Doc. No.: CBS(OOP/SOP/EW/8(01)

Subject: Choice Based Subject (EC218) Object Oriented Programming

EXPERIMENT NO.: 01

TITLE: Write a program in Java on branching and looping

LEARNING OBJECTIVES:

1. To learn the object oriented programming concepts
2. To make the students familiar with programs in Java for problem solving

EQUIPMENTS REQUIRED:

PC (Personal computer) with JDK/eclipse IDE

THEORY:-

What are Branching Statements in Java?

Branching statements allow the flow of execution to jump to a different part of the program. The common branching statements used within other control structures include: break, continue, and return.

Types of Branching Statement in Java:

In Java, there are three Branching Statements. They are as follows:

1. **Break Statement**
2. **Continue Statement**
3. **Return Statement**

Sample Program for break statement in Java:

```
class BreakStatementDemo
{
   public static void main(String args[])
   {
      // Initially loop is set to run from 0-9
      for (int i = 0; i < 10; i++)
      {
```

```
            // terminate loop when i is 5.
            if (i == 5)
                break;

            System.out.println("i: " + i);
        }
        System.out.println("Loop complete.");
    }
}
```

Output:

```
i: 0
i: 1
i: 2
i: 3
i: 4
Loop complete.
```

Sample Program for Continue statement in Java:

```
class ContinueStatementDemo
{
    public static void main(String args[])
    {
        for (int i = 0; i < 10; i++)
        {
            // If the number is even
            // skip and continue
            if (i%2 == 0)
                continue;

            // If number is odd, print it
            System.out.print(i + " ");
```

```
      }
   }
}
```

Output: 1 3 5 7 9

What are Looping Statements in Java?

Looping in programming languages is a feature that facilitates the execution of a set of instructions/functions repeatedly while some condition evaluates to true.

Types of looping statements in Java:

There are three types of looping statements in Java. They are as follows:

1. **For loop**
2. **While loop**
3. **Do while loop**

Sample Program for For Loop in Java:

```
class ForLoopDemo
{
   public static void main(String args[])
   {
 for(int num = 1; num <= 5; num++)
      {
        System.out.println(num);
      }
   }
}
```

Output:
1
2
3

4
5

Reference books:-

T. Budd, "Understanding OOP with Java", Pearson Education.
E. Balagurusamy, "Programming with Java A Primer", Tata McGraw Hill, (3rd Edition)
Deitel, H.M. & Deitel, "Java: How to Program", Prentice Hall (8th Ed.)

e-Resources:
1. https://nptel.ac.in/courses/106/105/106105151/

Program, Output, Conclusion:

Doc. No.: CBS(OOP/SOP/EW/8(02)
Subject: Choice Based Subject (EC218) Object Oriented Programming

EXPERIMENT NO.: 02

TITLE: Write a program in Java to implement a Calculator with operations add, subtract, multiply, divide

LEARNING OBJECTIVES:

1. To learn the object oriented programming concepts
2. To make the students familiar with programs in Java for problem solving
3. To study various java programming concept like Interface, exception handling, packages etc.

EQUIPMENTS REQUIRED:

PC (Personal computer) with JDK/eclipse IDE

THEORY:-

Class Fundamentals: The most important thing to understand about a class is that it defines a new data type. Once defined, this new type can be used to create objects of that type. Thus, a class is a *template* for an object, and an object is an *instance* of a class

The General Form of a Class: When a class is defined, we declare its exact form and nature. We do this by specifying the data that it contains and the code that operates on that data. While very simple classes may contain only code or only data, most real-world cslasses contain both. A class is declared by use of the **class** keyword. A simplified general form of a **class** definition is as shown below:

```
        class classname {
type instance-variable1;
type instance-variable2;

// ...

type instance-variableN;

type methodname1(parameter-list) {
// body of method
}
type methodname2(parameter-list) {
// body of method
}
// ...
type methodnameN(parameter-list) {
// body of method
}
}
```

The data, or variables, defined within a **class** are called *instance variables.* The code is contained within *methods.* Collectively, the methods and variables defined within a class are called *members* of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, as a general rule, it is the methods that determine how a class' data can be used. Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another. Consider simple java program.

```
public class FirstSample
{
public static void main(String[]
args)
{
```

```
System.out.println("Hello
World");
}
}
```

Program 6.2 FirstSample.java

The keyword *public* is called an *access modifier,* these modifiers control what other parts of a program can use this code. The keyword class indicates that everything in a Java program lives inside a class. Following the keyword class is the name of the class. The rules for class names in Java are quite generous. Names must begin with a letter, and after that, they can have any combination of letters and digits. The length is essentially unlimited. You cannot use a Java reserved word (such as public or if) for a class name.

As you can see in the name FirstSample, the convention is that class names are nouns that start with an uppercase letter.

We need to make the file name for the source code the same as the name of the *public* class, with the extension .java appended. Thus, we must store this code in a file called FirstSample.java. (Again, case is important—don't use firstsample.java.) If we don't do this, we'll get a error message when we try to run this source code through a Java compiler ("public class FirstSample must be defined in a file called 'FirstSample.java'"). after compilation of this source code, a file containing the bytecode for this class is generated. The Java compiler automatically names the bytecode file FirstSample .class and stores it in the same directory as the source file. Finally, run the bytecode file through the Java interpreter by issuing the command:

java FirstSample

When the program executes, it simply displays the string **'Hello World'** on the console.

Declaring Objects

Obtaining objects of a class is a two-step process. First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer* to an

object. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the **new** operator. The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by **new**.

This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.

Consider the example in which class is declared as below:

```
class Box {
double width;
double height;
double depth;
}
Box mybox = new Box();
```

This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

The first line declares **mybox** as a reference to an object of type **Box**. After this line executes, **mybox** contains the value **null**, which indicates that it does not yet point to an actual object. Any attempt to use **mybox** at this point will result in a compile-time error. The next line allocates an actual object and assigns a reference to it to **mybox**. After the second line executes, you can use **mybox** as if it were a **Box** object. But in reality, **mybox** simply holds the memory address of the actual **Box** object.

Reference books:-

T. Budd, "Understanding OOP with Java", Pearson Education.
E. Balagurusamy, "Programming with Java A Primer", Tata McGraw Hill, (3rd Edition)
Deitel, H.M. & Deitel, "Java: How to Program", Prentice Hall (8th Ed.)

e-Resources:
1. https://nptel.ac.in/courses/106/105/106105151/

Program, Output, Conclusion:

Doc. No.: CBS(OOP/SOP/EW/8(03)
Subject: Choice Based Subject (EC218) Object Oriented Programming

EXPERIMENT NO.: 03

TITLE: Write a program in Java to sort i) List of integers ii) List of names

LEARNING OBJECTIVES:

1. To learn the object oriented programming concepts
2. To make the students familiar with programs in Java for problem solving

EQUIPMENTS REQUIRED:

PC (Personal computer) with JDK/eclipse IDE

THEORY:-

Arrays:

An *array* is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

One-Dimensional Arrays

A *one-dimensional array* is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type. The general form of a one-dimensional array declaration is

***type var-name*[ ];**

Here, *type* declares the base type of the array. The base type determines the data type of each Element that comprises the array. Thus, the base type for the array determines what type of

data the array will hold. For example, the following declares an array named **month_days** with the type "array of int":

int month_days[];

Although this declaration establishes the fact that **month_days** is an array variable, no array actually exists. actually, the value of **month_days** is set to **null**, which represents an array with no value. To link **month_days** with an actual, physical array of integers, you must allocate one using **new** and assign it to **month_days**. **new** is a special operator that allocates memory. You will look more closely at **new** in a later chapter, but you need to use it now to allocate memory for arrays. The general form of **new** as it applies to one-dimensional arrays appears as follows:

array-var = new type[size];

Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array, and *array-var* is the array variable that is linked to the array. That is, to use **new** to allocate an array, you must specify the type and number of elements to allocate. The elements in the array allocated by **new** will automatically be initialized to zero. This example allocates a 12-element array of integers and links them to **month_days**.

month_days = new int[12];

After this statement executes, **month_days** will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero. obtaining an array is a two-step process. First, you must declare a variable of the desired array type. Second, you must allocate the memory that will hold the array, using **new**, and assign it to the array variable. Thus, in Java all arrays are dynamically allocated. Once you have allocated an array, you can access a specific element in the array by specifying its index within square brackets. All array indexes start at zero. For example, this statement assigns the value 28 to the second element of **month_days**.

month_days[1] = 28;

The next line displays the value stored at index 3.

System.out.println(month_days[3]);

Putting together all the pieces, here is a program that creates an array of the number of days in each month.

```
// Demonstrate a one-dimensional array.
class Array {
public static void main(String args[])
{
int month_days[];
month_days = new int[12];
month_days[0] = 31;
month_days[1] = 28;
month_days[2] = 31;
month_days[3] = 30;
month_days[4] = 31;
month_days[5] = 30;
month_days[6] = 31;
month_days[7] = 31;
month_days[8] = 30;
month_days[9] = 31;
month_days[10] = 30;
month_days[11] = 31;
System.out.println("April has " + month_days[3] + " days.");
}
```

```
}
```

Program 3.1 ArrayDemo.java

When you run this program, it prints the number of days in April. As mentioned, Java array indexes start with zero, so the number of days in April is **month_days[3]** or 30. It is possible to combine the declaration of the array variable with the allocation of the array itself, as shown here:

int month_days[] = new int[12];

This is the way that you will normally see it done in professionally written Java programs. Arrays can be initialized when they are declared. The process is much the same as that used to initialize the simple types. An *array initializer* is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array initializer. There is no need to use **new**.

String:

In Java a *string* is a sequence of characters. But, unlike many other languages that implement strings as character arrays, Java implements strings as objects of type String. The String class supports several constructors. To create an empty String, the default constructor is to be called. For example, ***String s = new String();*** will create an instance of String with no characters in it. Frequently, you will want to create strings that have initial values. The String class provides a variety of constructors to handle this. To create a String initialized by an array of characters, use the constructor shown here:

String(char name[ ])
char name[] = { 'a', 'b', 'c' };
String s = new String(name);

This constructor initializes **s** with the string "abc". You can construct a **String** object that contains the same character sequence as another String object using this constructor: ***String(String strObj)***

Here, *strObj* is a String object. Consider this example:

```
// Construct one String from another.
class MakeString {
public static void main(String args[]) {
char c[] = {'J', 'a', 'v', 'a'};
String s1 = new String(c);
String s2 = new String(s1);
System.out.println(s1);
System.out.println(s2);
}
```

Program 3.2 MakeString.java

Reference books:-

1. T. Budd, Understanding OOP with Java, Pearson Education.
2. E Balagurusamy, Programming with Java A Primer, Tata McGraw Hill, 3rd Edition.
3. Herbert Schildt , "Java : The Complete Reference" Tata McGraw-Hill (7th Edition)

e-Resources:

1. https://nptel.ac.in/courses/106/105/106105151/

Program, Output, Conclusion:

Doc. No.: CBS(OOP/SOP/EW/8(04)
Subject: Choice Based Subject (EC218) Object Oriented Programming

EXPERIMENT NO.: 04

TITLE: Write a program in Java to create class with members and methods, accept and display details for single object.

LEARNING OBJECTIVES:

1. To learn the object oriented programming concepts
2. To make the students familiar with programs in Java for problem solving

EQUIPMENTS REQUIRED:

PC (Personal computer) with JDK/eclipse IDE

THEORY:-

What is an object in Java

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

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

Optional Object Definitions:

- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.

- The object is *an instance of a class*.

An object has three characteristics:

- **State:** represents the data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw,etc.
- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

What is a class in Java

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

- Fields, Methods, Constructors, Blocks, Nested class and interface

Syntax to declare a class:

```
class <class_name>{

   field;

   method;

}
```

Instance variable in Java

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

Advantage of Method

- Code Reusability
- Code Optimization

new keyword in Java

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

Object and Class Example: main within the class

```
//Java Program to illustrate how to define a class and fields

//Defining a Student class.

class Student{

 //defining fields

 int id;//field or data member or instance variable

 String name;

 //creating main method inside the Student class

 public static void main(String args[]){

  //Creating an object or instance

  Student s1=new Student();//creating an object of Student

  //Printing values of the object

  System.out.println(s1.id);//accessing member through reference variable
```

```
    System.out.println(s1.name);
  }
}
```

Reference books:-

1. T. Budd, Understanding OOP with Java, Pearson Education.
2. E Balagurusamy, Programming with Java A Primer, Tata McGraw Hill, 3rd Edition.
3. Herbert Schildt , "Java : The Complete Reference" Tata McGraw-Hill (7th Edition)

e-Resources:

1. https://nptel.ac.in/courses/106/105/106105151/

Program, Output, Conclusion:

Doc. No.: CBS(OOP/SOP/EW/8(05)
Subject: Choice Based Subject (EC218) Object Oriented Programming

EXPERIMENT NO.: 05

TITLE: Write a program in Java to pass object as an argument and to return the object.

LEARNING OBJECTIVES:

1. To learn the object oriented programming concepts
2. To make the students familiar with programs in Java for problem solving

EQUIPMENTS REQUIRED:

PC (Personal computer) with JDK/eclipse IDE

THEORY:-

Method in Java

In general, a **method** is a way to perform some task. Similarly, the **method in Java** is a collection of instructions that performs a specific task. It provides the reusability of code.

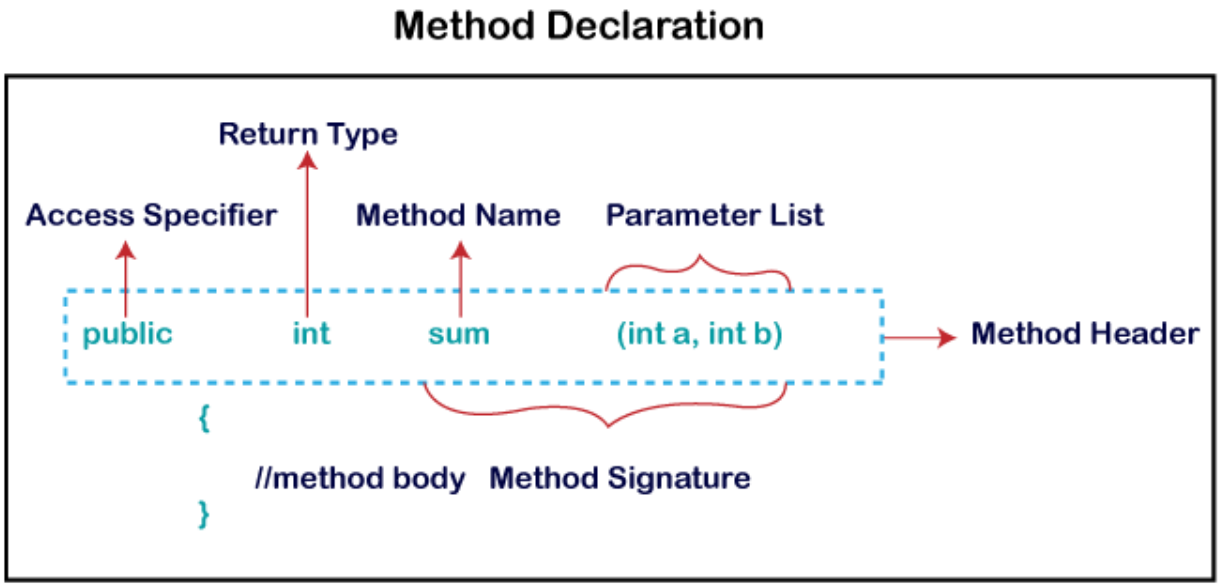
What is a method in Java?

A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the **reusability** of code. We write a method once and use it many times. We do not require to write code again and again. It also provides the **easy modification** and **readability** of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it.

The most important method in Java is the **main()** method.

Method Declaration

The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments. It has six components that are known as **method header**, as we have shown in the following figure.



Method Signature: Every method has a method signature. It is a part of the method declaration. It includes the **method name** and **parameter list**.

Access Specifier: Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides **four** types of access specifier:

- **Public:** The method is accessible by all classes when we use public specifier in our application.
- **Private:** When we use a private access specifier, the method is accessible only in the classes in which it is defined.
- **Protected:** When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.

- **Default:** When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

Return Type: Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.

Method Name: It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be **subtraction().** A method is invoked by its name.

Parameter List: It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.

Method Body: It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

Naming a Method

While defining a method, remember that the method name must be a **verb** and start with a **lowercase** letter. If the method name has more than two words, the first name must be a verb followed by adjective or noun. In the multi-word method name, the first letter of each word must be in **uppercase** except the first word.

Reference books:-

1. T. Budd, "Understanding OOP with Java", Pearson Education.
2. E. Balagurusamy, "Programming with Java A Primer", Tata McGraw Hill, (3rd Edition)
3. Deitel, H.M. & Deitel, "Java: How to Program", Prentice Hall (8th Ed.)

e-Resources:

1. https://nptel.ac.in/courses/106/105/106105151/

Program, Output, Conclusion:

Doc. No.: CBS(OOP/SOP/EW/8(05)
Subject: Choice Based Subject (EC218) Object Oriented Programming

EXPERIMENT NO.: 06

TITLE: Write a program in Java on constructor and constructor overloading

LEARNING OBJECTIVES:

1. To learn the object oriented programming concepts
2. To make the students familiar with programs in Java for problem solving
3. To study various java programming concept like Interface, exception handling, packages etc.

EQUIPMENTS REQUIRED: PC (Personal computer) with JDK/eclipse IDE

THEORY:- In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.

It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

Rules for creating Java constructor

There are two rules defined for the constructor.

1. Constructor name must be the same as its class name

2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

Example of default constructor

```
//Java Program to create and call a default constructor

class Bike1{

//creating a default constructor

Bike1(){System.out.println("Bike is created");}

//main method

public static void main(String args[]){

//calling a default constructor

Bike1 b=new Bike1();

}

}
```

Example of parameterized constructor

```
//Java Program to demonstrate the use of the parameterized constructor.

class Student4{

   int id;

   String name;
```

```
    //creating a parameterized constructor
    Student4(int i,String n){
    id = i;
    name = n;
    }
    //method to display the values
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    //creating objects and passing values
    Student4 s1 = new Student4(111,"Karan");
    Student4 s2 = new Student4(222,"Aryan");
    //calling method to display the values of object
    s1.display();
    s2.display();
   }
}
```

Example of Constructor Overloading

```
//Java program to overload constructors
class Student5{
   int id;
   String name;
   int age;
   //creating two arg constructor
   Student5(int i,String n){
   id = i;
   name = n;
   }
   //creating three arg constructor
   Student5(int i,String n,int a){
   id = i;
   name = n;
   age=a;
   }
   void display(){System.out.println(id+" "+name+" "+age);}
```

```
    public static void main(String args[]){

    Student5 s1 = new Student5(111,"Karan");

    Student5 s2 = new Student5(222,"Aryan",25);

    s1.display();

    s2.display();

   }

}
```

Reference books:-

1. T. Budd, "Understanding OOP with Java", Pearson Education.
2. E. Balagurusamy, "Programming with Java A Primer", Tata McGraw Hill, (3rd Edition)
3. Deitel, H.M. & Deitel, "Java: How to Program", Prentice Hall (8th Ed.)

e-Resources:

1. https://nptel.ac.in/courses/106/105/106105151/

Program, Output, Conclusion:

Doc. No.: CBS(OOP/SOP/EW/8(07)
Subject: Choice Based Subject (EC218) Object Oriented Programming

EXPERIMENT NO.: 07

TITLE: Write a java program which use try and catch for exception handling

LEARNING OBJECTIVES:

1. To learn the object oriented programming concepts
2. To make the students familiar with programs in Java for problem solving
3. To study various java programming concept like Interface, exception handling, packages etc.

EQUIPMENTS REQUIRED: PC (Personal computer) with JDK/eclipse IDE

THEORY:-

An *exception* is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error. In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on. This approach is as cumbersome as it is troublesome. Java's exception handling avoids these problems and, in the process, brings run-time error management into the object-oriented world.

Exception-Handling Fundamentals:

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and

processed. Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.

Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. Briefly, here is how they work. Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using **catch**) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword **throw**. Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that absolutely must be executed before a method returns is put in a **finally** block.

The general form of an exception-handling block is given here:

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed before try block ends
}
```

Exception Types:

All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy. Immediately below **Throwable** are two subclasses that partition exceptions into two distinct branches. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of **Exception**, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error. This chapter will not be dealing with exceptions of type **Error**, because these are typically created in response to catastrophic failures that cannot usually be handled by your program.

When the Java run-time system detects the attempt to divide by zero, it constructs anew exception object and then *throws* this exception. This causes the execution of **Exc** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately. In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system. Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program. Here is the output generated when this example is executed.

Using try and catch:

Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself. Doing so provides two benefits. First, it allows you to fix the error. Second, it prevents the program from automatically terminating.

Most users would be confused (to say the least) if your program stopped running and printed a stack trace whenever an error occurred! Fortunately, it is quite easy to prevent this. To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a **try** block. Immediately following the **try** block, include a **catch** clause that specifies the exception type that you wish to catch. To illustrate how easily this can be done, the following program includes a **try** block and a **catch** clause which processes the **ArithmeticException** generated by the division-by-zero error:

```
class Exc2 {
public static void main(String args[]) {
int d, a;
try { // monitor a block of code.
    d = 0;
    a = 42 / d;
    System.out.println("This will not be printed.");
} catch (ArithmeticException e) { // catch divide-by-zero error
    System.out.println("Division by zero.");
}
    System.out.println("After catch statement.");
}
}
```

Notice that the call to **println( )** inside the **try** block is never executed. Once an exception is thrown, program control transfers out of the **try** block into the **catch** block. Put differently, **catch** is not "called," so execution never "returns" to the **try** block from a **catch**. Thus, the line "This will not be printed." is not displayed. Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try**/**catch** mechanism.

try and its **catch** statement form a unit. The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement. A **catch** statement cannot catch an exception thrown by another **try** statement (except in the case of nested **try** statements,

described shortly). The statements that are protected by **try** must be surrounded by curly braces. (That is, they must be within a block.) You cannot use **try** on a single statement.

The goal of most well-constructed **catch** clauses should be to resolve the exceptional condition and then continue on as if the error had never happened. For example, in the next program each iteration of the **for** loop obtains two random integers. Those two integers are divided by each other, and the result is used to divide the value 12345. The final result is put into **a**. If either division operation causes a divide-by-zero error, it is caught, the value of **a** is set to zero, and the program continues.

Multiple catch Clauses:

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception.

```
// Demonstrate multiple catch statements.
class MultiCatch {
public static void main(String args[]) {
try {
int a = args.length;
System.out.println("a = " + a);
int b = 42 / a;
int c[] = { 1 };
c[42] = 99;
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
}
}
```

This program will cause a division-by-zero exception if it is started with no command line parameters, since **a** will equal zero. It will survive the division if you provide a command-line

argument, setting **a** to something larger than zero. But it will cause an **ArrayIndexOutOfBoundsException**, since the **int** array **c** has a length of 1, yet the program attempts to assign a value to **c[42]**.

When you use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass. Further, in Java, unreachable code is an error.

Nested try Statements:

The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**.

finally: When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code t hatcloses the file to be bypassed by the exception-handling mechanism. The **finally** keyword is designed to address such situations. **finally** creates a block of code that will be executed after a **try**/**catch** block has completed and before the code following the **try/catch** block.

Reference books:-

1. T. Budd, "Understanding OOP with Java", Pearson Education.
2. E. Balagurusamy, "Programming with Java A Primer", Tata McGraw Hill, (3rd Edition)
3. Deitel, H.M. & Deitel, "Java: How to Program", Prentice Hall (8th Ed.)

e-Resources:

1. https://nptel.ac.in/courses/106/105/106105151/

Program, Output, Conclusion:

Doc. No.: CBS(OOP/SOP/EW/8(08)
Subject: Choice Based Subject (EC218) Object Oriented Programming

EXPERIMENT NO.: 08

TITLE: Write a Program in java on interface demonstrating concept of multiple inheritance

LEARNING OBJECTIVES:

1. To learn the object oriented programming concepts
2. To make the students familiar with programs in Java for problem solving
3. To study various java programming concept like Interface, exception handling, packages etc.

EQUIPMENTS REQUIRED:

PC (Personal computer) with JDK/eclipse IDE

THEORY:-

Inheritance: Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a super class. The class that does the inheriting is called a *subclass.* Therefore, a subclass is a specialized version of a super class. It inherits all of the instance variables and a method defined by the superclass and adds its own, unique elements. To inherit a class, you simply incorporate the definition of one class into another by using extends keyword. To see how, let's begin with a short example. The following program creates a super class called A and a subclass called B. Notice how the keyword extends is used to create a sub class of **A**.

```
class A {
int i, j;
void showij() {
System.out.println("i and j: " + i + " " +
j);
}
}
// Create a subclass by extending class
A.
class B extends A {
int k;
void showk() {
System.out.println("k: " + k);
}
void sum() {
System.out.println("i+j+k: " + (i+j+k));
}
}
```

Program for DemoInheritance.java

The general form of a **class** declaration that inherits a superclass is shown here:

class subclass-name extends superclass-name {
// body of class
}

Although a subclass includes all of the members of its super class, it cannot access those members of the super class that have been declared as **private**. For example, consider the following simple class hierarchy:

```
class A {
int i;
private int j;
void setij(int x, int y) {
i = x;
j = y;
}
}
class B extends A {
int total;
void sum() {
total = i + j; // ERROR, j is not accessible
here
}
}
class Access {
public static void main(String args[]) {
B subOb = new B();
subOb.setij(10, 12);
subOb.sum();
System.out.println("Total is " +
subOb.total);
```

```
}
}
```

Program for Access.java

This program will not compile because the reference to j inside the *sum*( ) method of B causes an access violation. Since j is declared as private, it is only accessible by other members of its own class. Subclasses have no access to it.

Using super: Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword super.

super has two general forms. The first calls the superclass' constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass.

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

A subclass can call a constructor defined by its superclass by use of the following form of super:
super(arg-list);

Here, *arg-list* specifies any arguments needed by the constructor in the superclass. *super***( )** must always be the first statement executed inside a subclass' constructor. To see how *super***( )** is used, consider this improved version of the *BoxWeight***( )** class:

```
class BoxWeight extends Box {
double weight;
```

```
BoxWeight(double w, double h, double d,
double m) {
super(w, h, d);
weight = m;
}
}
```

Here, *BoxWeight*( ) calls *super*( ) with the arguments w, h, and d. This causes the *Box*( ) constructor to be called, which initializes width, height, and depth using these values. *BoxWeight* no longer initializes these values itself. It only needs to initialize the value unique to it: weight. This leaves Box free to make these values private if desired.

The second form of super acts somewhat like this, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form super.*member*

Here, *member* can be either a method or an instance variable. This second form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass. Consider this simple class hierarchy:

```
class A {
int i;
}
class B extends A {
int i;
B(int a, int b) {
super.i = a;
i = b;
}
```

```
void show() {
System.out.println("i in superclass: " + super.i);
System.out.println("i in subclass: " + i);
}
}
class UseSuper {
public static void main(String args[]) {
B subOb = new B(1, 2);
subOb.show();
}
}
```

Program for Demosuper.java

Although the instance variable i in B hides the i in A, super allows access to the i defined in the superclass. in a class hierarchy, constructors are called in order of derivation, from superclass to subclass. Further, since super( ) must be the first statement executed in a subclass' constructor, this order is the same whether or not super( ) is used. If super ( ) is not used, then the default or parameterless constructor of each superclass will be executed. The following program illustrates when constructors are executed:

```
class A {
A() {
System.out.println("Inside A's constructor.");
```

```
}
}
class B extends A {
B() {
System.out.println("Inside B's
constructor.");
}
}
// Create another subclass by extending B.
class C extends B {
C() {
System.out.println("Inside C's
constructor.");
}
}
class CallingCons {
public static void main(String args[]) {
C c = new C();
}
}
```

Program for SuperConstruct.java

Reference books:-

1. T. Budd, "Understanding OOP with Java", Pearson Education.
2. E. Balagurusamy, "Programming with Java A Primer", Tata McGraw Hill, (3rd Edition)
3. Deitel, H.M. & Deitel, "Java: How to Program", Prentice Hall (8th Ed.)

e-Resources:

1. https://nptel.ac.in/courses/106/105/106105151/

Program, Output, Conclusion:

Prepared by

(Mr. N. I. Bhopale)
Subject Teacher

Approved by

(Dr. B. S. Agarkar)
H.O.D. ECE Dept