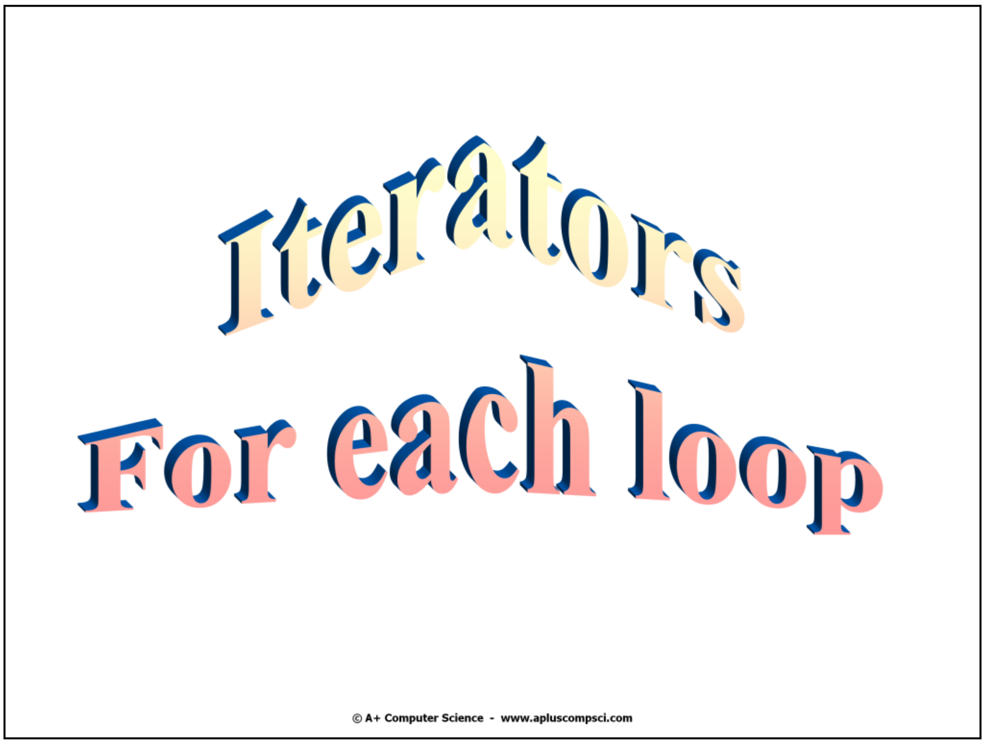
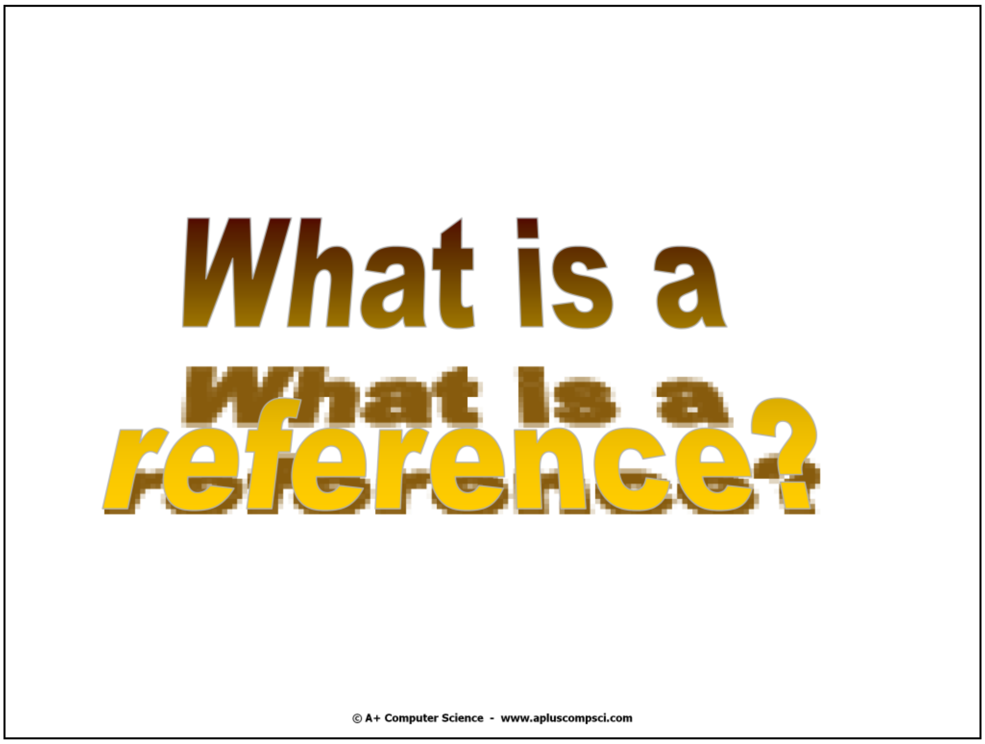
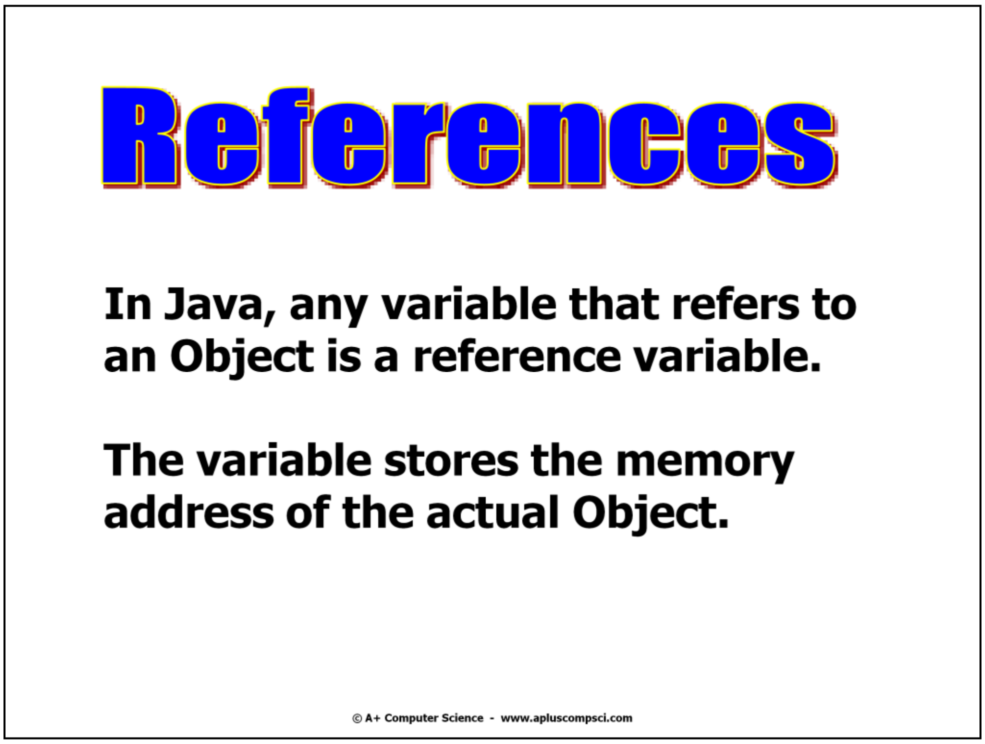


Iterators
For each loop



What is a reference?

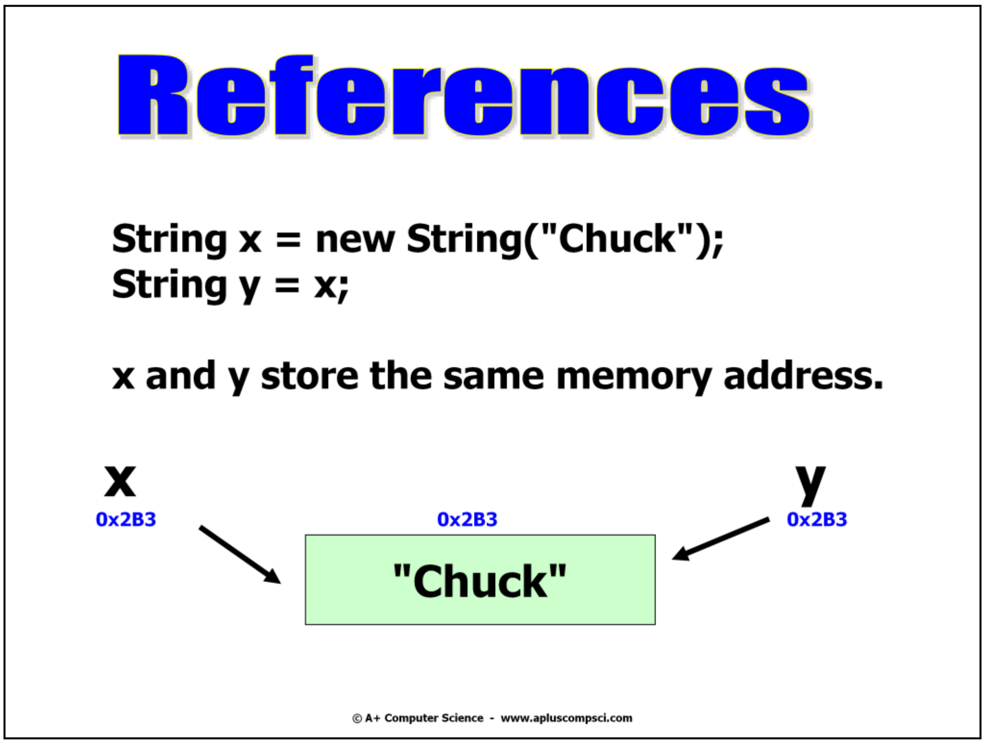


References

In Java, any variable that refers to an Object is a reference variable.

The variable stores the memory address of the actual Object.

All variables in Java that refer to Objects are called references. Reference variables store the location / memory address of the actual Object. For most situations, the value stored in a reference is a memory address.



References

```
String x = new String("Chuck");
String y = x;
```

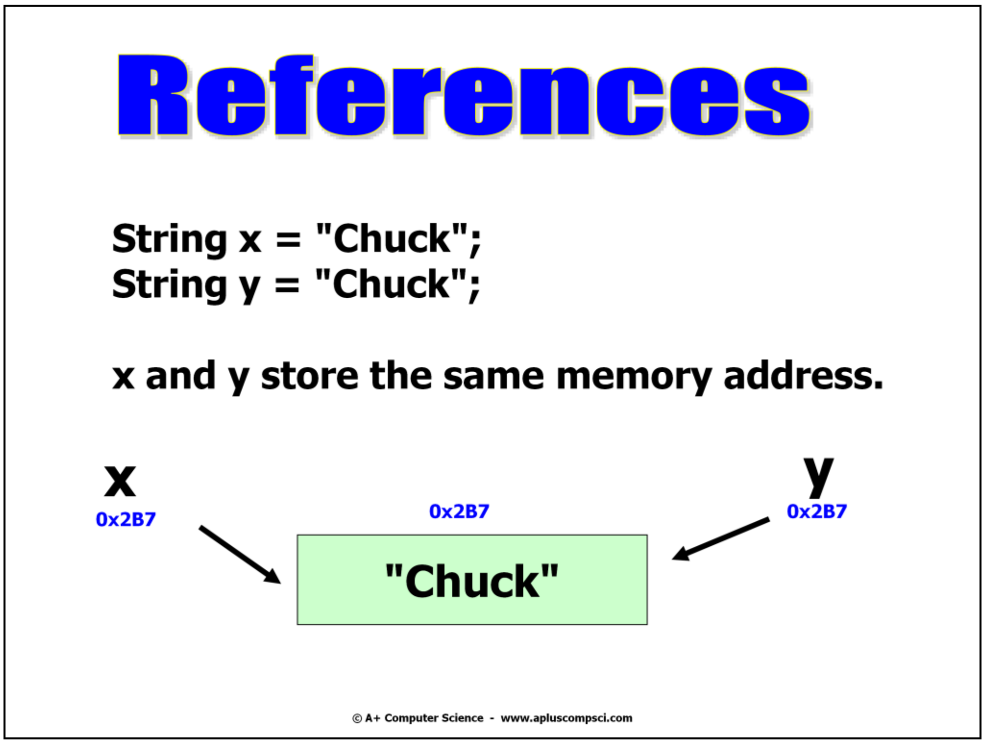
x and y store the same memory address.

x
0x2B3
y
0x2B3
0x2B3
"Chuck"

In this example, x and y both the store the location / address of `Chuck`. There is only one String containing `Chuck`. There are two reference variables storing the location / address of `Chuck`.

For this example, `x==y` is true. `x==y` compares the values stored in x and y. x and y both store the same location / address.

For this example, `x.equals(y)` is true. `x.equals(y)` compares the contents of the Objects referred to by x and y. `Chuck` is being compared to `Chuck`.



References

```
String x = "Chuck";
String y = "Chuck";
```

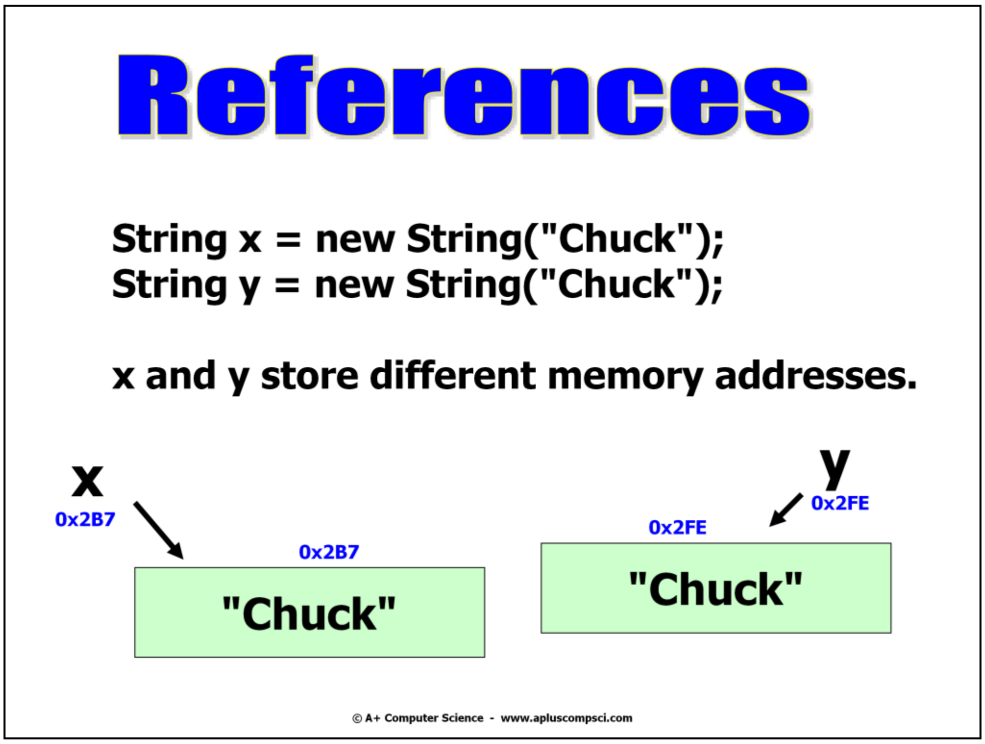
x and y store the same memory address.

x
0x2B7
0x2B7
"Chuck"
y
0x2B7

In this example, x and y both the store the location of `Chuck`. There is only one String containing `Chuck`. There are two reference variables storing the location / address of `Chuck`.

For this example, `x==y` is true. `x==y` compares the values stored in x and y. x and y both store the same location / address.

For this example, `x.equals(y)` is true. `x.equals(y)` compares the contents of the Objects referred to by x and y. `Chuck` is being compared to `Chuck`.



References

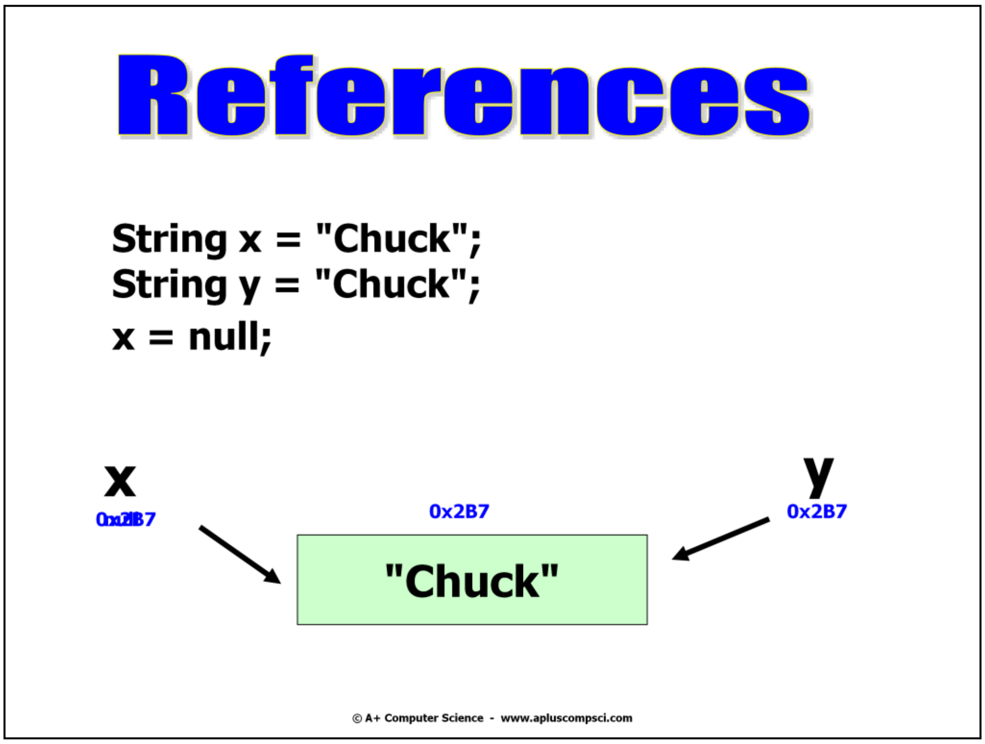
```
String x = new String("Chuck");
String y = new String("Chuck");
```

x and y store different memory addresses.

In this example, x stores the location / address of a String Object that stores the value `Chuck`. y also stores the location of a different String Object that stores the value `Chuck`. x and y do not store the same location / address.

For this example, `x==y` is false. x and y do not store the same location / address.

For this example, `x.equals(y)` is true.



References

```
String x = "Chuck";
String y = "Chuck";
x = null;
```

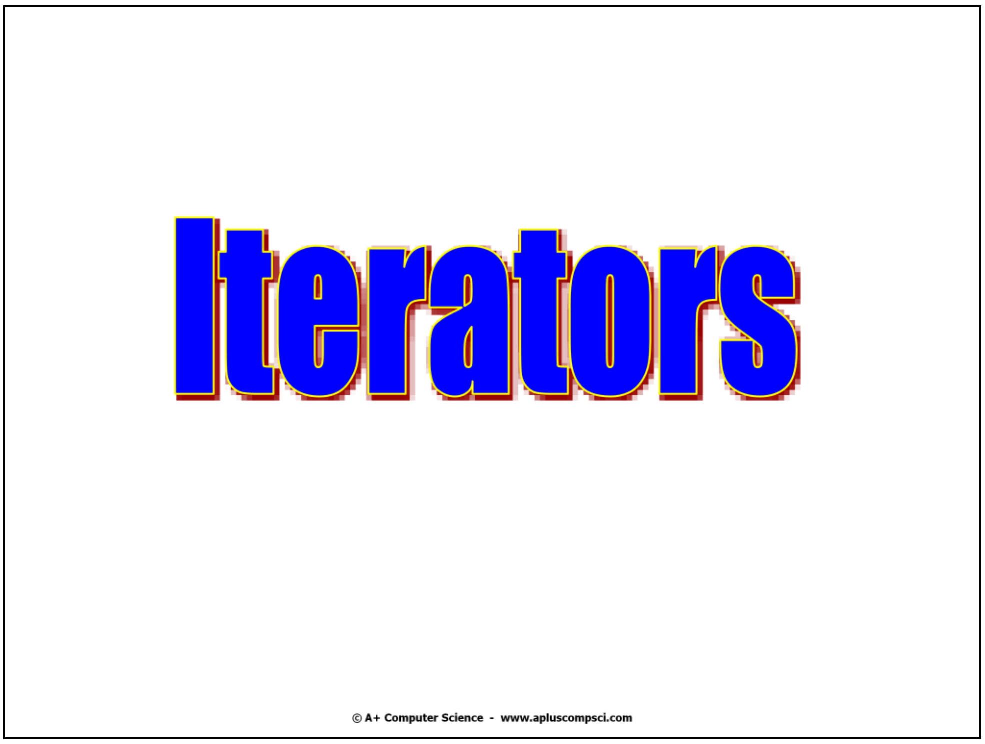
In this example, x and y both the store the location / address of `Chuck`. There is only one String containing `Chuck`. There are two reference variables storing the location / address of `Chuck`.

At the start, `x==y` is true.

x is then referred to null. x now stores null. y was in no way changed. y still stores the address of `Chuck`.

After changing the value of x, `x==y` is false.

references.java



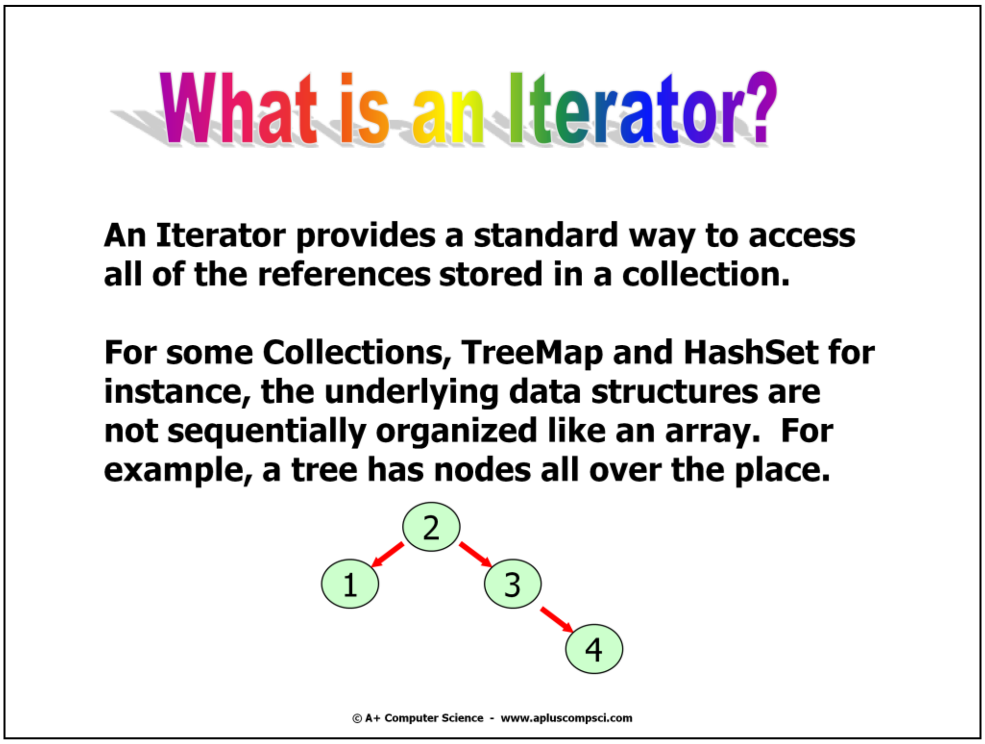
Iterators

Java Iterators

Collection, List, and Set all have methods that return iterators.

Iterators allow you to go from item to item through a collection.

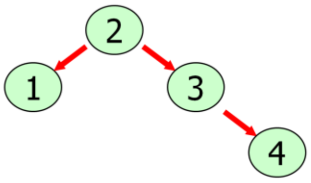
Map does not have an iterator, but it does have a keySet() method that returns a Set of all keys. You can get an iterator from the Set.



What is an Iterator?

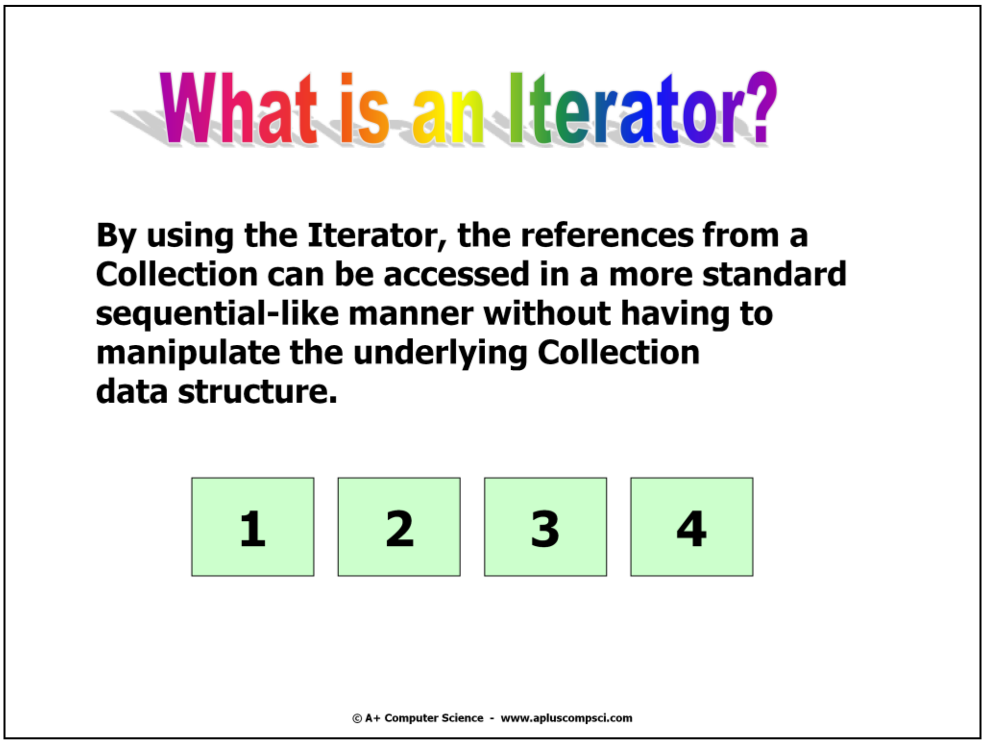
An Iterator provides a standard way to access all of the references stored in a collection.

For some Collections, TreeMap and HashSet for instance, the underlying data structures are not sequentially organized like an array. For example, a tree has nodes all over the place.



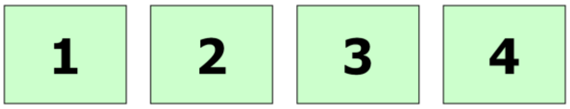
An Iterator provides a uniform way to traverse a data structure.

As ArrayList, LinkedList, and Set all have Iterators, you can access the references in these structures using the same set of methods. Iterators create uniformity and make accessing the data structure references a similar process.

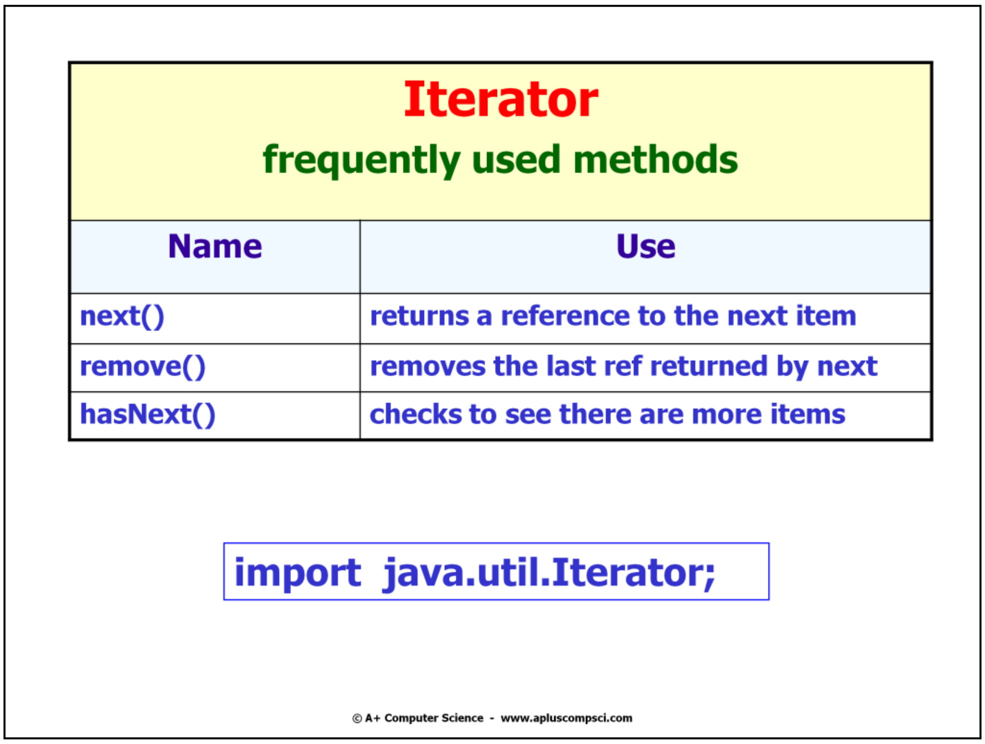


What is an Iterator?

By using the Iterator, the references from a Collection can be accessed in a more standard sequential-like manner without having to manipulate the underlying Collection data structure.



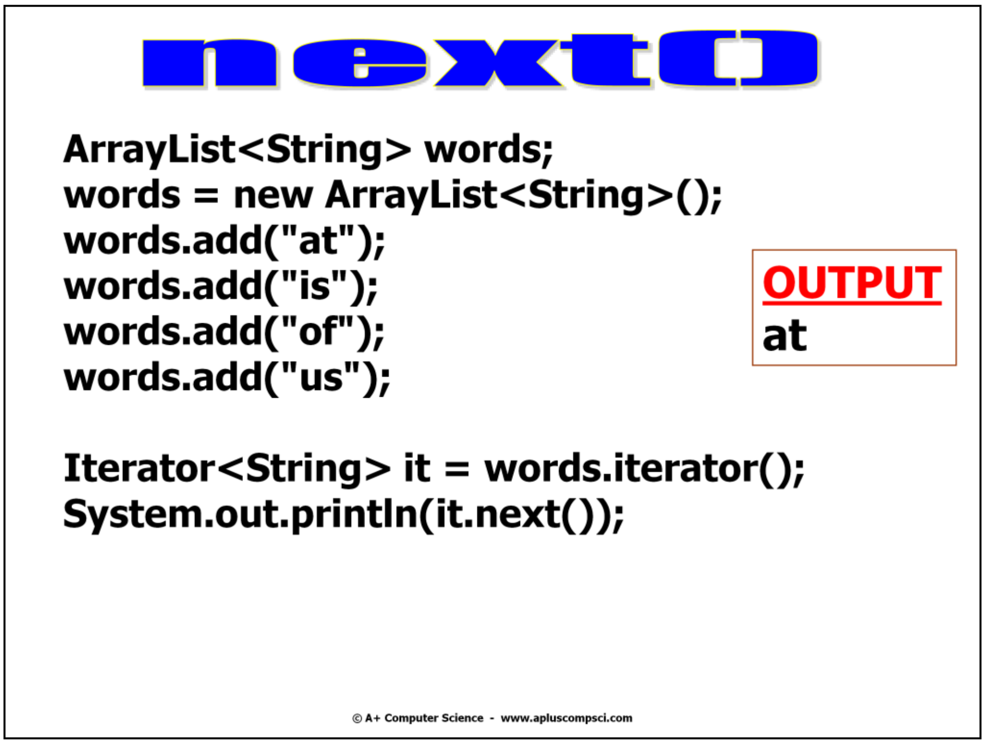
Iterator Interface



Iterator
frequently used methods

Name	Use
next()	returns a reference to the next item
remove()	removes the last ref returned by next
hasNext()	checks to see there are more items

```
import  java.util.Iterator;
```



next()

```
ArrayList<String> words;
words = new ArrayList<String>();
words.add("at");
words.add("is");
words.add("of");
words.add("us");

Iterator<String> it = words.iterator();
System.out.println(it.next());
```

OUTPUT

```
at
```

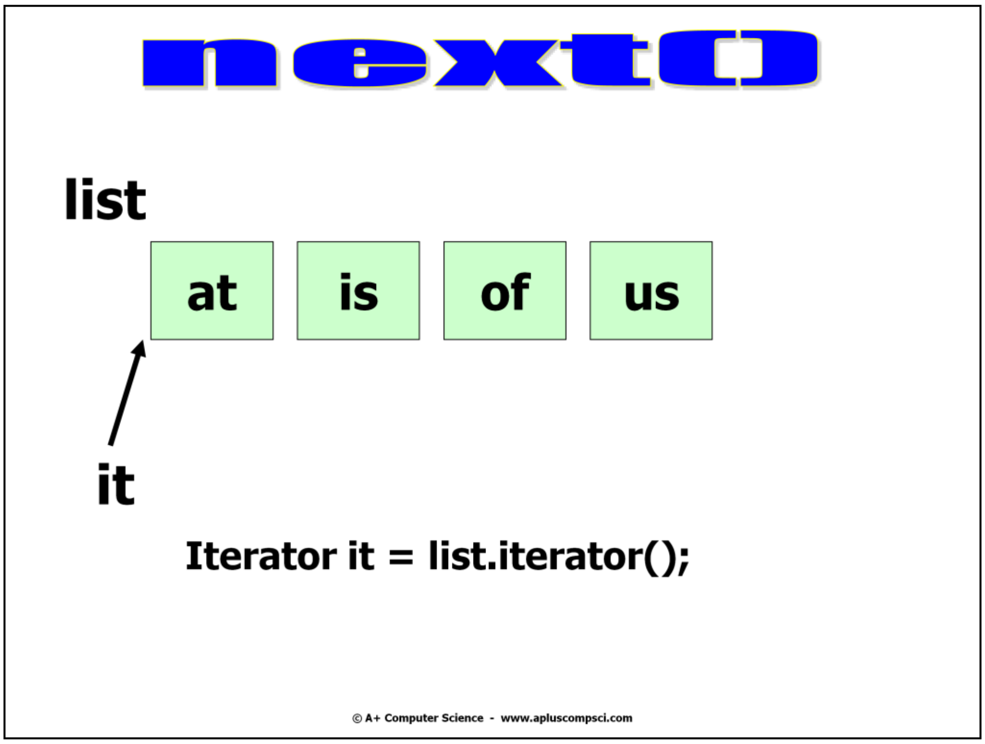
© A+ Computer Science - www.apluscompsci.com

An iterator provides a standard way to access all of the items in a data structure.

An iterator allows movement from one reference to the next.

When the `next()` method is called, the next reference in the list is returned and the iterator moves to the next reference.

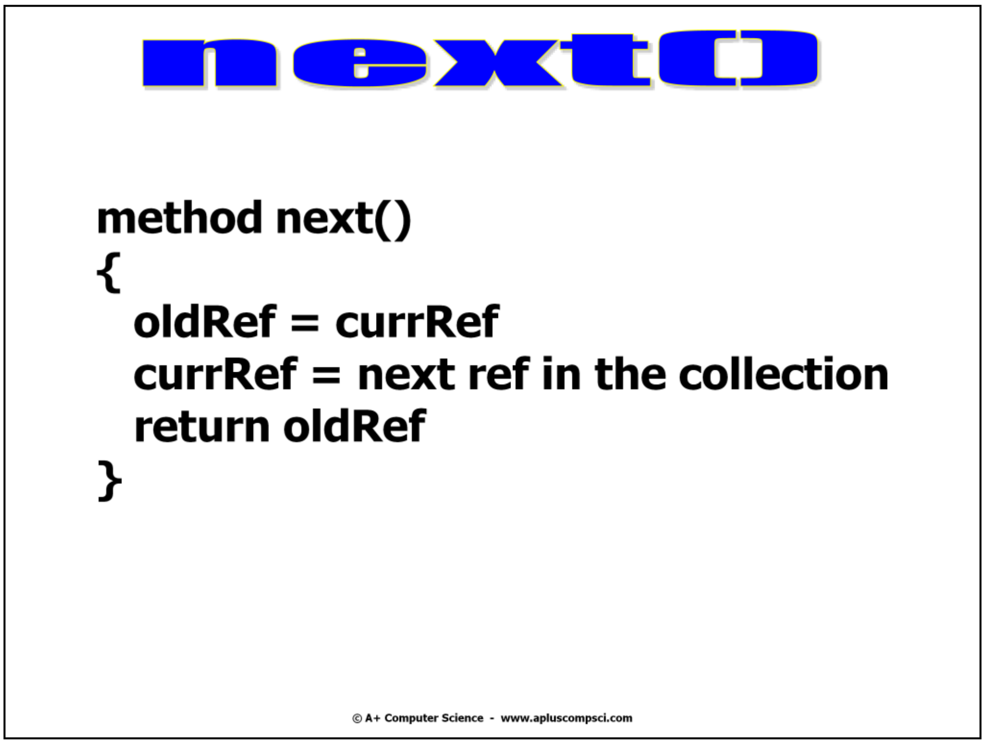
The next methods movement is based on the data structure that the iterator is working on.



next()

```
Iterator it = list.iterator();
```

An iterator essentially points to an area in front of each reference. It starts out pointing in front of the 1st reference.

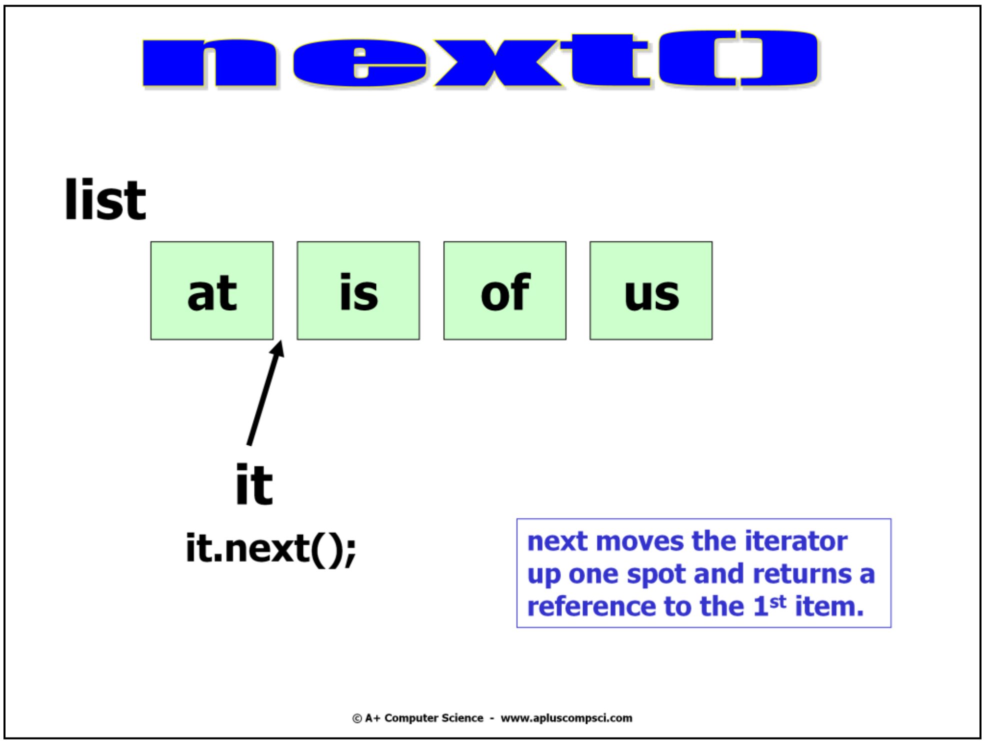


next()

```
method next()
{
  oldRef = currRef
  currRef = next ref in the collection
  return oldRef
}
```

When the `next()` method is called, a reference to the current item is returned and the iterator moves up one spot.

How does this happen? The Iterator saves the current position of the iterator. When the next() method is called, the old position is returned and the next position is saved as the current position of the iterator. Each time next() is called, the next position becomes the current position.

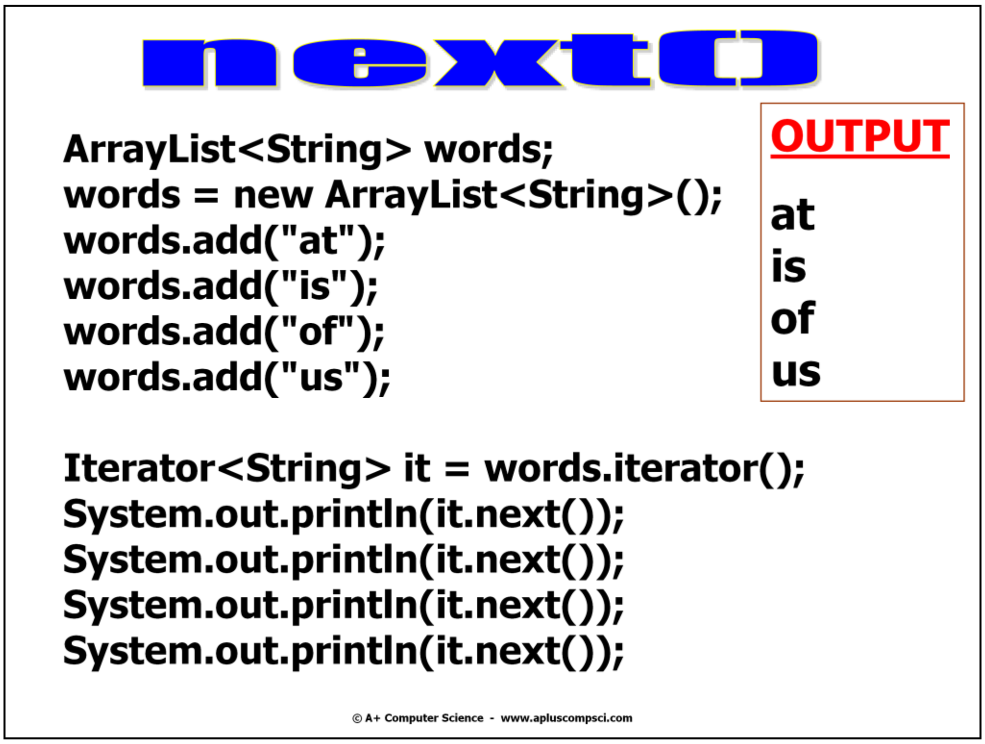

next()
list
at
is
of
us
it
it.next();
next moves the iterator up one spot and returns a reference to the 1st item.
© A+ Computer Science - www.apluscompsci.com

When the `next()` method is called, a reference to the current item is returned and the iterator moves up one spot.

How does this happen? The Iterator saves the current position of the iterator. When the next() method is called, the old position is returned and the next position is saved as the current position of the iterator. Each time next() is called, the next position becomes the current position.

In the example above, `next()` causes the iterator to slide past `"at"` and stop in front of `"is"`. `next()` returns the reference to `"at"`.

So, behind the scenes, `"at"` is returend and `"is"` is saved as the new current position.



next()

```
ArrayList<String> words;
words = new ArrayList<String>();
words.add("at");
words.add("is");
words.add("of");
words.add("us");

Iterator<String> it = words.iterator();
System.out.println(it.next());
System.out.println(it.next());
System.out.println(it.next());
System.out.println(it.next());
```

OUTPUT

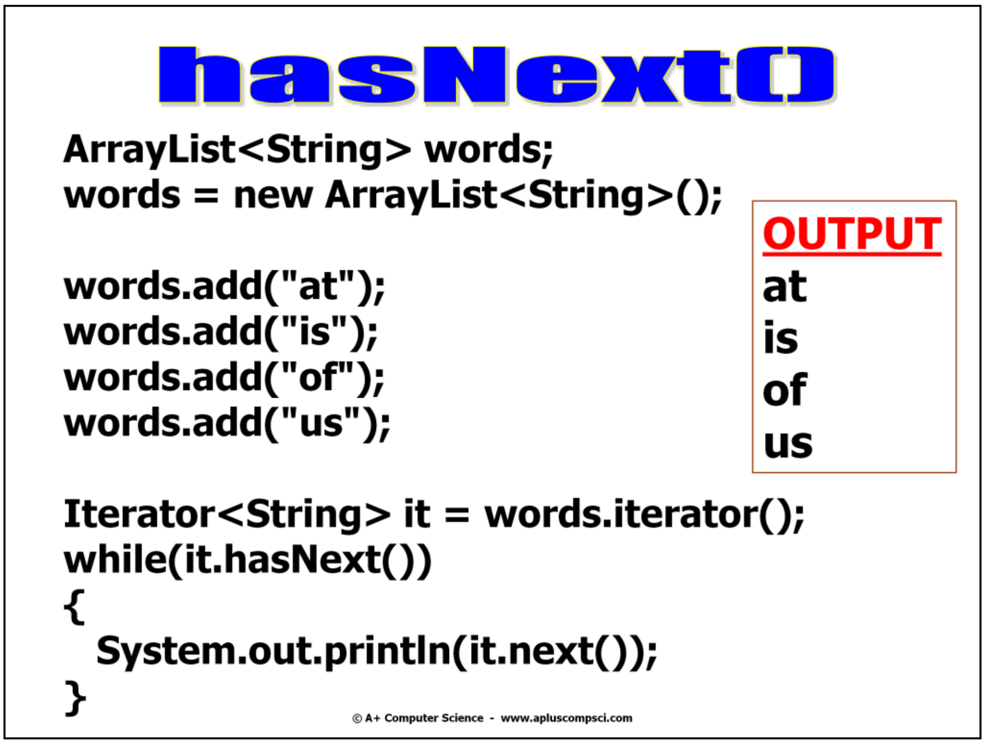
```
at
is
of
us
```

© A+ Computer Science - www.apluscompsci.com

When the `next()` method is called, a reference to the current item is returned and the iterator moves up one spot.

How does this happen? The Iterator saves the current position of the iterator. When the next() method is called, the old position is returned and the next position is saved as the current position of the iterator. Each time next() is called, the next position becomes the current position.

iteratorone.java



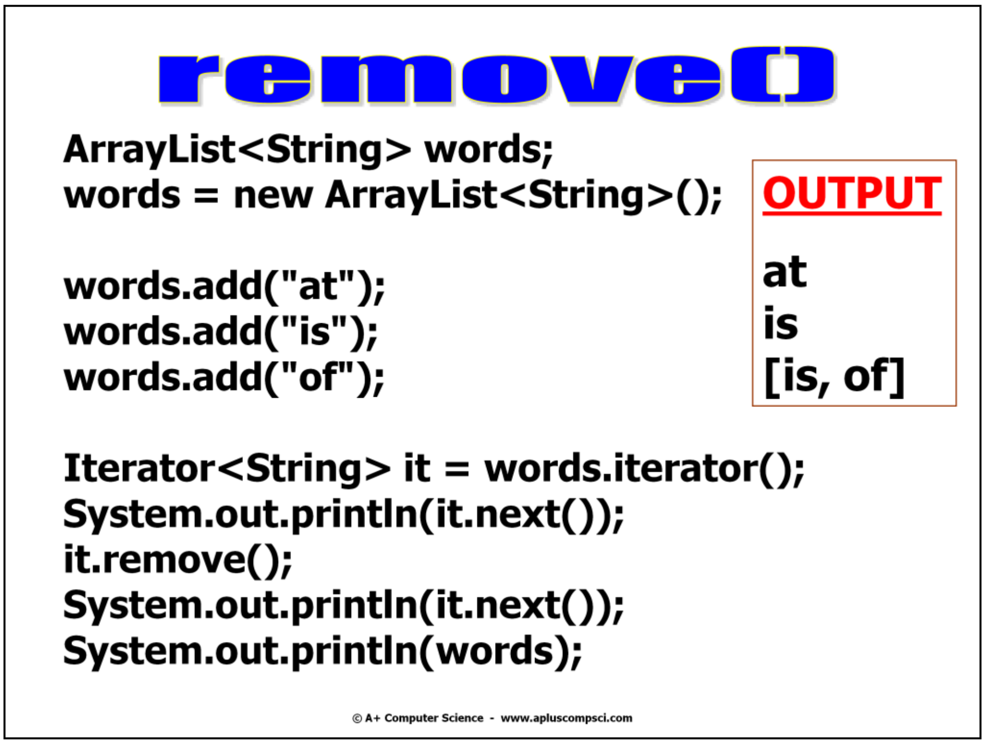
hasNext()

```
ArrayList<String> words;
words = new ArrayList<String>();

words.add("at");
words.add("is");
words.add("of");
words.add("us");

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

OUTPUT

```
at
is
of
us
```

hasnext.java

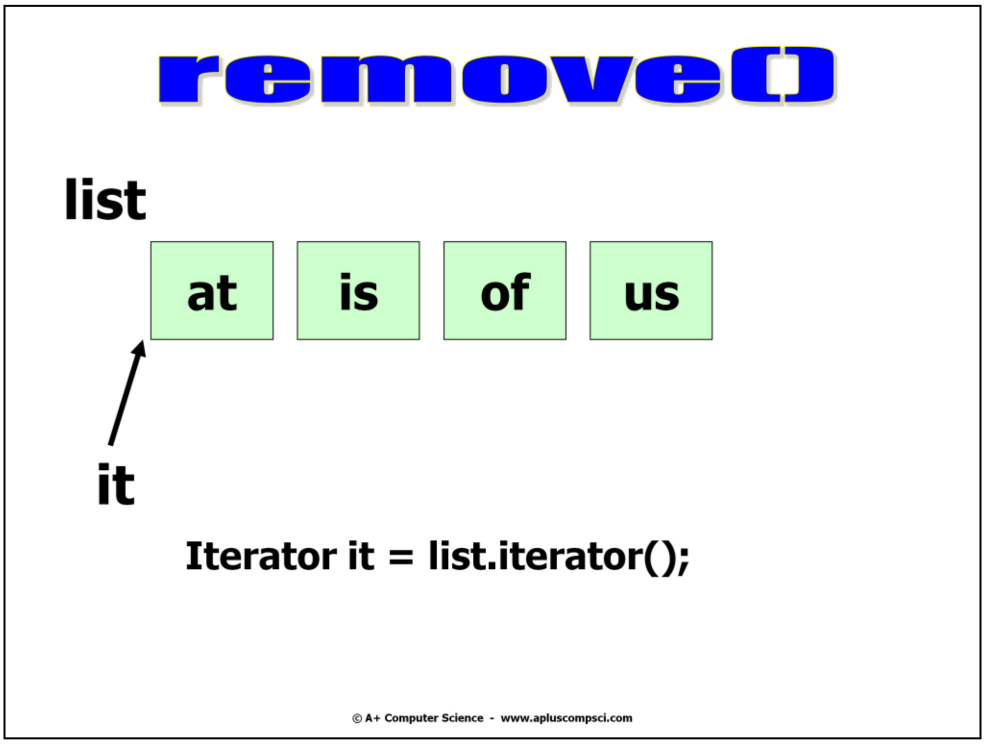


remove()

```
ArrayList<String> words;
words = new ArrayList<String>();

words.add("at");
words.add("is");
words.add("of");

Iterator<String> it = words.iterator();
System.out.println(it.next());
it.remove();
System.out.println(it.next());
System.out.println(words);
```

OUTPUT

```
at
is
[is, of]
```



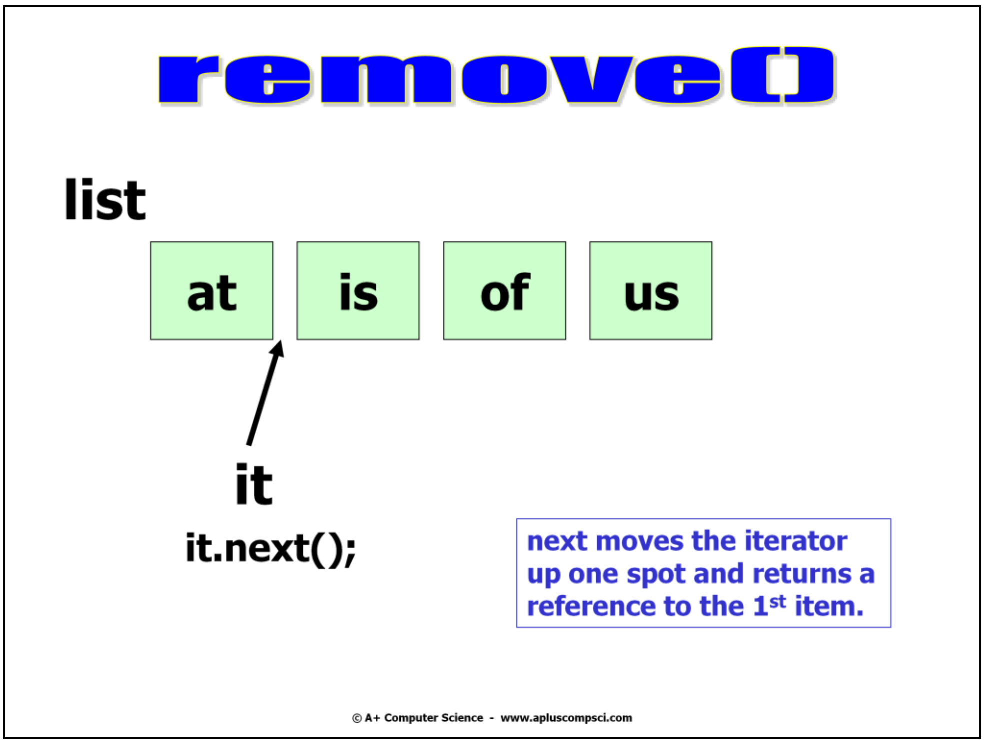
remove()

Iterator it = list.iterator();

© A+ Computer Science - www.apluscompsci.com

When the `next()` method is called, a reference to the current item is returned and the iterator moves up one spot.

How does this happen? The Iterator saves the current position of the iterator. When the next() method is called, the old position is returned and the next position is saved as the current position of the iterator. Each time next() is called, the next position becomes the current position.



remove()

list

it.next();

next moves the iterator up one spot and returns a reference to the 1st item.

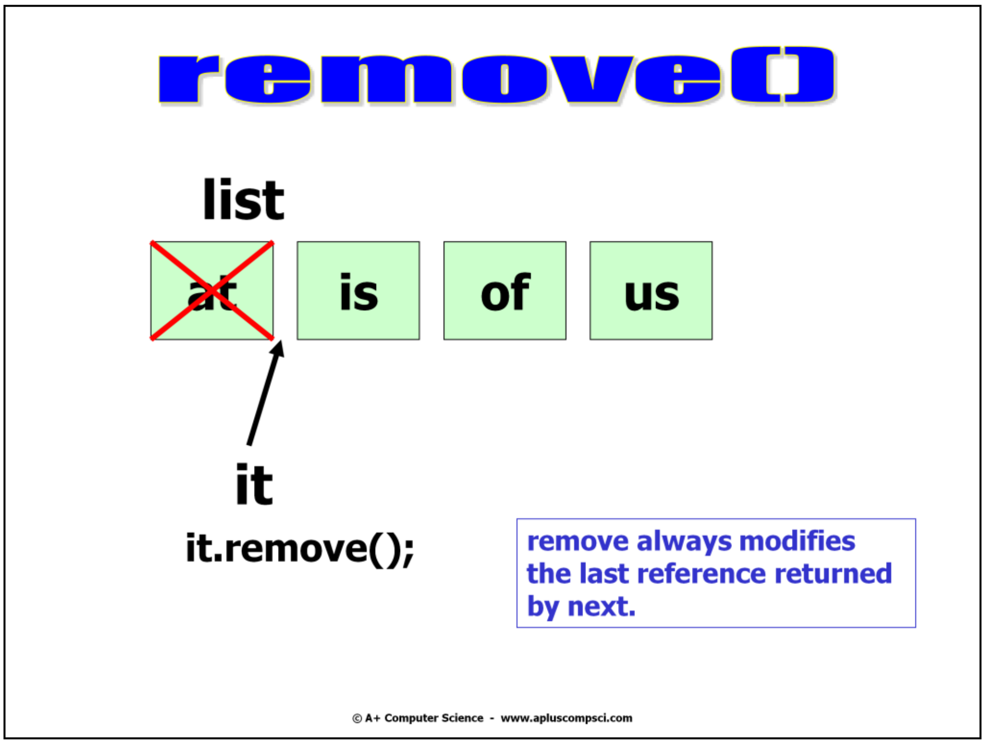
© A+ Computer Science - www.apluscompsci.com

at
is
of
us
it

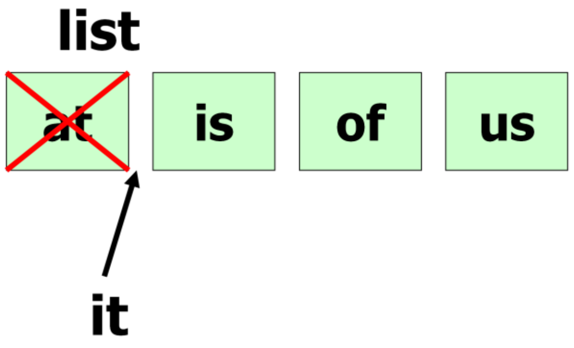
When the `next()` method is called, a reference to the current item is returned and the iterator moves up one spot.

How does this happen? The Iterator saves the current position of the iterator. When the next() method is called, the old position is returned and the next position is saved as the current position of the iterator. Each time next() is called, the next position becomes the current position.

In the example above, `next()` causes the iterator to slide past `"at"` and stop in front of `"is"`. `next()` returns the reference to `"at"`.



remove()



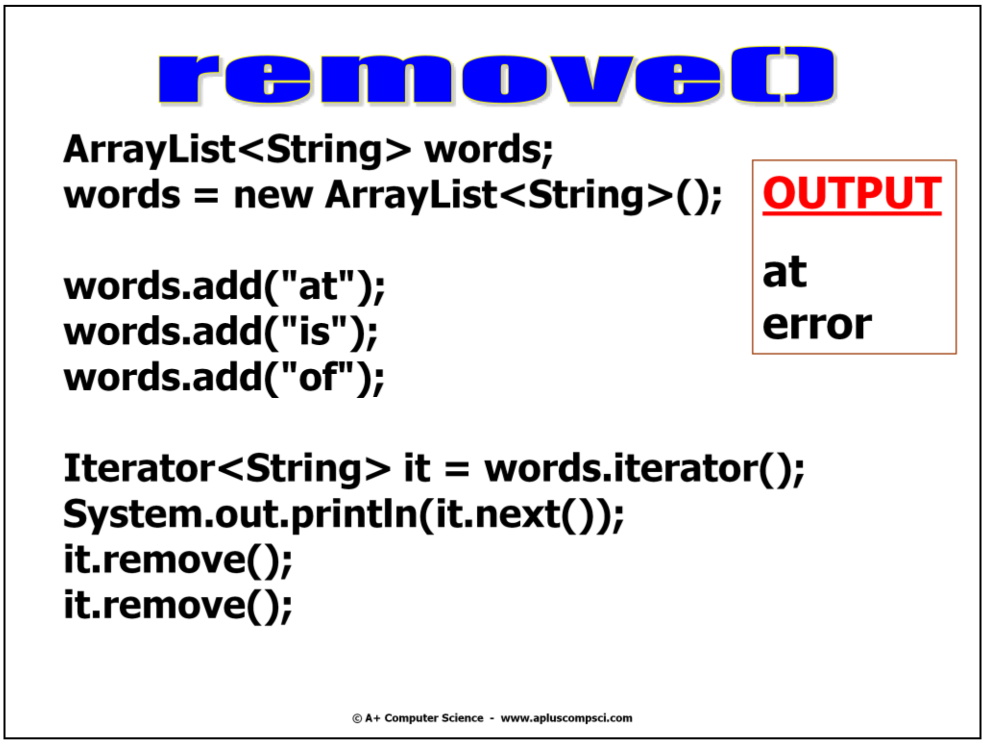
it.remove();

remove always modifies the last reference returned by next.

© A+ Computer Science - www.apluscompsci.com

Remove always removes the last reference returned by a call to `next()` or `previous()`. Remove can only be called after a call to `next()` or `previous()`.

```
next()
remove()
remove()   //blows up
```



remove()

```
ArrayList<String> words;
words = new ArrayList<String>();

words.add("at");
words.add("is");
words.add("of");

Iterator<String> it = words.iterator();
System.out.println(it.next());
it.remove();
it.remove();
```

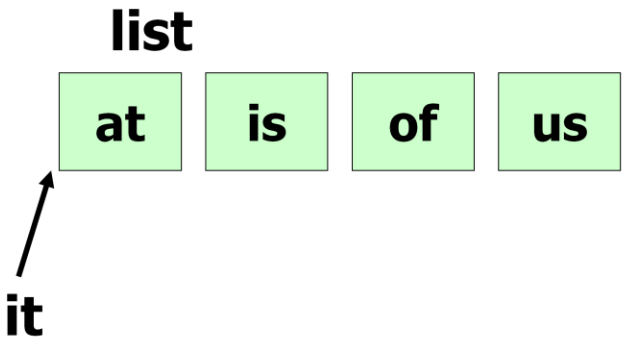
OUTPUT

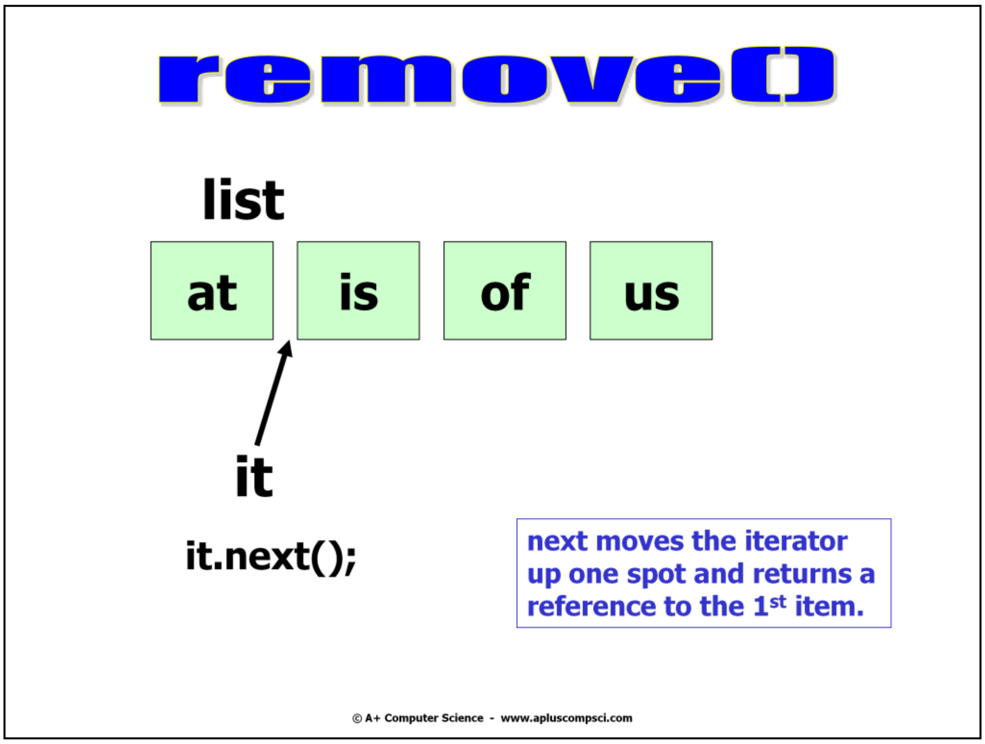
```
at
error
```

Remove always removes the last reference returned by a call to `next()` or `previous()`. Remove can only be called after a call to `next()` or `previous()`.

```
next()
remove()
remove()   //blows up
```

remove()





remove()

it.next();

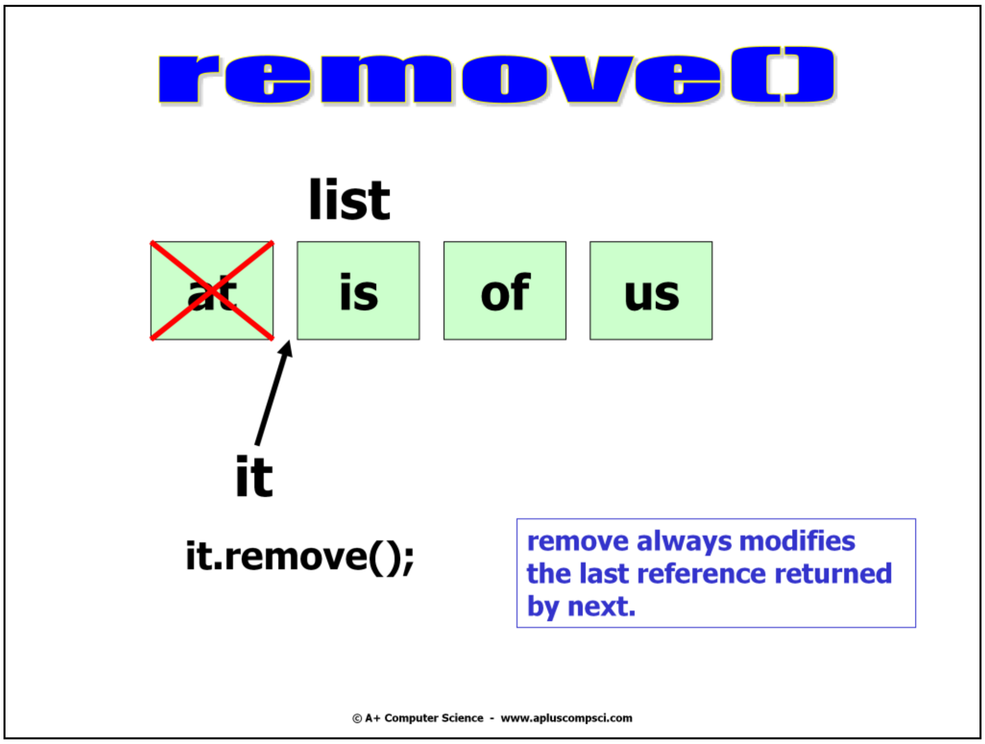
next moves the iterator up one spot and returns a reference to the 1st item.

© A+ Computer Science - www.apluscompsci.com

When the `next()` method is called, a reference to the current item is returned and the iterator moves up one spot.

How does this happen? The Iterator saves the current position of the iterator. When the next() method is called, the old position is returned and the next position is saved as the current position of the iterator. Each time next() is called, the next position becomes the current position.

In the example above, `next()` causes the iterator to slide past `"at"` and stop in front of `"is"`. `next()` returns the reference to `"at"`.



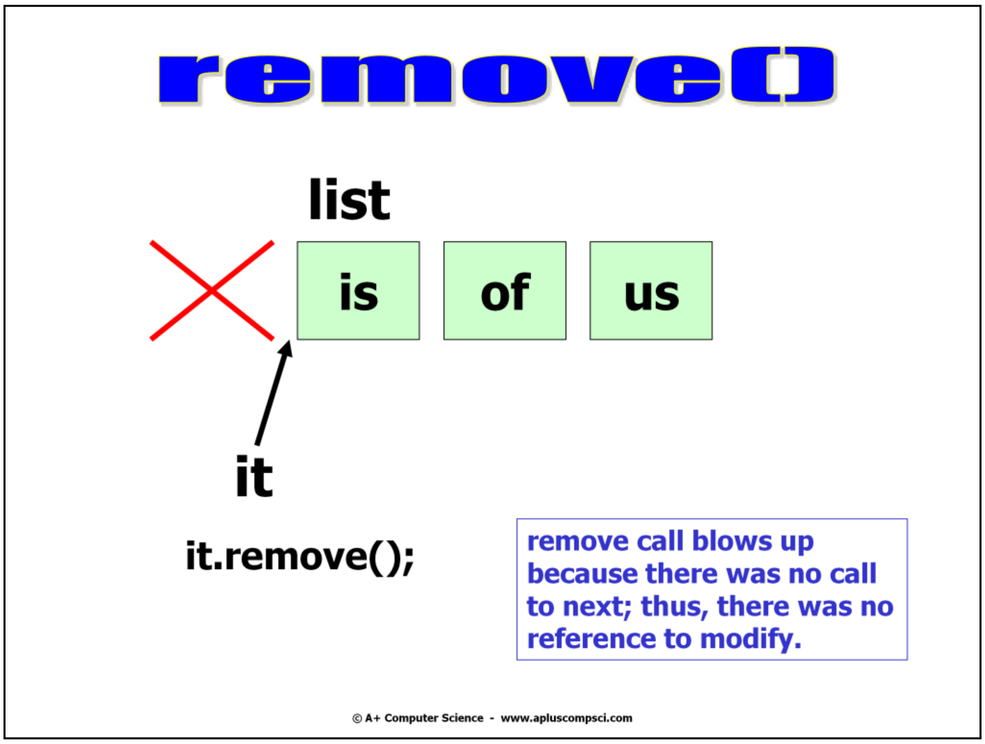
remove()

it.remove();

remove always modifies the last reference returned by next.

Remove always removes the last reference returned by a call to `next()` or `previous()`. Remove can only be called after a call to `next()` or `previous()`.

```
next()
remove()
remove()   //blows up
```



remove()

list
is
of
us
it

it.remove();

remove call blows up because there was no call to next; thus, there was no reference to modify.

Remove always removed the last reference returned by a call to `next()` or `previous()`. Remove can only be called after a call to `next()` or `previous()`.

```
next()
remove()
remove()   //blows up
```

removeone.java
removetwo.java

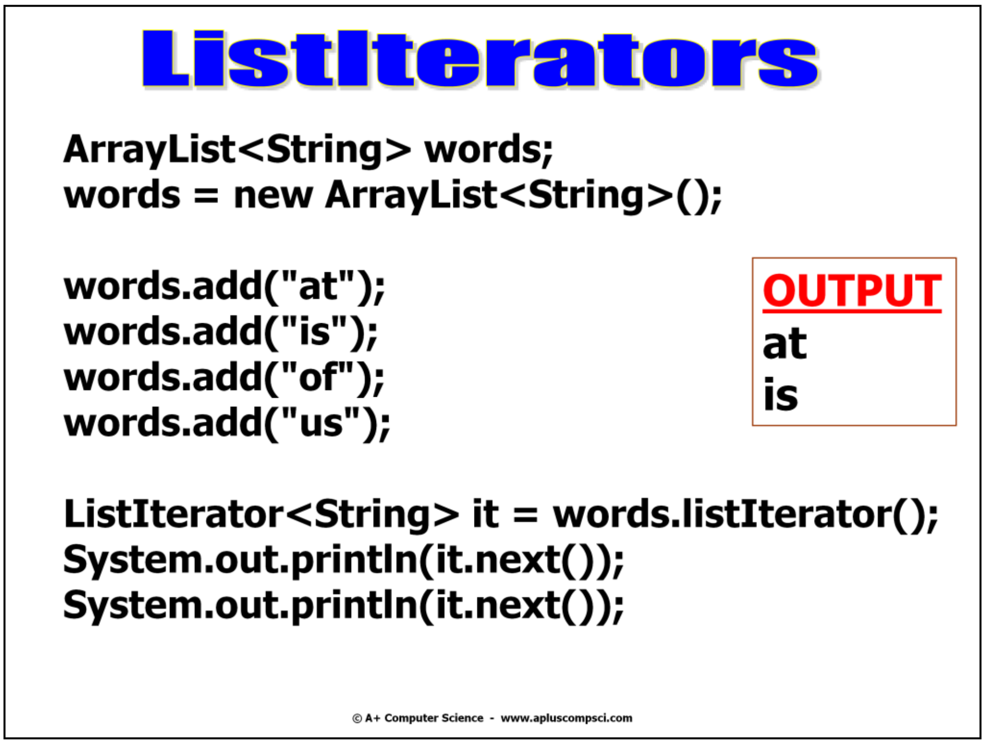
ListIterator Interface

ListIterator

frequently used methods

Name	Use
next()	returns a reference to the next item
remove()	removes the last ref returned by next or previous
hasNext()	checks to see there are more items
add()	adds in a new item
set()	sets the last ref returned by next or previous
previous()	goes back and returns a ref to prev item

```
import  java.util.ListIterator;
```



ListIterators

```
ArrayList<String> words;
words = new ArrayList<String>();

words.add("at");
words.add("is");
words.add("of");
words.add("us");

ListIterator<String> it = words.listIterator();
System.out.println(it.next());
System.out.println(it.next());
```

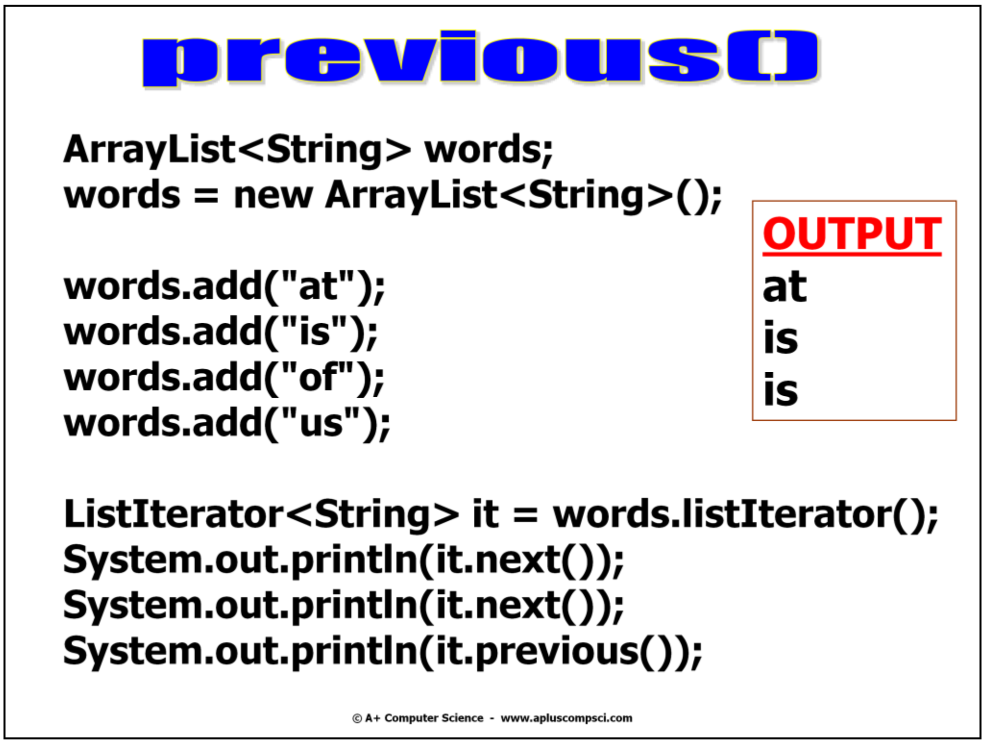
OUTPUT

```
at
is
```

When the `next()` method is called, a reference to the current item is returned and the iterator moves up one spot.

How does this happen? The Iterator saves the current position of the iterator. When the next() method is called, the old position is returned and the next position is saved as the current position of the iterator. Each time next() is called, the next position becomes the current position.

listiteratorone.java



previous()

```
ArrayList<String> words;
words = new ArrayList<String>();

words.add("at");
words.add("is");
words.add("of");
words.add("us");

ListIterator<String> it = words.listIterator();
System.out.println(it.next());
System.out.println(it.next());
System.out.println(it.previous());
```

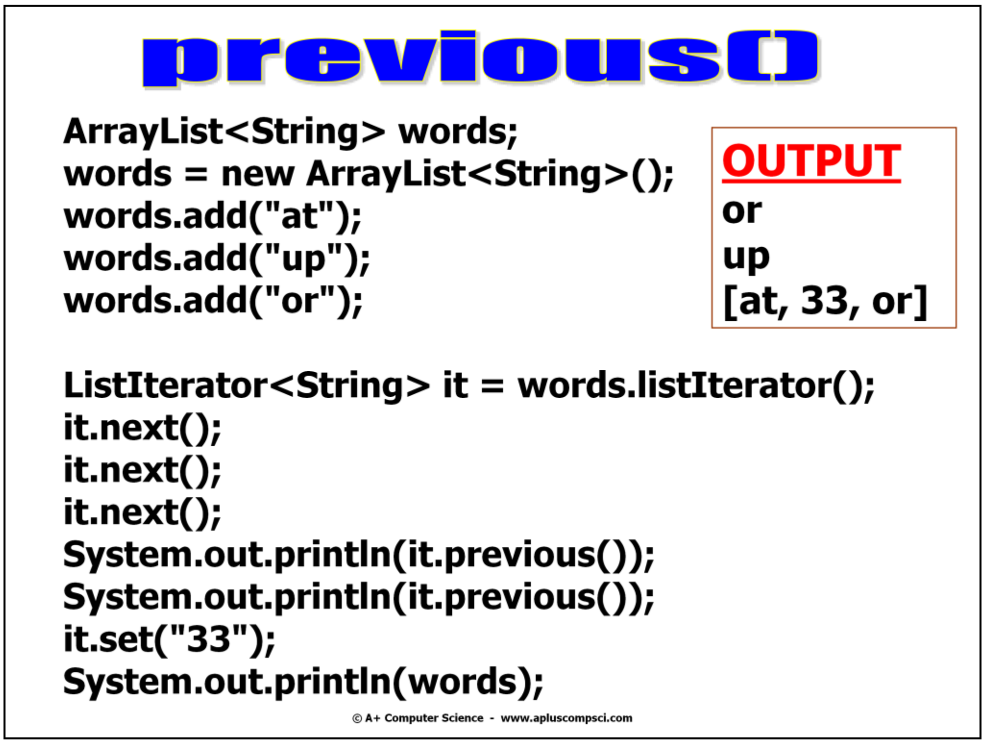
OUTPUT

```
at
is
is
```

When the `next()` method is called, the next reference in the list is returned and the iterator moves up one spot.

When the `previous()` method is called, the previous reference in the list is returned and the iterator moves back one spot.

previousone.java



previous()

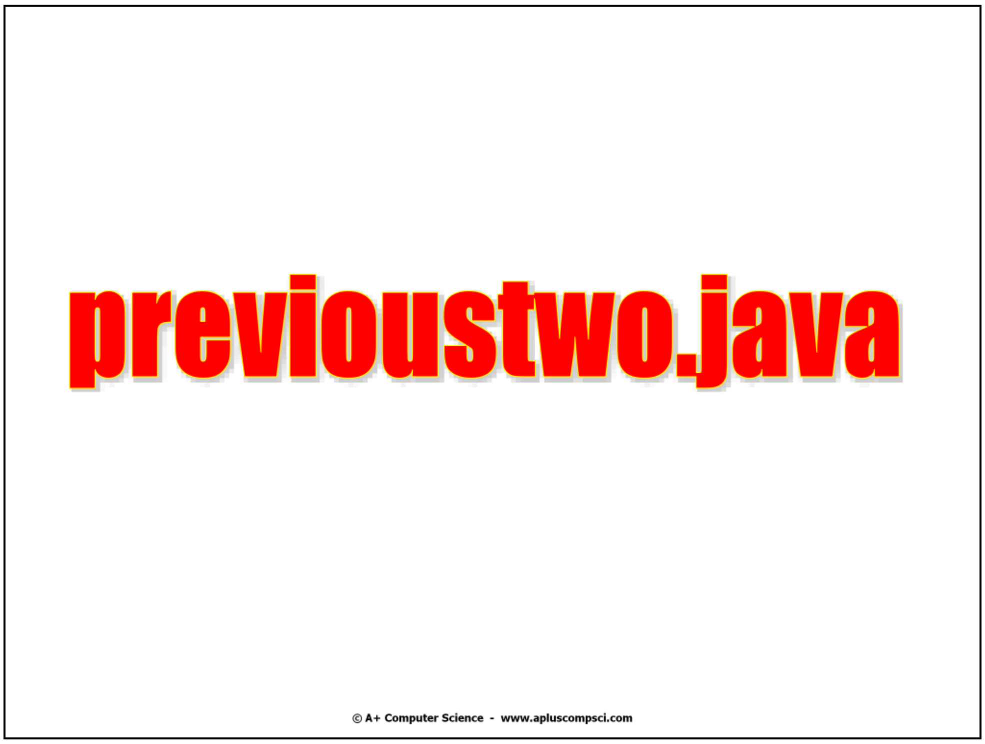
```
ArrayList<String> words;
words = new ArrayList<String>();
words.add("at");
words.add("up");
words.add("or");

ListIterator<String> it = words.listIterator();
it.next();
it.next();
it.next();
System.out.println(it.previous());
System.out.println(it.previous());
it.set("33");
System.out.println(words);
```

OUTPUT

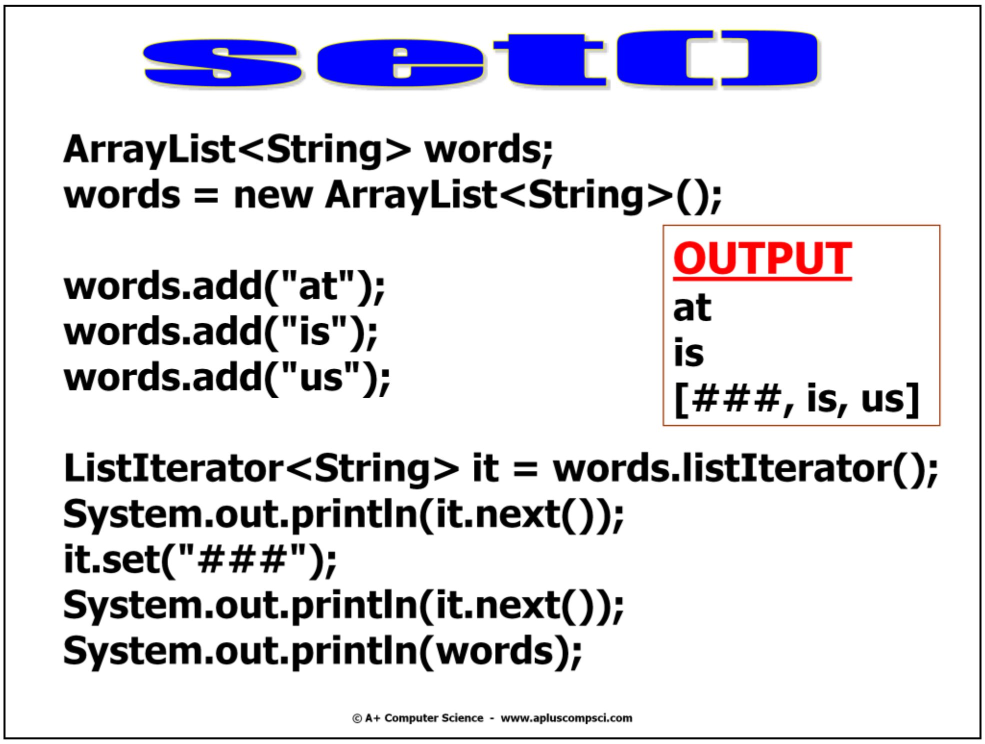
```
or
up
[at, 33, or]
```

When the `previous()` method is called, the previous reference in the list is returned and the iterator moves back one spot.

The `set()` method always modifies the last reference returned by a `next()` or `previous()` call.



previoustwo.java



set()

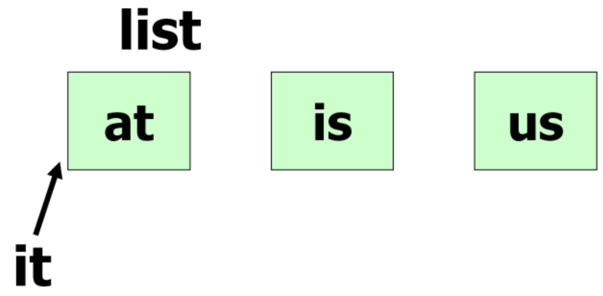
```
ArrayList<String> words;
words = new ArrayList<String>();

words.add("at");
words.add("is");
words.add("us");

ListIterator<String> it = words.listIterator();
System.out.println(it.next());
it.set("###");
System.out.println(it.next());
System.out.println(words);
```

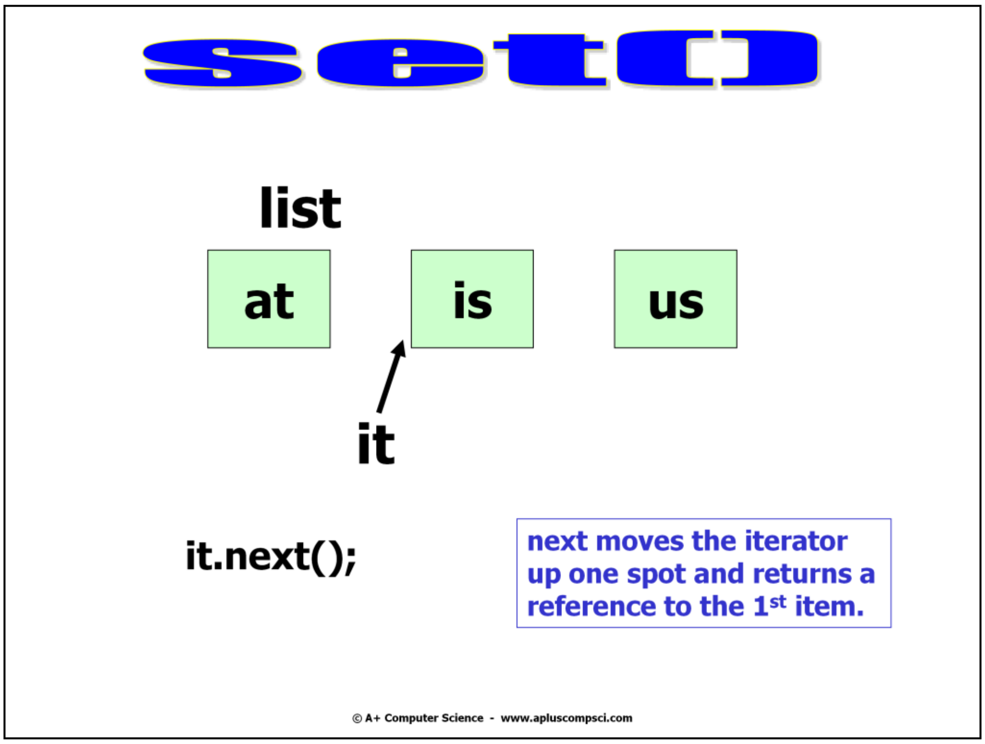
OUTPUT

```
at
is
[###, is, us]
```

The `set()` method always modifies the last reference returned by a `next()` or `previous()` call.

set()





set()

list

at is us

it

it.next();

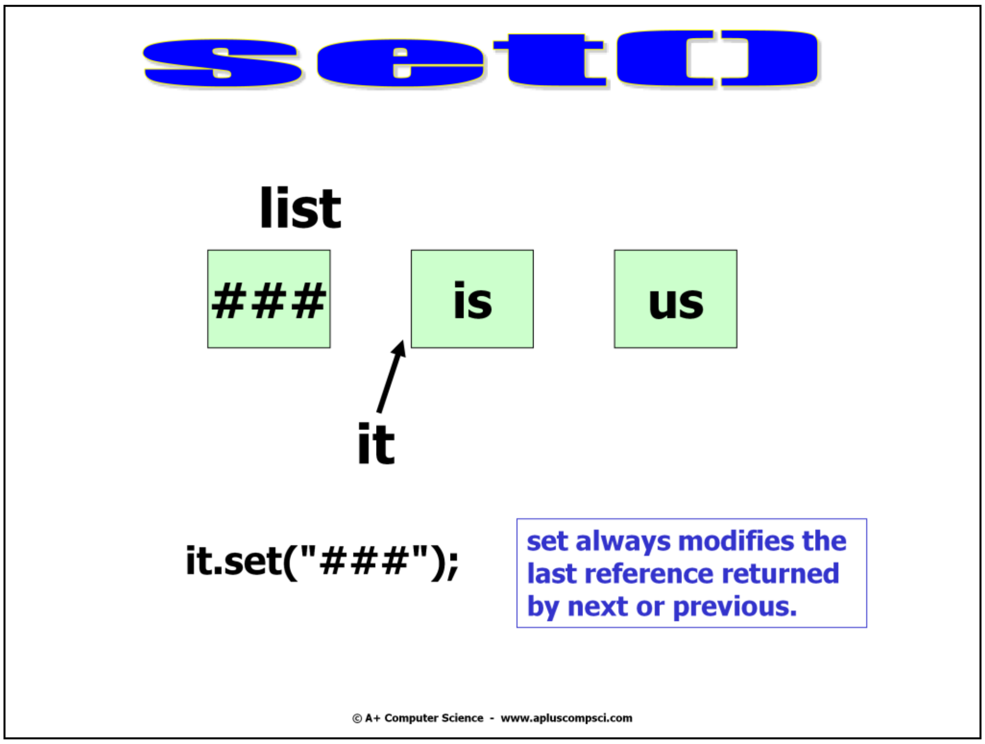
next moves the iterator up one spot and returns a reference to the 1st item.

© A+ Computer Science - www.apluscompsci.com

When the `next()` method is called, a reference to the current item is returned and the iterator moves up one spot.

How does this happen? The Iterator saves the current position of the iterator. When the next() method is called, the old position is returned and the next position is saved as the current position of the iterator. Each time next() is called, the next position becomes the current position.

In the example above, `next()` causes the iterator to slide past `"at"` and stop in front of `"is"`. `next()` returns the reference to `"at"`.



set()

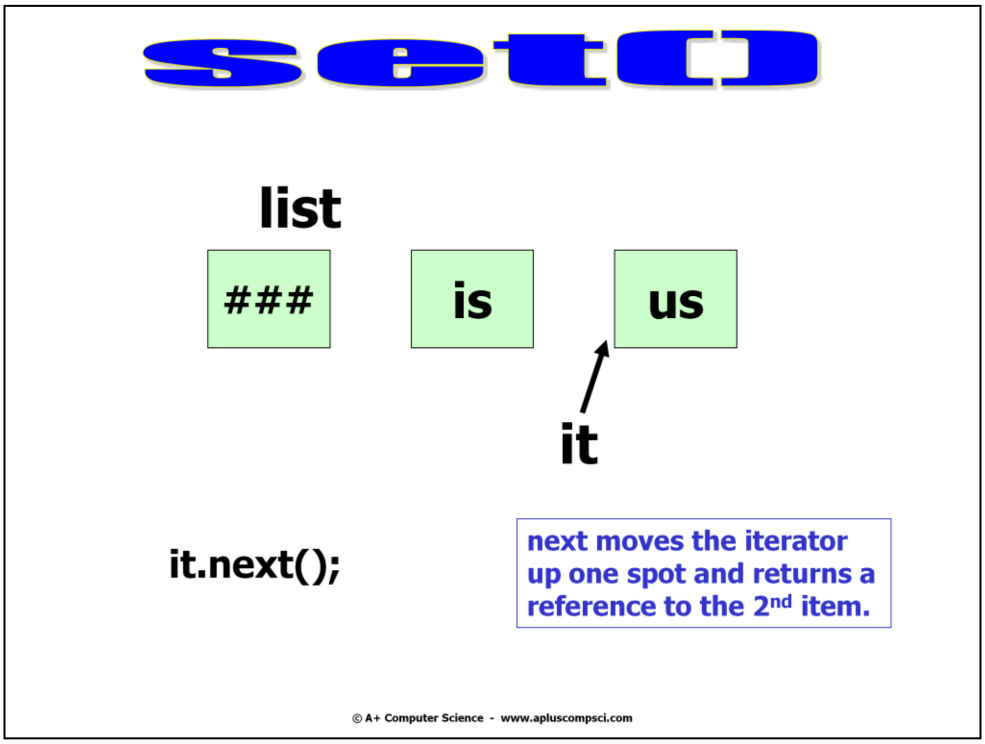
list
###
is
us
it

```
it.set("###");
```

set always modifies the last reference returned by next or previous.

The `set()` method always modifies the last reference returned by a `next()` or `previous()` call.

The set method can be called multiple times on the same spot.



set()

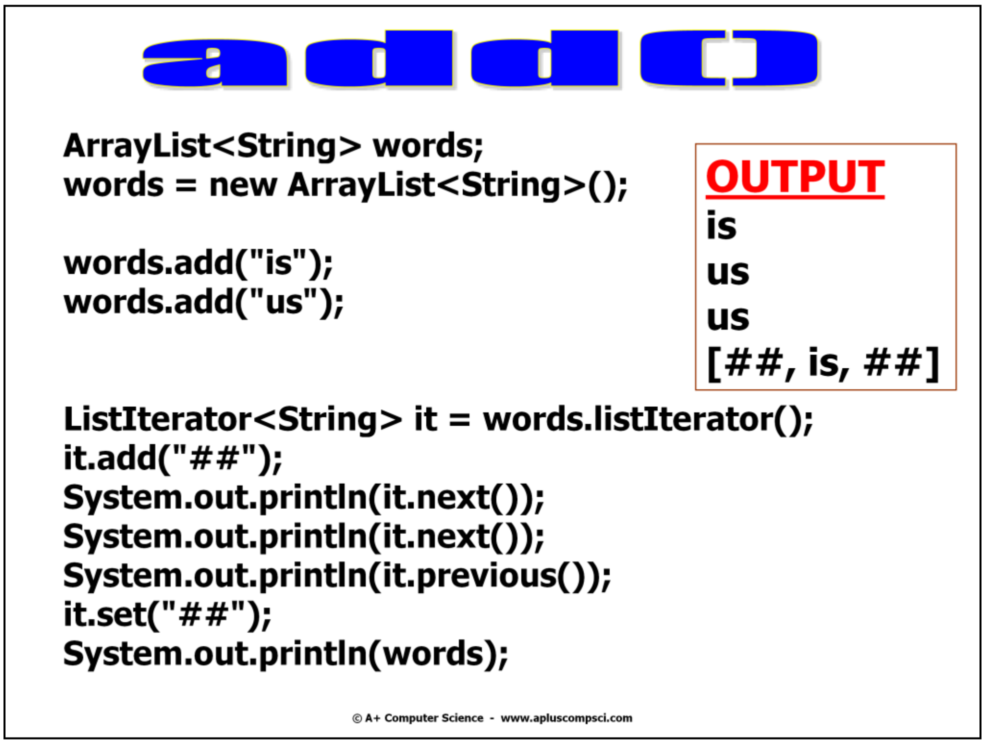
it.next();

next moves the iterator up one spot and returns a reference to the 2nd item.

When the `next()` method is called, a reference to the current item is returned and the iterator moves up one spot.

How does this happen? The Iterator saves the current position of the iterator. When the next() method is called, the old position is returned and the next position is saved as the current position of the iterator. Each time next() is called, the next position becomes the current position.

setone.java
settwo.java



add()

```
ArrayList<String> words;
words = new ArrayList<String>();

words.add("is");
words.add("us");


ListIterator<String> it = words.listIterator();
it.add("##");
System.out.println(it.next());
System.out.println(it.next());
System.out.println(it.previous());
it.set("##");
System.out.println(words);
```

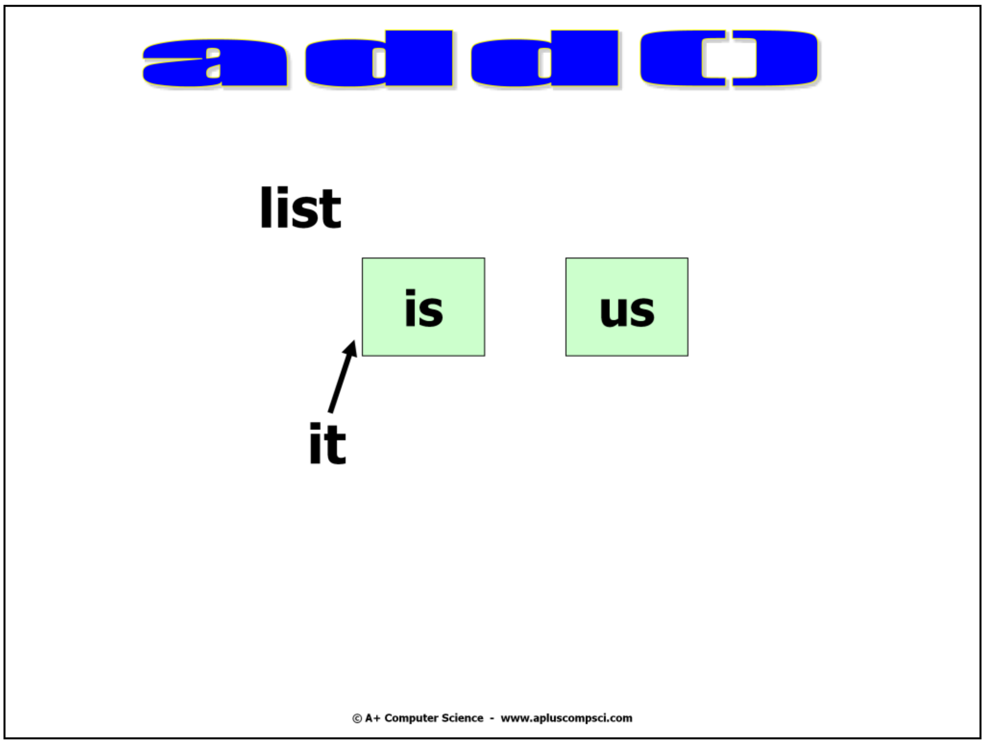
OUTPUT

```
is
us
us
[##, is, ##]
```

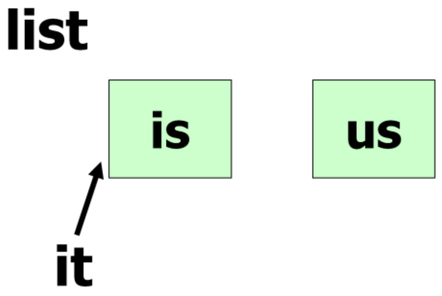
© A+ Computer Science - www.apluscompsci.com

An iterator essentially points to an area in front of each reference. It starts out pointing in front of the 1st reference.

The `add()` method always adds the new reference in front the iterator's current position.

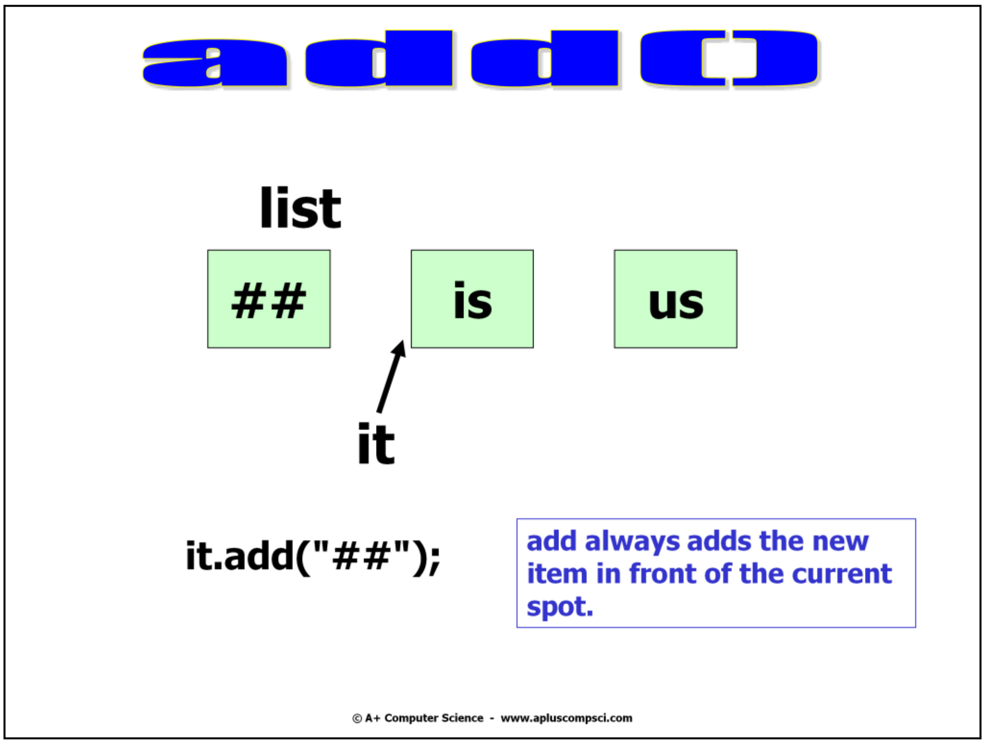


add()



An iterator essentially points to an area in front of each reference. It starts out pointing in front of the 1st reference.

The `add()` method always adds the new reference in front the iterator's current position.



add()

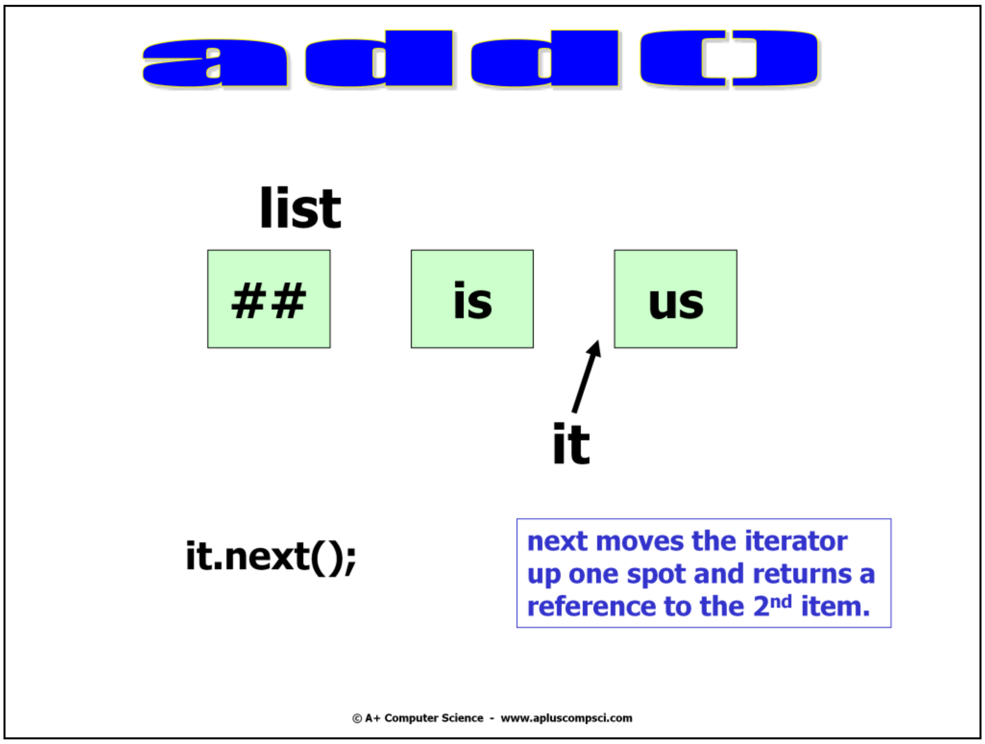
it.add("##");

add always adds the new item in front of the current spot.

© A+ Computer Science - www.apluscompsci.com

An iterator essentially points to an area in front of each reference. It starts out pointing in front of the 1st reference.

The `add()` method always adds the new reference in front the iterator's current spot/position.



add()

list
##
is
us
it

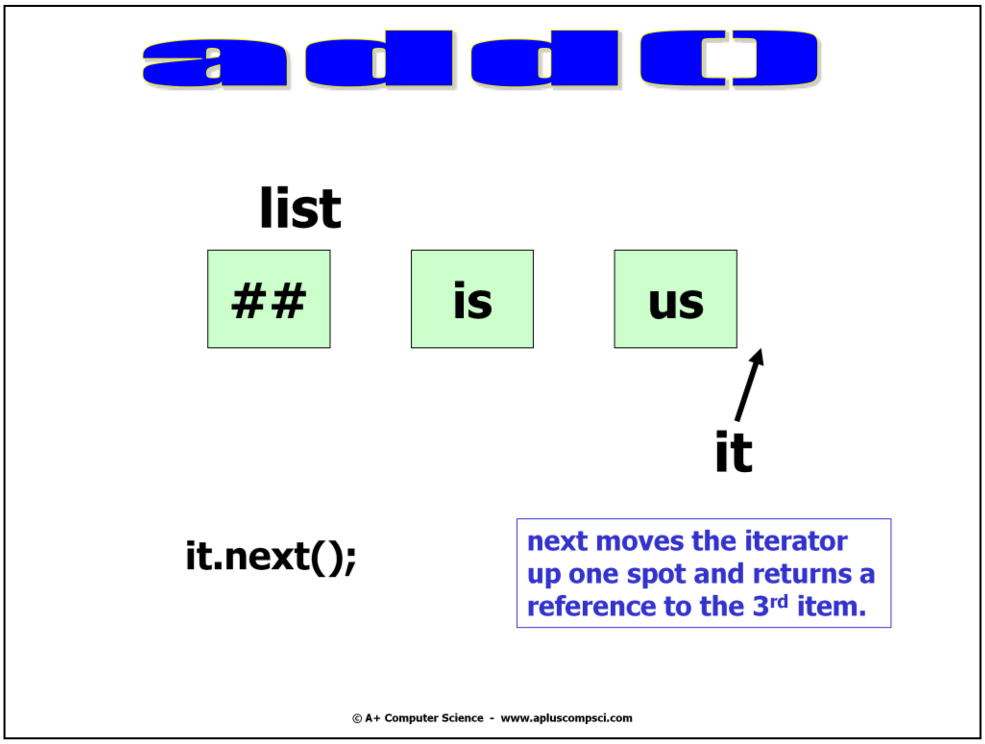
it.next();

next moves the iterator up one spot and returns a reference to the 2nd item.

© A+ Computer Science - www.apluscompsci.com

When the `next()` method is called, a reference to the current item is returned and the iterator moves up one spot.

How does this happen? The Iterator saves the current position of the iterator. When the next() method is called, the old position is returned and the next position is saved as the current position of the iterator. Each time next() is called, the next position becomes the current position.



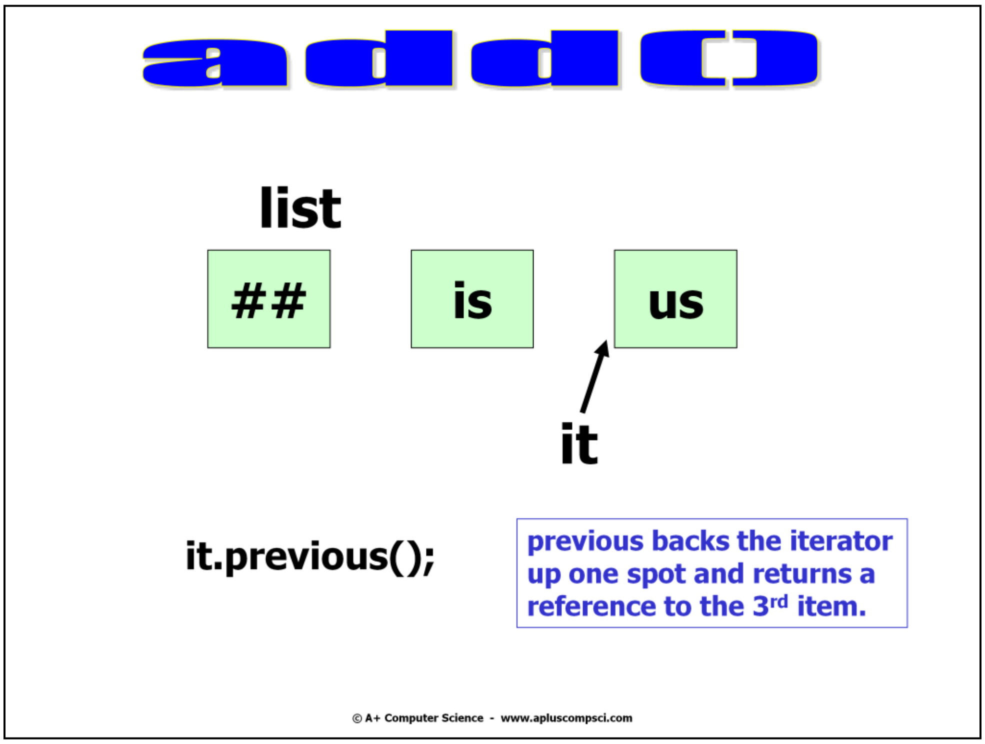
add()

it.next();

next moves the iterator up one spot and returns a reference to the 3rd item.

When the `next()` method is called, a reference to the current item is returned and the iterator moves up one spot.

How does this happen? The Iterator saves the current position of the iterator. When the next() method is called, the old position is returned and the next position is saved as the current position of the iterator. Each time next() is called, the next position becomes the current position.



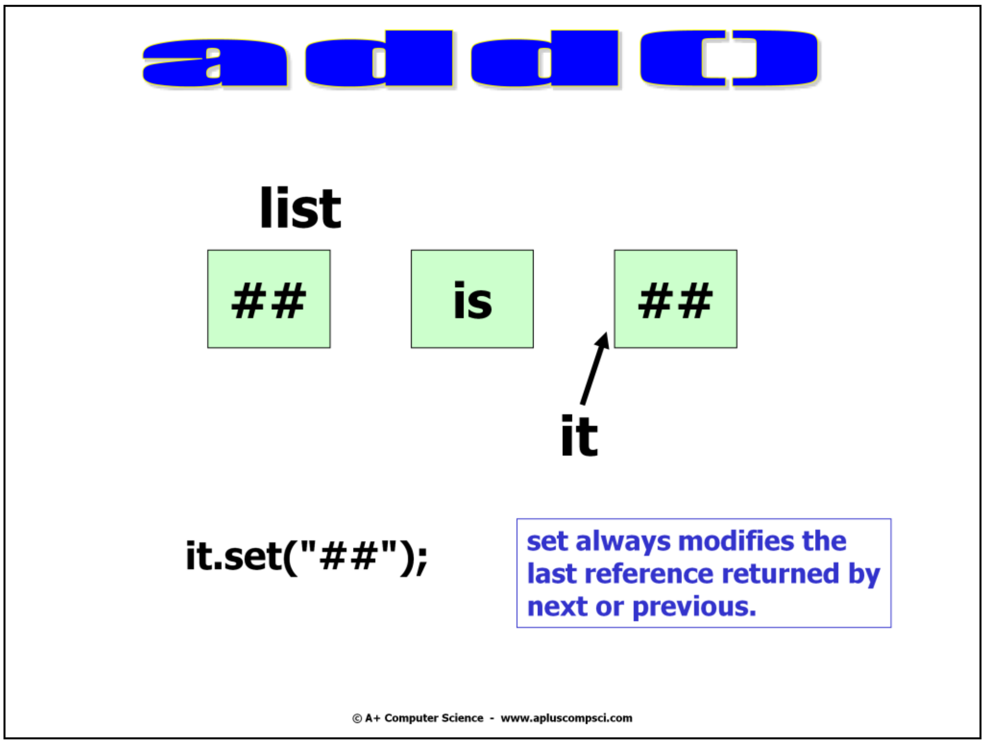
add()

it.previous();

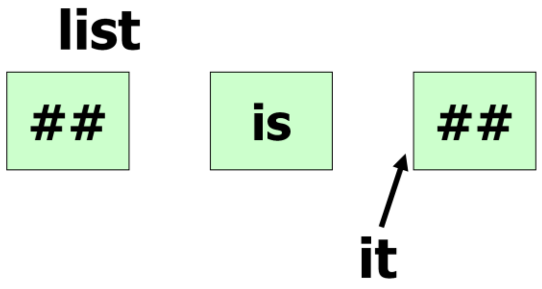
previous backs the iterator up one spot and returns a reference to the 3rd item.

© A+ Computer Science - www.apluscompsci.com

When the `previous()` method is called, the previous reference in the list is returned and the iterator moves back one spot.



add()

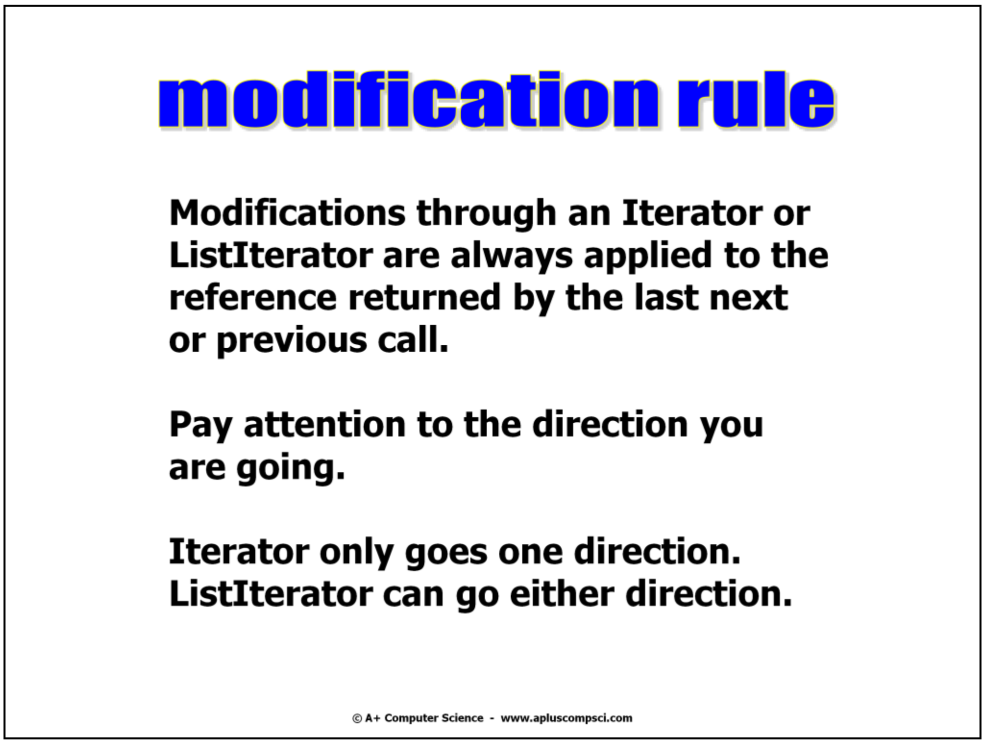


it.set("##");

set always modifies the last reference returned by next or previous.

The `set()` method always modifies the last reference returned by a `next()` or `previous()` call.

addone.java
addtwo.java



modification rule

Modifications through an Iterator or ListIterator are always applied to the reference returned by the last next or previous call.

Pay attention to the direction you are going.

Iterator only goes one direction.
ListIterator can go either direction.

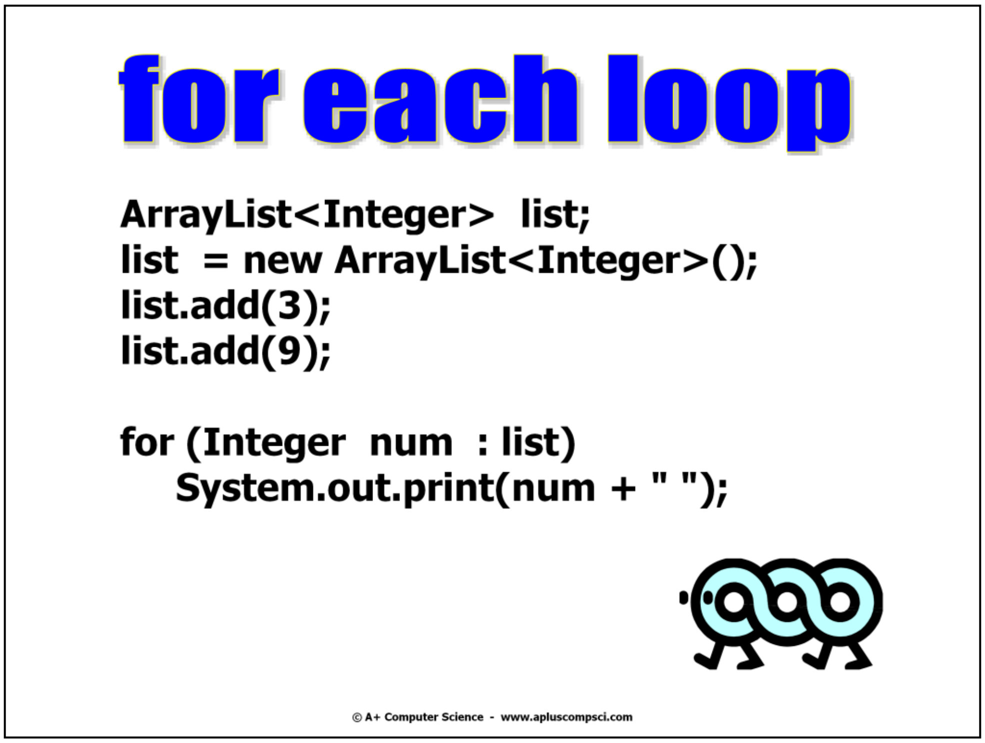
the for each loop

traditional for loop

```
int[] array = {4,5,6,7};
int sum = 0;

for(int i=0; i<array.length; i++)
{
  sum += array[i];
}
```

for each loop

```
int array[] = {4,9,6,2,3};
int sum = 0;

for (int num  : array)
   sum = sum + num;
System.out.println(sum);
```



for each loop

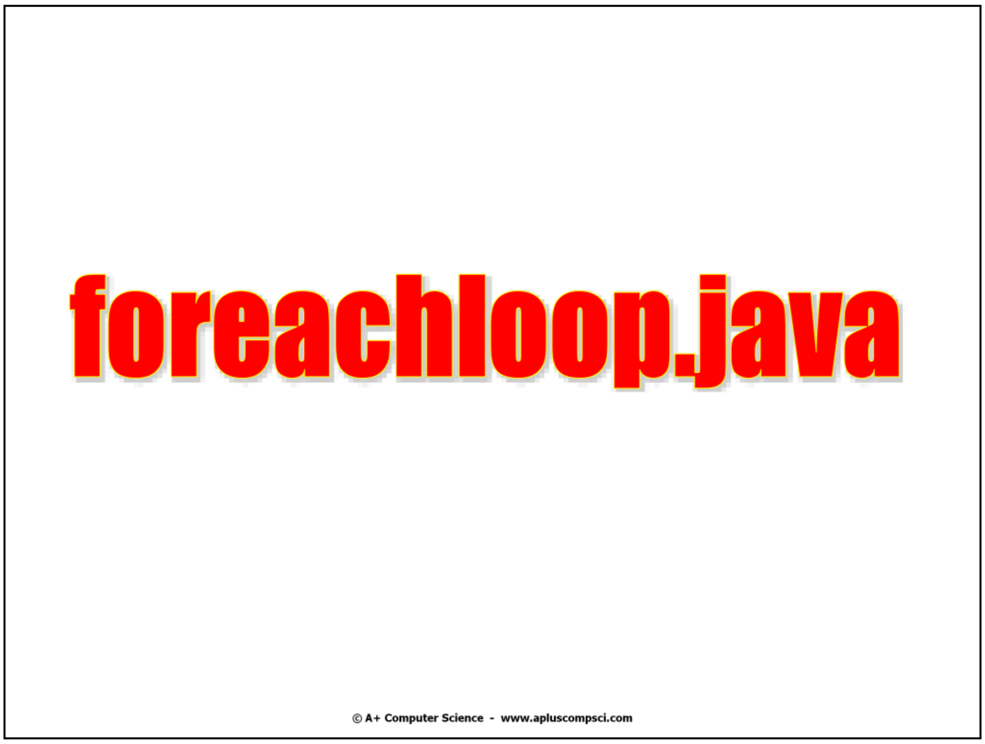
```
ArrayList<Integer>  list;
list  = new ArrayList<Integer>();
list.add(3);
list.add(9);

for (Integer  num  : list)
    System.out.print(num + " ");
```

for each loop

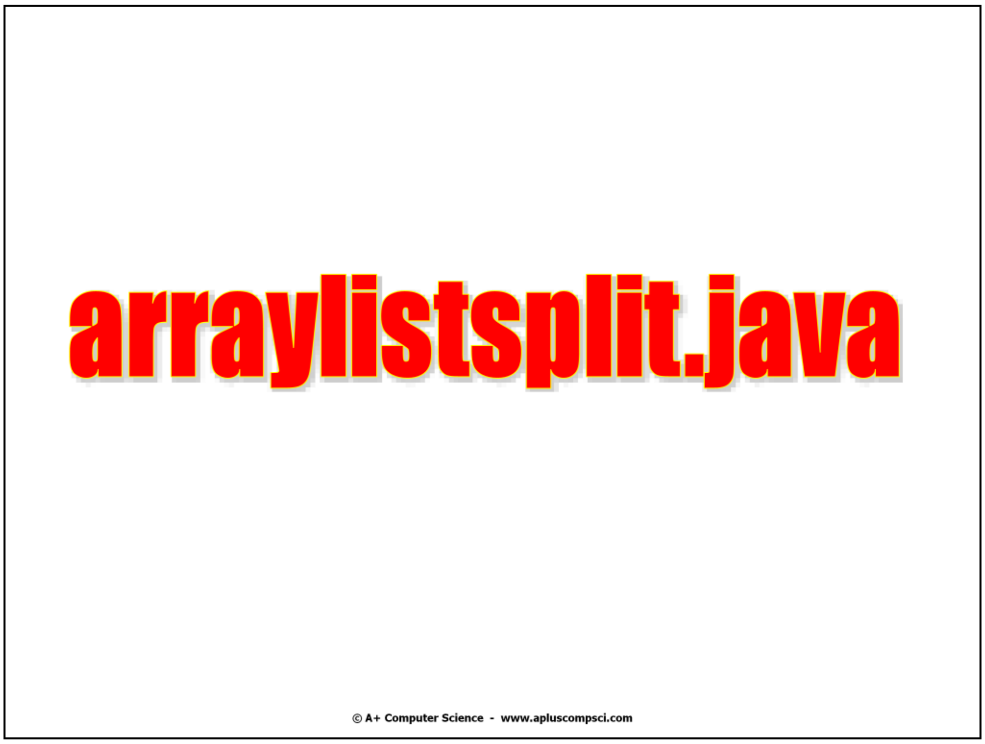
```
ArrayList<Integer>  list;
list  = new  ArrayList<Integer>();
list.add(3);
list.add(9);

for (int num  : list)
    System.out.print(num + " ");
```

old way

```
ArrayList list = new ArrayList();

//add stuff to list

Iterator it = list.iterator();
while(it.hasNext())
{
  System.out.println(it.next());
}
```



foreachloop.java



arraylistsplit.java

Start work on the labs