**INTERNITY FOUNDATION**

**TASK 1 – JAVA BASICS**

**Submitted By:**

**Amisha Singhal**

**Java Batch**

**Define the scope of variables.**

**Ans-** Scope of a variable defines how a specific variable is accessible within the program or across classes. An assigned value to a variable makes sense and becomes usable only when the variable lies within the scope of the part of the program that executes or accesses that variable.

There are three types of [variables in java](https://www.scientecheasy.com/2020/05/variables-in-java.html/), depending on their scope:

* **local variables-** When the local variable is created inside a [method](https://www.scientecheasy.com/2020/06/java-methods.html/), [constructor](https://www.scientecheasy.com/2020/06/constructor-in-java.html/), or block, their scope only remains within the method, block, or constructor.As we exit from the method or block then the scope of a local variable is destroyed. We cannot access local variables from outside the method, constructor, or block. We cannot change their values from outside of the block.

**Eg- public class Test**

**{**

**void method1()**

**{**

**// Local variable (Method level scope)**

**int x;**

**}**

**}**

* **Instance variables-** The scope of instance variables is inside the class. They are visible inside all the methods, constructors, and from the beginning of its program block to the end of program block in the class. Therefore, all the methods, constructors, and blocks inside the class can access instance variables. Normally, it is recommended to make these variables private in the class. However, the visibility of instance variables for the sub-classes can be given with the use of access modifiers.
* **Class variables-** It is also known as **“Static Variables”.** The scope of a static variable is within the class. All the methods, constructors, and blocks inside the class can access static variables by using the class name. The visibility of the static variable is similar to the instance variable.It has the following general form:  
  **Syntax:**

**Classname.variablename;**

**Eg-** **class** Demo{

**int** d=60;//instance variable

**static** **int** s=100;//static variable

**void** method(){

**int** n=80;//local variable

}

}//end of class

**Define the structure of a Java class.**

**Ans-** Java is an [**object-oriented programming**](https://www.javatpoint.com/java-oops-concepts), **platform-independent,** and **secure** programming language that makes it popular. Using the Java programming language, we can develop a wide variety of applications.

A typical structure of a [Java](https://www.javatpoint.com/java-tutorial) program contains the following elements:

## Documentation Section-The documentation section is an important section but optional for a Java program. It includes basic information about a Java program. The information includes the date of creation, version, program name, and description of the program. It improves the readability of the program. Whatever we write in the documentation section, the Java compiler ignores the statements during the execution of the program. The comments may be single-line, multi-line, and documentation comments.

* **Package Declaration-**A class in Java can be placed in different **directories/packages** based on the module they are used. For all the classes that belong to a **single parent source directory**, a path from source directory is considered as **package declaration**.
* **Import Statements-**There can be classes written in other **folders/packages** of our working java project and also there are many classes written by individuals, companies, etc which can be useful in our program. To use them in a class, we need to **import**the class that we intend to use. Many classes can be imported in a single program and hence multiple import statements can be written.
* **Interface Section-** It is an optional section. We can create an **interface** in this section if required. We use the **interface** keyword to create an interface. An [interface](https://www.javatpoint.com/interface-in-java) is a slightly different from the class. It contains only **constants** and **method** declarations. Another difference is that it cannot be instantiated. We can use interface in classes by using the **implements** keyword. An interface can also be used with other interfaces by using the **extends** keyword.
* **Class Definition-** A name should be given to a **class**in a java file. This name is used while creating an **object of a class**, in other classes/programs.
* **Class Variables and Variables-**  In a Java program, the variables and constants are defined just after the class definition. The variables and constants store values of the parameters. It is used during the execution of the program. We can also decide and define the scope of variables by using the modifiers. It defines the life of the variables.
* **Main Method Class-** Execution of a Java application starts from the main method. In other words, its an **entry point for the class** or **program**that starts in **Java Run-time**.
* **Methods and Behaviors-** Execution of a Java application starts from the main method. In other words, its an **entry point for the class** or **program**that starts in **Java Run-time**.A set of instructions which form a **purposeful functionality** that can be required to run multiple times during the execution of a program. To not repeat the same set of instructions when the same functionality is required, the instructions are enclosed in a method. A method’s behavior can be exploited by **passing variable values** to a method.

**Eg-** package Demo; // **A package declaration**

import java.util.\*; // **declaration of an import statement**

   public class JavaStructure { **// class name**

      int repeat = 7; // **global variable**

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

      JavaStructure obj = new JavaStructure();

         obj.printMessage("Hello World!");

   }

   public void printMessage(String msg) { // **method**

      Date date = new Date(); // **variable local to method**

      for(int i = 0; i < repeat; i++) { // **Here i - variable local to for loop**

         System.out.println(msg + "From" + date.toGMTString());

      }

   }

}

**Create executable Java applications with a main method; run a Java program from the command line; produce console output.  
Ans-** import java.io.Console;  
public class Demo {  
  
    /\*  
        Accessibility, from most restrictive to least:  
        - private: Accessible only to this class (and inner classes)  
        - default: Accessible only to classes in same package  
        - protected: Accessible to subclasses and classes in the same package  
        - public: Accessible to all classes  
    
     \*/  
    private String myString;  
    public int myInt;   
    /\*  
        Using Console class requires running from command line  
     \*/  
    public void printToConsole() {  
        Console console = System.console();  
        console.writer().write("Hello ");  
        console.flush();  
        System.out.println("World!");  
    }

**Import other Java packages to make them accessible in your code.**

**Ans-** The import statement enables us to use *simple names* instead of using *fully qualified names* for classes and interfaces defined in separate packages.

import java.util.\*; // star means import all classes in a package  
// import java.sql.\*;

// this would result in an error, as could not resolve Date class, as ambiguous  
import java.sql.Date;  
import java.util.ArrayList;  
  
public class Code {  
  
    public static void main(String[] args) {  
  
        /\*  
            Need to import or use fully-qualified name  
         \*/  
        // HashMap hashMap = new HashMap();  
   java.util.HashMap hashMap = new java.util.HashMap<>();  
  
        // for the ArrayList, we have used an import  
        ArrayList arrayList = new ArrayList();  
  
           
      //  Java has two date classes.  
        java.util.Date date = new java.util.Date(); // imported java.util.\*;  
        Date date2 = new Date(2000);                // imported java.sql.Date;  
    }  
}

**Compare and contrast the features and components of Java such as: platform independence, object orientation, encapsulation, etc.**

**Ans-**

**Benefits of Java:**

**1. Simple**

Java is a simple programming language since it is easy to learn and easy to understand. Its syntax is based on C++, and it uses automatic garbage collection; therefore, we don't need to remove the unreferenced objects from memory. Java has also removed the features like explicit pointers, operator overloading, etc., making it easy to read and write.

**2. Object-Oriented**

Java uses an object-oriented paradigm, which makes it more practical. Everything in Java is an object which takes care of both data and behavior. Java uses [object-oriented concepts](https://www.javatpoint.com/java-oops-concepts) like [object](https://www.javatpoint.com/object-and-class-in-java#object), [class](https://www.javatpoint.com/object-and-class-in-java#class), [inheritance](https://www.javatpoint.com/inheritance-in-java), [encapsulation](https://www.javatpoint.com/encapsulation), [polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java), and abstraction.

**3. Secured**

Java is a secured programming language because it doesn't use Explicit pointers. Also, Java programs run inside the virtual machine sandbox. [JRE](https://www.javatpoint.com/java-jre) also provides a [classloader](https://www.javatpoint.com/classloader-in-java), which is used to load the class into [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) dynamically. It separates the class packages of the local file system from the ones that are being imported from the network.

**4. Robust**

Java is a robust programming language since it uses strong memory management. We can also handle exceptions through the Java code. Also, we can use type checking to make our code more secure. It doesn't provide explicit pointers so that the programmer cannot access the memory directly from the code.

**5. Platform independent**

Java code can run on multiple platforms directly, I.e., we need not compile it every time. It is right once, runs anywhere language (WORA) which can be converted into byte code at the compile time. The byte code is a platform-independent code that can run on multiple platforms.

**6. Multi-Threaded**

Java uses a multi-threaded environment in which a bigger task can be converted into various threads and run separately. The main advantage of multi-threading is that we need not provide memory to every running thread.

**Java platform consists of the following components:**

* **Java language**
* **The Java Development Kit (JDK)-**It is used to develop Java apps and programs. It is a [software development](https://www.upgrad.com/blog/software-development-project-ideas-topics-for-beginners/) environment and contains multiple development tools including the JRE, a compiler, an interpreter, a documentation generator, and an archiver with others.
* **The Java Runtime Environment (JRE)-** The JRE builds a runtime environment where you can execute the Java programs. It takes the Java code and combines the same with the required libraries. The JRE also initiates the JVM for its execution. The Java Runtime Environment has the required software and libraries so you can run the programs.
* **The Java Compiler**
* **The Java Virtual Machine (JVM)-** The creators of Java wanted it to be WORA (Write Once Run Anywhere). This means you can run its applications on any platform, but the thing that gives Java this quality is JVM. The JVM provides the environment to execute Java code. It interprets the byte code and converts that into machine code so the machine could run the Java program. JVM loads verify and execute the code. It also provides the runtime environment to the code so it could run in the machine.

**Eg-**import .GarbageCollection;  
  
public class Solution {  
  
    public static void main(String[] args) {  
        GarbageCollection gc = new GarbageCollection();  
        gc.runGc();  
    }  
  
}

**"1- Working With Java Data Types  
Declare and initialize variables (including casting of primitive data types)  
Differentiate between object reference variables and primitive variables  
Know how to read or write to object fields  
Develop code that uses wrapper classes such as Boolean, Double, and Integer.**

**Ans1:**

A **datatype** is an attribute of a variable which tells the compiler or interpreter how the programmer intends to use the [variable](https://www.edureka.co/blog/java-tutorial/#variables). It defines the operations that can be done on the data and what type of values can be stored.Data types are divided into two groups:

1. [**Primitive Data Types**](https://www.edureka.co/blog/data-types-in-java/#PrimitiveDataTypes)
2. [**Non-Primitive Data Types**](https://www.edureka.co/blog/data-types-in-java/#Non-PrimitiveDataTypes)

**Primitive Data Types:** A primitive data type is pre-defined by the programming language. The size and type of variable values are specified, and it has no additional methods.

There are 8 primitive data types. They are as follows:

* [**boolean** data type](https://www.edureka.co/blog/data-types-in-java/#booleandatatype)
* [**byte** data type](https://www.edureka.co/blog/data-types-in-java/#bytedatatype)
* [**char**data type](https://www.edureka.co/blog/data-types-in-java/#chardatatype)
* [**short** data type](https://www.edureka.co/blog/data-types-in-java/#shortdatatype)
* [**int** data type](https://www.edureka.co/blog/data-types-in-java/#intdatatype)
* [**long** data type](https://www.edureka.co/blog/data-types-in-java/#longdatatype)
* [**float** data type](https://www.edureka.co/blog/data-types-in-java/#floatdatatype)
* [**double** data type](https://www.edureka.co/blog/data-types-in-java/#doubledatatype)

**Non-Primitive Data Types:** These data types are not actually defined by the programming language but are created by the programmer. They are also called **“reference variables” or “object references”** since they reference a memory location which stores the data.

**Example**

int a, b, c; // Declares three ints, a, b, and c.

int a = 10, b = 10; // Example of initialization

byte B = 22; // initializes a byte type variable B.

double pi = 3.14159; // declares and assigns a value of PI.

char a = 'a'; // the char variable a iis initialized with value 'a'

**Different types of Primitive Data Types:**

**\*Implicit casting (widening conversion)**

**A data type of lower size (occupying less memory) is assigned to a data type of higher size. This is done implicitly by the JVM. The lower size is widened to higher size. This is also named as automatic type conversion.**

**Examples:**

**int x = 10;                    // occupies 4 bytes  
double y = x;                  // occupies 8 bytes  
System.out.println(y);         // prints 10.0**

**\* Explicit casting (narrowing conversion)**

A data type of higher size (occupying more memory) cannot be assigned to a data type of lower size. This is not done implicitly by the JVM and requires **explicit casting**; a casting operation to be performed by the programmer. The higher size is narrowed to lower size.

**double x = 10.5;             // 8 bytes  
int y = x;                   // 4 bytes ;  raises compilation error**

In the above code, 8 bytes double value is narrowed to 4 bytes int value. It raises error. So we will explicitly type cast it.

**double x = 10.5;   
int y = (int) x;**

The double**x** is explicitly converted to int **y**. The thumb rule is, on both sides, the same data type should exist.  
 **\* Boolean casting**

A boolean value cannot be assigned to any other data type. Except boolean, all the remaining 7 data types can be assigned to one another either implicitly or explicitly; but boolean cannot. We say, boolean is **incompatible** for conversion. Maximum we can assign a boolean value to another boolean.

Following raises error.

**boolean x = true;**

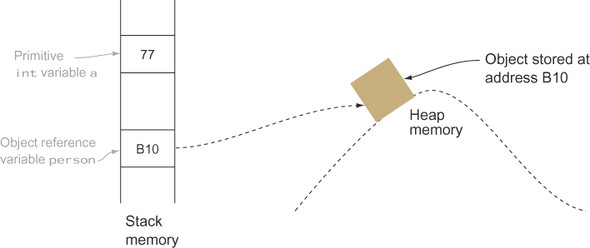
**int y = x;                    // error**

**byte –> short –> int –> long –> float –> double**In the above statement, left to right can be assigned implicitly and right to left requires explicit casting. That is,**byte** can be assigned to **short**implicitly but **short**to **byte** requires explicit casting.

**Example-A class Person is already defined. If W create an int variable a and an object reference variable person, they will store their values in memory.**

**int a = 77;**

**Person person = new Person();**

****

| **Properties** | **Primitive data types** | **Reference Variables** |
| --- | --- | --- |
| Origin | Pre-defined data types | User-defined data types |
| Stored structure | Stored in a stack | Reference variable is stored in stack and the original object is stored in heap |
| When copied | Two different variables is created along with different assignment(only values are same) | Two reference variable is created but both are pointing to the same object on the heap |
| When changes are made in the copied variable | Change does not reflect in the original ones. | Changes reflected in the original ones. |
| Default value | Primitive datatypes do not have null as default value | The default value for the reference variable is null |
| Example | byte, short, int, long, float, double, char, boolean | array, string class, interface etc. |

This variables become object or reference variables when inside instances of the class. We can populate these fields directly if they have the correct access modifier such as public.  Often we do reading and writing to fields via methods known as getters and setters or accessors and modifiers.

### Direct Access

It is the simplest form we can read and write to the fields without much effort or coding

package oca;

public class Rw {

  static String name;

  public static void main(String[] args) {

    name = "test";

    System.out.println(name);

  }

}

### In Direct Access

The object fields can be read and written using getter and setter method.The non static members of the object are required to be accessed using Object ref.These variable if private will not be accessible outside the class.

**Eg-**

package oca;

public class Rw {

private String name;

public String getter()

{

return this.name;

}

public void setter(String n)

{

this.name=n;

}

public static void main(String[] args) {

    Rw R1 = new R1();

String name=R1.setter("test");

    System.out.println(name);

  }

}

**2- Classes & Objects  
Explain an Object's Lifecycle (creation, ""dereference by reassignment"" and garbage collection).**

**Ans 2-** An object’s life cycle starts when it’s created and lasts until it goes out of scope or is no longer referenced by a variable. When an object is accessible, it can be referenced by a variable and other classes can use it by calling its methods and accessing its variables.

**An object is born**

An object comes into the picture when you use the keyword operator new. We can initialize a reference variable with this object.

public class person {}

class ObjLifeCycle{

person person1 = new person(); // Object gets created

}

**Object is accessible**

Once an object is created, it can be accessed using its reference variable. It remains accessible until it goes out of scope or its reference variable is explicitly set to null. Also, if we reassign another object to an initialized reference variable, the previous object becomes inaccessible *from that variable*.

class Exam {

String name;

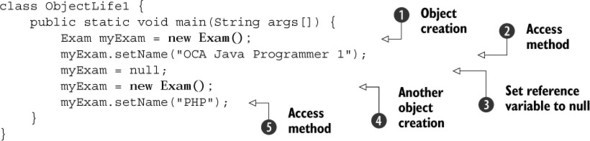
public void setName(String newName) {

name = newName;

}

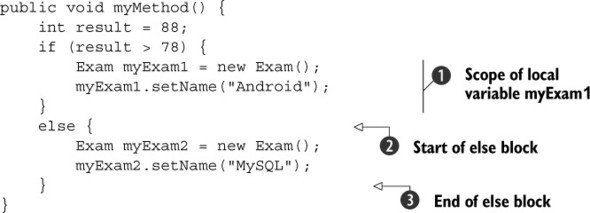
}

The class ObjectLife1 declares a variable of type Exam, creates its object, calls its method, sets it to null, and then reinitializes it:



**Object is inaccessible**

An object can become inaccessible if it goes out of scope or is dereferenced by reassignment.An object can become inaccessible if it goes out of scope:



When an object goes out of scope, it can no longer be referenced and is marked for garbage collection.

Objects are stored in java program's memory heap. Java have a process called garbage collection -- for those object no longer reachable, they are automatically deleted from memory heap. An object became unreachable when there is no reference variable refers to it or the object run out of scope.  
  
As programmer, we cannot control when garbage collection will happen. The statement System.gc(); only suggests JVM this is a good time to collect garbage. Function protected void finalize() is called once and only once when the object is garbage collected.

**3- Operator  
Using Operators and Decision Constructs  
Use Java operators; use parentheses to override operator precedence  
Test equality between Strings and other objects using == and equals ()  
Create if and if/else and ternary constructs  
Use a switch statement.**

**Ans 3-**

|  |  |
| --- | --- |
| **Operator Precedence Table** | |
| **Priority** | **Operator** |
| First (highest) | ( ) (parentheses, to group expressions) |
| [ ] (brackets, to concatenate arrays) |
| Second | . (structure field dereference) |
| [ ] (brackets, to subscript an array) |
| ( ) (parentheses, used in a function call) |
| Third | \* (pointer dereference) |
| ^ (exponentiation) |
| ++ (increment) |
| -- (decrement) |
| Fourth | \* (multiplication) |
| # and ## (matrix multiplication) |
| /(division) |
| MOD (modulus) |
| Fifth | + (addition) |
| - (subtraction and negation) |
| < (minimum) |
| > (maximum) |
| NOT (bitwise negation) |
| Sixth | EQ (equality) |
| NE (not equal) |
| LE (less than or equal) |
| LT (less than) |
| GE (greater than or equal) |
| GT (greater than) |
| Seventh | AND (bitwise AND) |
| OR (bitwise OR) |
| XOR (bitwise exclusive OR) |
| Eighth | && (logical AND) |
| || (logical OR) |
| ~ (logical negation) |
| Ninth | ?: (conditional expression) |

**Eg-**

class Precedence {

public static void main(String[] args) {

int a = 10, b = 5, c = 1, result;

result = a-++c-++b;

System.out.println(result);

}

}

**OUTPUT- 2**

The main difference between the .equals() method and == operator is that one is a method and the other is the operator.

We can use == operators for reference comparison (**address comparison**) and .equals() method for **content comparison**. In simple words, == checks if both objects point to the same memory location whereas .equals() evaluates to the comparison of values in the objects.

If a class does not [override the equals method](https://www.geeksforgeeks.org/overriding-equals-method-in-java/), then by default it uses the equals(Object o) method of the closest parent class that has overridden this method.

**The if-then Statement**

The if-then statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code *only if* a particular test evaluates to true.

**Eg-**

**class Test {**

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

**int num1 = 10, num2 = 20;**

**if (num1 < num2) System.out.println("num2 is greater than num1");**

**System.out.println("This statement is outside the body of if");**

**}**

**}**

**The if-then-else Statement**

The if-then-else statement provides a secondary path of execution when an “if” clause evaluates to false.

**Eg-**

**class Test {**

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

**int num1 = 20, num2 = 10;**

**if (num1 < num2) {**

**System.out.println("num2 is greater than num1");**

**} else {**

**System.out.println("num2 is less than num1");**

**}**

**System.out.println("This statement is outside the body of if and else");**

**}**

Unlike if-then and if-then-else statements, the switch statement can have a number of possible execution paths. A switch works with the byte, short, char, and int primitive data types. It also works with *enumerated types*, the [String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) class, and a few special classes that wrap certain primitive types: [Character](https://docs.oracle.com/javase/8/docs/api/java/lang/Character.html), [Byte](https://docs.oracle.com/javase/8/docs/api/java/lang/Byte.html), [Short](https://docs.oracle.com/javase/8/docs/api/java/lang/Short.html), and [Integer](https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html).

Eg-

**class Main {**

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

**int number = 44;**

**String size;**

**// switch statement to check size**

**switch (number) {**

**case 29:**

**size = "Small";**

**break;**

**case 42:**

**size = "Medium";**

**break;**

**// match the value of week**

**case 44:**

**size = "Large";**

**break;**

**case 48:**

**size = "Extra Large";**

**break;**

**default:**

**size = "Unknown";**

**break;**

**}**

**System.out.println("Size: " + size);**

**}**

**}**

**4- Array  
Creating and Using Arrays  
Declare, instantiate, initialize and use a one-dimensional array  
Declare, instantiate, initialize and use multi-dimensional arrays.**

**Ans 4-**A one-dimensional array is an object that refers to a collection of scalar values. A two-dimensional (or more) array is referred to as a multidimensional array. A two-dimensional array refers to a collection of objects in which each of the objects is a one-dimensional array. Multidimensional arrays may or may not contain the same number of elements in each row or column.

We can declare, instantiate and initialize the java array together by:

**int** a[]={23,9,1,65};//declaration, instantiation and initialization

**Eg-**

**class** Testarray{

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

**int** a[]=**new** **int**[5];//declaration and instantiation

a[0]=10;//initialization

a[1]=20;

a[2]=80;

a[3]=90;

a[4]=30;

//traversing array

**for**(**int** i=0;i<a.length;i++)//length is the property of array

System.out.println(a[i]);

}}

**OUTPUT-**

**10**

**20**

**80**

**90**

**30**

**Syntax to Declare Multidimensional Array in Java**

**data\_type array\_name[][];**

**Example to instantiate Multidimensional Array in Java**

**int**[][] arr=**new** **int**[3][3];//3 row and 3 column

**Example to initialize Multidimensional Array in Java**

arr[0][0]=1;

arr[0][1]=2;

arr[0][2]=3;

arr[1][0]=4;

arr[1][1]=5;

arr[1][2]=6;

arr[2][0]=7;

arr[2][1]=8;

arr[2][2]=9;

**Eg of Multidimensional Array**

//Java Program to illustrate the use of multidimensional array

**class** Testarray{

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

//declaring and initializing 2D array

**int** arr[][]={{1,2,3},{2,4,5},{4,4,5}};

//printing 2D array

**for**(**int** i=0;i<3;i++){

**for**(**int** j=0;j<3;j++){

   System.out.print(arr[i][j]+" ");

 }

 System.out.println();

}

}}

**5- Iteration Statements  
Using Loop Constructs  
Create and use while loops  
Create and use for loops including the enhanced for loop  
Create and use do/while loops  
Compare loop constructs  
Use break and continue**

**Ans 5-**

##### **While loop**

A while loop is used to repeatedly execute a set of statements as long as its condition evaluates to true. This loop checks the condition *before* it starts the execution of the statement.

**Eg-**

**public** **class** WhileEg {

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

**int** i=1;

**while**(i<=10){

        System.out.println(i);

    i++;

    }

}

}

**Output- 1 to 10**

#### The do-while loop

A do-while loop is used to repeatedly execute a set of statements until the condition that it uses evaluates to false. This loop checks the condition *after*it completes the execution of all the statements in its loop body.

**Eg:**

public class DoWhile {

public static void main(String[] args) {

    int i=1;

    do{

        System.out.println(i);

    i++;

    }while(i<=10);

}

}

**Output- 1 to 10**

#### For and Enhanced For loop

A for loop is *usually*used to execute a set of statements a fixed number of times. It takes the following form:

**for (initialization; condition; update) {**

**statements;**

**}**

**Eg-**

//Java For-each loop example which prints the

//elements of the array

public class ForEach {

public static void main(String[] args) {

    //Declaring an array

    int arr[]={12,23,44,56,78};

    //Printing array using for-each loop

    for(int i:arr){

        System.out.println(i);

    }

}

}

**OUTPUT-12 23 44 56 78**

**Comparing loop constructs:**

* Both the do-while and while loops execute a set of statements until the termination condition evaluates to false. The only difference between these two statements is that the do-while loop executes the code at least once, even if the condition evaluates to false.
* The regular for loop, though cumbersome to use, is much more powerful than the enhanced for loop.
* The enhanced for loop can’t be used to initialize an array and modify its elements. The enhanced for loop can’t be used to delete or remove the elements of a collection.
* The enhanced for loop can’t be used to iterate over multiple collections or arrays in the same loop.
* You should try to use a for loop when you know the number of iterations.For example, iterating through a collection or an array, or executing a loop for a fixed number of times, say, to ping a server five times.

**Loop statements (break and continue):**

* The break statement is used to *exit* or *break out of* the for, for-each, do, and do-while loops and the switch construct.
* The continue statement is used to skip the remaining steps in the current iteration and start with the next loop iteration. The continue statement works with the for, for-each, do, and do-while loops and the switch construct.
* When we use the break statement with nested loops, it exits the corresponding loop.
* When we use the continue statement with nested loops, it exits the current iteration of the corresponding loop.

**Eg of Break Statement:**

**public** **class** BreakEg {

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

    //using for loop

**for**(**int** i=1;i<=10;i++){

**if**(i==5){

            //breaking the loop

**break**;

        }

        System.out.println(i);

    }

}

}

**OUTPUT -1 2 3 4**

**Eg of Continue Statement:**

public class ContinueEg {

public static void main(String[] args) {

    //for loop

    for(int i=1;i<=8;i++){

        if(i==5){

            //using continue statement

            continue;//it will skip the rest statement

        }

        System.out.println(i);

    }

}

**OUTPUT- 1 2 3 4 6 7 8**

**=============================END========================================**