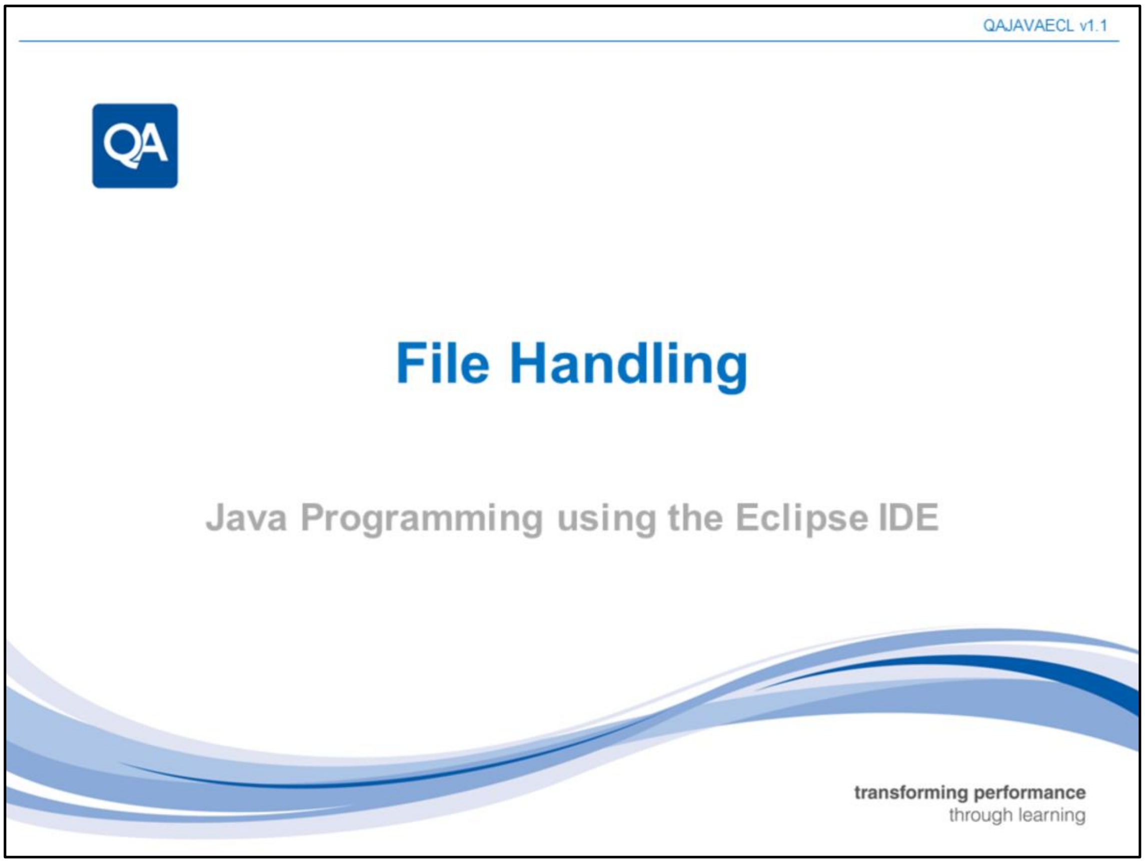
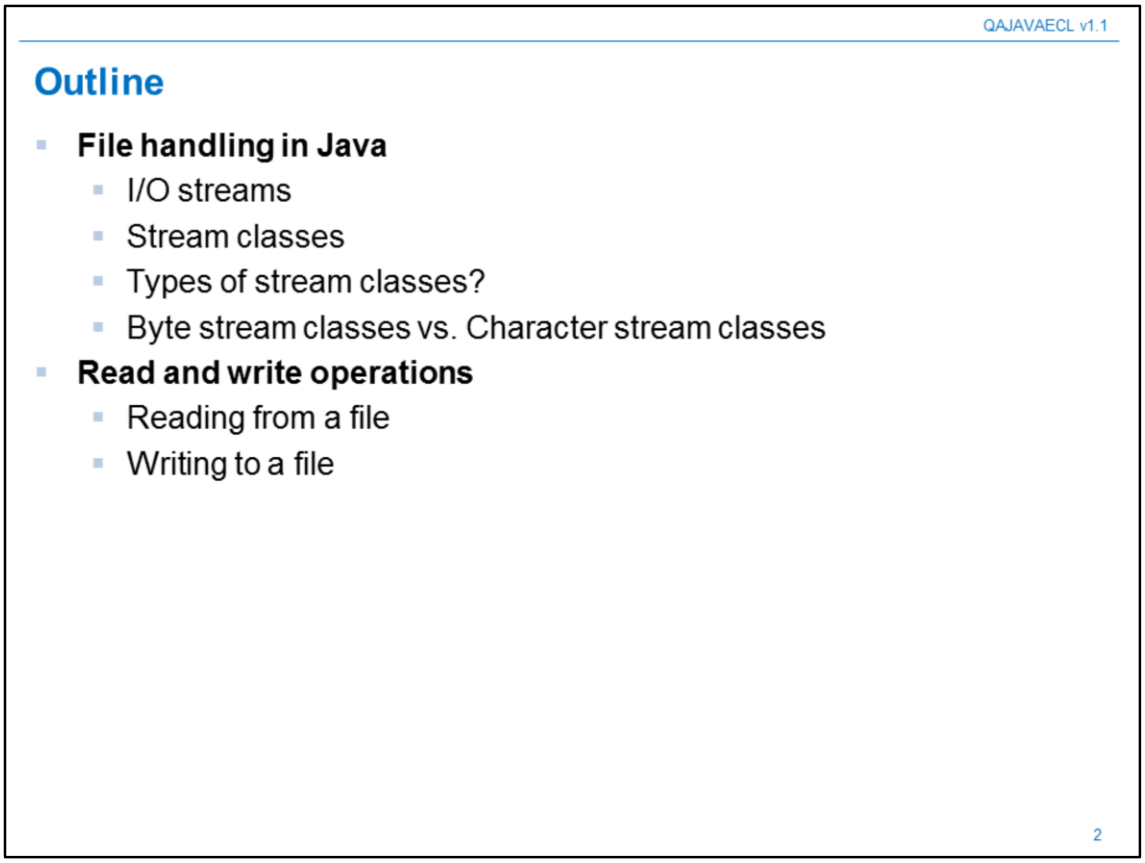


QAJAVAECL v1.1

File Handling

Java Programming using the Eclipse IDE

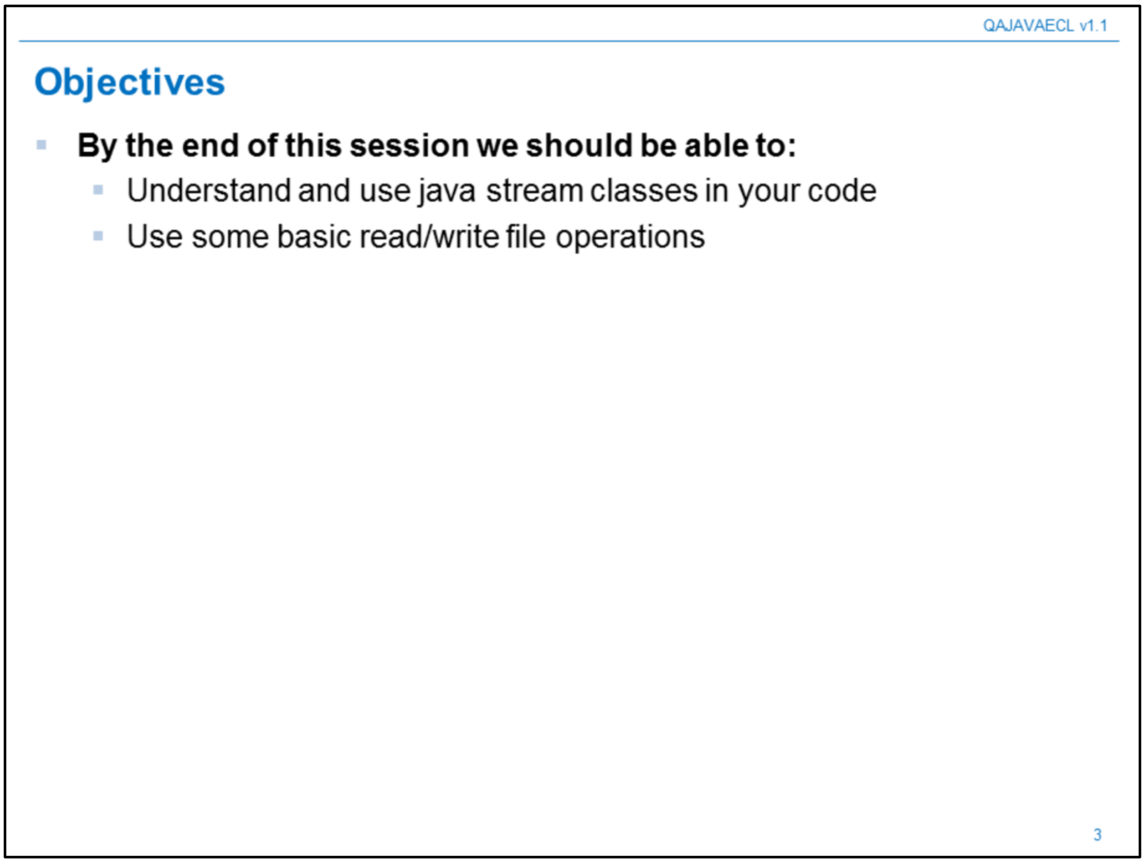
transforming performance
through learning



Outline

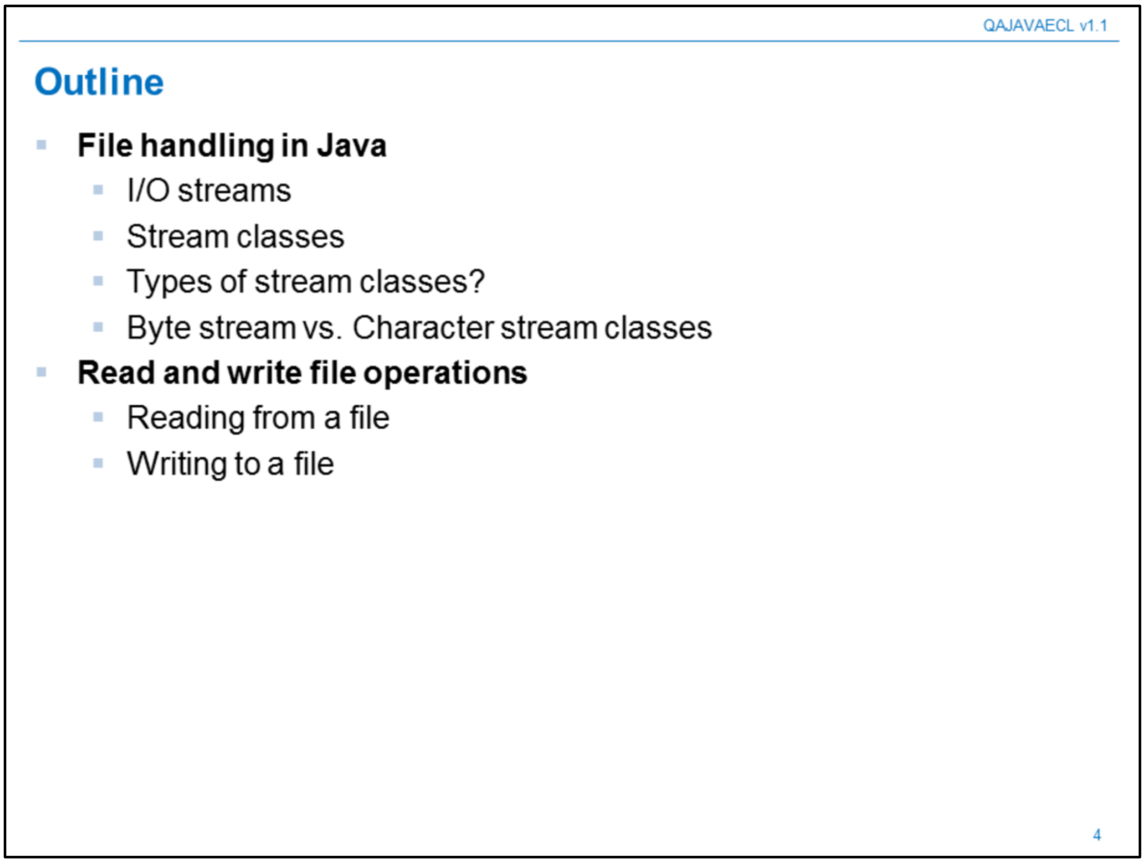
- **File handling in Java**
  - I/O streams
  - Stream classes
  - Types of stream classes?
  - Byte stream classes vs. Character stream classes
- **Read and write operations**
  - Reading from a file
  - Writing to a file

2



Objectives

- **By the end of this session we should be able to:**
  - Understand and use java stream classes in your code
  - Use some basic read/write file operations

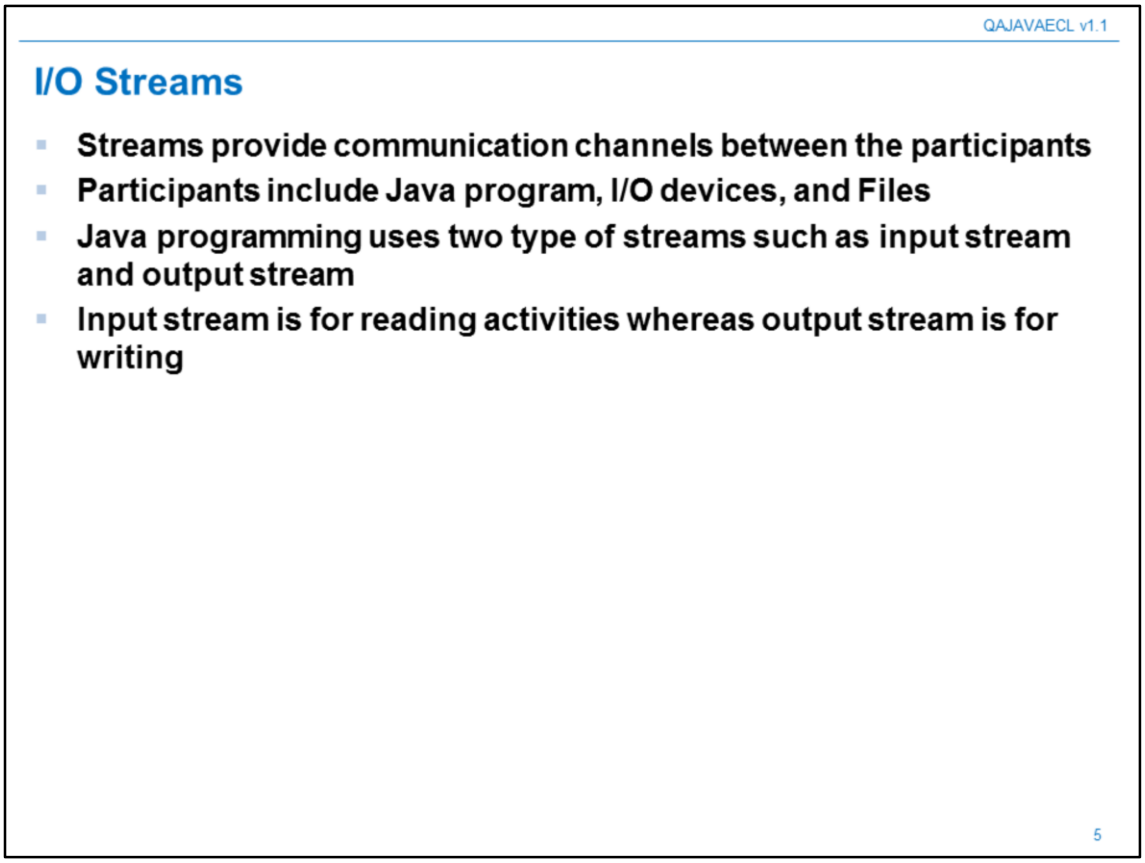


QAJAVAECL v1.1

Outline

- **File handling in Java**
  - I/O streams
  - Stream classes
  - Types of stream classes?
  - Byte stream vs. Character stream classes
- **Read and write file operations**
  - Reading from a file
  - Writing to a file

4



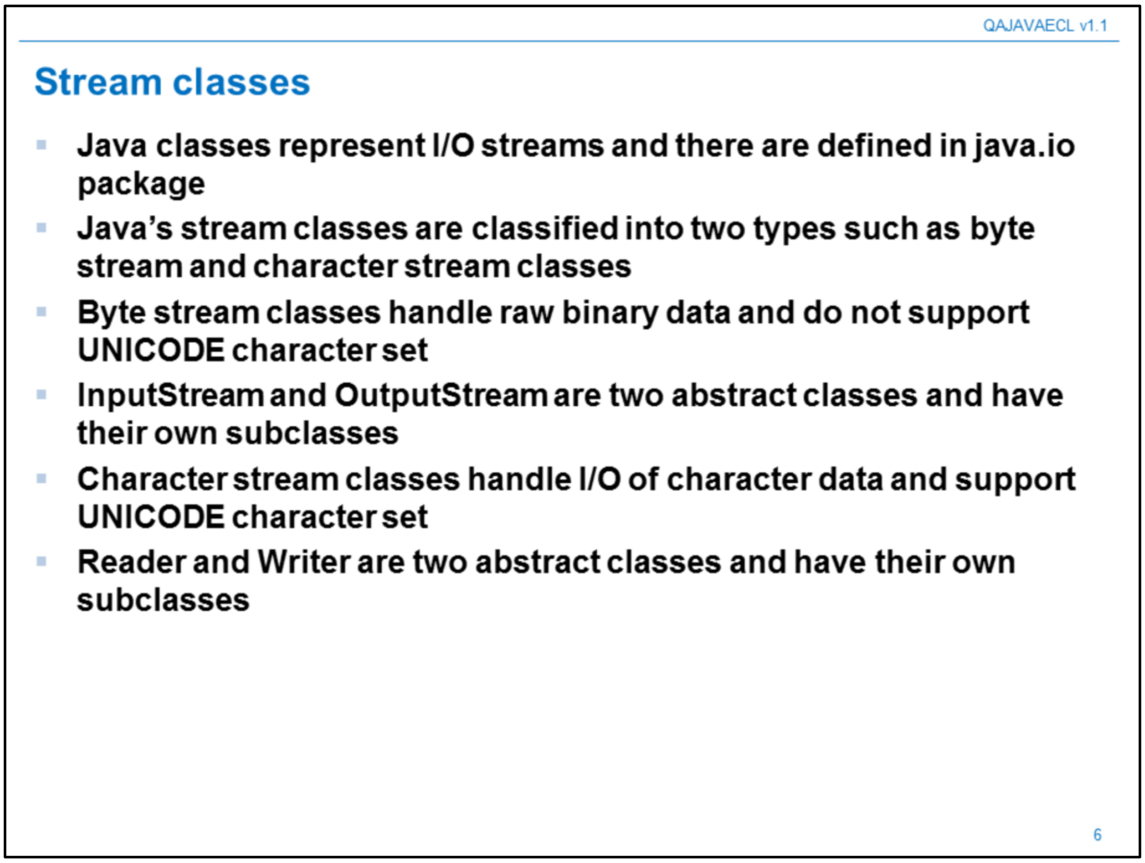
QAJAVAECL v1.1

I/O Streams

- Streams provide communication channels between the participants
- Participants include Java program, I/O devices, and Files
- Java programming uses two type of streams such as input stream and output stream
- Input stream is for reading activities whereas output stream is for writing

5

Definition from: https://docs.oracle.com/javase/tutorial/essential/io/index.html



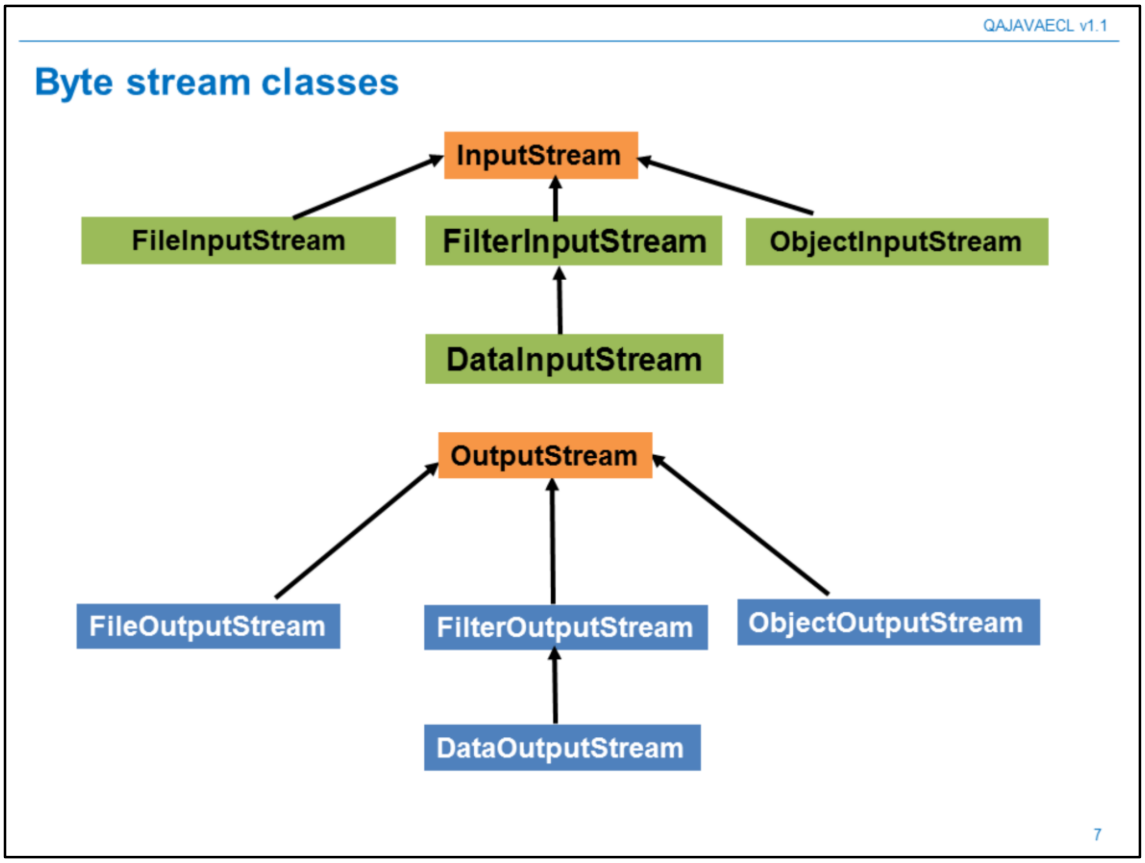
QAJAVAECL v1.1

Stream classes

- Java classes represent I/O streams and there are defined in java.io package
- Java's stream classes are classified into two types such as byte stream and character stream classes
- Byte stream classes handle raw binary data and do not support UNICODE character set
- InputStream and OutputStream are two abstract classes and have their own subclasses
- Character stream classes handle I/O of character data and support UNICODE character set
- Reader and Writer are two abstract classes and have their own subclasses

6

Definition from: https://docs.oracle.com/javase/tutorial/essential/io/index.html



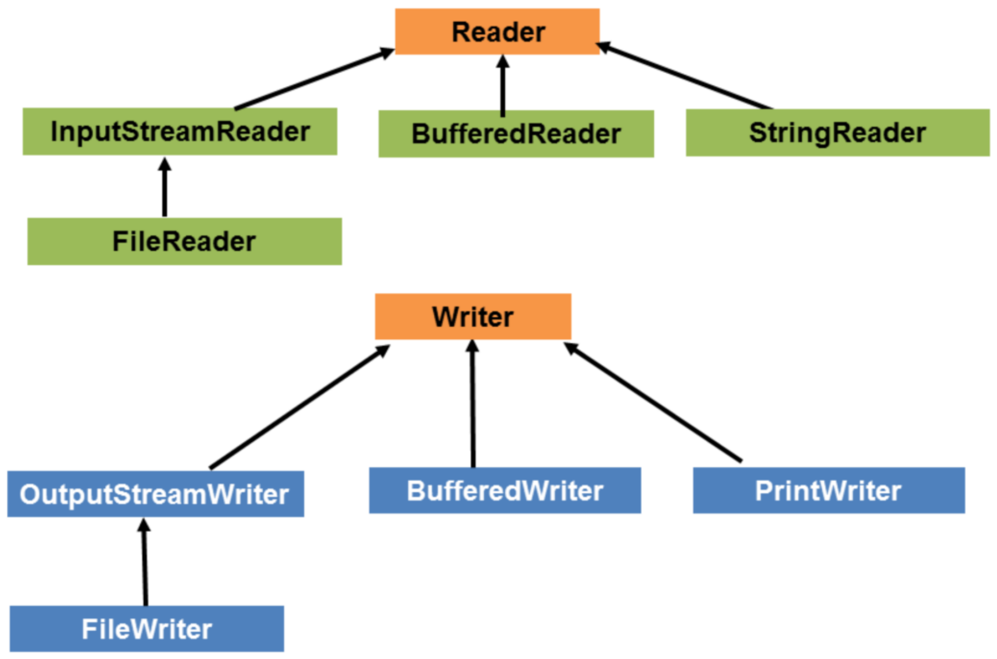
Byte stream classes

$$\begin{tikzcd}
 & \text{InputStream} & \\
\text{FileInputStream} \arrow[ur] & \text{FilterInputStream} \arrow[u] & \text{ObjectInputStream} \arrow[ul] \\
 & \text{DataInputStream} \arrow[u] &
\end{tikzcd}$$

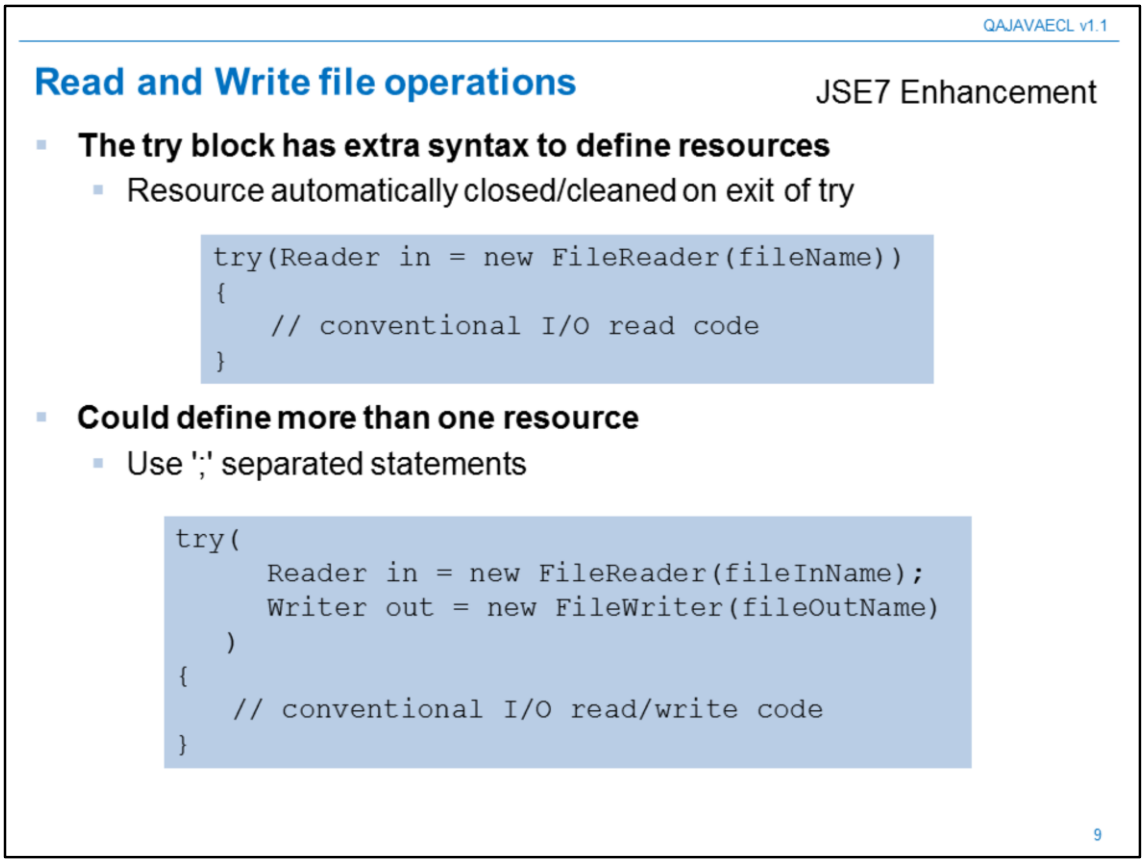
$$\begin{tikzcd}
 & \text{OutputStream} & \\
\text{FileOutputStream} \arrow[ur] & \text{FilterOutputStream} \arrow[u] & \text{ObjectOutputStream} \arrow[ul] \\
 & \text{DataOutputStream} \arrow[u] &
\end{tikzcd}$$

The slide shows the inheritance structure of some (!) common InputStream and OutputStream classes.

Character stream classes



The slide shows the inheritance structure of some (!) common Reader and Writer classes.



QAJAVAECL v1.1

Read and Write file operations

JSE7 Enhancement

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

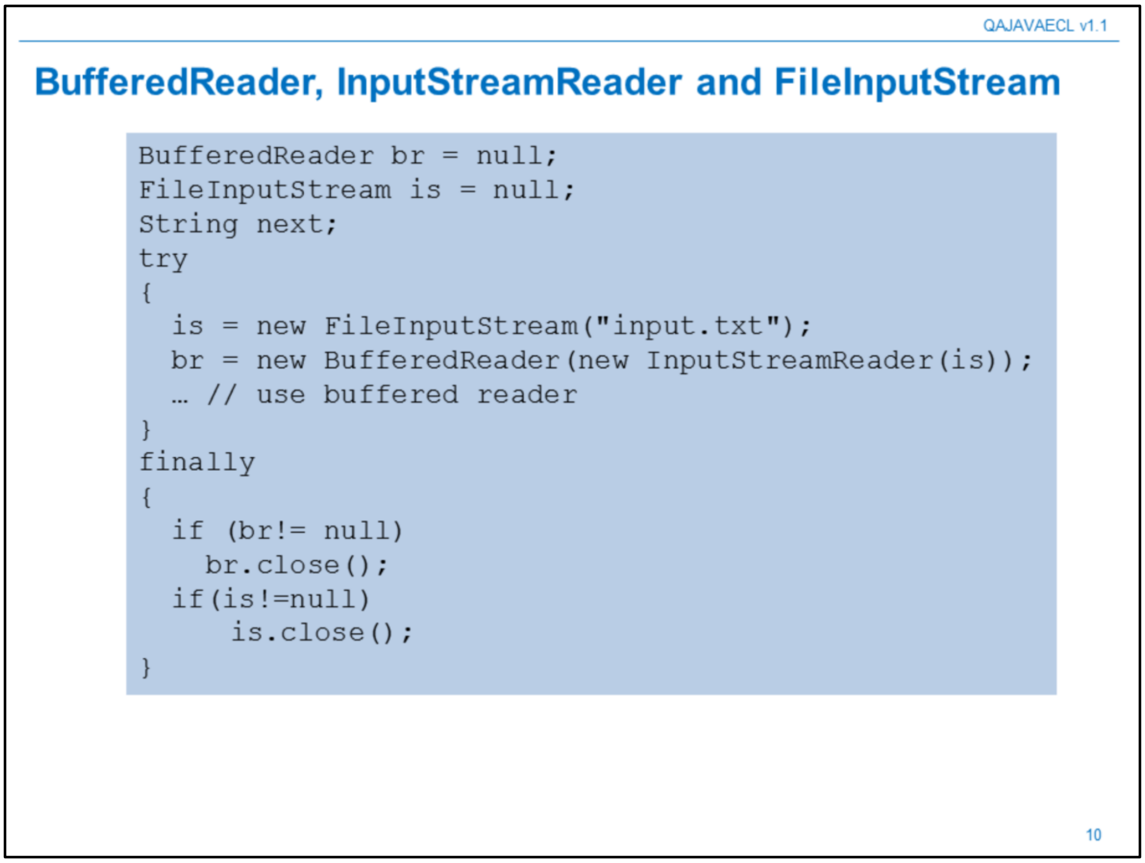
```
try(Reader in = new FileReader(fileName))
{
    // conventional I/O read code
}
```

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

```
try(
       Reader in = new FileReader(fileInName);
       Writer out = new FileWriter(fileOutName)
   )
{
    // conventional I/O read/write code
}
```

9

This slide shows the try-with-resources concept (recall exception handling). Reader is one the abstract classes and refers to an object of its subclass FileReader. Here FileReader object accesses the physical location of a file and assigns to Reader class. In the same way FileWriter gets the connection to a file destination and assigns it to Writer class. Now Reader and Writer class objects can use the appropriate methods and perform read and write file operations. The read and write file operations may raise FileNotFoundException and IOException, we should recall the concepts of Exception Handling.



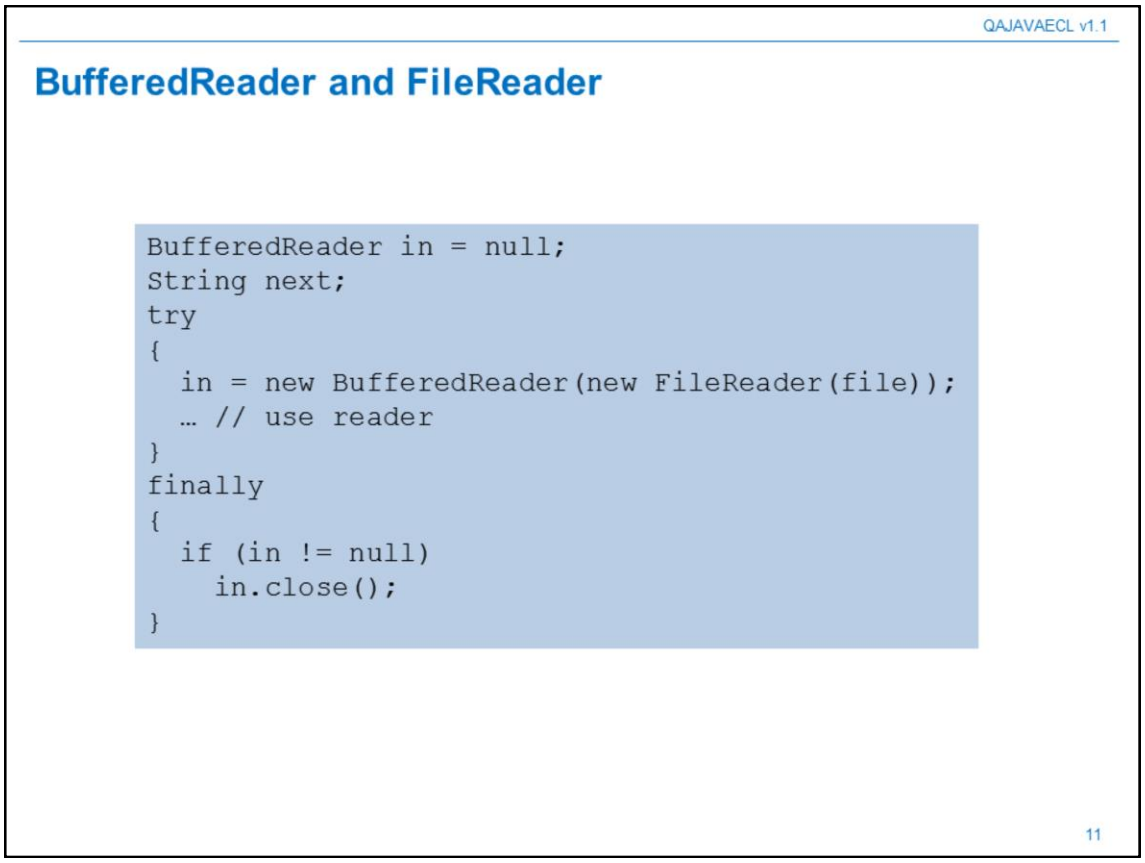
QAJAVAECL v1.1

BufferedReader, InputStreamReader and FileInputStream

```
BufferedReader br = null;
FileInputStream is = null;
String next;
try
{
  is = new FileInputStream("input.txt");
  br = new BufferedReader(new InputStreamReader(is));
  … // use buffered reader
}
finally
{
  if (br!= null)
    br.close();
  if(is!=null)
      is.close();
}
```

10

This slide shows the combination of byte stream and character stream classes. FileInputStream is one of the byte stream classes which handles raw binary data. BufferedReader and InputStreamReader classes belong to character stream category, InputStreamReader builds a bridge between FileInputStream and BufferedReader.



QAJAVAECL v1.1

BufferedReader and FileReader

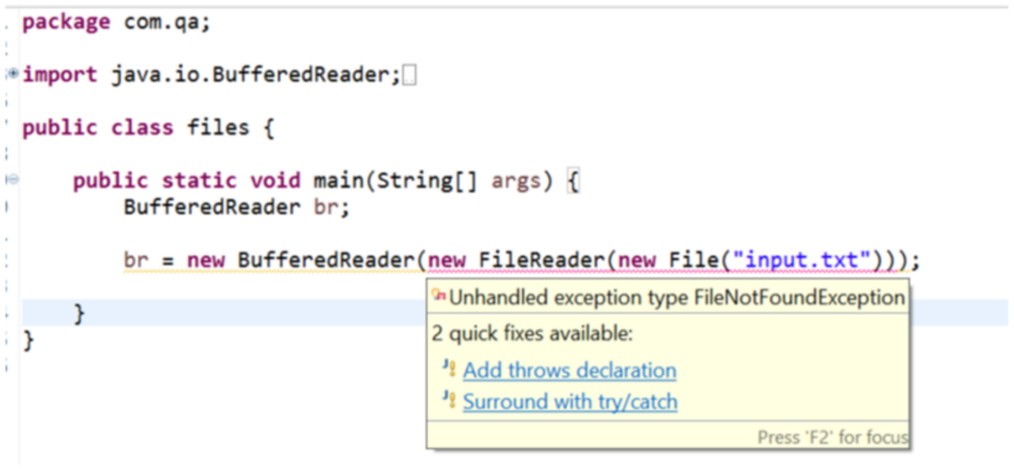
```
BufferedReader in = null;
String next;
try
{
  in = new BufferedReader(new FileReader(file));
  … // use reader
}
finally
{
  if (in != null)
    in.close();
}
```

11

BufferedReader and FileReader is known as a good combination because both classes belong to character stream and do not require any bridge class as explained in the previous slide.

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

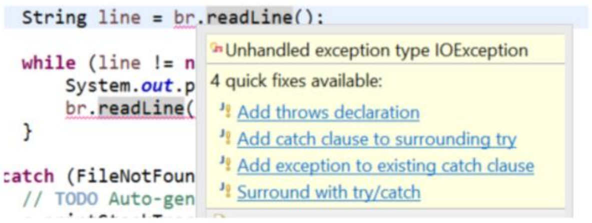


```
package com.qa;

import java.io.BufferedReader;

public class files {

    public static void main(String[] args) {
        BufferedReader br;

        br = new BufferedReader(new FileReader(new File("input.txt")));

    }
}
```

```
Unhandled exception type FileNotFoundException
2 quick fixes available:
  Add throws declaration
  Surround with try/catch
                                  Press 'F2' for focus
```

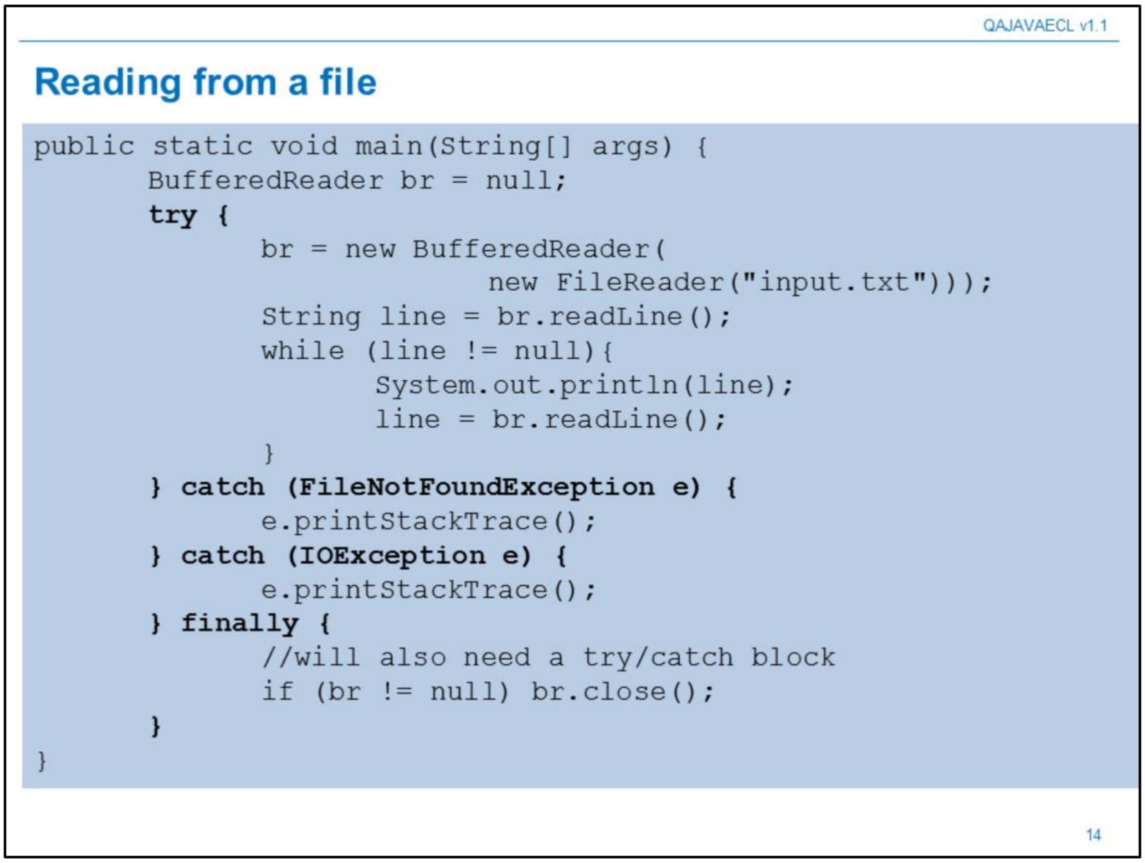
Eclipse Help

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



```
String line = br.readLine();

while (line != n
    System.out.p
    br.readLine(
}

catch (FileNotFoun
// TODO Auto-gen
```

Unhandled exception type IOException
4 quick fixes available:
Add throws declaration
Add catch clause to surrounding try
Add exception to existing catch clause
Surround with try/catch



QAJAVAECL v1.1

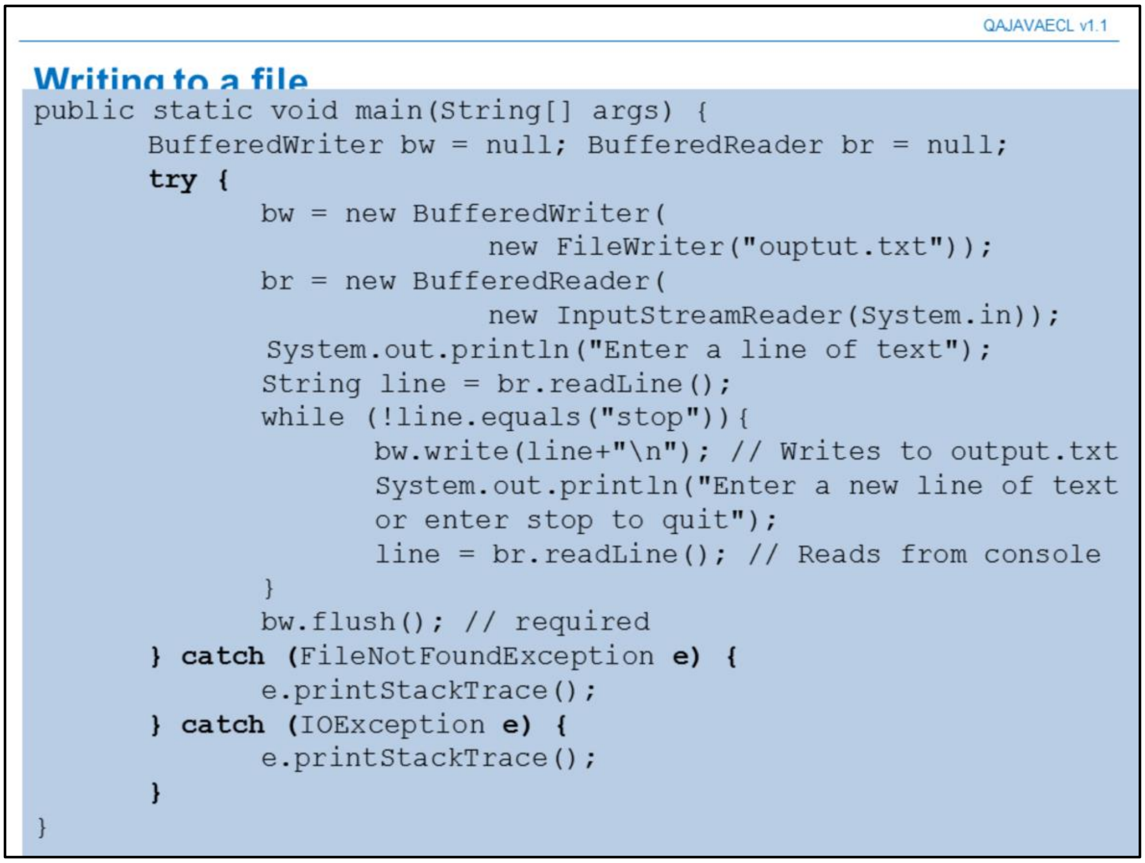
Reading from a file

```
public static void main(String[] args) {
        BufferedReader br = null;
        try {
                br = new BufferedReader(
                                new FileReader("input.txt"));
                String line = br.readLine();
                while (line != null){
                        System.out.println(line);
                        line = br.readLine();
                }
        } catch (FileNotFoundException e) {
                e.printStackTrace();
        } catch (IOException e) {
                e.printStackTrace();
        } finally {
                //will also need a try/catch block
                if (br != null) br.close();
        }
}
```

14

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();
}
```

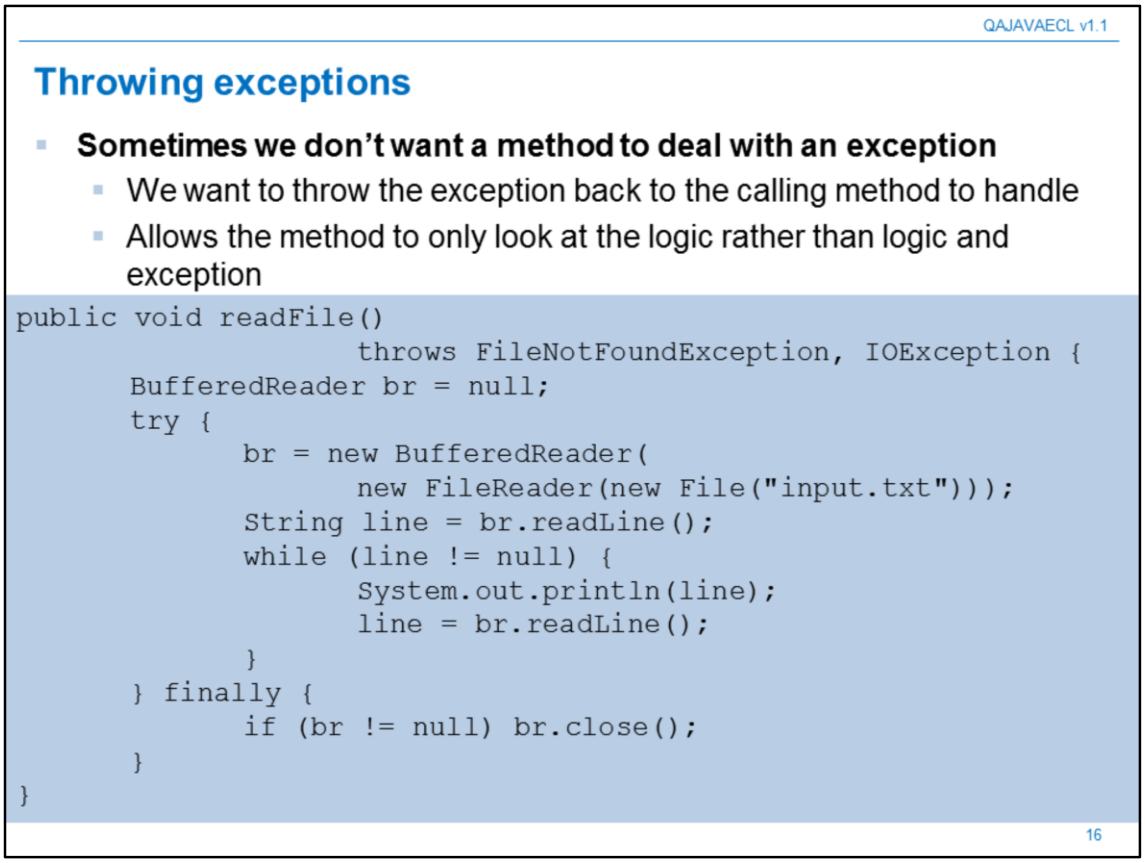

QAJAVAECL v1.1

Writing to a file

```
public static void main(String[] args) {
        BufferedWriter bw = null; BufferedReader br = null;
        try {
                bw = new BufferedWriter(
                                new FileWriter("ouptut.txt"));
                br = new BufferedReader(
                                new InputStreamReader(System.in));
                System.out.println("Enter a line of text");
                String line = br.readLine();
                while (!line.equals("stop")){
                        bw.write(line+"\n"); // Writes to output.txt
                        System.out.println("Enter a new line of text
                        or enter stop to quit");
                        line = br.readLine(); // Reads from console
                }
                bw.flush(); // required
        } catch (FileNotFoundException e) {
                e.printStackTrace();
        } catch (IOException e) {
                e.printStackTrace();
        }
}
```

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

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


QAJAVAECL v1.1

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

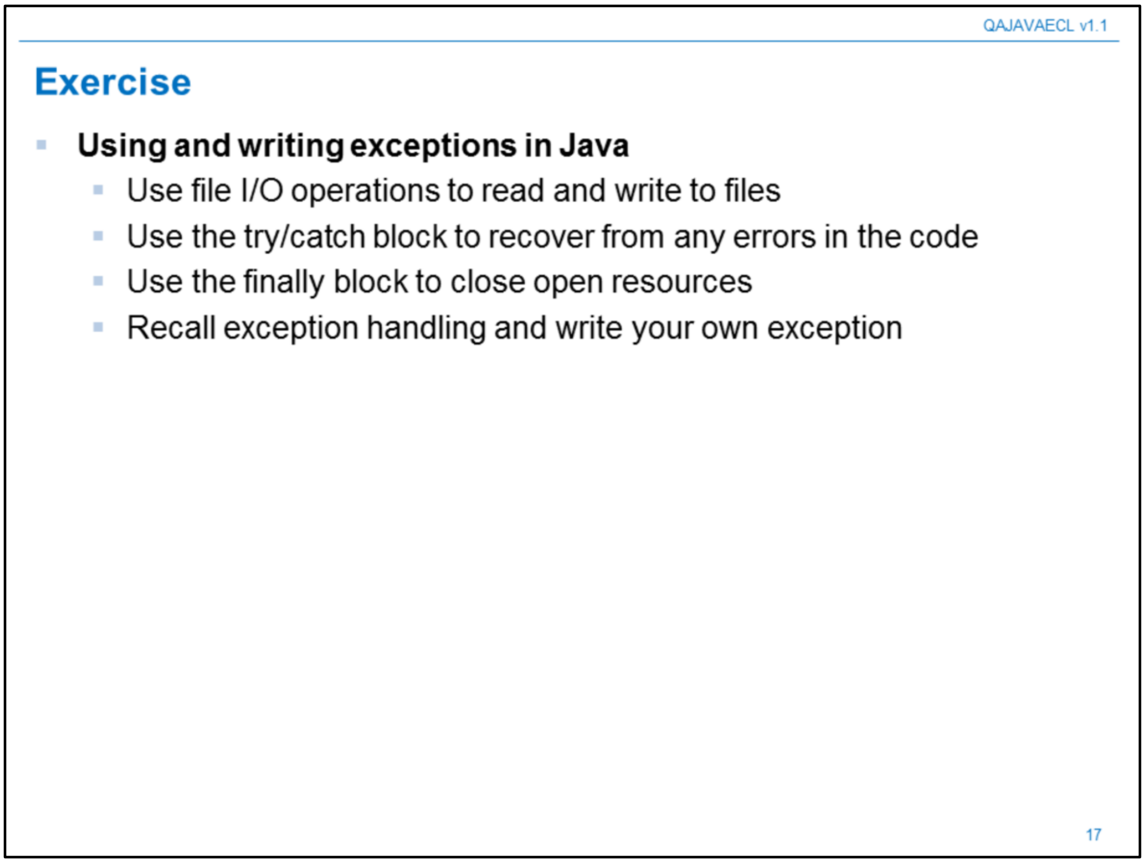
```
public void readFile()
                    throws FileNotFoundException, IOException {
        BufferedReader br = null;
        try {
                br = new BufferedReader(
                        new FileReader(new File("input.txt")));
                String line = br.readLine();
                while (line != null) {
                        System.out.println(line);
                        line = br.readLine();
                }
        } finally {
                if (br != null) br.close();
        }
}
```

16

When calling the method we need to catch the exception:

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

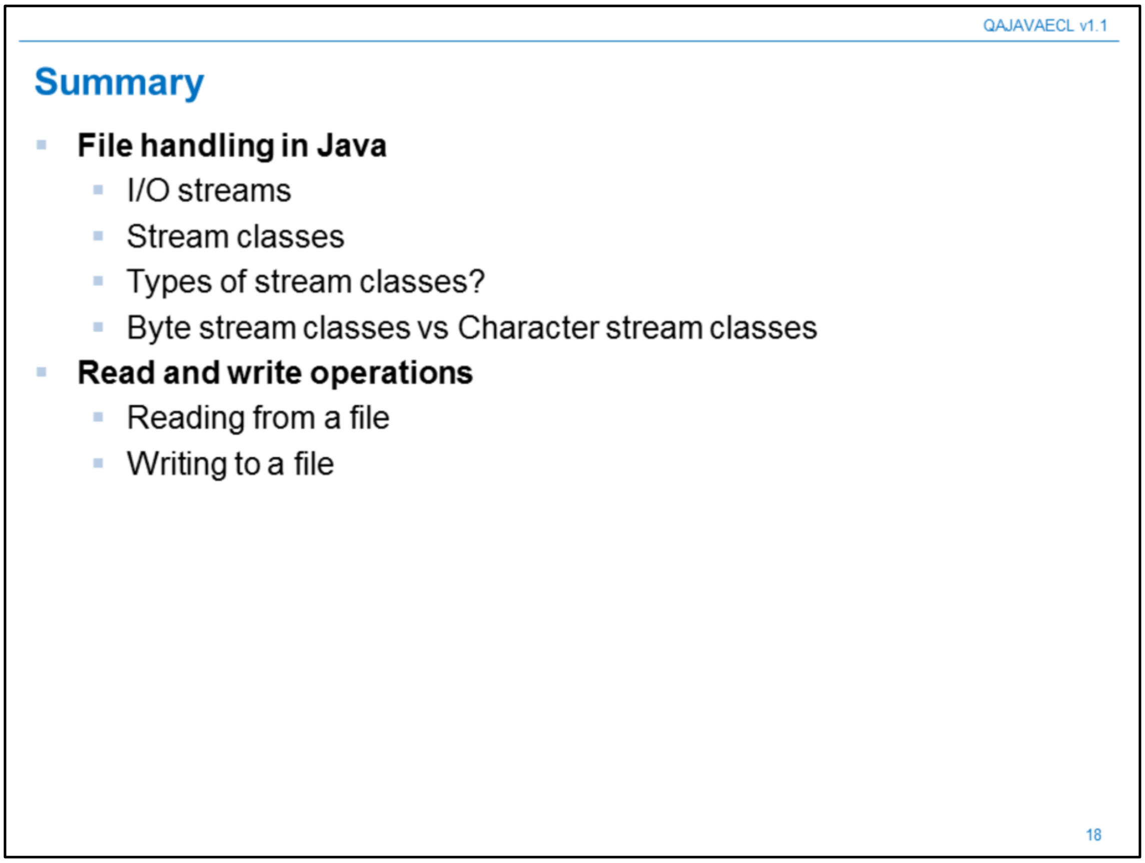
The finally block is called before the exception is thrown!


QAJAVAECL v1.1

Exercise

- **Using and writing exceptions in Java**
  - Use file I/O operations to read and write to files
  - Use the try/catch block to recover from any errors in the code
  - Use the finally block to close open resources
  - Recall exception handling and write your own exception

17



QAJAVAECL v1.1

Summary

- **File handling in Java**
  - I/O streams
  - Stream classes
  - Types of stream classes?
  - Byte stream classes vs Character stream classes
- **Read and write operations**
  - Reading from a file
  - Writing to a file

18