*7. Java I/O fundamentals*

*Working with class java.io.File*

* File is an abstract representation of a path to a file or a directory.
* You can use an object of class File to create a new file or directory, delete it, or inquire about or modify its attributes.
* A File instance might not be necessarily associated with an actual file or directory.
* File’s method isDirectory() returns true if the path it refers to is a directory.
* File’s method isFile() returns true if the path it refers to is a file.
* For a directory, File’s method list() returns an array of subdirectories and files.
* You can create a File instance that represents a non-existing file on your file system. And you can even invoke methods like isFile() without getting an exception.
* The objects of class File are immutable; the pathname represented by a File object can’t be changed.
* Methods createNewFile(), mkdir(), and mkdirs() can be used to create new files or directories.

*Using byte stream I/O*

* Class java.io.InputStream is an abstract base class for all the input streams.
* Class InputStream defines multiple overloaded versions of method read(), which can be used to read a single byte of data as int, or multiple bytes into a byte array.
* Method read() returns the next byte of data, or -1 if the end of the stream is reached. It doesn’t throw an EOFException.
* Method close() is another important method of class InputStream. Calling close() on a stream releases the system resources associated with it.
* Class java.io.OutputStream is an abstract class. It’s the base class for all the output streams in Java.
* The most important method of OutputStream class is write(), which can be used to write a single byte of data or multiple bytes from a byte array to a data destination.
* Class OutputStream also defines methods write(), flush(), and close(). So these are valid methods that can be called on any objects of classes that extend class OutputStream.
* All the classes that include OutputStream in their name—FileOutputStream, ObjectOutputStream, BufferedOutputStream, and DataOutputStream—extend abstract class OutputStream, directly or indirectly.
* To read and write raw bytes from and to a file, use FileInputStream and FileOutputStream.
* FileInputStream is instantiated by passing it a File instance or string value. It can’t be instantiated by passing it another InputStream.
* Instantiation of FileOutputStream creates a stream to write to a file specified either as a File instance or a string value. You can also pass a boolean value specifying whether to append to the existing file contents.
* Copying a file’s content might not copy its attributes. To copy a file, it’s advisable to use methods such as copy() from class java.nio.file.Files.
* I/O operations that require reading and writing of a single byte from and to a file are a costly affair. To optimize the operation, you can use a byte array.
* Unlike read(), read(byte[]) doesn’t return the read bytes. It returns the **count of bytes** read, or -1 if no more data can be read. The actual data is **read in the byte array** that’s passed to it as a method parameter.
* Method write(int) in class OutputStream writes a byte to the underlying output stream. If you write an int value by using this method, only the 8 low-order bits are written to the output stream; the rest are ignored.
* To buffer data with byte streams, you need classes BufferedInputStream and BufferedOutputStream.
* You can instantiate a BufferedInputStream by passing it an InputStream instance.
* A BufferedOutputStream can be instantiated by passing it an OutputStream instance.
* You can specify a buffer size or use the default size for both BufferedInputStream and BufferedOutputStream.
* To instantiate BufferedInputStream, you must pass it an object of InputStream. To instantiate BufferedOutputStream, you must pass it an object of OutputStream.
* You can use FileInputStream and FileOutputStream to read and write only byte data from and to an underlying file. These classes (FileInputStream and FileOutputStream) don’t define methods to work with any other specific primitive data types or objects.
* Data input and output streams let you read and write primitive values and strings from and to an underlying I/O stream in a machine-independent way. Data written with DataOutputStream can be read by DataInputStream.
* If a mismatch occurs in the type of data written by DataOutputStream and the type of data read by DataInputStream, you might not get a runtime exception. Because data streams read and write bytes, the read operation constructs the requested data from the available bytes, though incorrectly.
* An ObjectOutputStream can write primitive values and objects to an OutputStream, which can be read by an ObjectInputStream.
* To write objects to a file, their classes should implement java.io.Serializable, or the code will throw a java.io.NotSerializableException.
* If a class implements the Serializable interface, but its base class doesn’t, the class’s instance can be serialized.
* A class whose object fields don’t implement the Serializable interface can’t be serialized even though the class itself implements the Serializable interface. An attempt to serialize such object fields will throw a runtime exception.
* Retrieve the data (primitive and objects) in the order it was written using object streams, or it might throw a runtime exception.
* When you write objects to a file using ObjectOutputStream, its transient or static variables aren’t written to the file.

*Using character I/O with readers and writers*

* Reader and Writer are abstract base classes for reading and writing Unicode compliant character data.
* Classes Reader and Writer handle 16-bit Unicode well, which isn’t supported by the byte-oriented InputStream and OutputStream classes.
* Abstract class Reader defines overloaded read() methods to read character data from an underlying data stream.
* Class Reader implements Closeable (and its parent interface AutoCloseable). So Reader objects can be declared as resources with a try-with-resources statement.
* Compare the overloaded read() methods of class InputStream with the read() methods of class Reader. The read() methods of InputStream accept an array of byte as their method parameter, and the read() methods of Reader accept an array of char as their method parameter.
* Abstract class Writer defines overloaded write() methods to write character data to an underlying data source.
* With the overloaded write() methods of class Writer, you can write a single character or multiple characters stored in char arrays or string to a data source.
* FileReader and FileWriter are convenience classes for reading and writing character data from files.
* You can instantiate a FileReader by passing it the name of a file as a string value or as a File instance.
* You can instantiate a FileWriter by passing it the name of a file as a string value or as a File instance. You also have the option of specifying whether you want to override the existing content of a file or append new content to it by passing a boolean value to the constructor.
* To buffer data with character streams, you need classes BufferedReader and BufferedWriter.
* You can instantiate a BufferedReader by passing it a Reader instance.
* You can instantiate a BufferedWriter by passing it a Writer instance.
* You can also specify a buffer size or use the default size for both Buffered- Reader and BufferedWriter.
* Class PrintWriter can be used to print (write) formatted representations of objects to a file. This class implements all the print() methods found in class PrintStream.
* This essentially means that you can use all the overloaded print() methods that you’ve been using (via the class variable System.out) to write data to a file, a PrintWriter instance.

*Working with the console*

* Class java.io.Console defines methods to access the character-based console device associated with the current JVM.
* You may or may not be able to access the console associated with a JVM, depending on the underlying platform and how the JVM was started.
* If you invoke a JVM from the command line without redirecting the standard input and output streams, you’ll be able to access its console, which will typically be connected to the keyboard and display from which the virtual machine was launched.
* You may not be able to access the console associated with a JVM if it started automatically as a result of the execution of some other program.
* You will not get access to the console when using IDEs like Eclipse.
* You can access an object of class Console by calling System.console().
* If no console device is available, System.console() returns null. A null value signals that either the program was launched in a non-interactive environment or perhaps the underlying operating system doesn’t support the console operations.
* You can’t create an object of Console yourself. Class Console doesn’t define a public constructor.