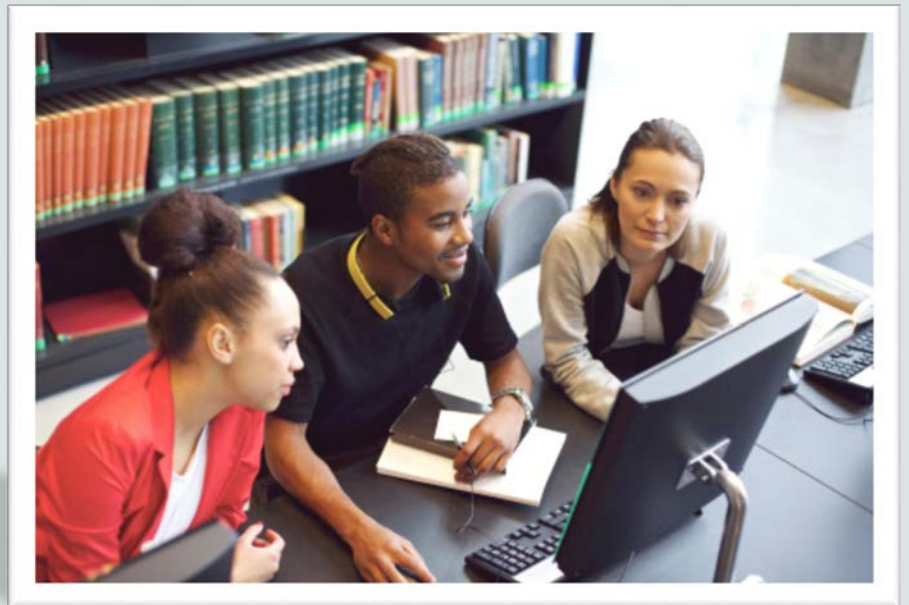




Java Programming

8-2

Input and Output Fundamentals



Objectives

This lesson covers the following topics:

- Use streams to read and write files
- Read and write objects by using serialization

Files Class Checks for File Existence

- The Files class checks to see if files exist, or do not exist.
- By default, symbolic links are not followed.
- If the `!exists()` method and `notExists()` method are both false, it means that they cannot determine whether the file exists.

```
public class FilesCheckDemo {  
    public static void main(String[] args) {  
        Path p1 = Paths.get("C:/BlueJ/NIO2");  
        boolean path_exists = Files.exists(p1);  
        System.out.println("Exists? " + path_exists);  
    } // end of main  
} //end of class FilesCheckDemo
```

- This will return a value of false as the path doesn't exist.



Files Class Checks File Properties

The Files class checks to see if files are:

- Readable
- Writeable
- Executable
- Hidden
- The same

Files Class Checks File Properties

- The Files class provides these static methods for checking file properties and duplication:

```
Files.isReadable(Path p);  
Files.isWritable(Path p);  
Files.isExecutable(Path p);  
Files.isHidden(Path p);  
Files.isSameFile(Path p1, Path p2);
```

- Sample output would be:

```
System.out.println(Files.isReadable(wof));           true  
System.out.println(Files.isWritable(wof));           true  
System.out.println(Files.isExecutable(wof));         true  
System.out.println(Files.isHidden(wof));              false  
System.out.println(Files.isSameFile(wof, buf));      false
```

Creating Files and Directories

- Create files with one of the following methods:

```
Files.createFile(Path p);  
Files.createDirectory(Path p);
```

- Create multiple levels of directories with this method:

```
Files.createDirectories(Path p);
```

Creating Files and Directories Example

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class FilesDemo {
    public static void main(String[] args) {
        Path p = Paths.get("C:/BlueJ/scores");
        Path p2 = Paths.get("Highscores.txt");
        Path p3 = p.resolve(p2);
        //used to test the value of p3
        System.out.println("The contents of p3 : " + p3.toString());
        try {
            if(Files.notExists(p))
                Files.createDirectories(p);
            //endif
            if(Files.notExists(p3))
                Files.createFile(p3);
            //endif
        } //end try
        catch (IOException x) {
            System.err.println(x);
        } //end catch
    } //end of main
} //end of class FilesDemo
```

Use resolve to add p2 to p as it does not already exist in that path

If the directory does not already exist create it using the Path p

If the file does not already exist create it using the Path p3

Deleting Files and Directories

- Delete files, directories, or links with this method.
- Throws a `NoSuchFileException`, `DirectoryNotEmptyException`, or `IOException` when the file is not found or the directory holds files or directories.

```
Files.delete(Path p);
```

- Delete files, directories, or links without exceptions by using this method.

```
Files.deleteIfExists(Path p);
```

Deleting Files and Directories Example

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class FilesDeleteDemo {
    public static void main(String[] args) {
        Path p = Paths.get("C:/BlueJ/scores");
        Path p2 = Paths.get("Highscores.txt");
        Path p3 = p.resolve(p2);
        //The following code will delete the file if it exists.
        try {
            if(Files.exists(p3)){
                Files.deleteIfExists(p3);
                System.out.println(p3.toString()+ " deleted!");
            }
            else
                System.out.println(p3.toString()+ " not found!");
        } //endif
    } //end try
    catch (IOException x) {
        System.err.println(x);
    } //end catch
} // end of main
} //end of class FilesDeleteDemo
```

Copying and Moving Files and Directories

- Import the `java.nio.file.StandardCopyOption.*` package when you want the ability to copy or move files and directories.
- Copy or move files or directories with these methods:

```
Files.copy(Path p, CopyOption ...);  
Files.move(Path p, CopyOption ...);
```


- An example would be:

```
Files.copy(source, target, REPLACE_EXISTING, NOFOLLOW_LINKS);
```

StandardCopyOption and LinkOption Enums

The StandardCopyOption and LinkOption enums are:

- REPLACE_EXISTING: Works with existing file or directory.
- COPY_ATTRIBUTES: Copies related attributes.
- NOFOLLOW_LINKS: Disables following symbolic links.



StandardCopyOption and LinkOption Enums Format

- The options must be prefaced with StandardCopyOption or LinkOption.
- Examples:
 - StandardCopyOption.REPLACE_EXISTING
 - StandardCopyOption.COPY_ATTRIBUTES
 - StandardCopyOption.NOFOLLOW_LINKS
 - LinkOption.REPLACE_EXISTING
 - LinkOption.COPY_ATTRIBUTES
 - LinkOption.NOFOLLOW_LINKS

File example

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;

public class FilesDemo {
    public static void main(String[] args) {
        //create path variables
        Path p = Paths.get("C:/BlueJ");
        Path p1 = Paths.get("scores");
        Path p2 = Paths.get("backup");
        Path p3 = Paths.get("Highscores.txt");
        //create path for the working directory
        Path woD = p.resolve(p1);
        //create path for the working file
        Path woF = p.resolve(p1.resolve(p3));
        //create path for the backup directory
        Path buD = p.resolve(p2);
        //create path for the backup file
        Path buF = p.resolve(p2.resolve(p3));
```

Creates paths for the working directory/file

Creates paths for the backup directory/file

Code continues on next slide...

File example

... code continued from previous slide

```
try {
    if(Files.exists(woF)){
        if(Files.notExists(buD)){
            Files.createDirectories(buD);
        }//endif
        Files.copy(woF, buF, StandardCopyOption.REPLACE_EXISTING,
StandardCopyOption.COPY_ATTRIBUTES);
    } //endif
    if(Files.notExists(woD))
        Files.createDirectories(woD);
    //endif
    if(Files.notExists(woF))
        Files.createFile(woF);
    //endif
} //end try
catch (IOException x) {
    System.err.println(x);
} //end catch
} // end of main
} //end of class FilesDemo
```

Existing file is copied to the backup directory

If the required directory/file does not exist then they are created.



File Permissions

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.

File Permissions

```
Path p1 = Paths.get("NIO2");  
Path p2 = Paths.get("Projects");  
  
// Output value of join between two paths.cd  
System.out.println("p1.realativize(p1) [" +  
                    p1.relativeize(p2).toString() + "]);
```

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



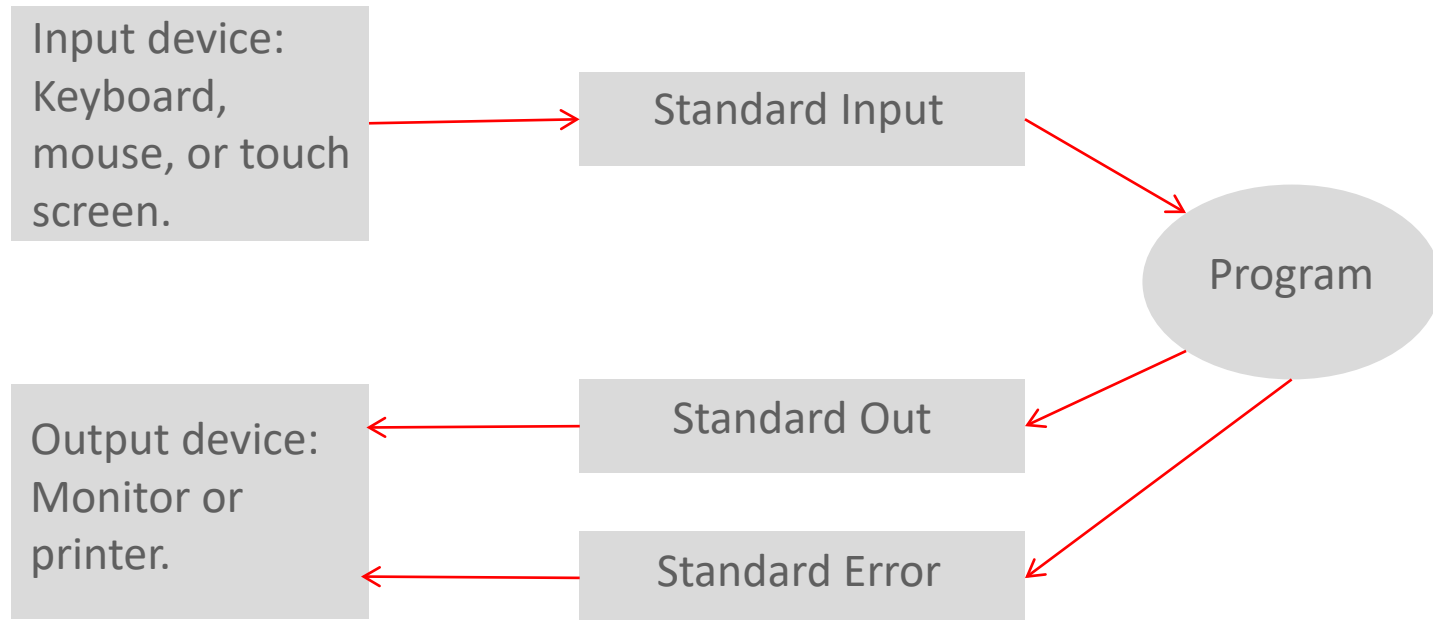
File Permissions and Operating Systems

- The file permissions differ from operating system to operating system.
- Linux Permissions
 - read/write/execute
- Windows Permissions
 - Full control/Modify/Read and execute/Read/Write

Input and Output Stream Basics

- Standard programming has three basic streams:
 - Standard in (stdin), input to programs
 - Standard out (stdout), output from programs
 - Standard error (stderr), error messages from programs
- Java has three basic streams:
 - System.in an InputStream (like standard in)
 - System.out a PrintStream (like standard out)
 - System.err a PrintStream (like standard error)

Input and Output Stream Diagram



Java Stream Basics

- Java provides specialized stream classes:
 - Input Streams
 - Output Streams
- Java stream libraries:
 - Simplify deployment
 - Handle most types of input and output

Reading an Input Stream by Character

Java reads an input stream:

- Character-by-character - Line-by-line

```
private static String readEntry() {  
    try {  
        int c;  
        StringBuffer buffer = new StringBuffer();  
        c = System.in.read();  
        while (c != '\n' && c != -1) {  
            buffer.append((char)c);  
            c = System.in.read();  
        } //endwhile  
        return buffer.toString().trim();  
    } //end try  
  
    catch (IOException e) {  
        return null;  
    } //endcatch  
} //end method readEntry
```

This reads the input stream character-by-character.

Reading an Input Stream by Line

- Line-by-line reads require a `BufferedReader`, which is a specialization of an `IO Reader` class.
- `System.in` provides a static method to create an instance of an `InputStream` class.

```
private static String readLine() {  
    String line = "";  
    InputStreamReader isr = new InputStreamReader(System.in);  
    BufferedReader in = new BufferedReader(isr);  
    try {  
        line = in.readLine();  
    } //end try  
    catch (IOException e) {} //end catch  
  
    return line;  
} //end method readLine
```

This is a static call to construct an input stream from the command-line.

Create a `BufferedReader` stream that provides the `readLine()` method.

This reads the input stream line-by-line.

Reading an Input from file

- Reading from a file since the introduction of Java 7 is a relatively straightforward process.
- The try with resources method includes an auto close that closes the file when the operation is complete.

```
private static String ReadFile() {  
    try(BufferedReader br = new BufferedReader(new FileReader("file.txt"))){  
        StringBuilder fileContents = new StringBuilder();  
        String line = br.readLine();  
  
        while (line != null) {  
            fileContents.append(line);  
            fileContents.append(System.lineSeparator());  
            line = br.readLine();  
        }//end while  
  
        String fileComplete = fileContents.toString();  
        System.out.println(fileComplete);  
    }//end try  
}//end ReadFile
```

Create a BufferedReader stream that provides the readLine() method.

This reads the input stream line-by-line and appends it to the String. Uses the line separator that corresponds to the current operating system.

Writing an Output Stream

- Output to the console is typically managed by calling the static `System.out`, which is a `PrintStream`.
- Other alternatives require combining streams.

```
public static void main(String[] args) {  
    StringBuffer sb = new StringBuffer();  
    char[] input;  
    System.out.print("Enter a string: ");  
    input = readEntry();  
    for (int i = 0; i < input.length; i++)  
    {  
        if (input[i] != '\n' && input[i] != '\0')  
            sb.append(input[i]);  
        //endif  
    } //end for  
    System.out.println(sb);  
} //end method main
```

Uses a modified `readEntry()` method that returns an array of `char`, which are then appended to a `StringBuffer` until the end of the output is found.

`System.out` is `PrintStream` that can be accessed by a static call.

Writing Output to File

- Output to a file is managed through the `PrintWriter` and `FileWriter`.
- A `println` statement is used to write the contents to the file.
- If you have created a `toString()` method to override the default output you can control the format of the text in the file.

```
public void WriteFile(ClassName objName) throws IOException{
    PrintWriter writer = new PrintWriter(new BufferedWriter(new
                                                FileWriter(filepath)));

    writer.println(objName);
    writer.close();
}
```

Writing Output to File

- If you want to append to the file instead of overwriting add the optional true parameter to the FileWriter call.

```
public void WriteFile(EmployeeInfo objName) throws IOException{
    PrintWriter writer = new PrintWriter(new BufferedWriter(new
                                                FileWriter(filepath, true)));

    writer.println(objName);
    writer.close();
}
```

- You can also write individual pieces of information by calling the get methods of the class.

```
public void WriteFile() throws IOException{
    PrintWriter writer = new PrintWriter(new BufferedWriter(new
                                                FileWriter(filepath, true)));

    writer.println(classname.methodname);
    writer.close();
}
```



Object Serialization

- Object serialization is the process of encoding objects as a byte stream, transmitting them, and reconstructing objects by decoding their byte stream.
- Encoding an object into a stream is serialization.
- Decoding a stream into an object is deserialization.
- Serialization is the standard method for Java beans.
- Serialized classes implement the Serializable interface.

Use Serialization Wisely

Use serialization wisely because serialized classes:

- Are less flexible to change.
- May have more likelihood of bugs and security vulnerabilities.
- Are more complex to test.

Serializing and Deserializing

- This serializes a file into an object.

```
public static void serialize( String outFile, Object serializableObject)
throws IOException {
    FileOutputStream fos = new FileOutputStream(outFile);
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(serializableObject);
} //end method serialize
```

- This deserializes an object.

```
public static Object deSerialize(String serializedObject) throws
    FileNotFoundException,
    IOException,
    ClassNotFoundException {
    FileInputStream fis = new FileInputStream(serializedObject);
    ObjectInputStream ois = new ObjectInputStream(fis);
    return ois.readObject();
} //end method deSerialize
```

Testing Serializing and Deserializing

- The main() method tests serialization by:
 - Serializing an object.
 - Deserializing an object.
 - Printing the transferred contents of the first object.
- The code for testing serialization in this way begins on the next slide.

Testing Serializing and Deserializing Example

- Create the following Course class in a project named serialDeserial:

```
public class Course implements java.io.Serializable {  
    public String name;  
    public String type;  
    public String courseCode;  
    public int passingScore;  
} //end class Course
```

- This will be used to create the object that you will serialize and deserialize.
- For a class to be serialized successfully it must implement the java.io.Serializable interface.

Testing Serializing and Deserializing

Create the following DemoSerialization class in your serialDeserial project:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class DemoSerialization {
    public static void main(String [] args)
    {
        Course c = new Course();//create a new object(c)
        serializeData(c);//pass the object to serializeData
        c = deSerializeData(c);//pass the object to deSerializeData
        if(c!=null)
            displayData(c);
        //endif
    }//end method main
```

Code continues on next slide...

Testing Serializing and Deserializing

... code continued from previous slide

```
public static void serializeData(Course c){
    //assign values to the Course class attributes
    c.name = "Java Programming";
    c.type = "Programming";
    c.courseCode = "JPL2";
    c.passingScore = 60;
    try
    {
        //try writing to the file
        FileOutputStream fileOut = new FileOutputStream("C:/BlueJ/details.ser");
        ObjectOutputStream out = new ObjectOutputStream(fileOut);
        out.writeObject(c);
        out.close();
        fileOut.close();
        System.out.printf("Serialized data is saved in C:/BlueJ/details.ser");
    }
    //end try
    catch(IOException i)
    {
        i.printStackTrace();
    }
    //end catch
}
//end method serializeData
```

Code continues on next slide...

Testing Serializing and Deserializing

... code continued from previous slide

```
public static Course deSerializeData(Course c){
    try
    { //try reading the file
        FileInputStream fileIn = new FileInputStream("C:/BlueJ/details.ser");
        ObjectInputStream in = new ObjectInputStream(fileIn);
        c = (Course) in.readObject();
        in.close();
        fileIn.close();
        return c;
    } //end try
    catch (IOException i)
    { //catch any IO exception error that is thrown
        i.printStackTrace();
        return null;
    } //end catch
    catch (ClassNotFoundException e)
    { //catch any error where the class is not found
        System.out.println("Course class not found");
        return null;
    } //end catch
} //end method deSerializeData
```

Code continues on next slide...

Testing Serializing and Deserializing

... code continued from previous slide

```
public static void displayData(Course c){
    //display the contents of the class to screen
    System.out.println("Deserialized Course Details...");
    System.out.println("Name: " + c.name);
    System.out.println("Type: " + c.type);
    System.out.println("Code: " + c.courseCode);
    System.out.println("Pass Score: " + c.passingScore);
} //end method displayData

} //end class DemoSerialization
```

End of code.

Terminology

Key terms used in this lesson included:

- Deserialization
- File Name
- Tree
- Resolve path
- Output Streams
- Standard input
- Standard output
- Standard error

Summary

In this lesson, you should have learned how to:

- Use streams to read and write files
- Read and write objects by using serialization

