CS20004: Object Oriented Programming using Java

Lec-9

In this Discussion . . .

- Classes & Objects
  - Garbage collection
  - this keyword
  - Method overloading
  - Constructor overloading
  - Passing objects to constructors
  - Argument passing
    - Command line arguments
  - Recursion
- References

finalize() method

- finalize method is invoked just before the object is destroyed
- finalize() method in Java is used to release all the resources used by the object before it is deleted/destroyed by the Garbage collector.
- finalize is not a reserved keyword, it's a method.
- Once the clean-up activity is done by the finalize() method, garbage collector immediately destroys the Java object.

finalize() method

- Java Virtual Machine(JVM) permits invoking of finalize() method only once per object.
- Implemented when the usual way of removing objects from memory is insufficient, and some special actions has to be carried out
- Implemented inside a class as:

protected void finalize() {.....}

- finalize() is a method of the Object class in Java. The finalize() method is a non-static and protected method of java.lang.Object class.

finalize() method

- In Java, the Object class is superclass of all Java classes. Being an object class method finalize() method is available for every class in Java.
- Hence, Garbage Collector can call finalize() method on any Java object for clean-up activity.
- Once object is finalized JVM sets a flag in the object header to say that it has been finalized, and won't finalize it again. If user tries to use finalize() method for the same object twice, JVM ignores it.

Garbage collector

- Java considers unreferenced objects that are not being used by any program execution or objects that are no longer needed, as garbage.
- Garbage collection is the process of destroying unused objects and reclaiming the unused runtime memory automatically. By doing this memory is managed efficiently by Java.
- Garbage collection is carried out by the garbage collector.

Garbage collector

- The garbage collector is a part of Java Virtual Machine(JVM).
- Garbage collector checks the heap memory, where all the objects are stored by JVM, looking for unreferenced objects that are no more needed. And automatically destroys those objects.
- Garbage collector calls finalize() method for clean up activity before destroying the object.
- Java does garbage collection automatically; there is no need to do it explicitly, unlike other programming languages.

Garbage collector

- The garbage collector in Java can be called explicitly using the following method:

```
System.gc();
```

  - System.gc() is a method in Java that invokes garbage collector which will destroy the unreferenced objects.
  - System.gc() calls finalize() method only once for each object.

How does finalize() method work with Garbage Collection?

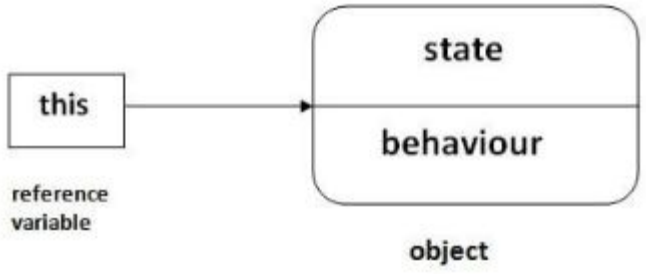
- The JVM calls the garbage collector to delete unreferenced objects. After determining the objects that have no links or references, it calls the finalize() method which will perform the clean activity and the garbage collector destroys the object.

```
public class Demo
{
    public static void main(String[] args)
    {
      String a = "Hello World!";
      a = null; //unreferencing string object
    }
}
```

- For the program, when the String object a holds value Hello World! it has a reference to an object of the String class.
- But, when it holds a null value it does not have any reference.
- Hence, it is eligible for garbage collection. The garbage collector calls finalize() method to perform clean-up before destroying the object.

this keyword

- In Java, this is a reference variable that refers to the current object.



- Usages of this keyword:
  - this can be used to refer current class instance variable.
  - this can be used to invoke current class method (implicitly)
  - this() can be used to invoke current class constructor.
  - this can be passed as an argument in the method call.
  - this can be passed as argument in the constructor call.
  - this can be used to return the current class instance from the method.

1) this: to refer current class instance variable

- The this keyword can be used to refer current class instance variable.
- If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.
- In order to find out the importance of this keyword, let us first understand what are the problems which can occur, if we do not use it.

1) this: to refer current class instance variable

-

```
class Student
{
	int rollno;
	String name;
	float fee;
	Student(int rollno,String name,float fee)
	{
		rollno=rollno;
		name=name;
		fee=fee;
	}
	void display()
	{
		System.out.println(rollno+" "+name+" "+fee);
	}
}
class This1
{
	public static void main(String args[ ])
	{
		Student s1=new Student(111,"ankit",5000f);
		Student s2=new Student(112,"sumit",6000f);
		s1.display();
		s2.display();
	}
}
```

1) this: to refer current class instance variable

-

```
class Student
{
        int rollno;
        String name;
        float fee;
        Student(int rollno,String name,float fee)
        {
                rollno=rollno;
                name=name;
                fee=fee;
        }
        void display()
        {
                System.out.println(rollno+" "+name+" "+fee);
        }
}
class This1
{
        public static void main(String args[ ])
        {
                Student s1=new Student(111,"ankit",5000f);
                Student s2=new Student(112,"sumit",6000f);
                s1.display();
                s2.display();
        }
}
```

```
0 null 0.0
0 null 0.0
```

1) this: to refer current class instance variable

-

```
class Student
{
        int rollno;
        String name;
        float fee;
        Student(int rollno,String name,float fee)
        {
                this.rollno=rollno;
                this.name=name;
                this.fee=fee;
        }
        void display()
        {
                System.out.println(rollno+" "+name+" "+fee);
        }
}
class This2
{
        public static void main(String args[ ])
        {
                Student s1=new Student(111,"ankit",5000f);
                Student s2=new Student(112,"sumit",6000f);
                s1.display();
                s2.display();
        }
}
```

The example on left side, has local variable parameters (i.e., formal arguments) and instance variables same.

Thus, we need to use this keyword to distinguish local variable and instance variable.

```
111 ankit 5000.0
112 sumit 6000.0
```

1) this: to refer current class instance variable

- However, if the local variables(formal arguments) and instance variables are different, there is no need to use this keyword.

```
class Student
{
	int rollno;
	String name;
	float fee;
	Student(int roll,String n,float f)
	{
		rollno=roll;
		name=n;
		fee=f;
	}
	void display()
	{
		System.out.println(rollno+" "+name+" "+fee);
	}
}
class This3
{
	public static void main(String args[])
	{
		Student s1=new Student(111,"ankit",5000f);
		Student s2=new Student(112,"sumit",6000f);
		s1.display();
		s2.display();
	}
}
```

```
111 ankit 5000.0
112 sumit 6000.0
```

2) this: to invoke current class method

- We can invoke the method of the current class by using the this keyword.
- If we don't use the this keyword, compiler automatically adds this keyword while invoking the method.

```
class Delhi
{
	void m()
	{
		System.out.println("Chole Bhature");
	}
	void n()
	{
		System.out.println("Tikki");
		//m();//same as this.m()
		this.m();
	}
}
class Meerut
{
	public static void main(String args[])
	{
		Delhi de=new Delhi();
		de.n();
	}
}
```

2) this: to invoke current class method

- We can invoke the method of the current class by using the this keyword.
- If we don't use the this keyword, compiler automatically adds this keyword while invoking the method.

```
class Delhi
{
	void m()
	{
		System.out.println("Chole Bhature");
	}
	void n()
	{
		System.out.println("Tikki");
		//m();//same as this.m()
		this.m();
	}
}
class Meerut
{
	public static void main(String args[])
	{
		Delhi del=new Delhi();
		del.n();
	}
}
```

Tikki
Chole Bhature

3) this() : to invoke current class constructor

- The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

```
class A
{
	A()
	{
		System.out.println("hello a");
	}
	A(int x)
	{
		this();
		System.out.println(x);
	}
}
class This4
{
	public static void main(String args[])
	{
		A a=new A(10);
	}
}
```

3) this() : to invoke current class constructor

- The program provided below helps in calling the default constructor from parameterized constructor:

```
class A
{
        A()
        {
                System.out.println("hello a");
        }
        A(int x)
        {
                this();
                System.out.println(x);
        }
}
class This4
{
        public static void main(String args[])
        {
                A a=new A(10);
        }
}
```

```
hello a
10
```

3) this() : to invoke current class constructor

- The program provided below helps in calling the parameterized constructor from default constructor:

```
class A
{
        A()
        {
                this(5);
                System.out.println("hello a");
        }
        A(int x)
        {
                System.out.println(x);
        }
}
class This5
{
        public static void main(String args[])
        {
                A a=new A();
        }
}
```

```
5
hello a
```

3) this() : to invoke current class constructor

- The this() constructor call should be used to reuse the constructor from the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining.

```
class Student{
int rollno;
String name,course;
float fee;
Student(int rollno,String name,String course){
this.rollno=rollno;
this.name=name;
this.course=course;
}
Student(int rollno,String name,String course,float fee){
this.fee=fee;
this(rollno,name,course);
}
void display(){System.out.println(rollno+" "+name+" "+course+"
"+fee);}
}
class This6{
public static void main(String args[]){
Student s1=new Student(111,"ankit","java");
Student s2=new Student(112,"sumit","java",6000f);
s1.display();
s2.display();
}}
```

3) this() : to invoke current class constructor

- The this() constructor call should be used to reuse the constructor from the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining.

```
class Student{
int rollno;
String name,course;
float fee;
Student(int rollno,String name,String course){
this.rollno=rollno;
this.name=name;
this.course=course;
}
Student(int rollno,String name,String course,float fee){
this.fee=fee;
this(rollno,name,course);
}
void display(){System.out.println(rollno+" "+name+" "+course+"
"+fee);}
}
class This6{
public static void main(String args[]){
Student s1=new Student(111,"ankit","java");
Student s2=new Student(112,"sumit","java",6000f);
s1.display();
s2.display();
}}
```

```
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Objects and Classes$ javac This6.java
This6.java:12: error: call to this must be first statement in constructor
this(rollno,name,course);
    ^
1 error
```

3) this() : to invoke current class constructor

- The this() constructor call should be used to reuse the constructor from the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining. Call to this() must be the first statement in constructor.

```
class Student{
int rollno;
String name,course;
float fee;
Student(int rollno,String name,String course){
this.rollno=rollno;
this.name=name;
this.course=course;
}
Student(int rollno,String name,String course,float fee){
this(rollno,name,course);
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+course+"
"+fee);}
}
class This6{
public static void main(String args[]){
Student s1=new Student(111,"ankit","java");
Student s2=new Student(112,"sumit","java",6000f);
s1.display();
s2.display();
}}
```

```
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Objects and Classes$ javac This6.java
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Objects and Classes$ java This6
111 ankit java 0.0
112 sumit java 6000.0
```

4) this: to pass as an argument in the method

- The this keyword can also be passed as an argument in the method. It is mainly used in the event handling.

```
class S2
{
        void m(S2 obj)
        {
                System.out.println("method is invoked");
        }
        void p()
        {
                m(this);
        }
        public static void main(String args[])
        {
                S2 s1 = new S2();
                s1.p();
        }
}
```

```
method is invoked
```

Application of this that can be passed as an argument: In event handling (or) in a situation where we have to provide reference of a class to another one. It is used to reuse one object in many methods.

5) this: to pass as argument in the constructor call

- We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes.

```
class B{
  A4 obj;
  B(A4 obj){
    this.obj=obj;
  }
  void display(){
    System.out.println(obj.data);//using data member of A4 class
  }
}

class A4{
  int data=10;
  A4(){
   B b=new B(this);
   b.display();
  }
  public static void main(String args[]){
   A4 a=new A4();
  }
}
```

5) this: to pass as argument in the constructor call

- We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes.

```
class B{
  A4 obj;
  B(A4 obj){
    this.obj=obj;
  }
  void display(){
    System.out.println(obj.data);//using data member of A4 class
  }
}

class A4{
  int data=10;
  A4(){
   B b=new B(this);
   b.display();
  }
  public static void main(String args[]){
   A4 a=new A4();
  }
}
```

10

6) this: keyword can be used to return current class instance

- We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive).

Syntax:

```
return_type method_name(){
return this;
}
```

6) this: keyword can be used to return current class instance

- We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive).

```
class A
{
        A getA()
        {
                return this;
        }
        void msg()
        {
                System.out.println("Hello java");
        }
}
class Test1
{
        public static void main(String args[])
        {
                new A().getA().msg();
        }
}
```

Hello java

Method Overloading

- If a class has multiple methods such that they have the same name but different parameters, it is known as Method Overloading.
- Different ways to overload the method:
  - By changing the number of arguments
  - By changing the data type
- In Java, Method Overloading is not possible by changing the return type of the method only.
- When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters.
- When no exact match can be found, Java's automatic type conversion can aid overload resolution

Method Overloading: Changing number of arguments

```
class Adder
{
	int add(int a,int b)
	{
		return a+b;
	}
	int add(int a,int b,int c)
	{
		return a+b+c;
	}
}
class Testoverloading
{
	public static void main(String[] args)
	{
		Adder a1 = new Adder();

		System.out.println(a1.add(11,11));
		System.out.println(a1.add(11,11,11));
	}
}
```

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

Method Overloading: Changing number of arguments

```
class Adder
{
        int add(int a,int b)
        {
                return a+b;
        }
        int add(int a,int b,int c)
        {
                return a+b+c;
        }
}
class Testoverloading
{
        public static void main(String[] args)
        {
                Adder a1 = new Adder();

                System.out.println(a1.add(11,11));
                System.out.println(a1.add(11,11,11));
        }
}
```

```
22
33
```

Method Overloading: Changing the datatype of arguments

```
class Adder
{
	int add(int a, int b)
	{
		return a+b;
	}
	double add(double a, double b)
	{
		return a+b;
	}
}
class Testoverloading1
{
	public static void main(String[] args)
	{
		Adder a1 = new Adder();
		System.out.println(a1.add(11,11));
		System.out.println(a1.add(12.3,12.6));
	}
}
```

In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

Method Overloading: Changing the datatype of arguments

```
class Adder
{
        int add(int a, int b)
        {
                return a+b;
        }
        double add(double a, double b)
        {
                return a+b;
        }
}
class Testoverloading1
{
        public static void main(String[] args)
        {
                Adder a1 = new Adder();
                System.out.println(a1.add(11,11));
                System.out.println(a1.add(12.3,12.6));
        }
}
```

```
22
24.9
```

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

```
class Adder
{
        int add(int a,int b)
        {
                return a+b;
        }
        double add(int a,int b)
        {
                return a+b;
        }
}
class Testoverloading2
{
        public static void main(String[] args)
        {
                Adder a1 = new Adder();
                System.out.println(a1.add(11,11));//ambiguity
        }
}
```

```
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Objects and Classes$ j
avac Testoverloading2.java
Testoverloading2.java:7: error: method add(int,int) is already defined in class
Adder
        double add(int a,int b)
               ^
1 error
```

Can we overload java main() method?

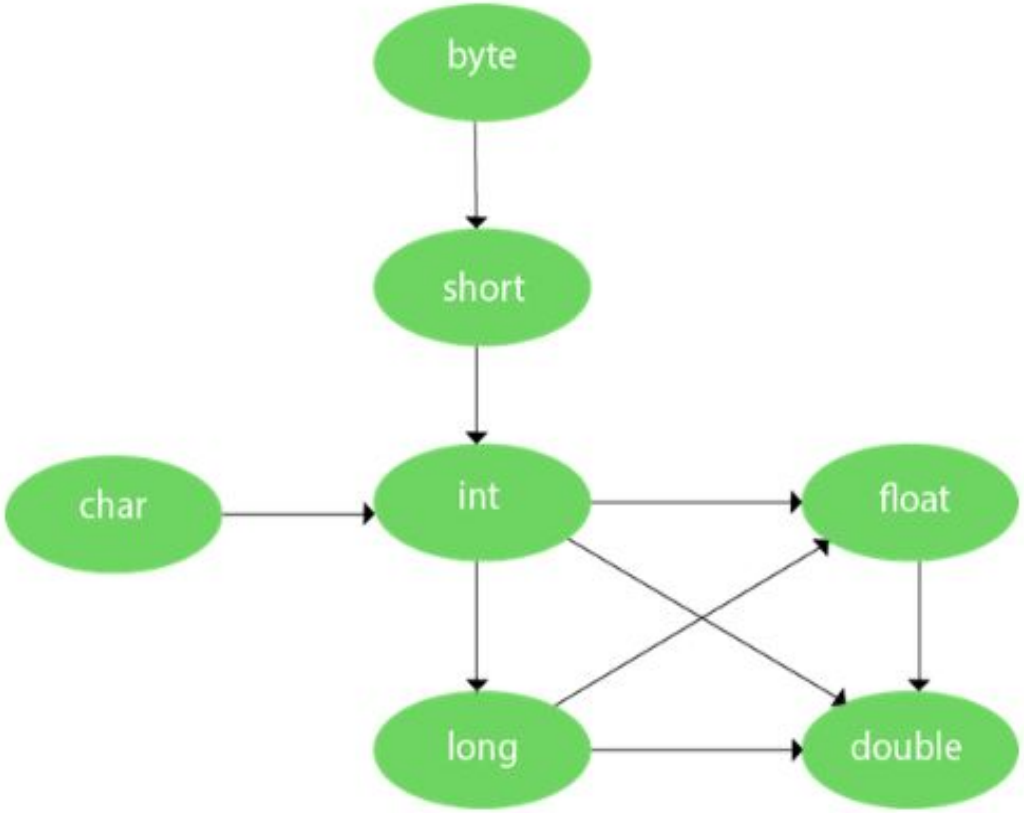
- Yes, by method overloading. We can have any number of main methods in a class by method overloading. But JVM calls main() method which receives String array as arguments only.

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

main with String[]

Method Overloading and Type Promotion

- One type is promoted to another implicitly if no matching datatype is found.



- As displayed in the diagram, byte can be promoted to short, int, long, float or double.
- The short datatype can be promoted to int, long, float or double.
- The char datatype can be promoted to int,long,float or double and so on.

Method Overloading and Type Promotion

- One type is promoted to another implicitly if no matching datatype is found.

```
class Calculation
{
        void sum(int a,long b)
        {
                System.out.println(a+b);
        }
        void sum(int a,int b,int c)
        {
                System.out.println(a+b+c);
        }

        public static void main(String args[])
        {
                Calculation obj=new Calculation();
  obj.sum(20,20);//now second int literal will be promoted to long
                obj.sum(20,20,20);

        }
}
```

Method Overloading and Type Promotion

- One type is promoted to another implicitly if no matching datatype is found.

```
class Calculation
{
        void sum(int a,long b)
        {
                System.out.println(a+b);
        }
        void sum(int a,int b,int c)
        {
                System.out.println(a+b+c);
        }

        public static void main(String args[])
        {
                Calculation obj=new Calculation();
  obj.sum(20,20);//now second int literal will be promoted to long
                obj.sum(20,20,20);

        }
}
```

40
60

Constructor Overloading

- In Java, we can overload constructors like methods.
- The constructor overloading can be defined as the concept of having more than one constructor with different parameters, so that every constructor can perform a different task.

Constructor Overloading

```
public class Stud
{
	//instance variables of the class
	int id;
	String name;
	Stud()
	{
		System.out.println("Default constructor");
	}
	Stud(int i, String n)
	{
		id = i;
		name = n;
	}
	public static void main(String[] args)
	{
		//object creation
		Stud s = new Stud();
		System.out.println("\nDefault Constructor values: \n");
		System.out.println("Student Id : "+s.id + "\nStudent Name : "+s.name);

		System.out.println("\nParameterized Constructor values: \n");
		Stud student = new Stud(10, "David");
		System.out.println("Student Id : "+student.id + "\nStudent Name :
"+student.name);
	}
}
```

Constructor Overloading

```
public class Stud
{
	//instance variables of the class
	int id;
	String name;
	Stud()
	{
		System.out.println("Default constructor");
	}
	Stud(int i, String n)
	{
		id = i;
		name = n;
	}
	public static void main(String[] args)
	{
		//object creation
		Stud s = new Stud();
		System.out.println("\nDefault Constructor values: \n");
		System.out.println("Student Id : "+s.id + "\nStudent Name : "+s.name);

		System.out.println("\nParameterized Constructor values: \n");
		Stud student = new Stud(10, "David");
		System.out.println("Student Id : "+student.id + "\nStudent Name :
"+student.name);
	}
}
```

```
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Objects and Classes$ j
ava Stud
Default constructor

Default Constructor values:

Student Id : 0
Student Name : null

Parameterized Constructor values:

Student Id : 10
Student Name : David
```

Constructor Overloading

- In the example on the previous slide, the Student class constructor is overloaded with two different constructors, I.e., default and parameterized.
- Sometimes, we use multiple constructors to initialize the different values of the class.
- We must also notice that the java compiler invokes a default constructor when we do not use any constructor in the class.
- However, the default constructor is not invoked if we have used any constructor in the class, whether it is default or parameterized. In this case, the java compiler throws an exception saying the constructor is undefined.

Constructor Overloading

```
public class Colleges
{
        String collegeId;
        Colleges(String collegeId)
        {
                this.collegeId = "IIT " + collegeId;
        }
        public static void main(String[] args)
        {
                // TODO Auto-generated method stub
                Colleges clg = new Colleges(); //this can't create
colleges constructor now.
        }
}
```

```
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Objects and Classes$ javac Colleges.java
Colleges.java:11: error: constructor Colleges in class Colleges cannot be applied to given types;
                Colleges clg = new Colleges(); //this can't create colleges constructor now.
                               ^
  required: String
  found: no arguments
  reason: actual and formal argument lists differ in length
1 error
```

Passing Objects to Constructor

```
import java.util.Scanner;
public class Employee {
   String name;
   int age;
   public Employee(){}
   public Employee(String name, int age){
      this.name = name;
      this.age = age;}
   public Employee copyObject(Employee std){
      this.name = std.name;
      this.age = std.age;
      return std; }
   public void displayData(){
      System.out.println("Name : "+this.name);
      System.out.println("Age : "+this.age);
   }
   public static void main(String[] args) {
      Scanner sc =new Scanner(System.in);
      System.out.println("Enter your name ");
      String name = sc.next();
      System.out.println("Enter your age ");
      int age = sc.nextInt();
      Employee std = new Employee(name, age);
      System.out.println("Contents of the original object");
      std.displayData();
      System.out.println("Contents of the copied object");
      Employee copyOfStd = new Employee().copyObject(std);
      copyOfStd.displayData();}}
```

instance variables

parameterized constructor initializing instance variables.

Accepts an object of the current class and initializes the instance variables with the variables of this object and returns it.

Instantiating the Employee class and making a copy by passing it as an argument to the coypObject() method.

Arguments passing

- Below mentioned are some of the different mechanisms used for passing arguments to functions:
  - value
  - reference
  - result
  - value-result
  - name
- However, the two most used and common mechanisms are pass by value and pass by reference.

Arguments passing

- **Pass by Value:** In the pass by value concept, the method is called by passing a copy of the original value. It does not affect the original parameter. Usually the parameters of simple types, i.e., primitive data types are utilized for serving the pass by value approach.
- **Pass by reference:** In the pass by reference concept, the method is called using an alias or reference of the actual parameter. It forwards the unique identifier of the object to the method. If we make any changes to the parameter's instance member, it would affect the original value.
- Note:- *Java does not support pass by reference concept.*

Arguments passing

Pass by value

```
public class Passbyvalexp
{
        int a=100;
        void update(int a)
        {
                a=a+100;
       }
       public static void main(String args[])
       {
        Passbyvalexp p=new Passbyvalexp();
                System.out.println(" Value (before change)="+p.a);
               p.update(500);
                System.out.println(" Value (after change)="+p.a);
        }
}
```

Arguments passing

Pass by value

```
public class Passbyvalexp
{
        int a=100;
        void update(int a)
        {
                a=a+100;
       }
       public static void main(String args[])
       {
        Passbyvalexp p=new Passbyvalexp();
                System.out.println(" Value (before change)="+p.a);
               p.update(500);
                System.out.println(" Value (after change)="+p.a);
        }
}
```

Value (before change)=100

Value (after change)=100

Command line arguments

- If arguments are passed at the time of running the java program, it is termed as command-line argument.
- The arguments passed from the console can be received in the java program and it can be used as an input.
- So, it provides a convenient way to check the behavior of the program for the different values. We can pass **N** (1,2,3 and so on) numbers of arguments from the command prompt.

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

```
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Objects and Classes$ j
avac Cmdlineargs.java
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Objects and Classes$ j
ava Cmdlineargs Sourajit
Your first argument is: Sourajit
```

Command line arguments

- Passing multiple arguments in the console at the same time

```
class Cmdlineargs1
{
        public static void main(String args[])
        {
                for(int i=0;i<args.length;i++)
                {
                        System.out.println(args[i]);
                }
        }
}
```

```
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Objects and Classes$ javac Cmdlineargs1.java
iitp@iitp-HP-Notebook:~/Desktop/Web-Technology/class/Java-Objects and Classes$ java Cmdlineargs1 abc 1 23
abc
1
23
```

Recursion

- Recursion in java is a process in which a method calls itself continuously. A method in java that calls itself is called recursive method.

Syntax

```
data_type methodname()
{
    //code to be executed
    methodname();//calling same method
}
```

```
public class Recurexp
{
        static void recur()
        {
                System.out.println("hello");
                recur();
        }

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

Recursion

- Recursion in java is a process in which a method calls itself continuously. A method in java that calls itself is called recursive method.

Syntax

```
data_type methodname()
{
    //code to be executed
    methodname();//calling same method
}
```

```
public class Recurexp
{
        static void recur()
        {
                System.out.println("hello");
                recur();
        }

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

```
hello
hello
hello
hello
hello
hello
hello
hello
hello
Exception in thread "main" java.lang.StackOverflowError
```

References

1. https://www.geeksforgeeks.org/type-conversion-java-examples/
2. https://www.javatpoint.com/scope-of-variables-in-java
3. https://i.stack.imgur.com/Ij3vJ.png
4. https://www.javatpoint.com/control-flow-in-java
5. https://www.scaler.com/topics/expression-in-java/
6.