Announcements

- Final exam: April 20th at 9:00am, at the GYM
- Review tutorial: April 12th from 4:00pm to 6:00pm at ENGMC 13
- Course evaluations on Minerva (before April 13th:)
  - Login to **Minerva for students** (from `http://www.mcgill.ca`)
  - Select **Student Menu** (or a pop-up window will appear)
  - Click on **MOLE** - McGill Online Evaluations
  - Select COMP-202
  - Fill out the evaluation (it's anonymous.)

Recursion

- A recursive method is a method that calls itself (directly or indirectly.)
- A recursive definition is a definition of something in terms of itself
- Some recursive definitions don't make sense, (e.g. from Webster's: growl: to utter a growl), but others do

Recursive functions

- Factorial: the factorial of a natural number n, written $n!$ is the multiplication of the first n positive integers, i.e.

$$n! = \underbrace{1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-2) \cdot (n-1)}_{(n-1)!} \cdot n$$

Recursive functions (contd.)

Summarizing:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \cdot n & \text{otherwise} \end{cases}$$

Recursive functions (contd.)

Summarizing:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \cdot n & \text{otherwise} \end{cases}$$

This can be implemented as a static recursive method:

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

Execution of recursive methods

- Consider the following client for this factorial function:

```
int r;
r = factorial(4);
```

Execution of recursive methods

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

- Execution proceeds as follows:

```
factorial(4)
return factorial(4-1) * 4
```

Execution of recursive methods

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

- Execution proceeds as follows:

```
factorial(4)
return factorial(3) * 4
```

Execution of recursive methods

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

- Execution proceeds as follows:

```
factorial(4)
return factorial(3) * 4
return (factorial(3-1) * 3) * 4
```

Execution of recursive methods

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

- Execution proceeds as follows:

```
factorial(4)
return factorial(3) * 4
return (factorial(2) * 3) * 4
```

Execution of recursive methods

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

- Execution proceeds as follows:

```
factorial(4)
return factorial(3) * 4
return (factorial(2) * 3) * 4
return ((factorial(2-1) * 2) * 3) * 4
```

Execution of recursive methods

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

- Execution proceeds as follows:

```
factorial(4)
return factorial(3) * 4
return (factorial(2) * 3) * 4
return ((factorial(1) * 2) * 3) * 4
```

Execution of recursive methods

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

- Execution proceeds as follows:

```
factorial(4)
return factorial(3) * 4
return (factorial(2) * 3) * 4
return ((factorial(1) * 2) * 3) * 4
return (((factorial(1-1) * 1) * 2) * 3) * 4
```

Execution of recursive methods

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

- Execution proceeds as follows:

```
factorial(4)
return factorial(3) * 4
return (factorial(2) * 3) * 4
return ((factorial(1) * 2) * 3) * 4
return (((factorial(0) * 1) * 2) * 3) * 4
```

Execution of recursive methods

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

- Execution proceeds as follows:

```
factorial(4)
return factorial(3) * 4
return (factorial(2) * 3) * 4
return ((factorial(1) * 2) * 3) * 4
return (((1 * 1) * 2) * 3) * 4
```

Execution of recursive methods

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

- Execution proceeds as follows:

```
factorial(4)
return factorial(3) * 4
return (factorial(2) * 3) * 4
return (((1) * 2) * 3) * 4
```

Execution of recursive methods

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

- Execution proceeds as follows:

```
factorial(4)
return factorial(3) * 4
return ((2) * 3) * 4
```

Execution of recursive methods

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

- Execution proceeds as follows:

```
factorial(4)
return (6) * 4
```

Execution of recursive methods

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

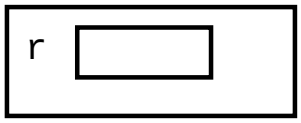
- Execution proceeds as follows:

```
factorial(4)
return 24
```

Execution of recursive methods

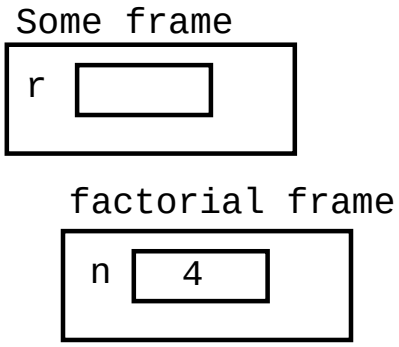
This is executed in some frame:

Some frame



Execution of recursive methods

When we call `factorial(4);` a new frame for the method is created:



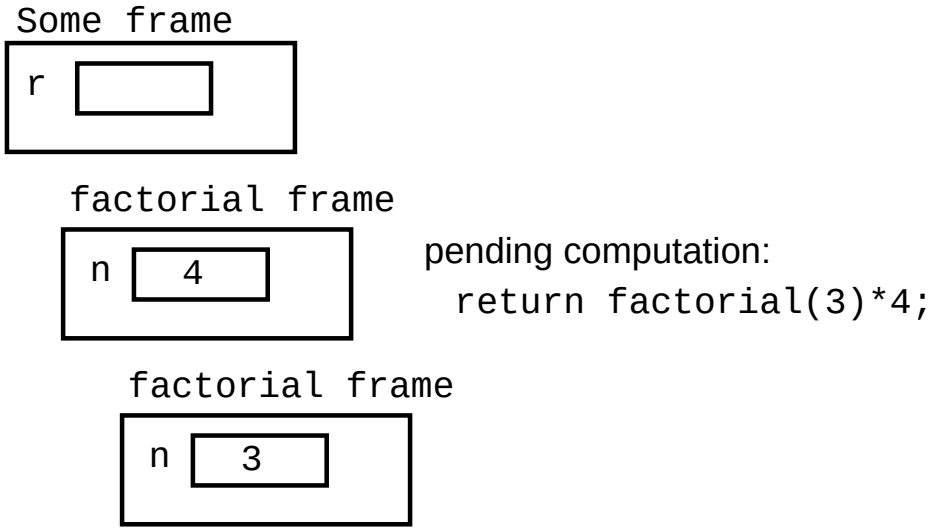
We execute the body of factorial; n is not 0 so we execute

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

which in this frame is the same as

```
return factorial(4-1)*4;
```

Execution of recursive methods



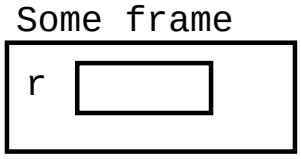
Again, we execute the body of factorial;
again, n is not 0 so we execute

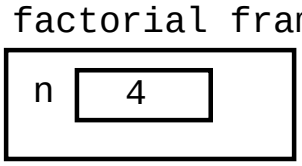
```
return factorial(n-1)*n;
```

which in this frame is the same as

```
return factorial(3-1)*3;
```

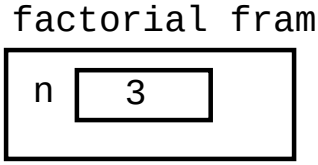
Execution of recursive methods





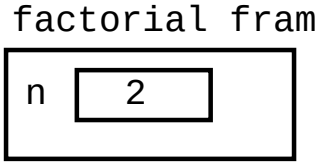
pending computation:

```
return factorial(3)*4;
```



pending computation:

```
return factorial(2)*3;
```



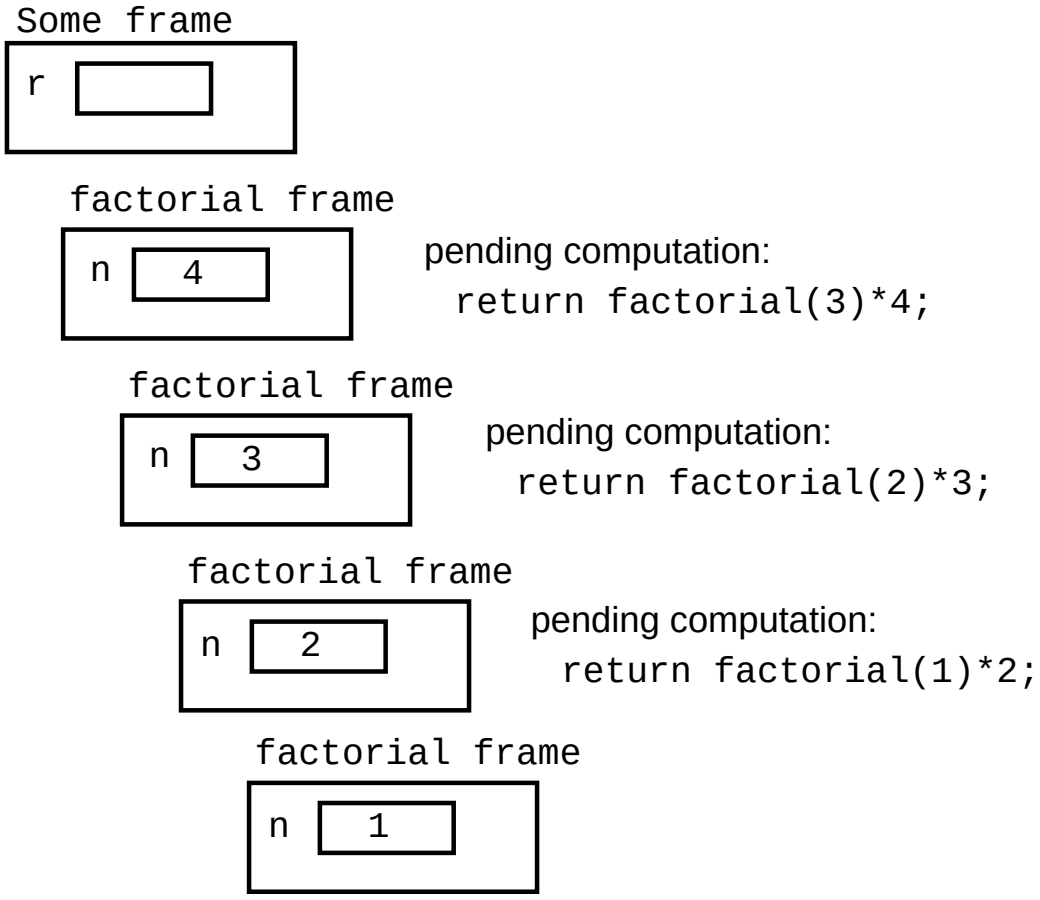
Again, we execute the body of factorial;
again, n is not 0 so we execute

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

which in this frame is the same as

```
return factorial(2-1)*2;
```

Execution of recursive methods



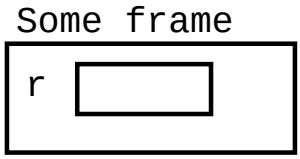
Again, we execute the body of factorial;
again, n is not 0 so we execute

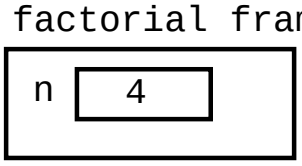
```
return factorial(n-1)*n;
```

which in this frame is the same as

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

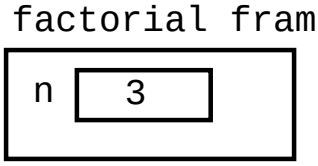
Execution of recursive methods





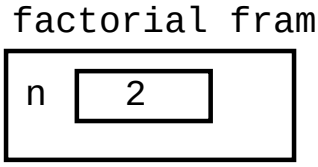
pending computation:

```
return factorial(3)*4;
```



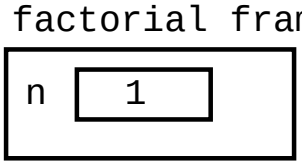
pending computation:

```
return factorial(2)*3;
```



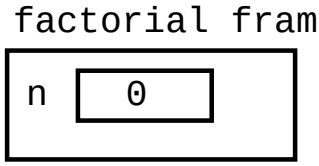
pending computation:

```
return factorial(1)*2;
```



pending computation:

```
return factorial(0)*1;
```

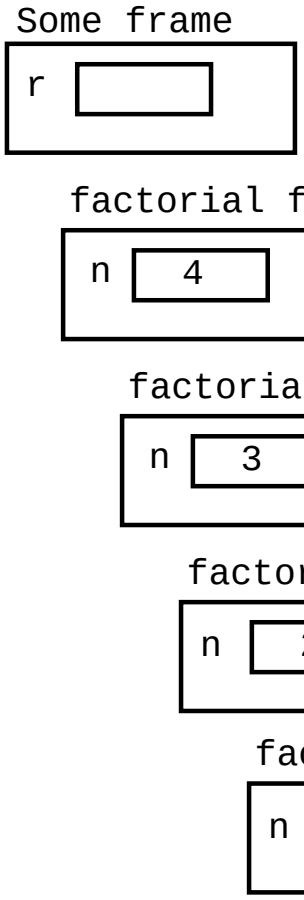


Now, we have reached the base case, and n is 0, so we execute:

```
return 1;
```

We get rid of the frame, and pass the returned value to the caller

Execution of recursive methods



The pending computation here was:

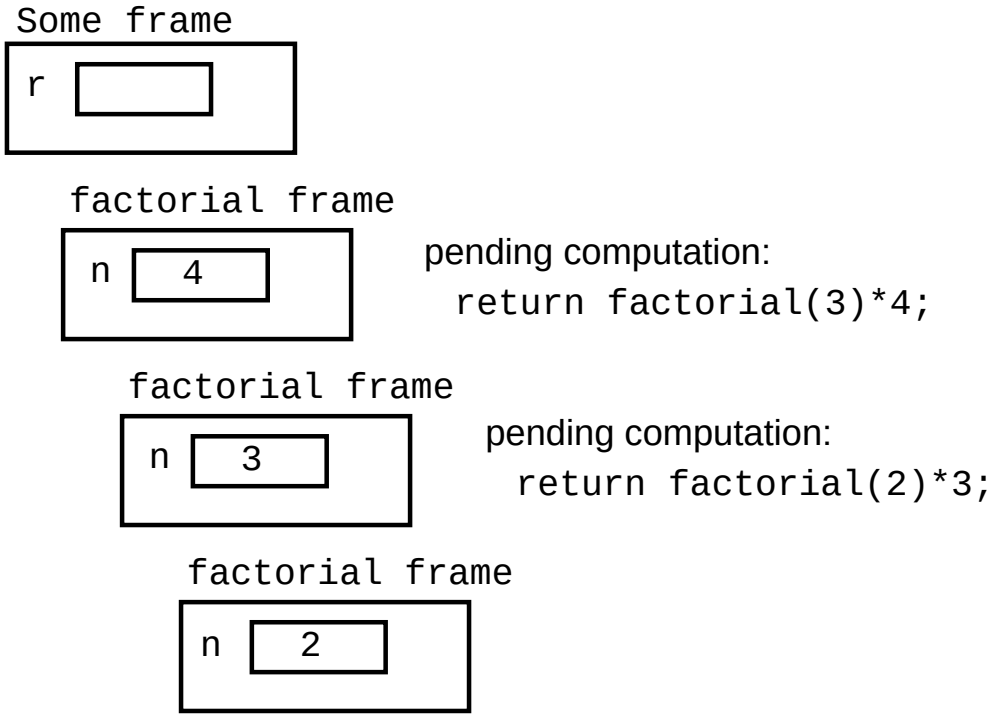
```
return factorial(0)*1;
```

and the method called `factorial(0)` returned 1, so this pending computation is now:

```
return 1*1;
```

We get rid of the frame, and pass the returned value to the caller

Execution of recursive methods



The pending computation here was:

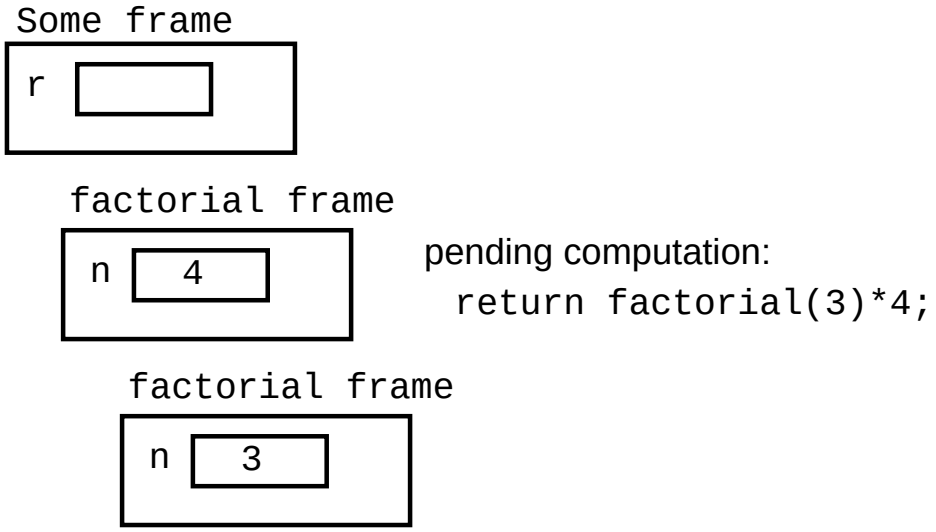
```
return factorial(1)*2;
```

and the method called `factorial(1)` returned 1, so this pending computation is now:

```
return 1*2;
```

We get rid of the frame, and pass the returned value to the caller

Execution of recursive methods



The pending computation here was:

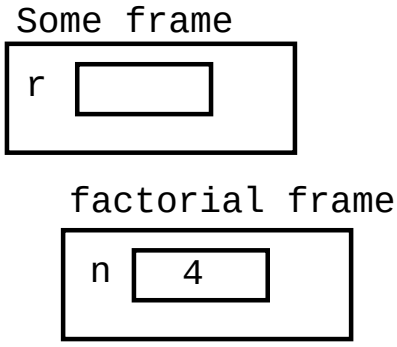
```
return factorial(2)*3;
```

and the method called `factorial(2)` returned 2, so this pending computation is now:

```
return 2*3;
```

We get rid of the frame, and pass the returned value to the caller

Execution of recursive methods



The pending computation here was:

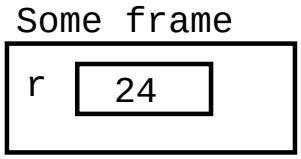
```
return factorial(3)*4;
```

and the method called `factorial(3)` returned 6, so this pending computation is now:

```
return 6*4;
```

We get rid of the frame, and pass the returned value to the caller

Execution of recursive methods



The pending computation here was:

```
r = factorial(4);
```

which returned 24, so this pending computation is now:

```
r = 24;
```

Reverse

```
public class MoreStringOperations
{
  static String reverse(String s)
  {
    if (s.equals(“”)) {
      return “”;
    }
    return reverse(rest(s))+s.charAt(0);
  }
  static String rest(String s)
  {
    String result =“”;
    int i = 1;
    while (i < s.length()) {
      result = result + s.charAt(i);
      i++;
    }
    return result;
  }
}
```

Double recursion

- Problem: Compute the n-th Fibonacci number
- Analysis: The Fibonacci sequence 1, 1, 2, 3, 5, 8, 13, 21, 34, ...is defined by:

$$fib(n) = \begin{cases} 1 & \text{if } n \leqslant 2 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

- Implementation:

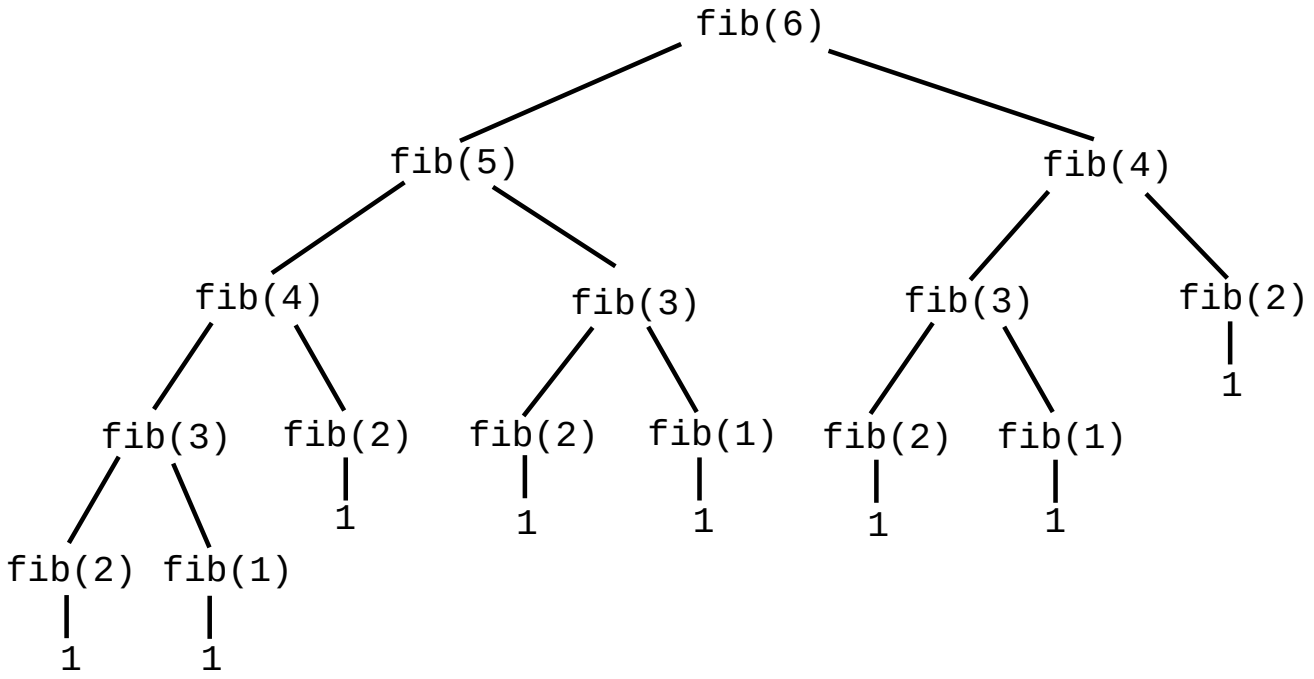
```
static int fib(int n)
{
    if (n <= 2) {
        return 1;
    }
    return fib(n - 1) + fib(n - 2);
}
```

Iteration vs recursion

- Iterative solution to the Fibonacci problem:

```
static int fib(int n)
{
    int a, b, c, i;
    a = 1;
    b = 1;
    c = 1;
    i = 3;
    while (i <= n)
    {
        c = a + b;
        a = b;
        b = c;
        i++;
    }
    return c;
}
```

Execution trees



Divide and Conquer

- To solve a problem:

1. Divide it into smaller sub-problems
2. Solve each subproblem separately
3. Combine the solution of the subproblems

Recursion

Problem: write a method that given a double b and a positive integer n, computes b^n

Recursion

$$b^n = \underbrace{b \cdot b \cdot b \cdot \cdots \cdot b \cdot b}_{n \text{ times}}$$

Recursion

$$b^n = b \cdot \underbrace{b \cdot b \cdot \cdots \cdot b \cdot b}_{n-1 \text{ times}}$$

Recursion

$$b^n = b \cdot b^{n-1}$$

Recursion

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{if } n > 0 \end{cases}$$

Recursion

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{if } n > 0 \end{cases}$$

```
double pow(double b, int n)
{
  if (n == 0) return 1;
  return b * pow(b, n - 1);
}
```

Recursion

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{if } n > 0 \text{ and } n \text{ is odd} \\ (b^{n/2})^2 & \text{if } n > 0 \text{ and } n \text{ is even} \end{cases}$$

Recursion

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{if } n > 0 \text{ and } n \text{ is odd} \\ (b^{n/2})^2 & \text{if } n > 0 \text{ and } n \text{ is even} \end{cases}$$

```
double fastpow(double b, int n)
{
  if (n == 0) return 1;
  if (n % 2 == 1) return b * fastpow(b, n - 1);
  double p = fastpow(b, n/2);
  return p * p;
}
```

Recursion

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{if } n > 0 \text{ and } n \text{ is odd} \\ (b^2)^{n/2} & \text{if } n > 0 \text{ and } n \text{ is even} \end{cases}$$

```
double fastpow(double b, int n)
{
  if (n == 0) return 1;
  if (n % 2 == 1) return b * fastpow(b, n - 1);
  return fastpow(b * b, n / 2);
}
```

Recursion and termination

```
static void f(int n)
{
  System.out.println(n);
  f(n);
}
```

Recursion and termination

```
f(5)
f(5)
f(5)
f(5)
.
.
.
```

Recursion and termination

```
static int f(int n)
{
  System.out.println(n);
  return f(n) + 1;
}
```

Recursion and termination

```
f(5)
return f(5) + 1
return (f(5) + 1) + 1
return ((f(5) + 1) + 1) + 1
.
.
.
```

Recursion and termination

```
static int f(int n)
{
  System.out.println(n);
  return f(n-1);
}
```

Recursion and termination

```
f(5)
return f(4)
return f(3)
return f(2)
return f(1)
return f(0)
return f(-1)
.
.
.
```

Recursion and termination

```
static int f(int n)
{
  if (n == 0) return 1;
  System.out.println(n);
  return f(n);
}
```

Recursion and termination

```
f(5)
return f(5)
return f(5)
return f(5)
return f(5)
return f(5)
return f(5)
.
.
.
```

Recursion and termination

```
static int f(int n)
{
  if (n == 0) return 1;
  System.out.println(n);
  return f(n-1);
}
```

Recursion and termination

```
f(5)
return f(4)
return f(3)
return f(2)
return f(1)
return f(0)
return 1
```

Recursion and termination

```
static int f(int n)
{
  if (n == 0) return 1;
  System.out.println(n);
  return f(n - 2) + 1;
}
```

Recursion and termination

```
static int f(int n)
{
  if (n == 0) return 1;
  System.out.println(n);
  return f(n / 2) + 1;
}
```

Recursion and termination

```
static int f(int n)
{
  if (n == 0) return 1;
  System.out.println(n);
  return f(n + 2);
}
```

Recursion and termination

```
f(5)
returns f(7)
returns f(9)
returns f(11)
returns f(13)
.
.
.
```

Recursion and termination

- For a recursive method to terminate it is necessary:
  - to have at least one base case
  - the recursive method call must change the parameters so that the subproblem is "smaller" than the original problem

Recursion and termination

```
static int f(int n)
{
  if (n == 0) return 1;
  else if (n % 2 == 0) return n / 2;
  else return f( f( 3 * n + 1 ) );
}
```

Recursion and termination

If n is odd then the third case applies, and we have to compute

$$f(f(3n+1))$$

But, since n is odd, then $n = 2k+1$ for some $k \geq 0$, which implies that

$$\begin{aligned} 3n+1 &= 3(2k+1)+1 \\ &= 6k+3+1 \\ &= 6k+4 \\ &= 2(3k+2) \end{aligned}$$

so $3n+1$ is even!

Therefore

$$f(3n+1) = \frac{3n+1}{2}$$

so we can rewrite

$$f(f(3n+1))$$

as

$$f(\frac{3n+1}{2})$$

Recursion and termination

```
static int f(int n)
{
  if (n == 0) return 1;
  else if (n % 2 == 0) return n / 2;
  else return f( (3 * n + 1) / 2 );
}
```

Recursion and termination

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n/2 & \text{if } n \text{ is even} \\ f(\frac{3n+1}{2}) & \text{if } n \text{ is odd} \end{cases}$$

Question: for all $n > 0$, is $\frac{3n+1}{2} < n$?

Answer: no! For $n > 0$

$$3n > 2n$$

so

$$3n + 1 > 2n$$

therefore

$$\frac{3n+1}{2} > n$$

Hence the argument of the recursive call increases!

Does this mean, then that f cannot terminate?

No! eventually $\frac{3n+1}{2}$ will be even, thus reaching a base case (but the proof is hard!)

Recursion and termination

Determining whether a program might terminate is in general hard

Furthermore, it is impossible to write a program that takes as input any program P and decides whether P terminates or not.

Normal Recursion

```
void p(...)
{
   ...
   p(...);
   ...
}
```

Indirect Recursion

```
void p(...)
{
  ...
  q(...);
  ...
}
void q(...)
{
  ...
  r(...);
  ...
}
void r(...)
{
  ...
  p(...);
  ...
}
```

Mutual Recursion

```
void p(...)
{
  ...
  q(...);
  ...
}
void q(...)
{
  ...
  p(...);
  ...
}
```

Recursion on two arguments

```
int h(int x, int y)
{
  if (x == 1) return 2;
  if (y == 1) return 3 * x;
  return h(x - 1, y) + h(x, y - 1);
}
```

Recursion on two arguments

```
int h(int x, int y)
{
  if (x == 1) return 2;
  if (y == 1) return 3 * x;
  return h(x - 1, y) + h(x, y - 1);
}
```

x\y	1	2	3	4	5
1	2	2	2	2	2
2					
3					
4					

Recursion on two arguments

```
int h(int x, int y)
{
  if (x == 1) return 2;
  if (y == 1) return 3 * x;
  return h(x - 1, y) + h(x, y - 1);
}
```

x\y	1	2	3	4	5
1	2	2	2	2	2
2	6				
3	9				
4	12				

Recursion on two arguments

```
int h(int x, int y)
{
  if (x == 1) return 2;
  if (y == 1) return 3 * x;
  return h(x - 1, y) + h(x, y - 1);
}
```

x\y	1	2	3	4	5
1	2	2	2	2	2
2	6	8	10	12	14
3	9	17	27	39	53
4	12	29	56	95	148

Recursion on two arguments

```
int h(int x, int y)
{
  if (x == 1) return 2;
  if (y == 1) return 3 * x;
  return h(x - 1, y) + h(x, y - 1);
}
```

x\y	1	2	3	4	5
1	2	2	2	2	2
2	6	8	10	12	14
3	9	17	27	39	53
4	12	29	56	95	148

Recursion and termination

Ackermann's function:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \text{ and } n \geqslant 0 \\ A(m-1,1) & \text{if } m \geqslant 1 \text{ and } n = 0 \\ A(m-1,A(m,n-1)) & \text{if } m \geqslant 1 \text{ and } n \geqslant 1 \end{cases}$$

m\n	0	1	2	3	4
0	1	2	3	4	5
1	2	3	4	5	6
2	3	5	7	9	11
3	5	13	29	61	125

Recursion and termination

Ackermann's function:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \text{ and } n \geqslant 0 \\ A(m-1,1) & \text{if } m \geqslant 1 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m \geqslant 1 \text{ and } n \geqslant 1 \end{cases}$$

m\n	0	1	2	3	4	n
0	1	2	3	4	5	n+1
1	2	3	4	5	6	n+2
2	3	5	7	9	11	2n+3
3	5	13	29	61	125	$2^{(n+3)} - 3$
4	13	65533	$2^{65536} - 3$			$2^{2^{\cdots^2}} - 3$

A(4,2) is greater than the number of particles in the universe raised to the power 200

A(5,2) cannot be written as a decimal expansion in the physical universe.

Recursion and termination

```
static int ackermann(int m, int n)
{
  if (m == 0 && n >= 0)
    return n + 1;
  else if (m >= 1 && n == 0)
    return ackermann(m - 1, 1);
  else
    return ackermann(m - 1, ackermann(m, n - 1));
}
```

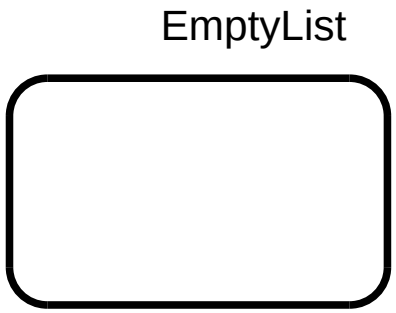
Recursive data-structures

- For example:
  - A *list of numbers* is either:
    * A single number, or
    * A number followed by a list of numbers.
  - For example:
    * 5 is a list of numbers
    * 7, 5 is a list of numbers (because 5 is a list)
    * 6, 7, 5 is a list of numbers (because 7, 5 is a list)
    * 8, 6, 7, 5 is a list of numbers (because 6, 7, 5 is a list)

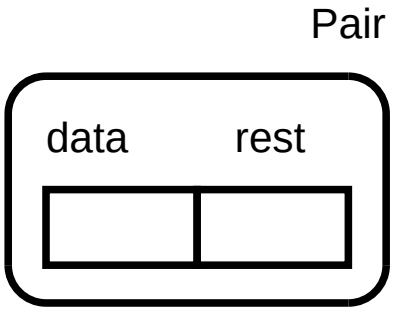
Recursive data-structures

- For example:
  - A *list of data* is either:
    - An *empty list* [], or
    - A *pair* consisting of:
      - Some data, and
      - A list of data.
  - For example:
    - [] is a list
    - [5, []] is a list
    - [7, [5, []]] is a list
    - [6, [7, [5, []]]] is a list
    - [8, [6, [7, [5, []]]]] is a list

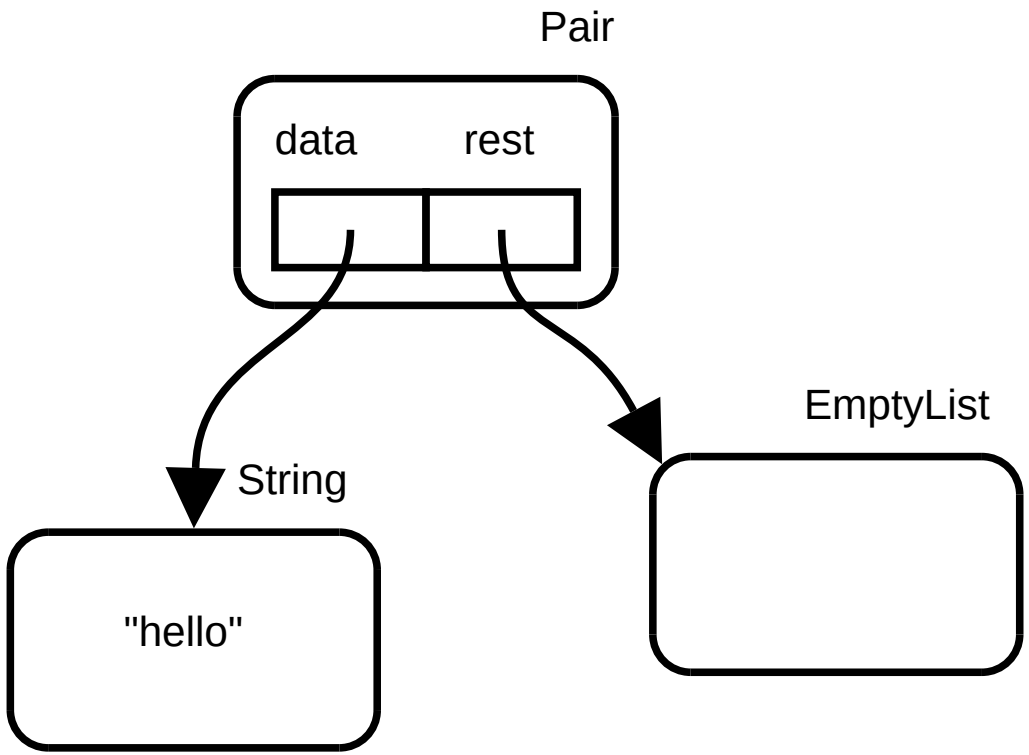
Recursive data-structures



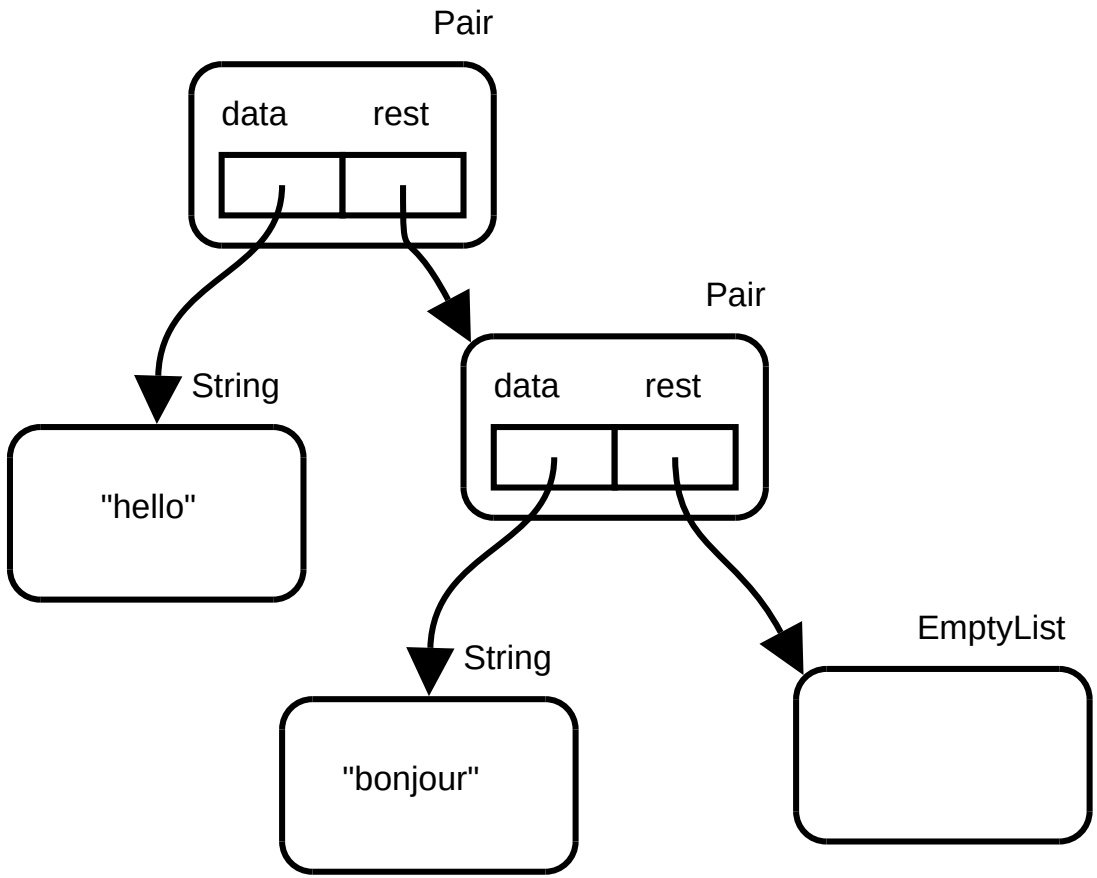
Recursive data-structures



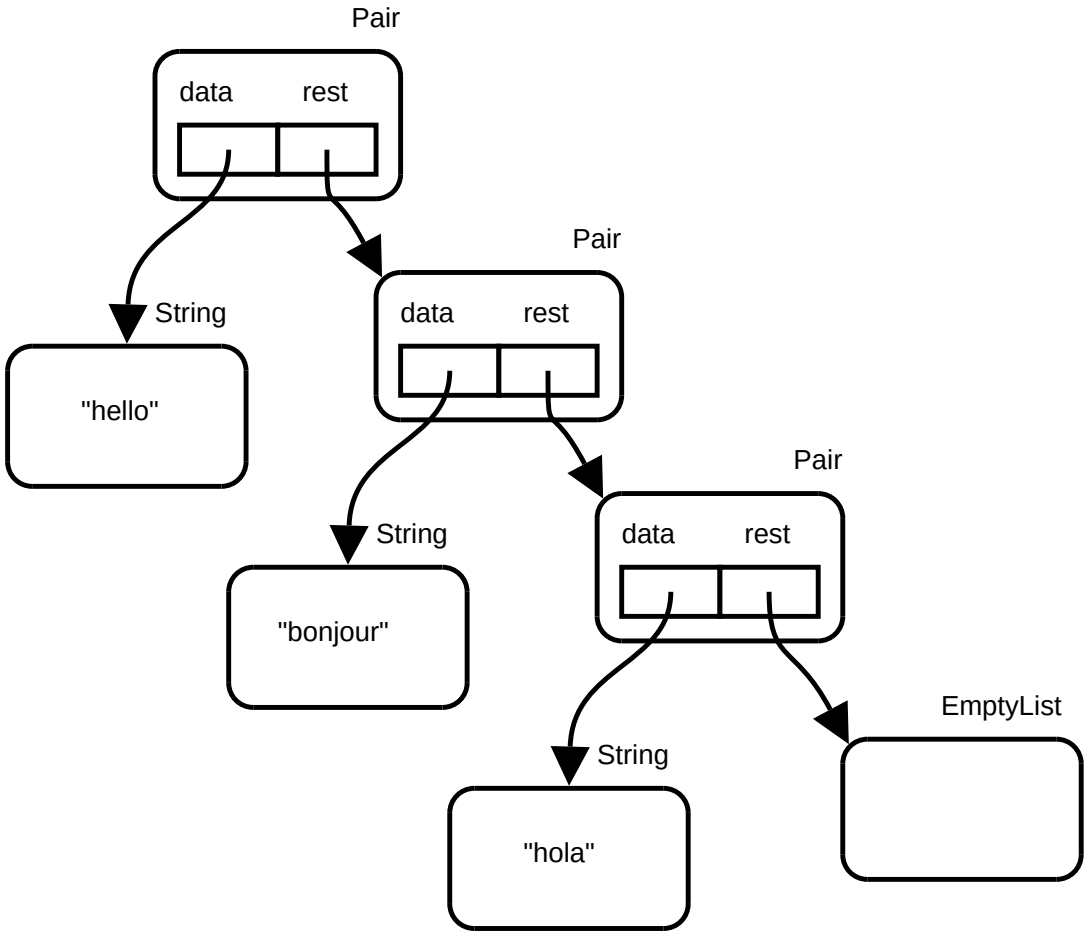
Recursive data-structures



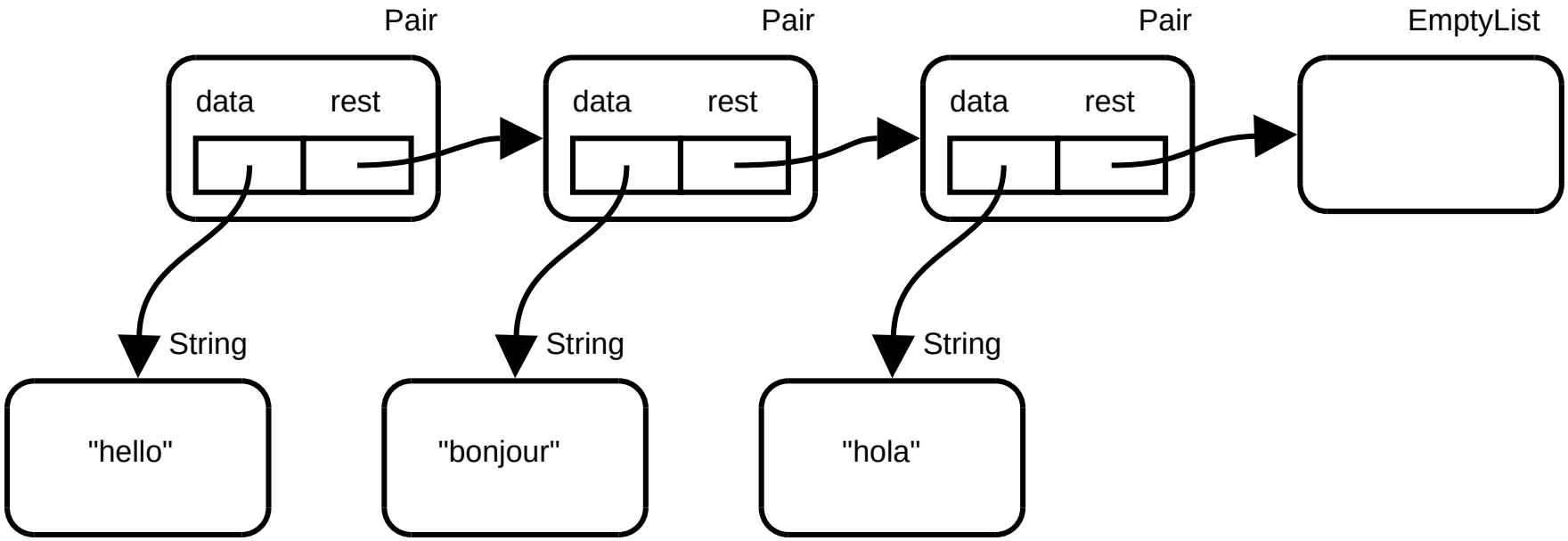
Recursive data-structures



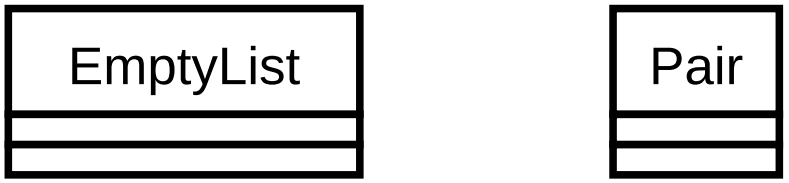
Recursive data-structures



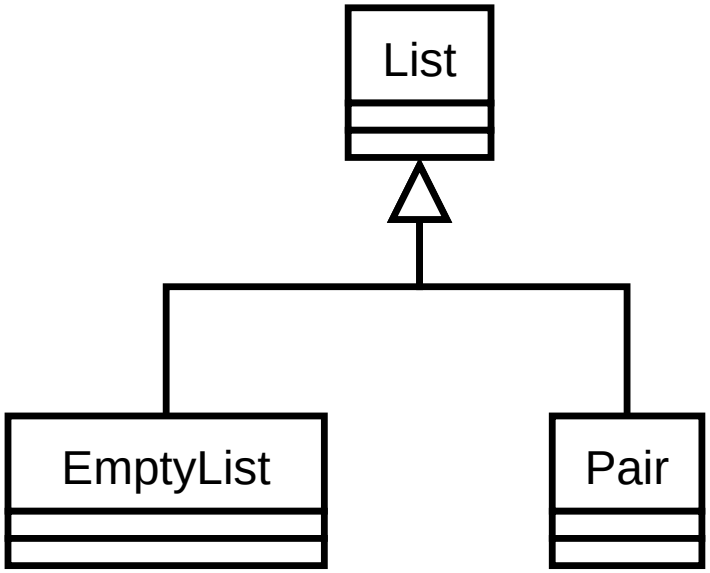
Recursive data-structures



Recursive data-structures

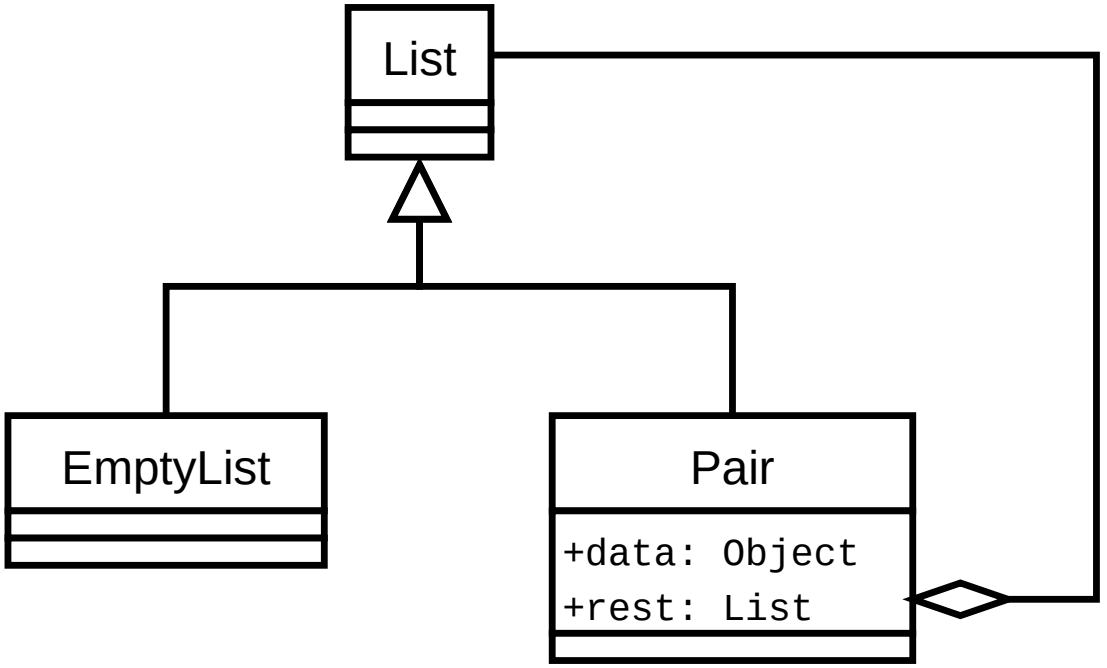


Recursive data-structures



- An empty list *is a* list
- A pair *is a* list

Recursive data-structures



- A pair *has a* data object
- A pair *has a* list (a reference to the rest or the list)

Recursive data-structures

```
class List
{
}
```

Recursive data-structures

```
class List
{
}

class EmptyList extends List
{
}

class Pair extends List
{
}
```

Recursive data-structures

```
class List
{
}

class EmptyList extends List
{
}

class Pair extends List
{
  Object data;
  List   rest;
}
```

Recursive data-structures

```
class Pair extends List
{
  Object data;
  List   rest;

  Pair(Object d, List l)
  {
    data = d;
    rest = l;
  }

  Object getData() { return data; }
  List getRest() { return rest; }
}
```

Recursive data-structures

```
public class ListTest
{
  public static void main(String[] args)
  {
    EmptyList nil = new EmptyList();
    List l1 = nil;
  }
}
```

Recursive data-structures

```
public class ListTest
{
  public static void main(String[] args)
  {
    List l1 = new EmptyList();
  }
}
```

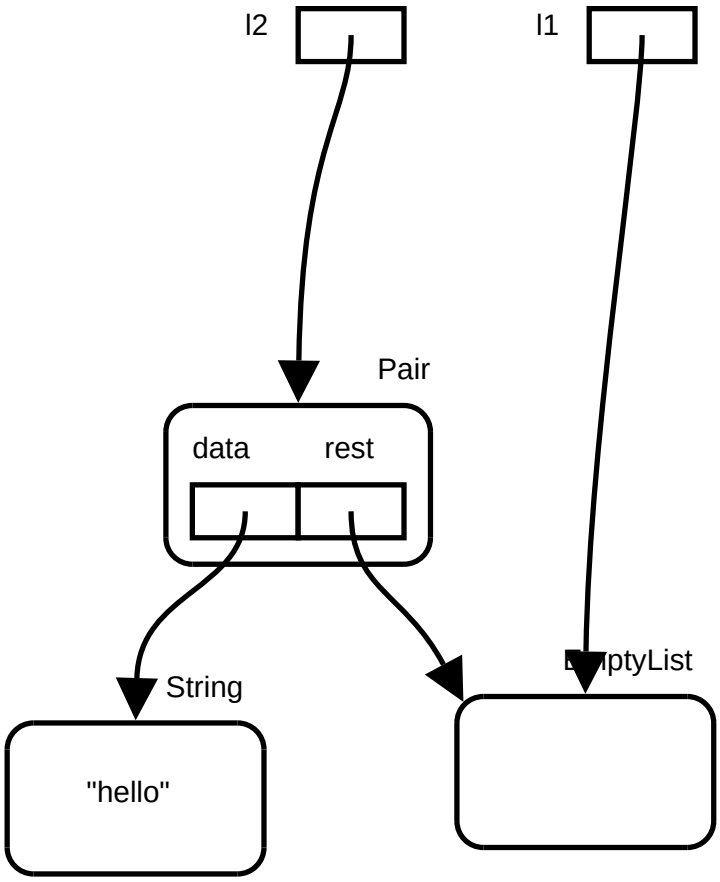
Recursive data-structures

```
public class ListTest
{
  public static void main(String[] args)
  {
    List l1 = new EmptyList();
    Pair p1 = new Pair("hello", l1);
  }
}
```

Recursive data-structures

```
public class ListTest
{
  public static void main(String[] args)
  {
    List l1 = new EmptyList();
    List l2 = new Pair(“hello”, l1);
  }
}
```

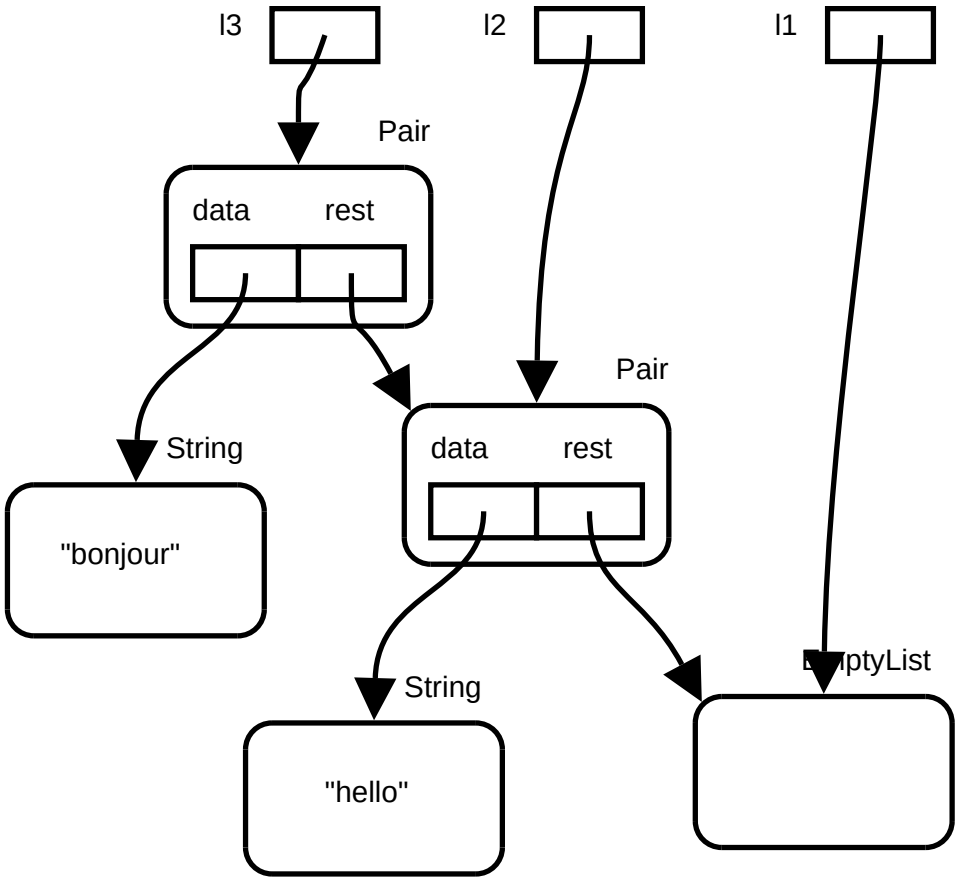
Recursive data-structures



Recursive data-structures

```
public class ListTest
{
  public static void main(String[] args)
  {
    List l1 = new EmptyList();
    List l2 = new Pair("hello", l1);
    List l3 = new Pair("bonjour", l2);
  }
}
```

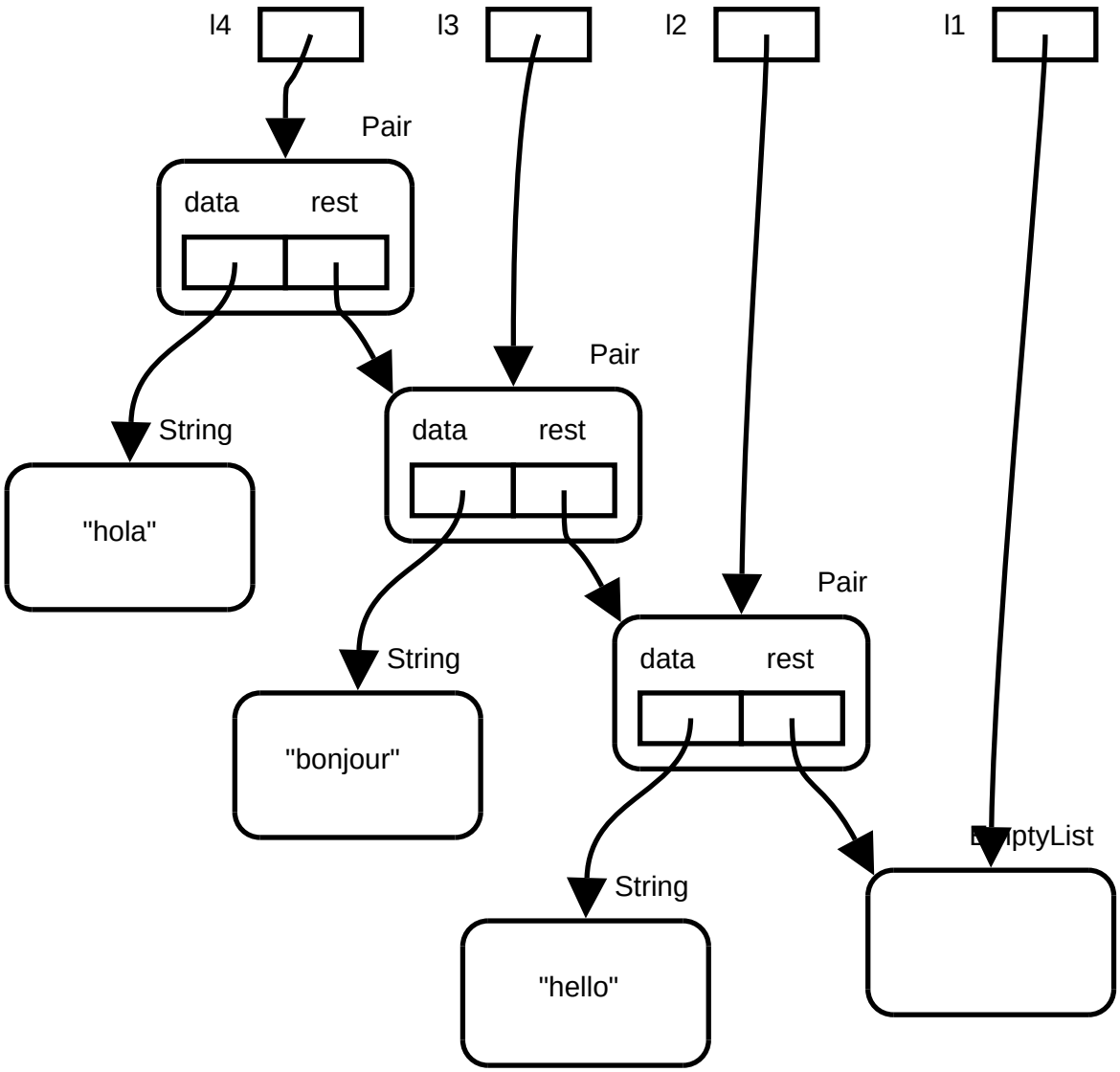
Recursive data-structures



Recursive data-structures

```
public class ListTest
{
  public static void main(String[] args)
  {
    List l1 = new EmptyList();
    List l2 = new Pair("hello", l1);
    List l3 = new Pair("bonjour", l2);
    List l4 = new Pair("hola", l3);
  }
}
```

Recursive data-structures



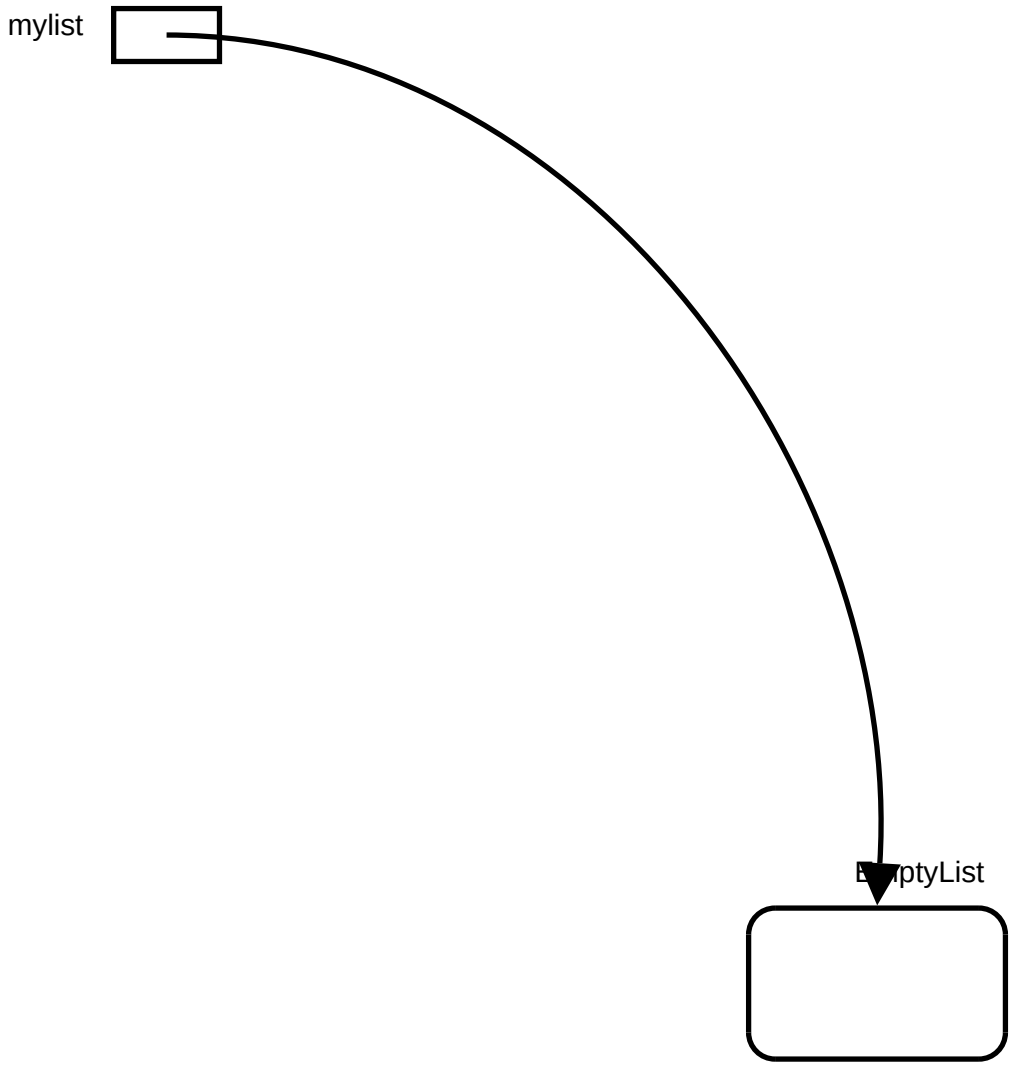
Recursive data-structures

```
public class ListTest
{
  public static void main(String[] args)
  {
    List l4 = new Pair(“hola”,
                       new Pair(“bonjour”,
                                new Pair(“hello”,
                                         new EmptyList())));
  }
}
```

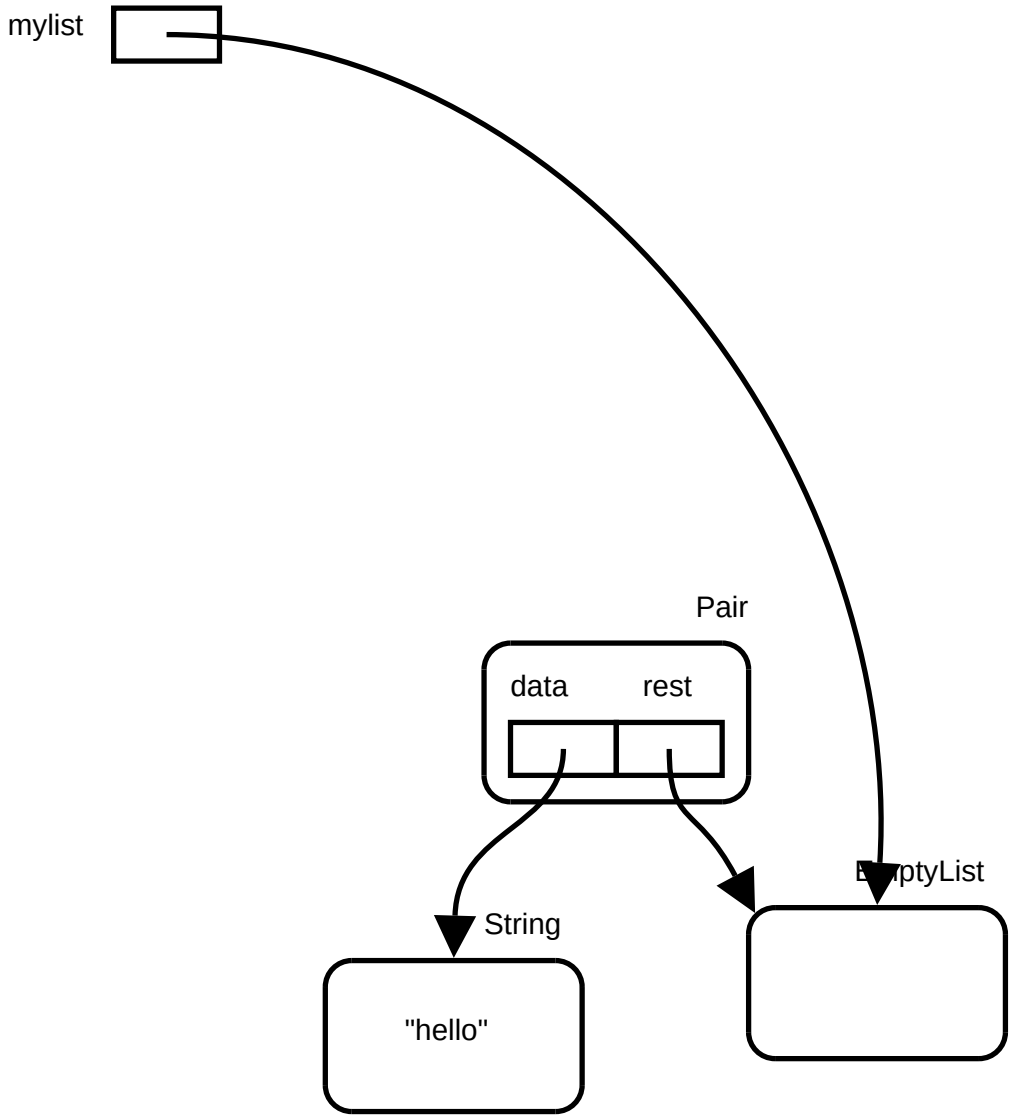
Recursive data-structures

```
public class ListTest
{
  public static void main(String[] args)
  {
    Scanner scanner = new Scanner(System.in);
    List mylist = new EmptyList();
    int i = 1;
    while (i <= 3)
    {
      System.out.print(“Enter a word: ”);
      String word = scanner.nextLine();
      mylist = new Pair(word, mylist);
      i++;
    }
  }
}
```

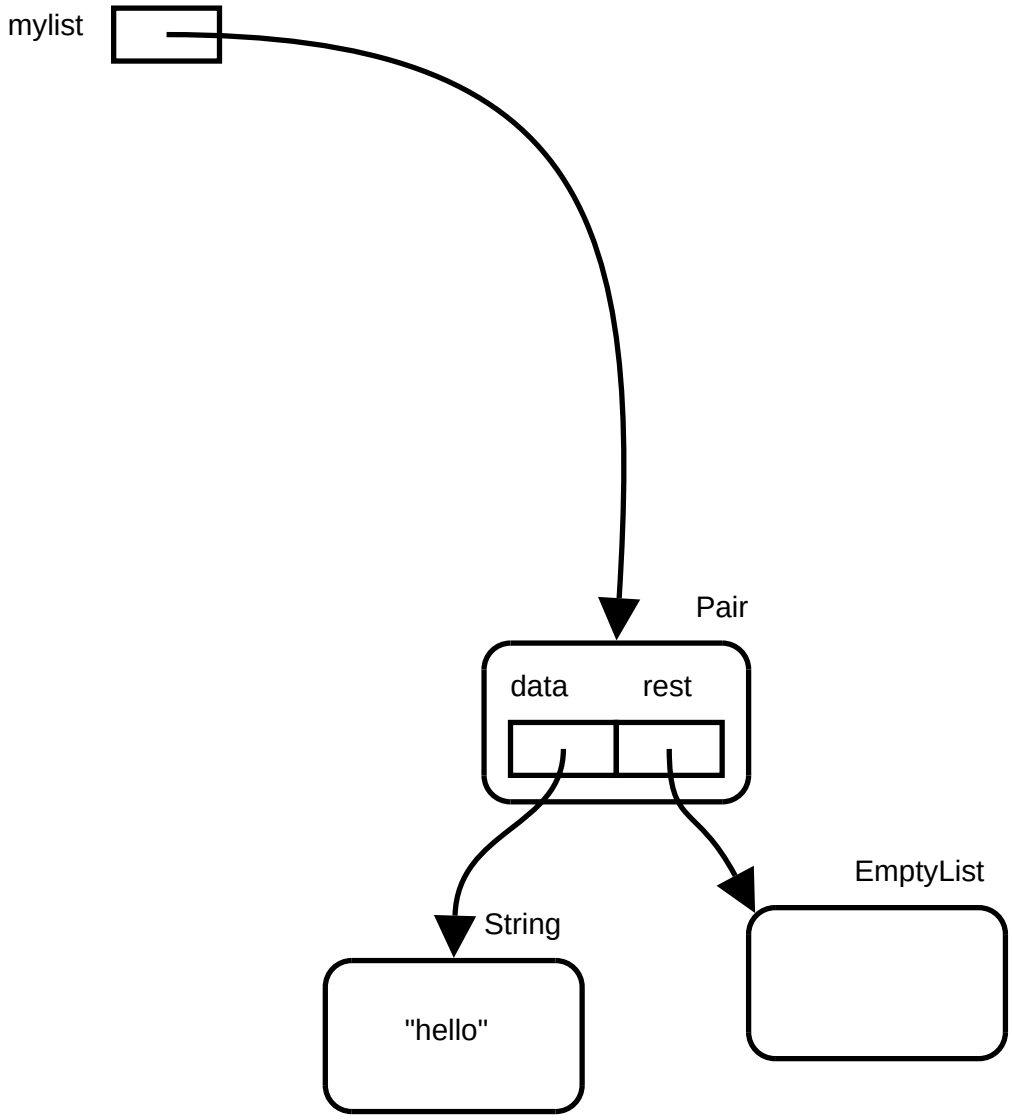
Recursive data-structures



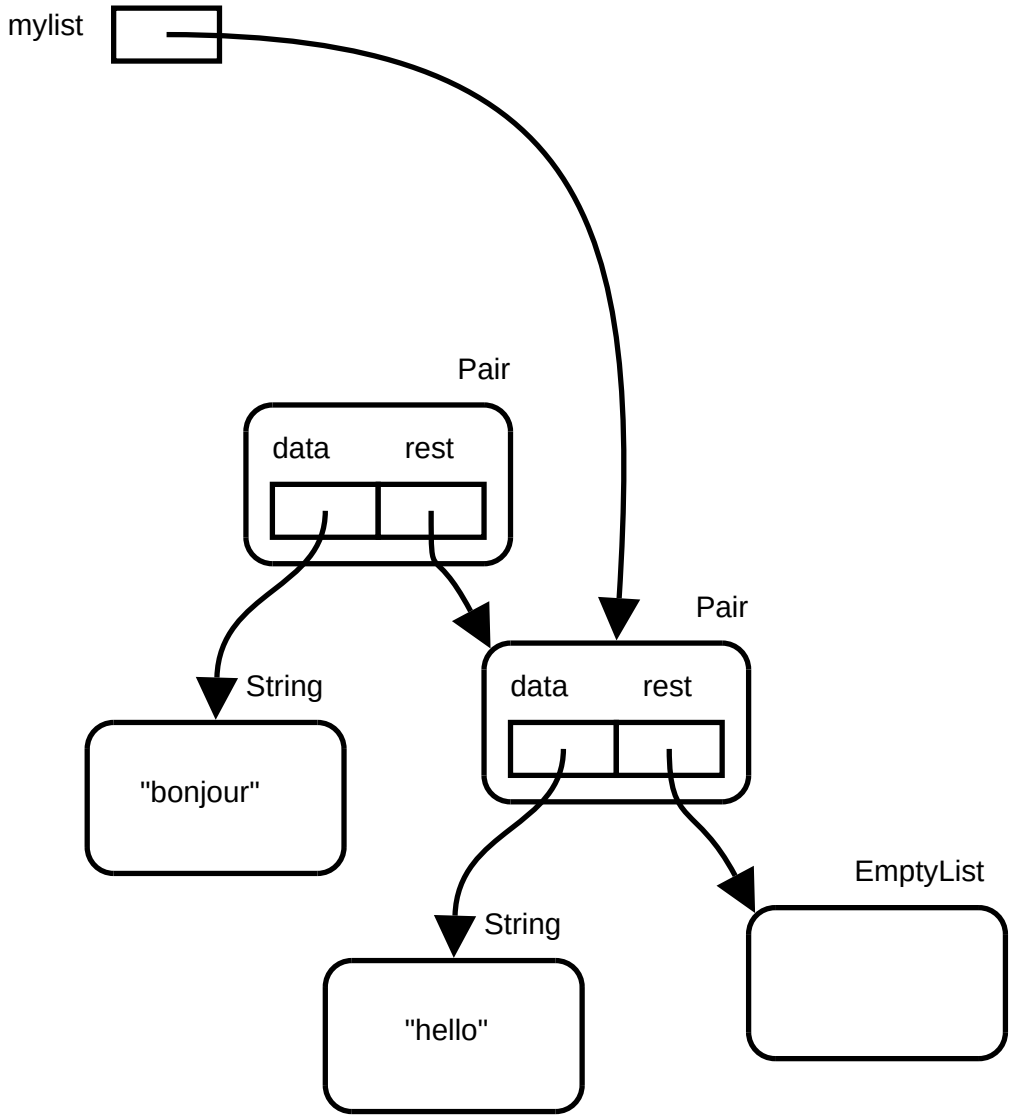
Recursive data-structures



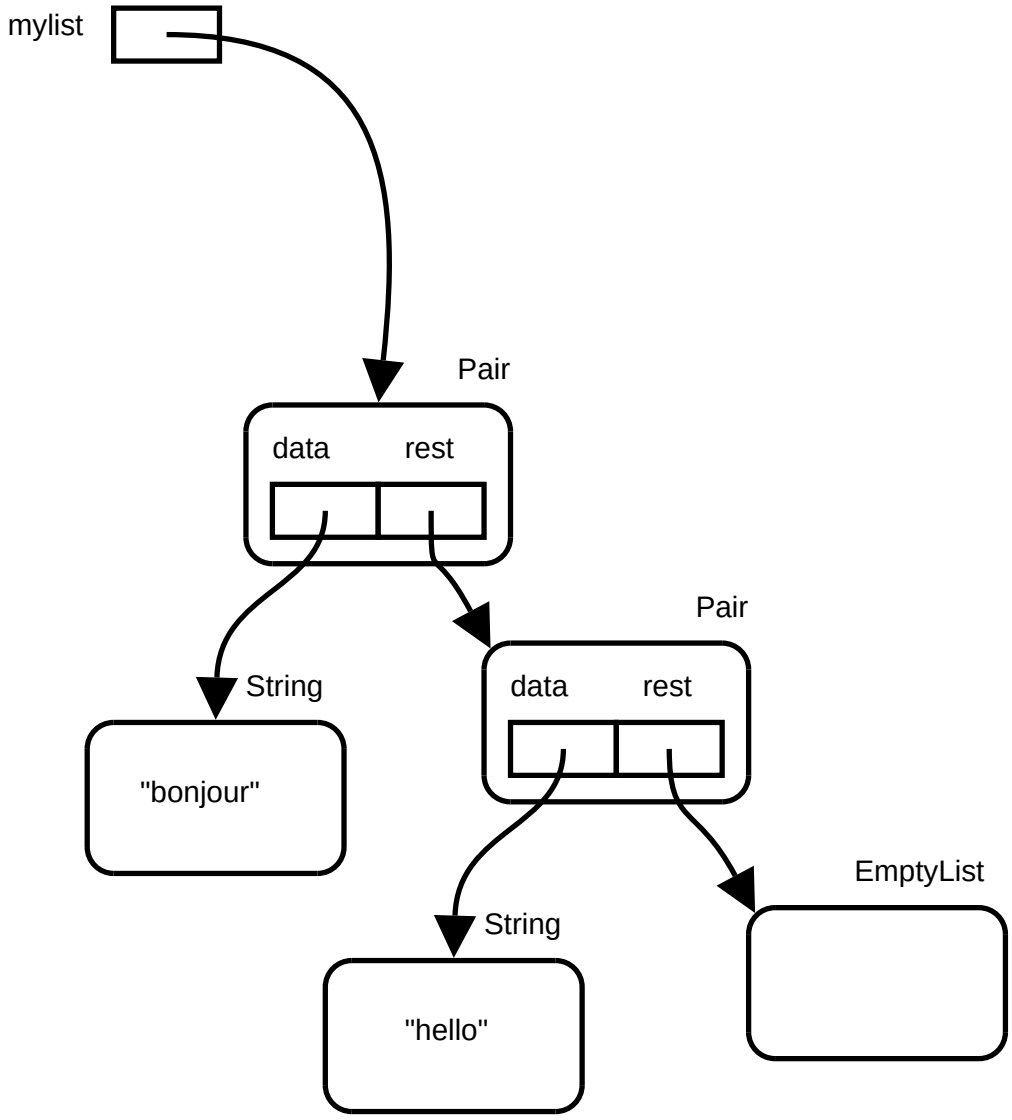
Recursive data-structures



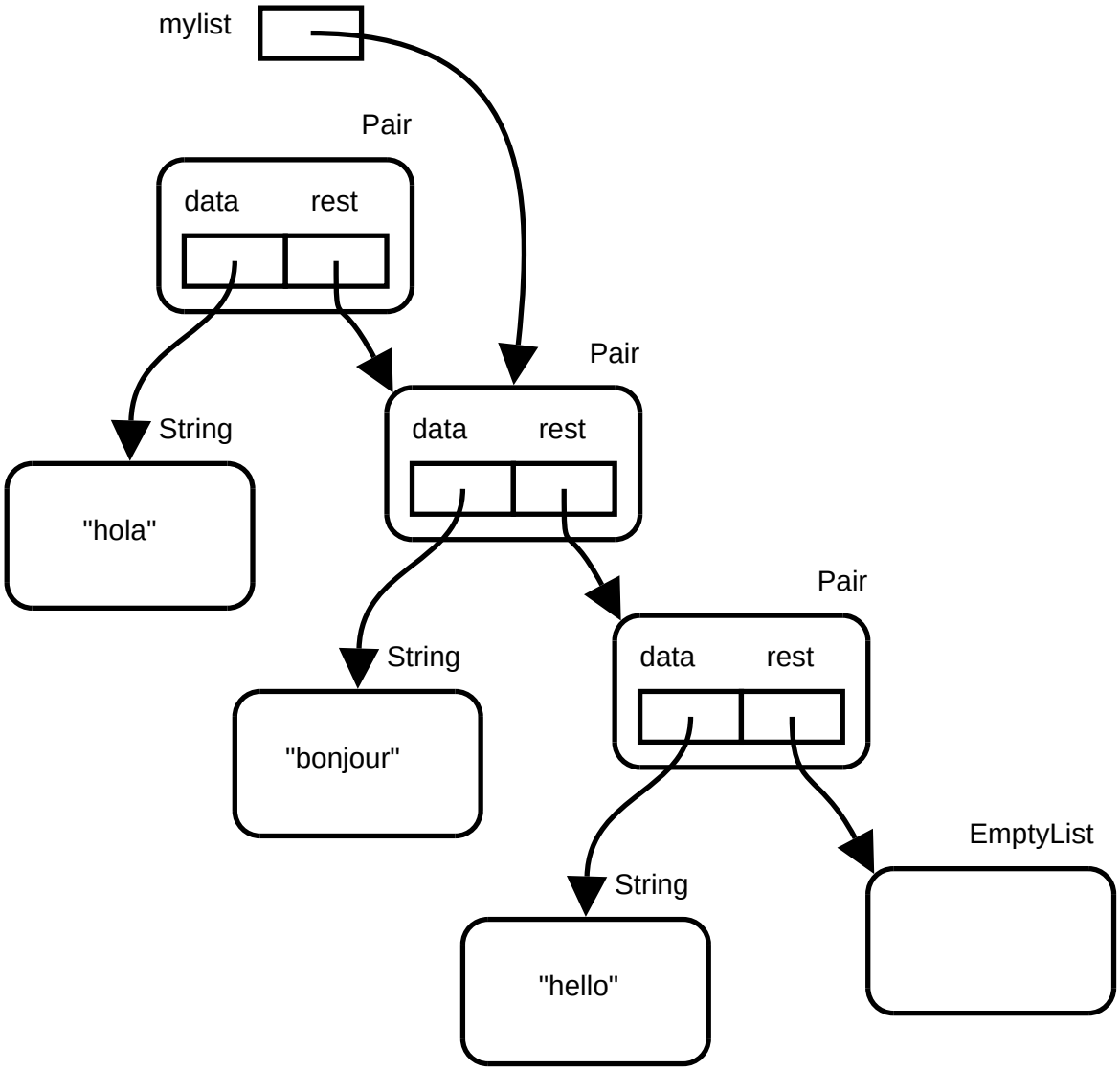
Recursive data-structures



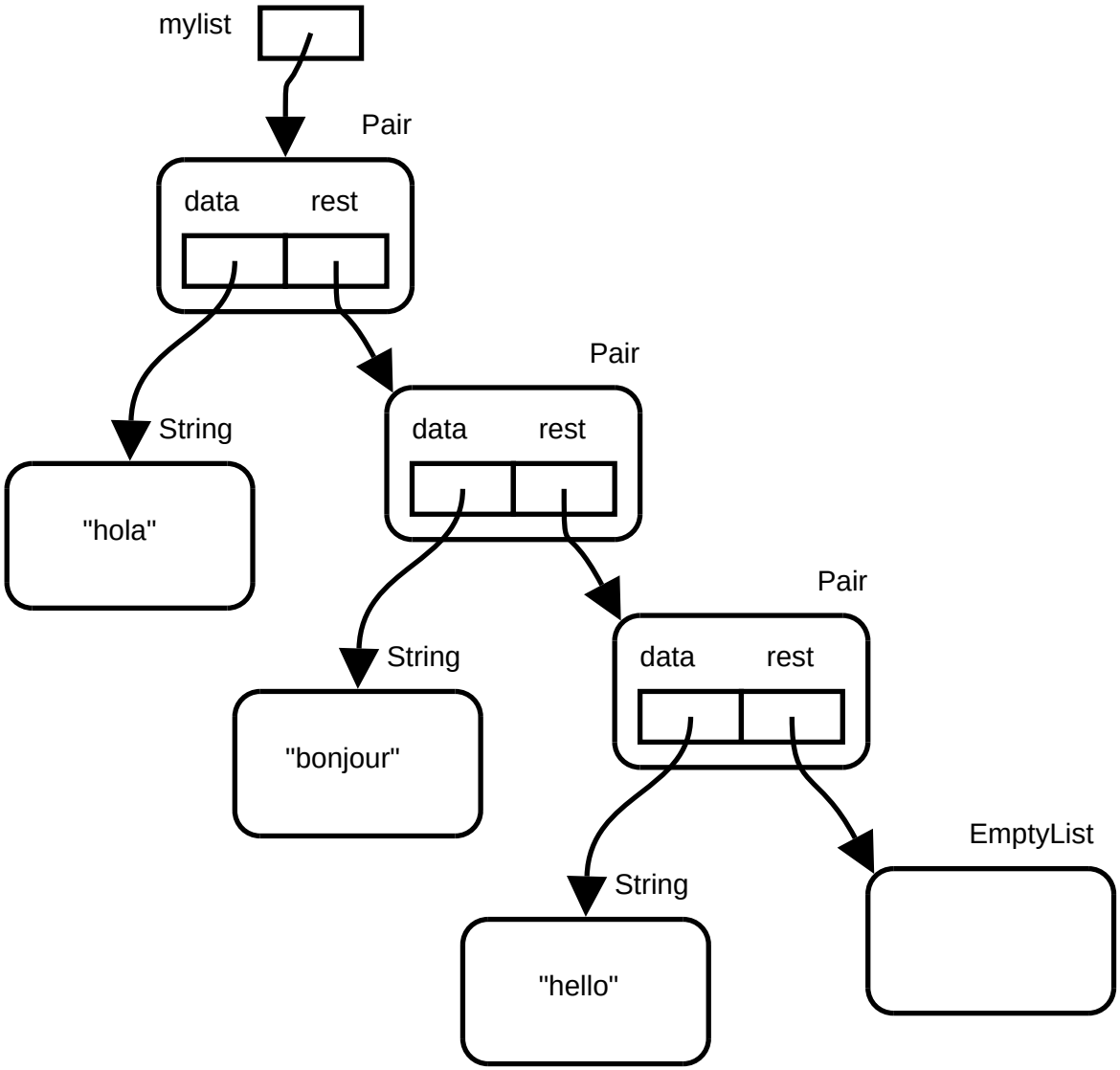
Recursive data-structures



Recursive data-structures



Recursive data-structures



Recursive data-structures

```
public class ListTest
{
  public static void main(String[] args)
  {
    List l = enter_list(4);
  }
  static List enter_list(int n)
  {
    Scanner scanner = new Scanner(System.in);
    List mylist = new EmptyList();
    int i = 1;
    while (i <= n)
    {
      System.out.print("Enter a word: ");
      String word = scanner.nextLine();
      mylist = new Pair(word, mylist);
      i++;
    }
    return mylist;
  }
}
```

Recursive data-structures

- Linked-lists are dynamically allocated data-structures:
  - Data-structures: data organized in a particular pattern
  - Dynamically allocated: elements are added at runtime (no predetermined fixed size, like arrays)

Recursive data-structures

```
public class ListTest
{
  public static void main(String[] args)
  {
    List l = enter_list(4);
    print_list(l);
  }
  static List enter_list()
  { ... }
  static void print_list(List l)
  { ... }
}
```

Recursive data-structures

To print a list *l*:

1. If *l* is an empty list:
   (a) print ""
2. Otherwise (it is a pair)
   (a) print the *data* of the pair, and
   (b) print the *rest* of the list

Recursive data-structures

```
class Pair extends List
{
  Object data;
  List   rest;

  Pair(Object d, List l)
  {
    data = d;
    rest = l;
  }

  Object getData() { return data; }
  List getRest() { return rest; }
}
```

Recursive data-structures

```
public class ListTest
{
  ...
  static void print_list(List l)
  {
    if (l instanceof EmptyList)
    {
      System.out.print("");
    }
    else
    {
      Pair p = (Pair)l;
      Object data = p.getData();
      List rest = p.getRest();
      System.out.print(data + ", ");
      print_list(rest);
    }
  }
}
```

Recursive data-structures

```
public class ListTest
{
  public static void main(String[] args)
  {
    List l = enter_list(4);
    print_list(l);
  }
  static List enter_list()
  { ... }
  static void print_list(List l)
  { ... }
  static boolean member(Object item, List l)
  { ... }
}
```

Recursive data-structures

```
public class ListTest
{
  public static void main(String[] args)
  {
    List l = enter_list(4);
    print_list(l);
    if (member(“beer”, l))
    {
      System.out.print("It’s there");
    }
  }
  static List enter_list()
  { ... }
  static void print_list(List l)
  { ... }
  static boolean member(Object item, List l)
  { ... }
}
```

Recursive data-structures

To determine whether an item x is in a list l, do:

1. If l is an empty list:
   (a) return false
2. Otherwise (it is a pair)
   (a) If the *data* of the pair is equal to x:
      i. return true
   (b) Otherwise:
      i. determine whether x is in the *rest* of the list, and return the result of that

Recursive data-structures

```
static boolean member(Object item, List l)
{
  if (l instanceof EmptyList)
  {
    return false;
  }
  else
  {
    Pair p = (Pair)l;
    Object data = p.getData();
    if (data.equals(item))
    {
      return true;
    }
    else
    {
      List rest = p.getRest();
      return member(item, rest);
    }
  }
}
```

Recursive data-structures

```
class List
{
  static void print_list(List l)
  { ... }
  static boolean member(Object item, List l)
  { ... }
}
```

Recursive data-structures

```
class List
{
  void print_list()
  { ... }
  boolean member(Object item)
  { ... }
}
```

Recursive data-structures

```
class ListTest
{
  static void print_list(List l)
  {
    if (l instanceof EmptyList)
    {
      System.out.print("");
    }
    else
    {
      Pair p = (Pair)l;
      Object data = p.getData();
      List rest = p.getRest();
      System.out.print(data + ", ");
      print_list(rest);
    }
  }
}
```

Recursive data-structures

```
class List
{
  void print_list()
  {
    if (this instanceof EmptyList)
    {
      System.out.print("");
    }
    else
    {
      Pair p = (Pair)this;
      Object data = p.getData();
      List rest = p.getRest();
      System.out.print(data + ", ");
      rest.print_list();
    }
  }
}
```

Recursive data-structures

```
static boolean member(Object item, List l)
{
  if (l instanceof EmptyList)
  {
    return false;
  }
  else
  {
    Pair p = (Pair)l;
    Object data = p.getData();
    if (data.equals(item))
    {
      return true;
    }
    else
    {
      List rest = p.getRest();
      return member(item, rest);
    }
  }
}
```

Recursive data-structures

```
class List {
  boolean member(Object item)
  {
    if (this instanceof EmptyList)
    {
      return false;
    }
    else
    {
      Pair p = (Pair)this;
      Object data = p.getData();
      if (data.equals(item))
      {
        return true;
      }
      else
      {
        List rest = p.getRest();
        return rest.member(item);
      }
    }
  }
}
```

Recursive data-structures

```
public class ListTest
{
  public static void main(String[] args)
  {
    List l = enter_list(4);
    print_list(l);
    if (member(“beer”, l))
    {
      System.out.print("It’s there");
    }
  }
  static List enter_list()
  { ... }
  static void print_list(List l)
  { ... }
  static boolean member(Object item, List l)
  { ... }
}
```

Recursive data-structures

```
public class ListTest
{
  public static void main(String[] args)
  {
    List l = enter_list(4);
    l.print_list();
    if (l.member(“beer”))
    {
      System.out.print("It’s there");
    }
  }
  static List enter_list()
  { ... }
}
```

Recursive data-structures

```
class List
{
  void print_list()
  { ... }
  boolean member(Object item)
  { ... }
}
```

Recursive data-structures

```
abstract class List
{
  abstract void print_list();

  abstract boolean member(Object item);
}
```

Recursive data-structures

```
class EmptyList extends List
{
}
```

Recursive data-structures

```
class EmptyList extends List
{
  void print_list()
  { ... }
  boolean member(Object item)
  { ... }
}
```

Recursive data-structures

```
class EmptyList extends List
{
  void print_list()
  {
    System.out.print("");
  }
  boolean member(Object item)
  {
    return false;
  }
}
```

Recursive data-structures

```
class Pair extends List
{
  Object data;
  List   rest;

  Pair(Object d, List l) { ... }

  Object getData() { return data; }
  List getRest() { return rest; }

  void print_list()
  { ... }
  boolean member(Object item)
  { ... }
}
```

Recursive data-structures

```
class Pair extends List
{
  Object data;
  List   rest;

  ...

  void print_list()
  {
    Pair p = (Pair)this;
    Object data = p.getData();
    List rest = p.getRest();
    System.out.print(data + ", ");
    rest.print_list();
  }
}
```

Recursive data-structures

```
class Pair extends List
{
  Object data;
  List   rest;

  ...

  void print_list()
  {
    System.out.print(data + ", ");
    rest.print_list();
  }
}
```

Recursive data-structures

```
class Pair extends List
{
  Object data;
  List   rest;

  ...

  void print_list()
  {
    System.out.print(data + ", ");
    rest.print_list();  // Dynamic-dispatch
  }
}
```

Recursive data-structures

```
class Pair extends List
{
  Object data;
  List   rest;

  ...

  boolean member(Object item)
  {
    Pair p = (Pair)this;
    Object data = p.getData();
    if (data.equals(item))
    {
      return true;
    }
    else
    {
      List rest = p.getRest();
      return rest.member(item);
    }
  }
}
```

Recursive data-structures

```
class Pair extends List
{
  Object data;
  List   rest;

  ...

  boolean member(Object item)
  {
    if (data.equals(item))
    {
      return true;
    }
    else
    {
      return rest.member(item);
    }
  }
}
```

Recursive data-structures

```
class Pair extends List
{
  Object data;
  List   rest;

  ...

  boolean member(Object item)
  {
    if (data.equals(item))
    {
      return true;
    }
    else
    {
      return rest.member(item); // Dynamic-dispatc
    }
  }
}
```

The end