COMP 6481
Programming and Problem Solving

CHAPTER 10 - JAVA I/O

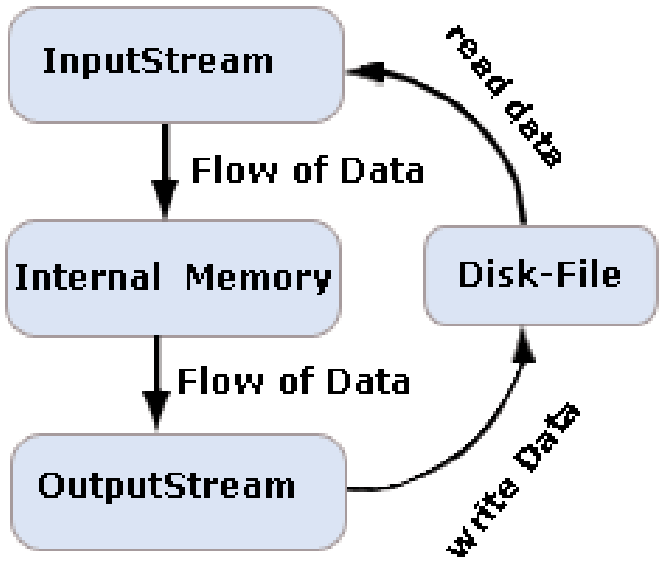
http://odditymall.com/input-output-baby-onsie

Streams

A ***stream*** is an object that enables the flow of data between a source and a destination

- ex: input file, output file, keyboard, screen, …

The `java.io` package contains many classes that allow us to define various streams.



http://letrungthang.blogspot.ca/2011/01/stream-is-type-of-object-from-which-we.html

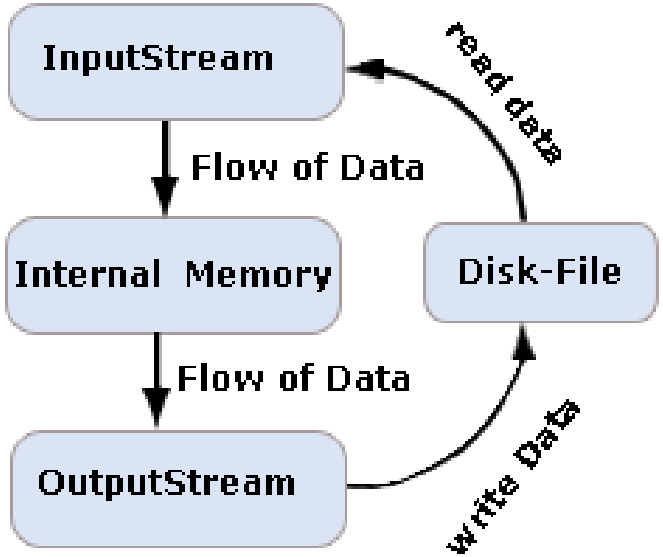
Streams

Input streams flow into your program

- ex: from the keyboard or from a file
  - `System.in` is an input stream

Output streams flow out of your program

- ex: to a screen or to a file
  - `System.out` is an output stream



http://letrungthang.blogspot.ca/2011/01/stream-is-type-of-object-from-which-we.html

Standard I/O

There are 3 standard I/O streams defined as 3 variables in the **`System`** class

1. **`static InputStream in`**
   - called *standard input*
   - defined by **`System.in`**
   - usually the keyboard
2. **`static PrintStream err`**
   - called *standard error*
   - defined by **`System.err`**
   - usually the monitor and in red
3. **`static PrintStream out`**
   - called *standard output*
   - defined by **`System.out`**
   - usually the monitor

Standard I/O: RedirectionDemo.java

Can change the standard I/O streams

- `System.setIn, System.setOut, System.setErr`

ex:

```
PrintStream out = new PrintStream(
                        new BufferedOutputStream(
                        new FileOutputStream("test.txt")));

System.setOut(out);
System.out.println("This is a test.");
out.close();
```

I/O Streams

Input vs **Output** streams

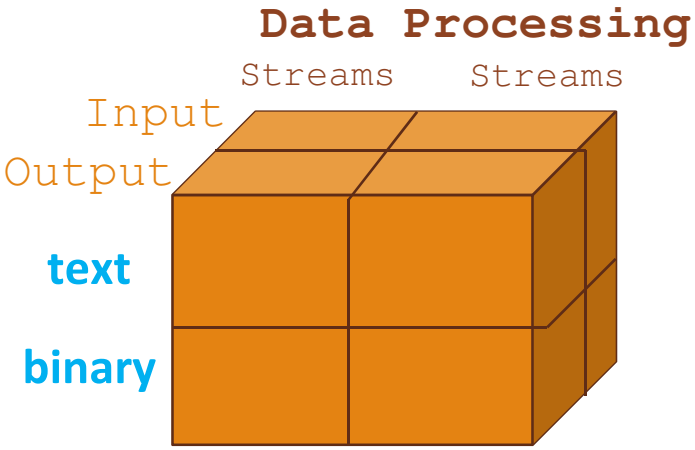
- input: read information
- output: write information

Text vs **Binary** Files

- text format (characters)
- byte format (binary information)

Data stream vs **Processing** streams

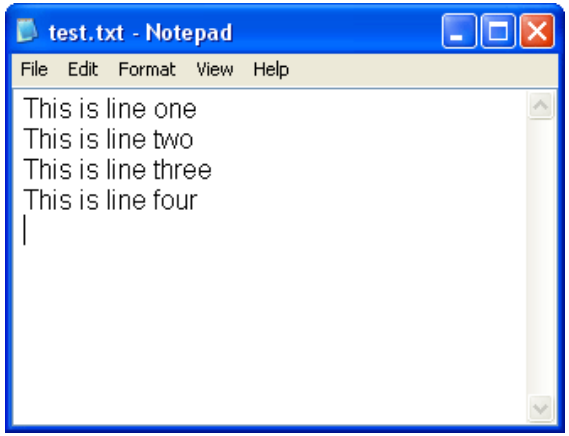
- *data stream*: acts as either a source or destination
- *processing stream*: alters or manipulates the basic data in the stream



Text File

Data is represented as a sequence of characters (also called ASCII file)

- Integer 12,345 stored as 5 characters (5 bytes)
  '1' '2' '3' '4' '5'



Advantages:

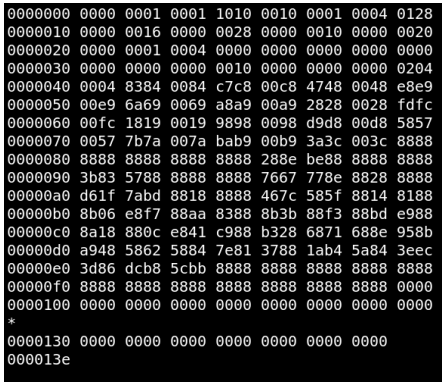
- Human-readable form

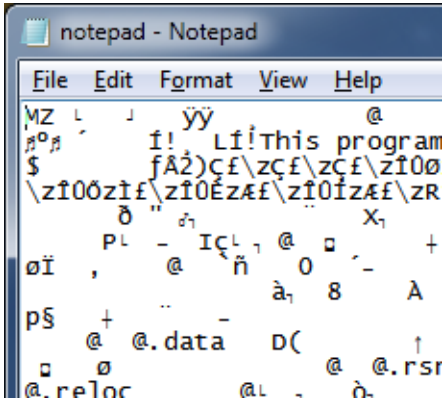
Disadvantages:

- less compact
- less efficient

Use classes **Reader** and **Writer** and their subclasses
(will look at details in a few slides)

Binary Files





Data is represented as a sequence of bytes

- ex: integer 12,345 is stored as four bytes **`0 0 48 57`**
- *(12,345 = 48x2^8 + 57)*

Advantages:

- More compact and more efficient (native representation)

Disadvantages:

- data can be stored differently from machine to machine
- cannot be read by humans

BUT: in Java, more portable

- can be read by Java on any machine

Typically used to read and write sounds and images

Use **`InputStream`** and **`OutputStream`** classes and their subclasses (will look at details in a few slides)

Data vs. Processing Streams

A *data stream* represents a particular *source* or *destination*

- ex: a string in memory or a file on disk

A *processing stream* (also called a *filtering stream*) *manipulates* the data in the stream

- ex: It may convert the data from one format to another
- ex: It may buffer the stream

Just Checking

The output stream connected to the computer screen is:

A. System.screen
B. System.keyboard
C. System.in
D. System.out

Just Checking

The stream that is automatically available to your Java code is:

A. System.out
B. System.in
C. System.err
D. All of the above
E. None of the above

Class Hierarchy for Package `java.io`

- `java.lang.Object`
  - `java.io.File`

binary input streams

- `java.io.InputStream`
  - `java.io.FileInputStream`
  - `java.io.ObjectInputStream`
  - `java.io.FilterInputStream`
  - …

binary output streams

- `java.io.OutputStream`
  - `java.io.FileOutputStream`
  - `java.io.ObjectOutputStream`
  - `java.io.FilterOutputStream`
  - …

= abstract class

text input streams

- `java.io.Reader`
  - `java.io.BufferedReader`
  - `java.io.InputStreamReader`
    - `java.io.FileReader`
  - …

text output streams

- `java.io.Writer`
  - `java.io.BufferedWriter`
  - `java.io.OutputStreamWriter`
    - `java.io.FileWriter`
  - `java.io.PrintWriter`
  - …

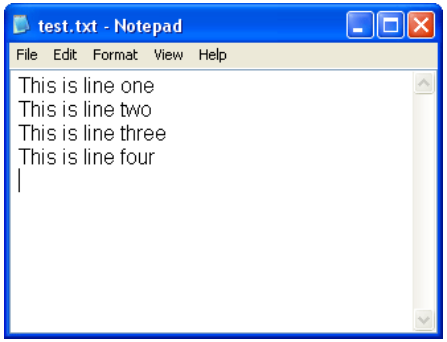
Random Access Files

- `java.io.RandomAccessFile`

Writing Text Files: `PrintWriter` class

- Can write strings, int, floats, Objects, etc. directly to output file
  - Using `print()`, `println()`, `printf()` methods
  - Same style as `System.out.println()`
- Simple and flexible

Writing Text Files



To write to a file:

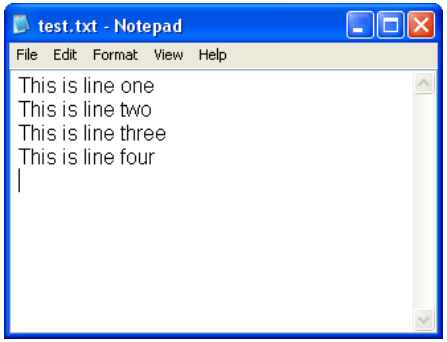
1. construct a **`PrintWriter`** object

```
PrintWriter out = new PrintWriter(
        new FileOutputStream("output.txt"));
```

```
PrintWriter out = new PrintWriter("output.txt");
```

- If file doesn't exist, an empty file is created
- If file already exists, it is emptied before the new data are written into it

Writing Text Files



To write to a file:

1. construct a **`PrintWriter`** object (continued)

   - If you want to **append** to an existing file:

```
PrintWriter out = new PrintWriter(
                        new FileOutputStream(
                             "output.txt", true));
```

Writing Text Files

2. Use **`print()`**, **`println()`** or **`printf()`** to write into a **`PrintWriter`**:

```
out.println(29.95);
out.println(new Rectangle(5, 10, 15, 25));
out.println("Hello, World!");
```

3. Close the file when you are done:

```
out.close();
```

- Otherwise, not all of the output may be written to the disk file

Error in book

Since Java 5, the class **`PrintWriter`** has a constructor that takes a file name as argument.

- So instead of (as the book says)

```
PrintWriter out =
          new PrintWriter(
                new FileOutputStream("output.txt"));
```

- you can do (as the slides say)

```
PrintWriter out = new PrintWriter("output.txt");
```

Output Buffering

Many text output streams (**`Writer`** class) are not buffered

- each write operation causes characters to be written immediately to the stream
- very inefficient
- so wrap a **`BufferedWriter`** around a **`Writer`** object that does not buffer (more efficient)

```
PrintWriter out = new PrintWriter( new
BufferedWriter(new FileWriter("myFile.out")));
```

Exceptions when writing to a Text File

- When a text file is opened, a **`FileNotFoundException`** can be thrown
  - In this context it actually means that the file could not be created
  - This type of exception can also be thrown when a program attempts to open a file for reading and there is no such file

- All exceptions from IO are **checked**
  - so the file should be opened inside a **`try`** block
  - A **`catch`** block should catch and handle the possible exception

Example:

`TextFileOutputDemo.java`	with Try/Catch Block
`TextFileOutputDemo2.java`	with no Try/Catch Block but Throws

TextFileOutputDemo.java

```
PrintWriter outputStream = null; // needs to be declared outside of try
try {
    outputStream = new PrintWriter (new BufferedWriter(
                        new PrintWriter("stuff.txt")));
    System.out.println("Writing to file.");
    outputStream.println("The quick brown fox");
    outputStream.println("jumped over the lazy dog.");
}
catch(FileNotFoundException e) {
    System.out.println("Error opening the file stuff.txt.");
    System.exit(0);
}
// finally must be right after catch
finally {
     if (outputStream != null)
       outputStream.close();
     // the close() in PrintWriter does not throw any exception...
}

System.out.println("End of program.");
```

TextFileOutputDemo2.java

```
 public static void main(String[] args) throws
                FileNotFoundException
{
        PrintWriter outputStream = null;

        outputStream = new PrintWriter (new
          BufferedWriter(new PrintWriter("stuff.txt")));

        System.out.println("Writing to file.");
        outputStream.println("The quick brown fox");
        outputStream.println("jumped over the lazy dog.");

        outputStream.close();
        System.out.println("End of program.");
}
```

Writing to a Text File

- When a program is finished writing to a file, it should always close the stream connected to that file

```
outputStreamName.close();
```

  - This allows the system to release any resources used to connect the stream to the file
  - If the program does not close the file before the program ends, Java will close it automatically, but it is safest to close it explicitly

Writing to a Text File

- Output streams connected to files are usually *buffered*
  - Rather than physically writing to the file as soon as possible, the data is saved in a temporary location (*buffer*)
  - When enough data accumulates, or when the method **`flush`** is invoked, the buffered data is written to the file all at once
  - This is more efficient, since physical writes to a file can be slow

Writing to a Text File

- The method **`close`** invokes the method **`flush`**, thus insuring that all the data is written to the file
  - If a program relies on Java to close the file, and the program terminates abnormally, then any output that was buffered may not get written to the file
  - Also, if a program writes to a file and later reopens it to read from the same file, it will have to be closed first anyway
  - The sooner a file is closed after writing to it, the less likely it is that there will be a problem

Some Methods of the Class `PrintWriter` (Part 1 of 3)

Display 10.2 Some Methods of the Class `PrintWriter`

`PrintWriter` and `FileOutputStream` are in the `java.io` package.

```
public PrintWriter(OutputStream streamObject)
```

This is the only constructor you are likely to need. There is no constructor that accepts a file name as an argument. If you want to create a stream using a file name, you use

```
new PrintWriter(new FileOutputStream(File_Name))
```

When the constructor is used in this way, a blank file is created. If there already was a file named *File_Name*, then the old contents of the file are lost. If you want instead to append new text to the end of the old file contents, use

```
new PrintWriter(new FileOutputStream(File_Name, true))
```

(For an explanation of the argument `true`, read the subsection "Appending to a Text File.")

When used in either of these ways, the `FileOutputStream` constructor, and so the `PrintWriter` constructor invocation, can throw a `FileNotFoundException`, which is a kind of `IOException`.

If you want to create a stream using an object of the class `File`, you can use a `File` object in place of the *File_Name*. (The `File` class will be covered in Section 10.3. We discuss it here so that you will have a more complete reference in this display, but you can ignore the reference to the class `File` until after you've read that section.)

(continued)

Some Methods of the Class **`PrintWriter`** (Part 2 of 3)

Display 10.2 Some Methods of the Class `PrintWriter`

```
public void println(Argument)
```

The *Argument* can be a string, character, integer, floating-point number, `boolean` value, or any combination of these, connected with + signs. The *Argument* can also be any object, although it will not work as desired unless the object has a properly defined `toString()` method. The *Argument* is output to the file connected to the stream. After the *Argument* has been output, the line ends, and so the next output is sent to the next line.

```
public void print(Argument)
```

This is the same as `println`, except that this method does not end the line, so the next output will be on the same line.

(continued)

Some Methods of the Class `PrintWriter` (Part 3 of 3)

Display 10.2 Some Methods of the Class `PrintWriter`

```
public PrintWriter printf(Arguments)
```

This is the same as `System.out.printf`, except that this method sends output to a text file rather than to the screen. It returns the calling object. However, we have always used `printf` as a `void` method.

```
public void close()
```

Closes the stream's connection to a file. This method calls `flush` before closing the file.

```
public void flush()
```

Flushes the output stream. This forces an actual physical write to the file of any data that has been buffered and not yet physically written to the file. Normally, you should not need to invoke `flush`.

Reading From a Text File Using **`Scanner`**

- **`Scanner`** is not in java.io (where is it?)
- **`Scanner`** can be used for reading from the keyboard as well as reading from a text file
- Same methods for reading from the keyboard
  - **`nextBoolean(), nextByte(), nextInt(), hasNextLine(), nextFloat(), nextDouble(), next(), nextLine()…`**

```
Scanner inFile = new Scanner( new FileReader("input.txt"));
int age = inFile.nextInt();
```

Reading Input from a Text File Using **`Scanner`** (Part 1 of 3)

Display 10.3 Reading Input from a Text File Using Scanner

```
import java.util.Scanner;
import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class TextFileScannerDemo
{
    public static void main(String[] args)
    {
        System.out.println("I will read three numbers and a line");
        System.out.println("of text from the file morestuff.txt.");

        Scanner inputStream = null;

        try
        {
            inputStream =
               new Scanner(new FileInputStream("morestuff.txt"));
        }
```

(continued)

Reading Input from a Text File Using **`Scanner`** (Part 2 of 3)

Display 10.3 Reading Input from a Text File Using Scanner

```
        catch(FileNotFoundException e)
        {
            System.out.println("File morestuff.txt was not found");
            System.out.println("or could not be opened.");
            System.exit(0);
        }
            int n1 = inputStream.nextInt( );
            int n2 = inputStream.nextInt( );
            int n3 = inputStream.nextInt( );

            inputStream.nextLine(); //To go to the next line

            String line = inputStream.nextLine( );

```

(continued)

Reading Input from a Text File Using **Scanner** (Part 3 of 3)

Display 10.3 Reading Input from a Text File Using Scanner

```
            System.out.println("The three numbers read from the file are:");
            System.out.println(n1 + ", " + n2 + ", and " +  n3);

            System.out.println("The line read from the file is:");
            System.out.println(line);

            inputStream.close( );
        }
    }
```

File morestuff.txt

```
1 2
3 4
Eat my shorts.
```

This file could have been made with a text editor or by another Java program.

Display 10.3 Reading Input from a Text File Using Scanner

Screen Output

```
I will read three numbers and a line
of text from the file morestuff.txt.
The three numbers read from the file are:
1, 2, and 3
The line read from the file is:
Eat my shorts.
```

Reading Input from a Text File Using **`Scanner`** (Part 4 of 4)

Display 10.3 Reading Input from a Text File Using Scanner

SCREEN OUTPUT

```
I will read three numbers and a line
of text from the file morestuff.txt.
The three numbers read from the file are:
1, 2, and 3
The line read from the file is:
Eat my shorts.
```

Reading From a Text File Using **Scanner**

```
Scanner inFile = new Scanner( new FileReader("input.txt"));
int age = inFile.nextInt();
```

- if you try to read beyond the end of a file, a **NoSuchElementException** will be thrown

- can use **hasNext…** methods to avoid going beyond EOF
  - **hasNextLine(), hasNextInt(), …**

Checking for the End of a Text File with **`hasNextLine`** (Part 1 of 4)

Display 10.4 Checking for the End of a Text File with `hasNextLine`

```
import java.util.Scanner;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.io.FileOutputStream;

public class HasNextLineDemo
{
    public static void main(String[] args)
    {
        Scanner inputStream = null;
        PrintWriter outputStream = null;
```

(continued)

Checking for the End of a Text File with **`hasNextLine`** (Part 2 of 4)

Display 10.4 Checking for the End of a Text File with `hasNextLine`

```
try
{
   inputStream =
      new Scanner(new FileInputStream("original.txt"));
   outputStream = new PrintWriter(
                     new FileOutputStream("numbered.txt"));
}
catch(FileNotFoundException e)
{
   System.out.println("Problem opening files.");
   System.exit(0);
}

String line = null;
int count = 0;
```

(continued)

Checking for the End of a Text File with **`hasNextLine`** (Part 3 of 4)

Display 10.4 Checking for the End of a Text File with hasNextLine

```
        while (inputStream.hasNextLine( ))
        {
            line = inputStream.nextLine( );
            count++;
            outputStream.println(count + " " + line);
        }

        inputStream.close( );
        outputStream.close( );
    }

}
```

(continued)

Checking for the End of a Text File with **`hasNextLine`** (Part 4 of 4)

Display 10.4 Checking for the End of a Text File with `hasNextLine`

File original.txt

```
Little Miss Muffet
sat on a tuffet
eating her curves away.
Along came a spider
who sat down beside her
and said "Will you marry me?"
```

File numbered.txt (after the program is run)

```
1 Little Miss Muffet
2 sat on a tuffet
3 eating her curves away.
4 Along came a spider
5 who sat down beside her
6 and said "Will you marry me?"
```

Example: Line ‘numberer’

Read all lines of a file and send them to an output file, preceded by line numbers

input file:

```
Mary had a little lamb
Whose fleece was white as snow.
And everywhere that Mary went,
The lamb was sure to go!
```

output file:

```
/* 1 */ Mary had a little lamb
/* 2 */ Whose fleece was white as snow.
/* 3 */ And everywhere that Mary went,
/* 4 */ The lamb was sure to go!
```

LineNumberer.java

Checking for the End of a Text File with **`hasNextInt`** (Part 1 of 2)

Display 10.5 Checking for the End of a Text File with hasNextInt

```
import java.util.Scanner;
import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class HasNextIntDemo
{
    public static void main(String[] args)
    {
        Scanner inputStream = null;

        try
        {
           inputStream =
              new Scanner(new FileInputStream("data.txt"));
        }
        catch(FileNotFoundException e)
        {
           System.out.println("File data.txt was not found");
           System.out.println("or could not be opened.");
           System.exit(0);
        }
```

(continued)

Checking for the End of a Text File with **`hasNextInt`** (Part 2 of 2)

Display 10.5 Checking for the End of a Text File with `hasNextInt`

```
        int next, sum = 0;
        while (inputStream.hasNextInt( ))
        {
            next = inputStream.nextInt( );
            sum = sum + next;
        }

        inputStream.close( );

        System.out.println("The sum of the numbers is " + sum);
    }
}
```

```
File data.txt

1  2
3  4 hi 5
```

Reading ends when either the end of the file is reach or a token that is not an **`int`** *is reached. So, the* **5** *is never read.*

SCREEN OUTPUT

```
The sum of the numbers is 10
```

Methods in the Class **`Scanner`** (Part 1 of 11)

Display 10.6 Methods in the Class Scanner

`Scanner` is in the `java.util` package.

```
public Scanner(InputStream streamObject)
```

There is no constructor that accepts a file name as an argument. If you want to create a stream using a file name, you can use

```
new Scanner(new FileInputStream(File_Name))
```

When used in this way, the `FileInputStream` constructor, and thus the `Scanner` constructor invocation, can throw a `FileNotFoundException`, which is a kind of `IOException`.
To create a stream connected to the keyboard, use
`new Scanner(System.in)`

(continued)

Methods in the Class **`Scanner`** (Part 2 of 11)

Display 10.6 Methods in the Class Scanner

```
public Scanner(File fileObject)
```

The `File` class will be covered in the section entitled "The `File` Class," later in this chapter. We discuss it here so that you will have a more complete reference in this display, but you can ignore this entry until after you've read that section.
If you want to create a stream using a file name, you can use

```
new Scanner(new File(File_Name))
```

```
public int nextInt()
```

Returns the next token as an `int`, provided the next token is a well-formed string representation of an `int`.

Throws a `NoSuchElementException` if there are no more tokens.

Throws an `InputMismatchException` if the next token is not a well-formed string representation of an `int`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

(continued)

Methods in the Class **`Scanner`** (Part 3 of 11)

Display 10.6 Methods in the Class Scanner

```
public boolean hasNextInt()
```

Returns `true` if the next token is a well-formed string representation of an `int`; otherwise returns `false`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

```
public long nextLong()
```

Returns the next token as a `long`, provided the next token is a well-formed string representation of a `long`.

Throws a `NoSuchElementException` if there are no more tokens.

Throws an `InputMismatchException` if the next token is not a well-formed string representation of a `long`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

(continued)

Methods in the Class **`Scanner`** (Part 4 of 11)

Display 10.6 Methods in the Class Scanner

```
public boolean hasNextLong()
```

Returns `true` if the next token is a well-formed string representation of a `long`; otherwise returns `false`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

```
public byte nextByte()
```

Returns the next token as a `byte`, provided the next token is a well-formed string representation of a `byte`.

Throws a `NoSuchElementException` if there are no more tokens.

Throws an `InputMismatchException` if the next token is not a well-formed string representation of a `byte`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

(continued)

Methods in the Class **`Scanner`** (Part 5 of 11)

Display 10.6 Methods in the Class Scanner

```
public boolean hasNextByte()
```

Returns `true` if the next token is a well-formed string representation of a `byte`; otherwise returns `false`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

```
public short nextShort()
```

Returns the next token as a `short`, provided the next token is a well-formed string representation of a `short`.

Throws a `NoSuchElementException` if there are no more tokens.

Throws an `InputMismatchException` if the next token is not a well-formed string representation of a `short`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

(continued)

Methods in the Class **`Scanner`** (Part 6 of 11)

Display 10.6 Methods in the Class Scanner

```
public boolean hasNextShort()
```

Returns `true` if the next token is a well-formed string representation of a `short`; otherwise returns `false`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

```
public double nextDouble()
```

Returns the next token as a `double`, provided the next token is a well-formed string representation of a `double`.

Throws a `NoSuchElementException` if there are no more tokens.

Throws an `InputMismatchException` if the next token is not a well-formed string representation of a `double`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

(continued)

Methods in the Class **`Scanner`** (Part 7 of 11)

Display 10.6 Methods in the Class Scanner

```
public boolean hasNextDouble()
```

Returns `true` if the next token is a well-formed string representation of an `double`; otherwise returns `false`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

```
public float nextFloat()
```

Returns the next token as a `float`, provided the next token is a well-formed string representation of a `float`.

Throws a `NoSuchElementException` if there are no more tokens.

Throws an `InputMismatchException` if the next token is not a well-formed string representation of a `float`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

(continued)

Methods in the Class **`Scanner`** (Part 8 of 11)

Display 10.6 Methods in the Class Scanner

```
public boolean hasNextFloat()
```

Returns `true` if the next token is a well-formed string representation of an `float`; otherwise returns `false`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

```
public String next()
```

Returns the `next` token.

Throws a `NoSuchElementException` if there are no more tokens.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

(continued)

Methods in the Class **Scanner** (Part 9 of 11)

Display 10.6 Methods in the Class Scanner

```
public boolean hasNext()
```

Returns `true` if there is another token. May wait for a next token to enter the stream.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

```
public boolean nextBoolean()
```

Returns the next token as a `boolean` value, provided the next token is a well-formed string representation of a `boolean`.

Throws a `NoSuchElementException` if there are no more tokens.

Throws an `InputMismatchException` if the next token is not a well-formed string representation of a `boolean` value.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

(continued)

Methods in the Class **`Scanner`** (Part 10 of 11)

Display 10.6 Methods in the Class Scanner

```
public boolean hasNextBoolean()
```

Returns `true` if the next token is a well-formed string representation of a `boolean` value; otherwise returns `false`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

```
public String nextLine()
```

Returns the rest of the current input line. Note that the line terminator `'\n'` is read and discarded; it is not included in the string returned.

Throws a `NoSuchElementException` if there are no more lines.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

(continued)

Methods in the Class **`Scanner`** (Part 11 of 11)

Display 10.6 Methods in the Class Scanner

```
public boolean hasNextLine()
```

Returns `true` if there is a next line. May wait for a next line to enter the stream.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

```
public Scanner useDelimiter(String newDelimiter);
```

Changes the delimiter for input so that `newDelimiter` will be the only delimiter that separates words or numbers. See the subsection "Other Input Delimiters" in Chapter 2 for the details. (You can use this method to set the delimiters to a more complex pattern than just a single string, but we are not covering that.)

Returns the calling object, but we have always used it as a `void` method.

Just Checking

In Java, when you open a text file you should account for a possible:

A. FileNotFoundException
B. FileFullException
C. FileNotReadyException
D. all of the above
E. None of the above

Just Checking

The scanner class has a series of methods that checks to see if there is any more well-formed input of the appropriate type. These methods are called _____ methods:

A. nextToken
B. hasNext
C. getNext
D. testNext
E. None of the above

Class Hierarchy for Package `java.io`

- `java.lang.Object`
  - `java.io.File`
  - `java.io.InputStream`
    - `java.io.FileInputStream`
    - `java.io.ObjectInputStream`
    - `java.io.FilterInputStream`
    - …
  - `java.io.OutputStream`
    - `java.io.FileOutputStream`
    - `java.io.ObjectOutputStream`
    - `java.io.FilterOutputStream`
    - …
  - `java.io.Reader`
    - `java.io.BufferedReader`
    - `java.io.InputStreamReader`
      - `java.io.FileReader`
    - …
  - `java.io.Writer`
    - `java.io.BufferedWriter`
    - `java.io.OutputStreamWriter`
      - `java.io.FileWriter`
    - `java.io.PrintWriter`
    - …

Random Access Files

- `java.io.RandomAccessFile`

Reading a Text File: BufferedReader

Before Java 5, to read a text file we used the class **BufferedReader**

- only 2 methods to read input: **read()** and **readLine()**
- **read()**:
  - reads a single character,
  - returns the next character as an **int** or -1 at EOF

```
BufferedReader in = new BufferedReader(
                new FileReader("in.txt"));

char c;
int next = in.read();
while (next != -1) {
   c = (char) next;
   next = in.read(); }
```

BufferedReadTest.java

Reading a Text File: `BufferedReader`

- `readLine()`
  - reads a line of character,
    - and returns a String
    - returns `null` when it tries to read beyond the EOF
- Unlike `Scanner`, `BufferedReader` has no methods to read a number from a text file
    - So … you read a string, then converted it to a number

```
BufferedReader in = new BufferedReader(
                          new FileReader("input.txt"));
…
String line = in.readLine();
int age = Integer.parseInt(line);
```

Some Methods of the Class **BufferedReader** (Part 1 of 2)

Display 10.8 Some Methods of the Class BufferedReader

BufferedReader and FileReader are in the java.io package.

```
public BufferedReader(Reader readerObject)
```

This is the only constructor you are likely to need. There is no constructor that accepts a file name as an argument. If you want to create a stream using a file name, you use

```
new BufferedReader(new FileReader(File_Name))
```

When used in this way, the FileReader constructor, and thus the BufferedReader constructor invocation, can throw a FileNotFoundException, which is a kind of IOException.
The File class will be covered in the section entitled "The File Class." We discuss it here so that you will have a more complete reference in this display, but you can ignore the following reference to the class File until after you've read that section.
If you want to create a stream using an object of the class File, you use

```
new BufferedReader(new FileReader(File_Object))
```

When used in this way, the FileReader constructor, and thus the BufferedReader constructor invocation, can throw a FileNotFoundException, which is a kind of IOException.

(continued)

Some Methods of the Class **`BufferedReader`** (Part 2 of 2)

Display 10.8 Some Methods of the Class `BufferedReader`

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

Reads a line of input from the input stream and returns that line. If the read goes beyond the end of the file, `null` is returned. (Note that an `EOFException` is not thrown at the end of a file. The end of a file is signaled by returning `null`.)

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

Reads a single character from the input stream and returns that character as an `int` value. If the read goes beyond the end of the file, then –1 is returned. Note that the value is returned as an `int`. To obtain a `char`, you must perform a type cast on the value returned. The end of a file is signaled by returning –1. (All of the "real" characters return a positive integer.)

```
public long skip(long n) throws IOException
```

Skips n characters.

```
public void close() throws IOException
```

Closes the stream's connection to a file.

The `IOException` Class

- Operations performed by the I/O classes may throw an **`IOException`**
  - A file intended for reading or writing might not exist or you don't have permission
  - Even if the file exists, a program may not be able to find it
  - The file might not contain the kind of data we expect
  - You were writing a file and disk space is not available anymore
  - Many more …
- An **`IOException`** is a checked exception

Path Names

- When a file name is used as an argument to a constructor for opening a file, it is assumed that the file is in the same directory or folder as the one in which the program is run
- If it is not in the same directory, the full or relative path name must be given
- A *path name* not only gives the name of the file, but also the directory or folder in which the file exists
- A *full path name* gives a complete path name, starting from the root directory
- A *relative path name* gives the path to the file, starting with the directory in which the program is located

Path Names

- The way path names are specified depends on the operating system
  - A typical UNIX path name that could be used as a file name argument is

```
"/user/sallyz/data/data.txt"
```

  - A **`BufferedReader`** input stream connected to this file is created as follows:

```
BufferedReader inputStream =
   new BufferedReader(new
   FileReader("/user/sallyz/data/data.txt"));
```

Class Hierarchy for Package `java.io`

- `java.lang.Object`
  - `java.io.File` ←

binary input streams

- `java.io.InputStream`
  - `java.io.FileInputStream`
  - `java.io.ObjectInputStream`
  - `java.io.FilterInputStream`
  - …

binary output streams

- `java.io.OutputStream`
  - `java.io.FileOutputStream`
  - `java.io.ObjectOutputStream`
  - `java.io.FilterOutputStream`
  - …

text input streams

- `java.io.Reader`
  - `java.io.BufferedReader`
  - `java.io.InputStreamReader`
    - `java.io.FileReader`
  - …

text output streams

- `java.io.Writer`
  - `java.io.BufferedWriter`
  - `java.io.OutputStreamWriter`
    - `java.io.FileWriter`
  - `java.io.PrintWriter`
  - …

Random Access Files

- `java.io.RandomAccessFile`

The `File` Class

- not really an I/O stream
- contains methods to check the properties of a file
  - ex: check if a file with a specific name exists, if a file can be written into, …
- constructor takes a filename (or directory name or URL) as argument
- useful methods:

```
boolean canRead()            boolean canWrite()
boolean createNewFile()      boolean delete()
boolean exists()             boolean isDirectory()
File[] listFiles()           boolean mkdir()
boolean renameTo(File dest) …
```

Example: `FileInfo.java`

Some Methods in the Class `File` (Part 1 of 5)

Display 10.12 Some Methods in the Class `File`

`File` is in the `java.io` package.

```
public File(String File_Name)
```

Constructor. *File_Name* can be either a full or a relative path name (which includes the case of a simple file name). *File_Name* is referred to as the **abstract path name**.

```
public boolean exists()
```

Tests whether there is a file with the abstract path name.

```
public boolean canRead()
```

Tests whether the program can read from the file. Returns `true` if the file named by the abstract path name exists and is readable by the program; otherwise returns `false`.

(continued)

Some Methods in the Class `File` (Part 2 of 5)

Display 10.12 Some Methods in the Class `File`

```
public boolean setReadOnly()
```

Sets the file represented by the abstract path name to be read only. Returns `true` if successful; otherwise returns `false`.

```
public boolean canWrite()
```

Tests whether the program can write to the file. Returns `true` if the file named by the abstract path name exists and is writable by the program; otherwise returns `false`.

```
public boolean delete()
```

Tries to delete the file or directory named by the abstract path name. A directory must be empty to be removed. Returns `true` if it was able to delete the file or directory. Returns `false` if it was unable to delete the file or directory.

(continued)

Some Methods in the Class `File` (Part 3 of 5)

Display 10.12 Some Methods in the Class `File`

```
public boolean createNewFile() throws IOException
```

Creates a new empty file named by the abstract path name, provided that a file of that name does not already exist. Returns `true` if successful, and returns `false` otherwise.

```
public String getName()
```

Returns the last name in the abstract path name (that is, the simple file name). Returns the empty string if the abstract path name is the empty string.

```
public String getPath()
```

Returns the abstract path name as a `String` value.

```
public boolean renameTo(File New_Name)
```

Renames the file represented by the abstract path name to *New_Name*. Returns `true` if successful; otherwise returns `false`. *New_Name* can be a relative or absolute path name. This may require moving the file. Whether or not the file can be moved is system dependent.

(continued)

Some Methods in the Class `File` (Part 4 of 5)

Display 10.12 Some Methods in the Class `File`

```
public boolean isFile()
```

Returns `true` if a file exists that is named by the abstract path name and the file is a normal file; otherwise returns `false`. The meaning of *normal* is system dependent. Any file created by a Java program is guaranteed to be normal.

```
public boolean isDirectory()
```

Returns `true` if a directory (folder) exists that is named by the abstract path name; otherwise returns `false`.

(continued)

Some Methods in the Class `File` (Part 5 of 5)

Display 10.12 Some Methods in the Class `File`

```
public boolean mkdir()
```

Makes a directory named by the abstract path name. Will not create parent directories. See `mkdirs`. Returns `true` if successful; otherwise returns `false`.

```
public boolean mkdirs()
```

Makes a directory named by the abstract path name. Will create any necessary but nonexistent parent directories. Returns `true` if successful; otherwise returns `false`. Note that if it fails, then some of the parent directories may have been created.

```
public long length()
```

Returns the length in bytes of the file named by the abstract path name. If the file does not exist or the abstract path name names a directory, then the value returned is not specified and may be anything.

Class Hierarchy for Package `java.io`

- `java.lang.Object`
  - `java.io.File`

binary input streams

- `java.io.InputStream`
  - `java.io.FileInputStream`
  - `java.io.ObjectInputStream`
  - `java.io.FilterInputStream`
  - …

binary output streams

- `java.io.OutputStream`
  - `java.io.FileOutputStream`
  - `java.io.ObjectOutputStream`
  - `java.io.FilterOutputStream`
  - …

text input streams

- `java.io.Reader`
  - `java.io.BufferedReader`
  - `java.io.InputStreamReader`
    - `java.io.FileReader`
  - …

text output streams

- `java.io.Writer`
  - `java.io.BufferedWriter`
  - `java.io.OutputStreamWriter`
    - `java.io.FileWriter`
  - `java.io.PrintWriter`
  - …

Random Access Files

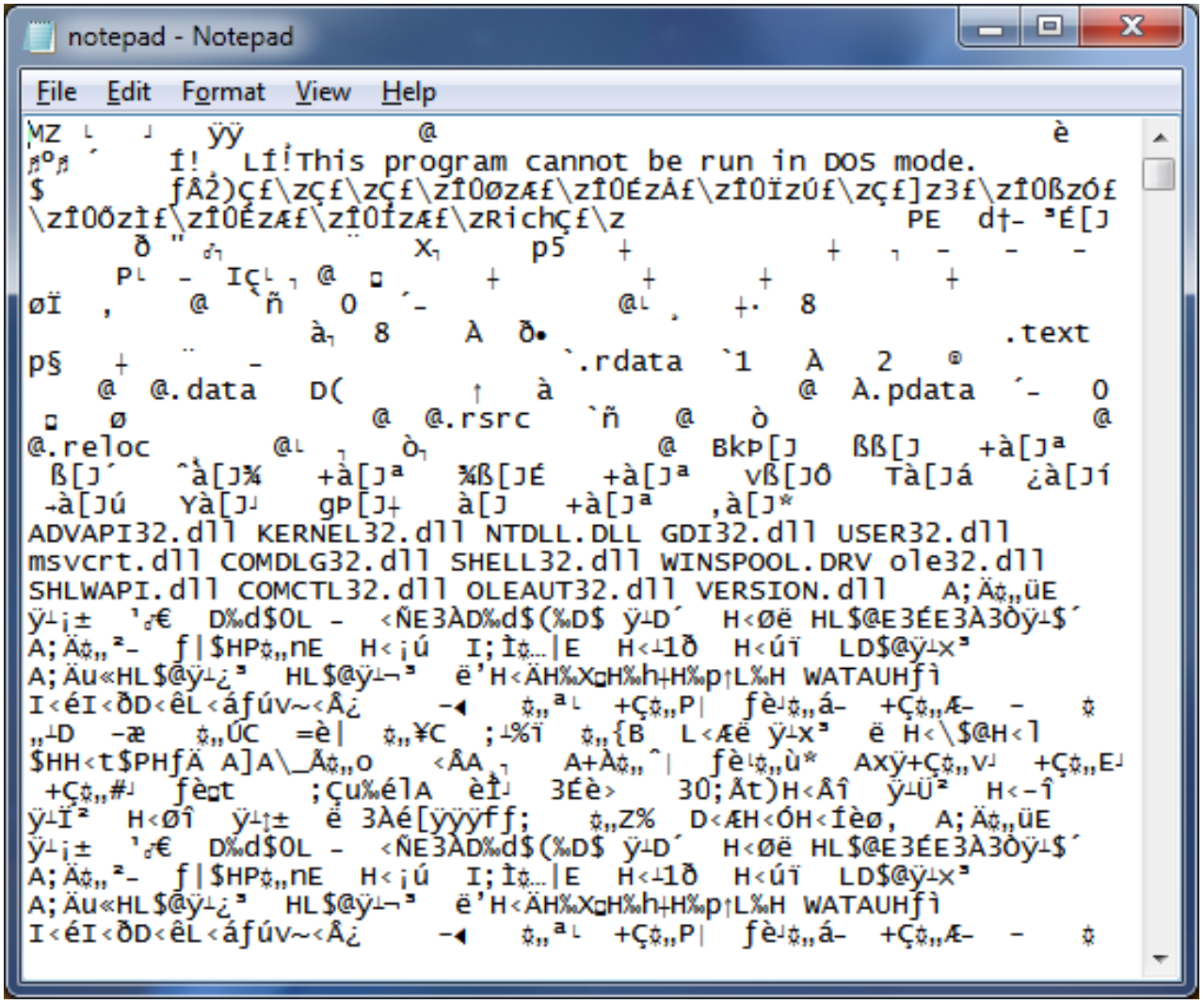
- `java.io.RandomAccessFile`

Binary Files

- Binary files store data in the same format used by computer memory to store the values of variables
  - No conversion needs to be performed when a value is stored or retrieved from a binary file

- Java binary files, unlike other binary language files, are portable
  - A binary file created by a Java program can be moved from one computer to another
  - These files can then be read by a Java program, but only by a Java program

Binary Files



Reading/Writing Binary Files

- Use the **`ObjectInputStream`** and **`ObjectOutputStream`** classes

- Can read/write strings, int, floats, Objects, etc. directly to binary files
  - to read: **`readInt()`**, **`readDouble()`**, **`readChar()`**, **`readBoolean()`** and **`readUTF()`** (to read strings)
  - to write: **`writeInt()`**, **`writeDouble()`**, **`writeChar()`**, **`writeBoolean()`** and **`writeUTF()`** (to write strings)

Reading/Writing Binary Files

- The method **`writeUTF()`** can be used to output values of type String, and **`readUTF()`** to read Strings

- UTF is an encoding scheme used to encode Unicode characters that favors the ASCII character set

Examples

```
ObjectOutputStream out =
            new ObjectOutputStream(
                new FileOutputStream("myFile.dat"));
out.writeInt(…);
out.writeDouble(…);
out.writeChar(…);
…
```

- If the file contains multiple types, each item type must be read in exactly the sa

```
ObjectInputStream in =
            new ObjectInputStream(
                new FileInputStream("myFile.dat"));
int x = in.readInt();
double y = in.readDouble();
String s = in.readUTF();
…
```

Example: BinaryOutputDemo.java

```
  ObjectOutputStream outputStream = null;

  try {
    outputStream = new ObjectOutputStream(new
  FileOutputStream("numbers2.dat"));
    int n;
     …
     do {
        n = keyboard.nextInt();
        outputStream.writeInt(n);
     } while (n >= 0);
  }
  catch(IOException e) {
     System.out.println("Problem with output to file numbers2.dat.");
}
  finally {
    try {  if (outputStream != null)
      outputStream.close( );    }
    catch(IOException e) {
       System.out.println("Can't seem to close the file...");}

  }
```

Example:

BinaryInputDemo.java

Checking for the End of a Binary File the Correct Way

- All of the **`ObjectInputStream`** methods that read from a binary file throw an **`EOFException`** when trying to read beyond the end of a file
  - This can be used to end a loop that reads all the data in a file
- Note that different file-reading methods check for the end of a file in different ways
  - Testing for the end of a file in the wrong way can cause a program to go into an infinite loop or terminate abnormally

Object Serializaton

- *“Like the Transporter on Star Trek, it's all about taking something complicated and turning it into a flat sequence of 1s and 0s, then taking that sequence of 1s and 0s (possibly at another place, possibly at another time) and reconstructing the original complicated "something."*

Object Serialization

- Java allows you to write the object's current state to disk
- Just values of instance variables, not methods
- Also called *object streams*
- Once serialized, the objects can be read again into another program
- Huge advantage over writing the object in text format by hand…
  - you don't have to break up the object yourself into numbers, strings, imbedded objects, … to read and write it
  - you just read/write an entire object (even an entire array of objects) in one shot

Object Serialization

- The idea that an object can "live" beyond the program execution that created it is called *persistence*
- Once serialized, the objects can be read again into another program
- Objects are saved in binary format,
  - so you use the **`ObjectInputStream`** / **`ObjectOutputStream`** classes
  - not the **`Reader`** / **`Writer`** classes

Object Serialization

To serialize an object:

1. It must implement the **`Serializable`** interface
   - more on interfaces later…
2. Use the **`ObjectOutputStream`** and **`ObjectInputStream`** classes
3. Use the methods:
   - **`writeObject()`** to serialize an object
   - **`readObject()`** to deserialize an object
     - you need to use a cast to convert the object to the right type

Example

To serialize:

```
BankAccount b = …;
ObjectOutputStream out = new ObjectOutputStream(
                            new FileOutputStream("bank.dat"));
out.writeObject(b);
```

To deserialize :

```
ObjectInputStream in = new ObjectInputStream(
                          new FileInputStream("bank.dat"));
BankAccount b = (BankAccount) in.readObject();
```

`readObject()` returns an **`Object`** reference, so you need to cast the result

`readObject()` can throw a (checked) **`ClassNotFoundException`** so you need to catch it or declare it

Object Serialization

- If the class has instance variables that are references to classes (composition)
  - these objects will be serialized too
  - so they must also implement the **`Serializable`** interface
- Many classes from the Java class library implement **`Serializable`**, including the **`String`** class
- Why don't all classes implement **`Serializable`** ????
- An entire array can be serialized (written/read in one operation)

Example: Bank Accounts

- Serialization of a *Bank* object (array of *BankAccount* objects)
- If a file with serialized data exists (in **`bank.dat`**), then it is loaded
  - Otherwise the program starts with a new *Bank* object.
- *BankAccount* objects are added to *Bank*.
- Then the *Bank* object is saved.

Example: Serialtester.java

```
import java.io.*;

public class SerialTester {
   public static void main(String[] args) throws IOException,
                                                 ClassNotFoundException
   {
      Bank myBank;
      File f = new File("bank.dat");
      if (f.exists())  {
        ObjectInputStream in = new ObjectInputStream(
                 new FileInputStream(f));
        myBank = (Bank) in.readObject();
        in.close();
      }
      else {
        myBank = new Bank();
        myBank.addAccount(new BankAccount(1001, 20000));
        myBank.addAccount(new BankAccount(1015, 10000));
      }
```

Example: Serialtester.java

```
 // Deposit some money
  BankAccount a = myBank.find(1001);
  a.deposit(100);
  System.out.println(a.getAccountNumber() + ":" + a.getBalance());
  a = myBank.find(1015);
  System.out.println(a.getAccountNumber() + ":" + a.getBalance());
  ObjectOutputStream out = new ObjectOutputStream(
                 new FileOutputStream(f));

  out.writeObject(myBank);

  out.close();
  }
}
```

- **See also:** Bank.java

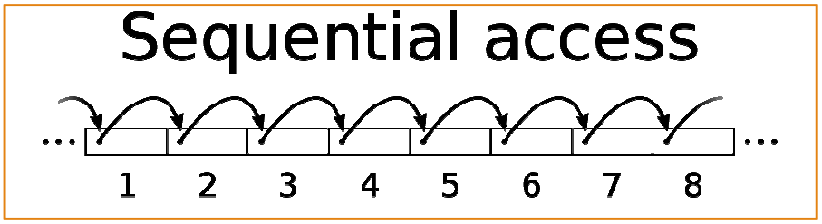
Example: Output

First Program Run

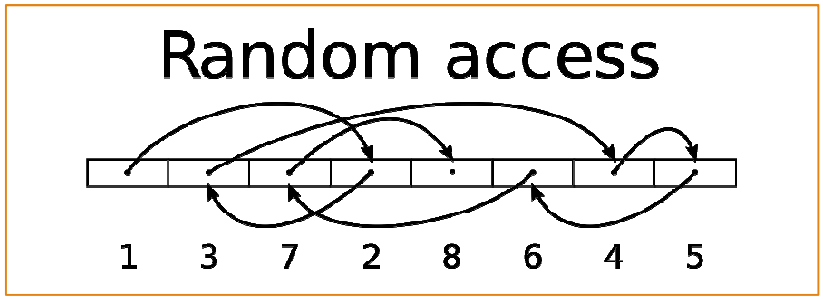
Second Program Run

Random Access vs. Sequential Access

- Sequential access
  - A file is processed one byte at a time
  - It can be inefficient

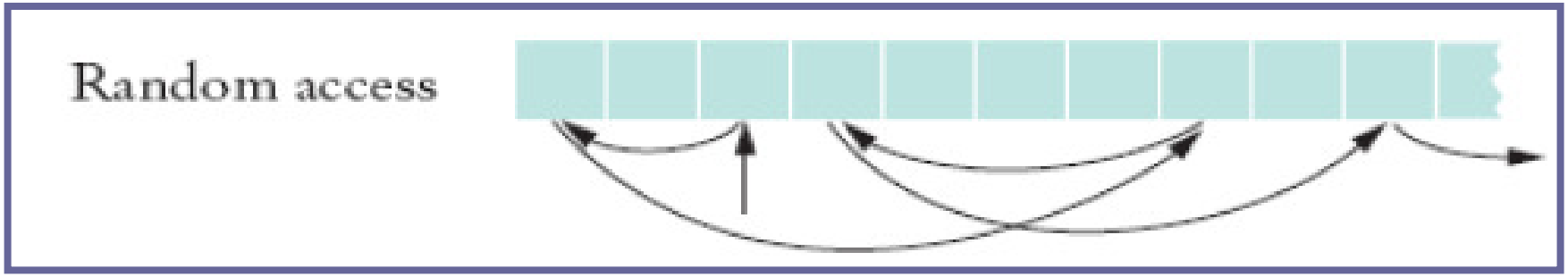


- Random access
  - Allows access at arbitrary locations in the file
  - Only binary disk files support random access
    - **`System.in`** and **`System.out`** do not



Random Access to binary files

- Each file has a special **file pointer** position
  - You can read or write at the position where the pointer is



Class Hierarchy for Package `java.io`

- `java.lang.Object`
  - `java.io.File`

binary input streams

- `java.io.InputStream`
  - `java.io.FileInputStream`
  - `java.io.ObjectInputStream`
  - `java.io.FilterInputStream`
  - …

binary output streams

- `java.io.OutputStream`
  - `java.io.FileOutputStream`
  - `java.io.ObjectOutputStream`
  - `java.io.FilterOutputStream`
  - …

text input streams

- `java.io.Reader`
  - `java.io.BufferedReader`
  - `java.io.InputStreamReader`
    - `java.io.FileReader`
  - …

text output streams

- `java.io.Writer`
  - `java.io.BufferedWriter`
  - `java.io.OutputStreamWriter`
    - `java.io.FileWriter`
  - `java.io.PrintWriter`
  - …

***Random** Access Files*

- `java.io.RandomAccessFile`

`RandomAccessFile` Class

- You can open a file either for
  - Reading only ("**`r`**")
  - Reading and writing ("**`rw`**")

```
RandomAccessFile f = new RandomAcessFile("bank.dat","rw");
```

- Writes binary data
- Read/write primitives
  - `writeDouble()` `readDouble()`
  - `writeInt()` `readInt()`
  - `writeChar()` `readChar()`
  - …

`RandomAccessFile` Class

To move the file pointer to a specific byte

```
f.seek(n);
```

Need to know size of primitives

Other languages (C) requires a `sizeof()` operator

Java uses standardized sizes

- `Byte.SIZE = 1;`
- `Character.SIZE = 2;`
- `Short.SIZE = 2;`
- `Integer.SIZE = 4;`
- `Long.SIZE = 8;`
- `Float.SIZE = 4;`
- `Double.SIZE = 8;`

Can use these values to calculate offsets into the file

Example

Save a database in a file:

- ex: age - gender - salary

```
100 43 'M' 19.55
110 83 'F' 85.60
120 25 'M' 143.45
130 37 'M' 29.99
140 11 'F' 5.50
```

access a specific record to change it

Example: RandomAccess.java

```
import java.io.*;
class EmployeeRecord {

    static final int RECORD_SIZE = Integer.SIZE + Byte.SIZE +
                   Character.SIZE + Double.SIZE;
    static final int ID_OFFSET = 0;
    static final int AGE_OFFSET = Integer.SIZE;
    static final int GENDER_OFFSET = Integer.SIZE + Byte.SIZE;
    static final int SALARY_OFFSET = Integer.SIZE + Byte.SIZE +
   Character.SIZE;

   static int currentID;
   final int ID;
   byte age;
   char gender;
   double salary;

   public EmployeeRecord(int ID, byte age, char gender, double salary){
        …
   }
…
}
```

Example: RandomAccess.java

```
public class RandomAccess {
   static int DBSIZE = 10;

   public static void main ( String[] aArguments ) throws
   IOException {
        String dbFile = "db.dat";
        EmployeeRecord[] dataBase = buildDataBase();

        // store the database to disk
        RandomAccessFile rf = new RandomAccessFile(dbFile, "rw");
        for(int i = 0; i < DBSIZE; i++){
           rf.writeInt(dataBase[i].getID());
           rf.writeByte(dataBase[i].getAge());
           rf.writeChar(dataBase[i].getGender());
           rf.writeDouble(dataBase[i].getSalary());
        }
        rf.close();
```

Example: RandomAccess.java

```
// update the database and store to disk
rf =  new RandomAccessFile(dbFile, "rw");
System.out.println("\nChanging Salary of employee #3 to 88.88...");
rf.seek((3 * EmployeeRecord.RECORD_SIZE) +
            EmployeeRecord.SALARY_OFFSET);
rf.writeDouble(88.88);
rf.close();
```

Example: RandomAccess.java

```
        // read the new database into an employee array
        rf = new RandomAccessFile(dbFile, "r");
        for (int i = 0; i < DBSIZE; i++){
           EmployeeRecord emp = new EmployeeRecord(rf.readInt(),
                                                   rf.readByte(),
                                                   rf.readChar(),
                                                   rf.readDouble());
           dataBase[i] = emp;
        }
        rf.close();
    }
```

Example: RandomAccess.java

```
   static EmployeeRecord[] buildDataBase(){
        EmployeeRecord[] dataBase = new EmployeeRecord[DBSIZE];

        dataBase[0] = new EmployeeRecord(100, (byte)43, 'M', 19.55);
        dataBase[1] = new EmployeeRecord(110, (byte)83, 'F', 85.60);
        …
        dataBase[9] = new EmployeeRecord(190, (byte)16, 'M', 13.56);

        return dataBase;
   }
}
```