CSE 120/130

Introduction to

Programming Languages and Techniques

Fall 2000

Handout 7

A closer look at array parameters

What does the following Java program print?

```
public class test {
  public static void f(int[] x) {
    x[1] = 5;
  }

  public static void main (String[] args) {
    int[] a = {0,1,2,3};
    f(a);
    System.out.println(a[1]);
  }
}
```

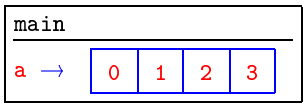
If we think carefully about it, this is a little surprising.

The rule we have seen for evaluating an application `f(a)` goes like this:

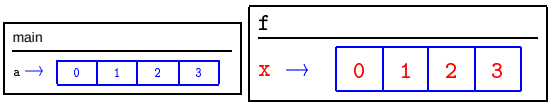
1. Evaluate `a` to produce a value `v`
2. Look up the function (in ML) or method (in Java) `f`, to find
   - its body, `b`
   - the name of its parameter
   - its definition environment
3. Extend the definition environment with bindings of the parameter variable to the argument value
4. execute the body `b`

So, going back to the old "pieces of paper" method, we might picture the sequence of evaluation steps like this:

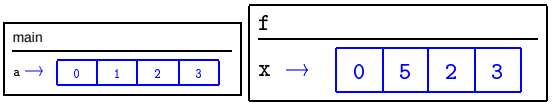
1. Bind `a` to a 4-element array:



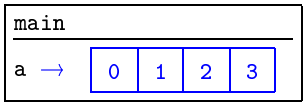
2. To execute `f(a)`, create a new piece of paper for the call to `f` and write down the binding of `x` to the array.



3. Execute the assignment `x[1] = 5`.



4. Since `f` is now finished, throw away its piece of paper and continue executing `main`:

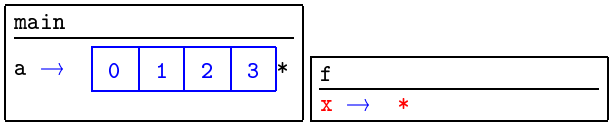


5. Print the current value of `a[1]`, which is `1`.

Since the Java program actually prints `5`, our model of how execution works is clearly not quite right. How do we fix it?

The mistake we made was in just *copying* the array from the first piece of paper to the second. Since arrays are mutable structures, we can't just copy them when we need to start a new piece of paper: we need to remember that the "copy" is just a new name for an array that we have already allocated.

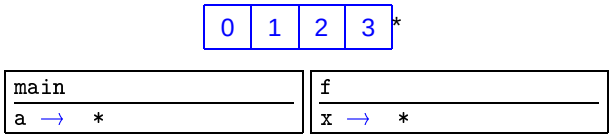
Pictorially, we can represent the situation like this. When we call `f`, instead of copying the array value to the new piece of paper, we put a reference back to the original array:



Now when we execute `x[1] = 5`, it is clear that we are changing the original array.

(Draw an arrow from the second * to the first.)

Our picture will be more consistent (and more accurate) if we draw it a little differently. Instead of writing the array on the first piece of paper and then putting an arrow to it from the second, we should really allocate the array outside of *either* piece of paper and draw arrows to it from both:



(Draw arrows from the second and third *'s to the first.)

Quick check

Sketch out what happens when this program executes:

```
public class test {
  public static int[] f() {
    int[] x = {0,1};
    x[1] = 5;
    return x;
  }

  public static void main (String[] args) {
    int[] a = f();
    System.out.println(a[1]);
  }
}
```

What about this one?

```
public class test {
  public static void main (String[] args) {
    int[] a = {0,1};
    int[] b = {2,3};
    b[1] = 99;
    a = b;
    System.out.println(a[1]);
  }
}
```


Aliasing

We can say that `a` and `x` in our original example are both *aliases* for the same array — two different ways of referring to the same piece of storage.

Moreover, aliasing does not just come up when we use arrays: we will soon see that it applies to objects as well.

The phenomenon of aliasing can sometimes make reasoning about Java programs quite subtle.

For example...

```
public static void f(int[] a, int[] b) {
  a[1] = 0;
  b[1] = 5;
  System.out.println(a[1]);
}
```

The method `f` always prints `0`, right?

Wrong.

```
public class test {
  public static void f(int[] a, int[] b) {
    a[1] = 0;
    b[1] = 5;
    System.out.println(a[1]);
  }

  public static void main (String[] args) {
    int[] a = {0,1,2,3};
    f(a, a);
  }
}
```

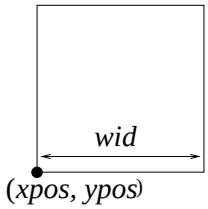
If the parameters `a` and `b` are aliases for the same array, it prints `5`.

Classes

Classes do many things for us in Java. We have already seen one use: each of the Java programs we have seen so far has been "wrapped up in" a class whose name corresponds to the name of the file in which the program is stored.

Classes in Java also play a role analogous to ML's datatypes. If we want to "package" several pieces of data together, we use a class. For example, Java does not have built-in tuple types such as `int * string`. When we want such a type, we build it using an appropriately defined class.

Recall that earlier this semester (Handout 4) we designed a type for a square, as it might be used in a rudimentary graphics program.



To build a new type `Square`, which contains the x and y co-ordinates of the lower left corner and a width, we can write:

```
class Square {
  double xpos;
  double ypos;
  double wid;
}
```


There are several things we can do with a `Square`; the first is to *create* one:

```
new Square ();
```

Another is to declare variables holding values of type `Square`. This declaration declares the variable `s` to hold a `Square` value and initialize `s` to a newly created square:

```
Square s = new Square ();
```

Note that the identifier `Square` in Java is used both for the type of squares and as the operation for constructing new values of this type.

In object-oriented parlance, we say that `s` holds an *object* that is an *instance* of the class `square`.

(We also sometimes revert to more ml-like terminology and say "`s` has type `square`.")

In our ml programs, `datatype` declarations were typically placed at the top of the program.

In java, class definitions like `square` are placed *outside* of the main class definition. Here, for example, is a complete program that defines the `square` class, uses it to create a square object, prints a useless message, and terminates.

```
class Square {
  double xpos;
  double ypos;
  double wid;
}

public class test {
  public static void main (String[] args) {
    Square s = new Square();
    System.out.println("That was fun");
  }
}
```


The fields of an object

The names `x`, `y` and `wid` are the parts of a square object that contain data. They are called the *fields* or *data members* of the object.

We can put data into a field by an assignment statement.

```
s.xpos = 1.5;
```

We can also extract data from a field by using the same notation (`s.xpos`) in an expression. E.g.,

```
System.out.println(2.5 + s.xpos);
```

We can put these ideas together to write methods (in the `main` class) that create new squares by transforming squares that they are passed as arguments:

```
public static Square move(Square s, double dx, double dy) {
   Square res = new Square();
   res.xpos = s.xpos + dx;
   res.ypos = s.ypos + dy;
   res.wid = s.wid;
   return res;
}

public static Square expand(Square s, double factor) {
  Square res = new Square ();
  res.xpos = s.xpos - wid * 0.5 * (factor - 1);
  res.ypos = s.ypos - wid * 0.5 * (factor - 1);
  res.wid = factor * s.wid;
  return res;
}
```


These behave like the ML functions we could write for the data type `square`.

(Note that expanding a square (about its center) moves the coordinates of its bottom-left corner.)

A complete program:

```
class Square {
  double xpos;
  double ypos;
  double wid;
}

public class test5 {
 // code for move and expand as above

  public static void  main (String[] args) {
    Square s = new Square();
    s.xpos = 1; s.ypos = 1; s.wid = 0.5;
    Square q = expand(move(s, 1.0, 1.0), 1.5);
    System.out.println(q.xpos + "  " + q.ypos);
  }
}
```


The output from this program is

```
1.875  1.875
```

Note that this is a complete program with *two classes*. `test5` is the class that matches the file name and is the one from which the `main` method is called when the program is run.

Note. We have been sloppy and have left out `public`. This means that the data membersa and methods are only visible within a given *package*. If all your code is in one file, it's all in the same package. More about packages later.

Programming with state

Objects have “state” – the values in the fields – and we can change the state of these fields just as we can change the contents of an array. For example, we might choose to make our functions on squares *change the state* of the square object they are given rather than creating a new square and returning it:

```
  public static void move(Square s, double dx, double dy) {
     s.xpos = s.xpos + dx;
     s.ypos = s.ypos + dy;
   }

  public static void expand(Square s, double factor) {
    s.xpos = s.xpos - s.wid * 0.5 * (factor - 1);
    s.ypos = s.ypos - s.wid * 0.5 * (factor - 1);
    s.wid = factor * s.wid;
  }
```


Now observe how the calling program differs. Rather than applying a sequence of functions to a square, resulting in new squares each time, we perform a sequence of commands that change the states of the fields of our original square:

```
public class test6 {
  public static void move(Square s, double dx, double dy) {
    ...  }  // as above

  public static void expand(Square s, double factor) {
    ...  }  // as above

  public static void main (String[] args) {
    Square s = new Square();
    s.xpos = 1; s.ypos = 1; s.wid = 0.5;
    move(s, 1.0, 1.0);
    expand(s, 1.5);
    System.out.println(s.xpos + "  " + s.ypos);
  }
}
```

Object-oriented programming

We've seen how to create objects that encapsulate multiple related pieces of data (e.g., the two coordinates of our squares) in a single structure that can be passed around as a unit. But the operations on these objects still look pretty much like the functions we have been writing all along; for example, we move a square object by passing it as the argument to the `move` method.

Object-oriented languages like Java also support a different way of organizing programs, in which data and associated operations are *packaged together* in objects.

- functional style... "Invoke the `move` method to move `s`."

```
move(s, 1.0, 1.0);
```

- OO style... "Tell `s` to move itself."

```
s.move(1.0, 1.0);
```


Packaging methods with objects

Here is a new `Square` class incorporating the `move` and `expand` methods.

```
class Square {
  double xpos;
  double ypos;
  double wid;

  public void move(double dx, double dy) {
    xpos = xpos + dx;
    ypos = ypos + dy;
  }

  public void expand(double factor) {
      xpos = xpos - wid * 0.5 * (factor - 1);
      ypos = ypos - wid * 0.5 * (factor - 1);
      wid = factor * wid;
  }}
```

Note that, inside the class, the bare field names are used to refer to the values of the fields of the current object (`xpos` rather than `s.xpos`).

Also, note that we drop the keyword `static` from the headers of the methods. Intuitively, the methods we have seen so far have been "static" in the sense that they are associated with classes. The methods we are seeing now are "dynamic" in the sense that they are associated with individual objects.

This change of terminology signals an important fact. For the moment, all our squares have the same implementation of the `move` method; but we should really think of each square object as choosing its own response to a request to move itself. Later on, we will see that different square objects can be made to respond to `move` in completely different ways.

Here is an example of the use of the class.

```
public class test7 {
  public static void main (String[] args) {
    Square s = new Square();
    s.xpos = 0.0; s.ypos = 0.0;
    s.move(1.0, 0.0);
    s.expand(1.5);
    System.out.println(s.xpos + "  " + s.ypos);
  }
}
```

The O-O programming style

In object-oriented programming, the operations on objects are regarded as *things that they do*, rather than *things that are done to them.* (In fact, even the terminology is different: we speak of an object's methods as constituting its "behavior.")

This shift of perspective has far-reaching consequences for software design: it focuses attention on what things can *do* rather than on how they are *represented*.

Constructors

So far we have used the "default" constructor `new Square()` to construct a square. The default constructor takes no arguments and does nothing special with the fields of the newly constructed object.

A Java class may also include an explicit constructor. This constructor can take arguments, and can include code for setting the initial values of the fields. For example, here is another variant of the `Square` class.

```
class Square {
  double xpos;
  double ypos;
  double wid;

  Square(double xposv, double yposv, double widv) {
    xpos = xposv;
    ypos = yposv;
    wid = widv;
  }

  ... // code for move and expand methods, as before
}
```

The constructor looks almost like a method, but its name is the same as the name of the class. We can use it to create a square like this:

```
new Square(1.0, 2.0, 3.4)
```


Things to note about constructors:

- The constructor has the same name as that of the class.
- There is no declared result type (or, if you like, the result type and the name of the constructor are rolled into one.)

For completeness, here is how the three-parameter constructor gets used.

```
public class test8 {
  public static void main (String[] args) {
    Square s = new Square(0.0,0.0,1.0);
    s.move(1.0, 0.0);
    s.expand(1.5);
    System.out.println(s.xpos + "  " + s.ypos);
  }
}
```


Using arrays to implement strings

The Java class `String` is probably implemented using arrays. Alternative implementations (e.g. lists) are certainly possible. We can use our knowledge of arrays to implement such a class (except the exceptions).

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. The contents of the character array are copied; subsequent modification of the character array does not affect the newly created string.

  Parameters: value - the initial value of the string.

  Throws: NullPointerException - if value is null.

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 16-bit Unicode characters in the string.

  Returns: the length of the sequence of characters represented by this object.

- `public char charAt(int index)`

  Returns the character at the specified index. An index ranges from 0 to length() - 1. The first character of the sequence is at index 0, the next at index 1, and so on, as for array indexing.

  Parameters: index - the index of the character.

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

  Throws: IndexOutOfBoundsException - if the index argument is negative or not less than the length of this string.

```
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 String object is returned. Otherwise, a new String object is created, representing a character sequence thatis the concatenation of the character sequence represented by this String object and the character sequence represented by the argument string.

  Examples: `"cares".concat("s")` returns `"caress"`
  `"to".concat("get").concat("her")` returns `"together"`

  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.

  Throws: NullPointerException - if str is null.

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.

```
  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
  }
```


Interfaces

Going back to our `Square` class, consider the implementation of another simple shape:

```
class Circle {
    double xpos;  //coordinates of center
    double ypos;
    double rad;   //radius

    Circle(double x, double y, double r) {xpos = x; ypos = y; rad = r;}

    public void move(double dx, double dy) {
        xpos = xpos + dx;
        ypos = ypos + dy;
    }
    public void expand(double factor) { rad = factor * rad; }

    public double diameter() { return(2 * rad); }
}
```

`Square` and `Circle` are different classes. An instance of `Square` is not an instance of `Circle` and *vice versa*.

In many situations, we'd like to be able to program in such a way that we can manipulate instances of either class with the same code. To this end we define a common *interface* for both classes:

```
interface Shape {
  void move(double x, double y);
  void expand(double f);
  Box boundingBox();
}
```

In the interface we see methods that we expect any shape to possess. The `boundingBox` method produces the smallest enclosing rectangle (we'll define this shortly); the other methods – `move` and `expand` – already exist in both the `Square` and `Circle` classes.

Note that our interface does not include methods like `diagonal`, which are specific to one class. Nor does it include fields like `xpos` whose meaning is different in the two classes.

Bounding boxes

First we define a class for a rectangular box.

```
class Box {
    double xpos;  // co-ordinates of corner
    double ypos;
    double wid;   // width
    double ht;    // height

    // Constructor
    Box(double x, double y, double w, double h) {
        xpos = x; ypos = y; wid = w; ht = h;
    }
}
```

Now we can augment our classes for `circle` and `square` with a `boundingBox` method and declare that it matches or *implements* the signature.

```
class Circle implements Shape {
    double xpos;
    double ypos;
    double rad;

    Circle(double x, double y, double r)     // as before

    public void move(double dx, double dy)   // as before
    public void expand(double factor)        // as before
    public double diameter()                 // as before

    public Box boundingBox(){
        return(new Box(xpos-rad, ypos-rad, 2*rad, 2*rad));
    }
}
```


The `Square` class is similarly augmented:

```
class Square implements Shape {
    double xpos;
    double ypos;
    double wid;

    Square(double x, double y, double w)   // as before

    public void move(double dx, double dy) // as before
    public void expand(double factor)      // as before
    public double diagonal()               // as before

    public Box boundingBox() {
        return(new Box(xpos, ypos, wid, wid));
    }
}
```

Note: if we hadn't added the `boundingBox` methods, these classes would not have compiled. By declaring

```
class Square implements Shape ...
```

we asserted that the class should match the interface. If it doesn't the compiler will complain.

Using interfaces

Like classes, interfaces can be used as the types of variables.

```
Shape c = new Circle(0,0,1);
c.move(10,10);
c.expand(2);
Box r = c.boundingBox();
System.out.println(r.xpos + " " + r.ht);

Shape s = new Square(0,0,1);
s.move(10,10);
s.expand(2);
Box t = s.boundingBox();
System.out.println(t.xpos + " " + t.ht);
```

(The outputs from these two sequences of statements are different — why?)

Note that the only time we use the classes `Square` and `Circle` directly is to *construct* objects. Everywhere else, we use the common type `Shape`.

We can use the interface `Shape` to write more methods that work equally well on circles or squares (or any class that implements `Shape`.)

```
public static void move_and_expand (Shape s) {
  s.move(1,1);
  s.expand(2);
}
```


Heterogeous arrays

We can construct arrays of "type" `Shape` and populate them with circles or squares.

```
public static void main (String[] args){
  Shape[] a = new Shape[3];
  a[0] = new Square(1,2,3);
  a[1] = new Circle(3,4,5);
  a[2] = new Square(7,7,8);
  for (int i=0; i< 3; i++) {System.out.println(a[i].boundingBox().xpos);}
  for (int i=0; i< 3; i++) {a[i].move(2,2);}
  for (int i=0; i< 3; i++) {System.out.println(a[i].boundingBox().xpos);}
}
```

We can also use `Shape` in building new "graphics" classes. For example, in graphics editors we often want to group a collection of objects into one.

```
class Group implements Shape{
  Shape [] contents;

  public void move(double x, double y){
    for (int i=0; i< contents.length; i++){
      contents[i].move(x,y);
    }
  }
  public void expand(double x){
    for (int i=0; i< contents.length; i++){
      contents[i].expand(x);
    }      // is this really how expand should work ???
  }
  public Box boundingBox(){  // code to find the maxima and minima of
                             // all boundaries of  the constituent shapes
  }
}
```


If we have a variable of type `Shape` whose current value is an object of type `Circle`, we are not allowed to use use data members or methods of the `Circle` class on this variable unless they are among the ones described in the interface `Shape`. In other words, storing a `Circle` object in a `Shape` variable "forgets" that it is really a circle.

For example

```
System.out.println(s.diagonal)
```

generates a compiler error.

Let's look in a little more detail at what's going on here...

Subtyping

Notice that the statement

```
Shape p = new Square(0,0);
```

assigns a value of type `Square` to a variable of type `Shape`.

This is an example of a general mechanism called *subtyping* (or *subtype polymorphism*).

When we declare that a class `C implements` some interface `I`, the compiler checks that the body of `C` really provides all of the facilities listed in `I`.

If it does, then `C` is said to be a subtype of `I`, and values of type `C` may be used in any situation where type `I` is required.

- a value of type `C` may be assigned to a variable of type `I`
- a value of type `C` may be passed to a method that expects an argument of type `I`
- etc.

(If it does not, then `C`'s definition is erroneous and the compiler stops.)

Note - once again – that the unqualified word *polymorphism* is ambiguous:

- In OCaml, “polymorphism” means parametric polymorphism — the ability to write functions that operate uniformly on data of any type. OCaml also has subtype polymorphism (it is more advanced than other versions of ML in this respect)
- In Java, “polymorphism” means subtype polymorphism—the ability to use objects of a class `C` where an interface `I` is expected, as long as `C` provides all of the features described by `I`.

  Java 2 does not allow parametric polymorphism. (A future release of Java will probably support ML-like parametric polymorphism.)

Recursive Data Types

Recursive types are types that are defined in terms of themselves. E.g., in ML:

```
datatype intlist =
    Nil
  | Cons of int * intlist
```

We can do something very similar in Java.

```
class IntList {
  int content;
  IntList next;
  IntList(int h, IntList t) { content = h; next = t; }
}
```


We can write head, tail and cons functions that work just as they do in ML. They all return results.

```
class IntList {
  int content;
  IntList next;
  IntList(int h, IntList t) { content = h; next = t; }
}

public class test9 {
  public static int hd(IntList l) { return l.content; }
  public static IntList tl(IntList l) { return l.next; }
  public static boolean isNull(IntList l) { return (l==null); }

  public static IntList cons(int i, IntList l) {
    return (new IntList(i,l));
  }
  ...
}
```

What happened to the empty list constructor `nil`?

There is a special Java value called `null` which is an instance of every class. It cannot be confused with any instance that is created by a constructor, so we use this to represent the empty list.

This explains the code for `isNull`.

Now we can program with lists just as we do in ML:

```
public static IntList snoc(int i, IntList l) {
  return (isNull(l) ? cons(i,null)
                    : cons(hd(l), snoc(i,tl(l))));
}

public static IntList reverse(IntList l) {
  return (isNull(l) ? null
                    : snoc(hd(l),reverse(tl(l))));
}
```


We also need to print lists:

```
public static void printIntList (IntList l) {
  if (! isNull(l)) {
    System.out.print(hd(l) + " ");
    printIntList(tl(l));
  }
}
```

This is how we might use these functions:

```
public static void main (String[] args) {
  IntList l = cons(1, cons(3, cons(7, cons(2, null))));
  printIntList(reverse(l));
}
```

Lists, imperative style

Our lists so far are (like the ones we saw in ML) completely immutable—we can build and traverse lists, but we cannot change the contents of lists that we've already built.

Also our definitions of lists are arguably not very "object oriented," since all the functions that operate on lists are placed outside the `IntList` class.

We will now develop an alternative implementation of lists using methods that (when appropriate) change the state of the list. If `l` is a list, they will operate like this:

- `l.cons(3)` "Cons 3 onto yourself"
- `l.hd()` "Return your head"
- `l.tl()` "Change yourself into your own tail"

Achieving this kind of behavior is a bit tricky.

As a first attempt, we could try to program this by adding methods to our `IntList` class...

```
class IntList {
  int content;
  IntList next;
  IntList(int h, IntList t) { content = h; next = t; }

  void tl() {
    content = next.content;
    next = next.next;
  }
}
```

But this does not work correctly for a one-element list: if `next` is `null`, then `tl` will fail with a "null pointer exception."

In other words, we cannot get `tl` to turn a one-element list into `null` if we represent lists in this way.

The solution is to use *two* classes.

```
class IntListCell {
  int content;
  IntListCell next;
  IntListCell(int i, IntListCell n) { content = i; next = n; }
}

class IntList {
  IntListCell root;
  IntList() { root=null; }
  int hd() { return(root.content); }
  boolean isNull() { return (root==null); }
  void etl() { root = root.next; }   //side-effecting tail
  void econs(int i) {                //side-effecting cons
    IntListCell lc = new IntListCell(i, root);
    root = lc;
  }
}
```


The class `IntListCell` gives us the recursive type. The class `IntList` has one data member, `root` which references the start of a IntList.

We are at liberty to assign `null` to `root`.

We used the names `econs` and `etl` to indicate that these work by having an “effect” rather than returning a result.

Some additional methods

```
void eappend(IntList l) {
  if (root == null) { root = l.root; }    //important special case
  else {
    IntListCell lc = root;
    while (lc.next != null) { lc = lc.next; }
    lc.next = l.root; }
}

void print() {
  IntListCell lc = root;
  while (lc !=null) {
    System.out.print(lc.content + " ");
    lc = lc.next; }
  System.out.println();
}
```



```
IntList l1 = new IntList();
IntList l2 = new IntList();
l1.econs(3); l1.econs(4); l1.econs(5);
l2.econs(6); l2.econs(8); l2.econs(6);
l1.print();
l1.etl();
l1.print();
l1.eappend(l2);
l1.print();
l1.eappend(l1);
l1.print();                    //What will happen?
```

Here's the output...

```
5 4 3
4 3
4 3 6 8 6
4 3 6 8 6 4 3 6 8 6 4 3 6 8 6 4 3 6 8 6 4 3 6 8 6 4 3 6 8 6 4 3 6 8
6 4 3 6 8 6 4 3 6 8 6 4 3 6 8 6 4 3 6 8 6 4 3 6 8 6 4 3 6 8 6 4 3 6
8 6 4 3 6 8 6 4 3 6 8 6 4 3 6 8 6 4 3 6 8 6 4 3 6 8 6 4 3 6 8 6 4 3
6 8 6 4 3 6 8 6 4 3 6 8 6 4 3 6 8 6 4 3 6 8 6 4 3 6 8 6 4 3 6 8 6 4
3 6 8 6 4 3 6 ...
```


The Object class

Although Java has no parametric polymorphism, there is a class `Object` which is a "supertype" of all other classes (and interfaces). What this means is that if we have a variable of type `Object` we can assign to it an object of any class whatsoever.

If we replace our `ListCell` class by

```
class ListCell {
  Object content;
  ListCell next;
  ListCell(Object i, ListCell n) { content = i; next = n; }
}
```

and change the types of the methods in `IntList` appropriately, we can now build lists containing instances of any class. We'll call this generic list class `List`.

Note that, unlike ML, Java lists can be *heterogeneous*: one list may contain objects of two or more different classes.

Not all types are classes

The basic types, `int`, `double`, `boolean`, etc. are not classes. However `String` (note the capital 'S') is a class. We can therefore build lists of strings. (It is also possible to build lists of integers etc., by turning integers into objects, but that's another story.)

```
public static void main (String[] args) {
  List l1 = new List();
  List l2 = new List();
  l1.econs("three"); l1.econs("four"); l1.econs("five");
  l2.econs("six"); l2.econs("eight"); l2.econs("six");
  l1.print();
  l1.etl();
  l1.print();
  l1.eappend(l2);
  l1.print();
}
```


The Java List class

The Java API contains a class `LinkedList` for mutable lists that bears some resemblance to what we have implemented. There is also a *signature* `List`, which is implemented by `LinkedList`, `Vector`, and `ArrayList` (the last two implementations are based on arrays)

If you look at the class `LinkedList` specification you will find many methods. Here are some correspondences.

Our `List` class	Java `LinkedList` class
`new List ()`	`new LinkedList()`
`l.isNull()`	`l.size == 0`
`l.hd()`	`l.getFirst()`
`l.etl()`	`l.removeFirst()`
`l.econs(x)`	`l.addFirst(x)`
`l.append(l1)`	—

(Apropos `append`, the Java `LinkedList` class appears to be designed in such a way that it is impossible to construct circular lists.)

The Enumeration Interface

Many data structures represent some kind of *collection*. A list is one example; a binary tree is another. (There are also arrays and vectors, which we'll see in detail later.) While each of these collection types is different, they have some common aspects. For example, it is often useful to be able to ask a collection to deliver its members to us in some order. For this purpose Java provides, in the package `java.util`, an `Enumeration` interface:

```
interface Enumeration {
  boolean hasMoreElements();
  Object nextElement();
}
```

A typical way of using this is (assume `v` is an instance of some collection class):

```
java.util.Enumeration e = v.elements();
while (e.hasMoreElements()) {
  System.out.println(e.nextElement()); }
```

Notice that `nextElement` both changes `e` and returns a result.

To add an `enumerate` method to our `List` class, we first write an class `Enum`, which implements `Enumeration`. The `enumerate` method constructs an appropriate instance of this class and returns it.

```
class ListEnum implements java.util.Enumeration {
    ListCell lc;
    ListEnum(List l) { lc = l.root; }
    public boolean hasMoreElements() { return( lc != null ); }
    public Object nextElement() {
        Object o = lc.content;
        lc = lc.next;
        return o; }
}
```

Using an enumeration object

```
public static void main(String[] args){
    List l1 = new List();
    l1.econs("interesting");
    l1.econs("getting");
    l1.econs("is");
    l1.econs("Java");
    java.util.Enumeration e1 = new ListEnum(l1);
    while (e1.hasMoreElements()) {
        System.out.print(e1.nextElement()+" "); }
    System.out.println();
}
```


Another Example

```
public static void main(String[] args) {
    List l2 = new List();
    l2.econs(new Square(0,0,1));
    l2.econs(new Circle(0,3,1));
    l2.econs(new Square(0,8,2));
    java.util.Enumeration e2 = new ListEnum(l2);
    while (e2.hasMoreElements()) {
        Object o = e2.nextElement();
        Shape g = (Shape) o;
        g.move(0,1); }
}
```

Note the use of the *coercion* (or *down-cast*) `(Shape) o`. This is a declaration to the typechecker that we "know" that the actual type of the object stored in `o` at run time will be `Shape`. At compile time, the typechecker just believes what we tell it (so the result type of this expression is `Shape`). At run time, it double-checks whether the object is really a `Shape` and raises an exception if it is not.