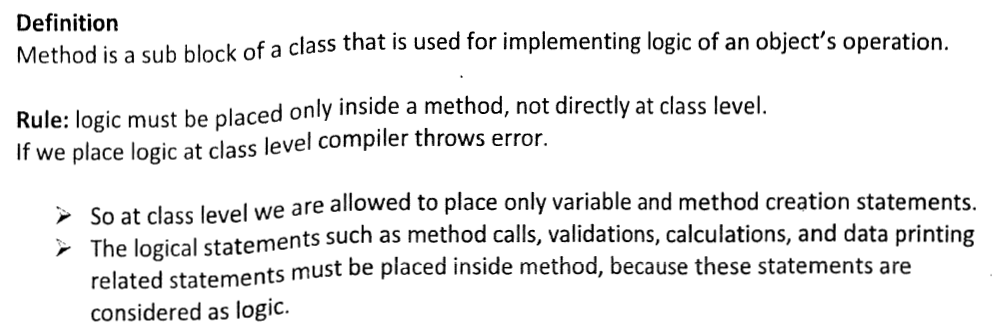
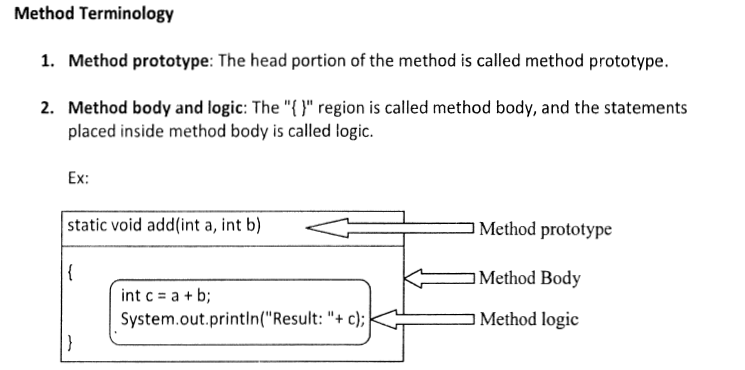
**Java methods**

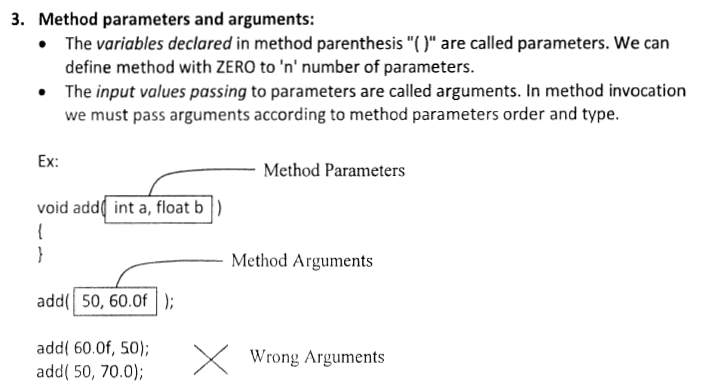
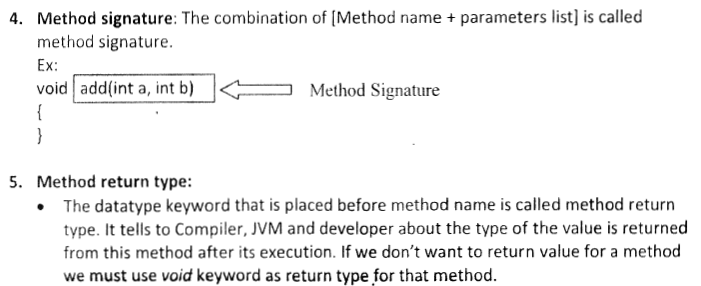
The procedures and functions are known as **methods** in Java.

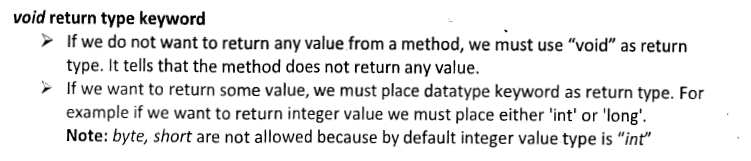
The methods scope is for the whole class. Now here, you can make a small difference. A function in C-lang can be called without object but a method is to be called with an object.

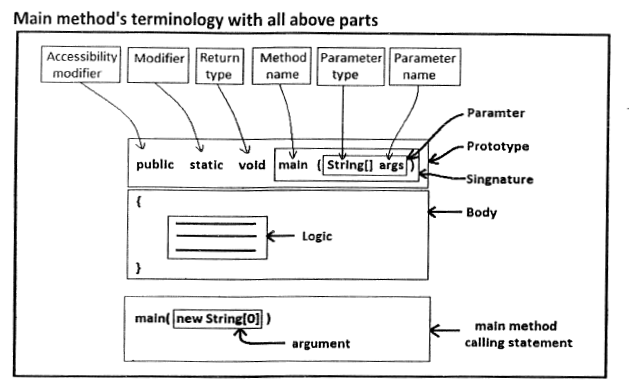
The first line of a method declaration is known as "**method signature**". A Java method signature may comprises of an access specifier, one or two optional access modifiers, return type, name, optional parameter list and also optional exceptions the method can throw. The parameter or a return type can be a variable, an object or an array.

****

****

** **

****

****

**Following program illustrates.**

*public class Employee{*

*public void nature(){*

*System.out.println("Hard working");*

*}*

*public int total(int basic, int da, int hra){*

*int sum = basic + da + hra;*

*return sum;*

*}*

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

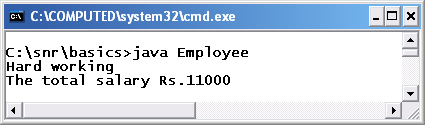
*Employee emp1 = new Employee();*

*emp1.nature();*

*int salary = emp1.total(8000, 2000, 1000);*

*System.out.println("The total salary Rs." + salary);*

*}  
}*



A method in a class defines the behavior of the objects.

A method is a named block of code.

The code that invokes the method is the caller of the method.

Optionally, a method may accept input values from the caller and it may return a value to the caller.

The list of input values is known as parameters. A method may have zero parameters. If a method has zero parameters, we say that method does not have any parameters or method does not take any parameters.

A method is always defined inside the body of a class.

The general syntax for a method declaration is of the form

*<modifiers> <return type> <method name> (<parameters list>) <throws clause>{*

*// Body of the method goes here*

*}*

* <modifiers> is an optional list of modifiers;
* <return type> is the data type of the value returned from the method;
* <method name> is the name of the method.

**Method Parameters**

The method name is followed by a pair of opening and closing parentheses.

Optionally, we can specify one or more parameters to the method within the parentheses.

Multiple parameters are separated by a comma.

The closing parenthesis may optionally be followed by a throws clause.

Finally, we specify the code for the method inside opening and closing braces.

**The four parts in a method declaration are mandatory:**

* the return type,
* method name,
* a pair of opening and closing parentheses, and
* a pair of opening and closing braces.

**The following is an example of a method:**

* it is named add;
* it takes two parameters of type int named n1 and n2, and
* it returns their sum:

*int add(int n1, int n2) {*

*int sum = n1 + n2;*

*return sum;*

*}*

* Sometimes, a method does not return a value to its caller. The keyword void is used as the return type if a method does not return any value to the caller.
* The method name must be a valid Java identifier.
* Conventionally, a Java method starts with a lowercase and subsequently a word cap is used.
* For example, getName, setName, getDogCount, and createDog are valid method names.
* A method may take input values from its caller. A parameter is used to take an input value from the caller.
* A parameter consists of two parts: a data type and a variable name. A method parameter is a variable declaration.
* The variables are used to hold the input values that are passed from the method's caller. A comma separates two parameters of a method.
* In the following example, the add method declares two parameters, n1 and n2. Both parameters are of the int data type.

*int add(int n1, int n2) {*

*int sum = n1 + n2;*

*return sum;*

*}*

When the add method is called, the caller must pass two int values.

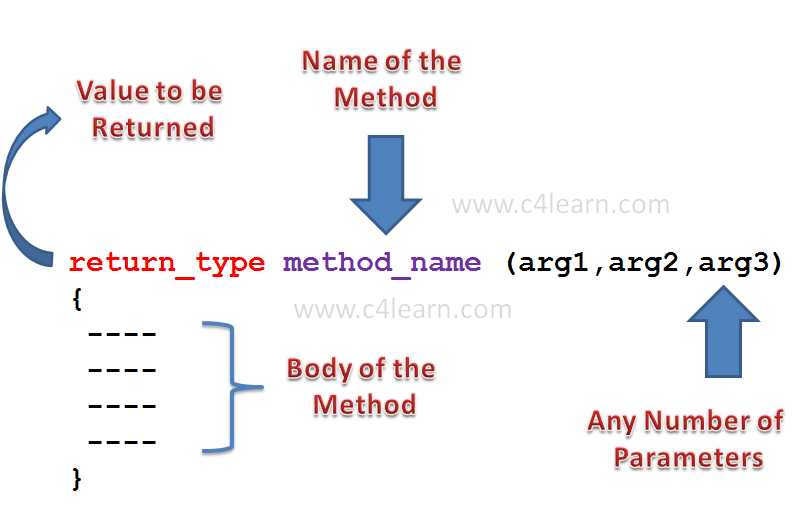
The first value passed from the caller is stored in n1, and the second value passed from the caller is stored in n2.

The parameters n1 and n2 are also known as formal parameters.

A method is uniquely identified by its signature in a particular context. The signature of a method is the combination of its name and its parameter's number, types, and order.

**Introducing Methods in Java Class : Class Concept in Java**

1. In Java Class, We can add user defined method.
2. Method is equivalent to Functions in C/C++ Programming.



**Syntax : Methods in Java Classes**

*return\_type method\_name (arg1 , arg2 , arg3)*

1. **return\_type** is nothing but the value to be returned to an calling method.
2. **method\_name** is an name of method that we are going to call through any method.
3. **arg1, arg2, arg3** are the different parameters that we are going to pass to a method.

**Return Type of Method :**

1. Method can return any type of value.
2. Method can return any Primitive data type

***int*** *sum (****int*** *num1,int num2);*

1. Method can return Object of Class Type.

*Rectangle sum (****int*** *num1,int num2);*

1. Method sometimes may not return value.

***void*** *sum (****int*** *num1,int num2);*

**Method Name :**

1. Method name must be valid identifier.
2. All [Variable naming rules](http://www.c4learn.com/naming-rule-conventions-variables-in-java-programming.html) are applicable for writing Method Name.

**Parameter List :**

1. Method can accept any number of parameters.
2. Method can accept any data type as parameter.
3. Method can accept Object as Parameter
4. Method can accept no Parameter.
5. Parameters are separated by Comma.
6. Parameter must have Data Type

**Live Example : Introducing Method in Java Class**

*class Rectangle {*

***double*** *length;*

***double*** *breadth;*

***void*** *setLength(****int*** *len) {*

*length = len;*

*}*

*}*

*class RectangleDemo {*

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

*Rectangle r1 = new Rectangle();*

*r1.length = 10;*

*System.out.println("Before Function Length : " + r1.length);*

*r1.setLength(20);*

*System.out.println("After Function Length : " + r1.length);*

*}*

*}*

**Output :**

java RectangleDemo

Before Function Length : 10.0

After Function Length : 20.0

## Explanation :

### Calling a Method :

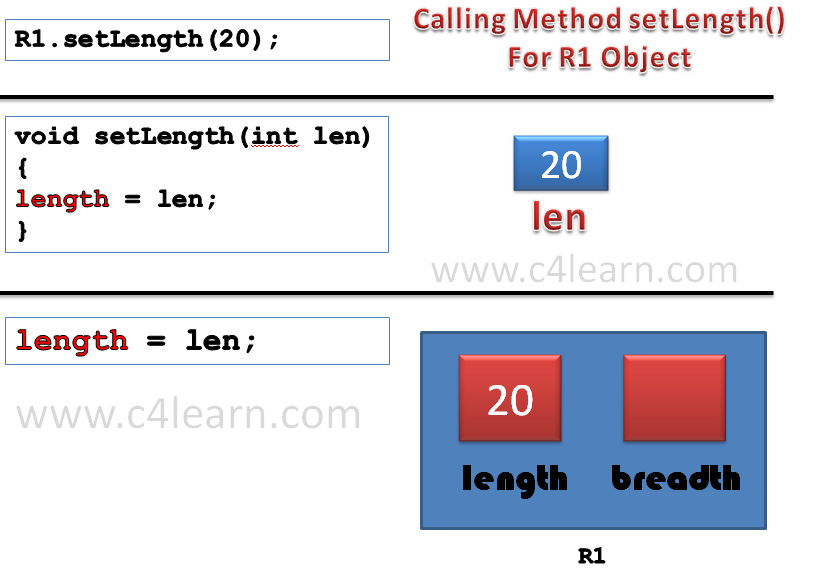
1. “**r1**” is an Object of Type Rectangle.
2. We are calling method “**setLength()**” by writing –

*Object\_Name [DOT] Method\_Name ( Parameter List ) ;*

1. Function call is always followed by **Semicolon**.

### Method Definition :

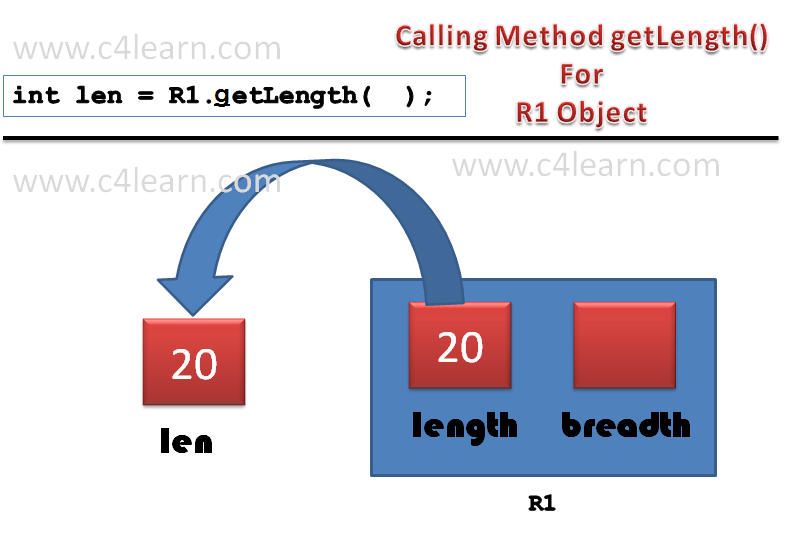
1. Method definition contains the **actual body of the method**.
2. Method **can take parameters and can return a value.**



**Java returning value from method**

**Returning Value From the Method :**

1. We can specify return type of the method as **“Primitive Data Type” or “Class name”**.
2. Return Type can be “Void” means **it does not return any value**.
3. Method can return a value by using “**return**” keyword.



## Example:

*public class Employee{*

*public void nature() {*

*System.out.println("Hard working");*

*}*

*public int total(int basic, int da, int hra) {*

*int sum = basic + da + hra;*

*return sum;*

*}*

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

*Employee emp1 = new Employee();*

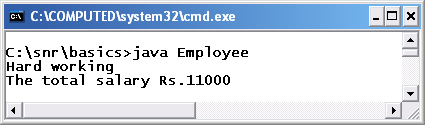
*emp1.nature();*

*int salary = emp1.total(8000, 2000, 1000);*

*System.out.println("The total salary Rs." + salary);*

*}*

*}*



## Live Example : Returning Value from the Method

*class Rectangle {*

***int*** *length;*

***int*** *breadth;*

***void*** *setLength(****int*** *len) {*

*length = len;*

*}*

***int*** *getLength() {*

***return*** *length;*

*}*

*}*

*class RectangleDemo {*

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

*Rectangle r1 = new Rectangle();*

*r1.setLength(20);*

***int*** *len = r1.getLength();*

*System.out.println("Length of Rectangle : " + len);*

*}*

*}*

## *O*utput :

***java RectangleDemo***

*Length of Rectangle : 20*

### There are two important things to understand about returning values :

1. The **type of data returned by a method must be compatible with the return type specified by the method**.

**For example**, if the return type of some method is boolean, you could not return an integer.

*boolean getLength() {*

***int*** *length = 10;*

***return****(length);*

*}*

1. **The variable receiving the value returned by a method (such as len, in this case) must also be compatible with the return type specified for the method**.

***int*** *getLength(){*

***return*** *length;*

*}*

*boolean len = r1.getLength();*

1. **Parameters should be passed in sequence and they must be accepted by method in the same sequence**.

***void*** *setParameters(String str,****int*** *len){*

*…*

*}*

*r1.setParameters(12,"Pritesh");*

**Instead it should be like –**

***void*** *setParameters(****int*** *length,String str){*

*…*

*}*

*r1.setParameters(12,"Pritesh");*

**Java passing object as parameter**

**Passing Object as Parameter :**

**package** com.pritesh.programs;

**class** Rectangle {

int length;

int width;

**Rectangle**(int l, int b) {

length = l;

width = b;

}

void area(**Rectangle** r1) {

int areaOfRectangle = r1.length \* r1.width;

**System**.out.println("Area of Rectangle : " + areaOfRectangle);

}

}

**class** RectangleDemo {

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

**Rectangle** r1 = **new** **Rectangle**(10, 20);

r1.area(r1);

}

}

**Output of the program :**

*Area of Rectangle : 200*

**Explanation :**

1. We can pass Object of any class as parameter to a method in java.
2. We can access the instance variables of the object passed inside the called method.

*area = r1.length \* r1.width*

1. It is good practice to initialize instance variables of an object before passing object as parameter to method otherwise it will take default initial values.

#### Note: In java, Methood Overloading is not possible by changing the return type of the method.

**Different Ways of Passing Object as Parameter :**

**Way 1 : By directly passing Object Name**

*void area(Rectangle r1) {*

***int*** *areaOfRectangle = r1.length \* r1.width;*

*System.out.println("Area of Rectangle : " + areaOfRectangle);*

*}*

*class RectangleDemo {*

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

*Rectangle r1 = new Rectangle(10, 20);*

*r1.area(r1);*

*}*

**Way 2 : By passing Instance Variables one by one**

*package com.pritesh.programs;*

*class Rectangle {*

***int*** *length;*

***int*** *width;*

***void*** *area(****int*** *length,* ***int*** *width) {*

***int*** *areaOfRectangle = length \* width;*

*System.out.println("Area of Rectangle : "+ areaOfRectangle);*

*}*

*}*

*class RectangleDemo {*

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

*Rectangle r1 = new Rectangle();*

*Rectangle r2 = new Rectangle();*

*r1.length = 20;*

*r1.width = 10;*

*r2.area(r1.length, r1.width);*

*}*

*}*

Actually this is not a way to pass the object to method. but this program will explain you how to pass instance variables of particular object to calling method.

**Way 3 : We can pass only public data of object to the Method.**

Suppose we made width variable of a class private, then we cannot update value in a main method, since it does not have permission to access it.

*private* ***int*** *width;*

after making width private –

*class RectangleDemo {*

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

*Rectangle r1 = new Rectangle();*

*Rectangle r2 = new Rectangle();*

*r1.length = 20;*

*r1.width = 10;*

*r2.area(r1.length, r1.width);*

*}*

*}*

**Java returning object**

**Returning the Object From Method**

In Java Programming A method can return any type of data, including class types that you create.

For example, in the following program, the **getRectangleObject( )** method returns an object.

**Java Program : Returning the Object From Method**

***package*** *com.pritesh.programs;*

***import*** *java.io.File;*

***import*** *java.io.IOException;*

***class*** *Rectangle {*

*int length;*

*int breadth;*

***Rectangle****(int l,int b) {*

*length = l;*

*breadth = b;*

*}*

***Rectangle*** *getRectangleObject() {*

***Rectangle*** *rect =* ***new******Rectangle****(10,20);*

***return*** *rect;*

*}*

*}*

***class*** *RetOb {*

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

***Rectangle*** *ob1 =* ***new******Rectangle****(40,50);*

***Rectangle*** *ob2;*

*ob2 = ob1.getRectangleObject();*

***System****.out.println("ob1.length : " + ob1.length);*

***System****.out.println("ob1.breadth: " + ob1.breadth);*

***System****.out.println("ob2.length : " + ob2.length);*

***System****.out.println("ob2.breadth: " + ob2.breadth);*

*}*

*}*

**Output of the Program :**

*ob1.length : 40*

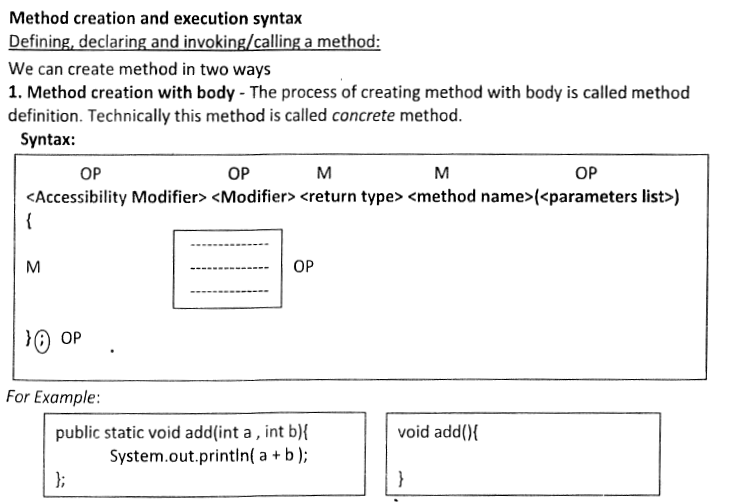
*ob1.breadth: 50*

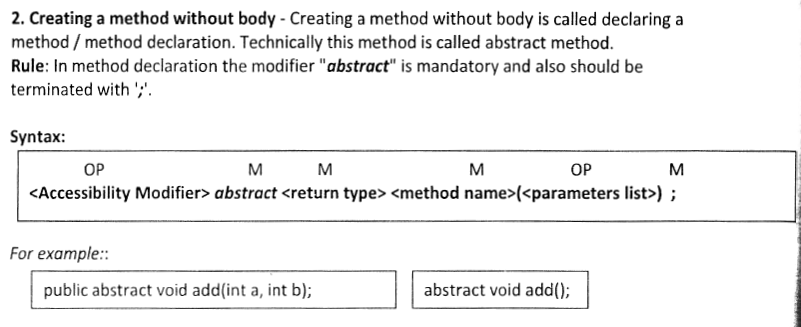
*ob2.length : 10*

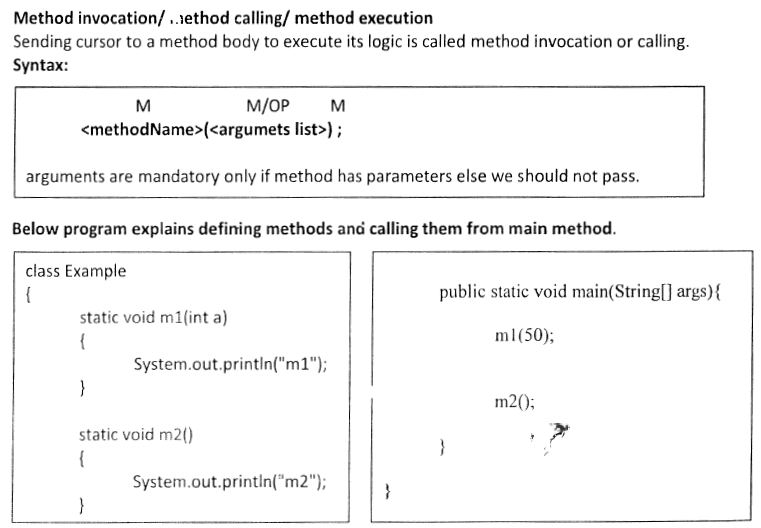
*ob2.breadth: 20*

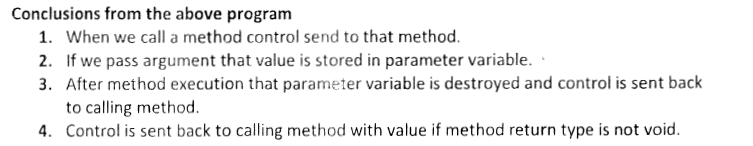
**Explanation :**

1. In the above program we have called a method **getRectangleObject**() and the method creates object of class from which it has been called.
2. All objects are dynamically allocated using **new**, you don’t need to worry about an object going out-of-scope because the method in which it was created terminates.
3. The object will continue to exist as long as there is a reference to it somewhere in your program. When there are no references to it, the object will be reclaimed the next time garbage collection takes place.

****

****

****

****

**User defined methods**

In Java we can also declare our own methods

Every method contains following 2 parts

1. Method declaration/ method prototype / method header

2. Method body/method definition/method implementation

**1. Method declaration/ method prototype / method header**

Method declaration means we are going to decide what is the name of the method , what are the parameters it will take and what kind of value is return by the method.

**syntax:**

returntype methodname(list of parameters);

**Returntype**

- returntype is data type that indicates what type of value is return by the particular method.

-returntype can be any primitive type or array type or reference type

-If method does not return any value then return type must be specified using a java keyword called " void ".

-Specifying returntype is mandatory

**Methodname**

-To identify and access the method there is a suitable name is given for a method which is called as method name.

-A methodname can be any valid java identifier.

-Specifying method name is mandatory

**List of parameters**

-Parameters are the variables that will receive the values that are passed into the. particular method on which data method will perform the operations.

- We can write 0 or more number of parameters of any primitive type or array type or reference type

- Specifying parameters is optional.

**2. Method body/method definition/method implementation**

-Method definition means we are going to write the group of statements that are executed by the method.

-A method definition can be written in between a pair of curry braces

**syntax:**

*returntype methodname(list of parameters){*

*//statements;*

*return value;*

*}*

-Here we can write 0 or more number of statements in between the pair of curly braces.

-When we write 0 statements then it is called as null implementation

-If the return type is specified as other than void then we must return a value from our method using java keyword called "**return**".

**Syntax:**

return value;

- The datatype of the value we return must be match with the datatype that we specify as return type.

- But if return type specified as void then we must not write any return value statement.

- In java we can create any number of methods which are in any of the following 4 combinations of methods

1. method without return type, without parameters

2. method without return type, with parameters

3. method with return type, without parameters

4. method with return type, with parameters

**Method Invocation**

Method invocation is a process of calling the method and executing the logic written within the method.

**Syntax:**

methodname(list of parameters);

**1. To design a method without return type, without parameters**

*class Method1{*

***//method declaration and definition***

*void add(){*

*int a,b,c;*

*a=10;*

*b=20;*

*c=a+b;*

*System.out.println("Addition="+c);*

*}*

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

*Method1 m = new Method1();*

*m.add();* ***//method invocation***

*m.add();*

*}*

*}*

**2. To design a method without return type, with parameters**

*class Method2{*

*void add(int x,int y){*

*int a,b,c;*

*a=x;*

*b=y;*

*c=a+b;*

*System.out.println("Addition="+c);*

*}*

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

*Method2 m = new Method2();*

*m.add(67,56);*

*m.add(789,567);*

*int p=10,q=90;*

*m.add(p,q);*

*}*

*}*

**3. To design a method with return type, without parameters**

*class Method3{*

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

*Method3 m = new Method3();*

*int res = m.add();*

*System.out.println("Addition1="+res);*

*System.out.println("Addition2="+m.add());*

*}*

*int add(){*

*int a,b,c;*

*a=10;*

*b=20;*

*c=a+b;*

*return c;*

*}*

*}*

1. **To design a method with return type, with parameters**

*class Method4{*

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

*Method4 m = new Method4();*

*int res = m.add(78,90);*

*System.out.println("Addition1="+res);*

*System.out.println("Addition2="+m.add(90,34));*

*}*

*int add(int x,int y){*

*int a,b,c;*

*a=x;*

*b=y;*

*c=a+b;*

*return c;*

*}*

*}*

**Using Variables from Methods**

An object is required to call instance variables but not local. You know, C/C++ global variables in Java are called as instance variables

**Example:**

*public class VariablesWithMethods{*

*int a = 100;*

*public void show() {*

*int b = 200;*  ***// calling instance variable a***

*System.out.println("a value from show(): " + a);*

***// calling local variable b***

*System.out.println("b value from show(): " + b);*

*a = 300;*

*}*

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

*VariablesWithMethods vwm = new VariablesWithMethods();*

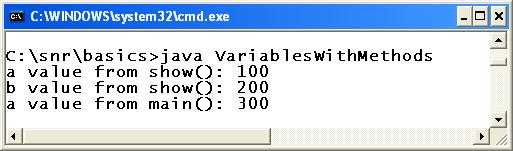
*vwm.show();*

*System.out.println("a value from main(): " + vwm.a);*

***// prints 300***

*}*

*}*



###### **Java behavior in using variables from methods**

Whenever a method is called with an object, like **vwm.show()**, the instance variables inside the method are linked with the object implicitly. For this reason, the a called from show() becomes **vwm.a**. One more principle is, whenever an instance variable is called with an object, a global location is created for **vwm.a**. This location even though created through**show()** method, the location can be accessed by any other method, but with **vwm** object only. The location a is private to object **vwm** only. This is how [encapsulation](http://way2java.com/oops-concepts/three-great-principles-%e2%80%93-data-binding-data-hiding-encapsulation/) is maintained. If another object is there like **vwm1** and called the **show()** as **vwm1.show()**, then there creates a location on the name of **vwm1.a** and this location is private to **vwm1**.

Now a value in **show()** method is changed from 100 to 300. But **a** is nothing but **vwm.a**. Earlier value in the location is 100 and now changed to 300. By the time control comes to**vwm.a** in main() method, the value of **vwm.a** is already changed. For this reason, **vwm.a** from main() method prints 300.

**//wap to design a method with array as parameter**

*class Method5{*

*void showArray(int ar[]){*

*for(int v:ar){*

*System.out.print(v+" ");*

*}*

*System.out.println();*

*}*

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

*Method5 m = new Method5();*

*int arr[] = {89,89,78,67,56,56,78};*

*m.showArray(arr);*

*m.showArray(new int[]{56,56,67});*

***//anonymous array***

***// m.showArray(10,20,30,40); -invalid***

*}*

*}*

**//wap to design a method with array as return type**

class Method6{

int[] getArray(){

int arr[] = {89,89,78,67,56,56,78};

return arr;

}

public static void main(String args[]){

Method6 m = new Method6();

int ar[]= m.getArray();

for(int v:ar){

System.out.print(v+" ");

}

}

}

**//wap to design a method with class or reference as a parameter.**

*class Employee{*

*int empno;*

*String ename;*

*}*

*class Method7{*

*void showEmployee(Employee e){*

*System.out.println("EMPNO:"+e.empno);*

*System.out.println("ENAME:"+e.ename);*

*}*

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

*Method7 m = new Method7();*

*Employee eee = new Employee();*

*eee.empno=7;*

*eee.ename="sehwagh";*

*m.showEmployee(eee);*

*}*

*}*

**//wap to design a method with class or reference as a return type.**

*class Employee{*

*int empno;*

*String ename;*

*}*

*class Method8{*

*Employee getEmployee(){*

*Employee eee = new Employee();*

*eee.empno=7;*

*eee.ename="sehwagh";*

*return eee;*

*}*

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

*Method8 m = new Method8();*

*Employee e = m.getEmployee();*

*System.out.println("EMPNO:"+e.empno);*

*System.out.println("ENAME:"+e.ename);*

*}*

*}*

**Method Recursion**

- Method Recursion is a process of calling a method within itself

- Method Recursion we mainly use for achieving reverse algorithm

**Eg:**

games, iterating any data structures,....

**Note:**

- In Method Recursion there is a chance to enter into infinite loop and it wil throw a Runtime exception saying StackOverFlowError

**//wap to demo on method recursion**

*class Method9{*

*void show(int n){*

*if(n==0){*

*return;*

*}*

*System.out.println(n);*

*show(n-1);*

*}*

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

*Method9 m = new Method9();*

*m.show(10);*

*}*

*}*

**//wap to calc factorial of given number using method recursion**

*class Method10{*

*int fact(int n){*

*if(n==1){*

*return 1;*

*}*

*else{*

*return n\*fact(n-1);*

*}*

*}*

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

*Method10 m = new Method10();*

*int res = m.fact(5);*

*System.out.println("5!="+res);*

*}*

*}*

**//wap to display all armstring numbers using methods**

*class Method11{*

*boolean isArmStrong(int n){*

*int r,sum=0,m=n;*

*while(n>0){*

*r=n%10;*

*sum=sum+r\*r\*r;*

*n=n/10;*

*}*

*if(sum==m){*

*return true;*

*}*

*else{*

*return false;*

*}*

*}*

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

*Method11 m = new Method11();*

*for(int i=1;i<=1000000;i++){*

*if(m.isArmStrong(i)==10){*

*System.out.println(i);*

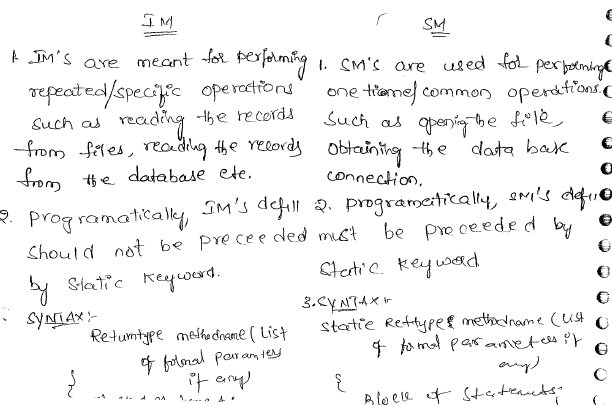
*}*

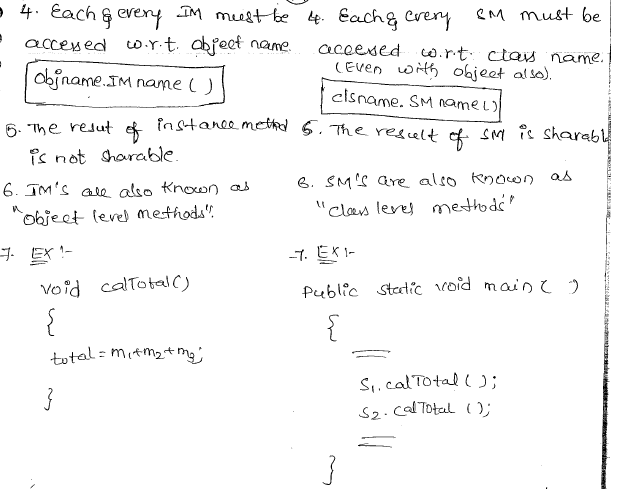
*}*

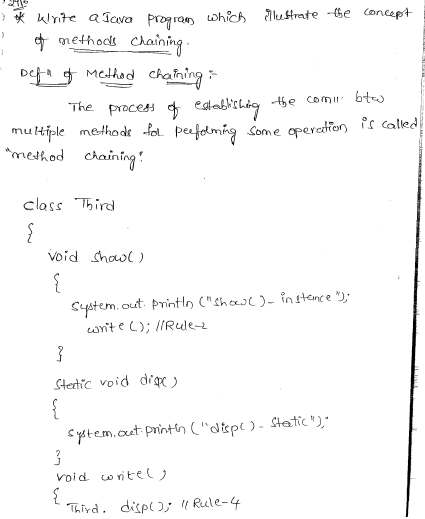
*}*

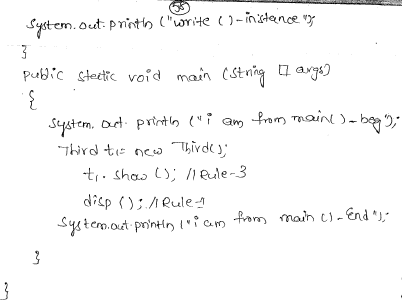
*}*

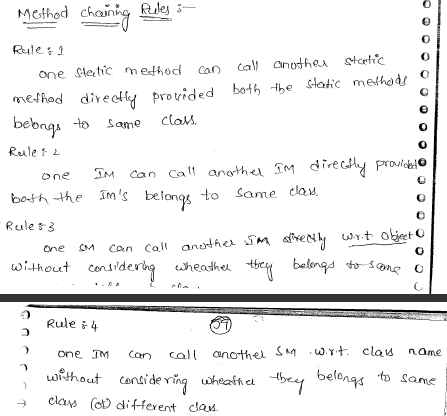
**Instance Methods Vs Static Methods:**

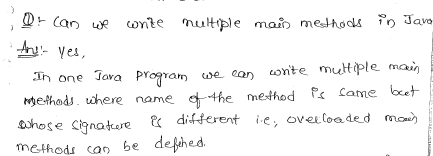
****

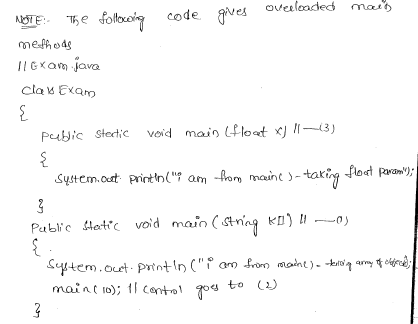
****

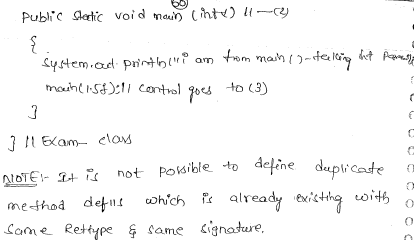
****

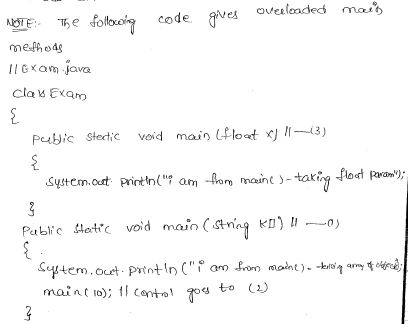
****

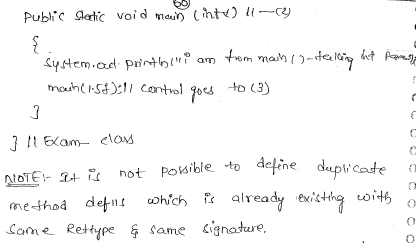
****

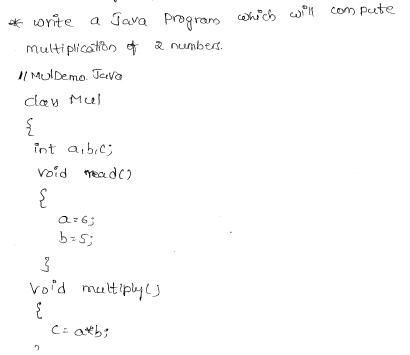
****

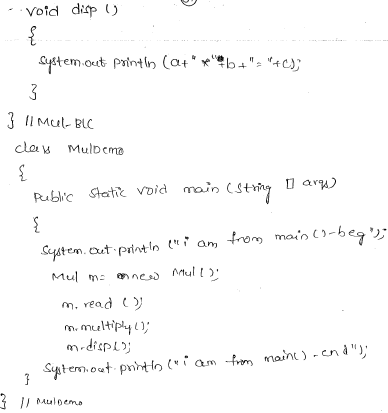
****

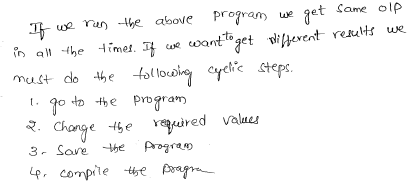
****

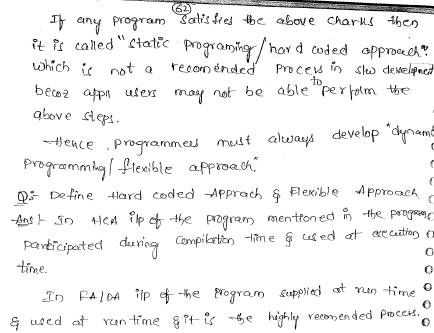
****

****

****

****

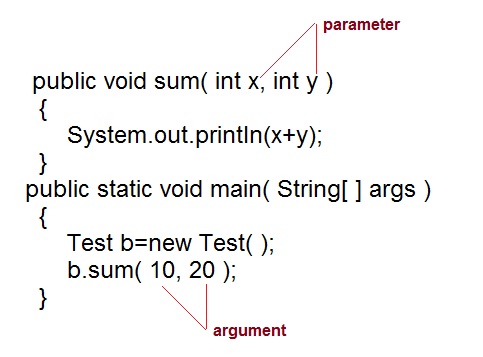
****

****

#### Parameter Vs. Argument

While talking about method, it is important to know the difference between two terms **parameter** and**argument**.

**Parameter** is variable defined by a method that receives value when the method is called. Parameter are always local to the method they dont have scope outside the method. While **argument** is a value that is passed to a method when it is called.



#### call-by-value and call-by-reference

There are two ways to pass an argument to a method

1. **call-by-value :** In this approach copy of an argument value is pass to a method. Changes made to the argument value inside the method will have no effect on the arguments.
2. **call-by-reference :** In this reference of an argument is pass to a method. Any changes made inside the method will affect the agrument value.

**NOTE :** In Java, when you pass a primitive type to a method it is passed by value whereas when you pass an object of any type to a method it is passed as reference.

#### Example of call-by-value

public class Test{

public void callByValue(int x){

x=100;

}

public static void main(String[] args){

int x=50;

Test t = new Test();

t.callByValue(x); //function call

System.out.println(x);

}

}

**Output :**

50

#### Example of call-by-reference

public class Test{

int x=10;

int y=20;

public void callByReference(Test t){

t.x=100;

t.y=50;

}

public static void main(String[] args){

Test ts = new Test();

System.out.println("Before "+ts.x+" "+ts.y);

ts.callByReference(ts);

System.out.println("After "+ts.x+" "+ts.y);

}

}

**Output :**

Before 10 20

After 100 50

# Java Method Return

Modifiers, return types, and parameter names are not part of the signature.

The signature of a method uniquely identifies the method within a class. It is not allowed to have more than one method in a class with the same signature.

The code for the method is specified in the method's body, which is enclosed in braces.

A method is invoked using its name with the values for its parameters, if any, within parentheses.

To call the add method, use the following statement:

*add(10, 12);*

The above call to the add method passes 10 and 12 as the values for parameters n1 and n2, respectively.

The two values, 10 and 12, that are used to call the add method are called actual parameters.

Java copies the actual parameters to the formal parameters before it executes the code inside the body of the method.

A return statement is used to return a value from a method. It starts with the return keyword.

If a method returns a value, the return keyword must be followed by an expression, which evaluates to the value being returned.

If the method does not return a value, its return type is specified as void. If the method's return type is void, the method does not have to include a return statement.

If a method with a void return type wants to include a return statement, the return keyword must not be followed by any expression; the return keyword is immediately followed by a semicolon to mark the end of the statement.

**Return**

A return statement returns the control to the caller of the method. A return statement is the last statement that is executed in a method's body.

To capture the value of a method call, use the method call expression anywhere you can use a value.

For example, the following code assigns the value returned from the add method to a variable call sum:

*int sum = add(10, 12); // sum variable will be assigned 22*

The following method declaration for a method printMessage;

*void printMessage() {*

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

*}*

The printMessage method specifies void as its return type, which means that it does not return a value to its caller.

It does not specify any parameters, which means it does not accept any input values from its caller.

To call the printMessage method, write the following statement:

*printMessage();*

Since the printMessage() method does not return any value, you cannot use a call to this method as part of any expression where a value is expected.

When a method's return type is void, it is not necessary to use a return statement because we do not have a value to return from the method.

# Java Method Overload

Having more than one method with the same name in the same class is called method overloading.

Methods with the same name in a class could be declared methods, inherited methods, or a combination of both.

Overloaded methods must have different number of parameters, different types of parameters, or both.

The return type, access level and throws clause of a method play no effect in making it an overloaded method.

***import*** *java.io.IOException;*

***class*** *MyClass {*

***public******void*** *m1(****int*** *a) {*

***// Code goes here***

*}*

***public******void*** *m1(****int*** *a,* ***int*** *b) {*

***// Code goes here***

*}*

***public******int*** *m1(String a) {*

***// Code goes here***

***return*** *0;*

*}*

***public******int*** *m1(String a,* ***int*** *b)* ***throws*** *IOException {*

***// Code goes here***

***return*** *0;*

*}*

*}*

**Example**

The following code shows how to use overload.

***public******class*** *Main {*

***public******double*** *add(****int*** *a,* ***int*** *b) {*

*System.out.println(****"Inside add(int a, int b)"****);*

***double*** *s = a + b;*

***return*** *s;  
 }*

***public******double*** *add(****double*** *a,* ***double*** *b) {*

*System.out.println(****"Inside add(double a, double b)"****);*

***double*** *s = a + b;*

***return*** *s;*

*}*

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

*Main ot =* ***new*** *Main();*

***int*** *i = 1;*

***int*** *j = 1;*

***double*** *d1 = 10.42;*

***float*** *f1 = 22.3F;*

***float*** *f2 = 24.5F;*

***short*** *s1 = 22;*

***short*** *s2 = 26;*

*ot.add(i, j);*

*ot.add(d1, j);*

*ot.add(i, s1);*

*ot.add(s1, s2);*

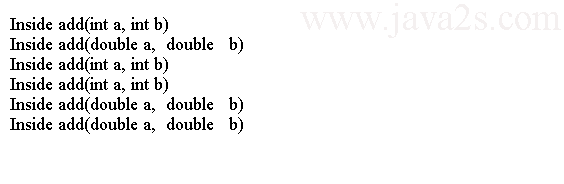
*ot.add(f1, f2);*

*ot.add(f1, s2);*

*}*

*}*

The code above generates the following result.



**Example**

Sometimes, overloaded methods and automatic type widening may confuse the compiler resulting in a compiler error.

***class*** *Adder {* ***public******double*** *add(****int*** *a,* ***double*** *b) {*

***return*** *a + b;*

*}*

***public******double*** *add(****double*** *a,* ***int*** *b) {*

***return*** *a + b;*

*}*

*}*

***public******class*** *Main {*

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

*Adder a =* ***new*** *Adder();*

***// double d = a.add(2, 3); // A compile-time error***

***double*** *d1 = a.add((****double****) 2, 3);* ***// OK. Will use add(double, int)***

***double*** *d2 = a.add(2, (****double****) 3);* ***// OK. Will use add(int, double)***

*}*

*}*

# Java Instance/Static Methods

A class can have two types of methods: instance methods and class methods. Instance methods and class methods are also called non-static methods and static methods, respectively.

An instance method is used to implement behavior for the instances of the class. An instance method can only be invoked in the context of an instance of the class.

A class method is used to implement the behavior for the class itself. A class method always executes in the context of a class.

The static modifier is used to define a class method. The absence of the static modifier in a method declaration makes the method an instance method.

**Example**

The following are examples of declaring some static and non-static methods:

*// A* ***static******or******class*** *method*

***static*** *void aClassMethod() {*

*}*

*// A non-****static******or*** *instance method*

*void anInstanceMethod() {*

*}*

**Note**

When a static method of a class is called, an instance of that class may not exist. Therefore, it is not allowed to refer to instance variables from inside a static method.

Class variables exist as soon as the class definition is loaded into memory. The class definition is loaded into memory before the first instance of a class is created.

A class method or static method can refer to only class variables or static variables of the class. An instance method or non-static method can refer to class variables as well as instance variables of the class.

The following code demonstrate the types of class fields that are accessible inside a method.

***public******class*** *Main {*

***static******int*** *m = 100;* ***// A static variable***

***int*** *n = 200;* ***// An instance variable***

***// Declare a static method***

***static******void*** *printM() {*

***/\****

***\* We can refer to only static variable m in this method because you are***

***\* inside a static method***

***\*/***

*System.out.println(****"printM() - m = "*** *+ m);*

*}*

***// Declare an instance method***

***void*** *printMN() {*

***/\* We can refer to both static and instance variables m and n in this method \*/***

*System.out.println(****"printMN() - m = "*** *+ m);*

*System.out.println(****"printMN() - n = "*** *+ n);*

*}*

*}*

**Invoking a Method**

Executing the code in the body of a method is called invoking (or calling) a method.

Instance methods and class methods are invoked differently.

An instance method is invoked on an instance of the class using dot notation.

*<instance reference>.<instance method name>(<actual parameters>)*

We must have a reference to an instance of a class before calling an instance method of that class.

The following code shows how to invoke the printMN() instance method of the Main class:

***// Create an instance of Main class and***

***// store its reference in mt reference variable***

*Main mt =* ***new*** *Main();*

***// Invoke the printMN() instance method using the mt reference variable***

*mt.printMN();*

To invoke a class method, use dot notation with the class name.

The following code invokes the printM() class method of the Main class:

***// Invoke the printM() class method***

*Main.printM();*

Whatever belongs to a class also belongs to all instances of that class. We can also invoke a class method using a reference of an instance of that class.

*Main mt =* ***new*** *Main();*

*mt.printM();* ***// Call the class method using an instance mt***

Using the class name to invoke a class method is more intuitive than using an instance reference.

# Java main Method

Let's discuss the main() method that we have been using to run our classes.

The main() method declaration is as follows:

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

*}*

Two modifiers, public and static, are used in the declaration of the main() method.

The public modifier makes it accessible from anywhere in the application as long as the class in which it is declared is accessible.

The static modifier makes it a class method, so it can be invoked using a class name.

Its return type is void, which means it does not return a value to its caller.

Its name is main and it accepts one parameter of type String array (String[]).

main() method is the entry method for Java application. For example, you would use the following command to run the Main class:

*java com.java2s.Main*

The main() method is invoked by the JVM when you run a class.

# Java Parameter Passing

Java supports two kinds of data types: primitive data type and reference data type.

A primitive data type is a simple data structure and it has only one value associated with it. A reference data type is a complex data structure and it represents an object.

A variable of a primitive data type stores the value directly at its memory address.

Things are different when you work with objects and reference variables.

All parameters in Java are passed by value.

When a parameter is a primitive data type, the value of the actual parameter is copied to the parameter.

Any changes made to the parameter's value inside the method's body will change only the copy of the formal parameter and not the value of the actual parameter.

When a parameter is passed by reference value, the reference stored in the actual parameter is copied to the formal parameter. Both the actual parameter and the formal parameter refer to the same object in memory.

You can assign a reference to another object to the formal parameter inside the method's body.

**Example**

The following code demonstrates the pass by reference mechanism in Java.

***class*** *Phone {* ***public*** *String model =* ***"Unknown"****;*

***public******int*** *year = 2014;*

***public******double*** *price = 0.0;*

*}*

***public******class*** *Main {*

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

*Phone myPhone =* ***new*** *Phone();*

*myPhone.model =* ***"iPhone"****;*

*myPhone.year = 2009;*

*myPhone.price = 16000.0;*

*System.out.println(****"#1: model = "*** *+ myPhone.model +* ***", year = "***

*+ myPhone.year +* ***", price = "*** *+ myPhone.price);*

*Main.test(myPhone);*

*System.out.println(****"#4: model = "*** *+ myPhone.model +* ***", year = "***

*+ myPhone.year +* ***", price = "*** *+ myPhone.price);*

*}*

***public******static******void*** *test(Phone xPhone) {*

*System.out.println(****"#2: model = "*** *+ xPhone.model +* ***", year = "***

*+ xPhone.year +* ***", price = "*** *+ xPhone.price);*

***// Let's make xyCar refer to a new object***

*xPhone =* ***new*** *Phone();*

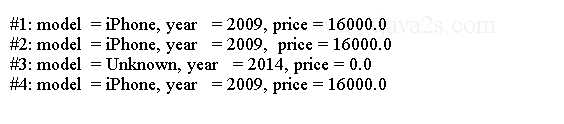
*System.out.println(****"#3: model = "*** *+ xPhone.model +* ***", year = "***

*+ xPhone.year +* ***", price = "*** *+ xPhone.price);*

*}*

*}*

The code above generates the following result.



**Note**

When a reference type parameter is passed to a method, the formal parameter can access the object the same way the actual parameter can access the object.

The formal parameter can modify the object by directly changing the values of the instance variables or by calling methods on the object.

Any modification made on the object through the formal parameter is immediately visible through the actual parameter because both hold the reference to the same object in memory.

The formal parameter itself can be modified to reference another object inside the method.

To disable the method to change the reference type formal parameter to reference a different object, use the keyword final in the reference type formal parameter declaration.

***public******class*** *Main {*

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

*Phone myPhone =* ***new*** *Phone();*

*myPhone.model =* ***"iPhone"****;*

*myPhone.year = 2009;*

*myPhone.price = 16000.0;*

*Main.test(myPhone);*

*}*

***public******static******void*** *test(****final*** *Phone xPhone) {*

*System.out.println(****"#2: model = "*** *+ xPhone.model +* ***", year = "***

*+ xPhone.year +* ***", price = "*** *+ xPhone.price);*

***// Let's make xyCar refer to a new object***

***//xPhone = new Phone();***

*}*

*}*

***class*** *Phone {*

***public*** *String model =* ***"Unknown"****;*

***public******int*** *year = 2014;*

***public******double*** *price = 0.0;*

*}*

**The code above generates the following result.**

http://www.java2s.com/Tutorials/JavaImage/myResult/N/NOTE__CACADAC67188C2EA2FF3.PNG

# Java varargs Method

The term "varargs" is shorthand for "variable-length arguments."

The varargs declares a method or constructor that accepts a variable number of arguments (or parameters).

**Declare varargs Method**

To declare varargs, add an ellipsis ... after the data type of the method's argument.

The following code shows a max() method declaration with one variable-length argument, num, which is of the int data type.

***public******static*** *int max(int... num) {*

*}*

Adding whitespaces before and after ellipsis is optional.

A varargs method can have more than one argument. The following code shows that aMethod() accepts three arguments, one of which is a variable-length argument:

***public******static*** *int aMethod(String str, double d1, int...num) {*

*}*

A varargs method can have a maximum of one variable-length argument.

The variable-length argument of a varargs method must be the last argument in the argument list.

*void m2(String str, int...n1) {*

*}*

**Example**

Let's rewrite the max() method to make it a varargs method:

***public******class*** *Main {*

***public******static******int*** *max(int... num) {*

***int*** *max = Integer.MIN\_VALUE;*

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

***if*** *(num[i] > max) {*

*max = num[i];//from ww w.j a v a 2 s . com*

*}*

*}*

***return*** *max;*

*}*

*}*

**Use varargs Method**

We can use a for loop to process the list of arguments for the variable-length argument.

The length property gives you the number of values that were passed for the variable-length argument.

To get the nth value in the variable-length argument, you need to use varArgsName[n-1].

We can use a foreach loop to work with variable-length argument.

***public******class*** *Main {*

***public******static*** *int max2(int... num) {*

*int max = Integer.MIN\_VALUE;*

***for*** *(int currentNumber : num) {*

***if*** *(currentNumber > max) {*

*max = currentNumber;*

*}*

*}*

*return max;*

*}*

*}*

We can call the Main.max() method as follows:

*int max1 = Main.max(1, 8);*

*int max2 = Main.max(1, 1, 3);*

You can use zero or more arguments for a variable-length argument in a method. The following code is a valid call to the max() method:

*int max = Main.max(); // Passing no argument is ok*

The following declaration of the max() method will force its caller to pass at least two integers:

***// Argumenets n1 and n2 are mandatory***

***public******static*** *int max(int n1, int n2, int... num) {*

*}*

The compiler will treat the first two arguments, n1 and n2, as mandatory and the third argument, num, as optional.

***public******class*** *Main {*

***public******static******int*** *max(****int*** *n1,* ***int*** *n2, int... num) {*

***// Initialize max to the maximum of n1 and n2***

***int*** *max = (n1 > n2 ? n1 : n2);*

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

***if*** *(num[i] > max) {*

*max = num[i];*

*}*

*}*

***return*** *max;*

*}*

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

*System.out.println(max(7, 9));*

*System.out.println(max(7, 9, 10));*

*System.out.println(max(7, 9, 10, 13));*

*}*

*}*

The code above generates the following result.



**Overloading a Varargs Method**

The same overloading rules for methods apply to a varargs method.

We can overload a method with a variable-length argument as long as the parameters for the methods differ in type, order, or number.

For example, the following is a valid example of an overloaded max() method:

***public******class*** *Main {*

***public******static*** *int max(int x, int y) {*

*}*

***public******static*** *int max(int...num) {*

*}*

*}*

Consider the following code

*int max = Main.max(12, 13); // which max() will be called?*

Java will call the max(int x, int y). Java first attempts to find a method declaration using an exact match for the number of parameters. If it does not find an exact match, it looks for a match using variable-length parameters.

If a varargs method is overloaded, Java uses the more specific version of the method instead of using a varargs method. java uses varargs method as the last resort to resolve a method call.

The overloading of the method itself may be valid. However, the call to it may cause an issue.

***public******class*** *Main {*

***public******static******int*** *max(int... num) {*

***return*** *0;*

*}*

***public******static******int*** *max(double... num) {*

***return*** *0;*

*}*

*}*

**Which max() would the following code call?**

*int max = Main.max(); // Which max() to call?*

The above statement will generate a compilation time error.

**Varargs Methods and the main() Method**

The signature for the main() method must be main(String[] args).

The following declaration of main() method for the Main class is valid.

***public******class*** *Main {*

***public******static*** *void main(String... args) {*

*System.out.println(****"Hello from varargs main()..."****);*

*}*

*}*

# Java Initialization Block

**Instance Initialization Block**

An instance initialization block is used to initialize objects of a class.

An instance initializer is simply a block of code inside the body of a class, but outside any methods or constructors.

An instance initializer does not have a name. Its code is simply placed inside an opening brace and a closing brace.

**Example**

The following code shows how to declare an instance initializer for the Test class.

Note that an instance initializer is executed in instance context and the keyword this is available inside the instance initializer.

**class** Test {  
 **private** **int** num;

**// An instance initializer**

{

this.num = 101;

**/\* Other code for the instance initializer\*/**

}

**/\* Other code for Test class\*/**

}

**Multiple instance initializers**

We can have multiple instance initializers for a class. All of them are executed automatically in textual order for every object we create.

Code for all instance initializers are executed before any constructor.

The following code demonstrates the sequence in which the constructor and instance initializers are executed.

***public******class*** *Main {*

*{  
 System.out.println(****"Inside instance initializer 1."****);*

*}*

*{*

*System.out.println(****"Inside instance initializer 2."****);*

*}*

***public*** *Main() {*

*System.out.println(****"Inside no-args constructor."****);*

*}*

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

*Main m =* ***new*** *Main();*

*}*

*}*

The code above generates the following result.



An instance initializer cannot have a return statement.

**static Initialization Block**

A static initialization block is also known as a static initializer. It is similar to an instance initialization block.

It is used to initialize a class. An instance initializer is executed once per object whereas a static initializer is executed only once for a class when the class definition is loaded into JVM.

We need to use the static keyword in the beginning of its declaration.

We can have multiple static initializers in a class. All static initializers are executed in textual order in which they appear, and execute before any instance initializers.

The following code demonstrates when a static initializer is executed.

***public******class*** *Main {*

***private******static******int*** *num;*

*{****// An instance initializer***

*System.out.println(****"Inside instance initializer."****);*

*}* ***// A static initializer. Note the use of the keyword static below.***

***static*** *{*

*num = 2014;*

*System.out.println(****"Inside static initializer."****);*

*}*

***// Constructor***

***public*** *Main() {*

*System.out.println(****"Inside constructor."****);*

*}*

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

*System.out.println(****"Inside main() #1. num: "*** *+ num);*

***// Declare a reference variable of the class***

*Main si;*

*System.out.println(****"Inside main() #2. num: "*** *+ num);*

***new*** *Main();* ***// Create an object***

*System.out.println(****"Inside main() #3. num: "*** *+ num);*

***new*** *Main();****// Create another object***

*}*

*}*

The code above generates the following result.



static initializer cannot throw checked exceptions and it cannot have a return statement.

**Recursive Methods:**

A method which calls itself is called *recursive* method. Recursive methods are very useful when we want to process data stored in list structure or in tree structure. Some types of functionality can not be implemented without recursive method. e.g.,printing the family tree.

Every recursive method calls itself in the body of its method. Also every recursive method will have a condition where it does not call itself. With out this condition, the method will go into infinite loop causing StackOverflowError.

*class FactorialUsingRecursion{  
    public static void main(String s[]){  
        System.out.println("Factorial of 5 is " + factorial(5) );  
    }      
    public static int factorial(int i){  
        if( i == 1 )    // LINE A  
        {  
            return 1;  
        }      
        return i \* factorial(i - 1); // LINE B  
    }  
}*

**OUTPUT**

Factorial of 5 is 120

**DESCRIPTION**

The method factorial defined here is a recursive method. As we can see at LINE B, it is calling itself. It also has a condition in LINE A. When this condition is true, it will not call itself, hence stopping the recursive calls.   
As we know the factorial of any number is the product of all the numbers before it. We know factorial of *n* is

***factorial(n) = 1 \* 2 \* 3 \* .... \* n-1 \* n***

This can be rewritten as

***factorial(n) = n \* factorial(n-1);  
factorial(n-1) = (n - 1) \* factorial(n-2);  
...  
...  
factorial(3) = 3 \* factorial(2);  
factorial(2) = 2 \* factorial(1);  
factorial(1) = 1;***

Using this principle, for calculating the factorial of 5 : *factorial(5) = 5 \* factorial(4)*   
and for 4 : *factorial(4) = 4 \* factorial(3)*   
and for 3 : *factorial(3) = 3 \* factorial(2)*   
and for 2 : *factorial(2) = 2 \* factorial(1)*.   
Since we have the given the *factorial(1)* as 1 in LINE A, it will not call the method again and simply returns 1.   
  
Since the recursion stops when the i value is 1, the factorials will be calculated, as shown below.

***factorial(1) = 1;  
factorial(2) = 2 \* factorial(1) = 2 \* 1 = 2;  
factorial(3) = 3 \* factorial(2) = 3 \* 2 = 6;  
factorial(4) = 4 \* factorial(3) = 4 \* 6 = 24;  
factorial(5) = 5 \* factorial(4) = 5 \* 24 = 120;***

simple interest using methods

This program explains how to calculate simple interest using methods. The formula for calculating simple interest is *I = P \* T \* R / 100.0* where *I* is interest, *P* is principal, *T* is time in years and *R* is interest rate per annum.

*class CalculateSimpleInterest{  
    public static void main(String s[]){  
        double principal = 5000; // Rs. 5000  
        double time = 2; // 2 years  
        double rate = 11.25; // 11.25 % per annum*

*double interest = calculateInterest( principal, time, rate ); //* ***LINE A****printSummary( principal, time, rate, interest );  
    }      
    public static double calculateInterest(double principal, double time, double rate){ //* ***LINE B*** *double result = principal \* time \* rate / 100.0; // LINE C  
        return result;  
    }      
    public static void printSummary(double p, double t, double r, double interest){  
        System.out.print("Interest for Rs " + p + " for " + t + " years ");  
        System.out.print("at the rate of " + r + "% p.a. is Rs " + interest);  
    }  
}*

**OUTPUT**

Interest for Rs 5000.0 for 2.0 years at the rate of 11.25% p.a. is Rs 1125.0

**DESCRIPTION**

This program shows how to calculate simple interest using methods. At LINE A, we are calling the calculateInterest method by passing the parameters principal, time and rate. The return value from the method is assigned to variable interest. In the calculateInterest method, we are calculating the interest using the passed parameters. The calculated interest is returned back using the return keyword. The return value is assigned to the variable interest in the main method. 

Here the calculateInterest method is the *called method*, where as the main method is the *calling method*. 

We also have another method printSummary which takes the parameters p, t, r and interest and prints the summary of the interest calculation. As shown in this method, it is not necessary to have the same parameter name in the calling method and the called method. In the calling method which is main, the variable is principal, where as in the called method printSummary it is p. Similarly time is t and rate is r.