G51OOP

5. Exceptions

Object Oriented Programming

Colin Higgins & Steve bagley

Exceptions

- Java (generally) uses *Exceptions* to inform about unexpected conditions
- When an error occurs, an *Exception* is thrown
- Java runtime then jumps to a block of code that has been registered as handling that type of exception

Exception Handlers

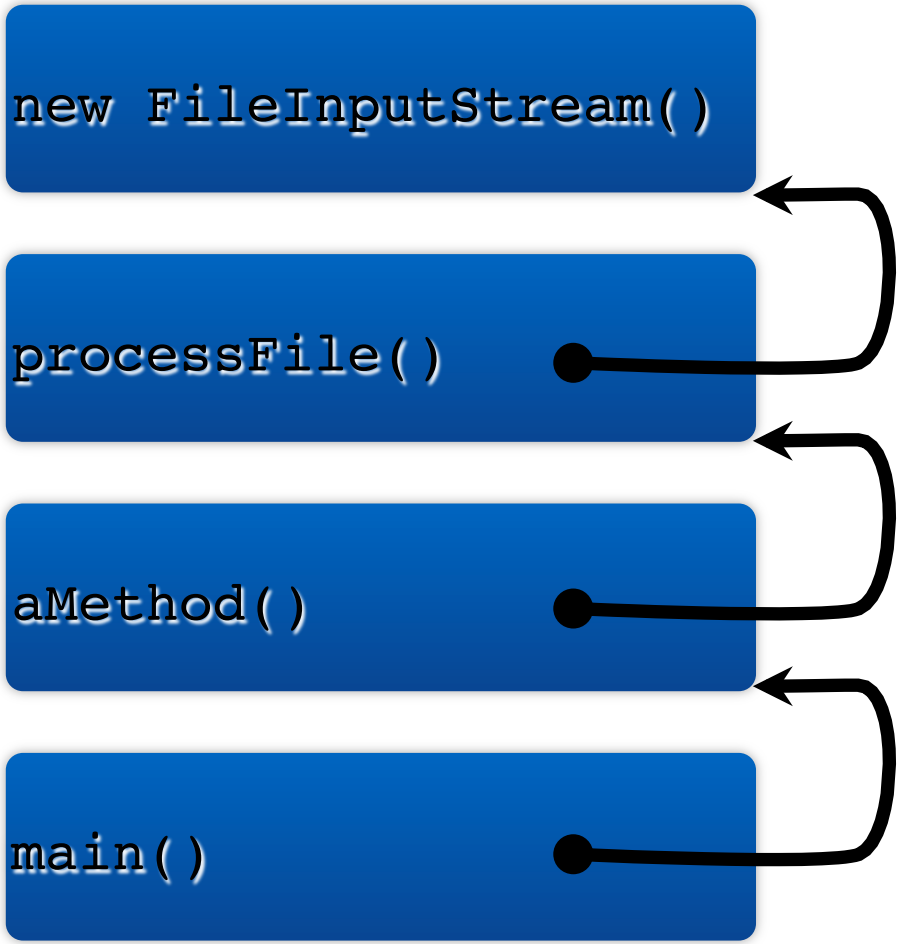
- Exceptions bubble up the call stack until they find a handler for that type of exception
- If not, the program crashes
- Exception Handlers specified by using `try { … } catch { … }` blocks

```
Try {
   FileInputStream fis = new FileInputStream(“file.txt”);
} catch(Exception e) {
   System.err.println(“You bozo, you’ve only gone and deleted
                      file.txt”);
}
```

Call Stack

- Call Stack is just the sequence of methods that have been called since the program started
- Already seen this when spotting errors while debugging

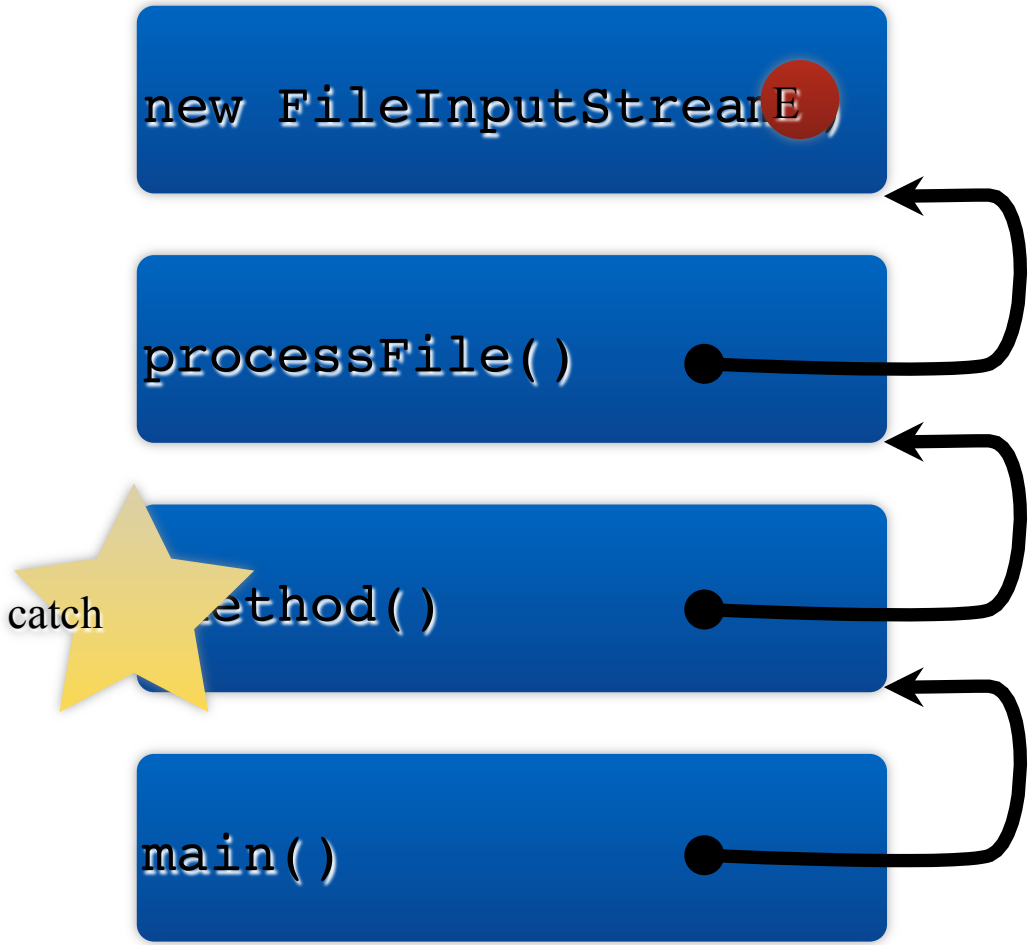
Call Stack



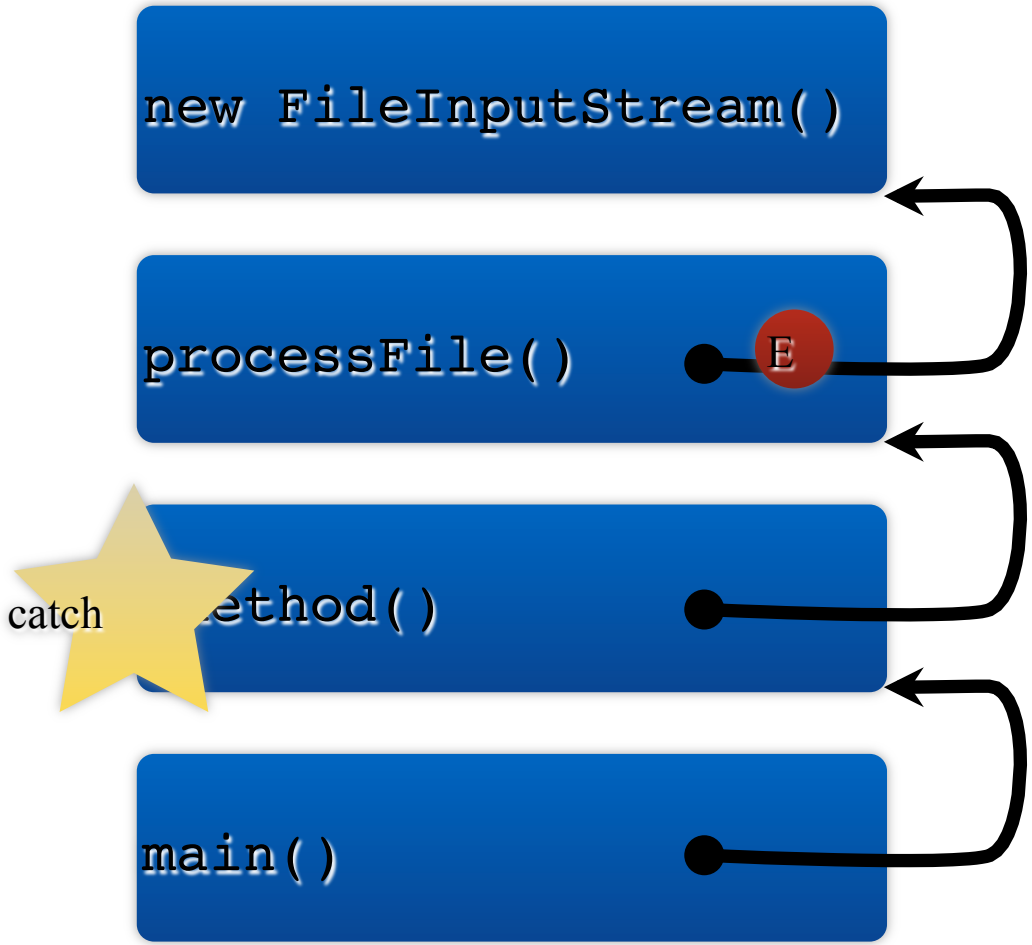
Exceptions and Call Stack

- When an exception occurs it first looks in the current method to see if there is a handler
- If not, it looks in the method that called it
- And so on, till it eventually gets to the bottom of the stack
- And the program will then crash

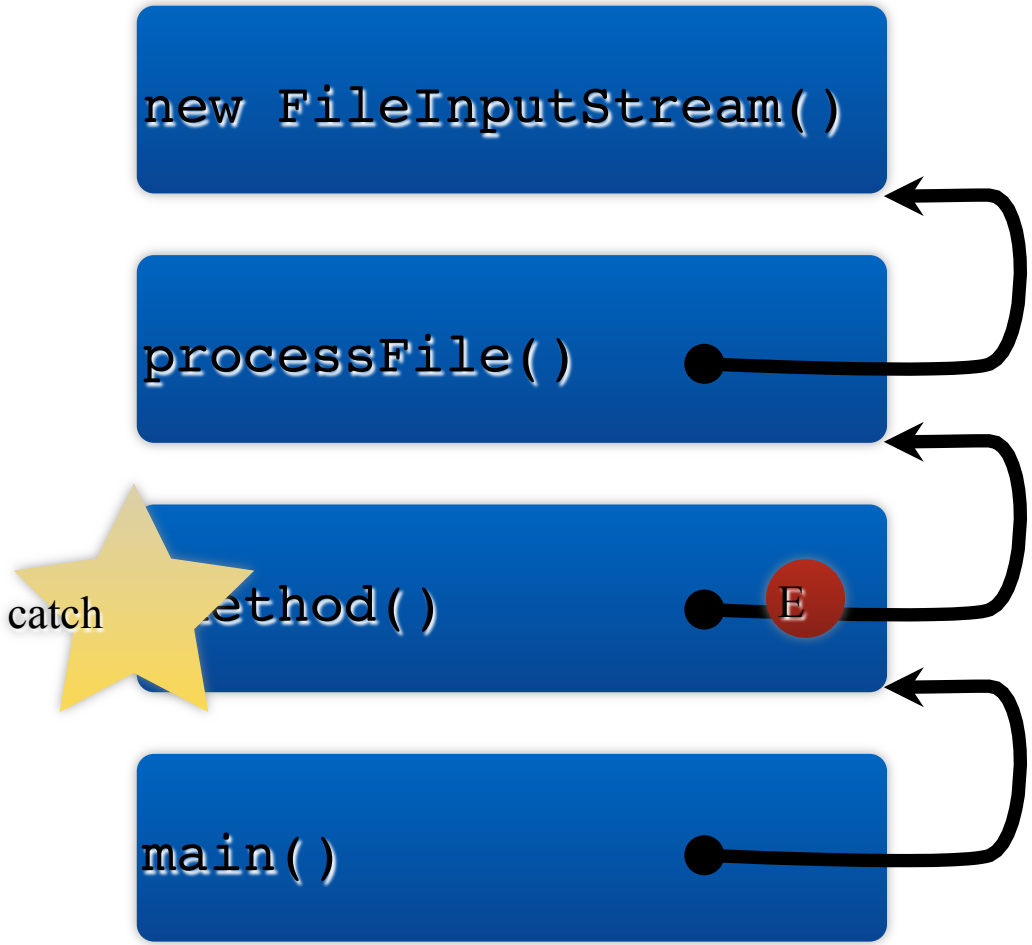
Call stack



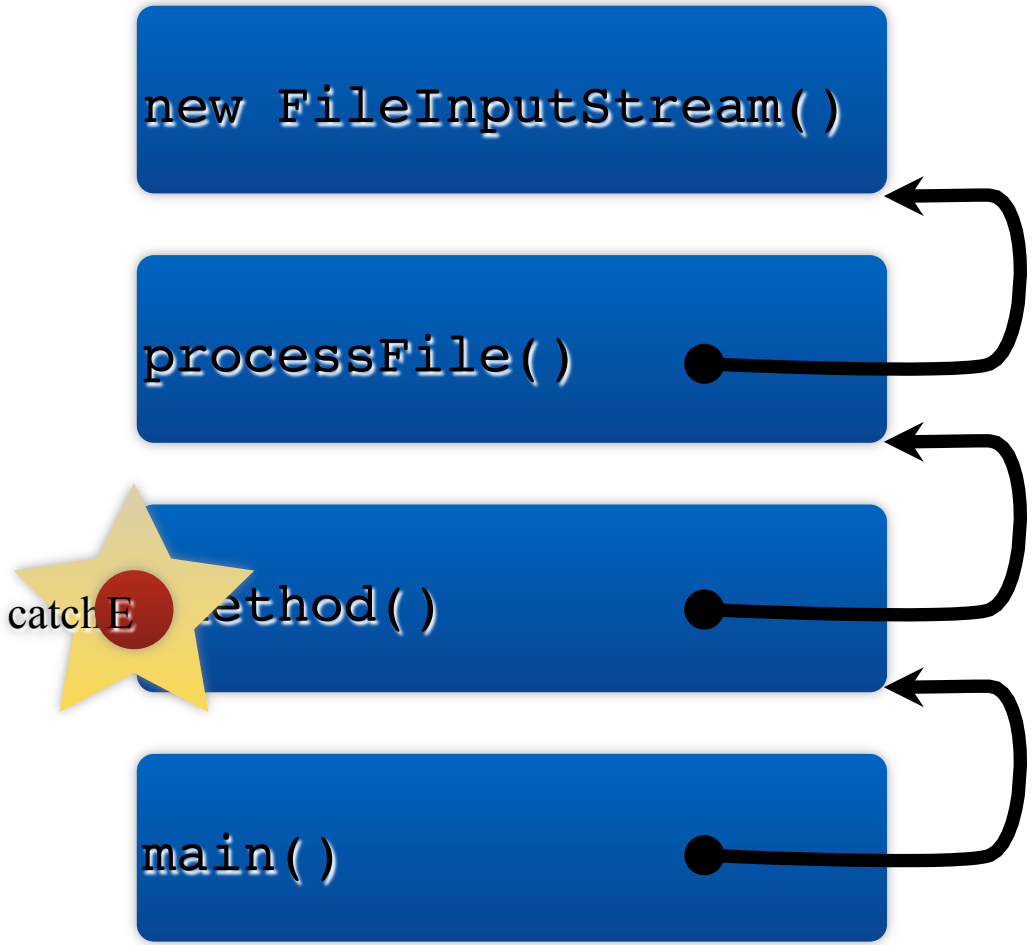
Call stack



Call stack



Call stack



Three Kinds of Exceptions

- Java has three kinds of Exceptions:
  - Checked Exceptions
    - things the application should recover from
  - Errors
    - things the application probably can't recover from
  - Runtime Exceptions
    - things happening from the Java Runtime

Catch or specify

- Checked Exceptions are subject to the *catch or specify* requirement
- Code that generates an exception must
  - Be in a `try … catch` block
  - Or be in a method which specifies it *throws* an exception
- Or the Java compiler will whinge…

Exceptions as Objects

- Exceptions are just objects
- Different exceptions are handled as objects of different classes
- All Exceptions inherit from the class `java.lang.Exception`
- Easy to create our own by sub-classing
  - (see inheritance lectures)

Handling Exceptions

- Seen this ‘in the wild’ already
- We put code that might generate an exception inside a `try {}` block
- This is followed by one or more `catch {}` blocks which handle different types of exceptions

```
try {
   FileInputStream fis = new FileInputStream("file.txt");
} catch(FileNotFoundException fe) {
   System.err.println("You bozo, you've only gone and deleted
       file.txt");
} catch(IOException e) {
   System.err.println("Some IO Exception happened");
}
```

Throws Exception

- The alternative is to say that the method *throws* an exception
- Which defers processing to some other bit of code
- Do this using the `throws` keyword
- Of course, this method will also need to be in a `try {} catch {}` now…

```
void processFile(String someFile) throws IOException,
       FileNotFoundException {
    FileInputStream fis = new FileInputStream(someFile);

    /* Do some more processing */

}


try {
    processFile("file.txt");
} catch(FileNotFoundException fe) {
    System.err.println("You bozo, you've only gone and deleted
              file.txt");
} catch(IOException e) {
    System.err.println("Some IO Exception happened");
}
```

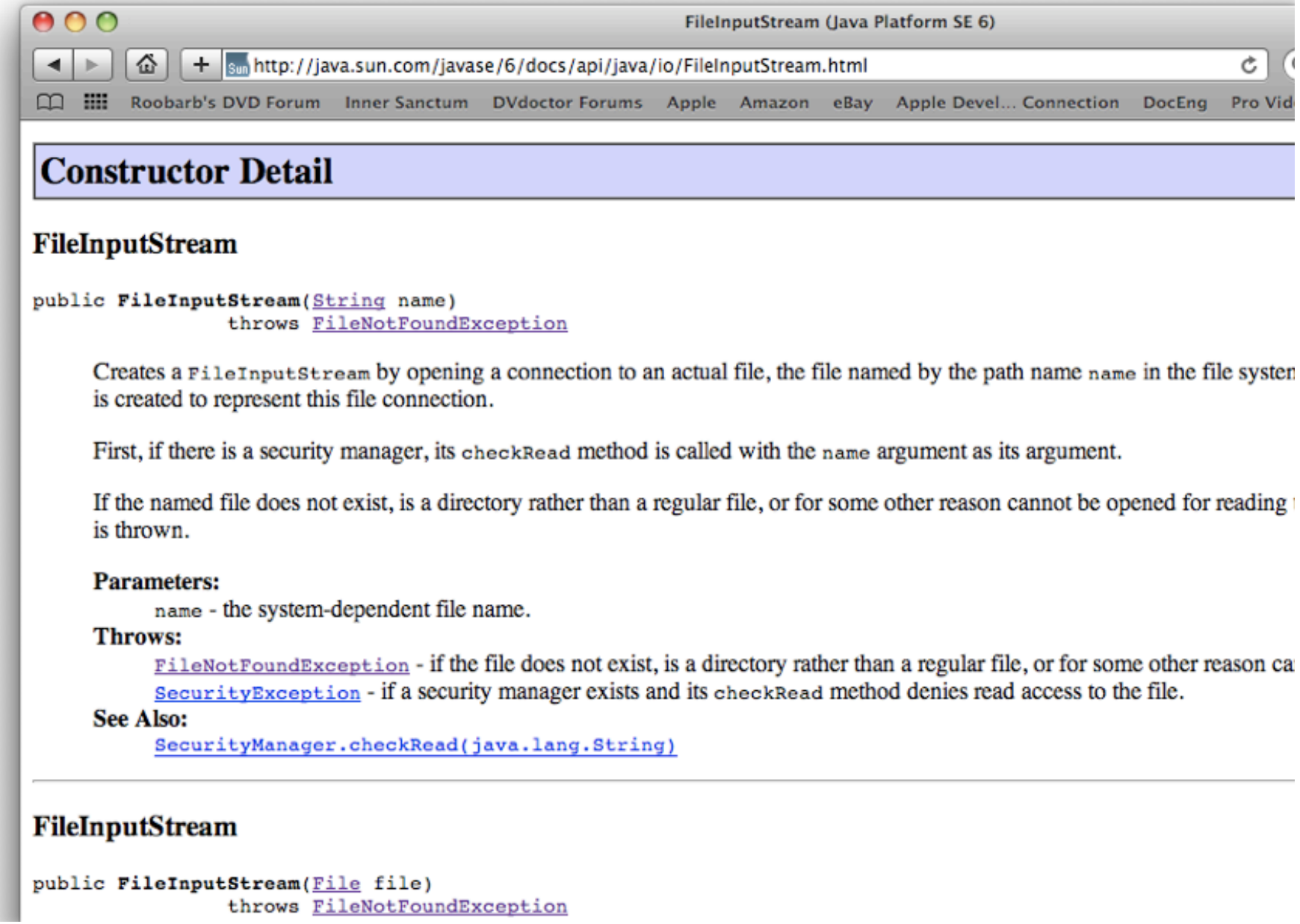
And Finally…

- Sometimes you may want to guarantee that some code is executed
- Regardless of the outcome of the `try` block
- Can do this using a `finally` block
- Code will execute regardless of how the `try` block is left
- Can have a `finally` without a `catch`

```
try {
    processFile(“file.txt”);
} catch(FileNotFoundException fe) {
    System.err.println(“You bozo, you’ve only gone and deleted
                file.txt”);
} catch(IOException e) {
    System.err.println(“Some IO Exception happened”);
} finally {
    System.err.println(“I’m always called”);
}
```

What Exceptions?

- Can see what exceptions a method may throw by looking at the API documentation
- It lists each exception a method can throw
- And the reasons when it may occur…
- So you can implement your handlers
- Don't just catch `Exception` and do nothing!



Constructor Detail

FileInputStream

```
public FileInputStream(String name)
                throws FileNotFoundException
```

Creates a `FileInputStream` by opening a connection to an actual file, the file named by the path name `name` in the file syste
is created to represent this file connection.

First, if there is a security manager, its `checkRead` method is called with the `name` argument as its argument.

If the named file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading
is thrown.

Parameters:

`name` - the system-dependent file name.

Throws:

`FileNotFoundException` - if the file does not exist, is a directory rather than a regular file, or for some other reason ca

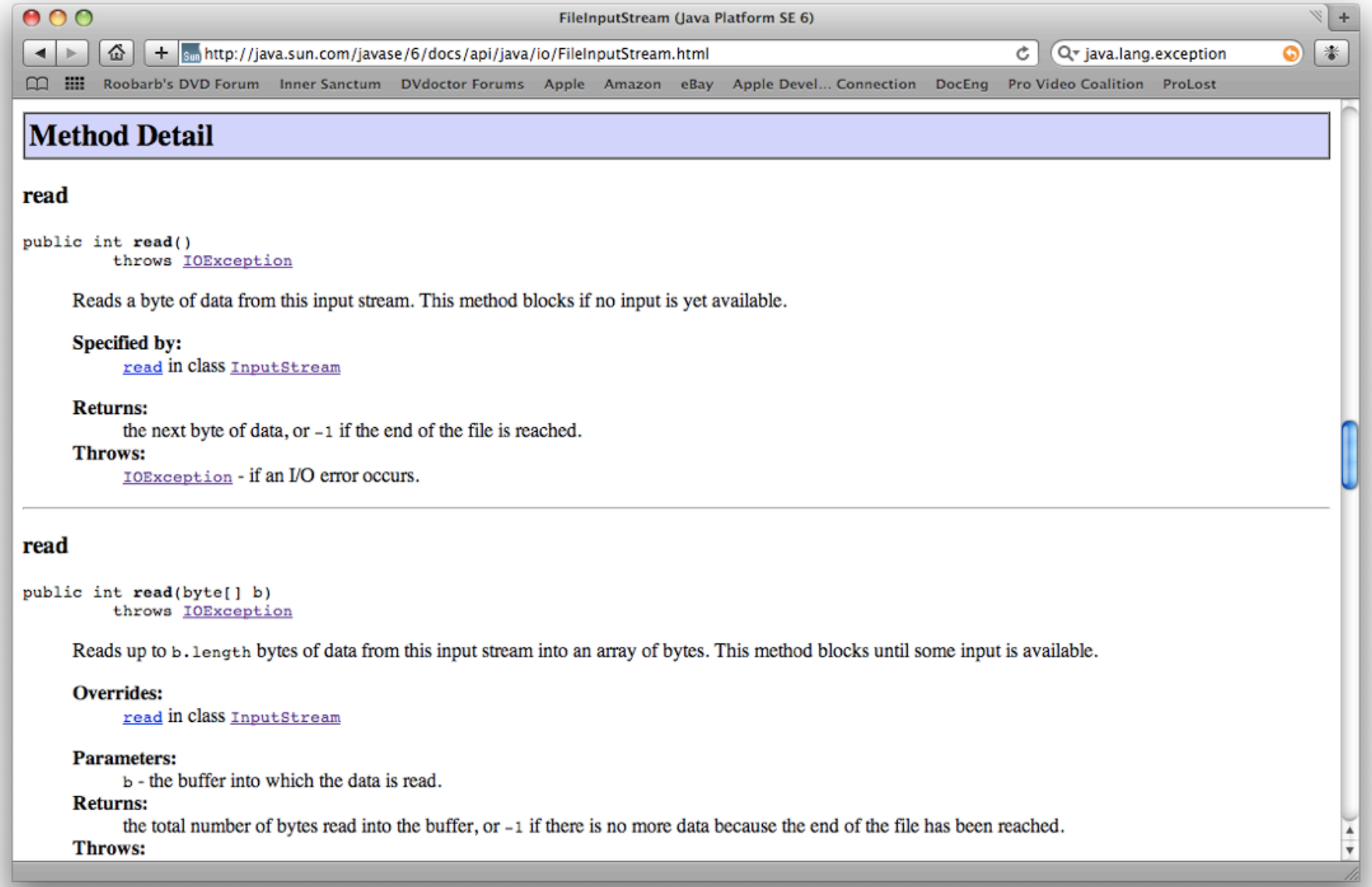
`SecurityException` - if a security manager exists and its `checkRead` method denies read access to the file.

See Also:

`SecurityManager.checkRead(java.lang.String)`

FileInputStream

```
public FileInputStream(File file)
                throws FileNotFoundException
```



Method Detail

read

```
public int read()
         throws IOException
```

Reads a byte of data from this input stream. This method blocks if no input is yet available.

Specified by:

read in class InputStream

Returns:

the next byte of data, or -1 if the end of the file is reached.

Throws:

IOException - if an I/O error occurs.

---

read

```
public int read(byte[] b)
         throws IOException
```

Reads up to b.length bytes of data from this input stream into an array of bytes. This method blocks until some input is available.

Overrides:

read in class InputStream

Parameters:

b - the buffer into which the data is read.

Returns:

the total number of bytes read into the buffer, or -1 if there is no more data because the end of the file has been reached.

Throws:

Personal Exceptions

- Very easy to create a new exception
- Just create a new class that sub-classes `Exception` or a sub-class of `Exception`
- Then when you find yourself in an exceptional circumstance just `throw` an instance of that class

```
public class MyException extends Exception {

	public MyException() {
		super();
	}

}
```

```
void someCode() throws MyException {
        int a;

        /* do some cool stuff */

        if (a == -1) {
                throw new MyException();
        }
}
```

Exception object

- Just a regular object, can be used to pass information up the chain
- But this should only be about the exception event
- Do not use it as a back channel to pass information

Why Exceptions?

- The downside to status codes and querying variables is it is very easy for the programmer to ignore it
- With Exceptions the compiler can enforce that the programmer handles it
- Or rather that they have written a try-catch block…

Why Exceptions?

- Still possible to just catch the exception
- And ignore it…
- But this is really not a good idea…
- The Exception tells us that something unexpected has happened
- Ignoring this leads to the possibility of security exploits