Exception Handling

CS 3: Computer Programming in Java
Objectives

- Show the difference between C++ and Java in exception handling
- Define Java exception handling
  - Try
  - Catch
  - Throw
- Talk about types of exceptions
- Catch all the multiple catches you can try
- Reachable vs. unreachable exceptions
- Look at code inside exception handling
- Design your own exception class
Let’s Start with Some Math

Mathematics defines two types of functions:

- Total functions
- Partial functions

TOTAL FUNCTIONS: the value of the function is defined for all values of the variables or arguments provided to the function

PARTIAL FUNCTIONS: there are domains of variable values, where the function behavior becomes indeterminate
Similarly, in Java...

- Methods can be categorized as:
  - Total methods
  - Partial methods

- **TOTAL METHODS**: no restriction on the variables passed to the method at runtime, as the method would handle all errors related to the data passed to it
  - Akin to a total mathematical function being defined for all values of the variables

- **PARTIAL METHODS**: “requires” user to refrain from passing certain variable values as arguments, as the method is not designed to handle such values
  - If a forbidden value were passed to a partial Java method, then an error condition would result
Simple Example:  Divide

- Takes two integers, num and den, as arguments and returns the result num/den
- The pre-condition to be followed in using this method is that user never passes a zero value for den to the method
- Has an undefined result for den equal to zero and would require error handling when user violates the pre-condition
public class DivideProblem1
{
    /*
     * pre-condition: The value of den passed to the method shall never be zero.
     * Returns the integer result after dividing argument num by argument den.
     */
    public static int divide (int num, int den)
    {
        return num/den;
    }
}
Exception Handling in C/C++

- A traditional language like C could pass an extra argument, a pointer to a partial method like `divide` to carry the results of the division of `num` and `den`.
- The method could return a zero if pre-condition is violated.
- In that case, the method would not alter the value of the pointee of the pointer passed to it.
- If denominator is non-zero, then the method could return a value of one and store the result of division in the pointee of the pointer passed to it.
Exception Handling in C/C++: Code Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

int divide(int num, int den, int *result)
{
    if(den == 0)
    {
        return 0;
    }
    else
    {
        *result = (num/den);
        return 1;
    }
}
```
void main()
{
    int * result = (int*) malloc(sizeof(int));
    int num = 5;
    int den = 0;
    if(!divide(num,den, result))
        printf("The illegal denominator of value = %d has been passed to the method divide",den);
    else
        printf("\nThe division %d/%d = %d\n",num,den,*result);
Exception Handling in C/C++: Code Example (3)

```c
num = 25;
den = 5;
if(!divide(num,den,result))
    printf("The illegal denominator of value = %d has been passed to the method divide",den);
else
    printf("\nThe division %d/%d = %d\n",num,den,*result);
```

Trying to do Exception Handling in Java C/C++-Style

- Due to security considerations, Java does not allow pointers to primitives, and for the same reason all wrappers to primitives are also immutable.

- Therefore, the technique used in C programming for error handling (where an error code is returned) cannot be used in Java without an elaborate measure, such as designing a method header as given below:
  
  ```java
  boolean divide (int num, int den, StringBuffer result)
  ```
Trying to do Exception Handling in Java C/C++-Style (2)

- In a divide method like above, if den is not zero, then method would return a true and the value num/den would be appended to (an initially empty) StringBuffer object result.

- Using the algorithm similar to the one shown in main method in the previous code example, the caller method can correctly parse the result of the division.

- The use of C like technology for error handling is tedious, as it requires the client to write extra code for error checking.
Exception Handling in Java

- The inventor of Java and other Object Oriented Languages designed a technology, which allows the methods to return an error condition to the caller method without using the traditional error code return mechanism.
- This technology is called exception handling/error handling.
Examples of Exceptions to Handle

- In an integer arithmetical division, user provides a zero value for the denominator
- In a file input, the wrong file name or non-existent disk drive is provided
- When program expects input of a numeric, a string composed entirely of alphabets is entered
Examples of Exceptions to Handle (2)

- When entering a person’s name in the customer database, only digits are entered
- Disk drive has a physical crash during the data read/write process
- Programmer makes a method call using an object reference, which is a null pointer
- The program tries to access an out of bounds array index
Errors vs. Exceptions

- Java makes a subtle difference between errors and exceptions and designs group of classes accordingly.
- **ERROR:** a program condition, which when manifested, is generally not correctable by adding a piece of code in a class or in the client code.
  - **EXAMPLE:** if, during execution of a Java program, the JVM breaks down, then user code cannot handle such error, and JVM would shut down the program after informing user of the error.
Errors vs. Exceptions (2)

- EXCEPTIONS: conditions that are to some degree controllable, and in some cases, correctable by adding an exception handling code in the classes and/or in the client program
- In this chapter we elucidate exception handling by examples and discuss error handling
The Throwable Class

```
 lombok.Throwables

 Throwable ()
 Throwable (String message)
 Throwable ((Throwable cause)
 Throwable (String message, Throwable cause)

 Accessors
 Throwable getCauses()
 String getLocalizedMessage()
 String getMessage()
 StackTraceElement[] get / set StackTrace()

 Object
 String toSring()

 Other Public Methods
 Throwable fillInStackTrace()
 Throwable initCause (Throwable cause)
 void printStackTrace()
 void printStackTrace (PrintStream s)
 void printStackTrace (PrintWriter s)
```

```
 java.lang.Error
 java.lang.Exception
```
**Key Points about Exception Handling in Java**

- Mechanism is built-in in Java, whereas a developer must add it in C/C++
  - Some exception and error handling is built-in in Java
  - All the user has to do is to learn to use it
- Java accomplishes the error checking and exception handling by instantiating an object of a Java class, which closely relates to the error (or exception) and then throwing that object at the error location
Handling Thrown Exceptions

- Done by putting the error causing code inside a block after the keyword `try` (referred to as a try block), and then catching any exceptions thrown by that code in a block called `catch`.

- Syntax

```java
try {
    //One or more program statements that may throw an exception
}
catch(Exception Ex) {
    //Zero or more statements that execute after thrown object is caught
}
```
About the Try/Catch Blocks

- If an exception condition takes place inside the try block, then an exception object is returned and control of program is immediately transferred to the first statement in the catch block and all the statements inside the try block, after the exception throwing statement are skipped.

- The catch block acts almost like a method call, in the sense that it takes an object of type Exception as argument and then executes the code inside the catch block.
  - The catch block is also called an “exception handler”

- The try and catch blocks are generally, used as a bonded pairs - at least one catch block following a try block.
public static void main(String[] args) {
    int numerator = 0;
    int denominator = 1;
    String Input = JOptionPane.showInputDialog("Please enter the numerator!");
    numerator = Integer.parseInt(Input);
    Input = JOptionPane.showInputDialog("Please enter the NON ZERO - denominator!");
    denominator = Integer.parseInt(Input);
    try {
        JOptionPane.showMessageDialog(null, "The result of division is = " + numerator/denominator);
        System.out.println("The result of division is = " + numerator/denominator);
    }
    catch(Exception e) {
        JOptionPane.showMessageDialog(null, "A Zero denominator has been entered.");
    }
    System.exit(0);
}
About the Divide Example with Exceptions

- When user enters a zero value of denominator, the second statement inside the try block is skipped and control transferred to the catch block, where a pop-up message is displayed.
- The behavior is different from the previous code example in the sense that now the user gets a more descriptive message shown through a JOptionPane and the program exits gracefully without a cryptic message that JVM would have produced.
Rules on Try/Catch Blocks

- Existence of a lone try or catch block is a compile error
- There shall be no other code between the try catch blocks (works similar to if/else structure)
- One or more catch blocks may follow a try block
  - Java will match the type of object (Exception) thrown in the try block with the objects in the arguments of various catch blocks, and wherever it finds the first best match, it will execute the code in that catch block
Rules on Try/Catch Blocks (2)

- In the try block, once the exception is thrown, the code after that point is skipped and java proceeds to execute the first matching catch block.
- The object thrown by the code in try block must be of type java.lang.Exception.
  - Throwing the objects of classes Throwable and of type Error is syntactically correct, but there is no added value in doing so.
Types of Java Exceptions

- Java neatly divides the exceptions in two categories:
  - Exceptions that would be checked by the Java compiler (compile time exceptions)
  - Exceptions that would be thrown by Java Virtual Machine (JVM) at runtime (also called runtime exceptions or unchecked exceptions)
Compile Time/Checked Exceptions

- Also called checked exceptions
- Compiler checks the Java code to ascertain that if the method called throws a checked exception, then the client of that method is forced to call that method only inside a try block
Compiler does not require the code that could throw a run time or unchecked exceptions to be placed in the try block

The objects of type java.lang.Error and java.lang.RuntimeException form the unchecked exception objects
Hierarchy of Exception Classes

- Throwable
  - Error
    - Runtime Exception
      - Derived Classes
  - Exception
    - Derived Classes

Unchecked Exceptions

Checked Exceptions
public class ExceptionTypes
{
    private ExceptionTypes (){}
}

/**
 * Method method1 throws an exception of type
 * Exception, which is a checked exception.
 */
public static int method1(int val) throws Exception
{
    System.out.println("From method1, that throws an Exception object");
    if(val == 0)
    {
        throw new Exception("From method method1");
    }
    return 1;
}
public static int method2(int val) { //throws clause optional
    System.out.println("From method2, that throws a RuntimeException object");
    if (val == 0)
    {
        throw new RuntimeException("From method method2");
    }
    return 1;
}
public static void main (String[] args) {
    // uncommenting the line below would cause a compile error
    // int val1 = method1(0);
    // However, the below will compile.
    int val2 = method2 (0);
    // The call above must be commented out to execute the try catch block below.
    try {
        int val3 = method1 (0);
    } catch (Exception ex) {
        System.out.println (ex);
    }
}
About the Code Example of Checked and Unchecked Exceptions

- Inside method1, there is a statement:
  
  ```java
  throw new Exception("From method method1");
  ```
  
  - The above statement uses the Java reserved word throw followed by the call to the constructor of Java class java.lang.Exception
  - This results in returning an object of type Exception, with the string “From method method1” as being part of the thrown object

- method2 also has a throw statement, which throws an object of type RuntimeException

- Understand that during execution, method1 or method2, could only do one of the two things
  - Perform a normal return, which would be return an integer value of one
  - Return an exception object following the word throw
About method1 in the Code Example of Checked and Unchecked Exceptions

public static int method1(int val) throws Exception //throws clause required

- Since method1 throws an exception of type java.lang.Exception (which is a checked exception), its header must broadcast that with the clause throws ClassName after the argument list
  - Skipping the throws clause in the header of methods that throw a checked exception would be a compile error; the reverse is also true
- If a method broadcasts a checked exception to be thrown, then such exception or its super type must be thrown
- A method can throw more than one exception, in which case they are listed after the keyword throws separated by comma
  - For example, a method that throws three exceptions Exception1, Exception2, and Exception3 (all checked exceptions), would require a header similar to the one given below:

  [Access specifier] return_type methodName ([Argument List]) throws Exception1, Exception2, Exception3, .................
Also clearly understand the difference between reserved words throw and throws

- The word throw is used preceding the instantiation or name of the exception object to be thrown
- On the other hand, the word throws is used as the first word to broadcast the exception(s) that a method would throw
  - The word throws is never used inside a method body and word throw is never used next to the method header after the argument list

A compile error would occur if caller method (main method in this case) tries to call a method that throws a checked exception with out placing such call in a try block followed by a catch block that can process the thrown exception

- Therefore uncommenting the statement
  ```java
  int val1 = method1 (0);
  ```
  - in main would be a compile error
About method2 in the Code Example of Checked and Unchecked Exceptions

- The method2 throws an object of type RuntimeException
  ```java
  int val2 = method2(0);
  ```
- Call to methods that throw an object of type RuntimeException is exempted from both requirements - that is to broadcast the exception being thrown (in the method header) and to place the method call statement in a try block
- Therefore the call below would compile, though the program execution would terminate right after this call is completed
public static void main(String[] args) throws IOException

- We made the main method throw an exception of type IOException because we were calling the method read or readLine inside the main to get user inputted data from the keyboard.
- You would recall that these are the methods in BufferedReader class, which can be used to read a character or a line from the keyboard input.
Includes one Java method and its description:

```java
public int read() throws IOException
public String readLine() throws IOException
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.io.IOException
```

- **Shows that the methods read and readLine throw an exception of type IOException that is derived from the class Exception and is not of type RuntimeException (or of type java.lang.Error)**
  - IOException is a checked exception and the methods that throw an object of this type must either be called inside a try block or the caller to these methods itself must throw IOException or its super class object
  - Using the clause “throws IOException” next to the header of main method gave us an exemption from placing calls to methods read and readLine inside a try block
- **Were we to remove the clause “throws IOException” from the header of main method, we would need to call the methods read and readLine using the syntax similar to the one we used to call the method1**
### Common Checked and Unchecked Exceptions

<table>
<thead>
<tr>
<th>Exception throwing code</th>
<th>Exception class object thrown</th>
<th>Type of the Object thrown</th>
</tr>
</thead>
<tbody>
<tr>
<td>int val = Integer.parseInt(“Hello”);</td>
<td>NumberFormatException</td>
<td>RuntimeException</td>
</tr>
<tr>
<td>int [ ] arr = new int [5]; arr[10] = 10;</td>
<td>ArrayIndexOutOfBoundsException</td>
<td>RuntimeException</td>
</tr>
<tr>
<td>String Str = null; Str.length ( );</td>
<td>NullPointerException</td>
<td>RuntimeException</td>
</tr>
<tr>
<td>Vector V = new Vector ( ); V.firstElement( );</td>
<td>NoSuchElementException</td>
<td>RuntimeException</td>
</tr>
</tbody>
</table>
Common Checked and Unchecked Exceptions (2)

<table>
<thead>
<tr>
<th>Exception throwing code</th>
<th>Exception class object thrown</th>
<th>Type of the Object thrown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object I = new Integer(5); Double Dbl = (Double)(I);</td>
<td>ClassCastException</td>
<td>RuntimeException</td>
</tr>
<tr>
<td>BufferedReader BR = null; try { BR = new BufferedReader (new FileReader (&quot;G:\text.txt&quot;)); } catch(Exception ex) { System.out.println (ex); }</td>
<td>FileNotFoundException</td>
<td>Exception (checked exception)</td>
</tr>
</tbody>
</table>
Java Exception Handling Principles

- Dictates that class designers and clients
  - Are required to provide exception handlers for checked exceptions
    - The methods throwing checked exception must broadcast them in the method header
  - Are not expected to provide any handlers for the exceptions of type java.lang.Error, since recovery from them is generally not possible
  - May provide exception handlers for exceptions of type java.lang.RuntimeException, if programmer from the design of program, can conclude that such handler would provide adequate program recovery
    - The broadcasting of thrown unchecked exceptions in the method header is optional; however, it is a good programming practice to do so
Multiple Catch Clauses

- Java allows programmer to place multiple catch clauses following a single try block.
- The reverse, however, is never true, as a catch block can only be preceded either by a single try block or by another catch block.
- Multiple catch blocks improve the granularity of exception handling and pinpoint the error location precisely.
  - Helps recovery from run time errors.
Multiple Catch Clauses Syntax

```plaintext
try
{
    //statements;
}
catch (exceptionType1 identifier1)
{
    // zero or multiple statements;
}
catch (exceptionType2 identifier2)
{
    // zero or multiple statements;
}
//More catch blocks as needed
```
## Exception Inheritance Hierarchy

<table>
<thead>
<tr>
<th>Exception Names</th>
<th>Inheritance Hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td>IllegalArgumentException and NumberFormatException</td>
<td>java.lang.Object</td>
</tr>
<tr>
<td></td>
<td>java.lang.Throwable</td>
</tr>
<tr>
<td></td>
<td>java.lang.Exception</td>
</tr>
<tr>
<td></td>
<td>java.lang.RuntimeException</td>
</tr>
<tr>
<td></td>
<td>java.lang.IllegalArgumentException</td>
</tr>
<tr>
<td></td>
<td>java.lang.NumberFormatException</td>
</tr>
<tr>
<td>NullPointerException</td>
<td>java.lang.Object</td>
</tr>
<tr>
<td></td>
<td>java.lang.Throwable</td>
</tr>
<tr>
<td></td>
<td>java.lang.Exception</td>
</tr>
<tr>
<td></td>
<td>java.lang.RuntimeException</td>
</tr>
<tr>
<td></td>
<td>java.lang.NullPointerException</td>
</tr>
</tbody>
</table>
### Exception Inheritance Hierarchy (2)

<table>
<thead>
<tr>
<th>Exception Names</th>
<th>Inheritance Hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td>SecurityException</td>
<td>java.lang.Object</td>
</tr>
<tr>
<td></td>
<td>java.lang.Throwable</td>
</tr>
<tr>
<td></td>
<td>java.lang.Exception</td>
</tr>
<tr>
<td></td>
<td>java.lang.RuntimeException</td>
</tr>
<tr>
<td></td>
<td>java.lang.SecurityException</td>
</tr>
<tr>
<td>IOException and</td>
<td>java.lang.Object</td>
</tr>
<tr>
<td>FileNotFoundException</td>
<td>java.lang.Throwable</td>
</tr>
<tr>
<td></td>
<td>java.lang.Exception</td>
</tr>
<tr>
<td></td>
<td>java.io.IOException</td>
</tr>
<tr>
<td></td>
<td>java.io.FileNotFoundException</td>
</tr>
</tbody>
</table>
Reachable vs. Unreachable Exceptions

- When arranging the multiple catch blocks to handle exceptions, Java requires that a subclass exception be caught first, before a super class exception is caught.

Example:
```java
try {
    //code
} catch (FileNotFoundException ex) //sub class
{
    //code
} catch (IOException ex) //super class
{
    //code
}
```
Reachable Exceptions

```java
try {
    //code
} catch (FileNotFoundException ex) //sub class {
    //code
} catch (IOException ex) //super class {
    //code
}
```

Notice that the catch blocks for exceptions are arranged in the order (from top to bottom) that is exact reverse of their inheritance hierarchy.
Java would issue a compile error when such “unreachable” code is encountered because the IOException would also catch the FileNotFoundException.
Example of Reachable Exceptions, Part Deux

```java
try {
    //code
} catch (NumberFormatException NFEx) //Sub class of IllegalArgumentException {
    //code
} catch (IllegalArgumentException IllegalEx) //unchecked exception {
    //code
} catch (NullPointerException NPex) //unchecked exception {
    //code
}
```
Example of Reachable Exceptions, Part Deux (2)

catch (SecurityException SC)  //unchecked exception
{
    //code
}
catch (FileNotFoundException FX)  //checked exception - subclass of IOException
{
    //code
}
catch (IOException ex)  //checked exception - subclass of Exception
{
    //code
}
catch (Exception ex)  //Super class of all exception classes
{
    //code
}
Enforcing the "reverse inheritance hierarchy" rule for the placement of exception handlers, the handler for NumberFormatException precedes its super class IllegalArgumentException

Notice that the handlers for the remaining Exceptions, NullPointerException and SecurityException, have only one requirement: that they be placed before the super class of all exceptions - Exception (if used)

- Their placement with respect to other unrelated classes is unimportant

Also, since the checked and unchecked exceptions are unrelated with respect to their inheritance hierarchy, their relative order of placement is also unimportant
Explanation of the Example of Reachable Exceptions, Part Deux (2)

- We follow this arbitrary order, in which we place the unchecked exception handlers first and then place the checked exception handlers afterwards.
- We place the last catch clause to catch any remaining exception that our code may throw.
- A handler with Exception as its formal argument would catch any exception that were not caught by other handlers preceding.
Multiple Catch Clause Class Design: Class ReadDataFile

- We design a class that would have code and methods to prompt user for the name of an input file and simply print its contents to the console.
- All input and output, involving user would be standard I/O type.
- Let us define the steps in our overall strategy:
  - Set a boolean variable done to false.
  - Inside a do/while loop, do the following:
    - Prompt user for input file name by displaying a message on console.
    - Accept user input for file name from the keyboard.
    - Create a File class object using the string entered in the previous step.
Multiple Catch Clause Class Design: Class ReadDataFile (2)

- Still inside the do/while loop, do the following:
  - Create a FileReader object from the File object created in the previous step
  - Test if file has the read permission; if not, then display the related message and set done to false
  - Ascertain that the file is not hidden; if hidden, then set done to false
  - Ascertain that file has some data in it; if the file is empty, then set done to false
  - Prompt user to provide the size of input buffer
  - Accept the user-inputted string
Multiple Catch Clause Class Design: Class ReadDataFile (3)

Still inside the do/while loop, do the following:

- Parse the String entered in the previous step into an integer
- Create a BufferedReader object by calling the constructor that would take the FileReader object and buffer size
- Pass the object created in the previous step to method readAndPrintFile
- Set done to true
- Handle all exceptions

Post-test the loop condition

NOTE: Code to come!
import java.io.*;

public class ReadDataFile
{
    private ReadDataFile()
    {
        public static void main (String[] args)
        {
            BufferedReader KB = new BufferedReader (new InputStreamReader (System.in));
            String Input = new String();
            boolean done = false;
        }
    }
}
do
{
    try
    {
        System.err.println("Please type the name of data input file and then press enter key.");
        Input = KB.readLine(); // readLine can throw IOException
        File FL = new File(Input); // File class constructor can throw NullPointerException
        FileReader FR = new FileReader(FL); // Can throw FileNotFoundException
        check for length, hidden status, and read permission
    }
}
//check for length, hidden status, and read permission
//Violation of any of them would cause SecurityException to be thrown
if(FL.canRead())
{
    if(!FL.isHidden())
    {
        if(FL.length() != 0)
        {
            System.err.println("Please specify the size of read buffer");
            Input = KB.readLine();
            int buf = Integer.parseInt(Input);
            BufferedReader FReader = new BufferedReader(FR, buf);
            readAndPrintFile(FReader);
            done = true;
        }
    }
}
else
{
    System.err.println("The file has no data in it.");
done = false;
}
}
else
{
    System.err.println("The file is hidden.");
done = false;
}
else
{
    System.err.println("The file does not have read permission");
done = false;
}
}//end of try block
Class ReadDataFile: The Code (5)

catch(NumberFormatException NFEx)//unchecked exception
{
    NFEx.printStackTrace(System.err);
    System.err.println("Non-numeric buffer size has been entered.");
}
catch(IllegalArgumentException IllegalEx)//unchecked exception
{
    IllegalEx.printStackTrace(System.err);
    System.err.println("Illegal buffer size has been entered.");
    done = false;
}
Class ReadDataFile: The Code (6)

catch(NullPointerException NPex) //unchecked exception
{
    NPex.printStackTrace(System.err);
    System.err.println("Null file name or null string has been entered.");
    done = false;
}
catch(SecurityException SC) //unchecked exception
{
    SC.printStackTrace(System.err);
    done = false;
}
Class ReadDataFile: The Code (7)

```java
catch(FileNotFoundException FX) // checked exception - subclass of IOException
{
    FX.printStackTrace(System.err);
    done = false;
}
catch(IOException ex) // checked exception - subclass of Exception
{
    ex.printStackTrace(System.err);
    System.err.println("Input reading error.");
    done = false;
}
catch(Exception ex)
{
    System.err.println("Unknown error/exception has occurred.");
    done = false;
}
} while(!done);
```
public static void readAndPrintFile(BufferedReader BR) {
    String Data = null;
    int count = 0;
    try {
        System.err.println("Printing your file to screen now.");
        while ((Data = BR.readLine()) != null) {
            System.err.println(Data);
            if (++count % 7 == 0) {
                // Additional code here
            }
        }
    } catch (Exception e) {
        // Handle exception
    }
}
try {
    System.err.println("Press enter to continue printing.");
    System.in.read();
} catch (IOException ex) {
    System.err.println("Keyboard input error.");
}

} catch (IOException ex) {
    System.err.println("File reading error.");
} System.err.println("Done printing.");
Classifying Exceptions from Class ReadDataFile

<table>
<thead>
<tr>
<th>Class and method/constructor used</th>
<th>Name of the Exception Thrown</th>
<th>Type of Exception (Checked or unchecked)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BufferedReader/readLine</td>
<td>IOException</td>
<td>checked</td>
</tr>
<tr>
<td>File/Constructor that takes file name string as argument</td>
<td>NullPointerException</td>
<td>unchecked</td>
</tr>
<tr>
<td>FileReader/Constructor that takes File object as argument</td>
<td>FileNotFoundException</td>
<td>checked</td>
</tr>
<tr>
<td>File/canRead</td>
<td>SecurityException</td>
<td>unchecked</td>
</tr>
</tbody>
</table>
Classifying Exceptions from Class ReadDataFile (2)

<table>
<thead>
<tr>
<th>Class and method/constructor used</th>
<th>Name of the Exception Thrown</th>
<th>Type of Exception (Checked or unchecked)</th>
</tr>
</thead>
<tbody>
<tr>
<td>File/isHidden</td>
<td>SecurityException</td>
<td>unchecked</td>
</tr>
<tr>
<td>File/length</td>
<td>SecurityException</td>
<td>unchecked</td>
</tr>
<tr>
<td>BufferedReader.readLine</td>
<td>IOException</td>
<td>checked</td>
</tr>
<tr>
<td>Integer/parseInt</td>
<td>NumberFormatException</td>
<td>unchecked</td>
</tr>
<tr>
<td>BufferedReader/constructor that takes FileReader and integer as buffer size as arguments</td>
<td>IllegalArgumentException</td>
<td>unchecked</td>
</tr>
</tbody>
</table>
We use the object System.err to print message to console, as in some cases where file processing is involved, the System.out may fail to print the message on the console.

On the other hand the System.err.println is always able to print message to the console.

For example, if a code snippet below (in the main method of Class1) is run from the command line with the following command:

```java
System.out.println("Can be directed to a file.");
System.err.println("Cannot be directed to a file.");
>java Class1 > file
```

Then the message “Cannot be directed to a file.” would print to the console, whereas the first message would be redirected to the file.

If user attention is drawn to messages from exception handlers, then it is best to use System.err.println/print methods.
For file handling, we need to check the following conditions:

- File has read permission
- File is not hidden
- File is not empty

We use the File Object FL and File class methods canRead, isHidden, and length for these authentications.

However, depending upon whether the file has read permission or not, these three methods can throw an object of class SecurityException, for which we add a handler.

A system of enclosing if/else blocks filters down to the last block, by when the following facts have been established:

- There is no error reading from keyboard
- A non-null file name has been provided and file exists
- File has read permission, is not hidden and has data in it
The integer parsing process can throw a NumberFormatException.
The BufferedReader constructor used can throw an IllegalArgumentException.
If no exceptions are thrown up, then the object Freader is passed to method readAndPrintFile to read the file and print it to console.
Inside each exception handler we add the code of the following type:

```java
    catch (ExceptionType Ex)
    {
        Ex.printStackTrace (System.err);
        System.err.println ("Message");
        done = false;
    }
```
Overloading printStackTrace and getMessage

- public void printStackTrace()
  - Prints this Throwable and its backtrace to the standard error stream
  - This method prints a stack trace for this Throwable object on the error output stream that is the value of the field System.err
  - The first line of output contains the result of the toString() method for this object
  - Remaining lines represent data previously recorded by the method fillInStackTrace()

- public void printStackTrace(PrintStream s)
  - Prints this Throwable and its backtrace to the specified print stream
  - Parameters: s - PrintStream to use for output
Overloading printStackTrace and getMessage

- public void printStackTrace(PrintWriter s)
  - Prints this Throwable and its backtrace to the specified print writer
  - Parameters: s - PrintWriter to use for output

- public String getMessage()
  - Returns the detail message string of this Throwable
  - Returns: the detail message string of this Throwable instance (which may be null)
Method readAndPrintFile and Nested try Blocks

- The purpose of the method readAndPrintFile is to get a valid BufferedReader reference to a data file (with data in it) and print its contents to console.
- The method uses a EOF controlled loop to read the input file line by line.
- Since readLine method can throw an IOException the loop is placed inside a try block and a handler is added.
- We add additional code to ascertain that user has a chance to pause the program after seven lines.
Method readAndPrintFile and Nested try Blocks (2)

- This requires creation of a “nested” try catch block
  - Notice that there is no limitation in nesting a try-catch block inside another try block as needed
  - The requirement however is that handlers for the nested try block, also must lie inside the enclosing try block
  - The nested try block prints a message for the user and prompts them to press enter key to continue printing the input file to console
- The user input is consumed by the method call System.in.read
- Since the call to read method throws IOException, we add the handler for it as well
- If execution of the method ends normally, then the input file is printed to the console
finally Clause

- There are situations when the exceptions are thrown, user is informed of the error condition and, if program resumption is not possible, then the way of recovery is that the program tries to reclaim resources as soon as possible.

- This reclamation of resources may include:
  - Closing output file, so that the data written so far is saved safely.
  - Closing the input file safely.
  - Closing connections to databases.
  - Close a socket or network connection.
  - Disposing the graphical components using the program memory.
Java allows one to put a clause labeled finally after the try/catch blocks.

The characteristic of the finally block is that, as long as `System.exit(0)` is not called from inside of any of the preceding try and catch blocks, then the code inside the finally block is always executed.
try
{
    //statements;
}
catch (exceptionType1 identifier1)
{
    // zero or multiple statements;
}
catch (exceptionType2 identifier2)
{
    // zero or multiple statements;
}
//more catch blocks if needed
finally
{
    //code
}
Another Acceptable finally Clause Syntax

- Java allows the finally block to be placed right after a try block - missing a catch block altogether
- Therefore the following will compile and could be useful in situations, where catching an exception does not help the situation and the finally block executes the rescue code

```
try
{
    //statements;
}
finally
{
    //code
}
```
Example of the finally Clause

- You have perhaps noticed by now that we did not actually close the input file in the Class ReadDataFile example.
- In order to safely close the input file, we add the always executed (as long as the preceding try and catch blocks do not call System.exit (0)) following piece of code using a finally clause.

```java
finally {
    try {
        BR.close();
    }
    catch (IOException Ex) {
        Ex.printStackTrace(System.err);
        System.err.println("File closing error");
    }
}
```
About the Example of the finally Clause

- Notice that, in this case, the try/catch block inside the finally block is needed as the call to method close would throw an IOException.
- A finally block can also contain try/catch and finally blocks.
- Use of finally block is very useful when the program is writing to an output file and file need to be closed safely, so that user data are saved in spite of a program crash.
- Similar situation exists when the program is writing to the database and connection to database is needed to be closed, lest the database gets corrupted upon program crash.
Coping with Unchecked Exceptions

- Generally checked exceptions have a well-defined hierarchy
  - Thus placing the catch blocks for them in reverse order of their inheritance hierarchy may locate their origin rather well
- Un-checked exceptions do not always have this advantage
  - For example, a NullPointerException can be thrown when either File class constructor gets a null string or a null string is entered for the buffer size
  - Thus in the handler for NullPointerException, we have to be satisfied with the message that “Null file name or null string has been entered.”
Coping with Unchecked Exceptions (2)

- Fortunately, the call to `printStackTrace` method would pinpoint the location as to where exception was thrown; still it is an inconvenient feature to use.
- For an un-checked exception that can be thrown at various locations, each such location needs to be isolated in a separate try block.
- Such “error-trapping” can be expensive in terms of program overhead, however in mission critical software applications, it may be a necessity.
Example of Handling Unchecked Exceptions

- Very often in databases all fields of stored objects are not required to be filled
- Let us assume that our database keeps records of student names, and their GPA
- We may end up creating two vectors from the queries made to our database, that are following

  Vector Names = new Vector(); //Fill Names Vector with student names for a certain class
  Vector GPA = new Vector(); //Fill GPA Vector with the GPA of students
Example of Handling Unchecked Exceptions (2)

- If the two vectors Names and GPA are used independently (as if they have nothing to do with each other) then there is no problem.
- For example, the Names vector can print student names and GPA vector can print a list of GPA and find average GPA, etc.
- The problem comes when we try operations such as:

```java
int ind = 5;
String Student_Name = (String)Names.elementAt(ind);
double current_gpa = Double.parseDouble(GPA.elementAt(ind).toString());
System.out.println("The GPA of student " + Student_Name + " is = " + current_gpa);
```
Example of Handling Unchecked Exceptions (3)

- The code above tries to print the GPA of student at the location index = 5, in the Names vector.
- Even after assuming that we have confirmed by calling the size method to make sure that ind = 5, is a valid element in vector Names, we are naively assuming that database stores GPA for all students!
  - For example, students who are attending college for the first time may not have a GPA and the database may not require a number to be stored in that field.
    - As a result the elementAt method called second time may throw ArrayIndexOutOfBoundsException.
  - Worse yet, knowing that Vector stores Objects, there is no certainty that element at location index = 5 is indeed a Double type element.
  - This can happen if some how the vector GPA became corrupted in the process of adding elements to it.
Code Example of Handling Unchecked Exceptions

//Assume that Vector Names and GPA are already populated with student names
//and GPA respectively. Code tasks to print the GPA of student at index = 5 in
//Names Vector. If all students have GPA then size of both vector must be same.
if(Names.size () != GPA.size ())
    throw new RuntimeException ("Vectors storing student names and GPA “ + “have
different sizes.”);

String Student_Name = new String ();
int ind = 5;

try {
    Student_Name = (String)Names.elementAt (ind); //Line #1
    Double Temp = GPA.elementAt (ind); //Line #2
Code Example of Handling Unchecked Exceptions (2)

```java
    catch(NumberFormatException Ex)
    {
        Ex.printStackTrace (System.err);
    }

    catch(ArrayIndexOutOfBoundsException Ex)
    {
        Ex.printStackTrace (System.err);
    }
```
About the Code Example of Handling Unchecked Exceptions

- The error trapping strategy uses a combination of Vector class method calls, throwing un-checked exception, and use of nested try/catch blocks to handle un-checked exceptions
  - The caller of this code, if they wish, could use a try/catch block also to handle the RuntimeException thrown, when Vector sizes are unequal
- Since we did not ascertain that ind is positive and less than size, we must use the try/catch block when we get the element at the index ind
- If an ArrayIndexOutOfBoundsException would be thrown at line #1 then program would abort and execute the catch block for it
- If Line#1 does not throw an exception, then Line #2 would not either as we have ascertained that size of two vectors are same
However, to trap the parsing error (in case the GPA vector has non Double element at location ind), we call the parseDouble method inside another try block, which is followed by a catch block to handle NumberFormatException.

If Line#3 throws the NumberFormatException, then the program is aborted there and catch block after it is executed.

However, if no exception is thrown on Line #3, then rest of the code executed error free.

This somewhat contrived example illustrates the strategy for coping from unchecked exceptions and their use for error trapping.

In an actual application, a better technique would be to create a Student class, where Name and gpa fields are populated first, and then use a Vector of Student objects to print record of each student, rather than trying to use two parallel vectors the way we used here.
The mechanics of implementing user defined exception classes is quite simple.

In order to design a new checked exception, the user would inherit the new class from the `java.lang.Exception` class or from any of its non-final unchecked exception subclasses.

For new unchecked exception classes, one would inherit the new class from the `java.lang.RuntimeException` class.
Exception and RuntimeException Class Constructors

- Exception()
  - Constructs a new exception with null as its detail message
- Exception(String message)
  - Constructs a new exception with the specified detail message
- Exception(String message, Throwable cause)
  - Constructs a new exception with the specified detail message and cause
- Exception(Throwable cause)
  - Constructs a new exception with the specified cause and a detail message of
    (cause==null ? null : cause.toString()) (which typically contains the class and detail
    message of cause)

- The constructors for class RuntimeException are exactly the same type as they take the same arguments
Example of Designing a User-Defined Exception Class

- Let us assume that we need to design an exception class to facilitate array processing, over and above what Java already provides for us.
- For example, when an array reference is passed to a method as an argument, two things can happen:
  - The array reference may be null, in which case the method can throw a NullPointerException as there is not much processing that method can do with such null reference.
  - Array has a zero size
    - In this case also no processing is possible
    - Method can check the size of array for non-zero value before proceeding further.
However, no standard exception is available that can be thrown if method receives a zero size array.

If method must return a value, then it can simulate array access using the following code and since the array has a zero length, it would throw a `ArrayIndexOutOfBoundsException`:

```java
int[] arr = new int[0];
int val = arr[0];
```
Example of Designing a User-Defined Exception Class (2)

- Client of the method can catch that exception and process it further to know that a zero array size was passed to the method.
- The problem with this is that ArrayIndexOutOfBoundsException is not very informative of the situation that a zero size array was passed to the method.
- Therefore, what is desirable is to define a new Exception class that would inform user of the fact that a zero size array was passed to the method.
  - Let us call this exception class as ZeroArraySizeException.
    - We design it to be a sub class of RuntimeException - which means that it is of unchecked type.
Code Example of Designing a User-Defined Exception Class

```java
public class ZeroArraySizeException extends RuntimeException {
    public ZeroArraySizeException() {
        super();
    }
    public ZeroArraySizeException(String Message) {
        super(Message);
    }
    public ZeroArraySizeException(String Message, Throwable Cause) {
        super(Message, Cause);
    }
    public ZeroArraySizeException(Throwerable Cause) {
        super(Cause);
    }
}
```

- All one has to do is to implement the four constructors and call super class constructor inside each!
- Additional method and fields may be added if further customization is needed
Test Class for the User-Defined Exception Class

public class Class1
{
    public static void main (String[] args)
    {
        int[] arr = new int[]((int)(10*Math.random()));
        boolean OK = true;

        if((int)(10*Math.random())%5 == 0)
        {
            arr =null;
            OK = false;
        }
    }
}
Test Class for the User-Defined Exception Class (2)

```java
if(OK)
{
    for(int ind = 0; ind<arr.length; ++ind)
        arr[ind] = (int)(100*Math.random());

    //print the array.
    System.out.println("Printing your array.");
    for(int ind = 0; ind<arr.length; ++ind)
        System.out.print(arr[ind] + " ");
    System.out.println();
}
```
Test Class for the User-Defined Exception Class (3)

```java
//get and print the smallest value
int min = 0;
try {
    try {
        min = findMin(arr);
        System.out.println("The smallest element in the array printed above is " + min);
    }
    catch(NullPointerException Ex) {
        Ex.printStackTrace(System.err);
    }
} catch(ZeroArraySizeException Ex) {
    Ex.printStackTrace(System.err);
}
```
public static int findMin(int[] arr)
{
    if(arr == null)
        throw new NullPointerException("findMin:The array passed to method is a null " + "reference.");
    if(arr.length == 0)
        throw new ZeroArraySizeException("findMin: The size of the array passed to" + "
\nthe method is zero.");

    int min = arr[0];
    for(int ind=0; ind<arr.length; ++ind)
        if(arr[ind]<min)
            min = arr[ind];
    return min;
}
Alternate Implementation for findMin()

```java
public static int findMin(int [] arr)
{
    int min = 0;

    try
    {
        min = arr[0];
    }
    catch(ArrayIndexOutOfBoundsException Ex)
    {
        throw new ZeroArraySizeException("findMin: The size of the array passed to the method is zero.", Ex);
    }
}
```
Whether to Use Checked Exceptions vs. Unchecked Exceptions

- A harder software engineering choice is to decide whether to design the user defined exception class to be an unchecked exception or checked one.
- There is one school of thought, which includes author Bruce Eckel, who has come out totally in favor of unchecked exceptions.
  - In fact new Microsoft language C# does not have checked exceptions.
- Other experts, such as Barbara Liskov from MIT, prefer a judicious mixture of both.
  - Liskov’s viewpoint is to use unchecked exceptions if the client can easily enforce the pre-conditions dictated by the method being called.
  - This may happen if confirmation techniques are available for easy enforcements of pre-conditions.
Whether to Use Checked Exceptions vs. Unchecked Exceptions (2)

- In our latest code example, the client (main method) can easily check if the array passed to the method `findMin` is a null pointer or a zero size array (in fact, the first version of `findMin` just does that)
  - Then, the exception condition would arise only occasionally through programming slips
    - For example, in Java, it is quite easy to avoid `ArrayIndexOutOfBoundsException` as arrays carry their length
- There are situations, however, when techniques to confirm whether the arguments used for the method would meet pre-condition or not may not exist or may be as expensive or more expensive to use than using a compile time exceptions
  - In such situations, the use of checked exceptions would be cost effective
Scenario for Whether to Use Checked Exceptions vs. Unchecked Exceptions

- Let us assume that a piece of software searches for a key in an array using a method, whose header is

  ```java
  public int search(int [] arr, int key)
  ```

- The method search is supposed to return the index of the array element where key is found scanning the array in the ascending order, otherwise search returns a sentinel value for array index = -1

- If numerous calls were made to search then each time, checking the return value for its sentinel state before it can be employed in further data processing would also be very expensive

- In such situations, using checked exceptions becomes cost effective in terms of processing time
Code Example of Checked Exceptions vs. Unchecked Exceptions

```java
public class ZeroArraySizeException extends RuntimeException {
    public ZeroArraySizeException() {
        super();
    }
    public ZeroArraySizeException(String Message) {
        super(Message);
    }
    public ZeroArraySizeException(String Message, Throwable Cause) {
        super(Message, Cause);
    }
    public ZeroArraySizeException(Throwable Cause) {
        super(Cause);
    }
}
```
public class Class1
{
    public static void main (String[] args)
    {
        int [] arr = new int[100];
        for(int ind = 0; ind<arr.length; ++ind)
            arr[ind] = (int)(100*Math.random());

        //print the array.
        System.out.println("Printing your array.");
        for(int ind = 0; ind<arr.length; ++ind)
        {
            System.out.print(arr[ind] + " ");
            if(ind%10 == 0)
                System.out.println();
        }
        System.out.println();
}
Code Example of Checked Exceptions vs. Unchecked Exceptions (3)

```java
//Just check for 10 randomly generated keys
for(int index = 0; index<10; ++index)
{
    int key = 0;
    try
    {
        key = (int)(100*Math.random());
        int val = search(arr,key);
        System.out.println("The value "+key+" is found in the array at location "+val);
    }
    catch(ElementNotFoundException Ex)
    {
        Ex.printStackTrace(System.err);
    }
}
```

public static int search(int[] arr, int key) throws ElementNotFoundException {
    // prime read to throw implicit NullPointerException and ArrayIndexOutOfBoundsException, in
    // case arr is a null reference or zero length array.
    int value = 0;

    try {
        value = arr[0];
    } catch (ArrayIndexOutOfBoundsException Ex) {
        throw new ZeroArraySizeException("search: The size of the array passed to\nthe method is zero.", Ex);
    }
boolean found = false;
for(int ind = 0; ind<arr.length && !found; ++ind)
{
    if(arr[ind] == key)
    {
        value = ind;
        found = true;
    }
}
if(!found)
    throw new ElementNotFoundException("The element " + key +" was not found in the array.");
return value;
About the Code Example of Checked Exceptions vs. Unchecked Exceptions

- It may be daunting for traditional programmers to see that we are actually using the exception throwing mechanism to print a legitimate message that certain key was not found in the array.

- While this is unconventional, when using checked exceptions, this mechanism is sound and effective.

- Since the technology used in the method search never returns an out of bound array index, such an index can be used further for any automatic data processing, since a check for its non-sentinel state is not required.
Re-throwing or Reflecting Exceptions

- If a method gets a thrown object and it does not have a handler for it, then it would re-throw that object to its caller.

- If the thrown object is checked exception type, then the method getting the object has two choices:
  - Provide a handler for it
  - Broadcast in its header to throw that object

- For example, the method search in throws a checked exception of type `ElementNotFoundException`:
  - Its caller main method must either provide the handler for it or re-throw this object.
Code Example of Re-throwing or Reflecting Exceptions

```java
public static void main(String[] args) {
    Func1();
}
public static void Func1() {
    Func2();
    System.out.println("In method Func1");
}
public static void Func2() {
    Func3();
    System.out.println("In Method Func2");
}
```
Code Example of Re-throwing or Reflecting Exceptions (2)

```java
public static void Func3()
{
    Func4();
    System.out.println("In Method Func3");
}
public static void Func4()
{
    throw new ErrType();
    System.out.println("In Method Func4");
}
```
Mechanisms of the Code Example of Re-throwing or Reflecting Exceptions

- The object is thrown by Func4
- Func3 that called Func4 gets the thrown object, but does not have a handler
- Func3 re-throws the Error Object to its caller Func2
- Func2 does have the handler either, so it also re-throws to its caller Func1
- Func1 has the handler, so it catches the object; program completes successfully; code following the handler in Func1 will be executed
To Re-Throw or to Provide a Handler

- The decision whether a method should provide a handler for explicit or implicit exceptions thrown to it depends upon as to where in the call sequence is the best place to process the thrown object.

- The call sequence we saw in the previous code example can be likened to software layers communicating thru each other.

- The lowest layer where the exception occurs is Func4 and the top layer where program begins is the main method.

- In this sense the main method is the client of all the other layers (Func1, Func2, Func3, and Func4).

- Certainly if Func4 threw a checked exception, the client would need to handle it.
To Re-Throw or to Provide a Handler (2)

- The situation is not so clear for unchecked exceptions, which client may or may not choose to handle.
- The most important consideration here is that if the designer of the class decides to change the implementation of class API, any design change should not break the older client code.
  - For example, if exception ErrType were an unchecked exception and the main method decides to provide a handler, thus altering the code in main as
    ```java
    try
    {
        Func1 ();
    }
    catch (ErrType Ex)
    {
        /*code*/
    }
    ```
To Re-Throw or to Provide a Handler (3)

- Now if at some later date the designer of Func4 changes the method design and decided to throw some exception other than ErrType, then it would break the client code.
- This can also happen if design of other layers (Func1, Func2 etc.) changes to throw some other exception.
- Therefore the guiding criterion in deciding to re-throw or handle an exception is to ascertain that changes do not break the client code.
Summary

Show the difference between C++ and Java in exception handling

Define Java exception handling
  - Try
  - Catch
  - Throw

Types of exceptions
  - Checked
  - Unchecked
Summary (2)

- Catch all the multiple catches you can try
- Reachable vs. unreachable exceptions
- finally clause
- Look at code inside exception handling
- Design your own exception class
- Re-throwing or reflecting exceptions