ORACLE

Academy

Java Programming

5-2

Input and Output Fundamentals

ORACLE
Academy

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

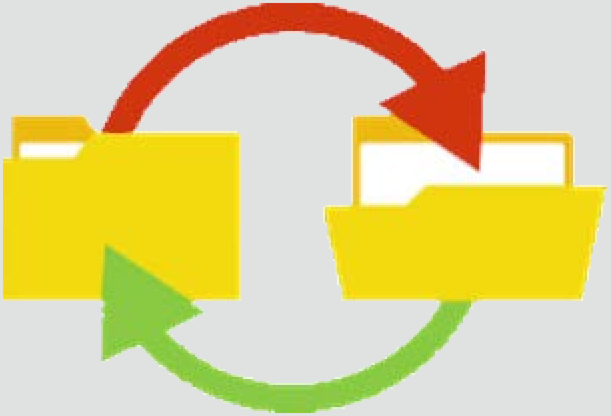
- The Files class checks to see if files exist, or do not exist
- By default, symbolic links are not followed
- If the !exists() method and notExists() method are both false, it means that they cannot determine whether the file exists

```java
public class FilesCheckDemo {
   public static void main(String[] args) {
      Path path = Paths.get("C:/JavaProgramming/IO2");
      boolean path_exists = Files.exists(path);
      System.out.println("Exists? " + path_exists);
   }// end main method
}//end 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



The Files class is not only useful for discovering if a file exists but also for identifying the state of the files operation.

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(absPath));      true
 System.out.println(Files.isWritable(absPath));      true
 System.out.println(Files.isExecutable(absPath));    true
 System.out.println(Files.isHidden(absPath));        false
 System.out.println(Files.isSameFile(absPath, dirPath));
                                                     false
```

All of these methods return a Boolean value.

Creating Files and Directories

- Create a file at a given path.

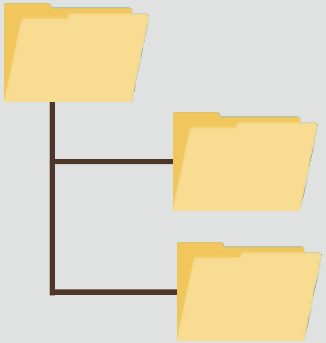
```
Files.createFile(Path p);
```

- Create a single directory at a given path.

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

- Create multiple levels of directories.

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



Creating Files and Directories Example

1. Create the following project and class:

```
package filesdemo;

public class FilesDemo {
   public static void main(String[] args) throws IOException {
   }// end main method

   static Path checkFiles(Path dirPath, Path filePath) {
   }//end method checkFiles

   static void displayFileStatus(Path users, Path settings)
throws IOException {

   }//end method displayFileStatus
}//end of class FilesDemo
```

Creating Files and Directories Example

2. Update main() to create the following paths:

```
public class FilesDemo {
   public static void main(String[] args) throws IOException {
      Path dirPath = Paths.get("C:/JavaProgramming/gameData");
      Path usersfilePath = Paths.get("Highscores.txt");
      Path settingsfilePath = Paths.get("Settings.txt");
   }// end of main
```

- dirPath stores the path for the directory structure for a game that requires permanent storage for its gameData files
- usersFilePath stores the path to the users high scores file that would be used in the game to display the highest scores
- settingsFilePath stores the path to the users settings file that would be used to load the player settings into a game

Creating Files and Directories Example

3. Update the checkFiles() method:

```
static Path checkFiles(Path dirPath, Path filePath) {
   Path absPath = dirPath.resolve(filePath);
   try {
   }//end try
   catch (IOException x) {
      System.err.println(x);
      return null;
   }//end catch
   return absPath;
}//end method checkFiles
```

Use resolve to add the directory path to the file path.

- Resolve a path based on the directory and file paths provided
- Implement a try catch that will handle any IO errors, it will display an error message to screen and return null
- If no errors have occurred then return the absolute path

Creating Files and Directories Example

4. Update the checkFiles() method to include:

```
Path absPath = dirPath.resolve(filePath);
try {
   if(Files.notExists(dirPath))
      Files.createDirectories(dirPath);
   //endif
   if(Files.notExists(absPath))
      Files.createFile(absPath);
  //endif
}//end try
catch (IOException x) {
```

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

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

- If the path does not exist create directory structure by using the createDirectories() method
- If the file does not exist on that path then use the createFile() method to create the file

Creating Files and Directories Example

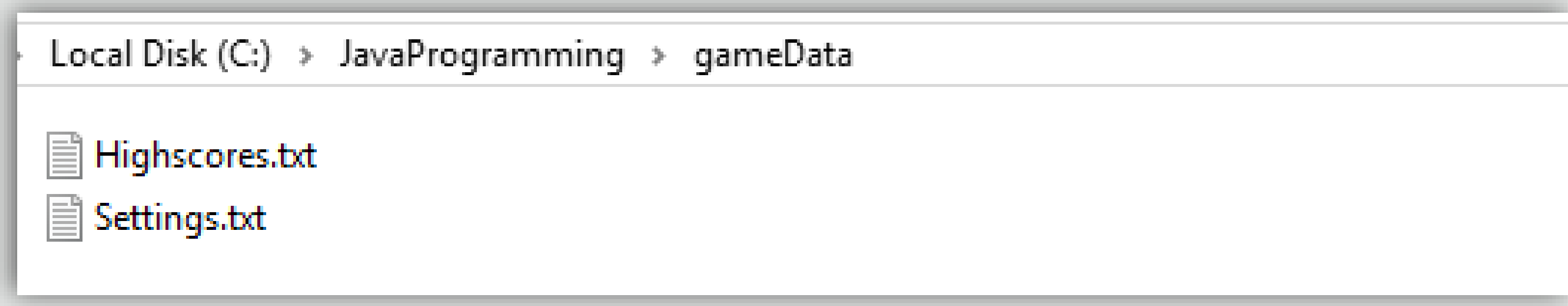
5. Update the main method to include:

```
public static void main(String[] args) throws IOException {
   Path dirPath = Paths.get("C:/JavaProgramming/gameData");
   Path usersfilePath = Paths.get("Highscores.txt");
   Path settingsfilePath = Paths.get("Settings.txt");
   Path users, settings;
   users = checkFiles(dirPath, usersfilePath);
   settings = checkFiles(dirPath, settingsfilePath);
}//end main method
```

- Create two new paths (users, settings) that will store the return values from the checkFiles() method
- Call checkFiles() passing the directory and highscores path
- Call checkFiles() passing the directory and settings path

Creating Files and DirectoriesExample

6. Run the program and check that the correct directory structure has been created at the path location:



Local Disk (C:) › JavaProgramming › gameData

Highscores.txt

Settings.txt

7. **TASK**: Update the code in the displayFileStatus() method to use the code from slide 6 to display the users file properties

8. **TASK**: Update main to only call the display method if the users path is not null

Creating Files and Directories Example

- Your completed code should look like this:

```
   settings = checkFiles(dirPath, settingsfilePath);
   if(users!=null)
      displayFileStatus(users, settings);
   //endif
}// end main method

static void displayFileStatus(Path users, Path settings) throws
                                                     IOException {
   System.out.println("Readable  : " + Files.isReadable(users));
   System.out.println("Writeable : " + Files.isWritable(users));
   System.out.println("Executable: " + Files.isExecutable(users));
   System.out.println("Hidden     : " + Files.isHidden(users));
   System.out.println("Same files: " + Files.isSameFile(users,
                                       settings));
}//end method displayFileStatus
```

Deleting Files and Directories

- With all file operations there is a potential for errors being thrown, if the file doesn’t exist or a directory is not empty
- Delete files, directories, or links with these methods

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

- When the file is not found or the directory holds files or directories it will throw:
  - NoSuchFileException
  - DirectoryNotEmptyException
  - IOException

Creating Files and Directories Example

9. Add the following method under the main() method in the FilesDemo class:

```
   static void deleteFile(Path filePath) {
      //This will delete the file/directory if it exists.
      try {
         if(Files.exists(filePath)){
            Files.delete(filePath);
            System.out.println(filePath.toString()+ " deleted!");
         }
         else
            System.out.println(filePath.toString()+ " not found!");
         //endif
      }//end try
      catch (IOException x) {
         System.err.println(x);
     }//end catch
}//end method deleteFile
```

Creating Files and Directories Example

10. Add the following method under the main() method in the FilesDemo class:

```
   static void deleteFile(Path filePath) {
      //This will delete the file/directory if it exists.
      try {
         if(Files.exists(filePath)){
            Files.delete(filePath);
            System.out.println(filePath.toString()+ " deleted!");
         }
         else
            System.out.println(filePath.toString()+ " not found!");
         //endif
      }//end try
      catch (IOException x) {
         System.err.println(x);
     }//end catch
}//end method deleteFile
```

If the file exists then delete it otherwise display not found

Catch any IO exception errors that occur.

Creating Files and Directories Example

11. Add a method call to the bottom of main that will call the deleteFiles() method passing the dirPath:

```
        //endif
        deleteFile(dirPath);
    }// end main method

    static void deleteFile(Path filePath) {
```

12. Run the code and identify the error reported!
13. **TASK**: Use a catch statement to display an appropriate error message that will deal with this error

Creating Files and Directories Example

- The following code will handle a method call that attempts to delete a non-empty directory

```
    }//end try
    catch(DirectoryNotEmptyException e) {
       System.err.println("The directory is not empty");
    }//end catch
    catch (IOException x) {
```

14. Change the argument in the deleteFiles() method call to pass the settings path instead

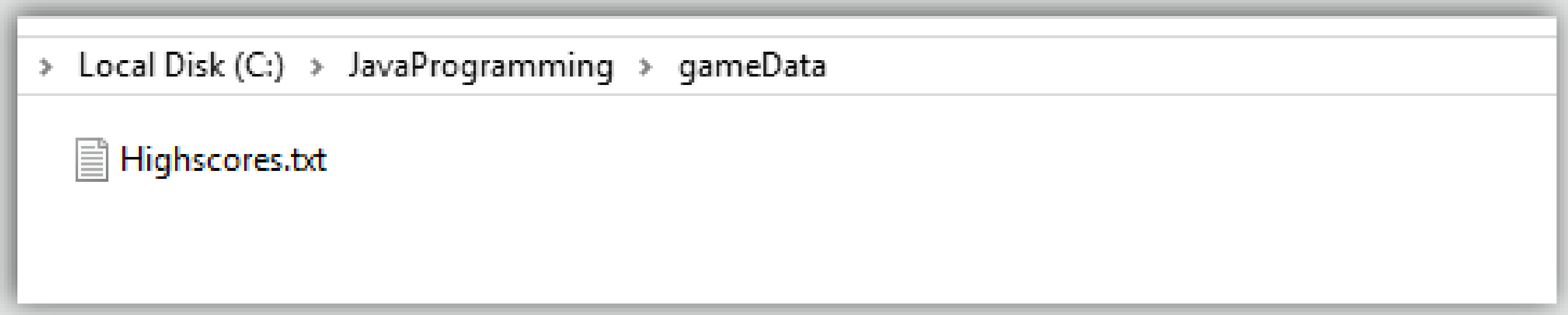
```
      //endif
      deleteFile(settings);
   }// end main method

   static void deleteFile(Path filePath) {
```

ORACLE
Academy

Creating Files and Directories Example

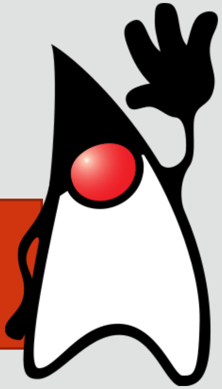
15. **TASK**: What message was displayed in the console?
16. **TASK**: Check the folder structure to ensure that the operation happened



› Local Disk (C:) › JavaProgramming › gameData

Highscores.txt

It's important to always add the correct catch statements when handling files so that the user knows what has gone wrong.



Copying and Moving Files and Directories

- Import the java.nio.file.StandardCopyOption.* package to copy or move files and directories.

```
import java.nio.file.StandardCopyOption.*;
```

- Copy or move files or directories with these methods:

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

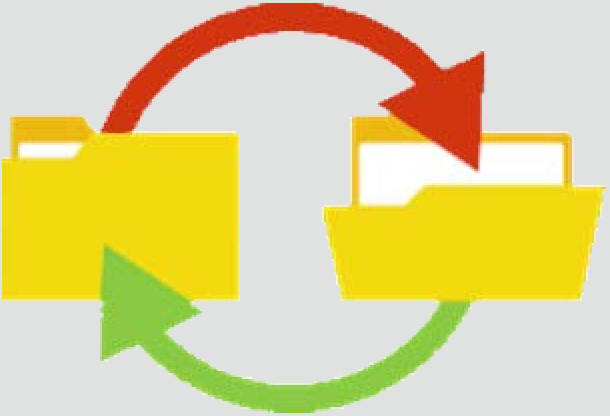
- An example would be:

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

Copying files has to be done with care but Java provides many options to make it easier for you.

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

Create the following code!

```
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 FilesCopyDemo {
   public static void main(String[] args) {
      //create path variables
      Path p = Paths.get("C:/JavaProgramming/gameData");
      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…

ORACLE Academy

File example

Use the debugging tools to explore the code!

... 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.

Academy

File Permissions

- The relativize() method constructs a path from one location to another:
  - 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 differ from operating system to operating system so always consider this when coding files in your application.

.relativize() Example

- This example will return the relative path between two relative paths in the same directory tree

```
Path path1 = Paths.get("JavaProgramming/gameData/backup");
Path path2 = Paths.get("JavaProgramming/IO/Logs");

// Output value of path between two relative addresses
System.out.println("The relative path from \"" + path1 + "\" to
\"" + path2 + "\" is [" + path1.relativize(path2).toString() +
"]");
```

- Will produce the following output:

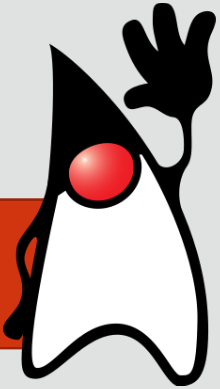
```
The relative path from "JavaProgramming\gameData\backup"
to "JavaProgramming\IO\Logs" is [..\..\IO\Logs]
```

Remember you can only use the relativize() method when the two paths are in the same directory structure.

File Permissions and Operating Systems

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

File permissions allow you to control access and also control what operations can be carried out on the files.



File Permissions and Operating Systems

- Windows Permissions:
  - Full control
    - View the contents of a file or folder, change existing files and folders, create new files and folders and run programs in a folder
  - Modify
    - Can change existing files and folders, but cannot create new ones
  - Read and execute
    - Can see the contents of existing files and folders and can run programs in a folder
  - Read
    - Can see the contents of a folder and open files and folders
  - Write
    - Can create new files and folders, make changes to existing files and folders

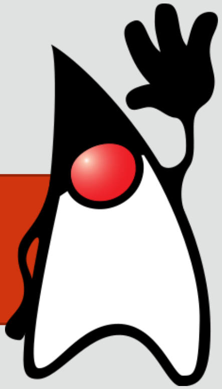
File Permissions and Operating Systems

- Linux Permissions:
  - Read
    - Can view the contents of the file
  - Write
    - Can change the contents of the file
  - Execute
    - Can execute or run the file if it is a program or script

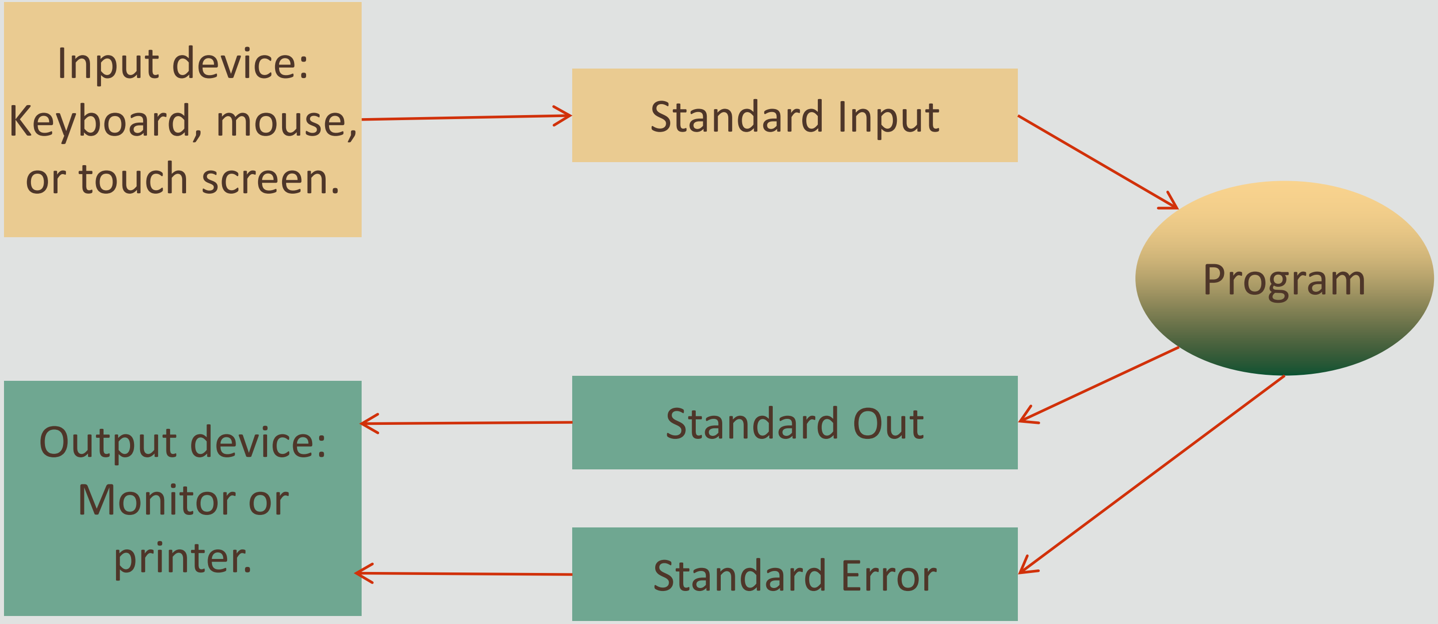
Input and Output Stream Basics

- 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)

This deals with both input and output as well as any errors that may occur during file operations.



Input and Output Stream Diagram


Input device: Keyboard, mouse, or touch screen.
Standard Input
Program
Standard Out
Standard Error
Output device: Monitor or printer.

All input goes through the standard input stream regardless of what generates the input.
Both the output and error streams are sent to the output device.

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

- This code reads in a character at a time until it reaches the new line character (\n)

```
    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) {
      System.err.println(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.

Closing Resources Prior to Java 7

- Using a resource previous to Java 7 required the manual closing of the resource after its use
  - This was normally executed with the use of a try-catch-finally block
  - The resource had to be declared outside of the try-catch-finally block so that it was accessible within both the try and finally sections
  - A resources implements the AutoCloseable interface and includes the Scanner, BufferedReader, PrintStream etc

Closing Resources Prior to Java 7

- This code closes the resource inside the finally block.

```
static int getAge() {
   int age=-1;
   Scanner in = new Scanner(System.in);
   try {
      System.out.print("Please enter your age: ");
      age = in.nextInt();
   }//end try
   catch(Exception e) {
      System.err.println(e);
   }//end catch
   finally {
      if (in != null)
         in.close();
   }//end finally
   return age;
}//end method getAge
```

The resource is declared outside of the try statement.

The resource is closed in the finally block

Closing Resources Java 7 and beyond

- Using resources since the introduction of Java 7 is a relatively straightforward process
  - The try with resources method includes an auto close to close the resource when the operation is complete
  - The new try with resources block replaces the previously used try-catch-finally block
  - The resources must be declared and initialized inside parenthesis for the try statement and implement the AutoCloseable interface
  - Multiple resources can be declared in a try-with-resources block

Closing Resources Java 7 and beyond

- This code closes the resource inside the try statement

```
static int getAge() {
   int age=-1;
   try (Scanner in = new Scanner(System.in);)
   {
      System.out.print("Please enter your age: ");
      age = in.nextInt();
   }//end try
   catch(Exception e) {
      System.err.println(e);
   }//end catch
   return age;
}//end method getAge
```

The resource is declared inside the parenthesis of the try statement.

The try-with-resources statement makes sure that all declared resources are closed at the end of the statement, ensuring the proper release of all close-able resources.

Closing Resources Java 7 and beyond

- Reading an Input from file

```
private static String readFile() {
   try(BufferedReader br = new BufferedReader
                 (new FileReader("C:/JavaProgramming/employees.txt"))){
      StringBuilder fileContents = new StringBuilder();
      String line = br.readLine();
      while (line != null) {
         fileContents.append(line);
         fileContents.append(System.lineSeparator());
         line = br.readLine();
      }//end while
      return fileContents.toString();
   }//end try
   catch (IOException e) {
      System.err.println(e);
   }//end catch
      return null;
}//end ReadFile
```

The resource is declared inside the parenthesis of the try statement.

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 resource
- 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 a 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 a toString() method was created to override the default output the format of the text in the file can be controlled

```
public void writeFile(ClassName objName) throws IOException{
   PrintWriter writer = new PrintWriter(new BufferedWriter
                                        (new FileWriter(filepath)));
   writer.println(objName);
   writer.close();
}//end method writeFile
```

Writing Output to File

- The previous example overwrites any content in the file
- To append to the file (save the new content to the end of the existing data) instead of overwriting then add the optional true parameter
- (FileWriter(filepath, true)) 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();
}//end method writeFile
```

Writing Output to File

- Individual pieces of information can be written by calling the get methods of the class

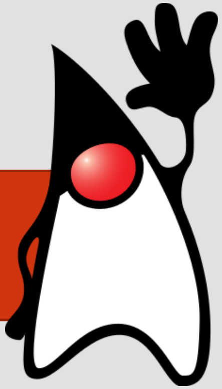
```
public void WriteFile(User usr) throws IOException{
   Path path = Paths.get("C:/JavaProgramming/usersNames.txt");
   PrintWriter writer = new PrintWriter(new BufferedWriter(new
                                FileWriter(path.toString(), true)));
    writer.println(usr.getName());
    writer.close();
}//end method writeFile
```

- If you are using a path field to store the filepath then you will need to use the path.toString() method to enable the FileWriter to identify the path
- The throws IOException should be used for the situations where the file cannot be created

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

This implementation is generally robust, tested, and architecture-independent.



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
  - For a class to be serialized successfully it must implement the java.io.Serializable interface

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

1. Create this class in a package called serialDeserial

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

2. Add a constructor for the class under the fields

```
public Course(String name, String type, String courseCode,
              int passingScore) {
   this.name = name;
   this.type = type;
   this.courseCode = courseCode;
   this.passingScore = passingScore;
}//end constructor
```

Testing Serializing and Deserializing

3. Add getters and setters for the instance fields

```
    public String getName() {
       return name;
    }//end method getName

    public void setName(String name) {
       this.name = name;
    }//end method setName

    public String getType() {
       return type;
    }//end method getType

    public void setType(String type) {
       this.type = type;
    }//end method setType
```

Continued on next slide...

Testing Serializing and Deserializing

3. Add getters and setters for the instance fields

```
public String getCourseCode() {
   return courseCode;
}//end method getCourseCode

public void setCourseCode(String courseCode) {
   this.courseCode = courseCode;
}//end method setCourseCode

public int getPassingScore() {
   return passingScore;
}//end method getPassingScore

public void setPassingScore(int passingScore) {
   this.passingScore = passingScore;
}//end method setPassingScore
```

Testing Serializing and Deserializing

4. A serialVersionUID variable is used by Java's object serialization API to determine if a deserialized object was serialized (written) with the same version of the class it is now attempting to deserialize in to
5. Any changes to the file would create a different object
6. Add a default final UID field to the class

```
public class Course implements java.io.Serializable {
   private static final long serialVersionUID = 1L;
   private String name;
   private String type;
   private String courseCode;
   private int passingScore;
}//end class Course
```

Testing Serializing and Deserializing

7. Create a **SerializationDemo** class that contains a main method that creates a Course object

```
public class SerializationDemo {
   public static void main(String[] args) {
      Course course = new Course("Java Programming", "Oracle",
                                 "JP", 60);
  }//end method main
}//end class SerializationDemo
```

- The main() method will test serialization by:
  - Creating a new Course object
  - Serializing the Course object
  - Deserializing the Course object
  - Printing the transferred contents of the Course object

Testing Serializing and Deserializing

8. Add the following methods to the driver class:

```
public class SerializationDemo {
   public static void main(String[] args) {
      Course course = new Course("Java Programming", "Oracle",
                                 "JP", 60);
   }//end method main

   static void serializeData(Course course, Path path){
   }//end method serializeData

   static Course deSerializeData(Path path){
   }//end method deSerializeData

   public static void displayData(Course course){
   }//end method displayData

}//end class SerializationDemo
```

Testing Serializing and Deserializing

9. Add the following try-with-resources code to the serializeData() method

```
static void serializeData(Course course){
   try()
   {
   }//end try
   catch(IOException e)
   {
     e.printStackTrace();
   }//end catch
}//end method serializeData
```

- The code to serialize the object to file will be written within the try statement
- The try-with-resources will ensure that all resources will be closed when the method is finished with them

Testing Serializing and Deserializing

10. Add the following code to the try statement

```
try(FileOutputStream fileOut = new FileOutputStream(path.toString());
    ObjectOutputStream objOut = new ObjectOutputStream(fileOut))
{//try writing to the file
   objOut.writeObject(course);
   System.out.println("Serialized data is saved in " + path.toString());
}//end try
```

- fileOut - creates the file output stream to the path specified
- objOut - creates the object output stream that allows the writing of objects
- objOut.writeObject – writes the object to the file specified through the ObjectOutputStream and the FileOutputStream

Testing Serializing and Deserializing

11. Add the following try-with-resources code to the deSerializeData() method

```
static Course deSerializeData(Path path){
      try()
      {//try reading the file
      }//end try
      catch(ClassNotFoundException e)
      {//catch any error where the class is not found
         System.out.println("Course class not found");
         return null;
      }//end catch
      catch(IOException i)
      {//catch any IO exception error that is thrown
         i.printStackTrace();
         return null;
      }//end catch
}//end method deSerializeData
```

Testing Serializing and Deserializing

12. Add the following code to the try statement

```
try(FileInputStream fileIn = new FileInputStream(path.toString());
    ObjectInputStream objIn = new ObjectInputStream(fileIn))
{//try reading the file
   Course course = (Course) objIn.readObject();
   return course;
}//end try
```

- fileIn - creates the file input stream to the path specified
- objIn - creates the object input stream that allows the reading of objects
- objIn.readObject - reads the object to the local Course object, the value is cast to a Course object as part of the read operation

Testing Serializing and Deserializing

13. Add the following code to the displayData method

```
public static void displayData(Course course){
   //display the contents of the class to screen
   System.out.println("Deserialized Course Details...");
   System.out.println("Name      : " + course.getName());
   System.out.println("Type      : " + course.getType());
   System.out.println("Code      : " + course.getCourseCode());
   System.out.println("Pass Score: " + course.getPassingScore());
}//end method displayData
```

- Remember, it was an object that was saved to and then read from file so to access its instance field values the getter() methods must be used

Testing Serializing and Deserializing

14. Update the code in the main method

```
public static void main(String[] args) {
   Course course = new Course("Java Programming", "Oracle", "JP", 60);
   Path path = Paths.get("C:/JavaProgramming/details.ser");
   serializeData(course, path);
   Course savedCourse = deSerializeData(path);
   if(course!=null)
      displayData(savedCourse);
   //endif
}//end method main
```

- path – stores the path to the file. If you do not have a JavaProgramming directory on the C drive create one
- serializeData – sends the object and path to save
- deSerializeData - Returns the Course object that was read from file, if an object is returned it is displayed to the console

Testing Serializing and Deserializing

15. Create a toString() method in the **Course** class to control the output of the object

```
   public String toString() {
      return    "Name      : " + this.name
            + "\nType      : " + this.type
            + "\nCode      : " + this.courseCode
            + "\nPass Score: " + this.passingScore;
   }//end method toString
}//end class Course
```

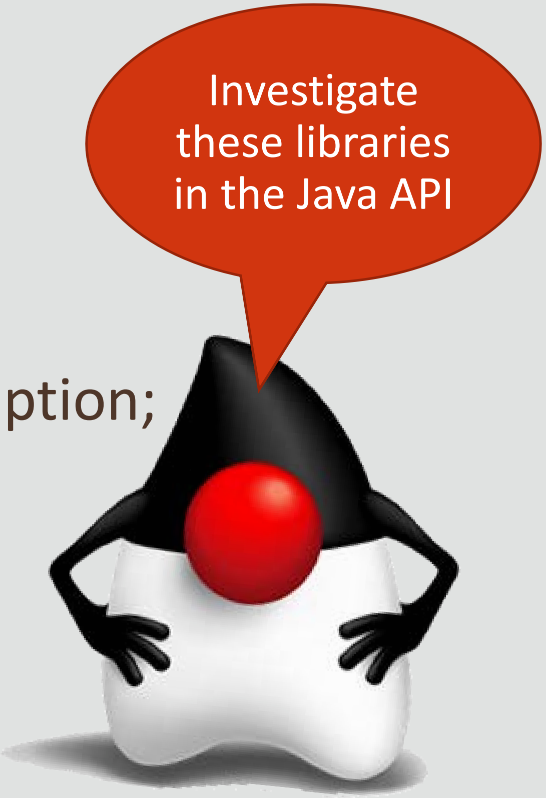
16. Update the displayData() method

```
   public static void displayData(Course course){
      //display the contents of the class to screen
      System.out.println("Deserialized Course Details...");
      System.out.println(course);
   }//end method displayData
```

ORACLE
Academy

Import libraries

- Throughout this section it has been required to import multiple Java Libraries:
  - import java.io.BufferedWriter;
  - import java.io.FileNotFoundException;
  - import java.io.FileWriter;
  - import java.io.IOException;
  - import java.io.PrintWriter;
  - import java.io.UnsupportedEncodingException;
  - import java.nio.file.Files;
  - import java.nio.file.Path;
  - import java.nio.file.Paths;



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

ORACLE

Academy