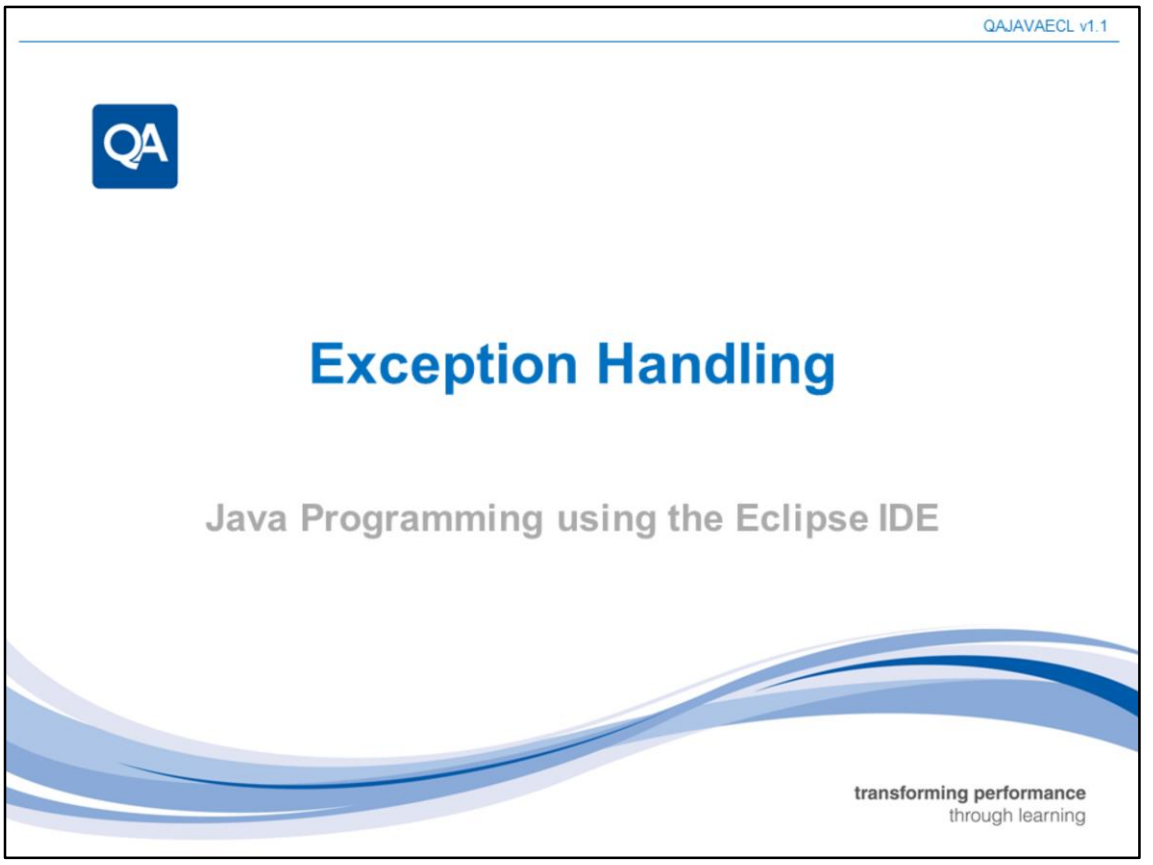

QAJAVAECL v1.1
QA
Exception Handling
Java Programming using the Eclipse IDE
transforming performance
through learning

Outline

- **Error handling in Java**
  - Debugger
- **Exceptions**
  - What is an exception?
  - Unchecked vs Checked
- **Handling Exceptions**
  - Try ... Catch
  - Finally
  - Throws
- **Writing our own exceptions**

Objectives

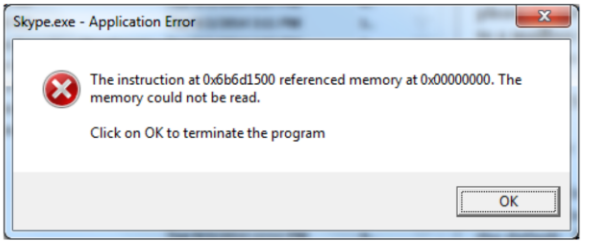
- **By the end of this session we should be able to:**
  - Understand and use exceptions in your code
  - Use the try, catch, finally blocks
  - Write our own exceptions
  - Use some basic read/write console operations

Outline

- **Error handling in Java**
  - Debugger
- **Exceptions**
  - What is an exception?
  - Unchecked vs Checked
- **Handling Exceptions**
  - Try ... Catch
  - Finally
  - Throws
- **Writing our own exceptions**

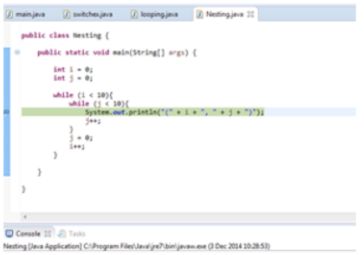
Program Errors

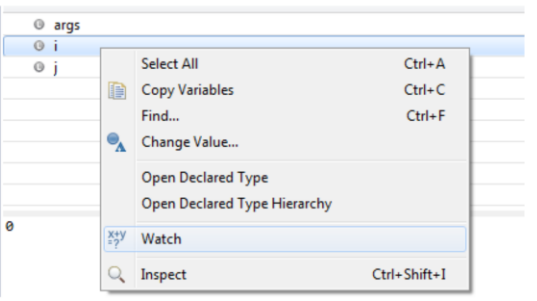
- **Many errors can occur when programming**
  - Syntax errors are picked up by IDEs and the compiler
  - Logical errors are harder to identify
- **Traditional languages return error codes to indicate a problem**
  - Difficult to see what was going on in the code
  - Needed to look up what the code meant!



The Eclipse Debugger

- **Debuggers allow us to step through code, seeing the state of the program after each instruction**
  - This can help identify the error
  - We can't ship a program that requires the user to do this!





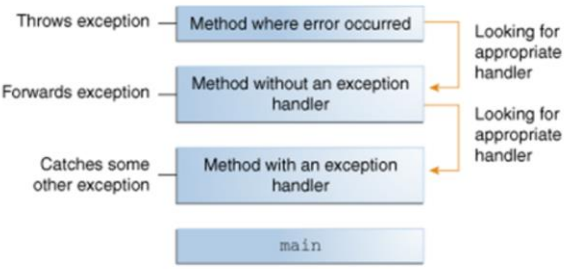
- **We need a way to recover from program failure gracefully and continue execution if possible!**

Outline

- **Error handling in Java**
  - Debugger
- **Exceptions**
  - What is an exception?
  - Unchecked vs Checked
- **Handling Exceptions**
  - Try ... Catch
  - Finally
  - Throws
- **Writing our own exceptions**

Exceptions

- **"An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions."**
- **Exceptions are thrown by methods when they reach an error state they cannot recover from themselves**
  - When an error occurs an Exception object is created by the run time and passed back up the call stack until there is a method that can handle it, or the program stops entirely



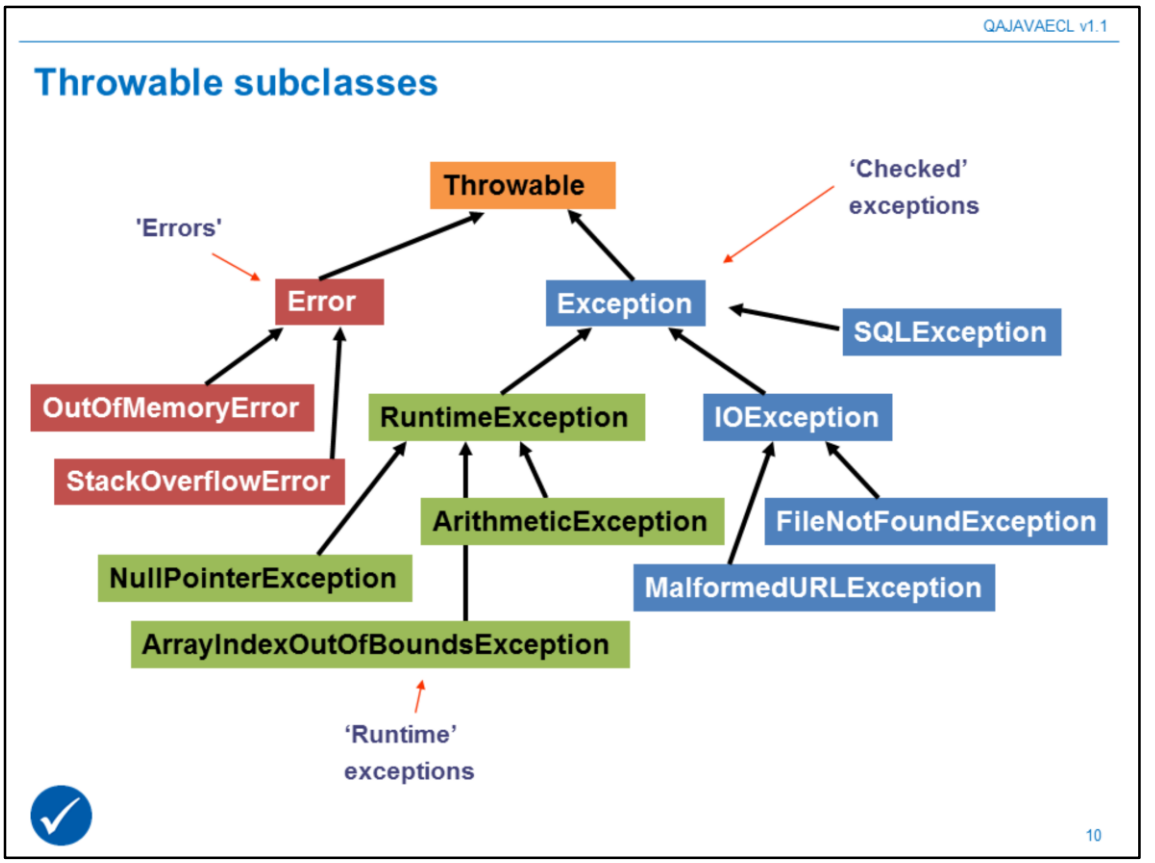
Definition and images from:
http://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

Types of built in exceptions

- **Parent object is of type Throwable**
  - There are two sub-classes, errors and exceptions
  - Errors are serious problems that should not be handled but should terminate the program
  - These are known as unchecked exceptions
- **There are many Exception classes for different situations**
  - http://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html
  - Exceptions contain a message that says what problem occurred during execution

`getMessage()`	Returns the message stored in the exception
`getLocalisedMessage()`	Localised version of the message
`getStackTrace()`	Print out the stack trace showing where the error occured
`toString()`	Description of the object

Unchecked exceptions are a bit controversial. The RuntimeError class and any Error or error sub-classes don't need to be checked for in the program. The idea behind this is that the problem isn't something that can be dealt with programmatically, as it refers to something outside Java's control (for example: a hard disk being unavailable, or a printer offline).



The slide shows the inheritance structure of some (!) common exception classes.

Outline

- **Error handling in Java**
  - Debugger
- **Exceptions**
  - What is an exception?
  - Unchecked vs Checked
- **Handling Exceptions**
  - Try ... Catch
  - Finally
  - Throws
- **Writing our own exceptions**

Handling Exceptions

- **It can be important for a program to continue running even after an exception has happened**
  - Roll back changes to a database
  - Close files
  - Allow the user to carry on working even if there has been a problem
- **Java uses a try/catch block to handle exceptions**
  - Allows the program to 'catch' exceptions that have been thrown and recover gracefully
  - You do not need to catch all exceptions
    - Checked exceptions need to be caught and dealt with, runtime exceptions do not (although it is a good idea to)

try... catch

- **The basic form of the try/catch block uses the following format:**
  - It is not best practice to catch all exceptions using the Exception class, you want to be as specific as possible when coding

```
try {
      //… some code …
} catch (SomeException e) {
      //… handle the exception …
} catch (AnotherException e1) {
      //… handle this exception …
}
```

try… catch example

```
public static void main(String[] args) {
	int[] array = new int[10];
	array[11] = 42;
	System.out.println("Continue to here");
}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 11
	at com.qa.Main.main(Main.java:12)
```

try… catch example

```
public static void main(String[] args) {

	int[] array = new int[10];

	try {
		array[11] = 42;
	} catch (ArrayIndexOutOfBoundsException e){
		System.out.println(e);
	}

	System.out.println("Continue to here");
}
```

```
java.lang.ArrayIndexOutOfBoundsException: 11
Continue to here
```

Enhanced try

JSE7 Enhancement

- **The try block has extra syntax to define resources**
  - Resource automatically closed/cleaned on exit of try

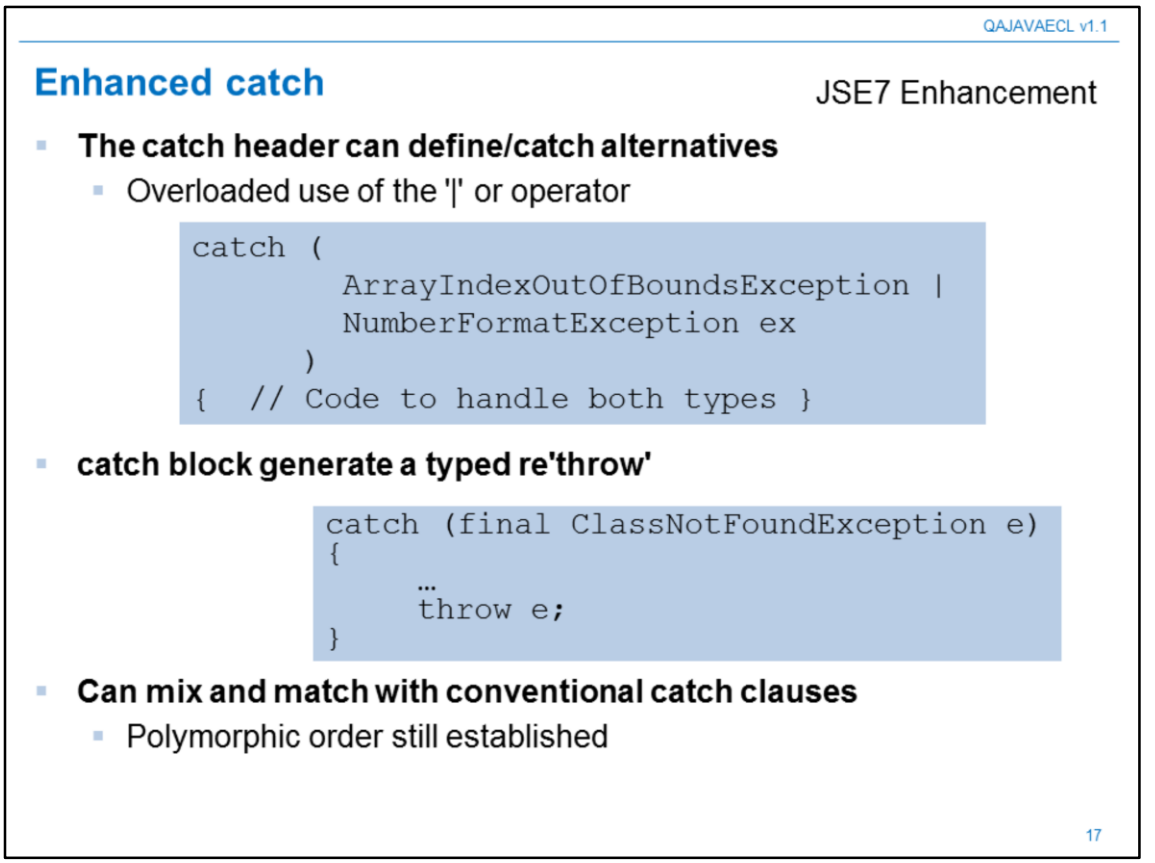
```
try(BufferedReader br= new BufferedReader(Source))
{
    // conventional I/O read code
}
```

- **Could define more than one resource**
  - Use ';' separated statements

```
try(
      BufferedReader br = new BufferedReader
                          (new InputStreamReader(System.in));
      BufferedWriter bw = new BufferedWriter
                         (new OutputStreamWriter(System.out));
   )
{
    // conventional I/O read/write code
}
```

16

JSE7 has introduced quite a number of enhancements related to exception handling. The enhanced try syntax supports the creating of objects local to the try block which require disposal. Usually the developer calls the close/dispose/release method on the object at the end of the try block. However, as long as the object is created as shown in the slide, i.e. within parentheses following the try keyword, the appropriate method is called by the system. Clearly, the object reference is only local to the try block itself (c.f. the for loop scope)



The catch block has also received enhancements in JSE7. Two are shown here.

The upper code fragment illustrates the use of the '|' operator to provide an Or relationship within a single catch block. This will provide the developer with a means of grouping the exception types logically over and above the polymorphic grouping that has always been supported.

The lower block fragment shows the use of the final keyword being 'overloaded' to inform the system that the throw will be the appropriate IOException subclass as opposed to the top level IOException itself. This is a useful mechanism when there is a catchall required that will simply throw the exception up a level, i.e. the lower block would have been written:

```
// Pre JSE7 version
    catch (MalformedURLException ex)
    {
      throw e;
    }
    catch (UnknownHostException ex)
    {
      throw e;
    }
    catch (IOException ex)
    {
      throw e;
    }
```

QAJAVAECL v1.1

Finally

- **There are some operations that we want to ensure always run, even if an exception is thrown, for this we use the finally block**
  - Most common use is to close resources that we've opened
  - When execution leaves try block…
    - After completing normally or exiting due to return or break
    - Or after exception thrown

```
BufferedReader br = null;
String next;
try
{
  br = new BufferedReader(new InputStreamReader(System.in));
  … // use reader
}
finally
{
  if (br != null)
    br.close();
}
```

18

The finally block will always be executed, even if the try or catch blocks are exited with a break, continue or return. This can be very useful for cleaning up some resource, such as a file, that a method has acquired.

Note though, that the close() method on the slide also generated checked exceptions - so more try / catch blocks will be required. In the above example, we should also check to see if in is null before trying to call close(), e.g.

```
finally
{
    if (in != null)
    {
        try
        {
            in.close();
        }
        catch (IOException ie)
        {
        System.out.println("error while closing");
        }
    }
}
```

Eclipse Help

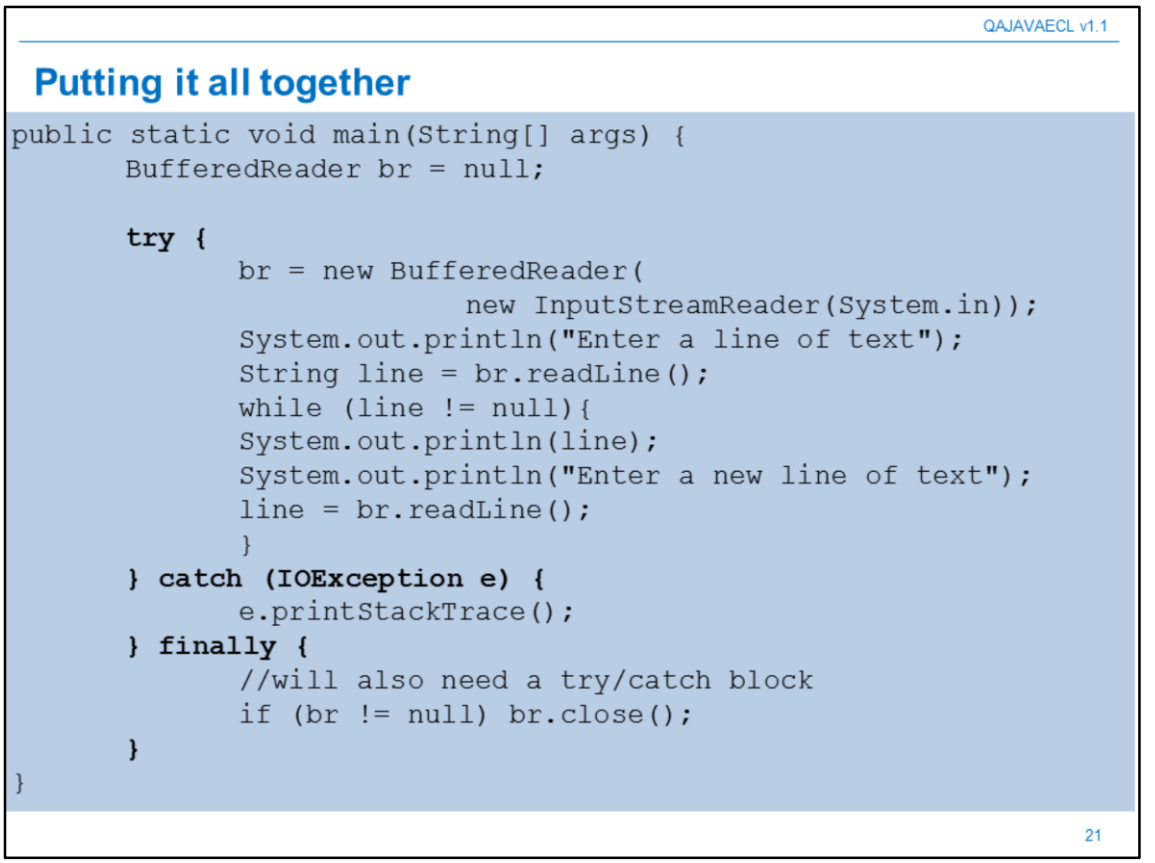
- **Eclipse will show errors if exceptions are not handled**
  - It can automatically surround the code in a try/catch block
  - Includes the specific exceptions that are being thrown

```
1  package com.qa;
2
3  import java.io.BufferedReader;
4  import java.io.InputStreamReader;
5
6  public class ReadInput {
7
8      public static void main(String[] args) {
9          // TODO Auto-generated method stub
10
11         BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
12
13         System.out.println("Enter a line of text");
14
15         String line = br.readLine();
16
17     }
18
19 }
```

Unhandled exception type IOException
2 quick fixes available:
Add throws declaration
Surround with try/catch

Eclipse Help

- **If there is already a try/catch block then eclipse offers the option to add additional exceptions to it automatically**



In the finally block we will need a try/catch block to compile the code, but there isn't enough space to show it all! The finally block should read

```
try {
    if (br != null) br.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

Throwing exceptions

- **We can see if an exception is thrown by a method by looking at the JavaDoc**
  - Uses the @throws annotation

readLine

```
public String readLine()
                throws IOException
```

Reads a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.

Returns:

A String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached

Throws:

IOException - If an I/O error occurs

See Also:

Files.readAllLines(java.nio.file.Path, java.nio.charset.Charset)

Throwing exceptions

- **Sometimes we don't want a method to deal with an exception**
  - We want to throw the exception back to the calling method to handle
  - Allows the method to only look at the logic rather than logic and exception

```
public void readInputLines()
                    throws IOException {
        BufferedReader br = null;
        try {
                br = new BufferedReader(
                        new InputStreamReader(System.in));
        System.out.println("Enter a line of text");
        String line = br.readLine();
        while (line != null) {
                System.out.println(line);
               System.out.println("Enter a new line of text");
                line = br.readLine();
        }
        } finally {
                if (br != null) br.close();
        }
}
```

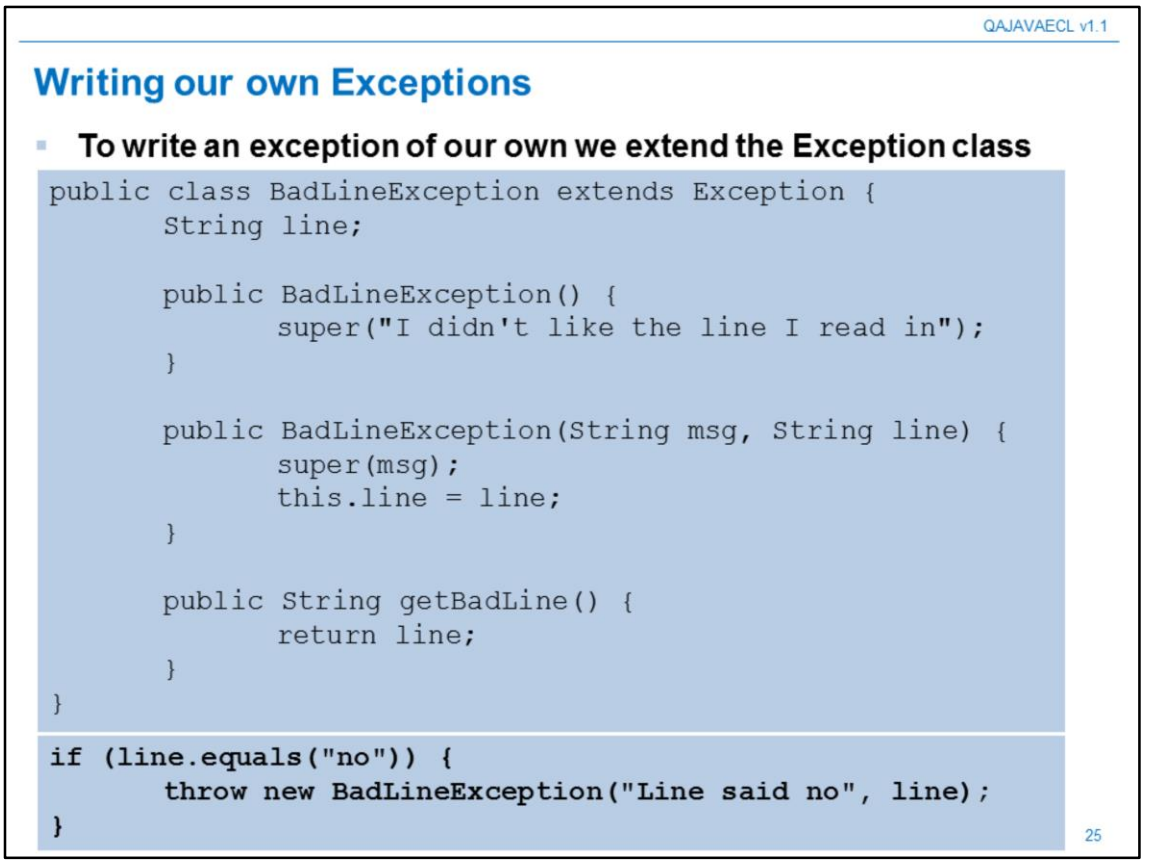
When calling the method we need to catch the exception:

```
try {
    new ReadInput().readInputLines();
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

The finally block is called before the exception is thrown!

Outline

- **Error handling in Java**
  - Debugger
- **Exceptions**
  - What is an exception?
  - Unchecked vs Checked
- **Handling Exceptions**
  - Try ... Catch
  - Finally
  - Throws
- **Writing our own exceptions**



As the BadLineException is a subclass of Exception we get access to all the Exception methods still, but we can also store our own information needed to try and recover from the exception

Summary

- **Error handling in Java**
  - Debugger
- **Exceptions**
  - What is an exception?
  - Unchecked vs Checked
- **Handling Exceptions**
  - Try ... Catch
  - Finally
  - Throws
- **Writing our own exceptions**

Exercise

- **Using and writing exceptions in Java**
  - Use console I/O operations to read and write to console
  - Use the try/catch block to recover from any errors in the code
  - Use the finally block to close open resources