

EXCEPTION HANDLING

CS302 – Introduction to Programming
University of Wisconsin – Madison
Lecture 28

By Matthew Bernstein – matthewb@cs.wisc.edu

What is Exception Handling?

- According to Horstmann Ch. 7:
 - “There are two aspects to dealing with program errors: *detection* and *handling*”
 - “In Java [and other programming languages], **exception handling** provides a flexible mechanism for passing control from the point of error detection to a handler that can deal with that error”

What is an Exception?

- Oracle defines an **exception** as “an event, which occurs during the execution of the program, that disrupts the normal flow of the program’s instructions”
- In Java, an **Exception object** is an object that stores information about an exception that occurred in your program

Throwing Exceptions

- When you detect an error in your program, you **throw** an Exception object
- Throwing an exception object tells your program that an error was detected that needs to be handled
- For example, let's say we have a method that withdraws money from an account
- We would throw an exception if the amount is greater than the balance:

```
if (amount > balance)
{
    throw new RuntimeException("Amount exceeds balance");
}
```

Error message stored in
the Exception object



throw is a Java reserved word



Create a new **Exception** object



Throwing Exceptions

- When you throw an exception, execution of your program is transferred to code that handles the exception.
- The next instruction is NOT executed
- Example:

```
if (amount > balance)
{
    throw new RuntimeException("Amount exceeds balance");
}
```

If this exception is thrown
this code is not executed

```
balance = balance - amount;
```

Catching Exceptions

- An exception is handled by **catching** the exception
- Think of it like a baseball player throwing execution from the point of detection to the handler. We **throw** the exception from the point of error and we **catch** it where we handle the error



The exception

Catching Exceptions

- We catch an exception in a **catch-block**
- The code inside a catch-block is executed when an exception is caught in the preceding **try-block**

```
try
{
    // An exception may be thrown somewhere
    // in here
}
catch (RuntimeException e)
{
    // If the exception occurs, we handle it here
}
```

try-block and catch-block

- The try-block tells the proceeding catch-block where an exception may be thrown
- If an exception is thrown anywhere inside the try-block **or in any methods called inside the try-block**, execution is immediately passed to the catch-block

```
try
{
}
catch (RuntimeException e)
{
}
```

The Exception object is passed to the catch-block



Simple Demonstration

```
try
{
    if (amount > balance)
    {
        throw new RuntimeException(
            "Amount exceeds balance");
    }

    balance = balance - amount;
}
catch(RuntimeException e)
{
    System.out.println( e.getMessage() );
}
```

The Exception Class

- All exception objects inherit from the **Exception** class
- Many different subclasses of Exception have already been implemented
- Each subclass of Exception is used for catching different types of errors

- Just a few examples:

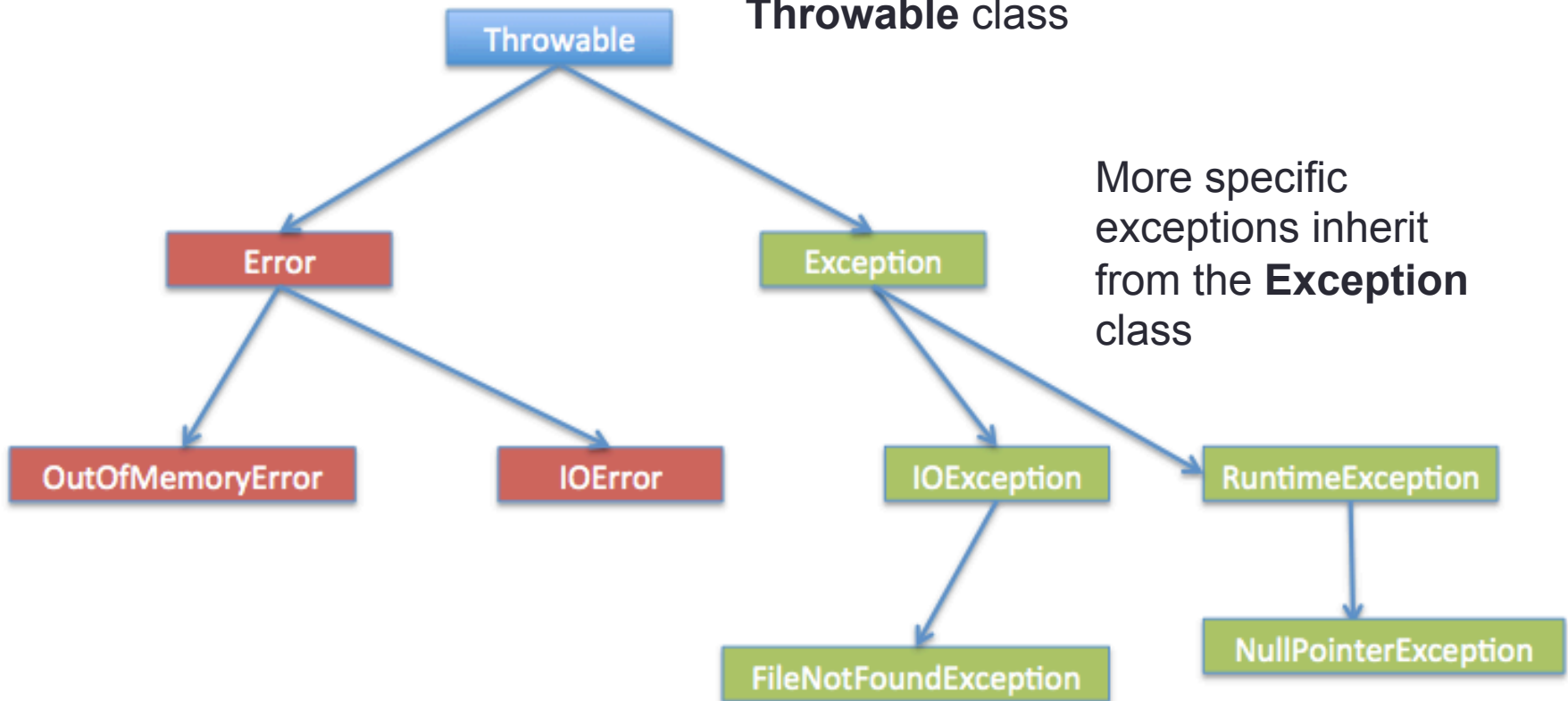
RuntimeException – Signals an ambiguous error

IOException – Signals an error during an I/O operation

IndexOutOfBoundsException – Signals attempted access to an illegal index in an array object

The Exception class in the Inheritance Hierarchy

The **Exception** class inherits from the **Throwable** class



Useful Methods in the **Exception** Class

- **printStackTrace()** – This method prints its stack trace to the **standard error stream**

Remember the stack trace is a String that looks like:

```
Exception in thread "main"  
java.lang.ArrayIndexOutOfBoundsException: 784  
    at Main.makeLowerCase(Main.java:29)  
    at Main.main(Main.java:10)
```


Useful Methods in the **Exception** Class

- **getMessage()** – Gets the error message associated with the Exception (not the entire stack trace). This error message is passed to the Exception object's constructor when it is created:

```
throw new RuntimeException( "Amount exceeds balance" );
```


Error Message

Useful Methods in the **Exception** Class

- Since all exception objects inherit these methods, you can call them in your catch-block:

```
...  
catch(RuntimeException e)  
{  
    e.printStackTrace();  
}
```

Creating your own Exception subclass

- You can design your own exception subclass with its own specific methods you want to call in case of a specific error that might occur in your program
- For example, your exception subclass might contain functionality for sending an error report to your database that will track all errors occurring in your product:




More on catch-blocks

- A single catch block can catch only one type of exception
- For example, if you have a catch-block to catch an IOException, it will not catch a NullPointerException
- Example:

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

The catch-block will not be executed if a NullPointerException is thrown in the preceding try-block




Multiple catch-blocks


- If your try-block can throw multiple types of exceptions, you can proceed your try-block with multiple catch-blocks.
- Each catch-block catches one type of exception
- Example:

```
try
{
    ...
}
catch(IOException e)
{
    ...
}
catch(NullPointerException e)
{
    ...
}
```

This code is executed
As soon as an **IOException**
Object is thrown in the try block



This code is executed
As soon as a
NullPointerException
object is thrown in the try block



The finally-block

- The finally-block denotes code that should ALWAYS be executed regardless of whether an exception occurs or doesn't occur in the try-block
- The finally-block must proceed a try-block

When an exception is NOT thrown in the try-block

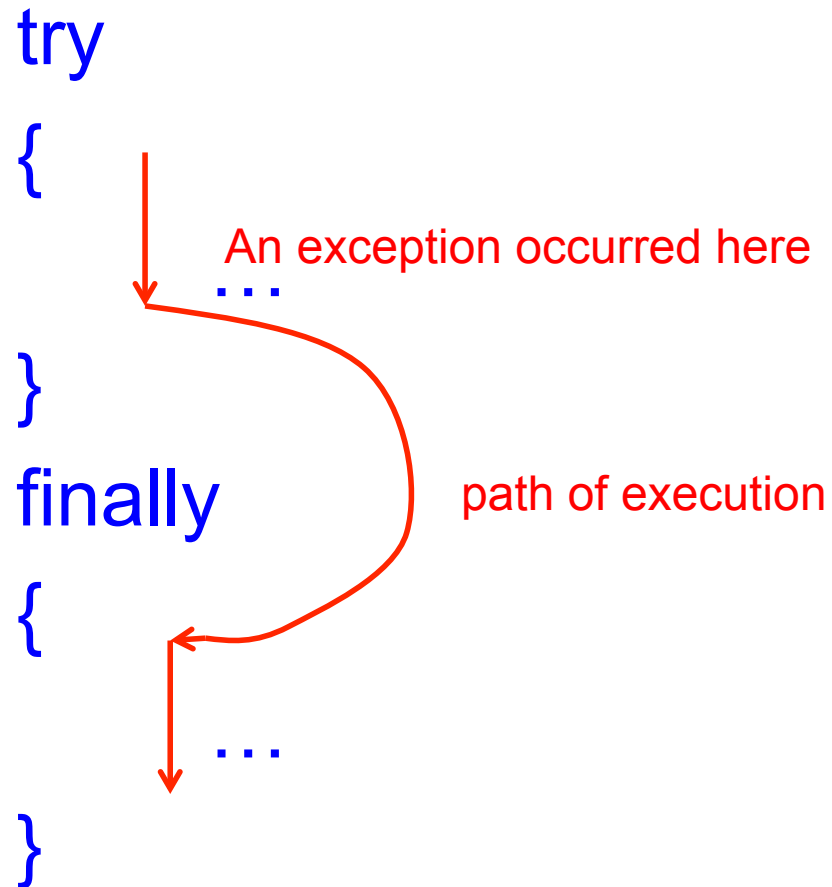
```
try
{
    ...
}
finally
{
    ...
}
```



path of execution

When an exception is thrown in the try-block

NOTE, This code structure does NOT handle the exception that occurs in the try-block. It simply denotes code that should always be executed even when an exception occurs in the try-block



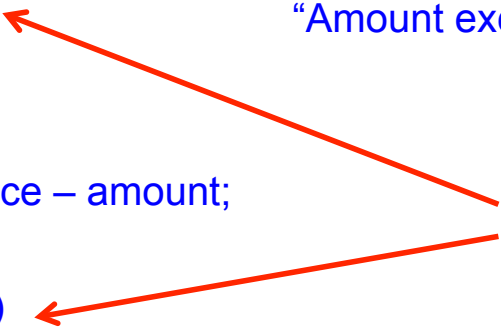
Not handling an exception in the same method it is thrown

- In most cases, Exceptions are NOT handled in the same method that throws them. The following is an UNREALISTIC example:

```
try
{
    if (amount > balance)
    {
        throw new RuntimeException(
            "Amount exceeds balance");
    }

    balance = balance - amount;
}
catch(RuntimeException e)
{
    System.out.println( e.getMessage() );
}
```

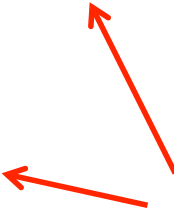
We usually don't throw AND handle an exception thrown in the same method



Not handling an exception in the same method it is thrown

- Instead, we throw the exception in one method and rely on the *calling* method to handle the exception
- Example:

```
try
{
    File myFile = new File("file.txt");
    Scanner scan = new Scanner( myFile );
}
catch(IOException e)
{
    ...
}
```




The Scanner's constructor might throw an exception, however, the Scanner's constructor doesn't catch it. It is up to us, the calling method, to catch that exception

Throwing an exception and not catching it

- If you write a method that might throw an exception, and you decide to NOT handle that exception inside that method, then you are required to tell any method that calls it that this method might throw an exception
- We do this by typing the Java reserved word **throws** in the method header of the method that might throw an exception
- Example:

```
public static void writeToFile(File output) throws IOException  
{  
    ...  
}
```



This denotes that this method might throw an IOException, and therefore, it is the caller's responsibility to handle the exception

Example

```
public static void writeToFile(File output) throws IOException  
{
```

```
    try  
    {
```

```
        PrintWriter writer = new PrintWriter( output );  
        writer.println("hello, world!");
```

```
    }  
    finally  
    {
```

```
        writer.close();
```

```
    }
```

```
}
```

This method
might throw
an IOException

If the writer throws
an exception here,
then we, in turn,
throw the exception to
our caller

If an exception occurs, this code is executed
BEFORE the method terminates and throws
the exception

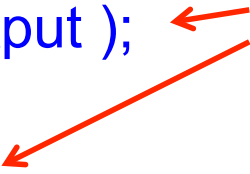
Example Continued

- Let's say we have the following main method that calls the method in the previous slide

```
public static void main(String[] args)
{
    File output = new File ("file.txt");

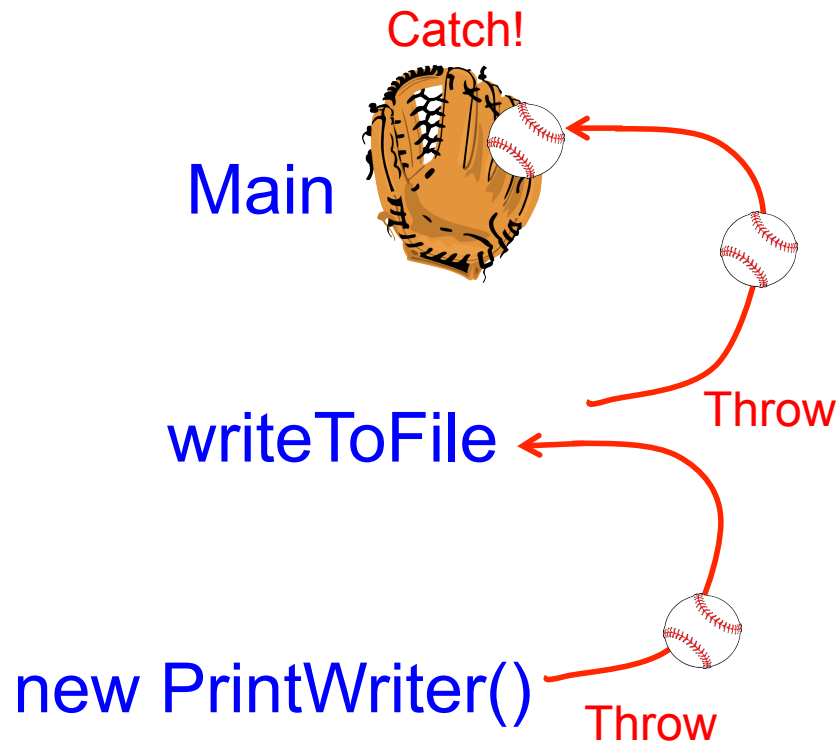
    try
    {
        writeToFile( output );
    }
    catch (IOException e)
    {
        System.out.println(e.getMessage());
    }
}
```

Catch the possible
IOException thrown
in the writeToFile
method



The path of the exception in the previous example

- Let's track the path of the exception in the previous example
- If an exception is thrown, the path of the exception is as follows:



Why Exception Handling?

- Exception handling:
 - provides a flexible mechanism that allows you to separate the code that is used to actually run the program (when things are going smoothly) and the code that is executed when bad things happen
 - Allows communication within your program about the types of errors that occur
 - Allows you to flexibly decide which parts of your program will be responsible for handling an error detected somewhere else within your program

When to use exception handling

- Exception handling is a very useful mechanism provided in many programming languages, but it is often difficult to understand when to use it
- Most beginners tend to overuse exception handling and use it to catch any little error
- So when is it appropriate?
- In general this is a debatable topic

When to use exception handling

- I once read a good explanation from Stack Overflow and it went as follows:
 - My personal guideline is: an exception is thrown when a fundamental assumption of the current code block is found to be false.
 - Example 1: say I have a function which is supposed to examine an arbitrary class and return true if that class inherits from List<>. This function asks the question, "Is this object a descendant of List?" This function should never throw an exception, because there are no gray areas in its operation - every single class either does or does not inherit from List<>, so the answer is always "yes" or "no".
 - Example 2: say I have another function which examines a List<> and returns true if its length is more than 50, and false if the length is less. This function asks the question, "Does this list have more than 50 items?" But this question makes an assumption - it assumes that the object it is given is a list. If I hand it a NULL, then that assumption is false. In that case, if the function returns *either* true *or* false, then it is breaking its own rules. The function cannot return *anything* and claim that it answered the question correctly. So it doesn't return - it throws an exception.

Cool Link

- A research group at Stanford used machine learning algorithms to train a computer how to fly an RC helicopter:
- <http://www.youtube.com/watch?v=VCdxqn0fcnE>

