



# Java Programming

8-1

Basics of Input and Output



# 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



# java.io Package Limitations

The java.io package limitations are:

- Many methods fail to throw exceptions.
- Operations are missing (like, copy, move, and such).
- No support for symbolic links.
- Many methods fail to scale with large files.



# 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.



# 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, which is one that 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.





# 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 a File Prior to Java 7 (Example1)

```
package input1;
import java.io.BufferedReader;
import java.io.EOFException;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class FileReading {
    public static void main(String[] args) {
        String line = "", fileContent = "";
        try {
            BufferedReader fileInput = new BufferedReader(new FileReader(new
                File("C:/BlueJ/Fellowship.txt")));
            line = fileInput.readLine();
            fileContent = line + "\n";
            while (line != null) {
                line = fileInput.readLine();
                if (line != null)
                    fileContent += line + "\n";
            }
            fileInput.close();
        }
    }
}
```

This creates a buffered reader for file from the file system.

Reads one line at a time.

Code continues on next slide...

# Reading a File Prior to Java 7 (Example1)

... code continued from previous slide

```
catch(EOFException eofe) {  
    System.out.println("No more lines to read.");  
    System.exit(0);  
}//end catch  
  
catch(IOException ioe) {  
    System.out.println("Error reading file.");  
    System.exit(0);  
}//end catch  
System.out.println(fileContent);  
}//end method main  
}//end class FileReading
```

This catches any errors that occur due to the code trying to go beyond the end of the file.

This catches any errors that occur due to the file not being 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.
- These methods are shown in operation throughout the next example which shows you how to define the character set that you want to impose on your file contents.

# Reading a File Prior to Java 7 (Example2)

```
package input2;

import java.io.*;
import java.nio.charset.Charset;
import java.nio.file.*;

public class FileReading2 {
    public static void main(String[] args) {
        String line = "", fileContent = "";
        Path p = Paths.get("C:/BlueJ/Fellowship.txt");
        try {
            BufferedReader fileInput =
                Files.newBufferedReader(p, Charset.forName("ISO-8859-1"));
            line = fileInput.readLine();
            fileContent = line + "\n";
            while (line != null) {
                line = fileInput.readLine();
                if (line != null)
                    fileContent += line + "\n";
            } //endif
            fileInput.close();
        } //end try
    }
}
```

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

Calls the static method to create new BufferedReader

Calls the static method and returns a Charset instance.

Code continues on next slide...

# Reading a File Prior to Java 7 (Example2)

... code continued from previous slide

```
catch(EOFException eofe) {  
    System.out.println("No more lines to read.");  
    System.exit(0);  
} //end catch
```

This catches any errors that occur due to the code trying to go beyond the end of the file.

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

This catches any errors that occur due to the file not being found.

```
System.out.println(fileContent);  
} //end method main  
} //end class FileReading2
```

# Writing a File Prior to Java 7

The `BufferedWriter.write()` method writes the entire String as a file.

```
public static void writeFile(String fileContent) {  
    Path p = Paths.get("C:/BlueJ/OutputFile.txt");  
    try {  
        BufferedWriter bw = Files.newBufferedWriter( p  
                                                    , Charset.forName("ISO-8859-1")  
                                                    , StandardOpenOption.CREATE  
                                                    , StandardOpenOption.APPEND );  
  
        bw.write(fileContent, 0, fileContent.length());  
        bw.close();  
    } //end try  
  
    catch(IOException ioe) {  
        System.out.println("Error reading file.");  
        System.exit(0);  
    } //end catch  
}
```

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

Calls the static method and returns a Charset instance.

The `StandardOpenOption` enum provides options for streams.

# Working with Files Prior to Java 7 (Full Example)

Putting it all together:

```
package input2;

import java.io.*;
import java.nio.charset.Charset;
import java.nio.file.*;

public class FileIO {
    public static void main(String[] args) {
        String fileContent = "";

        fileContent = readFile();
        if(fileContent.length()>0){
            writeFile(fileContent);
            System.out.println(fileContent);
        }
        else
            System.out.println("File has no contents");
        //endif
    } //end method main
}
```

Code continues on next slide...



# Working with Files Prior to Java 7 (Full Example)

```
/**
 * The readfile method will read text from a file
 * @return the contents of the file as a string
 */
public static String readFile(){
    String line = "", fileContent="";
    Path p = Paths.get("C:/BlueJ/Fellowship.txt");
    try {
        BufferedReader fileInput = Files.newBufferedReader(p ,Charset.forName("ISO-8859-1"));
        while (line != null) {
            line = fileInput.readLine();
            if (line != null)
                fileContent += line + "\n";
        } //endif
        fileInput.close();
    } //end try
    catch (EOFException eofe) {
        System.out.println("No more lines to read.");
        System.exit(0);
    } //end catch
    catch (IOException ioe) {
        System.out.println("Error reading file.");
        System.exit(0);
    } //end catch
    return fileContent;
} //end method readFile
```

... code continued from previous slide

Code continues on next slide...

# Working with Files Prior to Java 7 (Full Example)

Code continued from previous slide...

```
/**
 * The writeFile method will write the contents of the string to file
 * If the file does not exist it will create it.
 * If it does exist it will add the string to the end of the file contents
 * @param fileContent
 */
public static void writeFile(String fileContent) {
    Path p = Paths.get("C:/BlueJ/OutputFile.txt");
    try {
        BufferedWriter bw = Files.newBufferedWriter( p
            , Charset.forName("ISO-8859-1")
            , StandardOpenOption.CREATE
            , StandardOpenOption.APPEND );

        bw.write(fileContent, 0, fileContent.length());
        bw.close();
    } //end try

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

End of code



# File Interface and Files Class

- The new Path and File interfaces of Java 7 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.



# 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.

# Absolute and Relative Paths

- Absolute paths always start from:
  - A logical drive letter on Windows.
  - 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.

# 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:

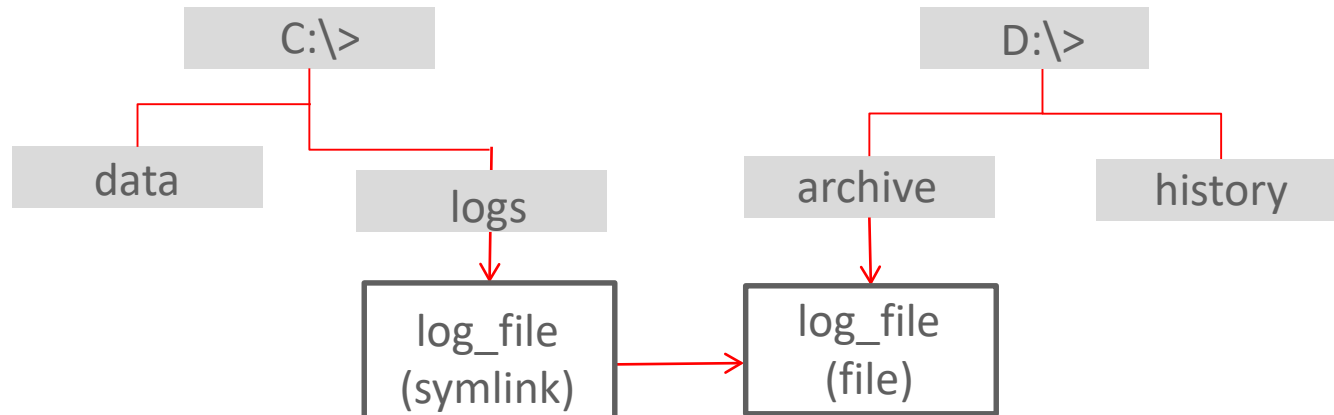
```
C:\Users\UserName\data\filename
```

- Relative file path on Windows:

```
data\filename
```

# 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.



# Symbolic Links Example

- For 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

- A file system class provides an interface to a file system and a factory for creating a Path class instance and other file system object instances.
- `java.nio.file.FileSystem` class
- All methods throw `IOException` that access 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

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

```
public static void main(String[] args) {  
    FileSystem fs = FileSystems.getDefault();  
    Path p = fs.getPath("C:\\\\BlueJ\\NIO2\\DemoFile.txt");  
    System.out.println("Default Directory [" + p + "]");  
    //end method main
```

- A backslash must precede the backslash in a string path when you use a Windows backslash.
- A \\ (double backslash) is unnecessary in Windows. A / (forward slash) is the preferred solution.
- NIO.2 converts the forward slash to a backslash, and you could use the following string to create a path:

```
Path p = fs.getPath("C:/BlueJ/NIO2/DemoFile.txt");
```

# Constructing Path Instances

These are all valid syntax for constructing Path instances.

```
import java.net.URI;
import java.nio.file.Path;
import java.nio.file.Paths;

public class PathDemo {
    public static void main(String[] args) {
        Path[] p = new Path[5];
        p[0] = Paths.get("C:\\\\BlueJ\\\\NI02\\\\DemoFile.txt");
        p[1] = Paths.get("C:/BlueJ/NI02/DemoFile.txt");
        p[2] = Paths.get("C:", "BlueJ", "NI02", "DemoFile.txt");
        p[3] = Paths.get("DemoFile.txt");
        p[4] = Paths.get(URI.create("file:///~/DemoFile.txt"));
        System.out.println("Default File Path p[0] [" + p[0] + "]");
        System.out.println("Default File Path p[1] [" + p[1] + "]");
        System.out.println("Default File Path p[2] [" + p[2] + "]");
        System.out.println("Default File Path p[3] [" + p[3] + "]");
        System.out.println("Default File Path p[4] [" + p[4] + "]");
    } //end method main
} //end class PathDemo
```

# Analyze a Path and its Contents

- The instance methods return the values, except `isAbsolute()`, which 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.

```
p.getFileName()      [DemoFile.txt]
p.getParent()        [C:\BlueJ\NIO2]
p.getNameCount()     [3]
p.isAbsolute()       [true]
p.toAbsolutePath()    [C:\BlueJ\NIO2\DemoFile.txt]
p.toUri()             [file:///C:/BlueJ/NIO2/DemoFile.txt ]
```

The `getNameCount` returns the number of nodes in the path.



# 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.

# Remove Path Redundancies

- This example navigates down to the IO directory, then up using the .. notation, then down to the NIO2 (sibling) directory.

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

- It prints the normalized, or redundancy free path:

```
rp.normalize() [C:\BlueJ\NIO2\DemoFile.txt];
```

- The .. notation takes you up a level in your directory structure.

# 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);
```



# Working with Subpaths Example 1

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:/BlueJ/IO/../../NIO2/Path/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 2

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

```
(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.

# Join Two Paths

The resolve() method allows:

- Adding a node as a String to a path (if the other path does not already exist in the current path).
- Removing a node as a String from a path (if the other path already exists in the current path it is removed).

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

# Join Two Paths

The following shows the syntax:

```
Path bp = Paths.get("C:/BlueJ");
Path np = Paths.get("NI02/Path");
//Add a path not found in it.
Path bp2 = bp.resolve(np.toString());
//Remove a path found in it.
Path np2 = np.resolve(bp.toString());
//display the resulting paths to the console.
System.out.println(bp2.toString());
System.out.println(np2.toString());
```

This adds the subpath to the path.

This removes what was added and leaves: C:\BlueJ as the path.

# 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.

```
// Declare Path instances.  
Path p1 = Paths.get("C:/BlueJ/NIO2");  
Path p2 = Paths.get("Projects");  
  
// Output value of join between two paths.cd  
System.out.println("p1.relativize(p1) [" +  
p1.relativize(p2).toString() + "]);
```

The `relativize()` method only works with two relative paths.

# 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.

```
public class RelativizePaths{  
  
    public static void main(String[] args) {  
        Path p1 = Paths.get("C:/blueJ/data/Notes/backup.dat");  
        Path p2 = Paths.get("C:/blueJ/data");  
        Path p3 = p2.relativize(p1);  
        System.out.println(p3);  
    } //end of method main  
} //end of class RelativizePaths
```

- The output of this code will be:

```
Notes\backup.dat
```

# 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.



# 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

