Fundamentals of Programming 2

Lecture 10

Aurelia Power, TU Dublin – Blanchardstown Campus, 2019

*** Notes based on the Java Oracle Tutorials (2018), Deitel & Deitel (2015), and Horstman (2013)**

Object References and Access Modifiers Recap

- Object reference variables **store the address of the object** (or references) , while primitive variable store the actual values;

- When you copy an object reference, both the original and the copy refer to the same object, but when you copy a primitive value the original value and the copy are independent.

- **REMEMBER:** make your instance variables **private**, while allowing them to be accessed only through the **public interface**: the getters and the setters.
- **Private access modifier:** members of a class (variables and methods) cannot be directly accessed or invoked outside the class where they have been declared.
- **Public accessed modifier**: members of a class (variables and methods) can be directly accessed or invoked outside the class where they have been declared.

Reference types	Primitive types
Unlimited number of reference types, as they are defined by the user.	Consists of boolean and numeric types: char, byte, short, int, long, f loat, and double.
Memory location stores a reference to the data.	Memory location stores actual data held by the primitive type.
When a reference type is assigned to another reference type, both will point to the same object.	When a value of a primitive is assigned to another variable of the same type, a copy is made.
When an object is passed into a method, the called method can change the contents of the object passed to it but not the address of the object.	When a primitive is passed into a method, only a copy of the primitive is passed. The called method does not have access to the original primitive value and therefore cannot change it. The called method can change the copied value.

Information hiding - RECAP

```
public Car(String aMake, String aRegNo,
        double anEngSize){
    if(!aMake.equals(""))
        make = aMake;
    else
        make = "some make";
    if(!aRegNo.equals(""))
        regNo = aRegNo;
    else
        regNo = "some registration";
    if(anEngSize >= 0.1 && anEngSize <= 7)
        engSize = anEngSize;
    else
        engSize = 0.1;
}
```

```
public Car(String aMake, String aRegNo,
        double anEngSize){
    if(aMake.equals(""))
        make = "some make";
    else
        make = aMake;
    if(aRegNo.equals(""))
        regNo =  "some registration";
    else
        regNo = aRegNo;
    if(anEngSize < 0.1 || anEngSize > 7)
        engSize = 0.1;
    else
        engSize = anEngSize;
}
```

• To ensure that *make* and *regNo* will never take the value of an empty string and that *engSize* will never take a value smaller than 0.1 or greater than 7 when you create Car objects, you can use either of the above modified constructors of the Car class (which work exactly the same).

•What would happen if you were to have both constructors in the same class??

- It would not compile, because they would have the same argument list, so they would not be overloaded;

Information hiding - RECAP

```
public void setMake(String aMake){
    if(aMake.equals(""))
        make = "some make";
    else
        make = aMake;
}
public void setRegNo(String aRegNo){
    if(aRegNo.equals(""))
        regNo = "some registration";
    else
        regNo = aRegNo;
}
```

```
public void setMake(String aMake){
    if(!aMake.equals(""))
        make = aMake;
    else
        make = "some make";
}
public void setRegNo(String aRegNo){
    if(!aRegNo.equals(""))
        regNo = aRegNo;
    else
        regNo = "some registration";
}
```

```
public void setEngSize(double anEngSize){
    if(anEngSize >= 0.1 && anEngSize <= 7)
        engSize = anEngSize;
    else
        engSize = 0.1;
}
```

```
public void setEngSize(double anEngSize){
    if(anEngSize < 0.1 || anEngSize > 7)
        engSize = 0.1;
    else
        engSize = anEngSize;
}
```

- To ensure that *make* and *regNo* will never take the value of an empty string and that *engSize* will never take a value smaller than 0.1 or greater than 7 when you modify an existing Car object, you can use either of the above modified setters of the Car class (which work exactly the same).

- So you can pick any one of the 2 versions for each setter to place in your class, but not both, as it would not compile.

String - RECAP

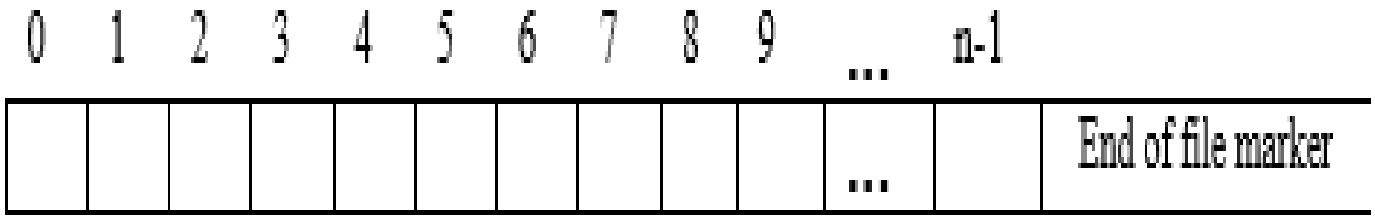
- Check for equality: equals and equalsIgnoreCase methods
- Check whether it contains a sequence of characters: contains method
- Checks whether a string starts with/ends with a certain sequence of characters: startsWith and endsWith metthods
- Replace a sequence of characters: replace method
- Get rid of begin and end spaces: trim method
- Get the index of character or sequence of characters: indexOf and lastIndexOf methods
- Get a subsequence of characters: substring method
- Get a single character: charAt method
- String Comparisons: compareTo and compareToIgnoreCase methods
- The number of characters in a string: length method
- Case conversion: toUpperCase and toLowerCase methods.

Today

- Streams
- Text files vs Binary files
- Reading text files
- Writing to text files

Files

- **Files** are means of storing data for long term retention;
- Data saved/stored in files is called **persistent data;**
- Java typically views a file as a **sequence of bytes**
- Each file ends with either an **end-of-file marker**, or a specific byte number;



Text Files

- **Text files** are accessed **sequentially**;
- They are organised into a sequence of lines separated by a line separator (a character determined by the underlying platform/OS);
- They are read or written character by character, word by word, or line by line;
- They are managed in text editors;
- They store ASCII text values;
- They are meant to be read by us/text editors.

Examples: .java files

.txt files

Binary Files

- Binary files can store **numeric data**, **but also strings** - all in binary format (the same format as it is stored in computer memory, but data are **written to disk)**;
- The read/write programs must use the same data format when processing the file;
- They cannot be read by us/text editors;
- They can be:

1. **Sequential binary files** – accessed sequentially from start to finish; new items can only be inserted at the end;
2. **Random access binary files** – allows us access anywhere in the file ➔ you can also insert items not only at the end, but also at the beginning, middle, or any other position.

Example: .class files (meant to be read and processed by the JVM);

.dat files

Files in java

- In java, support for files and file processing is found in **java.io package** (and other packages, but we only cover java.io package);
- The **File** class is used to create objects that represent files;
- File objects by themselves do NOT open files/streams, or read or write data;
- File objects are used with other classes to provide file processing operations;
- The File class has **several overloaded constructors** (check the API …);
- The most common constructor takes in a **String** that specifies the path and the name of the file:

```
File file = new File("C:\\users\\someFile.txt");
```

- The file class has many useful methods…check API

File class – useful methods

`boolean exists()`	Returns `true` if the file or directory represented by the `File` object exists; `false` otherwise.
`boolean isFile()`	Returns `true` if the name specified as the argument to the `File` constructor is a file; `false` otherwise.
`boolean isDirectory()`	Returns `true` if the name specified as the argument to the `File` constructor is a directory; `false` otherwise.
`boolean isAbsolute()`	Returns `true` if the arguments specified to the `File` constructor indicate an absolute path to a file or directory; `false` otherwise.
`String getAbsolutePath()`	Returns a `String` with the absolute path of the file or directory.
`String getName()`	Returns a `String` with the name of the file or directory.
`String getPath()`	Returns a `String` with the path of the file or directory.
`String getParent()`	Returns a `String` with the parent directory of the file or directory (i.e., the directory in which the file or directory is located).
`long length()`	Returns the length of the file, in bytes. If the `File` object represents a directory, an unspecified value is returned.

- An **absolute path** contains all directories, starting at the root directory;
- A **relative path** starts with the directory in which the application resides;

```java
import java.io.File;

/**
 * @author Aurelia Power
 *
 */
public class FileMethodsDemo {
    public static void main(String[] args) {
        //create the file object using the full path and name of the file
        File file = new File("C:\\Users\\Aurelia Power\\Desktop\\files\\names.txt");

        //check to see whether the file exists on your computer
        System.out.println(file.exists()); //true on my computer

        //check to see if it is directory
        System.out.println(file.isDirectory());//false, because it is not a folder

        //check to see if it is a file
        System.out.println(file.isFile()); // true, because it is a file

        //retrieve the absolute path of the file
        System.out.println(file.getAbsolutePath()); //C:\Users\Aurelia Power\Desktop\files\names.txt

        //set permissions so you can only read it in this program
        //we cannot write to it or modify its content in this program
        file.setReadOnly();

        //check to see if you can write to it/modify its contents
        System.out.println(file.canWrite()); //false
    }//end main
}//end class
```

NOTE: when you pass the path of the file you need to escape the backward slash using the escape character: another backward slash

Quiz on Files: True or False?

- Text files are accessed sequentially

true

- Binary files can be accessed randomly, but not sequentially.

False - binary files can be accessed both ways

- Binary files can be managed using a text editor.

False – binary files are written in binary format, not text;

- File objects don't open read/write streams

True

- The file: C\:Users\Aurelia\myJava.java is not a text file

False – java files are text files

- The following code: File file = new File("C\:Users\Aurelia\myJava.java"); creates a file object

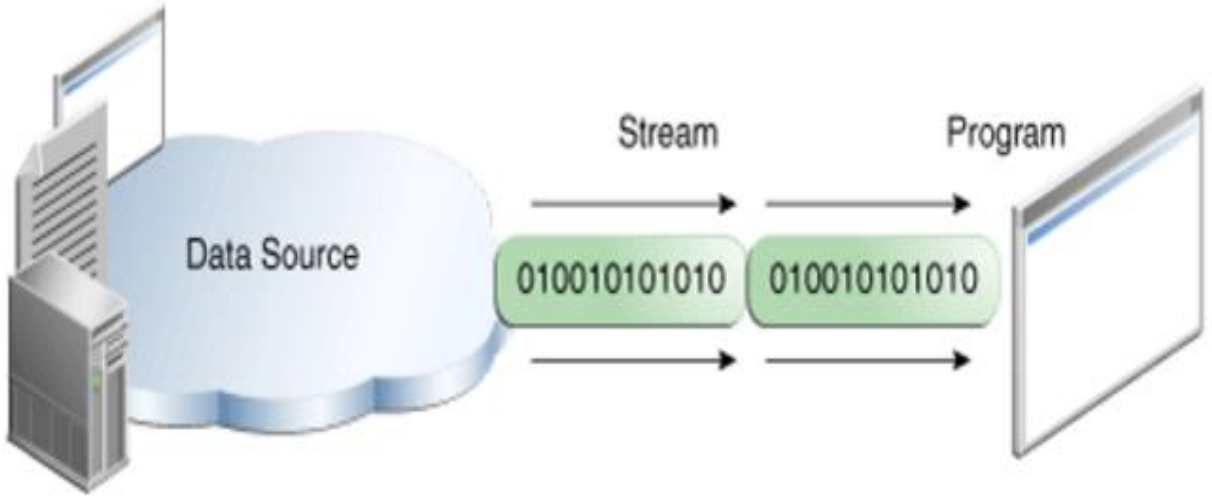
False – it does not compile; to fix it you need to escape the \

- Assuming the file object is already created appropriately using C\:Users\Aurelia\myJava.java, the following code file.exists(); returns the string "true"

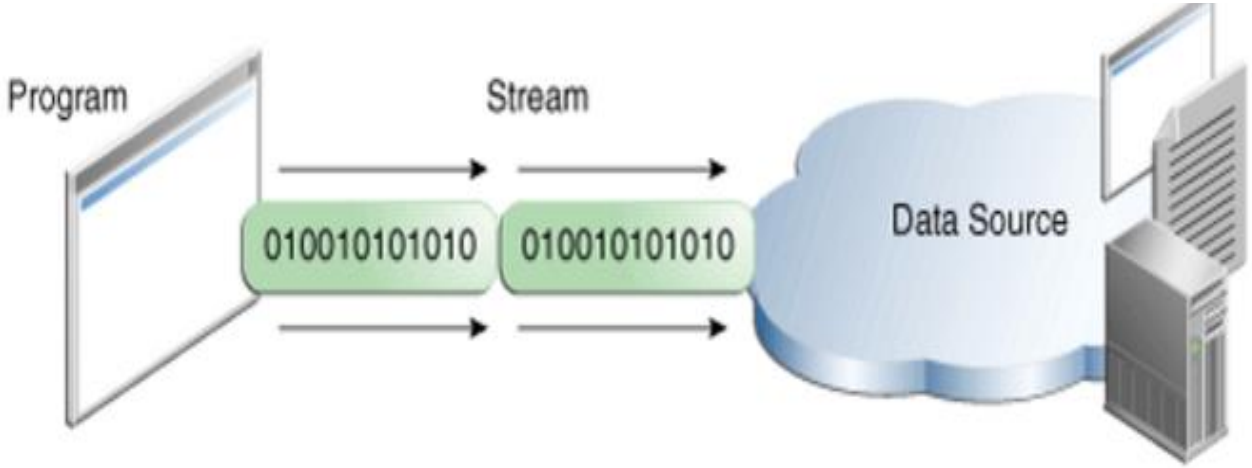
False – exists method returns a boolean, not a String

Streams

- **Streams** are ordered sequences of data;
- They represent **input sources** or **output destinations**, such as: files, devices, other programs, memory.
- In java, streams support **various data**: simple bytes, primitive data types, and even objects.
- An **input stream** is used to read data from a source into a program.
- An **output stream** is used to write data to a destination source from a program.



Reading information into a program.



Writing information from a program.

Source: oracle - java tutorials

Files and Streams

- We need appropriate **IO streams** to read/write to various types of files;
- We use **character streams** for text files;
- We use **byte streams** for binary files;
- If you try to use a byte stream to read a character stream and vice versa, you'll get unexpected results;

Character streams	Byte streams
Meant for reading or writing to character- or text-based I/O such as text files, text documents, XML, and HTML files.	Meant for reading or writing to binary data I/O such as executable files, image files, and files in low-level file formats such as `.zip`, `.class`, `.obj`, and `.exe`.
Data dealt with is 16-bit Unicode characters.	Data dealt with is bytes (i.e., units of 8-bit data).
Input and output character streams are called *readers* and *writers*, respectively.	Input and output byte streams are simply called *input streams* and *output streams*, respectively.
The abstract classes of `Reader` and `Writer` and their derived classes in the `java.io` package provide support for character streams.	The abstract classes of `InputStream` and `OutputStream` and their derived classes in the `java.io` package provide support for byte streams.

Character Streams and Text Files

- **Character Streams** are suited for read/write operations with text files;
- Support is provided in **java.io package**
- Typically, we use the **Reader** and **Writer** abstract classes and their respective subclasses.
- Because we deal with situations that sometimes are beyond the ability of the programmer to deal with, such as missing files, we need to ensure that we have a mechanism that will allow us to continue execution or exit without crashing even if they occur.
- In fact, in Java it is compulsory to ensure that you acknowledge the possibility of I/O exceptional situations or provide a recovery mechanism.

Dealing with IOException

- Throws clause

```
public static void main(String arguments[]) throws IOException{
    //code here to open resources that throws IOException and process data
}
```

- Acknowledges that your program may encounter problems, but it does not address them, so your program may still crash if an IOException occurs.

- Traditional try-catch;

```
try{
    //code here to open resources that throws IOException and process data
}catch(IOException e){
    /*code here to deal with exception, such as printing an error message or what went wrong*/
}
```

-This allows your program to recover if there are exceptional situations related to input/output

-But will not close the resources, and they must be closed explicitly, usually in a finally clause;

-It is more complex, as typically close methods also throw IOException, and you will need a nested try-catch inside the finally … so we won't use it (you will cover it next year)

Dealing with IOException

- Try -with –resources-catch:
- we will use this one as it will automatically close resources;
- it is suitable for our small programs
- It is less complex and no need to worry about closing resources once you have opened them in the header of the try-with-resources;

```
try( /*open streams/resources here*/ ){
    //code here to process your data
}catch(IOException e){
     /*code here to deal with exception, such as printing an
    error message or what went wrong*/
}
```

Character Streams and Reading Text Files

- There are various classes that will allow us to read files;
- We will focus on **FileReader** and **BufferedReader** classes;
- **FileReader** class is designed especially to read character files or streams of characters.
- You can use several **read** methods from the FileReader class to read data from a character file
- But the read operation in FileReader is *inefficient*: each invocation of the read method, could cause all the bytes to be read from the file, converted to characters, and then returned.
- So … it is recommended that we use a buffer that holds the bytes from the stream and can be accessed then as needed, without having to re-read the entire stream again and again;
- For this purpose we will use the **BufferedReader** class to wrap it around the FileReader.
- In addition, the BufferedReader class offers **additional functionalities**.

```
FileReader reader = new FileReader (file); //assume file already created
BufferedReader bufferR = new BufferedReader(reader);
```

Character Streams and Reading Text Files

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

/**
 * @author Aurelia Power
 *
 */
public class ReadTextFile {
    public static void main(String[] args) {
        File file = new File("myFile.txt");
        try(FileReader reader = new FileReader(file);
                BufferedReader bufferR = new BufferedReader(reader)){
            String line = "";
            while((line = bufferR.readLine()) != null){
                System.out.println(line);
            }
            /*int ch = 0;
            while((ch = bufferR.read())!= -1){
                System.out.print((char)ch);
            }*/
        }catch(IOException e){
            System.out.println("Error...");
            //e.printStackTrace();
        }
    }

}
```

Character Streams and Reading Text Files

- Steps typically involved in reading a text file:

1. **Create a file** with the path and name of the text file you want to read: File file = File("myFile.txt");
2. **Create a file reader** that will take as argument the previously created file *in the header of try*; this will allow the reading stream to be opened: FileReader reader = new FileReader(file);
3. **Create a buffered reader** that takes as argument the file reader created in step 2 *in the header of try*; this will buffer the input from the specified file, so the read operation would be more efficient: BufferedReader bufferR= new BufferReader(reader);

NOTE: Step 1, 2 and 3 can all be implemented using a single line of code as follows:

```
BufferedReader bufferR= new BufferReader(new FileReader(new
                                File("myFile.txt")));
```

File constructor is invoked first, then FileReader constructor, and finally BufferReader constructor (remember, it starts with the inner most invocation).

Character Streams and Reading Text Files

4. Then use a while loop *inside the try block* **to read each line** that the buffer holds, by invoking the method **readLine()** on the buffer object;

NOTE: We can use here the **read() method** to read each character at time instead of the entire line, but we will no longer work with String (see commented code);

5. We then ensure that, in case an exceptional situation occurs, we provide a way to **recover**, and instead print an appropriate error message *inside the catch block*, such as System.out.println("Error…"); or using various methods from the Exception class to print the exceptions as follows:

e.printStackTrace(); //outputs the stack of causing exceptions

NOTE: because we used try-with-resource, we do not need to explicitly **close the resources**; however, if we were to use a traditional try-catch mechanism, we would have to also close the resource in a finally clause.

Note: if the file *myFile.txt* does not exist, it will throw an IOException which will be caught by the catch block, and it will output Error…

Note: remember to import all the classes used (one by one), or you can use

import java.io.*;

Methods of BufferedReader class

Modifier and Type	Method and Description
void	**close**() Closes the stream and releases any system resources associated with it.
Stream<String>	**lines**() Returns a Stream, the elements of which are lines read from this BufferedReader.
void	**mark**(int readAheadLimit) Marks the present position in the stream.
boolean	**markSupported**() Tells whether this stream supports the mark() operation, which it does.
int	**read**() Reads a single character.
int	**read**(char[] cbuf, int off, int len) Reads characters into a portion of an array.
String	**readLine**() Reads a line of text.
boolean	**ready**() Tells whether this stream is ready to be read.
void	**reset**() Resets the stream to the most recent mark.
long	**skip**(long n) Skips characters.

Methods inherited from class java.io.Reader

read, read

Question time…

- Consider the following code:

```
public static void main (String [] a){
    File file = new File("myTextFile.txt");
    BufferedReader bufferR= new BufferReader(new FileReader(new
                                                File("myFile.txt")));
    String s = "";
    while( (s=bufferR.read()) != null ){
        System.out.println(s);
    } //end while
} //end main
```

What does it output?

It does not compile…

What are the mistakes?

1. We need to deal with the checked IOException, so we need to either declare the main method that it throws an IOException, or else use try-catch;
2. The method read returns an int…so you cannot assign an int value to s because s is declared as a string.

Character Streams and Writing to Text Files

- There are various classes that can be used to write to files.
- We will focus on **FileWriter and BufferedWriter** classes;
- **FileWriter** class is designed especially to write to a character/text file.
- You can use several **write** methods from the FileWriter class to write data to a character file.
- But, similar to file readers, the write operation in FileWriter is *inefficient*: each invocation of the write method would cause characters to be converted into bytes that would then be written immediately to the file, which can be very inefficient
- So … it is recommended that we use a buffer;
- For this purpose we will use the **BufferedWriter** class to wrap it around the FileWriter.
- In addition, the BufferedWriter class offers **additional functionalities**.

```
FileWriter writer = new FileWriter (file); //assume file already created
BufferedWriter bufferW = new BufferedWriter(writer);
```

Character Streams and Writing to Text Files

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

/**
 * @author Aurelia Power
 *
 */
public class WriteToTextFile {
    public static void main(String[] args) {
        //declare the array of names that you want to write to a file
        String[] names = {"Aurelia", "Arthur", "Elena", "John", "Luca", "Maria", "Michael"};
        //the file that you want to write to
        File namesFile =new File("names.txt");
        //open resources in a try
        try(FileWriter writer = new FileWriter(namesFile);
                BufferedWriter bufferW = new BufferedWriter(writer)){
            //use a loop to go through each element of the array
            for(int i = 0; i < names.length; i++){
                /*write each element of the array(the name) to the file,
                  using one of the overloaded write methods*/
                bufferW.write(names[i] + " ");
            }
        }catch(IOException e){//print the stack of errors in case an exception occurs
            e.printStackTrace();
        }//end catch
    }//end main
}//end class
```

Character Streams and Reading Text Files

- Steps typically involved in writing to a text file:

1. **Create a file** with the path and name of the text file you want to write to: File namesFile = File("names.txt");
2. **Create a file writer** that will take as argument the previously created file *in the header of try*; this will allow the writing stream to be opened: FileWriter writer = new FileWriter(namesFile);
3. **Create a buffered writer** that takes as argument the file writer created in step 2 *in the header of try*; this will buffer the output, so the write operation is much more efficient: BufferedWriter bufferW = new BufferWriter(writer);

NOTE: Step 1, 2 and 3 can be implemented in a single line of code as follows:

```
BufferedWriter bufferW = new BufferWriter(new FileWriter(new
                                    File("names.txt")));
```

Character Streams and Reading Text Files

4. Then use a **for loop** *inside the try block* to go through each element of the array and write to the file using one of the overloaded **write** methods – in this case the one that takes as argument a String;

NOTE: In this step, you can use the **newLine** method to write each name on a different line: bufferW.newLine(); or you can simply add \n to the string passed as argument: bufferW.write(names[i] + "\n");

5. We then ensure that, in case an exceptional situation occurs, we provide a way to **recover from it**, and print the stack of exceptions or an appropriate error message *inside the catch block*.

NOTE: because we used try-with-resource, we do not need to explicitly **close the resources**; however, if we were to use a traditional try-catch mechanism, we would have to also close the resource in a finally clause.

Note: if the file *names.txt* does not exist, the file will be created when opening the write stream, unlike when opening a read stream, in which case will throw an exception;

Note: remember to import all the classes used (one by one), or you can use import java.io.*;

Methods of BufferedWriter class

Modifier and Type	Method and Description
void	**close**() Closes the stream, flushing it first.
void	**flush**() Flushes the stream.
void	**newLine**() Writes a line separator.
void	**write**(char[] cbuf, int off, int len) Writes a portion of an array of characters.
void	**write**(int c) Writes a single character.
void	**write**(**String** s, int off, int len) Writes a portion of a String.

Methods inherited from class java.io.Writer

append, append, append, write, write

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Question time…

- Consider the following code (assume that the file *myTextFile.txt* does not exist):

```
public static void main (String [] a){
        File file = new File("myTextFile.txt");
        try(BufferedWriter bw = new BufferWriter(new FileWriter(new
                                                    File("myFile.txt")))){
            for(int k = 0; k < 10; k+=2){
               bw.write(k + " ");
            }
        }catch(IOException){
            e.printStackTrace();
        }
} //end main
```

Does it compile?

It does compile fine…

Does it throw an IOException because the *myTextFile.txt* does not exist?

No, the file will be created automatically.

What will be the contents of the file?

0 2 4 6 8