Unit-2
Exception Handling and Streams

Topics to cover:

- *) Arrays, Strings, Packages
- *) Javadoc comments
- *) Inheritance
- *) class hierarchy
- *) polymorphism
- *) dynamic binding
- *) final keyword
- *) abstract classes
- *) exception handling
- *) exception hierarchy
- *) Throwing and catching exceptions
- *) The object class
- *) Collection
- *) Interfaces
- *) object cloning
- *) Inner classes
- *) proxies
- *) I/O Streams

Arrays, strings, Packages

*) Array: They are collection of similar type of elements, immutable
In Java arrays are dynamically allocated and are considered objects
Terms: Index, size limit, Types, multidimensional Arrays
Types: one dimensional, multi dimensional

a) Strings and StringBuffer: They are group of multiple characters, Immutable, whereas StringBuffer are Mutable

	String	StringBuffer
Creating	String s = "hello"	(StringBuffer sb = new StringBuffer ("hallo"));
Methods	compare, concat, equals length, subString, toUpperCase etc..	append(), insert(), replace(), delete(), reverse(), capacity()

Codes: Array

Finding Minimum in Array (multidimensional also known as Jussed array

```
Public class arrays {
    Static void findMin(int arr[]) {  ⟶ Array as parameters
        int min = arr[0];
        for (int i=1; i<arr.length; i++) {
            if (min > arr[i]) {
                min = arr[i];
            }
        }
        S.O.P ("Minimum" + min);
    Static void oneDarray() {  → one dimensional
        int[] number = {33, 5, 4, 5};
        for (int i=0; i<number.length; i++) {  → To print them in O/P
            S.O.P (number[i]);
        }
        findMin (numbers);
    Static void twoDarrays {  → multidimensional
        int twoD[][] = new int[3][2]
        int k=0
        for (int i=0; i<3; i++) {
            for (int j=0; j<2; j++) {
                twoD[i][j] = k;
                k++;
    }   }   S.O.P (twoD);
```

M T W T F S S

Page No.:

Date:

YOUVA

Op:

One D arrays: 2 3 3 4 5

Minimum 3

Multi D arrays

0 1 2 3 4

5 6 7 8 9

10 11 12 13 14

15 16 17 18 19

String and String Buffer code:

```
public class StringAndBuffer {
    public static void main(String args[]) {
        String a = "Info";
        String b = "Tech";
        String concatResult = a.concat(b);
        System.out.println(a);
            S.O.P (b);
            S.O.P (concatResult);
            S.O.P (a.toUpperCase());
            S.O.P (a.charAt(2));
        StringBuffer sb = new StringBuffer("Hello");
        sb.append(" Computer");
        S.O.P(sb);
        sb.insert(1, "Java");
        S.O.P(sb);
        sb.reverse();
        S.O.P(sb);
```

→ String

→ String Buffer

Page No.:
Date:

OP:

Informtn
Info tech
INFORMATION
F

Hellocompiler
HJavallocompiler
olleH

Inheritance and polymorphism : Same as unit-I Just refer those notes, If you understand those codes well with types you are good to go ^_^

Class Hierachy

1) Core structure and Root of Hierarchy
*) Root (Parent Super)
*) Inheritance

Single
Multilevel
Hybrid
Hierarchical
Multiple

Then write Java code (refer unit 1)

Dynamic Binding / Dynamic Method Dispatch

Dynamic Method dispatch is the mechanism by which a call to an overridden method is resolved at run time

a) Mechanism : when an overridden method is called through a superclass reference variable the Java Virtual Machine (JVM) determines which version of the method to execute based on the actual type of the object being referred to at that moment, not the type of the reference variable

a) upcasting : A Superclass references variables can refer to a Subclass object, a concept known as upcasting. This fact is used by Java to resolve calls to overridden method dynamically

Code :

```
class A {
    void Hi() {
        S.O.P("Inside A's Hi method");
    }
}
class B extends A {
    void Hi() {
        S.O.P("Inside B's Hi method");
    }
}
class C extends A {
    void Hi() {
        S.O.P("Inside C's Hi method");
    }
}
```

```
public class DynamicDispatchDemo1
    public static void main(String args[])
        A a = new A();
        B b = new B();
        C c = new C();
        A ref;
        ref = a;
        ref.M1();
        ref = b;
        ref.M1();
        ref = c;
        ref.M1();
    }
}
```

Output:

Inside A's M1 method
Inside B's M1 method
Inside C's M1 method

Abstract class with Constructor and Concrete method

*) An abstract class is a class declared with the abstract keyword
*) Abstraction: It used to achive abstraction, focusing on what the object does instead of how it does it, by hiding implementation details
*) Code: Demostrates a like abstract class which cannot be instantiated but can contain an abstract method and a concrete method. It also shows the execution of the abstract class's constructor.

Code:

```
abstract class Bike {
    Bike() {
        S.O.P("1. Bike object is created (constructor)");
    }
    abstract void run();
    void changeGear() {
        S.O.P("3. Gear changed (Concrete method)");
    }
}
class Honda extends Bike {
    void run() {
        SO.P("2. Running Safely (abstract method)");
    }
}
Public class AbstractDemo {
    public static void main(String args[]) {
        Bike obj = new Honda();
        obj.run();
        obj.changeGear();
    }
}
```

Output:

1. Bike object is created (Constructor)
2. Running Safely (Implemented abstract method)
3. Gear Changed (Concrete method)

M T W T F S S

Page No.:
Date:
YOUVA

Final keyword

* The final keyword is a non-access modifier used to enforce constraints and create constants.
* Final variable: A variable declared as final cannot have its value changed once assigned; it remains a constant forever. Attempting to change its results in a compile-time error.
* Final method: A method declared as final cannot be overridden.
* Final class: A class declared as finale cannot be extended.

Code:

```
final class parent{
    final int speed=60;
    public final void disp(){
        System.out.println("Parent's final method. speed:"+speed);
    }
}
public class finalkeyworddemo{
    public static void main(String args[]){
        parent p=new parent();
        p.disp();
    }
}
```

Output:

Parent's final method. Speed: 60

M T W T F S S

Page No.: YOUVA

Date:

Exception Handling

*) An exception is a problem that arises during the execution of a program
When an exception occurs, the normal flow of the program is disrupted
and the program / application terminates abnormally

1. Difference between Error and exception

a) Errors indicate serious problems and abnormal conditions
that most applications should not try to handle, such as
OutOfMemoryError or JVM errors

b) Exceptions are conditions within the code that a developer can
handle and take necessary corrective actions, such as
ArithmeticException or ArrayIndexOutOfBoundsException

2. Exception Hierarchy

*) All exception classes are subtypes of the java.lang.Exception class.
The exception class is a subclass of the Throwable class

3. Keywords used in exception Handling

There are 5 keywords used in java exception handling

1. try : A try/catch block is placed around the code that might generate
an exception
2. catch : A statement following a try block that specifies the type
of exception to handle
3. Finally : A block of code that always executes, irrespective of the
occurrence of an ~~exc~~ exception. It is used for cleanup code,
such as closing connections
4. Throw : used to invoke an exception explicitly
5. Throws : used to postpone the handling of a checked exception

Code:

1) Division by zero

```
class Ex2 {
    public static void main (String [] args) {
        int d, a;
        try {
            d=0;
            a=24/d;
            System.out.println("This will not be printed");
        } catch (ArithmeticException e) {
                S.O.P ("Division by zero");
        }
        S.O.P ("After catch Statement");
    }
}
```

output:

```
Division by zero
After Catch statement
```

This demonstrates using try block of code and catch to handle runtime error

2) Predefined with finally

Code:

```
Public class Exceptest {
    public static void main (String args[]) {
        int a[] = new int[2];
        try {
            System.out.println ("Access elements three :" + a[3]);
        } Catch (ArrayIndexOutOfBounds exception e) {
            S.O.P ("Exception thrown");
        } finally {
            a[0] = 6
            S.O.P (a[0]);
        } : finally
        S.O.P ("The finally is executed");
    }
```

Output:

Exception thrown
First element : 6
The finally executed

3) Throwing user defined Exception

Code:

```
class CustomException extends Exception {
        public CustomException (String message) {
            super (message);
        }
    }
```

```
class userdefinedExceptionDemo {
    static void validateScore (int score) throws CustomException {
        if (score < 0 || score > 100) {
            throw new CustomException ("Score is out of range");
        }
        S.O.p ("Score is valid");
    }
    public static void main (String args[]) {
        try {
            validate (105);
        }
        catch (CustomException e) {
            S.O.p ("caught" : + e.getMessage());
        }
    }
}
```

Output:

caught : out of Range

I/O Streams

* A Java IO is used to process input and produce the output based on the input. Java uses the concept of a stream to make IO operation fast. A stream can be defined as a sequence of data.

The Java.io package contains all the classes required for input and output operations.

M T W T F S S

Page No.: YOUVA

Date:

Types of Streams

I -> Input
O -> output

*) Byte Streams I, used to read 8-bit (byte) from a source — file Input Stream, data input Stream

*) Byte Streams O, used to write 8-bit bytes to a destination — file output Stream, data output Stream.

*) Character Streams I, used to read 16-bit unicode characters — File Reader, Buffered Reader

*) Character Streams O, used to write 16 bit unicode characters — File writer, print writer

Code: Data input and output Streams

FIS -> File Input Stream

```
import java.io.*;
class DataStreamDemo {
    static final String file = "data.bin";
    public static void main (String[] args) throws IOException {
        try (FileOutputStream fos = new FileOutputStream(file);
             DataOutputStream data = new DataOutputStream(fos)) {
            data.writeInt(65);
            data.flush();
            S.O.P("Write success...");
        }

        try (InputStream input = new FIS(file);
             DataInputStream in = new DIS(input)) {
            int count = input.available();
            byte[] ary = new byte[count];
            in.read(ary);
            S.O.P("In Data read back (as bytes): ");
            for (byte b : ary) {
                S.O.P(b + " ");
            }
        }
    }
```

Output:

Write Success...
Data read back (as bytes) : 0 0 0 65 ...

FOS -> FileOutputStream
BOS -> BufferedOutputStream

Code: Buffered Streams

```
import java.io.*;
public class BufferedStreamExample {
    public static void main(String args[]) throws Exception {
        String filename = "fileout_buffered.txt";
        String s = "Welcome to Buffered streams";
        try(FileOutputStream fout = new FOS(filename);
            BOS bout = new BOS(fout)){
            bout.write(s.getBytes());
            bout.flush();
            S.O.S("Buffered write complete");
        }
        try{FIS fin = new FIS(filename);
            BIS bin = new BIS(fin);
            int i;
            S.O.P("Buffered read:");
            while((i = bin.read()) != -1){
                S.O.P((char) i);
            }
            S.O.P();
        }
    }
}
```

Op: Buffered write complete
Buffered Read: Welcome to buffered stream