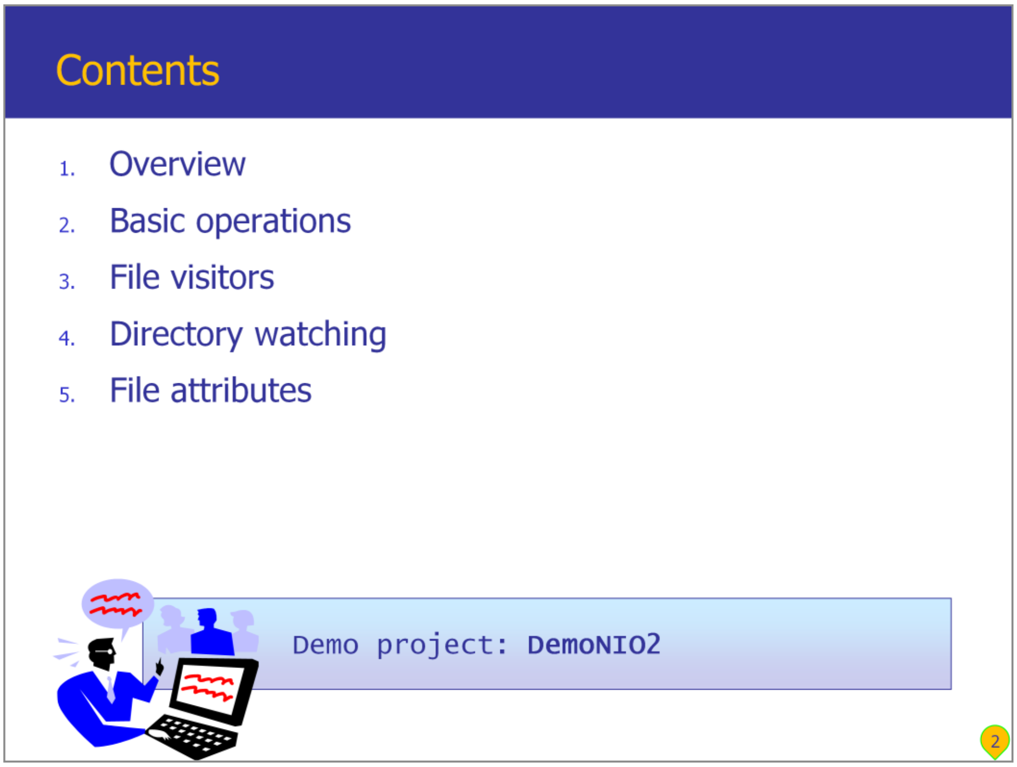
# NIO2

olsen software ltd

NIO 2 means "New I/O 2". This is a new set of file-related classes that provide more powerful access to the file system than was possible using the java.io file APIs.



## Contents

Demo project: DemoNIO2

2

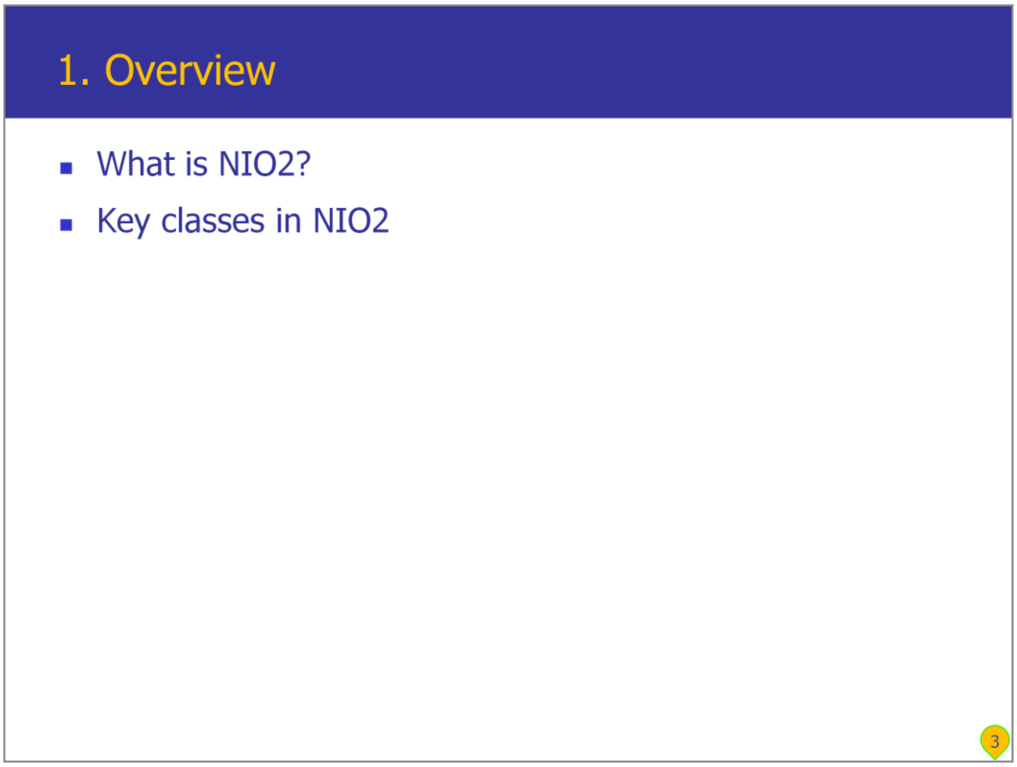
Section 1 provides sets the scene for the rest of the chapter, with a quick overview of the purpose, scope, and features in the NIO2 API.

Section 2 looks at some core classes and interfaces in NIO2 that allow you to perform basic operations, such as inspecting the properties of a file or directory, and copying/deleting/moving a file or directory.

Section 3 shows how to use NIO2 to walk a tree of directories and sub-directories, and perform some processing on all the files found.

Section 4 shows how to use NIO2 to watch directories for any changes, e.g. files added or removed.

Section 5 shows how to use NIO2 to access detailed attributes on files and directories, something that wasn't possible in the standard Java IO API.

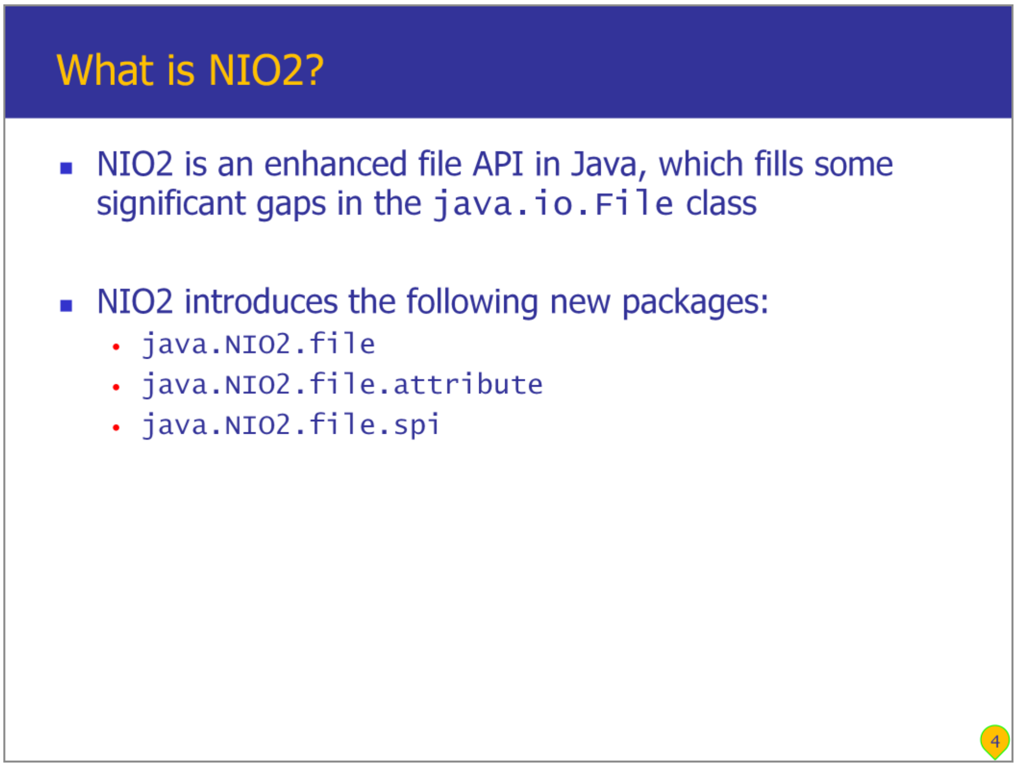


## 1. Overview

- What is NIO2?
- Key classes in NIO2

3

This quick section gets the ball rolling with some definition of terms and a summary of what's available in NIO2.



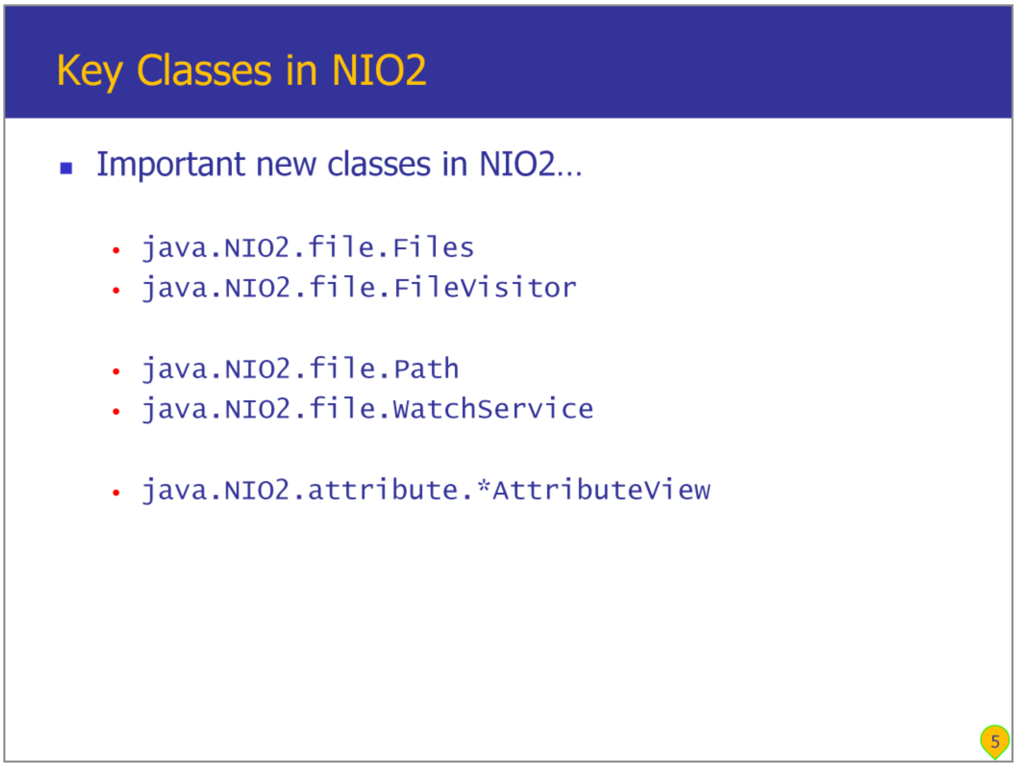
## What is NIO2?

- NIO2 is an enhanced file API in Java, which fills some significant gaps in the `java.io.File` class
- NIO2 introduces the following new packages:
  - `java.NIO2.file`
  - `java.NIO2.file.attribute`
  - `java.NIO2.file.spi`

4

NIO2 addresses many shortcomings in the standard `java.io.File` class:

- `java.io.File` doesn't handle filenames in a way that works consistently across platforms.
- `java.io.File` doesn't support efficient file-attribute access.
- `java.io.File` doesn't allow apps to take advantage of file-system-specific features (e.g. symbolic links) when available.



## Key Classes in NIO2

- Important new classes in NIO2…
  - `java.NIO2.file.Files`
  - `java.NIO2.file.FileVisitor`

  - `java.NIO2.file.Path`
  - `java.NIO2.file.WatchService`

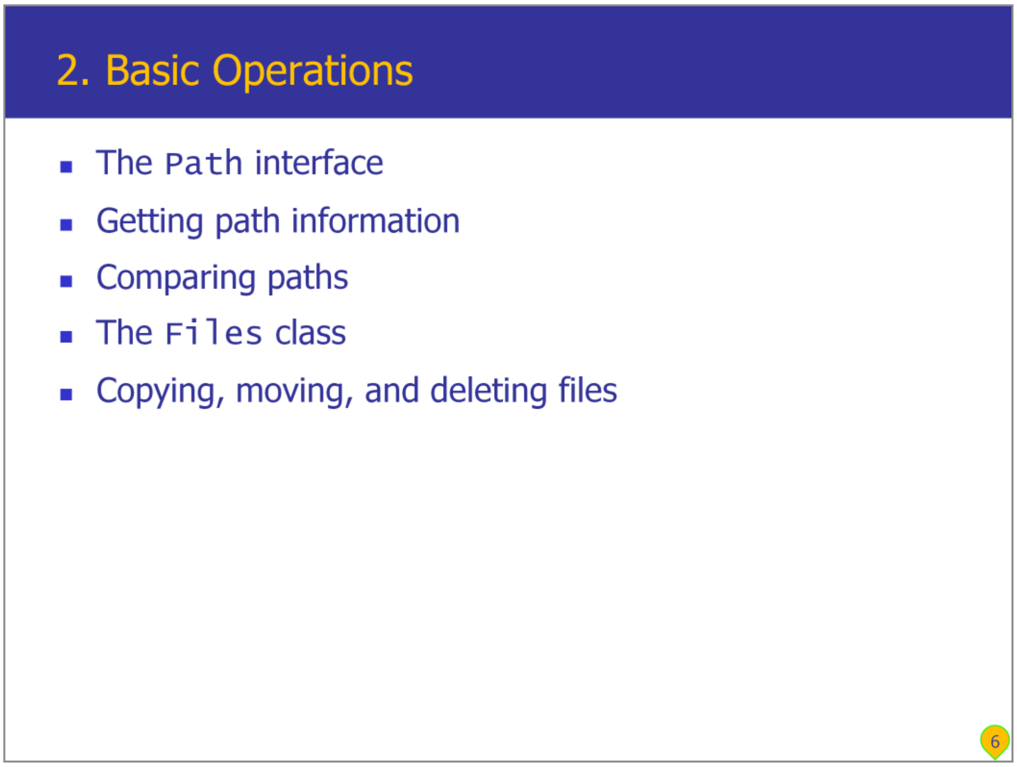
  - `java.NIO2.attribute.*AttributeView`

NIO2 introduces several new classes to simplify and extend file access in Java:

- `java.NIO2.file.Files` and `java.NIO2.file.FileVisitor`
  Allow you to walk through file systems, querying files or directories and executing user-implemented call-back methods for each one.

- `java.NIO2.file.Path` and `java.NIO2.file.WatchService`
  Allow you to register to "watch" a specific directory. An application watching directories receives notification if files in those directories are created, modified, or deleted.

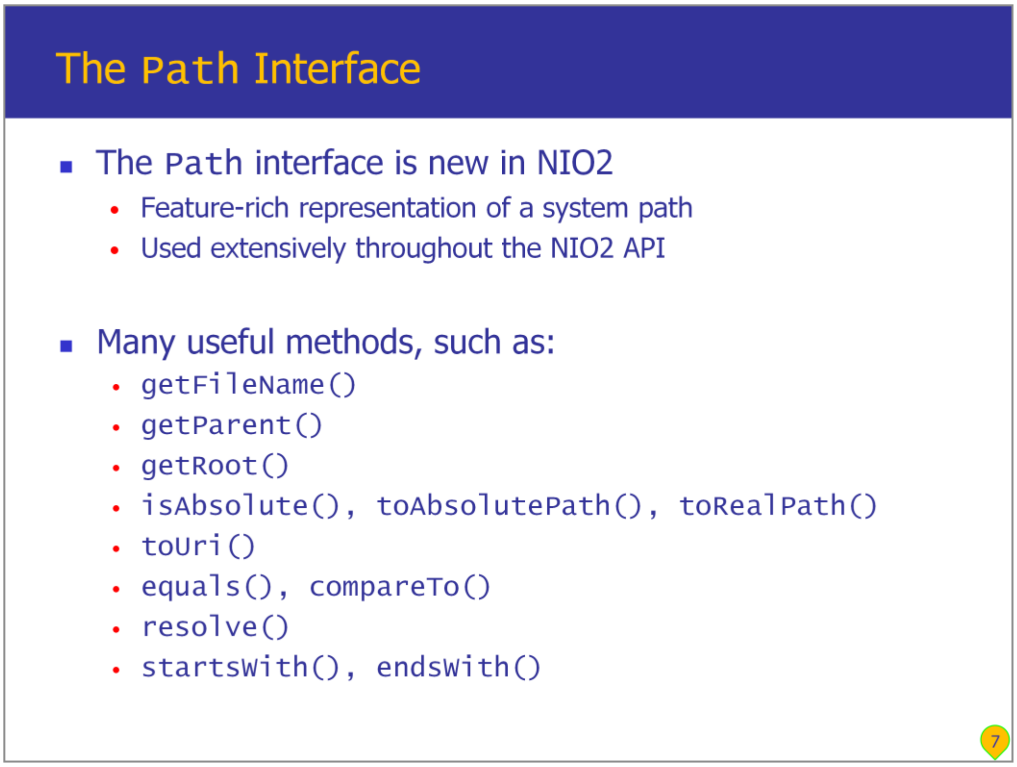
- `java.NIO2.attribute.*AttributeView`
  Allow you to view file and directory attributes previously hidden, e.g. file-owner and group permissions, ACLs, extended file attributes.



## 2. Basic Operations

- The `Path` interface
- Getting path information
- Comparing paths
- The `Files` class
- Copying, moving, and deleting files

In this section we'll see how to perform basic file / directory operations using the `Path` interface and the `Files` class in the `java.nio.file` package.



## The Path Interface

- The Path interface is new in NIO2
  - Feature-rich representation of a system path
  - Used extensively throughout the NIO2 API
- Many useful methods, such as:
  - `getFileName()`
  - `getParent()`
  - `getRoot()`
  - `isAbsolute(), toAbsolutePath(), toRealPath()`
  - `toUri()`
  - `equals(), compareTo()`
  - `resolve()`
  - `startsWith(), endsWith()`

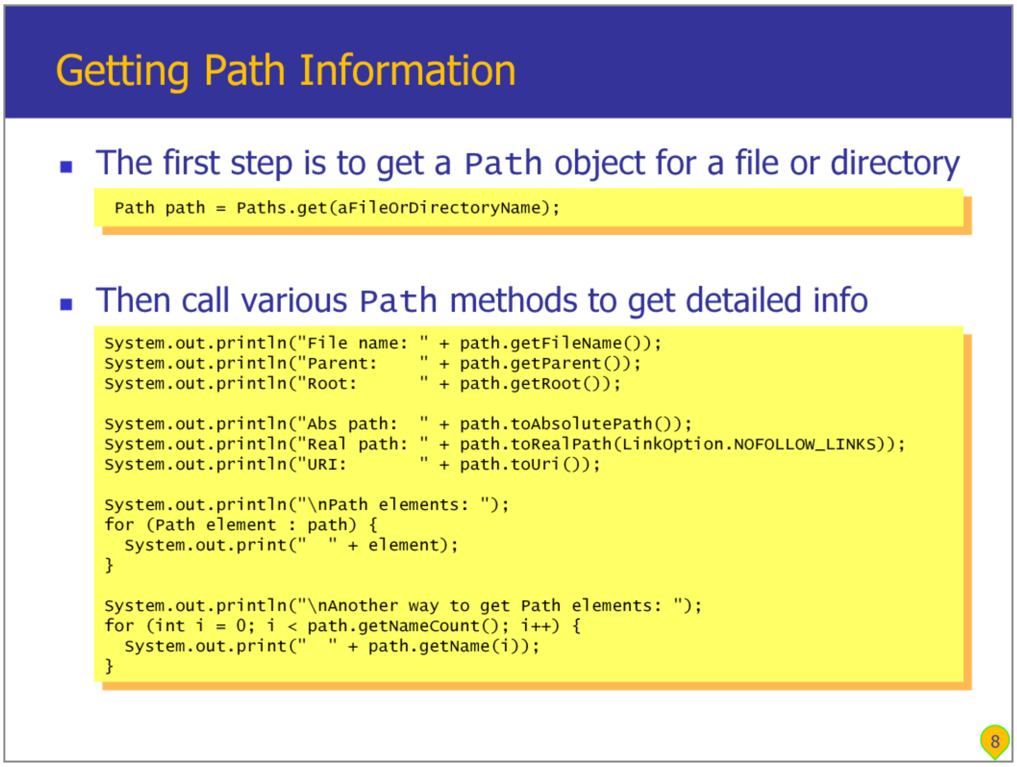
7

Java 7 introduces the `Path` interface, which is a new and improved representation of a path on the file system. The `Path` interface plays an important part in a lot of the new APIs in NIO2, so it's important you understand how it works.

A `Path` object represents a specific file or directory, and contains the names of all the directories and files that make the full path. The `Path` interface has methods that allow you to do all kinds of interesting things:

- Get the file name of a path
- Get the parent of a path
- Get the root of the path (e.g. C:\ in Windows)
- Determine if a path is absolute, convert it to an absolute path, and get the real path for symbolic links
- Get a URI that represents the path (such that you could open it in a browser)
- Determine if two paths are equal
- Resolve a sub-path beneath the current path
- Determine if a path starts or ends with another path
- Get the names of the path elements that make up the full path

We'll explore many of these capabilities in the following slides.



## Getting Path Information

- The first step is to get a Path object for a file or directory

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

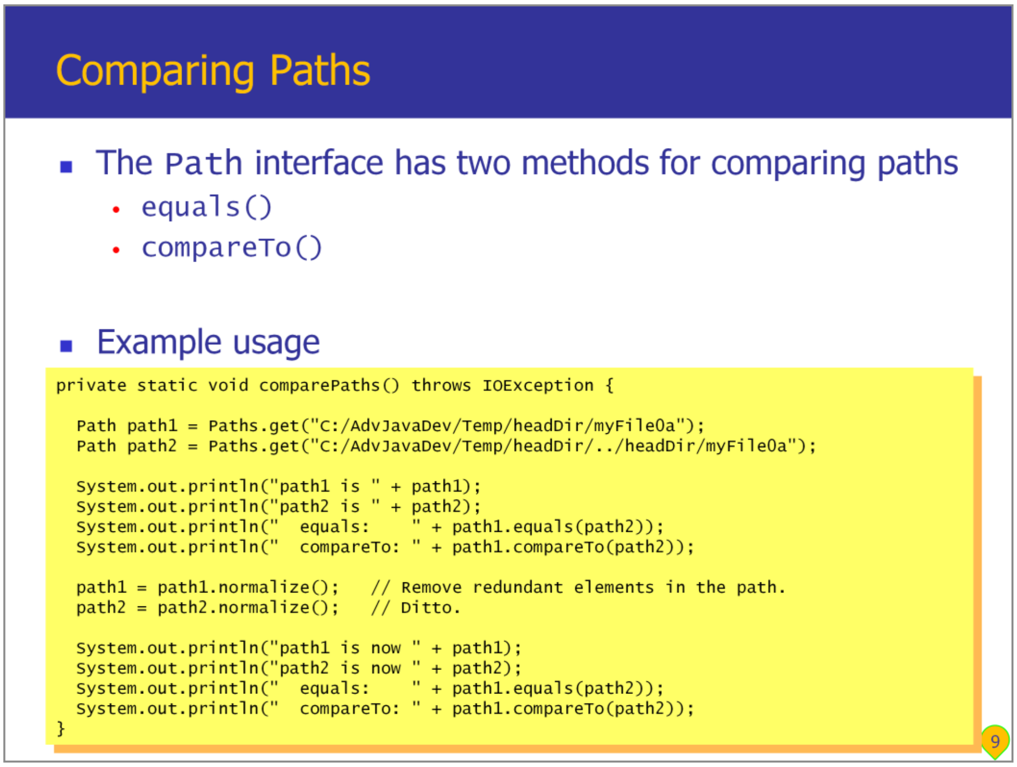
- Then call various Path methods to get detailed info

```
System.out.println("File name: " + path.getFileName());
System.out.println("Parent:    " + path.getParent());
System.out.println("Root:      " + path.getRoot());

System.out.println("Abs path:  " + path.toAbsolutePath());
System.out.println("Real path: " + path.toRealPath(LinkOption.NOFOLLOW_LINKS));
System.out.println("URI:       " + path.toUri());

System.out.println("\nPath elements: ");
for (Path element : path) {
  System.out.print("  " + element);
}

System.out.println("\nAnother way to get Path elements: ");
for (int i = 0; i < path.getNameCount(); i++) {
  System.out.print("  " + path.getName(i));
}
```

8

The code in the slide is available in `BasicOperationsExample.java`, in the `displayPathInfo()` method. In the demo, we create a `Path` that points to a file named "`C:/AdvJavaDev/Temp/headDir/myFile0a`". The `displayPathInfo()` method displays the following info for this `Path` object:

```
File name: myFile0a
Parent:    C:\AdvJavaDev\Temp\headDir
Root:      C:\
Abs path:  C:\AdvJavaDev\Temp\headDir\myFile0a
Real path: C:\AdvJavaDev\Temp\headDir\myFile0a
URI:       file:///C:/AdvJavaDev/Temp/headDir/myFile0a

Path elements:
    AdvJavaDev  Temp  headDir  myFile0a
Another way to get Path elements:
    AdvJavaDev  Temp  headDir  myFile0a
```



## Comparing Paths

- The `Path` interface has two methods for comparing paths
  - `equals()`
  - `compareTo()`
- Example usage

```
private static void comparePaths() throws IOException {

  Path path1 = Paths.get("C:/AdvJavaDev/Temp/headDir/myFile0a");
  Path path2 = Paths.get("C:/AdvJavaDev/Temp/headDir/../headDir/myFile0a");

  System.out.println("path1 is " + path1);
  System.out.println("path2 is " + path2);
  System.out.println("  equals:    " + path1.equals(path2));
  System.out.println("  compareTo: " + path1.compareTo(path2));

  path1 = path1.normalize();   // Remove redundant elements in the path.
  path2 = path2.normalize();   // Ditto.

  System.out.println("path1 is now " + path1);
  System.out.println("path2 is now " + path2);
  System.out.println("  equals:    " + path1.equals(path2));
  System.out.println("  compareTo: " + path1.compareTo(path2));
}
```

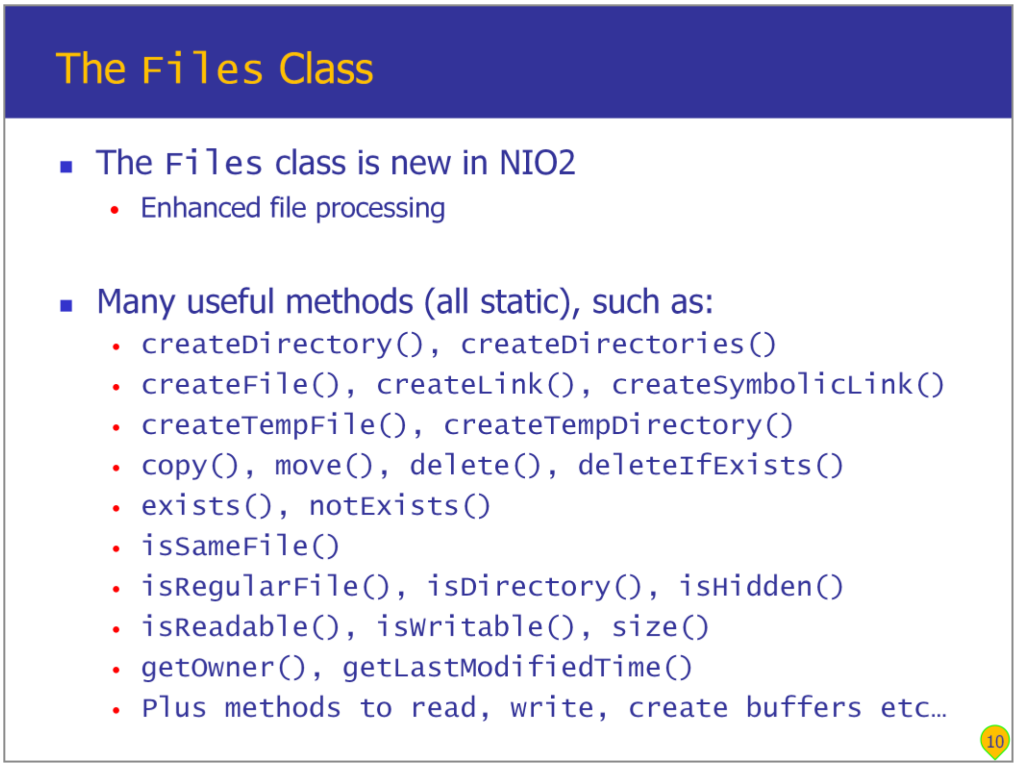
9

The `Path` interface has two methods that allow you to compare paths:

- `equals()` returns `true` if two `Path` objects are the same literal path.
- `compareTo()` returns a negative value, positive value, or 0 as per normal.

In the demo we create two `Path` objects that have different literal paths, but they refer to the same physical file on the file system. `comparePaths()` displays the following output for these paths:

```
path1 is C:\AdvJavaDev\Temp\headDir\myFile0a
path2 is C:\AdvJavaDev\Temp\headDir\..\headDir\myFile0a
  equals:    false
  compareTo: 31
path1 is now C:\AdvJavaDev\Temp\headDir\myFile0a
path2 is now C:\AdvJavaDev\Temp\headDir\myFile0a
  equals:    true
  compareTo: 0
```



## The Files Class

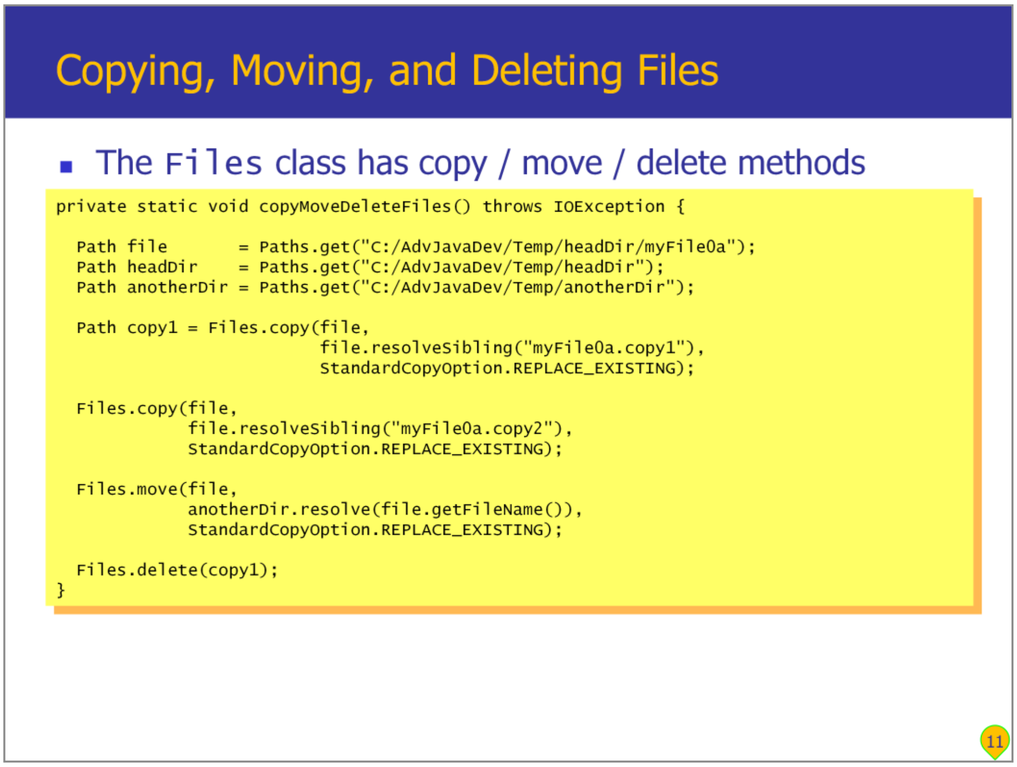
- The `Files` class is new in NIO2
  - Enhanced file processing
- Many useful methods (all static), such as:
  - `createDirectory(), createDirectories()`
  - `createFile(), createLink(), createSymbolicLink()`
  - `createTempFile(), createTempDirectory()`
  - `copy(), move(), delete(), deleteIfExists()`
  - `exists(), notExists()`
  - `isSameFile()`
  - `isRegularFile(), isDirectory(), isHidden()`
  - `isReadable(), isWritable(), size()`
  - `getOwner(), getLastModifiedTime()`
  - `Plus methods to read, write, create buffers etc…`

10

We're now going to turn our attention to the `Files` class. This class defines a large number of methods, all static, for doing just about anything you could care to mention to files and directories!

We've listed some of the common methods in the slide above. For full details, see the JavaDoc documentation here:

- http://docs.oracle.com/javase/7/docs/api/index.html?java/nio/file/Files.html



## Copying, Moving, and Deleting Files

- The `Files` class has copy / move / delete methods

```
private static void copyMoveDeleteFiles() throws IOException {

  Path file       = Paths.get("C:/AdvJavaDev/Temp/headDir/myFile0a");
  Path headDir    = Paths.get("C:/AdvJavaDev/Temp/headDir");
  Path anotherDir = Paths.get("C:/AdvJavaDev/Temp/anotherDir");

  Path copy1 = Files.copy(file,
                          file.resolveSibling("myFile0a.copy1"),
                          StandardCopyOption.REPLACE_EXISTING);

  Files.copy(file,
             file.resolveSibling("myFile0a.copy2"),
             StandardCopyOption.REPLACE_EXISTING);

  Files.move(file,
             anotherDir.resolve(file.getFileName()),
             StandardCopyOption.REPLACE_EXISTING);

  Files.delete(copy1);
}
```

11

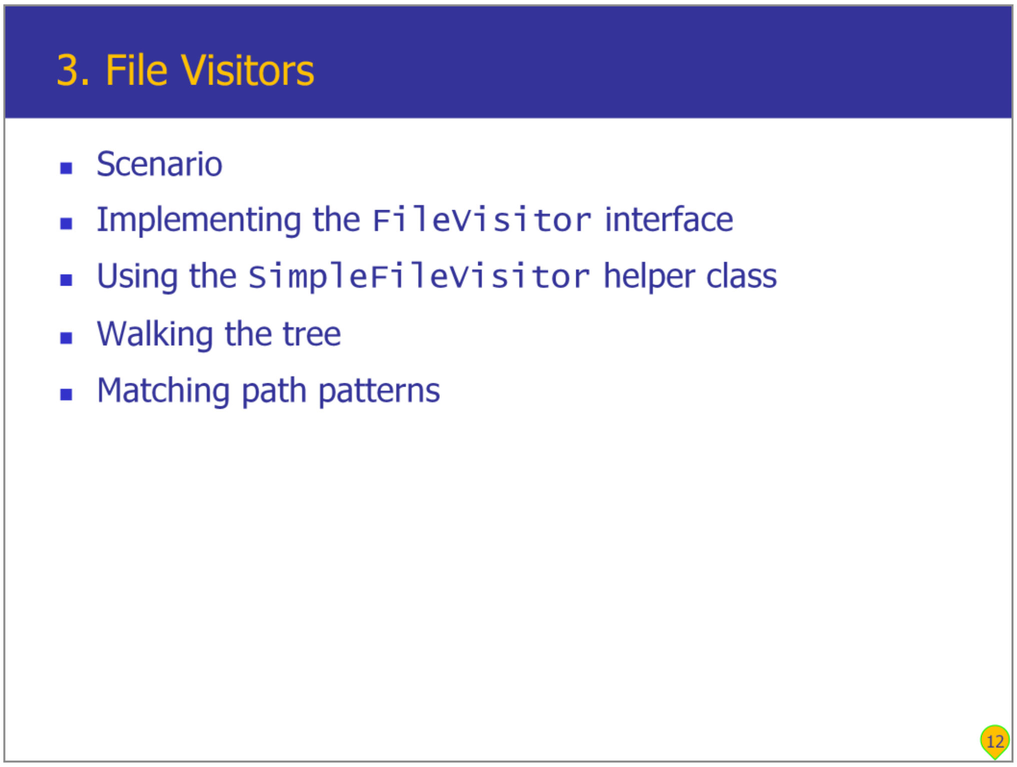
This example shows how to use the following static methods in the Files class:

- `copy(Path source, Path target, CopyOption… options)`
- `move(Path source, Path target, CopyOption… options)`
- `delete(Path path)`

For `copy()` and `move()`, you can pass a sequence of `StandardCopyOption` values such as `REPLACE_EXISTING`, `ATOMIC_MOVE`, and `COPY_ATTRIBUTES`.

Note the following points in our example:

- The two calls to `copy()` create copies of the original file in the same folder. Notice the use of `resolveSibling()` - this method is defined in the `Path` interface, and it gives you a `Path` that represents a filename in the same directory.
- The call to `move()` moves the specified file to another directory. Notice the use of `resolve()` - this method is defined in the `Path` interface as well, and it gives you a `Path` that represents a directory and filename combined.
- The call to `delete()` deletes the file identified by a `Path` object.

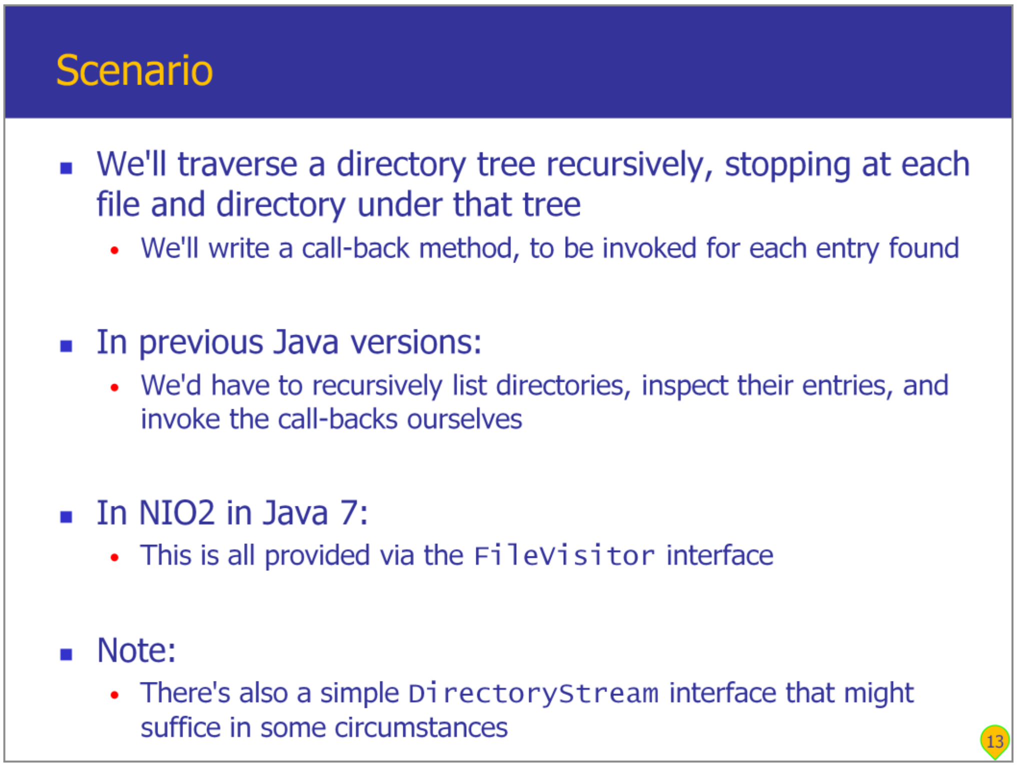


## 3. File Visitors

- Scenario
- Implementing the `FileVisitor` interface
- Using the `SimpleFileVisitor` helper class
- Walking the tree
- Matching path patterns

12

In this section we'll see how to use the `FileVisitor` interface to recursively traverse a folder and process each file found.



## Scenario

- We'll traverse a directory tree recursively, stopping at each file and directory under that tree
  - We'll write a call-back method, to be invoked for each entry found
- In previous Java versions:
  - We'd have to recursively list directories, inspect their entries, and invoke the call-backs ourselves
- In NIO2 in Java 7:
  - This is all provided via the `FileVisitor` interface
- Note:
  - There's also a simple `DirectoryStream` interface that might suffice in some circumstances

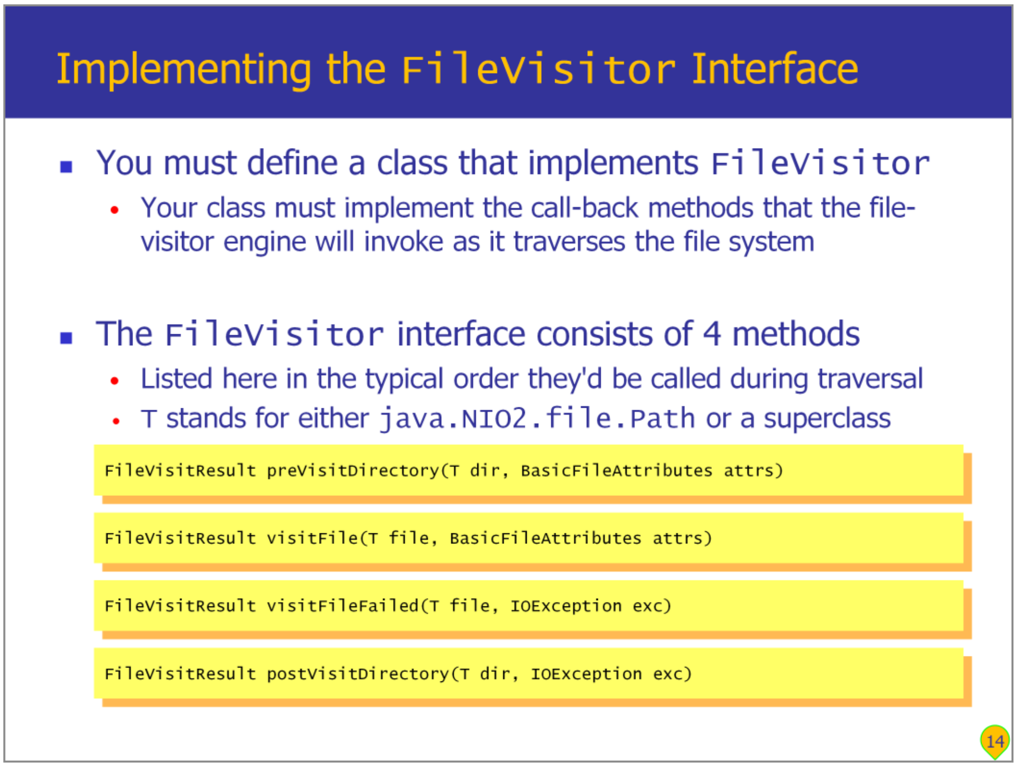
13

The code for this example is in the `demo.NIO2.file_apis` package - see the `FileVisitorExample.java` class.

Note:

NIO2 also defines a simple interface named `DirectoryStream` that is handy if you just want to scan a directory for specific file patterns. For an good example of how this works, see the JavaDoc documentation here:

- docs.oracle.com/javase/7/docs/api/index.html?java/nio/file/DirectoryStream.html



## Implementing the FileVisitor Interface

- You must define a class that implements `FileVisitor`
  - Your class must implement the call-back methods that the file-visitor engine will invoke as it traverses the file system
- The `FileVisitor` interface consists of 4 methods
  - Listed here in the typical order they'd be called during traversal
  - `T` stands for either `java.NIO2.file.Path` or a superclass

```
FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)
```

```
FileVisitResult visitFile(T file, BasicFileAttributes attrs)
```

```
FileVisitResult visitFileFailed(T file, IOException exc)
```

```
FileVisitResult postVisitDirectory(T dir, IOException exc)
```

14

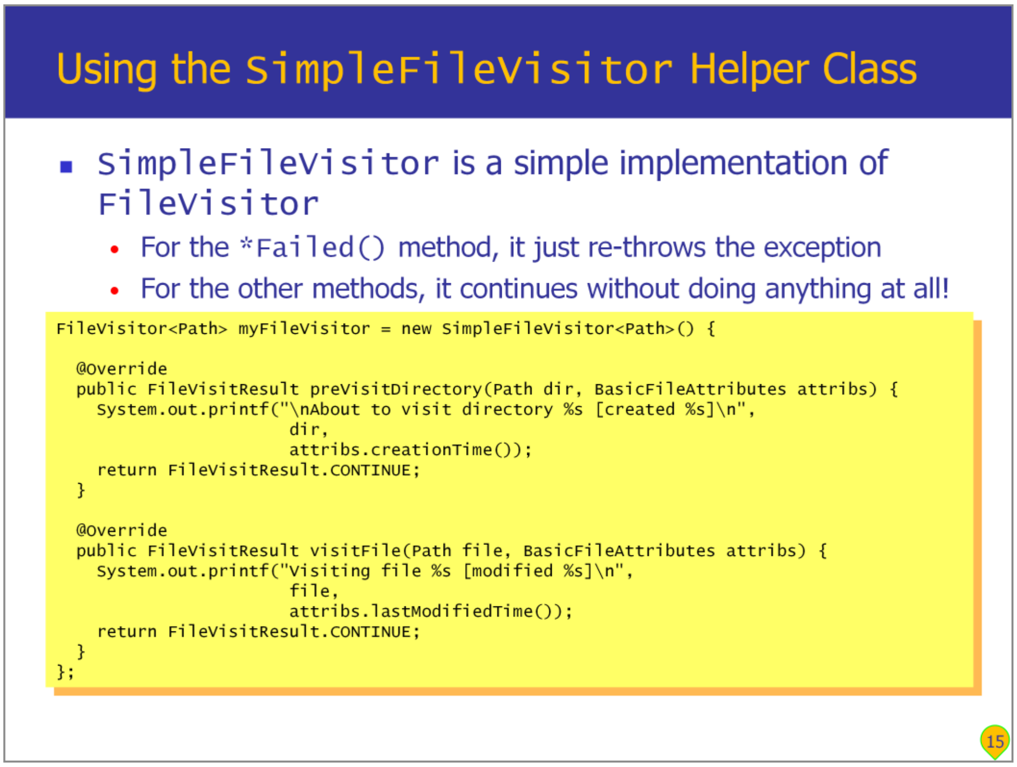
Here's a bit more detail on each of the methods in the `FileVisitor` interface:

`FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)`
Invoked before entries in the directory are visited. Returns a `FileVisitResult` enum, which tells file visitor what to do next.

`FileVisitResult visitFile(T file, BasicFileAttributes attrs)`
Invoked when a file in the current directory is being visited. The attributes for this file are passed into the 2nd parameter.

`FileVisitResult visitFileFailed(T file, IOException exc)`
Invoked if the visit to a file has failed. The 2nd parameter is the exception that caused the visit to fail.

`FileVisitResult postVisitDirectory(T dir, IOException exc)`
Invoked after the visit to a directory and all sub-directories completed OK. The exception parameter is `null` if all went OK.



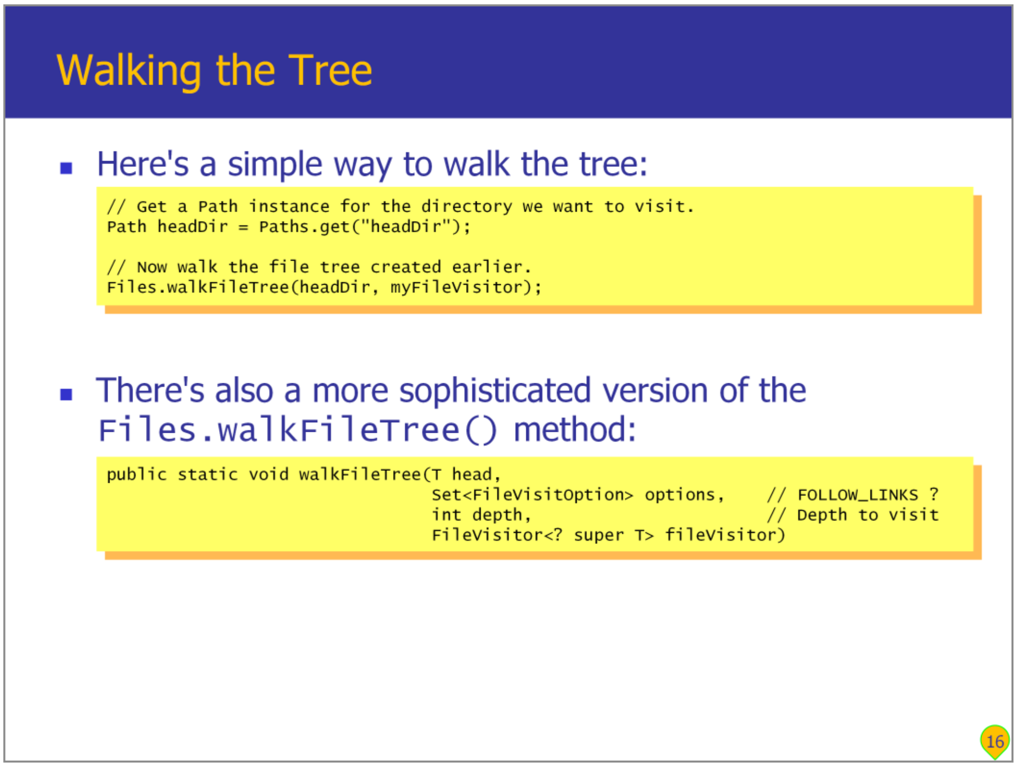
## Using the SimpleFileVisitor Helper Class

- SimpleFileVisitor is a simple implementation of FileVisitor
  - For the *Failed() method, it just re-throws the exception
  - For the other methods, it continues without doing anything at all!

```
FileVisitor<Path> myFileVisitor = new SimpleFileVisitor<Path>() {

  @Override
  public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attribs) {
    System.out.printf("\nAbout to visit directory %s [created %s]\n",
                       dir,
                       attribs.creationTime());
    return FileVisitResult.CONTINUE;
  }

  @Override
  public FileVisitResult visitFile(Path file, BasicFileAttributes attribs) {
    System.out.printf("Visiting file %s [modified %s]\n",
                       file,
                       attribs.lastModifiedTime());
    return FileVisitResult.CONTINUE;
  }
};
```

15

This slide shows a simple example of a class that extends SimpleFileVisitor . The class displays a message just before it visits each directory, and another message each time it visits a file.



## Walking the Tree

- Here's a simple way to walk the tree:

```
// Get a Path instance for the directory we want to visit.
Path headDir = Paths.get("headDir");

// Now walk the file tree created earlier.
Files.walkFileTree(headDir, myFileVisitor);
```

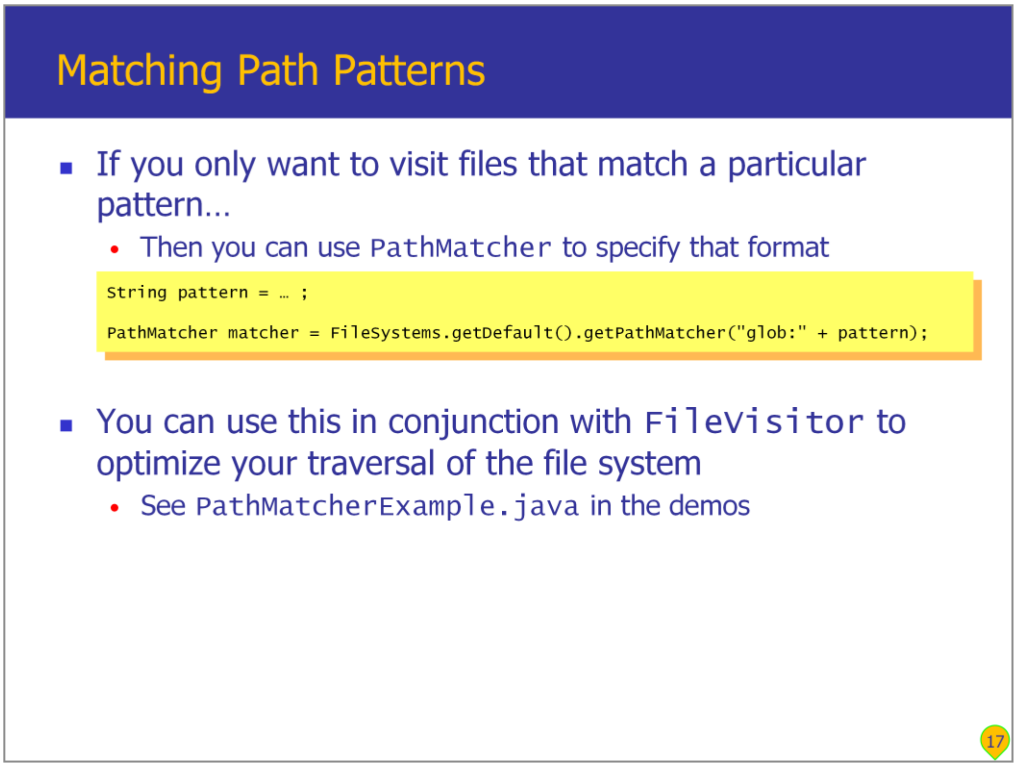
- There's also a more sophisticated version of the `Files.walkFileTree()` method:

```
public static void walkFileTree(T head,
                                Set<FileVisitOption> options,    // FOLLOW_LINKS ?
                                int depth,                       // Depth to visit
                                FileVisitor<? super T> fileVisitor)
```

16

The demo app creates some directories underneath C:\AdvJavaDev\Temp, and populates them with a few simple files. If you run the demo app, it displays the following output (we've omitted the creation and modification dates here, for brevity):

```
About to visit directory C:\AdvJavaDev\Temp\headDir [created…]
Visiting file C:\AdvJavaDev\Temp\headDir\myFile0a [modified…]
Visiting file C:\AdvJavaDev\Temp\headDir\myFile0b [modified…]
Visiting file C:\AdvJavaDev\Temp\headDir\myFile0c [modified…]

About to visit directory C:\AdvJavaDev\Temp\headDir\mySubDirectory1 [created...]
Visiting file C:\AdvJavaDev\Temp\headDir\mySubDirectory1\myFile1a [modified…]
Visiting file C:\AdvJavaDev\Temp\headDir\mySubDirectory1\myFile1b [modified…]

About to visit directory C:\AdvJavaDev\Temp\headDir\mySubDirectory2 [created…]
Visiting file C:\AdvJavaDev\Temp\headDir\mySubDirectory2\myFile2a [modified…]
Visiting file C:\AdvJavaDev\Temp\headDir\mySubDirectory2\myFile2b [modified…]
```



## Matching Path Patterns

- If you only want to visit files that match a particular pattern…
  - Then you can use `PathMatcher` to specify that format

```
String pattern = … ;
PathMatcher matcher = FileSystems.getDefault().getPathMatcher("glob:" + pattern);
```

- You can use this in conjunction with `FileVisitor` to optimize your traversal of the file system
  - See `PathMatcherExample.java` in the demos

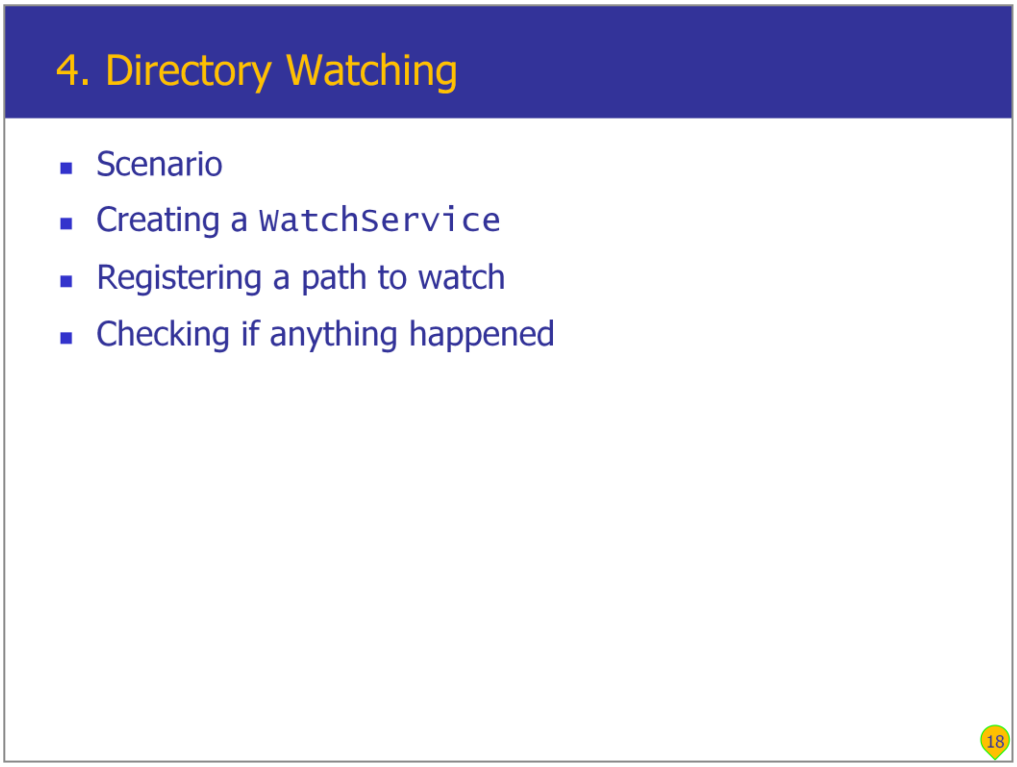
17

To conclude our discussion on file visitors, we'll consider the situation where you only want to visit paths that have a particular path format (e.g. *.java). To achieve this effect, you create a `PathMatcher` on the file system, and specify the pattern in a format such as "`glob:*.java`".

The `PathMatcherExample.java` demo shows how to integrate this capability into a file visitor.

You should also note that the following syntax is allowed in the pattern:

| | |
|---|---|
| `*` | Matches any string of any length (even zero length) |
| `**` | Similar to * but it crosses directory boundaries |
| `?` | Matches any single character |
| `[abc]` | Matches either a , b, or c |
| `[1-5]` | Matches any character in the range 1 to 5 |
| `[A-Z]` | Matches any uppercase letter |
| `{abc, def}` | Matches either abc or def |

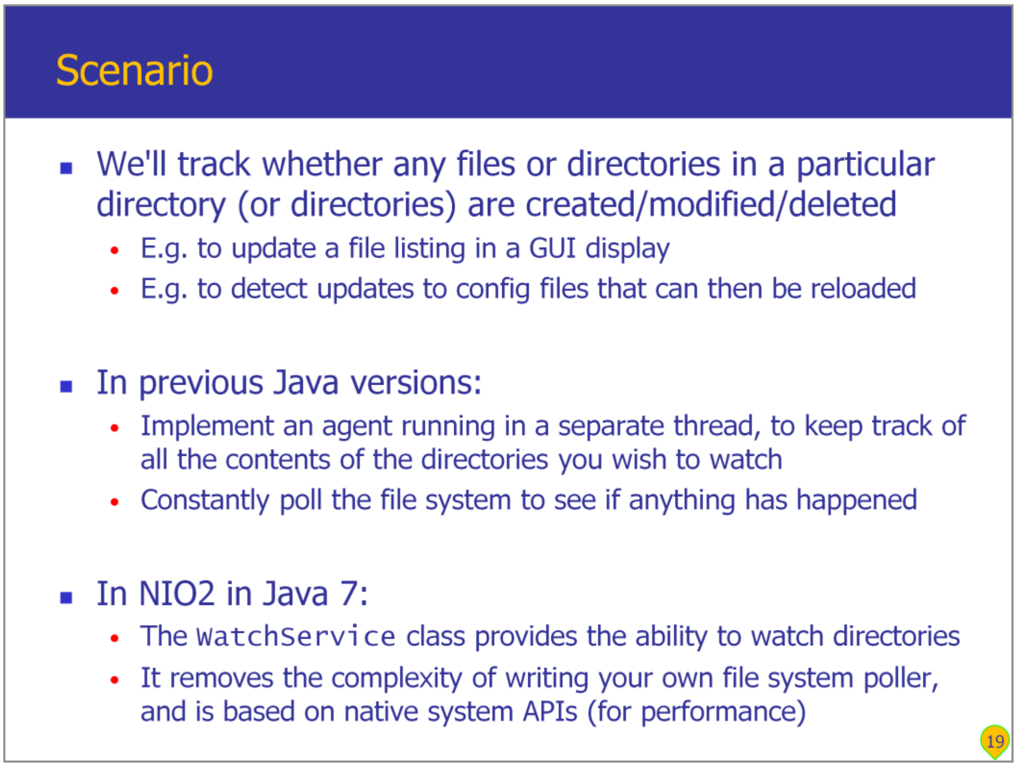


## 4. Directory Watching

- Scenario
- Creating a `WatchService`
- Registering a path to watch
- Checking if anything happened

18

In this section we'll see how to use the `WatchService` class to recursively traverse a folder and process each file found.

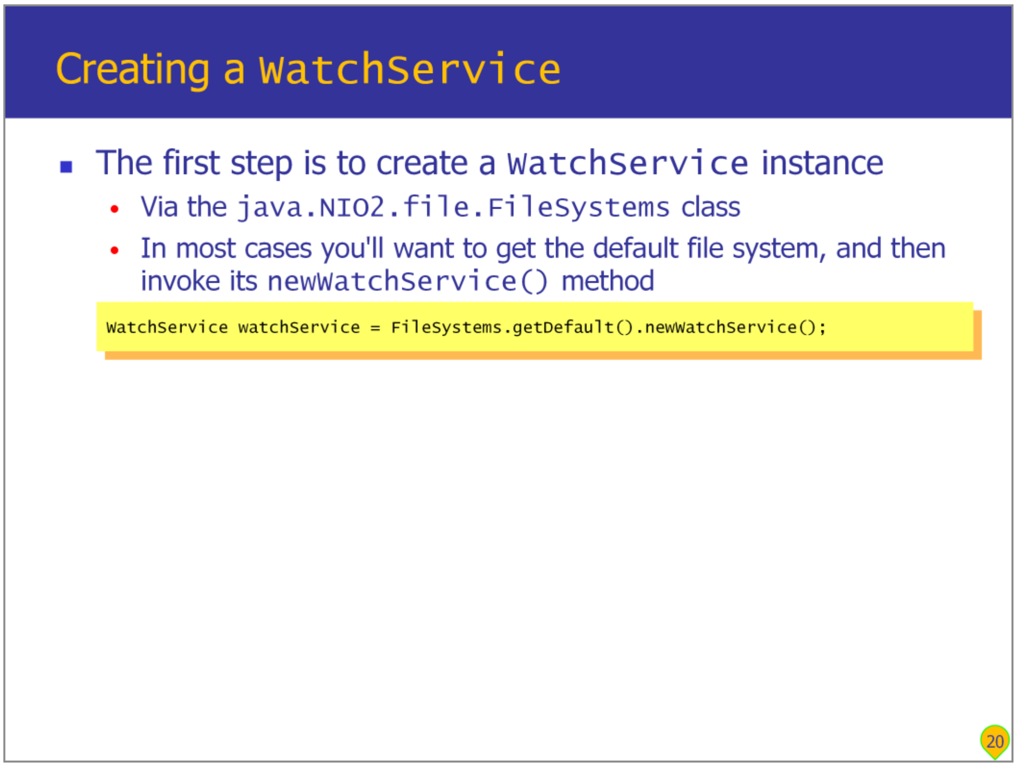


## Scenario

- We'll track whether any files or directories in a particular directory (or directories) are created/modified/deleted
  - E.g. to update a file listing in a GUI display
  - E.g. to detect updates to config files that can then be reloaded
- In previous Java versions:
  - Implement an agent running in a separate thread, to keep track of all the contents of the directories you wish to watch
  - Constantly poll the file system to see if anything has happened
- In NIO2 in Java 7:
  - The `WatchService` class provides the ability to watch directories
  - It removes the complexity of writing your own file system poller, and is based on native system APIs (for performance)

19

The code for this example is in the `demo.NIO2.file_apis` package - see the `DirectoryWatchingExample.java` class.



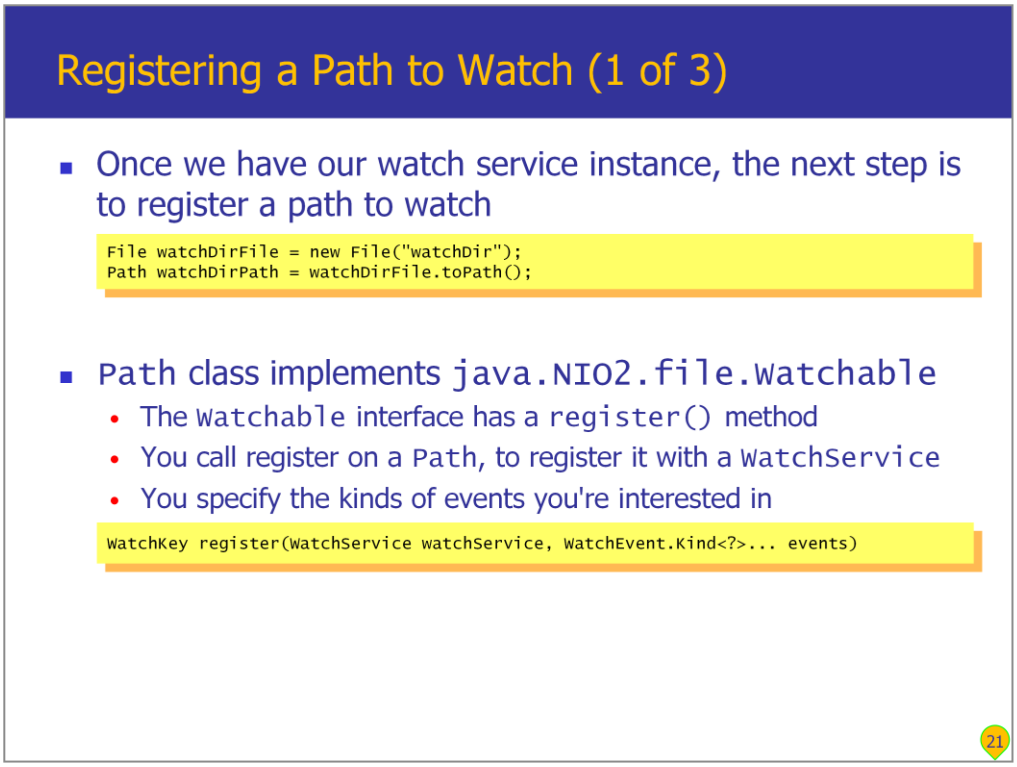
## Creating a `WatchService`

- The first step is to create a `WatchService` instance
  - Via the `java.NIO2.file.FileSystems` class
  - In most cases you'll want to get the default file system, and then invoke its `newWatchService()` method

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

20

The vital ingredient in all this is clearly the WatchService object. It's going to be responsible for watching the specified file system for file/directory events.



## Registering a Path to Watch (1 of 3)

- Once we have our watch service instance, the next step is to register a path to watch

```
File watchDirFile = new File("watchDir");
Path watchDirPath = watchDirFile.toPath();
```

- `Path` class implements `java.NIO2.file.Watchable`
  - The `Watchable` interface has a `register()` method
  - You call register on a `Path`, to register it with a `WatchService`
  - You specify the kinds of events you're interested in

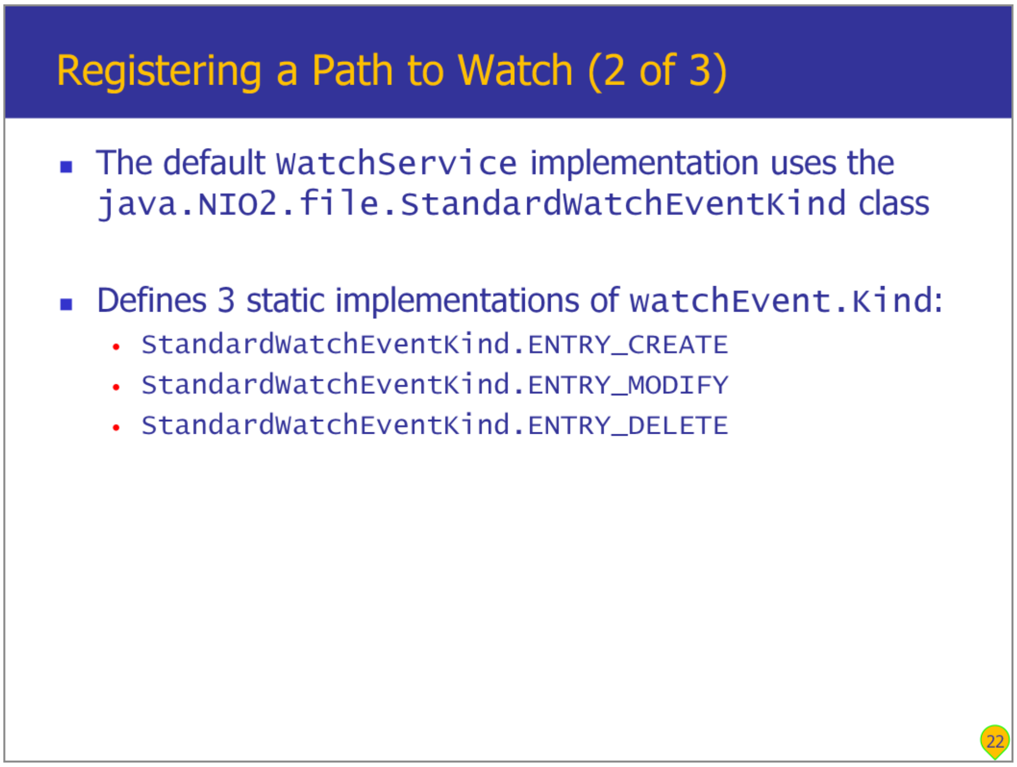
```
WatchKey register(WatchService watchService, WatchEvent.Kind<?>... events)
```

21

Once you've created a `WatchService` object, you're ready to start listening for file system events. To do this, you obviously have to specify which directory to watch. To do this, you use a `Path` object...

`Path` implements the `Watchable` interface, which means you can call `register()` on a `Path` object to register for specific kinds of event. The `register()` method takes the following parameters:

- The `WatchService` object that you want to be notified about events.
- A variadic sequence of `WatchEvent.Kind` values, specifying the kinds of event you want to listen for. See the next slide for more details.



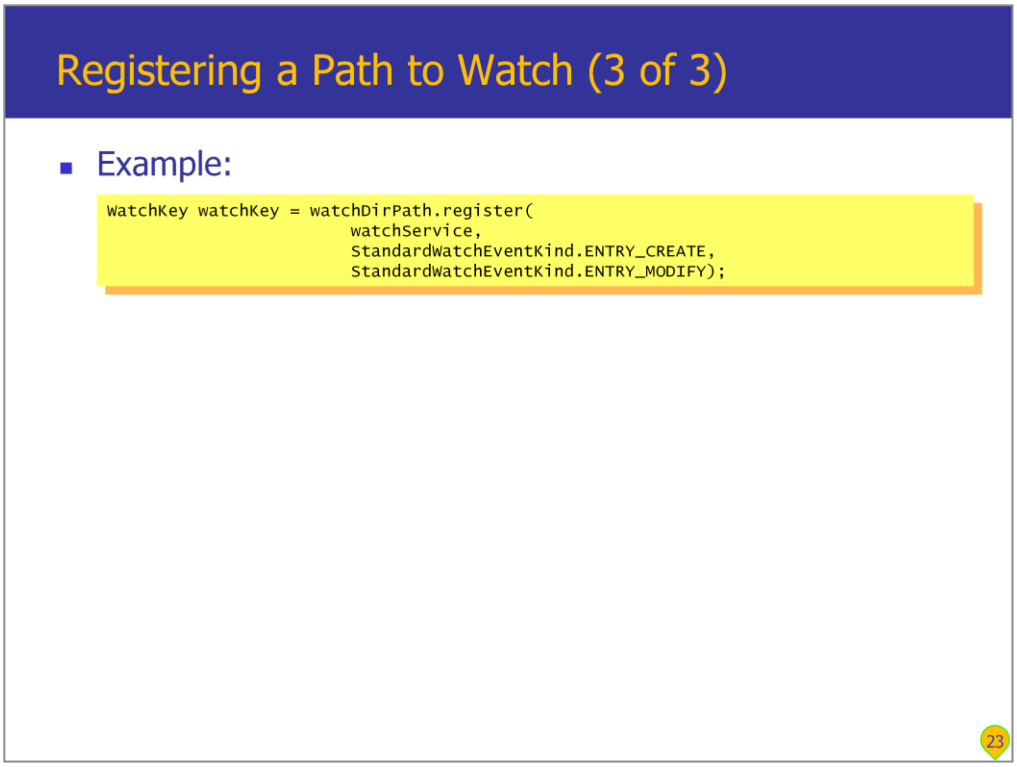
## Registering a Path to Watch (2 of 3)

- The default `WatchService` implementation uses the `java.NIO2.file.StandardWatchEventKind` class
- Defines 3 static implementations of `watchEvent.Kind`:
  - `StandardWatchEventKind.ENTRY_CREATE`
  - `StandardWatchEventKind.ENTRY_MODIFY`
  - `StandardWatchEventKind.ENTRY_DELETE`

22

Here's a bit more info about the 3 static implementations of `watchEvent.Kind`:

- `StandardWatchEventKind.ENTRY_CREATE`
  A file or directory has been created (or renamed or moved into dir)
- `StandardWatchEventKind.ENTRY_MODIFY`
  A file or directory has been modified (or file attributes modified?)
- `StandardWatchEventKind.ENTRY_DELETE`
  A file or directory has been deleted (or renamed or moved out of the directory)



## Registering a Path to Watch (3 of 3)

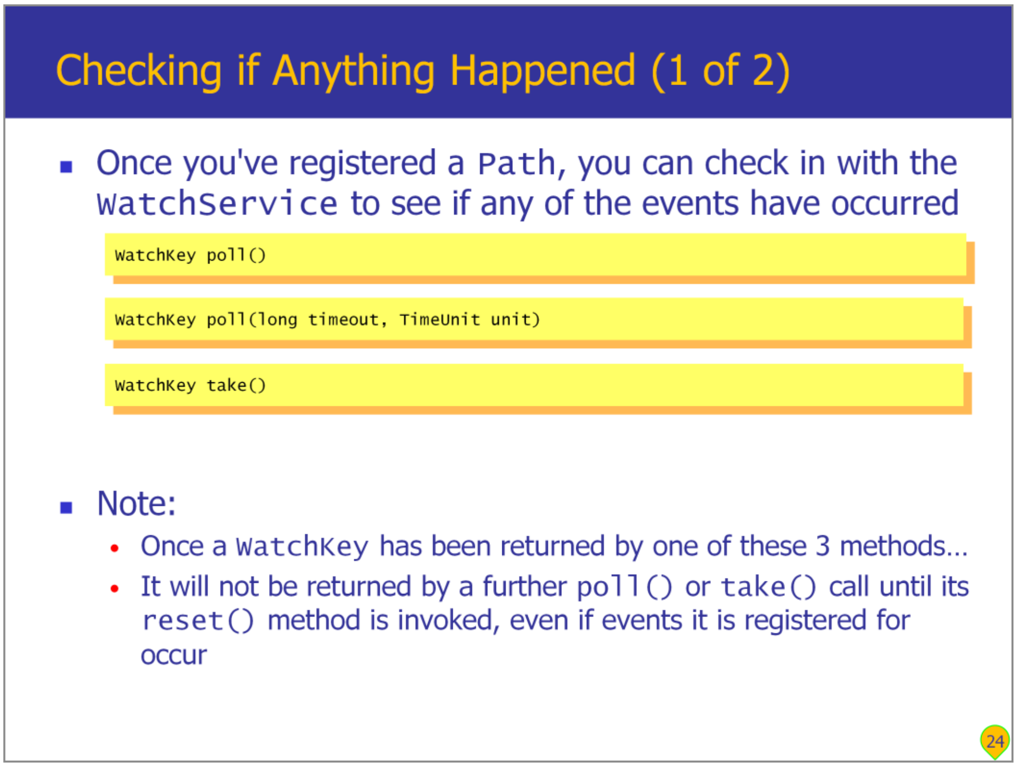
- Example:

```
WatchKey watchKey = watchDirPath.register(
                        watchService,
                        StandardWatchEventKind.ENTRY_CREATE,
                        StandardWatchEventKind.ENTRY_MODIFY);
```

23

This example sets up our `Path` object so that it will notify our `WatchService` object whenever files are created or modified in the directory.

The `WatchService` will work away silently in the background, watching that directory intently. The same `WatchService` instance can watch multiple directories by using the same `Path` creation and `register()` calls we've just seen.



## Checking if Anything Happened (1 of 2)

- Once you've registered a `Path`, you can check in with the `WatchService` to see if any of the events have occurred

```
WatchKey poll()
```

```
WatchKey poll(long timeout, TimeUnit unit)
```

```
WatchKey take()
```

- Note:
  - Once a `WatchKey` has been returned by one of these 3 methods…
  - It will not be returned by a further `poll()` or `take()` call until its `reset()` method is invoked, even if events it is registered for occur

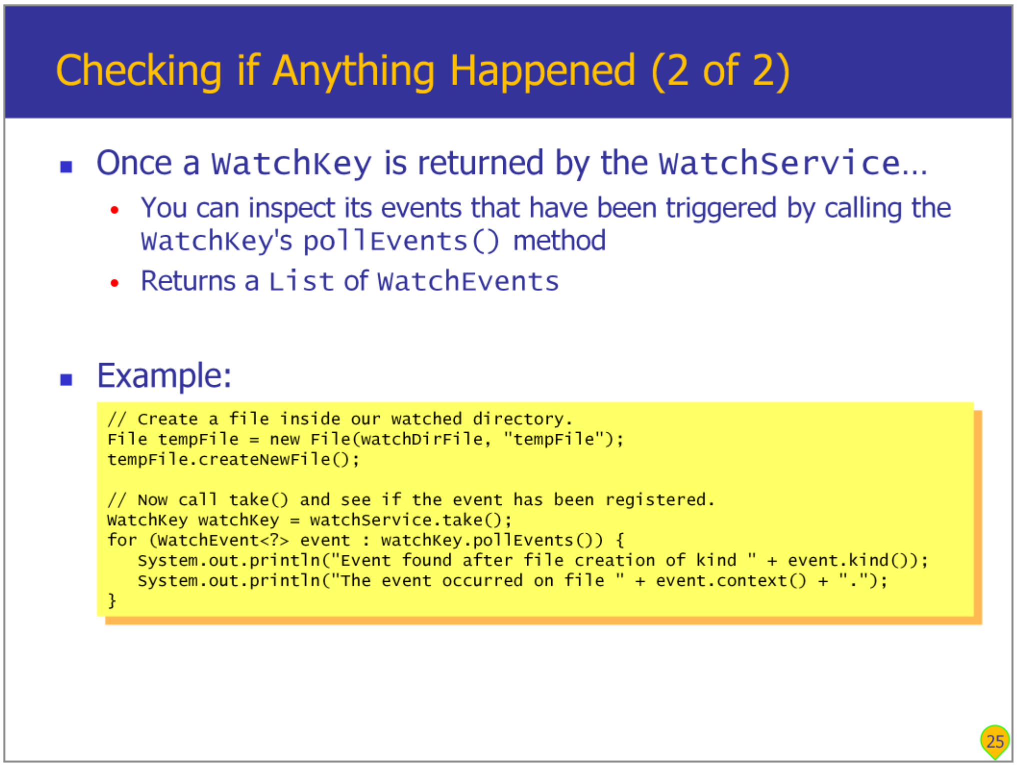
24

Here's a bit more info about the three methods shown on the slide:

- `WatchKey poll()`
  Returns the next `WatchKey` that has had some of its events occur, or `null` if no registered events occurred.

- `WatchKey poll(long timeout, TimeUnit unit)`
  If an event occurs during the specified time period, this method returns the relevant `WatchKey`. Otherwise it returns `null` at the end of the timeout period.

- `WatchKey take()`
  Similar to the preceding methods, except it waits indefinitely until a `WatchKey` is available to return.



## Checking if Anything Happened (2 of 2)

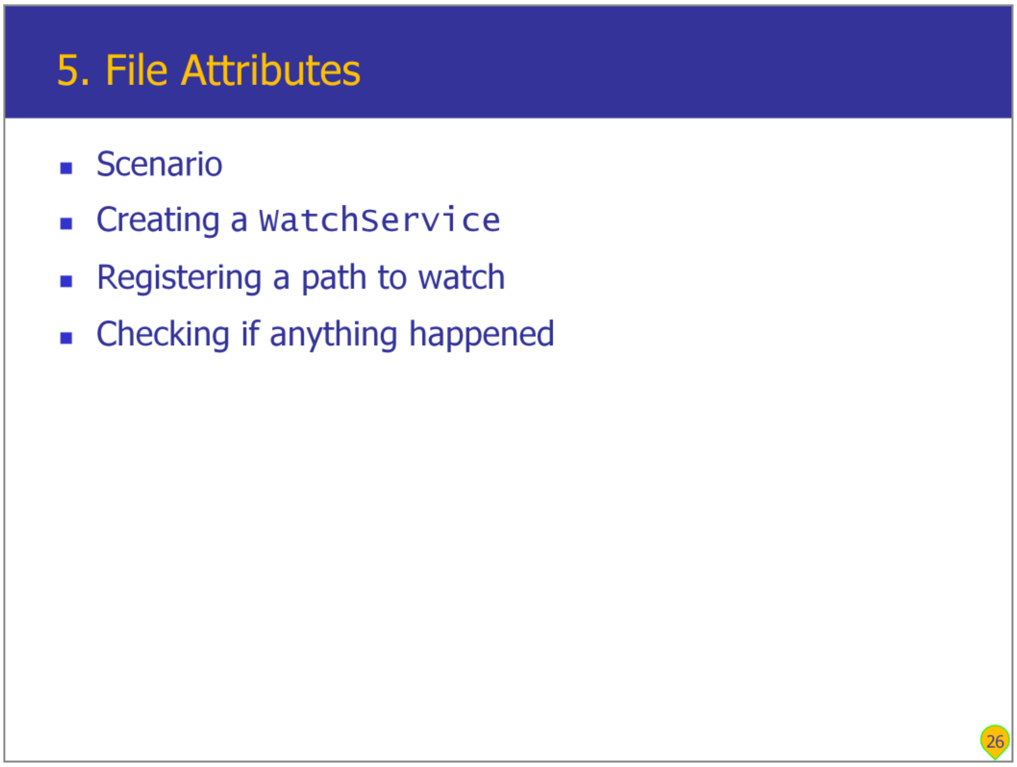
- Once a `WatchKey` is returned by the `WatchService`…
  - You can inspect its events that have been triggered by calling the `WatchKey`'s `pollEvents()` method
  - Returns a `List` of `WatchEvents`
- Example:

```
// Create a file inside our watched directory.
File tempFile = new File(watchDirFile, "tempFile");
tempFile.createNewFile();

// Now call take() and see if the event has been registered.
WatchKey watchKey = watchService.take();
for (WatchEvent<?> event : watchKey.pollEvents()) {
   System.out.println("Event found after file creation of kind " + event.kind());
   System.out.println("The event occurred on file " + event.context() + ".");
}
```

25

This slide shows simple usage of the `take()` method.

For full implementation details, see the code in the demo project.

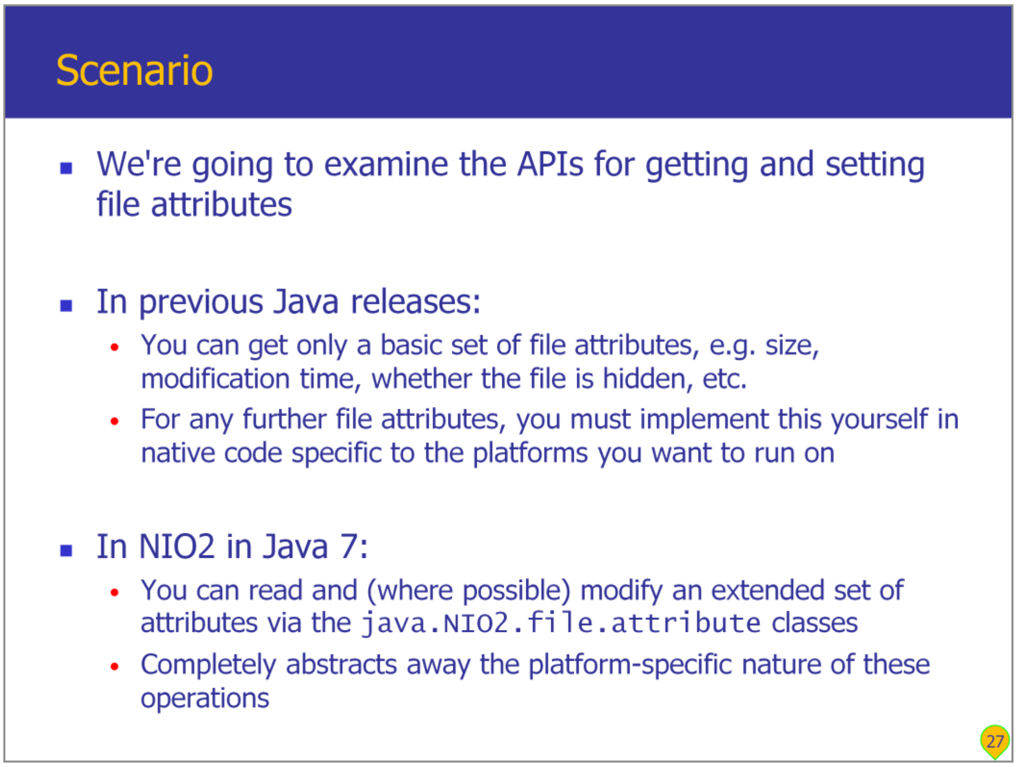


## 5. File Attributes

- Scenario
- Creating a `WatchService`
- Registering a path to watch
- Checking if anything happened

26

In this section we'll see how to use classes in the `java.NIO2.file.attribute` package to get and set attributes on file system entities.

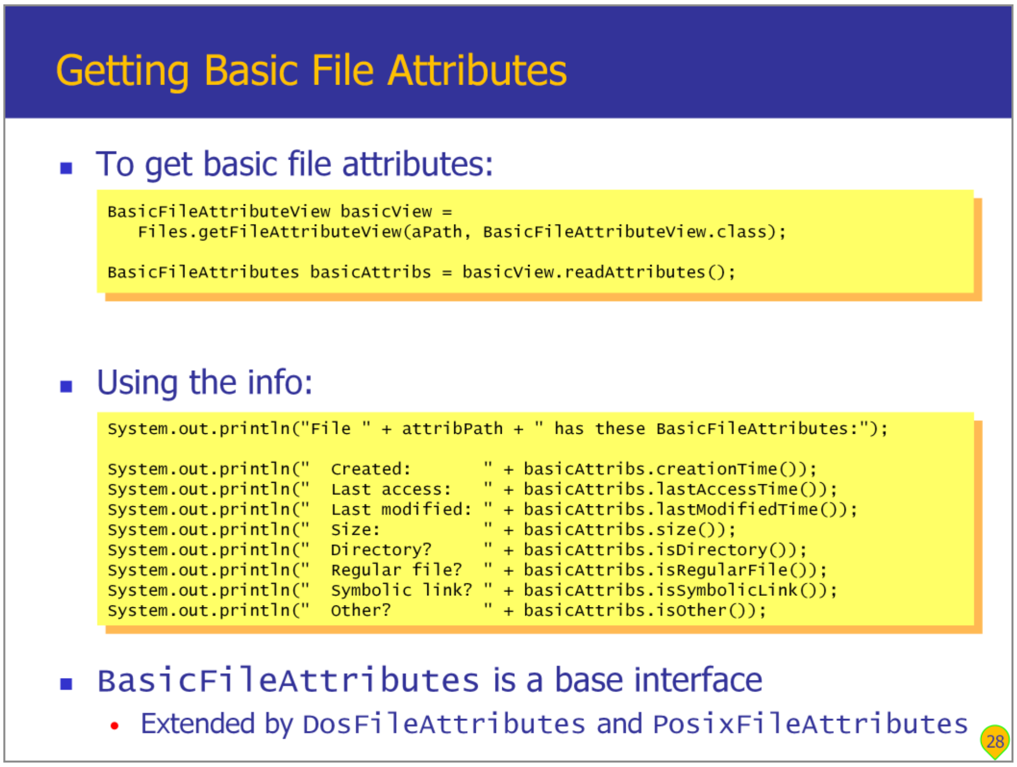


## Scenario

- We're going to examine the APIs for getting and setting file attributes
- In previous Java releases:
  - You can get only a basic set of file attributes, e.g. size, modification time, whether the file is hidden, etc.
  - For any further file attributes, you must implement this yourself in native code specific to the platforms you want to run on
- In NIO2 in Java 7:
  - You can read and (where possible) modify an extended set of attributes via the `java.NIO2.file.attribute` classes
  - Completely abstracts away the platform-specific nature of these operations

27

The code for this example is in the `demo.NIO2.file_apis` package - see the `FileAttributesExample.java` class.



## Getting Basic File Attributes

- To get basic file attributes:

```
BasicFileAttributeView basicView =
   Files.getFileAttributeView(aPath, BasicFileAttributeView.class);

BasicFileAttributes basicAttribs = basicView.readAttributes();
```

- Using the info:

```
System.out.println("File " + attribPath + " has these BasicFileAttributes:");

System.out.println("  Created:       " + basicAttribs.creationTime());
System.out.println("  Last access:   " + basicAttribs.lastAccessTime());
System.out.println("  Last modified: " + basicAttribs.lastModifiedTime());
System.out.println("  Size:          " + basicAttribs.size());
System.out.println("  Directory?     " + basicAttribs.isDirectory());
System.out.println("  Regular file?  " + basicAttribs.isRegularFile());
System.out.println("  Symbolic link? " + basicAttribs.isSymbolicLink());
System.out.println("  Other?         " + basicAttribs.isOther());
```

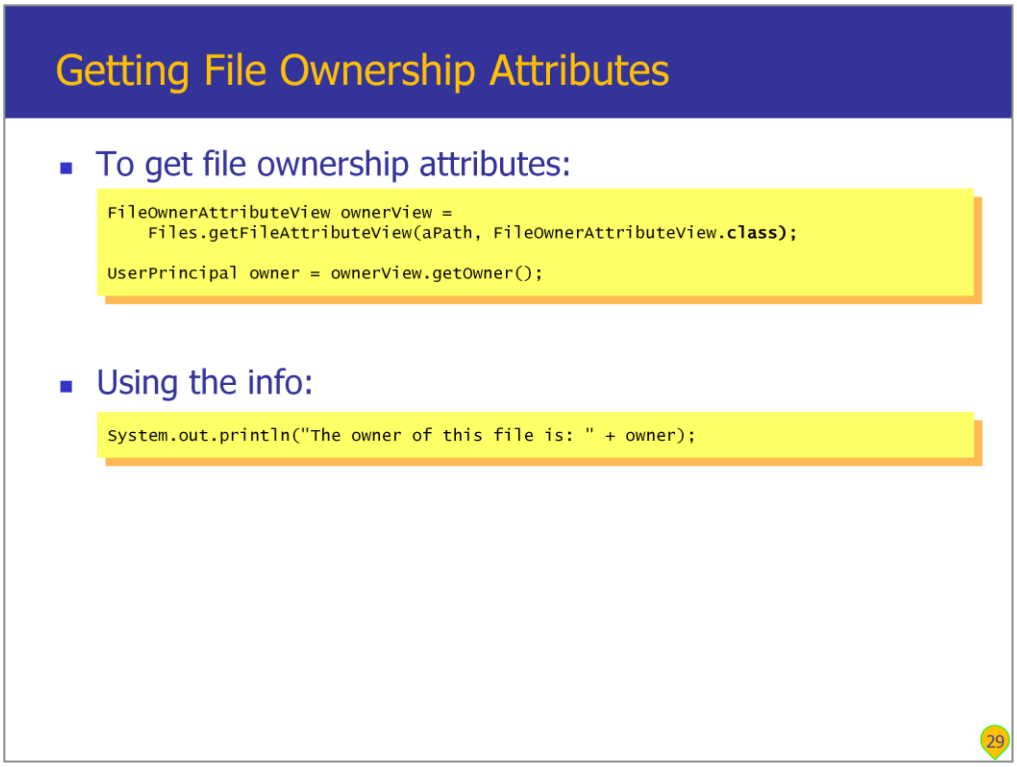
- `BasicFileAttributes` is a base interface
  - Extended by `DosFileAttributes` and `PosixFileAttributes`

28

The code in the slide outputs information such as the following:

```
File attribFile has the following BasicFileAttributes:
  Created:       2014-06-04T20:42:21.751108Z
  Last access:   2014-06-04T20:42:21.751108Z
  Last modified: 2014-06-04T20:42:21.751108Z
  Size:          0
  Directory?     false
  Regular file?  true
  Symbolic link? false
  Other?         false
```

Note that the `BasicFileAttributes` interface defines the basic attributes that are supported by all common platforms. DOS and UNIX environments define their own file attributes, which are captured by the `DosFileAttributes` and `PosixFileAttributes` interfaces respectively. These interfaces extend the `BasicFileAttributes` interface.



## Getting File Ownership Attributes

- To get file ownership attributes:

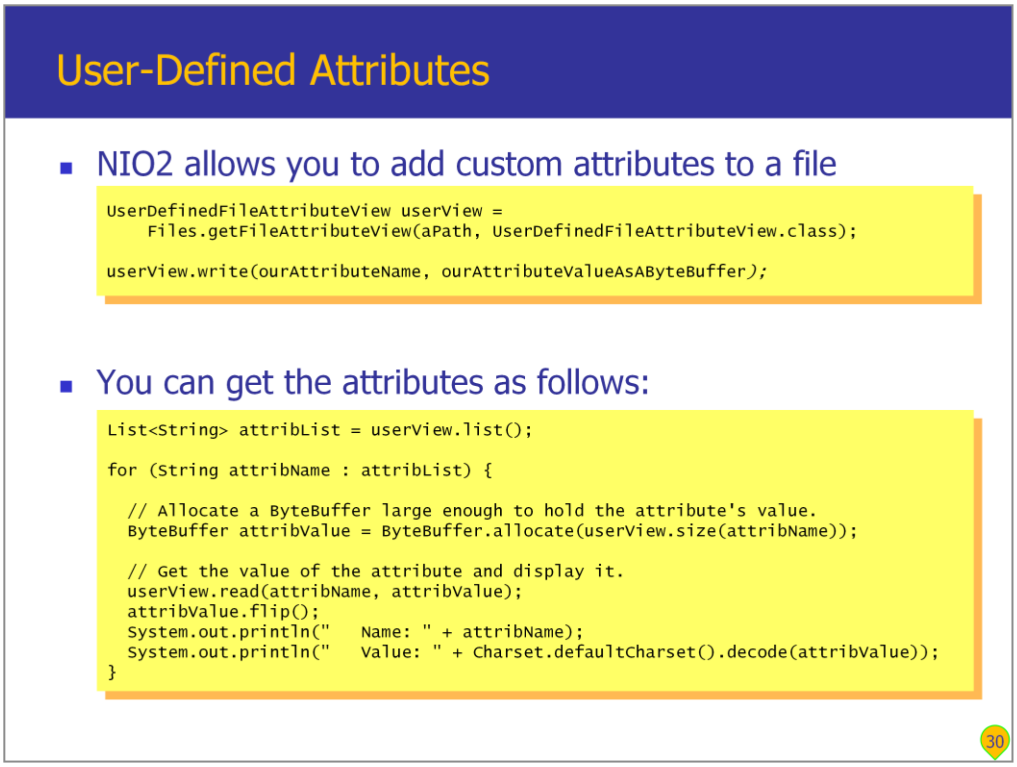
```
FileOwnerAttributeView ownerView =
    Files.getFileAttributeView(aPath, FileOwnerAttributeView.class);

UserPrincipal owner = ownerView.getOwner();
```

- Using the info:

```
System.out.println("The owner of this file is: " + owner);
```

This example gets the owner of the file. When we ran the demo on our setup, it displayed the following:

```
The owner of this file is: ANDYO_PC\andyo_000 (User)
```



## User-Defined Attributes

- NIO2 allows you to add custom attributes to a file

```
UserDefinedFileAttributeView userView =
    Files.getFileAttributeView(aPath, UserDefinedFileAttributeView.class);

userView.write(ourAttributeName, ourAttributeValueAsAByteBuffer);
```

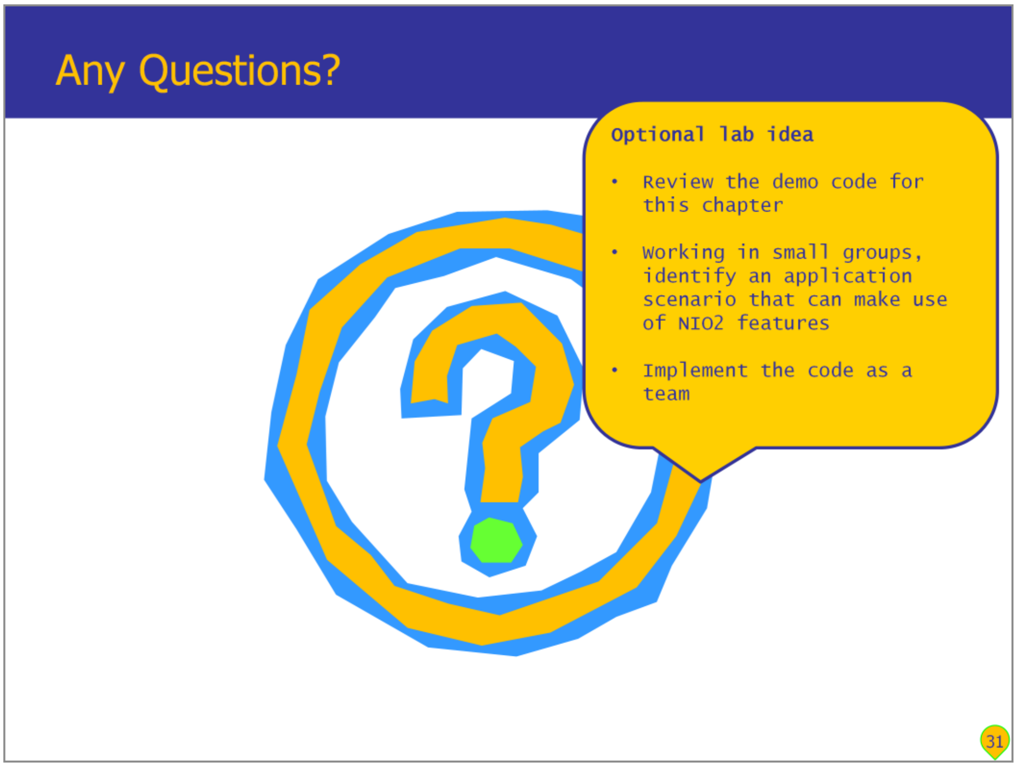
- You can get the attributes as follows:

```
List<String> attribList = userView.list();

for (String attribName : attribList) {

  // Allocate a ByteBuffer large enough to hold the attribute's value.
  ByteBuffer attribValue = ByteBuffer.allocate(userView.size(attribName));

  // Get the value of the attribute and display it.
  userView.read(attribName, attribValue);
  attribValue.flip();
  System.out.println("   Name: " + attribName);
  System.out.println("   Value: " + Charset.defaultCharset().decode(attribValue));
}
```

30

The code in the slide outputs the following information :

```
Attribute Name: Our Attribute Name!
Attribute Value: This is out attribute value!
```



## Any Questions?

**Optional lab idea**

- Review the demo code for this chapter
- Working in small groups, identify an application scenario that can make use of NIO2 features
- Implement the code as a team