CE251 : JAVA PROGRAMMING
July – December 2020

File IO

Prof. Mohammed Bohra
Assistant Professor

Devang Patel Institute of Advance Technology and Research

Files and Exceptions

- When creating files and performing I/O operations on them, the systems generates errors. The basic I/O related exception classes are given below:
  - EOFException – signals that end of the file is reached unexpectedly during input.
  - FileNotFoundException – file could not be opened
  - InterruptedIOException – I/O operations have been interrupted
  - IOException – signals that I/O exception of some sort has occurred – very general I/O exception

Syntax

- Each I/O statement or a group of I/O statements much have an exception handler around it/them as follows:

```
try {
…// I/O statements – open file, read, etc.
}
catch(IOException e) // or specific type exception
{
    …//message output statements
}
```

Example

```
import java.io.*;
class CountBytesNew {

    public static void main (String[] args)
            throws FileNotFoundException, IOException
     {
            FileInputStream in;
            try{
                    in = new FileInputStream("FileIn.txt");
                    int total = 0;
                    while (in.read() != -1)
                            total++;
                    System.out.println("Total = " + total);
            }
            catch(FileNotFoundException e1)
            {
                    System.out.println("FileIn.txt does not exist!");
            }
            catch(IOException e2)
            {
                    System.out.println("Error occured while read file FileIn.txt");
            }
      }}
```

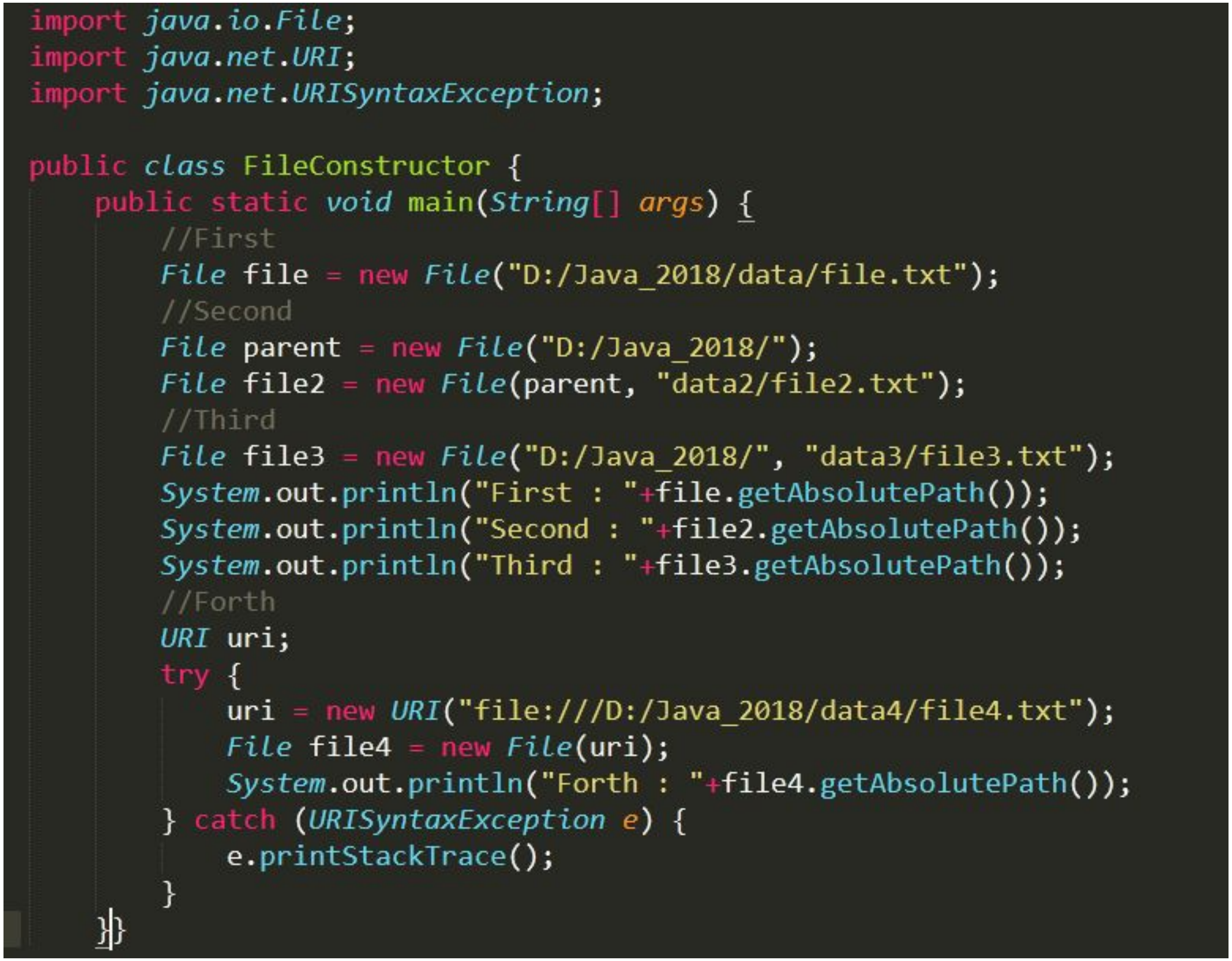
Java File Class

- Java File class is a part of java.io package.
- Java File class is an abstract representation of file and directory pathnames.

File Constructors

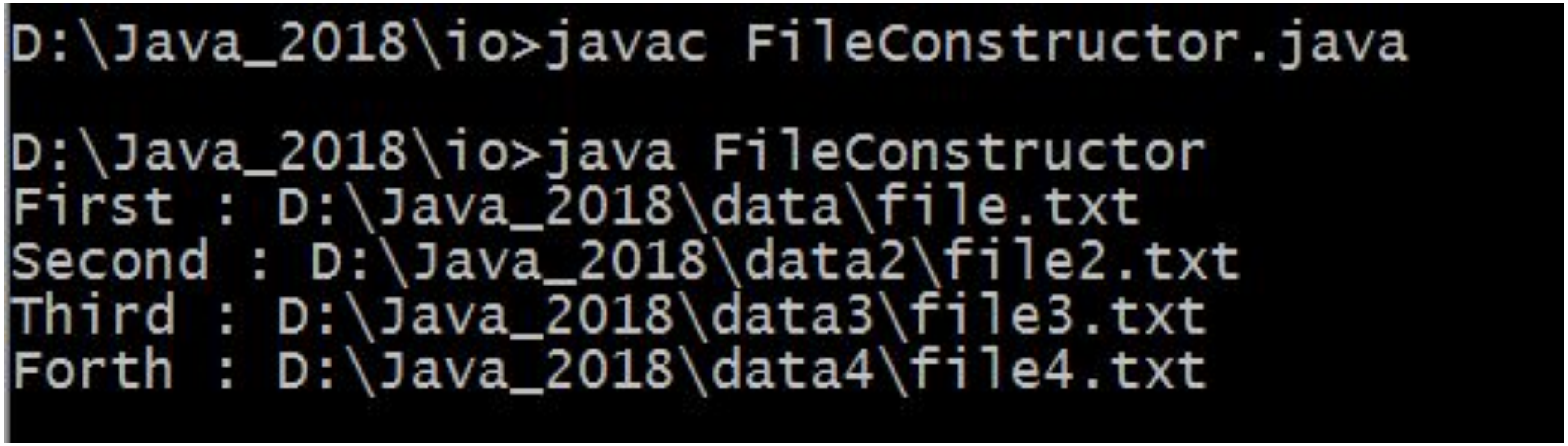
Constructor	Description
File(File parent, String child)	It creates a new File instance from a parent abstract pathname and a child pathname string.
File(String pathname)	It creates a new File instance by converting the given pathname string into an abstract pathname.
File(String parent, String child)	It creates a new File instance from a parent pathname string and a child pathname string.
File(URI uri)	It creates a new File instance by converting the given file: URI into an abstract pathname.

Example



```java
import java.io.File;
import java.net.URI;
import java.net.URISyntaxException;

public class FileConstructor {
    public static void main(String[] args) {
        //First
        File file = new File("D:/Java_2018/data/file.txt");
        //Second
        File parent = new File("D:/Java_2018/");
        File file2 = new File(parent, "data2/file2.txt");
        //Third
        File file3 = new File("D:/Java_2018/", "data3/file3.txt");
        System.out.println("First : "+file.getAbsolutePath());
        System.out.println("Second : "+file2.getAbsolutePath());
        System.out.println("Third : "+file3.getAbsolutePath());
        //Forth
        URI uri;
        try {
            uri = new URI("file:///D:/Java_2018/data4/file4.txt");
            File file4 = new File(uri);
            System.out.println("Forth : "+file4.getAbsolutePath());
        } catch (URISyntaxException e) {
            e.printStackTrace();
        }
    }}
```

Output

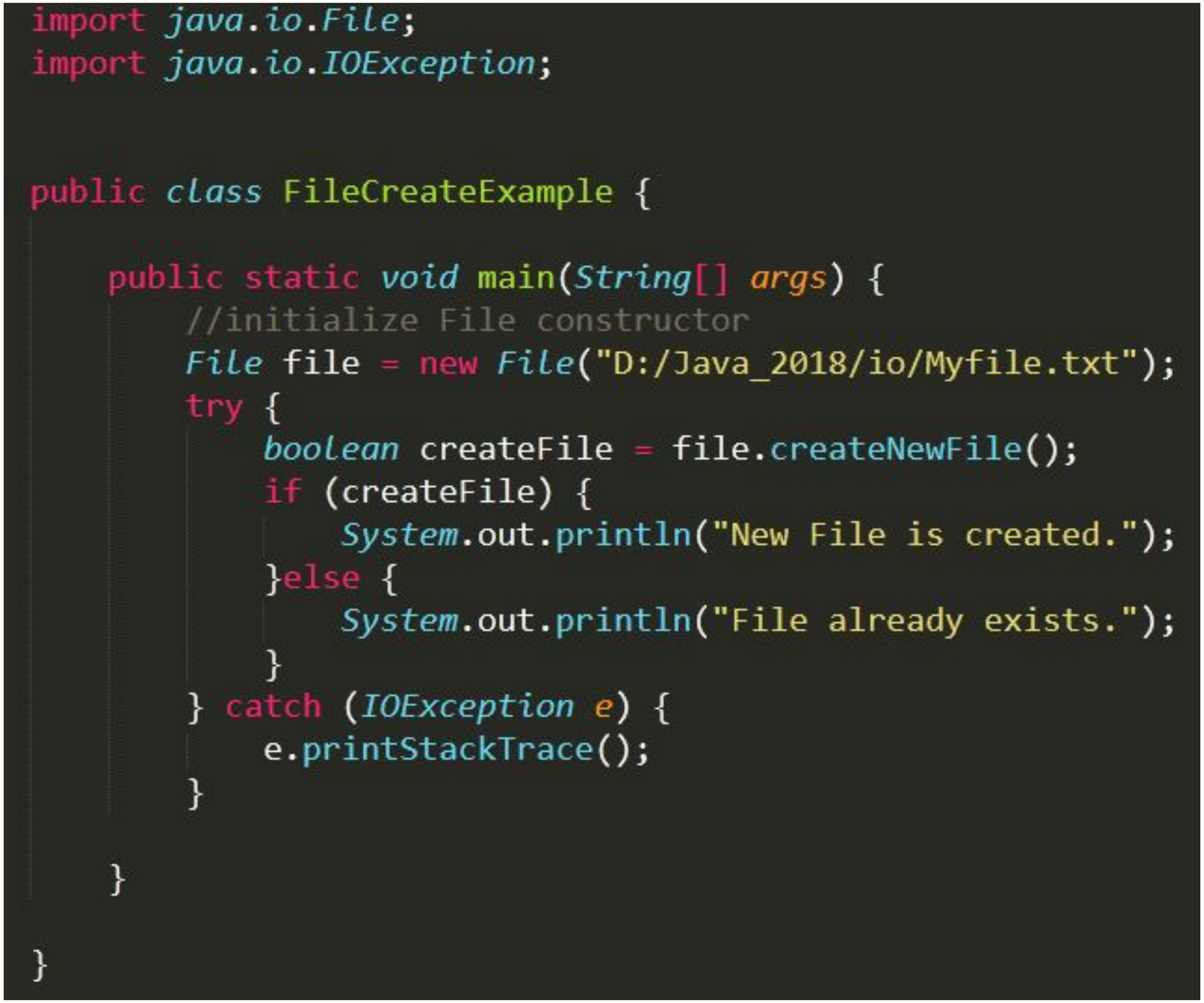


```
D:\Java_2018\io>javac FileConstructor.java

D:\Java_2018\io>java FileConstructor
First : D:\Java_2018\data\file.txt
Second : D:\Java_2018\data2\file2.txt
Third : D:\Java_2018\data3\file3.txt
Forth : D:\Java_2018\data4\file4.txt
```

File Class Useful Methods

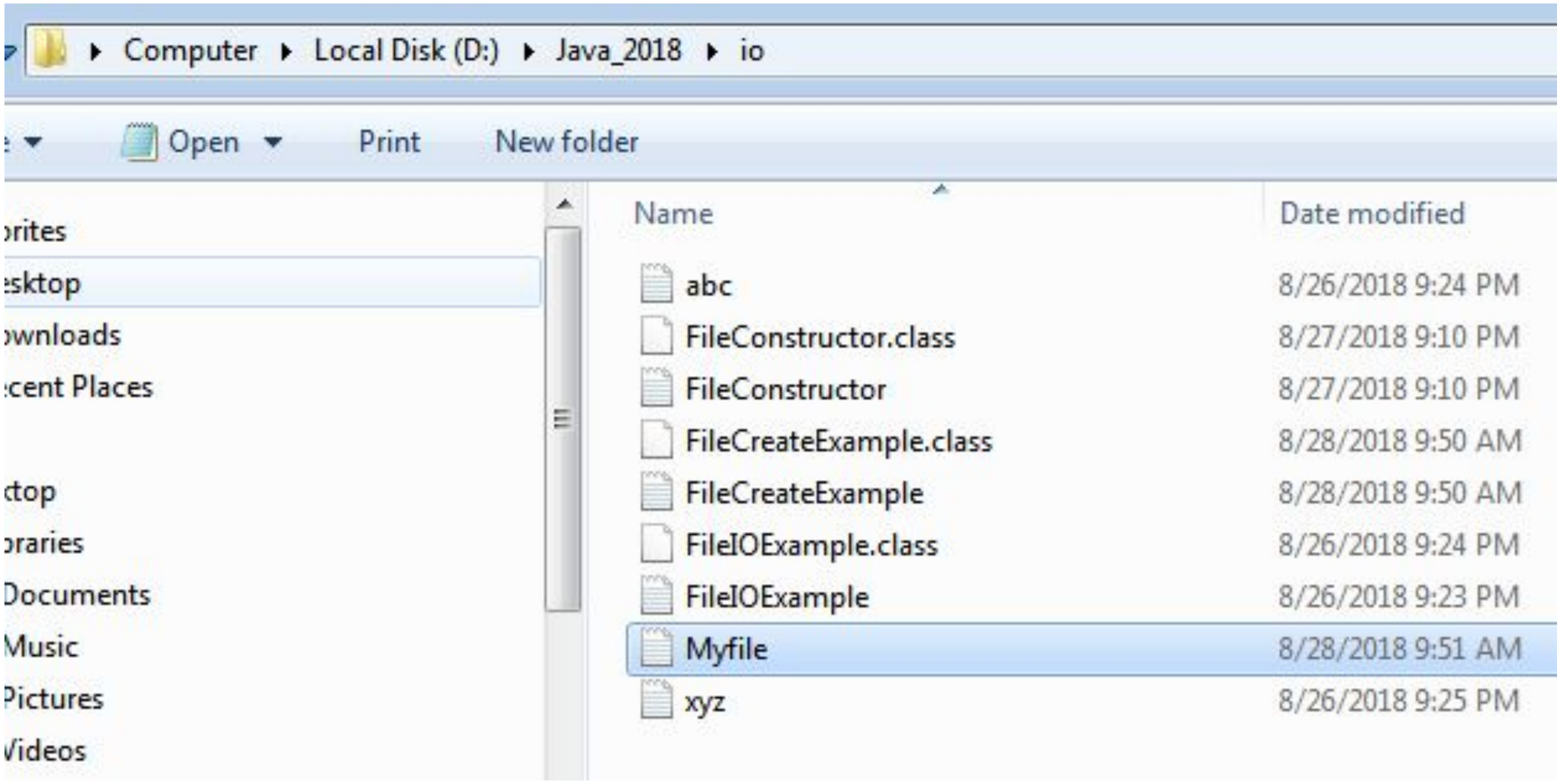
- 30+ methods

Create New File



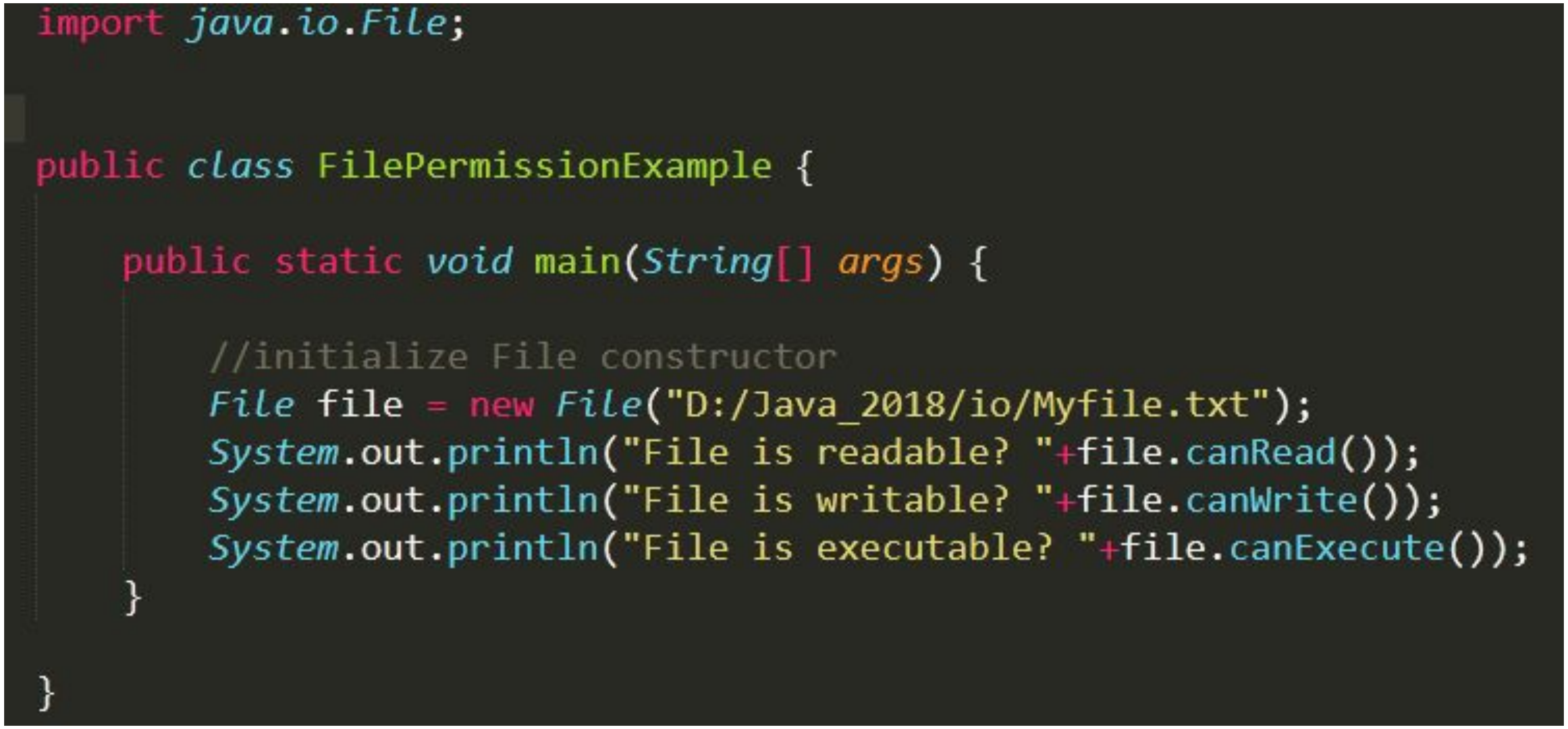
```java
import java.io.File;
import java.io.IOException;


public class FileCreateExample {

    public static void main(String[] args) {
        //initialize File constructor
        File file = new File("D:/Java_2018/io/Myfile.txt");
        try {
            boolean createFile = file.createNewFile();
            if (createFile) {
                System.out.println("New File is created.");
            }else {
                System.out.println("File already exists.");
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

    }

}
```

Output

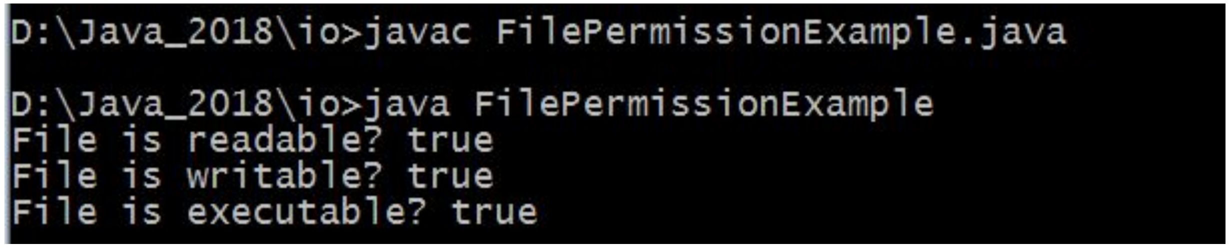


Name	Date modified
abc	8/26/2018 9:24 PM
FileConstructor.class	8/27/2018 9:10 PM
FileConstructor	8/27/2018 9:10 PM
FileCreateExample.class	8/28/2018 9:50 AM
FileCreateExample	8/28/2018 9:50 AM
FileIOExample.class	8/26/2018 9:24 PM
FileIOExample	8/26/2018 9:23 PM
Myfile	8/28/2018 9:51 AM
xyz	8/26/2018 9:25 PM

Check File Permissions

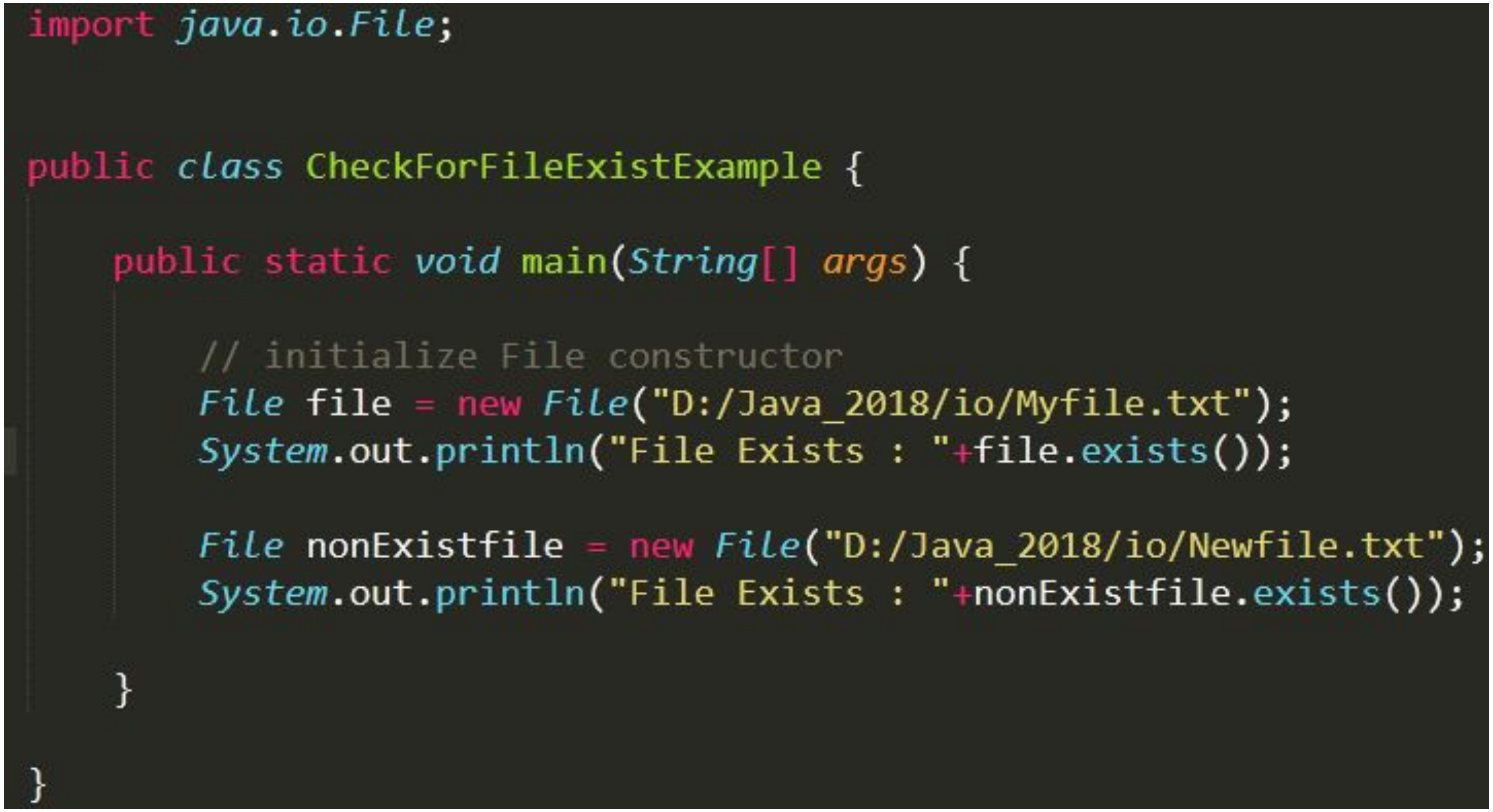


```java
import java.io.File;


public class FilePermissionExample {

    public static void main(String[] args) {

        //initialize File constructor
        File file = new File("D:/Java_2018/io/Myfile.txt");
        System.out.println("File is readable? "+file.canRead());
        System.out.println("File is writable? "+file.canWrite());
        System.out.println("File is executable? "+file.canExecute());
    }

}
```

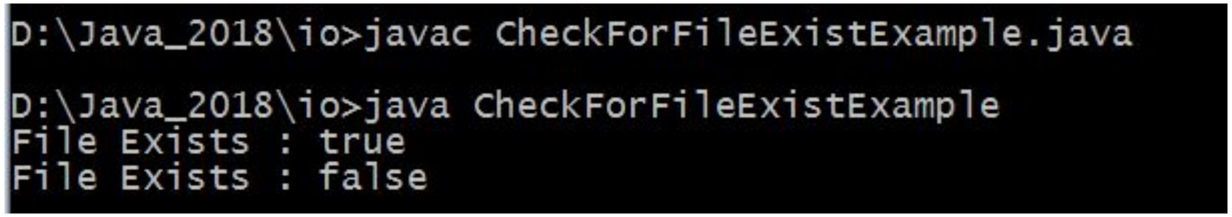


```
D:\Java_2018\io>javac FilePermissionExample.java

D:\Java_2018\io>java FilePermissionExample
File is readable? true
File is writable? true
File is executable? true
```

Check if File already exists?

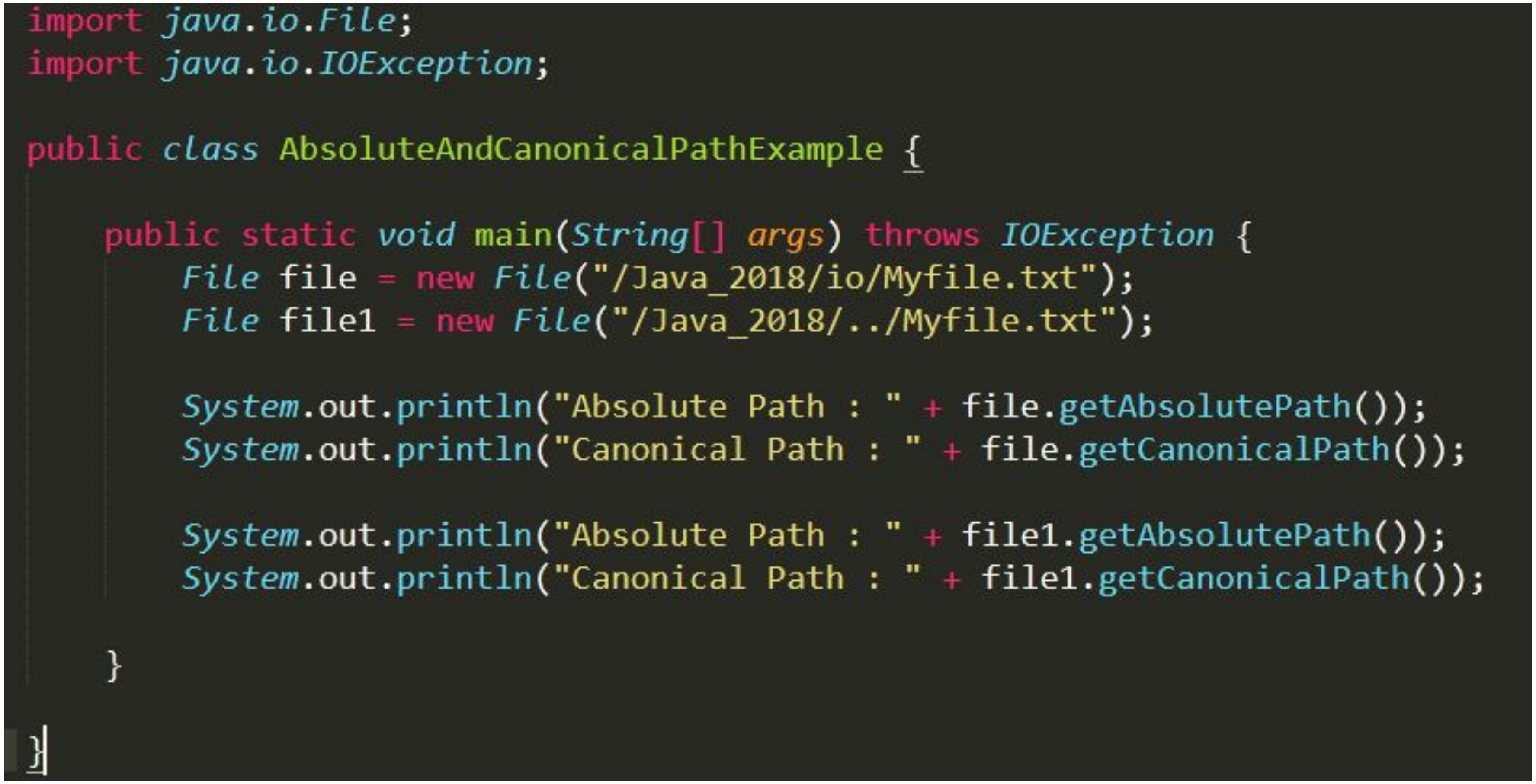


```java
import java.io.File;


public class CheckForFileExistExample {

    public static void main(String[] args) {

        // initialize File constructor
        File file = new File("D:/Java_2018/io/Myfile.txt");
        System.out.println("File Exists : "+file.exists());

        File nonExistfile = new File("D:/Java_2018/io/Newfile.txt");
        System.out.println("File Exists : "+nonExistfile.exists());

    }

}
```

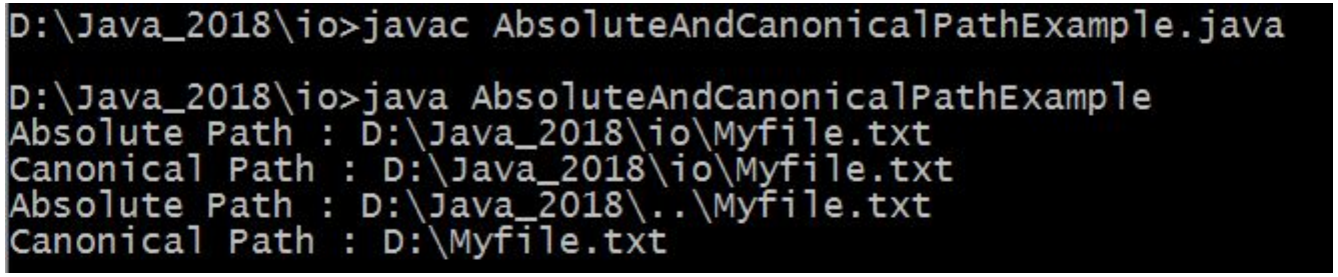


```
D:\Java_2018\io>javac CheckForFileExistExample.java

D:\Java_2018\io>java CheckForFileExistExample
File Exists : true
File Exists : false
```

Java File Absolute & Canonical Path

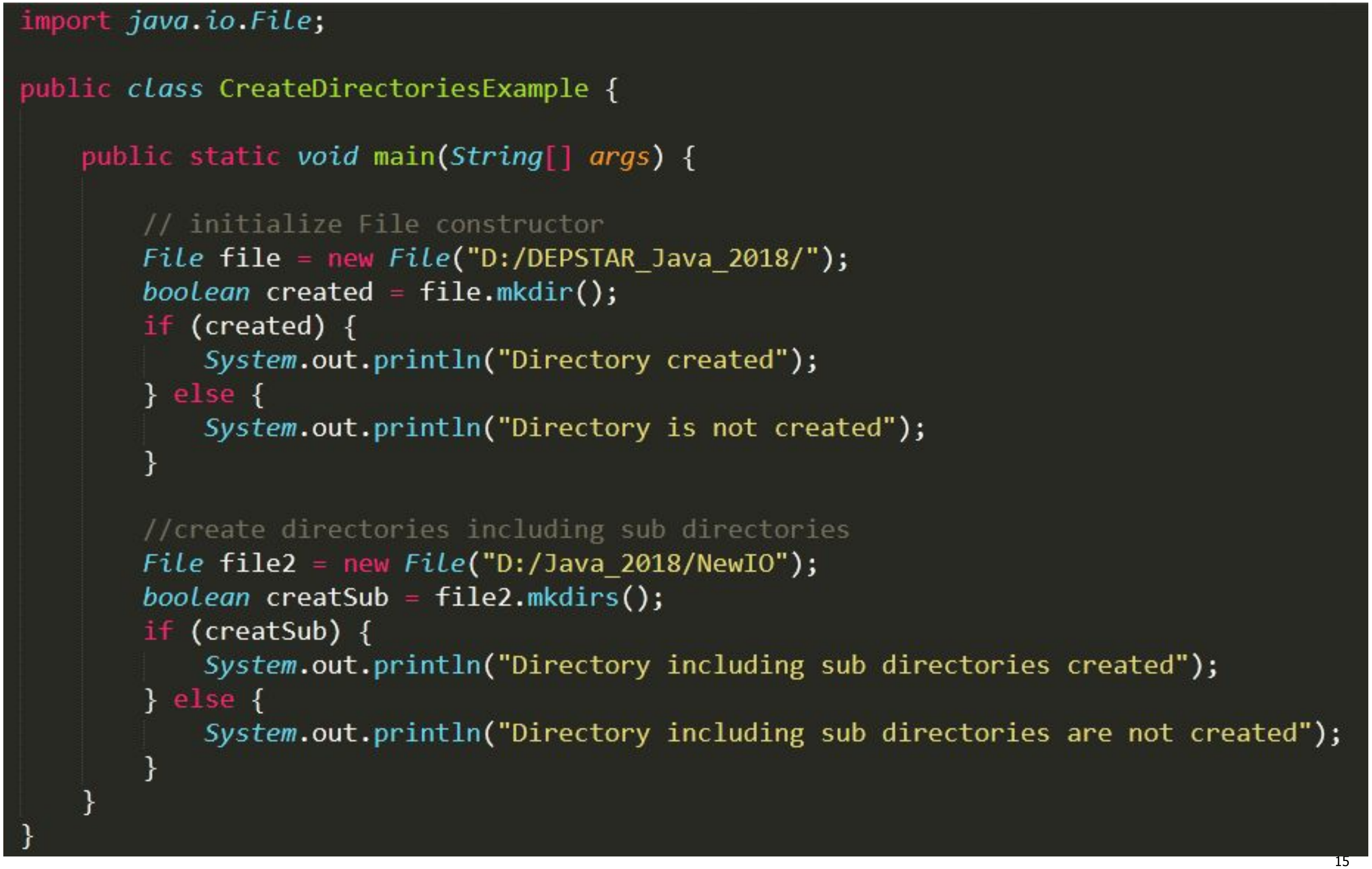


```java
import java.io.File;
import java.io.IOException;

public class AbsoluteAndCanonicalPathExample {

    public static void main(String[] args) throws IOException {
        File file = new File("/Java_2018/io/Myfile.txt");
        File file1 = new File("/Java_2018/../Myfile.txt");

        System.out.println("Absolute Path : " + file.getAbsolutePath());
        System.out.println("Canonical Path : " + file.getCanonicalPath());

        System.out.println("Absolute Path : " + file1.getAbsolutePath());
        System.out.println("Canonical Path : " + file1.getCanonicalPath());

    }

}
```



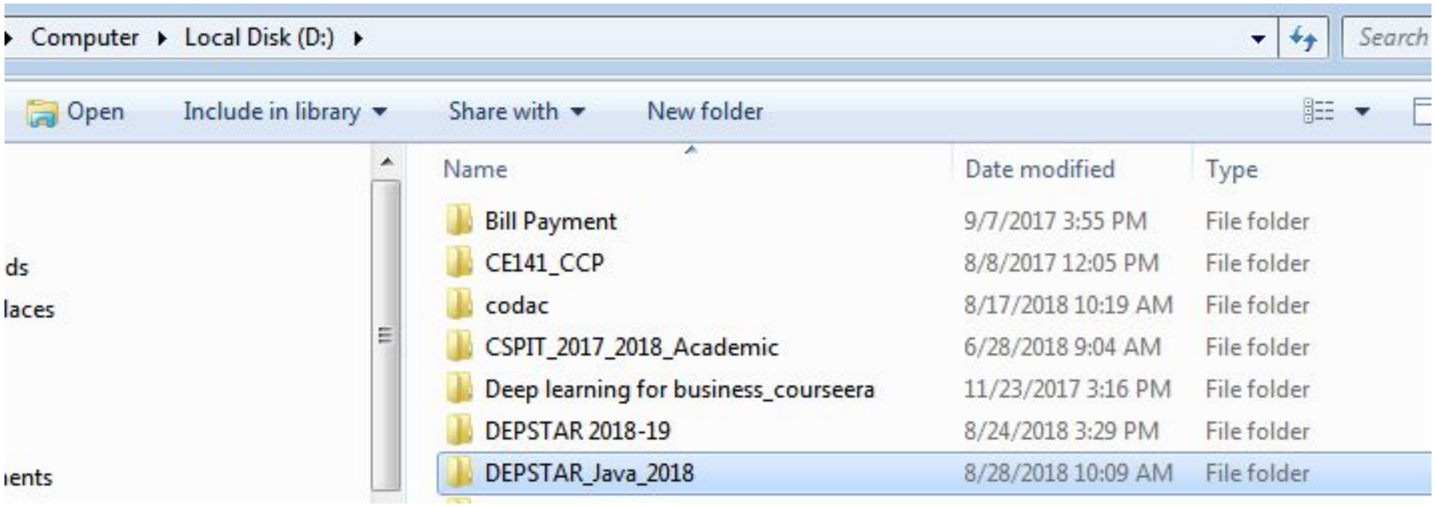
```
D:\Java_2018\io>javac AbsoluteAndCanonicalPathExample.java

D:\Java_2018\io>java AbsoluteAndCanonicalPathExample
Absolute Path : D:\Java_2018\io\Myfile.txt
Canonical Path : D:\Java_2018\io\Myfile.txt
Absolute Path : D:\Java_2018\..\Myfile.txt
Canonical Path : D:\Myfile.txt
```

Create Directories

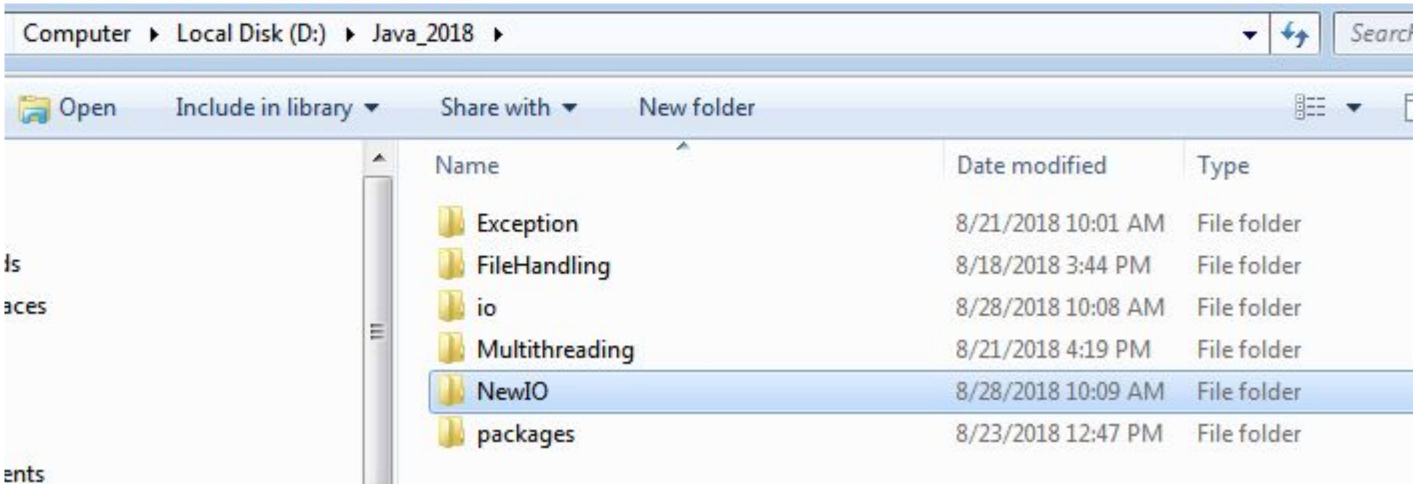


```java
import java.io.File;

public class CreateDirectoriesExample {

    public static void main(String[] args) {

        // initialize File constructor
        File file = new File("D:/DEPSTAR_Java_2018/");
        boolean created = file.mkdir();
        if (created) {
            System.out.println("Directory created");
        } else {
            System.out.println("Directory is not created");
        }

        //create directories including sub directories
        File file2 = new File("D:/Java_2018/NewIO");
        boolean creatSub = file2.mkdirs();
        if (creatSub) {
            System.out.println("Directory including sub directories created");
        } else {
            System.out.println("Directory including sub directories are not created");
        }
    }
}
```

Output


Computer ▸ Local Disk (D:) ▸

Name	Date modified	Type
Bill Payment	9/7/2017 3:55 PM	File folder
CE141_CCP	8/8/2017 12:05 PM	File folder
codac	8/17/2018 10:19 AM	File folder
CSPIT_2017_2018_Academic	6/28/2018 9:04 AM	File folder
Deep learning for business_courseera	11/23/2017 3:16 PM	File folder
DEPSTAR 2018-19	8/24/2018 3:29 PM	File folder
DEPSTAR_Java_2018	8/28/2018 10:09 AM	File folder

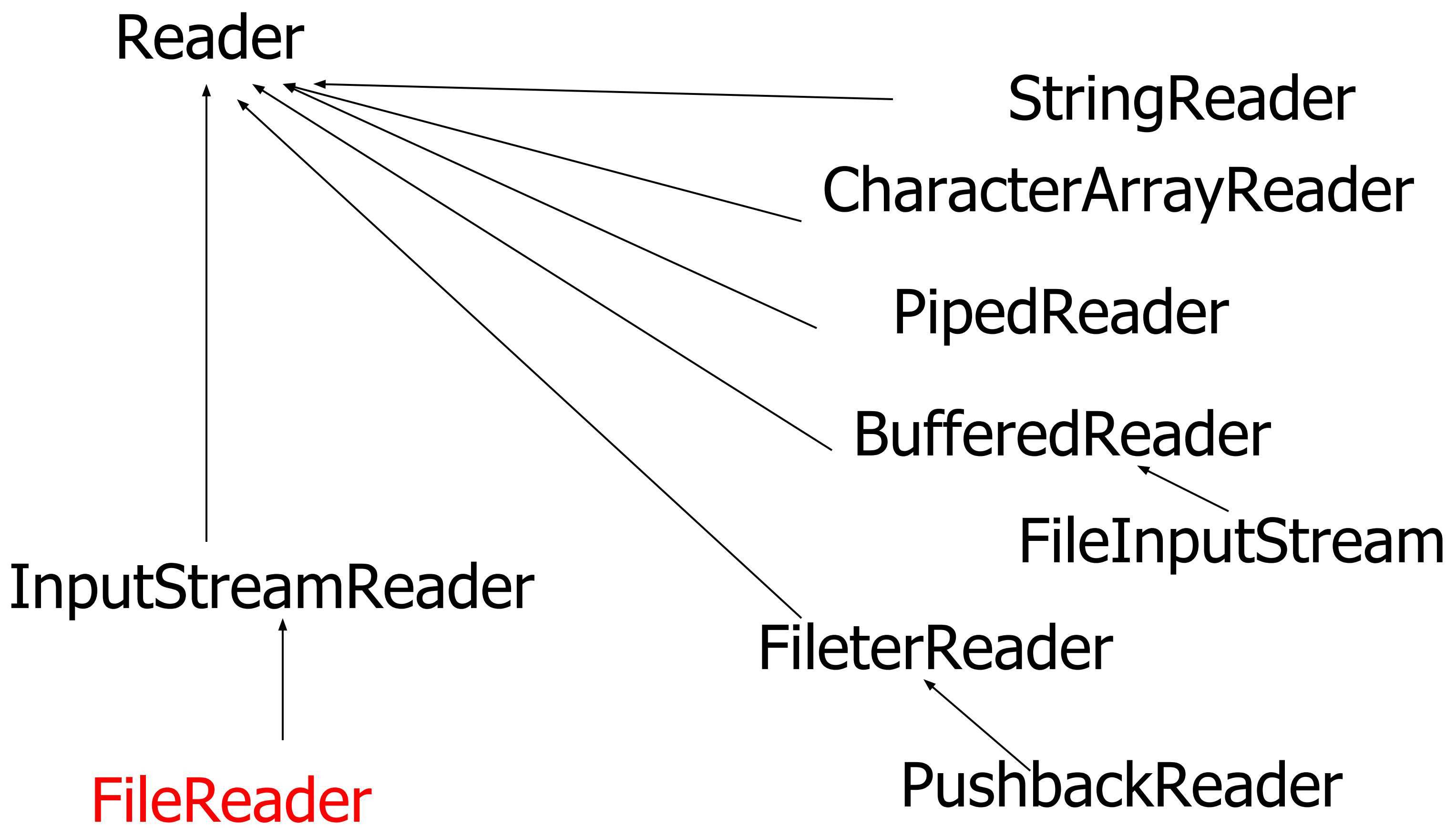

Computer ▸ Local Disk (D:) ▸ Java_2018 ▸

Name	Date modified	Type
Exception	8/21/2018 10:01 AM	File folder
FileHandling	8/18/2018 3:44 PM	File folder
io	8/28/2018 10:08 AM	File folder
Multithreading	8/21/2018 4:19 PM	File folder
NewIO	8/28/2018 10:09 AM	File folder
packages	8/23/2018 12:47 PM	File folder

Reading and Writing Characters

- As pointed out earlier, subclasses of Reader and Writer implement streams that can handle characters.
- The two subclasses used for handling characters in file are:
  - FileReader
  - FileWriter
- While opening a file, we can pass either file name or File object during the creation of objects of the above classes.

Reader Class Hierarchy



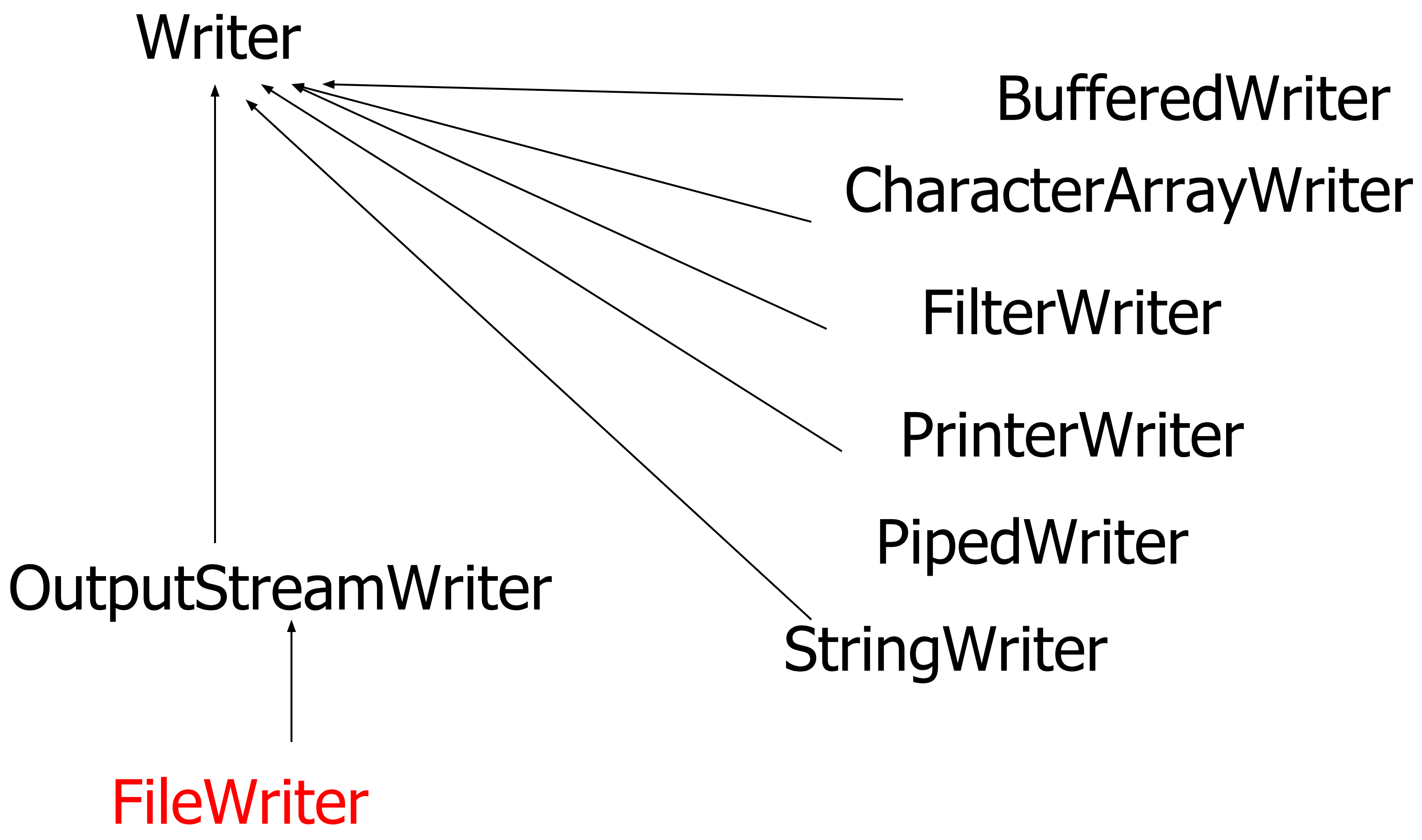
Reader - operations

public int read()	Reads a character and returns as a integer 0-255
public int read(char[] buf, int offset, int count)	Reads and stores the characters in *buf* starting at *offset*. *count* is the maximum read.
public int read(char[] buf)	Same as previous *offset*=0 and *length*=*buf.length*()
public long skip(long count)	Skips *count* characters.
public boolean()	Returns true if the stream is ready to be read.
public void close()	Closes stream

Reader - example

```
import java.io.*;
public class CountSpace {
        public static void main (String[] args)
                throws IOException
        {
                Reader in;  // in can also be FileReader
                in = new FileReader("FileIn.txt");
                int ch, total, spaces;

                spaces = 0;

                for (total = 0 ; (ch = in.read()) != -1; total++){
                        if(Character.isWhitespace((char) ch))
                        {
                                spaces++;
                        }
                }
                System.out.println(total + " chars " + spaces + " spaces ");
        }
}
```

Writer Class Hierarchy



Byte Output Streams - operations

public abstract void write(int ch)	Write *ch* as characters.
public void write(char[] buf, int offset, int count)	Write *count* characters starting from *offset* in *buf.*
public void write(char[] buf)	Same as previous *offset=0* and *count = buf.length()*
public void write(String str, int offset, int count)	Write *count* characters starting at *offset* of *str.*
public void flush()	Flushes the stream.
public void close()	Closes stream

Copying Characters from Files

- Write a Program that copies contents of a source file to a destination file.
- The names of source and destination files is passed as command line arguments.
- Make sure that sufficient number of arguments are passed.
- Print appropriate error messages.

FileCopy.java

```
import java.io.*;
public class FileCopy {
        public static void main (String[] args)
        {
                if(args.length != 2)
                {
                        System.out.println("Error: in sufficient arguments");
                        System.out.println("Usage - java FileCopy SourceFile DestFile");
                        System.exit(-1);
                }
                try {
                FileReader srcFile = new FileReader(args[0]);
                FileWriter destFile = new FileWriter(args[1]);

                int ch;
                while((ch=srcFile.read()) != -1)
                        destFile.write(ch);
                srcFile.close();
                destFile.close();
                }
                catch(IOException e)
                {
                        System.out.println(e);
                        System.exit(-1);
                } }
}
```

Runs and Outputs

- Source file exists:
  - java FileCopy FileIn.txt Fileout.txt
- Source file does not exist:
  - java FileCopy abc Fileout.txt

    java.io.FileNotFoundException: abc (No such file or directory)
- In sufficient arguments passed
  - java FileCopy FileIn.txt

    Error: in sufficient arguments

    Usage - java FileCopy SourceFile DestFile

Do the exercise

- Write a program to print the all files and directory name as well as total count.

- Hint: use File class methods
- 1. list()
- 2. isFile()
- 3. isDirectory()

Limitation of FileWriter

Buffered Streams

- Java supports creation of buffers to store temporarily data that read from or written to a stream. This process is known as *buffered I/O* operation.
- Buffered stream classes – BufferedInputStream, BufferedOutputStream, BufferedReader, BufferedWriter buffer data to avoid every read or write going to the stream.
- These are used in file operations since accessing the disk for every character read is not efficient.

Buffered Streams

- Buffered character streams understand lines of text.
- BufferedWriter has a newLine method which writes a new line character to the stream.
- BufferedReader has a readLine method to read a line of text as a String.

BufferedReader - example

- Use a BufferedReader to read a file one line at a time and print the lines to standard output

```
import java.io.*;

class ReadTextFile {
	public static void main(String[] args)
		throws FileNotFoundException, IOException
	{
		BufferedReader in;
		in = new BufferedReader( new FileReader(“Command.txt”));
		String line;
		while (( line = in.readLine())  !=  null )
			{
				System.out.println(line);
			}
	}
}
```

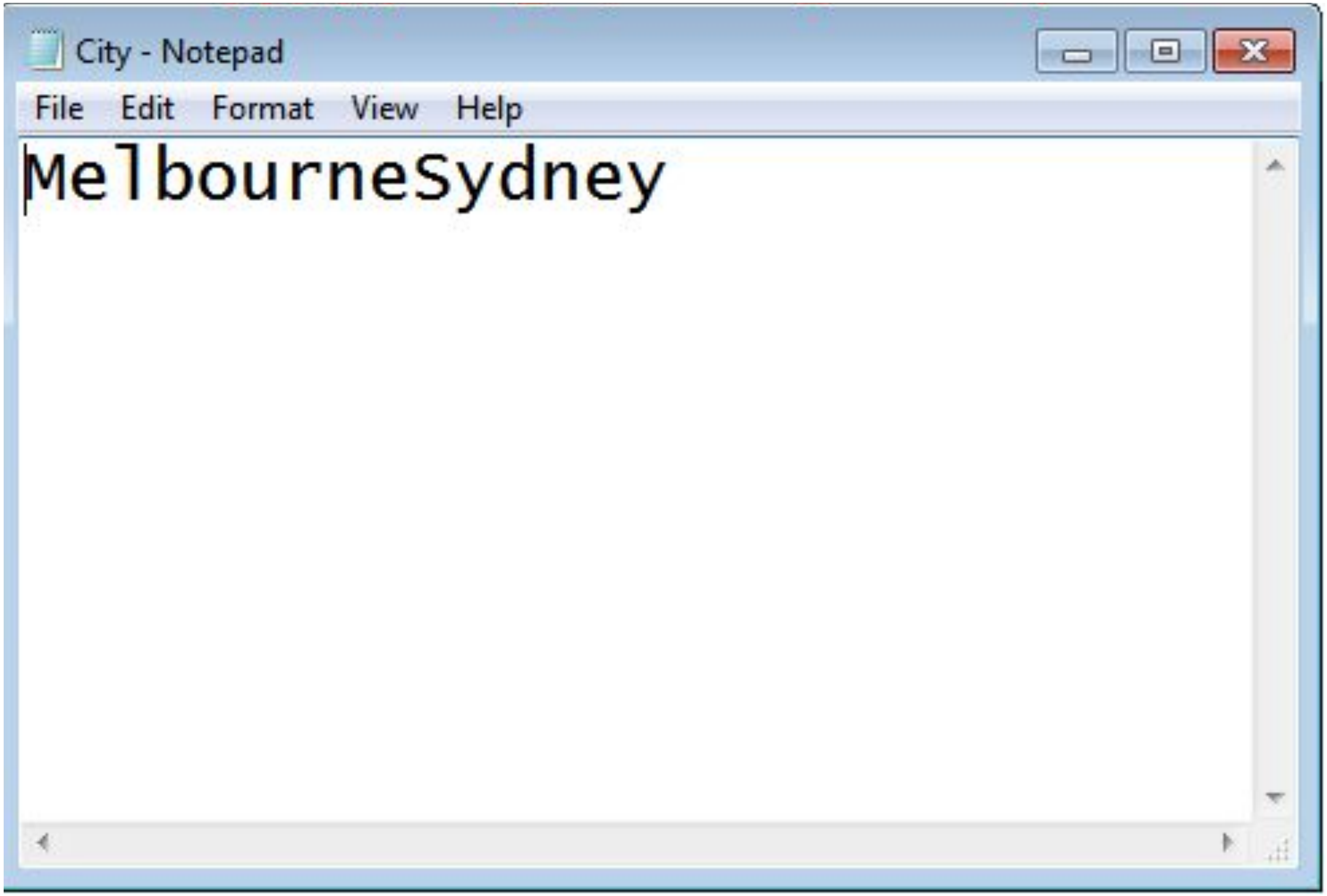
Reading/Writing Bytes

- The FileReader and FileWriter classes are used to read and write 16-bit characters.
- As most file systems use only 8-bit bytes, Java supports number of classes that can handle bytes. The two most commonly used classes for handling bytes are:
  - FileInputStream (discussed earlier)
  - FileOutputStream

Writing Bytes - Example

```
public class WriteBytes {

    public static void main (String[] args)
     {
       byte cities[] = {'M', 'e', 'l', 'b', 'o', 'u', 'r', 'n', 'e', '\n', 'S', 'y','d', 'n', 'e', 'y', '\n‵ };

            FileOutputStream outFile;
            try{
                outFile = new FileOutputStream("City.txt");
                outFile.write(cities);
                outFile.close();
            }
            catch(IOException e)
            {
                System.out.println(e);
                System.exit(-1);
            }
       }}
```

Output



```
MelbourneSydney
```

Summary

- All Java I/O classes are designed to operate with Exceptions.
- User Exceptions and your own handler with files to manger runtime errors.
- Subclasses FileReader / FileWriter support characters-based File I/O.
- FileInputStream and FileOutputStream classes support bytes-based File I/O.
- Buffered read operations support efficient I/O operations.