Sri Lanka Institute of Information Technology
Faculty of Computing

SE1020 - Object-Oriented Programming

Year 01 and Semester 02

Lecture 06
Exception Handling

Recall. . .

- Understand and apply abstract classes to model common but incomplete behaviors.
- Use interfaces for designing flexible and extensible code.
- Implement and explain the purpose of the static modifier in Java programs.

Learning Objectives

- Understand different types of errors and exceptions.
- Differentiate between compile-time and run-time errors.
- Describe the Java exception hierarchy.
- Use try-catch-finally for exception handling.
- Create and use custom exceptions.

Introduction

Rarely does a program run successfully at its very first attempt. It is common to make mistakes while developing as well as typing a program. A mistake might lead to an error causing the program to produce unexpected results. Errors are the wrongs that can make a program go wrong.

nAn error may produce an incorrect output or may terminate the execution of the program abruptly or even may cause the system to crash. It is therefore important to detect and manage properly all the possible error conditions in the program so that the program will not terminate or crash during execution.

What is an Error?

- An error is an issue that prevents a program from continuing normally.
- Java errors can occur during compilation or execution.

Types of Errors

- Compile-Time Errors
  - Detected during compilation
  - Syntax mistakes, undeclared variables, etc.
- Run-Time Errors
  - Occur during program execution
  - Division by zero, null pointer access, etc.
- Logical Errors
  - Program runs without crashing, but gives incorrect results
  - Caused by flaws in the program logic

Compile - Time Errors

All syntax errors will be detected and displayed by the Java compiler and therefore these errors are known as compile time errors. Whenever the compiler displays an error, it will not create the .class file. It is therefore necessary that we fix all the errors before we can successfully compile and run the program.


```
Program
public class Error1 {

    public static void main(String args[]) {
        System.out.println("Hello Java!")
                // Missing ;
    }
}
```

Error Described interrupting the compilation

```
Error1.java :5: ';' expected System.out.println
("Hello Java! ")

1 error
```

The Java compiler does a nice job of telling us where the errors are in the program. For example, if we have missed the semicolon at the end of print statement in the above Program, the following message will be displayed in the screen.

Most of the compile-time errors are due to typing mistakes. The most common problems are:

- Missing semicolons
- Missing (or mismatch of) brackets in classes and methods
- Misspelling of identifiers and keywords
- Missing double quotes in strings
- Use of undeclared variables
- Incompatible types in assignments, initializations.
- Bad references to objects
- Use of = in place of == operator

Run - Time Errors

Runtime errors are errors that occur while a program is running, **after it has successfully compiled**. These errors are NOT detected by the compiler but cause the program to crash or behave unexpectedly during execution.

- Dividing an integer by zero
- Accessing an element that is out of the bounds of an array
- Trying to store a value into an array of an incompatible class or type
- Trying to cast an instance of a class to one of its subclasses
- Passing a parameter that is not in a valid range or value for a method
- Attempting to use a negative size for an array
- Incompatible types in assignments, initializations

Example : Run-Time Errors

```java
class Error2 {
        public static void main (String args[ ]) {
                int a = 10;
                int b = 5;
                int c = 5;
                int x = a / (b-c) ; // Division by zero

                System.out.println("x = " + x) ;
                int y = a / (b+c);
                System.out.println("y = " + y);
        }
}
```

Program is syntactically correct and therefore does not cause any problem during compilation. However, while executing, it displays the following message and stops without executing further statements.

java. lang.ArithmeticException: / by zero at Error2.main(Error2.java:8)

When Java run-time tries to execute a division by zero, it generates an error condition, which causes the program to stop after displaying an appropriate message.

What is an Exception?

An exception is an abnormal event that arises during the execution (run time) of the program and disrupts the normal flow of the program.

- An exception is an object that represents an abnormal condition.
- It interrupts the normal flow of the program.
- Java provides a structured way to handle exceptions using try-catch blocks.

Exception Hierarchy in Java
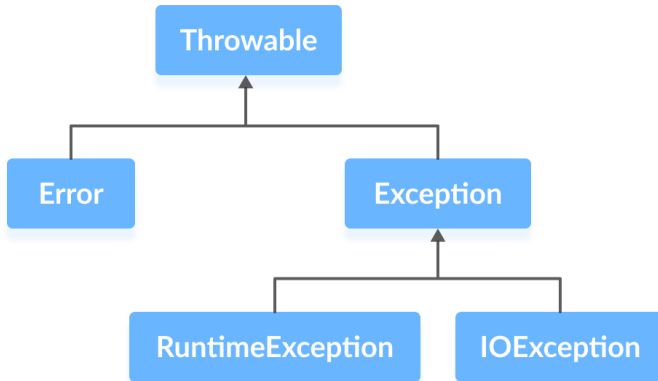
Error Class

- An Error is a subclass of Throwable that indicates serious problems.
- Errors represent critical errors, once occurs the system is not expected to recover from (irrecoverable).
- Errors can be generated from mistake in program logic or design.
- Most applications should not try to handle it.

e.g. OutOfMemoryError, VirtualMachineError, AssertionError

Exception Class

This class represents the exceptional conditions that user must handle or catch.

- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error.
- That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed.
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.

Types of Exceptions

- Checked Exceptions : Checked exceptions are exceptions that are checked at compile-time. These exceptions must be either caught or declared in the method signature using the throws keyword.

  Examples: IOException, SQLException, FileNotFoundException

- Unchecked Exceptions : Unchecked exceptions are exceptions that are not checked at compile-time. They are subclasses of RuntimeException. Examples: NullPointerException,

  ArrayIndexOutOfBoundsException, ArithmeticException

Type	Handled When	Examples
Checked Exception	Compile-time	IOException, SQLException
Unchecked Exception	Runtime	NullPointerException, ArithmeticException

Exception Handling

There are two ways to handle Exceptions:

- Catch and handle

  If a code segment is generating an Exception place it inside a try block. Then mention the way to handle the Exception inside a catch block use try-catch blocks

- Throw out and ignore

  When an Exception occur inside a code of program simply throw it out and ignore that an Exception occur declare the methods as throws Exception.

Method 01 :Exception Handling Using try and catch

1. Find the problem (Hit the exception).
2. Inform that an error has occurred (Throw the exception)
3. Receive the error information (Catch the exception)
4. Take corrective actions (Handle the exception)

Provides two benefits.

- Allows you to fix the error.
- Prevents the program from automatically terminating.

- To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block.
- Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch.
- Syntax:

```
try {
        statement ; // generates an exception

} catch ( Exception-type e ) {
        statement ; // processes the exception
}
```

Internal Working of Java try-catch Block
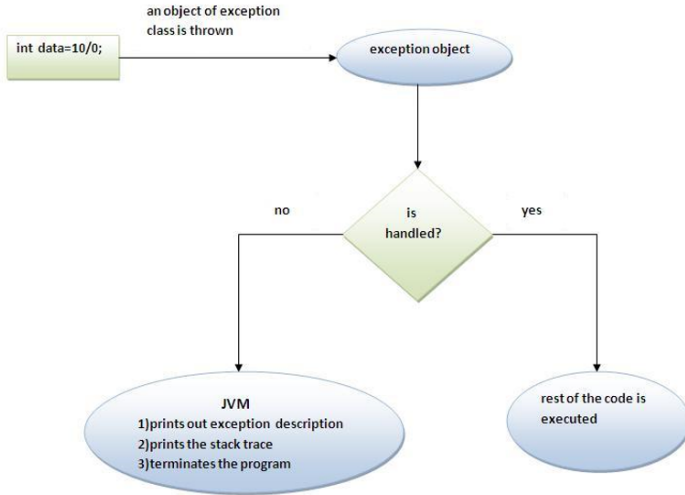
Example: try and catch

```java
class Exc2 {

    public static void main(String args[]) {
        int d, a;
        try { // monitor a block of code.
                d = 0;
                a = 42 / d;
                System.out.println("This will not be printed.");
        } catch (ArithmeticException e) { // catch divide-by-zero error
                System.out.println("Exception: " + e);
                System.out.println("Division by zero.");
        }
    }
}
```

The call to println( ) inside the **try block** is never executed. Once an exception is thrown, program control transfers out of the **try block** into the **catch block**. Put differently, catch is not "called," so execution never "returns" to the try block from a catch. Thus, the line "This will not be printed." is not displayed.

Once the **catch** statement has executed, program control continues with the next line in the program following the entire try / catch mechanism.

Exercise 01:

Write a Java program that prompts the user to enter two integers and then prints their quotient. Use a try–catch block to handle any attempt to divide by zero and display a friendly message.

Hints:

- Use Scanner to read input.
- Catch ArithmeticException to detect division by zero.

Sample answer

```java
import java.util.Scanner;

public class DivisionDemo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter numerator: ");
        int numerator = scanner.nextInt();

        System.out.print("Enter denominator: ");
        int denominator = scanner.nextInt();

        try {
            int result = numerator / denominator;
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero. Please enter a non-zero denominator.");
        }
    }
}
```

Multiple catch clauses

- More than one exception could be raised by a single piece of code.
- To handle this type of situation, programmer 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 others are bypassed, and execution continues after the try /catch block.

```
try {
        statement ; / / generates an exception

} catch ( Exception1-type e ) {
        statement ; / / processes the exception
} catch ( Exception2-type e ) {
        statement ; / / processes the exception
}
```

Example: Multiple catch clauses

```java
class MultipleCatches {

    public static void main(String args[]) {
        try {
            int a = args.length;
            System.out.println("a = " + a);
            int b = 42 / a;
            int c[] = {1};
            c[42] = 99;
        } catch (ArithmeticException e) {
            System.out.println("Divide by 0: " + e);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index oob: " + e);
        }
        System.out.println("After try/catch blocks.");
    }
}
```

Catching More Than One Type of Exception with One Exception Handler

- In Java SE 7 and later, a single catch block can handle more than one type of exception. This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.
- In the catch clause, specify the types of exceptions that block can handle, and separate each exception type with a vertical bar (|).

```java
catch (IOException | SQLException ex) {

        System.out.println(ex);

}
```

FACULTY OF COMPUTING

Nested try Statements

- When a try catch block is present inside another try block then it is called the nested try catch block.
- Each time a try block does not have a catch handler for a particular exception, then the catch blocks of parent try block are inspected for that exception, if match is found that that catch block executes.
- If neither catch block nor parent catch block handles exception then the system generated message would be shown for the exception, similar to what we see when we don't handle exception.

Syntax - Nested try Statements

```
//Main try block
try {
        statement 1;
        statement 2;
        //try-catch block inside another try block
        try {
                statement 3;
                statement 4;
                //try-catch block inside nested try block
                try {
                        statement 5;
                        statement 6;
                } catch(Exception e2) {
                        //Exception Message
                }
        } catch(Exception e1) {
                        //Exception Message
        }
} catch(Exception e3) {  //Catch of Main(parent) try block
        //Exception Message
}
```

Example: Nested try Statements

```java
public class NestingDemo {

    public static void main(String[] args) {
        try { //try-block2
            try { //try-block3
                try {
                    int arr[] = {1, 2, 3, 4};
                    System.out.println(arr[10]);
                } catch (ArithmeticException e) {
                    System.out.print("Arithmetic Exception");
                    System.out.println(" handled in try-block3");
                }
            } catch (ArithmeticException e) {
                System.out.print("Arithmetic Exception");
                System.out.println(" handled in try-block2");
            }
        } catch (ArithmeticException e3) {
            System.out.print("Arithmetic Exception");
            System.out.println(" handled in main try-block");
        } catch (ArrayIndexOutOfBoundsException e4) {
            System.out.print("ArrayIndexOutOfBoundsException");
            System.out.println(" handled in main try-block");
        } catch (Exception e5) {
            System.out.print("Exception");
            System.out.println(" handled in main try-block");
        }
    }
}
```
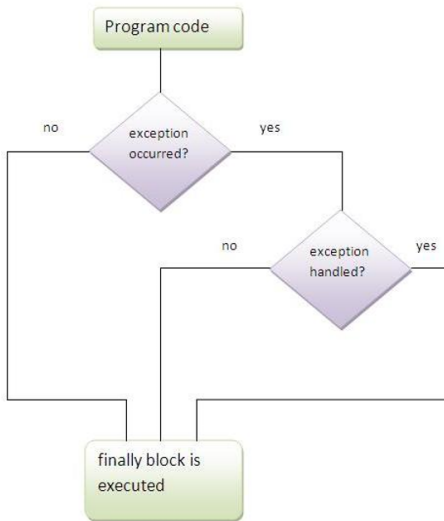
Finally Block

- **finally** statement that can be used to handle an exception that is not caught by any of the previous catch statements.
- **finally** block can be used to handle any exception generated within a try block. It may be added immediately after the try block or after the last catch block shown as follows.
- When a finally block is defined, this is guaranteed to execute, regardless of whether or not an exception is thrown. As a result, we can use it to perform certain house-keeping operations such as closing files and releasing system resources.

Syntax:

```
try {
        statement ;
} finally{
}
```

```
try {
        statement ;
} catch ( Exception1-type e ) {
        statement ;
} finally{
}
```

Internal Working of finally block

Example : Finally block

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

Method 02 :Exception Handling Using throws

- When an Exception occur inside a code of program simply throw it out and ignore.
- throws keyword is used for handling checked exceptions . By using throws we can declare multiple exceptions in one go.
- A throws clause lists the types of exceptions that a method might throw.

Syntax

```
type method-name(parameter-list) throws exception-list
{
        // body of method
}
```

Creating Your Own Exception Subclasses

There may be times when we would like to throw our own exceptions. We can do this by using the keyword **throw** as follows:

<p style="text-align:center;color:red;font-weight:bold;font-size:1.5em;">throw new Throwable-subclass;</p>

To define your own exception create a subclass of Exception (which is a subclass of Throwable).

The Exception class does not define any methods of its own. It inherits those methods provided by Throwable. Thus, all exceptions, including those that you create, have the methods defined by Throwable available to them.

Exception class defines four public constructors. Two are shown here:

- Exception( )
- Exception(String msg)

The first form creates an exception that has no description. The second form lets you specify a description of the exception.

Note:

- Use **throws** keyword to declare that a method may throw exceptions
- Use **throw** keyword to manually throw an exception

Example : Custom Exception Class

```java
class MyException extends Exception {
    MyException (String message) {
        super(message);
    }
}

class TestMyException {
    public static void main (String args[ ]) {
        int x = 5, y = 1000;

        try {
            float z = (float) x / (float) y;
            if(z < 0.01) {
                throw new MyException ("Number is too small") ;
            }
        } catch (MyException e) {
            System.out.println ("Caught my exception");
            System.out.println(e.getMessage ());
        } finally {
            System.out.println ("I am always here") ;
        }
    }
}
```

Commonly Used Exceptions

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.

Thank You!