→ Manav Mehta // 22BCE7785.
→ Assignment
→ Short Summer - 1.
→ OBJECT ORIENTED PROGRAMMING (JAVA) CSE-2005.
[→ "I have taken help from "Java the complete ref" and some online articles.]
5 June, 2024

Q.1) Explain while, do while, for and for each statement with an example program for each.

⇒ The while loop is Java's most fundamental loop statement. It repeats a block while its controlling expression is true.

General form →

Manav Mehta
Manav
22BCE7785.

```
→ while (condition) {
        // body of loop
  }
// Demonstration
class while {
      public static void main (String args[]) {
      int n=10;

      while (n>10) {
           System.out.println ("tick"+n);
           n--;
       }
      }
}
```

→ The output of above demonstrated programme will be,

```
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1.
```

## For loop

For loop is a more concise loop, when the no. of iteration is known based on given condition.
It consist of mainly three parts: initialization, condition and iteration.

→ General form of for loop.

```
for(initialization; condition; iteration) {
   // body
}
```

→ Demonstration //

```
Class ForTick {
    public static void main(String args[]) {
        int n=5;
        for(int i=1 ; i<6 ; i++){
            System.out.println("tick"+n);
        }
    }
}
```

Manav Mehta
22BCE7785

Output:

```
tick 1
tick2
tick3
tick4
tick5
```

## Do while Loop

The "do-while" loop is a control statement that allows codes to be executed on a boolean condition.
→ Unlike, the "while" loop which evaluates the condition before executing loops body, the "do-while" loop evaluates the condition after executing code.
This guarantee that code will run at least once.

→ General Form



```
do {
   // body of loop
} while (condition);
```

→ Demonstration //

```
class DoWhile {
  public static void main(String args[]) {
    int n = 5;

    do {
      System.out.println("tick" + n);
      n--;
    } while (n > 0);

  }
}
```

Manav
22BCE7785

The output of given code will be.

```
tick 5
tick 4
tick 3
tick 2
tick 1
```

#. Foreach loop

The 'foreach' loop, is also known as enhanced 'for' loop, provide a simpler and more readable way to iterate through elements in a array.

→ General form

```
for ( type element : collection) {
   // code.
}
```

→ Demonstration //

```
public class forLoop {
    public static void main(String args[]) {
        int[] array = {1,2,3,4,5};
        for(int element : array) {
            System.out.println("Element:" + element);
        }
    }
}
```

Output:

```
Element:1
Element:2
Element:3
Element:4
Element:5
```

---

2.> Explain method overloading and constructor overloading.

Method Overloading

→ In Java, it is possible to define two ore more method within the same class that share the same name, as long as their declaration parametre are different. When this is the case the methods is said to be overloaded, and the process is refered as "Method Overloading".

And when Java encounters a call to an overloaded method. It simply executes the version of method whose parametre matches the argument used in call.

```
// Demonstrating Method Overloading
class OverloadDemo{
    void test() {
        System.out.println("No parametre");
    }

    void test(int a) {
        System.out.println("a: " + a);
    }
    void test(int a, int b) {
        System.out.println("a and b are" + a + "  " + b);
```

```
    double test (double a) {
        System.out.println ("double a:" + a);
        return a*a;
    }
}

class Overload {
    public static void main (String args[]) {
        OverloadDemo ob = new OverloadDemo();
        double result;

        // Calling all version of test
        ob.test();
        ob.test(10);
        ob.test(10,20);
        result = ob.test(123.25);
        System.out.println ("Result of ob.test (123.25):" + result);
    }
}
```

Manav
22BCE7785.

Output

- No parameters
- a: 10
- a and b are 10 20
- double a: 123.25
- Result of ob.test(123.25): 15190.5625

thats what method overloading is,

## Constructor Overloading

⇒ Constructor Overloading is a feature in Java, that allows a class to have more than one constructor to different parametre. These constructor are distinguish by number, types or order of theiy parametre.

Overloading constructor provides flexibility in a way object can be created.

// Demonstration,

```
Class Box {
    double width;
    double height;
    double depth;

    Box ( double w , double h , double d) {
       width = w;
       height = h;
       depth = d;
    }

    Box ( ) {
       width = -1;
       height = -1;
       depth = -1;
    }

    Box ( double len) {
      width = height = length = len;
    }

    double volume() {
        return width * height * length;
    }

Class OverloadCons {
    public static void main ( String args[ ]) {
    // Create boxes using the various constructor.
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box();
    Box mybox3 = new Box(7);
```

```
        double vol;

        // Get volume of first box
        vol = mybox1.volume();
        System.out.println("volume of box is:" + vol);

        // Volume Second Box.
        vol = mybox2.volume();
        System.out.println("volume of mybox2:" vol);

        // Volume of mybox3.
        vol = mybox3.volume();
        System.out.println("volume of mybox3:" vol);
    }
}
```

Manav
22BCE7785

Output

```
volume of box is: 3000.0
volume of mybox2 is: -1.0
volume of mybox3 : 343.0
```

---

3.) Explain multilevel Inheritance. with an example programme.

⇒ Multilevel Inheritance refers to scenerio where a class is derived from another class which is relitself driven from another class.

→ Demonstration.

```
class Animal {
    String name;

    Animal(String name){
        this.name = name;
    }

    void eat(){
        System.out.println(name + "is eating");
    }
}.
```

```
class Mammal extends Animal {
    boolean hasfur;

    Mammal (String name, boolean hasfur) {
        super (name);
        this.fur = hasFur;
          has
    }

    void giveBirth() {
        System.out.println (name + " is giving Birth");
    }
}

class Dog extends Mammal {
    String breed;
    Dog (String name, boolean hasfur, String breed) {
        super(name, hasfur);
        this.breed = breed;
    }
    void bark() {
        System.out.println(name + " is Barking");
    }
}

// Main Class.
public class Main {
    public static void main (String args[]) {
    Dog dog = new ("Buddy", true, "Golden");
    // Accessing all method from class.
     dog.eat();
     dog.giveBirth();
     dog.bark();
```

```
5       System.out.println("Name:" + dog.name);
        }
}.
```

Output will like;

```
Buddy is eating
Buddy is giving birth
Buddy is barking.
```

Manav
22BCE7785

```
Name : Buddy
HasFur : True
Breed : Golden.
```

---

4.) Explain Abstract class and Abstract method with an example.

- An abstract class is a class that cannot be instantiated on its own and is meant to be subclass
- And methods that are declared without an implementation is known as abstract method.

| ABSTRACT CLASS | ABSTRACT METHOD. |
|---|---|
| (i) Cannot be instantiated<br>(ii) Can contain both abstract method (without a body) and non-abstract method (with a body). | (i) Declared without an implementation. (without body).<br>(ii) Must be overriden in any concrete subclass (Here, concret means i.e not abstract). |

⇒ Demonstration.

```
// Abstract class.
abstract class Animal {
    String name;
    Animal(String name) {
        this.name = name;
    }
    // Abstract method no implementation
    abstract void makeSound();
    // Non-abstract method with implementation.
    void eat() {
        System.out.println(name + " is eating");
    }
}
// Concrete Subclass of Animal.
class Dog extends Animal {
    Dog(String name) {
        super(name);
    }
    // Providing implementation for Abstract Method.
    @Override
    void makeSound() {
        System.out.print(name + " says woof!");
    }
}
Public class Main {
    public static void main(String args[]) {
        Dog dog = new Dog("Buddy");
        dog.eat();          // Calling non abstract method
        dog.makeSound();    // Calling Abstract method imp.
    }
}
```

Output will like

Buddy is eating
Buddy says woof!

---

5.) Define an interface, implement it and explain nested interface with example.

An interface in Java, is a reference type similar to a class that can contain only constants, method signature, default method, static method and nested types.

- It cannot contain instance field or constructor.
- They are a way to achieve abstraction and multiple Inheritance in Java.

* Example of an Interface and its applications.

→ Before, that general form of interface,

```
acces interface name {
        return-type method-name1(parametre-list);
        return-type method-name2(parametre-list);

        type final-varname1 = value;
        type final-varname2 = value;
        //...

        return-type method-nameN(parametre-list);
        type final-varnameN = value;
      }
```

→ Once interface has been define, one or more class can implement that.

general form,

```
class classname [extends superclass] [implement interface [, interface...]] {
  // class-body
}
```

Ex.

Code,

```
interface callback {
    void callback(int param);
}
// implementation.
class Client implements callback {
    public void callback(int p) {
        System.out.println("callback called:" + p);
    }
}
```

[→ Notice, the callback declared using public access modifier.
An interface method always declared public.]

⇒ Nested Interface

A interface can be declared a member of class or another interface. Such interface is called Nested Interface.

This differs from top level interface which must either be declared as public or use default access.

6.) Define a package and write a short package ex.

⇒ In Java, a package is a name space that organize a set of related class and interface.

→ General form of package statement,
{package pkg; }

pkg is name of package.

→ A short package example;

```
// A simple package example. (This is a short ex of package).
package mypack;
class Balance{
    String name;
    double bal;
    Balance (String n, double b){
        name = n;
        bal = b;
    }
    void show(){
        if(bal<0)
            System.out.print ("--> ");
        System.out.println (name + ": $" + bal);
    }
}
class AccountBalance {
    public static void main (String args[]) {
        Balance current[] = new Balance[2];
        current[0] = new Balance ("K.J", 123.23);
        current[1] = new Balance ("Will", 157.02);
        for(int i=0; i<2; i++) current[i].show();
    }
}
```

Manav
22BCE7785

6. Here, is an example that demonstrate a nested interface,

```
// A Nested Interface example.

// This class contains a member interface

class A {
    // this is nested Interface
    public interface NestedIF {
        boolean isNotNegative (int x);
    }
}

// B implements the nested Interface,
class B implement A.NestedIF{
    public boolean isNotNegative (int x) {
        return x<0 ? false: true;
    }
}.

class NestedIFdemo {
    public static void main ( String args[ ]) {
    // use a nested Interface ref.
        A.NestedIF nif = new B();
        if (nif.isNotNegative (10))
            System.out.println ("10 is not negative);
        if (nif. isNotNegative (-12))
            System.out. println(" this won't be display);
    }
}.
```

⇒ The above provided an example how an NESTED Interface gets implemented.

7.) Explain how multiple exception handled in java with an example programme.

⇒ In Java, multiple exceptions can be handled using a combination of 'try', 'catch' and 'finally' blocks.

- Multicatch Block

You can have several type of 'catch' block to handle different types of exceptions seperately.
Each catch block catches a specific type of exception.

(In Java 7 or later, you can use a single 'catch' block to handle multiple exceptions by seperating the exceptions with vertical bar).

→ Multiple 'catch' Blocks ex.

```
#.) public class MultipleExceptionsEx {
        public static void main (String [] args){
            try {
                int[] num = {1,2,3};
                System.out.println("Accessing element at index3:"+
                                                    number[3]);

                String text = null;
                System.out.println("Length of text: "+ text.length());
            }catch(ArrayIndexOutOfBoundsException e){
                System.out.println("Out of Bound :" + e.getMessage());
            }catch (NullPointerException e)
                System.out.println(" Null pointer exception"+e.getMessage);
            }finally {
                System.out.println(" finally Block executed");
            }

        }

}
```

Manav
22BCE7785

Output as follows:

Array index out of bound.
Finally block executed.

---

8.) Explain the throw, throws and finally keyword with an programme for each.

throw.

⇒ It is possible for your programme to throw an exception explicitly using throw keyword.

The General form of throw is given below.

```
throw ThrowableInstance;
```

Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable.

→ Primitive type such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions.

→ Two ways to you can obtain a Throwable object: using a parameter in catch clause or creating one with new operator.

Demonstration using a code.

```
// Demonstrate throw.
Class ThrowDemo {
    Static void demoproc() {
        try {
            throw new NullPointerException("demo");
        } catch(NullPointerException e) {
            System.out.println("Caught inside demoproc.");
            throw e; // rethrow the exception
        }
}
Public static void main(String args[])
        try {
            demoproc();
        } catch (NullPointerException e) {
            System.out.println("Recaught: "+e);
        }
    }
}
```

This program gets two chances to deal with same Exception.

Result: Output.

```
Caught inside demoproc
Recaught: java.lang.NullPointerException: demo.
```

## throws

If a method is causing exception that it is not able to handle, it must specify their behaviour so that callers of method can guard themselves against that exception.

- We have to do this by using throws keyword during method declaration.
- A throws clause lists the type of exception that a method might throw.

General form of method declaration that includes a throws clause.

```
type method-name (parameter-list) throws exception-list
{
    // body of method
}
```

Manav
22BCE7785

Demonstration of throws

```
class ThrowsDemo {
    static void throwOne() throws IllegalAccessException{
        System.out.println("Inside throwOne");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[]) {
        try{
            throwOne();
        }catch(IllegalException e){
            System.out.println("Caught "+e);
        }
    }
}.
```

Here, the Output generated.

```
inside throwOne
Caught java.lang.IllegalAccessException: demo.
```

## finally

The "finally" keyword ensures that certain code, always executes regardless of whether an exception occurs.

'finally' creates a block of code that will be executed after a try/catch block has completed and before the code follows a try/catch block.

- The finally block will be executed whether or not an exception is thrown.

```
// Demonstrate finally.
class FinallyDemo{
    // Throw an exception out of method.
    static void procA(){
        try{
            System.out.println("inside procA");
            throw new RuntimeException("demo");
        }finally{
            System.out.println("procA's finally");
        }
    }
    // Return from within a try block.
    static void procB(){
        try{
            System.out.println("inside procB");
            return;
        }finally{
            System.out.println("procB's finally");
        }
    }
    // Execute a try block normally
    static void procC(){
        try{
            System.out.println("inside procC.");
        }finally{
            System.out.println("procC's finally");
        }
    }
```

```
public static void main(String args[]){
    try{
      procA();
    } catch(Exception e){
      System.out.println("Exception caught");
    }
    procB();
    procC();
  }
}
```

Expl.

→ procA() prematurely breaks out by throwing an exception. The 'finally' clause is executed on the way out.
- procB() exit via a return statement.
- The finally clause is executed before procB return.
- In, procC(), the try statement executes normally without an error.

Output generated by previous code;

Manav
22BCE7785.

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

---

9.) Write a program that implements the compare() method for string that operates in reverse of normal.

→ So, we have written a program in java, that implement 'compar()' method for strings in reverse order using a custom comparator.

Code follows. //

```
import java.util.Arrays;
import java.util.Comparator;
public class ReverseStringComparator implements Comparator<String>{
    @Override.
    public int compare(String S1, String S2)
        return S2.compareTo(S1);
    }
    public static void main(String args[]) {
      // Sample array of strings
      String[] strings = {apple, orange, banana, grapes};
      Array.sort(strings, new ReverseStringComparator());

      // Print the sorted Array.
      System.out.println("String sorted in reverse order:");
      for(String S: strings)
          System.out.println(s);
      }
    }
}
```

→ Output as given.

String sorted in revers order.
orange
grape
banan
apple.

---

10.) Demonstrate iterator method with an example of programe.

```
import java.util.Iterator;
import java.util.ArrayList;

class MyIterator implements Iterator<Integer>{
    private ArrayList<Integer> data;
    private int index;

    public MyIterator(ArrayList<Integer> data){
        this.data = data;
        this.index = 0;
    }

    @Override
    public boolean hasNext(){
        return index < data.size();
    }
    @Override
    public Integer next(){
        if(hasNext()){
            return data.get(index++);
        }else{
            throw new IndexOutOfBoundException("No more in List");
        }
    }
}
public class Main{
    public static void main(String[] args){
        ArrayList<Integer> myList = new ArrayList<>();
        myList.add(1);
        myList.add(2);
        myList.add(3);
        myList.add(4);
        myList.add(5);

        MyIterator myIterator = new MyIterator(myList);
        while(my.Iterator.hasNext()){
            System.out.println(myIterator.next());
        }
    }
}
```

Manav
22BCE7785

Output of Code as follows.

1
2
3
4
5.

→ In this Java program, 'My Herator' is a class that implements 'Herator<Integer>' interface. It has hasNext() method, which return true if there are more element In data to Herate.

→ If there are no more elements. It throws an exception. IndexOutOfBoundException.

---

11.> Explain the two ways of creating a thread with an program for each.

⇒ In Java, there are two ways to create a thread.
1.> Extending the 'Thread' class.
2.> Implementing the 'Runnable' Interface.

– Extending Thread class.

```
class MyThread extend Thread {
    public void run(){
        for(int i=0; i<5; i++){
            System.out.println("Thread 1: " + i);
            try{
                Thread.sleep(1000);
            }catch(InterruptedException e){
                System.out.println(e);
            }
        }
    }
```

```
    public static void main(String args[]){
        MyThread t1 = new MyThread();
        t1.start();
    }
}
```

⇒ Output as follow.

```
Thread 1 : 0
Thread 1 : 1
Thread 1 : 2
Thread 1 : 3
Thread 1 : 4.
```

Manav
22BCE11185

Explanation.

- We created a new class 'MyThread' by extending the 'Thread' class. We override the the 'run()' method to define task of the thread.
- Inside the 'run' method, we have a loop that prints number from 0 to 4, with a delay of 1 second between each iteration.
- Then, in main method, we create an instance of 'MyThread' and start it using the start() method.

2. Implementing the Runnable Interface

```
class MyRunnable implements Runnable{
    public void run() {
        for(int i=0; i<5; i++);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println(e);
        }
    }
}
```

```
    public static void main (String[] args[]){
        MyRunnable myRunnable = new MyRunnable();
        Thread t2 = new Thread(myRunnable);
        t2.start();

    }

}.
```

⇒ In this example, we create a new class 'MyRunnable' that implement 'Runnable' Interface.
- We implement the 'run()' method to define the task of thread, which is same as previous example.
- Then, in main method we create instance of MyRunnable and pass it to 'Thread' constructor.
- Finaly, we use it by method start().

Output.

```
Thread 2: 0
Thread 2: 1
Thread 2: 2
Thread 2: 3
Thread 2: 4
```

---

12) Explain Synchoronized method & Statement with a example of each program.

In Java, Synchoronization is used to control the access of multiple thread to shared resources.
The two main synchoronization mechanism are synchoronized method and synchoronized statement.

Ex. Program.

## Synchoronized Methods

- A synchoronized method ensures that only one thredd can execute it at one time for a given instance of class.

// Demonstration.

```
class Counter {
   private int count=0;

   public synchoronized void increment(){
        count++;
   }
   public synchoronized int getCount(){
        return count;
   }
}

Class MyThread extends Thread {
      private Counter counter;

      public MyThread (Counter counter) {
        this.counter = counter;
      }
      public void run() {
        for(int i=0; i<1000; i++){
          Counter.increment();
        }
      }
}

public class SynchoronizedMethodExample {
      public static void main (String[] args) throws InterruptedException {
            Counter counter = new Counter();
```

Manav
22BCE7785

```
        MyThread t1 = new MyThread(counter);
        MyThread t2 = new MyThread(counter);

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println("Final Count: " + counter.getCount());
    }
}
```

= Output:
[ Final Count: 2000 ]

## Synchoronized Statement

Code

```
class Counter {
    private int count=0;

    Public void increment(){
        Sychoronized (this){
            count++;
        }
    }

    Public int getCount(){
        synchoronized (this){
            return count;
        }
    }
}
```

```
class MyThread extends Thread {
    private Counter counter;
    public MyThread (Counter counter) {
        this.counter = counter;
    }
    public void run() {
        for(int i=0; i<1000; i++) {
            counter.increment();
        }
    }
}
public class SynchoronizedBlockExample {
    public static void main (String args[]) throws IE {
        Counter counter = new Counter();
        MyThread t1 = new Mythread(counter);
        MyThread t2 = new MyThread(counter);

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println("Final Count: " + counter.getCount());
        }
}.
```

Manav
22BCE7785

Output

Final Count: 2000

→ Both example achieve thread-safe access `Counter` instances, ensuring that the final count is correctly incremented to 2000.

→ Synchoronized methods are simpler to use and read, while Synchoronized statement allow more control and can be more efficient in certain situations.

✗ ✗ ✗ .

Thank You.