8 Java file I/O (NIO.2)

Exam objectives covered in this chapter	What you need to know
[8.1] Operate on file and directory paths with the `Path` class	How to create and manipulate real, absolute, and symbolic paths to files and directories.
[8.2] Check, delete, copy, or move a file or directory with the `Files` class	How to work with class `Files` and `Path` objects to create files and directories, check for their existence, and delete, copy, and move them.
[8.3] Read and change file and directory attributes, focusing on the `BasicFileAttributes`, `DosFileAttributes`, and `PosixFileAttributes` interfaces	How to use class `Files` to access and modify the individual and group of file and directory attributes—namely, `basic`, `dos`, and `posix`. How to access or modify attributes that aren't supported by the underlying operating system.
[8.4] Recursively access a directory tree using the `DirectoryStream` and `FileVisitor` interfaces	How to use `DirectoryStream`, `FileVisitor`, and `SimpleFileVisitor` to walk a directory tree. How to define code that should execute when a file is visited, before and after a directory is visited.
[8.5] Find a file with the `PathMatcher` interface	How to use `PathMatcher` to find directory names and filenames that match a regex or glob pattern.
[8.6] Watch a directory for changes by using the `WatchService` interface	How to set a watch on a directory for creation, modification, and deletion of elements from it. How to listen to these events and define code that should execute when an event occurs.

Think of any real-world application—almost all of them need to communicate with the file system to store, access, or manipulate their data or configuration values. Prior to Java 7, file system access and management has always been full of challenges. Often developers used native code to work with simple or advanced operations like copying or moving files atomically, querying their attributes, monitoring changes in a folder, or traversing a directory structure. But Java applications with native code lose platform independence.

Released with Java 7, NIO.2 (New Input/Output version 2) extensively improved the existing I/O capabilities and added new ones. Though NIO was introduced with Java 1.4, Java 7 further extended the NIO API, which resulted in a name change, now called NIO.2. NIO.2 adds new file access and management functionality, such as defining new classes and interfaces to access files and file systems, accessing their metadata, the ability to traverse file trees, networking with the socket API, Asynchronous Channel API, WatchService API, the ability to support a new file system, and more.

Though Java NIO.2 defines a lot of functionality, the good news is that the exam doesn't include all of it. This chapter covers the NIO.2 topics that are relevant to the exam:

- Using the `Path` interface
- Using class `Files`
- Working with file and directory attributes
- Walking a directory tree using the `DirectoryStream` and `FileVisitor` interfaces
- Using the `PathMatcher` interface
- Using the WatchService API

The topics covered in this chapter include coverage of a lot of classes from the Java NIO.2 that deal with working with paths, files, and their attributes, and walking and watching directory structures and more. This boils down to using a *lot* of Java API classes and their methods, which can become boring. So let's make it interesting and fun. Let's work with building a sample application, Exam FlashCards, to help you apply most of the exam topics covered in this chapter.

In this application, you'll read and write your favorite exam tips and notes from and to text files on your system. The notes will be organized according to the main objective and subobjective numbers. Each main exam objective will map to a directory and the tips or notes for a main exam objective will be stored in a text file, on separate lines. Figure 8.1 shows the file and directory hierarchy for text files 8-1.txt, 8-2.txt, and 8-3.txt.

In this application, let's start with storing exam notes in a file. This application uses Swing components to create a simple UI for accepting the exam objective number, the subobjective, and notes to the file. Play around with creating directories and files using the application, and add notes and check your physical files. Figure 8.2 shows a screenshot of the application.

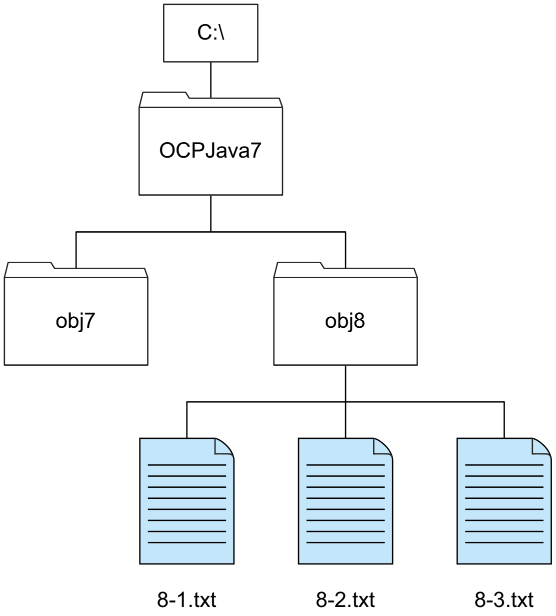
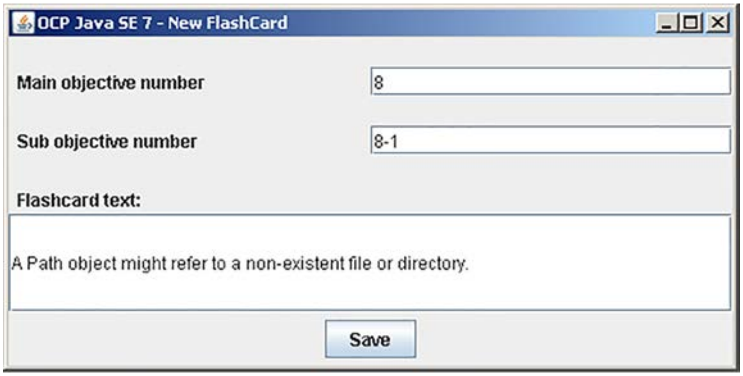


Figure 8.1 File and hierarchy directory for text files 8-1.txt, 8-2.txt, and 8-3.txt



Main objective number: 8

Sub objective number: 8-1

Flashcard text:

A Path object might refer to a non-existent file or directory.

Figure 8.2 Sample application to add a new flash card

Listing 8.1 shows the code for figure 8.1. Don't worry if you can't follow all the code—it uses Swing components, which aren't on the exam. I've annotated the code so that it's easier to follow.

Listing 8.1 Building sample application's UI

```
import java.io.*;
import java.nio.file.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

```
public class NewFlashCard implements ActionListener {
    JFrame f = new JFrame("OCP Java SE 7 - New FlashCard");
    JTextField tfMainObj = null;
    JTextField tfSubObj = null;
    JTextField tfNote = null;
    JButton btnSave = null;
    JButton btnClear = null;
    JButton btnExit = null;

    private void buildUI() {
        tfMainObj = new JTextField();
        tfSubObj = new JTextField();
        tfNote = new JTextField();
        btnSave = new JButton("Save");
        btnSave.addActionListener(this);

        JPanel topPanel = new JPanel();
        topPanel.setLayout(new GridLayout(6,2));
        topPanel.add(new JLabel(""));
        topPanel.add(new JLabel(""));

        topPanel.add(new JLabel("  Main objective number"));
        topPanel.add(tfMainObj);
        topPanel.add(new JLabel(""));
        topPanel.add(new JLabel(""));
        topPanel.add(new JLabel("  Sub objective number"));
        topPanel.add(tfSubObj);
        topPanel.add(new JLabel(""));
        topPanel.add(new JLabel(""));
        topPanel.add(new JLabel("  Flashcard text:"));
        topPanel.add(new JLabel(""));

        JPanel middlePanel = new JPanel();
        middlePanel.setLayout(new BorderLayout());
        middlePanel.add(tfNote);

        JPanel bottomPanel = new JPanel();
        bottomPanel.add(btnSave);

        JPanel mainPanel = new JPanel();
        mainPanel.setLayout(new BorderLayout());
        mainPanel.add(BorderLayout.NORTH, topPanel);
        mainPanel.add(BorderLayout.CENTER, middlePanel);
        mainPanel.add(BorderLayout.SOUTH, bottomPanel);

        f.getContentPane().setLayout(new BorderLayout());
        f.setSize(500, 250);
        f.getContentPane().add(mainPanel);
        f.setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        // Code to execute when Save button is activated
    }
```

Textfields to accept input for main and subobjective numbers and flash card text

Button to initiate saving of text to file

Initialize text fields

Initialize button and add ActionListener to it

Build UI

Code in actionPerformed will execute when user activates Save button

```
    public static void main(String[] args) {
        NewFlashCard nfc = new NewFlashCard();
        nfc.buildUI();
    }
}
```

Create NewFlashCard instance and build the UI

In this application, the main objective number corresponds to a directory and the sub-objective number corresponds to a text file, to which flash card text is written. You need to follow these steps to write an exam note to a file:

1. Get a `Path` referring to file /8/8-1.txt.
2. Create a corresponding file or directories if they don't already exist.
3. Open the file in append mode and copy the flash card text to the file.

Before you can move forward with the first step of creating a `Path` object, let's discuss `Path` objects in detail.

8.1 Path objects

[8.1] Operate on file and directory paths with the Path class

Objects of the `java.nio.file.Path` interface are used to *represent* the *path* of files or directories in a file system. The interface represents a hierarchal path containing directory names and filenames separated by platform-specific delimiters. A path can contain multiple directories, a single filename, or both.

NOTE Prior to Java 7, class `java.io.File` was used to represent the path of a file or directory in a file system. But it had several drawbacks. Its methods didn't throw exceptions when they failed, which was essential to determine the cause of failure. Most of the methods of class `File` didn't scale. For example, a request to a large directory listing could make a system hang. It didn't support much access to the metadata. These reasons and more were responsible for the introduction of `Path` in Java 7.

Figure 8.3 shows a sample directory tree in both a Windows and a Unix OS, with a root node, directories, and files.

In figure 8.3 (Windows OS), the path to file Hello.txt can be represented using the `Path` interface both as c:\users\harry\Hello.txt (absolute path) and as Hello.txt (relative path). An absolute path includes the complete path from the root directory to the file or directory. Because a path is system-specific, the paths that a `Path` object represent are platform-dependent. It's interesting to note that an object of `Path` might not be tied to a real file or directory on a system. The `Path` interface defines methods to work with and to manipulate `Path` objects, resolving one path to another. But it doesn't contain methods to work with the actual physical directories and files that a `Path` object refers to. To work with the actual files and directories, you

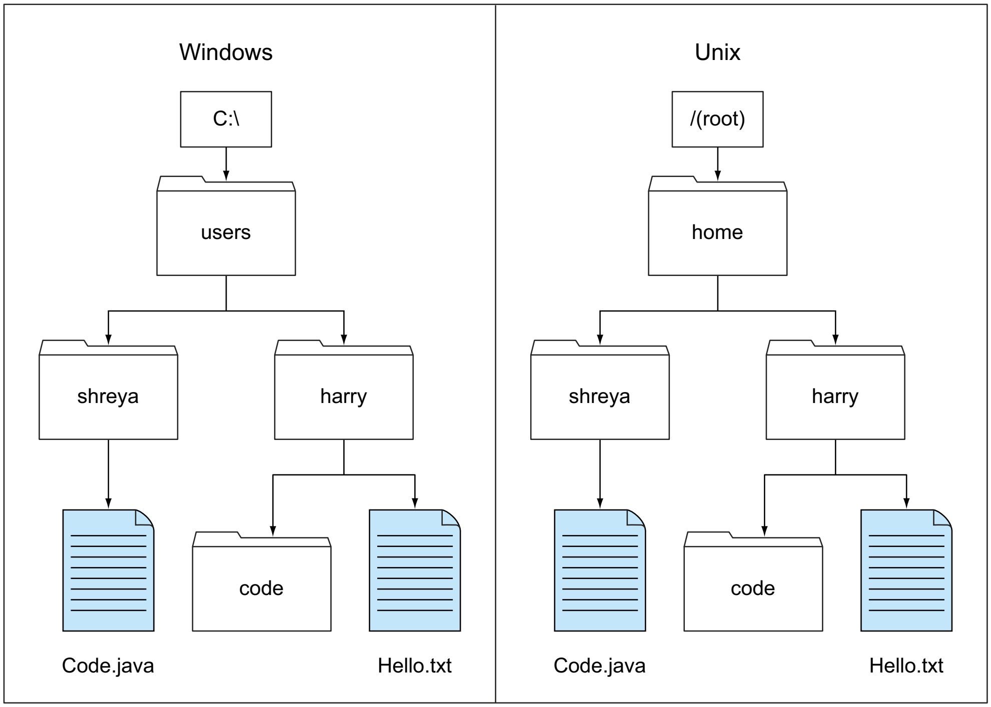


Figure 8.3 Sample directory tree structure on Windows and Unix

can use `java.nio.file.Files`. We'll use the directory structure shown in figure 8.1 to show you how to work with `Path` objects.

EXAM TIP Because a `Path` object might not be tied to a real file or directory on a system, it can refer to a nonexistent file or directory.

Apart from referring to a file and a directory, a `Path` object can also refer to a *symbolic link*. A symbolic link is a special file that refers to another file. The file referred to by a symbolic link is called its *target*. All operations with a symbolic link are channeled to the symbolic link's target. When you read data from or write data to a symbolic link, you write to its underlying target file. But if you delete a symbolic link, the target file isn't deleted. Apart from working with paths that refer to files and directories, NIO can work with symbolic links also. It includes multiple methods where you can specify whether you want the symbolic links to be followed or not. NIO can also determine circular references in symbolic links where a target refers back to the symbolic link.

A `Path` object is system-dependent. A `Path` can never be equal to a `Path` associated with another file system, even if they include exactly the same values. For example, a `Path` object to file 'Hello.txt' on Windows isn't equal to a `Path` object to file 'Hello.txt' on Solaris. Let's get started with creating some `Path` objects.

8.1.1 Multiple ways to create Path objects

You can create `Path` objects by using methods from multiple classes: `java.nio.file.Paths`, `java.nio.file.FileSystem`, and `java.io.File`. Table 8.1 shows a list of these classes, together with the relevant methods.

Table 8.1 List of classes and their respective methods that can be used to create objects of `Path`

Class name	Method declaration
`java.nio.file.Paths`	`public static Path get(String first, String... more)`
`java.nio.file.FileSystem`	`public abstract Path getPath(String first, String... more)`
`java.io.File`	`public Path toPath()`

`Paths` (notice an extra "s" at the end) is a *factory* class that defines static methods to create `Path` objects. Method `get(String first, String... more)` converts a string representation of a file or directory to a `Path` object. You can use a single string or sequence of string objects to create a `Path` object. You can use a forward slash (`/`) as a file and directory separator. Here are multiple ways of specifying a path to file 8-1.txt, using `Paths.get()`:

```
Paths.get("C:/OCPJava7/8/8-1.txt");
Paths.get("C:", "OCPJava7", "8", "8-1.txt");
Paths.get("C:\\OCPJava7\\8\\8-1.txt");
Paths.get("8-1.txt");
```

❶ Absolute path to file 8-1.txt

❷ Absolute path to file 8-1.txt, with directories and files as separate arguments

❸ Absolute path to file 8-1.txt, with backslashes escaped

❹ Relative path to file 8-1.txt

The code at ❶, ❷, and ❸ defines an absolute path to file 8-1.txt. The code at ❹ defines a relative path. For a relative path, the target file or directory is assumed to exist in your current directory. Note how the code at ❸ uses a backward slash (`\`) as a file separator. You *must* escape a backslash in a Java string value:

```
Path path = Paths.get("c:\users\harry\Hello.txt");
Path path2 = Paths.get("c:\\users\\harry\\Hello.txt");
```

Won't compile; need to escape \ used as a separator, by using another \.

Okay

When the path is provided as one string, where individual subpaths are separated by a path separator, only the last method argument is considered as a filename. All the others are assumed to be directory names.

Creating Path objects by using FileSystem class

You can also create `Path` objects by using method `getPath()` in class `FileSystem`. Because class `FileSystem` is an abstract class, you can get a reference to the current class `FileSystem` object by calling `getDefault()` on class `FileSystems`. Assuming that this application executes on a Unix or Linux machine, here's how you can create objects of `Path` to refer to file 8-1`.txt`:

```
FileSystems.getDefault().getPath("/home/OCPJava7/8/8-1.txt");
FileSystems.getDefault().getPath("/home", "OCPJava7", "8", "8-1.txt");
FileSystems.getDefault().getPath("\\home\\OCPJava7\\8\\8-1.txt");
FileSystems.getDefault().getPath("8-1.txt");
```

Absolute path to file 8-1.txt

Relative path to file 8-1.txt

A `Path` object can refer to a nonexistent file or directory. Watch out for this point on the exam. Even though you didn't create the file 8-1.txt until this point, a `Path` object that refers to it is valid.

EXAM TIP A `Path` object can refer to a nonexistent file or directory.

Creating Path objects by using File class

As discussed in chapter 7, prior to Java 7, objects of class `java.io.File` were used to represent the file and directory paths. Starting with Java 7, a new method, `toPath()`, was added to class `File` to bridge the gap between the existing I/O classes and NIO classes. You can create a `Path` object by using a `File` instance:

```
File file = new File("Hello.txt");
Path path = file.toPath();
```

Object of java.io.File

What happens if you create a `Path` object as follows?

```
Path path = Paths.get("");
```

Path created using one element that's empty.

Created using a zero-length string value, the preceding `path` variable refers to the current directory. Though `path.toString()` returns a zero-length string value, `path.getAbsolutePath()` would return its absolute path.

NOTE Behind the scenes, both `Paths.get()` and `File.toPath()` call `FileSystems.getDefault().getPath()`.

Let's refer back to the sample application Exam FlashCards and add functionality to it—that is, code that executes when a user activates the Save button. Let's create `Path` objects by adding code to method `actionPerformed()` as shown in the following listing.

Listing 8.2 Adding functionality to the sample application

```
public void actionPerformed(ActionEvent e) {
    String baseDir = "E:\\OCPJava7\\";
    String subDir = tfMainObj.getText();
    String fileName = tfSubObj.getText() + ".txt";
    Path path = Paths.get(baseDir, subDir, fileName);

    JOptionPane.showMessageDialog(f, path.toString());
}
```

Base dir

Directory for objectives

File name—one file per subobjective

Create Path object

Show Path in a message box

Until this point, you're only creating Path objects. They aren't tied to a real file or directory. Before you can move on with creating real files and directories that can be tied to this Path interface, you need to familiarize yourself with the Path objects and how they can be manipulated.

INTERFACE JAVA.NIO.FILE.PATH VERSUS CLASS JAVA.NIO.FILE.PATHS

Due to the similarity in their names, it's easy to confuse Paths with Path. But they are different—Path is an *interface* and Paths is a *class*. The class Paths is a utility class, with static methods to create objects that can be referred to by variables of the Path interface. The Path interface is used to represent the path of files or directories on a system. Figure 8.4 shows a UML representation of the Path interface and class Paths.

The Path interface extends interfaces Comparable, Iterable, and Watchable. This essentially means that the Path objects can be compared (with each other), iterated

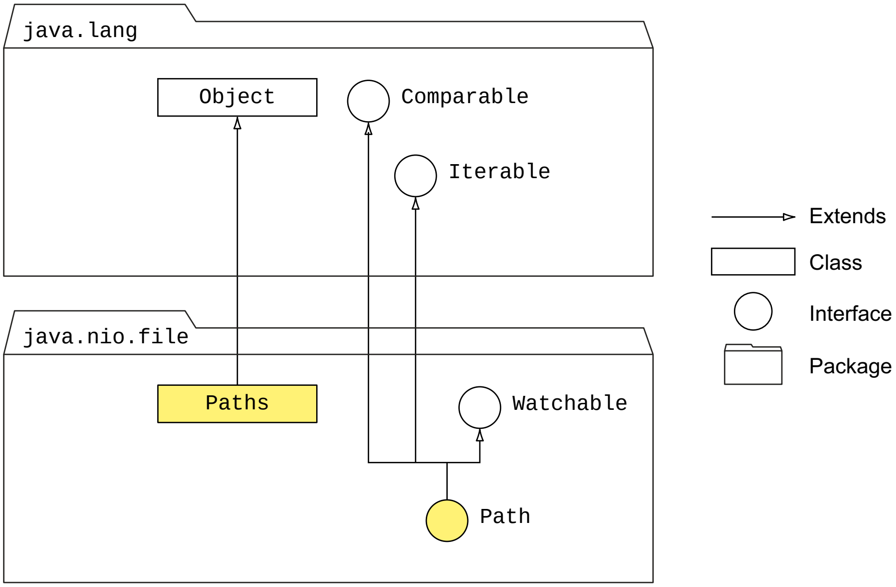


Figure 8.4 Comparing class Paths and interface Path

(over their directories and file components), and watched for changes. We'll cover these in the later sections of this chapter.

EXAM TIP Most of the `Path` methods perform syntactic operations. They manipulate the paths to a file or directory without accessing the file systems. They're logical operations on paths in memory.

Let's get started with working with `Path` methods that access its components.

8.1.2 Methods to access Path components

`Path` objects are used to work with the files and directories in a file system, and so the paths are system-dependent. On the exam, you'll be asked to use `Path` objects to retrieve information about a path like its subpath and root or parent directories. Let's work with a quick example:

```
class ManipulatePaths {
    public static void main(String args[]) {
        Path path = FileSystems.getDefault().getPath
                                 ("c:\\users\\obj8\\8-1.txt");
        System.out.println("toString()-> " + path.toString());
        System.out.println("getRoot()-> " + path.getRoot());
        System.out.println("getName(0)-> " + path.getName(0));
        System.out.println("getName(1)-> " + path.getName(1));
        System.out.println("getFileName()-> " + path.getFileName());
        System.out.println("getNameCount()-> " + path.getNameCount());
        System.out.println("getParent()-> " + path.getParent());
        System.out.println("subpath(0,2)-> " + path.subpath(0,2));

    }
}
```

Here's the output of the preceding code:

```
toString()-> c:\users\obj8\8-1.txt
getRoot()-> c:\
getName(0)-> users
getName(1)-> obj8
getFileName()-> 8-1.txt
getNameCount()-> 3
getParent()-> c:\users\obj8
subpath(0,2)-> users\obj8
```

Figure 8.5 shows a pictorial representation of the preceding code that makes it easy to retain this information. Note how the root of a path is not used in all the `Path` methods. Though it's used by a method like `getRoot()`, it's ignored by other methods like `subpath()` and `getName()`.

EXAM TIP Methods `getName()`, `getNameCount()`, and `subpath()` don't use the root directory of a path. Method `getRoot()` returns the root of an absolute path and `null` for relative paths. Play around with these methods—you might see them on the exam.

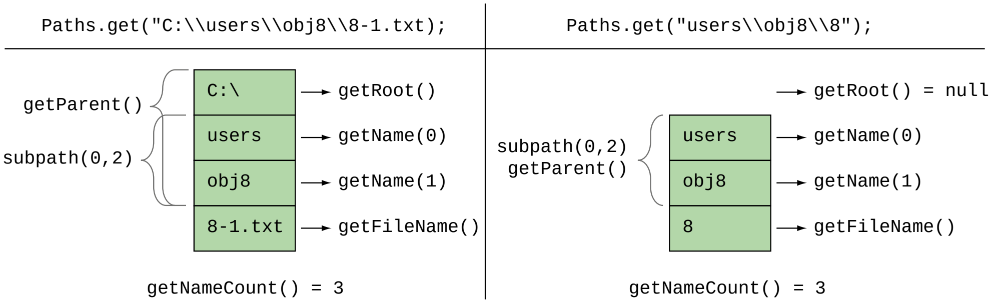


Figure 8.5 Representation of calling various `Path` methods

The `Path` methods that accept positions throw an `IllegalArgumentException` at runtime for invalid positions. For example, `getName()` and `subpath()` throw an `IllegalArgumentException` if you pass invalid path positions to them:

```
Path path = FileSystems.getDefault().getPath("c:\\users\\obj8\\8-1.txt");
System.out.println("subpath(0,4)-> " + path.subpath(0,4));
```

Throws IllegalArgumentException

The code `path.subpath(0,4)` will throw an `IllegalArgumentException` because it refers to an invalid position in the value referred by `path`.

8.1.3 Comparing paths

You can compare paths lexicographically using the method `compareTo(Path)`. To check whether a path starts or ends with another path, you can use `startsWith(String)`, `startsWith(Path)`, `endsWith(String)`, and `endsWith(Path)`:

```
class ComparePaths {
    public static void main(String args[]) {
        Path path1 = FileSystems.getDefault().
                     getPath("c:\\users\\obj8\\8-1.txt");
        Path path2 = Paths.get("d:\\users\\obj8\\8-1.txt");

        System.out.println(path1.compareTo(path2));
        System.out.println(path2.startsWith("\\"));

        System.out.println(path1.endsWith("-1.txt"));
        System.out.println(path1.endsWith(Paths.get("8-1.txt\\")));
    }
}
```

Returns a negative number—path1 is lexicographically smaller than path2.

Returns false

Returns false

Returns true—trailing separators aren't taken into account.

Methods `startsWith(String)` and `endsWith(String)` convert the method argument to a `Path` object before comparing it with the first or last element of the `Path` object. Although you're using `String`, a string comparison isn't executed. It's comparing paths, and path "1.txt" isn't the same as "8-1.txt" (two different files or directories). This explains why path1.endsWith("-1.txt") returns false.

EXAM TIP Methods startsWith() and endsWith() are overloaded: startsWith(String), startsWith(Path), endsWith(String), and endsWith(Path). So if you pass null to these methods, you'll get a compiler error.

8.1.4 Converting relative paths to absolute paths

The code in listing 8.2 uses a hard-coded value for the base directory. Let's see how you use your current directory and parent directories to create Path objects. Let's assume that your current working directory is E:/OCPJavaSE7/FileNIO. Now assume you enter the value of the subobjective as 8-1. The following will create an absolute path to 8-1.txt in your current working directory, that is, E:\OCPJavaSE7\FileNIO\8-1.txt:

```
Path file = Paths.get("8-1.txt");
Path path = file.toAbsolutePath();
```

Imagine that you want to create a text file 8-1.txt in the parent directory of your current working directory. Knowing that you can use .. to denote your parent directory, do you think the following will help?

```
Path file = Paths.get("..\\8-1.txt");
Path path = file.toAbsolutePath();
```

Yes, it will. Though the preceding code will insert .. in the Path object, if you use it to create the corresponding file, it will be created in the parent directory of the current working directory.

EXAM TIP Note that the method name to retrieve the absolute path from a Path object is toAbsolutePath() and not getAbsolutePath(). These method names are similar and might be used on the exam.

In the preceding code, path will refer to E:\OCPJavaSE7\FileNIO\..\8-1.txt. As you can see, inclusion of directories FileNIO and .. is redundant in the preceding path. You can remove these redundant values by calling method normalize() on Path:

```
Path file = Paths.get("..\\8-1.txt");
Path path = file.toAbsolutePath();
path = path.normalize();
System.out.println(path);
```

Prints E:\OCPJavaSE7\8-1.txt

EXAM TIP Path is immutable and calling normalize() on a Path object doesn't change its value.

Though implicit, it's common to use a period (.) to denote the current directory. For example, if you refer to file 8-1.txt, you refer to this path in the current directory. But it's common for programmers to refer to this path as ./8-1.txt. Again, when you include a period in a `Path` object, you're including redundant information, which can be removed too using method `normalize()`. The following code assumes your current working directory is E:/OCPJavaSE7/FileNIO:

```
Path file = Paths.get(".\\8-1.txt");
Path path = file.toAbsolutePath();
path = path.normalize();
System.out.println(path);
```

Prints E:\OCPJavaSE7\FileNIO\8-1.txt

EXAM TIP Method `normalize()` doesn't check the actual file system to verify if the file (or directory) the resulting path is referring to actually exists.

Do you think, when a `Path` object includes redundancies like `.` or `..`, that calling information retrieval methods like `subpath()` or `getName()` will also include these redundancies in the returned values? Let's see whether you can answer this question in the first "Twist in the Tale" exercise for this chapter.

Twist in the Tale 8.1

Examine the following code and select the correct answers.

```
import java.nio.file.*;
class Twist8_1{
    public static void main(String args[]) {
        Path path = Paths.get("c:\\OCPJavaSE7\\..\\obj8\\.\\8-1.txt");

        System.out.println(path.toString());       //line1
        System.out.println(path.getName(1));       //line2
        System.out.println(path.getParent());      //line3
        System.out.println(path.subpath(2,4));     //line4
    }
}
```

- **a** Code on line 1 outputs `c:\OCPJavaSE7\..\obj8\.\8-1.txt`
- **b** Code on line 1 outputs `c:\obj8\8-1.txt`
- **c** Code on line 2 outputs `OCPJavaSE7`
- **d** Code on line 2 outputs `c:\`
- **e** Code on line 2 outputs `..`
- **f** Code on line 3 outputs `c:\OCPJavaSE7\..\obj8\`
- **g** Code on line 3 outputs `c:\obj8\`
- **h** Code on line 4 throws an `IllegalArgumentException`

8.1.5 Resolving paths using methods resolve and resolveSibling

The overloaded methods `resolve(String)` and `resolve(Path)` are used to join a relative path to another path. If you pass an absolute path as a parameter, this method returns the absolute path:

```
Path path = Paths.get("/mydir/code");
System.out.println(path.resolve(Paths.get("world/Hello.java")));
System.out.println(path.resolve ("/world/Hello.java"));

Path absolutePath = Paths.get("E:/OCPJavaSE7/");
System.out.println(absolutePath.resolve(path));
System.out.println(path.resolve(absolutePath));
```

The output of the preceding code is:

```
\mydir\code\world\Hello.java
\world\Hello.java
E:\mydir\code
E:\OCPJavaSE7
```

Imagine you need to retrieve the path to a file in the same directory, say, to create its copy or to rename it. To do so, you can use the overloaded methods `resolveSibling(String)` and `resolveSibling(Path)`. These resolve a given path against a path's parent. If the given path is an absolute path, this method returns the absolute path. If you pass it an empty path, it returns the parent of the path:

```
Path path = Paths.get("/mydir/eWorld.java");
Path renamePath = path.resolveSibling(Paths.get("newWorld.java"));
Path copyPath = path.resolveSibling("backup/eWorld.java");
Path absolutePath = Paths.get("E:/OCPJavaSE7/");

System.out.println(renamePath);
System.out.println(copyPath);
System.out.println(path.resolveSibling(""));

System.out.println(absolutePath.resolveSibling(path));
System.out.println(path.resolveSibling(absolutePath));
```

The output of the preceding code is as follows:

```
\mydir\newWorld.java
\mydir\backup\eWorld.java
\mydir

E:\mydir\eWorld.java
E:\OCPJavaSE7
```

EXAM TIP Methods `resolve()` and `resolveSibling()` don't check the actual file system to verify if the file (or directory) the resulting path is referring to actually exists.

8.1.6 Method relativize()

Imagine you need a path to construct a path between two `Path` objects. To do so, you can use method `relativize()`. It can be used to construct a path between two relative or absolute `Path` objects:

```
Path dir = Paths.get("code");
Path file = Paths.get("code/java/IO.java");
System.out.println(file.relativize(dir));
System.out.println(dir.relativize(file));
```

Relative paths

```
dir = Paths.get("/code");
file = Paths.get("/java/IO.java");
System.out.println(file.relativize(dir));
System.out.println(dir.relativize(file));
```

Absolute paths

The output of the preceding code is:

```
..\..
java\IO.java
..\..\code
..\java\IO.java
```

What happens when you try to use `relativize()` to construct a path between a relative path and an absolute path or two absolute paths having different roots (say, C:\ and D:\)? Invocation of method `relativize()` in the following code would throw a runtime exception:

```
Path dir = Paths.get("/code");
Path dirC = Paths.get("C:/code/MyClass.java");
Path dirD = Paths.get("D:/notes/summary.txt");

System.out.println(dir.relativize(dirD));
System.out.println(dirC.relativize(dirD));
```

Would throw runtime exception

EXAM TIP You can't create a path from a relative path to an absolute path and vice versa using method `relativize()`. If you do so, you'll get a runtime exception (`IllegalArgumentException`). Also, method `relativize()` doesn't check the actual file system to verify if the file (or directory) the resulting path is referring to actually exists.

Unlike method `toRealPath()`, most of the `Path` methods perform syntactic operations—that is, logical operations on paths in memory. Let's see whether you remember this important point while determining the output of the `Path` methods in the next "Twist in the Tale" exercise.

Twist in the Tale 8.2

Assuming that the byte code for class `Twist8_2` (`Twist8_2.class`) is located in directory /home, what is the output of the following code?

```
class Twist8_2{
    public static void main(String args[]) {
        Path dir = Paths.get("code");
        Path file = Paths.get("code/java/IO.java");
        Path relative = file.resolve(file.relativize(dir));
        Path absolute = relative.toAbsolutePath();
        System.out.println(absolute);
    }
}
```

a `/home/code/`

b `/home/code/../..`

c `/home/java/IO.java/../..`

d `/home/code/java/IO.java/../..`

In this section you learned how to create `Path` objects and manipulate them in memory without connecting them to the real files or directories on your file system. In the next section, you'll see how class `Files` uses these `Path` objects to create, move, copy, delete, walk, and query files and directories on a system.

8.2 Class Files

[8.2] Check, delete, copy, or move a file or directory with the Files class

Class `java.nio.file.Files` consists entirely of static methods for manipulating files and directories. Because the creation and deletion of files and directories are dependent on the underlying platform, most of the methods of class `Files` delegate these tasks to the associated system provider.

Let's get started with the creation of files and directories.

8.2.1 Create files and directories

Class `Files` defines multiple methods to create files and directories:

```
public static Path createFile(Path path, FileAttribute<?>... attrs)
                                                      throws IOException
public static Path createDirectory(Path dir, FileAttribute<?>... attrs)
                                                      throws IOException
public static Path createDirectories(Path dir, FileAttribute<?>... attrs)
                                                      throws IOException
```

Method `createFile()` *atomically* checks for the existence of the file specified by the method parameter `path` and creates it if it doesn't exist. The (checking and creation) operation is atomic with respect to all the file system operations that might affect the directory. Method `createFile()` fails and throws an exception if the file already exists, if a directory with the same name exists, if its parent directory doesn't exist due to an I/O error, or if the specified file attributes can't be set.

Method `createDirectory()` creates the specified directory (not the parent directories) on the file system. This method also atomically checks for the existence of the specified directory and creates it if it doesn't exist. Method `createDirectory()` throws an exception if a file or directory exists with the same name, if its parent directory doesn't exist, if an I/O error occurs, or if the specified directory attributes can't be set.

Method `createDirectories()` creates a directory, creating all nonexistent parent directories. If the target directory already exists, `createDirectories()` doesn't throw any runtime exceptions. It throws an exception if the specified `dir` exists but isn't a directory, if an I/O occurs, or if the specified directory attributes can't be set.

EXAM TIP Specifying file or directory attributes is optional with methods `createFile()`, `createDirectory()`, and `createDirectories()`. All these methods declare to throw an `IOException`, which is a checked exception.

Let's use these methods and add some action to the sample application Exam FlashCards. Let's create directories corresponding to the objective number entered by a user and text file corresponding to the subobjective number. Directory and file creation will happen when a user activates the Save button. When a user activates the Save button, code defined in method `actionPerformed` will execute (the remaining code listing remains the same as shown in listing 8.1). Here's the code for method `actionPerformed()`:

```
public void actionPerformed(ActionEvent e) {
    String baseDir = "E:\\OCPJava7\\";
    String subDir = tfMainObj.getText();
    String fileName = tfSubObj.getText() + ".txt";

    Path filePath = Paths.get(baseDir, subDir, fileName);

    try {
        Files.createDirectories(filePath.getParent());
        Files.createFile(filePath);

        PrintWriter pw = new PrintWriter(
                    new FileWriter(filePath.toFile(), true));
        pw.println(tfNote.getText());
        pw.flush();
    }
    catch (IOException ioe) {
        JOptionPane.showMessageDialog(f, ioe.toString());
    }
}
```

❶ **Base directory to store data**

❷ **Retrieve main objective value entered by user**

❸ **Retrieve subobjective value entered by user**

❹ **Create Path object by converting base directory to absolute path**

❺ **Create complete directory tree on file system**

❻ **Create file on file system**

❼ **Open a PrintWriter to file and write text from text-field note to it**

❽ **Handle IOException from creation of directories, file, and writing to file**

The code at ❶ defines a base directory to store your flash card data: E:\OCPJava7\. The code at ❷ and ❸ retrieves the value of the main objective and subobjective entered by a user. Because you're writing the flash cards text to a text file, the extension .txt is appended to it. The code at ❹ creates an absolute path corresponding to the .txt file. The code at ❺ executes `createDirectories()` from class `Files`, creating all nonexistent parent directories on your system. The code at ❻ calls `createFile()` from class `Files` to create the target file on your system. If it can't create a file because the file already exists, it throws a `FileAlreadyExistsException`. The code at ❼ creates a `PrintWriter` and writes the flash card tip to the text file. Because it's a Swing application, I've defined a message box to show you the name of the thrown exception ❽.

EXAM TIP In class `Files`, method `createDirectories()` can create both the target directory and multiple nonexistent parent directories. If the directory already exists and it can't create a directory, no exceptions are thrown. Methods `createDirectory()` and `createFile()` create a single directory and file respectively. They throw a `FileAlreadyExistsException` if a directory or file with the same name already exists.

In the sample application, click the Save button twice; the code that creates the text file will throw a `FileAlreadyExistsException`. In the next section, let's see how you can check for the existence of files or directories before you issue a command to create them.

8.2.2 Check for the existence of files and directories

You can check for the existence of a file or directory referred by a `Path` object using methods `exists()` and `notExists()` in class `Files`:

```
public static boolean exists(Path path, LinkOption... options)
public static boolean notExists(Path path, LinkOption... options)
```

Method `exists()` checks whether a file or directory referred by a `Path` exists or not; it returns `true` if the file or directory exists and `false` if the target doesn't exist or its existence can't be determined.

Method `notExists()` is *not* a complement of method `exists()`. It returns `true` if a target doesn't exist. If these methods can't determine the existence of a file, both of them will return `false`. By default, these methods follow a symbolic link. To override this behavior, you can pass `LinkOption.NOFOLLOW_LINKS` to these methods (`java.nio.file.LinkOption` is an enum). What happens if you check for the existence of a target path and before you can create a new file or directory another application creates it? In this case, you must handle the exception accordingly.

Let's add checks to verify the existence of your target directory and file in the sample application:

```
public void actionPerformed(ActionEvent e) {
    String baseDir = "E:\\OCPJava7\\";
    String subDir = tfMainObj.getText();
    String fileName = tfSubObj.getText() + ".txt";

    Path filePath = Paths.get(baseDir, subDir, fileName);

    try {
        if (Files.notExists(filePath.getParent()))
            Files.createDirectory(filePath.getParent());
        if (!Files.exists(filePath))
            Files.createFile(filePath);
        PrintWriter pw = new PrintWriter(
                   new FileWriter(filePath.toFile(), true));
        pw.println(tfNote.getText());
        pw.flush();
    }
    catch (IOException ioe) {
        JOptionPane.showMessageDialog(f, ioe.toString());
    }
}
```

EXAM TIP Watch out for questions that state that `exists()` and `notExists()` will never return the same `boolean` value for the same `Path` object. Both methods `exists()` and `notExists()` would return `false` if they can't determine the existence of the target file or directory.

8.2.3 Copy files

Imagine you create a study group to prepare for this exam. To get going, the members plan to share their individual flash card notes that they created using this application. To do so, they email the text files with flash card notes to everyone in the group. How would you use the notes from these files in your application? You might

- Copy the file to the directory that's used by your application
- Copy the contents of the received file to a file that's used by your application

Class `Files`'s overloaded `copy()` method enables you to read from `InputStream` and write to a `Path` object, read from a `Path` object and write to `OutputStream`, and read from and write to `Path` objects:

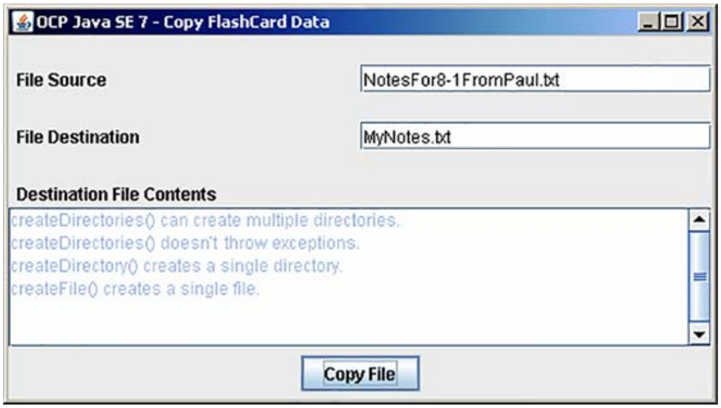
```
public static long copy(InputStream in, Path target, CopyOption... options)
public static long copy(Path source, OutputStream out)
public static Path copy(Path source, Path target, CopyOption... options)
```

EXAM TIP `Files.copy()` can copy only files, not directories. If the source is a directory, then in the target an empty directory is created (without copying the entries in the directory). This method returns a `long` or `Path` value, not a `boolean` value. Watch out for its invalid use in exam questions that use `copy()` to copy directories, use it in `try`-with-resources statements, or use its return value to test whether a file was copied or not.

Method `copy()` accepts objects of the `CopyOption` interface. You can use objects of enum `StandardCopyOption` (shown in table 8.2), which implements this interface.

Table 8.2 Enum constants of `StandardCopyOption` can be passed to methods that accept `CopyOption`. `StandardCopyOption` implements the `CopyOption` interface.

Enum constant	Description
`StandardCopyOption.ATOMIC_MOVE`	Hasn't been implemented yet
`StandardCopyOption.COPY_ATTRIBUTES`	Copy attributes
`StandardCopyOption.REPLACE_EXISTING`	Replace existing entity with the same name



OCP Java SE 7 - Copy FlashCard Data

File Source NotesFor8-1FromPaul.txt

File Destination MyNotes.txt

Destination File Contents

createDirectories() can create multiple directories.
createDirectories() doesn't throw exceptions.
createDirectory() creates a single directory.
createFile() creates a single file.

Figure 8.6 Swing application used to copy a source file to a destination

For the example, let's copy the flash card notes received in a .txt file to a folder that's used by your application. Figure 8.6 shows the screenshot of this application after the Copy File button is activated.

Here's the code of this Swing application, which accepts the path to the source and destination files from a user and copies the source file to the destination. It might seem unnecessary to you to define an elaborate example to copy files. Reading or browsing through the Java API, particularly the Java File I/O API, can be very boring. Try out the following code with different combinations of source and target files. It displays the error messages in a popup box and the contents on the copied file. Try it with different combinations of the source and target files, using relative or absolute paths.

```
import java.io.*;
import java.nio.file.*;
import java.nio.charset.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class CopyFlashCardData implements ActionListener {
    JFrame f = new JFrame("OCP Java SE 7 - Copy FlashCard Data");
    JTextField tfCopyFrom = null;
    JTextField tfCopyTo = null;
    JTextArea taFileContents = null;
    JButton btnCopyFile = null;

    private void buildUI() {
        tfCopyFrom = new JTextField();
        tfCopyTo = new JTextField();
        taFileContents = new JTextArea("Click 'Copy file' to " +
                                                  "view contents");
        taFileContents.disable();
        btnCopyFile = new JButton("Copy File");
        btnCopyFile.addActionListener(this);
```

Text field to accept file source

Text field to accept file destination

Text area to display copied file

Button to initiate file copy

Build application's UI

```
        JPanel topPanel = new JPanel();
        topPanel.setLayout(new GridLayout(6,2));
        topPanel.add(new JLabel(""));
        topPanel.add(new JLabel(""));

        topPanel.add(new JLabel("  File Source"));
        topPanel.add(tfCopyFrom);
        topPanel.add(new JLabel(""));
        topPanel.add(new JLabel(""));
        topPanel.add(new JLabel("  File Destination"));
        topPanel.add(tfCopyTo);

        topPanel.add(new JLabel(""));
        topPanel.add(new JLabel(""));
        topPanel.add(new JLabel("  Destination File Contents"));
        topPanel.add(new JLabel(""));

        JPanel middlePanel = new JPanel();
        middlePanel.setLayout(new BorderLayout());
        middlePanel.add(taFileContents);

        JPanel bottomPanel = new JPanel();
        bottomPanel.add(btnCopyFile);

        JPanel mainPanel = new JPanel();
        mainPanel.setLayout(new BorderLayout());
        mainPanel.add(BorderLayout.NORTH, topPanel);
        mainPanel.add(BorderLayout.CENTER, middlePanel);
        mainPanel.add(BorderLayout.SOUTH, bottomPanel);
        f.getContentPane().setLayout(new BorderLayout());
        f.setSize(500, 550);
        f.getContentPane().add(mainPanel);
        f.setVisible(true);
```

Build application's UI

```
        tfCopyFrom.setText("E:/OCPJava7/downloads/8-1.txt");
        tfCopyTo.setText("E:/OCPJava7/8/8-1.txt");
    }
```

Define default values for source and target fields.

```
    public void actionPerformed(ActionEvent e) {
        try {
            Path source = Paths.get(tfCopyFrom.getText());
            Path target = Paths.get(tfCopyTo.getText());

            Files.copy(source, target,
                               StandardCopyOption.REPLACE_EXISTING);
```

Create Path object from source value entered by user

Create Path object from target value entered by user

Call Files.copy to copy file to destination; StandardCopyOption.REPLACE_EXISTING replaces existing file

```
            byte[] bytes = Files.readAllBytes(target);
            taFileContents.setText(new String(
                                   bytes, Charset.defaultCharset()));
        }
        catch (IOException ioe) {
            JOptionPane.showMessageDialog(f, ioe.toString());
        }
    }
```

Reads target file and displays its contents in a text area.

Shows message for exception caught in a dialog box

```
    public static void main(String[] args) {
        CopyFlashCardData fc = new CopyFlashCardData();
        fc.buildUI();
    }
}
```

You can use absolute links and relative links to copy files. When you use a relative link to a target file, where do you think the target file is created? Is it created relative to your Java code (.class) and not relative to your source file?

EXAM TIP If you use a relative path to the target file, the file is created relative to your Java class (.class file) and not relative to the source file (passed as a parameter to method `Files.copy()`).

The preceding code shows you how to copy a file to another location. Method `copy()` in class `Files` doesn't allow you to append data to an existing file. If you want to add the notes sent to you by your study group to your own text files, you need to open the target file in append mode and use I/O streams or readers/writers to do so. If you're not sure how to do this, refer to chapter 7.

EXAM TIP Method `copy()` in class `Files` doesn't allow you to append data to an existing file; rather, it creates a new file or replaces an existing one.

The overloaded version of method `copy()` that reads from `InputStream` and writes to a `Path` object can also be used to read from your system's standard input stream using `System.in` and write it to a file as follows:

```
try (InputStream in = System.in){
    Path target = Paths.get("myNotesFromConsole.txt");
    Files.copy(in, target, StandardCopyOption.REPLACE_EXISTING);
}
catch (IOException ioe) {
    System.out.println(ioe);
}
```

You can also read from a `Path` object and write its contents to `OutputStream`:

```
import java.io.*;
import java.nio.file.*;
public class WriteDataFromPathToStream{
    public static void main(String[] args) {
        try (OutputStream out = new FileOutputStream("Copy.txt")){
            Path source = Paths.get("WriteDataFromPathToStream.java");
            Files.copy(source, out);
        }
        catch (IOException ioe) {
            System.out.println(ioe);
        }
    }
}
```

Copy Path object to OutputStream

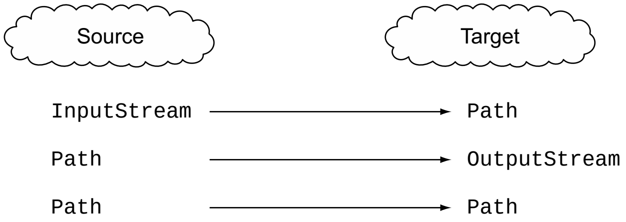


Figure 8.7 Types of sources and targets used by the overloaded method copy() to copy files

Figure 8.7 shows the types of sources and targets that can be used with the overloaded method `copy()` in class `Files`.

8.2.4 Move files and directories

Moving files and directories is a common requirement. To move files or directories programmatically, you can use `Files.move()`, which moves or renames a file to a target file:

```
public static Path move(Path source, Path target, CopyOption... options)
```

To rename a file notes.txt to copy-notes.txt, keeping the file in the same directory, you can use the following:

```
Path source = Paths.get("notes.txt");
Files.move(source, source.resolveSibling("copy-notes.txt"));
```

To move a file to a new directory, retaining the same filename and replacing any existing file of that name in the new directory, you can use the following:

```
Path source = Paths.get("notes.txt");
Path target = Paths.get("/home/myNotes/");
Files.move(source, target.resolve(source.getFileName()),
                                   StandardCopyOption.REPLACE_EXISTING);
```

EXAM TIP You can only move empty directories using method `Files.move()`. You can rename a nonempty directory by using `Files.move()`. But you can't move a file or directory to a nonexisting directory.

8.2.5 Delete files and directories

To delete a directory or a file referred to by a `Path` object, you can use the following methods from class `Files`:

```
public static void delete(Path path)
public static boolean deleteIfExists(Path path)
```

Deletes a file

Deletes a file if it exists

Both the preceding methods can delete a file or directory (if it's empty). If you try to delete a directory that isn't empty, these methods will throw a `DirectoryNotEmptyException`. If you try to delete a nonexistent file or directory using method `delete()`, it will throw a `NoSuchFileException`. But method `deleteIfExists()` won't throw an exception if the file or directory at the specified path doesn't exist—it will return `false`. The deletion operation might also fail if the target file is in use, because some operating systems don't allow deletions of files or directories if they're in use by an active program.

EXAM TIP Methods `delete()` and `deleteIfExists()` can be used to delete files and (nonempty) directories.

8.2.6 Commonly thrown exceptions

File creation, copying, renaming, moving, and deletion throw a couple of common checked and unchecked exceptions:

- `IOException` (checked exception)—The I/O devices are beyond the immediate control of JVM, so operations involving the I/O devices can fail at any time, throwing an `IOException`.
- `NoSuchFileException` (runtime exception)—This exception is thrown when you try to delete, move, or copy a nonexistent file or directory.

When you operate on files and directories, you might want to query their existing attributes. Let's see how you can do that in the next section.

8.3 Files and directory attributes

[8.3] Read and change file and directory attributes, focusing on the BasicFileAttributes, DosFileAttributes, and PosixFileAttributes interfaces

Imagine you want to set the last modification time stamp for a file or directory or add a new user-defined attribute to it. You can do so by adding, accessing, or modifying the attributes of a file or directory. The file or directory attributes refer to their metadata, or data about data. You use files or directories to store your data. The OS stores additional data about your data that helps it to determine when it was created, modified, accessed, and so on. You can access individual attributes or a group of attributes for files and directories.

Prior to version 7, Java supported limited access to the attributes of files and directories. To access non-supported attributes, developers often worked with native code. With its `java.nio.file.attribute` package, NIO.2 supports access to and (if allowed) modification of a larger set of file and directory attributes.

8.3.1 Individual attributes

Class `Files` defines static methods to access individual attributes of a file or directory referred by a `Path`, such as its size, when was it last modified, whether it's readable or writable, and whether it's a directory or a file. Following is an example to access some of the attributes of a java source file:

```
import java.nio.file.*;
public class AccessAttributes{
    public static void main(String[] args) throws Exception {
        Path path = Paths.get("MyAttributes.java");

        System.out.println("size:" + Files.size(path));
        System.out.println("isDirectory:" + Files.isDirectory(path));
        System.out.println("isExecutable:" + Files.isExecutable(path));
        System.out.println("isHidden:" + Files.isHidden(path));
        System.out.println("isReadable:" + Files.isReadable(path));
        System.out.println("isSameFile:" + Files.isSameFile(path, path));
        System.out.println("isDirectory:" + Files.isDirectory(path));
        System.out.println("isSymbolicLink:" + Files.isSymbolicLink(path));
        System.out.println("isWritable:" + Files.isWritable(path));
        System.out.println("getLastModifiedTime:" +
                                          Files.getLastModifiedTime(path));
        System.out.println("getOwner:" + Files.getOwner(path));
    }
}
```

Here's the output of the preceding code (the value of some of these attributes might vary depending on your system):

```
size:958
isDirectory:false
isExecutable:true
isHidden:false
isReadable:true
isSameFile:true
isDirectory:false
isSymbolicLink:false
isWritable:true
getLastModifiedTime:2014-12-09T08:03:54.609375Z
getOwner:AD-3B5C889B134A\Administrator (User)
```

You can also access the individual attributes of a file or directory by using method `Files.getAttribute()`, passing to it the name of the attribute as a string value. To modify the attributes of an existing file or directory, you can use `Files.setAttribute()`. Table 8.3 shows the relevant methods of class `Files` that can be used to update existing attributes.

Table 8.3 Methods to modify individual attributes of files or directories

Method	Description
`public static Path setAttribute(Path path, String attribute, Object value, LinkOption... options)`	Sets the value of a file attribute
`public static Path setLastModifiedTime(Path path, FileTime time)`	Updates a file's last modified time attribute
`public static Path setOwner(Path path, UserPrincipal owner)`	Updates the file owner

The following code displays the existing creation time of a Java source file using method `Files.getAttribute()` and then modifies it using method `Files.setAttribute()`:

```
import java.nio.file.*;
import java.nio.file.attribute.*;
public class ModifyAttributes{
    public static void main(String[] args) {

        Path path = Paths.get("ModifyAttributes.java");
        System.out.println("creationTime:" +
        Files.getAttribute(path, "creationTime"));

        FileTime newTime = FileTime.fromMillis(System.currentTimeMillis());
        Files.setAttribute(path, "creationTime", newTime);

        System.out.println("creationTime:" +
                                Files.getAttribute(path, "creationTime"));
    }
}
```

Retrieves value of attribute creationTime.

Updates value of attribute creationTime.

EXAM TIP Methods `Files.setAttribute()` and `Files.getAttribute()` can be used to access a file or directory attribute and modify it (if allowed). The attribute name is passed to these methods as a string value.

Rather than accessing individual attributes, you can also access a group of attributes, as discussed in the next section.

8.3.2 Group of attributes

Querying the file system multiple times to access all file or directory attributes can affect your application's performance. To get around this, you can access a group of file attributes by calling `Files.getFileAttributeView()` or `Files.readAttributes()`.

Interfaces to read and modify attribute sets

Different file systems might support different attribute sets. Java groups related attributes that correspond to a specific file system implementation like DOS or POSIX, or to a common functionality like file owner attributes. You can use multiple interfaces to access file and directory attributes and modify them. These groups are defined as interfaces, and the ones on the exam are as follows:

- `BasicFileAttributes` and `BasicFileAttributeView`—The `BasicFileAttributes` interface defines methods to access the basic attributes that should be supported by all the file systems. The `BasicFileAttributeView` interface can be used to modify the basic attributes.
- `DosFileAttributes` and `DosFileAttributeView`—The `DosFileAttributes` interface extends `BasicFileAttributes` and defines methods to access attributes specific to Windows files and directories. The `DosFileAttributeView` interface defines methods to modify the DOS file attributes.
- `PosixFileAttributes` and `PosixFileAttributeView`—The `PosixFileAttributes` interface also extends `BasicFileAttributes` and defines methods

to access attributes related to the POSIX family of standards, like Linux or UNIX. The `PosixFileAttributeView` interface defines methods to modify attributes related to the POSIX family.

- `AclFileAttributeView`—Available only for Windows OS, this interface supports access and updates of a file's access control list (ACL).
- `FileOwnerAttributeView`—This interface supports access and updates to the owner of a file or directory. It's supported by all systems that support the concept of file owners.
- `UserDefinedFileAttributeView`—This interface supports the addition, modification, and deletion of user-defined metadata.

NOTE For the preceding list of interfaces, the ones that end with the term "View" are collectively referred to as the *view interfaces*, and they're used to update file and directory attributes.

Figure 8.8 shows `BasicFileAttributes`, `DosFileAttributes`, and `PosixFileAttributes` interfaces with their methods to help you get a better understanding of the methods that they define.

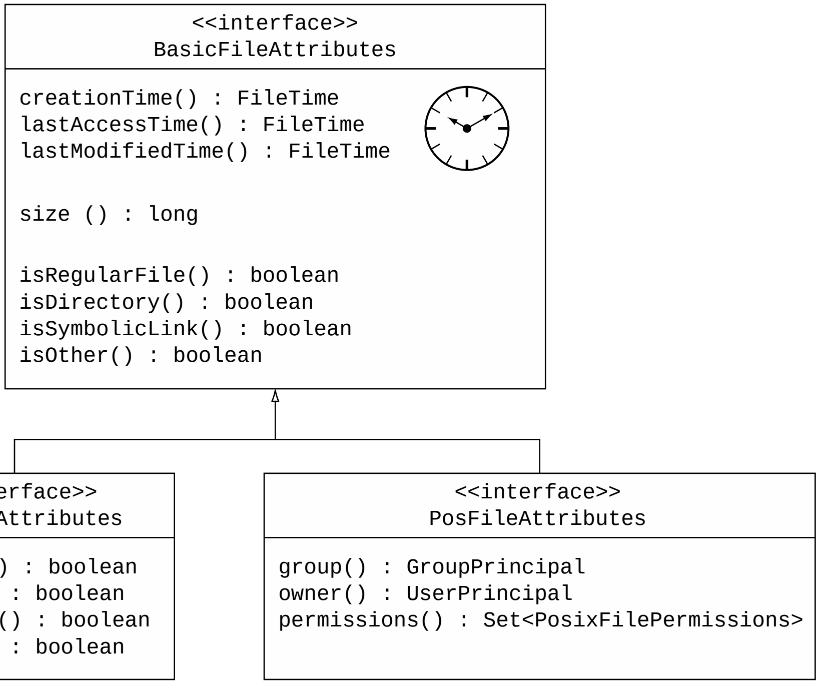


Figure 8.8 Interface `BasicFileAttributes` is extended by interfaces `DosFileAttributes` and `PosixFileAttributes`

EXAM TIP The `BasicFileAttributes`, `DosFileAttributes`, and `PosixFileAttributes` interfaces define methods to access attributes. They don't define methods to modify (or set) the attributes. Use class `Files` or view interfaces (covered in this section) to modify the attributes.

Figure 8.9 shows the view interfaces, which can be used to modify file or directory attributes.

To access and update a group of attributes for a directory or file, you can access an attribute view by using method `File.getFileAttributeView()`. To only access (not update) an attribute group, you can use method `File.readAttributes()`. You can also call method `readAttributes()` on an attribute view to get its corresponding attribute set. Following is an example that uses these methods:

```
Path path = Paths.get("pathToaFile");
PosixFileAttributeView view = Files.getFileAttributeView(path,
                                   PosixFileAttributeView.class);
PosixFileAttributes attr = view.readAttributes();
PosixFileAttributes attr2 = Files.readAttributes(path,
                                   PosixFileAttributes.class);
```

EXAM TIP If a file system doesn't support an attribute view, `Files.getFileAttributeView()` returns `null`. If a file system doesn't support an attribute set, `File.readAttributes()` will throw a runtime exception.

Let's get started with accessing basic attributes and modifying them.

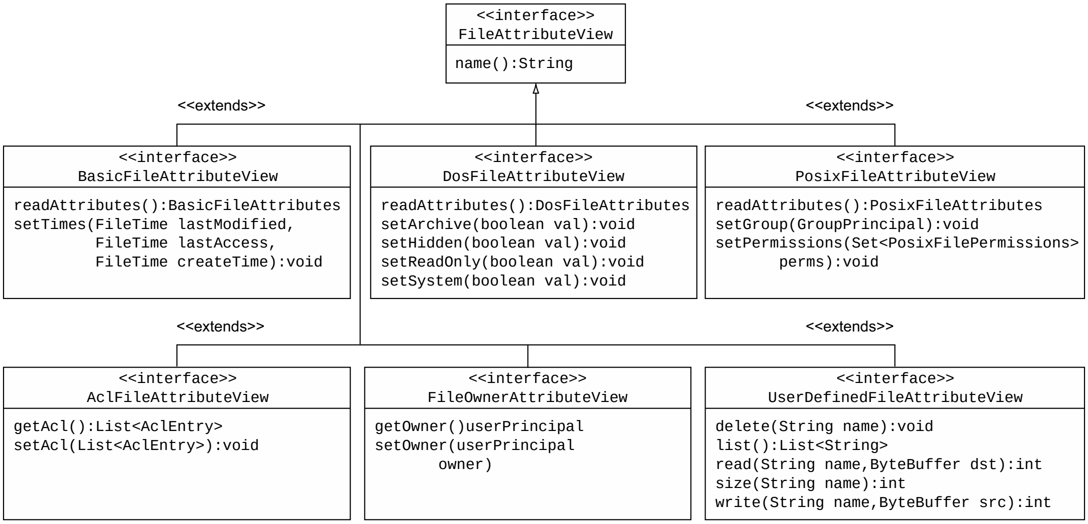

<<interface>>
FileAttributeView
name():String
<<extends>>
<<extends>>
<<interface>>
BasicFileAttributeView
readAttributes():BasicFileAttributes
setTimes(FileTime lastModified,
FileTime lastAccess,
FileTime createTime):void
<<interface>>
DosFileAttributeView
readAttributes():DosFileAttributes
setArchive(boolean val):void
setHidden(boolean val):void
setReadOnly(boolean val):void
setSystem(boolean val):void
<<interface>>
PosixFileAttributeView
readAttributes():PosixFileAttributes
setGroup(GroupPrincipal):void
setPermissions(Set<PosixFilePermissions>
perms):void
<<extends>>
<<extends>>
<<interface>>
AclFileAttributeView
getAcl():List<AclEntry>
setAcl(List<AclEntry>):void
<<interface>>
FileOwnerAttributeView
getOwner()userPrincipal
setOwner(userPrincipal
owner)
<<interface>>
UserDefinedFileAttributeView
delete(String name):void
list():List<String>
read(String name,ByteBuffer dst):int
size(String name):int
write(String name,ByteBuffer src):int

Figure 8.9 Interfaces `BasicFileAttributeView`, `DosFileAttributeView`, `PosixFileAttributeView`, `AclFileAttributeView`, `FileOwnerAttributeView`, and `UserDefinedFileAttributeView` can be used to update attribute values using Files.readAttributes() and Files.getFileAttributeView().

8.3.3 Basic attributes

Imagine you need to delete all files in a directory, whose creation time is older than one day, using your Java code. If you can access the creation time of a file, you can determine if the file needs to be deleted or not. Here's an example:

```
import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;
class DeleteOldFiles {
    public static void delDayOldFile(String fileName) throws IOException {
        Path file = Paths.get(fileName);

        BasicFileAttributes attr = Files.readAttributes(file,
                                          BasicFileAttributes.class);
        FileTime fileCreationTime = attr.creationTime();

        long currentTime = System.currentTimeMillis();
        FileTime dayOldFileTime = FileTime.fromMillis(
                                     currentTime - (24*60*60*1000));

        if (fileCreationTime.compareTo(dayOldFileTime) < 0)
            Files.delete(file);
    }
}
```

Create a path to filename

Get basic file attributes of file

Get file's creation timestamp

Get current time's milliseconds

Compute FileTime for a day old

Check if file creation-timestamp is older than one day

Delete file

You can access an object of a class that implements the BasicFileAttributes interface by calling Files.readAttributes() and passing it BasicFileAttributes.class. You don't need to be bothered about the exact type of the object that's returned to you.

EXAM TIP If an underlying system doesn't support all the basic timestamps—that is, creationTime, lastAccessTime, and lastModifiedTime—it might return system-specific information.

The next example uses BasicFileAttributeView to modify the file creation, modification, and access time of a file to 60, 50, and 30 seconds from the current time:

```
public class ModAttributes{
    public static void modifyDateAttr(String fileName) throws IOException {
        Path file = Paths.get(fileName);
        BasicFileAttributeView view = Files.getFileAttributeView(file,
                                          BasicFileAttributeView.class);

        long now = System.currentTimeMillis();
        FileTime creation = FileTime.fromMillis(now - 60000);
        FileTime lastModified = FileTime.fromMillis(now - 50000);
        FileTime lastAccess = FileTime.fromMillis(now - 30000);

        view.setTimes(lastModified, lastAccess, creation);
    }
}
```

Another method that you can use to read the file attributes is:

```
static Map<String,Object> readAttributes(Path path,
                                         String attributes,
                                         LinkOption... options)
                                      throws IOException
```

The parameter attributes take the form [view name:]attribute-list. The square brackets mean that the view name is optional—it can take values `basic`, `dos`, and `posix`. If this value is absent, it defaults to `basic`. The attributes are specified as a comma-separated list of the attributes to be read (without any spaces). You can specify * to read all the attributes for a group. Here's an example for accessing all the attributes of a file, printing them (attribute names and their values), and changing the `lastModifiedTime` of the file:

```
Path file = Paths.get(fileName);
Map<String,Object> values = Files.readAttributes(file, "*");

for (String attribute:values.keySet()) {
    System.out.println(attribute + " : " + values.get(attribute));
}

FileTime newTime = FileTime.fromMillis(System.currentTimeMillis());
Files.setAttribute(file, "lastModifiedTime", newTime);
```

You can also use a comma-delimited list of values:

```
Map<String,Object> values = Files.readAttributes(file,
                                   "lastModifiedTime,isDirectory");
```

To modify an attribute, you can call an `XxxFileAttributeView` interface, call `Files.setAttribute()`, or call a specific method from class `Files`, if it exists. For example, you can modify the last modified time of a `Path` object by calling `Files.setLastModifiedTime()`. The following line of code updates the last modified time for the file referred to by the variable `file`:

```
Files.setLastModifiedTime(file, dayOldFileTime);
```

EXAM TIP Methods `Files.setAttribute()` and `Files.getAttribute()` throw an `IllegalArgumentException` or `UnsupportedOperationException` if you pass them an invalid or unsupported attribute.

8.3.4 DOS attributes

As shown in figure 8.9, the `DosFileAttributes` interface makes the following attributes available:

- `archive`
- `hidden`
- `readonly`
- `system`

EXAM TIP The DOS attributes are available on a Windows system only. Trying to access them on other systems will throw a runtime exception.

The following example uses the `DosFileAttribute` and `DosFileAttributeView` interfaces to access DOS attributes of a file, updating them if a file is read only:

```
Path file = Paths.get("name-of-a-file");
DosFileAttributeView dosView = Files.getFileAttributeView(file,
                                          DosFileAttributeView.class);
DosFileAttributes dosAttrs = dosView.readAttributes();
if (dosAttrs.isReadOnly()) {
    dosView.setHidden(true);
    dosView.setArchive(false);
    dosView.setReadOnly(false);
    dosView.setSystem(true);
}
else
    System.out.println("Don't modify the attributes");
```

Use DosFileAttributes to access file attributes.

Use DosFileAttributeView to update file attributes.

You can also access file or directory attributes by using class `Files`. The following code reads DOS attributes:

```
Map<String,Object> values = Files.readAttributes(file,
                                        "dos:archive,hidden");
Map<String,Object> values2 = Files.readAttributes(file, "dos:*");
DosFileAttributes attr = Files.readAttributes(file,
                                         DosFileAttributes.class);
DosFileAttr
```

Read listed DOS attributes

Read all DOS attributes

EXAM TIP When you read *all* DOS attributes using method `Files.readAttributes()`, you also read the basic attributes.

To modify a DOS attribute, you must prefix the attribute name with `dos:` because an attribute is implicitly prefixed with `basic:` (which can result in an invalid attribute). Imagine that you wish to prevent listing a specific file in a files explorer on a Windows system. You can use `Files.setAttribute()` to modify its `hidden` property and set it to `true`:

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

EXAM TIP When you read or write an invalid value to a file attribute, the code throws the runtime exception `ClassCastException`.

8.3.5 POSIX attributes

The POSIX attributes are as follows:

- `group`
- `owner`
- `permissions`

EXAM TIP The POSIX attributes are available on the POSIX family of standards, like UNIX, LINUX, etc. Trying to access them on other systems will throw a runtime exception.

The following example updates the permissions of a file depending on whether it's owned by admin or not. It uses the `PosixFileAttributeView` and `PosixFileAttributes` interfaces:

```
PosixFileAttributeView posixView = Files.getFileAttributeView(file,
                                          PosixFileAttributeView.class);
PosixFileAttributes posixAttrs = posixView.readAttributes();
if (posixAttrs.owner().getName().equals("admin"))
    posixView.setPermissions(PosixFilePermissions.fromString("rwxrwxrwx"));
else
    posixView.setPermissions(PosixFilePermissions.fromString("rwxr-x---"));
```

You can also use class `Files` to read all POSIX file attributes:

```
Map<String,Object> values = Files.readAttributes(file, "posix:*");
PosixFileAttributes attr = Files.readAttributes(file,
                                         PosixFileAttributes.class);
```

8.3.6 *AclFileAttributeView interface*

The `AclFileAttributeView` interface supports reading and updating a file's ACL or file owner attributes. It defines methods `getAcl()` and `setAcl()`. This view is available only for Windows systems.

8.3.7 *FileOwnerAttributeView interface*

The `FileOwnerAttributeView` interface is supported by all file systems with a file owner concept, and this view includes methods to access and update the owner of a file or directory. If defines methods `getOwner()` and `setOwner(UserPrincipal)`.

EXAM TIP To read or update the owner of a file or directory you can use the `AclFileAttributeView`, `FileOwnerAttributeView`, and `PosixFileAttribute` interfaces.

8.3.8 *UserDefinedAttributeView interface*

The `UserDefinedAttributeView` interface can be used to add, delete, access, and modify additional user-defined attributes to a file or directory. It defines methods `delete(String)`, `list()`, `read(String, ByteBuffer)`, `size(String)`, and `write (String, ByteBuffer)` to, respectively, delete, list, read, get the attribute's size, and write attribute values.

Imagine you're developing a file management application that enables multiple users to delete files. But the files chosen for deletion are only marked for deletion and the actual deletion is executed separately. To accomplish this, you can add a user-defined attribute `delete` to a file with a value of, say, `true`. This attribute can be queried before actually deleting a file. The following sample code shows how you can do it. It defines code to write and read the user-defined attribute delete in method main(). But in a real application, this would usually be defined in separate classes or methods.

```
import java.io.*;
import java.nio.file.*;
import java.nio.*;
import java.nio.file.attribute.*;
import java.nio.charset.*;
public class UserAttrs{
    public static void main(String args[]) throws IOException {
        Path file = Paths.get("eJava.txt");

        UserDefinedFileAttributeView view= Files.getFileAttributeView(file,
                                       UserDefinedFileAttributeView.class);

        writeAttr(view,"delete",true);
        if (readAttr(view, "delete"))
            Files.delete(file);
    }
    static void writeAttr(  UserDefinedFileAttributeView view,
                            String attr,
                            boolean value) throws IOException {
        if (value)
            view.write(attr,Charset.defaultCharset().encode("true"));
        else
            view.write(attr,Charset.defaultCharset().encode("false"));
    }
    static boolean readAttr(  UserDefinedFileAttributeView view,
                              String attr) throws IOException {
        ByteBuffer buf = ByteBuffer.allocate(view.size(attr));
        view.read(attr, buf);
        buf.flip();
        String value = Charset.defaultCharset().decode(buf).toString();
        return (value.equalsIgnoreCase("true"));
    }
}
```

Use write(String, ByteBuffer) to write a user-defined attribute.

Use read(String, ByteBuffer) to read a user-defined attribute.

Use size() to get the size of attribute value delete.

The preceding code would delete the file referred by the variable file from your system. The possibilities are endless for how you can use a user-defined attribute that you add to a file or directory. In this section, you learned how to access and update the attributes of a single file or directory. If you combine it with recursive access of a directory tree, it can open a lot of possibilities. For example, you can create, read, identify, update, or delete files with a particular attribute or a group of attributes. Recursive access of a directory tree is a breeze with the supporting classes in NIO.2. Let's examine it in detail in the next section.

8.4 Recursively access a directory tree

[8.4] Recursively access a directory tree using the DirectoryStream and FileVisitor interfaces

Imagine you're ready to write your exam and want to view all the notes that you created using the sample application Exam FlashCards. Because you've been storing the text files corresponding to all the exam objectives in separate subdirectories, you can recursively access the directory, which includes all these subdirectories, thereby accessing all the text files.

Before we move further, let me briefly cover the difference between a recursive and nonrecursive access of a directory for directory OCP, as shown in figure 8.10.

A nonrecursive access of directory OCP will only access its immediate subdirectories: obj2 and obj8. On the other hand, a recursive access of the directory OCP will access all its subdirectories and files.

There are multiple ways to access a directory. Common and popular access algorithms are *breadth-first* and *depth-first*. In a breadth-first search algorithm, all the directories and files on the first level are accessed before moving on to the members on the next level, and so on until no more are remaining. In a depth-first algorithm, a direct subdirectory of the main directory is accessed recursively before moving forward with searching the next subdirectory. Further, there are multiple ways in which a directory and its members can be accessed in a depth-first search: *preorder*, *in-order*, and *postorder*.

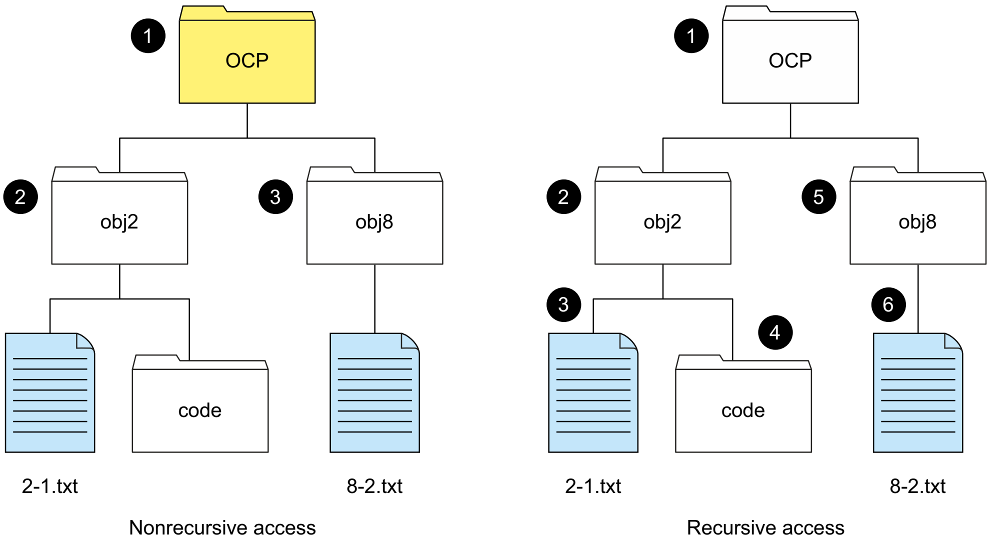


Figure 8.10 Sample directory structure to show recursive and nonrecursive access of directory OCP.

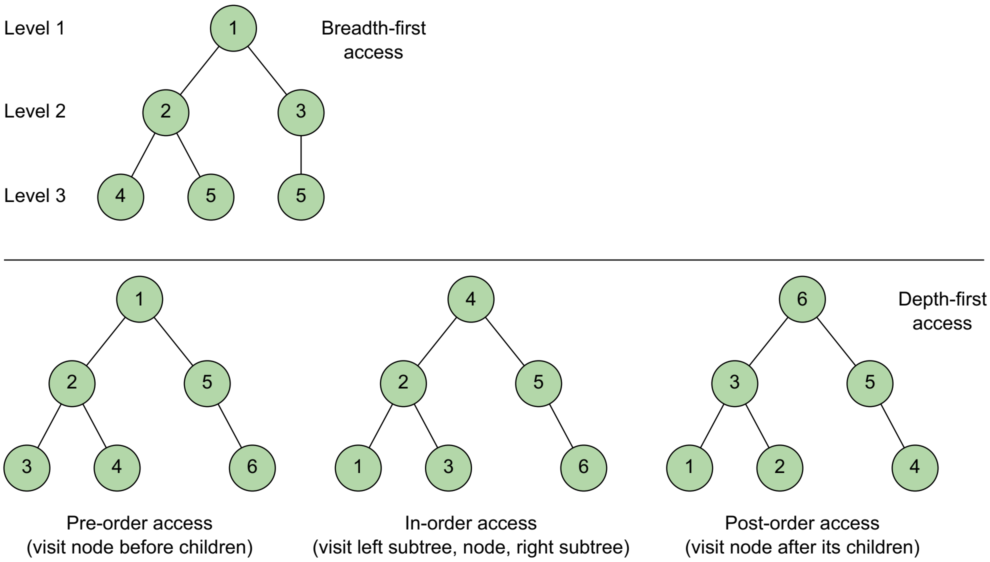


Figure 8.11 Breadth-first and depth-first directory tree access

You can compare the directory and its subdirectories and files with a *tree data structure*. A tree data structure has a root node, with multiple children. The preorder access will visit a node prior to accessing its children. The in-order access will access its left subtree, the node itself, and then its right subtree. The in-order access applies to binary tree structures, which have no more than two subtrees. In postorder access, a node is visited after its children are visited. Figure 8.11 shows example tree structures and the nodes numbered in the order they will be visited in a breadth-first access and in a depth-first access.

NOTE The exam won't query you on details of how to access a tree, breadth-first or depth-first. But it will help you to understand (and remember) how the access of a tree or a directory structure works.

Class `Files` defines overloaded method `walkFileTree()` to walk recursively through the specified path. To define the traversal behavior, this method accepts an object of `FileVisitor` interface, which is covered in the next section.

8.4.1 FileVisitor interface

You can use the `FileVisitor`, a generic interface to define the code that you want to execute during the traversal of a directory structure. When you traverse a directory structure, you can define what to do before or after you visit a directory, when you visit a file, or when access to a file is denied. The methods of the `FileVisitor` interface are listed in table 8.4.

Table 8.4 Methods of interface `FileVisitor<T>`

Method	Method signature
`FileVisitResult postVisitDirectory(T dir, IOException exc)`	Invoked for a directory after entries in the directory, and all of their descendants, have been visited.
`FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)`	Invoked for a directory before entries in the directory are visited.
`FileVisitResult visitFile(T file, BasicFileAttributes attrs)`	Invoked for a file in a directory.
`FileVisitResult visitFileFailed(T file, IOException exc)`	Invoked for a file that couldn't be visited.

Let's assume that you stored your exam tips in text files 2-1.txt and 8-2.txt using the directory structure shown in figure 8.12.

Let's create a class, say `MyFileVisitor`, which implements the `FileVisitor` interface and traverses this directory structure, storing the exam objective numbers (2-1 and 8-2) and their corresponding tips as a `List<String>` in a `HashMap<String, List<String>>`. The code is shown in listing 8.3.

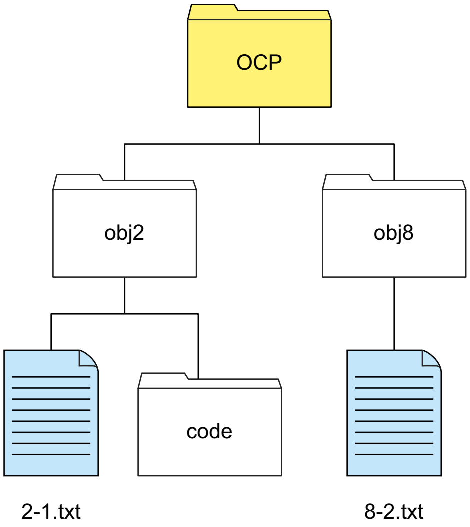


Figure 8.12 Example of directory structure used to store exam tips in text files 2-1.txt and 8-2.txt

Listing 8.3 Using FileVisitor to recursively access exam tips from files

```
class MyFileVisitor implements FileVisitor<Path> {

    Map<String, java.util.List<String>> flashcards = new HashMap<>();

    public FileVisitResult preVisitDirectory(Path dir,
                                         BasicFileAttributes attrs) {
        String dirName = dir.getFileName().toString();
        if (dirName.startsWith("code"))
            return FileVisitResult.SKIP_SUBTREE;
        else
            return FileVisitResult.CONTINUE;

    }
    public FileVisitResult postVisitDirectory(Path dir,
                                                   IOException exc) {
        return FileVisitResult.CONTINUE;
    }
    public FileVisitResult visitFile(Path file,
                                          BasicFileAttributes attrs) {

        String fileName = file.getFileName().toString();

        if (fileName.endsWith(".txt")) {
            java.util.List<String> tips = new ArrayList<>();

            try (
                BufferedReader reader = new BufferedReader(
                                    new FileReader(file.toFile()));
            ){
                String line = null;
                while((line = reader.readLine()) != null)
                    tips.add(line);
            }
            catch (Exception e) {
                System.out.println(e);
            }

            flashcards.put(fileName.substring(
                                     0, fileName.length()-4),tips);
        }
        return FileVisitResult.CONTINUE;
    }
    public FileVisitResult visitFileFailed(Path file, IOException exc) {
        System.out.println(exc);
        return FileVisitResult.SKIP_SUBTREE;
    }
    public Map<String, java.util.List<String>> getFlashCardsMap() {
        return flashcards;
    }
}
```

1 MyFileVisitor implements FileVisitor<Path>.

2 Hash map to store objective number and its corresponding exam tips as List<String>

3 Called prior to visiting subdirectories and files of a directory.

4 Called after visiting subdirectories and files of a directory.

5 Called when a file (not a directory) is visited.

6 Reads text files with exam tips using BufferedReader and adds them to List<String>.

7 Add objective number and its list of tips to a hash map

8 Called when a file can't be accessed.

NOTE In listing 8.3, I've deliberately excluded multiple checks to keep the example simple.

In listing 8.3, notice how all the methods of the `FileVisitor` interface return a constant of enum `FileVisitResult` to signal whether to continue with the traversal of a directory or to skip it (table 8.5 lists all the constant values for `FileVisitResult`).

The code at ❶ for class `MyFileVisitor` implements the `FileVisitor<Path>` interface. The code at ❷ defines a hash map to store the exam objectives as its keys and the corresponding exam tips as values. The code at ❸ for method `preVisitDirectory()` skips traversing the subtree for directories with the name "code" by returning `FileVisitResult.SKIP_SUBTREE`. The code at ❹ defines `postVisitDirectory()`, which is called after all the children of a directory are visited. Because the example didn't require any action in the method, it simply returns `FileVisitResult.CONTINUE`. The code at ❺ defines `visitFile()`, which is called when code accesses a file (not a directory). At ❻, method `visitFile()` includes a simple check of reading a file that ends with .txt; for this simple example, assume that all text files will define the exam objectives. It reads the file contents using a `BufferedReader` and adds each line to a `List<String>`. At ❼, the exam objective and the list of exam tips are added to the hash map. Method `visitFile()` is passed file attributes. If the file can't be visited for some reason—for example, reading the attributes failed due to an `IOException`—method `visitFileFailed()` is called, defined at ❽.

EXAM TIP Methods `preVisitDirectory()` and `visitFile()` are passed `BasicFileAttributes` of the path that they operate on. You can use these methods to query file or directory attributes.

Table 8.5 Enum `FileVisitResult` constants and their descriptions

Enum constant	Description
`CONTINUE`	Continue
`SKIP_SIBLINGS`	Continue without visiting the siblings of this file or directory
`SKIP_SUBTREE`	Continue without visiting the entries in this directory
`TERMINATE`	Terminate

You can also work with `SimpleFileVisitor` from the Java API, a class that implements the `FileVisitor` interface and all its methods. It saves you from implementing methods that you might not need.

8.4.2 Class SimpleFileVisitor

Class `SimpleFileVisitor` is a simple visitor of files with default behavior to visit all files and to rethrow I/O errors. It implements the `FileVisitor` interface (methods of

FileVisitor interface are listed in table 8.5). You can extend this class to implement methods for only the required behavior.

Imagine you need to recursively traverse a directory structure and print only the names of all the files. In this case, you can extend the SimpleFileVisitor class and override only the visitFile() method. Following is the code that accomplishes it:

```
import java.nio.file.*;
import java.nio.file.attribute.*;
class ListFileNames extends SimpleFileVisitor<Path> {
    public FileVisitResult visitFile(Path file,
                                BasicFileAttributes attributes) {
        System.out.println("File name :" + file.getFileName());
        return FileVisitResult.CONTINUE;
    }
}
public class TestListFiles2 {
    public static void main(String[] args) throws Exception {
        Path path = Paths.get("E:/OCPJavaSE7");
        ListFileNames listFileNames = new ListFileNames();
        Files.walkFileTree(path, listFileNames);
    }
}
```

❶ Class ListFileNames extends SimpleFileVisitor

❷ Initiate directory tree traversal

In the preceding code, at ❶, class ListFileNames declares to extend SimpleFileVisitor class. Because this example code doesn't need to define other directory traversal behavior, like pre- and post-directory visits or file visit failure, ListFileNames overrides only one method, visitFile(). The code at ❷ initiates the traversal of the directory structure (discussed in detail in the next section) referred by variable path in method main(). You can assign path to any valid directory on your system.

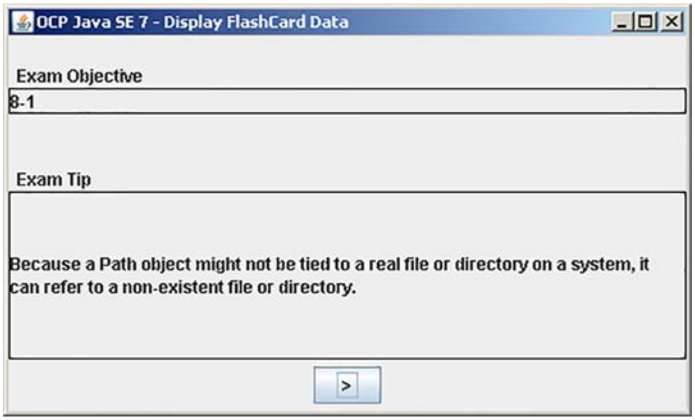
8.4.3 Initiate traversal for FileVisitor and SimpleFileVisitor

You can initiate traversal of a directory by calling the overloaded method walkFileTree() from class Files:

```
public static Path walkFileTree(Path start,
                                FileVisitor<? super Path> visitor)
                      throws IOException

public static Path walkFileTree(Path start,
                   Set<FileVisitOption> options,
                                int maxDepth,
                                FileVisitor<? super Path> visitor)
                      throws IOException
```

This method traverses through a directory structure, the root of which is specified by a Path object. The directory tree is traversed depth-first. The traversal is considered complete when either all the members of a tree are visited, any of method FileVisitor returns FileVisitResult.TERMINATE, or if an exception is thrown during the traversal.



Exam Objective

8-1

Exam Tip

Because a Path object might not be tied to a real file or directory on a system, it can refer to a non-existent file or directory.

Figure 8.13 Sample application displaying exam objective and exam tip

EXAM TIP A directory tree is traversed depth-first. But the order in which the subdirectories are traversed is unpredictable.

Let's see how you can initiate traversal of a directory and display the flash cards in the sample application. Figure 8.13 shows a screenshot of the sample application, displaying the exam tips from the text files as shown in the directory structure in figure 8.12.

Here's the code to display the exam tips (as shown in figure 8.13), persisted in the text files, as shown in figure 8.12.

Listing 8.4 Reading and displaying exam tips from files

```
import java.io.*;
import java.nio.file.*;
import java.nio.charset.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.nio.file.attribute.*;

public class DisplayFlashCards implements ActionListener{
    JFrame f = new JFrame("OCP Java SE 7 - Display FlashCard Data");
    JLabel lblObjectiveNo = null;
    JLabel lblFlashcard = null;
    JButton btnNext = null;

    private void buildUI() {
        lblObjectiveNo = new JLabel();
        lblFlashcard = new JLabel();
        lblObjectiveNo.setBorder(
                       BorderFactory.createLineBorder(Color.BLACK));
        lblFlashcard.setBorder(
                       BorderFactory.createLineBorder(Color.BLACK));
        btnNext = new JButton(" > ");
```

1 **Build UI of application**

```
    btnNext.addActionListener(this);

    JPanel topPanel = new JPanel();
    topPanel.setLayout(new GridLayout(6,1));
    topPanel.add(new JLabel(""));
    topPanel.add(new JLabel("  Exam Objective"));
    topPanel.add(lblObjectiveNo);
    topPanel.add(new JLabel(""));
    topPanel.add(new JLabel(""));
    topPanel.add(new JLabel("  Exam Tip"));

    JPanel middlePanel = new JPanel();
    middlePanel.setLayout(new BorderLayout());
    middlePanel.add(lblFlashcard);

    JPanel bottomPanel = new JPanel();
    bottomPanel.add(btnNext);

    JPanel mainPanel = new JPanel();
    mainPanel.setLayout(new BorderLayout());
    mainPanel.add(BorderLayout.NORTH, topPanel);
    mainPanel.add(BorderLayout.CENTER, middlePanel);
    mainPanel.add(BorderLayout.SOUTH, bottomPanel);

    f.getContentPane().setLayout(new BorderLayout());
    f.setSize(500, 550);
    f.getContentPane().add(mainPanel);
    f.setVisible(true);
}
```

1 Build UI of application

```
MyFileVisitor fileVisitor = new MyFileVisitor();
```

2 Instantiate MyFileVisitor, used by Files.walkFileTree().

```
Map<String, java.util.List<String>> flashcards = null;
Iterator<String> examObjIterator = null;
ListIterator<String> tipsIterator = null;
```

3 Variables to store flashcards as a hash map and iterators to its keyset and values.

```
public void accessAllExamTips() throws IOException {
    Files.walkFileTree(Paths.get("E:\\OCP"), fileVisitor);
    flashcards = fileVisitor.getFlashCardsMap();
}
```

4 Files.walkFileTree() is called to walk the file tree rooted at E:\\OCPJava7; accepts instance of FileVisitor.

```
public void initExamTips() {
    examObjIterator = flashcards.keySet().iterator();
    showNextTip();
}
```

5 Obtain iterator to all exam objectives, stored as keys in hash map flashcards; displays first exam tip.

```
private void showNextTip() {
    if(tipsIterator != null && tipsIterator.hasNext()) {
        lblFlashcard.setText("<html>" +
                        tipsIterator.next() + "</html>");
    }
    else {
        if(examObjIterator.hasNext()) {
            String currentExamObj = examObjIterator.next();
            lblObjectiveNo.setText(currentExamObj);

            tipsIterator = flashcards.get(currentExamObj)
                                               .listIterator();
            if(tipsIterator != null && tipsIterator.hasNext()) {
```

6 Shows the next tip

```
                    lblFlashcard.setText("<html>" +
                                  tipsIterator.next() + "</html>");
                }
            }
        }
    }
    public void actionPerformed(ActionEvent e) {
        try {
            showNextTip();
        }
        catch (Exception ioe) {
            JOptionPane.showMessageDialog(f, ioe.toString());
        }
    }

    public static void main(String[] args) throws Exception {
        DisplayFlashCards fc = new DisplayFlashCards();
        fc.buildUI();
        fc.accessAllExamTips();
        fc.initExamTips();
    }
}
```

❻ **Shows the next tip**

❼ **Called when Next button is clicked**

The preceding code creates a class, `DisplayFlashCards`, which reads text files stored in the directory structure as shown in figure 8.12, and displays the exam tips in a swing application. You can display the next exam tip by activating the Next button. The example shows how you can use `File.walkFileTree()` to walk a directory structure, read the exam tips from the text files, and store them in a hash map.

The code at ❶ builds the UI of the swing application. Don't worry if you can't follow all of it—swing components aren't on the exam. The code at ❷ instantiates `MyFileVisitor`, a class that implements the `FileVisitor` interface. Complete code for class `MyFileVisitor` is shown in listing 8.3. The code at ❸ defines variables to store the exam tips in a hash map, with iterators to its keyset (exam objectives) and values (`List<String>`). At ❹, `Files.walkFileTree()` initiates the traversal of the file tree rooted at E:\OCP using the `FileVisitor` instance. The code at ❺ obtains an `iterator` to all the exam objectives, stored as keys in the hash map `flashcards`. And also the first exam tip is displayed. Method `showNextTip()` displays the next available exam tip in the swing application ❻. The code at ❼ is called when the Next button is activated by a user. Method `main()` calls methods to build the UI of the swing application, traverses the mentioned path retrieving all exam tips from the file system, and displays the first tip.

NOTE This sample application doesn't include the Previous button, to keep the example code simple. If you want, you can add this functionality yourself to hone your skills with the collection classes.

8.4.4 DirectoryStream interface

The `DirectoryStream` interface can be used to iterate over all the files and directories in a directory. You can use an `Iterator` or `for-each` construct to iterate over a directory.

The order in which the directory contents are iterated is unpredictable. Following is an example that uses `DirectoryStream` with a `for-each` construct:

```
import java.io.*;
import java.nio.file.*;
public class DirStream{
    public static void main(String args[]) throws IOException {
        Path dir = Paths.get("E:/OCPJavaSE7");
        try (DirectoryStream<Path> stream=Files.newDirectoryStream(dir)) {
           for (Path value : stream) {
               System.out.println(value + ":" + Files.isDirectory(value));
           }
        }
    }
}
```

The preceding code outputs a list of all the files and directories in the directory referred by the variable `dir`. The code uses `Files.newDirectoryStream(Path)` to get a `DirectoryStream` object, the values of which are iterated over by using a `for-each` construct. What happens if you try to iterate a file (and not a directory) using `DirectoryStream`? In this case you'll get a runtime exception (`NotDirectoryException`).

EXAM TIP If you pass `Path` to a file (and not a directory) to `Files.newDirectoryStream()`, it will throw a runtime exception. The order of iteration of files and directories in a specified directory using `DirectoryStream` is unpredictable.

The next example uses an `Iterator` to iterate over the files and directories of a directory using method `Files.DirectoryStream(Path dir, String glob)` (`glob` is covered in the next section):

```
import java.io.*;
import java.nio.file.*;
import java.util.*;
public class DirStream{
    public static void main(String args[]) throws IOException {
        Path dir = Paths.get("E:/OCPJavaSE7/FileNIO");
        try (DirectoryStream<Path> stream = Files.newDirectoryStream(
                                                dir, "*.{txt,java}")) {
            Iterator iterator = stream.iterator();
            while (iterator.hasNext()) {
               System.out.println(iterator.next());
            }
        }
   }
}
```

The preceding code iterates over the files and directories of the directory referred to by the variable `dir`. The glob value *.{txt,java} passed to the overloaded method `newDirectoryStream()` limits the values that are iterated to the ones that end with "txt" or “java.” Method iterator() returns an Iterator associated with the DirectoryStream object. Iterator’s method hasNext() checks whether more elements are available or not and method next() retrieves the next object.

Other common examples that can be used to demonstrate walking a directory structure include copying, deleting, querying, or modifying the attributes of files and directories in a directory structure.

Another common requirement while traversing a directory structure is to search for files or directories with a pattern of, say, *.txt. Next we’ll cover the use of PathMatcher to find a file.

8.5 Using PathMatcher

[8.5] Find a file with the PathMatcher interface

Imagine you want to count all the text files in a directory, or count only the .pdf files whose names start with “report”, or perhaps delete all files whose names include “system”. To do so, you can create a pattern using special characters and then match your file or directory names against those patterns. You can define these patterns using regex, glob, or other syntax.

Regex syntax is covered in detail in chapter 5 of this book. A glob pattern supports a simpler form of pattern matching than the regex. It supports fewer special constructs. The asterisk (*) matches anything or nothing. The question mark (?) matches any single character and the characters classes (like [0-9] or [a-zA-Z]). A glob doesn’t support substring matches, which are supported by regex.

EXAM TIP In glob * matches zero or more characters. In regex .* matches zero or more characters.

To match a Path object with a pattern, you should create an object of java.nio.file.PathMatcher. PathMatcher is an interface with just one method: matches(). It returns true if a given path matches this matcher’s pattern:

```
boolean matches(Path path)
```

You can create a PathMatcher by calling FileSystem.getPathMatcher() and passing it the pattern to be matched:

```
public abstract PathMatcher getPathMatcher(String syntaxAndPattern)
```

The method parameter accepts a string of the form syntax:pattern. FileSystem supports glob and regex syntax. It can also support other syntax (which are beyond the scope of this book). Before moving on with an example of how to use the PathMatcher interface, let’s browse through some of the patterns that you can use to match Path objects, as described in table 8.6.

Table 8.6 A few examples of patterns that you can pass to `FileSystem.getPathMatcher` to match `Path` objects

Pattern to match	Description of pattern
`glob:*.{java,class}`	Match path names that end with extension .java or .class
`glob:*{java,class}`	Match path names that end with java or class. They might not include a dot just before (note the missing dot just before {).
`glob:*java,class`	Match path names that end with java,class; for example, Hellojava,class, and Hello.java,class (java,class doesn't evaluate to either java or class).
`glob:???.class`	Match path names that include exactly three characters followed by .class. For example, it will match abc.class, ab,.class, and a!b.class. It won't match abcd.class or abcde.class.
`glob:My[notes,tips,ans].doc`	Match an exact match of 'My' followed by any one character included within [], followed by .doc. For example, it matches Mys.doc and My,.doc. It doesn't match notes.doc, Mytips.doc, or Myans.doc.
`regex:MyPics[0-9].png`	Match MyPics followed by any digit, followed by .png. For example, MyPics0.png, MyPics1.png, and MyPics9.png.
`glob:/home/*/*`	Matches a path like home/shreys/code on UNIX platforms.
`glob:/home/**`	Matches a path like home/shreys/code and home/shreys on UNIX platforms.

In the sample application, recall that you stored all the exam tips in a text file with a name that reflects its exam objective number—for example, 2-1.txt, 8-2.txt, and 12-4.txt. The following code snippet uses class `PathMatcher` to determine whether a path matches the target regex pattern:

```
import java.nio.file.*;
class MyMatcher {
    public static void main(String args[]) {
        PathMatcher matcher = FileSystems.getDefault().getPathMatcher
                                        ("regex:[1-9]*[0-9]?-[1-9]?.txt");

        Path file = Paths.get("12-1.txt");
        if (matcher.matches(file)) {
            System.out.println(file);
        }
    }
}
```

❶ Get default FileSystem. Create PathMatcher object, passing it regex pattern.

❷ Create Path object to match against the pattern.

❸ file is matched against matcher's pattern.

Prints path

The code at ❶ defines a regex pattern that will match files and directories of the pattern `[1-9]*[0-9]?-[1-9]?.txt` against the `Path` object defined by the code at ❷. At ❸, `file` is matched against the `matcher`'s pattern, using `matcher.matches()`.

8.6 Watch a directory for changes

[8.6] Watch a directory for changes by using the WatchService interface

Imagine your application displays arrival and departure times of flights, which it reads from a file. What happens if this data is updated while it's being displayed by the application? In this case, you might want to update the displayed times also. To do so, your application should be able to detect when the file, which stores its data, is modified. The *WatchService API* can help you. It enables you to *watch* a directory for changes like addition, modification, or deletion of contents of a directory. A thread-safe option for watching directories for changes, WatchService API is one of the interesting additions to Java 7.

To see how you can work with the WatchService API, let's work with a simplified example of a command-line application (say, `WatchDirectories`) that watches changes to a directory and one of its directories and outputs the name of the affected file. The following sections will cover partial code of the application WatchDirectories to understand how to use the WatchService API. The application's complete code is included at the end.

8.6.1 Create WatchService object

The first step to watch a directory for changes is to create a `WatchService` object. A file system operates independently of a JVM. To create a `WatchService` object, you can use the `FileSystem` class, which provides an interface to a file system and is the factory for objects to access files and other objects in the file system. It defines method `newWatchService()` to create a `WatchService` object:

```
WatchService watchService = FileSystems.getDefault().newWatchService();
```

The next step is to register directories with the `WatchService` object.

8.6.2 Register with WatchService object

The directories that need to be watched for additions, modifications, or deletions must be registered with a `WatchService` object. A `WatchService` object watches a directory for the following events:

- `StandardWatchEventKinds.ENTRY_CREATE`—This event occurs when a new file or directory is created, moved, or renamed in the directory being watched.
- `StandardWatchEventKinds.ENTRY_DELETE`—This event occurs when an existing file or directory is deleted, moved, or renamed in the directory being watched.
- `StandardWatchEventKinds.ENTRY_MODIFY`—This event is platform-dependent. It usually occurs when contents of an existing file are modified. It can also occur if the attributes of a file or directory (in the directory being watched) are modified.
- `StandardWatchEventKinds.OVERFLOW`—This indicates that an event has been lost.

You can register multiple directories to be watched with the same WatchService object, by using method register() of Path. Multiple events can be registered in the same method call. Each registration process returns a WatchKey (discussed in the next section). The following example registers two directories (to be watched for multiple events) with the same WatchService object:

```
WatchService watchService = FileSystems.getDefault().newWatchService();
Path dir1 = Paths.get("E:/OCPJavaSE7");
Path dir2 = Paths.get("E:/OCPJavaSE7/8");

WatchKey regWatchKey = dir1.register(watchService,
                                     StandardWatchEventKinds.ENTRY_MODIFY,
                                     StandardWatchEventKinds.ENTRY_DELETE,
                                     StandardWatchEventKinds.ENTRY_CREATE);
dir2.register(watchService,
                                     StandardWatchEventKinds.ENTRY_MODIFY,
                                     StandardWatchEventKinds.ENTRY_DELETE,
                                     StandardWatchEventKinds.ENTRY_CREATE);
```

EXAM TIP You can watch a directory for changes. If you try to register a file for changes, you'll get a runtime exception (NotDirectoryException). Registering a directory for any event (create, modify, or delete) doesn't implicitly register its subdirectories.

In the next section, you'll see how the WatchService API stores the registered Watchable events, using the WatchKey objects.

8.6.3 Access watched events using WatchKey interface

The WatchKey object is a token that represents the registration of a Watchable object with a WatchService. As seen in the code in the previous section, a WatchKey object is created when you register your directory to be watched for create, modify, or delete events with a WatchService. A WatchKey can be in multiple states:

- *Ready*—A WatchKey is initially created with a ready state.
- *Signaled*—When an event is detected, the WatchKey is signaled and queued. It can be retrieved using WatchService's methods poll() or take().
- *Cancelled*—Calling method cancel() on a WatchKey or closing the WatchService cancels a WatchKey.
- *Valid*—A WatchKey in a ready or signaled state is in a valid state.

In the next section, you'll see how the detected events are processed using WatchService.

8.6.4 Processing events

A WatchService queues the registered events when they occur. The registered consumers can retrieve the queued WatchKeys and process the corresponding events. The WatchService interface defines method take() and overloaded method poll() to retrieve the queued WatchKeys. Once a key is processed, the consumer invokes the key's method reset() so that it can be signaled and requeued for further events.

To wait for registered events to occur, you need an infinite loop. Method take() of the WatchService interface retrieves and removes the next WatchKey, waiting if none are yet present. On the other hand, its poll() method retrieves and removes the next WatchKey, or returns null if none is present (no waiting). You can also use its overloaded method poll(long timeout, TimeUnit unit) to specify the waiting time if none is present.

When the registered events occur, methods WatchService's poll() or take() return to retrieve the queued WatchKeys. For each retrieved WatchKey, call the WatchKey's method pollEvents() to retrieve and remove all pending events for the key. Method pollEvents() returns a list of the events (WatchEvent) that were retrieved. Because you can register multiple paths and events with the same WatchService object, you can query the WatchEvent to determine the source of the event and its type, and process the event as required.

The following example watches two directories for create, modify, and delete events. Execute this code and, while it's running, use another system utility, like a file explorer, to modify, create, or delete files in the directories that this code is watching. The code will display the name of the newly created, modified, or deleted file or directory:

```
import java.io.*;
import java.nio.file.*;
import static java.nio.file.StandardWatchEventKinds.*;

public class WatchDirectories {

    public static void main(String args[]) {
        WatchService watchService = null;

        try {
            watchService = FileSystems.getDefault().newWatchService();
            Path dir1 = Paths.get("E:/OCPJavaSE7/");
            Path dir2 = Paths.get("E:/OCPJavaSE7/8");

            dir1.register(watchService,
                               ENTRY_MODIFY, ENTRY_DELETE, ENTRY_CREATE);
            dir2.register(watchService, ENTRY_DELETE);

            WatchKey watchKey = null;

            while(true) {
                watchKey = watchService.take();
                for (WatchEvent<?> watchEvent: watchKey.pollEvents()) {

                    WatchEvent.Kind<?> kind = watchEvent.kind();
                    Path path = ((WatchEvent<Path>)watchEvent).context();

                    if (kind == OVERFLOW) {
                        continue;
                    }
```

1 **Static import**

2 **Create WatchService**

3 **Register multiple directories and events with watchService**

4 **Wait for a registered event to occur**

5 **For a WatchKey, get a list of events**

6 **Get type of event**

7 **Retrieve event source**

8 **If event lost, skip remaining for loop**

```
                    else if (kind == ENTRY_CREATE) {
                        System.out.format("\nCreate - %s", path);
                    }
                    else if (kind == ENTRY_MODIFY) {
                        System.out.format("\nModify - %s", path );
                    }
                    else if (kind == ENTRY_DELETE) {
                        System.out.format("\nDelete - %s", path );
                    }
                }
                if(!watchKey.reset()) { break; }
            }
        }
        catch (IOException ioe) {
            System.out.println(ioe.toString());
        }
        catch (InterruptedException ioe) {
            System.out.println(ioe.toString());
        }
        finally {
            try {
                watchService.close();
            } catch (IOException e) {
                System.out.println(e);
            }
        }
    }
}
```

Process event ENTRY_CREATE ❾

Process event ENTRY_MODIFY ❿

Process event ENTRY_DELETE ⓫

Reset WatchKey so that it can be queued again ⓬

Close watchService ⓭

In the preceding code, the static import at ❶ enables the code to use constants `ENTRY_CREATE`, `ENTRY_MODIFY`, and `OVERFLOW` without prefixing them with `StandardWatchEventKinds`. The code at ❷ creates a `WatchService` object by calling method `newWatchService()` on `FileSystem`. The code at ❸ registers two directories, referred by paths `dir1` and `dir2`, with the same `watchService` object. The code uses an infinite loop to retrieve the queued `WatchKeys`. At ❹, `watchService.take()` waits for a registered event to occur and retrieves the corresponding `WatchKey`. For the `WatchKey` retrieved, the code at ❺ retrieves a list of events by calling `watchKey.pollEvents()`. At ❻ and ❼, the code retrieves the type and source of the event that has occurred. A file system might generate a lot of events, such that an event might be lost. The type of event reported is `OVERFLOW` ❽ if an event is lost. At ❾, ❿, and ⓫, the code processes events for creation (`ENTRY_CREATE`), modification (`ENTRY_MODIFY`), or deletion (`ENTRY_DELETE`) of a file or directory within the registered directories. At ⓬, the code resets the `watchKey` by calling `watchKey.reset()` so that it can be queued and signaled again. At the end, the code at ⓭ closes the `WatchService` object by calling `close()`.

The possibilities are immense in how you process the events, detected using `WatchService`, and are limited only by your needs or your imagination.

8.7 Summary

We started this chapter with an introduction to NIO.2 and the sample application. We covered the Path interface and worked with examples on how Path objects are created. You learned how to use Path methods to access path components, compare paths, convert relative paths to absolute paths, and resolve paths. Class Files is used to manipulate files and directories. It can be used to automatically check and create single or multiple files and directories, check for their existence, and copy, move, or delete them.

You learned how to access and update attributes of files and directories by using attribute views or class Files. NIO.2 defines a rich set of classes and interfaces that enable you to access and update basic, dos, Posix, Acl, and user-defined attributes of files or directories.

You can use the FileVisitor interface and classes SimpleFileVisitor and DirectoryStream to traverse and access a directory stream. You can find a file with a matching regex or glob pattern using the PathMatcher interface.

One of the most interesting features of NIO.2, the WatchService API enables you to watch a directory for creation, modification, and deletion of new or existing files and directories. When the registered events are triggered, they can be processed as required.

REVIEW NOTES

This section lists the main points covered in this chapter.

Path objects

- Objects of the Path interface are used to represent the path of files or directories in a file system.
- Because a Path object might not be tied to a real file or directory on a system, it can refer to a nonexistent file or directory.
- Apart from referring to a file or a directory, a Path object can also refer to a symbolic link. A symbolic link is a special file that refers to another file.
- When you read data from or write data to a symbolic link, you read from or write to its underlying target file. But if you delete a symbolic link, the target file isn't deleted.
- A Path can never be equal to a Path associated with another file system, even if they include exactly the same values.
- You can create Path objects by using Paths.get() or FileSystems.getDefault().getPath().
- You can convert a File instance to a Path object by calling toPath() on the File instance.
- Behind the scenes, both Paths.get() and File.toPath() call FileSystems.getDefault().getPath().

- Most of the Path methods perform syntactic operations. They manipulate the paths to a file or directory without accessing the file systems. They're logical operations on paths in memory.
- Methods getName(), getNameCount(), and subpath() don't use the root directory of a path. Method getRoot() returns the root of an absolute path and null for relative paths.
- The Path methods that accept positions throw an IllegalArgumentException at runtime for invalid positions. For example, getName() and subpath() throw an IllegalArgumentException if you pass invalid path positions to them.
- You can compare paths lexicographically using method compareTo(Path).
- To check whether a path starts or ends with another path, you can use startsWith(String), startsWith(Path), endsWith(String), and endsWith(Path).
- Methods startsWith() and endsWith() are overloaded—startsWith(String), startsWith(Path), endsWith(String), and endsWith(Path). So if you pass null to these methods, you'll get a compiler error.
- The method name to retrieve the absolute path from a Path object is toAbsolutePath() and not getAbsolutePath().
- You can remove redundant path values by calling method normalize() on Path.
- Path is immutable and calling normalize() on a Path object doesn't change its value.
- Method normalize() doesn't check the actual file system to verify if the file (or directory) the resulting path is referring to actually exists.
- If a Path object includes redundancies like . or .., calling information retrieval methods like subpath() or getName() will include these redundancies in the returned values.
- The overloaded methods resolve(String) and resolve(Path) are used to join a relative path to another path. If you pass an absolute path as a parameter, this method returns the absolute path.
- To retrieve the path to a file in the same directory, say, to create its copy or to rename it, you can use the overloaded methods resolveSibling(String) and resolveSibling(Path).
- Method resolveSibling() resolves a given path against a path's parent. If the given path is an absolute path, this method returns the absolute path. If you pass it an empty path, it returns the parent of the path.
- Methods resolve() and resolveSibling() don't check the actual file system to verify if the file (or directory) the resulting path is referring to actually exists.
- To construct a path between two Path objects, use method relativize(). It can be used to construct a path between two relative or absolute Path objects.
- You can't create a path from a relative path to an absolute path and vice versa using method relativize(). If you do so, you'll get a runtime exception (IllegalArgumentException).

- Method `relativize()` doesn't check the actual file system to verify if the file (or directory) the resulting path is referring to actually exists.

Class Files

- Class `java.nio.file.Files` defines static methods for manipulating files and directories.
- Method `createFile()` atomically checks for the existence of the file specified by the method parameter path and creates it if it doesn't exist.
- Method `createFile()` fails and throws an exception if the file already exists, a directory with the same name exists, its parent directory doesn't exist due to an I/O error, or the specified file attributes can't be set.
- Method `createDirectory()` creates the specified directory (not the parent directory) on the file system. It also atomically checks for the existence of the specified directory and creates it if it doesn't exist.
- Method `createDirectory()` throws an exception if a file or directory exists with the same name, its parent directory doesn't exist due to an I/O error, or the specified directory attributes can't be set.
- Method `createDirectories()` creates a directory, creating all nonexistent parent directories.
- If the target directory already exists, `createDirectories()` doesn't throw any runtime exception. It throws an exception if the specified `dir` exists but isn't a directory, an I/O occurs, or the specified directory attributes can't be set.
- Specifying file or directory attributes is optional with methods `createFile()`, `createDirectory()`, and `createDirectories()`. All these methods declare to throw an `IOException`, which is a checked exception.
- You can check for the existence of a file or directory referred by a `Path` object using methods `exists()` and `notExists()`.
- Method `notExists()` isn't a complement of method `exists()`. It returns `true` if a target doesn't exist or `false` if its existence can't be determined. If these methods can't determine the existence of a file, both of them will return `false`.
- Class `Files`'s overloaded method `copy()` enables you to read from `InputStream` and write to a `Path` object, read from a `Path` object and write to `OutputStream`, and read from and write to `Path` objects.
- `Files.copy()` can copy only files, not directories. If the source is a directory, then in the target an empty directory is created (without copying the entries in the directory). This method returns a `long` or `Path` value, not a `boolean` value.
- If you use a relative path to the target file, the file is created relative to your Java class file (.class) and not relative to the source file (passed as a parameter to method `Files.copy()`).
- To move files or directories programmatically, you can use `Files.move()`, which moves or renames a file to a target file.

- You can only move empty directories using method `Files.move()`. You can rename a nonempty directory by using `Files.move()`. But you can't move a file or directory to a nonexisting directory.
- To delete a directory or a file referred to by a `Path` object, you can use methods `delete(Path)` or `deleteIfExists(Path)`.
- If you try to delete a directory that isn't empty, methods `delete(Path)` and `deleteIfExists(Path)` will throw a `DirectoryNotEmptyException`.
- If you try to delete a nonexistent file or directory using method `delete()`, it will throw a `NoSuchFileException`. But method `deleteIfExists()` won't throw an exception if the file or directory at the specified path doesn't exist—rather, it will return `false`.
- Methods `delete()` and `deleteIfExists()` can be used to delete files and (nonempty) directories.

Files and directory attributes

- Class `Files` defines static methods to access individual attributes of a file or directory referred by a `Path`.
- You can access the individual attributes of a file or directory by using method `Files.getAttribute()`, passing to it the name of the attribute as a string value. To modify the attributes of an existing file or directory, you can use `Files.setAttribute()`.
- You can access a group of file attributes by calling `Files.getFileAttributeView()` or `Files.readAttributes()`.
- The `BasicFileAttributes` interface defines methods to access the basic attributes that should be supported by all the file systems.
- The `BasicFileAttributeView` interface can be used to modify the basic attributes.
- The `DosFileAttributes` interface extends `BasicFileAttributes` and defines methods to access attributes specific to Windows files and directories.
- The `DosFileAttributeView` interface defines methods to modify the DOS file attributes.
- The `PosixFileAttributes` interface also extends `BasicFileAttributes` and defines methods to access attributes related to the POSIX family of standards, like Linux or UNIX.
- The `PosixFileAttributeView` interface defines methods to modify attributes related to the POSIX family.
- Available only for Windows OS, the `AclFileAttributeView` interface supports access and updates of a file's ACL.
- The `FileOwnerAttributeView` interface supports access and updates to the owner of a file or directory. It is supported by all systems that support the concept of file owners.

- The `UserDefinedFileAttributeView` interface supports the addition, modification, and deletion of user-defined metadata.
- The `BasicFileAttributes`, `DosFileAttributes`, and `PosixFileAttributes` interfaces define methods to access attributes. They don't define methods to modify (or set) the attributes.
- The `BasicFileAttributeView`, `DosFileAttributeView`, `PosixFileAttributeView`, `AclFileAttributeView`, `FileOwnerAttributeView`, and `UserDefinedFileAttributeView` interfaces can be used to update attribute values.
- If a file system doesn't support an attribute view, `Files.getFileAttributeView()` returns `null`. If a file system doesn't support an attribute set, `File.readAttributes()` will throw a runtime exception.
- If an underlying system doesn't support all the basic timestamps—that is, `creationTime`, `lastAccessTime`, and `lastModifiedTime`—it might return system-specific information.
- Methods `Files.setAttribute()` and `Files.getAttribute()` throw an `IllegalArgumentException` or `UnsupportedOperationException` if you pass them an invalid or unsupported attribute.
- The `DosFileAttributes` interface makes the following attributes available:
  - `archive`
  - `hidden`
  - `readonly`
  - `system`
- The DOS attributes are available on a Windows system only. Trying to access them on other systems will throw a runtime exception.
- When you read all DOS attributes using method `Files.readAttributes()`, you also read the basic attributes.
- The POSIX attributes are as follows:
  - `group`
  - `owner`
  - `permissions`
- The POSIX attributes are available on the POSIX family of standards, such as UNIX and LINUX. Trying to access them on other systems will throw a runtime exception.
- To read or update the owner of a file or directory you can use the `AclFileAttributeView`, `FileOwnerAttributeView`, and `PosixFileAttributeView` interfaces.
- The `UserDefinedAttributeView` interface can be used to add, delete, access, and modify additional user-defined attributes to or from a file or directory. It defines methods `delete(String)`, `list()`, `read(String, ByteBuffer)`, `size(String)`, and `write(String, ByteBuffer)` to, respectively, delete, list, read, get an attribute's size, and write attribute values.

Recursively access a directory tree

- Class Files defines overloaded method walkFileTree() to walk recursively through the specified path. To define the traversal behavior, this method accepts an object of the FileVisitor interface.
- You can use the FileVisitor, a generic interface, to define the code that you want to execute during the traversal of a directory structure. When you traverse a directory structure, you can define what to do before or after you visit a directory, when you visit a file, or when access to a file is denied.
- Method postVisitDirectory() is invoked for a directory after entries in the directory and all of their descendants have been visited.
- Method preVisitDirectory() is invoked for a directory before entries in the directory are visited.
- Method visitFile() is invoked for a file in a directory.
- Method visitFileFailed() is invoked for a file that couldn't be visited.
- Methods preVisitDirectory() and visitFile() are passed BasicFileAttributes of the path that they operate on. You can use these methods to query file or directory attributes.
- Class SimpleFileVisitor is a simple visitor of files with default behavior to visit all files and to rethrow I/O errors. It implements the FileVisitor interface.
- You can initiate traversal of a directory by calling the overloaded method walkFileTree() from class Files.
- The DirectoryStream interface can be used to iterate over all the files and directories in a directory. You can use an Iterator or for-each construct to iterate over a directory. The order in which the directory contents are iterated is unpredictable.
- If you pass Path to a file (and not a directory) to Files.newDirectoryStream(), it will throw a runtime exception. The order of iteration of files and directories in a specified directory using DirectoryStream is unpredictable.

Using PathMatcher

- You can match your file or directory names against a regex or glob pattern by using PathMatcher.
- A glob pattern supports a simpler form of pattern matching than the regex. It supports fewer special constructs.
- In glob, * matches zero or more characters. In regex, .* matches zero or more characters.
- To match a Path object with a pattern, you should create an object of java.nio.file.PathMatcher. PathMatcher is an interface with just one method: matches(). It returns true if a given path matches this matcher's pattern.
- You can create a PathMatcher by calling FileSystem.getPathMatcher() and passing it the pattern to be matched.

Watch a directory for changes

- WatchService enables you to watch a directory for changes like addition, modification, or deletion of contents of a directory.
- The first step to watch a directory for changes is to create a WatchService object.
- A WatchService object watches a directory for the following events:
  - StandardWatchEventKinds.ENTRY_CREATE—This event occurs when a new file or directory is created, moved, or renamed in the directory being watched.
  - StandardWatchEventKinds.ENTRY_DELETE—This event occurs when an existing file or directory is deleted, moved, or renamed in the directory being watched.
  - StandardWatchEventKinds.ENTRY_MODIFY—This event is platform-dependent. It usually occurs when contents of an existing file are modified. It can also occur if the attributes of a file or directory (in the directory being watched) are modified.
  - StandardWatchEventKinds.OVERFLOW—This indicates that an event has been lost.
- You can register multiple directories to be watched with the same WatchService object by using method register() of Path.
- You can watch a directory for changes. If you try to register a file for changes, you'll get a runtime exception (NotDirectoryException). Registering a directory for any event (create, modify, or delete) doesn't implicitly register its subdirectories.
- The WatchKey object is a token that represents the registration of a Watchable object with a WatchService. A WatchKey object is created when you register your directory to be watched for create, modify, or delete events with a WatchService.
- A WatchKey can be in multiple states:
  - *Ready*—A WatchKey is initially created with a ready state.
  - *Signaled*—When an event is detected, the WatchKey is signaled and queued. It can be retrieved using method WatchService's poll() or take().
  - *Cancelled*—Calling method cancel() on a WatchKey or closing the WatchService cancels a WatchKey.
  - *Valid*—A WatchKey in a ready or signaled state is in a valid state.
- A WatchService queues the registered events when they occur. The registered consumers can retrieve the queued WatchKeys and process the corresponding events.
- The WatchService interface defines method take() and overloaded method poll() to retrieve the queued WatchKeys. Once a key is processed, the consumer invokes the key's method reset() so that it can be signaled and requeued for further events.

- Method `take()` of the `WatchService` interface retrieves and removes the next `WatchKey`, waiting if none is yet present.
- Method `poll()` of the `WatchService` interface retrieves and removes the next `WatchKey`, or `null` if none is present (no waiting). You can also use its overloaded method `poll(long timeout, TimeUnit unit)` to specify the waiting time if none is present.
- For each retrieved `WatchKey`, you can call the `WatchKey`'s method `pollEvents()` to retrieve and remove all pending events for the key.
- Method `pollEvents()` returns a list of the events (`WatchEvent`) that were retrieved.

Because you can register multiple paths and events with the same `WatchService` object, you can query the `WatchEvent` to determine the source of the event and its type, and process the event as required.

SAMPLE EXAM QUESTIONS

Q 8-1. Given the following directory structure

```
root dir
|- MyDir
   |- 8_1.java
   |- 8_1.class
   |- Hello.txt
```

which options when inserted at `/* INSERT CODE HERE */`will delete the file represented by the `Path` object `path`?

```
import java.nio.file.*;
class Q8_1 {
    public static void main(String... args) throws Exception {
        Path path = Paths.get("Hello.txt");
        Files.delete(/* INSERT CODE HERE */);
    }
}
```

a `path.toAbsolutePath()`

b `path.resolveSibling("Q8_1.class")`

c `path.toRealPath()`

d `path.resolve()`

Q 8-2. What is the output of the following code?

```
class Q8_2 {
    public static void main(String... args) {
        Path path1 = FileSystems.getDefault().getPath("/main/sub/notes/
     file.txt");
        Path path2 = Paths.get("code/Hello.java");
        System.out.println(path1.getRoot()+ ":" + path2.getRoot());
        System.out.println(path1.getName(0) + ":" + path2.getName(0));
        System.out.println(path1.subpath(1, 3) + ":" + path2.subpath(0,1));
    }
}
```

a
```
\main:null
file.txt:Hello.java
sub\notes\file.txt:code\Hello.java
```

b
```
\:null
file.txt:Hello.java
sub\notes:code
```

c
```
main:code
file.txt:Hello.java
sub\notes:code
```

d
```
\:null
main:code
sub\notes:code
```

e None of the above

f Compilation fails

Q 8-3. Which definition of class `MyFileVisitor` will enable the following code to delete recursively all files that are smaller than 100 bytes in size?

```
Path path = Paths.get("/myHomeDir");
Files.walkFileTree(path, new MyFileVisitor());
```

a
```
class MyFileVisitor implements FileVisitor<Path> {
     public FileVisitResult visitFile(Path file, BasicFileAttributes
attrs) throws IOException{
         if (attrs.size() <= 100) {
             Files.delete(file);
         }
         return FileVisitResult.CONTINUE;
     }
}
```

b
```
class MyFileVisitor extends SimpleFileVisitor<Path> {
     public FileVisitResult visitFile(Path file, BasicFileAttributes
attrs) throws IOException{
         if (attrs.size() <= 100) {
             Files.delete(file);
         }
         return FileVisitResult.CONTINUE;
     }
}
```

c
```
class MyFileVisitor implements FileVisitor<Path> {
     public FileVisitResult visitFile(Path file,
                      BasicFileAttributes attrs) throws IOException{
         if (attrs.size() <= 100) {
             Files.delete(file);
         }
```

```
            return FileVisitResult.CONTINUE;
        }
        public FileVisitResult preVisitDirectory(Path dir,
                                      BasicFileAttributes attrs) {}
        public FileVisitResult postVisitDirectory(Path dir,
                                      IOException exc) {}
        public FileVisitResult visitFileFailed(Path file,
                                      IOException exc) {}
    }
```

d

```
    class MyFileVisitor extends SimpleFileVisitor {
        public FileVisitResult visitFile(Path file, BasicFileAttributes
    attrs) throws IOException{
            if (attrs.getSize() <= 100) {
                Files.deleteFile(file);
            }
            return FileVisitResult.CONTINUE;
        }
    }
```

e Compilation error

f Runtime exception

Q 8-4. What is the output of the following code?

```
Path path1 = Paths.get("MyDir/hello.java");
Path path2 = Paths.get("FriendDir/code");
Path path3 = path1.relativize(path2);
for (Path path : path3)
    System.out.println(path);
```

a

```
..
..
FriendDir
code
```

b

```
..
..
MyDir
hello.java
```

c

```
FriendDir
code
```

d

```
MyDir
hello.java
```

e

```
..
MyDir
```

f Compilation error

g Runtime exception

Q 8-5. Select the correct statements:

- a `Paths.get()` can throw a `FileNotFoundException`.
- b `getParent()` in `Path` returns an empty path if a path doesn't have a parent.
- c `getNameCount()` in `Path` excludes the root of a path.
- d For absolute paths, `toAbsolutePath()` in `Path` returns `null`.

Q 8-6. Which code options when inserted at `/* INSERT CODE HERE */` will copy the specified file from the specified source to the destination?

```
public static void copyFile (String src, String dest) throws IOException{
    /* INSERT CODE HERE */
    java.nio.file.Files.copy(source, destination);
}
```

a
```
Path source = Paths.get(src);
Path destination = Paths.get(dest);
```

b
```
FileInputStream source = new FileInputStream(new File(src));
Path destination = Paths.get(dest);
```

c
```
Path source = Paths.get(src);
BufferedOutputStream destination = new BufferedOutputStream(
                                        new FileOutputStream(dest));
```

d
```
FileInputStream source = new FileInputStream(new File(src));
FileOutputStream destination = new FileOutputStream(new File(dest));
```

Q 8-7. Which options are true for the following code?

```
public static void check(Path path) throws Exception {
    if (!Files.exists(path)) {
        System.out.println("Not exists");
        Files.createDirectories(path.getParent());
        Files.createFile(path);
    }
    else if (!Files.notExists(path)) {
        System.out.println("exists");
        Files.delete(path);
    }
    else {
        System.out.println("can never reach here");
    }
}
```

- a `check()` will output `"Not exists"` and try to create a new file, if it doesn't exist.
- b `check()` will output `"exists"` and try to delete the file if it exists. It won't delete a directory.
- c `check()` can never output `"can never reach here"`.
- d `check()` can throw an `IOException`, `NoSuchFileException`.

Q 8-8. Select the incorrect statements for the following code:

```
public static void toggleFile(Path file) throws Exception {
    DosFileAttributes attr = Files.readAttributes(file,
                                             DosFileAttributes.class);
    if (attr.isHidden()) {                                    //line1
        Files.setAttribute(file, "dos:hidden", Boolean.FALSE);    //line2
    }
    else
        Files.setAttribute(file, "dos:hidden", Boolean.TRUE);     //line3
}
```

- **a** `toggleFile()` changes the attribute of a file or directory to `hidden` if it isn't, and vice versa (when executed on a Windows system).
- **b** If `"dos:hidden"` is changed to `"hidden"` on lines 2 and 3, `toggleFile()` can change the `"hidden"` attributes of files or directories on all OSs.
- **c** Replacing `attr.isHidden()` with `attr.hidden()` will not modify the code results.
- **d** `toggleFile` can throw an `UnsupportedOperationException`.

Q 8-9. What is the output of the following code?

```
Path path1 = Paths.get("/pin/./bin/tub/../code/../Hello.java");
System.out.println(path1.normalize());
```

- **a** `\pin\bin\tub\Hello.java`
- **b** `\pin\bin\tub\code\Hello.java`
- **c** `\bin\tub\code\Hello.java`
- **d** `\bin\Hello.java`
- **e** `\pin\bin\Hello.java`

Q 8-10. For which string values will the method `match` return `true`?

```
public static boolean match(String filename) throws Exception {
    Path path = Paths.get(filename);
    PathMatcher matcher = FileSystems.getDefault().getPathMatcher("glob:/
     mydir/*/*");
    return (matcher.matches(path));
}
```

- **a** `/mydir/notes/java/String`
- **b** `/mydir/notes/java`
- **c** `/mydir/notes`
- **d** `mydir/notes/java/String`
- **e** `mydir/notes/java`

Q 8-11. Given the following code, which code options can be inserted at `/*INSERT CODE HERE*/` without any compilation errors?

```
Path file = Paths.get("/mydir");
PosixFileAttributes attr = Files.readAttributes(file,
     PosixFileAttributes.class);
System.out.println(/*INSERT CODE HERE*/);
```

- **a** `attr.group()`
- **b** `attr.owner()`
- **c** `attr.size()`
- **d** `attr.permissions()`
- **e** `attr.creationTime()`
- **f** `attr.isReadOnly()`
- **g** `attr.isRegularFile()`

ANSWERS TO SAMPLE EXAM QUESTIONS

A 8-1. a, c

[8.1] Operate on file and directory paths with the Path class

Explanation: Option (b) is incorrect. The call `path.resolveSibling("Q8_1.class")` will resolve the path from file Q8_1.class against the parent directory of file Hello.txt. Because these files exist in the same directory, a (valid) path to file Q8_1.class is returned. So `Files.delete()` will not delete file Hello.txt, but Q8_1.class instead.

Option (d) will fail compilation. Method `resolve()` accepts either a `Path` or a `String`, resolving it against the path on which it's called.

A 8-2. d

[8.1] Operate on file and directory paths with the Path class

Explanation: The `getRoot` method returns the root of a path for an absolute path and `null` for a relative path. Because `"/main/sub/notes/file.txt"` starts with a `/`, it's considered an absolute path and `getRoot` will return / or \ depending on your underlying OS (\ for Windows and / for UNIX).

Method `getName()` excludes the root of a path and returns the element of this path, specified by the index parameter to this method. The element closest to the root in the directory hierarchy has an index of `0` and the element farthest from the root has an index of `count-1` (where count is the total number of `Path` elements).

Method `subpath()` (small p) returns the subsequence of the name elements of a `Path`, starting at the method parameter `startIndex` (inclusive) up to the name element at `endIndex` (exclusive).

A 8-3. b

[8.4] Recursively access a directory tree using the DirectoryStream and FileVisitor interfaces

Explanation: Option (a) is incorrect because it won't compile. The `FileVisitor` interface defines four methods: `preVisitDirectory()`, `postVisitDirectory()`, `visitFile()`, and `visitFileFailed()`. Class `MyFileVisitor` in this option implements only one method.

Option (c) is incorrect. Though class `MyFileVisitor` implements all the methods of the `FileVisitor` interface, methods `preVisitDirectory()`, `postVisitDirectory()`, and `visitFileFailed()` don't return a value. All these methods must return a value of type `FileVisitResult`.

Option (d) is incorrect because the correct method name to get the file size from `BasicFileAttributes` is `size`. The correct method name to delete a file using class `Files` is `delete`.

A 8-4. a

[8.1] Operate on file and directory paths with the Path class

Explanation: `path1.relativize(path2)` creates a relative path from `path1` to `path2`. To navigate from relative `path1` (MyDir/hello.java) to relative `path2` (FriendDir/code), you need to do the following steps:

1. (apply ..)—Navigate to the parent directory of MyDir/hello.java—that is, MyDir.
2. (apply ..)—Navigate to the parent directory of MyDir.
3. (apply /FriendDir)—Navigate to /FriendDir.
4. (apply /code)—Navigate to code.

The `Path` interface extends the `Iterable` interface. A `Path` object can iterate over its name elements using the `for-each` loop.

A 8-5. c

[8.1] Operate on file and directory paths with the Path class

Explanation: Option (a) is incorrect. A `Path` object might refer to a nonexistent file or directory. It can throw an `InvalidPathException` if the path string cannot be converted to a `Path` object.

Option (b) is incorrect because `getParent()` returns `null` if a `Path` object doesn't have a parent.

Option (d) in incorrect. For absolute paths, `toAbsolutePath()` returns the path itself. For a relative path, the absolute path is resolved in an implementation-dependent manner.

A 8-6. a, b, c

[8.2] Check, delete, copy, or move a file or directory with the Files class

Explanation: The overloaded `copy()` method in class `Files` can copy a source to a destination in the following formats:

- From `InputStream` to `Path`
- From `Path` to `OutputStream`
- From `Path` to `Path`

Therefore, options (a), (b), and (c) are correct. `FileInputStream` extends `InputStream`, `FileOutputStream` extends `OutputStream`, and `BufferedOutputStream` extends `OutputStream`. So their objects can be passed to `Files.copy()`.

Option (d) is incorrect because method `copy()` in class `Files` doesn't copy `InputStream` to `OutputStream`.

A 8-7. a, d

[8.2] Check, delete, copy, or move a file or directory with the Files class

Explanation: Option (b) is incorrect because `Files.delete()` can be used to delete files and empty directories. If you try to delete a nonempty directory, `Files.delete()` will throw a `DirectoryNotEmptyException`.

Option (c) is incorrect because both `exists()` and `notExists()` can return `false` if the underlying system can't determine the existence of a file.

A 8-8. b, c

[8.3] Read and change file and directory attributes, focusing on the `BasicFileAttributes`, `DosFileAttributes`, and `PosixFileAttributes` interfaces

Explanation: Option (a) is a correct statement because method `toggleFile()` changes the attribute `"hidden"` of a file to `true` if it's `false` and vice versa.

Option (b) is an incorrect statement. If `"dos:hidden"` is changed to `"hidden"`, the code will throw a runtime exception. `"hidden"` is a DOS file attribute. DOS and POSIX attributes must be prefixed with `dos:` or `posix:` in method `Files.setAttribute`. You don't need to prefix the basic file attributes with `basic:`. In the absence of any prefix, `basic:` is added automatically.

Option (c) is an incorrect statement because the correct method name to access the attribute `"hidden"` from `DosFileAttributes` is `isHidden`.

Option (d) is a correct statement because if you execute this code on an OS that doesn't support `DosFileAttributes`, `Files.readAttributes()` will throw an `UnsupportedOperationException`.

A8-9. e

[8.1] Operate on file and directory paths with the Path class

Explanation: Method normalize() removes the redundancies like "." (current directory) and ".." (parent directory) from a path. Because "." denotes the current directory, it's simply removed by normalize(). ".." is only removed if it is preceded by a non-".." name. In the following example, no redundancies are removed:

```
System.out.println(Paths.get("../../OCPJava7/8.1.txt").normalize());
```

A8-10. b

[8.5] Find a file with the PathMatcher interface

Explanation: The glob pattern /mydir/*/* evaluates to root (/), followed by dir mydir, followed by any two subdirectories. Only option (b) matches this pattern.

A8-11.a, b, c, d, e, g

[8.3] Read and change file and directory attributes, focusing on the BasicFileAttributes, DosFileAttributes, and PosixFileAttributes interfaces

Explanation: The PosixFileAttributes interface extends the BasicFileAttributes interface, so a PosixFileAttributes object can access methods creationTime(), size(), and isRegularFile() defined in the BasicFileAttributes interface. Method isReadOnly() is defined in the DosFileAttributes interface and it can't be used with an object of PosixFileAttributes.