Exceptions

CST 365 – Web Applications
Michael Ruth, Ph.D.
Assistant Professor
Computer Science & I.T.
mruth@roosevelt.edu

Objectives

- Discuss program robustness, possible errors, and the OOP concept of exceptions
- Examine the types of errors that a program can cause and the exception hierarchy in Java
- Discuss handling exceptions using try, catch, and finally blocks in Java
- Explain the exception handling process as it occurs in Java including the types of exceptions

Robustness

- ***robustness:*** a program's ability to spot exceptional conditions and deal with them or shutdown gracefully
- Things that can go wrong:
  - Logic error
  - Environment error
  - I/O error

Exceptions

- ***Exceptions*** are Java's way of telling you something has gone wrong
  - We say that exceptions are ***thrown***
  - An exception object is created storing information about the nature of the exception
    - Kind of exception, where it occurred, etc.
- The JVM looks for a block of code to ***catch*** or handle the exception
  - **catch =** do something with it!

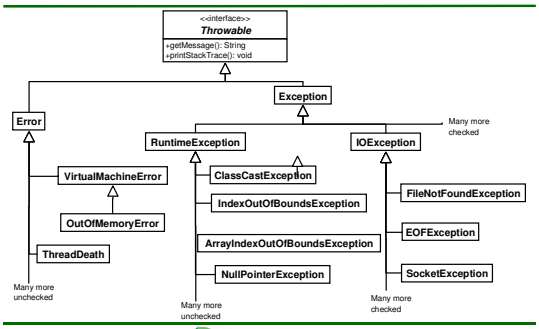
Handling Exceptions

- An ***exception handler*** is a section of code that gracefully responds to exceptions
- 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!
  - Has attributes/methods such as getMessage()
- 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 it

Exception Classes



Handling Exceptions

- To handle an exception, you use a ***try*** statement:

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


Handling Exceptions

- A *try block* is:
  - one or more statements that are executed
  - where an exception ***might be*** thrown
- After try blocks, at least one or more **`catch`** clause/block(s) will appear:
  - The code in the catch block is executed if the try block throws an exception of the type specified in the catch block clause

Catch Clause/Block

- A catch clause/block begins with the key word **`catch`**:

```
catch (ExceptionType ParameterName) {
   //body (AKA catch block)
}
```

  - **ExceptionType** is the name of an exception class that will be handled by this block
  - **ParameterName** is the variable name of the exception object being thrown here

Example

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


Polymorphism & Exceptions

- Using polymorphic references in the catch clause allows you to catch all derivatives of specific exceptions:
  - For instance,
    - Using **`IOException`** catches all classes derived from the **`IOException`** class
    - Using **`Exception`** catches all exceptions derived from the **`Exception`** class

Multiple Exceptions?

- The code in the try block may be capable of throwing more than one type of exception.
- A catch block needs to be written for each type of exception that could be thrown
  - The JVM will run the first compatible catch clause found
  - The catch clauses must be listed from most specific to most general (compiler error if !)

Multiple Exception Handlers (Ex)

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

```
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.");
}
```


`finally`

- 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...)
}
```

`finally` (cont.)

- The **`finally`** block is one or more statements that are executed
  - after the try block has executed
  - after any catch blocks have executed if an exception was thrown
- The statements in this 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 (usually in relation to an exception)
  - 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.

Exception Handling Process

- Every exception must be handled
  - by program or default exception handler
- If the code in a method throws an exception:
  - normal execution of that method stops
  - the JVM searches for a compatible exception handler inside the method
    - 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

Uncaught Exceptions

- In this process, 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
    - Usually, it prints a stack trace and exits ☹

Checked/Unchecked

- There are two categories of exceptions:
  - **unchecked** (derived from Error/RuntimeException)
  - **checked** (! derived from Error/RuntimeException)
- **`RuntimeException`** serves as a superclass for exceptions that result from programming errors
  - These exceptions can be avoided with properly written code.
- **unchecked exceptions**, in most cases, **should not be handled!**

checked

- The compiler checks to see who will handle the exception for checked exceptions!
- If the code in a method can throw a checked exception, the method must either:
  - handle the exception itself
  - have a **`throws`** clause listed in the method header
    - informs the compiler which exceptions can be thrown from a method
      - so the compiler can check somewhere else to see who will handle it

Checked Exceptions Example

```
// 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 Exceptions Example

- 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
```


Throwing Exceptions

- 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
  - 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

Summary

- Discussed program robustness, possible errors, and the OOP concept of exceptions
- Examined the types of errors that a program can cause and the exception hierarchy in Java
- Discussed handling exceptions using try, catch, and finally blocks in Java
- Explained the exception handling process as it occurs in Java including the types of exceptions

Questions?