

15

Java File IO (NIO)

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 Stream API with NIO2



New File I/O API (NIO.2)



Limitations of `java.io.File`



Scalability issues



Performance issues



Very basic file system access functionality

Does not work well with symbolic links

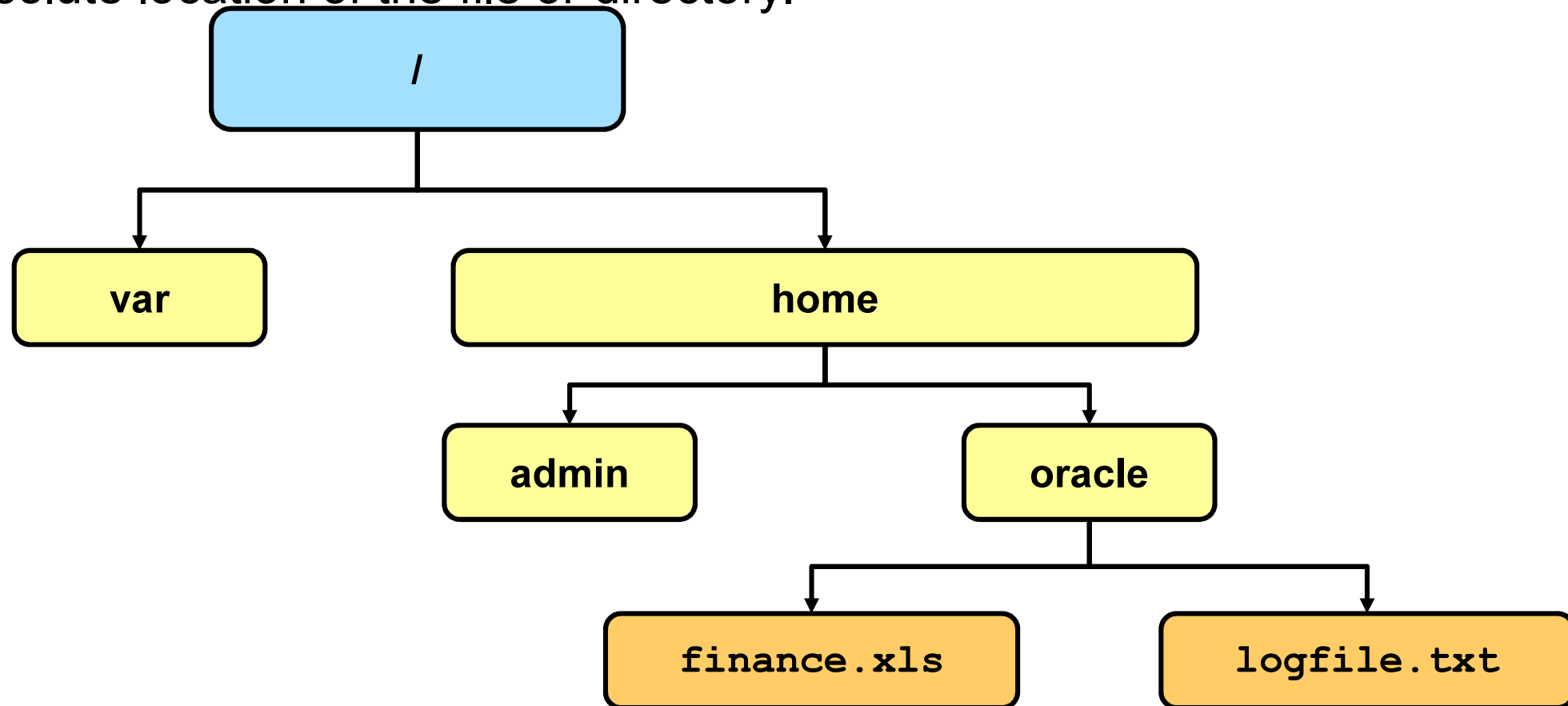


Very limited set of file attributes



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  
...
```

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.

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

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

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


Path Interface Features

- The `Path` interface defines the methods used to locate a file or a directory in a file system. These methods include:
 - To access the components of a path:
 - `getFileName`, `getParent`, `getRoot`, `getNameCount`
 - To operate on a path:
 - `normalize`, `toUri`, `toAbsolutePath`, `subpath`, `resolve`, `relativize`
 - To compare paths:
 - `startsWith`, `endsWith`, `equals`

Path: Example

```
1 public class PathTest
2     public static void main(String[] args) {
3         Path p1 = Paths.get(args[0]);
4         System.out.format("getFileName: %s\n", p1.getFileName());
5         System.out.format("getParent: %s\n", p1.getParent());
6         System.out.format("getNameCount: %d\n", p1.getNameCount());
7         System.out.format("getRoot: %s\n", p1.getRoot());
8         System.out.format("isAbsolute: %b\n", p1.isAbsolute());
9         System.out.format("toAbsolutePath: %s\n", p1.toAbsolutePath());
10        System.out.format("toURI: %s\n", p1.toUri());
11    }
12 }
```

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

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 than the `endIndex` value.
- Example:

home= 0
oracle = 1
Temp = 2

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

oracle/Temp

Include the element at index 2.

Joining Two Paths

➤ The `resolve` method is used to combine two paths.

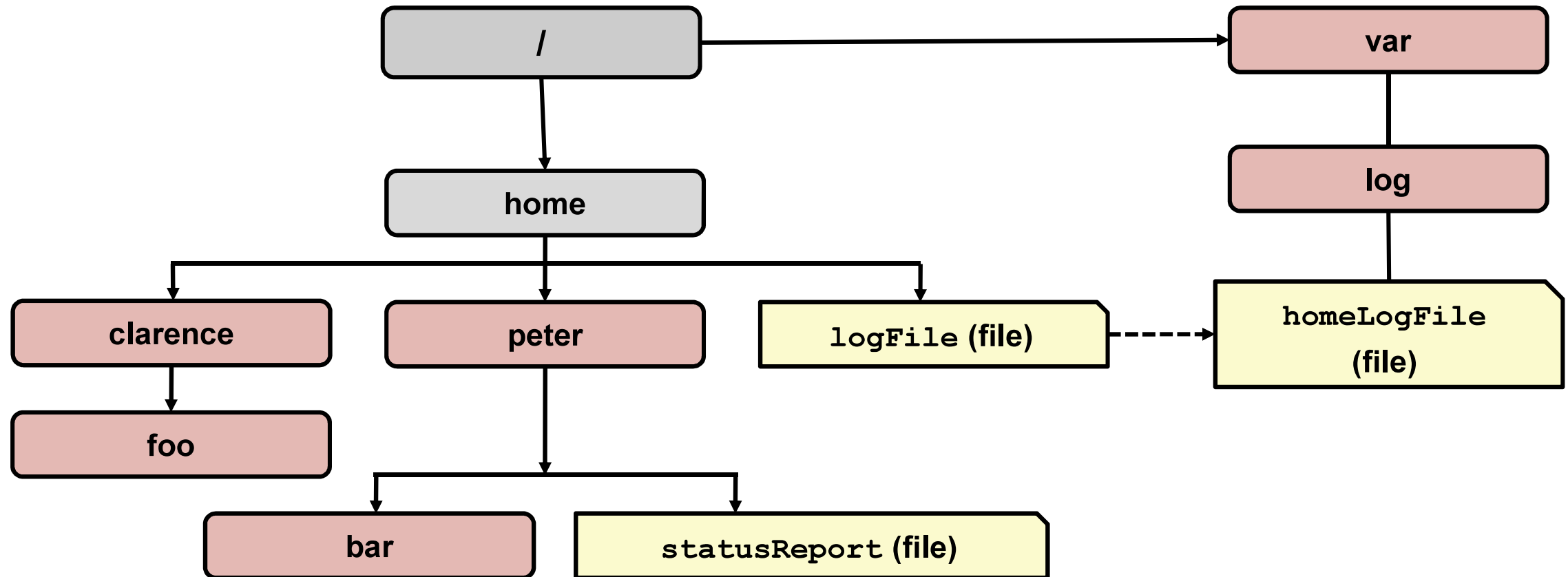
➤ Example:

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

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

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

Symbolic Links



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

```
createSymbolicLink(Path, Path, FileAttribute<?>)
```

Creating a symbolic link

```
createLink(Path, Path)
```

Creating a hard link

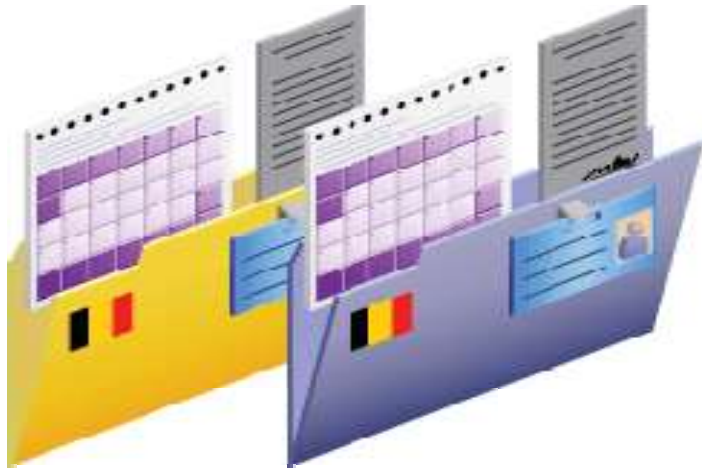
```
isSymbolicLink(Path)
```

Detecting a symbolic link

```
readSymbolicLink(Path)
```

Finding the target of a link

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);
```

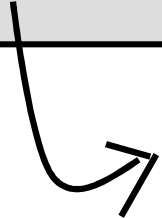
- The `createDirectories` method can be used to create directories that do not exist, from top to bottom:

```
• Files.createDirectories (Paths.get ("/home/oracle/Temp/foo/bar/example")) ;
```

Deleting a File or Directory

- You can delete files, directories, or links. The `Files` class provides two methods:
 - `delete(Path)`
 - `deleteIfExists(Path)`

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



Throws a `NoSuchFileException`,
`DirectoryNotEmptyException`, or
`IOException`

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

No exception thrown

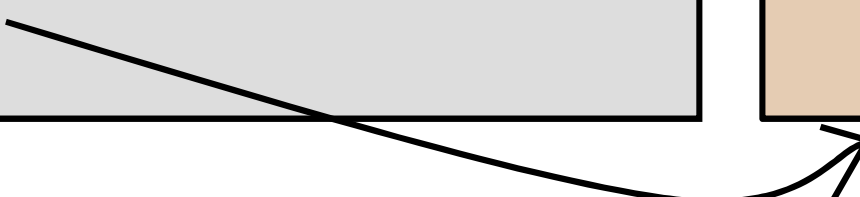
Copying a File or Directory

- 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);
```

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);
```

List the Contents of a Directory

To get a list of the files in the current directory, use the `Files.list()` method.

```
public class FileList {  
  
    public static void main(String[] args) {  
  
        try(Stream<Path> files = Files.list(Paths.get("."))) {  
  
            files  
                .forEach(line -> System.out.println(line));  
  
        } catch (IOException e) {  
            System.out.println("Message: " + e.getMessage());  
        }  
    }  
}
```

Walk the Directory Structure

The `Files.walk()` method walks a directory structure.

```
public class AllFileWalk {  
  
    public static void main(String[] args) {  
  
        try(Stream<Path> files = Files.walk(Paths.get("."))) {  
  
            files  
                .forEach(line -> System.out.println(line));  
  
        } catch (Exception e){  
            System.out.println("Message: " + e.getMessage());  
        }  
    }  
}
```


BufferedReader File Stream

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

```
public class BufferedRead {  
    public static void main(String[] args) {  
        try(BufferedReader bReader =  
            new BufferedReader(new FileReader("tempest.txt"))){  
  
            bReader.lines()  
                .forEach(line ->  
                    System.out.println("Line: " + line));  
  
        } catch (IOException e){  
            System.out.println("Message: " + e.getMessage());  
        }  
    }  
}
```

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

```
public class ReadNio {  
  
    public static void main(String[] args) {  
  
        try(Stream<String> lines =  
            Files.lines(Paths.get("tempest.txt"))){  
  
            lines.forEach(line ->  
                System.out.println("Line: " + line));  
  
        } catch (IOException e){  
            System.out.println("Error: " + e.getMessage());  
        }  
    }  
}
```

Read File into ArrayList

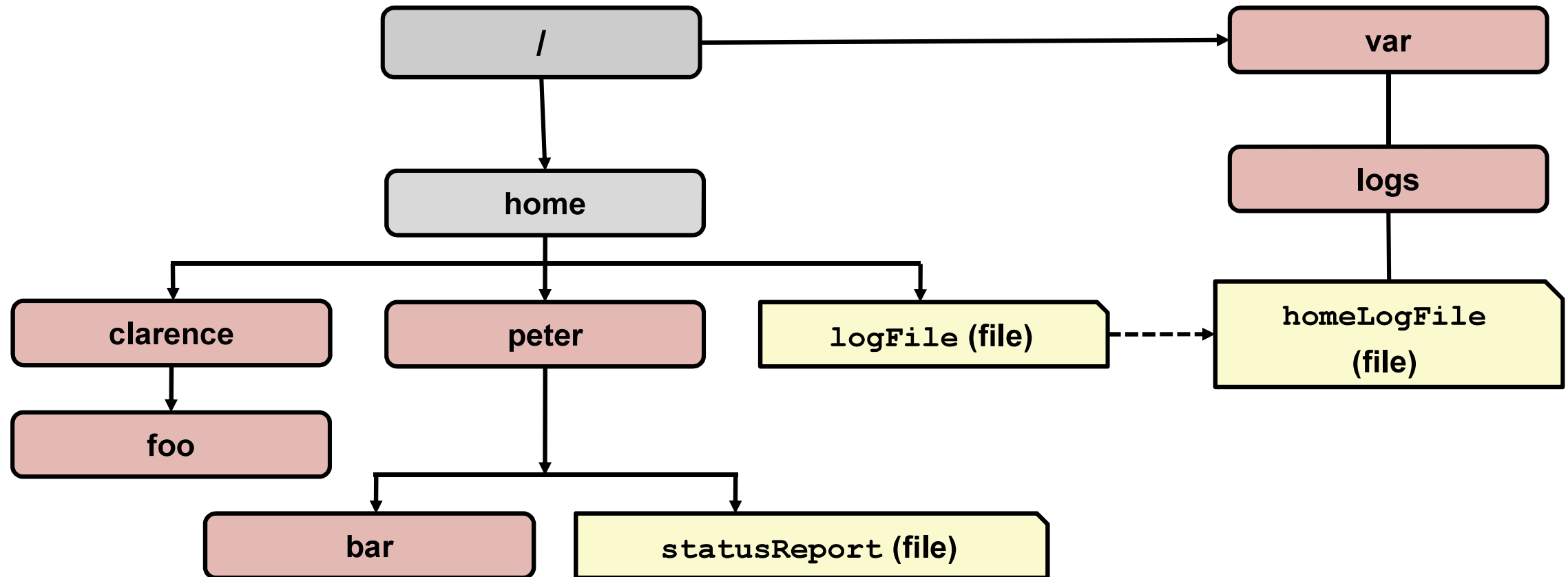
Use `readAllLines()` to load a file into an `ArrayList`.

```
public class ReadAllNio {  
    public static void main(String[] args) {  
        Path file = Paths.get("tempest.txt");  
        List<String> fileArr;  
  
        try{  
  
            fileArr = Files.readAllLines(file);  
  
            fileArr.stream()  
                .filter(line -> line.contains("PROSPERO"))  
                .forEach(line -> System.out.println(line));  
  
        } catch (IOException e){  
            System.out.println("Message: " + e.getMessage());  
        }  
    }  
}
```

Managing Metadata

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

Symbolic Links



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 Stream API with NIO2



Quiz

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

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

Quiz

Given a `Path` object with the following path:

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

What `Path` method would remove the redundant elements?

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

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?

- a. `DirectoryNotEmptyException`
- b. `NotDirectoryException`
- c. Directory `peter2` is created.
- d. Directory `peter` is copied to `peter2`.
- e. 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

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