final, finally, finalise
not examinable.

Lecture: File I/O (Part 1): Text files

File I/O

Data stored in variables is lost when a program ends

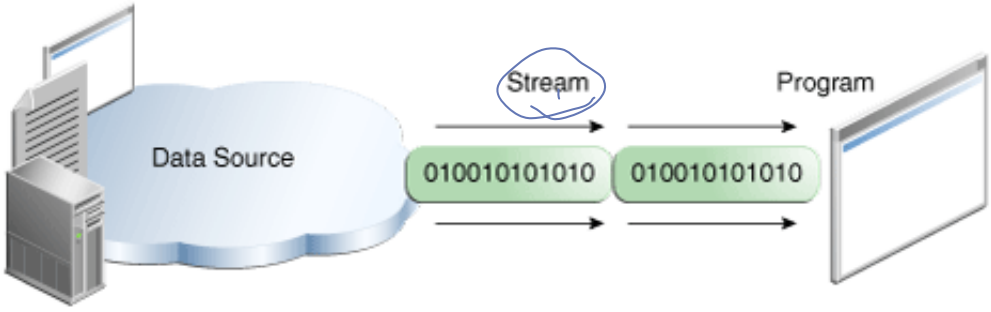
How can we save data, and load it back some time later?

We need to store it in a file, either on disk or on an SSD. This is called Input/Output, or I/O.

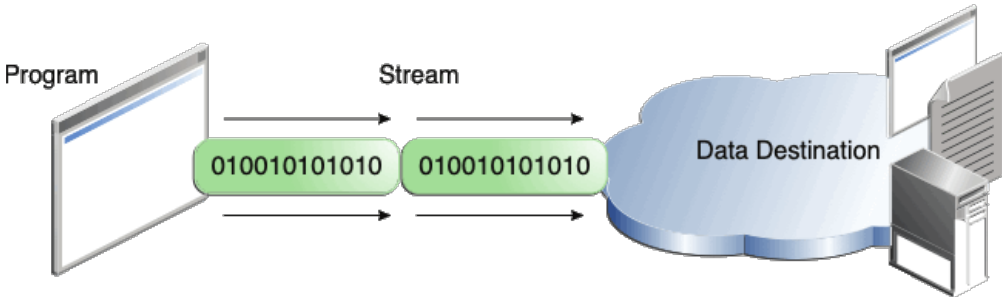
> Network connections are also I/O. The process of reading and writing to them is similar in some ways, but more complicated, and outside the scope of this course.

To communicate with the outside world, java uses *stream* objects.

- If the data flows into a program, then the stream is called an *input stream*.



- If the data flows out of a program, then the stream is called an *output stream*.



The name "stream" comes because a stream represents a file as a sequence of bytes one after the other. These are then interpreted by the program as characters, integers, strings or such like. However, because streams are sequential, data structures such as trees must be "serialized" (turned into a sequence of bytes) before they can be written to a stream.

System streams

You have already come across streams in the first lesson. `System.out` is a stream which, by default, sends data to the screen.

```
System.out.println("Output stream");
```

You can also read in from the keyboard using the standard stream `System.in`.

```
import java.util.Scanner;

class Echo {
    public static void main (String[] args) {
        System.out.println("Type a string, and I will echo it");

        Scanner keyboard = new Scanner(System.in); // <=======
        System.out.println(keyboard.nextLine());
    }
}
```

Text and binary files

All data in a computer is stored as "numbers", but those numbers can be *interpreted* in many ways.

An important example is "text files", in which the numbers are interpreted as letters or other text characters.

A computer would represent the string "Hi" as the numbers 72 and 105.

In a text file, numbers are represented as sequences of characters, so 1000 would be represented as '1', '0', '0', '0', which would in turn be represented as the numbers 49, 48, 48, 48.

In a binary file, the number 1000 be stored in a more efficient way, the way variables are stored in memory.

Text files

Text files are sequences of characters (letters, punctuation etc.), typically designed to be read and modified by humans, using a general-purpose text editor.

They're sometimes called ASCII files, because the mapping between characters as their numeric values is based on the ASCII scheme. However, these days they often contain Unicode text, which is an extension of ASCII.

Text file formats are often more portable between applications or types of computers than binary formats.

Binary files

Files called "binary files" are ones that are not intended to be read by people, such as executable files or compressed files. They also include most word processor and spreadsheet files; even though the documents may be intended to be viewed by people, they won't read the files one character at a time.

Binary files are used because they are more efficient to process than text files. The data is stored more similarly to the way it is in memory. Since that is done differently on different types of computer (PCs vs Macs vs mainframes), the files can be incompatible unless the format is deliberately standardized.

Java tries to make its binary files portable. (Non-assessable: It does that by storing all data big-endian, even on little-endian machines.)

Different classes and methods are used to work with the two types of files.

There are many ways to write into the file. but we focus on two: PrintWriter and BufferWriter

Writing to a text file: Print writer

The class `PrintWriter` is a stream class used for writing text to a file.

This class has methods `print` and `println`, just like `System.out` does.

```
import java.io.PrintWriter;
import java.io.FileOutputStream;

class PrintWriterDemo{

    public static void main(String[] args){
        PrintWriter outputStream = null;

        // The process of connecting a stream to a file is called "opening" the file
        // Use PrintWriter constructor with FileName argument
        // "new FileOutputStream(...) creates an object that PrintWriter can use
        outputStream = new PrintWriter(new FileOutputStream("example.txt"));
                                help you to write data    where to write data.
        // If "example.txt" already existed, its contents have been erased.
        // If it did not exist, it has been created.
        // Either way, it is now a zero-length file.

        // Now we can use the PrintWriter object:
        outputStream.print("Hello, world!");

    }
}
```

The above code currently has an error. You will fix it below, when we discuss exceptions.

After you fix the exception (you can either catch it using a `try-catch` block or your main method can throw it using the `throws` keyword) you should be able to run the program.

Opening

Note that in the above example, the `FileOutputStream` object is an anonymous argument. It cannot be accessed except by the `PrintWriter` object. The purpose of the `FileOutputStream` class is to connect a class that outputs to a stream, such as `PrintWriter`, to a particular file in the filesystem.

The process of connecting an output stream to a file is called *opening the file (for writing)*.

- If the file already exists then doing this causes the old contents to be lost, and the file size to be reduced to 0.
- If the file does not exist, then a new, empty file of the specified name is created.

use throws or try/catch block

```
port java.io.PrintWriter;          // the class that formats
port java.io.FileOutputStream;     // the class that actually writes to the fil
port java.io.FileNotFoundException;     // the class that actually writes to th

ass PrintWriterDemo {
  public static void main (String[] args) throws FileNotFoundException {
      PrintWriter outputStream = null;

      // The process of connecting a stream to a file is called "opening" the
      // Use PrintWriter constructor with FileName argument
      // "new FileOutputStream(...) creates an object that PrintWriter can use
      outputStream = new PrintWriter(new FileOutputStream("example.txt"));
```

PrintWriterDemo.java 7:72 Spaces: 4 (Auto) All changes save

```
        // Either way, it is now a zero-length file.

        // Now we can use the PrintWriter object:
        outputStream.print("Hello, world!");
        outputStream.print("Here is an example.");
    }catch(FileNotFoundException ex){
        System.out.println("File example.txt is not four
    }finally{
        outputStream.flush();
    }
}
```

you need to close the writer, and then the data will write into your text.file.

```
    }catch(FileNotFoundException ex){
        System.out.println("File example.txt is not found");

    }catch(Exception ex){
        System.out.println(ex.getMessage());
    }finally{
        System.out.println("In finally block");
        if(outputStream != null){
            System.out.println("not in outputstream  block");
            outputStream.flush();
            outputStream.close();
        }
    }
```

if the exception happen before the creation of outputstream object. It won't do this block of code.

After doing this, the methods print and println can be used to write to the file.

Exceptions

When a text file is opened in this way, a `FileNotFoundException` can be thrown. In this context it actually means that the file could not be *created*. The name was chosen because this type of exception can also be thrown when a program attempts to open a file for reading and there is no such file, and reading is more common than writing.

It is therefore necessary to enclose this code in exception handling blocks.

- The file should be opened inside a try block
- A catch block should catch and handle the possible exception
- The variable that refers to the `PrintWriter` object should be declared outside the block (and initialized to null) so that it is not local to the block

Exercise: Try to run the above code. Which method throws the error? Fix the code by catching the exception (and no other exceptions). The exception is defined in `java.io.FileNotFoundException`. In the `catch` block, print the error message "Could not open example.txt for writing" and exit with exit code 1. Note that the error message says quite clearly what the problem is: it mentions the file name, and the fact that it is writing, not reading, that is not possible.

Exercise: Then run your new code and check that the expected output file is created, and contains the expected text.

Closing

When a program is finished writing to a file, it should always close the stream connected to that file.

```
outputStreamName.close();
```

This allows the system to release any resources used to connect the stream to the file. If the program does not close the file before the program ends, Java will close it automatically, but it is safest to close it explicitly as soon as the writing is finished.

If the system crashes while the file is open, then it may be left in an "inconsistent" state, which means that it won't be able to be read when the computer reboots or, worse, it may contain corrupted data. Keeping the file open also means that other programs, such as system updates, cannot access the file. That is a common reason for an annoying reboot to be required when updating or installing new software.

Buffered writing

Output streams connected to files are usually buffered.

Rather than physically writing to the file as soon as possible, the data is saved in a temporary location (buffer). When enough data accumulates, or when the method `flush` is invoked, the buffered data is written to the file all at once. This is more efficient, since there is usually a significant overhead for each physical write that doesn't depend on the size of the write.

- For hard drives, that is the time it takes for the disk to spin to the right location and the read/write head to get to the right track
- For SSDs, it is a wear-and-tear overhead. There is a limit to the number of times each block in an SSD can be written to. In a simple system, writing a single byte involves reading a block into memory, changing one byte and then writing the block back.

Close and flush

The method `close` invokes the method `flush`, thus ensuring that all the data is written to the file.

If a program relies on Java to close the file, and the program terminates abnormally, then any output that was buffered may not get written to the file. Also, if a program writes to a file and later reopens it to read from the same file, it will have to be closed first anyway.

The sooner a file is closed after finishing writing to it, the less likely it is that there will be a problem.

File names

The rules for how file names should be formed depend on a given operating system (OS), not Java.

When a file name is given to a java constructor for a stream, it is just a string, not a Java identifier (e.g., "`fileName.txt`"). Any suffix used, such as `.txt` has no special meaning to a Java program.

Just as a java object can have multiple references to it, a file can be referred to in multiple ways.

1. It has a real file name used by the operating sytem
2. When it is open, it is referred to by the stream that is connected to the file.

Note: Depending on the OS, multiple programs can have read streams connected to the file. It is even possible (though rare) for one program to have multiple read streams connected to the file. However, only one stream can be connected for writing.

The class `FileOutputStream` is used to create a stream, and connect it to the file with the specified OS file name. The stream "name" is only a temporary name for the file, while the program is running and the file is open. If it is closed and reopened, then the old stream object is destroyed and a new stream object is created.

Exceptions

In slide "Writing to a text file", we met the FileNotFoundException.

When performing file I/O there are many situations in which this or another exception may be thrown.

Many of these exception classes are subclasses of the class IOException. The class IOException is the root class for a variety of exception classes having to do with input and/or output.

These exception classes are all checked exceptions. Therefore, they must be caught or declared in a throws clause for the program to compile.

Unchecked Exceptions

In contrast to these checked exceptions, there are many unchecked exceptions, such as

- `NoSuchElementException`
- `InputMismatchException`
- `IllegalStateException`

Your code will compile even if unchecked exceptions are neither caught nor declared in a throws clause, but if they occur and are not caught then your code will crash.

Pitfall: a `try` block limits the scope of a variable

Since opening a file can result in an exception, it should be placed inside a `try` block.

If the variable for a `PrintWriter` object needs to be used outside that block, then the variable must be declared outside the block. Otherwise it would be local to the block, and could not be used elsewhere. If it were declared in the block and referenced elsewhere, the compiler will generate a message indicating that it is an undefined identifier.

This is not specific to file I/O, but that is a common case in which this pitfall arises.

(short) Appending text to a file

keep appending to the file.

To create a `PrintWriter` object and connect it to a text file for appending, a second argument, set to **`true`**, must be used in the constructor for the `FileOutputStream` object.

```
outputStreamName = new PrintWriter(new FileOutputStream(FileName, true));
//
```

After this statement, the methods `print`, `println` and/or `printf` can be used to write to the file. The new text will be written *after the old text* in the file.

Appending text to a file

Write a program that will read lines of text from the keyboard.

If the line is the empty string, then exit.

If the line is equal to "reset", then set the file "output.txt" to be the empty file.

Otherwise, append the line to the file "output.txt", and read the next line from the keyboard.

If you run the code a second time, it should not overwrite "output.txt" until a line "reset" is entered.

Try it on the text

```
one
reset
two
three
```

and

```
one
two
three
```

If you run the two tests in that order, the final output file should start with the line "two". (Why?)

Useful methods for text file output

If a class has a suitable `toString()` method, and `anObject` is an object of that class, then `anObject` can be used as an argument to `System.out.println`, and it will produce sensible output.

The same thing applies to the methods `print` and `println` of the class `PrintWriter`:

```
outputStreamName.println(anObject);
```

Some methods of the class `PrintWriter`

```
public PrintWriter(OutputStream streamObject)
```

This is the only constructor you are likely to need. There is no constructor that accepts a file name as an argument. If you want to create a stream using a file name, you use

```
        new PrintWriter( new FileOutputStream(fileName));
```

When the constructor is used in this way, a blank file is created. If there already was a file named fileName, then the old contents of the file are lost. If you want instead to append new text to the end of the old file contents, use

```
        new PrintWriter( new FileOutputStream(fileName, true));
```

(For an explanation of the argument `true`, read the subsection "Appending to a Text File".)

When used in either of these ways, the `FileOutputStream` constructor, and so the `PrintWriter` constructor invocation, can throw a `FileNotFoundException`, which is a kind of `IOException`.

If you want to create a stream object using an object of the class `File`, you can use a `File` object in place of the `fileName`. (The `File` class is covered in Section 10.3 of the text book. It is mentioned here so that you will have a more complete reference in this slide, but you can ignore the reference to the class `File` until after you've read that section.)

```
public void println(argument)
```

The `argument` can be a string, character, integer, floating-point number, boolean value, or any combination of these, connected with + signs. (Note that each of these types is converted to a string, and the + is simple string concatenation. If you use `println(2+2)`, you will get 22. To get 4, use `println((2+2))`.) The `argument` can also be any object, although it will not work as desired unless the object has a properly defined `toString()` method. The `argument` is output to the file connected to the stream. After the `argument` has been output, the line ends, and so the next output is sent to the next line.

```
public void print(argument)
```

This is the same as println, except that this method dos not end the line, so the next output will be on the same line.

```
public PrintWriter printf(arguments)
```

This is the same as `System.out.printf`, except that this method sends output to a text file rather than to the screen. It returns the calling object. However, it is usual to use `printf` as if it were a void method.

```
public void close()
```

Closes the stream's connection to a file. This method calls `flush` before closing the file.

```
public void flush()
```

Flushes the output stream. This forces an actual physical write to the fiel of any data that has been buffered and not yet physically written to the file. Normally, you should not need to invoke flush. It is useful, for example, if some other program is reading from the file as it is being written.

Reading from a text file using Scanner

The class `Scanner` can be used for reading from a text file as well as reading from the keyboard. Simply replace the argument `System.in` (to the `Scanner` constructor) with a suitable stream that is connected to the text file.

```
    Scanner StreamObject = new Scanner(new FileInputStream(FileName));
```

Methods of the `Scanner` class for reading input behave the same whether reading from the keyboard or reading from a text file. For example, the `nextInt` and `nextLine` methods.

Here is a simple example.

```
import java.util.Scanner;
import java.io.FileInputStream;
import java.io.FileNotFoundException;

import java.io.FileOutputStream;
import java.io.PrintWriter;

public class TextFileScannerDemo {
    public static void main (String[] args) {
        String filename = "morestuff.txt";

        // Create the file to be read
        createMorestuff ("1 2\n3 4\nEatMyShorts", filename);

        System.out.print  ("I will read three numbers and a line ");
        System.out.println("of text from the file " + filename + ".");

        Scanner inputStream = null;

        try {
            inputStream = new Scanner(new FileInputStream(filename));
        } catch (FileNotFoundException e) {
            System.err.print  ("File " + filename + " was not found ");
            System.err.println("or could not be opened.");
            System.exit(1);
        }
        int n1 = inputStream.nextInt();
        int n2 = inputStream.nextInt();
        int n3 = inputStream.nextInt();

        //To go to the next line.  What happens if this line is omitted?
        inputStream.nextLine();

        String line = inputStream.nextLine();
```

```
        System.out.println("The three numbers read from the file are:");
        System.out.println(n1 + ", " + n2 + ", and " + n3);

        System.out.println("The line read from the file is:");
        System.out.println(line);

        inputStream.close();
    }

    // Create the file to be read above.
    static void createMorestuff (String s, String filename) {
        try {
            PrintWriter p = new PrintWriter(new FileOutputStream(filename));
            p.print(s);
            p.close();
        } catch (Exception e) {
            System.err.println("oops");
            System.exit(1);
        }

    }
}
```

write sth into the file

Testing for the end of a text file with `Scanner`

A program that tries to read beyond the end of a file using methods of the `Scanner` class will cause an exception to be thrown.

However, instead of having to rely on an exception to signal the end of a file, the `Scanner` class provides methods such as `hasNextInt` and `hasNextLine`. These methods can also be used to check that the next
token to be input is a suitable element of the appropriate type.

Exercise: Modify the above code to use `hasNextInt` and `hasNextLine` to check before reading. Modify the input to cause the reads to fail.

Here is another example that uses `hasNextLine`.

do it in loop.

```
import java.util.Scanner;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.io.FileOutputStream;

public class HasNextLineDemo {
    public static void main(String[] args) {
        Scanner inputStream = null;
        PrintWriter outputStream = null;

        createOriginal ("A photon checks into a hotel.\n"
                    + "The clerk asks \"Do you have any luggage?\"\n"
```

```
                    + "The photon replies \"No, I'm travelling light\"\n",
                    "original.txt");

        try {
            inputStream = new Scanner(new FileInputStream("original.txt"));
            outputStream = new PrintWriter(new FileOutputStream("numbered.txt"));
        } catch (FileNotFoundException e) {
            System.err.println("Problem opening files.");
            System.exit(1);
        }

        String line = null;
        int count = 0;

        while (inputStream.hasNextLine()) {
            line = inputStream.nextLine();
            count++;
            outputStream.println(count + " " + line);
        }

        inputStream.close();
        outputStream.close();
    }

        // Create the file to be read above.
    static void createOriginal (String s, String filename) {
        try {
            PrintWriter p = new PrintWriter(new FileOutputStream(filename));
            p.print(s);
            p.close();
        } catch (Exception e) {
            System.err.println("oops");
            System.exit(1);
        }

    }
}
```

Exercise: Read and understand the code above. Modify the original file so that the lines are numbered. Modify the loop to check if each line starts with a number, using `hasNextInt`. If it does, then output the line with the number removed. If the line doesn't start with a number, then prepend the number to the line as it currently does.

Here is one dedicated to `hasNextInt`.

```
import java.util.Scanner;
import java.io.FileInputStream;
import java.io.FileNotFoundException;

import java.io.PrintWriter;
import java.io.FileOutputStream;
```

```
public class HasNextIntDemo {
    public static void main(String[] args) {
        Scanner inputStream = null;
        String filename = "data.txt";

        createData("1 2\n3 4 hi 5\n6 7\n8 9", filename);

        try {
            inputStream = new Scanner(new FileInputStream(filename));
        } catch (FileNotFoundException e) {
            System.err.print  ("File " + filename + " was not found ");
            System.err.println("or could not be opened.");
            System.exit(1);
        }

        int next, sum = 0;
        while (inputStream.hasNextInt()) {
            next = inputStream.nextInt();
            sum = sum + next;
        }
        inputStream.close();

        System.out.println("The sum of the numbers is " + sum);
    }

            // Create the file to be read above.
    static void createData (String s, String filename) {
        try {
            PrintWriter p = new PrintWriter(new FileOutputStream(filename));
            p.print(s);
            p.close();
        } catch (Exception e) {
            System.err.println("oops");
            System.exit(1);
        }
    }
}
```

Exercise: Modify the loop so that, if there isn't a next int, but there is a next line, the rest of the current line is discarded, and the loop continues from the next line. The output should be 40.

Methods in the class `Scanner`

not figure out yet

We have seen several methods in `Scanner`, but here is a more complete list.

```
public Scanner (InputStream streamObject)
```

There is no constructor that accepts a file name as an argument. If you want to create a stream using a file name you can use

```
        new Scanner(new FileInputStream(fileName))
```

When used in this way, the `FileINputStream` constructor, and thus the `Scanner` constructor invocation, can throw a `FileNotFoundException`, which is a kind of `IOException`.

To create a stream connected to the keyboard, use

```
        new Scanner(System.in)
```

```
public Scanner(File fileObject)
```

The File class will be covered later in this lesson. It is mentioned here only so that you will have a more complete reference here, butyou can ignore this entry until after you have read that section.

If you want to create a stream using a file name, you can use

```
        new Scanner(new File(fileName))
```

```
public int nextInt()
```

Returns the next token as an `int`, provided the next token is a well-formed string representation of an `int`.

Throws a `NoSuchElementException` if there are no more tokens.

Throws an `InputMismatchException` if the next token is not a well-formed string representation of an `int`.

Throws an `IllegalStateException` if the `Scanner` stream is closed.

```
public boolean hasNextInt()
```

Returns `true` if the next token is a well-formed string representation of an `int`; otherwise returns `false`.

Throws an `IllegalStateException` if the Scanner stream is closed.

```
public long nextLong()
public boolean hasNextLong()
public byte nextByte()
public boolean hasNextByte()
public short nextShort()
public boolean hasNextShort()
public double nextDouble()
public boolean hasNextDouble()
public float nextFloat()
public boolean hasNextFloat()
```

```
public boolean nextBoolean()
public boolean hasNextBoolean()
```

For the description of these, replace each use of "`int`" by the appropriate type in the above descriptions of `nextInt` and `hasNextInt`.

```
public String next()
```

Returns the next token.

Throws a `NoSuchElementException` if there are no more tokens.

Throws an `IllegalStateException` if the Scanner stream is closed.

```
public boolean hasNextInt()
```

Returns `true` if there is another token. It may wait for a next token to enter the stream.

Throws an `IllegalStateException` if the Scanner stream is closed.

```
public String nextLine()
```

Returns the restu of the current inptu line. Note that the line terminator '\n' is read and discarded; it is not included in the string returned.

Throws a `NoSuchElementException` if there are no more lines.

Throws an `IllegalStateException` if the Scanner stream is closed.

```
public boolean hasNextLine()
```

Returns `true` if there is a next line. It may wait for a next line to enter the stream.

Throws an `IllegalStateException` if the Scanner stream is closed.

```
public Scanner useDelimiter(String newDelimiter)
```

The above methods often mention a "token". A token is the string between a pair of delimiters (or between the start of the file and the first delimiter, or the last delimiter and the end of file).

This function changes the string that acts as the delimiter that separates tokens (words or numbers). It replaces the previous value. See the subsection "Other Input Delimiters" in Chapter 2 of the text book for the details. (You can use this method to set the delimiters to a mor complex pattern than just a single string, but we are not covering that.)

Returns the calling object, but it is usually used as if it were a `void` method.

Reading from a text file using Buffered Reader

The class `BufferedReader` is another stream class that can be used to read from a text file. It is older and less powerful than `Scanner`, but can still be useful.

An object of the class `BufferedReader` has the methods `read` and `readLine`.

A program using BufferedReader will start with a set of import statements

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

Like the classes `PrintWriter` and Scanner, `BufferedReader` has no constructor that takes a file name as its argument. It needs to use another class, `FileReader`, to convert the file name to an object that can be used as an argument to its (the `BufferedReader`) constructor.

A stream of the class `BufferedReader` is created and connected to a text file as follows:

```
BufferedReader readerObject;
readerObject = new BufferedReader(new FileReader(FileName));
```

This opens the file for reading.

After these statements, the methods read and `readLine` can be used to read from the file.

- The `readLine` method is the same method used to read from the keyboard, but in this case it would read from a file.
- The `read` method reads a single character, and returns a value (of type `int`) that corresponds to the character read

Since the read method does not return the character itself, a type cast must be used:

```
char next = (char)(readerObject.read());
```

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

import java.io.PrintWriter;
import java.io.FileOutputStream;

public class TextFileInputDemo {
    public static void main (String[] args) {
```

```
        String filename = "morestuff2.txt";

        createFile("1 2 3\nJack jump over\nthe candle stick.", filename);

        try {
            BufferedReader inputStream
                    = new BufferedReader(new FileReader(filename));
            String line = inputStream.readLine();
            System.out.println("The first line read from this file is:");
            System.out.println(line);

            line = inputStream.readLine();
            System.out.println("The second line read from this file is:");
            System.out.println(line);

            inputStream.close();
        } catch(FileNotFoundException e) {
            System.err.print  ("File " + filename + " was not found ");
            System.err.println("or could not be opened.");
        } catch (IOException e) {
            System.err.println("Error reading from " + filename + ".");
        }
    }

    // Create the file to be read above.
    static void createFile (String s, String filename) {
        try {
            PrintWriter p = new PrintWriter(new FileOutputStream(filename));
            p.print(s);
            p.close();
        } catch (Exception e) {
            System.err.println("oops");
            System.exit(1);
        }
    }
}
```

A program using a `BufferedReader` object in this way may throw two kinds of exceptions.

- An attempt to open the file may throw a `FileNotFoundException` (which in this case has the expected meaning)
- An invocation of `readLine` may throw an `IOException`

Both of these exceptions should be handled.

Methods for the class `BufferedReader`

```
public BufferedReader(Reader readerObject)
```

This is the only constructor you are likely to need. There is no constructor that accepts a file name as an argument. If you want to create a stream using a file name, you use

```
public BufferedReader(new FileReader (fileName))
```

If you want to create a stream object using an object of the class `File`, you can use a `File` object in place of the `fileName`.

When used in either of these ways, the `FileReader` constructor, and so the `BufferedReader` constructor invocation, can throw a `FileNotFoundException`, which is a kind of `IOException`.

```
public String readLine() throws IOException
```

Reads a line of input from the input stream and returns that line If the read goes beyond the end of the file, `null` is returned. (Note that an `EOFException` is not thrown at the end of a file. The end of a file is signalled by returning `null`.)

public int read() throws IOException

Reads a single character from the input stream and returns that character as an `int` value. If the read goes beyond the end of the file, then -1 is returned. Note that the value is returned as an `int`. To obtain a `char`, you must perform a type cast on the value returned. The end of a file is signalled by returning a -1. (All of the "real" characters return a non-negative integer.)

What makes a "character" depends on the `FileReader` object. By default, it uses the OS default. You can provide a second argument such as `StandardCharsets.UTF_8` to force it to use a particular character encoding.

```
public long skip(long n) throws IOException
```

Skips n characters.

```
public void close() throws IOException
```

Closes the stream's connection to a file.

Reading numbers

Unlike the Scanner class, the class `BufferedReader` has no methods to read a number from a text file.

Instead, a number must be read in as a string, and then converted to a value of the appropriate numeric type using one of the wrapper classes.

To read in a single number on a line by itself, first use the method `readLine`, and then use `Integer.parseInt`, `Double.parseDouble`, etc. to convert the string into a number.

If there are multiple numbers on a line, `StringTokenizer` can be used to decompose the string into tokens, and then the tokens can be converted as described above.

Testing for the end of a text file

The method `readLine` of the class `BufferedReader` returns `null` when it tries to read beyond the end of a text file.

- A program can test for the end of the file by testing for the value `null` when using `readLine`.

The method `read` of the class `BufferedReader` returns -1 when it tries to read beyond the end of a text file

- A program can test for the end of the file by testing for the value -1 when using `read`.

It is common to see a construct such as

```
while ((value = br.readLine()) != null) {
    // process value
}
```

That means that a `readLine` is called on a `bufferedReader` object referred to by `br` and the value is assigned to the variable `value`. This is then compared with `null`. If it is equal to `null`, the loop terminates. Otherwise, the iteration of the loop can process the value in variable `value`.

Path names

When a file name (i.e., with no '/' or '\' characters) is used as an argument to a constructor for opening a file, it is assumed that the file is in the same directory or folder as the one in which the program is run.

If it is not in the same directory, a full or relative path name must be given.

A path name not only gives the name of the file, but also the directory or folder in which the file exists.

- A *full path name (also called absolute path)* gives a complete path name, starting from the root directory
- A *relative path name* gives the path to the file, starting with the directory from which the program was run.

The way path names are specified depends on the operating system.

In Unix-like systems (e.g., MacOS, Linux), directory names are separated by a forward slash, '/'. A typical path name that could be used as a file name argument is `"/user/sallyz/data/data.txt"`.

A BufferedReader input stream connected to this file is created as follows:

```
BufferedReader inputStream = new BufferedReader(
                                    new FileReader("/user/sallyz/data/data.txt"));
```

The Windows operating system uses a backslash '\' to separate path components, and optionally has a drive specifier at the start.

A typical Windows path name is "C:\dataFiles\goodData\data.txt".

A BufferedReader input stream connected to this file is created as follows:

```
BufferedReader inputStream = new BufferedReader(
                                    new FileReader ("C:\\dataFiles\\goodData\\data.txt"));
```

Note that a Windows pat must use \\ in place of \, since a single backslash denotes an the beginning of an escape sequence. This is part of the process of parsing the .java file to convert it to a .class file. It does not apply to file names read from a file, such as the keyboard, or to file names constructed some other way.

Problems with escape characters can be avoided altogether by always using Unix conventions when writing a path name. A Java program will accept a path name written in either Unix or Windows format, regardless of the OS on which it is run. This is one of the ways in which Java attempts to be

platform-independent.

An absolute path name starts with either a slash ('/' or '\'), or a drive specifier followed by a slash ("C:\").

A relative path name starts either with a './' for the current directory, a '../' for the parent of the current directory, or a directory name denoting a subdirectory of the current directory.

Nested constructors

Each of the Java I/O library classes serves only one function, or a small number of functions. Normally two or more class constructors are combined to obtain full functionality.

Therefore, expressions with two constructors are common when dealing with Java I/O classes.

```
new BufferedReader(new FileReader("stuff.txt"))
```

Above, the anonymous `FileReader` object establishes a connection with the `stuff.txt` file. However, it provides only very primitive methods for input. For example, it is responsible for handling different file encodings (UTF-8, UTF-16, BIG5 etc.)

The constructor for `BufferedReader` takes this `FileReader` object and adds a richer collection of input methods. This transforms the inner object into an instance variable of the outer object.

Standard input, standard output, standard error

Unix-like operating systems and Windows attach three standard streams to any running program.

- Standard output is used for normal output. By default it goes to the screen, but it can be "redirected" to a file using "java Prog > dest_file".
- Standard error is used to output error messages. By default it also goes to the screen, and even if standard output is redirected, standard error still goes to the screen. On Unix-like systems, standard error can also be redirected using "java Prog 2>& dest_file"
- Standard input is normally used for keyboard input. However, it can be redirected to come from a file using "java Prog < intput_file". More powerfully, it can use the output of another program. The command "java Prog1 | java Prog2" takes the standard output of running Prog1 and uses it as the standard input of Prog2. This is called a "pipe", and Unix shell scripts often have chains of half a dozen piped commands.

Java's access to these three streams is through System.out, System.err and System.in.

As well as the redirection from the command line described above, it is possible to redirect the streams in `System` using the following methods:

```
public static void setOut(PrintStream outStream)
public static void setErr(PrintStream outStream)
public static void setIn(InputStream inStream)
```

Using these methods, any of the three standard streams can be redirected. For example, instead of appearing on the screen, error messages could be redirected to a file.

In order to redirect a standard stream, a new stream object is created. Like other streams created in a program, a stream object used for redirection should be closed after I/O is finished.

Note, standard streams do not need to be closed.

Redirecting System.err:

```
public void getInput() {
    //. . .
    PrintStream errStream = null;
    try {
        errStream = new PrintStream(new FileOuptputStream("errMessages.txt"));
        System.setErr(errStream);
        //. . . Set up input stream and read
    } catch(FileNotFoundException e) {
        System.err.println("Input file not found");
    } finally {
        // . . .
        errStream.close();
    }
}
```

This is useful if we sometimes want to send the output to the operating system's standard error and sometimes want to send it to a file. We can either call or not call the `setErr` depending on a configuration variable.

What have these got to do with nested constructors?

System.in, System.out and System.err are low-level I/O streams. Being able to mix-and-match different sources of streams-of-character, we can use different classes on top of these special streams. Other low-level streams come from networking, and again we get to choose which skin to apply on top of a network connection.

File class

The `File` class is like a wrapper class for file names.

The constructor for the class `File` takes a name (known as the *abstract name*) as a string argument, and produces an object that represents the file with that name.

The `File` object and methods of the class `File` can be used to determine information about the file and its properties.

```
import java.util.Scanner;
import java.io.File;
import java.io.PrintWriter;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;

public class FileClassDemo
{

    public static void main(String[] args)
    {
        Scanner console = new Scanner(System.in);
        String line = null;
        String filename = null;

        System.out.println("Enter a line of text to store:");
        line = console.nextLine();

        System.out.println("Enter a filename you would like to write to:");
        filename = console.nextLine();

        File fileObject = new File(filename);

        while(fileObject.exists())
        {
            System.out.println(filename + " already exists. Enter a different filename:");
            filename = console.nextLine();
            fileObject = new File(filename);
        }

        PrintWriter outputStream = null;
        try
        {
            outputStream = new PrintWriter(new FileOutputStream(filename));
        }
        catch (FileNotFoundException e)
        {
            System.out.println("Could not write file: " + filename);
            System.exit(0);
```

```
        }

        outputStream.println(line);
        outputStream.close();
        System.out.println("Line written to: " + filename);
    }
}
```

Some methods in class `File`

```
public File (String fileName)
```

Constructor. *fileName* can be either a full or a relative path name (which includes the case of a simple file name). *fileName* is referred to as the **abstract path name**.

```
public boolean exists()
```

Tests whether there is a file with the abstract path name.

```
public boolean canRead()
```

Tests whether the program can read from the file. Returns `true` if the file named by the abstract path name exists and is readable. Otherwise, returns `false`. A file may be unreadable if its permissions don't include reading, or if it is locked by another program.

```
public boolean setReadOnly()
```

Sets the file represented by this object to be read only. Returns `true` if successful; otherwise returns `false`.

```
public boolean canWrite()
```

Tests whether the program can write to the file. Returns `true` if the file represented by this object exists and is writable by the program; otherwise returns `false`. A file may be unwritable if its permissions don't include writing, or if it is locked by another program.

```
public boolean delete()
```

Tries to delete the file or directory represented by this object. A directory must be empty to be removed. Returns `true` if it was able to delete the file or directory. Returns `false` if it was unable to delete the file or directory.

```
public boolean createNewFile() throws IOException
```

Creates a new empty file named by the abstract path name, provided that a file of that name does not already exist. Returns `true` if successful and returns `false` otherwise. Throws `IOException` if, for example, the parent directory of the file does not exist.

```
public String getName()
```

Returns the last name in the abstract path name (which is typically a simple file name). Returns the empty string if the abstract path name is the empty string.

```
public String getPath()
```

Returns the abstract name as a `String` value.

```
public boolean renameTo (File newName)
```

Renames the file represented by the abstract path name to *newName*. Returns `true` if successful; otherwise returns `false`. *newName* can be a relative or absolute path name. This may require moving the file to a different device. Whether or not the file can be moved is system dependent.

```
public boolean isFile()
```

Returns `true` if a file exists that is named by the abstract path name, and if the file is a normal file; otherwise returns `false`. The meaning of "normal" is system dependent, but typically excludes directories. Files created by the methods discussed in this lecture -- except `mkdir` and `mkdirs` below -- are always normal files.

```
public boolean isDirectory()
```

Returns `true` if a directory (folder) exists that is named by the abstract path name; otherwise returns `false`.

```
public boolean mkdir()
```

Makes a directory named by the abstract path name. Will not create parent directories. See `mkdirs`. Returns `true` if successful; otherwise returns `false`.

```
public boolean mkdirs()
```

Makes a directory named by the abstract path name. Will create any necessary but non-existent parent directories. Returns `true` if successful; otherwise returns `false`. Note that if it fails, then some of the parent directories may have been created.

```
public long length()
```

Returns the length in bytes of the file named by the abstract path name. If the file does not exist or the abstract path name names a directory, then the value is not specified and may be anything.

Exercise: Modify the example above to ask for a new file name, and then rename it just before exiting.

Exercise: Modify the example above to check if the file name entered contains "/" or "\". If so, check if the directory exists and create it if necessary before creating the file.