Exceptions
What Is an Exception?

- Exception is shorthand for the phrase "exceptional event."
  
  **Definition**: An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

- When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error.

- Creating an exception object and handing it to the runtime system is called *throwing an exception*. 
Implicitly Throwing Exceptions

- When an exception is *thrown*, the method will stop execution at the line where the error originated.

```java
class ThrowingExceptions {
    public static void main(String[] args) {
        divByZero(1);
    }

    private static int divByZero(int i) {
        System.out.println("About to divide by zero! \n");
        int result = i / 0;
        System.out.println("I will never ever print!");
        return result; // this will never happen!!
    }
}
```
The Stack Trace

- Because we do nothing to prevent it, the exception thrown from line 8 terminates our program with an error...

```
About to divide by zero!
Exception in thread "main"
java.lang.ArithmeticException: / by zero
    at ThrowingExceptions.divByZero(ThrowingExceptions.java:10)
    at ThrowingExceptions.main(ThrowingExceptions.java:5)
```

- This error output is known as a “stack trace”

- Note three things…
  - Every method call in the call stack will has a line.
  - The third line is where the exception was thrown from.
  - Exceptions have a type, in this case `ArithmeticException`
Avoiding Exceptions

- So what can we do?

- Sometimes we can program “defensively”.

- However this solution often has shortcomings.

- And not all exceptions can prevented in this way since you don’t always know if a given operation will fail or not before it is invoked.
Exception Handling

- The way we handle these cases in Java is by doing what is called “exception handling”.

- When an exception is thrown…
  - The runtime system searches the call stack for a method that contains a block of code that can handle the exception. (This block of code is called an exception handler.)
  - The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called.
  - An exception handler is a match if the type of the exception object thrown matches the type that can be handled by the handler.
The exception handler chosen is said to *catch* the exception.

If the runtime system searches all the methods on the call stack without finding an exception handler, the program terminates.
Try/Catch Blocks

- If we expect that a method may emit an exception, then the call is wrapped with a *try/catch block*.

- A *try block* wraps the line(s) that *may* throw exceptions. The *catch blocks* provides handler(s) for the exception(s).
  - When an exception in a *try block* is encountered, execution will stop at the line from where it was *thrown*.

- Control flow then proceeds to the *catch blocks* that appear directly after the *try block*
  - Catch blocks look just like method definitions – they have arguments inside parenthesis that define what type of exception they handle.
Try Block

- In general, a try block looks like the following:

```java
try {
    // code
    // code
}
catch blocks . . .
```

- The segment in the example labeled `code` contains one or more statements that *could* throw an exception.
Catch Block(s)

- You associate exception handlers with a try block by providing one or more catch blocks directly after the try block.

```java
try {
    // code
} catch (ExceptionType name) {
    // code to handle exception
} catch (ExceptionType name) {
    // code to handle exception
}
```

- Each catch block is an exception handler that handles the type of exception indicated by its argument.
Catch Blocks *con’t*

- The catch block contains code that is executed if and when the exception handler is invoked.

- The runtime system invokes the exception handler when the handler is the first one in the call stack whose `ExceptionType` matches the type of the exception thrown.

- The system considers it a match if the thrown object can legally be assigned to the exception handler's argument.
Try/Catch Example

```java
public class CatchingExceptions {
    public static void main(String[] args) {
        try {
            int result = quotient(1, 0);
            System.out.println("The result is "+ result);
        } catch (ArithmeticException e) {
            System.out.println(e);
        }
    }

    private static int quotient(int a, int b) {
        int result = a / b;
        return result;
    }
}
```
Explicitly Throwing Exceptions

- We can throw exceptions explicitly using the `throw` keyword.

```java
public static void main(String[] args) {
    int r = (int) (Math.random() * 2);
    if (r % 2 == 0) {
        throw new IllegalArgumentException("A nonsense exception!");
    } else {
        throw new ArrayIndexOutOfBoundsException("Also a nonsense exception!");
    }
}
```

- If we know how to throw and handle exceptions, this enables much more robust error handling.
public class ExceptionHandling {
    public static void main(String[] args) {
        int numerator = (int) (Math.random() * 2);
        int denominator = (int) (Math.random() * 2);
        try {
            int i = quotient(numerator, denominator);
            System.out.printf("%d / %d = %d.", numerator, denominator, i);
        } catch (IllegalArgumentException e) {
            System.out.println(e);
        }
    }

    private static int quotient(int a, int b) {
        if (b == 0) {
            throw new IllegalArgumentException("Cannot divide by zero.");
        }
        int result = a / b;
        return result;
    }
}
As mentioned previously, thrown exceptions are objects that contain information about the error.

Exception is the superclass of all exceptions.

Most programs throw and catch objects that derive from the Exception class.

Java defines many descendants of Exception which indicate various types of problems that can occur.

Ex. IllegalAccessException signals that a particular method could not be found, and

Ex. NegativeArraySizeException indicates that a program attempted to create an array with a negative size.
Exception Class Hierarchy

- Any class that extends Throwable can be “thrown”.

- Any class that extends Exception indicates that a problem occurred, but it is not a serious system problem.

- Note RuntimeException
Subclasses of Exception

- **`RuntimeException`** is a subclass of Exception with some special characteristics.
  - For now, all the exceptions we look at will be descendants of `RuntimeException`. (We’ll learn about why this is soon.)

- In other respects, `Exception` and its subclasses are just a plain old Java classes, which means you can create your own exceptions by extending subclasses of Exception.
  - We might do this to model a particular type of error we have in our program.
Programming Example

- Write a Pokemon class that allows you to create a Pokemon with a particular type and a name.
- Define `type` as a String in the constructor for the class
- If the String is not one of our known types then throw a new “PokemonTypeException”
- See `exceptions/pokemon/*.java`
Java code must honor the ‘Catch or Specify Requirement’.

Moreover, code that might throw certain exceptions must be enclosed by either of the following:

- A `try/catch` block that catches the exception.
- The method that contains the code must provide a `throws` clause that lists the exception,

Code that fails to honor this requirement will not compile.**
public static void writeSpecify() throws FileNotFoundException {
    File f = new File("test.txt");
    PrintWriter writer = new PrintWriter(f);
    writer.println("Hello");
    writer.close();
}

public static void writeCatch() {
    File f = new File("test.txt");
    PrintWriter writer;
    try {
        writer = new PrintWriter(f);
        writer.println("Hello");
        writer.close();
    } catch (FileNotFoundException e) {
        System.out.println(e);
    }
}
3 Kinds of Exceptions

- Not all exceptions are subject to the ‘Catch or Specify Requirement’.
- There are 3 kinds of exceptions
  - Checked
  - Unchecked
  - Error
- Only ‘Checked’ exceptions have to abide this rule.
3 Kinds of Exceptions: Checked

- Checked exceptions
  - Exceptional conditions that a well-written application should anticipate and recover from.
  - Example: Attempting to read from a file that does not exist.
  - Checked exceptions are subject to the ‘Catch or Specify Requirement’.
3 Kinds of Exceptions: Errors

- Errors
  - Exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from.
  - Example, suppose that an application successfully opens a file for input, but is unable to read the file because of a hardware or system malfunction.
  - Errors are not subject to the ‘Catch or Specify Requirement’.
3 Kinds of Exceptions: Unchecked

- **Unchecked exceptions**
  - Exceptional conditions that are internal to the application, and that the application *usually cannot anticipate or recover from*.
  - These usually indicate programming bugs, such as logic errors or improper use of an API.
  - Example, consider our Pokemon class from earlier. What if a bug caused `null` to be passed as the type, the constructor will throw `NullPointerException`.
  - The application can catch this exception, but it may make more sense to eliminate the bug that caused the exception to occur.
Files
All of the programs that we have written so far have reset themselves each time they execute.

This is because the data associated with the program are stored in your computer’s memory (RAM).

RAM is volatile – it serves as a computer’s short term memory. Data stored in RAM that is being used by a program disappears when the program terminates.

If programs are to retain data between executions then we need use non-volatile storage.
**Long term storage**

- A computer will almost always have at least one type of long-term storage device at its disposal (usually a ‘hard drive’).

- We can use the long term storage capabilities of a computer to store data in the form of a file. This requires us to request access to write and/or read from/to the file from the operating system.

- Once we save a file it will remain on the long term storage device after the program is finished running, and can be accessed and retrieved later on.
Files & file types

- This is a common technique and is one way programs that need to keep track of some kind of information between executions ‘persist’ data.

- Data is written to files on the file system. Its common for applications to have their ‘file format’.

- There are two types of files that a computer uses.
  - Text Files
  - Binary Files

- Every file can be categorized as one of these two types.
Text Files

- Text files are files that contain data that is encoded as text (i.e. ASCII or Unicode characters)

- Data stored in a text file is visible and can be read by a program that is designed to view / edit textual data (i.e. a word processing program)

- Examples:
  - Notepad/Sublime/TextEdit documents
  - Cookies
  - HTML files
  - JSON files

- We will be working with text files exclusively in this class
Binary Files

- Binary files contain data that is not encoded as a text format
- Binary files are intended to be read by other programs and not by humans directly
- Binary files generally appear unintelligible when viewed via a word processing program

Examples
- Image files (PNG / GIF / JPG files)
- Movie Files (MOV / MP4 / AVI files)
- Application files (EXE / APP files)
The File class

- The **File** class is Java Standard Library class that can be used to interface with files stored in long term storage on a computer.

- The **File** class in Java is used to define a connection between a file and your program. It has no built in capability to read data from or write data to the file it connects to.
The File class

- In order to use the `File` class you need to import it from `java.io.File`.

- Next, you can instantiate an object of type `File` which takes a `String` as an argument for construction. This `String` should represent the *relative path* to the file you wish to work with.

```java
File testFile = new File("src/data/myfile.txt");
```
The File Class

- Once you create a `File` object you can call a number of methods on the object to learn more about the file you are pointing to, such as:

  ```java
  System.out.println("File exists? " + testFile.exists());
  System.out.println("Full path: " + testFile.getAbsolutePath());
  System.out.println("Can I read? " + testFile.canRead());
  System.out.println("Can I write? " + testFile.canWrite());
  ```

- See files/CreateFileObject.java
### Writing to a File

- Writing to a file involves data being sent from your program to the `File` object and (eventually) into the appropriate file on your hard drive.

- The `File` class does not have the ability to write directly to a `File`. To do this we need to use a new class called `PrintWriter` (which must be imported from `java.io`) – here’s an example:

  ```java
  PrintWriter myWriter = new PrintWriter(testFile);
  ```
Important!

- If the **File** passed to the **PrintWriter** does not exist it will be created for you.

- Upon opening, the **PrintWriter** class will destroy the contents of a file. This isn’t an error – it’s just the behavior of the class.

- No warning will be given to the user that this will happen, so be careful when opening up files for writing!
Important!

- **The PrintWriter class can throw a FileNotFoundException.**
  - If the given file object does not denote an existing, writable regular file and a new regular file of that name cannot be created, or if some other error occurs while opening or creating the file

- **FileNotFoundException** is a Checked Exception, which means you must ‘Catch or Specify’
Writing to a File

- Once a file has been opened we can call methods on the PrintWriter class to write data to it.

- These methods mirror the output methods that we have been using via the System.out class since the beginning of the term.

- Example:

```java
File a = new File("src/data/test1.txt");
PrintWriter writer = new PrintWriter(a);

writer.println("Hi there!");
writer.println("What’s up?");
```
Writing to a File

- When you are finished writing to a file you must close your PrintWriter object.

  PrintWriter writer = new PrintWriter(someFile);
  writer.println(“some data”);
  // close the file!
  writer.close();

- If you omit the close() method your file may end up empty.
  - This is due to a concept known as ‘stream buffering’
  - You can alternatively called the flush method on the writer.
Programming Example

- Write a program that asks the user to enter in a series of city names
- Store these names in a file called “city_database.txt”
- Store one city per line until the user chooses to stop inputting data

See files/WriteCityDatabase.java
Reading from a File

- Reading from a File requires you to use the Scanner class.
- When we want to get input from the console we can instantiate a new Scanner like this:

  ```java
  Scanner input = new Scanner(System.in);
  ```

- When we want to get input from a file we can instantiate a new Scanner like this:

  ```java
  File theFile = new File("src/data/file1.txt");
  Scanner fileInput = new Scanner(theFile);
  ```
Opening a File with Scanner is non-destructive – you won’t affect the contents in the file by opening up a File in this way.

Note that compiler will force you to wrap the instantiation of a Scanner inside a try/catch block, or declare a “throws” statement at the top of the method.
Reading from a File

- Once the file has been opened you can read from it using standard Scanner methods, like this:

```java
File theFile = new File("src/data/file1.txt");
Scanner input = new Scanner(theFile);

// read the first line of data
String data = input.nextLine();

// close the Scanner to be nice
input.close();
```
Programming Example

- Write a program to open up your city database for reading
- Read in the first city and print it out to the user
- See files/ ReadOneLineOfData.java
Reading Multiple Lines from a File

- Reading one line is useful, but you will often need to read many lines of data from a File

- You can do this by using the hasNextLine() method on the Scanner class, like this:

```java
Scanner input = new Scanner(theFile);

// while there is more data to read
while (input.hasNextLine())
{
    String line = input.nextLine();
    System.out.println("Line is: " + line);
}

input.close();
```
Programming Example

- Write a program that opens up your city database file and reads in each line.
- Store each line in an ArrayList for future processing
- See files/ReadAllLinesInArrayList.java
One way to store many items in a file is to come up with a record keeping system.

For example, we could say that every line in a file represents a single student in our class. On that line we will store the following:

- First Name
- Last Name
- Role (Student, TA or Grader)
- Student ID
- Test Score #1
- Test Score #2
- Test Score #3
The problem is that we have to find a way to logically separate each piece of data so that they don’t all run together.

We can do this by “delimiting” the data with a known character.

For example, we could store all of our fields delimited by the “dash” character, like this:

John-Smith-Student-N1234-100-96-85

We could then “split” the string whenever we see that delimiter and extract out the necessary components.
Programming Example

- Write a program that opens up our student information text file, which is dash delimited
- Extract out each line and print out the student’s name to the user (not the whole line)
- See files/ReadAndProcessDelimitedData.java
Programming Example

- Extend your program to include a Student object and store each student’s data from the file in a new object
- Write a method on this object called “getGrade()” which calculates a student’s grade
- Store these students in an ArrayList and get the grades for the class
- See files/
  ReadAndProcessDelimitedDataIntoObjects.java
Programming Example

- Attempt to run your program on the “delimited_with_bad_data.txt” file
- Handle any errors that may occur during the parsing process
- See files/
  `ReadAndProcessDelimitedDataIntoObjectsWithMoreErrorHandling.java`