Computer Science 2

Lecture 6

Exception Handling and Streams

Extra material in Canvas

Overview

1. Exception handling
   - Why do we need exception handling?
   - Exceptions and exception handling
   - Checked and unchecked exceptions
   - Examples
2. Streams: Byte streams and character streams
   - FileReader and FileWriter
   - File and JFileChooser
   - Command line arguments
   - Random access files
   - Object streams

Learning goals

- Understand and be able to utilize *exception handling* techniques
- Identify differences between *checked* and *unchecked exceptions* and their implications
- Be able to implement different input/output methods based on *streams* (using files, command line, RAM, objects, etc.)
- Be able to use *command line arguments* in our classes

EXCEPTION HANDLING

Why do we need exception handling?

```
import java.util.Scanner;

public class Input {
    public static void main(String[] args) {
      System.out.print("Introduce a number: ");
      Scanner input = new Scanner(System.in);
      int number = input.nextInt();
      System.out.print(number);
    }
}
```

```
E:\                                                    >java Input
Introduce a number:  2
2
```

Why do we need exception handling?

```
import java.util.Scanner;

public class Input {
    public static void main(String[] args) {
      System.out.print("Introduce a number: ");
      Scanner input = new Scanner(System.in);
      int number = input.nextInt();
      System.out.print(number);
    }
}
```

```
E:\                                                    >java Input
Introduce a number:  2
2
E:\                                                    >java Input
Introduce a number: a
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:864)
        at java.util.Scanner.next(Scanner.java:1485)
        at java.util.Scanner.nextInt(Scanner.java:2117)
        at java.util.Scanner.nextInt(Scanner.java:2076)
        at Input.main(Input.java:7)
```

The task

Code

```
readFile()
  {
     openFile();
          determineFileSize();
          allocateMemory();
          readFileIntoMemory();
          closeFile();
    }
```

Possible Errors

- the file can't be opened
- the length of the file can't be determined
- enough memory can't be allocated
- the read fails
- the file can't be closed

Traditional approach to error handling

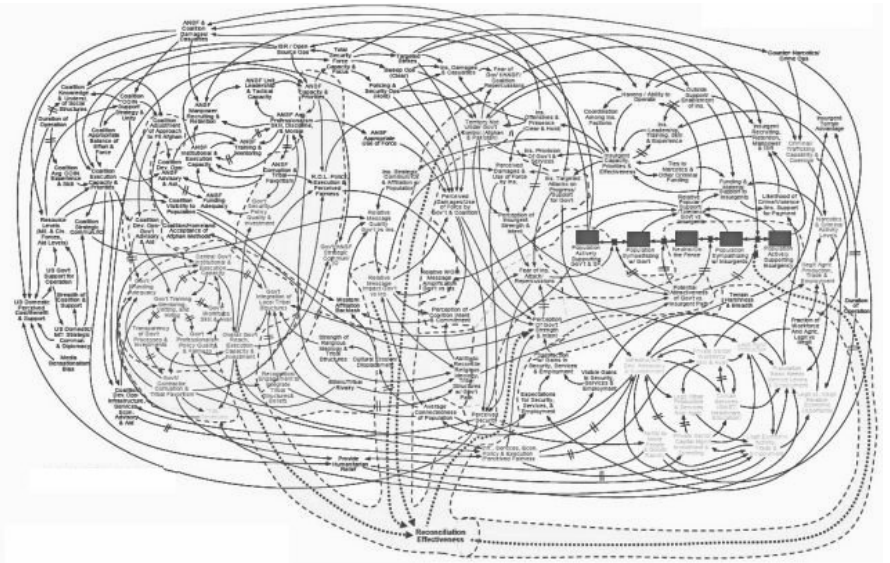
- Each method returns true if successful, false otherwise
- Assign an integer code to denote the type of problem

```
int readFile() {
    errorCode = 0;
    if (!openFile())
    errorCode = 1;
    else if (!determineFileSize())
        errorCode = 2;
    else if (!allocateMemory())
        errorCode = 3;
    else if (!readFileIntoMemory())
        errorCode = 4;
    else if (!closeFile())
        errorCode = 5;
    return errorCode;
}
```

Traditional approach to error handling

Problems of the error code solution:

- Calling method may forget to check for error code
- Codes are arbitrary
  - What does code 3 mean?
- Overly-complex code
- Programming for success is replaced by programming for failure



Exceptions: Errors in Java

Definition: An *exception* is an event that occurs during the execution of a program that *disrupts* the normal flow of instructions.

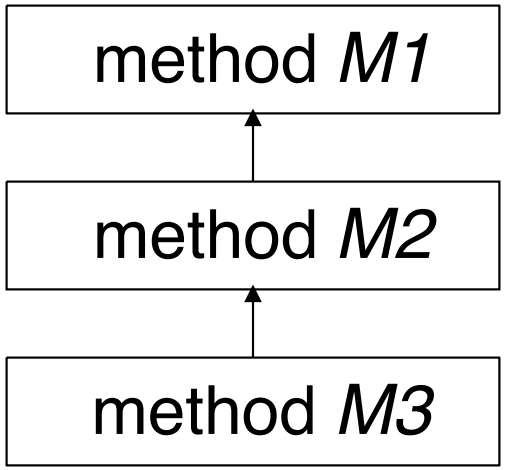
```
readFile()
{
   openFile();
   determineFileSize();
   allocateMemory();
   readFileIntoMemory();
   closeFile();
}
```

More specific exceptions must be placed before more general ones

```
readFile() {
    try {
     openFile();
     determineFileSize();
     allocateMemory();
     readFileIntoMemory();
     closeFile();
    }
    catch (fileOpenFailed a1) { Act1 }
    catch (fileSizeFailed a2) { Act2 }
    catch (memoryAllFailed a3) { Act3 }
    catch (readFailed a4) { Act4 }
    catch (fileCloseFailed a5) { Act5 }
    finally { EndAct }
}
```

Always executed (Optional)

Mechanism of handling exceptions



Advantages:

1. Separating error handling code from "regular" code
2. Propagating errors up the call stack
3. Grouping exceptions

The Java language requires that methods either catch or specify all checked exceptions that can be thrown within the scope of that method!!!

```
void readFileIntoMemory() throws readFailed, IOException
void openFile() throws fileOpenFailed, EOFException
```

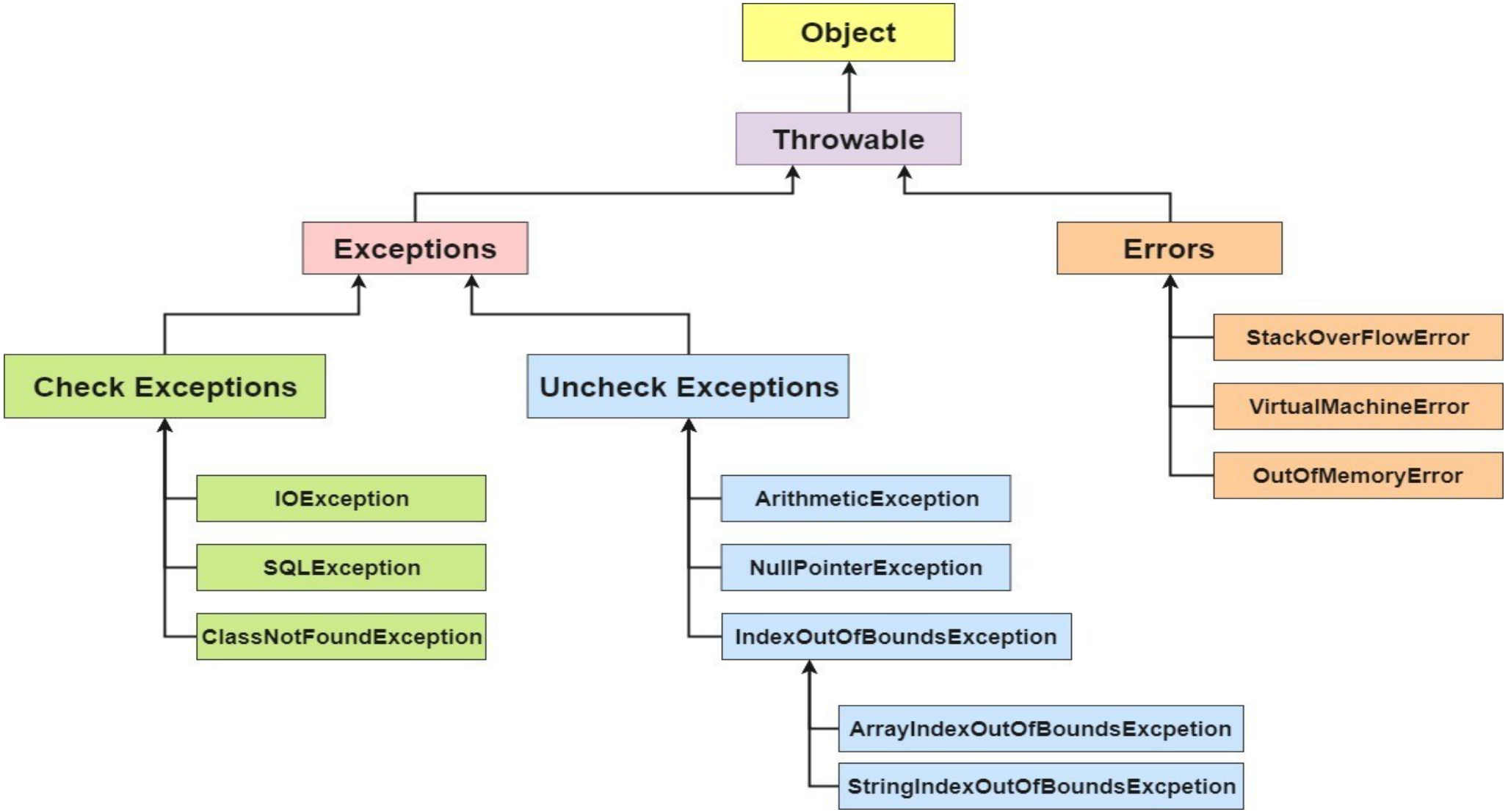
Different categories

- Exceptions come in three categories
- Errors
  - Things that can go wrong at any time, and are generally unfixable
  - Running out of memory, the JVM crashing
- Unchecked exceptions
  - Things that can go wrong at any time, and are generally fixable
  - It would clutter the code having to handle these everywhere
  - Null pointers, array index out of bounds
- Checked exceptions
  - You are trying something that has a well-known risk that a programmer should pay some attention to
  - Opening a file without permissions, invalid SQL statement

Different categories

- Why are there different categories?
- If one tried to specify all the different things that could possibly go wrong every time an operation was performed, code would be nothing but warnings about things that could go wrong
  - For example, every calculation would need warnings about division by zero, numeric overflow, etc.
- Sometimes one cannot do anything to fix things
  - Say, when the JVM runs out of memory
- The three categories attempt to provide a way to handle problems without overwhelming the code

Checked and unchecked exceptions



Common build-in exceptions

1. **ArithmeticException:** division by zero, square root of negative numbers, etc.
2. **ArrayIndexOutOfBoundsException:** array index does not exist
3. **IOException:** failure during reading, writing and searching file
   - **FileNotFoundException**
4. **NullPointerException:** use an object reference that has the *null* value
5. **NumberFormatException:** String parsed to any numerical value
6. **StringIndexOutOfBoundsException:** string index does not exist

Exception handling

- Doing something when an exception occurs is called *handling* the exception
- This is done by using a *try-catch block*
- The basic idea is that all the lines of code that will be executed if there is no exception are all together in one place
- And the code that is executed when there is an exception is all together in another place

Basic structure

```
try {
        //Block of code to try
} catch(<ExceptionClass> e) {
        //Block of code to handle errors
}
```

- Where <ExceptionClass> is the class (or parent class) of the type of exception you want to handle
- e is an object of class <ExceptionClass>
  - e will be created and filled in by the JVM if/when the exception occurs

Basic structure

```
try {
    <line of code 1>
    <line of code 2>
    ...
    <line of code n>
} catch(<ExceptionClass> e) {
        //Block of code to handle errors
}
```

- The try block stops as soon as line i throws an exception
  - All lines after line i are skipped
  - Go directly to the catch block

Basic structure

```
try {
        //Block of code to try
} catch(<ExceptionClass1> e1) {
        //Block of code to handle errors
} catch(<ExceptionClass2> e2) {
        //Block of code to handle errors
}
```

- You can have as many catch blocks as you want, as long as the exception classes are unique

Basic structure

- Only one of the catch blocks can happen
  - None is an option, if no exception occurs
- When an exception occurs in the try block

1. Java starts at the first catch block
2. Compares the class of the exception to the class declared in the catch block
3. If they are the same (or the actual exception class descends from the declared exception class) that catch block is done and all the rest are skipped
4. Otherwise, it checks the next catch block
5. Until either some block matches, or none do

Basic structure

```
try {
        //Block of code to try
} catch(<ExceptionClass> e) {
        //Block of code to handle errors
} finally {
        //Block of code to clean up
}
```

- The finally block is executed
  - After any catch block, or
  - After the try block, if there are no exceptions
- The finally block will be executed no matter what

Example of exception handler

Designing exception types

- Exceptions are classes just like any other
- Which means you can declare your own exception types
- This can by extending any of four classes
- java.lang.Throwable
  - Do not extend this class
- java.lang.Error
  - If you do this, your class will be an Error class
  - In short, unchecked and not something that should be handled by anybody who uses it
  - It is unlikely this is the one you want

Designing exception types

- java.lang.RuntimeException
  - For standard unchecked exceptions
- java.lang.Exception
  - For standard checked exceptions
  - Probably the one you want
- For the most part there is no point in adding methods or fields to an exception
  - All the important stuff is in the stack trace, which is automatically inserted into the exception object by the Throwable superclass
- Here we just override the constructor that takes a String parameter so we can add a customized message

Designing exception types

```
public class InsufficientFundsException
    extends Exception
{
    public InsufficientFundsException()
    {

    }

    public InsufficientFundsException(String reason)
    {
        super(reason);
    }
}
```


https://docs.oracle.com/javase/7/docs/api/java/lang/RuntimeException.html

Example

```
public class BankAccount {
   public void withdraw(double amount)
      throws InsufficientFundsException
   {
         if (amount > balance) {
               InsufficientFundsException exception =
            new InsufficientFundsException("Amount exceeds balance");
               throw exception;
         }
         balance = balance – amount;
   }
   …
}
```

Note: if the exception is thrown the execution does not continue

Example: short version

```
public class BankAccount {
   public void withdraw(double amount)
      throws InsufficientFundsException
   {
         if (amount > balance) {
            throw new InsufficientFundsException("Amount exceeds balance");
         }
         balance = balance – amount;
    }
    …
}
```

Guidelines

- Say you are calling a method, foo( ), that declares that it throws an exception of class E
- If you call foo, you should surround it with a try-catch block that catches E
- If you do not have a try-catch block, you must declare that your method throws E, also
  - Also true if you have a try-catch block that does not catch an exception of class E
- If you throw a checked exception using `throws`, you must declare that your method throws that exception

More detailed information

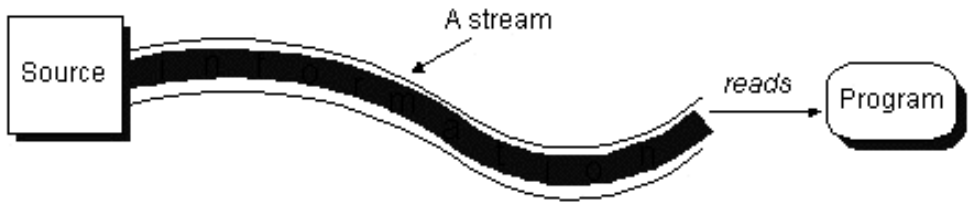
Visit the following link for further (more detailed) information:
https://www.javatpoint.com/exception-handling-in-java

STREAMS: BYTE STREAMS AND CHARACTER STREAMS

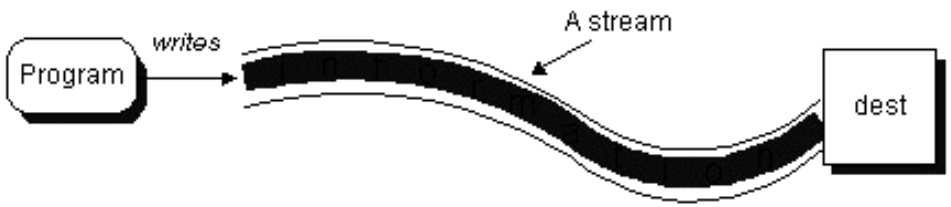
Streams

Programs receive information from an external source or send out information to an external destination. These two processes are realised with streams. ***A stream is an abstraction of a sequence of bytes.***

• To receive information, a program opens a stream on an information source (a file, memory) and reads the information serially, like this:



• A program can send information to an external destination by opening a stream to a destination and writing the information out serially, like this:



Streams

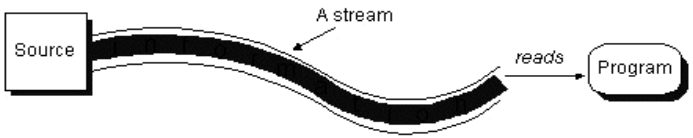
No matter **where** the information is coming from or going to and no matter **what type of data** is being read or written, the algorithms for reading and writing data are pretty much always the same.

Reading

```
open a stream
while more information
    read information
close the stream
```

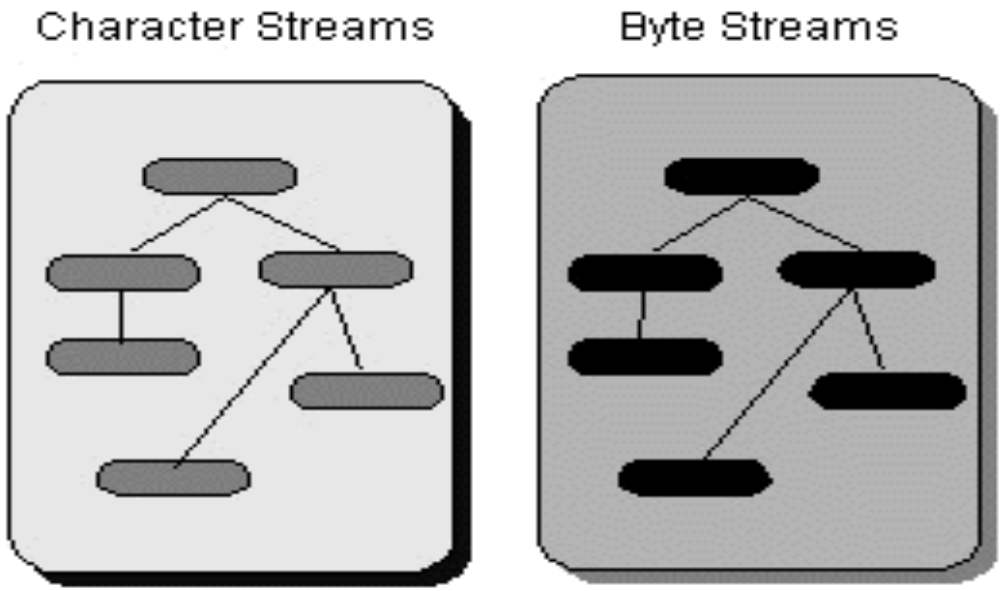
Writing

```
open a stream
while more information
    write information
close the stream
```



Streams

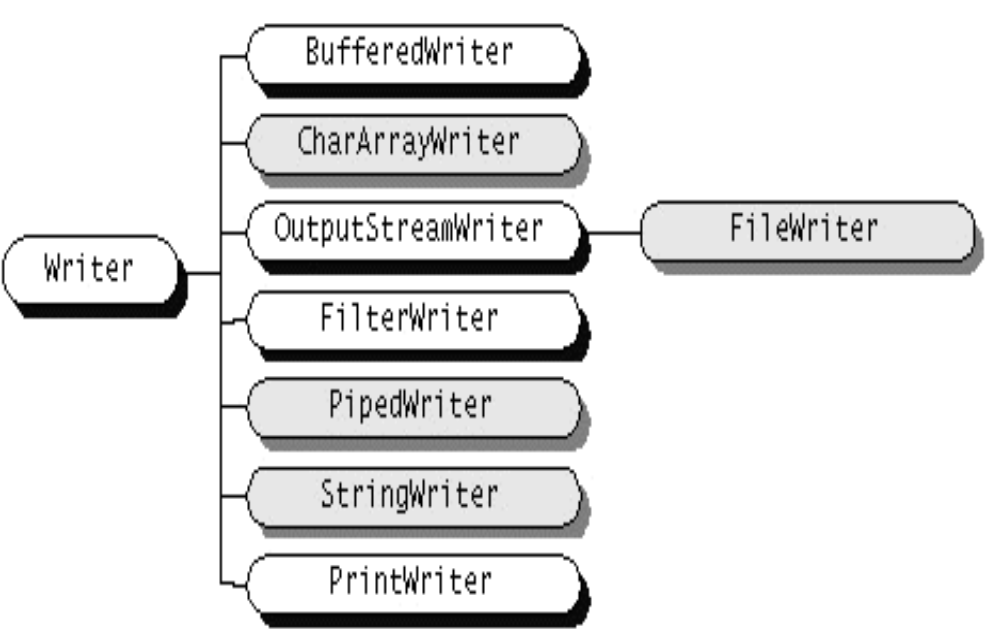
The **java.io** package contains a set of stream classes that support algorithms for reading and writing. The classes are divided into two class hierarchies based on the data type (either characters or bytes) on which they operate.

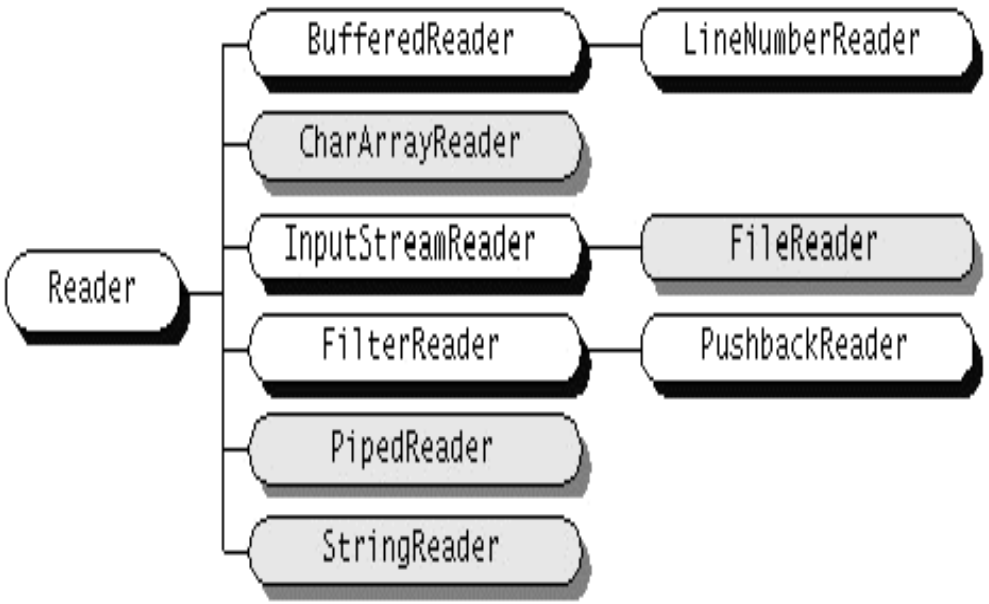


Character streams

Reader and **Writer** are the abstract superclasses for character streams in **java.io**. (**Reader/Writer)**provides the API and partial implementation for readers (writers) -- streams that read (write) 16-bit characters.

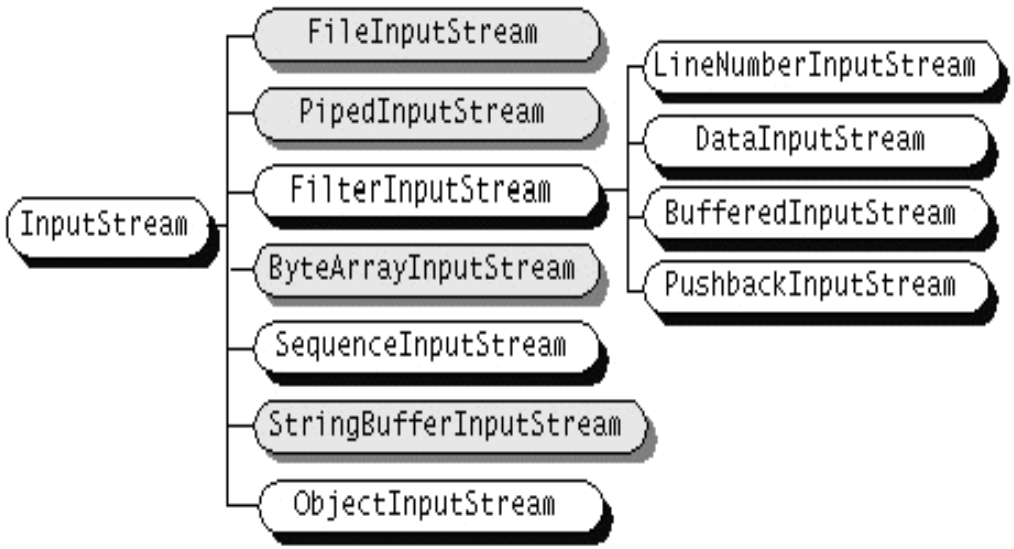
Subclasses of **Reader** and **Writer** implement specialized streams and are divided into two categories: those that read from or write to data sinks, and those that perform some sort of processing.

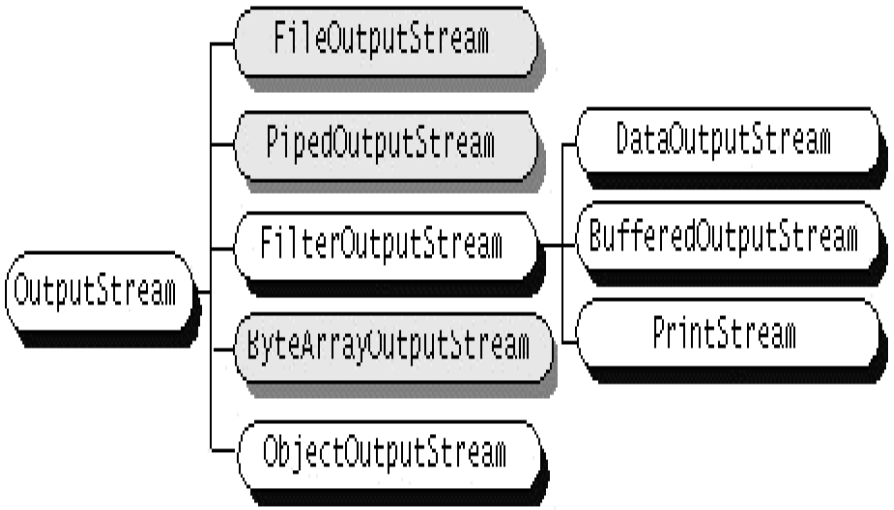




Byte streams

Programs can use the byte streams to read and write 8-bit bytes. Abstract classes **InputStream** and **OutputStream** provide the API and some implementation for input streams (streams that read 8-bit bytes) and output streams (streams that write 8-bit bytes). These streams are typically used to read and write binary data such as images and sounds.

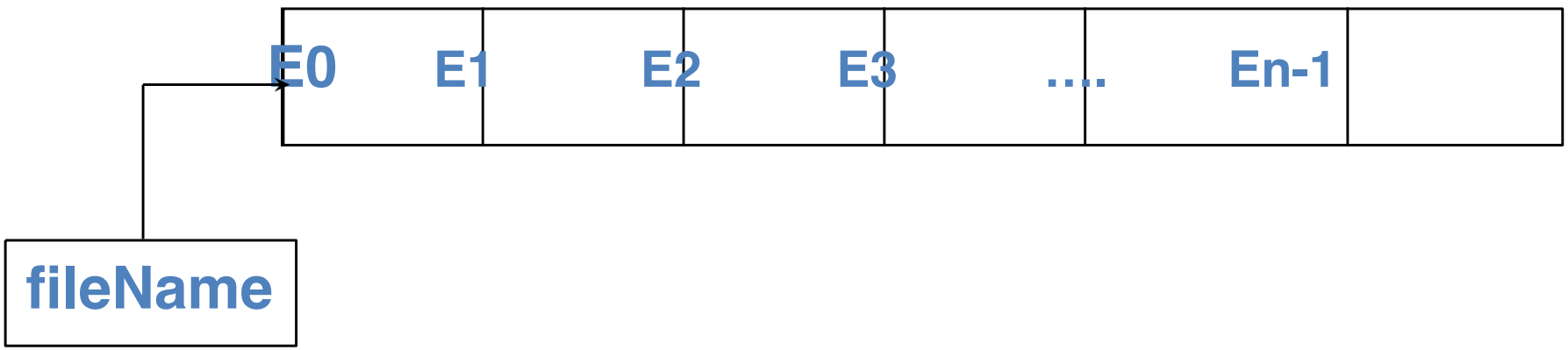




FileReader and FileWriter

These are two classes reading and writing character files. Before using them, we should first we should define the notion of file.

Definition. File is a named finite sequence of elements of a certain type that is stored in *a non-volatile storage* medium.



FileReader

`int read()` — The method returns an integer in the range 0 to 65535, or -1 if the end of stream has been reached.

```
int read(char cbuf[])
int read(char cbuf[], int offset, int length)
```

```
FileReader reader = new FileReader("Input.txt");
int next = reader.read();
char c;
if (next != -1)
    c = (char) next;
```

BufferedReader

The method **read()** of the class **FileReader** is not always convenient. That is why **BufferedReader** class is used for the efficient buffering of characters and lines. The class **BufferedReader** has the following interesting methods:

- **String readLine()**

  The method reads a line of text. A line is considered to be terminated by a line feed (**'\n'**), or a carriage return (**'\r '**). The method 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.

```
FileReader reader = new FileReader("Input.txt");
BufferedReader in = new BufferedReader(reader);
String inputLine = in.readLine();
double x = Double.parseDouble(inputLine);
reader.close();
```

Scanner

Another option is to use class **Scanner** instead of class **BufferedReader**. In this case, we can use methods like **next(), nextLine(), nextInt(),** and **nextDouble().**

```
FileReader reader = new FileReader("Input.txt");
Scanner in = new Scanner(reader);
String inputLine = in.nextLine();
double x = Double.parseDouble(inputLine);
reader.close();
```

FileWriter

void write(int c)
void write(char cbuf[])
void write(char cbuf[], int offset, int length)

- If all you want to do is write arrays of characters to a file, FileWriter will do that
- This comes in handy when working with networks or low-level disk operations

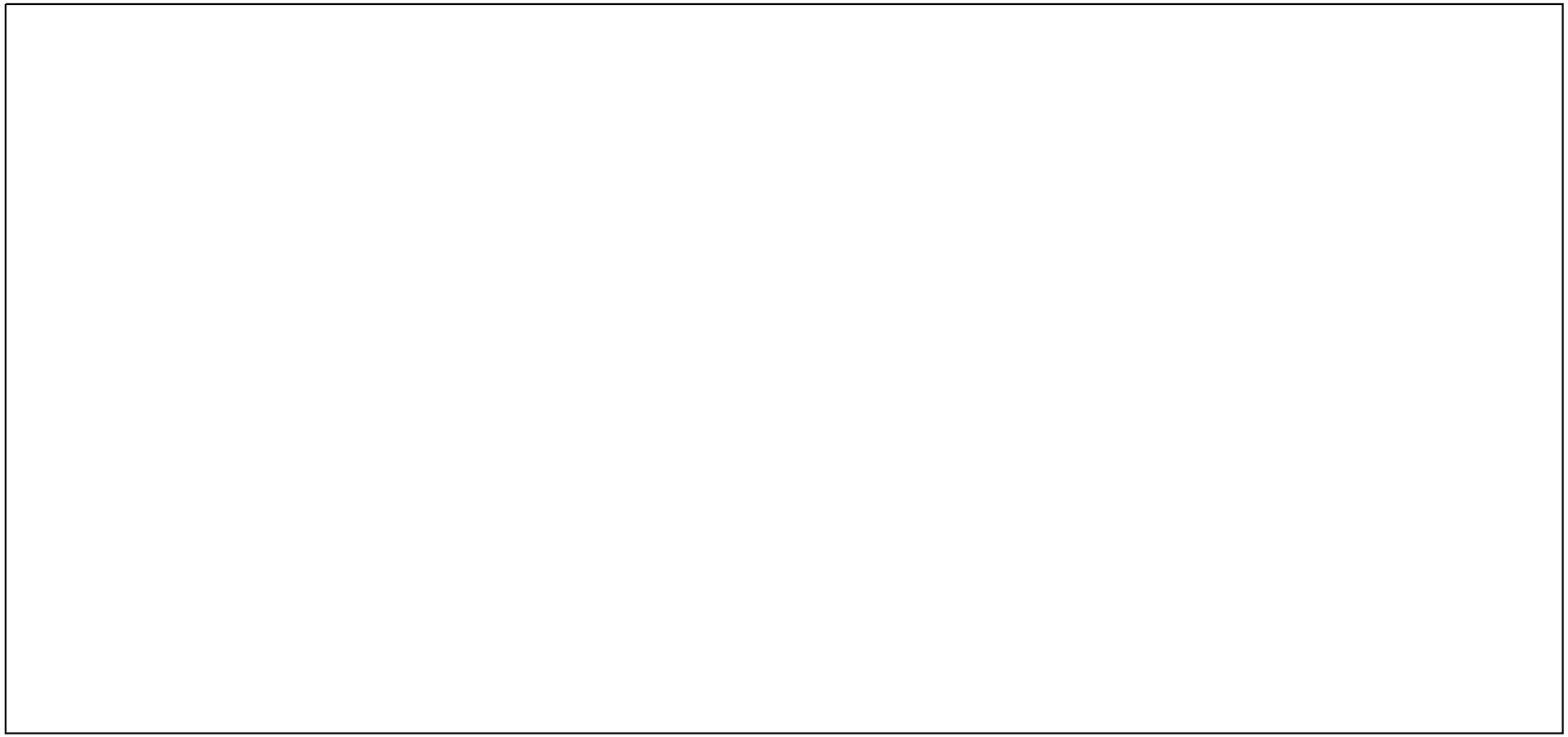
PrintWriter

- The class **PrintWriter** is generally more friendly for most applications
- It understands the different types of values that can be printed out, not just characters
- It is very similar to System.out

```
PrintWriter out = new PrintWriter("output.txt");
out.println(29.95);
out.println(new Rectangle(5,10,15,25));
out.print("Hello, World!");
out.close();
```

Example

Example: Part 2



Example: Part 3

```
public void openProcessFile() {
    Scanner in = null;
    ArrayList<Product> result = new ArrayList<Product> ();
    try { // select file name
      in = new Scanner(new FileReader(fileName));
      result = readProducts(in);
      ...
    }

    catch(FileNotFoundException e)
    { System.out.println("Bad file name");}
    catch(IOException e)
    { System.out.println("Corrupted file");}
    finally
    { if (in != null)
      try { in.close(); }
      catch(IOException e)
      { System.out.println("Error closing file"); }
    }
}
```

Example: Part 3 (short version)

```
public void openProcessFile() {
      ArrayList<Product> result = new ArrayList<Product> ();
      try(Scanner in = new Scanner(new FileReader(fileName));
      { // select file name
       result = readProducts(in);
       ...
      }

      catch(FileNotFoundException e)
      { System.out.println("Bad file name");}
      catch(IOException e)
      { System.out.println("Corrupted file");}
}
```

Summary

- Java's IO framework is powerful, but complex
  - One can change from using a file to a network connection to the keyboard/screen with a very small change in code
  - Using its full power requires understanding the Decorator pattern
  - Which is too much for right now
- Your best bet is
- Decide whether to use characters or standard types
- If you use characters, stick with FileReader/BufferedReader and FileWriter
- If you use standard types, stick with Scanner and PrintWriter

File Class

- File class describes disk files and directories
- Create a File object

```
File inputFile = new File("input.txt");
```

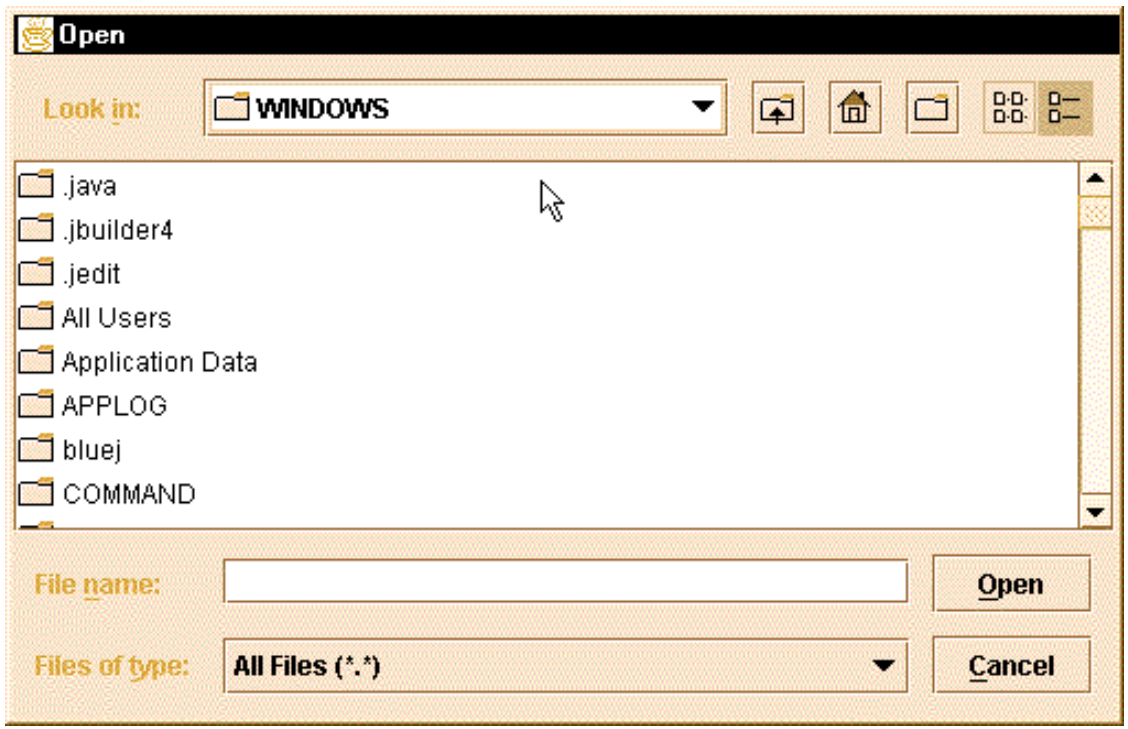
- Some file methods:

delete
renameTo
exists

- Constructing a **FileReader** from a **File** object

```
FileReader reader = new FileReader(inputFile);
```

JFileChooser Dialog



JFileChooser

Methods:

- Construct a file chooser object: **JFileChooser**
- To create dialog window: **showOpenDialog** or **showSaveDialog**

  (You can specify null or the user interface component over which to pop up the dialog)

- If the user chooses a file, these methods return: **JFileChooser.APPROVE_OPTION**
- If the user cancels the selection, these methods return: **JFileChooser.CANCEL_OPTION**
- If a file is chosen, you can use **GetSelectedFile** method to obtain a **File** object describing the file.

JFileChooser: Example

```
JFileChooser chooser = new JFileChooser();
FileReader in = null;
if (chooser.showOpenDialog(null) ==
      JFileChooser.APPROVE_OPTION )
{
      File selectedFile = chooser.getSelectedFile();
      in = new FileReader(selectedFile);
}
```

Command line arguments

- A Java application can accept any number of arguments from the command line.
- This allows the user to specify configuration information when the application is launched.

https://docs.oracle.com/javase/tutorial/essential/environment/cmdLineArgs.html

Command line arguments

- In the following example we will:
- Accept the name of a file from the command line
- Write a couple of doubles to it
- Read those doubles from it

Command line arguments

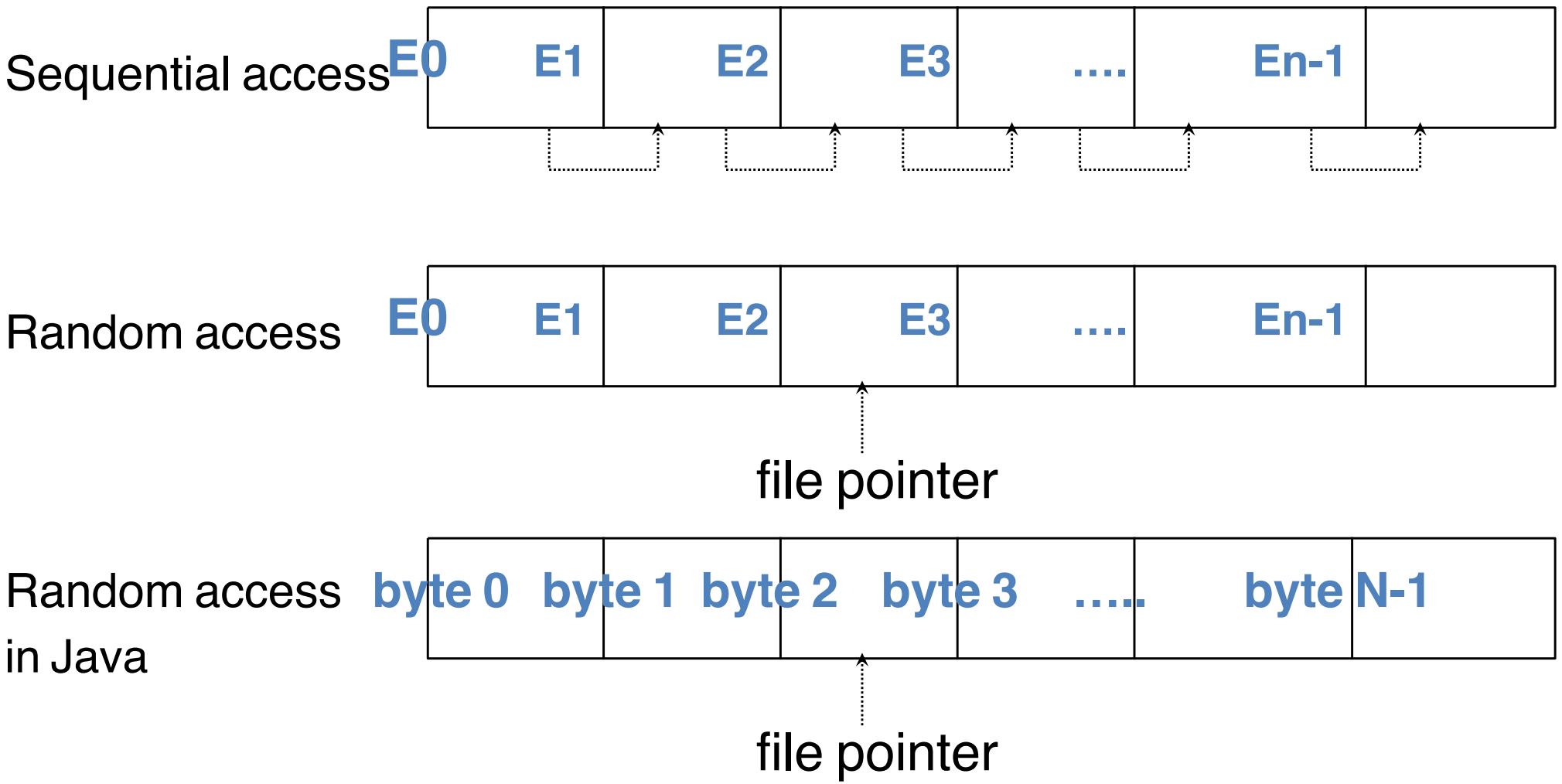
```
public static void main(String[] args) {
    try {
        FileWriter writer = new FileWriter(args[0]);
        PrintWriter out = new PrintWriter(writer);
        out.println(29.95);
        out.println(39.05);
        writer.close();

        FileReader reader = new FileReader(args[0]);
        BufferedReader in = new BufferedReader(reader);
        String inputLine = in.readLine();
        double x = Double.parseDouble(inputLine);
        System.out.println(x);
        inputLine = in.readLine();
        x = Double.parseDouble(inputLine);
        System.out.println(x);
        reader.close();
    }
    catch(Throwable e){ System.out.println("ourError");}
} // to run the program type "java readWrite io.txt"
```

Random access files

- A random access file behaves like a large array of bytes stored in the file system
  - This is different than a stream, which was one char/byte after another

https://docs.oracle.com/javase/7/docs/api/java/io/RandomAccessFile.html

Random access files



Class: RandomAccessFile (java.io.* package)

Defining RandomAccessFile objects:

```
RandomAccessFile <fileVariable>;
<fileVariable> = new RandomAccessFile(<file>, "rw");
```

File Pointer's Methods:

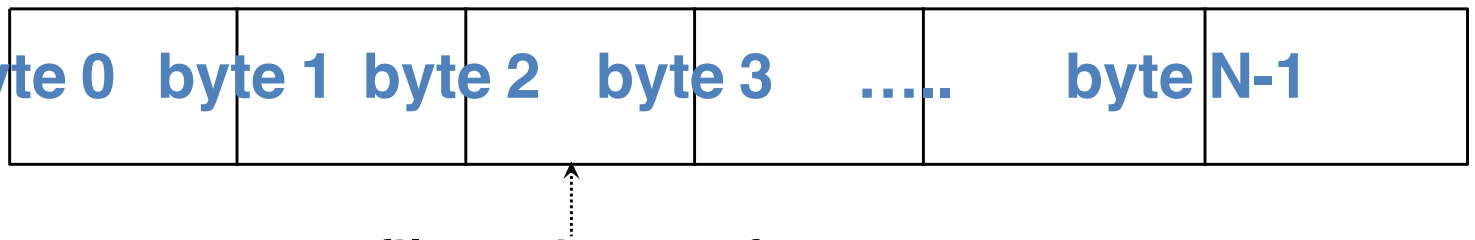
```
<fileVariable>.getFilePointer();
<fileVariable>.seek(<positionInFile>);
```

The Length Method:

```
<fileVariable>.length();
```

The Close Method:

```
<fileVariable>.close(); //don't use <fileVariable> after!
```

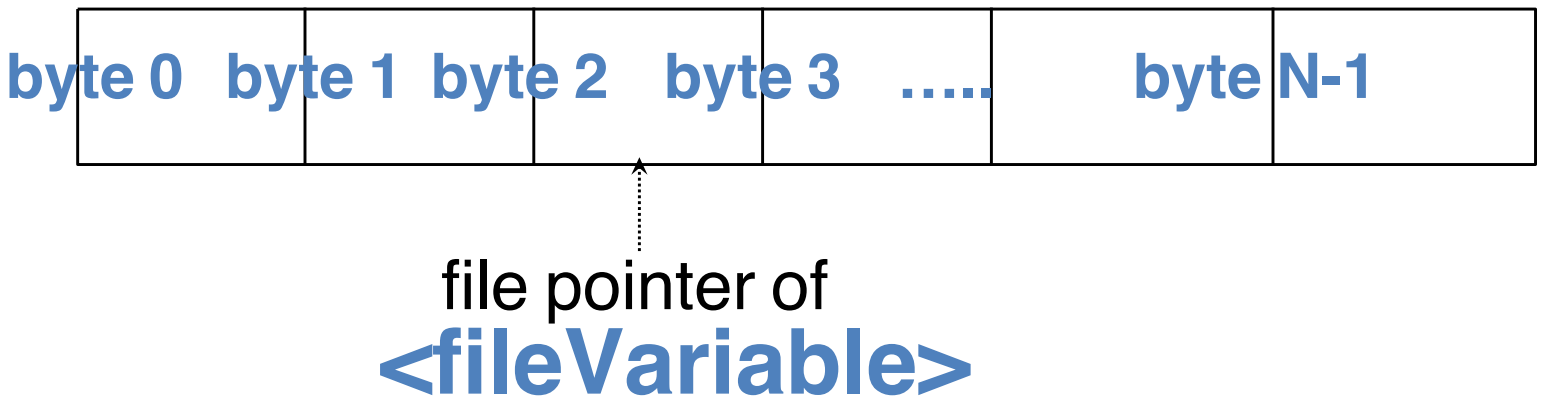


https://docs.oracle.com/javase/7/docs/api/java/io/RandomAccessFile.html#constructor_detail

Class: RandomAccessFile: 2

The *write* methods:

<fileVariable>.writeChars(<String>); //<strLength>*2 bytes
<fileVariable>.writeInt(<int>); // 4 bytes
<fileVariable>.writeDouble(<double>);// 8 bytes



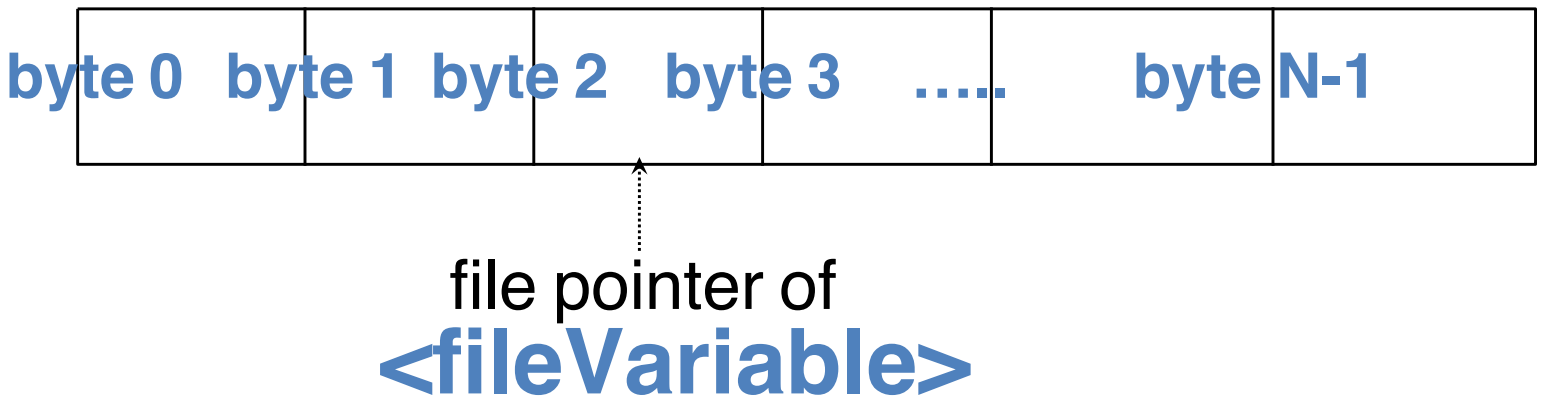
https://www.javatpoint.com/java-randomaccessfile-class

Class: RandomAccessFile: 3

The *read* methods:

```
<fileVariable>.readChar();  //  2  bytes
<fileVariable>.readInt();   //  4  bytes
<fileVariable>.readDouble();   //  8  bytes
```

All RandomAccessFile methods throw exceptions (IO, EOF, etc.) !!!

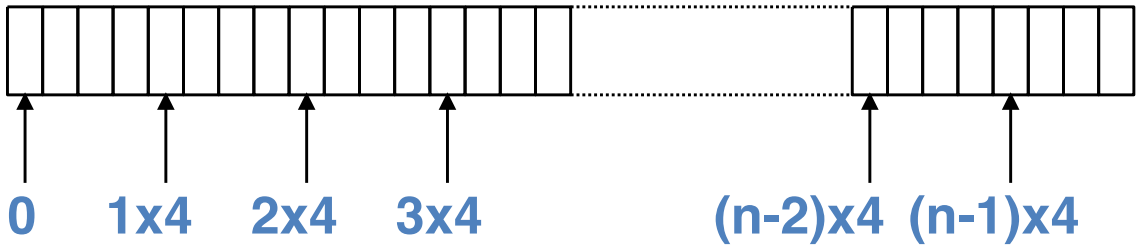


Reading/writing random access files

1. Integers

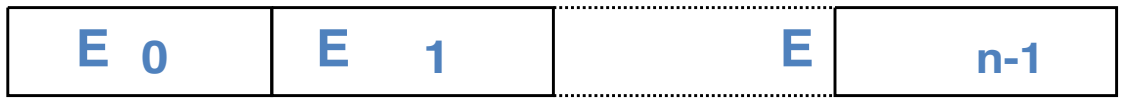
Logical format

Physical format

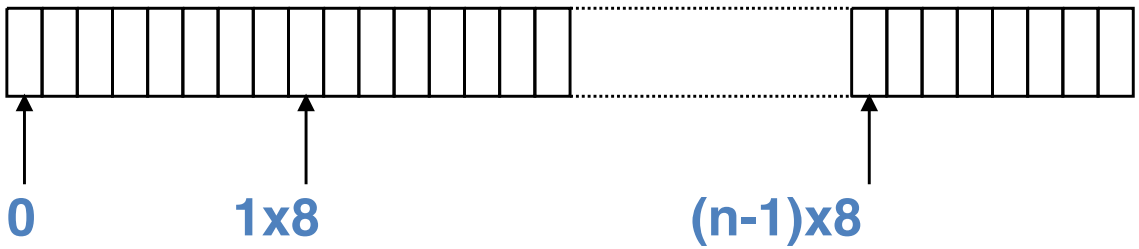


2. Doubles

Logical format



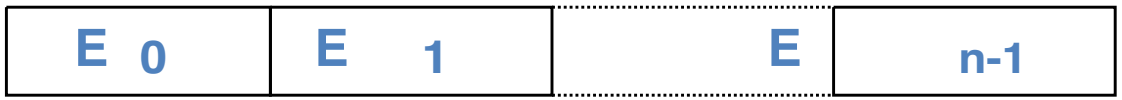
Physical format



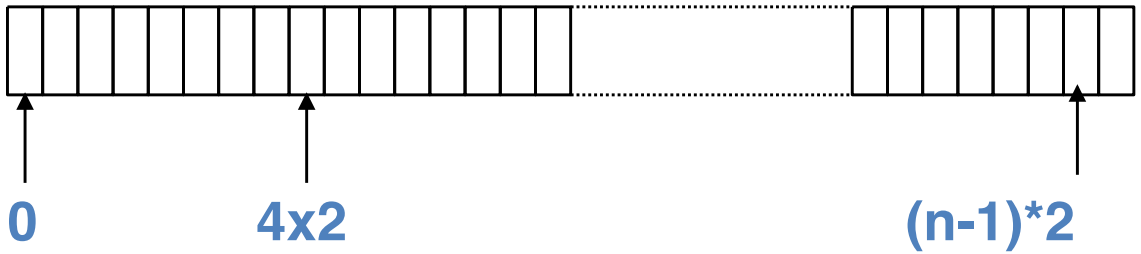
Reading/writing random access files

3. Strings

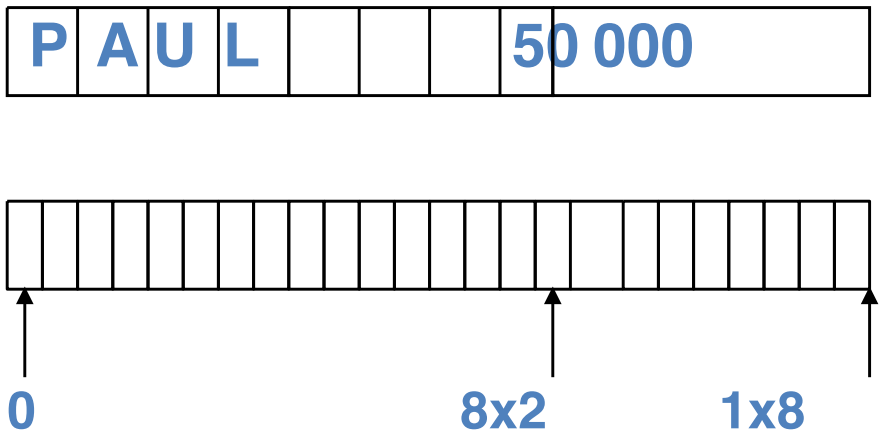
Logical format



Physical format



4. Mixed Data



Object streams

- The class **ObjectOutputStream** can save entire objects in files
- The class **ObjectInputStream** can read entire objects from files
- Never use them
- There is an entire field of techniques for this that work better
  - Called “persistence”

Learning goals

- Understand and be able to utilize *exception handling* techniques
- Identify differences between *checked* and *unchecked exceptions* and their implications
- Be able to implement different input/output methods based on *streams* (using files, command line, RAM, objects, etc.)
- Be able to use *command line arguments* in our classes

What we have learned

1. Exceptions and exception handling
2. Checked and unchecked exceptions
3. Streams: Byte Streams and Character Streams
4. FileReader and FileWriter
5. File and JFileChooser
6. Command Line Arguments
7. Random Access Files
8. Object Streams