Input and Output

The System Class

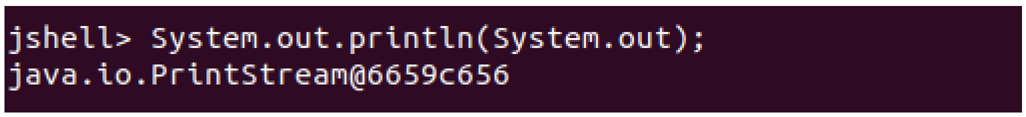
So far you have seen that messages and variables can be printed to the terminal using System.out.println.

System.out.println("Hi there");

String name = "Frank N. Stein";
System.out.println(name);

System is a class that provides methods related to the "system" or environment where our Java programs run. It also provides **System.out,** which is a special ***object*** that provides methods for displaying output, including println.

In fact, we can use System.out.println to display the value of System.out:

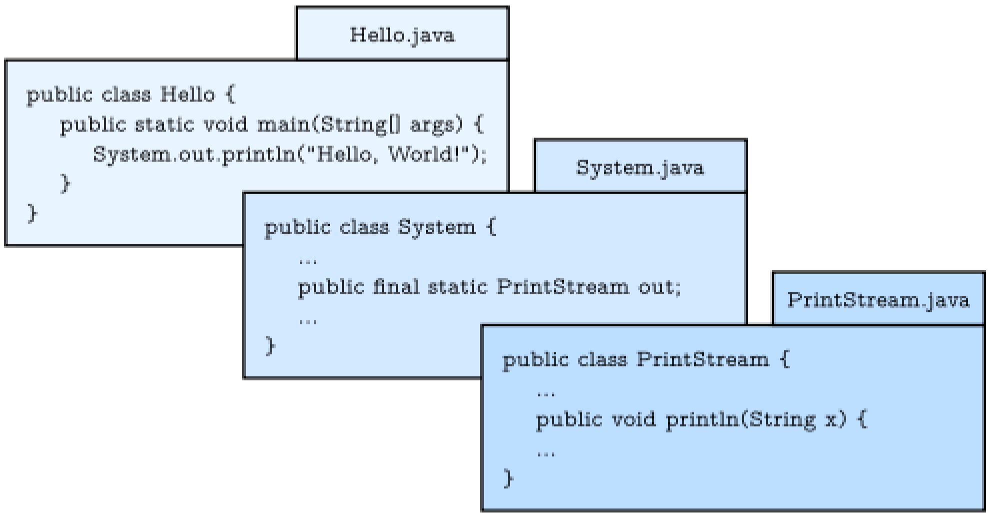


```
jshell> System.out.println(System.out);
java.io.PrintStream@6659c656
```

This output indicates that System.out is a **PrintStream object**, which is defined in a package called **java.io**. A package is a collection of related classes; java.io contains classes for "I/O" which stands for input and output. This output is a message that states

The object System.out is an instance of the class java.io.PrintStream and is located at internal JVM address 6659c656

System is defined in a file called **System.java**, and PrintStream is defined in **PrintStream.java.** These files are part of the Java library, which is an extensive collection of classes you can use in your programs. The System class does not need to be imported because it is so commonly used.



```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

```
public class System {
    ...
    public final static PrintStream out;
    ...
}
```

```
public class PrintStream {
    ...
    public void println(String x) {
    ...
}
```

The Scanner Class

The System class also provides the special value **System.in**, which is an **InputStream** object that provides methods for ***reading input from the keyboard*** or any external data source that we can connect. These methods are not easy to use; fortunately, Java provides other classes that make it easier to handle common input tasks. For example, Scanner is a class that provides methods for inputting words, numbers, and other data. Scanner is provided in the **java.util** package, which is a package that contains classes so useful they are called "utility classes". Before you can use Scanner, you have to import it like this:

import java.util.Scanner;

This import statement tells the compiler that when you say Scanner, you mean the one defined in **java.util.** This is necessary because there might be another class named Scanner in another package. Using an import statement makes your code unambiguous. ***Import statements can't be inside a class definition. By convention, they are usually at the beginning of the file.***

Creating an instance of the Scanner class

Before you can use an object from the Java API you must ***create an instance*** of the class. This is accomplished with the keyword **new** and a call to the class's constructor method. We will spend time on constructors in our discussion of classes and objects. For now, you just need to know that each class has a special method called a constructor, and it is used to construct objects. The **new** keyword actually allocates the memory from a special area in the JVM called **the heap.** We will cover this in more detail (Don't confuse this with the heap data structure)

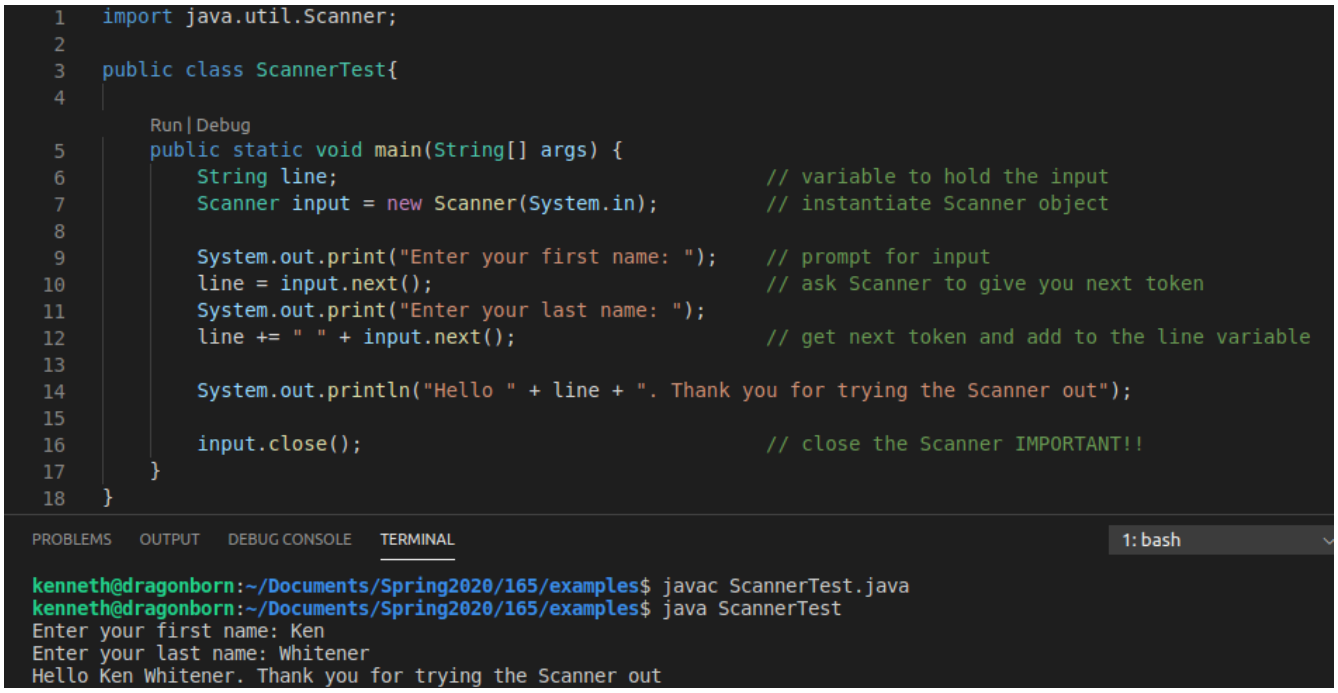
Scanner input = new Scanner(System.in);

This line declares a Scanner variable named **input** and creates a **new Scanner instance** that takes input from **System.in**. The **in** object is an instance of the class **InputStream** similar to how the **out** object is an instance of the class **OutputStream.** The Scanner object will scan this stream for input data, and we can send messages to this object to **get** data from the stream.

Scanner provides a method called **next()** that reads a token from the input stream (entered from the keyboard) and returns a String.

https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/Scanner.html

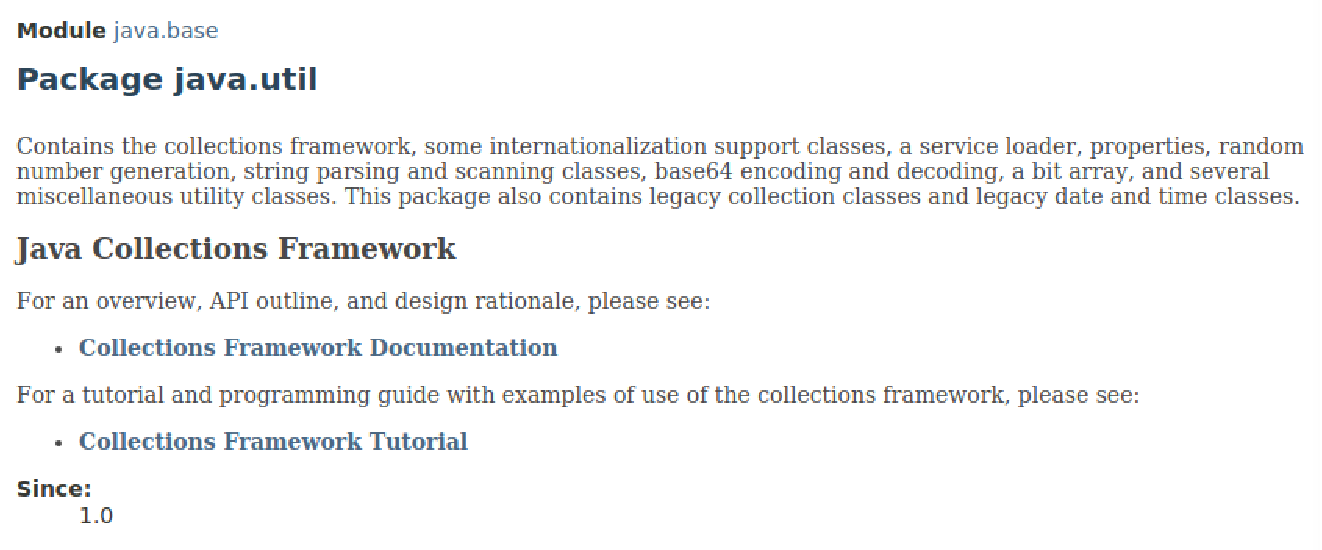
The following program reads two String tokens from the stream connected to your keyboard and then echoes the input along with a brief message.


```java
import java.util.Scanner;

public class ScannerTest{

    Run | Debug
    public static void main(String[] args) {
        String line;                                    // variable to hold the input
        Scanner input = new Scanner(System.in);         // instantiate Scanner object

        System.out.print("Enter your first name: ");    // prompt for input
        line = input.next();                            // ask Scanner to give you next token
        System.out.print("Enter your last name: ");
        line += " " + input.next();                     // get next token and add to the line variable

        System.out.println("Hello " + line + ". Thank you for trying the Scanner out");

        input.close();                                  // close the Scanner IMPORTANT!!
    }
}
```

```
kenneth@dragonborn:~/Documents/Spring2020/165/examples$ javac ScannerTest.java
kenneth@dragonborn:~/Documents/Spring2020/165/examples$ java ScannerTest
Enter your first name: Ken
Enter your last name: Whitener
Hello Ken Whitener. Thank you for trying the Scanner out
```

The Scanner API

Let’s take an opportunity to examine the Scanner API to see what it has to offer. I begin this process by finding the link for the **java.util** package. This can be found in the **java.base module**.



Module java.base

Package java.util

Contains the collections framework, some internationalization support classes, a service loader, properties, random number generation, string parsing and scanning classes, base64 encoding and decoding, a bit array, and several miscellaneous utility classes. This package also contains legacy collection classes and legacy date and time classes.

Java Collections Framework

For an overview, API outline, and design rationale, please see:

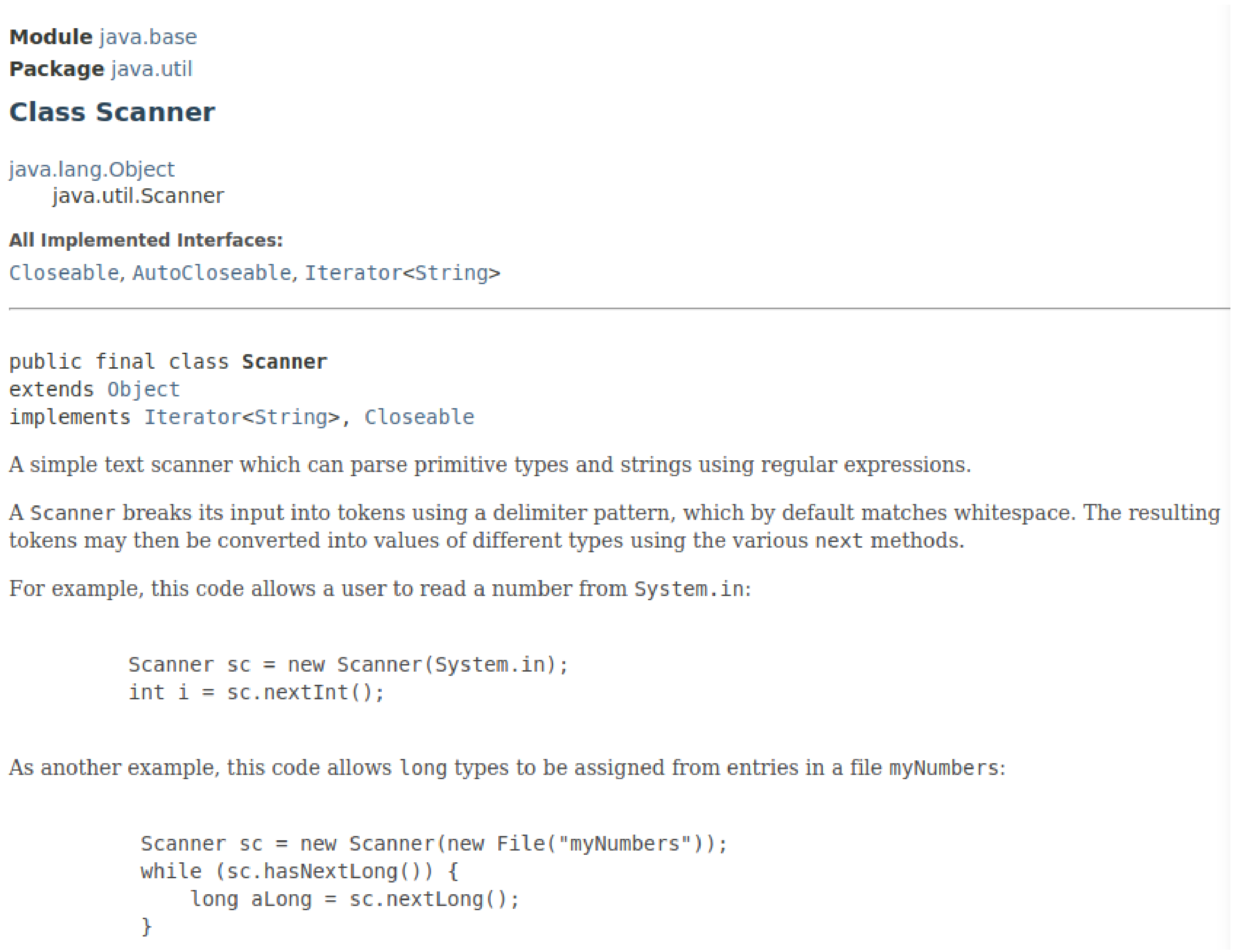
- Collections Framework Documentation

For a tutorial and programming guide with examples of use of the collections framework, please see:

- Collections Framework Tutorial

Since:
1.0

Scroll down until you see the entry for **Scanner**. Click on it to begin your exploration. Here’s a preview


Module java.base
Package java.util

Class Scanner

java.lang.Object
    java.util.Scanner

All Implemented Interfaces:
Closeable, AutoCloseable, Iterator<String>

```
public final class Scanner
extends Object
implements Iterator<String>, Closeable
```

A simple text scanner which can parse primitive types and strings using regular expressions.

A `Scanner` breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various `next` methods.

For example, this code allows a user to read a number from `System.in`:

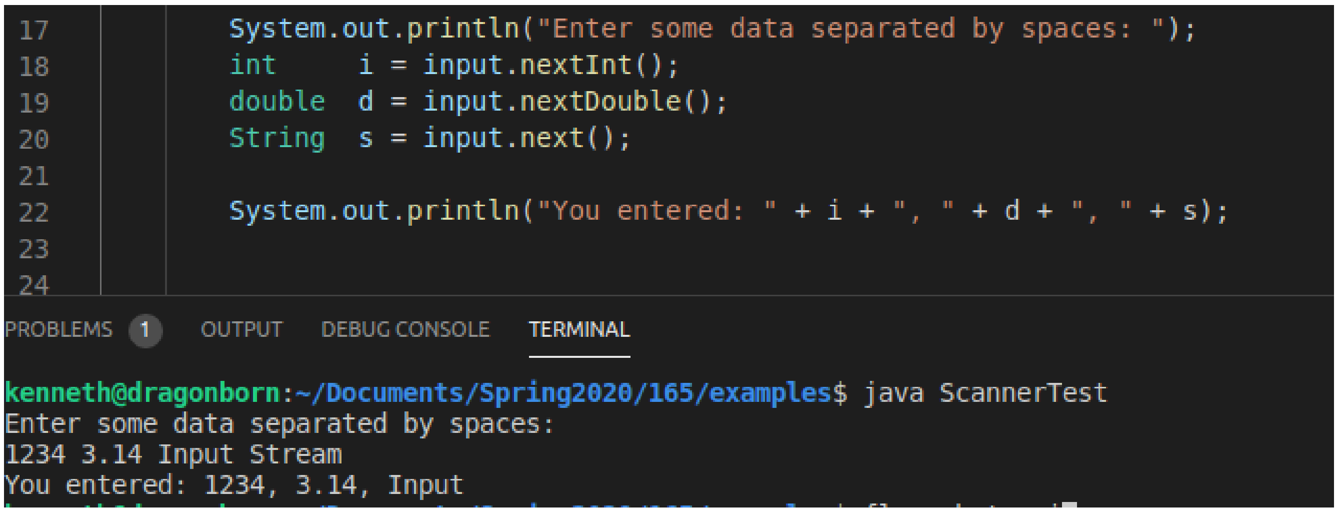
```
     Scanner sc = new Scanner(System.in);
     int i = sc.nextInt();
```

As another example, this code allows `long` types to be assigned from entries in a file `myNumbers`:

```
      Scanner sc = new Scanner(new File("myNumbers"));
      while (sc.hasNextLong()) {
          long aLong = sc.nextLong();
      }
```

To continue on with our discussion of Scanner input operations, scroll down the method summary until you see the beginning of a large block of methods that begin with **next.** These **next** methods are the way we ask the Scanner to extract data from the input stream provided by **System.in**.

Notice that there is a **next<Type>** method for each primitive data type supported by Java, and a couple of object types. You use these these **next<Type>** methods to extract data from the input stream. The raw data in an input stream is treating as a String and can be returned as a String by using plain ole **next** or **nextLine.** Otherwise, the **next<Type>** methods will perform a conversion and return the data as the converted type. Obviously, there are some issues to keep in mind. If a token read from the input stream cannot be converted to the requested type an exception will occur. Examine the following example. Remember, the variable **input** is a reference to our Scanner object.



```
        System.out.println("Enter some data separated by spaces: ");
        int     i = input.nextInt();
        double  d = input.nextDouble();
        String  s = input.next();

        System.out.println("You entered: " + i + ", " + d + ", " + s);

```

```
kenneth@dragonborn:~/Documents/Spring2020/165/examples$ java ScannerTest
Enter some data separated by spaces:
1234 3.14 Input Stream
You entered: 1234, 3.14, Input
```

You can enter as much data as you like and terminate this operation by pressing Enter. When Enter is pressed, the data is added to the underlying input stream and resembles the following collection of tokens.

Input Stream

"1234"	White space	"3.14"	White space	"Input"	White space	"Stream"	"\n"

At this point there is an implicit **cursor** that points to the beginning of the stream. The call to **input.nextInt()** will begin reading at the cursor and will continue reading until it encounters a white space character or a new line character, at which point it will halt and return the data, converted to the type expressed by the method name. If the underlying data cannot be converted successfully to the desired type you will receive an **InputMismatchException.**

Cursor

"1234"	White space	"3.14"	White space	"Input"	White space	"Stream"	"\n"

After the statement

int i = input.nextInt();

The variable **i** will contain the value **1234** and the cursor will advance. The scanning of the input stops at the white space character.

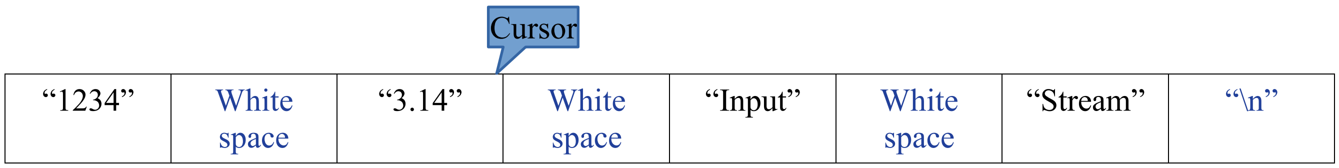
Cursor

"1234"	White space	"3.14"	White space	"Input"	White space	"Stream"	"\n"

After the statement

double d = input.nextDouble();

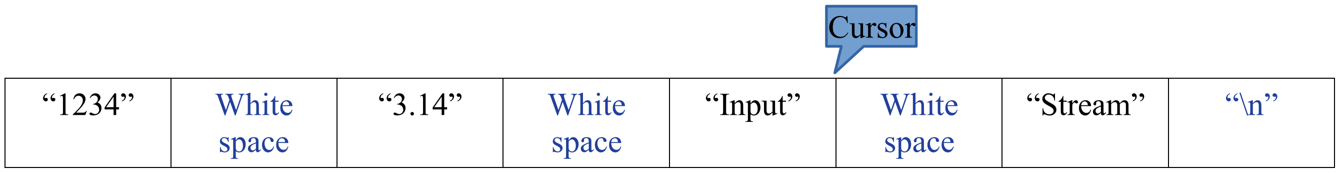
The variable **d** will contain the value **3.14** and the cursor will advance. The scanning of the input stops at the white space character.



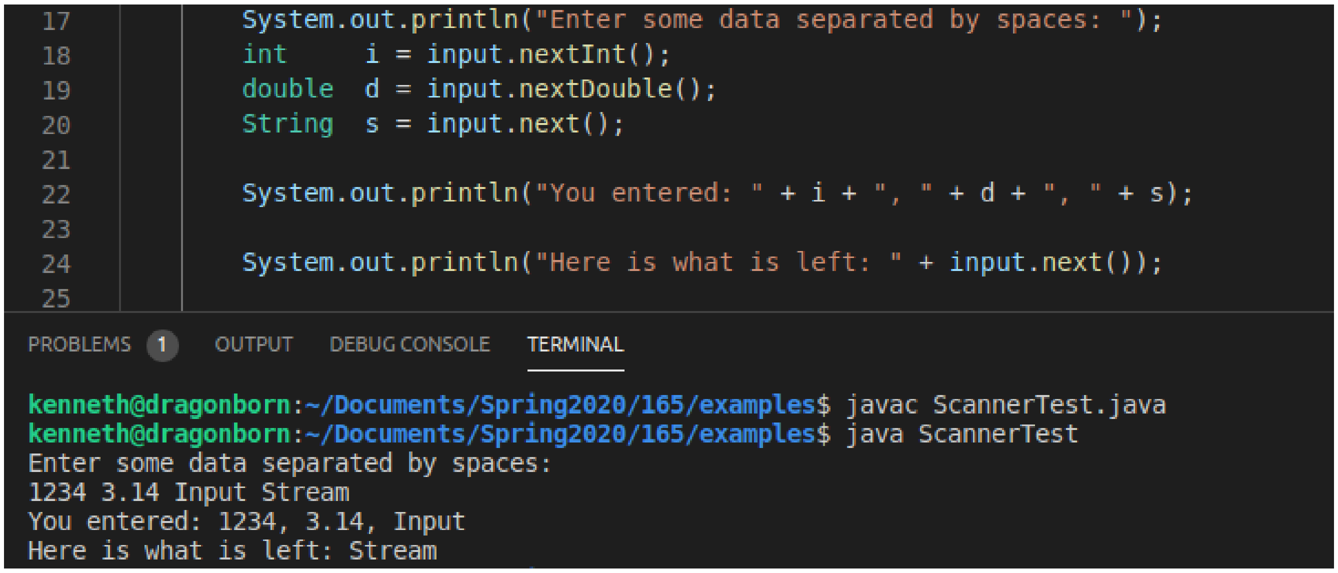
After the statement

String s = input.next();

The variable **s** will contain the value **“Input”** and the cursor will advance. The scanning of the input stops at the white space character.



Because we only wrote code to extract three values from the stream, the cursor stays where it is, and the residual data is ***left in the stream***. You can see this by executing an additional **next** call and printing the results. The line break character will be left in the stream. This presents challenges for complex input operations.

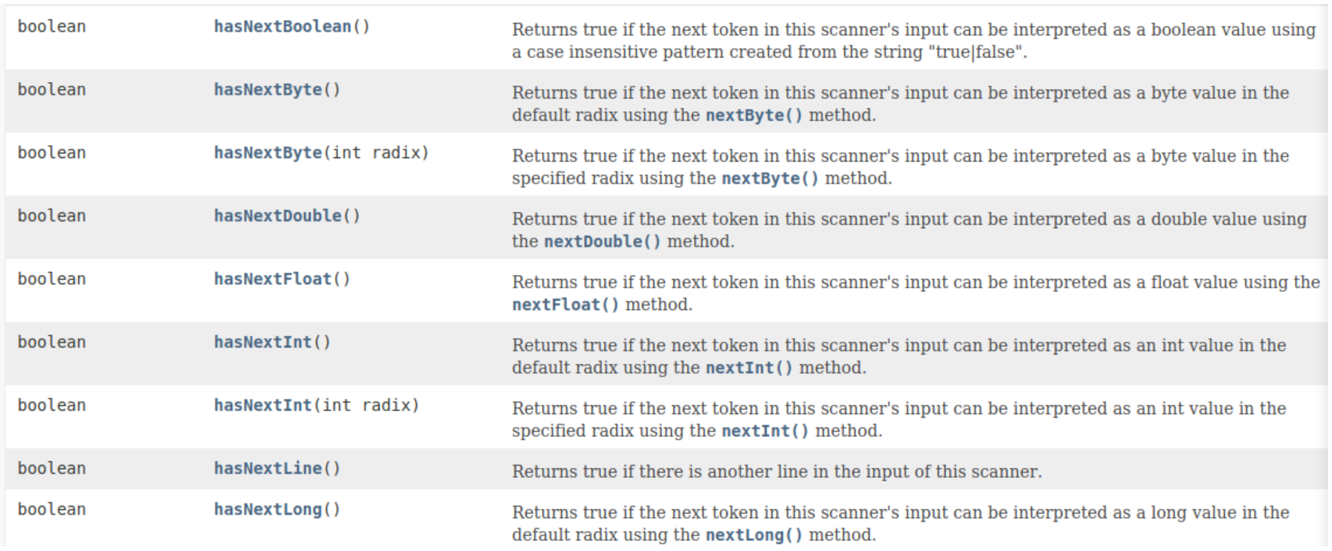


```java
        System.out.println("Enter some data separated by spaces: ");
        int     i = input.nextInt();
        double  d = input.nextDouble();
        String  s = input.next();

        System.out.println("You entered: " + i + ", " + d + ", " + s);

        System.out.println("Here is what is left: " + input.next());

```

```
kenneth@dragonborn:~/Documents/Spring2020/165/examples$ javac ScannerTest.java
kenneth@dragonborn:~/Documents/Spring2020/165/examples$ java ScannerTest
Enter some data separated by spaces:
1234 3.14 Input Stream
You entered: 1234, 3.14, Input
Here is what is left: Stream
```

This situation can lead to some tricky coding. Luckily the Java API provides a collection of Boolean **hasNext<Type>** methods that we can use to test the input stream. I have included a brief snippet of these methods.

Consult the API for examples of usage.



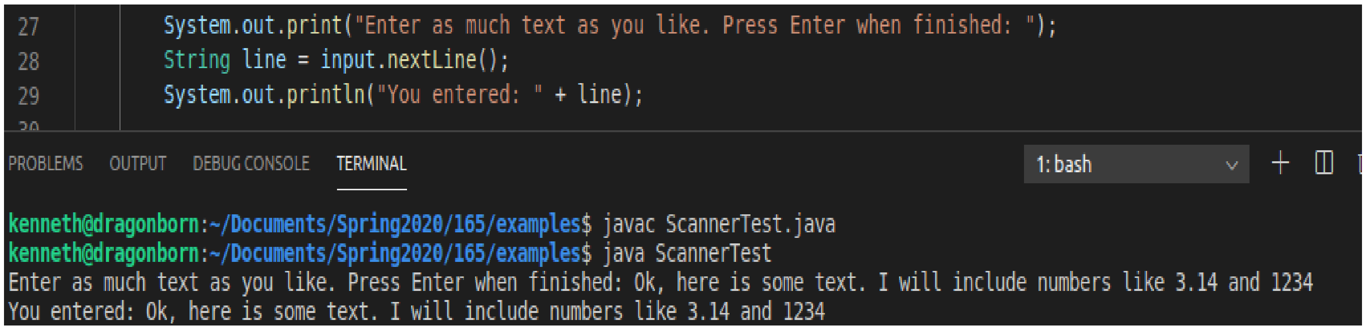
`boolean`	`hasNextBoolean()`	Returns true if the next token in this scanner's input can be interpreted as a boolean value using a case insensitive pattern created from the string "true|false".
`boolean`	`hasNextByte()`	Returns true if the next token in this scanner's input can be interpreted as a byte value in the default radix using the `nextByte()` method.
`boolean`	`hasNextByte(int radix)`	Returns true if the next token in this scanner's input can be interpreted as a byte value in the specified radix using the `nextByte()` method.
`boolean`	`hasNextDouble()`	Returns true if the next token in this scanner's input can be interpreted as a double value using the `nextDouble()` method.
`boolean`	`hasNextFloat()`	Returns true if the next token in this scanner's input can be interpreted as a float value using the `nextFloat()` method.
`boolean`	`hasNextInt()`	Returns true if the next token in this scanner's input can be interpreted as an int value in the default radix using the `nextInt()` method.
`boolean`	`hasNextInt(int radix)`	Returns true if the next token in this scanner's input can be interpreted as an int value in the specified radix using the `nextInt()` method.
`boolean`	`hasNextLine()`	Returns true if there is another line in the input of this scanner.
`boolean`	`hasNextLong()`	Returns true if the next token in this scanner's input can be interpreted as a long value in the default radix using the `nextLong()` method.

The nextLine method

The methods we explored above all terminate the scanning operation at either a white space or a new line character. These termination tokens are called **delimiters.** The Scanner supports using custom delimiters which you can set by calling the **useDelimiter** method (check the API spec for this).

The **nextLine** method needs a little bit of exposition as it can lead to some problems if not understood well. Whereas the **next<Type>** methods will terminate and return upon encountering either a white space or new line character, the **nextLine** method will skip all white space and scan until a new line character is encountered. This next line character is the delimiter when scanning with a **nextLine** call.

The issue lies in the fact that the **next<Type>** methods **leave the new line character in the stream.** This can become a problem when mixing **next<Type>** calls with **nextLine** calls. Let's start by examining the functionality of **nextLine**

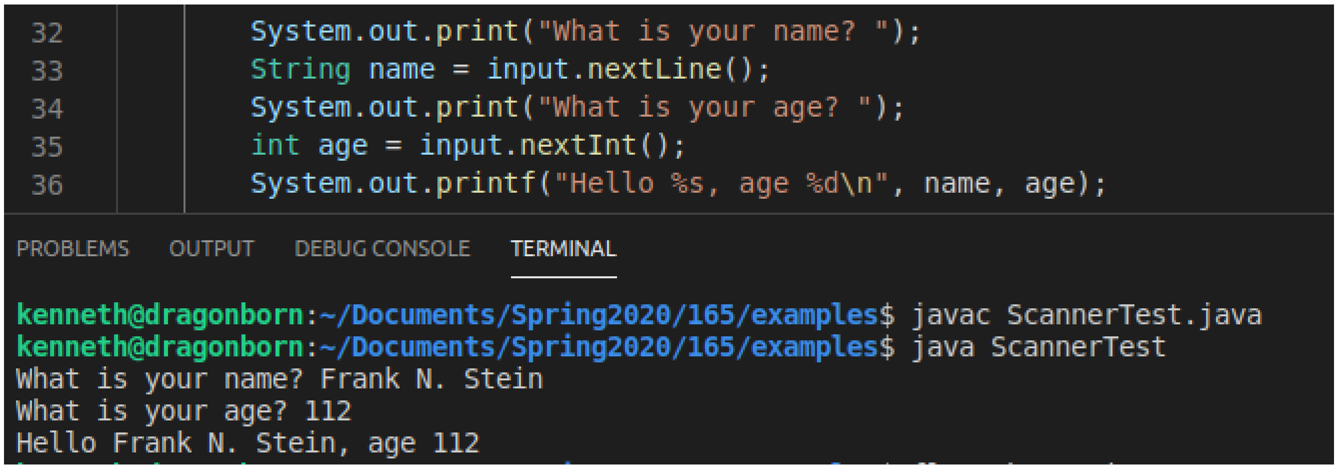


```java
        System.out.print("Enter as much text as you like. Press Enter when finished: ");
        String line = input.nextLine();
        System.out.println("You entered: " + line);
```

```
kenneth@dragonborn:~/Documents/Spring2020/165/examples$ javac ScannerTest.java
kenneth@dragonborn:~/Documents/Spring2020/165/examples$ java ScannerTest
Enter as much text as you like. Press Enter when finished: Ok, here is some text. I will include numbers like 3.14 and 1234
You entered: Ok, here is some text. I will include numbers like 3.14 and 1234
```

Notice how **nextLine** ignores the white space delimiter and reads up until the new line character. This is perfectly fine behavior and can be combined with other **next<Type>** calls to scan various types of input.

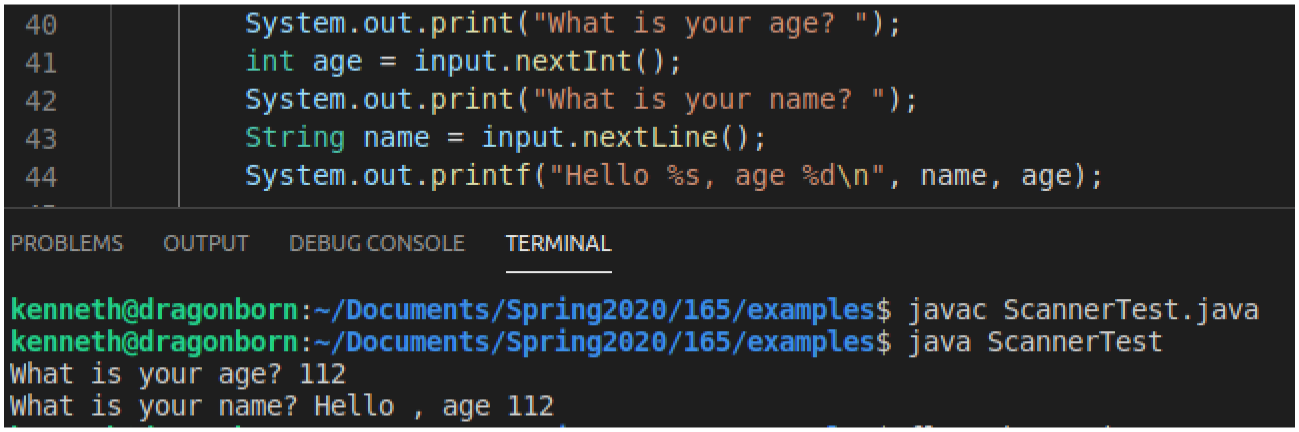
Analyze the following example, I have included a instance of the ***printf*** method. "Print Formatted" . . . stay tuned.



```java
        System.out.print("What is your name? ");
        String name = input.nextLine();
        System.out.print("What is your age? ");
        int age = input.nextInt();
        System.out.printf("Hello %s, age %d\n", name, age);
```

```
kenneth@dragonborn:~/Documents/Spring2020/165/examples$ javac ScannerTest.java
kenneth@dragonborn:~/Documents/Spring2020/165/examples$ java ScannerTest
What is your name? Frank N. Stein
What is your age? 112
Hello Frank N. Stein, age 112
```

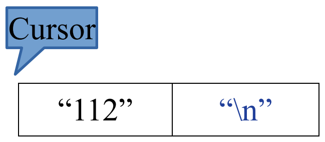
This behaves as expected. But if we reorder those lines to ask for the age first something unexpected occurs.



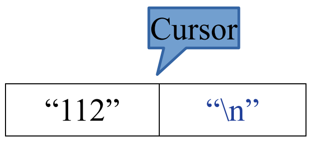
```java
        System.out.print("What is your age? ");
        int age = input.nextInt();
        System.out.print("What is your name? ");
        String name = input.nextLine();
        System.out.printf("Hello %s, age %d\n", name, age);
```

```
kenneth@dragonborn:~/Documents/Spring2020/165/examples$ javac ScannerTest.java
kenneth@dragonborn:~/Documents/Spring2020/165/examples$ java ScannerTest
What is your age? 112
What is your name? Hello , age 112
```

Notice that the Scanner reads the int as expected but blows right past the line 43 and does not pause to allow us to enter the name. What is going on here? Here is the composition of the input stream after entering the age.



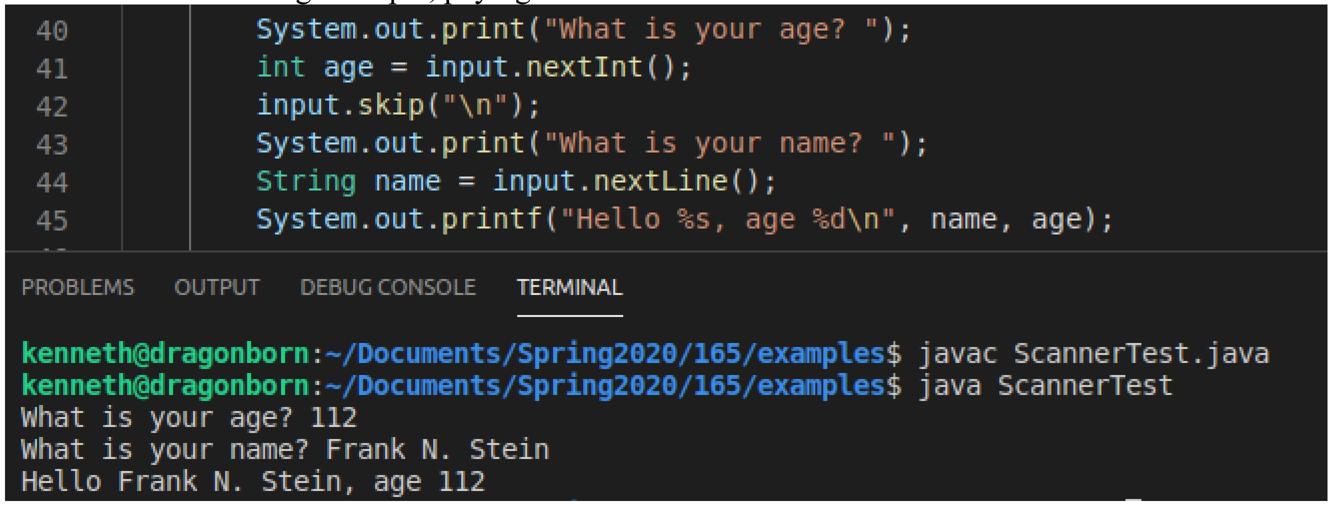
When line 41 executes, the cursor scans until it encounters a delimiter, which happens to be **“\n”** advances to the next token in the stream. **NextInt** returns the converted **112** and stops at the new line character.



The new line character remains in the stream. When the **nextLine** method is executed on line 43 it immediately encounters the **“\n”** (which is its delimiter) and returns. It doesn’t pause to allow us to enter anything because it immediately “sees” the token that tells it to stop. This behavior will happen anytime you use a **next<Type>** method before a **nextLine**. This is certainly an annoying bug, but it can be worked around. If you know you want to sequence next calls like this you can tell the Scanner to **skip** the **“\n”** character by invoking the **skip** method, along with the String pattern to skip. This is from the API

`skip(String pattern)`	Skips input that matches a pattern constructed from the specified string.

Check out the following example, paying close attention to line 42.



```
40        System.out.print("What is your age? ");
41        int age = input.nextInt();
42        input.skip("\n");
43        System.out.print("What is your name? ");
44        String name = input.nextLine();
45        System.out.printf("Hello %s, age %d\n", name, age);
```

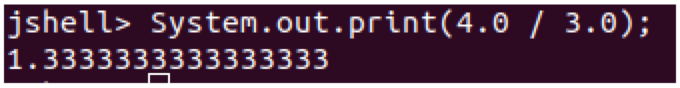
```
kenneth@dragonborn:~/Documents/Spring2020/165/examples$ javac ScannerTest.java
kenneth@dragonborn:~/Documents/Spring2020/165/examples$ java ScannerTest
What is your age? 112
What is your name? Frank N. Stein
Hello Frank N. Stein, age 112
```

Skipping the “\n” character allows the program to successfully process the desired input.

Formatting output with printf

Java comes with a print method that applies formatting in a similar fashion to the C function **printf** and the Python function **String.format.**

When you output a double using print or println, it displays up to 16 decimal places:



```
jshell> System.out.print(4.0 / 3.0);
1.3333333333333333
```

That might be more than you want. System.out provides another method, called printf, that gives you more control of the format. The “f” in printf stands for “formatted”. Here’s an example:

```
System.out.printf(”Four thirds = %.3f”,4.0/3.0);
```

The first value in the parentheses is a format string that specifies how the output should be displayed. This format string contains ordinary text followed by a format specifier, which is a special sequence that starts with a percent sign. The format specifier **%.3f** indicates that the following value should be displayed as **floating-point, rounded to three decimal places**.

The result is:

Four thirds = 1.333

The format string can contain any number of format specifiers; here’s an example with two:

```
int inch = 100;
double cm = inch * CM_PER_INCH;
System.out.printf(”%d in = %f cm n”,inch,cm);
```

The result is:

100 in = 254.000000 cm

Like print, printf does not append a newline. So, formatted strings often end with a newline character. The format specifier **%d** displays integer values (“d” stands for “decimal”). The values are matched up with the format specifiers in order, so inch is displayed using **%d,** and cm is displayed using **%f**.

Learning about format strings is like learning a sub-language within Java. There are many options, and the details can be overwhelming. The following table lists a few common uses, to give you an idea of how things work.

%d	decimal integer	12345
%08d	padded with zeros, at least 8 digits wide	00012345
%f	floating-point	6.789000
%.2f	rounded to 2 decimal places	6.79

For more details, refer to the documentation of **java.util.Formatter**.

https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/Formatter.html

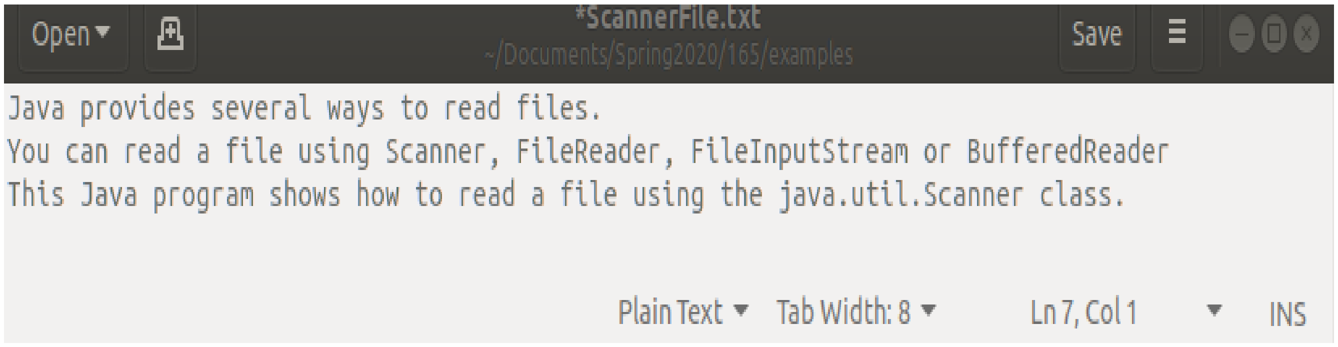
Reading a Text File with the Scanner Object

We can use the same **next** and **hasNext** methods to scan input from a file. An input stream can be created, connecting our Java program with a data source like a file saved on the hard drive. There are a couple of additional concerns we'll need to handle when dealing with files, and these will be explained and demonstrated.

Here are some details:

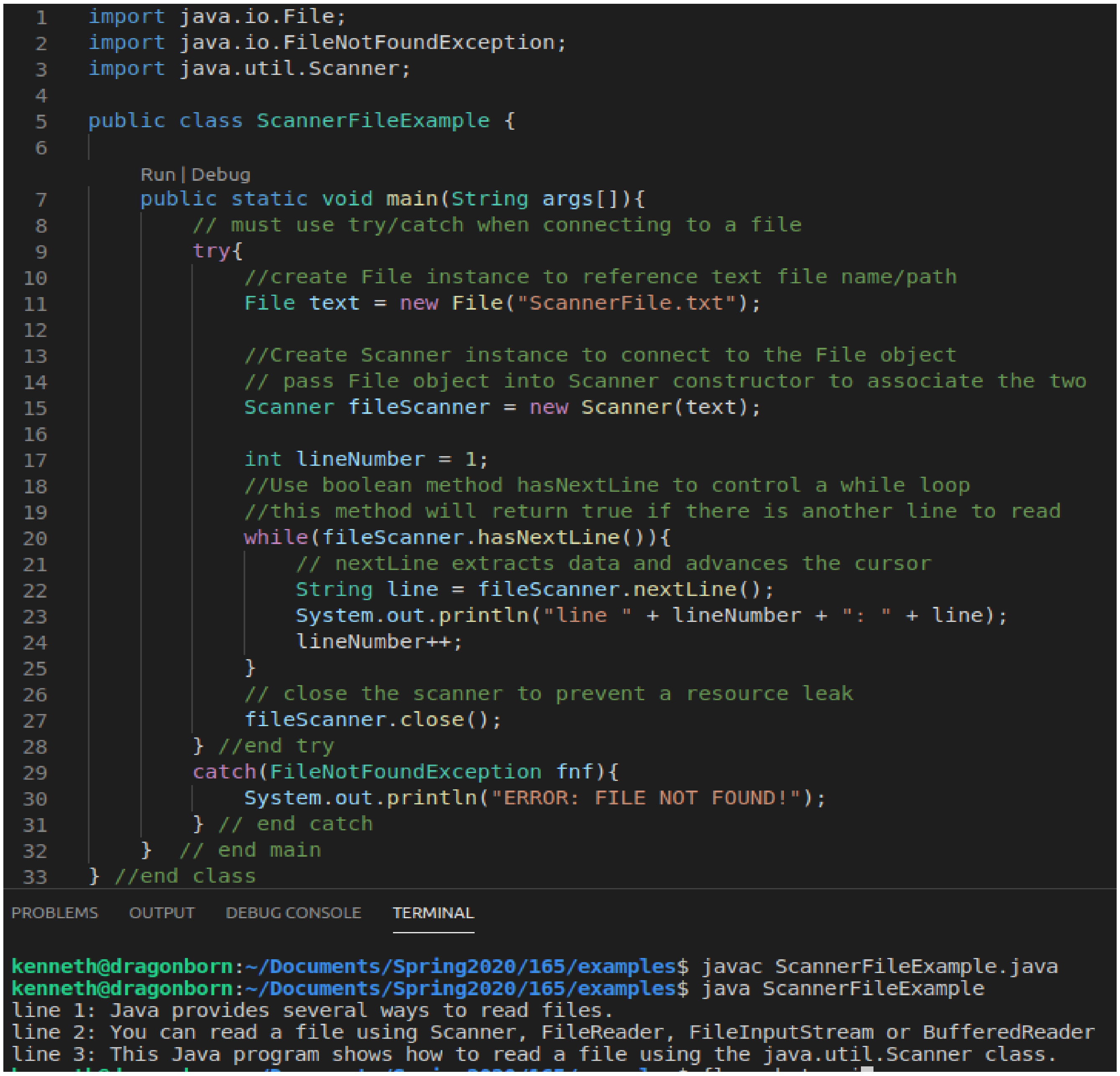
1. I have a text file stored in the same directory as the Java class file. This is convenient as we do not have to supply the full path when connecting our stream to the file
2. Because we are connecting to an external resource, we have to include some error checking. We will cover the **try/catch paradigm** in detail later in the course. For now, just realize that Java ***requires*** this error handling in certain situations. Connecting your program to an external file is one of these situations.

Here is a screenshot of the file named ***ScannerFile.text*** (you can also download the file and associated source code)



```
Java provides several ways to read files.
You can read a file using Scanner, FileReader, FileInputStream or BufferedReader
This Java program shows how to read a file using the java.util.Scanner class.
```

The code below simply reads the entirety of each line. If you need to pick out individual tokens research the API to choose the correct **next<Type>** method. Obviously, you will need to be familiar with the structure of the text file to determine which method to use.



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

public class ScannerFileExample {

    public static void main(String args[]){
        // must use try/catch when connecting to a file
        try{
            //create File instance to reference text file name/path
            File text = new File("ScannerFile.txt");

            //Create Scanner instance to connect to the File object
            // pass File object into Scanner constructor to associate the two
            Scanner fileScanner = new Scanner(text);

            int lineNumber = 1;
            //Use boolean method hasNextLine to control a while loop
            //this method will return true if there is another line to read
            while(fileScanner.hasNextLine()){
                // nextLine extracts data and advances the cursor
                String line = fileScanner.nextLine();
                System.out.println("line " + lineNumber + ": " + line);
                lineNumber++;
            }
            // close the scanner to prevent a resource leak
            fileScanner.close();
        } //end try
        catch(FileNotFoundException fnf){
            System.out.println("ERROR: FILE NOT FOUND!");
        } // end catch
    }  // end main
} //end class
```

```
kenneth@dragonborn:~/Documents/Spring2020/165/examples$ javac ScannerFileExample.java
kenneth@dragonborn:~/Documents/Spring2020/165/examples$ java ScannerFileExample
line 1: Java provides several ways to read files.
line 2: You can read a file using Scanner, FileReader, FileInputStream or BufferedReader
line 3: This Java program shows how to read a file using the java.util.Scanner class.
```

Writing to Files

Java contains a variety of tools that allow programmers to write to files. Writing to external resources can be expensive. Java provides a variety of buffering tools that allow programmers to pool write requests so they can be performed in bulk. This decreases the number of actual writes to the underlying file system.

1. **FileWriter:** Convenience class for writing character files. The constructors of this class assume that the default character encoding and the default byte-buffer size are acceptable. To specify these values yourself, construct an OutputStreamWriter on a FileOutputStream. Whether or not a file is available or may be created depends upon the underlying platform. Some platforms, in particular, allow a file to be opened for writing by only one FileWriter (or other file-writing object) at a time. In such situations the constructors in this class would fail if the file involved is already open. FileWriter is the simplest way to write a file in Java. It provides an overloaded write method to write int, byte array, and Strings to a File. You can also write a portion of the String or byte array. FileWriter is meant for writing streams of characters. For writing streams of raw bytes, consider using a FileOutputStream.

   https://docs.oracle.com/javase/7/docs/api/java/io/FileWriter.html

2. **BufferedWriter:** Writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings. The buffer size may be specified, or the default size may be accepted. The default is large enough for most purposes. A newLine() method is provided, which uses the platform's own notion of line separator as defined by the system property **line.separator**. Not all platforms use the newline character ('\n') to terminate lines. Calling this method to terminate each output line is therefore preferred to writing a newline character directly. In general, a Writer sends its output immediately to the underlying character or byte stream. Unless prompt output is required, it is advisable to wrap a BufferedWriter around any Writer whose write() operations may be costly, such as FileWriters and OutputStreamWriters. The buffering pools the write requests so that the actual writes are fewer in number.

   https://docs.oracle.com/javase/7/docs/api/java/io/BufferedWriter.html

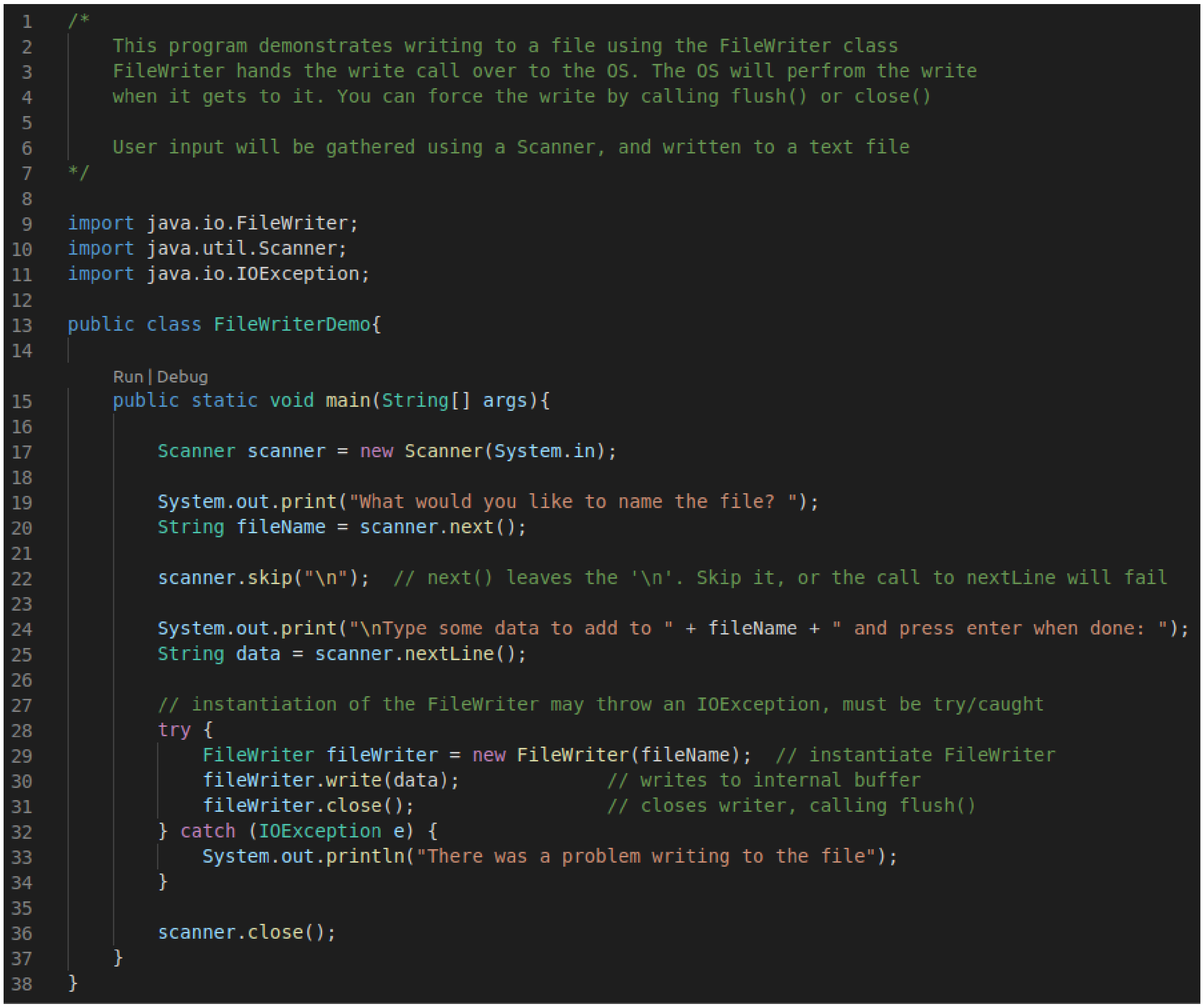
3. **FileOutputStream:** A file output stream is an output stream for writing data to a File or to a FileDescriptor. FileOutputStream is meant for writing streams of raw bytes such as image data. For writing streams of characters, consider using FileWriter.

   https://docs.oracle.com/javase/7/docs/api/java/io/FileOutputStream.html

4. **Files:** Java 7 introduced the Files utility class, and you can write a file using its write function. Internally it's using OutputStream to write byte array into files and serves as a convenience class. This class consists exclusively of static methods that operate on files, directories, or other types of files. In most cases, the methods defined here will delegate to the associated file system provider to perform the file operations. This class simplifies the process of creating new files.

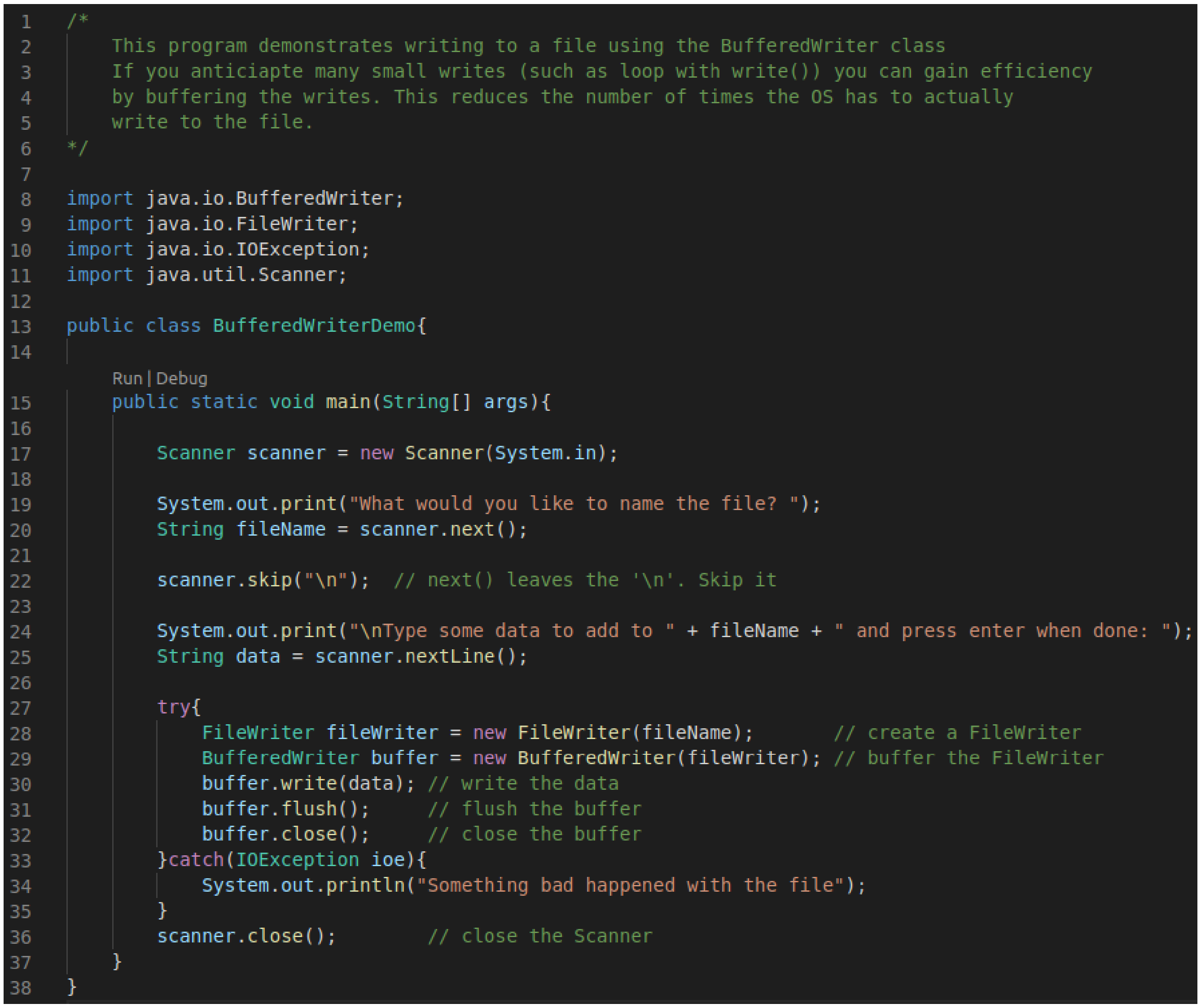
   https://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html

I will include examples of the FileWriter and the BufferedWriter and leave the research of the other classes to you. Here is an example of the unbuffered FileWriter.



```java
/*
    This program demonstrates writing to a file using the FileWriter class
    FileWriter hands the write call over to the OS. The OS will perfrom the write
    when it gets to it. You can force the write by calling flush() or close()

    User input will be gathered using a Scanner, and written to a text file
*/

import java.io.FileWriter;
import java.util.Scanner;
import java.io.IOException;

public class FileWriterDemo{

    public static void main(String[] args){

        Scanner scanner = new Scanner(System.in);

        System.out.print("What would you like to name the file? ");
        String fileName = scanner.next();

        scanner.skip("\n");  // next() leaves the '\n'. Skip it, or the call to nextLine will fail

        System.out.print("\nType some data to add to " + fileName + " and press enter when done: ");
        String data = scanner.nextLine();

        // instantiation of the FileWriter may throw an IOException, must be try/caught
        try {
            FileWriter fileWriter = new FileWriter(fileName);  // instantiate FileWriter
            fileWriter.write(data);             // writes to internal buffer
            fileWriter.close();                 // closes writer, calling flush()
        } catch (IOException e) {
            System.out.println("There was a problem writing to the file");
        }

        scanner.close();
    }
}
```

Here is an example using the BufferedWriter class. The code is almost identical but notice that the FileWriter instance is sent to the BufferedWriter constructor (lines 28, 29) . . . to **buffer the writer.** The difference here is not in the code, but what happens underneath the hood. Understanding these differences is the key to writing efficient code.



```
 1  /*
 2      This program demonstrates writing to a file using the BufferedWriter class
 3      If you anticiapte many small writes (such as loop with write()) you can gain efficiency
 4      by buffering the writes. This reduces the number of times the OS has to actually
 5      write to the file.
 6  */
 7
 8  import java.io.BufferedWriter;
 9  import java.io.FileWriter;
10  import java.io.IOException;
11  import java.util.Scanner;
12
13  public class BufferedWriterDemo{
14
        Run | Debug
15      public static void main(String[] args){
16
17          Scanner scanner = new Scanner(System.in);
18
19          System.out.print("What would you like to name the file? ");
20          String fileName = scanner.next();
21
22          scanner.skip("\n");  // next() leaves the '\n'. Skip it
23
24          System.out.print("\nType some data to add to " + fileName + " and press enter when done: ");
25          String data = scanner.nextLine();
26
27          try{
28              FileWriter fileWriter = new FileWriter(fileName);       // create a FileWriter
29              BufferedWriter buffer = new BufferedWriter(fileWriter); // buffer the FileWriter
30              buffer.write(data); // write the data
31              buffer.flush();     // flush the buffer
32              buffer.close();     // close the buffer
33          }catch(IOException ioe){
34              System.out.println("Something bad happened with the file");
35          }
36          scanner.close();        // close the Scanner
37      }
38  }
```