The differences between standard IO and NIO:

The standard IO (java.io) is based on streams (byte streams and character streams), whereas the NIO (java.nio) is based on buffers and channels.

- With standard IO, you can only read from an input stream or write to an output stream.

- With NIO, you can both read from and write to a channel asynchronously.

Another key difference is that the old IO is blocking and the NIO has non-blocking mode. Let me explain:

- In old IO, when you call `read()` or `write()`, the current thread is blocked until the data is fully read or written. In other words, the program has to wait while doing IO operations.

- In NIO, the network channels (`DatagramChannel`, `SocketChannel` and `ServerSocketChannel`) can be configured for non-blocking mode, in which the current thread returns immediately after invoking a read or write data over the network. This feature can be handy in multi-threaded applications. Let take a chat application for example, the user is not blocked while sending a message to another user, and he can chat with different users at the same time. Hence NIO is also referred as Non-blocking IO.

Does NIO replace standard IO?

No, NIO doesn't replace old IO. The NIO supplements the old IO. In other words, NIO is an extension to standard IO.

Both APIs have interoperability between each other. For example, you can get a channel from a file or get an input/output stream from a channel.

NIO brings new ways for working with input and output via buffers and channels. It also provides non-blocking feature as mentioned above, and consists of some utility classes for simplifying common tasks.

Is NIO really faster than standard IO?

No, there's no guarantee that NIO is faster than old IO. I was wrong when saying that NIO increases performance over the standard IO. Actually, NIO is designed for the ease of scalability with channels and selectors. That means you can use NIO for developing client-server applications that manage thousand concurrent connections easier than using the standard IO.

When to use NIO?

First, use NIO when you are lazy so you want to do common tasks easily. For example, use the utility class `Files` for reading all bytes or all lines from files with ease.

Second, use NIO for developing non-blocking network client-server applications that requires scalability.

When to use standard IO?

Of course you can use the standard IO for trivial Input/Output operations. And it doesn't hurt if you don't use NIO

1. Creating Paths

A `Path` object represents path name of a file or directory in the file system. Almost File I/O operations involve in using `Path`objects, so you need to know how to create a `Path`object by using the `Paths` helper class. Here are some examples:

Create a path which is relative to the current directory:

```
Path path1 = Paths.get("NioExamples");
```

Create an absolute path on Windows:

```
Path path2 = Paths.get("E:\\Java\\JavaSE\\NIO");
```

Create an absolute path on Linux:

```
Path path3 = Paths.get("/home/yourname/Java/NIO");
```

Create a relative path: Java/NIOExamples.java

```
Path path4 = Paths.get("Java", "NIOExamples.java");
```

Create a path by joining two paths:

```
Path path5 = Paths.get("/home/john");
// path6 will be /home/john/codejava
Path path6 = path5.resolve("codejava");
```

Create a relative path between two paths:

```
Path path7 = Paths.get("Java");
Path path8 = Paths.get("Projects");
// path9 will be ..\Projects
Path path9 = path7.relativize(path8);
```

2. Creating Empty Files

You can use the **`Files.createFile()`** method to create an empty file from a given `Path` with default file attribute or a specified file attribute. For example, the following code creates the Hello.java file in the current directory:

```
Path file = Paths.get("Hello.java");
try {
      Files.createFile(file);
} catch (FileAlreadyExistsException ex) {
      System.err.format("File %s already exists%n", file);
} catch (IOException ex) {
      System.err.format("Error creating file %s%n", file);

}
```

Note that the `createFile()` method throws `FileAlreadyExistsException` if the file already exists and return a `Path` object represents the file created.

The following code creates a file with a given permission:

```
Path file = Paths.get("NIO.java");
Set< PosixFilePermission > permssion =
      PosixFilePermissions.fromString("rwxr-xr--");
FileAttribute< Set< PosixFilePermission > > attributes =
      PosixFilePermissions.asFileAttribute(permssion);
try {
      Files.createFile(file, attributes);
} catch (FileAlreadyExistsException ex) {
      System.err.format("File %s already exists%n", file);
} catch (IOException ex) {
      System.err.format("Error creating file %s%n", file);
}
```

Note that this doesn't work on Windows because Windows doesn't support Posix file permissions.

On Windows, you have to specify permissions by using the methods of the `java.io.File` class: `setExecutable()`, `setReadable()`, `setReadOnly()`, and`setWritable()`.

3. Creating Directories

Use the `Files.createDirectory()` method to create a single directory from a given Path. The example below creates a directory relative to the current directory:

```
Path dir = Paths.get("NIO Examples");
Files.createDirectory(dir);
```

Note that the `createDirectory()` method throws `FileAlreadyExistsException` if a file of that name already exists and return a `Path` object represents the directory created.

You can also use the `Files.createDirectories()` method to create a directory by creating all nonexistent parent directories first. Here's an example:

```
Path dirs = Paths.get("E:\\JavaCode\\NIO\\Directories");
Files.createDirectories(dirs);
```

If a parent directory in the path does not exist, it is created before the last child one gets created. Note that this method doesn't throw an exception if a directory already exists.

4. Creating a Temporary File or Directory

Use the `Files.createTempFile(dir, prefix, suffix)` to create a temporary file in a given directory. The system generates the file name with the specified prefix and suffix. The following code creates a temporary file in the user's home directory:

```
Path dir = Paths.get(System.getProperty("user.home"));
try {
      Files.createTempFile(dir, "CodeJava", ".log");
} catch (IOException ex) {
      System.err.format("Error creating temp file");
}
```

You can see a file named something like `CodeJava1234805573144823192.log` created.

Similarly, you can use the `Files.createTempDirectory(dir, prefix)` to create a temporary directory in a given directory with the directory name starts with the specified prefix. For example:

```
Path dir = Paths.get(System.getProperty("user.home"));
try {
      Files.createTempDirectory(dir, "CodeJava");
} catch (IOException ex) {
      System.err.format("Error creating temp dir");
}
```

You can see a directory named something like `CodeJava3020552128289338354` created in the user's home directory.

5. Listing a Directory

The `Files.newDirectoryStream(dir)` returns a `DirectoryStream` to iterate over all entries in the given directory. For example, the following code lists content of the current directory:

```
Path dir = Paths.get(".");
try (DirectoryStream< Path > stream = Files.newDirectoryStream(dir)) {
      for (Path file: stream) {
            System.out.println(file.getFileName());
      }
} catch (IOException | DirectoryIteratorException x) {
      System.err.println(x);
```

```
        }
```

This code lists all files and directories in the current directory. In case if you want to apply a filter, supply a `DirectoryStream.Filter` for the overload method `Files.newDirectoryStream(dir, filter)`. For example, the following code lists only sub directories in the current directory:

```
        Path dir = Paths.get(".");
        DirectoryStream.Filter< Path > filter =
              new DirectoryStream.Filter< Path >() {
                    public boolean accept(Path path) throws IOException {
                          return (Files.isDirectory(path));
                    }
        };
        try (DirectoryStream< Path > stream = Files.newDirectoryStream(dir, filter)) {
              for (Path file: stream) {
                    System.out.println(file.getFileName());
              }
        } catch (IOException | DirectoryIteratorException x) {
              System.err.println(x);
        }
```

As you can see in the above code, the filter condition is specified by the accept() method of a `DirectoryStream.Filter` class.

6. Checking a File or Directory

The `Files` class provides some methods for verifying the existence and accessibility of a file/directory.

You can use the `Files.exists()` method to verify the existence of a file/directory, for example:

```
        Path dir = Paths.get("NIO Examples");
        boolean exist = Files.exists(dir);
```

The `Files.notExists()` method tests if a file/directory not exists, for example:

```
        Path file = Paths.get("hello.cpp");
        boolean notExist = Files.notExists(file);
```

Note that the `notExists()` method is not the complement of the `exists()` method, as both can return false if it's impossible to determine the existence.

And there are other checking methods as described below:

- `isDirectory(Path)`: tests whether a file is a directory.
- `isExecutable(Path)`: tests whether a file is executable.
- `isHidden(Path)`: tests whether a file is hidden or not.
- `isReadable(Path)`: tests whether a file is readable.
- `isSameFile(Path path, Path path2)`: tests if two paths locate the same file.
- `isSymbolicLink(Path)`: tests whether a file is a symbolic link.

- `isWritable(Path)`: tests whether a file is writable.

Note that for brevity I don't show a variable argument `LinkOption` of some methods. This argument indicates how symbolic links are handled, e.g. no follow links. By default, symbolic links are followed.

7. Copying a File or Directory

The Files class provides three different methods for copying a file or directory:

- copy(InputStream in, Path target, CopyOption... options): copies all bytes from an input stream to a file.
- copy(Path source, OutputStream out): copies all bytes from a file to an output stream.
- copy(Path source, Path target, CopyOption... options): copy a file to a target file.

Use the enum `StandardCopyOption` for the `CopyOption` parameter with the following values:

- `ATOMIC_MOVE`: move the file as an atomic file system operation.
- `COPY_ATTRIBUTES`: copy attributes to the new file, e.g. last-modified-time attribute.
- `REPLACE_EXISTING`: replace an existing file if it exists.

The following code copies a video file from the current directory to another sub directory:

```
Path source = Paths.get("NioVideo.mp4");
Path target = Paths.get("NIO Examples\\NioVideo.mp4");
try {
      Files.copy(source, target,
            StandardCopyOption.COPY_ATTRIBUTES,
            StandardCopyOption.REPLACE_EXISTING);
} catch (IOException ex) {
      System.err.format("Error copying file");
}
```

If the source is a directory, then only the directory is created even when the original directory contains files.

8. Deleting a File or Directory

The `Files` class provides two methods for deleting a file or an empty directory:

- `delete(Path)`: deletes a file specified by the given `Path`. It throws an exception if the deletion fails such as the file does not exist.
- `deleteIfExists(Path)`: also deletes a file, but no exception is thrown if the file does not exist.

Here's an example:

```
Path path = Paths.get("Gau.mp4");
try {
      Files.delete(path);
} catch (NoSuchFileException ex) {
			System.out.format("%s: no such file or directory", path);
		} catch (DirectoryNotEmptyException ex) {
			System.out.format("%s not empty", path);
		} catch (IOException ex) {
			System.err.println(ex);
		}
```
Note that if the path is a directory, the directory must be empty.

9. Moving a File or Directory

You can move or rename a file to a target file using this method:

```
		Files.move(Path source, Path target, CopyOption… options)
```

The copy options can be used are `ATOMIC_MOVE` and `REPLACE_EXISTING` of the `StandardCopyOption` enum.

For example, the following code renames a file in the current directory:

```
		Path source = Paths.get("ReadTextFileNIO.java");
		Path target = Paths.get("ReadTextFile.java");
		try {
			Files.move(source, target);
		} catch (IOException ex) {
			System.err.println(ex);
		}
```
The following code moves a file in the current directory to another directory:

```
		Path source = Paths.get("ReadTextFile.java");
		Path target = Paths.get("e:\\Java\\JavaSE\\NIO\\ReadTextFile.java");
		try {
			Files.move(source, target);
		} catch (IOException ex) {
			System.err.println(ex);
		}
```

Note that a directory can be moved only if the move operation does not require moving the contents of that directory, and this behavior depends on directories, file systems and operating systems.

10. Working with the File System

The `FileSystem` class represents a file system, the `FileStore` class represents a partition or volume, and the `FileSystems` class provides factory methods for file systems, such as the `getDefault()` method to get the default file system.

The following code snippet uses the above classes to list all partitions on the default file system and get the capacity and free space of each:

```
try {
      FileSystem fs = FileSystems.getDefault();
      for (FileStore store: fs.getFileStores()) {
            System.out.println(store);
            System.out.format("\tTotal Space: %d%n", store.getTotalSpace());
            System.out.format("\tFree Space: %d%n",
store.getUnallocatedSpace());
      }
} catch (IOException ex) {
      System.err.println(ex);
}
```

Here's a sample output on my computer:

```
(C:)
        Total Space: 73402363904
        Free Space: 8049229824
DATA (D:)
        Total Space: 106151542272
        Free Space: 4770100736
CODE (E:)
        Total Space: 106232282624
        Free Space: 38338893312
```

11. Reading All Bytes or Lines from a File

The `Files`class provides three methods for reading entire content of small files:

- `byte[]` **`readAllBytes`**`(Path path)`: reads entire content of a file to a byte array.
- `List< String >` **`readAllLines`**`(Path path, Charset cs)`: reads all lines from a file using the specified charset.
- `List< String >` **`readAllLines`**`(Path path)`: reads all lines from a file using the UTF-8 charset.

Note that these methods ensure that the file is closed when all bytes have been read or an I/O error or exception is thrown.

The following code reads a file to a byte array, which can be used for other purpose such as input for encryption or compression:

```
Path file = Paths.get("Photos.png");
try {
      byte[] bytes = Files.readAllBytes(file);
} catch (IOException ex) {
      System.err.println(ex);
}
```

And the following code reads all lines from a file and prints them on the screen:

```
Path file = Paths.get("ReadingFileExamples.java");
try {
      List< String > lines = Files.readAllLines(file);
      for (String line : lines) {
            System.out.println(line);
      }
} catch (IOException ex) {
      System.err.println(ex);
}
```

12. Writing All Bytes or Lines to a File

The `Files` class also provides methods for writing bytes or lines to a small file:

- **write**(Path path, byte[] bytes, OpenOption... options)
- **write**(Path path, Iterable< ? extends CharSequence > lines, OpenOption... options)
- **write**(Path path, Iterable< ? extends CharSequence > lines, Charset cs, OpenOption... options)

These methods ensure that the file is closed when all bytes (or lines) have been written (or an I/O error or exception is thrown). If the open options are not specified, then these methods work as if the `CREATE`, `TRUNCATE_EXISTING` and `WRITE` options are present. And they return the path of the specified file.

For example, the following code appends bytes read from file1 to file2:

```
Path file1 = Paths.get("Java Course.txt");
Path file2 = Paths.get("Java Frameworks Courses.txt");
try {
      byte[] bytes = Files.readAllBytes(file1);
      Files.write(file2, bytes, StandardOpenOption.APPEND);
} catch (IOException ex) {
      System.err.println(ex);
}
```

And the following code appends three lines of text to an existing text file:

```
Path file = Paths.get("Java Course.txt");
List< String > lines = new ArrayList< >();
lines.add("----------------------------");
lines.add("Copyright (C) by Nam Ha Minh");
lines.add("All rights Reserved");
try {
      Files.write(file, lines, StandardOpenOption.APPEND);
} catch (IOException ex) {
      System.err.println(ex);
}
```

13. Reading and Writing a File by using Buffered Stream I/O

The `Files` class provides some factory methods that return a buffered character stream (`BufferedReader` or `BufferedWriter`) for reading or writing a file in an efficient manner. These methods are:

- newBufferedReader(Path path)
- newBufferedReader(Path path, Charset cs)
- newBufferedWriter(Path path, Charset cs, OpenOption... options)
- newBufferedWriter(Path path, OpenOption... options)

For example, the following code snippet uses the `newBufferedReader()` method to read from a file using a `BufferedReader`:

```
Path file = Paths.get("ReadingFileExamples.java");
try (BufferedReader reader = Files.newBufferedReader(file)) {
      String line = null;
      while ((line = reader.readLine()) != null) {
            System.out.println(line);
      }
} catch (IOException ex) {
      System.err.println(ex);
}
```

And the following code snippet uses the `newBufferedWriter()` method to write 3 lines of text to a new file using a `BufferedWriter`:

```
Path file = Paths.get("Report.txt");
try (BufferedWriter writer = Files.newBufferedWriter(file)) {
      writer.write("Java NIO is clearer");
      writer.newLine();
      writer.write("Java NIO is more extensive");
      writer.newLine();
      writer.write("Java NIO is more convenient");
} catch (IOException ex) {
      System.err.println(ex);
}
```

14. Reading and Writing a File by Using Stream I/O

In case you want to work with byte streams (`InputStream` and `OutputStream`), use the following methods of the `Files` class:

- `newInputStream(Path path, OpenOption... options)`
- `newOutputStream(Path path, OpenOption... options)`

Note that the returned byte streams are not buffered.

For example, the following code reads the first 8 bytes (signature) of a PNG image file:

```
Path file = Paths.get("NioAPI.png");
try (InputStream inputStream = Files.newInputStream(file)) {
      byte[] signature = new byte[8];
      inputStream.read(signature);
} catch (IOException ex) {
      System.err.println(ex);
}
```

And the following code writes the first 8 bytes as signature of a PNG image file:

```
Path file = Paths.get("FileIO.png");
int[] signature = {137, 80, 78, 71, 13, 10, 26, 10};
try (OutputStream outputStream = Files.newOutputStream(file)) {
      for (int i = 0; i < signature.length; i++) {
            outputStream.write(signature[i]);
      }
} catch (IOException ex) {
      System.err.println(ex);
}
```

15. Reading and Writing a File by Using Channel I/O

In case you want to work with a channel directly, use the following method of the Files class:

- **newByteChannel**(Path path, OpenOption... options)

This method returns a `SeekableByteChannel` from a given `Path`, which is actually `FileChannel` on the default file system. Then you can read or write on this file channel. If the open option is ignored, then the file is opened for reading.

For example, the following code reads a file and prints its content to the standard output:

```
Path file = Paths.get("ByteChannelExamples.java");
try (SeekableByteChannel channel = Files.newByteChannel(file)) {
      ByteBuffer buffer = ByteBuffer.allocate(32);
      while (channel.read(buffer) != -1) {
            buffer.flip();
            while (buffer.hasRemaining()) {
                  char c = (char) buffer.get();
                  System.out.print(c);
            }
            buffer.clear();
      }
} catch (IOException ex) {
      System.err.println(ex);
}
```

And the following code opens a byte channel for writing to a file:

```
Path file = Paths.get("NIO.java");
OpenOption create = StandardOpenOption.CREATE;
OpenOption write = StandardOpenOption.WRITE;
try (SeekableByteChannel channel = Files.newByteChannel(file, create, write))
{
      String text = "import java.nio.*";
      ByteBuffer buffer = ByteBuffer.wrap(text.getBytes());
      channel.write(buffer);
} catch (IOException ex) {
      System.err.println(ex);
}
```

16. Walking the File Tree

Sometimes we need to traverse through a directory recursively to visit all sub directories and files in that directory. The Java NIO API makes it easier (and saves our time) to do that by providing the following method in the `Files` class:

```
Files.walkFileTree(Path start, FileVisitor< ? super Path > visitor)
```

You can use this method to walk recursively through the file tree of a directory for finding, copying or deleting files and directories.

This method is easy to use: you provide the starting `Path` and an implementation of a `FileVisitor` and override the following methods:

- `postVisitDirectory()`: invoked after a directory is visited.
- `preVisitDirectory()`: invoked before a directory is visited.
- `visitFile()`: invoked for a file that is visited.
- `visitFileFailed()`: invoked for a file that could not be visited.

That sounds good, and you don't have to implement all these methods if you don't need, by extending the `SimpleFileVisitor` class that already implement s these methods. Then you can override only methods you need.

The following program demonstrates how to walk the file tree of a directory and print out the names of the sub files and sub directories:

```
import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;
public class WalkFileTree extends SimpleFileVisitor< Path > {
      @Override
      public FileVisitResult visitFile(Path file,
                  BasicFileAttributes attributes) {
            System.out.format(" %s%n", file.toAbsolutePath());
            return FileVisitResult.CONTINUE;
      }
      @Override
      public FileVisitResult preVisitDirectory(Path dir,
                  BasicFileAttributes attributes) {
            System.out.format("[%s]%n", dir.toAbsolutePath());
            return FileVisitResult.CONTINUE;
      }
      public static void main(String[] args) throws IOException {
            Path start = Paths.get("e:\\Java\\JavaSE\\Swing");
            Files.walkFileTree(start, new WalkFileTree());
      }
}
```

This program prints the name of a directory inside in brackets [] and appends a white space before the name of a file.

17. Finding Files

The Java NIO API provides the **`PathMatcher`** interface (in `java.nio.file` package) which can be used to test if a given path matches a specified pattern:

`boolean PathMatcher.match(Path)`

You can use this method to write files search functionality along with the walk file tree feature. The search pattern can be a Glob or a regular expression (regex). Here’s a quick example shows you how to the `PathMatcher`:

```
String pattern = "*.{htm,html}";

PathMatcher matcher =

      FileSystems.getDefault().getPathMatcher("glob:" + pattern);

Path fileName = Paths.get("/home/john/java/nio.html").getFileName();

if (matcher.matches(fileName)) {

      System.out.println(fileName);

}
```

Here, the pattern `*.{htm, html}` matches any string ending with `.htm` or `.html`.

And the following program uses the walk file tree and `PathMatcher` together to search for Java-related files in a given path:

```
import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;
public class FindFiles extends SimpleFileVisitor {
      String pattern = "*.{java,jsp,jar,class}";
      PathMatcher matcher =
            FileSystems.getDefault().getPathMatcher("glob:" + pattern);
      @Override
      public FileVisitResult visitFile(Path file,
                  BasicFileAttributes attributes) {
            Path fileName = file.getFileName();
            if (matcher.matches(fileName)) {
                  System.out.println(file);
            }
            return FileVisitResult.CONTINUE;
      }
      public static void main(String[] args) throws IOException {
            Path start = Paths.get("e:\\Java\\JavaSE\\Swing");
            Files.walkFileTree(start, new FindFiles());
       }

}
```

When running, this program searches recursively in the given directory for files whose names ending with `.java`, `.class`, `.jar` or `.jsp`.

18. Interoperation Between Channels and Streams

The helper class `Channels` in the `java.nio.channels` package defines static methods that support the interoperation of the stream classes of the `java.io` package with the NIO channel classes. That allows you to get a channel from a byte input/output stream or a character input/output stream, and vice versa, get a byte/character input/output stream from a channel. Here’s a summary of these methods:

Construct a byte channel from an input/output stream:

- `ReadableByteChannel` **`newChannel`**`(InputStream in)`
- `WritableByteChannel` **`newChannel`**`(OutputStream out)`

Construct a byte input/output stream from a channel:

- `InputStream` **`newInputStream`**`(AsynchronousByteChannel ch)`
- `InputStream` **`newInputStream`**`(ReadableByteChannel ch)`
- **`OutputStream newOutputStream`**`(AsynchronousByteChannel ch)`
- **`OutputStream newOutputStream`**`(WritableByteChannel ch)`

Construct a character input/output stream from a channel:

- `Reader` **`newReader`**`(ReadableByteChannel ch, CharsetDecoder dec, int minBufferCap)`
- **`Reader newReader`**`(ReadableByteChannel ch, String csName)`
- `Writer` **`newWriter`**`(WritableByteChannel ch, CharsetEncoder enc, int minBufferCap)`
- **`Writer newWriter`**`(WritableByteChannel ch, String csName)`

For example, the following code reads data from a channel constructed from an input stream:

```
InputStream inputStream = new FileInputStream("NIO.java");
ReadableByteChannel channel = Channels.newChannel(inputStream);
ByteBuffer buffer = ByteBuffer.allocate(1024);
int bytesRead = 0;
while ((bytesRead = channel.read(buffer)) != -1) {
      buffer.flip();
      while (buffer.hasRemaining()) {
            char c = (char) buffer.get();
            System.out.print(c);
      }
      buffer.clear();
}
```

19. Reading and Writing File Attributes

The Java NIO also provides extensive support for reading and writing attributes for files. You can read or write a single attribute by using the specific methods of the `Files` class (to name a few):

- `boolean Files.`**`isHidden`**`(Path)`
- `FileTime` **`getLastModifiedTime`**`(Path, LinkOption…)`
- `Path` **`setLastModifiedTime`**`(Path, FileTime)`
- etc

Or by using these general methods:

- `Object` **`getAttribute`**`(Path path, String attribute, LinkOption... options)`
- `Path` **`setAttribute`**`(Path path, String attribute, Object value, LinkOption... options)`

In addition, the `Files` class provides two methods for reading file attributes in bulk:

- `Map< String, Object >` **`readAttributes`**`(Path path, String attributes, LinkOption... options)`
- `< A extends BasicFileAttributes > A` **`readAttributes`**`(Path path, Class< A > type, LinkOption... options)`

The `java.nio.file.attribute` package contains file attributes classes returned by the last `readAttributes()` method above such as `BasicFileAttributes`, `DosFileAttributes`, `PosixFileAttributes`, etc.

For example, the following code reads basic attributes of a given file:

```
Path file = Paths.get("NIO.java");
BasicFileAttributes attr = Files.readAttributes(file,
BasicFileAttributes.class);
System.out.println("creationTime: " + attr.creationTime());
System.out.println("lastAccessTime: " + attr.lastAccessTime());
System.out.println("lastModifiedTime: " + attr.lastModifiedTime());
System.out.println("isDirectory: " + attr.isDirectory());
System.out.println("isOther: " + attr.isOther());
System.out.println("isRegularFile: " + attr.isRegularFile());
System.out.println("isSymbolicLink: " + attr.isSymbolicLink());
System.out.println("size: " + attr.size());
```

The following code sets the last modified time attribute of a file to the current time:

```
FileTime time = FileTime.fromMillis(System.currentTimeMillis());
Path file = Paths.get("NIO.java");
Files.setLastModifiedTime(file, time)
```

The following code marks a file as hidden using the DOS file attribute:

```
Path file = Paths.get("NIO.java");
Files.setAttribute(file, "dos:hidden", true)
```

And the following code sets POSIX permission (read and write permission for the file owner) :

```
Path file = Paths.get("NIO.java");
Set< PosixFilePermission > permissions =
      PosixFilePermissions.fromString("rw-------");
Files.setPosixFilePermissions(file, permissions);
Note that POSIX permission works only on Unix operating systems.

20. Watching a Directory for Changes
One interesting feature of the Java NIO API is that it allows you to monitor
changes happening to a particular directory such as when a file is created,
deleted or modified. This can be implemented by creating a WatchService and
registering it to track a directory for a set of events. For example:
WatchService watcher = FileSystems.getDefault().newWatchService();
Path dir = Paths.get("Directory");
```

```
dir.register(watcher, ENTRY_CREATE, ENTRY_DELETE, ENTRY_MODIFY);
```

The following program watches a directory given from the command-line argument and it prints what file/directory has been created, deleted or modified:

```
import java.io.*;
import java.nio.file.*;
import static java.nio.file.StandardWatchEventKinds.*;
public class DirectoryWatchDemo {
    public static void main(String[] args) {
        try {
            WatchService watcher = FileSystems.getDefault().newWatchService();
            Path dir = Paths.get(args[0]);
            dir.register(watcher, ENTRY_CREATE, ENTRY_DELETE, ENTRY_MODIFY);
            while (true) {
                WatchKey key;
                try {
                    key = watcher.take();
                } catch (InterruptedException ex) {
                    return;
                }
                for (WatchEvent< ? > event : key.pollEvents()) {
                    WatchEvent.Kind< ? > kind = event.kind();
                    WatchEvent< Path > ev = (WatchEvent< Path >) event;
                    Path fileName = ev.context();
                    System.out.println(kind.name() + ": " + fileName);
                }
                boolean valid = key.reset();
                if (!valid) {
                    break;
                }
            }
        } catch (IOException ex) {
            System.err.println(ex);
        }
    }
}
```

Understanding Serialization

* Why Serialization?

Serialization is a mechanism for storing an object’s states into a persistent storage like disk files, databases, or sending object’s states over the network. The process of retrieval and construction of objects from disk files, databases and network is called de-serialization.

Here are some examples of using serialization:

- Storing data in an object-oriented way to files on disk, e.g. storing a list of `Student` objects.

- Saving program's states on disk, e.g. saving state of a game.

- Sending data over the network in form objects, e.g. sending messages as objects in chat application.

* How Serialization Works?

An object is eligible for serialization if and only if its class implements the `java.io.Serializable`interface. `Serializable`is a marker interface (contains no methods) that tell the Java Virtual Machine (JVM) that the objects of this class is ready for being written to and read from a persistent storage or over the network.

By default, the JVM takes care of the process of writing and reading serializable objects. The serialization/deserialization functionalities are exposed via the following two methods of the object stream classes:

- ObjectOutputStream.**writeObject**(Object): writes a serializable object to the output stream. This method throws `NotSerializableException` if some object to be serialized does not implement the `Serializable` interface.

- ObjectInputStream.**readObject**(): reads, constructs and returns an object from the input stream. This method throws `ClassNotFoundException` if class of a serialized object cannot be found.

Both methods throw `InvalidClassException` if something is wrong with a class used by serialization, and throw `IOException` if an I/O error occurs.
Both `NotSerializableException` and `InvalidClassException` are sub classes of `IOException.`

Let's see a quick example. The following code serializes a String object to a file named 'data.ser'. String objects are serializable because the String class implements the `Serializable`interface:

```
String filePath = "data.ser";
String message = "Java Serialization is Cool";
try (
      FileOutputStream fos = new FileOutputStream(filePath);
      ObjectOutputStream outputStream = new ObjectOutputStream(fos);
) {
      outputStream.writeObject(message);
} catch (IOException ex) {
```

```
            System.err.println(ex);
      }
```
And the following code deserializes a String object in the file ‘data.ser’:

```
      String filePath = "data.ser";
      try (
            FileInputStream fis = new FileInputStream(filePath);
            ObjectInputStream inputStream = new ObjectInputStream(fis);
      ) {
            String message = (String) inputStream.readObject();
            System.out.println("Message: " + message);
      } catch (ClassNotFoundException ex) {
            System.err.println("Class not found: " + ex);
      } catch (IOException ex) {
            System.err.println("IO error: " + ex);
      }
```
Note that the `readObject()` return an object of type `Object` so you need to cast it to the serializable class, `String` class in this case.

Let’s see a more complex example that involves in using a custom class.

Given the following `Student` class:

```
      public class Student implements Serializable {
            public static final long serialVersionUID = 1234L;
            private long studentId;
            private String name;
            private transient int age;
            public Student(long studentId, String name, int age) {
                  this.studentId = studentId;
                  this.name = name;
                  this.age = age;
                  System.out.println("Constructor");
            }
            public String toString() {
                  return String.format("%d - %s - %d", studentId, name, age);
            }
            private void foo() { }
      }
```
There are 2 new things you see in this class:

- The constant of type long `serialVersionUID`.
- The member variable `age` is marked as `transient`.

Let me explain about them right now.

* What is the serialVersionUID constant?

`serialVersionUID` is a constant that uniquely identifies a version of a serializable class. The JVM checks this constant during the deserialization process when an object is being constructed from an input stream. If the object being read has a `serialVersionUID` different than the one specified in the class, the JVM throws an `InvalidClassException`. This is to make sure that the object being constructed is compatible with its class in having the same `serialVersionUID`.

Note that the `serialVersionUID` is optional. That means the Java compiler will generate one if you don't explicitly declare it.

So why should you declare a `serialVersionUID` explicitly?

Here's the reason: The auto-generated `serialVersionUID` is calculated based on elements of the class: member variables, methods, constructors, etc. If one of these elements get change, the `serialVersionUID` will be changed as well. Imagine this situation:

- You wrote a program that stores some objects of the Student class to a file. The Student class doesn't have a `serialVersionUID` explicitly declared.
- Some times later you update the Student class (e.g. adding a new private method), and now the auto-generated `serialVersionUID` gets changed as well.
- Your program fails to deserialize the `Student` objects written previously because there `serialVersionUID` are different. The JVM throws an `InvalidClassException`.

That's why it's recommended to add a `serialVersionUID` explicitly for a serializable class.

* What is a transient variable?

In the `Student` class above, you see the member variable `age` is marked as `transient`, right? The JVM skips transient variables during the serialization process. That means the value of the `age` variable is not stored when the object is being serialized.

So if a member variable needs not to be serialized, you can mark it as transient.

The following code serializes a `Student` object to a file called 'students.ser':

```
String filePath = "students.ser";
Student student = new Student(123, "John", 22);
try (
	FileOutputStream fos = new FileOutputStream(filePath);
	ObjectOutputStream outputStream = new ObjectOutputStream(fos);
) {
	outputStream.writeObject(student);
} catch (IOException ex) {
	System.err.println(ex);
}
```

Notice that the variable `age` has the value of `22` before the object is serialized.

And the following code deserializes the `Student` object from the file:

```
String filePath = "students.ser";
try (
      FileInputStream fis = new FileInputStream(filePath);
      ObjectInputStream inputStream = new ObjectInputStream(fis);
) {
      Student student = (Student) inputStream.readObject();
      System.out.println(student);
} catch (ClassNotFoundException ex) {
      System.err.println("Class not found: " + ex);
} catch (IOException ex) {
      System.err.println("IO error: " + ex);
}
```

This code would print the following output:

```
123 - John - 0
```

You see, the value of the `age` variable is `0`, which means it is not serialized.

* More about Serialization:

There's some important information with regard to serialization you should know:

- When an object is serialized, all other objects it refers to, are serialized as well, and so on, until the complete objects tree is serialized.
- If a super class implements `Serializable`, then its sub classes do automatically.
- When an instance of a serializable class is deserialized, the constructor doesn't run.
- If a super class doesn't implement `Serializable`, then when a subclass object is deserialized, the super class constructor will run.
- Static variables are not serialized because they are not part of the object itself.
- If you serialize a collection or an array, every element must be serializable. A single non-serializable element will cause the serialization to fail (`NotSerializableException`).
- Serializable classes in JDK include primitive wrappers (`Integer`, `Long`, `Double`, etc), `String`, `Date`, collection classes… For other classes, consult relevant Javadoc to know if they are serializable.

Understanding Externalization

* Why Externalization?

You know, in serialization, the JVM takes care of the process of writing and reading serializable objects. This is good in general as you don't have to write detailed code for saving and restoring objects. But what if you want to have more control over this process, instead of relying on the default mechanism?

Imagine you have an object that carries a large amount of data and you want to compress this data during the serialization process and decompress it during the deserialization process? Or what if you want to encode your object's data for security reason? Or what if you want to save and restore objects by your own ways? In such cases, the default serialization mechanism is not sufficient. Thus Java provides an extension version called ***externalization*** that gives you full control over the process of writing and reading serializable objects, by your own ways and in your own format.

* How does Externalization Work?

To implement your own way of serialization and deserialization, make your class implements the `java.io.Externalizable` interface, and override the methods `writeExternal()` and `readExternal()` as following:

- **`writeExternal(ObjectOutput)`**: You put your own code for saving content of the object here, by calling methods of the `ObjectOutput` for writing primitive types, Strings, objects and arrays.
- **`readExternal(ObjectInput)`**: You put your own code for restoring the content of the object here, by calling methods of the `ObjectInput` for reading primitive types, Strings, objects and arrays.

Before an object is serialized, the Java Virtual Machine (JVM) checks if the class implements `Externalizable`, then invokes the `writeExternal()` method. On the reverse side, the JVM invokes the `readExternal()` method if the class is externalizable.

Not e that when an externalizable object is reconstructed, an instance is created using the public no-arg constructor, then the `readExternal()` method is called. So remember to provide a public no-arg constructor for your class. If not, you will get `java.io.InvalidClassException: no valid constructor`.

* Externalization Example:

Let's update the Student serialization example in the previous lesson: remove the field `age`, add new field `birthday` of type `Date`. Make the Student class implements the `Externalizable` interface and override the `writeExternal()` and `readExternal()` methods as below:

```
import java.io.*;
import java.util.*;
import java.text.*;
```

```
public class Student implements Externalizable {
      public static final long serialVersionUID = 1234L;
      private long studentId;
      private String name;
      private Date birthday;
      public Student() {
      }
      public Student(long studentId, String name, Date birthday) {
            this.studentId = studentId;
            this.name = name;
            this.birthday = birthday;
      }
      public void writeExternal(ObjectOutput output) {
            try {
                  output.writeLong(studentId);
                  output.writeUTF(name);
                  output.writeObject(birthday);
            } catch (IOException ex) {
                  ex.printStackTrace();
            }
      }
      public void readExternal(ObjectInput input) {
            try {
                  this.studentId = input.readLong();
                  this.name = input.readUTF();
                  this.birthday = (Date) input.readObject();
            } catch (IOException | ClassNotFoundException ex) {
                  ex.printStackTrace();
            }
      }
      public String toString() {
            DateFormat dateFormat = new SimpleDateFormat("MM-dd-yyyy");
            String info = "Id: " + this.studentId;
            info += "\nName: " + this.name;
            info += "\nBirthday: " + dateFormat.format(this.birthday);
            return info;
      }
}
```

As you can see in the `writeExternal()` method, we use the `writeLong()`, `writeUTF()` and `writeObject()` methods to save a primitive type (`long`), a `String` and an object (`Date`).

And in the `readExternal()` method, we use the `readLong()` , `readUTF()` and `readObject()` methods to restore a primitive type (`long`), a `String` and an object (`Date`).

Also note that, the read operation must read the content in the same order as the write operation does. And a public no-arg constructor is added as required.

Here’s the code to save a `Student` object to a file:

```
String filePath = "students.ser";
DateFormat dateFormat = new SimpleDateFormat("MM-dd-yyyy");
try (
      FileOutputStream fos = new FileOutputStream(filePath);
      ObjectOutputStream outputStream = new ObjectOutputStream(fos);
) {
      Date birthday = dateFormat.parse("12-23-1990");
      Student student = new Student(123, "John", birthday);
      outputStream.writeObject(student);
} catch (IOException | ParseException ex) {
      System.err.println(ex);
}
```

And the following code is for restoring the Student object from the file:

```
String filePath = "students.ser";
try (
      FileInputStream fis = new FileInputStream(filePath);
      ObjectInputStream inputStream = new ObjectInputStream(fis);
) {
      Student student = (Student) inputStream.readObject();
      System.out.println(student);
} catch (ClassNotFoundException ex) {
      System.err.println("Class not found: " + ex);
} catch (IOException ex) {
      System.err.println("IO error: " + ex);
}
```

This prints the following output:

```
Id: 123
Name: John
Birthday: 12-23-1990
```