

Text and Binary Files

Michael C. Hackett
Computer Science Department

Community
College
of Philadelphia

Lecture Topics

- File Objects
- Text Files
 - Reading text files
 - Writing text files
 - Appending data
- Binary Files
 - Writing binary files
 - Reading binary files
- Random Access Files
 - Writing random access files
 - Reading random access files
- Object Serialization

Colors/Fonts

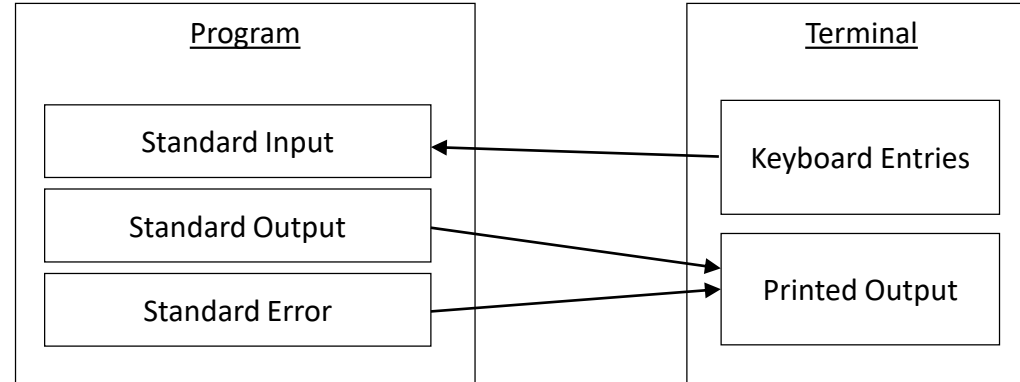
• Local Variable Names	—	Brown
• Primitive data types	—	
• Literals	—	Blue
• Keywords	—	Orange
• Object names	—	Green
• Operators/Punctuation	—	Black
• Field Names	—	Lt Blue
• Method Names	—	Purple
• Parameter Names	—	Gold
• Comments	—	Gray
• Package Names	—	Pink

Source Code	— Consolas
Output	— Courier New

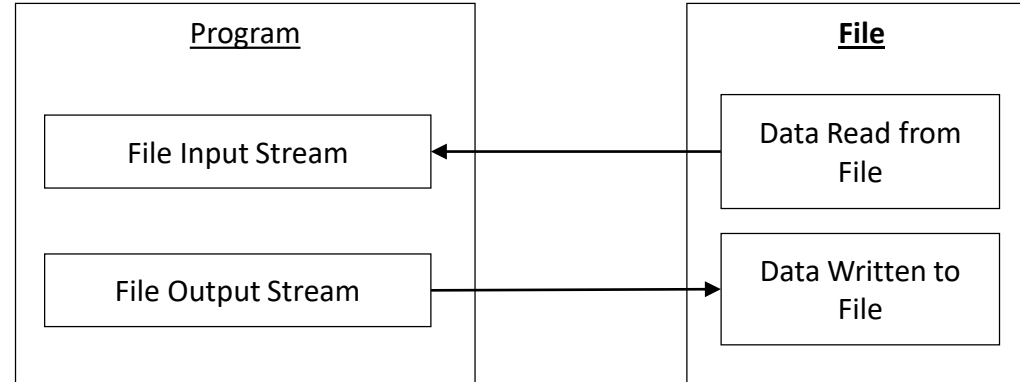
What are files?

- A ***file*** is stream of binary, digital information typically kept on a long-term storage device.
 - Word documents, Powerpoint presentations, and PDFs are all examples of different types of files.
- Can be used as an input data stream. (“Reading a file”)
- Can be used as an output data stream. (“Writing to a file”)

Standard Data Streams



File Data Streams



Extensions

- A file has a name which normally includes an extension.
 - Textfile.**txt**
 - WordDocument.**docx**
- You can have files without extensions.
 - Extensions are primarily used by the operating system, so it knows what program to use to open and read the file.
 - Some programs will only accept files with certain extensions.

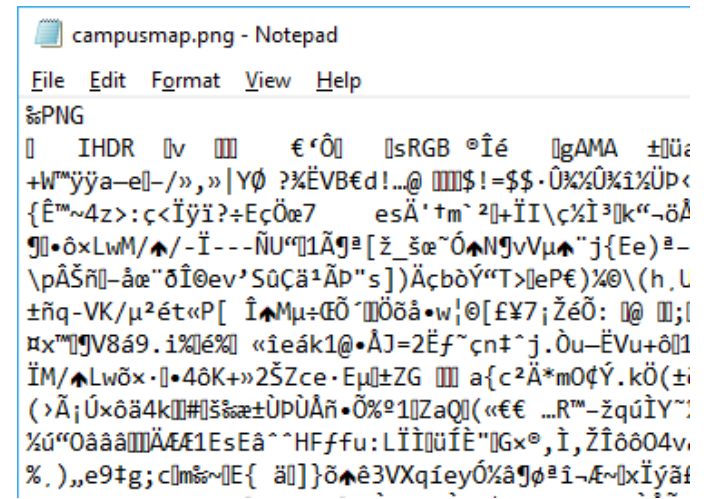
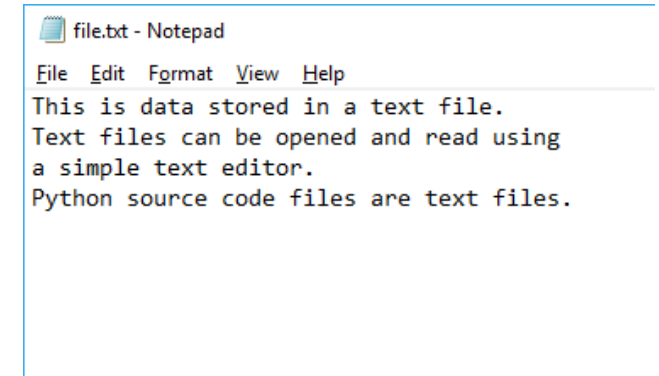
Types of Files

- Text Files

- The binary information contained in the file is encoded with ASCII plaintext.
- Can be opened in any text editor (like Notepad.)
- “Human readable”

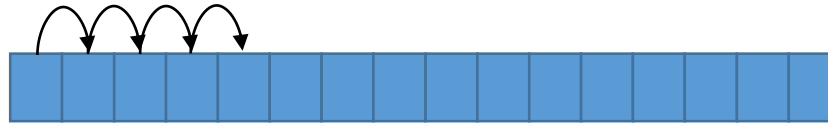
- Data/Binary Files

- Files that are not encoded in plaintext, like images and compiled programs.
- Normally cannot be opened in a program like a text editor.
- Raw binary- “Computer readable”

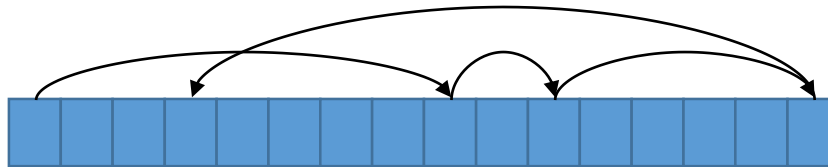


File Access

- Using ***sequential access***, data is read/accessed from the beginning of the file through the end of the file.



- Using ***direct/random access***, data can be accessed from any location in the file.



File objects


- The File object provides a number of methods that gives us information about a file.
- It must be imported.

```
import java.io.File;
```

File objects

- There are a number of constructors for a File object, but we'll be using the constructor with one String parameter- the path to the file.

```
File myTextFile = new File("C:\\path\\to\\my\\file.txt");
```



\ indicates the start of an escape sequence.
Need to use \\ in Strings containing a file's path.

File objects

- MAC AND LINUX USERS:

```
File myTextFile = new File("/path/to/my/file.txt");
```



Use forward slashes.

Determining if a file exists

- To check if a file exists before you try to read or write to it, use the exists method.
 - This prevents FileNotFoundExceptions when trying to read a file that doesn't exist.
 - You can check to see if a file exists because you might not want to overwrite an existing file.

```
File myTextFile = new File("C:\\path\\to\\my\\file.txt");  
if(myTextFile.exists()) {  
  
}
```

- The exists() method returns true or false.

Determining if a file is writable

- To check if permissions allow you to change/overwrite it, use the `canWrite` method.
 - This prevents `IOExceptions` when trying to change a file that you don't have permission to modify.

```
File myTextFile = new File("C:\\path\\to\\my\\file.txt");  
if(myTextFile.canWrite()) {  
  
}
```

- The `canWrite()` method returns `true` or `false`.

Determining if a file is readable

- To check if permissions allow you to read the file, use the `canRead` method.
 - This prevents `IOExceptions` for trying to read a file that you don't have permission to read.

```
File myTextFile = new File("C:\\path\\to\\my\\file.txt");  
if(myTextFile.canRead()) {  
  
}
```

- The `canRead()` method returns true or false.

Determining the file's name

- The File object keeps the entire path of the file.
 - To get the file's name, use the getName method.

```
File myTextFile = new File("C:\\path\\to\\my\\file.txt");  
System.out.println(myTextFile.getName());
```

```
file.txt
```

- The getName() method returns a String- the file name.

Determining the file's path

- To get the directory path of the file, use the getParent method.

```
File myTextFile = new File("C:\\path\\to\\my\\file.txt");  
System.out.println(myTextFile.getParent());
```

```
C:\\path\\to\\my
```

- The getParent() method returns a String- the file's directory path.

Determining the file's full path

- To get the full file path, use the `getPath()` method.

```
File myTextFile = new File("C:\\path\\to\\my\\file.txt");  
System.out.println(myTextFile.getPath());
```

```
C:\path\to\my\file.txt
```

- The `getPath()` method returns a `String`- the full path, including name.

Determining the file's full path

- If the file is in the same directory as the compiled class file, you don't need to specify the file's path for Java to find it.
 - If the text file is in the top level of your IntelliJ project folder, you don't have to specify the path.

```
File myTextFile = new File("file.txt");  
System.out.println(myTextFile.getPath());
```

file.txt

- However, the file's path will just be the file name.

Determining the file's absolute path

- The File object's `getAbsolutePath` method will always return the full path to the file, even if we only gave the File object the file's name or a limited path.

```
File myTextFile = new File("file.txt");  
System.out.println(myTextFile.getAbsolutePath());
```

```
C:\path\to\my\file.txt
```

Reading text files in Java using a Scanner

- Pass your file to a new Scanner object.
 - This is the same Scanner object you have been using to get keyboard input.
 - Now, we are using the file as the input stream instead of System.in

```
File myTextFile = new File("C:\\path\\to\\my\\file.txt");  
Scanner fileReader = new Scanner(myTextFile);
```

Exception Handling for Files

```
try {  
    File myTextFile = new File("C:\\path\\to\\my\\file.txt");  
    Scanner fileReader = new Scanner(myTextFile);  
    //Process contents of file...  
}  
catch(FileNotFoundException e) {  
    System.out.println("File not found");  
}
```

```
import java.io.FileNotFoundException;
```

Reading lines from a text file

- The `nextLine` method will return the next line of the file as a `String`.
 - In this example, it would return line 1 from `file.txt`, since this is the first time we called the `nextLine` method.
- This only reads a single line. How can we read through an entire file with an unknown number of lines?

```
File myTextFile = new File("C:\\path\\to\\my\\file.txt");  
Scanner fileReader = new Scanner(myTextFile);  
String line = fileReader.nextLine();
```

Reading lines from a text file

- The Scanner's hasNextLine method returns true if there are more lines to be read and false if it reached the end of the file.
- The below while loop will iterate as long as there are still lines to be read.

```
File myTextFile = new File("C:\\path\\to\\my\\file.txt");
Scanner fileReader = new Scanner(myTextFile);

while(fileReader.hasNextLine()) == true {
    System.out.println(fileReader.nextLine());
}
```


Closing the file

- The Scanner's close method releases it's hold on the file.
- Only call this method when you are done using the resource/file.

```
File myTextFile = new File("C:\\path\\to\\my\\file.txt");
Scanner fileReader = new Scanner(myTextFile);

while(fileReader.hasNextLine() == true) {
    System.out.println(fileReader.nextLine());
}

fileReader.close();
```

Reading text files in Java using a Scanner

```
Scanner fileReader = null;
try {
    File myTextFile = new File("C:\\path\\to\\my\\file.txt");
    fileReader = new Scanner(myTextFile);
    //Process contents of file...
}
catch(FileNotFoundException e) {
    System.out.println("File not found");
}
finally {
    try {
        fileReader.close();
    }
    catch(Exception e) {
        //File was never opened/fileReader is still null; Do nothing.
    }
}
```

Visible in try and finally

File would not be opened

Need to be ready for this to be null

Writing to files in Java

- The `PrintWriter` object provides an easy way to write data to a new or existing file.
- The `PrintWriter` object must be imported.

```
import java.io.PrintWriter;
```

Writing to text files in Java

```
File myTextFile = new File("C:\\path\\to\\my\\file.txt");  
PrintWriter fileWriter = new PrintWriter(myTextFile);
```

- The above example initializes a new PrintWriter object with a File object.
 - The file the PrintWriter will write to.
- Like the Scanner constructor, this PrintWriter constructor may also throw a FileNotFoundException.

Exception Handling for Files

```
try {  
    File myTextFile = new File("C:\\path\\to\\my\\file.txt");  
    PrintWriter fileWriter = new PrintWriter(myTextFile);  
    //Process contents of file...  
}  
catch(FileNotFoundException e) {  
    System.out.println("File not found");  
}
```

- If the file doesn't exist, the PrintWriter will create it.
 - You could get a FileNotFoundException because the path specified doesn't exist/was mistyped.
- If the file already exists, it will be completely overwritten.
 - There will be an example of how to append to files later in the lecture.

Writing to text files in Java

```
File myTextFile = new File("C:\\path\\to\\my\\file.txt");  
PrintWriter fileWriter = new PrintWriter(myTextFile);  
fileWriter.println("This will be printed to my file.");
```

- The `PrintWriter`'s **`println`**, **`print`**, and **`printf`** methods work in an identical fashion to `System.out.println/print/printf`.
 - The “out” in `System.out` is a `PrintWriter`. It just prints to the console/“standard output” instead of to a file.

Closing the file

```
File myTextFile = new File("C:\\path\\to\\my\\file.txt");  
PrintWriter fileWriter = new PrintWriter(myTextFile);  
fileWriter.println("This will be printed to my file.");  
fileWriter.print("This will also be printed to my file.");  
fileWriter.println("And so will this.");  
fileWriter.close();
```

- If you do not close your `PrintWriter` after you are finished writing to the file, your changes will **not** be made permanent and the file will be empty.

Writing text files in Java using a PrintWriter

```
PrintWriter fileWriter = null;
try {
    File myTextFile = new File("C:\\path\\to\\my\\file.txt");
    fileWriter = new PrintWriter(myTextFile);
    //Write contents to file...
}
catch(FileNotFoundException e) {
    System.out.println("File not found");
}
finally {
    try {
        fileWriter.close();
    }
    catch(Exception e) {
        //File was never opened/fileWriter is still null; Do nothing.
    }
}
```

Visible in try and finally

File would not be opened

Need to be ready for this to be null

Appending to files in Java

```
import java.io.BufferedWriter;  
import java.io.FileWriter;
```

```
PrintWriter fileWriter = new PrintWriter(new BufferedWriter(new FileWriter("OutputFile.txt", true)));  
fileWriter.println("This will be appended to the existing file.");  
fileWriter.close();
```

- A few more steps than using just PrintWriter and File objects.
- The above will append the line “This will be appended to the existing file.” to the end of OutputFile.txt instead of completely overwriting its existing data.
- This is just for your reference. Any labs/assignments will have you create new files/overwrite old ones.

Text Files

- Up until now, the files you worked with were text files: The data stored in those files were encoded in plaintext that could be opened with a simple text editor like Notepad.
- Scanners and PrintWriters are objects that can be used to reading and writing plaintext from/to text files.

```
File myTextFile = new File("sampleTextFile.txt");
PrintWriter fileWriter = new PrintWriter(myTextFile);
int x = 1297;
fileWriter.println(x);
fileWriter.close();
```

- The code above writes the contents of x to sampleTextFile.txt

Text Files

- However, the contents of x is written to the file as individual characters.

1297 expressed as characters

'1'	'2'	'9'	'7'
-----	-----	-----	-----

- Encoded in ASCII Plaintext (How it really is in the text file)

1297 expressed as 4 ASCII characters

00110001	00110010	00111001	00110111
----------	----------	----------	----------

(Reference: <http://www.asciitable.com/>)

Text Files

- In a computer's memory, numeric data like ints are not stored as literal values or encoded in plaintext.

1297 expressed as a 32-bit (unsigned) binary number

00000000	00000000	00000101	00010001
----------	----------	----------	----------

(Recall that ints are 32 bits/4 bytes in size.)

- *Raw Binary Data*

Binary Files

- Raw binary data can be stored in and written to a file.
 - Often called a **binary file** to distinguish it from ASCII plaintext files.
- Storing data in binary files is often more efficient than text files.
 - Imagine you have a file that contains a large amount of data.
 - If the data is stored in plaintext, the data will need to be converted from the individual ASCII characters to the correct form of raw binary data.

Binary Files

- Some types of data, like images, are always be stored in binary files.
- Unlike text files, binary files cannot be displayed correctly by simple text editors like Notepad.
 - Though, a program that understands the binary data of an image file (like Photoshop) can interpret and display that binary data correctly.

Binary Files

```
import java.io.FileOutputStream;  
import java.io.DataOutputStream;
```

- To write data to a binary file, use the following classes in conjunction with one another:
 - **FileOutputStream**: Allows you to open a file for writing binary data. It only provides basic functionality for writing bytes to the file.
 - **DataOutputStream**: Allows you to write the data of primitives and/or Strings to binary files. DataOutputStreams cannot directly access files, it must use a FileOutputStream.

```
FileOutputStream fileStream = new FileOutputStream("SomeFile.dat");  
DataOutputStream dataStream = new DataOutputStream(fileStream);
```

Binary data files typically end with .dat extension

Writing to Binary Files

- If you have no reason to reference the `FileOutputStream` outside of constructing the `DataOutputStream` object, you can combine the two statements we just saw into one:

```
FileOutputStream fileStream = new FileOutputStream("SomeFile.dat");  
DataOutputStream dataStream = new DataOutputStream(fileStream);
```



```
DataOutputStream dataStream = new DataOutputStream(new FileOutputStream("SomeFile.dat"));
```


Writing to Binary Files

- Some notes about `FileOutputStream`s:
 - The `FileOutputStream` object's constructor throws an `IOException` (a checked exception) if an error occurs when attempting to open the file.
 - The `FileOutputStream` is used for writing to a file. If the file already exists, the existing file is erased and a new file of the same name is created.
 - The `DataOutputStream` object handles passing the data correctly to the `FileOutputStream`

Common DataOutputStream Methods

Method	Return Type	Description	Possible Exceptions
writeBoolean(boolean)	void	Writes the boolean value to the file.	IOException
writeByte(byte)	void	Writes the byte value to the file.	IOException
writeChar(int)	void	It is assumed the int is a character code. The character it represents is written to the file as a two-byte Unicode character.	IOException
writeDouble(double)	void	Writes the double value to the file.	IOException
writeFloat(float)	void	Writes the float value to the file.	IOException
writeInt(int)	void	Writes the int value to the file.	IOException
writeLong(long)	void	Writes the long value to the file.	IOException
writeShort(short)	void	Writes the short value to the file.	IOException
writeUTF(String)	void	Writes the String to the file using Unicode Text Format.	IOException
close()	void/none	Closes the file.	IOException

Writing to Binary Files

```
DataOutputStream dataStream =  
    new DataOutputStream(new FileOutputStream("SomeFile.dat"));  
dataStream.writeInt(42);  
dataStream.writeDouble(98.6);  
dataStream.writeUTF("Hello World!");  
int x = 1297;  
dataStream.writeInt(x);  
dataStream.close()
```

Reading Binary Files

```
import java.io.FileInputStream;  
import java.io.DataInputStream;
```

- To read data from a binary file, use the following classes in conjunction with one another:
 - **FileInputStream**: Allows you to open a file for reading binary data. It only provides basic functionality for reading bytes from the file.
 - **DataInputStream**: Allows you to read the data of primitives and/or Strings from binary files. DataInputStreams cannot directly access files, it must use a FileInputStream.

```
FileInputStream fileStream = new FileInputStream("SomeFile.dat");  
DataInputStream dataStream = new DataInputStream(fileStream);
```

Reading Binary Files

- If you have no reason to reference the `FileInputStream` outside of constructing the `DataInputStream` object, you can combine the two statements we just saw into one:

```
FileInputStream fileStream = new FileInputStream("SomeFile.dat");  
DataInputStream dataStream = new DataInputStream(fileStream);
```



```
DataInputStream dataStream = new DataInputStream(new FileInputStream("SomeFile.dat"));
```

- A note about `FileInputStream`s:
 - The `FileInputStream` object's constructor throws an `FileNotFoundException` (a checked exception) if the file named in the argument can't be found.

Common DataInputStream Methods

Method	Return Type	Description	Possible Exceptions
readBoolean()	boolean	Reads a boolean value from the file.	EOFException, IOException
readByte()	byte	Reads a byte value from the file.	EOFException, IOException
readChar()	char	Reads a char value from the file. It is assumed the character was stored to the file as a two-byte Unicode character, as written by a DataOutputStream	EOFException, IOException
readDouble()	double	Reads a double value from the file.	EOFException, IOException
readFloat()	float	Reads a float value from the file.	EOFException, IOException
readInt()	int	Reads an int value from the file.	EOFException, IOException
readLong()	long	Reads a long value from the file.	EOFException, IOException
readShort()	short	Reads a short value from the file.	EOFException, IOException
readUTF()	String	Reads a String value from the file. The String must have been originally written to the file with the DataOutputStream's or RandomAccessFile's writeUTF method.	EOFException, IOException, UTFDataFormatException
close()	void/none	Closes the file.	IOException

Exceptions

- EOFException – “End Of File” Exception.
 - Caused by the program trying to read data beyond the end of the file.
- UTFDataFormatException
 - Caused by the binary data not being encoded in UTF.
 - eg. Trying to read in a String when the data is really a float, double, etc.

Reading Binary Files

```
DataInputStream dataStream = new DataInputStream(new FileInputStream("SomeFile.dat"));
```

```
int a = dataStream.readInt();           //Reads in 42 (int)  
double b = dataStream.readDouble();     //Reads in 98.6 (double)  
String c = dataStream.readUTF();        //Reads in "Hello World" (String)  
int d = dataStream.readInt();           //Reads in 1297 (int)
```

```
dataStream.close()
```



Reading any more would cause an EOFException

Note: The data must be read in the order the binary data was written.

More About Reading/Writing Strings

- The `DataOutputStream`'s `writeUTF` method uses UTF Encoding:
 - How it works:
 - A two byte integer indicating the number of bytes that the String uses is written.
 - Then, the String's individual characters are written to the file in Unicode.
- The `DataInputStream`'s `readUTF` method expects:
 - The first two bytes to contain the number of bytes, "X", this String needs.
 - The next X bytes in the file are the Unicode characters that comprise the String.
 - (All of this is abstracted by the `DataInputStream` and `DataOutputStream` objects/You don't have to worry about it as a programmer.)

More info:

[What Every Programmer Absolutely, Positively Needs To Know About Encodings And Character Sets To Work With Text](#)

More About Reading/Writing Binary Files

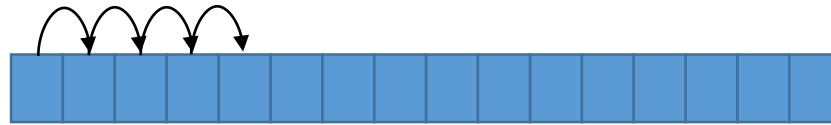
- A bit trickier to work with than plaintext files.
 - Every binary data file will be different.
- The program reading a binary file needs to know how the binary data is arranged in the file, so that the information is read correctly.

Sequential File Access

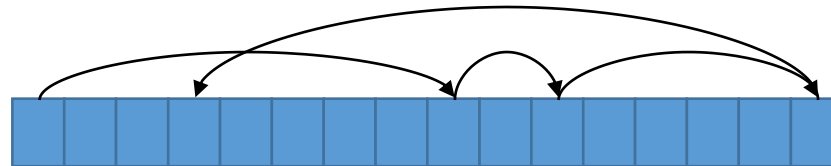
- So far in this lecture, we have been accessing files sequentially.
 - That is, we opened a file and the “read position” began at the beginning of the file. As we read through the lines/data, the read position advances from the beginning to the end of the file.
- The problem with sequential access is that to access specific data in the file, we must read through all of the file’s data that precedes it.

Direct/Random File Access

- Unlike sequential file access, random file access allows a program to immediately jump to any location in the file.



Sequential Access – Items/bytes are accessed one after the other



Random Access – Items/bytes are accessed in any order

Random Access Files

```
import java.io.RandomAccessFile;
```

- The RandomAccessFile class is used to read binary data files.
 - This object can jump around the file, whereas DataInputStream reads the file sequentially.
- The RandomAccessFile class's constructor takes two String parameters:
 - The first is the file name.
 - The second is the mode, either "r" for read-only access or "rw" for read and write access.

```
RandomAccessFile rFile = new RandomAccessFile("SomeFile.dat", "r");
```

or

```
RandomAccessFile rFile = new RandomAccessFile("SomeFile.dat", "rw");
```

Random Access Files – Access Modes

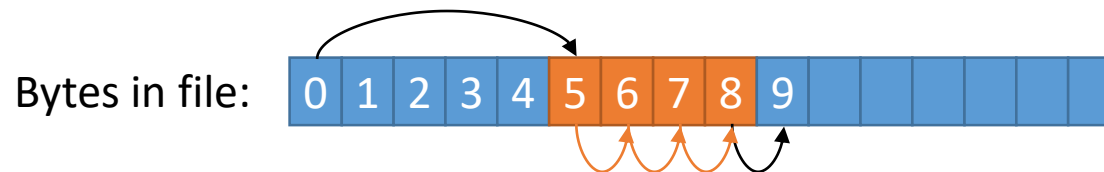
- Read Access / “r” mode:
 - If the filename provided does not exist, the RandomAccessFile’s constructor will throw a FileNotFoundException.
 - If you try to write to an “r” mode RandomAccessFile, an IOException will be thrown.
- Read/Write Access / “rw” mode:
 - If you open an existing file, neither it nor its contents will not be deleted.
 - If you open a file that does not exist it will be created.

Random Access Files

- RandomAccessFile objects treat a file as a stream of bytes.
 - The bytes are numbered from 0 to one less than the total number of bytes.
 - The byte numbers are similar to an array's subscripts.
- RandomAccessFile objects maintain a value known as the file pointer.
 - The file pointer “points” to the location of the current byte number.
 - When a RandomAccessFile object's file is first opened, the pointer is set to 0: the first byte of the file.
 - When a value is read from the file, it is read from the byte starting where the pointer is currently pointing to.

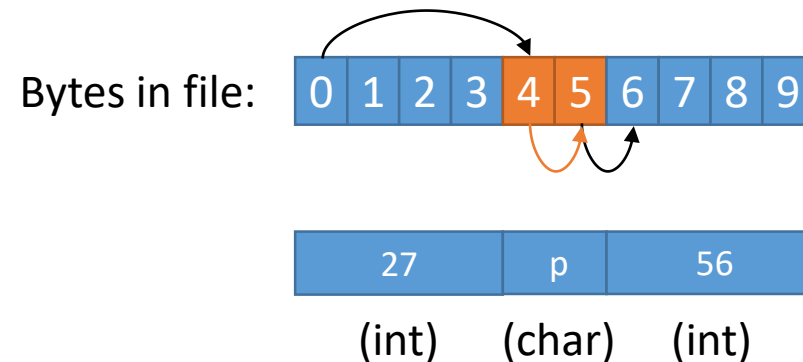
File Pointer Movement (Reading)

- Reading values causes the pointer to move ahead to the byte just beyond the value that was read.
 - Let's say our pointer is moved to byte number 5 and we want to read an int.
 - An int is 32-bits (or 4 bytes) so four bytes will be read from the file.
 - The pointer reads byte number 5, then byte number 6, then byte number 7, then byte number 8.
 - After the four bytes are read and the value is returned, the pointer will advance to the next byte to read, in this case byte number 9.
 - This way, we are ready to read the next sequential value from the file.
 - Or, we can move the pointer somewhere else.



Moving the File Pointer (Reading)

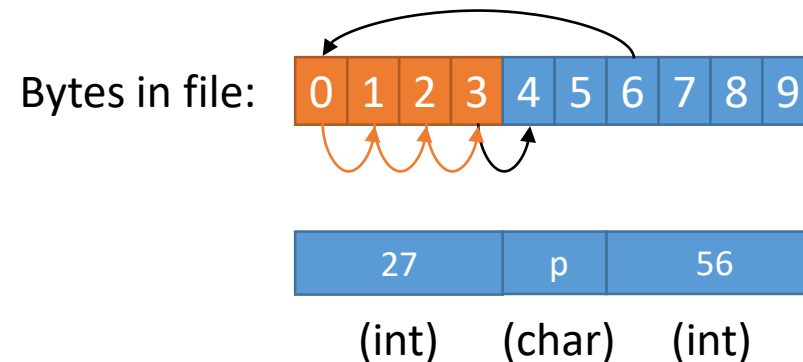
```
RandomAccessFile rFile = new RandomAccessFile("MyFile.dat", "rw");  
rFile.seek(4);  
char letter = rFile.readChar(); //Assigned 'p'
```



```
import java.io.RandomAccessFile;
```

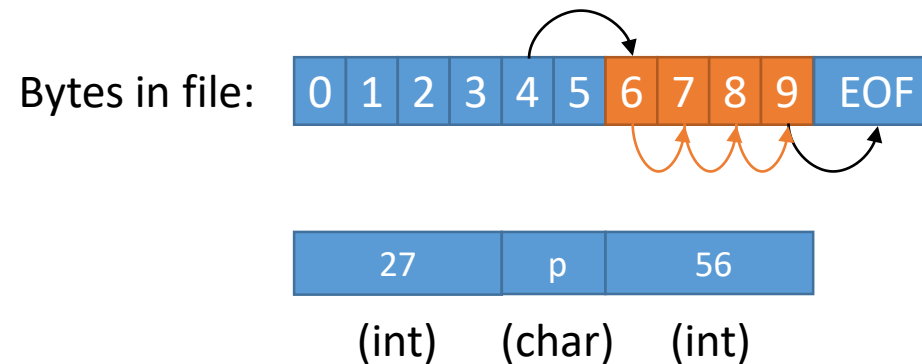
Moving the File Pointer (Reading)

```
RandomAccessFile rFile = new RandomAccessFile("MyFile.dat", "rw");  
rFile.seek(4);  
char letter = rFile.readChar(); //Assigned 'p'  
rFile.seek(0);  
int number = rFile.readInt();    //Assigned 27
```



Moving the File Pointer (Reading)

```
RandomAccessFile rFile = new RandomAccessFile("MyFile.dat", "rw");
rFile.seek(4);
char letter = rFile.readChar(); //Assigned 'p'
rFile.seek(0);
int number = rFile.readInt(); //Assigned 27
rFile.seek(6);
int number2 = rFile.readInt(); //Assigned 56
```



Attempting to *read* any data from here will result in an EOFException.

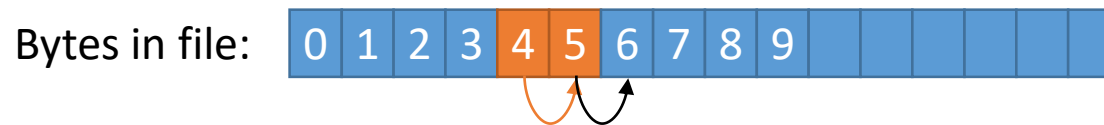
The pointer would need to be moved backwards.

File Pointer Movement (Writing)

- Writing values also takes place at the file pointer's current location.
- If the file pointer points to the end (last byte) of the file, data is written to the end of the file.
 - No EOFException
- If the file pointer points to a location in the file where there is existing data, the existing data is overwritten.

File Pointer Movement (Writing)

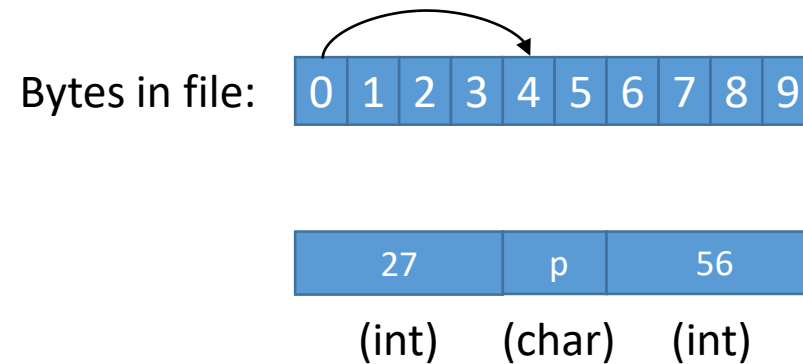
- Let's say our pointer is currently at byte number 4 and we want to write a char value to the file.
- A char is 16-bits (or 2 bytes) so the char's data will be written to bytes 4 and 5.
- After the two bytes are written, the pointer will advance to the next byte to read/written to, in this case byte number 6.



Note: We have now lost the int that was previously written to bytes 5-8 because we changed byte 5. The rest of the int's data is still in bytes 6, 7, and 8 but that data is effectively useless now.

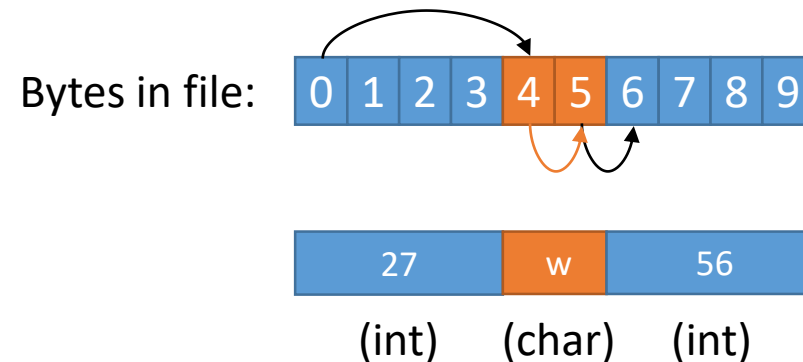
Moving the File Pointer (Writing)

```
RandomAccessFile rFile = new RandomAccessFile("MyFile.dat", "rw");  
rFile.seek(4);
```



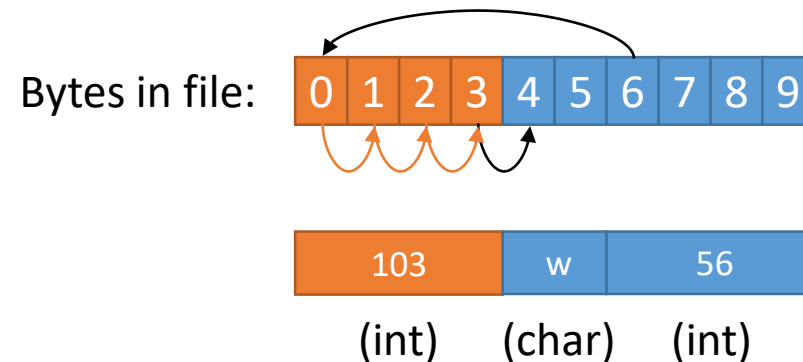
Moving the File Pointer (Writing)

```
RandomAccessFile rFile = new RandomAccessFile("MyFile.dat", "rw");  
rFile.seek(4);  
char letter = 'w';  
rFile.writeChar(letter); //Writes 'w' to this location in the file
```



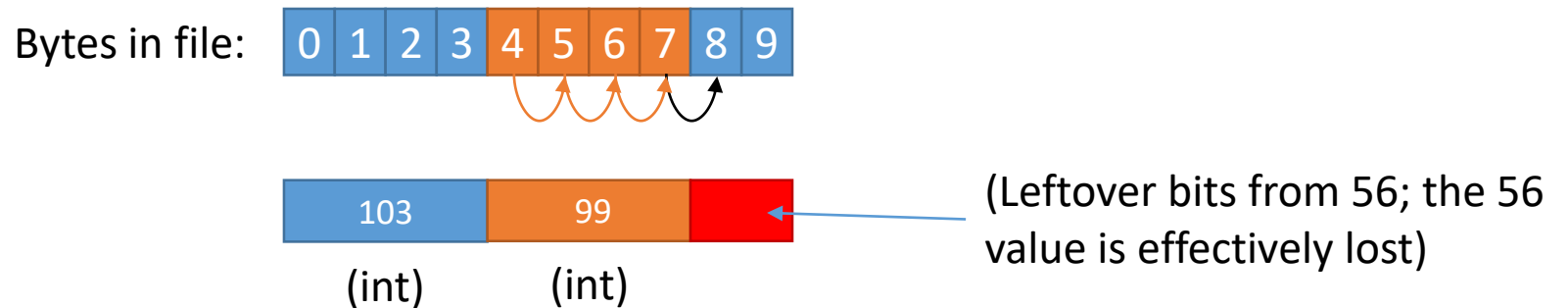
Moving the File Pointer (Writing)

```
RandomAccessFile rFile = new RandomAccessFile("MyFile.dat", "rw");  
rFile.seek(4);  
char letter = 'w';  
rFile.writeChar(letter);           //Writes 'w' to this location in the file  
rFile.seek(0);  
rFile.writeInt(103);               //Writes 103 to this location in the file
```



Moving the File Pointer (Writing)

```
RandomAccessFile rFile = new RandomAccessFile("MyFile.dat", "rw");
rFile.seek(4);
char letter = 'w';
rFile.writeChar(letter);           //Writes 'w' to this location in the file
rFile.seek(0);
rFile.writeInt(103);              //Writes 103 to this location in the file
rFile.writeInt(99);              //Writes 99 to current location in the file
```



Common RandomAccessFile Methods (Reading)

Method	Return Type	Description	Exceptions
readBoolean()	boolean	Reads a boolean value from the file.	EOFException, IOException
readByte()	byte	Reads a byte value from the file.	EOFException, IOException
readChar()	char	Reads a char value from the file. It is assumed the character was stored to the file as a two-byte Unicode character, as written by a DataOutputStream	EOFException, IOException
readDouble()	double	Reads a double value from the file.	EOFException, IOException
readFloat()	float	Reads a float value from the file.	EOFException, IOException
readInt()	int	Reads an int value from the file.	EOFException, IOException
readLong()	long	Reads a long value from the file.	EOFException, IOException
readShort()	short	Reads a short value from the file.	EOFException, IOException
readUTF()	String	Reads a String value from the file. The String must have been originally written to the file with the DataOutputStream's or RandomAccessFile's writeUTF method.	EOFException, IOException, UTFDataFormatException
close()	void/none	Closes the file.	IOException

Common RandomAccessFile Methods (Writing)

Method	Return Type	Description	Exceptions
getFilePointer()	long	Current pointer location.	IOException
length()	long	Total number of bytes in the file	IOException
seek(long)	void	Repositions the file pointer	IOException
writeBoolean(boolean)	void	Writes the boolean value to the file.	IOException
writeByte(byte)	void	Writes the byte value to the file.	IOException
writeChar(int)	void	It is assumed the int is a character code. The character it represents is written to the file as a two-byte Unicode character.	IOException
writeDouble(double)	void	Writes the double value to the file.	IOException
writeFloat(float)	void	Writes the float value to the file.	IOException
writeInt(int)	void	Writes the int value to the file.	IOException
writeLong(long)	void	Writes the long value to the file.	IOException
writeShort(short)	void	Writes the short value to the file.	IOException
writeUTF(String)	void	Writes the String to the file using Unicode Text Format.	IOException
close()	void/none	Closes the file.	IOException

Random Access Files

- Same general idea writing/reading binary files as with `DataInputStream` and `DataOutputStream`.
- With `RandomAccessFile` objects, we can move around in the file instead of needing to read/write sequentially.

Object Serialization

- We have seen how data like primitives and Strings can be saved to and retrieved from files.
- We can also store and retrieve *entire objects* (including their current states) to and from files through **serialization**.
- When an object is serialized, it is converted to a set of bytes that contain the object's data.
 - The resulting set of bytes can be written to a binary file.

Object Serialization

- In order for an object to be serializable, it must implement the Serializable interface.
- The Serializable interface has **no fields or methods to actually implement**; it simply informs the compiler that this object might be serialized.
- If your object aggregates other objects (such as an instance variable of another class type) those other objects must also implement the Serializable interface, in order for them to be serialized as well.

Serializing/Writing Objects to Binary Files

- Use the following classes in conjunction with one another:
 - `FileOutputStream`: Allows you to open a file for writing binary data. It only provides basic functionality for writing bytes to the file. (Same one used previously)
 - **`ObjectOutputStream`**: Can handle serializing objects to a series of bytes. (Can also write primitives and Strings like `DataOutputStream`.)

```
FileOutputStream fileStream = new FileOutputStream("SomeFile.dat");  
ObjectOutputStream dataStream = new ObjectOutputStream(fileStream);
```

Or

```
ObjectOutputStream dataStream = new ObjectOutputStream(new FileOutputStream("SomeFile.dat"));
```

Writing Objects to Binary Files

```
import java.io.ObjectOutputStream;
```

- Use ObjectOutputStream's writeObject method to serialize the object and store it to a file.

```
ObjectOutputStream dataStream = new ObjectOutputStream(new FileOutputStream("SomeFile.dat"));
```

```
Rectangle r1 = new Rectangle(5, 7);
```

```
Rectangle r2 = new Rectangle(8, 9);
```

```
dataStream.writeObject(r1);
```

```
dataStream.writeUTF("Hello World");
```

```
dataStream.writeObject(r2);
```

```
//Serializes r1 to the file
```

```
//Writes a String to the file
```

```
//Serializes r2 to the file
```

```
dataStream.close();
```

```
public class Rectangle implements Serializable {  
    Fields, Constructors, Methods  
}
```


Deserializing/Reading Objects from Binary Files

- Use the following classes in conjunction with one another:
 - `FileInputStream`: Allows you to open a file for reading binary data. It only provides basic functionality for reading bytes from the file. (Same one used previously)
 - **`ObjectInputStream`**: Can handle deserializing objects from a series of bytes. (Can also read primitives and Strings like `DataInputStream`.)

```
FileInputStream fileStream = new FileInputStream("SomeFile.dat");  
ObjectInputStream dataStream = new ObjectInputStream(fileStream);
```

Or

```
ObjectInputStream dataStream = new ObjectInputStream(new FileInputStream("SomeFile.dat"));
```

```
import java.io.ObjectInputStream;
```

Reading Objects from Binary Files

- Use ObjectInputStream's read Object method to deserialize the object from a file.

```
ObjectInputStream dataStream = new ObjectInputStream(new FileInputStream("SomeFile.dat"));

Rectangle r1 = (Rectangle)dataStream.readObject();
String text = dataStream.readUTF();
Rectangle r2 = (Rectangle)dataStream.readObject();

dataStream.close();
```

- You must typecast the object returned from the readObject method to the correct object type.

Common ObjectOutputStreams Methods

Method	Return Type	Description	Exceptions
writeBoolean(boolean)	void	Writes the boolean value to the file.	IOException
writeByte(byte)	void	Writes the byte value to the file.	IOException
writeChar(int)	void	It is assumed the int is a character code. The character it represents is written to the file as a two-byte Unicode character.	IOException
writeDouble(double)	void	Writes the double value to the file.	IOException
writeFloat(float)	void	Writes the float value to the file.	IOException
writeInt(int)	void	Writes the int value to the file.	IOException
writeLong(long)	void	Writes the long value to the file.	IOException
writeShort(short)	void	Writes the short value to the file.	IOException
writeObject(Object)	void	Serializes and writes the object to the file.	InvalidClassException, NotSerializableException, IOException
writeUTF(String)	void	Writes the String to the file using Unicode Text Format.	IOException
close()	void/none	Closes the file.	IOException

Common ObjectInputStream Methods

Method	Return Type	Description	Exceptions
readBoolean()	boolean	Reads a boolean value from the file.	EOFException, IOException
readByte()	byte	Reads a byte value from the file.	EOFException, IOException
readChar()	char	Reads a char value from the file. It is assumed the character was stored to the file as a two-byte Unicode character, as written by a DataOutputStream	EOFException, IOException
readDouble()	double	Reads a double value from the file.	EOFException, IOException
readFloat()	float	Reads a float value from the file.	EOFException, IOException
readInt()	int	Reads an int value from the file.	EOFException, IOException
readLong()	long	Reads a long value from the file.	EOFException, IOException
readShort()	short	Reads a short value from the file.	EOFException, IOException
readUTF()	String	Reads a String value from the file. The String must have been originally written to the file with the DataOutputStream's, ObjectOutputStream's, or RandomAccessFile's writeUTF method.	EOFException, IOException, UTFDataFormatException
readObject()	Object	Reads and deserializes an object, returning the object in the same state as it was prior to being serialized to the file.	ClassNotFoundException, InvalidClassException, StreamCorruptedException, OptionalDataException, IOException
close()	void/none	Closes the file.	IOException