Lecture 19
Programming Exceptions
CSE11 Fall 2013

When Things go Wrong

- We've seen a number of run time errors
  - Array Index out of Bounds
    - e.g., `Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2 at TestWNS.main(TestWNS.java:14)`
  - String Index out of Bounds
  - Null Pointers <-- what does this mean?
  - String format exceptions
- These are called Exceptions
  - They are logic or other kind errors in your program

Throwing and Catching Exceptions

- An exception is an object in java
- A <u>method</u> can generate an exception and tell whomever has invoked it
    - This is called “throwing an exception”
    - **methods** throw exceptions, classes do not
- A method can also intercept an exception and process (handle) it without the program failing.
    - This is an “exception handler”
    - The method is said to “catch the exception”

try ... catch

- A block of code can be “tried”
  - if NO exceptions occur, the catch block (exception handler) is not invoked
  - if an exception happens that is defined in the catch block(s), the program can gracefully handle the exception.

```
try {
 ..... code under normal circumstances
}
catch (Exception e)
{
.... code that executes when exception of type Exception occurs
};
```

A Very Simple Exception Handler

```
import java.util.*;
public class SimpleException {
    public static void main(String[] args)
    {
        Scanner parser=new Scanner(System.in);
        String input;
        String [] vals;
        System.out.println("Enter numbers: \n");
        try {
            while ((input = parser.nextLine()) != null)
            {
                System.out.format("I read number: %f \n",
                    Double.parseDouble(input));
            }
        }
        catch (NoSuchElementException err) {};
    }
}
```

handles when we no more input. Not bad numbers

What happens when we type in a "bad" number

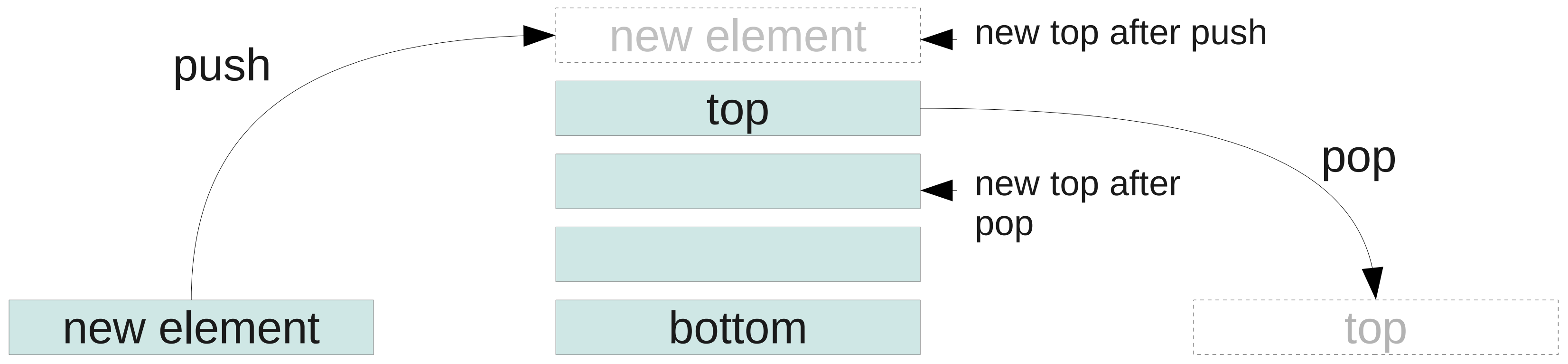
```
$ java SimpleException
Enter numbers:

123.45
I read number: 123.450000
this is not a number
Exception in thread "main" java.lang.NumberFormatException: For input string:
    at sun.misc.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:1241
    at java.lang.Double.parseDouble(Double.java:540)
    at SimpleException.main(SimpleException.java:12)
```

Can we clean this up a bit with a better exception handler?

Stack trace: our code on line 12, was calling Double line 540, was calling readJavaFormatString at line 1241

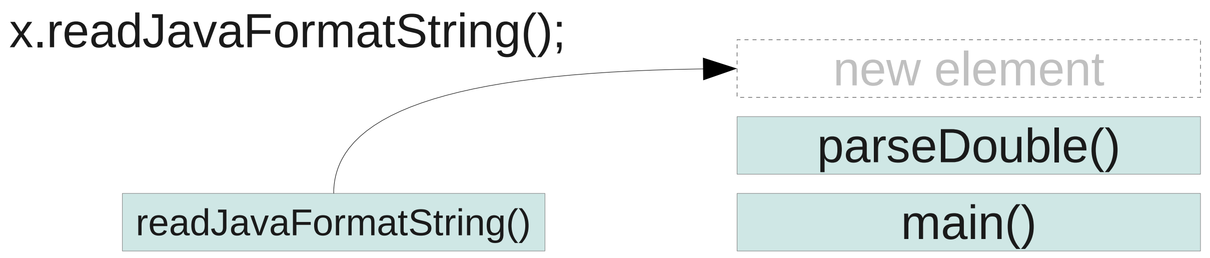
A Digression on Stack Traces

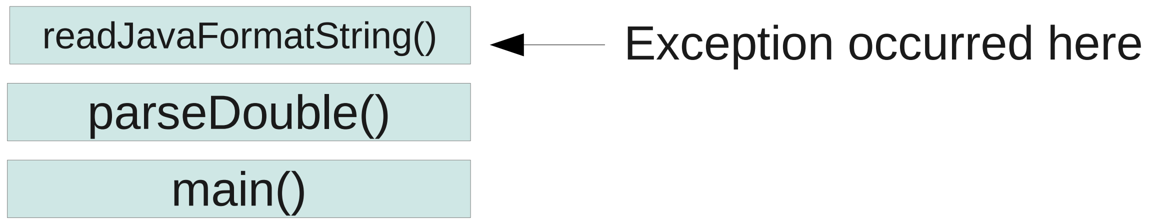
- What is a stack?
  - it is like a stack of dishes,
  - you can place something on top (push)
  - You can remove only the top item (pop)
  - you can tell when the stack is empty (but not how tall it is)



The Call Stack

- Every time a method is invoked (called), a record of that call is placed on the call stack
- Local variables are allocated from the stack, too
- Nothing below the top can return until the top returns.

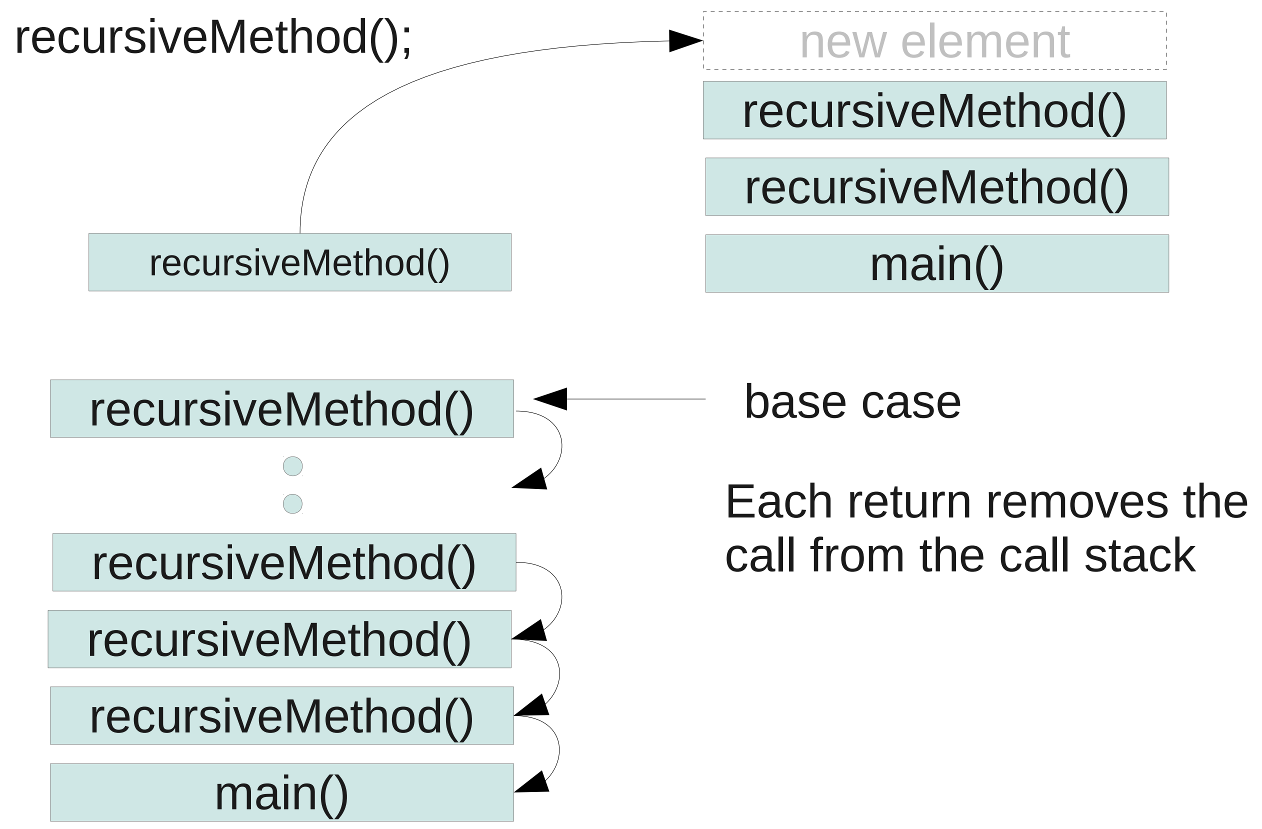




A print out of the call stack at the time of an exception is call the **stack trace**

A Recursive Call Stack

- Same thing happens with recursive calls, you just have many copies on the call stack
- When the recursion hits the base case, the calls below it can return one-by-one



Modified Simple Exception Handler

```
import java.util.*;
public class SimpleException2 {
    public static void main(String[] args)
    {
        Scanner parser=new Scanner(System.in);
        String input ="";
        String [] vals;
        System.out.println("Enter numbers: \n");
        try {
            while ((input = parser.nextLine()) != null)
            {
                System.out.format("I read number: %
                    Double.parseDouble(input));
            }
        }
        catch (NoSuchElementException err) {}
        catch (NumberFormatException err)
        {
            System.out.format("I could not understand '%s' as a number\
            System.out.format(" Error '%s' had reason '%s' \n",
                err.getClass().getName(), err.getMessage());
        }
    }
}
```

handles when we no more input. Not bad numbers

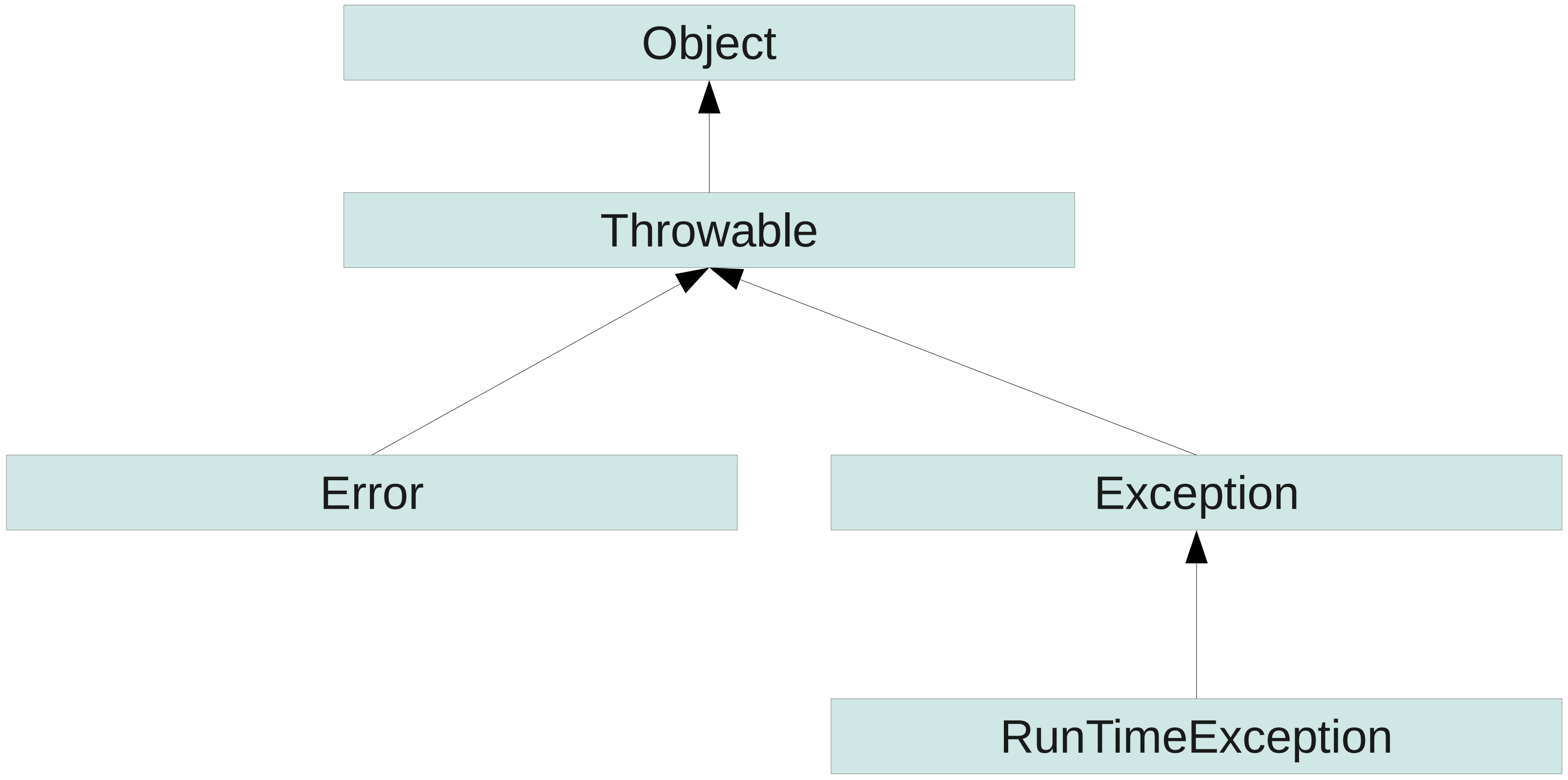
handles format errors

Multiple Catches for the same try block

- One can handle multiple exceptions with multiple catch blocks
- They are processed in the order they are defined
- The first catch block that matches the exception is the <u>first and only</u> catch block to execute.
- Rule of thumb: catch specialized errors first and more general errors later
  - Why? Exceptions are Objects and are inherited

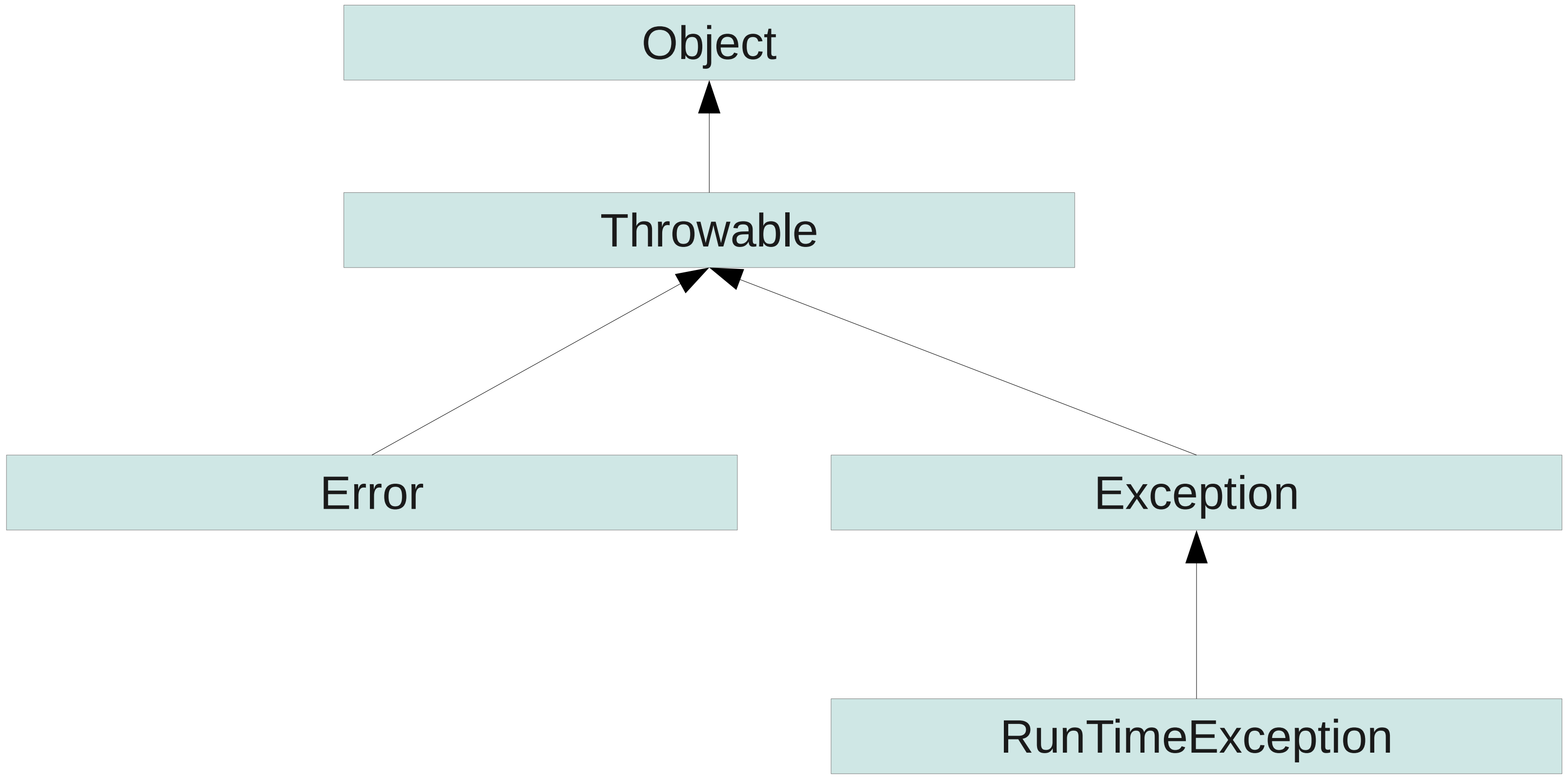
Exceptions create an inheritance Hierarchy

- HIGHEST Level view (Important for Checked vs. Unchecked Exceptions)



Exceptions create an inheritance Hierarchy

- HIGHEST Level view (Important for Checked vs. Unchecked Exceptions)



What are some “Throwable”s under the Error Class

- VirtualMachineError (Java Interpreter runtime)
  - OutOfMemoryError
  - StackOverflowError

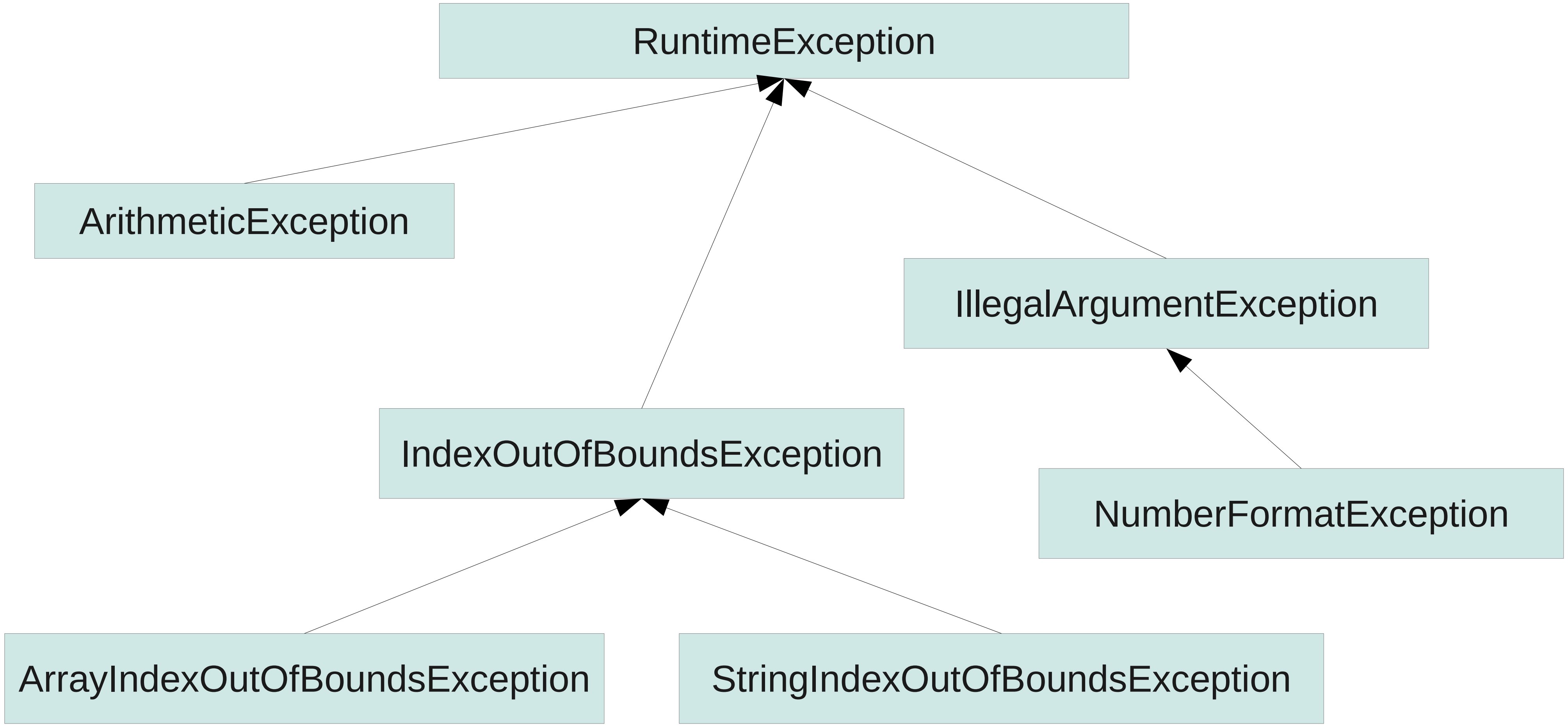
- AWTError (Problems with AWT Engine)
- LinkageError (Problems with finding other classes)
  - NoClassDefError
  - ClassFormatError

- Generally, System errors.
- Usually, we don't catch these errors

What are Some Runtime Exceptions

- ArithmeticException
- ArrayStoreException
- ClassCastException
- EmptyStackException
- IllegalArgumentException
  - IllegalParameterException
  - IllegalThreadStateException
  - NumberFormatException
- IndexOutOfBoundsException
- MissingResourceException
- NegativeArraySizeException
- NoSuchElementException
- NullPointerException
- RasterFormatException
- SecurityException
- SystemException
- UndeclaredThrowableException
- UnsupportedOperationException

Some Runtime Exceptions


RuntimeException
ArithmeticException
IllegalArgumentException
IndexOutOfBoundsException
NumberFormatException
ArrayIndexOutOfBoundsException
StringIndexOutOfBoundsException

Are there other reasons to program with Exceptions?

- Yes!
  - It can be much simpler to program the main logic of the code WITHOUT testing for all special cases at every step
  - Then catch exceptions when they occur
  - Basic idea is the code runs properly most of the time, and code logic should favor getting the common case “right” (and debugged)

Checked vs. Unchecked Exceptions

- Java exception classes are categorized as either "checked" or "unchecked".
  - categorization affects compile-time behavior only;
  - Exceptions are handled identically at runtime. Java determines determine in which category each exception is defined.
- An unchecked exception is any class that IS A SUBCLASS of RuntimeException (as well as RuntimeException itself).
- A checked exception is any class that is NOT A SUBCLASS of RuntimeException.

Some Checked Exceptions

- IOException
- ChangedCharSetException
- CharConversionException
- EOFException
- FileNotFoundException
- InterruptedIOException
- MalformedURLException
- ObjectStreamException
- ProtocolException
- RemoteException
- TooManyListenersException
- UnsupportedAudioFileException

Java Complains about <u>Checked</u> Exceptions

- With unchecked exceptions, we don't have to do anything, they will propagate
- If one calls (invokes) methods that throw checked exceptions
  - The caller. i.e., the <u>code that invokes</u> the method that throws a checked exception must either
    - explicitly catch the checked exception
    - (re)throw the exception via `throws` method modifier
- It's a compiler error if you do NOT catch or re-throw a Checked Exception.

finally

- not covered in book
- A finally clause always executes after try...catch block.
  - Enables clean-up processing after either normal operation OR an exception has occurred
- The default finally block is empty

```
try {
... normal code
}
catch (ExceptionClass1 err) { ... exception code }
catch (ExceptionClass2 err) { ... exception code }
finally {
   ..... clean up code
};
```

Always Executes

Some clarifying exercises

Is there anything wrong with the following exception handler
as written? Will this code compile?

```
try {

} catch (Exception e) {

} catch (ArithmeticException a) {

}
```

What exception types can be caught by the following handler?

```
catch (Exception e) {

}
```

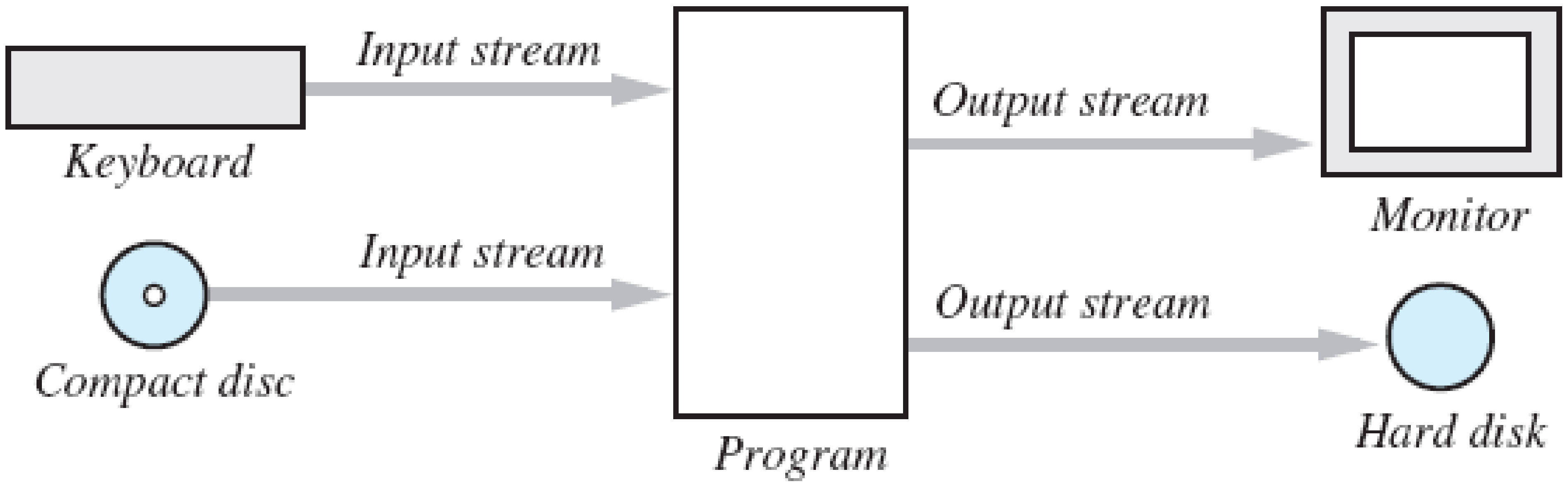
What is “bad” about using this type of exception handler?

The Concept of a Stream

- Use of files
  - Store Java classes, programs
  - Store pictures, music, videos
  - Can also use files to store program I/O
- A *stream* is a flow of input or output data
  - Characters
  - Numbers
  - Bytes

The Concept of a Stream

- Streams are implemented as objects of special stream classes
  - Class **`Scanner`**
  - Object **`System.out`**



3 Streams in Unix

- Standard input (stdin in C)
- Standard output (stdout in C)
- Standard error (stderr in C)

- These are available in the unix shell

  ```
  % program < inputfile
  ```

  - Send the input file as the stdin to the program

  ```
  % program > outputfile
  ```

  - Send the output of a program to a file

  ```
  % program1 | program2
  ```

  - Send the output of program1 to the input of program2
    - (this is called a pipe)

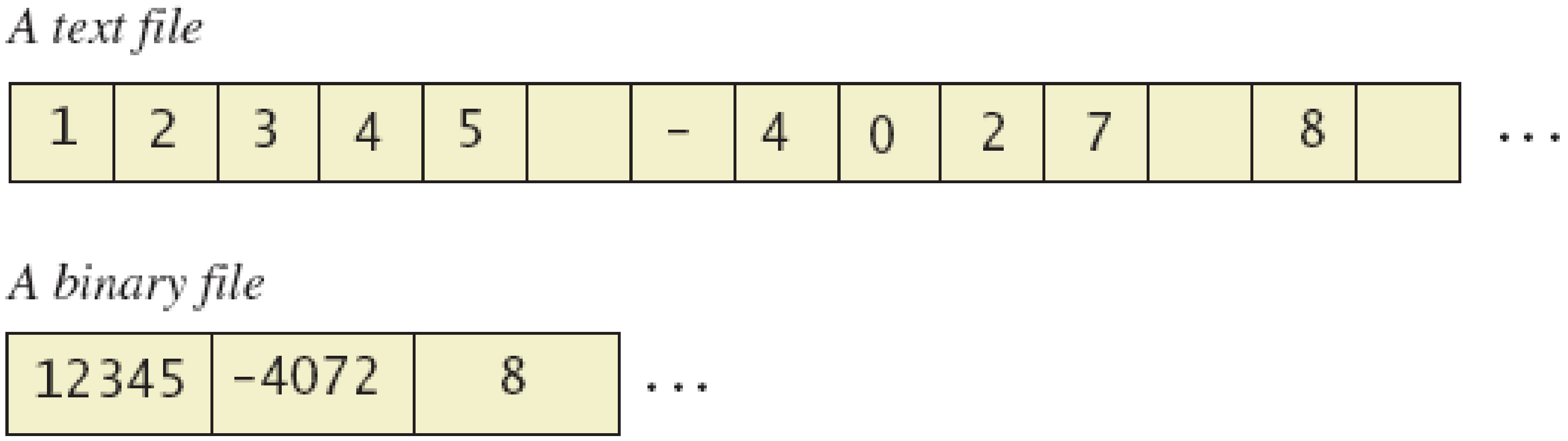
Why Use Files for I/O

- Keyboard input, screen output deal with temporary data
  - When program ends, data is gone
- Data in a file remains after program ends
  - Can be used next time program runs
  - Can be used by another program

Text Files and Binary Files

- All data in files stored as binary digits
  - Long series of zeros and ones
- Files treated as sequence of characters called *text files*
  - Java program source code
  - Can be viewed, edited with text editor
- All other files are called *binary files*
  - Movie, music files
  - Access requires specialized program

Text Files and Binary Files



Creating a Text File

- Class `PrintWriter` defines methods needed to create and write to a text file
  - Must import package `java.io`
- To open the file
  - Declare *stream variable* for referencing the stream
  - Invoke `PrintWriter` constructor, pass file name as argument
  - Requires `try` and `catch` blocks

Creating a Text File

- File is empty initially
    - May now be written to with method `println`
- Data goes initially to memory buffer
    - When buffer full, goes to file
- Closing file empties buffer, disconnects from stream

Creating a Text File

- When creating a file
  - Inform the user of ongoing I/O events, program should not be "silent"
- A file has two names in the program
  - File name used by the operating system
  - The stream name variable
- Opening, writing to file overwrites pre-existing file in directory