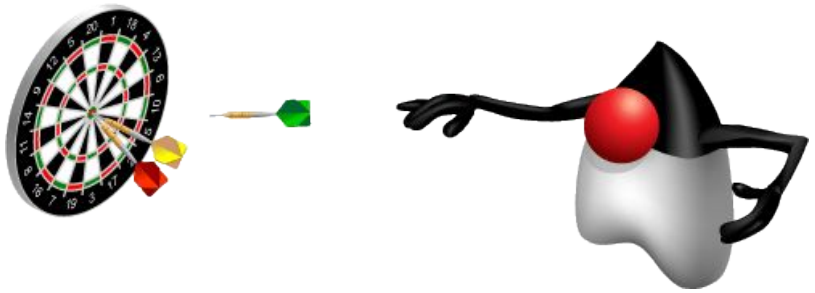
Lesson 10
Java File I/O (NIO.2)

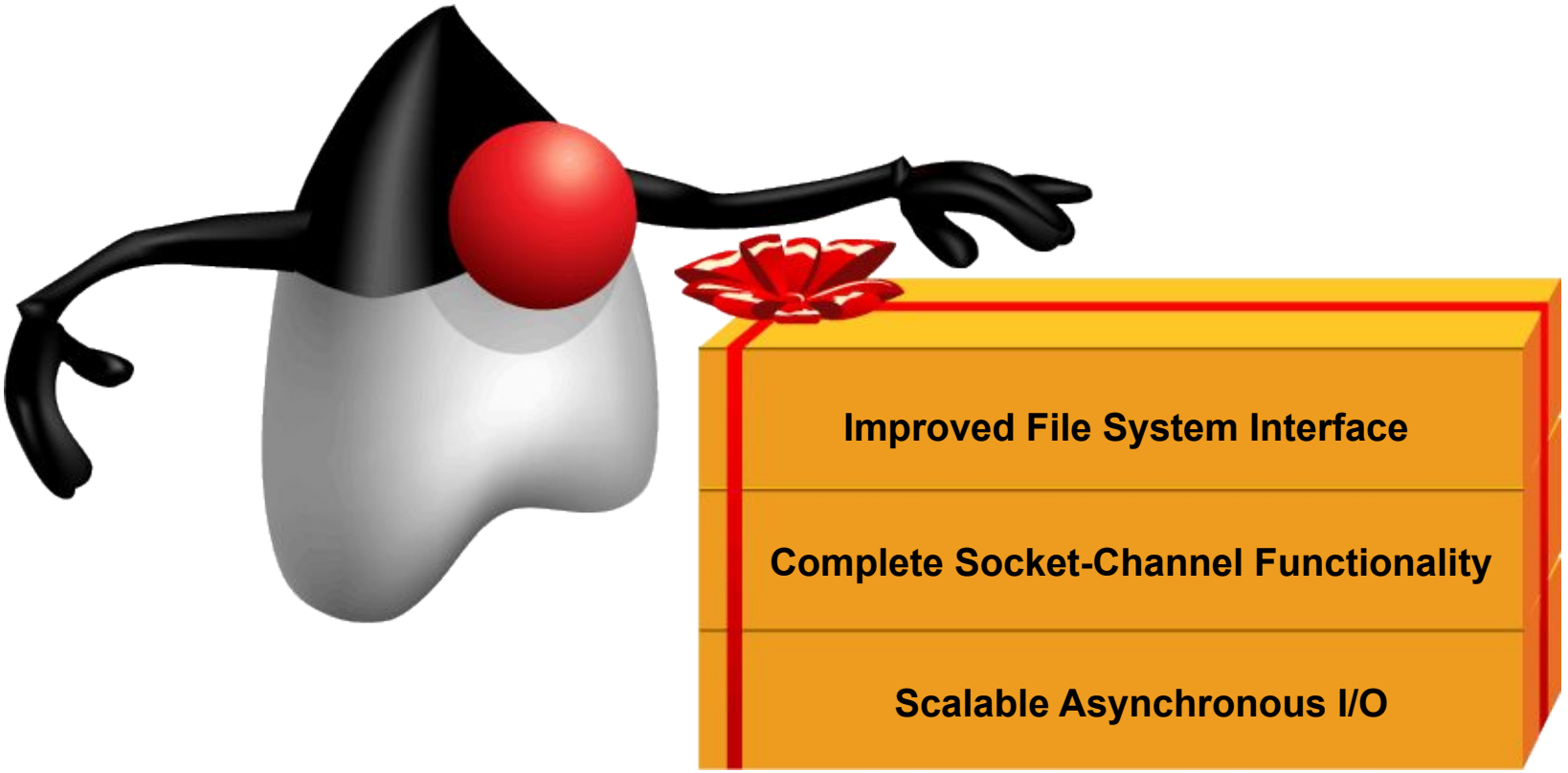
Objectives

After completing this lesson, you should be able to:

- 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 `Files` class methods to read and write files using channel I/O and stream I/O
- Read and change file and directory attributes
- Recursively access a directory tree
- Find a file by using the `PathMatcher` class

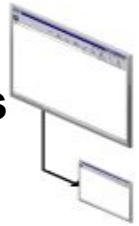


New File I/O API (NIO.2)

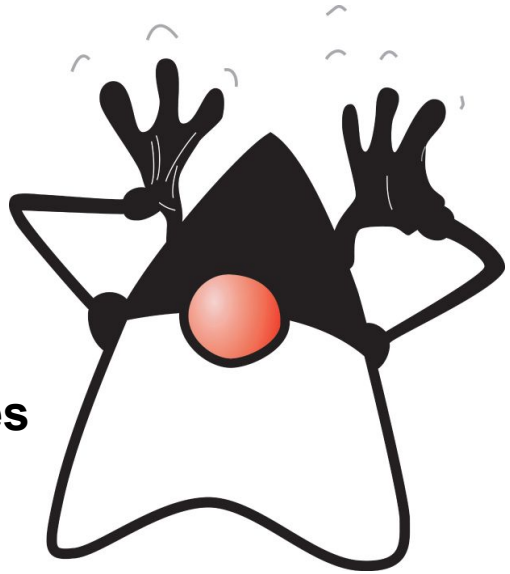


Limitations of `java.io.File`

Does not work well with symbolic links



Scalability issues



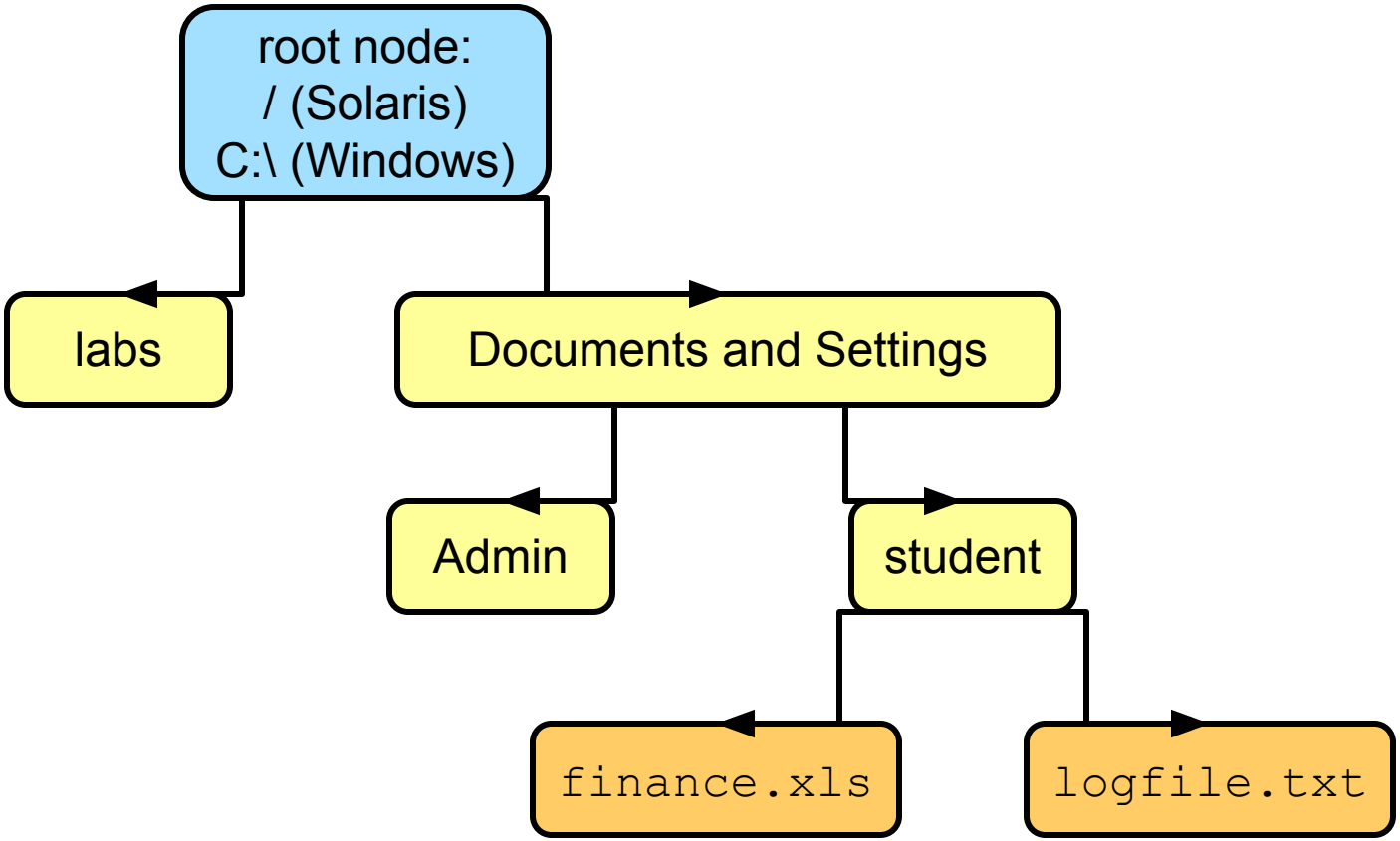
Performance issues

Very limited set of file attributes

Very basic file system access functionality

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.



Relative Path Versus Absolute Path

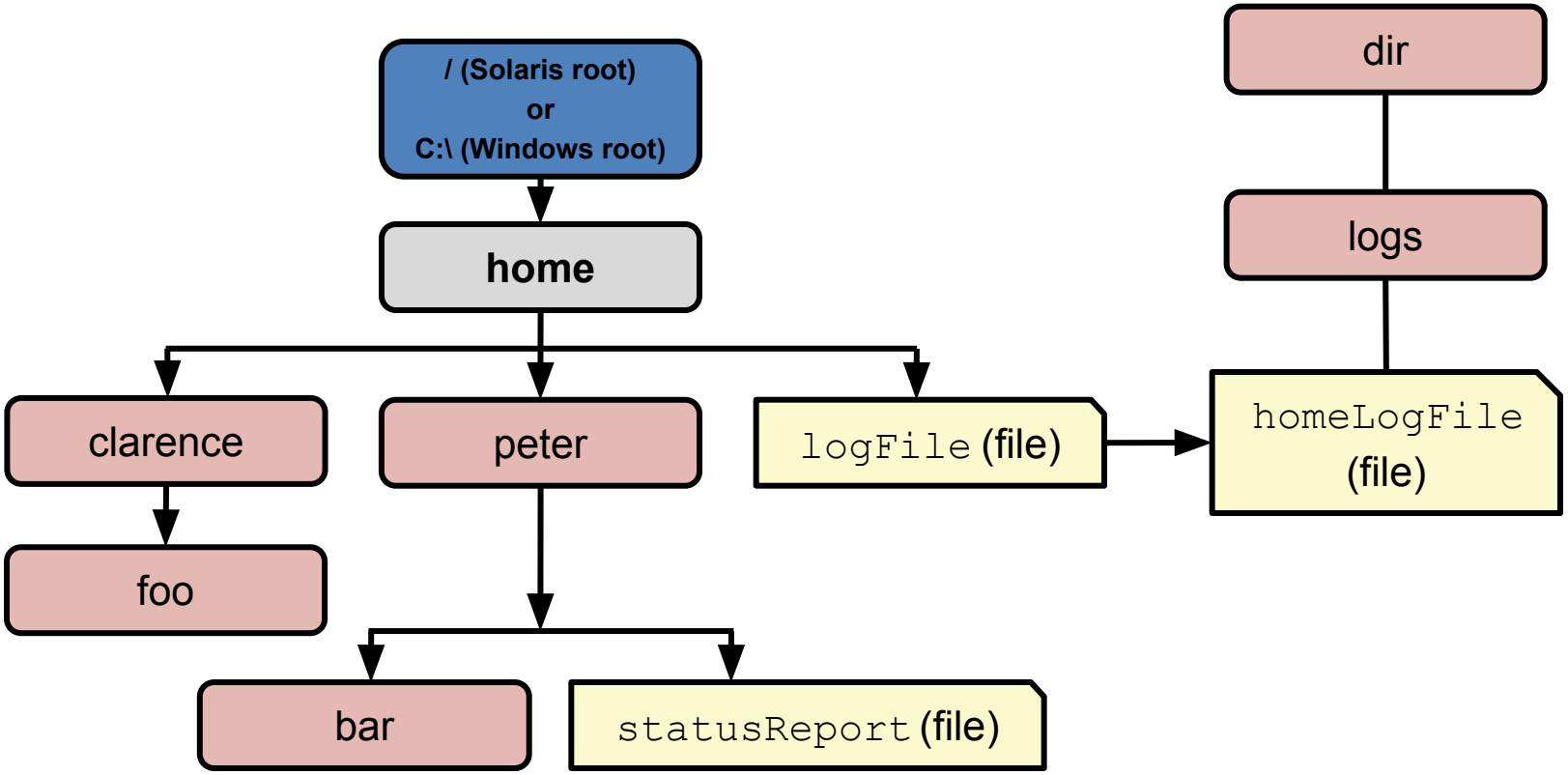
- A path is either *relative* or *absolute*.
- An absolute path always contains the root element and the complete directory list required to locate the file.

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

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

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

Symbolic Links



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.

`Path` Interface

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

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

Escaped backward slash

```
FileSystem fs = FileSystems.getDefault();
Path p1 = fs.getPath ("D:\\labs\\resources\\myFile.txt");
```

The `java.nio.file` package also provides a static final helper class called `Paths` to perform `getDefault`:

```
Path p1 = Paths.get ("D:\\labs\\resources\\myFile.txt");
Path p2 = Paths.get ("D:", "labs", "resources", "myFile.txt");
Path p3 = Paths.get ("/temp/foo");
Path p4 = Paths.get (URI.create ("file:///~/somefile");
```

`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: 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 D:/Temp/Foo/file1.txt
getFileName: file1.txt
getParent: D:\Temp\Foo
getNameCount: 3
getRoot: D:\
isAbsolute: true
toAbsolutePath: D:\Temp\Foo\file1.txt
toURI: file:///D:/Temp/Foo/file1.txt
```

Run on a Windows machine. Note that except in a `cmd` shell, forward and backward slashes are legal.

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
```

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 that the `endIndex` value.

Example:

Temp = 0
foo = 1
bar = 2

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

```
foo\bar
```

Include the element at index 2.

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

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

Creating a Path Between Two Paths

The `relativize` method enables you to construct a path from one location in the file system to another location.

The method constructs a path originating from the original path and ending at the location specified by the passed-in path.

The new path is relative to the original path.

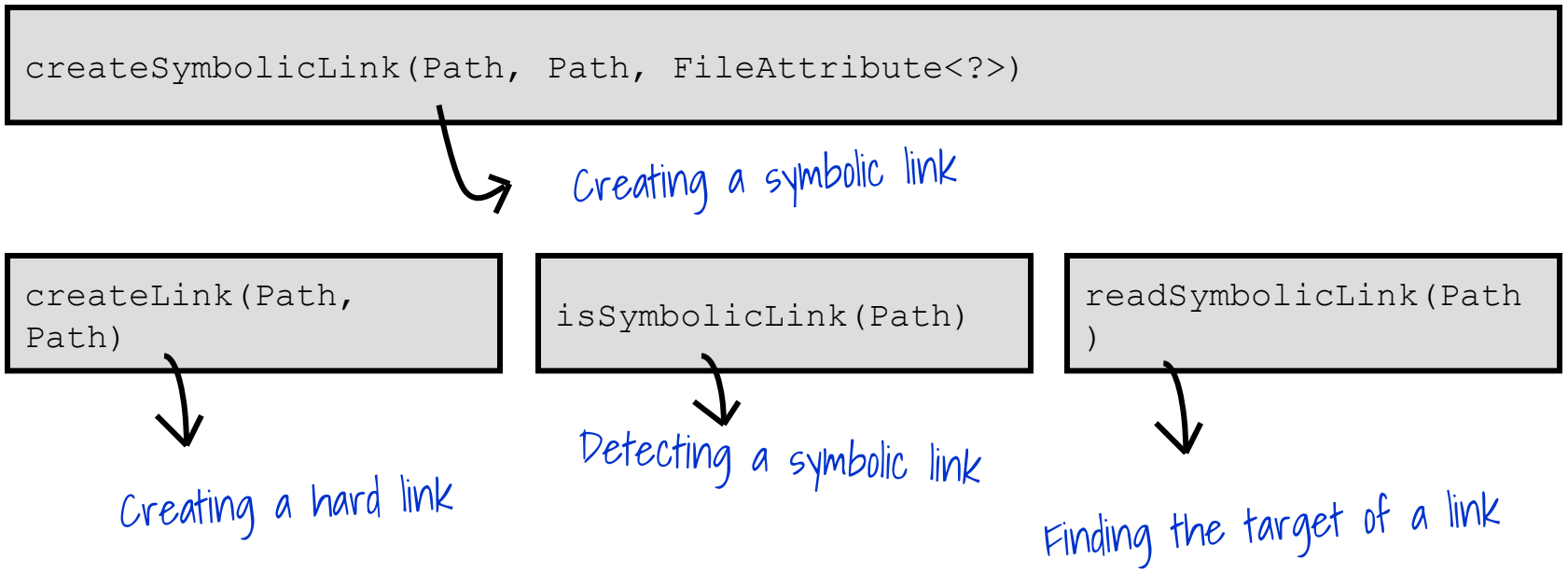
```
Path p1 = Paths.get("peter");
Path p2 = Paths.get("clarence");

Path p1Top2 = p1.relativize(p2);   // Result is ../clarence
Path p2Top1 = p2.relativize(p1);   // Result is ../peter
```

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



Quiz

Given a `Path` object with the following path:

```
/export/home/heimer/../williams/./documents
```

What `Path` method would remove the redundant elements?

a. `normalize`
b. `relativize`
c. `resolve`
d. `toAbsolutePath`

Quiz

Given the following path:

```
Path p = Paths.get
   ("/home/export/tom/documents/coursefiles/JDK7");
```

and the statement:

```
Path sub = p.subPath (x, y);
```

What values for x and y will produce a `Path` that contains

`documents/coursefiles`?

a. x = 3, y = 4
b. x = 3, y = 5
c. x = 4, y = 5
d. x = 4, y = 6

Quiz

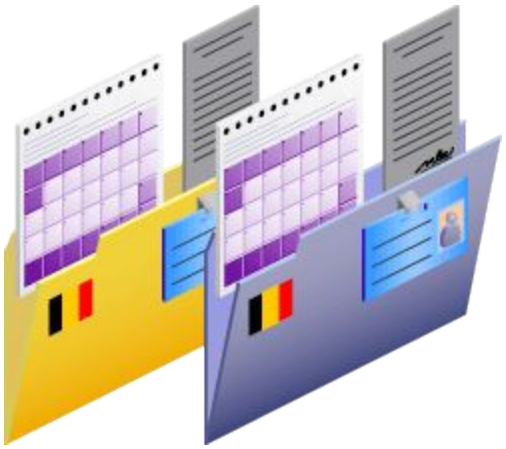
Given this code fragment:

```
Path p1 = Paths.get("D:/temp/foo/");
Path p2 = Paths.get("../bar/documents");
Path p3 = p1.resolve(p2).normalize();
System.out.println(p3);
```

What is the result?

a. Compiler error
b. IOException
c. `D:\temp\foo\documents`
d. `D:\temp\bar\documents`
e. `D:\temp\foo\..\bar\documents`

`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

Checking a File or Directory

A `Path` object represents the concept of a file or a directory location. Before you can access a file or directory, you should first access the file system to determine whether it exists using the following `Files` methods:

`exists(Path p, LinkOption... option)`
Tests to see whether a file exists. By default, symbolic links are followed.

`notExists(Path p, LinkOption... option)`
Tests to see whether a file does not exist. By default, symbolic links are followed.

Example:

```
Path p = Paths.get(args[0]);
System.out.format("Path %s exists: %b%n", p,
                  Files.exists(p, LinkOption.NOFOLLOW_LINKS));
```

Optional argument

Checking a File or Directory

To verify that a file can be accessed, the `Files` class provides the following `boolean` methods.

- `isReadable(Path)`
- `isWritable(Path)`
- `isExecutable(Path)`

Note that these tests are not atomic with respect to other file system operations. Therefore, the results of these tests may not be reliable once the methods complete.

- The `isSameFile (Path, Path)` method tests to see whether two paths point to the same file. This is particularly useful in file systems that support symbolic links.

Creating Files and Directories

Files and directories can be created using one of the following methods:

```
Files.createFile (Path dir);
Files.createDirectory (Path dir);
```

The `createDirectories` method can be used to create directories that do not exist, from top to bottom:

```
Files.createDirectories(Paths.get("D:/Temp/foo/bar/example"));
```

Deleting a File or Directory

You can delete files, directories, or links. The `Files` class provides two methods:

`delete(Path)`

```
//...
Files.delete(path);
//...
```

Throws a NoSuchFileException, DirectoryNotEmptyException, or IOException

```
//...
Files.deleteIfExists(Path)
//...
```

No exception thrown

Copying a File or Directory

You can copy a file or directory by using the `copy(Path, Path, CopyOption...)` method.

When directories are copied, the files inside the directory are not copied.

StandardCopyOption parameters

```
//...
copy(Path, Path, CopyOption...)
//...
```

REPLACE_EXISTING
COPY_ATTRIBUTES
NOFOLLOW_LINKS

Example:

```
import static java.nio.file.StandardCopyOption.*;
//...
Files.copy(source, target, REPLACE_EXISTING, NOFOLLOW_LINKS);
```

Copying Between a Stream and Path

You may also want to be able to copy (or write) from a `Stream` to file or from a file to a `Stream`. The `Files` class provides two methods to make this easy:

```
copy(InputStream source, Path target, CopyOption... options)
copy(Path source, OutputStream out)
```

An interesting use of the first method is copying from a web page and saving to a file:

```
 Path path = Paths.get("D:/Temp/oracle.html");
URI u = URI.create("http://www.oracle.com/");
try (InputStream in = u.toURL().openStream()) {
     Files.copy(in, path, StandardCopyOption.REPLACE_EXISTING);
} catch (final MalformedURLException | IOException e) {
     System.out.println("Exception: " + e);
}
```

Moving a File or Directory

You can move a file or directory by using the `move(Path, Path, CopyOption...)` method.

– Moving a directory will not move the contents of the directory.

StandardCopyOption parameters

```
//...
move(Path, Path, CopyOption...)
//...
```

REPLACE_EXISTING

ATOMIC_MOVE

– Example:

```
import static java.nio.file.StandardCopyOption.*;
//...
Files.move(source, target, REPLACE_EXISTING);
```

Listing a Directory's Contents

The `DirectoryStream` class provides a mechanism to iterate over all the entries in a directory.

```
Path dir = Paths.get("D:/Temp");
// DirectoryStream is a stream, so use try-with-resources
// or explicitly close it when finished
try (DirectoryStream<Path> stream =
                    Files.newDirectoryStream(dir, "*.zip")) {
    for (Path file : stream) {
        System.out.println(file.getFileName());
    }
} catch (PatternSyntaxException | DirectoryIteratorException |
        IOException x) {
    System.err.println(x);
}
```

- `DirectoryStream` scales to support very large directories.

Reading/Writing All Bytes or Lines from a File

- The `readAllBytes` or `readAllLines` method reads entire contents of the file in one pass.
- Example:

```
Path source = ...;
List<String> lines;
Charset cs = Charset.defaultCharset();
lines = Files.readAllLines(file, cs);
```

- Use `write` method(s) to write bytes, or lines, to a file.

```
Path target = ...;
Files.write(target, lines, cs, CREATE, TRUNCATE_EXISTING, WRITE);
```

`StandardOpenOption` enums.

Channels and `ByteBuffers`

Stream I/O reads a character at a time, while channel I/O reads a buffer at a time.

The `ByteChannel` interface provides basic read and write functionality.

A `SeekableByteChannel` is a `ByteChannel` that has the capability to maintain a position in the channel and to change that position.

The two methods for reading and writing channel I/O are:

```
newByteChannel(Path, OpenOption...)
newByteChannel(Path, Set<? extends OpenOption>,
FileAttribute<?>...)
```

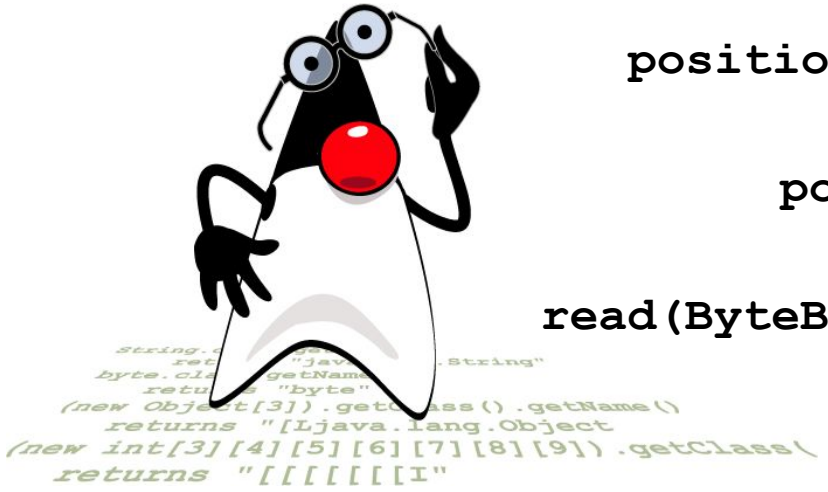
The capability to move to different points in the file and then read from or write to that location makes random access of a file possible.

Random Access Files

Random access files permit non-sequential, or random, access to a file's contents.

To access a file randomly, open the file, seek a particular location, and read from or write to that file.

Random access functionality is enabled by the `SeekableByteChannel` interface.



`position()`

`write(ByteBuffer)`

`position(long)`

`read(ByteBuffer)`

`truncate(long)`

Buffered I/O Methods for Text Files

The `newBufferedReader` method opens a file for reading.

```
//...
BufferedReader reader = Files.newBufferedReader(file, charset);
line = reader.readLine();
```

The `newBufferedWriter` method writes to a file using a `BufferedWriter`.

```
//...
BufferedWriter writer = Files.newBufferedWriter(file, charset);
writer.write(s, 0, s.length());
```

Byte Streams

NIO.2 also supports methods to open byte streams.

```
    InputStream in = Files.newInputStream(file);
    BufferedReader reader = new BufferedReader(new
InputStreamReader(in));
    line = reader.readLine();
```

To create a file, append to a file, or write to a file, use the `newOutputStream` method.

```
    import static java.nio.file.StandardOpenOption.*;
    //...
    Path logfile = ...;
    String s = ...;
    byte data[] = s.getBytes();
    OutputStream out =
           new BufferedOutputStream(file.newOutputStream(CREATE,
APPEND);
    out.write(data, 0, data.length);
```

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`	Returns or sets the specified file's last modified time
`setLastModifiedTime`	
`getAttribute`	Returns or sets the value of a file attribute
`setAttribute`	

File Attributes (DOS)

File attributes can be read from a file or directory in a single call:

```
DosFileAttributes attrs =
   Files.readAttributes (path, DosFileAttributes.class);
```

DOS file systems can modify attributes after file creation:

```
Files.createFile (file);
Files.setAttribute (file, "dos:hidden", true);
```

DOS File Attributes: Example

```
DosFileAttributes attrs = null;
Path file = ...;
try { attrs =
          Files.readAttributes(file, DosFileAttributes.class);
} catch (IOException e) { ///... }
FileTime creation = attrs.creationTime();
FileTime modified = attrs.lastModifiedTime();
FileTime lastAccess = attrs.lastAccessTime();
if (!attrs.isDirectory()) {
   long size = attrs.size();
}
// DosFileAttributes adds these to BasicFileAttributes
boolean archive = attrs.isArchive();
boolean hidden = attrs.isHidden();
boolean readOnly = attrs.isReadOnly();
boolean systemFile = attrs.isSystem();
```

POSIX Permissions

With NIO.2, you can create files and directories on POSIX file systems with their initial permissions set.

```
Path p = Paths.get(args[0]);
  Set<PosixFilePermission> perms =
      PosixFilePermissions.fromString("rwxr-x---");
  FileAttribute<Set<PosixFilePermission>> attrs =
      PosixFilePermissions.asFileAt
  try {
      Files.createFile(p, attrs);
  } catch (FileAlreadyExistsException f) {
      System.out.println("FileAlreadyExists" + f);
 } catch (IOException i) {
     System.out.println("IOException:" + i);
 }
```

Create a file in the `Path p` with optional attributes.

Quiz

Given the following fragment:

```
Path p1 = Paths.get("/export/home/peter");
Path p2 = Paths.get("/export/home/peter2");
Files.move(p1, p2, StandardCopyOption.REPLACE_EXISTING);
```

If the `peter2` directory does not exist, and the `peter` directory is populated with subfolders and files, what is the result?

- `DirectoryNotEmptyException`
- `NotDirectoryException`
- Directory `peter2` is created.
- Directory `peter` is copied to `peter2`.
- Directory `peter2` is created and populated with files and directories from `peter`.

Quiz

Given this fragment:

```
Path source = Paths.get(args[0]);
Path target = Paths.get(args[1]);
Files.copy(source, target);
```

Assuming `source` and `target` are not directories, how can you prevent this copy operation from generating `FileAlreadyExistsException`?

a. Delete the `target` file before the copy.
b. Use the `move` method instead.
c. Use the `copyExisting` method instead.
d. Add the `REPLACE_EXISTING` option to the method.

Quiz

Given this fragment:

```
Path source =
   Paths.get("/export/home/mcginn/HelloWorld.java");
Path newdir = Paths.get("/export/home/heimer");
Files.copy(source, newdir.resolve(source.getFileName());
```

Assuming there are no exceptions, what is the result?

a. The contents of `mcginn` are copied to `heimer`.
b. `HelloWorld.java` is copied to `/export/home`.
c. `HelloWorld.java` is coped to `/export/home/heimer`.
d. The contents of `heimer` are copied to `mcginn`.

Recursive Operations

The `Files` class provides a method to walk the file tree for recursive operations, such as copies and deletes.

`walkFileTree (Path start, FileVisitor<T>)`

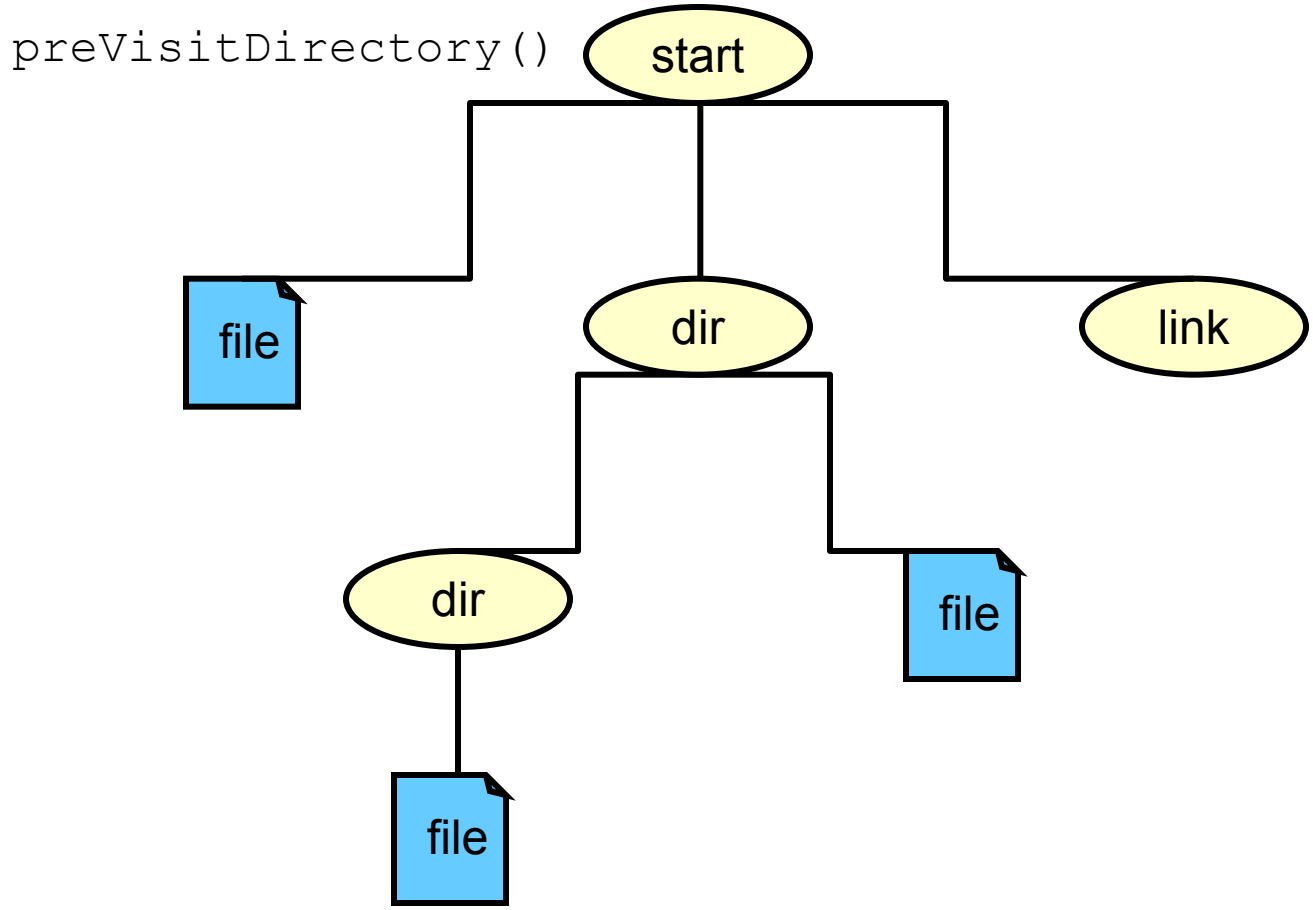
Example:

```
public class PrintTree implements FileVisitor<Path> {
    public FileVisitResult preVisitDirectory(Path, BasicFileAttributes){}
    public FileVisitResult postVisitDirectory(Path, BasicFileAttributes){}
    public FileVisitResult visitFile(Path, BasicFileAttributes){}
    public FileVisitResult visitFileFailed(Path, BasicFileAttributes){}
}
```

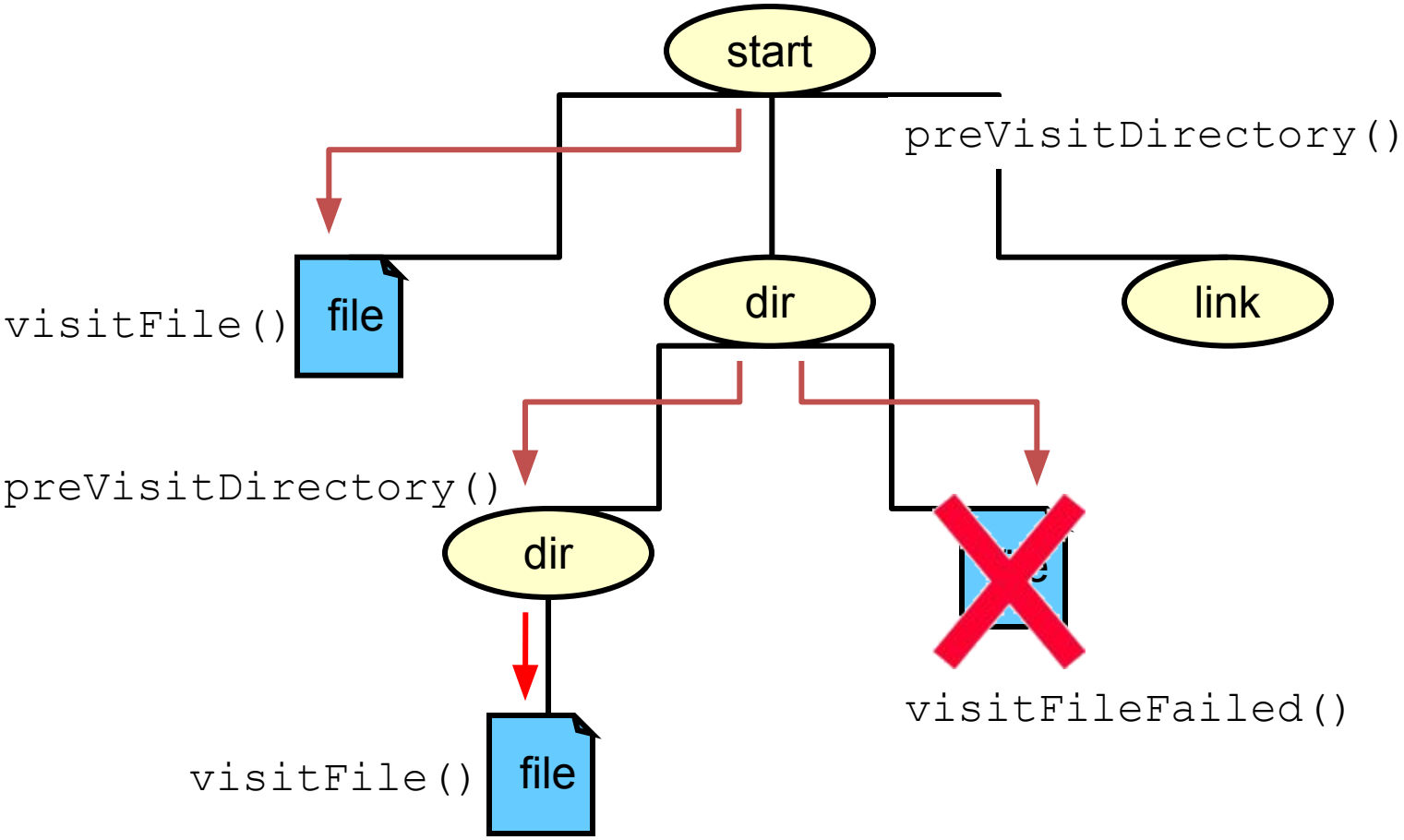
```
public class WalkFileTreeExample {
    public printFileTree(Path p) {
        Files.walkFileTree(p, new PrintTree());
    }
}
```

The file tree is recursively explored. Methods defined by `PrintTree` are invoked as directories and files are reached in the tree. Each method is passed the current path as the first argument of the method.

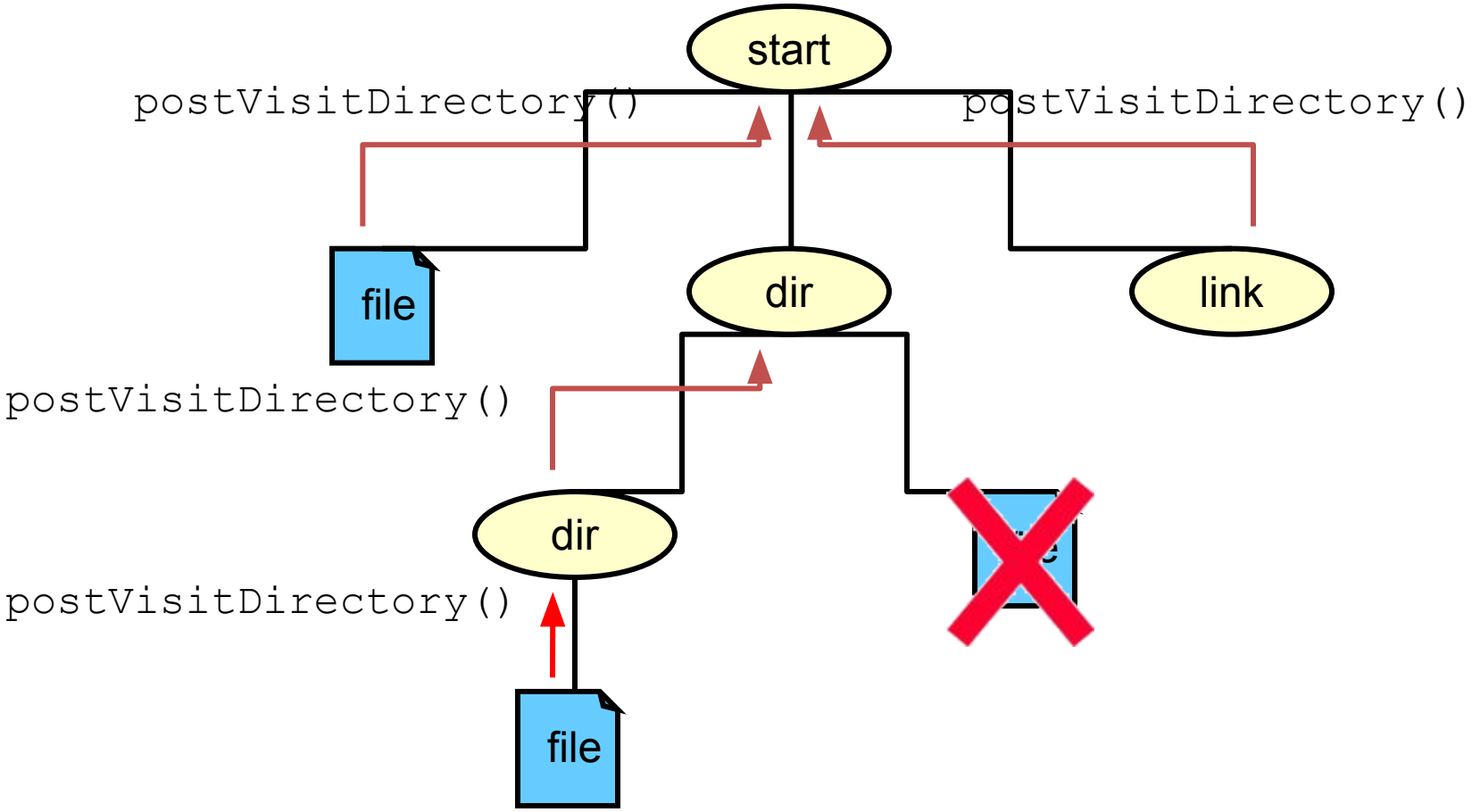
`FileVisitor` Method Order



`FileVisitor` Method Order

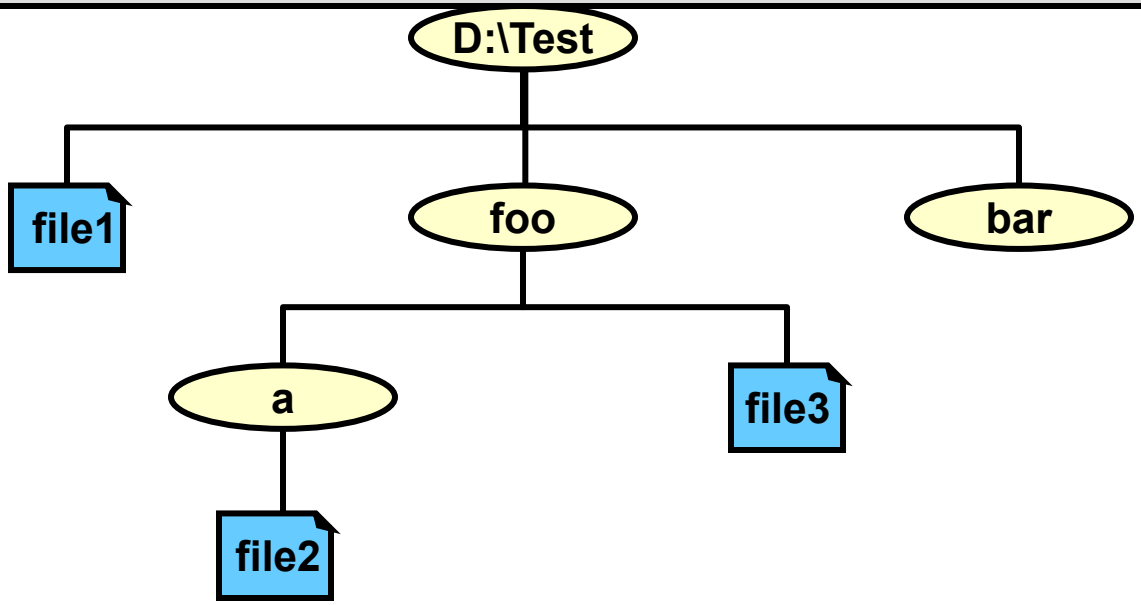


`FileVisitor` Method Order



Example: WalkFileTreeExample

```
Path path = Paths.get("D:/Test");
try {
    Files.walkFileTree(path, new PrintTree());
} catch (IOException e) {
    System.out.println("Exception: " + e);
}
```



Finding Files

To find a file, typically, you would search a directory. You could use a search tool, or a command, such as:

```
dir /s *.java
```

This command will recursively search the directory tree, starting from where you are for all files that contain the `java` extension.

The `java.nio.file.PathMatcher` interface includes a match method to determine whether a `Path` object matches a specified search string.

Each file system implementation provides a `PathMatcher` that can be retrieved by using the `FileSystems` factory:

```
PathMatcher matcher = FileSystems.getDefault().getPathMatcher
   (String syntaxAndPattern);
```

PathMatcher Syntax and Pattern

- The `syntaxAndPattern` string is of the form:

  `syntax:pattern`

  Where *`syntax`* can be “glob” and “regex”.
- The glob syntax is similar to regular expressions, but simpler:

Pattern Example	Matches
`*.java`	A path that represents a file name ending in `.java`
`*.*`	Matches file names containing a dot
`*.{java,class}`	Matches file names ending with `.java` or `.class`
`foo.?`	Matches file names starting with `foo.` and a single character extension
`C:\\*`	Matches `C:\foo` and `C:\bar` on the Windows platform (Note that the backslash is escaped. As a string literal in the Java Language, the pattern would be `C:\\\\*`.)

PathMatcher: Example

```
public static void main(String[] args) {
    // ... check for two arguments
    Path root = Paths.get(args[0]);
    // ... check that the first argument is a directory
    PathMatcher matcher =
        FileSystems.getDefault().getPathMatcher("glob:" + args[1]);
    // Finder is class that implements FileVisitor
    Finder finder = new Finder(root, matcher);
    try {
        Files.walkFileTree(root, finder);
    } catch (IOException e) {
        System.out.println("Exception: " + e);
    }
    finder.done();
}
```

Finder Class

```
public class Finder extends SimpleFileVisitor<Path> {
    private Path file;
    private PathMatcher matcher;
    private int numMatches;
    // ... constructor stores Path and PathMatcher objects
    private void find(Path file) {
        Path name = file.getFileName();
        if (name != null && matcher.matches(name)) {
            numMatches++;
            System.out.println(file);
        }
    }
    @Override
    public FileVisitResult visitFile(Path file,
                                     BasicFileAttributes attrs) {
        find(file);
        return CONTINUE;
    }
    //...
}
```

Other Useful NIO.2 Classes

The `FileStore` class is useful for providing usage information about a file system, such as the total, usable, and allocated disk space.

```
Filesystem                  kbytes          used         avail
System (C:)              209748988      72247420     137501568
Data (D:)                 81847292        429488      81417804
```

An instance of the `WatchService` interface can be used to report changes to registered `Path` objects. `WatchService` can be used to identify when files are added, deleted, or modified in a directory.

```
ENTRY_CREATE: D:\test\New Text Document.txt
ENTRY_CREATE: D:\test\Foo.txt
ENTRY_MODIFY: D:\test\Foo.txt
ENTRY_MODIFY: D:\test\Foo.txt
ENTRY_DELETE: D:\test\Foo.txt
```

Moving to NIO.2

A method was added to the `java.io.File` class for JDK 7 to provide forward compatibility with NIO.2.

```
Path path = file.toPath();
```

- This enables you to take advantage of NIO.2 without having to rewrite a lot of code.
- Further, you could replace your existing code to improve future maintenance—for example, replace `file.delete();` with:

```
Path path = file.toPath();
Files.delete (path);
```

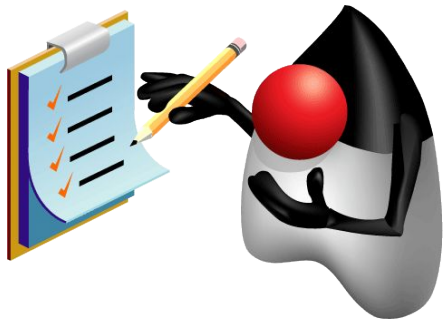
- Conversely, the `Path` interface provides a method to construct a `java.io.File` object:

```
File file = path.toFile();
```

Summary

In this lesson, you should have learned how to:

- 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 `Files` class methods to read and write files using channel I/O and stream I/O
- Read and change file and directory attributes
- Recursively access a directory tree
- Find a file by using the `PathMatcher` class



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`

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 an application where you want to count the depth of a file tree (how many levels of directories), which FileVisitor method should you use?

a. `preVisitDirectory()`
b. `postVisitDirectory()`
c. `visitFile()`
d. `visitDirectory()`