1 **Handling Exceptions and Text I/O**

CST141

2 **Exceptions (Page 1)**

- *Exceptional events* that occur during run-time that disrupt the program's normal flow
- Including but not limited to:
  - An array index out-of-bounds
  - Arithmetic overflow
  - Division by zero for integers
  - Trying to parse a string with a invalid numeric format to a numeric value

3 **Exceptions (Page 2)**

- Exceptions may come from abnormal events generated during run-time, or …
- They also may be generated *manually* by programmers to handle events that could result in an invalid operation taking place

7 **Using if Statements to Catch Potential Exceptions**

- In previous example, a run-time exception may be generated if a user enters:
  - The value zero (0) for denominator (computers *cannot* divide integers by zero)
  - *Non-integer* values entered for either ints from nextInt() method of the Scanner object
- *Disadvantage* of if processing:
  - Validation checking must take place for both valid and invalid values
  - Easy to miss some errors

13 **Using Methods and System.exit to Catch Potential Exceptions**

- It is possible to use if processing in a called method and the System.exit() method to handle potential exceptions
  - Terminates program execution
- Example:

```
if (denominator == 0)
{
   System.out.println("Cannot divide by zero");
   System.exit(2);
}
```

19 **Exception Processing (Page 1)**

- Deal with abnormal events occurring as a result of some process *during* program execution
- Makes sense to use exception processing when the alternative is that program will:
  - Crash, or …
  - Place the application into an *inconsistent state*
- Used in large systems to handle abnormal events in a *standardized* manner

20 **Exception Processing (Page 2)**

- Generation of an non-normal event is called "throwing an exception"
- Occurs when a method detects an event during run-time with which it cannot deal
- Checks the *type* of exception to see if its parameters *match* one of a set of exception handling procedures

21 **Keywords try and catch (Page 1)**

- A *block* of code in which statements with the potential to "throw" an exception are placed within a try block
- In the same method, the try block is followed immediately by one or more catch blocks:
  - Each catch block is an *exception handling routine* (procedure)
  - Specifies the *type* of exception that it can handle—each exception type is a Class name

22 **Keywords try and catch (Page 2)**

- Format:

```
try
{
   code that may throw an exception
}
catch (ExceptionType exceptionObject)
{
   exception handling code
}
catch (ExceptionType exceptionObject)
{
   another exception handling code
}
```

  – *objectVariable* references exception information

23 **Keywords try and catch (Page 3)**

- Example:

```
try
{
   number1 = reader.nextInt();
   number2 = reader.nextInt();
   quotient = number1 / number2;
}
catch (InputMismatchException ex)
{
   System.out.print("Non-integer input");
}
```

```
catch (ArithmeticException ex)
{
    System.out.print("Division by zero");
}
```

24 **Keywords try and catch (Page 4)**

- If an exception occurs, the program:
  - *Abandons* the try block
  - Attempts to find a catch block that *matches* the exception type
- If an exception *type* matches the exception, its catch block is executed

25 **Keywords try and catch (Page 5)**

- If the exception fails to match the type of any catch block, none of the catch blocks are executed:
  - Application may terminate ("crash" or "hang")
  - Or the application could be placed into an inconsistent state (e.g. if an arithmetic overflow occurred, an invalid result may be stored)

26 **Keywords try and catch (Page 6)**

- If no exceptions are thrown during execution of the try block, the try block *completes* and the catch blocks are *ignored*
- Program execution continues with any statements that follow the last catch block
  - True either way whether there was any exception was thrown or not

27 **The toString() Method of Exception Objects**

- A method of the exception object variable (ex) that returns a String representation of the error message
- Format:

  exceptionObject.toString()
- Examples:

```
System.out.println( ex.toString() );
System.out.println(ex);
```

32 **The Javadoc @throws Tag**

- Javadoc tag that names an exception that a method may throw and provides additional explanation how that exception might occur
- Example:

```
/**
 * ...
 * @throws ArithmeticException if denominator is zero (0)
 * ...
 */
```

33 **The Exception Class (Page 1)**

- Exceptions generated by the occurrence of an *exception* which are built into the JVM

(Java Virtual Machine)
- Exception is the super class for all exceptions

The Exception class *automatically* throws an exception so programmer does *not* have to write the if logic to check for it

34 **The Exception Class (Page 2)**

For example, InputMismatchException is an Exception thrown whenever nextInt() for a Scanner object fails to return an int
- This includes non-integers as well as strings
- True of all "next" methods for primitive types for the Scanner class
- Effectively an Exception is thrown every time a InputMismatchException is thrown (as is any exception subclass)

36 **The Exception Class (Page 3)**

Would it not be simpler just to specify the Exception class all the time?

Implementing every potential exception lets the programmer provide *specific* feedback to the user

40 **Sequencing of catch Blocks (Page 1)**

Every catch block must be *reachable*

Superclass exception catch blocks must *follow* their respective subclass exception catch blocks

Failure to adhere to this principle will result in compile errors

41 **Sequencing of catch Blocks (Page 2)**

Example of *invalid* sequencing:

```
catch (Exception ex)
{
   System.out.print("Invalid input");
}
catch (InputMismatchException ex)
{
   System.out.print("Invalid input");
}
catch (ArithmeticException ex)
{
   System.out.print("Divide by zero");
}
```

42 **Some Exception Classes (Page 1)**

Exception
- Generated when *any* exception occurs
- The *superclass* of all exception classes

NullPointerException

– Attempting to reference an object that does not exist (declared but has not been instantiated)

43 **Some Exception Classes (Page 2)**

- ArithmeticException
  - Division by zero (0) (for *integers only*) and some other arithmetic exceptions
- InputMismatchException
  - For a Scanner object, the input does not match the expected pattern for the method, e.g. for a nextInt()
  - Located in the java.util package

```
import java.util.InputMismatchException;
```

44 **Some Exception Classes (Page 3)**

- ArrayIndexOutOfBoundsException
  - Array index is outside the allowable range
- NegativeArraySizeException
  - Declaring an array with a negative integer size
- NumberFormatException
  - Attempt to *parse* a non-numeric string to a numeric value
  - Or attempt to parse a string with digits and a decimal to an integer type

45 **Which Exceptions to Catch?**

- How do I know which exceptions will be thrown by the methods I use in my programs?
- All potential exceptions are listed in the on-line Java API documentation for each method

52 **The Keyword finally (Page 1)**

- Specifies a *block* that will be executed after the try ... catch blocks have been evaluated
- Guaranteed to be executed:
  - Whether or not an exception is thrown
  - No matter which catch block is executed
  - Whether or not one of the catch blocks executes after an exception is thrown

53 **The Keyword finally (Page 2)**

- Usually designed to release resources that may have been assigned (but *not released* if an exception occurred) during the try block
  - E.g. files, memory, etc.
- If application does not catch the exception, the finally block still will execute before the program *crashes*

54 **The Keyword finally (Page 3)**

- Format:

```
try
```

```
{ statements ... }
catch (OneException objectName)
{ statements ... }
catch (AnotherException objectName)
{ statements ... }
finally
{
   statements;
}
```

60 **The Keyword throws**

- The keyword throws sometimes is used in a method header to declare the exceptions that are thrown by that method
- Format:

  private/public *type methodName*([*params*]) throws *ExceptionList* { ...
- Format:

  public int quotient(int numerator, int denominator) throws ArithmeticException { ...

61 **The Keyword throw (Page 1)**

- *Manually* throws an exception from the called method back to the location of the method call in the try block
- Used in methods of *programmer-defined* exception classes that throw exceptions
- Required if an exception will be thrown in a "called" method but the try...catch logic is located in the "calling" method

62 **The Keyword throw (Page 2)**

- Format:

  throw new *ExceptionType*([*argumentList*]);
- Example:

```
if (denominator == 0)
{
   throw new DivideByZeroException();
}
```

  – The functionality is similar to a return statement (terminates processing of the method and passes new exception object back to calling method)

66 **Exception Classes**

- Written by a programmer to extend some Java API exception type
- They typically have two constructors (which is similar to Java API exception classes):
  – One that takes no arguments and specifies a hard-coded default exception message
  – One that takes a string argument—usually a more specific exception message

72 **The getMessage() Method**

- Method of the exception object that returns a descriptive String message stored in an exception object reference
  - Either a default or programmer custom message
- Format:
  exceptionObject.getMessage()
- Example:
  JOptionPane.showMessageDialog(null, ex.getMessage());

73 **The printStackTrace() Method (Page 1)**

- Displays the following:
  - The exception *type*
  - The exact *statement* in the execution of the Java class that threw the exception
  - If more than one method was involved, the sequence of *method calls* leading to the exception (in reverse order of the calls)
- Outputs to the standard error stream …
  - Usually the command line or console window

74 **The printStackTrace() Method (Page 2)**

- Format:
  exceptionObjectName.printStackTrace();
- Example:
  ex.printStackTrace();
  - *Not* a returned String that can be displayed (the statement *stands alone*)

79 **File and Database Examples**

- Banking records including ATM's
- Order entry and billing
- Personnel and payroll
- Customer, client, contacts, etc.
- Inventory control
- Course scheduling, student records including schedule as well as tuition and fees

82 **Files in Java**

- Java has no specific functionality to impose structure on data in a file
- Programmer writes code to organize the data manually into files, records and fields

83 **Streams**

- Java represents text data in Unicode characters composed of two bytes
- A stream is a series of characters used for input or output
- In Java there are several classes used for I/O (input/output) stored in (and imported from) the java.io package

84 **The Standard Output Stream**

- Member of class System, called System.out
- By default, it is associated with the console (terminal window) …
  - But can be changed to another output device, e.g. a disk output file
- Normally uses print() and println() to direct output to the console
  - But if redirected to disk, print() and println() will write to a file

85 **The Standard Input Stream**

- Member of class System, called System.in
- By default, it is associated with the console (keyboard) …
  - But can be changed to another input device, e.g. an disk input file
- Normally a Scanner object uses next(), nextInt(), etc. to read input from console
  - But if redirected to disk, next() and nextInt() will read from a file

86 **Output to a Sequential (Text) Disk File**

- Requires four (4) steps:
  1. Create a File object and associate it with a disk file
  2. Assign the File object as the argument to the constructor of a new PrintWriter object giving it output functionality
  3. Write information to the PrintWriter output file using methods print() and println()
  4. At the end close() the file

87 **The File Class (Page 1)**

- A *reference* type (object) that encapsulates (stores) information about a file
  - E.g. filename, path, etc.
  - As per *Step 1* previously
- Found in the java.io package

```
import java.io.File;
```

88 **The File Class (Page 2)**

- Format:

```
File fileObject = new File("path/filename")
```

  - The *path* is the folder structure that is relative to the folder that contains the ".class" file
- Example:

```
File file = new File("App_Data/names.txt");
```

89 **The PrintWriter Class (Page 1)**

- Takes a File object as argument to the constructor
  - As per *Step 2* previously
- A object of type PrintWriter gives "*write to file*" functionality to the print() and println() methods
- Found in the java.io package

```
import java.io.PrintWriter;
```

90 **The PrintWriter Class (Page 2)**

- Format:
  PrintWriter *printWriterObject* = new PrintWriter(*fileWriterObject*);
- Examples:
  PrintWriter outWriter = new PrintWriter(file);

91 **Using println() and print() with a PrintWriter Object (Page 1)**

- Members of an object instantiated from the PrintWriter class
- Writes characters to a text output file
  – An alternate to sending output to System.out
  – As per *Step 3* previously

92 **Using println() and print() with a PrintWriter Object (Page 2)**

- Formats:
  printWriterObject.println(*outputObject*);
  printWriterObject.print(*outputObject*);
- Example:
  outWriter.println(name);

93 **The close() Method (Page 1)**

- A method of most I/O objects that *closes* a file stream object
  – As per *Step 4* previously
- For output files:
  – Ensures that all data is written to disk (none remains in RAM output buffer)
  – Places trailer labels (needed by O/S) at the *end of file*

94 **The close() Method (Page 2)**

- Format:
  inputOrOutputObject.close();
  – Either PrintWriter or Scanner object
- Example:
  outWriter.close();

95 **The IOException Class (Page 1)**

- An exception that has the potential to be thrown for any statement that reads from or writes to a file
- Super class to FileNotFoundException and EOFException

96 **The IOException Class (Page 2)**

- All I/O operations have the potential to throw an exception
  – Always use try...catch with an IOException for I/O operations (or declare the IOException in a throws clause in the method header)
  – The *compiler* requires this "catch" since IOException is considered a "checked" exception
- Located in the java.io package
  import java.io.IOException;

97 **The IOException Class (Page 3)**

Format:

```
catch (IOException exceptionObject)
```

Example:

```
try
{
   ...
}
catch (IOException ex)
{
   ...
}
```

98 **The IOException Class (Page 4)**

For documentation it may be named in a throws clause, e.g.

```
public static void main(String[] args) throws IOException
```

103 **Combining Object Definitions (File and PrintWriter)**

File and PrintWriter objects can be declared together in a single statement

The two (2) statements:

```
File file = new FileWriter("App_Data/names.txt");
PrintWriter outWriter = new PrintWriter(file);
```

Combined become a single statement:

```
PrintWriter outWriter = new PrintWriter( new File("App_Data/names.txt") );
```

105 **Input from a Sequential (Text) Disk File**

Requires four (4) steps:

1. Create a File object and associate it with a disk file
2. Assign the File object (rather than System.in) as the argument to the constructor of a new Scanner object giving it input functionality
3. Read information from the Scanner input file using methods next() , nextInt(), etc.
4. At the end close() the file

106 **The Scanner Class for File Input (Page 1)**

The same Scanner class used for input from the keyboard (System.in) but which uses a File object to give it disk file input functionality

– As per *Step 2* previously

Object gives "*read from file*" functionality to next(), nextInt() and other Scanner methods

Located in the java.util package

```
import java.io.Scanner;
```

107 **The Scanner Class for File Input (Page 2)**

Format:

Scanner *scannerObject* = new Scanner(*fileObject*);

- Examples:

```
Scanner inReader = new Scanner(file);
Scanner inReader = new Scanner( new File("App_Data/names.txt") );
```

108 **The next*Primitive*() Methods for File Input (Page 1)**

- Same set of methods learned previously from the Scanner class but applied to objects instantiated using a File object
- Reads string and/or primitive token (value) from an input file
  - As per *Step 3* previously
  - A token is all the characters up to the next blank space or a new line

109 **The next*Primitive*() Methods for File Input (Page 2)**

- Format:

scannerObject.next*PrimitiveType*();

- Examples:

```
String name = inReader.next();
int age = inReader.nextInt();
```

113 **The JFileChooser Class (Page 1)**

- Class used for selecting files with "Open" and "Save" dialogs
- Optional String argument in the call to the constructor method is the *path* of the *default directory* when the dialog first displays ...
- Located in the javax.swing package

```
import javax.swing.JFileChooser;
```

114 **The JFileChooser Class (Page 2)**

- Format:

JFileChooser *fileChooserObject* = new JFileChooser(["*path*"]);

  - The *path* is the folder structure that is relative to the folder that contains the ".class" file

- Example:

```
JFileChooser fileChooser = new JFileChooser("App_Data");
```

115 **The showOpenDialog() Method (Page 1)**

- Method is a member of objects instantiated from the JFileChooser object
- Displays a GUI "Open" dialog window
- Allows users to select *path* and *filename* of the file to be opened
- Argument specifies *where* the dialog will be displayed:
  - this—centered in the dialog's *parent* window
  - null—centered on the screen

116 **The showOpenDialog() Method (Page 2)**

- Returns an int which specifies which button in dialog was clicked, <Open> or <Cancel>

- Programmer may test the *return value* by comparing it to static constants:
  - JFileChooser.APPROVE_OPTION (value = 0)
  - JFileChooser.CANCEL_OPTION (value = 1)
- Consider the following example:

```
if (result == JFileChooser.APPROVE_OPTION)
```

117 **The showOpenDialog() Method (Page 3)**

- Format:

```
jFileChooserObject.showOpenDialog( this/null );
```

- Example:

```
result = fileChooser.showOpenDialog(null);
```

- The int return value indicates which button was clicked, <Open> or <Cancel>

```
if (result == JFileChooser.CANCEL_OPTION)
```

118 **The getSelectedFile() Method (Page 1)**

- A method of a JFileChooser object that returns a File object
- The returned object is the JFileChooser's *path* and *filename* selected by the user in an "Open" or "Save" dialog
- Returns object of type File (not a String)

119 **The getSelectedFile() Method (Page 2)**

- Format:

```
jFileChooserObject.getSelectedFile();
```

- Example:

```
File fileName = fileChooser.getSelectedFile();
```

  - Method getSelectedFile() is considered a helper method since it returns an object of type File even though is not a constructor

124 **The showSaveDialog() Method (Page 1)**

- Method is a member of objects instantiated from the JFileChooser object
- Displays a GUI "Save" dialog window
- Allows users to select the *path* and enter the *filename* of the file to be saved
- Argument is the same as showOpenDialog (specifies *where* dialog will be displayed):
  - this—centered in the dialog's *parent* window
  - null—centered on the screen

125 **The showSaveDialog() Method (Page 2)**

- Format:

```
jFileChooserObject.showSaveDialog( this/null );
```

- Example:

```
result = fileChooser.showSaveDialog(null);
if ( fileChooser.showSaveDialog(null) == JFileChooser.APPROVE_OPTION )
```