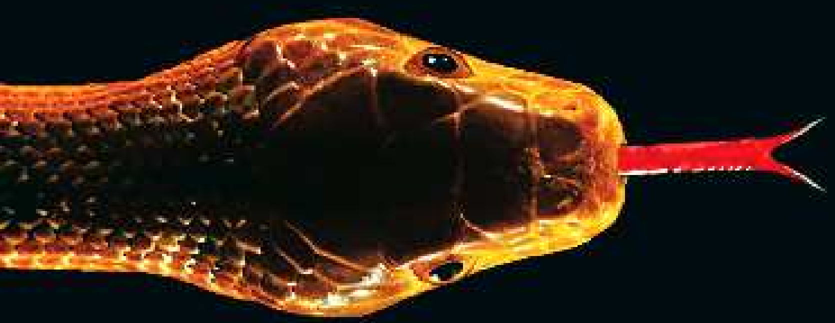


Teaching Strategies for Reinforcing Structural Recursion with Lists

Michael H. Goldwasser   David Letscher

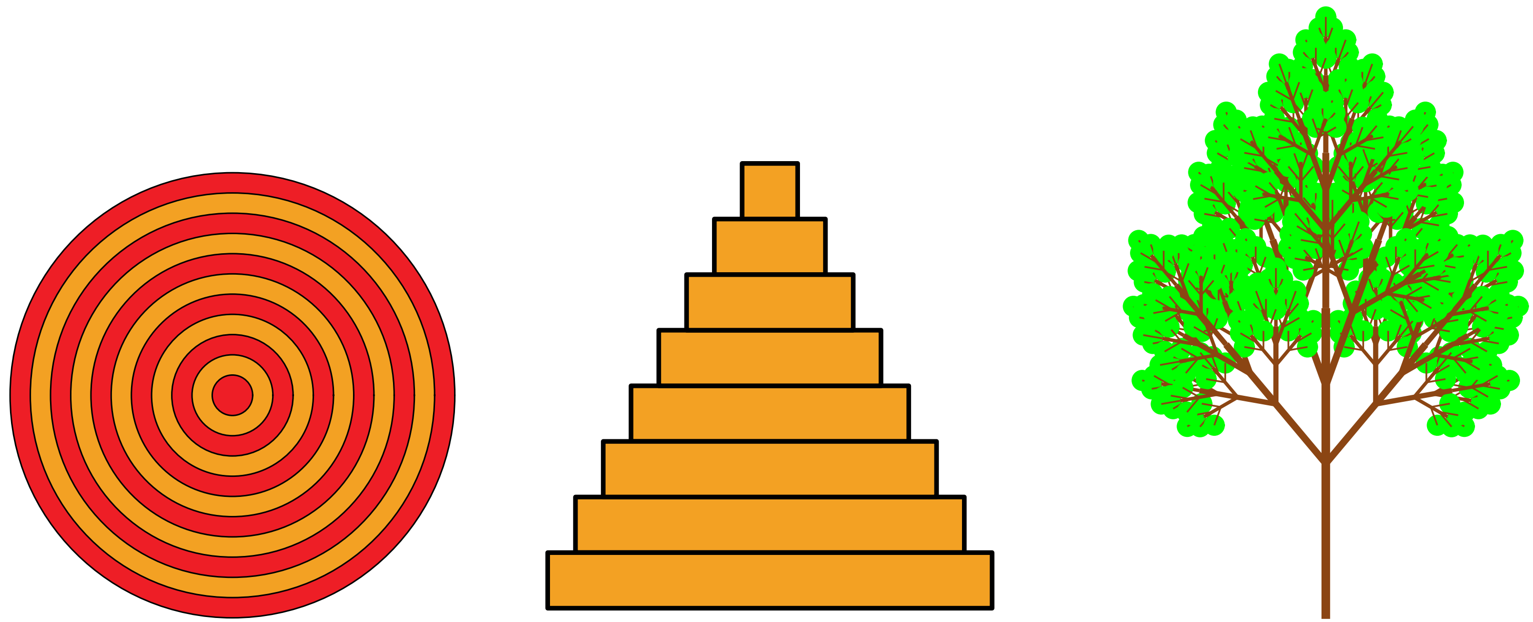
Saint Louis University

Active Learning

I need some volunteers for today

Structural Recursion

For an object-oriented CS1, structural recursion can be more natural than functional recursion.



An object is composed of a basic shape and a (recursive) instance of the same class.

Beyond Graphics

Pro: Graphics are fun and tangible.

Beyond Graphics

Pro: Graphics are fun and tangible.

Con: recursive patterns are generally limited ("draw outer, draw rest"; "move outer, move rest")

Beyond Graphics

Pro: Graphics are fun and tangible.

Con: recursive patterns are generally limited (“draw outer, draw rest”; “move outer, move rest”)

Our goal is to provide a tangible presentation for a non-graphical example of structural recursion (namely purely-recursive lists).

Python's List Class

Python supports a list class as a standard container.

Python supports a list class as a standard container.

Disclaimer: the internal implementation is not truly recursive; its an expandible array akin to Java's ArrayList or C++'s vector.

Python's List Class

Python supports a list class as a standard container.

Disclaimer: the internal implementation is not truly recursive; its an expandible array akin to Java's ArrayList or C++'s vector.

Public Interface: Our students are very familiar with use of this class and its menu of behaviors (we use lists from the opening weeks of CS1).

Python’s List Class

Overview
Active Learning
Structural Recursion
Beyond Graphics
Python’s List Class
Role Playing
Variants
Mutators
Implementation
Conclusions

Python supports a list class as a standard container.

Disclaimer: the internal implementation is not truly recursive; its an expandible array akin to Java’s ArrayList or C++’s vector.

Public Interface: Our students are very familiar with use of this class and its menu of behaviors (we use lists from the opening weeks of CS1).

This allows us to decouple two potentially intertwined concepts:

1. the use of recursion
2. the abstraction of a container class

Emulating Python’s List Class

We rely on the familiar public interface by precisely emulating Python’s list class, including behaviors such as:

count(value)	__len__( )
index(value)	__contains__(value)
append(value)	__getitem__(index)
insert(index, value)	__setitem__(index, value)
remove(value)	__repr__( )

This allows us to sidestep the design issue of parameterizing the recursion.

Role Playing

Role Playing

Classic activity for teaching object orientation.

Role Playing

Classic activity for teaching object orientation.

Classic activity for teaching (functional) recursion.

Role Playing

Classic activity for teaching object orientation.

Classic activity for teaching (functional) recursion.

Limited history for the combination of these ideas.

Ground Rules for Students

OurList Class: an instance will be represented recursively using two attributes:

- _head: a reference to the first element (if any)
- _rest: a reference to a secondary list with all remaining elements (if any)

Our base case is an empty list, represented with both _head and _rest set to the **None** reference.

An empty list is a natural concept for our students because Python's default list instance is empty.

State Information

Each actor is given a slip of paper that represents his/her state information.

: OurList
_head: _rest:

State Information

Each actor is given a slip of paper that represents his/her state information.

Sharon : OurList
_head: `'E'` _rest: Per

Example: here is the slip currently held by Sharon .

State Information

Overview
Role Playing
Role Playing
Ground Rules
State Information
Message Passing
Point of View
Sequence Diagram
Local View
Variants
Mutators
Implementation
Conclusions

Each actor is given a slip of paper that represents his/her state information.

Matthew : OurList
_head: **None** _rest: **None**

Example: here is the slip currently held by Matthew .

Message Passing

We enforce strict “message passing” for all communication.

Activation records are sent inside a tennis ball.

ACTIVATION RECORD
Sent to:
Method:
Parameters (if any):
Please return to:
Return Value (if any):

Message Passing

We enforce strict “message passing” for all communication.

Activation records are sent inside a tennis ball.

ACTIVATION RECORD	
Sent to:	Errol
Method:	count
Parameters (if any):	'E'
Please return to:	Michael
Return Value (if any):	

Let’s get started with a call Errol .count('E')

Errol 's Point of View

Errol : OurList	
_head:	'F'
_rest:	Sharon

ACTIVATION RECORD	
Sent to:	Errol
Method:	count
Parameters (if any):	'E'
Please return to:	Michael
Return Value (if any):	

Errol 's Point of View

Errol : OurList
_head: 'F' _rest: Sharon

ACTIVATION RECORD	
Sent to:	Errol
Method:	count
Parameters (if any):	'E'
Please return to:	Michael
Return Value (if any):	

ACTIVATION RECORD	
Sent to:	Sharon
Method:	count
Parameters (if any):	'E'
Please return to:	Errol
Return Value (if any):	

Sharon : OurList	
_head:	'E'
_rest:	Per

ACTIVATION RECORD	
Sent to:	Sharon
Method:	count
Parameters (if any):	'E'
Please return to:	Errol
Return Value (if any):	

Sharon 's Point of View

Sharon : OurList	
_head:	'E'
_rest:	Per

ACTIVATION RECORD	
Sent to:	Sharon
Method:	count
Parameters (if any):	'E'
Please return to:	Errol
Return Value (if any):	

ACTIVATION RECORD	
Sent to:	Per
Method:	count
Parameters (if any):	'E'
Please return to:	Sharon
Return Value (if any):	

Sharon 's Point of View

Sharon : OurList	
_head:	'E'
_rest:	Per

ACTIVATION RECORD	
Sent to:	Sharon
Method:	count
Parameters (if any):	'E'
Please return to:	Errol
Return Value (if any):	

ACTIVATION RECORD	
Sent to:	Per
Method:	count
Parameters (if any):	'E'
Please return to:	Sharon
Return Value (if any):	1

Sharon 's Point of View

Sharon : OurList	
_head:	'E'
_rest:	Per

ACTIVATION RECORD	
Sent to:	Sharon
Method:	count
Parameters (if any):	'E'
Please return to:	Errol
Return Value (if any):	2

ACTIVATION RECORD	
Sent to:	Per
Method:	count
Parameters (if any):	'E'
Please return to:	Sharon
Return Value (if any):	1

Errol ’s Point of View

Errol : OurList	
_head:	'F'
_rest:	Sharon

ACTIVATION RECORD	
Sent to:	Errol
Method:	count
Parameters (if any):	'E'
Please return to:	Michael
Return Value (if any):	

ACTIVATION RECORD	
Sent to:	Sharon
Method:	count
Parameters (if any):	'E'
Please return to:	Errol
Return Value (if any):	2

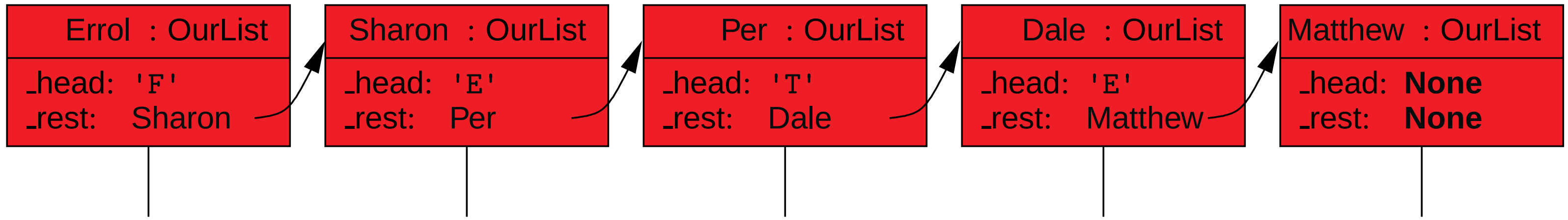
Errol 's Point of View

Errol : OurList	
_head:	'F'
_rest:	Sharon

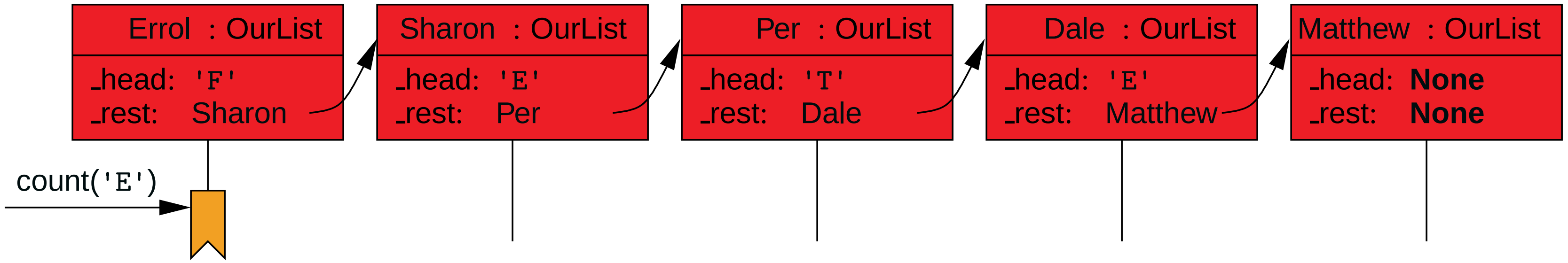
ACTIVATION RECORD	
Sent to:	Errol
Method:	count
Parameters (if any):	'E'
Please return to:	Michael
Return Value (if any):	2

ACTIVATION RECORD	
Sent to:	Sharon
Method:	count
Parameters (if any):	'E'
Please return to:	Errol
Return Value (if any):	2

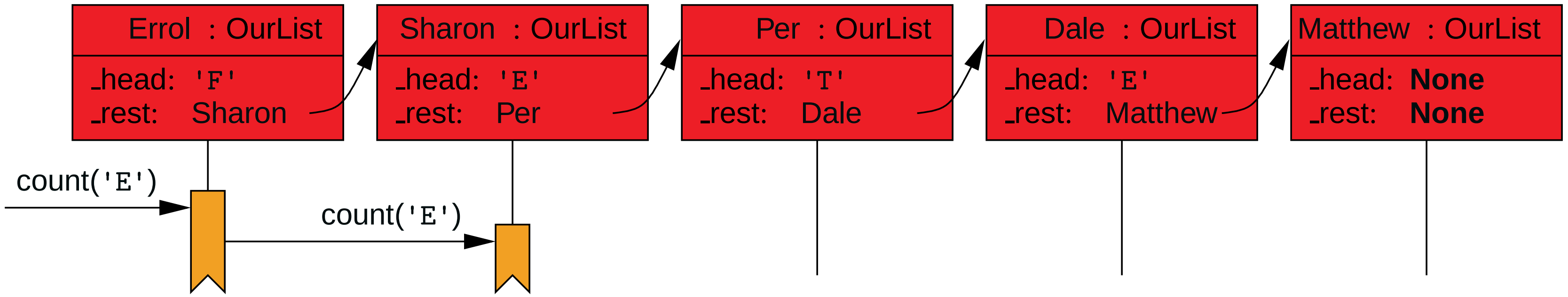
Sequence Diagram



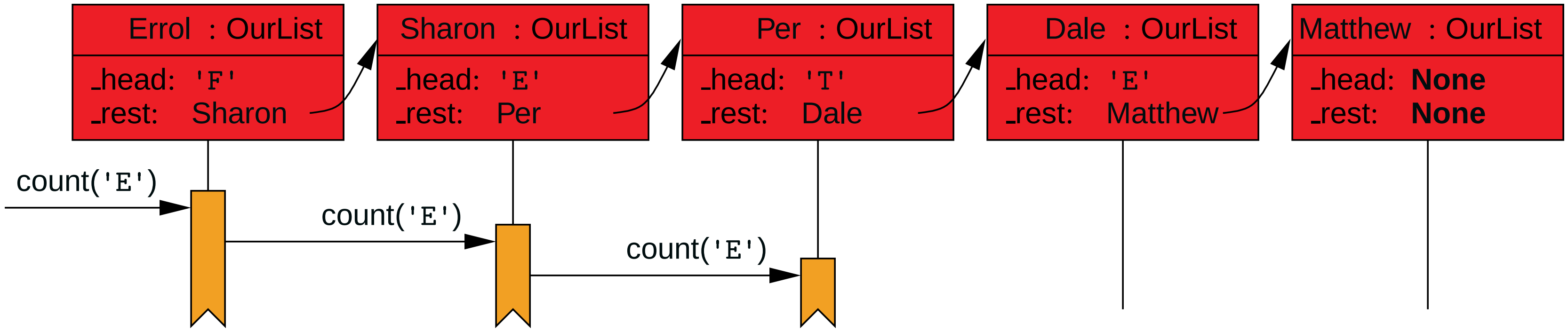
Sequence Diagram



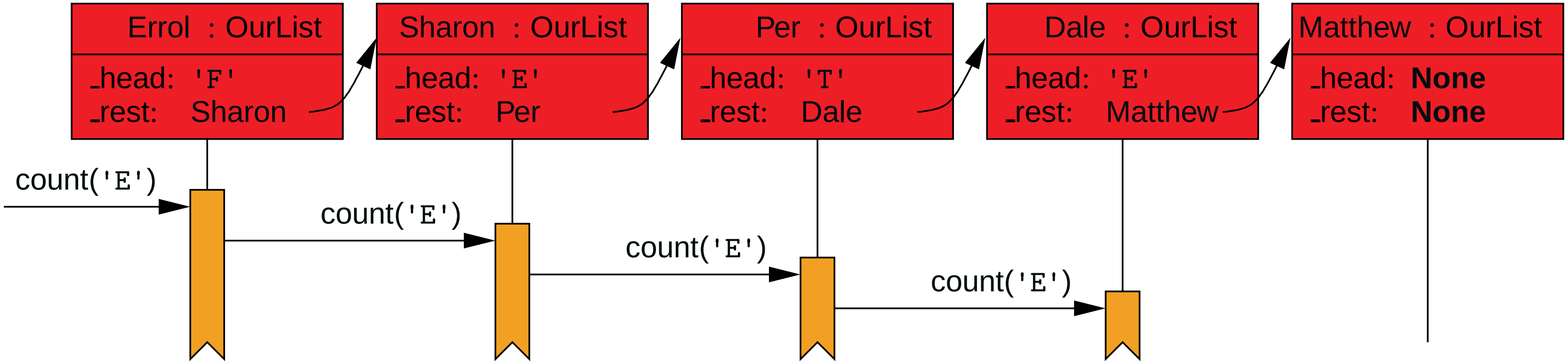
Sequence Diagram



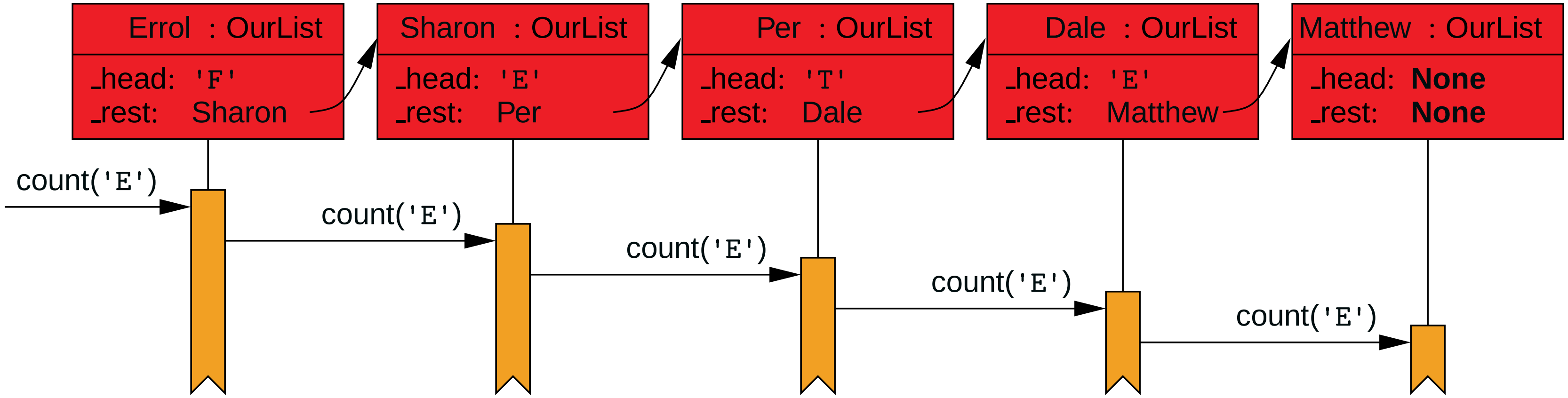
Sequence Diagram



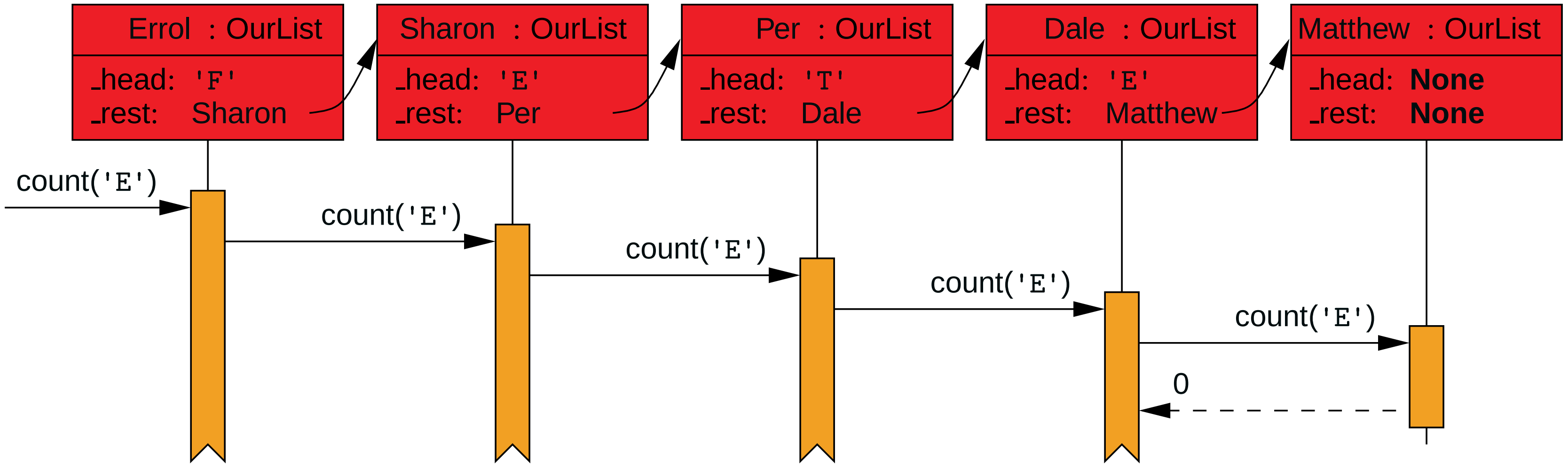
Sequence Diagram



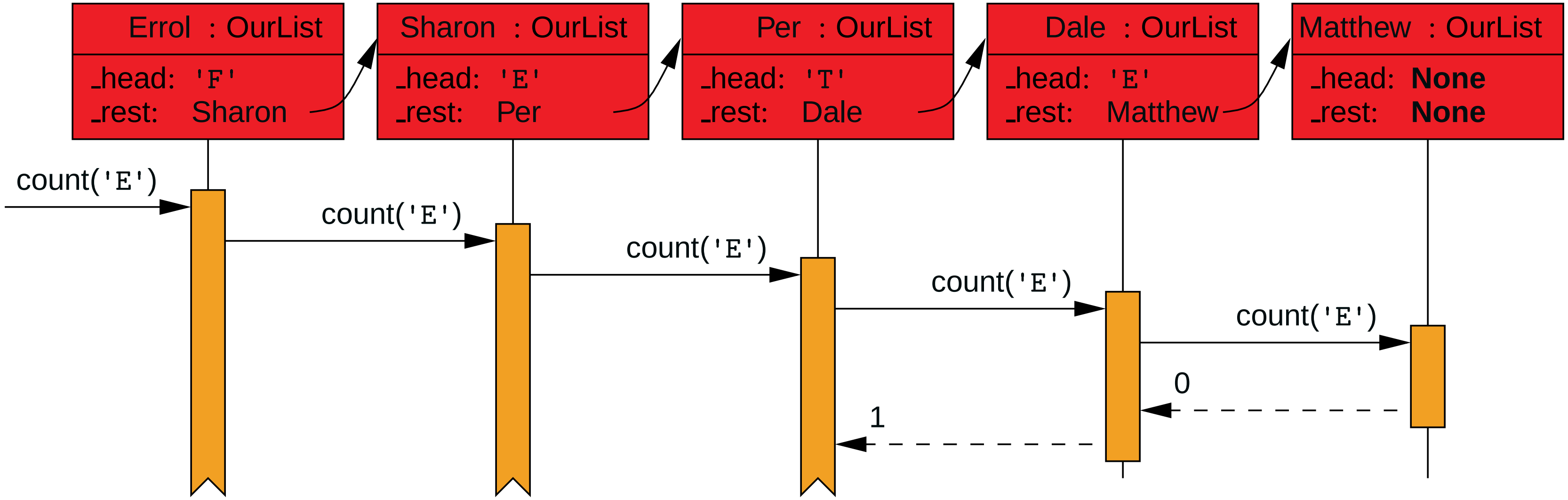
Sequence Diagram



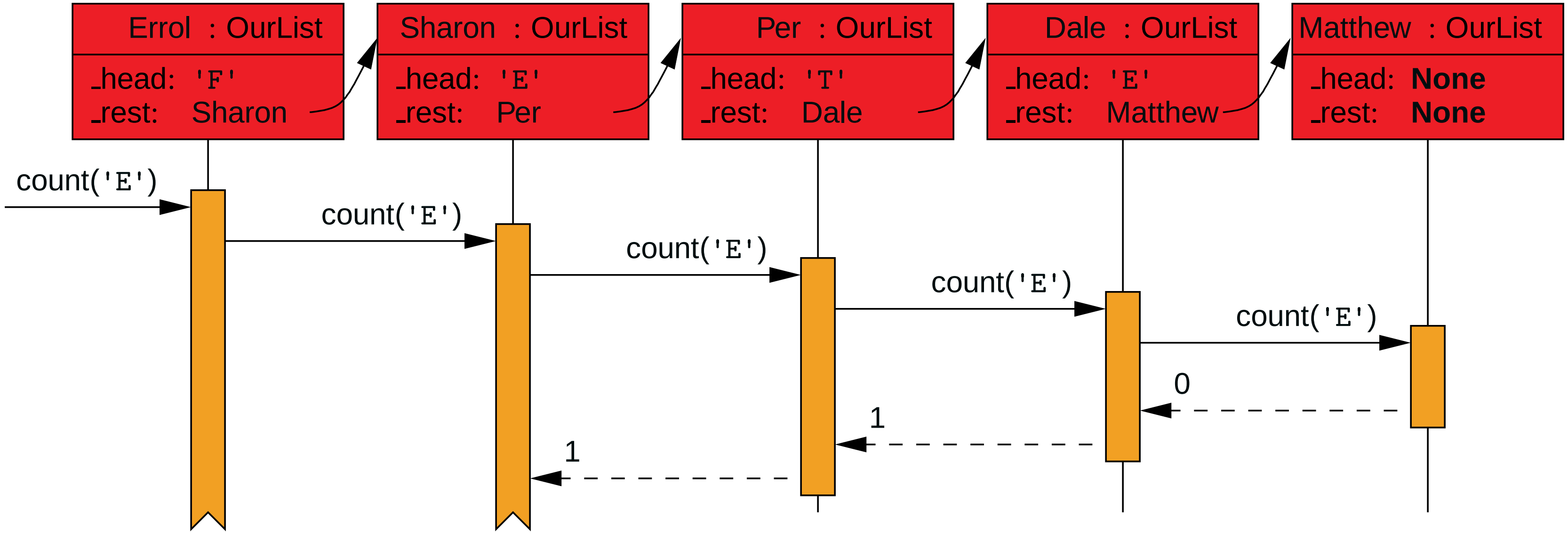
Sequence Diagram



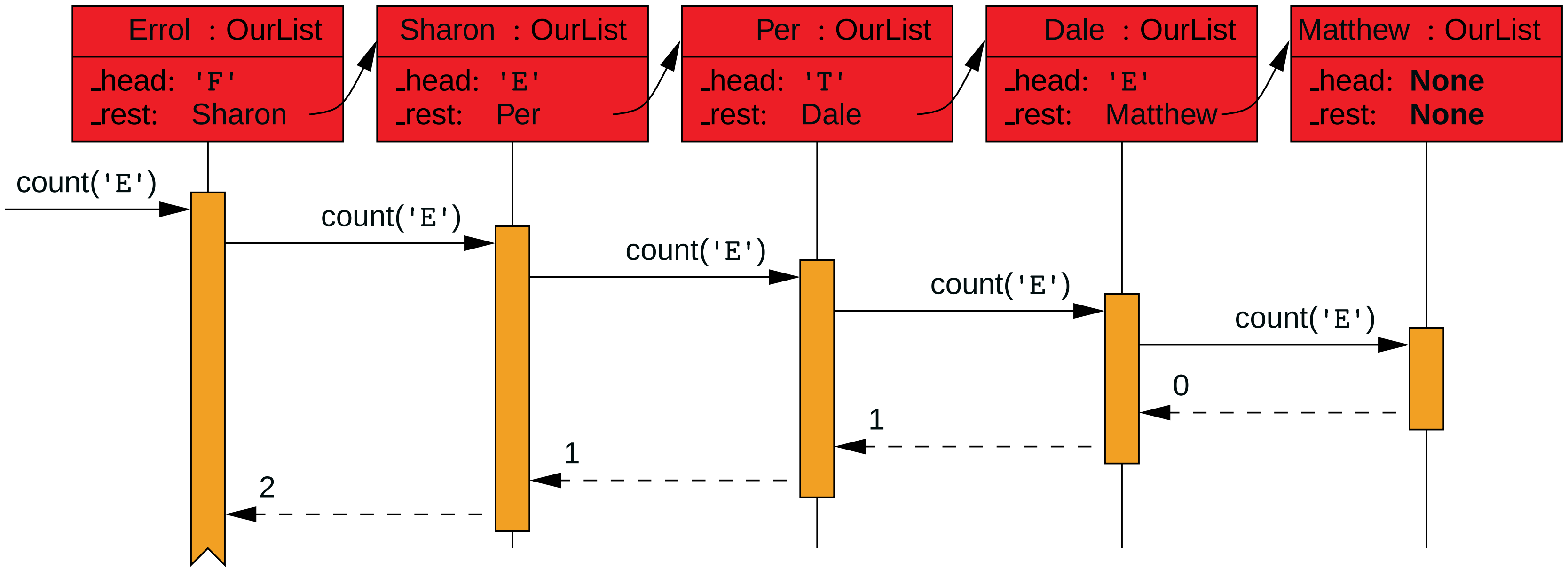
Sequence Diagram



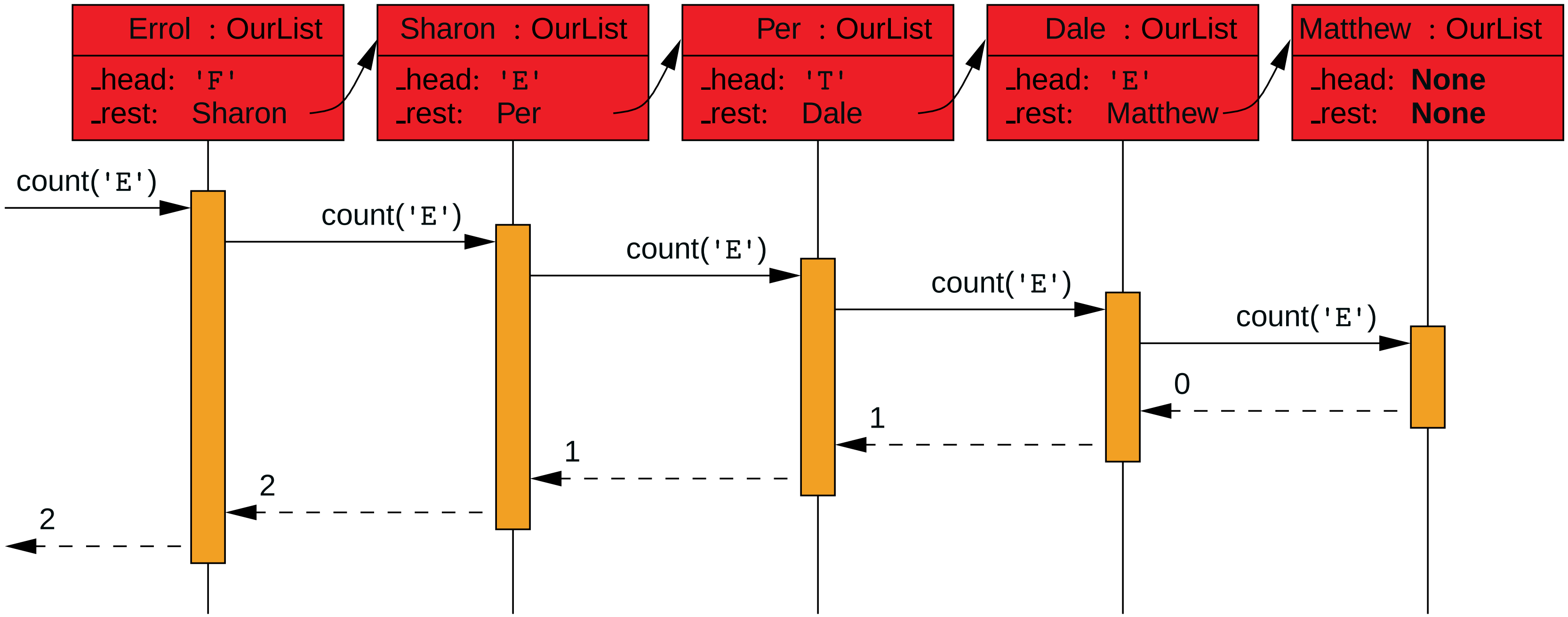
Sequence Diagram



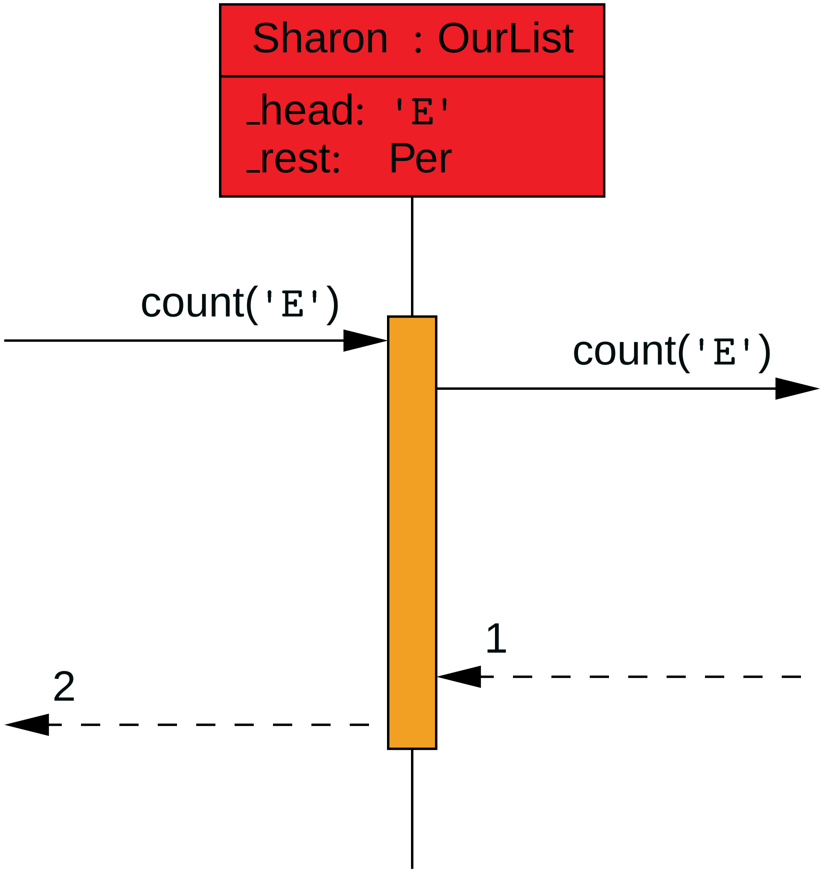
Sequence Diagram



Sequence Diagram

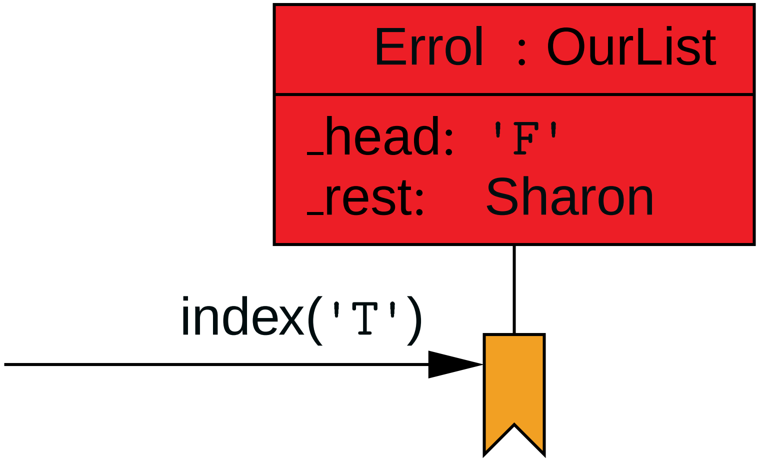


Local View

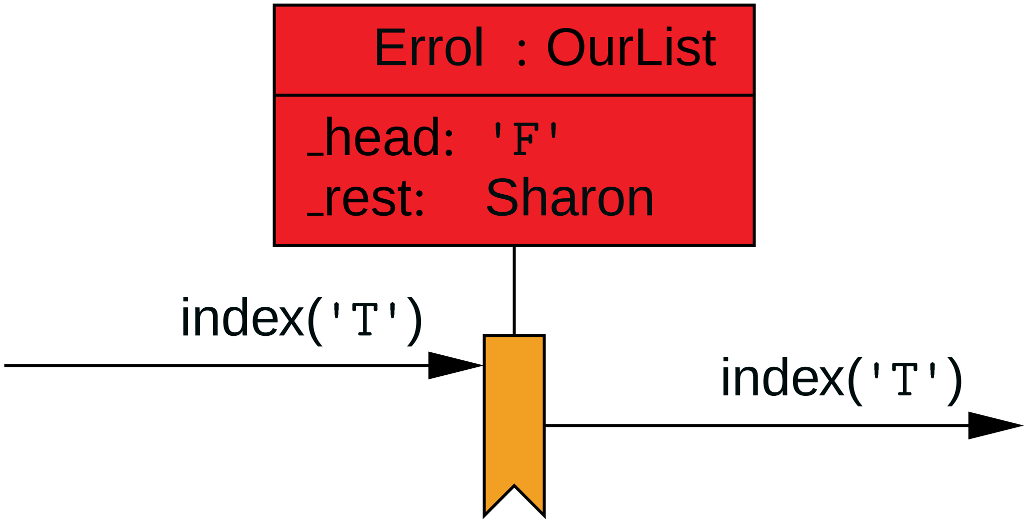


Variants

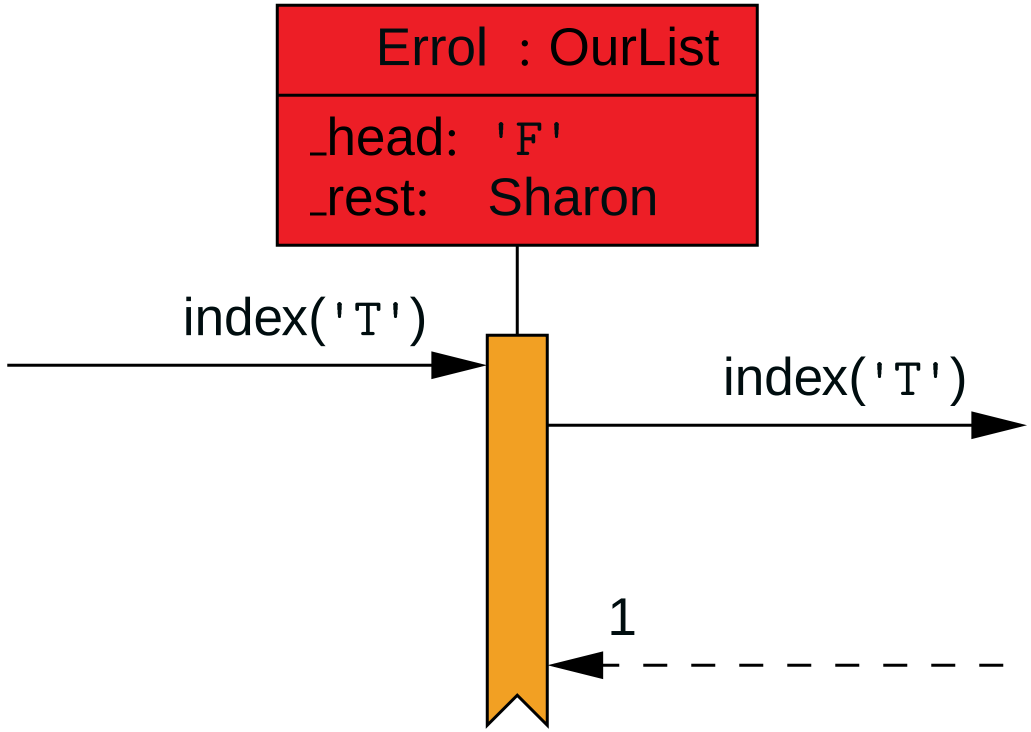
The index method



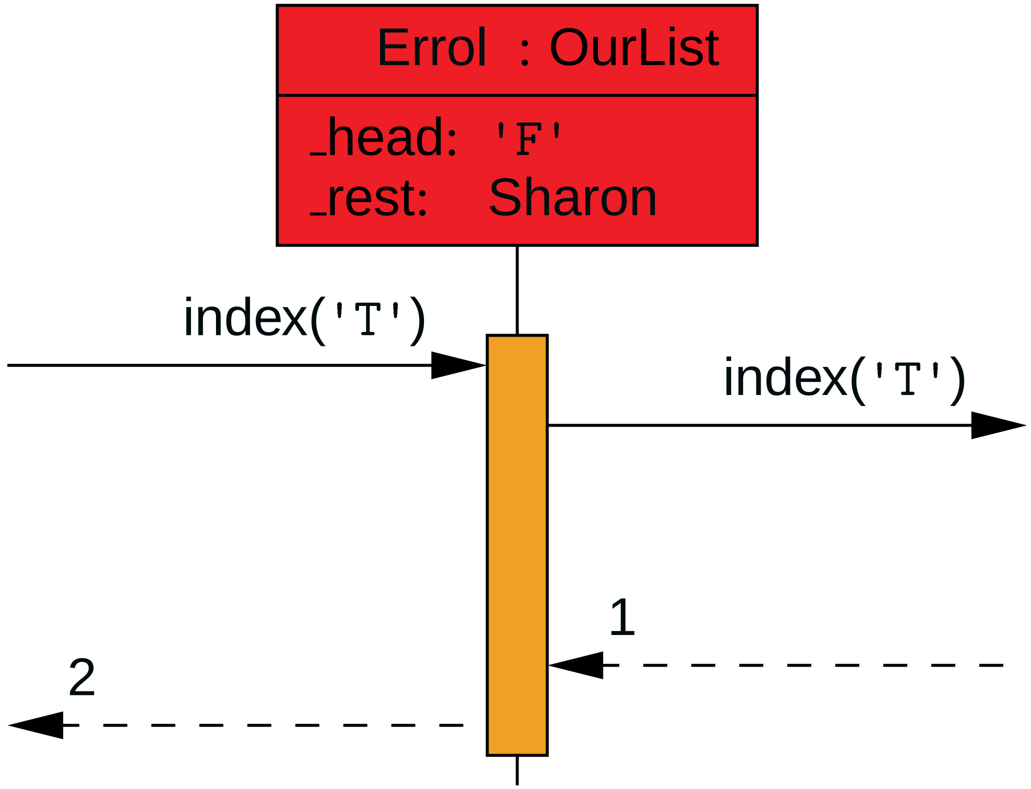
The index method



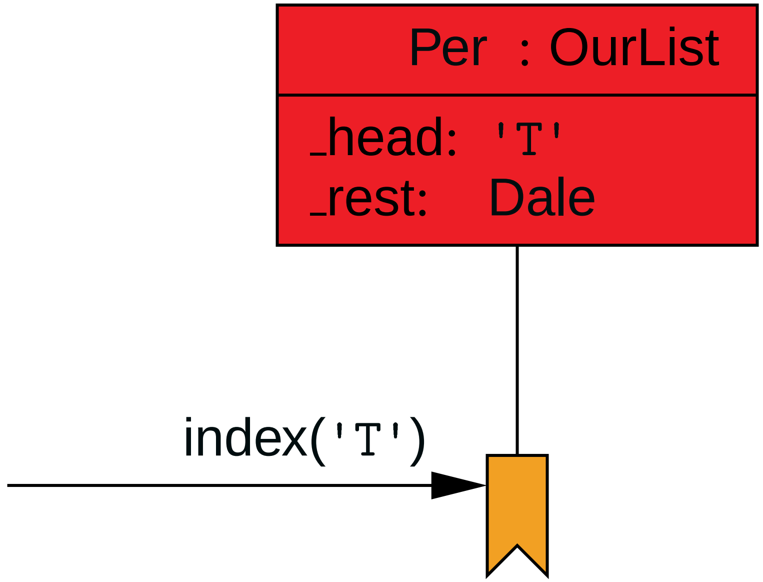
The index method



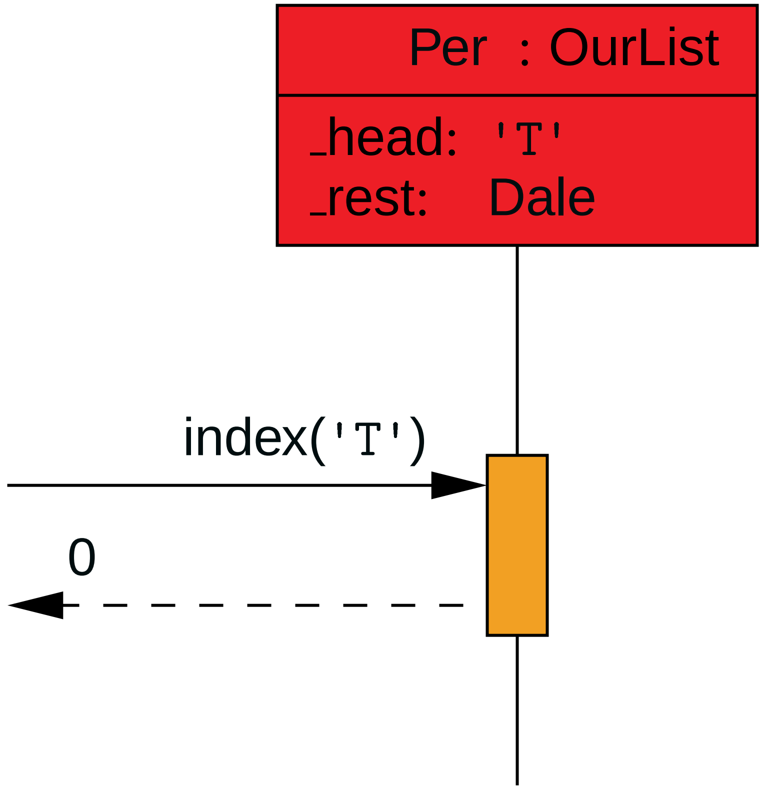
The index method



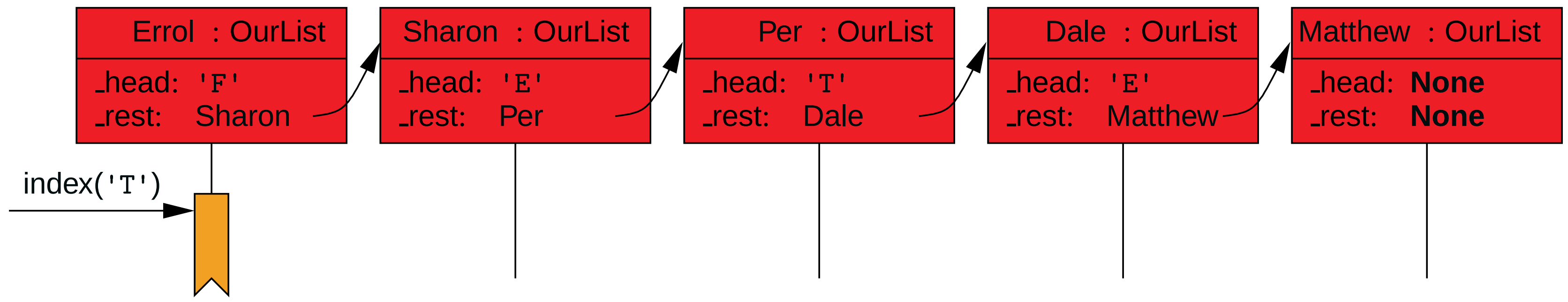
The index method



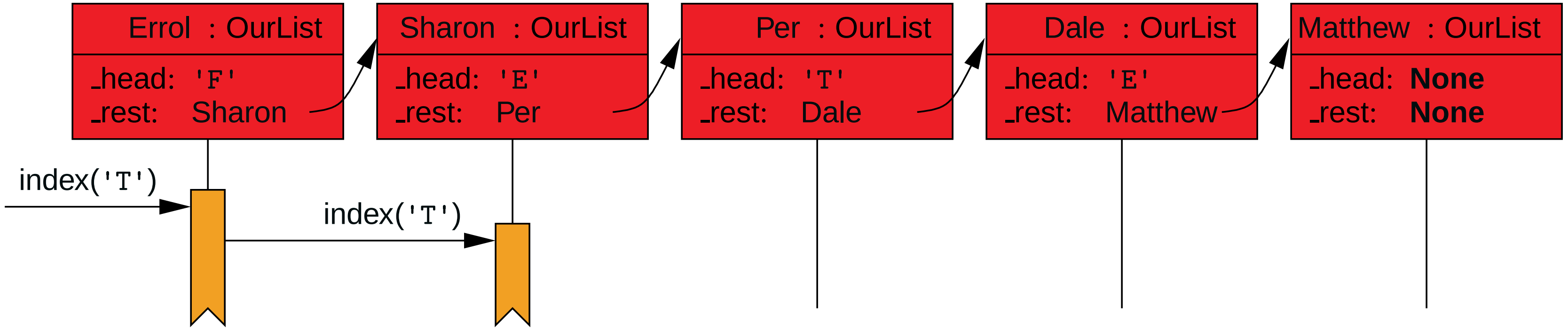
The index method



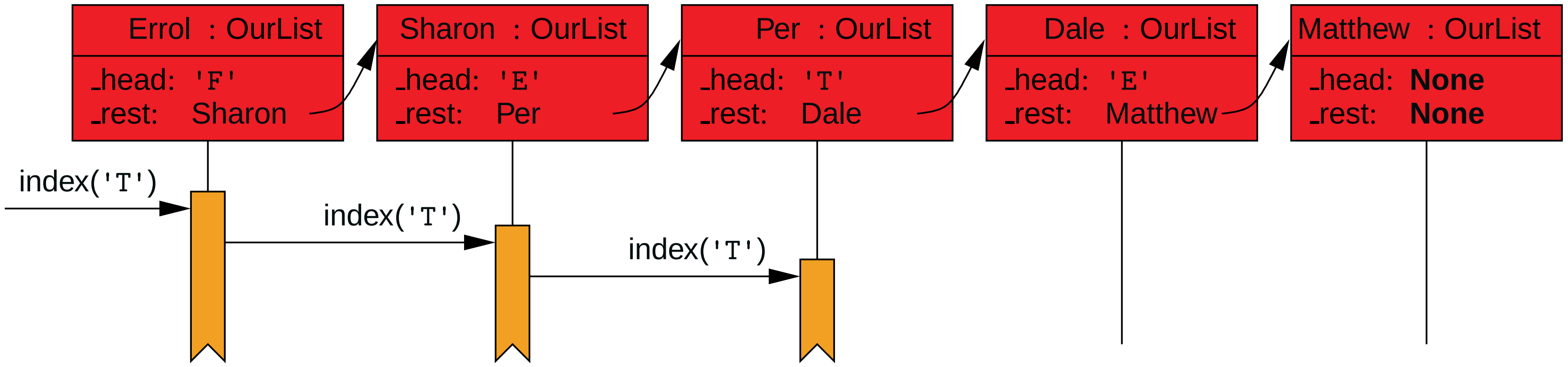
The index method



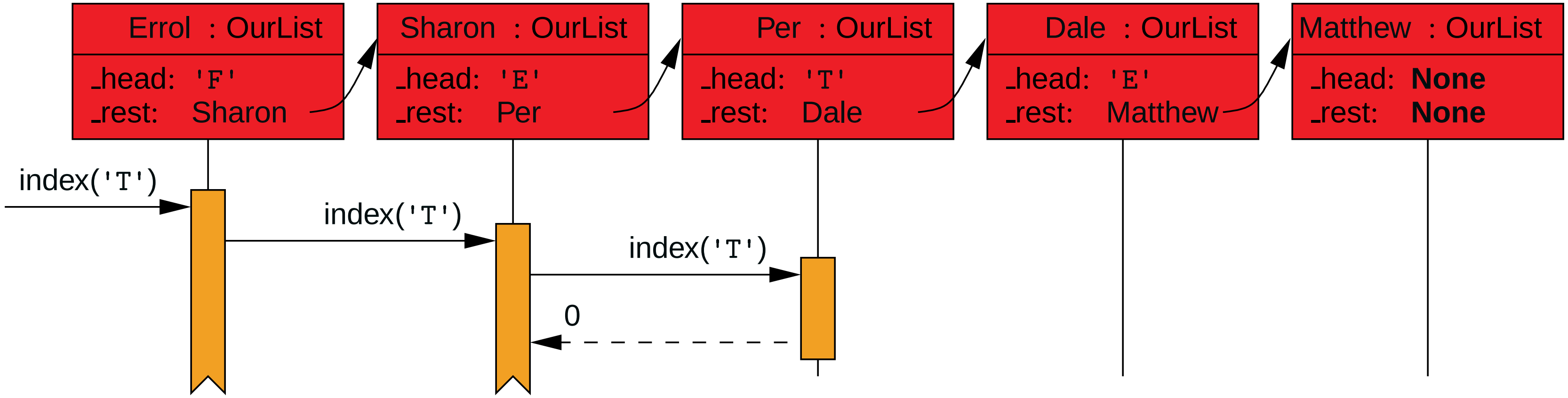
The index method



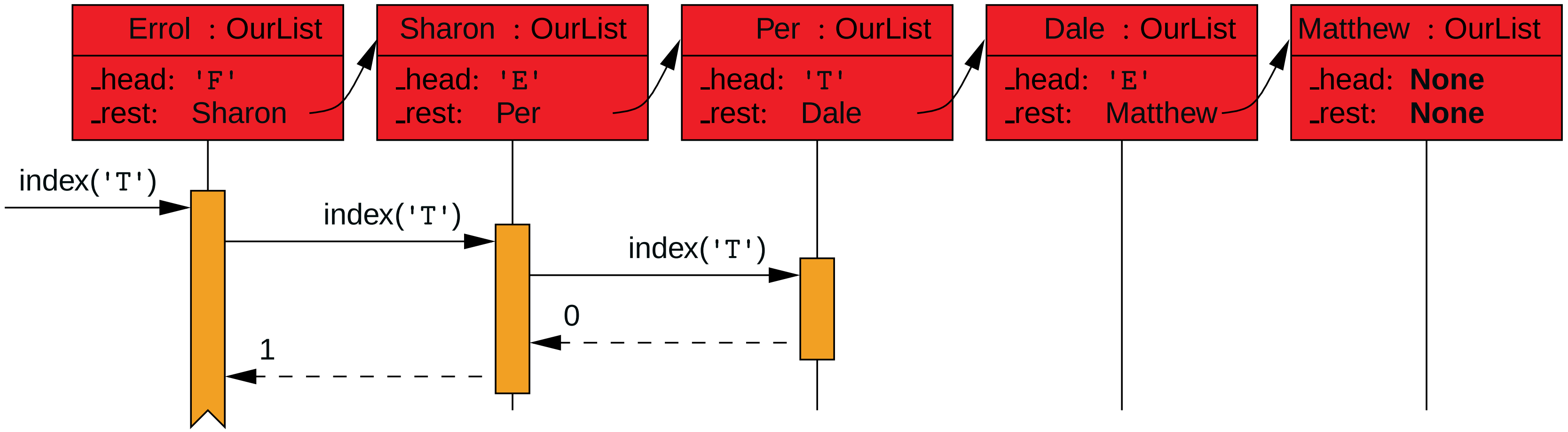
The index method



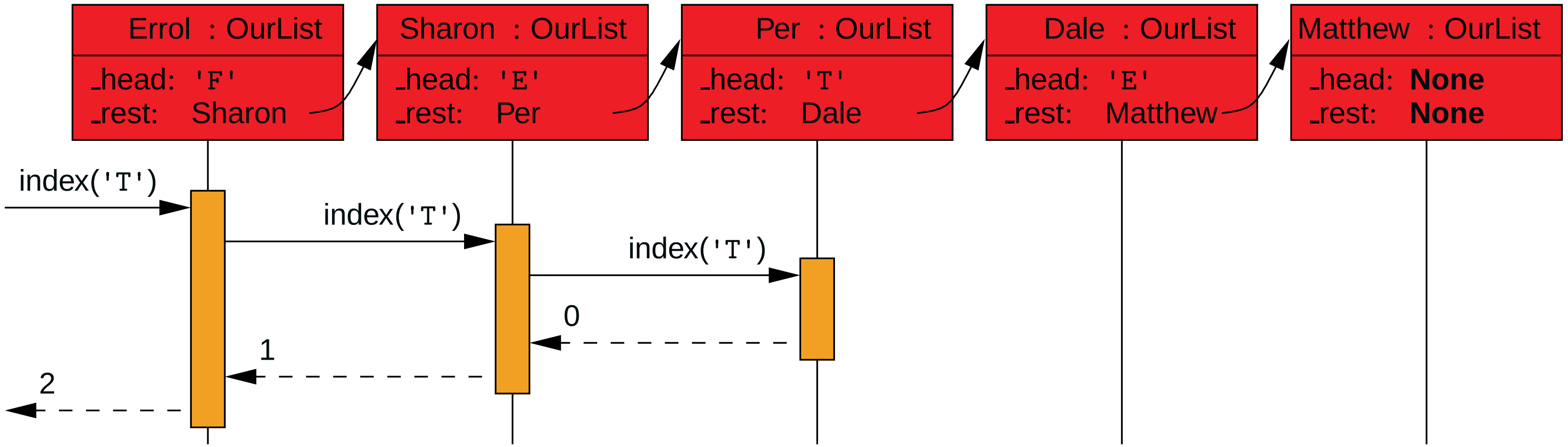
The index method



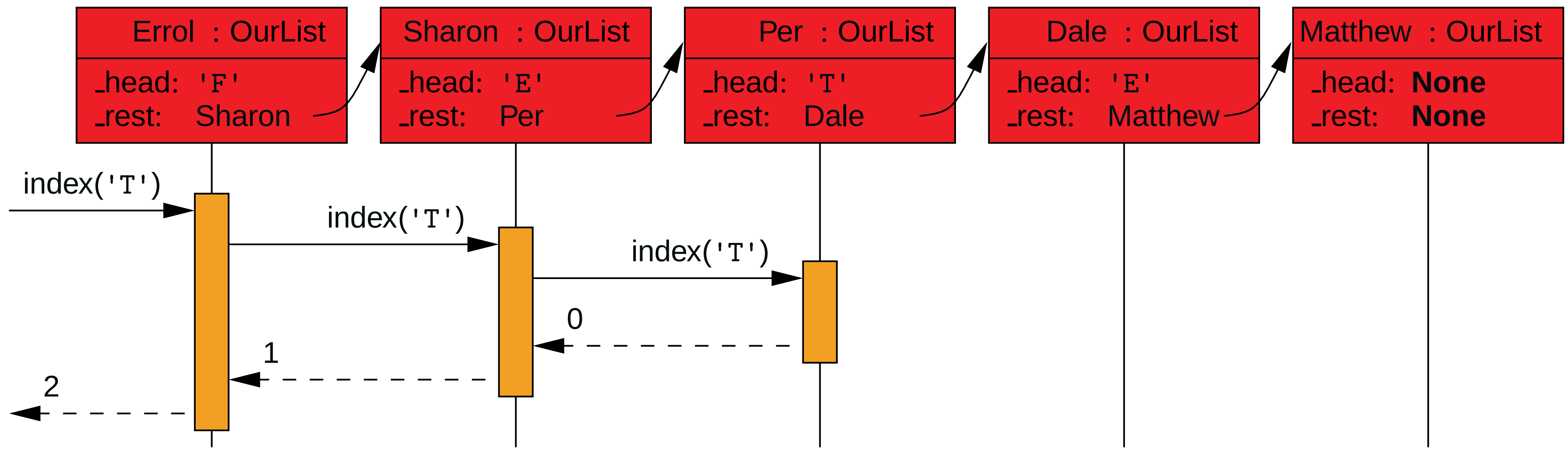
The index method



The index method



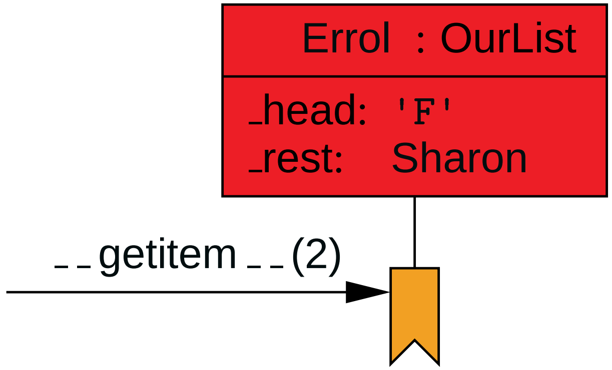
The index method



This differs from count because the recursion does not necessarily proceed to an empty list.

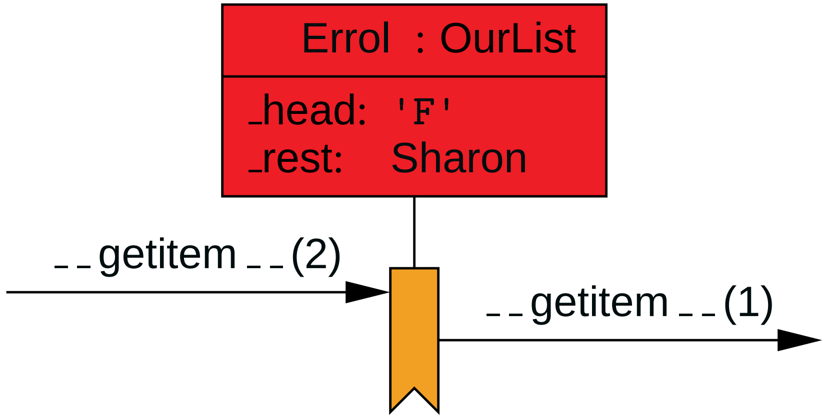
The __getitem__ method

In Python, the operator syntax data[2] is implemented with a call to data.__getitem__(2).



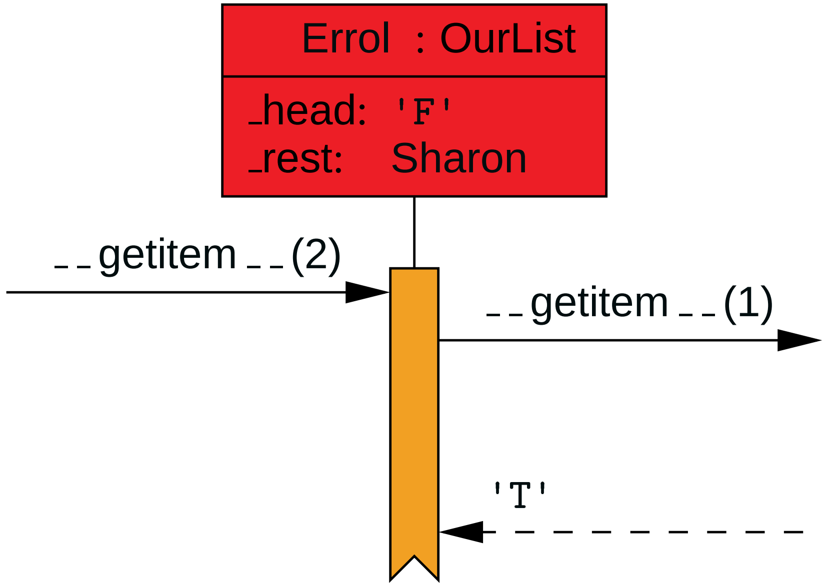
The __getitem__ method

In Python, the operator syntax data[2] is implemented with a call to data.__getitem__(2).



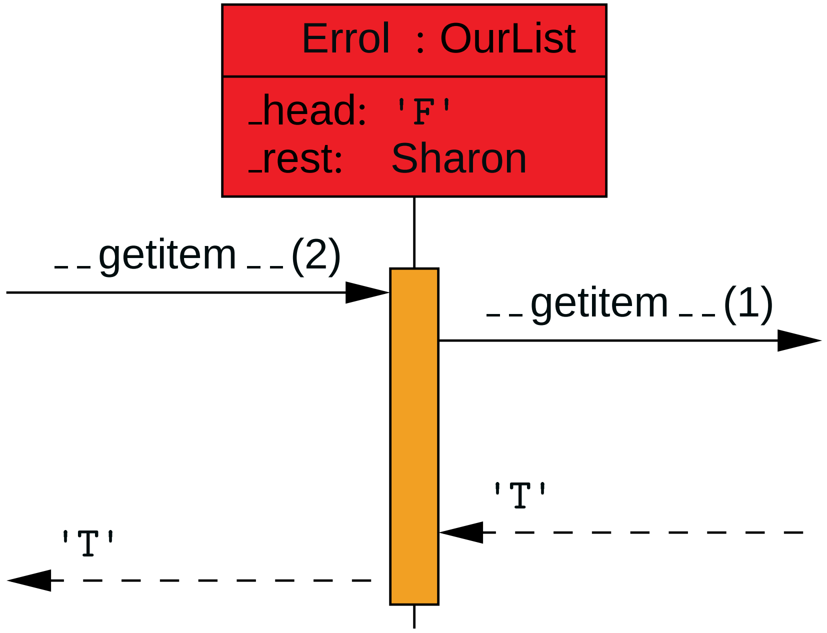
The __getitem__ method

In Python, the operator syntax data[2] is implemented with a call to data.__getitem__(2).



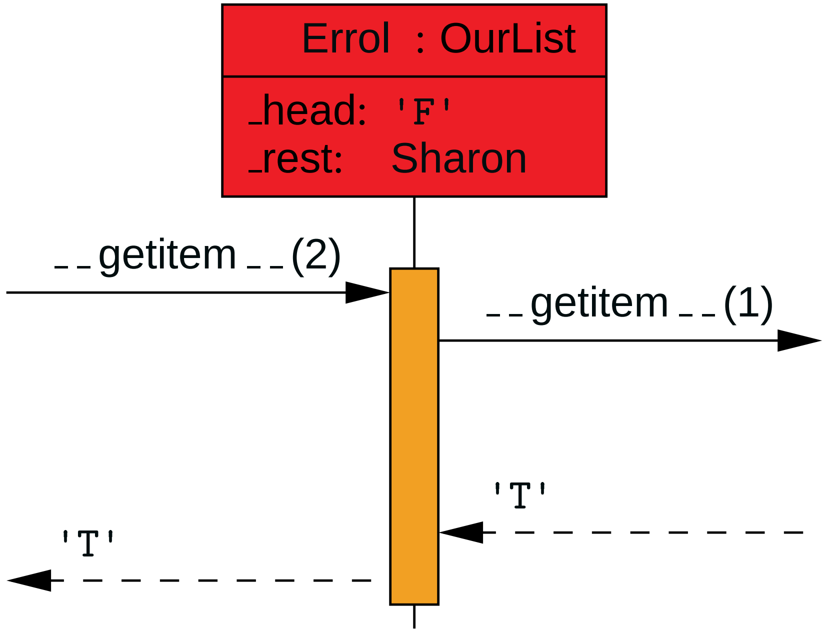
The __getitem__ method

In Python, the operator syntax data[2] is implemented with a call to data.__getitem__(2).



The __getitem__ method

In Python, the operator syntax data[2] is implemented with a call to data.__getitem__(2).



Note: the parameter value changes during the recursion; the return value does not change.

Recursive Patterns

method	base case			parameters			return value		
	empty	head	index	same	vary	none	same	vary	none
__len__	✓					✓		✓	
__contains__	✓	✓		✓			✓		
__getitem__	✓		✓		✓		✓		
__setitem__	✓		✓		✓				✓
__repr__	✓					✓		✓	
count	✓			✓				✓	
index	✓	✓		✓				✓	
append	✓			✓					✓
insert	✓		✓		✓				✓
remove	✓	✓		✓					✓

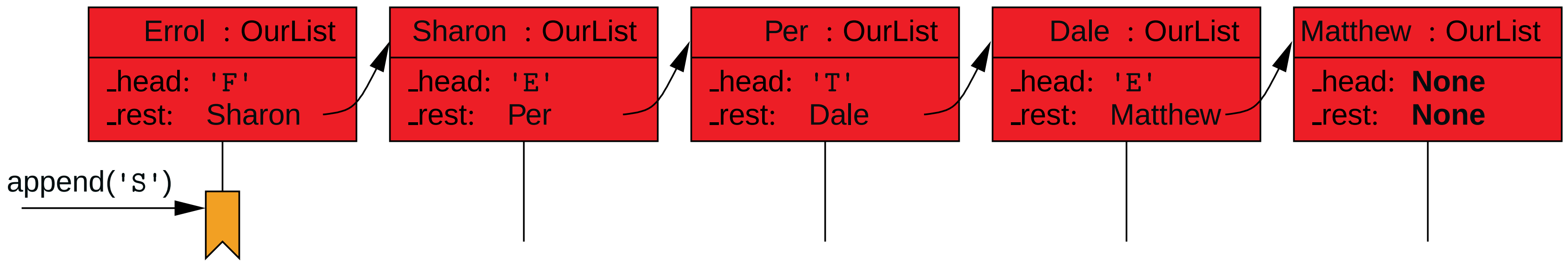
Mutators

Easiest: `__setitem__`

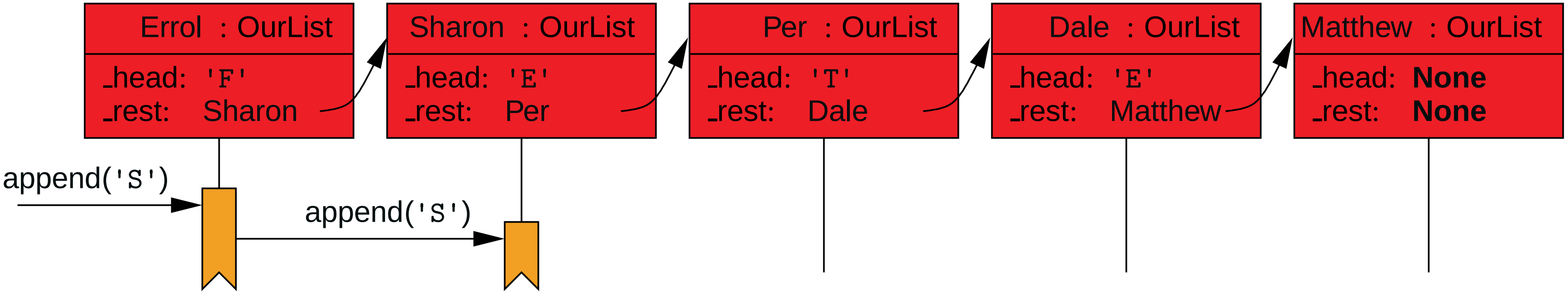
It is a one-for-one change of data, without any structural change on the list.

(very similar pattern to `__getitem__`)

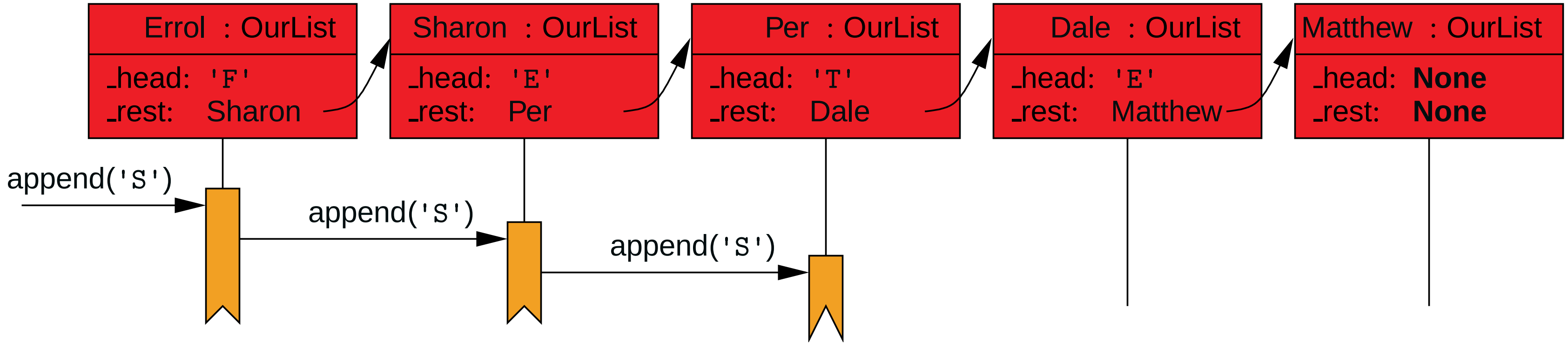
The append method



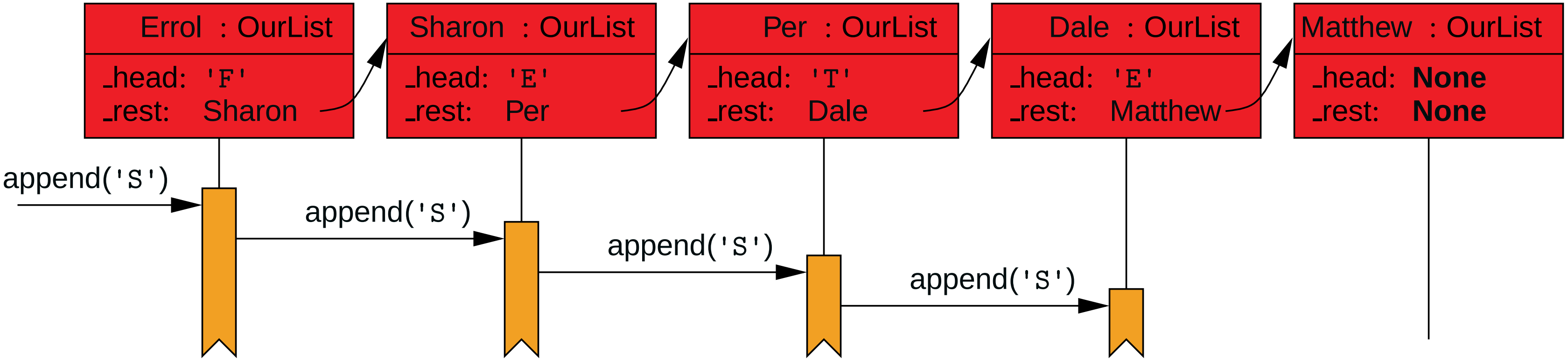
The append method



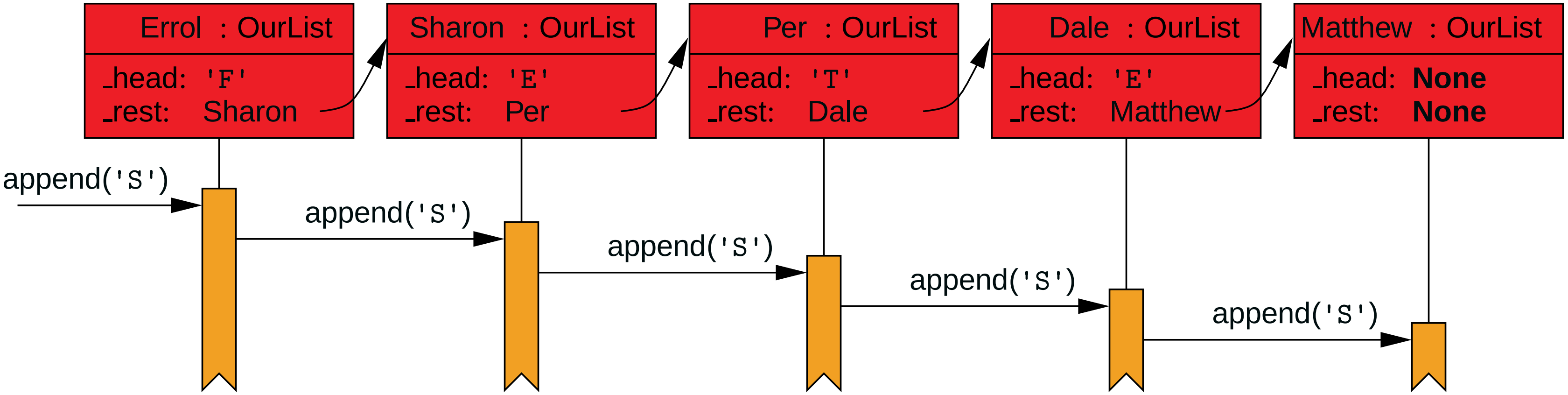
The append method



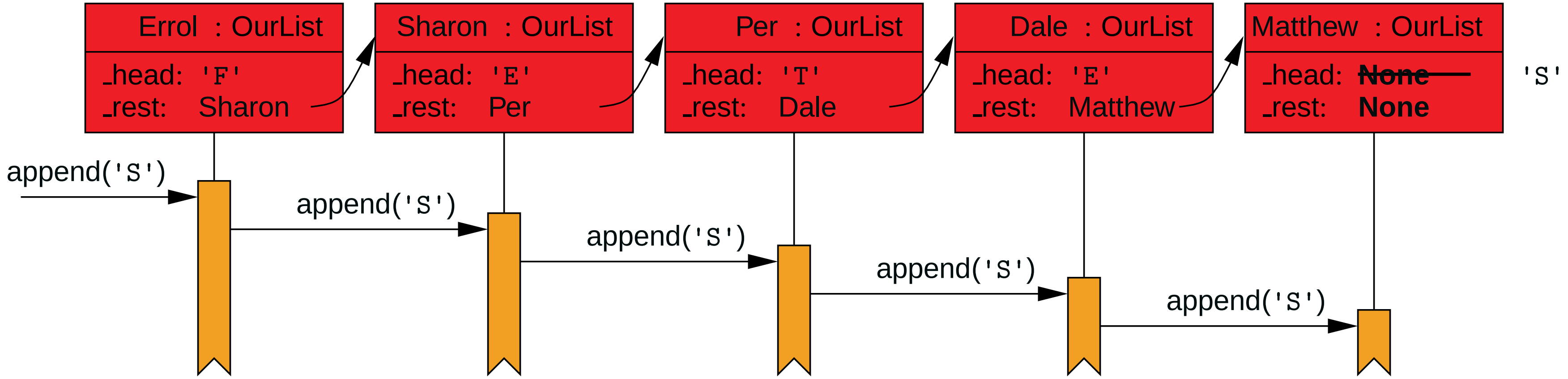
The append method



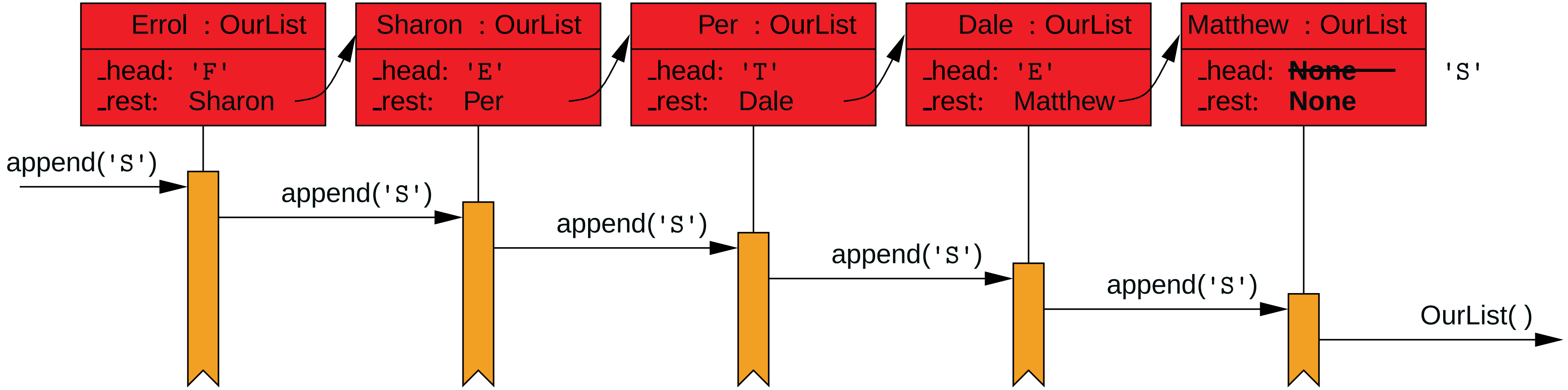
The append method



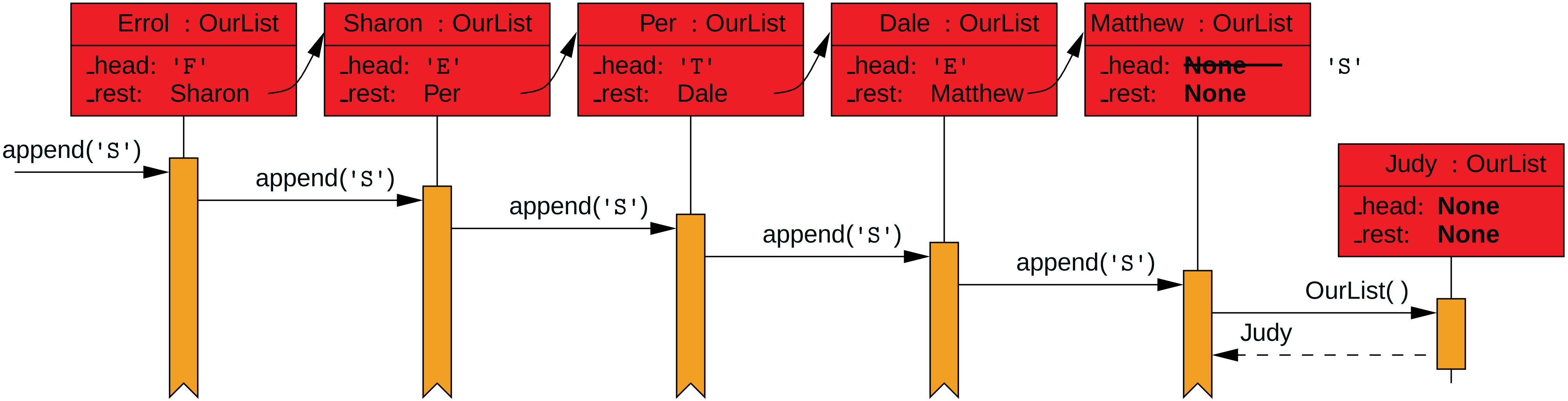
The append method



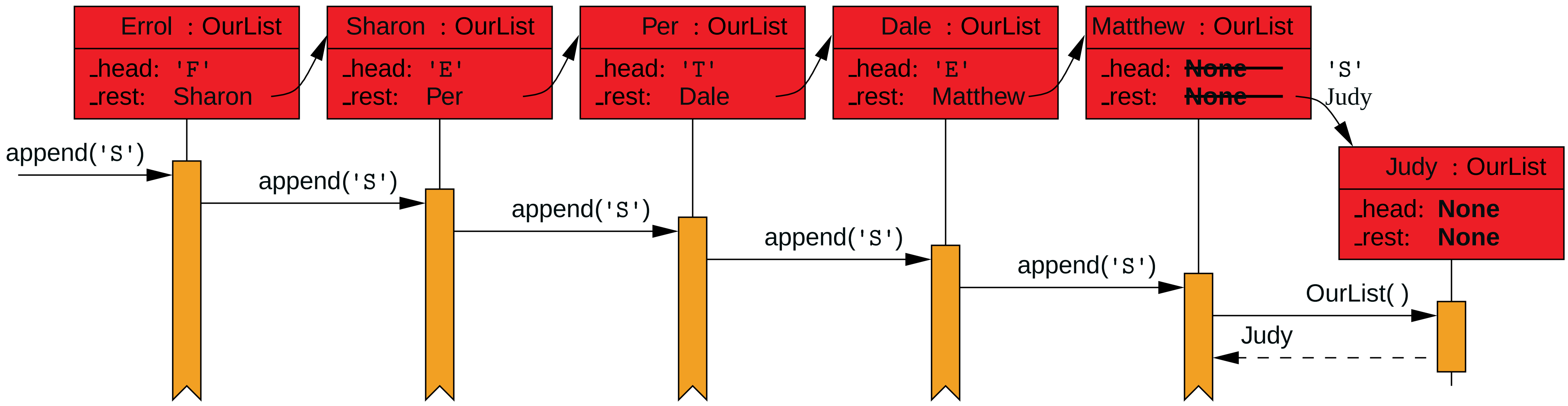
The append method



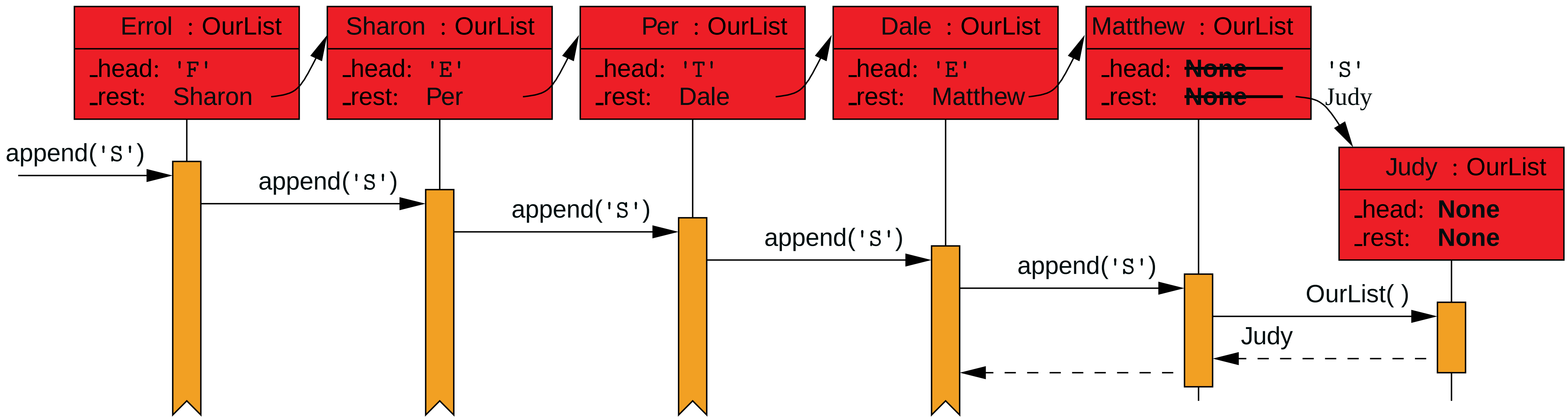
The append method



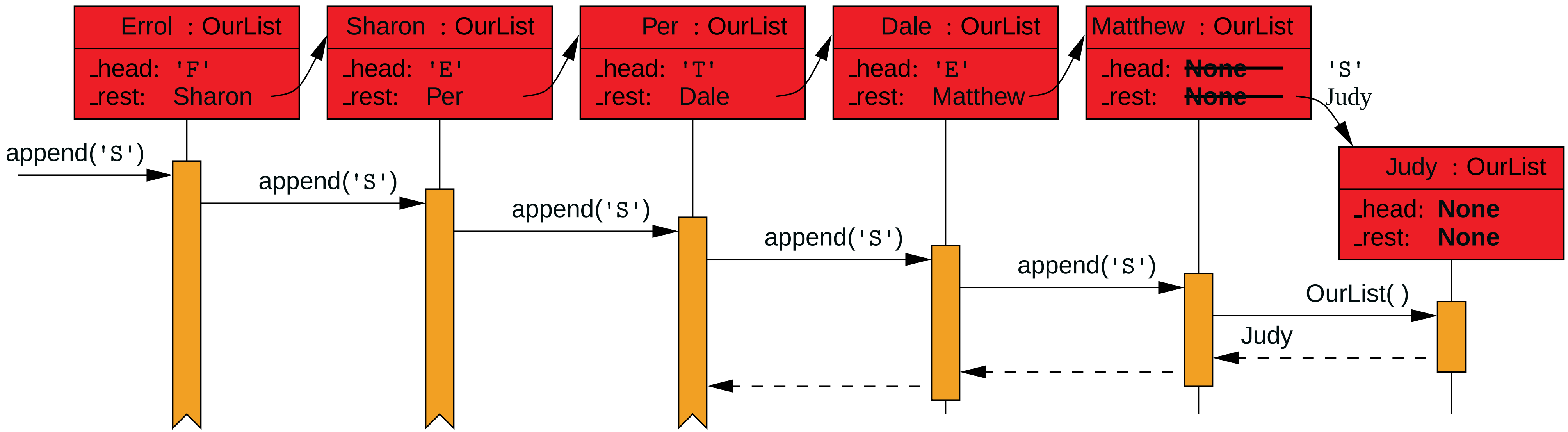
The append method



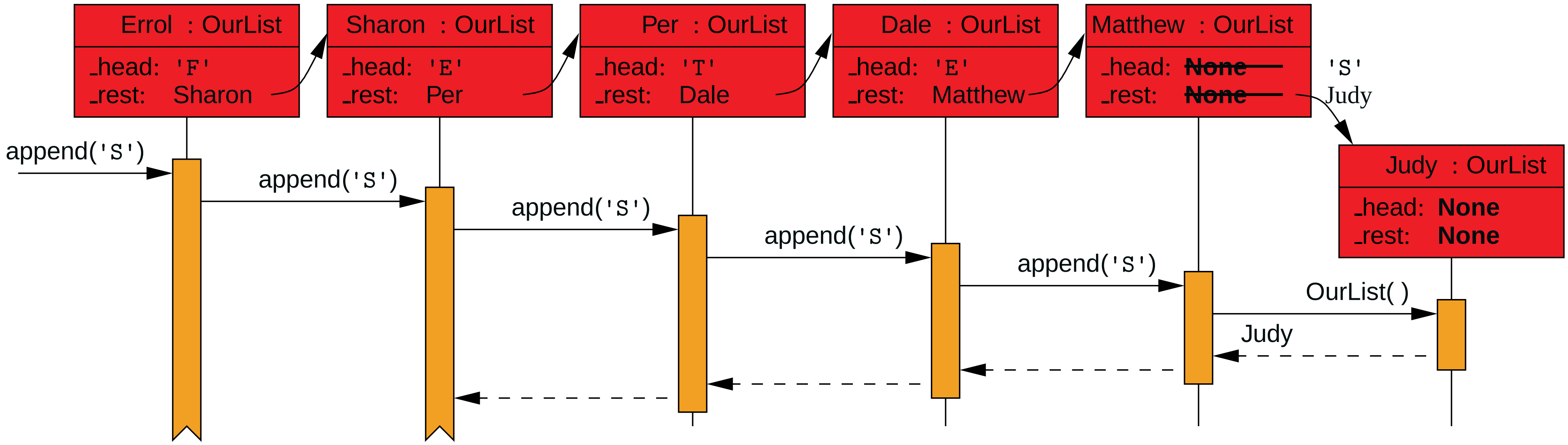
The append method



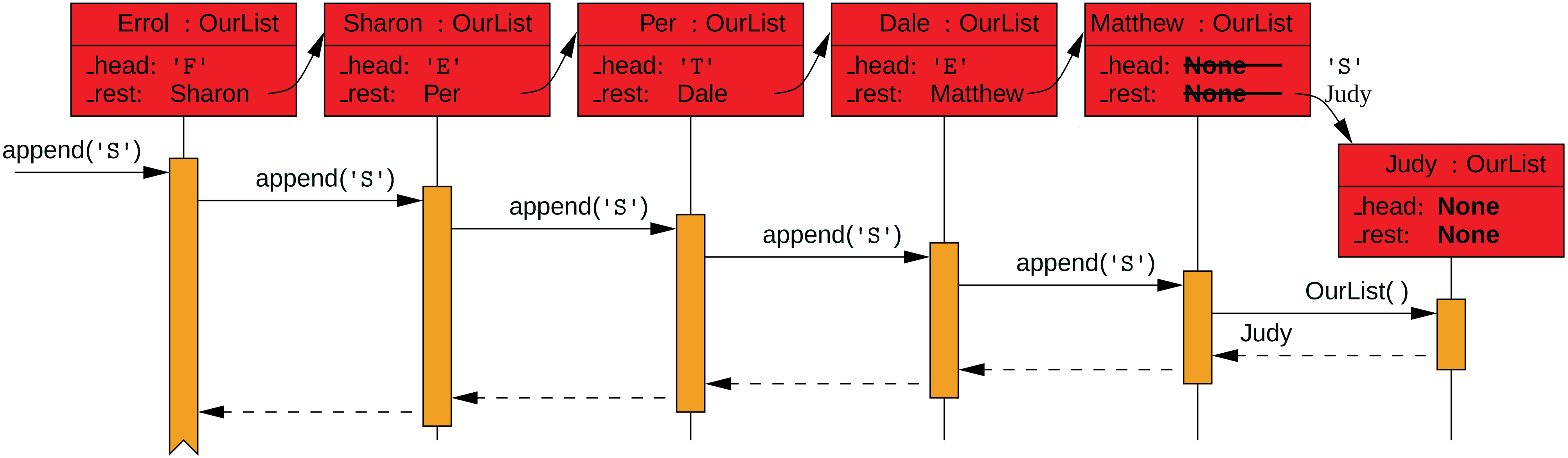
The append method



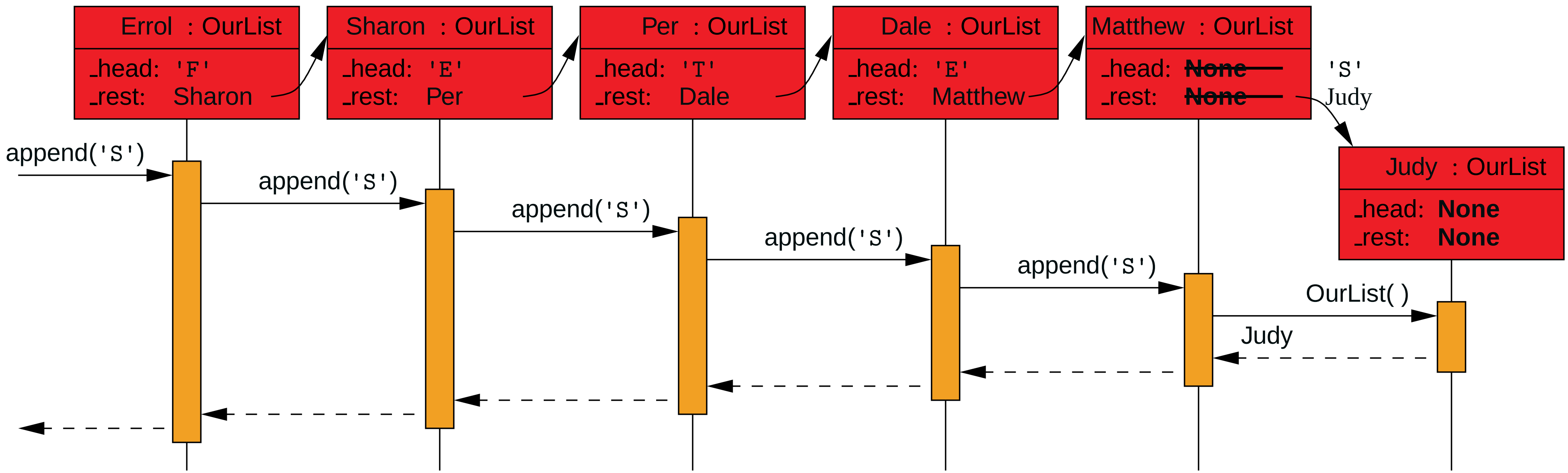
The append method



The append method



The append method



Instructor can highlight the system's memory management.

insert, remove, pop

Arbitrary insertions and deletions can be performed (more on this in the conclusion...)

Implementation

Getting Started


```
class OurList:
  def __init__(self):
    self._head = None
    self._rest = None

  def _isEmpty(self):           # a private utility
    return self._rest is None
```

The append method

Has a base case and a simple recursion

```
def append(self, value):
    if self._isEmpty( ):
        self._head = value       # we have one item
        self._rest = OurList( ) # followed by empty list
    else:
        self._rest.append(value)       # recurse
```

The count method

Has a base case and a non-trivial recursion

```
def count(self, value):
    if self._isEmpty( ):
        return 0
    else:
        answer = self._rest.count(value)
        if self._head == value:  # additional match
            answer += 1
        return answer
```

The __contains__ method

Has two distinct base cases

```
def __contains__(self, value):
    if self._isEmpty( ):
        return False
    elif self._head == value:
        return True
    else:
        return value in self._rest  # implicit recursion
```

Conclusions

- Lots of fun
- Strategic challenges
- Varying recursive patterns
- Instills a local perspective
- Coherent transition to source code
- We have really used functional recursion as well as structural recursion.

Conclusions

- Lots of fun
- Strategic challenges
- Varying recursive patterns
- Instills a local perspective
- Coherent transition to source code
- We have really used functional recursion as well as structural recursion.

Conclusions

- Lots of fun
- Strategic challenges
- Varying recursive patterns
- Instills a local perspective
- Coherent transition to source code
- We have really used functional recursion as well as structural recursion.

Conclusions

- Lots of fun
- Strategic challenges
- Varying recursive patterns
- Instills a local perspective
- Coherent transition to source code
- We have really used functional recursion as well as structural recursion.

Conclusions

- Lots of fun
- Strategic challenges
- Varying recursive patterns
- Instills a local perspective
- Coherent transition to source code
- We have really used functional recursion as well as structural recursion.

Conclusions

- Lots of fun
- Strategic challenges
- Varying recursive patterns
- Instills a local perspective
- Coherent transition to source code
- We have really used functional recursion as well as structural recursion.

Conclusions

- Lots of fun
- Strategic challenges
- Varying recursive patterns
- Instills a local perspective
- Coherent transition to source code
- We have really used functional recursion as well as structural recursion.

Advanced Lessons

- Ample opportunities for advanced lessons (time permitting)
  - Error handling
  - Default parameter values
  - More complex recursive patterns

Error handling


```
def __getitem__(self, i):
    if self._isEmpty( ):
        raise IndexError('index out of range')
    elif i == 0:
        return self._head
    else:
        return self._rest.__getitem__(i−1)
```

Error handling


```
def __getitem__(self, i):
  if self._isEmpty( ):
    raise IndexError('index out of range')
  elif i == 0:
    return self._head
  else:
    try:
      return self._rest.__getitem__(i−1)
    except IndexError:
      raise IndexError('index out of range')
```

insert and remove


```
def insert(self, index, value):
    if self._isEmpty( ):          # "append" to end
        self._head = value
        self._rest = OurList( )
    elif index > 0:               # insert recursively
        self._rest.insert(index-1, value)
    else:
        # reinsert our head as the front of the rest
        self._rest.insert(0, self._head)
        # and then store the new value here
        self._head = value
```

The insert method (alternative)


```
def insert(self, index, value):
    if self._isEmpty( ):        # "append" to end
        self._head = value
        self._rest = OurList( )
    elif index > 0:             # insert recursively
        self._rest.insert(index−1, value)
    else:                       # new item goes here!
        shift = OurList( )
        shift._head = self._head
        shift._rest = self._rest
        self._head = value
        self._rest = shift
```

The remove method


```
def remove(self, value):
    if self._isEmpty( ):
        raise ValueError('value not in list')
    elif self._head == value:
        self._head = self._rest._head     # private
        self._rest = self._rest._rest     # private
    else:
        self._rest.remove(value)
```

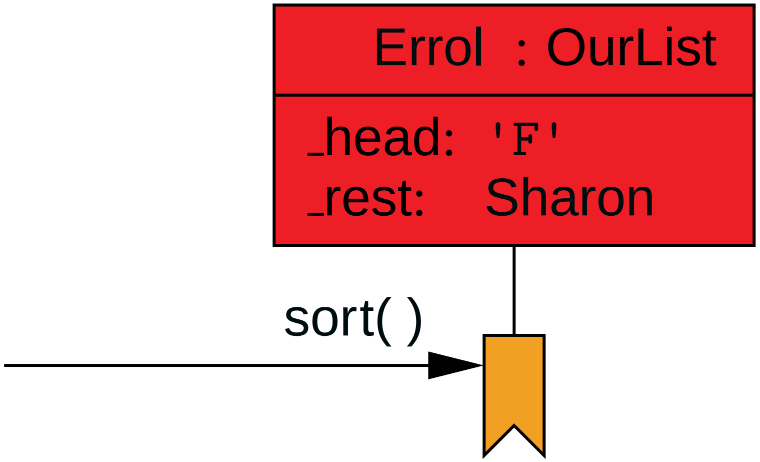
Default parameters

```
def pop(self, index=None):
  if self._isEmpty( ):
    raise IndexError('pop from empty list')
  else:
    if index is None:
      index = len(self) − 1
    if index == 0:
      answer = self._head
      self._head = self._rest._head
      self._rest = self._rest._rest
      return answer
    else:
      return self._rest.pop(index−1)
```

reverse

Make use of other existing methods together with one recursive call.

```
def reverse(self):
    if not self._isEmpty( ):
        self._rest.reverse( )
        self._rest.append(self._head)
        self.remove(self._head)
```



Errol : OurList
_head: 'F'
_rest: Sharon
sort( )