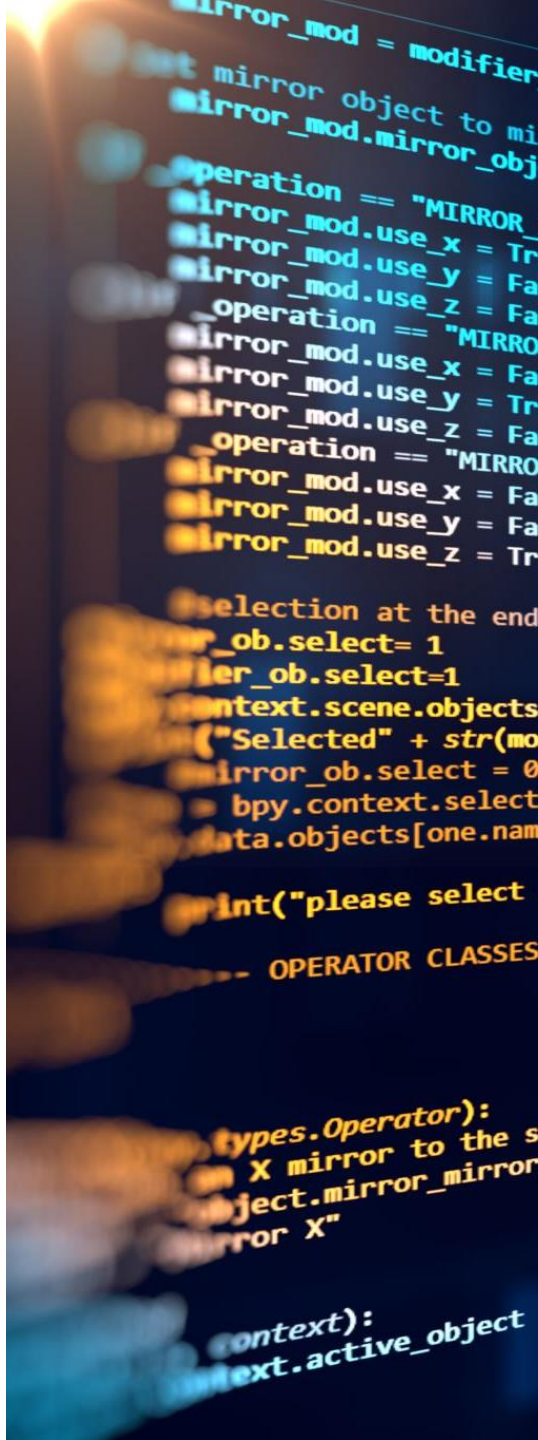
Please use the following QR code to check in and record your attendance.

CS 1027
Fundamentals of Computer Science II

Exceptions in Java

Ahmed Ibrahim



Exception

- An unexpected condition that occurs during runtime.
- Examples include:
  - Accessing an array with an invalid index
  - Dividing a number by zero
  - Incorrectly casting an object to an incompatible type
  - Dereferencing a null object or pointer

Exception

- Such type of situations cause the **Java virtual machine** to throw an exception
- A program can also throw exceptions when it detects an anomalous condition during its **execution**.
- An exception is an object, so an exception must be created with the NEW operator.
- An exception is thrown with the throw statement.

Example

Consider this method:

```
public int foo (int rate) {
    int value = 100 / rate;
    if (value > 50) return value;
    else {
      rate = rate * 2;
      return value – rate * value;
  }
}
```

- What will happen when we call the function with Zero value for its parameter.
- The program will crash with this error message:

```
Exception in thread "main"
  java.lang.ArithmeticException: / by zero
```

Handling Division by Zero

- To prevent the program from crashing, we can test for an invalid division by zero and throw an exception when needed:

```
public int foo (int rate) throws Exception {
    if (rate == 0) throw new Exception();
    int value = 100 / rate;
    if (value > 50) return value;
    else {
      rate = rate * 2;
      return value – rate * value;
   }
}
```

The method declaration must specify that an exception could be thrown

An exception is created with the NEW operator

If an exception is thrown, this part of the method is not executed, and the method terminates

Throwing an Exception vs. Printing an Error Message

- Throwing an Exception stops the method's execution and signals an error to the calling code. This allows the program to handle the error properly through try-catch blocks, ensuring safer error handling and potential recovery.

- Printing an Error Message only **logs** the issue but doesn't stop the execution. This allows the program to continue, which can lead to further errors or unpredictable behavior.

Catching Exceptions

```java
import java.util.Scanner;
public static void main (String[] args) {
   Scanner reader = new Scanner (System.in);
   System.out.println ("Enter new value:");
   int value = reader.nextInt();
   try { int result = foo(value);
      System.out.println ("Result = "+result);
   } catch (Exception e) {
   System.out.println ("Invalid input zero");
   // Call the same function with a default value
   result = foo(DEFAULTRATE);
  }
}
```

Import class Scanner from the Java libraries

Read a number from the keyboard

If an exception is thrown, this statement is not executed

Recover from the wrong input

```java
public int foo (int rate) throws
Exception {
   if (rate == 0) throw new Exception();
   int value = 100 / rate;
   if (value > 50) return value;
   else {
     rate = rate * 2;
     return value – rate * value;
  }
}
```

Exception e

```
import java.util.Scanner;
public static void main (String[] args) {
   Scanner reader = new Scanner (System.in);
   System.out.println ("Enter new value:");
   int value = reader.nextInt();
   try { int result = foo(value);
      System.out.println ("Result = "+result);
   } catch (Exception e) {
   System.out.println("Error:" + e.getMessage());
   recovery();
 }
}
```

This allows another method to recover from the exception

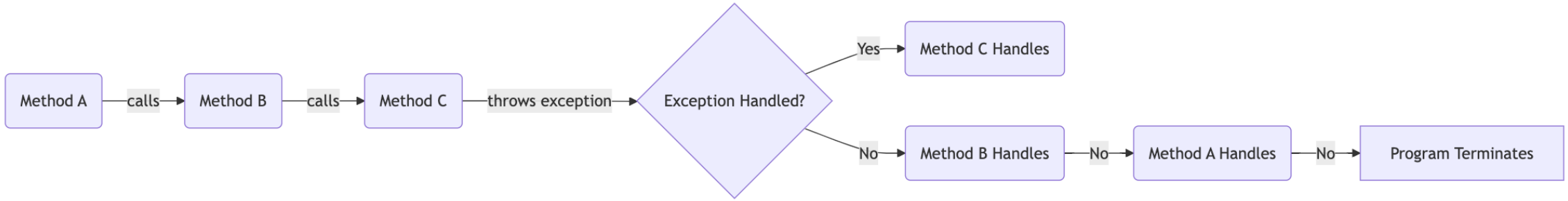
- **e**: The variable representing the exception object (commonly used as a short name).
- An exception contains useful information, such as:
  - The type of exception (e.g., ArithmeticException, NullPointerException).
  - The error message (e.getMessage()).
  - The stack trace (e.printStackTrace()).

How it works?

- When a try-catch statement is executed, the statements in the try block are executed.
- If no exception is thrown:
  - Processing continues as normal; the catch part of the try-catch statement is not executed.
- If an exception is thrown:
  - When an exception is thrown, the program enters a state of urgency, often called 'panic mode'. Control is immediately passed to the first catch block whose specified exception corresponds to the class of the thrown exception.

Exception Propagation

- If an exception isn't handled in the method where it occurs, control passes to the calling method.
- Control continues up the call stack if that method doesn't handle it.
- This process is called exception propagation.
- Exception propagation continues until:
  - The exception is caught, or
  - The program reaches the main method and terminates.



Predefined vs. Custom Exceptions

```
public static int foo(int rate) throws Exception {
        // Throw an exception for invalid input values
        if (rate == 0) throw new ArithmeticException("Division by zero is not allowed.");
        if (rate < 0) throw new IllegalArgumentException("Rate cannot be negative.");
        // Potential division by zero is covered
        int value = 100 / rate;
        // Adding another exception for an unusual condition
        if (value > 1000) throw new Exception("Result too large to handle.");
        if (value > 50) {
            return value;}
        else
           {rate = rate * 2;
            return value - rate * value; }
}
```

Predefined exceptions in Java

Custom Exception for Large Results

Catching Exceptions

```
try { result = foo(value);
System.out.println("Result = " + result);}
catch (ArithmeticException e) {
   System.out.println("Error: Division by zero.");
   result = foo(DEFAULTRATE);}
catch (IllegalArgumentException e) {
   System.out.println("Error: Invalid input. Rate
cannot be negative.");
   result = foo(DEFAULTRATE);}
catch (Exception e) {
   System.out.println("Error: " + e.getMessage());
   recovery();}

System.out.println("Final result = " + result);
```

Control passes here after the exception

Code Snippet

Multiple Exceptions

- A **single** catch block can handle more than one type of exception.
- In the catch clause, we specify the types of exceptions that the block can handle and separate each exception type with a vertical bar (|).

The try-catch syntax:

```
try {
  // you code here
 code1
}
    catch(Exception1 e) {statements}
    catch(Exception2 e) {statements}
    catch(Exception3|Exception4 e){
    statements}
code2
```

If the **code** throws an exception of type Exception1, only these statements are executed and then control passes to code2

Pipe Operator

This combined catch block will execute the same statements if **either** Exception3 or Exception4 is thrown.

Practical Exercise:

Consider the following code snippet from the **ExceptionExample** class:

```
public class ExceptionExample {
    private static int x = 1;
    private static String s = "";

    public static void main(String[] args) {
        try {
            method1(2);
            method1(1);
            x = x + 3;
        } catch (Exception1 e) {
            x = 0;
        } catch (Exception2 ex) {
            x = x + 5;
        }
        System.out.println(x + ", " + s);
    }
```

What will the output of the program be?

A) 4, hello

B) Empty string

7, hello

C) Empty string

4, hi

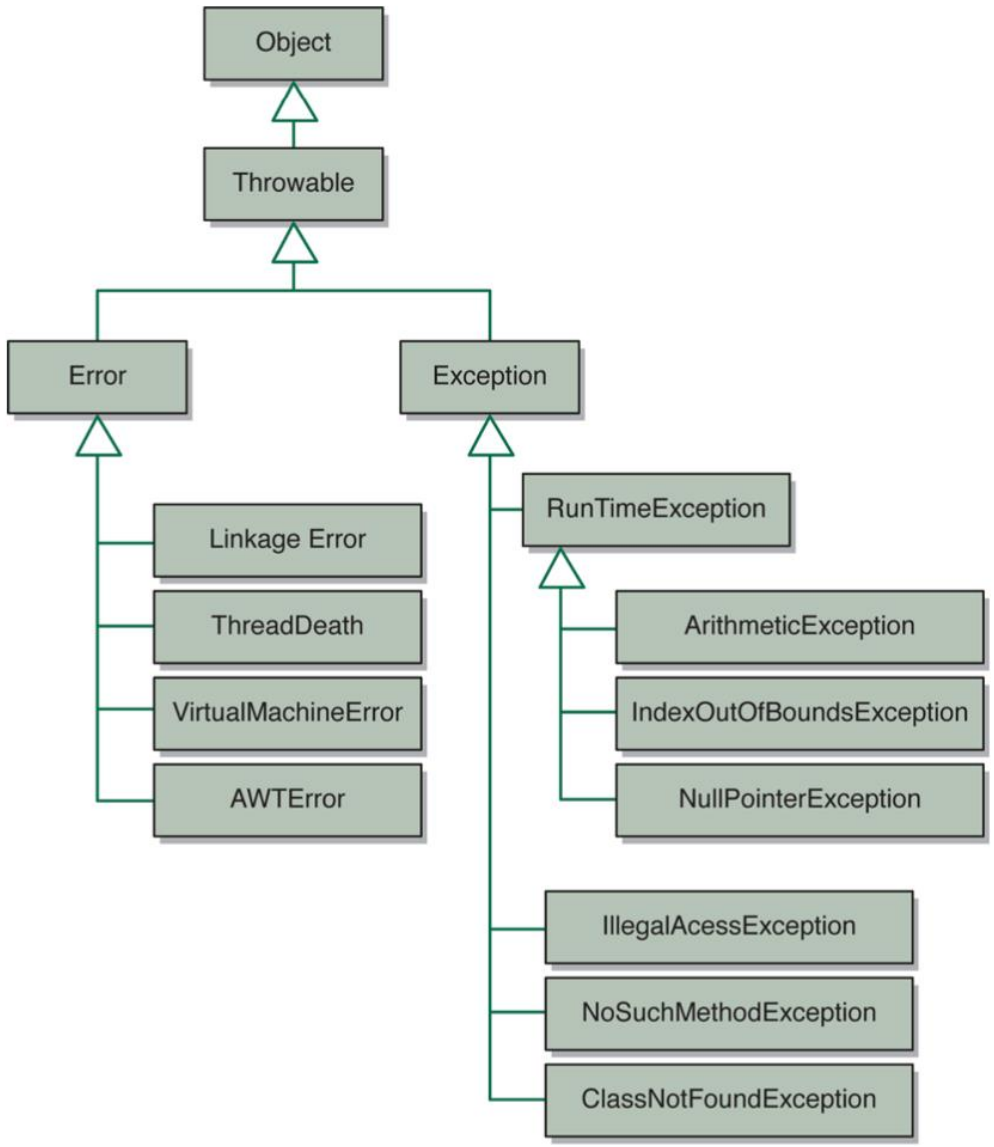
D) 5, hello

```
private static void method1(int param) throws
Exception1, Exception2 {
        try {
            if (param == 1) method2("hello");
            else method2(s);
            ++x;
        } catch (Exception1 e) {
            System.out.println(e.getMessage());
            s = "hi";
        }
    }

    private static void method2(String str) throws
Exception1, Exception2 {
        if (str.length() > 0) {
            ++x;
        } else {
            throw new Exception1("Empty string");
        }
        s = "hello";
        throw new Exception2("Long string");
    }
}
```

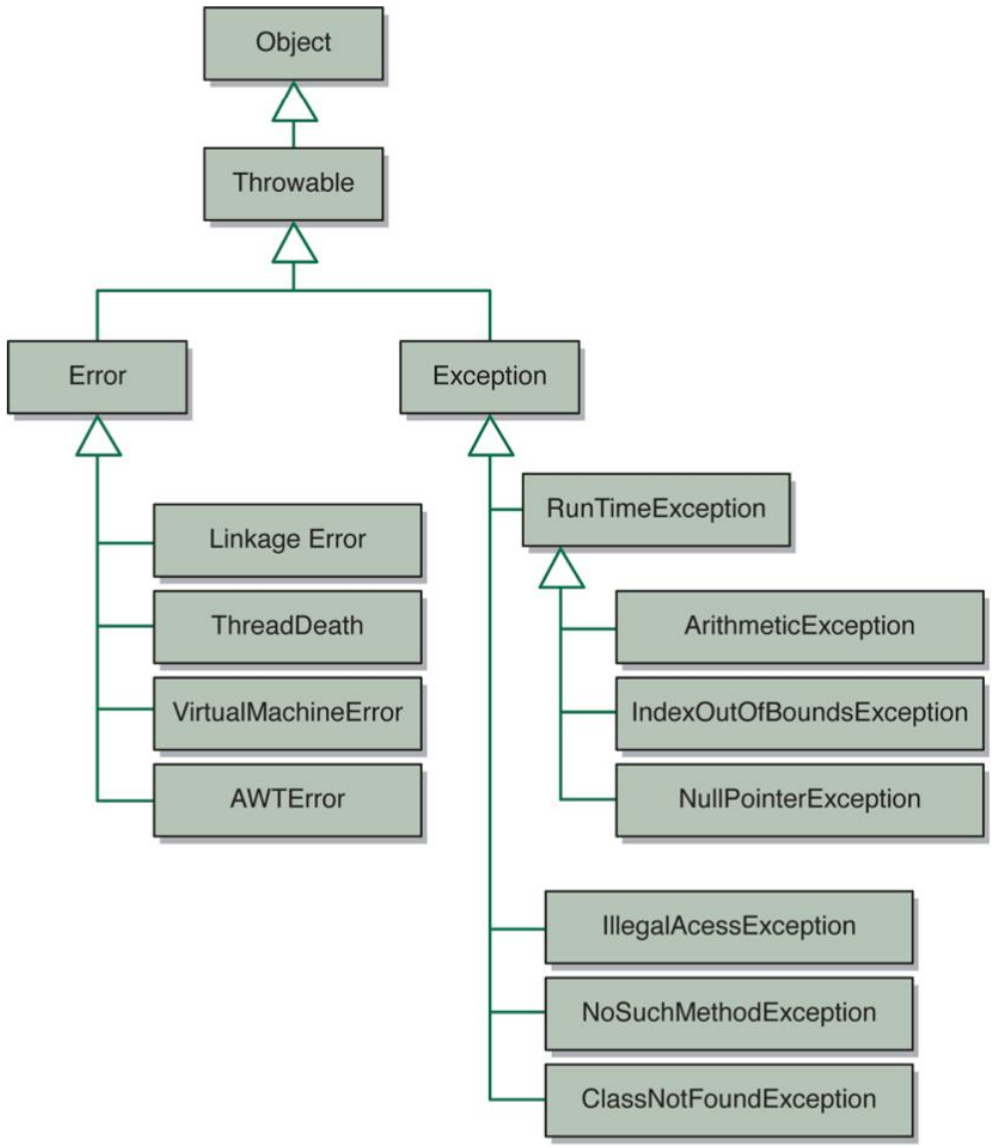
Java Exceptions

- In Java, an exception is an Object
- There are Java predefined exception classes, like
  - ArithmeticException
  - IndexOutOfBoundsException—This exception is thrown when accessing an index that is outside the bounds of an array, list, or other data structure that uses indexes.
  - IOException – This exception is thrown when an input or output operation fails or is interrupted. It commonly occurs during **file handling** or **network operations**.
  - NullPointerException – This exception is thrown when you try to access an object or call a method on an object that is null. <u>It is one of the most common exceptions in Java</u>.



Declaring Exception Classes

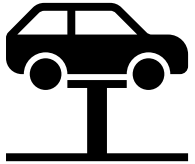
```
public class MyException extends
RuntimeException {
public MyException (String msg) {
super (msg);
  }
}
```



Runtime Errors vs. Exceptions

- Java differentiates between **runtime errors** and **exceptions**.
- Errors are unrecoverable situations, so the program must be terminated
- Example: <u>running out of memory</u>

- Exceptions are abnormal or erroneous situations detected at **runtime** and from which a program can recover
- Examples:
  - Division by zero
  - Array index out of bounds
  - Null pointer exception



Compiler & Exceptions

```java
import java.io.*;
 class Main {
  public static void main(String[] args) {
   FileReader file = new FileReader("test.txt");
   BufferedReader fileInput = new BufferedReader(file);
   System.out.println(fileInput.readLine());
   fileInput.close();
 }
}
```

The compiler gives the error:

Main.java:5: error: unreported exception FileNotFoundException; must be caught or declared to be thrown `FileReader file = new FileReader("test.txt");`

Checked Exceptions

We fix the compilation error like this:

```
import java.io.*;
class Main {
public static void main(String[] args) {
try {
  FileReader file = new FileReader("test.txt");
  BufferedReader fileInput = new BufferedReader(file);
  System.out.println(fileInput.readLine());
  fileInput.close();
}
catch (FileNotFoundException e) { ... }
catch (IOException e) { ... }
  }
}
```

Unchecked Exceptions

```
class Main {
  public static void main(String[] args) {
     int x = 10;
     int y = 0;
     int z = x / y;
   }
}
```

The compiler does not give an error even though we are dividing by zero.

try-catch-finally Blocks

- The finally block always executes when the try block exits, whether an exception was thrown or not (even if the exception was not caught by any of the catch statements!)
- The finally block is executed even if there is a return statement inside the try or catch blocks or if a new exception is thrown.

The try-catch syntax:

```
try {
  // you code here
 code1
}
    catch(Exception1 e) {statements}
    catch(Exception2 e) {statements}
    catch(Exception3|Exception4 e){
    statements}
    finally {statements}
```

No finally Block

- In this example, the exception caused by dividing 5 by 0 (an ArithmeticException) is not caught, because there's no specific catch block.
- As a result, the file is not closed (out.close() is not executed).

```
PrintWriter out;
 try {
    out = new PrintWriter(new
FileWriter("OutFile.txt"));
    out.println("Data");
    // ArithmeticException occurs here
    int x = 5 / 0;
}
catch(FileNotFoundException e) {…}
catch(IOException e) {…}

if (out != null) out.close();
```

Code with finally Block

- In this version, the finally block ensures that the file will always be closed, even if an exception occurs, such as an ArithmeticException.
- This is important for resource management, ensuring that no file handles or resources remain open unintentionally.

```
PrintWriter out = null;
try {
    out = new PrintWriter(new
FileWriter("OutFile.txt"));
    out.println("Data");
    // ArithmeticException occurs here
    int x = 5 / 0;}

catch(FileNotFoundException e) {…}
catch(IOException e) {…}

finally  {
    // This ensures the file is closed
    if (out != null) out.close();
}
```


thank you