**COP 2250 – Chapter 12 – Exception Handling and Text I/O**

Exception Handling

* Unless coded to prevent it, runtime errors in a Java program will cause the program to crash and present a stack trace to the user. Java will throw an **exception**.

Try **Quotient,java**

* As a coder, you could deal with this problem with a simple if…else block as shown next.

Try **QuotientWithIf**

* You could also write a method to cope with the possible runtime issue.

Try **QuotientWithMethod**

A better method would throw an exception that could then be handled by the calling method.

Try **QuotientWithException**

* Study closely the explanation of this latter example on page 453. Here is a summary:
* The code that might throw the exception (the call to **quotient**) is put in a **try block**.
* Remember the **try** block code might execute without throwing an exception, too.
* A selection inside quotient is coded to throw an ArithmeticException if division is by 0.
* The **catch** block executes if the exception is thrown, handling it gracefully.
* The catch block takes a **parameter** of **type** ArithmeticException.
* This catch block parameter object has methods you can use (especially **getMessage()**).
* The **getMessage()** method will return the string supplied with the throw statement.
* As noted atop page 454, a method separates detection of an exception from handling it.
* Many methods in the Java API have been coded to throw exceptions.
* Note the nifty use of the **do…while** loop and the **Boolean** variable in the next example.

Try **InputMismatchExceptionDemo**

**Exception Types**

* Exceptions are objects, and they all inherit from the superclass **Throwable**.
* Examine Figure 12.1 showing the inheritance hierarchy of **Throwable** and **Exception**.
* You can extend class Exception, or one of its subclasses, to make your own exceptions, too.

Three types of exception classes:

1. System Errors

* Class **Error** describes these.

1. Subclasses of Exception

* IOException, FileNotFoundException, and dozens more including RunTimeException (covered next). See class Exception in the Java API.

1. Subclasses of RuntimeException

* ArithmeticException, NullPointerException, IndexOutOfBoundsException, etc.

Examine Tables 12.1, 12.2, and 12.3 on pages 456-457 to get familiar with some common exceptions and the situations that will generate them.

Unchecked and Checked Exceptions

* RunTimeException, Error, and their subclasses are **unchecked** (by the compiler).
* If you miss these when coding, they will become apparent at runtime.
* All other exceptions are **checked** exceptions.
* The compiler, and a good IDE, forces you to handle these checked exceptions.
* The preferred technique for handling checked exceptions is by writing try-catch blocks.

More on Exception Handling

* Java handles exceptions with three operations:

1. Declaring an exception

* Every method must declare, in a comma delimited list, all of the exception types it might throw with the **throws** keyword. This is added to the method’s header.

1. Throwing an exception

* The **throw** keyword is used to throw an exception of the appropriate type if an error is detected.
* The new keyword is used to run the exception’s constructor, often with a string argument.
* If an instance of the exception class is specified (see Note on p459), the **getMessage()** method can be used to retrieve the ctor string.

1. Catching an exception

* This done with a **try** block and as many **catch** blocks as needed to handle the possible exception types that might be thrown.
* If the try block throws an exception, each catch block is examined in an attempt to find a match for the exception type.
* If none is found, the exception is passed to the main method and an error message is displayed in red on the console (and we know about that don’t we?☺).
* If multiple catch blocks are required, they should be ordered with subclasses of Exception first.
* Study the scenario on page 460 and examine Figure 12.3. Note the bullet list, too.

Getting Information About an Exception

* The object thrown by an exception holds data about the exception.
* Figure 12.4 on page 461 shows a UML diagram with some useful methods of the exception object that you can use.
* Especially useful are **getMessage(), printStackTrace(),** and **toString()**.

Try **TestException**

Examine **CircleWithException** and try **TestCircleWithException**

The finally clause

* A **finally** block can be optionally included after the catch block(s) if you want some code to execute if an exception is thrown, or not.
* The code in a finally block will always be executed.
* Good usages for the finally block include closing open files and open network connections.

When to Use Exceptions

* Simple exceptions should be handled with try-catch blocks right in the method.
* Don’t use exceptions where simple selections logic is sufficient.

Rethrowing Exceptions (see page 468-4690

* Rethrowing an exception gives other handlers in the caller a shot at handling the exception.

Chained Exceptions

* A second exception can be thrown along with the original to provide additional information.

Custom Exception Classes

* You can extend the Exception class to make custom exceptions for your application.
* You might need an exception to adhere to a business rule, for example.

Examine **InvalidRadiusException** and try **TestCircleWithCustomException**

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> Text I/O <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

The File Class

* This class was created in the API so an object can represent a file.
* The class contains methods for file properties and file manipulation.
* The UML diagram on page 474 shows the names and usages of common File class methods.
* Text files are ASCII files that can be read with programs like Notepad or WordPad.
* The File class creates File objects and has methods for file properties, but reading and writing with files requires other Java classes.

Try **TestFileClass**

File Input and Output

* The **PrintWriter** class in the **java.io** package can be used to write data to a plain text file.
* View its methods on page 476.

**T**ry **WriteData**

Note this line (line 2) in WriteData.java:

public static void main(String[] args) throws Exception {

* Java programs that work with files can encounter issues. When they do, they throw an **Exception**, so we must add the phrase “throws exception” here.

Try **WriteDataWithAutoClose**

Reading A File with Scanner

* The **Scanner** class used earlier in this course can read disk files as well as keyboard input.
* Java, like some other languages, considers files and the keyboard as similar entities.
* Both are viewed as possible sources of a stream of characters.
* Review the methods of Scanner in the UML diagram on page 478.

**T**ry **ReadData**

* We need “throws Exception” here too (see line 4), since the file may not exist etc.
* Scanner class and some skills for it are explained very well on pages 479-482.
* Scanner can be used to replace text in a file. See page 480.

**T**ry **ReplaceText**

Reading Data from the Web

* A Java program can read a file from across the Internet, if the URL of the file is known.
* The java.net.URL class is used to create an object for connecting with the remote resource.
* The openStream() method of the URL object is specified in the Scanner’s ctor to make the connection. Then a loop is used to read the remote file line-by-line.
* The connection could fail for a number of reasons, so several exceptions must be handled.

**T**ry **ReadFileFromURL**

**T**ry **WebCrawler**