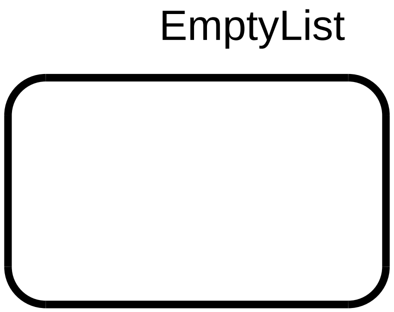
Announcements

- 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.)

Recursive data-structures

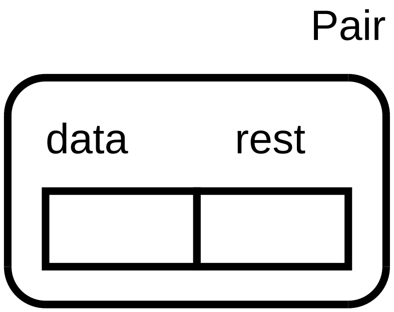
- For example:
  - A *list of data* is either:
    - An *empty list* [], or
    - A *pair* consisting of:
      - Some data, and
      - A list of data.
  - For example:
    - [] is a list
    - [5, []] is a list
    - [7, [5, []]] is a list
    - [6, [7, [5, []]]] is a list
    - [8, [6, [7, [5, []]]]]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

Recursive data-structures

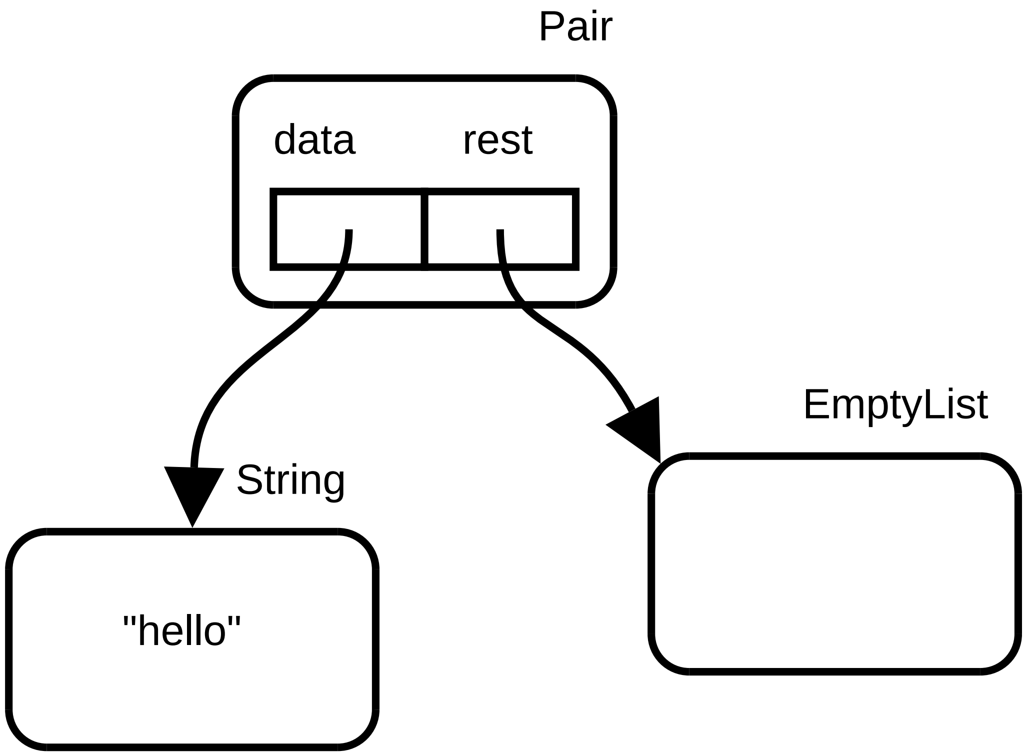


EmptyList

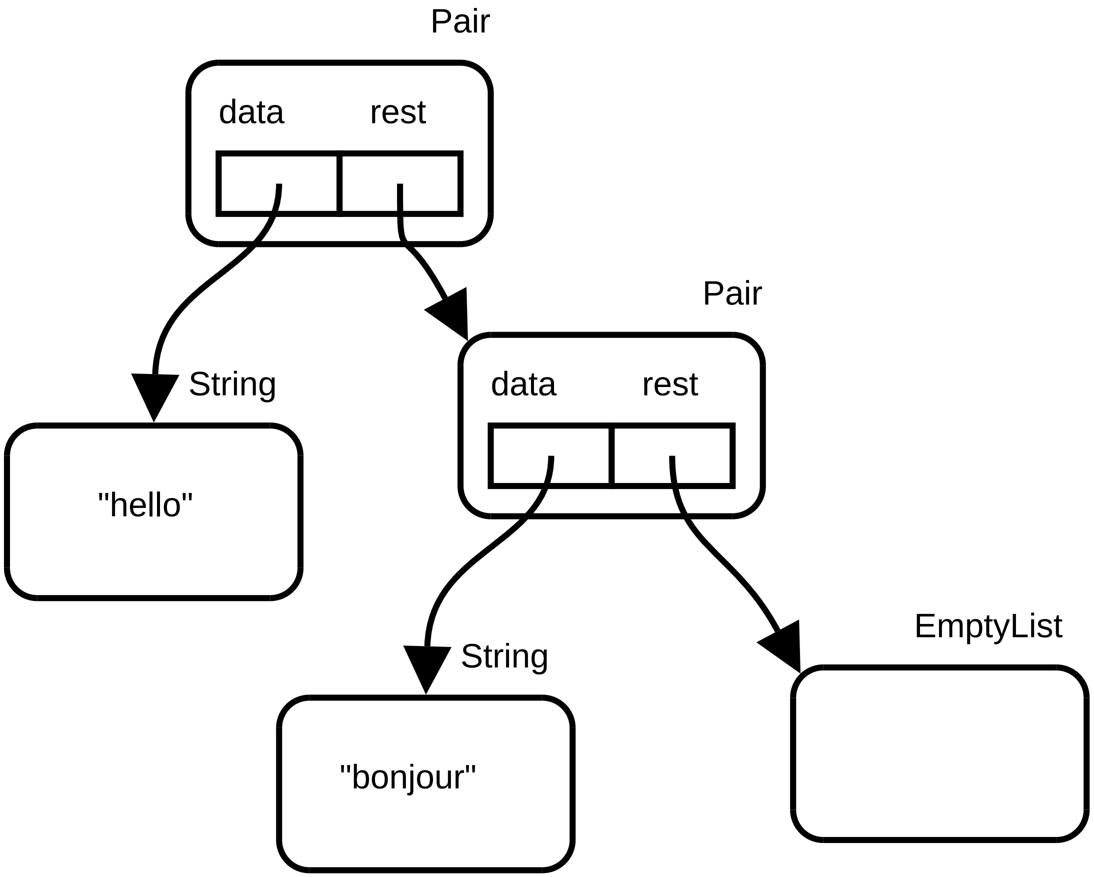
Recursive data-structures



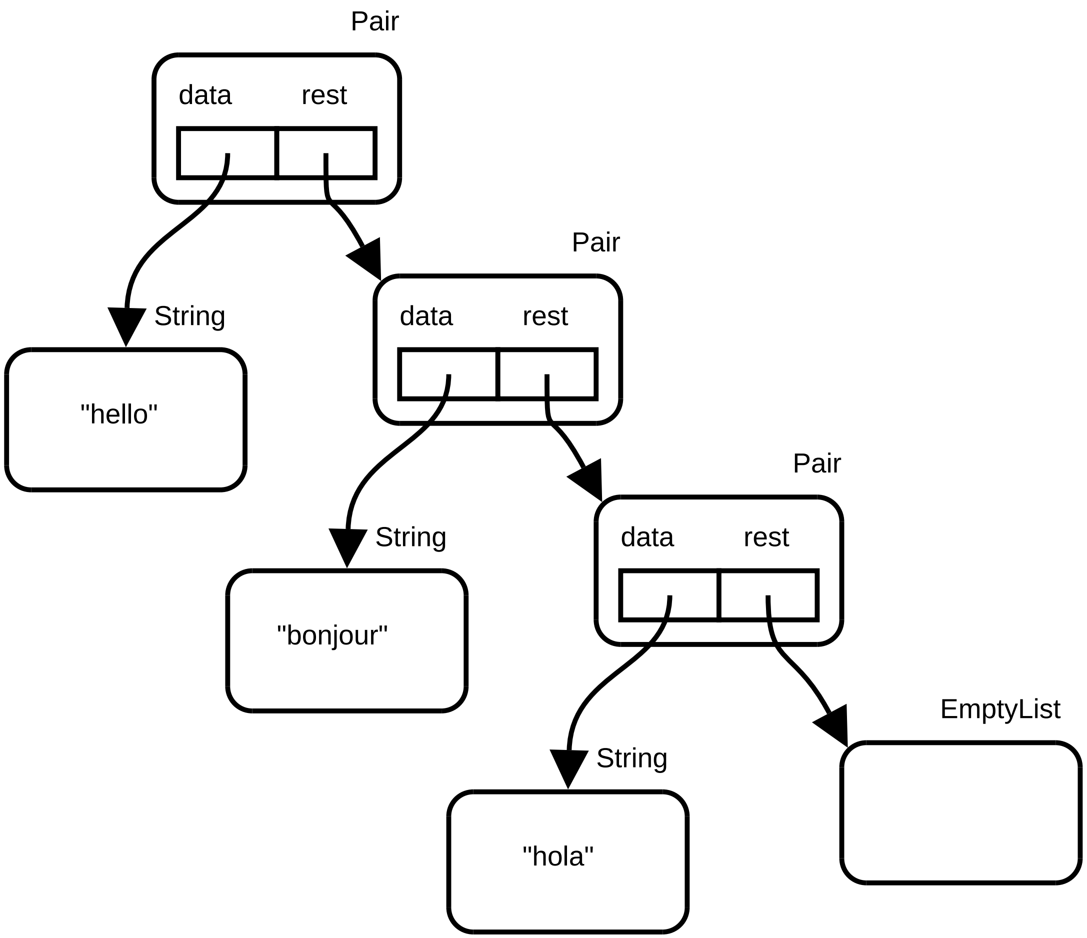
Recursive data-structures



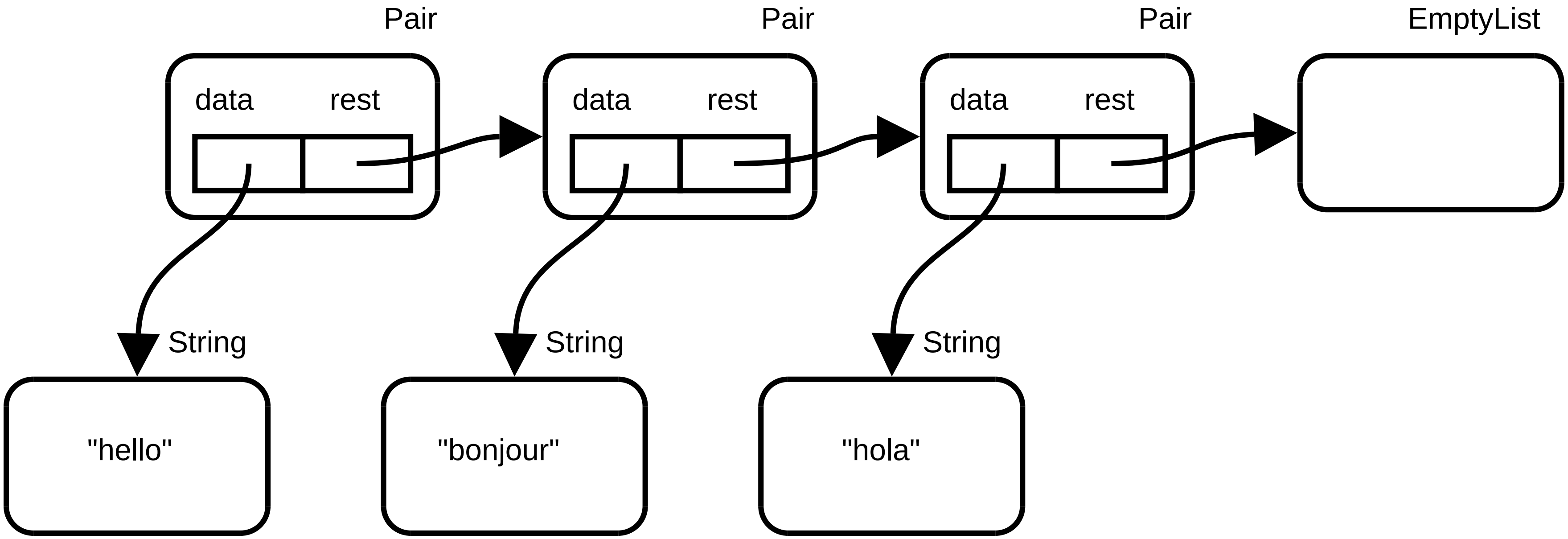
Recursive data-structures



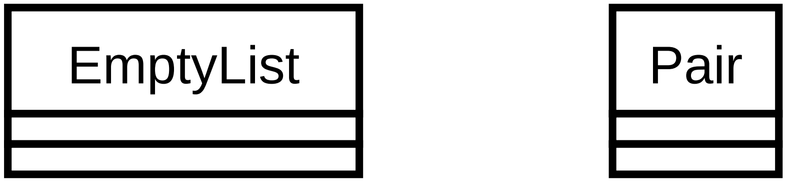
Recursive data-structures



Recursive data-structures



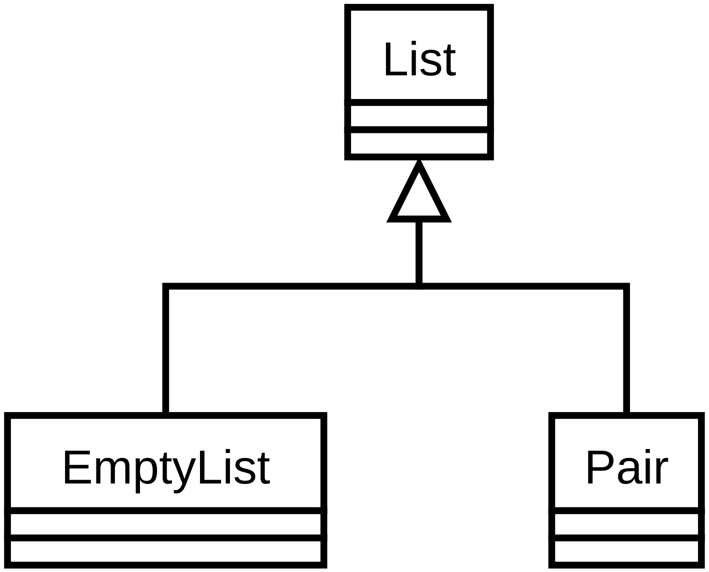
Recursive data-structures



EmptyList

Pair

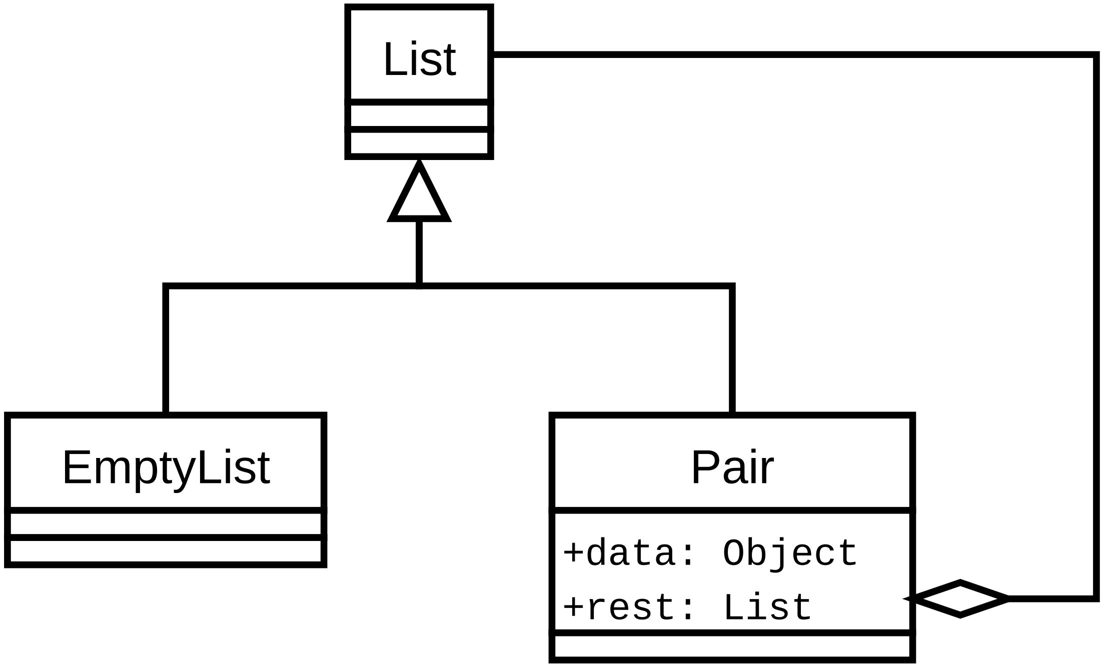
Recursive data-structures



- 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
{
}

class EmptyList extends List
{
}

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

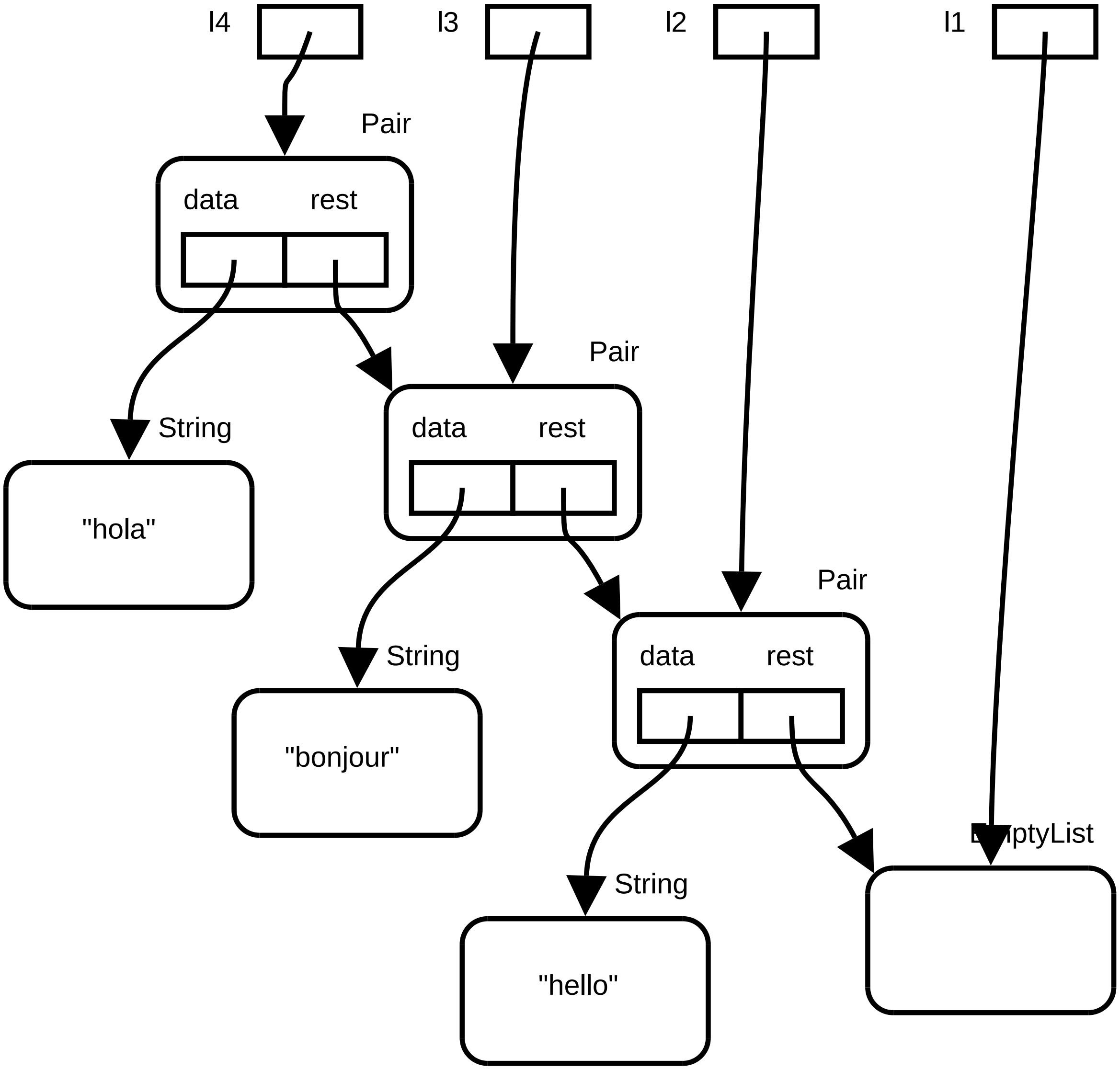
Recursive data-structures

```
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)
  {
    List l1 = new EmptyList();
    List l2 = new Pair(“hello”, l1);
    List l3 = new Pair(“bonjour”, l2);
    List l4 = new Pair(“hola”, l3);
  }
}
```

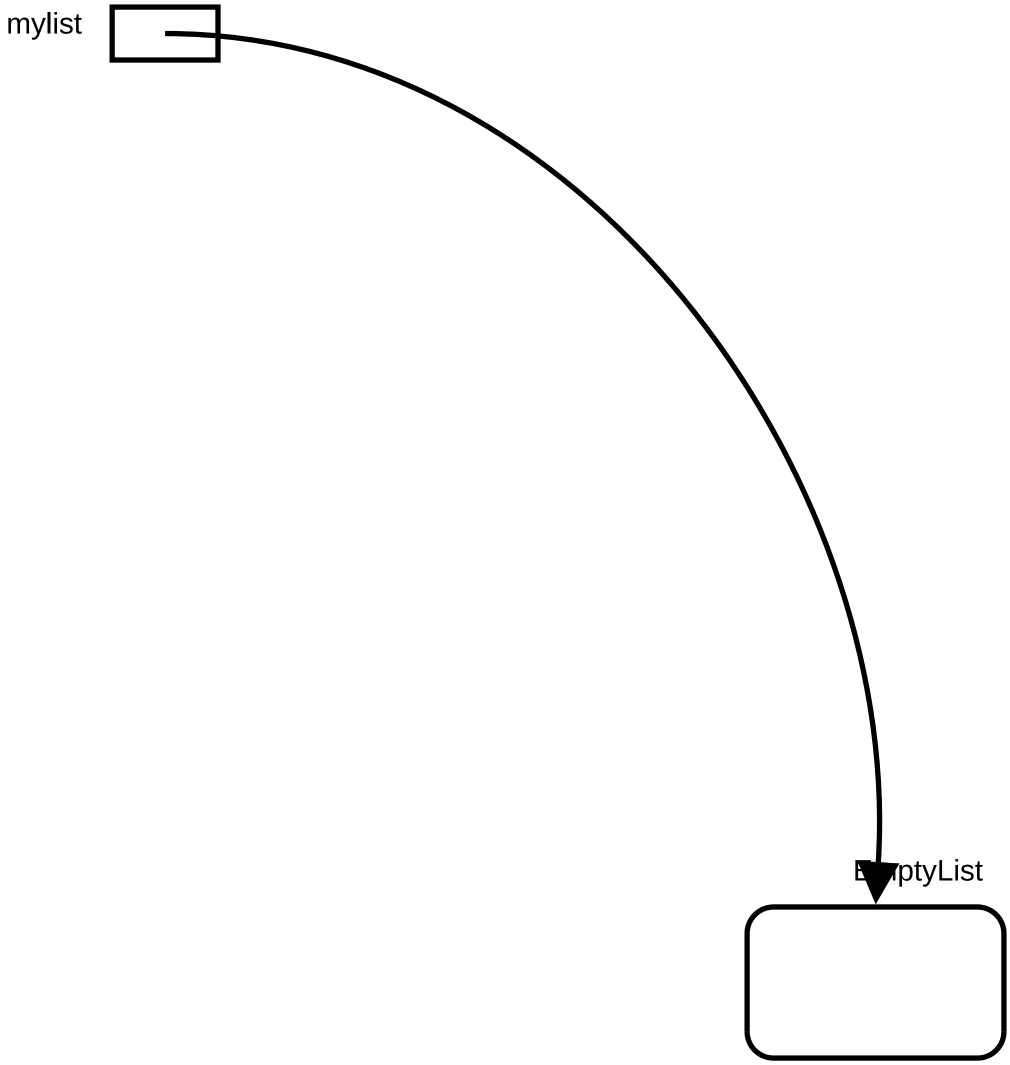
Recursive data-structures



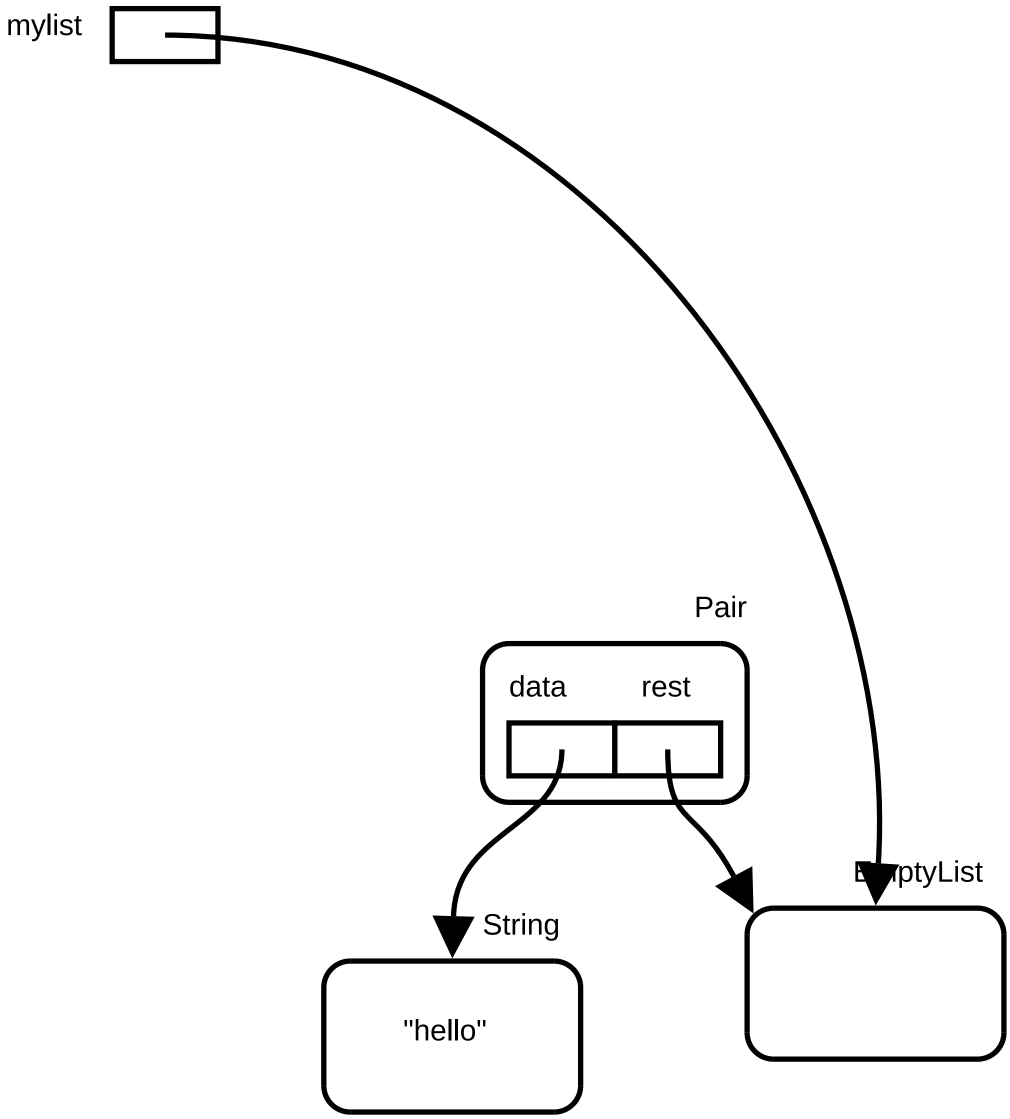
Recursive data-structures

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

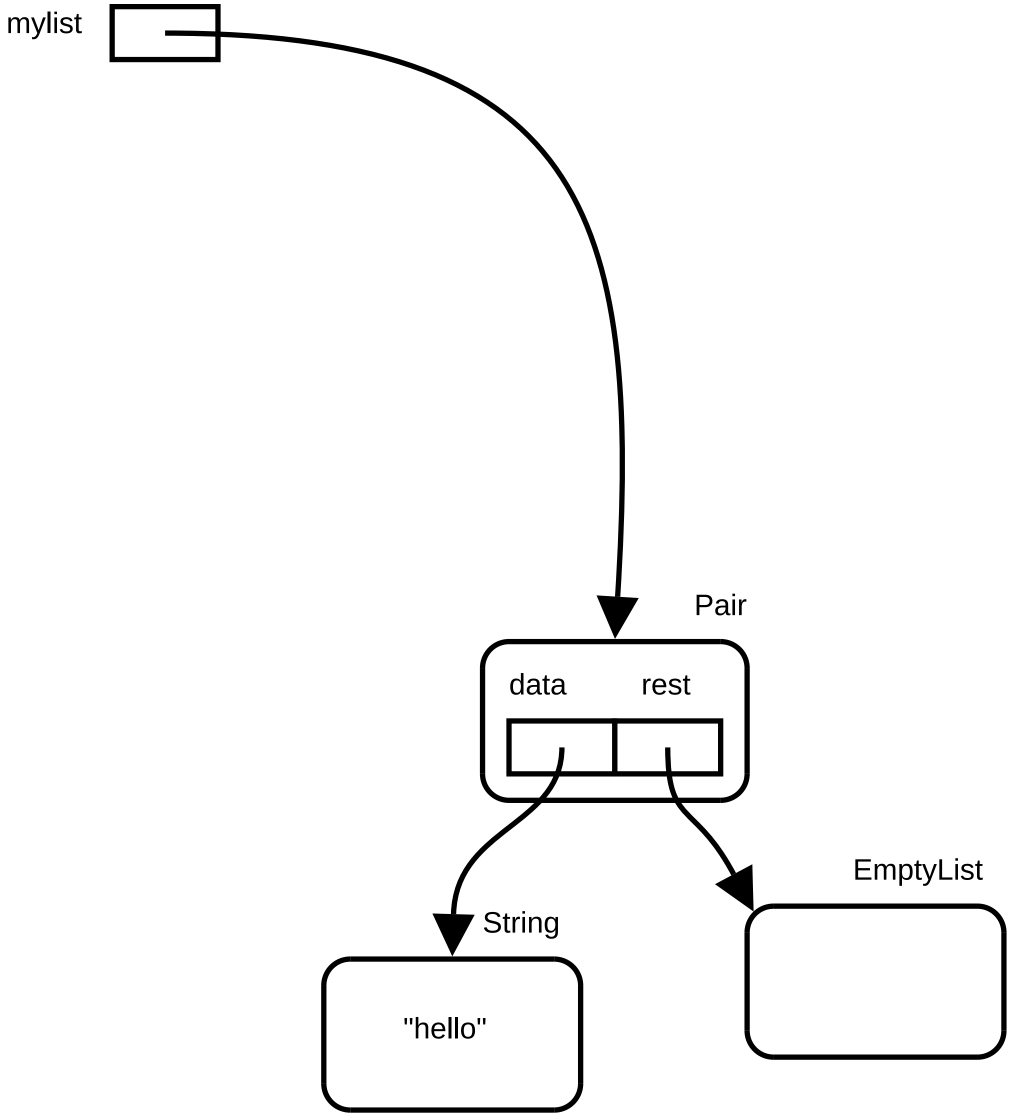
Recursive data-structures



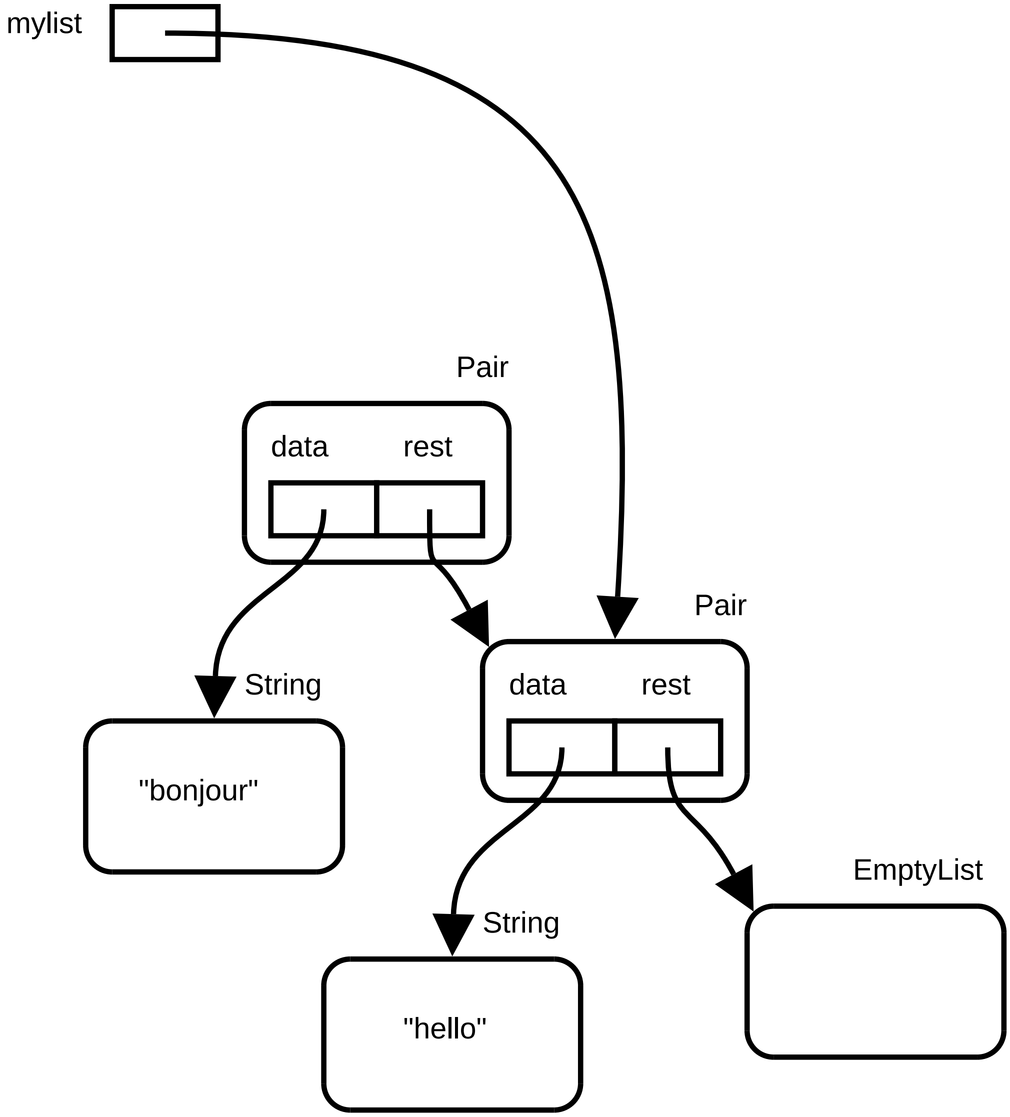
Recursive data-structures



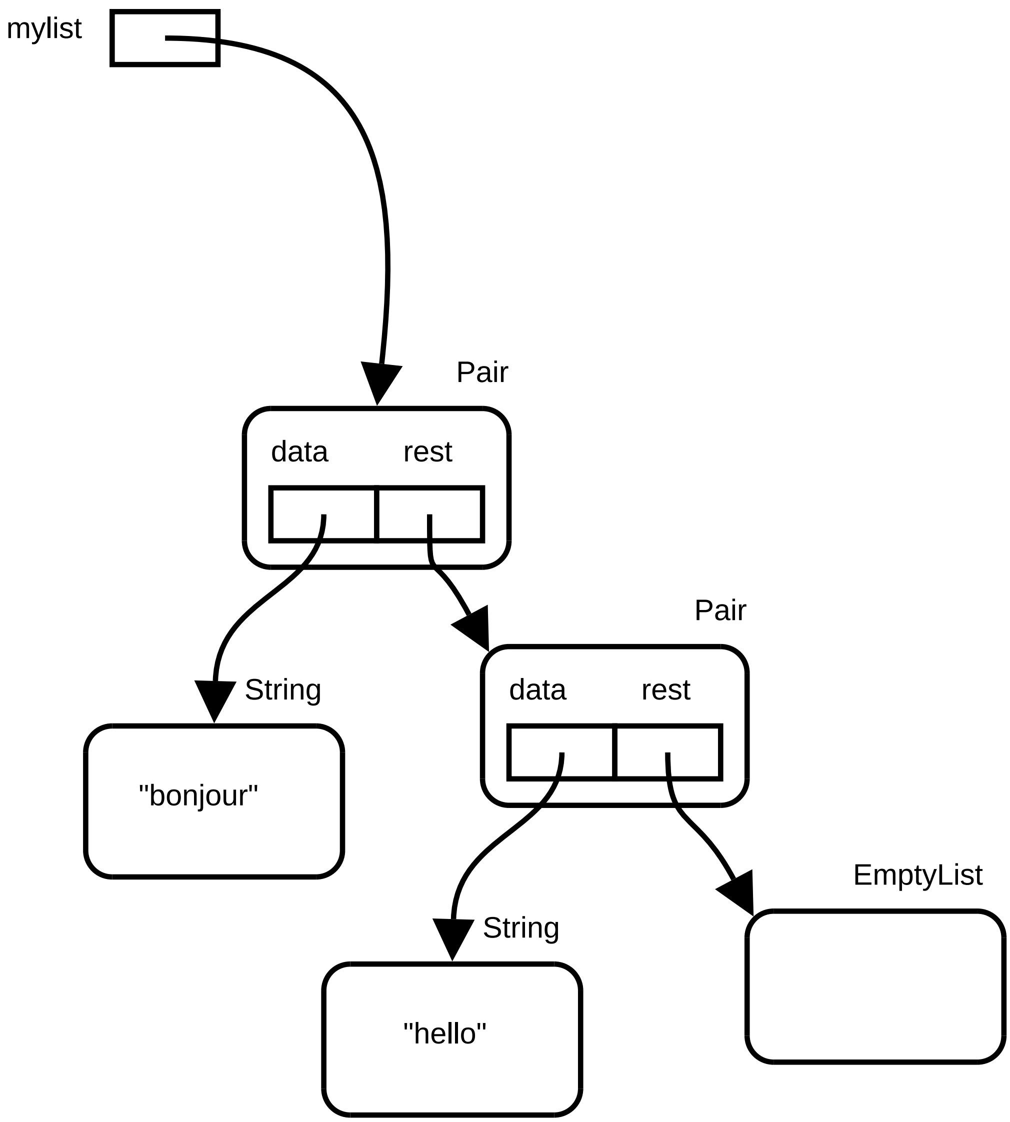
Recursive data-structures



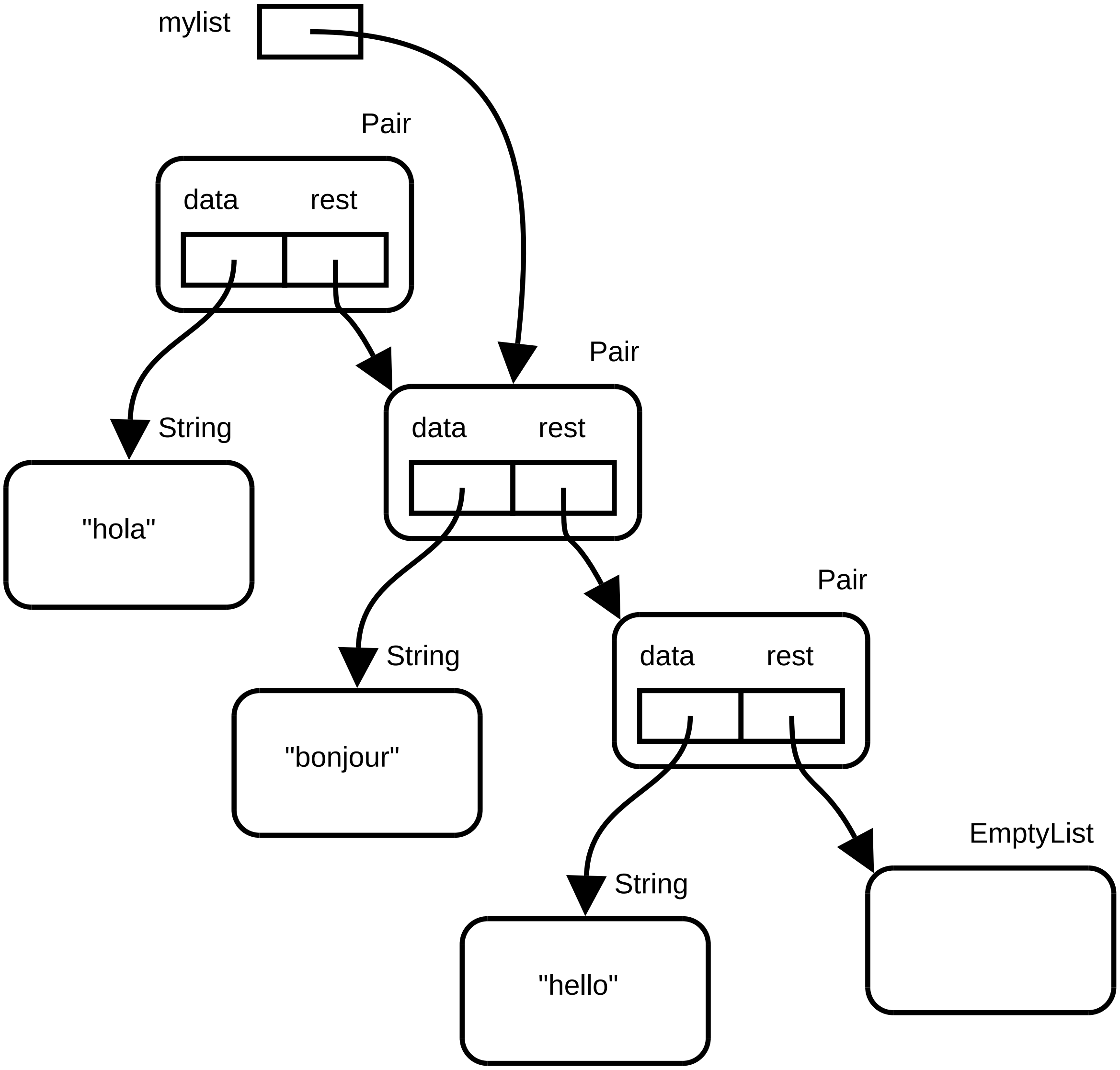
Recursive data-structures



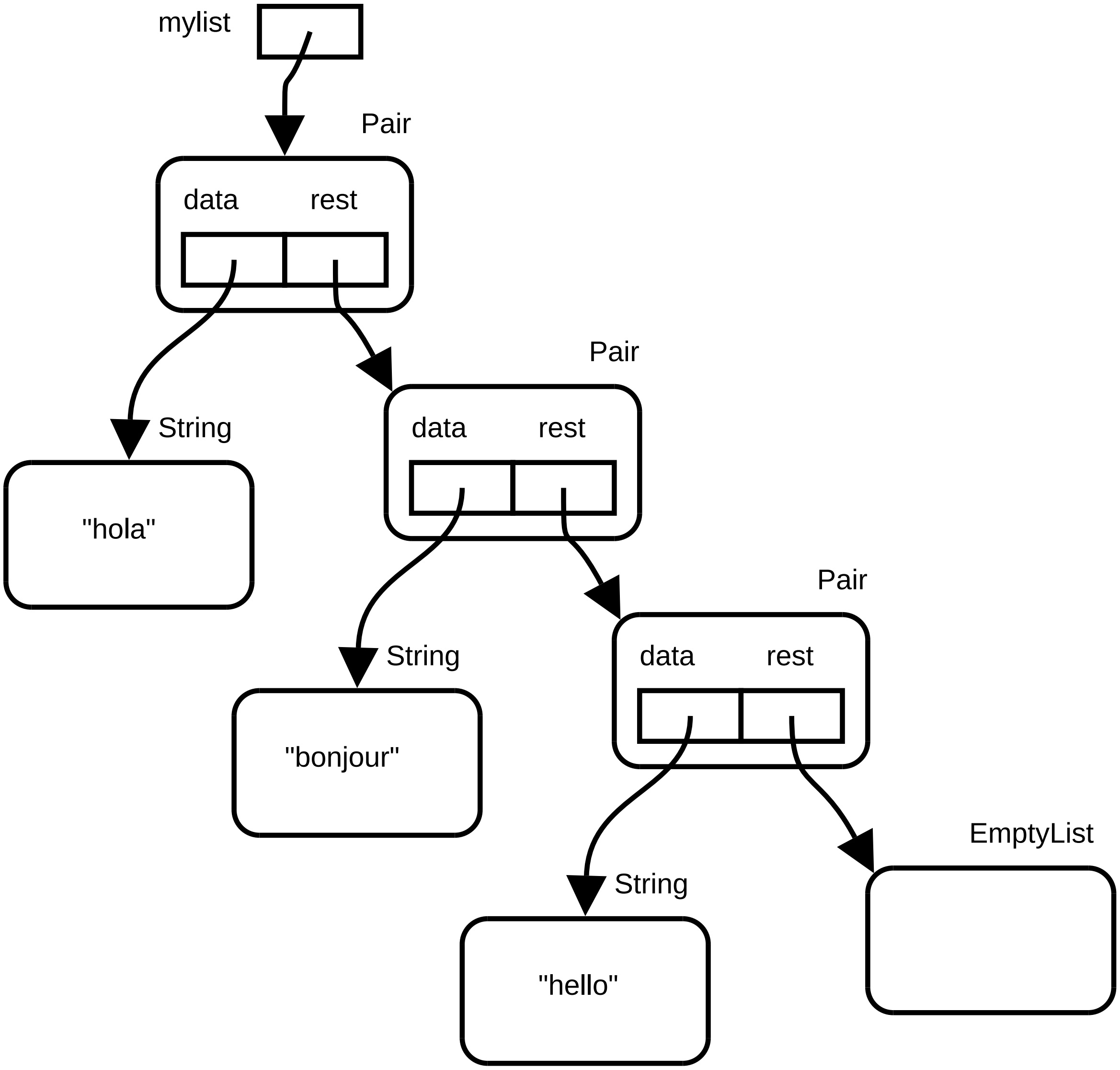
Recursive data-structures



Recursive data-structures



Recursive data-structures



Recursive data-structures

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

To obtain item number n in a list *l*, do:

1. If *l* is an empty list:

   (a) throw an exception

2. Otherwise (it is a pair)

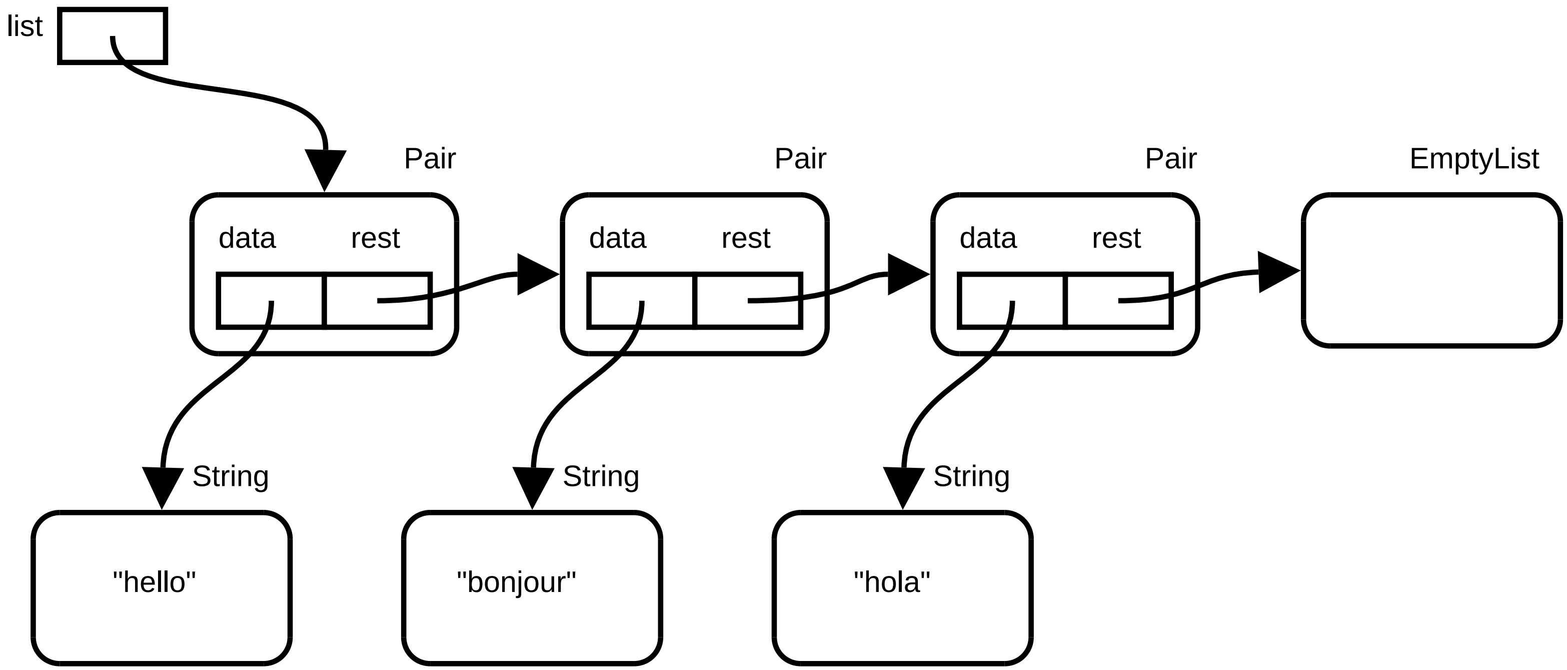
   (a) If n is 0:
      i. return the *data* of the pair

   (b) Otherwise:
      i. return item number $n - 1$ of the *rest* of the list

Recursive data-structures

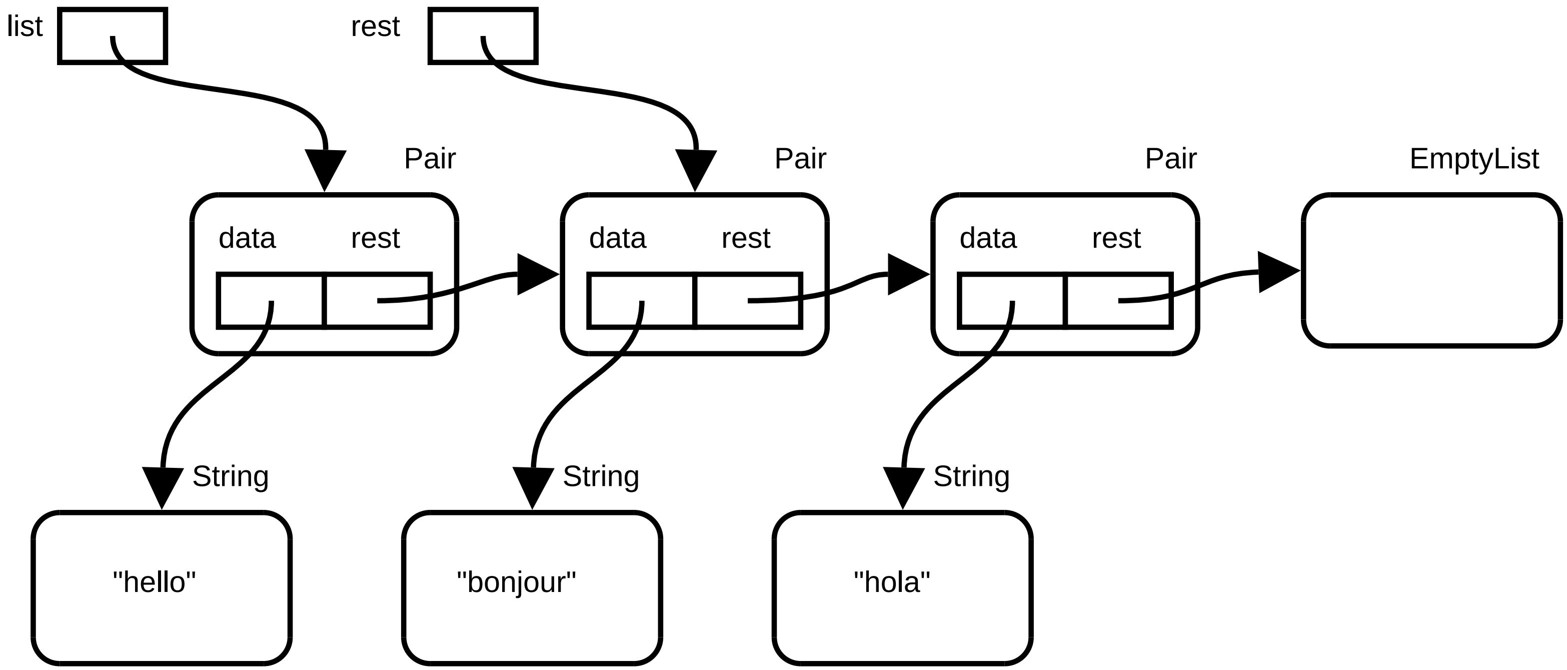
```
static Object get(List l, int n)
throws NoSuchElementException
{
  if (l instanceof EmptyList)
  {
    throw new NoSuchElementException();
  }
  else
  {
    Pair p = (Pair)l;
    if (n == 0)
    {
      Object data = p.getData();
      return data;
    }
    else
    {
      List rest = p.getRest();
      return get(rest, n - 1);
    }
  }
}
```

Recursive data-structures



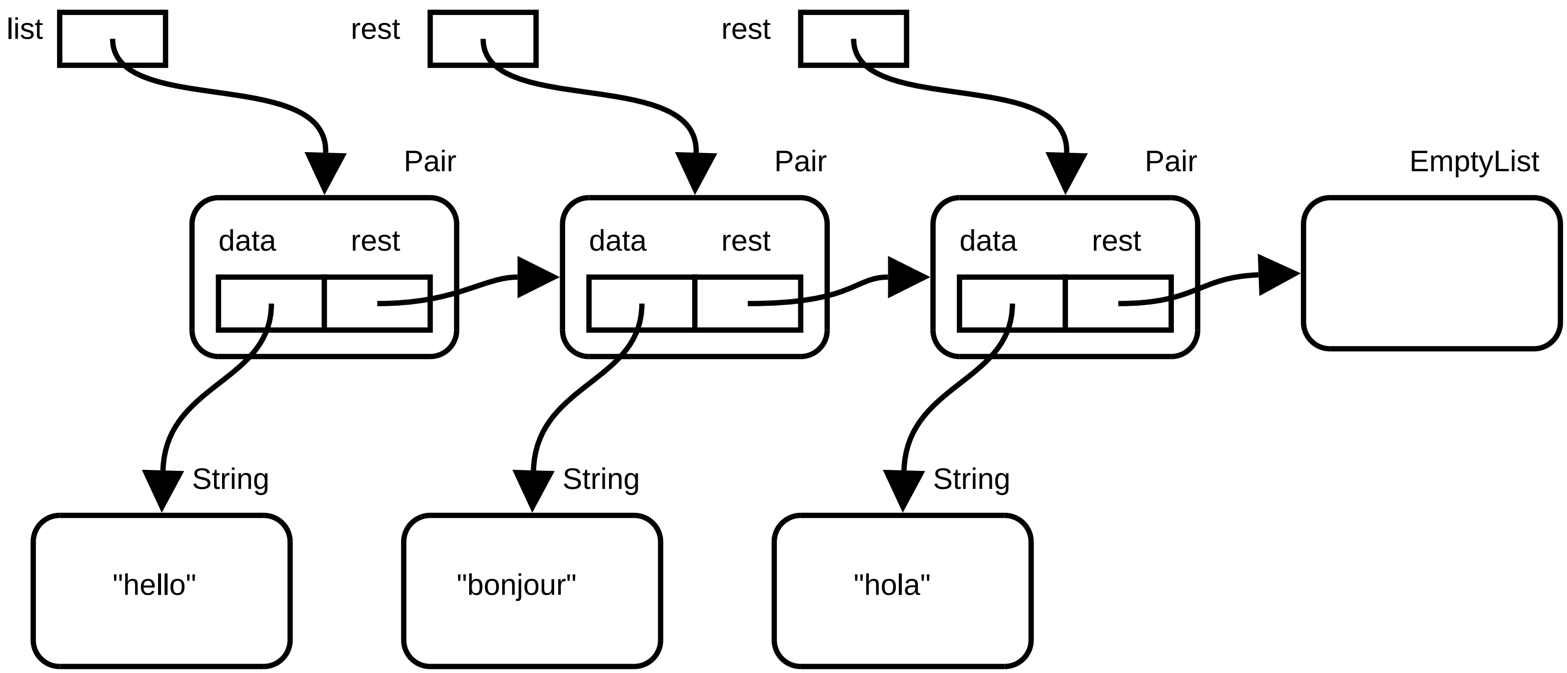
```
get(list, 2)
```

Recursive data-structures



```
return get(rest, 1)
```

Recursive data-structures



```
return get(rest, 0)
```

returns “hola”

Recursive data-structures

Design principle:

Operations on a particular kind of object should be implemented as methods in the object's class

Recursive data-structures

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

Recursive data-structures

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

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 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

```
static Object get(List l, int n)
throws NoSuchElementException
{
  if (l instanceof EmptyList)
  {
    throw new NoSuchElementException();
  }
  else
  {
    Pair p = (Pair)l;
    if (n == 0)
    {
      Object data = p.getData();
      return data;
    }
    else
    {
      List rest = p.getRest();
      return get(rest, n - 1);
    }
  }
}
```

Recursive data-structures

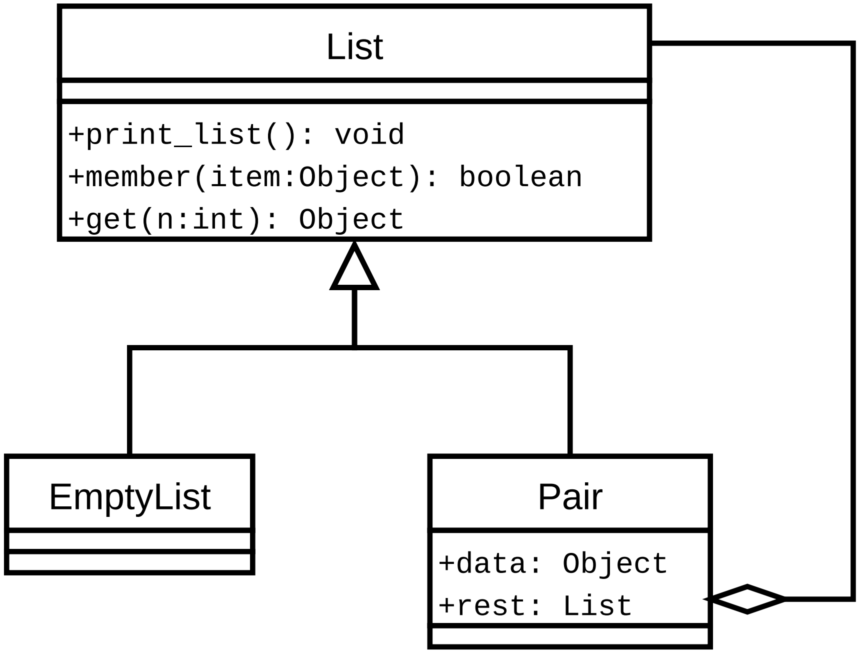
```
Object get(int n)
throws NoSuchElementException
{
  if (this instanceof EmptyList)
  {
    throw new NoSuchElementException();
  }
  else
  {
    Pair p = (Pair)this;
    if (n == 0)
    {
      Object data = p.getData();
      return data;
    }
    else
    {
      List rest = p.getRest();
      return rest.get(n - 1);
    }
  }
}
```

Recursive data-structures

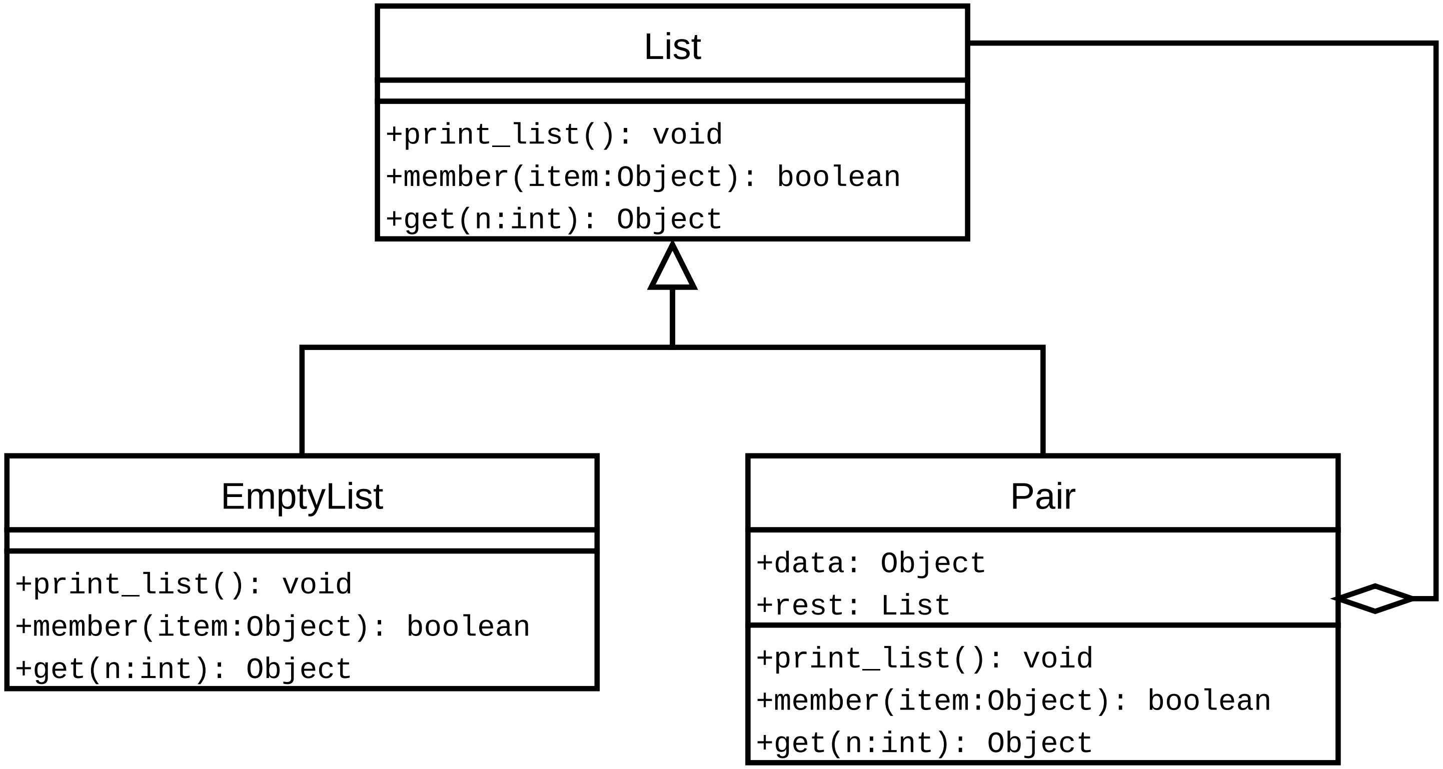
Design principle:

Sometimes a set of classes can be simplified by pushing responsibilities from a class to its subclasses

Recursive data-structures



Recursive data-structures



Recursive data-structures

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

Recursive data-structures

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

  abstract boolean member(Object item);

  abstract Object get(int n);
}
```

Recursive data-structures

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

Recursive data-structures

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

Recursive data-structures

```
class EmptyList extends List
{
  void print_list()
  {
    System.out.print("");
  }
  boolean member(Object item)
  {
    return false;
  }
  Object get(int n)
  throws NoSuchElementException
  {
    throw new NoSuchElementException();
  }
}
```

Recursive data-structures

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

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

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

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)
  { ... }
  Object get(int n)
  { ... }
}
```

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
    }
  }
}
```

Recursive data-structures

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

  ...

  Object get(int n) throws NoSuchElementException
  {
    Pair p = (Pair)this;
    if (n == 0)
    {
      Object data = p.getData();
      return data;
    }
    else
    {
      List rest = p.getRest();
      return rest.get(n - 1);
    }
  }
}
```

Recursive data-structures

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

  ...

  Object get(int n) throws NoSuchElementException
  {
    if (n == 0)
    {
      return data;
    }
    else
    {
      return rest.get(n - 1);
    }
  }
}
```

Recursive data-structures

- Terminology:
  - Linked-list: List
  - Node: Pair
  - Next: Rest

Recursive data-structures

- Alternative implementations of linked-lists:
  - Use constant `null` instead of class `EmptyList`
  - Class list has attributes:
    - First node (front)
    - Last node (tail)
  - Doubly-linked-lists:
    - Each node has a reference to the *next* and the *previous* node
  - Circular linked-lists:
    - The next of the last node is the first node

Recursive data-structures

- Useful applications of linked-lists:
  - Queues
  - Stacks

Recursive data-structures

- Useful applications of linked-lists:
  - Queues: (FIFO or LILO)
    - Elements can be removed only from the front
    - Elements can be added only at the end
  - Stacks: (LIFO or FILO)
    - Elements can be removed only from the front (top)
    - Elements can be added only at the front (top)

Recursive data-structures

Other recursive data-structures:

- Trees:
  - A tree is either:
    - A node with two subtrees, or
    - A leaf (a node with no subtrees)

What this course is about

- This course is an introduction to *computer programming*
- Computer programming: solving problems involving information by means of a computer

What this course is *not* about

- This course is ***not*** about...
  - ...how to use a computer
  - ...how to use software applications
  - ...how to use the Operating System
  - ...how to send e-mail
  - ...how to surf the Web
  - ...how to create Web pages
  - ...how to fix your printer
  - ...how to become a hacker
  - ...how to manage a computer system (installing software, fixing problems, etc.)
- There is no course in Computer Science about how to use computers, in the same way that there is no course in Mechanical Engineering that teaches how to drive a car or operate some machinery.

Objectives

- To learn:
  - ...a methodology to understand and solve problems involving information
  - ...how to think computationally
  - ...how to create simple algorithms
  - ...how to design and implement computer programs using the Java programming language
  - ...how to solve problems in an Object-Oriented manner

- **This is neither a "computers course" nor a "Java course."**

Why is computer programming useful

- General benefits
  - Introduces a structured way of thinking, analysing and solving problems

- Applications
  - Engineering and Physical sciences: modelling and simulation
  - Biological sciences: Bioinformatics, Eco-system modelling
  - Geography, Enviromental Studies and Urbanism: Geographic Information Systems
  - Economics: Economic forecasting and analysis, Economic modelling
  - Management: Databases, Information Systems, Process optimization
  - Software development

Computer Science

“Computer Science is no more about computers than Astronomy is about telescopes” - Edsger Dijkstra

- Computer Science:
  - Theory
  - Building systems

Computer Science

“Computer Science is no more about computers than Astronomy is about telescopes” - Edsger Dijkstra

- Computer Science:
  - Theory
  - Algorithms
  - Building systems

Computer Science

"Computer Science is no more about computers than Astronomy is about telescopes" - Edsger Dijkstra

- Computer Science:
  - Theory
  - Algorithms: studies algorithms in the abstract
    - Algorithm design techniques
    - Complexity analysis
    - Data-structures
    - etc.
  - Building systems

Computer Science

“Computer Science is no more about computers than Astronomy is about telescopes” - Edsger Dijkstra

- Computer Science:
  - Theory: studies the mathematical foundations of computation
  - Algorithms
  - Building systems

Computer Science

"Computer Science is no more about computers than Astronomy is about telescopes" - Edsger Dijkstra

- Computer Science:
  - Theory: studies the mathematical foundations of computation
    - Theory of Computation: Automata and Formal languages
    - Complexity Theory
    - Semantics
  - Algorithms
  - Building systems

Computer Science

"Computer Science is no more about computers than Astronomy is about telescopes" - Edsger Dijkstra

- Computer Science:
  - Theory: studies the mathematical foundations of computation
  - Algorithms
  - Building systems
    - Computer Graphics
    - Robotics
    - Artificial Intelligence
    - Distributed Processing
    - Operating Systems
    - Modelling and Simulation
    - Numerical computation
    - Computer games
    - Databases
    - ...etc.

The end