CSE 120/130

Introduction to

Programming Languages and Techniques

Fall 2000

Handout 6

Lots of information on Java (including links to Sun's Java pages) can be found at:

`http://www.apl.jhu.edu/~hall/java/`

What we've seen so far in this course

- Basic techniques for "programming in the small"
- Several important algorithms (sorting and others)
- Fundamental concepts:
  - abstraction ("write things only once")
  - recursion ("solve a problem by solving similar, simpler, subproblems")
  - generic programming / polymorphism
  - invariants ("properties of a data structure that are preserved by the operations that manipulate it")
  - organizing and hiding information in programs
- Some common data structures: lists, trees, etc.
- A significant part of one particular programming language (OCaml)
- A careful, formal definition of how the meaning of a program in this language can be calculated

Where we're going

- Larger-scale software engineering issues ("programming in the large")

  There will be much more about this in 121
- Linguistic features supporting large-scale programming

  objects, classes, subtyping, ...
- Core features of the Java language
- A better understanding of algorithms and efficiency (mostly in 121)

Java (at last!)

- a modern object-oriented programming language
- based on many previous languages, especially C++, Mesa, and Modula-3
- superficially similar to C++, but omitting many of its complex and/or dangerous features and adding a few new ones
- emphasizes (like ML)
  - safety
    - static typing
    - garbage collection
  - portability (of both whole programs and tiny "applets" used to enhance web pages)

A tiny Java program:

```
public class test1 {
  public static void main (String[] args) {
    int x = 2;
    int y = x + 3;
    System.out.println (y + 4);
  }
}
```

The equivalent program in OCaml:

```
let x = 2 in
let = x + 3 in
  y + 4
```

Observe:

- The main goal of an ML program is to calculate some value
- The main goal of a Java program is to cause some effect

The java program also includes some mysterious incantations that simply have to be there. We'll explain this stuff as we go along, but for now just consider it as "magic."

Running java programs

To run our java program, we create a file `test1.java` containing the text

```
public class test1 {
  public static void main (String[] args) {
    int x = 2;
    int y = x + 3;
    System.out.println (y + 4);
  }
}
```

(Note that the file name is the same as the class name)

We then compile it with the command

```
/home/bcpierce/CIS/120/JAVA> javac test1.java
```

This create a file `test1.class`, which we run with the command

```
/home/bcpierce/CIS/120/JAVA> java test1
9
```

The Java compiler vs. ML's top-level command loop

Running a Java program is more tedious than interacting with ML's top level. At least this is how it appears for small programs.

However, once programs get large, we put them in files in any case, and we organize them into classes and structures (OCaml) or classes (Java), so the difference between the two languages starts to even out. We did not cover structures and classes in OCaml.

The only problem is that Java makes us do all this on day one!

Basic types and operators

	OCaml	Java
Built-in types	`int` (`123`, `-99`)	`int` (`123`, `-99`) (also `byte`, `short`, `long`)
	`float`	`float` (also `double`)
	`bool`	`boolean`
	`string` (`"ccccc\n"`)	`String` (`"ccccc\n"`)
	`char` (`'c'`)	`char` (`'c'`)
Arithmetic operators	`+`, `-`, `*`, `/`, `+.`, `-.`, `*.`, `/.`	`+`, `-`, `*`, `/` (overloaded)
	`mod`	`%`
Comparisons	`>`, `>=`, `<`, `<=`	`>`, `>=`, `<`, `<=`
	`=`, `<>`	`==`, `!=`
Boolean operators	`&&`, `||`, `not`	`&&`, `||`, `!`

Functions in Java

A basic OCaml function definition:

```
let rec FUNCTIONNAME (ARGNAME : ARGTYPE) = EXP
```

A basic Java function definition:

```
public static RESULTTYPE FUNCTIONNAME (ARGTYPE ARGNAME) {
  return EXP;
}
```

(No distinction between recursive and non-recursive functions in Java)

Java's terminology for functions is *methods*.

Points to note:

- Java is explicitly typed (no type inference)
- The result type of a function is always indicated explicitly

For example, here is the factorial function in Java:

```
public class test {
  public static int fact (int x) {
    return (x==0 ? 1 : x*fact(x-1));
  }

  public static void main (String[] args) {
    System.out.println(fact(5));
  }
}
```

Points to note:

- Different syntax for equality test (`==`, not `=`)
- Different syntax for conditional (`e1 ? e2 : e3`, not `if e1 then e2 else e3`)
- The `return` keyword
- The printing function `System.out.println`

For another example, recall (from Handout 2) Euclid's algorithm for calculating the greatest common divisor of two numbers.

In ML:

```
let rec gcd (x:int) (y:int) =
    if x = y then y
    else if x > y then gcd(x-y, y)
         else gcd(x, y-x)
```

In Java:

```
public static int gcd (int x, int y) {
  return (x==y
          ? y
          : (x>y ? gcd(x-y,y) : gcd(x,y-x)));
}
```


Comments

Comments in Java can be indicated in two ways:

- anything bracketed with `/*` and `*/`
- end of line prefixed with `//`

```
/* The following method defines the GCD
   function recursively using Euclid's algorithm */
public static int gcd (int x, int y) {
  return (x==y    // Are we done?
          ? x     // Yes
          : (x>y ?        // No: continue with one of
             gcd(x-y,y) :    // (x-y,y)
             gcd(x,y-x)));   // (x,y-x)
}
```

Local variables

In OCaml:

```
let rec f x  =
  let y = x*x in
    y*x + 3*y +3*x + 1
```

In Java:

```
public static int f(int x) {
  int y = x*x;
  return (y*x + 3*y + 3*x + 1);
}
```


Expressions vs. commands

Java is a “statement oriented” language, while ML uses an “expression oriented” syntax:

- Not only a whole ML program, but *every part* of the program, an an expression whose role is to calculate a value.
- Not only a whole Java program, but also *some parts* of it, are commands whose role is to cause some effect.

Assignment

The simplest and most characteristic command is the *assignment statement*.

```
public static int f(int x) {
  int y = x * x;
  y = y + x;
  return (y * x + 3 * y +3 * x + 1);
```

The statement "`y = y+x;`" *changes the value* of `y` by adding `x` to it. We never used an operation like this in ML (though ML does support a similar operation).

Note the confusing use of `=`. It is used for assignment in Java (and C and C++), and *not* as an equality predicate. Assignment is written `:=` in Pascal.

Watch out for the "=" confusion. Even experienced C/Java programmers make it!

"Functional" vs. "imperative" programming styles

Advantages of functional style:

- often clearer — more *declarative* presentation of algorithms
- no "side-effects" $\longrightarrow$ easier to read, maintain, and reason about programs

Advantages of imperative style:

- more efficient (sometimes!): modify data structures in-place instead of rebuilding them
- sometimes clearer — avoids passing too many parameters to functions that don't need them
- the machines we run our programs on operate in an imperative fashion, so we can get "closer to" the machine

  (This doesn't really apply to Java, but it does to lower-level imperative languages like C)

Both styles are important in practical programming.

Other kinds of imperative statements

Many useful functions do not return useful information. For example, calling `System.out.println` causes something to happen on your screen. It doesn't return useful infomation.

Another useful imperative form is the `if` statement.

```
public static void f(int x) {
  if (x > 5) {
    System.out.println(x); }
}
```

The general form is `if (TEST) { STATEMENTS }`. The block of statements gets evaluated only if the expression evaluates to `true`. Otherwise nothing happens.

Note that the *result type* of the function `f` is `void`. This is the "nothing" type (like `unit` in OCaml). So a call to this function, such as

```
f(4);
```

is itself a statement.

An `if` statement may have an `ELSE` branch

```
public static void f(int x) {
  if (x > 5) {
    System.out.println(x); }
  else {
    System.out.println(0); }
}
```

but this is optional.

Also, if the block of `STATEMENTS` (in either the `THEN` or the `ELSE` branch) contains just a single statement, Java allows us to drop the enclosing braces:

```
public static void f(int x) {
  if (x > 5) System.out.println(x);
  else       System.out.println(0);
}
```

In this class, though, we will always keep the braces, for consistency.

Sequencing

Along with the idea of a command is the idea that we program by giving *sequences* of commands.

```
public class test2 {
  public static void f(int x) {
    x = x + 5;
    if (x < 10) { x = x*x; }
    if (x < 30) { System.out.println(x); }
  }

  public static void main (String[] args) {
    f(1); f(9); f(10);
  }
}
```

Note that each command in `f` changes the "state" of the world (in this case `x`) and in doing so communicates with the next command.

Try to predict the behavior of this program.

Iteration

There are various ways of causing a statement to be repeatedly executed in Java. The most basic is the `while` loop:

```
while (TEST) { STATEMENTS }
```

Example:

```
while (x < 100) { x = x*x; }
```

Notice that `TEST` is typically a boolean expression involving some variables and, in order for the `while` statement to terminate, `STATEMENTS` must change one or more of those variables.

The whole `while` statement is — as its name implies — a statement.

```
public class test3 {
  public static void f(int x) {
    while (x < 100) {
      x = x*x; }
    System.out.println(x);
  }

  public static void main (String[] args) {
    f(2); f(3); f(5); f(100);
  }
}
```

This program produces the output...

```
256
6561
625
100
```


Factorial Again!

```
public static int fact(int n) {
  int i = 1;
  int accum = 1;
  while (i <= n) {
    accum = accum*i;
    i = i+1; }
  return accum;
}
```

Here is the same GCD algorithm as we saw a few slides ago, but this time written *iteratively* (using assignment and `while`, rather than recursion):

```
public static int gcd (int x, int y) {
  while (x!=y) {
    if (x>y) {
      x = x-y; }
    else {
      y = y-x; }}
  return x;
}
```


Arrays

For each type `eltType` in Java, there is a type `eltType[]` of *arrays* whose elements are of type `eltType`.

If `a` is an array, we can obtain the `i`th element of `a` with `a[i]`.

We can change the `i`th element of `a` with `a[i] = ...`

Arrays are created by the special syntax `new eltType[iexp]` where `iexp` is an integer expression.

```
int[] a = new int[10];       // create an array of 10 ints
a[5] = 7;                    // put 7 in the 5th position
a[6] = a[5] + 6;             // put 13 in the 6th position
System.out.println(a[6]);    // print value in 6th position
```

Note that type of an array in Java includes the type of its elements, Arrays are the *only* "parametric types" in Java.

Arrays vs. Lists

Both arrays and lists are used to store ordered sequences of elements.

	Arrays	OCaml Lists
type name	`int[]`	`int list`
lookup of first element	`a[0]`	`hd l`
lookup of `n`th element	`a[n-1]`	`nth n l`
update of `n`th element	`a[n-1] = ...`	—
extension	—	`x :: l`
shortening	—	`tl l`

Note that Java arrays are *zero-indexed*. The "first" element is `a[0]`.

The main advantage of arrays is *efficiency*:

- Accessing the `n`th element of a list takes `n` steps
- Accessing (or updating) the `n`th element of an array takes `1` step

The main advantage of lists is *extensibility*:

- Lists can be extended or shortened (with `::`, `tl`, etc.)
- Arrays, once built, cannot be extended or shortened

(Java provides a built-in class `Vector` that combines some of the features of arrays and lists. We will not cover it here.)

The length of an array

We find the length of an array `a` with the expression `a.length`. Why this special syntax? Why not have `length` be a function as in Ocaml and write `length(a)`?

It turns out this is not special syntax. What we are seeing is an example of a rather general syntax for "communicating" with an object. In this case the array `a` is the object and `length` is a piece of data (it's called a *field* or *data member*) associated with `a`.

We'll learn more about objects, data members and methods when we have finished our review of arrays.

For the time being, just take `a.length` as special syntax.

For example, here is a method that sets all the entries in an array to one.

```
public static void setToOnes (int[] a) {
  int i = 0;
  while (i < a.length) {
    a[i] =1;
    i = i+1; }
}
```

(Since arrays are zero-indexed, the last element is `a[a.length-1]`.)

Some demystification

Our “top level” or “calling” program has the header

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

indicating that it is a function with a parameter `args` that is an array of strings. What is in the array is the “command line” arguments. This provides a sometimes useful method of feeding data to your Java program.


```
public class MainTest{
  public static void main (String[] args) {
    int i = 0;
    while (i < args.length) {
      System.out.println(args[i]);
      i = i+1;}
  }
}

/home/peter/120/java>java MainTest one two three ... testing
one
two
three
...
testing
/home/peter/120/java>
```

Array initialization

An other method of constructing and initializing short arrays is to use the syntax $\{e_1, e_2, \ldots e_n\}$.

Examples:

```
String[] french = {"zero", "un", "deux", "trois"};
int[] aa = {1, 17, 3*3 + 4*4, 12-89};
```

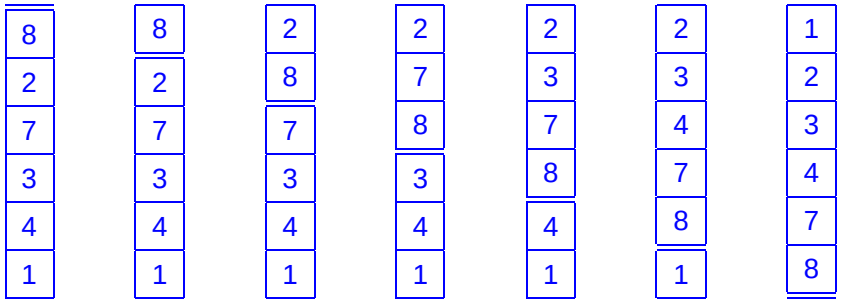
This is shorthand for:

```
String[] french = new String[4];
french[0] = "zero";   french[1] = "un";
french[2] = "deux";   french[3] = "trois";

int[] aa = new int[4];
aa[0] = 1;            aa[1] = 17;
aa[2] = 3*3 + 4*4;    aa[4] = 12-89;
```


Sorting arrays – bubblesort

Imagine an array as a column of “liquid” elements. Initially the column is frozen. We then unfreeze the column from the top, allowing elements to “float” up to their appropriate positions. We assume that elements can be compared for “weight” (larger elements are heavier).



We’ll think of the arrays as running “downwards”. Position 0 is at the top.

```
public static void bubblesort(int[] a) {
  int i = 0;  // i is the index of the first element in the frozen part
  while (i < a.length) {
    int x = a[i];      // x is the value to be "bubbled up"
    int k = i-1;
    while (k >= 0 && x <= a[k]) {
      a[k+1] = a[k];   // move a[k] down to make room
      k = k-1; }
    a[k+1] = x;
    i = i+1; }
}
```



```
public class BubblesortTest{
  public static void bubblesort(int[] a) {
    // ...as before...
  }

  public static void main (String[] args) {
    int[] a = {8,2,7,3,4,1};
    int i = 0;
    bubblesort(a);
    while (i< a.length) {
      System.out.print(" " + i + ":" + a[i]);
      i = i+1; }
    System.out.println();
  }
}
```

The program prints `0:1 1:2 2:3 3:4 4:7 5:8`

Some comments on sorting.

- Our sorting function works *in place*. It doesn't require another array, and doesn't create any auxiliary structures or garbage in the process. Compare this with the sorting functions we wrote on lists, where we made liberal use of `::` to create new lists. Writing in place sorting functions is often important to scientists who want to program with the biggest arrays that they can fit into main memory.
- Bubblesort is not as efficient as the mergesort or quicksort algorithms that we saw (for lists) in ML. It may require (when?) approximately $n^2/2$ iterations of the inner loop (where n is the length of the array). Constructing an efficient in-place algorithm is something of a challenge.
- As you can see, getting the arithmetic and the loop termination conditions can be tricky. Of course, you should always use your brain to be sure that your "logic" is correct, but in practice it never hurts to try out your code on pathological examples: the empty array (length 0), the array whose elements are all equal, etc.

Searching

We often want to find where in an array a given element occurs. The simplest way to do this is to iterate over the array.

The following function returns the first position in `a` at which `x` occurs, or -1 if `x` does not occur at all in `a`.

```
public static int search (int[] a, int x) {
  int i = 0;
  while (i < a.length) {
    if (x == a[i]) { return (i); }
    i = i+1; }
  return -1;
}
```

Binary Search

If the array is sorted, we can do much better than the previous method. We look at the element in the middle of the array. If it is equal we have the answer, if it is greater than the given element we look in the left half of the array, and if it is less we look in the right half.


```
public static int binSearch (int[] a, int x) {
  return(searchAux(a, x, 0, a.length));
}

// look for x between lo and hi-1
public static int searchAux(int[] a, int x, int lo, int hi) {
  int mid;
  if (lo >= hi) { return(-1); }
  mid = (lo+hi-1)/2;
  if (a[mid] == x) { return(mid); }
  else if (x < a[mid]) { return(searchAux(a,x,lo,mid)); }
  else { return(searchAux(a,x,mid+1, hi)); }
}
```

(Does this version always return the index of the *first* occurrence of `x` in `a`?)

Notes on binary search:

- Our algorithm terminates when `hi` becomes equal to or less than `lo`. Is this guaranteed to happen eventually?
- The function `searchAux` is called $n \log_2 n$ times (where n = length of array).
- We could generalize this function to perform dictionary-style lookups.
- How does this approach compare with binary search *trees*?

Some extra syntax

It's time to introduce some extra syntax – inherited from C. It is inessential, but appears so often that we should explain it now.

- `i++`. This adds 1 to `i`. In this sense it is equivalent to `i = i+1`.

  However `i++` can also be used as an expression: it returns the current value of `i` but increments `i` in the process.

  You'll sometimes see tasteless programs like

  ```
  while (i++ > 0) {...}
  ```

  Please don't write things like this — conserve neurons, not keystrokes!
- `i--`. Analogous to `i++`.
- "For" loops.

  $$\texttt{for}(c_1; e; c_2)c_3$$

  in which c_1, c_2, and c_3 are statements and e is a boolean expression. This is equivalent to

  $$c_1;\ \texttt{while}\ (e)\{c_3; c_2\}$$

More on for loops

The for loop

```
int i;
for (i = 1; i < 10; i++) {
  System.out.print(i + " "); }
```

is equivalent to

```
while (i < 10) {
  System.out.print(i + " ");
  i++;       // or i = i+1
}
```

(You'll sometimes see things like

```
for (int i = 1; i < 10; i++) {...}
```

in which the variable `i` is introduced inside the loop header. The translation still works.)

Implementing quicksort using arrays

Recall how quicksort worked on lists. We started by choosing an element `x` at random (e.g., by taking the first element) and then partitioned our list into those elements that are less than `x`, those that are equal to `x`, and those that are greater. Repeating this process recursively on the partitions and appending the results led eventually to a completely sorted list.

Can we do this in place?

To begin with, we need to figure out how to partition an array into those elements that are less than or equal to `x` and those that are greater.

We place a marker at each end of the array. Suppose the chosen element is 4.

3*	9	7	4	1	8	2	4	5	6*

While the left-hand marker is on an element less than or equal to 4, move it to the right. Similarly, while the right-hand marker is on an element greater than 4, move it to the left.

3	9*	7	4	1	8	2	4*	5	6

Now interchange the marked values and move each of the markers in by one.

3	4	7*	4	1	8	2*	9	5	6

Repeat until the markers collide.

3	4	2	4*	1	8*	7	9	5	6

3	4	2	4	1*	8*	7	9	5	6

This gives us our partition:

3	4	2	4	1 ‖	8	7	9	5	6

Now we use an almost identical process to split the elements in the left partition into two sub-partitions: those less than x, and those equal to x.

3*	4	2	4	1* ‖	8	7	9	5	6

3	4*	2	4	1* ‖	8	7	9	5	6

3	1	2*	4*	4 ‖	8	7	9	5	6

3	1	2 ‖	4	4 ‖	8	7	9	5	6

Now to implement quicksort, we recursively quicksort the left and right partitions. Note that this means we want, in general, to apply our partitioning algorithms to a *segment* of the array. The tricky part is getting the details right.

Just for variation, we'll assume that we now want to sort arrays of real numbers

Since the details are tricky (especially the termination conditions) we'll try to structure the code so that it is "provably" correct. Formally proving a program correct is a big topic and well beyond the scope of CSE120. However you might be interested in seeing some of the ingredients of a proof. Even an informal proof is a great help in understanding the behavior of a program. We'll show this at the end of the program.

Our first function is `PartitionG`, which rearanges the array so that everything *greater* than some value is to the right and everything *less than or equal* is to the left.

Look at the code for partitionG. It has various assertions attached to the code. From these assertions one can deduce that the program does what it is supposed to do. In particular, there is an important assertion attached to a point in the while loop. This assertion is always true, no matter which iteration of the while loop is executed. This is called a loop invariant and is crucial in proving correctness of the program.


```
class QuickSortTest{

    // Partition a[lo] .. a[hi-1] so that a[lo]..a[mid-1]
    // are all <= x and a[mid]..a[hi-1] are all > x.
    // Return mid.
    static int partitionG(double[] a, double x, int lo, int hi) {
        double temp;
        int ll = lo-1; int hh = hi;
        while (ll <  hh-1) {
            if  (a[ll+1] <= x) {ll++;}
            else if (a[hh-1] > x) {hh--;}
            else {
                ll++; hh--;
                temp = a[ll]; a[ll] = a[hh]; a[hh]=temp;
            }
        }
        return (hh);
    }
```

```
    // Same as partitionG, but now a[lo]..a[mid-1] are all < x
    // and a[mid]..a[hi-1] are all >= x
    static int partitionGE(double[] a, double x, int lo, int hi) {
        double temp;
        int ll = lo-1; int hh = hi;
        while (ll < hh-1) {
            if (a[ll+1] < x) {ll++;}
            else if (a[hh-1] >= x) {hh--;}
            else {
                ll++; hh--;
                temp = a[ll]; a[ll] = a[hh]; a[hh]=temp;
            }
        }
        return (hh);
    }
```



```
    //  sort array a between lo and hi-1
    static void quickAux(double[] a, int lo, int hi) {
        if (lo >= hi - 1) return;
        double x = a[lo];        //choose an element from a "at random"
        int midhi = partitionG(a,x,lo,hi);
        int midlo = partitionGE(a,x,lo,midhi);
    }

    public static void quicksort(double [] a) {
        quickAux(a,0,a.length);
    }

}  //end of class QuickSortTest
```

Here is the code for `PartitionG` annotated with *assertions* these are statements that are true at the point at which they are made. A *loop invariant* is true no matter which iteration we are in.


```
static int partitionG(double[] a, double x, int lo, int hi) {
   // Assumption: the method is called with 0 <= lo <= hi <= a.length
    double temp;
    int ll = lo-1; int hh = hi;
    while (ll <  hh-1) {
if  (a[ll+1] <= x) {ll++;}
else if (a[hh-1] > x) {hh--;}
else {                          // We know a[ll+1] > x and a[hh-1] < x
    ll++; hh--;                 // So the elements can be interchanged
    temp = a[ll]; a[ll] = a[hh]; a[hh]=temp;
}
    // (loop invariant) At this point we know:
    //    (a) Elements a[lo] ... a[ll] are all <= x
    //    (b) Elements a[hh] ... a[hi-1] are all > x
    //    (c) ll < hh .   Why?
    }
    // At this point we know ll = hh-1 .  Why?
    return (hh);
}
```

In the loop invariant for `partitionG we asserted that \verb`ll ¡ hh—. To prove this takes a little thought. At the start we know that `ll < hh-1`. Only one of the three branches of the conditionals is executed. The first two change `ll` or `hh` by 1, so if either of these is executed `ll < hh` will still be true. What about the third branch? This increments `ll` and decrements `hh`, so we would violate `ll < hh` if, at the start of the loop we had `ll = hh+2`. Suppose this were the case. The third branch is executed only when `a[ll+1] > x` and `a[hh-1] <= x`. But now `a[ll+1] = a[hh-1]` so in this case we would not have executed the third branch.

Using arrays to implement strings

The Java class `String` is probably implemented using arrays. Alternative implementations (e.g. lists) are certainly possible.

You will find the class specification in `java.lang`. The method types and descriptions given here are taken from the documentation for that class. Among the constructors for this class is

- `public String(char value[])`

  Allocates a new String so that it represents the sequence of characters currently contained in the character array argument.

We can start to build our own string class:

```
class MyString{
  private char [] data;

  public MyString(char value[]) {
    int i;
    data = new char[value.length];
    for (i=0; i< value.length; i++) { data[i] = value[i]; }
  }
  ...
}
```

This allows us to construct strings as follows

```
char[] catarray = {'c', 'a', 't'};
MyString catstring = new MyString(catarray);
```


There are numerous methods listed for the `String` class. Two important ones are:

- `public int length()`

  Returns the length of this string. The length is equal to the number of characters in the string.

- `public char charAt(int index)`

  Returns the character at the specified index. An index ranges from 0 to length() - 1.

  Parameters: index - the index of the character. Returns: the character at the specified index of this string. The first character is at index 0.

```
class MyString{
  private char[] data;
  public int length() { return(data.length); }
  public char charAt(int i) { return (data[i]); }
  ...
}
```

Another method is

- `public String concat(String str)`

  Concatenates the specified string to the end of this string.

  If the length of the argument string is 0, then this object is returned.

  Parameters: str - the String that is concatenated to the end of this String.

  Returns: a string that represents the concatenation of this object's characters followed by the string argument's characters.

Here is the code for `concat`

```
public MyString concat(MyString s) {
  int i;
  int j;
  char [] temp;
  if (s.length() == 0) {return(this);}
  // Create array for result
  temp = new char[data.length+ s.length()];
  // Copy from data
  for (i=0; i < data.length; i++) {temp[i]=data[i];}
  // Copy from s
  for (j=0; j < s.length(); j++) {temp[i]=s.charAt(j); i++;}
  //  Create the MyString
  return new MyString(temp);
}
```

Yet another method provides lexicographic ordering:

- `public int compareTo(String anotherString)`

  Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings.

  Parameters: anotherString - the String to be compared.

  Returns: the value 0 if the argument string is equal to this string; a value less than 0 if this string is lexicographically less than the string argument; and a value greater than 0 if this string is lexicographically greater than the string argument.

Why this bizarre method of describing the comparison result? A hangover from C.

Incidentally, C programmers will note that arrays in Java have some similarity to arrays in C, but the programming details are rather different.


```
public int compareTo(MyString s) {
  int i = 0;
  int j = 0;
  while(i < data.length && j < s.length() && data[i] == s.charAt(j)) {
    i++;
    j++; }
  if (i == data.length && j==s.length()) { return(0); }
  if (i == data.length) { return(1); }
  if (j == s.length()) { return(-1); }
  return(s.charAt(i) - data[i]);  // The C hangover
}
```