Programming in Java for Ecology and Evolutionary Biology

Written for a beginning programming course for biology graduate students
at the University of Arizona (ECOL 597a), Spring 1998
Wayne Maddison[1]

Overview of Java

Java is a programming language that received most of its fame for its use on the Internet, but it is in fact a general language that can be used for programs that are written to reside and run on a single computer. Our goal in this course is to learn how to program in Java for use in basic calculations in ecology and evolutionary biology.

Java is an object-oriented programming language. In fact, it is almost pure in its object-oriented approach, in that all of the program's instructions must exist within classes (which are the blueprints for objects). Objects are like organisms that are born, have certain properties and can perform certain tasks, and eventually are disposed. Classes of objects are like clades of organisms, in that you can define a new class as a modification of an existing classes. The new class inherits all the functionality of the class on which it is based, but can have new or modified behavior (thus, descent with modification). All this may be confusing, but it turns out that this approach allows you to program certain things much more easily than with other languages.

However, we will not begin by focusing on objects and how to make them. Rather, we will focus on the processes that occur within objects. Within an object, you can do a lot of programming without worrying about how the object is created and used. It is a little bit like studying physiology instead of population biology. We will begin studying physiology, and later consider the population biology of objects. One of the reasons we are doing this is that many of you may find that, for your projects, you don't really need to do much with objects.

In fact, we will only barely scratch the surface of Java, because the course is intended as an introduction to programming. (If you already know C or C++, you may find parts of this disappointingly tedious or incomplete. Even some basic concepts like subclasses and inheritance will be covered only

---

[1] DISCLAIMER: I'm not a computer scientist, and the last programming course I had was FORTRAN, in high school. I have had a fair bit of programming experience, especially via MacClade (http://phylogeny.arizona.edu/macclade/macclade.html) and Mesquite (http://mesquite.biosci.arizona.edu/mesquite/mesquite.html). Use this guide at your own risk.

©W. Maddison 1998-2000

briefly.) Java extends from basic calculations with numbers, to features that draw and manipulate windows and buttons, to features that retrieve information over the Internet, and so on. Some of you might need or want to go further than others, and so we'll try to be flexible on how much is covered for each of you.

©W. Maddison 1998-2000

Chapter 1. Introduction to programming in Java: Variables and operators

Objectives

After this chapter, you will be a real live programmer. You will learn about
•the concepts of variables as containers for values, and operators as manipulators of values
•The primitive types, including the whole-number (integer) and the decimal-number (floating point, real) types.
•basic arithmetic operations of addition, subtraction, multiplication, and division.

A simple program

For the most part, you can think of a program as a series of instructions that tells the machine to add, subtract, multiply, divide and perform other operations on numbers (and other logical structures) that are stored in containers. Here is a program written in English[2]:

> Get a cup and call it "Fred's Cup".
> Put five jelly beans into Fred'sCup.
> Get a cup and call it "Katie's Cup".
> Put four jelly beans into Katie's Cup.
> Get a cup and call it "My Cup".
> Add the jelly beans in Fred's Cup to the jelly beans in Katie's Cup, and put them all into My Cup.
> Say how many beans are in My Cup.

The same program looks like this in Java:

```
int fredsCup;
fredsCup = 5;
int katiesCup;
katiesCup = 4;
int myCup;
myCup = fredsCup + katiesCup;
System.out.println("My cup has " + myCup);
```

Names and comments

Two quick notes. First, words in Java are **case-sensitive**. This means that fredsCup is not the same as FredsCup which is not the same as Fredscup and so on. Java interprets each of these as a different word. Therefore, be careful with the shift key. To help you remember capitalization, stick to consistent rules. It is convention in Java that variable names start with lower case

[2]I apologize that this English program is grammatically incorrect. I (and some other programmers I know), when writing in English, sometimes refuse to follow the completely illogical tradition, considered a grammatical rule by some, of putting the period inside the quotation marks. If you understand why we violate this rule, then you will understand a bit about the logical nesting of expressions in programming.

©W. Maddison 1998-2000

(hence the use of fredsCup instead of FredsCup). If the name is a compound of several English words, words after the first usually have their first letter in upper case for ease of identifying the start of the word (hence fredsCup instead of fredscup).

Second, in programming you often want to write notes to yourself that the computer won't see, sort of like doodling in the margins. You can do this in Java using **comments**, of which there are two sorts. If you put two slashes in a row (//), everything after them on that line will be ignored. If you put a slash and a star (/*), everything after them, up until the next star then slash (*/) will be ignored, even if it is several lines later. Comments are useful both for writing notes to yourself as well as for temporarily "deleting" lines of code you have written.

Variables

The computer doesn't go to the cupboard to get jellybeans and cups of course, so the program uses numbers instead of jellybeans, and variables instead of cups. You can think of a **variable** as a place within the memory of the computer that serves as a container for a number. Let us look at the program line by line.

```
int fredsCup;
```

This line says: "make a variable that will hold an integer number, and call the variable 'fredsCup'". This line, which asks for the variable to be made, is said to be a **declaration** of the variable. (Actually, it really says "reserve a place in memory that is big enough for an integer number, and refer to this place by the name 'fredsCup'". It is easier, I think, to imagine a variable to be like a container that has the name fredsCup, without worrying about some physical location in the computer's memory.)

The line:

```
fredsCup = 5;
```

says "place the value 5 into the variable fredsCup". This is your basic **assignment statement**, perhaps the most important statement in programming. Different programming languages do it differently:

```
fredsCup <- 5 (APL)
fredsCup = 5; (C, C++, Java)
fredsCup := 5; (Pascal)
put 5 into fredsCup (Hypertalk)
```

My personal favorite of these is the APL style[3]; it's very clear you are placing 5 into fredsCup. My least favorite is that of C or Java, because without knowing the language you might think that this statement is making a claim that fredsCup is 5, which it might or might not be (for instance, "is fredsCup = 5?), instead of actually putting 5 into fredsCup. But, hey, we use

---

[3]At least, if I am remembering it correctly. I was in high school the last time I remember seeing APL.

©W. Maddison 1998-2000

English, and I've already complained about one grammatical rule. We can live with a few annoyances in an otherwise cool language like Java[4].

The line

```
myCup = fredsCup + katiesCup;
```

says "add the contents of fredsCup to the contents of katiesCup and put the result into myCup".

Output and Strings

The line

```
System.out.println("My cup has " + myCup);
```

should be more or less self-explanatory, correct? The only issue might be the `System.out.println` business, which is a request to output some text to the Java output window. Within the parentheses you put some text, called a "String", and this `String` will be written into the output window.

Strings are special objects that you will use a lot for input and output. You can concatenate two Strings using "+". Thus,

```
"Num" + "ber"
```

yields "Number". So our little program involves not just numbers, but also Strings.

Remember that the String "64" is not the same as the number 64, any more than your signature, your photograph and you are the same thing[5]. However, it turns out that you can write either "My lucky number is " + "7" or "My lucky number is " + 7. In the former, two Strings are concatenated; in the latter, it looks as if a number is being concatenated onto a String. In fact, when you try to do this, Java automatically converts the number to its String representation before the concatenation[6]. That's what happens in the little program with "My cup has " + myCup. [7]

What do you predict will be the text output by the program?

---

[4]Java's syntax is based on that of C and C++, primarily so that C and C++ programmers, of whom there are many, feel at home with Java. C and C++ are languages loved by those who prefer to write the most dense, uninterpretable source code , thereby with great pride. C has been called a "write-only language". Java suffers a bit from its heritage, but we can get over it.

[5]Make sure you understand this. Ask if confused.

[6] You could have written `System.out.println(myCup)`, since `System.out.println` would have automatically converted the number to a string.

[7] Beware: `System.out.println(5 + 2 + " is the answer")` yields "7 is the answer", `System.out.println("The answer is " + 5 + 2)` yields "The answer is 52", and `System.out.println("The answer is " + (5 + 2))` yields "The answer is 7". In the first case the computer initally assumes integer addition is being done, and only switches to String concatenation when it hits the String. In the second case the 5 is concatenated to the String, then 2 is concatenated. In the third case, the parentheses force the numbers to be dealt with first, on their own.

©W. Maddison 1998-2000

Note, by the way, that Java uses semicolons, not periods, to end its statements; this makes some sense, because you can think of them as being like complete ideas; in English, complete ideas placed in a single sentence are separated by semicolons; you can think of the entire program as being like the sentence; however, in Java, the program doesn't end in a period.

Making the program run

It turns out the "program" I showed you won't really run just as shown above. First of all, in Java, you need to surround the code[8] by some stuff to get the program off the ground and running, as follows:

```
public class MyProgram {
     public static void main(String args[]) {
          int fredsCup;
          fredsCup = 5;
          int katiesCup;
          katiesCup = 4;
          int myCup;
          myCup = fredsCup + katiesCup;
          System.out.println("My cup has " + myCup);
     }

}
```

We will explain just what all this means later. For now, you are going to make the program run.

Sadly, you can't simply type the above program into a window and say "computer, run program" (actually, you can nowadays, but not with the technology we have in our course). The program needs to be converted into the form that will run on your computer. If you were to look inside a copy of Microsoft Word, for instance, you wouldn't find code that looks like Pascal or C or Java. You'd see mostly gobbledygook, which are very concise and simple instructions to the microprocesser in your computer. This is the "object code" or "machine language", which the computer can understand, as opposed to "source code", written in Java or Pascal or C, which you can understand. You need to use a special program that translates your source code into object code. This program is called a **compiler**. Since the object code a Macintosh can understand is different from that a Windows machine can understand, you need to compile your source code into different object code for Mac and Windows. That's why you can't copy a program from a Windows machine and expect it to run on a Mac.

---

[8]the word "code" means, more or less, the programming instructions written in a programming language. "Code" is to a program what "text" is to a novel. Programmers probably say "code" to sound mysterious. To write code that will be respected, you have to make it difficult and mysterious (even to yourself).

©W. Maddison 1998-2000

Actually, what I have said doesn't quite apply in the case of Java. A Java compiler doesn't quite compile the source code to object code for a Macintosh, or Windows PC, or any other particular sort of computer. Instead, it compiles the source code to a special sort of object code that will run on a **Java Virtual Machine**. The Java Virtual Machine is a program that runs on your computer. It pretends to be a special Java computer (neither Mac nor Windows nor...). Your compiled Java object code runs happily on it, thinking that it's running on a Java computer, when in fact it's only running on a Virtual Machine (VM) program that is pretending to be a Java computer. Every time the Java program feeds an instruction to the VM, the VM quickly translates the Java "object" code to the object code for the particular computer the VM is running on (e.g., a Mac) which executes the instruction[9]. That way the instruction actually gets carried out on your computer, and the VM successfully maintains its ruse. Your exact same compiled Java code will run under a Java Virtual Machine on a Mac, or on a PC, because these different Virtual Machines present the exact same smiling Java VM face to the Java code, even if behind closed doors the one VM translates to Mac code and the other translates to PC code.

So, one advantage of Java is that the exact same program can be copied from one computer to another and run more or less the same. One disadvantage is that the Java VM is constantly translating instructions from Java byte code to the object code of the computer it's running on. This constant translation slows down your program (it's exactly like speaking through an interpreter). Java programs run more slowly than programs compiled with typical languages like C. In case this makes you depressed, (1) computers are awful fast nowadays, and (2) there are other advantages to Java that we will deal with later, including the possibility you will finish writing your program much faster. As a programmer, would you rather save your time, or the computer's time?

Find the folder "template project"[10]. Copy the folder. Within the copy, find and open up the file "template.prj". This will start up the compiler program, Metrowerks Codewarrior.
The project window will open up, and you will see the source code file "source.java" listed. Double click on it to open up the file. You should see the following:

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

[9] This isn't quite true — because of JIT compilers, not every instruction needs to be translated every time the program comes to it. But that's a detail you don't need to worry about.
[10] In the course for which this guide was written, we used Metrowerks Codewarrior as the compiler.

©W. Maddison 1998-2000

Replace the line "`System.out.println("Hello World");`" so as to obtain our little program. Run it to see what it will say.

©W. Maddison 1998-2000

Operators

In the little example program we have discussed, there are various points at which manipulations or comparisons of numbers are made, and there is an implicit resulting value. The most obvious might be addition:

```
fredsCup + katiesCup
```

This is an **expression**, and its value is the sum of the contents of the two variables. The "+" symbol in this expression is an **operator** — it operates on the two numbers on either side of it to yield their sum as its result.

It turns out there are other operators as well in the little program. The "+" symbol in the output is a String concatenation operator, as we have already seen. Even the "=" in the assignment statement is considered an operator, but we won't worry about that now[11].

It probably does not surprise you that "+" is used for addition. Thus,

```
myCup = fredsCup + katiesCup;
```

says "add fredsCup to katiesCup and put the result in myCup". The symbols for subtraction, multiplication, and division are "-", "*", and "/", as follows:

```
myCup = fredsCup - katiesCup;
myCup = fredsCup * katiesCup;
myCup = fredsCup / katiesCup;
```

Rerun the program modified by changing the "+" to "-" to check the output. Then, rerun it with "*", then with "/". You will notice that the result from 5 / 4 is not 1.25, but rather 1. It is important that you remember that this will happen. Because myCup, fredsCup and katiesCup are all variables of type "int", they can contain only whole numbers. Thus, the results of division get truncated to the next lowest whole number. 4/4, 5/4, 6/4 and 7/4 will all give the same result: 1.

Types

In our little example program, the variables were indicated to be for containing integer numbers by the "`int`" in front of their names where they first appear in the program. Integer numbers are whole numbers, namely ... -3, -2, -1, 0, 1, 2, 3, ... Java gives four different types of integer variables that you can use. These are **`byte`**, **`short`**, **`int`**, and **`long`**. Generally, we will use "`int`", but you should be aware of the differences. If you declare a variable to be of type "`byte`", it will take up only 8 bits of space, which is great if you are space conscious, but it can take values only between -128 and

[11]Well, if you insist. After performing its obvious function, to assign a value to the variable on the left-hand side, an assignment statement takes on a value (the value assigned). Thus, you can do wierd things like "z = ((x = 1) + (y = 2));", which assigns 1 to x and 2 to y. Since "(x=1)" takes on the value 1, and "(y=2)" the value 2, their sum is 3, which is then put into z.

©W. Maddison 1998-2000

+127. Variables of type `short` take up 16 bits, `int` 32 bits, and `long` 64 bits, each with a respectively greater range of integers that can be represented[12].

Computer programs need to do more than manipulate whole numbers, and so other basic sort of variable types are supplied in Java. Most obviously, fractional or decimal numbers are needed. Often called real or floating-point numbers in programming languages, these can be stored in Java in variables of types **`float`** (32 bits) or **`double`** (64 bits).

Finally, variables of the **`boolean`** type can take on the values `true` or `false`.

> Rerun the program modified so that each of the variables is declared to be a **`double`** instead of `int`. Try each of the operators **+**, -, / and *.

Exercises

> 1. Write a short program that outputs the value of "5 > 3". You can do this by outputting the String:
>
> ```
> "The answer is " + (5>3)
> ```
>
> Modify the program to output the value with ">" replaced by each of the operators <, ==, and !=
>
> 2. Try to figure out the meaning of the "%" operator, which can be applied to integer type variables, as in "8 % 3". Do this by writing a short program that performs some calculations and writes out the results.
> (Optional: you might also try to figure out the following other operators for integer type variables:
>
> >> (e.g., 3 >> 2)
> << (e.g., 8 << 1)
> & (e.g., 3 & 5)
> | (e.g., 3 | 5).)
>
> 3. Write a short program that (a) assigns the value 5 to the variable x, and 3 to the variable y, (b) outputs the values, (c) trades the two values (so that x takes on y's value, and vice versa), and (d) outputs the values again to show they have traded.

---

[12] Don't assume that the smaller the type, the faster your program will run. There may be an optimal size that depends on the most "natural" size for your computer; for instance, a computer designed primarily to deal with 32 byte numbers may run calculations with int's faster than those with byte, short, or long.

©W. Maddison 1998-2000

Chapter 2. If's and Loops

Objectives

In this chapter, you will learn philosophy and how to go around in circles. We will cover the basic topic of flow control — how to control what instruction the program executes next. In particular, you will learn how to program with:

- if's and else's
- loops of various kinds: for, while, do-while.

Flow Control

The little programs we have run so far start with the first line of instructions, then proceed line by line to the bottom, like a person reading in English. You can think of the computer travelling through the program line by line, executing instructions. However, every so often a special instruction is encountered that tells the computer to jump to a particular place in the program, not necessarily the next line down. Such instructions (like switches on a train track) control the flow of the program, and are important parts of programming.

If's and else's

Often you will want to make a program's calculations depend on some condition, and an "if" statement will do the job, as in:

```
String response;
if (n>1)
     response = "Plural";
else if (n==1)
     response = "Singular";
else
     response = "None";
```

With an "if" statement, you can add as many "else if"s as you want, to ask about all sorts of alternative conditions. You can have no "else if"s if you want:

```
if (n>3)
     response = "Big";
else
     response = "Small";
```

or no else's:

```
if (n<0)
     response = "Negative";
else if (n>0)
     response = "Positive";
```

or only the if:

```
if (n==0)
     response = "Error: n is zero"
```

©W. Maddison 1998-2000

Note the syntax: the condition to be tested is represented by an expression *in parentheses*.[13] The expression within parentheses must take on a true or false value (a boolean expression[14]).

Blocks

In each of the examples above, a single statement occurs after the if, or else. What if you want to execute two statements when the condition is true? Will the following work?

```
if (n>3)
     response = "Big";
     m = 8;
else
     response = "Small";
```

The answer is no. Indentation is actually only for human reading; it doesn't have any effect on how the program actually runs. The above is equivalent to:

```
if (n>3)
     response = "Big";
m = 8;
else
     response = "Small";
```

Thus "m=8" happens regardless of what n is, and the "else" is an illegal statement since it appears to be without an "if". To group the two statements together, you need to frame them using braces:

```
if (n>3)
     {
          response = "Big";
          m = 8;
     }
else
     response = "Small";
```

The braces group the two statements together into a **block**. Some programmers (including me) prefer an alternative way of writing this[15]:

```
if (n>3) {
     response = "Big";
     m = 8;
}
```

[13]If you are an old Pascal programmer like me you might forget the parentheses. Also, if you are an old Pascal programmer, you might forget the semicolons in front of the else's.

[14] Boolean algebra (calculations with the values **true** and **false**, using the operators **and**, or, **if**, and **not**) may be the most important thing you never learned in school. In programming you will more often find yourself wishing you had learned that **not (A and B)** is the same as **(not A or not B)** than you will find yourself wishing you could remember sines and cosines.

[15]I use the form I do because it saves space, both vertically and horizontally (since it saves one level of indentation). In general, Java ignores carriage returns/linefeeds (except for a few circumstances, like the use of the // comment). Thus, you could write:

```
if (n>3) {response = "Big"; m = 8;} else response = "Small";
```

©W. Maddison 1998-2000

```
else
     response = "Small";
```

Blocks can be nested. In the following, the block containing m=8; is nested within the block for the if (n>3):

```
if (n>3) {
     response = "Big";
     if (q == 1) {
          m = 8;
     }
}
```

Here is something to write on the back of your hand: *Variables are valid only within the block in which they are defined*. Thus the following won't work:

```
String response;
boolean yes = false;
if (n>3) {
     int m;
     response = "Big";
     if (q == 1) {
          m = 8;
          boolean yes = true;
     }
}
m = m + 2;
if (yes)
     System.out.println("yes!!!");
```

In fact, this is pretty badly screwed up. The `String response` is recognized throughout this code. The integer m, however, is declared within the block for (n>3), and thus it is not valid outside the block, and hence the statement m = m + 2 is dealing with a variable that appears to be undefined. The compiler will give an error. The `boolean yes` is declared twice, once outside the first if and once within the nested block. *These are treated as two separate variables, with the inner "yes" understood only within the inner block*!!! Thus, the value of the outer "yes" at the end will be false!

Loops: while, do-while, and for

Computers are especially useful for repetitive tasks, and to perform them there are several structures that can repeat calculations again and again until some condition causes the repetition to end. The first and easiest to understand is the **`while`** statement, which is just like an "if" except that the statements in the block following are repeated again and again until the condition is no longer satisfied. For instance:

```
int n = 1;
while (n<100) {
     n = n * 2;
}
System.out.println("n is now " + n);
```

©W. Maddison 1998-2000

Each time, before entering the block, the condition "(n<100)" is tested. If it is true, then the block is executed. When the condition becomes false, the block is not executed and the program resumes after the block. What will be output?

A second sort of loop is the **`do-while`** loop, in which the condition is tested at the end of the loop:

```
int n = 1;
do {
     n = n * 2;
} while (n<100);
System.out.println("n is now " + n);
```

The while and do-while loops are often interchangeable, but they can have different behavior in certain circumstances. For instance, a do-while loop will always be entered at least once, whereas a while loop might not be entered at all, if its condition is not met from the beginning.

One of the most powerful sort of loop is the **`for`** loop. Its general form is as follows:

```
for (initializing statements; condition to continue; incrementing statements) {
     statements...
}
```

When the loop is started off, the initalizing statements are performed. Before entering the loop each time, the condition is tested to see if it is true; when it is false the loop is exited. After each trip through the loop, the incrementing statements are executed. The `for` loop is quite flexible, but a very common form is to increment a variable through a series of values, as follows:

```
int i;
for (i=0; i<4; i=i+1) {
     System.out.println("Next number is " + i);
}
```

What is the output of the above?

Exercises

1. Write a program that outputs numbers from 1 to 20, using each of the three sorts of loops (for, while, and do-while).

2. Write a program that adds up the sum of all the whole numbers from 1 to 100 (1 + 2 + 3 + ... + 100).

Some special assignment operators

In several of our examples we have needed to add 1 to a variable, as in "i=i+1;" This says, "take i, add 1, and put the result back into i". There is a

©W. Maddison 1998-2000

better, quicker way to add 1 to a number. This is the increment operator ++, which is used as follows:

```
i++;
```

This simply adds 1 to the variable i[16]. Thus a `for` loop would typically be written in more compact form as follows[17]:

```
int i;
for (i=0; i<4; i++) {
     statements...
}
```

A similar approach can be used to subtract 1 using -- (i-- and --i).

A slightly more general shortcut are the operators that modify an existing variable. Often you will want to add something other than 1 to a variable, as in:

```
i = i + 5;
```

or you will want to multiply i by something:

```
i = i * j;
```

This says, take i, multiply it by j, and put the answer back into i. These and similar statements that modify an existing variable can be accomplished by a special series of operators (+=, -=, *=, /=, etc.) that leave the second "i" implicit. Thus,

```
i += 5;
```

says "add 5 to i" and is equivalent to "i=i+5;". Likewise,

```
i *= j;
```

says "multiply i by j", and is equivalent to "i = i * j;".

More Exercises

3. Write three examples of infinite loops (a loop that will never end on its own) using `while`, `do-while`, and `for`. Before you try running the program, you might check to see where is the fire extinguisher.

4. Write a bank program that starts with an interest rate (`double`), a deposit (`double`) and a number of years (`int`). You can enter the interest rate directly as a multiplication factor, for instance, "1.075" instead of 7.5%. Have the program calculate the amount of money you will have at the end.

---

[16]It turns out there are two forms of this, i++ and ++i. If you use the statement on its own, as "i++;", it doesn't matter which form you use. However, the expressions i++ and ++i are often used within other expressions, as in "x = y + (i++);" Such statement accomplishes two functions, first to add y and i and put the result into x, the second to add one to i. Thus, i++ is both used in the expression, as well as accomplishes something on its own (adds 1 to i). The difference between i++ and ++i is that i++ uses i in the expression and *then* adds 1 to i, while ++i adds 1 to i and then uses i in the expression. There are two forms of -- also, i-- and --i.

[17]To be even more compact you can declare the variable i within the parenthesis of the `for` statement, as follows:

```
for (int i=0; i<4; i++) {
     statements...
```

©W. Maddison 1998-2000

5. Drawing a grid. This time, go back to the hard disk and find the folder "Template with frame". Duplicate it, and open the project "frameTemplate.prj". We will be using this project because it is set up to give you a drawing window. Run the program to see what happens. It should give you a window with the word "testing" written in blue. Now, go back and open the file source.java. You will find it is a bit more complex, but for now you don't need to know the details. Just find the lines:

```
    public void paint (Graphics g) {
          g.setColor(Color.blue);
          g.drawString("Testing ", 50, 50);
    }
```

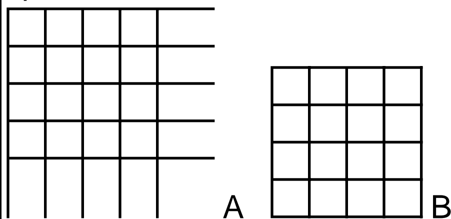
You will want to replace the two middle lines, so as to make the program draw a grid in the window. The goal is to draw a grid of 5 horizontal and 5 vertical lines. To draw a line, use the following:

```
    g.drawLine(<starting x>, <starting y>, <ending x>, <ending y> );
```

where you replace <starting x> with the starting horizontal coordinate of the line, and so on. For instance, to draw a line from point (0,0) to point (50, 100) you would use:

```
    g.drawLine(0, 0, 50, 100);
```

The coordinate system is different from a customary graph, with 0,0 being the top left corner of the window, and the numbers increasing as you move right and down. This window has dimensions 320x320. I suggest you put the top left corner of the grid at about (20, 20) and space the lines about 20 units apart.



If you get a grid like A, great. Now try to tidy it up, so it looks like B. Now, replace any constants with variables so as to make the program general. That is, so that you only have to replace the variable in one place and the grid will change. Use variables for the margin, spacing, and number of lines.

6. Write a program to output out all possible sequences of numbers between 1 and 4. For example, for numbers between 1 and 3, the sequences would be 123, 132, 213, 231, 312, 321).

7. Write a program to output out all prime numbers between 2 and 100. You might find it easiest to do this program in stages, solving simplest problems first.

8. Write a program to find the square root of a number that is greater than 1, for instance 10. Use the following scheme for approximation. You know that the square root of x has to be between 1 and x. So, you have a lower bound (to begin with, 1) and upper bound (to begin with, x) on the square root. If you

}

©W. Maddison 1998-2000

have a lower and upper bound, then you can try out the average of the two. If the average squared is greater than x, then the average is too high, and you know the square root is between the lower bound and the average, and you can reset your upper bound to be the average. If the average squared is less than x, then in a similar way you can reset your lower bound to be the average. In this way the lower and upper bounds have gotten closer together, and if you keep going like this, you will eventually squeeze the lower and upper bounds as close together as you like. When they are closer than some fixed limit, say 0.000001, then you can decide you've found the approximate square root to your satisfaction. Have the program calculate the square root of 10 and output the result. If you want to compare it to the correct answer, use

```
Math.sqrt(10);
```

©W. Maddison 1998-2000

Chapter 3. Arrays & Methods

Objectives

Now that you know about loops, you are ready to think in multiples. We will here learn:

• Arrays (for storage of multiple values) and,

• Methods (little bundles of instructions that are packaged so that they can be easily used again and again).

Arrays

So far we have used variables of the basic number types (int, double, etc.). Each such variable is like a simple container that holds a number value. Sometimes, however, we need to have a whole trainload of values. Each train car holds a single value, and thus the train as a whole can hold many values. Such a series of containers can be represented by a special type of object in Java, the **array**.

To create the array, you must do two things. First, you need to make a variable that will refer to the array. If you want an array of int's, write something like:

```
int[] k;
```

This says: "Please make me variable k that will be a reference to an array of integers of type int". The computer knows it's an array because of the "[]". You can remember this because "[]" is used to refer to elements of an array, as we are about to see. In Java, you don't have to say how many elements are in the array when its reference variable is declared. The reason for this is that this first statement merely tells the computer that the variable k will refer to an integer array, but it doesn't actually make the integer array.

Thus, once the variable k has been so declared, you need to ask the computer to make the array. For this a special statement is needed, the "new" statement:

```
k = new int[5];
```

This says: "Please make an array (by reserving the memory) of int's. In particular, make the array big enough for 5 elements. Once you've made the array, place into the variable k a reference to the array so that I can later refer to the array."[18]

---

[18]Sorry, this may seem a bit complex, but you can ignore all the profundities and just write down the necessary two statements. Besides, once you've learned objects it might be easier. The basic scoop is that there are two pieces of computer memory being used up here, one for the array itself (which is somewhere in the computer's memory), and one which is a reference to the array, sort of like a business card with a telephone number on it. While the array is the business, the variable k is only the business card. However, we can access the business anytime by using k to phone it up and ask it what it contains. Programmers can think of k as

©W. Maddison 1998-2000

The array k is ready for use. You can assign values to its elements by refering to the ith element as k[i], for instance,

```
k[0]=2;
```

This brings up an important point: In Java, *the elements of an array are always numbered from 0 to n-1, where n is the number of elements in the array*[19]. You will get used to counting from 0 to n-1.

The number of elements in an array can be found by adding ".length" to the array's name. Thus, `k.length` is 5.[20] A for-loop to set all the elements of an array to 0 is as follows:

```
int[] k;
k = new int[5];
for (int i=0; i<k.length; i++)
     k[i]=0;
```

(Note that the variable i can be declared within the for's initialization statements, as long as you don't need to use it outside of the loop.)

Exercise

1. Write a program that makes a small array of 5 elements, and assigns values for each. Write a loop that sums the elements of the array.

Methods

---

being almost like a pointer. Another thing — in Java, arrays are almost like regular objects, but not quite. However, you don't yet know what an object is (in formal Java terminology), so I'd better shut up... Oh — Strings are also ambiguous in their behavior.

[19]In Pascal, you can have an array numbered as you like, more or less. Thus, in Pascal you can ask for an array whose three elements are k[1], k[2], and k[3]. Java arrays always start with 0. This is a remnant inherited from C where pointer arithmetic (which doesn't exist in Java) focused heavily on offsets from the first memory location, and the first array element has offset 0. Starting always with zero is a big nuisance since it often means translating from a real-world in which many numbering systems start with the number 1 (was George Washington the 0th President of the United States?). Once again Java suffers from its historical origins.

[20] If you start playing with arrays, you may encounter error messages for the first time. For instance, suppose you wrote:

```
int[] k;
System.out.println("Array k has this many elements: " + k.length);
```

This wouldn't work, because you never actually made the array; you've only declared a variable to contain a reference to some array. Thus, "k" isn't yet referring to any array, and when you ask for k.length, Java thinks "the length of what array????" and it generates what's called a NullPointerException, which basically means "you tried to use a reference to an object to find out something about the object, but your variable wasn't actually referring to any object!!!".

©W. Maddison 1998-2000

We have already seen loops and if's to control the flow of a program's execution. Now, we will make the next major step in structuring our program by introducing **methods**[21]. If objects are like organisms, then methods are their internal organs.

[21]They are called functions in C or C++, functions and procedures in Pascal, but they are not quite equivalent because of the object-oriented nature of Java, which we haven't come to yet.

©W. Maddison 1998-2000

What if you had 3 different arrays, k1, k2 and k3, and you wanted to find the sum of elements in each. You could write:

```
int sum = 0;
for (int i=0; i<k1.length; i++)
     sum += k1[i];
System.out.println("Sum of k1 is " + sum);
sum = 0;
for (int i=0; i<k2.length; i++)
     sum += k2[i];
System.out.println("Sum of k2 is " + sum);
sum = 0;
for (int i=0; i<k3.length; i++)
     sum += k3[i];
System.out.println("Sum of k3 is " + sum);
```

But since the three loops have basically the same structure, you should be able to reuse the same loop for each of the three sums. You can, if you use a method. A method is a piece of code that has a name, and if you use that name in some other part of the source code, the method is invoked. However, the method often takes some input (in the form of parameters passed into the method, written in parentheses after the method name), and often has output (in that the reference to the method can take on a value after the method is completed). If that makes no sense to you yet, have patience. It is important, however, to realize how powerful methods are: building a method is like building a little tool, and you can later reuse that tool many times.[22]

To write the above code with methods you would first invent a general purpose method that sums the elements of an integer array, then you would call it three times. The method could look like this:

```
int arraySum (int[] x) {
     int sum = 0;
     for (int i=0; i<x.length; i++)
          sum += x[i];
     return sum;
}
```

The name of the method is "arraySum". Its name is preceded by "int" to indicate that when it is done, it returns (i.e., takes on the value of) an int. Within the parentheses is "int[] x" to indicate that the method must be passed as input an integer array. Internally, the method will use the name "x" to refer to the integer array (regardless of what the array had been called outside the method).

---

[22]I sometimes like to think of a method as being a named room dug into the ground, with a little skylight on top and a captive gnome inside. When you call the method, you toss in the parameters through the skylight (in the parentheses). The method's gnome sees the parameters fall in, works with them as information, then when he's done his work he might or might not scream back an answer (the result returned).

©W. Maddison 1998-2000

The program would do its thing by calling the method three times, as follows:

```
int sum = arraySum(k1);
System.out.println("Sum of k1 is " + sum);
sum = arraySum(k2);
System.out.println("Sum of k2 is " + sum);
sum = arraySum(k3);
System.out.println("Sum of k3 is " + sum);
```

Note that "arraySum" is used here as if it were a number. The reason is that it returns an integer when it is done. In fact, you could even have written this as follows:

```
System.out.println("Sum of k1 is " + arraySum(k1));
System.out.println("Sum of k2 is " + arraySum(k2));
System.out.println("Sum of k3 is " + arraySum(k3));
```

Here is a whole little program that uses a method in this way.[23] For now, ignore the word "static" added to the start of the method:

```
public class MainClass {
     public static void main(String args[]) {
          int[] k1 = new int[4];
          int[] k2 = new int[4];
          int[] k3 = new int[4];
          for (int i=0; i<4; i++) {
               k1[i]= i;
               k2[i]= i*i;
               k3[i]= i*3;
          }
          System.out.println("Sum of k1 is " + arraySum(k1));
          System.out.println("Sum of k2 is " + arraySum(k2));
          System.out.println("Sum of k3 is " + arraySum(k3));
     }

     static int arraySum (int[] x) {
          int sum = 0;
          for (int i=0; i<x.length; i++)
               sum += x[i];
          return sum;
     }
}
```

Parameters

In the definition of a method, just after the method's name, there must appear parentheses. Within these parentheses are placed a list of the input

[23] This program, like most of the programs in this guide, isn't perfect. For instance, it would be best to insert as the first line of the method "arraySum" the following line, just to prevent disaster:

```
if (x==null) return 0;
```

"null" means that x isn't referring to any array.

©W. Maddison 1998-2000

information expected. These are the **parameters**. In the little arraySum method, the parameter input is an integer array.

Each parameter is indicated by first its type (in the example, "int[]" to indicate integer array) and then by the name used locally, within the method, for the parameter passed (in the example, x). If more than one parameter is passed, they are separated by commas. Thus, a simple method to add two integers would appear as follows:

```
int addTwoNumbers(int x, int y) {
     return x+y;
}
```

which could be used as follows:

```
z = addTwoNumbers(a, b);
```

where a and b are two int variables. This example of course is silly, because you could have written "`z= a+b;`". But it serves as an example. Note that the values passed as parameters into the method are the values of the variables a and b, but internally the method uses the names "x" and "y" for these incoming values.

Even if no parameters are needed by the method, the parentheses are still needed when the method is defined, and when the method is called. Thus:

```
double pi () {
     return 3.14159265358979323846;
}
```

would be called as follows:

```
perimeter = 2 * pi() * r;
```

The empty "()" might seem like unnecessary baggage, and I guess they are, but if nothing else they help you remember that you are dealing with a method.

Void

The methods we have seen return a value — for instance, the little arraySum method above returns an integer value that is the sum of elements in the array. Where the method is called, it can be used as if it has a value, and that value is the value that it returns when it is done. The type of value a method returns is indicated by the type's name in front of the method's name. Thus, a method that returns the area of a circle of radius r might return the answer as a double, as follows:

```
double circleArea (double r) {
     return 3.14159265*r*r;
}
```

Such a method might be used in a statement as follows:

```
double theArea;
theArea = circleArea(1.5);
```

Sometimes, however, you want a method that doesn't return any value. It doesn't yield an answer, it just does something[24]. To indicate that the

[24]The gnome doesn't shout anything back when he's done.

©W. Maddison 1998-2000

method doesn't return any value, you simply write "void" where you normally would write the type of the value returned, as in:

```
void sayHello () {
     System.out.println("Hello");
}
```

©W. Maddison 1998-2000

Thought exercises: Mystery methods

Identify what each of the following mystery methods does. Try to figure these first ones out (0, A, B) without running any programs. Each is passed as input an array of integers. Don't try to program these; just see if you can figure them out using pencil and paper.

2. mystery method 0:

```
String mystery0 (int[] x) {
        String s = "";
        for (int i=0; i<x.length; i++)
              s += "[" + x[i] + "]";
        return s;
}
```

3. mystery method A:

```
double mysteryA (int[] x) {
        double s=0;
        for (int i=0; i<x.length; i++)
              s += x[i];
        return s/x.length;
}
```

4. mystery method B:

```
int mysteryB (int[] x) {
        int s=x[0];
        for (int i=1; i<x.length; i++) {
              if (x[i]>s)
                    s = x[i];
        }
        return s;
}
```

©W. Maddison 1998-2000

Programming Exercises

5. Write a program with a method that takes two integers as parameters and returns the maximum of the two.

6. Write a method that takes an integer array as a parameter and returns the number of elements of the array that have value 0.

7. Write a method that takes an double array as a parameter and returns the maximum value among elements of the array.

8. When rolling two dice, there is only one way to make a sum of 2 (snake eyes — each die needs to show 1). There are many ways to achieve 7. Write a program using arrays to calculate how many different ways there are to make a particular sum (from 2 to 12) with a roll of two dice. The program should also count the total number of ways to roll two dice; you can check that your program is working by seeing if it yields 36 (6 * 6).

9. Write a program with a method `magnitude` that takes an `int` number as a parameter, and returns an `int` that is the highest power of ten not greater than the number (i.e., 3 would return 1, 23 would return 10, 5307 would return 1000, 10000 would return 10000). Return 0 if a negative number is passed to it. Next, add a method `findDigit` that is passed two `int` parameters, the second being a power of 10. The method should return an `int` that is the number's digit in the appropriate place (e.g., passing 5307 and 100 should yield the result 3). Next, add a method `decomposeInt` that takes an `int` parameter and returns a String that is a decomposition of the parameter digit by digit. For instance, if 5307 is input, the method should return the string::

```
"5307 is 5 * 1000 + 3 * 100 + 0 * 10 + 7 * 1".
```

This last method will probably make use of the first two.

10. (optional, do only if bored) Write a program that starts as follows:

```
int x = 120;
int y = 7;
```

Then add to it code that performs long division to divide x by y. By "long division" I mean that you use only whole-number arithmetic (no real/floating point/decimal arithmetic), and that you work from left to right on the digits of x, exactly like long division.

©W. Maddison 1998-2000

Bonus mystery methods

Identify what each of these mystery methods does. These methods are particularly difficult, and don't sweat if you can't figure them out. Try them out in the mystery methods project on the disk to see how they behave.

mystery method C:

```
int[] mysteryC (int[] x) {
      int[] y = new int[x.length];
      int[] n = new int[x.length];
      int maxN = 0;
      int cats = 1;
      y[0] = x[0];
      n[0] = 1;
      for (int i=1; i<n.length; i++)
            n[i]=0;

      for (int i=1; i<x.length; i++) {
            boolean found=false;
            for (int c=0; c<cats; c++) {
                  if (x[i]==y[c]) {
                        n[c]++;
                        if (n[c]>maxN)
                              maxN = n[c];
                        found=true;
                  }
            }
            if (!found) {
                  n[cats]=1;
                  if (n[cats]>maxN)
                        maxN = n[cats];
                  y[cats]=x[i];
                  cats++;
            }
      }
      int nm=0;
      for (int i=0; i<cats; i++)
            if (n[i]==maxN)
                  nm++;

      int[] result = new int[nm];
      nm=0;
      for (int i=0; i<cats; i++) {
            if (n[i]==maxN) {
                  result[nm]=y[i];
                  nm++;
            }
      }
      return result;
}
```

©W. Maddison 1998-2000

mystery method D:

```
int[] mysteryD (int[] x) {
	int[] result;
	int sL=0;
	boolean bL=false;
	int sG=0;
	boolean bG=false;
	for (int i=0; i<x.length; i++) {
		int s=x[i];
		int jL=0;
		int jG=0;
		int jE=0;
		for (int j=0; j<x.length; j++) {
			if (x[j]>s)
				jG++;
			else if (x[j]<s)
				jL++;
			else
				jE++;
		}
		if (jG==jL) {
			result = new int[1];
			result[0] = s;
			return result;
		}
		else if (jL<jG) {
			if (jL+jE>jG) {
				result = new int[1];
				result[0] = s;
				return result;
			}
			else if (s>sL || !bL) {
				sL = s;
				bL = true;
			}
		}
		else if (jL>jG) {
			if (jG+jE>jL) {
				result = new int[1];
				result[0] = s;
				return result;
			}
			else if (s<sG || !bG) {
				sG = s;
				bG = true;
			}
		}
	}
	result = new int[2];
	result[0] = sL;
	result[1] = sG;
	return result;
}
```

©W. Maddison 1998-2000

Chapter 4: Classes and Objects

Objectives

This chapter will cover the basic way that Java is structured into objects. After it you should be familiar with:

- Classes, which you can think of as blueprints for
- Objects.

You should know how to instantiate an object, what a constructor is, and what are static methods and fields.

Classes and Objects

Drum roll please. So far we have been studying physiology *in vitro*; now we will study it *in vivo*. All of the expressions, statements, and methods we have discussed in Java must exist within **objects**[25]. Objects are like organisms: they are born, live, have characteristics and responses, and die. They don't have a genome to specify how they are to be built; instead, their specifications are given by defining **classes**. You can think of classes as being like blueprints, or molds, for making objects, and each time you make an object of that class, you are creating an **instance** of the class. Thus, at one part of the program you could define a class Organism[26] as follows:

```
class Organism {
     public void poke () {
          System.out.println("Ouch");
     }
}
```

This class definition merely creates the blueprint for Organism objects. To actually create an Organism object, you need to do something very similar to what you did for arrays. Namely, you must declare a variable that will contain the reference to the object:

```
Organism fido;
```

Then you need to actually create or **instantiate** the object by using "**new**":

```
fido = new Organism();
```

Once you have done so, the variable fido will contain a reference to the new object so created. The variable does not actually contain the whole object; it merely contains a reference to it[27], just as your name is not you, but rather a

---

[25]Or, at least, within the blueprints for objects (i.e., classes).

[26]It is traditional in Java that all class names begin with upper case letters (e.g. Organism), and all variables and method names begin with lower case letters (e.g., headWidth, getNoseColor()). All constants, which we have not yet dealt with, traditionally are in all upper case letters (e.g., RED, BLUE), but I don't like that, since it's shouting.

[27]A variable containing a reference to an object is the closest thing in Java to a pointer. Those of you with experience with pointers can fairly safely think of these references like pointers. Thus, the variable fido will contain a pointer to the object created. Being aware of the distinction between a reference to an object and the object itself is important in several contexts, including passing parameters to methods. In Java, parameters are always passed by

reference to you (recall that in Spanish you don't say "I am <*fill in your name here*>" but rather "I am called <*fill in your name here*>"). Usually, however, you can think of "fido" as being the object.

Once fido exists, you might wonder what you can do with it. It's not an array, so you can't store numbers in it. What you can do with fido depends entirely on its characteristics as implicit in its class definition. The only thing an Organism has, according to the definition, is the method "poke". Since the method "poke" is listed as public, objects outside of Organism can "see" it and access it. Thus, other parts of the program can invoke fido's poke method as follows:

```
fido.poke();
```

The object fido will respond by outputting "ouch".

Exercise

1. Write a little program that defines the Organism class as written above. Make an Organism object and poke it.

Fields

Here is an example of a little program that creates Conversationalist objects, then converses with them:

---

value, that is, a copy of the parameter is passed into the method. However, since a reference to an object and a copy of that reference both point to the same object, if a method expects an object reference as a parameter, the method receives a reference to the original object, not a copy of the object. Thus if you use the reference within a method and alter the object's state, you will be altering the original object's state. This applies to objects (except Strings) and to arrays. Strings behavely since they are immutable once made, and thus the moment you attempt to change a String passed as a parameter, you're actually working on a new copy of the String. Primitive types are also passed by value, but since variables of primitive types contain the value itself (not a reference to it), the receiving method gets a copy of this value and no reference to the original variable. Thus passing a reference to a primitive type so as to be able to change the original copy (e.g. using var in Pascal or passing a pointer in C) is not exactly easy in Java, though you can do it using a wrapper object. There is more to this story that I cannot right now relate, because the hour is much too late, ...

©W. Maddison 1998-2000

```
class Conversationalist {
     int statements = 0;
     public String chitchat () {
          statements++;
          if (statements==1)
               return "Hello!";
          else if (statements==2)
               return "Cold lately, eh?";
          else if (statements==3)
               return "Good bye.";
          else if (statements==4)
               return "Sorry, I must be going now....";
          return "Duh - uh?";
     }
}

public class MainClass {

     public static void main(String args[]) {
          Conversationalist c1, c2, c3;
          System.out.println("====================");
          System.out.println(" Scene: 3 conversationalists");
          c1 = new Conversationalist();
          c2 = new Conversationalist();
          c3 = new Conversationalist();
          System.out.println(c1.chitchat());
          System.out.println(c1.chitchat());
          System.out.println(c2.chitchat());
          System.out.println(c3.chitchat());
          System.out.println(c1.chitchat());
          System.out.println(c2.chitchat());
          System.out.println(c1.chitchat());
          System.out.println(c1.chitchat());
     }

}
```

This example shows one of the most important aspects of objects. Note that the Conversationalist objects each has a special integer variable called "statements". Such variables that appear underneath the first line of the class definition and outside of any methods are called **fields** of the class. These variables represent the state of the object, much as the fields of a record in a data base indicate the salary, weight, favorite color, and telephone number of that entry in the data base. In each Conversationalist object, "statements" counts the number of statements that the object has already made. Thus, each of the Conversationalists c1, c2 and c3 are a bit different, because at any point in the program they might differ in how many statements each has already made.

Find the Conversationalist project on the hard disk and run it to see what happens. Modify it as you wish.

©W. Maddison 1998-2000

Anatomy of a class

Writing a class definition is a basically inventing a new type of variable. Once you've defined it, you can make new instances and refer to them. The difference between classes and types like integers and arrays is that the class's objects not only store information, but also do things (via their methods). Let us consider in more detail the anatomy of a class. I've changed the Organism class to be a bit more elaborate:

```
class Organism {
    intnumberGametes;                          // fields
    int[]genotype;
    static final int X = 0;
    static final int Y = 1;

    public Organism (int gender) {             // constructor method
        genotype = new int[2];
        genotype[0] = X;
        if (gender == 0) {
            numberGametes = 1000; //male
            genotype[1] = Y;
        }
        else {
            numberGametes = 10; //female
            genotype[1] = X;
        }
    }

    public int[] makeGametes () {              // method
        int[]gametes = newint[numberGametes];
        Random random = new Random();
        for (int i=0; i<numberGametes; i++) {
            gametes[i]=genotype[random.nextInt(2)]%
        }
        returngametes;
    }
}
```

Recall that the variables that belong to the object or class in general are called fields. They store the state of the object. (Two of the fields have "static" in front of them — these are about to be discussed.)

There are two methods, one of which is a special method called a "**constructor**". This is the first method shown; its name is the same name as the class ("Organism"), and there is no separate return type indicated (e.g., "void", "int"). This method in a sense creates new Organisms. As you have seen, when you create a new object of the class Organism, you do it as follows:

```
Organism rover = new Organism(0);
```

What you are doing is calling the constructor method. Since the constructor method takes an integer parameter, an integer must be passed to it. The

©W. Maddison 1998-2000

constructor does two things. First, it implicitly creates a new object of the class, and second, it performs whatever instructions are within the constructor method.

You can think of the constructor as being like the developmental process of the object. Normally, you put into the constructor any instructions needed to prepare an object for its later functioning. You might set default values for variables, allocate arrays for later use, and so on. Once the constructor method is done, the object is an adult, ready to head out into the world.

Thought Exercise

2. Are the Organism objects in the example above haploid or diploid?

Scope of variables

One thing you may have discovered by now is that you can declare a variable in one part of a program and another part of the program acts as if it doesn't recognize the variable. This has to do with the **scope** of the variable.

The general rule is that a variable is recognized only within the block (squiggly braces) in which it is defined. Thus a variable declared within the body of a loop is only recognized within the loop.

For instance, consider the following:

```
public class MainClass {
     int numOne = 36;
     public static void main(String args[]) {
          int numTwo = 0;
          if (numOne == numTwo) {
               String s = "They're the same!";
          }
          else {
               String s = "They're different!";
          }
          System.out.println(s);
          sayGoodbye();
     }

     static void sayGoodbye(){
          int numThree;
          numThree = numOne*numTwo;
          System.out.println("But their product is " + numThree);
          System.out.println("and I must be going now, so 'bye!");
     }
}
```

This class has two methods (`main` and `sayGoodbye`) and one field (`numOne`). Perhaps it looks fine but in fact it is full of errors! The errors are as follows: `numTwo` is declared within the main method (between its outer squiggly brackets), and yet it is used within `sayGoodbye`. As far as `sayGoodbye` is

©W. Maddison 1998-2000

concerned, `numTwo` has never been declared, and it won't recognize it. The String `s` is declared within the block following the first if, and also within the block following the else. These are two separate blocks, and they are treated as two separate variables. Outside of the scope of each block the variable is not defined, and thus the `System.out.println(s)` won't function.

You might wonder about `numOne`. It's fine, because it's defined within the block of the entire class `MainClass`. Thus, within sub-blocks (such as within the two methods) `numOne` is recognized. Fields like `numOne` are useful if you want all methods within a class to have a storehouse of shared information. Such variables can be understood and modified by any of the methods of the class.[28]

Static methods and fields

With the exception of static methods, which I am about to discuss, methods must always be invoked by reference to an object that contains the method. Thus, you couldn't say simply

```
System.out.println(chitchat());
```

in the above example. Rather, you had to say for which object you wanted to call the method chitchat. You can think of methods of objects as being like buttons on the objects, and by invoking the method you are saying "please push the button chitchat on the object c1". You have to say whose button you want to push.

The exceptions are static methods. Often, you will need to have a method that can be called regardless of whether an object exists of the class containing the method. To understand static methods, let us recall the method "poke" of the Organism class we discussed earlier. To invoke the method, we had to use fido.poke() where fido is a reference to an Organism object (the Organism object doesn't need to be called fido; it just needs to be some instantiated Organism). If we hadn't made the Organism fido, we couldn't invoke the poke method. It would be like asking to push the button on an Organism when you didn't have an Organism in front of you. However, if we had put in the "static" keyword as follows:

```
class Organism {
     public static void poke () {
          System.out.println("Ouch");
     }
}
```

Then it turns out you can invoke poke without having any existing Organism object. You can invoke a static method by prefixing it with the name of the class itself (instead of a name of an object of the class):

```
Organism.poke();
```

[28]There are not truly global variables within Java. All variables must exist within classes. However, by declaring them as "public", they can be accessed from other classes.

©W. Maddison 1998-2000

Note that you use the name of the Class itself. It is as if you are hitting the button on the blueprint, instead of a button on the object itself. For this reason, static methods are often called **class methods**. In contrast, methods that aren't static are often called **instance methods**, in that they can only be used in reference to an instance of the class, i.e. an object that contains them.

Class (static) methods are often used for utility functions that you might want to use quickly without bothering to create an object. For instance, if you want to find the square root of a number, what built-in function do you call? Since all methods (functions) in Java must belong within classes, you have to make reference to a class or object. It turns out that the built-in square root function is called sqrt and it is within the Math class. If you quickly calculate a square root, you don't want to create some special Math object. The square root method is a static method, and thus you can use:

```
root = Math.sqrt(x);
```

Just as methods can be static or not, fields can also be static or not. A static field has the same value for all objects of the class. If you change the value of the variable in one object, it gets changed for all objects of the class.[29] You can use static fields for various purposes, for instance to keep count of how many objects of a particular class have been created.

Exercise

1. Write a program that includes an Organism class that is constructed as follows.

The Organism class has two fields to store references to baby Organisms (that is, each Organism can have at most two baby Organisms).

The Organism class contains a method that takes an integer parameter. This method, in a sense, "eats" the integers passed to it. When the total amount eaten by an Organism is greater than 10, then the Organism gives birth to a baby Organism.

Once the Organism has a baby, half of the integers it eats should go to feed each of its babies (while it has only one baby, it eats the other half itself). There should be a method that returns the number of direct offspring (its babies) that the Organism has.

There should be a method that returns the total number of descendants the Organism has. Its total number of descendants is, of course, the number of its F1 offspring added to the total number of each of the offsprings' descendants.

Start the program and keep feeding it until the integers that are between 1 and 10 until you have fed it total 1000. Periodically along the way ask how many

[29]Thus, a static field is a bit like a global variable in other languages. In fact, since both static fields and static methods can be used without creating an object, they allow you "cheat" on the otherwise strictly object-oriented programming of Java. You could write a Java program that effectively is not object oriented using static fields and methods. Nonetheless, Java is still closer to being purely object oriented than some other languages like C++.

©W. Maddison 1998-2000

total descendants it has. If you want, adjust it or its feeding regime to learn more about how it behaves.

What else there is

You now have an overview of Java. However, there is more to Java that we will get into only as we need to. Some of the basic elements of the language that we haven't covered are: interfaces, exceptions, threads, input and output, wrapper classes for primitive types, utility classes like Vector, security issues, web applets. Some of the standard packages deal with windows, components, menus, drawing, fonts, images, events, clipboards, reflection, network access, remote method invocation, data compression, databases. When you include all of these standard packages, Java is HUGE, but you can do a lot with

Supplemental stuff: Subclasses, inheritance and overriding

Here is a brief introduction to a fundamental aspect of Java: subclasses. Suppose we have created a basic Organism class that you can poke, say hello to, and find out its name:

```
class Organism {
     String myName;
     public Organism (String name) {
          myName = name;
     }
     public String getName() {
          return myName;
     }
     public void sayHello() {
          System.out.println("Hello");
     }
     public void poke() {
          System.out.println("Squeak");
     }
}
```

You've got that class working just fine, but you discover you need to have two sorts of organism, males and females. One approach would be to create two classes instead of one: class Male, and class Female. Trouble is, you'd have to duplicate the class, including the getName, sayHello and poke methods. The natural way to do this in Java is to leave the definition of Organism just the way it is, but then create two subclasses. You can say "I want a class of Organisms called Male, which will be exactly like your generic organism except for modifications and additions that I indicate". Likewise for Females. Here is a little program that defines two subclasses of Organism, Male and Female:

©W. Maddison 1998-2000

```
class Organism {
    String myName;
    public Organism (String name) {
        myName = name;
    }
    public String getName() {
        return myName;
    }
    public void sayHello() {
        System.out.println("Hello");
    }
    public void poke() {
        System.out.println("Squeak");
    }
}
/*==================================================================*/
class Male extends Organism {
    public Male (String name) {
        super(name);
    }
    public void poke() {
        System.out.println("Squaaaaaaak");
    }
    public void sweat() {
        System.out.println("Phew, it's boiling in here!");
    }
}
/*==================================================================*/
class Female extends Organism {
    public Female (String name) {
        super(name);
    }
    public void poke() {
        System.out.println("Squiiiiiiik");
    }
    public void glow() {
        System.out.println("Insufferable, isn't it?");
    }
}
/*==================================================================*/
public class MainClass {
    public static void main(String args[]) {
        Organism lindsey = new Organism("Lindsey");
        Male joe = new Male("Joe");
        Female mary = new Female("Mary");
        interactWith(lindsey);
        applyHeat(lindsey);
        interactWith(joe);
        applyHeat(joe);
        interactWith(mary);
        applyHeat(mary);
    }
    static void interactWith(Organism who) {
        System.out.println("===interacting with " + who.getName() + "===");
        who.sayHello();
        who.poke();
    }
    static void applyHeat(Organism who) {
        if (who instanceof Male)
            ((Male)who).sweat();
        else if (who instanceof Female)
            ((Female)who).glow();
    }
}
```

©W. Maddison 1998-2000

joe is a Male. It also is an Organism, and hence you can still sayHello to it and get its name. You can say joe.sayHello(), just as you can say mary.sayHello() and lindsey.sayHello(). Thus, the subclass (Male) **inherits** from its superclass (Organism) all the methods of that superclass, and they don't need to be repeated. You can treat a Male as if it were just any old Organism, hence you can pass it as a parameter to a method expecting an Organism (as was done in passing Joe into interactWith).

However, Males have an **additional method** that Females don't have: sweat(). Females have a method, glow(), that Males don't have. Thus, you could say joe.sweat() and mary.glow(), but you couldn't say joe.glow() or mary.sweat(). Nor could you say lindsey.sweat(), since lindsey is a genderless Organism.

What happens with poke()? Here Organisms, Males and Females all have the method. The subclasses having the same method **overrides** the method in organism, and thus if you say joe.poke() you'll get the Male version of poke. This is true even if you've temporarily forgotten that joe is a male. The method interactWith doesn't know whether the Organism "who" is a Male or Female, and yet when it call's who's poke method, it gets the gender-appropriate response (because, I suppose, "who" knows what gender it is).

Two other things to note. First, the constructor of Organism's takes a String, and thus the constructors of Male and Female must pass the String to their superclass Organism. This is done by the call to **super**(). This means "call the constructor of my superclass". Second, if you have an Organism and it might be either male or female, you can find out which using **instanceof**. This is a general way to find out two what class an object belongs.

Subclasses are incredibly useful, because they allow you to make various forms of some basic class, without having to repeat your work on the basic class.

More Supplement: overloading

One more note: you can have more than one method with the same name in the same class, as long as the list of parameters passed to them are distinct (in terms of the types of variables passed). This is called **overloading**. For example, you could add a second poke method:

```
class Organism {
    String myName;
    public Organism (String name) {
        myName = name;
    }
    public String getName() {
        return myName;
    }
```

©W. Maddison 1998-2000

```
    public void sayHello() {
        System.out.println("Hello");
    }
    public void poke() {
        System.out.println("Squeak");
    }
    public void poke(int intensity) {
        if (intensity>10)
            System.out.println("Yiiiiaaaaaaaaaaaah");
        else
            System.out.println("Squeak");
    }
}
```

If the Organism's poke method was called and passed an integer, as in lindsey.poke(8);, its second poke method would be called. If no integer was passed, its first poke method would be called. In effect, the two poke methods are truly different methods. We just happened to have given them the same name, and the computer knows which is being called because of the differences in parameters. The reason to give them the same name is that it helps us understand the code. If you wanted to give a default poke, without specifying intensity, you could use the first method. If you wanted to specify its intensity explicitly, use the second.

©W. Maddison 1998-2000

Chapter 5. Random numbers and Population simulations

Objectives

- to learn about random number generation in Java
- to use packages and type casting
- to begin writing simulations of population genetics and population dynamics.

Random numbers

In Java you can generate random numbers[30], but since Java is fully object-oriented you don't do it by calling a simple function as you would in Pascal or C. Rather, you create an object that generates random numbers, and then get the object to generate random numbers for you by calling its methods.

Packages

But first a little note on packages. Java's built-in classes are bundled together as packages. To access these classes, you have to make sure you indicate at the top of the program that you want to use the package. You do this using the `import` statement.

For instance, the Random class is part of the utilities package whose name is "`java.util`". To use Random in your program, you need to put the following line at the very top of your program:

```
import java.util.*;
```

This tells the compiler you want access to all of the classes in the util package. (Technically, the full name of the Random class is java.util.Random.)

You can create your own packages if you want. They can act as libraries that programs can access for already-built classes.

Back to Random

If you've put the import statement in correctly, then you can use Random. Start off by creating an object of the class Random, as follows:

```
Random myRandomNumberGenerator = new Random();
```

[30]Some random number generators are better than others. All of them are only pseudo-random. The bad ones generate patterns that can cause simulations to give strange results. My guess is that you don't need to worry much about this for now, but I really don't know how good are the Java random number generators.

©W. Maddison 1998-2000

(You can give it a name other than "myRandomNumberGenerator".) Now, you have a random number generating object called "myRandomNumberGenerator", and you can use its methods to give you random numbers.

For instance, the methods nextDouble() and nextFloat() return a value of type double and float respectively that is uniformly distributed between 0.0 and 1.0. Thus, if you wanted to put a random number between 0 and 1 in the double variable "d" you would use:

```
double d;
d = myRandomNumberGenerator.nextDouble();
```

The methods nextLong() and nextInt() return long and int values respectively that are uniformly distributed across the range of those types (thus nextInt() returns a number between -2147483648 and 2147483647).

The method nextGaussian() returns a double that is normally distributed with mean 0 and variance 1.

The method setSeed(long seed) can be used to set the seed for the random number generator. The default seed is the current time. You may want to set the seed yourself (after creating the Random object but before asking for random numbers) so as to generate a repeatable series of pseudo-random numbers.

When to create a random number generator

As described, you have to instantiate a random number generator object using new Random(). When should you create this? One common error is to create it within a method that returns a random number. Imagine that you want a coinFlip method that returns true for heads, false for tails. You might write a method as follows:

```
boolean coinFlip() {
     Random myRandomNumberGenerator = new Random();
     return (myRandomNumberGenerator.nextDouble()<0.5);
}
```

Every time you want a coin flip, you could call this. However, it's not a good procedure for two reasons. First, a new random number generator is created for each coin flip. A random number generator is an object, and you can keep calling its procedure nextDouble again and again. Creating a new random number generator for each coin flip is a bit like buying a new car for every trip to the grocery store. This is extremely wasteful in time and memory.

Second, it turns out that random number generators instantiated very soon after one another tend to generate similiar numbers for their first numbers, at least on the MacOS, for reasons that are beyond me. This means that your numbers won't be too random.

©W. Maddison 1998-2000

One solution is to create the Random object outside of the method, and use it again and again within the method. This could be done by passing a reference to the Random as a parameter to the method, as in

```
boolean coinFlip(Random myRandomNumberGenerator) {
     return (myRandomNumberGenerator.nextDouble()<0.5);
}
```

Another is to have a variable at the scope of the class:

```
Random myRandomNumberGenerator = new Random();
.
.
.
boolean coinFlip() {
     return (myRandomNumberGenerator.nextDouble()<0.5);
}
```

Exercises

1. Write a program that does a basic check on the random number generator by using it many times and seeing how the distribution of numbers looks. Do this for both the nextDouble and nextGaussian methods. For nextDouble, break the interval between 0 and 1 in 10 pieces and see that each of the pieces gets hit about as many times when 10000 random numbers are generated. For the nextGaussian, try 10 intervals also (the tail intervals could stretch to infinity if you want).

2. Make a method to generate a random integer number between 1 and 6.

3. Make a program that randomly rolls two dice 10000 times and records the frequency of various sums (from 2 to 12). Check these against your calculations of Chapter 2 exercise 8.

4. Make a method that randomly reshuffles the elements of an integer array. The way to randomly reshuffle is as follows. Let's suppose the array has 10 elements. Choose element #0 as the target. Randomly choose a number between 0 and 9; let's call the number R. Interchange the value in element #0 with the value in element R (unless of course R=0 in which case you don't have to do anything). Now choose element#1 as a target. Randomly choose a number between 1 and 9; call this number R. Interchange the value in element #1 with the value in element R. Continue like this, choosing 2, then 3, ..., then 8 as the target. Each time, choose a random number between the target number and the last element number of the array, and interchange with the target. When you are done the last shuffle, the elements of the array are randomly reshuffled. Try it out on a few arrays.

5. Now make a method that fills the elements of an array of length L with the numbers between 0 and L-1 in random order (*so that the numbers don't repeat*). Do it *without* first filling the array and reshuffling it as in exercise 4 (that would be too easy). Instead, fill the array element by element, choosing the next number randomly.

©W. Maddison 1998-2000

Population simulations

Simulations of population dynamics or genetics might be broken down into two main types: (1) Those that represent individuals or genes individually, e.g. as elements in array, or as objects. These individuals or genes behave according to rules like the behavior of their real-world analogs. The simulation must keep track of all of the individuals or genes and make them do their thing. (2) Those that store only the numbers or frequency of genes or individuals. Thus for a population only its size (N) might be remembered, and individuals are nowhere represented. A stochastic model of change might govern changes in population density, gene frequency, etc. We will begin with an example of the second type, using population growth.

Population growth

Let's suppose we want to program a population's growth. Each generation, the population grows according to the old-fashioned logistic equation (which has lots of problems, but it's easy to program), with two parameters: the intrinsic rate of growth (r) and the carrying capacity (K):

$$N_{t+1} = rN_t(1-\frac{N_t}{K})$$

Let's suppose we want to program the population's growth for 10 generations starting at size 10, with r = 2 and K=10000. (Note that this is purely deterministic.) The first thing to do is to write a small program that merely checks that we have the equation correct:

```
public class MainClass {
     public static void main(String args[]) {
          //setting things up
          int r = 2;
          int k = 10000;
          int n = 10;
          n = (n*r*(1 - n/k)); // the equation
          System.out.println(n);
     }
}
```

If you run it, it gives the answer 20, which seems right (each of the 10 offspring have 2 offspring, given that the population size is still so much smaller than the carrying capacity).

Now you make a loop to make it run for 15 generations. It gives the answer:

```
20
40
80
160
320
640
1280
2560
5120
10240
0
0
0
0
0
```

Something seems wrong. Note that the population seems to grow purely exponentially (multiplied by 2 each time) until it suddenly crashes to 0.

©W. Maddison 1998-2000

What's wrong? The problem is that n/k is done as an integer calculation, and always is 0 as long as n is less than k. Thus, the term that is supposed to slow the growth down never gets turned on until it's too late and N_t has already passed K. You need to force the equations to be done as double or float numbers, using casting.

Type casting

Sometimes you have a floating point number and you want it to be treated as an integer. Or, you have the integer 1 and you want it to be treated as the floating point number 1.0 (remember the computer uses different calculations for each). In general, you may have a variable of one type and you want its value to be treated as if it were another type. Sometimes, if the types are compatible enough, you can do this by casting.

For instance, let's say you had an int variable and you wanted its value to be treated as a double within a calculation. You simply put "`(double)`" in front of it to tell it to convert to double. Thus our program's line should have been:

```
n = (int)(n*r*(1 - ((double)n)/k));
```

The "(double)" tells it to get the value of n and turn it into a double before using it[31]. Thus, the division by k will be forced to be a floating point division, and its accuracy preserved. This forces the whole calculation to be done with floating point (since a calculation involving an integer and a floating point is automatically "upgraded" to a floating point calculation). For that reason, the (int) is needed at the start, since that forces the result back into an int before trying to stuff it into n.[32]

The program should thus look like this:

```
public class MainClass {
     public static void main(String args[]) {
          int r = 2;
          int k = 10000;
          int n = 10;
          for (int i=0; i<15; i++) {
               n = (int)(n*r*(1 - ((double)n)/k));
               System.out.println(n);
          }
     }
}
```

and its output should look more like a logistic (it will converge more or less to 5000, not 10000).

[31]The expression `(double)n` does not actually change the type of the variable n. n remains a container for an integer value. It's just that the value retrieved from n is converted to a double before it is divided by the value in k.

[32]BEWARE: casting a double to an int has the effect of truncating the decimal places. Thus, 1.999 gets truncated to 1. Obviously, you might not want to do that. If you want to round up or down to the nearest integer, you could add 0.5 before truncating (think about it) or you could use the static method `Math.round(double a)` which returns a `long` or `Math.round(float a)` which returns an `int`.

©W. Maddison 1998-2000

Population genetics

For the second example, suppose we want to simulate a population of dioecious diploid organisms and look at the Hardy-Weinberg equilibrium. Just for fun, let's have all males start off with one allele and all females start off with the other. Let's assume for simplicity the population has a fixed number of males and females (1000 each) every generation. Each individual has one locus, with alleles 0 and 1. Have random mating among males and females, and produce the next generation of 2000 offspring.

We can start off with a smaller population of 8 individuals (4 male, 4 female) and see if we can make it work. We need to store for each individual its two copies of the gene. Thus, we could have two arrays, one a 4x2 array (individuals X copies) for males and another for females. Let's make the arrays, give all the males allele 0, and all the females 1, and write it out just to check:

```
public class MainClass {
	public static void main(String args[]) {
		System.out.println("==== H-W eq simulation ====");
		int[][] males, females;
		males = new int[4][2];
		females = new int[4][2];
		for (int individual=0; individual<4; individual++) {
			for (int copy=0; copy<2; copy++) {
				males[individual][copy] = 0;
				females[individual][copy] = 1;
			}
		}
		writeGenotypes(males, false);
		writeGenotypes(females, true);
	}

	static void writeGenotypes(int[][] individuals, boolean isFemale) {
		for (int individual=0; individual<individuals.length; individual++){
			if (isFemale) System.out.print("fe");
			System.out.print("male " + individual + ": ");
			for (int copy=0; copy<2; copy++)
				System.out.print(individuals[individual][copy] + " ");
			System.out.println("");
		}
	}
}
```

Note that a separate method is used to write the genotypes. Now we need to make the next generation. Suppose that sons and daughters are formed by choosing a random sperm and a random egg from the population. Thus, we need to do two things: (1) make arrays of sons and daughters to store the genotypes of the next generation and (2) cycle through these arrays and populate them with sons and daughters by assigning genotypes taken from the males and females (i.e., by choosing a random sperm and a random egg). To begin this process, let's prepare it so that everything is ready except the method to choose random gametes. Here's a first version:

```
public class MainClass {

	public static void main(String args[]) {
		System.out.println("==== H-W eq simulation ====");
		int[][] males, females;
```

©W. Maddison 1998-2000

```
            males = new int[4][2];
            females = new int[4][2];
            for (int individual=0; individual<4; individual++) {
                  for (int copy=0; copy<2; copy++) {
                        males[individual][copy] = 0;
                        females[individual][copy] = 1;
                  }
            }
            writeGenotypes(males, false);
            writeGenotypes(females, true);

            //NEXT GENERATION
            int[][] sons, daughters;  // make arrays for next generation
            sons = new int[males.length][2]; //make just as many sons as fathers
            daughters = new int[females.length][2];  //and daughters as mothers

            // making sons
            for (int son=0; son<males.length; son++) {
                  sons[son][0] = chooseRandomGamete(males);
                  sons[son][1] = chooseRandomGamete(females);
            }
            // making daughters
            for (int daughter=0; daughter<females.length; daughter++) {
                  daughters[daughter][0] = chooseRandomGamete(males);
                  daughters[daughter][1] = chooseRandomGamete(females);
            }
            writeGenotypes(sons, false);
            writeGenotypes(daughters, true);
      }

      static int chooseRandomGamete(int[][] individuals) {
            return individuals[0][0];  //doesn't really work yet
      }

      static void writeGenotypes(int[][] individuals, boolean isFemale) {
            for (int individual=0; individual<individuals.length; individual++){
                  if (isFemale) System.out.print("fe");
                  System.out.print("male " + individual + ": ");
                  for (int copy=0; copy<2; copy++)
                        System.out.print(individuals[individual][copy] + " ");
                  System.out.println("");
            }
      }
}
```

This works to a certain point, but `chooseRandomGamete` doesn't really work. It merely passes back the first gene copy of the first individual always. Instead it needs to choose a father or mother randomly, then choose which copy to donate into the gamete, then return the allele at that place. Thus we need to first create a random number generator; let's suppose we call it `randomNG`. Then we use it in `chooseRandomGamete` as follows:

```
static int chooseRandomGamete(int[][] individuals) {
      int randomIndivid = (int)(individuals.length * randomNG.nextFloat());
      int randomCopy = (int)(2 * randomNG.nextFloat());
      return individuals[randomIndivid][randomCopy];
}
```

Note that we are converting the random float which is between 0 and 1 into a number between 0 and individuals.length-1. Think carefully about why it works as shown above (note that it truncates instead of rounding).

©W. Maddison 1998-2000

Now it would be useful to rewrite the writeGenotypes procedure so it gives a summary instead of every individual. This will be important when we move to a population of 1000. Here is an alternative:

```
static void writeGenotypes(int[][] individuals, boolean isFemale) {
    int num00, num01, num11;
    num00=num01=num11=0;  //check out this trick!
    for (int i=0; i<individuals.length; i++){
        if (individuals[i][0] ==0 && individuals[i][1] ==0)
            num00++;
        else if (individuals[i][0] ==1 && individuals[i][1] ==1)
            num11++;
        else
            num01++;
    }
    if (isFemale)
        System.out.println("females 00: "+ num00 + "  01: " + num01 + "  11: "+ num11);
    else
        System.out.println("  males 00: "+ num00 + "  01: " + num01 + "  11: "+ num11);
}
```

Now we are ready to make it multigenerational. Here is the final program:

©W. Maddison 1998-2000

```
import java.util.*;

public class MainClass {
    static Random randomNG;
    static int numIndivid=1000;

    public static void main(String args[]) {
        System.out.println("==== H-W eq simulation ====");
        randomNG = new Random();
        int[][] males, females;
        males = new int[numIndivid][2];
        females = new int[numIndivid][2];
        for (int individual=0; individual<numIndivid; individual++) {
            for (int copy=0; copy<2; copy++) {
                males[individual][copy] = 0;
                females[individual][copy] = 1;
            }
        }
        writeGenotypes(males, false);
        writeGenotypes(females, true);

        int[][] sons, daughters;  // make arrays for next generations
        sons = new int[numIndivid][2]; //make just as many sons as fathers
        daughters = new int[numIndivid][2];  //and daughters as mothers

        for (int generation = 1; generation<=10; generation++) {
            // making sons
            for (int son=0; son<numIndivid; son++) {
                sons[son][0] = chooseRandomGamete(males);
                sons[son][1] = chooseRandomGamete(females);
            }
            // making daughters
            for (int daughter=0; daughter<numIndivid; daughter++) {
                daughters[daughter][0] = chooseRandomGamete(males);
                daughters[daughter][1] = chooseRandomGamete(females);
            }
            System.out.println("Generation F" + generation);
            writeGenotypes(sons, false);
            writeGenotypes(daughters, true);
            for (int i = 0; i<numIndivid; i++) {
                males[i][0] = sons[i][0];
                males[i][1] = sons[i][1];
                females[i][0] = daughters[i][0];
                females[i][1] = daughters[i][1];
            }
        }
    }

    static int chooseRandomGamete(int[][] individuals) {
        int randomIndivid = (int)(individuals.length * randomNG.nextFloat());
        int randomCopy = (int)(2 * randomNG.nextFloat());
        return individuals[randomIndivid][randomCopy];
    }

    static void writeGenotypes(int[][] individuals, boolean isFemale) {
        int num00, num01, num11;
        num00=num01=num11=0;   //check out this trick!
        for (int i=0; i<individuals.length; i++){
            if (individuals[i][0] ==0 && individuals[i][1] ==0)
                num00++;
            else if (individuals[i][0] ==1 && individuals[i][1] ==1)
                num11++;
            else
                num01++;
        }
        if (isFemale)
            System.out.println("females 00: " + num00 + "    01: " + num01 + "    11: " + num11);
        else
            System.out.println("  males 00: " + num00 + "    01: " + num01 + "    11: " + num11);
    }
}
```

There it is. One thing to note: see that at the end of each generation the sons and daughters are transfered into the parental arrays so that they can become the next parents.

©W. Maddison 1998-2000

Excercises

6. Write a program to simulate population growth with a bit of noise (random changes in quality of the season). Start with the same logistic program discussed above, but modify it so that the r is variable. Make it so that each generation, r is chosen as a normal variable with mean 2 and variance 0.5. Try changing the variance on the r to see how it affects population fluctuation.

7. Write a program that simulates population genetics in a haploid population. Assume the population size of 100 stays constant each generation. 99 of the gametes will start off with allele 0, 1 with allele 1. Fill each slot in one generation with a gene from an individual chosen randomly from the population before. You can choose the same individual more than once (i.e., an individual can have more than one offspring). Continue generation after generation until allele 1 either goes extinct or goes to fixation. Your goal is to have the program run the simulation 100 or more times so as to record (1) the frequency of allele 1 going extinct, (2) the average number of generations to extinction (among those cases that it goes extinct), and (3) the average number of generations to fixation (among those cases that it goes to fixation). Try it again with population of 20 individuals.

8. Write a program that simulates coalescence in a population of haploid individuals of constant size 100. Start off with the population at time = present. Have each individual choose its parent in the previous generation randomly (every so often, more than one individual will choose the same parent — coalescence!). Then, for the next generation deeper, you need to choose parents only for those that had been previously chosen themselves, and so on, until you are left with only one ancestor. Run the simulation until you get full coalescence. Rerun the simulation 100 or more times so as to record the average number of generations to full coalescence. (Hint: you don't need to store different alleles in this case, but rather only whether an individual had been chosen as a parent.) Find also the average number of generations to full coalescence for a population of 20 individuals.

©W. Maddison 1998-2000

Chapter 6. Recursion and Tree calculation

Objectives

- To use recursive methods
- To learn how to build and use trees

Recursion

You are about to enter the twilight zone. Space will be warped, time will wrap around itself, and nothing will be as it seems.

At least, that's how some people react to recursion. Perhaps you won't react that way, but even if you do, sooner or later it should seem as natural as riding a bicycle. Recursion is a repetitive process in which a method refers to itself. Perhaps the best-known example is a magazine cover that has a picture of a person looking at a magazine that happens to be the same magazine. Thus you can see in his/her copy, a picture of him/her looking at a magazine, on which is a picture of him/her looking at a magazine... and so on. The magazine cover is refering to the magazine cover. As soon as you see it you might fear circular logic, or an infinite process. But the magazine cover doesn't explode in your hands, so it's all perfectly safe. The main thing to realize is that, if well done, there is some rule to stop the recursion, so that it doesn't become infinite. In the magazine case the rule is simple: the artist kept drawing the magazine within the magazine until the resolution of the paintbrush (or whatever) didn't allow any finer details. At that point the recursion stopped. Recursion in a computer is very similar.

Example: Drawing a magazine cover

Imagine that the magazine cover was being drawn by a computer program written in Java. Imagine that there is a method called

```
drawPerson(int x, int y, int width, int height)
```

that draws the person within the rectangle on the screen whose top left coordinate is x,y and with the specified width and height. But suppose that this method doesn't fill in the cover of the magazine that the person is holding; it leaves it blank. We won't specify the contents of this `drawPerson` method; let's just assume that somebody has already written it. It is a simple, non-recursive procedure that merely draws a picture of a person with a blank magazine cover.

Let 's suppose that we know that `drawPerson()` draws the picture so that the top left corner of the blank magazine cover appears exactly in the middle of the specified rectangle, and that the blank magazine cover is 1/4 as wide and as high as the one containing it.

©W. Maddison 1998-2000

Now we want to write a method that uses the drawPerson() method but which draws the whole magazine cover, including the cover within the cover etc.... Here it is, plain and simple:

```
void drawMagazineCover(int x, int y, int width, int height) {
     drawPerson(x, y, width, height);  // first draw person reading a blank magazine

     // now need to calculate rectangle for contained magazine cover
     int containedX = x + width/2;   //half way across width
     int containedY = y + height/2;  //half way down height
     int containedWidth = width/4;    //one quarter as wide
     int containedHeight = height/4;  //one quarter as high

     /* before drawing the contained cover, check that we haven't yet hit limit
          of resolution.  Limit of resolution is hit when width or height of
          rectangle will be 0 pixels because of shrinking by 1/4 and fact that
          pixels use integer arithmetic */
     if (containedWidth != 0 || containedHeight != 0)
          drawMagazineCover(containedX, containedY, containedWidth, containedHeight);
}
```

Of course, something outside of drawMagazineCover has to call it initially to get it started. When it is initially called, the parameters x, y, width and height will refer to the actual, entire magazine cover. However, within the method, the call to drawMagazineCover will be passed the dimensions of the contained drawing of the little magazine cover. Thus the second time drawMagazineCover is called (within the first time), it will receive a x, y, width and height that represents a smaller rectangle. It doesn't care, though. It just does its job.

Now you might worry about drawMagazineCover calling itself. The method is pretty stupid though. It doesn't know it's calling itself; it's not conscious and has no psychological complexes built up over years of being a human. To it, it is simply calling any old method; the method it is calling just happens to be itself.

Example: square roots revisited

Let's revisit the method to calculate a square root of a number greater than 1 (Chapter 2 exercise 8). Recall the technique started with a lower and upper bound on the square root, and took their average. If the average was too big, set the new upper bound to be the average. If the average was too small, set the new lower bound to be the average. Here is a method to do this using loops:

©W. Maddison 1998-2000

```
double squareRoot(double x, double tolerance) {
      double lower = 1.0;
      double upper = x;
      double average = 0;
      while (upper-lower > tolerance) {
            average = (lower + upper)/2.0;
            if (average*average > x)
                  upper = average;
            else
                  lower = average;
      }
      return average;
}
```

Here is the same procedure done recursively:

```
double squareRoot(double x, double lower, double upper, double tolerance) {
      double average = (lower + upper)/2.0;
      double result;
      if (upper-lower <= tolerance)
            result = average;
      else if (average*average > x)
            result = squareRoot(x, lower, average, tolerance);
      else
            result = squareRoot(x, average, upper, tolerance);
      return result;
}
```

It's shorter! Note that you have to start it off a bit differently, since it needs to be passed an initial lower and upper bound.

A dramatization

Let us see that we know just what is happening here. Imagine that methods are workers, and that a method is called by its receiving a message that consists of a work order with the parameters written on the message. You start the process off by sending a message to squareRoot, and you give it the parameters 10, 1, 10, and 0.00001 in order to find the square root of 10.

squareRoot wakes up from his nap at the knock on the door, which he opens. A messenger from FedEx passes him a work order with the numbers 10, 1, 10 and 0.00001 written on them. He sits down at a workspace at his enormous 10 megabyte desk and starts some calculations. He calculates the average to be (1+10)/2 = 5.5. He sees that the upper-lower is 9, much bigger than the tolerance. (5.5*5.5) is bigger than 10, so he sees that he has to do the line:

```
            result = squareRoot(x, lower, average, tolerance);
```

But low and behold, this line calls the method squareRoot. squareRoot, being dim of mind, simply realizes a method must be called. So he writes up a work order for squareRoot with the parameters 10, 1, 5.5, and 0.00001. He phones up FedEx who shows up at his door for a pickup, and he gives the messenger the work order to be delivered to squareRoot, then he goes back to his workspace and makes a note to himself that he is waiting on squareRoot for the answer. The workspace has a unique serial number,

©W. Maddison 1998-2000

which he wrote on the work order before he sent it, so that when the answer comes back he knows which job requested the answer.

He starts to doze off, but a nanosecond later the messenger knocks and hands him a work order (which happened to be the one he himself had just written) with 10, 1, 5.5 and 0.00001 written on it. He's sort of annoyed that he got this new work order while he was still waiting for his other job to get finished, but times are tough and he's lucky to have this job. So he clears off a new workspace on his giant desk (being careful not to disturb the other work in progress), and sits down to do some calculations. He calculates the average to be (1+5.5)/2 = 3.25. He sees that the `upper-lower` is much bigger than the tolerance, and that 3.25*3.25 is bigger than 10, so he sees he needs to get an answer from the method called squareRoot. So he writes up a work order with the numbers 10, 1, 3.25, and 0.00001, calls FedEx, gives the messenger the message, and makes note at his second workspace that he is waiting for an answer from squareRoot.

Well, I think you can see the eventual result. He will keep sending work orders to himself, and thus have a bigger and bigger backlog of works in progress at more and more workspaces at his desk until *presto*! the time will come when `upper-lower` is below the tolerance, in which case he won't need to send a request to squareRoot. Instead, he'll simply write the answer of his calculated average in the "answer" box on the work order, phone up FedEx and return the work order to its sender. He's quite satisfied that he completed that job without having to wait on anyone else, and he brushes the eraser shavings off this last workspace and starts to doze off nicely. But a nanosecond later the pesky FedEx messenger knocks with a completed work order containing the answer that his second last workspace had been waiting on. He thinks "low and behold, that lazy method finally gave me an answer!" He sees from the completed work order that he had requested it from his second last workspace, so he goes there, sees from the note he had written as to exactly where in the instructions he had been. He sees that he has to put the answer into the variable result, which he does. He then goes to the last line of his instructions, which is to return the answer in result. So he takes the work order he had received for the second last workspace, writes the answer on it, and calls FedEx and gives the completed work order to the messenger.

A nanosecond later the the FedEx messenger returns the completed work order, which our exhausted squareRoot worker sees had been requested from his third last workspace. And so it goes, with our worker completing the answer from his third last workspace and returning it, only to find it was the answer he had requested from his fourth last workspace, whose answer what what his fifth last workspace had requested, and so on. His backlog of pending jobs is quickly cleared up until he finally gets the answer

©W. Maddison 1998-2000

he had requested from his first workspace. He returns the result, dusts off the workspace, and falls asleep for a nice, uninterrupted nap.

Recursion is as simple as that.

Exercises

1. Use recursive method instead of a loop to calculate the sum all numbers from 1 to 100. (Hint # 1: the body of the recursive method can be written with only four lines! Hint # 2: you might want the recursion to work backward, starting at 100 and going deeper in the recursion until it arrives at 1.)

2. Let's do something like the magazine cover example. Use the template with frame project that you had used in Chapter 2 exercise 5 (the grid drawing exercise). The goal will be to draw a rectangle within a rectangle within a rectangle and so on (more or less like the magazine cover, but without the person and with only the border on the cover being drawn). You will need to call your recursive method from within the paint method, and you will need to pass to your recursive method the Graphics object (g) so that it can use its methods. The way to draw a rectangle is to call g's drawRect method:

```
        g.drawRect(x, y, width, height);
```

The first two parameters are the left and top of the rectangle; the second two are the rectangle's width and height in pixels.

Trees

Trees in programming are typically done as a series of things (objects in Java; memory structures in C or Pascal) that are the nodes of the tree. Each node points to two or more nodes that are its descendants. In this way you can move up[33] the tree, visiting a node, looking to see who its descendants are, then moving up to them. Once you are there, look to see who are their descendants, and move up to them. Thus, you can move progressively up the tree. As you might guess, this is most easily done using recursion[34].

In the Organism exercise in which you fed organisms and they had babies, a tree of descendants formed naturally, as a natural result of the process of

---

[33]That is, from the root towards the tips. Phylogenetic biologists and paleontologists orient their trees with the root at the bottom, and the tips at the top, like a stratigraphic column, and thus to them "up" is tipward. Large woody vascular plants do likewise. Population geneticists, despite these precedences, often orient their trees with the root at the top, making "up" rootward. I follow the convention root=down, and hope that population geneticists can reorient themselves.

[34]Normally in programming courses a simpler thing is introduced first: a linked list. A linked list consists of a series of objects, each of which has a reference to the next object in the list. Thus you can bounce along the set of objects, at each looking to see which to go to next. A tree is basically the same, except that at each node you can go to more than one node next (i.e. to each of the daughter nodes)

©W. Maddison 1998-2000

reproduction. The same happens with speciation, and your computer can mimic it well.

Let's build a class of objects that will be the nodes of a dichotomous (bifurcating) tree:

```
class Node {
    public Node leftDaughter;
    public Node rightDaughter;
}
```

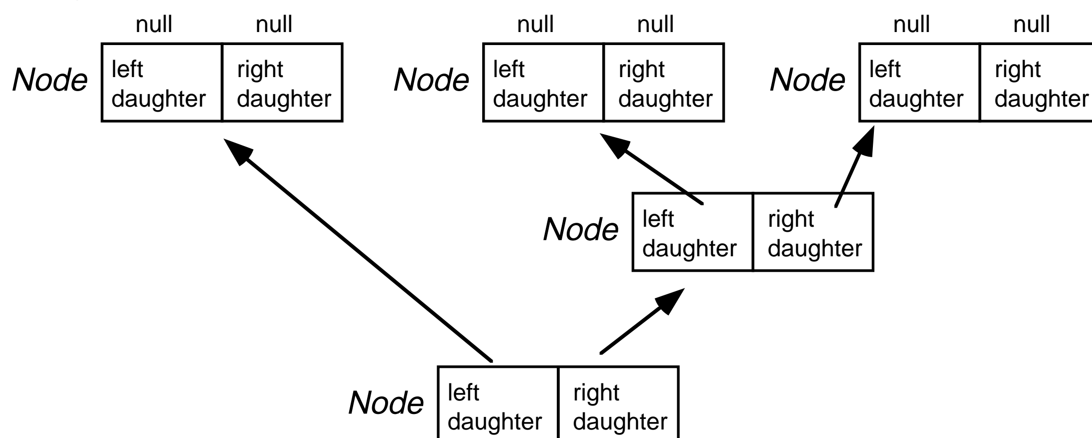
Here in this little example a tree is made:

```
class Node {
    public Node leftDaughter;
    public Node rightDaughter;
}

public class MainClass {

    public static void main(String args[]) {
        Node root = new Node();
        root.leftDaughter = new Node();
        root.rightDaughter = new Node();
        root.rightDaughter.leftDaughter = new Node();
        root.rightDaughter.rightDaughter = new Node();
    }
}
```

Nothing is done with the tree; it is merely made. It is a small three taxon tree with the root having a left and right descendant. The left goes no further, but on the right, there are two grand-daughters of the root. By the time we're done, five Node objects have been created. Here is how they are linked:



One Node object, known to the MainClass as "root", has references to two Node objects stored in its leftDaughter and rightDaughter fields. The first of these Node objects (root.leftDaughter) has null stored in its leftDaughter and rightDaugher fields, because those nodes were never created. The

©W. Maddison 1998-2000

second (root.rightDaugher) has references to nodes stored in its leftDaughter and rightDaughter fields. Those nodes, however, both have null in their fields because their daughter nodes were not created.[35]

Let's add a simple method to the Node class to count the number of terminal taxa in the clade above the Node:

```
class Node {
     public Node leftDaughter;
     public Node rightDaughter;

     public int countTerminalTaxa() {
          if (leftDaughter==null || rightDaughter == null)
               return 1;
          else
               return leftDaughter.countTerminalTaxa() + rightDaughter.countTerminalTaxa();
     }
}
```

Let's examine how countTerminalTaxa works. If the node is terminal, which can be determined by the fact that its fields for storing references to daughter Nodes are null, then it returns the value 1 because that's how many terminal taxa are in a terminal taxon. Otherwise, it adds up the number of terminal taxa in the clades of its two daughter nodes.

Now if this sounds confusing, realize that if you focus just on the local issue at one node, it is clear that the number of terminal taxa in the node's clade is simply the sum of the numbers in its daughter's clades. Typically that's the way recursive methods work. Within the body of the method, you are simply focusing on one node. If the node is an internal node, you assume the recursion will work in the daughter's clades, and you simply use their results to give you what you need at the node you are at.

Reading and writing trees

Now let's add a simple method to construct trees from a descriptive String in parenthesis notatation, and a simple method for writing the trees back to a string. The readClade method of the MainClass uses a treeDescription String to make the tree, and the cladeToString method of the Node class returns a string describing the node's clade.

[35]There are many ways to represent trees in a computer progamming. This is just one: a series of node objects linked to one another. Later, we will see a variant on this that allows polytomous trees. Another way to represent trees is via arrays. For instance, nodes could be represented by numbers, and two integer arrays might store the left and right daughter node numbers of each node. The left daughter of node 5 would be the number stored in element 5 of the left daughter array. By bouncing from one array element to another you are basically following a path up a tree.

©W. Maddison 1998-2000

```
class Node {
      public Node leftDaughter;
      public Node rightDaughter;
      public char name=0;
      public int countTerminalTaxa() {
            if (leftDaughter==null || rightDaughter == null)
                  return 1;
            else
                  return leftDaughter.countTerminalTaxa() + rightDaughter.countTerminalTaxa();
      }

      public String cladeToString() {
            if (leftDaughter==null || rightDaughter == null)  //terminal; return just  name
                  return String.valueOf(name);
            else {  //internal; return parentheses and daughter clades
                  return "(" + leftDaughter.cladeToString() + "," + rightDaughter.cladeToString() + ")";
            }
      }
}
public class MainClass {
      static String treeDescription;
      static int position;
      public static void main(String args[]) {
            Node root = new Node();
            treeDescription = "((A,B),(C,D))";
            position = -1;
            readClade(root);
            System.out.println(root.cladeToString());
      }

      public static void readClade(Node n) {
            position ++;  // go to next character in tree description
            if (treeDescription.charAt(position) == '(') {  //internal node; make daughters
                  n.leftDaughter= new Node();
                  readClade(n.leftDaughter);
                  position++;  // to skip the comma
                  n.rightDaughter= new Node();
                  readClade(n.rightDaughter);
                  position++;  // to skip the right parenthesis
            }
            else {  //terminal node; record name of terminal taxon
                  n.name=treeDescription.charAt(position);
            }
      }
}
```

Exercises

3. Write a method that returns the maximum number of nodes from the root of a clade to a terminal taxon (i.e. the longest path up the clade, counting nodes).

4. Simulated character evolution. Add an integer field to the Node class to represent the character state at the node. In this program, first build a tree of 8 taxa (using the readClade method). Then, character evolution will be simulated as follows. Assign the root node the state 0. Then use a recursive method to move up the tree, assigning a state to a node as follows. Flip a coin that has a 25% chance of coming up heads. If tails, assign the node the same state as its mother (see hint about mothers, below). If head, assign it the state opposite that of its mother (0 if 1, 1 if 0). After the recursive method is done, make a new version of cladeToString called "statesToString" that writes the character state at a terminal node instead of the name of the node.

©W. Maddison 1998-2000

Supplemental stuff: More about trees

Mothers

You may want to be able to find the immediate ancestor of a node (its mother). You can do this if you add a field called "mother" (of type Node) to the Node class and if you record the mother as you go when you create the tree, as follows:

```
public static void readClade(Node n) {
     position ++;  // go to next character in tree description
     if (treeDescription.charAt(position) == '(') {  //internal node; make daughters
          n.leftDaughter= new Node();
          n.leftDaughter.mother = n;  // record who its mother is
          readClade(n.leftDaughter);
          position++;  // to skip the comma
          n.rightDaughter= new Node();
          n.rightDaughter.mother = n;  // record who its mother is
          readClade(n.rightDaughter);
          position++;  // to skip the right parenthesis
     }
     else {  //terminal node; record name of terminal taxon
          n.name=treeDescription.charAt(position);
     }
}
```

Recursion within and outside a Node

In an object-oriented programming language, recursing up a tree can take slightly different forms depending on whether you are doing it from within the Node object, or from outside it. Recall our countTerminalTaxa() method that was in the Node class:

```
    public int countTerminalTaxa() {
        if (isTerminal())
            return 1;
        else
            return leftDaughter.countTerminalTaxa() + rightDaughter.countTerminalTaxa();
    }
```

(note that we're assuming we've added the method isTerminal that returns whether the node is terminal or not). This method could be called for the root's object as follows:

```
      int numTaxa = root.countTerminalTaxa();
```

The method within the root node would ask for its left and right daughters also to perform the method and return their numbers of terminal taxa, while these daughters in turn would ask their daughters, and so on, until the terminal nodes were reached. In a certain way this is not a true recursion, because each node is calling the method belonging to its daughter nodes, not its own method. But it basically behaves just like any recursion.

A method outside of the Node object can also recurse up the tree of nodes, using the linking of nodes as a map for following from node to node. The recursion takes a slightly different form. If we were to rewrite countTerminalTaxa and put it in the MainClass, it might look like this:

```
      static int countTerminals(Node n) {
```

©W. Maddison 1998-2000

```
        if (n.isTerminal())
            return 1;
        else
            return countTerminals(n.leftDaughter) + countTerminals(n.rightDaughter);
    }
```

Note that the method each time it is invoked is passed the node on which it is to work. The number of terminal taxa in the whole tree can be obtained as follows:

```
    int numTaxa = countTerminals(root);
```

This is more like the recursions we saw earlier, in that the method is calling itself, just passing a different parameter.

Passing information into the recursion

Sometimes, all of the information you need in a recursive method can be passed up and down the tree via the parameters passed and via the values returned by the method. The methods countTerminalTaxa and cladeToString do this. This means that all of the information needed is self-contained within the recursion. Other times, it's hard to do that, and a variable at a larger scope needs to be used. Thus, the variable sits outside of the recursion, and each level of the recursion through the tree refers to the same copy of the variable. The method readClade uses such variables: the position variable and the treeDescription variable. To illustrate the difference between these two approaches, let's rewrite the countTerminals method just mentioned above as follows:

```
static int count;

static int countTerminals2(Node n) {
     count = 0;
     count2Recurse(n);
     return count;
}

static void count2Recurse(Node n) {
     if (n.isTerminal())
          count++;
     else {
          count2Recurse(n.leftDaughter);
          count2Recurse(n.rightDaughter);
     }
}
```

Note that a variable count is declared outside of the methods. It is therefore understood at the scope of the class, and all the methods have access to the same variable. Thus, if the variable is changed in one method, the other method feels the change. The number of terminal taxa can be obtained as follows:

```
    int numTaxa = countTerminals2(root);
```

The method countTerminals2 sets count to 0 and then calls the recursive method count2Recurse, which proceeds up the tree, bumping up count each time a terminal node is found. This is a very different system than the other methods we had to count terminal taxa. The other methods were very self contained. They passed information needed up and down through

©W. Maddison 1998-2000

parameters and return statements. The method countTerminals2 is sort of sloppy, in that it uses the variable count that all methods have access to, and hopes (desperately) that noone else will change it, or that it won't change it in a way that causes later problems. Using variables like this is sometimes needed (in this case, it isn't) but it must be done with great caution.

Going up and going down

Sometimes recursing up a tree, you will want to do some calculations at each node as you are visiting it on the way up the tree; other times you will want to do some calculations at each node as you go down the tree. For instance, suppose you have stored at each Node two variables: the number of nodes from that node to the root (pathToRoot), and the number of nodes to the closest terminal (pathToTerminal). Here is a stripped-down Node class that has two methods to calculate these numbers for each Node.

```
class Node {
     public Node leftDaughter=null; //initialize to null
     public Node rightDaughter=null;
     public Node mother=null;
     public char name=0;
     public int pathToTip = 0;
     public int pathToRoot = 0;
     /*-----------------------------------------*/
     public boolean isTerminal() {
         return (leftDaughter==null && rightDaughter == null);
     }
     /*-----------------------------------------*/
     public void calculateShortestPathToTip() {
         if (isTerminal())
             pathToTip = 1;
         else {
             leftDaughter.calculateShortestPathToTip();
             rightDaughter.calculateShortestPathToTip();

             if (leftDaughter.pathToTip>rightDaughter.pathToTip)
                 pathToTip = 1 + leftDaughter.pathToTip;  //add 1 for current node
             else
                 pathToTip = 1 + rightDaughter.pathToTip;
         }
     }
     /*-----------------------------------------*/
     public void calculatePathToRoot() {
         if (mother == null)  // is root
             pathToRoot = 1;
         else
             pathToRoot = 1 + mother.pathToRoot;

         if (!isTerminal()){
             leftDaughter.calculatePathToRoot();
             rightDaughter.calculatePathToRoot();
         }
     }
}
```

The first method, calculateShortestPathToTip(), calculates the shortest path to a terminal (in number of nodes) from the current Node. To calculate this shortest path to a Node, you already need to have calculated the shortest paths for each of the daughter nodes. Thus, the recursion needs to call the

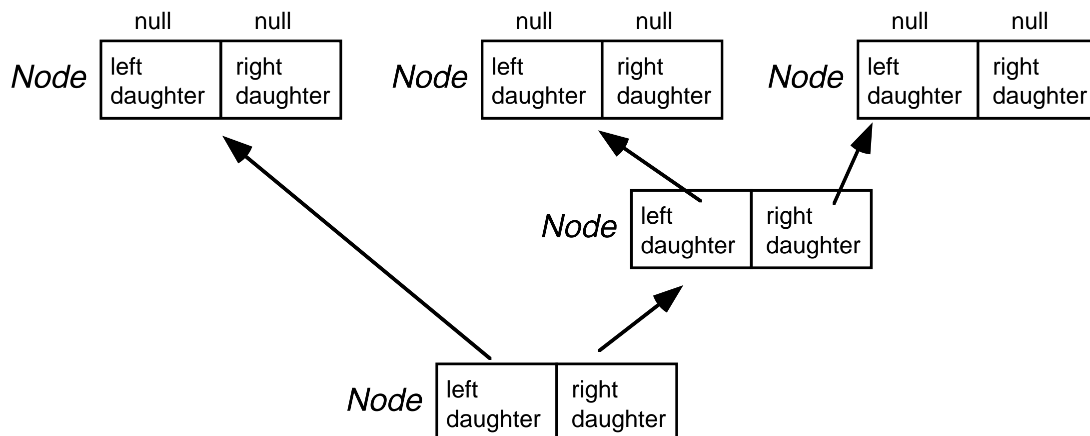
©W. Maddison 1998-2000

method for daughter nodes first before asking which daughter node has the shortest path to a tip. Otherwise, the shortest paths wouldn't yet have been calculated for the daughters. Thus, by putting the calculations of pathToTip after the recursive call to the daughters, these calculations are effectively done on the way <u>down</u> the recursion.

In calculatePathToRoot(), the calculations of pathToRoot are done before the recursive call is made to the daughters. This needs to be done so that the pathToRoot of the current node can be used by the daughters in their calculations. Thus, by putting the calculations before the recursive call, the calculations are effectively done on the way <u>up</u> the recursion. Simulations of character evolution typically work this way, since the character evolves up the tree.

Dichotomies and Polytomies

The trees discussed above are dichotomous; each node has at most two daughter nodes. Here is how five Node objects can contain a reference for a small tree with three terminal taxa:



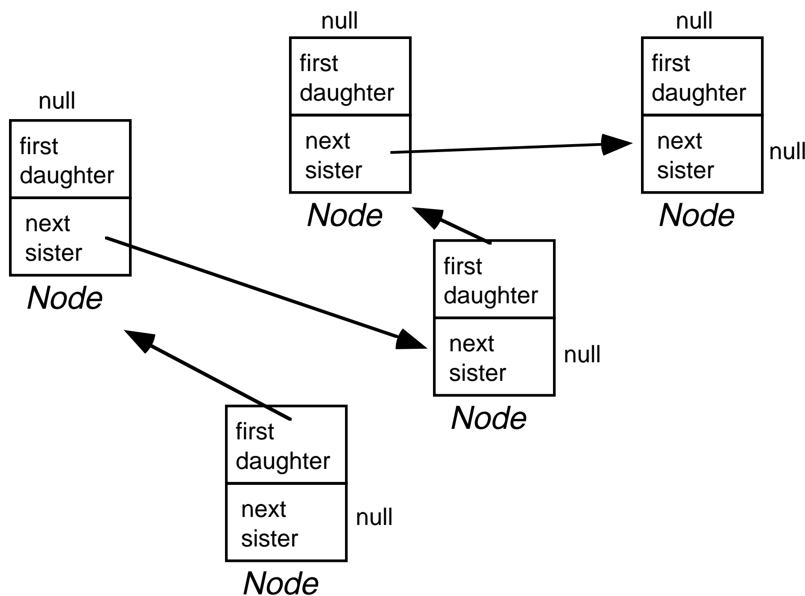
How to handle polytomous trees? One way is to maintain the dichotomous system, but mark some of the nodes as being "ghost" nodes, as not really existing. This in a sense collapses the nodes to generate a polytomy. This system can be used (in fact it's what MacClade 3 uses), but it's a nuisance in some respects. For instance, to find the daughter nodes of a polytomous node, you have to recurse through the ghost region up until you hit non-ghost nodes, which are the daughters.

You might think each Node could have an array of daughters (instead of just a left and right daughters), and just fill up the array with as many nodes as it has daughters. The problem with this is that you either are going to have a maximum number of daughters allowed (the array size) or you have to do fancy things to adjust array size if more daughters are added.

©W. Maddison 1998-2000

The elegant solution to this is a bit strange, but it works great. (If recursion still baffles you, you might want to stop reading at this point.) Each Node stores references to just two other nodes, its first daughter and its next sister. Thus, any given node does not directly store references to all of its daughters. It stores a reference to its first daughter (in a sense, the left-most of its daughters; its firstborn reading the tree left to right). If the node wants to find out all of its daughter nodes, it needs to go up to that daughter, then ask the daughter who its first sister is. Then it asks who that sister's sister is. And so on, bouncing along the sisters (if it's a polytomy there will be more than one) until it encounters a sister with no "next sister". At that point, you've found all the daughters of our original node. The great thing about this system is that you can build polytomies just by adding sisters.

With this storage system for trees, here is how our simple three taxon dichotomous tree would look:



To recurse through such a tree, you need to go up to the first daughter then cycle along through its sisters. Here is a little example program rewritten in the firstDaughter-nextSister style. Note the recursion in countTerminalTaxa and cladeToString.

©W. Maddison 1998-2000

```
class Node {
    public Node firstDaughter=null;
    public Node nextSister=null;
    public Node mother=null;
    public char name=0;
    /*-------------------------------------------*/
    public boolean isTerminal() {
        return (firstDaughter==null);
    }
    /*-------------------------------------------*/
    public int countTerminalTaxa() {
        if (isTerminal())
            return 1;
        else {
            int count = 0;
            for (Node d = firstDaughter; d !=null; d = d.nextSister)
                count += d.countTerminalTaxa();
            return count;
        }
    }
    /*-------------------------------------------*/
    public String cladeToString() {
        if (isTerminal())  //terminal; return just  name
            return String.valueOf(name);
        else {
            String s = "(";
            boolean first = true;
            for (Node d = firstDaughter; d !=null; d = d.nextSister) {
                if (!first)
                    s += ",";
                first = false;
                s += d.cladeToString();
            }
            return s + ")";
        }
    }
}
/*==============================================================*/
public class MainClass {
    static String treeDescription;
    static int position;
    public static void main(String args[]) {
        Node root = new Node(); //root node of tree
        treeDescription = "((A,B,H,I),(C,K,((D,(F,G)),E,J)))";
        position = -1;
        readClade(root);
        System.out.println("Tree is " + root.cladeToString());
        System.out.println("countTerminalTaxa " + root.countTerminalTaxa());
    }
    public static void readClade(Node n) {
        position ++;  // go to next character in tree description
        if (treeDescription.charAt(position) == '(') {  //internal node; make daughters
            n.firstDaughter= new Node();
            n.firstDaughter.mother = n;
            readClade(n.firstDaughter);
            Node daughter = n.firstDaughter;
            position++;
            while (treeDescription.charAt(position) == ',') {
                daughter.nextSister = new Node();
                daughter.nextSister.mother = n;
                readClade(daughter.nextSister);
                daughter = daughter.nextSister;
                position ++;
            }
        }
        else  //terminal node; record name of terminal taxon
            n.name=treeDescription.charAt(position);
    }
}
```

©W. Maddison 1998-2000

Chapter 7. Elements of Style

Objectives

- to learn a few pointers of good programming.

CyberStrunk and CyberWhite

Here's an attempt to give some Elements of Style for programming. Since the last programming course I took was in grade 11, a computer scientist might find them merely amusing. There you have it. They apply to just about any language.

Use mnemonic variable names

You *will* forget what variables were for. Calling a variable "Xy6Zi0" is legal but likely you will not have a clue what is stores when you try to look at your program in 6 months.

Use comments, lots of comments

You *will* forget what methods and classes were designed to do. Add comments to explain methods and classes. Add comments to explain what important blocks are for, especially if it took you a long while to figure out just how to do it. Chances are, if you look at the code in 6 months, it will take you almost as long to figure out why you had written it the first place. You might think that comments are used for other people to understand your code, but actually they are mostly for you.

Visualize blocks and flow

Before you start writing, try to get a clear picture in your mind of the major blocks in the program, and the flow through them. And even once you have started to write the program, you should every so often sit back, stare at a fluourescent light fixture, and try to put into your mind a landscape picture of the overall structure and dynamics of your program.

All names must be defined

Every single symbol, name, etc. in a program has to be defined (except for literal text written in the contents of Strings, and so forth). Either you have to define it, or it has to be pre-defined by the language or some library you are using. If your compiler tells you something is undefined, it means either that it doesn't exist even though you had been hoping it did, or you misspelled it, or you forgot to tell the compiler were to look for it (you forgot to include or refer to some library, such as with the import statement in Java).

Check loop boundaries

Most problems with loops, at least for me, happen because I haven't thought hard about either the first or last time through the loop. The problem might be that you don't have the conditions for exiting the loop

©W. Maddison 1998-2000

correct (did you mean < or <= ?). The problem might be that you forgot that the first time through something wouldn't yet have a value. Think hard about loop boundaries (start and end), and perhaps if things seem to be going wrong, do some checks to see that values are the way they are supposed to be the first and last time through the loop.

Try small cases first

If your program is designed to run a loop from 1 to a trillion, start off writing the program so it only goes from 1 to 10. That way you won't have to wait forever to see that it is basically working. Begin with small, simple cases first, make sure they are working, then build to complexity.

Try cases whose answer is known in advance

To begin with, run simple cases whose answer is known to you. Try asking for the cube root of 27, not 15.318. Try flipping a 50-50 coin, not a fancy multinomial set of dice. That way, you can get the kinks out before getting more complex.

Design one step at a time

Besides trying small and known cases first, solve problems one at a time. If you want a program that has a loop, and each time through the loop it calls one fancy method, then another fancy method, then combines their results, don't try to write the whole program all at once. Write it with the loop and two simple methods that don't really do what you want, but which are simple, and which produce results that you can still combine. Once you get that working, then fix the one method so it does what you want, and get it working. Then fix the other method, and get it working.

Try programming in English (or French, or...) first

Sometimes it's best to scribble lines of English as you are initially designing your program. These can serve as placeholders, reminding you what code you need to write there. Of course, the code might not compile until you clean out the English. For instance, if you wanted to add up numbers from 1 to 100 you might start out writing:

```
start out a variable to containing a running sum
go through all numbers from 1 to 100
     for each, add it to the running sum
Now write out the final sum
```

Debug one step at a time

If something goes wrong and you think it might be due to this line or that line, fix one at a time. If 5 things go wrong, fix them one at a time. That way, in case there is an interaction between the problems, you can track it down. Debugging is a process of scientific experimentation. Use the same logic you would in tracking down what factors are affecting the behavior of your biological system. Try alternative conditions that you expect could

©W. Maddison 1998-2000

distinguish between alternative theories about what's causing the problem. Try to isolate problems one at a time.

Return impossible results in case of error

Many times you will have a series of statements that you expect must find some useful result. For instance, perhaps you know that at least one element in the array is supposed to have value "1", and you have a method that returns which element has that value. If by some strange twist of fate you get through the entire array and don't find a "1", have your method return some impossible result (like -1). That way, the impossible result will serve as a warning that something went wrong, and you will be more likely to find what went wrong in the array. Assigning an impossible result might cause an error message to be generated, but that's OK. It's better for that to happen than for some apparently correct but in fact misleading result to be returned. In a similar spirit, return the value null if an object was not found or successfully created.

Initialize variables

Initialize variables when they are created to values that make sense. What makes sense might be 0 (to start out a sum, or as a default value) or null. Or, it might be some "impossible" value (like -1 for a variable that is intended to element number of an array), which you expect will be changed but which in case of error might not be (see comment on returning impossible results).

Use self-contained methods where possible

Try to arrange, as much as possible, that methods use only the information passed to them as parameters. Avoid using variables belonging to the object/class except were necessary. It often will be necessary (especially where the purpose of the method is to adjust an object's variable!), but undisciplined programmers go overboard in making variables at too large a scope (e.g. for the object) then use them all over everywhere. The more you use variables belonging to the object within methods, the more you can have wierd effects happening where different methods are fighting to adjust the values of the variables. If methods are self-contained, then they will be easier to debug and modify.

Compiler errors might not make sense

There are two sorts of errors: compiler errors, and run-time errors. The latter occur when you try to run your program. The former occur as it is compiling, and typically if they occur, you can't even attempt to run the program. It is very difficult to write a compiler so that it explains the errors it has found in some code. To give the appropriate error, the compiler has to guess what your really should have written, which may be difficult to do if you've done a good job of writing something confusing. For instance, what is wrong with this sentence:

©W. Maddison 1998-2000

```
"I he, and, she said, is happy".
```

To describe what is wrong, you are implying how to fix it, but there are many ways you might fix it. You could throw in some quotation marks, move some commas, change the conjugation of the verb, delete a word or two. Different combinations of these might all give grammatically correct sentences, but in each case the sentence may be different and have a different meaning. The compiler can't guess which meaning you had intended, so it simply assumes something about what you intended, which may not be correct, and the error message may seem nonsensical to you.

©W. Maddison 1998-2000