BBM 102 – Introduction to Programming II

Spring 2017

Streams and Input/Output

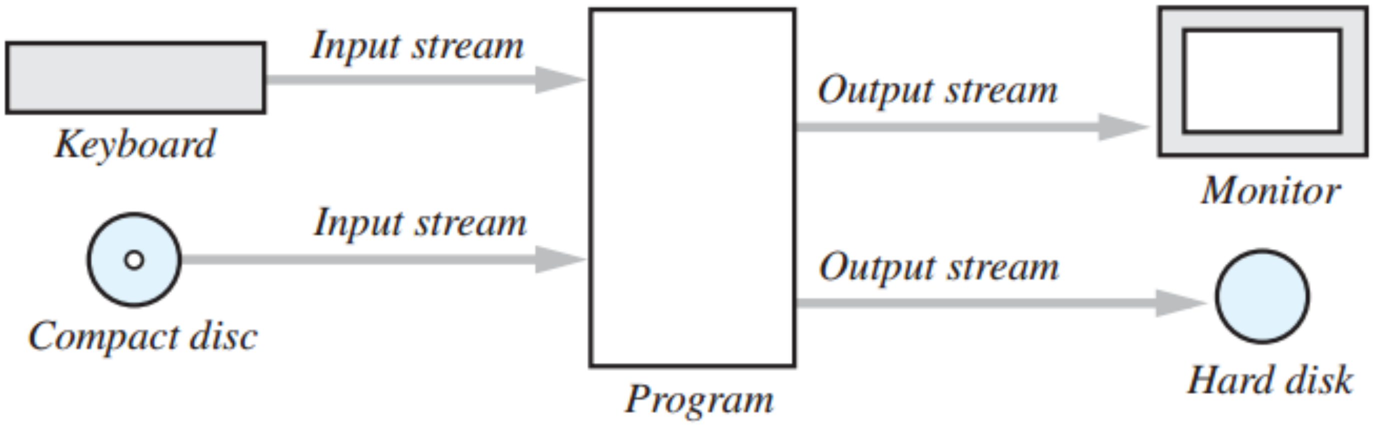
Instructors: *Ayça Tarhan, Fuat Akal, Gönenç Ercan, Vahid Garousi*

Today

- Streams and Files
- Text/Binary Files
- java.io.File class
- Revisiting java.util.Scanner
- Java I/O Library
- Decorator Pattern
- **InputStream**s and **OutputStream**s
- **Reader**s and **Writer**s
- Sequential Access vs Random Access
- java.io.RandomAccessFile
- Serialization

Streams

- A **stream** is a flow of data. The data might be characters, numbers, or bytes consisting of binary digits.
- If the data flows into your program, the stream is called an **input stream** (example: **System.in**).
- If the data flows out of your program, the stream is called an **output stream** (example: **System.out**).



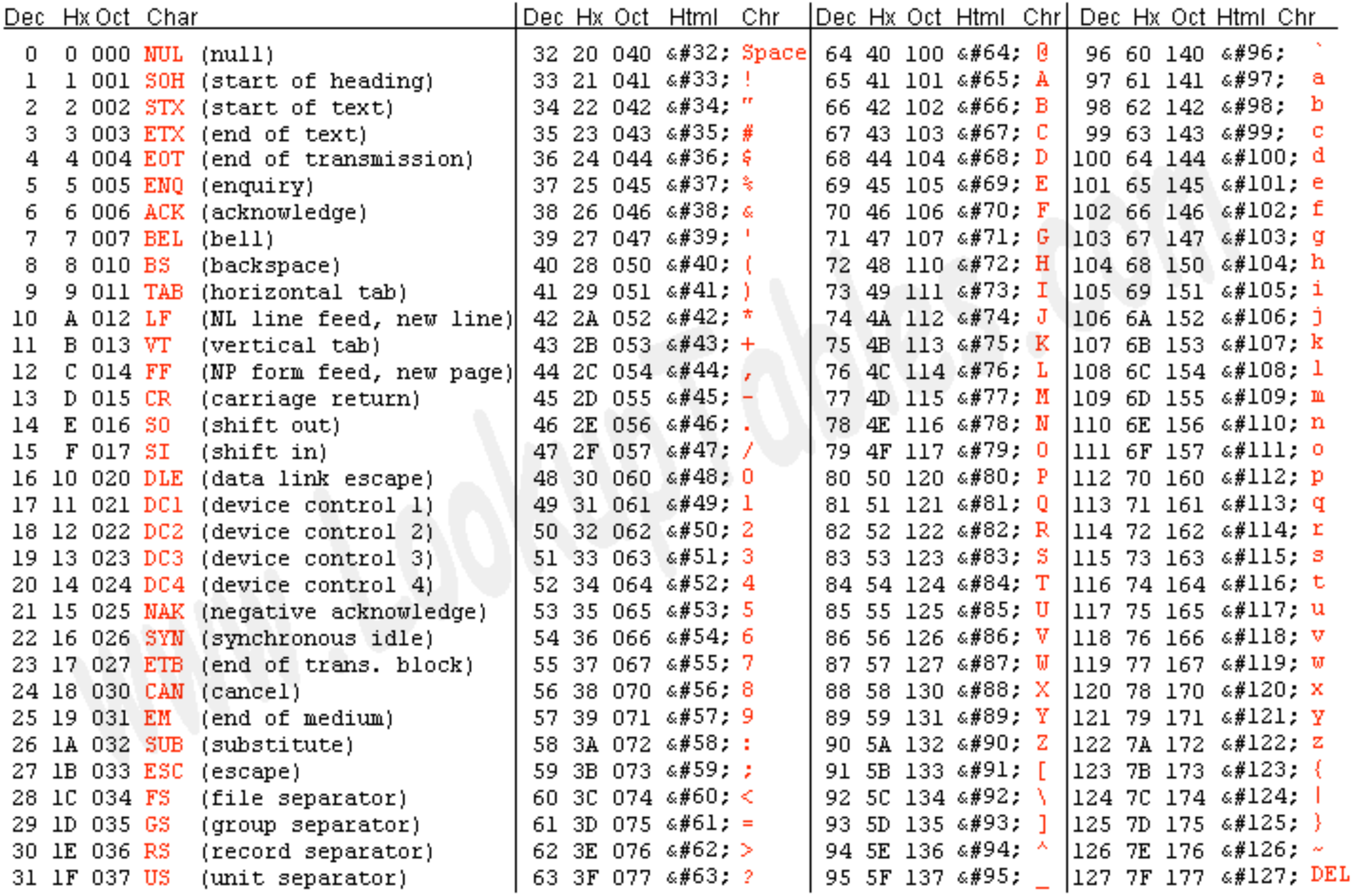
Files

- The keyboard and the screen deal with temporary data
- Files provide a way to store data permanently
- All of the data in any file is stored as bits, or 0s and 1s.
- Files are categorized as **text files** and **binary files**

Text/Binary Files

- Text files
  - The bits represent printable (easily readable by humans when printed) characters.
  - The characters are coded with a "character set", ASCII, ISO-8859-1, utf-8..
  - They can be edited with a " text editor "
  - Examples: Program source files (.java, .c), files saved with a text editor, e.g. *Notepad.exe*
- Binary Files
  - The bits represent other types of encoded information, such as executable instructions or numeric data
  - They are easily read by the computer but not humans
  - They are *not* "printable" files
  - Examples: Executables (.exe), images (.jpg, .png), music (.mp3), or video (.avi, .mov) files

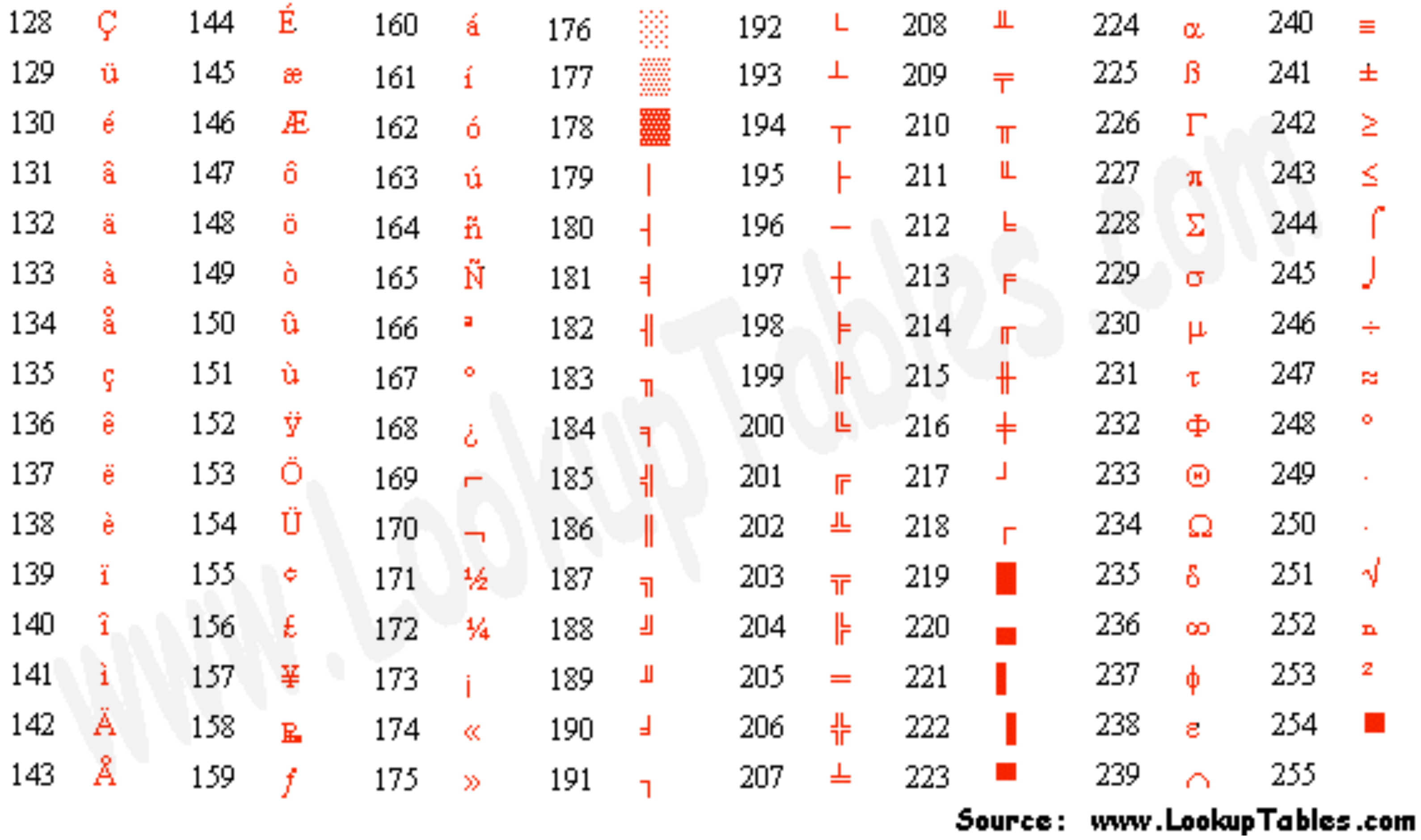
ASCII (American Standard Code for Information Interchange) Code Table



Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH	(start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX	(start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX	(end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT	(end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ	(enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK	(acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL	(bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS	(backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB	(horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT	(vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR	(carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO	(shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI	(shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE	(data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1	(device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2	(device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3	(device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4	(device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK	(negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB	(end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN	(cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM	(end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB	(substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC	(escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS	(file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS	(group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS	(record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US	(unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Extended ASCII Codes



128	Ç	144	É	160	á	176	░	192	└	208	╨	224	α	240	≡
129	ü	145	æ	161	í	177	▒	193	┴	209	╤	225	ß	241	±
130	é	146	Æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
131	â	147	ô	163	ú	179	│	195	├	211	╙	227	π	243	≤
132	ä	148	ö	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
133	à	149	ò	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
134	å	150	û	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
135	ç	151	ù	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
136	ê	152	ÿ	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
137	ë	153	Ö	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	∙
138	è	154	Ü	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
139	ï	155	¢	171	½	187	╗	203	╦	219	█	235	δ	251	√
140	î	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	ⁿ
141	ì	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	²
142	Ä	158	₧	174	«	190	╛	206	╬	222	▐	238	ε	254	■
143	Å	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	

Source: www.LookupTables.com

Text/Binary Files

- Confused? Let's see an example: We want to write the number 127 into a file.

- If we write it into an ASCII coded text file:
  - Three bytes will be used for each character: 1 , 2, and 7
  - Binary values of these characters: 00110001, 00110010, 00110111

- If we write it into a binary file:
  - One byte (variable is defined as byte): 01111111
  - Two bytes (variable is defined as short): 00000000 01111111
  - Four bytes (variable is defined as int):
    00000000 00000000 00000000 01111111

java.io.File

- Do not be deceived with the name of it! Class represents a path rather than a file!
- Can be used to
  - Check if the path exists or not
  - Check if the path is a file or a directory
  - Check/edit the file/directory’s readable, writable, executable, hidden properties
  - Create/delete file/directory
  - Get the contents of a directory
  - Get the last modification date and time of the file/directory

FileExample Program

```
public class FileExample {
    public static void main(String[] args) {
        File path = new File("h:\\example");
        if (!path.exists()) {                // It does not exist, create a directory!
            path.mkdir();
        } else if (path.isDirectory()) { // It is a directory! List the contents
            String[] contentOfDirectory = path.list();
            for (String filename : contentOfDirectory) {
                System.out.println(filename);
            }
        } else {                             // It is a file! Display the properties of the file
            System.out.println("Read:" + path.canRead() +
                ", Write: " + path.canWrite() + ", Hidden: " + path.isHidden());
        }
    }
}
```

Revisiting java.util.Scanner

- Class Scanner is an easy way to read input from keyboard, remember?

```
 // create a scanner System.in (keyboard)
Scanner scanner = new Scanner(System.in);
// read a string from keyboard and write it to System.out (monitor)
System.out.println(scanner.next());
```

- It takes an inputstream to its constructor and reads from it

- What if we give a File object to the constructor?

```
 // create a scanner for the file example.txt
scanner = new Scanner(new File("c:example.txt"));
// read a string from the file and write it to System.out (monitor)
System.out.println(scanner.next());
```

Scanner example: display contents of a file

```
public static void main(String[] args) {
        Scanner scanner = null;
        try {
                scanner = new Scanner(new File(args[0]));
                while (scanner.hasNext()) {
                        System.out.println(scanner.nextLine());
                }
        } catch (Exception e) {
                e.printStackTrace();
        } finally {
                if (scanner != null) scanner.close();
        }
}
```

Java I/O Library

- Mostly under the package **java.io**
- Includes classes, interfaces and exceptions for
  - Input/Output
  - Binary/Text
  - Sequential/Random Access
- JDK versions improved the library in time, adding new classes/interfaces.

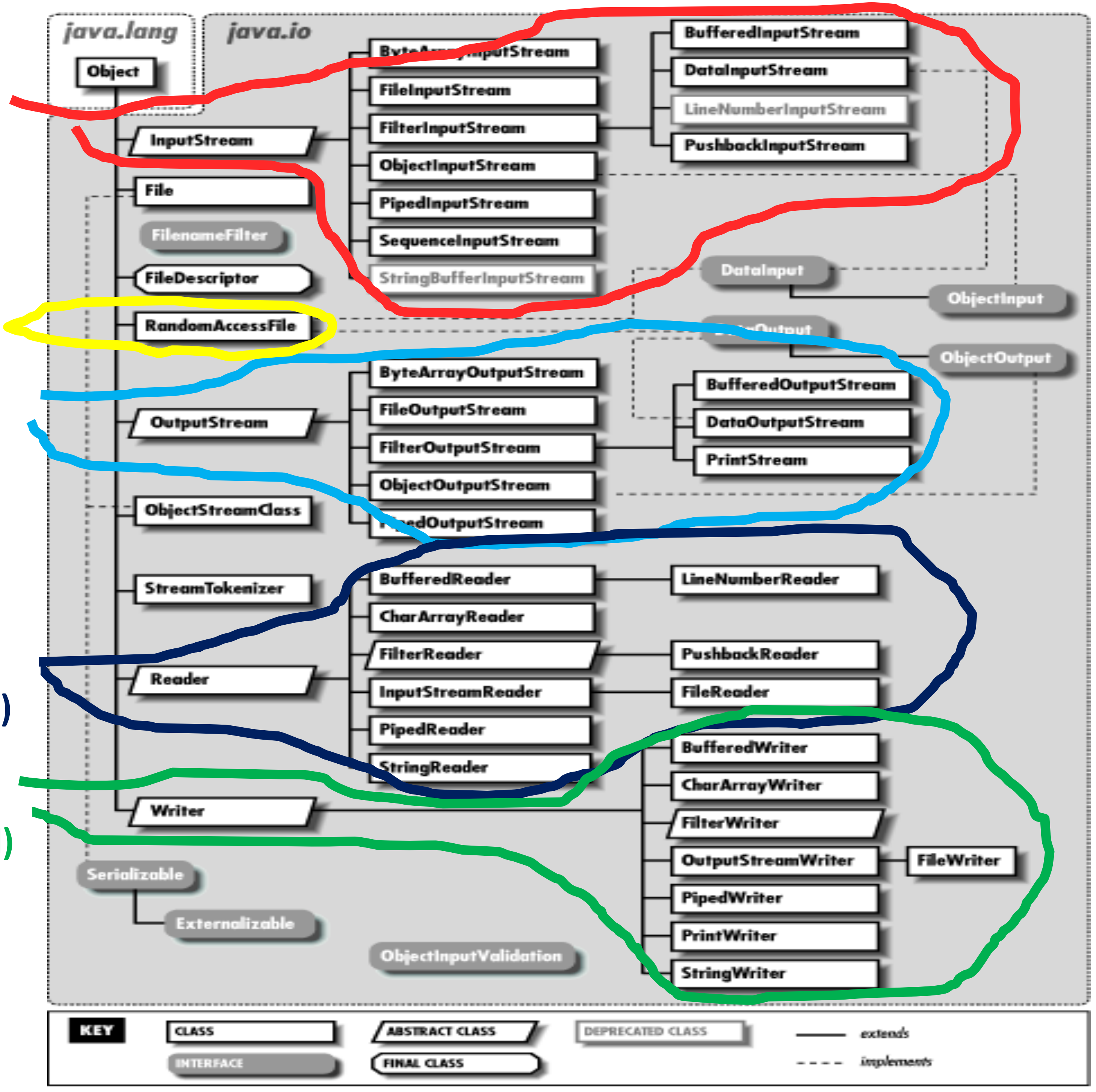
Binary Input
(byte oriented)

Random Access

Binary Output
(byte oriented)

Text Input
(character oriented)

Text Output
(character oriented)



Creating a text file

- An easy way to create a text file is using **java.io.PrintWriter**

```
public static void main(String[] args) {
        PrintWriter outputStream = null;
        try {
                outputStream = new PrintWriter("c:out.txt");  // open the file
                outputStream.println("Example line.."); // write something to the file
        } catch(FileNotFoundException e) {
                System.out.println("Error opening the file!");
        } finally {
                if (outputStream != null)     outputStream.close(); // close the file
        }
}
```

Example: from keyboard to file

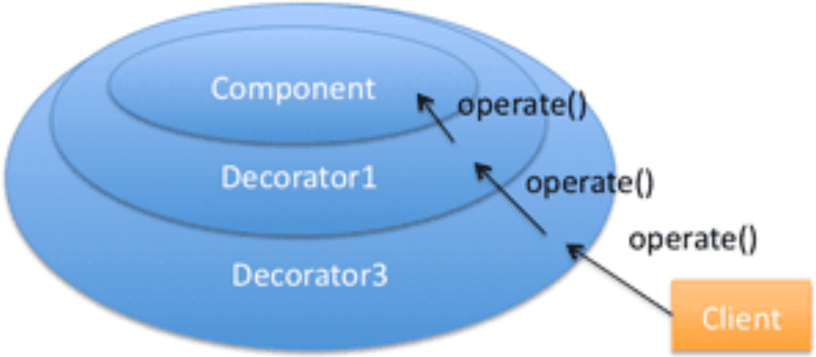
```
public static void main(String[] args) {
    PrintWriter outputStream = null;
    Scanner scanner = null;
    try {
        outputStream = new PrintWriter(args[0]);  // open the file
        scanner = new Scanner(System.in);         // create scanner for keyboard
        String str = scanner.nextLine();          // get the first line
        while (!str.equalsIgnoreCase("exit")) {   // if it is not «exit»
            outputStream.println(str);            // write it to the file
            str = scanner.nextLine();             // get a new line
        }
    } catch(FileNotFoundException e) {
        System.out.println("Error opening the file!");
    } finally {
        if (outputStream != null) outputStream.close();      // close the file
        if (scanner != null) scanner.close();                // close the scanner
    }
}
```

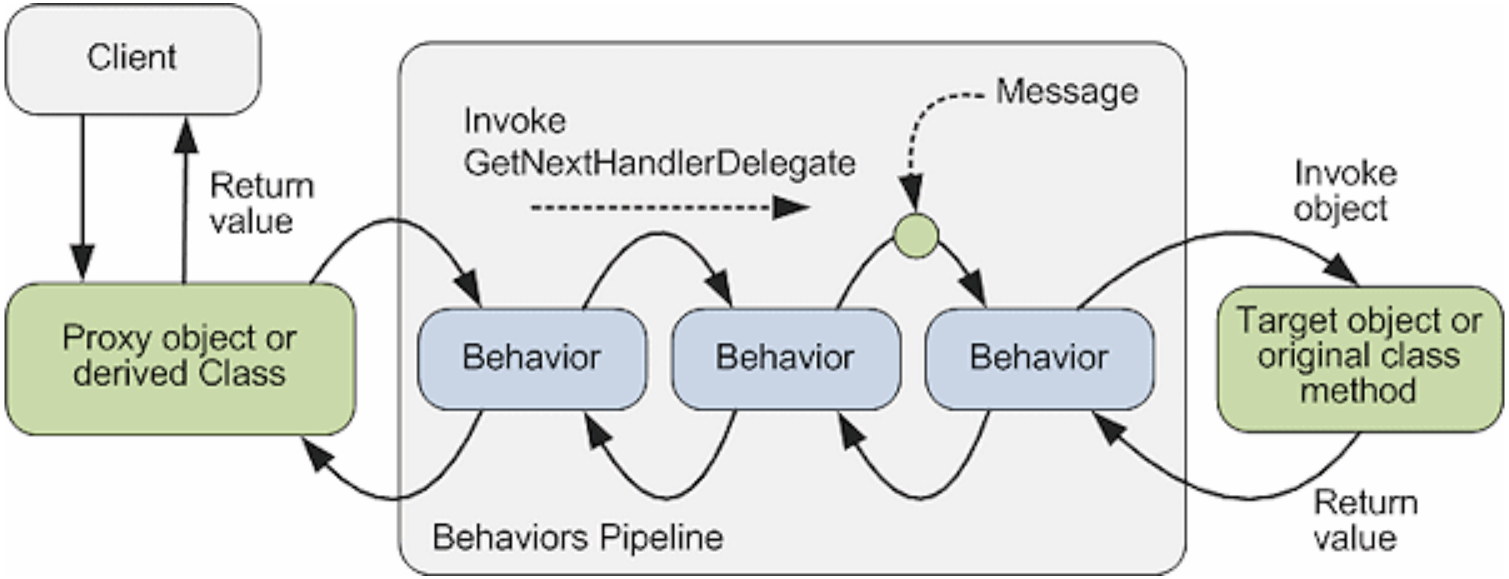
Decorator Pattern

- Software Design Patterns
  - "In software engineering, a **design pattern** is a general reusable solution to a commonly occurring problem within a given context in software design" (wikipedia)
  - Design patterns gained popularity in computer science after the book ***Design Patterns: Elements of Reusable Object-Oriented Software*** was published in 1994 by the so-called "Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides), which is frequently abbreviated as ”GoF".

Decorator Pattern

- Decorator Pattern adds a new functionality to an existing object
- A decorator class decorates an inner object and uses its methods to serve in a different way




Client
Return value
Proxy object or derived Class
Invoke GetNextHandlerDelegate
Message
Behavior
Behavior
Behavior
Behaviors Pipeline
Invoke object
Target object or original class method
Return value

Decorator Pattern in java.io

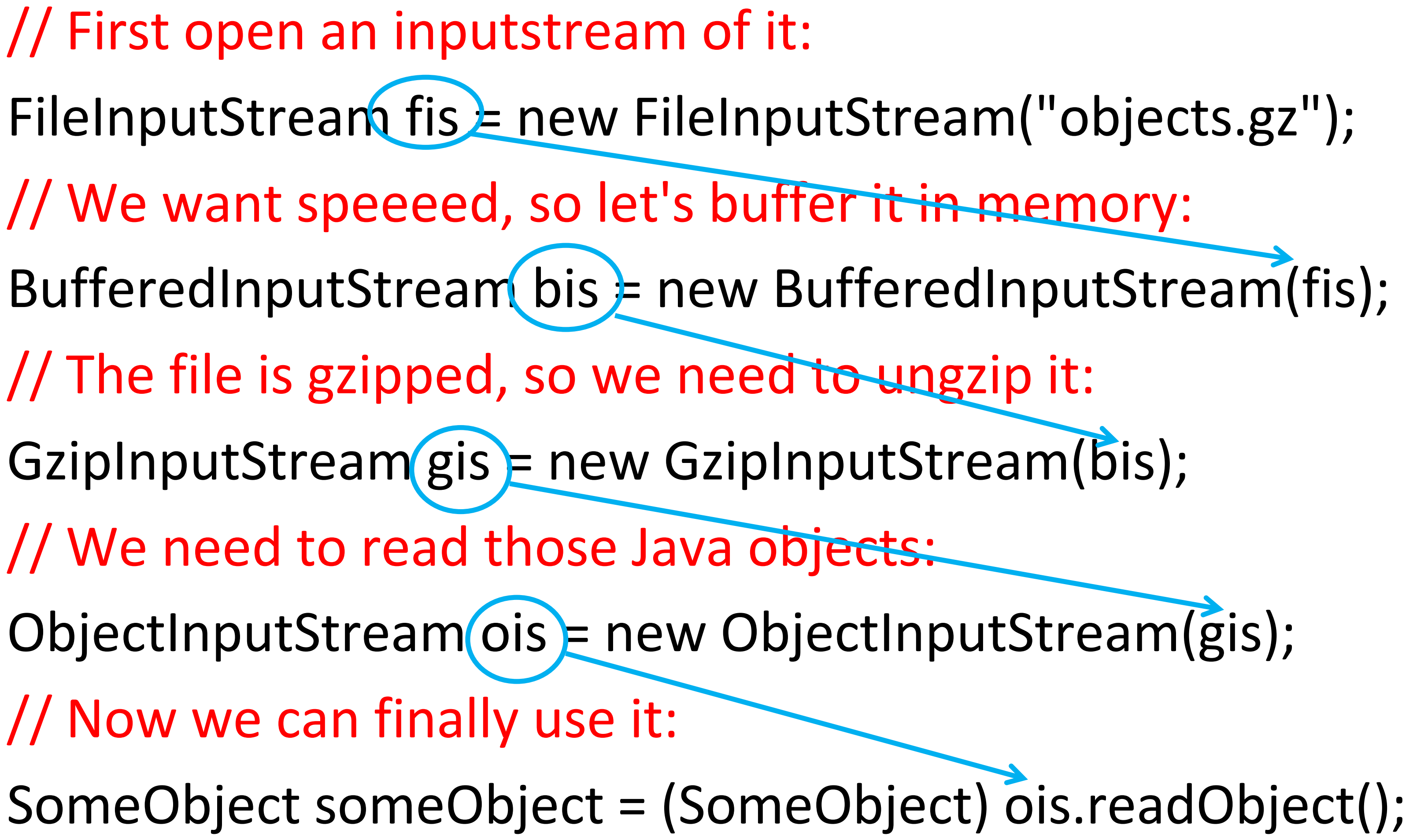
- **InputStream** and **Reader** classes (and their subclasses) has basic methods called **read()** for reading a single byte or an array of bytes
- **OutputStream** and **Writer** classes (and their subclasses) has basic methods called **write()** for writing a single byte or an array of bytes
- Problem: A new access to the disk for each byte will slow down the application seriously
- Solution: Bytes may be collected before reading from or writing to the disk. This will reduce the number of physical disk operations
- Decorator classes
  - java.io.BufferedInputStream, java.io.BufferedReader
  - java.io.BufferedOutputStream, java.io.BufferedWriter

BufferedReader example

```
public static void main(String[] args) {
    BufferedReader reader = null;
    try {
        reader = new BufferedReader(new FileReader(new File(args[0])));
        String line;
        while ((line = reader.readLine()) != null) {
            System.out.println(line);
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (reader != null) reader.close();
    }
}
```

A more complicated decoration example

- Let's say that we have a bunch of Java objects in a Gzipped file named ‘objects.gz’ and that we want to read them a bit quickly



```
// First open an inputstream of it:
FileInputStream fis = new FileInputStream("objects.gz");
// We want speeeed, so let's buffer it in memory:
BufferedInputStream bis = new BufferedInputStream(fis);
// The file is gzipped, so we need to ungzip it:
GzipInputStream gis = new GzipInputStream(bis);
// We need to read those Java objects:
ObjectInputStream ois = new ObjectInputStream(gis);
// Now we can finally use it:
SomeObject someObject = (SomeObject) ois.readObject();
```

InputStream and subclasses

InputStream’s job is to represent classes that produce input from different sources. These sources can be:

- An array of bytes (**java.io.ByteArrayInputStream**)
- A String object (**java.io.StringBufferInputStream**)
- A file (**java.io.FileInputStream**)
- A "pipe," (**java.io.PipedInputStream**)
  - Pipe works like a physical pipe: You put things in at one end and they come out the other.
- A sequence of other streams, so you can collect them together into a single stream (**java.io.SequenceInputStream**)
- Other sources, such as an Internet connection

OutputStream and subclasses

- An array of bytes (**java.io.ByteArrayOutputStream**)
- A file (**java.io.FileOutputStream**)
- A "pipe," (**java.io.PipedOutputStream**)
  - Pipe works like a physical pipe: You put things in at one end and they come out the other.

Homework

- Go over the input and out stream classes mentioned in the previous two slides!
- Try to understand at least how they basically work.

Decorating InputStreams

- **java.io.DataInputStream**: read primitives (int, char, long, etc.) from a stream in a portable fashion.

- **java.io.BufferedInputStream**: prevents a physical read every time you want more data.

- **java.io.LineNumberInputStream**: Keeps track of line numbers in the input stream; you can call getLineNumber( ) and setLineNumber (int).
  - *This class incorrectly assumes that bytes adequately represent characters.*

- **java.io.PushbackInputStream**: Has a one-byte pushback buffer so that you can push back the last character read.

Decorating OutputStreams

- **java.io.DataOutputStream**: write primitives (int, char, long, etc.) from a stream in a portable fashion.

- **java.io.BufferedOutputStream**: prevent a physical write every time you send a piece of data.

- **java.io.PrintStream**: For producing formatted output. While DataOutputStream handles the storage of data, PrintStream handles display

Example Program: create a copy of a file

```
public static void main(String[] args) throws Exception {
        BufferedInputStream bis = null;
        BufferedOutputStream bos = null;
        try {
                bis = new BufferedInputStream(new FileInputStream(new File(args[0])));
                bos = new BufferedOutputStream(new FileOutputStream(new File(args[1])));
                byte oneByte;
                // read a byte. -1 will be returned at the end of the file.
                while ((oneByte = bis.read()) != -1) {
                        bos.write(oneByte);                    //  write the byte to the output
                }
        } finally {
                if (bis != null) bis.close();                  // close the streams
                if (bos != null) bos.close();
        }
}
```

Is it too slow?

```
public static void main(String[] args) throws Exception {
    BufferedInputStream bis = null;
    BufferedOutputStream bos = null;
    byte[] bytes = new byte[1024 * 16];   // bytes will be read in this by 16K chunks
    try {
        bis = new BufferedInputStream(new FileInputStream(new File(args[0])));
        bos = new BufferedOutputStream(new FileOutputStream(new File(args[1])));
        int size;
        while ((size = bis.read(bytes)) > -1) {
            bos.write(bytes);
        }
    } finally {
        if (bis != null) bis.close();
        if (bos != null) bos.close();
    }
}
```

Another example: download a web page

```
// please note that exception handling is not coded properly!!

public static void main(String[] args) throws Exception {
        URL url = new URL("http://web.cs.hacettepe.edu.tr/~bbm102/");
        BufferedInputStream bis = new BufferedInputStream(url.openStream());
        BufferedOutputStream bos = new BufferedOutputStream(
                        new FileOutputStream(new File("downloadedPage.html")));
        for (int c = bis.read(); c != -1; c = bis.read()) {
                bos.write(c);
        }
        bis.close();
        bos.close();
}
```

Readers and Writers

- **InputStream** and **OutputStream** classes provide functionality in the form of **byte oriented I/O**
- **Reader** and **Writer** were added to the library with Java 1.1. These classes provide **Unicode-compliant, character-based I/O**
- Almost all of the original Java I/O stream classes have corresponding **Reader** and **Writer** classes

InputStream/OutputStream Reader/Writer correspondings

InputStream/OutputStream	Reader/Writer
InputStream	Reader adapter: InputStreamReader
OutputStream	Writer adapter: OutputStreamWriter
FileInputStream	FileReader
FileOutputStream	FileWriter
StringBufferInputStream	StringReader
(no corresponding class)	StringWriter
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter
PipedInputStream	PipedReader
PipedOutputStream	PipedWriter

Decorator correspondings

InputStream/OutputStream	Reader/Writer
BufferedInputStream	BufferedReader
BufferedOutputStream	BufferedWriter
PrintStream	PrintWriter
LineNumberInputStream	LineNumberReader
PushbackInputStream	PushbackReader

Example program: copy a file line by line

```
public static void main(String[] args) throws Exception {
    BufferedReader br = null;
    PrintWriter pw = null;
    try {
        br = new BufferedReader(new FileReader(new File(args[0])));
        pw = new PrintWriter(
                new BufferedWriter(new FileWriter(new File(args[1]))));
        String line;
        while ((line = br.readLine()) != null) {    // read a line. null at the end of the file
            pw.println(line);                       // write a line
        }
    } finally {
        if (br != null) br.close();
        if (pw != null) pw.close();
    }
}
```

Random Access

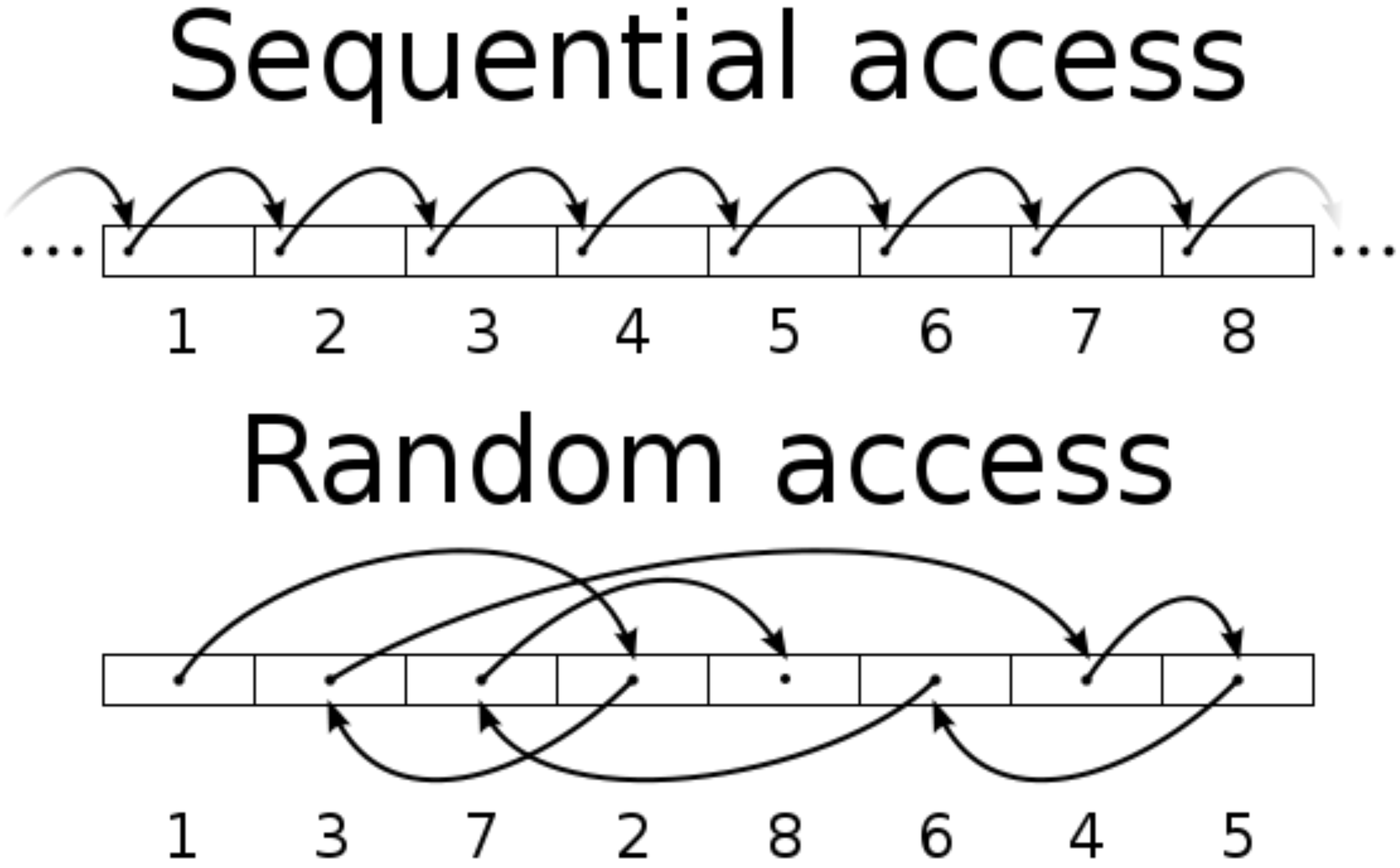
- Reading the next byte/string/number or writing to the next location is called sequential access.

- Sequential access is easy and efficient when you don’t know the contents of a file or just want to create a copy of it for example.

- On the other hand, if you know the sizes of records in a file, you can move in the file to read or change a specific record. This is random access.

- All records don’t have to be the same size; you just have to determine how big they are and where they are placed in the file.

Let’s clarify it by an image



java.io.RandomAccessFile

- Used for random access.

- Is not part of the InputStream or OutputStream hierarchy. *It's a completely separate class, written from scratch.*

- Some methods:
  - **getFilePointer( )**: find out where you are in the file
  - **seek( )**: move to a new point in the file
  - **length()**: return the length of the file

- the constructors require a second argument (*identical to fopen() in C*) indicating whether you are just randomly reading ("r") or reading and writing ("rw"). There's no support for write-only files

Example program: Editing courses Course.java

```
public class Course {
    private String code;
    private String name;
    private int credit;

    public Course(String c, String n, int cr) {
        this.setCode(c);
        this.setName(n);
        this.credit = cr;
    }
    public int getCredit() { return credit; }
    public void setCredit(int c) { this.credit = c; }

    public String getCode() { return code; }
    public void setCode(String code) {
        this.code = to40Chars(code);
    }
```

```
    public String getName() { return name; }
    public void setName(String name) {
        this.name = to40Chars(name);
    }
    private String to40Chars(String str) {
        String tmp = str;
        for (int i = str.length(); i < 40; i++) {
            tmp += ' ';
        }
        return tmp.substring(0, 40);
    }
    public String toString() {
        return code + " - " +
            name + " - " +
            credit;
    }
}
```

Program.java

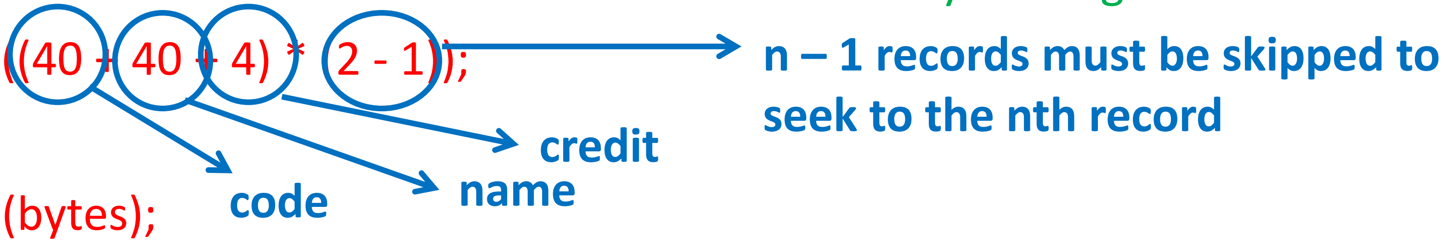
RandomAccessFile is given a mode while opening the file. rw: read/write (similar to c)

```
public static void main(String[] args) throws Exception {
    // create course objects
    Course[] courses = new Course[4];
    courses[0] = new Course("BBM101", "Programlamaya Giris I", 3);
    courses[1] = new Course("BBM102", "Programlamaya Giris II", 3);
    courses[2] = new Course("BBM103", "Programlamaya Giris Lab I", 3);
    courses[3] = new Course("BBM104", "Programlamaya Giris Lab II", 3);
    // open the file. It will be accessed randomly
    RandomAccessFile raf = new RandomAccessFile(new File("courses.txt"), "rw");
    // write the courses to a file. Each course is a record
    for (int i = 0; i < courses.length; i++) {
        raf.writeBytes(courses[i].getCode());    // write the code as string
        raf.writeBytes(courses[i].getName());    // write the name as string
        raf.writeInt(courses[i].getCredit());    // write the credit as int
    }
```

Program.java (continued)

```
// let’s read the second course’s data and create a course object
byte[] bytes = new byte[40];          // data will be read in this as chunks of 40 bytes

// seek to the 2nd record. each record is 40 + 40 + 4 bytes long.
raf.seek((40 + 40 + 4) * (2 - 1));
```



```
raf.read(bytes);
String code = new String(bytes);     // first 40 byte is the code of the course

raf.read(bytes);                     // second 40 byte is the name of the course
String name = new String(bytes);

raf.read(bytes, 0, 4);               // read 4 bytes: the credit
int credit = ByteBuffer.wrap(bytes).getInt();             // convert byte array to int
System.out.println(new Course(code, name, credit));  // create and print the course
```

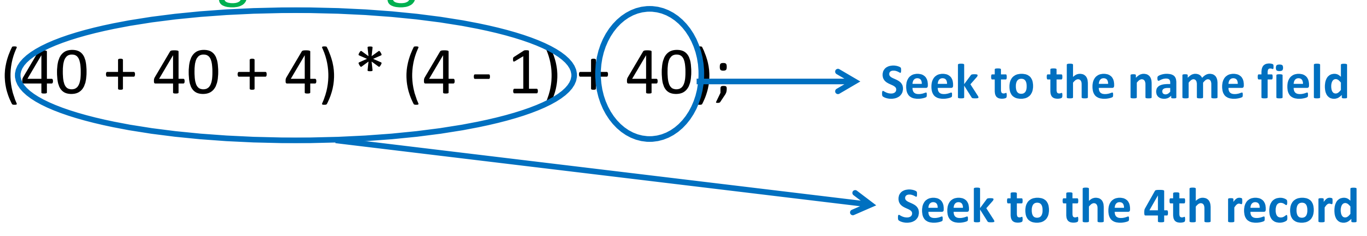
Program.java

```
    // let’s update the name of the 4th course

    // seek to the beginning of name of 4th course
    raf.seek((40 + 40 + 4) * (4 - 1) + 40);

    // write the new name of the course
    raf.writeBytes("Programlamaya Giris Laboratory II");

    // close the file
    raf.close();
} // end of main
```



Seek to the name field

Seek to the 4th record

Homework

- Investigate file opening modes in Java!
  - read, write, append, …

Serialization

- "**Serialization** is the process of translating data structures or object state into a format that can be stored (for example, in a file or memory buffer, or transmitted across a network connection link) and reconstructed (**deserialization**) later in the same or another computer environment" (*ref: wikipedia*)

- In Java, serialization is usually used to save/read objects to/from files using ObjectOutputStream and ObjectInputStream

- A class must implement **java.io.Serializable** interface to be serializable. It is a marker interface (has no methods to implement)

Serialization Rules in Java

- All primitive types are serializable.

- Transient fields (with transient modifier) are NOT serialized, (i.e., not saved or restored). A class that implements Serializable must mark transient fields of classes that do not support serialization (e.g., a file stream).

- Static fields (with static modifier) are Not serialized.

- If member variables of a serializable object reference to a non-serializable object, the code will compile but a RuntimeException will be thrown.

Example Program: save/read the students

```
public class Student
     implements java.io.Serializable {

    private int id;
    private String firstName;
    private String lastName;
    transient private String dummy;

    public Student(int id, String firstName,
            String lastName, String dummy) {
            this.id = id;
            this.firstName = firstName;
            this.lastName = lastName;
            this.dummy = dummy;
    }
```

```
    // getters and setters are written here

    public String toString() {
            return id + " - " +
                        firstName + " " +
                        lastName +  " " +
                        dummy;
     }

}
```

Program.java

```
public static void main(String[] args) throws Exception {
    // create students
    Student[] students = new Student[2];
    students[0] = new Student(20131234, "Ali", "Doğru", "dummy1");
    students[1] = new Student(20135678, "Veli", "Yanlış", "dummy2");
    // create the file
    ObjectOutputStream oos = new ObjectOutputStream(new
                    FileOutputStream(new File("students.dat")));
    for (int i = 0; i < students.length; i++) {
        oos.writeObject(students[i]); // write the object to file serializing
        System.out.println(students[i]);  // print the object
    }
    oos.close();              // close the file
```

Program.java (continued)

```
    // let’s read and display the saved objects on the screen

    // open the file
    ObjectInputStream ois = new ObjectInputStream(
                   new FileInputStream(new File("students.dat")));
    for (int i = 0; i < students.length; i++) {
        // read the student object from file deserializing
        Student s = (Student) ois.readObject();
        System.out.println(s);
    }
    ois.close();
}
```

Output of the program

```
Objects written to the file:
20131234 - Ali Doğru - dummy1
20135678 - Veli Yanlış - dummy2

Objects read from the file:
20131234 - Ali Doğru - null
20135678 - Veli Yanlış – null
```

Note that, transient field named dummy is not serialized. So, it is null when the objects are deserialized!

java.nio.*

- Be aware of a bit more complex library of Java: The "new" I/O
- It was introduced in JDK 1.4 in the java.nio.* packages
- It’s main goal is speed. It uses ***channels*** and ***buffers*** for I/O (closer to the operating system’s way of performing I/O)
- It supports a non-blocking I/O model.

Summary

- A stream is an object that either
  - Delivers data from your program to a destination, such as a file or screen, (output stream) or
  - Takes data from a source, such as a file or the keyboard, and delivers data to your program (input stream)
- Files are handled as text or binary files
- Java has classes to handle binary (byte oriented) or text (character oriented) files
- Decoration is used to give extra functionality to existing objects. Java I/O library benefits the decoration pattern
- Java supports both sequential and random file access
- Serialization is the job of converting an object to a bit stream that can be saved or transferred to be deserialized later
- Java's nio library is a fast option for I/O

Acknowledgements

- The course material used to prepare this presentation is partially taken/adopted from the list below:
  - Thinking in Java 4th Ed., Bruce Eckel, Prentice Hall, 2006
  - Java - An Introduction to Problem Solving and Programming, Walter Savitch, Pearson, 2012