25. Recursion
Loops are one mechanism for making a program execute a statement a variable number of times. Recursion offers an alternative mechanism, considered by many to be more elegant and intuitive. It is the primary mechanism for repeating execution in some languages. The repetition is achieved, not by using some special new kind statement, but by simply making a method call itself.

Recursion in the non-computer world
This concept is not restricted to the computer world. Words in dictionaries are often defined, directly or indirectly, in terms of themselves. If you look up the Oxford dictionary, “adequate” is defined as “satisfactory,” and vice versa. One can imagine defining “recursion” as “recursion” to illustrate the concept in a very direct manner! The Oxford dictionary actually defines this word as an “expression giving successive terms of a series,” pointing out the use of recursion in Mathematics. In fact, as we will see here, we many of the recursive methods we will implement simply encode mathematical definitions of various kinds of number series. The Webster dictionary defines it as a “procedure repeating itself indefinitely or until condition met, such as grammar rule”. Let us look at grammar rules in some depth, as they are the foundation of many computer science concepts you will study in future courses, and perhaps best illustrate the kind of recursion on which we will focus here.

Consider noun phrases such as:
boy
little boy
smart little boy
naughty smart little boy
We can see here a systematic way of creating noun phrases. A noun phrase can be a noun such as a “boy”. Or it can be an adjective such as “little” followed by a noun phrase. Thus, we have recursively defined a noun phrase in terms of itself. This is illustrated more formally in the following definitions:

\[<\text{Noun Phrase}> \rightarrow <\text{Noun}>\]
\[<\text{Noun Phrase}> \rightarrow <\text{Adjective}> <\text{Noun Phrase}>\]

The terms within angle brackets are called non-terminals. These don’t actually appear in sentence fragments we are describing. The components of sentence fragments such as “smart” and “boy” are called terminals. Think of non-terminals as types such as int and String and think of terminals as values of these types such as 3 and “hello”. The matching of terminals such as boy with corresponding non terminals such as nouns is described outside the grammar.

Each of the above rules describes a non-terminal on the left hand side in terms of non-terminals and terminals on the right hand side. There may be more than one rule describing the same non-terminal. In fact, when one of the rules is recursive, such as the second rule above, there must be another non-recursive rule to ensure the derivation process halts. Alternative definitions of the same non-terminal will correspond to use of conditionals in the recursive methods we will see.
Let us see how different sentence fragments can be recognized by the rules above. Consider the fragment “boy”. Figure 1 (a) shows how it can be derived from the rules above. Similarly, Figure 1 (b), (c), and (d) show how the sentence fragments “little boy,” and “smart little boy” are derived from these rules.

(a) Deriving “boy”  
(b) Deriving “little boy”  
(c) Deriving “smart little boy”

**Figure 1 Deriving various noun phrases from the rules above**

The process of writing recursive methods will follow a similar structure, as we see below.

**Developing a Recursive Solution**

To illustrate the nature of recursion, consider the function, `factorial()`, which takes as an argument a positive integer parameter, `n`, and returns the product of the first `n` positive integers. We can solve this problem using iteration (loops). Here we will develop a recursive solution to it.

Before we look at the steps of the function, let us look at what this function computes for different values, `n`.

<table>
<thead>
<tr>
<th><code>factorial(n)</code></th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>factorial(0)</code></td>
<td>1 * factorial(0)</td>
</tr>
<tr>
<td><code>factorial(1)</code></td>
<td>1 * factorial(0)</td>
</tr>
<tr>
<td><code>factorial(2)</code></td>
<td>2 * factorial(1)</td>
</tr>
<tr>
<td><code>factorial(3)</code></td>
<td>3 * factorial(2)</td>
</tr>
</tbody>
</table>

Based on the pattern in these examples, we can say:

- `factorial(0) = 1`  
- `factorial(n) = n * factorial(n-1)` if `n > 0`

What we have above, in fact, is a precise definition of the problem we must solve. As it turns out, it also gives us an algorithm for solving the problem:

```java
public static int factorial (int n) {
    if (n == 0)
        return 1;
    if (n > 0)
        return n * factorial(n-1);
}
```

**Figure 2 Recursive Factorial**

---

2 It is not clear what the product of the first 0 positive integers is. The convention is to assume it is 1.
Recursion

If we were to try and compile this method, the compiler would complain, saying that the function does not return a value. This is because we have not covered all possible values of \(n\). We are assuming here that \(n\) is a positive integer. Though that is the expected value, Java will not prevent a negative integer to be passed as an actual argument. Therefore, we must specify what value should be returned in this case. Let us return the factorial of the absolute value of \(n\), for negative values of \(n\):

```java
public static int factorial (int n) {
    if (n == 0)
        return 1;
    else if (n < 0)
        return factorial (-n);
    else
        return n*factorial(n-1);
}
```

A method such as factorial that calls itself is called a *recursive method*.

Notice how compact our recursive solution is. It can be considered more elegant than the iterative solution because the algorithm for the problem is also the definition of the problem! As a result, it is easy to convince ourselves that our solution meets the requirements laid out by the definition, and we do not need to risk the off-by-one errors of loop iteration.

The key to using recursion is to identify a definition that meets the following two requirements:

1. **Recursive Reduction Step(s):** It defines the result of solving a larger problem in terms of the results of one or more smaller problems. A problem is considered smaller than another problem if we can solve the former without having to solve the latter. In our example, the problem of computing the factorial of positive \(n\) is reduced to the problem of computing the factorial of a smaller number, \(n-1\); and the problem of computing the factorial of a negative integer is reduced to the problem of computing the factorial of its positive, absolute value.

2. **Base Terminating Case(s):** It has terminating condition(s) indicating how the base case(s), that is, the smallest size problem(s), can be solved. In the example, it tells us that factorial(0) = 1. In general, there can be more than one base case, for instance one for negative numbers and another for 0.

Once we define the problem in this way, we can use recursion to solve it. The general form of a recursive algorithm is:

```java
if (base case 1 )
    return solution for base case 1
else if (base case 2)
    return solution for base case 2
....
else if (base case n)
    return solution for base case n
else if (recursive case 1)
    do some preprocessing
    recurse on reduced problem
    do some postprocessing
....
else if (recursive case m)
    do some preprocessing
    recurse on reduced problem
```
Recursion

do some postprocessing

**Stacking/Unstacking Recursive Calls**

Before we look at other recursive problems, let us try to better understand `factorial` by tracing its execution. Assume a main method of invokes:

```
factorial(2)
```

When this call is made, we know the value of the formal parameter of the method, but not its return value. This situation can be expressed as:

<table>
<thead>
<tr>
<th>Invocation</th>
<th>n</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>factorial(2)</td>
<td>2</td>
<td>?</td>
</tr>
</tbody>
</table>

Once the formal parameter is assigned, the method executes the if statement. The if test fails, so the else part is executed, as shown by the window below:

```
public class RecursiveFactorial {
    public static void main (String args[]) {
        System.out.println(factorial(2));
    }
    public static int factorial (int n) {
        if (n == 0)
            return 1;
        else if (n < 0)
            return factorial (-n);
        else
            return n*factorial(n-1);
    }
}
```

**Figure 3 Recursive Call to factorial**

The else part contains another call to `factorial()`, with the parameter this time being 1. Thus, we now have two executions of the same method running concurrently. Each execution has its own copy of the
formal parameter and computes its own copy of the return value. The following table identifies the formal parameters and return values of the two invocations when the second call to `factorial()` is made:

<table>
<thead>
<tr>
<th>Invocation</th>
<th>n</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>factorial(1)</td>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>factorial(2)</td>
<td>2</td>
<td>?</td>
</tr>
</tbody>
</table>

In a trace like this, it is usual to stack up the calls, that is, put an invoked method above the invoker. The topmost call with an uncomputed return value is the one the computer is currently executing.

The following screen dump creates an alternative view of a call stack, showing in the pull down menu all the calls active at this point, including the call to the main method made by the interpreter:

<table>
<thead>
<tr>
<th>Locals</th>
</tr>
</thead>
<tbody>
<tr>
<td>RecursiveFactorial.factorial(n=1) &lt;Java&gt;</td>
</tr>
<tr>
<td>RecursiveFactorial.factorial(n=1) &lt;Java&gt;</td>
</tr>
<tr>
<td>RecursiveFactorial.factorial(n=2) &lt;Java&gt;</td>
</tr>
<tr>
<td>RecursiveFactorial.main(args=(length=0);) &lt;Java&gt;</td>
</tr>
</tbody>
</table>

**Figure 4 Call Stack**

Let us now trace what happens as `factorial(1)` executes its body. Again the test fails, and another call to `factorial()` is made, this time with the actual parameter 0:

<table>
<thead>
<tr>
<th>Invocation</th>
<th>n</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>factorial(0)</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>factorial(1)</td>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>factorial(2)</td>
<td>2</td>
<td>?</td>
</tr>
</tbody>
</table>

When `factorial(0)` executes the if statement, the test finally succeeds, and the function returns 1 and terminates execution.

<table>
<thead>
<tr>
<th>Invocation</th>
<th>n</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>factorial(0)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>factorial(1)</td>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>factorial(2)</td>
<td>2</td>
<td>?</td>
</tr>
</tbody>
</table>

At this point control transfers back to `factorial(1)`, which finishes execution of the statement:

```
return n*factorial(n-1);
```

that is, multiplies the value returned by `factorial(0)` to its copy of `n`, which is 1, returns the result, and terminates:

<table>
<thead>
<tr>
<th>Invocation</th>
<th>n</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>factorial(0)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>factorial(1)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>factorial(2)</td>
<td>2</td>
<td>?</td>
</tr>
</tbody>
</table>
Recursion

Control transfers now to factorial(2), which multiplies the return value of factorial(1) to its value of n, returns the result, and terminates.

<table>
<thead>
<tr>
<th>Invocation</th>
<th>n</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>factorial(0)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>factorial(1)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>factorial(2)</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 5 Values of Formal Parameters and Return Values of Each Call

This return value is received by main, which prints it, giving us the output:

2

Thus, as this trace of the recursive function shows, successive calls to reduced versions of the problem get stacked until a base case is reached, after which they get unstacked as reduced versions successively return their results to their callers. Each stacked call get its own copies of the formal parameters and return value.

Recursion Pitfalls

We must be careful to have a terminating condition in a recursive program. Consider the following definition of factorial(), which does not have a terminating condition:

```java
public static int factorial (int n) {
    return n*factorial(n-1);
}
```

Let us trace again the execution of:

```java
factorial(2)
```

It computes:

2*factorial (1)

factorial (1) computes

1* factorial (0)

factorial (0) computes

0*factorial(-1)

factorial (-1) computes

-1*factorial (-2)

and so on. Thus, we make an infinite number of calls to factorial.

We must also be careful to make the recursive step compute a reduced version of the problem. Let us say we write the factorial function as follows:

```java
public static int factorial (int n) {
    if (n == 0)
        return 1;
    else if (n < 0)
        return factorial (-n);
    else
        return factorial(n+1)/n+1;
}
```

Here, the second recursive step solves factorial(n) in terms of a harder problem: factorial (n+1). Consider again the call:

```java
factorial(2)
```

It computes:
Recursion

factorial(3) /3
factorial(3) computes:
factorial(4)/ 4
factorial(4) computes
factorial(5)/5
and so on. Thus, we make an infinite number of calls to factorial again. Even though we have a terminating condition, we never reach it since we do not reduce the problem.

In summary, if we do not have a base case or reduce the problem, we can end up making an infinite number of calls. When this happens, the computer gives an error message saying there has been a stack overflow. Each time a function is called, space must be created for its arguments and its return value from an area of memory called the stack. (It is so called because it stacks an invoked method’s data over the data of the calling method, as we showed in our tables above). Thus, a non-terminating sequence of recursive calls uses up the complete stack – hence the message of stack overflow.

**Recursive Function with two Parameters**

The recursive function above took one parameter. Let us consider one with two parameters. Suppose we are to write a `power()` function that takes as arguments, a base and a positive exponent, and raises the base to the power of the exponent, that is, computes:

\[
\text{base}^{\text{exponent}}
\]

For instance:

\[
power(2,3) == 8
\]

When we have two parameters, we need to know which parameter(s) to recurse on, that is, which parameters to reduce in the recursive call(s). Let us try first the base:

\[
\begin{align*}
power(0, 0) &= 1 \\
power(0, \text{exponent}) &= 0 \\
power(1, \text{exponent}) &= 1 \\
power(2, \text{exponent}) &= 2*2* \ldots 2 \text{ (exponent times)} \\
power(3, \text{exponent}) &= 3*3* \ldots 3 \text{ (exponent times)}
\end{align*}
\]

There seems to be no relation between:

\[
power(\text{base}, \text{exponent})
\]

and

\[
power(\text{base}-1, \text{exponent})
\]

Let us try again, this time recursing on the exponent parameter:

\[
\begin{align*}
power(\text{base}, 0) &= 1 \\
power(\text{base}, 1) &= \text{base} \cdot 1 = \text{base} \cdot \text{power (base, 0)} \\
power(\text{base}, 2) &= \text{base} \cdot \text{base} = \text{base} \cdot \text{power (base, 1)}
\end{align*}
\]

We can use the pattern above to give the general solution:

\[
\begin{align*}
power(\text{base}, \text{exp}) &= 1 \quad \text{if exp <= 0} \\
power(\text{base}, \text{exp}) &= \text{power (base, exp-1)} \quad \text{otherwise}
\end{align*}
\]

Our recursive function, again, follows straightforwardly:

```java
public static int power (int base, int exp) {
    if (exp <= 0)
        return (1);
    else
        return base * power(base, exp-1);
}
```
Recursion

In this example, this two-parameter function recurses on only one of its parameters. It is possible to write recursive functions that recurse on both parameters, as you see in an exercise. In general, when a recursive method has multiple parameters, we must carefully choose the parameters whose size we will reduce in the recursive step.

Recursive Procedures
Like functions, procedures can be recursive. Consider the problem of saying “hello world” n times. We can write a recursive definition of this problem:

\[
greet(0): \text{do nothing} \\
greet(1): \text{print “hello world”} = \text{print “hello world”; } \greet(0) \\
greet(2): \text{print “hello world”; print “hello world”} = \text{print “hello world”; } \greet(1)
\]

The general pattern, thus, is:

\[
greet(n): \text{do nothing} \quad \text{if } n \leq 0 \\
greet(n): \text{print “hello world”; } \greet(n-1); \text{if } n > 0
\]

Our recursive procedure, then, is:

```java
public static void greet (int n) {
    if (n > 0) {
        System.out.println("hello world");
        greet(n-1);
    }
}
```

A recursive function is a mechanism for creating a recursive compound expression - it evaluates part of a compound expression and calls itself to evaluate the remainder. For instance, `factorial(n)`, computes a compound expression that multiplies \( n \) with the result returned by `factorial(n-1)`. In contrast, a recursive procedure is a mechanism for creating a recursive compound statement - it performs part of the steps in a statement list, and calls itself recursively to perform the other steps. For instance, `greet(n)` executes a compound statement that prints one greeting and calls `greet(n-1)` to print the remaining greetings. Recursive functions are perhaps more intuitive because, from our study of mathematics, we are used to recursive expressions. Like recursive functions, recursive procedures are also very useful, often leading to more succinct implementations than loops.

Number-based vs List-based Recursion
So far, we have looked at number-based recursion, where the recursive method receives one or more numbers as arguments, does some computation involving the numbers, and reduces the problem in the recursive step by changing one or more of these numbers. In the problems we saw, a number was always reduced in the recursive call but it is possible to define recursive methods in which the number is increased. This corresponds to writing loops in which the loop index is increased or decreased.

Another important form of recursion involves list processing. By lists we mean any ordered sequence of items such as an array, vector, enumeration, or input stream. In some of these lists such as enumerations and streams, accessing an item removes it from the list, while in others such as arrays and vectors, that is not the case. In list-based recursion, the recursive method receives one or more lists are arguments does some computation involving some of the list elements, and in the recursive step, reduces the problems by removing one or more elements of the list arguments.
We see below several examples of list-based recursion.

**Stream-based recursion**

Like loops, we can use recursion for processing input streams that are terminated by sentinels. Consider the problem of multiplying a list of positive integers terminated by a negative sentinel. Let us look at some example cases:

- \( \text{multiplyList (inputStream)} = 1 \) if remaining input is -1
- \( \text{multiplyList (inputStream)} = 30 \times 1 \) if remaining input is 30 –1
- \( \text{multiplyList (inputStream)} = 2 \times 30 \times 1 \) if remaining input is 2 30 –1

Our recursive definition would be:

\[
\text{multiplyList (inputStream)} = 1 \quad \text{if next input value is < 0}
\]

\[
\text{multiplyList (inputStream)} = \text{readNextValue(inputStream)} \quad \text{otherwise}
\]

Here the base step returns 1 because the list of remaining items is empty. The reduction step removes one item from a non-empty list, and multiplies it to the product of the items in the remaining list.

The recursive function, thus, is:

```
public static int multiplyList(BufferReader inputStream) {
    int nextValue = readNextValue(inputStream);
    if (nextValue < 0) // no more items in the list
        return 1;
    else
        return nextValue * multiplyList(); // multiply nextValue to product of remaining items
}
```

To complete our solution to the problem, we need a method to start the recursion, providing the argument to the first call to the recursive function:

```
public static int multiplyInputs() {
    return multiplyList(new BufferReader(new InputStreamReader(System.in)));
}
```

Contrast this solution with the ones we have seen before to see the difference between number-based and list-based recursion. As mentioned before, in the previous cases the base cases corresponded to small values of one or more numbers, and reducing the problem amounted to reducing the values of these numbers. On the other hand, in this example, the base case corresponds to list items, and reducing the problem involves reducing the size of the list. Each call to readNextValue() reduces the size of the list by one. The number at the front of the list in a variable so that it can be used in the multiplication. Each recursive call get is own copy of this variable, since it is local to the method.

\text{multiplyList()} is an impure function, returning different values depending on how much of the input has been read. For instance, if the remaining input is:

\[
2 \quad 30 \quad -1
\]

the call:

\text{multiplyList()}

returns 60; but if the remaining input is:

\[
30 \quad -1
\]

the same call returns 30.
Recursion

To better understand this form of list-based recursion, let us trace the call to `multiplyList()` when the input is:

\[ 2 \ 30 \ -1 \]

The following table shows the input stream when the first call to `multiplyList()` is made:

<table>
<thead>
<tr>
<th>Invocation</th>
<th>Remaining input</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>multiplyList()</code></td>
<td>2 \ 30 \ -1</td>
<td>??</td>
</tr>
</tbody>
</table>

The method reads the first item, the number 2, of the list of remaining input values, and makes another call to `multiplyList()`. The list of remaining input values is different for this call, having one less item – the value read by the previous call, as shown below:

<table>
<thead>
<tr>
<th>Invocation</th>
<th>Remaining input</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>multiplyList()</code></td>
<td>30 \ -1</td>
<td>??</td>
</tr>
<tr>
<td><code>multiplyList()</code></td>
<td>2 \ 30 \ -1</td>
<td>??</td>
</tr>
</tbody>
</table>

Again, the method removes the first item, this time the number 30, from the input list, and makes another call to `multiplyList()`:

<table>
<thead>
<tr>
<th>Invocation</th>
<th>Remaining input</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>multiplyList()</code></td>
<td>-1</td>
<td>??</td>
</tr>
<tr>
<td><code>multiplyList()</code></td>
<td>30 \ -1</td>
<td>??</td>
</tr>
<tr>
<td><code>multiplyList()</code></td>
<td>2 \ 30 \ -1</td>
<td>??</td>
</tr>
</tbody>
</table>

The input seen by the third call is the base case, hence it simply returns 1:

<table>
<thead>
<tr>
<th>Invocation</th>
<th>Remaining input</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>multiplyList()</code></td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td><code>multiplyList()</code></td>
<td>30 \ -1</td>
<td>??</td>
</tr>
<tr>
<td><code>multiplyList()</code></td>
<td>2 \ 30 \ -1</td>
<td>??</td>
</tr>
</tbody>
</table>

The second call returns the value returned by the third call multiplied by 30:

<table>
<thead>
<tr>
<th>Invocation</th>
<th>Remaining input</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>multiplyList()</code></td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td><code>multiplyList()</code></td>
<td>30 \ -1</td>
<td>30</td>
</tr>
<tr>
<td><code>multiplyList()</code></td>
<td>2 \ 30 \ -1</td>
<td>??</td>
</tr>
</tbody>
</table>

Finally, the first call returns the value returned by the second call multiplied by 2:

<table>
<thead>
<tr>
<th>Invocation</th>
<th>Remaining input</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>multiplyList()</code></td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td><code>multiplyList()</code></td>
<td>30 \ -1</td>
<td>30</td>
</tr>
</tbody>
</table>
Recursion

Thus, each call processes a list with one less item, until we come to the base case of a list containing only – 1.

**Enumeration-based Recursion**

To illustrate enumeration-based recursion, let us consider the following method we saw earlier:

```java
static void print(StringHistory strings) {
    System.out.println("******************");
    StringEnumeration stringEnumeration = strings.elements();
    while (stringEnumeration.hasMoreElements()) {
        System.out.println(stringEnumeration.nextElement());
    }
    System.out.println("******************");
}
```

We need to replace the loop iterating through the enumeration with an equivalent recursive method that takes the enumeration as an argument. The original method, thus becomes:

```java
static void print(StringHistory strings) {
    System.out.println("******************");
    print(strings.elements());
    System.out.println("******************");
}
```

This method itself is not recursive. Like `multiplyInputs`, it simply makes the first call to the recursive method (with the same name) that processes the enumeration. This method follows the structure of the previous solution, processing the item in the front of the enumeration, and recursing on the rest of the enumeration:

```java
static void print(StringEnumeration stringEnumeration) {
    if (!stringEnumeration.hasMoreElements()) return;
    System.out.println(stringEnumeration.nextElement());
    print(stringEnumeration);
}
```

**Index-based Recursion**

In the above example, the StringHistory provided an elements() method to create an enumeration of its elements. What if such an enumeration was not available? As we saw earlier, it is possible to use the fact that it is indexed by the elementsAt() method to write a loop to iterate over its elements:

```java
static void print(StringHistory strings) {
    System.out.println("******************");
    int elementNum = 0;
    while (elementNum < strings.size()) {
        System.out.println(strings.elementAt(elementNum));
        elementNum++;
    }
}
```

How do we replace the index based loop with a recursive method? This method, like the previous list-based recursive methods we have seen, must know how many elements have been processed so far. Enumeration and stream objects store this information implicitly in their state. In this method, we must keep track of this information explicitly through another parameter, which starts of as 0 to indicate that no elements have been processed so far, as shown below:
Recursion

```java
static void print(StringHistory strings) {
    System.out.println("******************");
    print(strings, 0);
    System.out.println("******************");
}
```

The recursive method uses this extra parameter to access the next element to be printed, and then calls the recursive method with the next value of this parameter to indicate the next element to be printed:

```java
static void print(StringHistory strings, int elementNo) {
    if (elementNo == strings.size()) return;
    System.out.println(strings.elementAt(elementNo));
    print(strings, elementNo + 1);
}
```

**Var ++ vs. ++ Var**

What if we had used the incrementing assignment:

```java
    elementNo++;
```

in our call to the recursive method?

```java
static void print(StringHistory strings, int elementNo) {
    if (elementNo == strings.size()) return;
    System.out.println(strings.elementAt(elementNo));
    print(strings, elementNo++);
}
```

As it turns out, this function will recurse indefinitely. The reason is that `elementNo` is incremented after its use, that is, after the value of the parameter is calculated. As a result, all calls to the method will have the value 0 for this parameter.

The incrementing assignment,

```java
    ++elementNo
```

on the other hand, works:

```java
static void print(StringHistory strings, int elementNo) {
    if (elementNo == strings.size()) return;
    System.out.println(strings.elementAt(elementNo));
    print(strings, ++elementNo);
}
```

It is the converse of the previous assignment, as it increments the variable before its use.

**Single assignment in function calls**

However, both solutions have the drawback that they increment the value of `elementNo`, which is not necessary. In loop-based solutions, we are used to incrementing indices as the same variable is used to index multiple elements in a list. However, in recursion-based solutions, each recursive call gets its own copy of the index variable as a parameter, which needs to be assigned once, when the function is called. Thus, in the above solution, the `elementNo` of the called function needs to be assigned one plus the value of the `elementNo` of the calling function. The `elementNo` of the calling function does not need to be changed! In fact, in functional languages, which have no loops and rely only on recursion to execute a variable number of steps, all variables are final, that is, variables can be assigned values only once. Mathematically impossible statements such as:

```java
i = i + 1
```

are impossible in such languages. As we see in the examples above, in recursive solutions, single assignment is not a restriction.
Common aspects of list-based recursion

We have seen above three different examples of list based recursion, which are reproduced in Figure 6. They are different because they operate on different kinds of lists: streams, enumerations, and indexed collections respectively. But as the figure shows, they are very similar if we abstract to the common concept of a list.

```
public static int multiplyList(BufferedReader inputStream) {
    int nextVal = readInt(inputStream);
    if (nextVal < 0) return 1;
    else return nextVal * multiplyList(inputStream);
}
```

```
static void print(StringHistory strings, int elementNo) {
    if (elementNo == strings.size()) return;
    System.out.println(strings.elementAt(elementNo));
    print(strings, elementNo + 1);
}
```

```
static void print(StringEnumeration stringEnumeration) {
    if (!stringEnumeration.hasMoreElements()) return;
    System.out.println(stringEnumeration.nextElement());
    print(stringEnumeration);
}
```

```
public static int multiplyList(BufferedReader inputStream) {
    int nextVal = readInt(inputStream);
    if (nextVal < 0) return 1;
    else return nextVal * multiplyList(inputStream);
}
```

```
public Course matchTitle(String theTitle) {
    for (int courseIndex = 0; courseIndex < size; courseIndex++) {
        if (contents[courseIndex].getTitle().equals(theTitle))
            return contents[courseIndex];
    }
    return null;
}
```

Figure 6 Elements of list-based recursion

Each of these methods receives some representation of an ordered sequence of unprocessed items as one or more parameters. If the list is empty, the method returns, otherwise it processes the first item or *head* of this list in the base case and recourses on the remainder or *tail* of the list. We can write the following pseudo code in terms of the head and tail operations to illustrate the common thread in a recursive list-based method, m:

```
m(List l) {
    if (l.isEmpty()) return;
    process l.head();
    m(l.tail());
}
```

In general, the processing of the head and the recursive call can occur in the opposite order and the method can extract more than one item from the list before recursing.

In functional languages, there is one data type to describe lists, which provides standard ways to invoke the isEmpty, head and tail operations. As a result, list-based recursive methods are easy to write and understand in these languages. In Java, on the other hand, we must provide our own encoding of lists and implementations of these operations, and hence list-based recursive methods are a bit more awkward to write and implement. However, one can explore the essentials of list-based recursion in these languages, as the examples above show.

To gain more practice with list-based recursion, let us convert the following loop-based procedure to a recursive one:

```
public Course matchTitle(String theTitle) {
    for (int courseIndex = 0; courseIndex < size; courseIndex++) {
        if (contents[courseIndex].getTitle().equals(theTitle))
            return contents[courseIndex];
    }
    return null;
}
```
Recursion

The key here is to check the item at the head of the list. If it matches the title, then return it, else recurse to check the tail of the list, as shown by the pseudo code below:

```java
Course matchTitle (title, courses) {
    if courses.isEmpty() return null;
    if title equals courses.head().getTitle()
        return courses.head()
    else
        return matchTitle(title, courses.tail())
}
```

The actual code is slightly less intuitive as we must encode the list in two parameters and define a separate function to make the first recursive call:

```java
public Course matchTitle (String theTitle) {
    return matchTitle(theTitle, 0, contents);
}
```

```java
public Course matchTitle (String theTitle, int courseIndex, Course[] contents) {
    if (courseIndex == size) return null;
    if (contents[courseIndex].getTitle().equals(theTitle))
        return contents[courseIndex];
    return matchTitle(theTitle, courseIndex + 1, contents);
}
```

Perhaps even less intuitive is the following alternative definition of these methods:

```java
public Course matchTitle (String theTitle) {
    return matchTitle(theTitle, 0);
}
```

```java
public Course matchTitle (String theTitle, int courseIndex) {
    if (courseIndex == size) return null;
    if (contents[courseIndex].getTitle().equals(theTitle))
        return contents[courseIndex];
    return matchTitle(theTitle, courseIndex + 1);
}
```

This implementation relies on the fact that the indexable course list is stored a global variable whose value is not changed by the recursive calls. As a result, it does not have to be explicitly passed as a parameter to the calls, and becomes an implicit parameter to the calls. It is a good idea to pass global lists as parameters for program understanding purposes. Perhaps more compelling, such a recursive method is no restricted to processing only that global variable, it can process other lists of the same type.

**Recursion vs. Loops and Stacks**

All of the solutions we have seen so far could have been done using loops. As mentioned before, the recursive solutions have the advantage that the solution to the problem is the same as its definition. Recursive solutions may be a bit “cleaner” though the awkward representations of lists may be an issue. Moreover, it is amazing that method invocations can take the place of loops, or in other words, it is possible to ban loops from general-purpose programming languages!

Are there problems that can be done using recursion but not loops? The answer is no, as a language supporting recursion can be implemented using loops. Thus, indirectly, any recursive solution can have an underlying loop-based implementation. However, there are problems that are easier to do using recursion.
Recursion

Suppose we had to write a method that printed all items in an enumeration in reverse order. The recursive solution simply switches the order of the recursive and base case in the method that prints them in order:

```java
static void print(StringEnumeration stringEnumeration) {
    if (!stringEnumeration.hasMoreElements()) return;
    String head = stringEnumeration.nextElement();
    print(stringEnumeration); // print the tail
    System.out.println(head);   // print the head element after printing the tail
}
```

The above method makes sure that the first element is printed after 2nd, 3rd, and other elements; the 2nd element after the 3rd and later elements, and so on. Essentially it prints the tail of a list before the head of the list to ensure reverse printing.

The loop approach needs a more elaborate solution in which the enumerated items are stored in some object that allows these items to be then accessed in reverse order. Earlier, we used an indexable collection to store objects to be printed in the reverse order. This solution was not ideal as it gave users the ability to retrieve objects in arbitrary order, violating the principle of least privilege. The ideas candidate for such an object is a stack. We have used this term to describe chains of method calls. It is a more general abstraction “stacking” items from top to bottom, defined by four operations:

- isEmpty(): returns true if the stack has no items.
- push(item): pushes an item on top of existing items in the stack.
- top(): returns the top item of the stack.
- pop(): removes the top item from a stack.

Figure 7 diagrammatically illustrates the nature of a stack. Figure 7(a) and (b) show the result of pushing items A and B on an empty stack, Figure 7(c) shows that the last item pushed is the one returned by the top operation, and Figure 7(d) shows that the top item is removed by the pop operation.

```
A
B
A
B
A
```

Figure 7 Illustrating semantics of push, top and pop operations

As we see above, a stack has the Last In First Out (LIFO) property, that is the last item pushed to the stack if the first one popped. Now we can understand why a method call chain is stored as a stack – the last method called is the first one to return.

The following loop-based implementation of the problem above shows how a stack is used:

```java
static void print(StringHistory strings) {
    StringStack stringStack = new AStringStack();
    System.out.println("****************");
    StringEnumeration stringEnumeration = strings.elements();
    while (stringEnumeration.hasMoreElements()) {
        stringStack.push(stringEnumeration.nextElement());
    }
    while (!stringStack.isEmpty()) {
        System.out.println(stringStack.top());
        stringStack.pop();
    }
    System.out.println("****************");
}
```
Recursion

In comparison to the recursion-based solution, this one requires the implementation and use of a stack. The recursive solution does not require the programmer to explicitly use and implement a stack because the stack of recursive method calls (automatically created by the programming language implementation) serves the same purpose. There are many problems that require loop-based solutions to create and use stacks in far more complex ways and thus are much easier to code using recursion. We see below one such problem.

Print an object array

Consider the following array:

```java
static Object[] introProg = {new Integer(14), "Intro. Prog.", "COMP"};
```

For now ignore the fact the array is being used to encode information about a course. Suppose we used println to display it:

```java
System.out.println(introProg);
```

The output is a string indicating the type and address of the object:

```
[Ljava.lang.Object;@3f5d07
```

We might wish to, instead, see the elements of the array following the syntax used above to define array literals:

```
{14, Intro. Prog., COMP}
```

To create such a display, we have to write our own print method. Here is a loop-based implementation of it:

```java
static void print (Object[] objArray) {
    System.out.print("{");
    for (int index = 0; index < objArray.length; index++) {
        System.out.print(objArray[index]);
        if (index < objArray.length -1) System.out.print(",");
    }
    System.out.print ("} ");
}
```

For the example array, it does indeed the desired output. But for a nested array, that is, an array containing arrays, it is behavior is unsatisfactory. For example, given the additional definitions:

```java
static Object[] foundProg = {new Integer(114), "Found. Prog.", "COMP"};
static Object[] programming = {introProg, foundProg};
```

the following call:

```java
print(programming)
```

outputs:

```
{{Ljava.lang.Object;@3f5d07, [Ljava.lang.Object;@f4a24a}
```

What we would like is the array elements to be also decomposed as arrays:

```
{{14, Intro. Prog., COMP}, {114, Found. Prog., COMP}}
```

We can write a nested loop to get the desired output for this case. But what if the elements of the nested array are themselves nested? That would require a three-level nested loop.

We see the pattern here. A single-level loop is needed to print single-level array, a two-level nested loop is needed to print a two-level nested array, ..., an N-level nested loop is needed to print an N-level nested array. If N was known at program writing time, we can write such a loop. But when the array elements are determined at execution time, it is not possible to follow the above pattern. The solution is to use a stack in a complex manner, which we will not explore here.

Instead, let us abandon the loop-based approach and write a recursive method to print the array. When encountering an array element that is itself an array, the method can call itself recursively. Otherwise, it can call println() as before, relying on the toString() method of a non-array object to do the right thing. Java provides a way to determine if an object is an array, which is used in the implementation below.

```java
static void print (Object[] objArray) {
```
Recursion

```java
System.out.print("{\nfor (int index = 0; index < objArray.length; index++) {
    Object element = objArray[index];
    if (element.getClass().isArray())
        print ((Object[]) element);
    else
        System.out.print(objArray[index]);
    if (index < objArray.length -1)
        System.out.print(",");
}
System.out.print ("}\n```}

Now each array nesting level corresponds to a recursive call. Since the chain of recursive calls is not fixed, this solution can handle arbitrarily nested arrays.

We see here an example of a method that uses both loops and recursion. This is not uncommon in a non functional language such as Java, where you use recursion only when it is much more convenient than loops. Later we will see how we can create a pure recursive solution to this problem. For now, let us try to better understand the nature of the recursion we use above.

**Trees**

The object being processed by the procedure above is more general than a list. It is a list that contain sublists, which in turn can contain sublists, and so on. Such a structure is called a tree. A tree is represented, as shown in Figure 7, as a set of items, called nodes, connected by directed arrows, which describe parent child relationships. The item at the source of an arrow is called the parent of the item at the destination of the arrow. A node can be both a parent and a child, such as item B in the figure. A tree has a special item, called the root, which has no parent. Items with no children are called leaf nodes. A non-leaf node is called a composite or internal node. All tree nodes except leaf nodes are composites. The part of a tree containing a node and all of its descendents is called a subtree rooted by the node. Figure 7(b) shows the subtree rooted by node C. A node that is a child of another node is called a component. All tree nodes except the root node are components. There are no cycles in a tree, that is, if we follow the arrows from a node, we cannot return to the node. This is another way of saying that the structure is hierarchical. Each tree node is associated with a height which is the length of the longest path from it to a leaf node. Thus, the height of node G is 0, the height of node B is 1, and the height of node of node A is 3. The height of the tree is the height of its root. Similarly, a node is associated with a level, which is the length of the (unique) path to the root node. Thus, the levels of nodes A, C, E, and F are 0, 1, 2 and 3, respectively.
As mentioned before, the object arrays we considered above are trees. Consider the following extended example:

```java
static Object[] introProg = {new Integer(14), "Intro. Prog.", "COMP"};
static Object[] foundProg = {new Integer(114), "Found. Prog.", "COMP"};
static Object[] programming = {introProg, foundProg};
static Object[] algorithms = {new Integer(122), "Algorithms", "COMP"};
static Object[] compAnimation = {"Comp. Animation", "COMP"};
static Object[] otherCompSci = {algorithms, compAnimation};
static Object[] compSci = {programming, otherCompSci};
```

It can be described by the following tree representation:

**Tree-based recursion**

Our algorithm for printing all elements of the object array is an example of tree-based recursion. In general, a method implementing such recursion receives a tree node as a parameter. It processes the node, possibly doing different things for leaf and composite nodes. If the parameter node is a leaf node, the method returns. Otherwise, it calls itself recursively on the children of the composite. Figure 9 shows how this general scheme is followed in the print() method above.
Recursion

```
static void print (Object[] objArray) {
    System.out.print("{ ");
    for (int index = 0; index < objArray.length; index++) {
        Object element = objArray[index];
        if (element.getClass().isArray())
            print ((Object[]) element);
        else
            System.out.println(objArray[index]);
        if (index < objArray.length -1)
            System.out.print(" , ");
    }
    System.out.println("} ");
}
```

Figure 10 The print method as an example of tree-based recursion

**Tracing tree-based recursion**

Tracing tree-based recursion, let us trace the following call:
```
print(compSci)
```

The following table shows the output just before the function is called:

<table>
<thead>
<tr>
<th>Invocation</th>
<th>Tree node</th>
<th>Output created so far</th>
</tr>
</thead>
<tbody>
<tr>
<td>print()</td>
<td>compSci</td>
<td>{}</td>
</tr>
</tbody>
</table>

The first thing the function does is print a curly brace:

<table>
<thead>
<tr>
<th>Invocation</th>
<th>Tree node</th>
<th>Output created so far</th>
</tr>
</thead>
<tbody>
<tr>
<td>print()</td>
<td>compSci</td>
<td>{</td>
</tr>
</tbody>
</table>

Next it finds the first child of its parameter node, and calls itself recursively on this node:

<table>
<thead>
<tr>
<th>Invocation</th>
<th>Tree node</th>
<th>Output created so far</th>
</tr>
</thead>
<tbody>
<tr>
<td>print()</td>
<td>compSci</td>
<td>{</td>
</tr>
<tr>
<td>print()</td>
<td>programming</td>
<td></td>
</tr>
</tbody>
</table>

The second recursive call also prints a curly brace and recurses on its first child:

<table>
<thead>
<tr>
<th>Invocation</th>
<th>Tree node</th>
<th>Output created so far</th>
</tr>
</thead>
<tbody>
<tr>
<td>print()</td>
<td>compSci</td>
<td>{</td>
</tr>
<tr>
<td>print()</td>
<td>programming</td>
<td></td>
</tr>
<tr>
<td>print()</td>
<td>introProg</td>
<td></td>
</tr>
</tbody>
</table>

The third recursive call again prints a curly brace. However, after doing so, it does not make another recursive call. Since the elements of its parameter are leaf nodes, it chooses the non recursive path through
the loop, printing these elements with the separator ‘,’ between them, and finally printing the right curly brace before it returns.

<table>
<thead>
<tr>
<th>Invocation</th>
<th>Tree node</th>
<th>Output created so far</th>
</tr>
</thead>
<tbody>
<tr>
<td>print()</td>
<td>compSci</td>
<td>{</td>
</tr>
<tr>
<td>print()</td>
<td>programming</td>
<td>{</td>
</tr>
<tr>
<td>print()</td>
<td>introProg</td>
<td>{14, Intro. Prog., COMP}</td>
</tr>
</tbody>
</table>

Together, the three calls have created the following output so far:
{{14, Intro. Prog., COMP}}

The third call returns to the loop in the second call:

<table>
<thead>
<tr>
<th>Invocation</th>
<th>Tree node</th>
<th>Output created so far</th>
</tr>
</thead>
<tbody>
<tr>
<td>print()</td>
<td>compSci</td>
<td>{</td>
</tr>
<tr>
<td>print()</td>
<td>programming</td>
<td>{</td>
</tr>
</tbody>
</table>

The second call prints a comma followed by a space, and goes to the next loop iteration, which results in a recursive call to the next child of its parameter node.

<table>
<thead>
<tr>
<th>Invocation</th>
<th>Tree node</th>
<th>Output created so far</th>
</tr>
</thead>
<tbody>
<tr>
<td>print()</td>
<td>compSci</td>
<td>{</td>
</tr>
<tr>
<td>print()</td>
<td>programming</td>
<td>{</td>
</tr>
<tr>
<td>print()</td>
<td>foundProg</td>
<td>114, Found. Prog., COMP}</td>
</tr>
</tbody>
</table>

Again, the third call finds that the children of its parameter are leaf items, prints them:

<table>
<thead>
<tr>
<th>Invocation</th>
<th>Tree node</th>
<th>Output created so far</th>
</tr>
</thead>
<tbody>
<tr>
<td>print()</td>
<td>compSci</td>
<td>{</td>
</tr>
<tr>
<td>print()</td>
<td>programming</td>
<td>{</td>
</tr>
<tr>
<td>print()</td>
<td>foundProg</td>
<td>114, Found. Prog., COMP}</td>
</tr>
</tbody>
</table>

and returns to the loop in the second call:

<table>
<thead>
<tr>
<th>Invocation</th>
<th>Tree node</th>
<th>Output created so far</th>
</tr>
</thead>
<tbody>
<tr>
<td>print()</td>
<td>compSci</td>
<td>{</td>
</tr>
<tr>
<td>print()</td>
<td>programming</td>
<td>{</td>
</tr>
</tbody>
</table>

Since the second child has visited all of its children, it terminates its loop, prints a right curly brace:

<table>
<thead>
<tr>
<th>Invocation</th>
<th>Tree node</th>
<th>Output created so far</th>
</tr>
</thead>
<tbody>
<tr>
<td>print()</td>
<td>compSci</td>
<td>{</td>
</tr>
<tr>
<td>print()</td>
<td>programming</td>
<td>{, }</td>
</tr>
</tbody>
</table>
Recursion

and returns to the loop in the first call:

<table>
<thead>
<tr>
<th>Invocation</th>
<th>Tree node</th>
<th>Output created so far</th>
</tr>
</thead>
<tbody>
<tr>
<td>print()</td>
<td>compSci</td>
<td>{</td>
</tr>
</tbody>
</table>

At this point, the output is:

\[\{{14, Intro. Prog.,COMP}, {114,Found. Prog.,COMP}\}\]

The first call prints the separator, calls itself recursively on its second child, and this process continues until we get, at the end of the first call, the following output:

\[\{{14, Intro. Prog.,COMP}, {114, Found. Prog., COMP}, {122, Algorithms, COMP}, \{Comp. Animation, COMP\}\}\]

Indirect Recursion

In the print method above, we use iteration to go across the nodes at particular tree level, and recursion to go down a level. We could have used recursion to do both tasks – the loop would be replaced by method supporting list-based recursion. Thus, in a pure recursive solution, we need two different recursive methods, supporting list-based and tree-based recursion, respectively, as shown in Figure 8.

```java
static void
    print (Object[] objArray, int elementNo) {
        if (elementNo >= objArray.length)
            return;
        else {
            Object element = objArray[elementNo];
            if (element.getClass().isArray())
                print ((Object[]) element);
            else
                System.out.print(objArray[elementNo]);
            if (elementNo < objArray.length - 1)
                System.out.print (", ");
            print (objArray, elementNo + 1);
        }
    }

static void
    print (Object[] objArray) {
        System.out.print ("{");
        print (objArray, 0);
        System.out.print ("}"");
    }
```

The method:

```java
static void
    print(Object[] objArray, int elementNo)
```

is the one supporting the, by now familiar, index-based recursion. It returns when the tail of the list represented by its parameters is null. Otherwise, it prints the head and recurses on the tail. The method:

```java
static void
    print(Object[] objArray)
```

is the one supporting tree-based recursion. It is called by the previous method to go down one level in the tree. It is not as clear that it is a recursive method, as it does not call itself directly. As shown in Figure 11, it indirectly calls itself by calling the previous method, which in turn calls it. Methods such as these that indirectly recurse are called mutually recursive methods.

Figure 11 Mixing Tree and List-based and Direct and Indirect Recursion

The method:

```java
static void
    print(Object[] objArray, int elementNo)
```
Recursion

**Instance Method Recursion and Traces**

As a second example of tree-based recursion and to learn some important new concepts arising in such recursion, let us revisit our course example.

```java
static void fillCourses() {
    courses.addElement(new ARegularCourse("Intro. Prog.", "COMP", 14));
    courses.addElement(new ARegularCourse("Found. of Prog.", "COMP", 114));
    courses.addElement(new AFreshmanSeminar("Comp. Animation", "COMP"));
    courses.addElement(new AFreshmanSeminar("Lego Robots", "COMP"));
}
```

As shown in Figure 12, we created flat course lists, that is, course lists composed of courses. With this structure we were easily able to map a course title to a course in the list.

Let us now create an extension of this example that creates hierarchical course lists or course trees instead of flat lists, as shown in Figure 13.

**Figure 12 Flat Course List**

```java
static void fillCourses() {
    CourseList prog = new ACourseList();
    prog.addElement(new ARegularCourse("Intro. Prog.", "COMP", 14));
    prog.addElement(new ARegularCourse("Found. of Prog.", "COMP", 114));
    courses.addElement(prog);
    courses.addElement(new AFreshmanSeminar("Comp. Animation", "COMP"));
    courses.addElement(new AFreshmanSeminar("Lego Robots", "COMP"));
}
```

**Figure 13 Course Trees**

A course tree is composed of courses or course tree themselves. Thus, the leaf nodes of a course tree are courses. The problem now is to map a course title to a leaf node in a course tree.

Before we try to solve this problem consider why we might want hierarchical rather than flat course lists. Information may come in hierarchical form from organizations. For example, the programming and other groups submits courses to computer science department, the computer science and other departments submit courses to the school of arts and science, and the arts and science and other schools submit courses to the university. Moreover, a hierarchical course tree easily allows the answering of hierarchical queries such as “show me all computing science courses” or “show me all programming courses”. We will not attempt to support such queries here, pointing them out simply to motivate the problem.

What makes the new course list a tree is the ability to call addElement() with both courses and course lists as arguments. How should we offer this feature to the users of a course list? One alternative is to provide an addElement() that takes an arbitrary object as an argument:
Recursion

public void addElement(Object element);
However, such a method does not prevent illegal objects such as BMISpreadsheet to be added to the course list. Another alternative is to provide two different overloaded addElement() methods:

public void addElement(Course element);
public void addElement(CourseList element);

However, this raises an implementation question?. What should be the type of the array that stores the added elements. In the flat course list solution we declared it as:

Course[] contents = new Course[MAX_SIZE];

Ideally a want a single type T that unites both a course and a course list and can be used to declare the array and addElement() method. Such a type does not currently exist, so we must essentially define it as an interface that both objects implement. Let us call it TreeNode as both courses and course lists are nodes in the course tree. We don’t currently know what, if any, methods go into this interface, so let us define it currently as an empty interface:

package courseTree;
public interface TreeNode {
}

we will now make both Course and CourseList extend this interface so that instances of these two types also become instances of TreeNode:

package courses;
public interface Course extends TreeNode {
public String getTitle();
public String getDepartment();
public int getNumber();
public void init (String theTitle, String the Dept);
}

package collections;
public interface CourseList extends TreeNode {
public void addElement(TreeNode element);
public int size();
public Course elementAt(int index);
public Course matchTitle (String theTitle);
}

We now know how to type the array and addElement() parameter:

TreeNode[] contents = new TreeNode[MAX_SIZE];

The implementation of the addElement() method simply adds the element to the array in the manner it did before – in fact, its body does not change. Our next task is to look at the other methods of CourseList to see if they need re-implementation. The only method that needs to be changed is matchTitle() so let us focus now on it.

As a starting point, let us look at the implementation of the method in flat course lists:

public Course matchTitle (String theTitle) {
for (int courseIndex = 0; courseIndex < size; courseIndex++) {
    if (contents[courseIndex].getTitle().equals(theTitle))
    return contents[courseIndex];
}
return null;
}
Recursion

If we used this implementation in the hierarchical course lists, we would only match courses in level 1. So the trick is to make it recurse to the next level for list elements that are themselves lists. We can use the instanceof operation to distinguish courses from course lists:

```java
public Course matchTitle (String theTitle) {
    for (int courseIndex = 0; courseIndex < size; courseIndex++) {
        TreeNode element = contents[courseIndex];
        if (element instanceof Course) {
            if (((Course)element).getTitle().equals(theTitle))
                return (Course) element;
        } else { // instanceof CourseList
            Course course = ((CourseList) element).matchTitle(theTitle);
            if (course != null) return course;
        }
    }
    return null;
}
```

However, as mentioned before, using instanceof are not a good idea in non user-interface code. Instead of using this operation to execute different pieces of code for different kinds of instances, it is better to put each piece of code in a dynamically dispatched method in the class of the instances that trigger the execution of the code. The method must be declared in an interface common to the classes of the different kinds of instances. In our example, it should be put in TreeNode. Both pieces of code are matching the title in the leaf nodes in the subtree rooted by the node. So the key to removing the use of instanceof is to move matchTitle() to TreeNode and require even a course to implement it. Since this implementation is independent of whether the course is a freshman seminar or a regular course, it belongs to the common abstract class ACourse. Thus, we must now not only change the above implementation of the method in ACourseList but also provide one in ACourse.

The implementation in ACourseList is simpler than the one we had above:

```java
public Course matchTitle (String theTitle) {
    for (int courseIndex = 0; courseIndex < size; courseIndex++) {
        Course course = contents[courseIndex].matchTitle(theTitle);
        if  ( course != null) return course;
    }
    return null;
}
```

It goes through each child, looking for a match in the leaf nodes in the subtree rooted by the child. If it gets a match, it returns, otherwise it looks at the next node. It returns null if no match occurred in any of its subtrees.

The implementation in ACourse is even simpler:

```java
package courseTree;
public abstract class ACourse implements Course {
    public Course matchTitle(String theTitle) {
        if ( title.equals(theTitle))
            return this;
        else
            return null;
    }
    ...
}
```
Recursion

In case of a course, the subtree it roots consists only of that node. So all match title has to do is return the node in case of a successful match and return null otherwise.

To better understand how this method works, let us trace an execution of it. Assuming the following course tree:

```
courses
  |--- prog
  |    |--- new ARegularCourse
  |    |    |--- “Intro. Prog”, “COMP”, 14
  |    |--- new ARegularCourse
  |    |    |--- “Found. Prog.”, “COMP”, 114
  |    |--- new AFreshmanSeminar
  |    |    |--- “Comp. Animation”, “COMP”
  |    |--- new AFreshmanSeminar
  |    |    |--- “Lego Robots”, “COMP”
```

Let us trace the execution of:
```
courses.matchTitle(“Comp. Animation”)```

The following picture shows the first three calls to this function:

```
function      object          arg          result
matchTitle    courses         “Comp. Animation”
matchTitle    courses.contents[0]  “Comp. Animation”
matchTitle    courses.contents[0].contents[0]  “Comp. Animation”
```

At this point none of the calls has calculated a result. Unlike traces we have seen before, this one includes, in addition to the function name, arguments, and result, a column for object. This is because, unlike previous methods, this one is an instance method, and the object field indicates the target of the method call. Unlike the case in previous traces, some argument does not get smaller with each recursive call. Instead, the target of the instance method gets smaller. In fact, because the method is dynamically dispatched, it has multiple implementations; the first and second call to the method use the implementation in ACourseList while the third one uses the one in ACourse.

Let us continue the trace. The last call to the function calculates null as its result:

```
function            object          arg          result
matchTitle    courses         “Comp. Animation”
matchTitle    courses.contents[0]  “Comp. Animation”
matchTitle    courses.contents[0].contents[0]  “Comp. Animation”  null
```

and returns to the second call.

```
function            object          arg          result
matchTitle    courses         “Comp. Animation”
matchTitle    courses.contents[0]  “Comp. Animation”
```

This call then recurses on the second child of its target node, which also calculates null as its return value:
Recursion

and returns to the second call. The second call has now explored all of its children, so it completes its for loop, calculates null as its result:

and returns to the first call:

The first call recurses on the second child of its target, which is a leaf node whose title matches the argument. Hence the node is calculated as the result of this call:

When the first call receives the non null value returned by the recursive call, it returns this value, terminating the loop without looking at the third child of its target, thus completing the trace:

Visiting/Traversing Tree Nodes

A method visits or traverses a tree node when it accesses the (methods or instance variables in the) node. At any point in time, there can be multiple pending executions of the method, each accessing a different node. Only one of them, the last one invoked, is actually executing. It is this the node being accessed by this execution that is said to be visited at that point in time. We can abstract the trace above by simply tracing the different nodes visited during the execution.

Initially the root node is visited, as shown below:

Next it is the first child of the root:
Recursion

Then the first child of the first child of the root:

The method then returns to its parent with a null match.

The method then visits the second child of the parent:

Again it returns to the parent with a null match.

Since there are no more children of the current node, the method returns to revisit the root.
Recursion

It then visits the second child of the root:

This time the node returns with a non-null match to the root.

The recursion, thus, ends without visiting the last child of the root.

When you study data structures, you will learn various strategies for visiting nodes in a tree. For now, it is sufficient to understand the concept of visiting tree nodes.

**Recursively Creating a Tree**

In the example above, the tree visited recursively was created by non-recursive code. This was possible because the number of levels in the tree was fixed a program writing time. When this is not the case, we must create the tree also recursively (unless we implement our own stack). Let us look at a variation of the above problem to see how this may be done.

Let us assume that we have to convert from the array representation of a course list to the special CourseList object we created for storing the courses. The header of the method we want to write, thus, is:

```java
static CourseList courseArrayToCourseList (Object[] courseArray)
```

The following figure illustrates the role of the function.
Recursion

The top figure is an illustration of the array representation of a course tree. The figure on the bottom depicts the CourseList object we want to create from this tree.

Before we consider how the function may be implemented, let us explicitly state the rules used to create an array representation of a course list:

- A freshman seminar is represented by an object array of exactly two strings defining the course title and department, respectively.
- A regular course is represented by an object array of exactly three elements: the first one is an Integer object defining the course number and the other two are strings defining the title and department, respectively.
- A course tree is represented by arrays whose elements are array representations of freshman seminars, regular objects, and course trees.

We will assume that the argument to the recursive function is guaranteed to have an array representing a course tree. Thus the argument cannot be a representation of a freshman seminar, regular course, or some other arbitrary object array.

The recursive function will recursively visit nodes in the array tree to create nodes in the CourseList tree. The following figures illustrate this process for the example array representation shown above. When the function visits the root node of the representation, it creates a CourseList object with no elements.
Since the CourseList object has no children, there are no arrays emanating from it. Next the function visits the first child of the top tree, and again creates a CourseList object:

The new CourseList must become a child of the previous CourseList, as its position in the diagram indicates. The currently active recursive function execution, however, cannot perform this task as it does not have a reference to the previous CourseList. Its caller created this object thus has a reference to it. Therefore it is the caller that will perform this task when the recursive call returns.

The following figure shows the next array node visited and the object created:
Recursion

Since the currently visited node represents a regular course rather than a course tree, a RegularCourse rather than CourseList instance is created. Because the visited node has no more children, we now go back and revisit its parent, and add the RegularCourse to the CourseList we created when we first visited the parent:

The next task is to visit the second child of the array, and repeat the process of creating a RegularCourse:
Revisiting the parent, and making adding the RegularCourse to the second CourseList:

Since the currently visited node has no more children, we return to its parent, and connect the CourseList created for it to the one created for the parent:
Recursion

This next child of the root node is next visited:

This time, the function realizes that the array represents a freshman seminar, so it creates a FreshmanSeminar instance. This process continues until the remainder of the course tree is created.

Based on the discussion above, we need code to determine if an array encodes a regular course or a freshman seminar and if so to create from it a RegularCourse or FreshmanSeminar object, respectively. Let us write functions to perform these four tasks:

```java
static boolean isFreshmanSeminar (Object[] courseArray) {
    if (courseArray.length != 2) return false;
    if (courseArray[0] instanceof String) return true;
    else return false;
}
static boolean isRegularCourse (Object[] courseArray) {
    // Code to determine if the array encodes a regular course
    // and create a RegularCourse object
}
```
Recursion

```java
if (courseArray.length != 3) return false;
if (courseArray[0] instanceof Integer) return true;
else return false;
}
static FreshmanSeminar courseArrayToFreshmanSeminar (Object[] courseArray) {
    return (new AFreshmanSeminar((String) courseArray[0], (String) courseArray[1]));
}
static Course courseArrayToRegularCourse (Object[] courseArray) {
    return (new ARegularCourse ((String) courseArray[1], (String) courseArray[2], ((Integer) courseArray[0]).intValue()));
}

The last function shows how to convert between an Integer object and an int value.

We can now define our recursive function:

```java
// gets an array representing a set of courses rather than an individual course
static CourseList courseArrayToCourseList (Object[] courseArray) {
    if (isFreshmanSeminar(courseArray))
        return courseArrayToFreshmanSeminar(courseArray);
    else ifRegularCourse(courseArray))
        return courseArrayToRegularCourse(courseArray);
    else {
        CourseList courseList = new ACourseList();
        for (int index = 0; index < courseArray.length; index++) {
            Object[] element = (Object[]) courseArray[index];
            if (isFreshmanSeminar(element))
                courseList.addElement(courseArrayToFreshmanSeminar(element));
            else if (isRegularCourse(element))
                courseList.addElement(courseArrayToRegularCourse(element));
            else
                courseList.addElement(courseArrayToCourseList(element));
        }
        return courseList;
    }
}
```

Thu function visits each array node recursively, and when a recursive call returns, the caller adds the value returned to the CourseList node it created.

As it turns out, this function does not meet its requirements, which are shown in the comment before its header. The function returns a CourseList and is guaranteed to receive an array representing a course tree rather than an individual course. The Java compiler would complain that the values returned in the base cases are not CourseList objects. This would be the desired function if we were to allow the argument to be a representation of a course tree or an individual node and defined the return type to be TreeNode.

The trick to meeting the requirement is stop the recursion when the next node to be visited represents a freshman seminar or regular course, as shown below:

```java
// gets an array representing a set of courses rather than an individual course
static CourseList courseArrayToCourseList (Object[] courseArray) {
    CourseList courseList = new ACourseList();
    for (int index = 0; index < courseArray.length; index++) {
        Object[] element = (Object[]) courseArray[index];
        if (isFreshmanSeminar(element))
            courseList.addElement(courseArrayToFreshmanSeminar(element));
        else if (isRegularCourse(element))
            courseList.addElement(courseArrayToRegularCourse(element));
        else
            courseList.addElement(courseArrayToCourseList(element));
    }
    return courseList;
}
```
**Composite Design Pattern**

The relationships between the nodes in the CourseList tree follow a general pattern, called the Composite design pattern.

```java
package courseTree;
public interface TreeNode {
public Course matchTitle(String theTitle);
}
```

```java
package courseTree;
public interface Course extends TreeNode {
public String getTitle();
public String getDepartment();
public int getNumber();
}
```

```java
package courseTree;
public interface CourseList extends TreeNode {
public void addElement(TreeNode element);
public int size();
public Course elementAt(int index);
}
```

As we see from the figure above, a common interface, TreeNode, unites the leaf and composite nodes. The interfaces of these two kinds of nodes extend this interface by adding leaf-specific and composite-specific nodes respectively. This general pattern is shown below:

In general, it is possible for the leaf and/or composite interfaces to be empty as shown above depending on whether the two kinds of nodes need special operations. Moreover, there may be multiple kinds of composites and leaf nodes as shown below:

We have described the pattern so far in terms of interfaces and inheritance relationships among them. It is possible to describe it in terms of classes and interfaces and the inheritance and implementation relationships among them.
Now a composite and leaf node implements two interfaces, one capturing the common operations and another defining the node-specific operations which does not inherit from the common interface. In general, it is always possible to replace inheritance relationships among interfaces with extends relationships among classes and interfaces. We compare these two approaches in the next chapter. In either case, we rely on the ability to define interfaces. The following figure shows how the pattern can be implemented in a language such as C++ that has classes only.

This time, instead the leaf and composite classes implementing a common interface, the two classes subclass from a common abstract class that defines the common operations. In general, an abstract class can always take the place of an interface through interfaces should be the first choice, as we discussed earlier.

The common aspect in all these versions of the pattern is that there is a type (extended interface, unextended interface, abstract class) that defines operations common to all nodes, which can be implemented differently in different kinds of nodes to remove the need for performing the instanceof operations on code that operates on the tree.

**Composite Pattern in Toolkits**

A version of the class-only pattern appears in most toolkits, which allow us to create trees of “widgets” (user-interface objects). These trees are defined by containment relationships among the screen representation of these objects. The following figure shows this tree for an example user-interface. Here the boundaries of the various widgets are marked by rectangles. The outermost rectangle defines a frame in which there are two panels. The left panel contains two text fields and two buttons while the right panel is a leaf node.

Figure 15 shows the inheritance structure created for the widget classes shown in Figure 14. The methods common to all widget nodes are defined by the class Component. Methods common to composite nodes, that are defined by the class Container, which is a subclass of Component, with its own subclasses defining Panel and Frame-specific methods. Also subclasses of Component are TextField and Button, which implement leaf nodes. Unlike the case in our example, different instances of the same type can be both composites and leaf nodes. In particular, the left instance of a Panel is a non-leaf node while the right instance is a leaf node. We will all a type (class or interface) a composite if any instance of it can be a composite tree node. Such a type must define the functionality of a composite even if some instance of it is not actually a composite.
Recursion

Figure 14 Example Widget Tree

The following code shows how the tree of Figure 14 is created:

```java
Frame frame = new Frame("Points Plotter");
Panel control = new Panel();
frame.add(control);
frame.add(new Panel());
control.add(new Button("New Point"));
control.add(new Button("Quit");
control.add(new TextField("0");
control.add(new TextField("0");
```

The processes of creating this tree is very similar to the one used to create our course trees. Let us try to deduce the class in which the add method we used above is defined and the type of its argument. It is possible to add only to composite nodes, so the method must be defined in the Container class. It should be possible to add any component to a container. Therefore its argument type should be Component.

This method is an example of a Container-specific operation not supported by leaf nodes such as a TextField. Conversely, getText() and setText() are examples of methods that apply to a TextField but not to a Containers or a Button. More interesting are dynamically dispatched methods defined in the abstract class Component. An example is the paint() method, which can be implemented differently by each subclass of Component. A Container can recursively implement its paint() method in terms of the paint() methods of its components, without being aware of how these methods are actually implemented. As we add new kinds of component classes, we don’t have to change the classes of the containers in which these components are added, as the container classes do not perform the infamous instanceof operation. Yet another example to show the power of dynamic dispatch and the Composite design pattern!

Grammars and structure-based Recursion

We started this chapter with grammars. Let us end it by tying them to recursive methods that operate on lists and trees. Both lists and trees are structured objects in that these are objects that can contain components.

Recall that a grammar creates rules mapping non-terminals to terminals. For example, the grammar:

```
<Noun Phrase>   →   <Noun>
<Noun Phrase>   →   <Adjective> <Noun Phrase>
```

tells us the sequences of words, which are the terminals in this example, to which the non-terminal <Noun Phrase> can be mapped. There can be multiple rules defining the same non-terminal, as shown above, which serve as alternate definitions. Thus, the above grammar say that a noun phrase is either a noun or an adjective followed by a noun phrase. Not all mappings between non-terminals and terminals are provided by the grammar. For example, the grammar does not specify the mapping between the non-terminal <Noun> and actual terminals such as “boy” and “girl”. It is assumed that these mappings are well known, that is, implicitly defined by some mechanism other than the grammar. A non-terminal is mapped to a sequence of
terminals if we can derive the sequence from the non-terminals using the explicit and implicit rules of the grammar. We saw earlier how we could derive word sequences such as “smart little boy” from the non-terminal \textless \textit{Noun Phrase}\textgreater.

What does this have to do with recursive methods? As we see above, both recursive methods and grammars use the concept of recursion to define entities. More important, often the way a recursive method decomposes a structure object on which it operates (such as a list or tree) can be defined by a grammar.

Consider a list of items. We informally define it as an ordered list of items. We can define it recursively using the following grammar:

\[
\textless \text{List}\rangle \rightarrow \text{empty} \\
\textless \text{List}\rangle \rightarrow \textless \text{Item}\rangle \textless \text{List}\rangle
\]

The first rule says that the list can be an empty sequence and the second one, recursively, says that it is an item followed by a list. We leave the mapping between \textless \text{Item}\rangle and actual list elements undefined, and this mapping would be different from different element types such as int, double and so on. The following figure shows how the \textless \text{List}\rangle non-terminal can derive various kinds of int lists:

\[
\begin{array}{c}
\text{[ 6 ]} \\
\text{[ 6 5 ]}
\end{array}
\]

\[
\begin{array}{c}
\text{[ empty ]} \\
\text{[ 5 empty ]}
\end{array}
\]

The derivations used here are similar to the one we used for noun phrases earlier. What is interesting here is that they parallel the way our list-based recursive methods decomposed their list arguments.

```
static void print (StringEnumeration stringEnumeration) {
    if (!stringEnumeration.hasMoreElements()) return;
    else {
        System.out.println(stringEnumeration.nextElement());
        print (stringEnumeration);
    }
}
```

\[
\begin{array}{c}
\text{\textless \text{List}\rangle \rightarrow \text{empty}} \\
\text{\textless \text{List}\rangle \rightarrow \textless \text{Item}\rangle \textless \text{List}\rangle}
\end{array}
\]

The two cases of the conditional used in the method correspond to the two rules defining its argument – when the argument is empty the method does one thing and when it is an item followed by a list, it does another thing. In general different ways of decomposing a structure in a method, (reflected by a conditional such as if or switch) correspond to different grammar rules defining the structure.
Recursion

A `<List>` could also have been defined in a non-recursive way:

```
<List> → <Item>*
```

The “*” above is called Kleene * and denotes an arbitrary number of occurrences (including 0) of the element to which it is attached. Thus, the rule above says a list consists of sequences of zero or more items. Since this way of decomposing is non-recursive, it corresponds to a method that uses a loop to iterate over the elements:

```java
static void print(StringHistory strings) {
    System.out.println("******************");
    StringEnumeration stringEnumeration = strings.elements();
    while (stringEnumeration.hasMoreElements()) {
        System.out.println(stringEnumeration.nextElement());
    }
    System.out.println("******************");
}
```

```
<List> → <Item>*
```

Both implementations work on the same argument, a flat list, but decompose it differently. The corresponding grammar describes the tree of method calls made to process the argument. In one case, a single method call is made while in the second case a sequence of recursive method calls is made which corresponds to the recursive grammar derivation of the argument.

There are two advantages of tying a method implementation to a grammar. First, a grammar is a more abstract and hence concise description of the method that applies to all other methods that decompose an object in the same way, even though they may do different things in the base and recursive steps. In particular, the grammar above describes the structure of all list-based recursive methods we have seen here. Thus, it serves as documentation of a method and a high-level step before the method is implemented. More important, sometimes the grammar is given, in which case it can guide the implementation. For example, the grammar of Java programs can be given to us, which can then guide a recursive processing of these programs.

Consider now alternate ways of describing tree nodes. One approach is to use a mixed representation combining the Kleene * with recursive rules:

```
<TreeNode> → <TreeNode>*
<TreeNode> → <Item>
```

The first rule says that a tree node is a list of zero or more children nodes, and thus defines a composite node. The second one says that it is an atomic item, and thus defines a leaf node. Figure 16 shows how these rules can be used to derive a tree represented as recursive lists. Figure 17 shows the correspondence between this grammar and the version of the print method we wrote that mixed loops with recursion.
Recursion

The following grammar describes the print solution that used only recursion.

```
<TreeNode> → <TreeNodeList>
<TreeNode> → <Item>
<TreeNodeList> → empty
<TreeNodeList> → <TreeNode><TreeNodeList>
```

It essentially combines the two grammars above, replacing `<TreeNode>*` in the second grammar with rules describing a list of `<TreeNode>` elements.

```
static void print (Object[] objArray) {
    System.out.print("{ ");
    for (int index = 0; index < objArray.length; index++) {
        Object element = objArray[index];
        if (element.getClass().isArray())
            print ((Object[]) element);
        else
            System.out.print(objArray[index]);
        if (index < objArray.length -1)
            System.out.print(" , ");
    }
    System.out.print (" }");
}
```

Figure 16 Example derivation

Figure 17 Relating method implementation to grammar

Figure 18 Pure recursive decomposition of a tree

Figure 17 uses this grammar for the example we used in Figure 16. As we see from both the rules and the derivation, the rules describing a tree node list are directly recursive, a tree node list is decomposed directly into another tree node list. The rules describing a tree node, on the other hand, are indirectly recursive. A tree node is defined in terms of a tree node list, which is then defined in terms of a tree node. This was exactly how our purely recursive print solution was structured, and the figure below shows the relationship between the solution and the grammar:
Recursion

As mentioned before, sometimes a grammar comes before the corresponding methods. Consider a grammar describing simple expressions you evaluated in assignment 3:

<Expression> → <Number>
<Expression> → <Number> <Operator> <Expression>

The following figure shows how it can be used to derive the expression:

\[ 5 + 4 \times 5 \]

For extra credit, redo assignment 3 by writing a recursive expression evaluation function that corresponds to this grammar.

Nested expressions, that is, expressions containing subexpressions marked by parentheses can be described as follows:

<Expression> → <Number>
<Expression> → (<Expression>) <Operator> <Expression>
<Expression> → (<Expression>)

Grammars are covered in more depth in courses on compilers and theory of computation. Here we have looked at them as a tool for developing recursive methods.

**Summary**

Recursion offers an alternative to loops that is considered more elegant and less error-prone, since the solution to the problem is the same as its definition.
Recursion

- Each recursive method must have a recursive reduction step, which solves the problem in terms of the same problem of a smaller size; and one or more terminating base cases, which solve it for the smallest-size version(s) of the problems.
- Successive calls to reduced versions of the problem get stacked until a base case is reached, after which they get unstacked as reduced versions successively return their results to their callers.
- Each stacked call gets its own copies of the formal parameters and return value from an area of memory called the stack.
- When a recursive method has multiple parameters, we must carefully choose the parameters whose size we will reduce in the recursive step.
- Like functions, procedures can also be recursive. While recursive functions create recursive expressions, recursive procedures create recursive statements.
- Number and list-based problems are particularly amenable for a recursive solution. In the former, the reduced problem involves a smaller number; while in the latter, it involves a smaller list.

**Exercises**

1. Trace the following two calls. The number of rows created for tracing the calls may be less than the number of actual calls:
   a) f(3)

   ```java
   public static int f(int n) {
       if (n <= 1)
           return 0;
       else if (n % 2 == 0)
           return f(n - 1) + n;
       else
           return f(n - 1) – n;
   }
   ``

<table>
<thead>
<tr>
<th>Invocation</th>
<th>n</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>f (3)</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

   b) g(3,2)

   ```java
   public static int g(int n1, int n2) {
       if (n1 <= 0)
           return n2;
       else if (n2 <= 0)
           return n1;
       else
           return f (n1 – 1, n2 – 1);
   }
   ``

<table>
<thead>
<tr>
<th>Invocation</th>
<th>n1</th>
<th>n2</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>g (3, 2)</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


Recursion

2. Trace the call to each of the following definitions of the recursive procedure, p by showing the output produced by each call. Assume that the user enters the four lines:

```
hello
hello
goodbye
goodbye
```

a) ```
   public static void p (){
      if (Keyboard.readString().equals("hello")) {
         System.out.println ("ca va");
         p();
      }
   }
```  

<table>
<thead>
<tr>
<th>Invocation</th>
<th>Output produced</th>
</tr>
</thead>
<tbody>
<tr>
<td>p ()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

b) ```
   public static void p (){
      if (Keyboard.readString().equals("hello")) {
         p();
         System.out.println ("ca va");
      }
   }
```  

<table>
<thead>
<tr>
<th>Invocation</th>
<th>Output produced</th>
</tr>
</thead>
<tbody>
<tr>
<td>p ()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c) 
   public static void p (){
      if (Keyboard.readString().equals("hello")) {
      }
      System.out.println ("ca va");
      p();
   }
```  

<table>
<thead>
<tr>
<th>Invocation</th>
<th>Output produced</th>
</tr>
</thead>
<tbody>
<tr>
<td>p ()</td>
<td></td>
</tr>
</tbody>
</table>
Recursion

3. In this problem, you will solve a famous mathematical problem. The problem is to compute elements of the Fibonacci sequence. The first two elements of the sequence are 1 and 1 and a subsequent element is the sum of the two previous elements. Thus, the element of the sequence are:

1 1 2 3 5 8 13 ....

Your program will take as input a number \( n \) and output the \( n \)th element of the Fibonacci sequence. Thus, if the input is 2, the output should be 1; and if the input is 7, the output should be 13. Do not use a loop; instead write a recursive function.

4. Extend your solution to Q3 by allowing the user to enter a series of values for \( n \) ending with a 0 or a negative number. Output the value of Fibonacci(\( n \)) for each user input, as shown below. Do not write a use a loop to process the list; write a recursive procedure instead.
Recursion