Object Oriented Programming

ICT2122

Introduction to Java Exceptions

P.H.P. Nuwan Laksiri
Department of ICT
Faculty of Technology
University of Ruhuna

Lesson 05

Recap - Interfaces

- Interface - Introduction
- Interface Syntax
- Interface – Hands - On
- Interface - Example
- Interface vs Abstract Class
- Interface – Strict Rules
- Implementing Interfaces
- Adding Fields to an Interface
- Extending Interfaces
- Using an Interface as a Type
- Using Interfaces for Callbacks
- Using default methods
- Using static methods
- Key Things to Remember
- UML - Interface Representation

Outline

- What is an exception
- Reasons for exceptions
- Exception handling
- Types of exceptions
- Throwable class
- Catch exceptions
- Finally block
- Throw/Throws
- Declaring own exceptions

Program Failures

- A program can fail for just about any reason.
  - some of these are coding mistakes
    - Example ?
    - Can handle by the programmer or the language itself
  - Others are completely beyond your control
    - Example ?
    - Cannot handle ,need to deal with the situation

Handling Exceptions

- In the early stages programming languages dealt with them rudely,
  - by abruptly terminating the program and printing out the entire contents of the computer's memory in hexadecimal.
  - called a dump.
- Later programming languages tried various ways to keep the program running when serious errors occurred.
  - Adding special elements
  - Returning error codes
  - Using special error processing section etc.
- Java handles errors by using special exception objects that are created when an error occurs.

The Role of Exceptions in Java

- An exception is Java's way of saying,

  "I give up. I don't know what to do right now. You deal with it."

- When you write a method,
  - you can either deal with the exception or
  - make it the calling code's problem.

What is an Exception?

- The term exception is shorthand for the phrase "exceptional event."
- **An exception**
  - **is an event,**
  - **which occurs during the execution of a program,**
  - **that disrupts the normal flow of the program's instructions.**
- When an exception occurs program processing gets terminated and doesn't continue further.
  - In such cases we get a system generated error message.
  - The good thing about exceptions is that they can be handled.

Reasons for Exceptions

- There can be several reasons for an exception.
  - Opening a non-existing file,
  - Network connection problem,
  - Operands being manipulated are out of prescribed ranges,
  - Class file missing which was supposed to be loaded.
  - User has entered invalid data.

- Some of these exceptions are caused by
  - user error,
  - others by programmer error,
  - and others by physical resources that have failed in some manner.

When an Exception can Occur ?

- Exception can occur
  - at runtime

  (known as runtime exceptions) as well as
  - at compile-time

  (known as compile-time exceptions)

What is Exception Handling?

- Exception Handling is a mechanism to handle runtime errors.

  (such as ClassNotFound, IO, SQL, Remote etc)

- Exception handling ensures that the flow of the program doesn't break when an exception occurs.

Types of Exception

- Exceptions are categorized into three category,
  - Checked Exception
  - Unchecked Exception
  - Error

Checked Exception

- The exception that can be predicted by the programmer.
- All exceptions other than Runtime Exceptions are known as Checked exceptions as the compiler checks them during compilation to see whether the programmer has handled them or not.

  Example :

  File that need to be opened is not found.

- These type of exceptions must be checked at compile time

Checked Exception - Example

```
import java.io.*;
public class CheckedExample
{
        public static void main(String args[])
        {
                FileInputStream fis = new FileInputStream("F:/myfile.txt");
        }
}
```

Solution : The catch-or-throw rule

Unchecked Exception

- Unchecked Exceptions are also known as Runtime Exceptions.
- The compiler do not check whether the programmer has handled them or not but it's the duty of the programmer to handle these exceptions and provide a safe exit.
- Unchecked exception include programming bugs, such as logic errors or improper use of an API etc.
- Unchecked exceptions are ignored at the time of compilation.

Example

Arithmetic error

Division by zero

Unchecked Exception - Example

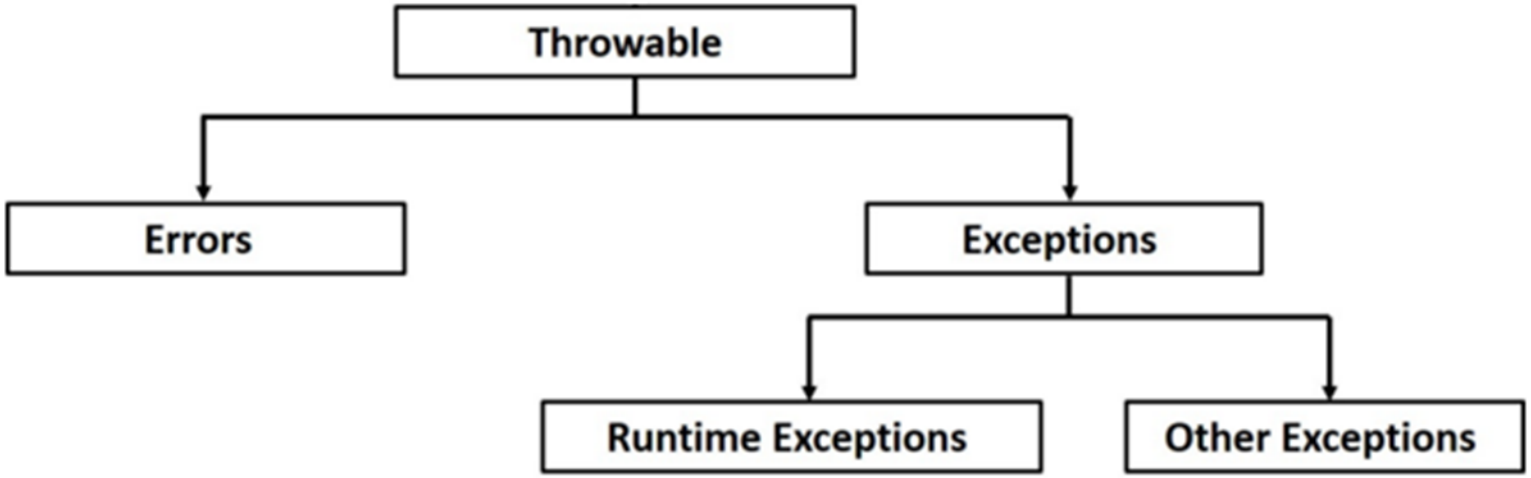
```
public class UncheckedExample
{
        public static void main(String args[])
        {
                int myNumbers[] ={1,2,3,4,5};
                System.out.println(myNumbers[7]);
        }
}
```

Error

- These are not exceptions at all.
- Errors are the problems arise beyond the control of programmer or user.
- Errors are typically ignored in code because you can rarely do anything about an error.
- For example,

  memory error, hardware error, JVM error etc.

Exception Hierarchy



- All exception types and errors are subclasses of class **Throwable**, which is at the top of exception class hierarchy.
- Exception class can be divided into runtime exceptions and other exceptions.

Types of Exceptions - Summary

Type	How to recognize	Okay for program to catch?	Is program required to handle or declare?
Runtime exception	Subclass of `RuntimeException`	Yes	No
Checked exception	Subclass of `Exception` but not subclass of `RuntimeException`	Yes	Yes
Error	Subclass of `Error`	No	No

Most Common Java's Built-in Exceptions

Exception	Description
ClassNotFoundException	Class not found.
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.

Most Common Java's Built-in Exceptions

Exception	Description
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.
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.

Most Common Java's Built-in Exceptions

Exception	Description
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.
UnsupportedOperationException	An unsupported operation was encountered.

Important Methods Available in *Throwable* Class

- public String getMessage()
  - Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor
- public Throwable getCause()
  - Returns the cause of the exception as represented by a Throwable object.
- public String toString()
  - Returns the name of the class concatenated with the result of getMessage()
- public void printStackTrace()
  - Prints the result of toString() along with the stack trace to System.err, the error output stream.

Important Methods Available in *Throwable* Class

- public StackTraceElement [] getStackTrace()
  - Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
- public Throwable fillInStackTrace()
  - Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

Flow Control of a Program Under Exceptions

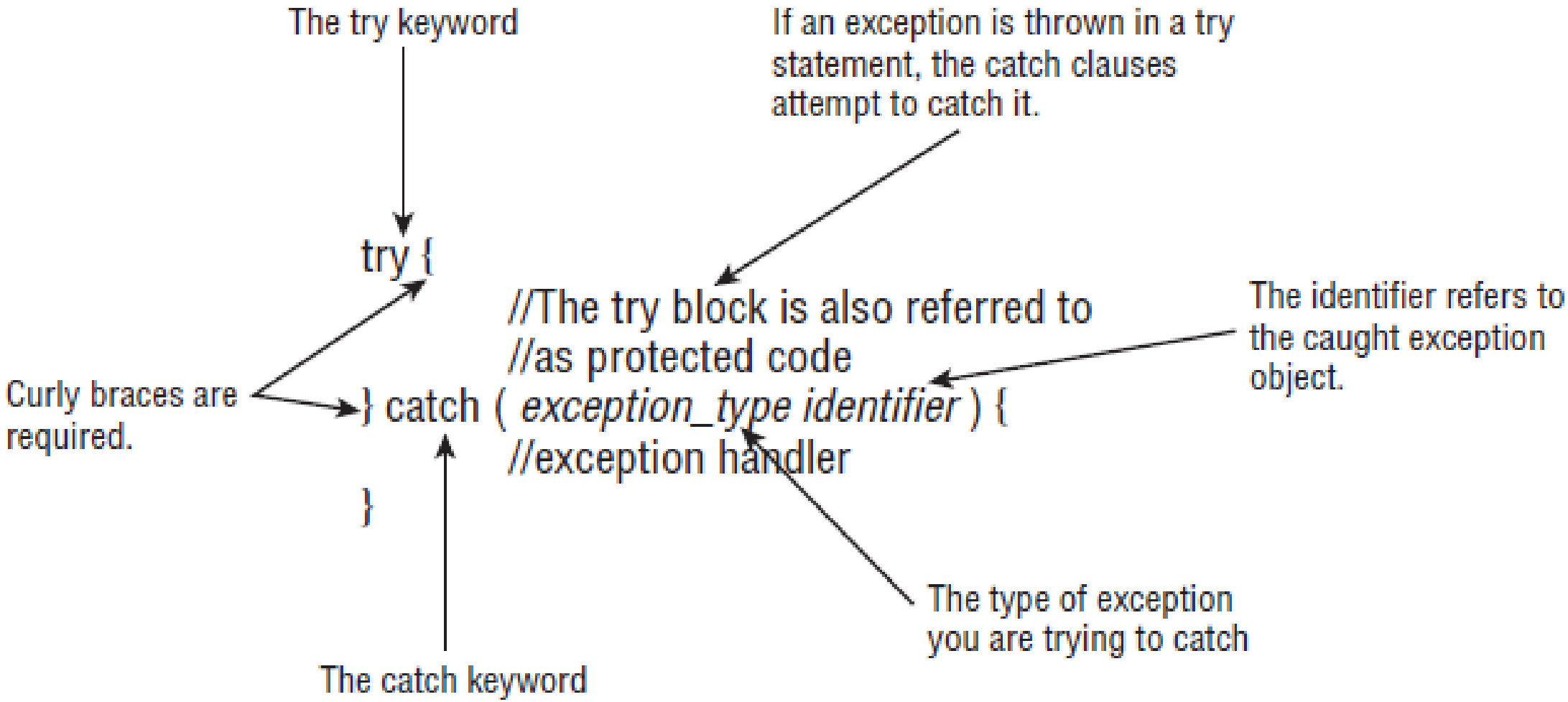
- Exceptions break the normal flow of control.
- When an exception occurs, the statement that would normally execute next is not executed.
- What happens instead depends on:
  - whether the exception is caught,
  - where it is caught,
  - what statements are executed in the 'catch block',
  - and whether you have a 'finally block'.

Handling Exceptions Using a *try* Statement

- A method catches an exception using a combination of the try and catch keywords.
- A try/catch block is placed around the code that might generate an exception.
- Code within a try/catch block is referred to as protected code.

```
try
{
        //statement that could throw an exception
} catch ( ExceptionClassName e1 )
{
        // statements that handle the exception
}
```

The Syntax of a *try* Statement



Example (try/catch)

```
public class ExcepTest{
    public static void main(String args[]){
        try{
                int a[] = new int[2];
                System.out.println("Access element in third index :" + a[3]);
        }catch(ArrayIndexOutOfBoundsException e){
                System.out.println("Exception thrown :" + e);
        }
        System.out.println("Out of the block");
     }
}
```

Output :

Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3 Out of the block

Using multiple catch blocks

```
try
{
        //Protected code
} catch(ExceptionType1 e1) {
        //Catch block to catch exception type 01
} catch(ExceptionType2 e2) {
        //Catch block to catch exception type 02
} catch(ExceptionType3 e3) {
        //Catch block to catch exception type 03
}
```

Using multiple catch blocks - Example

```
try {
        file = new FileInputStream(fileName);
        x = (byte) file.read();
} catch(IOException i) {
        i.printStackTrace();
        return -1;
} catch(FileNotFoundException f) {
        f.printStackTrace();
        return -1;
}
```

Nested try/catch blocks

- The try catch blocks can be nested.
- One try-catch block can be present in another try's body. This is called Nesting of try catch blocks.

- Why use nested try block ?
  - Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error.
  - In such cases, exception handlers have to be nested.

Nested try/catch blocks -Syntax

```
try
{
        statement 1;
        statement 2;
        try
        {
                statement 1;
                statement 2;
        }
        catch(Exception e)
        {
        }
}
catch(Exception e)
{
}
```

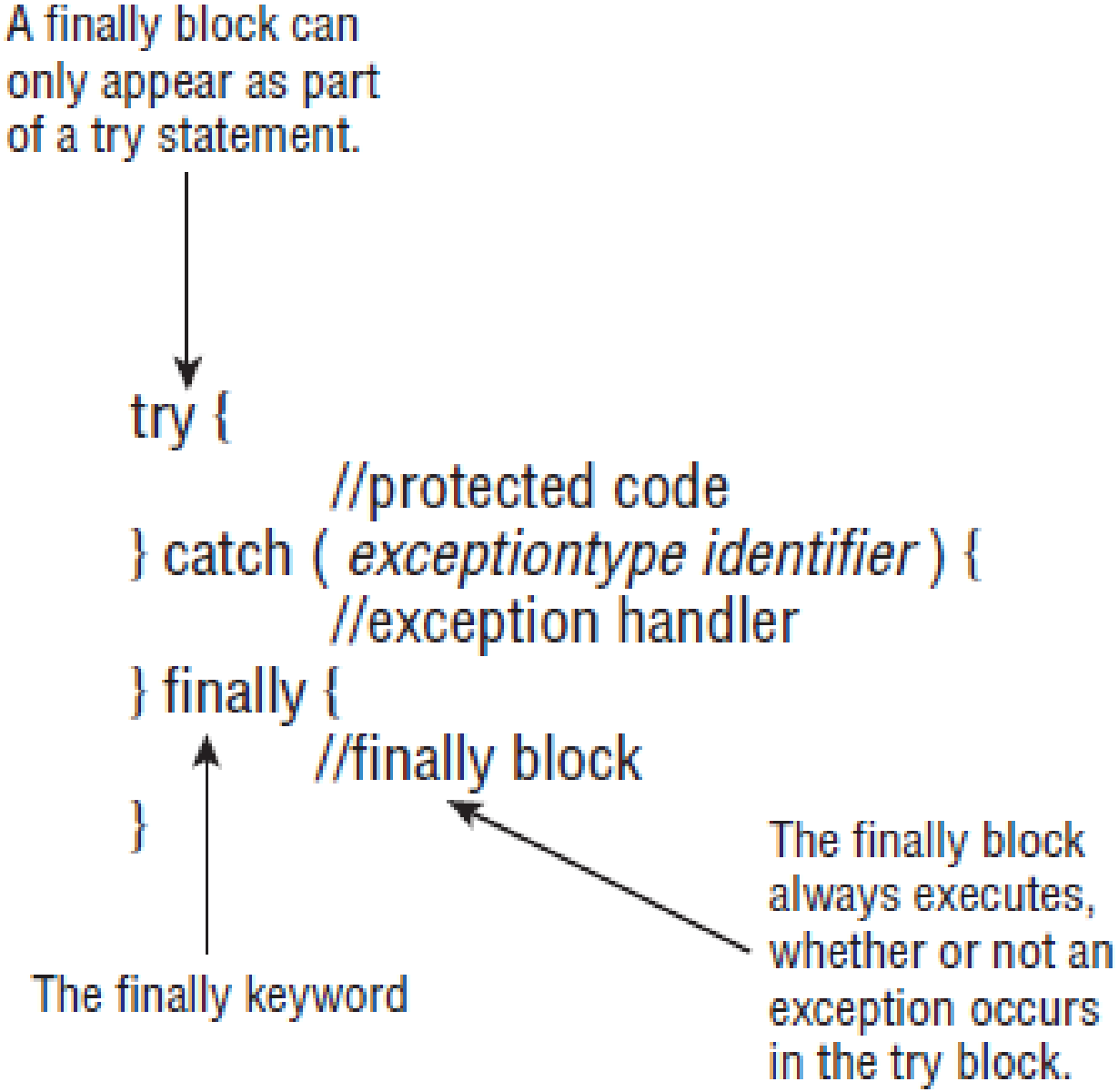
Nested try/catch blocks -Example

```
public class Excepnested{
 public static void main(String args[]){
  try{
                try{
                                System.out.println("going to divide");
                                int b =39/0;
                }catch(ArithmeticException e){
                                System.out.println(e);        }
                try{
                                int a[]=new int[5];
                                a[5]=4;
                }catch(ArrayIndexOutOfBoundsException e){
                                System.out.println(e);
                }
                System.out.println("Other Statement");
    }
    catch(Exception e){
                System.out.println("handeled");
     }
    System.out.println("normal flow..");
  }
}
```

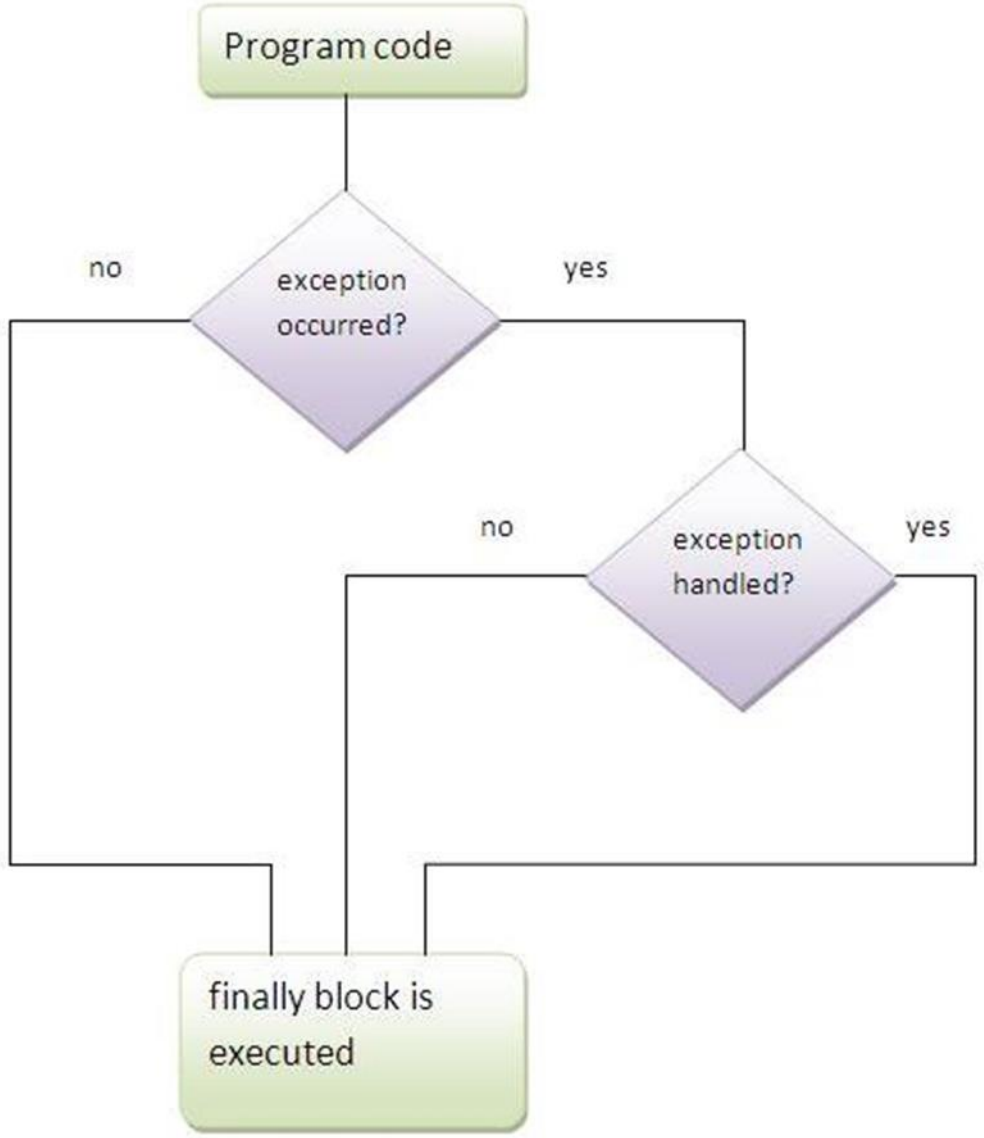
Using *finally* block

- Java finally block is a block that is used to execute important code such as closing connection, stream etc.
- Java finally block is always executed whether exception is handled or not.
- Java finally block must be followed by try or catch block.

The Syntax of a *try* Statement with *finally*



Execution of *finally* Block



Why use *finally* Block?

- Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

- For each try block there can be zero or more catch blocks, but only one finally block.

Using a *finally* Block - Example

```
public class TestFinallyBlock{
  public static void main(String args[]){
  try{
            int data=25/0;
            System.out.println(data);
  }
  catch(NullPointerException e){
            System.out.println(e);
}
  finally{
            System.out.println("finally block is always executed");
}
  System.out.println("rest of the code...");
  }
}
```

Output: finally block is always executed
Exception in thread main java.lang.ArithmeticException:/ by zero

Important Facts

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses when ever 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.

Swallowing *exceptions*

- You can do that by catching the exception in the catch block of a try statement but leaving the body of the catch block empty.

```
public static void openFile(String name){
    try{
            FileInputStream f = new FileInputStream(name);
    } catch (FileNotFoundException e)
    {
    }
}
```

- Swallowing an exception is considered to be bad programming practice.

Java *throw* and *throws* Keywords

- The Java throw keyword is used to explicitly throw an exception.
- We can throw either checked or unchecked exception in java by throw keyword.
- The throw keyword is mainly used to throw custom exception.
  - We will see custom exceptions later.
- The throws keyword appears at the end of a method's signature.

Java *throw* and *throws* Keywords

- Throws clause is used to declare an exception, which means it works like the try-catch block. On the other hand, throw keyword is used to throw an exception explicitly.
- If we see syntax wise than **throw** is followed by an instance of Exception class and **throws** is followed by exception class names.
- 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.
- You can throw one exception at a time, but you can handle multiple exceptions by declaring them using throws keyword.

Java throw and throws Keywords- Example

```
import java.io.*;
public class className {
public void deposit(double amount) throws IOException {
        // Method implementation
                throw new IOException();
        }

        //Remainder of class definition
}
```

Java *throw* and *throws* Keywords- Example

- A method can be declared, so that it throws more than one exception, in which case the exceptions are declared in a list separated by commas.

```
public void withdraw(double amount) throws IOException,
InterruptedException
{
        // Method implementation
}
```

throw Example

```
public class Example{
   void checkAge(int age){
        if(age<18)
          throw new ArithmeticException("Not Eligible for voting");
        else
          System.out.println("Eligible for voting");
   }
   public static void main(String args[]){
        Example1 obj = new Example1();
        obj.checkAge(13);
        System.out.println("End Of Program");
   }
}
```

throws Example

```
public class Example{
   int division(int a, int b) throws ArithmeticException{
          int t = a/b;
          return t;
   }
   public static void main(String args[]){
          Example1  obj = new Example1();
          try{
             System.out.println(obj.division(15,0));
          }
          catch(ArithmeticException e){
             System.out.println("You shouldn't divide number by zero");
          }
   }
}
```

Difference Between *throw* and *throws* in Java

- 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.
- If we see syntax wise than throw is followed by an instance of Exception class and throws is followed by exception class names.
  - throw new ArithmeticException("Arithmetic Exception");
  - throws ArithmeticException;

Difference Between *throw* and *throws* in Java

- 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.
- You can throw one exception at a time but you can handle multiple exceptions by declaring them using throws keyword.
  - void myMethod() {
    throw new ArithmeticException("divided by 0");
    }
  - void myMethod() throws ArithmeticException, NullPointerException{…..}

Declaring Own *exceptions*

- You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes,
  - All exceptions must be a child of Throwable.
  - If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
  - If you want to write a runtime exception, you need to extend the RuntimeException class.

Declaring Own *exceptions*- Example

Syntax:

```
public class MyException extends Exception{
}
```

- You just need to extend the predefined Exception class to create your own Exception.
- These are considered to be checked exceptions.
- The following InsufficientFundsException class is a user-defined exception that extends the Exception class, making it a checked exception.
- An exception class is like any other class, containing useful fields and methods.

Declaring own exceptions- Example

Example :

```
import java.io.*;
public class InsufficientFundsException extends Exception
{
   private double amount;
   public InsufficientFundsException(double amount)
   {
      this.amount = amount;
   }
   public double getAmount()
   {
      return amount;
   }
}
```

Declaring own exceptions - HandsOn

Advantages of Exceptions

- Propagating Errors Up the Call Stack
- Grouping and Differentiating Error Types
- Separating Error-Handling Code from "Regular" Code

Homework

By Providing Examples explain below given statements

- Difference between final, finally and finalize
- By default, Unchecked Exceptions are forwarded in calling chain (propagated).
- By default, Checked Exceptions are not forwarded in calling chain (propagation of exception does not happen).

Homework

- What is “Call Stack” in JAVA.
- What is “Exception Handler” in JAVA.
- Explain how “Exception Handler” use “call stack” to handle the exception.

Summary

- What is an exception
- Reasons for exceptions
- Exception handling
- Types of exceptions
- Throwable class
- Catch exceptions
- Finally block
- Throw/Throws
- Declaring own exceptions

References

- https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

- How To Program (Early Objects)
  - By H .Deitel and P. Deitel
- Headfirst Java
  - By Kathy Sierra and Bert Bates

Questions ???

Thank You