Java IO

System properties

The Java platform uses a Properties object to maintain its own configuration. The System class maintains a Properties object that describes the configuration of the current working environment. System properties include information about the current user, the current version of the Java runtime, and the character used to separate components of a file path name.

- "file.separator" Character that separates components of a file path. This is "/" on UNIX and "" on Windows.
- "java.class.path" Path used to find directories and JAR archives containing class files.
- "java.home" Installation directory for Java Runtime Environment (JRE)
- "java.vendor" JRE vendor name
- "java.vendor.url" JRE vendor URL
- "java.version" JRE version number
- "line.separator" Sequence used by operating system to separate lines in text files
- "os.arch" Operating system architecture
- "os.name" Operating system name
- "os.version" Operating system version
- "path.separator" Path separator character used in java.class.path
- "user.dir" User working directory
- "user.home" User home directory
- "user.name" User account name

```
System.getProperty("file.separator");
```

The `getProperty` method returns a string containing the value of the property. If the property does not exist, this version of getProperty returns null.

The "file.separator" property is useful for writing portable paths as shown below.

```
String desktopDir = String.format("%s%s%s", System.getProperty("user
```

The File class

There is a class called `File` in the `java.io` package. An object of this class represents an existing or non-existing file or a directory. The class can be used to manipulate files and directories: creating, removing, accessing properties and more.

```
File fileOnUnix = new File("/home/username/Documents");    // a dir
File fileOnWin = new File("D:\\Materials\\java-materials.pdf"); // 
```

The code will work even if a file or a directory does not actually exist in your file system. It does not create a new file or directory. It just represents "a virtual" file or directory that exists already or may be created in the future.

Absolute and relative paths

A path is **absolute** if it starts with the root element of the file system. It has the complete information about the file location including the type of the operating system.

A **relative path** is a path that doesn't include the root element of the file system. This always starts from your **working directory**. This directory is represented by a `.` (dot). A relative path is not complete and needs to be combined with the current directory path in order to reach the requested file.

Here is an example with a file inside the images directory which is in your working directory:

```
File fileOnUnix = new File("./images/picture.jpg");
File fileOnWin = new File(".\\images\\picture.jpg");
```

As you can see, both paths look exactly the same, which provides platform independence. Interestingly, the dot character can be skipped, so the path `images/picture.jpg` is also correct.

In order to construct platform-independent programs, it is a common convention to use relative paths whenever possible.

Basic methods

An instance of `File` has a list of methods. Take a look at some of them:

- `String getPath()` returns the string path to this file or directory;
- `String getName()` returns the name of this file or directory (just the last name of the path)
- `boolean isDirectory()` returns `true` if it is a directory and exists, otherwise, `false`;

- `boolean isFile()` returns `true` if it is a file that exists (not a directory), otherwise, `false`;
- `boolean exists()` returns `true` if this file or directory actually exists in your file system, otherwise, `false`;
- `String getParent()` returns the string path to the parent directory that contains this file or directory.

The list is not complete, but for now, we will focus on these. For other methods, see here.

```
File file = new File("/home/username/Documents/javamaterials.pdf");

System.out.println("File name: " + file.getName());
System.out.println("File path: " + file.getPath());
System.out.println("Is file: " + file.isFile());
System.out.println("Is directory: " + file.isDirectory());
System.out.println("Exists: " + file.exists());
System.out.println("Parent path: " + file.getParent());
```

This code prints the following:

```
File name: javamaterials.pdf
File path: /home/username/Documents/javamaterials.pdf
Is file: true
Is directory: false
Exists: true
Parent path: /home/username/Documents
```

There is also a group of methods `canRead()`, `canWrite()`, `canExecute()` to test whether the application can **read/modify/execute** the file denoted by the path.

Creating files and directories

After creating an instance of `File` we should invoke the method `createNewFile`. The method returns `true` if the file was successfully created and `false` if it already exists. It does not erase the content of an existing file.

```
File file = new File("/home/username/Documents/file.txt");
try {
    boolean createdNew = file.createNewFile();
    if (createdNew) {
        System.out.println("The file was successfully created.");
    } else {
        System.out.println("The file already exists.");
    }
} catch (IOException e) {
    System.out.println("Cannot create the file: " + file.getPath());
}
```

To create a directory we also need to start by creating an instance of `java.io.File`. After that, we should call one of the two methods of this instance:

- `boolean mkdir` creates the directory; it returns `true` only if the directory was created, otherwise, it returns `false`.
- `boolean mkdirs` creates the directory including all necessary non-existing parent directories; it returns `true` only if the directory was created along with all the specified parent directories.

Both methods do not throw `IOException`, unlike the `createNewFile` method.

```
File file = new File("/home/art/Documents/dir");

boolean createdNewDirectory = file.mkdir();
if (createdNewDirectory) {
    System.out.println("It was successfully created.");
} else {
    System.out.println("It was not created.");
}
```

Here is another example, demonstrating the `mkdirs` method. It creates the target directory and all parent directories if they do not exist.

```
File file = new File("/home/art/Documents/dir/dir/dir");

boolean createdNewDirectory = file.mkdirs();
if (createdNewDirectory) {
    System.out.println("It was successfully created.");
} else {
    System.out.println("It was not created.");
}
```

Removing files and directories

There is a method named `delete` to remove a file or a directory. It returns `true` if and only if the file or directory is successfully deleted, otherwise, it returns `false`. The method returns `false` if the file or directory does not exist. It is important to remember that it also returns `false` if the directory contains subdirectories or files. It means that the method will not remove a hierarchy, only a particular file or an empty directory.

```
File file = new File("/home/art/Documents/dir/dir/dir");

if (file.delete()) {
    System.out.println("It was successfully removed.");
} else {
    System.out.println("It was not removed.");
}
```

To delete a directory that is not empty, at first you have to delete all the nested files and directories. Take a look at the code below. It recursively deletes directories with their content. Note that the method assumes that the passed directory `dir` does exist. Otherwise, `children == null` and `NullPointerException` will be thrown.

```
public void deleteDirRecursively(File dir) {
    File[] children = dir.listFiles();
    for (File child : children) {
        if (child.isDirectory()) {
            deleteDirRecursively(child);
        } else {
            child.delete();
        }
    }
    dir.delete();
}
```

The method `delete` never throws an `IOException`.

Renaming files and directories

The method `renameTo` changes the name of the file by editing it in the abstract path. It returns `true` if and only if the renaming succeeded, otherwise, `false`.

```
File file = new File("/home/art/Documents/dir/filename.txt");

boolean renamed = file.renameTo(new File("/home/art/Documents/dir/ne
```

The same method can be used to move the file or directory from the current location to another one:

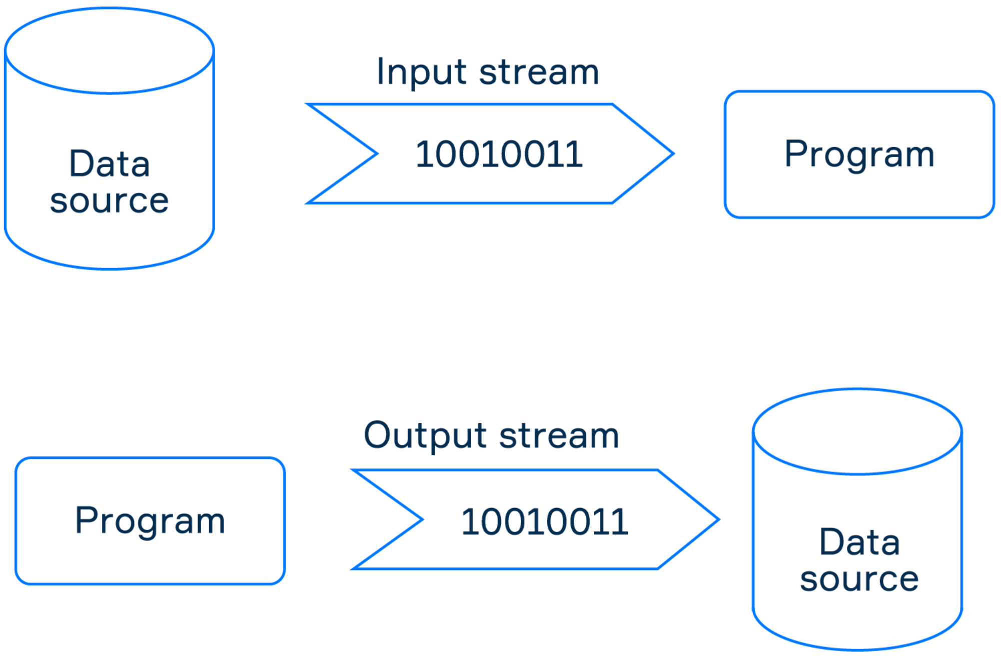
```
File file = new File("/home/art/Documents/dir/file.txt");

boolean renamed = file.renameTo(new File("/home/art/Documents/anothe
```

Input and output streams

In some sense, a Java stream is similar to a real-world water stream which has a beginning (source) and an end (destination). Based on the same principles, IO streams can be categorized into two groups:

- **input streams** which read data from a source;
- **output streams**, which write data to a specified destination.

The picture below demonstrates it.



In fact, you've already known two specific examples of IO streams: **System.in** and **System.out**. We used them to read/write data from/to the console before.

Byte and char streams

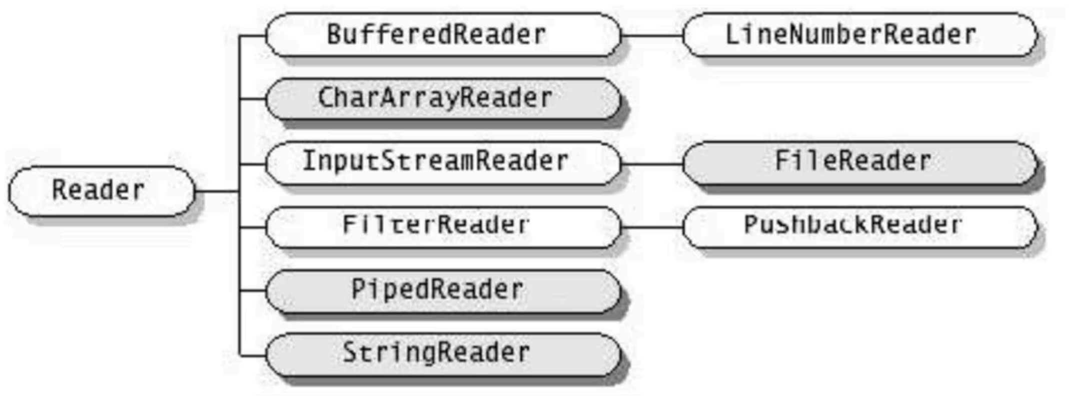
Streams can be further classified into two categories based on how they represent sequences of data:

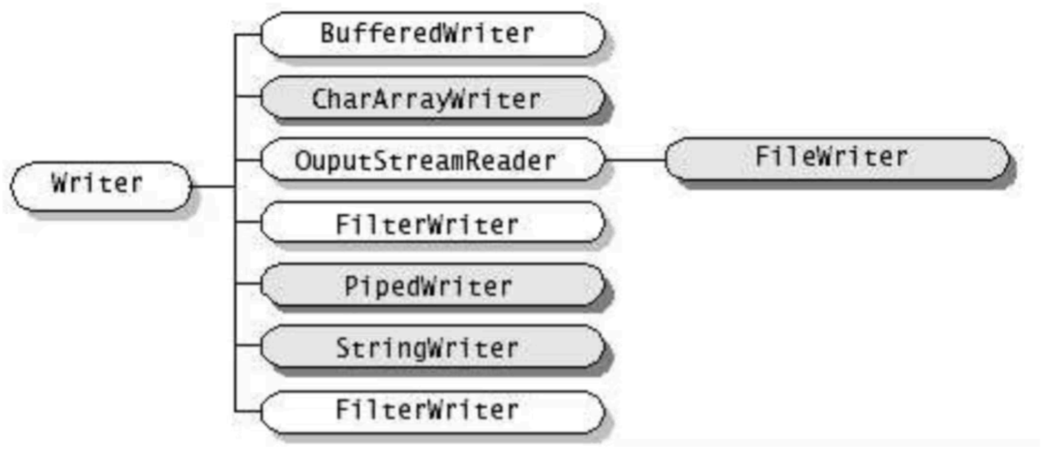
- **byte streams** that are used to read and write data in bytes;
- **char streams** that are used to read and write data in characters according to the 16-bit Unicode format.

A computer understands only sequences of bytes. From this perspective, any data is a set of bytes and byte input streams are a common way of reading any kind of data. For computers, characters are still combinations of bytes defined by a charset specification.

On the contrary, as human beings, we are used to dealing with sequences of characters. Character input streams are aimed to read data which consists of characters. Under the hood, they still read bytes, but they immediately encode bytes to characters.

Reader and Writer interfaces





Character streams (input)

There are several classes for reading text. They are called character input streams and allow reading text data: `char` or `String`. For instance, there are `FileReader`, `CharArrayReader`, `StringReader`, etc. The class name indicates what type of source it uses as input and usually ends with `Reader`, since all such classes extend the `java.io.Reader` class.

Each class provides a set of useful methods while they also have common methods for reading data:

- `int read()` reads a single character. If the end of the stream is reached, the method returns the value `-1`. Otherwise, it returns the numerical representation of the character according to the current encoding;
- `int read(char[] cbuf)` reads a sequence of characters into the passed array up to its capacity and returns the number of characters that were actually read. It can also return `-1` in case no data was read;
- `int read(char[] cbuf, int off, int len)` reads characters into a portion of an array.

Each character stream class also has a `void close()` method to release system resources. If you're familiar with the try-with-resources construction, you know it is a better way to prevent resource leaks.

```
Reader reader = new FileReader("file.txt");

char first = (char) reader.read(); // i
char second = (char) reader.read(); // n

char[] others = new char[12];
int number = reader.read(others); // 10
```

After running the code, `others` will contain `['p', 'u', 't', ' ', 's', 't', 'r', 'e', 'a', 'm', '\u0000', '\u0000']`.

Let's explain the result. Since we've read the first two letters into other variables, the first 10 characters of `others` are filled starting from the third letter. When the stream reached the end of the file it stopped reading, so the last two characters are not updated.

Character streams (output)

Character output streams allow writing text data: `char` or `String`. You might have already used such streams as`FileWriter` and `PrintWriter` earlier for writing text data to files. Both of them, as well as other character output streams, have a common abstract ancestor `java.io.Writer`.

The class contains a group of methods for writing. Some of them are listed here:

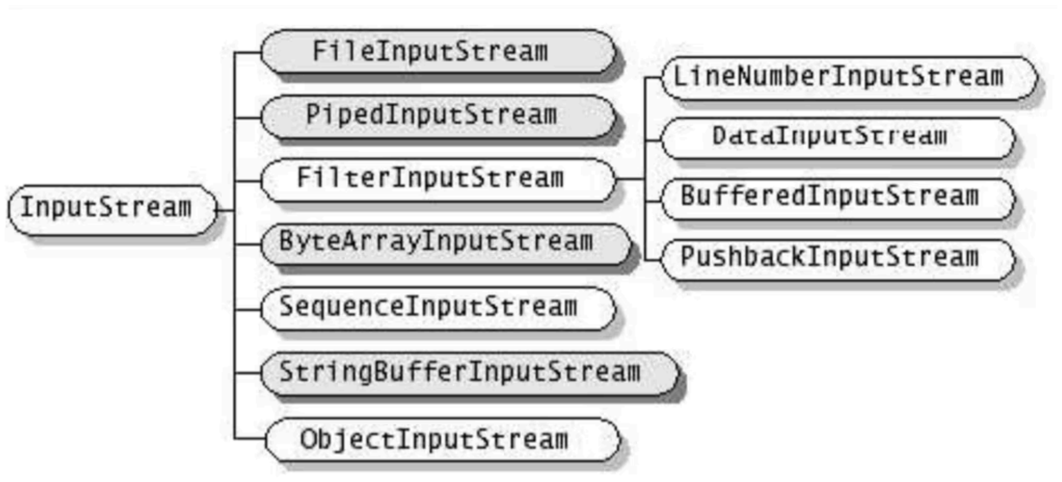
- `void write(char[] cbuf)` writes a char array

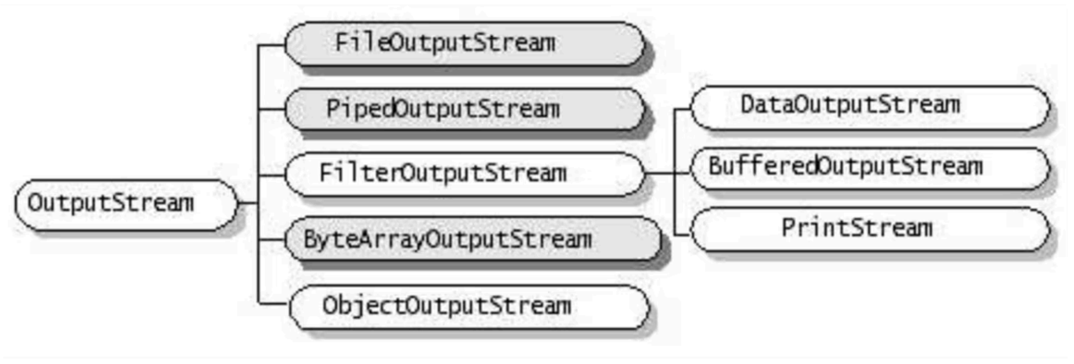
- `void write(char[] cbuf, int off, int len)` writes a portion of a char array
- `void write(int c)` writes a single character
- `void write(String str)` writes a string
- `void write(String str, int off, int len)` writes a portion of a string

`Writer` has several direct subclasses for different purposes in the standard library. For example, `FileWriter` is intended for writing to files. `StringWriter` is designed to construct a string. `CharArrayWriter` uses `char[]` as a destination.

```
// copy a text file line by line
File src = new File("src/test/resources/copiedWithIo.txt");
File dst = new File("src/test/resources/copiedWithIo.txt.bak");
try (BufferedReader in = new BufferedReader(new FileReader(src));
     BufferedWriter out = new BufferedWriter(new FileWriter(dst))) {
    String line;
    while ((line = in.readLine()) != null) {
        out.write(line);
    }
}
```

Inputstream and Outputstream interfaces





Byte streams (input)

The class name of a byte stream indicates what type of source it uses as input and usually ends with `InputStream`, since all such classes extend the `java.io.InputStream`class.

All byte stream classes have methods for reading similar to character input streams:

- `abstract int read()` reads a single byte;
- `int read(byte[] b)` reads a number of bytes and stores them in a byte array;
- `byte[] readAllBytes()` reads all bytes.

The method that reads bytes into an array, returns an `int` value. It is the number of bytes that were actually read from the source. If `-1` value is returned it is a sign that no bytes were read.

Each input stream class also has a `void close()` method to release system resources.

```
FileInputStream inputStream = new FileInputStream("file.txt");
byte[] bytes = new byte[5];
int numberOfBytes = inputStream.read(bytes);
System.out.println(numberOfBytes); // 5
inputStream.close();
```

Now `bytes`contains `['i', 'n', 'p', 'u', 't']`.

The byte-by-byte approach also works here, similar to the character streams example.

Byte streams (output)

Byte output stream classes from the standard library extend `java.io.OutputStream` abstract class. The class contains three methods for writing:

- `void write(byte[] b)` writes a byte array
- `void write(byte[] b, int off, int len)` writes a portion of a byte array
- `abstract void write(int b)` writes a single byte

Just like character streams, byte streams have `void close()` that should be invoked in a similar way.

Let's look at some direct subclasses of `OutputStream` from the standard library.`FileOutputStream` is intended for writing data to a file as a destination. `ByteArrayOutputStream` as you may guess allows writing to `byte[]` destination. Such classes like `FilterOutputStream` or `PipedOutputStream` have no endpoint destination and write data to other output streams. These classes are supposed to be intermediate streams for data transformation or possibly providing additional functionality.

```
// copy a binary file in chunks of 1K
File src = new File("src/test/resources/copiedWithIo.txt");
File dst = new File("src/test/resources/copiedWithIo.txt.bak");
try (InputStream in = new FileInputStream(src));
     OutputStream out = new FileOutputStream(dst))) {

    byte[] buffer = new byte[1024];
    int lengthRead;
    while ((lengthRead = in.read(buffer)) > 0) {
        out.write(buffer, 0, lengthRead);
    }
}
```

References

- https://www.baeldung.com/java-io-file
- https://www.baeldung.com/java-io