INDUSTRIAL / SOFTWARE TRAINING
REPORT ON DSA WITH CORE JAVA

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR

SIX MONTH INDUSTRIAL TRAINING

at

UNDER THE TRAINING INSTITUDE

CODING BLOCK DELHI

SUBMITTED BY

Rohit Tiwari
Branch : ECE

Roll No. : 224/19

Univ. Roll No. : 1903781

Department of Electronics & Communication Engineering
DAV Institute of Engineering & Technology, JALANDHAR-144008

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my trainer and project mentor of Industrial training, Mr. LAKSHYA KUMAR providing his invaluable guidance comments, and suggestions throughout the course of the project. I would specially thank Dr. NEERU MALHOTRA (HOD, DEPT OF ECE), for constantly motivating me to work harder. During my Industrial training, the staff at CODING BLOCK Pvt ltd and the person guiding me were very helpful and extended their valuable guidance and help whenever required for the projects which I worked on.

we are extremely thankful to the entire ECE department for the mentorship we received.

CONTENTS

CERTIFICATE

INTRODUCTION

What is Java?

Java is a **programming language** and a **platform**. Java is a high level, robust, object-oriented and secure programming language.

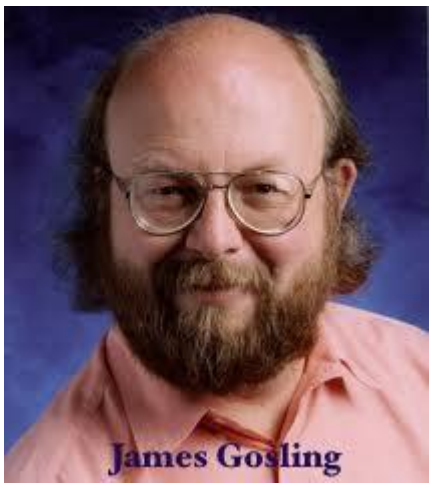
Java was developed by *Sun Microsystems* (which is now the subsidiary of Oracle) in the year 1995. *James Gosling* is known as the father of Java. Before Java, its name was *Oak*. Since Oak was already a registered company, so James Gosling and his team changed the name from Oak to Java.

Platform: Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime environment (JRE) and API, it is called a platform.

History of java

The history of Java is very interesting. Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time. The history of Java starts with the Green Team. Java team members (also known as Green Team), initiated this project to develop a language for digital devices such as set-top boxes, televisions, etc. However, it was best suited for internet programming. Later, Java technology was incorporated by Netscape.

The principles for creating Java programming were "Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted, and Dynamic". Java was developed by James Gosling, who is known as the father of Java, in 1995. James Gosling and his team members started the project in the early '90s.



Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc. Following are given significant points that describe the history of Java.

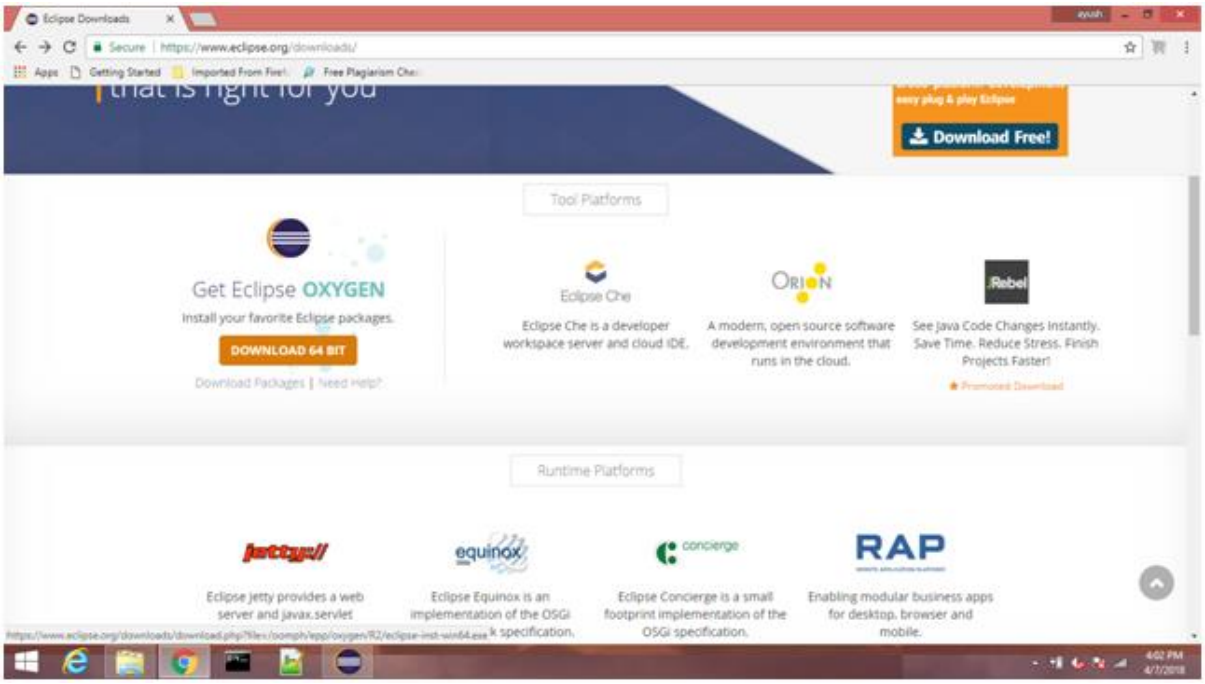
INSTALL ECLIPSE IDE

Install Eclipse

In order to run the JavaFX application, we need to set up eclipse. Follow the instructions given below to install the eclipse and configure to execute the JavaFX application.

Step 1: Download the Latest version

Click the link **Download Eclipse** to visit the download page of eclipse. You can download the latest version of eclipse i.e. eclipse oxygen from that page. The opened page will look like following, click on **DOWNLOAD 64 BIT** to proceed the download.

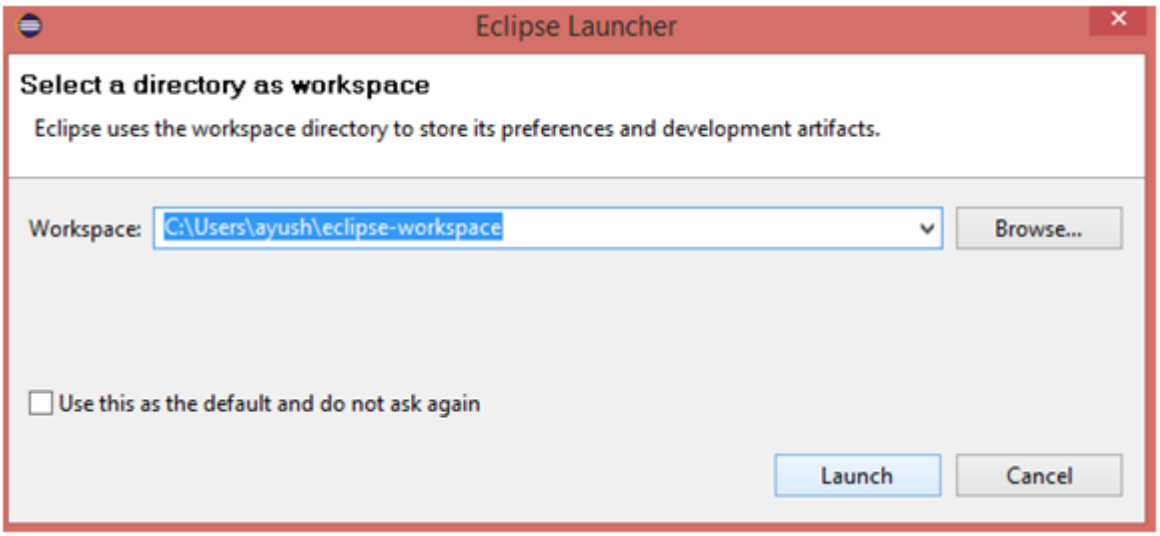


Step 2: Install Eclipse

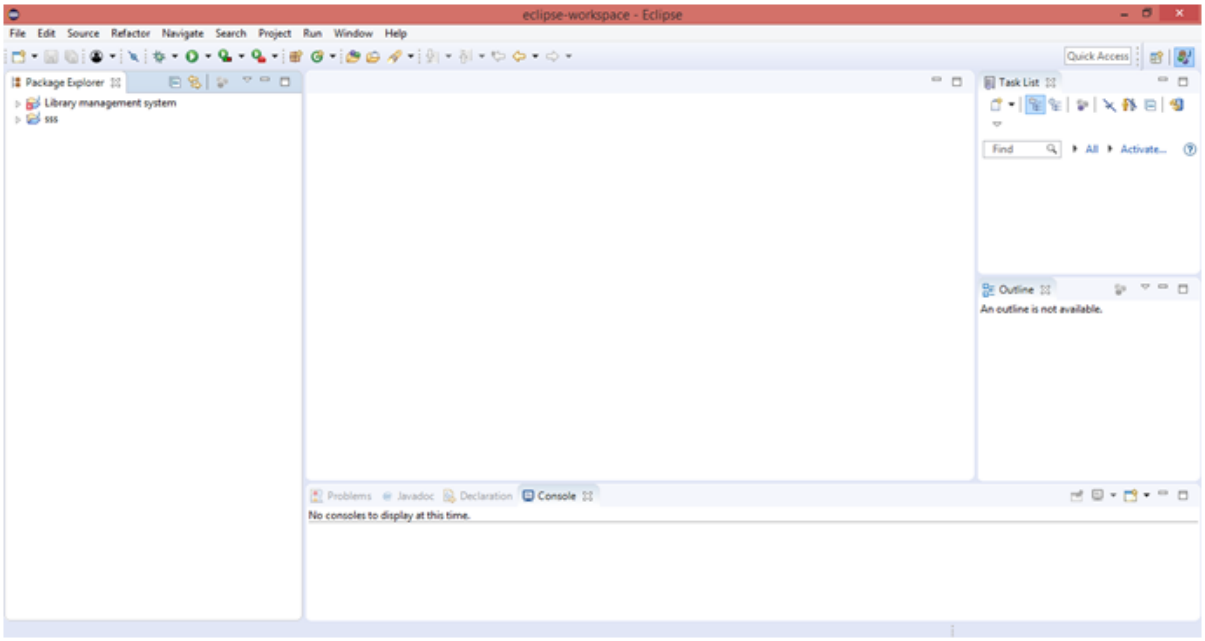
Double click on the **exe** file which has just been downloaded. The screen will look like following. Click **Run** to proceed the installation.

Choose the software suit which you want to install. In our case, we have chosen **Eclipse IDE for Java Developers** which is recommended in our case.

Now, the Set up is ready to install Eclipse oxygen 64 bit in the directory shown in the image. However, we can select any destination folder present on our system. Just click install when you done with the directory selection.



We have got the Eclipse IDE opened on our system. However,the screen will appear like following. Now, we are all set to configure Eclipse in order to run the JavaFX application.



JAVA TOOLS

JDK

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets. It physically exists. It contains JRE + development tools.

JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:

- Standard Edition Java Platform
- Enterprise Edition Java Platform
- Micro Edition Java Platform

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.

JRE

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

The implementation of JVM is also actively released by other companies besides Sun Micro Systems.

JVM

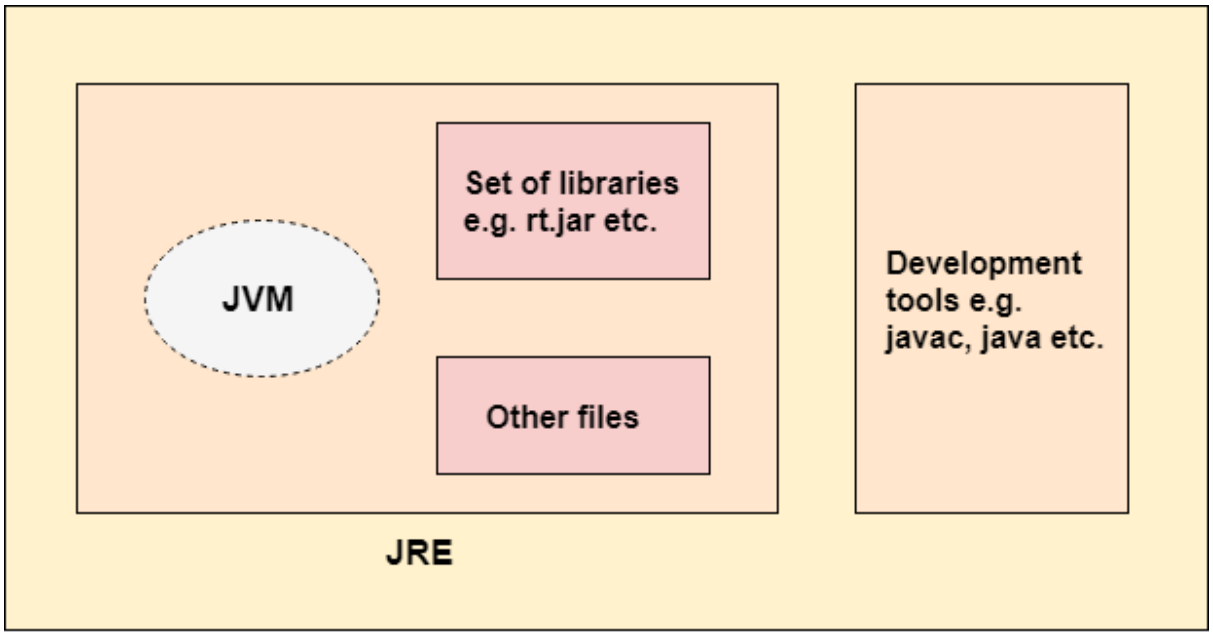
JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.

JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each OS is different from each other. However, Java is platform independent. There are three notions of the JVM: *specification*, *implementation*, and *instance*.

The JVM performs the following main tasks:

- Loads code
- Verifies code
- Executes code

- Provides runtime environment



VARIABLE AND DATA TYPES

What is Variables

A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type.

Variable is a name of memory location. There are three types of variables in java: local, instance and static.

There are two types of data types in Java: primitive and non-primitive.

Variable

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

Types of Variables

There are three types of variables in Java:

- local variable
- instance variable
- static variable

1) Local Variable

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

A local variable cannot be defined with "static" keyword.

2) Instance Variable

A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as static.

It is called an instance variable because its value is instance-specific and is not shared among instances.

3) Static variable

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

Example to understand the types of variables in java

```
public class A
 {
  static int m=100;
  void method()
  {
     int n=90;
  }
public static void main(String args[])
  {
     int data=50;
          }
```

Data Types in Java

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

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

Java Primitive Data Types

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

There are 8 types of primitive data types:

- boolean data type
- byte data type
- char data type
- short data type
- int data type
- long data type
- float data type

- double data type

Data Type	Default Value	Default size
Boolean	false	1 bit
Char	'\u0000'	2 byte
Byte	0	1 byte
Short	0	2 byte
Int	0	4 byte
Long	0L	8 byte
Float	0.0f	4 byte
Double	0.0d	8 byte

OBJECT ORINENTED PROGRAMING

In this page, we will learn about the basics of OOPs. Object-Oriented Programming is a paradigm that provides many concepts, such as **inheritance**, **data binding**, **polymorphism**, etc.

Simula is considered the first object-oriented programming language. The programming paradigm where everything is represented as an object is known as a truly object-oriented programming language.

Smalltalk is considered the first truly object-oriented programming language.

The popular object-oriented languages are Java, C#, PHP, Python, C++, etc.

What is OOPs (Object-Oriented Programming System)

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

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

Object

Any entity that has state and behaviour is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.
An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.
Example: A dog is an object because it has states like colour, name, breed, etc. as well as behaviours like wagging the tail, barking, eating, etc.

Class

Collection of objects is called class. It is a logical entity.

A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

Inheritance

When one object acquires all the properties and behaviours of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

If *one task is performed in different ways*, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.

In Java, we use method overloading and method overriding to achieve polymorphism.

Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example phone call, we don't know the internal processing.

In Java, we use abstract class and interface to achieve abstraction.

Encapsulation

Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines.

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Coupling

Coupling refers to the knowledge or information or dependency of another class. It arises when classes are aware of each other. If a class has the details information of another class, there is strong coupling. In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field. You can use interfaces for the weaker coupling because there is no concrete implementation.

Cohesion

Cohesion refers to the level of a component which performs a single well-defined task. A single well-defined task is done by a highly cohesive method. The weakly cohesive method will split the task into separate parts. The java.io package is a highly cohesive package because it has I/O related classes and interface. However, the java.util package is a weakly cohesive package because it has unrelated classes and interfaces.

Association

Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects:

- One to One
- One to Many
- Many to One, and
- Many to Many

Let's understand the relationship with real-time examples. For example, One country can have one prime minister (one to one), and a prime minister can have many ministers (one to many). Also, many MP's can have one prime minister (many to one), and many ministers can have many departments (many to many).

Association can be unidirectional or bidirectional.

Aggregation

Aggregation is a way to achieve Association. Aggregation represents the relationship where one object contains other objects as a part of its state. It represents the weak relationship between objects. It is also termed as a *has-a* relationship in Java. Like, inheritance represents the *is-a* relationship. It is another way to reuse objects.

Composition

The composition is also a way to achieve Association. The composition represents the relationship where one object contains other objects as a part of its state. There is a strong relationship between the containing object and the dependent object. It is the state where containing objects do not have an independent existence. If you delete the parent object, all the child objects will be deleted automatically.

Inheritance in Java

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

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

Terms used in Inheritance

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

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

```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
 }
}
```

OUTPUT

```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.

- Single-level inheritance.
- Multi-level Inheritance.
- Hierarchical Inheritance.
- Multiple Inheritance.
- Hybrid Inheritance.

Single Inheritance Example

When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

Multilevel Inheritance Example

When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

Hierarchical Inheritance Example

When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

Aggregation in Java

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Consider a situation, Employee object contains many informations such as id, name, emailId etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

Why use Aggregation?

- For Code Reusability.

Polymorphism

Method Overloading in Java

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behaviour of the method because its name differs.

Method Overriding in Java

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.

Abstraction in Java

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

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstract class in Java

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

Before learning the Java abstract class, let's understand the abstraction in Java first.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

Abstract Method in Java

A method which is declared as abstract and does not have implementation is known as an abstract method.

Example of Abstract class that has an abstract method

```
abstract class Bike{
  abstract void run();
}
class Honda4 extends Bike{
```

```
void run(){System.out.println("running safely");}
public static void main(String args[]){
 Bike obj = new Honda4();
 obj.run();
}
}
```

Encapsulation in Java

Encapsulation in Java is a *process of wrapping code and data together into a single unit*, for example, a capsule which is mixed of several medicines.

We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

ARRAY AND ARRAY LIST

ARRAY

Normally, an array is a collection of similar type of elements which has contiguous memory location.

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

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Example of Java Array

```
class Testarray{
public static void main(String args[]){
int a[]=new int[5];
a[0]=10;
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
for(int i=0;i<a.length;i++)
System.out.println(a[i]);
}
}
```

Output:

```
10
20
70
40
50
```

Java ArrayList

Java **ArrayList** class uses a *dynamic array* for storing the elements. It is like an array, but there is *no size limit*. We can add or remove elements anytime. So, it is much more flexible than the traditional array. It is found in the *java.util* package. It is like the Vector in C++.

The ArrayList in Java can have the duplicate elements also. It implements the List interface so we can use all the methods of the List interface here. The ArrayList maintains the insertion order internally.

It inherits the AbstractList class and implements List interface.

The important points about the Java ArrayList class are:

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because the array works on an index basis.
- In ArrayList, manipulation is a little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.
- We can not create an array list of the primitive types, such as int, float, char, etc. It is required to use the required wrapper class in such cases. For example:

ArrayList class declaration

```
public class ArrayList<E> extends AbstractList<E> implements List<E>
```

Methods of ArrayList

Method	Description
void add(int index, E element)	It is used to insert the specified element at the specified position in a list.
boolean add(E e)	It is used to append the specified element at the end of a list.
boolean addAll(Collection<? extends E> c)	It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
boolean addAll(int index, Collection<? extends E> c)	It is used to append all the elements in the specified collection, starting at the specified position of the list.
void clear()	It is used to remove all of the elements from this list.

void ensureCapacity(int requiredCapacity)	It is used to enhance the capacity of an ArrayList instance.
E get(int index)	It is used to fetch the element from the particular position of the list.
boolean isEmpty()	It returns true if the list is empty, otherwise false.
Iterator()	
listIterator()	
int lastIndexOf(Object o)	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
Object[] toArray()	It is used to return an array containing all of the elements in this list in the correct order.
<T> T[] toArray(T[] a)	It is used to return an array containing all of the elements in this list in the correct order.
Object clone()	It is used to return a shallow copy of an ArrayList.
boolean contains(Object o)	It returns true if the list contains the specified element.
int indexOf(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
E remove(int index)	It is used to remove the element present at the specified position in the list.
boolean remove(Object o)	It is used to remove the first occurrence of the specified element.
boolean removeAll(Collection<?> c)	It is used to remove all the elements from the list.

STACK AND QUEUE

Stack

The **stack** is a linear data structure that is used to store the collection of objects. It is based on **Last-In-First-Out** (LIFO). Java collection framework provides many interfaces and classes to store the collection of objects. One of them is the **Stack class** that provides different operations such as push, pop, search, etc.

In this section, we will discuss the **Java Stack class**, its **methods,** and **implement** the stack data structure in a Java program. But before moving to the Java Stack class have a quick view of how the stack works.

The stack data structure has the two most important operations that are **push** and **pop**. The push operation inserts an element into the stack and pop operation removes an element from the top of the stack. Let's see how they work on stack.

The following are the operations that can be performed on the stack:

- **push(x):** It is an operation in which the elements are inserted at the top of the stack. In the **push** function, we need to pass an element which we want to insert in a stack.
- **pop():** It is an operation in which the elements are deleted from the top of the stack. In the **pop()** function, we do not have to pass any argument.
- **peek()/top():** This function returns the value of the topmost element available in the stack. Like pop(), it returns the value of the topmost element but does not remove that element from the stack.
- **isEmpty():** If the stack is empty, then this function will return a true value or else it will return a false value.
- **isFull():** If the stack is full, then this function will return a true value or else it will return a false value.

In stack, the **top** is a pointer which is used to keep track of the last inserted element. To implement the stack, we should know the size of the stack. We need to allocate the memory to get the size of the stack. There are two ways to implement the stack:

- **Static:** The static implementation of the stack can be done with the help of arrays.
- **Dynamic:** The dynamic implementation of the stack can be done with the help of a linked list.

Java Stack Class

In Java, **Stack** is a class that falls under the Collection framework that extends the **Vector** class. It also implements interfaces **List, Collection, Iterable, Cloneable, Serializable.** It represents the LIFO stack of objects. Before using the Stack class, we must import the java.util package. The stack class arranged in the Collections framework hierarchy, as shown below.

Java Stack Class

In Java, **Stack** is a class that falls under the Collection framework that extends the **Vector** class. It also implements interfaces **List, Collection, Iterable, Cloneable, Serializable.** It represents the LIFO stack of objects. Before using the Stack class, we must import the java.util package. The stack class arranged in the Collections framework hierarchy, as shown below.

Creating a Stack

```
Stack<type> stk = new Stack<>();
```

Methods of the Stack Class

We can perform push, pop, peek and search operation on the stack. The Java Stack class provides mainly five methods to perform these operations. Along with this, it also provides all the methods of the Java Vector class.

Method	Modifier and Type	Method Description
empty()	boolean	The method checks the stack is empty or not.
push(E item)	E	The method pushes (insert) an element onto the top of the stack.
pop()	E	The method removes an element from the top of the stack and returns the same element as the value of that function.
peek()	E	The method looks at the top element of the stack without removing it.
search(Object o)	int	The method searches the specified object and returns the position of the object.

Java Queue

A queue is another kind of linear data structure that is used to store elements just like any other data structure but in a particular manner. In simple words, we can say that the queue is a type of data structure in the Java programming language that stores elements of the same kind. The components in a queue are stored in a FIFO (First In, First Out) behavior. There are two ends in the queue collection, i.e., front & rear. Queue has two ends that is front and rear.

The generic representation of the Java Queue interface is shown below:

```
public interface Queue<T> extends Collection<T>
```

As we have discussed above that the Queue is an interface, therefore we can also say that the queue cannot be instantiated because interfaces cannot be instantiated. If a user wants to implement the functionality of the Queue interface in Java, then it is mandatory to have some solid classes that implement the Queue interface.

In Java programming language, there are two different classes which are used to implement the Queue interface. These classes are:

- LinkedList
- PriorityQueue

Characteristics of the Java Queue

The Java Queue can be considered as one of the most important data structures in the programming world. Java Queue is attractive because of its properties. The significant properties of the Java Queue data structure are given as follows:

- Java Queue obeys the FIFO (First In, First Out) manner. It indicates that elements are entered in the queue at the end and eliminated from the front.
- The Java Queue interface gives all the rules and processes of the Collection interface like inclusion, deletion, etc.
- There are two different classes that are used to implement the Queue interface. These classes are LinkedList and PriorityQueue.
- Other than these two, there is a class that is, Array Blocking Queue that is used to implement the Queue interface.
- There are two types of queues, Unbounded queues and Bounded queues. The Queues that are a part of the java.util package are known as the Unbounded queues and bounded queues are the queues that are present in java.util.concurrent package.
- The Deque or (double-ended queue) is also a type of queue that carries the inclusion and deletion of elements from both ends.
- The deque is also considered thread-safe.
- Blocking Queues are also one of the types of queues that are also thread-safe. The Blocking Queues are used to implement the producer-consumer queries.
- Blocking Queues do not support null elements. In Blocking queues, if any work similar to null values is tried, then the NullPointerException is also thrown.

Interfaces used in implementation of Queue

The Java interfaces are also used in the implementation of the Java queue. The interfaces that are used to implement the functionalities of the queue are given as follows:

- Deque
- Blocking Queue
- Blocking Deque

Java Queue Array Implementation

Queue implementation is not as straightforward as a stack implementation.

To implement queue using Arrays, we first declare an array that holds n number of elements.

Then we define the following operations to be performed in this queue.

1) Enqueue: An operation to insert an element in the queue is Enqueue (function queue Enqueue in the program). For inserting an element at the rear end, we need first to check if the queue is full. If it is full, then we cannot insert the element. If rear < n, then we insert the element in the queue.

2) Dequeue: The operation to delete an element from the queue is Dequeue (function queue Dequeue in the program). First, we check whether the queue is empty. For dequeue operation to work, there has to be at least one element in the queue.

3) Front: This method returns the front of the queue.

4) Display: This method traverses the queue and displays the elements of the queue.

QueueArrayImplementation.java

```
1.  class Queue {
2.
3.      private static int front, rear, capacity;
4.      private static int queue[];
5.
6.      Queue(int size) {
7.          front = rear = 0;
8.          capacity = size;
9.          queue = new int[capacity];
10.     }
11.
12.     // insert an element into the queue
```

```
13.  static void queueEnqueue(int item)  {
14.      // check if the queue is full
15.      if (capacity == rear) {
16.          System.out.printf("\nQueue is full\n");
17.          return;
18.      }
19.
20.      // insert element at the rear
21.      else {
22.          queue[rear] = item;
23.          rear++;
24.      }
25.      return;
26.  }
27.
28.  //remove an element from the queue
29.  static void queueDequeue()  {
30.      // check if queue is empty
31.      if (front == rear) {
32.          System.out.printf("\nQueue is empty\n");
33.          return;
34.      }
35.
36.      // shift elements to the right by one place uptil rear
37.      else {
38.          for (int i = 0; i < rear - 1; i++) {
39.              queue[i] = queue[i + 1];
40.          }
41.
42.
43.     // set queue[rear] to 0
44.          if (rear < capacity)
45.              queue[rear] = 0;
46.
47.          // decrement rear
48.          rear--;
49.      }
```

```
50.         return;
51.     }
52.
53.     // print queue elements
54.     static void queueDisplay()
55.     {
56.         int i;
57.         if (front == rear) {
58.             System.out.printf("Queue is Empty\n");
59.             return;
60.         }
61.
62.         // traverse front to rear and print elements
63.         for (i = front; i < rear; i++) {
64.             System.out.printf(" %d , ", queue[i]);
65.         }
66.         return;
67.     }
68.
69.     // print front of queue
70.     static void queueFront()
71.     {
72.         if (front == rear) {
73.             System.out.printf("Queue is Empty\n");
74.             return;
75.         }
76.         System.out.printf("\nFront Element of the queue: %d", queue[front]);
77.         return;
78.     }
79. }
80.
81. public class QueueArrayImplementation {
82.     public static void main(String[] args) {
83.         // Create a queue of capacity 4
84.         Queue q = new Queue(4);
85.
86.         System.out.println("Initial Queue:");
```

```
87.        // print Queue elements
88.         q.queueDisplay();
89.
90.         // inserting elements in the queue
91.         q.queueEnqueue(10);
92.         q.queueEnqueue(30);
93.         q.queueEnqueue(50);
94.         q.queueEnqueue(70);
95.
96.         // print Queue elements
97.         System.out.println("Queue after Enqueue Operation:");
98.         q.queueDisplay();
99.
100.                // print front of the queue
101.                q.queueFront();
102.
103.                // insert element in the queue
104.                q.queueEnqueue(90);
105.
106.                // print Queue elements
107.                q.queueDisplay();
108.
109.                q.queueDequeue();
110.                q.queueDequeue();
111.                System.out.printf("\nQueue after two dequeue operations:");
112.
113.                // print Queue elements
114.                q.queueDisplay();
115.
116.                // print front of the queue
117.                q.queueFront();
118.            }
119.        }
```

Output:

```
Initial Queue:
Queue is Empty
Queue after Enqueue Operation:
```

```
 10 ,  30 ,  50 ,  70 ,
Front Element of the queue: 10
Queue is full
 10 ,  30 ,  50 ,  70 ,
Queue after two dequeue operations: 50 ,  70 ,
Front Element of the queue: 50
```

RECURSION

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.

ava Recursion Example 4: Fibonacci Series

```
1.  public class RecursionExample4 {
2.      static int n1=0,n2=1,n3=0;
3.       static void printFibo(int count){
4.          if(count>0){
5.              n3 = n1 + n2;
6.              n1 = n2;
7.              n2 = n3;
8.              System.out.print(" "+n3);
9.              printFibo(count-1);
10.         }
11.     }
12.
13. public static void main(String[] args) {
14.     int count=15;
15.      System.out.print(n1+" "+n2);
16.      printFibo(count-2);
17. }
18. }
```

Call by Value and Call by Reference in Java

There is only call by value in java, not call by reference. If we call a method passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method.

LINKED LIST

Java LinkedList class

Java LinkedList class uses a doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

The important points about Java LinkedList are:

- Java LinkedList class can contain duplicate elements.
- Java LinkedList class maintains insertion order.
- Java LinkedList class is non synchronized.
- In Java LinkedList class, manipulation is fast because no shifting needs to occur.
- Java LinkedList class can be used as a list, stack or queue.

Hierarchy of LinkedList class

As shown in the above diagram, Java LinkedList class extends AbstractSequentialList class and implements List and Deque interfaces.

Doubly Linked List

In the case of a doubly linked list, we can add or remove elements from both sides.

Hierarchy of LinkedList class

As shown in the above diagram, Java LinkedList class extends AbstractSequentialList class and implements List and Deque interfaces.

Doubly Linked List

In the case of a doubly linked list, we can add or remove elements from both sides.

LinkedList class declaration

```
public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>
```

Java LinkedList Example

```
1. import java.util.*;
2. public class LinkedList1{
3.  public static void main(String args[]){
4.
```

```
5.    LinkedList<String> al=new LinkedList<String>();
6.    al.add("Ravi");
7.    al.add("Vijay");
8.    al.add("Ravi");
9.    al.add("Ajay");
10.
11.   Iterator<String> itr=al.iterator();
12.   while(itr.hasNext()){
13.    System.out.println(itr.next());
14.   }
15.  }
16. }
```

Output:

```
Ravi
Vijay
Ravi
Ajay
```

HASH MAP AND HASH TABLE

Java HashMap

Java **HashMap** class implements the Map interface which allows us *to store key and value pair*, where keys should be unique. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the java.util package.

Points to remember

- Java HashMap contains values based on the key.
- Java HashMap contains only unique keys.
- Java HashMap may have one null key and multiple null values.
- Java HashMap is non synchronized.
- Java HashMap maintains no order.
- The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

HashMap class Parameters

Let's see the Parameters for java.util.HashMap class.

- **K**: It is the type of keys maintained by this map.
- **V**: It is the type of mapped values.

Method	Description
void clear()	It is used to remove all of the mappings from this map.
boolean isEmpty()	It is used to return true if this map contains no key-value mappings.
Object clone()	It is used to return a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
Set entrySet()	It is used to return a collection view of the mappings contained in this map.
Set keySet()	It is used to return a set view of the keys contained in this map.
V put(Object key, Object value)	It is used to insert an entry in the map.
void putAll(Map map)	It is used to insert the specified map in the map.

V putIfAbsent(K key, V value)	It inserts the specified value with the specified key in the map only if it is not already specified.
V remove(Object key)	It is used to delete an entry for the specified key.
boolean remove(Object key, Object value)	It removes the specified values with the associated specified keys from the map.
V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)	It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null.
V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null.
boolean containsValue(Object value)	This method returns true if some value equal to the value exists within the map, else return false.
boolean containsKey(Object key)	This method returns true if some key equal to the key exists within the map, else return false.
boolean equals(Object o)	It is used to compare the specified Object with the Map.
void forEach(BiConsumer<? super K,? super V> action)	It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.
V get(Object key)	This method returns the object that contains the value associated with the key.
V getOrDefault(Object key, V defaultValue)	It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key.
boolean isEmpty()	This method returns true if the map is empty; returns false if it contains at least one key.
V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)	If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
V replace(K key, V value)	It replaces the specified value for a specified key.
boolean replace(K key, V oldValue, V newValue)	It replaces the old value with the new value for a specified key.

void replaceAll(BiFunction<? super K,? super V,? extends V> function)	It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
Collection<V> values()	It returns a collection view of the values contained in the map.
int size()	This method returns the number of entries in the map.

Java HashMap Example

Let's see a simple example of HashMap to store key and value pair.

```
1. import java.util.*;
2. public class HashMapExample1{
3.  public static void main(String args[]){
4.    HashMap<Integer,String> map=new HashMap<Integer,String>();//Creating HashMap
5.    map.put(1,"Mango");  //Put elements in Map
6.    map.put(2,"Apple");
7.    map.put(3,"Banana");
8.    map.put(4,"Grapes");
9.
10.   System.out.println("Iterating Hashmap...");
11.   for(Map.Entry m : map.entrySet()){
12.    System.out.println(m.getKey()+" "+m.getValue());
13.   }
14. }
15. }
```

```
Iterating Hashmap...
    1 Mango
    2 Apple
    3 Banana
    4 Grapes
```

Java Hashtable class

Java Hashtable class implements a hashtable, which maps keys to values. It inherits Dictionary class and implements the Map interface.

Points to remember

- A Hashtable is an array of a list. Each list is known as a bucket. The position of the bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.

- Java Hashtable class contains unique elements.
- Java Hashtable class doesn't allow null key or value.
- Java Hashtable class is synchronized.
- The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.

Hashtable class declaration

Let's see the declaration for java.util.Hashtable class.

```
public class Hashtable<K,V> extends Dictionary<K,V> implements Map<K,V>
```

Hashtable class Parameters

Let's see the Parameters for java.util.Hashtable class.

- **K**: It is the type of keys maintained by this map.
- **V**: It is the type of mapped values.

TREE

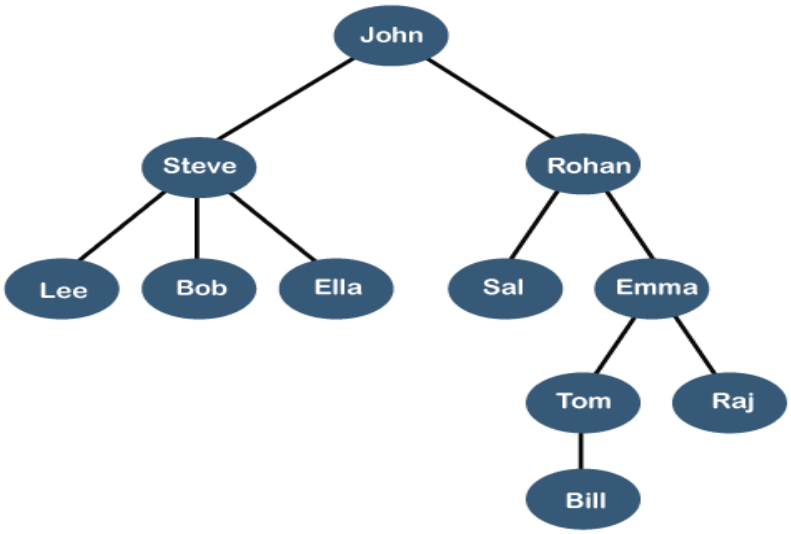
What is Tree Data Structure

We read the linear data structures like an array, linked list, stack and queue in which all the elements are arranged in a sequential manner. The different data structures are used for different kinds of data.

Some factors are considered for choosing the data structure:

- **What type of data needs to be stored**?: It might be a possibility that a certain data structure can be the best fit for some kind of data.
- **Cost of operations:** If we want to minimize the cost for the operations for the most frequently performed operations. For example, we have a simple list on which we have to perform the search operation; then, we can create an array in which elements are stored in sorted order to perform the ***binary search***. The binary search works very fast for the simple list as it divides the search space into half.

Memory usage: Sometimes, we want a data structure that utilizes less memory



The above tree shows the **organization hierarchy** of some company. In the above structure, ***john*** is the **CEO** of the company, and John has two direct reports named as ***Steve*** and ***Rohan***. Steve has three direct reports named ***Lee, Bob, Ella*** where ***Steve*** is a manager. Bob has two direct reports named ***Sal*** and ***Emma***. **Emma** has two direct reports named ***Tom*** and ***Raj***. Tom has one direct report named ***Bill***. This particular logical structure is known as a ***Tree***. Its structure is similar to the real tree, so it is named a ***Tree***. In this structure, the ***root*** is at the top, and its branches are moving in a downward direction. Therefore, we can say that the Tree data structure is an efficient way of storing the data in a hierarchical way.

Let's understand some key points of the Tree data structure.

- A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.
- A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in a Tree are arranged in multiple levels.
- In the Tree data structure, the topmost node is known as a root node. Each node contains some data, and data can be of any type. In the above tree structure, the node contains the name of the employee, so the type of data would be a string.
- Each node contains some data and the link or reference of other nodes that can be called children.

Some basic terms used in Tree data structure.

- **Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is **the root node of the tree.** If a node is directly linked to some other node, it would be called a parent-child relationship.
- **Child node:** If the node is a descendant of any node, then the node is known as a child node.
- **Parent:** If the node contains any sub-node, then that node is said to be the parent of that sub-node.
- **Sibling:** The nodes that have the same parent are known as siblings.
- **Leaf Node:-** The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.
- **Internal nodes:** A node has atleast one child node known as an ***internal***
- **Ancestor node:-** An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.
- **Descendant:** The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

Properties of Tree data structure

- **Recursive data structure:** The tree is also known as a ***recursive data structure***. A tree can be defined as recursively because the distinguished node in a tree data structure is known as a ***root node***. The root node of the tree contains a link to all the roots of its subtrees. The left subtree is

shown in the yellow color in the below figure, and the right subtree is shown in the red color. The left subtree can be further split into subtrees shown in three different colors. Recursion means reducing something in a self-similar manner. So, this recursive property of the tree data structure is implemented in various applications.

- **Number of edges:** If there are n nodes, then there would n-1 edges. Each arrow in the structure represents the link or path. Each node, except the root node, will have atleast one incoming link known as an edge. There would be one link for the parent-child relationship.
- **Depth of node x:** The depth of node x can be defined as the length of the path from the root to the node x. One edge contributes one-unit length in the path. So, the depth of node x can also be defined as the number of edges between the root node and the node x. The root node has 0 depth.
- **Height of node x:** The height of node x can be defined as the longest path from the node x to the leaf node.

DYNAMIC PROGRAMING

What is Dynamic Programming

Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again. The subproblems are optimized to optimize the overall solution is known as optimal substructure property. The main use of dynamic programming is to solve optimization problems. Here, optimization problems mean that when we are trying to find out the minimum or the maximum solution of a problem. The dynamic programming guarantees to find the optimal solution of a problem if the solution exists.

The definition of dynamic programming says that it is a technique for solving a complex problem by first breaking into a collection of simpler subproblems, solving each subproblem just once, and then storing their solutions to avoid repetitive computations.

Consider an example of the Fibonacci series. The following series is the Fibonacci series:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ,…

The numbers in the above series are not randomly calculated. Mathematically, we could write each of the terms using the below formula:

F(n) = F(n-1) + F(n-2),

With the base values F(0) = 0, and F(1) = 1. To calculate the other numbers, we follow the above relationship. For example, F(2) is the sum **f(0)** and **f(1),** which is equal to 1.

Approaches of dynamic programming

There are two approaches to dynamic programming:

- Top-down approach
- Bottom-up approach

Top-down approach

The top-down approach follows the memorization technique, while bottom-up approach follows the tabulation method. Here memorization is equal to the sum of recursion and caching. Recursion means calling the function itself, while caching means storing the intermediate results.

Advantages

- It is very easy to understand and implement.
- It solves the subproblems only when it is required.
- It is easy to debug.

Bottom-Up approach

The bottom-up approach is also one of the techniques which can be used to implement the dynamic programming. It uses the tabulation technique to implement the dynamic programming approach. It solves the same kind of problems but it removes the recursion. If we remove the recursion, there is no stack overflow issue and no overhead of the recursive functions. In this tabulation technique, we solve the problems and store the results in a matrix.

How does the dynamic programming approach work?

The following are the steps that the dynamic programming follows:

- It breaks down the complex problem into simpler subproblems.
- It finds the optimal solution to these sub-problems.
- It stores the results of subproblems (memoization). The process of storing the results of subproblems is known as memorization.
- It reuses them so that same sub-problem is calculated more than once.
- Finally, calculate the result of the complex problem.

The above five steps are the basic steps for dynamic programming. The dynamic programming is applicable that are having properties such as:

Those problems that are having overlapping subproblems and optimal substructures. Here, optimal substructure means that the solution of optimization problems can be obtained by simply combining the optimal solution of all the subproblems.

In the case of dynamic programming, the space complexity would be increased as we are storing the intermediate results, but the time complexity would be decreased.

GRAPH AND BITMASKING

Graph

A **graph** is a data structure that stores connected data. In other words, a graph G (or g) is defined as a set of vertices (V) and edges (E) that connects vertices. The examples of graph are a social media network, computer network, Google Maps, etc.

Each graph consists of **edges** and **vertices** (also called nodes). Each vertex and edge have a relation. Where vertex represents the data and edge represents the relation between them. Vertex is denoted by a circle with a label on them. Edges are denoted by a line that connects nodes (vertices).

Graph Terminology

Vertex: Vertices are the point that joints edges. It represents the data. It is also known as a node. It is denoted by a circle and it must be labeled. To construct a graph there must be at least a node. For example, house, bus stop, etc.

Edge: An edge is a line that connects two vertices. It represents the relation between the vertices. Edges are denoted by a line. For example, a path to the bus stop from your house.

Weight: It is labeled to edge. For example, the distance between two cities is 100 km, then the distance is called weight for the edge.

Path: The path is a way to reach a destination from the initial point in a sequence

Types of Graph

- **Weighted Graph:** In a weighted graph, each edge contains some **data** (weight) such as distance, weight, height, etc. It denoted as w(e). It is used to calculate the cost of traversing from one vertex to another. The following figure represents a weighted graph.
- **Unweighted Graph:** A graph in which edges are not associated with any value is called an unweighted graph. The following figure represents an unweighted graph
- **Directed Graph:** A graph in which edges represent direction is called a directed graph. In a directed graph, we use arrows instead of lines (edges). Direction denotes the way to reach from one node to another node. Note that in a directed graph, we can move either in one direction or in both directions. The following figure represents a directed graph.
- **Undirected Graph:** A graph in which edges are bidirectional is called an undirected graph. In an undirected graph, we can traverse in any direction. Note that we can use the same path for return through which we have traversed. While in the directed graph we cannot return from the same path.

- **Connected Graph:** A graph is said to be connected if there exists at least one path between every pair of vertices. Note that a graph with only a vertex is a connected graph.

Bitmasking

Bitmasking allows us to store multiple values inside one numerical variable. **Instead of thinking about this variable as a whole number, we treat its every bit as a separate value**. Because a bit can equal either zero or one, we can also think of it as either false or true. We can also slice a group of bits and treat them as a smaller number variable or even a *String*.

CONCLUSION

This course covered the basics of data structures. With this we have only scratched the surface. Although we have built a good foundation to move ahead.

Data Structures is not just limited to Stack, Queues, and Linked Lists but is quite a vast area. There are many more data structures which include Maps, Hash Tables, Graphs, Trees, etc. Each data structure has its own advantages and disadvantages and must be used according to the needs of the application. A computer science student at least know the basic data structures along with the operations associated with them.

Many high level and object oriented programming languages like C#, Java, Python come built in with many of these data structures. Therefore, it is important to know how things work under the hood.