
ORACLE
Academy

Java Programming

5-1

Basics of Input and Output

ORACLE
Academy

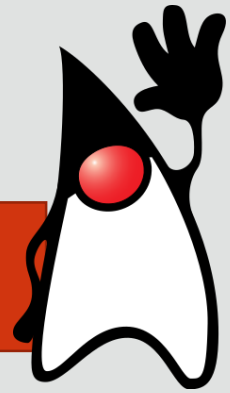
Objectives

- This lesson covers the following topics:
  - Describe the basics of input and output in Java
  - Read data from and write data to the console

Basics of Input and Output

- Java Applications read and write files
- Whether you are reading or writing data from files or transferring data across the internet, input and output is the basis for accomplishing that
- There are two options:
  - java.io package
  - java.nio.file package

These are the two packages that you will be working with to allow you to include Input/Output in your code.



java.io Package Limitations

- The java.io package limitations are:
  - Many methods fail to throw exceptions
  - Operations are missing (like copy and move)
  - No support for symbolic links
  - Many methods fail to scale with large files

The java.io package although useful has some major drawbacks to it. Although it is important to be aware of its existence the course will concentrate mainly on the java.nio.file package.

java.nio.file Package

- The java.nio.file package:
  - Works more consistently across platforms
  - Provides improved access to more file attributes
  - Provides improved exception handling
  - Supports non-native files systems, plugged-in to the system

The java.nio.file package gives you more control when dealing with IO.

When Input and Output Occurs

- Input occurs when Java:
  - Reads the hierarchical file system to find directories
  - Reads physical files from the file system
  - Reads physical files through a symbolic link
  - Reads streams from other programs
- Output occurs when Java:
  - Writes physical files to a directory
  - Writes physical files through a symbolic link
  - Writes streams to other programs

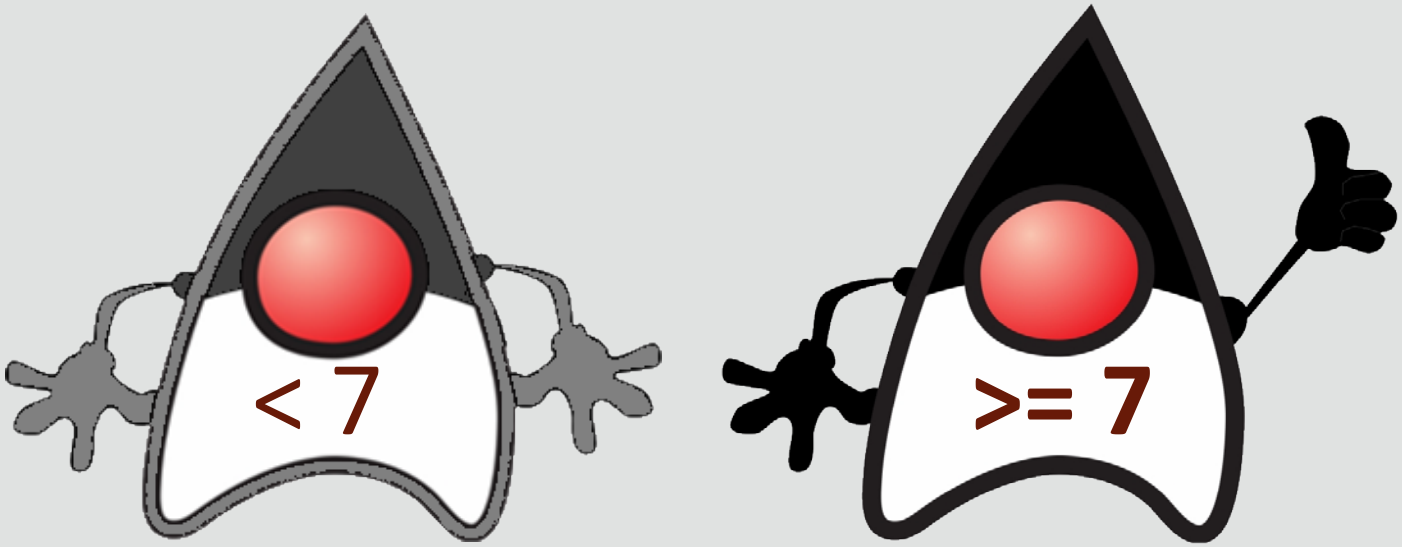
Reading a File Prior to Java 7

- The old way has you construct:
- A new instance of File with a fully qualified file name (includes the path and file name)
- A new FileReader with a File
  - FileReader is meant for reading streams of characters
- A new BufferedReader with a FileReader
  - BufferedReader reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines

The path and filename together make the fully qualified file name. This would be something like **C:\tempfile\temp.txt**.

Reading a File Prior to Java 7

- The changes between pre-Java 7 and Java 7 involve:
  - The division of the Path and File into separate interfaces
  - The delivery of the Paths and Files classes that implement the new interfaces



Reading from file Example 1

1. Create a directory named Java Programming on the C: drive of your computer
2. Create an employees.txt file in the directory with these values

```
Steven King
Neena Kochhar
Jennifer Whalen
Shelley Higgins
William Gietz
Eleni Zlotkey
Ellen Abel
Jonathon Taylor
Kimberely Grant
Kevin Mourgos
Trenna Rajs
Curtis Davies
```

C:\JavaProgramming\employees.txt

Reading from file Example 1

3. Create a project named filereading
4. Create a FileReading class that includes the following methods

```
public class FileReading {
    public static void main(String[] args) {
    }//end method main

    static void displayEmployees(ArrayList<String> employees) {
    }//end method displayEmployees

    static void readFile(ArrayList<String> employees) {
    }//end method readFile
}//end class FileReading
```

Reading from file Example 1

5. Create a local String variable named line that will store each line read from the file
6. Create a BufferedReader object named fileInput that will be used to read the text from the file
7. Add the beginning of a try catch statement as there are lots of potential errors that require handling when working with files

```
static void readFile(ArrayList<String> employees){
   String line = "";
   BufferedReader fileInput = null;

   try {
}//end method readFile
```

Reading from file Example 1

8. Assign the file, using the fully qualified file name to a file reader object that will read the stream of characters
9. Then pass the character stream to the BufferedReader object to read the text from it
10. Use the readLine method to read the first line of text from the file and store it in the String line variable

```
    try {
        fileInput = new BufferedReader(new FileReader(
                    new File("C:/JavaProgramming/employees.txt")));
        line = fileInput.readLine();

}//end method readFile
```

Reading from file Example 1

11. If the line variable has a value then add the line to the ArrayList named employees
12. Read the next line from the file
13. Continue this until there are no more lines to be read from the file and then close the file

```
            line = fileInput.readLine();
            while (line != null) {
               employees.add(line);
               line = fileInput.readLine();
            }//endwhile
            fileInput.close();
         }//end try
      }//end method readFile
}//end class FileReading
```

Reading from file Example 1

14. Add the following catch statements that deal with no file being found, trying to go beyond the end of the file and any input/output errors that occur during the file read

```
      }//end try
      catch(FileNotFoundException e) {
         System.out.println("File not found");
      }//end catch
      catch(EOFException eofe) {
         System.out.println("No more lines to read.");
      }//end catch
      catch(IOException ioe) {
         System.out.println("Error reading file.");
      }//end catch
   }//end method readFile
}//end class FileReading
```

Reading from file Example 1

15. Create a displayEmployees() method that uses an enhanced for loop to display the employees
16. Update the main method with the appropriate variables and method calls

```
public class FileReading {
   public static void main(String[] args) {
      ArrayList<String> employees = new ArrayList<>();
      readFile(employees);
      displayEmployees(employees);
   }//end method main

   static void displayEmployees(ArrayList<String> employees) {
      for(String employee: employees)
         System.out.println(employee);
      //endfor
}//end method displayEmployees
```

Reading from file Example 1

17. By the time you have completed all of the code for this program you will have had to import the following libraries from the java.io and java.util libraries

```
import java.io.BufferedReader;
import java.io.EOFException;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.FileReader;
import java.util.ArrayList;

public class FileReading {

   public static void main(String[] args) {
```

Reading from file Example 1

18. Run the program, you should see the list of employees from the file displayed in the console

```
Steven King
Neena Kochhar
Jennifer Whalen
Shelley Higgins
William Gietz
Eleni Zlotkey
Ellen Abel
Jonathon Taylor
Kimberely Grant
Kevin Mourgos
Trenna Rajs
Curtis Davies
```

19. Change the name of the file in the directory to test the FileNotFound exception handler

```
File not found
```

Reading a File Prior to Java 7

- A static method call to Paths.get() returns a valid path
- A static method call to Files.newBufferedReader() returns a file as a BufferedReader instance
- Inside the call to the newBufferedReader() method, another static call is made to the Charset.forName() method
- This approach uses static method calls to return a BufferedReader class instance

This process has been greatly simplified with the introduction of Java 7 and beyond.

Reading a File Prior to Java 7

- The methods shown in operation throughout the next example shows how to define the character set that is to be imposed on the file contents
- This program reads a file and defines its character set
- Character sets that are supported include:
  - UTF-16 (The native character encoding of Java)
  - UTF-16BE
  - UTF-16LE
  - UTF-8
  - US-ASCII
  - ISO-8859-1

For more information on character sets investigate the Java documentation.

ORACLE Academy

Reading from file Example 2

1. Create a project named filehandling
2. Create a FileHandling class that includes the following methods

```
public class FileHandling {
    public static void main(String[] args) {
    }//end method main

    static void displayEmployees(ArrayList<String> employees) {
    }//end method displayEmployees

    static void readFile(ArrayList<String> employees) {
    }//end method readFile
}//end class FileHandling
```

Reading from file Example 2

3. Create a local String variable named line that will store each line read from the file
4. Create a Path object named path that holds the absolute file path returned from the static method get() from the Path class
5. Add the beginning of a try catch statement to handle the errors that may occur when working with files

```
   static void readFile(ArrayList<String> employees){
      String line = "";
      Path path = Paths.get("C:/JavaProgramming/employees.txt");
      try {
   }//end method readFile
}//end class FileHandling
```

This static call returns an instance of an absolute file path.

Reading from file Example 2

6. Use the Files class that consists exclusively of static methods that operate on files, directories, or other types of files to return a BufferedReader object from the path value supplied
7. As part of this operation the character set is defined

```
try {
   BufferedReader fileInput =
         Files.newBufferedReader(path, Charset.forName("ISO-8859-1"));
   line = fileInput.readLine();
   while (line != null) {
      employees.add(line);
      line = fileInput.readLine();
   }//endwhile
   fileInput.close();
}//end try
```

Calls the static method and returns a Charset instance.

Reading from file Example 2

8. Complete the catch statements to handle the errors in exactly the same way as the previous example
9. Create the display method to use the enhanced for loop to display the values to the console
10. Add the variables and method calls to main

```
public static void main(String[] args) {
   ArrayList<String> employees = new ArrayList<>();
   readFile(employees);
   displayEmployees(employees);
}//end method main

static void displayEmployees(ArrayList<String> employees) {
for(String employee: employees)
   System.out.println(employee);
}//end method displayEmployees
```

Writing a File Prior to Java 7

- Writing to files works in a very similar fashion to reading from files
- A **BufferedReader** object is used to read from a file
- A **BufferedWriter** object is used to write to a file
- The `BufferedWriter.write()` method writes a String to the file
- You can set the StandardOpenOption of a file to:
  - CREATE – create a new file if it doesn't exist
  - WRITE – Overwrite the contents of the file
  - APPEND – Add the new text to the end of the file

Reading from file Example 2

11. Add the following writeFile method to the FileHandling program, import any libraries as needed

```
public class FileHandling {
   public static void main(String[] args) {
      ArrayList<String> employees = new ArrayList<>();
      readFile(employees);
      displayEmployees(employees);
      writeFile(employees);
   }//end method main

   static void displayEmployees(ArrayList<String> employees) {
   }//end method displayEmployees

   public static void writeFile(ArrayList<String> employees) {
   }//end method writeFile

   static void readFile(ArrayList<String> employees) {
   }//end method readFile
}//end class FileHandling
```

Reading from file Example 2

12. Set the path to write to a file named userNames.txt
13. Create the file if it doesn't exist and overwrite the contents, a for-each is used to provide the Strings

```
public static void writeFile(ArrayList<String> employees) {
   Path path = Paths.get("C:/JavaProgramming/userNames.txt");
   try {
      BufferedWriter bw = Files.newBufferedWriter(path,
         Charset.forName("ISO-8859-1"),
         StandardOpenOption.CREATE, StandardOpenOption.WRITE);
      for(String employee: employees) {
         bw.write(employee);
         bw.newLine();
      }//endfor
      bw.close();
   }//end try
}//end method writeFile
```

The StandardOpenOption enum provides options for streams.

Reading from file Example 2

14. Create a catch statement that will handle any IO errors that are produced when writing to the file
15. The program should exit if an error occurs

```
   }//end try
   catch(IOException ioe) {
      System.out.println("Error reading file.");
      System.exit(0);
   }//end catch
}//end method writeFile
```

16. Test your program to ensure that the employees are added to the `userNames.txt` file in the JavaProgramming directory

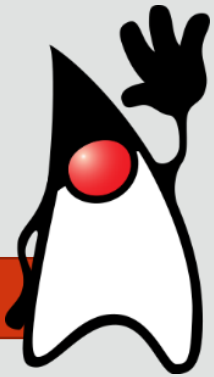
File Interface and Files Class

- The new Path and File interfaces of Java 7 and beyond replace the old way of managing directories and files
- The File interface and Files class:
  - Creates files and symbolic links
  - Discovers and sets file permissions
  - Reads and writes files

These methods replace the static new FileReader method for a File and the new BufferedReader for a FileReader that were used prior to Java 7.

Paths Interface and Paths Class

- The Path interface and Paths class:
  - Navigates the file system
  - Works with relative and absolute paths
- Paths are hierarchical structures:
  - Sometimes called inverted trees because they start at one top-most point and branch out downward
- Path elements are directories or nodes
- Directories may hold other directories or files



The path is the route that has to be followed in order to get to a file.

Absolute and Relative Paths

- Absolute paths always start from:
  - A logical drive letter on Windows (C:\ or D:\)
  - A / (forward slash) or mount point on Unix/Linux
- Relative paths are directories in a path and they:
  - May be the top-most (or root node) directory
  - May be the bottom-most (or leaf node) directory
  - May be any directory between the top and bottom

An absolute path starts with a logical mount, like **C:** or **D:** in Windows.
A relative path starts somewhere other than the root node and ends in a file name.

Absolute versus Relative File Paths

- Absolute file path on Unix/Linux:

```
/home/username/data/filename
```

- Relative file path on Unix/Linux:

```
data/filename
```

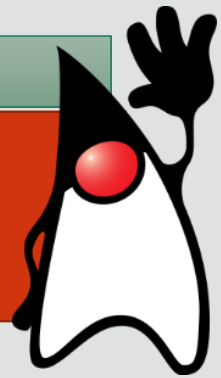
- Absolute file path on Windows:

```
C:\Users\UserName\data\filename
```

- Relative file path on Windows:

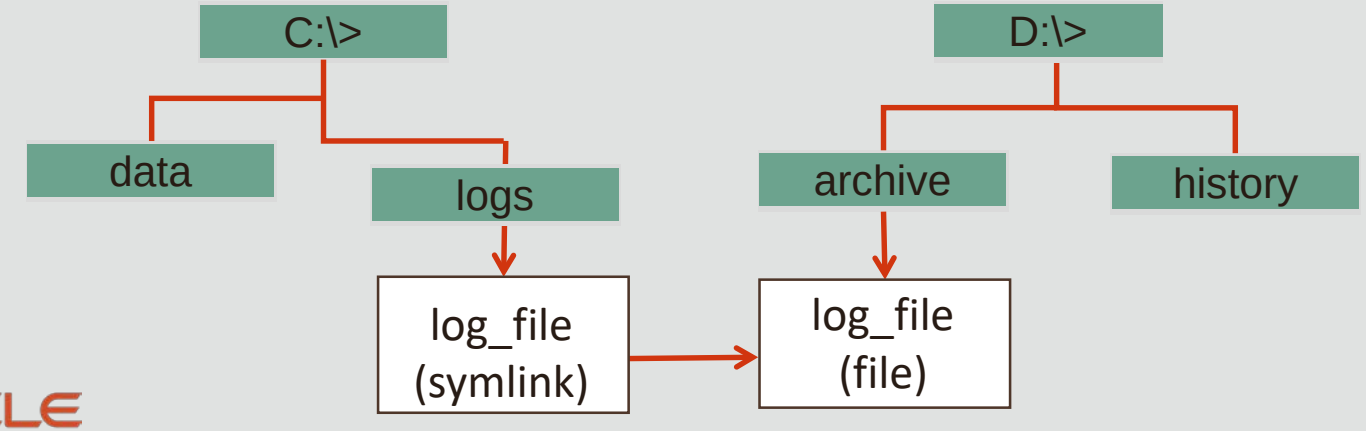
```
data\filename
```

The absolute path takes you from the root of the drive to the file location.
The relative path takes you to the file from your current location.



Symbolic Links

- Symbolic Links are special files that reference another file (sometimes called symlinks)
- Symbolic Links can reference:
  - A file in the same directory
  - A file in another directory of the same path
  - A file in another directory of a different path



ORACLE Academy

Symbolic Links

- Example:
  - When a program has minor releases and the compiled program should still call the program, how do you manage that when the program name includes the numbers that represent the minor release numbering?
- Solution: Create a major release symbolic link that includes only the major release number, and have it point to the most current minor release
- This way the compiled program doesn't have to change as frequently

Java NIO.2 Interfaces and Classes

- Locate a file or directory by using a system dependent path with:
  - java.nio.file.Path interface
  - java.nio.file.Paths class
- Using a Path class, perform operations on files and directories:
  - java.nio.file.File interface
  - java.nio.file.Files class

Java NIO.2 Interfaces and Classes

- java.nio.file.FileSystem class
  - A FileSystem class provides an interface to a file system and a factory for creating a Path class instance and other file system object instances
  - All methods throw an IOException that accesses the file system with Java NIO.2 classes
  - Interfaces give developers more possibilities
  - The Paths and Files classes provide many static methods that let you construct streams, which simplifies the process

Construct a Path

- There are 2 classes here, FileSystem which is abstract and FileSystems plural which is concrete
- The getDefault method returns the default file system to allow the Java Virtual Machine to access the data

```
import java.nio.file.FileSystem;
import java.nio.file.FileSystems;
import java.nio.file.Path;

public class FileHandling7 {
   public static void main(String[] args) {
      FileSystem fs = FileSystems.getDefault();
      Path path = fs.getPath("C:\\JavaProgramming\\employees.txt");
      System.out.println("Default Directory [" + path + "]");
   }//end method main
}//end class FileHandling7
```

This is a concrete class that implements the abstract FileSystem class.

For more information on these consult the Java API.

Construct a Path

```
public static void main(String[] args)  {
    FileSystem fs = FileSystems.getDefault();
    Path path = fs.getPath("C:\\JavaProgramming\\employees.txt");
    System.out.println("Default Directory [" + path + "]");
}//end method main
```

- A backslash must precede the backslash \\ in a string path when a Windows system is being used
- A \\ (double backslash) is unnecessary in Windows as a / (forward slash) is the preferred solution

Output:

```
Default Directory [C:\JavaProgramming\employees.txt]
```

Construct a Path

- NIO.2 converts the forward slash to a backslash, so the following string could be used to create the exact same path:

```
public static void main(String[] args) {
   FileSystem fs = FileSystems.getDefault();
 //Path path = fs.getPath("C:\\JavaProgramming\\employees.txt");
   Path path = fs.getPath("C:/JavaProgramming/employees.txt");
   System.out.println("Default Directory [" + path + "]");
}//end method main
```

Output:

```
Default Directory [C:\JavaProgramming\employees.txt]
```

Using the Path class Example 2

1. Create a the following code that uses the different options available with the Paths.get() method

```
public class PathDemo {
   public static void main(String[] args) {
      Path[] paths = new Path[5];
      paths[0] = Paths.get("C:\\JavaProgramming\\NIO2\\DemoFile.txt");
      paths[1] = Paths.get("C:/JavaProgramming/NIO2/DemoFile.txt");
      paths[2] = Paths.get("C:","JavaProgramming","NIO2","DemoFile.txt");
      paths[3] = Paths.get("DemoFile.txt");
      paths[4] = Paths.get(URI.create("file:///~/DemoFile.txt"));

      for(int i = 0; i<paths.length; i++)
         System.out.println("Default File Path p[" + i + "] - “
                            + paths[i]);
      //endfor
   }//end method main
}//end class PathDemo
```

Constructing Path Instances

- The previous code showed examples of valid syntax for constructing Path instances
- Output would look like the following:

```
Default File Path p[0] - C:\JavaProgramming\NIO2\DemoFile.txt
Default File Path p[1] - C:\JavaProgramming\NIO2\DemoFile.txt
Default File Path p[2] - C:\JavaProgramming\NIO2\DemoFile.txt
Default File Path p[3] - DemoFile.txt
Default File Path p[4] - \~\DemoFile.txt
```

As you can see there are lots of different ways of working with paths.

Analyze a Path and its Contents

- The Path class has many methods that can be used to return values based on a given path

Method	Description
`getFileName()`	Returns the name of the file or directory denoted by this path as a Path object.
`getParent()`	Returns the parent path, or null if this path does not have a parent.
`getNameCount()`	Returns the number of name elements in the path.
`isAbsolute()`	Tells whether or not this path is absolute.
`toAbsolutePath()`	Returns a Path object representing the absolute path of this path.
`toUri()`	Returns a URI to represent this path.

More Path methods can be found in the Java API

ORACLE Academy

Analyze a Path and its Contents

- The instance methods return the Path values
- isAbsolute() returns a boolean that has been converted to a String for display
- The isAbsolute() method returns true when the path contains the root node of a file hierarchy, like / in Unix or Linux or C:\ in Windows
- toAbsolutePath() will return the absolute path from a given relative Path

Make sure you review the methods used to analyse a constructed Path until you are happy with how it works.

Using the Path class Example 2

2. Add the following code to the bottom of your PathDemo class and look at the given output to view the result of each method call

```
      for(int i = 0; i<paths.length; i++)
         System.out.println("Default File Path p[" + i + "] - " +
                            paths[i]);
      //endfor

      System.out.println(paths[0].getFileName());
      System.out.println(paths[0].getParent());
      System.out.println(paths[0].getNameCount());
      System.out.println(paths[0].isAbsolute());
      System.out.println(paths[3].toAbsolutePath());
      System.out.println(paths[0].toUri());
   } //end method main
} //end class PathDemo
```

Remove Path Redundancies

- The following symbols can be used to reduce redundancies:
  - The **.** (dot) refers to the present working directory in Unix, Linux, or Windows operating systems
  - The **..** (double dot) refers to the parent directory of the present working directory
  - The **normalize()** method creates the direct path to the absolute file path

If you think of a hierarchical tree structure the single . (dot) represents the directory you are currently in whereas the .. (double dot) takes you up a branch in the directory structure.

Remove Path Redundancies

- This example navigates down to an IO directory in the JavaProgramming directory before then coming back up a level (parent) using the .. Notation
- It then navigates down to the NIO2 (sibling) directory

```
Path rp =
Paths.get("C:/JavaProgramming/IO/../NIO2//DemoFile.txt");
System.out.println("rp.normalize() [" + rp.normalize() + "]");
```

- The rp relative path is then normalized to create a redundancy free path

3. Add the above rp code to the bottom of the PathDemo class!

Working with Subpaths

- A Path's subpath method is similar to a String's substring method, but it segments a Path, not a String
- Given the beginning and ending index, this method returns the part of the path from the beginning index to the ending index

```
Path subpath(int beginIndex, int endIndex);
```

This method allows you to create sub paths from the absolute path, allowing you to create relative paths from an absolute path.

Working with Subpaths

- In the following example:
  - The beginning node is 0
  - The ending node is the maximum number of nodes in the path, or the return value of getNameCount()
  - Normalizes the first path by shortening it from 5 to 3 nodes

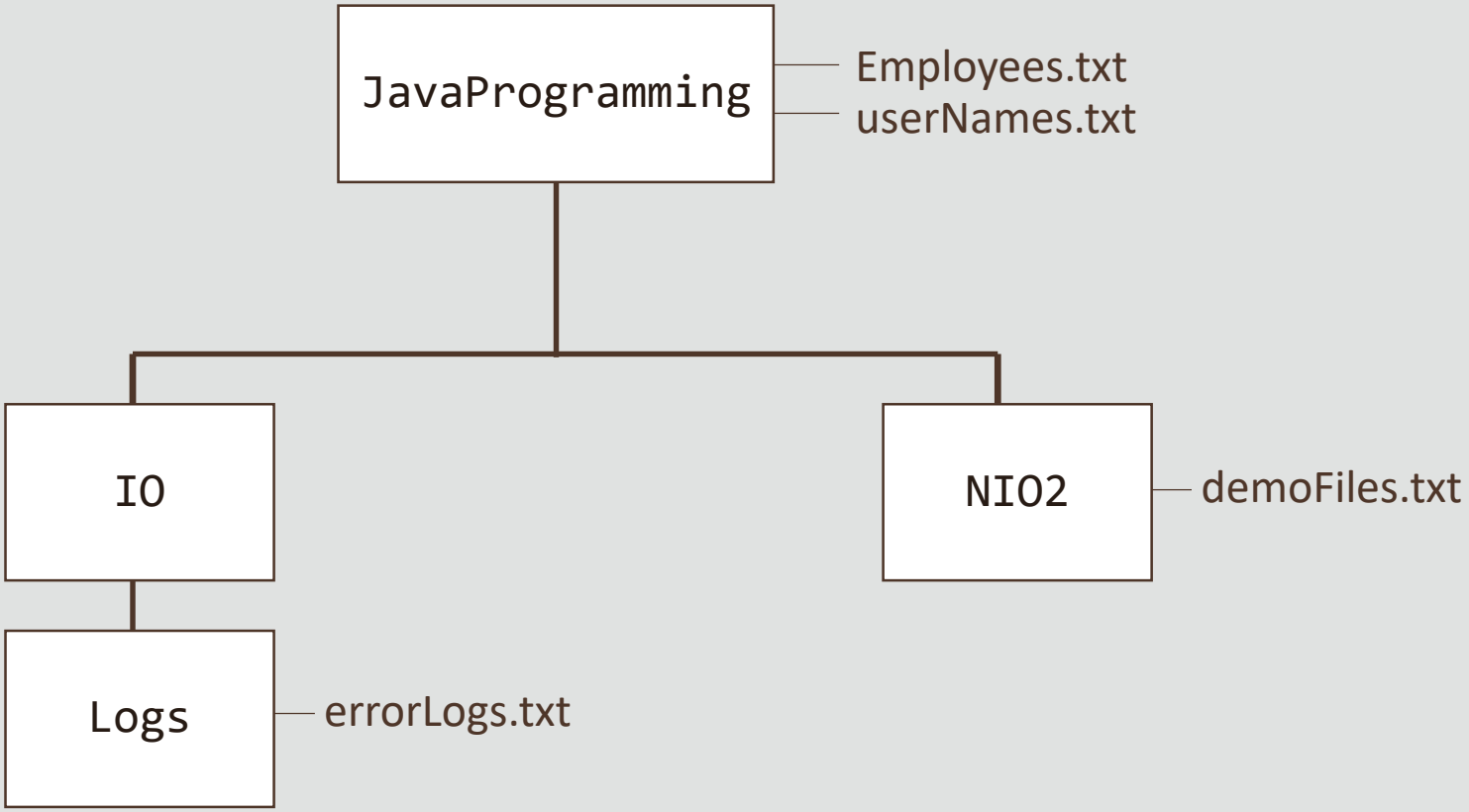
```
Path sp =
Paths.get("C:/JavaProgramming/IO/../NIO2//DemoFile.txt");

System.out.println("p.subpath() [" + sp.getNameCount() +
                   "][" + sp.subpath(0,5) + "]");

System.out.println("p.subpath() [" + sp.getNameCount() +
                   "][" + (sp.normalize()).subpath(0,3) + "]");
```

Working with Subpaths Example

4. Create the following folder structure on your C drive:



Working with Subpaths Example

5. Add the following code to the bottom of the PathDemo!

```
Path sp = Paths.get("C:/JavaProgramming/IO/../NIO2//demoFiles.txt");
System.out.println("path.subpath() [" + sp.getNameCount() +
                   "][" + sp.subpath(0,5) + "]");
System.out.println("path.subpath() [" + sp.getNameCount() +
                   "][" + (sp.normalize()).subpath(0,3) + "]");
```

- Output would look like the following:

```
path.subpath() [5][JavaProgramming\IO\..\NIO2\DemoFile.txt]
path.subpath() [5][JavaProgramming\NIO2\DemoFile.txt]
```

Working with Subpaths Example 2

- The following normalizes the nodes and then uses the getNameCount() method to find the last node in the path

```
System.out.println("path.subpath() [" + sp.getNameCount() + "][" +
(sp.normalize()).subpath(0,sp.normalize().getNameCount()-1) + "]");
```

- This line removes any redundancies from the path's sub-path and uses getNameCount – 1 to return the number of the last node on the path

The getNameCount() method is used for discovering the number of node levels in a directory tree.

Join Two Paths

- The overloaded resolve() method can be used with a Path
- Path resolve(Path other)
  - If the other component is an absolute path then that is returned. If other does not have a root component then it is added to the end of the Path

```
Path.resolve(Path path)
```

- Path resolve(String other)
  - Converts a given path string to a Path and resolves it against this Path in exactly the manner specified by the resolve(Path other) method

```
Path.resolve(String path)
```

Join Two Paths

1. Create the following class:

```
public class JoinPathDemo {
   public static void main(String[] args) {
      Path basicPath = Paths.get("C:/JavaProgramming");
      Path newPath = Paths.get("IO/Logs");

      //display the paths to the console.
      System.out.println(basicPath.toString());
      System.out.println(newPath.toString());

      //Add a path not found in it(adds newPath to basicPath).
      Path basicPath2 = basicPath.resolve(newPath.toString());
      //Returns the absolute portion(basicPath).
      Path newPath2 = newPath.resolve(basicPath.toString());

      //display the resulting paths to the console.
      System.out.println(basicPath2.toString());
      System.out.println(newPath2.toString());
   }//end method main
}//end class JoinPathDemo
```

newPath is not an absolute address so it is added to the end of basicPath.

basicPath is an absolute address so that is returned.

Find the Relative Path Between Two Paths

- The relativize() method constructs a path from one location to another when:
  - It requires relative paths
  - It only works when working between nodes of the same file directory tree (hierarchy)
  - It raises an IllegalArgumentException when given a call parameter in another directory tree

```
Path p1 = Paths.get("C:/JavaProgramming");
Path p2 = Paths.get("Projects");



// Output value of join between two paths.cd
System.out.println("p1.realativize(p1) ["
                   + p1.relativize(p2).toString() + "]");
```

Will cause an illegalArgumentsException error as both paths are not from the same directory tree.

Find the Relative Path Between Two Paths

- When you provide two paths that originate in the same directory tree then the relative path will be produced

1. Create the following class:


```
public class RelativizePathsDemo {
   public static void main(String[] args) {
        Path p1 = Paths.get("C:/JavaProgramming/IO/Logs/errorLogs.txt");
        Path p2 = Paths.get("C:/JavaProgramming/IO");
        Path p3 = p2.relativize(p1);
        System.out.println(p3);
    }//end of method main
}//end of class RelativizePathsDemo
```

P1 is the Absolute path
P2 is the Base path
P3 is the Relative path

- The output of this code will be:

```
Logs\errorLogs.txt
```

ORACLE Academy

Paths Class is Link Aware

- Every Paths method:
  - Detects what to do when encountering a symbolic link
  - Provides configurations options for symbolic links
- The Files class provides these static methods for symbolic links:

```
Files.createSymbolicLink(Path, Path, FileAttribute <V>);
Files.createLink(Path, Path);
Files.readSymbolicLink(Path);
Files.isSymbolicLink(Path);
```

- On a windows system you will require administrator privileges to create a symbolic link!

Paths Class is Link Aware

- Target link must exist
- No hard links allowed on directories
- No hard links across partitions or volumes
- Hard links behave like files and the isSymbolicLink() method discovers them
- Symbolic links are used to link files across multiple volumes or drives
- It is important to ensure that the target actually exists before setting up the link

You will probably not have the permission to work with symbolic links on a Windows based system.

Terminology

- Key terms used in this lesson included:
  - Absolute Path
  - Relative Path
  - Normalize path
  - Path Name

Summary

- In this lesson, you should have learned how to:
  - Describe the basics of input and output in Java
  - Read data from and write data to the console


ORACLE
Academy