Lecture 23

Java

File Handling

Dr. Martin O’Connor
CA166
www.computing.dcu.ie/~moconnor

Topics

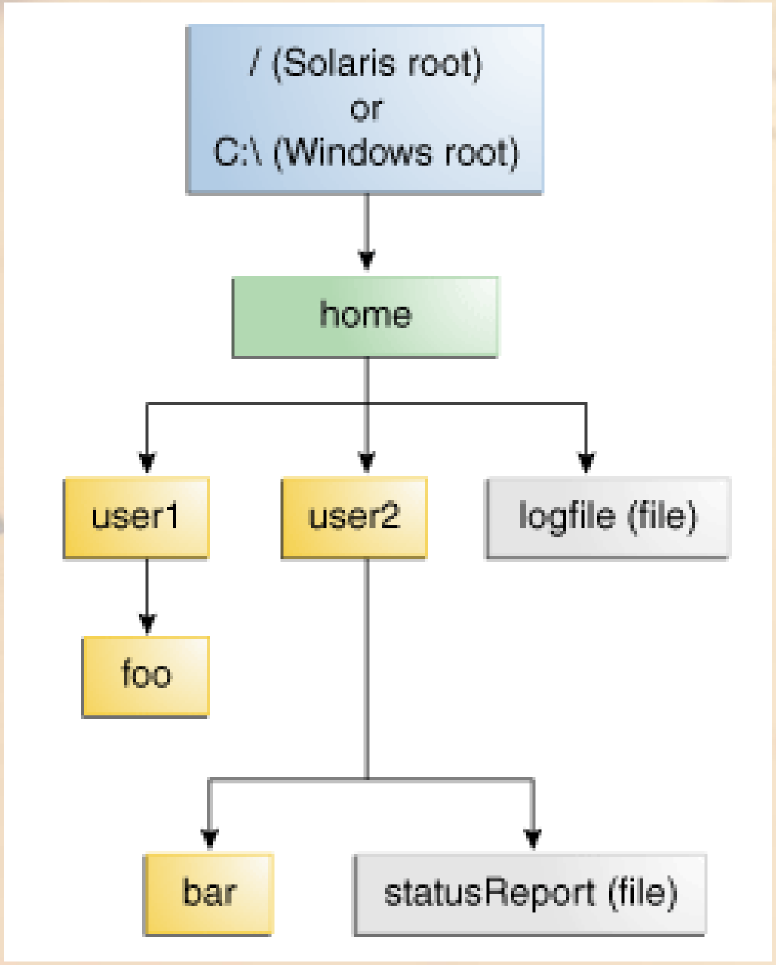
- File Systems
- What is a path?
- Symbolic links
- How to create a file
- How to obtain the file path from a file
- How to append data to a file
- How to delete a file
- How to rename a file
- How to get a file's last modified date
- How to check if a file exists
- Java version 7+ New File handling classes

File Systems

- 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

File Systems

- Consider the following sample directory tree



What is a Path?

- Microsoft Windows supports multiple root nodes. Each root node maps to a volume, such as C:\ or D:\ and so on

- The Linux OS supports a single root node, which is denoted by the slash character, /

- A file is identified by its path through the file system, beginning from the root node

- For example, the *statusReport* file is described by the following notation:
  - /home/user2/statusReport (In Linux)
  - C:\home\user2\statusReport (In Microsoft Windows)

What is a Path?

- The character used to separate the directory names (also called the *delimiter* or *file separator*) is specific to the file system
  - The Linux OS uses the forward slash (/)
  - Microsoft Windows uses the backslash slash (\)
- 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, **/home/user2/statusReport** is an absolute path. All of the information needed to locate the file is contained in the path string

What is a Path?

- A relative path needs to be combined with another path in order to access a file
  - For example, joe/foo is a relative path
  - Without more information, a program cannot reliably locate the joe/foo directory in the file system

Symbolic Links

- File system objects are most typically directories or files

- But some file systems also support the notion of symbolic links.
  - A symbolic link is also referred to as a *symlink* or a *soft link*.

- A *symbolic link* is a special file that serves as a reference to another file

Symbolic Links

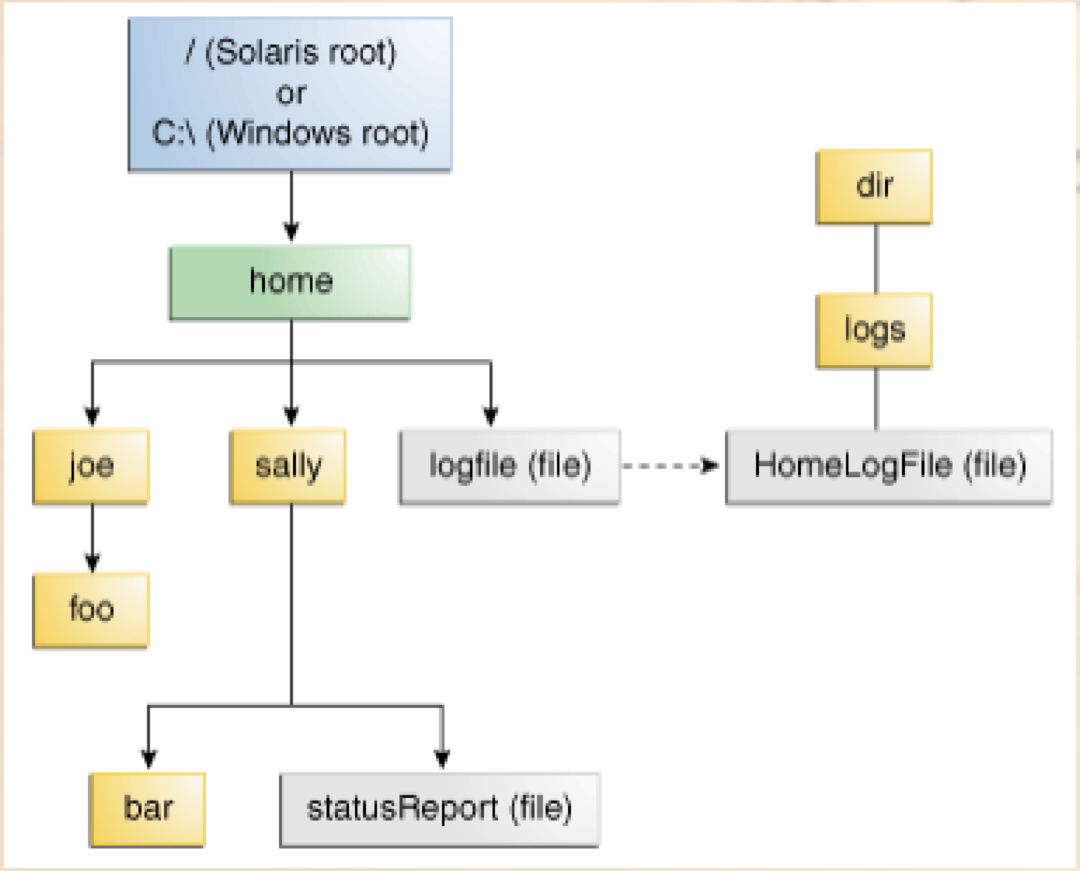
- For the most part, symbolic links are transparent to applications, and operations on symbolic links are automatically redirected to the target of the link

- The file or directory being pointed to is called the *target* of the link

- Exceptions are when a symbolic link is deleted, or renamed in which case the link itself is deleted, or renamed and not the target of the link

Example of a Symbolic Link

- In the following example, logFile appears to be a regular file to the user, but it is actually a symbolic link to dir/logs/HomeLogFile



Example of a Symbolic Link

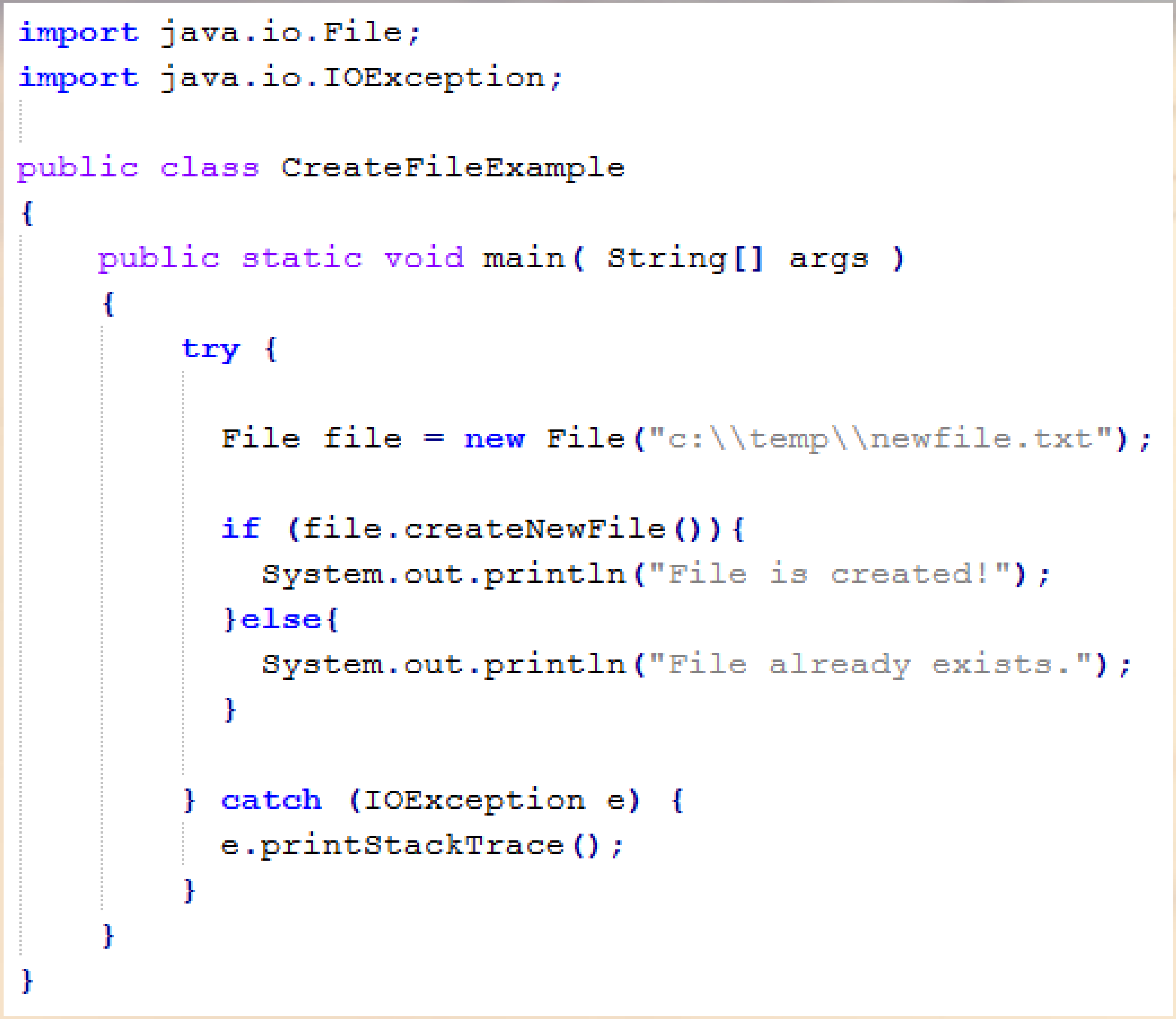
- A symbolic link is usually transparent to the user

- Reading or writing to a symbolic link is the same as reading or writing to any other file or directory

- The phrase *resolving a link* means to substitute the actual location in the file system for the symbolic link

- In the previous example, resolving logFile yields dir/logs/HomeLogFile

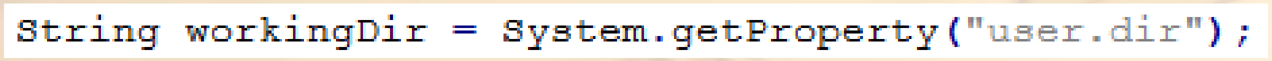
How to create a file
(one way among many)



```java
import java.io.File;
import java.io.IOException;

public class CreateFileExample
{
    public static void main( String[] args )
    {
        try {

          File file = new File("c:\\temp\\newfile.txt");

          if (file.createNewFile()){
            System.out.println("File is created!");
          }else{
            System.out.println("File already exists.");
          }

        } catch (IOException e) {
          e.printStackTrace();
        }
    }
}
```

How to construct a file path (independent of Operating System)

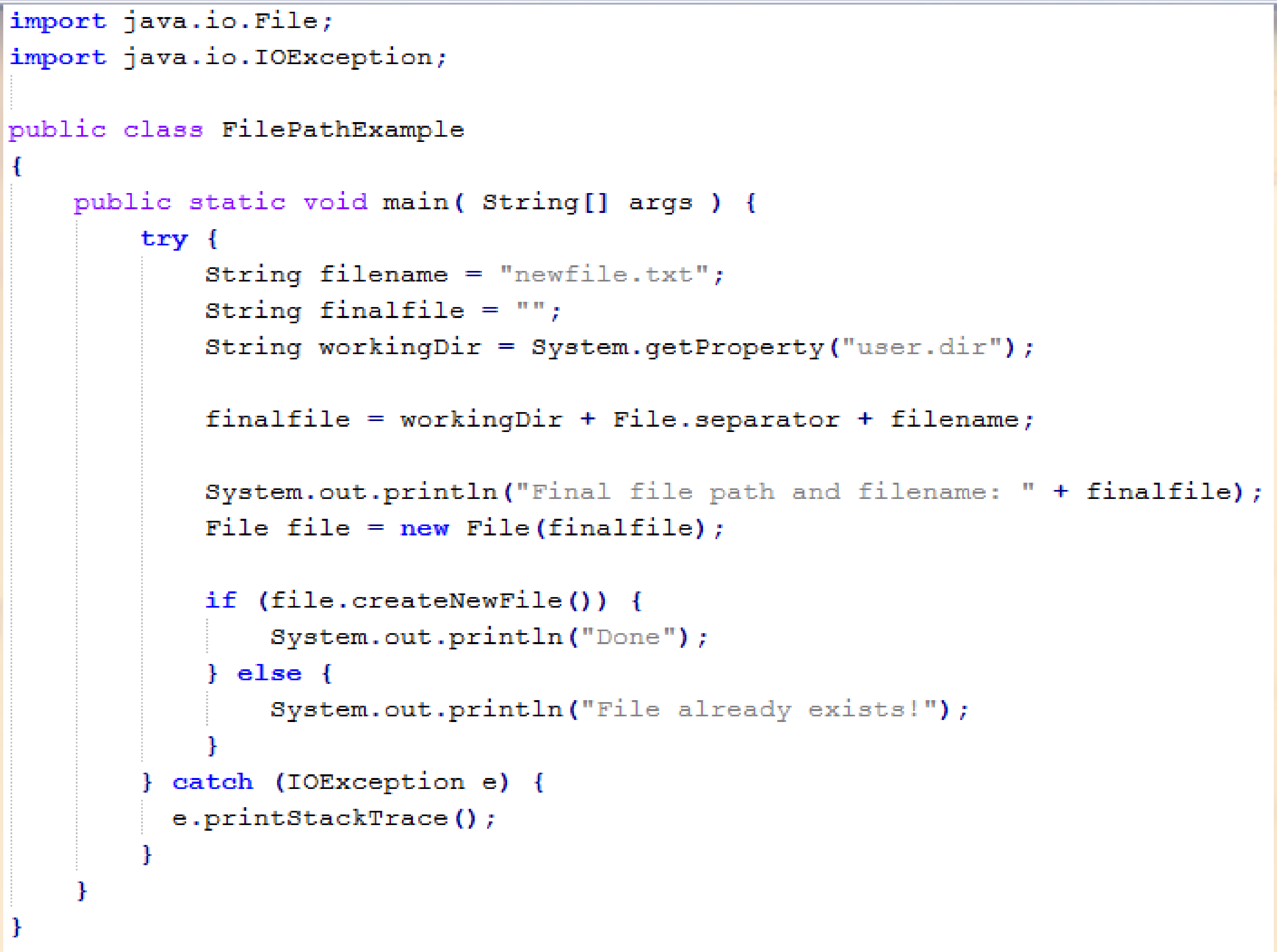
- Use the following to get the current working directory (folder)



```
String workingDir = System.getProperty("user.dir");
```

- Use File.separator to obtain the separator for the underlying operating system
  - Windows = \ E.g.: C:\temp\test.txt
  - Unix = / E.g.: /home/users/text.txt

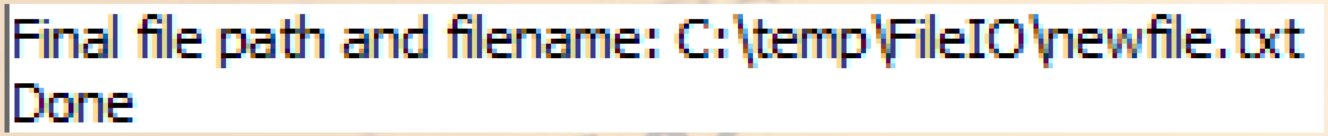
How to construct a file path (independent of the Operating System)



```java
import java.io.File;
import java.io.IOException;

public class FilePathExample
{
    public static void main( String[] args ) {
        try {
            String filename = "newfile.txt";
            String finalfile = "";
            String workingDir = System.getProperty("user.dir");

            finalfile = workingDir + File.separator + filename;

            System.out.println("Final file path and filename: " + finalfile);
            File file = new File(finalfile);

            if (file.createNewFile()) {
                System.out.println("Done");
            } else {
                System.out.println("File already exists!");
            }
        } catch (IOException e) {
          e.printStackTrace();
        }
    }
}
```

How to construct a file path (independent of Operating System)

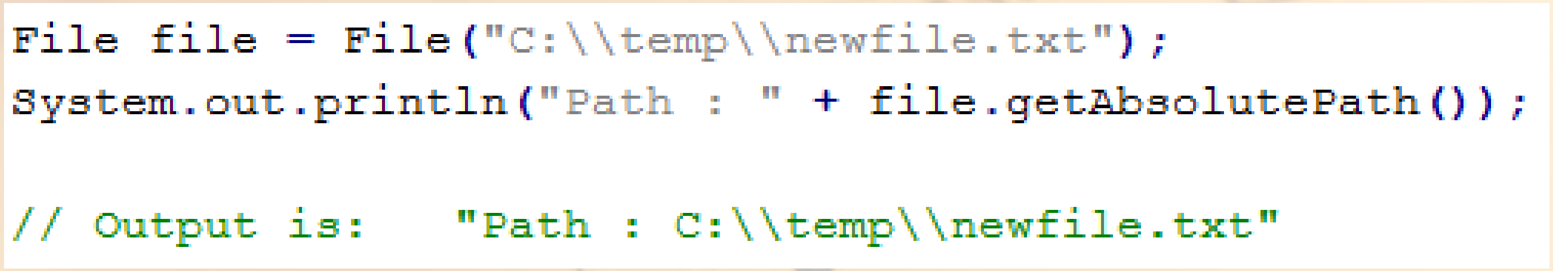
- The output from program FilePathExample is:



```
Final file path and filename: C:\temp\FileIO\newfile.txt
Done
```

How to get the file path of a file

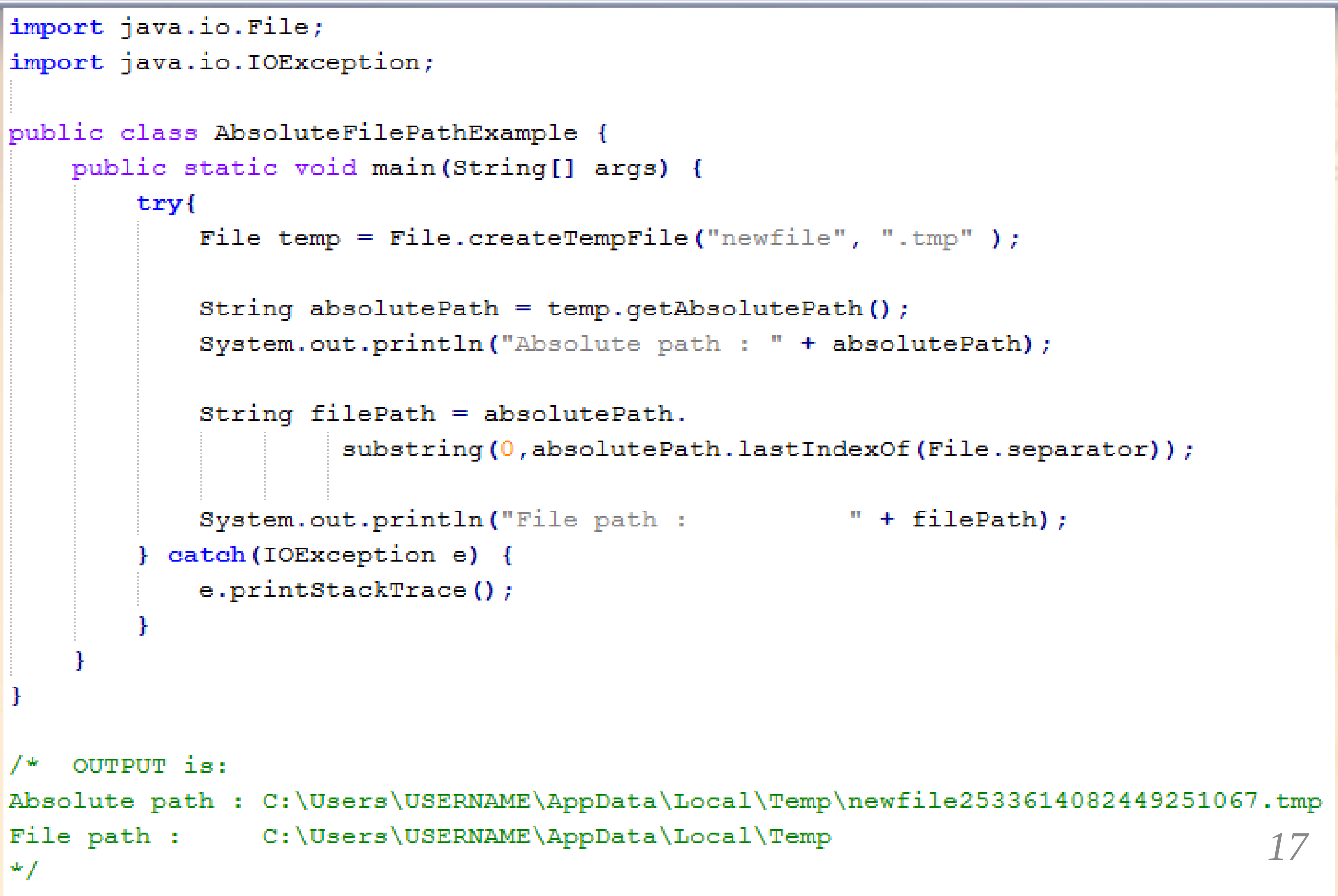
- The **File.getAbsolutePath()** will give you the full complete path name (filepath + filename) of a file



```
File file = File("C:\\temp\\newfile.txt");
System.out.println("Path : " + file.getAbsolutePath());

// Output is:   "Path : C:\\temp\\newfile.txt"
```

- To retrieve the file path without the file name, use substring() and lastIndexOf()

How to get the file path of a file



```
import java.io.File;
import java.io.IOException;

public class AbsoluteFilePathExample {
    public static void main(String[] args) {
        try{
            File temp = File.createTempFile("newfile", ".tmp" );

            String absolutePath = temp.getAbsolutePath();
            System.out.println("Absolute path : " + absolutePath);

            String filePath = absolutePath.
                        substring(0,absolutePath.lastIndexOf(File.separator));

            System.out.println("File path :          " + filePath);
        } catch(IOException e) {
            e.printStackTrace();
        }
    }
}

/*  OUTPUT is:
Absolute path : C:\Users\USERNAME\AppData\Local\Temp\newfile2533614082449251067.tmp
File path :     C:\Users\USERNAME\AppData\Local\Temp
*/
```

How to append data to a file

- Recall: **FileWriter**, a character stream to write characters to file
- By default, it will replace all the existing content with new content
- However, when you specified a **true (boolean) value** as the second argument in **FileWriter constructor**, it will keep the existing content and append the new content in the end of the file

How to append data to a file

- Thus:
  to Replace all existing content with new content, use:

  new FileWriter(file);

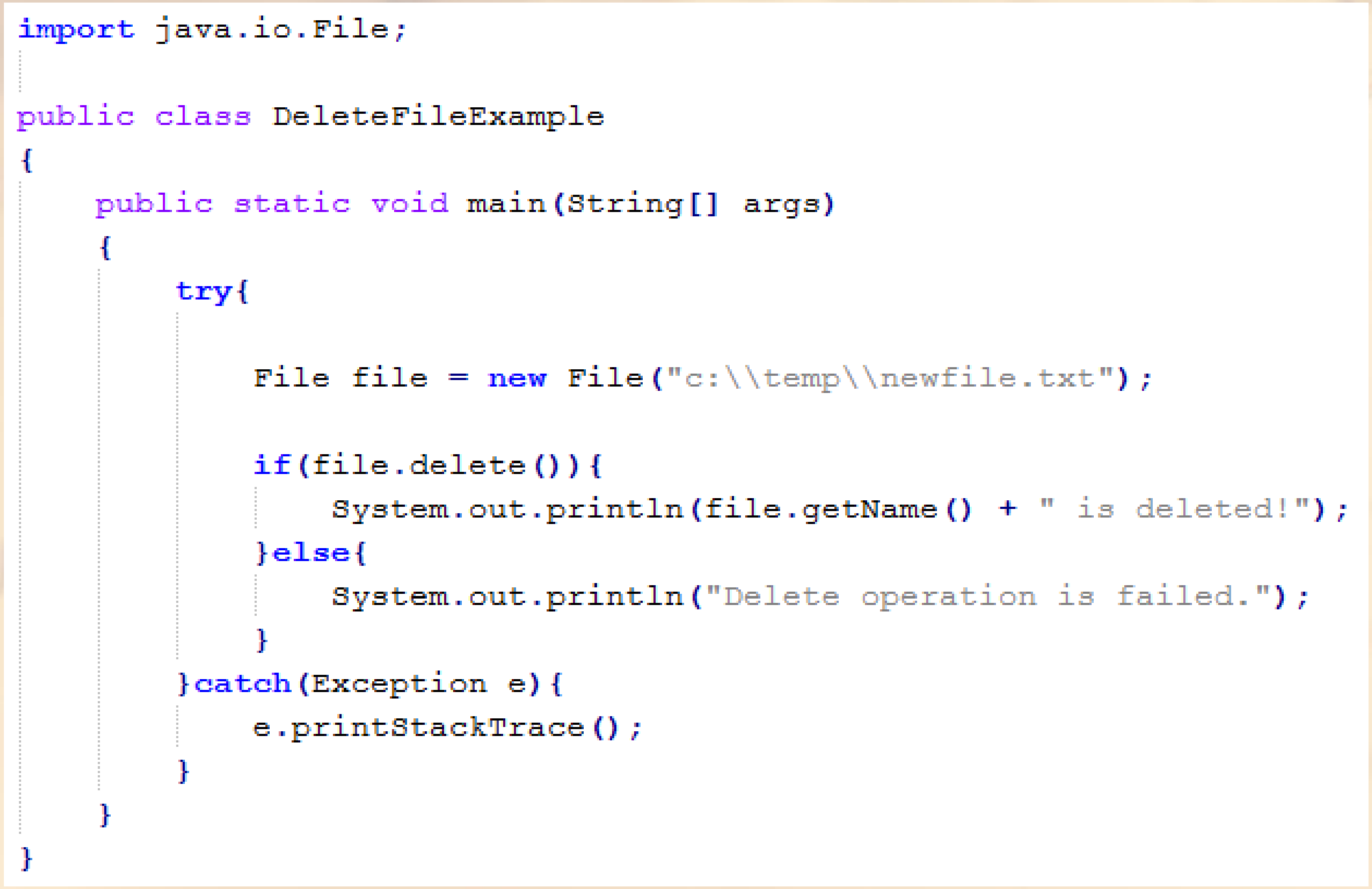
- To keep the existing content and append the new content in the end of the file, use:

  new FileWriter(file, **true**);

- **You should use the techniques described in the lecture on Input/Output (I/O) streams to read input and write output to files.**

How to delete a file

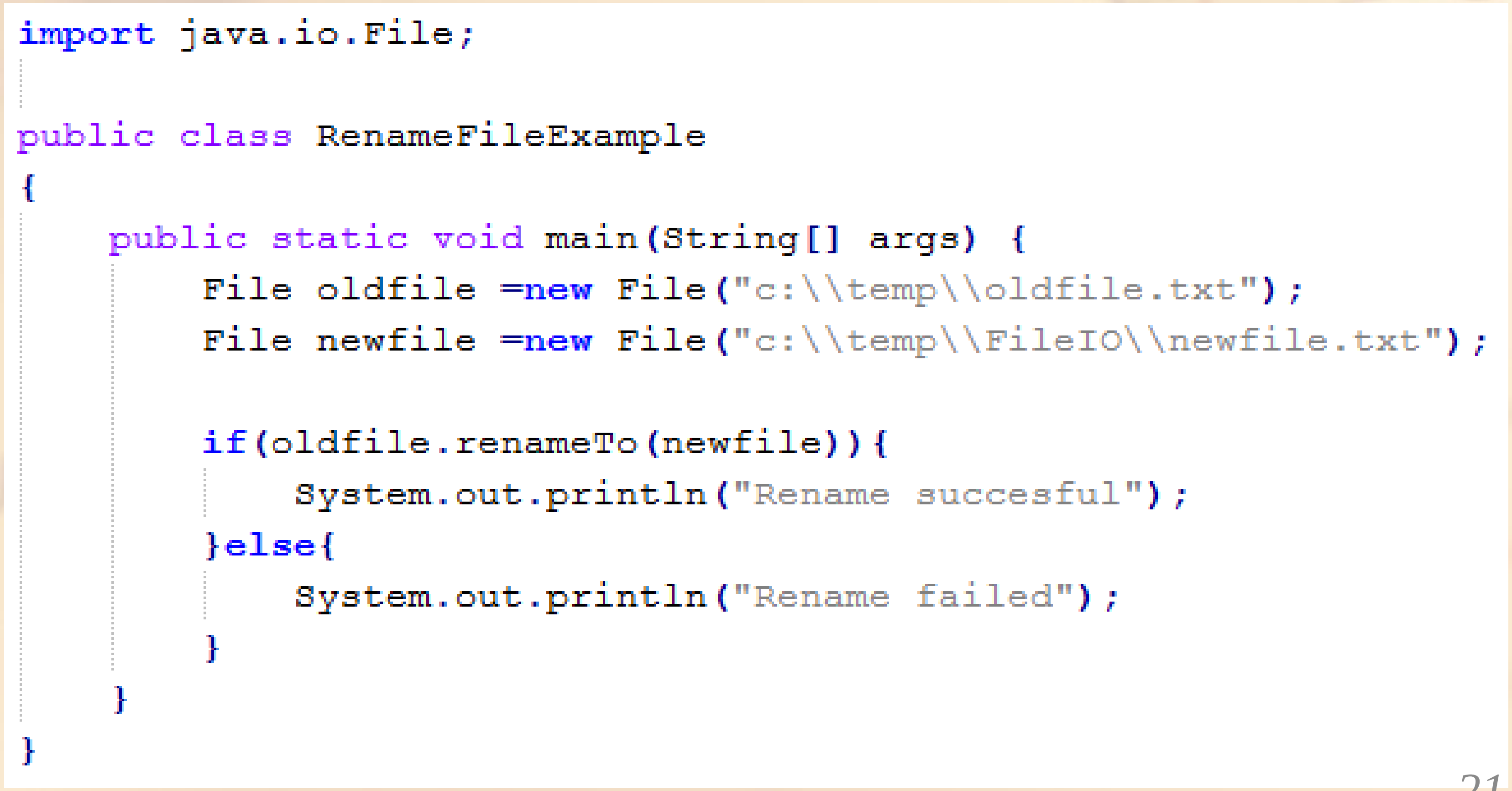
- Invoke the **File.delete()** method to delete a file



```java
import java.io.File;

public class DeleteFileExample
{
    public static void main(String[] args)
    {
        try{

            File file = new File("c:\\temp\\newfile.txt");

            if(file.delete()){
                System.out.println(file.getName() + " is deleted!");
            }else{
                System.out.println("Delete operation is failed.");
            }
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

How to rename a file

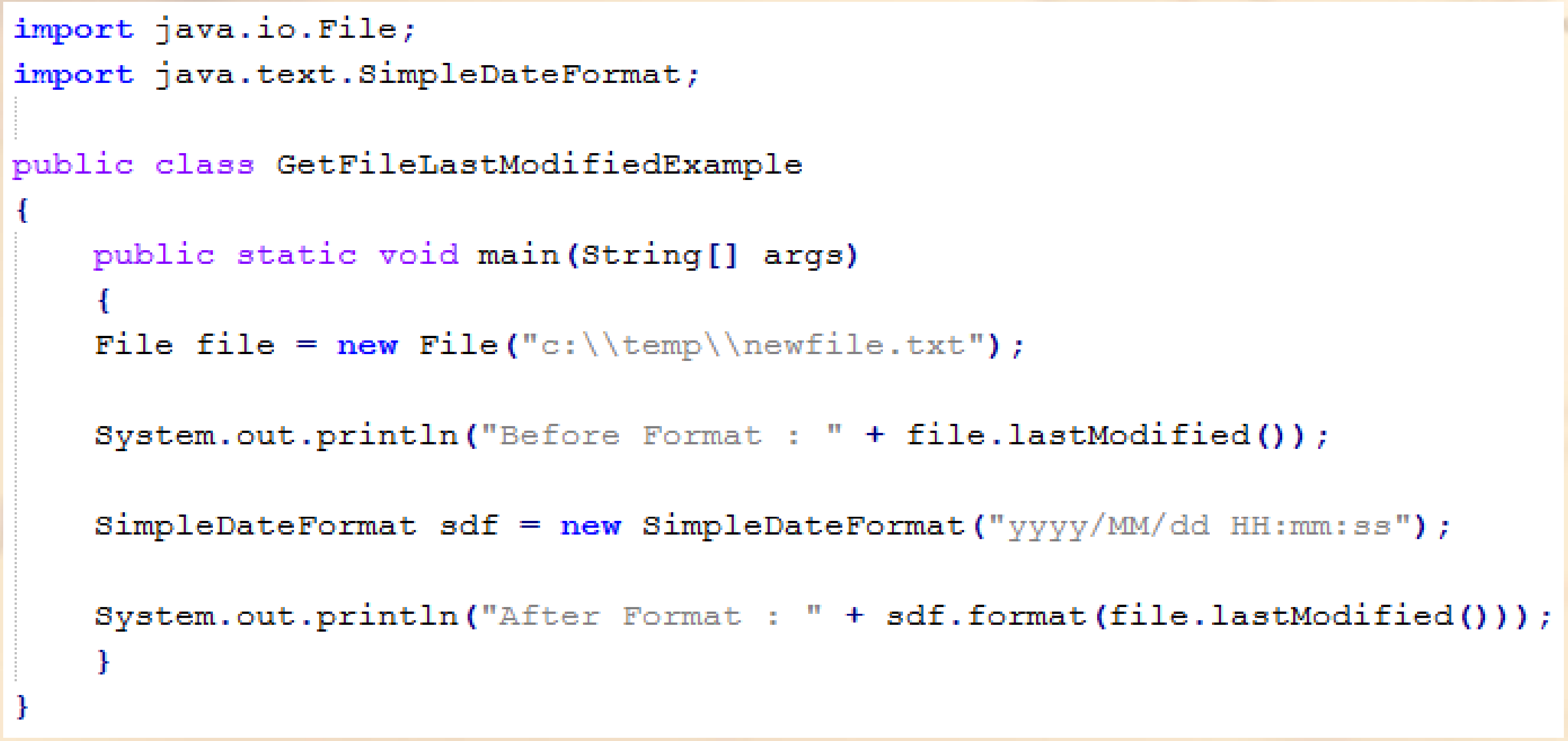
- Invoke the **File.renameTo()** method to rename (or move) a file



```
import java.io.File;

public class RenameFileExample
{
    public static void main(String[] args) {
        File oldfile =new File("c:\\temp\\oldfile.txt");
        File newfile =new File("c:\\temp\\FileIO\\newfile.txt");

        if(oldfile.renameTo(newfile)){
            System.out.println("Rename succesful");
        }else{
            System.out.println("Rename failed");
        }
    }
}
```

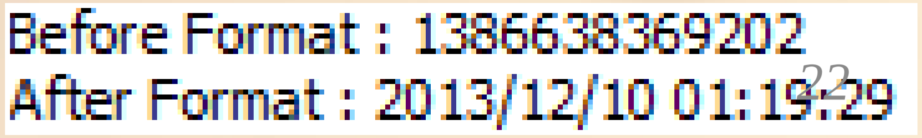
How to get the file last modified date

- Invoke the **File.lastModified()** method to get the file's last modified timestamp



```
import java.io.File;
import java.text.SimpleDateFormat;

public class GetFileLastModifiedExample
{
    public static void main(String[] args)
    {
    File file = new File("c:\\temp\\newfile.txt");

    System.out.println("Before Format : " + file.lastModified());

    SimpleDateFormat sdf = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");

    System.out.println("After Format : " + sdf.format(file.lastModified()));
    }
}
```

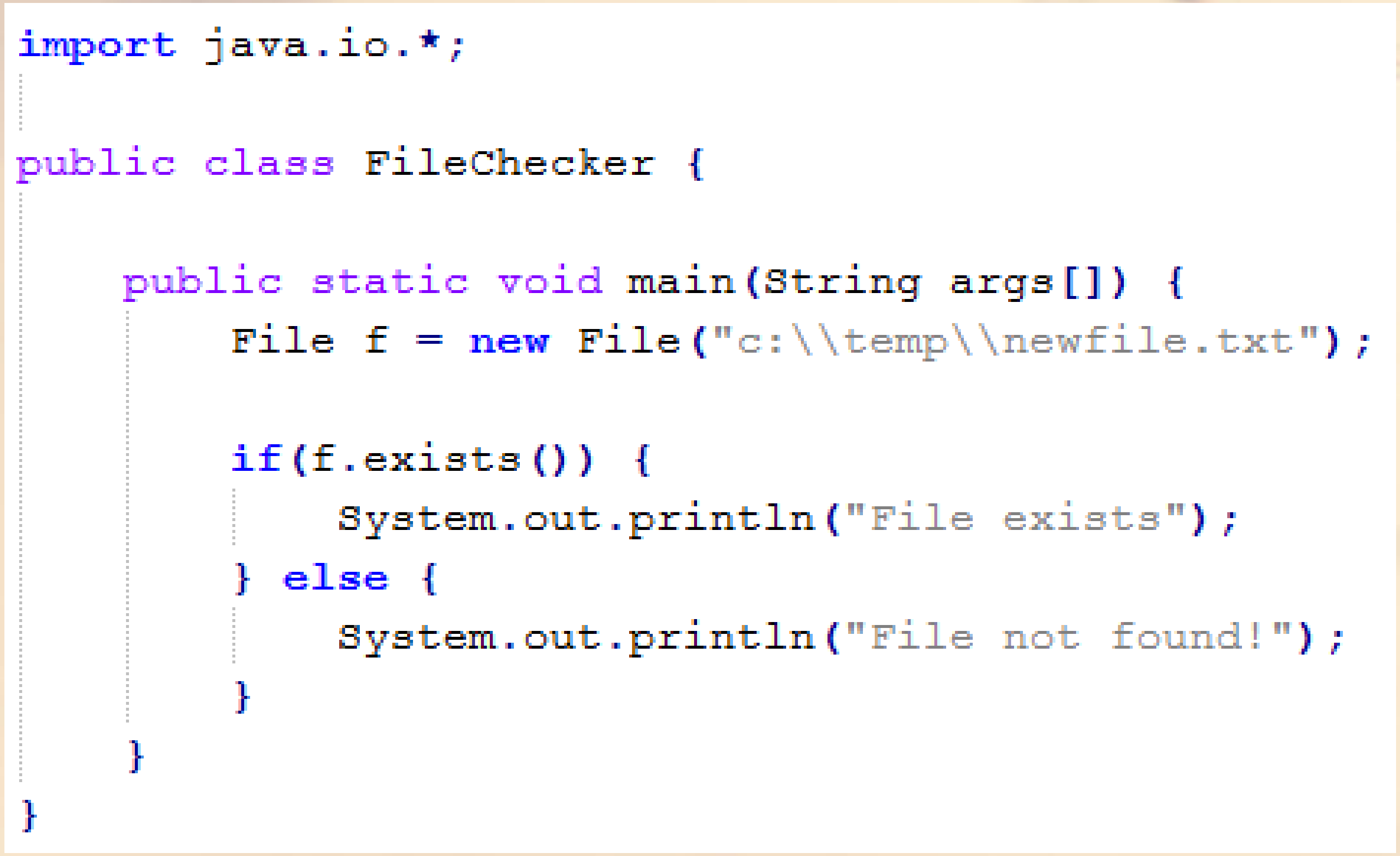
Output:



```
Before Format : 1386638369202
After Format : 2013/12/10 01:19:29
```

How to check if a file exists

- Invoke the **File.exists()** method



```java
import java.io.*;

public class FileChecker {

    public static void main(String args[]) {
        File f = new File("c:\\temp\\newfile.txt");

        if(f.exists()) {
            System.out.println("File exists");
        } else {
            System.out.println("File not found!");
        }
    }
}
```

File Processing

- You are expected to be able to use all of the programming constructs and structures encountered during this module in conjunction with file processing. For example:
  - String and numeric processing
  - Iterative processing (For loops and while loops)
  - Branch processing (Break, continue, return)
  - Process Command line arguments
  - Read and process text (and data) from input files
  - Write processed text (and data) to output files
  - Use exceptions where required
- Don’t forget to close streams in a finally block!

Java 7+

- Up to and including Java version 6, The **java.io.File** class provided both file location and file system operations.

- With Java 7, a new second approach was introduced that splits the functionality of the java.io.File class in two:

  – The new **Path** class provides just file location operations and additional path-related operations.

  – The new **Files** class provides file system operations (e.g.: create, copy, move, delete, read, write and so on)

Java 7+

- Java 7 (nio stands for New I/O)
  - import java.nio.file.Path
  - import java.nio.file.Files
- In Java7+, the java.io.File class provides the **toPath** method, which converts an old style File instance to a java.nio.file.Path instance
- Example of updating Java 6 code to Java 7 code:
  To delete a file:
  - PreJava7 Code: file.delete();
  - Java7+ Code: Path fp = file.toPath();
    Files.delete(fp);

Acknowledgements

- The slides are based on:
  - The Oracle Online Java Tutorial and
  - The Mkyong Java I/O Tutorial http://www.mkyong.com/tutorials/java-io-tutorials/