

# File Input and Output and Exception Handling

# **FILE INPUT AND OUTPUT**

# Data

- Data in variables and arrays is temporary.
  - When your program ends, the data is lost.
- If you want to retain data beyond the running of your program, you need to use files (or a database, etc.).

# Streams

- A *stream* is an object that enables data to flow between a program and a file or other I/O device.
  - Data going from the program into a file is called an *output stream*.
  - Data coming from a file into a program is called an *input stream*.

# Streams

- A *stream* is an object that enables data to flow between I/O devices.
  - Data going to an output device is called an *output stream*.
  - Data coming from an input device is called an *input stream*.

## Important Note!

You might hear the term “streams” referring to the changes made to Java 8. This is a very different idea than the I/O streams we are covering. (It’s also very cool stuff for another day!)

# Streams

- Data can come into the program from the keyboard or from a file.
  - `System.in` is an input stream that connects to the keyboard.
- Data can go out of the program to the screen or to a file.
  - `System.out` is an output stream that connects to the screen.

# Files

- A text file contains data that both a computer and a human can read.
  - Also called ASCII files.
  - Usually the same on different computers.
- A binary file contains binary digits only and can **only** be read by a computer (not a human).
  - Binary files are more efficient to process but are language/platform dependent.
- The .class file is a *Java binary file* that is platform independent.

# Writing to a Text File

- The `PrintWriter` class can be used to write to a text file.
  - Methods: `print` and `println`
  - Same functionality as `System.out.print` and `System.out.println` but to a file instead of the screen.
- One very nice benefit is that these methods call `toString()` automatically!



# Writing to a Text File

- To create a `PrintWriter` object, we pass a `FileOutputStream` object to the constructor
- To create a `FileOutputStream` object, we pass the file name (as a `String`) to the constructor.

```
PrintWriter out =  
    new PrintWriter(  
        new FileOutputStream(  
            "filename.txt"));
```

# Writing to a Text File

- There are other ways to create the output stream, including the one below.

```
PrintWriter out =  
    new PrintWriter(  
        new BufferedWriter(  
            new FileWriter(  
                "filename.txt"))));
```

# Writing to a Text File

- Different setups differ in minor ways, often related to efficiency of writes.
- Each of the Java I/O classes has a very specific function. So normally you need two or more constructors combined (nested) to get the functionality you want.

# Writing to a Text File

- When you write to a file, if the file does not exist, a new, empty file is created.
- If the file already exists, all of the old contents will be lost.
  - **IMPORTANT!** Writing to an existing file replaces the contents of the file!

# Writing to a Text File

- When a file is opened in these ways, a `FileNotFoundException` can be thrown (e.g., because the file cannot be created).
- So we must enclose the code in an *exception handling block*.
  - More to come on this!
  - For now, we'll just "ignore" the exceptions and focus on the file functionality.

# Writing to a Text File

- Closing the output stream ensures that all data is written to the file:
  - `out.close();`
- **IMPORTANT:** You must close the writer in order for it to accurately write to the file!

# Writing to a Text File

- You can append data to the end of a file, rather than overwriting it.
- To do this, send `true` into the `FileOutputStream` constructor as the second parameter.

```
PrintWriter out =  
    new PrintWriter(  
        new FileOutputStream(  
            "filename.txt", true));
```

# Reading Text from a File

- The Scanner class can be used to read from a text file.
  - Methods: `nextLine()`, `nextInt()`, etc.
  - Same functionality as reading from the user but to a file instead of from the keyboard.



# Reading Text from a File

- To create a Scanner object, pass a FileInputStream object to the constructor.
- To create a FileInputStream object, pass the file name (as a String) to the constructor.

```
Scanner fileScan =  
    new Scanner(  
        new FileInputStream(  
            "filename.txt"));
```

# Reading Text from a File

- You cannot read beyond the end of a file. So you need to make sure there is more to be read before you read.
  - Use a while loop!
  - Methods: hasNextLine, hasNextInt, etc.

# Reading Text from a File

```
while(fileScan.hasNext()) {  
    String line  = fileScan.nextLine();  
    System.out.println(line);  
    // do something with the line  
}  
fileScan.close();
```

# Reading Text from a File

- Just like writing to a file, you need to account for `FileNotFoundException`.
  - Again, more on this to come!

# Reading Text from a File

- There are other ways to create an input stream, including the one below.

```
BufferedReader fileRead =  
    new BufferedReader(  
        new FileReader(  
            "filename.txt"));
```

# The File Class

- The File class can be used as a wrapper for a file name.
  - Send the name of the file to the constructor.
- Methods:
  - exists()
  - canRead()
  - canWrite()
  - delete()
  - isFile()
  - isDirectory()

# The File Class

- Create a file object to avoid using the String file name in multiple places.
- Create a file object if you need to invoke methods to get information about the file.

# The File Class

```
PrintWriter out =  
    new PrintWriter(  
        new FileOutputStream(  
            new File(  
                "filename.txt"))));
```



# The File Class

```
Scanner fileScan =  
    new Scanner(  
        new FileInputStream(  
            new File(  
                "filename.txt"))));
```

# The File Class

```
File file = new File("filename.txt");
```

```
Scanner fileScan = new Scanner(  
    new FileInputStream(file);
```

```
PrintWriter out = new PrintWriter(  
    new FileOutputStream(file));
```

# Practice

- Read the contents of a file with one-word-per-line and write the contents out to a different file with all words on the same line (separated by a space).
  - For now, ignore exceptions.

# Parsing

- The Scanner class can also be used to break down input based on white space or other delimiters that you choose.

```
String line = fileScan.nextLine();
Scanner lineScan = new Scanner(line);
lineScan.useDelimiter(",");
while(lineScan.hasNext()) {
    String s = lineScan.next();
}
```

# Practice

- Review the user-data file created in Excel (and saved as a csv file). Write a program to read in the data and create objects from the data.

# **JAVA 7 FILE MANAGEMENT**

# File Management

- As of Java 7, there is a `java.nio.file` package that allows us to manage files: deleting, renaming, seeing when last modified, etc.
- Some of the classes introduced:
  - `Paths`
  - `Files`

# Path Names

- When you send in a file name to a constructor, it assumes the file is in the same directory or folder where the program is.
- If it's not, you need to give more information.



# Path Names

- A *path name* gives the directory or folder where the file exists and (optionally) the name of the file.
- A *full path name* gives a complete path from the root directory.
- A *relative path name* gives the path to the file starting in the directory where the program is located.

# Path Names

- Path names differ depending on the operating system.
  - Example: UNIX: `"/user/jmasters/data/file.txt"`
  - Example: Windows: `"C:\Users\Jessica\Files\file.txt"`
- Windows Note!
  - You must use `\\` in place of `\` in hard-coded paths because `\` denotes an escape sequences:
  - `"C:\\Users\\Jessica\\Files\\file.txt"`
- Java will accept a path name written in either format regardless of the operating system on which the program is run.

# The Path Class

- You can also use the Path class to store a path.
- You can use the Path`s` class to get the Path object.
- Example:

```
Path myPath = Paths.get("location");
```

# The Path Class

```
Path myPath = Paths.get("location");
```

- The "location" can be a single String or multiple Strings that each list a director in the path.
- Example:

```
Path myPath =
```

```
    Paths.get("home", "file", "java");
```

# The Path Class

- `toString()`
- `getFileName()`
- `getName(int)`
  - Returns the path element corresponding to the number
  - 0 is the element closest to the root
- `getNameCount()`
  - Returns the number of elements in the path
- `getParent()`
- `getRoot()`

# The FileS Class

- FileS provides static methods for working with files and directories.
- Methods work on Path objects.
- Static Methods:
  - exists()
  - notExists()
  - isReadable()
  - isWritable()
  - isDirectory()
  - copy()
  - move()
  - delete()

# The FileS Class

- Files can also read and write from text files:

```
List<String> lines =  
    FileS.readAllLines(path);
```

```
FileS.write(newPath, lines);
```

# The FileS Class

- Can use enums to append:

```
FileS.write(newPath, lines,  
            StandardOpenOption.APPEND);
```



# The FileS Class

- Can use enums to replace:

```
FileS.copy(old path, new path,  
           StandardCopyOption.REPLACE_EXISTING);
```

# Practice

- Review the program that copies a user-specified file to a new directory.
- Review the program to read a file that contains the following and updates the files accordingly.
  - a list of other file names
  - true/false- true means append, false means overwrite
  - data to either append or write

# **SERIALIZATION**

# Object Serialization

- To *serialize* an object means to convert it to a byte stream so that it can be stored in a file.
- *Deserializing* converts back from the byte representation to the object representation.
  - A serialized object contains information about the instance data variables- including information about their values **and** type.

# Making Your Object Serializable

- Implement `java.io.Serializable`
- Make sure all of your instance data is serializable too
  - Primitives are automatically serializable
  - Check whether objects are
- That's it!

# To Output/Input Serialized Objects

- `ObjectInputStream`
  - `readObject` and cast
    - checked exception: `ClassNotFoundException`
- `ObjectOutputStream`
  - `writeObject`

# Output Serialized Objects

- Use `ObjectOutputStream` and `writeObject`

```
ObjectOutputStream out = new ObjectOutputStream(  
    new FileOutputStream("object.ser"));  
out.writeObject(myObj);  
out.close();
```

# Input Serialized Objects

- Use `ObjectInputStream` and `readObject` (with a cast)
  - checked exception: `ClassNotFoundException`

```
try {
    ObjectInputStream in = new ObjectInputStream(
        new FileInputStream("object.ser"));
    Student myStudent = (Student) in.readObject();
    in.close();
} catch (ClassNotFoundException ex) {
    ex.printStackTrace();
}
```



# Practice

- Use the Student class. Write a serialized Student object to a file and read it in from a file.

# EXCEPTION HANDLING

# Exception Handling

- We expect our code to run in a certain way.
- Exception handling is how we can deal with unusual, erroneous, or unexpected situations.

# Exception Handling

- When something unusual happens, we say that *an exception is thrown* or that the code *throws an exception*.
- We need to account for these things happen. We need to deal with exceptions that are thrown. This is called *handling the exception*.

# Exception Handling

- If an exception is thrown and not handled, the program will crash.

# Practice

- Write code to read in a positive number from the user.

# TRY-CATCH

# try-catch Blocks

- The basic way to handle exceptions is with a *try-catch* block.
- The **try** block contains the code for your algorithm.
  - This is the code that will run if everything goes smoothly and according to plan.
- Somewhere in the **try** block, there is also code that might throw an exception if something unplanned happens.



# try-catch Blocks

- When a statement inside the **try** block throws an exception, the execution of the program stops.
- The JVM now looks for some code that tells it what to do in response to the exception- how to handle the exception.
- That code is placed inside of a **catch** block.

# try-catch Block Syntax

```
try {  
    // code to execute when all goes well;  
    // this code might also throw an exception  
} catch (ExceptionType ex) {  
    // code describing how to react or handle  
    // exceptions of type ExceptionType  
}
```

# try-catch Block Syntax

```
try {  
    // code to execute when all goes well;  
    // this code might also throw an exception  
} catch (ExceptionType ex) {  
    // code describing how to react or handle  
    // exceptions of type ExceptionType  
}
```

reserved words



# try-catch Block Syntax

```
try {  
    // code to execute when all goes well;  
    // this code might also throw an exception  
} catch (ExceptionType ex) {  
    // code describing how to react or handle  
    // exceptions of type ExceptionType  
}
```

catch-block parameter: the exception object



# Exceptions are Objects!

- When the unusual event happens, an exception is thrown.
- That exception is an object!
  - It has a data type.
- The exception object has information about what went wrong:
  - the type of event
  - a message
  - information about where the unusual event occurred

# Exceptions are Objects!

- When an exception is thrown, it's as if this code is executed:

```
throw new ExceptionType(messageString)
```

- The throw keyword looks for catch blocks to handle an exception of that type.

# The Exception Object

- Exception objects have a String instance data variable that contains a message.
  - Usually describes the reason for the exception.
  - Accessed with `ex.getMessage();`
- You can also see a stack trace of how you got to the statement where the exception occurred: the *call stack trace*
  - Accessed with `ex.printStackTrace();`
  - This is best for debugging, not for showing to a user.

# try-catch Block Syntax

```
try {  
    // code to execute when all goes well;  
    // this code might also throw an exception  
} catch (ExceptionType ex) {  
    // code describing how to react or handle  
    // exceptions of type ExceptionType  
}
```

this code inside the try block brackets  
is the *normal execution flow*



# try-catch Block Syntax

```
try {  
    // code to execute when all goes well;  
    // this code might also throw an exception  
} catch (ExceptionType ex) {  
    // code describing how to react or handle  
    // exceptions of type ExceptionType  
}
```

if a statement in this block throws an exception, the execution stops and control is passed to the catch block

# try-catch Block Syntax

```
try {  
    // code to execute when all goes well;  
    // this code might also throw an exception  
} catch (ExceptionType ex) {  
    // code describing how to react or handle  
    // exceptions of type ExceptionType  
}
```

if the exception was type `ExceptionType`,  
the code inside the catch block brackets  
is executed

# try-catch Block Syntax

```
try {  
    // code to execute when all goes well;  
    // this code might also throw an exception  
} catch (ExceptionType ex) {  
    // code describing how to react or handle  
    // exceptions of type ExceptionType  
}
```

this code is only executed if the type of the exception is ExceptionType

the actual object (ex) is passed to the catch block so it can be used- we can invoke methods on it

# try-catch Block Syntax

```
try {  
    // code to execute when all goes well;  
    // this code might also throw an exception  
} catch (ExceptionType ex) {  
    // code describing how to react or handle  
    // exceptions of type ExceptionType  
}
```

// code after the try-catch



after the catch block code is done,  
we resume **after the try-catch** block

# try-catch Block Syntax

```
try {  
    // code to execute when all goes well;  
    // this code might also throw an exception  
} catch (ExceptionType ex) {  
    // code describing how to react or handle  
    // exceptions of type ExceptionType  
}  
// code after the try-catch
```



Important!!! We do **NOT** return to the try block! We move on to the code **after** the try-catch.

# Practice

- Update the positive number program to use a try-catch block. If the user enters a non-number, provide a message before the program ends *gracefully*.

# What type of exception to catch?

- How do you know what type of exception to put in your catch block?
- You can run the program and have it crash!
- Or you can look at the API page.
  - Example: Java Integer API (parseInt method)

# Tracing Exception Handling Code

- Situation 1: No exception is thrown in the try block.
  - The code in the try block is executed.
  - The catch block is skipped.
  - Execution continues with the code after the catch block.



# Tracing Exception Handling Code

- No exception is thrown in the try block.

*code before the try-catch block*

```
try {
```

*code in the try block*

```
} catch (ExceptionA ex) {
```

*code in the catch block*

```
}
```

*code after the try-catch block*

# Tracing Exception Handling Code

- No exception is thrown in the try block.

*code before the try-catch block*

```
try {
```

```
    code in the try block
```

```
} catch (ExceptionA ex) {
```

```
    code in the catch block
```

```
}
```

*code after the try-catch block*

code before the  
try-catch block  
executes

# Tracing Exception Handling Code

- No exception is thrown in the try block.

*code before the try-catch block*

```
try {
```

*code in the try block*

```
} catch (ExceptionA ex) {
```

*code in the catch block*

```
}
```

*code after the try-catch block*

code inside the try  
block executes

no exception is  
thrown

# Tracing Exception Handling Code

- No exception is thrown in the try block.

*code before the try-catch block*

```
try {
```

*code in the try block*

```
} catch (ExceptionA ex) {
```

*code in the catch block*

```
}
```

*code after the try-catch block*

the catch block  
is skipped

control resumes  
with the code after  
the try-catch block

# Tracing Exception Handling Code

- No exception is thrown in the try block.

*code before the try-catch block*

```
try {
```

*code in the try block*

```
} catch (ExceptionA ex) {
```

~~code in the catch block~~

```
}
```

*code after the try-catch block*

# Tracing Exception Handling Code

- Situation 2: An exception is thrown and caught.
  - The code in the try block is executed up until the exception is thrown.
  - The rest of the try block code is skipped.
  - Control transfers to the catch block.
  - The code in the catch block is executed.
  - We continue on after the try-catch block.
    - We never return back up to the try.

# Tracing Exception Handling Code

- An exception is thrown and caught.

*code before the try-catch block*

```
try {
```

*code before an exception is thrown*

*code that throws an object of type ExceptionA*

*code after statement that might throw the exception*

```
} catch (ExceptionA ex) {
```

*code in the catch block*

```
}
```

*code after the try-catch block*

# Tracing Exception Handling Code

- An exception is thrown and caught.

*code before the try-catch block*

```
try {
```

*code before an exception is thrown*

*code that throws an object of type ExceptionA*

*code after statement that might throw the exception*

```
} catch (ExceptionA ex) {
```

*code in the catch block*

```
}
```

*code after the try-catch block*

code before the  
try-catch block  
executes



# Tracing Exception Handling Code

- An exception is thrown and caught.

*code before the try-catch block*

```
try {
```

*code before an exception is thrown*

*code that throws an object of type ExceptionA*

*code after statement that might throw the exception*

```
} catch (ExceptionA ex) {
```

*code in the catch block*

```
}
```

*code after the try-catch block*

code in the try  
block begins to  
execute

# Tracing Exception Handling Code

- An exception is thrown and caught.

*code before the try-catch block*

```
try {
```

*code before an exception is thrown*

*code that throws an object of type `ExceptionA`*

*code after statement that might throw the exception*

```
} catch (ExceptionA ex) {
```

*code in the catch block*

```
}
```

*code after the try-catch block*

we reach the  
statement that  
throws the  
exception of type  
`ExceptionA`

# Tracing Exception Handling Code

- An exception is thrown and caught.

*code before the try-catch block*

```
try {
```

*code before an exception is thrown*

*code that throws an object of type `ExceptionA`*

*code after statement that might throw the exception*

```
} catch (ExceptionA ex) {
```

*code in the catch block*

```
}
```

*code after the try-catch block*

control jumps to the  
matching catch  
block and executes  
the code in the  
catch block

# Tracing Exception Handling Code

- An exception is thrown and caught.

*code before the try-catch block*

```
try {
```

*code before an exception is thrown*

*code that throws an object of type ExceptionA*

*code after statement that might throw the exception*

```
} catch (ExceptionA ex) {
```

*code in the catch block*

```
}
```

*code after the try-catch block*

control resumes with  
the code after the  
try-catch block

we do **NOT** return to  
the try block!

# Tracing Exception Handling Code

- An exception is thrown and caught.

*code before the try-catch block*

```
try {
```

*code before an exception is thrown*

*code that throws an object of type `ExceptionA`*

~~*code after statement that might throw the exception*~~

```
} catch (ExceptionA ex) {
```

*code in the catch block*

```
}
```

*code after the try-catch block*

# Practice

- Modify the code to use `nextInt()` instead of `nextLine()`. What happens?

# Tracing Exception Handling Code

- Situation 3: An exception is thrown in the try block and cannot be caught by the catch block.
  - The code in the try block is executed up until the exception is thrown.
  - The rest of the try block code is skipped.
  - Control transfers to the catch block.
  - If the exception type is **not** a match for that block, the code in the catch block is skipped.
  - Control returns to the previous method and we again look for a matching catch block.
  - If no matching catch is ever found, the program crashes.

# Tracing Exception Handling Code

- An exception is thrown and **not** caught.

*code before the try-catch block*

```
try {
```

*code before an exception is thrown*

*code that throws an object of type `ExceptionB`*

*code after statement that might throw the exception*

```
} catch (ExceptionA ex) {
```

*code in the catch block*

```
}
```

*code after the try-catch block*



# Tracing Exception Handling Code

- An exception is thrown and **not** caught.

*code before the try-catch block*

```
try {
```

*code before an exception is thrown*

*code that throws an object of type `ExceptionB`*

*code after statement that might throw the exception*

```
} catch (ExceptionA ex) {
```

*code in the catch block*

```
}
```

*code after the try-catch block*

code before the  
try-catch block  
executes

# Tracing Exception Handling Code

- An exception is thrown and **not** caught.

*code before the try-catch block*

```
try {
```

code in the try  
block begins to  
execute

*code before an exception is thrown*

*code that throws an object of type `ExceptionB`*

*code after statement that might throw the exception*

```
} catch (ExceptionA ex) {
```

*code in the catch block*

```
}
```

*code after the try-catch block*

# Tracing Exception Handling Code

- An exception is thrown and **not** caught.

*code before the try-catch block*

```
try {
```

*code before an exception is thrown*

*code that throws an object of type `ExceptionB`*

*code after statement that might throw the exception*

```
} catch (ExceptionA ex) {
```

*code in the catch block*

```
}
```

*code after the try-catch block*

we reach the  
statement that  
throws the  
exception of type  
`ExceptionB`

# Tracing Exception Handling Code

- An exception is thrown and **not** caught.

*code before the try-catch block*

```
try {
```

*code before an exception is thrown*

*code that throws an object of type `ExceptionB`*

*code after statement that might throw the exception*

```
} catch (ExceptionA ex) {
```

*code in the catch block*

```
}
```

*code after the try-catch block*

we check the catch  
block, but it's not for  
this kind of  
exception

# Tracing Exception Handling Code

- An exception is thrown and **not** caught.

*code before the try-catch block*

```
try {
```

*code before an exception is thrown*

*code that throws an object of type `ExceptionB`*

*code after statement that might throw the exception*

```
} catch (ExceptionA ex) {
```

*code in the catch block*

```
}
```

we leave!

*code after the try-catch block*

# Tracing Exception Handling Code

- An exception is thrown and **not** caught.

*code before the try-catch block*

```
try {
```

*code before an exception is thrown*

*code that throws an object of type `ExceptionB`*

*code after statement that might throw the exception*

```
} catch (ExceptionA ex) {
```

*code in the catch block*

```
}
```

*code after the try-catch block*

*... Leave the method! and don't come back!*

*we return to the method that invoked this code and search for a matching catch block in that method*

# Tracing Exception Handling Code

- An exception is thrown and **not** caught.

*code before the try-catch block*

```
try {
```

*code before an exception is thrown*

*code that throws an object of type `ExceptionB`*

*code after statement that might throw the exception*

```
} catch (ExceptionA ex) {
```

*code in the catch block*

```
}
```

*code after the try-catch block*

*... Leave the method! and don't come back!*

*we return to the method that invoked this code and  
search for a matching catch block in that method*

*if no match is ever found, the program crashes*

# Tracing Exception Handling Code

- An exception is thrown and **not** caught.

*code before the try-catch block*

```
try {
```

*code before an exception is thrown*

*code that throws an object of type `ExceptionB`*

*code after statement that might throw the exception*

```
} catch (ExceptionA ex) {
```

*code in the catch block*

```
}
```

*code after the try-catch block*

*... Leave the method! and don't come back!*

*we return to the method that invoked this code and  
search for a matching catch block in that method*

*if no match is ever found, the program crashes;*

*if a match is found, we follow the trace of caught exceptions*



# Tracing Exception Handling Code

- An exception is thrown and **not** caught.

*code before the try-catch block*

```
try {
```

*code before an exception is thrown*

*code that throws an object of type `ExceptionB`*

~~*code after statement that might throw the exception*~~

```
} catch (ExceptionA ex) {
```

~~*code in the catch block*~~

```
}
```

~~*code after the try-catch block*~~

*... Leave the method! and don't come back!*

*we return to the method that invoked this code and  
search for a matching catch block in that method*

*if no match is ever found, the program crashes;*

*if a match is found, we follow the trace of caught exceptions*

# Exception Propagation

- An exception that is passed to the previous, invoking method is said to be *propagated*.
- Exceptions *propagate* up through the runtime call stack until they are caught and handled or until they reach the main method and then the program crashes.
- In the calling method, a try block can call a method that throws an exception and that exception can be caught in the calling method.

# Exceptions and Loops

- Sometimes you want to keep doing something until it executes without an exception.
- To do this, put the exception handling code **inside** the loop.

# Exceptions and Loops

```
boolean success = false;
while( !success) {
    try {
        // code to execute when all goes well;
        // this code might also throw an exception
        // if we got to here- we succeeded!
        success = true;
    } catch (ExceptionType ex) {
        // code describing how to react or handle
        // exceptions of type ExceptionType
    }
}
```

# Practice

- Modify the positive number program to keep asking the user for a number until the number is valid.

# Practice

- Write a program that reads in five 5-digit zip codes from the user.
  - Use exception handling to account for bad input.
  - Count the number of valid and invalid inputs.

# Throwing Exceptions

- So far, we've looked at exceptions thrown automatically by Java.
- You can also explicitly throw exceptions!
- Use the keyword **throw**

```
if(some condition) {  
    throw new IllegalArgumentException(message);  
}
```

# Practice

- Update the zip code program to throw an exception if the code is too short.



# Multiple catch Blocks

- A try can contain code that throws more than one type of exception.
- You can include multiple catch blocks after the try.
- Each catch block only catches exceptions of that specific type.
- When a program is run, only one catch block is ever executed.

# Multiple catch Blocks

```
try {  
    // code to execute when all goes well;  
    // this code might also throw an exception  
} catch (ExceptionA ex) {  
    // catch exceptions of type ExceptionA  
} catch (ExceptionB ex) {  
    // catch exceptions of type ExceptionB  
} catch (ExceptionC ex) {  
    // catch exceptions of type ExceptionC  
}
```

# Multiple catch Blocks

- You can have any number of catch blocks.
- But order matters!
- Exception objects come from a *hierarchy*.
- You must catch the most specific exception first because when an exception is thrown, we look for the **first** matching catch block.
  - Review the example.

# Handling Multiple Exception Type

- Often you have the same code inside of multiple catch blocks.
  - Example: `ex.printStackTrace();`
- As of Java 7, you can have a catch block catch multiple types of exceptions.
- Use this only if they will all be handled the same way.

# Multiple catch Blocks

```
try {  
    // code to execute when all goes well;  
    // this code might also throw an exception  
} catch (ExceptionA | ExceptionB ex) {  
    // catch exceptions of type ExceptionA or  
    // type ExceptionB  
} catch (ExceptionC ex) {  
    // catch exceptions of type ExceptionC  
}
```

**FINALLY**

# The `finally` Block

- A try statement can have a finally block after the catch blocks.
- The statements in the finally clause are always executed.
  - Executed if no exception is thrown
  - Executed if an exception is thrown and caught
  - Executed if an exception is thrown and not caught
- Example: closing an input/output stream

# The `finally` Block

- If no exception is generated:
  - The statements in the `finally` block are executed after the statements in the `try` block
- If an exception is generated and caught:
  - The statements in the `finally` clause are executed after the statements in the appropriate `catch` block
- If an exception is generated and not caught:
  - The statements in the `finally` clause are executed before control is handed over to the invoking method



# Tracing Exception Handling Code

- No exception is thrown in the try block.

*code before the try-catch block*

```
try {
```

*code in the try block*

```
} catch (ExceptionA ex) {
```

~~code in the catch block~~

```
} finally {
```

*code in the finally block*

```
}
```

*code after the try-catch block*



# Tracing Exception Handling Code

- An exception is thrown and caught.

*code before the try-catch block*

```
try {
```

*code before an exception is thrown*

*code that throws an object of type `ExceptionA`*

~~*code after statement that might throw the exception*~~

```
} catch (ExceptionA ex) {
```

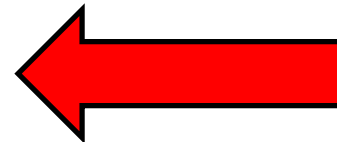
*code in the catch block*

```
} finally {
```

*code in the finally block*

```
}
```

*code after the try-catch block*



# Tracing Exception Handling Code

- An exception is thrown and **not** caught.

*code before the try-catch block*

```
try {
```

*code before an exception is thrown*

*code that throws an object of type `ExceptionB`*

~~*code after statement that might throw the exception*~~

```
} catch (ExceptionA ex) {
```

~~*code in the catch block*~~

```
} finally {
```

*code in the finally block*

```
}
```

~~*code after the try-catch block*~~



*... Leave the method! and don't come back! BUT WAIT! before Leaving, execute the code in the finally block!*

*we return to the method that invoked this code and search for a matching catch block in that method*

*if no match is ever found, the program crashes;  
if a match is found, we follow the trace of caught exceptions*

# The finally Block

- You can have a try-catch-finally.
- You can have a try-catch-catch-...-catch-finally.
- You can also have a try-finally.
  - With this setup, exceptions are propagated up to another method. But you still get to execute the finally code first.

# Practice

- Modify the zip code program to print out the number of valid and invalid codes entered with each pass through the loop.

# Practice

- Review the ExceptionTrace example.

# **CHECKED EXCEPTIONS**

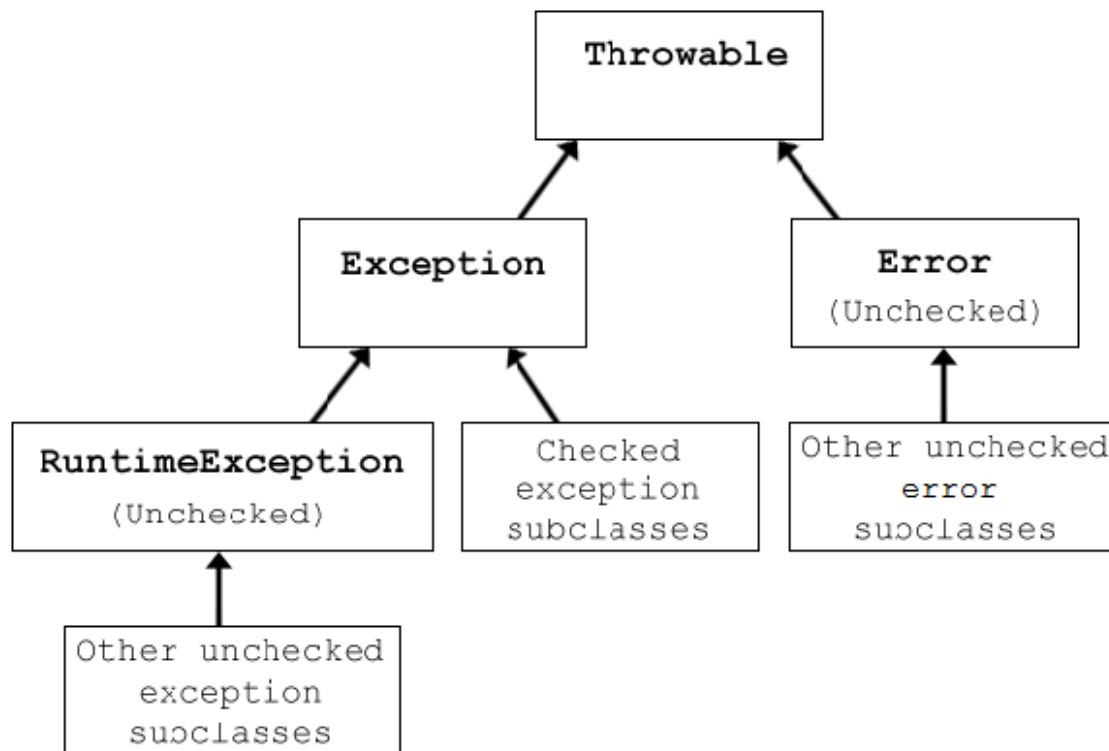
# Checked and Unchecked Exceptions

- Exceptions come in two types: checked and unchecked.
- Checked exceptions have to be dealt with.
  - The compiler *checks* to make sure you have addressed these possible conditions.
  - These follow the *catch-or-declare* rule.
- Unchecked exceptions do not have to be explicitly handled.



# The Exception Class Hierarchy

- Exception classes are related by inheritance, forming an exception class hierarchy.



# Errors

- Errors are similar to exceptions.
- Errors represent a serious problem from which your program cannot recover.
- You do *not* need to handle (catch) or plan for errors.
- Errors usually result from external conditions.

# Unchecked Exceptions

- Up until now, we've looked at examples of unchecked exceptions.
  - Examples: array out bounds, divide by zero, non-numeric input.
- We could write the code without any exception handling and it would compile fine.
  - It might crash under some conditions, but it would compile and run!
- Unchecked exceptions are *runtime* exceptions.

# Caution!

- Some unchecked exceptions can be caused by logical errors in your code.
  - Example: lack of an if-statement to make sure the denominator is not zero
  - Example: off-by-one error in array processing
- You should not use exception handling to deal with these. It's often better to write code to avoid the errors being generated in the first place.

# Checked Exceptions

- Checked exceptions cannot be left out of the code.
- They must be accounted for or the code will not compile.
- A checked exception must be either:
  - caught in the method
  - propagated up to the invoking method

# The Catch or Declare Rule

- For a checked exception, you can catch it in the method with a try-block. ("catch")
- Or, you can propagate it.
  - However, if you propagate it, you must tell the compiler! ("declare")
  - You do this with the throws keyword in the method header.
  - Using throws, you declare the type of checked exception that you are propagating.

```
public void method() throws CheckedException {
```

# throw vs. throws

- This is a classic interview question!
- throw is used when you create a new exception object.
  - throw is part of a complete statement (throw new SomeExceptionType(...))
- throws is used to declare the type of checked exceptions thrown (but not handled) by a method.
  - throws goes in the method header

# The IOException Class

- Operations performed by some I/O classes may throw an IOException
- For example:
  - A file might not exist
  - A file might exist but the program can't find it
  - A file might not contain the expected type of data
- IOExceptions are checked exceptions
  - You must either catch them or declare that your method throws them.



# Writing to a Text File

- Recall how we create a `PrintWriter` object:

```
PrintWriter out =  
    new PrintWriter(  
        new FileOutputStream(  
            "filename.txt"));
```

# Writing to a Text File

- When a file is opened in these ways, a `FileNotFoundException` can be thrown (e.g., because the file cannot be created).
- This is a **checked exception** so we have to either catch or declare!

# Writing to a Text File

- So we know the file needs to be opened inside of a try-block.
- We also know that the writer **must** be closed no matter what.
  - So that should go in a finally block!



# Writing to a Text File

- Does this work?

```
try {  
    PrintWriter out = new PrintWriter(  
        new  
FileOutputStream("filename.txt"));  
    ...  
} catch(FileNotFoundException ex) {  
    ...  
} finally {  
    out.close();  
}
```

# Writing to a Text File

- Sadly, no! A try block is like other blocks: variables declared inside of the block cannot be visible outside the block.

```
try {  
     declared here  
    PrintWriter out = new PrintWriter(  
        new FileOutputStream("filename.txt"));  
    ...  
} catch(FileNotFoundException ex) {  
    ...  
} finally {  
    out.close();  cannot be seen here  
}
```

# Writing to a Text File

- Declare the writer outside the block, initialize to null, then open the file in the try block.

```
PrintWriter out = null;
try {
    out = new PrintWriter(
        new FileOutputStream(filename.txt));
    ...
} catch(FileNotFoundException ex) {
    ...
} finally {
    out.close();
}
```

# Writing to a Text File

- We can also use a newer setup called *try with resources*.
  - More to come on this!

# Reading from a Text File

- Just like writing to a file, reading from a file can generate a `FileNotFoundException`.
- This is a **checked exception** so we have to either catch or declare!

```
Scanner fileScan = null;
try {
    fileScan = new Scanner(new FileInputStream("filename.txt"));
    ...
} catch(FileNotFoundException ex) {
    ...
} finally {
    fileScan.close();
}
```



# Practice

- Write a string of random numbers to a file.
- Modify the two previous I/O programs so they do not ignore the exception:
  - InputOutputPractice
  - UserDataRead


# Try-with-Resources

- As of Java 7, there is a simpler way to deal with some I/O exceptions.


# Old Method

```
File f = null;
try {
    f = new File(...);
    // use the file
} catch (IOException ex) {
    ex.printStackTrace();
} finally {
    if(f!=null) {
        f.close();
    }
}
```

Must declare outside  
of the try/catch/finally  
block so it can be  
seen inside the  
finally statement




Must make sure not  
null and then always  
close- sometimes  
this line can also  
generate an  
exception- nested  
blocks- yikes!



# Try-with-Resources

```
try (File f = new File(...) ) {  
    // use the file  
} catch (IOException ex) {  
    ex.printStackTrace();  
} finally {  
    // no need to close!  
}
```



Declare and initialize  
resources inside  
of ( ) after try

# Try-with-Resources

- No matter how the try block exits (with or without exception, with or without finding a matching catch block), the resource is closed.
- You can declare multiple resources inside of the try. (Separate with ; )
- You can only declare objects that implements the `AutoCloseable` interface.
  - This will essentially be any of your standard IO objects. But you can always check the API to be sure!

# Practice

- Update the IO examples to use try with resources:
  - `InputOutputPractice`
  - `UserDataRead`

# **WRITING YOUR OWN EXCEPTIONS**

# Exception Hierarchy

- Java provides many classes that describe unexpected situations.
- You might also have unexpected situations in your own programs that you want to handle.
  - Sometimes an if-else structure is best.
  - Sometimes exception handling is best.
  - This is an important design decision.
    - Often, companies will have a standard approach that defines when to use each approach.



# Defining Exceptions

- You can define your own exceptions by extending the `Exception` class or any of its descendants.

# Writing Exception Classes

- All exception classes in the Java standard library have these properties:
  - a constructor that takes one String parameter
  - an accessor method getMessage() that returns the String sent to the constructor
  - the name "Exception" in the class name
- Your class will extend an existing exception class and should have these same properties!

# Example

```
public class NegativeNumberException extends Exception {  
    private static final String MESSAGE = "Negative number";  
  
    public NegativeNumberException() {  
        super(MESSAGE);  
    }  
    public NegativeNumberException(String message) {  
        super(message);  
    }  
}
```

# The throw Statement

- When you use a Java-defined exception, Java will throw it when needed.
- When you define your own exceptions, *you* must throw them when the situation arises.
- Exceptions are thrown using the throw statement.

# The throw Statement

- A throw statement can throw an exception object of any exception class.
- Usually a throw statement is executed inside an `if`-statement that evaluates a condition to see if the exception should be thrown

# The throw Statement

- When you throw an exception, you then have to handle it, too!
- It becomes a catch-or-declare situation.

# Practice

- Review the ExceptionTester program.

# Practice

- Update the ZipCodeProcessor program.
- Create an exception class to represent an invalid 5-digit zip code:
  - The largest zip code is 99950
  - The smallest zip code is 00501



# **SUMMING UP**

# Tracing Exception Handling

1. When an exception is thrown, search for the first matching catch block in the current method.
  - A. If you find a match in the current method:
    - i. execute catch
    - ii. execute finally
    - iii. execute first line after the try-catch block (never go back up into the try!)
  - B. If you do not find a match in the current method:
    - i. execute finally
    - ii. leave the method and return to the “paused” line in the previous invoking method
    - iii. repeat Step 1 starting from that line

# Checked vs UnChecked

- Checked
  - **Must** be handled in order to compile
  - Either a) catch or b) declare and propagate using “throws” in the method header
- Unchecked
  - Design code to avoid these (e.g., `NullPointerException`, `ArrayIndexOutOfBoundsException`)
  - Use them to your advantage (e.g., reading in numeric data)
  - *Can* catch or propagate to handle

# Tips for Using Exceptions

1. Don't replace simple conditionals with exceptions.
  - If a list is empty, if a number is negative, etc.
  - Use exceptions for unusual or unexpected situations that cannot easily be handled in another way.
2. Wrap full tasks in try-blocks, not individual statements.
  - Throw early, catch late.

# Tips for Using Exceptions

3. Throw and catch specific exceptions.
  - Do not catch objects of type Exception!
  - This is bad design.
4. Handle exceptions in the appropriate place.

# Exceptions: Why do we bother?

- There are several advantages to using exceptions.
  1. It separates the error-handling code from the normal execution code.
    - The try block contains all the code we expect to run if everything goes smoothly.
  2. It allows us to propagate errors up the call stack so only the methods that care about the errors have to worry about them.
  3. It allows us to group and categorize our errors.