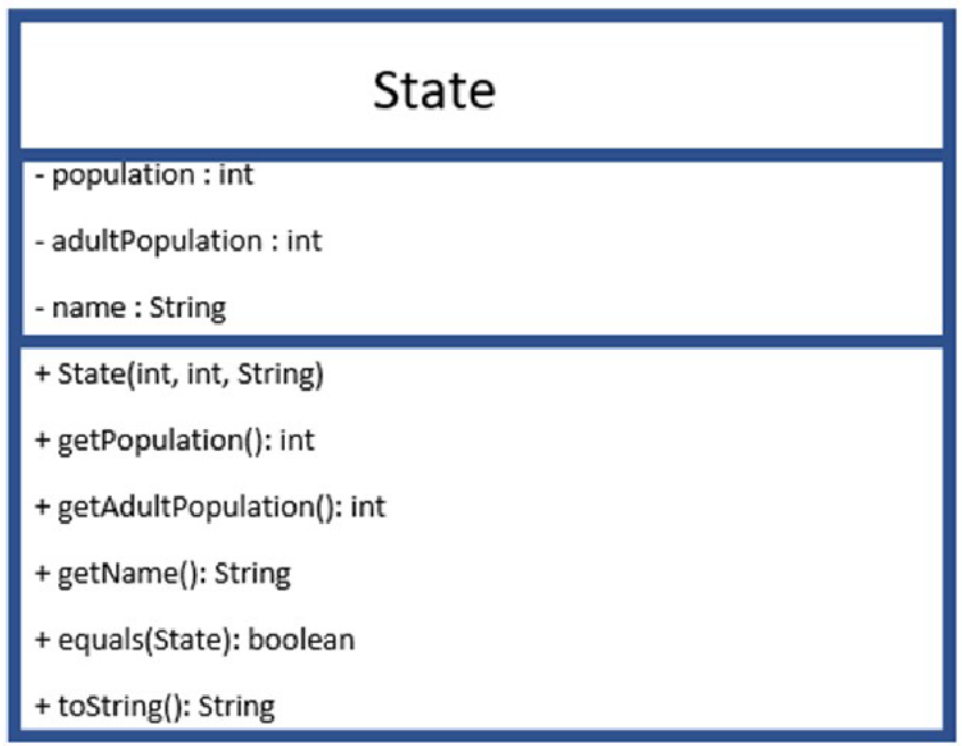
ICS 141 Programming with Objects

Jessica Maistrovich

Metropolitan State University

Try it! (Review)

- Open Eclipse and create a new project called State Application.
- Create a class based on the following UML:



File class

Introduction

- Almost all programs of any importance do massive amounts of input and output from and to disk files.
- File I/O is one of the most important parts of programming.
- Files are extremely useful because often the data for a program comes from files and often the results are stored in files.
- In this lecture, we will learn how to write Java programs that read from and and/or write to files.

Information in a File

- Files are stored on disk in binary format.
- However, a file can contain any type of information, such as:
  - numbers (integers and floating point types of any size)
  - characters (text files)
  - characters with format information (word processor files)
  - images
  - audio
  - program source files (`MyProgram.java, cprogram.c`)
  - bytecodes (`MyProgram.class`)
  - machine language (`cprogram.exe`)
  - many other types

Notes on the `File` class

- The `File` class contains methods for:
  - obtaining the properties of a file/directory
  - Renaming and deleting file/directory.

- The `File` class does not contain methods for:
  - Reading and writing file contents

- The `File` class is intended to provide an abstraction that deals with most of the machine-dependent complexities of files and path names in a machine-independent fashion.

- Constructing a `File` instance does not create a file on the machine

- A `File` instance can be created for any file name regardless of whether it exists or not.

- You can invoke `exists()` method on a file instance to check whether the file exists or no.

Try it!

- Create a driver class for the State Application. Include the main method.
- Inside the main method, ask the user for a file name and store the response in a variable.
- Create a File object using the file name that is entered by the user in the previous step.

Throwable class

Exceptions

What if someone tosses you a hot potato?

- Drop it
- Toss it to someone else
- Figure out what to do with the potato

What if someone tosses you a problem in the computer (Throwable class)

- Drop it (crash)

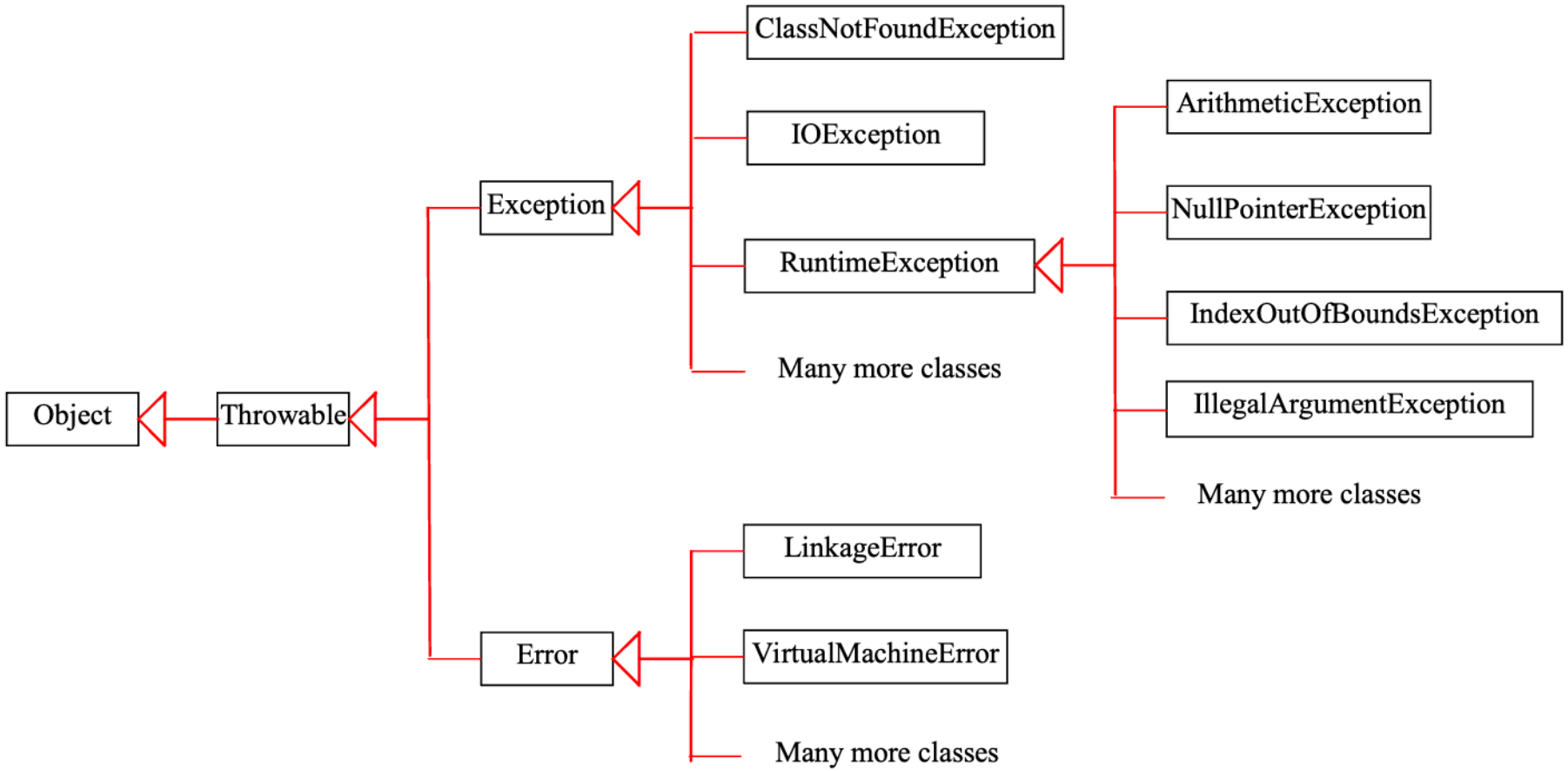
- Toss it to someone else (throw)

- Figure out what to do with the Throwable (handle)

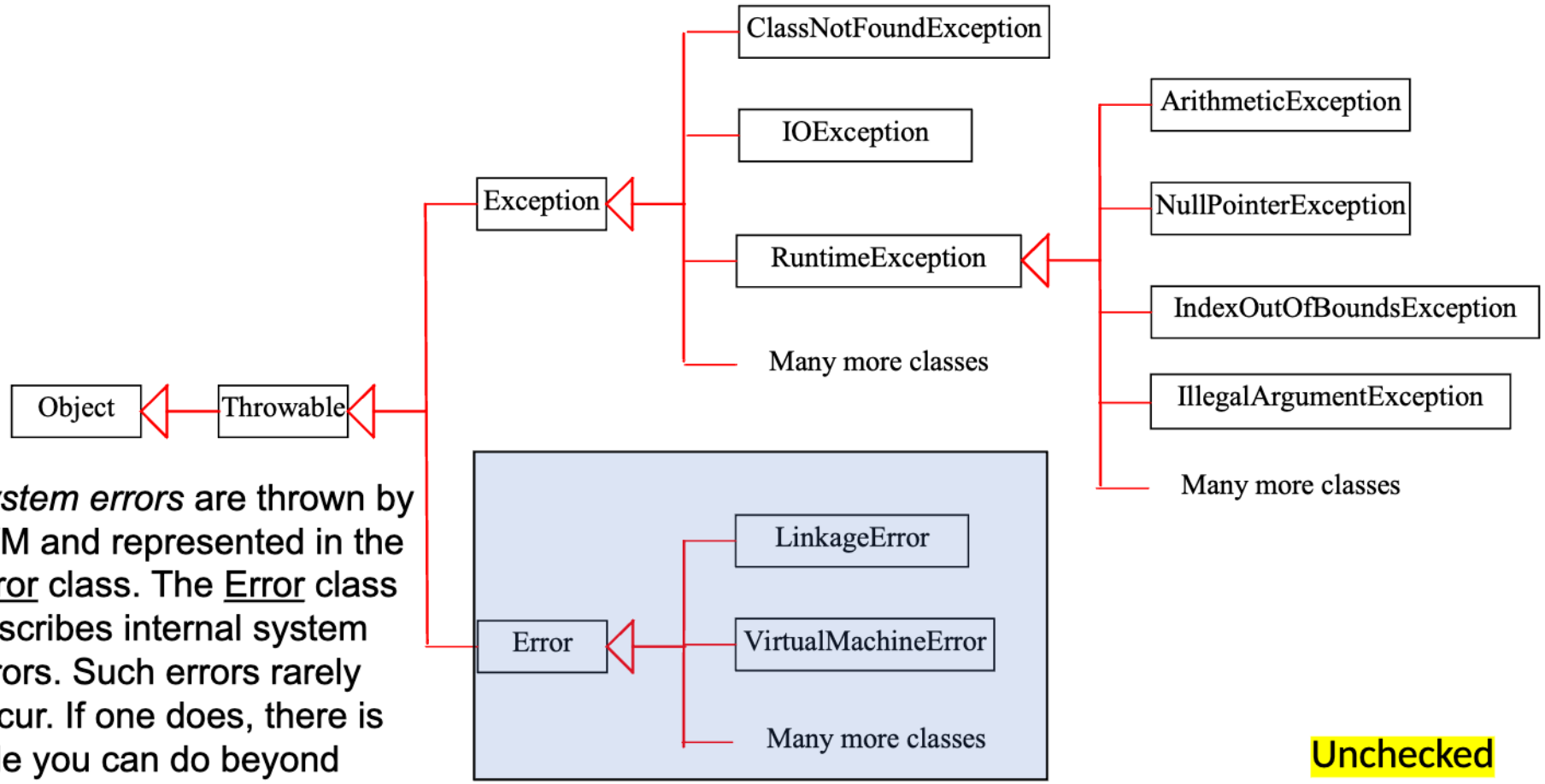
**When a problem occurs, the Java virtual machine creates an object of class Throwable which holds information about the problem.


ORACLE

Summary of Oracle discussion



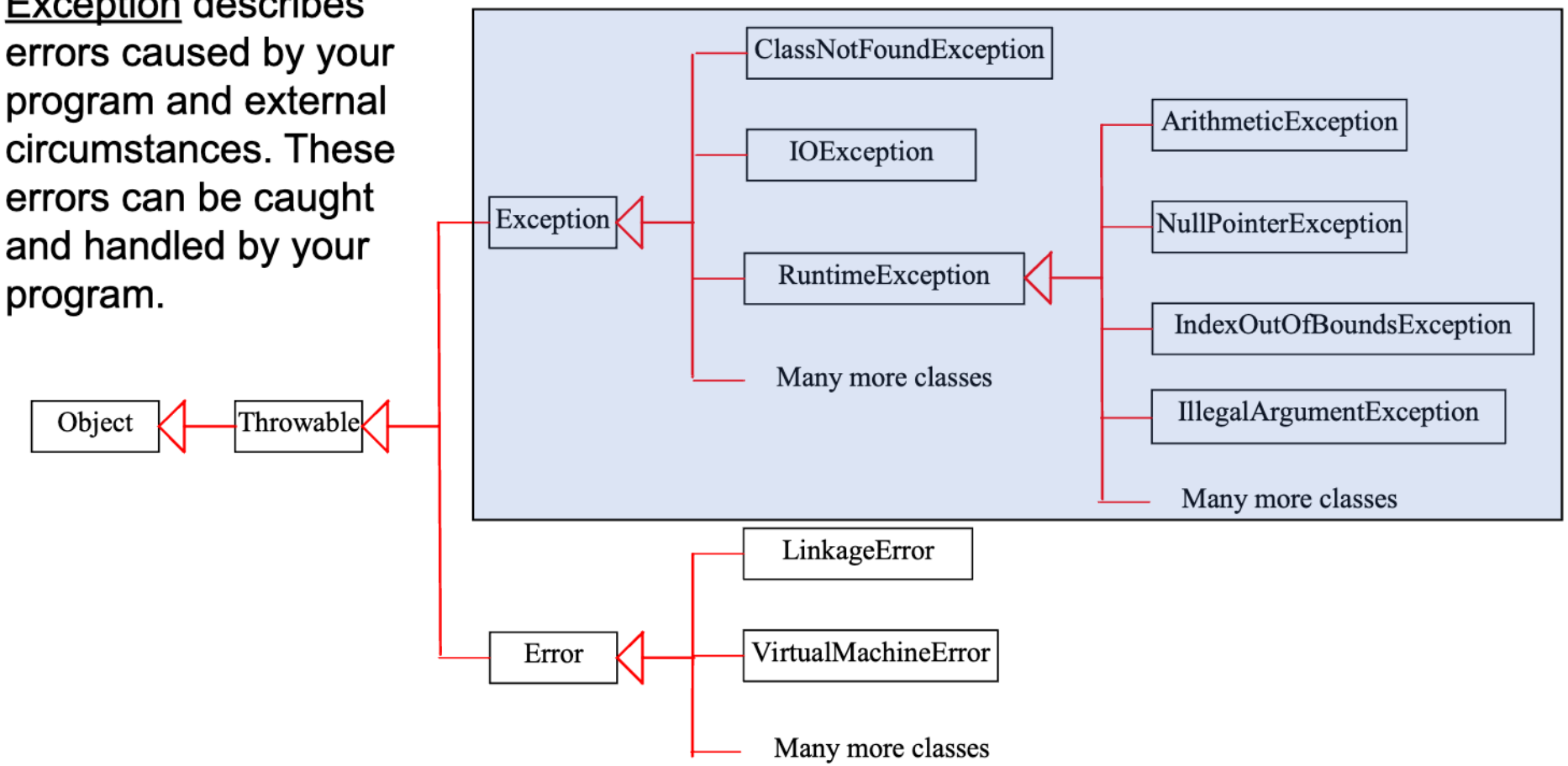
System Errors



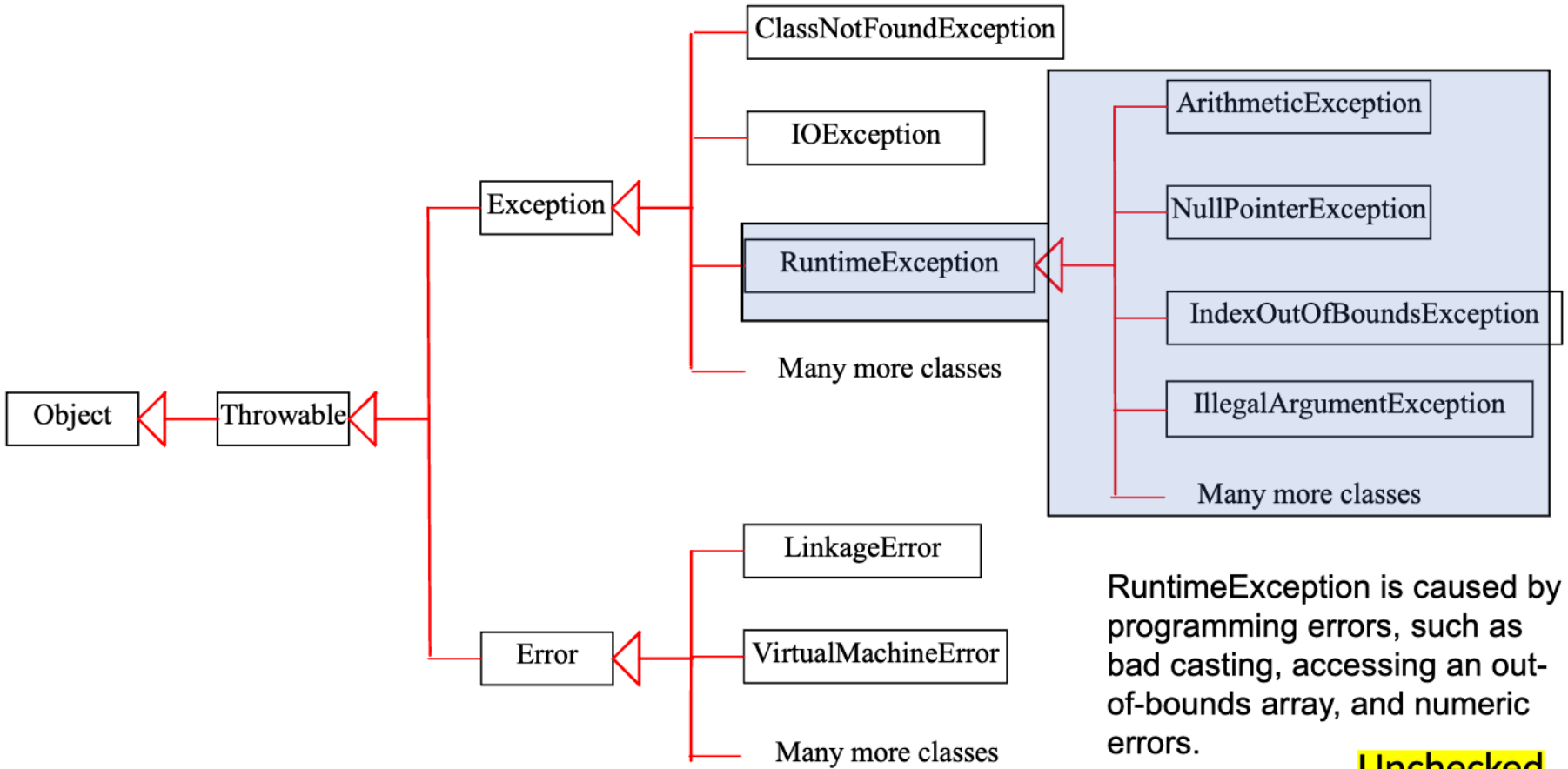
System errors are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

Exceptions

Exception describes errors caused by your program and external circumstances. These errors can be caught and handled by your program.



Runtime Exceptions (Most Common)



RuntimeException is caused by programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors.

Unchecked

Try it!

What exception is thrown by each of the following?

```
1- System.out.println(1/0);


2- int[] list = new int[5];
System.out.println(list[5]);


3- String s = "abc";
System.out.println(s.charAt(3));


4- Student s = null;
System.out.println(s.toString());
```

Input from a file

Use Scanner class

`hasNext` methods

- The `hasNext` methods are used to read files with different data types
- This table shows the different `hasNext` methods and corresponding `next`
- The `hasNext()` and `next()` methods work with *tokens* which are groups of characters delimited (surrounded) by one or more spaces.

Methods of Scanner	
HasNext Method	Reading Method
hasNext()	next()
hasNextDouble()	nextDouble()
hasNextFloat()	nextFloat()
hasNextInt()	nextInt()
hasNextLine()	nextLine()
hasNextLong()	nextLong()

No data consumed with `hasNext`

- No data is "consumed" when a `hasNext()` method is used.
- This means that the **next** token in the file can be tested by using several `hasNext()` methods to determine what it is.

```
// is there more data to process?
while( scan.hasNext() )    {
  if (scan.hasNextInt()){
      int num = scan.nextInt();
      System.out.println("Integer: " + num);
  }else if (scan.hasNextDouble()){
      double num = scan.nextDouble();
      System.out.println("Double: " + num);
  }else if (scan.hasNextLine()){
      String str = scan.nextLine();
      System.out.println("String: " + str);
  }
}
```

Try it!

- Create a Scanner object to read the file object created in the previous step.
- Add the throws declaration to the main method. (Hover over the error and chose the option for throwing the exception).
- Write a while loop that uses hasNextLine() as a condition. Inside the loop, read one line from the file using nextLine() and print that line to the screen.
- Write another while loop that uses hasNext() as a condition. Inside the loop, read one entry from the file using next() and print that to the screen.

Note that we are not running this yet – since there isn't a file to read

Try it!

- Download the data file, called StatePopulationData.txt, from D2L.
- This file includes population information about the 50 USA states. Each line in the file consists of three fields where the first number is the overall state population, the second number is the adult population and the third field is the state name. Open the file and look at it. Do you understand how the information is being presented? In other words, can you find the adult population of California? It is 30476517.
- Run the program, enter _____StatePopulationData.txt when prompted. (The blank is for the file path specific to where your computer has the file stored. How many times did the file print?
- The second loop will not read anything because the Scanner object is already at the end of the file. In order to read the file again, you will have to create a new Scanner. How does this differ from reading information from the keyboard? Why do you think that is?
- Between the two loops, close the scanner using the method close(). Then create a new scanner to read the file again. You can use the same variable name as before. For example, if my Scanner was called fileScan, and my file was called inputFile, I would write:
  - fileScan.close();
  - fileScan = new Scanner(inputFile);

**********IMPORTANT: Do NOT close the scanner for system.in !!!!!! ********

- Rerun the program. How does the output from the two loops vary? Do you understand why?

Writing to a file

Using PrintWriter

Try it!

- Inside Eclipse, open the driver class for “StateApplication”.
- Create a PrintWriter object.
- Change the print statement so that the text prints to a file instead of the console.
- Refresh the project. Do you see a copy of your input file?

Create objects

Using the file data to create objects

Try it!

- In the main method, create an array to hold 50 State objects.
- Declare and initialize a counter for the number of states currently in the array.
- Modify the body of the while loop so that the information contained in each line is used to create a State object. Store the State in the array you created above.
  - **Make sure you are using nextline() and not next() for the name of the state since some of them have two words instead of one.
- Use a for loop to print the contents of the array.
- Run the program to make sure it works.

Try/Catch

Taking care of the problem

Example: What happens if user enter 0 for `num2`?

```
import java.util.Scanner;
public class ExceptionHandling {

        public static void main(String[] args) {
                Scanner input = new Scanner(System.in);
                System.out.println("Enter two numbers:");
                int num1 = input.nextInt();
                int num2 = input.nextInt();

                System.out.println("Division result: "+num1/num2);
                System.out.println("Multiplication result: "+num1*num2);
        }
}
```

Example 1: How to avoid breaking the program?

```
import java.util.Scanner;
public class ExceptionHandling {

     public static void main(String[] args) {
          Scanner input = new Scanner(System.in);
          System.out.println("Enter two numbers:");
          int num1 = input.nextInt();
          int num2 = input.nextInt();

          try{
               System.out.println(num1/num2);
          }catch (Exception e){
               System.out.println("Cannot divide by zero");
          }

          System.out.println(num1*num2);
     }
}
```

`Exception` Objects

- When a `catch` block receives control it has a reference to an object of class `Exception` (or a subclass of Exception).

- The parameter of the `catch{}` block (**exp** in the below code) refers to the Exception object.

```
try {
      // statements, some of which might throw an exception
} catch ( SomeExceptionType exp ) {
      // statements to handle this type of exception
}
```

`Exception` objects (Continued)

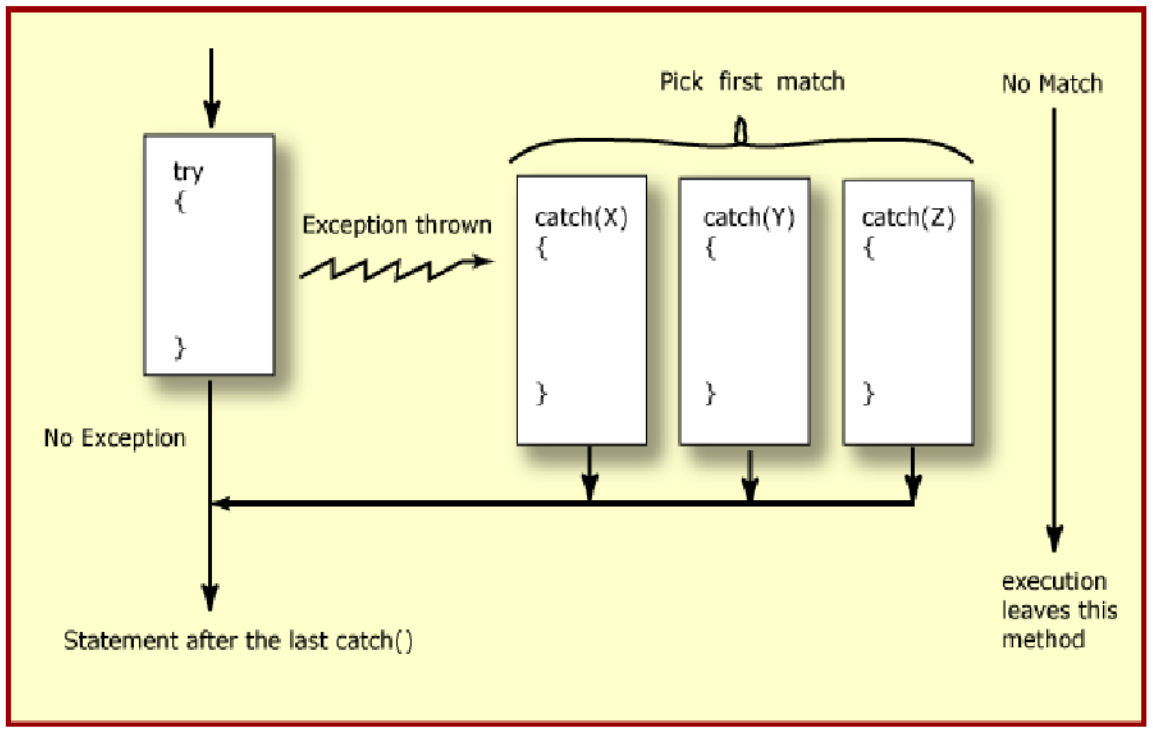
- Exception objects contain data and methods, as does any object. Here are some methods:
  - `public void` **`printStackTrace()`** — prints a stack trace, a list that shows the sequence of method calls up to this exception.
  - `public String` **`getMessage()`** — returns a string that may describe what went wrong.

```
try {
      // statements, some of which might throw an exception
} catch ( SomeExceptionType exp ) {
      System.out.println(exp.getMessage());
      exp.printStackTrace();

}
```

Syntax of `try-catch` statement

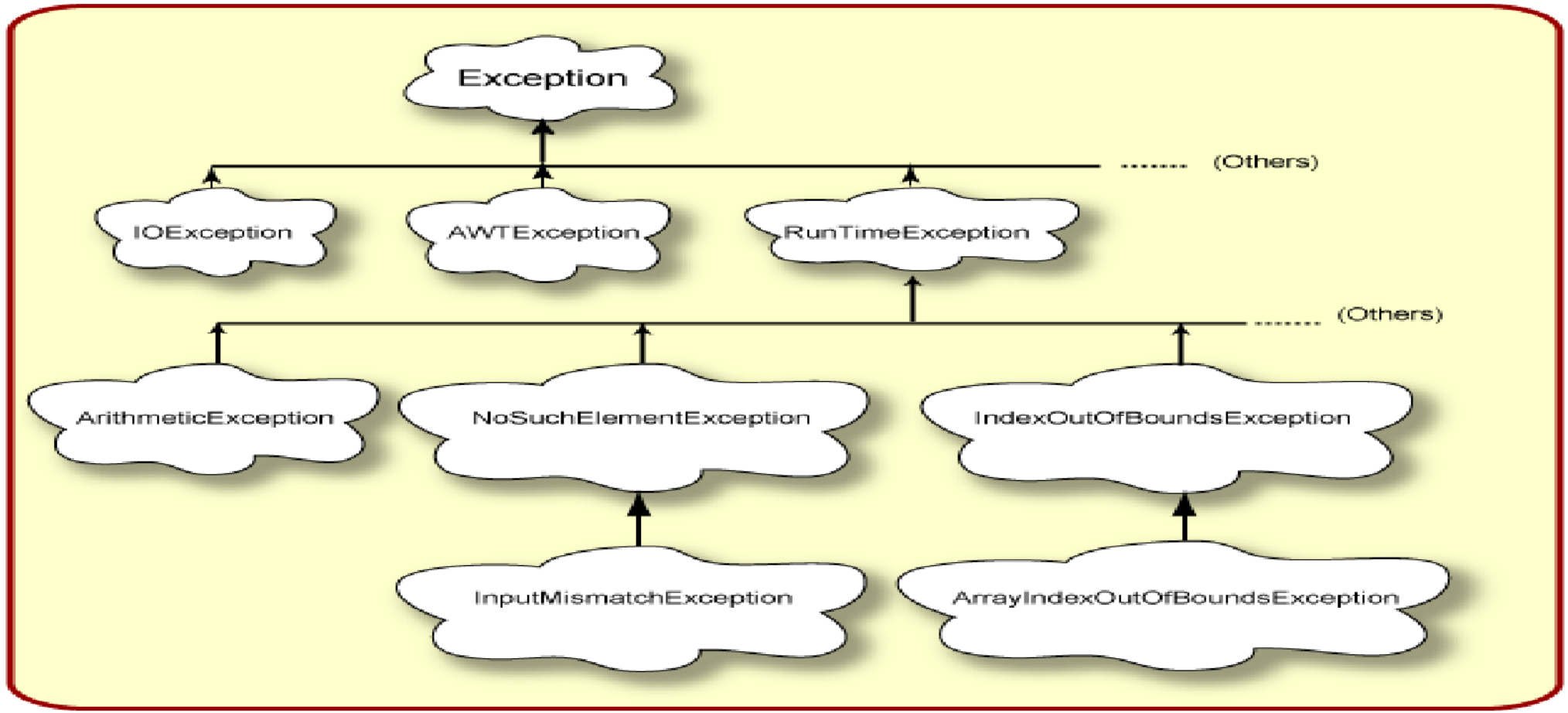
```
try {
      // statements which might throw various types of exceptions
} catch ( SomeExceptionType ex ) {
      // statements to handle this type SomeExceptionType
} catch ( AnotherExceptionType ex ) {
      // statements to handle this AnotherExceptionType
} catch ( YetAnotherExceptionType ex ) {
      // statements to handle this YetAnotherExceptionType
}

// Statements following the structure
```

How `try-catch` work

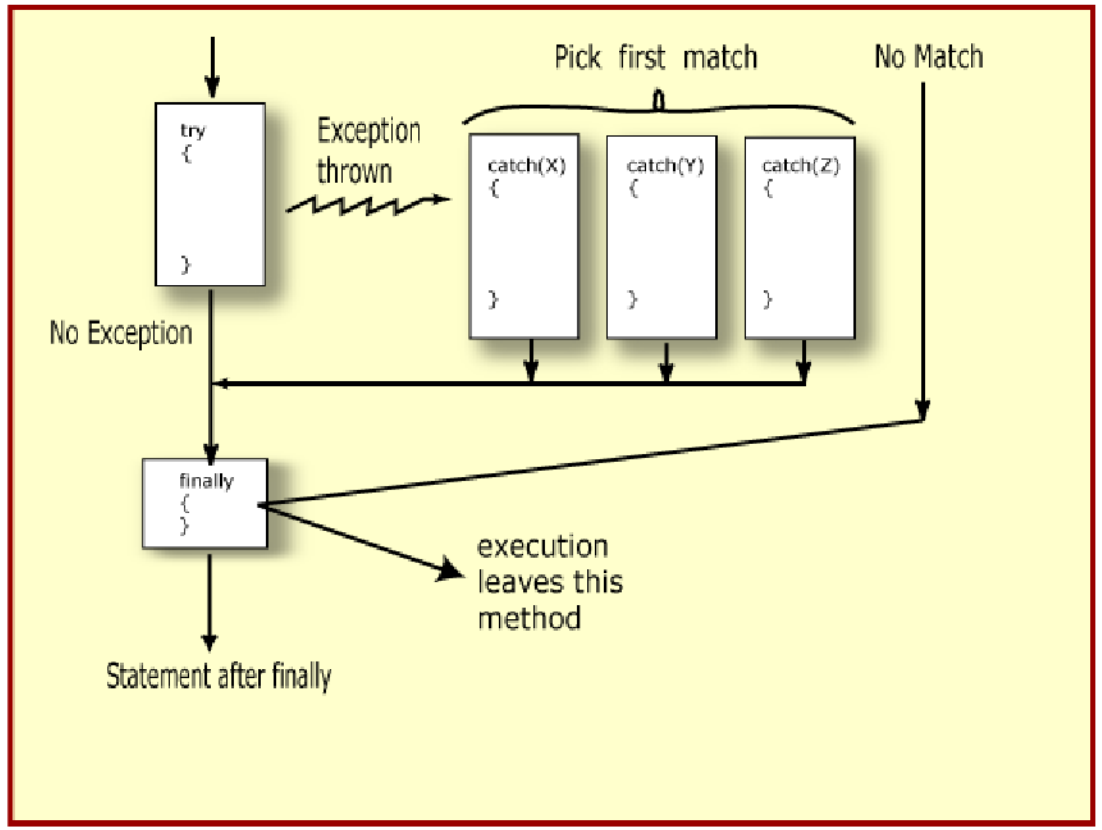


- The **first** catch block to match the type of Exception thrown gets control.
- The **most specific Exception types** should appear first in the structure, followed by the more general Exception types.
- In arranging the try blocks, a child class should appear before any of its ancestors. If class A is not an ancestor or descendant of class B, then it doesn't matter which appears first.

Hierarchy of Exceptions



The `finally{}` block



- Sometimes you need to have a block of code that should always execute, no matter which Exception was thrown and even if no exception.
- By using a finally block, you can ensure that some statements will always run, no matter what happened.

try-catch-finally syntax

```
try {
    // statements, some of which might throw an exception
} catch ( SomeExceptionType ex ) {
    // statements to handle this type of exception
} catch ( AnotherExceptionType ex ) {
    // statements to handle this type of exception
}
// more catch blocks
finally {
    // statements which will execute no matter how the try
    //block was exited and no matter which catch block was executed (if any)
}

// Statements following the structure
```

Checked vs. Unchecked exceptions

- Some exception types are **checked exceptions**, which means that a method MUST do something about them.
  - The compiler checks to make sure that this is done.
  - There are two things a method can do with a checked exception. It can:
    - handle the exception in a `catch{}` block, or
    - `throw` the exception to the caller of the method.
- For example, an `IOException` is a checked exception.
- `RunTimeExceptions` (e.g., `ArrayIndexOutofBounds`) are **unchecked exception**.

Exception **CHECKED**

IOException **CHECKED**

AWTException **CHECKED**

RunTimeException

- ArithmeticException
- IllegalArgumentException
- NumberFormatException
- IndexOutOfBoundsException
- ArrayIndexOutOfBoundsException
- NoSuchElementException
- InputMismatchException
- Others

Example 2: `InputMismatchException`

- Assume you wrote a Java program that has the following line:

```
int num = scan.nextInt()
```

- What happens if the user enters `Hello` while the program is expected an integer?

- A problem happens because `scan.nextInt()` cannot convert `Hello` into an `int`.

- When `nextInt()` is called, then the program **throws** an `InputMismatchException`.

Throwing a method-specific Exception

- A program or method may **throw** an exception when it detects a problem.

```
public static int calcInsurance( int birthYear ) throws Exception {

        final int currentYear = 2000;
        int age = currentYear - birthYear;
        if ( age < 16 ) {
                throw new Exception("Age is: " + age );
        } else {
                int drivenYears = age - 16;
                if ( drivenYears < 4 )
                        return 1000;
                else
                        return 600;
        }
}
```

```
public static void main ( String[] a ) {
        Scanner scan = new Scanner( System.in );
        System.out.println("Enter birth year:");

        int inData = scan.nextInt();
        try {
                System.out.println( "Your
insurance is: " + calcInsurance( inData ) );
        } catch ( Exception oops ) {

System.out.println( oops.getMessage() + " Too young
for insurance!" ); } }
```

Try it!

- Delete the throws clause from the main method header.
- Surround the file Scanner object and any code that uses that scanner with a try/catch.
- Run the program to make sure nothing broke.