ENGR/CS 143

Errors and Exceptions (Chapter 18)
Streams (Chapter 19.1, Appendix A.2)

What Can Go Wrong With Programs

- Programs can have bugs and try to do things they shouldn't.
  - For example, trying to send a message to null
- Users can ask for things that they shouldn't (we can't control the users)
  - For example, a user might try to withdraw more money than what's in their account.
- The environment may not be able to provide some resource that's needed
  - Program runs out of money or disk space
  - Expected file is not found
  - etc.

Coping Strategies

- Check all user input!
  - But what should the program do if it's wrong? If you're writing a supplier class, you might not have an appropriate answer.
- Test whether required resources are available
  - But what should the program do if they aren't?
- Other strategies?

Reporting Errors with Status Codes

- If a method cannot complete properly because of some problem, how can it report the problem to the rest of the program?
- One approach: return a **status code (error code)**
- Boolean status flags are very common
  - A boolean flag: true means o.k., false means failure
- Integers or other types could be used as well:
  - An integer flag: 0 means o.k., 1 means error type #1, etc.
  - Can return a value that would never be returned by a method that ran properly. For example, when a non-negative result is expected, returning a negative value could indicate an error.
  - For object types, returning null often means error/failure.

Status Codes in Bank Account

- From the original design of the bank account operations:

```
public boolean deposit (double amount){ return this.updateBalance(amount);}
public boolean withdraw(double amount){ return this.updateBalance(-1*amount); }

private boolean updateBalance(double amount){
        if( this.balance + amount < 0 )
                System.out.println("Insufficient Funds");
                return false;
        else{
                this.balance = this.balance + amount;
                return true;
        }
}
```

- What's bad about using this boolean error flag?

Status Codes: Pro and Con

- Easy to program, in the method that detects the error.

```
myType methodThatMightFail(…){
        …
        if( weirdError){
                return null;
        }
        //continue here
        …
}
```

- Can be bothersome for callers (why?)
- Can be unreliable (why?)

An Alternative: Throwing Exceptions

- Java (and many other modern languages) include **exceptions** as a more sophisticated way to report and handle errors.
- If something bad happens, program can **throw** an exception
  - A throw statement terminates the throwing method
  - throw sends back a value, the exception object itself.
- So far, this is very similar to the return statement
  - A return statement terminates the method
  - return can send a value back to the caller

Revised BankAccount Methods

```
public void deposit (double amount){ return this.updateBalance(amount);}
public void withdraw(double amount){ return this.updateBalance(-1*amount); }

private void updateBalance(double amount){
        if( this.balance + amount < 0 )
                throw new IllegalArgumentException("Insufficient Funds");
        else{
                this.balance = this.balance + amount;
                return true;
        }
}
```

- Methods now have a void return type, not boolean
- Error message and "return false" replaced with throw of a new exception object
- Callers can choose to ignore the exception in the case that they don't know how to deal with it
  - In this case, the exception object will be passed on to the caller's caller, and so on, until some caller that can cope

Return vs. Throw

- A return takes the exchange right back to where the method was called.
  - Sometimes referred to as the "call site"
- A throw takes the execution to code designated specifically to deal with the exception.
  - This code is often called the **handler**, and is said to **catch** the exception
- The handler might not be at or near the call site
- The calling module (client) might not even have a handler
  - If a handler doesn't exist anywhere, the program aborts
- You might throw an exception that you never catch
  - You're merely supplying clients with the ability to catch the errors

Throw Statement Syntax

- To throw an exception object, use a throw statement
  - Syntax pattern:
    throw <*expression*>;
- The expression must be an object of type Throwable
  - There are many such classes already defined
  - BankAccount example used IllegalArgumentException
  - Many have an optional String argument
- Throwing an exception doesn't just return to the caller – it ends the execution of the caller, and the caller's caller, etc., until a handler is found (explained later), or the whole program is terminated.
  - It's bad practice for a complete program to die with an unhandled exception

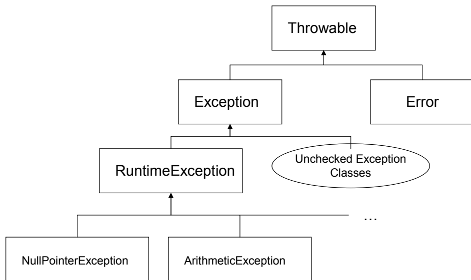
Exception Objects in Java

- Exceptions are regular objects in Java
- Exceptions are subclasses of the predefined Throwable class
- Some predefined Java Exception Classes
  - RuntimeException
  - NullPointerException
  - IndexOutOfBoundsException
  - ArithmeticException
  - IllegalArgumentException
- Most exceptions (but not all) have constructors that take a String argument

Two "Main" Types of Exceptions

- There are two main types of exceptions
  - *unchecked exceptions* include *RuntimeException* as well as any of its subclasses
  - *checked exceptions* include *Exception* as well as any of its subclasses
- General Rule of Thumb:
  - unchecked exceptions should be used in situations that should lead to program termination. Typically these are used to help the programmer discover logical errors in their code.
  - checked exceptions should be used in situations that might arise in normal circumstances and where the client has a possibility of recovering from the error.

Exception Class Hierarchy



Unchecked Exceptions

- Very simple since the compiler enforces very few rules on their use.
  - The compiler enforces no special rules on the methods that throw these exceptions nor on the place from which these methods were called.
  - In fact, Java often throws these exceptions as part of its standard run-time checking. Perhaps you've run into a couple of them?

Checked Exceptions

- The compiler imposes restrictions on the use of checked exceptions
  - A method that throws a checked exception **must** declare that it does so in a throws clause added to the header:
    public void saveFile(String dest) **throws** IOException{…}
  - The caller of the exception throwing method must make provisions for dealing with the thrown exception.
    - Put the caller within a try-catch block (the caller handles the exception)
    - The method that contains the caller may throw the exception (pass the problem onto somebody else)

Back to BankAccount (Reminder)

```
public void deposit (double amount){ return this.updateBalance(amount);}
public void withdraw(double amount){ return this.updateBalance(-1*amount);
}

private void updateBalance(double amount){
        if( this.balance + amount < 0 )
                throw new IllegalArgumentException("Insufficient Funds");
        else{
                this.balance = this.balance + amount;
                return true;
        }
}
```


Specifying an Exception Handler

- If a caller knows how to cope with an exception, then it can specify an appropriate handler using a **try-catch block**

```
try{
      mySavingsAccount.withdraw(100.00);
      myCheckingAccount.deposit(100.00);
}catch( IllegalArgumentException e){
      System.out.println("Transaction Failed: " + e.getMessage( ) );
}
```

- The **catch** part of the block constitutes the handler
- If an exception is thrown anywhere inside the body of the try block (in this case, it must be an IllegalArgumentException), then the exception is caught and the catch block is run.

Try-Catch Blocks: Syntax

- Syntax:

```
try
{
      <body, a sequence of statements>
}
catch(<exceptionType 1><name1> )
{
      <handler1, a sequence of statements>
}
catch(<exceptionType2><name2>)
{
      <handler2, a sequence of statements>
}
…
```

- Can have one or more catch clauses for a single try block

Try-Catch Blocks: Semantics

- First, execute the code in the try block
- If no exception is thrown during the execution of the try block, or all the exceptions that are thrown are handled somewhere in the body of the try block, then we're done with the try-catch block. All catch blocks are skipped.
- If an exception is thrown and not handled within the try block, then check each catch block in turn
  - See if the exception is of type <exceptionType1>
    - If so, the exception is caught. Bind <name1> to the exception and execute <handler1>; skip remaining catch blocks
  - If not, then continue checking with the next catch block (if any)
- If no catch block handles the exception, then continue searching by searching in the try-catch block surrounding the caller of this method
  - Note, if the exception is a checked exception, you must have a handler

Exception Matching

- If you have several catch clauses, the catch clauses should be ordered from the most specific type of exception to less specific types (why?).

Overriding Methods That Throw Exceptions

- When you override a method, you can only throw the exceptions that have been specified in the base-class version of the method. You cannot add new exceptions.
  - Useful restriction – insures that code which works well with the base class will also work with the derived class.

Using Exceptions

- Note that it is not entirely clear what to do when an exception is thrown. Here are some guidelines of what is reasonable, as well as some useful ways in which to use exceptions:
  - Fix the problem and call the method that caused the exception again.
  - Patch things up and continue without retrying the method
  - Calculate some alternative result instead of what the method was supposed to produce.
  - Do whatever you can in the current context and rethrow the same exception or some different exception
  - Terminate the program
  - Make your library and program safer by throwing exceptions when errors occur instead of discovering them later, when it will be more difficult to debug

One Final Note On Exceptions

- It is very easy for you to create your own exception classes.
- Exceptions are like any other class – they can be extended.
  - Simply decide an appropriate exception class to subclass from
  - Typically, the big choice is to decide whether you want your exception to be checked or to be unchecked

5 Minute Break

Data Representation

- Underneath, it's all bits (binary digits – 0 or 1)
- Byte – a group of 8 binary digits
  - Smallest addressable unit of memory
- The meaning of a particular byte depends upon the interpretation
  - Non-negative base-10 integers represented as base-2 integers
  - Character formats include ASCII (1 byte) or Unicode (2 byte)
    - 01000001 = int 65 = ASCII 'A'
    - 00110110 = integer 54 = ACSII '6'

Representation of Primative Java Types

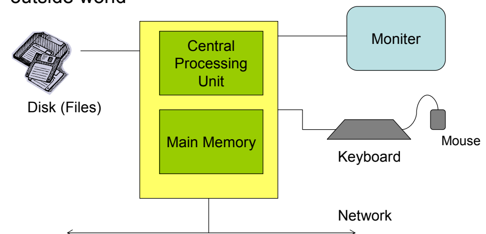
- boolean – ??? (0 = false; 1 = true)
- Integer types
  - byte – 1 byte (-128 to 127)
  - short – 2 bytes (-32768 to 32767)
  - int – 4 bytes (-2147483648 to 2147483647)
  - long – 8 bytes (-9223372036854775808 to 9223372036854775807)
- Floating-point (real number) types
  - float – 4 bytes (approx. 6 decimal digits precision)
  - double – 8 bytes (approx. 15 decimal digits precision)
- Character type
  - char – 2 bytes; **Unicode** characters 0 to 65535

Unicode

- International standard
  - Java was the first major language to adopt Unicode
- Intended to include all the world's writing systems
- Characters are 2 bytes (16 bits)
  - Given by two Hex digits, e.g. 4EB9
- Specifications: www.unicode.org
- Luckily, we usually don't need to deal directly with the byte representation.

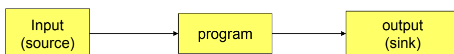
Input and Output

- Communicating with the outside world



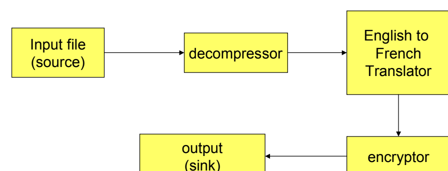
Streams

- Java model of communication; streams
  - Sequence of data flowing from a source to a program, or from a program to a destination (sink)
  - Files are common sources and sinks



Stream after Stream …

- Streams are a useful model for processing data along the way in a pipeline



Streams vs. Files

- Many languages don't make a clear distinction between streams and files
- In Java:
  - "file" is the collection of data, managed by the operating system
  - "stream" is a flow of data from one place to another
- Streams can connect several different types of sources and sinks

Java Stream Library

- Huge variety of stream classes in java.io.*
  - Some are data sources or sinks
  - Others are converters that take data from a stream and transform it somehow to produce a stream with different characteristics
- Highly Modular
  - Lots of different implementations all sharing a common interface; can be mixed, matched, and chained relatively easily
  - Great OO design example, in principle
  - In practice, it can be very confusing

Common Stream Processing Pattern

- Basic idea is the same for input and output

```
// input
open a stream
while more data{
    read & process next data
}
close stream
```

```
//output
open a stream
while more data{
            read & process next data
}
close stream
```


Opening & Closing Streams

- Before a stream can be used, it must be *opened*
  - Create a stream object and connect it to a source or destination of the stream data
  - Connection to the source or sink is often done implicitly as part of the stream object construction
- When you're done with a stream, it should be *closed*
  - Cleans up any unfinished operations, then breaks the connection between the program and the data source/destination

Java Streams

- 2 major families of stream classes, based on the type of data
  - **Byte streams** – read/write byte values
    - corresponds to physical data – network and disk I/O streams
    - Abstract classes: InputStream and OutputStream
  - **Character streams** – read/write char values
    - text input/output stream classes
    - Abstract classes: Reader and Writer
- System.out is a stream – what kind of stream do you think it is?
  - There is also another stream – System.in. What kind of stream do you think this is?

More Complications: Exceptions

- All IO operations can throw IOExceptions
  - These are checked exceptions, so you **MUST** deal with them
  - It might be that a specific subclass of IOException is thrown. This will depend on the actual error.

Basic Reader/Writer Operations

- Reader
  int read( )   // return Unicode value of next character; -1 if end-of-stream
  int read(char[ ] cbuf)   // read several characters into an array; -1 if end-of-stream
  void close( )   // close the stream

- Writer
  void write(int c);   // write character whose Unicode value is c
  void write(char[ ] cbuf) // write array contents
  void write(String s)   // write String
  void close( )   // close the stream

- To convert a Unicode value to a char, or vice-versa: use cast Syntax

File Readers and Writers

- To read a text file (not a binary data file), instantiate FileReader.
  - A subclass of Reader; implements read and close operations
  - Constructor takes the name of the file to open and read from
- To write a text file, instantiate FileWriter
  - A subclass of Writer: implements write and close operations
  - Constructor takes the name of the file to open/create and overwrite (can also append to an existing file using a different constructor)

Copy a Text File, One Character at a Time

```
public void copyFile(String sourceFile, String destFile){
    FileReader inFile;
    FileWriter outFile;
    try{
        inFile = new FileReader(sourceFile);
        outFile = new FileWriter(destFile);
        int ch = inFile.read( );
        while(ch != -1){
            outFile.write(ch);
            ch = inFile.read( );
        }
    }
    catch(IOException e){
        System.out.println("Something Horrible has Happened " + e);
    }
    inFile.close( );
    outFile.close( );
}
```


More Efficient I/O – Buffered Reader/Writer

- Can improve efficiency by reading/writing many characters at a time
- BufferedReader: a converter stream that performs this chunking
  - BufferedReader constructor takes any kind of Reader as an argument – can make any read stream buffered
  - BufferedReader supports standard Reader operations – clients don't have to change to benefit from buffering
  - Also supports readLine( ) [read an entire line of text]
- BufferedWriter: a converter stream that performs chunking of writes
  - BufferedWriter constructor takes any kind of Writer as an argument
  - BufferedWriter supports standard Writer operations
  - Also supports newLine( ) [write an end-of-line character]

Copy a Text File One Line at a Time

```
public void copyFile(String sourceFile, String destFile){
    BufferedReader inFile;
    BufferedWriter outFile;
    try{
        inFile = new BufferedReader(new FileReader(sourceFile));
        outFile = new BufferedWriter(new FileWriter(destFile));
        String line = inFile.readLine( );
        while(line != null){
            outFile.write(line);
            line = inFile.readLine( );
        }
    }
    catch(IOException e){
        System.out.println("Something Horrible has Happened " + e);
    }

    inFile.close( );
    outFile.close( );
}
```


One Final Complication

- What "bad thing" could would happen if an exception is thrown in the try block in the code on the previous slide?
- We need some code to execute regardless of whether or not an exception is thrown.
  - Solution: a finally block
  - A finally block is used in conjuction with a try-catch block. The code within a finally block is executed regardless of whether or not an exception is thrown.

Using *finally*

```
public void copyFile(String sourceFile){
        BufferedReader inFile = null;
        try{
                inFile = new BufferedReader(new FileReader(sourceFile));
                String line = inFile.readLine( );
                while(line != null){
                        System.out.println(line);
                        line = inFile.readLine( );
                }
        }
        catch(IOException e){
                System.out.println("Something Horrible has Happened " + e);
        }
        finally{
                //Close the stream
                try{
                      if(inFile != null)
                        inFile.close( );
                }
                catch(IOException e)
                { // Can't do anything about this one }
        }
}
```


That Was a Pain!!!!

- What makes this so tricky?
  - inFile must be declared outside of the try block so that it is visible inside of the finally block
  - inFile must be initialized to null to avoid compiler complaints about a possibly uninitialized variables
  - Prior to calling close, we must check that the value of inFile is not null (could be null if the file was never found)
  - close can throw a checked exception – thus, a try-catch block is required when closing the file

Conclusion

- We covered a lot today!!!!!!
- Error Handling
  - status codes
  - throwing exceptions
  - handling exceptions with try-catch blocks
- Streams & Files
  - files are common stream sources and/or sinks (there are others)
  - opening and closing files
  - basic file operations
- You will get practice with all these things in your next homework.