APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

KTU SPECIAL

We can together dream the B.tech

APJ ABDUL KALAM
TECHNOLOGICAL UNIVERSITY
സാങ്കേതിക ശാസ്ത്ര സർവകലാശാല

• KTU STUDY MATERIALS	• SYLLABUS
• KTU LIVE NOTIFICATION	• SOLVED QUESTION PAPERS

JOIN WITH US

WWW.KTUSPECIAL.IN KTUSPECIAL t.me/ktuspecial1

Module 3

Packages and Interfaces –

Packages - Defining a Package, CLASSPATH, Access Protection, Importing Packages.

Interfaces - Interfaces v/s Abstract classes, defining an interface, implementing interfaces, accessing implementations through interface references, extending interface(s).

Exception Handling - Checked Exceptions, Unchecked Exceptions, try Block and catch Clause, Multiple catch Clauses, Nested try Statements, throw, throws and finally, Java Built-in Exceptions, Custom Exceptions.

Introduction to design patterns in Java : Singleton and Adaptor.

Java Packages

A package in Java is used to group related classes. Think of it as a folder in a file directory. We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:

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

Built-in Packages

The Java API is a library of prewritten classes that are free to use, included in the Java Development Environment.

The library contains components for managing input, database programming, and much much more. The complete list can be found at Oracle's website: The library is divided into packages and classes. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contains all the classes that belong to the specified package.

To use a class or a package from the library, you need to use the import keyword:

Import a Class

If you find a class you want to use, for example, the Scanner class, which is used to get user input, write the following code:

Example

```
import java.util.Scanner;
```

In the example above, java.util is a package, while Scanner is a class of the java.util package.

To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation. In our example, we will use the nextLine() method, which is used to read a complete line:

Example

Using the Scanner class to get user input:

```
import java.util.Scanner;

class MyClass {

  public static void main(String[] args) {

    Scanner myObj = new Scanner(System.in);

    System.out.println("Enter username");

    String userName = myObj.nextLine();

    System.out.println("Username is: " + userName);

  }

}
```

Syntax

import *package.name.Class*; // Import a single class

import *package.name*.*; // Import the whole package

User-defined Packages

To create your own package, you need to understand that Java uses a file system directory to store them. Just like folders on your computer:

Example

```
└── root
    └── mypack
        └── MyPackageClass.java
```

To create a package, use the package keyword:

MyPackageClass.java

```
package mypack;
class MyPackageClass {
  public static void main(String[] args) {
    System.out.println("This is my package!");
  }
}
```

Save the file as MyPackageClass.java, and compile it:

```
C:\Users\Your Name>javac MyPackageClass.java
```

Then compile the package:

```
C:\Users\Your Name>javac -d . MyPackageClass.java
```

This forces the compiler to create the "mypack" package.

The -d keyword specifies the destination for where to save the class file. You can use any directory name, like c:/user (windows), or, if you want to keep the package within the same directory, you can use the dot sign ".", like in the example above.

Note: The package name should be written in lower case to avoid conflict with class names.

When we compiled the package in the example above, a new folder was created, called "mypack".

To run the MyPackageClass.java file, write the following:

```
C:\Users\Your Name>java mypack.MyPackageClass
```

The output will be:

```
This is my package!
```

Detailed Explanation of CLASSPATH in Java

The CLASSPATH is a parameter—either set as an environment variable or passed as a command-line argument—that tells the Java compiler (javac) and Java Virtual Machine (java) where to look for .class files or Java packages used in your program.

 Why is CLASSPATH Important?

When you compile or run Java programs that use external packages or classes, Java needs to know where to find those classes. That's where the CLASSPATH comes in.

Access Protection in Packages

Java provides four levels of access control:

Modifier	Same Class	Same Package	Subclass	Other Package
private	□	✕	✕	✕
(default)	□	□	✕	✕
protected	□	□	□	✕
public	□	□	□	□

1. Same Class

- All modifiers work inside the same class.
- So everything is here.
- Example: A method can always access private or public variables if it's in the same class.

2. Same Package

- public, protected, and default (no modifier) can be accessed by other classes in the same package.

- private is not accessible outside its own class.

3. Subclass (Other Package)

- This means: A class that extends your class but is in a different package.

- public: Yes — visible everywhere.

- protected: Yes — visible to subclasses, even in different packages.

- default: No — not visible in a different package.

- private: No — not visible anywhere outside the class.

---

4. Other Packages (not subclass)

- public: Yes — accessible from anywhere.

- protected, default, and private: No — not accessible from other packages unless you are a subclass (for protected).

How to Set CLASSPATH in Ubuntu (Linux)

What is CLASSPATH?

- It tells Java where to find your .class files or .jar files.

- Useful when you're working with user-defined packages or external libraries.

Example: Suppose You Have This Project

project/

```
├── mypackage/
│   └── MyClass.java
└── Test.java
```

mypackage/MyClass.java

```
package mypackage;

public class MyClass {
    public void display() {
        System.out.println("Hello from MyClass!");
    }
}
```

Test.java

```
import mypackage.MyClass;

public class Test {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.display();
    }
}
```

Step-by-Step: Set CLASSPATH and Compile

1. Open Terminal and go to your project directory:
   cd /path/to/project

2. Compile the package class first:
   javac mypackage/MyClass.java

3. Compile the main class with the classpath:
   javac -cp . Test.java

4. Run the program with the classpath:
   java -cp . Test

   -cp . means: "Look for classes in the current directory."

Difference Between Abstract Class and Interface in Java

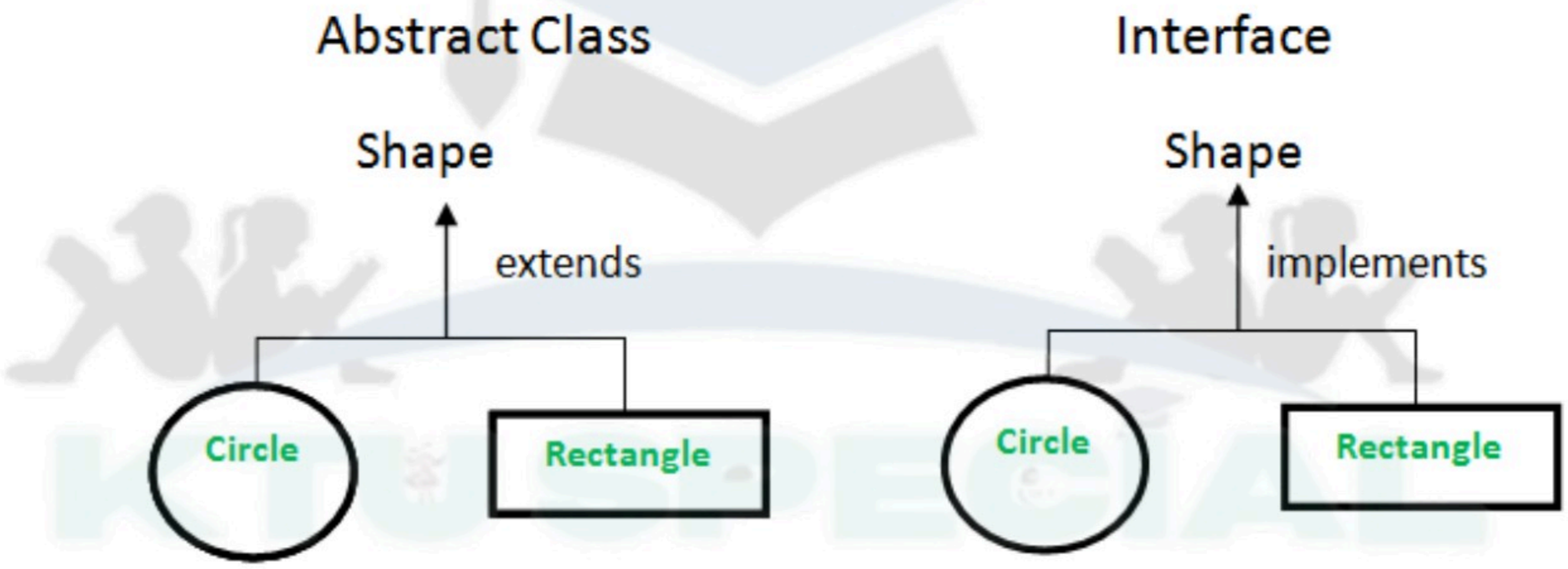
In object-oriented programming (OOP), both abstract classes and interfaces serve as fundamental constructs for defining contracts. They establish a blueprint for other classes, ensuring consistent implementation of methods and behaviors. However, they each come with distinct characteristics and use cases.

Difference Between Abstract Class and Interface

Points	Abstract Class	Interface
Definition	Cannot be instantiated; contains both abstract (without implementation) and concrete methods (with implementation)	Specifies a set of methods a class must implement; methods are abstract by default.
Implementation Method	Can have both implemented and abstract methods.	Methods are abstract by default; Java 8, can have default and static methods.
Inheritance	class can inherit from only one abstract class.	A class can implement multiple interfaces.

Access Modifiers	Methods and properties can have any access modifier (public, protected, private).	Methods and properties are implicitly public.
Variables	Can have member variables (final, non-final, static, non-static).	Variables are implicitly public, static, and final (constants).

As we know that abstraction refers to hiding the internal implementation of the feature and only showing the functionality to the users. i.e., showing only the required features, and hiding how those features are implemented behind the scene. Whereas, an Interface is another way to achieve abstraction in Java. Both abstract class and interface are used for abstraction



Interfaces

An interface is a completely "abstract class" that is used to group related methods with empty bodies:

Example

```
// interface

interface Animal {

  public void animalSound(); // interface method (does not have a body)

  public void run(); // interface method (does not have a body)

}
```

To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class with the implements keyword (instead of extends). The body of the interface method is provided by the "implement" class:

Example

```
// Interface

interface Animal {

  public void animalSound(); // interface method (does not have a body)

  public void sleep(); // interface method (does not have a body)

}


// Pig "implements" the Animal interface

class Pig implements Animal {

  public void animalSound() {

    // The body of animalSound() is provided here

    System.out.println("The pig says: wee wee");

  }

  public void sleep() {
```

```
    // The body of sleep() is provided here
    System.out.println("Zzz");
  }
}

class Main {
  public static void main(String[] args) {
    Pig myPig = new Pig();  // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Design Patterns in Java

Design patterns are proven solutions to common problems in software design. They help create more maintainable, scalable, and efficient code by offering reusable approaches to solving architectural problems.

1. Singleton Pattern

Purpose:

Ensures a class has only one instance and provides a global point of access to it.

Use Cases:

- Configuration settings
- Logger classes

- Database connections

Implementation:

```
public class Singleton {
    // Step 1: Create a private static instance
    private static Singleton instance;

    // Step 2: Make the constructor private to prevent instantiation
    private Singleton() {
        System.out.println("Singleton instance created.");
    }

    // Step 3: Provide a public static method to get the instance
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton(); // Lazy initialization
        }
        return instance;
    }
}
```

Usage:

```
public class Main {

    public static void main(String[] args) {

        Singleton s1 = Singleton.getInstance();

        Singleton s2 = Singleton.getInstance();



        System.out.println(s1 == s2); // true - both refer to the same instance

    }

}
```

---

2. Adapter Pattern

Purpose:

Allows incompatible interfaces to work together. It converts one interface into another expected by the client.

Use Cases:

- When integrating legacy code with new systems
- When working with third-party libraries

Example Scenario:

You have a class that uses a method request(), but a new system uses specificRequest(). You use an adapter to bridge the gap.

Class Diagram:

Client --> Target (expected interface)

^

|

Adapter --> Adaptee (incompatible interface)

Implementation:

```
// Target interface

interface Target {
    void request();
}


// Adaptee class with incompatible method

class Adaptee {
    public void specificRequest() {
        System.out.println("Specific request in Adaptee");
    }
}


// Adapter class that makes Adaptee compatible with Target

class Adapter implements Target {
    private Adaptee adaptee;


    public Adapter(Adaptee adaptee) {
```

```
    this.adaptee = adaptee;
}

@Override
public void request() {
    adaptee.specificRequest(); // Delegating the call
}
}
```

Usage:

```
public class Main {
    public static void main(String[] args) {
        Adaptee adaptee = new Adaptee();
        Target adapter = new Adapter(adaptee);
        adapter.request(); // Output: Specific request in Adaptee
    }
}
```

Pattern	Purpose	Key Feature

Singleton	Ensure a single instance of a class	Controlled instantiation
Adapter	Connect incompatible interfaces	Converts interface of a class

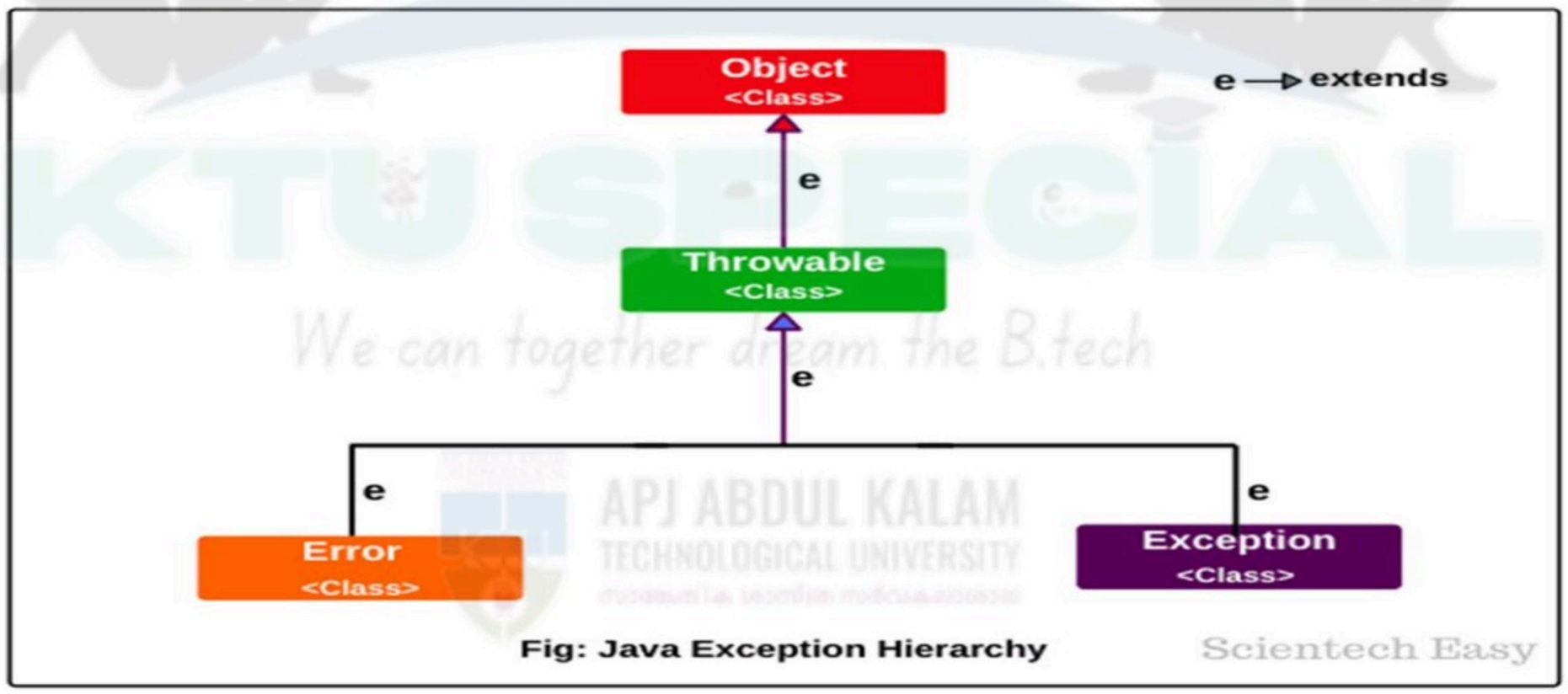
Exception Handling in Java

The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained

What is Exception in Java?

Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions. Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates an object. This object is called the exception object. It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred

Exception Hierarchy



The Throwable class is the root of exception hierarchy and is an immediate subclass of Object class.Exception Hierarchy

1.Throwable class:

As shown in the above figure, Throwable class which is derived from Object class, is a top of exception hierarchy from which all exception classes are derived directly or indirectly. It is the root of all exception classes. It is present in the java.lang package.

Throwable class is the superclass of all exceptions in java. This class has two subclasses: Error and Exception. Errors or exceptions occurring in java programs are objects of these classes. Using Throwable class, you can also create your own custom exceptions

2. Error:

Error class is the subclass of Throwable class and a superclass of all the runtime error classes. It terminates the program if there is problem-related to a system or resources (JVM).

An error generally represents an unusual problem or situation from which it is difficult to recover. It does not occur by programmer mistakes. It generally occurs if the system is not working properly or resources are not allocated properly.

VirtualMachineError, StackOverFlowError, AssertionError, LinkageError, OutOfMmeoryError, etc are examples of error.

3. Exception:

It is represented by an Exception class that represents errors caused by the program and by external factors. Exception class is a subclass of Throwable class and a superclass of all the exception classes.

All the exception classes are derived directly or indirectly from the Exception class. They originate from within the application.

The exception class provides two constructors:

public Exception() (Default constructor)

public Exception(String message) (It takes a message string as argument)

Each of the exception classes provides two constructors: one with no argument and another with a String type argument. Exception class does not provide its own method. It inherits all methods provided by the Throwable class.

Unchecked exception

- Unchecked exception classes are defined inside

  java.lang package.

  - The unchecked exceptions are subclasses of the standard type RuntimeException.
  - In the Java language, these are called *unchecked exceptions because the compiler does not check to see whether there is a method that handles or* throws these exceptions.

  - If the program has an unchecked exception then it will *compile without error* but an exception occurs when the program runs.
    - .g Exceptions under Error , ArrayIndexOutOfBoundException

- E.g Exceptions under Error , ArrayIndexOutOfBoundException

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Checked exception

- There are some exceptions that are defined by java.lang that must be included in a

method's throws list, if a method generates such exceptions and that *method does not handle it itself*. These are called checked exceptions

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

- IOException
- FileNotFoundException
- SQLException

Difference Between Checked Exception and Unchecked Exception

www.smartprogramming.in

Checked Exception / Compile Time Exception	Unchecked Exception / Runtime Exception
1. Checked Exceptions are the exceptions that are checked and handled at compile time.	1. Unchecked Exceptions are the exceptions that are not checked at compiled time.
2. The program gives a compilation error if a method throws a checked exception.	2. The program compiles fine because the compiler is not able to check the exception.
3. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword.	3. A method is not forced by compiler to declare the unchecked exceptions thrown by its implementation. Generally, such methods almost always do not declare them, as well.
4. A checked exceptions occur when the chances of failure are too high.	4. Unchecked exception occurs mostly due to programming mistakes.
5. They are direct subclass of Exception class but do not inherit from RuntimeException.	5. They are direct subclass of RuntimeException class.

try Block and *catch Clause*

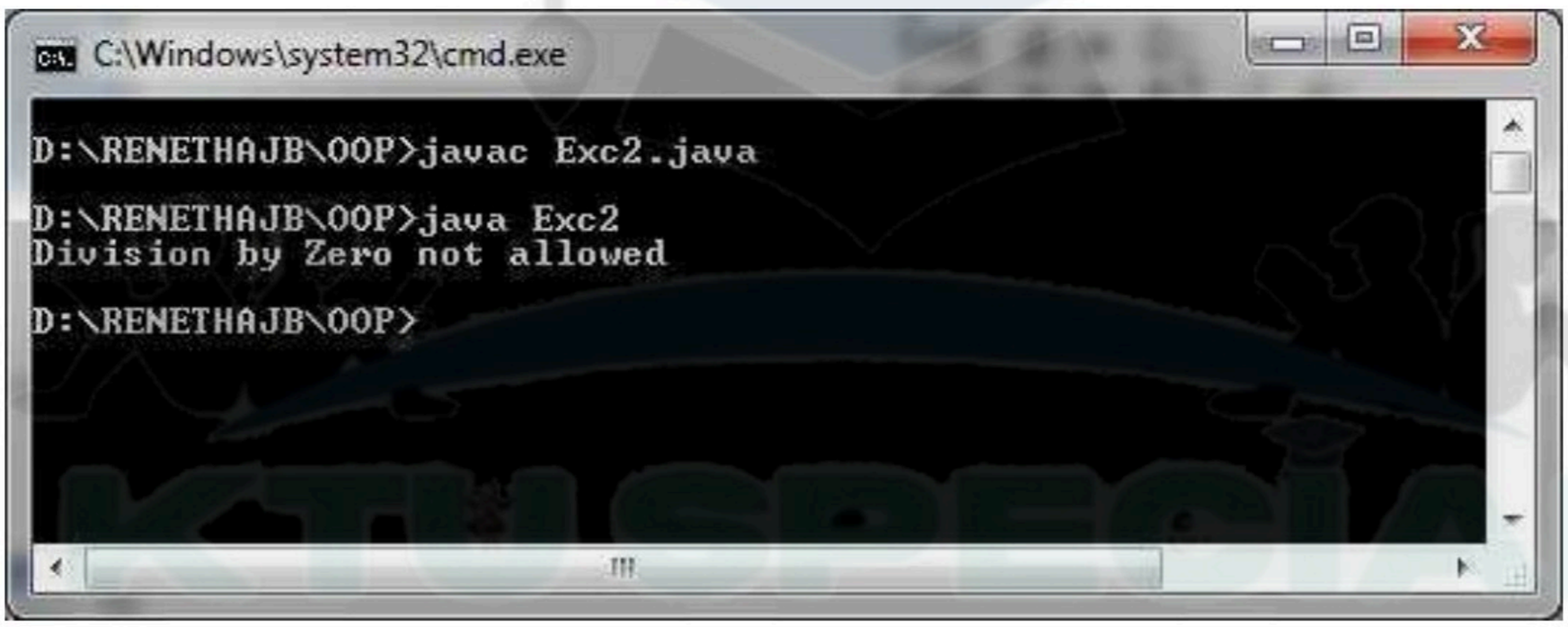
Benefits of exception handling

- First, it allows us to fix the error.
- Second, it prevents the program from terminating.
- To guard against and handle a run-time error, simply enclose the code that we want to monitor inside a try block.

- Immediately *after the try block*, there is a catch clause that <u>specifies the exception type</u> that we wish to catch . The catch block can process that exception..

<u>Example</u>

```
class Exc2{
                public static void main(String args[])
                    {
                        t ry
{
int d = 0;
int a = 42 / d;
}
catch(ArithmeticException ae)
{
System.out.println("Division by Zero not allowed");
}
}
}
```



<u>try-catch block to handle division by zero exception</u>

```
import java.io.*;
class GFG {
    public static void main(String[] args)
    {
        int a = 5;
        int b = 0;
        try {
            System.out.println(a / b); // throw Exception
```

```
        }
        catch (ArithmeticException e) {
            // Exception handler
            System.out.println(
                "Divided by zero operation cannot possible");
        }
    }
}
```

```
mayur@mayur:~$ java GFG
Divided by zero operation cannot possible
mayur@mayur:~$
```

- A 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.
  - Each catch block can catch exceptions in statements inside immediately preceding the try block.
- A catch statement cannot catch an exception thrown by another try statement (except in the case of nested try statements).
- The statements that are protected by try must be surrounded by curly braces. (That is, they must be within a block.)
- We cannot use try on a single statement

Multiple catch Clauses

- There can be more than one exception in a single piece of code.
  - To handle this type of situation, we can specify two or more catch clauses, each catching a *different type of exception*.
- When an exception is thrown,
  - each catch statement is inspected in order, and

– the first one whose type matches that of the exception is executed.

- After one catch statement executes, the other catch statements are bypassed(ignored), and execution continues after the try/catch block.

- Why do we need multiple catch blocks in Java?
- Multiple catch blocks in Java are used to handle different types of exceptions. When statements in a single try block generate multiple exceptions, we require multiple catch blocks to handle different types of exceptions. This mechanism is called a multi-catch block in java.
- A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. You can catch different exceptions in different catch blocks if it contains a different exception handler. When an exception occurs in a try block, the corresponding catch block that handles that particular exception executes. So, if you have to perform different tasks at the occurrence of different exceptions, you can use the multi-try catch in Java.

Syntax

```
try {
  //  code
} catch (ExceptionType1 e1) {
  // Catch block
} catch (ExceptionType2 e2) {
  // Catch block
} catch (ExceptionType3 e3) {
  // Catch block
}
```

Example

```
public class MultipleCatchBlock4 {
```

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

        try{
            String s=null;
            System.out.println(s.length());
           }
          catch(ArithmeticException e)
           {
            System.out.println("Arithmetic Exception occurs");
           }
        catch(ArrayIndexOutOfBoundsException e)
           {
            System.out.println("ArrayIndexOutOfBounds Exception occurs");
           }
        catch(Exception e)
           {
            System.out.println("Parent Exception occurs");
           }
                System.out.println("rest of the code");
    }
}
```

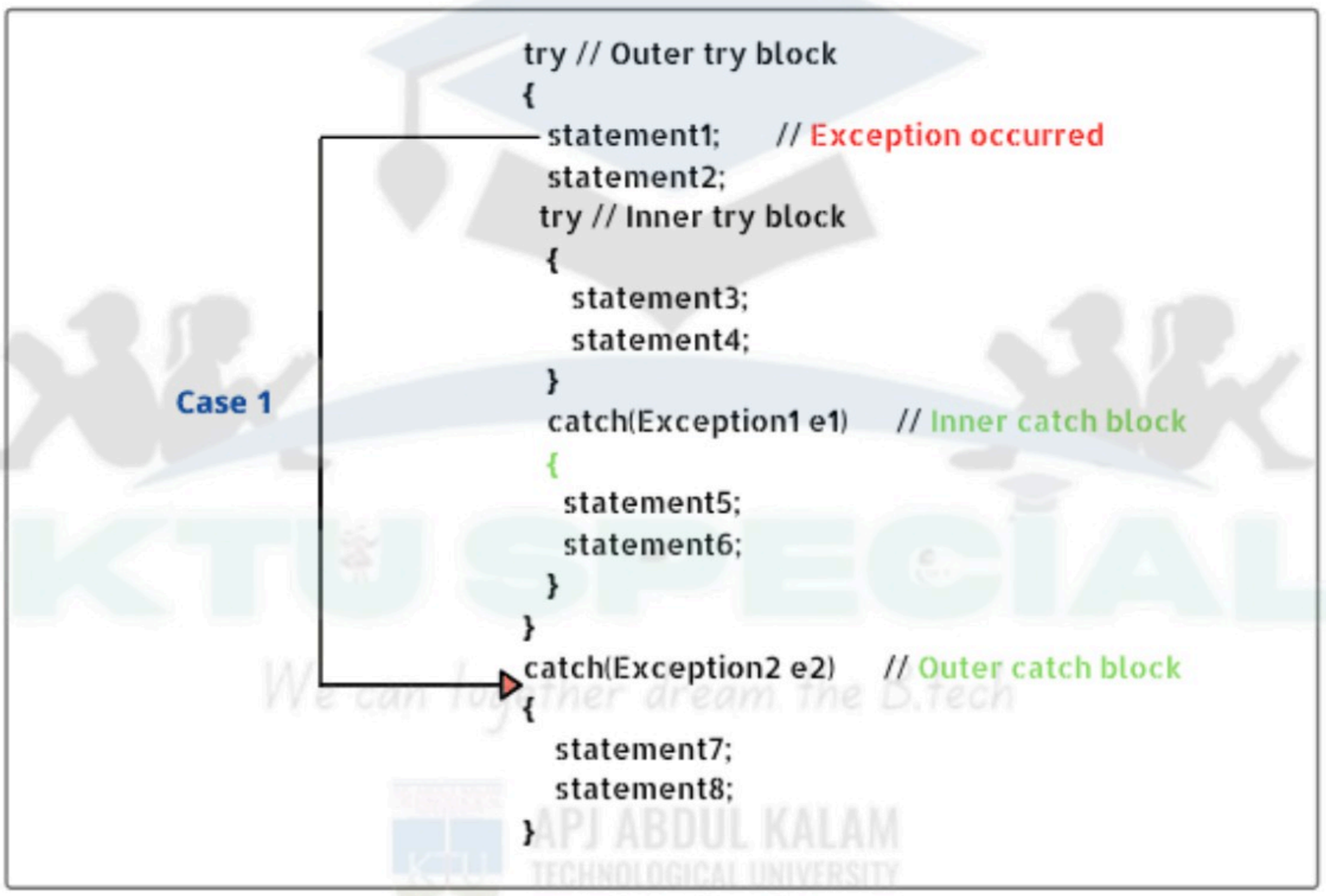
Output

Parent Exception occurs

rest of the code

Nested *try* Statements

- The try statement can be nested.
  - A try statement can be inside the block of another try.
- Each time a try statement is entered, the context of that exception is pushed on the stack.
  - If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match.
  - This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted.
  - If no catch statement matches, then the Java run-time system will handle the exception.



Example

class Geeks {

```
// Main method
public static void main(String args[]) {
    // Main try block
    try {
        // Initializing array
        int a[] = { 1, 2, 3, 4, 5 };

        // Trying to print element at index 5
        System.out.println(a[5]);

        // Inner try block (try-block2)
        try {
            // Performing division by zero
            int x = a[2] / 0; // This will throw ArithmeticException
        } catch (ArithmeticException e2) {
            System.out.println("Division by zero is not possible");
        }
    } catch (ArrayIndexOutOfBoundsException e1) {
        System.out.println("ArrayIndexOutOfBoundsException");
        System.out.println("Element at such index does not exist");
    }
  }
}
```

Output

```
ArrayIndexOutOfBoundsException
Element at such index does not exist
```

More features of Java :

Exception Handling:

- *throw*
- *throws*
- *Finally*

1.throw

We know that if an error occurs, an exception object is getting created and then Java runtime starts processing to handle them. Sometimes we might want to generate exceptions explicitly in our code. For example, in a user authentication program, we should throw exceptions to clients if the password is null. The throw keyword is used to throw exceptions to the runtime to handle it.

2.throws

When we are throwing an exception in a method and not handling it, then we have to use the throws keyword in the method signature to let the caller know the exceptions that might be thrown by the method. The caller method might handle these exceptions or propagate them to its caller method using the throws keyword. We can provide multiple exceptions in the throws clause, and it can be used with the main() method also

Throw vs Throws in java

1. Throws clause is used to declare an exception, which means it works similar to the try-catch block. On the other hand, throw keyword is used to throw an exception explicitly.
2. If we see syntax wise then throw is followed by an instance of Exception class and throws is followed by exception class names.

For example:
throw new ArithmeticException("Arithmetic Exception");
and
throws ArithmeticException;
3. Throw keyword is used in the method body to throw an exception, while throws is used in method signature to declare the exceptions that can occur in the statements present in the method.
For example:
Throw:

```
...
void myMethod() {
   try {
      //throwing arithmetic exception using throw
      throw new ArithmeticException("Something went wrong!!");
   }
   catch (Exception exp) {
      System.out.println("Error: "+exp.getMessage());
   }
}
...
```

Throws:

```
...
```

```
//Declaring arithmetic exception using throws
void sample() throws ArithmeticException{
  //Statements
}
```

…

4. You can throw one exception at a time but you can handle multiple exceptions by declaring them using throws keyword.

For example:

Throw:

```
void myMethod() {
  //Throwing single exception using throw
  throw new ArithmeticException("An integer should not be divided by zero!!");
}
```

..

Throws:

```
//Declaring multiple exceptions using throws
void myMethod() throws ArithmeticException, NullPointerException{
  //Statements where exception might occur
}
```

finally – the finally block is optional and can be used only with a try-catch block. Since exceptions halt the process of execution, we might have some resources open that will not get closed, so we can use the finally block. The finally block always gets executed, whether an exception occurred or not.

Finally Block in Java | A "finally" is a keyword used to create a block of code that follows a try or catch block.

A finally block contains all the crucial codes such as closing connections, stream, etc that is always executed whether an exception occurs within a try block or not.

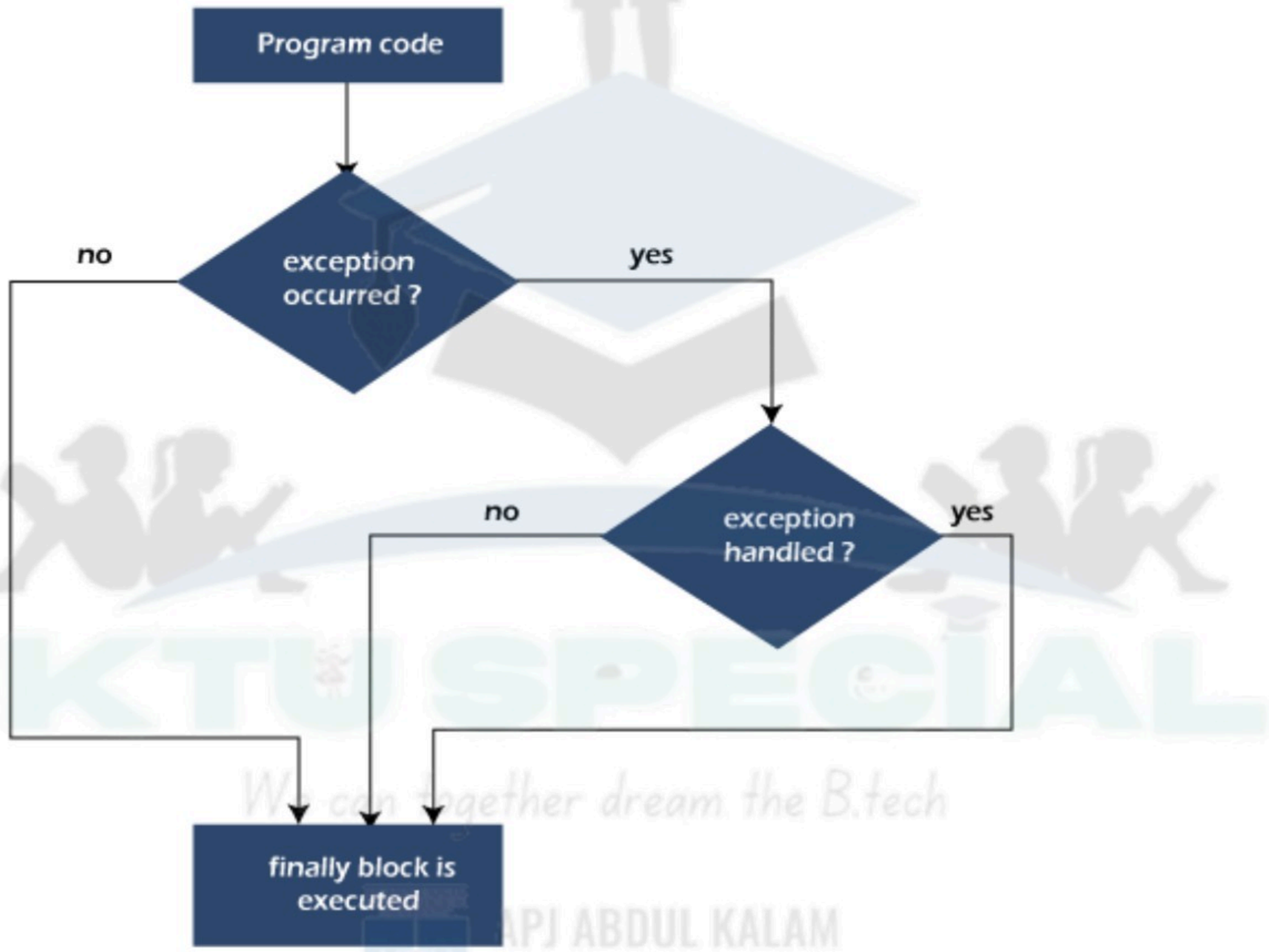
When the finally block is attached with a try-catch block, it is always executed whether the catch block has handled the exception thrown by the try block or not.

```
try
{
  statement1;
  statement2;
}
catch(Exceptiontype e1)
{
   statement3;
 }
 statement4;
finally
{
  statement5;
```

}

Why finally is needed?

- When exceptions are thrown, execution in a method takes a nonlinear path and changes the normal flow through the method.
  - Sometimes exceptions cause the method to return prematurely.
  - This may cause problems in some cases.
  - E.g a method opens a file upon entry and closes it upon exit, then we will not want the code that closes the file to be bypassed by the exception-handling mechanism.
    - In such situations the code for closing that file and other codes that should not be bypassed should be written inside finally block
    - This will ensure that necessary codes are not skipped because of exception handling.



Points To Remember While Using Finally Block in Java

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses whenever a try/catch block is present.

- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.
- finally block is not executed in case exit() method is called before finally block or a fatal error occurs in program execution.
- finally block is executed even method returns a value before finally block.

Why Java Finally Block Used?

- Java finally block can be used for clean-up (closing) the connections, files opened, streams, etc. those must be closed before exiting the program.
- It can also be used to print some final information.

Example 1

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

OUTPUT

```
Error: Division by zero
Finally block executed
```

Types of Exception in Java with Examples

Built-in Exceptions in Java

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations. Below is the list of important built-in exceptions in Java.

1. **ArithmeticException:** It is thrown when an exceptional condition has occurred in an arithmetic operation.
2. **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.
3. **ClassNotFoundException:** This Exception is raised when we try to access a class whose definition is not found
4. **FileNotFoundException:** This Exception is raised when a file is not accessible or does not open.
5. **IOException:** It is thrown when an input-output operation failed or interrupted
6. InterruptedException: It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.
7. **NoSuchFieldException:** It is thrown when a class does not contain the field (or variable) specified
8. **NoSuchMethodException:** It is thrown when accessing a method that is not found.
9. **NullPointerException:** This exception is raised when referring to the members of a null object. Null represents nothing

10. **NumberFormatException:** This exception is raised when a method could not convert a string into a numeric format.
11. **RuntimeException:** This represents an exception that occurs during runtime.
12. **StringIndexOutOfBoundsException:** It is thrown by String class methods to indicate that an index is either negative or greater than the size of the string
13. **IllegalArgumentException :** This exception will throw the error or error statement when the method receives an argument which is not accurately fit to the given relation or condition. It comes under the unchecked exception.
14. **IllegalStateException :** This exception will throw an error or error message when the method is not accessed for the particular operation in the application. It comes under the unchecked exception.

Examples of Built-in Exception:

1. **Arithmetic exception :** It is thrown when an exceptional condition has occurred in an arithmetic operation.

```
// Java program to demonstrate
// ArithmeticException
class ArithmeticException_Demo {
  public static void main(String[] args) {
     try {
        int a = 30, b = 0;
        int c = a / b; // cannot divide by zero
        System.out.println("Result = " + c);
     } catch (ArithmeticException e) {
        System.out.println("Can't divide a number by 0");
     }
  }
}
```

Output
Can't divide a number by 0

2. ArrayIndexOutOfBounds Exception: 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.

```
// Java program to demonstrate
// ArrayIndexOutOfBoundException
class ArrayIndexOutOfBound_Demo {
   public static void main(String[] args) {
      try {
         int[] a = new int[5];
         a[5] = 9; // accessing 6th element in an array of
                   // size 5
      } catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("Array Index is Out Of Bounds");
      }
   }
}
```

Output
Array Index is Out Of Bounds

3. ClassNotFoundException : This Exception is raised when we try to access a class whose definition is not found.

```
// Java program to illustrate the
// concept of ClassNotFoundException
class Bishal {

}
```

```
class Geeks {

}

class MyClass {
   public static void main(String[] args) {
      try {
         Object o = Class.forName(args[0]).newInstance();
         System.out.println("Class created for " + o.getClass().getName());
      } catch (ClassNotFoundException | InstantiationException | IllegalAccessException e) {
         System.out.println("Exception occurred: " + e.getMessage());
      }
```

```
    }
}
```

Output:

```
ClassNotFoundException
```

4. FileNotFoundException : This Exception is raised when a file is not accessible or does not open.

```
// Java program to demonstrate
// FileNotFoundException
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;

class File_notFound_Demo {
    public static void main(String[] args) {
        try {
            // Following file does not exist
            File file = new File("file.txt");
            FileReader fr = new FileReader(file);
        } catch (FileNotFoundException e) {
            System.out.println("File does not exist");
        }
    }
}
```

Output

```
File does not exist
```

5. IOException : It is thrown when an input-output operation failed or interrupted

```
// Java program to illustrate IOException
import java.io.*;

class Geeks {
    public static void main(String[] args) {
        try {
            FileInputStream f = new FileInputStream("abc.txt");
            int i;
            while ((i = f.read()) != -1) {
                System.out.print((char)i);
```

```
        }
        f.close();
      } catch (IOException e) {
        System.out.println("IOException occurred: " + e.getMessage());
      }
   }
}
```

Output:

```
error: unreported exception IOException; must be caught or declared to be thrown
```

How to Create User Defined Exceptions in Java?

To create a custom exception in java, we have to create a class by extending it with the Exception class from the java.lang package.

```
//user-defined exception in java
public class SimpleCustomException extends Exception{

}
```

Example

```
// class representing custom exception
class InvalidAgeException  extends Exception
{
   public InvalidAgeException (String str)
   {
      // calling the constructor of parent Exception
      super(str);
   }
}

// class that uses custom exception InvalidAgeException
public class TestCustomException1
{

   // method to check the age
   static void validate (int age) throws InvalidAgeException{
     if(age < 18){

      // throw an object of user defined exception
      throw new InvalidAgeException("age is not valid to vote");
   }
     else {
      System.out.println("welcome to vote");
```

```
    }
        } // main method
public static void main(String args[])
{
    try
    {
        // calling the method
        validate(13);
    }
    catch (InvalidAgeException ex)
    {
        System.out.println("Caught the exception");

        // printing the message from InvalidAgeException object
        System.out.println("Exception occured: " + ex);
    }

    System.out.println("rest of the code...");
  }
}
```

```
C:\Users\Anurati\Desktop\abcDemo>javac TestCustomException1.java

C:\Users\Anurati\Desktop\abcDemo>java TestCustomException1
Caught the exception
Exception occured: InvalidAgeException: age is not valid to vote
rest of the code...
```

Short answer Questions

1. Define a Java package. How do you import a user-defined package?
2. What is CLASSPATH? How is it related to packages?
3. List any three access modifiers and explain their use in packages.
4. Write any three differences between interfaces and abstract classes.
5. Write the syntax for defining and implementing an interface.
6. What is a custom exception? When and why is it used?
7. Differentiate between throw and throws in Java with examples.
8. Explain the use of finally block in exception handling.
9. What is the Singleton design pattern? Mention its advantages.
10. When is the Adapter pattern used? Explain with a real-world analogy.

11. Give an example of nested try-catch in Java.
12. What are built-in exceptions in Java? Mention any three.
13. How can you access interface methods using interface references?
14. Write a short note on extending interfaces in Java.
15. Give an example to demonstrate multiple catch blocks.

Essay questions

1. Explain the steps to create and use a **user-defined package** in Java. How is CLASSPATH used in this context?
2. Define and compare **interfaces** and **abstract classes**. Write a Java program to implement and access interface methods.
3. Explain **exception handling** in Java. Describe try, catch, throw, throws, and finally with examples.
4. Write a Java program that demonstrates **custom exception handling**. Explain how custom exceptions are declared and used.
5. Discuss the **Singleton Design Pattern** in detail. Write a Java program to implement it and explain how it restricts object creation.
6. What is the **Adapter Design Pattern**? How is it implemented in Java? Write code to demonstrate class-based or object-based adapter.
7. Describe how **multiple catch blocks** and **nested try statements** work in Java. Give a complete example.
8. Explain the difference between **checked** and **unchecked exceptions** with appropriate code snippets.
9. Write a Java program to create a package named mathops containing arithmetic operations. Import and use it in another class.
10. Write a program that handles **file reading** using throws clause. Explain how exception propagation works.

KTU SPECIAL
CONNECT WITH US

WHATSAPP GROUP

FOLLOW US NOW

TELEGRAM CHANNEL

SUBSCRIBE NOW

www.ktuspecial.in

SUBSCRIBE

FOLLOW US

Visit Website