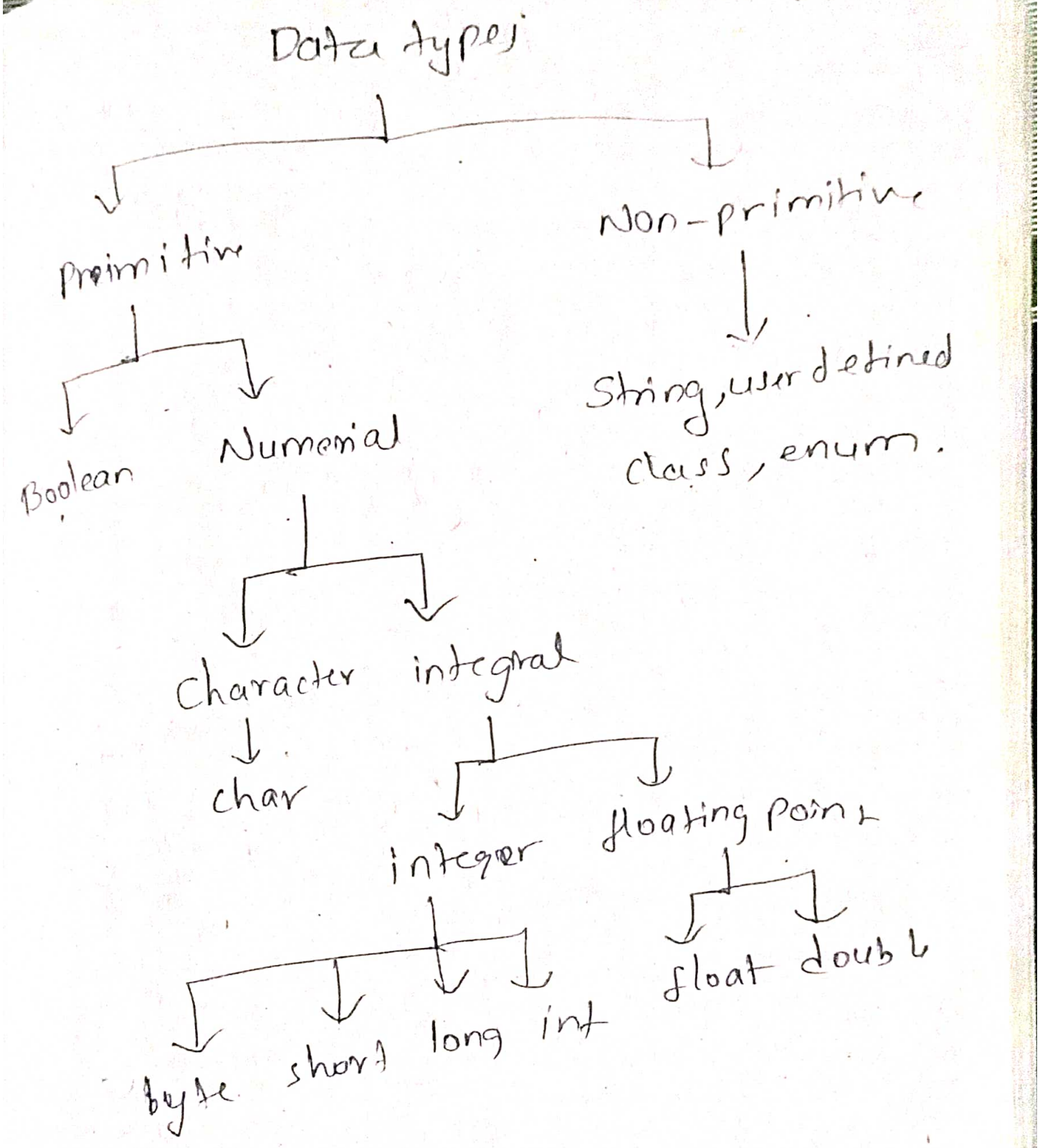

Data types
Primitive
Non-primitive
Boolean
Numerial
String, user defined
class, enum.
Character
integral
char
integer
floating point
byte
short
long
int
float
doub l

Datatype	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

Buzzword

1. Platform Independence (Jvm use)
2. Object - oriented
3. Multi-threading.
4. Exception handling.

5) Security.

6) portability

7) Robustness — features like compile-time check, exception handling reducing the errors

8) open source

9) performance

10) Standard library.

Inheritance.

It is an object oriented technology used for deriving the features of one class in to another class. When one class accessing the properties of another class is called inheritence

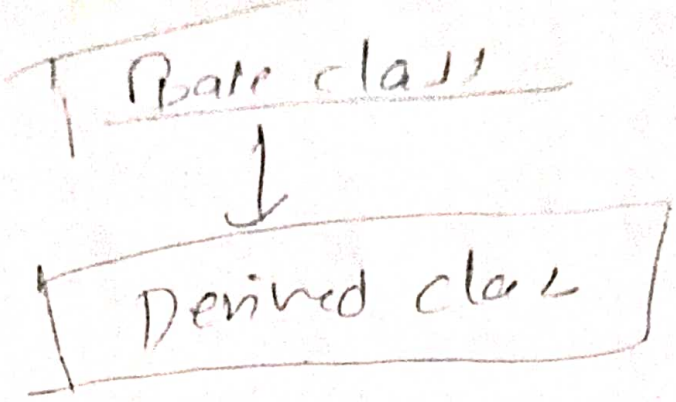
```
class A {
    super class
    parent class
    Base class
}

class B extend : A {
    sub class
    child class
    derived class
}
```

Types of inheritance

1. single inheritance
2. Hierarichical inheritance
3. multilevel inheritance

(1) single inheritance :-

It refers to the process of driving a class from a single base class. It has only one base class and one derived class

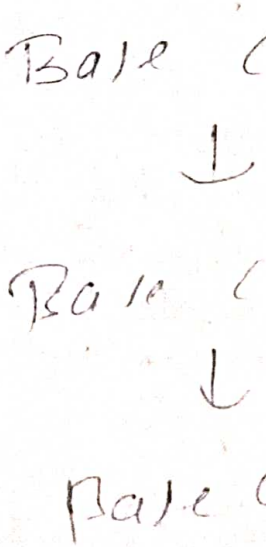


Syntax

```
class Parent {

}
class child extends Parent {

}
```

Multilevel inheritance :-

- it is a process of deriving a class from another derived class as shown

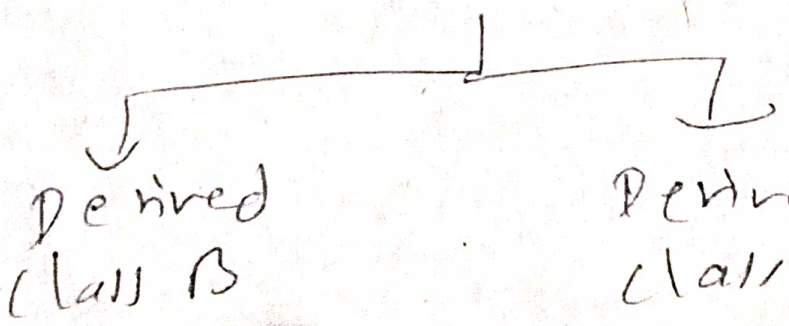


Syntax:-

```
class A {

}
class B extends A {

}
class C extends B {

}
```

Hierarchial inheritance

is a process of deriving many classes from single base class. All the derived classes can be further inherited by other classes in the same way



Syntax:-

```
class A {

}
class B extends A {

}
class C extends A {

}
```

Polymorphism:-

is a greek word derived from ~~no~~ poly means many and morphism means many forms.

two types of Polymorphism:-

1. method overloading
2. method overriding.

Method overloading:-

allows you to define multiple methods in a class with the same name but different parameters

Example

```
class Mathunits {
    int add (int a, int b) {
        return a+b;
    }
    int add (double a, double b) {
        return a+b;
    }
}
```

Method overriding:-

It occurs when a subclass provides a specific implementation of a method that is already

defined in superclass. It has same name, parameters and return type as in method in the superclass.

Example

```
class Animal {
    void makeSound() {
        System.out.println("Animal sound");
    }
}
class Dog extend Animal {
    void makeSound() {
        System.out.println("Dog bark");
    }
}
```

Constructor overloading:-

In Java that allows you to define multiple constructors for a class with different parameter lists. It is similar to method overloading.

Example

```
class Person{
    private string. name;
    privat int age;

    Public person(){
        this.name = "unknown";
        this.age = 0;
    }
    Public person(strins name){
        this.name = name;
        this.age = 0;
    }
```

```
public Person(String name, int age){
    this.name = name;
    this.age = age;
}
public String getName(){
    return name;
}
public int getAge(){
    return Age;
}
}
```

Output

Super

'super' is a keyword used to refer to the superclass (parent class) of the ~~subject~~ current object.

① Superclass Constructor :-
You can use 'super' to call a constructor of the superclass, from a subclass constructor

Ex :-
```
class Animal {
    int legs;
    Animal(int legs) {
        this.legs = legs;
    }
}

class Dog extends Animal {
    String breed;
    Dog(int legs, String breed) {
        super(legs);
        this.breed = breed;
    }
}
```

⑤ Superclass Method :-
like method overriding

```
class Animal {
    void makeSound() {
        System.out.println("Animal sound");
    }
}
class Dog extends Animal {
    void makeSound() {
        super.makeSound();
        System.out.println("Dog barks");
    }
}
```

* final :-

- 'final' is a keyword used to indicate the class, method, or variable cannot be extended, overridden, or modified, respectively.

Ex:-

```
final class MyClass {
    // cannot be extended
}

final int my_constant = 49;
// A constant variable
```

This

'this' is keyword used to refer to the current instance of an object. within its own class.

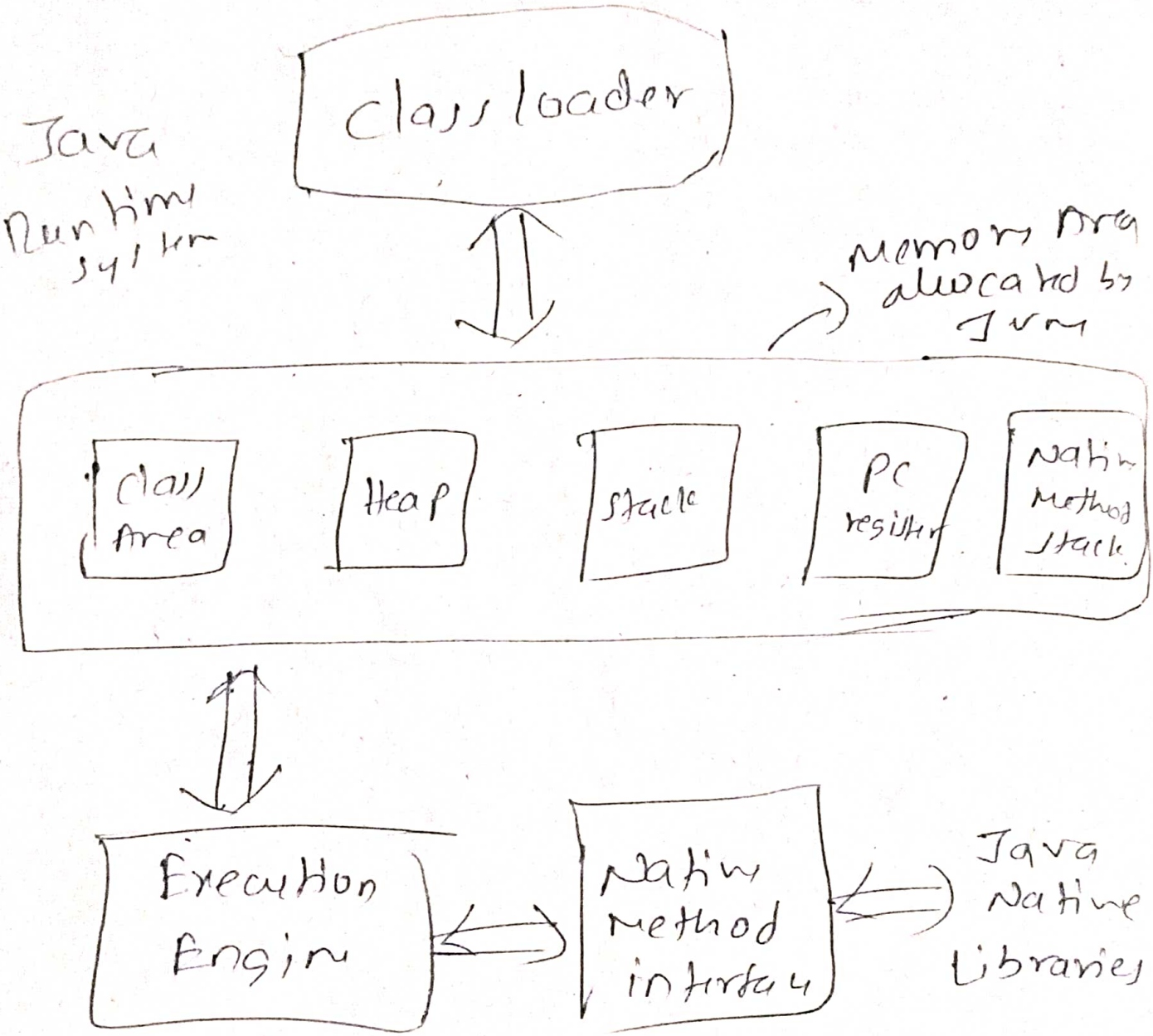
Ex:-

```
class Person {
    String name;
    Person(String name) {
        this.name = name;
    }
}
```

* It is often used to distinguish between instance variable and parameter or to call constructor with other constructors.

```
Rectangle(){
    this(0, 0);
}
Rectangle(int width, int length){
    this.width = width;
    this.length;
}
}
```

JVM Architecture (Java virtual machine)



String handling functions

① create string:-

```
string str = "Hello world";
string str = new string("Helloworld");
```

↳ string object

② string length:-

```
int length = str.length();
```

③ concatenation:-

```
string concatenate = str1 + str2;
```

another method `concat`

```
string concatenated = str1.concat(str2);
```

④ string comparison:-

```
boolean isEqual = str1.equal(str2);
```

(5) Case Conversion

String uppercase = str.toUpperCase();

String lowercase = str.toLowerCase();

Java provides a wide range of string handling functions and methods as part of the 'Java.lang.String'

Unit-2

Package

The packages are a way to organize and manage classes and interfaces into namespaces. It provides avoid naming conflicts, improve code organization.

* It is a direction that contains a group of similar types of classes, interfaces and sub packages

Package Declaration

Package com.example.myapp;

Java standard packages

1. Java.util
2. Java.awt.

Types of Packages

1. predifined Packages.
2. Werdefined packages.

- Predifined Packages

Packages are part of the Java standard Library and come bundled with the Java Development Kit (JDK).

Some common packages are

1. 'Java.lang'
2. 'Java.util'
3. 'Java.io'
4. 'Java.awt'
5. 'Java.swing'

User-defined Packages

User defined packages are packages created by developers to organize their own classes and Interfaces.

* This packages are not a part of Java standard libaraies.

Steps to Create

- Declare Package statement in beginning of Java source code
- Organize your sourcod.

Ex:-

```
Package com.example.myapp;

Public Class Myclass {
    // code.
}
```

Now in another Java file we access this package

```
import com.example.myapp.MyClass;
```

Interface in Java

Interface is a core concept of object-oriented programming that defines a contract for classes to follow. it specifies a set of methods that implementing class must provide, without specifing the actual implementation.

Declaring an interface :-

```
public interface MyInterface {
    void method1();
    int method2();
}
```

implementing a interface

- A class can implement one or more interfaces by using the 'implement' keyword.

Ex :-

```
public class MyClass implements
                    MyInterface{

    public void method1() {
    }
    public int method2() {
    }
}
```

Interface inheritance :-

- Interface can extend other interfaces using the 'extend' keyword

public interface MySubInterface extends
↑MyInterface {

}

Multiple interface implementation:-

A class can implement multiple interfaces by seperating them with commas.

public class MyMultipleInterface
implements MyInterface1, MyInterface2
{

}

Polymorphism with interface

interface enable polymorphism, allowing objects to different classes that implement the same interface

MyInterface obj1 = new MyClass1
MyInterface obj2 = new MyClass2
obj1.method1();
obj2.method2();

Example of interface

```
interface shape() {
        double getArea();
}
class Circle implements shape{
        private double radius;
        public Circle (double radius){
            this.radius =radius;
        }
        public double getArea() {
            return Math.PI* radius*rad
        }
}
class Rectangle implements shape{
        private double width;
        private double height;
        public Rectangle (double width,
                          double height){
            this.width =width;
            this.height =height;
```

```
}
    public double getArea() {
        return width * height;
    }
}

public class ShapeExample {
    public static void main(String[] args) {
        Circle circle = new Circle(5.0);
        Rectangle rectangle = new Rectangle
                              (4.0, 6.0);

        System.out.println("Circle Area" + circle.
                           getArea());

        System.out.println("Rectangle Area" +
                           rectangle.getArea());
    }
}
```

Access Modifiers in Java

In Java modifiers are keywords used to specify the visibility and accessibility of class, method, field, and constructor

1. public
2. private
3. Protected
4. default

(1) public ('public') :-

The public access modifier means a class, method and constructor accessible from any other class.

Syntax:

```
public class MyClass {
    public int MyField;
    public void myMethod() {
        // public method
    }
}
```

(2) Private ('private'):

. The 'private' access modifier restricts the visibility of a class member to only the class in which it is declared.

```
public class myclass {
    private int myfield;
    private void mymethod()
        // private ...
}
```

(3) Protected:-

It allows a class member to be accessed with the same class, subclass with the same package, and subclass outside the package.

```
public class myclass {
    protected int myfield;
    protected void mymethod() {
        // code
    }
}
```

4. Default

- When no access modifier is specified it is known as the default access also called package-private.

```
class Myclass {
    int myfield;
    void mymethod() {
        //code
    }
}
```

Random Access files

It allows to perform random access file operations, such as reading, writing and seeking on a file.

(1) Creating a RandomAccessFile:-

```
import java.io.RandomAccessFile;

RandomAccessFile raf
    = new RandomAccessFile("example.txt",
                           "rw");
```

Writing Data :-

```
raf.writeUTF("Hello, world");
```

Seeking to a position :-

~~String data raf.~~

```
raf.seek(10);
```

Reading Data :-

```
String data = raf.readUTF();
```

closing the file :-

```
raf.close();
```

Exception handling

```
try {
} catch (IOException e) {
        e.printStackTrace();
}
```

Code for RandomAccessFile

```
import java.io.RandomAccessFile;

Public class RandomAccessFileExample {
    public static void main (String[] args)
    {
        try {
            RandomAccessFile raf
                = new RandomAccessFile("example.
                  txt", "rw");
            raf.writeUTF("HelloWorld");
            raf.seek(0);
            String data = raf.readUTF();
            System.out.println("Data read
                    from the file: " + data);
            raf.close();
        } catch (Exception e) {
        }
    }
}
```

Type wrapper and role of auto boxing

* Type wrapper is also known as wrapper class, it a class that provides an object representation of primitive data type.
The Java lang provides a set of standard wrapper class for each primitive data type

1. integer - wraps 'int'
2. Long - wraps 'long'
3. short - wraps 'short'
4. 'float' - wraps 'float'
5. 'Boolean' - wrap 'boolean'

The primary role of wrapper class is to allow primitive data type to be treated as Objects.

Auto Boxing is a feature introduced in Java that automatically converts primitive types to their corresponding wrapper objects when needed. It simplifies the process of conversions.

```
Integer num1 = 42;  // Auto-boxing int to Integer

int num2 = num1;  // Auto-unboxing Integer to int
```

Character Stream

It is to provide a way to read and write characters making them suitable to work with the text data. It is a part of 'Java.io'

The two primary classes for character streams in Java are 'Reader' and 'Writer'.

Methods available in 'Java.io.Reader':

1. int read() → reads single char
2. int read(char[] cbuf) → reads chara -cter into an array
3. boolean ready() → check if the reader is ready to read
4. void close() → closes the reader.

Methods available in 'Java.io.Writer':-

1. void write(int c): → writes the single character
2. void write(char[] cbuf): → writes the array of characters
3. void write(String str) → writes a string to the output

void close() → To close the writer.

Byte stream classes

It is used to performing input and output operations on binary data. This is a part of Java.io package

1. 'InputStream' and 'OutputStream'
   This are the abstract classes that serve as the base classes for byte-oriented input and output
2. 'FileInputStream' and 'FileOutputStream'
   - ↓ read bytes from file
   - ↓ write bytes to the file
3. 'ByteArrayInputStream' and 'ByteArrayOutputStream'
   - ↓ read bytes from byte array
   - ↓ write bytes from byte array

4. 'object Input stream' and 'object output stream'.

CLASSPATH Environment Variable.

'classpath' is a environment variable in Java is used to specify the locations where the Java virtual machine (JVM) should look for classes and resources.

1. Purpose of the 'classpath' is to tell the where to find the compiled classes and resources required for a Java application
2. if we don't set the 'classpath' then JVM uses a default classpath
3. if a class is not found in classpath then a "classNotFoundException" will be thrown at runtime

Unit - 3

Exception handling in Java

This allow to gracefully handle runtime errors and exceptional conditions that may occur during the execution of program.

~~Exception class~~

Types of Exception

checked Exception :-

These are exceptions that are checked at compile time and must be either declared using 'throws' clause in method signature or caught using try-catch blocks. Example

IO Exception, SQL Exception.

Unchecked Exception

These are exceptions that occur at runtime and are not checked at compile time. They do not need to declare or catch.

Ex:-
NullPointerException
ArrayIndexOutOfBoundsException.
ArithmeticException.

---

try-catch Block

try - block encloses the code that may throw an exception.

```
try {
    // code may throw exception
} catch(Exception ex1) {
    // handle ex1
} catch( Exception ex2) {
    // handle ex2
}
```

finally Block.

The code that must be executed regardless of whether the exception is thrown or not.

```
try {
} catch (Exception e) {
} finally {
}
```

throw and throws

throw - keyword is used to explicitly throw an exception within a method.

throws - keyword is used in method signature to declare that a method may throw certain types of exceptions, but doesn't

handle them.

Ex:

```
void mymethod() throws MyException{
    if (someCondition){
        throw new MyException
        ("An error occurd.");
    }
}
```

Java program that demonstrates how certain exception types are not allowed to be thrown.

```
Public class NotAllowedToThrowException{
    public static void main (String[] args){
        try{
            throw outofmemoryError("Custom out");
        }catch(OutofMemoryError e){
            System.out.println("caught an outofmem:
                " + e.getMessage());
        }
```

```
try {
    throw new StackOverflowError("This is a custom Stack
                                  Overflow error");
} catch (StackOverflowError e) {
    System.out.println("caught a
    StackOverflowError: " + e.getMessage());
}
}
}
```

Advantages of Exception handling

1. Error detection.
2. Robustness.
3. Maintenance and Debugging
4. Reusability
5. Extensibility.
6. Consistency
7. user-friendly messages.

Inter Thread Communication

It is a mechanism that allows multiple threads to synchronize and communicate with each other in a multithread program.

How Inter thread communication works in Java :-

(1) wait(), notify(), ~~notify~~ notifyAll() :-

wait() → goes to the waiting state until another thread notify it

notify() → it is to wake up one waiting thread.

notifyAll() → It is to wake up all waiting thread.

(2) Synchronization :-

To use Inter thread communication we typically, wrap the wait(), notify(), notifyAll() call within synchronized blocks.

To ensure one thread at a time can execute the synchronized block

(3) Producer - consumer Problem

- one common scenario for inter thread communication is the producer-consumer problem, where one or more thread produce data, and one or more thread consume data.

Example for ITC comm (ITC)

```
class Customer {
    int amount = 10000;

    synchronized void withdraw(int amount) {
        System.out.println("going to withdraw");
        if (this.amount < amount) {
            System.out.println("Less balance
                waiting for deposit");
            try { wait(); }
            wait();
        }
```

```
catch(Exception e){

  }

}
this.amount -= amount;
System.out.println("withdraw completed");

}
Synchronized void deposit(int amount){
      System.out.println("going to deposit");
      this.amount += amount;
      System.out.println("deposit completed");
      notify();
  }

}
class Test{
  Public static void main(String args[]){
      final Customer c = new Customer();
      new Thread(){
          public void run(){
              c.withdraw(15000);
  }.start();
```

```
new Thread(){
    public void run(){
        c.deposit(10000);
    }
}.start();

}}
```

Output

goins to withdraw...
less balance; waiting for deposit...
goins to deposit...
deposit completed...
withdraw completed...

Process-based multitasking	Thread-based multitasking
(1) two or more processes and programs can be run concurrently	(1) two or more thread, can be run concurrently.
(2) Program is a bigger unit	(2) Thread is a smaller unit.

It is comparatively heavyweight.	It is comparatively less weight.
The process require its own address space	In thread share the same address space
It requires more overhead	It requires less overhead

different ways to create multiple threaded program.

(1) Extending 'Thread' class

- you can create a thread by defining a class that extends in Java.lang.Thread class

(2). Implementing Runnable interface

Creating thread by extending Thread

class

```
Public class MyThread extends Thread {
    public void run() {
        for (int i=0; i<10; i++){
            System.out.println("Child Thread");
        }
    }
}
class ThreadDemo {
    Public static void main (String [] args){
        MyThread t = new MyThread();
        t.start();
        for(i=0; i<10; i++) {
            S.O.P.ln ("Main Thread");
        }
    }
}
```

① Creating thread using Runnable Interface.

```
public class RunnableDemo {
    P.S.V.main (String[] arg) {
        Runnable myRunnable
            = new MyRunnable();
        Thread t = new Thread
            (myRunnable);
        t.start();
    }
    S.O.P.ln("Main Thread is Running");
}
class MyRunnable implements Runnable {
    @Override
    Public void run() {
        S.O.P.ln ("child thread");
    }
}
```

child thread
Thread-0

Runtime Exception Example

```
Public class RuntimeExceptionExample
{
    public static void main(String[] args)
    {
        int dividend = 10;
        int divisor = 0;
```

```
try {
    int result = dividend/divisor;
    System.out.println("Result: " + result);
}
catch (ArithmeticException e){
    System.err.println
        ("An Arethm exception
        occured" + e.getMessage);
    }
}
```

Null pointer Exception:-

```
public class NullPointerException{
    public static void main(String[] args){
        String text = null;
        try {
            int length = text.length();
            system.out.println("Length
                " + length);
```

```
        }
        catch (NullPointerException e){
            System.err.println("An null
            pointer exception occured");
            e.getMessage();
        }
    }
}
```

Multi-thread program

```
public class MyThreadExample{
    public static void main(String[]
        args) {
    Thread thread1 = new Thread(new
                    MyRunnable("Thread1"));
Thread thread2 = new Thread(new MyRunnable
                    ("Thread 2"));
Thread thred3 = new Thred(new myRunnable
                    ("Thread 3"));
```

```
        Thread1.start();
        Thread2.start();
        Thread3.start();
    }
}

class MyRunnable implements Runnable{
    private String threadName;
    public MyRunnable(String threadName){
        this.threadName = threadName;
    }
    public void run(){
        for(int i=1;i<=5;i++){
            System.out.println(
            Thread.currentThread().getName()+" - count: "+i);
        }
        System.out.println(threadName
                + " finished.");
    }
}
```

Producer consumer problem for

IIC

```
class item {
    int i;
    boolean produced = false;
    public void synchronized Produce
                                (int x)
    {
      if (produced) {
          try {
              wait();
          } catch(){
          }
      }
      i = x;
      produced = true;
      notify();
    }
    consume() {
        if (!produced){
```

```
try {
    wait();
} catch(){
  }
}
produced = false;
 notify();
}
```

Unit-4

Arraylist in Java

It is commonly used from the Java collection framework, specifically in the 'java.util' package.

It is dynamic array implementation which it grow or shrink in size as needed.

Program

```
import java.util.ArrayList;

Public class ArrayListExample{

    Public static void main(String[] args
    {

        ArrayList<String> fruit = new
                                ArrayList<>();

        //addIns
        fruit.add("Apple");
        fruit.add("Banana");
        fruit.add("cherry");

            //removIns
        firuit: remove("Banana");

            //size total noof in
        .fruit.size().

    }
}
```

Array list	vector
* every method present in Array list is non synchronized	every method present in LinkedList is Synchronized.
* Array list is not a legacy class.	* vector is legacy class.
* Array list is fast because it is not synchronized	* vector is slow because it is Synchronized.
* It use iterator interface to traverse the elements.	* It use Enumeration interface to traverse the elements.

Iterator :

It is an interface provided by the 'java.util' package that allows you to iterate (loop) through the elements of a collection, such as 'List', 'Set' or 'Map'. Iterator provides a consistent way to access and traverse elements in a collection.

There are 3 main Methods

1. boolean hasNext() - Return true if there are more elements to be iterated
2. E next() - Return next element in the collection
3. void remove() - Remove the last Element returned by next()

Example

```
import java.util.ArrayList;
import java.util.Iterator;
public class IteratorExample {
    public static void main(String[] args){

        ArrayList<Integer> numbers
                = new ArrayList<>();

        number.add(1);
        number.add(2);
        number.add(3);
        number.add(4);
        number.add(5);

        Iterator<Integer> iterator
                = numbers.iterator();

        while (iterator.hasNext()){
            Integer number = iterator.next();
            System.out.println(number);
        }
    }
}
```

Output

1
2
3
4
5

String Tokenizer :-

It is used to break a string into tokens, wher each token is typically a substring separated by a delimiter character or ~~characters~~ characters. It is Part of java.util. package.

(1) creation:

StringTokenizer tokenizer
= new StringTokenizer (inputString, delimiter);

Example

```
import java.util.StringTokenizer;

public class StringTokenizerExample {

    public static void main(String[] args)
    {

    String input = "apple,banana,cherry";

    StringTokenizer tokenizer = new
            StringTokenizer(input, ",");

    while (tokenizer.hasMoreTokens()){

        String token = tokenizer.nextToken();

        System.out.println("Token: " +
                token);

    }

}

}
```

Output

token : apple

token : banana

token : cherry

Java Collections framework (JCF)

it is a set of classes and interfaces in Java that provide a comprehensive, unified, and high performance work with collection of object. It was introduced in Java 2. It is an efficient way to store, retrieve, and manuplicate data structure in Java applications. It is a part of Java.util. package and includes interfaces, classes and algorithms for working with collection of object

components

(1) Interface :-

→ collection.

→ List → LinkedList, ArrayList

→ Set → HashSet, TreeSet

→ Queue → LinkedList, PriorityQueue

→ Map

(2) classes

↳ ArrayList, HashMap, HashSet.

(3) Algorithm

The Framework provides uses of algorithms for common operations like sorting, searching and manuplation.

Benifits of Collection framework

(1) consistency

(2) Efficiency.

(3) Safety.

(4) Scalability.

(5) Ease of use

(6) Thread safety.

It offers a robust and versatile set of tools for managing collections of data in Java applications, making development more efficient, consistent, and reliable. It is widely used in various applications and libraries.

1) Vector class

* It is a class.
* It is belongs to java.util package
* It this class can be used to create a generic dynamic array known as vector.
* It hold object of any type
* Every method present in vector class is synchronized

* functions of array list and vector is same

Creating

```
vector obj = new vector();
```

```
vector <type> obj
        = new vector <type>();
```

method

add(), addAll(), addElement(), clear(), firstElement().

Program

```
class vector {
    public static void main(String
                            args[])
    {
      vector <Integer> obj = new
                          vector<>();
      obj.add(10);
      obj.add(20);
      obj.add(30);
```

```
obj.add(40);
obj.add(50);
System.out.println(obj);
System.out.println("size of
        vector: "+obj.size());
obj.add(3,100)
    System.out.println(obj);
}
}
```

Output

[10, 20, 30, 40, 50]

size of vector = 5

[10, 20, 30, 100, 40, 50]

Collection class

It is usually refer to a class or interface within the Java collections framework (JCF) that is designed for storing and manipulating groups of objects. It is a part of Java. Util. commonly used collection classes and interfaces are

1. List Interface :-

~~Array<S~~

ArrayList <String> name = new ArrayList <>();

2. Set Interface :-

HashSet <Integer> number = new HashSet<>();

3. Map Interface :-

HashMap <String, Integer> scores = new HashMap<>();

4. Queue interface:-

Queue<String> queue = new LinkedList<>();

5. Stack class:-

Stack<String> stack = new Stack<>

6. Vector class:-

Vector<String> vector = new Vector<>()

7. Arraylist class:-

ArrayList<Integer> list = new ArrayList<>();

8. LinkedList class:-

LinkedList<String> linkedlist = new
LinkedList<>();

9. HashMap class:-

HashMap<String, Integer> map =
new HashMap<>();

10. HashSet class:-

HashSet<Double> set = new HashSet<>()

Legacy class

It is to a class that predates or was developed before certain modern practices, standards.

These classes may not adhere to modern design principles or programming paradigms.

legacy classes also refer to classes that has been part of Java's standard library for a long time.

Example vector class, ArrayList.