Exception Handling (Abbreviated presentation)
13.1 Introduction

• Exception – an indication of a problem that occurs during a program’s execution

• Exception handling – resolving exceptions that may occur so program can continue or terminate gracefully

• Exception handling enables programmers to create programs that are more robust and fault-tolerant
13.1 Introduction

• Examples
  – `ArrayIndexOutOfBoundsException` – an attempt is made to access an element past the end of an array
  – `ClassCastException` – an attempt is made to cast an object that does not have an *is-a* relationship with the type specified in the cast operator
  – `NullPointerException` – when a null reference is used where an object is expected
13.2 Exception-Handling Overview

- Intermixing program logic with error-handling logic can make programs difficult to read, modify, maintain and debug
- Exception handling enables programmers to remove error-handling code from the “main line” of the program’s execution
- Improves clarity
- Enhances modifiability
13.3 Example: Divide By Zero Without Exception Handling

• Thrown exception – an exception that has occurred
• Stack trace
  – Name of the exception in a descriptive message that indicates the problem
  – Complete method-call stack
• ArithmeticException – can arise from a number of different problems in arithmetic
• Throw point – initial point at which the exception occurs, top row of call chain
• InputMismatchException – occurs when Scanner method `nextInt` receives a string that does not represent a valid integer
// Fig. 13.1: DivideByZeroNoExceptionHandling.java
// An application that attempts to divide by zero.
import java.util.Scanner;

public class DivideByZeroNoExceptionHandling {
    // demonstrates throwing an exception when a divide-by-zero occurs
    public static int quotient( int numerator, int denominator ) {
        return numerator / denominator; // possible division by zero
    } // end method quotient

    public static void main( String args[] ) {
        Scanner scanner = new Scanner( System.in ); // scanner for input

        System.out.print( "Please enter an integer numerator: " );
        int numerator = scanner.nextInt();
        System.out.print( "Please enter an integer denominator: " );
        int denominator = scanner.nextInt();

        int result = quotient( numerator, denominator );
        System.out.printf("\nResult: %d / %d = %d\n", numerator, denominator, result );
    } // end main
} // end class DivideByZeroNoExceptionHandling

Please enter an integer numerator: 100
Please enter an integer denominator: 7
Result: 100 / 7 = 14
Please enter an integer numerator: 100
Please enter an integer denominator: 7
Result: 100 / 7 = 14

Please enter an integer numerator: 100
Please enter an integer denominator: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at DivideByZeroNoExceptionHandling.quotient(DivideByZeroNoExceptionHandling.java:10)
    at DivideByZeroNoExceptionHandling.main(DivideByZeroNoExceptionHandling.java:22)

Please enter an integer numerator: 100
Please enter an integer denominator: hello
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Unknown Source)
    at java.util.Scanner.next(Unknown Source)
    at java.util.Scanner.nextInt(Unknown Source)
    at java.util.Scanner.nextInt(Unknown Source)
    at DivideByZeroNoExceptionHandling.main(DivideByZeroNoExceptionHandling.java:20)
13.4 Example: Handling ArithmeticExceptions and InputMismatchExceptions

• With exception handling, the program catches and handles (i.e., deals with) the exception.

• Next example allows user to try again if invalid input is entered (zero for denominator, or non-integer input).
Enclosing Code in a `try` Block

- `try` block – encloses code that might throw an exception and the code that should not execute if an exception occurs
- Consists of keyword `try` followed by a block of code enclosed in curly braces
Software Engineering Observation 13.1

Exceptions may surface through explicitly mentioned code in a try block, through calls to other methods, through deeply nested method calls initiated by code in a try block or from the Java Virtual Machine as it executes Java bytecodes.
Catching Exceptions

• catch block – catches (i.e., receives) and handles an exception, contains:
  – Begins with keyword catch
  – Exception parameter in parentheses – exception parameter identifies the exception type and enables catch block to interact with caught exception object
  – Block of code in curly braces that executes when exception of proper type occurs

• Matching catch block – the type of the exception parameter matches the thrown exception type exactly or is a superclass of it

• Uncaught exception – an exception that occurs for which there are no matching catch blocks
  – Cause program to terminate if program has only one thread; Otherwise only current thread is terminated and there may be adverse effects to the rest of the program
Common Programming Errors

• It is a syntax error to place code between a `try` block and its corresponding `catch` blocks.

• Each `catch` block can have only a single parameter—specifying a comma-separated list of exception parameters is a syntax error.

• It is a compilation error to catch the same type in two different `catch` blocks in a single `try` statement.
Termination Model of Exception Handling

• When an exception occurs:
  – try block terminates immediately
  – Program control transfers to first matching catch block

• After exception is handled:
  – *Termination model* of exception handling – program control does not return to the throw point because the try block has expired; Flow of control proceeds to the first statement after the last catch block
  – Alternately (not JAVA), a *resumption model* of exception handling – program control resumes just after throw point

• try statement – consists of try block and corresponding catch and/or finally blocks
Defining your own exceptions for user-defined methods:
Using the *throws* Clause

- *throws* clause – specifies the exceptions a method may throw
  - Appears after method’s parameter list and before the method’s body
  - Contains a comma-separated list of exceptions
  - Exceptions can be thrown by statements in method’s body of by methods called in method’s body
  - Exceptions can be of types listed in *throws* clause or subclasses
// Fig. 13.2: DivideByZeroWithExceptionHandling.java
// An exception-handling example that checks for divide-by-zero.
import java.util.InputMismatchException;
import java.util.Scanner;

public class DivideByZeroWithExceptionHandling {
    public static int quotient( int numerator, int denominator )
        throws ArithmeticException {
        return numerator / denominator; // possible division by zero
    } // end method quotient

    public static void main( String args[] ) {
        Scanner scanner = new Scanner( System.in ); // scanner for input
        boolean continueLoop = true; // determines if more input is needed

        do {
            try // read two numbers and calculate quotient
                System.out.print( "Please enter an integer numerator: " );
                int numerator = scanner.nextInt();
                System.out.print( "Please enter an integer denominator: " );
                int denominator = scanner.nextInt();
            } catch (InputMismatchException e) {
                System.out.println( "Invalid input, please enter valid integers." );
            } // end catch
        } while (continueLoop);
    } // end main
} // end class DivideByZeroWithExceptionHandling
```
int result = quotient( numerator, denominator );
System.out.printf( "Result: %d / %d = %d\n", numerator,
    denominator, result );
continueLoop = false; // input successful; end looping
} // end try
catch ( InputMismatchException inputMismatchException )
{
    System.err.printf( "Exception: %s\n", inputMismatchException );
    scanner.nextLine(); // discard input so user can try
    System.out.println(
        "You must enter integers. Please try again.\n" );
} // end catch
catch ( ArithmeticException arithmeticException )
{
    System.err.printf( "Exception: %s\n", arithmeticException );
    System.out.println(
        "Zero is an invalid denominator. Please try again.\n" );
} // end catch
} while ( continueLoop ); // end do...while
} // end class DivideByZeroWithExceptionHandling
```

Call method `quotient`, which may throw `ArithmeticException`.

If we have reached this point, input was valid and `denominator` was non-zero, so looping can stop.

Catching `InputMismatchException` (user has entered non-integer input)

Exception parameters

Read invalid input but do nothing with it

Catching `ArithmeticException` (user has entered zero for denominator)

If line 32 was never successfully reached, loop continues and user can try again.
Please enter an integer numerator: 100
Please enter an integer denominator: 7
Result: 100 / 7 = 14

Please enter an integer numerator: 100
Please enter an integer denominator: 0
Exception: java.lang.ArithmeticException: / by zero
Zero is an invalid denominator. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7
Result: 100 / 7 = 14

Please enter an integer numerator: 100
Please enter an integer denominator: hello
Exception: java.util.InputMismatchException
You must enter integers. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7
Result: 100 / 7 = 14
13.5 When to Use Exception Handling

- Exception handling designed to process synchronous errors
- Synchronous errors – occur when a statement executes
- Asynchronous errors – occur in parallel with and independent of the program’s flow of control
Software Engineering Observation 13.2

Incorporate your exception-handling strategy into your system from the design process’s inception. Including effective exception handling after a system has been implemented can be difficult.
Avoid using exception handling as an alternate form of flow of control. These “additional” exceptions can “get in the way” of genuine error-type exceptions.
13.6 Java Exception Hierarchy

• All exceptions inherit either directly or indirectly from class Exception
• Exception classes form an inheritance hierarchy that can be extended
• Class Throwable, superclass of Exception
  – Only Throwable objects can be used with the exception-handling mechanism
  – Has two subclasses: Exception and Error
    • Class Exception and its subclasses represent exception situations that can occur in a Java program and that can be caught by the application
    • Class Error and its subclasses represent abnormal situations that could happen in the JVM – it is usually not possible for a program to recover from Errors
Fig. 13.3 | Portion of class Throwable’s inheritance hierarchy.
13.6 Java Exception Hierarchy

- Two categories of exceptions: checked and unchecked
- Checked exceptions
  - Exceptions that inherit from class `Exception` but not from `RuntimeException`
  - Compiler enforces a catch-or-declare requirement
  - Compiler checks each method call and method declaration to determine whether the method `throws` checked exceptions. If so, the compiler ensures that the checked exception is caught or is declared in a `throws` clause. If not caught or declared, compiler error occurs.
- Unchecked exceptions
  - Inherit from class `RuntimeException` or class `Error`
  - Compiler does not check code to see if exception is caught or declared
  - If an unchecked exception occurs and is not caught, the program terminates or runs with unexpected results
  - Can typically be prevented by proper coding
13.7 finally block

• Programs that obtain certain resources must return them explicitly to avoid resource leaks

• finally block
  – Consists of finally keyword followed by a block of code enclosed in curly braces
  – Optional in a try statement
  – If present, is placed after the last catch block
  – Executes whether or not an exception is thrown in the corresponding try block or any of its corresponding catch blocks
  – Will not execute if the application exits early from a try block via method System.exit
  – Typically contains resource-release code
Error-Prevention Tip 13.7

A subtle issue is that Java does not entirely eliminate memory leaks. Java will not garbage collect an object until there are no more references to it. Thus, memory leaks can occur, if programmers erroneously keep references to unwanted objects.
Fig. 13.4 | Position of the finally block after the last catch block in a try statement.
13.7 finally block

• If no exception occurs, catch blocks are skipped and control proceeds to finally block.
• After the finally block executes control proceeds to first statement after the finally block.
• If exception occurs in the try block, program skips rest of the try block. First matching the catch block executes and control proceeds to the finally block. If exception occurs and there are no matching catch blocks, control proceeds to the finally block. After the finally block executes, the program passes the exception to the next outer the try block.
• If catch block throws an exception, the finally block still executes.
Throwing Exceptions Using the `throw` Statement

- `throw` statement – used to throw exceptions
- Programmers can throw exceptions themselves from a method if something has gone wrong
- `throw` statement consists of keyword `throw` followed by the exception object
13.9 printStackTrace, getStackTrace and getMessage

• Methods in class Throwable retrieve more information about an exception
  – printStackTrace – outputs stack trace to standard error stream
  – getStackTrace – retrieves stack trace information as an array of StackTraceElement objects; enables custom processing of the exception information
  – getMessage – returns the descriptive string stored in an exception
An exception that is not caught in an application causes Java’s default exception handler to run. This displays the name of the exception, a descriptive message that indicates the problem that occurred and a complete execution stack trace.
13.9 printStackTrace, getStackTrace and getMessage

- StackTraceElement methods
  - getClassName
  - getFileName
  - getLineNumber
  - getMethodName

- Stack trace information follows pattern –
  className.methodName(fileName:lineNumber)
13.11 Declaring New Exception Types

• You can declare your own exception classes that are specific to the problems that can occur when another program uses your reusable classes.

• New exception class must extend an existing exception class.

• Typically contains only two constructors:
  – One takes no arguments, passes a default exception messages to the superclass constructor.
  – One that receives a customized exception message as a string and passes it to the superclass constructor.
13.13 Assertions

• Assertions are conditions that should be true at a particular point in a method
• Help ensure a program’s validity by catching potential bugs
• Preconditions and Postconditions are two kinds of assertions
• Assertions can be stated as comments or assertions can be validated programmatically using the assert statement
13.13 Assertions

• **assert statement**
  – Evaluates a boolean expression and determines whether it is true or false
  – Two forms
    • `assert expression;` -- `AssertionError` is thrown if `expression` is false
    • `assert expression1 : expression2;` -- `AssertionError` is thrown if `expression1` is false, `expression2` is error message
  – Used to verify intermediate states to ensure code is working correctly
  – Used to implement preconditions and postconditions programmatically

• By default, assertions are disabled

• Assertions can be enabled with the `-ea` command-line option
```java
// Fig. 13.9: AssertTest.java
// Demonstrates the assert statement
import java.util.Scanner;

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

        System.out.print( "Enter a number between 0 and 10: " );
        int number = input.nextInt();

        // assert that the absolute value is >= 0
        assert ( number >= 0 && number <= 10 ) : "bad number: " + number;

        System.out.printf( "You entered %d
", number );
    } // end main
} // end class AssertTest
```

Enter a number between 0 and 10: 5
You entered 5

Enter a number between 0 and 10: 50
Exception in thread "main" java.lang.AssertionError: bad number: 50
    at AssertTest.main(AssertTest.java:15)