SD3.
Applications Programming.
File IO.

LIMERICK INSTITUTE
OF TECHNOLOGY
SCHOOL OF SCIENCE,
ENGINEERING & I.T.
Department of Information Technology

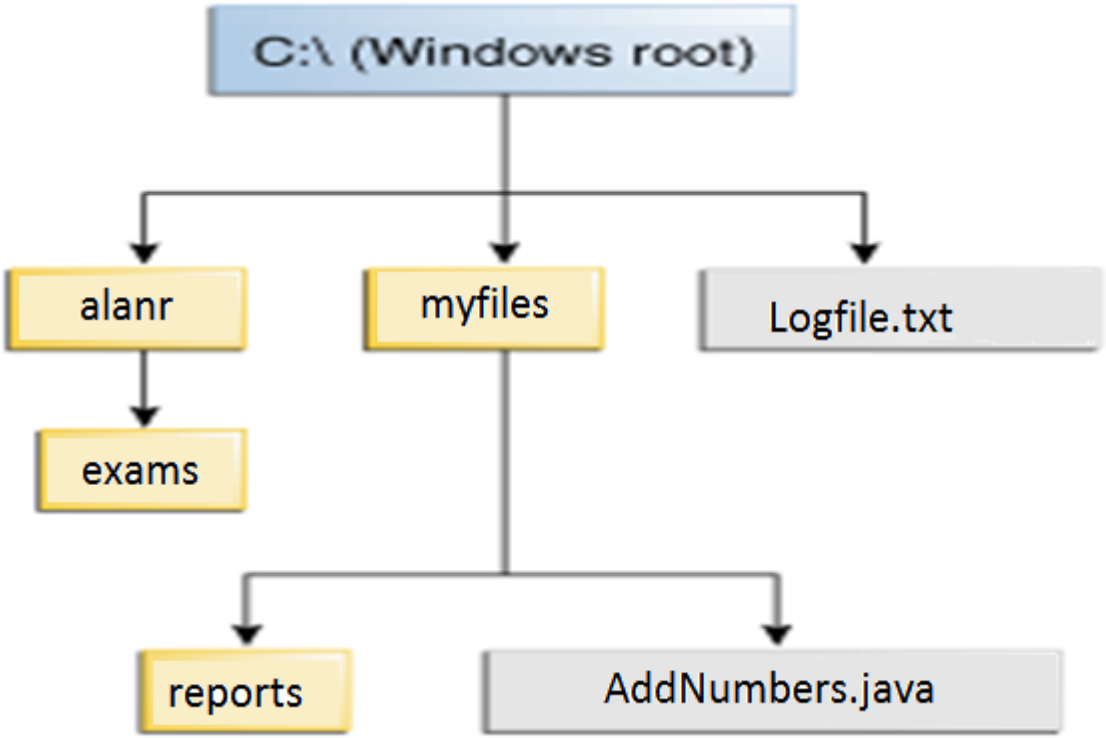
Java IO in Java 7

- There were many refinements included in Java SE 7.
    - Better error handling capabilities.
    - The ability to switch on Strings.
    - The ability to include underscores (_)in literals.
- There was also the inclusion of a package called java.nio.file.
- This package added classes and interfaces that make working with files and directories in Java much easier. First and foremost of these changes is the ability to copy or move files.
- These relatively simple tasks previously required programmers to write lots and lots of code.
- The main classes we will concern ourselves with are Path and Files.

What is a Path

- A file system stores and organizes files on some form of media, generally one or more hard drives, in such a way that they can be easily retrieved.
- Most file systems in use today store the files in a tree (or *hierarchical*) structure.
- At the top of the tree is one (or more) root nodes.
- Under the root node, there are files and directories (*folders* in Microsoft Windows).
- Each directory can contain files and subdirectories, which in turn can contain files and subdirectories, and so on, potentially to an almost limitless depth.
- The diagram on the following side shows a sample directory tree containing a single root node.
    - Microsoft Windows supports multiple root nodes. Each root node maps to a volume, such as C:\ or U:\.

What is a Path



What is a Path

- A file is identified by its path through the file system, beginning from the root node.
- For example, the AddNumbers.java file on the previous slide is described by the following notation in Microsoft Windows:

C:\myfiles\AddNumbers.java

- The character used to separate the directory names (also called the *delimiter*) is specific to the file system:
  - Many Linux based OS uses the forward slash (/), while Microsoft Windows uses the backslash slash (\).

What is a Path

- A path is either *relative* or *absolute*.
- An absolute path always contains the root element and the complete directory list required to locate the file.
- For example, C:\myfiles\AddNumbers.java is an absolute path.
- All of the information needed to locate the file is contained in the path string.
- A relative path needs to be combined with another path in order to access a file.
- For example, myfiles\AddNumbers.java is a relative path.
- Without more information, a program cannot reliably locate the myfiles\AddNumbers.java file in the file system.

The Path Class

- As its name implies, the Path class is a programmatic representation of a path in the file system.
- A Path object contains the file name and directory list used to construct the path, and is used to examine, locate, and manipulate files.

Creating a Path

- A Path instance contains the information used to specify the location of a file or directory.
- At the time it is defined, a Path is provided with a series of one or more names.
- You can easily create a Path object by using one of the following get methods from the Paths (note the plural) helper class:

```
Path p1 = Paths.get("C:\\alanr\\exams");

Path p2 = Paths.get(args[0]);
```

Verifying The Existence of a File or Directory

- To verify that the program can access a file as needed, you can use the isReadable(Path), isWritable(Path), and isExecutable(Path) methods.

- The following code snippet verifies that a particular file exists and that the program has the ability to execute the file.

```
21  Path file = Paths.get("C:\\alanr\\exams\\finalexam.docx");
22
23  if (Files.isRegularFile(file) & Files.isReadable(file) & Files.isExecutable(file))
24              System.out.println("Is A File, Is Readable and is Executable");
```

- When working with files its especially important to check the API for the Files class and Path interface to see what methods are available to you as a developer.

Deleting a File or Directory

- You can delete files, directories or links. With links, the link is deleted and not the target of the link. With directories, the directory must be empty, or the deletion fails.
- The Files class provides two deletion methods.
- The **delete(Path)** method deletes the file or throws an exception if the deletion fails. For example, if the file does not exist a NoSuchFileException is thrown.
- You can catch the exception to determine why the delete failed as follows:

Deleting a File or Directory

```
24  Path file = Paths.get("C:\\alanr\\exams\\finalexam.docx");
25
26  try {
27          Files.delete(file);
28  }//end try
29
30  catch (NoSuchFileException x) {
31           System.out.println("no such file or directory exists");
32  }//end catch
33
34  catch (DirectoryNotEmptyException x) {
35           System.out.println("Dir is not empty");
36  }//end catch
37
38  catch (IOException x) {
39         // File permission problems are caught here.
40         System.out.println(x);
41  }
```

- The **deleteIfExists(Path)** method also deletes the file, but if the file does not exist, no exception is thrown (this is called failing silently)

Copying a File or Directory

- You can copy a file or directory by using the **copy(Path, Path, CopyOption...)** method.
- The copy fails if the target file exists, unless the REPLACE_EXISTING option is specified.
- Directories can be copied. However, files inside the directory are not copied, so the new directory is empty even when the original directory contains files.
- The CopyOption argument can any combination of the following two options.

Copying a File or Directory

REPLACE_EXISTING – Performs the copy even when the target file already exists.

COPY_ATTRIBUTES – Copies the file attributes associated with the file to the target file. Attributes such as who created the file, whether it was hidden or not, what time it was created etc.

Copying a File or Directory

- The following shows how to use the copy method:

```
25  Path source= Paths.get("C:\\aFolder\\afile.txt");
26
27  Path target= Paths.get("C:\\aFolder\\anotherfile.txt");
28
29  Files.copy(source, target, REPLACE_EXISTING);
```

Alternatively….

```
26  Path source= Paths.get("C:\\aFolder\\afile.txt");
27
28  Path target= Paths.get("C:\\aFolder\\anotherfile.txt");
29
30  Files.copy(source, target, REPLACE_EXISTING, COPY_ATTRIBUTES);
```

Moving a File or Directory

- You can move a file or directory by using the **move(Path, Path, CopyOption...)** method.

- The move fails if the target file exists, unless the REPLACE_EXISTING option is specified.

- Empty directories can be moved.
- If the directory is not empty, the move is allowed when the directory can be moved without moving the contents of that directory.

- This method takes a number of options – although often you will only need to use the following one.

  REPLACE_EXISTING – Performs the move even when the target file already exists.

Moving a File or Directory

- The following shows how to use the move method:

```
26    Path source= Paths.get("C:\\aFolder\\afile.txt");
27
28    Path target= Paths.get("C:\\aFolder\\anotherfile.txt");
29
30    Files.move(source, target, REPLACE_EXISTING);
```

Reading, Writing and Creating Files

- To read, write and create files there are a great number of options available to you.
- For most situations, the utility methods **readAllBytes**, **readAllLines**, and the **write** methods will suffice.
    - They are designed for simple, common cases.

Reading, Writing and Creating Files

- Several of the methods in this section take an optional **OpenOptions** parameter. This parameter is optional and parameter can be one of the following:

  WRITE – Opens the file for write access.
  APPEND – Appends the new data to the end of the file. This option is used with the WRITE or CREATE options.
  CREATE_NEW – Creates a new file and throws an exception if the file already exists.
  CREATE – Opens the file if it exists or creates a new file if it does not.
  DELETE_ON_CLOSE – Deletes the file when the stream is closed. This option is useful for temporary files.

Reading, Writing and Creating Files

- If you have a small-ish file and you would like to read its entire contents in one pass, you can use the **readAllBytes(Path)** or **readAllLines(Path, Charset)** methods.

- These methods take care of most of the work for you but are not intended for handling large files.

- The code on the following slide shows how to use the **readAllLines** method to read the contents of a file into a program.

Reading from Files

```
13  Path pathToTargetFile = Paths.get("SampleFile.txt");
14
15  List<String> lines;
16  try {
17      lines = Files.readAllLines(pathToTargetFile, Charset.forName("UTF-8"));
18      String output = "";
19
20      System.out.println("The file \"" + pathToTargetFile.getFileName() + "\" has " + lines.size() + " lines\n");
21
22      for (String item : lines) {
23          output += item + "\n";
24      }//end for
25
26      System.out.println(output);
27
28  }//end try
29  catch (IOException ex) {
30      System.out.println("Something has gone wrong, \nEnsure that the file you are trying to view exists & that the path is correct");
31  }//end catch
```

Writing to a file

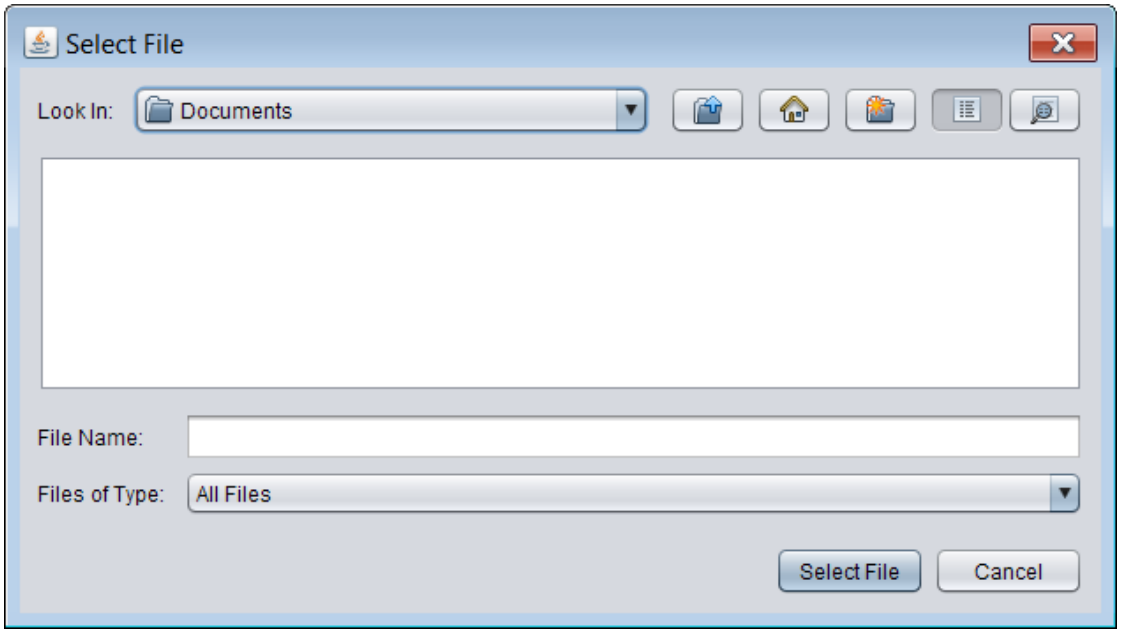
- To write to a file you can use one of the write methods.
- The following code shows how to write a String to a file.

```
12  String text = "This is the text I want to write to a file\n";
13
14
15  Path target = Paths.get("outputFile.txt");
16
17  try {
18       Files.write(target, text.getBytes());
19
20       //If you want to append text to a file use this
21       //Files.write(target, text.getBytes(), StandardOpenOption.APPEND);
22  }//end try
23  catch (IOException ex) {
24      System.out.println("Error: " + ex.toString());
25  }//end catch
26
27  System.out.println("Done!");
```

- If you are reading from/writing to large files, the use of Buffers is recommended.

Using JFileChooser (optional)

- File choosers provide a GUI for navigating the file system, and then either choosing a file or directory from a list, or entering the name of a file or directory.



Using JFileChooser (optional)

- To display a JFileChooser you need to firstly instantiate an object of that class and then invoke a method called “showOpenDialog” or “showSaveDialog” on it.
- The code snippet on the next slide displays a JFileChooser on the screen, allows the user to select a file and then displays the files contents in a text area (called jta).

Using JFileChooser (optional)

```
4   JFileChooser jfc = new JFileChooser();
5
6   int choice = jfc.showOpenDialog(this);
7
8   if (choice == JFileChooser.APPROVE_OPTION) {
9
10      String output = "";
11      List<String> lines = null;
12      File f = jfc.getSelectedFile();
13      Path source = Paths.get(f.getAbsolutePath());
14
15  try {
16          lines = Files.readAllLines(source,Charset.forName("UTF-8"));
17  }//end try
18  catch (IOException ex) {
19          JOptionPane.showMessageDialog(null, "Error");
20  }//end catch
21
22  for(String item : lines ){
23      output += item + "\n";
24  }//end for
25  jta.setText(output);
```

Using JFileChooser (optional)

- The code snippet on the next slide displays a JFileChooser on the screen, allows the user to enter a file name and then proceeds to save the contents of a JTextArea (called jta) to the file the user has entered.

Using JFileChooser (optional)

```
4   JFileChooser jfc = new JFileChooser();
5
6   int option = jfc.showSaveDialog(this);
7
8   if (option == JFileChooser.APPROVE_OPTION) {
9       File f = jfc.getSelectedFile();
10      Path pathToFile = Paths.get(f.getPath());
11      String linesotoadd = jta.getText();
12
13      try {
14          Files.write(pathToFile, linesotoadd.getBytes(), StandardOpenOption.APPEND);
15      }//end try
16
17      catch (IOException ex) {
18          JOptionPane.showMessageDialog(null, "Error");
19      }//end catch
20
21  }//end if
```

Object Streams

- Object streams enable you to perform input and output at the object level.
- To enable an object to be read or written, the object's defining class has to implement the `java.io.Serializable` interface or the `java.io.Externalizable` interface.
- The process of writing objects is referred to as object serialization.
- The process of reading objects is referred to as object deserialization.

Object Streams

- The `Serializable` interface is a marker interface. It has no methods, so you don't need to add additional code in your class that implements `Serializable`.

- Implementing this interface enables the Java serialization mechanism to automate the process of storing the objects and arrays.

- The `Externalizable` interface extends the `Serializable` interface and defines the `readExternal` and `writeExternal` methods to enable customization of object streams.

Example 3: Saving Objects

For an example of saving and retrieving objects, consult Person.java and PersonTester.java

The transient Keyword

- By default, all the nonstatic variables of a serialized object are written to the object stream.
- But, not all nonstatic variables can be serialized.
- For example, `java.awt.Thread` class does not implement `Serializable`, so a `Thread` object cannot be serialized.
- The transient keyword marks a data field so that it will not be serialized.

```
1  public class Foo {
2      private int v1;
3      private static double v2;
4      transient int v3;
5  }
```

Parsing Text Files

The `StreamTokenizer` class lets you take an input stream and parse it into words, which are known as *tokens*. The tokens are read one at a time. The following is the `StreamTokenizer` constructor:

```
StreamTokenizer st =  StreamTokenizer(Reader is)
```

Parsing Text Files

`StreamTokenizer` Constants

- `TT_WORD`

  The token is a word.

- `TT_NUMBER`

  The token is a number.

- `TT_EOL`

  The end of the line has been read.

- `TT_EOF`

  The end of the file has been read

Parsing Text Files

`StreamTokenizer` Variables

- `int ttype`

  Contains the current token type, which matches one of the constants listed on the preceding slide.

- `double nval`

  Contains the value of the current token if that token is a number.

- `String sval`

  Contains a string that gives the characters of the current token if that token is a word.

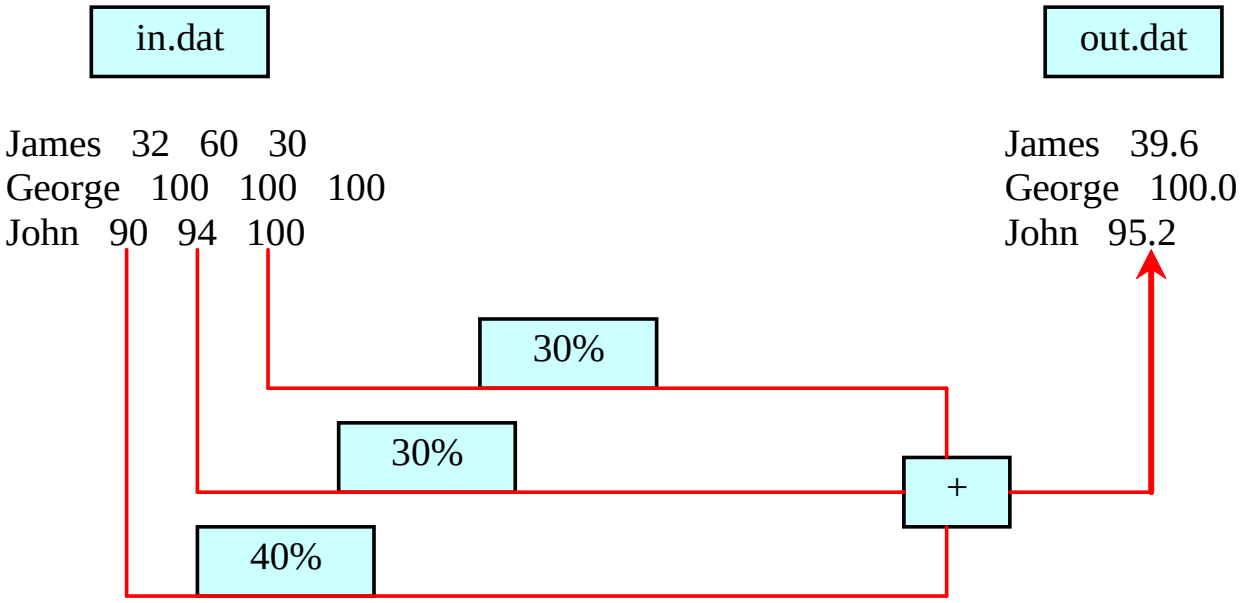
Parsing Text Files

`StreamTokenizer` Methods

```
public int nextToken() throws IOException
```

- Parses the next token from the input stream of this `StreamTokenizer`.
- The type of the next token is returned in the `ttype` field.
- If `ttype == TT_WORD`, the token is stored in `sval`.
- If `ttype == TT_NUMBER`, the token is stored in `nval`.

Example 4: Using StreamTokenizer



For an example of using StreamTokenizer, consult
ParsingTextFile.java