In the Java API, an object from which we can read a sequence of bytes is called an **input stream**

An object to which we can write a sequence of bytes is called an **output stream**

When you have finished reading or writing to an input/output stream, close it by calling the **close** method. This call frees up the operating system resources that are in limited supply. If an application opens too many input/output streams without closing them, system resources can become depleted. Closing an output stream also flushes the buffer used for the output stream: Any bytes that were temporarily placed in a buffer so that they could be delivered as a larger packet are sent off. In particular, if you do not close a file, the last packet of bytes might never be delivered. You can also manually **flush** the output with the flush method.

**InputStream** and **OutputStream** classes let you read and write individual bytes and arrays of bytes

**DataInputStream** and **DataOutputStream** let you read and write all the primitive Java types in binary format

**ZipInputStream** and **ZipOutputStream** let you read and write files in the familiar ZIP compression format

For **Unicode** text, on the other hand, you can use subclasses of the abstract classes Reader and Writer

The read method returns either a UTF-16 code unit (as an integer between 0 and 65535) or -1

**FileInputStream** and **FileOutputStream** give you input and output streams attached to a disk file

When saving data, you have the choice between binary and text formats. For example, if the integer 1234 is saved in binary, it is written as the sequence of bytes 00 00 04 D2 (in hexadecimal notation). In text format, it is saved as the string "1234".

When saving text strings, you need to consider the **character encoding**

* UTF-16 encoding that Java uses internally, the string "José" is encoded as 00 4A 00 6F 00 73 00 E9 (in hex)
* UTF-8, the encoding most commonly used on the Internet, the string would be written as 4A 6F 73 C3 A9

**InputStreamReader** class turns an input stream that contains bytes (specifying characters in some character encoding) into a reader that emits Unicode code units.

The **OutputStreamWriter** class turns an output stream of Unicode code units into a stream of bytes

Java uses the Unicode standard for characters. Each character, or “code point,” has a 21-bit integer number. There are different character encodings—methods for packaging those 21-bit numbers into bytes

For text output, use a **PrintWriter**

to print to a file, construct a **PrintStream** from a file name and a character encoding

you can read a short text file into a string like this:

* var content = Files.readString(path, charset);
* List lines = Files.readAllLines(path, charset);
* …..

Read & Write binary data

* Data input & output interfaces

The **RandomAccessFile** class lets you read or write data anywhere in a file

In Java, you can use a **ZipInputStream** to read a ZIP archive

Java support object **serialization**, that makes it possible to write any object to an output stream and read it again later. The class must implement the **Serializable interface**

class Employee implements Serializable { . . . }

To save object data, you first need to open an ObjectOutputStream object

* var out = new ObjectOutputStream(new FileOutputStream("employee.dat"));

Now, to save an object, simply use the writeObject method

* var boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
* out.writeObject(boss);      //save object to byte

To read the objects back

* var in = new ObjectInputStream(new FileInputStream("employee.dat"));

Then, retrieve the objects in the same order in which they were written

* var e1 = (Employee) in.readObject();

You have to pay particular attention to serializing and deserializing objects that are assumed to be unique. This commonly happens when you are implementing **singletons**  a **typesafe enumerations**

Before you add a serialVersionUID field to a class, ask yourself why you made your class serializable. If serialization is used only for short-term persistence, such as distributed method calls in an application server, there is no need to worry about versioning and the serialVersionUID. The same applies if you extend a class that happens to be serializable, but you have no intent to ever persist its instances. If your IDE gives you pesky warnings, change the IDE preferences to turn them off, or add an annotation @SuppressWarnings("serial"). This is safer than adding a serialVersionUID that you may later forget to change

Creating files & Directories

* Files.createDirectory(path);
* Files.createDirectories(path); // Path must exist
* Files.createFile(path); // Create empty file

Copying, moving, deleting files

* Files.copy
* Files.move
* Files.delete

Getting file information

* exists, isHidden, etc…

Memory mapped files

* Most operating systems can take advantage of a virtual memory implementation to “map” a file, or a region of a file, into memory. Then the file can be accessed as if it were an in-memory array, which is much faster than the traditional file operations.

Buffer data structure

* When you use memory mapping, you make a single buffer that spans the entire file or the area of the file that you’re interested in
* A buffer is an array of values of the same type

File locking

* When multiple simultaneously executing programs need to modify the same file, they need to communicate in some way, or the file can easily become damaged. File locks can solve this problem. A file lock controls access to a file or a range of bytes within a file.

Regex