COMPUTER ENGINEERING

THIRD YEAR (5TH SEM)

PROGRAMMING IN JAVA

CONTENTS:

1.Introduction to Java:

The Basics of Java-A brief history of Java, The Java, Architecture, Java Features. Importance of Java to the Internet Java Applets and Applications, Fundamentals of Object-Oriented Programming, Concepts of OOP, Benefits of OOP, Java and C++, Java Environment, Java Development Kit, Application programming Interface (API). Getting started with JDK, Java program structure, Using Java with Other Tools.

2.Language Basics:

Java tokens, Java character set, Keywords, Identifiers, Literals, Separators. Constant Variables. Data types. Type casting Constants, Variables and their Scope, Operator and Expressions, Arithmetic Operators, Relational & Conditional Operators, Logical Operators: Assignment Operators, Increment & Decrement Bitwise operator, special Operators, Precedence of Operators, Control Flow statements If & If else statements, switch Statement, for loop, while do loop, Branching.

3. Objects and Classes in Java:

Introduction to classes, Defining a class. Creating objects, Methods, Constructors and Access Specifiers Application of Constructor. Parameterized Constructors, Overloading Methods and Constructors, Access control Modifiers Public, Private and Protected. Static. Final and Abstract Modifiers and Method overriding, Inheritance basics, Method overriding

4. Arrays Strings and Vectors:

Arrays one-dimensional array Multidimensional array, Strings, String class, Working with Strings, String Buffer class, Vector and wrapper class, Vector Constructors, Working with vector methods, wrapper Class.

5. Packages and Interfaces:

Using Java interfaces, defining an interface, implementing an interface, Extending an Interface, Using Java Packages, defining a Package, Brief discussion on CLASSPATH, Access Protection, Importing a package, Java API Package,

6. Exception handling:

Introduction to Exception Handling, why use Exception Handling, Fundamentals of Exception Handling. Exceptions & their types, Common Exceptions, Using Exception Handling using try and catch, Multiple Catch Statement, Nested try Statements, Methods available to Exceptions, Throwing your own Exception.

7. Applet programming:

Writing Applets, The Basics of Applets, Life Cycle of an Applet, Painting the Applet, The Applet Tag, Security Restrictions when using Applets. Taking, Advantage of the Applet API, Finding and Loading data Files, displaying short Status Strings, Displaying Documents in the Browser Playing Sounds Defining and Using Applet Parameters.

8. Working with Graphics:

The Graphic class, Java. Awt graphics, use of class, java. Awt. Graphics, Custom painting, Drawing Lines, Drawing Rectangles Drawing ellipses and circles, Drawing Arcs, Drawing Polygons, Practice: Ex Based on above concepts.

UNIT- 1 INTRODUCTION TO JAVA:

- Java is a programming language & a platform. It is a high level, robust, object oriented & secure programming language.
- It is developed by **James Gosling** at sun microsystems in the year 1995.
- James Gosling is known as father of java.
- In 1995, sun microsystems changed the name of java because its name was oak before.
- In 2009, sun microsystems takeover by oracle corporation.
- The principle for creating java programming were "simple, robust, portable, platform independent, secured, high performance, multi-threaded, architecture neutral, object oriented, interpreted & dynamic.

JDK 1.0 released in (jan23, 1996). After the first release of java, there have been many additional features added to the language. Now, Java is being used in windows application, web application, enterprise application, mobile application, cards, etc.

Why uses java?

- Java works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.)
- It is one of the most popular programming languages in the world.
- It is easy to learn and simple to use.
- It is open-source and free.
- It is secure, fast and powerful.
- It has a huge community support
- Java is an object – oriented language which gives a clear structure to programs and allow code to be reused, lowering development costs.

EDITIONS OF JAVA: -

- **Java standard editions (JSE): -**
  -It is used to create programs for a desktop computer.
- **Java enterprise edition (JEE): -**
  -It is used to create large programs that run on the server & manages heavy traffic & complex transactions.
- **Java micro edition (JME): -**
  -It is used to develop application for small devices such as setup boxes, phones etc.

FEATURES OF JAVA: -

- **Simple: -**

Java is a simple language because its syntax is simple, clean and easy to understand. Complex & ambiguous concepts of c++ are either eliminated or re - reimplemented in java. There is no need to remove unreferenced objects because there is an automatic garbage collection in java.

- **Object-oriented: -**

In java, everything is in form of the object. It means it has some data and behaviour. A program must have at least one class & object.

- **Robust: -**

Java makes an effort to check error at run time & compile time. It uses a strong memory management system called garbage collector. Exception handling & garbage collection features make it strong.

- **Secure: -**

It is a secure programming language because it has no explicit pointer and programs runs in the virtual machine. Java contains a security manager that defines the access of java classes.

- **Platform independent: -**

Java provides a guarantee that code writes once & run anywhere. This byte code is platform independent & can be any on any machine.

JAVA PROGRAM--->COMPILE-->BYTE CODE -->LINUX or WINDOWS or MAC.

- **Architectural neutral: -**

Compiler generates bytecodes, which have nothing to do with a particular computer architecture, hence a java program is easy to interpret on any machine. Java provides a software-based programs: - It has 2 components -->

RUNTIME ENVIRONMENT OR APPLN PROGM INTERFACE(API)

- **Portable: -**

Java bytecode can be carried to any platforms. No implementations dependent features. Everything related to storage is predefined. For e.g.: the size of primitive data types.

- **Dynamic: -**

It means classes are loaded on demand & also supports functions from its native language. It supports dynamic complication & automatic memory management (garbage collection).

- **High performance: -**

Java is an interpreted language. Java enables high performance with the use of the **JUST IN TIME** compiler.

- **Distributed: -**

Java also has networking facilities. It is designed for the distributed environment of the internet because it supports TCP\IP protocol. It can run over the internet. EJB and RMI are used to create a distributed system.

- **Multi - threaded: -**

Java also supports multi-threading. It means to handle more than one job a time. Main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area.

JAVA ARCHITECTURE: -

Java architecture is a collection of components, i.e., JVM, JRE & JDK.

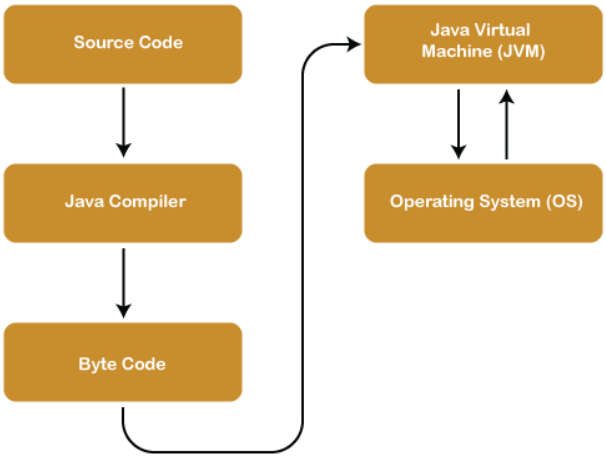
It integrates the process of interpretation & compilation.

It defines all the processes involved in creating a java program.

Java architecture each & every step of how a program is compiled & executed.

JAVA ARCHITECTURE CAN BE EXPLAINED BY USING THE FOLLOWING STEPS:

- There is a process of compilation & interpretation in java.
- Java compiler converts the java code into byte code.
- After that, the JVM converts the byte code into machine code.
- The machine code is executed by the machine.



COMPONENTS OF JAVA ARCHITECTURE: -

There are three main components: -

1.JAVA VIRTUAL MACHINE(JVM)

2.JAVA RUNTIME ENVIRONMENT(JRE)

3.JAVA DEVELOPMENT KIT(JDK)

- **JAVA VIRTUAL ENVIRONMENT: -**

The main features of java are WORA. WORA stands for WRITE ONCE RUN ANYWHERE. It provides runtime environment to execute java byte code. The JVM doesn't understand java code directly. That's why you need to compile your *.java files to obtain *.class files that contain the bytecodes understandable by the JVM. Main task is to convert byte code into machine code.

JVM performs the following functions:

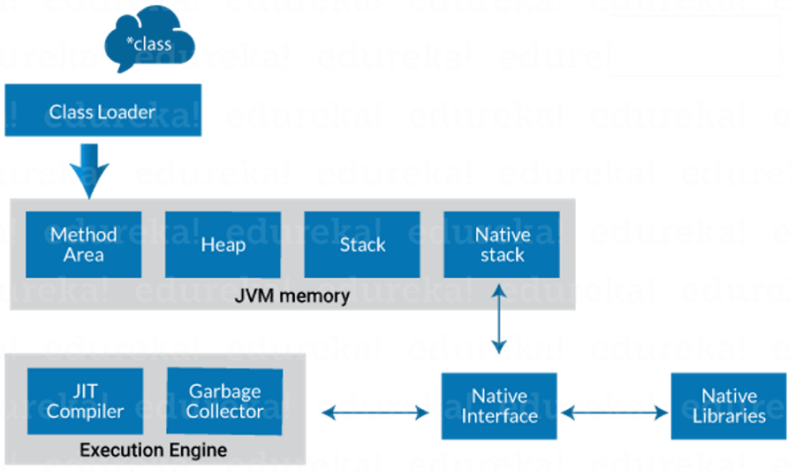
- Loads the code
- Verifies the code
- Executes the code
- Provides runtime environment

Now, let me show you the JVM architecture. Here goes!

Explanation:

Class Loader: Class loader is a subsystem of JVM. It is used to load class files. Whenever we run the java program, class loader loads it first.

Class method area: It is one of the Data Area in JVM, in which Class data will be stored. Static Variables, Static Blocks, Static Methods, Instance Methods are stored in this area.



Heap: A heap is created when the JVM starts up. It may increase or decrease in size while the application runs.

Stack: JVM stack is known as a thread stack. It is a data area in the JVM memory which is created for a single execution thread. The JVM stack of a thread is used by the thread to store various elements i.e.; local variables, partial results, and data for calling method and returns.
Native stack: It subsumes all the native methods used in your application.

Execution Engine:

- JIT compiler
- Garbage collector

JIT compiler: The Just-In-Time (JIT) compiler is a part of the runtime environment. It helps in improving the performance of Java applications by compiling bytecodes to machine code at run time. The JIT compiler is enabled by default. When a method is compiled, the JVM calls the compiled code of that method directly. The JIT compiler compiles the bytecode of that method into machine code, compiling it "just in time" to run.

Garbage collector: As the name explains that Garbage Collector means to collect the unused material. Well, in JVM this work is done by Garbage collection. It tracks each and every object available in the JVM heap space and removes unwanted ones.
Garbage collector works in two simple steps known as Mark and Sweep:

- Mark – it is where the garbage collector identifies which piece of memory is in use and which are not
- Sweep – it removes objects identified during the "mark" phase.

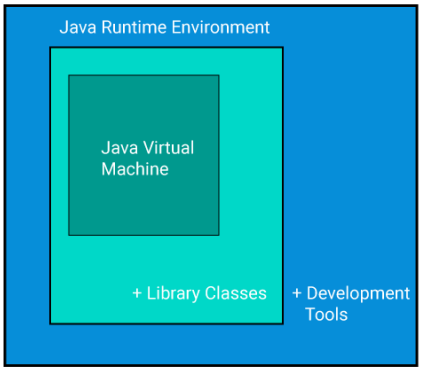
- **JAVA RUNTIME ENVIRONMENT: -**

The JRE software builds a runtime environment in which Java programs can be executed. The JRE is the on – disk system that takes your Java code, combines it with the needed libraries, and starts the JVM to execute it. The JRE contains libraries and software needed by your Java programs to run. JRE is a part of JDK but can be downloaded separately.

JVM-->libraries + other components-->JRE

- **JAVA DEVELOPMENT KIT: -**

The Java Development Kit (JDK) is a software development environment used to develop Java applications and applets. It contains JRE and several development tools, an interpreter/loader(java), a compiler (javac), an archiver(jar), a documentation generator (Javadoc) accompanied with another tool.



IMPORTANCE OF JAVA TO THE INTERNET:

Java has had a profound effect on the Internet because it allows objects to move freely in Cyberspace. In a network there are two categories of objects that are transmitted between the Server and the Personal computer.

- Passive information
- Dynamic active programs

The Dynamic Self-executing programs cause serious problems in the areas of Security and probability. But Java addresses those concerns and by doing so has opened the door to an exciting new form of program called the Applet.

JAVA APPLETS AND APPLICATIONS:

Java Application:
Java Application is just like a Java program that runs on an underlying operating system with the support of a virtual machine. It is also known as an **application program**. The graphical user interface is not necessary to execute the java applications, it can be run with or without it.
Java Applet:
An applet is a Java program that can be embedded into a web page. It runs inside the web browser and works at client side. An applet is embedded in an HTML page using the **APPLET** or **OBJECT** tag and hosted on a web server. Applets are used to make the web site more dynamic and entertaining.

FUNDAMENTALS OF OOPS:

The four basics of OOP are abstraction, encapsulation, inheritance, and polymorphism. These are the main ideas behind Java's Object-Oriented Programming.

CONCEPTS OF OOPS:

Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

- Object
- Class
- Encapsulation
- Data Abstraction
- Inheritance
- Polymorphism

- **OBJECT:**
  It is a basic unit of Object-Oriented Programming and represents the real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of:

1. **State:** It is represented by attributes of an object. It also reflects the properties of an object.
2. **Behaviour:** It is represented by methods of an object. It also reflects the response of an object with other objects.
3. **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

Example: A dog is an object because it has states like colour, name, breed, etc. as well as behaviours like wagging the tail, barking, eating, etc.

- **CLASS:**

Collection of objects is called class. It is a logical entity.A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type.

- **ENCAPSULATION:**

Binding (or wrapping) code and data together into a single unit are known as encapsulation. It is a protective shield that prevents the data from being accessed by the code outside this shield. in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared.

As in encapsulation, the data in a class is hidden from other classes, so it is also known as data-hiding.

Encapsulation can be achieved by Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.

- **DATA ABSTRACTION:**

Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or the non-essentials units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components, phone call, we don't know the internal processing. In java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.

- **INHERITENCE:**

Inheritance is an important pillar of Object-Oriented Programming. It is the mechanism in java by which one class (child class) is allowed to inherit the features (fields and methods) of another class (parent class). It provides code reusability. It is used to achieve runtime polymorphism.

- **POLYMORPHISM:**

The word “poly” means many and “morphs” means forms, so it means many forms. If one task is performed in different ways, it is known as polymorphism. In Java, we use method overloading and method overriding to achieve polymorphism. For example: speak something - a cat speaks meow, dog barks woof, etc.

- **METHOD:**

A method is a collection of statements that performs some specific task & return result to the caller. A Method can perform some specific tasks without returning anything. Methods are time savers and helps us to reuse the code without retyping the code.

- **MESSAGE PASSING:**

Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving objects that generates the derived results. Message passing involves specifying the name of the objects, name of the function and the information to be sent.

BENEFITS OF OOPS:

- We can build the programs from standard working modules that communicate with one another, rather than having to start writing the code from scratch which leads to saving of development time and higher productivity.
- OOP language allows to break the program into the bit-sized problems that can be solved easily.
- The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost.
- OOP systems can be easily upgraded from small to large systems.
- It is possible that multiple instances of objects co-exist without any interference.
- It is very easy to partition the work in a project based on objects.
- It is possible to map the objects in problem domain to those in the program.
- The principle of data hiding helps the programmer to build secure programs which cannot be invaded by the code in other parts of the program.
- By using inheritance, we can eliminate redundant code and extend the use of existing classes.
- Message passing techniques is used for communication between objects which makes the interface descriptions with external systems much simpler.
- The data-centered design approach enables us to capture more details of model in an implementable form.

JAVA AND C++:

Major differences between C++ and Java.

Java	C++
Java does not support pointers, unions, operator overloading and structure.	C++ supports pointers, unions, operator overloading and structure.
Java supports garbage collection.	C++ does not support garbage collection.
Java is platform independent.	C++ is platform dependent.
Java supports inheritance except for multiple inheritance	C++ supports inheritance including multiple inheritances

Java is interpreted.	C++ is compiled.
Java does not support destructor	C++ supports destructors.

JAVA ENVIRONMENT:

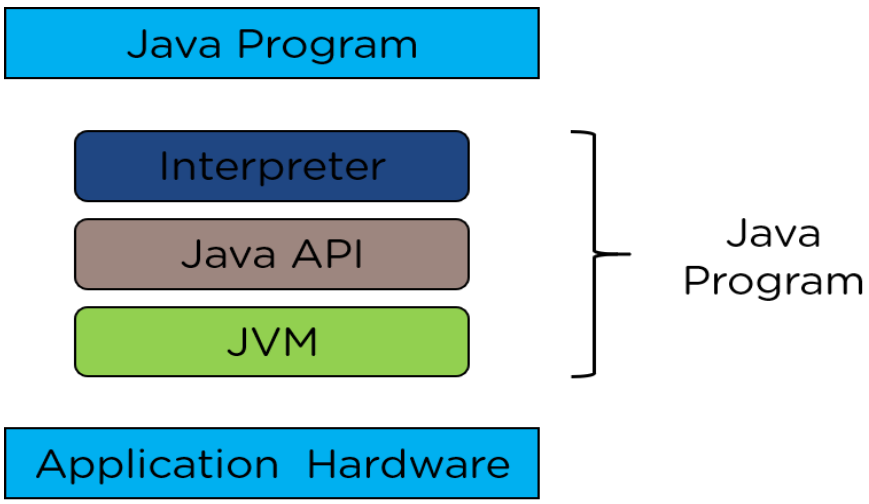
- Java runtime environment (JRE) is the part of the Java development kit (JDK).
- It is a software layer that runs on top of a computer's operating system software and provides the class libraries and other resources that a specific Java program needs to run.
- It is most common environment available on devices to run java programs.
- The source Java code gets compiled and converted to java bytecode.
- If you wish to run this byte code on any platform, you require JRE. The JRE loads classes, verify access to memory, & retrieves the system resources.

JAVA DEVELOPMENT KIT:

- JDK is an acronym for java development kit.
- The Java Development Kit (JDK) is a software development environment used for developing Java applications and applets.
- It includes tools useful for developing and testing programs written in the Java programming language and running on the Java platform.
- The Java Runtime Environment (JRE), an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc) and other tools needed in Java development.

➢ APPLICATION PROGRAMMING INTERFACE(API):

- API is the acronym for Application Programming Interface.
- APIs are important software components bundled with the JDK.
- APIs in Java include classes, interfaces, and user Interfaces.
- They enable developers to integrate various applications and websites and offer real-time information.



➢ ADVANTAGES OF API IN JAVA:

Automation: With Java APIs, instead of people, computer systems can control the work. Throughout APIs, organizations can upgrade workflows to create them faster and even more effective.

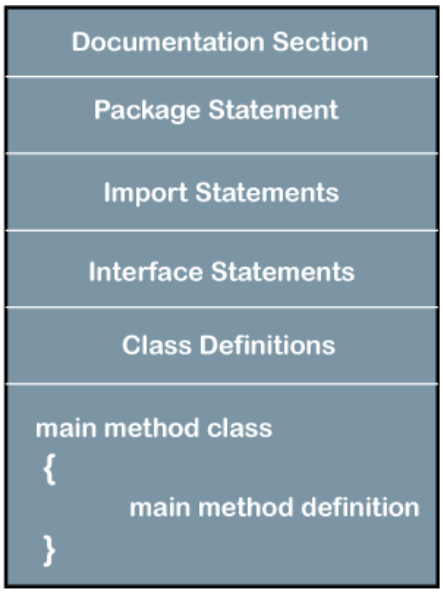
Application: Since Java APIs can easily access the software components, the delivery of services, as well as data, is much more flexible.

Efficiency: Once access is offered for a Java API, the content produced could be released instantly, and it is readily available for every single channel. This enables it to be distributed as well as sent out quickly.

Integration: Java APIs enables content to be embedded by any site or perhaps software easier. This ensures additional fluid data delivery and then a built-in user experience.

JAVA PROGRAM STRUCTURE:

A typical structure of a Java program contains the following elements:



Structure of Java Program

➢ 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 **author's name, date of creation, version, program name, company 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. To write the statements in the documentation section, we use **comments**. The comments may be **single-line, multi-line,** and **documentation** comments.

➢ PACKAGE STATEMENTS:

- The package declaration is optional.
- It is placed just after the documentation section.
- In this section, we declare the **package name** in which the class is placed.
- It must be defined before any class and interface declaration. It is necessary because a Java class can be placed in different packages and directories based on the module they are used.
- For all these classes package belongs to a single parent directory.

➢ IMPORT STATEMENTS:

- The package contains the many predefined classes and interfaces.
- If we want to use any class of a particular package, we need to import that class.
- The import statement represents the class stored in the other package.
- We use the **import** keyword to import the class.
- It is written before the class declaration and after the package statement.
- We use the import statement in two ways, either import a specific class or import all classes of a particular package. In a Java program, we can use multiple import statements.

➢ INTERFACE STATEMENTS:

- It is an optional section.
- We can create an **interface** in this section if required. We use the **interface** keyword to create an interface.
- 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 DEFINATION:

- It is **vital** part of a Java program.
- Without the class, we cannot create any Java program.
- A Java program may conation more than one class definition.
- We use the **class** keyword to define the class.
- It contains information about user-defined methods, variables, and constants. Every Java program has at least one class that contains the main() method.
- **s**

➢ MAIN METHOD CLASS:

- It is essential for all Java programs. Because the execution of all Java programs starts from the main() method.
- In other words, it is an entry point of the class.

- It must be inside the class. Inside the main method, we create objects and call the methods.
- We use the following statement to define the main() method:

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

For example:

```
package com. company;


public class Main {


   public static void main (String [] args) {
      System.out.println("hello world");
   }
}
```

output: - hello world

USING JAVA WITH OTHER TOOLS:

There are multiple Java Tools available that developers use to develop the application. Here, we are going to discuss those tools: -

JDK (java development kit)

ECLIPSE

NETBEANS

INTELLIJ IDEA

SPARK

GRADLE

JAVA DECOMPILER

JRAT

GROOVY

APACHE JMETER

UNIT – 2 LANGUAGE BASICS

JAVA TOKENS:

It is the smallest element of java program.

The java compiler breaks the line of code into text (words) is called Java tokens.

The java compiler identified these words as tokens. These tokens are separated by the delimiters.

It is useful for compilers to detect errors.

Types of tokens: -

- Keywords
- Identifiers
- Literals
- Operators
- Separators
- Comments

JAVA CHARACTER SET:

Characters are the smallest units (elements) of Java language that are used to write Java tokens. These characters are defined by the Unicode character set.

A character set in Java is a set of alphabets, letters, and some special characters that are valid in java programming language.

Java language uses the character sets as the building block to form the basic elements such as identifiers, variables, array, etc in the program. These are as follows:

- Letters: Both lowercase (a, b, c, d, e, etc.) and uppercase (A, B, C, D, E, etc.) letters.
- Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
- Special symbols: _, (,), {, }, [,], +, -, *, /, %, !, &, |, ~, ^, <, =, >, $, #, ?, Comma (,), Dot (.), Colon (:), Semi-colon (;), Single quote (‘), Double quote (“), Back slash (\).
- White space: Space, Tab, New line.

KEYWORDS:

- Java keywords are also known as reserved words.
- Keywords are particular words that act as a key to a code.
- These are predefined words by Java so they cannot be used as a variable or object name or class name.
- It is always written in lower case.
- Java provides the following keywords:

01. abstract	02. Boolean	03. byte	04. break	05. class
06. case	07. catch	08. char	09. continue	10. default
11. do	12. double	13. else	14. extends	15. final
16. finally	17. float	18. for	19. if	20. implements
21. import	22. instanceof	23. int	24. interface	25. long
26. native	27. new	28. package	29. private	30. protected
31. public	32. return	33. short	34. static	35. super
36. switch	37. synchronized	38. this	39. thro	40. throws
41. transient	42. try	43. void	44. volatile	45. while
46. assert	47. const	48. enum	49. goto	50. strictfp

IDENTIFIERS:

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.

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.
- There should be no special symbol expect on underscore.
- The whitespace cannot be included in the identifier.
- Identifiers are case sensitive.
- For example: Java_program , Java124 are valid identifiers & 123java , a-java are an invalid identifiers etc.

LITERALS:

literals are the constant values that appear directly in the program.

It can be assigned directly to a variable.

It can be categorized as an integer literal, string literal, Boolean literal, etc.

Once it has been defined cannot be changed.

Java provides five types of literals are as follows:

- Integer
- Floating Point
- Character
- String
- Boolean

SEPERATORS:

Separators help us defining the structure of a program. In Java, there are few characters used as separators. The most commonly used separator in java is a semicolon (;).

These separators are pictorially depicted below as follows:

Symbol	Name	Purpose
( )	Parentheses	used to contain a list of parameters in method definition and invocation. also used for defining precedence in expressions in control statements and surrounding cast types
{ }	Braces	Used to define a block of code, for classes, methods and local scopes Used to contain the value of automatically initialised array
[ ]	Brackets	declares array types and also used when dereferencing array values
;	Semicolon	Terminates statements
,	Comma	Used to separates package names from sub-package and class names and also selects a field or method from an object
.	Period	separates consecutive identifiers in variable declarations also used to chains statements in the test, expression of a for loop
:	Colon	Used after labels

CONSTANT VARIABLES:

Constant is a value that cannot be changed after assigning it. Java does not directly support the constants. There is an alternative way to define the constants in Java by using the non-access modifiers static and final.

How to declare constant in Java?

In Java, to declare any variable as constant, we use static and final modifiers. It is also known as **non-access** modifiers. According to the Java naming convention the identifier name must be in **capital letters**.

Static & Final Modifiers:

- The purpose to use the static modifier is to manage the memory.
- It also allows the variable to be available without loading any instance of the class in which it is defined.
- The final modifier represents that the value of the variable cannot be changed. It also makes the primitive data type immutable or unchangeable.

The syntax to declare a constant is as follows:

```
static final datatype identifier_name=value;
```

For example, **price** is a variable that we want to make constant.

```
static final double PRICE=432.78;
```

Where static and final are the non-access modifiers. The double is the data type and PRICE is the identifier name in which the value 432.78 is assigned.

Points to Remember:

- Write the identifier name in capital letters that we want to declare as constant. For example, **MAX=12**.
- If we use the **private** access-specifier before the constant name, the value of the constant cannot be changed in that particular class.
- If we use the **public** access-specifier before the constant name, the value of the constant can be changed in the program.

Using Enumeration (Enum) as Constant:

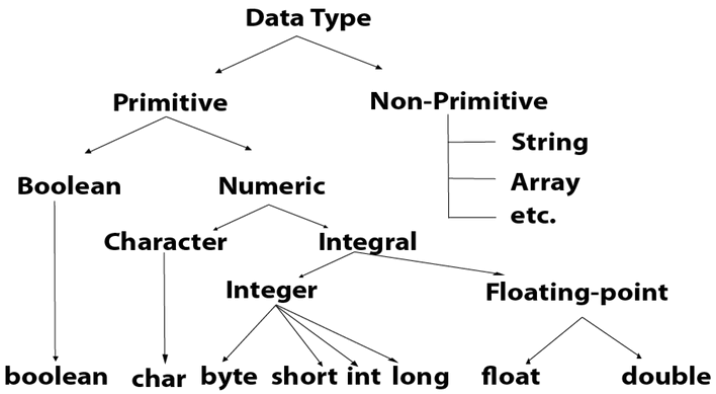
- It is the same as the final variables.
- It is a list of constants.
- Java provides the Enum keyword to define the enumeration.
- It defines a class type by making enumeration in the class that may contain instance variables, methods, and constructors.

DATA TYPES:

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

- **Primitive data types:** The primitive data types include Boolean, char, byte, short, int, long, float and double.
- **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

Java primitive data types:



Data Type	Default Value	Default size
Boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Boolean Data Type: The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions. The Boolean data type specifies one bit of information, but its "size" can't be defined precisely. **Example:** Boolean one = **false**.

Byte Data Type: The byte data type is an example of primitive data type. It isan 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0. The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type. **Example: byte** a = 10, **byte** b = -20.

Short Data Type: The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0. The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer. **Example: short** s = 10000, **short** r = -5000.

Int Data Type: The int data type is a 32-bit signed two's complement integer. Its value-range lies between - 2,147,483,648 (-2^31) to 2,147,483,647 (2^31 -1) (inclusive). Its minimum value is - 2,147,483,648and maximum value is 2,147,483,647. Its default value is 0. The int data type is generally used as a default data type for integral values unless if there is no problem about memory. **Example: int** a = 100000, **int** b = -200000**.**

Long Data Type: The long data type is a 64-bit two's complement integer. Its value-range lies between -9,223,372,036,854,775,808(-2^63) to 9,223,372,036,854,775,807(2^63 - 1)(inclusive). Its minimum value is - 9,223,372,036,854,775,808and maximum value is 9,223,372,036,854,775,807. Its default value is 0. The long data type is used when you need a range of values more than those provided by int. **Example: long** a = 100000L, **long** b = -200000L.

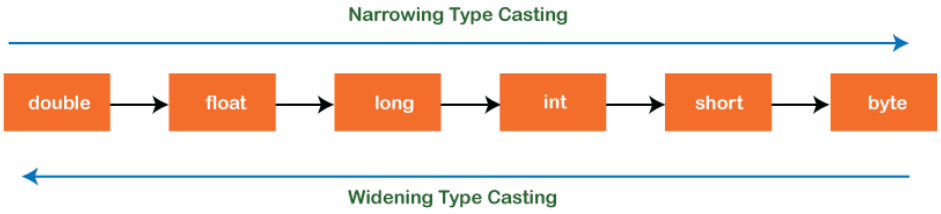
Float Data Type: The float data type is a single-precision 32-bit IEEE 754 floating point.Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating-point numbers. The float data type should never be used for precise values, such as currency. Its default value is 0.0F. **Example: float** f1 = 234.5f.

Double Data Type: The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d. **Example: double** d1 = 12.3.

Char Data Type: The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive). The char data type is used to store characters. Example: char letterA = 'A'.

TYPE CASTING:

Type casting is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer.



Type Casting in Java

Two Types of Type Casting:

Widening Type Casting: Converting a lower data type into a higher one is called widening type casting. It is also known as implicit conversion or casting down**.** It is done automatically. It is safe because there is no chance to lose data. It takes place when:

- Both data types must be compatible with each other.
- The target type must be larger than the source type.

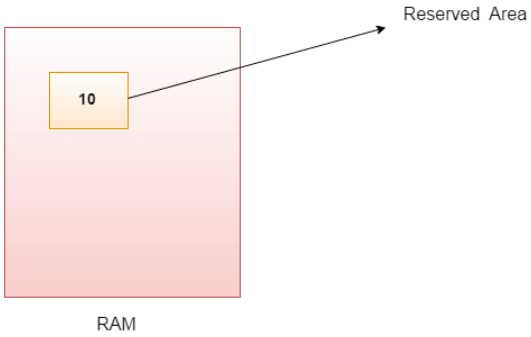
byte -> **short** -> **char** -> **int** -> **long** -> **float** -> **double**

Narrowing Type Casting: Converting a higher data type into a lower one is called **narrowing** type casting. It is also known as **explicit conversion** or **casting up**. It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

double -> **float** -> **long** -> **int** -> **char** -> **short** -> **byte**

VARIABLE:

A variable is the name of a reserved area allocated in memory. In other words, it is a name of the memory location. It is a combination of "vary + able" which means its value can be changed.




```
int data=50;//Here data is variable
```

Types of Variables:

There are three types of variables in Java:

Local variable: A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists. A local variable cannot be defined with "static" keyword.

Instance variable: A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as static. It is called an instance variable because its value is instance-specific and is not shared among instances.

Static variable: A variable that is declared as static is called a static variable. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

SCOPE OF VARIABLES:

Scope of variable defines how a specific variable is accessible within the program or across classes.

There are **three types of variables** based on their scope:

1. **Member Variables (Class Level Scope):** These are the variables that are declared inside the class but outside any function have class-level scope. We can access these variables anywhere inside the class. Note that the access specifier of a member variable does not affect the scope within the class. Java allows us to access member variables outside the class with the following rules:

Access Modifier	Package	Subclass	Word
public	Yes	Yes	Yes
protected	Yes	Yes	No
private	No	No	No
default	Yes	No	No

2. **Local Variables (Method Level Scope):** These are the variables that are declared inside a method, constructor**,** or block have a method-level or block-level scope and cannot be accessed outside in which it is defined. Variables declared inside a pair of curly braces **{}** have block-level scope.

   Note: Local variables don't exist after method's execution is over.

3. **Loop Variables (Block Scope)**: A variable declared inside pair of brackets "{" and "}" in a method has scope within the brackets only.

OPERATOR & EXPRESSIONS:

Operator in Java is a symbol that is used to perform operations.

They are classified based on the functionality they provide. Some of the types are-

Arithmetic Operators: They are used to perform simple arithmetic operations on primitive data types.

*** :** Multiplication
/ : Division
% : Modulo
+ : Addition
– : Subtraction

Unary Operators: Unary operators need only one operand. They are used to increment, decrement or negate a value.

- **– :Unary minus**, used for negating the values.
- **+ :Unary plus**, indicates positive value (numbers are positive without this, however). It performs an automatic conversion to int when the type of its operand is byte, char, or short. This is called unary numeric promotion.

Assignment Operator: '=' Assignment operator is used to assign a value to any variable. It has a right to left associativity, i.e value given on right hand side of operator is assigned to the variable on the left and therefore right-hand side value must be declared before using it or should be a constant.

General format of assignment operator is,

variable = value;
In many cases assignment operator can be combined with other operators to build a shorter version of statement called **Compound Statement**. For example, instead of a = a+5, we can write a += 5.

- +=, for adding left operand with right operand and then assigning it to variable on the left.
- -=, for subtracting left operand with right operand and then assigning it to variable on the left.
- *=, for multiplying left operand with right operand and then assigning it to variable on the left.
- /=, for dividing left operand with right operand and then assigning it to variable on the left.
- %=, for assigning modulo of left operand with right operand and then assigning it to variable on the left.

Relational Operators : These operators are used to check for relations like equality, greater than, less than. They return boolean result after the comparison and are extensively used in looping statements as well as conditional if else statements. General format is,

variable **relation_operator** value
Some of the relational operators are-

- **==, Equal to :** returns true if left hand side is equal to right hand side.
- **!=, Not Equal to :** returns true if left hand side is not equal to right hand side.
- **<, less than :** returns true if left hand side is less than right hand side.
- **<=, less than or equal to :** returns true if left hand side is less than or equal to right hand side.
- **>, Greater than :** returns true if left hand side is greater than right hand side.
- **>=, Greater than or equal to:** returns true if left hand side is greater than or equal to right hand side.

Logical Operators: These operators are used to perform “logical AND” and “logical OR” operation, i.e., the function similar to AND gate and OR gate in digital electronics. One thing to keep in mind is the second condition is not evaluated if the first one is false, i.e., it has a short-circuiting effect. Used extensively to test for several conditions for making a decision.
Conditional operators are-

- **&&, Logical AND:** returns true when both conditions are true.
- **||, Logical OR:** returns true if at least one condition is true.

Ternary operator: Ternary operator is a shorthand version of if-else statement. It has three operands and hence the name ternary. General format is-

condition? if true: if false

Bitwise Operators: These operators are used to perform manipulation of individual bits of a number. They can be used with any of the integer types. They are used when performing update and query operations of Binary indexed tree.

- **&, Bitwise AND operator:** returns bit by bit AND of input values.
- **|, Bitwise OR operator:** returns bit by bit OR of input values.
- **^, Bitwise XOR operator:** returns bit by bit XOR of input values.
- **~, Bitwise Complement Operator:** This is a unary operator which returns the one's complement representation of the input value, i.e. with all bits inversed.

Shift Operators: These operators are used to shift the bits of a number left or right thereby multiplying or dividing the number by two respectively. They can be used when we have to multiply or divide a number by two. General format-

number **shift_op** number_of_places_to_shift;

- **<<, Left shift operator:** shifts the bits of the number to the left and fills 0 on voids left as a result. Similar effect as of multiplying the number with some power of two.
- **>>, Signed Right shift operator:** shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit depends on the sign of initial number. Similar effect as of dividing the number with some power of two.
- **>>>, Unsigned Right shift operator:** shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit is set to 0.

EXPRESSION:

We can convert an expression into a statement by terminating the expression with a `;`. These are known as expression statements. For example,

```
// expression
number = 10
// statement
number = 10;
```

In the above example, we have an expression `number = 10`. Here, by adding a semicolon (`;`), we have converted the expression into a statement (`number = 10;`).

INSTANCE OF OPERATOR:

Instance of operator is used for type checking. It can be used to test if an object is an instance of a class, a subclass or an interface. General format-

```
object instance of class/subclass/interface
```

PRECEDENCE & ASSOCIATIVITY OF OPERATORS:

Precedence and associative rules are used when dealing with hybrid equations involving more than one type of operator. In such cases, these rules determine which part of the equation to consider first as there can be many different valuations for the same equation. The below table depicts the precedence of operators in decreasing order as magnitude with the top representing the highest precedence and bottom shows the lowest precedence.

Operators	Associativity	Type
++ --	Right to left	Unary postfix
++ -- + - ! (type)	Right to left	Unary prefix
/ * %	Left to right	Multiplicative
+ -	Left to right	Additive
< <= > >=	Left to right	Relational
== !==	Left to right	Equality
&	Left to right	Boolean Logical AND
^	Left to right	Boolean Logical Exclusive OR
|	Left to right	Boolean Logical Inclusive OR
&&	Left to right	Conditional AND
||	Left to right	Conditional OR
?:	Right to left	Conditional
= += -= *= /= %=	Right to left	Assignment

CONTROL FLOW STATEMENTS:

Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides **three types of control flow statements:**

1. Decision Making statements
    - if statements
    - switch statement
2. Loop statements
    - do while loop
    - while loop
    - for loop
    - for-each loop
3. Jump statements
    - break statement
    - continue statement

DECISION- MAKING STATEMENTS:

if statement is used to test the condition. It checks Boolean condition: *true* or *false*. There are various types of if statement in Java.

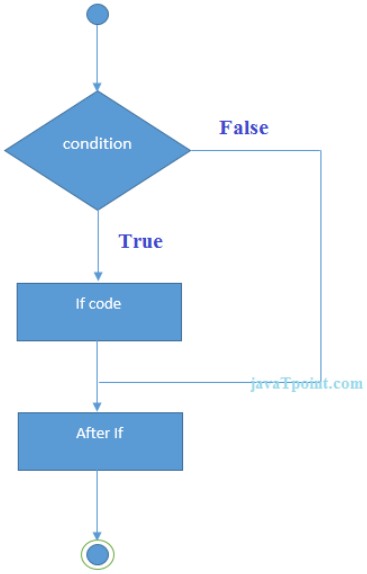
- if statement

- if-else statement
- if-else-if ladder
- nested if statement

Java if statement: The Java if statement tests the condition. It executes the *if block* if condition is true.

Syntax:

```
if(condition){
//code to be executed
}
```



Example:

```
//Java Program to demonstate the use of if statement.
public class IfExample {
public static void main(String[] args) {
int age=20;
if(age>18){
System.out.print("Age is greater than 18");
}
}
}
```

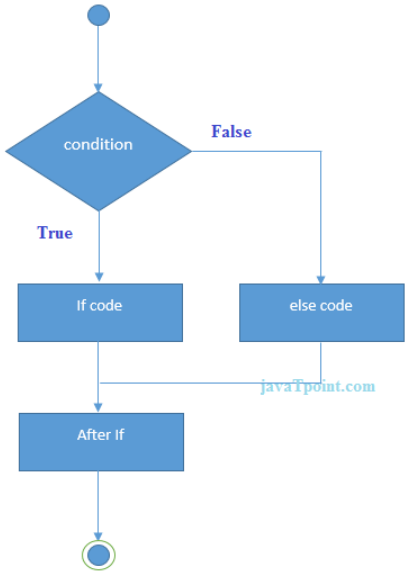
Output:

```
Age is greater than 18
```

Java if-else statement: The Java if-else statement also tests the condition. It executes the *if block* if condition is true otherwise *else block* is executed.

Syntax:

```
if(condition){
//code if condition is true
}else{
//code if condition is false
}
```



Example:

```
//A Java Program to demonstrate the use of if-else statement.
//It is a program of odd and even number.
public class IfElseExample {
public static void main(String[] args) {
int number=13;
if(number%2==0){
System.out.println("even number");
}else{
System.out.println("odd number");
}
}
}
```

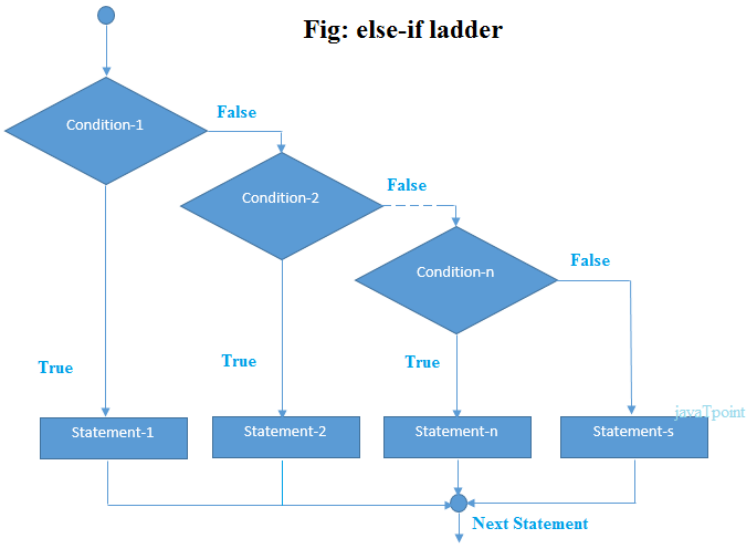
Output:

```
odd number
odd number
```

Java if – else – if ladder statement: The if-else-if ladder statement executes one condition from multiple statements.

Syntax:

```
if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}
...
else{
//code to be executed if all the conditions are false
}
```



Example:

Program to check POSITIVE, NEGATIVE or ZERO:

```
public class PositiveNegativeExample {
public static void main(String[] args) {
 int number=-13;
  if(number>0){
 System.out.println("POSITIVE");
  }else if(number<0){
  System.out.println("NEGATIVE");
   }else{
        System.out.println("ZERO");
  }
}
}
```

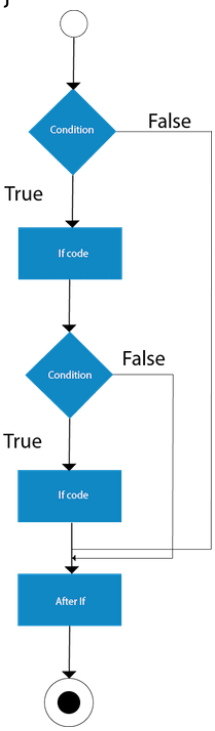
Output:

```
NEGATIVE
```

Java nested if statement: The nested if statement represents the *if block within another if block*. Here, the inner if block condition executes only when outer if block condition is true.

Syntax:

```
if(condition){
//code to be executed
   if(condition){
  //code to be executed
   }
}
```



Example:

```
//Java Program to demonstrate the use of Nested If Statement.
public class JavaNestedIfExample {
public static void main(String[] args) {
 //Creating two variables for age and weight
 int age=20;
int weight=80;
//applying condition on age and weight
 if(age>=18){
    if(weight>50){
 System.out.println("You are eligible to donate blood");
  }
```

```
  }
}}
```

Output:

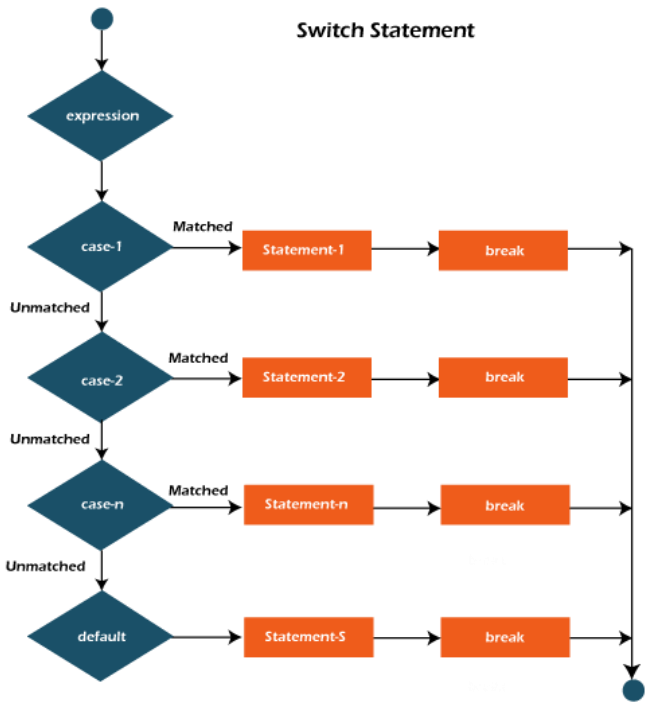
```
You are eligible to donate blood
```

Switch Statements: The Java ***switch statement*** executes one statement from multiple conditions. It is like if-else-if ladder statement. The switch statement works with byte, short, int, long, Enum types, String and some wrapper types like Byte, Short, Int, and Long. in other words, the switch statement tests the equality of a variable against multiple values.

Syntax:

```
switch(expression){
case value1:
 //code to be executed;
 break;  //optional
case value2:
 //code to be executed;
 break;  //optional
......
default:
  code to be executed if all cases are not matched;
}
```

Flowchart of Switch Statement:



Example:

```
public class SwitchExample {
public static void main(String[] args) {
   //Declaring a variable for switch expression
   int number=20;
   //Switch expression
   switch(number){
   //Case statements
   case 10: System.out.println("10");
   break;
   case 20: System.out.println("20");
   break;
   case 30: System.out.println("30");
   break;
   //Default case statement
   default:System.out.println("Not in 10, 20 or 30");
   }
}
}
```

Output:

```
20
```

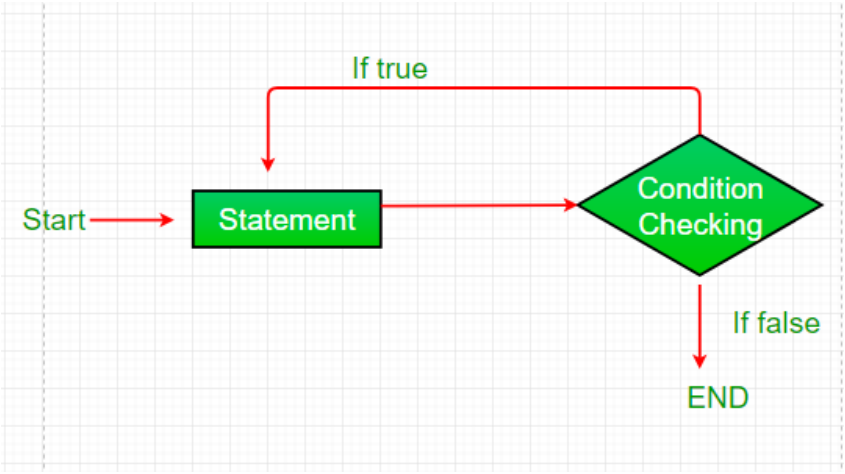
LOOP STATEMENTS:

Looping in programming languages is a feature which facilitates the execution of a set of instructions/functions repeatedly while some condition evaluates to true.
Java provides three ways for executing the loops. While all the ways provide similar basic functionality, they differ in their syntax and condition checking time.

Do while loop: The Java *do-while loop* is used to iterate a part of the program repeatedly, until the specified condition is true. If the number of iterations is not fixed and you must have to execute the loop at least once, it is recommended to use a do-while loop. Java do-while loop is called an **exit control loop**.

Syntax:

```
do{
//code to be executed / loop body
//update statement
}while (condition);
```



Example:

```
public class DoWhileExample {
public static void main(String[] args) {
    int i=1;
    do{
        System.out.println(i);
    i++;
    }while(i<=10);
}
}
```

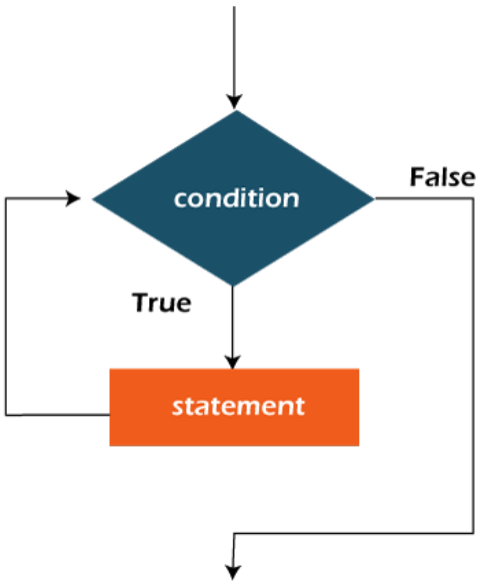
Output:

```
1
2
3
4
5
6
7
8
9
10
```

While loop: The Java *while loop* is used to iterate a part of the program repeatedly until the specified Boolean condition is true. As soon as the Boolean condition becomes false, the loop automatically stops. The while loop is considered as a repeating if statement. If the number of iterations is not fixed, it is recommended to use the while loop.

Syntax:

```
while (condition){
//code to be executed
Increment / decrement statement
}
```



Example:

```
public class WhileExample {
public static void main(String[] args) {
    int i=1;
    while(i<=10){
        System.out.println(i);
    i++;
    }
}
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

For loop: The Java *for loop* is used to iterate a part of the program several times. If the number of iterations is **fixed**, it is recommended to use for loop.

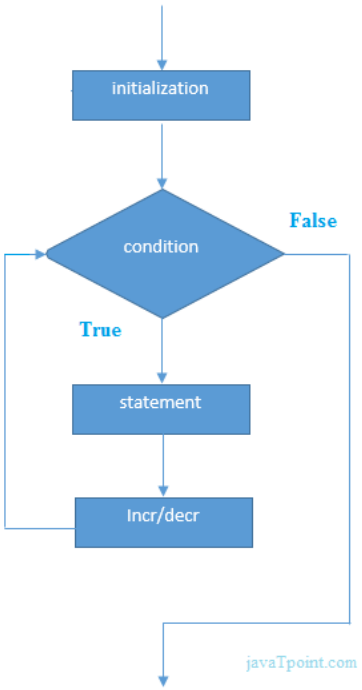
There are three types of for loops in Java.

- Simple for Loop
- For-each or Enhanced for Loop
- Labelled for Loop

Syntax:

```
for(initialization; condition; increment/decrement){
//statement or code to be executed
}
```

Flowchart:



Example:

```
//Java Program to demonstrate the example of for loop
//which prints table of 1
public class ForExample {
public static void main(String[] args) {
   //Code of Java for loop
   for(int i=1;i<=10;i++){
       System.out.println(i);
   }
}
}
```

Output:

```
1
2
3
4
5
6
7
```

```
8
9
10
```

- **For – each loop:** The for-each loop is used to traverse array or collection in Java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation. It works on the basis of elements and not the index. It returns element one by one in the defined variable.

Syntax:

```
for(data_type variable : array_name){
//code to be executed
}
```

Example:

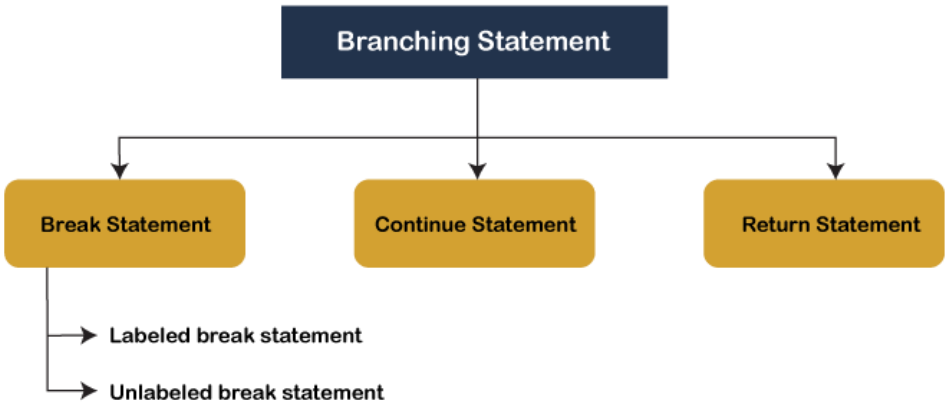
```
//Java For-each loop example which prints the
//elements of the array
public class ForEachExample {
public static void main(String[] args) {
   //Declaring an array
   int arr[]={12,23,44,56,78};
   //Printing array using for-each loop
   for(int i:arr){
       System.out.println(i);
   }
}
}
```

Output:

```
12
23
44
56
78
```

BRANCHING STATEMENTS:

Branching statements are the statements used to jump the flow of execution from one part of a program to another. The **branching statements** allow us to exit from a control statement when a certain condition meet. The **continue** and **break** statements are two essential branching statements used with the control statements. The **break** statement breaks or terminates the loop and transfers the control outside the loop. The **continue** statement skips the current execution and pass the control to the start of the loop. The **return** statement returns a value from a method and this process will be done explicitly.



The break Statement:

The **labeled** and **unlabeled** break statement are the two forms of break statement in Java. The break statement is used for terminating a loop based on a certain condition.

1) Unlabelled break statement: The unlabeled break statement is used to terminate the loop that is inside the loop. It is also used to stop the working of the switch statement. We use the unlabeled break statement to terminate all the loops available in Java.

Syntax:

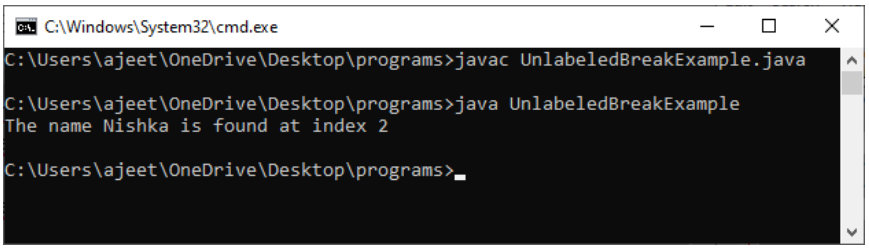
```
for (int; testExpression; update){
   //Code
   if(condition to break){
      break;
   }
}
```

UnlabeledBreakExample:

```
class UnlabeledBreakExample {
   public static void main(String[] args) {

      String[] arr = { "Shubham", "Anubhav", "Nishka", "Gunjan", "Akash" };
       String searchName = "Nishka";

       int j;
      boolean foundName = false;

      for (j = 0; j < arr.length; j++) {
          if (arr[j] == searchName) {
             foundName = true;
              break;
         }
```

```
        }

        if (foundName) {
            System.out.println("The name " + searchName + " is found at index " + j);
        } else {
            System.out.println("The name " +searchName + " is not found in the array");
        }
    }
}
```

Output:



Explanation: In the above program, we search for a name in an array of type string. The **break** keyword is used in the **for loop** using a conditional statement. When the condition is met for the search name, the **break** statement exit us from the loop and pass the control flow to the outside of the loop.

2) Labeled break statement: **Labeled break statement** is another form of break statement. If we have a nested loop in Java and use the break statement in the innermost loop, then it will terminate the innermost loop only, not the outermost loop. The **labeled break** statement is capable of terminating the outermost loop.

Syntax:

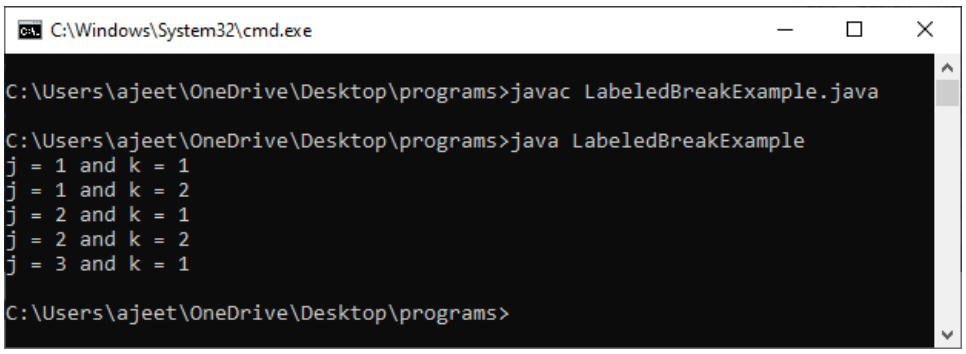
```
label:
for (int; testExpression; update){
    //Code
    for (int; testExpression; update){
        //Code
        if(condition to break){
            break label;
        }
    }
}
```

LabeledBreakExample:

```
class LabeledBreakExample {
```

```
    public static void main(String[] args) {
        int j, k;

        // Labeling the outermost loop as outerMost
        outerMost:
        for(j=1; j<5; j++) {

            // Labeling the innermost loop as innerMost
            innerMost:
            for(k=1; k<3; k++ ) {
                System.out.println("j = " + j + " and k = " +k);

                // Terminating the outemost loop
                if ( j == 3)
                    break outerMost;
            }
        }
    }
}
```

Output:



Explanation: In the above program, we have created nested **for loop**. In the innermost loop, we set a condition to break the outermost loop. When the condition is met, the break statement terminates that loop whose label is associated with the break keyword.

The continue Statement: The **continue** statement is another branching statement used to immediately jump to the next iteration of the loop. It is a special type of loop which breaks current iteration when the condition is met and start the loop with the next iteration. In simple words, it continues the current flow of the program and stop executing the remaining code at the specified condition.

When we have a nested for loop, and we use the continue statement in the innermost loop, it continues only the innermost loop. We can use the continue statement for any control flow statements like **for, while**, and **do-while**.

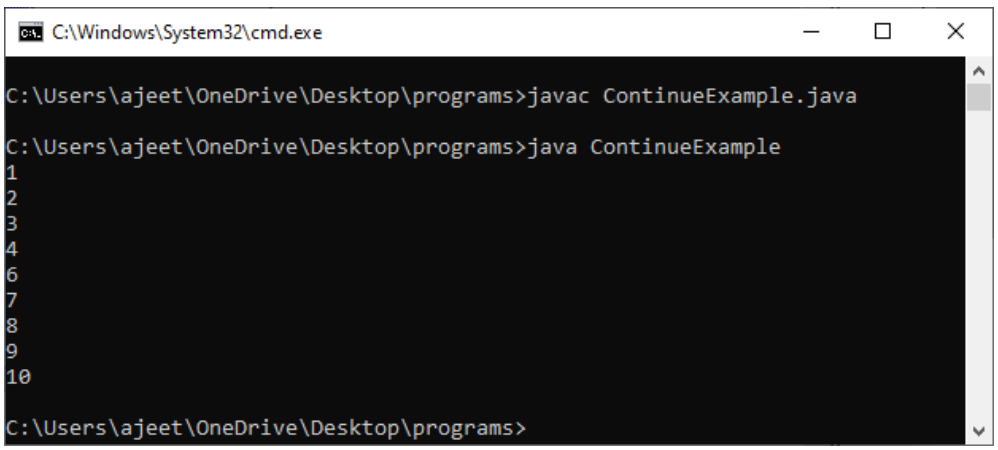
Syntax

```
control-flow-statement;
continue;
```

ContinueExample:

```
public class ContinueExample {
   public static void main(String[] args) {
      //Declare variables
      int x = 1;
       int y = 10;
      //Using do while loop for using contiue statement
       do{
         if(x == y/2){
                        x++;
            continue;//The continue statement skips the remaining statement
          }
         System.out.println(x);
          x++;
      }while(x <= y);
   }
}
```

Output:



Explanation: In the above program, we use a **do-while** loop. We declare two variables **x** and **y**. The **do-while** loop executes until the **x<=y**. In the **do** block of the loop, we check whether the x is equal to y/2 or not. If the condition is matched, the statement skips the print and increment statement and continue the loop.

The return Statement: The **return** statement is also a branching statement, which allows us to explicitly return value from a method. The return statement exits us from the calling method and passes the control flow to where the calling method is invoked. Just like

the break statement, the return statement also has two forms, i.e., one that passes some value with control flow and one that doesn't.

Syntax

```
return value;
```

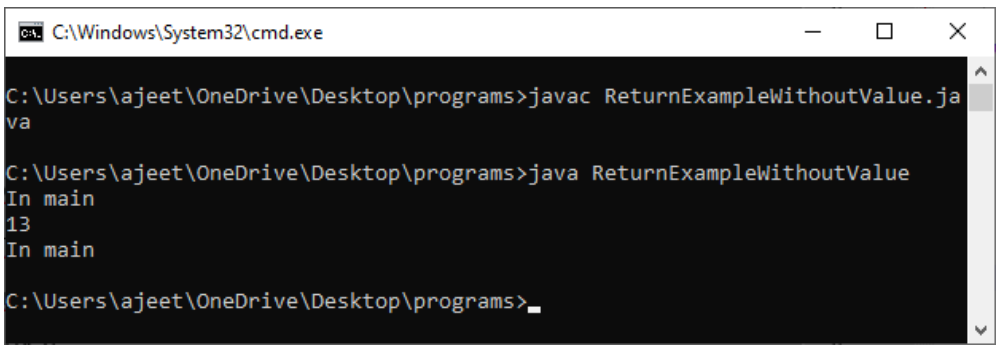
Or,

```
return;
```

ReturnExampleWithoutValue:

```
class ReturnExampleWithoutValue {
   //Declare calling method
    void increment(int number)
   {
       if (number < 10)
          return; //pass the control flow to where this method call
       number++;
      System.out.println(number);
   }
   public static void main(String[] args)
   {
      ReturnExampleWithoutValue obj = new ReturnExampleWithoutValue();
       obj.increment(4);
      System.out.println("In main");
       obj.increment(12);
      System.out.println("In main");
   }
}
```

Output:



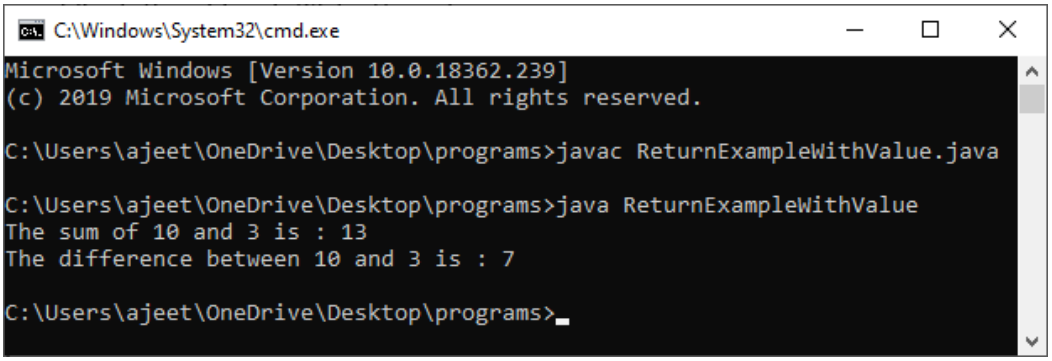
Explanation: In the above code, we create a class having the **increment()** method. In this method, we check whether the number smaller than 10 or not. If the number is less than 10,

the return statement passes the control flow to where the method calls and doesn't execute the increment and print statement.

ReturnExampleWithValue:

```
class ReturnExampleWithValue {
   int sum(int x, int y)
   {
      int sum = 0;
       sum = x + y;
      return sum;
   }
   int difference(int x, int y)
   {
      int diff = 0;
       diff = x - y;

       return diff;
   }
    public static void main(String[] args)
   {
       ReturnExampleWithValue obj = new ReturnExampleWithValue();
      System.out.println("The sum of 10 and 3 is : "+ obj.sum(10, 3));
       System.out.println("The difference between 10 and 3 is : "+ obj.difference(10, 3));
   }
}
```

Output:



Explanation: In the above program, we create two methods that return the integer value. The first method returns the sum of numbers and the second method returns the difference between the numbers. In both methods, an integer value is associated with the return statement to pass the value with the control flow to where the method calls. In the main method, both the methods are called from the print statement, so the print statement directly prints the value returned from the methods.

UNIT – 3 OBJECTS & CLASSES IN JAVA

Object means a real-world entity such as a pen, chair, table, computer, watch, etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Apart from these concepts, there are some other terms which are used in Object-Oriented design:

- Coupling
- Cohesion
- Association
- Aggregation
- Composition

INTRODUCTION TO CLASSES:

A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

- **Modifiers**: A class can be public or has default access.
- **Class keyword:** class keyword is used to create a class.
- **Class name:** The name should begin with an initial letter (capitalized by convention).
- **Superclass (if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
- **Interfaces (if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
- **Body:** The class body surrounded by braces, { }.

Constructors are used for initializing new objects. Fields are variables that provides the state of the class and its objects, and methods are used to implement the behaviour of the class and its objects.

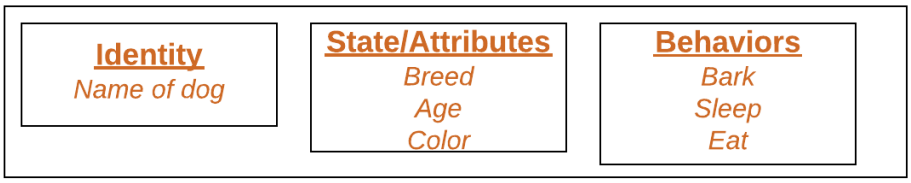
There are various types of classes that are used in real time applications such as **nested classes, anonymous classes, lambda expressions.**

CREATING OBJECTS:

It is a basic unit of Object-Oriented Programming and represents the real life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of:

- **State**: It is represented by attributes of an object. It also reflects the properties of an object.
- **Behaviour**: It is represented by methods of an object. It also reflects the response of an object with other objects.
- **Identity**: It gives a unique name to an object and enables one object to interact with other objects.

Example of an object: dog



Objects correspond to things found in the real world. For example, a graphics program may have objects such as “circle”, “square”, “menu”. An online shopping system might have objects such as “shopping cart”, “customer”, and “product”.

Ways to create object of a class:

There are four ways to create objects in java. Strictly speaking there is only one way(by using *new* keyword),and the rest internally use *new* keyword.

- **Using new keyword:** It is the most common and general way to create object in java. Example:

```
// creating object of class Test

Test t = new Test();
```

- **Using Class.forName(String className) method:** There is a pre-defined class in java.lang package with name Class. The forName(String className) method returns the Class object associated with the class with the given string name.We have to give the fully qualified name for a class. On calling new Instance() method on this Class object returns new instance of the class with the given string name.

```
// creating object of public class Test

// consider class Test present in com.p1 package
Test obj = (Test)Class.forName("com.p1.Test").newInstance();
```

- **Using clone() method:** clone() method is present in Object class. It creates and returns a copy of the object.

```
// creating object of class Test

Test t1 = new Test();
```

```
// creating clone of above object
Test t2 = (Test)t1.clone();
```

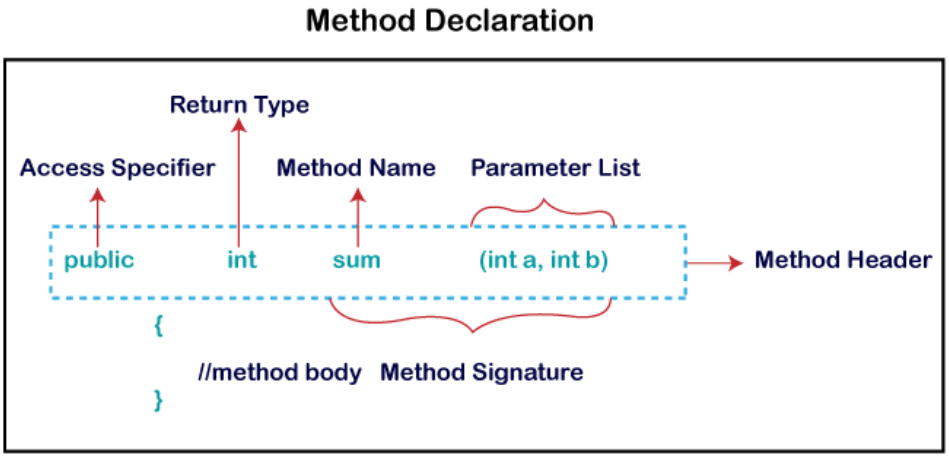
- **Deserialization:** De-serialization is technique of reading an object from the saved state in a file.

```
FileInputStream file = new FileInputStream(filename);
ObjectInputStream in = new ObjectInputStream(file);
Object obj = in.readObject();
```

METHOD:

A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the **reusability** of code. We write a method once and use it many times. We do not require to write code again and again. It also provides the **easy modification** and **readability** of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it. The most important method in Java is the **main()** method.

METHOD DECLARATION: The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments. It has six components that are known as **method header**, as we have shown in the following figure:



Method Signature: Every method has a method signature. It is a part of the method declaration. It includes the **method name** and **parameter list**.

Access Specifiers: Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides **four** types of access specifier:

- **Public:** The method is accessible by all classes when we use public specifier in our application.
- **Private:** When we use a private access specifier, the method is accessible only in the classes in which it is defined.
- **Protected:** When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.

- **Default:** When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

Return Type: Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.

Method Name: It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be **subtraction().** A method is invoked by its name.

Parameter List: It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.

Method Body: It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

CONSTRUCTORS:

- A constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.
- It is a special type of method which is used to initialize the object.
- Every time an object is created using the new() keyword, at least one constructor is called.
- It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

Rules for creating Java constructor:

- Constructor name must be the same as its class name.
- A Constructor must have no explicit return type.
- A Java constructor cannot be abstract, static, final, and synchronized.

Types of Java constructors:

1. Default constructor (no-arg constructor)**.**
2. Parameterized constructor.

1. Default Constructor:

A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax: <class_name>(){}

Purpose: The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

Example:

```
//Java Program to create and call a default constructor
class Bike1{
//creating a default constructor
Bike1(){System.out.println("Bike is created");}
//main method
public static void main(String args[]){
//calling a default constructor
Bike1 b=new Bike1();
}
}
```

Output:

```
Bike is created
```

2.Parameterized constructor:

A constructor which has a specific number of parameters is called a parameterized constructor.

Purpose: The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example:

```
//Java Program to demonstrate the use of the parameterized constructor.
class Student4{
  int id;
  String name;
  //creating a parameterized constructor
  Student4(int i,String n){
  id = i;
  name = n;
  }
  //method to display the values
  void display(){System.out.println(id+" "+name);}

  public static void main(String args[]){
  //creating objects and passing values
  Student4 s1 = new Student4(111,"Karan");
```

```
    Student4 s2 = new Student4(222,"Aryan");
    //calling method to display the values of object
    s1.display();
    s2.display();
   }
}
```

Output:

```
111 Karan
222 Aryan
```

OVERLOADING METHODS & CONSTRUCTORS:

➢ CONSTRUCTOR OVERLOADING:

The constructor overloading can be defined as the concept of having more than one constructor with different parameters so that every constructor can perform a different task.

Use of this () in constructor overloading:

However, we can use this keyword inside the constructor, which can be used to invoke the other constructor of the same class.

Consider the following example to understand the use of this keyword in constructor overloading.

```
public class Student {
//instance variables of the class
int id,passoutYear;
String name,contactNo,collegeName;

Student(String contactNo, String collegeName, int passoutYear){
this.contactNo = contactNo;
this.collegeName = collegeName;
this.passoutYear = passoutYear;
}

Student(int id, String name){
this("9899234455", "IIT Kanpur", 2018);
this.id = id;
this.name = name;
}

public static void main(String[] args) {
//object creation
```

```
Student s = new Student(101, "John");
System.out.println("Printing Student Information: \n");
System.out.println("Name: "+s.name+"\nId: "+s.id+"\nContact No.: "+s.contactNo+"\nCollege Name: "+
s.contactNo+"\nPassing Year: "+s.passoutYear);
}
}
```

Output:

```
Printing Student Information:

Name: John
Id: 101
Contact No.: 9899234455
College Name: 9899234455
Passing Year: 2018
```

➢ METHOD OVERLOADING:

If a Class has multiple methods having same name but different in parameters, it is known as **Method Overloading**. If we have to perform only one operation, having same name of the methods increases the readability of the program.

Advantages: Method overloading increases the readability of the program.

Different ways to overload the method:

- By changing number of arguments.
- By changing the data type.

Method Overloading: changing no. of arguments

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

```
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}}
```

Output:

```
22
```

```
33
```

Method Overloading: changing data type of arguments

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

```
class Adder{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```

Output:

```
22
24.9
```

METHOD OVERRIDING:

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**. In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding:

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism.

Rules for Java Method Overriding

- The method must have the same name as in the parent class.
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).

Example of method overriding:

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

```
//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class.
```

```
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");}

  public static void main(String args[]){
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }
}
```

Output:

```
Bike is running safely
```

MODIFIERS:

Java provides a number of non-access modifiers to achieve many other functionalities.

- The *static* modifier for creating class methods and variables.
- The *final* modifier for finalizing the implementations of classes, methods, and variables.
- The *abstract* modifier for creating abstract classes and methods.

The Static Modifier:

The *static* keyword is used to create variables that will exist independently of any instances created for the class. Only one copy of the static variable exists regardless of the number of instances of the class. Static variables are also known as class variables. Local variables cannot be declared static.

The static keyword is used to create methods that will exist independently of any instances created for the class.Static methods do not use any instance variables of any object of the class they are defined in. Static methods take all the data from parameters and compute something from those parameters, with no reference to variables.

Class variables and methods can be accessed using the class name followed by a dot and the name of the variable or method.

Example: The static modifier is used to create class methods and variables, as in the following example −

```
public class InstanceCounter {

   private static int numInstances = 0;
```

```
   protected static int getCount() {
      return numInstances;
   }

   private static void addInstance() {
      numInstances++;
   }

   InstanceCounter() {
      InstanceCounter.addInstance();
   }

   public static void main(String[] arguments) {
      System.out.println("Starting with " + InstanceCounter.getCount() + "
instances");

      for (int i = 0; i < 500; ++i) {
         new InstanceCounter();
      }
      System.out.println("Created " + InstanceCounter.getCount() + "
instances");
   }
}
```

Output:

```
Started with 0 instances
Created 500 instances
```

The Final Modifier:

A final variable can be explicitly initialized only once. A reference variable declared final can never be reassigned to refer to a different object. However, the data within the object can be changed. So, the state of the object can be changed but not the reference.

With variables, the *final* modifier often is used with *static* to make the constant a class variable.

Example

```
public class Test {
   final int value = 10;

   // The following are examples of declaring constants:
   public static final int BOXWIDTH = 6;
   static final String TITLE = "Manager";

   public void changeValue() {
      value = 12;   // will give an error
   }
}
```

A final method cannot be overridden by any subclasses. As mentioned previously, the final modifier prevents a method from being modified in a subclass. The main intention of making a method final would be that the content of the method should not be changed by any outsider.

Example: You declare methods using the *final* modifier in the class declaration, as in the following example −

```
public class Test {
   public final void changeName() {
      // body of method
   }
}
```

The **main purpose** of using a class being declared as *final* is to prevent the class from being subclassed. If a class is marked as final then no class can inherit any feature from the final class. **Example:**

```
public final class Test {
   // body of class
}
```

The Abstract Modifier:

An abstract class can never be instantiated. If a class is declared as abstract then the sole purpose is for the class to be extended. A class cannot be both abstract and final (since a final class cannot be extended). If a class contains abstract methods, then the class should be declared abstract. Otherwise, a compile error will be thrown. An abstract class may contain both abstract methods as well normal methods.

Example

```
abstract class Caravan {
   private double price;
   private String model;
   private String year;
   public abstract void goFast();   // an abstract method
   public abstract void changeColor();
}
```

An abstract method is a method declared without any implementation. The methods body (implementation) is provided by the subclass. Abstract methods can never be final or strict. Any class that extends an abstract class must implement all the abstract methods of the super class, unless the subclass is also an abstract class.

If a class contains one or more abstract methods, then the class must be declared abstract. An abstract class does not need to contain abstract methods. The abstract method ends with a semicolon. Example: public abstract sample ();

Example:

```
public abstract class SuperClass {
   abstract void m();   // abstract method
}

class SubClass extends SuperClass {
   // implements the abstract method
   void m() {
      .........
   }
}
```

INHERITANCE BASICS:

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviours of a parent object. It is an important part of OOPS. The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a parent-child relationship.

Use of Inheritance:

- For Method Overloading (So, Runtime Polymorphism can be achieved.)
- For Code Reusability.

Terms used in Inheritance:

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

Syntax of Java Inheritance:

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

Example: Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

```
class Employee{
 float salary=40000;
}
```

```
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

Types of Inheritance:

There can be three types of inheritance in java: single, multilevel and hierarchical.

- When a class inherits another class, it is known as a ***single inheritance*.**
- When there is a chain of inheritance, it is known as ***multilevel inheritance*.**
- When two or more classes inherits a single class, it is known as ***hierarchical inheritance*.**

UNIT- 4 ARRAYS, STRINGS & VECTORS

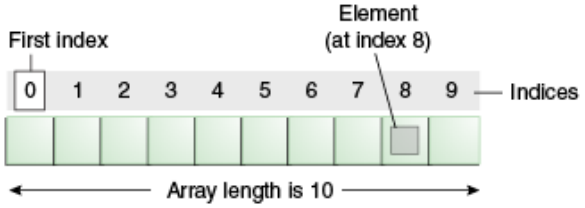
ARRAYS:

Java array is an object which contains elements of a similar data type. Additionally, the elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on. Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the size of operator.

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimensional or multidimensional arrays in Java.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.



Advantages:

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.

- **Random access:** We can get any data located at an index position.

Disadvantages:

- **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

There are two types of arrays:

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array:

Syntax to Declare an Array in Java:

dataType[] arr; (or)
dataType []arr; (or)
dataType arr[];

Instantiation of an Array in Java: arrayRefVar=**new** datatype[size];

Example of Java Array:

```
//Java Program to illustrate how to declare, instantiate, initialize
//and traverse the Java array.
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]=70;
a[3]=40;
a[4]=50;
//traversing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

Output:

```
10
20
70
40
```

Multidimensional Array in Java:

In such case, data is stored in row and column-based index (also known as matrix form).

Syntax to Declare Multidimensional Array in Java:

```
dataType[][] arrayRefVar; (or)
dataType [][]arrayRefVar; (or)
dataType arrayRefVar[][]; (or)
dataType []arrayRefVar[];
```

Example to instantiate Multidimensional Array:

```
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;
```

Example of Multidimensional Java Array:

```
//Java Program to illustrate the use of multidimensional array
class Testarray3{
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();
}
}}
```

Output:

```
1 2 3
2 4 5
4 4 5
```

STRINGS:

In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string. **Java String** class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

Syntax: `<String_Type> <string_variable> = "<sequence_of_string>";`

String Example:

```
public class StringExample{
public static void main(String args[]){
String s1="java";//creating string by Java string literal
char ch[]={'s','t','r','i','n','g','s'};
String s2=new String(ch);//converting char array to string
String s3=new String("example");//creating Java string by new keyword
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}}
```

Output:

```
java
strings
example
```

➢ String class:

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

Method	Description
char charAt(int index)	It returns char value for the particular index
int length()	It returns string length
static String format(String format, Object... args)	It returns a formatted string.
static String format(Locale l, String format, Object... args)	It returns formatted string with given locale.
String substring(int beginIndex)	It returns substring for given begin index.

String substring(int beginIndex, int endIndex)	It returns substring for given begin index and end index.
boolean contains(CharSequence s)	It returns true or false after matching the sequence of char value.
static String join(CharSequence delimiter, CharSequence... elements)	It returns a joined string.
static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	It returns a joined string.
boolean equals(Object another)	It checks the equality of string with the given object.
boolean isEmpty()	It checks if string is empty.
String concat(String str)	It concatenates the specified string.
String replace(char old, char new)	It replaces all occurrences of the specified char value.
String replace(CharSequence old, CharSequence new)	It replaces all occurrences of the specified CharSequence.
static String equalsIgnoreCase(String another)	It compares another string. It doesn't check case.
String[] split(String regex)	It returns a split string matching regex.
String[] split(String regex, int limit)	It returns a split string matching regex and limit.
String intern()	It returns an interned string.
int indexOf(int ch)	It returns the specified char value index.
int indexOf(int ch, int fromIndex)	It returns the specified char value index starting with given index.
int indexOf(String substring)	It returns the specified substring index.
int indexOf(String substring, int fromIndex)	It returns the specified substring index starting with given index.
String toLowerCase()	It returns a string in lowercase.
String toLowerCase(Locale l)	It returns a string in lowercase using specified locale.
String toUpperCase()	It returns a string in uppercase.
String toUpperCase(Locale l)	It returns a string in uppercase using specified locale.
String trim()	It removes beginning and ending spaces of this string.
static String valueOf(int value)	It converts given type into string. It is an overloaded method.

➢ String Buffer class:

Java String Buffer class is used to create mutable (modifiable) String objects. The StringBuffer class in Java is the same as String class except it is mutable i.e. it can be changed.

Important Constructors of String Buffer Class:

Constructor	Description
StringBuffer()	It creates an empty String buffer with the initial capacity of 16.
StringBuffer(String str)	It creates a String buffer with the specified string..
StringBuffer(int capacity)	It creates an empty String buffer with the specified capacity as length.

Important methods of String Buffer class:

Modifier and Type	Method	Description
public synchronized StringBuffer	append(String s)	It is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
public synchronized StringBuffer	insert(int offset, String s)	It is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
public synchronized StringBuffer	replace(int startIndex, int endIndex, String str)	It is used to replace the string from specified startIndex and endIndex.
public synchronized StringBuffer	delete(int startIndex, int endIndex)	It is used to delete the string from specified startIndex and endIndex.
public synchronized StringBuffer	reverse()	is used to reverse the string.
public int	capacity()	It is used to return the current capacity.
public void	ensureCapacity(int minimumCapacity)	It is used to ensure the capacity at least equal to the given minimum.
public char	charAt(int index)	It is used to return the character at the specified position.
public int	length()	It is used to return the length of the string i.e. total number of characters.
public String	substring(int beginIndex)	It is used to return the substring from the specified beginIndex.
public String	substring(int beginIndex, int endIndex)	It is used to return the substring from the specified beginIndex and endIndex.

VECTOR CLASS:

Vector is like the *dynamic array* which can grow or shrink its size. Unlike array, we can store n-number of elements in it as there is no size limit. It is a part of Java Collection framework since Java 1.2. It is found in the java. util package and implements the *List* interface, so we can use all the methods of List interface here.

It is recommended to use the Vector class in the thread-safe implementation only. If you don't need to use the thread-safe implementation, you should use the Array List, the Array List will perform better in such case. The Iterators returned by the Vector class are *fail-fast*. In case of concurrent modification, it fails and throws the ConcurrentModificationException.

It is similar to the Array List, but with two differences-

- Vector is synchronized.

- Java Vector contains many legacy methods that are not the part of a collections framework.

Java Vector class Declaration:

```
public class Vector<E>
extends Object<E>
implements List<E>, Cloneable, Serializable
```

Java Vector Constructors:

Constructor	Description
vector()	It constructs an empty vector with the default size as 10.
vector(int initialCapacity)	It constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.
vector(int initialCapacity, int capacityIncrement)	It constructs an empty vector with the specified initial capacity and capacity increment.
Vector(Collection<? extends E> c)	It constructs a vector that contains the elements of a collection c.

Java Vector Methods:

Method	Description
add()	It is used to append the specified element in the given vector.
addAll()	It is used to append all of the elements in the specified collection to the end of this Vector.
addElement()	It is used to append the specified component to the end of this vector. It increases the vector size by one.
capacity()	It is used to get the current capacity of this vector.
clear()	It is used to delete all of the elements from this vector.
clone()	It returns a clone of this vector.
contains()	It returns true if the vector contains the specified element.
containsAll()	It returns true if the vector contains all of the elements in the specified collection.
copyInto()	It is used to copy the components of the vector into the specified array.
elementAt()	It is used to get the component at the specified index.
elements()	It returns an enumeration of the components of a vector.
ensureCapacity()	It is used to increase the capacity of the vector which is in use, if necessary. It ensures that the vector can hold at least the number of components specified by the minimum capacity argument.
equals()	It is used to compare the specified object with the vector for equality.
firstElement()	It is used to get the first component of the vector.
forEach()	It is used to perform the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
get()	It is used to get an element at the specified position in the vector.
hashCode()	It is used to get the hash code value of a vector.
indexOf()	It is used to get the index of the first occurrence of the specified element in the vector. It returns -1 if the vector does not contain the element.
insertElementAt()	It is used to insert the specified object as a component in the given vector at the specified index.
isEmpty()	It is used to check if this vector has no components.

iterator()	It is used to get an iterator over the elements in the list in proper sequence.
lastElement()	It is used to get the last component of the vector.
lastIndexOf()	It is used to get the index of the last occurrence of the specified element in the vector. It returns -1 if the vector does not contain the element.
listIterator()	It is used to get a list iterator over the elements in the list in proper sequence.
remove()	It is used to remove the specified element from the vector. If the vector does not contain the element, it is unchanged.
removeAll()	It is used to delete all the elements from the vector that are present in the specified collection.
removeAllElements()	It is used to remove all elements from the vector and set the size of the vector to zero.
removeElement()	It is used to remove the first (lowest-indexed) occurrence of the argument from the vector.
removeElementAt()	It is used to delete the component at the specified index.
removeIf()	It is used to remove all of the elements of the collection that satisfy the given predicate.
removeRange()	It is used to delete all of the elements from the vector whose index is between fromIndex, inclusive and toIndex, exclusive.
replaceAll()	It is used to replace each element of the list with the result of applying the operator to that element.
retainAll()	It is used to retain only that element in the vector which is contained in the specified collection.
set()	It is used to replace the element at the specified position in the vector with the specified element.
setElementAt()	It is used to set the component at the specified index of the vector to the specified object.
setSize()	It is used to set the size of the given vector.
size()	It is used to get the number of components in the given vector.
sort()	It is used to sort the list according to the order induced by the specified Comparator.
spliterator()	It is used to create a late-binding and fail-fast Spliterator over the elements in the list.
subList()	It is used to get a view of the portion of the list between fromIndex, inclusive, and toIndex, exclusive.
toArray()	It is used to get an array containing all of the elements in this vector in correct order.
toString()	It is used to get a string representation of the vector.
trimToSize()	It is used to trim the capacity of the vector to the vector's current size.

Example:

```
import java.util.*;
public class VectorExample {
    public static void main(String args[]) {
      //Create a vector
       Vector<String> vec = new Vector<String>();
      //Adding elements using add() method of List
       vec.add("Tiger");
```

```
        vec.add("Lion");
         vec.add("Dog");
        vec.add("Elephant");
         //Adding elements using addElement() method of Vector
        vec.addElement("Rat");
         vec.addElement("Cat");
        vec.addElement("Deer");

        System.out.println("Elements are: "+vec);
      }
}
```

Output:

```
Elements are: [Tiger, Lion, Dog, Elephant, Rat, Cat, Deer]
```

WRAPPER CLASS:

A Wrapper class is a class whose object wraps or contains primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types. In other words, we can wrap a primitive value into a wrapper class object. Since J2SE 5.0, **autoboxing** and **unboxing** feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Need of Wrapper Classes:

- **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
- **Synchronization:** Java synchronization works with objects in Multithreading.
- **java.util package:** The java.util package provides the utility classes to deal with objects.
- **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (Array List, LinkedList, Vector, HashSet, LinkedHashSet, Tree Set, Priority Queue, Array Deque, etc.) deal with objects only.

The eight classes of the *java.lang* package are known as wrapper classes in Java. The list of eight wrapper classes is given below:

Primitive Type	Wrapper class

boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

- **Autoboxing:** Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example – conversion of int to Integer, long to Long, double to Double etc.
- **Unboxing:** It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing. For example – conversion of Integer to int, long to long, Double to double, etc.

UNIT – 5 PACKAGES & INTERFACES

PACKAGES:

Package in Java is a mechanism to encapsulate a group of classes, sub packages and interfaces. Packages are used for:

- Preventing naming conflicts. For example, there can be two classes with name Employee in two packages, college.staff.cse. Employee and college.staff.ee. Employee
- Making searching/locating and usage of classes, interfaces, enumerations and annotations easier
- Providing controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.
- Packages can be considered as data encapsulation (or data-hiding).

Types of packages:

1.Built-in Packages:

These packages consist of a large number of classes which are a part of Java **API**.Some of the commonly used built-in packages are:

java.lang: Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.
java.io: Contains classed for supporting input / output operations.
java.util: Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.
java.applet: Contains classes for creating Applets.
java.awt: Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).
java.net: Contain classes for supporting networking operations.

2.User-defined packages:
These are the packages that are defined by the user. First we create a directory **myPackage** (name should be same as the name of the package). Then create the **MyClass** inside the directory with the first statement being the **package names**.

➢ BRIEF DISCUSSION ON CLASSPATH:

CLASSPATH is an environment variable which is used by Application ClassLoader to locate and load the .class files. The CLASSPATH defines the path, to find third-party and user-defined classes that are not extensions or part of Java platform. Include all the directories which contain .class files and JAR files when setting the CLASSPATH. You need to set the CLASSPATH if:

- You need to load a class that is not present in the current directory or any sub-directories.
- You need to load a class that is not in a location specified by the extensions mechanism.

The CLASSPATH depends on what you are setting the CLASSPATH. The CLASSPATH has a directory name or file name at the end. The following points describe what should be the end of the CLASSPATH.

- If a JAR or zip, the file contains class files, the CLASSPATH end with the name of the zip or JAR file.
- If class files placed in an unnamed package, the CLASSPATH ends with the directory that contains the class files.
- If class files placed in a named package, the CLASSPATH ends with the directory that contains the root package in the full package name, that is the first package in the full package name.

The default value of CLASSPATH is a dot (.). It means the only current directory searched. The default value of CLASSPATH overrides when you set the CLASSPATH variable or using the -class path command (for short -cp). Put a dot (.) in the new setting if you want to include the current directory in the search path.

➢ IMPORTING A PACKAGE:

In java, the ***import*** keyword used to import built-in and user-defined packages. When a package has imported, we can refer to all the classes of that package using their name directly. The import statement must be after the package statement, and before any other statement.

Using an import statement, we may import a specific class or all the classes from a package.

- Using one import statement, we may import only one package or a class.
- Using an import statement, we cannot import a class directly, but it must be a part of a package.
- A program may contain any number of import statements.

Importing specific class:

Using an importing statement, we can import a specific class. The following syntax is employed to import a specific class.

Syntax:

```
import packageName.ClassName;
```

Example:

```
package myPackage;

import java.util.Scanner;

public class ImportingExample {

    public static void main(String[] args) {

        Scanner read = new Scanner(System.in);

        int i = read.nextInt();

        System.out.println("You have entered a number " + i);
    }
}
```

In the above code, the class **Importing Example** belongs to **my Package** package, and it also importing a class called **Scanner** from **java.util** package.

➢ JAVA API PACKAGE:

An API can be described as a way to enable computers to possess a common interface, to allow them to communicate with each other. Java Application Programming Interface (API) is the area of Java development kit (JDK). An API includes classes, interfaces, packages and also their methods, fields, and constructors.
All these built-in classes give benefits to the programmer. Only programmers understand how to apply that class. A user interface offers the basic user interaction between user and computer; in the same manner, the API works as an application program interface that gives connection amongst the software as well as the consumer. API includes classes and packages which usually assist a programmer in minimizing the lines of a program.

Advantages of API in Java:

- **Automation**: With Java APIs, instead of people, computer systems can control the work. Throughout APIs, organizations can upgrade workflows to create them faster and even more effective.

- **Application**: Since Java APIs can easily access the software components, the delivery of services, as well as data, is much more flexible.

- **Efficiency**: Once access is offered for a Java API, the content produced could be released instantly, and it is readily available for every single channel. This enables it to be distributed as well as sent out quickly.

- **Integration**: Java APIs enables content to be embedded by any site or perhaps software easier. This ensures additional fluid data delivery and then a built-in user experience.

Fundamentals of API:

These are some fundamentals of API; once you figure out all of these, you can begin to try out:

- Build an API to respond OK then that serve a string or web content
- To respond and consume JSON/XML
- To handle form submissions
- To consume data (as form/JSON/XML), validate data and persist in a database
- To connect to another API
- To persist data to various data stores SQL/NoSQL
- To update data in the database
- To delete data in a database
- Secure your API

INTERFACES:

- An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.
- The interface in Java is *a* mechanism to achieve abstraction.
- There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.
- In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.
- Java Interface also **represents the IS-A relationship**.

Syntax:

```
interface <interface_name> {

    // declare constant fields

    // declare methods that abstract

    // by default.
```

```
}
```

Why do we use interface?

- It is used to achieve total abstraction.
- Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance.
- It is also used to achieve loose coupling.
- Interfaces are used to implement abstraction. So, the question arises why use interfaces when we have abstract classes? The reason is, abstract classes may contain non-final variables, whereas variables in interface are final, public and static.

Implementing an interface:

To declare an interface, use **interface** keyword. It is used to provide total abstraction. That means all the methods in an interface are declared with an empty body and are public and all fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface. To implement interface use **implements** keyword.

Extending an interface:

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface. The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

UNIT- 6 EXCEPTION HANDING

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IO Exception, SQL Exception, Remote Exception, etc. The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions.

Why use EXCEPTION HANDLING?

Handling the exception is nothing but converting system error generated message into user friendly error message. Whenever an exception occurs in the java application, JVM will create an object of appropriate exception of sub class and generates system error message, this system generated messages are not understandable by user so need to convert it into user friendly error message. You can convert system error message into user friendly error message by using exception handling feature of java.

Fundamentals of Exception Handling:

Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.

Keyword	Description

try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Types of Exception Handling:

There are three types of exceptions namely:

Checked Exception: These are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using the throws keyword. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

Unchecked Exception: These are the exceptions that are not checked at compile time. In C++, all exceptions are unchecked, so it is not forced by the compiler to either handle or specify the exception. It is up to the programmers to be civilized, and specify or catch the exceptions. In Java exceptions under *Error* and *Runtime Exception* classes are unchecked exceptions, everything else under throwable is checked. For example, Arithmetic Exception, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

Error: Error is irrecoverable. Some examples of errors are OutOfMemoryError, VirtualMachineError, Assertion Error etc.

Using Exception Handling using try and catch:

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method. Java **catch** block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

Syntax:

```
try{
//code that may throw an exception
}catch(Exception_class_Name ref){}
```

TryCatchExample:

```
public class TryCatchExample2 {

  public static void main(String[] args) {
    try
    {
    int data=50/0; //may throw exception
    }
      //handling the exception
    catch(ArithmeticException e)
    {
      System.out.println(e);
    }
    System.out.println("rest of the code");
  }

}
```

Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

Using Exception Handling using Multiple catch statements:

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block. At a time only one exception occurs and at a time only one catch block is executed. All catch blocks must be ordered from most specific to most general, i.e., catch for Arithmetic Exception must come before catch for Exception.

MultipleCatchBlock:

```
public class MultipleCatchBlock2 {

  public static void main(String[] args) {

      try{
        int a[]=new int[5];

        System.out.println(a[10]);
        }
       catch(ArithmeticException e)
```

```
            {
            System.out.println("Arithmetic Exception occurs");
            }
        catch(ArrayIndexOutOfBoundsException e)
            {
            System.out.println("ArrayIndexOutOfBounds Exception occurs");
            }
        catch(Exception e)
            {
            System.out.println("Parent Exception occurs");
            }
        System.out.println("rest of the code");
    }
}
```

Output:

```
ArrayIndexOutOfBounds Exception occurs
rest of the code
```

Using Exception Handling using Nested try statements:

- In Java, using a try block inside another try block is permitted. It is called as nested try block. Every statement that we enter a statement in try block, context of that exception is pushed onto the stack.
- For example, the **inner try block** can be used to handle **ArrayIndexOutOfBoundsException** while the **outer try block** can handle the **ArithemeticException** (division by zero).
- Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax:

```
....
//main try block
try
{
    statement 1;
    statement 2;
//try catch block within another try block
    try
    {
        statement 3;
```

```
      statement 4;
//try catch block within nested try block
      try
      {
          statement 5;
         statement 6;
    }
      catch(Exception e2)
      {
//exception message
      }

   }
   catch(Exception e1)
   {
//exception message
   }
}
//catch block of parent (outer) try block
catch(Exception e3)
{
//exception message
}
....
```

Throwing your own Exception:

In Java, we can create our own exceptions that are derived classes of the Exception class. Creating our own Exception is known as custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user need.

Using the custom exception, we can have your own exception and message. Here, we have passed a string to the constructor of superclass i.e., Exception class that can be obtained using getMessage() method on the object we have created.

Following are few of the reasons to use custom exceptions:

- To catch and provide specific treatment to a subset of existing Java exceptions.
- Business logic exceptions: These are the exceptions related to business logic and workflow. It is useful for the application users or the developers to understand the exact problem.

UNIT – 7 APPLET PROGRAMMING

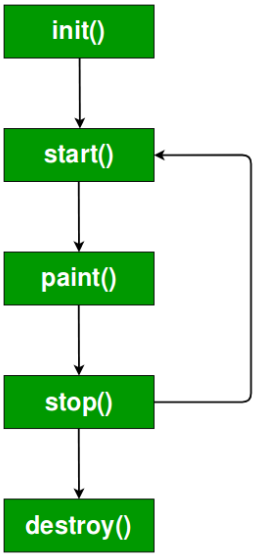
An applet is a Java program that can be embedded into a web page. It runs inside the web browser and works at client side. An applet is embedded in an HTML page using the APPLET or OBJECT tag and hosted on a web server. Applets are used to make the website more dynamic and entertaining.

Advantages:

- It works at client side so less response time.
- Secured
- It can be executed by browsers running under many platforms, including Linux, Windows, Mac Os etc.

Drawbacks: Plugin is required at client browser to execute applet.

Life cycle of an applet:



When an applet begins, the following methods are called, in this sequence:

- **init( ):** The **init( )** method is the first method to be called. This is where you should initialize variables. This method is called **only once** during the run time of your applet.
- **start( ):** The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped. Note that **init( )** is called once i.e. when the first time an applet is loaded whereas **start( )** is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start( )**.
- **paint( ):** The **paint( )** method is called each time an AWT-based applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored. **paint( )** is also called when the applet begins execution. Whatever the cause,

whenever the applet must redraw its output, **paint( )** is called.

When an applet is terminated, the following sequence of method calls takes place:

- **stop( ):**The **stop( )** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When **stop( )** is called, the applet is probably running. You should use **stop( )** to suspend threads that don't need to run when the applet is not visible. You can restart them when **start( )** is called if the user returns to the page
- **destroy( ):**The **destroy( )** method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The **stop( )** method is always called before **destroy( )**.

APPLET TAG:

Applet tag <*applet*> is required to load and start an applet either in a browser window or in an **appletviewer**(provided by **JDK**). At present, many current version of browsers such a Google Chrome and Mozilla Firefox(except Internet Explorer) have stopped displaying applets or recognizing <applet> tag, hence the only reliable tool to display an applet is **appletviewer**.

Security Restrictions when using Applets:

Due to security reasons, the following restrictions are imposed on Java applets:

1. An applet cannot load libraries or define native methods.
2. An applet cannot ordinarily read or write files on the execution host.
3. An applet cannot read certain system properties.
4. An applet cannot make network connections except to the host that it came from.
5. An applet cannot start any program on the host that's executing it.

Features of Applets over HTML:

- Displaying dynamic web pages of a web application.
- Playing sound files.
- Displaying documents
- Playing animations

APPLET PARAMETER:

We can get any information from the HTML file as a parameter. For this purpose, Applet class provides a method named getParameter().

Syntax: `public String getParameter(String parameterName)`

Example:

```
import java.applet.Applet;
```

```
import java.awt.Graphics;

public class UseParam extends Applet{

public void paint(Graphics g){
String str=getParameter("msg");
g.drawString(str,50, 50);
}
}
```

UNIT – 8 WORKING WITH GRAPHICS

- Java AWT (Abstract Window Toolkit) is *an API to develop Graphical User Interface (GUI) or windows-based applications* in Java.
- Java AWT components are platform-dependent i.e., components are displayed according to the view of operating system. AWT is heavy weight i.e., its components are using the resources of underlying operating system (OS).
- The java.awt package provides classes for AWT API such as Text Field, Label, Text Area, Radio Button, Checkbox, Choice, List etc.
- The AWT tutorial will help the user to understand Java GUI programming in simple and easy steps.
- Following is the declaration for **java.awt. Graphics** class:

```
public abstract class Graphics
   extends Object
```

Class constructors:

S.N.	Constructor & Description
1	**Graphics() ()** Constructs a new Graphics object.

Class methods:

S.N.	Method & Description
1	**abstract void clearRect(int x, int y, int width, int height)** Clears the specified rectangle by filling it with the background color of the current drawing surface.
2	**abstract void clipRect(int x, int y, int width, int height)** Intersects the current clip with the specified rectangle.

3	**abstract void copyArea(int x, int y, int width, int height, int dx, int dy)** Copies an area of the component by a distance specified by dx and dy.
4	**abstract Graphics create()** Creates a new Graphics object that is a copy of this Graphics object.
5	**Graphics create(int x, int y, int width, int height)** Creates a new Graphics object based on this Graphics object, but with a new translation and clip area.
6	**abstract void dispose()** Disposes of this graphics context and releases any system resources that it is using.
7	**void draw3DRect(int x, int y, int width, int height, boolean raised)** Draws a 3-D highlighted outline of the specified rectangle.
8	**abstract void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)** Draws the outline of a circular or elliptical arc covering the specified rectangle.
9	**void drawBytes(byte[] data, int offset, int length, int x, int y)** Draws the text given by the specified byte array, using this graphics context's current font and color.
10	**void drawChars(char[] data, int offset, int length, int x, int y)** Draws the text given by the specified character array, using this graphics context's current font and color.
11	**abstract boolean drawImage(Image img, int x, int y, Color bgcolor, ImageObserver observer)** Draws as much of the specified image as is currently available.
12	**abstract boolean drawImage(Image img, int x, int y, ImageObserver observer)** Draws as much of the specified image as is currently available.
13	**abstract boolean drawImage(Image img, int x, int y, int width, int height, Color bgcolor, ImageObserver observer)** Draws as much of the specified image as has already been scaled to fit inside the specified rectangle.
14	**abstract boolean drawImage(Image img, int x, int y, int width, int height, ImageObserver observer)** Draws as much of the specified image as has already been scaled to fit inside the specified rectangle.
15	**abstract boolean drawImage(Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, Color bgcolor, ImageObserver observer)** Draws as much of the specified area of the specified image as is currently available, scaling it on the fly to fit inside the specified area of the destination drawable surface.

16	**abstract boolean drawImage(Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, ImageObserver observer)** Draws as much of the specified area of the specified image as is currently available, scaling it on the fly to fit inside the specified area of the destination drawable surface.
17	**abstract void drawLine(int x1, int y1, int x2, int y2)** Draws a line, using the current color, between the points (x1, y1) and (x2, y2) in this graphics context's coordinate system.
18	**abstract void drawOval(int x, int y, int width, int height)** Draws the outline of an oval.
19	**abstract void drawPolygon(int[] xPoints, int[] yPoints, int nPoints)** Draws a closed polygon defined by arrays of x and y coordinates.
20	**void drawPolygon(Polygon p)** Draws the outline of a polygon defined by the specified Polygon object.
21	**abstract void drawPolyline(int[] xPoints, int[] yPoints, int nPoints)** Draws a sequence of connected lines defined by arrays of x and y coordinates.
22	**void drawRect(int x, int y, int width, int height)** Draws the outline of the specified rectangle.
23	**abstract void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)** Draws an outlined round-cornered rectangle using this graphics context's current color.
24	**abstract void drawString(AttributedCharacterIterator iterator, int x, int y)** Renders the text of the specified iterator applying its attributes in accordance with the specification of the TextAttribute class.
25	**abstract void drawString(String str, int x, int y)** Draws the text given by the specified string, using this graphics context's current font and color.
26	**void fill3DRect(int x, int y, int width, int height, boolean raised)** Paints a 3-D highlighted rectangle filled with the current color.
27	**abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)** Fills a circular or elliptical arc covering the specified rectangle.
28	**abstract void fillOval(int x, int y, int width, int height)** Fills an oval bounded by the specified rectangle with the current color.
29	**abstract void fillPolygon(int[] xPoints, int[] yPoints, int nPoints)** Fills a closed polygon defined by arrays of x and y coordinates.

30	**void fillPolygon(Polygon p)** Fills the polygon defined by the specified Polygon object with the graphics context's current color.
31	**abstract void fillRect(int x, int y, int width, int height)** Fills the specified rectangle.
32	**abstract void fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)** Fills the specified rounded corner rectangle with the current color.
33	**void finalize()** Disposes of this graphics context once it is no longer referenced.
34	**abstract Shape getClip()** Gets the current clipping area.
35	**abstract Rectangle getClipBounds()** Returns the bounding rectangle of the current clipping area.
36	**Rectangle getClipBounds(Rectangle r)** Returns the bounding rectangle of the current clipping area.
37	**Rectangle getClipRect()** Deprecated. As of JDK version 1.1, replaced by getClipBounds().
38	**abstract Color getColor()** Gets this graphics context's current color.
39	**abstract Font getFont()** Gets the current font.
40	**FontMetrics getFontMetrics()** Gets the font metrics of the current font.
41	**abstract FontMetrics getFontMetrics(Font f)** Gets the font metrics for the specified font.
42	**boolean hitClip(int x, int y, int width, int height)** Returns true if the specified rectangular area might intersect the current clipping area.
43	**abstract void setClip(int x, int y, int width, int height)** Sets the current clip to the rectangle specified by the given coordinates.
44	**abstract void setClip(Shape clip)** Sets the current clipping area to an arbitrary clip shape.

45	**abstract void setColor(Color c)** Sets this graphics context's current color to the specified color.
46	**abstract void setFont(Font font)** Sets this graphics context's font to the specified font.
47	**abstract void setPaintMode()** Sets the paint mode of this graphics context to overwrite the destination with this graphics context's current color.
48	**abstract void setXORMode(Color c1)** Sets the paint mode of this graphics context to alternate between this graphics context's current color and the new specified color.
49	**String toString()** Returns a String object representing this Graphics object's value.
50	**abstract void translate(int x, int y)** Translates the origin of the graphics context to the point (x, y) in the current coordinate system.

DRAWING LINES:

Lines are drawn by means of the drawLine() method.

Syntax: `drawLine(int x1, int y1, int x2, int y2)`

Parameters: The drawLine method takes four arguments:

- **x1** – It takes the first point's x coordinate.
- **y1** – It takes first point's y coordinate.
- **x2** – It takes second point's x coordinate.
- **y2** – It takes second point's y coordinate

DRAWING RECTANGLES:

The drawRect() and fillRect() methods display an outlined and filled rectangle, respectively.

Syntax:
void drawRect(int top, int left, int width, int height)
void fillRect(int top, int left, int width, int height)

The upper-left corner of the rectangle is at(top, left). The dimensions of the rectangle are specified by width and height.
Use drawRoundRect() or fillRoundRect() to draw a rounded rectangle. A rounded rectangle has rounded corners.

DRAWING ELLIPSES & CIRCLE:

The Graphics class does not contain any method for circles or ellipses. To draw an ellipse, use drawOval(). To fill an ellipse, use fillOval().

Syntax
void drawOval(int top, int left, int width, int height)
void fillOval(int top, int left, int width, int height)

The ellipse is drawn within a bounding rectangle whose upper-left corner is specified by (top,left) and whose width and height are specified by width and height. To draw a circle, specify a square as the bounding rectangle i.e get height = width.

DRAWING ARCS:

An arc is a part of oval. Arcs can be drawn with draw Arc() and fillArc() methods.

Syntax
void drawArc(int top, int left, int width, int height, int startAngle, int sweetAngle)
void fillArc(int top, int left, int width, int height, int startAngle, int sweetAngle)

The arc is bounded by the rectangle whose upper-left corner is specified by (top,left) and whose width and height are specified by width and height. The arc is drawn from startAngle through the angular distance specified by sweepAngle. Angles are specified in degree. '0º' is on the horzontal, at the 30' clock's position. The arc is drawn conterclockwise if sweepAngle is positive, and clockwise if sweetAngle is negative.

DRAWING POLYGONS:

Polygons are shapes with many sides. It may be considered a set of lines connected together. The end of the first line is the beginning of the second line, the end of the second line is the beginning of the third line, and so on. Use drawPolygon() and fillPolygon() to draw arbitrarily shaped figures.

Syntax
void drawPolygon(int ,[] , int y[], int numPointer)
void fillPolygon(int ,[] , int y[], int numPointer)

The polygon's endpoints are specified by the coordinate pairs contained within the x and y arrays. The number of points defined by x and y is specified by numPoints.

It is obvious that x and y arrays should be of the same size and we must repeat the first point at the end of the array for closing the polygon.