Suppose I have the following classes:

```
public interface Talker {
    public void Talk(int x);
}
```

```
public class Repeater implements Talker {
public void Talk(int x) {
        System.out.println(x + "is" + x);
    }
}
```

```
public class RepeatShouter extends Repeater{
public void Talk(int x) {
 System.out.println("LOUD " + x + "" + x);
}
public void Foo() { }
}
```

Which of the following will compile and what will they print?

```
Talker a = new Talker();

Talker b = new Repeater();
b.Talk(3);
```

```
Talker c = new RepeatShouter();
c.Foo();
```

```
Repeater d = new RepeatShouter();
((RepeatShouter)d).Foo();
```

Last week:

-Inheritance
-Polymorphism
-Interfaces

This Week:

-Reading and writing from and to files
-Try/Catch statements
-Scope of variables

Reading and Writing From a File

Reading and Writing To and From a File is very similar to reading and writing to and from the keyboard and screen.

Reading and Writing From a File

To read from a file, we will use the Scanner class

To do this, we will do the same thing as to read from the keyboard, except instead of telling the Scanner to look at the keyboard, we will tell it where the file is.

System.in

1)Inside the class System, there is a static attribute called “in”

2)System.in is of type “InputStream”

System.in

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

4)This tells us we can create a Scanner based on an InputStream

5)Once we have the Scanner created it doesn't matter how it was created as far as the way we'll use it

Calling the Scanner constructor

There are generally two input types to the Scanner constructor:

1)InputStream
2)File

Calling the Scanner constructor

What we want to be able to do is convert a “path” (i.e. C:\documents\foo.txt) to a File object. Once we do that, we can convert the File object to a scanner using the constructor.

Calling the Scanner constructor

If we look at the File constructor, we'll see there is a method that takes as input a String representing a path and creates a File.

Calling the Scanner constructor

Combining these, we have:

```
File f = new
File("C:\documents\foo.txt");
Scanner s = new Scanner(f);
```

Calling the Scanner constructor

Or

```
Scanner s  = new Scanner(new
File("C:\documents\foo.txt");
```

Eek! Careful!

```
Scanner s  = new Scanner(new
File("C:\documents\foo.txt");
```

The above uses the "escape characters" This is not what we want. So we'll have to do

```
Scanner s  = new Scanner(new
File("C:\\documents\\foo.txt");
```

Reusability

Once we have set up the scanner, we can use it the same way we used the Scanner to read from the keyboard.

s.nextInt();
s.next();
s.nextLine();
etc

Closing a Scanner

After you are finished reading from a file, you must close the file. This lets the OS clean up things. You can do this by writing

```
s.close()
```

If you don't do this, changes to the file may not be saved!

Try / Catch error :(

The constructor for Scanner listed shows the following :

```
public Scanner(File source)
        throws FileNotFoundException
```

Try / Catch error :(

public Scanner(File source) throws FileNotFoundException

What this means is we *have* to put the call to this constructor inside of a Try/Catch statement. Otherwise there is a compiler error

Try / Catch

```
try {
      //some commands
}
catch (typeofexception e) {
      //do something
}
//rest of code
```

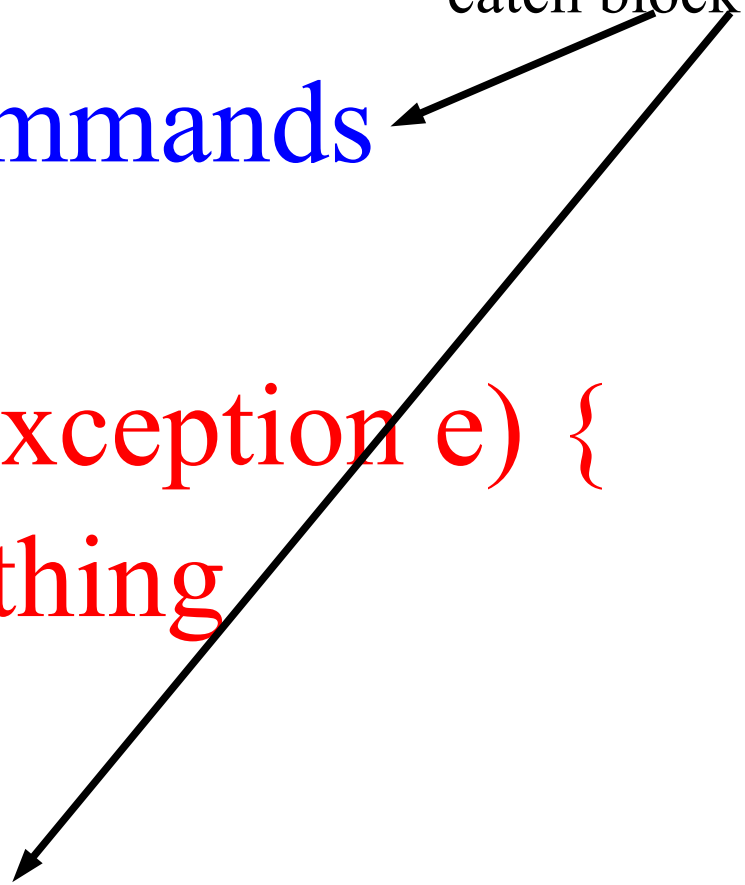
```
try {
      //some commands
}
catch (typeofexception e) {
      //do something
}
//rest of code
```

First the computer will “try” to do whatever is in the “try” statement.

There are 3 cases:
1)It works
2)It fails because of an error “typeofexception”
3)It fails because of a different error

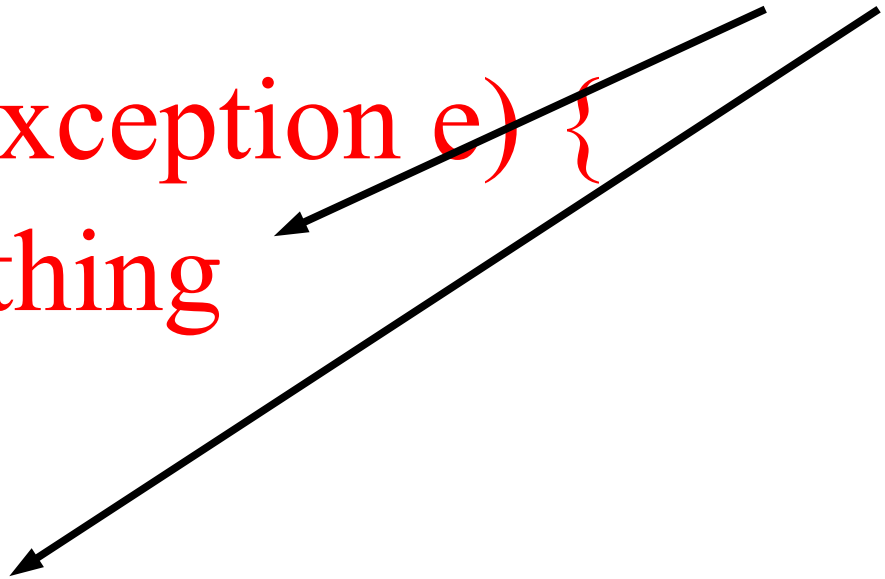
If it succeeds, it will execute everything inside of the try and then go to the part "rest of code," skipping over the catch block

```
try {
      //some commands
}
catch (typeofexception e) {
      //do something
}
//rest of code
```



```
try {
      //some commands
}
catch (typeofexception e) {
      //do something
}
//rest of code
```

If it fails because of an error of type “typeofexception” it will immediately jump to the catch block. It will then do whatever is in the catch block and then continue with “rest of code” (unless the catch block has a return statement or something)



If it fails because of an error that is not of type "typeofexception" your program will crash as normal with a run-time error

```
try {
    //some commands
}
catch (typeofexception e) {
    //do something
}
//rest of code
```

typeofexception:

There are many types of exceptions:

IOException
FileNotFoundException
NumberFormatException
Exception

These various types of exceptions have a hierarchy of “is-a” relationships as well.

For example a FileNotFoundException “is-a” IOException “is-a” Exception

```
try {
      //some commands
}
catch (typeofexception1 e) {
      //do something
}
catch (typeofexception2 e) {


}
```

When you have 2 catch statements attached to one try, the Java run time environment will figure out which type of exception was "thrown" and go to that one.

```
try {
      //some commands
}
catch (Exception e) {
}
catch (FileNotFoundException e) {
//this code never happens since a
//FileNotFoundException “is-a”
//Exception
}
```

Note that in this case only the **first** matching catch is executed

```
try {
      //some commands
}
catch (typeofexception1 e) {
      //do something
}
finally {
}
```

Sometimes you will have code that you want to happen after both the try and the catch no matter what. You want this code to happen even if the try or the catch statement had, for example, a return statement in them or threw another error

```
try {
      //some commands
}
catch (typeofexception1 e) {
      //do something
      throw e;
}
finally {
}
```

Sometimes you will want to "throw an exception" for different reasons. To do this, use the keyword throw

```
public class MisleadingError {
  public static void main(String[] args) {
     throw new
ArrayIndexOutOfBoundsException(1);
  }
}
```

```
public class MisleadingError {
  public static void main(String[] args) {
     throw new
ArrayIndexOutOfBoundsException(1);
  }
}
```

```
daniels-computer:~ daniel$ java MisleadingError
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: Array index out
of range: 1
        at MisleadingError.main(MisleadingError.java:3)
```

Suppose I have the following classes:

```
public class Crasher {
public void Boom(int x) {
        throw new Exception("PARENT");
      }
}

public class BigCrasher extends Crasher{
public void Boom(string x) {
throw new Exception("STRING BOOM!");
}
public void Boom(int x) {
        throw new Exception("INT BOOM");
 }
 public void Boom(double x) {
        throw new Exception("DOUBLE
BOOM");
}
public void Boom(Object o) {
        throw new Exception("OBJECT
BOOM!");
}
}
```

```
Crasher c = new
BigCrasher();
try {
  c.Boom(3);
}
catch (Exception e) {

System.out.println(e.getMe
ssage();
}
try {
  ((BigCrasher)c).Boom(3
+ " ");
}
catch (Exception e) {

System.out.println(e.getMe
ssage();
}
```

Try / Catch error :(

The constructor for Scanner listed shows the following :

```
public Scanner(File source)
        throws FileNotFoundException
```

Checked vs Unchecked Exceptions

When a method header has "throws _____" in it, it means it is a checked exception.

This means when you call the method, you **must** add a catch statement to catch at least that sort of Exception, OR alternatively, YOU can add the "throws _____" to your code

Checked vs Unchecked Exceptions

Any time you throw an Exception that is NOT a RuntimeException (e.g. ArrayOutOfBounds) you MUST add throws to your method header.

Note: There are also Errors such as OutOfMemory that are not considered Exceptions. These are unchecked so you don't need to add a catch

Exceptions vs Fixing the Error

A good way to think about exceptions:

In general to avoid our program crashing we can do one of two things:

1)Fix the error (i.e. check if something is null or an index of an array is in bounds)

2)Catch the exception

Exceptions vs Fixing the Error

Usually, when we can, we like to use the first method of avoiding the error in the first place. This is what we have been doing for the most part in this course

Exceptions vs Fixing the Error

Sometimes though, it is impossible to easily fix an error such as this for a few reasons:

1)Hard to predict what will happen (e.g. user entering a double when we are looking for an int)

2)The error is a normal behavior (e.g. file not found)

Exceptions in lieu of returning

Another use of an exception is a round about way to “return” an additional type.

Exceptions in lieu of returning

```
public static int search(ArrayList<String> array,
String target) {
        if (array.contains(target)){
                return array.indexOf(target);
        else {
                throw new Exception("not found");
        }
}
```

Exceptions in lieu of returning

```
public static void caller() {
        ArrayList<String> foo = new ArrayList<String>();
        ...
        int target = -1;
        try {
        target = search(foo, “findme”);
        }
        catch (Exception e) {
                System.out.println(“It must not have been”
                                                + “ found”);
        }
}
```

Defining your own exception class

```
public class NotFoundInArrayListException extends
Exception {
        public NotFoundInArrayListException() {
                super();
        }
}
```

now I can catch a

NotFoundInArrayListException

Defining your own exception class

```
public class NotFoundInArrayListException extends
Exception {
        public NotFoundInArrayListException() {
                super();
        }
}
```

If I did not have the red part, then I would have a compiler error when I try to create a new NotFoundInArrayListException()

Exceptions in lieu of returning

```
public static int search(ArrayList<String> array,
String target) {
        if (array.contains(target)){
                return array.indexOf(target);
        else {
                throw new
NotFoundInArrayListException("not found");
        }
}
```

Exceptions in lieu of returning

```
public static void caller() {
        ArrayList<String> foo = new ArrayList<String>();
        ...
        int target = -1;
        try {
        target = search(foo, “findme”);
        }
        catch (NotFoundInArrayListException e) {
                System.out.println(“It must not have been”
                                                + “ found”);
        }
}
```

Because of the Scanner constructor throwing an exception, we must write:

```
try {
        Scanner s = new Scanner(new
File(“C:\\documents\\foo.txt”);
}
catch (FileNotFoundException e) {
        System.out.println(“The file was not
found”);
        return;
}
```

```
try {
        Scanner s = new Scanner(new
File("C:\\documents\\foo.txt");
}
catch (FileNotFoundException e) {
        System.out.println("The file was not
found");
        return;
}
int x = s.nextInt();
// Compile time error! s not defined!
```

Scope:

The scope of a variable refers to exactly what part of the code it is defined in.

We have seen that variables defined inside of a method are not “available” outside of that method. We have seen the same for loops:

This concept can be applied more generally to ANY time we have { }

```
Scanner s;
try {
        s = new Scanner(new
File("C:\\documents\\foo.txt");
}
catch (FileNotFoundException e) {
        System.out.println("The file was not
found");
        return;
}
int x = s.nextInt();
// Compile time error! s may not be initialized
```

```
Scanner s = null;
try {
        s = new Scanner(new
File("C:\\documents\\foo.txt");
}
catch (FileNotFoundException e) {
        System.out.println("The file was not
found");
        return;
}
int x = s.nextInt();
// Compile time error! s may not be initialized
```

Writing to a file: Writing to a file is similar to writing to the screen.

We just have to tell Java where to write to instead of writing to the screen.

System.out is of type PrintStream in Java.

```
PrintStream writer = new PrintStream(new
File(“foo.txt”);
writer.println(“I'm writing to a file!”);
writer.close();
```

http://www.artima.com/designtechniques/exceptionsP.html

Important Java ideas:

-Control flow

-Variables

-Classes

-Recursion

-Is-A relationships (inheritance / interfaces

-Exception handling

Important Programming concepts:

-Using methods to avoid code duplication and keep things organized

-black box programming

-debugging

Final exam information: (not an exhaustive list)

-About half will be programming (includes recursion)

-Debugging question

-Scope question

-“is-a” relationships and “has-a” relationships

-Some “short answer” questions where the goal is to demonstrate you understand code flow

General tips:

-Make sure not to spend too much time on any one question. Pay attention to how much the question is worth.

-SKIM over things before hand. Don't try to understand everything in a lot of detail necessarily until you understand what the question is.

-If something is unclear, ask. I will be there during at least parts of the exam.