1495 UNIVERSITY OF ABERDEEN

JC2002 Java Programming

Lecture 37: Reading and writing to text files

References and learning objectives

- Today's sessions are mostly based on:
  - Evans & Flanagan, **Java in a Nutshell**, *7th edition*, 2018. O'Reilly Media.
  - Deitel, **Java How to Program**, 2018, Chapter 15.
- After today's session, you should be able to:
  - Create empty files, write data to, and read data from text and binary files
  - Write and read formatted data (e.g., in CSV format) in files
  - Select the appropriate method for using files in different applications

Files

- Data stored in variables and arrays is *temporary*: *d*ata in computer memory is lost, when a local variable goes out of scope or when the program terminates
- For long-term retention of data, even after the programs that create the data terminate, computers use *files*
  - Computers store files on *secondary storage devices*, or *mass storage devices*, including hard disks, flash drives, DVDs and others; files in mass storage are usually organised in *directories* forming a tree-like structure
  - Files are also often transferred over the Internet to share with others

Streams

- Java views each file as a *sequential stream of bytes*.
- A Java program processing a stream of bytes receives an indication from the operating system when it reaches the *end of the stream*
    - The program does not need to know how the underlying platform represents files or streams
    - In some cases, the *end-of-file* indication occurs as an exception
    - In others, the indication is a *return value* from a method invoked on a stream-processing object

Byte-based and character-based streams

- File streams can be used to handle data as *bytes* or *characters*
  - *Byte-based streams*: output and input data in its binary format: char is two bytes, int is four bytes, double is eight bytes, etc.
  - *Character-based streams*: output and input data as a sequence of characters in which every character is two bytes (i.e., char)
- Files created using byte-based streams are *binary files*, whereas files created using character-based streams are *text files*
  - Text files can be read by text editors, whereas binary files are read by programs that understand the specific content and format of the file

Creating and using files

- A Java program opens a file by creating an object and associating a stream of bytes or characters with it
  - The constructor of the object interacts with the operating system to open the file
- To work with files, we can use class **`File`** from the **`java.io`** package
  - To use a file, create an object of class `File`, and specify the filename or directory name as an argument to the constructor:

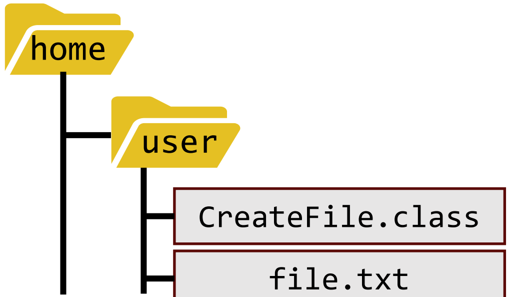
```
File myFile = new File("filename.txt");
```

Java file paths

- If no path name is defined in `File` constructor, the file is assumed to be in the working directory where the program is running
- Relative (abstract) path or absolute path can be used to define the location of the file in the directory structure
  - Note that absolute path names are system dependent: paths in e.g., Windows and Linux (Unix) are defined differently
  - Method **`getAbsolutePath()`** of `File` can be used to get a `String` object with the absolute path of the `File` object

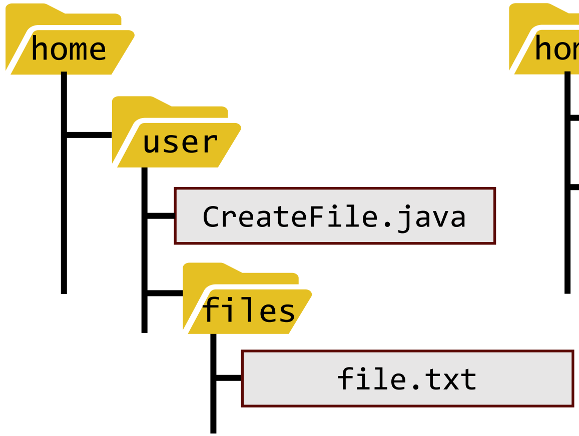
Java file path examples

- Relative path



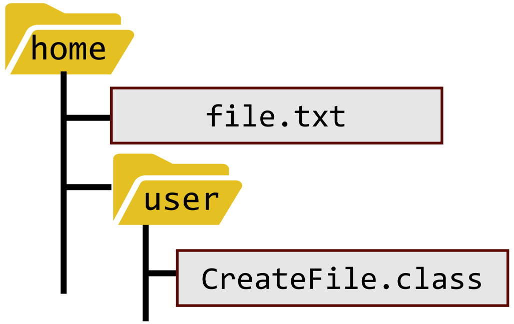
```
File("file.txt");
```

- Relative path



```
File("files/file.txt");
```

- Absolute path



```
File("/home/file.txt");
```

Methods of class File

- Some of the most useful methods of class `File` include:
  - **`createNewFile()`**: Creates a new empty file
  - **`delete()`**: Deletes the file
  - **`exists()`**: Tests if the file exists
  - **`getName()`**: Returns a `String` object with the name of the file
  - **`getAbsolutePath()`**: Returns a string with the absolute path
  - **`length()`**: Returns length of the file
  - **`list()`**: Returns an array of strings with the names of the files and directories located in the directory denoted by the `File` object pathname
  - **`mkdir()`**: Creates a new empty directory using the file name

Create file example (1)

```java
import java.io.*;
public class CreateFile {
  public static void main(String[] args) {
    try {
      File myFile = new File("myfilename.txt");
      if (myFile.createNewFile()) {
        System.out.println("File created: " + myFile.getName());
      } else {
        System.out.println("File already exists.");
      }
    } catch(IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

Method **`createNewFile()`** returns `true` if file created successfully

Method **`getName()`** returns the name of the `File` object

`IOException` must be caught in case of file error (e.g., disk full)

Create file example (2)

```java
import java.io.*;
public class CreateFile {
  public static void main(String[] args) {
    try {
      File myFile = new File("myfilename.txt");
      if (myFile.createNewFile()) {
        System.out.println("File created: " + myFile.getName());
      } else {
        System.out.println("File already exists.");
      }
    } catch(IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

```
$ java CreateFile
File created: myfilename.txt
$ java CreateFile
File already exists.
$
```

UNIVERSITY OF ABERDEEN

Writing to files using FileWriter class

- There are many ways to write data to a text file: using class **`FileWriter`** is one of the most straightforward
  - Constructor of `FileWriter` takes either `File` object or a `String` object with the file name as input argument
  - Additional `boolean` argument can be used to allow appending to the end of an existing file, rather than rewriting to the beginning
- Method **`write()`** can be used for writing
  - Take a `String` object or a `char[]` array as an input argument
  - Note that the data is not physically written in the file before closing the file with the method **`close()`** or flushing it with the method **`flush()`**

Writing to file example (1)

```java
import java.io.*;
public class WriteFile {
  public static void main(String[] args) {
    String fileName = "myfile.txt";
    FileWriter writer = null;
    try {
      writer = new FileWriter(fileName,true);
      writer.write("Data written in the file!\n");
      writer.close();
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

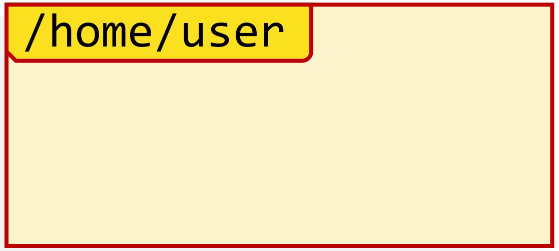
Creates a new file for writing, or if the file exists already, opens the existing file for appending

Writes or appends the data

Writes the data to the physical file and closes the file

Writing to file example (2)

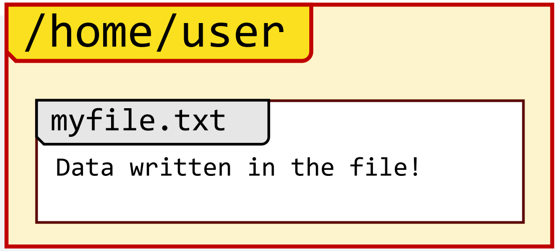
```java
import java.io.*;
public class WriteFile {
  public static void main(String[] args) {
    String fileName = "myfile.txt";
    FileWriter writer = null;
    try {
      writer = new FileWriter(fileName,true);
      writer.write("Data written in the file!\n");
      writer.close();
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```



```
$ javac WriteFile.java
$
```

Writing to file example (3)

```java
import java.io.*;
public class WriteFile {
  public static void main(String[] args) {
    String fileName = "myfile.txt";
    FileWriter writer = null;
    try {
      writer = new FileWriter(fileName,true);
      writer.write("Data written in the file!\n");
      writer.close();
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```



```
Data written in the file!
```

```
$ javac WriteFile.java
$ java WriteFile
$
```

Writing to file example (4)

```java
import java.io.*;
public class WriteFile {
  public static void main(String[] args) {
    String fileName = "myfile.txt";
    FileWriter writer = null;
    try {
      writer = new FileWriter(fileName,true);
      writer.write("Data written in the file!\n");
      writer.close();
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

/home/user

myfile.txt

```
Data written in the file!
Data written in the file!
```

```
$ javac WriteFile.java
$ java WriteFile
$ java WriteFile
$
```

Reading from files using FileReader class

- There are several different ways to read from files in Java; using class **`FileReader`** is symmetric to using `FileWriter` for writing
  - `FileReader` provides method **`read()`** that can be used to read characters from the file
- Since `FileReader` functionality is limited, it is more recommended to use **`BufferedReader`** class for reading text files line by line
  - `BufferedReader` object is created by using a `FileReader` object as input argument to the constructor
  - Method **`readLine()`** can then be used to read text file line by line

Reading lines from file example (1)

```
import java.io.*;
public class ReadFile {
  public static void main(String[] args) {
    try {
      FileReader reader = new FileReader("myfile.txt");
      BufferedReader bufReader = new BufferedReader(reader);
      String line;
      while ((line = bufReader.readLine()) != null) {
        System.out.println(line);
      }
      reader.close();
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

Creates `FileReader` and `BufferedReader` objects

Method `readLine()` reads the next line, returns `null` if no more lines left in the file

Reading lines from file example (2)

```java
import java.io.*;
public class ReadFile {
  public static void main(String[] args) {
    try {
      FileReader reader = new FileReader("myfile.txt");
      BufferedReader bufReader = new BufferedReader(reader);
      String line;
      while ((line = bufReader.readLine()) != null) {
        System.out.println(line);
      }
      reader.close();
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

myfile.txt

```
Data written in the file!
Data written in the file!
```

```
$ javac ReadFile.java
$ java ReadFile
Data written in the file!
Data written in the file!
$
```

Classes Files, Path, and Paths

- The legacy `File` class in package `java.io` has several weaknesses
  - Poor handling of errors, poor metadata support, scaling problems…
- Java package `java.nio` provides classes **`Files`**, **`Path`**, and **`Paths`** for improved management of files and directories
  - For basic file creation, writing, and reading operations, it is still appropriate to use `File`, `FileWriter`, `Scanner`, etc.
  - If more advanced or large-scale processing of files and directories (e.g., renaming or moving files), it is recommended to use `java.nio` package
  - Due to time constraints, we do not cover `java.nio` on this course

Questions, comments?