Build COMPETENCY
across your TEAM

SYNERGETICS GET IT RIGHT

Microsoft Partner
Gold Cloud Platform
Silver Learning

FileIO, Property Files

Google Cloud Platform

Windows Azure

amazon web services

APACHE kafka A distributed streaming platform

Apache SOFTWARE FOUNDATION

APACHE Spark

Java

hadoop

Context And Dependency Injection

Overview of I/O Streams

Types of Streams

The Byte-stream I/O hierarchy

Character Stream Hierarchy

Buffered Stream

The File class

The Path class

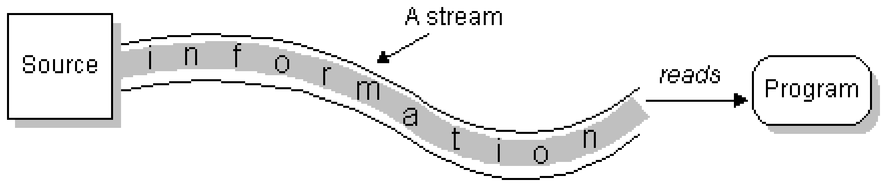
Object Stream

Property Files

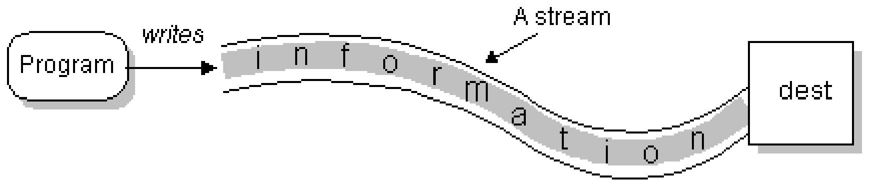
Overview of I/O Streams

Overview of I/O Streams

- To bring in information, a program opens a stream on an information source (a file, memory, a socket) and reads the information sequentially, as shown here



- Similarly, a program can send information to an external destination by opening a stream to a destination and writing the information out sequentially, like this:



- No matter where the data is coming from or going to and no matter what its type, the algorithms for sequentially reading and writing data are basically the same

Overview of I/O Streams

Reading	Writing
open a stream while more information read information close the stream	open a stream while more information write information close the stream

Overview of I/O Streams

- The java.io (in the API reference documentation) package contains a collection of stream classes that support these algorithms for reading and writing.
- To use these classes, a program needs to import the java.io (in the API reference documentation) package.
- The stream classes are divided into two class hierarchies, based on the data type (either characters or bytes) on which they operate.

SYNERGETICS GET IT RIGHT

Microsoft Partner
Gold Cloud Platform
Silver Learning

Types of Streams

SYNERGETICS GET IT RIGHT | Microsoft Partner Gold Cloud Platform Silver Learning

- Types of Stream
  - Char Stream
  - Bin Stream

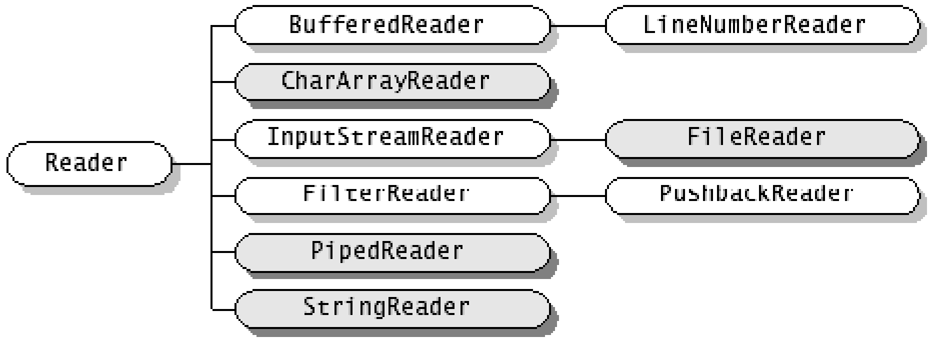
SYNERGETICS GET IT RIGHT | Microsoft Partner Gold Cloud Platform Silver Learning

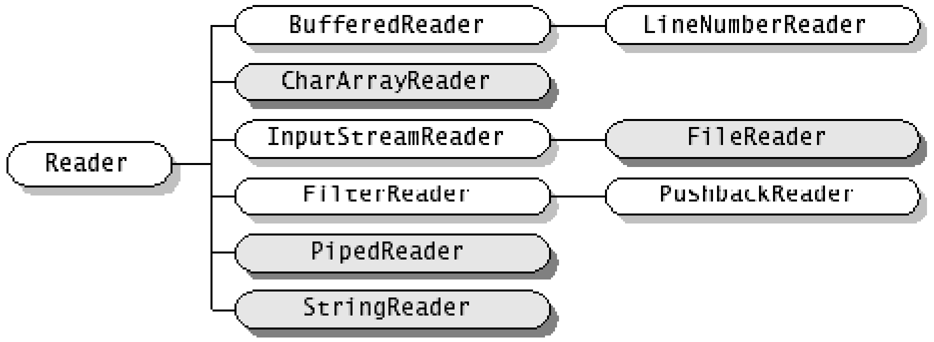
Character Stream Hierarchy

Character Streams

- Reader (in the API reference documentation) and Writer (in the API reference documentation) are the abstract superclasses for character streams in java.io.
- Reader provides the API and partial implementation for readers--streams that read 16-bit characters--and Writer provides the API and partial implementation for writers--streams that write 16-bit characters.
- Subclasses of Reader and Writer implement specialized streams and are divided into two categories: those that read from or write to data sinks and those that perform some sort of processing.

Character Streams

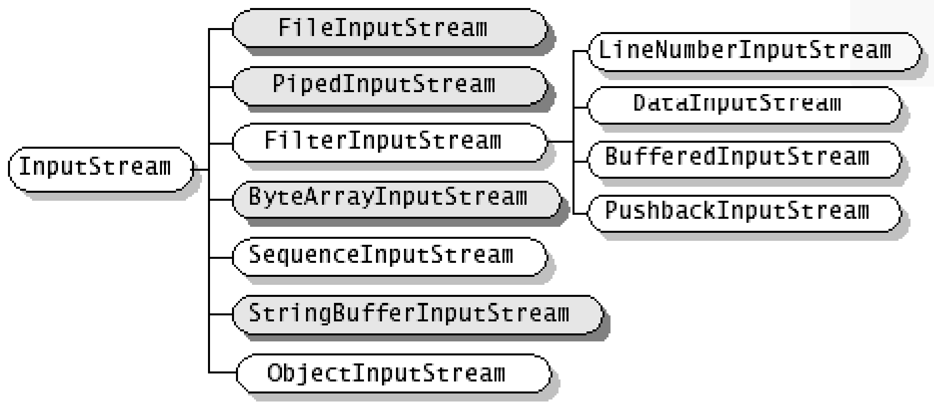


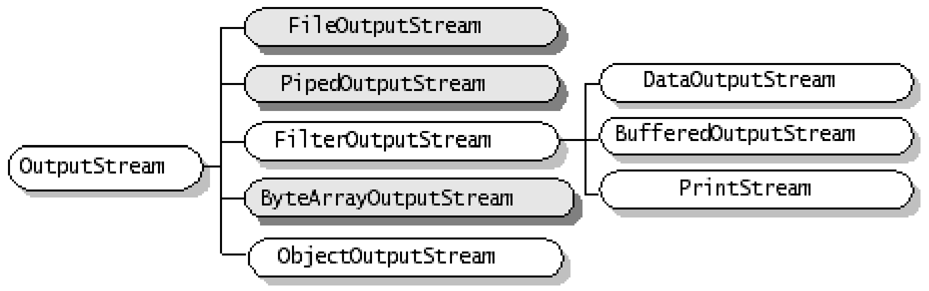


Byte Streams

- To read and write 8-bit bytes, programs should use the byte streams, descendants of InputStream and OutputStream.
- InputStream and OutputStream provide the API and partial implementation for input streams and output streams.
- These streams are typically used to read and write binary data such as images and sounds.
- Two of the byte stream classes, ObjectInputStream and ObjectOutputStream, are used for object serialization.
- These classes are covered in Object Serialization.
- As with Reader and Writer, subclasses of InputStream and OutputStream provide specialized I/O that falls into two categories: data sink streams and processing streams

Byte Streams





Buffered Stream

Buffered Stream

- This means each read or write request is handled directly by the underlying OS.
- This can make a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive.
- To reduce this kind of overhead, the Java platform implements buffered I/O streams.
- Buffered input streams read data from a memory area known as a buffer; the native input API is called only when the buffer is empty.
- Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full.
- program can convert an unbuffered stream into a buffered stream using the wrapping idiom we've used several times now, where the unbuffered stream object is passed to the constructor for a buffered stream class.

Buffered Stream

```
inputStream = new BufferedReader(new FileReader("xanadu.txt"));
outputStream = new BufferedWriter(new FileWriter("characteroutput.txt"));
```

- There are four buffered stream classes used to wrap unbuffered streams: BufferedInputStream and BufferedOutputStream create buffered byte streams, while BufferedReader and BufferedWriter create buffered character streams.

Flushing Buffered Streams

- It often makes sense to write out a buffer at critical points, without waiting for it to fill. This is known as flushing the buffer.
- Some buffered output classes support autoflush, specified by an optional constructor argument. When autoflush is enabled, certain key events cause the buffer to be flushed.
- For example, an autoflush PrintWriter object flushes the buffer on every invocation of println or format. See Formatting for more on these methods.
- To flush a stream manually, invoke its flush method. The flush method is valid on any output stream, but has no effect unless the stream is buffered.

The File class

File Class in Java

- he File class is Java's representation of a file or directory path name.
- Because file and directory names have different formats on different platforms, a simple string is not adequate to name them.
- The File class contains several methods for working with the path name, deleting and renaming files, creating new directories, listing the contents of a directory, and determining several common attributes of files and directories.

The File class

- It is an abstract representation of file and directory pathnames.
- A pathname, whether abstract or in string form can be either absolute or relative. The parent of an abstract pathname may be obtained by invoking the getParent() method of this class.
- First of all, we should create the File class object by passing the filename or directory name to it. A file system may implement restrictions to certain operations on the actual file-system object, such as reading, writing, and executing. These restrictions are collectively known as access permissions.
- Instances of the File class are immutable; that is, once created, the abstract pathname represented by a File object will never change.

The File class

- A File object is created by passing in a String that represents the name of a file, or a String or another File object.

File a = new File("/usr/local/bin/geeks");

Constructors

- File(File parent, String child) : Creates a new File instance from a parent abstract pathname and a child pathname string.
- File(String pathname) : Creates a new File instance by converting the given pathname string into an abstract pathname.
- File(String parent, String child) : Creates a new File instance from a parent pathname string and a child pathname string.
- File(URI uri) : Creates a new File instance by converting the given file: URI into an abstract pathname.

Some Methods of the File Class

- boolean canExecute() : Tests whether the application can execute the file denoted by this abstract pathname.
- boolean canRead() : Tests whether the application can read the file denoted by this abstract pathname.
- boolean canWrite() : Tests whether the application can modify the file denoted by this abstract pathname.
- int compareTo(File pathname) : Compares two abstract pathnames lexicographically.
- boolean createNewFile() : Atomically creates a new, empty file named by this abstract pathname .
- static File createTempFile(String prefix, String suffix) : Creates an empty file in the default temporary-file directory.
- boolean delete() : Deletes the file or directory denoted by this abstract pathname.
- boolean equals(Object obj) : Tests this abstract pathname for equality with the given object.
- boolean exists() : Tests whether the file or directory denoted by this abstract pathname exists.

SYNERGETICS GET IT RIGHT | Microsoft Partner Gold Cloud Platform Silver Learning

The Path class

The Path class

- The Path class includes various methods that can be used to obtain information about the path, access elements of the path, convert the path to other forms, or extract portions of a path.
- There are also methods for matching the path string and methods for removing redundancies in a path.
- This lesson addresses these Path methods, sometimes called syntactic operations, because they operate on the path itself and don't access the file system.

The Path class

- Creating a Path
- A Path instance contains the information used to specify the location of a file or directory. At the time it is defined, a Path is provided with a series of one or more names. A root element or a file name might be included, but neither are required.
- A Path might consist of just a single directory or file name.

The Path class

- You can easily create a Path object by using one of the following get methods from the Paths (note the plural) helper class:

```
Path p1 = Paths.get("/tmp/foo");
Path p2 = Paths.get(args[0]);
Path p3 = Paths.get(URI.create("file:///Users/joe/FileTest.java"));
```

The Paths.get method is shorthand for the following code:

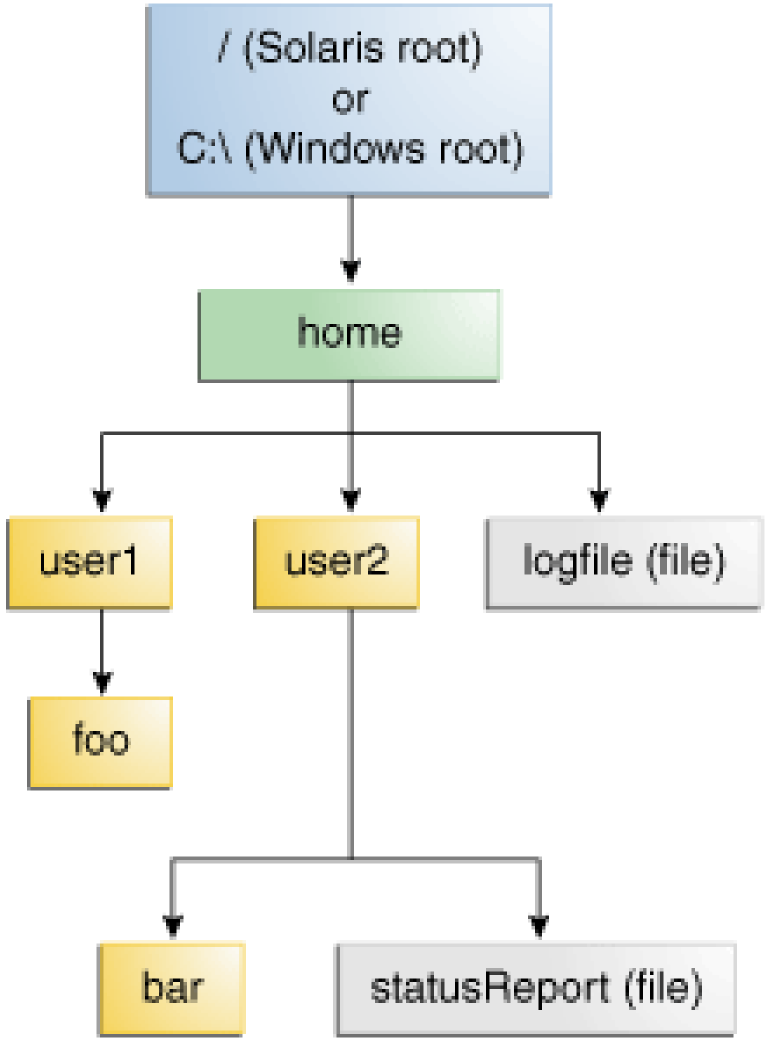
```
Path p4 = FileSystems.getDefault().getPath("/users/sally");
```

- The following example creates /u/joe/logs/foo.log assuming your home directory is /u/joe, or C:\joe\logs\foo.log if you are on Windows.

```
Path p5 = Paths.get(System.getProperty("user.home"),"logs", "foo.log");
```

Retrieving Information about a Path

- You can think of the Path as storing these name elements as a sequence. The highest element in the directory structure would be located at index 0.
- The lowest element in the directory structure would be located at index [n-1], where n is the number of name elements in the Path.
- Methods are available for retrieving individual elements or a subsequence of the Path using these indexes.



Retrieving Information about a Path

The following code snippet defines a Path instance and then invokes several methods to obtain information about the path:

// None of these methods requires that the file corresponding

// to the Path exists.

// Microsoft Windows syntax

Path path = Paths.get("C:\\home\\joe\\foo");

// Solaris syntax

- Path path = Paths.get("/home/joe/foo");

- System.out.format("toString: %s%n", path.toString());
- System.out.format("getFileName: %s%n", path.getFileName());
- System.out.format("getName(0): %s%n", path.getName(0));
- System.out.format("getNameCount: %d%n", path.getNameCount());
- System.out.format("subpath(0,2): %s%n", path.subpath(0,2));
- System.out.format("getParent: %s%n", path.getParent());
- System.out.format("getRoot: %s%n", path.getRoot());

Object Stream

Object Stream

- ObjectStream
  - ObjectInputStream
  - ObjectOutputStream

ObjectInputStream

- The Java ObjectInputStream class (java.io.ObjectInputStream) enables you to read Java objects from an InputStream instead of just raw bytes.
- You wrap an InputStream in a ObjectInputStream and then you can read objects from it.
- The bytes read must represent a valid, serialized Java object. Otherwise reading objects will fail.
- Normally you will use the ObjectInputStream to read objects written (serialized) by a Java ObjectOutputStream . You will see an example of that later.

ObjectInputStream

```
ObjectInputStream objectInputStream =
    new ObjectInputStream(new FileInputStream("object.data"));

MyClass object = (MyClass) objectInputStream.readObject();
//etc.

objectInputStream.close();
```

For this ObjectInputStream example to work the object you read must be an instance of MyClass, and must have been serialized into the file "object.data" via an ObjectOutputStream.

Before you can serialize and de-serialize objects the class of the object must implement java.io.Serializable.

ObjectInputStream

- **Closing a ObjectInputStream**

- When you are finished reading data from the ObjectInputStream you should remember to close it. Closing a ObjectInputStream will also close the InputStream instance from which the ObjectInputStream is reading.

- Closing a ObjectInputStream is done by calling its close() method.

- objectInputStream.close();

ObjectOutputStream

- The Java ObjectOutputStream class (java.io.ObjectOutputStream) enables you to write Java objects to an OutputStream instead of just raw bytes. You wrap an OutputStream in a ObjectOutputStream and then you can write objects to it.
- The Java ObjectOutputStream is often used together with a Java ObjectInputStream. The ObjectOutputStream is used to write the Java objects, and the ObjectInputStream is used to read the objects again.

ObjectOutputStream

```
ObjectOutputStream objectOutputStream =
    new ObjectOutputStream(new FileOutputStream("object.data"));

MyClass object = new MyClass();

output.writeObject(object);

output.close();
```

First this examples creates a OutputOutputStream connected to a FileOutputStream. Then the example creates a MyClass object and writes it to the ObjectOutputStream. Finally the example closes the ObjectOutputStream.

Before you can serialize and de-serialize objects the class of the object must implement java.io.Serializable. For more info, see Java Serializable.

ObjectOutputStream

- **Closing a ObjectOutputStream**
- When you are finished writing data to the ObjectOutputStream you should remember to close it. Closing a ObjectOutputStream will also close the OutputStream instance to which the ObjectOutputStream is writing.
- Closing a ObjectOutputStream is done by calling its close() method. Here is how closing a ObjectOutputStream looks:
- objectOutputStream.close();

Property Files

Java Property File Processing

- Properties is a file extension for files mainly used in Java related technologies to store the configurable parameters of an application.
- Java Properties files are amazing resources to add information in Java. Generally, these files are used to store static information in key and value pair.
- Things that you do not want to hard code in your Java code goes into properties files.
- The advantage of using properties file is we can configure things which are prone to change over a period of time without the need of changing anything in code. Properties file provide flexibility in terms of configuration.

- Sample properties file is shown below, which has information in key-value pair.
- Each parameter is stored as a pair of strings, left side of equal (=) sign is for storing the name of the parameter (called the key), and the other storing the value.

#This is property file having my configuration

- FileName=Data.txt
- Author_Name=Amit Himani
- Topic=Properties file Processing

First Line which starts with # is called comment line. We can add comments in properties which will be ignored by java compiler.

Reading the property File using load method

```
Properties prop = new Properties();
		try {
			// load a properties file for reading
			prop.load(new FileInputStream("myConfig.properties"));
			// get the properties and print
			prop.list(System.out);
			//Reading each property value
			System.out.println(prop.getProperty("FileName"));
			System.out.println(prop.getProperty("Author_Name"));
			System.out.println(prop.getProperty("TOPIC"));
```

Reading the property File using load method

- The program uses load( ) method to retrieve the list. When the program executes, it first tries to load the list from a file called myConfig.properties.
- If this file exists, the list is loaded else IO exception is thrown.

Use of getProperties() Method

- One useful capability of the Properties class is that you can specify a default property that will be returned if no value is associated with a certain key.
- For example, a default value can be specified along with the key in the getProperty( ) method—such as getProperty("name","default value").
- If the "name" value is not found, then "default value" is returned.
- When you construct a Properties object, you can pass another instance of Properties to be used as the default properties for the new instance.

Writing Properties file

- At any time, you can write a Properties object to a stream or read it back.
- This makes property lists especially convenient for implementing simple databases.
- For Example below program writes states can capital cities. “capitals.properties” file having state name as keys and state capital as values.

Writing Properties file

```
Properties prop = new Properties();
        try {
            // set the properties value
            prop.setProperty("Gujarat", "Gandhinagar");
            prop.setProperty("Maharashtra", "Mumbai");
            prop.setProperty("Madhya_Pradesh", "Indore");
            prop.setProperty("Rajasthan", "Jaipur");
            prop.setProperty("Punjab", "mkyong");
            prop.setProperty("Uttar_Pradesh", "Lucknow");
            // save properties to project root folder
            prop.store(new FileOutputStream("capitals.properties"), null);
```

After running the program we can see a new file created named “capitals.properties” under project root folder as shown below.

SYNERGETICS GET IT RIGHT | Microsoft Partner Gold Cloud Platform Silver Learning

Q & A

Contact: amitmahadik@synergetics-india.com

SYNERGETICS GET IT RIGHT | Microsoft Partner Gold Cloud Platform Silver Learning

Thank You