Chapter 7 – Text Files

Contents

[1 Introduction 2](#_Toc128057950)

[2 Text Files 2](#_Toc128057951)

[3 File Formats 3](#_Toc128057952)

[4 The File Class 4](#_Toc128057953)

[4.1 Introduction 4](#_Toc128057954)

[4.2 Location of Files 4](#_Toc128057955)

[5 Writing Text Files 5](#_Toc128057956)

[6 Reading Text Files 7](#_Toc128057957)

[6.1 Example 1 – Read *Employees* as Primitives 8](#_Toc128057958)

[6.2 Example 2 – Read *Person*🡪\**Dog* as Primitives 9](#_Toc128057959)

[6.3 Exercises 10](#_Toc128057960)

[7 String.split() 11](#_Toc128057961)

[7.1 Example 3 – Read *Person*🡪\**Dog* with *nextLine* and *split* 12](#_Toc128057962)

[7.2 Example 4 – Read *Person*🡪\**Dog* without Number of Dogs Specified 13](#_Toc128057963)

[7.3 General Approach to Reading 14](#_Toc128057964)

[7.4 Exercises 14](#_Toc128057965)

[8 Parsing Numbers 16](#_Toc128057966)

[8.1 Example 5 – Read Select Words 17](#_Toc128057967)

[8.2 Exercises 18](#_Toc128057968)

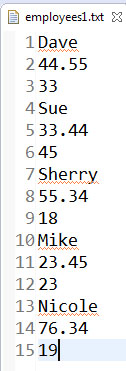
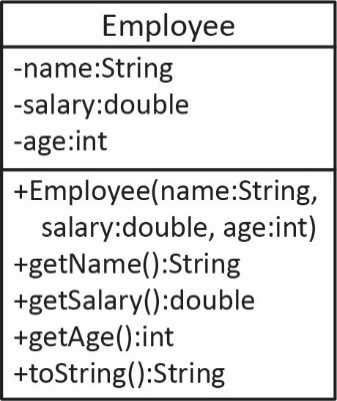
[Appendix 1 The *InputStream* Classes 19](#_Toc128057969)

[Appendix 2 Appending Text Files 20](#_Toc128057970)

[Appendix 3 More on Parsing Numbers 20](#_Toc128057971)

# Introduction

Almost all real systems employ some mechanism for *data persistence,* storing the state of the system on a storage device so that the system can be closed. Then, when the system is started again, the data is read back into data structures in memory. We can use *text files[[1]](#footnote-1)* for this purpose. A *text file* is a file that contains data composed of readable characters (for our purposes) and that exists on a non-volatile storage device.

For example, we may have:

ArrayList<Employee> employees = **new** ArrayList<>();

in memory and before the system closes, it writes the state of each *Employee* objects in memory to a text file as shown on the far right. When the system is started again, the text file is read, *Employee* objects are created, and stored in the list. There are three general approaches to data persistence:

1. When changed – any time the data in memory is changed, the text file is automatically updated. Example-Google Docs
2. Periodic – at some time interval (or some other criteria), the data is automatically saved. Example-MS Word, autosave
3. On demand – the data is saved when the user issues a command for this to happen. MS Word, Ctrl+s.

The approach we will take on a homework assignment is the third approach with this implementation: the data will be read into memory when the system is first loaded. Later, the user can choose to save the data at any time. Thus, we need to learn how to use Java to read and write text files. First, though, we look a bit more closely about what text files are.

# Text Files

A *text file* is actually a file filled with binary digits, 1’s and 0’s (which itself is an abstraction[[2]](#footnote-2)). However, the software (NotePad, Word, Eclipse, *etc.*) that accesses a text file uses *character decoding.* For example, the binary stream:

01001010011000010111011001100001

is decoded in 8-bit blocks each of which is an ASCII character.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 8-bit Blocks | 01001010 | 01100001 | 01110110 | 01100001 |
| Decoded ASCII Text | J | a | v | a |

|  |  |  |  |
| --- | --- | --- | --- |
| **ASCII** | **Dec** | **Hex** | **Binary** |
| 0 | 48 | 30 | 0011 0000 |
| 1 | 49 | 31 | 0011 0001 |
| 2 | 50 | 32 | 0011 0010 |
| … | … | … | … |
| A | 65 | 41 | 0100 0001 |
| B | 66 | 42 | 0100 0010 |
| C | 67 | 43 | 0100 0011 |
| … | … | … | … |
| a | 97 | 61 | 0110 0001 |
| b | 98 | 62 | 0110 0010 |
| c | 99 | 63 | 0110 0011 |
| … | … | … | … |

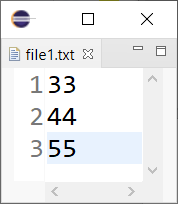
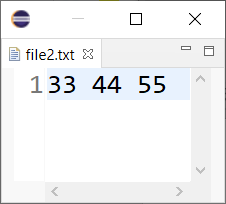
The encoding/decoding scheme shown above is [*ASCII*](https://en.wikipedia.org/wiki/ASCII). For example, the (partial) table on the right shows how text is encoded/decoded. For example, the ASCII letter “a” is represented in binary as: 01100001, which is the decimal number 97, for shorthand.

ASCII was the standard on the internet until 2007. Now, [*UTF-8*](https://en.wikipedia.org/wiki/UTF-8)is the standard. UTF-8 was designed to be backwards compatible with ASCII, so they are the same. By default, Java reads and writes using ASCII, however you can set a parameter to use UTF-8.

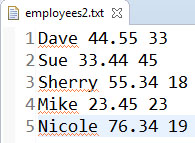
We will use the *Scanner* class to read text files and the *PrinterWriter* class to write them. These classes transparently handle decoding and encoding, respectively, automatically. These are also convenience classes. If you need more control over the reading process, you can use classes from the *InputStream* hierarchy, briefly considered in an [Appendix](#The_InputStream_Classes). For larger problems, you should definitely not use *Scanner* as it is very slow[[3]](#footnote-3).

# File Formats

To read and write text files, it is important to understand the format/layout of the data in the file. For example, if we have a program that writes a list of integers to a text file, do we write one number on each line, or do we write all the numbers on a single line with a space between them, *etc*?

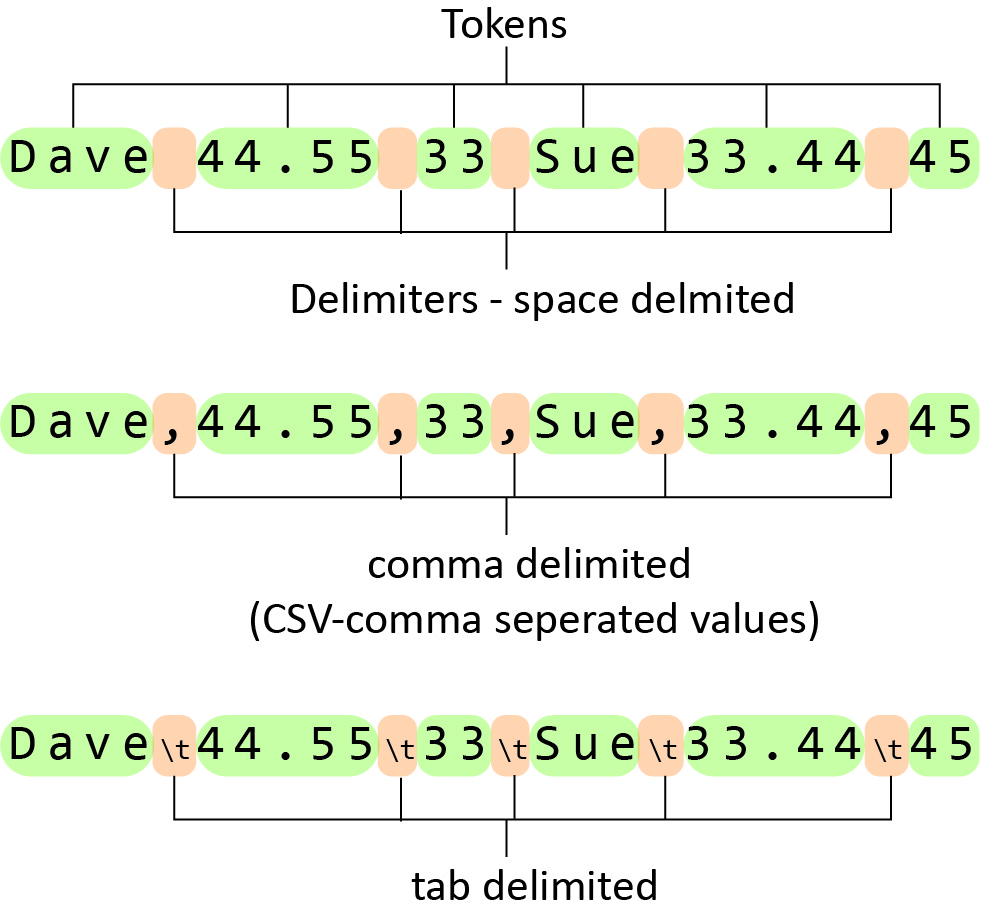
 

Similarly, we need to document exactly what each data value represents. For the example below, the first value is a name, the second is a salary, and the third is the age.

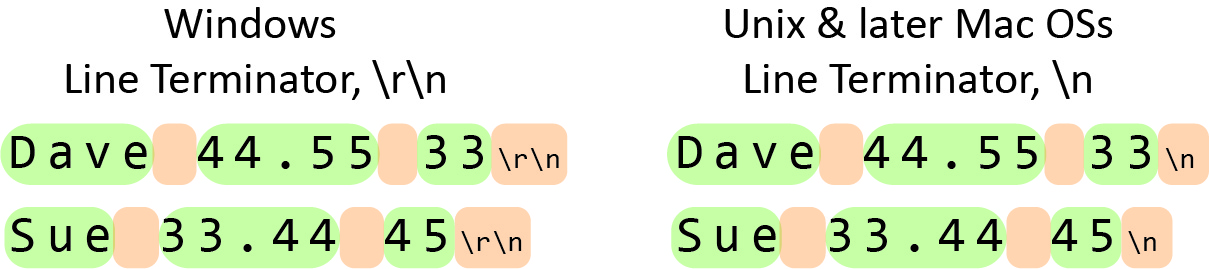
****

When writing a text file, it is important to specify and document the format of the file so that others can understand it. And, in particular, many times another program may need to read a text file into memory. To do that successfully, the programmer must know what the data means and how it is arranged in the file.

The items that we are interested in in a text file are called *tokens* and tokens are separated by a *delimiter.* Three common formats are shown below.



*Space delimited* files have a space (or spaces) in between each token. A limitation is that we cannot have multi-word phrases considered as a single token. For the first example above, “Dave” corresponds to the name token. However, if the name were “Juan Antonio”, then this would be considered two tokens if the data is space delimited. *Comma-delimited* and *tab-delimited* take care of this situation. Comma-delimited is very common and usually referred to as a *CSV file* (comma separated values, [CSV](https://en.wikipedia.org/wiki/Comma-separated_values#See_also)). Most banking and credit card apps allow you to save your transaction history as a CSV file. Then, programs like Google Sheets, or MS Excel will import a CSV file into the expected rows and columns. We will only consider space delimited files, though we show how to use code to specify the delimiter and so handling different delimiters is similar. Another common format is tab-separated values ([TSV](https://en.wikipedia.org/wiki/Tab-separated_values)).

Line termination is handled differently with Windows and Unix and later Mac OSs as shown in the figure on the right. In Windows, lines are terminated with ASCII code 13, “\r”, carriage return followed by ASCII code 10, “\n”, line feed. Unix and later Mac OSs use only the line feed character. If you are using a Mac or Unix, this may come into play when I provide you sample text files to read which are from a Windows machine.

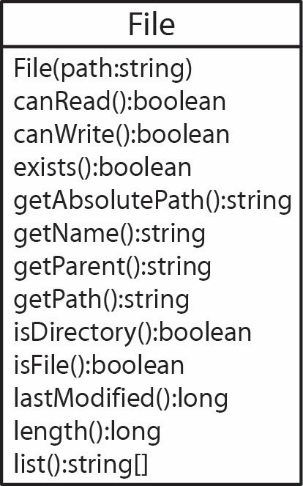
# The File Class

The code for the example in this section is in the *example\_file\_class* package.

## Introduction

To write to a text file, first create an instance of the *File* class, which contains the location of the file and then pass the *File* object to the *PrintWriter,* which has methods to actually write to the file. For example:

File outFile = **new** File("employees1.txt");

PrintWriter writer = **new** PrintWriter(outFile);

The [*File*](https://docs.oracle.com/javase/8/docs/api/java/io/File.html) class is used as an abstraction of the path to a file. Such a path is system dependent and so the *File* class deals with the machine-dependent intricacies of files, folders and paths. Some of the methods in the *File* class are shown on the right. A *File* object is not the actual file; it simply has methods that return *metadata* about the file. For example, with a *File* object, we can ask: “does the file exist?”, “is it a file or a folder?”, “can we write to it?”, *etc.*

The *File* class also contains a number of methods (not shown in the class diagram on the right) that operate on files and folders: *createNewFile, createTempFile, delete, listFiles, mkdir, reanmeTo, setReadable, setWritable*.

For more advanced file and folder manipulation, use the [*Files*](https://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html)class, which has more than 60 static methods. In order to use these methods, you also need the *Path, FileSystem, FileSystems* classes.

## Location of Files

If you are running your program in Eclipse, the default location for accessing text files (or other types) **is the project folder**. Thus, if you have an Eclipse project named, *hw1* and a package named *prob1* where your code is, the when you create a text file to write to:

File outFile = **new** File("employees1.txt");

PrintWriter writer = **new** PrintWriter(outFile);

The file will be created in the *hw1* folder, while your code is in *hw1/src/prob1* folder*.* It can be convenient while developing and testing your code to have the file created in the folder where your code is. In this case, simply specify the path from the default location:

File outFile = **new** File("src/package\_name/employees1.txt");

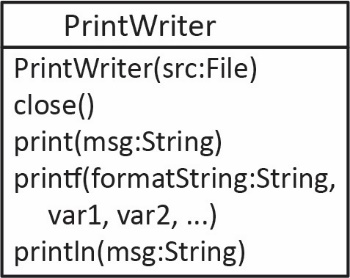
Thus, for the example above:

File outFile = **new** File("hw1/prob1/employees1.txt");

This is the approach we use for homework assignments and the code examples in the text. However, this approach is not *portable*, meaning that it will only run properly in Eclipse (provided the package structure is maintained). For other courses you take, you will need to run code outside of Eclipse, from the command line. I describe how to do this in a separate document where the Chapter 7 download is. The short answers is, don’t define your class in a package – leave the package statement out (technically, the default package). Note, you can change the default location[[4]](#footnote-4) so that you can continue to develop in Eclipse, but the program will run on the command line. There is a separate document on the Text webpage, *Command Line* that goes into more detail and provides examples.

# Writing Text Files

The code for the example in this section is in the *example\_write\_integers* package.

One way to write data to a text file is to use the [*PrintWriter*](https://docs.oracle.com/javase/7/docs/api/java/io/PrintWriter.html)class as shown in the class diagram on the right, which only shows the methods we will use. The constructor accepts a File as an argument:

File outFile = **new** File("src/write\_examples/numbers.txt");

PrintWriter writer = **new** PrintWriter(outFile);

Probably the three most useful (at least for our class) methods of the *PrintWriter* class are *print(…), println(…),* and *printf(…)* which work identically to the *System.out.print(…)* methods. For example, to loop over an array of integers (*nums*) and write them space-delimited to a text file:

**for**(**int** i=0; i<nums.length; i++) {

writer.print(nums[i] + " ");

}

writer.close();

When we are done writing, we must close the writer: writer.close();

As another example, to print the name and balance of each *Account* in a list, we might compose a line as a string, and then print the line with the *println* method.

**for**(Account a : accounts) {

String line = String.*format*("%s %.2f", a.getName(), a.getBalance());

writer.println(line);

}

The *PrintWriter* class can throw a *checked exception* so this means that we must either *try/catch* file operations or declare that it throws an exception. For example, using *try/catch*:

**public** **static** **void** main(String[] args) {

**int**[] nums = {33, 44, 55, 66, 12, 33, 55, 66, 77, 22};

File outFile = **new** File("src/example\_write\_integers/output.txt");

**try** {

PrintWriter writer = **new** PrintWriter( outFile );

**for**( **int** i=0; i<nums.length; i++ ) {

writer.print( nums[i] + " " );

}

writer.close();

}

**catch** (IOException ioe) {

System.***out***.println("Problem creating file or writing");

}

}

Or, using the *throws* approach:

**public** **static** **void** main(String[] args) **throws** FileNotFoundException {

**int**[] nums = {33, 44, 55, 66, 12, 33, 55, 66, 77, 22};

File outFile = **new** File("src/example\_write\_integers/output.txt");

PrintWriter writer = **new** PrintWriter( outFile );

**for**( **int** i=0; i<nums.length; i++ ) {

writer.print( nums[i] + " " );

}

writer.close();

}

Using *throws* is a bit more tedious when the file operations code is inside a method. As shown below, *writeExample2* throws the exception.

**public** **static** **void** writeExample2() **throws** FileNotFoundException {

**...**

File outFile = **new** File("src/example\_write\_integers/output.txt");

PrintWriter writer = **new** PrintWriter( outFile );

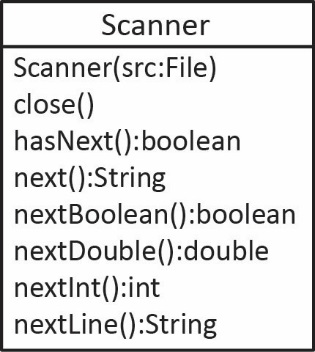
**...**

}

Thus, *main*, which calls it, must either *try/catch* the method call or declare a *throws* clause.

|  |  |
| --- | --- |
| **Option 1** | **Option 2** |
| **public** **static** **void** main(String[] args) {  **try** {  *writeExample2*();  }  **catch** (FileNotFoundException e) {  System.***out***.println(e);  }  } | **public** **static** **void** main(String[] args)  **throws** FileNotFoundException {  *writeExample2*();  } |

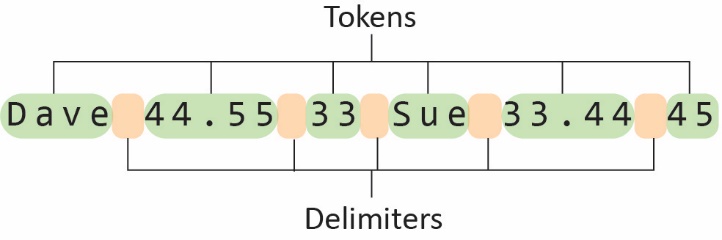
# Reading Text Files

Java provides a number of classes[[5]](#footnote-5) (see [Appendix 2](#The_InputStream_Classes)) for reading text files but we will consider only the [*Scanner*](https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html)class, the same one you used in an earlier course where you read from the keyboard (*System.in*). We will only use the methods shown on the right; however, there are many more that could be useful in practice.

How does a scanner work?

“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.”[[6]](#footnote-6)

For example:



We call such data *space delimited*. Note that any number of spaces is treated as a single delimiter, and, line breaks are also treated as a delimiter. Thus, in some sense, it doesn’t matter how many spaces there are between tokens nor whether the data is all on a single line, or on multiple lines. However, it is much easier to deal with when we know the structure of the data. We will only be using the default delimiter. However, it is easy use other delimiters as the *Scanner* class defines the *useDelimiter(String pattern)* and *useDelimiter(Pattern pattern)* methods. A brief description of the *Scanner* methods is provided in the table below.

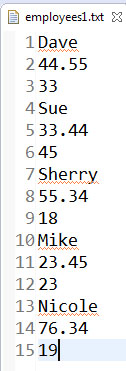
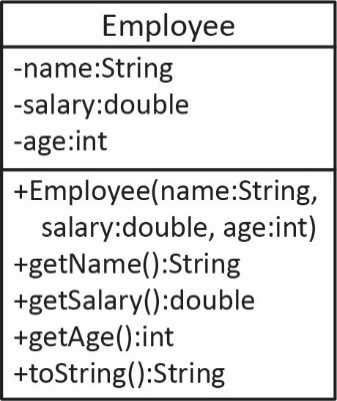
| **Method** | **Description** |
| --- | --- |
| *close()* | Closes the scanner. You should always do this. |
| *hasNext():boolean* | Returns *true* if the scanner has another token. Note: there are a bunch of specialized methods: *hasNextInteger(), hasNextDouble(), etc.* |
| *next():String* | Returns the next token as a string. |
| *nextBoolean():boolean* | Returns the next token as a *boolean*. This method will throw *InputMismatchException* if the token is not a *boolean*. |
| *nextDouble():double* | Returns the next token as a double. This method will throw *InputMismatchException* if the token is not a double. |
| *nextInt():int* | Returns the next token as an integer. This method will throw *InputMismatchException* if the token is not a integer. |
| *nextLine()* | Returns everything from the current position of the cursor to the end of the line and the cursor is moved to the beginning of the next line. |

The key to reading data from a text file is:

* Understanding the structure of the data in the text file. For the data immediately above, the data represents name, salary, age, space delimited, repeat. In this chapter we provide several examples that each illustrate types of structured data.
* What the data means and how it maps to classes in the program.

## Example 1 – Read *Employees* as Primitives

The code for the example in this section is in the *example\_read\_employees* package.

Suppose we have an *Employee* class as shown on the right. There, we can see that an *Employee* has a name, salary, and age. Next, suppose we have a text file as shown on the far right. The line numbers (1-15) are shown on the left and are not a part of the file. Each set of three lines represents an *Employee*: *name*, *salary*, *age*.

Finally, a snippet of code to read these values in and build an *ArrayList<Employee>* is shown below.

ArrayList<Employee> employees = **new** ArrayList<>();

File inFile = **new** File("src/examples1/employees.txt");

**try** {

Scanner input = **new** Scanner( inFile );

**while**( input.hasNext() ) {

String name = input.next();

**double** salary = input.nextDouble();

**int** age = input.nextInt();

employees.add(**new** Employee( name, salary, age ));

}

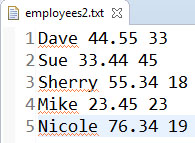
input.close();

}

**catch**( FileNotFoundException e ) {

System.***out***.println(e);

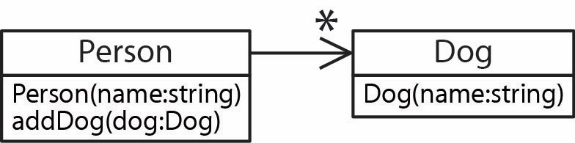
}

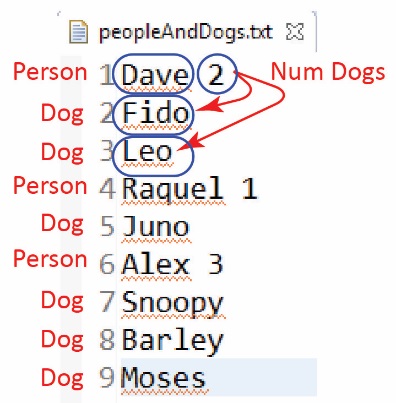
****The text file above has a single data value on each line. However, the exact same code will work if the text file is arranged in either of the two ways shown on the right and below. In other words, a line break (ASCII character code 10) is treated as whitespace, and thus, used as a delimiter.

**Text file of employees where every data value is on a the same line in the format: name, salary, age, name, .... i.e. just one long line of text**

## Example 2 – Read *Person*🡪\**Dog* as Primitives

The code for the example in this section is in the *example\_read\_people\_and\_dogs* package, in *ReadPeopleAndDogs\_Ver1* class.

Suppose we have a *Person* class as shown on the right. There, we can see that a *Person* has a name and any number of *Dog*s, where a *Dog* has a name.

Next, suppose we have a file as shown on the right. A person (Dave) is shown on line 1 and the number of dogs he has (2). The next two lines provide the dogs names.

Thus, to read in this data, where each person can have a different number of dogs, we could read a person’s name

String name = input.next();

and then read their number of dogs

**int** numDogs = input.nextInt();

and then loop over the number of dogs reading each dog’s name:

**for**(**int** i=0; i<numDogs; i++){

String dogName = input.next();

...

}

Finally, a snippet of code to read the file and create an *ArrayList<Person>* is shown below:

ArrayList<Person> people = **new** ArrayList<>();

...

**while**( input.hasNext() ) {

String name = input.next();

Person p = **new** Person(name);

**int** numDogs = input.nextInt();

**for**(**int** i=0; i<numDogs; i++){

String dogName = input.next();

Dog dog = **new** Dog(dogName);

p.addDog(dog);

}

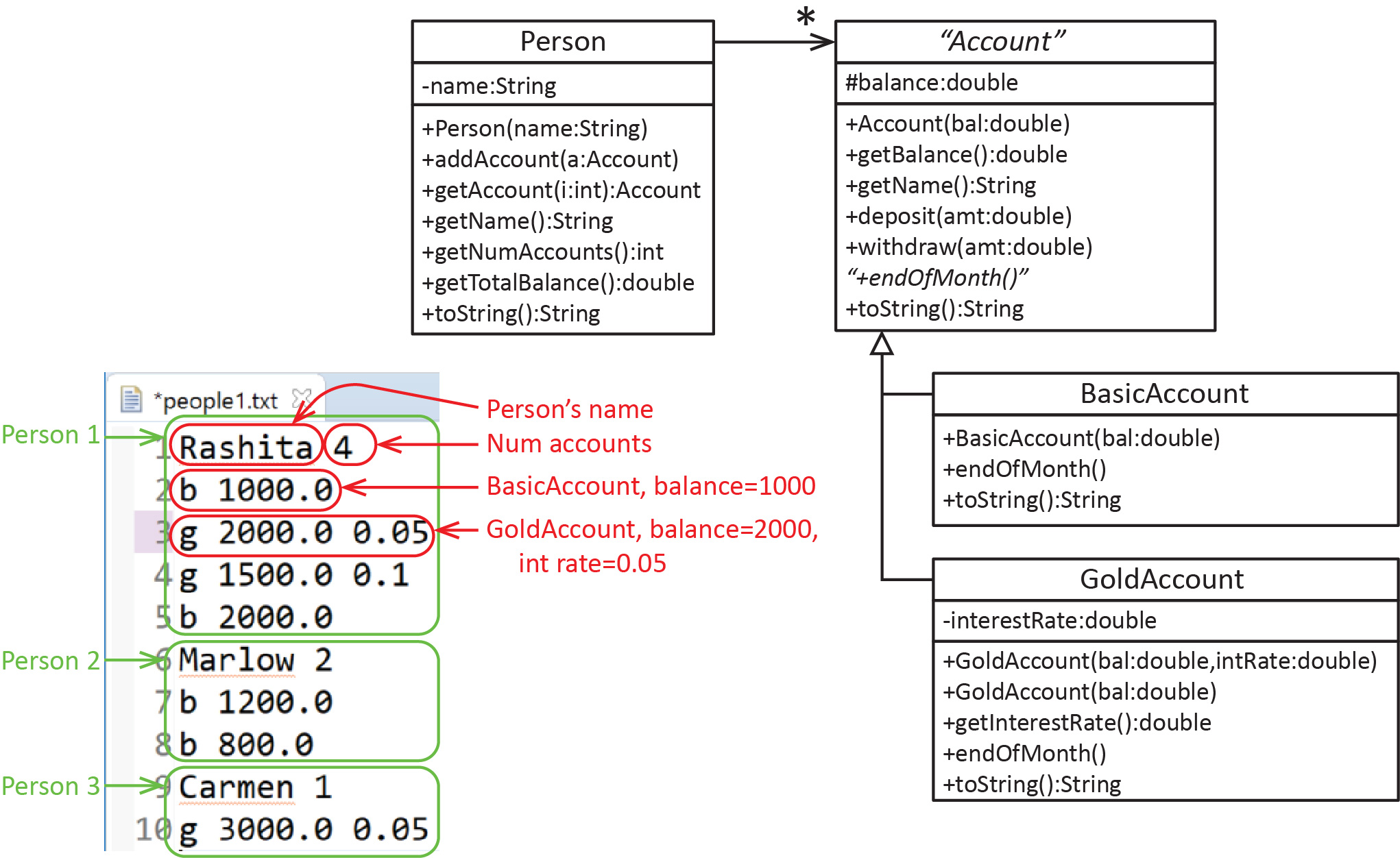
people.add(p);

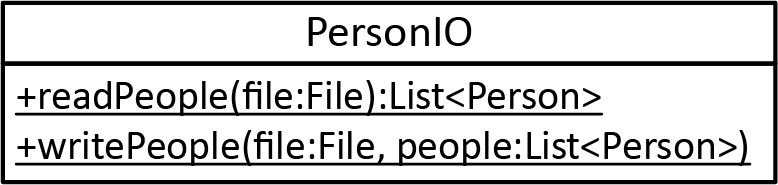
}

input.close()

## Exercises

1. (Support code and solution in *exercises\_person\_accounts\_ver1* package. Consider the classes shown below, which shows that a *Person* can have any number of *Accounts*. Next, consider a text file as shown below, on the left which explains how the data is structured.



Write a class, *PersonIO* with two static methods:

1. readPeople(file:File):List<Person> – Accepts a file argument and reads from that file *Person* and associated *Account* objects as shown above and returns a list of *Person* objects. You can assume the *file* exists and all data is valid. Write code to test.
2. writePeople(file:File, people:List<Person>) – Accepts a file argument and a list of *Person* objects. The method writes the *Person* and associated *Account* objects in the format shown above. Write code to test.

# String.split()

The code for the examples in this section is in the *example\_string\_split* package.

The approach we used in the examples above where we read tokens (*i.e.* the *next, nextInt, etc.* methods) is useful for simple text files. However, frequently, the data is arranged in a more complex fashion which forces us to read the file line-by-line. In other words, we read an entire line (*i.e.* using the *nextLine* method) into a string, and then break the string into pieces to see what is there and decide what to do with it. This is called *parsing* data. Towards that end, we must learn how to break a string into tokens.

The *String* class has a *split* method that breaks a string into “tokens” based on a delimiter and returns the tokens in an array.For example:

String s1 = "43.85 66.239 8.223";

String[] vals = s1.split(" ");

Results in the *vals* array containing: "43.85", "66.239", "8.223"

The delimiter in this case is a space (" "). Thus, everything between the spaces is a token.

Above, we say that we “split the string on a space.” Though this will work for our examples, technically there are many different types of spaces. This delimiter, “\s” will catch more of them. Remember, to put something with a slash in a string in Java, it must be delimited with a “\”. Thus, we would write it like this:

String[] vals = s1.split("\\s");

The example above will split on a single space. In the example below, there are two spaces between the first and second values; however, we are splitting on a single space.

String s1 = "43.85 66.239 8.223";

String[] vals = s1.split("\\s");

Thus, the *vals* array contains: "43.85", "", "66.239", "8.223"

|  |
| --- |
| **Tip**  When you split a string, you should always print it out, while debugging, to make sure you are getting what you think you are getting. For example, a quick way to print the array is:  System.***out***.println(Arrays.*toString*(vals));  Which produces: [43.85, , 66.239, 8.223]  Sometimes it is useful to surround the values with “’s:  **for**(**int** i=0; i<vals1.length; i++) {  String msg = String.*format*("\"%s\", ", vals1[i]);  System.***out***.print(msg);  }  System.***out***.println();  Which produces: "43.85", "", "66.239", "8.223", |

If we want to split on any number of spaces, we add a “+” to the delimiter to indicate, “one or more”):

String s1 = "43.85 66.239 8.223";

String[] vals = s1.split("\\s+");

Results in the *vals* array containing: "43.85", "66.239", "8.223"

To split a string any of multiple characters, we surround the characters with “[ ]”. For example, to split on any number of commas or spaces:

String s1 = "4,3 5,,,2, 8";

String[] vals = s1.split("[,\\s]+");

Results in the *vals* array containing: "4", "3", "5", "2", "8"

To split on common punctuation characters:

String s3 = "This. Is,,, funny; yes: why? now!";

String[] vals3 = s3.split("[.,;:?!\\s]+");

System.***out***.println(Arrays.*toString*(vals3));

Output: [This, Is, funny, yes, why, now]

The “delimiter” parameter is actually a regular expression, which can be composed to delimit and pattern-match sophisticated situations[[7]](#footnote-7).

## Example 3 – Read *Person*🡪\**Dog* with *nextLine* and *split*

A more robust way (code in *example\_read\_people\_and\_dogs* package, *ReadPeopleAndDogs\_Ver2* class) to do the example in a [previous section](#_Read_Example_2) is to read each line of the file, using *nextLine*, and then parse the data on each line (using *split*):

**while**( input.hasNext() ) {

String line = input.nextLine(); // Read Person line

String[] tokens = line.split(" "); // Split the line into tokens

String name = tokens[0]; // Extract name from

// first token

**int** numDogs = Integer.*parseInt*(tokens[1]); // Extract num dogs from

// second token

Person p = **new** Person(name); // Create Person

**for**(**int** i=0; i<numDogs; i++){ // Loop over number of dogs

String dogName = input.nextLine(); // Read dog name

Dog dog = **new** Dog(dogName); // Create Dog

p.addDog(dog); // Add Dog to Person

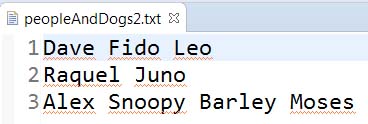
}

people.add(p); // Add Person to ArrayList

}

input.close()

## Example 4 – Read *Person*🡪\**Dog* without Number of Dogs Specified

The code for the example in this section is in the *example\_read\_people\_and\_dogs* package, in *ReadPeopleAndDogs\_Ver3* class. Let’s consider the same problem from the previous two sections, except that the text file doesn’t contain the number of dogs, and each person and their dogs are on the same line as shown below, on the right. Here is the approach, inside the loop:

1. Read the entire line:

String line = input.nextLine();

1. Split the line on a space:

String[] tokens = line.split("\\s");

1. The first token has the name:

String name = tokens[0];

Person p = **new** Person(name);

1. The rest of the tokens are dog names. Notice that the loop below starts at 1.

**for**(**int** i=1; i<tokens.length; i++){

String dogName = tokens[i];

Dog dog = **new** Dog(dogName);

p.addDog(dog);

}

people.add(p);

The complete code is shown below:

**public** **static** **void** main(String[] args) {

ArrayList<Person> people = **new** ArrayList<>();

File inFile = **new** File( "src/read\_examples3/peopleAndDogs\_ver3.txt" );

**try** {

Scanner input = **new** Scanner(inFile);

**while**(input.hasNext()) {

String line = input.nextLine();

String[] tokens = line.split("\\s");

String name = tokens[0];

Person p = **new** Person(name);

**for**(**int** i=1; i<tokens.length; i++){

String dogName = tokens[i];

Dog dog = **new** Dog(dogName);

p.addDog(dog);

}

people.add(p);

}

input.close();

}

**catch**( IOException e ) {

System.***out***.println(e);

}

**for**(Person p : people) {

System.***out***.println(p);

}

}

## General Approach to Reading

Below is the general approach for reading an arbitrary number of objects and putting them in a list (or a manager class).

AL<Stuff> things = new …

File f = new File(…

Scanner s = new Scanner(f)

while(s.hasNext())

String line = s.nextLine()

String[] tokens = line.split(“\\s+“)

// Use tokens to figure out what object needs to be created and the parameters, etc

Stuff stuff = new Stuff(...)

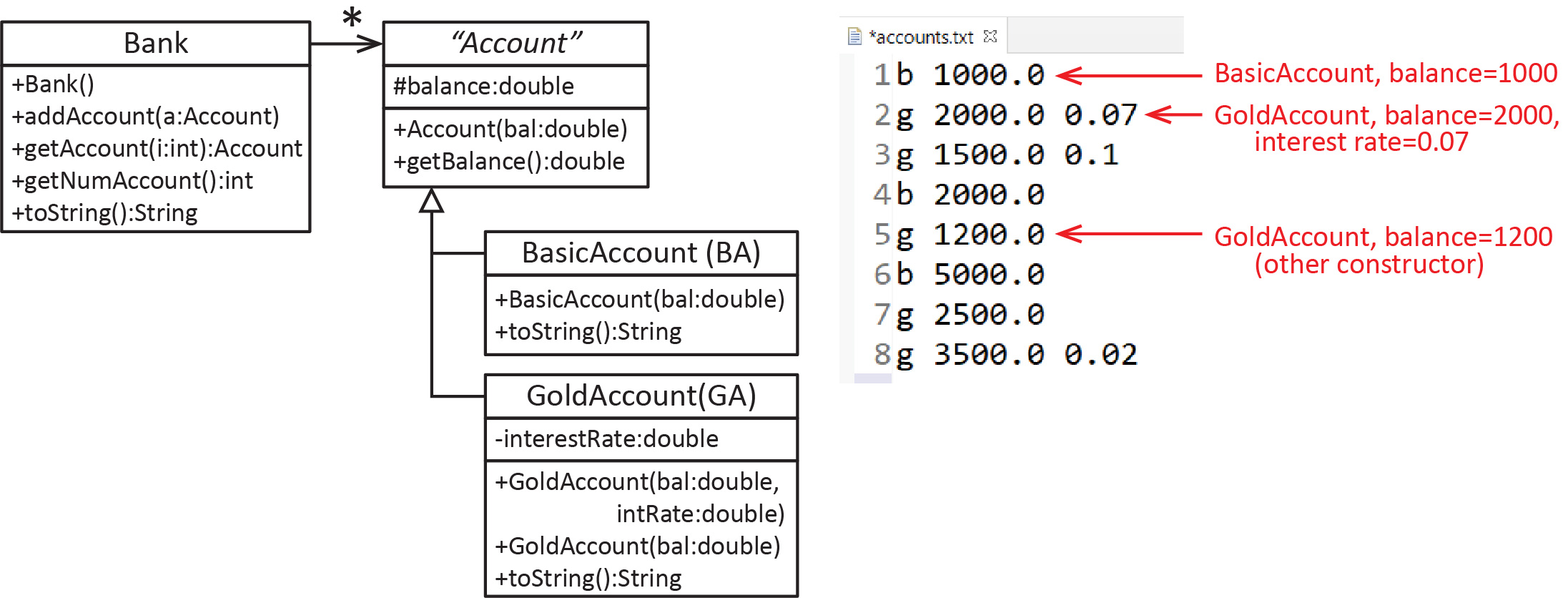
things.add(stuff)

s.close()

return things

## ­­Exercises

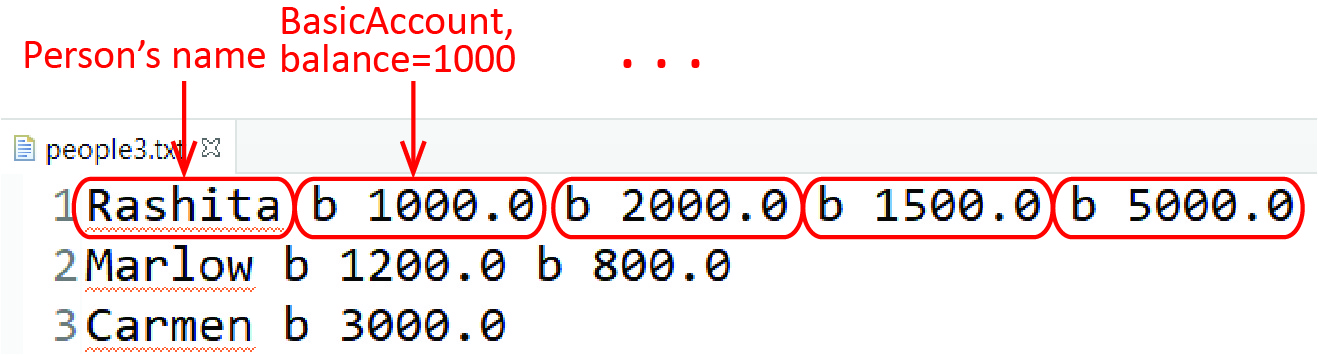
1. (Support code in *exercises\_bank\_accounts\_ver1* package). Consider the classes shown on the right. Notice that the *GoldAccount* has two constructors, one that takes an interest rate and one that does not. The one that doesn’t take an interest rate simply sets the interest rate to the default value, 0.05. Next, consider a text file as shown below which contains information for a number of accounts.



Write this static method:

readAccounts(file:File):Bank – Accepts a file argument and reads from that file the *Account* objects and adds them to a *Bank,* returning the *Bank* object. You can assume the *file* exists and all data is valid.

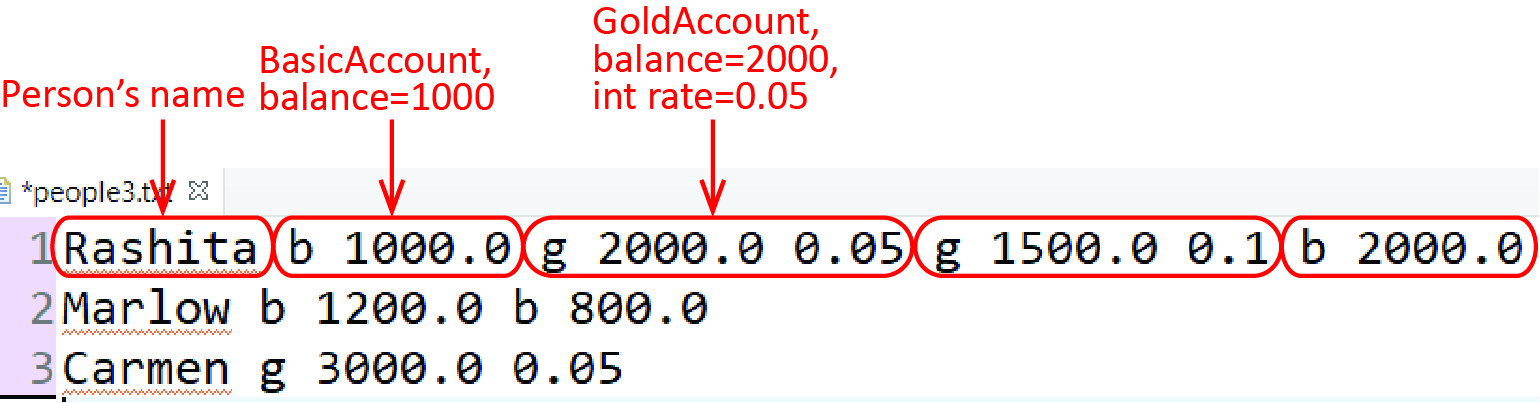
1. (Support code in *exercises\_person\_accounts\_ver2* package). Consider the classes from an earlier [Exercise](#_Exercises) (*Person* can have any number of *Accounts*); however, in this exercise, we will only consider *BasicAccounts.* Next, consider a text file as shown below which is similar to Exercise 1, except that the number of accounts for each person is not provided and there are only *BasicAccounts*.



Write this static method:

readPeople(file:File):List<Person> – Accepts a file argument and reads from that file *Person* and associated *BasicAccount* objects as shown above and returns a list of *Person* objects. You can assume the *file* exists and all data is valid. Your solution should also handle the case where a person has no accounts.

1. (Support code in *exercises\_person\_accounts\_ver3* package; however). Consider the classes from an earlier [Exercise](#_Exercises) (*Person* can have any number of *Accounts*). This problem is similar to the previous one except that we also include *GoldAccounts.* Consider a text file as shown below.



Write this static method:

readPeople(file:File):List<Person> – Accepts a file argument and reads from that file *Person* and associated *Account* objects as shown above and returns a list of *Person* objects. You can assume the *file* exists and all data is valid. Your solution should also handle the case where a person has no accounts.

This version of the problem is more challenging than the previous one with only *BasicAccounts.* Hint: two different approaches:

1. Inside major read loop: read a line, split into tokens, create a person from first token, introduce a *curToken* variable. Then, start a *while* loop, where each iteration processes the next account, advancing *curToken* as appropriate. Stop the inner loop when *curToken<tokens.length.*
2. Inside major read loop: read a line, split into tokens, create an *ArrayList* from the tokens, create a person from first token. Then, start a while loop that ends when the list is empty. Inside the while loop, remove the first token and decide what to do with it.

# Parsing Numbers

The code below is in the *example\_parsing\_numbers* package.

When parsing data, sometimes we need to know whether the characters in a string are actually a number, say an integer, or a double. The simplest way to do this is to write a helper method:

**public** **static** **boolean** isInteger(String str) {

**try** {

**int** x = Integer.*parseInt*(str);

**return** **true**;

}

**catch**(NumberFormatException nfe) {

**return** **false**;

}

}

Similarly, to determine if a string is a double:

**public** **static** **boolean** isDouble(String str) {

**try** {

**double** x = Double.*parseDouble*(str);

**return** **true**;

}

**catch**(NumberFormatException nfe) {

**return** **false**;

}

}

However, these are very inefficient if there are lots of values that are not integers, *i.e.* an exception is thrown. This is adequate for this course. Another approach is found in an [Appendix](#More_on_Parsing_Numbers). The best approach is to use regular expressions. Some sources:

|  |  |
| --- | --- |
| **Site** | **Description** |
| <https://regexlib.com/Search.aspx?k=integer&c=3&m=5&ps=20> | Matching integers |
| <http://www.java2s.com/example/java-book/write-code-to-check-if-a-string-is-positive-integer-using-regular-expr.html> | Matching a positive integer |
| <https://www.regular-expressions.info/floatingpoint.html> | Matching floating point numbers |
| <https://www.regular-expressions.info/quickstart.html> | Regex introduction |
| <https://www.regexbuddy.com/regex.html> | Regex app that contains a library of regexs |

## Example 5 – Read Select Words

The code for the example in this section is in the *example\_read\_previous\_words* package.

Write a method that accepts a *File* object and reads a file that contain a number of tokens spread across any number of lines. The method should return an *ArrayList* of strings composed in the following way: if a token is an integer, say *n*, then store the *n* words before the integer. For example, if the file is:

cat dog horse pen bird 2 house rain

rat dent

fire 4 guitar shoe pig 1 ant

wheel sack

Then the method returns an *ArrayList* with these elements: [pen, bird, rain, rat, dent, fire, pig]

You can assume the file exists and that the data in the file is valid, *i.e* you won’t be directed to read words that don’t exist.

The simplest approach is to read all the tokens in the text file into a list. Then, go through the list and ask each token if it is an integer. If it is, then start a loop to pull out the previous number of tokens requested.

**public** **static** ArrayList<String> readWords(File file) {

ArrayList<String> allTokens = **new** ArrayList<>();

ArrayList<String> words = **new** ArrayList<>();

// Read all tokens

**try** {

Scanner input = **new** Scanner(file);

**while**( input.hasNext() ) {

String token = input.next();

allTokens.add(token);

}

input.close();

}

**catch**( IOException e ) {

System.***out***.println(e);

}

// Extract words

**for**(**int** i=0; i<allTokens.size(); i++) {

String token = allTokens.get(i);

**if**(*isInteger*(token)) {

**int** num = Integer.*parseInt*(token);

**int** start = i-num;

**for**(**int** j=start; j<i; j++) {

words.add(allTokens.get(j));

}

}

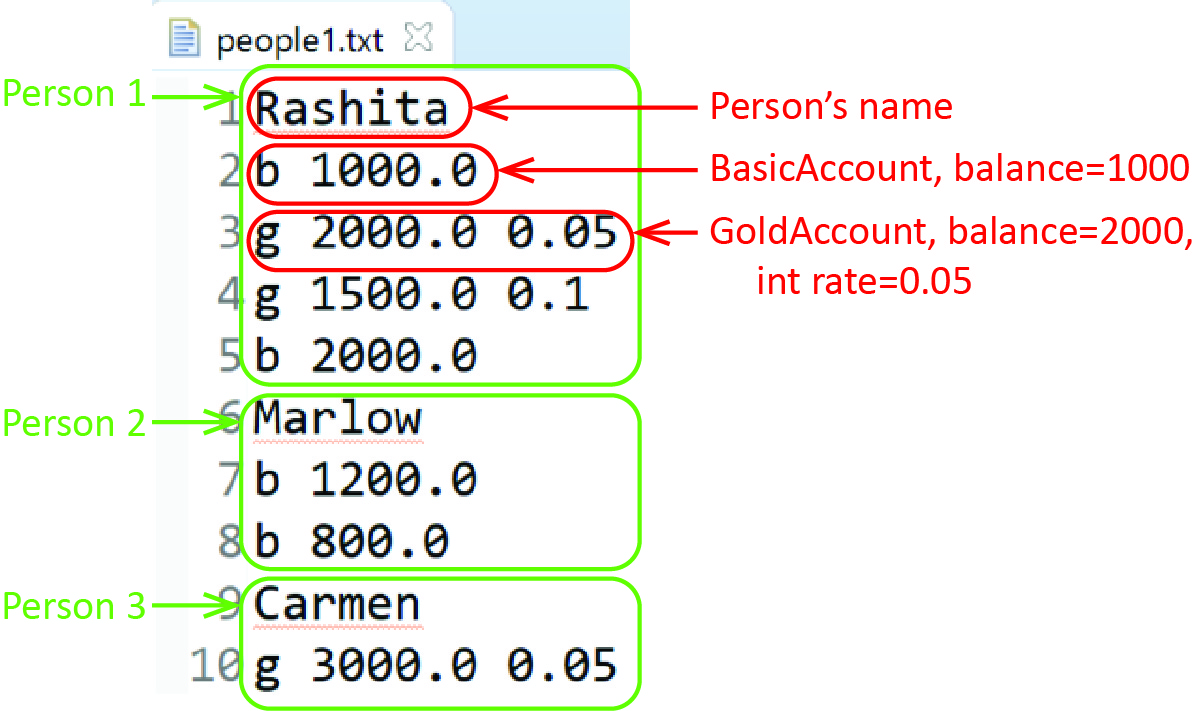
}

**return** words;

}

## Exercises

1. (Support code in *exercises\_person\_accounts\_ver4* package; however). Consider the classes from an earlier [Exercise](#_Exercises) (*Person* can have any number of *Accounts*). Next, consider a text file as shown below which is similar to Exercise 1, except that the number of accounts for each person is not provided.



Write this static method:

readPeople(file:File):List<Person> – Accepts a file argument and reads from that file *Person* and associated *Account* objects as shown above and returns a list of *Person* objects. You can assume the *file* exists and all data is valid. Your solution should also handle the case where a person has no accounts. Note: there are several ways to solve this. **This problem is a bit challenging!** It is a bit simpler (just a bit) if you use the approach in a previous [example](#_Read_Example_4). For example, I wrote a method that reads each line and puts them into an *ArrayList*, each line in the text file is an entry in the list. Then, process the list.

Appendix

1. The *InputStream* Classes

The java.io package is organized around the Decorator pattern. Consider some of the classes from the *InputStream* hierarchy. Typically, one or more *FilterInputStream* instances will wrap either a *FileInputStream* or an *ObjectInputStream.* Also shown are the Reader classes, also organized with the Decorator pattern as well, which are convenience classes for reading character strings.

E:\Data-Classes\CS 4322 - Software Engineering 2\Notes\03-Decorator\pics\d1.tif

A brief description of some of the classes above:

|  |  |
| --- | --- |
| **Class** | **Description** |
| FileInputStream | Reads raw bytes from a file |
| BufferedInputStream | Adds the ability to buffer the input. Uses private methods to fill the buffer using the inherited read from FilterInputStream. |
| DataInputStream | Reads Java primitive types. For example, *readInt* calls *read* on the *InputStream* four times and then uses bitwise operators to construct the integer. |
| FileReader | Convenience class for reading character files. |
| BufferedReader | Provides buffering and supplies the convenience method *readLine* |

The output classes are organized similarly around *OutputStream*.

1. Appending Text Files

The example in this section is in the *example\_append\_file* package.

Sometimes it is useful to add information onto then end of an existing file. This is called *appending* a file. We must use the *FileWriter* class to create an interface that allows us to append a text file. The constructor for *FileWriter* accepts a *File* object and has a second parameter that when set to *true* indicates that we want to append the file. Finally, the *PrintWriter* class has a constructor that accepts a *FileWriter* object.

For example, suppose we want to append the four integers in the array below to the end of an existing file, *output3.txt*:

int[] nums = {33, 44, 55, 66 };

File outFile = new File( "src\\textfile\_examples\\output3.txt" );

// Create a FileWriter

FileWriter fw = new FileWriter( outFile, true );

// Create a PrintWriter.

PrintWriter writer = new PrintWriter( fw );

// Loop over the array of numbers and append to the file.

for( int i=0; i<nums.length; i++ ) {

writer.println( nums[i] );

}

writer.close();

1. More on Parsing Numbers

The code below is the *example\_parsing\_numbers* package.

A much faster approach than the *try/catch* approach earlier in the notes, to determine if a string is an integer, is to check each character in the string:

**public** **static** **boolean** isIntegerByCharacter(String str) {

**if** (str == **null**) {

**return** **false**;

}

**int** length = str.length();

**if** (length == 0) {

**return** **false**;

}

**int** beg = 0;

**if** (str.charAt(0) == '-') {

**if** (length == 1) {

**return** **false**;

}

beg = 1;

}

**for** (**int** i=beg; i < length; i++) {

**char** c = str.charAt(i);

**if** (c < '0' || c > '9') {

**return** **false**;

}

}

**return** **true**;

}

The algorithm above has at least two errors. For example:

isIntegerByCharacter("+0")=false

isIntegerByCharacter("+100")=false

This is easily fixed by putting a check for a leading “+”. The *parseInt* approach consider earlier does work correctly:

isIntegerParseInt("+0")=true

isIntegerParseInt("+100")=true

Another problem is that it doesn’t handle leading or trailing spaces, which is easily fixed by using *trim* on the input.

There are at least two other approaches for validating an integer.[[8]](#footnote-8)

We could validate a double by looking at each character as we did with an integer. The code would be slightly more involved to make sure that there is at most one decimal point.

Here is a discussion on Stack Overflow[[9]](#footnote-9).

1. Other choices: database, XML file, JSON file, binary file [↑](#footnote-ref-1)
2. <https://www.khanacademy.org/computing/computers-and-internet/xcae6f4a7ff015e7d:computers/xcae6f4a7ff015e7d:from-electricity-to-bits/a/from-electricity-to-bits> [↑](#footnote-ref-2)
3. <https://cpe.ku.ac.th/~jim/java-io.html> [↑](#footnote-ref-3)
4. You can change the default location in Eclipse by choosing: Run, Run Configurations…. Then choose: the Arguments tab, scroll down and choose the Other radio button. Finally, choose: File System and navigate to the package. This specifies the path absolutely, beginning from the sourse drive, *e.g. c:/…/workspace/project\_name/src/package\_name*. [↑](#footnote-ref-4)
5. <https://docs.oracle.com/javase/tutorial/essential/io/index.html> [↑](#footnote-ref-5)
6. <https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html> [↑](#footnote-ref-6)
7. <https://www.vogella.com/tutorials/JavaRegularExpressions/article.html#common-matching-symbols> [↑](#footnote-ref-7)
8. <https://stackoverflow.com/questions/237159/whats-the-best-way-to-check-if-a-string-represents-an-integer-in-java/237204#237204> [↑](#footnote-ref-8)
9. <https://stackoverflow.com/questions/3133770/how-to-find-out-if-the-value-contained-in-a-string-is-double-or-not> [↑](#footnote-ref-9)