JAVA PROGRAMMING

[Subject Code: 314317]

DEPARTMENT OF COMPUTER TECHNOLOGY

MAULANA MUKHTAR AHMAD NADVI TECHNICAL CAMPUS, MALEGAON

Java – James Gosling

"James Gosling is a famous Canadian software developer who has been with Sun Microsystems since 1984 and is considered as father of Java programming language. Gosling did the original design of Java and implemented its original compiler and virtual machine."

Unit I: Basic Syntactical Constructs in Java

Introduction

Object-Oriented Programming supports all features of normal programming languages. In addition, it supports some important concepts and terminologies which has made it a popular programming language. OOP allows to divide the problem into number of entities called objects and then build the data and functions around these entities. The combination of data and methods creates object. The data of an object can be only accessed by the methods associated by that object. However, method of one object can access method of another object. C++ is basically a procedural language with object-oriented extension. Java is purely object-oriented language.

1.1 History of Java

- In 1990, Sun Micro Systems Inc. (US) was conceived a project to develop software for consumer electronic devices that could be controlled by a remote.
- This project was called Stealth Project but later its name was changed to Green Project. In January 1991, Project Manager James Gosling and his team members Patrick Naughton, Mike Sheridan, Chris Wrath, and Ed Frank met to discuss about this project. Gosling thought C and C++ would be used to develop the project, but the problem he faced with them is that they were system dependent languages.
- They are designed to be compiled for a specific target and could not be used on various processors, which the electronic devices might use.
- James Gosling with his team started developing a new language, which was completely System independent. This language was initially called OAK.
- Since this name was registered by some other company, later it was changed to Java.
- James Gosling and his team members were consuming a lot of coffee while developing language.
- Good quality of coffee was supplied from a place called "Java Island'. Hence, they fixed the name of the language as Java. The symbol for Java language is cup and saucer.
- Sun formally announced Java at Sun World conference in 1995. On January 23rd 1996, JDK1.0 version was released.

Features of Java

Object Oriented: Java is an object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporate both data and behaviour.

Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

Platform Independent:
ava is platform independent because it is different from other languages like C, C++, etc. which are compiled into platform specific machines while Java is a write once, run anywhere language.
Java code can be executed on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere (WORA).

Simple: Java is designed to be easy to learn. If you understand the basic concept of OOP java would be easy to master.

Secured:
Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

- **No explicit pointer**
- **Java Programs run inside a virtual machine sandbox**

Classloader: Classloader in Java is a part of the Java Runtime Environment (JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.

Bytecsode Verifier: It checks the code fragments for illegal code that can violate access rights to objects.

Security Manager: It determines what resources a class can access such as reading and writing to the local disk.

Java language provides these securities by default. Some security can also be provided by an application developer explicitly through SSL, JAAS, Cryptography, etc.

With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.

Architectural- neutral: Java compiler generates an architecture-neutral object file format which makes the compiled code to be executable on many processors, with the presence Java runtime system.

Portable: Being architectural neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler and Java is written in ANSI C with a clean portability boundary which is a POSIX subset.

Robust

- It uses strong memory management.
- There is a lack of pointers that avoids security problems.
- Java provides automatic garbage collection which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
- There are exception handling and the type checking mechanism in Java. All these points make Java robust.

Multi-threaded: With Java's multi-threaded feature it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications.

Interpreted: Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light weight process.

High Performance: With the use of Just-In-Time compilers Java enables high performance.
Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++). Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

Distributed: Java is designed for the distributed environment of the internet. Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

Dynamic: Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run- time information that can be used to verify and resolve accesses to objects on run-time.

Java Environment

JDK (Java Development Kit) can be downloaded from Sun Microsystems web site: http://www.oracle.com/technetwork/java/javase/downloads/index.html, which also contains java compiler and runtime engine. Install JDK after downloading. By default JDK will be installed in C:\Program Files\Java\jdk1.7.0_05 or C:\Program Files (x86)\Java\jdk1.7.0_05

Setting up Java Environment

After installing the JDK, we need to set at least one environment variable to compile and run Java programs. A PATH environment variable enables the operating system to find the JDK executables when working directory (working directory means the directory in which we are saving java program) is not the JDK's binary directory.

Setting environment variables from a command prompt: If we set the variables from a command prompt, they will only hold for that session. To set the PATH from a command prompt:

```
set path=C:\Program Files\Java\jdk1.7.0_05
```

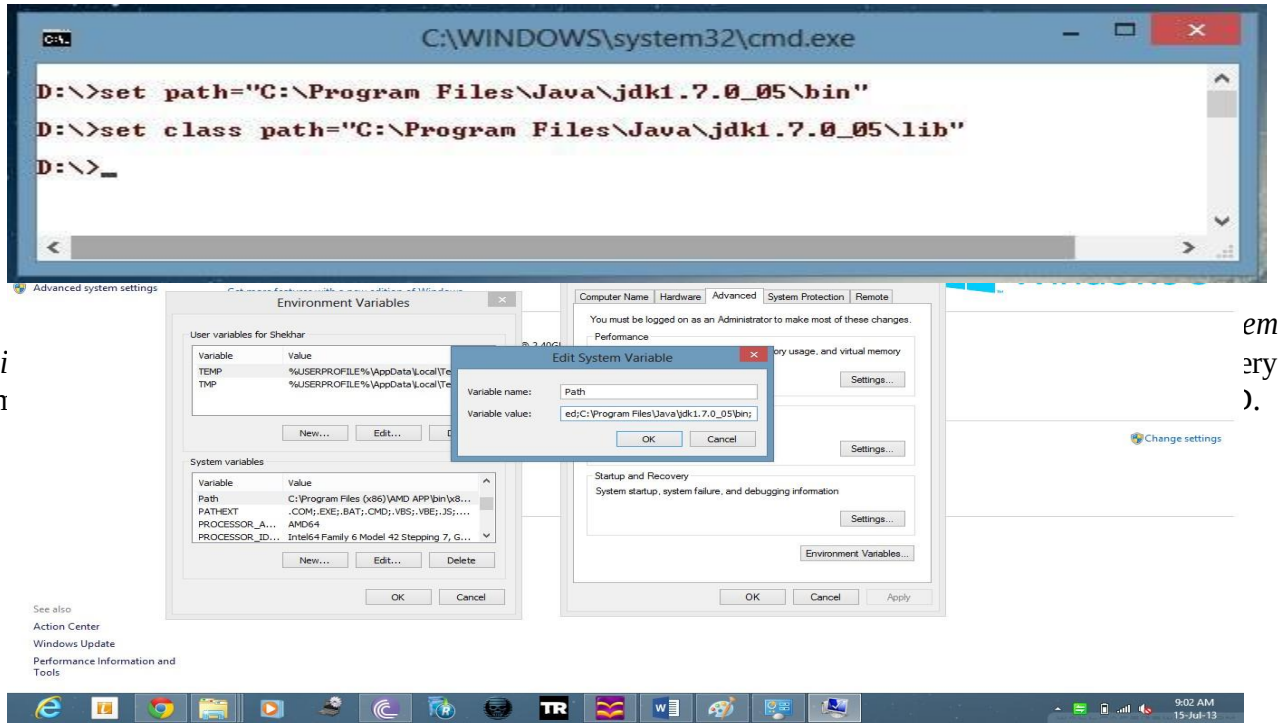


Fig. *em*
vari *ery*
tin *).*

Fig. Setting up environmental variables as a system variable

Following are steps for setting environment variables as a system variable:

1. Right-click on My Computer
2. Choose Properties
3. Select the Advanced tab
4. Click the Environment Variables button at the bottom
5. In system variables tab, select path (system variable) and click on edit button
6. A window with variable name-path and its value will be displayed.
7. Don't disturb the default path value that is appearing and just append (add) to that pathat the end:
8. ; C:\Program Files\Java\jdk1.7.0_05;
9. Finally press OK button.

Class in Java

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

Syntax of Class

```
class <class_name>{
   field;
   method;
}
```

Object and Class Example: main within the class:

```
//Java Program to illustrate how to define a class and fields
//Defining a Student class.
class Student{
  int id=20;     //field or data member or instance variable
 String name;
            //creating main method inside the Student class
public static void main(String args[]){
  Student s1=new Student();     //creating an object of Student
                            //Printing values of the object
System.out.println(s1.id); //accessing   member   through   reference
variable
  System.out.println(s1.name);
 }
}
```

Output:

```
id=20
null
```

Explanation

Two fields are defined for the Student class in this Java program: an int type for the id and a string type for the name. The Student class itself defines the primary method. The new keyword is used to create an object s1 of type Student inside the main procedure. The fields' default values-20 for int and null for String-are printed because they are not explicitly initialized. The values of the s1 object's name and id fields are finally printed by the program.

Object and Class Example: main( ) Method Outside the Class

In real-world development, it is usual practice to organize Java classes into distinct files and to place the main method outside of the class it is intended to execute from. This strategy improves the readability,

maintainability, and reusability of the code.

In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.

We can have multiple classes in different Java files or single Java file. If you define multiple classes in a single Java source file, it is a good idea to save the file name with the class name which has main( ) method.

File: TestStudent1.java

```
/Java Program to demonstrate having the main method in
//another class
//Creating Student class.
class Student{
 int id;
 String name;
}
//Creating another class TestStudent1 which contains the main method
class TestStudent1{
 public static void main(String args[]){
  Student s1=new Student();
  System.out.println(s1.id);
  System.out.println(s1.name);
 }
}
```

Output:

```
id = 20
null
```

Explanation

The main method in this Java program is shown to be in a different class than the Student class. There are no methods defined for the two fields, name and id, in the Student class. The main method then resides in another class called TestStudent1, where the default constructor is used to generate an object s1 of type Student. The fields name and id are written with their default values, which are null for String and 0 for int, since they are not explicitly initialized.

Object:

An object is a real-world entity.

An object is a runtime entity.

The object is an entity which has state and behavior.

The object is an instance of a class.

Initializing Object in Java

There are the following three ways to initialize object in Java.

- By reference variable
- By method
- By constructor

1) Initialization through Reference Variable

Initializing an object means storing data in the object. Let's see a simple example where we are going to initialize the object through a reference variable.

File: TestStudent2.java

```
class Student{
 int id;
 String name;
}
class TestStudent2{
 public static void main(String args[]){
  Student s1=new Student();
  s1.id=1726;
  s1.name="MMANTC";
  System.out.println(s1.id+" "+s1.name);
                       //printing members with a white space
 }
}
```

1726 MMANTC

Explanation

There are two classes in this Java code: Student and TestStudent2. The two fields that the Student class defines, id and name, stand for the student's ID and name, respectively. The main method, which is the program's entry point, is specified in the TestStudent2 class. The new keyword is used to create an object s1 of type Student inside the main procedure. Next, values 101 and "Sonoo" are initialised in the id and name fields of s1.

We can also create multiple objects and store information in it through reference variable.

```
File: TestStudent3.java
class Student{
 int id;
 String name;
}
class TestStudent3{
 public static void main(String args[]){
  //Creating objects
  Student s1=new Student();
  Student s2=new Student();
  //Initializing objects
  s1.id=1726;
  s1.name="MMANTC";
  s2.id=4116;
  s2.name="Computer";
  //Printing data
  System.out.println(s1.id+" "+s1.name);
  System.out.println(s2.id+" "+s2.name);
 }
}
```

output

```
1726 MMANTC
4116 Computer
```

Explanation

This Java code shows how to create and initialise many Student class objects in the TestStudent3 class. Using the new keyword, two Student objects, s1 and s2, are first created. Subsequently, each object's name and id fields are initialized independently. Id is set to 101, and the name is set to "Sonoo" for S1, and 102 and name are set to "Amit" and S2, respectively.

2) Initialization through Method

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method.

Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

File: TestStudent4.java

```
class Student{
 int rollno;
 String name;
 void insertRecord(int r, String n){
  rollno=r;
  name=n;
 }
 void displayInformation(){System.out.println(rollno+" "+name);}
}
class TestStudent4{
 public static void main(String args[]){
  Student s1=new Student();
  Student s2=new Student();
  s1.insertRecord(111,"Karan");
  s2.insertRecord(222,"Aryan");
  s1.displayInformation();
  s2.displayInformation();
 }
}
```

output

```
111 Karan
222 Aryan
```

Explanation

The provided Java code includes two classes: Student and TestStudent4. The Student class includes rollno and name fields, along with the methods insertRecord and displayInformation to initialise and print the respective fields' data. Two Student objects are created in the main method of the TestStudent4 class, and their corresponding insertRecord methods are called to set their rollno and name.

3) Initialization through a Constructor

The concept of object initialization through a constructor is essential to object-oriented programming in Java. Special methods inside a class called constructors are called when an object of that class is created with the new keyword. They initialise the state of objects by entering initial values in their fields or carrying out any required setup procedures. The constructor is automatically invoked upon object instantiation, guaranteeing correct initialization of the object prior to usage.

Here's an example demonstrating object initialization through a constructor:

File: ObjectConstructor.java

```
class Student {
    int id;
    String name;
    // Constructor with parameters
    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }
```

```
    // Method to display student information
    public void displayInformation() {
        System.out.println("Student ID: " + id);
        System.out.println("Student Name: " + name);
    }
}
public class ObjectConstructor {
    public static void main(String[] args) {
        // Creating objects of Student class with constructor
        Student student1 = new Student(1, "John Doe");
        Student student2 = new Student(2, "Jane Smith");
        // Displaying information of the objects
        student1.displayInformation();
        student2.displayInformation();
    }
}
```

Output:

```
Student ID: 1
Student Name: John Doe
Student ID: 2
Student Name: Jane Smith
```

Explanation

In this example, the id and name fields of a Student object are initialised using a constructor defined by the Student class, which accepts two parameters: id and name. Upon creating objects student1 and student2 using this constructor, the fields of each are initialised with the values supplied. This method makes ensuring that objects are created with the correct starting values, which makes it easier to instantiate and use objects later on in the programme.

Object and Class Example: Employee

Let's see an example where we are maintaining records of employees.

File: TestEmployee.java

```
class Employee{
    int id;
    String name;
    float salary;
    void insert(int i, String n, float s) {
        id=i;
        name=n;
        salary=s;
    }
    void display(){System.out.println(id+" "+name+" "+salary);}
}
public class TestEmployee {
public static void main(String[] args) {
    Employee e1=new Employee();
    Employee e2=new Employee();
    Employee e3=new Employee();
    e1.insert(101,"ajeet",45000);
    e2.insert(102,"irfan",25000);
    e3.insert(103,"nakul",55000);
    e1.display();
    e2.display();
    e3.display();
```

```
}
}
```

Output:

```
101 ajeet 45000.0
102 irfan 25000.0
103 nakul 55000.0
Explanation
```

The employee class in this Java code has three fields: id, name, and salary. It also has two methods: insert, which sets the values for these fields, and display, which prints the values. The main function of the TestEmployee class creates three Employee objects (e1, e2, and e3). To initialise the id, name, and salary fields of each object with precise values, the insert method is called on each of the objects. Then, each object's display method is called, displaying object initialization and information display via method invocation. Each object's id, name, and salary values are printed to the console.

Object and Class Example: Rectangle

There is given another example that maintains the records of Rectangle class.

```
File: TestRectangle1.java

class Rectangle{
 int length;
 int width;
  void insert(int l, int w)
  {
        length=l;
        width=w;
  }
  void calculateArea()
  {
        System.out.println(length*width);
  }
}
class TestRectangle1{
 public static void main(String args[]){
  Rectangle r1=new Rectangle();
  Rectangle r2=new Rectangle();
  r1.insert(11,5);
  r2.insert(3,15);
  r1.calculateArea();
  r2.calculateArea();
}
}
Output:
55
45
```

Explanation

This Java code defines a Rectangle class with fields for length and width, along with methods to insert dimensions and calculate the area. In the TestRectangle1 class's main method, two Rectangle objects are instantiated, and their dimensions are set using the insert method.

Different ways to create an object in Java:

There are the following ways to create an object in Java.

1. By new keyword

The most common way to create an object in Java is by using the new keyword followed by a constructor. For example: ClassName obj = new ClassName();. This allocates memory for the object and calls its constructor to initialize it.

2. By newInstance() method

This method is part of the java.lang.Class class and is used to create a new instance of a class dynamically at runtime. It invokes the no-argument constructor of the class.
For example: ClassName obj = (ClassName) Class.forName("ClassName").newInstance();.

3. By clone() method

The clone() method creates a copy of an existing object by performing a shallow copy. It returns a new object that is a duplicate of the original object. For example: ClassName obj2 = (ClassName) obj1.clone();.

4. By deserialization

Objects can be created by deserializing them from a stream of bytes. This is achieved using the ObjectInputStream class in Java. The serialized object is read from a file or network, and then the readObject() method is called to recreate the object.

5. By factory method

Factory methods are static methods within a class that return instances of the class. They provide a way to create objects without directly invoking a constructor and can be used to encapsulate object creation logic.
For example: `ClassName obj = ClassName.createInstance().`

File: TestAccount.java

```
//Java Program to demonstrate the working of a banking-system
//where we deposit and withdraw amount from our account.
//Creating an Account class which has deposit() and withdraw() methods
class Account{
int acc_no;
String name;
float amount;
//Method to initialize object
void insert(int a,String n,float amt){
acc_no=a;
name=n;
amount=amt;
}
//deposit method
void deposit(float amt){
amount=amount+amt;
System.out.println(amt+" deposited");
}
//withdraw method
void withdraw(float amt){
if(amount<amt){
System.out.println("Insufficient Balance");
}else{
```

```
amount=amount-amt;
System.out.println(amt+" withdrawn");
}
}
//method to check the balance of the account
void checkBalance(){System.out.println("Balance is: "+amount);}
//method to display the values of an object
void display(){System.out.println(acc_no+" "+name+" "+amount);}
}
//Creating a test class to deposit and withdraw amount
class TestAccount{
public static void main(String[] args){
Account a1=new Account();
a1.insert(832345,"Ankit",1000);
a1.display();
a1.checkBalance();
a1.deposit(40000);
a1.checkBalance();
a1.withdraw(15000);
a1.checkBalance();
}
}
```

Output:

```
832345 Ankit 1000.0
Balance is: 1000.0
40000.0 deposited
Balance is: 41000.0
15000.0 withdrawn
Balance is: 26000.0
```

Explanation

With methods for depositing, withdrawing, checking balance, and showing account details, the Account class in this Java programme emulates a simple banking system. Other characteristics include account number, name, and amount. The creation, initialization, and presentation of an Account object a1 with account information take place in the main method of the TestAccount class. Next, the account is used for deposits and withdrawals, and after each transaction, the balance is checked.

Java Tokens:

In Java, Tokens are the smallest elements of a program that is meaningful to the compiler. They are also known as the fundamental building blocks of the program. Tokens can be classified as follows:

1) Keywords
2) Identifiers
3) Constants/Literals
4) Operators
5) Separators

1. Keyword

Keywords are pre-defined or reserved words in a programming language. Each keyword is meant to perform a specific function in a program. Since keywords are referred names for a compiler, they can't be used as variable names because by doing so, we are trying to assign a new meaning to the keyword which is not allowed. Java language supports the following keywords:

abstract	assert	boolean	break	byte
case	catch	char	class	const
continue	default	do	double	else
enum	exports	extends	final	finally
float	for	goto	if	implements
import	instanceof	int	interface	long
module	native	new	open	opens
private	protected	provides	public	Requires
short	static	strictfp	super	switch
synchronized	this	throw	throws	To
transient	transitive	try	uses	Void
volatile	while	with		

Identifier: Identifiers are used to name a variable, constant, function, class, and array. It usually defined by the user. It uses letters, underscores, or a dollar sign as the first character. The label is also known as a special kind of identifier that is used in the goto statement. Remember that the identifier name must be different from the reserved keywords. There are some rules to declare identifiers are:
The first letter of an identifier must be a letter, underscore or a dollar sign. It cannot start with digits but may contain digits.
The whitespace cannot be included in the identifier.
Identifiers are case sensitive.

Literals: In programming literal is a notation that represents a fixed value (constant) in the source code. It can be categorized as an integer literal, string literal, Boolean literal, etc. It is defined by the programmer. Once it has been defined cannot be changed. Java provides five types of literals are as follows:

- Integer
- Floating Point
- Character
- String
- Boolean

Constants/Literals:

Constants are also like normal variables. But the only difference is, their values cannot be modified by the program once they are defined. Constants refer to fixed values. They are also called as literals. Constants may belong to any of the data type. Syntax:

final data_type variable_name;

```
import java.io.*;
class GFG {
    public static void main (String[] args) {

        // Here final keyword is used
        // to define the constant PI
```

```
        final double PI = 3.14; // Use double instead of int

        // Example usage of PI
        System.out.println("The value of PI is: " + PI);
    }
}
```

Output

```
The value of PI is: 3.14
```

Operators:

Java provides many types of operators which can be used according to the need. They are classified based on the functionality they provide. Some of the types are-

Arithmetic Operators:

Unary Operators

Assignment Operator

Relational Operators

Logical Operators

Ternary Operator

Bitwise Operators

Shift Operators

instance of operator

Precedence and Associativity

Separators:

Separators are used to separate different parts of the codes. It tells the compiler about completion of a statement in the program. The most commonly and frequently used separator in java is semicolon (;).

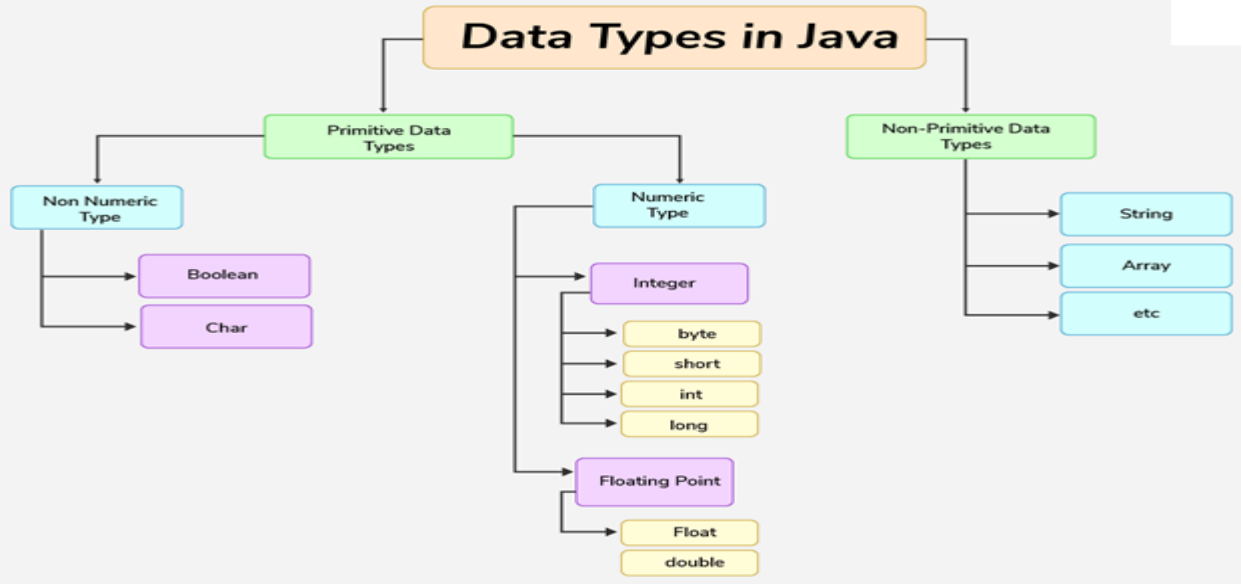
`int variable;` //here the semicolon (;) ends the declaration of the variable

```
import java.io.*;
class GFG {
    public static void main (String[] args) {
              //Here the semicolon (;) used to end the print statement
        System.out.println("GFG!");
    }
}
```

Output

```
GFG!
```

Basic Datatypes



Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

There are two data types available in Java:

1. Primitive Data Types
2. Reference/Object Data Types

Primitive Data Types:

There are eight primitive data types supported by Java. Primitive data types are predefined by the language and named by a key word. Let us now look into detail about the eight primitive datatypes.

1. byte:

The byte data type in Java is a primitive data type that represents an 8-bit signed two's complement integer. One common use of the **byte** data type is in reading and writing binary data, such as files or network streams. Since binary data is often represented using bytes, the **byte** data type provides a convenient way to work with such data. Additionally, the **byte** data type is sometimes used in performance-critical applications where memory usage needs to be minimized.

Example: byte a = 100, byte b = -50

2. short:

- Short data type is a 16-bit signed two's complement integer.
- Minimum value is -32,768 (-2^15)
- Maximum value is 32,767(inclusive) (2^15 -1)
- Short data type can also be used to save memory as byte data type. A short is 2 timessmaller than an int
- Default value is 0.
  Example: short s= 10000 , short r = -20000

3. int:

- Int data type is a 32-bit signed two's complement integer.
- Minimum value is - 2,147,483,648.(-2^31)
- Maximum value is 2,147,483,647(inclusive).(2^31 -1)
- Int is generally used as the default data type for integral values unless there is aconcern

about memory.

- The default value is 0.

Example-+ : int a = 100000, int b = -200000

4. long:

- Long data type is a 64-bit signed two's complement integer.
- Minimum value is -9,223,372,036,854,775,808.(-2^63)
- Maximum value is 9,223,372,036,854,775,807 (inclusive). (2^63 -1)
- This type is used when a wider range than int is needed.
- Default value is 0L.
  Example: long a = 100000L, int b = -200000L

5. float:

- Float data type is a single-precision 32-bit IEEE 754 floating point.
- Float is mainly used to save memory in large arrays of floating point numbers.
- Default value is 0.0f.
- Float data type is never used for precise values such as currency.Example : float f1 = 234.5f

6. double:

- double data type is a double-precision 64-bit IEEE 754 floating point.
- This data type is generally used as the default data type for decimal values. generallythe default choice.
- Double data type should never be used for precise values such as currency.
- Default value is 0.0d.
  Example: double d1 = 123.4

7. boolean:

- boolean data type represents one bit of information.
- There are only two possible values: true and false.
- This data type is used for simple flags that track true/false conditions.
- Default value is false.
  Example: boolean one = true

8. char:

- char data type is a single 16-bit Unicode character.
- Minimum value is '\u0000' (or 0).
- Maximum value is '\uffff' (or 65,535 inclusive).
- Char data type is used to store any character.
  Example. char letterA ='A'

Filename: datatype.java

```
class datatype
{
    public static void main(String args[])
    {
        char a = 'G';
        int i = 89;
        byte b = 4;
        short s = 56;
```

```
        double d = 4.355453532;
        float f = 4.7333434f;
        long l = 12121;

        System.out.println("char: " + a);
        System.out.println("integer: " + i);
        System.out.println("byte: " + b);
        System.out.println("short: " + s);
        System.out.println("float: " + f);
        System.out.println("double: " + d);
        System.out.println("long: " + l);
    }
}
```

Non-Primitive Data Types in Java

In Java, non-primitive data types, also known as reference data types, are used to store complex objects rather than simple values. Unlike primitive data types that store the actual values, reference data types store references or memory addresses that point to the location of the object in memory. This distinction is important because it affects how these data types are stored, passed, and manipulated in Java programs.

Class

One common non-primitive data type in Java is the class. Classes are used to create objects, which are instances of the class. A class defines the properties and behaviors of objects, including variables (fields) and methods. For example, you might create a Person class to represent a person, with variables for the person's name, age, and address, and methods to set and get these values.

Interface

Interfaces are another important non-primitive data type in Java. An interface defines a contract for what a class implementing the interface must provide, without specifying how it should be implemented. Interfaces are used to achieve abstraction and multiple inheritance in Java, allowing classes to be more flexible and reusable.

Arrays

Arrays are a fundamental non-primitive data type in Java that allow you to store multiple values of the same type in a single variable. Arrays have a fixed size, which is specified when the array is created, and can be accessed using an index. Arrays are commonly used to store lists of values or to represent matrices and other multi-dimensional data structures.

Enum

Java also includes other non-primitive data types, such as enums and collections. Enums are used to define a set of named constants, providing a way to represent a fixed set of values. Collections are a framework of classes and interfaces that provide dynamic data structures such as lists, sets, and maps, which can grow or shrink in size as needed.

Overall, non-primitive data types in Java are essential for creating complex and flexible programs. They allow you to create and manipulate objects, define relationships between objects, and represent complex data structures. By understanding how to use non-primitive data types effectively, you can write more efficient and maintainable Java code.

Java Literals

Java Literals are syntactic representations of Boolean, character, numeric, or string data. Literals provide a means of expressing specific values in java program. A literal is a source code representation of a fixed value. They are represented directly in the code without any computation. Literals can be

assigned to any primitive type variable. For example:

```
byte a = 68;
char a = 'A'
```

byte, int, long, and short can be expressed in decimal(base 10), hexadecimal(base 16) oroctal(base 8) number systems as well.

Prefix 0 is used to indicate octal and prefix 0x indicates hexadecimal when using these numbersystems for literals. For example:

```
int decimal = 100;
int octal  0144;
int hexa =  0x64;
```

String literals in Java are specified like they are in most other languages by enclosing a sequenceof characters between a pair of double quotes. Examples of string literals are:

```
"Hello World"
"two\nlines"
"\"This is in quotes\""
```

String and char types of literals can contain any Unicode characters. For example:

```
char a = '\u0001';
String a = "\u0001";
```

1.2 Java Symbolic Constants

Symbolic constants are named constants like:

```
final double PI = 3.14 ;
```

These are constants because of the 'final' keywords, so they can NOT be reassigned a new valueafter being declared as final and they are symbolic, because they have a name.
A NON symbolic constant is like the value of '2' in expression int multi = 2 * 3

Scope of variable

Variables are an essential part of data storage and manipulation in the realm of programming. In addition to making values available within a programme, they offer a means of holding them temporarily. Not all variables, though, are made equally. Each variable has a scope that specifies how long it will be seen and used in a programme. Java code must be efficient and error-free, which requires an understanding of variable scope.

There are four scopes for variables in Java:
local,
instance,
class,
method parameters.

Local Variables:
Local variables are those that are declared inside of a method, constructor, or code block. Only the precise block in which they are defined is accessible. The local variable exits the block's scope after it has been used, and its memory is freed. Temporary data is stored in local variables, which are frequently initialised in the block where they are declared. The Java compiler throws an error if a local

variable is not initialised before being used. The range of local variables is the smallest of all the different variable types.

```
public SumExample
{
public void calculateSum() {
   int a = 5; // local variable
   int b = 10; // local variable
   int sum = a + b;
   System.out.println("The sum is: " + sum);
} // a, b, and sum go out of scope here
}
Output:The sum is: 15
```

Instance Variables:

Within a class, but outside of any methods, constructors, or blocks, instance variables are declared. They are accessible to all methods and blocks in the class and are a part of an instance of the class. If an instance variable is not explicitly initialised, its default values are false for boolean types, null for object references, and 0 for numeric kinds. Until the class instance is destroyed, these variables' values are retained.

```
public class Circle {
   double radius; // instance variable
   public double calculateArea() {
      return Math.PI * radius * radius;
   }
}
```

Class Variables (Static Variables):

In a class but outside of any method, constructor, or block, the static keyword is used to declare class variables, also referred to as static variables. They relate to the class as a whole, not to any particular instance of the class. Class variables can be accessed by using the class name and are shared by all instances of the class. Like instance variables, they have default values, and they keep those values until the programme ends.

```
public class Bank {
   static double interestRate; // class variable
   // ...
}
```

Method Parameters:

Variables that are supplied to a method when it is invoked are known as method parameters. They serve as inputs for method execution and are used to receive values from the caller. The scope of method parameters is restricted to the method in which they are defined, making them local to that method. The values of the arguments given are allocated to the respective parameters when a method is called.

```
public void printName(String name) { // name is a method parameter
   System.out.println("Hello, " + name + "!");
}
```

Loop variable (Block scope):

```
public class MainClass{
public static void main(String[] args){
  for (int x = 0; x < 5; x++)
  {
       System.out.println(x);
  }
 }
}
```

Output:

```
0
```

```
1
2
3
4
```

Example 2:

```
public class MainClass {
public static void main(String[] args) {
  for (int x = 0; x < 5; x++)
  {
  for (int y = 0; y < 3; y++)
       {
            System.out.println(x); System.out.println(y);
       }
  }
 }
}
```

Type Casting & Conversion

Java supports two type of type Casting – Primitive Data Type Casting & Reference Type Casting. Reference type casting means assigning one Java object to another Java object. It comes with very strict rules.

Java Data Type Casting comes with two flavors:

1. Implicit Casting (widening conversion)
2. Explicit Casting (narrowing conversion)

Implicit Casting (Widening Conversion)

A data type of lower size (requires less memory) is assigned to a data type of higher memory size. This is done implicitly (automatically) by JVM. The lower size is widened to higher size. This is also called as automatic type conversion.

Examples:

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

In above code 4-byte integer value is assigned to 8-byte double value.

Explicit Type Casting (Narrowing Conversion)

A data type of higher size cannot be assigned to a data type of lower size. This is not doneimplicitly by JVM and requires explicit casting; a casting is performed manually by theprogrammer. The higher size is narrowed to lower size.

```
double x = 10.5; // 8 bytes
       // 4 bytes ; raises compilation
```

In above code 8-byte double value is narrowed to 4-byte int value but it will generate an error.So, this can be done manually as:

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

Operators and Expressions

Java provides a rich set of operators to manipulate variables. We can divide all the Java operatorsinto the following groups:

1 Arithmetic Operators
2 Relational Operators
3 Bitwise Operators
4 Logical Operators
5 Assignment Operators
6 Misc Operators

Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used inalgebra. The following table lists the arithmetic operators:
Assume integer variable A holds 10 and variable B holds 20 then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	A + B will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	A - B will give -10
*	Multiplication - Multiplies values on either side of the operator	A * B will give 200
/	Division - Divides left hand operand by right hand operand	B / A will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	B % A will give 0
++	Increment - Increase the value of operand by 1	B++ gives 21
--	Decrement - Decrease the value of operand by 1	B-- gives 19

The following simple example program demonstrates the arithmetic operators. Copy and pastefollowing Java program in Test.java file and compile and run this program:

```
public class Test {
public static void main(String args[]) {
int a = 10;
int b = 20;
int c = 25;
int d = 25;
System.out.println("a + b = " + (a + b) );
System.out.println("a - b = " + (a - b) );
System.out.println("a * b = " + (a * b) );
System.out.println("b / a = " + (b / a) );
System.out.println("b % a = " + (b % a) );
System.out.println("c % a = " + (c % a) );
System.out.println("a++    = " + (a++) );
System.out.println("b--    = " + (a--) );
System.out.println("d++    = " + (d++) );
System.out.println("++d    = " + (++d) );
}
```

```
}
```

Output

```
a + b = 30
a - b = -10
a * b = 200
b / a = 2
b % a = 0
c % a = 5
a++    = 10
b--    = 11
d++    = 25
++d    = 27
```

Relational Operators

There are following relational operators supported by Java language Assume variable A holds 10and variable B holds 20 then:

Operator	Description	Example
==	Checks if the value of two operands are equal or not, if yes then conditionbecomes true.	(A == B) is nottrue.
!=	Checks if the value of two operands are equal or not, if values are not equalthen condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of rightoperand, if yes then condition becomes true.	(A > B) is nottrue.
<	Checks if the value of left operand is less than the value of right operand, ifyes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value ofright operand, if yes then condition becomes true.	(A >= B) is nottrue.
<=	Checks if the value of left operand is less than or equal to the value of rightoperand, if yes then condition becomes true.	(A <= B) is true.

The following simple example program demonstrates the relational operators:

```
public class Test {
public static void main(String args[]) {
int a = 10;
int b = 20;
System.out.println("a == b = " + (a == b) );
System.out.println("a != b = " + (a != b) );
System.out.println("a > b = " + (a > b) );
System.out.println("a < b = " + (a < b) );
System.out.println("b >= a = " + (b >= a) );
System.out.println("b <= a = " + (b <= a) );
}
}
```

output

```
a == b = false
a != b = true
a > b = false
```

```
a < b = true
b >= a = true
b <= a = false
```

Bitwise Operators

Java defines several bitwise operators which can be applied to the integer types, long, int, short,char, and byte.

Bitwise operator works on bits and perform bit by bit operation. Assume if a = 60; and b = 13;Now in binary format they will be as follows:

a = 0011 1100
b = 0000 1101
---
a&b = 0000 1100
a|b = 0011 1101
a^b = 0011 0001
~a = 1100 0011

The following table lists the bitwise operators:

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in eatheroperand.	(A | B) will give 61 which is 0011 1101
!	Called Logical NOT Operator. Use to reverses the logical state of its operand.If a condition is true then Logical NOT operator will make false.	!(A && B) is true.
^	Binary XOR Operator copies the bit if it is set in oneoperand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has theefect of 'flipping' bits.	(~A) will give -60 which is 11000011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the rightoperand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is1111
>>>	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

Assume integer variable A holds 60 and variable B holds 13 then:
The following simple example program demonstrates the bitwise operators:

```
public class Test {

  public static void main(String args[]) {
     int a = 60; /* 60 = 0011 1100 */
     int b = 13; /* 13 = 0000 1101 */
     int c = 0;

     c = a & b;        /* 12 = 0000 1100 */
     System.out.println("a & b = " + c );

     c = a | b;        /* 61 = 0011 1101 */
     System.out.println("a | b = " + c );

     c = a ^ b;        /* 49 = 0011 0001 */
     System.out.println("a ^ b = " + c );

     c = ~a;           /*-61 = 1100 0011 */
     System.out.println("~a = " + c );

     c = a << 2;     /* 240 = 1111 0000 */
     System.out.println("a << 2 = " + c );

     c = a >> 2;     /* 215 = 1111 */
     System.out.println("a >> 2  = " + c );

     c = a >>> 2;     /* 215 = 0000 1111 */
     System.out.println("a >>> 2 = " + c );
  }
}
```

This would produce following result:

```
a & b = 12
a | b = 61
a ^ b = 49
~a = -61
a << 2 = 240
a >> 15
a >>> 15
```

1.2.1 Logical Operators

The following table lists the logical operators: Assume Boolean variables A holds true and variableB holds false then:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero then thencondition becomes true. (A && B) is false.	(A && B) is false
||	Called Logical AND operator. If both the operands are non-zero then thencondition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand.If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

The following simple example program demonstrates the logical operators. Copy and pastefollowing Java program in Test.java file and compile and run this program:

```
public class Test {
public static void main(String args[]) {
boolean a = true;
```

```
boolean b = false;
System.out.println("a && b = " + (a&&b));
System.out.println("a || b = " + (a||b) );
System.out.println("!(a && b) = " + !(a && b));
}
}
```

Output

```
a && b = false a || b = true
!(a && b) = true
```

1.2.2 Assignment Operators

There are following assignment operators supported by Java language

Operator	Description	Example
=	Simple assignment operator, Assigns values from right sideoperands to left side operand	C = A + B will assign value ofA + B into C
+ =	Add AND assignment operator, It adds right operand to the leftoperand and assign the result to left operand	C += A is equivalent to C = C + A
- =	Subtract AND assignment operator, It subtracts right operandfrom the left operand and assign the result to left operand	C -= A is equivalent to C = C -A
* =	Multiply AND assignment operator, It multiplies right operandwith the left operand and assign the result to left operand	C *= A is equivalent to C = C *A
/ =	Divide AND assignment operator, It divides left operand withthe right operand and assign the result to left operand	C /= A is equivalent to C = C / A
% =	Modulus AND assignment operator, It takes modulus using twooperands and assign the result to left operand	C %= A is equivalent to C = C % A
<< =	Left shift AND assignment operator	C <<= 2 is same as C = C <<2
>> =	Right shift AND assignment operator	C >>= 2 is same as C = C >>2
& =	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^ =	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

The following simple example program demonstrates the assignment operators:

```
public class Test {
public static void main(String args[]) {
int a = 10; int b = 20; int c = 0;
c = a + b;
System.out.println("c = a + b = " + c );
c += a ;
System.out.println("c += a = " + c );
c -= a ;
System.out.println("c -= a = " + c );
c *= a ;
```

```
System.out.println("c *= a = " + c );
a = 10;
c = 15;
c /= a ;
System.out.println("c /= a = " + c );
a = 10;
c = 15;
c %= a ;
System.out.println("c %= a = " + c );
c <<= 2 ;
System.out.println("c <<= 2 = " + c );
c >>= 2 ;
System.out.println("c >>= 2 = " + c );
c >>= 2 ;
System.out.println("c >>= a = " + c );
c &= a ;
System.out.println("c &= 2 = " + c );
c ^= a ;
System.out.println("c ^= a = " + c );
c |= a ;
System.out.println("c |= a = " + c );
}
}
```

output

```
c = a + b = 30 c += a = 40
c -= a = 30
c *= a = 300 c /= a = 1 c %= a = 5 c <<= 2 = 20
c >>= 2 = 5
c >>= 2 = 1
c &= a = 0 c ^= a = 10
c |= a = 10
```

Miscellaneous Operators

There are few other operators supported by Java Language.Conditional Operator (? :):

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide which value should be assigned to the variable. The operator is written as:

Syntax:

variable x = (expression) ? value if true : value if false

Example:

```
public class Test {
public static void main(String args[]){
int a , b;
a = 10;
b = (a == 1) ? 20: 30;
System.out.println( "Value of b is : " + b );
b = (a == 10) ? 20: 30;
System.out.println( "Value of b is : " + b );
}
}
Value of b is : 30
Value of b is : 20
```

instanceOf Operator:

This operator is used only for object reference variables. The operator checks whether the objectis of a particular type(class type or interface type). instanceOf operator is wriiten as:

```
( Object reference variable ) instanceOf (class/interface type)
```

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side then the result will be true.

Following is the example:

```
String name = = 'James';
boolean result = name instanceOf String;
// This will return true since name is type of String
class Vehicle {}
```

This operator will still return true if the object being compared is the assignment compatible withthe type on the right. Following is one more example:

```
public class Car extends Vehicle {
public static void main(String args[]){
Vehicle a = new Car();
boolean result = a instanceof Car;
System.out.println( result);
}
}
```

output

```
true
```

Precedence of Java Operators

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others;
for example, the multiplication operator has higher precedence than the addition operator:

x = 7 + 3 * 2;

Here x is assigned 13, not 20 because operator * has higher precedence than + so it first getmultiplied with 3*2 and then adds into 7.

Here operators with the highest precedence appear at the top of the table, those with the lowestappear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	**Operator**	**Associative**
Postfix	( ) [ ]	Left to right
Unary	++ - - ! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>> >>> <<	Left to right

Relational	> >= < <=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Decision Making & Branching

if Statement

An if statement consists of a Boolean expression followed by one or more statements.**Syntax**:
The syntax of if statement is:

```
if(Condition)
{
  //Statements will execute if the Boolean expression is true
}
```

If the boolean expression evaluates to true then the block of code inside the if statement will be executed. If not the first set of code after the end of the if statement (after the closing curly brace)will be executed.

Example:

```
public class Test {
public static void main(String args[]){
int x = 10;
   if( x < 20 )
   {
        System.out.print("This is if statement");
   }
}
}
```

output

```
This is if statement
```

if…else Statement

An if statement can be followed by an optional *else* statement, which executes when the Condition is false.

Syntax:

The syntax of a if...else is:

```
if(Boolean_expression){
//Executes when the Boolean expression is true
}else{
//Executes when the Boolean expression is false
}
```

Example:

```
public class Test {
public static void main(String args[]){
int x = 30;
  if( x < 20 )
  {
       System.out.print("This is if statement");
  }
  else
  {
       System.out.print("This is else statement");
  }
}
}
```

output

```
This is if else statement
```

else…if Ladder

if statement can be followed by an optional *else if...else* statement, which is very use full to testvarious conditions using single if...else if statement.

When using if , else if , else statements there are few points to keep in mind.

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of he remaining else if's or else's will be tested.

Syntax:

```
if(condition1){
//Executes when the condition1 is true
}else if(condition2){
//Executes when the condition2 is true
}else if(condition3){
//Executes when the condition3 is true
}else {
//Executes when the none of the above condition is true.
}
```

Example:

```
public class Test {
public static void main(String args[]){
int x = 30;
  if( x == 10 )
  {
       System.out.print("Value of X is 10");
  }
  else if( x == 20 )
  {
```

```
        System.out.print("Value of X is 20");
   }
   else if( x == 30 )
   {
        System.out.print("Value of X is 30");
   }
   Else
   {
        System.out.print("This is else statement");
   }
}
}
```

output

```
Value of X is 30
```

Nested if…else Statement

It is always legal to nest if-else statements, which means you can use one if or else if statementinside another if or else if statement.

Syntax:

```
if(Condition1){
//Executes when the Boolean expression 1 is true
if(Condition2){
//Executes when the Boolean expression 2 is true
}
}
```

***You can nest else if...else in the similar way as we have nested if statement*.Example:**

```
public class Test {
public static void main(String args[]){
int x = 30;
int y = 10;
if( x == 30 )
{
   if( y == 10 )
   {
        System.out.print("X = 30 and Y = 10");
   }
}
}
}
```

output

```
X = 30 and Y = 10
```

switch Statement

A *switch* statement allows a variable to be tested for equality against a list of values. Each value iscalled a case, and the variable being switched on is checked for each case.

Syntax:

```
switch(expression){
case value :
```

```
  //Statements break; //optional
case value :
  //Statements break; //optional
  //You can have any number of case statements.
default : //Optional
  //Statements
}
```

The following rules must be considered while using switch statement:

1. The variable used in a switch statement can only be a byte, short, int, or char.
2. You can have any number of case statements within a switch. Each case is followed by thevalue to be compared to and a colon.
3. The value for a case must be the same data type as the variable in the switch, and it mustbe a constant or a literal.
4. When the variable being switched on is equal to a case, the statements following that casewill execute until a *break* statement is reached.
5. When a *break* statement is reached, the switch terminates, and the flow of control jumpsto the next line following the switch statement.
6. Not every case needs to contain a break. If no break appears, the flow of control will *fallthrough* to subsequent cases until a break is reached. A *switch* statement can have an optional default case, which must appear at the end of theswitch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Example:

```
public class Test {
public static void main(String args[]){
char grade = args[0].charAt(0);
switch(grade)
{
case 'A' : System.out.println("Excellent!"); break;
case 'B' :
System.out.println("Very good!"); Break;
case 'C' :
System.out.println("Well done"); break;
case 'D' :
System.out.println("You passed"); case 'F' :
System.out.println("Better try again"); break;
default :
System.out.println("Invalid grade");
}
System.out.println("Your grade is " + grade);
}
```

Compile and run above program using various command line arguments. This would producefollowing result:

```
$ java Test a Invalid grade Your grade is a
$ java Test A Excellent!
Your grade is a A
$ java Test C Well done
Your grade is a C
$
```

Decision Making & Looping

There may be a situation when we need to execute a block of code several number of times, andis often referred to as a loop.
Java has very flexible three looping mechanisms. You can use one of the following three loops:

1. while Loop
2. do...while Loop
3. for Loop
   As of java 5 the *enhanced for loop* was introduced. This is mainly used for Arrays.

while Loop

A while loop is a control structure that allows you to repeat a task a certain number of times.

Syntax:

```
while(Condition)
{
   //Statements
   Increment / decrement
}
```

When executing, if the *Condition* result is true then the actions inside the loop will be executed. This will continue as long as the expression result is true.
Here key point of the *while* loop is that the loop might not ever run. When the expression is testedand the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example:

```
public class Test {
public static void main(String args[]) {
int x = 10;
while( x < 20 )
  {
        System.out.print("value of x : " + x );
        x++;
        System.out.print("\n");
  }
}
}
```

Output

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

do while Loop

A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to executeat least

one time.

Syntax:

```
do
{
//Statements
Increment/decrement
}while(Condtion);
```

The Condition appears at the end of the loop, so the statements in the loop execute once before the condition is tested.

If the Boolean expression is true, the flow of control jumps back up to do, and the statements in the loop execute again. This process repeats until the Condition is false.

Example:

```
public class Test {
public static void main(String args[]){
int x = 10;
do
{
  System.out.print("value of x : " + x );
  x++;
  System.out.print("\n");
}while( x < 20 );
}
}
```

output

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

for Loop

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

A for loop is useful when you know how many times a task is to be repeated.

Syntax:

```
for(initialization; Boolean_expression; update)
{
//Statements
}
```

Flow of control in a for loop:

7. The initialization step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
8. Next, the Boolean expression is evaluated. If it is true, the body of the loop is executed. If it is

false, the body of the loop does not execute and flow of control jumps to the next statement past the for loop.

9. After the body of the for loop executes, the flow of control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the Boolean expression.
10. The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then update step,then Boolean expression). After the Boolean expression is false, the for loop terminates.

Example:

```
public class Test {
public static void main(String args[]) {
for(int x = 10; x < 20; x = x+1)
  {
        System.out.print("value of x : " + x );
        System.out.print("\n");
  }
}
}
```

output

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

Enhanced For Loop in Java

As of java 5 the enhanced for loop was introduced. This is mainly used for Arrays.

Syntax:

The syntax of enhanced for loop is:

```
for(declaration : expression)
{
//Statements
}
```

- Declaration . The newly declared block variable, which is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.
- Expression . This evaluate to the array you need to loop through. The expression can be an array variable or method call that returns an array.

Example

```
public class Test {
public static void main(String args[]){
int [] numbers = {10, 20, 30, 40, 50};
  for(int x : numbers )
```

```
  {
       System.out.print( x );
     System.out.print(",");
  }
System.out.print("\n");
String [] names ={"James", "Larry", "Tom", "Lacy"};
  for( String name : names )
  {
       System.out.print( name );
       System.out.print(",");
  }
}
}
```

output

```
10,20,30,40,50,
James,Larry,Tom,Lacy,
```

break Keyword

The *break* keyword is used to stop the entire loop. The break keyword must be used inside anyloop or a switch statement.

The break keyword will stop the execution of the innermost loop and start executing the next lineof code after the block.

Syntax:

```
Break;
```

Example:

```
public class Test {
public static void main(String args[]) {
int [] numbers = {10, 20, 30, 40, 50};
  for(int x : numbers )
  {
       if( x == 30 )
       {
            break;
       }
  System.out.print( x );
  System.out.print("\n");
  }
}
}
```

Output

```
10
20
```

continue Keyword

The *continue* keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.

- In for loop, the continue keyword causes flow of control to immediately jump to the update statement.
- In a while loop or do/while loop, flow of control immediately jumps to the Boolean expression.

Syntax:

```
continue;
```

The syntax of continue is a single statement inside any loop:

Example

```
public class Test {
public static void main(String args[]) {
int [] numbers = {10, 20, 30, 40, 50};
  for(int x : numbers )
  {
       if( x == 30 )
      {
        continue;
      }
  System.out.print( x );
  System.out.print("\n");
  }
 }
 }
```

Output

```
10
20
40
50
```