NCP2103: Object-Oriented Programming (Java Programming)

Exception Handling

Module 11

Errol John M. Antonio
Assistant Professor
Computer Engineering Department
University of the East – Manila Campus

Objectives:

I. Learn about exceptions
II. Try code and catch `Exceptions`
III. Throw and catch multiple `Exceptions`
IV. Use the `finally` block
V. Understand the advantages of exception handling
VI. Specify the `Exceptions` a method can throw
VII. Trace `Exceptions` through the call stack
VIII. Create your own `Exceptions`
IX. Use an assertion

I: Learning About Exceptions

- **Exceptions**
  - Unexpected or error conditions
  - Not usual occurrences
  - Causes
    - Call to file that does not exist
    - Try to write to full disk
    - User enters invalid data
    - Program attempts to divide value by 0

Learning About Exceptions (cont'd.)

- **Exception handling**
  - Object-oriented techniques used to manage `Exception` errors
  - Runtime exceptions
- **Exceptions**
  - `Objects`
  - Descend from `Throwable` class

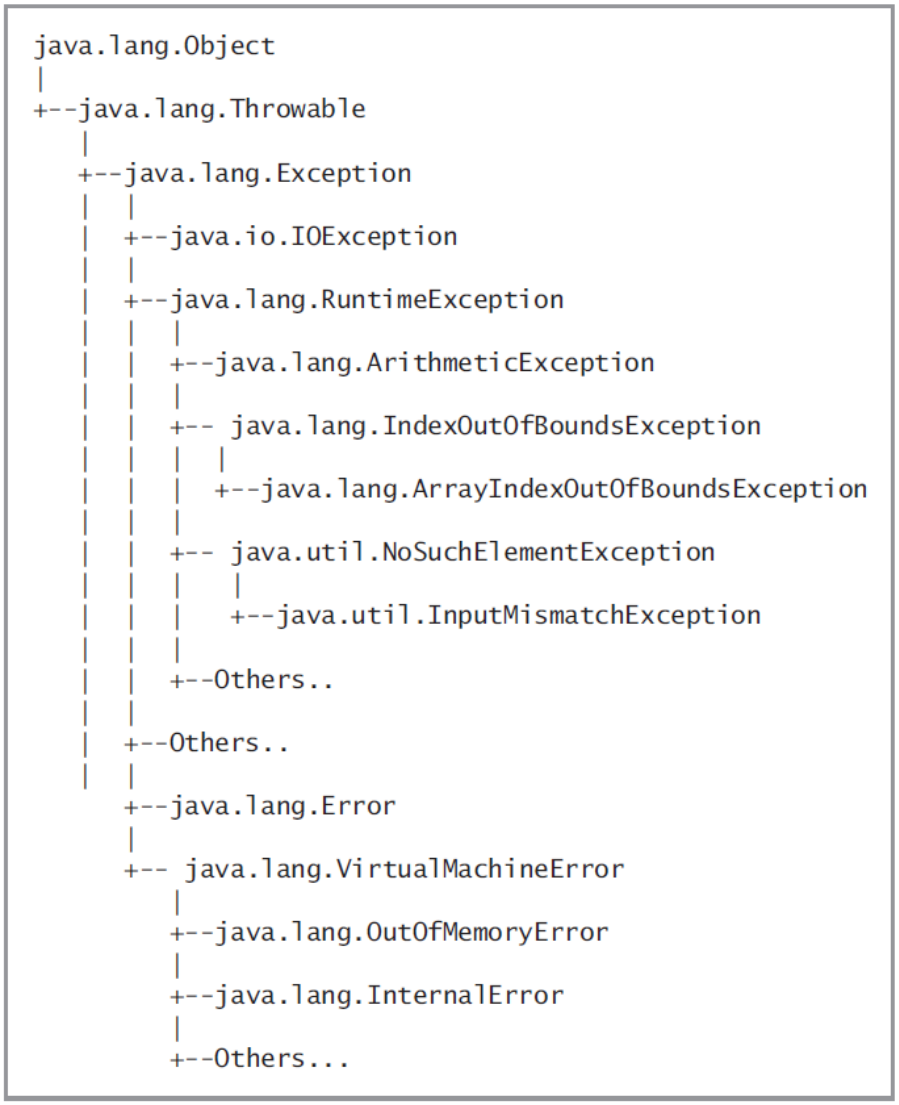

Figure 12-1 The Exception and Error class inheritance hierarchy

Learning About Exceptions (cont'd.)

- **`Error` class**
  - Represents serious errors from which program usually cannot recover
  - `Error` condition
    - Program runs out of memory
    - Program cannot locate required class

Learning About Exceptions (cont'd.)

- **`Exception` class**
  - Less serious errors
  - Unusual conditions
  - Program can recover
- **`Exception` class errors**
  - Invalid array subscript
  - Performing illegal arithmetic operations

Learning About Exceptions (cont'd.)

```
import java.util.Scanner;
public class Division
{
   public static void main(String[] args)
   {
      Scanner input = new Scanner(System.in);
      int numerator, denominator, result;
      System.out.print("Enter numerator >> ");
      numerator = input.nextInt();
      System.out.print("Enter denominator >> ");
      denominator = input.nextInt();
      result = numerator / denominator;
      System.out.println(numerator + " / " + denominator +
         " = " + result);
   }
}
```

Figure 12-2 The Division class

Learning About Exceptions (cont'd.)

Figure 12-3 Two typical executions of the `Division` application

Learning About Exceptions (cont'd.)

- Do not necessarily have to deal with `Exception`
  - Let the offending program terminate
  - Abrupt and unforgiving
- Can write programs without using exception-handling techniques
  - Use a decision to avoid an error
- **Exception handling**
  - Provides a more elegant solution for handling error conditions

Learning About Exceptions (cont'd.)

- Fault-tolerant
  - Designed to continue to operate when some part of system fails
- Robustness
  - Represents degree to which system is resilient to stress

II: Trying Code and Catching `Exceptions`

- **`try`** block
  - Segment of code in which something might go wrong
  - Attempts to execute
    - Acknowledging exception might occur
- **`try`** block includes:
  - Keyword `try`
  - Opening and closing curly brace
  - Executable statements
    - Which might cause exception

Trying Code and Catching `Exceptions` (cont'd.)

- **`catch`** block
  - Segment of code
  - Immediately follows `try` block
  - Handles exception thrown by `try` block preceding it
  - Can "catch"
    - Object of type `Exception`
    - Or `Exception` child class
- **`throw`** statement
  - Sends `Exception` out of method
  - Can be handled elsewhere

Trying Code and Catching `Exceptions` (cont'd.)

- **`catch`** block includes:
  - Keyword `catch`
  - Opening and closing parentheses
    - `Exception` type
    - Name for `Exception` object
  - Opening and closing curly braces
    - Statements to handle error condition

Trying Code and Catching Exceptions (cont'd.)

```
returnType methodName(optional arguments)
{
   // optional statements prior to code that is tried
   try
   {
      // statement or statements that might generate an exception
   }
   catch(Exception someException)
   {
      // actions to take if exception occurs
   }
   // optional statements that occur after try,
   // whether catch block executes or not
}
```

Figure 12-6 Format of try...catch pair

Trying Code and Catching `Exceptions` (cont'd.)

- If no `Exception` occurs within the `try` block
  - `catch` block does not execute
- `getMessage()` method
  - Obtain information about `Exception`
- Within `catch` block
  - Might want to add code to correct the error

Trying Code and Catching Exceptions (cont'd.)

```
import java.util.Scanner;
public class DivisionMistakeCaught
{
   public static void main(String[] args)
   {
      Scanner input = new Scanner(System.in);
      int numerator, denominator, result;
      System.out.print("Enter numerator >> ");
      numerator = input.nextInt();
      System.out.print("Enter denominator >> ");
      denominator = input.nextInt();
      try
      {
         result = numerator / denominator;
         System.out.println(numerator + " / " + denominator +
            " = " + result);
      }
      catch(ArithmeticException mistake)
      {
         System.out.println("Attempt to divide by zero");
      }
   }
}
```

Figure 12-7 The DivisionMistakeCaught application

III: Throwing and Catching Multiple `Exceptions`

- Can place multiple statements within `try` block
  - Only first error-generating statement throws `Exception`
- Catch multiple `Exceptions`
  - Examined in sequence
    - Until match found for `Exception` type
  - Matching `catch` block executes
  - Each remaining `catch` block bypassed

```
import java.util.*;
public class DivisionMistakeCaught3
{
   public static void main(String[] args)
   {
      Scanner input = new Scanner(System.in);
      int numerator, denominator, result;
      try
      {
         System.out.print("Enter numerator >> ");
         numerator = input.nextInt();
         System.out.print("Enter denominator >> ");
         denominator = input.nextInt();
         result = numerator / denominator;
         System.out.println(numerator + " / " + denominator +
            " = " + result);
      }
      catch(ArithmeticException mistake)
      {
         System.out.println(mistake.getMessage());
      }
      catch(InputMismatchException mistake)
      {
         System.out.println("Wrong data type");
      }
   }
}
```

Figure 12-12 The DivisionMistakeCaught3 class

Throwing and Catching Multiple `Exceptions` (cont'd.)

- **"Catch-all" block**
  - Accepts more generic `Exception` argument type:

    ```
    catch(Exception e)
    ```
- **Unreachable code**
  - Program statements that can never execute under any circumstances
- In Java 7, a catch block can also be written to catch multiple exception types
- Poor style for method to throw more than three or four types

```
import java.util.*;
public class DivisionMistakeCaught4
{
   public static void main(String[] args)
   {
      Scanner input = new Scanner(System.in);
      int numerator, denominator, result;
      try
      {
         System.out.print("Enter numerator >> ");
         numerator = input.nextInt();
         System.out.print("Enter denominator >> ");
         denominator = input.nextInt();
         result = numerator / denominator;
         System.out.println(numerator + " / " + denominator +
            " = " + result);
      }
      catch(Exception mistake)
      {
         System.out.println("Operation unsuccessful");
      }
   }
}
```

Figure 12-14 The DivisionMistakeCaught4 application

IV: Using the `finally` Block

- **`finally`** block
  - Use for actions you must perform at end of `try...catch` sequence
  - Use `finally` block to perform cleanup tasks
  - Executes regardless of whether preceding `try` block identifies an `Exception`

Using the `finally` Block (cont'd.)

```
try
{
   // statements to try
}
catch(Exception e)
{
   // actions that occur if exception was thrown
}
finally
{
   // actions that occur whether catch block executed or not
}
```

Figure 12-16 Format of `try...catch...finally` sequence

Using the `finally` Block (cont'd.)

- When **`try`** code fails
  - Throws **`Exception`**
  - **`Exception`** caught
  - **`catch`** block executes
    - Control passes to statements at end of method

Using the `finally` Block (cont'd.)

- Reasons final set of statements might never execute
  - Unplanned `Exception` might occur
  - `try` or `catch` block might contain `System.exit();` statement
- `try` block might throw `Exception` for which you did not provide `catch` block
  - Program execution stops immediately
  - `Exception` sent to operating system for handling
  - Current method abandoned

Using the `finally` Block (cont'd.)

- When `finally` block used
  - `finally` statements execute before method abandoned
- `finally` block executes no matter what outcome of `try` block occurs
  - `try` ends normally
  - `catch` executes
  - `Exception` causes method to abandon prematurely

V: Understanding the Advantages of Exception Handling

- Before object-oriented programming languages
  - Errors handled with confusing, error-prone methods
  - When any method fails
    - Program sets appropriate error code
  - Difficult to follow
    - Application's purpose and intended outcome lost in maze of `if` statements
    - Coding mistakes because of complicated nesting

Understanding the Advantages of Exception Handling (cont'd.)

```
call methodA()
if methodA() worked
{
   call methodB()
   if methodB() worked
   {
      call methodC()
      if methodC() worked
         everything's okay, so display finalResult
      else
         set errorCode to 'C'
   }
   else
      set errorCode to 'B'
}
else
   set errorCode to 'A'
```

Figure 12-18 Pseudocode representing traditional error checking

Understanding the Advantages of Exception Handling (cont'd.)

```
try
{
   call methodA() and maybe throw an exception
   call methodB() and maybe throw an exception
   call methodC() and maybe throw an exception
   everything's okay, so display finalResult
}
catch(methodA()'s error)
{
   set errorCode to "A"
}
catch(methodB()'s error)
{
   set errorCode to "B"
}
catch(methodC()'s error)
{
   set errorCode to "C"
}
```

Figure 12-19 Pseudocode representing object-oriented exception handling

Understanding the Advantages of Exception Handling (cont'd.)

- Java's object-oriented, error-handling technique
  - Statements of program that do "real" work
  - Placed together where logic is easy to follow
  - Unusual, exceptional events
    - Grouped
    - Moved out of the way
- Advantage to object-oriented exception handling
  - Flexibility in handling of error situations

Understanding the Advantages of Exception Handling (cont'd.)

- Appropriately deal with `Exceptions` as you decide how to handle them

VI: Specifying the `Exceptions` a Method Can Throw

- Use keyword `throws` followed by `Exception` type in method header
- `Exception` specification
  - Lists exceptions method may `throw`
- Every Java method has potential to throw an `Exception`
  - For most Java methods, do not use `throws` clause
  - Let Java handle any `Exception` by shutting down program
  - Most exceptions never have to be explicitly thrown or caught

Specifying the `Exceptions` a Method Can Throw (cont'd.)

- Checked exceptions
  - Programmers should anticipate
  - Programs should be able to recover
- Unchecked exceptions
  - Inherit from the `Error` class or the `RuntimeException` class
  - You are not required to handle these exceptions
    - You can simply let the program terminate
    - Dividing by zero

Specifying the `Exceptions` a Method Can Throw (cont'd.)

- Throw checked exception
  - Catch it
  - Or declare exception in method header's `throws` clause
- Must know to use method to full potential
  - Method's name
  - Method's return type
  - Type and number of arguments method requires
  - Type and number of `Exceptions` method throws

VII: Tracing `Exceptions` Through the Call Stack

- **Call stack**
  - Memory location where computer stores list of method locations to which system must return
- When method throws `Exception`
  - `Exception` thrown to next method up call stack
  - Allows methods to handle `Exceptions` wherever programmer has decided it is most appropriate
    - Including allowing operating system to handle error

Tracing `Exceptions` Through the Call Stack (cont'd.)

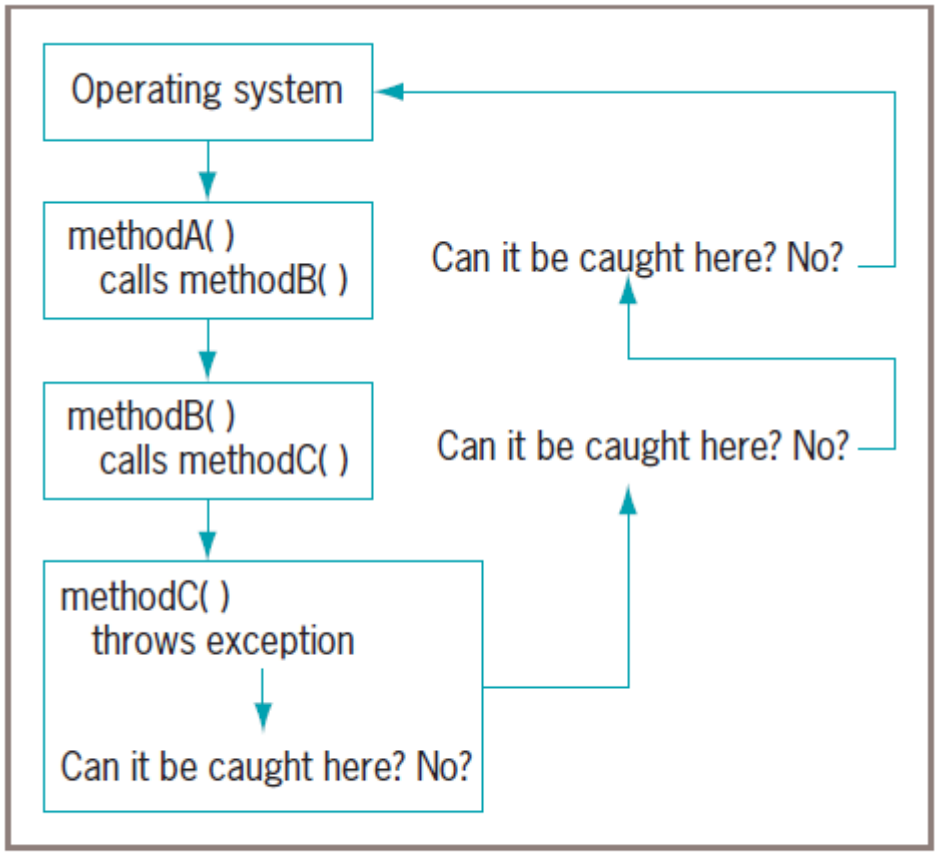

Figure 12-24 Cycling through the call stack

Tracing `Exceptions` Through the Call Stack (cont'd.)

- **`printStackTrace()`** method
  - Display list of methods in call stack
  - Determine location of `Exception`
  - Do not place in finished program
    - Most useful for diagnosing problems

VIII: Creating Your Own `Exceptions`

- Java provides over 40 categories of `Exceptions`
- Java allows you to create your own `Exceptions`
  - Extend a subclass of `Throwable`
- `Exception` class constructors

```
Exception()
Exception(String message)
Exception(String message,
          Throwable cause)
Exception(Throwable cause)
```

IX: Using Assertions

- Assertion
  - Java language feature
  - Detect logic errors
  - Debug programs
- `assert` statement
  - Create assertion

    ```
    assert booleanExpression :
    optionalErrorMessage
    ```
  - Boolean expression should always be `true` if program working correctly

Using Assertions (cont'd.)

- **`AssertionError`** thrown
  - When condition `false`
- Enable assertion
  - Execute program using `-ea` option

You Do It

- Throwing and catching `Exceptions`
- Creating a class that automatically throws `Exceptions`
- Creating a class that passes on an `Exception`
- Creating an application that can catch `Exceptions`
- Extending a class that throws `Exceptions`
- Creating an `Exception` class
- Using an `Exception` you created

Don't Do It

- Don't forget that all the statements in a `try` block might not execute
- Don't forget to place more specific `catch` blocks before more general ones
- Don't forget to write a `throws` clause for a method that throws an exception but does not handle it
- Don't forget to handle any checked exception thrown to your method

Summary

- Exception
  - Unexpected or error condition
- Exception handling
  - Object-oriented techniques to manage errors
- Basic classes of errors `Error` and `Exception`
- Exception–handling code
  - `try` block
  - `catch` block
  - `finally` block

Summary (cont'd.)

- Use clause `throws <name>Exception` after method header
  - Indicate type of `Exception` that might be thrown
- Call stack
  - List of method locations where system must return
- Java provides over 40 categories of `Exceptions`
  - Create your own `Exceptions`
- Assertion
  - State condition that should be true
  - Java throws `AssertionError` when it is not

End of Module.

REFERENCE:

Farrell, J. (2016). ***Java Programming***. 8th Edition. Course Technology, Cengage Learning.