*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.

*8. Java file I/O (NIO.2)*

***Path objects***

* Objects of the Path interface are used to represent the path of files or directories in a file system.
* Because a Path object might not be tied to a real file or directory on a system, it can refer to a nonexistent file or directory.
* Apart from referring to a file or a directory, a Path object can also refer to a symbolic link. A symbolic link is a special file that refers to another file.
* When you read data from or write data to a symbolic link, you read from or write to its underlying target file. But if you delete a symbolic link, the target file isn’t deleted.
* A Path can never be equal to a Path associated with another file system, even if they include exactly the same values.
* You can create Path objects by using Paths.get() or FileSystems.getDefault().getPath().
* You can convert a File instance to a Path object by calling toPath() on the File instance.
* Behind the scenes, both Paths.get() and File.toPath() call FileSystems.getDefault().getPath().
* Most of the Path methods perform syntactic operations. They manipulate the paths to a file or directory without accessing the file systems. They’re logical operations on paths in memory.
* Methods getName(), getNameCount(), and subpath() don’t use the root directory of a path. Method getRoot() returns the root of an absolute path and null for relative paths.
* The Path methods that accept positions throw an IllegalArgumentException at runtime for invalid positions. For example, getName() and subpath() throw an IllegalArgumentException if you pass invalid path positions to them.
* You can compare paths lexicographically using method compareTo(Path).
* To check whether a path starts or ends with another path, you can use startsWith(String), startsWith(Path), endsWith(String), and endsWith(Path).
* Methods startsWith() and endsWith() are overloaded—startsWith(String), startsWith(Path), endsWith(String), and endsWith(Path). So if you pass null to these methods, you’ll get a compiler error.
* The method name to retrieve the absolute path from a Path object is toAbsolutePath() and not getAbsolutePath().
* You can remove redundant path values by calling method normalize() on Path.
* Path is immutable and calling normalize() on a Path object doesn’t change its value.
* Method normalize() doesn’t check the actual file system to verify if the file (or directory) the resulting path is referring to actually exists.
* If a Path object includes redundancies like . or .., calling information retrieval methods like subpath() or getName() will include these redundancies in the returned values.
* The overloaded methods resolve(String) and resolve(Path) are used to join a relative path to another path. If you pass an absolute path as a parameter, this method returns the absolute path.
* To retrieve the path to a file in the same directory, say, to create its copy or to rename it, you can use the overloaded methods resolveSibling(String) and resolveSibling(Path).
* Method resolveSibling() resolves a given path against a path’s parent. If the given path is an absolute path, this method returns the absolute path. If you pass it an empty path, it returns the parent of the path.
* Methods resolve() and resolveSibling() don’t check the actual file system to verify if the file (or directory) the resulting path is referring to actually exists.
* To construct a path between two Path objects, use method relativize(). It can be used to construct a path between two relative or absolute Path objects.
* You can’t create a path from a relative path to an absolute path and vice versa using method relativize(). If you do so, you’ll get a runtime exception (IllegalArgumentException).
* Method relativize() doesn’t check the actual file system to verify if the file (or directory) the resulting path is referring to actually exists.

***Class Files***

* Class java.nio.file.Files defines static methods for manipulating files and directories.
* Method createFile() atomically checks for the existence of the file specified by the method parameter path and creates it if it doesn’t exist.
* Method createFile() fails and throws an exception if the file already exists, a directory with the same name exists, its parent directory doesn’t exist due to an I/O error, or the specified file attributes can’t be set.
* Method createDirectory() creates the specified directory (not the parent directory) on the file system. It also atomically checks for the existence of the specified directory and creates it if it doesn’t exist.
* Method createDirectory() throws an exception if a file or directory exists with the same name, its parent directory doesn’t exist due to an I/O error, or the specified directory attributes can’t be set.
* Method createDirectories() creates a directory, creating all nonexistent parent directories.
* If the target directory already exists, createDirectories() doesn’t throw any runtime exception. It throws an exception if the specified dir exists but isn’t a directory, an I/O occurs, or the specified directory attributes can’t be set.
* Specifying file or directory attributes is optional with methods createFile(), createDirectory(), and createDirectories(). All these methods declare to throw an IOException, which is a checked exception.
* You can check for the existence of a file or directory referred by a Path object using methods exists() and notExists().
* Method notExists() isn’t a complement of method exists(). It returns true if a target doesn’t exist or false if its existence can’t be determined. If these methods can’t determine the existence of a file, both of them will return false.
* Class Files’s overloaded method copy() enables you to read from InputStream and write to a Path object, read from a Path object and write to OutputStream, and read from and write to Path objects.
* Files.copy() can copy only files, not directories. If the source is a directory, then in the target an empty directory is created (without copying the entries in the directory). This method returns a long or Path value, not a boolean value.
* If you use a relative path to the target file, the file is created relative to your Java class file (.class) and not relative to the source file (passed as a parameter to method Files.copy()).
* To move files or directories programmatically, you can use Files.move(), which moves or renames a file to a target file.
* You can only move empty directories using method Files.move(). You can rename a nonempty directory by using Files.move(). But you can’t move a file or directory to a non-existing directory.
* To delete a directory or a file referred to by a Path object, you can use methods delete(Path) or deleteIfExists(Path).
* If you try to delete a directory that isn’t empty, methods delete(Path) and deleteIfExists(Path) will throw a DirectoryNotEmptyException.
* If you try to delete a nonexistent file or directory using method delete(), it will throw a NoSuchFileException. But method deleteIfExists() won’t throw an exception if the file or directory at the specified path doesn’t exist—rather, it will return false.
* Methods delete() and deleteIfExists() can be used to delete files and (nonempty) directories.

***Files and directory attributes***

* Class Files defines static methods to access individual attributes of a file or directory referred by a Path.
* You can access the individual attributes of a file or directory by using method Files.getAttribute(), passing to it the name of the attribute as a string value. To modify the attributes of an existing file or directory, you can use Files.setAttribute().
* You can access a group of file attributes by calling Files.getFileAttributeView() or Files.readAttributes().
* The BasicFileAttributes interface defines methods to access the basic attributes that should be supported by all the file systems.
* The BasicFileAttributeView interface can be used to modify the basic attributes.
* The DosFileAttributes interface extends BasicFileAttributes and defines methods to access attributes specific to Windows files and directories.
* The DosFileAttributeView interface defines methods to modify the DOS file attributes.
* The PosixFileAttributes interface also extends BasicFileAttributes and defines methods to access attributes related to the POSIX family of standards, like Linux or UNIX.
* The PosixFileAttributeView interface defines methods to modify attributes related to the POSIX family.
* Available only for Windows OS, the AclFileAttributeView interface supports access and updates of a file’s ACL.
* The FileOwnerAttributeView interface supports access and updates to the owner of a file or directory. It is supported by all systems that support the concept of file owners.
* The UserDefinedFileAttributeView interface supports the addition, modification, and deletion of user-defined metadata.
* The BasicFileAttributes, DosFileAttributes, and PosixFileAttributes interfaces define methods to access attributes. They don’t define methods to modify (or set) the attributes.
* The BasicFileAttributeView, DosFileAttributeView, PosixFileAttributeView, AclFileAttributeView, FileOwnerAttributeView, and UserDefinedFileAttributeView interfaces can be used to update attribute values.
* If a file system doesn’t support an attribute view, Files.getFileAttributeView() returns null. If a file system doesn’t support an attribute set, File.readAttributes() will throw a runtime exception.
* If an underlying system doesn’t support all the basic timestamps—that is, creationTime, lastAccessTime, and lastModifiedTime it might return system specific information.
* Methods Files.setAttribute() and Files.getAttribute() throw an IllegalArgumentException or UnsupportedOperationException if you pass them an invalid or unsupported attribute.
* The DosFileAttributes interface makes the following attributes available:
* archive
* hidden
* readonly
* system
* The DOS attributes are available on a Windows system only. Trying to access them on other systems will throw a runtime exception.
* When you read all DOS attributes using method Files.readAttributes(), you also read the basic attributes.
* The POSIX attributes are as follows:
* group
* owner
* permissions
* The POSIX attributes are available on the POSIX family of standards, such as UNIX and LINUX. Trying to access them on other systems will throw a runtime exception.
* To read or update the owner of a file or directory you can use the AclFileAttributeView, FileOwnerAttributeView, and PosixFileAttributeView interfaces.
* The UserDefinedAttributeView interface can be used to add, delete, access, and modify additional user-defined attributes to or from a file or directory. It defines methods delete(String), list(), read(String, ByteBuffer), size(String), and write(String, ByteBuffer) to, respectively, delete, list, read, get an attribute’s size, and write attribute values.

***Recursively access a directory tree***

* Class Files defines overloaded method walkFileTree() to walk recursively through the specified path. To define the traversal behavior, this method accepts an object of the FileVisitor interface.
* You can use the FileVisitor, a generic interface, to define the code that you want to execute during the traversal of a directory structure. When you traverse a directory structure, you can define what to do before or after you visit a directory, when you visit a file, or when access to a file is denied.
* Method postVisitDirectory() is invoked for a directory after entries in the directory and all of their descendants have been visited.
* Method preVisitDirectory() is invoked for a directory before entries in the directory are visited.
* Method visitFile() is invoked for a file in a directory.
* Method visitFileFailed() is invoked for a file that couldn’t be visited.
* Methods preVisitDirectory() and visitFile() are passed BasicFileAttributes of the path that they operate on. You can use these methods to query file or directory attributes.
* Class SimpleFileVisitor is a simple visitor of files with default behavior to visit all files and to rethrow I/O errors. It implements the FileVisitor interface.
* You can initiate traversal of a directory by calling the overloaded method walkFileTree() from class Files.
* The DirectoryStream interface can be used to iterate over all the files and directories in a directory. You can use an Iterator or for-each construct to iterate over a directory. The order in which the directory contents are iterated is unpredictable.
* If you pass Path to a file (and not a directory) to Files.newDirectoryStream(), it will throw a runtime exception. The order of iteration of files and directories in a specified directory using DirectoryStream is unpredictable.

***Using PathMatcher***

* You can match your file or directory names against a regex or glob pattern by using PathMatcher.
* A glob pattern supports a simpler form of pattern matching than the regex. It supports fewer special constructs.
* In glob, \* matches zero or more characters. In regex, .\* matches zero or more characters.
* To match a Path object with a pattern, you should create an object of java .nio.file.PathMatcher. PathMatcher is an interface with just one method: matches(). It returns true if a given path matches this matcher’s pattern.
* You can create a PathMatcher by calling FileSystem.getPathMatcher() and passing it the pattern to be matched.

***Watch a directory for changes***

* WatchService enables you to watch a directory for changes like addition, modification, or deletion of contents of a directory.
* The first step to watch a directory for changes is to create a WatchService object.
* A WatchService object watches a directory for the following events:
* StandardWatchEventKinds.ENTRY\_CREATE—This event occurs when a new file or directory is created, moved, or renamed in the directory being watched.
* StandardWatchEventKinds.ENTRY\_DELETE—This event occurs when an existing file or directory is deleted, moved, or renamed in the directory being watched.
* StandardWatchEventKinds.ENTRY\_MODIFY—This event is platform-dependent. It usually occurs when contents of an existing file are modified. It can also occur if the attributes of a file or directory (in the directory being watched) are modified.
* StandardWatchEventKinds.OVERFLOW—This indicates that an event has been lost.
* You can register multiple directories to be watched with the same WatchService object by using method register() of Path.
* You can watch a directory for changes. If you try to register a file for changes, you’ll get a runtime exception (NotDirectoryException). Registering a directory for any event (create, modify, or delete) doesn’t implicitly register its subdirectories.
* The WatchKey object is a token that represents the registration of a Watchable object with a WatchService. A WatchKey object is created when you register your directory to be watched for create, modify, or delete events with a WatchService.
* A WatchKey can be in multiple states:
* *Ready*—A WatchKey is initially created with a ready state.
* *Signaled*—When an event is detected, the WatchKey is signaled and queued. It can be retrieved using method WatchService’s poll() or take().
* *Cancelled*—Calling method cancel() on a WatchKey or closing the WatchService cancels a WatchKey.
* *Valid*—A WatchKey in a ready or signaled state is in a valid state.
* A WatchService queues the registered events when they occur. The registered consumers can retrieve the queued WatchKeys and process the corresponding events.
* The WatchService interface defines method take() and overloaded method poll() to retrieve the queued WatchKeys. Once a key is processed, the consumer invokes the key’s method reset() so that it can be signaled and required for further events.
* Method take() of the WatchService interface retrieves and removes the next WatchKey, waiting if none is yet present.
* Method poll() of the WatchService interface retrieves and removes the next WatchKey, or null if none is present (no waiting). You can also use its overloaded method poll(long timeout, TimeUnit unit) to specify the waiting time if none is present.
* For each retrieved WatchKey, you can call the WatchKey’s method pollEvents() to retrieve and remove all pending events for the key.
* Method pollEvents() returns a list of the events (WatchEvent) that were retrieved. Because you can register multiple paths and events with the same WatchService object, you can query the WatchEvent to determine the source of the event and its type, and process the event as required.