UNIT - IV

PACKAGES:

A Package is a collection of classes , interfaces and sub-packages. It is used to group related code, avoid name conflicts, and make easier to manage.

Think of it like a folder in a computer that contains java files.

Packages are divided into two categories:

- Built-in Packages (packages from the Java API)
- User-defined Packages (create your own packages)

Built-in Packages:

In Java, Built-in packages are predefined libraries provided by the Java Development Kit (JDK) that contain a wide range of classes and interfaces to perform common tasks such as input/output operations, data structures, networking, database connectivity and more. These packages help developers avoid writing code from scratch by offering ready-to-use functionalities and are organized under the java and javax namespaces. These packages consist of a large number of classes which are parts of java API.

Some of the commonly used **built-in packages** are:

 java.lang: Contains language support classes (e.g classes which defines primitive data types, math operations). This package is automatically imported.

 java.io: Contains classes for supporting input / output operations.

 java.util: Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.

 java.applet: Contains classes for creating Applets.

 java.awt: Contain classes for implementing the components for graphical user interfaces (like button, menus, etc.)

 java.net: Contain classes for supporting networking operations.

User-defined packages:

These are the packages that are defined by the user.

Creating Packages :

Syntax:

package package-name;

Here, package-name should be a valid identifier.

Steps to Create User-defined Packages

Step 1: Creating a package in a Java class.

package example1;

Step 2: Include the class in a Java package.

```
package example1;
class Geeks {
public static void main(String[] args)

{
------------------ // Body of the class

}
}
```

Step 3: Now the user-defined package is successfully created, we can import it into other packages and use it's functions.

Note: *If we do not specify any package for a class, it is placed in the default package.*

In this example, we'll create a user-defined package name "mypackage" under that we've class named "MyClass" which has a function to print message.

```
package mypackage;
// Class
public class MyClass
 {
   public void show( )
   {
      System.out.println("Hello geeks!! How are you?");
   }
   }
}
```

Naming Conventions :

Naming conventions play a key role by making the purpose of variables, classes, methods and constants clear.

Why Naming Conventions are Important

In Java, it is good practice to name classes, variables and methods name as what they are supposed to do instead of naming them randomly.

Java uses CamelCase as a practice for writing names of methods, variables, classes, packages and constants

- *In package, everything is small even while we are combining two or more words in java*
- *In constants, we do use everything as uppercase and only '_' character is used even if we are combining two or more words in java*

Packages

- The prefix of a unique package name is always written in **all-lowercase ASCII letters** and should be one of the top-level domain names, like com, edu, gov, mil, net, org.
  - *java.util.Scanner ;*
    java.io.;*
  - As the name suggests in the first case we are trying to access the Scanner class from the java.util package and in other all classes (* standing for all) input-output classes making it so easy for another programmer to identify.

Note:

- For class, interfaces and constants, the first letter has to be uppercase.
- For method , variable and package name, the first letter has to be lowercase.

<u>Importing Packages:</u>

There are two main ways to import in Java, which are listed below:

- Import a specific class
- Import all classes in a package

1. Import a Specific Class

We can import a single class from a package using the syntax below.

Syntax:

import package_name.ClassName;

Example:

import java.util.ArrayList;

2. Import All Classes in a Package

We can import all classes in a package using the asterisk * wildcard which is shown below.

Syntax:

import package_name.;*

Example:

import java.util.;*

Import and Use the Class

Now, in another Java file Main.java, we can import and use the class from mypackage as follows:

```
import mypackage.MyClass;

public class Main {
  public static void main(String[] args) {
    MyClass myClass = new MyClass( );
    myClass.show( );
  }
}
```

Static import

With the help of static import, we can access the static members of a class directly without class name or any object. For Example: we always use sqrt() method of Math class by using Math class i.e. **Math.sqrt()**, but by using static import we can access sqrt() method directly.

```
// Java to illustrate calling of static member of  System class without Class name

import static java.lang.Math.*;
import static java.lang.System.*;
class Geeks {
    public static void main(String[] args)
    {
        // We are calling static member of System class
        // directly without System class name
        out.println(sqrt(4));
        out.println(pow(2, 2));
        out.println(abs(6.3));
    }
}
```

Hiding Classes:

If suppose we want to hide the class to be accessed from outside of the packages for some security reasons then remove the access modifier public in front of class declaration.

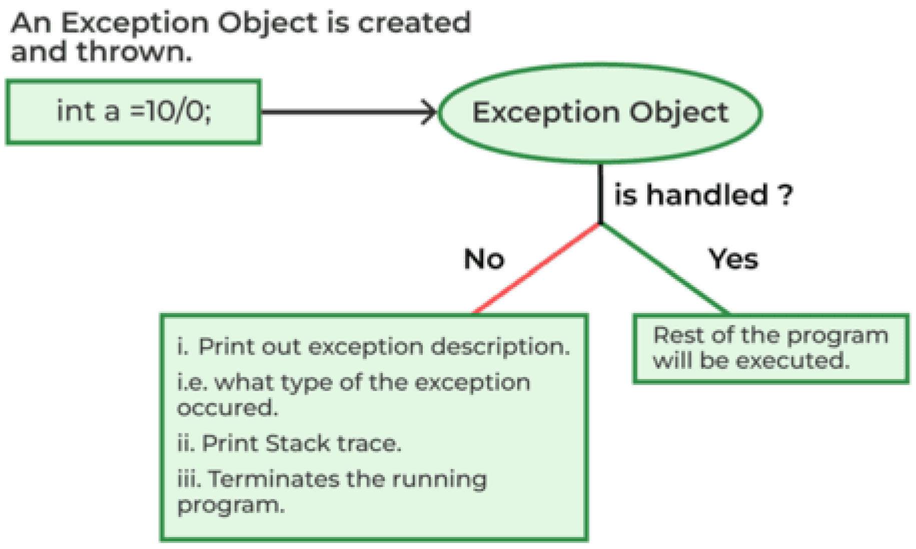
Table of Access Modifiers in Java

Context	Default	Private	Protected	Public
Same Class	Yes	Yes	Yes	Yes
Same Package Subclass	Yes	No	Yes	Yes
Same Package Non-Subclass	Yes	No	Yes	Yes
Different Package Subclass	No	No	Yes	Yes
Different Package Non-Subclass	No	No	No	Yes

Exception Handling

Exception handling in Java is an effective mechanism for managing runtime errors to ensure the application's regular flow is maintained. Some Common examples of exceptions include ClassNotFoundException, IOException, SQLException, RemoteException, etc. By handling these exceptions, Java enables developers to create robust and fault-tolerant applications.

Example:


An Exception Object is created and thrown.
int a =10/0;
Exception Object
is handled ?
No
Yes
i. Print out exception description.
i.e. what type of the exception occured.
ii. Print Stack trace.
iii. Terminates the running program.
Rest of the program will be executed.

Java Exception Hierarchy

In Java, all exceptions and errors are subclasses of the Throwable class. It has two main branches

1. Exception.
2. Error

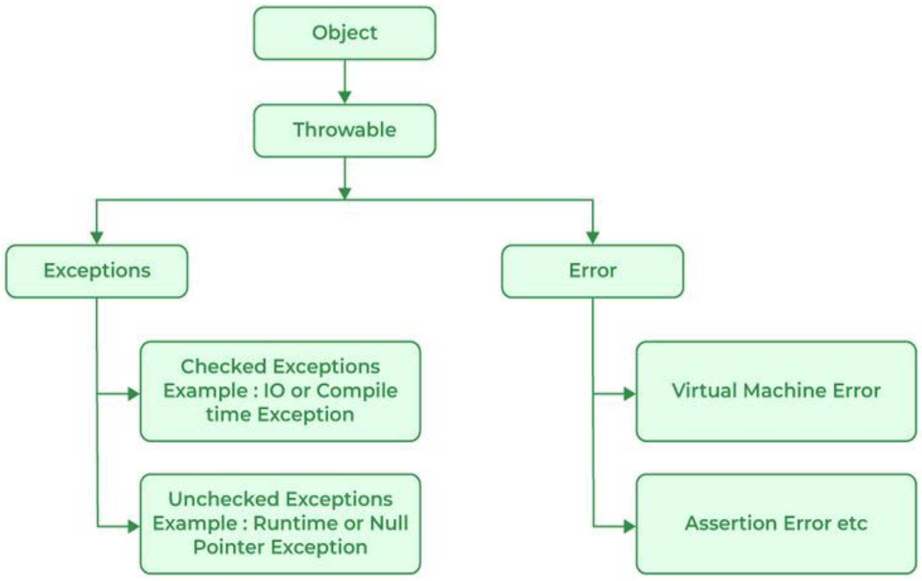
Exceptions can be categorized in two ways:

1. Built-in Exceptions

✓ Checked Exception

✓ Unchecked Exception

2. user-defined Exceptions


Object
Throwable
Exceptions
Error
Checked Exceptions Example : IO or Compile time Exception
Unchecked Exceptions Example : Runtime or Null Pointer Exception
Virtual Machine Error
Assertion Error etc

1. Built-in Exceptions:

Build-in Exception are pre-defined exception classes provided by Java to handle common errors during program execution. There are two type of built-in exception in java.

✓ **Checked Exceptions**

Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler. Examples of Checked Exception are listed below:

- **ClassNotFoundException:** Throws when the program tries to load a class at runtime but the class is not found because it's belong not present in the correct location or it is missing from the project.
- **InterruptedException:** Thrown when a thread is paused and another thread interrupts it.
- **IOException:** Throws when input/output operation fails.
- **InstantiationException:** Thrown when the program tries to create an object of a class but fails because the class is abstract, an interface or has no default constructor.
- **SQLException:** Throws when there is an error with the database.
- **FileNotFoundException**: Thrown when the program tries to open a file that does not exist.

✓ **Unchecked Exceptions**

The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception and even if we did not handle or declare it, the program would not give a compilation error. Examples of Unchecked Exception are listed below:

- **ArithmeticException**: It is thrown when there is an illegal math operation.
- **ClassCastException:** It is thrown when we try to cast an object to a class it does not belong to.
- **NullPointerException:** It is thrown when we try to use a null object (e.g. accessing its methods or fields).
- **ArrayIndexOutOfBoundsException:** This occurs when we try to access an array element with an invalid index.
- **ArrayStoreException:** This happens when we store an object of the wrong type in an array.
- **IllegalThreadStateException:** It is thrown when a thread operation is not allowed in its current state.

2. User-Defined Exception

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions, which are called "user-defined Exceptions".

try-catch block:

A try-catch block in Java is a mechanism to handle exceptions. This make sure that the application continues to run even if an error occurs. The code inside the try block is executed, and if any exception occurs, it is then caught by the catch block.

Syntax of try Catch Block

```
try {
// Code that might throw an exception
}
catch (ExceptionType e)
 {
// Code that handles the exception
}
```

try block:

The try block contains a set of statements where an exception can occur.

```
try
{
// statement(s) that might cause exception
}
```

catch block:

The catch block is used to handle the uncertain condition of a try block. A try block is always followed by a catch block, which handles the exception that occurs in the associated try block.

```
catch
{
// statement(s) that handle an exception
}
```

Working of try-catch Block

- If an exception occurs, the remaining code in the try block is skipped, and the JVM starts looking for the matching catch block.
- If a matching catch block is found, the code in that block is executed.
- If no matching catch block is found the exception is passed to the JVM default exception handler.

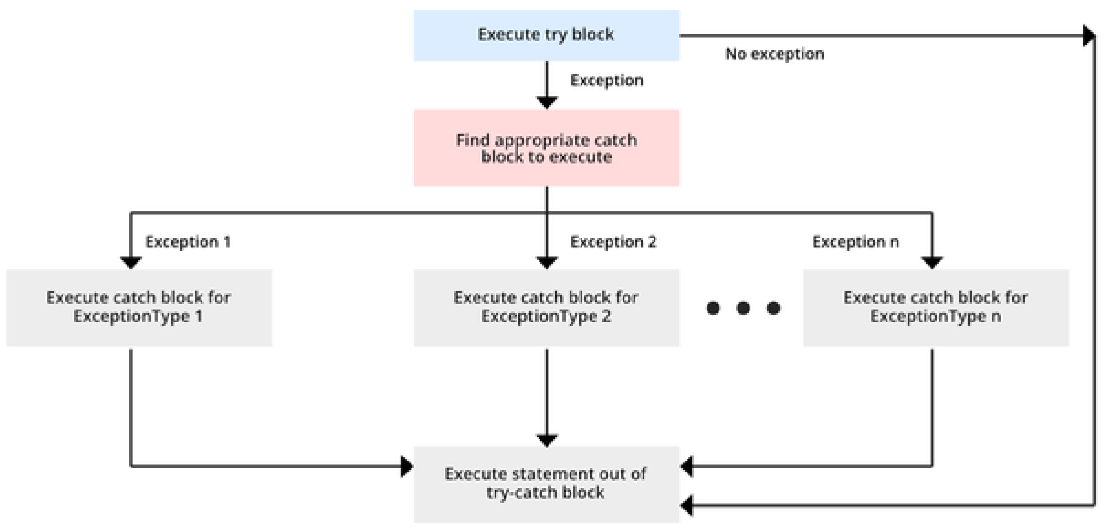
Example:

```
import java.io.*;

class Geeks {

  public static void main(String[] args) {

    try {

       int res = 10 / 0;

    }

    // Here we are Handling the exception
```

```
                catch (ArithmeticException e) {
                    System.out.println("Exception caught: " + e);
                }
            }
    }
```

Multiple Catch blocks:

```
        try {
            // Code that might throw an exception
        } catch (ExceptionType1 e1) {
            // Handling logic for ExceptionType1
        } catch (ExceptionType2 e2) {
            // Handling logic for ExceptionType2
    }
```


Execute try block
No exception
Exception
Find appropriate catch block to execute
Exception 1
Exception 2
Exception n
Execute catch block for ExceptionType 1
Execute catch block for ExceptionType 2
Execute catch block for ExceptionType n
Execute statement out of try-catch block

In Java, a try block can be followed by multiple catch blocks, allowing us to handle different types of exceptions separately. Each catch block is responsible for handling a specific exception type that might be thrown by the code in the try block. This approach provides a flexible way to manage various errors and implement different recovery logic for each.

```
 class MultipleCatchBlocksDemo
{
    public static void main(String[] args)
 {
        try {
            int[ ] numbers = {1, 2, 3};
            System.out.println(numbers[4]); // This will cause an
ArrayIndexOutOfBoundsException
            int result = 10 / 0; // This will cause an ArithmeticException
        }
catch (ArrayIndexOutOfBoundsException e)
{
   System.out.println("Error: Attempted to access an invalid array index.");
  }
 catch (ArithmeticException e)
 {
           System.out.println("Error: Cannot divide by zero!");
  }
 catch (Exception e)
{ // A general catch-all for other exceptions
            System.out.println("An unexpected error occurred.");
     }
        }
}
```

Finally block:

The finally block executes whether exception rise or not and whether exception handled or not.

- After the catch block, control moves to the finally block (if present).
- The final block is executed after the try catch block. regardless of whether an exception occurs or not.

We can use finally block after a try block or else after a catch block in our program.

Type 1:

```
try
{
----------
}
finally
{
----------
}
```

Type 2:

```
try
{
----------
}
catch(…)
{
……
}
catch(…)
{
….
}
finally
{
….
}
```

<u>Example:</u>

```
class FinallyExample1 {
  public static void main(String[] args) {
    try {
      int result = 10 / 0; // This will cause an ArithmeticException
      System.out.println("Result: " + result);
    }
catch (ArithmeticException e) {
      System.out.println("Error: Division by zero");
    }
finally {
      System.out.println("Finally block executed.");
    }
    System.out.println("Program continues after exception handling.");
  }
}
```

<u>throw and throws keyword:</u>

<u>throw keyword:</u>

The throw keyword in Java is used to explicitly throw an exception from a method or any block of code. We can throw either checked or unchecked exception. The throw keyword is mainly used to throw custom exceptions.

Syntax:

throw new Throwable subclass;

Example:

throw new ArithmeticException("/ by zero");

But this exception i.e., Instance must be of type **Throwable** or a subclass of **Throwable**.

The flow of execution of the program stops immediately after the throw statement is executed and the nearest enclosing **try** block is checked to see if it has a **catch** statement that matches the type of exception. If it finds a match, controlled is transferred to that statement otherwise next enclosing **try** block is checked and so on. If no matching **catch** is found then the default exception handler will halt the program.

Example:

```
class Geeks {
  static void fun()
  {
    try {
      throw new NullPointerException("demo");
    }
    catch (NullPointerException e) {
      System.out.println("Caught inside fun().");
      throw e;     // rethrowing the exception
    }
  }
  public static void main(String args[ ])
  {
    try {
      fun( );
    }
    catch (NullPointerException e) {
      System.out.println("Caught in main.");
    }
  }
}
```

Output

```
Caught inside fun().
Caught in main.
```

Explanation: The above example demonstrates the use of the throw keyword to explicitly throw a NullPointerException. The exception is caught inside the **fun()** method and rethrown, where it is then caught in the main() method.

Throws keyword:

throws is a keyword in Java that is used in the signature of a method to indicate that this method might throw one of the listed type exceptions. The caller to these methods has to handle the exception using a try-catch block.

Syntax:

type method_name(parameters) throws exception_list

where, exception_list is a comma separated list of all the exceptions which a method might throw.

In a program, if there is a chance of raising an exception then the compiler always warns us about it and we must handle that checked exception, Otherwise, we will get compile time error.

To prevent this compile time error we can handle the exception in two ways:

1. By using try catch
2. By using the **throws** keyword

We can use the throws keyword to delegate the responsibility of exception handling to the caller (It may be a method or JVM) then the caller method is responsible to handle that exception.

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