17

Java I/O Fundamentals and File I/O (NIO.2)

ORACLE

Objectives

After completing this lesson, you should be able to:

- Describe the basics of input and output in Java
- Read data from and write data to the console
- Use I/O streams to read and write files
- Read and write objects by using serialization
- Use the `Path` interface to operate on file and directory paths
- Use the `Files` class to check, delete, copy, or move a file or directory
- Use Stream API with NIO2

Java I/O Basics

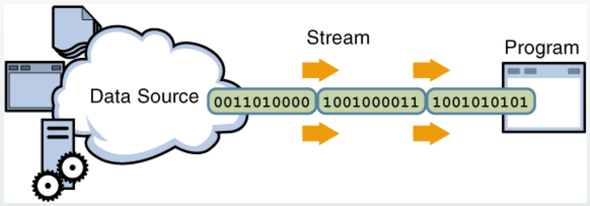
The Java programming language provides a comprehensive set of libraries to perform input/output (I/O) functions.

- Java defines an I/O channel as a stream.
- An I/O stream represents an input source or an output destination.
- An I/O stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.
- I/O streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects.

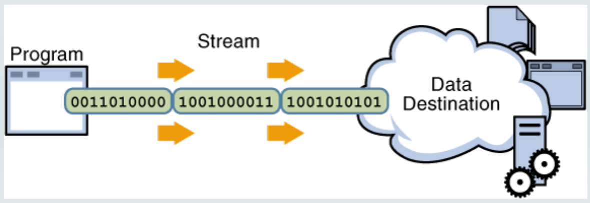
Some I/O streams simply pass on data; others manipulate and transform the data in useful ways.

I/O Streams

- A program uses an input stream to read data from a source, one item at a time.



- A program uses an output stream to write data to a destination (sink), one item at a time.



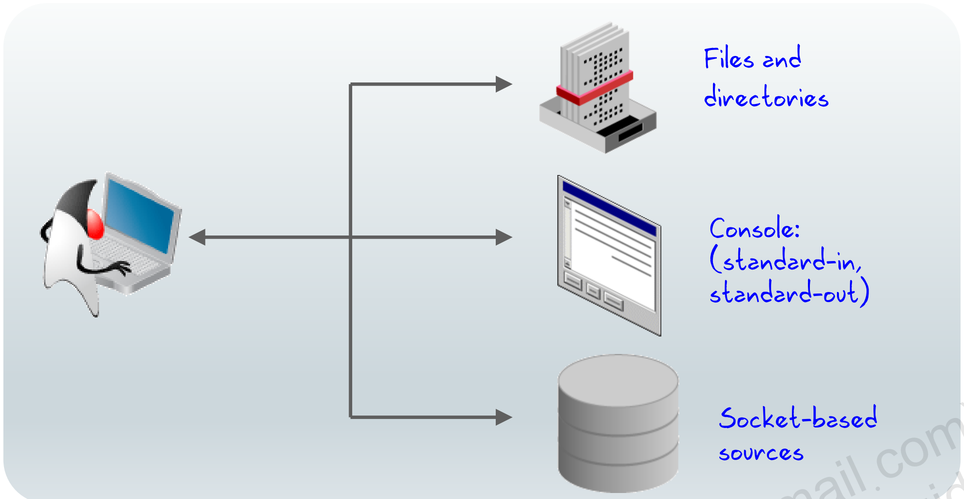
No matter how they work internally, all streams present the same simple model to programs that use them. A stream is a sequential flow of data. A stream can come from a source or can be generated to a sink.

- A source stream initiates the flow of data, also called an input stream.
- A sink stream terminates the flow of data, also called an output stream.

Sources and sinks are both node streams. Types of node streams are files, memory, and pipes between threads or processes.

I/O Application

Typically, a developer uses input and output in three ways:



An application developer typically uses I/O streams to read and write files, to read information from and write information to some output device, such as the keyboard (standard in) and the console (standard out). Finally, an application may need to use a socket to communicate with another application on a remote system.

Data Within Streams

- Java technology supports two types of streams: character and byte.
- Input and output of character data is handled by readers and writers.
- Input and output of byte data is handled by input streams and output streams:
  - Normally, the term *stream* refers to a byte stream.
  - The terms *reader* and *writer* refer to character streams.

Stream	Byte Streams	Character Streams
Source streams	InputStream	Reader
Sink streams	OutputStream	Writer

Java technology supports two types of data in streams: raw bytes and Unicode characters. Typically, the term *stream* refers to byte streams and the terms *reader* and *writer* refer to character streams.

More specifically, byte input streams are implemented by subclasses of the `InputStream` class, and byte output streams are implemented by subclasses of the `OutputStream` class. Character input streams are implemented by subclasses of the `Reader` class, and character output streams are implemented by subclasses of the `Writer` class.

Byte streams are best applied to reading and writing of raw bytes (such as image files, audio files, and objects). Specific subclasses provide methods to provide specific support for each of these stream types.

Character streams are designed for reading characters (such as in files and other character-based streams).

Byte Stream `InputStream` Methods

`InputStream` Methods:

The three basic read methods are:

```
int read()
int read(byte[] buffer)
int read(byte[] buffer, int offset, int length)
```

Other methods include:

```
void close();
//Close an open stream
int available();
// Number of bytes available
long skip(long n);
// Discard n bytes from stream
```

`OutputStream` Methods:

The three basic `write` methods are:

```
void write(int c)
void write(byte[] buffer)
void write(byte[] buffer, int offset, int length)
```

Other methods include:

```
void close();
// Automatically closed in try-with-resources
void flush();
// Force a write to the stream
```


`InputStream` Methods

The `read()` method returns an `int`, which contains either a byte read from the stream or a `-1`, which indicates the end-of-file condition. The other two read methods read the stream into a byte array and return the number of bytes read. The two `int` arguments in the third method indicate a subrange in the target array that needs to be filled.

Note: For efficiency, always read data in the largest practical block or use buffered streams.

When you have finished with a stream, close it. If you have a stack of streams, use filter streams to close the stream at the top of the stack. This operation also closes the lower streams.

`InputStream` implements `AutoCloseable`, which means that if you use an `InputStream` (or one of its subclasses) in a `try`-with-resources block, the stream is automatically closed at the end of the try.

The `available` method reports the number of bytes that are immediately available to be read from the stream. An actual read operation following this call might return more bytes.

The `skip` method discards the specified number of bytes from the stream.

Byte Stream: Example

```java
import java.io.FileInputStream; import java.io.FileOutputStream;
import java.io.FileNotFoundException; import java.io.IOException;

public class ByteStreamCopyTest {
    public static void main(String[] args) {
        byte[] b = new byte[128];
        // Example use of InputStream methods
        try (FileInputStream fis = new FileInputStream (args[0]);
             FileOutputStream fos = new FileOutputStream (args[1])) {
             System.out.println ("Bytes available: " + fis.available());
            int count = 0; int read = 0;
            while ((read = fis.read(b)) != -1) {
                 fos.write(b);
                count += read;
             }
             System.out.println ("Wrote: " + count);
         } catch (FileNotFoundException f) {
             System.out.println ("File not found: " + f);
         } catch (IOException e) {
             System.out.println ("IOException: " + e);
         }
    }
}
```

Note that you must keep track of how many bytes are read into the byte array each time.

This example copies one file to another by using a byte array. Note that `FileInputStream` and `FileOutputStream` are meant for streams of raw bytes, such as image files.

Note: The `available()` method, according to Javadocs, reports "an estimate of the number of remaining bytes that can be read (or skipped over) from this input stream without blocking."

Character Stream Methods

`Reader` Methods:

The three basic read methods are:

```
int read()
int read(char[] cbuf)
int read(char[] cbuf, int offset, int length)
```

Other methods include:

```
void close()
boolean ready()
long skip(long n)
boolean markSupported()
void mark(int readAheadLimit)
void reset()
```

`Writer` Methods:

The three basic `write` methods are:

```
void write(int c)
void write(char[] cbuf)
void write(char[] cbuf, int offset, int length)
void write(String string)
void write(String string, int offset, int length)
```

Other methods include:

```
void close();
// Automatically closed in try-with-resources
void flush();
// Force a write to the stream
```


`Reader` Methods

The first method returns an `int`, which contains either a Unicode character read from the stream or a `-1`, which indicates the end-of-file condition. The other two methods read into a character array and return the number of bytes read. The two `int` arguments in the third method indicate a subrange in the target array that needs to be filled.

`Writer` Methods

These methods are analogous to the `OutputStream` methods.

Character Stream: Example

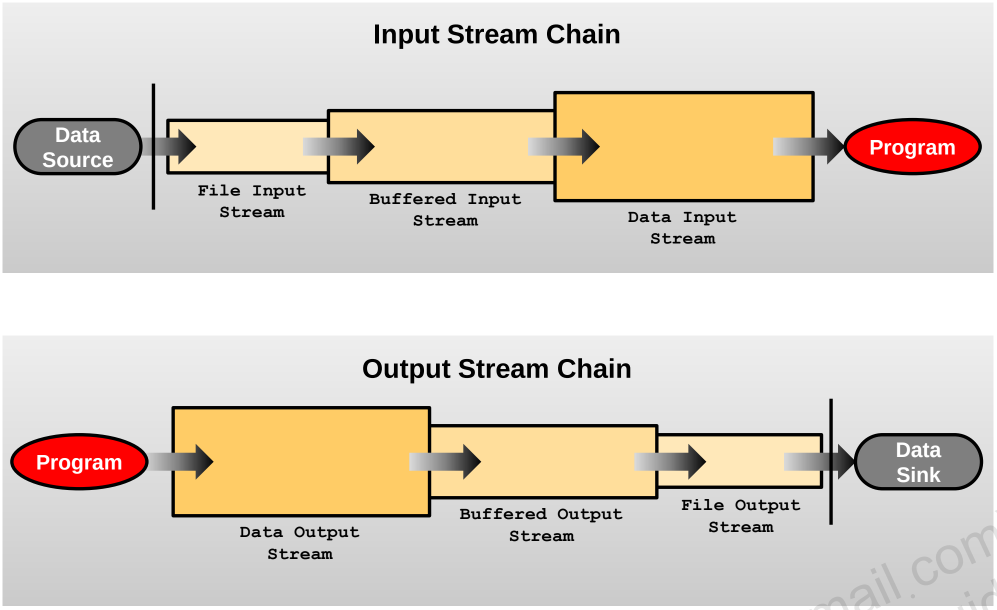
```
import java.io.FileReader; import java.io.FileWriter;
import java.io.IOException; import java.io.FileNotFoundException;

public class CharStreamCopyTest {
    public static void main(String[] args) {
        char[] c = new char[128];
        // Example use of InputStream methods
        try (FileReader fr = new FileReader(args[0]);
             FileWriter fw = new FileWriter(args[1])) {
             int count = 0;
             int read = 0;
             while ((read = fr.read(c)) != -1) {
                  fw.write(c);
                 count += read;
             }
             System.out.println("Wrote: " + count + " characters.");
         } catch (FileNotFoundException f) {
             System.out.println("File " + args[0] + " not found.");
         } catch (IOException e) {
             System.out.println("IOException: " + e);
         }
     }
}
```

Now, rather than a byte array, this version uses a character array.

Similar to the byte stream example, this example copies one file to another by using a character array instead of a byte array. `FileReader` and `FileWriter` are classes designed to read and write character streams, such as text files.

I/O Stream Chaining



A program rarely uses a single stream object. Instead, it chains a series of streams together to process the data. The first graphic in the slide demonstrates an example of input stream; in this case, a file stream is buffered for efficiency and then converted into data (Java primitives) items. The second graphic demonstrates an example of output stream; in this case, data is written, then buffered, and finally written to a file.

Chained Streams: Example

```
import java.io.BufferedReader; import java.io.BufferedWriter;
import java.io.FileReader; import java.io.FileWriter;
import java.io.FileNotFoundException; import java.io.IOException;

public class BufferedStreamCopyTest {
   public static void main(String[] args) {
       try (BufferedReader bufInput
                 = new BufferedReader(new FileReader(args[0]));
            BufferedWriter bufOutput
                 = new BufferedWriter(new FileWriter(args[1]))) {
            String line = "";
            while ((line = bufInput.readLine()) != null) {
                 bufOutput.write(line);
                 bufOutput.newLine();
            }
       } catch (FileNotFoundException f) {
            System.out.println("File not found: " + f);
       } catch (IOException e) {
            System.out.println("Exception: " + e);
       }
   }
}
```

A `FileReader` chained to a `BufferedFileReader`: This allows you to use a method that reads a String.

The character buffer replaced by a `String`. Note that `readLine()` uses the newline character as a terminator. Therefore, you must add that back to the output file.

The slide shows the copy application one more time. This version illustrates the use of a `BufferedReader` chained to the `BufferedFileReader` that you saw before.

The flow of this program is the same as before. Instead of reading a character buffer, this program reads a line at a time using the line variable to hold the `String` returned by the `readLine` method, which provides greater efficiency. The reason is that each read request made of a `Reader` causes a corresponding read request to be made of the underlying character or byte stream. A `BufferedReader` reads characters from the stream into a buffer. (The size of the buffer can be set, but the default value is generally sufficient.)

Console I/O

The `System` class in the `java.lang` package has three static instance fields: `out`, `in`, and `err`.

- The `System.out` field is a static instance of a `PrintStream` object that enables you to write to *standard output.*
- The `System.in` field is a static instance of an `InputStream` object that enables you to read from *standard input.*
- The `System.err` field is a static instance of a `PrintStream` object that enables you to write to *standard error.*

Console I/O Using System

- **`System.out`** is the "standard" output stream. This stream is already open and ready to accept output data. Typically, this stream corresponds to display output or another output destination specified by the host environment or user.
- **`System.in`** is the "standard" input stream. This stream is already open and ready to supply input data. Typically, this stream corresponds to keyboard input or another input source specified by the host environment or user.
- **`System.err`** is the "standard" error output stream. This stream is already open and ready to accept output data.
  Typically, this stream corresponds to display output or another output destination specified by the host environment or user. By convention, this output stream is used to display error messages or other information that should come to the immediate attention of a user even if the principal output stream, the value of the variable `out`, has been redirected to a file or other destination that is typically not continuously monitored.

Writing to Standard Output

- The `println` and `print` methods are part of the `java.io.PrintStream` class.
- The `println` methods print the argument and a newline character (`\n`).
- The `print` methods print the argument without a newline character.
- The `print` and `println` methods are overloaded for most primitive types `(boolean`, `char`, `int`, `long`, `float`, and `double)` and for `char[]`, `Object`, and `String`.
- The `print(Object)` and `println(Object)` methods call the `toString` method on the argument.

Reading from Standard Input

```
 7 public class KeyboardInput {
 8
 9     public static void main(String[] args) {
10         String s = "";
11         try (BufferedReader in = new BufferedReader(new
InputStreamReader(System.in))) {
12             System.out.print("Type xyz to exit: ");
13             s = in.readLine();
14             while (s != null) {
15                 System.out.println("Read: " + s.trim());
16                 if (s.equals("xyz")) {
17                     System.exit(0);
18                 }
19                 System.out.print("Type xyz to exit: ");
20                 s = in.readLine();
21             }
22         } catch (IOException e) { // Catch any IO exceptions.
23             System.out.println("Exception: " + e);
24         }
25     }
26 }
```

Chain a buffered reader to an input stream that takes the console input.

The `try`-with-resources statement on line 6 opens `BufferedReader`, which is chained to an `InputStreamReader`, which is chained to the static standard console input `System.in`.

If the string read is equal to "`xyz`," the program exits. The purpose of the `trim()` method on the String returned by `in.readLine` is to remove any whitespace characters.

Note: A null string is returned if an end of stream is reached (the result of a user pressing Ctrl + C in Windows, for example), thus the test for null on line 13.

Channel I/O

Introduced in JDK 1.4, a channel reads bytes and characters in blocks, rather than one byte or character at a time.

```
import java.io.FileInputStream; import java.io.FileOutputStream;
import java.nio.channels.FileChannel; import java.nio.ByteBuffer;
import java.io.FileNotFoundException; import java.io.IOException;

public class ByteChannelCopyTest {
    public static void main(String[] args) {
        try (FileChannel fcIn = new FileInputStream(args[0]).getChannel();
             FileChannel fcOut = new FileOutputStream(args[1]).getChannel()) {
            ByteBuffer buff = ByteBuffer.allocate((int) fcIn.size());
            fcIn.read(buff);
            buff.position(0);
            fcOut.write(buff);
        } catch (FileNotFoundException f) {
            System.out.println("File not found: " + f);
        } catch (IOException e) {
            System.out.println("IOException: " + e);
        }
    }
}
```

Create a buffer sized the same as the file size and then read and write the file in a single operation.

In this example, a file can be read in its entirety into a buffer and then written out in a single operation. Channel I/O was introduced in the `java.nio` package in JDK 1.4.

Persistence

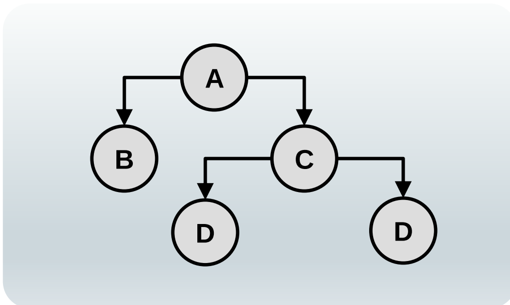
Saving data to some type of permanent storage is called persistence. An object that is persistent-capable can be stored on disk (or any other storage device) or sent to another machine to be stored there.

- A nonpersisted object exists only as long as the Java Virtual Machine is running.
- Java serialization is the standard mechanism for saving an object as a sequence of bytes that can later be rebuilt into a copy of the object.
- To serialize an object of a specific class, the class must implement the `java.io.Serializable` interface.

The `java.io.Serializable` interface defines no methods and serves only as a marker to indicate that the class should be considered for serialization.

Serialization and Object Graphs

- When an object is serialized, only the fields of the object are preserved.
- When a field references an object, the fields of the referenced object are also serialized, if that object's class is also serializable.
- The tree of an object’s fields constitutes the *object graph*.

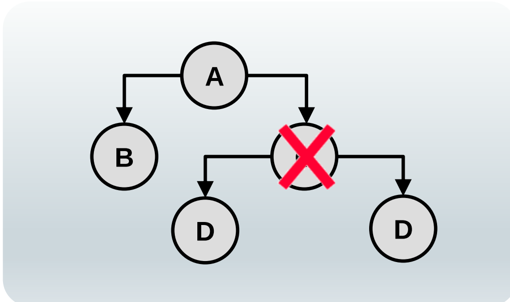


Object Graphs

Serialization traverses the object graph and writes that data to the file (or other output stream) for each node of the graph.

Transient Fields and Objects

- Some object classes are not serializable because they represent transient operating system–specific information.
- If the object graph contains a nonserializable reference, a `NotSerializableException` is thrown and the serialization operation fails.
- Fields that should not be serialized or that do not need to be serialized can be marked with the keyword `transient`.



Transient

If a field containing an object reference is encountered that is not marked as serializable (implement `java.io.Serializable`), a `NotSerializableException` is thrown and the entire serialization operation fails. To serialize a graph containing fields that reference objects that are not serializable, those fields must be marked using the keyword `transient`.

Transient: Example

```
public class Portfolio implements Serializable {
    public transient FileInputStream inputFile;
    public static int BASE = 100;
    private transient int totalValue = 10;
    protected Stock[] stocks;
}
```

`static` fields are not serialized.

Serialization includes all of the members of the `stocks` array.

- The field access modifier has no effect on the data field being serialized.
- The values stored in static fields are not serialized.
- When an object is deserialized, the values of static fields are set to the values declared in the class. The value of nonstatic transient fields is set to the default value for the type.

When an object is deserialized, the values of static and transient fields are set to the values defined in the class declaration. The values of nonstatic fields are set to the default value of their type. So in the example shown in the slide, the value of BASE will be 100, per the class declaration. The values of nonstatic transient fields, `inputFile` and `totalValue`, are set to their default values, `null` and `0`, respectively.

Serial Version UID

- During serialization, a version number, serialVersionUID, is used to associate the serialized output with the class used in the serialization process.
- After deserialization, the serialVersionUID is checked to verify that the classes loaded are compatible with the object being deserialized.
- If the receiver of a serialized object has loaded classes for that object with different serialVersionUID, deserialization will result in an `InvalidClassException`.
- A serializable class can declare its own serialVersionUID by explicitly declaring a field named `serialVersionUID` as a static final and of type long:

```
private static long serialVersionUID = 42L;
```

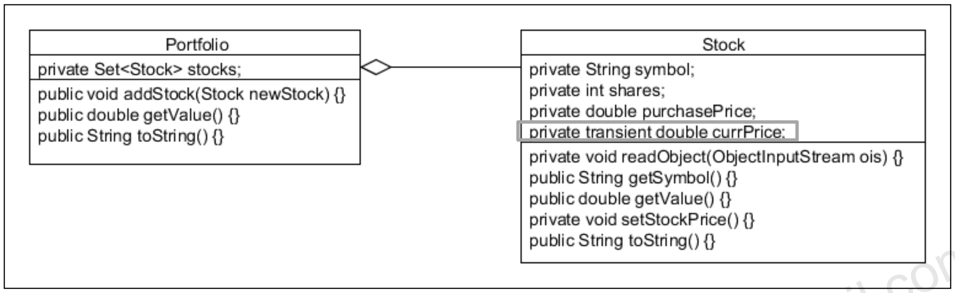

Note: The documentation for `java.io.Serializable` states the following:

> "If a serializable class does not explicitly declare a serialVersionUID, then the serialization run time will calculate a default serialVersionUID value for that class based on various aspects of the class, as described in the Java(TM) Object Serialization Specification. However, it is strongly recommended that all serializable classes explicitly declare serialVersionUID values, since the default serialVersionUID computation is highly sensitive to class details that may vary depending on compiler implementations and can thus result in unexpected InvalidClassExceptions during deserialization. Therefore, to guarantee a consistent serialVersionUID value across different Java compiler implementations, a serializable class must declare an explicit serialVersionUID value. It is also strongly advised that explicit serialVersionUID declarations use the private modifier where possible, since such declarations apply only to the immediately declaring class--serialVersionUID fields are not useful as inherited members. Array classes cannot declare an explicit serialVersionUID, so they always have the default computed value, but the requirement for matching serialVersionUID values is waived for array classes."

Serialization: Example

In this example, a Portfolio is made up of a set of Stocks.

- During serialization, the current price is not serialized and is, therefore, marked `transient`.
- However, the current value of the stock should be set to the current market price after deserialization.



Writing and Reading an Object Stream

```
1 public static void main(String[] args) {
2     Stock s1 = new Stock("ORCL", 100, 32.50);
3     Stock s2 = new Stock("APPL", 100, 245);
4     Stock s3 = new Stock("GOOG", 100, 54.67);
5     Portfolio p = new   Portfolio(s1, s2, s3);
6     try (FileOutputStream fos = new FileOutputStream(args[0]);
7          ObjectOutputStream out = new ObjectOutputStream(fos)) {
8         out.writeObject(p);
9     } catch (IOException i) {
10         System.out.println("Exception writing out Portfolio: " + i);
11     }
12     try (FileInputStream fis = new FileInputStream(args[0]);
13          ObjectInputStream in = new ObjectInputStream(fis)) {
14         Portfolio newP = (Portfolio)in.readObject();
15     } catch (ClassNotFoundException | IOException i) {
16         System.out.println("Exception reading in Portfolio: " + i);
17 }
```

`Portfolio` is the root object.

The `writeObject` method writes the object graph of `p` to the file stream.

The `readObject` method restores the object from the file stream.

The `SerializeStock` class.

- **Line 6 – 8:** A `FileOutputStream` is chained to an `ObjectOutputStream`. This allows the raw bytes generated by the `ObjectOutputStream` to be written to a file through the `writeObject` method. This method walks the object's graph and writes the data contained in the nontransient and nonstatic fields as raw bytes.
- **Line 12 – 14:** To restore an object from a file, a `FileInputStream` is chained to an `ObjectInputStream`. The raw bytes read by the `readObject` method restore an `Object` containing the nonstatic and nontransient data fields. This `Object` must be cast to expected type.

Serialization Methods

An object being serialized (and deserialized) can control the serialization of its own fields.

```
public class MyClass implements Serializable {
    // Fields
    private void writeObject(ObjectOutputStream oos) throws IOException {
        oos.defaultWriteObject();
        // Write/save additional fields
        oos.writeObject(new java.util.Date());
    }
}
```

`defaultWriteObject` is called to perform the serialization of this class' fields.

- For example, in this class, the current time is written into the object graph.
- During deserialization, a similar method is invoked:

```
private void readObject(ObjectInputStream ois) throws ClassNotFoundException, IOException {}
```


The `writeObject` method is invoked on the object being serialized. If the object does not contain this method, the `defaultWriteObject` method is invoked instead.

- This method must also be called once and only once from the object's `writeObject` method.

During deserialization, the `readObject` method is invoked on the object being deserialized (if present in the class file of the object). The signature of the method is important.

```
private void readObject(ObjectInputStream ois) throws
                              ClassNotFoundException, IOException {
    ois.defaultReadObject();
    // Print the date this object was serialized
    System.out.println ("Restored from date: " +
                               (java.util.Date)ois.readObject()));
}
```

readObject: Example

```
public class Stock implements Serializable {
    private static final long serialVersionUID = 100L;
    private String symbol;
    private int shares;
    private double purchasePrice;
    private transient double currPrice;

    public Stock(String symbol, int shares, double purchasePrice) {
        this.symbol = symbol;
        this.shares = shares;
        this.purchasePrice = purchasePrice;
        setStockPrice();
    }

    // This method is called post-serialization
    private void readObject(ObjectInputStream ois)
                        throws IOException, ClassNotFoundException {
        ois.defaultReadObject();
        // perform other initialization
        setStockPrice();
    }
}
```

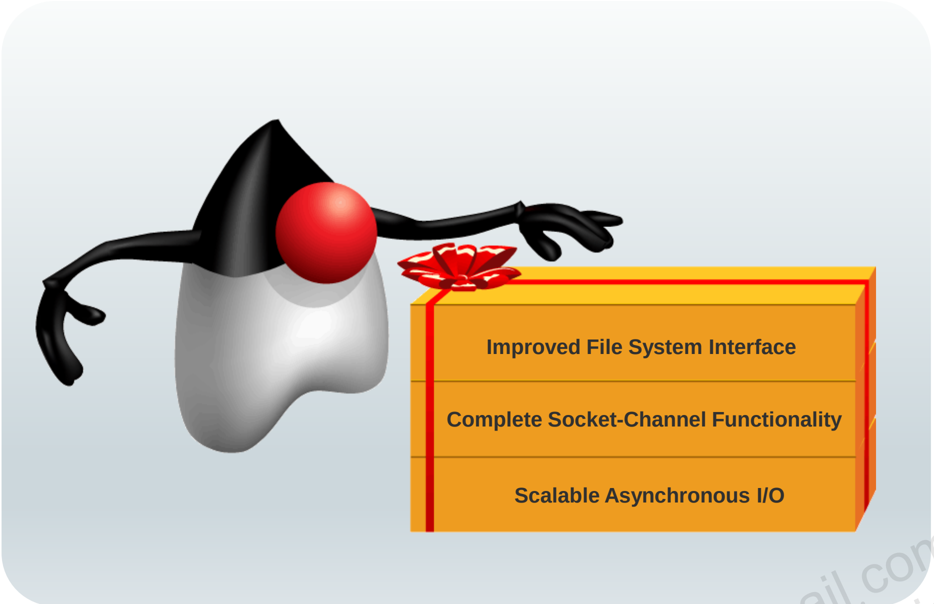
Stock currPrice is set by the setStockPrice method during creation of the Stock object, but the constructor is not called during deserialization.

Stock currPrice is set after the other fields are deserialized.

In the Stock class, the `readObject` method is provided to ensure that the stock's `currPrice` is set (by the `setStockPrice` method) after deserialization of the Stock object.

Note: The signature of the `readObject` method is critical for this method to be called during deserialization.

New File I/O API (NIO.2)



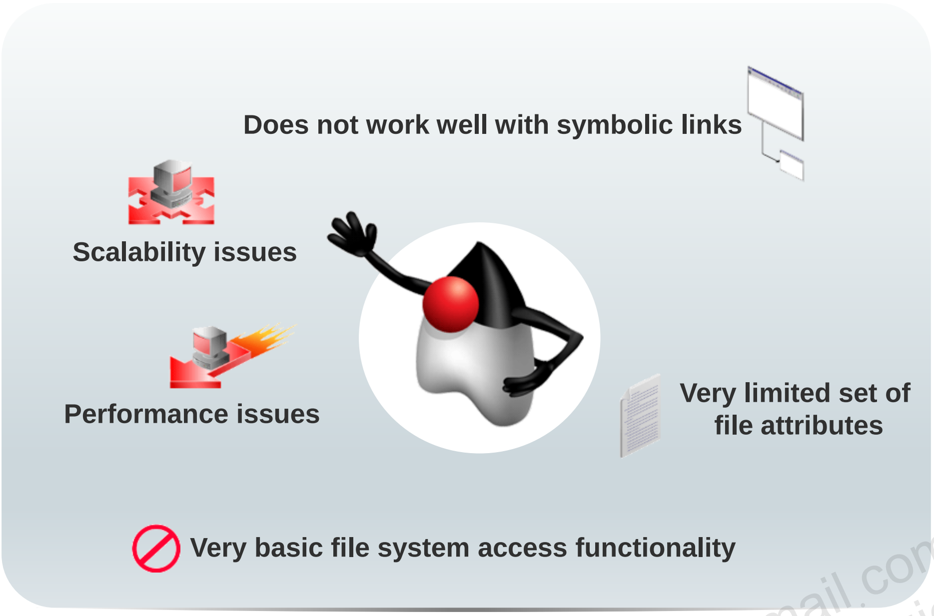
NIO API in JSR 51 established the basis for NIO in Java, focusing on buffers, channels, and charsets. JSR 51 delivered the first piece of the scalable socket I/Os into the platform, providing a nonblocking, multiplexed I/O API, thus allowing the development of highly scalable servers without having to resort to native code.

For many developers, the most significant goal of JSR 203 is to address issues with `java.io.File` by developing a new file system interface.

The new API:

- Works more consistently across platforms
- Makes it easier to write programs that gracefully handle the failure of file system operations
- Provides more efficient access to a larger set of file attributes
- Allows developers of sophisticated applications to take advantage of platform-specific features when absolutely necessary
- Allows support for non-native file systems, to be “plugged in” to the platform

Limitations of `java.io.File`



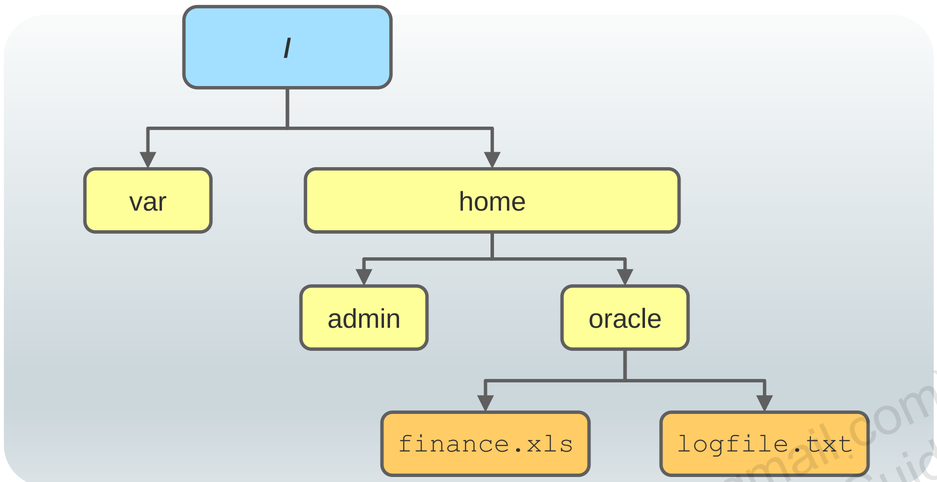
The Java I/O File API (`java.io.File`) presented challenges for developers.

- Many methods did not throw exceptions when they failed, so it was impossible to obtain a useful error message.
- Several operations were missing (file copy, move, and so on).
- The rename method did not work consistently across platforms.
- There was no real support for symbolic links.
- More support for metadata was desired, such as file permissions, file owner, and other security attributes.
- Accessing file metadata was inefficient—every call for metadata resulted in a system call, which made the operations very inefficient.
- Many of the File methods did not scale. Requesting a large directory listing on a server could result in a hang.
- It was not possible to write reliable code that could recursively walk a file tree and respond appropriately if there were circular symbolic links.

Further, the overall I/O was not written to be extended. Developers had requested the ability to develop their own file system implementations, for example, by keeping a pseudofile system in memory or by formatting files as zip files.

File Systems, Paths, Files

In NIO.2, both files and directories are represented by a path, which is the relative or absolute location of the file or directory.



File Systems

Prior to the NIO.2 implementation in JDK 7, files were represented by the `java.io.File` class.

In NIO.2, instances of `java.nio.file.Path` objects are used to represent the relative or absolute location of a file or directory.

File systems are hierarchical (tree) structures. File systems can have one or more root directories. For example, typical Windows machines have at least two disk root nodes: C:\ and D:\.

Note that file systems may also have different characteristics for path separators, as shown in the slide.

Relative Path Versus Absolute Path

- A path is either *relative* or *absolute*.
- An absolute path always contains the root element and the complete directory list required to locate the file.
- Example:

```
...
/home/peter/statusReport
...
```

- A relative path must be combined with another path in order to access a file.
- Example:

```
...
clarence/foo
...
```


A path can either be relative or absolute. An absolute path always contains the root element and the complete directory list required to locate the file. For example, `/home/peter/statusReport` is an absolute path. All the information needed to locate the file is contained in the path string.

A relative path must be combined with another path in order to access a file. For example, `clarence/foo` is a relative path. Without more information, a program cannot reliably locate the `clarence/foo` directory in the file system.

Java NIO.2 Concepts

Prior to JDK 7, the `java.io.File` class was the entry point for all file and directory operations. With NIO.2, there is a new package and classes:

- `java.nio.file.Path`: Locates a file or a directory by using a system-dependent path
- `java.nio.file.Files`: Using a `Path`, performs operations on files and directories
- `java.nio.file.FileSystem`: Provides an interface to a file system and a factory for creating a `Path` and other objects that access a file system
- All the methods that access the file system throw `IOException` or a subclass.

Java NIO.2

A significant difference between NIO.2 and `java.io.File` is the architecture of access to the file system. With the `java.io.File` class, the methods used to manipulate path information are in the same class with methods used to read and write files and directories.

In NIO.2, the two concerns are separated. Paths are created and manipulated using the `Path` interface, while operations on files and directories are the responsibility of the `Files` class, which operates only on `Path` objects.

Finally, unlike `java.io.File`, `Files` class methods that operate directly on the file system throw an `IOException` (or a subclass). Subclasses provide details on what the cause of the exception was.

`Path` Interface

- The `java.nio.file.Path` interface provides the entry point for the NIO.2 file and directory manipulation.

```
FileSystem fs = FileSystems.getDefault();
Path p1 = fs.getPath ("/home/oracle/labs/resources/myFile.txt");
```

- To obtain a `Path` object, obtain an instance of the default file system and then invoke the `getPath` method:

```
Path p1 = Paths.get("/home/oracle/labs/resources/myFile.txt");
Path p2 = Paths.get("/home/oracle", "labs", "resources", "myFile.txt");
```

`Path` Interface Features

The `Path` interface defines the methods used to locate a file or a directory in a file system. These methods include:

- To access the components of a path:
  - `getFileName`, `getParent`, `getRoot`, `getNameCount`
- To operate on a path:
  - `normalize`, `toUri`, `toAbsolutePath`, `subpath`, `resolve`, `relativize`
- To compare paths:
  - `startsWith`, `endsWith`, `equals`

`Path` Objects Are Like `String` Objects

It is best to think of `Path` objects in the same way you think of `String` objects. `Path` objects can be created from a single text string or a set of components:

- A *root component*, which identifies the file system hierarchy
- A *name element*, farthest from the root element, that defines the file or directory the path points to
- Additional elements may be present as well, separated by a special character or delimiter that identify directory names that are part of the hierarchy.

`Path` objects are immutable. Once created, operations on `Path` objects return new `Path` objects.

Path: Example

```
public class PathTest
    public static void main(String[] args) {
        Path p1 = Paths.get(args[0]);
        System.out.format("getFileName: %s%n", p1.getFileName());
        System.out.format("getParent: %s%n", p1.getParent());
        System.out.format("getNameCount: %d%n", p1.getNameCount());
        System.out.format("getRoot: %s%n", p1.getRoot());
        System.out.format("isAbsolute: %b%n", p1.isAbsolute());
        System.out.format("toAbsolutePath: %s%n", p1.toAbsolutePath());
        System.out.format("toURI: %s%n", p1.toUri());
    }
}
```

```
java PathTest /home/oracle/file1.txt
getFileName: file1.txt
getParent: /home/oracle
getNameCount: 3
getRoot: /
isAbsolute: true
toAbsolutePath: /home/oracle/file1.txt
toURI: file:///home/oracle/file1.txt
```


Unlike the `java.io.File` class, files and directories are represented by instances of `Path` objects in a *system-dependent* way.

The `Path` interface provides several methods for reporting information about the path:

- `Path getFileName`: The end point of this `Path`, returned as a `Path` object
- `Path getParent`: The parent path or null. Everything in `Path` up to the file name (file or directory)
- `int getNameCount`: The number of name elements that make up this path
- `Path getRoot`: The root component of this `Path`
- `boolean isAbsolute`: `true` if this path contains a system-dependent root element. **Note:** Because this example is being run on a Windows machine, the *system-dependent* root element contains a drive letter and colon. On a UNIX-based OS, `isAbsolute` returns true for any path that begins with a slash.
- `Path toAbsolutePath`: Returns a path representing the absolute path of this path
- `java.net.URI toUri`: Returns an absolute URI

Note: A `Path` object can be created for any path. The actual file or directory need not exist.

Removing Redundancies from a `Path`

- Many file systems use "`.`" notation to denote the current directory and "`..`" to denote the parent directory.
- The following examples both include redundancies:

```
/home/./clarence/foo
/home/peter/../clarence/foo
```

- The `normalize` method removes any redundant elements, which includes any "`.`" or "`directory/..`" occurrences.
- Example:

```
Path p = Paths.get("/home/peter/../clarence/foo");
Path normalizedPath = p.normalize();
```

```
/home/clarence/foo
```


Many file systems use "`.`" notation to denote the current directory and "`..`" to denote the parent directory. You might have a situation where a `Path` contains redundant directory information. Perhaps a server is configured to save its log files in the "`/dir/logs/.`" directory, and you want to delete the trailing "`/.`" notation from the path.

The `normalize` method removes any redundant elements, which includes any "`.`" or "`directory/..`" occurrences. The slide examples would be normalized to `/home/clarence/foo`.

It is important to note that `normalize` does not check the file system when it cleans up a path. It is a purely syntactic operation. In the second example, if `peter` were a symbolic link, removing `peter/..` might result in a path that no longer locates the intended file.

Creating a Subpath

- A portion of a path can be obtained by creating a subpath using the `subpath` method:

```
Path subpath(int beginIndex, int endIndex);
```

- The element returned by `endIndex` is one less than the `endIndex` value.
- Example:

home= 0
oracle = 1
Temp = 2

```
Path p1 = Paths.get ("/home/oracle/Temp/foo/bar");
Path p2 = p1.subpath (1, 3);
```

```
oracle/Temp
```

Include the element at index 2.

The element name closest to the root has index 0.

The element farthest from the root has index `count-1`.

Note: The returned `Path` object has the name elements that begin at `beginIndex` and extend to the element at index `endIndex-1`.

Joining Two Paths

- The `resolve` method is used to combine two paths.
- Example:

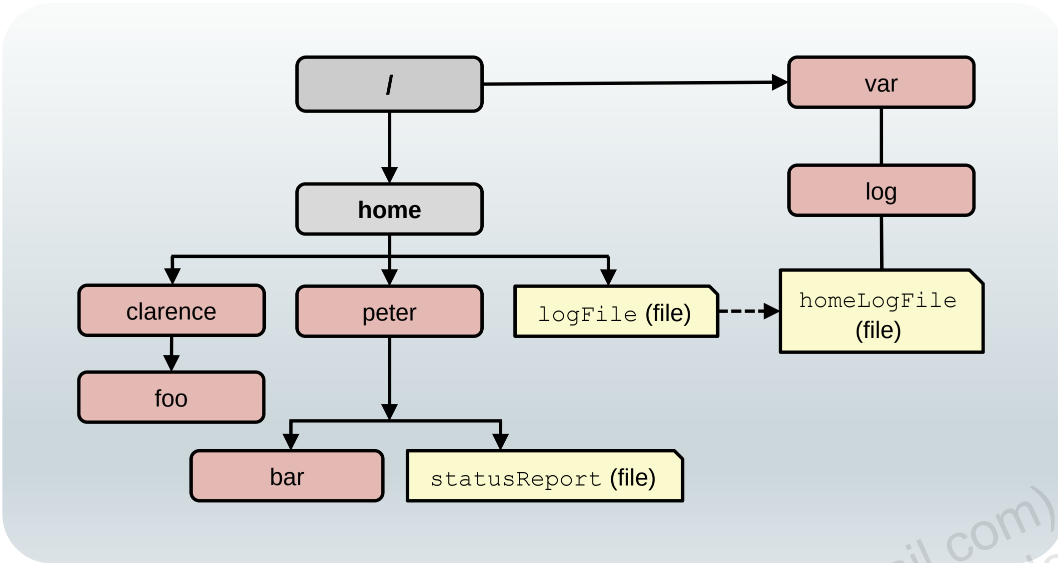
```
Path p1 = Paths.get("/home/clarence/foo");
p1.resolve("bar");     // Returns /home/clarence/foo/bar
```

- Passing an absolute path to the `resolve` method returns the passed-in path.

```
Paths.get("foo").resolve("/home/clarence"); // Returns /home/clarence
```


The `resolve` method is used to combine paths. It accepts a partial path, which is a path that does not include a root element, and that partial path is appended to the original path.

Symbolic Links



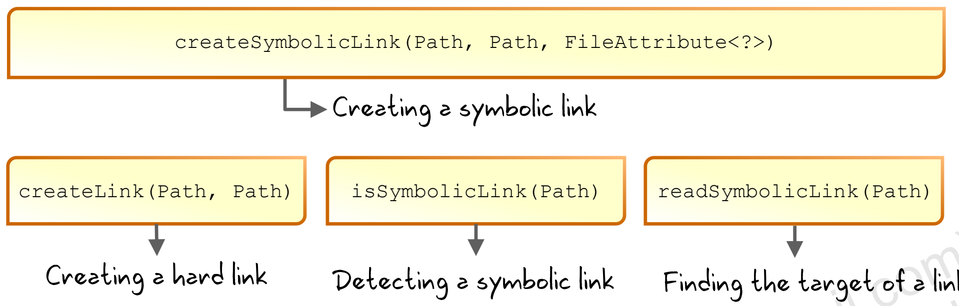
File system objects are most typically directories or files. Everyone is familiar with these objects. But some file systems also support the notion of symbolic links. A symbolic link is also referred to as a “symlink” or a “soft link.”

A symbolic link is a special file that serves as a reference to another file. A symbolic link is usually transparent to the user. Reading or writing to a symbolic link is the same as reading or writing to any other file or directory.

In the slide’s diagram, `logFile` appears to the user to be a regular file, but it is actually a symbolic link to `/var/log/homeLogFile`. `homeLogFile` is the target of the link.

Working with Links

- `Path` interface is “link aware.”
- Every `Path` method either:
  – Detects what to do when a symbolic link is encountered or
  – Provides an option enabling you to configure the behavior when a symbolic link is encountered



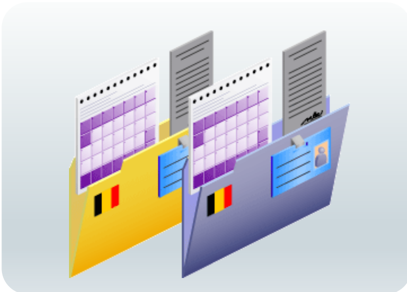
The `java.nio.file` package and the `Path` interface in particular are “link aware.” Every `Path` method either detects what to do when a symbolic link is encountered, or it provides an option enabling you to configure the behavior when a symbolic link is encountered.

Some file systems also support hard links. Hard links are more restrictive than symbolic links, as follows:

- The target of the link must exist.
- Hard links are generally not allowed on directories.
- Hard links are not allowed to cross partitions or volumes. Therefore, they cannot exist across file systems.
- A hard link looks, and behaves, like a regular file, so they can be hard to find.
- A hard link is, for all intents and purposes, the same entity as the original file. They have the same file permissions, time stamps, and so on. All attributes are identical.

Because of these restrictions, hard links are not used as often as symbolic links, but the `Path` methods work seamlessly with hard links.

`File` Operations



- Checking a File or Directory
- Deleting a File or Directory
- Copying a File or Directory
- Moving a File or Directory
- Managing Metadata
- Reading, Writing, and Creating Files
- Random Access Files
- Creating and Reading Directories

The `java.nio.file.Files` class is the primary entry point for operations on `Path` objects.

Static methods in this class read, write, and manipulate files and directories represented by `Path` objects.

The `Files` class is also link aware—methods detect symbolic links in `Path` objects and automatically manage links or provide options for dealing with links.

Please refer to the examples provided for this lesson to implement the following operations using File:

Checking a File or Directory

Deleting a File

Copying a File

Moving a File

Managing Metadata

BufferedReader File Stream

The new `lines()` method converts a `BufferedReader` into a stream.

```
public class BufferedRead {
  public static void main(String[] args) {
    try(BufferedReader bReader =
      new BufferedReader(new FileReader("tempest.txt"))){

      bReader.lines()
        .forEach(line ->
            System.out.println("Line: " + line));

    } catch (IOException e){
        System.out.println("Message: " + e.getMessage());
    }
  }
}
```

NIO File Stream

The `lines()` method can be called using NIO classes.

```
public class ReadNio {

  public static void main(String[] args) {

    try(Stream<String> lines =
        Files.lines(Paths.get("tempest.txt"))){

      lines.forEach(line ->
        System.out.println("Line: " + line));

    } catch (IOException e){
      System.out.println("Error: " + e.getMessage());
    }
  }
}
```

Read File into `ArrayList`

Use `readAllLines()` to load a file into an `ArrayList`.

```
public class ReadAllNio {
  public static void main(String[] args) {
    Path file = Paths.get("tempest.txt");
    List<String> fileArr;

    try{

      fileArr = Files.readAllLines(file);

      fileArr.stream()
        .filter(line -> line.contains("PROSPERO"))
        .forEach(line -> System.out.println(line));

    } catch (IOException e){
      System.out.println("Message: " + e.getMessage());
    }
  }
}
```

Managing Metadata

Method	Explanation
size	Returns the size of the specified file in bytes
isDirectory	Returns true if the specified Path locates a file that is a directory
isRegularFile	Returns true if the specified Path locates a file that is a regular file
isSymbolicLink	Returns true if the specified Path locates a file that is a symbolic link
isHidden	Returns true if the specified Path locates a file that is considered hidden by the file system
getLastModifiedTime setLastModifiedTime	Returns or sets the specified file's last modified time
getAttribute setAttribute	Returns or sets the value of a file attribute

If a program needs multiple file attributes around the same time, it can be inefficient to use methods that retrieve a single attribute. Repeatedly accessing the file system to retrieve a single attribute can adversely affect performance. For this reason, the `Files` class provides two `readAttributes` methods to fetch a file's attributes in one bulk operation.

- `readAttributes(Path, String, LinkOption...)`
- `readAttributes(Path, Class<A>, LinkOption...)`

Summary

In this lesson, you should have learned how to:

- Describe the basics of input and output in Java
- Read data from and write data to the console
- Use I/O streams to read and write files
- Read and write objects by using serialization
- Use the `Path` interface to operate on file and directory paths
- Use the `Files` class to check, delete, copy, or move a file or directory
- Use Stream API with NIO2

Quiz

To prevent the serialization of operating system–specific fields, you should mark the field:

A. `private`
B. `static`
C. `transient`
D. `final`

Quiz

Given the following fragments:

```
public MyClass implements Serializable {
    private String name;
    private static int id = 10;
     private transient String keyword;
    public MyClass(String name, String keyword) {
        this.name = name; this.keyword = keyword;
    }
}
```

```
MyClass mc = new MyClass ("Zim", "xyzzy");
```

Assuming no other changes to the data, what is the value of `name` and `keyword` fields after deserialization of the `mc` object instance?

A. `Zim, ""`

B. `Zim, null`

C. `Zim, xyzzy`

D. `"", null`

Quiz

Given any starting directory path, which `FileVisitor` method(s) would you use to delete a file tree?

A. `preVisitDirectory()`

B. `postVisitDirectory()`

C. `visitFile()`

D. `visitDirectory()`

Quiz

Given a `Path` object with the following path:

`/export/home/duke/../peter/./documents`

Which `Path` method would remove the redundant elements?

A. `normalize`

B. `relativize`

C. `resolve`

D. `toAbsolutePath`

Quiz

To copy, move, or open a file or directory using NIO.2, you must first create an instance of:

A. `Path`

B. `Files`

C. `FileSystem`

D. `Channel`

Practice 17: Overview

This practice covers the following topics:

- 17-1**:** Writing a simple console I/O Application
- 17-2: Working with files