Object-Oriented Programming (OOPS-2)

What you will learn in this lecture?

- Components of OOPs.
- Access modifiers with inheritance and protected modifiers.
- All about exception handling.

Encapsulation

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Another way to think about encapsulation is, it is a protective shield that prevents the data from being accessed by the code outside this shield.

- Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of its own class in which they are declared.
- As in encapsulation, the data in a class is hidden from other classes using the data hiding concept which is achieved by making the members or methods of class as private and the class is exposed to the end user or the world without providing any details behind implementation using the abstraction concept, so it is also known as combination of data-hiding and abstraction..

- Encapsulation can be achieved by: Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.

Inheritance

- Inheritance is a powerful feature in Object-Oriented Programming.
- Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance, the information is made manageable in a hierarchical order.
- The class which inherits the properties of the other is known as **subclass** *(derived class or child class)* and the class whose properties are inherited is known as **superclass** *(base class, parent class)*.

Super Keyword:

The super keyword in Java is a reference variable which is used to refer to an immediate parent class object.

Whenever you create an instance of a subclass, an instance of the parent class is created implicitly which is referred to by a super reference variable.

Let us take a real-life example to understand inheritance. Let's assume that **Human** is a class that has properties such as **height**, **weight**, **age**, etc and functionalities (or methods) such as **eating()**, **sleeping()**, **dreaming()**, **working()**, etc.

Now we want to create **Male** and **Female** classes. Both males and females are humans and they share some common properties (like **height**, **weight**, **age**, etc) and behaviors (or functionalities like **eating()**, **sleeping()**, etc), so they can inherit these properties and functionalities from the **Human** class. Both males and females also have some characteristics specific to them (like men have short hair and females have long hair). Such properties can be added to the **Male** and **Female** classes separately.

This approach makes us write less code as both the classes inherited several properties and functions from the superclass, thus we didn't have to re-write them. Also, this makes it easier to read the code.

Java Inheritance Syntax

```
class SuperClass{
    // Body of parent class
}

class SubClass extends SuperClass{
    // Body of derived class
}
```

To inherit properties of the parent class, **extends** keyword is used followed by the name of the parent class.

Example of Inheritance in Java

To demonstrate the use of inheritance, let us take an example.

A polygon is a closed figure with 3 or more sides. Say, we have a class called Polygon defined as follows.

```
class Polygon{
    int n;
    int[] sides;
```

```
    public Polygon(int no_of_sides){ //Constructor
        this.n = no_of_sides;
        this.sides = new int[no_of_sides];
    }

    void inputSides(){ //Take user input for side lengths
       Scanner s = new Scanner(System.in);
       for (int i=0; i<this.n; i++){
           System.out.println("Enter side: ");
           this.sides[i] = s.nextInt();
       }
    }

    void displaySides(): //Print the sides of the polygon
        for (int i=0; i<this.n; i++){
            System.out.println("Side " + i+1 +" is" + this.sides[i]);
        }
}
```

This class has **data attributes** to store the number of sides **n** and magnitude of each side as a list called **sides**.

The `inputSides()` method takes in the magnitude of each side and `dispSides()` displays these side lengths.

Now, a triangle is a polygon with 3 sides. So, we can create a class called **Triangle** which inherits from **Polygon**. In other words, we can say that every triangle is a polygon. This makes all the attributes of the **Polygon** class available to the **Triangle** class.

Constructor in Subclass

The constructor of the subclass must call the constructor of the superclass using super keyword:

```
super.Polygon(<Parameter1>,<Parameter2>,...)
```

Note: The parameters being passed in this call must be the same as the parameters being passed in the superclass' constructor/ function, otherwise it will throw an error.

The **Triangle** class can be defined as follows.

```
class Triangle extends Polygon{
    public Triangle(){
        super.Polygon(3); //Calling constructor of superclass
    }

    void findArea(){
        int a = super.sides[0];
        int b = super.sides[1];
        int c = super.sides[2];
        // calculate the semi-perimeter
        int s = (a + b + c) / 2;
        int area = Math.sqrt(s*(s-a)*(s-b)*(s-c));
        print('The area of the triangle is ' + area);
    }
}
```

However, the class **Triangle** has a new method `findArea()` to find and print the area of the triangle. This method is only specific to the **Triangle** class and not **Polygon** class.

Here, even though we did not define methods like `inputSides()` or `displaySides()` for class Triangle separately, we will be able to use them. If an attribute is not found in the subclass itself, the search continues to the superclass.

Access Modifiers

Various object-oriented languages like C++, Java, Python control access modifications which are used to restrict access to the variables and methods of the class. There are four types of access modifiers available in java, which are **Public**, **Private,** and **Protected** in a class, then there is a **default** case (we don't write any keyword in this case), which lies somewhere in between public and private.

Public Modifier

The public access modifier is specified using the keyword public.

- The public access modifier has the widest scope among all other access modifiers.
- Classes, methods, or data members that are declared as public are accessible from everywhere in the program. There is no restriction on the scope of public data members.

Consider the given example:

```
// Package 1
public class Student{
    public String name; // public member
    public int age;  // public member

     // constructor
    public void Student(String name, int age){
        this.name = name;
        this.age = age;
    }
}

-----------------------------------------------------------------------

// Package 2
```

```
class Test{
    public static void main(String[] args) {
        Student obj = Student("Boy", 15)
        System.out.println(obj.age); //calling public member of class
        System.out.println(obj.name); //calling public member
    }
}
```

We will get the output as:

```
10
Boy
```

We will be able to access both **name** and **age** of the object from outside the class and package as they are **public**. However, this is not a good practice due to *security concerns.*

Private Modifier

The members of a class that are declared **private** are accessible within the class only. A private access modifier is the most secure access modifier. Data members of a class are declared private by adding a private keyword before the data member of that class. Consider the given example:

```
// Package 1
public class Student{
    private String name; // private member
    public int age;  // public member

     // constructor
    public void Student(String name, int age){
        this.name = name;
        this.age = age;
    }
}
----------------------------------------------------------------------
```

```
// Package 2
class Test{
    public static void main(String[] args) {
        Student obj = Student("Boy", 15)
        System.out.println(obj.age); //calling public member of class
        System.out.println(obj.name); //calling private member
    }
}
```

We will get the output as:

```
10
AttributeError: 'Student' object has no attribute 'name'
```

We will get an **AttributeError** when we try to access the **name** attribute. This is because **name** is a **private** attribute and hence it cannot be accessed from outside the class.

Note: *We can even have* ***public*** *and* ***private*** *methods.*

Private and Public modifiers with Inheritance

- The subclass will be able to access any **public** method or instance attribute of the superclass.
- The subclass will not be able to access any **private** method or instance attribute of the superclass.

Protected Modifier

The members of a class that are declared protected are only accessible to a class derived from it. Data members of a class are declared **protected** by adding a protected keyword before the data member of that class.

The given example will help you get a better understanding:

```
// superclass
public class Student{
    protected String name; // private member

     // constructor
    public void Student(String name){
        this.name = name;
    }
}
```

This is the parent class **Student** with a **protected** instance attribute **name**. Now consider a subclass of this class:

```
class Display extends Student{
       // constructor
       public Display(String name){
                super.Student(name);
       }
       public void displayDetails(){
                // accessing protected data members of the superclass
                System.out.println("Name: ", super.name);
       }
}

class Test{
    public static void main(String[] args) {
        Display obj = Student("Boy");   // creating objects of the
                                        // derived class
        obj.displayDetails();     // calling public member functions
                                  // of the class
        System.out.println(obj.name);    // trying to access
                                         // protected attribute
    }
}
```

This class **Display** inherits the **Student** class. The method `displayDetails()` accesses the **protected** attribute **_name**. Further, we try to access it again outside this class.

Output:

```
Name: Boy
AttributeError: 'Display' object has no attribute 'name'
```

You can observe that we were able to access the **protected** attribute **_name** from inside the `displayDetails()` method in the subclass. However, we were not able to access it outside the subclass and we got an **AttributeError**. This justifies the definition of the **protected** modifier.

Polymorphism

Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word “poly” means many and “morphs” means forms, So it means many forms.

In Java polymorphism is mainly divided into two types:

- Compile time Polymorphism
- Runtime Polymorphism

1. **Compile-time polymorphism**: It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading. But **Java supports the Operator Overloading with only the ‘+’ symbol**. ‘+’ symbol in java works for adding two integer numbers and it can also be used for string concatenation.

Function/ Method Overloading: When there are multiple functions with the same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments.

Example 1: Polymorphism in addition(+) operator

We know that the **+** operator is used extensively in Java programs. But, it does not have a single usage. For integer data types, the **+** operator is used to perform arithmetic addition operation.

```
int num1 = 1;
int num2 = 2;
System.out.println(num1+num2);
```

Hence, the above program outputs **3**.

Similarly, for string data types, the **+** operator is used to perform concatenation.

```
String str1 = "Java"
String str2 = "Programming"
print(str1+" "+str2)
```

As a result, the above program outputs `"Java Programming"`.

Here, we can see that a single operator + has been used to carry out different operations for distinct data types. This is one of the most simple occurrences of **polymorphism** in Python.

Example 2: Polymorphism with methods/ functions in Java

Let's look at an example.

```
// Java program for Method overloading
```

```
class MultiplyFun {
    // Method with 2 parameter
    static int Multiply(int a, int b){
        return a * b;
    }
    // Method with the same name but 3 parameter
    static int Multiply(int a, int b, int c){
        return a * b * c;
    }
}

class Test{
    public static void main(String[] args) {
        System.out.println(MultiplyFun.Multiply(2, 4));
        System.out.println(MultiplyFun.Multiply(2, 7, 3));
    }
}
```

Output

```
8
42
```

2. **Runtime Polymorphism:** It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding.

   It occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

   Let us see see this in code:

```
// Java program for Method overriding

class Parent {
      void Print() {
            System.out.println("parent class");
      }
```

```
}

class subclass1 extends Parent {
	void Print() {
		System.out.println("subclass1");
	}
}

class subclass2 extends Parent {
	void Print() {
		System.out.println("subclass2");
	}
}

class TestPolymorphism3 {
	public static void main(String[] args) {
		Parent a;

		a = new subclass1();
		a.Print();

		a = new subclass2();
		a.Print();
	}
}
```

Output

```
subclass1
subclass2
```

Exception Handling

Error in Java can be of two types i.e. normal unavoidable errors and Exceptions.

- Errors are the problems in a program due to which the program will stop the execution.
- On the other hand, exceptions are raised when some internal events occur which changes the normal flow of the program.

Difference between Syntax Errors and Exceptions

Error: An Error "indicates serious problems that a reasonable application should not try to catch."

Both Errors and Exceptions are the subclasses of java.lang.Throwable class. Errors are the conditions which cannot get recovered by any handling techniques. It surely causes termination of the program abnormally. Errors belong to unchecked type and mostly occur at runtime. Some of the examples of errors are Out of memory error or a System crash error. Also, there are syntax errors that are caused by the wrong syntax in the code. It leads to the termination of the program in compile time itself.

Example:

When you are using recursion to solve any problem, you must have seen errors which say "Stack overflow". In your case, this might have arised due to the incorrect or absence of base case. But this has a deeper explanation. This stack overflow error may also arise when the input is huge and to solve the problem you need too many recursive calls one above the other, this will lead to overflow of the main stack space provided. So there comes the need to solve this problem iteratively. You will practically experience these errors in Dynamic Programming lecture.

For a 64 bits Java 8 program with minimal stack usage, the maximum number of nested method calls is about 7 000. Generally, we don't need more, except in very specific cases. You can

Exceptions: Exceptions are raised when the program is syntactically correct but the code resulted in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.

Example:

```
int marks = 10000;
int a = marks / 0;
System.out.println(a);
```

Output:

```
ZeroDivisionError: division by zero
```

The above example raised the **ZeroDivisionError** exception, as we are trying to divide a number by 0 which is not defined.

Exceptions in Java

- Java has many built-in exceptions that are raised when your program encounters an error (something in the program goes wrong).
- When these exceptions occur, the Java interpreter stops the current process and passes it to the calling process until it is handled.
- If not handled, the program will crash.
- For example, let us consider a program where we have a function A that calls function B, which in turn calls function C. If an exception occurs in function C but is not handled in C, the exception passes to B and then to A.
- If never handled, an error message is displayed and the program comes to a sudden unexpected halt.

Some Common Exceptions

A list of common exceptions that can be thrown from a standard Java program is given below.

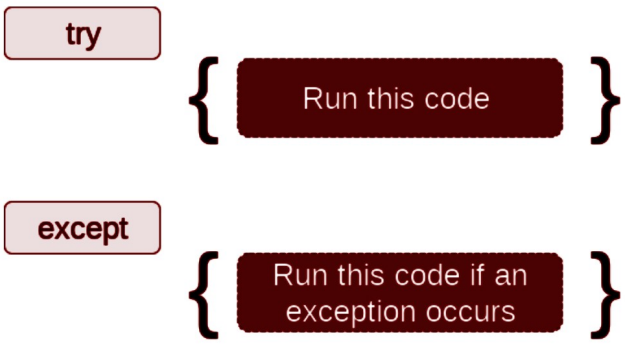
- **ArithmeticException**
  It is thrown when an exceptional condition has occurred in an arithmetic operation.
- **ArrayIndexOutOfBoundsException**
  It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
- **ClassNotFoundException**
  This Exception is raised when we try to access a class whose definition is not found
- **FileNotFoundException**
  This Exception is raised when a file is not accessible or does not open.
- **IOException**
  It is thrown when an input-output operation failed or interrupted
- **InterruptedException**
  It is thrown when a thread is waiting , sleeping , or doing some processing, and it is interrupted.
- **NoSuchFieldException**
  It is thrown when a class does not contain the field (or variable) specified
- **NoSuchMethodException**
  It is thrown when accessing a method which is not found.
- **NullPointerException**
  This exception is raised when referring to the members of a null object. Null represents nothing

- **NumberFormatException**
  This exception is raised when a method could not convert a string into a numeric format.
- **RuntimeException**
  This represents any exception which occurs during runtime.
- **StringIndexOutOfBoundsException**
  It is thrown by String class methods to indicate that an index is either negative than the size of the string

Catching Exceptions

In Java, exceptions can be handled using `try-catch` blocks.

- If the Java program contains suspicious code that may throw the exception, we must place that code in the `try` block.
- The `try` block must be followed by the `catch` statement, which contains a block of code that will be executed in case there is some exception in the `try` block.
- We can thus choose what operations to perform once we have caught the exception.



- Here is a simple example:

```
int[] arr = {1, 0, 2};
for (int ele : arr){
    try{ //This block might raise an exception while executing
        System.out.println("The entry is" + ele);
        int r = 1/int(ele);
    }
    catch(Exception e) { //This block executes in case of an
                         // exception in "try"
        System.out.println("Oops! An error occurred: "+e.toString());
    }
    System.out.println();
}
```

We get the output to this code as:

```
The entry is 1

The entry is 0
Oops! An error occurred: java.lang.ArithmeticException: / by zero

The entry is 2
```

- In this program, we loop through the values of an array arr.
- As previously mentioned, the portion that can cause an exception is placed inside the try block.
- If no exception occurs, the catch block is skipped and normal flow continues.
- But if any exception occurs, it is caught by the catch block (second value of the array).
- Here, we print the name of the exception using the `e.toString()` function.
- We can see that element 0 causes ZeroDivisionError.

Every exception in Java inherits from the base **Exception** class.

Catching Specific Exceptions in Java

- In the above example, we did not mention any specific exception in the `catch` clause.
- This is not a good programming practice as it will catch all exceptions and handle every case in the same way.
- We can specify which exceptions a `catch` clause should catch.
- A try clause can have any number of `catch` clauses to handle different exceptions, however, only one will be executed in case an exception occurs.
- You can use multiple `catch` blocks for different types of exceptions.

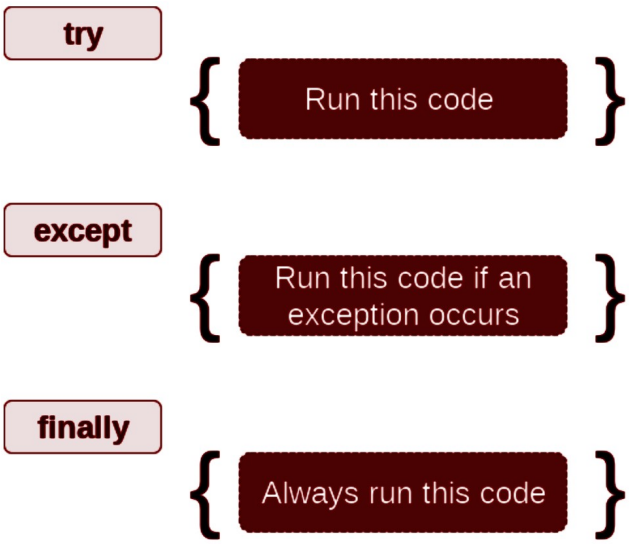
Here is an example to understand this better:

```
try{
    a=10/0;
}
catch(ArithmeticError e){
    System.out.println("Arithmetic Exception");
}
catch(IOException e){
    System.out.println("input output Exception");
}
```

Output:

```
Arithmetic Exception
```

`finally` Statement



The `try` statement in Java can have an optional `finally` clause. This clause is executed no matter what and is generally used to release external resources. Here is an example of file read and close to illustrate this:

```
FileReader f = null;
try{
    f = new FileReader(file);
    BufferedReader br = new BufferedReader(f);
    String line = null;
}
catch (FileNotFoundException fnf) {
    fnf.printStackTrace();
}
finally {
    if( f != null)
        f.close();
}
```

This type of construct makes sure that the file is closed even if an exception occurs during the program execution.