| INTRODUCTION TO PROGRAMMING PARADIGMS & CORE LANGUAGE DESIGN ISSUES

Basics of Programming Languages

Programming Languages- The process of telling the computer what to do also known as coding.

Language Translator- A translator is a computer program that translates a program written in a given programming language into a functionally equivalent program in a different computer language, without losing the functional or logical structure of the original code.

Types of Language Translator

Compiler- A compiler is a computer program that transforms human readable complete code of computer program into the machine-readable code that a CPU can execute.

How compiler works: Source code à Compiler à Machine code à output

Interpreter- An interpreter is a computer program that reads the source code of another computer program and executes that program. Because it is interpreted line by line, it is a much slower way of running a program than one that has been compiled but is easier for learners because the program can be stopped, modified and rerun without time-consuming compiles.

How interpreter works: Source code -> Interpreter -> output

Assembler- Assembler converts code written in assembly language into machine language. It works same like interpreter and compiler. The assembler program takes each program statement in the code and generates a corresponding bit stream or pattern (a series of 0's and 1's of a given length).

Testing & Debugging

Testing- The tasks performed to determine the existence of defects

Debugging- The tasks performed to detect the exact location of defects. Defects are also called bugs or errors

Types of Error

Syntax Errors- They are caused by the code that somehow violates the rules of the language, easy to detect and fix errors.

Semantic Errors- Occur when a statement executes and has an effect not intended by the programmer, hard to detect during normal testing.

Run-Time Errors- Occur when the program is running and tries to do something that is against the rules.

| Introduction to Different Programming Paradigms |

A programming paradigm is a fundamental style of computer programming. Paradigms differ in the concepts and methods used to represent the elements of a program (such as objects, functions, variables, constraints). And also steps that comprise a computation (such as assignations, evaluation, continuations, data flows).

Programming paradigm is an approach to solve problem using some programming language or also we can say it is a method to solve a problem using tools and techniques that are available to us following some approach.

Programming paradigms are a way to classify programming languages based on their features. Languages can be classified into multiple paradigms. Some languages make it easy to write in some paradigms but not others.

There are basic two programming paradigms:

Imperative	Declarative
1.Structured languages	1.Logic
2.Procedural languages	2.Functional
3.OOP languages	3.Database processing approaches

One paradigm programming languages:

Some languages are designed to support one paradigm (Smalltalk support OOP, Haskell supports functional programming)

Multiple Paradigm languages:

These programming languages support multiple paradigms such as object, pascal, C++, Java, JavaScript, C#, PHP, Python, etc.

Imperative Programming Paradigm

Imperative programming is a paradigm of computer programming in which the program describes a sequence of steps that change the state of the computer.

It tells the computer "how" to accomplish a task.

Programs written this way often compile to binary executables that run more efficiently since all the CPU instructions are themselves imperative statements.

Programs written this way often compile to binary executables that run more efficiently since all the CPU instructions are themselves imperative statements. To make program simple to understand, statements have been grouped into sections using blocks.

Procedural and object-oriented programming languages belong to Imperative paradigms, e.g. C, C++, C#, PHP, Java and assembly language.

Declarative Programming Paradigm

In this programming paradigm, programmer defines what needs to be accomplished by the program rather how it needs to be implemented.

Imperative programming is like how you do something, and declarative programming is more like what you do.

Logic, functional and domain-specific languages belong to declarative

paradigm.

Declarative code focuses on building logic of software without actually describing its flow. Example would be HTML, XML, CSS, SQL, Prolog, Haskell and Lisp.

Structured Programming Paradigm

Programming with clean, goto-free, nested control structures.

Program is divided into many subprograms or functions.

Reusability and scalability is not possible.

Difficult to maintain since debugging is very difficult.

It is used for embedded system design, where objects are not so important but procedures are important.

Top-down design.

There is a separation between system data and processes.

Procedural Programming Paradigm

It comes under Imperative programming paradigm.

Program is list of instructions to tell computer to do specific task.

The focus is on processing, algorithm needed to perform specific task.

The procedure is given fist-class treatment and the data is given second-class treatment.

Larger program is divided into procedures.

Debugging is easy with subprograms.

One subprogram can call other.

It follows top-down approach.

Global and local variables are used.

Global variable's data can move freely in the entire program.

If global variable changes, its change should be reflected to all subprogram using it, otherwise error will occur.

Real world problems containing object like student management system and all cannot be handled using this paradigm.

Object-Oriented Programming Paradigm (Imperative Programming Paradigm)

It is designed to remove some drawbacks of POP.

OOP treats data as primary element, data is secure here.

OOPP has many objects as individual task or problem.

Data and functions are built around the object.

Data communicates with functions within an object.

Functions communicate outside the object.

In OOPP emphasis is on data rather than procedures.

It follows bottom-up approach design.

Scripting Programming Paradigm

Scripts: These are program codes written inside HTML page or these programs can be executed on the client side or server side.

Scripting language: It is a type of programming language in which we can write a code to control other software application.

It is light weight programming language.

Script is a sequence of commands written as plain text and run by an interpreter.

They are interpreted rather than compiled.

Executable file is not created.

Line by line interpretation of code.

It is used to create dynamic web-pages.

HTML + CSS -> static web- pages,

HTML + CSS + JavaScript -> dynamic web-page.

Logic (Rule-based) Programming Paradigm

Logic programming languages belong to Declarative programming paradigm.

It is knowledge based, whereas C, C++, Java are not knowledge based.

Problems are divided into objects in OOPP.

Objects are very different in Logic programming.

Objects in OOPP are data structures.

Objects in Logic PP are only things not data structures.

Maintenance is very easy since code is easy to read. E.g. John owns car.

Express program in the form of symbolic logic.

Use a logical inferencing process to produce results.

Prepositions: A logic statement that may or may not be true.

Symbolic logic:

Logic, which can be used for the basic needs of formal logic. -> Express Preposition -> Express relation between two prepositions.

Object representation: Object in preposition are represented by simple term like constants and variables.

Constant: A symbol that represents an object.

Variable: A symbol that represents different objects at different times.

Compound terms: Atomic prepositions consists of compound terms.

Functional Programming Paradigm

It is Programming with function calls that avoid any global state.

Data and functions are totally separated.

Many programming languages support functional PP.

Function will not refer any global variable.

It is self-intact code. It is not depending on data, which is passed to it.

State change or modification of variable is very easy

Functions are treated as first class. Functions can be considered as a variable.

Name, Scope and Binding

Name: Identifiers that allow us to refer to variables, constants, functions, types, operations, and so on

Binding: An association of a name with an object. A binding is an association between two things, such as a name and the thing it names.

Scope: The lifetime of a binding of a name to an object

The Scope and Lifetime of Variables

We can declare variables within any block.

Block is begun with an opening curly brace and ended by a closing curly brace.

1 block equal to 1 new scope in Java thus each time you start a new block, you are creating a new scope.

A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

Storage Management

Static- Given absolute address maintained throughout program execution.

Stack- Allocated and deallocated in LIFO manner. Applicable to subroutines.

Heap- Allocated and deallocated at arbitrary time. (Run time and costliest type allocation)

Type Systems

A type system consists of (1) a mechanism to define types and associate them with certain language constructs, and (2) a set of rules for type equivalence, type compatibility, and type inference. The constructs that must have types are precisely those that have values, or that can refer to objects that have values. These constructs include named constants, variables, record fields, parameters, and sometimes subroutines; literal constants and more complicated expressions containing these.

Type equivalence rules determine when the types of two values are the same. Type compatibility rules determine when a value of a given type can be used in a given context. Type inference rules define the type of an expression based on the types of its constituent parts or (sometimes) the surrounding context.

Type Checking

Type checking is the process of ensuring that a program obeys the language's type compatibility rules. A violation of the rules is known as a type clash. A language is said to be strongly typed if it prohibits, in a way that the language implementation can enforce, the application of any operation to any object that is not intended to support that operation. A language is said to be statically typed if it is strongly typed and type checking can be performed at compile time.

OOPP Java

public: Modifier/Access specifier

public,private,protected,default (keywords)

class: Keyword of java language(type of object)

Classname: Identifier decided by programmer

static: without creation of object for a class, methods/variables are accessed outside the current class

void: Method is returning empty data

main: Execution of program starts here

String [ ] args: Input taken in java is String form input and saved in args array

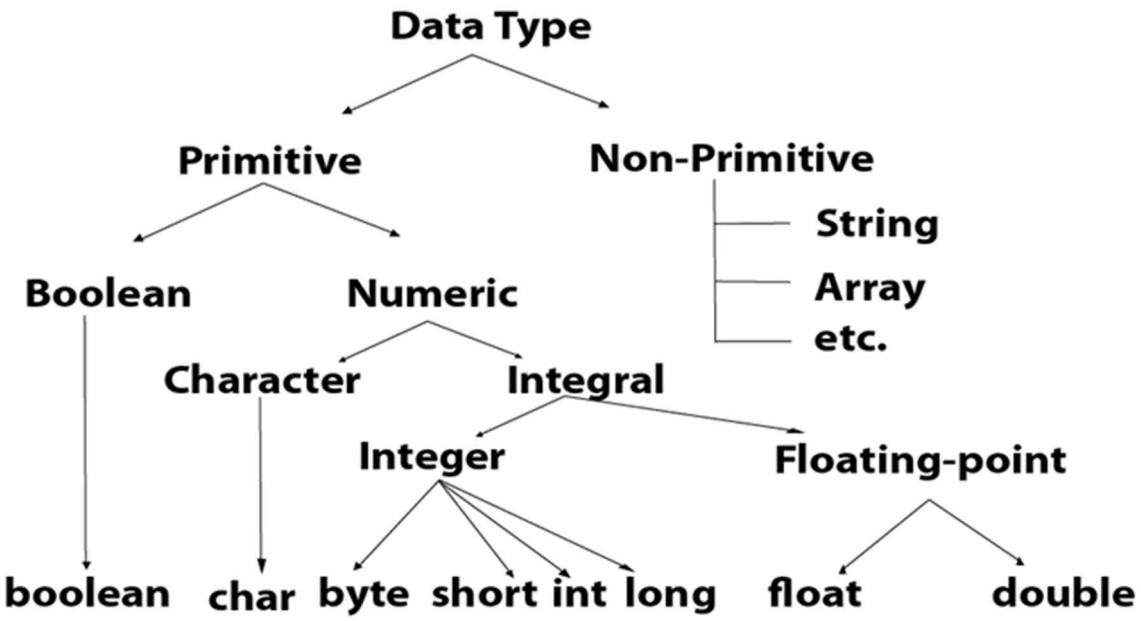
{….} : within opening and closing brackets, we write code.

println: It is the java statement to display output

javac: Compiler of java

java: Interpreter of java

Data Types in Java



Operators and operands in Java

Operator in Java is a symbol which is used to perform operations. For example: +, -, *, / etc.

Operators in Java are given below:

Unary Operator, Arithmetic Operator, Shift Operator, Relational Operator, Bitwise Operator, Logical Operator, Ternary Operator and Assignment Operator

ARRAYS IN JAVA

Java array is a derived data type 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.

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type with square brackets:

String[] arrayname;

To insert values to it, we can use an array literal - place the values in a comma-separated list, inside curly braces

String[] arrayname = {"value1 ", "value2", .. , "valueN"};

Array index

Used to Access the Elements of an Array.

You access an array element by referring to the index number.

The first value is found from index 0 and the last from length-1.

Array Bounds Checking

Whenever and array is accessed, the index is checked to ensure that it within the bounds of the array.

Attempts to access an array element outside the bounds of the array will cause an ArrayIndexOutOfBounds exception to be thrown.

COMMAND LINE ARGUMENT

Command line argument is user input from the command line

Argument array is initialized automatically for you

STRINGS IN JAVA

Every string we create is actually an object of type String.

String constants are actually String objects.

Objects of type String are immutable i.e. once a String object is created, its contents cannot be altered.

String handling is required to perform operations on Strings:

Some of the operations are –

1. charAt()
2. contains()
3. getChars()
4. indexOf()
5. replace()
6. toCharArray()
7. toLowerCase()
8. toUpperCase()

STRING CLASESS IN JAVA

In java, four predefined classes are provided that either represent strings or provide functionality to manipulate them.

Those classes are:

◦ String ◦ StringBuffer ◦ StringBuilder ◦ StringTokenizer

String, StringBuffer, and StringBuilder classes are defined in java.lang package and all are final. All three implement the CharSequence interface.

SUBROUTINE & CONTROL ABSTRACTION

Abstraction is a process by which the programmer can associate a name with a potentially complicated program fragment, which can then be thought of in terms of its purpose or function, rather than in terms of its implementation. We sometimes distinguish between control abstraction, in which the principal purpose of the abstraction is to perform a well-defined operation, and data abstraction, in which the principal purpose of the abstraction is to represent information.

Subroutines are the principal mechanism for control abstraction in most programming languages. A subroutine performs its operation on behalf of a caller, who waits for the subroutine to finish before continuing execution.

A subroutine that returns a value is usually called a function. A subroutine that does not return a value is usually called a procedure.

Stack Layout

Each instance of a subroutine at run time has its own frame (also called an activation record) on the stack, containing arguments and return values, local variables, temporaries, and bookkeeping information. Arguments to be passed to subsequent routines lie at the top of the frame, where the callee can easily find them.

When a subroutine returns, its frame is popped from the stack. At any given time, the stack pointer register contains the address of either the last used location at the top of the stack, or the first unused location, depending on convention. The frame pointer register contains an address within the frame. Objects in the frame are accessed via displacement addressing with respect to the frame pointer.

Calling Sequences

Maintenance of the subroutine call stack is the responsibility of the calling sequence—the code executed by the caller immediately before and after a subroutine call—and of the prologue (code executed at the beginning) and epilogue (code executed at the end) of the subroutine itself. Sometimes the term "calling sequence" is used to refer to the combined operations of the caller, the prologue, and the epilogue.

Tasks that must be accomplished on the way into a subroutine include passing parameters, saving the return address, changing the program counter, changing the stack pointer to allocate space, saving registers (including the frame pointer) that contain important values and that may be overwritten by the callee, changing the frame pointer to refer to the new frame, and executing initialization code for any objects in the new frame that require it. Tasks that must be accomplished on the way out include passing return parameters or function values, executing finalization code for any local objects that require it, deallocating the stack frame (restoring the stack pointer), restoring other saved registers (including the frame pointer), and restoring the program counter.

Parameter Passing

Most subroutines are parameterized: they take arguments that control certain aspects of their behaviour, or specify the data on which they are to operate. Parameter names that appear in the declaration of a subroutine are known as formal parameters. Variables and expressions that are passed to a subroutine in a particular call are known as actual parameters.

GENERIC SUBROUTINES AND MODULES

Subroutines provide a natural way to perform an operation for a variety of different object (parameter) values. In large programs, the need also often arises to perform an operation for a variety of different object types.

Generic modules or classes are particularly valuable for creating containers - data abstractions that hold a collection of objects, but whose operations are generally oblivious to the type of those objects.

Generic subroutines (methods) are needed in generic modules (classes), and may also be useful in their own right.

METHODS IN JAVA

A Java method is a collection of statements that are grouped together to perform an operation. When you call the System.out.println() method, for example, the system actually executes several statements in order to display a message on the console. Methods are time savers and help us to reuse the code without retyping the code.

Method definition consists of a method header and a method body.

modifier returnType nameOfMethod (Parameter List)

{ // method body }

In general, method declarations has six components :

Modifier-: Defines access type of the method i.e. from where it can be accessed in your application. In Java, there 4 type of the access specifiers(Modifiers).

public: accessible in all class in your application.

protected: accessible within the class in which it is defined and in its subclass(es)

private: accessible only within the class in which it is defined.

default (declared/defined without using any modifier) : accessible within same class and package within which its class is defined.

The return type : The data type of the value returned by the method or void if does not return a value.

Method Name : the rules for field names apply to method names as well, but the convention is a little different.

Parameter list : Comma separated list of the input parameters are defined, preceded with their data type, within the enclosed parenthesis. If there are no parameters, you must use empty parentheses ().

Method body : it is enclosed between braces. The code you need to be executed to perform your intended operations.

A method can also be called multiple times.

Java methods can be called many times with different parameters.

Java Methods are of four types:

Methods without return type and without parameters

Methods without return type and with parameters

Methods with return type and without parameters

Methods with return type and with parameters

In Java methods parameters accept arguments with three dots. These are known as ***variable arguments.***

Once you use variable arguments as a parameter method, while calling you can pass as many number of arguments to this method (variable number of arguments) or, you can simply call this method without passing any arguments. void demoMethod(String... args)

Java does not support the concept of default parameter however, you can achieve this using Method overloading: Using method overloading if you define method with no arguments along with parametrized methods. Then you can call a method with zero arguments.

final methods:

If a class is declared as final then by default all of the methods present in that class are automatically final but variables are not

RECURSION IN JAVA

Recursion in java is a process in which a method calls itself continuously. A method in java that calls itself is called recursive method. It makes the code compact but complex to understand.

Recursion should stop using terminating condition otherwise infinite call of the method and program will not stop.

Direct and indirect recursion

A function is called direct recursive if it calls the same function.

A function fun is called indirect recursive if it calls another function.

Advantages of recursion

Recursion provides a clean and simple way to write code.

Some problems are inherently recursive like tree traversals, Tower of Hanoi, etc. For such problems, it is preferred to write recursive code.

We can write such codes also iteratively with the help of a stack data structure.

Disadvantages of recursion

The recursive program has greater space requirements than iterative program as all functions will remain in the stack until the base case is reached.

It also has greater time requirements because of function calls and returns overhead.

EXCEPTIONS IN JAVA

An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e. at run time, that disrupts the normal flow of the program's instructions.

Error: An Error indicates serious problem that a reasonable application should not try to catch.

Exception: Exception indicates conditions that a reasonable application might try to catch.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
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 important 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 doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Checked Exception: The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.(javac file.java)

Unchecked Exception : The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

Error : Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError

EXCEPTION HANDLING IN JAVA

- In a method, there can be more than one statements that might throw exception, So put all these statements within its own try block and provide separate exception handler within own catch block for each of them.
- If an exception occurs within the try block, that exception is handled by the exception handler associated with it. To associate exception handler, we must put catch block after it. There can be more than one exception handlers.
- Each catch block is a exception handler that handles the exception of the type indicated by its argument. The argument, ExceptionType declares the type of the exception that it can handle and must be the name of the class that inherits from Throwable class.
- For each try block there can be zero or more catch blocks, but only one finally block. The finally block is optional. It always gets executed whether an exception occurred in try block or not. If exception occurs, then it will be executed after try and catch blocks. And if exception does not occur then it will be executed after the try block. The finally block in java is used to put important codes such as clean up code e.g. closing the file or closing the connection.

User defined exception

User Defined Exception or custom exception is creating your own exception class and throws that exception using 'throw' keyword. This can be done by extending the class Exception.

COROUTINES

A coroutine is represented by a closure (a code address and a referencing environment), into which we can jump by means of a nonlocal goto, in this case a special operation known as transfer. The principal difference between the two abstractions is that a continuation is a constant—it does not change once created—while a coroutine changes every time it runs.

Coroutines are execution contexts that exist concurrently, but that execute one at a time, and that transfer control to each other explicitly, by name. Coroutines can be used to implement iterators.

Coroutines are computer program components that generalize subroutines for non-preemptive multitasking, by allowing execution to be suspended and resumed. Coroutines are well-suited for implementing familiar program components such as cooperative tasks, exceptions, event loops, iterators, infinite lists and pipes.

Decision Making in Java

Java's Selection statements:

if	If-else-if
If-else	Switch-case
Nested if-else	jump – break, continue, return

if-else ladder Statement

User can decide among multiple options.

The if statements are executed from the top down.

As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed.

If none of the conditions is true, then the final else statement will be executed.

switch-case Statement

The switch statement is a multiway branch statement. It provides an easy way of execution of different parts of code based on the value of the expression.

Jump Statements

jump: Java supports three jump-statement: break, continue and return.

These three statements transfer control to other part of the program.

break In Java, break is majorly used for:

- Terminate a sequence in a switch statement (discussed above).
- To exit a loop.
- Used as a "civilized" form of go to.

The return statement is used to explicitly return from a method. That is, it causes a program control to transfer back to the caller of the method.

Continue Statement is used to skip the loop for a given condition and continue iterating again.

LOOPS IN JAVA

while loop - Entry level control structure

for loop

do while loop - Similar to while loop with only difference that it checks for condition after executing the statements, and therefore is an example of Exit Control Loop.

EVENTS

An event is something to which a running program (a process) needs to respond, but which occurs outside the program, at an unpredictable time. The most common events are inputs to a graphical user interface (GUI) system: keystrokes, mouse motions, button clicks. They may also be network operations or other asynchronous I/O activity: the arrival of a message, the completion of a previously requested disk operation.

A handler - a special subroutine is invoked when a given event occurs. Handlers are sometimes known as callback functions, because the run-time system calls back into the main program instead of being called from it. In an object-oriented language, the callback function may be a method of some handler object, rather than a static subroutine.

Output Statements in Java

System.out.println(); or

System.out.print(); or

System.out.printf();

System is a class

out is a public static field: it accepts output data.

Input Statement in Java

```
import java.util.Scanner;
Scanner input = new Scanner(System.in); // create an object of Scanner
int number = input.nextInt();   // take input from the user
```

WRAPPER CLASSES IN JAVA

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections.

Wrapper classes in Java convert primitive data types in Java objects.

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.

The java.util package provides the utility classes to deal with objects.

Autoboxing

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

```
int a=20;
Integer i=Integer.valueOf(a);//converting int into Integer explicitly
Integer j=a;//autoboxing
```

Unboxing

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

```
Integer a=new Integer(3);
int i = a.intValue();//converting Integer to int explicitly
int j = a;//unboxing
```

JAVA ARRAYS

Java Arrays class is a utility class to perform various common operations on arrays in java.

This class is part of java collections framework.

Java Arrays class contains methods for sorting, searching and comparing arrays.

Java Arrays provides ***asList method*** that returns a list made by the specified array.

```
String[] strings = {"one", "two", "three", "four", "five"};
// strings array is converted into a List
List<String> list = Arrays.asList(strings);
System.out.println(list);
```

Java Arrays provides ***sort*** method that sorts the element of specified array and also sorts the specified range of the given array into ascending order.

Sorting Specific Range of Integers

```
int[] ints = {5, 2, 1, 4, 3, 9, 6, 8, 7, 10};
int fromIndex = 2; int toIndex = 7;
Arrays.sort(ints, fromIndex, toIndex);
System.out.print("\nSorted Integers of Specific Range : ");
for (int i : ints) { System.out.print(i+" "); //Output: 5 2 1 3 4 6 9 8 7 10
```

Java Arrays binarySearch

Java Arrays binarySearch method uses binary search algorithm to search specified value from the elements of specified array and also searches from the specified range of the given array.

Searching a value from array of integer with specific range

```
int fromIndex = 2;

int toIndex = 7;

int index2 = Arrays.binarySearch(integers, fromIndex, toIndex, 9);

if (index2 >= 0)

{ System.out.println("Element is found at the index :" + index2); }

else

{ System.out.println("Element is not found"); }
```

Java Arrays equals

Java Arrays provides equals method that is used to compare two arrays of the same type and returns boolean result. It returns true if two given arrays are equal to one another.

```
int[] a1 = { 1, 2, 3 };

int[] a2 = { 1, 2, 3 };

boolean equal = Arrays.equals(a1, a2); //true
```

Java Arrays deepEquals

equals method is not able to compare nested arrays, for that Arrays have another method called deepEquals.

```
int[] a1 = { 1, 2, 3 };

int[] a2 = { 1, 2, 3 };

Object[] b1 = {a1};

Object[] b2 = {a2};

boolean equal = Arrays.deepEquals(b1, b2); //true
```

***

| IMPERATIVE PARADIGM: DATA ABSTRACTION IN OBJECT ORIENTATION

OBJECT ORIENTED PROGRAMMING IN JAVA

Java uses object-oriented programming paradigm

It is made up of object and classes

An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.

An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, Student etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

Characteristics of an object

State: represents the data (value) of an object. (Attributes)

Behavior: represents the behavior (functionality) of an object such as deposit, withdraw, etc. (Methods)

Identity: An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

Object Definitions:

- An object is a real-world entity.
- An object is a runtime entity.
- The object is an entity which has state and behaviour.
- The object is an instance of a class.

Class

A class is a group of objects which have common properties. It is a logical entity. It can't be physical.

A class in Java can contain:

- Fields(Data)
- Methods
- Constructors
- Blocks
- Nested class and interface

CORE CONCEPTS

Instance variable in Java

A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

Method in Java

In Java, a method is like a function which is used to expose the behavior of an object.

Advantage of Method

- Code Reusability
- Code Optimization

new keyword in Java

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

Initialization through reference

```
class Student{
 int id;
 String name;
}
class TestStudent2{
 public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name=”abc";
  System.out.println(s1.id+" "+s1.name);
 }
}
```

Initialization through method

```
class Student{
 private int id;
 private String name;
 public void setValue(int r, String n){
  id=r;
  name=n;
 }
 public void displayInformation(){System.out.println(rollno+" "+name);}
}
class TestStudent4{
 public static void main(String args[]){
  Student s1=new Student();  //s1 object is created
  Student s2=new Student();   //s2 object is created
  // s1.id=13; //error
  // System.out.println(s1.id};//error
  s1. setValue(111,"Karan");  //call of method
  s2. setValue(222,"Aryan");
  s1.displayInformation();
  s2.displayInformation();
 }
}
```

Anonymous object

Anonymous simply means nameless. An object which has no reference is known as an anonymous object. It can be used at the time of object creation only. If you have to use an object only once, an anonymous object is a good approach.

e.g. new Calculation().fact(5);

INITIALIZATION & FINALIZATION

Most object-oriented languages provide some sort of special mechanism to initialize an object automatically at the beginning of its lifetime. When written in the form of a subroutine, this mechanism is known as a constructor. Though the name might be thought to imply otherwise, a a constructor does not allocate space; it initializes space that has already been allocated. A few languages provide a similar destructor mechanism to finalize an object automatically at the end of its lifetime.

CONSTRUCTORS IN JAVA

A constructor is a special method that is used to initialize an object.

Every class has a constructor either implicitly or explicitly.

If we don't declare a constructor in the class then JVM builds a default constructor for that class.

A constructor has same name as the class name in which it is declared.

Constructor must have no explicit return type.

Constructor in Java cannot be abstract, static, final.

Syntax: className (parameter-list)

```
{
statements
}
```

- className is the name of class, as constructor name is same as class name.
- parameter-list is optional, because constructors can be parameterized and non-parameterize as well.

Types of Constructor

- Default Constructor
- Parameterized constructor

Each time a new object is created at least one constructor will be invoked.

Constructor chaining

It is a process of calling one constructor from another constructor with respect to current object.

'this' keyword is used to refer current object.

Constructor chaining is used when we want to perform multiple tasks by creating a single object of the class.

Following are the two ways to do constructor chaining

- Within same Class: It can be done using this() keyword.

- From Base class: By using super() keyword to call constructor from base class

Constructor chaining occurs through Inheritance. The main motive of class is to call constructor of its super class until it reaches the topmost class. This process is mainly used when we want to perform multiple tasks in single constructor rather than creating functions for each task in single constructor, we create separate constructor for each task and make their chain, making program more readable.

Private Constructors:

- In Java, we can create private constructor to prevent class being instantiate.

- It means by declaring a private constructor, it restricts to create object of that class.

- In private constructor, only one object can be created and the object is created within the class and also all the methods are static.

- An object cannot be created if a private constructor is present inside a class.

- A class which have a private constructor and all the methods are static then it is called Utility class

Copy Constructors

A Copy Constructor in Java is a special type of Constructor, which enables us to get a copy of an existing object. Copy Constructors can take only one parameter, which is a reference of the same class.

The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.

```
class Student{
    int id;
    String name;
    //constructor to initialize integer and string
    Student(int i,String n){
    id = i;
    name = n;
    }
    //constructor to initialize another object
    Student(Student s){
    id = s.id;
    name =s.name;
    }
    void display(){System.out.println(id+" "+name);}
    public static void main(String args[]){
    Student s1 = new Student(111,"Karan");
    Student s2 = new Student(s1);
    s1.display();
    s2.display();
   }
}
```

Constructor overloading

In constructor loading, we create multiple constructors with the same name but with different parameters types or with different no of parameters.

ENCAPSULATION AND INHERITANCE

Encapsulation mechanisms enable the programmer to group data and the subroutines that operate on them together in one place, and to hide irrelevant details from the users of an abstraction.

In object-oriented programming (OOP), encapsulation refers to the bundling of data with the methods that operate on that data, or the restricting of direct access to some of an object's components. Encapsulation is used to hide the values or state of a structured data object inside a class, preventing unauthorized parties' direct access to them.

Inheritance in Java

It is special feature of OOPP

One object inherits the state and behaviour of the parent object.

Using inheritance , one can create new classes that are built upon existing classes.

When class is inherited from an existing class, class can reuse methods and fields of the parent class.

Child class can add new methods and fields in it.

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

Inheritance is used for

- Method overriding
- Reusability of the code

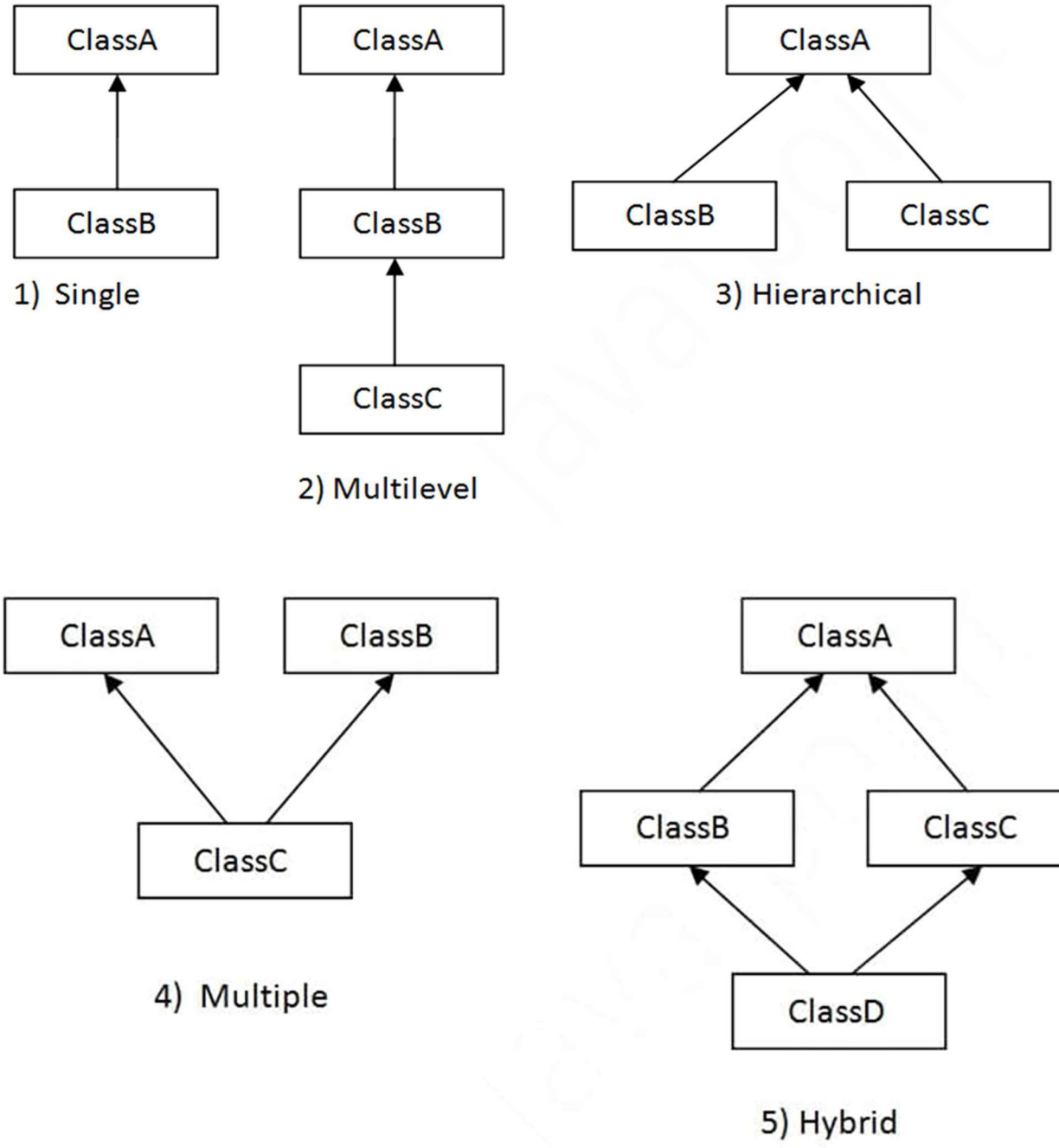
Class: A class is a group of objects which have common properties.

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.

TYPES OF INHERITANCE



Multiple inheritance is not supported by Java with class, it is possible with interface.

Method overloading

In method loading, we create multiple methods with the same name but with different data types of arguments or with different no of parameters.

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).

super Keyword in Java

super is used to refer immediate parent class instance variable.

super can be used to invoke parent class method

super is also used to invoke parent class constructor

POLYMORPHISM IN JAVA

Single action in different way

Poly-morphs = many-forms

Polymorphism Types : Compile time , Runtime

Overloading of static method is called as compile time polymorphism.

Run time polymorphism is also called as dynamic method dispatch.

Overriding of methods are used in run time polymorphism.

Run time polymorphism uses upcasting in inheritance.

Upcasting

```
class A{ }
class B extends A{ }
A a=new B();//upcasting
```

Run time polymorphism :

Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time. In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

```
class Bank{
float getRateOfInterest()  {return 0;}
}
class SBI extends Bank{
float getRateOfInterest()  {return 8.4f;}
}
class ICICI extends Bank{
float getRateOfInterest(). {return 7.3f;}
}
class AXIS extends Bank{
float getRateOfInterest(). {return 9.7f;}
}
class TestPolymorphism{
public static void main(String args[]){
Bank b;
b=new SBI();  //upcasting
System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
b=new ICICI();
```

```
System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());
b=new AXIS();
System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());
}
}
```

Runtime Polymorphism with Data Member

A method is overridden, not the data members. Runtime polymorphism can't be achieved by data members.

STATIC BINDING AND DYNAMIC BINDING

Connecting a method call to the method body is known as binding. There are two types of binding -

Static Binding (also known as Early Binding): Type of object determined at compile time

Dynamic Binding (also known as Late Binding) : Type of object determined at run time

Type of variable : int data=30;

Type of Reference: class Dog{

```
public static void main(String args[]){
Dog d1;//Here d1 is a type of Dog
}
}
```

Type of Object: class Animal{}

```
class Dog extends Animal{
public static void main(String args[]){
Dog d1=new Dog();
 }
}
```

static binding

```
class Dog{
 private void eat(){System.out.println("dog is eating...");}
 public static void main(String args[]){
  Dog d1=new Dog();
  d1.eat();  }  }
```

dynamic binding

```
class Animal{
 void eat(){System.out.println("animal is eating...");}
}
class Dog extends Animal{
 void eat(){System.out.println("dog is eating...");}
 public static void main(String args[]){
  Animal a=new Dog();
  a.eat();   }  }
```

In this example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal. So, compiler doesn't know its type, only its base type.

instanceof operator

```
class Animal{}
class Dog extends Animal{//Dog inherits Animal
 public static void main(String args[]){
 Dog d=new Dog();
 Animal a = new Animal();
 System.out.println(d instanceof Animal);//true
 System.out.println(a instanceof dog);// false
 }  }
```

Downcasting : In class-based programming, downcasting or type refinement is the act of casting a reference of a base class to one of its derived classes.

```
class Animal{}
class Dog extends Animal{
 public static void main(String args[]){
 Dog d=new Dog();
 Animal a = new Animal();
 Animal b= new Dog(); // auto – upcasting
 Dog c = new Animal(); // compile time error
 Dog c = (Dog)new Animal(); // downcasting
 System.out.println(d instanceof Animal);//true
 System.out.println(a instanceof Dog);// false
 }
}
```

ABSTRACT CLASS

Class declared with the keyword abstract is called as abstract class.

Abstract class has abstract and not abstract methods in it.

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Ways to achieve Abstraction : Abstract class (0 to 100%)

Interface (100%)

An abstract class must be declared with an abstract keyword.

It can have abstract and non-abstract methods.

It cannot be instantiated.

It can have constructors and static methods also.

It can have final methods which will force the subclass not to change the body of the method.

Abstract class: abstract class A{}

Abstract Method: abstract void prints();//no method body and abstract

If there is an abstract method in a class, that class must be abstract.

If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.

Example of an abstract class that has abstract and non-abstract methods

```
abstract class Bike{
   Bike(){System.out.println("bike is created");}
   abstract void run();
   void changeGear(){System.out.println("gear changed");}
 }

class Honda extends Bike{
 void run(){System.out.println("running safely..");}
 }

class TestAbstraction{
 public static void main(String args[]){
  Bike obj = new Honda();
  obj.run();
  obj.changeGear();
 }
}
```

```
bike is created
running safely..
gear changed
```

INTERFACE

An interface in Java is a type of unimplemented 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.

Interface also represents the IS-A relationship.

It cannot be instantiated just like the abstract class.

One can have default and static methods in an interface.

One can have private methods in an interface.

By interface, we can support the functionality of multiple inheritance.

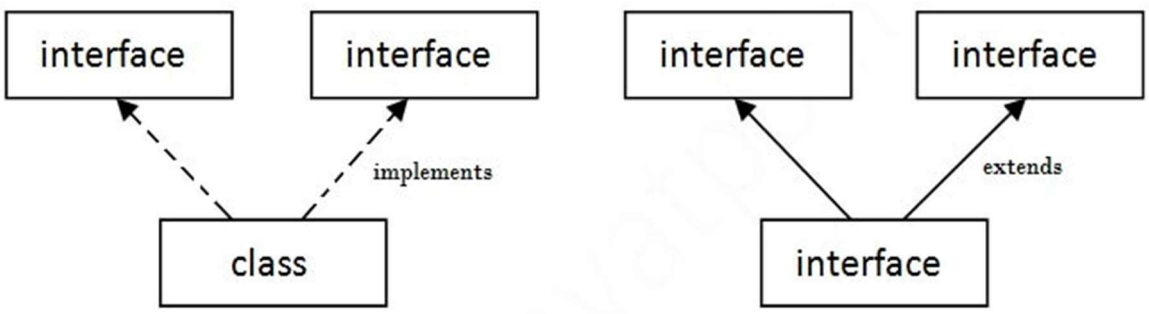
It can be used to achieve loose coupling.

```
interface <interface_name>
{
    // declare constant fields
    // declare methods that abstract
}
```

The Java compiler adds public and abstract keywords before the interface method.

It adds public, static and final keywords before data members

Multiple inheritance by using interface



Multiple inheritance is not supported through class in java, but it is possible by an interface because interface implementation is provided by the implementation class.

```
interface Printable{
void print();
}
interface Showable{
void print();
}
class TestInterface implements Printable, Showable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
TestInterface obj = new TestInterface();
obj.print();
 }
}
```

Interface inheritance

A class implements an interface, but one interface extends another interface.

```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
```

Default and statics method in Interface

```
interface Drawable{
void draw();
default void msg(){System.out.println("default method");}
static int cube(int x){return x*x*x;}
}
```

```
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class TestInterfaceDefault{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
d.msg();
System.out.println(Drawable.cube(3));
}}
drawing rectangle
default method
27
```

Difference between abstract class and interface

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance.**	Interface **supports multiple inheritance.**
3) Abstract class **can have final, non-final, static and non-static variables.**	Interface has **only static and final variables.**
4) Abstract class **can provide the implementation of interface.**	Interface **can't provide the implementation of abstract class.**
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.