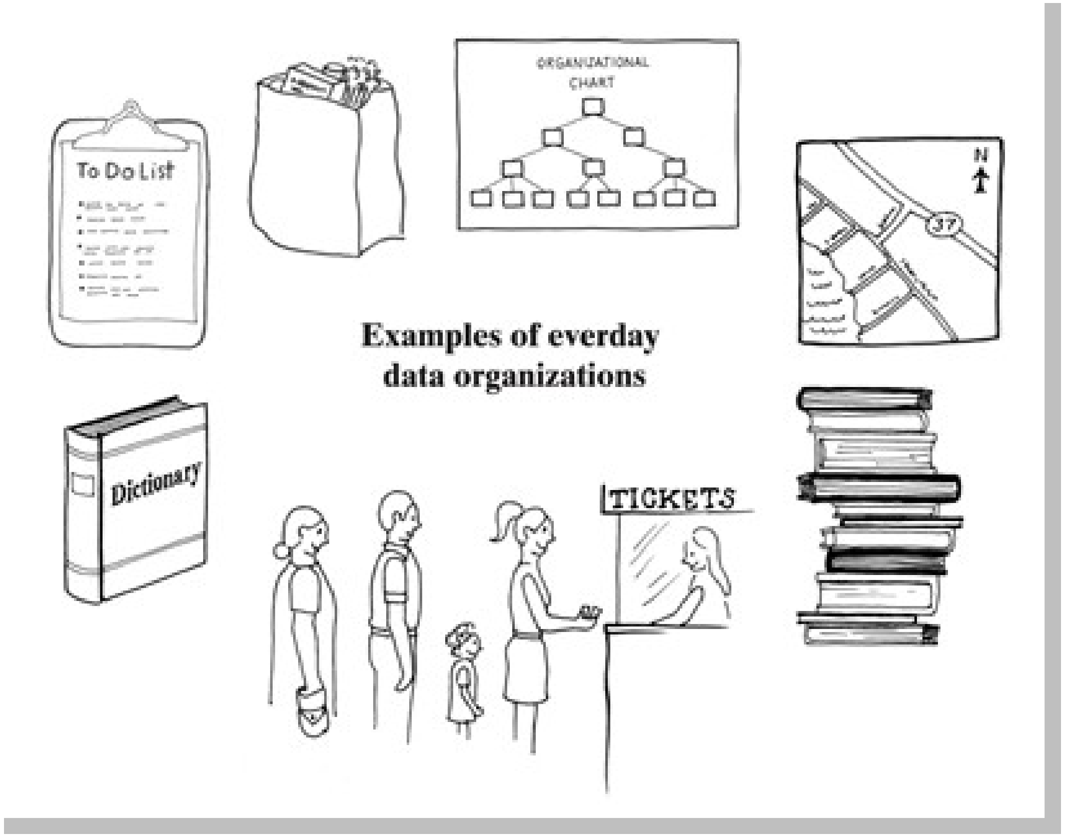
Dynamic Data Structures and Generics

Reading:
Savitch ch. 12

Objectives

- Introduce Abstract Data Types (ADTs) and review interfaces
- Introduce Java's **`ArrayList`** class
- Learn about linked lists and inner classes
- Introduce Generics

Organizing Objects



- The objects in each example are organized in a specific way.

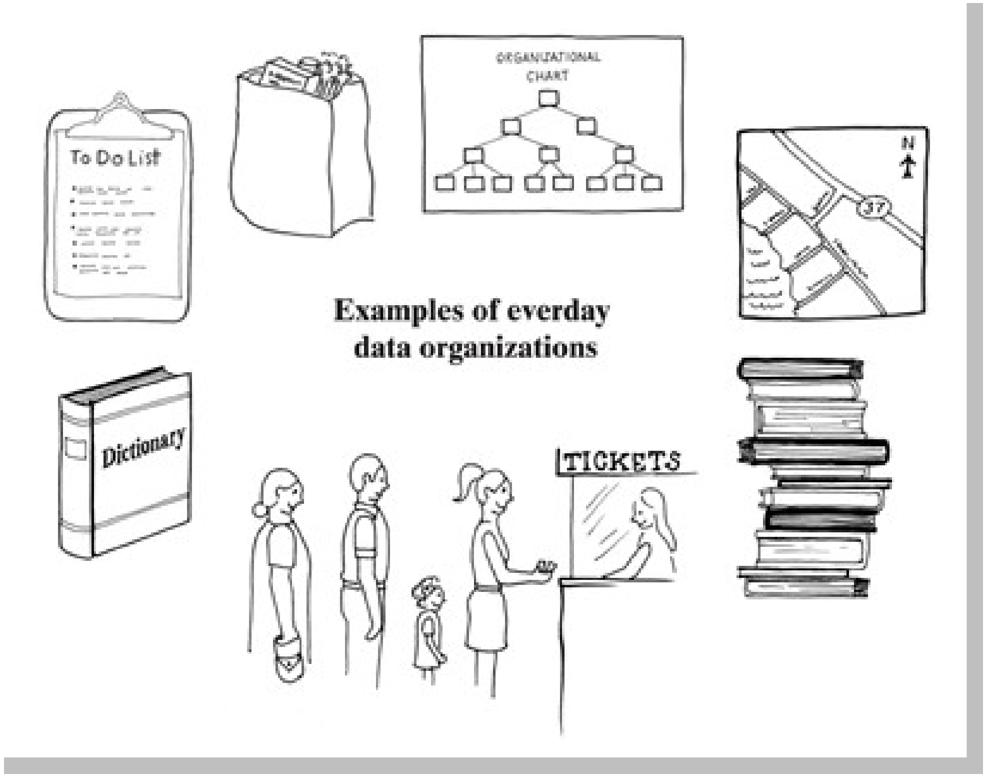
Abstract Data Type (ADT)

- Computers store/organize items similarly to the examples.
- Ways of organizing data are represented by Abstract Data Types (**ADT**s).
- An ADT specifies
  - data that is stored
  - operations that can be done on the data

An ADT is Abstract

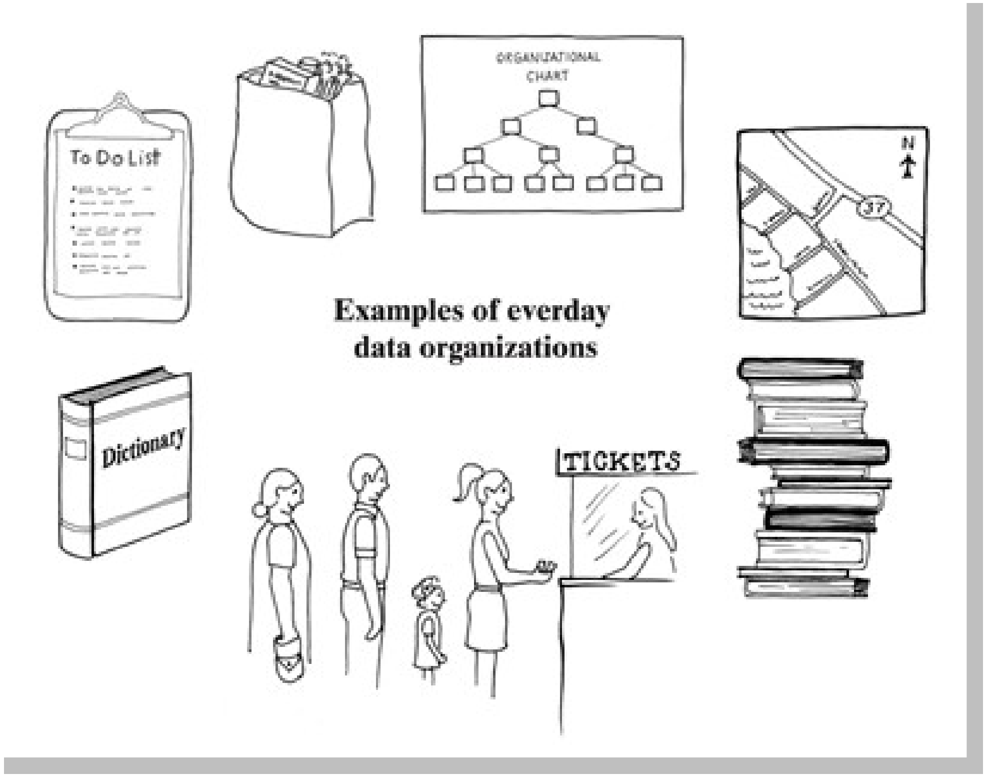
- The data type is abstract.
  - Implementation details are NOT part of an ADT.
  - An ADT does NOT specify how the data is to be represented.
- We can discuss ADTs independently of any programming language.

Types of ADTs



- Bag
  - Unordered collection, may contain duplicates
- List
  - A collection that numbers its items
- Stack
  - Orders items chronologically
  - Last In, First out

Types of ADTs



- Queue
  - Orders items chronologically
  - First in, First out
- Dictionary
  - Pairs of items – one is a key
  - Can be sorted or not
- Tree
  - Arranged in a hierarchy
- Graph
  - Generalization of a tree

ADT Terminology

- **Data structure**: implementation of an ADT within a programming language.
- **Collection**: an ADT that contains a group of objects
- **Container**: a class that implements the collection
- The terms Collection and Container can be used interchangeably

Interfaces

- In Java, an ADT is represented as an *interface*, e.g., **`List<T>`**
- In a Java interface, the operations are expressed as *abstract methods*.
- An *abstract method* is a method that does not have an implementation.
- In an interface, all of the methods are abstract.

Interface - Example

- The following interface, called **`ListADT`** has 3 abstract methods, **`add`**, **`remove`**, and **`get`**:

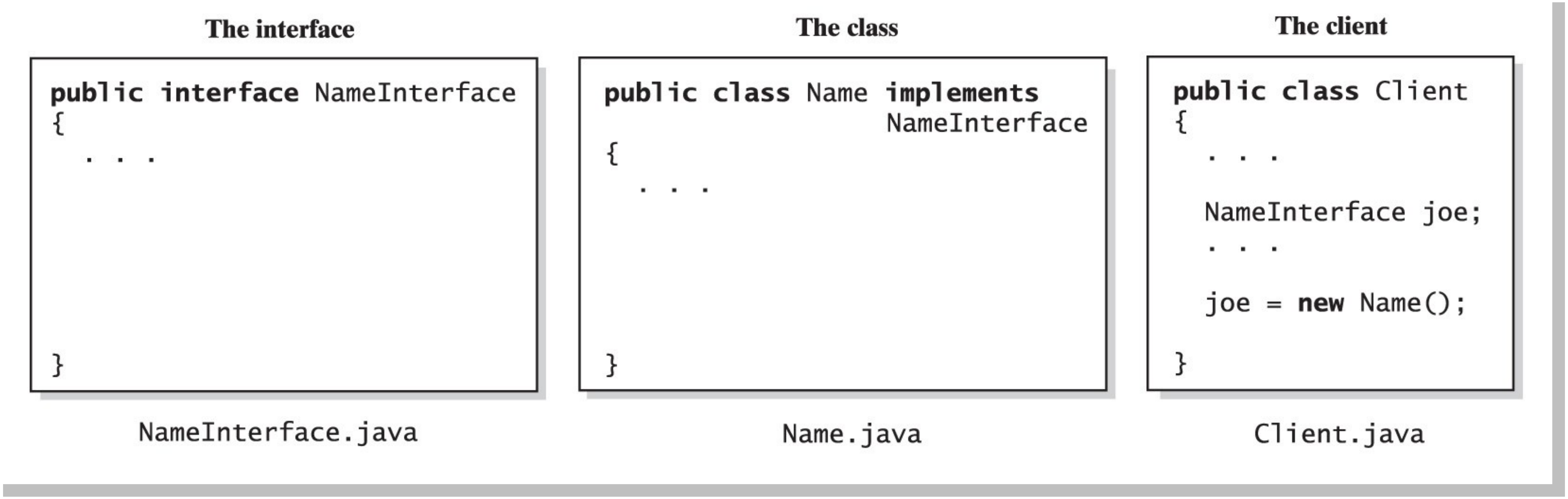
```java
// file ListADT.java
public interface ListADT {
    public void add(String element);
    public void remove(String element);
    public String get(int index);
}
```

Implementing an Interface ☆

- A class implements an interface by providing method implementations for each of the abstract methods.
- A class that implements an interface uses the reserved word **`implements`** followed by the interface name. i.e. `implements` + interface name
- A class can implement more than 1 interface:

```
public class MyClass
   implements interface1, interface2, … { …
```

Implementing an Interface



The interface

```
public interface NameInterface
{
   . . .






}
```

NameInterface.java

The class

```
public class Name implements
                      NameInterface
{
   . . .






}
```

Name.java

The client

```
public class Client
{
   . . .

   NameInterface joe;
   . . .

   joe = new Name();

}
```

Client.java

- The object **joe** has the types **NameInterface** and **Name**

`Comparable<T>` Interface ☆

- The **`Comparable<T>`** interface is defined in the java standard class library.
- It contains one method, **`compareTo`**, which takes an object as parameter and returns an integer
    - e.g.: **`int result = obj1.compareTo(obj2);`**
- The intention of this interface is to provide a common way to compare one object to another
- The integer that is returned should be negative if **`obj1`** is less than **`obj2`**, 0 if they are equal, and positive if **`obj1`** is greater than **`obj2`**
- The **`String`** class implements this interface

List<T> Interface

- The **`List<T>`** interface is part of the **`java.util`** package
- The intention of this interface is to provide a common way to store and maintain an ordered collection (sequence/list) of data
- It contains several abstract methods, e.g.: **`add`**, **`contains`**, **`get`**, **`indexOf`**, **`remove`**, **`set`**, **`size`**, etc.
- The **`ArrayList`** class implements this interface

`ArrayList` Introduction

- A *data structure* is used to organize data in a specific way
- An array is a *static* data structure
- *Dynamic* data structures can grow and shrink while a program is running
- **`ArrayLists`** are dynamic
- **`ArrayLists`** are similar to arrays, but are more flexible

ArrayLists

- We can think of **`ArrayLists`** as arrays that grow and shrink while a program is running.
- At the time an array is created, its length is fixed.
    - The length sometimes turns out to be too small and we need to resize the array.
    - If the length is too large, we waste space

ArrayLists

- **ArrayLists** perform the resizing operation that we implemented ourselves up to now
- **ArrayLists** serve the same purposes as arrays, but can change in length while a program runs.
- The added flexibility comes at a price:
  - **ArrayLists** are less efficient than arrays
  - The base type of an **ArrayList** can't be a primitive type (use wrapper classes)

Using ArrayLists

- The definition of class **ArrayList** must be imported:

```
import java.util.*;
```

- To create and name an **ArrayList**:

```
ArrayList<String> list =
                     new ArrayList<String>(50);
```

- The **ArrayList list** stores objects of type **String** and has an initial capacity of 50.
  - The capacity will grow if more than 50 items are added.

Creating an ArrayList

- Syntax:

```
ArrayList<BaseType> name =
     new ArrayList<BaseType>();
```

or:

```
ArrayList<BaseType> name =
     new ArrayList<BaseType>(initialCapacity);
```

  - ***BaseType*** can be any class type.
    - Use the wrapper classes (**Integer**, **Double**,...) to store primitive types.

Adding and Getting Elements

Create a list of **Word** objects and add **Words**:

```
ArrayList<Word> list = new ArrayList<Word>();
list.add(new Word(“the”));
list.add(new Word(“dog”));
list.add(new Word(“bites”));
```

Get the second element:

```
Word aWord = list.get(1); // dog
aWord = list.get(3); //ERROR: index >= size()
```

Adding and Getting Elements

Inserting into the middle:

```
list.add(1, new Word(“vicious”));
```

- The **`Word`** “vicious” is now at index 1
- The other **`Words`** get moved down

before:	after:
`0 the`	**`0 the`**
`1 dog`	**`1 vicious`**
`2 bites`	**`2 dog`**
	`3 bites`

Adding and Getting Elements

Inserting into the middle:

```
0 the
1 vicious
2 dog
3 bites
```

- `list.add(5, new Word(“children”));`
  - error – index must be less than size
- `list.add(4, new Word(“children”));`
  - ok - adds to the end of the list

Removing an Element

```
list.remove(1);
```

before:	after:
0 the	0 the
1 vicious	1 dog
2 dog	2 bites
3 bites	3 children
4 children	

0 <= index < size()

Removing an Element

Remove the first occurrence of a **Word** with the form “children”:

```
list.remove(new Word(“children”));
```

before:	after:
0 the	**0 the**
1 dog	**1 dog**
2 bites	**2 bites**
3 children	

Word must have a well-defined **equals** method!

Finding an Element

Find out if there is an occurrence of a **Word** with the form “dog”, or “cat”:

```
    0 the
    1 dog
    2 bites
boolean dogFound = list.contains(
                      new Word(“dog”)); // true
boolean catFound = list.contains(
                      new Word(“cat”)); // false
```

Finding an Element

Get the index of the first occurrence of a **`Word`** with the form “dog”, or “cat”:

```
    0 the
    1 dog
    2 bites
int dogIndex = list.indexOf(new Word(“dog”));
// dogIndex is 1
int catIndex = list.indexOf(new Word(“cat”));
// catIndex is -1 (not in the list)
```

Setting an Element

Set the element at index **`dogIndex`** to a **`Word`** with the form “cat”:

```
int dogIndex = list.indexOf(new Word(“dog”));
if (dogIndex >= 0) {
    list.set(dogIndex, new Word(“cat”));
}
```

```
before:          after:

0 the            0 the
1 dog            1 cat
2 bites          2 bites
```

`ArrayList` Exercises

1. Write a static method that takes a **`String`** array and returns an **`ArrayList`** of type **`String`** with the same elements.

2. Write a static method that takes an **`ArrayList`** of type **`String`** AND a **`String`**, and deletes all instances of the string in the **`ArrayList`**.

Exercise 1 – sample solution

```
public static ArrayList<String>
                   arrayToArrayList(String[] s) {

    ArrayList<String> result =
                 new ArrayList<String>(s.length);

    for (int i=0; i < s.length; i++) {
        result.add(s[i]);
    }

    return result;

}
```

Exercise 2 – sample solution

```
public static void
      removeFromArrayList(ArrayList<String> list,
                          String s) {

    int foundAtIndex = list.indexOf(s);

    while (foundAtIndex >= 0) {
        list.remove(foundAtIndex);
        foundAtIndex = list.indexOf(s);
    }

}
```

Parameterized Classes
Generics

- Java's **`ArrayList`** class allows us to specify the type of the objects stored in the list; it is a *parameterized class*:

  `ArrayList<`*`BaseType`*`>`

- Its parameter, the **`BaseType`**, can be replaced by any class type
- These definitions are called *generic definitions*, or simply *generics*

Collection Classes

- A new group of classes implement the `Collection` interface.
- These classes are know as *collection* classes
- `ArrayList` is a collection class
- There is a special for-loop syntax that can be used with collection classes

Collection Classes – "for-each" loop

- Syntax:

```
for (BaseType variable : collectionObject)
{

        // statements

}
```

Collection Classes – “for-each” loop

- Example:

```
ArrayList<String> list =
                    new ArrayList<String>();
list.add(“hello”);
list.add(“world”);

//we say: “for each String element in list”
for (String element : list) {
    System.out.println(element);
}
```

Linked Data Structures: Outline

- Linked lists
- Inner classes
- Node inner classes
- Iterators

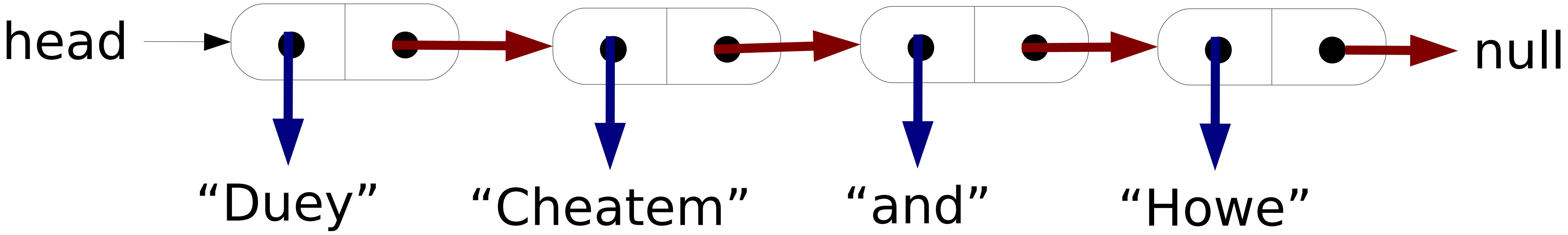
Linked Data Structures

- A linked data structure is a group of objects (called *nodes*) that are connected by references (called *links*)
- Java has a predefined **`LinkedList`** class, which is part of the **`java.util`** package
- In order to learn how linked data structures work, we will construct our own linked list class.

String Linked List

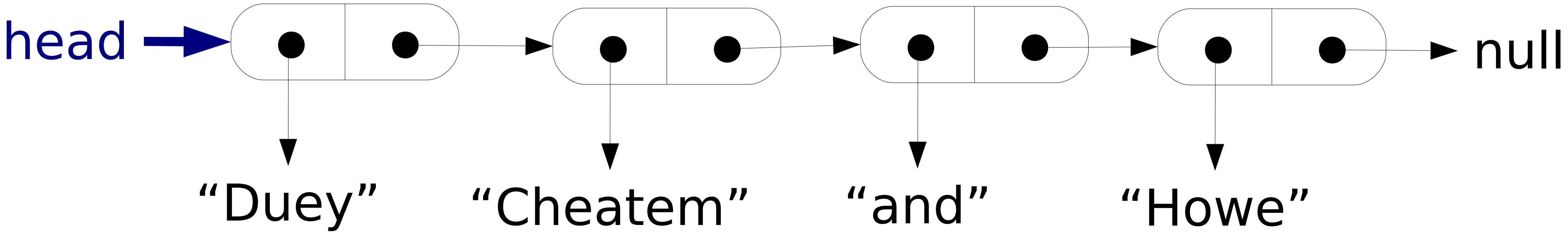
- We will call our linked list class **`StringLinkedList`**.
- **`StringLinkedList`** *has-a* reference to the first node in the list – also called the **`head`** of the list.
- We will define a separate class called **`ListNode`** to represent a node.
- A **`ListNode`** *has* data and a link to the next node.

Linked Lists



- The two references in each node are its instance variables.
    - One refers to the node's data (“Duey”, “Cheatem”, ...)
    - The other refers to the next node in the list. It links one node to the next.

Linked Lists



- The reference called **head** is an *instance variable* of the **StringLinkedList** class. It references an object of the node type.
- **head** is a reference to the first node in the list, but is not itself one of the nodes.

The `ListNode` Class

Savitch p 837

- Two instance variables to reference the node's **`data`** and **`link`**
- Simple constructors
- Getters and setters for the instance variables

The First and Last Nodes

- There has to be a way of determining which node is the last node in the list.
- The node that has a **`null link`** instance variable is the last node.
- The value of the **`link`** instance variable is tested for **`null`** with **`==`**

  ```
  if (link == null) //this is the last node
  ```
- **`head`** is the reference to the first node.
- **`if (head == null)`**, the list is empty.

StringLinkedList - methods
Savitch p 839

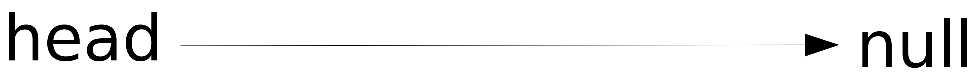
- **addANodeToStart**
- **length**
- **deleteHeadNode**
- **showList**
- **onList**
- **find (private)**

StringLinkedList - addANodeToStart

add the first node

```
public void addANodeToStart(String addData)  {
    head = new ListNode(addData, head);
}
```

Before:

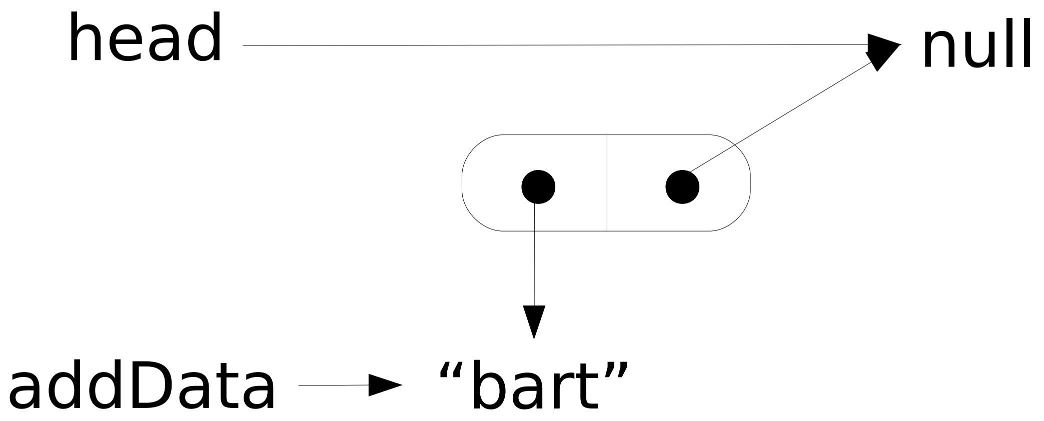


StringLinkedList - addANodeToStart

add the first node

```
public void addANodeToStart(String addData)  {
    head = new ListNode(addData, head);
}
```

Create new node:

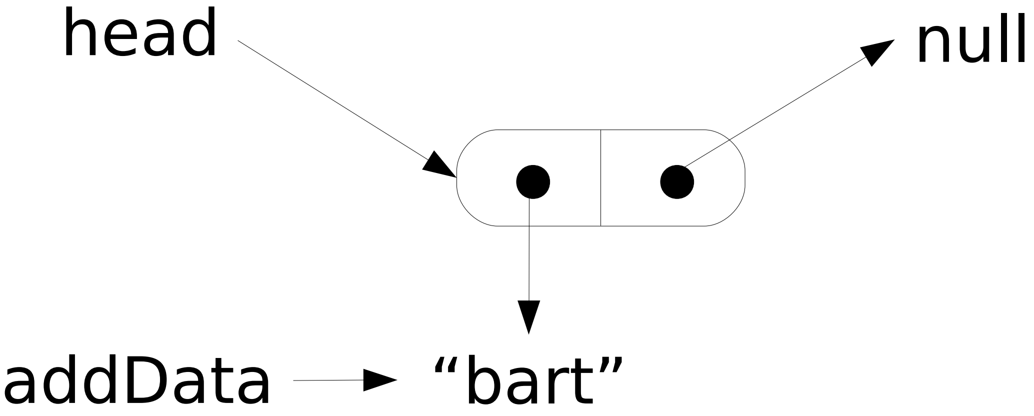


StringLinkedList - addANodeToStart

add the first node

```
public void addANodeToStart(String addData)  {
    head = new ListNode(addData, head);
}
```

After:

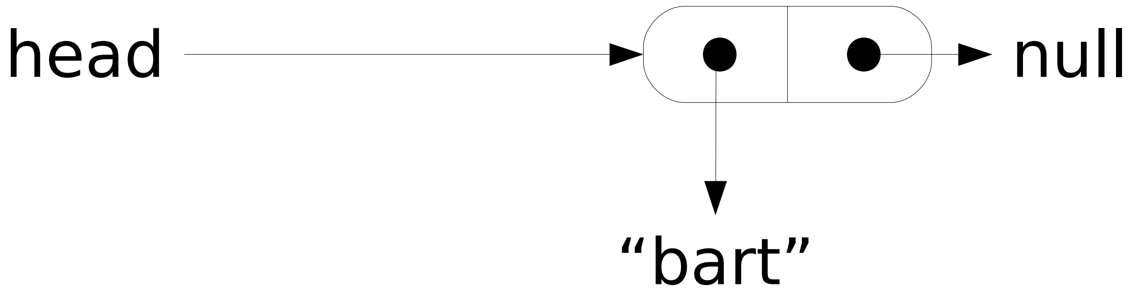


StringLinkedList - addANodeToStart

add a second node

```
public void addANodeToStart(String addData)  {
    head = new ListNode(addData, head);
}
```

Before:

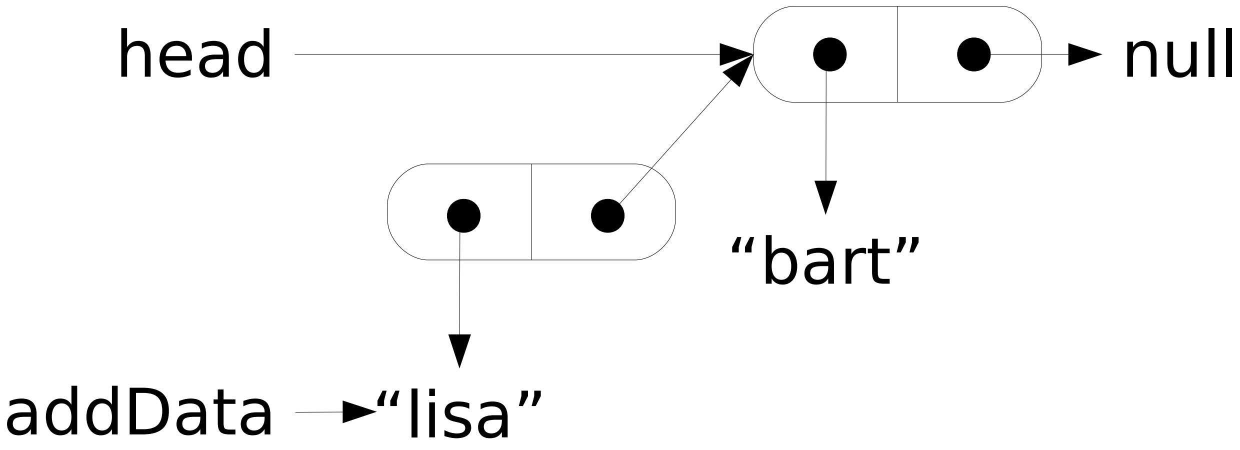


StringLinkedList - addANodeToStart

add a second node

```
public void addANodeToStart(String addData)  {
    head = new ListNode(addData, head);
}
```

Create new node:

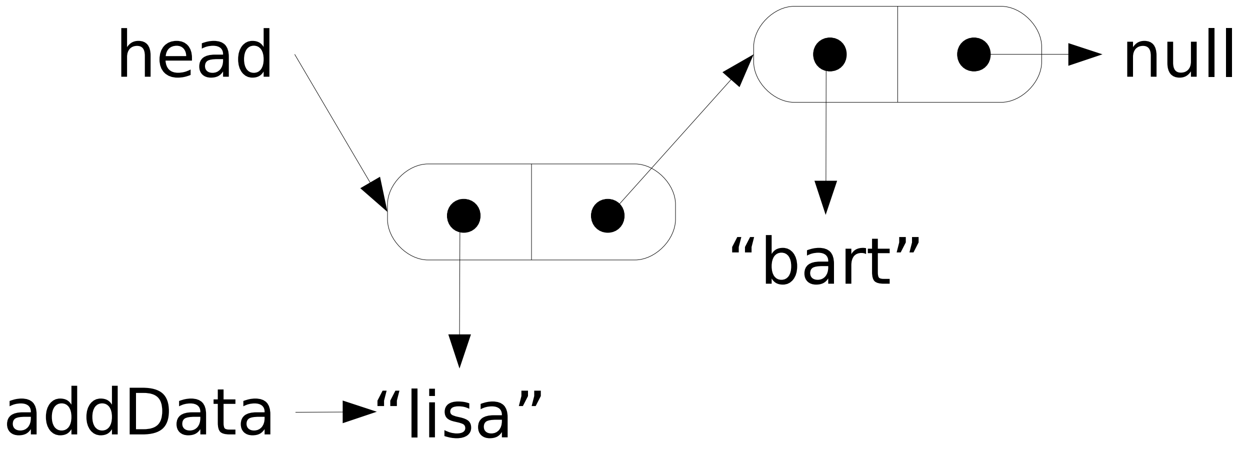


StringLinkedList - addANodeToStart

add a second node

```
public void addANodeToStart(String addData)  {
    head = new ListNode(addData, head);
}
```

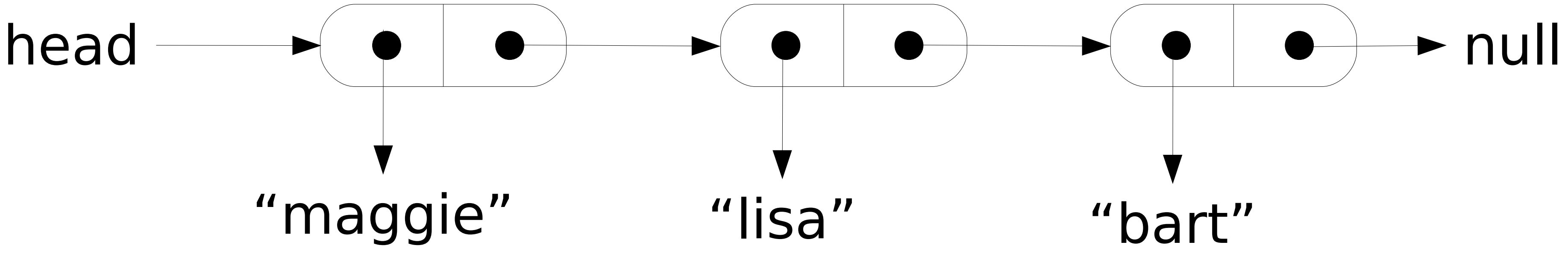
After:



StringLinkedList - length

```
public int  length()  {
   int count = 0;
   ListNode position = head;
   while (position != null) {
      count++;
      position = position.getLink();
   }
   return count;
}
```

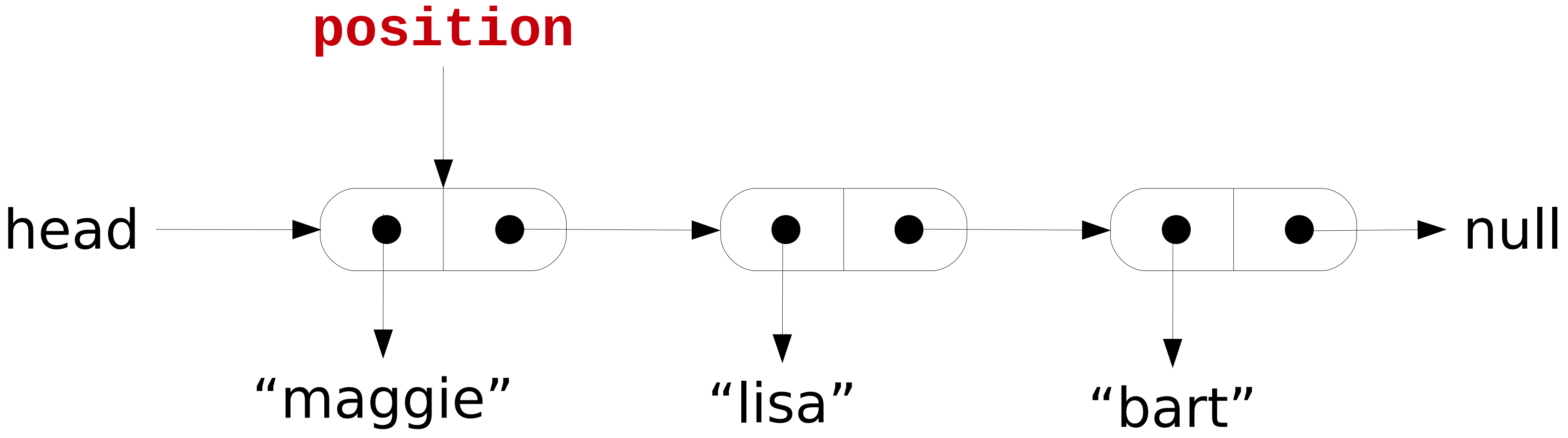
count: 0



StringLinkedList - length

```
public int  length()  {
   int count = 0;
   ListNode position = head;
   while (position != null) {
      count++;
      position = position.getLink();
   }
   return count;
}
```

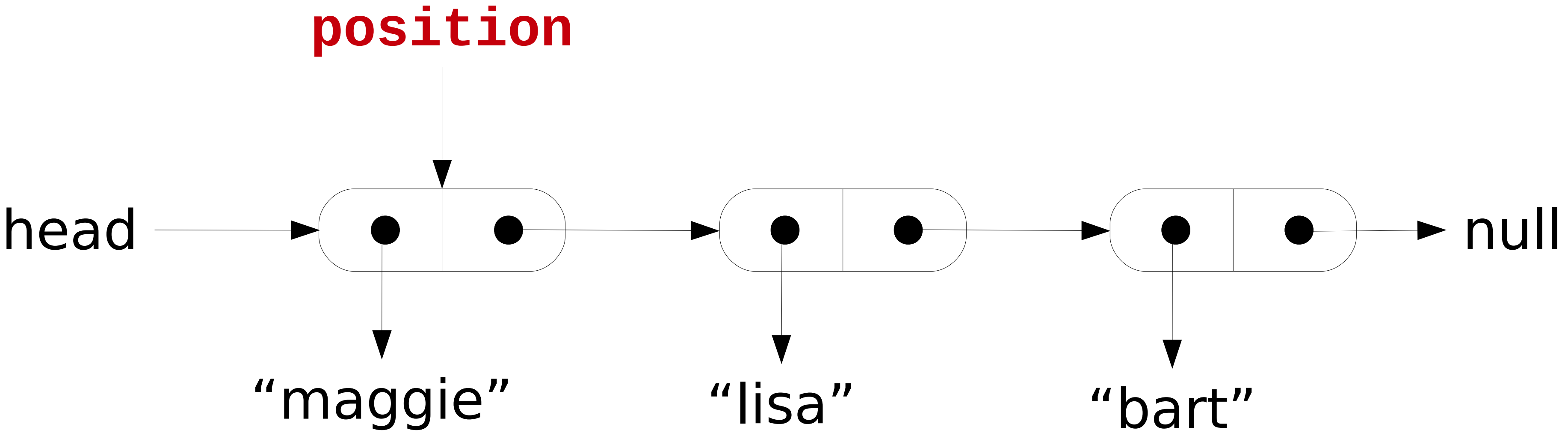
count: 0



StringLinkedList - length

```
public int  length()  {
   int count = 0;
   ListNode position = head;
   while (position != null) {
      count++;
      position = position.getLink();
   }
   return count;
}
```

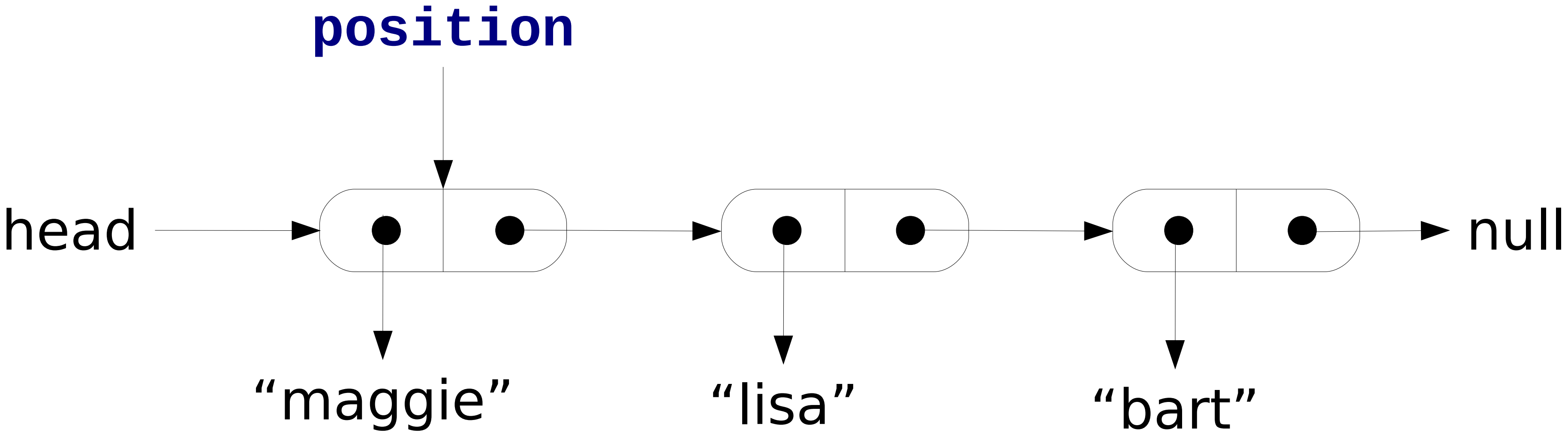
count: 0



StringLinkedList - length

```
public int  length()  {
   int count = 0;
   ListNode position = head;
   while (position != null) {
      count++;
      position = position.getLink();
   }
   return count;
}
```

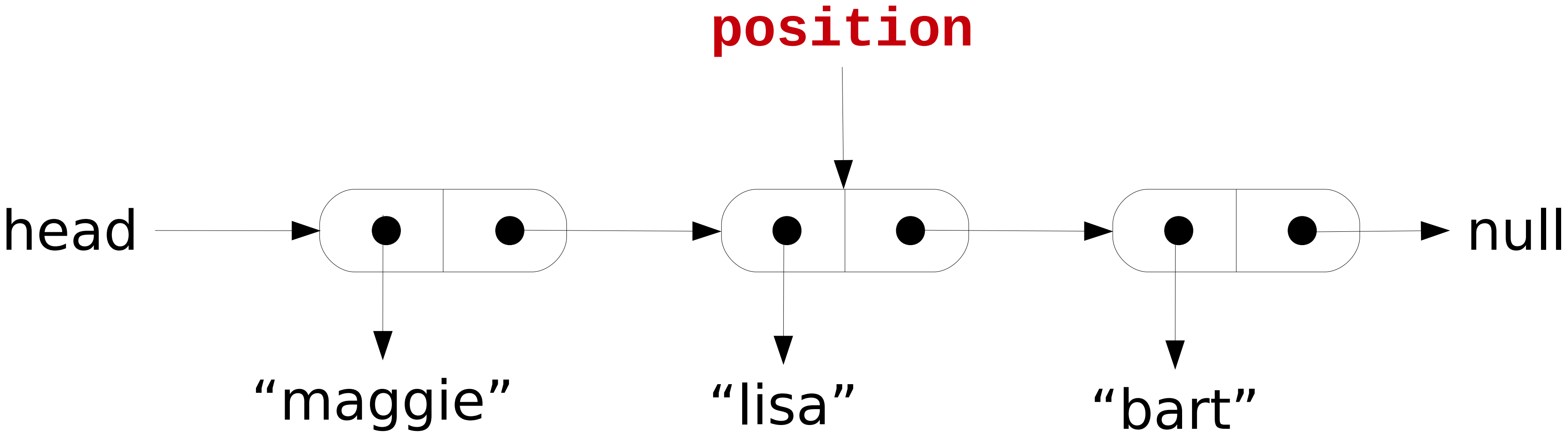
count: 1



StringLinkedList - length

```
public int  length()  {
   int count = 0;
   ListNode position = head;
   while (position != null) {
      count++;
      position = position.getLink();
   }
   return count;
}
```

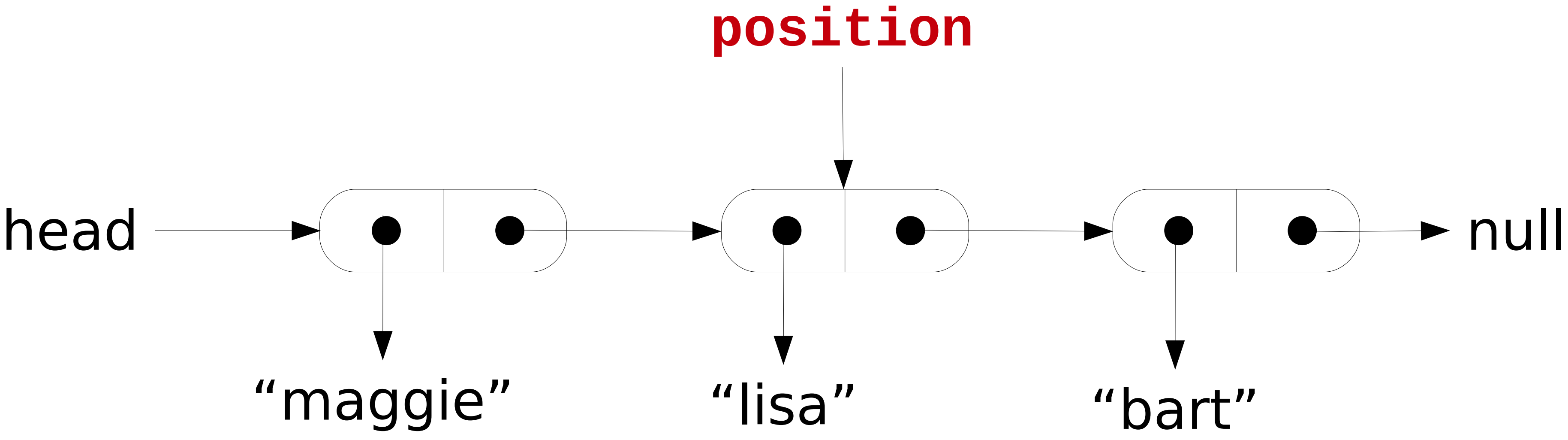
count: 1



StringLinkedList - length

```
public int  length()  {
   int count = 0;
   ListNode position = head;
   while (position != null) {
      count++;
      position = position.getLink();
   }
   return count;
}
```

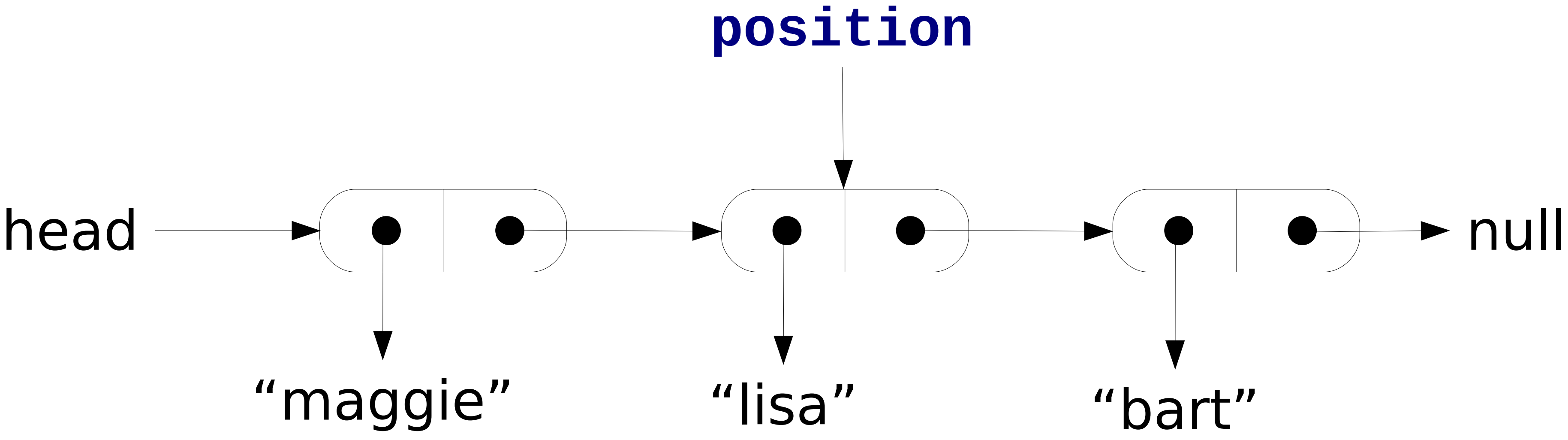
count: 1



StringLinkedList - length

```
public int  length()  {
   int count = 0;
   ListNode position = head;
   while (position != null) {
      count++;
      position = position.getLink();
   }
   return count;
}
```

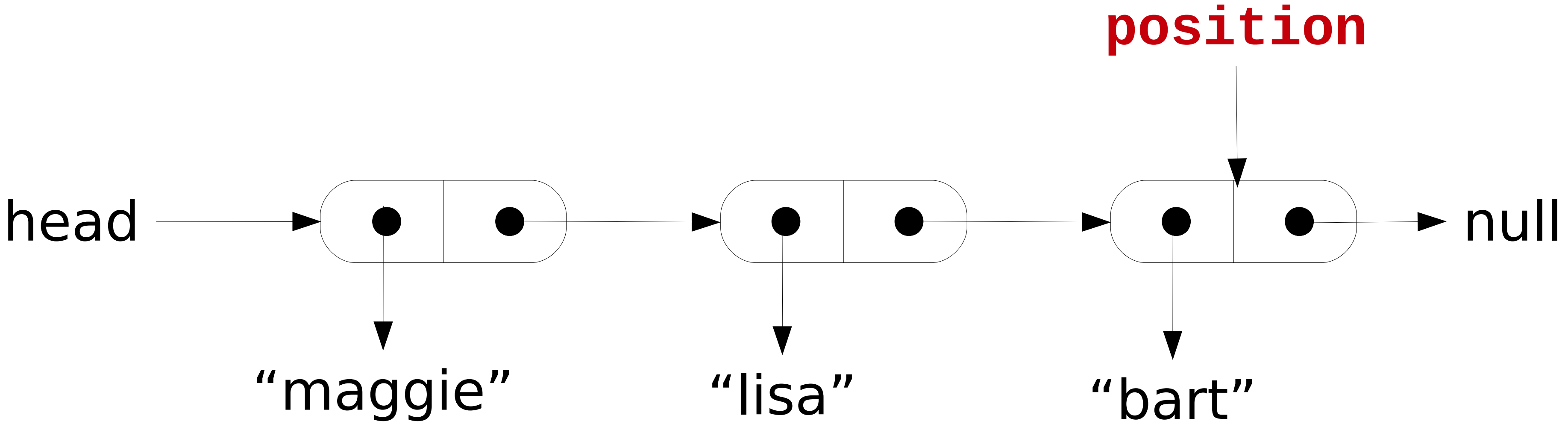
count: 2



StringLinkedList - length

```
public int  length()  {
   int count = 0;
   ListNode position = head;
   while (position != null) {
      count++;
      position = position.getLink();
   }
   return count;
}
```

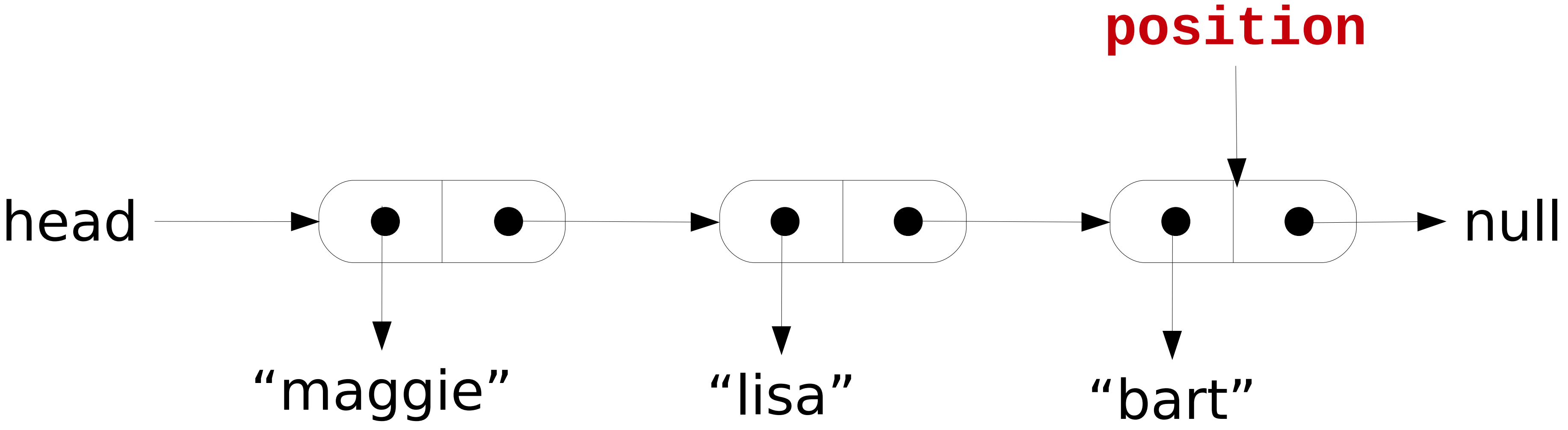
count: 2



StringLinkedList - length

```
public int  length()  {
   int count = 0;
   ListNode position = head;
   while (position != null) {
      count++;
      position = position.getLink();
   }
   return count;
}
```

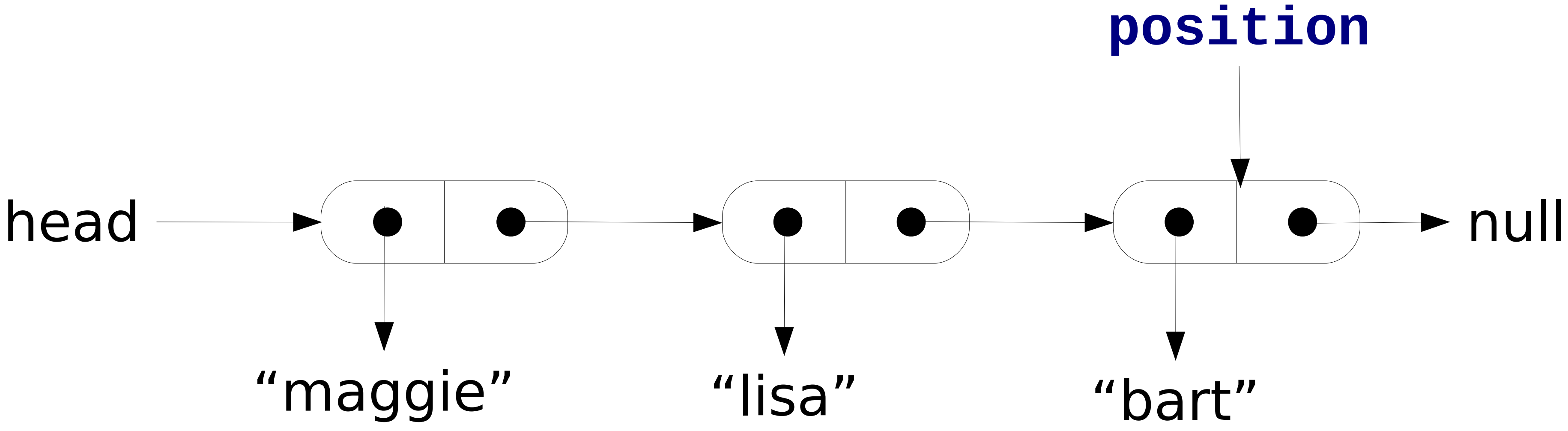
count: 2



StringLinkedList - length

```
public int  length()  {
   int count = 0;
   ListNode position = head;
   while (position != null) {
      count++;
      position = position.getLink();
   }
   return count;
}
```

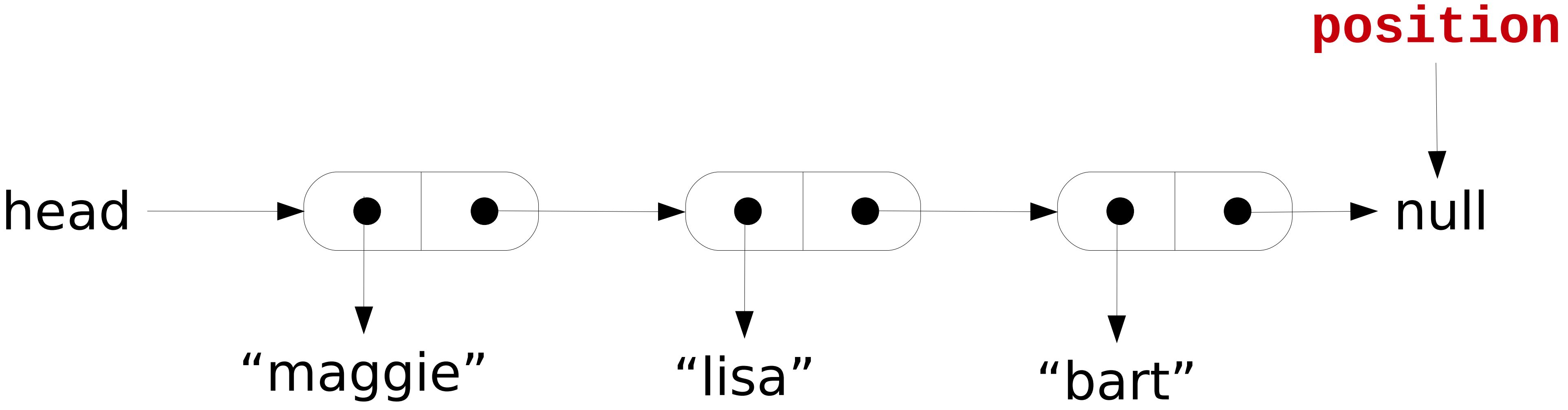
count: 3



StringLinkedList - length

```
public int  length()  {
   int count = 0;
   ListNode position = head;
   while (position != null) {
      count++;
      position = position.getLink();
   }
   return count;
}
```

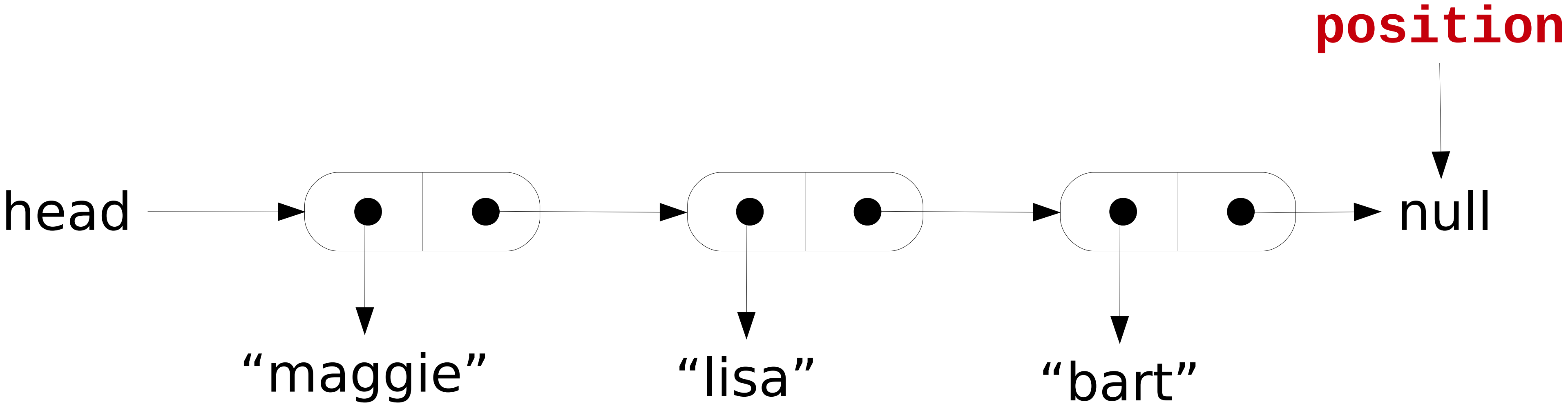
count: 3



StringLinkedList - length

```
public int  length()  {
   int count = 0;
   ListNode position = head;
   while (position != null) {
      count++;
      position = position.getLink();
   }
   return count;
}
```

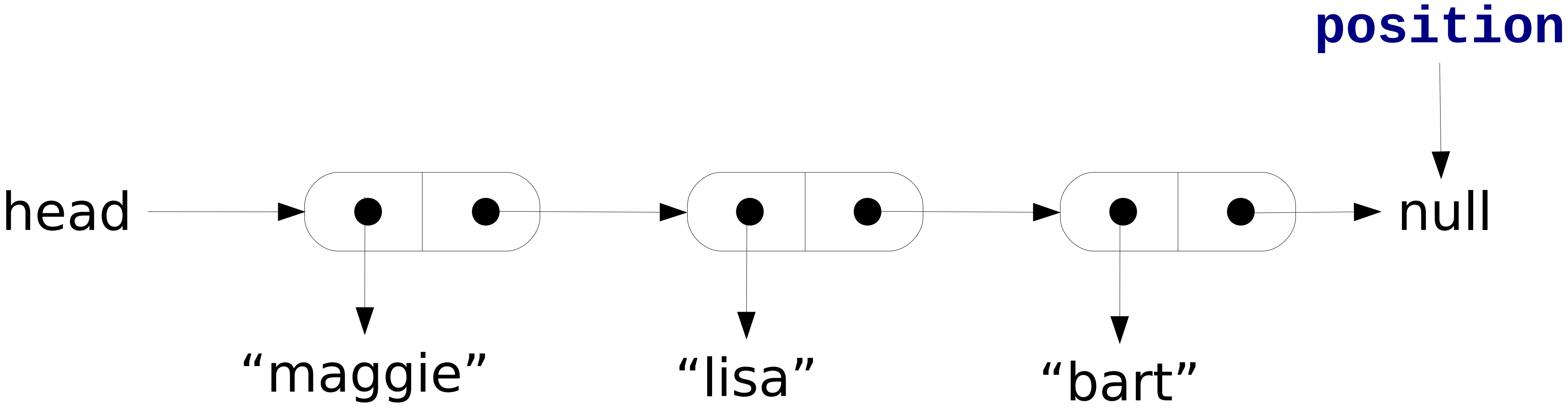
count: 3



StringLinkedList - length

```
public int  length()  {
   int count = 0;
   ListNode position = head;
   while (position != null) {
      count++;
      position = position.getLink();
   }
   return count;
}
```

count: 3



StringLinkedList - deleteHeadNode

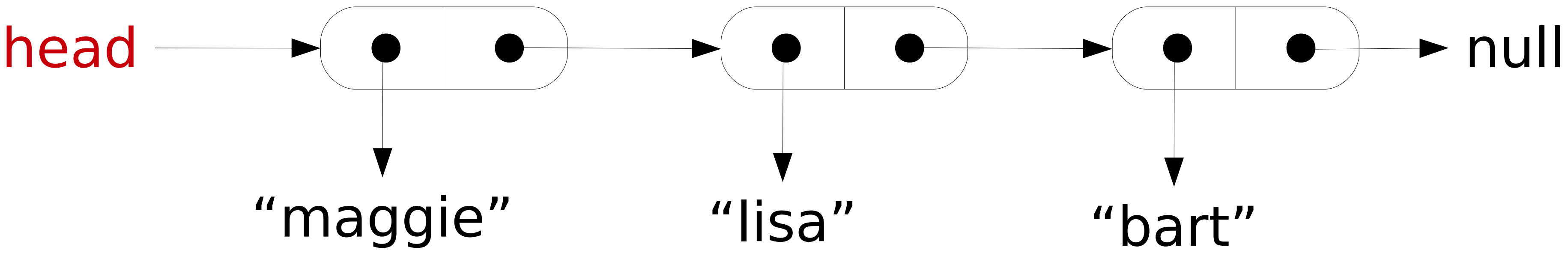
- Can't delete a node from an empty list
- A list is empty if **`head == null`**
- Could also throw an exception if deleting from an empty list (Savitch prints a message and exits)

NullPointerException

- A **`NullPointerException`** indicates that access to an object has been attempted using a **`null`** reference.
- A **`null`** reference means that no object is referenced by the variable.
- A **`NullPointerException`** does not need to be caught or declared in a throws clause. It indicates that the code needs to be fixed.

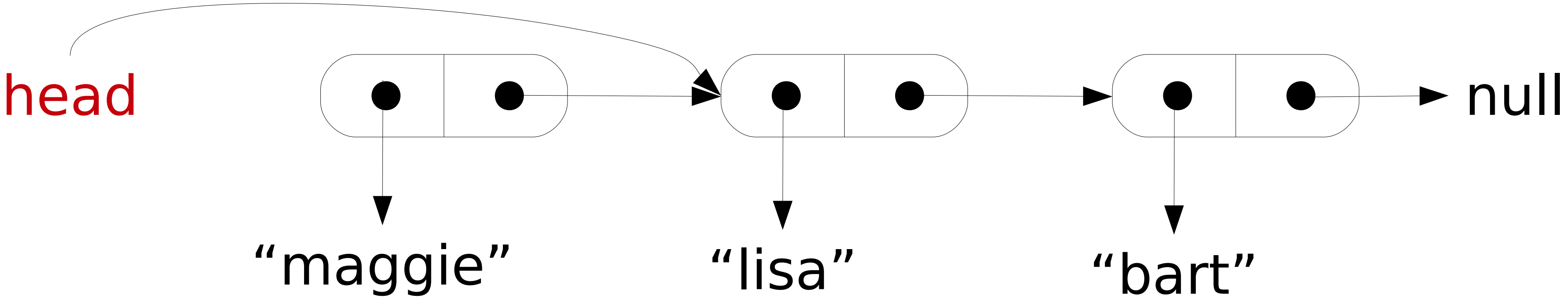
StringLinkedList - deleteHeadNode

```
public void deleteHeadNode()  {
   if (head != null) {
      head = head.getLink();
   } else {
      throw new NullPointerException(“Deleting from empty list”);
   }
}
```



StringLinkedList - deleteHeadNode

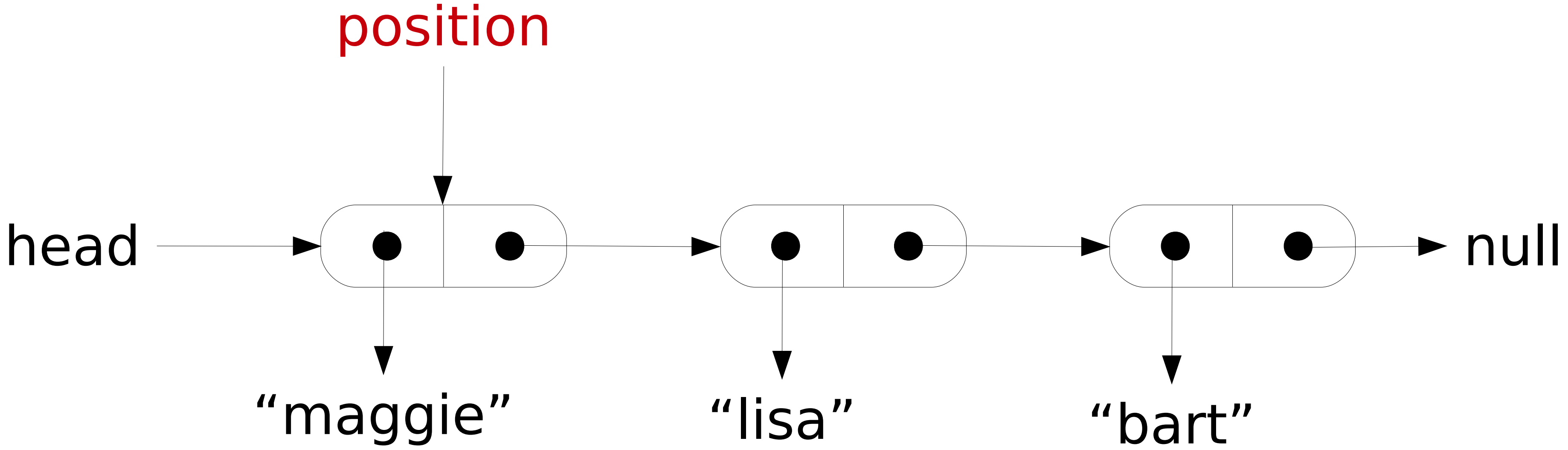
```
public void deleteHeadNode()  {
   if (head != null) {
      head = head.getLink();
   } else {
      throw new NullPointerException(“Deleting from empty list”);
   }
}
```



The “old” head node no longer has a reference to it and is lost.

StringLinkedList - showList()

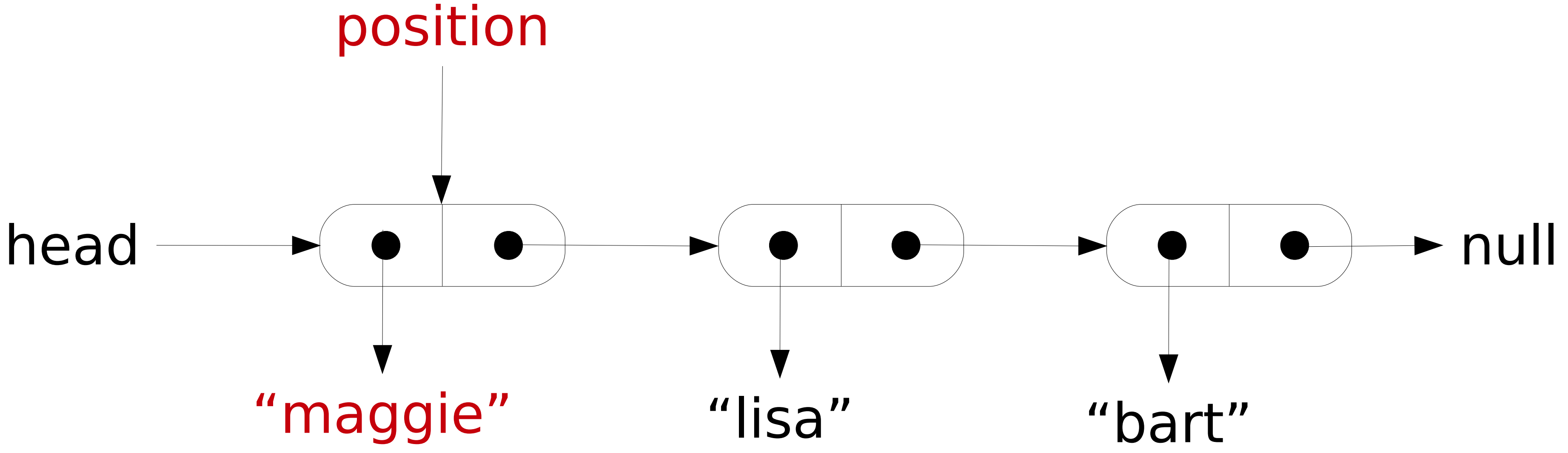
```
public void showList()  {
   ListNode position = head;
   while (position != null) {
      System.out.println(position.getData());
      position = position.getLink();
   }
}
```



StringLinkedList - showList()

```
public void showList()  {
   ListNode position = head;
   while (position != null) {
      System.out.println(position.getData());
      position = position.getLink();
   }
}
```

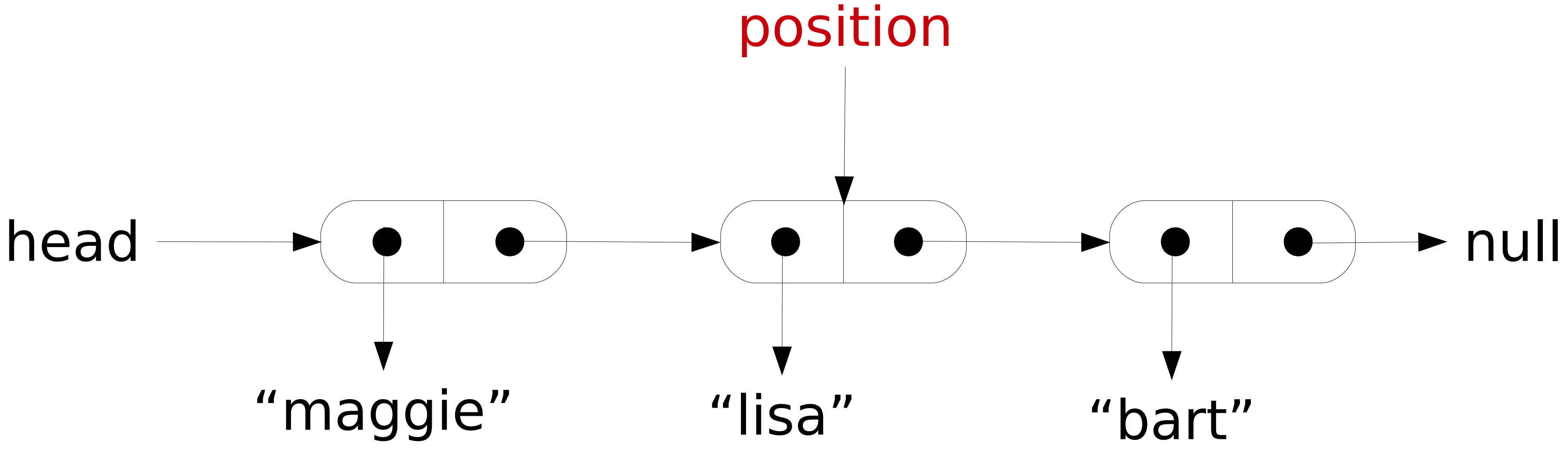
maggie



StringLinkedList - showList()

```
public void showList()  {
   ListNode position = head;
   while (position != null) {
      System.out.println(position.getData());
      position = position.getLink();
   }
}
```

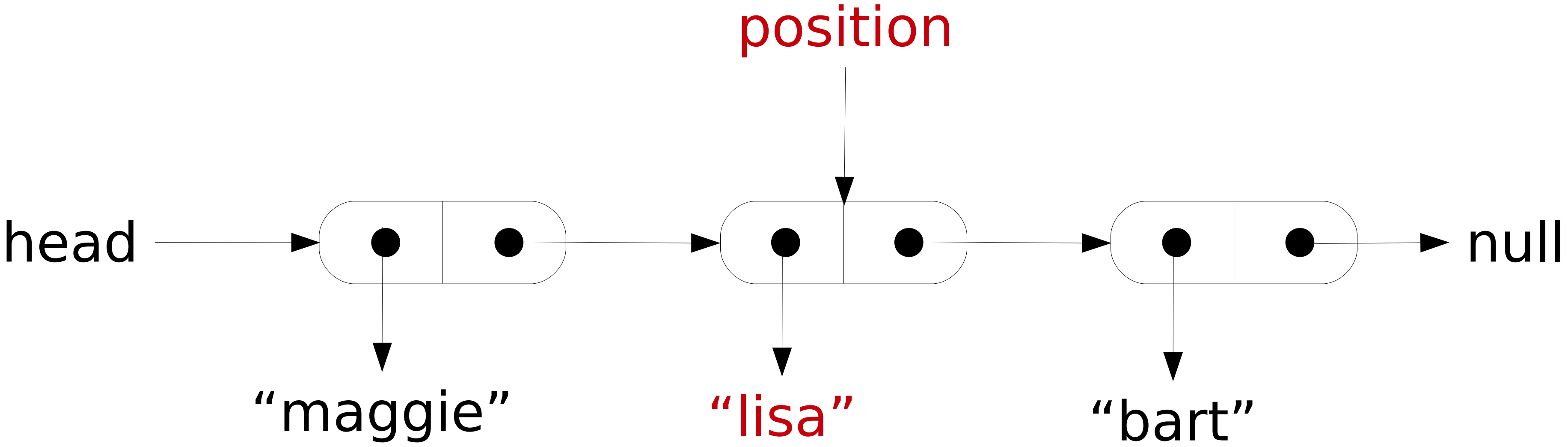
maggie



StringLinkedList - showList()

```
public void showList()  {
   ListNode position = head;
   while (position != null) {
      System.out.println(position.getData());
      position = position.getLink();
   }
}
```

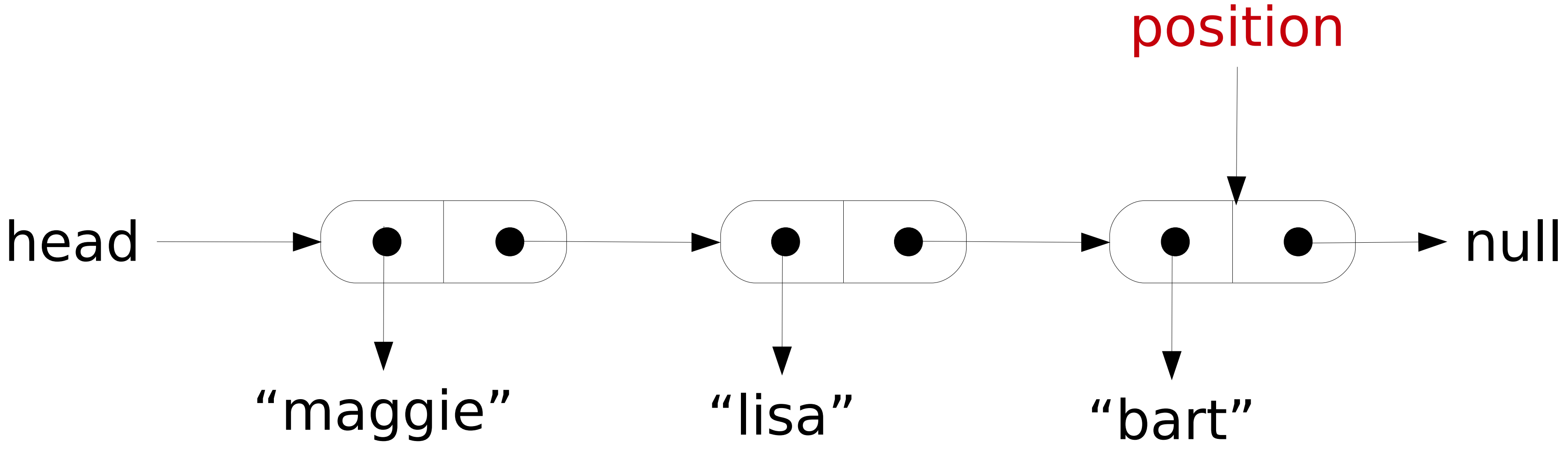
maggie
lisa



StringLinkedList - showList()

```
public void showList()  {
   ListNode position = head;
   while (position != null) {
      System.out.println(position.getData());
      position = position.getLink();
   }
}
```

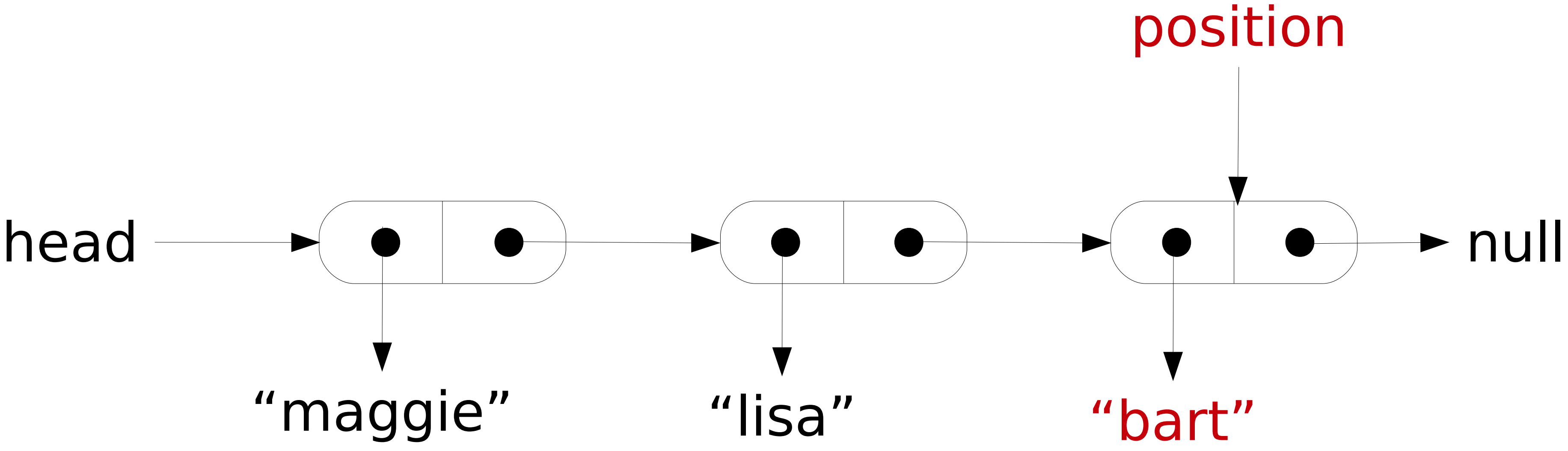
```
maggie
lisa
```



StringLinkedList - showList()

```
public void showList()  {
   ListNode position = head;
   while (position != null) {
      System.out.println(position.getData());
      position = position.getLink();
   }
}
```

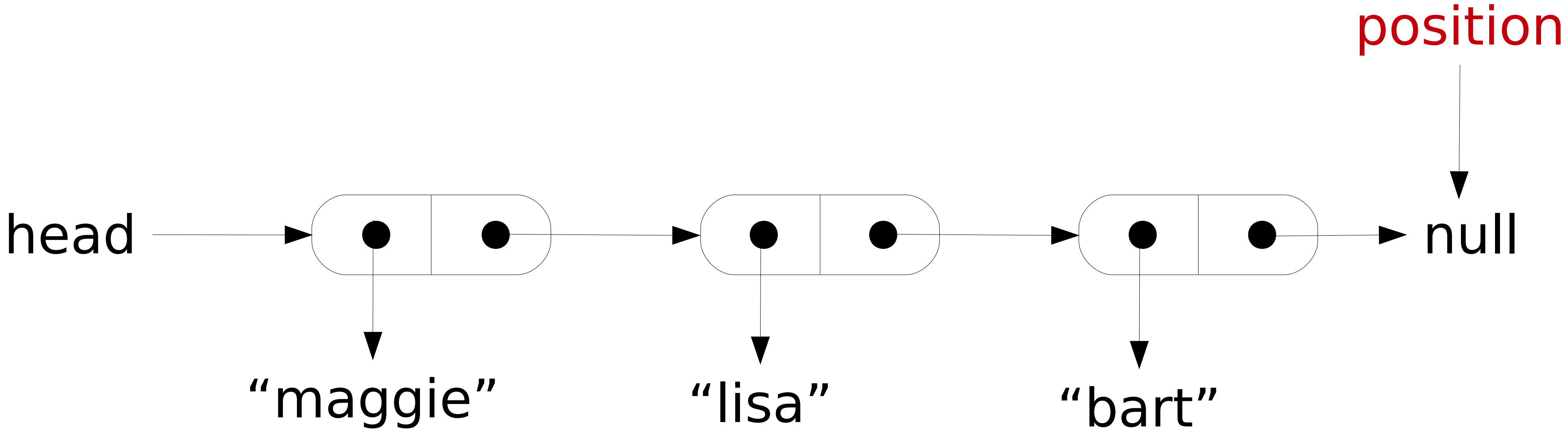
maggie
lisa
bart



StringLinkedList - showList()

```
public void showList()  {
   ListNode position = head;
   while (position != null) {
      System.out.println(position.getData());
      position = position.getLink();
   }
}
```

```
maggie
lisa
bart
```



Privacy Leaks

- The **`getLink`** method in class **`ListNode`** (page 837) returns an instance variable which is a reference to a node.
- A user could modify the data stored in that node, defeating the **`private`** restriction on the instance variable **`link`**.
- This problem can be fixed by making the class **`ListNode`** a private inner class of **`StringLinkedList`**.

Inner Classes

- An *inner class* is a class defined within another class
- Defining an inner class:

```
public class OuterClass {

   // OuterClass instance variables
   // OuterClass methods

   private class InnerClass {
       // InnerClass instance variables
       // InnerClass methods
   }
}
```

Inner Classes - Access

- The inner and outer classes have access to each other's methods and instance variables, even if they are declared **`private`**.

Node Inner Class

`StringLinkedListSelfContained`

Savitch listing 12.5, p 849-851

- By making the node class in inner class of the linked list class, the linked list class becomes self-contained.
- The accessor (get-) and mutator (set-) methods can be eliminated from the node class.
- They are no longer needed because the instance variables are directly accessible.

List Iteration

- We often need to “step through” all the objects in a list and perform some operation on each object.
- An *iterator* allows us to “step through” a collection of objects (in this case a list of nodes).

List Iteration

- The loop control variable of a for-loop functions as an iterator for an array:

```
for (int i=0; i < a.length; i++)
    System.out.println(a[i]);
```

- We could place all the elements of a linked list into an array and “step through” the elements by iterating the array. This is called *external* iteration.

List Iteration

- We will implement an *internal* iterator – one that uses an instance variable to step through the nodes.
- An instance variable in the linked list class capable of referencing a node can serve the same purpose as a loop control variable in a for loop.
- This allows us to “step through” the list and access or change data contained in the nodes or to insert and delete nodes.

StringLinkedListWithIterator

Savitch listing 12.7, p 852-855

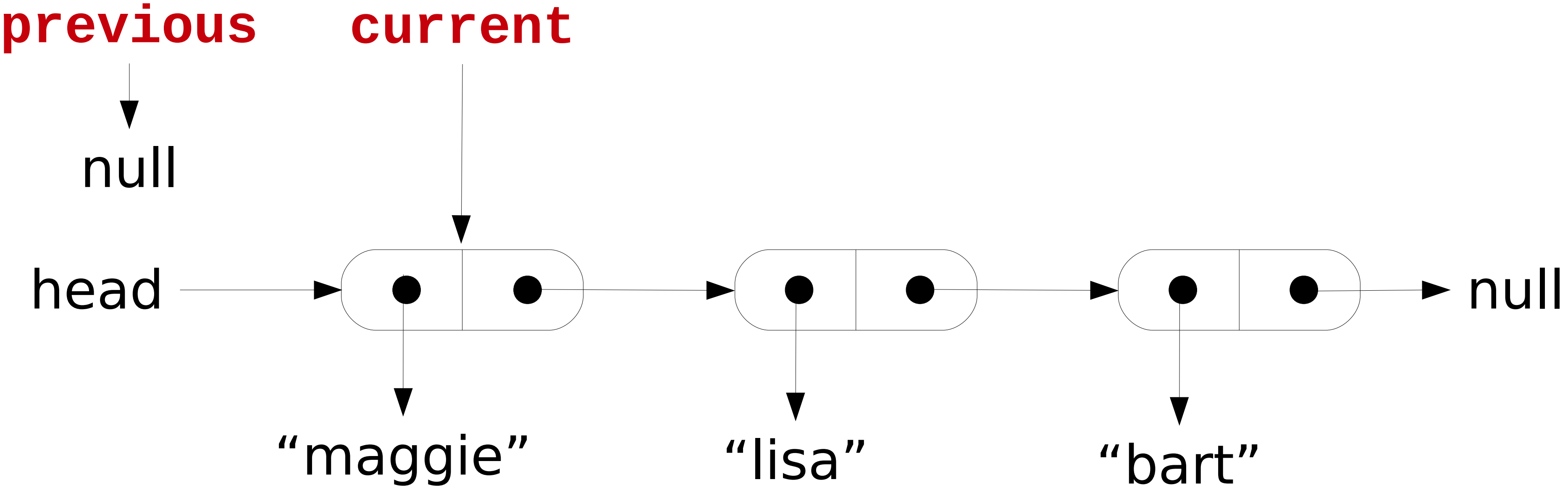
- Additional methods:
  - **`resetIteration`**
  - **`goToNext`**
  - **`deleteCurrentNode`**
  - **`insertNodeAfterCurrent`**
  - **`moreToIterate`**
  - **`getDataAtCurrent`**
  - **`setDataAtCurrent`**
- Must modify **`addANodeToStart`** method

StringLinkedListWithIterator

- We need 2 more instance variables:
  - **`private ListNode current;`**
  - **`private ListNode previous;`**
- **`current`** should always reference the node currently being processed
- **`previous`** should always reference the node behind current and is needed if we want to delete the current node

StringLinkedListWithIterator - resetIteration

```
public void resetIteration()  {
   // current, previous, head are all instance variables
   current = head;
   previous = null;
}
```



Advancing to the next node

Possible situations:

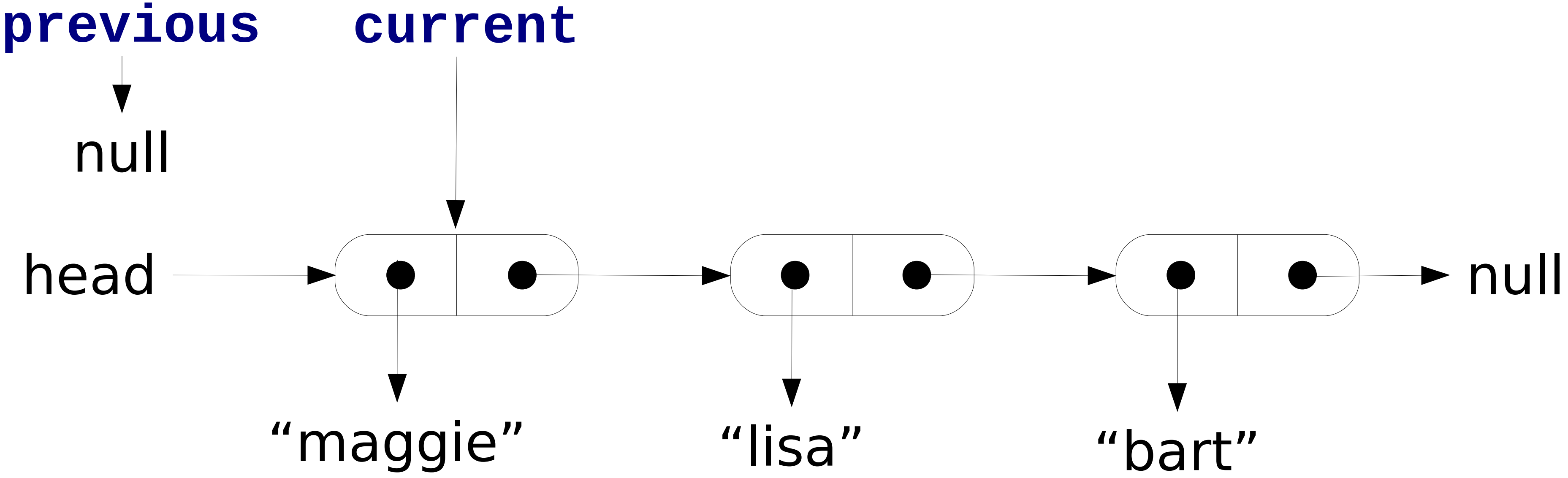
- There is a next node to advance to
- Errors: There is **not** a next node because:
  - the iteration was not initialized by calling **`resetIteration`**

    or
  - the iteration moved past end of list

    or
  - the list is empty

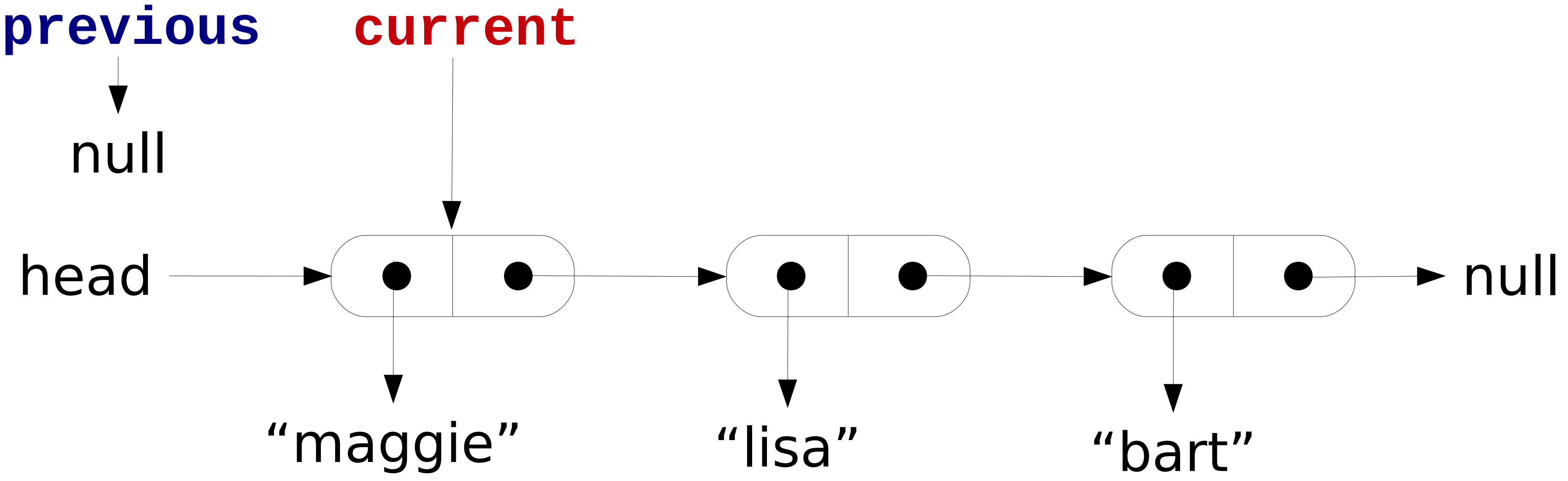
StringLinkedListWithIterator - goToNext

- The user must call **`resetIteration`** before the first call to **`goToNext`**
- If **`current != null`**, it is safe to go to the next node



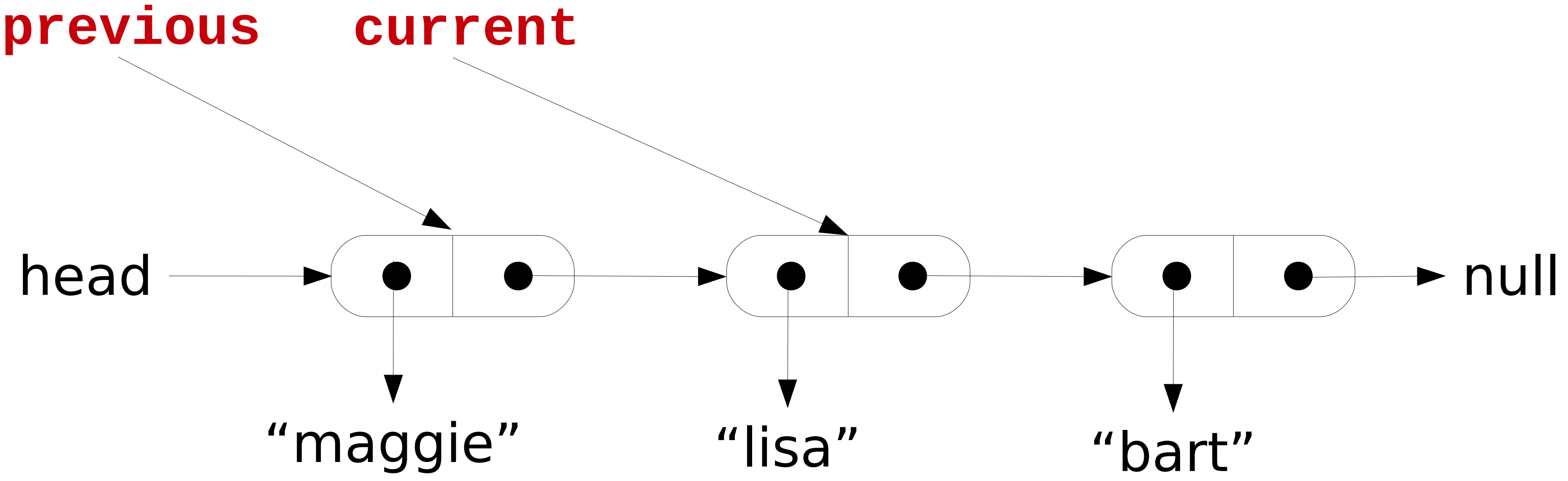
StringLinkedListWithIterator - goToNext success

```
public void goToNext()  {
   if (current != null) {
      previous = current;
      current = current.link;
   } else if (head != null) {
      throw new LinkedListException(“Iterated too many times “ +
                                    “or uninitialized iteration”);
   } else {
      throw new LinkedListException(“Iterating empty list”);
   }
}
```



StringLinkedListWithIterator - goToNext success

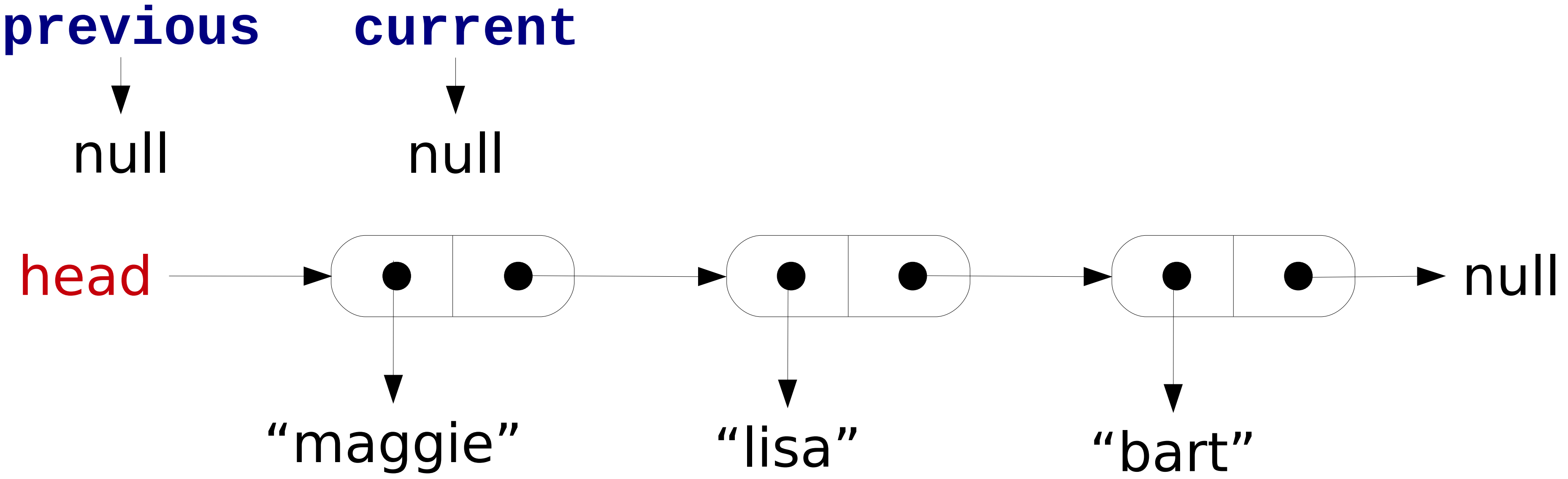
```
public void goToNext()  {
   if (current != null) {
      previous = current;
      current = current.link;
   } else if (head != null) {
      throw new LinkedListException(“Iterated too many times “ +
                                    “or uninitialized iteration”);
   } else {
      throw new LinkedListException(“Iterating empty list”);
   }
}
```



StringLinkedListWithIterator - goToNext

uninitialized iteration

```
public void goToNext()  {
   if (current != null) {
      previous = current;
      current = current.link;
   } else if (head != null) {
      throw new LinkedListException(“Iterated too many times “ +
                                    “or uninitialized iteration”);
   } else {
      throw new LinkedListException(“Iterating empty list”);
   }
}
```



previous

current

null

null

head

null

“maggie”

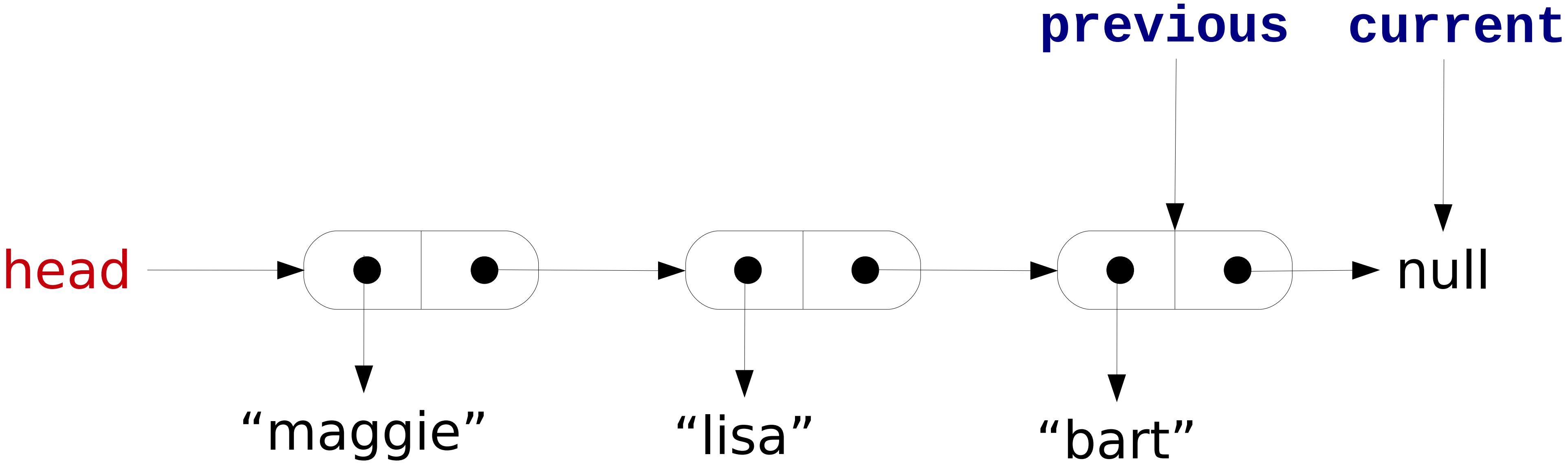
“lisa”

“bart”

StringLinkedListWithIterator - goToNext

done iterating

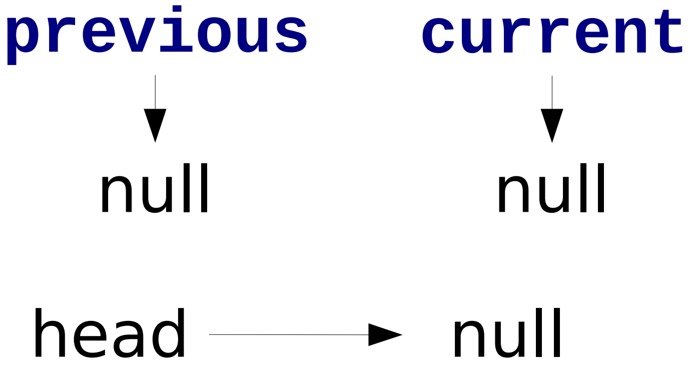
```
public void goToNext()  {
   if (current != null) {
      previous = current;
      current = current.link;
   } else if (head != null) {
      throw new LinkedListException("Iterated too many times " +
                                    "or uninitialized iteration");
   } else {
      throw new LinkedListException("Iterating empty list");
   }
}
```



StringLinkedListWithIterator - goToNext

empty list

```
public void goToNext()  {
   if (current != null) {
      previous = current;
      current = current.link;
   } else if (head != null) {
      throw new LinkedListException(“Iterated too many times “ +
                                    “or uninitialized iteration”);
   } else {
      throw new LinkedListException(“Iterating empty list”);
   }
}
```



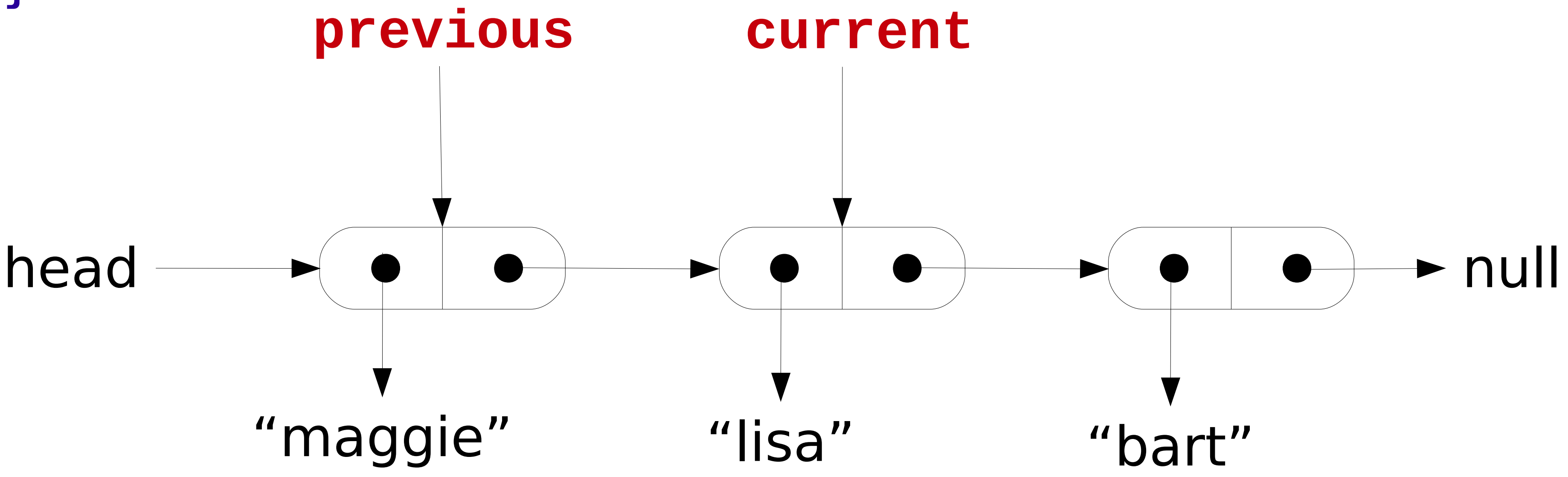
StringLinkedListWithIterator - deleteCurrentNode

Possible situations:

- We are deleting a node from the middle of the list
- We are deleting the head node
- Error: there is no current node to delete

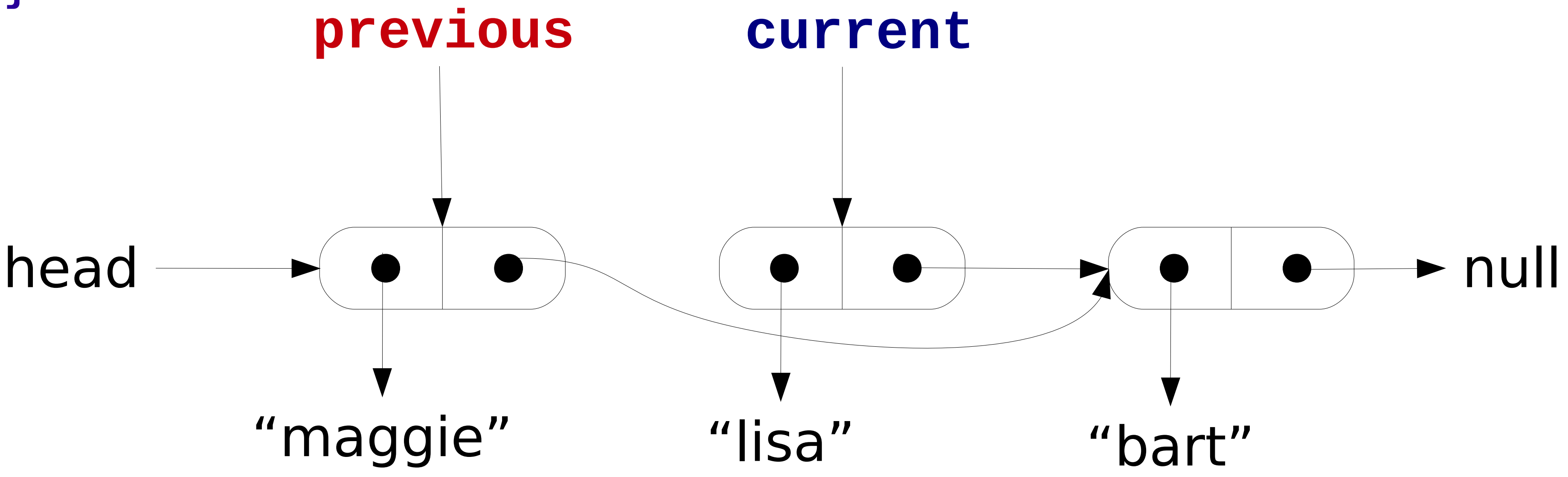
StringLinkedListWithIterator - deleteCurrentNode from the middle

```
public void deleteCurrentNode()  {
  if ((current != null) && (previous != null)) {
      previous.link = current.link;
      current = current.link;
   } else if ((current != null) && (previous == null)) {
      head = current.link;
      current = head;
   } else {
      throw new LinkedListException("Deleting uninitialized " +
                                    "current or list is empty");
   }
}
```



StringLinkedListWithIterator - deleteCurrentNode
from the middle

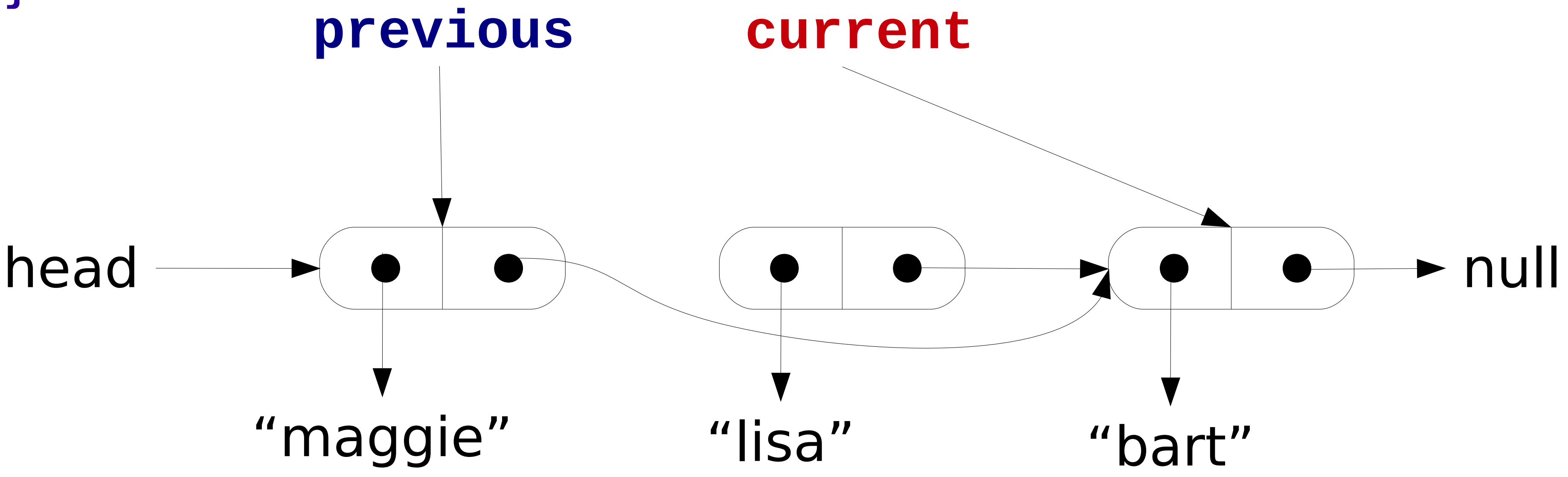
```
public void deleteCurrentNode()  {
  if ((current != null) && (previous != null)) {
      previous.link = current.link;
      current = current.link;
   } else if ((current != null) && (previous == null)) {
      head = current.link;
      current = head;
   } else {
      throw new LinkedListException("Deleting uninitialized " +
                                    "current or list is empty");
   }
}
```



StringLinkedListWithIterator - deleteCurrentNode

from the middle

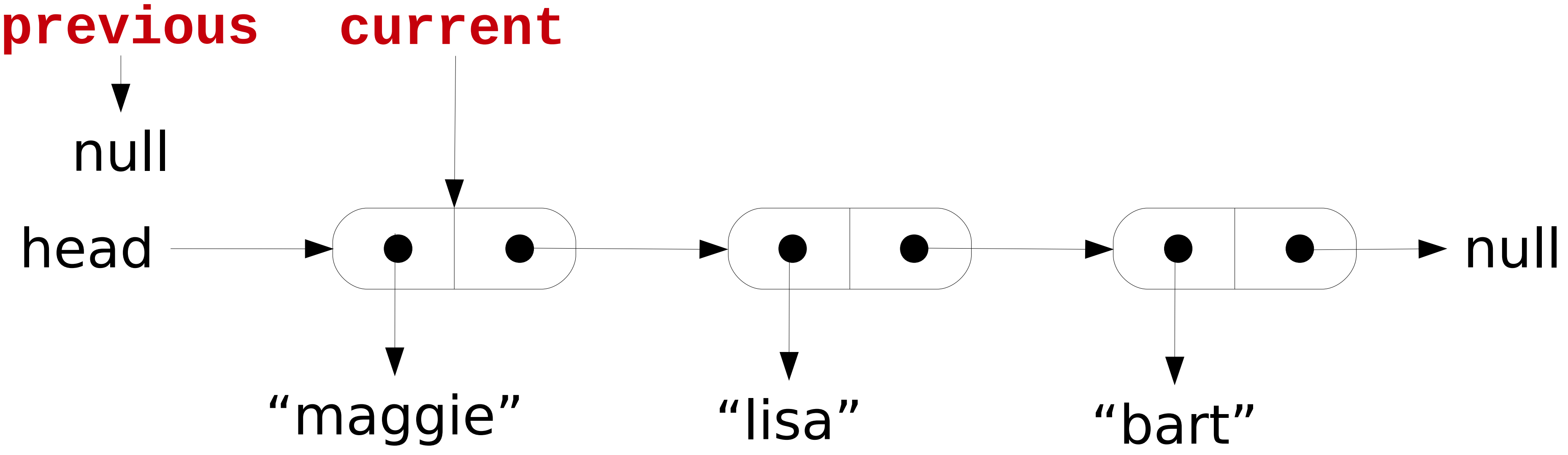
```
public void deleteCurrentNode()  {
  if ((current != null) && (previous != null)) {
      previous.link = current.link;
      current = current.link;
   } else if ((current != null) && (previous == null)) {
      head = current.link;
      current = head;
   } else {
      throw new LinkedListException("Deleting uninitialized " +
                                    "current or list is empty");
   }
}
```



StringLinkedListWithIterator - deleteCurrentNode

head node

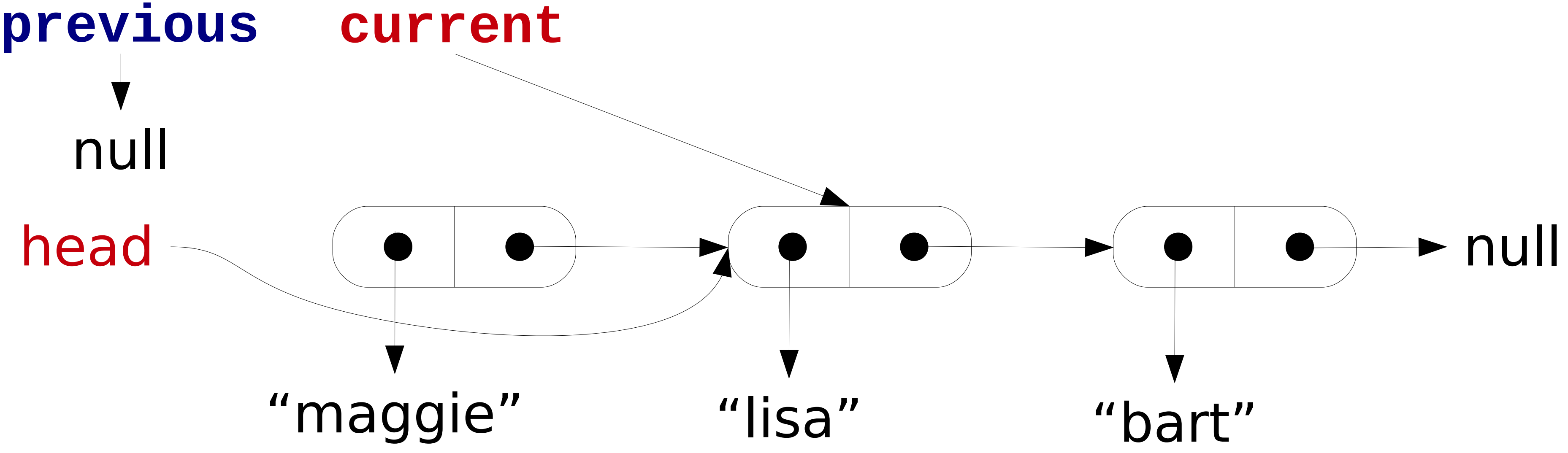
```
public void deleteCurrentNode()  {
  if ((current != null) && (previous != null)) {
      previous.link = current.link;
      current = current.link;
   } else if ((current != null) && (previous == null)) {
      head = current.link;
      current = head;
   } else {
      throw new LinkedListException("Deleting uninitialized " +
                                    "current or list is empty");
   }
}
```



StringLinkedListWithIterator - deleteCurrentNode

head node

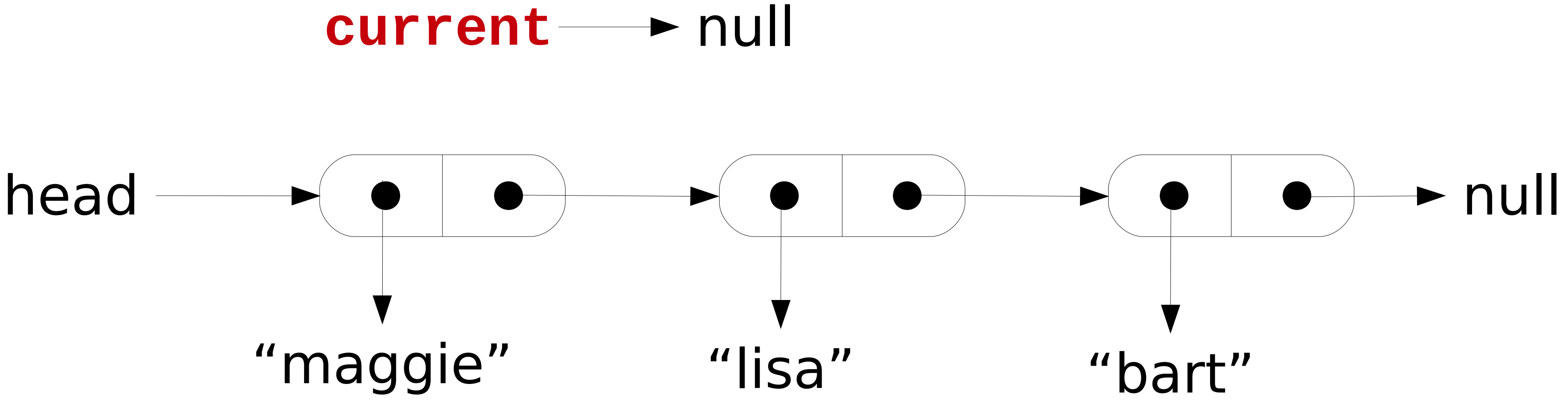
```
public void deleteCurrentNode()  {
  if ((current != null) && (previous != null)) {
      previous.link = current.link;
      current = current.link;
   } else if ((current != null) && (previous == null)) {
      head = current.link;
      current = head;
   } else {
      throw new LinkedListException("Deleting uninitialized " +
                                    "current or list is empty");
   }
}
```



StringLinkedListWithIterator - deleteCurrentNode

no current node

```
public void deleteCurrentNode()  {
  if ((current != null) && (previous != null)) {
      previous.link = current.link;
      current = current.link;
   } else if ((current != null) && (previous == null)) {
      head = current.link;
      current = head;
   } else {
      throw new LinkedListException("Deleting uninitialized " +
                                    "current or list is empty");
   }
}
```



StringLinkedListWithIterator - insertNodeAfterCurrent

Possible situations:

- There is a current node to insert after
- Errors: There is **not** a current node because:
  - the iteration was not initialized by calling **resetIteration**

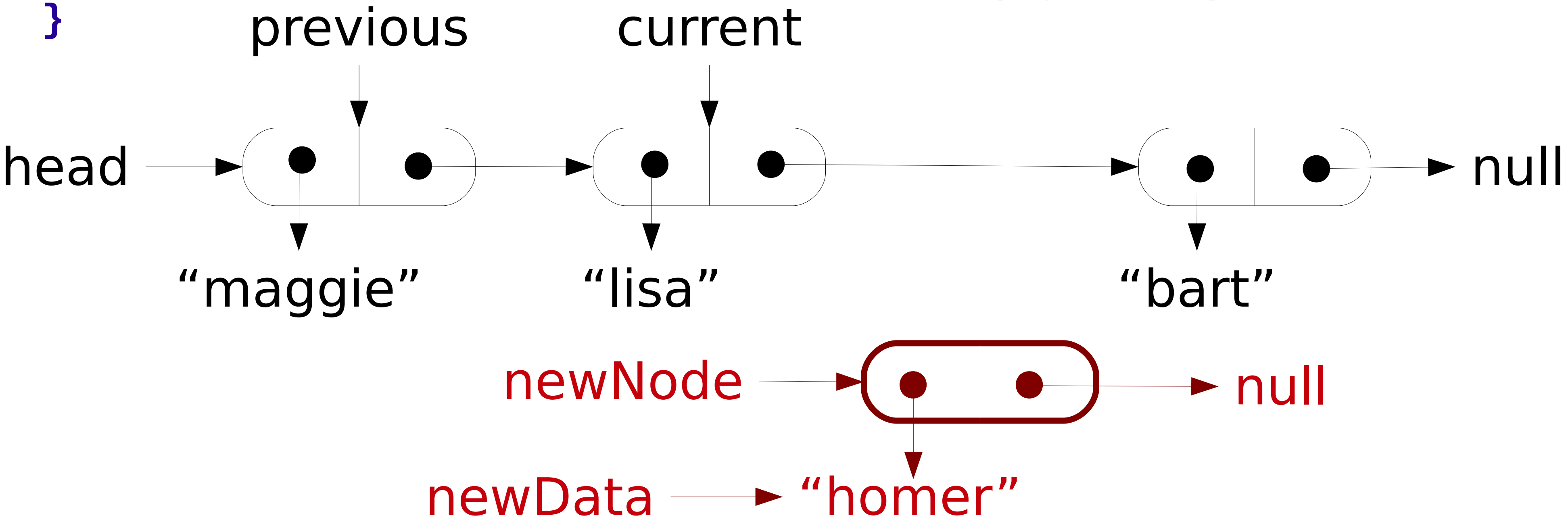
    or
  - the iteration moved past end of list

    or
  - the list is empty

StringLinkedListWithIterator - insertNodeAfterCurrent

success

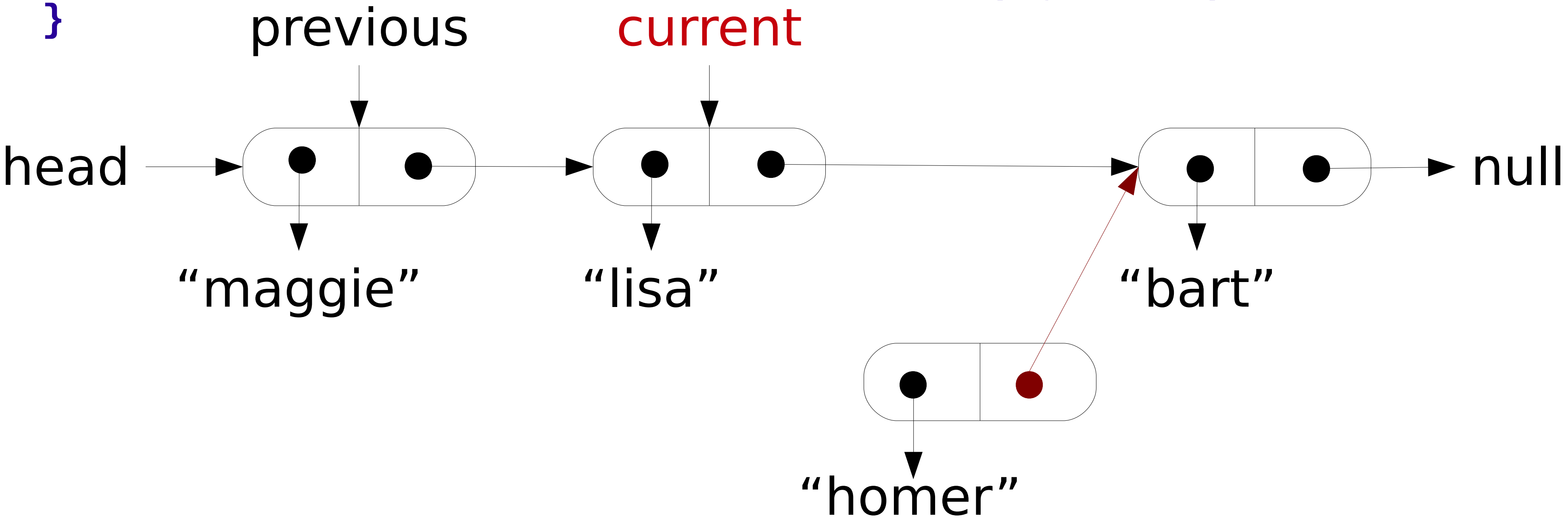
```
public void insertNodeAfterCurrent(String newData)  {
   ListNode newNode = new ListNode(newData, null);
   if (current != null) {
      newNode.link = current.link;
      current.link = newNode;
   } else if (head != null) {
      throw new LinkedListException(“Inserting when iterator is “ +
                                 “past all nodes or uninitialized iterator”);
   } else {
      throw new LinkedListException(“Using insertNodeAfterCurrent “ +
                                          “with empty list”);
   }
}
```



StringLinkedListWithIterator - insertNodeAfterCurrent

success

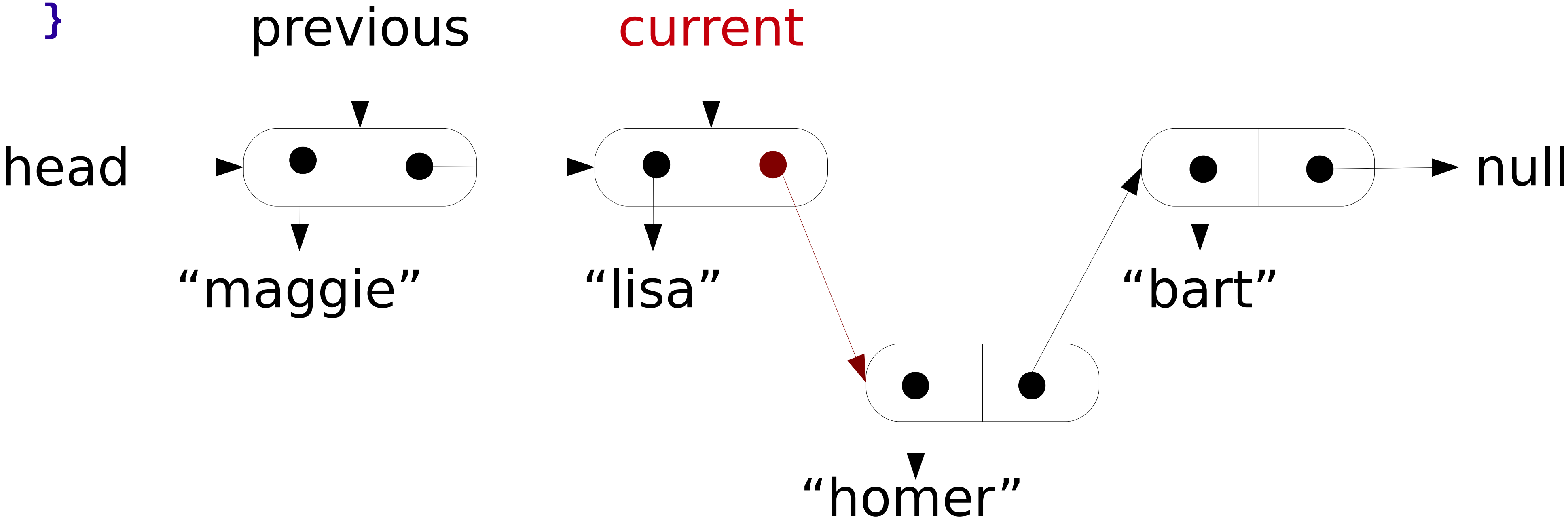
```
public void insertNodeAfterCurrent(String newData)  {
   ListNode newNode = new ListNode(newData, null);
   if (current != null) {
      newNode.link = current.link;
      current.link = newNode;
   } else if (head != null) {
      throw new LinkedListException(“Inserting when iterator is “ +
                       “past all nodes or uninitialized iterator”);
   } else {
      throw new LinkedListException(“Using insertNodeAfterCurrent “ +
                                    “with empty list”);
   }
}
```



StringLinkedListWithIterator - insertNodeAfterCurrent

success

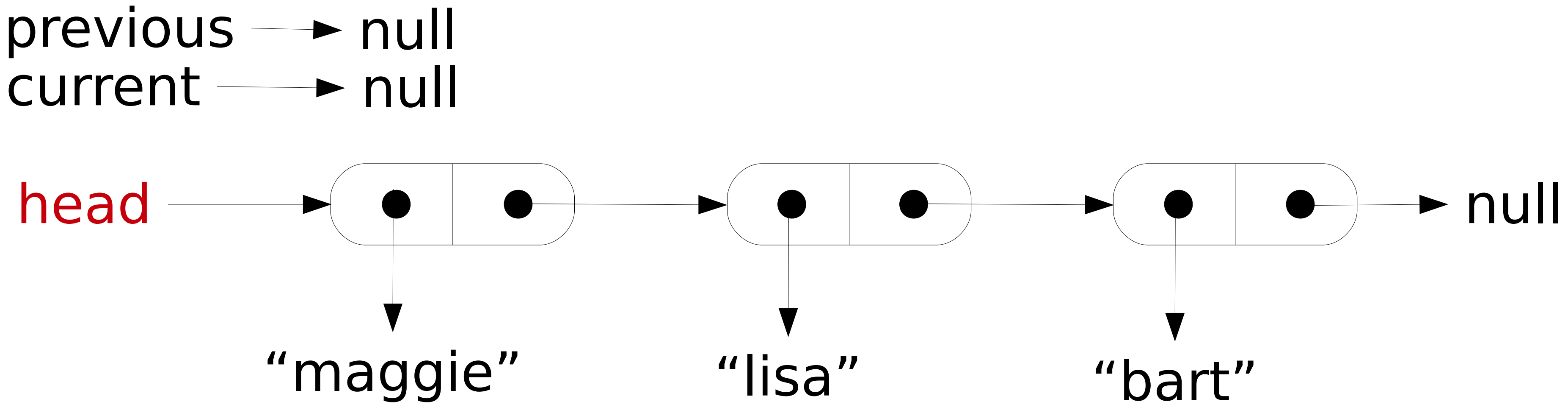
```
public void insertNodeAfterCurrent(String newData)  {
   ListNode newNode = new ListNode(newData, null);
   if (current != null) {
      newNode.link = current.link;
      current.link = newNode;
   } else if (head != null) {
      throw new LinkedListException(“Inserting when iterator is “ +
                       “past all nodes or uninitialized iterator”);
   } else {
      throw new LinkedListException(“Using insertNodeAfterCurrent “ +
                                    “with empty list”);
   }
}
```



StringLinkedListWithIterator - insertNodeAfterCurrent

uninitialized iteration

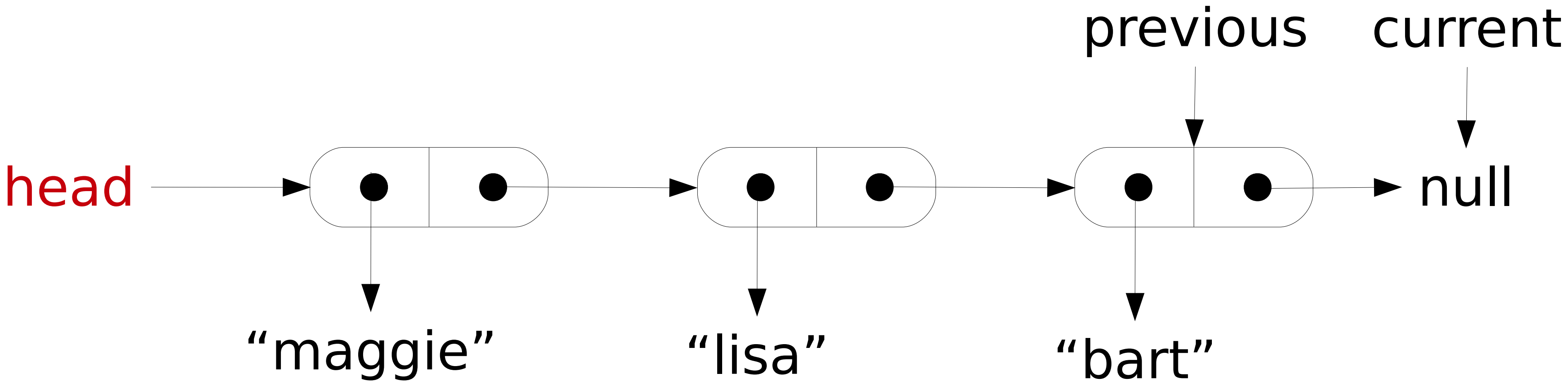
```
public void insertNodeAfterCurrent(String newData)  {
   ListNode newNode = new ListNode(newData, null);
   if (current != null) {
      newNode.link = current.link;
      current.link = newNode;
   } else if (head != null) {
      throw new LinkedListException(“Inserting when iterator is “ +
                                 “past all nodes or uninitialized iterator”);
   } else {
      throw new LinkedListException(“Using insertNodeAfterCurrent “ +
                                               “with empty list”);
   }
}
```



StringLinkedListWithIterator - insertNodeAfterCurrent

done iterating

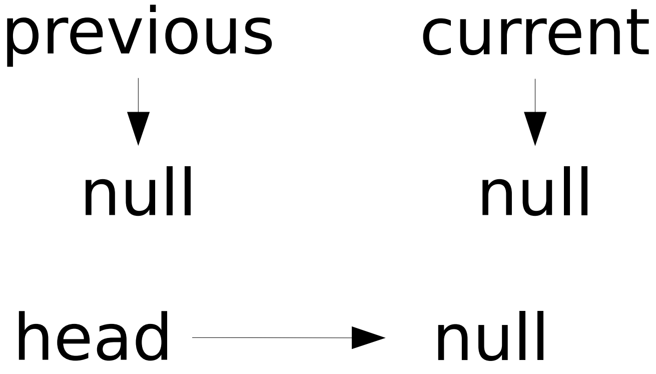
```
public void insertNodeAfterCurrent(String newData)  {
   ListNode newNode = new ListNode(newData, null);
   if (current != null) {
      newNode.link = current.link;
      current.link = newNode;
   } else if (head != null) {
      throw new LinkedListException("Inserting when iterator is " +
                                "past all nodes or uninitialized iterator");
   } else {
      throw new LinkedListException("Using insertNodeAfterCurrent " +
                                             "with empty list");
   }
}
```



StringLinkedListWithIterator - insertNodeAfterCurrent

empty list

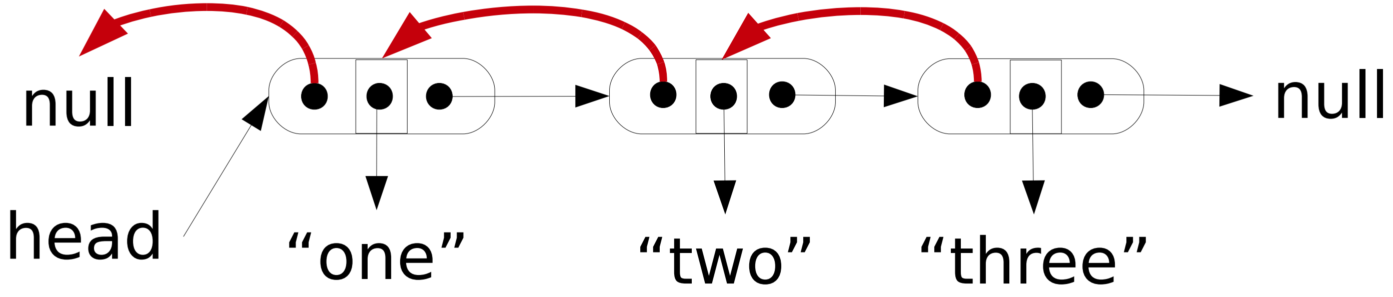
```
public void insertNodeAfterCurrent(String newData)  {
   ListNode newNode = new ListNode(newData, null);
   if (current != null) {
      newNode.link = current.link;
      current.link = newNode;
   } else if (head != null) {
      throw new LinkedListException(“Inserting when iterator is “ +
                       “past all nodes or uninitialized iterator”);
   } else {
      throw new LinkedListException(“Using insertNodeAfterCurrent “ +
                                    “with empty list”);
   }
}
```



Doubly Linked Lists

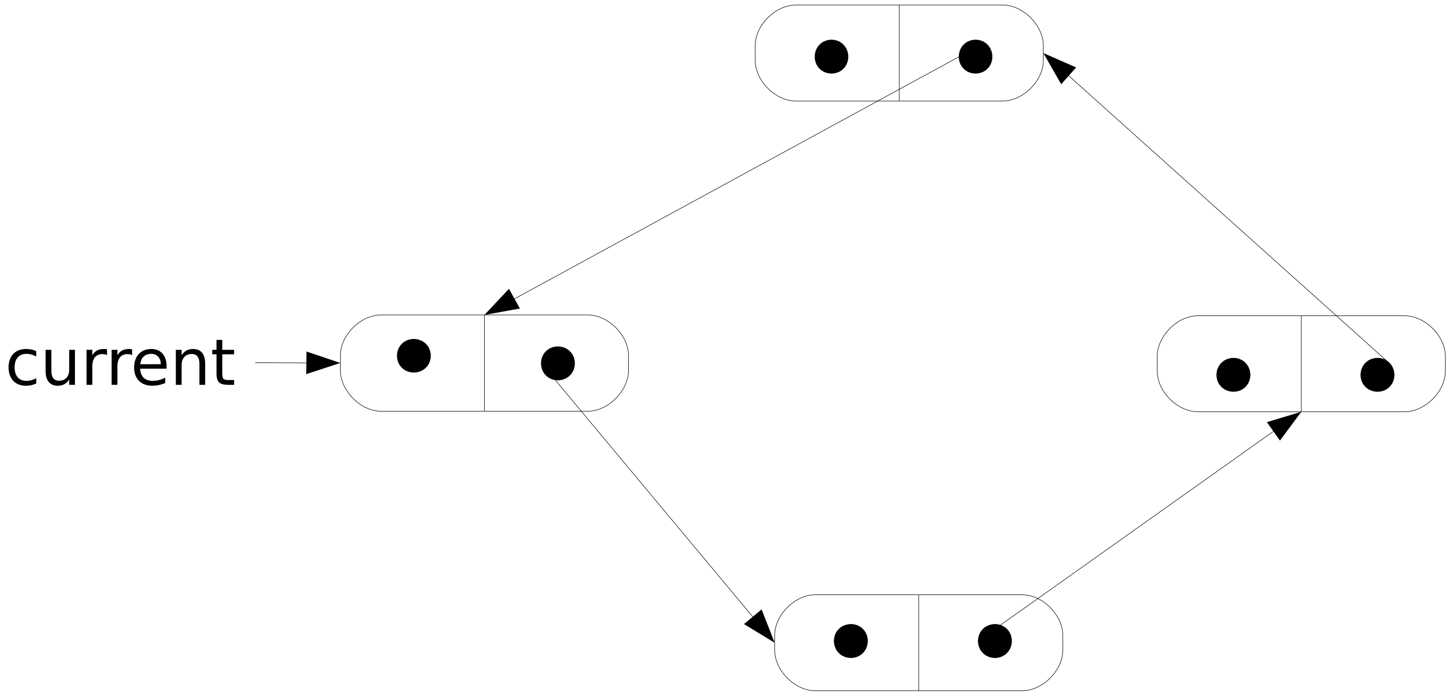
- An additional reference – to the previous node - can be added to the node class, producing a *doubly-linked list*.

```
private class ListNode {
    private String data;
    private ListNode next;
    private ListNode prev;
    ...
}
```



Circularly Linked Lists

- The last node in a singly-linked list can reference the first node, producing a *circularly-linked list*.



Generics

- Java (starting with version 5.0) allows definitions that include parameters for types. This is known as *generics*.
- For example, Java's **`ArrayList`** class allows us to specify the type of the objects stored in the list.
- Generics can be difficult to program.
- We will learn enough about generics to use them to define flexible data structure classes.

Generics

- Classes and methods can have a type parameter (like **`ArrayList<BaseType>`**)
- Any class type can be substituted for the **`BaseType`**

```
ArrayList<String> = new ArrayList<String>();
ArrayList<Integer> = new ArrayList<Integer>();
ArrayList<EnglishNoun> =
                    new ArrayList<EnglishNoun>();
```

- We can define our own generic classes.
- In the class definition, a *type parameter* **`T`** is used as a placeholder for the **`BaseType`**.

Generics - Sample<T>

Savitch listing 12.9, p 866

Using the **Sample<T>** class:

```
Sample<String> s1 = new Sample<String>();
s1.setData(“Hello”);
System.out.println(s1.getData());

Sample<Word> s2 = new Sample<Word>();
Word aWord = new Word(“blah”);
s2.setData(aWord);
```

Generics - `LinkedList<E>`

Savitch listing 12.10, p 870-871

- Use the type parameter **`E`** instead of a particular base type.
- By using a type parameter instead of a particular base type (**`String`**, for example), we can create a linked list with any type of data (**`Word`**, **`Car`**, **`BankAccount`**,...)

Generics - LinkedList<E>

- The constructor heading does NOT include the type parameter in angle brackets.

```
// constructor
public LinkedList<E>() { //ILLEGAL
    // constructor code
}

// constructor
public LinkedList() {    //GOOD
    // constructor code
}
```

Generics - `LinkedList<E>`

- The **`ListNode`** inner class heading also does NOT include the type parameter, but it can use the type parameter anyway.

```
// inner node class
private class ListNode {
    private E data;

    ...

}
```

Limited Use of a Type Parameter

- Within the definition of a parameterized class, there are places where a type name is allowed, but a type parameter is NOT allowed.
- Type parameters cannot be used in simple expressions that use **new** to create a new object.

Limited Use of a Type Parameter

- Examples:

```
T someObject;                 // LEGAL
someObject = new T();  // ILLEGAL

T[] someArray;                // LEGAL
someArray = new T[10]; // ILLEGAL
```

In both cases, the first **T** is legal, but the second **T** is illegal.

Iteration

- The most common way to implement iteration of a collection of objects is by implementing **`Iterator<T>`**.
- Java provides an interface **`Iterator<T>`**
  - used by classes that represent a collection of objects
  - providing a way of moving through the collection one object at a time
  - defined in the java standard class library

Iterator<T> Interface

- The two primary methods of the interface **`Iterator<T>`** are:
    - **`public T next();`** (returns an object)
    - **`public boolean hasNext();`** (returns a boolean value)
- There is also an optional **`remove`** method. Even if you choose not to implement it, you must provide a **`remove`** method that simply throws an **`UnsupportedOperationException`**.
- The **`Scanner`** class implements this interface

`Iterator<T>` Object

- The idea is to provide an **object** that iterates over the collection.
- This object can then call the **`hasNext`** and **`next`** methods to process the collection.

Using an Iterator<T> Object

- Sample use of an **Iterator<T>** object:

```
LinkedList<String> list = new LinkedList<String>();
list.add(“Hello”);
list.add(“you”);
list.add(“there”);

Iterator<String> iter = list.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```

Implementing Iterator<T>

- We need to provide a method (normally called **iterator**) that returns an object of type **Iterator<T>**.

```
public Iterator<T> iterator() {
   return new LinkedListIterator();
}
```

Implementing `Iterator<T>`

- Where does the **`LinkedListIterator`** come from?
- **`LinkedListIterator`** is implemented as an **inner class** of the collection class (**`LinkedList`** in this case).
- **`LinkedListIterator`** implements the **`Iterator<T>`** interface and provides the required **`next`**, **`hasNext`**, and **`remove`** methods.

Implementing `Iterator<T>`

- The **`LinkedListIterator`** inner class also needs:
  - an instance variable **`current`** that refers to the current node
  - A constructor that initializes **`current`** to the head of the list.

Implementing Iterator<T>

```
import java.util.*;
public class LinkedList<T> {

    private ListNode head;

    // constructors and other methods...

    public Iterator<T> iterator() {
        return new LinkedListIterator();
    }
```

Continued...

Implementing Iterator<T>

```
...Continued
    private class LinkedListIterator
                      implements Iterator<T> {
       private ListNode current;

       private LinkedListIterator() {
           current = head;
       }
       public boolean hasNext() { ... }
       public T next() { ... }
       public void remove() { ... }
    }

    private class ListNode {
       ...
    }
}
```