

## Topic 4

### Exceptions and File I/O

"A slipping gear could let your M203 grenade launcher fire when you least expect it. That would make you quite unpopular in what's left of your unit."

- *THE U.S. Army's PS magazine, August 1993*, quoted in *The Java Programming Language, 3rd edition*

## When Good Programs Go Bad

- ▶ A variety of errors can occur when a program is running. For example:
  - (real) user input error. bad url
  - device errors. remote server unavailable
  - physical limitations. full disk
  - code errors. interact with code that does not fulfill its contract (pre and post conditions)
- ▶ when an error occurs
  - return to safe state, save work, exit gracefully
- ▶ error handling code may be far removed from code that caused the error

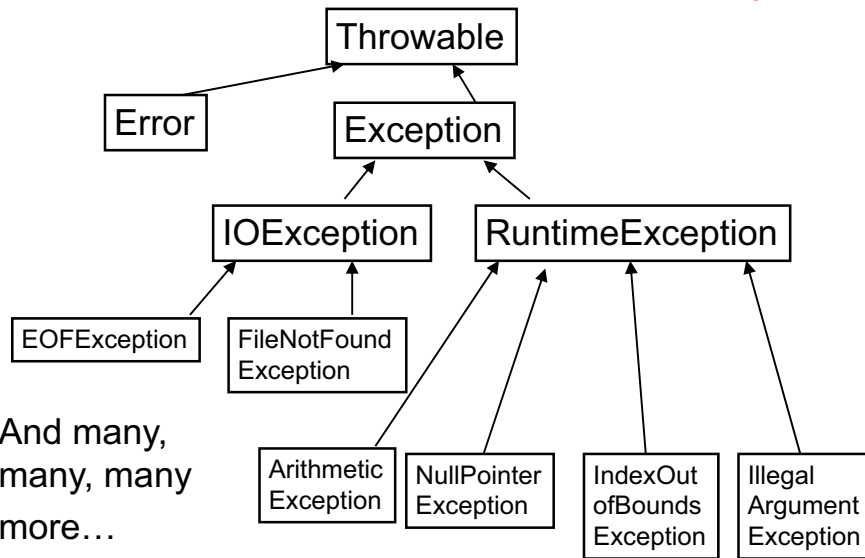
## How to Handle Errors?

- ▶ It is possible to detect and handle errors of various types.
- ▶ Problem: this complicates the code and makes it harder to understand.
  - the error detection and error handling code have little or nothing to do with the *real* code is trying to do.
- ▶ A tradeoff between ensuring correct behavior under all possible circumstances and clarity of the code

## Exceptions

- ▶ Many languages, including Java use a mechanism known as *Exceptions* to handle errors at runtime
  - In Java Exception is a class with many descendants.
  - **ArrayIndexOutOfBoundsException**
  - **NullPointerException**
  - **FileNotFoundException**
  - **ArithmeticException**
  - **IllegalArgumentException**

## Partial Exceptions Hierarchy



CS 307 Fundamentals of  
Computer Science

Java Basics - Exceptions and File I/O

5

## Creating Exceptions

- ▶ As a program runs, if a situation occurs that is handled by exceptions then an Exception is *thrown*.
  - An Exception object of the proper type is created
  - flow of control is transferred from the current block of code to code that can handle or deal with the exception
  - the normal flow of the program stops and error handling code takes over (if it exists.)

CS 307 Fundamentals of  
Computer Science

Java Basics - Exceptions and File I/O

6

## Attendance Question 1

Is it possible for the following method to result in an exception?

```
// pre: word != null
public static void printLength(String word){
    String output = "Word length is " + word.length();
    System.out.println( output );
}
```

- A. Yes
- B. No

CS 307 Fundamentals of  
Computer Science

Java Basics - Exceptions and File I/O

7

## Unchecked Exceptions

- ▶ Exceptions in Java fall into two different categories
  - *checked (other than Runtime) and unchecked (Runtime)*
- ▶ unchecked exceptions are **completely preventable** and should never occur.
  - They are caused by logic errors, created by us, the programmers.
- ▶ Descendents of the RuntimeException class
- ▶ Examples: ArrayIndexOutOfBoundsException, NullPointerException, ArithmeticException
- ▶ There does not *need* to be special error handling code
  - just regular error prevention code
- ▶ If error handling code was required programs would be unwieldy because so many Java statements have the possibility of generating or causing an unchecked Exception

CS 307 Fundamentals of  
Computer Science

Java Basics - Exceptions and File I/O

8

## Checked Exceptions

- ▶ "Checked exceptions represent conditions that, although exceptional, can reasonably be expected to occur, and if they do occur must be dealt with in some way.[other than the program terminating.]"
  - *Java Programming Language third edition*
- ▶ Unchecked exceptions are due to a programming logic error, our fault and preventable if coded correctly.
- ▶ Checked exceptions represent errors that are unpreventable by us!

## Required Error Handling Code

- ▶ If you call a method that can generate a checked exception you must choose how to deal with that possible error
- ▶ For example one class for reading from files is the `FileReader` class

```
public FileReader (String fileName)  
    throws FileNotFoundException
```

- ▶ This constructor has the possibility of throwing a `FileNotFoundException`
- ▶ `FileNotFoundException` is a checked exception

## Checked Exceptions in Code

- ▶ If we have code that tries to build a `FileReader` we must deal with the possibility of the exception

```
import java.io.FileReader;  
  
public class Tester  
{  
    public int countChars(String fileName)  
    {  
        FileReader r = new FileReader(fileName);  
        int total = 0;  
        while( r.ready() )  
        {  
            r.read();  
            total++;  
        }  
        r.close();  
        return total;  
    }  
}
```

- ▶ The code contains a syntax error. "unreported exception `java.io.FileNotFoundException`; must be caught or declared to be thrown."

## Handling Checked Exceptions

- ▶ In the code on the previous slide there are in fact 4 statements that can generate checked exceptions.
  - The `FileReader` constructor
  - the `ready` method
  - the `read` method
  - the `close` method
- ▶ To deal with the exceptions we can either state this method throws an Exception of the proper type or handle the exception within the method itself

## Methods that throw Exceptions

- ▶ It may be that we don't know how to deal with an error within the method that can generate it
- ▶ In this case we will pass the buck to the method that called us
- ▶ The keyword `throws` is used to indicate a method has the possibility of generating an exception of the stated type
- ▶ Now any method calling ours must also throw an exception or handle it

## Using the `throws` Keyword

```
public int countChars(String fileName)
    throws FileNotFoundException, IOException
{
    int total = 0;
    FileReader r = new FileReader(fileName);
    while( r.ready() )
    {
        r.read();
        total++;
    }
    r.close();
    return total;
}
```

- ▶ Now any method calling ours must also throw an exception or handle it

## Using `try-catch` Blocks

- ▶ If you want to handle the a checked exception locally then use use the keywords `try` and `catch`
- ▶ the code that could cause an exception is placed in a block of code preceded by the keyword `try`
- ▶ the code that will handle the exception if it occurs is placed in a block of code preceded by the keyword `catch`

## Sample `try` and `catch` Blocks

```
public int countChars(String fileName)
{
    int total = 0;
    try
    {
        FileReader r = new FileReader(fileName);
        while( r.ready() )
        {
            r.read();
            total++;
        }
        r.close();
    }
    catch(FileNotFoundException e)
    {
        System.out.println("File named "
            + fileName + "not found. " + e);
        total = -1;
    }
    catch(IOException e)
    {
        System.out.println("IOException occurred " +
            "while counting chars. " + e);
        total = -1;
    }
    return total;
}
```

## Mechanics of try and catch

- Code that could cause the checked exception is placed in a try block
  - note how the statements are included in one try block.
  - Each statement could be in a separate try block with an associated catch block, but that is very unwieldy (see next slide)
- Each try block must have 1 or more associated catch blocks
  - code here to handle the error. In this case we just print out the error and set result to -1

## Gacky try catch Block

```
public int countChars3(String fileName)
{
    int total = 0;
    FileReader r = null;
    try
    {
        r = new FileReader(fileName);
    }
    catch(FileNotFoundException e)
    {
        System.out.println("File named "
            + fileName + "not found. " + e);
        total = -1;
    }

    try
    {
        while( r.ready() )
        {
            try
            {
                r.read();
            }
            catch(IOException e)
            {
                System.out.println("IOException "
                    + "occurred while counting "
                    + "chars. " + e);
                total = -1;
            }
            total++;
        }
    }
    catch(IOException e)
    {
        System.out.println("IOException occurred while counting chars. " + e);
        total = -1;
    }

    try
    {
        r.close();
    }
    catch(IOException e)
    {
        System.out.println("IOException occurred while counting chars. " + e);
        total = -1;
    }

    return total;
}
```

## More try catch Mechanics

- If you decide to handle the possible exception locally in a method with the try block you must have a corresponding catch block
- the catch blocks have a parameter list of 1
- the parameter must be Exception or a descendant of Exception
- Use multiple catch blocks with one try block in case of multiple types of Exceptions

## What Happens When Exceptions Occur

- If an exception is thrown then the normal flow of control of a program halts
- Instead of executing the regular statements the Java Runtime System starts to search for a matching catch block
- The first matching catch block based on data type is executed
- When the catch block code is completed the program does not "go back" to where the exception occurred.
  - It finds the next regular statement after the catch block

## Counting Chars Again

```
public int countChars(String fileName)
{
    int total = 0;
    try
    {
        FileReader r = new FileReader(fileName);
        while( r.ready() )
        {
            r.read(); // what happens in an exception occurs?
            total++;
        }
        r.close();
    }
    catch(FileNotFoundException e)
    {
        System.out.println("File named "
            + fileName + "not found. " + e);
        total = -1;
    }
    catch(IOException e)
    {
        System.out.println("IOException occurred " +
            "while counting chars. " + e);
        total = -1;
    }
    return total;
}
```

## Throwing Exceptions Yourself

- if you wish to throw an exception in your code you use the `throw` keyword
- Most common would be for an unmet precondition

```
public class Circle
{
    private int iMyRadius;

    /** pre: radius > 0
     */
    public Circle(int radius)
    {
        if (radius <= 0)
            throw new IllegalArgumentException
                ("radius must be > 0. "
                 + "Value of radius: " + radius);
        iMyRadius = radius;
    }
}
```

## Attendance Question 2

What is output by the method `badUse` if it is called with the following code?

```
int[] nums = {3, 2, 6, 1};
badUse( nums );
```

```
public static void badUse(int[] vals){
    int total = 0;
    try{
        for(int i = 0; i < vals.length; i++){
            int index = vals[i];
            total += vals[index];
        }
    }
    catch(Exception e){
        total = -1;
    }
    System.out.println(total);
}
```

- A. 1      B. 0      C. 3      D. -1      E. 5

## Attendance Question 3

Is the use of a try-catch block on the previous question a proper use of try-catch blocks?

- A. Yes  
B. No

## Error Handling, Error Handling Everywhere!

- Seems like a lot of choices for error prevention and error handling
  - normal program logic, e.g. if's for loop counters
  - assertions
  - try – catch block
- When is it appropriate to use each kind?

## Error Prevention

- Us program logic, (ifs , fors) to prevent logic errors and unchecked exceptions
  - dereferencing a null pointer, going outside the bounds of an array, violating the preconditions of a method you are calling. e.g. the `charAt` method of the `String` class
  - use assertions as checks on your logic
    - you checked to ensure the variable `index` was within the array bounds with an if 10 lines up in the program and you are SURE you didn't alter it.
    - Use an `assert` right before you actually access the array

```
if( inbounds(index) )
{ // lots of related code
  // use an assertion before accessing
  arrayVar[index] = foo;
}
```

## Error Prevention

- in 307 asserts can be used to check preconditions
  - Standard Java style is to use Exceptions
- Use try/catch blocks on checked exceptions
  - In general, don't use them to handle unchecked exceptions like NPE or AIOBE
- One place it is reasonable to use try / catch is in testing suites.
  - put each test in a try / catch. If an exception occurs that test fails, but other tests can still be run

## File Input and Output

- Programs must often read from and write to files
  - large amounts of data
  - data not known at runtime
  - data that changes over time
- Each programming language has its own way of handling input and output
  - involves dealing with the operating system
  - if possible try to hide that fact as much as possible
- Java attempts to standardize input and output with the notion of a *stream*, an ordered sequence of data that has a source or destination.

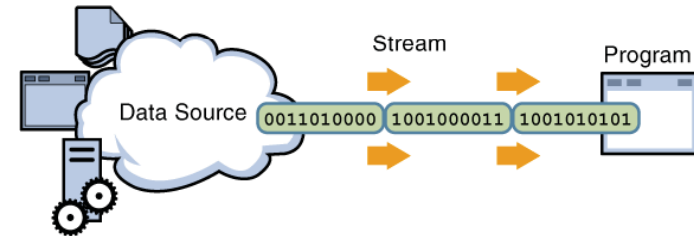


## Streams?

Dr. Egon Spengler: Don't cross the streams.  
Dr. Peter Venkman: Why not?  
Dr. Egon Spengler: It would be bad.  
Dr. Peter Venkman: I'm fuzzy on the whole good/bad thing. what do you mean by "bad"?  
Dr. Egon Spengler: Try to imagine all life as you know it stopping instantaneously and every molecule in your body exploding at the speed of light.  
Dr. Peter Venkman: That's bad. Okay. Alright, important safety tip. Thanks Egon.

## Streams

- ▶ A stream serves as a connection between your program and an external source or destination for bytes and bits
  - could be standard input or output, files, network connections, or other programs



## Lots of Streams

- ▶ Java has over sixty (60!) different stream types in its java.io package
- ▶ part of the reason input and output are so difficult to understand in Java is the size and diversity of the IO library
- ▶ The type of stream you use depends on what you are trying to do
  - even then there are multiple options

## Working with Files in Java

- ▶ Data is stored in digital form, 1s and 0s
- ▶ Work with these in packages of 8, the byte
- ▶ The IO library creates higher level abstractions so we think we are working with characters, Strings, or whole objects
- ▶ Some abstract classes
  - InputStream, OutputStream, Reader, and Writer
  - InputStream and OutputStream represent the flow of data (a stream)
  - Reader and Writer are used to read the data from a stream or put the data in a stream
  - *convenience classes* exist to make things a little easier



## The Scanner class

- A class to make reading from source of input easier. New in Java 5.0
- Constructors
  - Scanner(InputStream)
  - Scanner(File)
- Methods to read lines from input
  - boolean hasNextLine()
  - String nextLine()
- Methods to read ints
  - int readInt(), boolean hasNext()

## Scanner and Keyboard Input

- No exceptions thrown!
  - no try – catch block necessary!
- Set delimiters with regular expressions, default is whitespace

```
Scanner s = new Scanner(System.in);
```

```
System.out.print("Enter your name: ");  
String name = s.nextLine();
```

```
System.out.print("Press Enter to continue: ");  
s.nextLine();
```

## Hooking a Scanner up to a File

```
import java.util.Scanner;  
import java.io.File;  
import java.io.IOException;  
  
public class ReadAndPrintScores  
{  
    public static void main(String[] args)  
    {  
        try  
        {  
            Scanner s = new Scanner( new File("scores.dat") );  
            while( s.hasNextInt() )  
            {  
                System.out.println( s.nextInt() );  
            }  
            s.close();  
        }  
        catch(IOException e)  
        {  
            System.out.println( e );  
        }  
    }  
}
```

```
12 35 12  
12 45  
12  
  
12  
  
13 57
```

scores.dat

## Writing to a File

//sample code to write 100 random ints to a file, 1 per line

```
import java.io.PrintStream;  
import java.io.IOException;  
import java.io.File;  
  
import java.util.Random;  
  
public class WriteToFile  
{  
    public static void main(String[] args)  
    {  
        try  
        {  
            PrintStream writer = new PrintStream( new  
                File("randInts.txt"));  
            Random r = new Random();  
            final int LIMIT = 100;  
  
            for(int i = 0; i < LIMIT; i++)  
            {  
                writer.println( r.nextInt() );  
            }  
            writer.close();  
        }  
        catch(IOException e)  
        {  
            System.out.println("An error occurred " +  
                + "while trying to write to the file");  
        }  
    }  
}
```

# Reading From a Web Page

```
public static void main(String[] args) {
    try {
        String siteUrl = "http://www.cs.utexas.edu/~scottm/cs307";
        URL mySite = new URL(siteUrl);
        URLConnection yc = mySite.openConnection();
        Scanner in =
            new Scanner(new InputStreamReader(yc.getInputStream()));
        int count = 0;
        while (in.hasNext()) {
            System.out.println(in.next());
            count++;
        }
        System.out.println("Number of tokens: " + count);
        in.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

CS 307 Fundamentals of  
Computer Science