What is Java ?

- **Java** is a programming language originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995 as a core component of Sun Microsystems' Java platform.
- The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities.
- Java applications are typically compiled to bytecode(class file) that can run on any Java Virtual Machine (JVM) regardless of computer architecture.
- Java is a general-purpose, concurrent, class-based, object-oriented language that is specifically designed to have as few implementation dependencies as possible.
- It is intended to let application developers "write once, run anywhere." Java is currently one of the most popular programming languages in use, particularly for client-server web applications.
- The original and reference implementation Java compilers, virtual machines, and class libraries were developed by Sun from 1995.
- As of May 2007, in compliance with the specifications of the Java Community Process, Sun relicensed most of its Java technologies under the GNU General Public License. Others have also developed alternative implementations of these Sun technologies, such as the GNU Compiler for Java and GNU Classpath.

Why Java

- **Object Oriented:** Object Oriented Programming (OOP) is the idea that a program can be broken up into "objects" and those objects have their own attributes and methods which define them.
- **Easy to Use:** The fundamentals of Java came from a programming language called C++. Although C++ is a powerful language, it was felt to be too complex in its syntax, and inadequate for all of Java's requirements. Java built on, and improved the ideas of C++, to provide a programming language that was powerful and simple to use.
- **Reliability:** Java needed to reduce the likelihood of fatal errors from programmer mistakes. With this in mind, object-oriented programming was introduced. Once data and its manipulation were packaged together in one place, it increased Java's robustness.
- **Secure:** Java does not use memory pointers explicitly. All the programs in java are run under an area known as the sand box. Security manager determines the accessibility options of a class like reading and writing a file to the local disk. Java uses the public key encryption system to allow the java applications to transmit over the internet in the secure encrypted form. The bytecode Verifier checks the classes after loading.
- **Robust:** Java has the strong memory allocation and automatic garbage collection mechanism. It provides the powerful exception handling and type checking mechanism as compare to other programming languages. Compiler checks the program whether there any error and interpreter checks any run time error and makes the system secure from crash. All of the above features makes the java language robust.

- **Platform Independent:** Programs needed to work regardless of the machine they were being executed on. Java was written to be a portable language that doesn't care about the operating system or the hardware of the computer.
- **Performance:** Java uses native code usage, and lightweight process called threads. In the beginning interpretation of bytecode resulted the performance slow but the advance version of JVM uses the adaptive and just in time compilation technique that improves the performance.
- **Multithreaded:** Multithreading means a single program having different threads executing independently at the same time. Multiple threads execute instructions according to the program code in a process or a program. Multithreading works the similar way as multiple processes run on one computer.
- **Interpreted:** Code is compiled to bytecodes that are interpreted by a Java virtual machines (JVM). This provides portability to any machine for which a virtual machine has been written. The two steps of compilation and interpretation allow for extensive code checking and improved security.

Flavors Of Java

Java comes in 3 editions

– J2SE Standard Edition (which you already know)

– J2EE Enterprise Edition

– J2ME Micro Edition

J2SE enterprise edition

- Complete environment for application execution
  - Stand-alone server applications
  - Stand-alone client applications
  - Stand-alone client-server applications
  - Applets
  - Web-start applications - rich applications deployed via Web
- Considered 'core' to all editions

J2EE standard edition

- Extension of Java SE
- Uses Java SE run-time environment
- Targeted at enterprise applications; applications that span all areas of an enterprise
  - From customer to back-office
  - From web to legacy
- Enables distributed multi-tier solutions

J2ME Micro Edition

- Targeted at consumer and embedded market;
- constrained devices
- Two major categories:
  - Connected Device Configuration (CDC)
  - Connected Limited Device Configuration (CLDC)

Major release versions of Java, along with their release dates:

- JDK 1.0 (January 23, 1996)
- JDK 1.1 (February 19, 1997)
- J2SE 1.2 (December 8, 1998)
- J2SE 1.3 (May 8, 2000)
- J2SE 1.4 (February 6, 2002)

- J2SE 5.0 (September 30, 2004)
- Java SE 6 (December 11, 2006)
- Java SE 7 (July 28, 2011)

Features Of Java

- **Java Virtual Machine (JVM):** It is an imaginary machine that is implemented by emulating software on a real machine. Provides the hardware platform specifications to which you compile all Java technology code.

- **Bytecode:** It is a special machine language that can be understood by the Java Virtual Machine (JVM). It is independent of any particular computer hardware, so any computer with a Java interpreter can execute the compiled Java program, no matter what type of computer the program was compiled on.

- **Garbage collection thread**: It is responsible for freeing any memory that can be freed. This happens automatically during the lifetime of the Java program. Programmer is freed from the burden of having to deallocate that memory themselves.

- **Code security:** Code security is attained in Java through the implementation of its Java Runtime Environment (JRE).

- **JRE:** It runs code compiled for a JVM and performs class loading (through the class loader), code verification (through the bytecode verifier) and finally code execution.

- **Class Loader**: It is responsible for loading all classes needed for the Java program. It adds security by separating the namespaces for the classes of the local file system from those that are imported from network sources. After loading all the classes, the memory layout of the executable is then determined. This adds protection against unauthorized access to restricted areas of the code since the memory layout is determined during runtime.

- **Bytecode verifier:** It tests the format of the code fragments and checks the code fragments for illegal code that can violate access rights to objects.

JVM (The Heart of Java)

- A Java virtual machine (JVM) is a virtual machine capable of executing Java bytecode. Sun Microsystems stated that there are over 4.5 billion JVM-enabled devices.
- A Java virtual machine is software that is implemented on non-virtual hardware and on standard operating systems. A JVM provides an environment in which Java bytecode can be executed, enabling such features as automated exception handling, which provides "root-cause" debugging information for every software error (exception), independent of the source code. A JVM is distributed along with a set of standard class libraries that implement the Java application programming interface (API). Appropriate APIs bundled together form the Java Runtime Environment (JRE).
- JVMs are available for many hardware and software platforms. The use of the same bytecode for all JVMs on all platforms allows Java to be described as a "compile once, run anywhere" programming language, as opposed to "write once, compile anywhere", which describes cross-platform compiled languages. Thus, the JVM is a crucial component of the Java platform.

- Java bytecode is an intermediate language which is typically compiled from Java, but it can also be compiled from other programming languages. For example, Ada source code can be compiled to Java bytecode and executed on a JVM.
- Oracle, the owner of Java, produces a JVM, but JVMs using the "Java" trademark may be developed by other companies as long as they adhere to the JVM specification published by Oracle and to related contractual obligations.
- The Oracle JVM is written in the C programming language.

The JVM has instructions for the following groups of tasks:

- Load and store
- Arithmetic
- Type conversion
- Object creation and manipulation
- Operand stack management (push / pop)
- Control transfer (branching)
- Method invocation and return
- Throwing exceptions
- Monitor-based concurrency

Java's Magic : The Byte Code

- The key that allows java to solve both the security and the portability problems just described is that the output of a java compiler is not executable code. Rather, it is byte code, byte code is a highly optimized set of instructions designed to be executed by the java runtime system, which is called the java virtual machine (JVM). That is, in its standard form, the JVM is an interpreter for byte code, this may come as a bit of a surprise.
- Translating a java program into bytecode helps make it much easier ot run a program in a wide variety of environments. The reason is straightforward: only the JVM needs to be implemented for each platform, all interpret the same java bytecode. If a java program were compiled to native code, the different versions of the same program would have to exist for each type of CPU connected to the Internet. This is, of course, not a feasible solution. Thus, the interpretation of bytecode is the easiest way to create truly portable programs.
- The fact that a java program is interpreted also helps to make it secure. Because the execution of every java program is under the control of the JVM, the JVM can contain the program and prevent it from generating side effects outside of the system. As you will see, safety is also enhanced by certain restrictions that exist in the java language.

The Java Language Specification (JLS)

- This specification describes all the aspects of the java programming language including the semantics, statements, and expressions, threads and a lot more.
- Written by the inventor of java technology
    - James Gosling
    - Bill Joy
    - Guy Steele
    - Gilad Bracha

What is Platform ?
A platform is the hardware and software environment in which a program runs.

Java Platform

- Java Platform, Enterprise Edition or Java EE is a widely used platform for server programming in the Java programming language.
- The Java platform differs from the Java Standard Edition Platform in that it adds libraries which provide functionality to deploy fault-tolerant, distributed, multi-tier Java software, based largely on modular components running on an application server.
- Java Virtual Machine (**JVM**)
- Java Programming Interface (**Java API**)
    - It is a large collection of ready-made software components that provides many useful capabilities.
    - It is grouped into libraries of related classes and interfaces. These libraries are known as **Packages**.

Programming in Java

You can use Java to create two types of programs:

- **Applications**
    - An application is a program that you can execute at the command prompt. Stand-alone applications can either be window-based applications or console applications.
- **Applets**
    - Applets are Java programs that execute within a Web page. Therefore, unlike applications, applets require a Java-enabled browser like Microsoft Internet Explorer 4.0 or later version. An applet is loaded and executed when a user loads a Web page, which contains the applet, through a Web browser.

Java SE Development Kit (JDK)

Development Tools
- Javac
- Java
- Javadoc

- Application Programming Interface
- Deployment Technology

    - JavaWebStart
    - Java Plug-in Software
- User Interface Toolkit
    - Swing
    - AWT (Abstract Windowing Toolkit)
    - Java 2D
- Integration Libraries
    - JDBC (Java Database Connectivity)
    - RMI (Remote Method Invocation)
    - JNI (Java Native Interface)
- Java Runtime Environment

Java Runtime Environment (JRE)

- The JRE contains libraries, java virtual machine and other resources required to enable java applications and applets to run.
- **JRE** = **JVM** + **Java Packages** Classes(like util, math, lang, awt,swing etc)+runtime libraries.

Java Architecture

The Java architecture consists of the following four components:

- Java programming language
- Java class file
- Java Application Programming Interface (Java API)
- Java Virtual Machine (JVM)

Java Tools

Executable	Tool Name	Description
appletviewer	Java applet viewer	Displays applets.
Java	Java interpreter	Runs Java bytecode.
Javac	Java compiler	Compiles Java programs into bytecode.
javadoc	Java documentation generator	Creates documentation in HTML format from Java source code.
Javah	Java header and stubs file generator	Creates C language header and stubs files from a Java class.
Javap	Java class file disassembler	Disassembles Java files and prints out a representation of Java bytecode.
Jdb	Java language debugger	Finds problems in your Java code.

A Simple Java Program

File Name: Program.java

```
//My first Java program
Class Program
{
        Public static void main(String args[])
        {
        System.out.println ("Welcome to LIT");
        }
}
```

To compile: javac Program.java
To Run : java Program

Java Program Execution:

- A JVM starts up by loading a specified class and then invoking the main method from the specified class, providing it a single parameter of array of string.
- A class must be **initialize** before it can be invoked.
- A class must be **linked** before it initialized.

Linking:

- ***Verification***
    - Well-formed
    - Proper Symbol Table
    - semantics
- ***Preparation***
    - Allocation of Static Storage
    - Create data structures required By JVM
- ***Resolution***
    - it is the process of checking symbolic references from a class to other classes and interface

Initialize:

- Class variables initalizers and static initalizers are executed.
- Before a class gets initializes its direct super class must be initialized.

Loading of Classes and Interfaces:

- Loading refers to the process of finding the binary form of a class or interface.
- The loading process is implemented by the class **ClassLoader**.

Just-In-Time(JIT) Compiler

- The Just-In-Time compiler is a component of Java Run-Time environment. It improves the performance of Java applications by compiling bytecodes to native machine code at run-time.
- It is also referred as dynamic compilation

Java Tokens

The smallest individual item in program is known as Tokens.
Java provides six different types of tokens. They are:

- Reserved Words (Keywords)
- Identifiers
- Literals
- Operators
- Separators
- Comments

KeyWords

Keywords are predefined, reserved identifiers that have special meanings to the compiler. They cannot be used as identifiers in your program.

abstract	continue	for	new	synchronized
assert	default	goto	package	switch
boolean	do	if	private	this
break	double	implements	protected	throw
byte	enum	import	public	throws
case	else	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Identifiers

- The word which is defined by the user is known as identifiers.
- Identifier may be name of the class, name of the method, name of the variable, object, namespace and interface.
- Rules for naming identifiers :
    - ❖ You cannot have keywords as names.
    - ❖ All identifiers have to start with an alphabet.
    - ❖ Except '_' and '$' no other special character is allowed.
    - ❖ Where there are multiple words in an identifier we usually capitalize the first alphabets of each word.

```
int    num1=20;
```

Literals

- Literals are java tokens containing set of characters. Literals are used to represent a constant that has to be stored in a variable.
- In Java, there five types of Literals. They are:
    - Integer Literals
    - Floating Point Literals
    - Character Literals
    - String Literals
    - Boolean Literals

Example:

1. x=10; // Integer Literal
2. x=12.3f // Floating Point Literal
3. x='a' // Character Literal
4. x="Welcome" // String Literal
5. x=true // Boolean Literal

Escape Sequence:

Notation	Character represented
\n	Newline (0x0a)
\r	Carriage return (0x0d)
\f	Formfeed (0x0c)
\b	Backspace (0x08)
\s	Space (0x20)
\t	tab
\"	Double quote
\'	Single quote
\\	backslash
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)

Operators in Java

- Operators are the symbols that specify which operations to perform in an expression.
- Operators are used in programs to manipulate data and variables.
- The different types of operators are:
    1. Assignment Operator
    2. Increment Decrement Operators
    3. Arithmetic Operators
    4. Bitwise Operators

5. Relational Operators
6. Logical Operators
7. Ternary Operator

1. Assignment Operator :

- The assignment operators are binary. The most basic is the operator '='. It assigns the value (or reference) of its second argument to its first argument.
- The first argument of the assignment operator is typically a variable where as the second argument may be a variable (having some value) or a value.

Ex: -

- a=10;
- a+=10;
- a-= 10;
- a*= 10;
- a /= 10
- a %=10;

2. Increment Decrement Operator

- Using this operator we can increase or decrease one value.
- It is 2 types : Pre & Post

Ex : a++;
++a;
a--;
--a;

3. Arithmetic Operators

- Arithmetic operators operate on numeric operands.

Ex : a+b, a-b, a*b, a/b, a%b .

4. Bitwise Operators

- Bitwise and bit shift operators are used to manipulate the contents of variable at a bit level according to binary format.

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in eather operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the efect of 'flipping' bits.	(~A) will give -60 which is 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000

>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>>	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

5. Relational Operators

- The binary relational operators are used for relational operations and for type comparision.

Ex:

- o a==b
- o a!=b
- o a<b
- o a>b
- o a<=b
- o a>=b

6. Logical Operators

- Logical operators operate on Boolean or integral operands.
- The Logical operators are :
  - o And(&&)
  - o Or(||)
  - o Not(!)

7. Ternary Operator

- Conditional operator consists of three operands and is used to evaluate boolean operations.
- The goal of the operator is to decide which value should be assigned to the variable.

Ex : c=(a>b)?a:b

Precedence of Java Operators

Category	Operator	Associativity
Postfix	() [] . (dot operator)	Left to right
Unary	++ - - ! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>> >>> <<	Left to right
Relational	> >= < <=	Left to right
Equality	== !=	Left to right

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

Separators

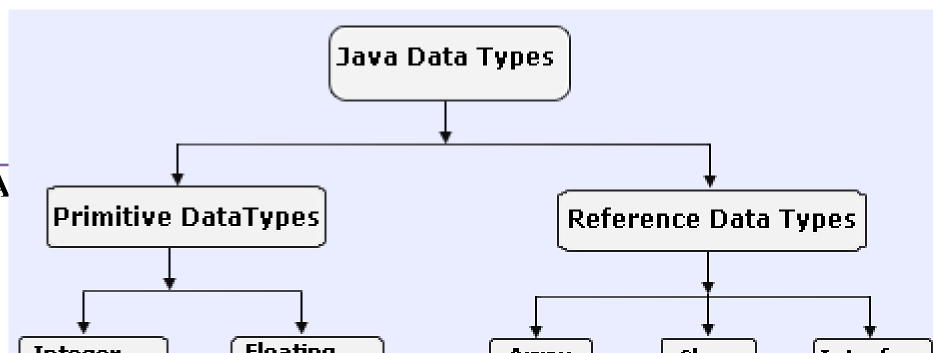
- Separators help define the structure of a program. The Java separators are
  - ()
  - {}
  - []
  - ;
  - ,
  - .
  - :

Comments

- Comments are used in programs to make the programmer feel convenient to understand the logic of the program.
- There are mainly two types of Comments are present in Java.
  - //………. (Single line comment)
  - /*………*/ (Multiple line comment)

Datatypes in Java

- Every variable in Java is associated with a data type.
- A variable is an identifier that denotes a storage location used to store a data value.
- Java is a strongly typed language; therefore every variable and object must have a declared type.
- Data types specify the size and type of values that can be stored.
- Proper utilization of correct data types allows developers to make the most of the language.
- There are two types of data types present in Java.
  - ❖ Primitive Data Types
  - ❖ Reference Data Types



Primitive Datatypes

- The primitive data types are **predefined** data types, which always hold the value of the same data type, and the values of a primitive data type don't share the **state** with other primitive values.
- These data types are named by a **reserved keyword** in Java programming language.
- There are **eight primitive data types** supported by Java programming language

  - Byte
  - Short
  - Integer
  - Long
  - Float
  - Double
  - Character
  - Boolean

Data Type	**Default Value (for fields)**	**Size (in bits/ Bytes)**	**Minimum Range**	**Maximum Range**
byte	0	1 Byte	-128	+127
short	0	2 Byte	-32768	+32767
int	0	4 Byte	-2147483648	+2147483647
long	0L	8 Byte	-9223372036854775808	+9223372036854775807
float	0.0f	4 Byte	1.40129846432481707e-45	3.40282346638528860e+38
double	0.0d	8 Byte	4.94065645841246544e-324 d	1.79769313486231570e+308d
char	'\u0000'	2 Byte		0 to 65,535
boolean	false	1 Bit	NA	NA

Decision Making and Looping

Control the flow of the Java Program

Types:

- Decision Control Structure
  - if … else
- Selection Control Structure
  - switch … case
- Jump Control Structure
  - Break
  - Continue
  - Return
- Repetition Control Structure
  - The **while** statement
  - The **do** statement
  - The **for** statement

Decision Control Structure (if … else)

- ☐ Checking rang – for Example "age between 15 and 25"
- ☐ Checking yes/no or true/false problem – for example "choice is 'yes' "

Example 1

```
class Test
{
        public static void main (String args[])
        {
                int age=20;
                if( age >= 15)
                        System.out.println("LIT");
                //      System.out.println("DotNet");   Hanging if Statement
                else
                        System.out.println ("Training");
        }
}
```

Example 2

```
import java.util.*;
class Test
{
        public static void Main (String args[])
        {
                int m1,m2,m3;
                float avg;
                Scanner s=new Scanner(System.in);
                System.out.println("Input mark for m1 m2 m3");
                m1= s.nextInt();
                m2= s.nextInt();
                m3= s.nextInt();
                avg=(m1+m2+m3)/3;
                if( avg  >= 60)
                        System.out.println("First");
                else if( avg >=50 )
                        System.out.println("Second");
        }
```

}

Selection Control Structure (switch … case)

Case :- This is the option to be selected or matched with the switch expression or value.

Default :- The default case is executed when none of the cases match with the value in switch.

Break :- Though it is not necessary for the switch case statement but it prevents unnecessary flowing of control to the statement subsequent to the case where a matched has been found the argument a switch statement must be a byte , char , int or short .

Example:

```
import java.util.*;
class xxx
{
	public static void main(String args[])
	{
		Scanner s=new Scanner(System.in);
		System.out.println("Enter a Number");
		int a=s.nextInt();
		switch(a)
		{
			case 1 :
				System.out.println("Mon");
				break;
			case 2 :
				System.out.println("Tue");
				break;
			case 3 :
				System.out.println("Wed");
				break;
			case 4 :
				System.out.println("Thurs");
				break;
			case 5 :
				System.out.println("Fri");
				break;
			case 6 :
				System.out.println("Sat");
				break;
			case 7 :
				System.out.println("Sun");
				break;
			default :
				System.out.println("Invalid Day");

		}
	}
}
```

Jump Control Structure

1. Break:

- It terminates the enclosing switch statement, and flow of control transfers to the statement immediately following the switch.
- This can also be used to terminate a for, while, or do-while loop

Example:

```
class xxx
{
	Public static void main(String args[])
	{
		String names[]={"Raja","Rama","Sita","Gita","Hari"};
		String searchName = "Sita";
		boolean foundName = false;
		for( int i=0; i< names.length; i++ )
		{
			if( names[i].equals( searchName ))
			{
				foundName = true;
				break;
			}
		}
		if( foundName )
			System.out.println( searchName + " found!" );
		else
			System.out.println( searchName + " not found." );
	}
}
```

2. Continue:

- It skips to the end of the innermost loop's body and evaluates the boolean expression that controls the loop, basically skipping the remainder of this iteration of the loop.

Example:

```
class xxx
{
	public static void main(String args[])
	{
		String names[] = {"Raja","Rama","Sita","Gita","Rama"};
		int count = 0;
		for( int i=0; i<names.length; i++ )
		{
			if( !names[i].equals("Rama") )
			{
				continue; //skip next statement
			}
		count++;
		}
	System.out.println("There are "+count+" Rama in the list");
	}
}
```

3. Return:

- It is used to exit from the current method.
- The flow of control returns to the statement that follows the original method call.

Repetition Control Structure

- The C# language provides for four constructs for performing loop operations. They are :
    - ☐ The **while** statement
    - ☐ The **do** statement
    - ☐ The **for** statement
- The statements are known as iteration or looping statements. We shall discuss the features and applications of each these statements in this chapter.
- They are used for repeating some parts of the Program.

The while statement

- ☐ The simplest of all the looping structures in Java is the **while** statement. The basic format of the **while** statement is

```
Initialization:
While (test condition)
{
        Body of the loop
}
```

- ☐ Suitable for unknown time

The do statement

- ☐ Suitable for at least once

```
Initialization:
do
{
        Body of the loop
}
while (test condition);
```

The for statement

- ☐ Suitable for finite times

```
for (initialization; test condition; increment/decrement)
{
        Body of the loop
}
```

Example 1(while):

```
class xxx
{
        public static void main(String args[])
        {
                int n1=1;
                int n2=1;
                while(n1<200)
                {
                        System.out.println(n1);
                        n2+=n1;
                        n1=n2-n1;
                }
        }
}
```

Example 2(do while):

```
class xxx
{
        public static void main(String args[])
        {
                int i=11;
                do
                {
                        System.out.println(i);
                        i++;
                }while(i<10);
        }
}
```

Example 3(For):

```
import java.util.*;
class xxx
{
        Public static void main(String args[])
        {
                Scanner s=new Scanner(System.in);
                int n;
                int sum=0;
                n=s.nextInt();
                for(int i=0;i<n;i++)
                {
                        sum=sum+n;
                }
                System.out.println(sum);
        }
}
```

Object Oriented Concepts

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

Class

- ☐ Class is specification/blueprint.
- ☐ Class is collection of similar type of object.
- ☐ Class as an abstract data type (ADT).
- ☐ Class is an abstract concept. It is simple a template.
- ☐ Class contains two types of Members.
  - Properties are called data Member.
  - Behaviors are called Method Member.

Object

- ☐ It is an instance of the class.
- ☐ It is a bundle of variables and related methods.
- ☐ Object is physical Memory is occupied by object creation.
- ☐ Object is variable of Class type.
- ☐ Many objects can be created from a single Class but each Object is unique.

Abstraction

- ☐ It is that process of the hiding unnecessary details and specifying the relevant things only.
- ☐ Class is an abstract data type as if follows abstraction.
- ☐ Abstraction can be
  - Data abstraction
  - Method abstraction
- ☐ Abstraction means simplifying the complex reality of a problem. In a lay man's language abstraction means "hiding the details".

Encapsulation:

- ☐ Encapsulation is binding of data and function in to a single entity called class.
- ☐ The objective of encapsulation is not to expose the internal details of the class to outside wall.
- ☐ Encapsulation enables data hiding.
- ☐ In object-oriented(OO) programming language data hiding is achieved by some access specifies like private, public, protected e.t.c.

Inheritance:

- ☐ It is the process acquiring data Members and Method Members of another Class.
- ☐ The Class from which properties and behaviors are inherited is base class/super class/parent class.
- ☐ The class to which Members are inherited is known as derived class/sub class/child class.

☐ Inheritance enables code reusability and extensibility.

☐ There are different types of inheritance.
- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Heretical Inheritance
- Hybrid Inheritance

Polymorphism (poly=many, morphism=ability to take from)

☐ Polymorphism is the ability of taking many forms.

☐ Polymorphism is usually of two types.
- Compile time polymorphism / overloading/static binding early blinding.
- Runtime polymorphism overriding / dynamic / blinding / late binding.

☐ In function overloading many functions can be defined having the same name but different signature. Signature consists of no of arguments and sequence of arguments.

☐ The compiler chooses the appropriate function depending on function call.

☐ In case of function overriding there exist many functions having same name and same signature but of different level (parent, child)

☐ The appropriate function is determined according to object at the run time.

Object and Object Reference:

Student s=new Student();

☐ Here s is not an object; it's a variable which contains a reference to an object. Objects don't have names, just types and locations in memory.

☐ In this example we create a new Student object in memory, and when created, assign a reference to that object to the Student variable s. s is a reference or object type variable which may reference a Student object or an object of any subclass of Student.

Object Lifetime and Garbage Collection

Object Lifetime:

☐ In computer science, the object lifetime (or life cycle) of an object in object-oriented programming is the time between an object's creation till the object is no longer used, and is destructed or freed.

☐ In object-oriented programming (OOP), the meaning of creating objects is far more subtle than simple allocating of spaces for variables. First, this is because, in the OOP paradigm, the lifetime of each object tends to vary more widely than in the case in conventional programming.

☐ There are many subtle questions, including whether the object be considered alive in the process of creation, and concerning the order of calling initializing code. In some sense, the creation can happen before the beginning of the program when objects are placed in a global scope.

In typical case, the process is as follows:

☐ Calculate the size of an object - the size is mostly the same as that of the class but can vary. When the object in question is not derived from a class, but from

a prototype instead, the size of an object is usually that of the internal data structure (a hash for instance) that holds its slots.

- Allocation - allocating memory space with the size of an object plus the growth later, if possible to know in advance
- Binding methods - this is usually either left to the class of the object, or is resolved at dispatch time, but nevertheless it is possible that some object models bind methods at creation time.
- Calling an initializing code (namely, *constructor*) of Superclass
- Calling an initializing code of class being created

Garbage Collection:

- Efficient memory management is essential in a runtime system. Storage for objects is allocated in a designated part of memory called the *heap*.
- The size of the heap is finite. Garbage collection is a process of managing the heap efficiently; that is, reclaiming memory occupied by objects that are no longer needed and making it available for new objects.
- Java provides automatic garbage collection, meaning that the runtime environment can take care of memory management concerning objects without the program having to take any special action.
- Storage allocated on the heap through the new operator is administered by the automatic garbage collector.
- The automatic garbage collection scheme guarantees that a reference to an object is always valid while the object is needed in the program. The object will not be reclaimed, leaving the reference dangling.

Creating and Operating Objects

- Before a Java object can be created the class byte code must be loaded from the file system (with .class extension) to memory.
- This process of locating the byte code for a given class name and converting that code into a Java Class class instance is known as class loading. There is one Class created for each type of Java class.
- All objects in java programs are created on heap memory. An object is created based on its class.
- When an object is created, memory is allocated to hold the object properties. An object reference pointing to that memory location is also created.
- To use the object in the future, that object reference has to be stored as a local variable or as an object member variable.
- The Java Virtual Machine (JVM) keeps track of the usage of object references. If there is no more reference to the object, the object cannot be used anymore and becomes garbage.
- After a while the heap memory will be full of unused objects. The JVM collects those garbage objects and frees the memory they allocated, so the memory can be reused again when a new object is created.
- **Ex: -**

```
import java.util.*;
class Calculate
{
```

```
        int add(int a,int b)
        {
                int c=a+b;
                return c;
        }
        int sub(int a,int b)
        {
                int c=a-b;
                return c;
        }
}
class xxx
{
        public static void main(String args[])
        {
                Scanner s=new Scanner(System.in);
                int a=s.nextInt();
                int b=s.nextInt();
                Calculate c=new Calculate();
                int p=c.add(a,b);
                int q=c.sub(a,b);
                System.out.println("Addition of two numbers are:"+p);
                Syatem.out.println("Substraction of two numbers are:"+q);
        }
}
```

Constructor

- Constructors are used to assign initial values to instance variables of the class.
- A default constructor with no arguments will be called automatically by the Java Virtual Machine (JVM).
- Constructor is always called by new operator.
- Constructors are declared just like as we declare methods, except that the constructors don't have any return type.
- Constructor can be overloaded provided they should have different arguments because JVM differentiates constructors on the basis of arguments passed in the constructor.

Types of constructors :

1. **Default Constructor:**Default constructors define the actions to be performed by the compiler when a class object is instantiated without actual parameter.
2. **Parameterized constructors:**The constructors that can take arguments are termed as parameterized constructors.

   For example: class example

```
{
    int p, q;
    example(int a, int b)
        {
            p = a;
            q = b;
        }
}
```

Access Modifiers

- ☐ The access to classes, constructors, methods and fields are regulated using access modifiers i.e. a class can control what information or data can be accessible by other classes.
- ☐ To take advantage of encapsulation, you should minimize access whenever possible.
- ☐ Java provides a number of access modifiers to help you set the level of access you want for classes as well as the fields, methods and constructors in your classes.
- ☐ A member has package or default accessibility when no accessibility modifier is specified.

Types of Access Modifiers

1. Private
2. Protected
3. Default
4. Public

Public:
Fields, methods and constructors declared public (least restrictive) within a public class are visible to any class in the Java program, whether these classes are in the same package or in another package.

Private:
The private (most restrictive) fields or methods cannot be used for classes and Interfaces. It also cannot be used for fields and methods within an interface. Fields, methods or constructors declared private are strictly controlled, which means they cannot be accesses by anywhere outside the enclosing class. A standard design strategy is to make all fields private and provide public getter methods for them.

Protected:
The protected fields or methods cannot be used for classes and Interfaces. It also cannot be used for fields and methods within an interface. Fields, methods and constructors declared protected in a super class can be accessed only by subclasses in other packages. Classes in the same package can also access protected fields, methods and constructors as well, even if they are not a subclass of the protected member's class.

Default:
Java provides a default specifier which is used when no access modifier is present. Any class, field, method or constructor that has no declared access modifier is accessible only by classes in the same package. The default modifier is not used for fields and methods within an interface.

Methods/Variables can be seen within:	Private	Public	Protected	Default
Same class	Yes	Yes	Yes	Yes
Any class in the same package	No	Yes	Yes	Yes
Subclass of the same package	No	Yes	Yes	Yes
Any class of a different package	No	Yes	No	No
Any subclass of a different package	No	Yes	Yes	No

Nested Class

☐ When a class is declared inside another class then that class is called as nested class.
☐ So the class defined inside another class is called as inner class.

A class can be nested

- Inside another class
- Inside a method

Nesting of class inside another class

```
public class OuterClass
{
    public class InnerClass
  {
    public void Show()
    {
      System.out.println( "Hi i am in inner class");
    }
  }
}
public class xxx
{
      public static void main(String args[])
      {
                OuterClass outobj=new OuterClass();
                OuterClass.InnerClass inobj=outobj.new InnerClass();
                inobj.Show();
      }
}
```

Nesting of class inside a Method:

A class which is defined within a block of code is called as local class.

```
public class OuterClass
{
public void Display()
{
    public class InnerClass
  {
    public void Show()
    {
      System.out.println( "Hi i am in inner class");
    }
  }
}
}
```

Anonymous class

- A kind of local class that does not have any name is called Anonymous Class.
- This class combines the steps of class definition, and object instantiation in a single java expression.
- As the class is instantiated in the same expression that defines it, it can be instantiated only once.
- Otherwise, anonymous classes are quite similar to local classes in terms of behaviour and use.

Abstract class

- Java Abstract classes are used to declare common characteristics of subclasses.
- An abstract class cannot be instantiated.
- It can only be used as a super class for other classes that extend the abstract class.
- Abstract classes are declared with the abstract keyword.
- Abstract classes are used to provide a template or design for concrete subclasses down the inheritance tree.
- Like any other class, an abstract class can contain fields that describe the characteristics and methods that describe the actions that a class can perform.
- An abstract class can include methods that contain no implementation. These are called abstract methods.
- The abstract method declaration must then end with a semicolon rather than a block.
- If a class has any abstract methods, whether declared or inherited, the entire class must be declared abstract.
- Abstract methods are used to provide a template for the classes that inherit the abstract methods.

```
abstract class Shape
{
    public String color;
    public Shape() {
    }
    public void setColor(String c) {
            color = c;
    }
    public String getColor() {
            return color;
    }
    abstract public double area();
}
public class Point extends Shape
```

```
{
    static int x, y;
    public Point() {
        x = 0;
        y = 0;
    }
    public double area() {
        return 0;
    }
    public double perimeter() {
        return 0;
    }
    public static void print() {
        System.out.println("point: " + x + "," + y);
    }
    public static void main(String args[]) {
        Point p = new Point();
        p.print();
    }
}
```

Interface

In Java, this multiple inheritance problem is solved with a powerful construct called **interfaces**. Interface can be used to define a generic template and then one or more abstract classes to define partial implementations of the interface. Interfaces just specify the method declaration (implicitly public and abstract) and can only contain fields (which are implicitly public static final). Interface definition begins with a keyword interface. An interface like that of an abstract class cannot be instantiated.

Multiple Inheritance is allowed when extending interfaces i.e. one interface can extend none, one or more interfaces. Java does not support multiple inheritance, but it allows you to extend one class and implement many interfaces.

If a class that implements an interface does not define all the methods of the interface, then it must be declared abstract and the method definitions must be provided by the subclass that extends the abstract class.

```
interface Shape
{
```

```
        public double area();
        public double volume();
}
public class Point implements Shape {

        static int x, y;
        public Point() {
                x = 0;
                y = 0;
        }
        public double area() {
                return 0;
        }
        public double volume() {
                return 0;
        }
        public static void print() {
                System.out.println("point: " + x + "," + y);
        }
        public static void main(String args[]) {
                Point p = new Point();
                p.print();
        }
}
```

Methods In Java

- A method is an operation on a particular object.
- An object is an instance of a class.
- When we define a class we define its member variables and its methods.
- For each method we need to give a name, we need to define its input parameters and we need to define its return type.
- We also need to set its visibility(private, package, or public). If the method throws an Exception, that needs to be declared as well.

Argument Passing Mechanism:

Arguments can be passed to a method by two ways.

1) Pass By Value
2) Pass By Reference

1) Pass By Value

In case of pass by value we pass value from one function to another.

2) Pass By Reference

In case of pass by reference we pass an object rather than any data.

```
Public class Point
{
        int x,y;
        public void tricky(Point arg1, Point arg2)
        {
                arg1.x = 100;
                arg1.y = 100;
                Point temp = arg1;
                arg1 = arg2;
```

```
            arg2 = temp;
        }
        public static void main(String [] args)
        {
            Point pnt1 = new Point(0,0);
            Point pnt2 = new Point(0,0);
            System.out.println("X: " + pnt1.x + " Y: " +pnt1.y);
            System.out.println("X: " + pnt2.x + " Y: " +pnt2.y);
            System.out.println(" ");
            tricky(pnt1,pnt2);
            System.out.println("X: " + pnt1.x + " Y:" + pnt1.y);
            System.out.println("X: " + pnt2.x + " Y: " +pnt2.y);
        }
}
```

Method Overloading

For the same class we can define two methods with the same name. However the parameter types and/or the number of parameters must be different for those two methods. In the java terminology, this is called **method overloading**. It is useful to use method overloading when we need to do something different based on a parameter type.

Method Overriding

If we define a method that exist in the super class then we override the super class method. The terminology for this is **method overriding**. This is different from method overloading. Method overloading happens with methods with the same name different signature. Method overriding happens with same name, same signature between inherited classes.

The return type can cause the same problem we saw above. When we override a super class method the return type also must be the same. In fact if that is not the same, the compiler will give you an error.

Method overriding is related dynamic linking, or runtime binding. In order for the Method Overriding to work, the method call that is going to be called can not be determined at compilation time.

Recursion

When a function calls itself then the concept is known as recursion.

```
int myFactorial( int integer)
{
if( integer == 1)
    return 1;
else
      {
        return(integer*(myFactorial(integer-1);
      }
}
```

Static Members of a class

When a member of a class is declared as static then no separate copies of that member is generated to individual objects. That’s why the static members are called from outside of the class by their respective class names instead of the objects.
A class can have two types of static members.
1) Static data member
2) Static member function

Example

class abc

```
{
    int a;
    static int b;
    void display()
    {
            System.out.println(a);
    }
    static void show()
    {
            System.out.println(b)
    }
    public  static  void main(String args[])
    {
            abc obj=new abc();
            obj.a=10;
            abc.b=20;
            obj.display();
            abc.show();
    }
}
```

Finalize()

- It is a special type of function which can be automatically called at the time of releasing of a particular object. We can also override that function using our own Finalize()

Native()

This method is used to create a C Header file using java code.

```
Ex:-
public class NativeDemo
{
        int i;
        public static void main(String args[])
        {
                NativeDemo ob = new NativeDemo();
                ob.i = 10;
                System.out.println("This is ob.i before the native method:" + ob.i);
                ob.test();
                System.out.println("This is ob.i after the native method:" + ob.i);
        }
        public native void test() ;
        static
        {
                System.loadLibrary("NativeDemo");
        }
}
How to compile a native method:
Javah –jni  NativeDemo
```

Inheritance

- ☐ It is the process acquiring data Members and Method Members of another Class.
- ☐ The Class from which properties and behaviors are inherited is base class/super class/parent class.
- ☐ The class to which Members are inherited is known as derived class/sub class/child class.
- ☐ Inheritance enables code reusability and extensibility.

Advantages of Inheritance

- ☐ Code reusability i.e. it facilitates classes to reuse the existing code.
- ☐ The new class acquires those members of the old class that are already tested and debugged.
- ☐ Saves time and increases Reliability.

Types of Inheritance

Inheritance allows the creation of a logical relationship between two or more classes. Depending on the number of classes involved and the way the classes are inherited, inheritance may take different forms, namely, single inheritance, multilevel inheritance, multiple inheritance, hierarchical inheritance, hybrid inheritance.

1. Single Inheritance:

In single inheritance, a class is derived from a single super class.

Ex:

```
class A
{
    protected void show()
    {
            System.out.println("I am in class A");
    }
}
class B extends A
{
    void display()
    {
            System.out.println("I am in class B");
    }
}
class C
{
    public static void main(String args[])
    {
            B obj=new B();
            obj.show();
            obj.display();
    }
}
```

2. Multilevel Inheritance:

In multilevel inheritance, one class is inherited from another class, which in turn is inherited from some other class. Multilevel inheritance comprises two or more levels. The sub class has all the properties of its direct super class as well as its indirect super class. This is known as the transitive nature of multilevel inheritance.

Ex:

```
class A
{
    protected void show()
    {
            System.out.println("I am in class A");
    }
}
class B extends A
{
    protected void display()
    {
            System.out.println("I am in class B");
    }
}
class C extends B
{
    void out()
    {
            System.out.println("I am in class C");
    }
}
class D
{
    public static void main(String args[])
    {
            C obj=new C();
            obj.show();
            obj.display();
            obj.out();
    }
}
```

3. Multiple Inheritance:

In multiple inheritance, a class inherits properties from more than one super class. In java, it is not possible to implement multiple inheritance directly. However there is a concept known as interface through which multiple inheritance can be implemented.

Ex:

```
interface A
{
    protected void show()
    {}
}
```

```
interfece B
{
    protected void display()
    {}
}
class C implements A,B
{
    void show()
    {
            System.out.println("Hi");
    }
    void display()
    {
            System.out.println("How r u ?");
    }

}
class D
{
    public static void main(String args[])
    {
            C obj=new C();
            obj.show();
            obj.display();
    }
}
```

4. Hierarchical Inheritance:

In hierarchical inheritance, more than one class is derived from a single super class. In this inheritance, super class includes all the properties that are common to all of its sub class.

Ex:

```
class A
{
    protected void show()
    {
            System.out.println("I am in class A");
    }
}
class B extends A
{
    protected void display()
    {
            System.out.println("I am in class B");
    }
}
class C extends A
{
```

```
    void out()
    {
            System.out.println("I am in class C");
    }
}
class D
{
    public static void main(String args[])
    {
            B ob1=new B();
            ob1.show();
            ob1.display();
            C ob2=new C();
            ob2.show();
            ob2.out();
    }
}
```

5. Hybrid Inheritance:

It is the combination of both multilevel and multiple inheritance.

Ex:

```
interfece A
{
    protected void show()
    {}
}
class B implements A
{
    void display()
    {
            System.out.println("Hi");
    }
    void show()
    {
            System.out.println("How are you?");

    }
}
class C
{
    void out()
    {
            System.out.println("I am fine");
    }
}
Class D
{
    B ob1=new B();
```

```
        ob1.show();
        ob1.display();
        C ob2=new C();
        ob2.out();
    }
```

Super():

- Super() is a special type of function through which the constructor of a base class can be called from the derived class constructor.

Ex:

```
class abc
{
    int a;
    abc(int x)
    {
        a=x;
    }
    void display()
    {
        System.out.println(a);
    }
}
class pqr extends abc
{
    int b,c;
    pqr(int x,int y)
    {
        super(x);
        b=y;
    }
    void sum()
    {
        c=a+b;
        System.out.println(c);
    }
}
class xyz
{
    public static void main(String args[])
    {
        abc ob1=new abc();
        ob1.display();
        pqr ob2=new pqr();
        ob2.sum();
    }
}
```

Error

There are mainly two types of errors.

- Compile time error
- Run time error

- ☐ The compile time errors occur due to syntactical errors like missing semicolons, missing brackets, miss spelt keywords, use of undeclared variables and so on.
- ☐ These errors can be detected and removed by exhaustive testing and debugging.
- ☐ The run time errors may occur due to problems in arithmetic calculations like dividing by zero, trying to access an out-of-bound array element, converting invalid string to a number and so on.
- ☐ These run time errors are usually referred as exceptions.

Exception

- ☐ Exception is an unpredicted event that occurs while the program is executing and thus disrupts the normal flow of the program or terminates the program abnormally.
- ☐ When an error occurs within a method, Java run time system creates an exception object.
- ☐ This exception object contains information about the exception which includes its type and the state of the program when the error occurred.
- ☐ Once an exception object is created, it is thrown and the runtime system searches for a method that contains an appropriate exception handler.
- ☐ An exception handler is considered appropriate if the type of exception handled by the handler is same as the type of exception thrown.

Example of an exception:

```
class xxx
{
        public static void main(String args[])
        {
                int a=10;
                int b=0;
                int c=a/b;                      // Exceptin occured
                System.out.println(c);
        }
}
```

Types of exceptions

Exception	Description
ArithmeticException	Thrown when arithmetic error occurs in the program such as divide-by-zero.
NullPointerException	Thrown when the user tries to use an object without initializing the object.
IOException	Thrown when error occurs during input/output of data.
ArrayIndexOutOfBoundsException	Thrown when an attempt is made to access an array element with invalid index value.
ArrayStoreException	Thrown when an attempt is made to store the incompatible type of data in an array.

IllegelAccessException	Thrown when an illegal attempt is made to access a class.
NumberFormatException	Thrown when an invalid conversion of a string to a numeric format takes place.
StringIndexOutOfBoundsException	Thrown when an attempt is made to access a string element that is beyond the index of the string.
IllegalArgumentException	Thrown when an illegal argument is used to invoke a method.
NegativeArraySizeException	Thrown when an array of negative size is created.

Exception Handling

- Java exception handling mechanism detects the run time error and provides a way to separate the error handling code from the rest of the program.
- Java exception handling is governed by the following five keywords.
  - Try
  - Catch
  - Finally
  - Throw
  - Throws

- A set of statements that needs to be monitored for the exception is contained in the *try* block.
- Whenever an exception occurs within the *try* block, it is thrown. This passes the control to the *catch* block which handles the exception appropriately.
- Java run time system automatically throws system generated exception.
- If the user wants to throw the exception explicitly, *throw* keyword is used. Sometimes, a method may throw exceptions which it cannot handle; this must be specified by using *throws* keyword.
- The code which must be executed whenever an exception is thrown or not is kept within the *finally* block, which is optional.

Syntax of exception handling code using *try* and *catch*:

```
try
{
        //code that may cause an exception
}
catch(exception_type e)
{
        //code to handle the exception
}
```

Example of using *try catch* block

```
class xxx
{
        public static void main(String args[])
        {
                int a=10;
                int b=0;
                int c=0;
                try
```

```
                {
                        c=a/b;
                        System.out.println(c);
                }
                catch(ArithmeticException e)
                {
                        System.out.println("Exception occurred");
                        System.out.println("The value of b cannot be zero");
                }
                finally
                {
                        System.out.println("Thak you");
                }
        }
}
```

Multiple *catch* statements

It is not necessary that the code enclosed within the try block throws a single exception. In case, multiple exceptions are thrown within a try block, Java allows using multiple catch statements for handling all these exceptions. The syntax to define multiple catch block is

```
try
{
        //code that generates exception
}
catch(exception_type e1)
{
        // code to handle the exception
}
catch(exception_type e2)
{
        // code to handle the exception
}
catch(exception_type e3)
{
        // code to handle the exception
}
.
.
.
```

Example to demonstrate the concept of multiple *catch* statements

```
import java.util.*;
class xxx
{
        public static void main(String args[])
        {
                Scanner s=new Scanner(System.in);
                try
                {
                        int i=10;
                        int j=0;
```

```
            int array[]={1,2,3,4,5};
            int k=i/j;                        //Divide by zero exception
            System.out.println(array[7]);     //Array out of bound exception
        }
        catch(ArrayIndexOutOfBoundException a)
        {
            System.out.println(a.getMessage());
        }
        catch(ArithmeticException e)
        {
            System.out.println(e.getMessage());
        }
        finally
        {
            System.out.println("Thank you");
        }
    }
}
```

User defined Exceptions (Use of *throw*)

- The built-in exceptions provided by the Java platform are used to handle errors occurring in the program.
- But, sometimes the programmer wants to create his own application specific exceptions.
- User-Defined exceptions can be created simply by defining a subclass of *Exception-Class* and using the *throw* keyword.

```
class MyException extends Exception
{
    MyException(String s)
    {
        Super(s);
    }
}
class xxx
{
    public static void main(String args[])
    {
        int x,y,z;
        x=5;
        y=3;
        try
        {
            z=x/y;
            if(z<=2)
            {
                throw new MyException("Less than two");
            }
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
        }
    }
```

```
}
```

Using *throws* keyword

☐ Sometimes a method may throw exception which it cannot handle, this must be specified using *throws* keyword.

```
class MyException extends Exception
{
	MyException(String s)
	{
		super(s);
	}
}
class xxx
{
	public static int divide(int i, int j) throws MyException
	{
		if(j==0)
			throw new MyException("j should not be Zero");
		return(i/j);
	}
	public static void main(String args[])
	{
		int i=10;
		int j=0;
		try
		{
			System.out.println("Division is "+divide(i,j));
		}
		catch(MyException e)
		{
			System.out.println(e.getMessage());
		}
	}
}
```