

An abstract representation of file and directory pathnames.

Construction: `File(String pathname)`

Some useful methods:

```
boolean exists()
```

```
boolean createNewFile()
```

```
boolean delete()
```

```
long length()
```

Supports reading/writing to a random access file.

Construction: `RandomAccessFile(File file, String mode)`  
`RandomAccessFile(String name, String mode)`  
mode: `"r"` `"rw"` (`"rws"` `"rwd"`)

Logical view is that underlying file is a sequence (i.e., array) of bytes.

Maintains an internal *file pointer* to the current location within the file.

Reads/writes advance the file pointer.

Writes at the end of the file cause it to be extended.

```
public class rafExample {

    public static void main(String[] args) {
        try {
            long offset = 0;
            RandomAccessFile raf = new RandomAccessFile(args[0], "r");

            //Get the position of the first record (should be 0):
            offset = raf.getFilePointer();

            //Grab first line (first complete record):
            String record = raf.readLine();
            //Tell the world:
            System.out.println("The record offset is: " + offset);
            System.out.println("The record is: " + record);
            raf.close();
        } catch (FileNotFoundException e) {
            System.err.println("Could not find file: " + args[0]);
        } catch (IOException e) {
            System.err.println("Writing error: " + e);
        }
    }
}
```

A simple text scanner which can parse primitive types and strings using regular expressions.

A Scanner breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various next methods.

Construction:    `Scanner(InputStream source)`  
                  `Scanner(String source)`

Configuration:  `useDelimiter(String pattern)`

Some useful methods:

```
String next()
```

```
byte    nextByte()
```

```
int     nextInt()
```

```
. . .
```

```
boolean hasNext()
```

```
boolean hasNextByte()
```

```
boolean hasNextInt()
```

```
boolean hasNextLine()
```

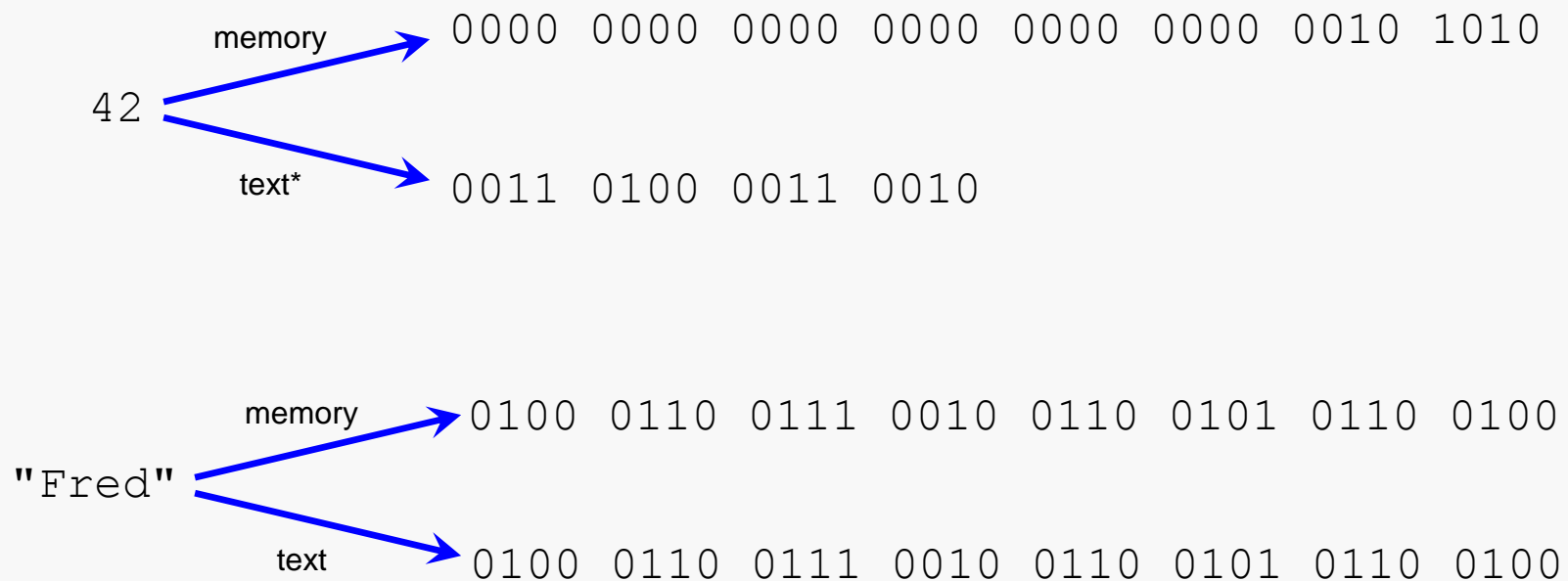
```
. . .
```

```
void close()
```

```
public class scannerExample {  
  
    public static void main(String[] args) {  
  
        String line = "foo\tbar\twidget";  
  
        Scanner s = new Scanner(line);  
        s.useDelimiter("\t");  
        String token1 = s.next();  
        String token2 = s.next();  
        String token3 = s.next();  
  
        System.out.println(token1 + " " + token2 + " " + token3);  
    }  
}
```

In a text file, data is represented using some sort of text-relevant encoding scheme, such as ASCII or Unicode.

In a binary file, data is represented in precisely the same way it would be represented in memory.



\*ASCII

Viewing data in pure binary form is often not very illuminating; here's a view of part of a binary file in hex form:

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000	2C	01	29	00	62	00	F9	00	A2	01	32	00	AA	00	6E	00
00001	DE	01	16	00	CD	00	97	00	B5	01	09	00	E5	00	C7	00
00002	9F	00	C4	00	FD	01	3A	01	22	00	D3	01	0E	01	10	00
00003	F3	00	88	00	82	01	3D	00	5B	00	71	00	5F	01	02	00
00004	A7	00	8D	01	37	00	9A	00	BD	00	DA	00	78	00	AF	01
00005	07	00	EB	01	1A	00	B2	00	68	00	00	03	68	69	6D	02
00006	61	73	05	74	68	65	72	65	05	42	75	72	6E	73	02	73
00007	6F	06	62	75	72	6E	65	64	09	77	61	6E	64	65	72	69
00008	6E	67	05	6E	65	27	65	72	04	68	61	74	68	09	66	6F
...																

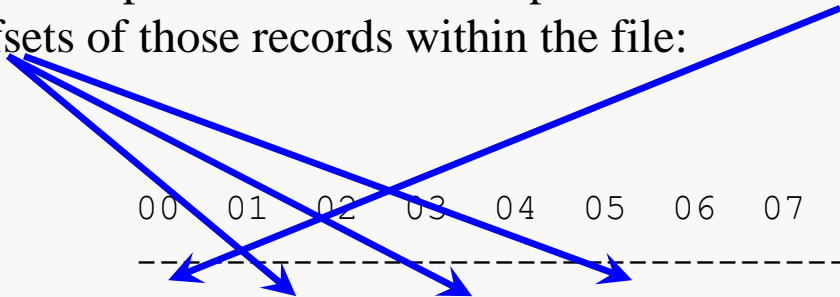
1 byte == 2 nybbles: 0010 1100

2 bytes == short int in big-endian order



The binary file shown before is logically divided into two parts.

The first part is a header that specifies the number of records in the file and then the offsets of those records within the file:

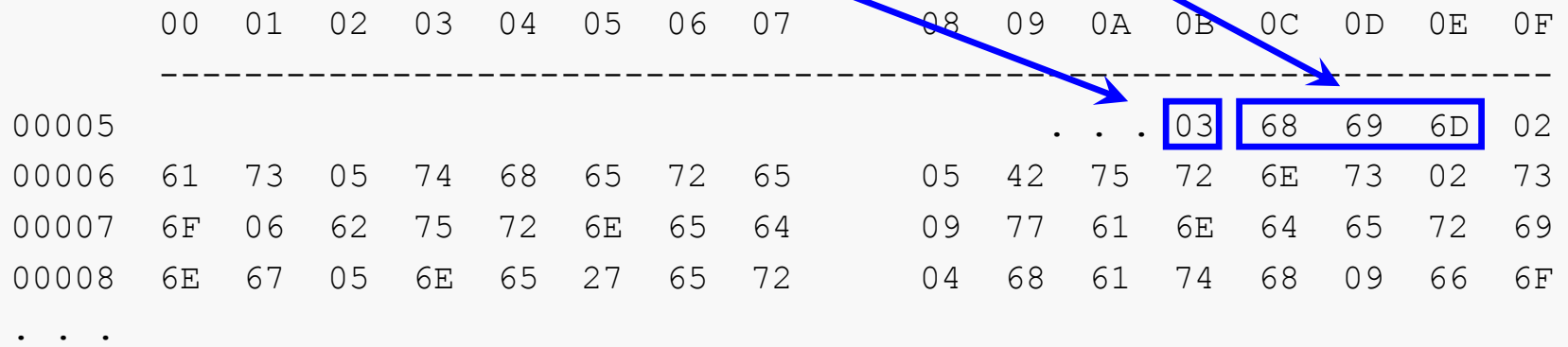
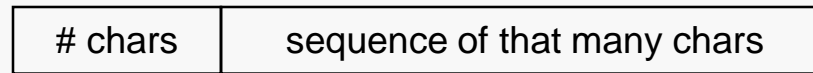


	00	01	02	03	04	05	06	07		08	09	0A	0B	0C	0D	0E	0F
00000	2C	01	29	00	62	00	F9	00		A2	01	32	00	AA	00	6E	00
00001	DF	01	16	00	CD	00	97	00		B5	01	09	00	E5	00	C7	00
00002	9F	00	C4	00	FD	01	3A	01		22	00	D3	01	0E	01	10	00
00003	F3	00	88	00	82	01	3D	00		5B	00	71	00	5F	01	02	00
00004	A7	00	8D	01	37	00	9A	00		BD	00	DA	00	78	00	AF	01
00005	07	00	EB	01	1A	00	B2	00		68	00	00	.	.	.		

The number of records is specified using a 1-byte integer value.

The offsets of those records are specified as 2-byte integer values.

The remainder of the file consists of a sequence of two-part records:



The number of characters is specified using a 1-byte integer value.

The characters are specified using ASCII codes.

The file actually contains a quotation. Each record specifies a "word" that makes up part of the quotation, and we must create the proper quotation from the file.

Logically, we need to:

- get the # of records in the file
- get the offset of the first record
- get the record from that offset and print it
- go back and get the offset of the second record
- get the second record and print it
- . . . until we've processed the specified number of records

```
public class FileReader {  
  
    public static void main(String[] args) {  
        try {  
            byte numWords = 0;  
            short wordOffset = 0;  
            byte wordLength = 0;  
  
            RandomAccessFile raf = new RandomAccessFile("1.Data.bin", "r");  
  
            // Get the number of words in the quotation:  
            numWords = raf.readByte();  
  
            System.out.println("The number of words is: " + numWords);  
  
            . . .  
        }  
    }  
}
```

```
...  
    for (int pos = 0; pos < numWords; pos++) {  
        // Get the offset of the first word:  
        wordOffset = raf.readShort();  
  
        // Save offset to return for next offset:  
        long offsetOfNextOffset = raf.getFilePointer();  
  
        // Go to that position.  
        raf.seek(wordOffset);  
  
        // Get the length of the word:  
        wordLength = raf.readByte();  
  
        // Get the word (in ASCII):  
        byte Word[] = new byte[wordLength];  
        raf.read(Word);  
  
        // Make Java happy:  
        String sWord = new String(Word);  
  
    ...
```

```
. . .

    // Report results:
    System.out.println("The next word is at offset:      " +
        wordOffset);
    System.out.println("The length of the next word is: " +
        wordLength);
    System.out.println("The next word is:              " + sWord);

    // Seek back to position of next offset:
    raf.seek(offsetOfNextOffset);
}
raf.close();

} catch (FileNotFoundException e) {
    System.err.println("This shouldn't happen: " + e);
} catch (IOException e) {
    System.err.println("Writing error: " + e);
}
}
}
```

```
The number of words is: 44
The next word is at offset:      297
The length of the next word is: 8
The next word is:                Breathes
The next word is at offset:      98
The length of the next word is: 5
The next word is:                there
The next word is at offset:      249
The length of the next word is: 3
The next word is:                the
The next word is at offset:      162
The length of the next word is: 4
The next word is:                man,
. . .
The next word is at offset:      178
The length of the next word is: 2
The next word is:                --
The next word is at offset:      104
The length of the next word is: 5
The next word is:                Burns
```

Java provides a number of operators for manipulating operands at the bit level:


```
&      bit-wise AND
|      bit-wise OR
^      bit-wise XOR
<<     shift bits to left 1
>>     shift bits to right 1, preserving sign
>>>   shift bits to right 1, shifting in 0
. . .
```



```
public class PrintBits
{
    public static void main( String args[] )
    {
        // get input integer
        Scanner scanner = new Scanner( System.in );
        System.out.println( "Please enter an integer:" );
        int input = scanner.nextInt();

        // display bit representation of an integer
        System.out.println( "\nThe integer in bits is:" );

        // create int value with 1 in leftmost bit and 0s elsewhere
        int displayMask = 1 << 31;
        . . .
        1000 0000 0000 0000 0000 0000 0000 0000
```



```
. . .
// for each bit display 0 or 1
for ( int bit = 1; bit <= 32; bit++ )
{
    // use displayMask to isolate bit
    System.out.print( ( input & displayMask ) == 0 ? '0' : '1' );

    input <<= 1; // shift value one position to left

    if ( bit % 8 == 0 )
        System.out.print( ' ' ); // display space every 8 bits
    } // end for
} // end main
} // end class PrintBits
```

mask off all but the high bit of input

move next bit into high position for next masking

Example adapted from Deitel JHTP7