Advanced Java

LESSON 2: EXCEPTIONS AND IO

Creative Commons

Attribution 4.0 International (CC BY 4.0)

This workforce product was funded by a grant awarded by the U.S. Department of Labor's Employment and Training Administration. The product was created by the grantee and does not necessarily reflect the official position of the U.S. Department of Labor. The U.S. Department of Labor makes no guarantees, warranties, or assurances of any kind, express or implied, with respect to such information, including any information on linked sites and including, but not limited to, accuracy of the information or its completeness, timeliness, usefulness, adequacy, continued availability, or ownership. This is an equal opportunity program. Assistive technologies are available upon request and include Voice/TTY (771 or 800-947-6644).

Agenda

- Exceptions
- Throws
- Text I-O

Exceptions

Handling Exceptions

- An exception is an object that is generated as the result of an error or an unexpected event.
- Exception are said to have been “thrown.”
- It is the programmers responsibility to write code that detects and handles exceptions.
- Unhandled exceptions will crash a program.

Example: BadArray.java

- Java allows you to create exception handlers.

Handling Exceptions

- An exception handler is a section of code that gracefully responds to exceptions.
- The process of intercepting and responding to exceptions is called exception handling.
- The default exception handler deals with unhandled exceptions.
- The default exception handler prints an error message and crashes the program.

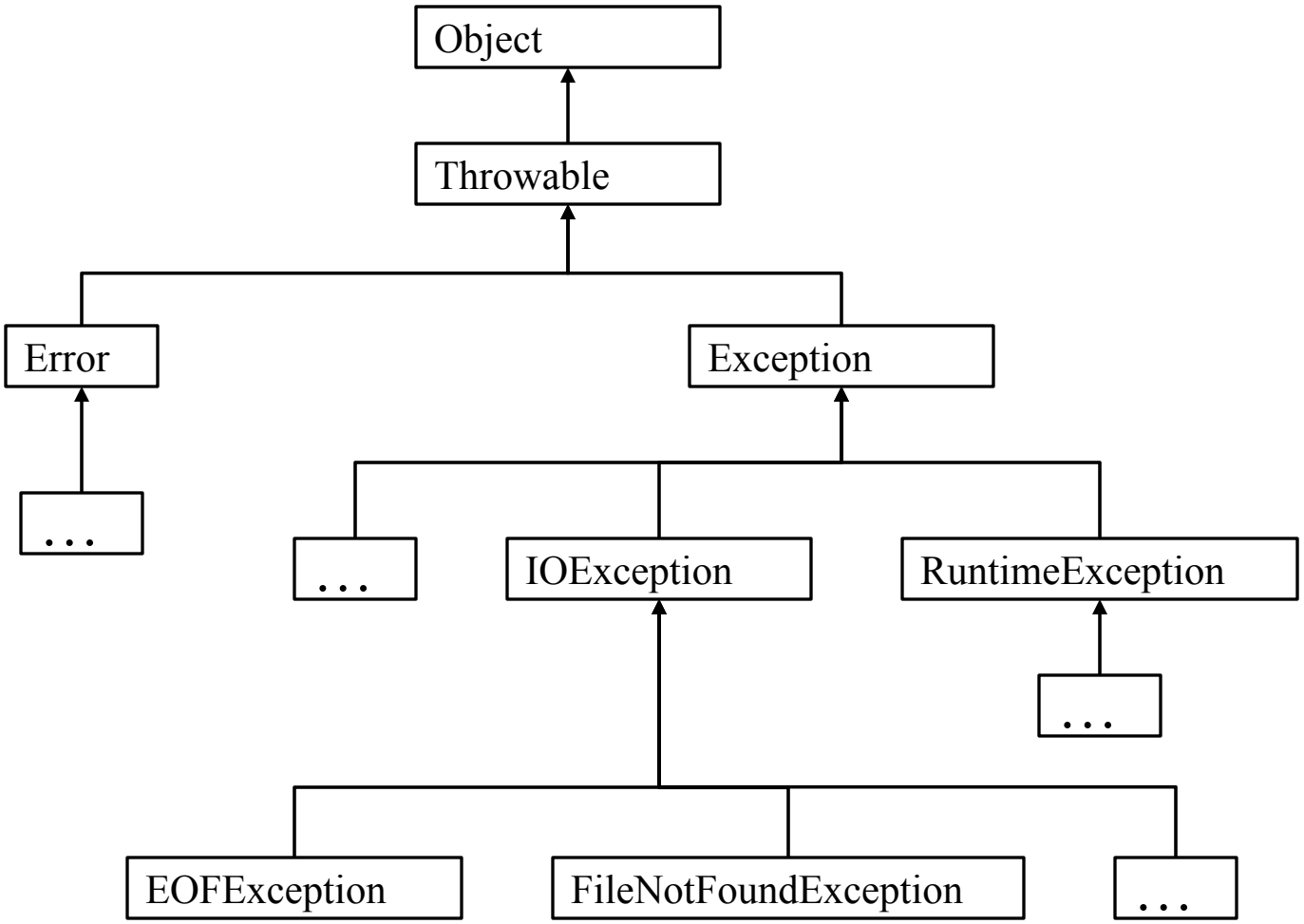
Exception Classes

- An exception is an object.
- Exception objects are created from classes in the Java API hierarchy of exception classes.
- All of the exception classes in the hierarchy are derived from the `Throwable` class.
- `Error` and `Exception` are derived from the `Throwable` class.

Exception Classes

- Classes that are derived from `Error`:
  - are for exceptions that are thrown when critical errors occur. (i.e.)
    - an internal error in the Java Virtual Machine, or
    - running out of memory.
- Applications should not try to handle these errors because they are the result of a serious condition.
- Programmers should handle the exceptions that are instances of classes that are derived from the `Exception` class.

Exception Classes



Handling Exceptions

To handle an exception, you use a *try* statement.

```
try
{
   (try block statements...)
}
catch (ExceptionType ParameterName)
{
   (catch block statements...)
}
```

First the keyword `try` indicates a block of code will be attempted (the curly braces are required).

This block of code is known as a *try block*.

Handling Exceptions

A *try block* is:

- one or more statements that are executed, and
- can potentially throw an exception.

The application will not halt if the try block throws an exception.

After the try block, a `catch` clause appears.

Handling Exceptions

A catch clause begins with the key word `catch`:

- **`catch (ExceptionType ParameterName)`**
- *`ExceptionType`* is the name of an exception class and
- *`ParameterName`* is a variable name which will reference the exception object if the code in the try block throws an exception.

The code that immediately follows the catch clause is known as a *catch block* (the curly braces are required).

The code in the catch block is executed if the try block throws an exception.

Handling Exceptions

This code is designed to handle a `FileNotFoundException` if it is thrown.

```
try
{
   File file = new File ("MyFile.txt");
   Scanner inputFile = new Scanner(file);
}
catch (FileNotFoundException e)
{
   System.out.println("File not found.");
}
```

The Java Virtual Machine searches for a `catch` clause that can deal with the exception.

Example: OpenFile.java

Handling Exceptions

- The parameter must be of a type that is compatible with the thrown exception's type.
- After an exception, the program will continue execution at the point just past the catch block.

Default Error Message

Each exception object has a method named getMessage that can be used to retrieve the default error message for the exception.

Example:

- ExceptionMessage.java
- ParseIntError.java

Polymorphic References To Exceptions

- When handling exceptions, you can use a polymorphic reference as a parameter in the catch clause.
- Most exceptions are derived from the Exception class.
- A catch clause that uses a parameter variable of the Exception type is capable of catching any exception that is derived from the Exception class.

Polymorphic References To Exceptions

```
try
{
  number = Integer.parseInt(str);
}
catch (Exception e)
{
System.out.println("The following error occurred: "
                   + e.getMessage());
}
```

The `Integer` class's `parseInt` method throws a `NumberFormatException` object.

The `NumberFormatException` class is derived from the `Exception` class.

Handling Multiple Exceptions

- The code in the try block may be capable of throwing more than one type of exception.
- A catch clause needs to be written for each type of exception that could potentially be thrown.
- The JVM will run the first compatible catch clause found.
- The catch clauses must be listed from most specific to most general.
- Example: SalesReport.java, SalesReport2.java

Exception Handlers

There can be many polymorphic catch clauses.

A try statement may have only one `catch` clause for each specific type of exception.

```
try
{
  number = Integer.parseInt(str);
}
catch (NumberFormatException e)
{
  System.out.println("Bad number format.");
}
catch (NumberFormatException e) // ERROR!!!
{
  System.out.println(str + " is not a number.");
}
```

Exception Handlers

The `NumberFormatException` class is derived from the `IllegalArgumentException` class.

```
try
{
  number = Integer.parseInt(str);
}
catch (IllegalArgumentException e)
{
  System.out.println("Bad number format.");
}
catch (NumberFormatException e) // ERROR!!!
{
  System.out.println(str + " is not a number.");
}
```

Exception Handlers

The previous code could be rewritten to work, as follows, with no errors:

```
try
{
  number = Integer.parseInt(str);
}
catch (NumberFormatException e)
{
  System.out.println(str
                     + " is not a number.");
}
catch (IllegalArgumentException e) // OK
{
  System.out.println("Bad number format.");
}
```

The `finally` Clause

The try statement may have an optional `finally` clause.

If present, the `finally` clause must appear after all of the `catch` clauses.

```
try
{
   (try block statements...)
}
catch (ExceptionType ParameterName)
{
   (catch block statements...)
}
finally
{
   (finally block statements...)
}
```

The finally Clause

The *finally block* is one or more statements,

- that are always executed after the try block has executed and
- after any catch blocks have executed if an exception was thrown.

The statements in the finally block execute whether an exception occurs or not.

The Stack Trace

- The *call stack* is an internal list of all the methods that are currently executing.
- A *stack trace* is a list of all the methods in the call stack.

It indicates:

- the method that was executing when an exception occurred and
- all of the methods that were called in order to execute that method.

Example: StackTrace.java

Uncaught Exceptions

- When an exception is thrown, it cannot be ignored.
- It must be handled by the program, or by the default exception handler.
- When the code in a method throws an exception:
  - normal execution of that method stops, and
  - the JVM searches for a compatible exception handler inside the method.

Uncaught Exceptions

- If there is no exception handler inside the method:
  - control of the program is passed to the previous method in the call stack.
  - If that method has no exception handler, then control is passed again, up the call stack, to the previous method.
- If control reaches the main method:
  - the main method must either handle the exception, or
  - the program is halted and the default exception handler handles the exception.

Checked and Unchecked Exceptions

- There are two categories of exceptions:
  - unchecked
  - checked.
- Unchecked exceptions are those that are derived from the Error class or the RuntimeException class.
- Exceptions derived from Error are thrown when a critical error occurs, and should not be handled.
- RuntimeException serves as a superclass for exceptions that result from programming errors.

Checked and Unchecked Exceptions

- These exceptions can be avoided with properly written code.
- Unchecked exceptions, in most cases, should not be handled.
- All exceptions that are not derived from Error or RuntimeException are checked exceptions.

Checked and Unchecked Exceptions

- If the code in a method can throw a checked exception, the method:
  - must handle the exception, or
  - it must have a throws clause listed in the method header.
- The throws clause informs the compiler what exceptions can be thrown from a method.

Checked and Unchecked Exceptions

```
// This method will not compile!
public void displayFile(String name)
{
   // Open the file.
   File file = new File(name);
   Scanner inputFile = new Scanner(file);
   // Read and display the file's contents.
   while (inputFile.hasNext())
   {
     System.out.println(inputFile.nextLine());
   }
   // Close the file.
   inputFile.close();
}
```

Checked and Unchecked Exceptions

- The code in this method is capable of throwing checked exceptions.
- The keyword throws can be written at the end of the method header, followed by a list of the types of exceptions that the method can throw.

```
public void displayFile(String name)
    throws FileNotFoundException
```

In-Class Activity

1. Download “Animals” from Blackboard and open in IntelliJ.
2. Add a class for user input.
3. Replace the code in the Main class between the comments with instantiation of the User Input Class and a call to the method that accepts input. (hint: pass the list to the IO Object)
4. Using Decisions and Exceptions, verify all the data entered by the user is of the correct data type.

Throws

Throwing Exceptions

- You can write code that:
  - throws one of the standard Java exceptions, or
  - an instance of a custom exception class that you have designed.
- The `throw` statement is used to manually throw an exception.

```
throw new ExceptionType(MessageString);
```

- The `throw` statement causes an exception object to be created and thrown.

Throwing Exceptions

- The *MessageString* argument contains a custom error message that can be retrieved from the exception object's `getMessage` method.
- If you do not pass a message to the constructor, the exception will have a null message.

```
throw new Exception("Out of fuel");
```

Note: Don't confuse the `throw` statement with the `throws` clause.

Example:

- InventoryItem.java
- InventoryDemo.java

Creating Exception Classes

- You can create your own exception classes by deriving them from the `Exception` class or one of its derived classes.

  Example:
  - BankAccount.java
  - NegativeStartingBalance.java
  - AccountTest.java

Creating Exception Classes

- Some examples of exceptions that can affect a bank account:
  - A negative starting balance is passed to the constructor.
  - A negative interest rate is passed to the constructor.
  - A negative number is passed to the deposit method.
  - A negative number is passed to the withdraw method.
- The amount passed to the withdraw method exceeds the account's balance.
- We can create exceptions that represent each of these error conditions.

In-Class Activity

1. Create a class, “InputTypeVerifier” in “Animals” that checks for correct datatype and throws an exception if incorrect. (hint: always pass user input as a String and return as the correct datatype)
2. Modify “Animals” to use “InputTypeVerifier” inside of a try/catch block .
3. Add a loop that accepts user input, adds to the ArrayList which then follows by adding to the output file.
4. Test the InputVerifier to the program.

I-O

Readers, Writers, and Streams

- Input and output in Java is accomplished by classes called streams
- Input streams provide ways to move bytes of data from an input device to a program
- Output streams move the data in the opposite direction

Predefined Streams

- **`System.in`**
  - InputStream object, usually for the keyboard
- **`System.out`**
  - a buffered PrintStream object, usually the screen or an active window
- **`System.err`**
  - an unbuffered PrintStream object usually associated with the screen or console window

Wrapping Classes

- **`InputStream`** class provides methods for reading bytes only
  - to read at a higher level we must "wrap" System.in with another class

  Example:

```
BufferedReader inFile =
    new BufferedReader (
    new InputStreamReader(System.in));
```

- Now we can send inFile either
  - the **`.read()`** message for a single char value or …
  - the **`.readLine()`** message for an entire line of text

Using a Reader

- Wrap **`FileReader`** in **`BufferedReader`** class

```
BufferedReader inFile =
    new BufferedReader (
    new FileReader (inputFilename) );
```

- Now the **`BufferedReader`** object (inFile) can be sent a **`.readLine()`** message
- A **`BufferedReader`** is so named because it buffers the input
  - this improves program performance

Using a Reader

- How to know when all the data in a file has been read?
  - the **`readLine()`** method returns the value null

```
String line = inFile.readLine():
while (line != null)
   {    … // process valueString
     line = inFile.readLine();  }
```

- Also possible to check for other things such as an empty line with no text

```
if (line.equals("")) continue;
```

Using a Reader

Steps for processing text input

1. Build a **`BufferedReader`** by wrapping one around a **`FileReader`**
2. Use an input loop to read lines of text from the **`BufferedReader`**
   - Convert those lines to numeric values if necessary
3. Close the **`BufferedReader`**

Using a Writer

- When writing values to a text file, author suggests use a Writer class
  - increase **`FileWriter`** capabilities by wrapping it in **`BufferedWriter`** and **`PrintWriter`** classes

```
PrintWriter outFile = new PrintWriter(
    new BufferedWriter (
    new FileWriter (file_name) ) );
```

Classes

- **`FileWriter`**
  - A convenience class for writing character files.
- **`BufferedWriter`**
  - Writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings.
- **`BufferedReader`**
  - reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.
- **`FileReader`**
  - A convenience class for writing character files.

In-Class Activity

1. Add all JavaDoc comments to “Animals”.
2. Create Online Documentation and show instructor.
3. Create Unit tests.
4. Run and show instructor.