java4python

Release 3.0

Brad Miller and James Riely

February 24, 2015

There is a *search* and *genindex*.

A PDF version and source code are available.

Contents:

CHAPTER

ONE

PREFACE

1.1 Preface to the third edition

Welcome to *Java for Python Programmers*. This is a quick introduction to Java for programmers that already know another language, preferably Python.

We will begin by looking at a very simple Java program, just to see what the language looks like and how we get a program to run. Next, we will look at the basic syntax of Java, and relate it to python. We will then spend some time looking at inductively defined functions and array. Finally, we will look at how to define our own classes in Java.

This edition of the text is adapted to the data structures classes taught at DePaul. As our primary text, we use Algorithms 4e. The main changes are to the section on classes. I've also added a new section on inductive programming over arrays using iteration and recursion.

James Riely jriely@cs.depaul.edu December, 2013

1.2 Preface to the second edition

Welcome to *Java for Python Programmers*. This short ebook is an ongoing project to help Computer Science students who have had one or two semesters of Python learn the Java programming language. If you are not a part of that audience you may still find this a useful way to learn about Java. This book is written using the build on what you know philosophy. In order to help you learn Java I will start with a Python example and then implement the example in Java. Along the way we will examine the strengths, weaknesses and differences between those two languages.

This book does not attempt to replace the many good Java reference books that are available, in fact I use this in my course along with Horstman's *Core Java* volumes. Please feel free to use this book for yourself, or if it fits a class you are teaching you are welcome to use this as a resource for your own class.

I have published this article using a Creative Commons license to encourage you to use it, change it, and modify it for your own purposes. I would appreciate knowing what you think if you do use this book, and I would love to see any modifications or additions you make.

Brad Miller bmiller@luther.edu January, 2008

1.2.1 Shameless Plug

At Luther college we use Python for CS1 and CS2. When we decided to make the switch to introducing CS with Python we wanted to provide our students with two semesters of Python. The reason is that after one semester students are just getting comfortable with the language and it does not make sense to push them into a brand new language just as they are getting some comfort. The second reason is that Python really is a great language for teaching data structures. The language makes the algorithms evident by clearing away all of the syntactic clutter. So we (David Ranum and I) wrote a CS2 book called Problem Solving with Algorithms and Data Structures using Python. After we finished the CS2 book we decided to write our own CS1 book as well. This book will be available for Fall 2008 classes from Jones and Bartlett publishers. The tentative title is *Python in Context*. Please feel free to contact me for more information.

CHAPTER

TWO

GETTING STARTED

2.1 Why Java? Dynamic versus Static Languages

Python is a nice language for beginning programming for several reasons.

- The syntax is sparse and clear.
- The underlying model is very simple. Everything is an object.
- You can write powerful and interesting programs without a lot of work.

Python is representative of one kind of language, called a *dynamic* language. Dynamic languages can be *interpreted* directly, which means that the actual text of the program — the *source* code — is used while the program is running. In contrast, a *static* language is executed in two phases: first the program is translated from source code to *binary* code, and then the binary code is interpreted. Although the terms dynamic and static language are widely used, the distinction is a fuzzy one. Most execution engines do both translation and interpretation.

- *Static* refers to what the translater does. The translater is called a *compiler*.
- *Dynamic* refers to what the interpreter does.

Dynamic languages do *more* in the interpreter. By doing more in the interpreter, dynamic languages have more flexibility. By doing more in the compiler, static languages gain their own advantages.

- Compiled languages are fast. They gain speed in at least two ways:

  First, by distinguishing objects from builtin types — aka *base types*. In order to do basic arithmetic, a dynamic language may need to convert from an object to base type and then back; a static language avoids the conversions.

  Second, by performing name lookups in advance. Dynamic languages use the name of a method (or a derived selector) to look up the method during interpretation. Compilation makes this much more efficient, often resolving the actual method in advance.

- Compiled languages are excellent for writing large programs that are maintained over time. Python and other scripting languages require that the programmer keep track of a lot of information. For example if you set variable `x` to reference a turtle, and forget later that `x` is a turtle but try to invoke a string method on it, you will get an error. Compiled languages track this information using *types* for variables. This means that programmers need to keep track of less information informally, reducing the chance of miscommunication between to programmers.

  In addition compiled languages support a notion of *hiding*. Java allows you to declare some data to be `private`, meaning that it is only accessible by certain method and not others. Python uses naming conventions for the same thing; variables starting with __ are meant to be ignored by most methods, but there is nothing that enforces this.

Python is representative of a whole class of languages, sometimes referred to as *scripting* languages. Other languages in the same category as Python are Ruby and Perl. Java is representative of what I will call *industrial strength*

languages, which include C++, C# and Scala. Industrial strength languages are good for projects with several people working on the project where being formal and careful about what you do may impact lots of other people.

Why Java?

- Java is the most widely taught programming language.
- Java is one of the most widely used programming languages.
- Java is industrial strength used for large systems by large groups of people
- Java is simpler than C++ or Objective-C.

Although Java is an enormous language, the core language is very small. The features that we will use are common to every static language. If you want to program in C++ or Objective-C, you should learn Java first.

2.2 Hello World

A time honored tradition in Computer Science is to write a program called "hello world." The "hello world" program is simple and easy. There are no logic errors to make, so getting it to run relies only on understanding the syntax. To be clear lets look a a "complicated" version of hello world for Python:

```
def main():
    print "Hello World!"
```

Remember that we can define this program right at the Python command line and then run it:

```
>>> main()
"Hello World!"
>>>
```

Now lets look at the same program written in Java:

```
1 public class Hello {
2     public static void main(String[] args) {
3         System.out.println("Hello World!");
4     }
5 }
```

What we see is that at the core there are a few similarities, such as a main and the string "Hello World" However there is a lot more stuff around the edges that make it harder to see the core of the program. Do not worry! An important skill for a computer scientist is to learn what to ignore and what to look at carefully. You will soon find that there are some elements of Java that will fade into the background as you become used to seeing them. One thing that will help you is to learn a little bit about Java *Naming Conventions*.

2.3 Running from the command line

The first question you probably have about this little program is "How do I run it?" Running a Java program is not as simple as running a Python program. The first thing you need to do with a Java program is compile it. First we must type the hello world program into a file and save that file using the name `Hello.java` The file name must be the same as the public class you define in the file. Once we have saved the file we *compile* it from the command line as follows:

```
1 $ ls -l Hello.*
2 -rw-r--r--   1 bmiller  bmiller  117 Jul 19 17:46 Hello.java
3 $ javac Hello.java
4 $ ls -l Hello.*
```

```
-rw-r--r--   1 bmiller  bmiller  391 Jul 19 17:47 Hello.class
-rw-r--r--   1 bmiller  bmiller  117 Jul 19 17:46 Hello.java
```

The command `javac` compiles our java source code into compiled byte code and saves it in a file called `Hello.class`. `Hello.class` is a binary file so you won't learn much if you try to examine the class file with an editor. Hopefully you didn't make any mistakes, but if you did you may want to consult the *Common Mistakes* section for helpful hints on compiler errors.

Now that we have compiled our java source code we can run the compiled code using the `java` command.

```
$ java Hello
Hello World!
$
```

Now you may be wondering what good is that extra step? What does compiling do for us? There are a couple of important benefits we get from compiling:

- Early detection of errors
- Faster Program Execution

The job of the compiler is to turn your java code into language that the Java Virtual Machine (JVM) can understand. We call the code that the JVM understands *byte code*. The JVM interprets the byte code much like the Python interpreter interprets your Python. However since byte code is much closer to the native language of the computer it can run faster.

When the compiler does the translation it can find many different kinds of errors. For example if you make a typo the compiler will find the typo and point it out to you before you ever run the program. We will look at some examples of compiler errors shortly. Chances are you will create some on your own very soon too.

2.4 The Anatomy of HelloWorld

Now that we have run our hello world program, lets go back and look at it carefully to see what we can learn about the Java language.

```
1 public class Hello {
2     public static void main(String[] args) {
3         System.out.println("Hello World!");
4     }
5 }
```

This simple example illustrates a few very important rules:

1. Every Java program must define a class, all code is inside a class.
2. Everything in Java must have a type
3. Every Java program must have a function called `public static void main(String[] args)`

Lets take the hello world example a line at a time to see how these rules are applied. On line 1 we see that we are declaring a class called Hello. As rule 1 says all Java code resides inside a class. Unlike Python where a program can simply be a bunch of statements in a file, Java programs must be inside a class. So, we define a class, `Hello` is not a very useful class it has no instance variables, and only one method. You will also notice the curly brace `{` In Java blocks of code are identified by pairs of curly braces. The block starts with a `{` and ends with a `}`. You will notice that I indented my code that followed the left brace, but in Java this is only done by convention it is not enforced.

On the next line we start our method definition. The name of this method is:

```
public static void main(String[] args)!
```

Everything on this line is significant, and helps in the identification of this method. For example the following lines look similar but are in fact treated by Java as completely different methods:

- `public void main(String[] args)`
- `public static void main(String args)`
- `public static void main()`
- `void main(String args)`

Just digging in to this one line will take us deep into the world of Java, so we are going to start digging but we are not going to dig too deeply right away. Much of what could be revealed by this one line is better understood through other examples, so be patient.

- The first word, *public* indicates to the Java compiler that this is a method that anyone can call. We will see that Java enforces several levels of security on the methods we write, including *public*, *protected*, and *private* methods.
- The next word, *static* tells Java that this is a method that is part of the class, but is not a method for any one instance of the class. The kind of methods we typically wrote in Python required an instance in order for the method to be called. With a static method, the object to the left of the . is a class, not an instance of the class. For example the way that we would call the `main` method directly is: `Hello.main(parameter1)`. For now you can think of static methods the same way you think of methods in Python modules that don't require an instance, for example the math module contains many methods: sin, cos, etc. You probably evaluated these methods using the names `math.cos(90)` or `math.sin(60)`.
- The next word, `void` tells the Java compiler that the method `main` will not return a value. This is roughly analogous to omitting the return statement in a Python method. In other words the method will run to completion and exit but will not return a value that you can use in an assignment statement. As we look at other examples we will see that every Java function must tell the compiler what kind of an object it will return. This is in keeping with the rule that says everything in Java must have a type. In this case we use the special type called `void` which means no type.
- Next we have the proper name for the method: `main`. The rules for names in Java are similar to the rules in Python. Names can include letters, numbers, and the _. Names in Java must start with a letter.
- Finally we have the parameter list for the method. In this example we have one parameter. The name of the parameter is `args` however, because everything in Java must have a type we also have to tell the compiler that the value of `args` is an array of strings. For the moment You can just think of an array as being the same thing as a list in Python. The practical benefit of declaring that the method main must accept one parameter and the parameter must be a an array of strings is that if you call `main` somewhere else in your code and and pass it an array of integers or even a single string, the compiler will flag it as an error.

That is a lot of new material to digest in only a single line of Java. Lets press on and look at the next line: `System.out.println("Hello World!");`. This line should look a bit more familiar to you. Python and Java both use the dot notation for finding names. In this example we start with `System`. System is a class. Within the system class we find the object named `out`. The `out` object is the standard output stream for this program. Having located the `out` object Java will now call the method named `println(String s)` on that object. The `println` method prints a string and adds a newline character at the end. Anywhere in Python that you used the `print` function you will use the `System.out.println` method in Java.

Now there is one more character on this line that is significant and that is the `;` at the end. In Java the `;` signifies the end of a statement. Unlike Python where statements are almost always only one line long java statements can spread across many lines. The compiler knows it has reached the end of a statement when it encounters a `;`. This is a very important difference to remember. In Java the following statements are all legal and equivalent. I would not encourage you to write your code like this, but you should know that it is legal.

```
System.out.println("Hello World");
System.out.println("Hello World")
;
System.out.println
    (
     "Hello World"
    )     ;
System.
  out.
    println("Hello World")
    ;
```

The last two lines of the hello world program simply close the two blocks. The first or outer block is the class definition. The second or inner block is the function definition.

If we wanted to translate the Java back to Python we would have something like the following class definition.

```
class Hello(object):
    @staticmethod
    def main(args):
        print "Hello World!"
```

Notice that we used the decorator `@staticmethod` to tell the Python interpreter that `main` is going to be a static method. The impact of this is that we don't have to, indeed we should not, use `self` as the first parameter of the main method! Using this definition we can call the main method in a Python session like this:

```
>>> Hello.main("")
Hello World!
>>>
```

CHAPTER

THREE

BASICS

3.1 Base Types

One of the great things about Python is that all of the basic data types are objects. Integers are objects, floating point numbers are objects, lists are objects, everything. In Java that is not the case. In Java some of the most basic data types like integers and floating point numbers are not objects. The benefit of having these primitive data types be non-objects is that operations on the primitives are fast; however, this increases the complexity of the language.

In Java, it is very important to distinguish *base types* from *object types*. The base types are integers and floating point numbers. There are several integer types of different sizes.

integer type	size	values (approximately)
void	0 bits	none
boolean	1 bit	true, false
byte	8 bits	-128 .. 127
char	16 bits	-/+ 32,000 (5 decimal digits)
int	32 bits	-/+ 2 billion (10 decimal digits)
long	64 bits	-/+ 9 quintillion (19 decimal digits)

For the most part, we will use `int` to represent integer values. We may also use `long`, when we need to count above 2 billion. The types `void`, `boolean` and `char` are typically used for special purposes. We will not use `byte` often.

- `void` is used for the type of functions that do not return any value.
- `boolean` is used for conditional execution (`if` statements and loops).
- `char` is used for unicode characters

floating point type	size	precision
float	32 bits	24 bits (about 7 decimal digits)
double	64 bits	54 bits (about 16 decimal digits)

3.2 Strings

Strings are objects in Java, but they are important enough to have their own special status in the language.

Strings in Java and Python are quite similar. Like Python, Java strings are immutable — ie, the value of a string cannot change. However, manipulating strings in Java is not quite as obvious since Strings do not support an indexing or slicing operator. That is not to say that you can't index into a Java string — you can. You can also pull out a substring just as you can with slicing. The difference is that Java uses method calls where Python uses Operators.

In fact this is the first example of another big difference between Java and Python. Java does not support any operator overloading. Table 3 maps common Python string operations to their Java counterparts. For the examples shown in the table we will use a string variable called `str`.

Python	Java	Description
`str[3]`	`str.charAt(3)`	Return character in 3rd position
`str[2:5]`	`str.substring(2,4)`	Return substring from 2nd to 4th
`len(str)`	`str.length()`	Return the length of the string
`str.find('x')`	`str.indexOf('x')`	Find the first occurrence of x
`str.split()`	`str.split('\s')`	Split the string on whitespace into a list/array of strings
`str.split(',')`	`str.split(',')`	Split the string at `','` into a list/array of strings
`str1 + str2`	`str1.concat(str2)`	Concatenate two strings together
`str.strip()`	`str.trim()`	Remove any whitespace at the beginning or end

Java does support exactly one operator on strings: the plus operator can be used as shorthand for `concat`. Thus, you can write `str1 + str2` as shorthand for `str1.concat(str2)`.

3.3 Conversions

Java allows for conversions between the base types and between strings and base types. Unlike python, these conversions must often be written explicitly. Some of these conversions may result in a loss of information if the source number is too large or has too great a precision to be stored in the target.

The following table shows the syntax for some conversions in the typical case where a variable is used to store the converted value.

Conversion	Syntax	Lossy?
`int` from `boolean`	`anInt = aBoolean ? 1 : 0;`	No
`int` from `char`	`anInt = aChar;`	No
`int` from `long`	`anInt = (int) aLong;`	Yes
`int` from `float`	`anInt = (int) aFloat;`	Yes
`int` from `double`	`anInt = (int) aDouble;`	Yes
`int` from `String`	`anInt = Integer.parseInt(aString);`	Yes
`float` from `int`	`aFloat = int;`	Yes
`float` from `long`	`aFloat = aLong;`	Yes
`float` from `double`	`aFloat = (float) aDouble;`	Yes
`float` from `String`	`aFloat = Float.parseFloat(aString);`	Yes
`double` from `int`	`aDouble = int;`	No
`double` from `long`	`aDouble = aLong;`	Yes
`double` from `float`	`aDouble = aFloat;`	No
`double` from `String`	`aDouble = Double.parseDouble(aString);`	Yes
`String` from `boolean`	`aString = Boolean.toString(aBoolean)`	No
`String` from `int`	`aString = Integer.toString(anInt)`	No
`String` from `long`	`aString = Long.toString(aLong)`	No
`String` from `float`	`aString = Float.toString(aFloat)`	No
`String` from `double`	`aString = Double.toString(aDouble)`	No

3.4 Example: Fahrenheit to Celsius

Lets go back in time and look at another of our very early Python programs. Here is a simple Python function to convert a Fahrenheit temperature to Celsius.

```
def main():
    fahr = input("Enter the temperature in F: ")
    cel = (fahr - 32) * 5.0/9.0
    print "the temperature in C is: ", cel
```

Next, lets look at the Java Equivalent.

```
import java.util.Scanner;

public class TempConv {
    public static void main(String[] args) {
        double fahr;
        double cel;
        Scanner in;

        in = new Scanner(System.in);
        System.out.println("Enter the temperature in F: ");
        fahr = in.nextDouble();

        cel = (fahr - 32) * 5.0/9.0;
        System.out.println("The temperature in C is: " + cel);

        System.exit(0);
    }

}
```

There are several new concepts introduced in this example. We will look at them in the following order:

- Import
- Variable Declaration
- Input/Output and the Scanner Class

3.4.1 Import

In Java you can use any class that is available without having to import the class subject to two very important conditions:

1. The javac and java must know that the class exists.
2. You must use the full name of the class

You first question might be how do the `java` and `javac` commands know that certain classes exist. The answer is the following:

1. Java knows about all the classes that are defined in .java and .class files in your current working directory.
2. Java knows about all the classes that are shipped with java.
3. Java knows about all the classes that are included in your `CLASSPATH` environment variable. Your `CLASSPATH` environment variable can name two kinds of structures.
   (a) A jar file that contains java classes
   (b) Another unix directory that contains java class files

You can think of the import statement in Java as working a little bit like the `from module import xxx` statement in Python. However, behind the scenes the two statements actually do very different things. The first important difference to understand is that the class naming system in Java is very hierarchical. The *full* name of the Scanner class is really `java.util.Scanner`. You can think of this name as having two parts: The first part `java.util` is called the *package* and the last part is the class. We'll talk more about the class naming system a bit later. The second important difference is that it is the Java class loader's responsibility to load classes into memory, not the import statement's.

So, what exactly does the import statement do? What it does is tell the compiler that we are going to use a shortened version of the class's name. In this example we are going to use the class `java.util.Scanner` but we can refer to it as just `Scanner`. We could use the `java.util.Scanner` class without any problem and without any import statement provided that we always referred to it by its full name. As an Experiment you may want to try this yourself. Remove the import statement and change the string Scanner to `java.util.Scanner` in the rest of the code. The program should still compile and run.

3.4.2 Declaring Variables

Here is where we run into one of the most important differences between Java and Python. Python is a *dynamically typed* language. In a dynamically typed language a variable can refer to any kind of object at any time. When the variable is used, the interpreter figures out what kind of object it is. Java is a *statically typed* language. In a statically typed language the association between a variable and the type of object the variable can refer to is determined when the variable is *declared*. Once the declaration is made it is an error for a variable to refer to an object of any other type.

In the example above, lines 5—7 contain variable declarations. Specifically we are saying that `fahr` and `cel` are going to variables holding values of type `double`. The variable `in` will reference a Scanner object. This means that if we were to try an assignment like `fahr = "xyz"` the compiler would generate an error because `"xyz"` is a string and `fahr` is supposed to be a double.

For Python programmers the following error is likely to be even more common. Suppose we forgot the declaration for `cel` and instead left line 6 blank. What would happen when we type `javac TempConv.java` on the command line?

```
TempConv.java:13: cannot find symbol
symbol  : variable cel
location: class TempConv
         cel = (fahr - 32) * 5.0/9.0;
         ^
TempConv.java:14: cannot find symbol
symbol  : variable cel
location: class TempConv
         System.out.println("The temperature in C is: " + cel);
                                                          ^
2 errors
```

When you see the first kind of error, where the symbol is on the left side of the equals sign it usually means that you have not declared the variable. If you have ever tried to use a Python variable that you have not initialized the second error message will be familiar to you. The difference here is that we see the message before we ever try to test our program. More common error messages are discussed in the section *Common Mistakes*.

The general rule in Java is that you must decide what kind of an object your variable is going to reference and then you must declare that variable before you use it. There is much more to say about the static typing of Java but for now this is enough.

3.4.3 Input / Output / Scanner

In the previous section you saw that we created a `Scanner` object. In Java Scanner objects make getting input from the user, a file, or even over the network relatively easy. In our case we simply want to ask the user to type in a number at the command line, so in line 9 we construct a Scanner by calling the constructor and passing it the `System.in` object. Notice that this Scanner object is assigned to the name `in`, which we declared to be a `Scanner` on line 7. `System.in` is similar to `System.out` except of course it is used for input. If you are wondering why we must create a Scanner to read data from `System.in` when we can write data directly to `System.out` using `println`, you are not alone. We will talk about the reasons why this is so later when we talk in depth about Java streams. You

will also see in other examples that we can create a Scanner by passing the Scanner a File object. You can think of a scanner as a kind of "adapter" that makes low level objects easier to use.

On line 11 we use the Scanner object to read in a number. Here again we see the implications of Java being a strongly typed language. Notice that we must call the method `nextDouble`. Because the variable `fahr` was declared as a double. So, we must have a function that is guaranteed to return each kind of object we might want to read. In this case we need to read a `double` so we call the function `nextDouble`. The compiler matches up these assignment statments and if you try to assign the results of a method call to the wrong kind of variable it will be flagged as an error.

Table 2 shows you some commonly used methods of the scanner class. There are many more methods supported by this class and we will talk about how to find them in the next chapter.

Return type	Method name	Description
boolean	hasNext()	returns true if more data is present
boolean	hasNextInt()	returns true if the next thing to read is an integer
boolean	hasNextFloat()	returns true if the next thing to read is a float
boolean	hasNextDouble()	returns true if the next thing to read is a double
int	nextInt()	returns the next thing to read as an integer
Float	nextFloat()	returns the next thing to read as a float
double	nextDouble()	returns the next thing to read as a double
String	next()	returns the next thing to read as a String

Java applications may also employ a graphical user interface (GUI). Lets look at a version of our temperature control application that uses dialog boxes for input and output.

```
1  import javax.swing.*;
2
3  public class TempConvGUI {
4
5      public static void main(String[] args) {
6          String fahrString;
7          double fahr, cel;
8
9          fahrString = JOptionPane.showInputDialog("Enter the temperature in F");
10         fahr = Double.parseDouble(fahrString);
11         cel = (fahr - 32) * 5.0/9.0;
12
13         JOptionPane.showMessageDialog(null,"The temperature in C is, " + cel);
14     }
15
16 }
```

This example illustrates a couple of interesting points:

First, the function call `JOptionPane.showInputDialog` pops up a dialog box to allow you to enter a temperature. But, since you could enter anything into the text input box it returns a `String`. On the next line the string is converted into a `double` by the function `Double.parseDouble`. This is similar to what happens in Python when you call `float()` with either a string or an integer as the argument.

The next dialog box is `JOptionPane.showMessageDialog`. Notice that the first parameter is `null` In Java `null` serves the same purpose as `None` in Python. The first parameter is null because we do not have a 'main window' for this little application. When we look at creating full blown java programs with user interfaces, we will learn more about this parameter.

The second parameter is `"The temperature in C is, " + cel`. Now you may be thinking to yourself that this must surely be a violation of the strong typing I have been describing to you. After all you should not be able to add together a string and a double right? You are correct, however, all java objects have a method called `tostring`. The `tostring` method acts much like the Python method `__str__()` and is called automatically by the compiler

whenever it makes sense to convert a Java object to a string.

3.5 List

Lets look at another early Python program. We are going to read numbers from a file and produce a histogram that shows the frequency of the various numbers. The data file we will use has one number between 0 and 9 on each line of the file. Here is a simple Python program that creates and prints a histogram.

```
def main():
    count = [0]*10
    data = open('test.dat')

    for line in data:
        count[int(line)] = count[int(line)] + 1

    idx = 0
    for num in count:
        print idx, " occured ", num, " times."
        idx += 1
```

Now if we run this program on a data file that looks like this:

```
9 8 4 5 3 5 2 1 5
```

We will get output that looks like this:

```
0 occurred 0 times
1 occurred 1 times
2 occurred 1 times
3 occurred 1 times
4 occurred 1 times
5 occurred 3 times
6 occurred 0 times
7 occurred 0 times
8 occurred 1 times
9 occurred 1 times
```

Lets review what is happening in this little program. In the first line we create a list and initialize the first 10 positions in the list to be 0. Next we open the data file called 'test.dat' Third, we have a loop that reads each line of the file. As we read each line we convert it to an integer and increment the counter at the position in the list indicated by the number on the line we just read. Finally we iterate over each element in the list printing out both the position in the list and the total value stored in that position.

To write the Java version of this program we will have to introduce several new Java concepts. First, you will see the Java equivalent of a list, called an `ArrayLlist`. Next you will see three different kinds of loops used in Java. Two of the loops we will use are going to be very familiar, the third one is different from what you are used to in Python but is easy when you understand the syntax:

while Used with boolean expression for loop exit condition.

for Used to iterate over a sequence. This is very similar to `for i in xxx` where xxx is a list or string or file.

for Used to iterate through a sequence of numbers. This is most similar to for `i in range()`, except the syntax is different.

Here is the Java code needed to write the exact same program:

```
1 import java.util.Scanner;
2 import java.util.ArrayList;
```

```
3  import java.io.File;
4  import java.io.IOException;
5
6  public class Histo {
7
8      public static void main(String[] args) {
9          Scanner data = null;
10         ArrayList<Integer> count;
11         int idx;
12
13         try {
14                 data = new Scanner(new File("test.dat"));
15         }
16         catch ( IOException e) {
17             System.out.println("Sorry but I was unable to open your data file");
18             e.printStackTrace();
19             System.exit(0);
20         }
21
22         count = new ArrayList<Integer>(10);
23         for (int i =0; i<10;i++) {
24             count.add(i,0);
25         }
26
27         while(data.hasNextInt()) {
28             idx = data.nextInt();
29             count.set(idx,count.get(idx)+1);
30         }
31
32         idx = 0;
33         for(int i : count) {
34             System.out.println(idx + " occured " + i + " times.");
35             idx++;
36         }
37     }
38 }
```

Before going any further, I suggest you try to compile the above program and run it on some test data that you create.

Now, lets look at what is happening in the Java source. As usual we declare the variables we are going to use at the beginning of the method. In this example we are declaring a Scanner variable called data, an integer called idx and an `ArrayList` called count. However, there is a new twist to the `ArrayList` declaration. Unlike Python where lists can contain just about anything, in Java we let the compiler know what kind of objects our array list is going to contain. In this case the `ArrayList` will contain Integers. The syntax we use to declare what kind of object the list will contain is the <``*Type*>`` syntax.

Technically, you don't *have* to declare what is going to be on an array list. The compiler will allow you to leave the <``*Type*>`` off the declaration. If you don't tell Java what kind of object is going to be on the list Java will give you a warning message like this:

```
Note: Histo.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Without the <Integer> part of the declaration Python simply assumes that *any* object can be on the list. However, without resorting to an ugly notation called casting, you cannot do anything with the objects on a list like this! So, if you forget you will surely see more errors later in your code. (Try it and see what you get)

Lines 13—20 are required to open the file. Why so many lines to open a file in Java? The additional code mainly comes form the fact that Java forces you to reckon with the possibility that the file you want to open is not going to be

there. If you attempt to open a file that is not there you will get an error. A try/catch construct allows us to try things that are risky, and gracefully recover from an error if one occurs. The following example shows the general structure of a try catch block.

```
try {
   Put some risky code in here.... like opening a file
}
catch (Exception e) {
   If an error happens in the try block an exception is thrown.
   We will catch that exception here!
}
```

Notice that in line 16 we are catching an `IOException`. In fact we will see later that we can have multiple catch blocks to catch different types of exceptions. If we want to be lazy and catch any old exception we can catch an `Exception` which is the parent of all exceptions.

On line 22 we create our array list and give it an initial size of 10. Strictly speaking it is not necessary to give the `ArrayList` any size. It will grow or shrink dynamically as needed just like a list in Python. On line 23 we start the first of three loops. The for loop on lines 23–25 serves the same purpose as the Python statement `count = [0]*10`, that is it initializes the first 10 positions in the `ArrayList` to hold the value 0.

The syntax of this for loop probably looks very strange to you, but in fact it is not too different from what happens in Python using range. In fact `for(int i = 0; i < 10; i++)` is exactly equivalent to the Python `for i in range(10)` The first statement inside the parenthesis declares and initializes a loop variable i. The second statement is a Boolean expression that is our exit condition. In other words we will keep looping as long as this expression evaluates to true. The third clause is used to increment the value of the loop variable at the end of iteration through the loop. In fact `i++` is Java shorthand for `i = i +` Java also supports the shorthand `i--` to decrement the value of i. Like Python you can also write `i += 2` as shorthand for `i = i + 2` Try to rewrite the following Python for loops as Java for loops:

- `for i in range(2,101,2)`
- `for i in range(1,100)`
- `for i in range(100,0,-1)`
- `for x,y in zip(range(10),range(0,20,2))` [hint, you can separate statements in the same clause with a ,]

The next loop (lines 27–30) shows a typical Java pattern for reading data from a file. Java while loops and Python while loops are identical in their logic. In this case we will continue to process the body of the loop as long as `data.hasNextInt()` returns true.

Line 29 illustrates another important difference between Python and Java. Notice that in Java we can not write `count[idx] = count[idx] + 1`. This is because in Java there is no overloading of operators. Everything except the most basic math and logical operations is done using methods. So, to set the value of an `ArrayList` element we use the `set` method. The first parameter of `set` indicates the index or position in the `ArrayList` we are going to change. The next parameter is the value we want to set. Notice that once again we cannot use the indexing square bracket operator to retrieve a value from the list, but we must use the `get` method.

The last loop in this example is similar to the Python for loop where the object of the loop is a Sequence. In Java we can use this kind of for loop over all kinds of sequences, which are called Collection classes in Java. The for loop on line 33 `for(int i : count)` is equivalent to the Python loop `for i in count:` This loop iterates over all of the elements in the ArrayList called count. Each time through the loop the int variable i is bound to the next element of the `ArrayList`. If you tried the experiment of removing the `<Integer>` part of the `ArrayList` declaration you probably noticed that you had an error on this line. Why?

3.6 Arrays

As I said at the outset of this Section we are going to use Java `ArrayLists` because they are easier to use and more closely match the way that Python lists behave. However, if you look at Java code on the internet or even in your Core Java books you are going to see examples of something called arrays. In fact you have already seen one example of an array declared in the 'Hello World' program. Lets rewrite this program to use primitive arrays rather than array lists.

```
1  import java.util.Scanner;
2  import java.io.File;
3  import java.io.IOException;
4
5  public class HistoArray {
6      public static void main(String[] args) {
7          Scanner data = null;
8          int[] count = {0,0,0,0,0,0,0,0,0,0};
9          int idx;
10
11
12
13         try {
14                 data = new Scanner(new File("test.dat"));
15         }
16         catch ( IOException e) {
17             System.out.println("Sorry but I was unable to open your data file");
18             e.printStackTrace();
19             System.exit(0);
20         }
21
22         while(data.hasNextInt()) {
23             idx = data.nextInt();
24             count[idx] = count[idx] + 1;
25         }
26
27         idx = 0;
28         for(int i : count) {
29             System.out.println(idx + " occured " + i + " times.");
30             idx++;
31         }
32     }
33 }
```

The main difference between this example and the previous example is that we declare count to be an Array of integers. We also can initialize short arrays directly using the syntax shown on line 8. Then notice that on line 24 we can use the square bracket notation to index into an array.

3.7 Conditionals

Conditional statements in Python and Java are very similar. In Python we have three patterns:

```
if condition:
    statement1
    statement2
    ...
```

In Java this same pattern is simply written as:

```
if (condition) {
    statement1
    statement2
    ...
}
```

Once again you can see that in Java the curly braces define a block rather than indentation. In Java the parenthesis around the condition are required because if is technically a function that evaluates to True or False.

```
if condition:
    statement1
    statement2
    ...
else:
    statement1
    statement2
    ...
```

In Java this is written as:

```
if (condition) {
    statement1
    statement2
    ...
} else {
    statement1
    statement2
    ...
}
```

Java does not have an elif pattern like Python. In Java you can get the functionality of an elif statement by nesting if and else. Here is a simple example in both Python and Java.

```
if grade < 60:
    print 'F'
elif grade < 70:
    print 'D'
elif grade < 80:
    print 'C'
elif grade < 90:
    print 'B'
else:
    print 'A'
```

In Java we have a couple of ways to write this

```
if (grade < 60>) {
    System.out.println('F');
} else {
    if (grade < 70) {
        System.out.println('F');
    } else {
        if (grade < 80) {
            System.out.println('F');
        } else {
            if (grade < 90) {
                System.out.println('F');
            } else {
                System.out.println('F');
```

```
        }
```

We can get even closer to the elif statement by taking advantage of the Java rule that a single statement does not need to be enclosed in curly braces. Since the if is the only statement used in each else we can get away with the following.

```
if (grade < 60) {
    System.out.println('F');
} else if (grade < 70) {
    System.out.println('D');
} else if (grade < 80) {
    System.out.println('C');
} else if (grade < 90) {
    System.out.println('B');
} else  System.out.println('A');
```

Java also supports a `switch` statement that acts something like the elif statement of Python under certain conditions. To write the grade program using a switch statement we would use the following:

```
int tempgrade = grade / 10;
switch(tempgrade) {
case 10:
case 9:
    System.out.println('A');
    break;
case 8:
    System.out.println('B');
    break;
case 7:
    System.out.println('C');
    break;
case 6:
    System.out.println('A');
    break;
default:
    System.out.println('F');
}
```

The `switch` statement is not used very often, and I recommend you do not use it! First, it is not as powerful as the `else if` model because the switch variable can only be compared for equality with an integer or enumerated constant. Second it is very easy to forget to put in the `break` statement. If the break statement is left out then then the next alternative will be automatically executed. For example if the grade was 95 and the `break` was omitted from the `case 9:` alternative then the program would print out both A and B. The conditionals used in the if statement can be boolean variables, simple comparisons, and compound boolean expressions.

Java also supports the boolean expression. `condition ? trueValue : falseValue` This expression can be used to test a condition as part of an assignment statement. For example `a = a % 2 == 0 ? a*a : 3*x -1` In the previous assignment statement the expression `a%2 ==0` is first checked. If it is true then a is assigned the value `a * a` if it is false then a is assigned the value of `3*x-1`. Of course all of this could have been accomplished using a regular if else statement, but sometimes the convenience of a single statement is too much to resist.

3.8 Loops and Iteration

You have already seen a couple of examples of iteration and looping in Java. So this section will just serve as a reference for the differences in Snytax.

In Python the easiest way to write a definite loop is using the for loop in conjunction with the range function. For example:

```
for i in range(10):
   print i
```

In Java we would write this as:

```
for (int i = 0; i < 10; i++ ) {
    System.out.println(i);
}
```

Recall that the `range` function provides you with a wide variety of options for controlling the value of the loop variable.

```
range(stop)
range(start,stop)
range(start,stop,step)
```

The Java for loop is really analogous to the last option giving you explicit control over the starting, stopping, and stepping in the three clauses inside the parenthesis. You can think of it this way:

```
for (start clause; stop clause; step clause) {
    statement1
    statement2
    ...
}
```

If you want to start at 100, stop at 0 and count backward by 5 the Python loop would be written as:

```
for i in range(100,-1,-5):
    print i
```

In Java we would write this as:

```
for (int i = 100; i >= 0; i -= 5)
    System.out.println(i);
```

In Python the for loop can also iterate over any sequence such as a list, a string, or a tuple. Java also provides a variation of its for loop that provides the same functionality in its so called `for each` loop.

In Python we can iterate over a list as follows:

```
l = [1, 1, 2, 3, 5, 8, 13, 21]
for fib in l:
   print fib
```

In Java we can iterate over an ArrayList of integers too:

```
ArrayList<Integer> l = new ArrayList<Integer>();
l.add(1); l.add(1); l.add(2); l.add(3);
for (int i : l) {
    System.out.println(i)
}
```

This example stretches the imagination a bit, and in fact points out one area where Java' s primitive arrays are easier to use than an array list. In fact all primitive arrays can be used in a `for each` loop.

```
int l[] = {1,1,2,3,5,8,13,21};
for(int i : l) {
    System.out.println(i);
}
```

To iterate over the characters in a string in Java do the following:

```
String t = "Hello World";
for (char c : t.toCharArray()) {
    System.out.println(c);
}
```

Both Python and Java support the while loop. Recall that in Python the while loop is written as:

```
while  condition:
   statement1
   statement2
   ...
```

In Java we add parenthesis and curly braces to get:

```
while (condition) {
    statement1
    statement2
    ...
}
```

Java adds an additional, if seldom used variation of the while loop called the do loop. The do loop is very similar to while except that the condition is evaluated at the end of the loop rather than the beginning. This ensures that a loop will be executed at least one time. Some programmers prefer this loop in some situations because it avoids an additional assignment prior to the loop. For example:

```
do {
    statement1
    statement2
    ...
} while (condition);
```

3.9 Common Mistakes

- **Forgetting to declare your variables.**

  ```
  Histo.java:21: cannot find symbol
  symbol  : variable count
  location: class Histo
          count = new ArrayList<Integer>(10);
          ^
  ```

- **Not importing a class.**

  ```
  Histo.java:9: cannot find symbol
  symbol  : class Scanner
  location: class Histo
          Scanner data = null;
          ^
  ```

- **Forgetting to use the new keyword to create an object.** Here's an example of the error message that occurs when you forget to use the new keyword. Notice that the message is pretty unhelpful. Java *thinks* you are trying to call the Method Scanner, but there are two problems. First Scanner is not really a method it is a constructor.:

  ```
  Histo.java:14: cannot find symbol
  symbol  : method Scanner(java.io.File)
  location: class Histo
                  data = Scanner(new File("test.dat"));
                         ^
  ```

- **Forgetting a Semicolon.**

```
Histo.java:19:
';' expected
            System.exit(0);
            ^
```

- **Forgetting to declare the kind of object in a container.**

```
Note: Histo.java uses unchecked or unsafe operations. Note:
Recompile with -Xlint:unchecked for details.
```

3.10 Naming Conventions

Java has some very handy naming conventions.

- Class names always start with an upper case letter. For example, `Scanner`, `System`, `Hello`
- Method names always start with a lower case letter, and use camelCase to represent multiword method names. for example `nextInt()`
- Instance variables of a class start with a lower case letter and use camelCase
- Constants are in all upper case letters. for example `Math.MAXINT`

3.11 Java Documentation Online

All Java class libraries are documented and available online. Here are two good resources for you to use:

- JavaDoc The Javadocs.org website provides a nice searchable interface. Search for a classname and you will get the documentation you are looking for.
- JavaAPI contains the same information but in a browsable format. If you don't know the class name exactly this is a good way to see what is close.

In general the Javadoc page for any class contains information about:

- Where this class falls in the class hierarchy. What classes are its parents and what classes are its decendents.
- A summary and some examples of using the class.
- A summary listing of instance variables
- A summary listing of Constructors
- A summary listing of Methods
- Detailed documentation on constructors and methods.

Typically the Javadoc pages are constructed from the source code where the class is implemented. This encourages Java programmers to do a good job of documenting their code, while providing a user friendly way to read the documentation without looking at the code directly.

CHAPTER

FOUR

LISTS AND MAPS

4.1 Numeric

One of the great things about Python is that all of the basic data types are objects. Integers are objects, floating point numbers are objects, lists are objects, everything. In Java that is not the case. In Java some of the most basic data types like integers and floating point numbers are not objects. The benefit of having these primitive data types be non-objects is that operations on the primitives are fast. The problem is that it became difficult for programmers to combine objects and non-objects in the way that we do in Python. So, eventually all the non-object primitives ended up with Objectified versions.

Primitive	Object
int	Integer
float	Float
double	Double
char	Char
boolean	Boolean

In older versions of Java it was the programmers responsibility to convert back and forth from a primitive to an object whenever necessary. This processing of converting a primitive to an object was called "boxing." The reverse process is called "unboxing." In Java 5, the compiler became smart enough to know when to convert back and forth and is called "autoboxing." In general, you should always use the primitive form, since computation always happens on the unboxed form, and boxing and unboxing are not free.

Lets go back in time and look at another of our very early Python programs. Here is a simple Python function to convert a Fahrenheit temperature to Celsius.

```
def main():
    fahr = input("Enter the temperature in F: ")
    cel = (fahr - 32) * 5.0/9.0
    print "the temperature in C is: ", cel
```

Next, lets look at the Java Equivalent.

```
1  import java.util.Scanner;
2
3  public class TempConv {
4      public static void main(String[] args) {
5          double fahr;
6          double cel;
7          Scanner in;
8
9          in = new Scanner(System.in);
10         System.out.println("Enter the temperature in F: ");
11         fahr = in.nextDouble();
12
```

```
13          cel = (fahr - 32) * 5.0/9.0;
14          System.out.println("The temperature in C is: " + cel);
15
16          System.exit(0);
17      }
18
19  }
```

There are several new concepts introduced in this example. We will look at them in the following order:

- Import
- Variable Declaration
- Input/Output and the Scanner Class

4.1.1 Import

In Java you can use any class that is available without having to import the class subject to two very important conditions:

1. The javac and java must know that the class exists.
2. You must use the full name of the class

You first question might be how do the `java` and `javac` commands know that certain classes exist. The answer is the following:

1. Java knows about all the classes that are defined in .java and .class files in your current working directory.
2. Java knows about all the classes that are shipped with java.
3. Java knows about all the classes that are included in your `CLASSPATH` environment variable. Your `CLASSPATH` environment variable can name two kinds of structures.
   (a) A jar file that contains java classes
   (b) Another unix directory that contains java class files

You can think of the import statement in Java as working a little bit like the `from module import xxx` statement in Python. However, behind the scenes the two statements actually do very different things. The first important difference to understand is that the class naming system in Java is very hierarchical. The *full* name of the Scanner class is really `java.util.Scanner`. You can think of this name as having two parts: The first part `java.util` is called the *package* and the last part is the class. We'll talk more about the class naming system a bit later. The second important difference is that it is the Java class loader's responsibility to load classes into memory, not the import statement's.

So, what exactly does the import statement do? What it does is tell the compiler that we are going to use a shortened version of the class's name. In this example we are going to use the class `java.util.Scanner` but we can refer to it as just `Scanner`. We could use the `java.util.Scanner` class without any problem and without any import statement provided that we always referred to it by its full name. As an Experiment you may want to try this yourself. Remove the import statement and change the string Scanner to `java.util.Scanner` in the rest of the code. The program should still compile and run.

4.1.2 Declaring Variables

Here is where we run into one of the most important differences between Java and Python. Python is a *dynamically typed* language. In a dynamically typed language a variable can refer to any kind of object at any time. When the variable is used, the interpreter figures out what kind of object it is. Java is a *statically typed* language. In a statically

typed language the association between a variable and the type of object the variable can refer to is determined when the variable is *declared*. Once the declaration is made it is an error for a variable to refer to an object of any other type.

In the example above, lines 5—7 contain variable declarations. Specifically we are saying that `fahr` and `cel` are going to variables holding values of type `double`. The variable `in` will reference a Scanner object. This means that if we were to try an assignment like `fahr = "xyz"` the compiler would generate an error because `"xyz"` is a string and `fahr` is supposed to be a double.

For Python programmers the following error is likely to be even more common. Suppose we forgot the declaration for `cel` and instead left line 6 blank. What would happen when we type `javac TempConv.java` on the command line?

```
TempConv.java:13: cannot find symbol
symbol  : variable cel
location: class TempConv
         cel = (fahr - 32) * 5.0/9.0;
         ^
TempConv.java:14: cannot find symbol
symbol  : variable cel
location: class TempConv
         System.out.println("The temperature in C is: " + cel);
                                                          ^
2 errors
```

When you see the first kind of error, where the symbol is on the left side of the equals sign it usually means that you have not declared the variable. If you have ever tried to use a Python variable that you have not initialized the second error message will be familiar to you. The difference here is that we see the message before we ever try to test our program. More common error messages are discussed in the section *Common Mistakes*.

The general rule in Java is that you must decide what kind of an object your variable is going to reference and then you must declare that variable before you use it. There is much more to say about the static typing of Java but for now this is enough.

4.1.3 Input / Output / Scanner

In the previous section you saw that we created a `Scanner` object. In Java Scanner objects make getting input from the user, a file, or even over the network relatively easy. In our case we simply want to ask the user to type in a number at the command line, so in line 9 we construct a Scanner by calling the constructor and passing it the `System.in` object. Notice that this Scanner object is assigned to the name `in`, which we declared to be a `Scanner` on line 7. `System.in` is similar to `System.out` except of course it is used for input. If you are wondering why we must create a Scanner to read data from `System.in` when we can write data directly to `System.out` using `println`, you are not alone. We will talk about the reasons why this is so later when we talk in depth about Java streams. You will also see in other examples that we can create a Scanner by passing the Scanner a File object. You can think of a scanner as a kind of "adapter" that makes low level objects easier to use.

On line 11 we use the Scanner object to read in a number. Here again we see the implications of Java being a strongly typed language. Notice that we must call the method `nextDouble`. Because the variable `fahr` was declared as a double. So, we must have a function that is guaranteed to return each kind of object we might want to read. In this case we need to read a `double` so we call the function `nextDouble`. The compiler matches up these assignment statments and if you try to assign the results of a method call to the wrong kind of variable it will be flagged as an error.

Table 2 shows you some commonly used methods of the scanner class. There are many more methods supported by this class and we will talk about how to find them in the next chapter.

Return type	Method name	Description
boolean	hasNext()	returns true if more data is present
boolean	hasNextInt()	returns true if the next thing to read is an integer
boolean	hasNextFloat()	returns true if the next thing to read is a float
boolean	hasNextDouble()	returns true if the next thing to read is a double
int	nextInt()	returns the next thing to read as an integer
Float	nextFloat()	returns the next thing to read as a float
double	nextDouble()	returns the next thing to read as a double
String	next()	returns the next thing to read as a String

Of course Java is more well known for producing applications that have more of a user interface to them than reading and writing from the command line. Lets look at a version of our temperature control application that uses dialog boxes for input and output.

```
1  import javax.swing.*;
2
3  public class TempConvGUI {
4
5      public static void main(String[] args) {
6          String fahrString;
7          double fahr, cel;
8
9          fahrString = JOptionPane.showInputDialog("Enter the temperature in F");
10         fahr = Double.parseDouble(fahrString);
11         cel = (fahr - 32) * 5.0/9.0;
12
13         JOptionPane.showMessageDialog(null,"The temperature in C is, " + cel);
14     }
15
16 }
```

This example illustrates a couple of interesting points:

First, the function call `JOptionPane.showInputDialog` pops up a dialog box to allow you to enter a temperature. But, since you could enter anything into the text input box it returns a `String`. On the next line the string is converted into a `double` by the function `Double.parseDouble`. This is similar to what happens in Python when you call `float()` with either a string or an integer as the argument.

The next dialog box is `JOptionPane.showMessageDialog`. Notice that the first parameter is `null` In Java `null` serves the same purpose as `None` in Python. The first parameter is null because we do not have a 'main window' for this little application. When we look at creating full blown java programs with user interfaces, we will learn more about this parameter.

The second parameter is `"The temperature in C is, " + cel`. Now you may be thinking to yourself that this must surely be a violation of the strong typing I have been describing to you. After all you should not be able to add together a string and a double right? You are correct, however, all java objects have a method called `tostring`. The `tostring` method acts much like the Python method `__str__()` and is called automatically by the compiler whenever it makes sense to convert a Java object to a string.

4.2 String

Strings in Java and Python are quite similar. Like Python, Java strings are immutable. However, manipulating strings in Java is not quite as obvious since Strings do not support an indexing or slicing operator. That is not to say that you can't index into a Java string, you can. You can also pull out a substring just as you can with slicing. The difference is that Java uses method calls where Python uses Operators.

In fact this is the first example of another big difference between Java and Python. Java does not support any operator overloading. Table 3 maps common Python string operations to their Java counterparts. For the examples shown in the table we will use a string variable called "str"

Python	Java	Description
`str[3]`	`str.charAt(3)`	Return character in 3rd position
`str[2:5]`	`str.substring(2,4)`	Return substring from 2nd to 4th
`len(str)`	`str.length()`	Return the length of the string
`str.find('x')`	`str.indexOf('x')`	Find the first occurrence of x
`str.split()`	`str.split('\s')`	Split the string on whitespace into a list/array of strings
`str.split(',')`	`str.split(',')`	Split the string at `','` into a list/array of strings
`str + str`	`str.concat(str)`	Concatenate two strings together
`str.strip()`	`str.trim()`	Remove any whitespace at the beginning or end

4.3 List

Lets look at another early Python program. We are going to read numbers from a file and produce a histogram that shows the frequency of the various numbers. The data file we will use has one number between 0 and 9 on each line of the file. Here is a simple Python program that creates and prints a histogram.

```
def main():
    count = [0]*10
    data = open('test.dat')

    for line in data:
        count[int(line)] = count[int(line)] + 1

    idx = 0
    for num in count:
        print idx, " occured ", num, " times."
        idx += 1
```

Now if we run this program on a data file that looks like this:

```
9 8 4 5 3 5 2 1 5
```

We will get output that looks like this:

```
0 occurred 0 times
1 occurred 1 times
2 occurred 1 times
3 occurred 1 times
4 occurred 1 times
5 occurred 3 times
6 occurred 0 times
7 occurred 0 times
8 occurred 1 times
9 occurred 1 times
```

Lets review what is happening in this little program. In the first line we create a list and initialize the first 10 positions in the list to be 0. Next we open the data file called 'test.dat' Third, we have a loop that reads each line of the file. As we read each line we convert it to an integer and increment the counter at the position in the list indicated by the number on the line we just read. Finally we iterate over each element in the list printing out both the position in the list and the total value stored in that position.

To write the Java version of this program we will have to introduce several new Java concepts. First, you will see the Java equivalent of a list, called an `ArrayLlist`. Next you will see three different kinds of loops used in Java. Two

of the loops we will use are going to be very familiar, the third one is different from what you are used to in Python but is easy when you understand the syntax:

while Used with boolean expression for loop exit condition.

for Used to iterate over a sequence. This is very similar to `for i in xxx` where xxx is a list or string or file.

for Used to iterate through a sequence of numbers. This is most similar to for `i in range()`, except the syntax is different.

Here is the Java code needed to write the exact same program:

```
import java.util.Scanner;
import java.util.ArrayList;
import java.io.File;
import java.io.IOException;

public class Histo {

    public static void main(String[] args) {
        Scanner data = null;
        ArrayList<Integer> count;
        int idx;

        try {
                data = new Scanner(new File("test.dat"));
        }
        catch ( IOException e) {
            System.out.println("Sorry but I was unable to open your data file");
            e.printStackTrace();
            System.exit(0);
        }

        count = new ArrayList<Integer>(10);
        for (int i =0; i<10;i++) {
            count.add(i,0);
        }

        while(data.hasNextInt()) {
            idx = data.nextInt();
            count.set(idx,count.get(idx)+1);
        }

        idx = 0;
        for(int i : count) {
            System.out.println(idx + " occured " + i + " times.");
            idx++;
        }
    }
}
```

Before going any further, I suggest you try to compile the above program and run it on some test data that you create.

Now, lets look at what is happening in the Java source. As usual we declare the variables we are going to use at the beginning of the method. In this example we are declaring a Scanner variable called data, an integer called idx and an `ArrayList` called count. However, there is a new twist to the `ArrayList` declaration. Unlike Python where lists can contain just about anything, in Java we let the compiler know what kind of objects our array list is going to contain. In this case the `ArrayList` will contain Integers. The syntax we use to declare what kind of object the list will contain is the <``*Type*>`` syntax.

Technically, you don't *have* to declare what is going to be on an array list. The compiler will allow you to leave the

<``*Type*>`` off the declaration. If you don't tell Java what kind of object is going to be on the list Java will give you a warning message like this:

```
Note: Histo.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Without the <Integer> part of the declaration Python simply assumes that *any* object can be on the list. However, without resorting to an ugly notation called casting, you cannot do anything with the objects on a list like this! So, if you forget you will surely see more errors later in your code. (Try it and see what you get)

Lines 13—20 are required to open the file. Why so many lines to open a file in Java? The additional code mainly comes form the fact that Java forces you to reckon with the possibility that the file you want to open is not going to be there. If you attempt to open a file that is not there you will get an error. A try/catch construct allows us to try things that are risky, and gracefully recover from an error if one occurs. The following example shows the general structure of a try catch block.

```
try {
   Put some risky code in here.... like opening a file
}
catch (Exception e) {
   If an error happens in the try block an exception is thrown.
   We will catch that exception here!
}
```

Notice that in line 16 we are catching an `IOException`. In fact we will see later that we can have multiple catch blocks to catch different types of exceptions. If we want to be lazy and catch any old exception we can catch an `Exception` which is the parent of all exceptions.

On line 22 we create our array list and give it an initial size of 10. Strictly speaking it is not necessary to give the `ArrayList` any size. It will grow or shrink dynamically as needed just like a list in Python. On line 23 we start the first of three loops. The for loop on lines 23–25 serves the same purpose as the Python statement `count = [0]*10`, that is it initializes the first 10 positions in the `ArrayList` to hold the value 0.

The syntax of this for loop probably looks very strange to you, but in fact it is not too different from what happens in Python using range. In fact `for(int i = 0; i < 10; i++)` is exactly equivalent to the Python `for i in range(10)` The first statement inside the parenthesis declares and initializes a loop variable i. The second statement is a Boolean expression that is our exit condition. In other words we will keep looping as long as this expression evaluates to true. The third clause is used to increment the value of the loop variable at the end of iteration through the loop. In fact `i++` is Java shorthand for `i = i +` Java also supports the shorthand `i--` to decrement the value of i. Like Python you can also write `i += 2` as shorthand for `i = i + 2` Try to rewrite the following Python for loops as Java for loops:

- `for i in range(2,101,2)`
- `for i in range(1,100)`
- `for i in range(100,0,-1)`
- `for x,y in zip(range(10),range(0,20,2))` [hint, you can separate statements in the same clause with a ,]

The next loop (lines 27–30) shows a typical Java pattern for reading data from a file. Java while loops and Python while loops are identical in their logic. In this case we will continue to process the body of the loop as long as `data.hasNextInt()` returns true.

Line 29 illustrates another important difference between Python and Java. Notice that in Java we can not write `count[idx] = count[idx] + 1`. This is because in Java there is no overloading of operators. Everything except the most basic math and logical operations is done using methods. So, to set the value of an `ArrayList` element we use the `set` method. The first parameter of `set` indicates the index or position in the `ArrayList` we

are going to change. The next parameter is the value we want to set. Notice that once again we cannot use the indexing square bracket operator to retrieve a value from the list, but we must use the `get` method.

The last loop in this example is similar to the Python for loop where the object of the loop is a Sequence. In Java we can use this kind of for loop over all kinds of sequences, which are called Collection classes in Java. The for loop on line 33 `for(int i : count)` is equivalent to the Python loop `for i in count:` This loop iterates over all of the elements in the ArrayList called count. Each time through the loop the int variable i is bound to the next element of the `ArrayList`. If you tried the experiment of removing the `<Integer>` part of the `ArrayList` declaration you probably noticed that you had an error on this line. Why?

4.4 Arrays

As I said at the outset of this Section we are going to use Java `ArrayLists` because they are easier to use and more closely match the way that Python lists behave. However, if you look at Java code on the internet or even in your Core Java books you are going to see examples of something called arrays. In fact you have already seen one example of an array declared in the 'Hello World' program. Lets rewrite this program to use primitive arrays rather than array lists.

```
1  import java.util.Scanner;
2  import java.io.File;
3  import java.io.IOException;
4
5  public class HistoArray {
6      public static void main(String[] args) {
7          Scanner data = null;
8          int[] count = {0,0,0,0,0,0,0,0,0,0};
9          int idx;
10
11
12
13         try {
14                 data = new Scanner(new File("test.dat"));
15         }
16         catch ( IOException e) {
17             System.out.println("Sorry but I was unable to open your data file");
18             e.printStackTrace();
19             System.exit(0);
20         }
21
22         while(data.hasNextInt()) {
23             idx = data.nextInt();
24             count[idx] = count[idx] + 1;
25         }
26
27         idx = 0;
28         for(int i : count) {
29             System.out.println(idx + " occured " + i + " times.");
30             idx++;
31         }
32     }
33 }
```

The main difference between this example and the previous example is that we declare count to be an Array of integers. We also can initialize short arrays directly using the syntax shown on line 8. Then notice that on line 24 we can use the square bracket notation to index into an array.

4.5 Dictionary

Just as Python provides the dictionary when we want to have easy access to key, value pairs, Java also provides us a similar mechanism. Rather than the dictionary terminology, Java calls these objects Maps. Java provides two different implementations of a map, one is called the `TreeMap` and the other is called a `HashMap`. As you might guess the `TreeMap` uses a balanced binary tree behind the scenes, and the `HashMap` uses a hash table.

Lets stay with a simple frequency counting example, only this time we will count the frequency of words in a document. A simple Python program for this job could look like this:

```
def main():
    data = open('alice30.txt')
    wordList = data.read().split()
    count = {}
    for w in wordList:
        w = w.lower()
        count[w] = count.setdefault(w,0) + 1

    keyList = count.keys()
    keyList.sort()
    for k in keyList:
        print "%-20s occured %4d times"%(k, count[k])

main()
```

Notice that the structure of the program is very similar to the numeric histogram program.

```
import java.util.Scanner;
import java.util.ArrayList;
import java.io.File;
import java.io.IOException;
import java.util.TreeMap;

public class HistoMap {

  public static void main(String[] args) {
    Scanner data = null;
    TreeMap<String,Integer> count;
    int idx;
    String word;
    int wordCount;

    try {
      data = new Scanner(new File("alice30.txt"));
    }
    catch ( IOException e) {
      System.out.println("Sorry but I was unable to open your data file");
      e.printStackTrace();
      System.exit(0);
    }

    count = new TreeMap<String,Integer>();

    while(data.hasNext()) {
      word = data.next().toLowerCase();
      wordCount = count.get(word);
      if (wordCount == null) {
        wordCount = 0;
```

```
32          }
33          count.put(word,++wordCount);
34        }
35
36        for(String i : count.keySet()) {
37          System.out.printf("%-20s occured %5d times\n", i, count.get(i) );
38        }
39      }
40  }
```

CHAPTER

FIVE

CLASSES

5.1 Defining Classes in Java

You have already seen how to define classes in Java. Its unavoidable for even the simplest of programs. In this section we will look at how we define classes to create our own data types. Lets start by creating a fraction class to extend the set of numeric data types provided by our language. The requirements for this new data type are as follows:

- Given a numerator and a denominator create a new Fraction.
- When a fraction is printed it should be simplified.
- Two fractions can be added or subtracted
- Two fractions can be multiplied or divided
- Two fractions can be compared
- A fraction and an integer can be added together.
- Given a list of Fractions that list should be sortable by the default sorting function.

Here is a mostly complete implementation of a Fraction class in Python that we will refer to throughout this section:

```
class Fraction:

    def __init__(self,top,bottom):

        self.num = top         #the numerator is on top
        self.den = bottom      #the denominator is on the bottom


    def __repr__(self):
        if self.num > self.den:
            retWhole = self.num / self.den
            retNum = self.num - (retWhole * self.den)
            return str(retWhole) + " " + str(retNum)+"/"+str(self.den)
        else:
            return str(self.num)+"/"+str(self.den)

    def show(self):
        print self.num,"/",self.den

    def __add__(self,other):
        # convert to a fraction
        other = self.toFract(other)

        newnum = self.num*other.den + self.den*other.num
```

```
25          newden = self.den * other.den
26
27          common = gcd(newnum,newden)
28
29          return Fraction(newnum/common,newden/common)
30
31      def __radd__(self,leftNum):
32          other = self.toFract(leftNum)
33          newnum = self.num*other.den + self.den*other.num
34          newden = self.den * other.den
35
36          common = gcd(newnum,newden)
37
38          return Fraction(newnum/common,newden/common)
39
40      def __cmp__(self,other):
41
42          num1 = self.num*other.den
43          num2 = self.den*other.num
44
45          if num1 < num2:
46             return -1
47          else:
48             if num1 == num2:
49                return 0
50             else:
51                return 1
52
53      def toFract(self,n):
54          if isinstance(n,int):
55              other = Fraction(n,1)
56          elif isinstance(n, float):
57              wholePart = int(n)
58              fracPart = n - wholePart
59              # convert to 100ths???
60              fracNum = int(fracPart * 100)
61              newNum = wholePart * 100 + fracNum
62              other = Fraction(newNum,100)
63          elif isinstance(n,Fraction):
64              other = n
65          else:
66              print "Error: cannot add a fraction to a ", type(n)
67              return None
68          return other
69
70  #gcd is a helper function for Fraction
71
72  def gcd(m,n):
73      while m%n != 0:
74          oldm = m
75          oldn = n
76
77          m = oldn
78          n = oldm%oldn
79
80      return n
```

The instance variables (data members) we will need for our fraction class are the numerator and denominator. Of course in Python we can add instance variables to a class at any time by simply assigning a value to `objectReferenc.variableName` In Java all data members must be declared up front.

The declarations of instance variables can come at the beginning of the class definition or the end. Cay Horstman, Author of the *Core Java* books puts the declarations at the end of the class. I like them at the very beginning so you see the variables that are declared before you begin looking at the code that uses them. With that in mind the first part of the Fraction class definition is as follows:

```
public class Fraction {
    private int numerator;
    private int denominator;
```

Notice that we have declared the numerator and denominator to be private. This means that the compiler will generate an error if another method tries to write code like the following:

```
Fraction f = new Fraction(1,2);
int y = f.numerator * 10;
```

Direct access to private members is not allowed from outside the class. Therefore, if we do not include a way to update the numerator or denominator, then our fractions will be *immutable*. For private fields, we typically provide access to outsiders using *getter* methods.

```
1  public Integer getNumerator() {
2      return numerator;
3  }
4
5  public Integer getDenominator() {
6      return denominator;
7  }
```

If we want to allow the numerator and denominator to change, then we would have a *mutable* fraction. One way to do this would be to make the fields public. However, it is more common in Java to provide *setter* methods.

```
1  public void setNumerator(Integer numerator) {
2      this.numerator = numerator;
3  }
4
5  public void setDenominator(Integer denominator) {
6      this.denominator = denominator;
7  }
```

5.2 Writing a constructor

Once you have identified the instance variables for you class the next thing to consider is the constructor. In Java, constructors have the same name as the class and are declared public. They are declared without a return type. So any function that is named the same as the class and has no return type is a constructor. Our constructor will take two parameters the numerator and the denominator.

```
public Fraction(int top, int bottom) {
    num = top;
    den = bottom;
}
```

There are a couple of important things to notice here. First, you will notice that the constructor does not have a self parameter. You will also notice that we can simply refer to the instance variables by name without the self prefix, because they have already been declared. This allows the Java compiler to do the work of dereferencing the current

Java object. Java does provide a special variable called `this` that works like the self variable. In Java, `this` is typically only used when it is needed to differentiate between a parameter or local variable and an instance variable. For example this alternate definition of the the Fraction constructor uses `this` to differentiate between parameters and instance variables.

```
public Fraction(int num, int den) {
    this.num = num;
    this.den = den;
}
```

5.3 Methods or Member Functions

Now we come to one of the major differences between Java and Python. The Python class definition used the special methods for addition, and comparison that have the effect of redefining how the standard operators behave. In Java there is *no operator overloading*. So we will have to write member functions to do addition, subtraction, multiplication, and division. Lets begin with addition.

```
1 public Fraction add(Fraction other) {
2     int newNum, newDen, common;
3
4     newNum = other.denominator*this.numerator +
5                             this.denominator*other.numerator;
6     newDen = this.denominator * other.denominator;
7     common = gcd(newNum,newDen);
8     return new Fraction(newNum/common, newDen/common );
9 }
```

First you will notice that the add member function is declared as `public Fraction` The `public` part means that any other method may call the add method. The `Fraction` part means that `add` will return a fraction as its result.

Second, you will notice that on line two all of the local variables used in the function are declared. In this case there are three local variables: `newNum`, `newDen`, and `common`. It is a good idea for you to get in the habit of declaring your local variables at the beginning of your function. This declaration section provides a simple road map for the function in terms of the data that will be used. The listing above also makes use of the `this` variable, you may find it useful to use `this` until you are comfortable with abandoning your Pythonic ideas about `self`.

Declaring your variables at the top is not a requirement, it is just a recommended practice for you. Java only requires that you declare your variables before they are used. The following version of Fraction is also legal Java, but may be somewhat less readable.

```
1 public Fraction add(Fraction other) {
2     int newNum = other.denominator*numerator +
3                             denominator*other.numerator;
4     int newDen = denominator * other.denominator;
5     int common = gcd(newNum,newDen);
6     return new Fraction(newNum/common, newDen/common );
7 }
```

The addition takes place by multiplying each numerator by the opposite denominator before adding. This procedure ensures that we are adding two fractions with common denominators. Using this approach the denominator is computed by multiplying the two denominators. The greatest common divisor function is used to find a common divisor to simplify the numerator and denominator in the result.

Finally on line 8 a new fraction is returned as the result of the computation. The value that is returned by the return statement must match the value that is specified as part of the declaration. So, in this case the return value on line 8 must match the declared value on line 1.

5.3.1 Method Signatures and Overloading

Our specification for this project said that we need to be able to add a `Fraction` to an `int`. In Python we can do this by checking the type of the parameter using the `isinstance` function at runtime. Recall that `isinstance(1,int)` returns `True` to indicate that 1 is indeed an instance of the `int` class. See lines 22 and 53—68 of the Python version of the Fraction class to see how our Python implementation fulfills this requirement.

In Java we can do runtime type checking, but the compiler will not allow us to pass an `int` to the add function since the parameter has been declared to be a Fraction. The way that we solve this problem is by writing another `add` method with a different set of parameters. In Java this practice is legal and common we call this practice *overloading*.

This idea of overloading raises a very important difference between Python and Java. In Python a method is known by its name only. In Java a method is known by its signature. The signature of a method includes its name, and the types of all of its parameters. The name and the types of the parameters are enough information for the Java compiler to decide which method to call at runtime.

To solve the problem of adding an `int` and a `Fraction` in Java we will overload both the constructor and the add function. We will overload the constructor so that if it only receives a single `int` it will convert the `int` into a `Fraction`. We will also overload the add method so that if it receives an `int` as a parameter it first construct a `Fraction` from that integer and then add the two `Fractions` together. The new methods that accomplish this task are as follows:

```
1  public Fraction(int num) {
2      this.numerator = num;
3      this.denominator = 1;
4  }
5
6  public Fraction add(int other) {
7      return add(new Fraction(other));
8  }
```

Notice that the overloading approach can provide us with a certain elegance to our code. Rather than utilizing if statements to check the types of parameters we just overload functions ahead of time which allows us to call the method we want and allow the compiler to make the decisions for us. This way of thinking about programming takes some practice.

Our full Fraction class to this point would look like the following. You may want to try to compile and run the short test program provided just to see what happens.

```
1  public class Fraction {
2
3      private int numerator;
4      private int denominator;
5
6      public Fraction(int num, int den) {
7          this.numerator = num;
8          this.denominator = den;
9      }
10
11     public Fraction(int num) {
12         this.numerator = num;
13         this.denominator = 1;
14     }
15
16     public Fraction add(Fraction other) {
17         int newNum, newDen, common;
18
19         newNum = other.denominator*this.numerator + this.denominator*other.numerator;
20         newDen = this.denominator * other.denominator;
```

```
21          common = gcd(newNum,newDen);
22          return new Fraction(newNum/common, newDen/common );
23      }
24
25      public Fraction add(int other) {
26          return add(new Fraction(other));
27      }
28
29      private static int gcd(int m, int n) {
30          while (m % n != 0) {
31              int oldm = m;
32              int oldn = n;
33              m = oldn;
34              n = oldm%oldn;
35          }
36          return n;
37      }
38
39      public static void main(String[] args) {
40          Fraction f1 = new Fraction(1,2);
41          Fraction f2 = new Fraction(2,3);
42
43          System.out.println(f1.mul(f2));
44          System.out.println(f1.add(1));
45      }
46
47  }
```

5.4 Inheritance

If you ran the program above you probably noticed that the output is not very satisfying. Chances are your output looked something like this:

```
Fraction@7b11a3ac
Fraction@6c22c95b
```

The reason is that we have not yet provided a friendly string representation for our Fraction objects. The truth is that, just like in Python, whenever an object is printed by the `println` method it must be converted to string format. In Python you can control how that looks by writing an `__str__` method for your class. If you do not then you will get the default, which looked something like the above.

5.4.1 The `Object` Class

In Java, the equivalent of `__str__` is the `toString` method. Every object in Java already has a `toString` method defined for it because every class in Java automatically inherits from the Object class. The object class provides default implementations for the following functions (among others).

- equals
- getClass
- toString

To make our output nicer we will implement the `toString` method for the `Fraction` class. A simple version of the method is provided below.

```
public String toString() {
    return numerator + "/" + denominator;
}
```

The other important class for us to implement from the list of methods inherited from Object is the `equals` method. When two objects are compared in Java using the == operator they are tested to see if they are exactly the same object, that is do the two objects occupy the same exact space in the computers memory. This is the default behavior of the `equals` method provided by Object. The `equals` method allows us to decide if two objects are equal by looking at their instance variables. However it is important to remember that since Java does not have operator overloading if you want to use your equals method you must call it directly. Therefore once you write your own `equals` method:

```
object1 == object2
```

is NOT the same as

```
object1.equals(object2)
```

Here is the equals method we would like to write for the Fraction class:

```
public boolean equals(Fraction other) {
    int num1 = this.numerator * other.denominator;
    int num2 = this.denominator * other.numerator;
    return num1 == num2;
}
```

However, this is not quite correct, since `equals` must work for `null` and for objects of classes other than `Fraction`. The proper implementation of `equals` include a bit of boilerplate before the code above.

```
1 public boolean equals(Object that) {
2     if (that == null) return false;
3     if (this.getClass() != that.getClass()) return false;
4     Fraction other = (Fraction) that;
5
6     int num1 = this.numerator * other.denominator;
7     int num2 = this.denominator * other.numerator;
8     return num1 == num2;
9 }
```

One important thing to remember about `equals` is that it only checks to see if two objects are equal it does not have any notion of less than or greater than. We'll see more about that shortly.

5.4.2 Abstract Classes and Methods

If we want to make our Fraction class behave like Integer, Double, and the other numeric classes in Java, we need to make a couple of additional modifications to the class. The first thing we will do is plug `Fraction` into the Java class hierarchy at the same place as Integer and its siblings. If you look at the documentation for Integer you will see that Integer's parent class is `Number`. Number is an *abstract class* that specifies several methods that all of its children must implement. In Java an abstract class is more than just a placeholder for common functions. In Java an abstract class has the power to specify certain functions that all of its children *must* implement. You can trace this power back to the strong typing nature of Java.

The that makes the Fraction class a child of Number is as follows:

```
public class Fraction extends Number {
    ...
}
```

The keyword extends tells the compiler that the class `Fraction` extends, or adds new functionality to the `Number` class. A child class always extends its parent.

The methods we must implement if `Fraction` is going to be a child of `Number` are:

- `longValue`
- `intValue`
- `floatValue`
- `doubleValue`

This really isn't much work for us to implement these functions as all we have to do is some conversion of our own and some division. The implementation of these methods is as follows:

```
1  public double doubleValue() {
2      return ((double) numerator) / ((double) denominator);
3  }
4
5  public float floatValue() {
6      return ((float) numerator) / ((float) denominator);
7  }
8
9  public int intValue() {
10     return numerator / denominator;
11 }
12
13 public long longValue() {
14     return ((long) numerator) / ((long) denominator);
15 }
```

By having the `Fraction` class extend the `Number` class we can now pass a `Fraction` to any Java function that specifies it can receive a `Number` as one of its parameters. For example many Java user interface methods accept any object that is a subclass of `Number` as a parameter. In Java the class hierarchy and the IS-A relationships are very important. Whereas in Python you can pass any kind of object as a parameter to any function the strong typing of Java makes sure that you only pass an object as a parameter that is of the type specified in the function call or one of its children. So, in this case when you see a parameter of type `Number` its important to remember that an `Integer` *is-a* `Number` and a `Double` *is-a* `Number` and a `Fraction` *is-a* `Number`.

However, and this is a big however, it is also important to remember that if you specify `Number` as the type on a particular parameter then the Java compiler will only let you use the methods of a `Number`. In this case longValue, intValue, floatValue, and doubleValue.

Lets suppose you define a method in some class as follows:

```
public void test(Number a, Number b) {
    a.add(b);
}
```

The Java compiler would give an error because `add` is not a defined method of the `Number` class. Even if you called the add method and passed two `Fractions` as parameters.

5.5 Interfaces

Lets turn our attention to making a list of fractions sortable by the standard Java sorting method `Collections.sort`. In Python all we would need to do is implement the `__cmp__` method. But in Java we cannot be that informal. In Java Things that are sortable must be `Comparable`. Your first thought might be that

`Comparable` is Superclass of `Number`. That would be a good thought but it would not be correct. Java only supports single inheritance, that is, a class can have only one parent. Although it would be possible to add an additional Layer to the class hierarchy it would also complicate things dramatically. Because Not only are Numbers comparable, but Strings are also Comparable as would many other types. For example we might have a `Student` class and we want to be able to sort Students by their gpa. But `Student` already extends the class `Person` for which we have no natural comparison function.

Java's answer to this problem is the `Interface` mechanism. Interfaces are like a combination of Inheritance and contracts all rolled into one. An interface is a *specification* that says any object that claims it implements this interface must provide the following methods. It sounds a little bit like an abstract class, however it is outside the inheritance mechanism. You can never create an instance of `Comparable`. Many objects, however, do implement the `Comparable` interface. What does the Comparable interface specify?

The `Comparable` interface says that any object that claims to be `Comparable` must implement the `compareTo` method.

```
int compareTo(T o)
```

The following is the documentation for the `compareTo` method as specified by the Comparable interface.

> Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object. The implementor must ensure `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))` for all `x` and `y`. (This implies that `x.compareTo(y)` must throw an exception iff `y.compareTo(x)` throws an exception.)
>
> The implementor must also ensure that the relation is transitive: `(x.compareTo(y)>0 && y.compareTo(z)>0)` implies `x.compareTo(z)>0`.
>
> Finally, the implementor must ensure that `x.compareTo(y)==0` implies that `sgn(x.compareTo(z)) == sgn(y.compareTo(z))`, for all `z`.
>
> It is strongly recommended, but not strictly required that `(x.compareTo(y)==0) == (x.equals(y))`. Generally speaking, any class that implements the Comparable interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."
>
> In the foregoing description, the notation `sgn(expression)` designates the mathematical signum function, which is defined to return one of -1, 0, or 1 according to whether the value of expression is negative, zero or positive.

To make our `Fraction` class `Comparable` we must modify the class declaration line as follows:

```
public class Fraction extends Number implements Comparable<Fraction> {
    ...
}
```

The specification `Comparable<Fraction>` makes it clear that Fraction is only comparable with another Fraction. The `compareTo` method could be implemented as follows:

```
public int compareTo(Fraction other) {
    int num1 = this.numerator * other.denominator;
    int num2 = this.denominator * other.numerator;
    return num1 - num2;
}
```

Unlike the `equals` method, `compareTo` can assume that it's argument has type `Fraction`. In addition, `compareTo` need not return a value if `other` is `null`; in this case, the function will throw a `NullPointerException`.

5.6 Static member variables

Suppose that you wanted to write a Student class so that the class could keep track of the number of students it had created. Although you could do this with a global counter variable that is an ugly solution. The right way to do it is to use a static variable. In Python we could do this as follows:

```
class Student:
    numStudents = 0

    def __init__(self, id, name):
        self.id = id
        self.name = name

        Student.numStudents = Student.numStudents + 1

def main():
    for i in range(10):
        s = Student(i,"Student-"+str(i))
    print 'The number of students is: ', Student.numStudents

>>> main()
The number of students is:  10
>>>
```

In Java we would write this same example using a static declaration.

```
public class Student {
    public static int numStudents = 0;

        private int id;
        private String name;

        public Student(int id, String name) {
        this.id = id;
        this.name = name;

        numStudents = numStudents + 1;
    }

    public static void main(String[] args) {
        for(int i = 0; i < 10; i++) {
            Student s = new Student(i,"Student"+i.toString());
        }
        System.out.println("The number of students: "+Student.numStudents.toString());
    }
}
```

In this example notice that we create a static member variable by using the static modifier on the variable declaration. Once a variable has been declared static in Java it can be access from inside the class without prefixing the name of the class as we had to do in Python.

5.7 Static Methods

We have already discussed the most common static method of all, `main`. However in our Fraction class we also implemented a method to calculate the greatest common divisor for two fractions (`gdc`). There is no reason for this method to be a member method since it takes two `int` values as its parameters. Therefore we declare the method to

be a static method of the class. Furthermore since we are only going to use this `gcd` method for our own purposes we can make it private.

```
private static int gcd(int m, int n) {
    while (m % n != 0) {
        int oldm = m;
        int oldn = n;
        m = oldn;
        n = oldm%oldn;
    }
    return n;
}
```

5.8 Full Implementation of the Fraction Class

A final version of the Fraction class that exercises all of the features we have discussed is as follows.

```
import java.util.Arrays;

public class Fraction extends Number implements Comparable<Fraction> {
    private int numerator;
    private int denominator;

    public Fraction(int num, int den) {
        this.numerator = num;
        this.denominator = den;
    }
    public Fraction(int num) {
        this.numerator = num;
        this.denominator = 1;
    }

    public Integer getNumerator()                   { return numerator; }
    public Integer getDenominator()                 { return denominator; }
    public void setNumerator  (Integer numerator)   { this.numerator = numerator; }
    public void setDenominator(Integer denominator) { this.denominator = denominator; }

    public Fraction add(int other) {
        return add(new Fraction(other));
    }
    public Fraction add(Fraction other) {
        int newNum = other.denominator*this.numerator + this.denominator*other.numerator;
        int newDen = this.denominator * other.denominator;
        int common = gcd(newNum,newDen);
        return new Fraction(newNum/common, newDen/common );
    }
    private static int gcd(int m, int n) {
        while (m % n != 0) {
            int oldm = m;
            int oldn = n;
            m = oldn;
            n = oldm%oldn;
        }
        return n;
    }

    public double doubleValue() { return ((double) numerator) / ((double) denominator); }
```

```
41      public float  floatValue()  { return ((float) numerator) / ((float) denominator); }
42      public int    intValue()    { return numerator / denominator; }
43      public long   longValue()   { return ((long) numerator) / ((long) denominator); }
44      public String toString()    { return numerator + "/" + denominator; }
45      public boolean equals(Object that) {
46          if (that == null) return false;
47          if (this.getClass() != that.getClass()) return false;
48          Fraction other = (Fraction) that;
49
50          int num1 = this.numerator * other.denominator;
51          int num2 = this.denominator * other.numerator;
52          return num1 == num2;
53      }
54      public int compareTo(Fraction other) {
55          int num1 = this.numerator * other.denominator;
56          int num2 = this.denominator * other.numerator;
57          return num1 - num2;
58      }
59
60      // This is a unit test program to test the class.
61      public static void main(String[] args) {
62          Fraction f1 = new Fraction(1,2);
63          Fraction f2 = new Fraction(2,3);
64          Fraction f3 = new Fraction(1,4);
65
66          System.out.println(f1.add(1));
67          System.out.println(f1.intValue());
68          System.out.println(f1.doubleValue());
69
70          Fraction[] myFracs = { f1, f2, f3 };
71          Arrays.sort(myFracs);
72
73          for(Fraction f : myFracs) {
74              System.out.println(f);
75          }
76      }
77  }
```