Object-Oriented Programming (OOPS-3)

Abstract Classes

An abstract class can be considered as a blueprint for other classes. Abstract classes are classes that contain one or more abstract methods. An abstract method is a method that has a declaration but does not have an implementation. This set of methods must be created within any child classes which inherit from the abstract class. *A class that contains one or more abstract methods is called an* ***abstract class****.*

Creating Abstract Classes in Java

- By default, Java does not provide abstract classes.
- A method becomes abstract when decorated with the keyword **`abstract`**.
- An abstract class cannot be directly instantiated i.e. we cannot create an object of the abstract class.
- However, the subclasses of an abstract class that have definitions for all the abstract methods declared in the abstract class, can be instantiated.
- While declaring abstract methods in the class, it is not mandatory to use the **`abstract`** decorator (i.e it would not throw an exception). However, it is considered a good practice to use it as it notifies the compiler that the user has defined an abstract method.

The given Java code uses the **ABC** class and defines an abstract base class:

```
 abstract class ABC{
    int value;
    Abstract int do_something(){ //Our abstract method declaration
        // TO_DO
    }
}
```

We will do it in the following example, in which we define two classes inheriting from our abstract class:

```
class add extends ABC{
    int do_something(){
        return value + 42;
    }
}

class mul extends ABC{
    int do_something(){
        return value * 42;
    }
}

class Test{
    public static void main(String[] args) {
        add x = new add(10);
        mul y = new mul(10);

        System.out.println(x.do_something());
        System.out.println(y.do_something());
    }
}
```

We get the output as:

```
52
420
```

Thus, we can observe that a class that is derived from an abstract class cannot be instantiated unless all of its abstract methods are overridden.

Note: Concrete classes contain only concrete (normal) methods whereas abstract classes may contain both concrete methods and abstract methods.

- An abstract method can have an implementation in the abstract class.

- However, even if they are implemented, this implementation shall be overridden in the subclasses.
- If you wish to invoke the method definition from the abstract superclass, the abstract method can be invoked with `super()` call mechanism. (*Similar to cases of "normal" inheritance*).
- Similarly, we can even have concrete methods in the abstract class that can be invoked using `super()` call. Since these methods are not abstract it is not necessary to provide their implementation in the subclasses.
- Consider the given example:

```
abstract class ABC{

    abstract int do_something(){ //Abstract Method
        System.out.println("Abstract Class AbstractMethod");
    }

    int do_something2(){ //Concrete Method
        System.out.println("Abstract Class ConcreteMethod");
    }
}

class AnotherSubclass extends ABC{
    int do_something(){
        //Invoking the Abstract method from super class
        super().do_something();
    }

    //No concrete method implementation in subclass
}

class Test{
    public static void main(String[] args) {
        AnotherSubclass x = new AnotherSubclass()
        x.do_something() //calling abstract method
        x.do_something2() //Calling concrete method
```

```
    }
}
```

We will get the output as:

```
Abstract Class AbstractMethod
Abstract Class ConcreteMethod
```

Another Example

The given code shows another implementation of an abstract class.

```
// Java program showing how an abstract class works
abstract class Animal{ //Abstract Class
    abstract move();
}

class Human extends Animal{ //Subclass 1
    void move(){
        System.out.println("I can walk and run");
    }
}

class Snake extends Animal{ //Subclass 2
    void move(){
        System.out.println("I can crawl")
}

class Dog extends Animal{ //Subclass 3
    void move(){
        System.out.println("I can bark")
    }
}

// Driver code
```

```
class Test{
    public static void main(String[] args) {
        Animal R = new Human();
        R.move();
        Animal K = Snake();
        K.move();
        R = Dog();
        R.move();
    }
}
```

We will get the output as:

```
I can walk and run
I can crawl
I can bark
```

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

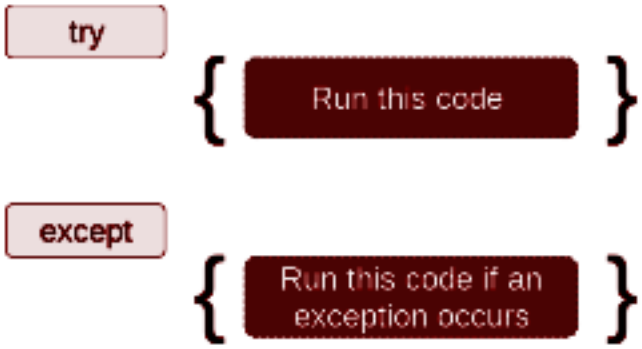
- **InterruptedException**
  It is thrown when a thread is waiting , sleeping , or doing some processing, and it is interrupted.
- **NoSuchFieldException**
  It is thrown when a class does not contain the field (or variable) specified
- **NoSuchMethodException**
  It is thrown when accessing a method 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:

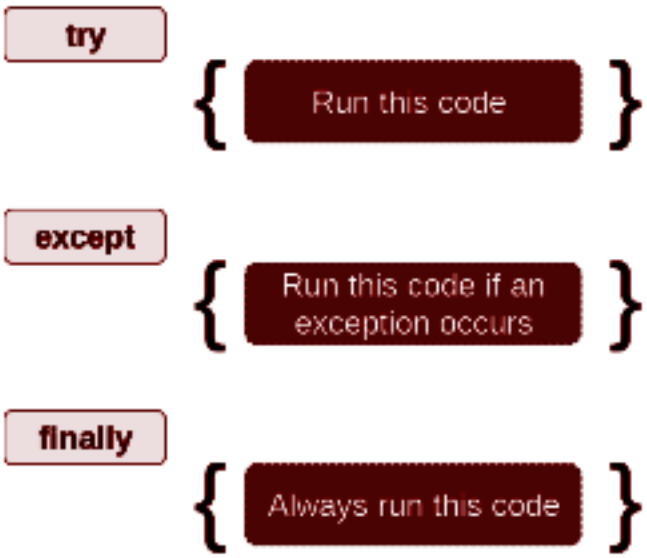
```
try{
    a=10/0;
}
catch(ArithmeticError e){
    System.out.println("Arithmetic Exception");
}
catch(IOException e){
```

```
    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.