Iterables, Iterators and Collections

Computer Science 112
Boston University

Christine Papadakis-Kanaris

Collections

Java provides the internal mechanism that allow application work with and process a collection of Objects!

Specifically, Java allows you to work on any group of Objects as a single unit.

And all Collections share a common interface.

Iteration Abstraction

Iterators provide the ability to *iterate* over arbitrary types of data.

```
For all elements of the set
    Perform some action
```

On a custom *Collection* class how can java know what the next object should be?

Iteration Abstraction

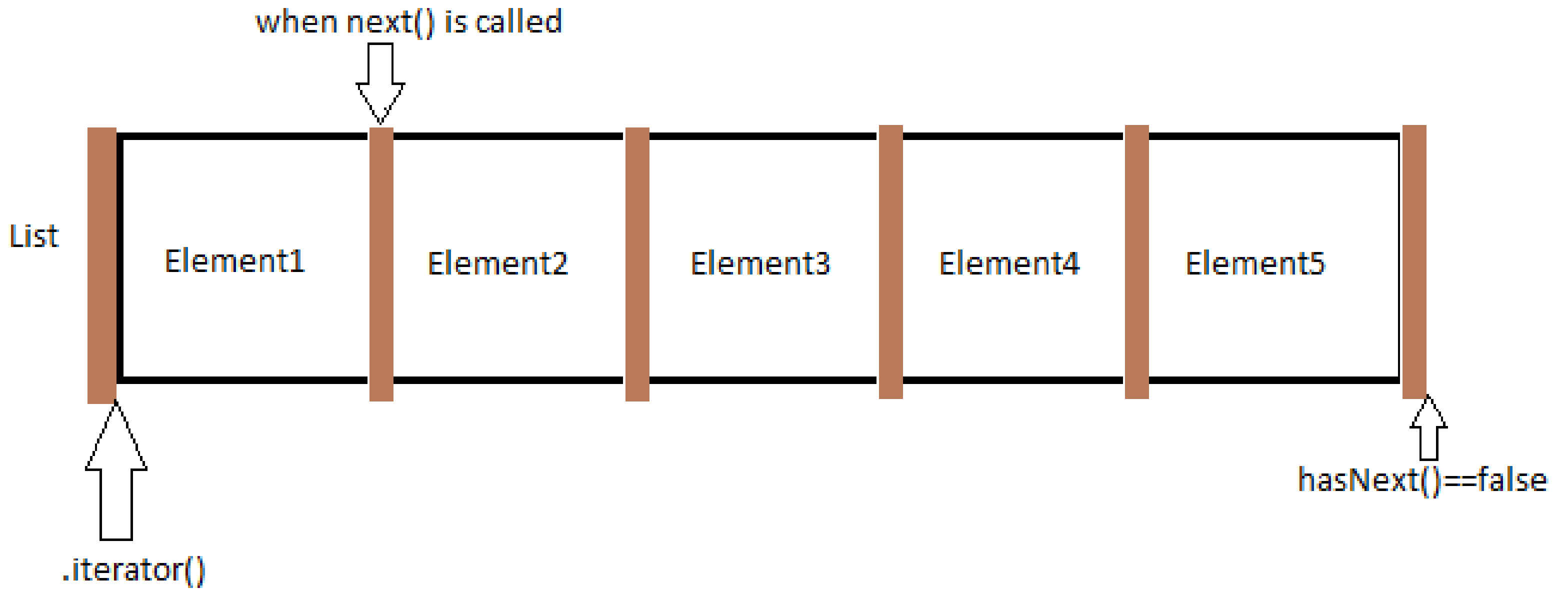
Iterators provide the ability to *iterate* over arbitrary types of data.

```
For all elements of the set
    Perform some action
```

An iterator is method that returns a *generator*. A class can have one or more iterator methods, returning different generators, each allowing you to iterate through in multiple ways.

Iterators

Iterators are used in ***Collection*** Classes in Java to retrieve (the elements of the Collection) one by one.



Collection Classes

A Collection represents a single unit of objects, a group.

The **Collection classes in Java** provide a framework to *store* and *manipulate* objects of a specific group. They provide the operations that can be performed on a specific Collection, such as searching, sorting, insertion, manipulation, and deletion.

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It is comprised of Interfaces and their class implementations, along with the algorithms that can be performed on that collection.

Collection classes includes `ArrayList`, `LinkedList`, `PriorityQueue`, `HashSet`, to name a few.

Collection Classes

A Collection represents a single unit of objects, a group.

The **Collection classes in Java** provide a framework to *store* and *manipulate* objects of a speci group. They provide the operations that can be performed on a specific Collection, such as searchi manipulation, and deleti

Example:

A List, A Set,
A Queue…

The Collection framew ure for storing and manipu comprised of Interfaces ations, along with the algorithms tha b med on that collection.

Collection classes includes `ArrayList`, `LinkedList`, `PriorityQueue`, `HashSet`, to name a few.

Collection Classes

A Collection represents a single unit of objects, a group.

The **Collection classes in Java** provide a framework to *store* and *manipulate* objects of a specific group. They provide the operations that can be performed on a specific Collection, such as *searching*, *sorting*, *insertion*, *manipulation*, and *deletion*.

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It is comprised of Interfaces and their class implementations, along with the algorithms that can be performed on that collection.

Collection classes includes `ArrayList`, `LinkedList`, `PriorityQueue`, `HashSet`, to name a few.

Collection Classes

A Collection represents a single unit of objects, a group.

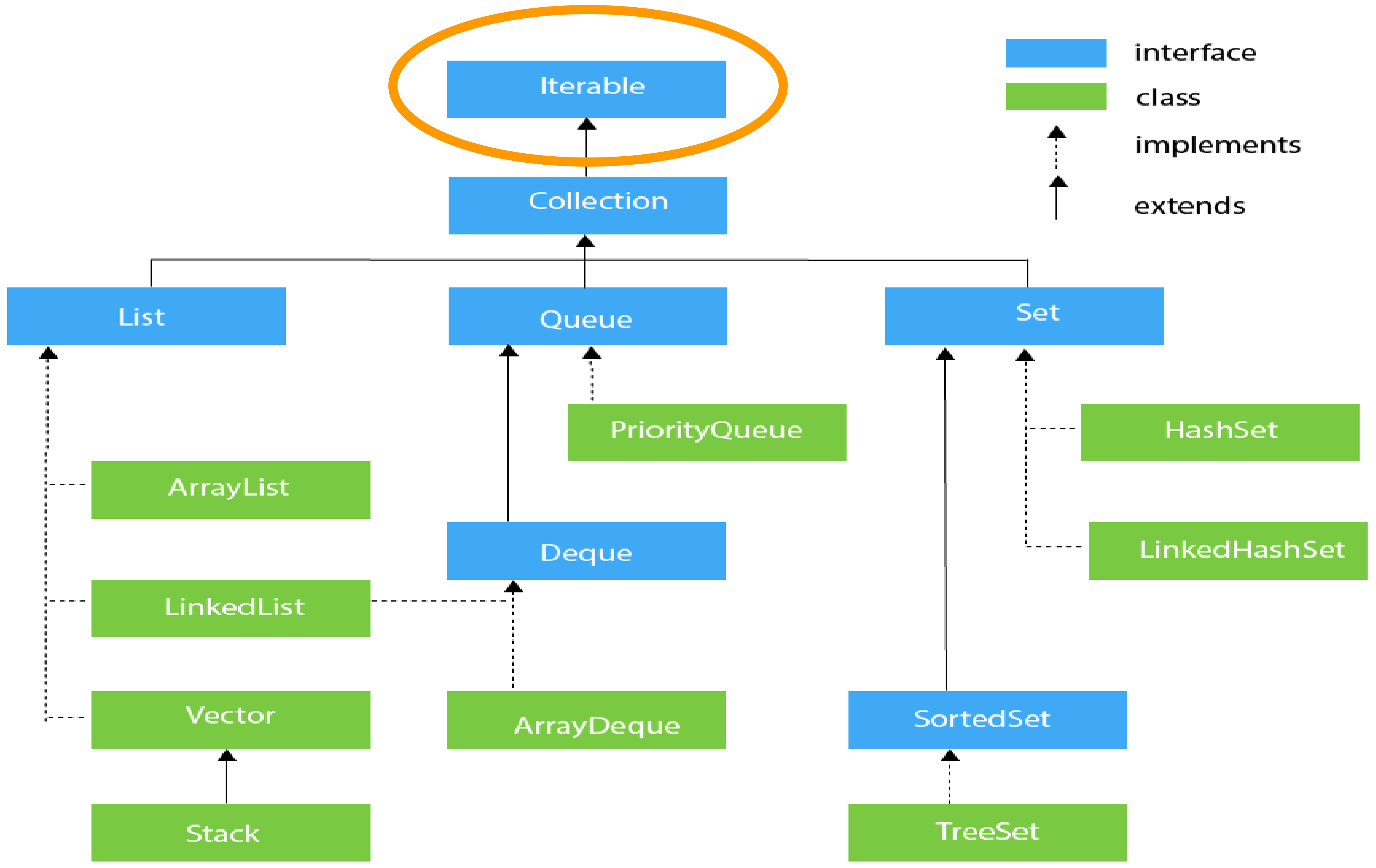
The **Collection classes in Java** provide a framework to *store* and *manipulate* objects of a specific group. They provide the operations that can be performed on a specific Collection, such as searching, sorting, insertion, manipulation, and deletion.

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It is comprised of Interfaces and their class implementations, along with the algorithms that can be performed on that collection.

Collection classes includes `ArrayList`, `LinkedList`, `PriorityQueue`, `HashSet`, to name a few.

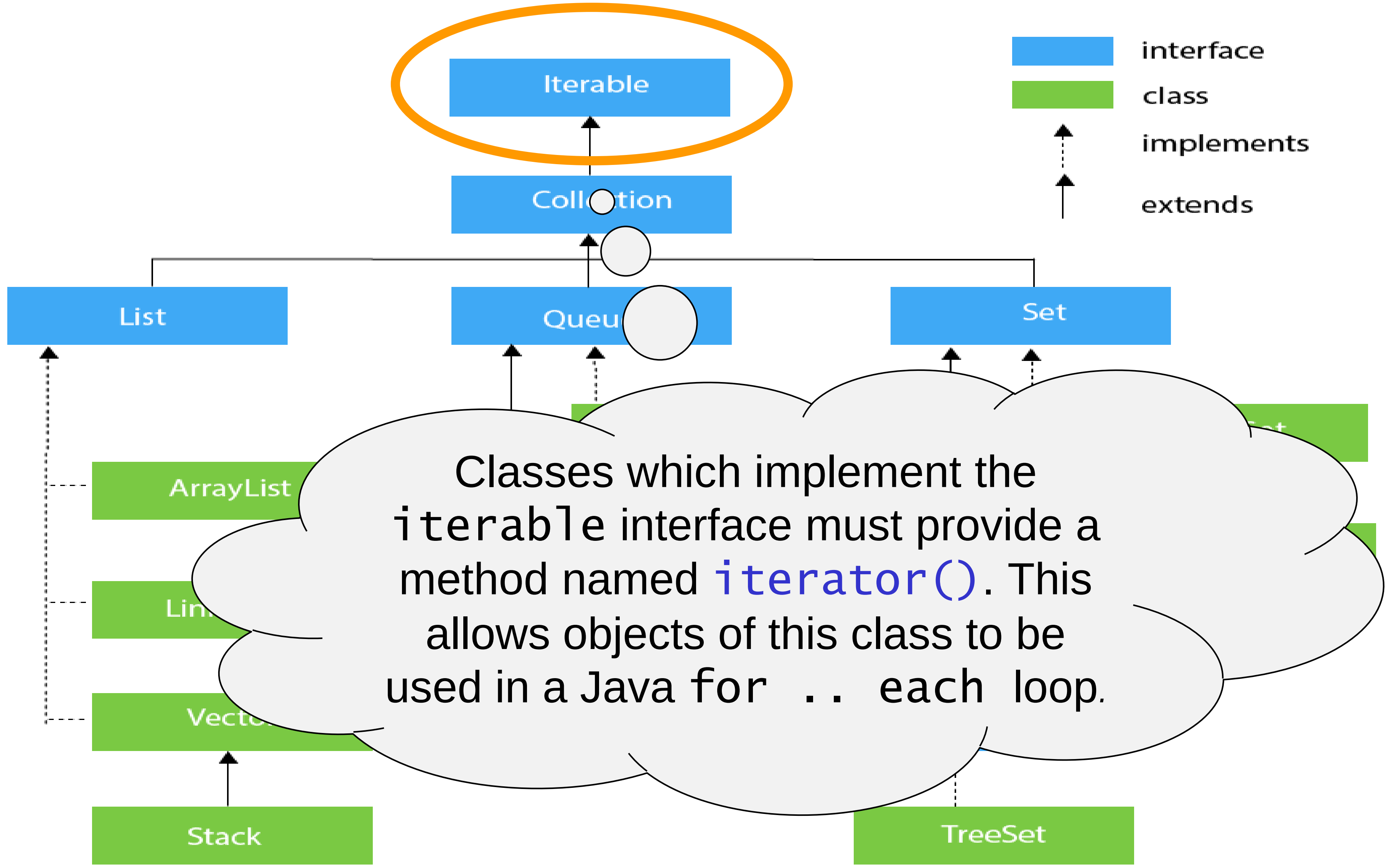
Java Collection Classes

Java Tutorial Point



Java Collection Classes

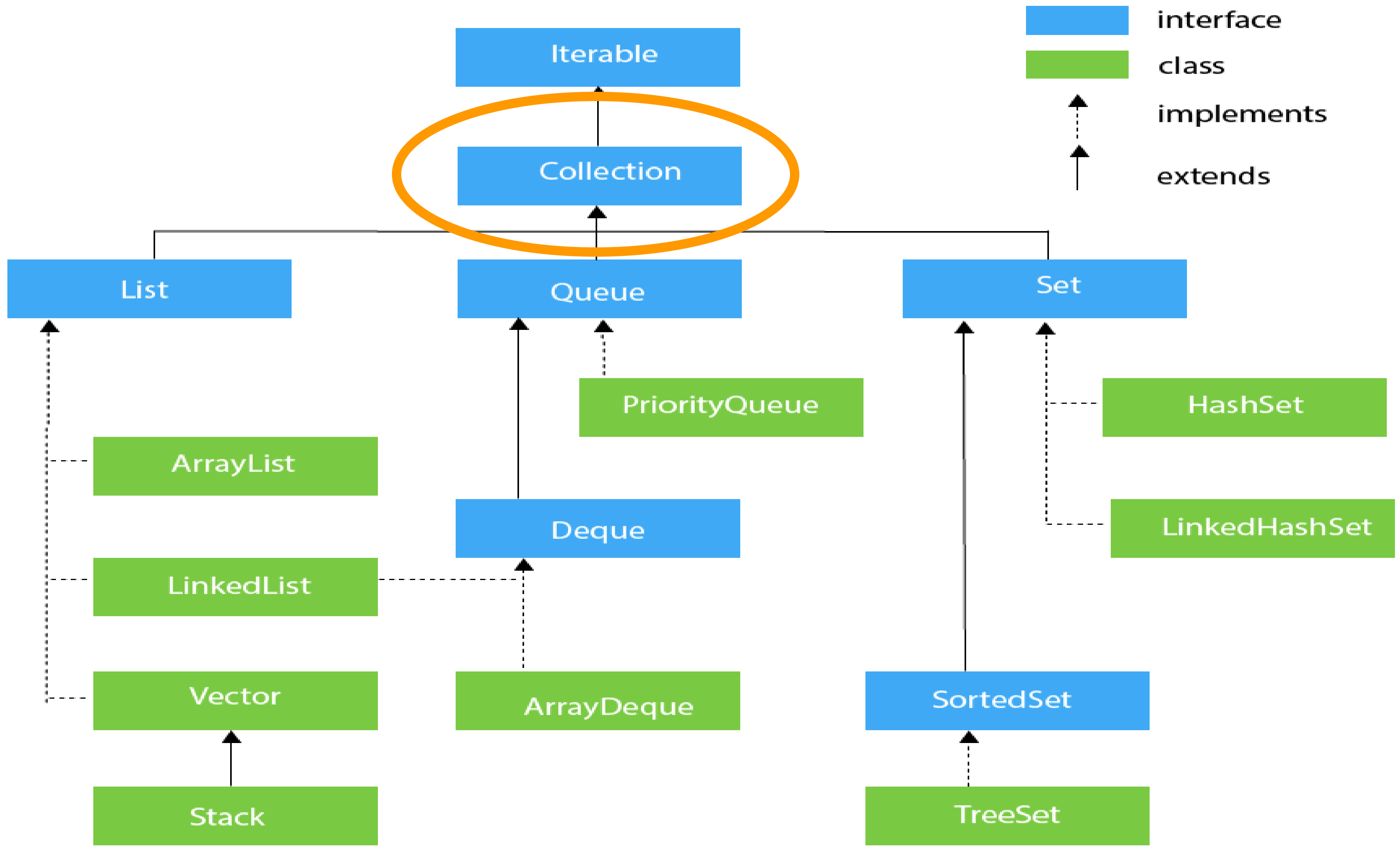
Java Tutorial Point



Classes which implement the `iterable` interface must provide a method named `iterator()`. This allows objects of this class to be used in a Java `for .. each` loop.

Java Collection Classes

Java Tutorial Point



Methods of the Collection Interface

Java Tutorial Point

The Collection interface is the interface which is implemented by all the classes in the collection framework.

public boolean add(E e)	
public boolean addAll(Collection<? extends E> c)	
public boolean remove(Object element)	
public int size()	[illegible] ction.
public void clear()	It removes the total number of elements from the collection.
public boolean contains(Object element)	It is used to search an element.
public boolean containsAll(Collection<?> c)	It is used to search the specified collection in the collection.
public Iterator iterator()	It returns an iterator.
public Object[] toArray()	It converts collection into array.
public boolean equals(Object element)	It matches two collections.
public int hashCode()	It returns the hash code number of the collection.

Methods of the Collection Interface

Java Tutorial Point

The Collection interface builds the foundation on which the collection framework depends. It declares the methods that every collection will have.

public boolean add(E e)	
public boolean addAll(Collection<? extends E> c)	
public boolean remove(Object element)	
public int size()	
public void clear()	ection.
public boolean contains(Object element)	It is used t n element.
public boolean containsAll(Collection<?> c)	It is used to search the specified collection in the collection.
public Iterator iterator()	It returns an iterator.
public Object[] toArray()	It converts collection into array.
public boolean equals(Object element)	It matches two collections.
public int hashCode()	It returns the hash code number of the collection.

Methods of the Collection Interface

Java Tutorial Point

public boolean add(E e)	
public boolean addAll(Collection<? extends E> c)	
public boolean remove(Object element)	
public int size()	
public void clear()	ection.
public boolean contains(Object element)	It is used t n element.
public boolean containsAll(Collection<?> c)	It is used to search the specified collection in the collection.
public Iterator iterator()	It returns an iterator.
public Object[] toArray()	It converts collection into array.
public boolean equals(Object element)	It matches two collections.
public int hashCode()	It returns the hash code number of the collection.

The Collection interface builds the foundation on which the collection framework depends. It declares the methods that every collection will have.

Methods of the Collection Interface

Java Tutorial Point

public boolean add(E e)	It is used to insert an element in this collection.
public boolean addAll(Collection<? extends E> c)	It is used to insert the specified collection elements in the invoking collection.
public boolean remove(Object element)	It is used to delete an element from the collection.
public int size()	It returns the total number of elements in the collection.
public void clear()	It removes the total number of elements from the collection.
public boolean contains(Object element)	It is used to search an element.
public boolean containsAll(Collection<?> c)	It is used to search the specified collection in the collection.
public Iterator iterator()	It returns an iterator.
public Object[] toArray()	It converts collection into array.
public boolean equals(Object element)	It matches two collections.
public int hashCode()	It returns the hash code number of the collection.

Collection Classes

```
public class testClass {
    public static void main( String [] args ) {
        List<String> summer_fruits = new ArrayList<String>();




    }
}
```

Interface *Class*

Collection Classes

```
public class testClass {
    public static void main( String [] args ) {
        List<String> summer_fruits = new ArrayList<String>();

        summer_fruits.add( “figs" );
        summer_fruits.add( "Mango" );
```

Calling the method on an *object* of *ArrayList*

```
    }
}
```

Collection Classes

```
public class testClass {
    public static void main( String [] args ) {
        List<String> summer_fruits = new ArrayList<String>();

        summer_fruits.add( “figs" );
        summer_fruits.add( "Mango" );

        List<String> fruits = new ArrayList<String>();

        fruits.addAll( summer_fruits );


    }
}
```

Calling the method on an *object* *ot* *ArrayList*

CollectionS Class

```
public class testClass {
    public static void main( String [] args ) {
        List<String> summer_fruits = new ArrayList<String>();

        summer_fruits.add( “figs" );
        summer_fruits.add( "Mango" );

        List<String> fruits = new ArrayList<String>();

        fruits.addAll( summer_fruits );

        Collections.addAll(fruits,"Apples","Oranges","Kiwi");

    }
}
```

Calling a static method of the CollectionS class and passing an `object` ot `ArrayList`

CollectionS Class

```
public class testClass {
    public static void main( String [] args ) {
        List<String> summer_fruits = new ArrayList<String>();

        summer_fruits.add( “figs" );
        summer_fruits.add( "Mango" );

        List<String> fruits = new ArrayList<String>();

        fruits.addAll( summer_fruits );

        Collections.addAll(fruits,"Apples","Oranges","Kiwi");

    }
}
```

`Collection` is the interface that Java collection classes implement. `Collections` is a Java utility class that has some set of operations you can perform on a Java Collection.

CollectionS Class

```
public class testClass {
    public static void main( String [] args ) {
        List<String> fruits = new ArrayList<String>();

        Collections.addAll(fruits,”Banana”, “Mango”
                            , "Apples","Oranges","Kiwi");

        for ( String s : fruits )       // element-based loop
            System.out.println( s );




    }
}
```

```
Banana
Mango
Apples
Oranges
Kiwi
```

Note that the elements are output based on the order they were added to the `Collection`.

CollectionS Class

```
public class testClass {
    public static void main( String [] args ) {
        List<String> fruits = new ArrayList<String>();

        Collections.addAll(fruits,”Banana”, “Mango”
                           , "Apples","Oranges","Kiwi");

        for ( String s : fruits )       // element-based loop
            System.out.println( s );

        // What if we wanted to see the collection is some sorted order?



    }
}
```

CollectionS Class

```
public class testClass {
    public static void main( String [] args ) {
        List<String> fruits = new ArrayList<String>();

        Collections.addAll(fruits,”Banana”, “Mango”
                           , "Apples","Oranges","Kiwi");

        for ( String s : fruits )       // element-based loop
            System.out.println( s );

        Collections.sort( fruits ); // Reorder the collection


    }
}
```

How are the elements of the Collection sorted?

CollectionS Class

```
public class testClass {
    public static void main( String [] args ) {
        List<String> fruits = new ArrayList<String>();

        Collections.addAll(fruits,"Banana", "Mango"
                           , "Apple"                  ");

        for ( String s : frui                              loop
            System.out.printl

        Collections.sort( fruits )                   the collection


    }
}
```

The String class implements the Comparable Interface!

CollectionS Class

```
public class testClass {
    public static void main( String [] args ) {
        List<String> fruits = new ArrayList<String>();

        Collections.addAll(fruits,”Banana”, “Mango”
                              , "Apples","Oranges","Kiwi");

        for ( String s : fruits )       // element-based loop
            System.out.println( s );

        Collections.sort( fruits );
        for ( String s : fruits )       // element-based loop
            System.out.println( s );
    }
}
```

```
Apples
Banana
Kiwi
Mango
Oranges
```

Note that the elements are sorted based on their *natural order* in the `Collection`.

CollectionS Class

```
public class testClass {
    public static void main( String [] args ) {
        List<String> fruits = new ArrayList<String>();

        Collections.addAll(fruits,”Banana”, “Mango”
                          , "Apples","Oranges","Kiwi");

        for ( String s : fruits )       // element-based loop
            System.out.println( s );

        Collections.sort( fruits );
        for ( String s : fruits )       // element-based loop
            System.out.println( s );
    }
}
```

```
Apples
Banana
Kiwi
Mango
Oranges
```

The natural order is established by the `compareTo` method.

CollectionS Class

```
public class testClass {
    public static void main( String [] args ) {
        List<String> fruits = new ArrayList<String>();

        Collections.addAll(fruits,”Banana”, “Mango”
                           , "Apples","Oranges","Kiwi");

        for ( String s : fruits )       // element-based loop
            System.out.println( s );

        Collections.sort( fruits );
        for ( String s : fruits )       // element-based loop
            System.out.println( s );
    }
}
```

```
Apples
Banana
Kiwi
Mango
Oranges
```

What if we wanted to bypass the natural order and specify a specific order for our collection?

Comparator Interface

```
public class lengthComparator implements Comparator<String>
{
    public int compare(String s1, String s2){

        return( s1.length() - s2.length() );


    }
} // class
```

CollectionS Class

```
public class testClass {
    public static void
        List<String                                            ();

        Collections.                                          go"
                                                     anges","Kiwi");

        for ( String s : fruits )        // element-based loop
            System.out.println( s );

        Collections.sort( fruits, new lengthComparator() );
        for ( String s : fruits )        // element-based loop
            System.out.println( s );
    }
}
```

Creating an instance of a class for the sole purpose calling a method on that instance.

```
Kiwi
Mango
Apples
Banana
Oranges
```

CollectionS Class

Strategy pattern!

```
public class testClass {
    public static voi
        List<Strin                                         );

        Collections                                  go"
                                               anges","Kiwi");

        for ( String s : fruits )          //  element-based loop
            System.out.println( s );

        Collections.sort( fruits, new lengthComparator() );
        for ( String s : fruits )          // element-based loop
            System.out.println( s );
    }
}
```

```
Kiwi
Mango
Apples
Banana
Oranges
```

Comparator Interface

```
public class lengthComparator implements Comparator<String>
{
    public int compare(String s1, String s2){

        return(s1.length() - s2.length());


    }
} // class
```

```
public class reverselengthComparator implements
Comparator<String>
{
    public int compare(String s1, String s2){

        return(s2.length() - s1.length();


    }
} // class
```

CollectionS Class

```
public class testClass {
    public static void main( String [] args ) {
        List<String> fruits = new ArrayList<String>();

        Collections.addAll(fruits,”Banana”, “Mango”
                           , "Apples","Oranges","Kiwi");

        for ( String s : fruits )       // element-based loop
            System.out.println( s );

        Collections.sort(fruits, new reverselengthComparator());
        for ( String s : fruits )       // element-based loop
            System.out.println( s );
    }
}
```

```
Oranges
Apples
Banana
Mango
Kiwi
```

CollectionS Class

```
public class testClass {
    public static void main( String [] args ) {
        List<String> fruits = new ArrayList<String>();

        Collections.addAll(fruits,”Banana”, “Mango”
                           , "Apples","Oranges","Kiwi");

        for ( String s : fruits )       // element-based loop
            System.out.println( s );

        Collections.sort( fruits );      // natural order
        for ( String s : fruits )       // element-based loop
            System.out.println( s );
    }
}
```

```
Apples
Banana
Kiwi
Mango
Oranges
```

CollectionS Class

```
public class testClass {
    public static void main( String [] args ) {
        List<String> fruits = new ArrayList<String>();

        Collections.addAll(fruits,”Banana”, “Mango”
                          , "Apples","Oranges","Kiwi");

        for ( String s : fruits )       // element-based loop
            System.out.println( s );

        Collections.sort( fruits );
        fruits.forEach(System.out::println); // alternative

    }
}
```

```
Apples
Banana
Kiwi
Mango
Oranges
```

CollectionS Class

```
public class testClass {
    public static void main( String [] args ) {
        List<String> fruits = new ArrayList<String>();

        Collections.addAll(fruits,”Banana”, “Mango”
                           , "Apples","Oranges","Kiwi");

        for ( String s : fruits )       // element-based loop
            System.out.println( s );

        Collections.sort( fruits );
        fruits.forEach(System.out::println);
    }
}
```

```
Apples
Banana
Kiwi
Mango
Oranges
```

Internally the `forEach` method uses an element based `for-each` loop.

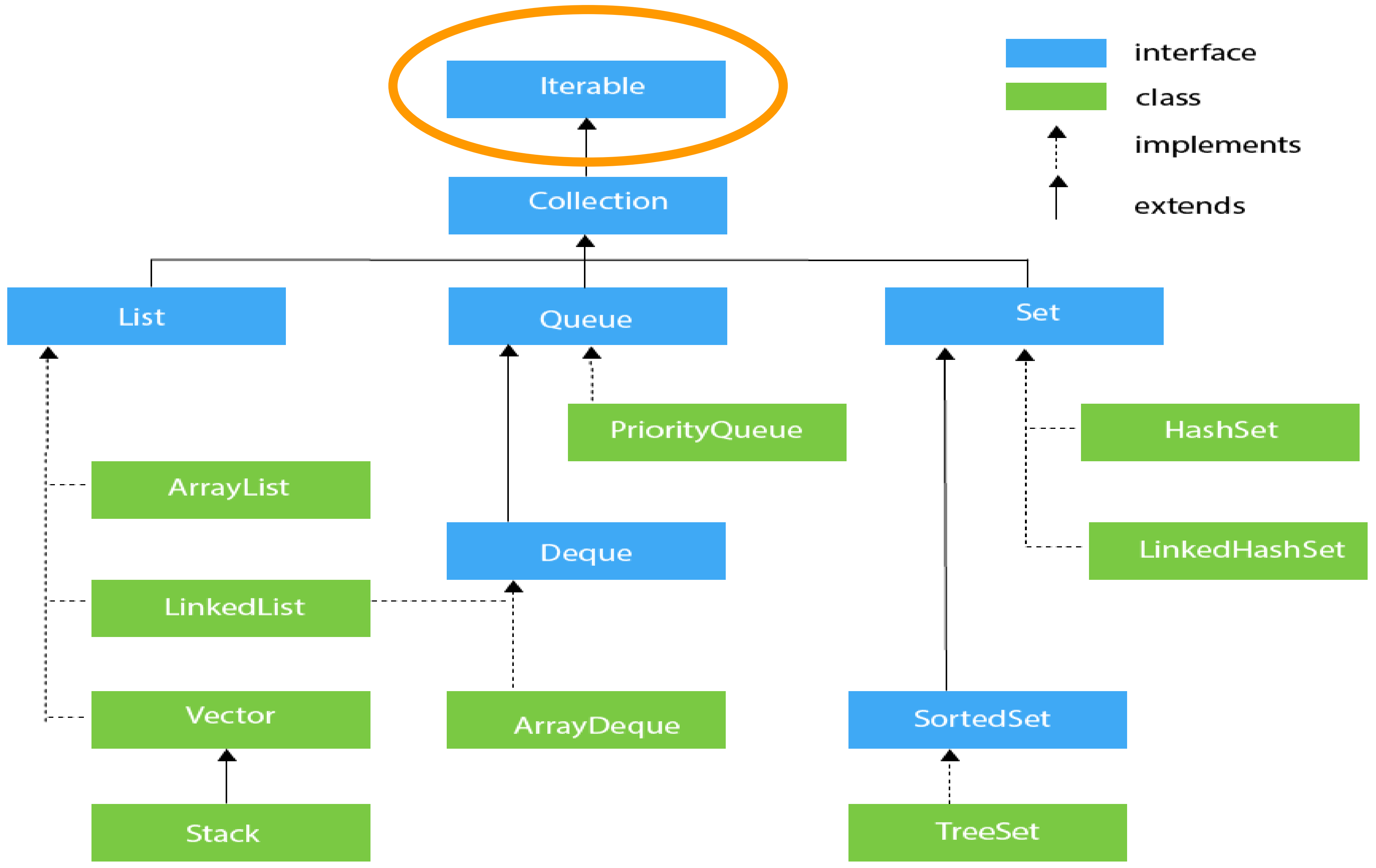
CollectionS Class

```
public class testClass {
    public static void main( String [] args ) {
        List<String> fruits = new ArrayList<String>();

        Collections.addAll(fruits,”Banana”, “Mango”
                                   , "Apples","Oranges","Kiwi");

        for ( String s : fruits )
            System.out.println( s );

        Collections.sort( fruits );
        fruits.forEach(System.out::println);
    }
}
```

```
Apples
Banana
Kiwi
Mango
Oranges
```

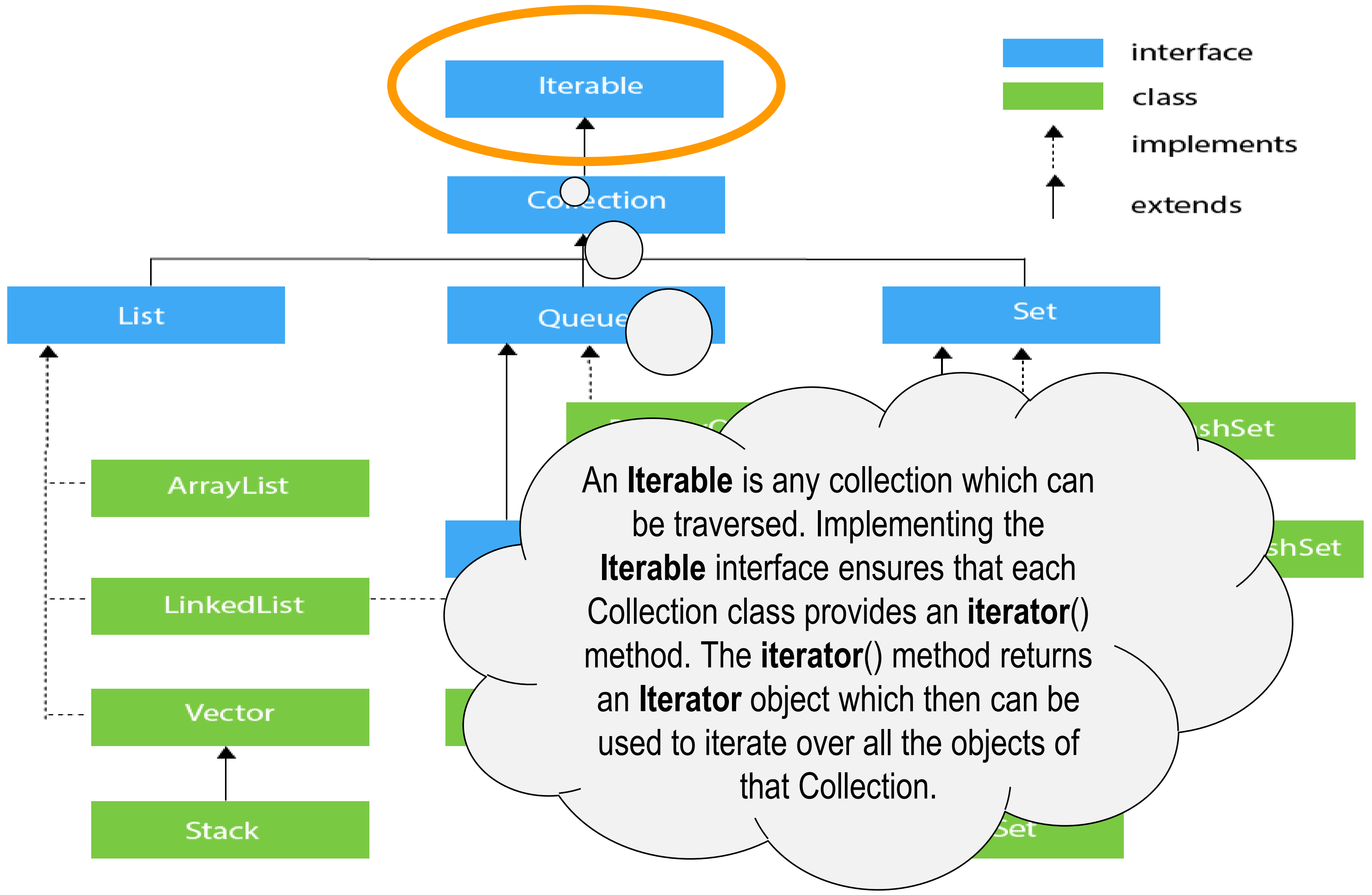
How are the loops actually able to *iterate* over the objects in the `Collection`?

Java Collection Classes



Java Collection Classes

Java Tutorial Point



An **Iterable** is any collection which can be traversed. Implementing the **Iterable** interface ensures that each Collection class provides an **iterator**() method. The **iterator**() method returns an **Iterator** object which then can be used to iterate over all the objects of that Collection.

Methods of the Collection Interface

Java Tutorial Point

public boolean add(E e)	It is used to insert an element in this collection.
public boolean addAll(Collection<? extends E> c)	It is used to insert the specified collection elements in the invoking collection.
public boolean remove(Object element)	It is used to delete an element from the collection.
public int size()	It returns the total number of elements in the collection.
public void clear()	It removes the total number of elements from the collection.
public boolean contains(Object element)	It is used to search an element.
public boolean containsAll(Collection<?> c)	It is used to search the specified collection in the collection.
public Iterator iterator()	It returns an iterator.
public Object[] toArray()	It converts collection into array.
public boolean equals(Object element)	It matches two collections.
public int hashCode()	It returns the hash code number of the collection.

Collection Classes

```
public class testClass {
    public static void main( String [] args ) {
        List<String> fruits = new ArrayList<String>();

        fruits.add( "Banana" );
        fruits.add( "Mango" );
        Collections.addAll(fruits,"Apples"  Oranges","Kiwi");

        for ( String s : fruits )      //
            System.out.println( s

        Collections.sort( fruit

        fruits.forEach(Syste
    }
}
```

```
Apples
Banana
Kiwi
Mango
Oranges
```

Any class that implements the `Iterable` interface must provide an `iterator` method which creates an *`iterator`* object that is then used by the `forEach` loop.

The Iterator Interface

Provides the methods used to traverse the collection!

Using an Iterator

```
public class testClass {
    public static void main( String [] args ) {
        List<String> fruits = new ArrayList<String>();

        fruits.add( "Banana" );
        fruits.add( "Mango" );
        Collections.addAll(fruits,"Apples","Oranges","Kiwi");

        // Invoke the iterator method to create the iterator!
        Iterator itr = fruits.iterator();

        // check for availability of the next element
        while (itr.hasNext())
           // return the element at the current position and
           // move the cursor to next element
          System.out.println( (String) itr.next() );



    }
}
```

The Iterator Interface

The Iterator interface provides the facility to create an iterator object which is used to *traverse* over the elements in the Collection, but in a forward direction only.

Method	Description
public boolean hasNext()	It returns true if the iterator has more elements otherwise it returns false.
public Object next()	It returns the element and moves the cursor pointer to the next element.
public void remove()	It removes the last elements returned by the iterator. It is less used.

The Iterator Interface

The Iterator interface provides the facility to create an iterator object which is used to *traverse* over the elements in the Collection, but in a forward direction only.

Method	escriptio
public boolean hasNext()	
public Object next()	
public void remove()	ed by the d.

Known as the Universal iterator because it is the core interface of all the Collection classes.

```
public class TestIterator
{
    public static void main(String[] args) {
        // Create an array list
        ArrayList al = new ArrayList();

        // Add the numbers 0 .. 9 to the list
        for (int i = 0; i < 10; i++)
            al.add(i);

        // at beginning itr(cursor) will point to
        // index just before the first element in al
        Iterator itr = al.iterator();

        // check for availability of the next element
        while (itr.hasNext()) {
           // return the element at the current position and
           // move the cursor to next element
           int i = (int)itr.next();




        }

    } // main

} // class
```

```
public class TestIterator
{
    public static void main(String[] args) {
        // Create an array list
        ArrayList al = new ArrayList();

        // Add the numbers 0 .. 9 to the list
        for (int i = 0; i < 10; i++)
            al.add(i);

        // at beginnin  itr(cursor) will point to
        // index just befo   the first element in al
        Iterator itr = al.    ator();

        // check for availabi
        while (itr.hasNext()
           // return the el
           // move the cu
           int i = (int)




        }

    } // main

} // class
```

Java implements a feature called auto-boxing which creates an instance of the appropriate wrapper class!

```
public class TestIterator
{
    public static void main(String[] args) {
        // Create an array list
        ArrayList al = new ArrayList();

        // Add the numbers 0 .. 9 to the list
        for (int i = 0; i < 10; i++)
            al.add(i);

        // at beginning itr(cursor) will point to
        // index just before the first element in al
        Iterator itr = al.iterator();

        // check for availability of the next element
        while (itr.hasNext()) {
           // return the element at the current position and
           // move the cursor to next element
           int i = (int)itr.next();




        }

    } // main

} // class
```

```
public class TestIterator
{
    public static void main(String[] args) {
        // Create an array list
        ArrayList al = new ArrayList();

        // Add the numbers 0 .. 9 to the list
        for (int i = 0; i < 10; i++)
            al.add(i);

        // at beginning itr(cursor) will point to
        // index just before the first element in al
        Iterator itr = al.iterator();

        // check for availability of the next element
        while (itr.hasNext()) {
           // return the element at the current position and
           // move the cursor to next element
          int i = (int)itr.next();




        }

    } // main

} // class
```

```
public class TestIterator
{
    public static void main(String[] args) {
        // Create an array list
        ArrayList al = new ArrayList();

        // Add the numbers 0 .. 9 to the list
        for (int i = 0; i < 10; i++)
            al.add(i);

        // at beginning itr(cursor) will point to
        // index just before the first element in al
        Iterator itr = al.iterator();

        // check for availability of the next element
        while (itr.hasNext()) {
           // return the element at the current position and
           // move the cursor to next element
           int i = (int)itr.next();

           // Can even remove elements while iterating
           if (i % 2 != 0)
              itr.remove();
        }

    } // main

} // class
```

ListIterator Interface **extends** Iterator

The List Iterator interface provides the facility of iterating over List style collection classes that provides bi-directional iteration.

```
ListIterator ltr = l.listIterator();
```

There are two additional methods that are provided by the ListIterator Interface:

Method	Description
public boolean hasPrevious()	It returns true if the iterator has more elements while traversing backward otherwise it returns false.
public **Object** previous()	It returns the previous element in the iteration and moves the cursor pointer to the next previous element.

Iterator vs. ListIterator
summary

- The basic difference between Iterator and ListIterator is that the Iterator can traverse elements in a collection only in forward direction. On the other hand, the ListIterator can traverse in both forward and backward directions.
- Using iterator you can not add any element to a collection. But, by using ListIterator you can add elements to a collection.
- Using Iterator, you can not remove an element in a collection where, as you can remove an element from a collection using ListIterator.
- Using Iterator you can traverse all collections like *Map*, *List*, *Set*. But, by ListIteror you can traverse List implemented objects only.

Implementing the List Iterator Interface

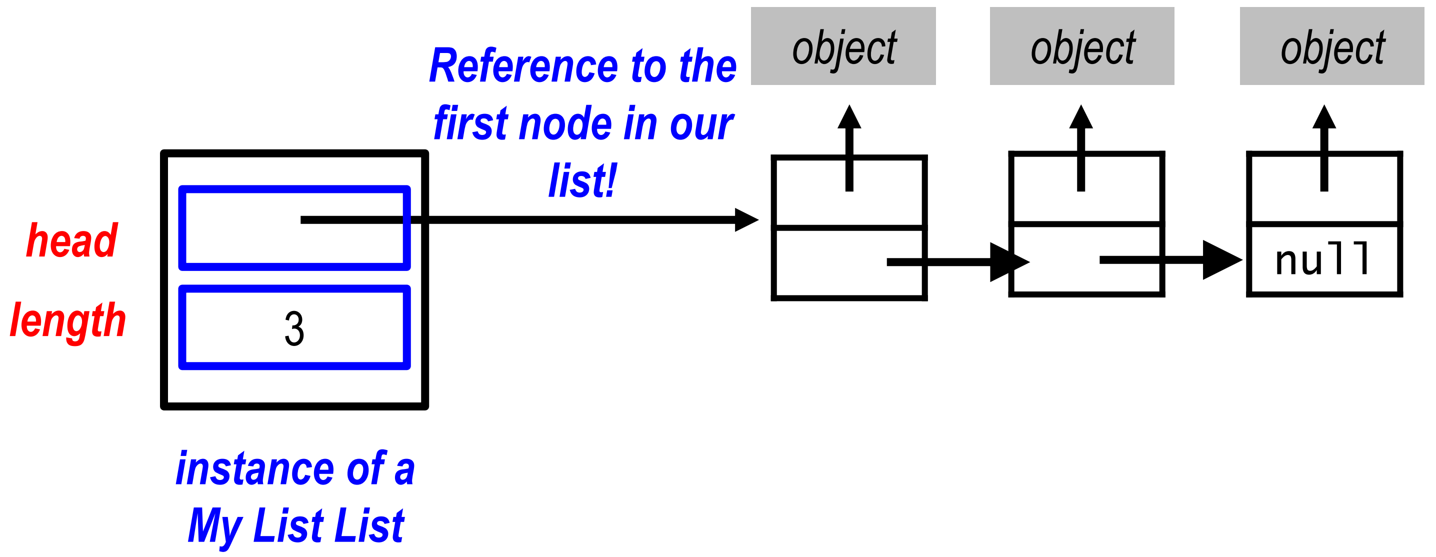
- Here again, the interface only includes the method headers:

```
public interface ListIterator { // in ListIterator.java
    boolean hasNext();
    Object next();
}
```

- We can then implement this interface for our own list:
  - `Assume a class MyList that simulates a linked lists`

MyList Class

- Implementing the List interface with a **Linked List**


Reference to the first node in our list!
object
object
object
head
length
3
null
instance of a My List List

A Linked List Class

```
public class MyList implements List {
    private Node head;
    private int length;









    ...
```

A Linked List class

```
public class MyList implements List, Iterable {
    private class Node {
        private Object item;
        private Node next;

        private Node() {
            next = null;
        }
    }
    private Node head;
    private int length;

    public MyList() {
        head = null; length = 0;
    }
    public boolean add( Object item ) { ... };
    public Object remove( Object item ) { ... }; etc.

    public ListIterator iterator() {
        return new MyListIterator();
    }

}
```

A Linked List class

```
public class MyList implements List, Iterable {
    private class Node {
        private Object item;
        private Node next;

        private Node() {
            next = null;
        }
    }
    private Node head;
    private int length;

    public MyList() {
        head = null; length = 0;
    }
    public boolean add( Object item ) { ... };
    public Object remove( Object item ) { ... }; etc.

    public ListIterator iterator() {
        return new MyListIterator();
    }

}
```

What do we need to iterate over?

A Linked List class

```
public class MyList implements List, Iterable {
    private class Node {
        private Object item;
        private Node next;

        private Node() {
            next = null;
        }
    }
    private Node head;
    private int length;

    public MyList() {
        head = null; length = 0;
    }
    public boolean add( Object item ) { ... };
    public Object remove( Object item ) { ... }; etc.

    public ListIterator iterator() {
        return new MyListIterator();
    }

}
```

Nodes!

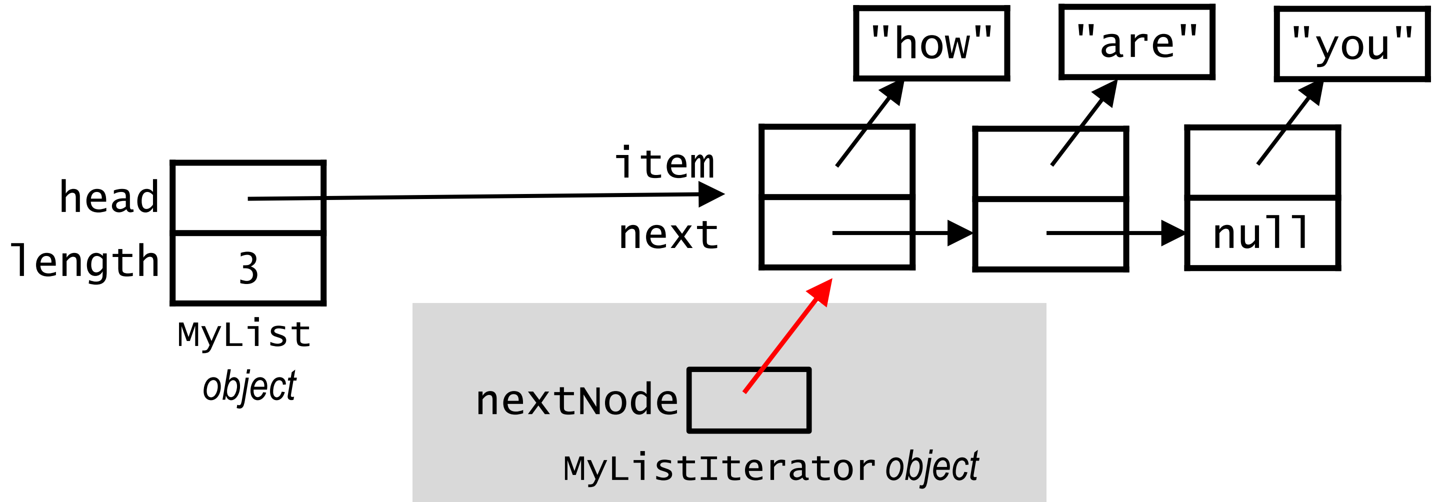
An Inner Class for the Iterator

```
public class MyList … {
    private Node head;
    private int length;

    private class MyListIterator implements ListIterator {
        private Node nextNode;  // points to node with the next item

        public MyListIterator() {
            nextNode = head;
        }
        ...
    }

    public ListIterator iterator() {
        return new MyListIterator();
    }
    ...
```

Making this an inner class gives this class access to all the private data members of the MyList class!

Full `LLListIterator` Implementation

```
private class MyListIterator implements ListIterator {
    private Node nextNode;     // points to node with the next item

    public MyListIterator() {
        nextNode = head;
    }

    public boolean hasNext() {
        return (nextNode != null);
    }

    public Object next() {
        // throw an exception if nextNode is null

        Object item = nextNode.item;
        nextNode = nextNode.next;

        return item;
    }
}
```



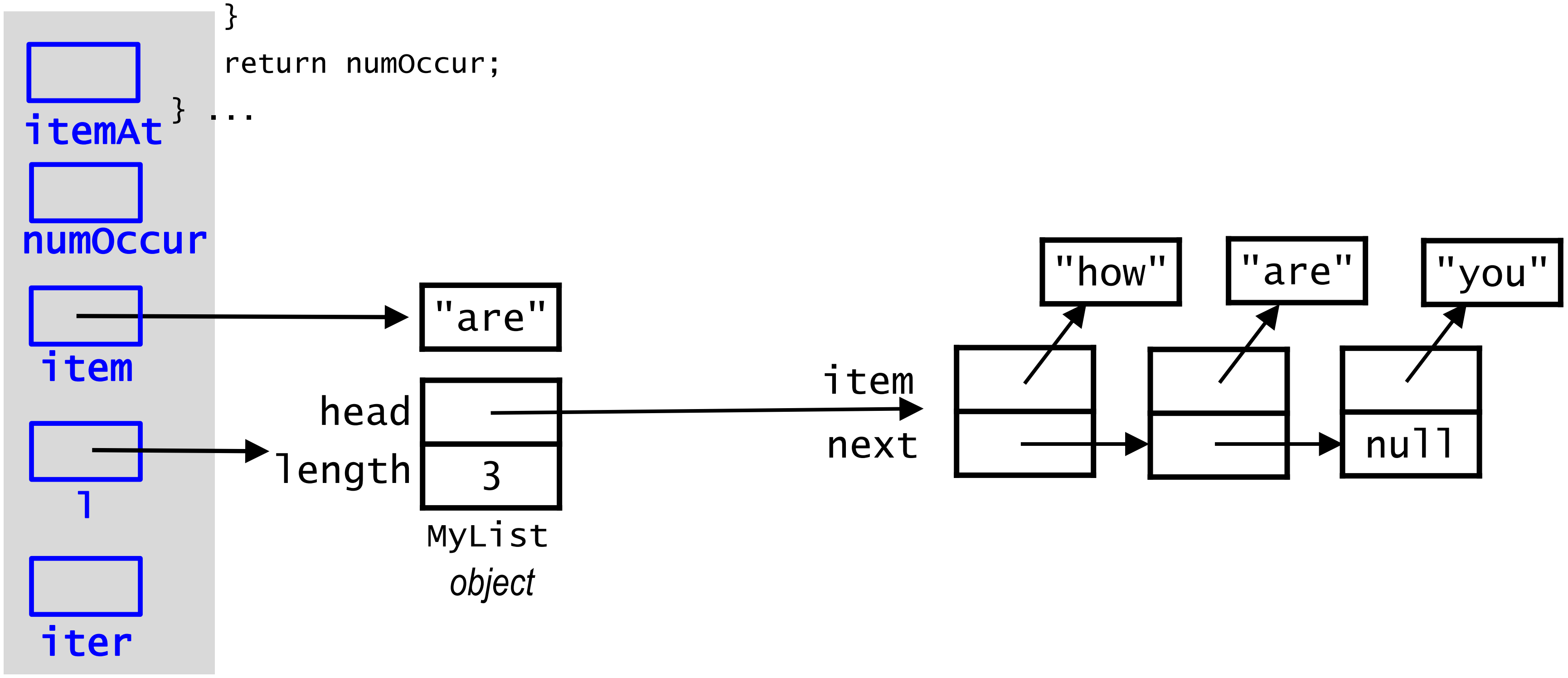
An Interface for List Iterators:
summary

Once the iterator interface has been implemented, we can create an instance of it and use it to externally traverse the list - regardless of the specific implementation of the List:

```
public class MyClass {

   public static int numOccur(List l, Object item) {

      int numOccur = 0;
      ListIterator iter = l.iterator();
      while ( iter.hasNext() ) {
         Object itemAt = iter.next();

         if (itemAt.equals(item)) {
            numOccur++;
         }
      }
      return numOccur;
   } ...
```

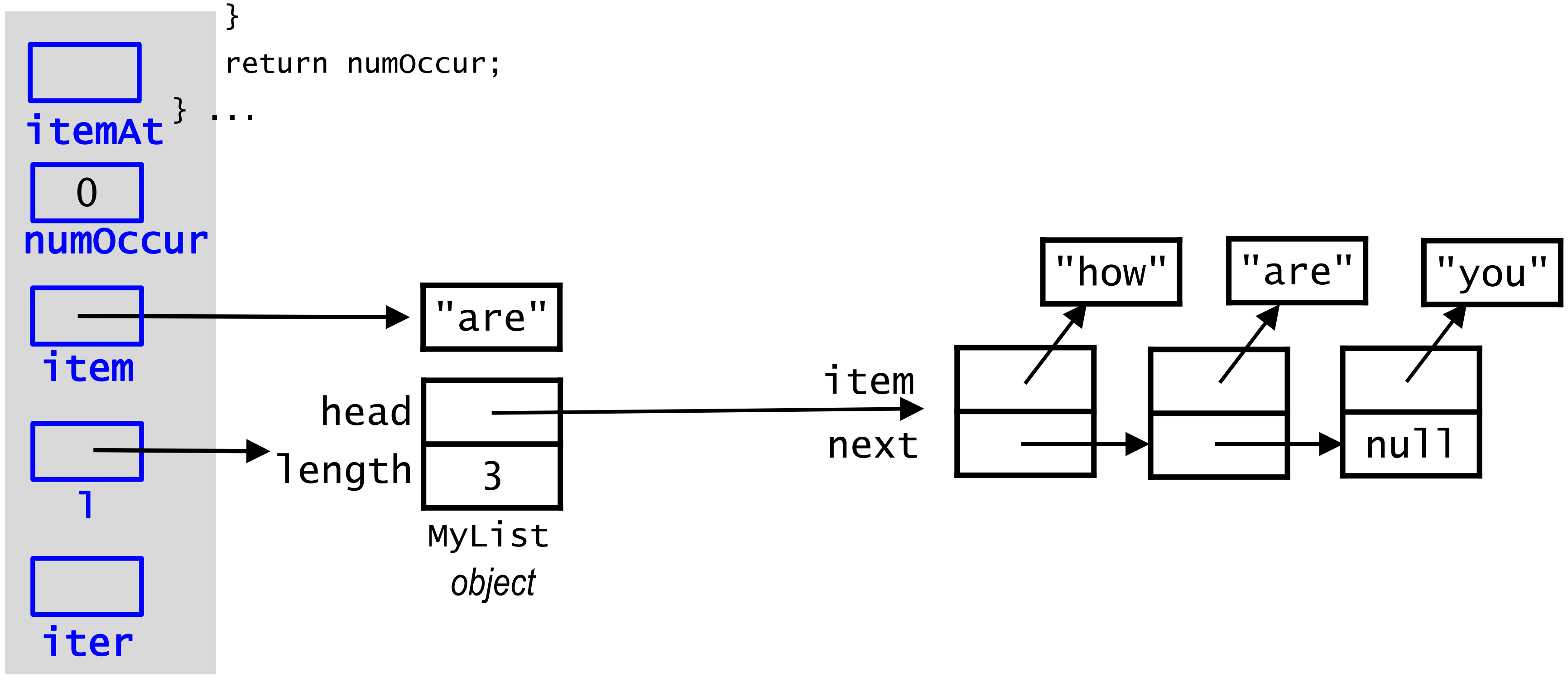
Example: *MyList* list

```
public class MyClass {

    public static int numOccur(List l, Object item) {

      int numOccur = 0;
      ListIterator iter = l.iterator();
       while ( iter.hasNext() ) {
           Object itemAt = iter.next();
           if (itemAt.equals(item)) {
               numOccur++;
           }
       }
       return numOccur;
    } ...
```



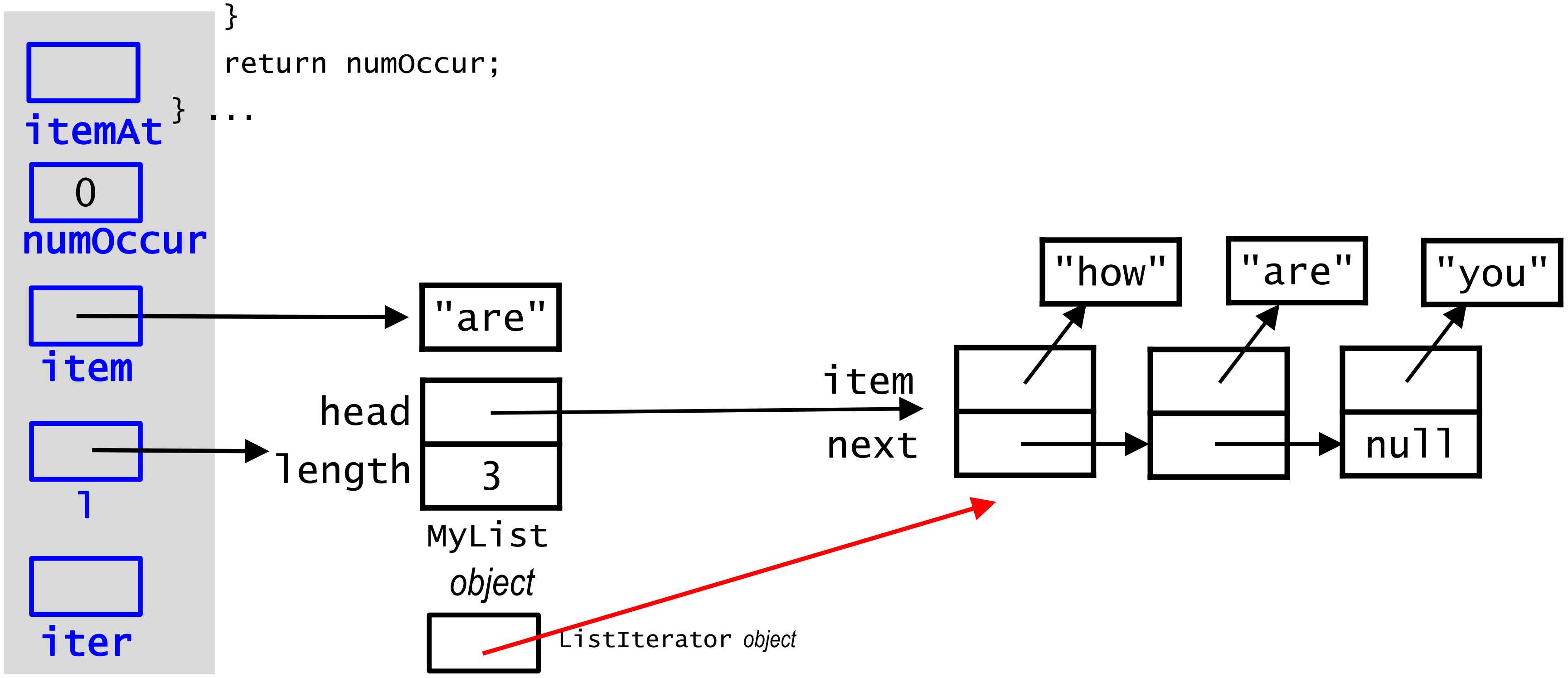
Example: *MyList* list

```
public class MyClass {

    public static int numOccur(List l, Object item) {

      int numOccur = 0;
      ListIterator iter = l.iterator();
       while ( iter.hasNext() ) {
           Object itemAt = iter.next();
           if (itemAt.equals(item)) {
               numOccur++;
           }
      }
      return numOccur;
    } ...
```



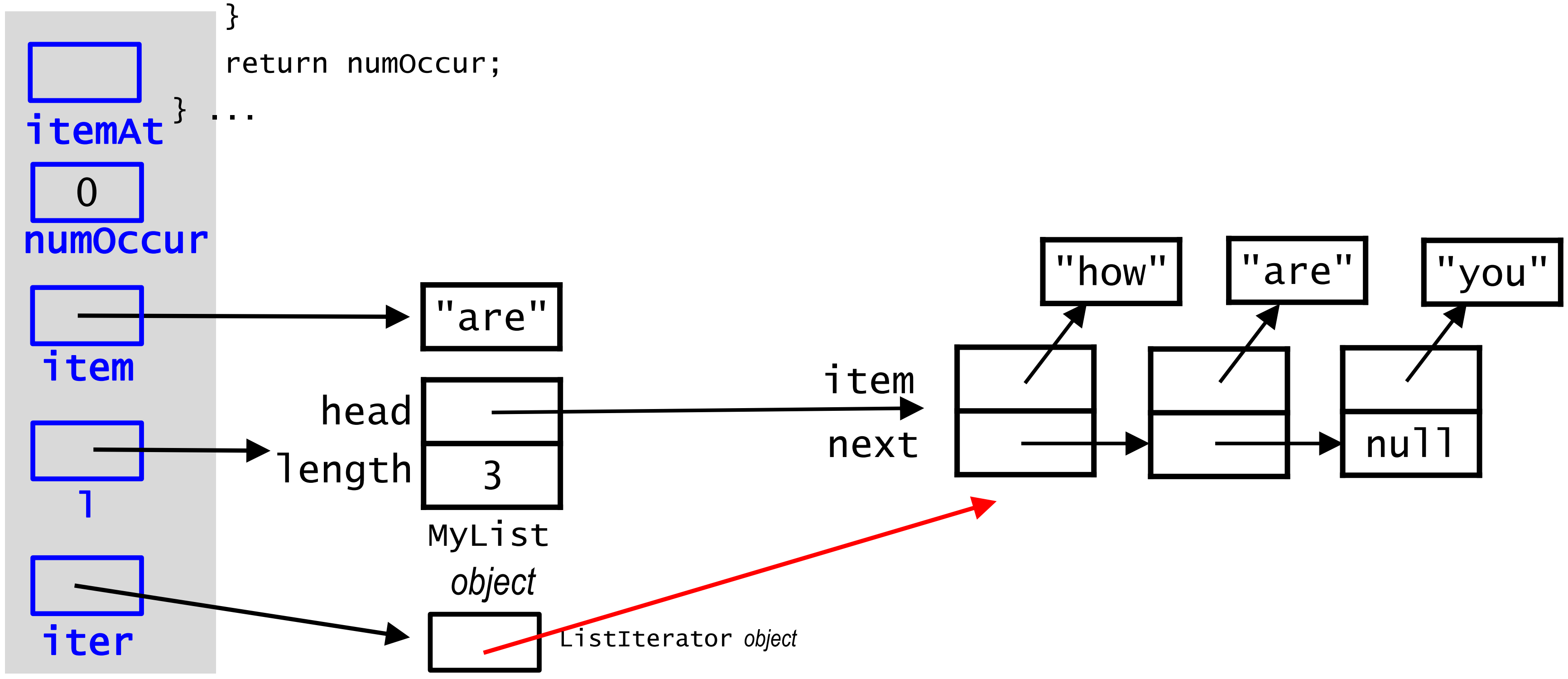
Example: *MyList* list

```
public class MyClass {

    public static int numOccur(List l, Object item) {

       int numOccur = 0;
       ListIterator iter = l.iterator();
        while ( iter.hasNext() ) {
            Object itemAt = iter.next();
            if (itemAt.equals(item)) {
                numOccur++;
            }
        }
        return numOccur;
    } ...
```



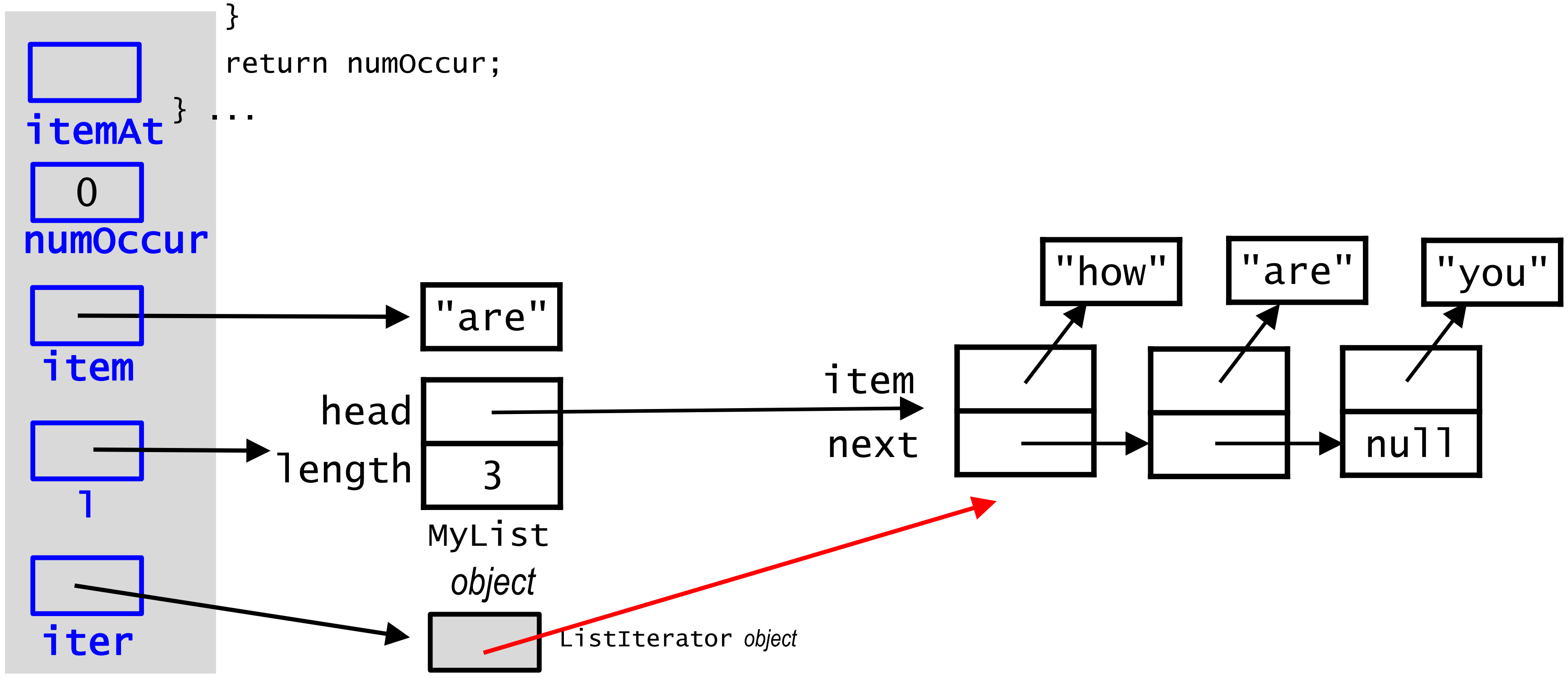
Example: *MyList* list

```
public class MyClass {

    public static int numOccur(List l, Object item) {

      int numOccur = 0;
      ListIterator iter = l.iterator();
       while ( iter.hasNext() ) {
           Object itemAt = iter.next();
           if (itemAt.equals(item)) {
               numOccur++;
           }
       }
       return numOccur;
    } ...
```



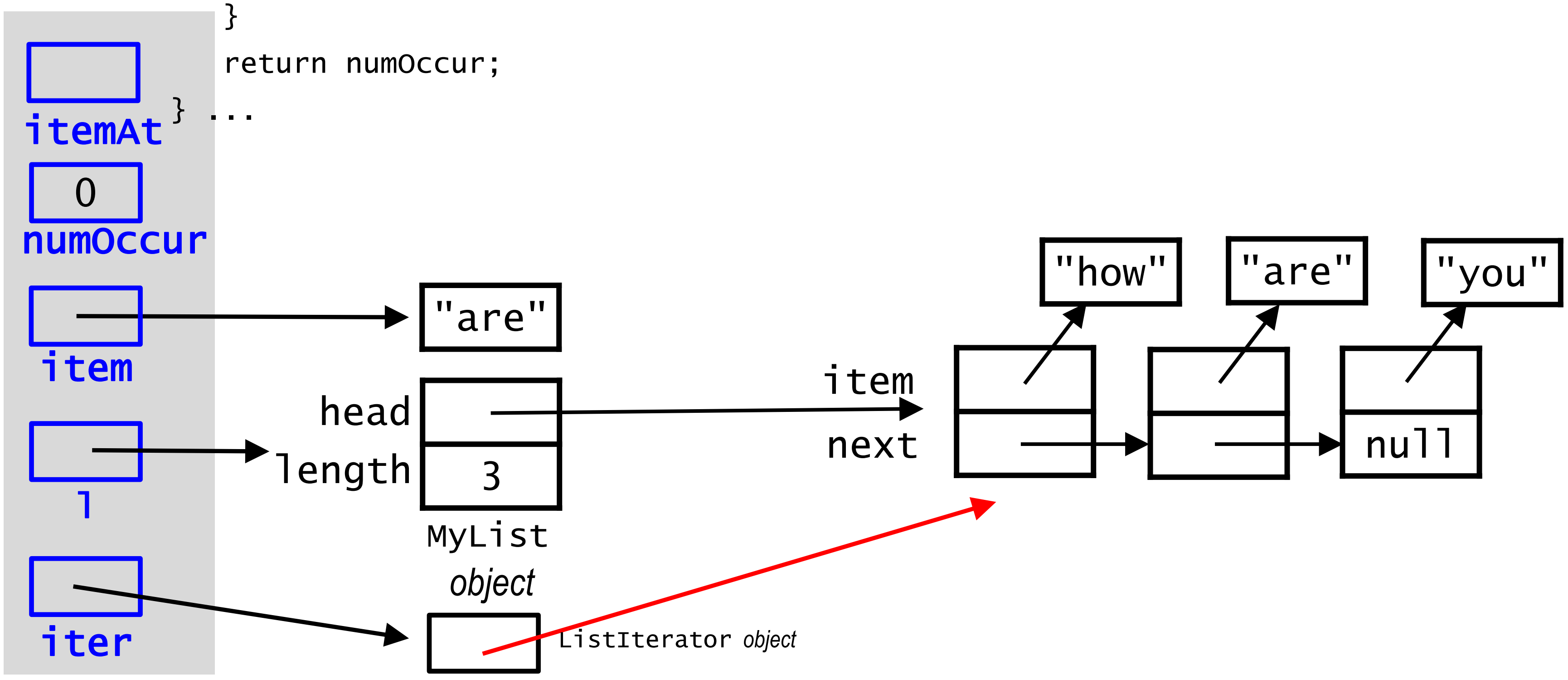
Example: *MyList* list

```
public class MyClass {

   public static int numOccur(List l, Object item) {

     int numOccur = 0;
     ListIterator iter = l.iterator();
      while ( iter.hasNext() ) {
          Object itemAt = iter.next();
          if (itemAt.equals(item)) {
              numOccur++;
          }
      }
      return numOccur;
   } ...
```



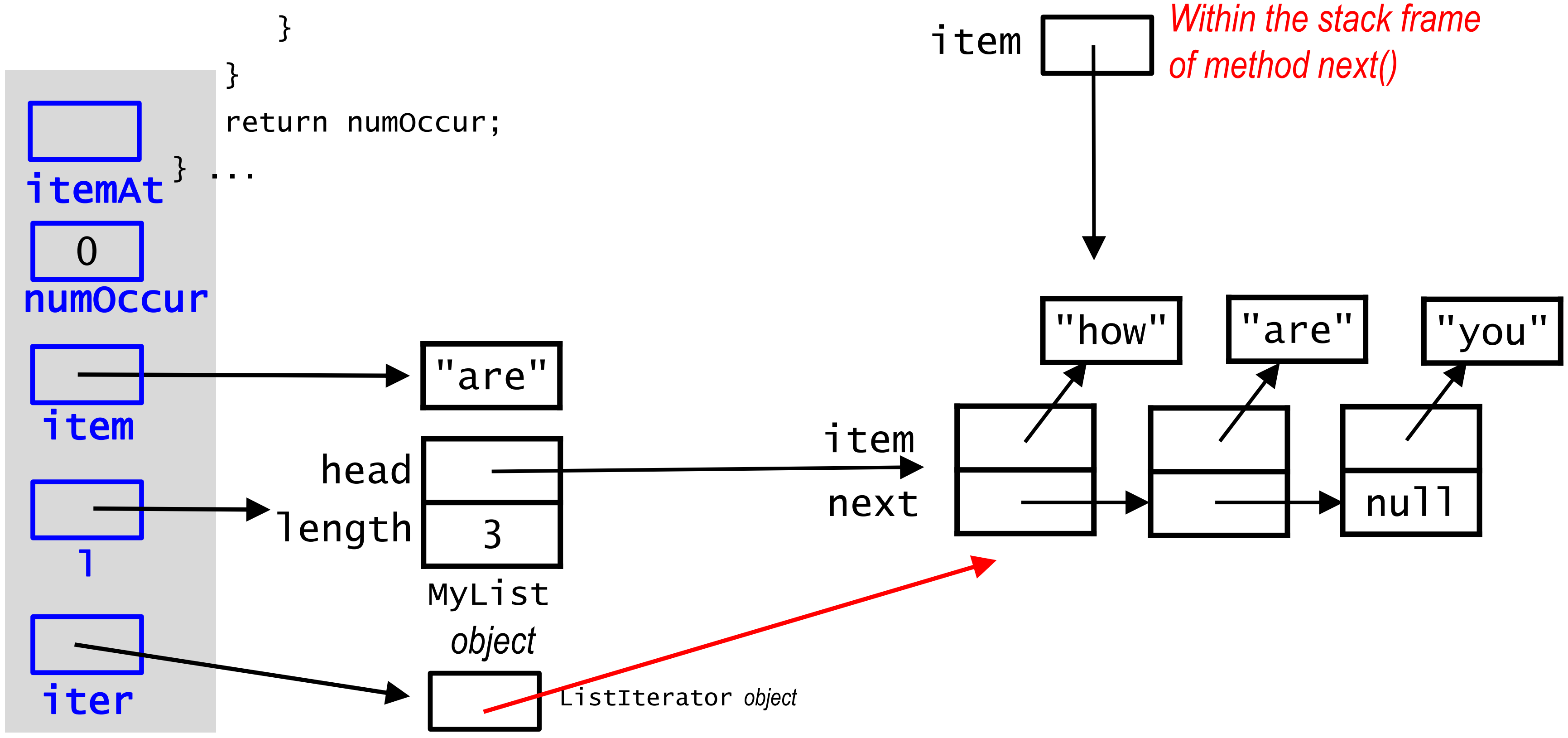
Example: *MyList* list

```
public class MyClass {

    public static int numOccur(List l, Object item) {

       int numOccur = 0;
       ListIterator iter = l.iterator();
        while ( iter.hasNext() ) {
            Object itemAt = iter.next();
            if (itemAt.equals(item)) {
                numOccur++;
            }
        }
        return numOccur;
    } ...
```



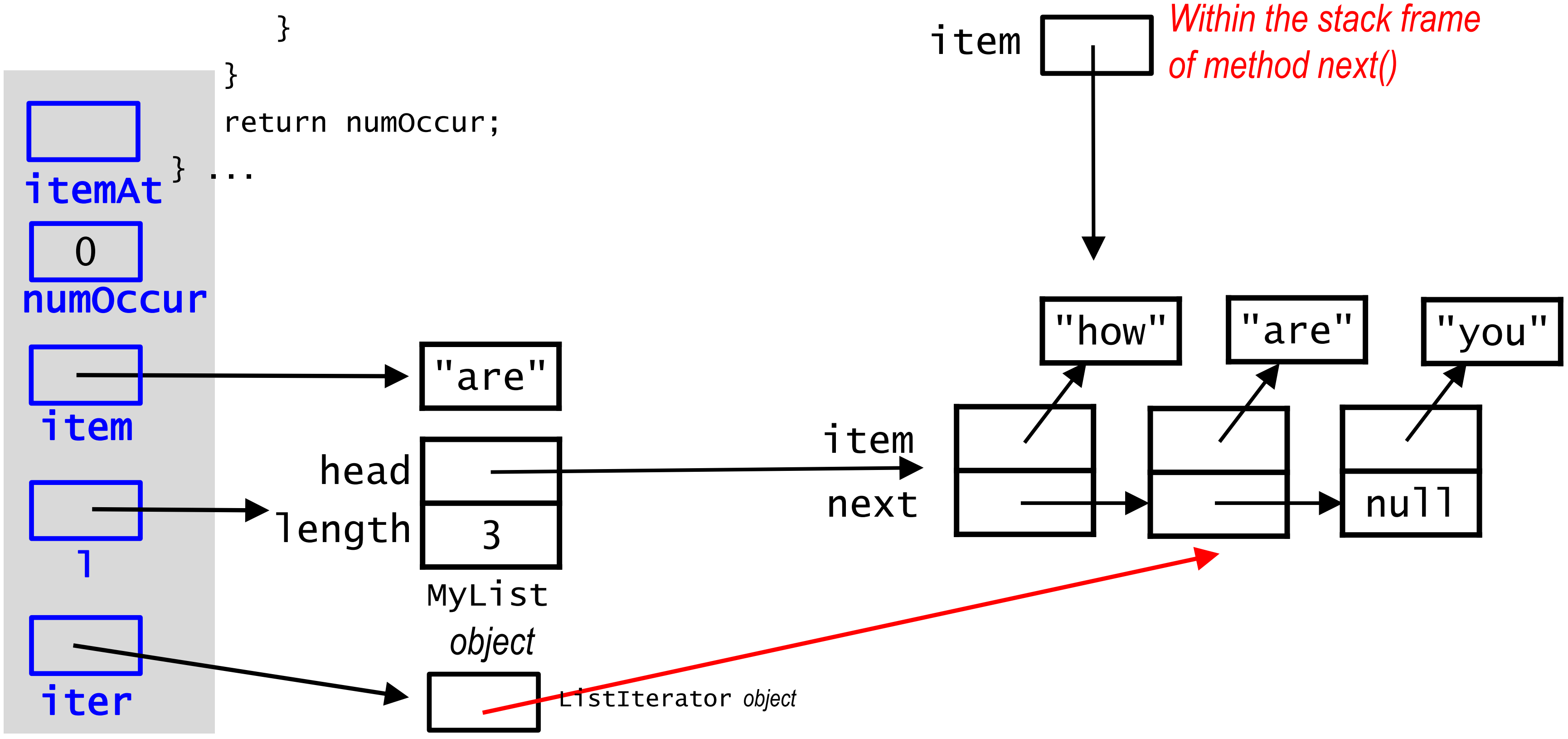
Example: *MyList* list

```
public class MyClass {

    public static int numOccur(List l, Object item) {

      int numOccur = 0;
      ListIterator iter = l.iterator();
       while ( iter.hasNext() ) {
           Object itemAt = iter.next();
           if (itemAt.equals(item)) {
               numOccur++;
           }
       }
       return numOccur;
    } ...
```



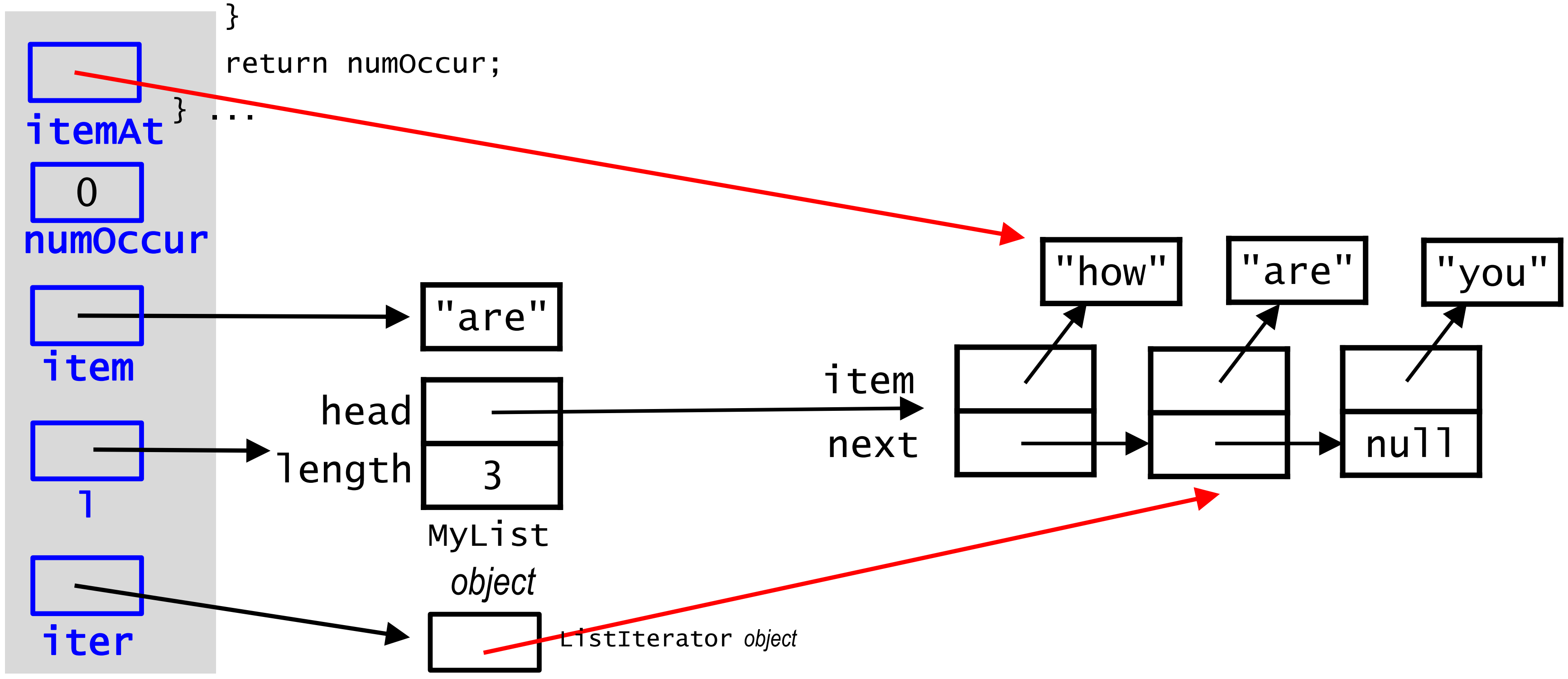
Example: *MyList* list

```
public class MyClass {

    public static int numOccur(List l, Object item) {

       int numOccur = 0;
       ListIterator iter = l.iterator();
        while ( iter.hasNext() ) {
            Object itemAt = iter.next();
            if (itemAt.equals(item)) {
                numOccur++;
            }
        }
        return numOccur;
    } ...
```



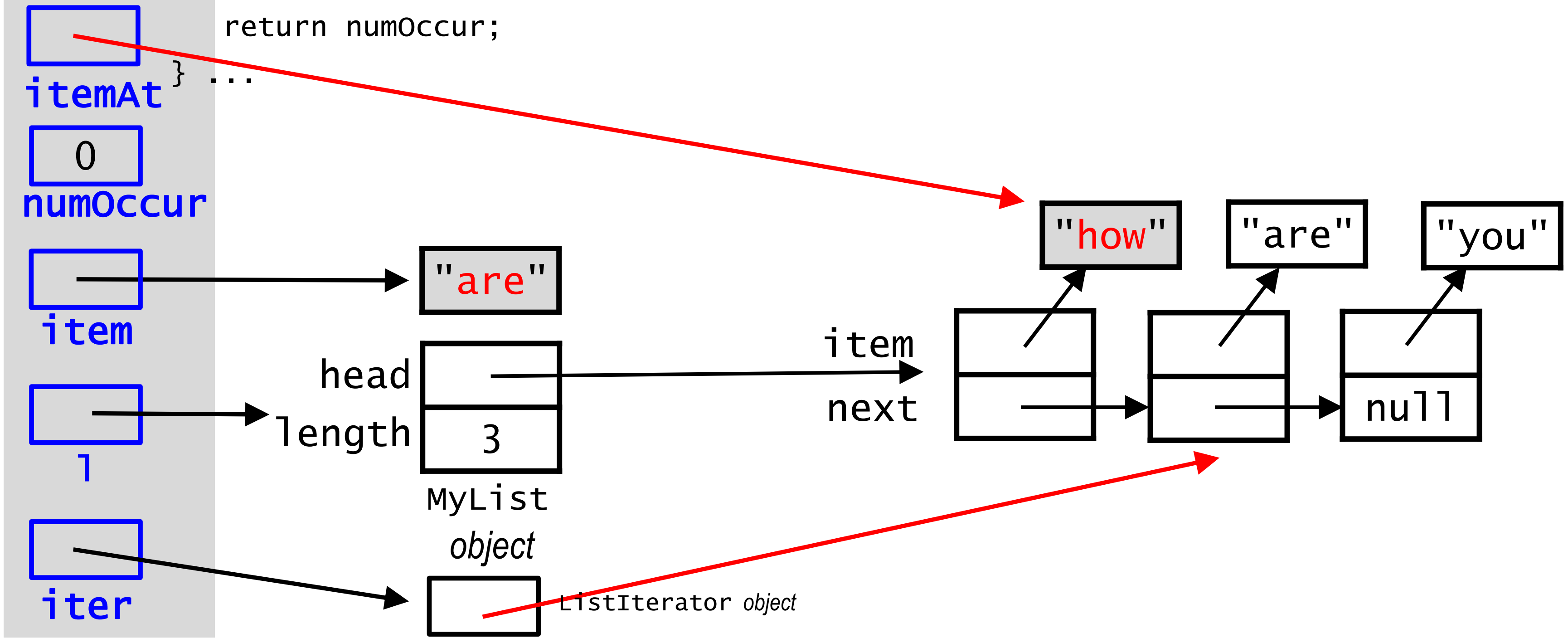
Example: *MyList* list

```
public class MyClass {

    public static int numOccur(List l, Object item) {

       int numOccur = 0;
       ListIterator iter = l.iterator();
        while ( iter.hasNext() ) {
            Object itemAt = iter.next();
            if (itemAt.equals(item)) {
                numOccur++;
            }
        }
        return numOccur;
    } ...
```



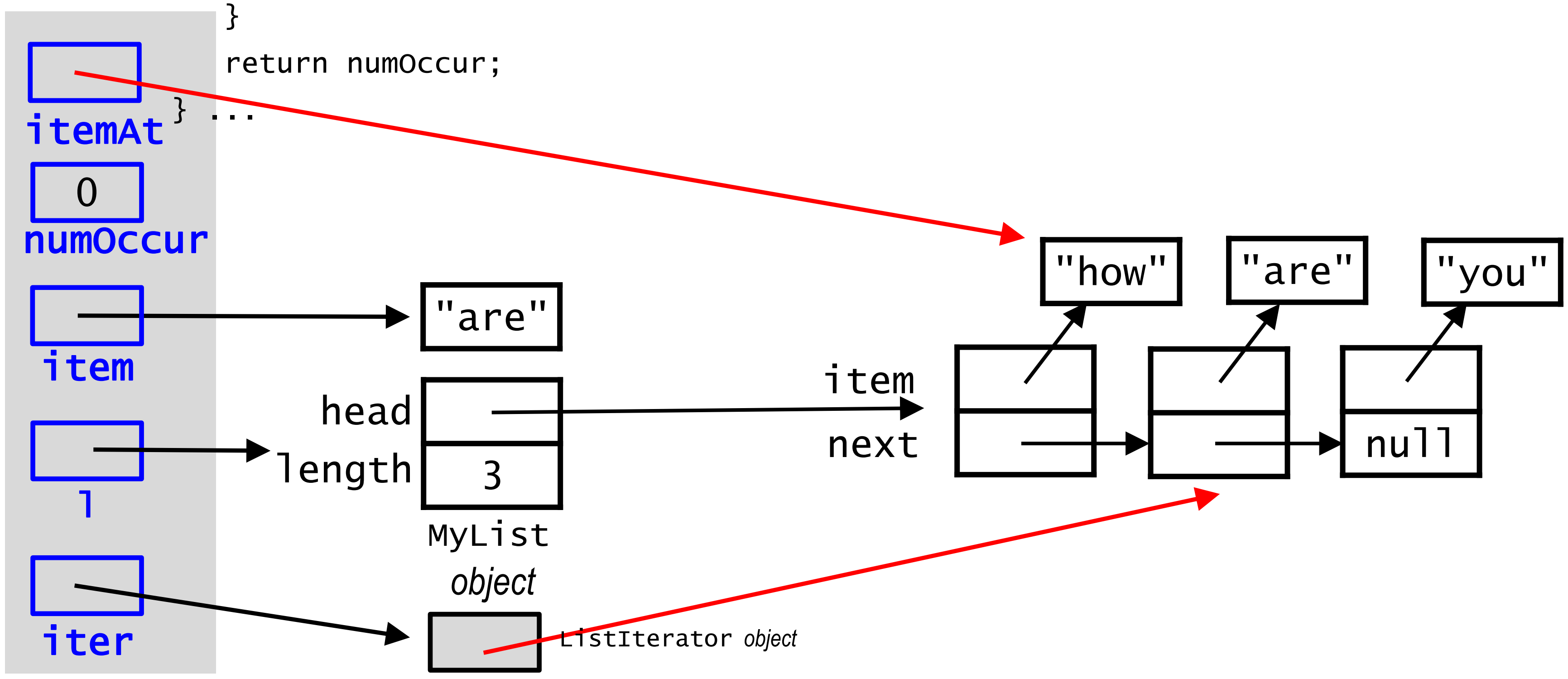
Example: *MyList* list

```
public class MyClass {
    public static int numOccur(List l, Object item) {
      int numOccur = 0;
      ListIterator iter = l.iterator();
       while ( iter.hasNext() ) {
           Object itemAt = iter.next();
           if (itemAt.equals(item)) {
               numOccur++;
           }
        }
        return numOccur;
     } ...
```



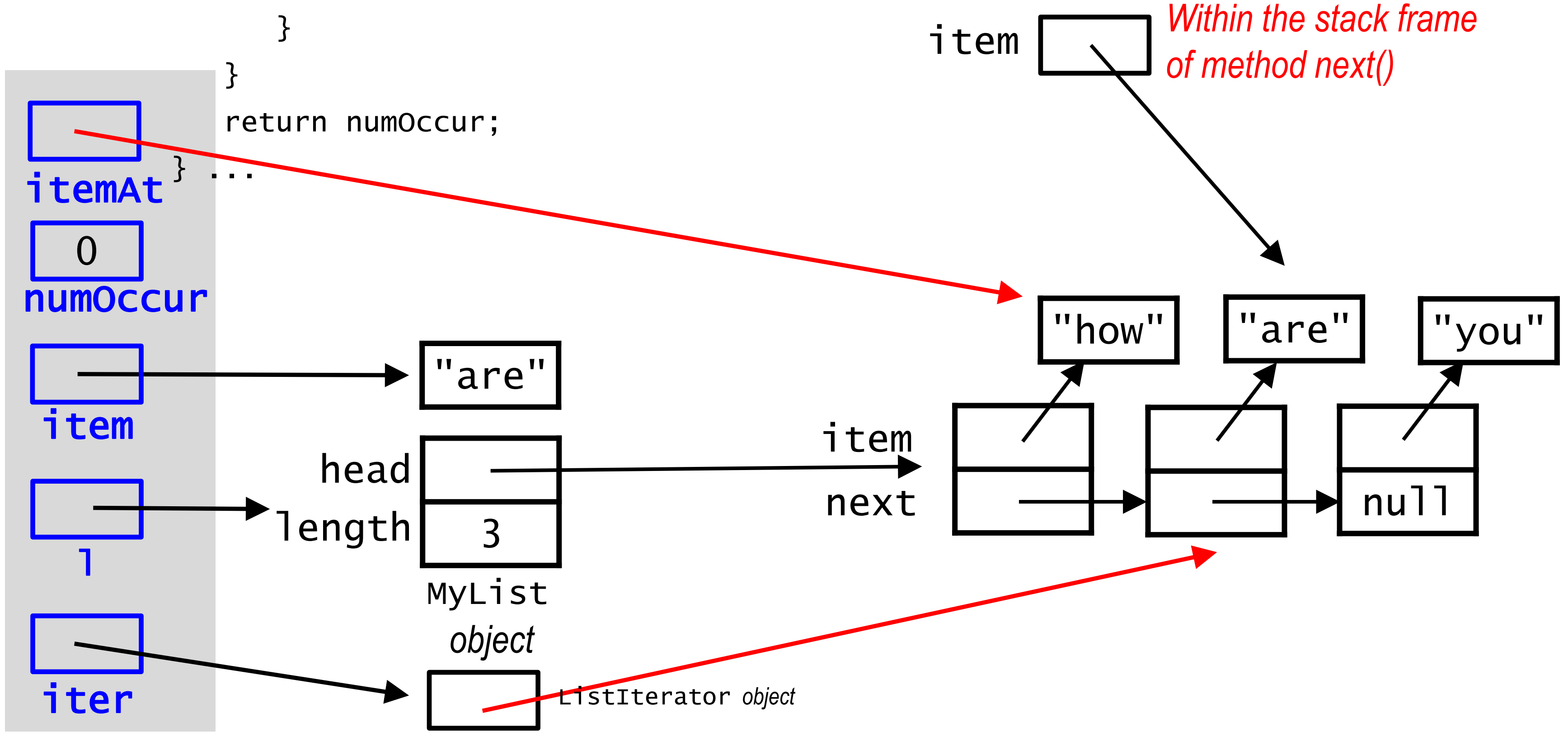
Example: *MyList* list

```
public class MyClass {

    public static int numOccur(List l, Object item) {

       int numOccur = 0;
       ListIterator iter = l.iterator();
        while ( iter.hasNext() ) {
            Object itemAt = iter.next();
            if (itemAt.equals(item)) {
                numOccur++;
            }
        }
        return numOccur;
    } ...
```



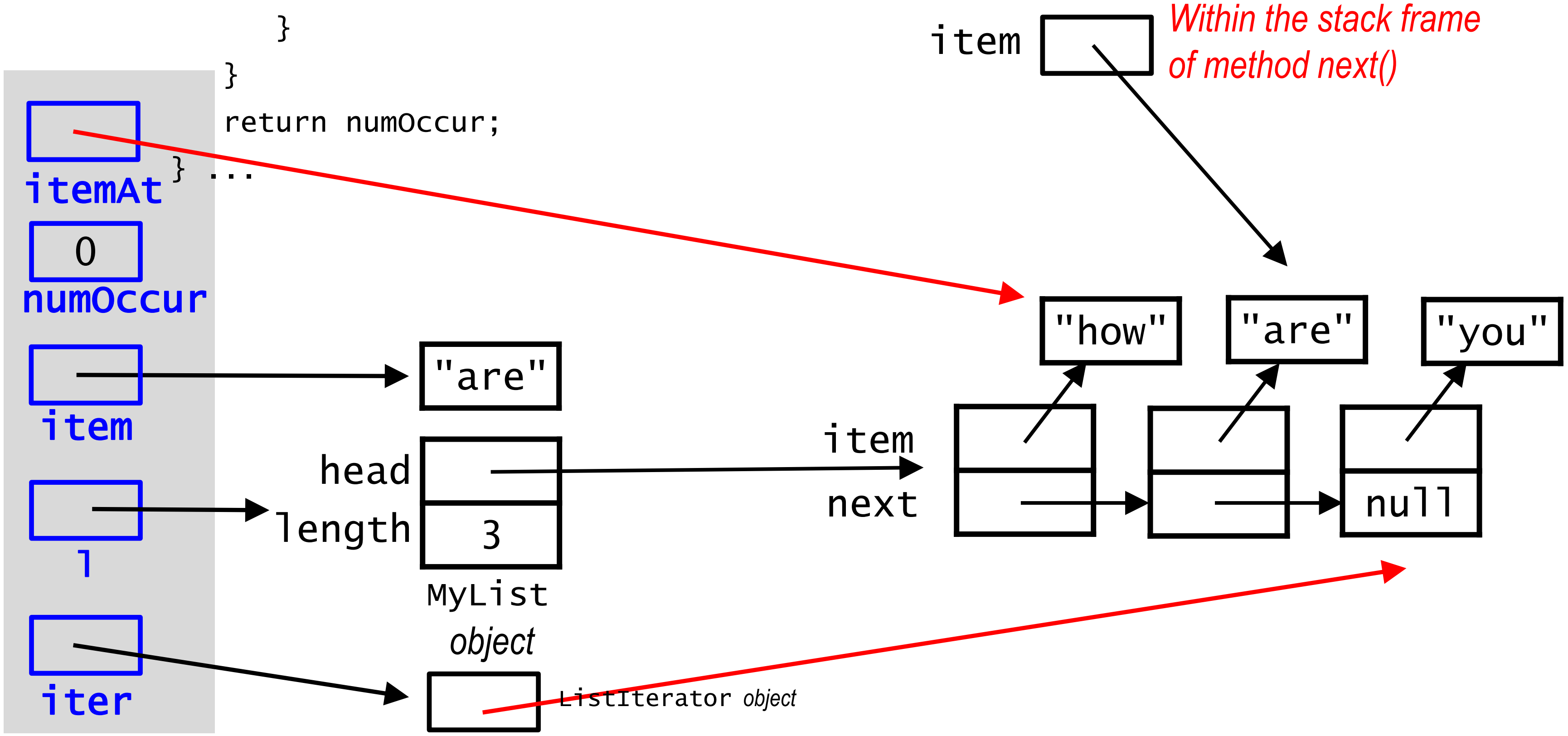
Example: *MyList* list

```
public class MyClass {
    public static int numOccur(List l, Object item) {
      int numOccur = 0;
      ListIterator iter = l.iterator();
       while ( iter.hasNext() ) {
           Object itemAt = iter.next();
           if (itemAt.equals(item)) {
               numOccur++;
           }
        }
        return numOccur;
     } ...
```



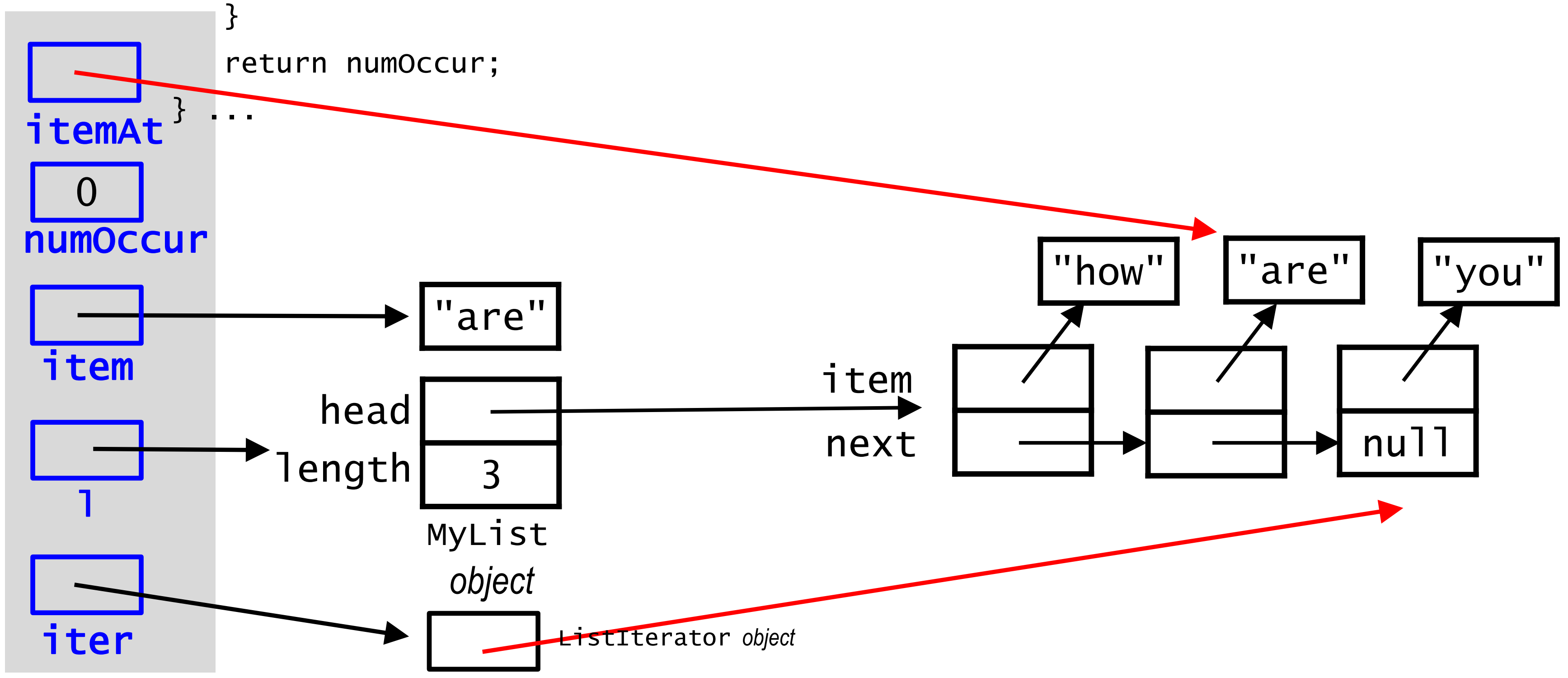
Example: *MyList* list

```
public class MyClass {

    public static int numOccur(List l, Object item) {

       int numOccur = 0;
       ListIterator iter = l.iterator();
        while ( iter.hasNext() ) {
            Object itemAt = iter.next();
            if (itemAt.equals(item)) {
                numOccur++;
            }
        }
        return numOccur;
    } ...
```



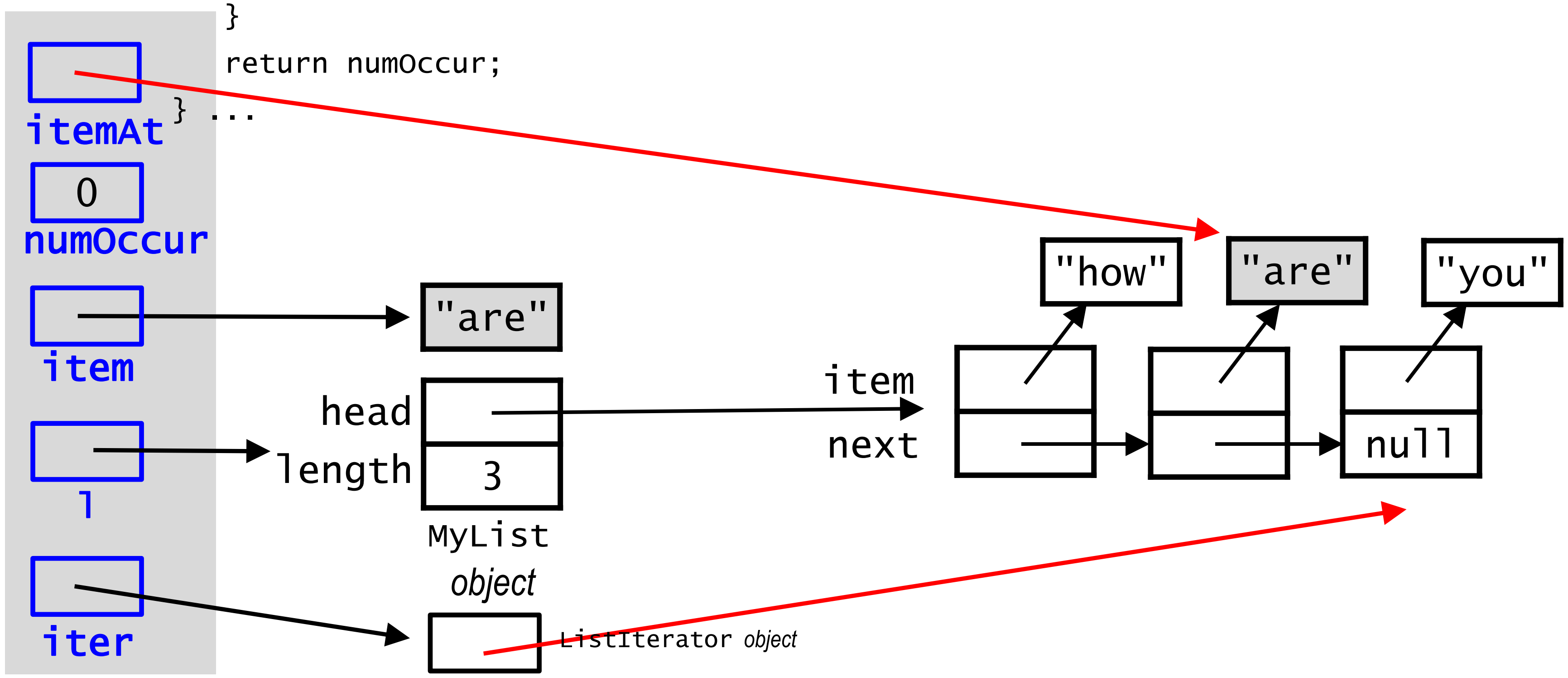
Example: *MyList* list

```
public class MyClass {

    public static int numOccur(List l, Object item) {

       int numOccur = 0;
       ListIterator iter = l.iterator();
        while ( iter.hasNext() ) {
            Object itemAt = iter.next();
            if (itemAt.equals(item)) {
                numOccur++;
            }
       }
       return numOccur;
    } ...
```



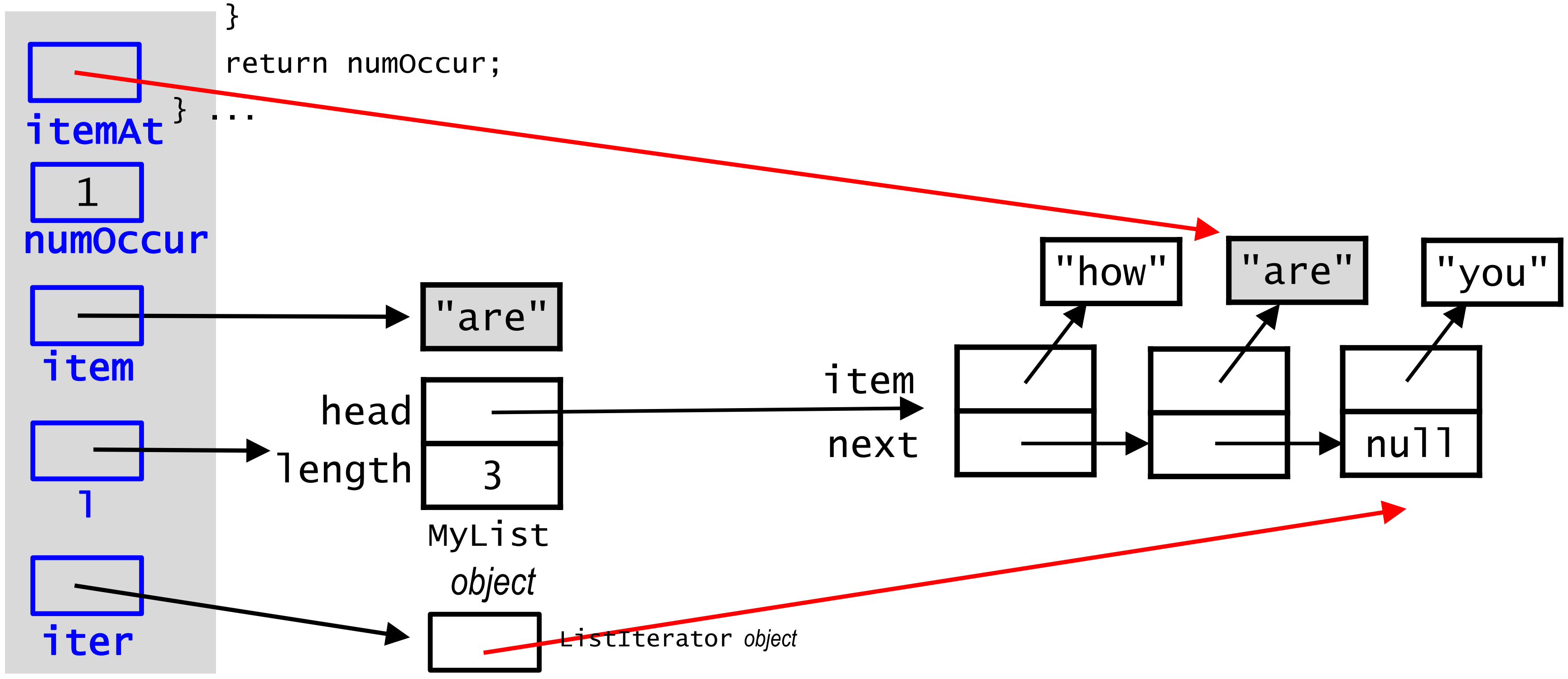
Example: *MyList* list

```
public class MyClass {

    public static int numOccur(List l, Object item) {

       int numOccur = 0;
       ListIterator iter = l.iterator();
        while ( iter.hasNext() ) {
            Object itemAt = iter.next();
            if (itemAt.equals(item)) {
                numOccur++;
            }
        }
        return numOccur;
    } ...
```



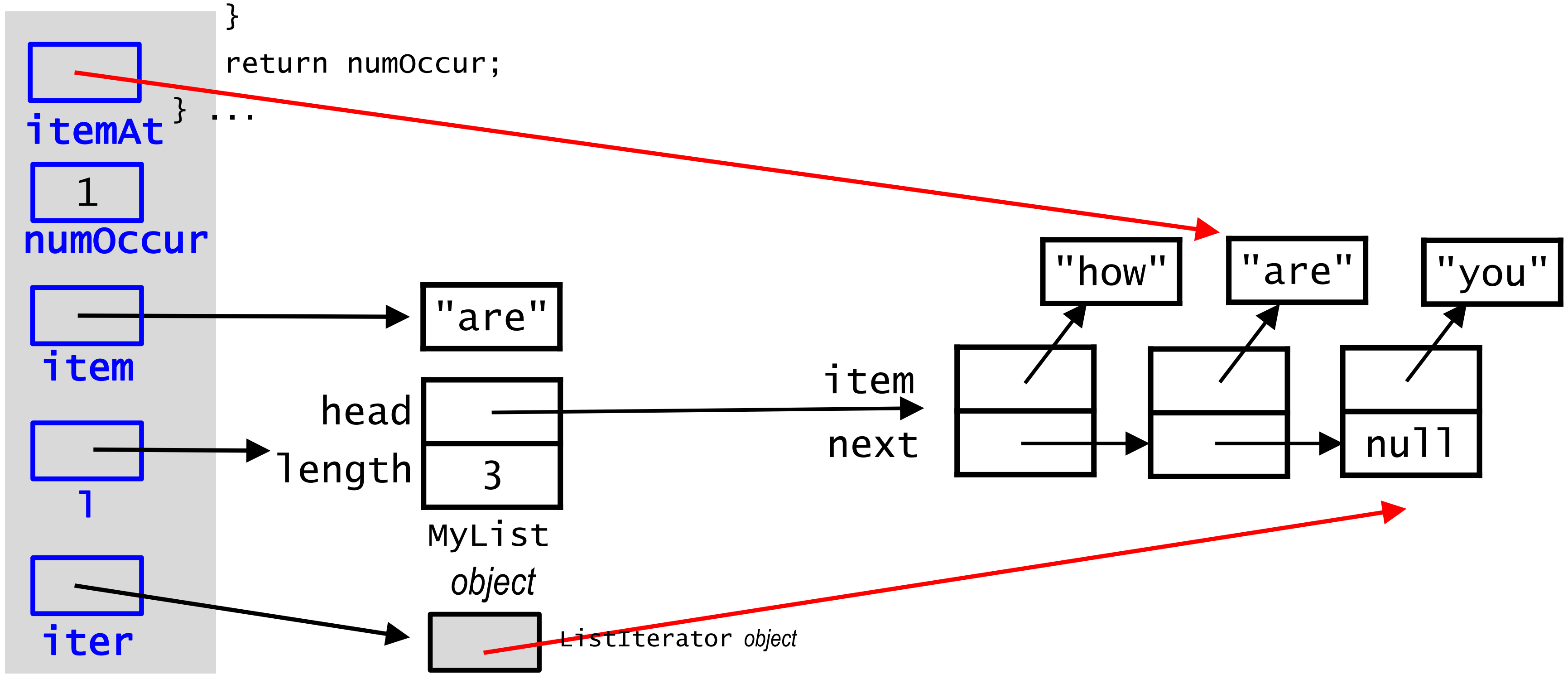
Example: *MyList* list

```
public class MyClass {

    public static int numOccur(List l, Object item) {

       int numOccur = 0;
       ListIterator iter = l.iterator();
        while ( iter.hasNext() ) {
            Object itemAt = iter.next();
            if (itemAt.equals(item)) {
                numOccur++;
            }
        }
        return numOccur;
    } ...
```



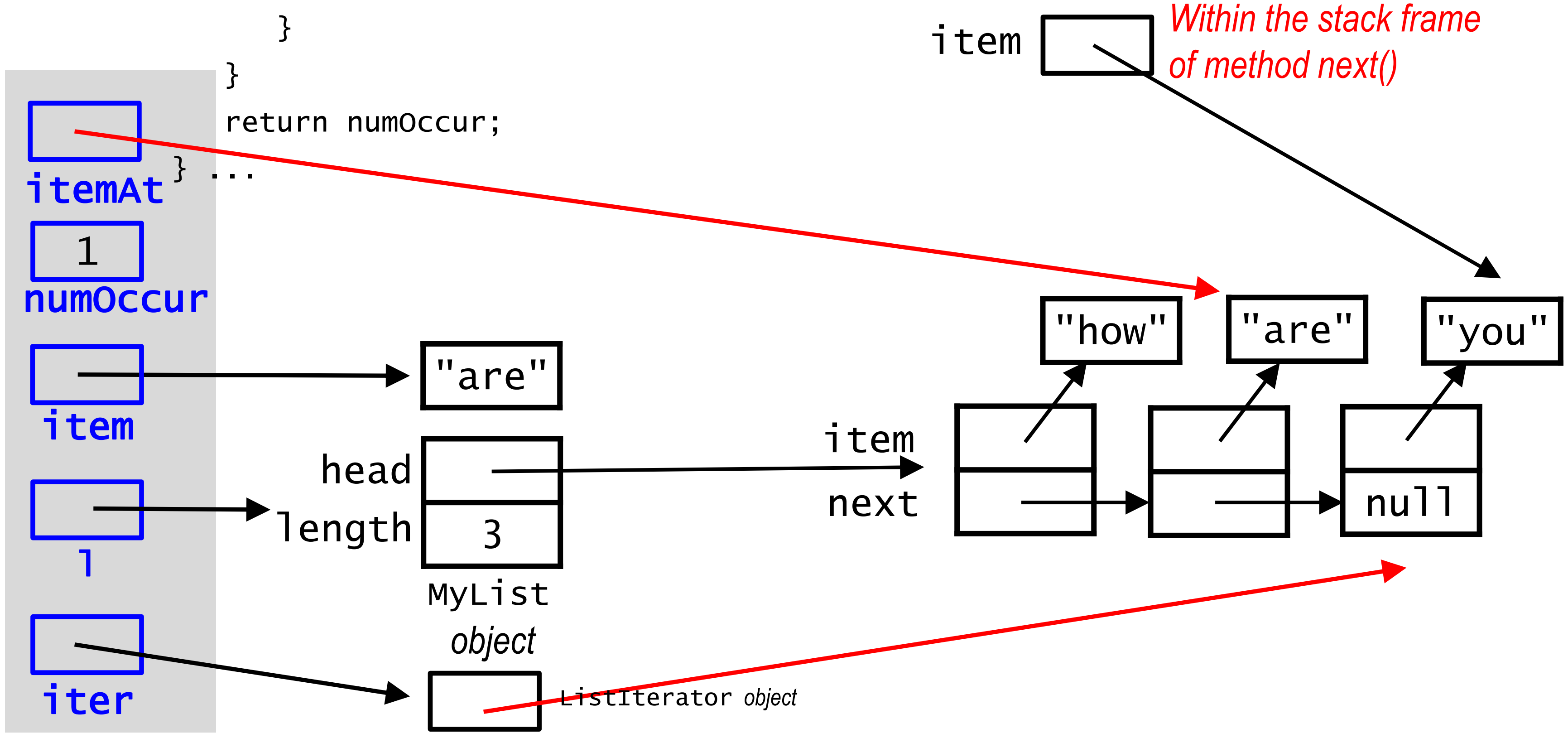
Example: *MyList* list

```
public class MyClass {
    public static int numOccur(List l, Object item) {
      int numOccur = 0;
      ListIterator iter = l.iterator();
       while ( iter.hasNext() ) {
           Object itemAt = iter.next();
           if (itemAt.equals(item)) {
               numOccur++;
           }
        }
        return numOccur;
     } ...
```



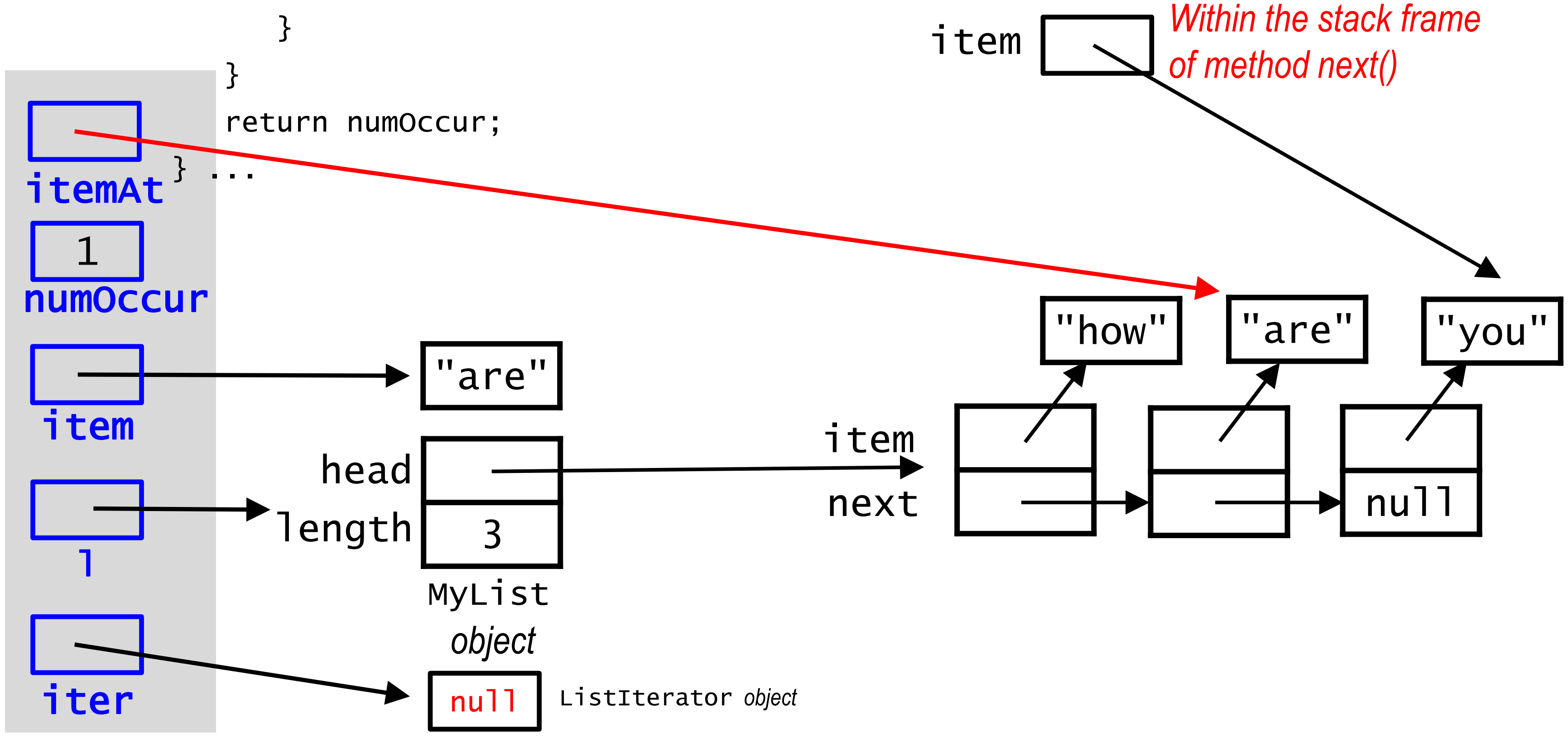
Example: *MyList* list

```
public class MyClass {

    public static int numOccur(List l, Object item) {

       int numOccur = 0;
       ListIterator iter = l.iterator();
        while ( iter.hasNext() ) {
            Object itemAt = iter.next();
            if (itemAt.equals(item)) {
                numOccur++;
            }
        }
        return numOccur;
    } ...
```



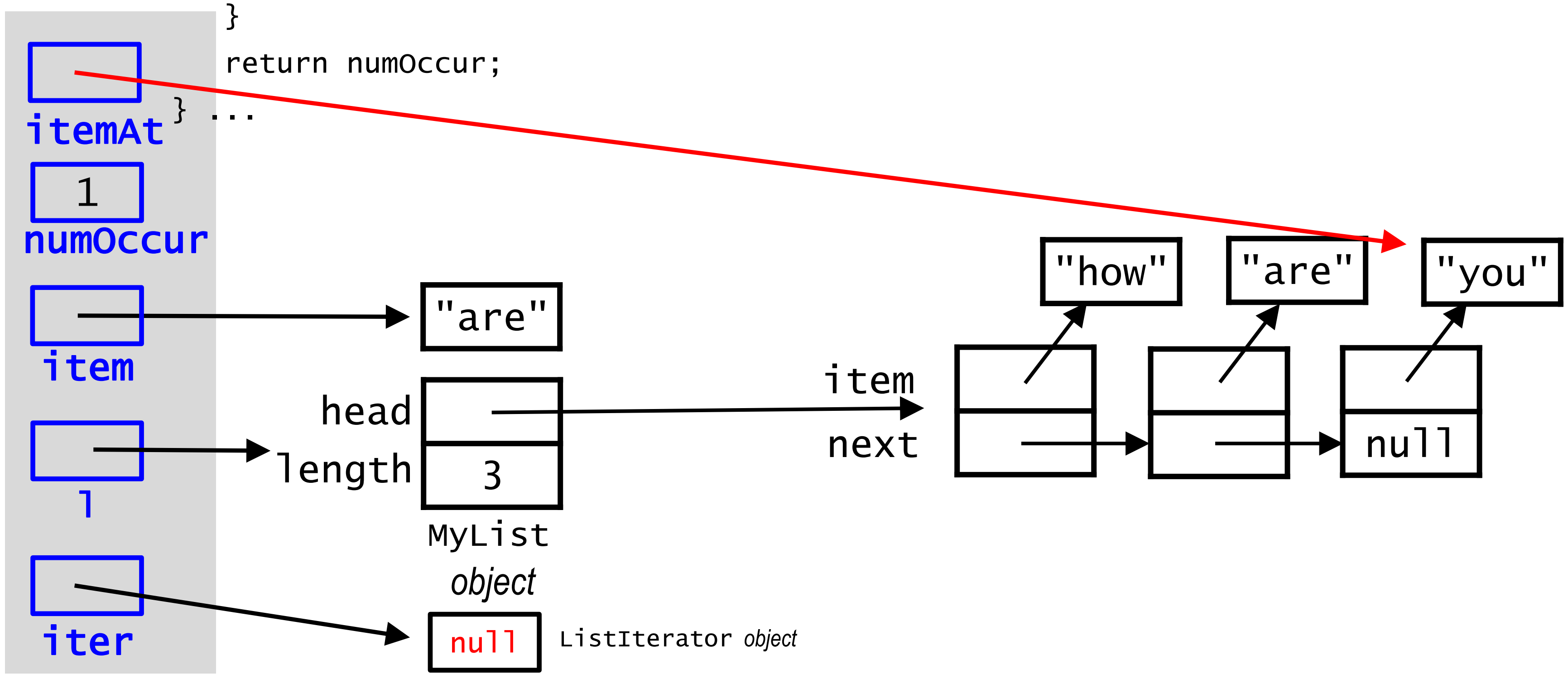
Example: *MyList* list

```
public class MyClass {

    public static int numOccur(List l, Object item) {

       int numOccur = 0;
       ListIterator iter = l.iterator();
        while ( iter.hasNext() ) {
            Object itemAt = iter.next();
            if (itemAt.equals(item)) {
                numOccur++;
            }
        }
        return numOccur;
    } ...
```



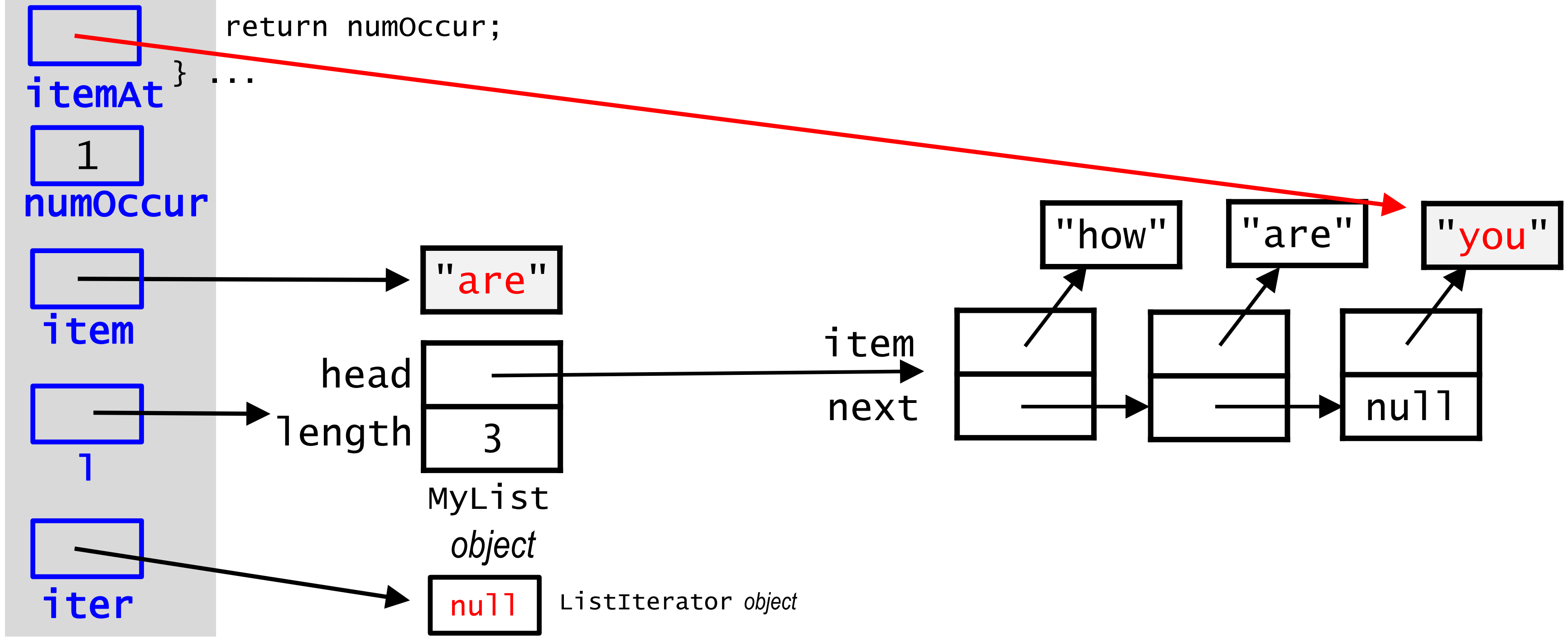
Example: *MyList* list

```
public class MyClass {

    public static int numOccur(List l, Object item) {

       int numOccur = 0;
       ListIterator iter = l.iterator();
        while ( iter.hasNext() ) {
            Object itemAt = iter.next();
            if (itemAt.equals(item)) {
                numOccur++;
            }
        }
        return numOccur;
    } ...
```



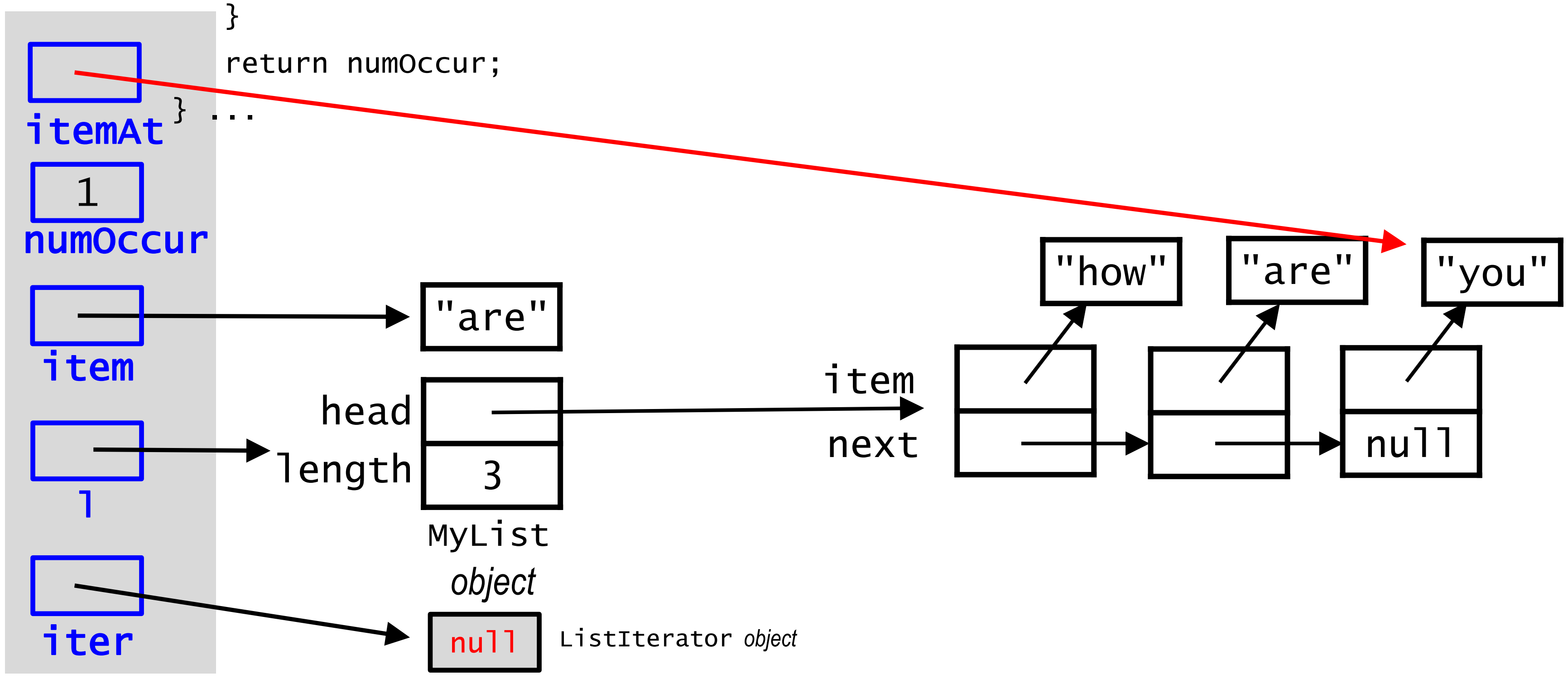
Example: *MyList* list

```
public class MyClass {

    public static int numOccur(List l, Object item) {

       int numOccur = 0;
       ListIterator iter = l.iterator();
        while ( iter.hasNext() ) {
            Object itemAt = iter.next();
            if (itemAt.equals(item)) {
                numOccur++;
            }
        }
        return numOccur;
    } ...
```



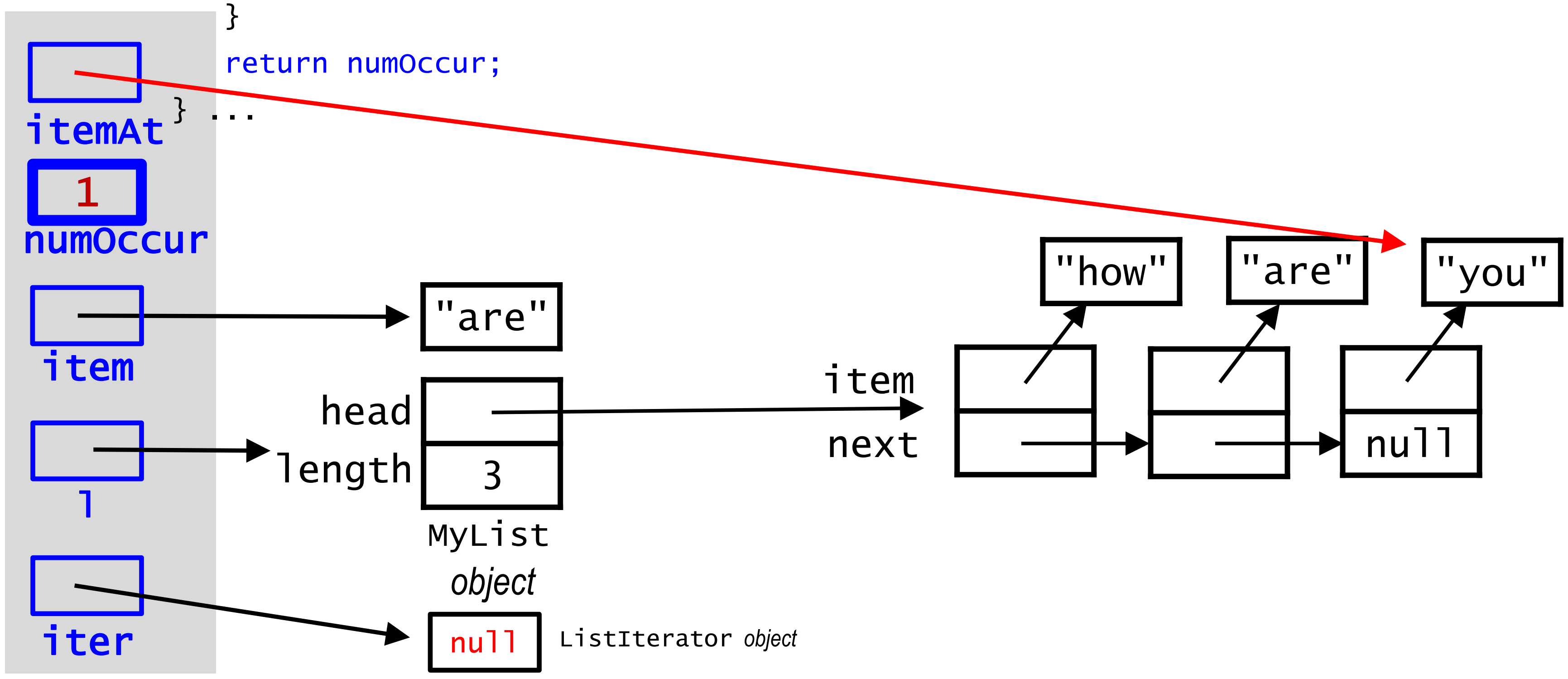
Example: *MyList* list

```
public class MyClass {

    public static int numOccur(List l, Object item) {

       int numOccur = 0;
       ListIterator iter = l.iterator();
        while ( iter.hasNext() ) {
            Object itemAt = iter.next();
            if (itemAt.equals(item)) {
                numOccur++;
            }
        }
        return numOccur;
    } ...
```



Example: *MyList* list

```
public class MyClass {

    public static int numOccur(List l, Object item) {

      int numOccur = 0;
      ListIterator iter = l.iterator();
       while ( iter.hasNext() ) {
           Object itemAt = iter.next();
           if (itemAt.equals(item)) {
               numOccur++;
           }
       }
       return numOccur;
    } ...
```



An Interface for List Iterators:
summary

Once the iterator interface has been implemented, we can create an instance of it and use it to externally traverse the list - regardless of the specific implementation of the List:

```
public class MyClass {

   public static int numOccur(List l, Object item) {

      int numOccur = 0;
      ListIterator iter = l.iterator();
      while ( iter.hasNext() ) {
         Object itemAt = iter.next();

         if (itemAt.equals(item)) {
            numOccur++;
         }
      }
      return numOccur;
   } ...
```

Note that this method can be called on any implementation of the List Interface.
What are the implications of this?

An Interface for List Iterators:

summary

Once the iterator interface has been implemented, we can create an instance of it and use it to externally traverse the list - regardless of the specific implementation of the List:

```
public class MyClass {

   public static int numOccur(List l, Object item) {

      int numOccur = 0;
      ListIterator iter = l.iterator();
      while ( iter.hasNext() ) {
         Object itemAt = iter.next();

         if (itemAt.equals(item)) {
            numOccur++;
         }
      }
      return numOccur;
   } ...
```

Note that this method can be called on any implementation of the List Interface.

What are the implications of this?

Collection of Students:
example

```
public class Students implements Iterable {
    private List<Student> students = null;

    public Students(){
        students = new ArrayList<Student>();
        students.add( new UndergraduateStudent() );
        students.add( new GraduateStudent() );
    }

    public Iterator iterator() {
        return students.iterator();
    }

    public static void main( String[] args ) {
        Students slist = new Students();
        Iterator iter = slist.iterator();

        while( iter.hasNext() )
            System.out.println( iter.next() );
    }
}
```

Collection of Students:
example

```
public class Students implements Iterable {
    private List<Student> students = null;

    public Students(){
        students = new ArrayList<Student>();
        students.add( new Undergradu
        students.add( new Gr
    }

    public Iterator iterat
        return students.itera
    }

    public static void main( String[] args ) {
        Students slist = new Students();
        Iterator iter = slist.iterator();

        while( iter.hasNext() )
            System.out.println( iter.next() );
    }
}
```

This iterator method is a method of which class?

Collection of Students:
example

```
public class Students implements Iterable {
    private List<Student> students = null;

    public Students(){
        students = new ArrayList<Student>();
        students.add( new UndergraduateStudent() );
        students.add( new GraduateStudent() );
    }

    public Iterator iterator() {
        return students.iterator();
    }

    public static void main( String[] args ) {
        Students slist = new Students();
        Iterator iter = slist.iterator();

        while( iter.hasNext() )
            System.out.println( iter.next() );
    }
}
```

Collection of Students:
example

```
public class Students impleme
    private List<Stude

    public Students()
        students = n
        students.add( ne                    ent() );
        students.add( new GraduateStudent() );
    }

    public Iterator iterator() {
        return students.iterator();
    }

    public static void main( String[] args ) {
        Students slist = new Students();
        Iterator iter = slist.iterator();

        while( iter.hasNext() )
            System.out.println( iter.next() );
    }
}
```

This iterator method is a method of which class?

Collection of Students:
example

```
public class Students implements Iterable {
    private List<Student> students = null;

    public Students(){
        students = new ArrayList<Student>();
        students.add( new UndergraduateStudent() );
        students.add( new GraduateStudent() );
    }

    public Iterator iterator() {
        return students.iterator();
    }

    public static void main( String[] args ) {
        Students slist = new Students();
        Iterator iter = slist.iterator();

        while( iter.hasNext() )
            System.out.println( iter.next() );
    }
}
```

Collection of Students:
example

```
public class Students implements Iterable {
    private List<Student> students = null;

    public Students(){
        students = new ArrayList<Student>();;
        students.add( new UndergraduateStudent() );
        students.add( new GraduateStudent() );
    }

    public Iterator iterator() {
        return students.iterator();
    }

    public static void main
        Students slist = new Stu

        for (Student s : slist )
            System.out.println( s );

    }
}
```

Compiler Error

Collection of Students:

example

```
public class Students implements Iterable<Student> {
    private List<Student> students = null;

    public Students(){
        students = new ArrayList<Student>();;
        students.add( new UndergraduateStudent() );
        students.add( new GraduateStudent() )
    }

    public Iterator iterator() {
        return students.iterator()
    }

    public static void main( Str
        Students slist = new Stude

        for (Student s : slist )
            System.out.println( s );

    }
}
```

Has to know that the method next() is returning a Student object.

Collection of Students:
example

```
public class Students implements Iterable<Student> {
    private List<Student> students = null;

    public Students(){
        students = new ArrayList<Student>();
        students.add( new UndergraduateStudent() );
        students.add( new GraduateStudent() );
    }

    public Iterator iterator() {
        return students.iterator();
    }

    public static void main
        Students slist =

        for (Student s :
            System.out.print

    }
}
```

Compiler warning of possible mistype.

Collection of Students:
example

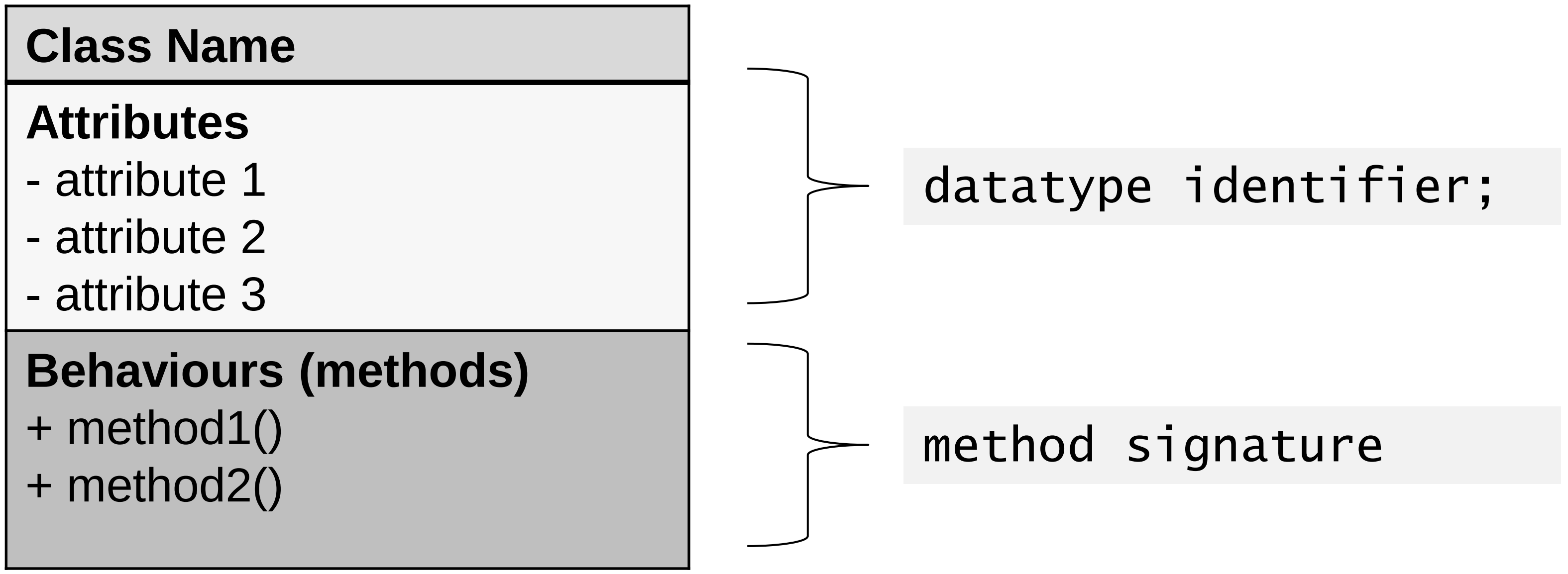
```
public class Students implements Iterable<Student> {
    private List<Student> students = null;

    public Students(){
        students = new ArrayList<Student>();;
        students.add( new UndergraduateStudent() );
        students.add( new GraduateStudent() );
    }

    public Iterator<Student> iterator() {
        return students.iterator();
    }

    public static void main( String[] args ) {
        Students slist = new Students();

        for (Student s : slist )
            System.out.println( s );

    }
}
```

Collection of Students:
example

```
public class Students implements Iterable<Student> {
    private List<Student> students = null;

    public Students(){
        students = new ArrayList<Student>();;
        students.add( new UndergraduateStudent() );
        students.add( new GraduateStudent() );
    }

    public Iterator<Student> iterator() {
        return students.iterator();
    }

    public static void main( String[] args ) {
        Students slist = new Students();

        slist.forEach(System.out::println);


    }
}
```

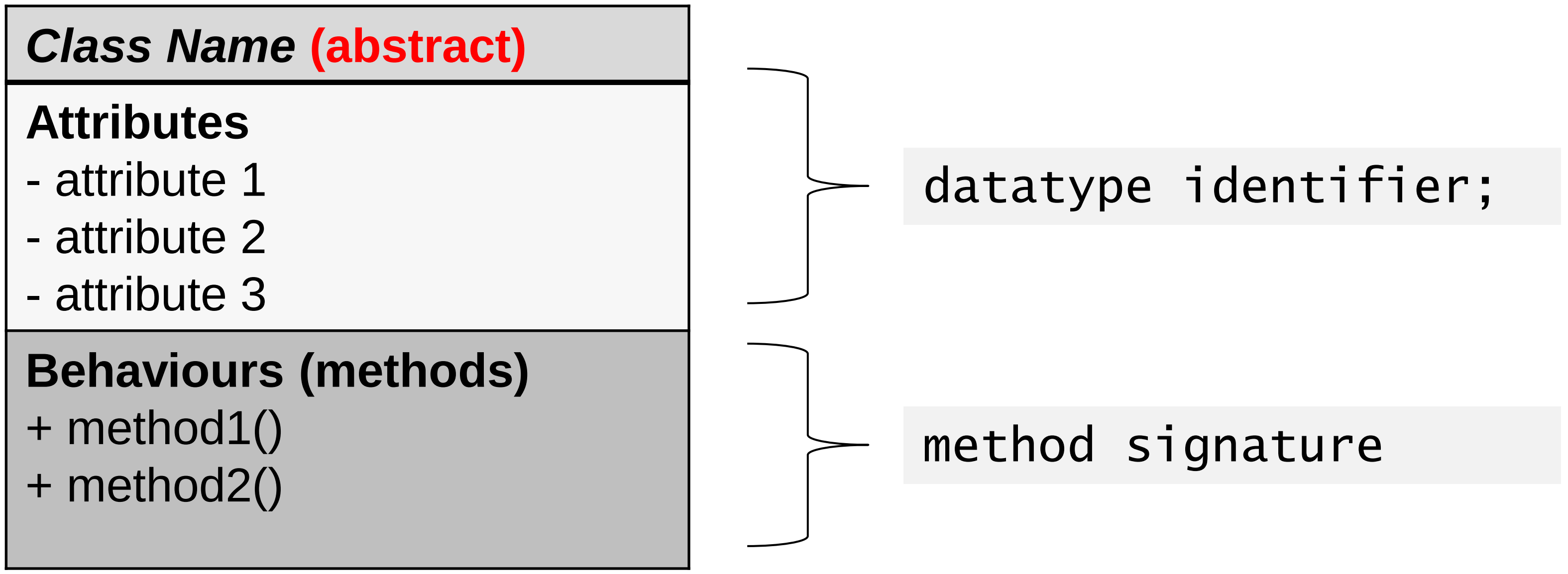
UML Diagrams

UML Tutorial

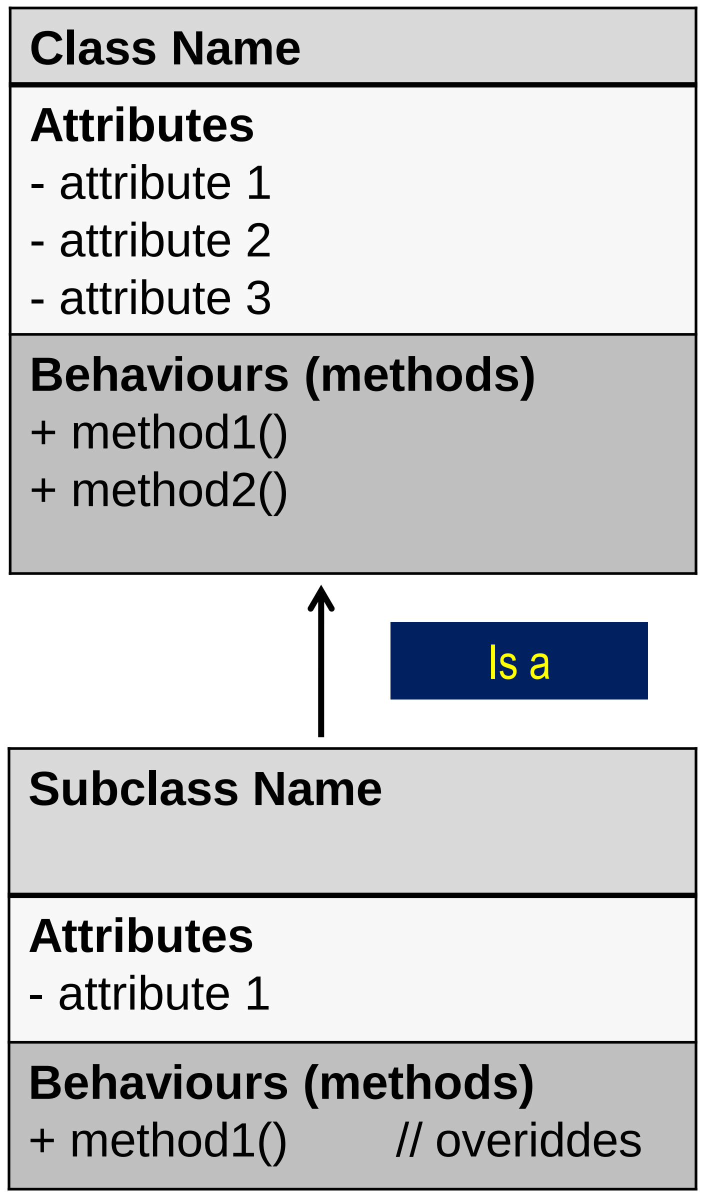


UML Diagrams

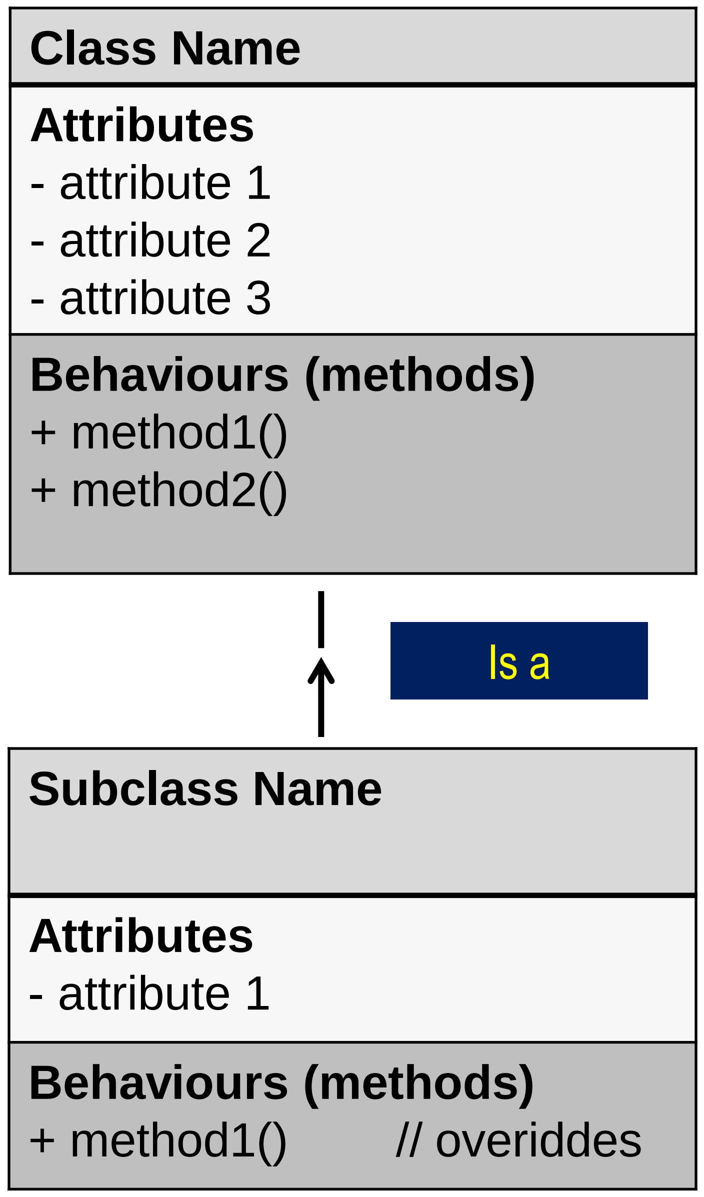
UML Tutorial



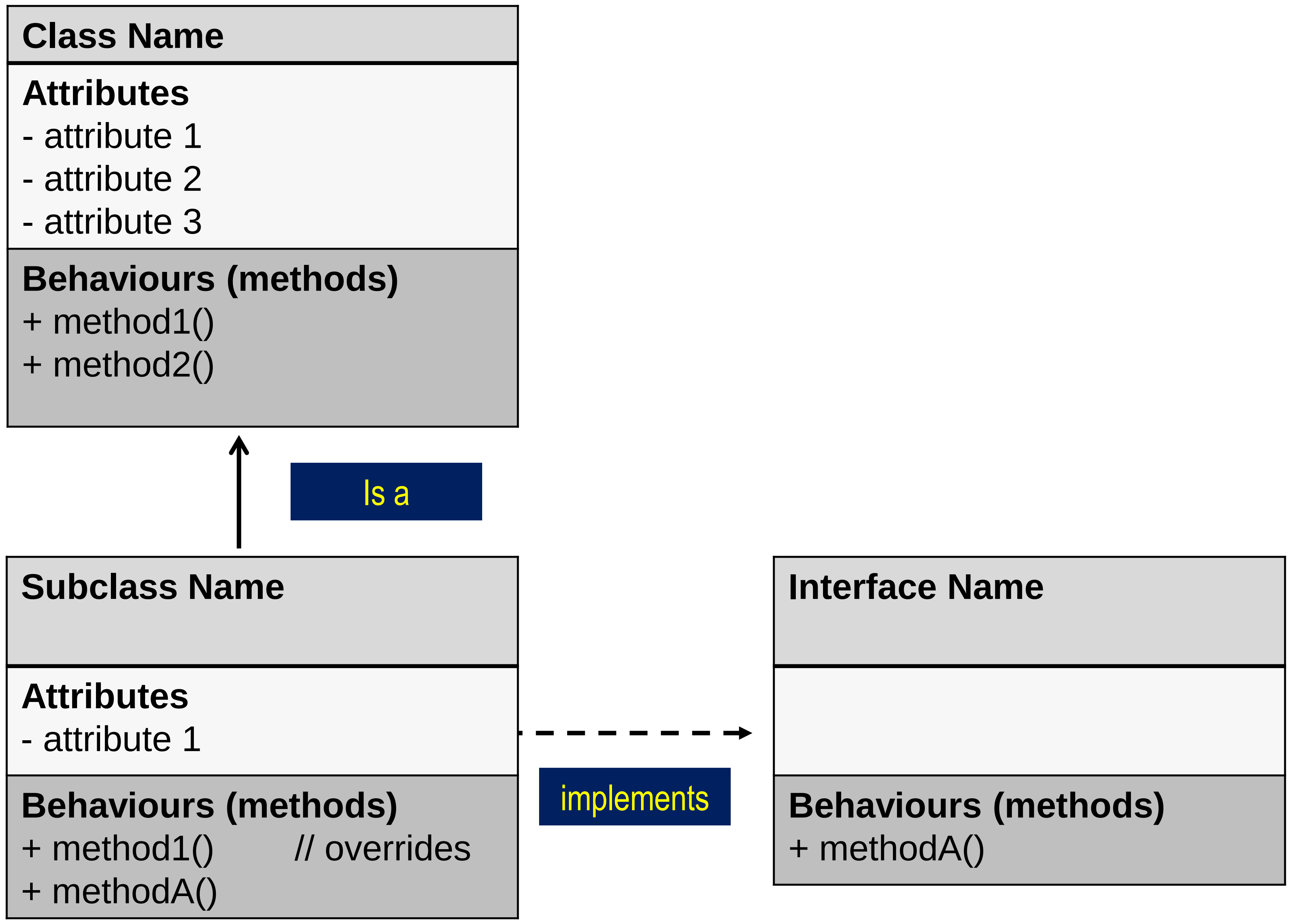
UML Diagrams



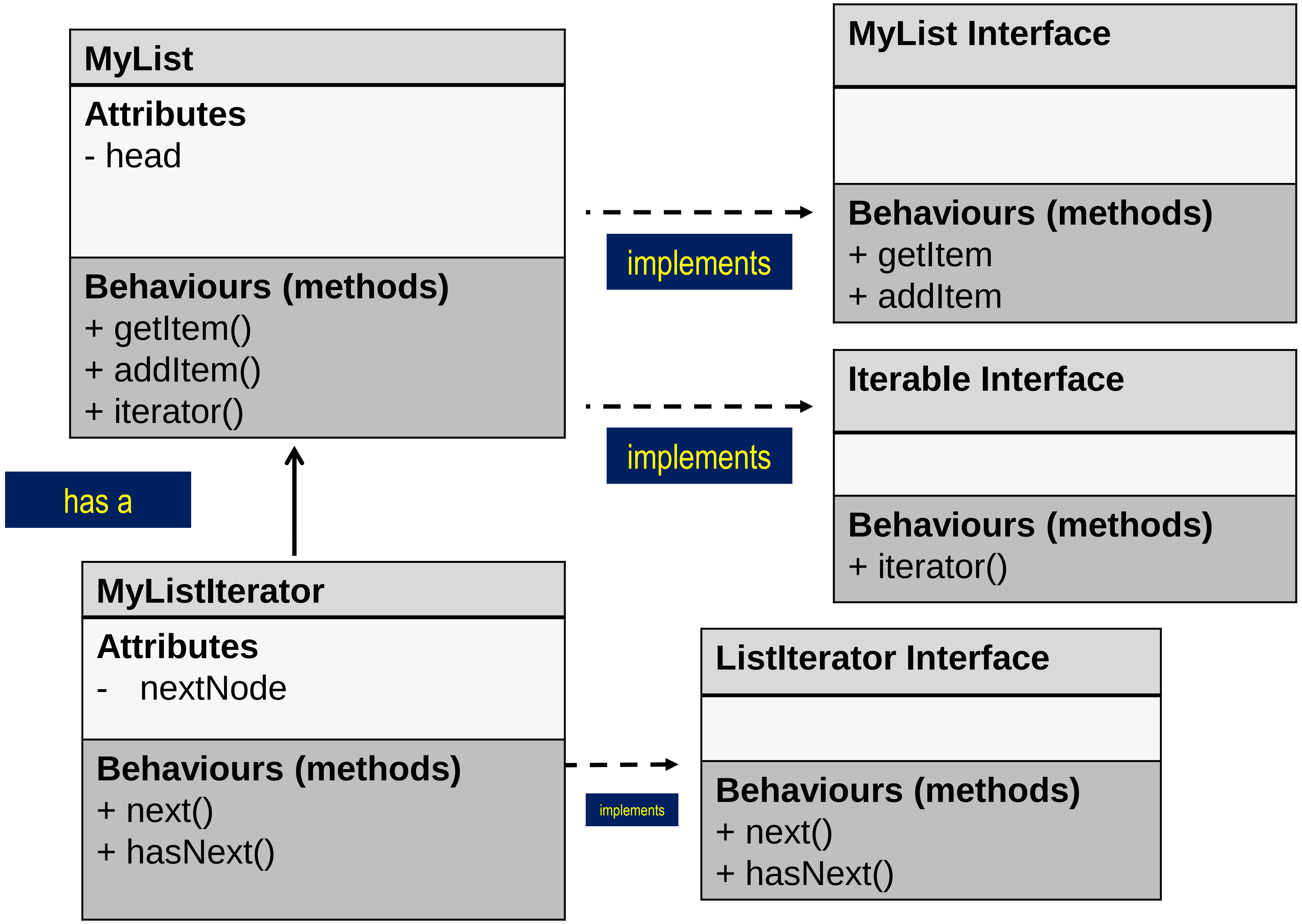
UML Diagrams



UML Diagrams



Recall out Iterator Example


MyList
Attributes
- head
Behaviours (methods)
+ getItem()
+ addItem()
+ iterator()
implements
MyList Interface
Behaviours (methods)
+ getItem
+ addItem
implements
Iterable Interface
Behaviours (methods)
+ iterator()
has a
MyListIterator
Attributes
- nextNode
Behaviours (methods)
+ next()
+ hasNext()
implements
ListIterator Interface
Behaviours (methods)
+ next()
+ hasNext()