

Chapter 6:

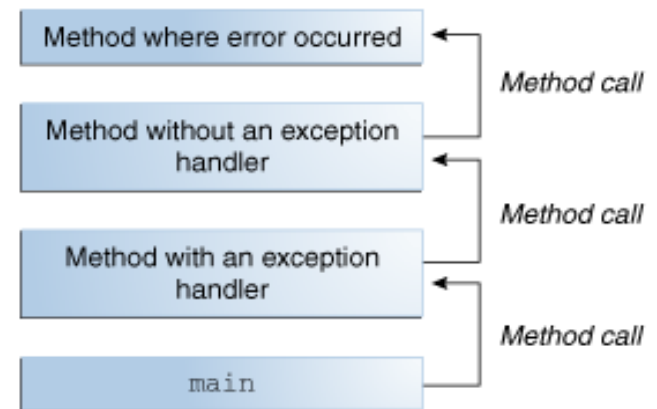
Exception Handling

Exception

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 error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an **exception object**, contains information about the error, including its type and the state of the program when the error occurred.

After a method throws an exception object and the runtime system is called, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the



The call stack

Exception

The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an **exception handler**.

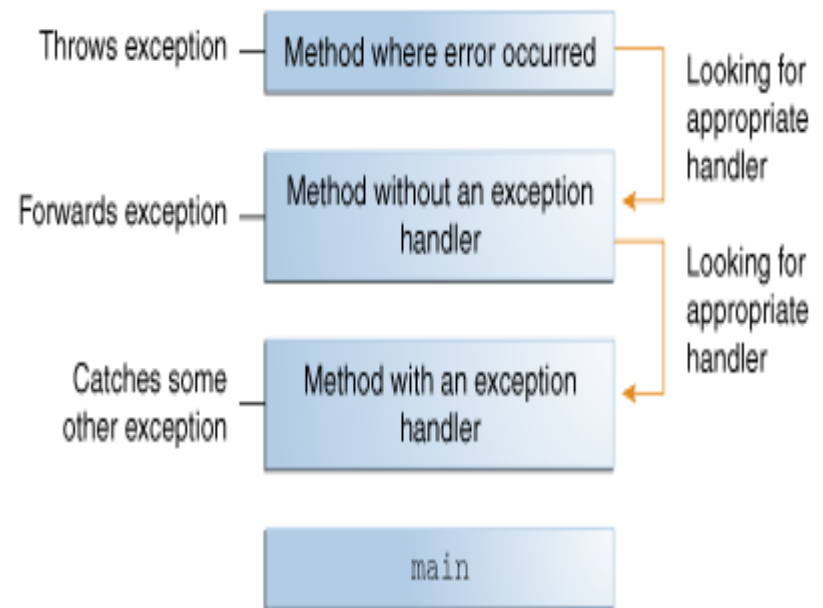
The search begins with the method in which the error occurred and proceeds through **the call stack in the reverse**

order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler.

An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.

The exception handler chosen is said to **catch the exception**.

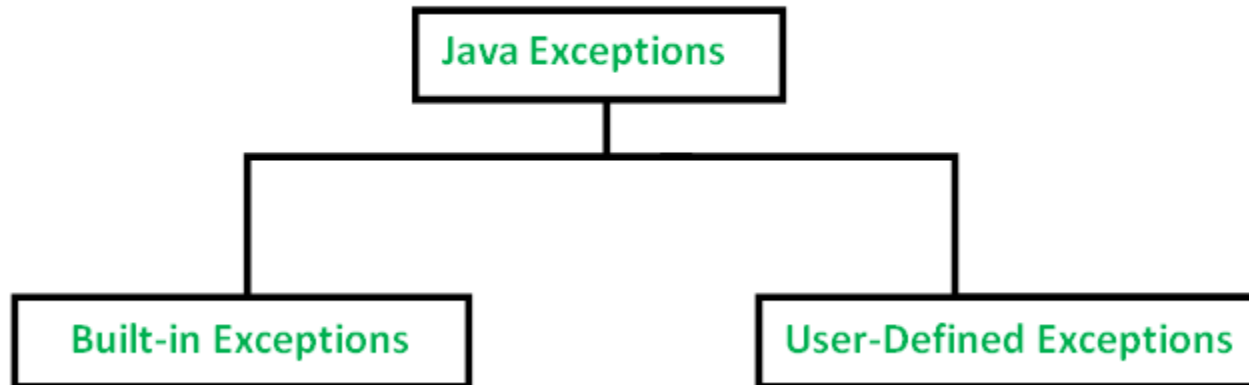
If no appropriate exception handler is found, the runtime system



Exception Classification

Java exception can be classified in to two:

1. Built-in Exceptions
2. User defined Exceptions



Built-in Exceptions

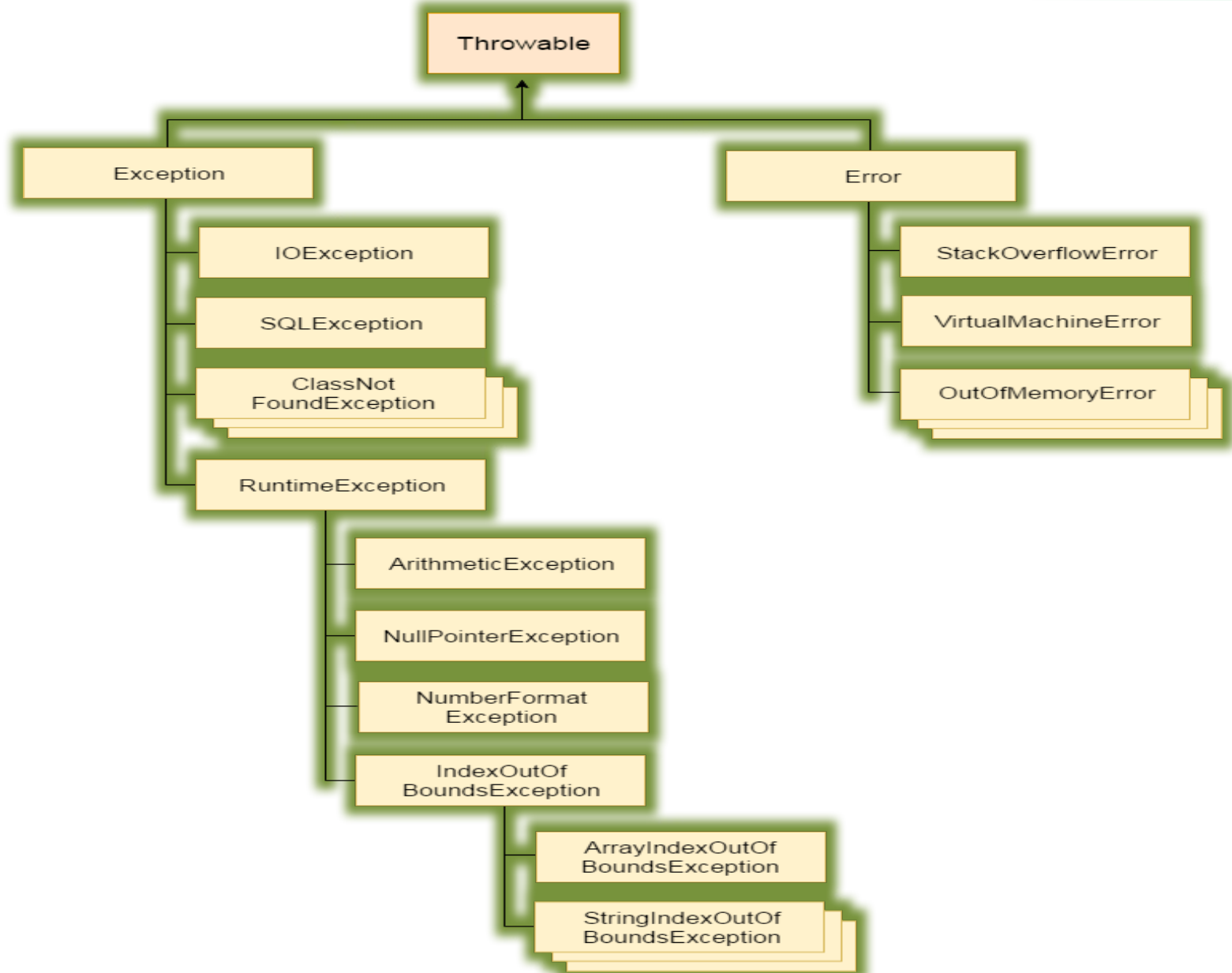
Built-in exceptions are the exceptions which are available in Java libraries

1. **ArithmeticException**: Arithmetic error, such as divide-by-zero.
2. **ArrayIndexOutOfBoundsException**: Array index is out-of-bounds.
3. **ArrayStoreException**: Assignment to an array element of an incompatible type.
4. **ClassCastException**: Invalid cast.
5. **IllegalArgumentException**: Illegal argument used to invoke a method.
6. **IllegalMonitorStateException**: Illegal monitor operation, such as waiting on an unlocked thread.
7. **IllegalStateException**: The Java environment or Java application is not in an appropriate state for the requested operation. Signals that a method has been invoked at an illegal or inappropriate time. In

Built-in Exceptions

- 7. `IllegalThreadStateException`:** Requested operation not compatible with the current thread state.
- 8. `IndexOutOfBoundsException`:** Some type of index is out-of-bounds.
- 9. `NegativeArraySizeException`:** Array created with a negative size.
- 10. `NullPointerException`:** Invalid use of a null reference.
- 11. `NumberFormatException`:** Invalid conversion of a string to a numeric format.
- 12. `SecurityException`:** Attempt to violate security.
- 13. `StringIndexOutOfBoundsException`:** Attempt to index outside the bounds of a string.
- 14. `UnsupportedOperationException`:** An

Hierarchy of Java Exception classes



Error: is a subclass of the built-in class “Throwable”. Errors are **the critical conditions** that occur due to the lack of **the system resources**, and it can **not be handled by the code** of the program.

- Errors **can not be recovered** by any means because they can not be created, thrown, caught or replied. Errors are caused due to the **catastrophic failure** which usually can not be handled by your program.
- **Errors are always of unchecked type**, as compiler do not have any knowledge about its occurrence.
- **Errors always occur in the run-time environment.** The error can be explained with the help of an example, the program has an error of stack overflow, out of memory error, or a system crash error, this kind of error are due to the system.
- **The code is not responsible for such errors.** The consequence of the occurrence of the error is that **the program gets terminated abnormally.**

Types of Exception

There are generally two classes of exceptions

1. Checked Exception

These are exception that **are checked at compile time.**

These are not subclass of RuntimeException and Error

- Checked exceptions are the type that programmers should anticipate and from which programs should be able to recover
- All Java exceptions are checked exceptions except those of the Error and RuntimeException classes and their subclasses
- A checked exception is an exception which the Java source code must deal with, either by catching it or declaring it to be thrown.
- Checked exceptions are generally caused by faults **outside of the code itself** - missing resources, networking errors, and problems with threads come to mind.

Types of Exception...

```
import java.io.*;
class Example {
    public static void main(String args[]) {
        FileInputStream fis = null;
        /*This constructor FileInputStream(File filename)
        throws FileNotFoundException which is a checked
        exception*/
        fis = new FileInputStream("B:/myfile.txt");
        int k;
        /* Method read() of FileInputStream class also throws
        a checked exception: IOException*/
        while(( k = fis.read() ) != -1) {
            System.out.print((char)k);
        }
        /*The method close() closes the file input stream It throws
        IOException*/
        fis.close();
    }
}
```

**Exception in thread
"main"
java.lang.Error:
Unresolved
compilation
problems:
Unhandled
exception type
FileNotFoundException
Unhandled
exception type
IOException
Unhandled
exception type
IOException**

**Since we didn't handled/declared the exceptions, our
program gave the compilation error.**

Types of Exception...

2. Unchecked Exception

Unchecked exceptions are **not checked at compile-time**, but **they are checked at runtime**. Compiler do not have any knowledge about its occurrence. **Errors and**

RuntimeException are unchecked exception

- Unchecked exceptions inherit from the Error class or the RuntimeException class
- RuntimeException is caused by **a misuse of code** by the programmer - passing a null or otherwise incorrect argument. The program **can reasonably recover from it**. Programmer may prevent them from occurring. Example NullPointerException, IllegalArgumentException, ArrayIndexOutOfBoundsException
- Error is caused due to lack of system resources. It represent the type of errors from which programs cannot

Types of Exception...

```
class Example {
    Scanner input=new Scanner("System.in);
    public static void main(String args[])
    {
        double num1, num2;
        System.out.println("Enter the dividend and the divisor: ");
        num1=input.nextInt();
        num2=input.nextInt();
        /*if the user supply 0 for num2
        * it should throw ArithmeticException
        */
        int res=num1/num2;
        System.out.println(res);}
```

If you compile this code, it would compile successfully however when you will run it, it may throw ArithmeticException. That clearly shows that unchecked exceptions are not checked at compile-time, they occurs

Exceptions constructors

All types of Exception have two constructors like

1. No argument Exception constructor which initializes a new instance of the exception class

```
NullPointerException()
```

```
NullPointerException e=new
```

```
NullPointerException();
```

2. String argument exception constructor which initializes a new instance of the exception class with a specified error message.

```
IllegalArgumentException e =
```

```
new IllegalArgumentException("number must  
be positive");
```

Exceptions Method

All exception classes have the following methods:

1. `public String getMessage()`: Returns a detailed message about the exception that has occurred. This message is initialized in the `Throwable` constructor.
2. `public String toString()`: Returns the name of the class concatenated with the result of `getMessage()`.
3. `public void printStackTrace()`: Prints the result of `toString()` along with the stack trace to `System.err`, the error output stream.
4. `public StackTraceElement [] getStackTrace()`: Returns an array containing each element on the stack trace. The element at index 0

Exception Message Format

Exception message format:

[exception class]: [additional description of exception]

at [class].[method]([file]:[line number])

Example:

java.lang.ArrayIndexOutOfBoundsException: 2

at ExceptionExample.main(ExceptionExample.java:4)

- What exception class?
`ArrayIndexOutOfBoundsException`
- Which array index is out of bounds?
`2`
- What method throws the exception?
`main`
- What file contains the method?
`ExceptionExample.java`
- What line of the file throws the exception?

Resolving Exception

Java exception can be resolved or handled in to ways:

1. Using try-catch mechanism
2. Using the throws keywords

Exception Resolving-Try Catch in Java

Try and catch both are Java keywords and used for exception handling. The try block is used to enclose the suspected code. Suspected code is a code that may raise an exception during program execution.

The catch block also known as handler is used to handle the exception. It handles the exception thrown by the code enclosed into the try block

Syntax

```
try {  
    // suspected code  
} catch (ExceptionType1 e1) {  
    // handler code 1  
} catch (ExceptionType2 e2) {  
    // handler code 2  
} catch (ExceptionType3 e3) {  
    // handler code 3  
}
```

If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is

```
import java.io.*;
class Example {
    public static void main(String args[])
    {
        FileInputStream fis = null;
        try{
            fis = new
FileInputStream("B:/myfile.txt");
            }catch(FileNotFoundException fnfe){
                System.out.println("The specified file
is not " +
                    "present at the given path");
            }
        int k;
```

```
try{  
    while(( k = fis.read() ) != -1)  
    {  
        System.out.print((char)k);  
    }  
    fis.close();  
}catch(IOException ioe){  
    System.out.println("I/O error occurred:  
"+ioe);  
}  
}
```

Try Catch in Java...

```
class Example2{  
    public static void main(String args[]){  
        Scanner reader=new Scanner(System.in);  
        int idx;  
        try{  
            float temp[]=new float[7];  
            for(int i=0; i<7;i++){  
                System.out.println("Enter temperature for  
day "<<temp[i]<<" : ");  
                temp[i]=reader.nextFloat();  
            }  
            System.out.println("Which day temperature you  
want to see: ");  
            idx=reader.nextInt();  
            System.out.println("Temperature for day " +
```

Try Catch in Java...

```
    }  
    catch(NumberFormatException e){  
        System.out.println("Invalid input  
format!!!");  
    }  
    catch(ArrayIndexOutOfBoundsException e){  
        System.out.println("Invalid value for the  
day!!!");  
    }  
    catch(Exception e){  
        System.out.println("Unexpected error  
occurred!!!");  
    }  
    System.out.println("Have a gooday!!!");  
}
```

The finally Block

A try-catch block may include a finally block. A finally block contains all the crucial statements that must be executed whether exception occurs or not. The statements present in this block will always execute regardless of whether exception

occurs in try block or not
try {
 connection, stream etc

//Statements

that may cause an
exception

}

catch {

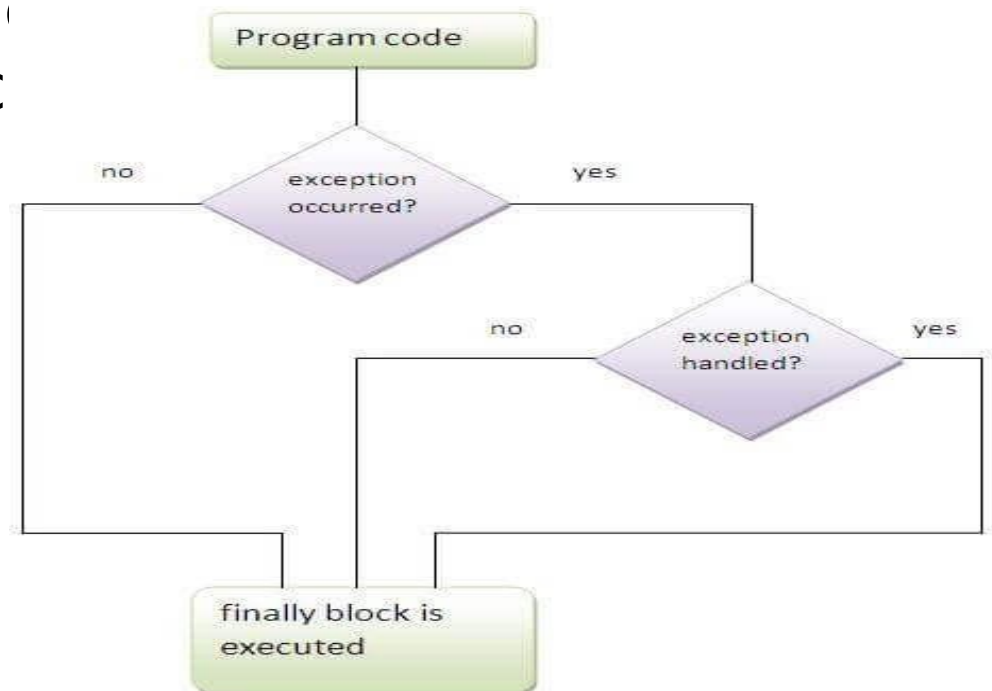
//Handling

exception

}

finally {

//Statements to



Exception Resolving-Throws Keyword

One way to overcome exception is by using throws :
By declaring the exceptions in the method signature using throws and handle the exceptions where you are calling this method by using try-catch.

The throws keyword indicates what exception type may be thrown by a method.

.It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Syntax of java throws

```
accessModifier returnType methodName()  
throws ExceptionType1, ExceptionType2 ... {  
    // code  
}
```

Exception Resolving-Throws...

```
import java.io.*;
class Main {
    public static void findFile() throws
NullPointerException, IOException,
InvalidClassException {
    // code that may produce NullPointerException
    ... ..

    // code that may produce IOException
    ... ..

    // code that may produce InvalidClassException
    ... ..

}
```

Exception Resolving-Throws...

```
public static void main(String[] args) {  
    try{  
        findFile();  
    } catch(IOException e1){  
        System.out.println(e1.getMessage());  
    } catch(InvalidClassException e2){  
        System.out.println(e2.getMessage());  
    }  
}  
}
```

Exception Resolving-Throws...

```
import java.io.*;
class Example {
    public static void main(String args[]) throws IOException
    {
        FileInputStream fis = null;
        fis = new FileInputStream("B:/myfile.txt");
        int k;

        while(( k = fis.read() ) != -1)
        {
            System.out.print((char)k);
        }
        fis.close();
    }
}
```

Throwing and Overriding

An overriding method can throw any unchecked exceptions, regardless of whether the overridden method throws exceptions or not. However, the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method

Creating Custom Exception

1. To create a custom checked exception, we have to extend the `java.lang.Exception` class.

```
public class IncorrectFileNameException extends  
Exception {  
    public IncorrectFileNameException(String  
errorMessage) { super(errorMessage);  
    }  
}
```

2. To create a custom unchecked exception we need to extend the `java.lang.RuntimeException` class:

```
public class InvalidAgeException extends  
RuntimeException {  
    public InvalidAgeException(String errorMessage) {  
        super(errorMessage);  
    }  
}
```

Throwing an Exception

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 can be used to throw both built-in and custom exceptions.

Syntax:

throw ExceptionInstance;

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.

Example

```
throw new Exception("Exception message");
```

Throwing an Exception

```
public static int average(int[] a) {  
    if (a.length == 0) {  
        throw new  
        IllegalArgumentException("array is empty");  
    }  
    int total = 0;  
    for(int i = 0; i < a.length; i++) {  
        total += a[i];  
    }  
    return total / a.length;  
}
```

```
class Example1{
    void method1() throws ArithmeticException{
        throw new ArithmeticException("Calculation error"); }
    void method2() throws ArithmeticException{
        method1(); }
    void method3(){
        try{
            method2();
        }
        catch(ArithmeticException e){
            System.out.println("ArithmeticException handled");
        } }
    public static void main(String args[]){
        Example1 obj=new Example1();
        obj.method3();
        System.out.println("End Of Program");
    }
}
```

What is the output?