main(String[ ] args)
{
   int x = 2; // x represents an int value
   int [ ] y = int { 89,20,37 }; // y represents an array of int values
   mEx(x, y); // Invoke mEx with actual arguments x and y
   System.out.println("x is : " + x);
   System.out.println("y[0] is : " + y[0]);
} // main
```

To call a method that takes an array as argument, simply type the name of the array in the parentheses of the called method.

This would produce

x is : 2
y[0] is 55

```
 public static void mEx(int number, int[ ] numbers)
{
     number = 100; // Assign a new value to formal parameter number
     numbers[0] = 55; // Assign a new value to numbers[0] ⟺ y[0]
 } // mEx
```

Returning an Array from a method

Like a normal variable, an array can be returned from a method. This means that the method would return a variable (memory address) that carries various values.

For example:

The method ***reverse*** returns an array that is the reversal of another array:

```
public static int [ ]  reverse(int[ ] list)
{
   int [ ] result = new int[ list.length ];
      for ( int j = 0, int i = result.length - 1; i >= 0; i-- , j++ )
                                                result[ j ] = list[ i ];
   return result ;
} //reverse
```

When declaring the method, you must specify its data type.

When the method ends, it would return an array represented *by the name* of its variable.

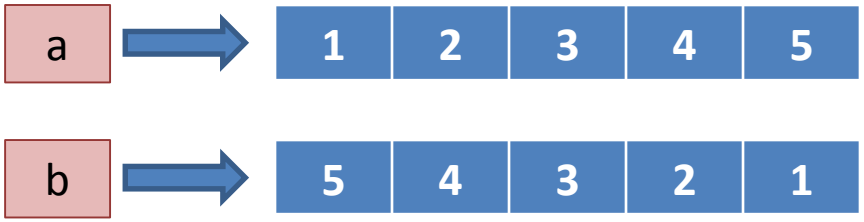
Example

```
public static void main(String[ ] args)
{
    int [ ] a = { 1,2,3,4,5 }; // a represents an array of int values
    printArr(a); // invoke method printArr with actual parameter a
    int [ ] b = reverse(a); // b array declaration and invoke method  revers
    printArr(b); // invoke method printArr with actual parameter b
} // main
```

This would produce :

1 2 3 4 5

5 4 3 2 1



NOTE: Declaration of a reference to an integer array *is not the same* as declaring an array.
A reference (הפניה) is simply a pointer to an array.

Last example

Next program reads student's grades and calculates their average grades.

If student's grade is less than the average grade, then program adds 5 points factor to this grade.

The program prints all entered grades before and after upgrading.

```
public static void main(String[ ] args)
{
      System.out.print( "Enter number of students : “ );
      int num = reader.nextInt();  // student's number input
      int[ ] arrGrades = inputGrades(num); // input student's grades
      printGrades(arrGrades); // print grades before update
      int avg = avgGrades(arrGrades); // calculate average grade
      updGrades(arrGrades,avg); // update student's grades
      printGrades(arrGrades); // print grades after update
}  // main
```

Method inputGrades

```
static int [ ] inputGrades(int n1)
{
      int grade; // student grade
      int [ ] a = new int[n1]; // array of grades
      int j = 0; // array index place holder
      while ( j < n1)
      {
          do
          {
              System.out.print("Enter the grades : ");
              grade = reader.nextInt();
          }  while (grade < 0 || grade > 100);
          a[ j++ ] = grade;
      } // outer while
      return a;
} // method inputGrades
```

Rest methods

```
static int avgGrades(int [ ] b)
{
    int sum = 0; // sum of grades
    for(int i = 0;i < b.length;i++)
    sum+ = b[i];
    return sum / b.length;
} // avgGrades
```

```
static void updGrades(int[ ] c, int avg)
{
        for(int i =0;i < c.length;i++)
            if(c[i] < avg)
                c[i]+ = 5;
} // updGrades
```

```
static void printGrades(int[ ] d)
{
    for(int i = 0;i < d.length; i++)
        System.out.println(" The "+ (i+1) + " student's  grade is " + d[i]);
} //printGrades
```

The main method

- The **main method** must be declared public and static, it must not return any value, and it must accept a String array as a parameter. The method declaration must look like the following:

public static void main(String[ ] args)

- Sometimes you will want to pass information into a program when you run it. This is accomplished by passing command-line arguments to **main( ).**
- A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy. They are stored as strings in the String array passed to **main( ).**

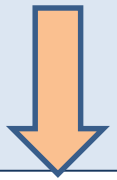
Java method overloading

In Java method **overloading** (**העמסה**) means creating more than a single method *with same name* with *different signatures (חתימה /כותרת הפעולה).*

For example:

```
public class Overload {
    public static void test(int a) {
            System.out.println("a: " + a);
    } // test
    public static void test(int a, int b) {
             System.out.println("a and b: " + a + "," + b);
    } // test
    public static double test(double a) {
             System.out.println("double a: " + a);
             return a*a; }
    } // test
```

Next slide cont.



Java method overloading, cont.

```
  public static void main(String args [ ])
  {
        double result;
        test(10);
        test(10, 20);
        result = test(5.5);
        System.out.println("Result : " + result);
  } // main
} // Overload
```

Output will be displayed as:

a :10
a and b : 10,20
double a : 5.5
Result : 30.25

Java method overloading, cont.

As you can see, method **test( )** is overloaded **three times**.

The first version takes one integer parameter:

```
public static void test(int a)
```

The second takes two integer parameters:

```
public static void test(int a, int b)
```

The third takes one double parameter:

```
public static double test(double a)
```

When an overloaded method is called, Java looks for :

- A match between the arguments used to call the method
- The method's parameters.

However, this match need not always be exact.