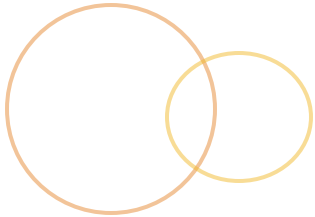
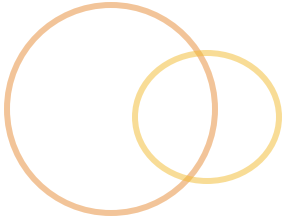
Java NIO File Handling



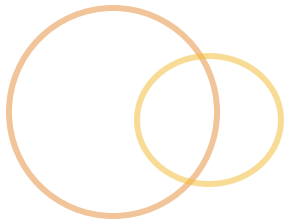


Objectives

Paths, Path, and Files

- Java NIO2 provides greatly improved facilities for handling files and disk storage generally
- Paths is a factory for creating Path objects
- Path objects describe paths on the file system, including the “roots” of those paths, such as Windows-style device names
- Files is a class containing static utility methods for performing a variety of tasks such as copying or moving files, opening files, interacting with directories and file permissions

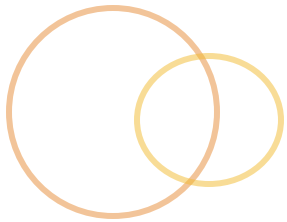
The Paths Class



- Paths is a factory for Path objects, it has two get methods, one that takes a series of Strings describing elements of the desired path, and one that takes a URI

```
String home = System.getenv("HOME");
Path homeDir = Paths.get(home);
Path alsoHome = Paths.get("home", "tony");
```

Using Path Objects



- Several methods allow investigation of the path and the elements that make it up

Method	Effect
`getFileName`	Get the last element of the patth
`getName(idx)`	Get the path element at idx
`startsWith`	Test if the past starts with this element
`endsWith`	Test if the path ends with this element
`equals`	Test if the paths represent the same sequence
`getParent`	Get a path for “up one level”
`getRoot`	Get a path for the root of this path (e.g. “/” or “C:”
`getNameCount`	How many elements in this path (not counting root)
`Iterator`	Iterate the elements of the path
`forEach`	Iterate the elements of the path

Using Path Objects

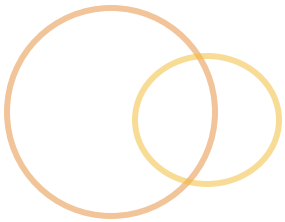
- The Path object provides more utilities

Method	Effect
`isAbsolute`	Test if the path starts at a root
`toAbsolutePath`	Anchor the path at a root, if not already
`toRealPath`	Find real path, following links
`toFile`	Convert to an old-style File object
`normalize`	Clean up references such as “.” and “..”
`relativize`	Compute path from this Path to the arbument Path
`resolve`	Usually, concatenate the argument to this Path, unless the argument Is absolute, in which case return that
`register`	Start watching a Path for changes (discussed later)

Watching Directories

- Java's NIO2 facilities include the ability to monitor a directory for changes
- Steps:
  - Start a Watcher
  - Register a Path for certain types of change
  - Get a notification (passive/pull) from the Watcher when changes occur
    - Each time the Watcher reports, it reports changes as a list, all of which should be investigated
  - Tell the Watcher we've handled the notifications
  - Loop round and wait for more changes

Watching Directories



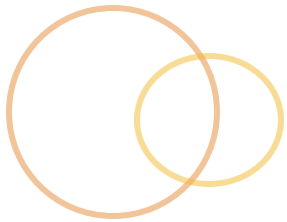
- Set up the watcher

```
Path homeDir = Paths.get(System.getEnv("HOME"));
WatchService watcher =
  FileSystems.getDefault().newWatchService();
```

- Register the directory

```
homeDir.register(watcher,
    StandardWatchEventKinds.ENTRY_CREATE,
    StandardWatchEventKinds.ENTRY_DELETE,
    StandardWatchEventKinds.ENTRY_MODIFY);
```

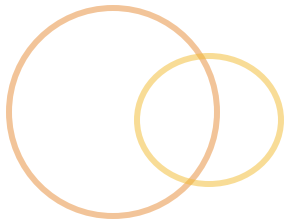
Watching Directories



- Loop processing lists of events in keys

```
for (;;) {
  WatchKey key = watcher.take();
  List<WatchEvent<?>> events = key.pollEvents();
  events.forEach(
    ev -> {
      Path affected = (Path) ev.context();
      System.out.println("File " + affected);
      System.out.println("event is " +
        ev.kind().name());
  });
  key.reset();
}
```

The Files Class



- The Files class is home to static utility methods covering a variety of operations, including:
  - Testing properties (readable, writable, exists)
  - Testing if two Path objects lead to the same file
  - Copying
  - Moving
  - Deleting
  - Opening / creating files for reading/writing
  - Iterating directories and walking file trees

Files Principles

- Files methods might allow “globbing”
  - That is wildcard operations
- Some common globbing include:
  - * indicates any number of characters in a path segment
  - ** indicates any number of path segments
  - {a,b,c} indicates any of a, or b, or c
    - Also, a, b, and c can use other globbing features}
  - [abcdef] any one of the listed characters

Files Principles

- Some Files operations can be atomic
  - They either succeed completely, or leave no change
- Several new exceptions exist, usually with utility methods to get the file(s) affected and more

Interesting Files Methods

- Files has many methods. These are some general purpose ones

Method	Effect
`copy`	Copy files, with options to overwrite / copy attributes
`createXxx`	Options to create files, directories, links, temp files
`delete`	Removes file/directory
`getXxx`	Get various attributes; e.g. owner. permissions
`ixXxx`	Test various aspects such as writeability, directory/file
`move`	Move a file, with overwrite and link control options
`probeContentType`	Attempt to report content type (text, image, etc.)
`setXxx`	Set various attributes
`size`	Get File size

Accessing Data Using Files

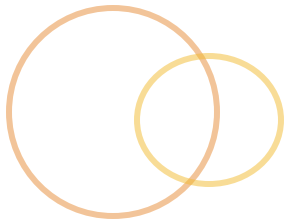
- Files also provides access to disk data

Method	Effect
`lines`	Returns Stream<String> for the text in the file
`list`	Returns Stream<Path> for the paths in a directory
`newBufferedReader`	Opens a file for reading
`newBufferedWriter`	Opens / creates a file for writing
`newDirectoryStream`	Allows reading a directory, with filtering / globbing
`newInputStream`	Opens a file for reading
`newOutputStream`	Opens / creates a file for writing
`readAllXxxx`	Reads all bytes or lines into memory

Traversing Directories Using Files

- Files provides two mechanisms for traversing directory structures
  - Walking
  - Finding

Simple Path Walking

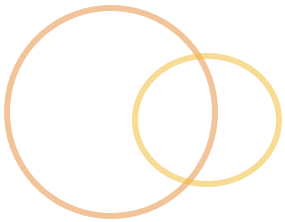


- Two simple `Files.walk` methods produce a Stream<Path> reporting files under the specified path
- This code is a recursive directory listing

```
Path someDir = // directory initialization elided
Stream<Path> paths = Files.walk(someDir);
paths.forEach(System.out::println);
```

- Optional arguments to walk allow
  - Limiting the depth of the search
  - Following links

Simple Path Walking

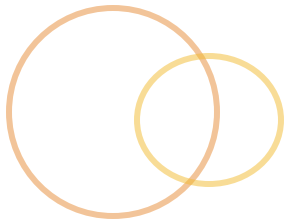


- The starting path should be a directory if any walking is to occur
- The stream is lazy, so if you find what you're looking for, and quit the stream, then less disk IO will occur
- The stream should be closed after use to release kernel level resources
  - The Stream is auto-closeable, so can be used in a try-with-resources construction
- This method can fail with an exception if a directory in the tree is inaccessible

Advanced Path Walking

- The methods Files.walkFileTree allow more control
- A FilesVisitor<Path> should be implemented, which provides four methods:
  - preVisitDirectory
  - visitFile
  - visitFileFailed
  - postVisitDirectory
- Each method has a chance to indicate the behavior that should follow from continue, skip siblings, skip subtree, and terminate

The Find Method



- Files has a find method, with a basic behavior similar to the Unix find command

```
static Stream<Path> find(
 Path start,
 int maxDepth,
 BiPredicate<Path,BasicFileAttributes> matc,
 FileVisitOption... options)
```

- Note that find will throw an exception if it finds an unreadable directory

Lab Exercise

- Count the number of lines in a text file
- List all the files in the current directory that do not begin with a period
  - Indicate if each file is a directory or file and whether it's readable, writable, and/or executable