# CS 251
# Intermediate Programming
# Java I/O – File I/O

Brooke Chenoweth

University of New Mexico

Spring 2024

# Paths

- Most file systems store files in a hierarchical structure.
- The top of the directory tree is a root node (or more than one). Root node on Linux is `/`, on Windows is a drive letter, such as `C:\`
- Root directory contains files and directories, directories can contain subdirectories, and so on.
- A file is identified by its path through the file system.
- Directory names separated by system-specific *delimiter*. (Windows uses backslash, most others use forward slash)

# Absolute vs Relative Paths

- An absolute path always starts at the root.
  - `/home/sally/javacode`
  - `C:\home\sally\javacode`
- All info needed to locate file is contained in absolute path.
- Relative path is *relative* to another path.
  - `bob/foo`
- Relative path needs to be combined with another path to locate a file.

# The `Path` class

- New in Java SE 7
- Located in `java.nio.file` package
- Represents a path in the file system.
- Just a path, does not guarantee that corresponding file or directory actually exists.

# Creating a Path

Create `Path` object by using methods in the `Paths` helper class.

```
Path p1 = Paths.get("/tmp/foo");
Path p2 = Paths.get("/", "tmp", "foo");
```

# Path Operations

- `toString` – string representation
- `getFileName` – get file name (last element in sequence)
- `getParent` – path of parent directory
- `getRoot` – get root of path (`null` for relative paths)
- `toAbsolutePath` – convert path to absolute path relative to current working directory.
- `resolve(otherpath)` – combine this path with another
- `relativize(otherpath)` – create path from this path to other path

# The `Files` class

- A `Path` object represents a file or directory, but says nothing about whether that file exists.
- The `Files` class provides methods access the file system and examine and manipulate files.

# Existence and Accessibility

- `Files.exists(path)` and `Files.notExists(path)` verify if particular `Path` exists or not. Possible for both to return false if file's status is unknown.
- `isRegularFile`, `isDirectory` – What sort of file is this?
- `isReadable`, `isWritable`, `isExecutable` – what can we do with the file?
- `isSameFile` – Do two paths refer to same file?

# Deleting Files

- `Files.delete(path)` – deletes file or throws exception if it fails.
- `Files.deleteIfExists(path)` – deletes file, doesn't complain if file isn't there

# Copying Files

- Use `Files.copy(source, target, copyOptions)` to copy a file.
- Directories can be copied, but files inside are not copied. (Use `walkFileTree` to recursively copy.)
- Copy takes zero or more `CopyOption` arguments.
  - `REPLACE_EXISTING` – If target file exists, replace it instead of throwing `FileAlreadyExistsException`
  - `COPY_ATTRIBUTES` – Try to give target same attributes (access permissions, last modified time, etc.) as source.
  - `NOFOLLOW_LINKS` – If source is a symbolic link, copy the link itself, not the file the link refers to.
- `Files` also has copy methods to copy from an input stream to a file and from a file to an output stream.

# Moving Files

- Use `Files.move(source, target)` to move (or rename) a file or directory.
- Source and target paths should not refer to same file.

# Using Stream I/O with Files

- `Files.newInputStream` creates a new byte input stream from a file path.
- `Files.newOutputStream` creates a new byte output stream from a file path.
- These methods provide *unbuffered* streams.
- For text files, use `Files.newBufferedReader` and `Files.newBufferedWriter`.

# Input Stream From a File

```
Path file = Paths.get("myfilename");

try (InputStream in = Files.newInputStream(file);
  BufferedReader reader =
    new BufferedReader(new InputStreamReader(in))) {

  String line = null;
  while ((line = reader.readLine()) != null) {
    System.out.println(line);
  }

} catch (IOException x) {
  System.err.println(x);
}
```

# Creating Files

```
Path file = Paths.get("myfilename");
try {
    // Create empty file with default permissions, etc.
    Files.createFile(file);

} catch (FileAlreadyExistsException x) {
    System.err.format("file named %s" +
        " already exists%n", file);

} catch (IOException x) {
    // Some other sort of failure, such as permissions.
    System.err.format("createFile error: %s%n", x);
}
```

# Creating Temporary Files

```
try {
   // null Path creates file in system temp directory
   Path tempFile = Files.createTempFile(null, ".junk");
   System.out.format("The temporary file" +
      " has been created: %s%n", tempFile);
} catch (IOException x) {
   System.err.format("IOException: %s%n", x);
}
```

# Walking the File Tree

- `Files.walkFileTree(path, fileVisitor)` will visit all files in the directory given by `path` and perform operations specified by `fileVisitor`.
- File tree is walked *depth first*, but you cannot make any assumptions about order that subdirectories will be visited.

# File Visitor

- The `FileVisitor` interface has four methods
  - `preVisitDirectory` – Invoked before a directory's entries are visited.
  - `postVisitDirectory` – Invoked after all the entries in a directory are visited.
  - `visitFile` – Invoked on the file being visited.
  - `visitFileFailed` – Invoked when the file cannot be accessed.
- The `FileVisitor` methods return `FileVisitResult`
  - `CONTINUE` – Continue walking the tree.
  - `SKIP_SIBLINGS` – Continue without visiting siblings of this file or directory.
  - `SKIP_SUBTREE` – Continue without visiting entries in this directory.
  - `TERMINATE` – Stop walking the tree.

# Example: Printing File Sizes

```
public class PrintFiles
    extends SimpleFileVisitor<Path> {

    public FileVisitResult visitFile(Path file,
                                     BasicFileAttributes attr) {
        System.out.println(file + " (" + attr.size() + "bytes)");
        return FileVisitResult.CONTINUE;
    }

    public FileVisitResult postVisitDirectory(Path dir,
                                              IOException exc) {
        System.out.format("Directory: %s%n", dir);
        return FileVisitResult.CONTINUE;
    }

    public FileVisitResult visitFileFailed(Path file,
                                           IOException exc) {
        System.err.println(exc);
        return FileVisitResult.CONTINUE;
    }
}
```

# Example: Copy File Tree

```
// source, target are Path objects
Files.walkFileTree(source, new SimpleFileVisitor<Path>() {

    public FileVisitResult preVisitDirectory(Path dir,
            BasicFileAttributes attrs) throws IOException {
      Path targetdir = target.resolve(source.relativize(dir));
      try {
        Files.copy(dir, targetdir);
      } catch (FileAlreadyExistsException e) {
        if (!Files.isDirectory(targetdir)) {
          throw e;
        }
      }
      return FileVisitResult.CONTINUE;
    }

    public FileVisitResult visitFile(Path file,
            BasicFileAttributes attrs) throws IOException {
      Files.copy(file, target.resolve(source.relativize(file)));
      return FileVisitResult.CONTINUE;
    }
  });
```

# Dealing with Old API

- Before Java 7, most file I/O used `java.io.File`
- Lots of legacy code out there.
- Can convert between old and new API with `File.toPath` and `Path.toFile` methods.
- Consult the Java I/O tutorial for more information.