Java Chapter 12 – Exception Handling

**EXCEPTIONS:**

* An unexpected error or condition
* Runtime Exception: occur during program execution
* Syntax Errors: discovered during program compilation

**2 Basic Classes of Errors:**

* **Error:**
* More serious errors from which the program usually can’t recover
* Programs cannot recover from error conditions on its own
* **Exception:**
* Less serious errors that represent unusual conditions that arise while a program is running, & from which a program can recover
* Exception message displays when the program code could have prevented an error

Ex: Division class – p. 595

* User tries to divide by 0 🡪 Java does not allow integer division by 0 🡪 crashes program

🡪 Results in an ArithmeticException – 1 of many subclasses of Exception

Error returns:

* Type of exception – ArithmeticException
* Info about the error – ( “/ by 0” )
* The method that generated the error
* The file & line number for the error
* User enters non integer values (string, float) 🡪 InputMismatchException
* Stack Trace: list of error messages after each attempted execution

🡪shows each method that was called as the program ran

* Don’t necessarily have to deal w exception 🡪 can let program terminate prematurely

🡪 bad for mission critical tasks

* Most common error handling solution 🡪 using a decision to avoid an error
* Programs that can handle exceptions appropriately 🡪 more fault tolerant & robust

**TRYING CODE & CATCHING EXCEPTIONS:**

\*In Object-Oriented terminology, you “try” a procedure that might cause an error. A method that detects an error condition “throws an exception”, and if you write a block of code that processes the error, that block is said to “catch the exception”

* **Try Block:**
* Block of code you “try” / attempt to execute, in which an exception might occur
* Syntax:
* Try { // executable statements that might cause exceptions }

\*To handle a thrown exception 🡪 code 1+ catch blocks immediately after the try block

* **Catch Block: ‘Catch Clause’**
* Block of code that can handle exceptions thrown by the try block that precedes it
* Exception might be thrown automatically or you may write a throw statement explicitly
* Each catch block can “catch” one type of exception
* That is, one object that is an object of type Exception or one of its child classes
* Looks a lot like a method named catch() that takes an arg of type Exception
* Syntax:
* Catch (exception someException) { //statements used to handle the error condition }
* someException represents an object of the Exception class or any of its subclasses
* \*If an exception occurs during the try, the exception is thrown & the catch executes
* \*If no exception occurs – catch is bypassed
* **getMessage()** Method:
* Inherited from the Throwable class that retrieves Java’s built-in message about an exception
* Alternative to writing your own message in a catch block
* Should also add code to remedy / correct the error
* Using Try blocks to make programs “Foolproof”:

\*One of most common uses for a try block is to circumvent user data entry

* Try blocks can allow you to handle potential data conversion exceptions caused by careless users

**\*\*Recall:** Add an extra nextLine() call after any next(), nextInt(), or nextDouble() to absorb the Enter key remaining in the input buffer before subsequent nextLine() calls\*\*

* When you attempt to convert numeric data in a try block and the effort is followed by another attempted conversion, you must also remember to account for the potential remaining characters left in the input buffer
* Add a commented out //input.nextLine(); after the catch but before the method ends so it’s there if needed

Declaring & Initializing variables in try.. catch Blocks:

\*If you declare a variable before a try block but wait to assign its initial usable value w/in the try.. catch block – must be careful that the variable receives a useful value 🡪 otherwise when you use the variable after the try.. catch pair ends, the program will not compile

**import java.util.Scanner;**

**public class UninitializedVariableTest**

{

public static void main(String[] args)

{

int x;

Scanner input = new Scanner(System.in);

try {

System.out.print(“Enter an integer: ”);

x = input.nextInt();

}

catch(Exception e) {

System.out.println(“Exception occurred”);

}

System.out.println(“x is ” + x);

}

}

* Because the user might not enter an integer, the conversion to an integer might fail & an exception might be thrown
* 3 options to fix this error:
* Assign a value to x before the try block. That way, even if an exception is thrown, x will still have a useable value to display in the last statement
* Assign a useable value to x w/in the catch block. That way, if an exception is thrown, x will hold a usable value
* Move the output statement w/in the try block. If the conversion of the user’s entry to an integer is successful, the try block finishes and x is displayed. However, if the conversion fails, the try block is abandoned, the catch block executes, and x is not used

**THROWING & CATCHING MULTIPLE EXCEPTIONS:**

* If you try more than one statement, only the first error-generating statement throws an exception
* Exception occurs 🡪 logic moves to catch block 🡪 rest of try statements are unexecuted
* Programs containing multiple catch blocks are examined in sequence until a match is found for the type of exception that occurred
* Matching catch block executes 🡪 each remaining catch block is bypassed
* \*When listing multiple catch blocks, be careful that some catch blocks don’t become unreachable
* Arrange catch blocks so that the ‘bigger’ basket is always below the smaller one
* AKA more specific at the top, more general at the bottom
* \*Sometimes you want to execute the same code no matter which Exception type occurs

Write a generic catch block that can catch any type of Exception object

catch (Exception mistake) {

System.out.println(“Operation Unsuccessful”);

}

* The generic catch block can act as a ‘catch-all’ block
* When any error subclass of Exception occurs (arithmetic error, incorrect data input type error) the thrown exception is “promoted” to an Exception error in the catch block
* \*Java 7 & 8 allow a catch block to be written to catch specific, multiple exception types

catch (ArithmeticException, InputMismatchException e) { }

* When either exception is caught, its local identifier is e
* Poor style for a method to throw/catch more than 3 or 4 Exception types:
* The method may be trying to accomplish too many diverse tasks & should be broken up into smaller methods
* The Exception types thrown are too specific & should be generalized

**THE FINALLY BLOCK:**

* Used when you have actions you must perform at the end of a try.. catch sequence
* This code executes regardless of whether the preceding try block identifies an exception

(executes before control returns to the Operating System)

* Usually used to perform cleanup tasks that must happen regardless of whether exceptions occurred and whether any exceptions that occurred were caught
* Different than just a normal set of statements placed after the try & catch blocks; these may never execute for a couple reasons:
* Any try block might throw an Exception object for which you did not provide a catch block. In this case of an unhandled exception, program execution stops immediately, the exception is sent to the OS for handling, & the current method is abandoned
* The try or catch block might contain a System.exit(); statement which stops execution immediately
* Using a finally block ensures that the finally statements will execute before the method is abandoned, even if it concludes prematurely
* \*Programmers often use this when a program uses data files that must be closed

**\*Recap:**

* Try all statements that might throw exceptions, include all the needed catch blocks, & an optional finally block

try { //statements to try that might generate an exception }

catch (Exception e) { //actions to take if exception(s) was thrown }

finally { //actions that occur whether a catch block executed or not }

**ADVATAGES OF EXCEPION HANDLING:**

\*Before the inception of Object-Oriented programming languages, potential program errors were handled using somewhat confusing, error-prone methods

* Object-Oriented Exception Handling Provides:
* clarity & flexibility
* ability to appropriately deal w/ exceptions as you decide how to handle them

\*Methods are flexible partly because they’re reusable – well written methods might be used by many applications\*

* each calling application might need to handle a thrown error differently, depending on its purpose
* ex: method that contains a division statement can throw the error, but each calling program can assume responsibility for handling the error detected by the method in an appropriate way

**SPECIFYING EXCEPTIONS THAT A METHOD CAN THROW:**

\*If a method throws an exception that it will not catch but that it will be caught by a different method🡪 throw clause

* Creating Throw Clause:
* Using the keyword throws followed by an Exception type in the method header
* This is known as Exception Specification

Ex:

**public class PriceList**

{

private static final double[] price = { 15.99, 27.88, 34.56, 45.89 };

public static void displayPrice(int item) throws IndexOutOfBoundsException

{

System.out.println(“The price is $” + price[item]);

}

}

* The displayPrice() method accepts a parameter to use as the array subscript, but because the subscript could be out of bounds, the method contains a throws clause, acknowledging it could throw an exception
* Other programmers writing other applications using the PriceList class can choose to handle this potential exception differently, but they can all use the flexible displayPrice() method because it doesn’t limit the calling methods choice of recourse

**\*For most Java methods you write, you do not use a throws clause**

* Most exceptions never have to be explicitly thrown or caught, nor do you have to include a throws clause in the headers of methods that automatically throw these exceptions
* The only exceptions that must be caught or named in a throws clause are Checked Exceptions
* **Checked Exceptions:**
* The type of exceptions that programmers should anticipate & from which programs should be able to recover
* Include all exceptions that you explicitly throw & that descend from the Exception class
* Checked exceptions are subject to the **Catch or Specify Requirement:**

🡪If you throw a checked exception from a method, you must do one of the following:

-Catch it w/in the method

-Specify the exception in your method header’s throw clause

\*Code that uses a checked exception will not compile if this requirement isn’t followed

\*If you write a method w/ a throws clause in the header, then any method that uses your method must do one of the following:

-Catch & handle the possible exception

-Declare the exception in its throw clause – the called method can then rethrow the

exception to yet another method that might either catch it or yet throw it again

\*So, when an exception is a checked type, client programmers are forced to deal

w/ the possibility that an exception will be thrown

* If you write a method that explicitly throws a checked exception that is not caught win the method, Java requires you to use the throws clause in the header of the method
* Using a throws clause does not mean that the method will throw an exception –

🡪 means it might throw an exception

🡪 using throws notifies applicants using the method of the potential for an exception

\*A class containing multiple methods whose signatures differ only in their return type 🡪 not overloaded

\*Methods w/ signatures that only differ in their throws clauses 🡪 compiler considers them identical

\*Throws clause isn’t part of the methods signature – rather its part of the method’s interface

* To be able to use a method to its full potential – must know methods name & 3 additional infos:
* Methods return type
* Type & number of arguments the method requires
* Type & number of exceptions the method throws

\*Cant call a method w/out knowing types of args required BUT

\*Can call a method w/out knowing its return type – if you don’t want to use the value it returns

* When a method might throw 1+ exception type, you can specify a list of potential exceptions in the method header by separating them w commas
* Alternative 🡪 if all exceptions descend from the same parent – can specify the more general parent class

Ex: if method might throw an ArithmeticException or IndexOutOfBoundsException

* Can specify that the method throws a RuntimeException
* Extreme Alternative 🡪 simply specify that the method throws a general Exception object

**TRACING EXCEPTIONS THROUGH CALL STACK:**

\*When one method calls another, the computer’s OS must keep track of where the method call came from, and program control must return to the calling method after the called method is completed

* **Call Stack:**
* The memory location where the computer stores a list of memory locations to which the system must return when methods end
* When a method throws an exception & the method does not catch it, the exception is thrown to the next method up the call stack – AKA to the method that called the offending method

Ex:

* methodA() 🡪 calls methodB() 🡪 calls methodC() 🡪 throws an exception 🡪 Java looks for catch block in C 🡪 then B 🡪 then A 🡪 thrown to JVM/OS which displays message at cmd prompt
* **printStackTrace() method:**
* displays a list of methods in the call stack so you can determine the location of the statement that caused the exception
* while developing an application, this method is useful for diagnosing your class’s problems, BUT – do not want to place this method in a finished program

**CREATING YOUR OWN EXCEPTION CLASSES:**

\*Can handle potential error situations w/ IF statements, but Java also allows you to create your own Exception classes

\*Recall: Throwable has 2 subclasses: Exception (recoverable errors) & Error (nonrecoverable errors)

* To create your own throwable Exception class, must extend a subclass of Throwable
* Always want to create exceptions for recoverable errors 🡪 extend Exception
* Can extend any existing Exception subclass, but usually inherit directly from Exception
* Conventional to end each Exception subclass with ‘Exception’
* The Exception class contains 4 constructors:
* Exception() – constructs a new Exception object w/ null as its detail message
* Exception(String message) – constructs a new Exception object w/ the specified detail message
* Exception(String message, Throwable cause) – constructs a new Exception object w/ the specified detail message and cause
* Exception(Throwable cause) - constructs a new Exception object w/ the specified cause and a detail message of cause.toString(), which typically contains the class & the detail message of cause, or null if the cause argument is null

**public class HighBalanceException extends Exception**

{

public HighBalanceException()

{

super(“Customer balance is high”);

}

}

* Constructor passes a description of the error to the parent Exception constructor
* The string would be retrieved if you called the getMessage() method w/ a HighBalanceException object

**public class CustomerAccount**

{

Private in accNum;

Private double balance;

Public static double HIGH\_CREDIT\_LIMIT = 20000.00;

Public CustomerAccount(int num, double bal) throws HighBallanceException

{

accNum = num;

balance = bal;

if(balance > HIGH\_CREDIT\_LIMIT)

throw(new HighBalanceException());

}

}

* This classes constructor header indicates that it might throw a HighBalanceException
* If the balance used as an arg to the constructor exceeds a set limit, a new, unnamed instance of the HighBalanceException class is thrown

**import javax.swing.\*;**

**public class UseCustomerAccount**

{

public static void main(String[] args)

{

int num;

double balance;

String input;

input = JOptionPane.showInputDialog(null, “Enter account number”);

num = Integer.parseInt(input);

input = JOptionPane.showInputDialog(null, “Enter balance due”);

balance = Double.parseDouble(input);

try {

**CustomerAccount ca = new CustomerAccount(num, balance);**

JOptionPane.showMessageDialog(null, “Customer #” + num +

“ has a balance of $” + balance);

}

**catch(HighBalanceException hbe) {**

**JOptionPane.showMessageDialog(null, “Customer #” + num +**

**“ has a balance of $” + balance + “ which is higher than the credit limit”);**

**}**

}

}

* User is prompted for an account number & balance
* Try block attempts to construct a CustomerAccount
* If CustomerAccount constructor doesn’t throw an exception – info displayed
* If it does throw an exception – catch block displays message
* Different apps can take different actions:

-display return of getMessage(), construct a CustomerAccount object w/ lower balance

* Instead of hard coding error messages into your exception classes – create a catalog of possible messages to use
* All messages are stored in one location
* The list
* Other applications
* If your application

**USING ASSERTIONS:**

\*Syntax errors 🡪 mistakes using Java language; compile-time errors that prevent a program from compiling & creating an executable file with a .class extension

\*Some logic errors cause runtime errors that cause a program to terminate

* Some logic errors do not cause a program to terminate but still produce incorrect results/ output
* **Assertion:** a feature that can help you detect such logic errors & debug a program
* **Assert Statement:**
* Used to create an assertion
* States a condition that should be true, & Java throws an assertion when it is not
* Syntax:

Assert booleanExpression : optionalErrorMessage

* The bool expression should always be true if the program is working correctly
* The optionalErrorMessage is displayed if the bool expression is false

import java.util.Scanner;

**public class EvenOdd**

{

public static void main(String[] args)

{

Scanner input = new Scanner(System.in);

int number;

System.out.print(“Enter a number:” );

number = input.nextInt();

if(isEven(number))

System.out.println(number + “ is even”);

else

System.out.println(number + “ is even”);

}

public static boolean isEven(int number)

{

boolean result;

if(number % 2 == 1)

result = false;

else

result = true;

return result;

}

}

* This application prompts user for a number & passes it to a method that determines whether value is even or not
* W/in **isEven()** – the remainder is taken when the passed parameter is divided by 2

-If remainder is 1, result is set to false

-If remainder is not 1, result is set to true (even numbers all have a 0 remainder when % 2 is applied to them)

* New version of the isEven() method:

**public static boolean isEven(int number)**

{

boolean result;

if(number % 2 == 1)

result = false;

else {

result = true;

assert number % 2 == 0 : number + “ % 1 is ” + number % 2;

}

return result;

}

* The assert statement asserts that when the remainder of a number is divided by 2 is not 1, it must be 0
* If the expression is not true, a message is created using the values of number & its remainder after dividing by 2

**\*\*\*To enable the assertion** 🡪 **must use ‘-ea’ option (cmd prmt) when you execute the program\*\*\***

* User enters -5 🡪 program displays messages instead of incorrect output
* Seeing that -5 % 2 resulted in -1🡪 AssertionError thrown 🡪 remainder operator results in a negative value when one of its operands is negative, making this program incorrect
* Seeing that -5 % 2 resulted in -1 🡪 must return to source code & change the logic 🡪 options:
* Could change the IF statement to test for even values by comparing number % 2 to 0 first:

If(number % 2 == 0)

Result = true;

Else

Result = false; //Then values of both 1 & -1 would be classified as not even

\*\*Any display statements should be removed in the final product, BUT any assert statements can be left in place – if the user doesn’t use the -ea option when running the program 🡪 won’t see existence of any assert statements\*\*

\*\*\*You do not want to use assertions to check for every type of error that could occur in a program. For example, if you want to ensure that a user enters numeric data you should use exception-handling techniques that provide the means for your program to recover from the mistake. If you want to ensure that the data falls w/in a specific range, you should use a decision or a loop. Assertions are meant to be helpful in the development stage of a program, not when it is in production and in the hands of users\*\* \*

DISPLAYING THE VIRTUAL KEYBOARD:

\*You can write many functional Java programs w/out using exception handling techniques 🡪 However, you will sometimes have to employ exception-handling techniques if you want to use methods written by others that throw exceptions

To Display the Virtual Keyboard of the Windows System:

* Must accommodate a Thrown Exception

**import java.util.Scanner;**

**import java.io.IOException;**

**public class VirtualKeyboardDemo**

{

public static void main(String [] args)

{

Scanner input = new Scanner(System.in);

try {

process proc = Runtime.getRtuntime().exec

(“cmd /c C:\\Windows\\System32\\osk.exe”);

}

catch(IOException e) {

System.out.println(e.getMessage());

}

String name;

System.out.print(“Enter name: ”);

name = input.nextLine();

System.out.println(“Hello, ” + name + “!”);

}

}

* Standard keyboard input & output statements using the Scanner class
* Statement that defines a Process 🡪 new
* \*Every Java application has a single instance of the Runtime class that allows the program to interface w/ its environment
* **exec()** Method: executes the Operating System program ‘osk.exe’ (on screen keyboard)

🡪 Throws an uncaught IOException 🡪 therefore it is placed in a try block