Collections

- Why Generic Collections?
- ArrayList
  - Access via Iterator / ListIterator
  - For-Each
  - User-defined class in Collection
  - Obtaining an Array from an ArrayList
- LinkedList
- Object Ordering
  - Writing your own Comparable

B.Senthil Kumar
Asst. Professor, CSE

Collections

- Algorithms
  - Sorting
  - Shuffling
  - Routine Data Manipulation
  - Searching
  - Composition
  - Finding Extreme Values

Why Generic Collections ?

- Prior to generics, all collections stored references of type Object.
- This allowed any type of reference to be stored in the collection.
- Unfortunately, a pre-generics collection stored Object references could easily lead to errors.

Why Generic Collections ?

- Two Limitations:
- First, it required that you, **ensure that only objects of the proper type** be stored in a specific collection.
  - Again the compiler had no way to know the type of object stored in the pre-generic Collections.
- The second problem with pre-generics collections is that when you retrieve a reference from the collection, you must ***manually cast*** that reference into the proper type.

```
Integer i = (Integer) itr.next();
```

Why Generic Collections ?

```
import java.util.*;
class OldStyle {
public static void main(String args[]) {
   ArrayList list = new ArrayList();
   list.add("one");
   list.add("two");
   list.add("three");
   list.add("four");
   list.add(new Integer(100));  //invalid, error
   Iterator itr = list.iterator();
   while(itr.hasNext()) {
      String str = (String) itr.next(); // explicit cast
      Integer i = (Integer) itr.next(); //run-time error
      System.out.println(str + " is " + str.length()
                                      + " chars long.");
      }
   }
}
```

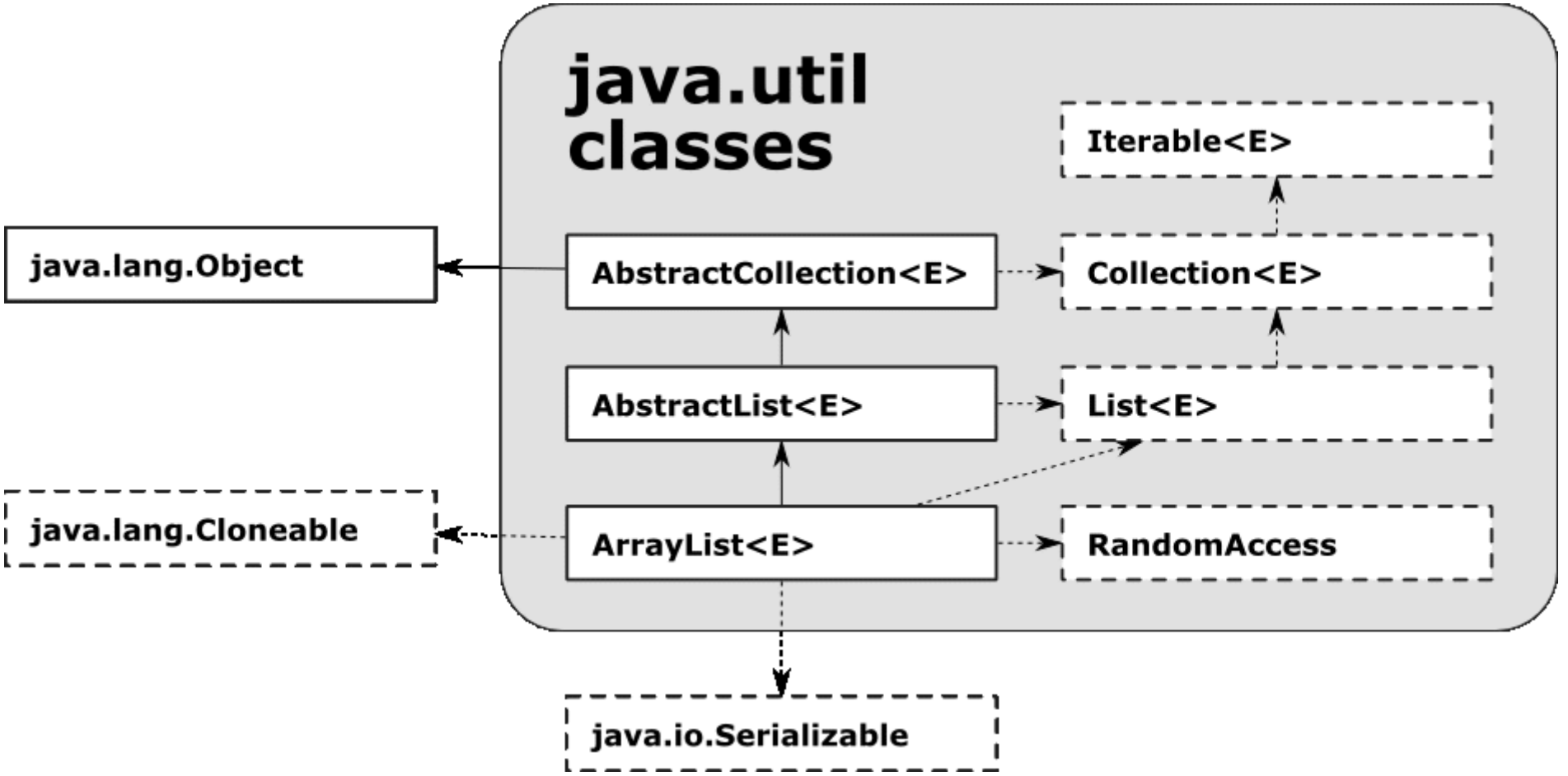
Why Generic Collections ?

```
import java.util.*;
class NewStyle {
public static void main(String args[]) {
   ArrayList<String> list = new ArrayList<String>();
   list.add("one");
   list.add("two");
   list.add("three");
   list.add("four");
   Iterator<String> itr = list.iterator();
//Iterator<Integer> itr = list.iterator(); //Errror!
   while(itr.hasNext()) {
      String str = itr.next(); // no cast needed
      // compile-time, rather than run-time, error.
      Integer i = itr.next(); // this won't compile
      System.out.println(str + " is " + str.length()
                         + " chars long.");
      }
   }
}
```

Why Generic Collections ?

- The addition of generics fundamentally improves the usability and safety of collections because it:
  - Ensures that only references to objects of the proper type can actually be stored in a collection. Thus, a collection will always contain references of a known type.
  - Eliminates the need to cast a reference retrieved from a collection. Instead, a reference retrieved from a collection is automatically cast into the proper type. This prevents run-time errors due to invalid casts and avoids an entire category of errors.

ArrayList


java.util
classes
Iterable<E>
java.lang.Object
AbstractCollection<E>
Collection<E>
AbstractList<E>
List<E>
java.lang.Cloneable
ArrayList<E>
RandomAccess
java.io.Serializable

ArrayList

- ArrayList class extends AbstractList and implements the List interface.
- Can dynamically increase or decrease in size.
- ArrayList has the constructors shown here:
  - `ArrayList( )`
  - `ArrayList(Collection<? extends E> c)`
  - `ArrayList(int capacity)`

ArrayList

- How to create an ArrayList?
- To create an ArryList of `String` type:

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

- Similarly we can create ArrayList that accepts `int` elements.

```
ArrayList<Integer> list=new ArrayList<Integer>();
```

- To add an element:

```
alist.add(“Kumar”);
```

- To add an element at the specified position (index):

```
alist.add(3,“Kumar”);
```

ArrayList

- To remove an element of the specified index from a list:

```
alist.remove(2);
```

- To remove the specified object from the list:

```
alist.remove("Kumar");
```

- To returns a view of the portion of the list between the specified `fromIndex` and `toIndex`:

```
List<E> subList(int fromIndex, int toIndex);
```

```
alist.subList(1,3);
```

- To get the element at the specified position(index):

```
alist.get(3);
```

ArrayList

- To search an element from a list:

```
alist.contains("Kumar");      //returns boolean
```

- To sort the specified list into ascending order:

```
Collections.sort(alist);
```

- To sort the elements in a reverse order:

```
Collections.sort(alist, Collections.reverseOrder());
Collections.reverse(alist);
```

- To replace the element at the specified position in the list with the specified element:

```
alist.set(3,"Ramesh");
```

ArrayList

- To appends all of the elements in the specified collection to the end of this list:

```
boolean addAll(Collection<? extends E> c)
```

```
alist1.addAll(alist);
```

ArrayList – Example

```
import java.util.*;
class ArrTest
{
    public static void main(String args[])
    {
      ArrayList<String> alist=new ArrayList<String>();
      alist.add("India");
      alist.add("China");
      alist.add("US");
      alist.add("Germany");
      alist.add("Bhutan");
      System.out.println("Original List: "+alist);
      alist.add(3, "SriLanka");
      System.out.println("After add: "+alist);
      alist.remove("China");
      alist.remove(3);
      System.out.println("After remove: "+alist);
```

ArrayList – Example

```
        Collections.sort(alist);
        System.out.println("After Sort: "+alist);

    //Collections.sort(alist, Collections.reverseOrder());
        Collections.reverse(alist);
        System.out.println("After reverse: "+alist);

        System.out.println("Using get(): "+alist.get(2));
        ArrayList<String> alist2 = new ArrayList<String>
                                    (alist.subList(1, 3));
        System.out.println("After subList: "+alist2);

        alist2.set(0,"Nepal");
        System.out.println("After set: "+alist2);

        alist.addAll(alist2);
        System.out.println("After Combine: "+alist);
      }
}
```

ArrayList – Example

Output:

```
Original List: [India, China, US, Germany, Bhutan]
After add: [India, China, US, SriLanka, Germany, Bhutan]
After remove: [India, US, SriLanka, Bhutan]
After Sort: [Bhutan, India, SriLanka, US]
After reverse: [US, SriLanka, India, Bhutan]
Using get(): India
After subList: [SriLanka, India]
After set: [Nepal, India]
After Combine: [US, SriLanka, India, Bhutan, Nepal, India]
```

Access via Iterator

- Use the Iterator object provided by each Collection class, to access each element in the Collection.

```
Iterator<E> iterator()
```

  - 1. Obtain an iterator to the start of the collection by calling the collection's `iterator( )` method.
  - 2. Set up a loop that makes a call to `hasNext( )`. Have the loop iterate as long as `hasNext( )` returns true.
  - 3. Within the loop, obtain each element by calling `next( )`.

ListIterator

Method	Description
void add(E *obj*)	Inserts *obj* into the list in front of the element that will be returned by the next call to **next( )**.
boolean hasNext( )	Returns **true** if there is a next element. Otherwise, returns **false**.
boolean hasPrevious( )	Returns **true** if there is a previous element. Otherwise, returns **false**.
E next( )	Returns the next element. A **NoSuchElementException** is thrown if there is not a next element.
int nextIndex( )	Returns the index of the next element. If there is not a next element, returns the size of the list.
E previous( )	Returns the previous element. A **NoSuchElementException** is thrown if there is not a previous element.
int previousIndex( )	Returns the index of the previous element. If there is not a previous element, returns –1.
void remove( )	Removes the current element from the list. An **IllegalStateException** is thrown if **remove( )** is called before **next( )** or **previous( )** is invoked.
void set(E *obj*)	Assigns *obj* to the current element. This is the element last returned by a call to either **next( )** or **previous( )**.

Table 17-9 The Methods Defined by **ListIterator**

Access via ListIterator

- ListIterator is available only to those collections that implement the List interface.
- For collections that implement List, obtain an iterator by calling `listIterator( )`
- A list iterator has the ability to access the collection in either the *forward or backward* direction and allows to modify an element.

Access via Iterator

```
import java.util.*;
class ArrIter
{
    public static void main(String args[])
    {
      ArrayList<String> al = new ArrayList<String>();
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("D");
        al.add("F");
        System.out.print("Original contents of al: ");
        Iterator<String> itr = al.iterator();
        while(itr.hasNext()) {
            String element = itr.next();
            System.out.print(element + " ");
            }
```

Using ListIterator – Example

```
    System.out.println();
    ListIterator<String> litr = al.listIterator();
    while(litr.hasNext()) {
       String element = litr.next();
       litr.set(element + "+");
       }
    System.out.print("Modified list backwards: ");
    while(litr.hasPrevious()) {
       String element = litr.previous();
       System.out.print(element + " ");
       }
    }
}
```

```
Output:

Original contents of al: C A E B D F
Modified list backwards: F+ D+ B+ E+ A+ C+
```

For-Each

- The for-each version is often a more convenient if:
  - won’t be modifying the contents of a collection
  - obtaining elements in reverse order
- The ***for*** can cycle through any collection of objects that implement the Iterable interface.

For-Each

```
import java.util.*;
class ForEachDemo {
public static void main(String args[]) {
   ArrayList<Integer> vals = new ArrayList<Integer>();
   vals.add(1);
   vals.add(2);
   vals.add(3);
   vals.add(4);
   System.out.print("Original contents of vals: ");
   int sum = 0;
   for(int v : vals)
   {
      System.out.print(v + " ");
      sum += v;
      }
   System.out.println();
   System.out.println("Sum of values: " + sum);
  }
}
```

User-defined class in Collection

- The collections are not limited to the storage of built-in objects.
- The collections can store any type of object, including objects of classes that you create.

User-defined class in Collection

```
class Book
{
private String title;
private int price;
Book(String title, int price)
{
   this.title=title;
   this.price=price;
   }
@Override
public String toString() {
   return "Title is: "+this.title+"\n"
               +" Price is: "+this.price;
    }
}
```

User-defined class in Collection

```
class BookList {
public static void main(String args[]) {
   ArrayList<Book> al= new ArrayList<Book>();
   al.add(new Book("Java Fundamentals", 340));
   al.add(new Book("Data Structure", 140));
   al.add(new Book("Python Programming", 320));
   for (Book tmp: al)
      System.out.println(tmp);
  }
}
```

Output:

```
Title is: Java Fundamentals
 Price is: 340
Title is: Data Structure
 Price is: 140
Title is: Python Programming
 Price is: 320
```

Obtaining an Array

- When working with ArrayList, sometimes you need an actual array that contains the contents of the list.
- You can do this by calling toArray( ), which is defined by Collection.
- Several reasons exist why you might want to convert a collection into an array, such as:
  - To obtain faster processing times for certain operations
  - To pass an array to a method that is not overloaded to accept a collection
  - To integrate collection-based code with legacy code that does not understand collections

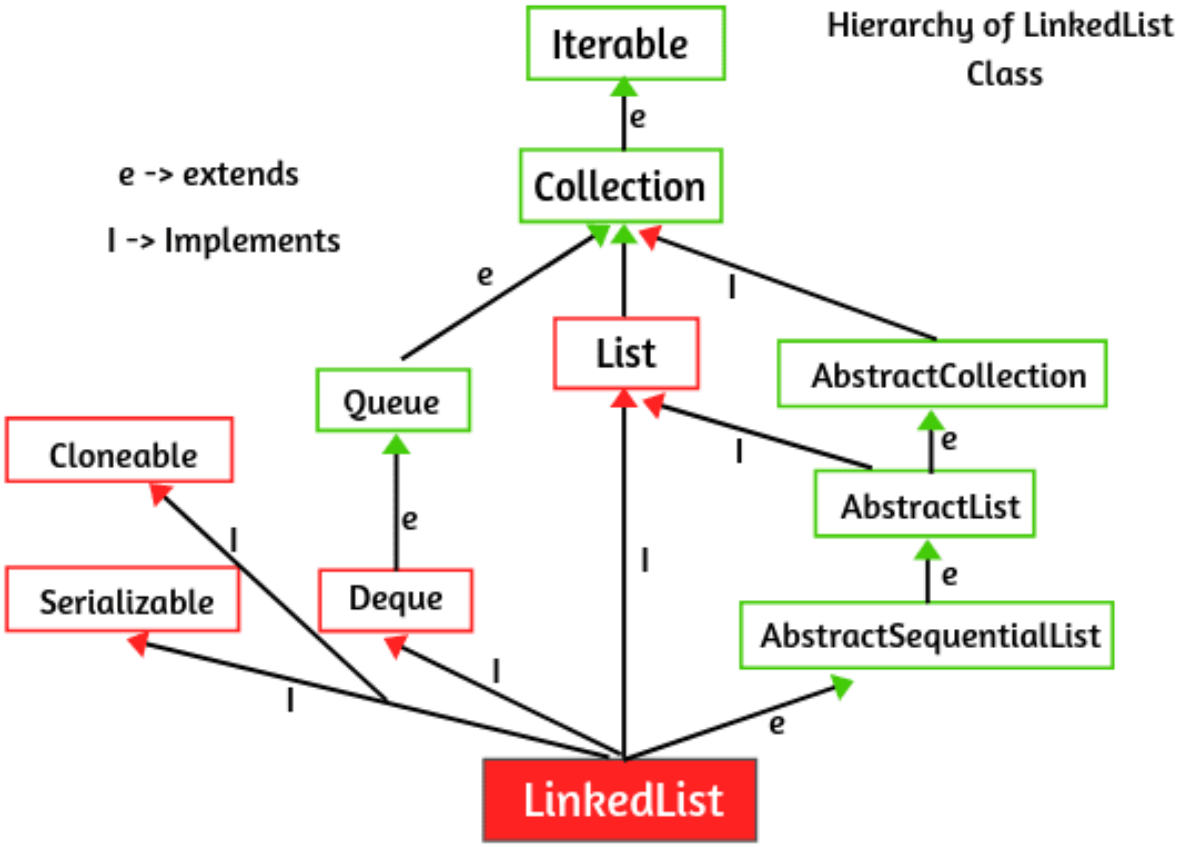
Obtaining an Array

```
import java.util.*;

class ArrayListToArray {
public static void main(String args[]) {

   ArrayList<Integer> al = new ArrayList<Integer>();

   al.add(1);
   al.add(2);
   al.add(3);
   al.add(4);
   System.out.println("Contents of al: " + al);

   Integer ia[] = new Integer[al.size()];
   ia = al.toArray(ia);
   int sum = 0;
```

Obtaining an Array

```
    for(int i : ia) sum += i;
    System.out.println("Sum is: " + sum);
    }
}
-----------------------------------------------------

Contents of al: [1, 2, 3, 4]
Sum is: 10
```

LinkedList

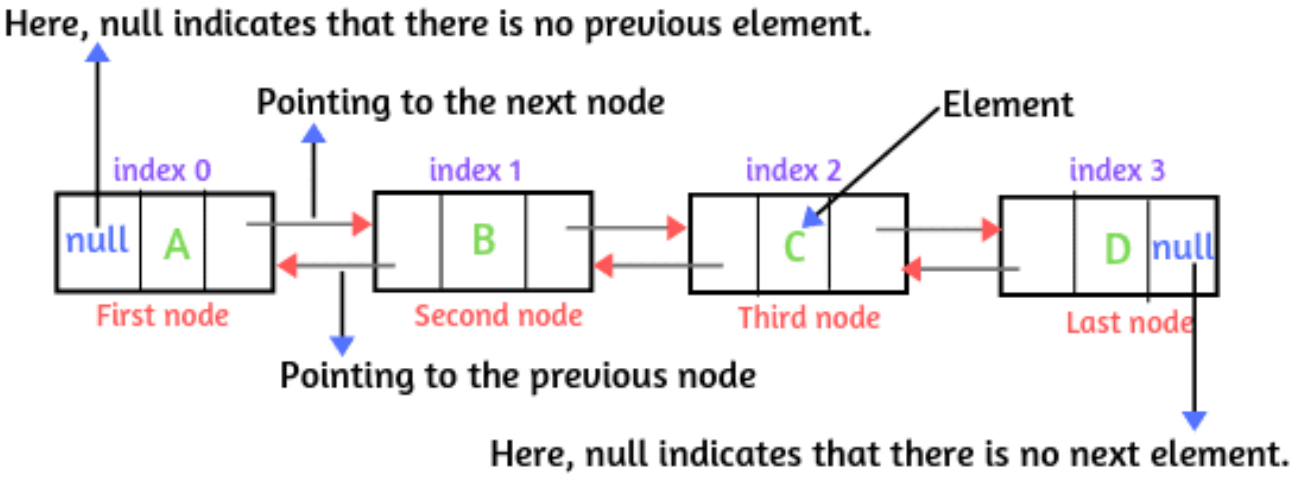

Hierarchy of LinkedList Class
e -> extends
I -> Implements
Iterable
e
Collection
e
I
List
Queue
AbstractCollection
Cloneable
e
I
AbstractList
e
Serializable
Deque
I
e
AbstractSequentialList
I
I
I
e
LinkedList

LinkedList

- Doubly-linked list implementation of the `List` and `Deque` interfaces.
- Implements all optional list operations, and permits all elements (including `null`).
- All of the operations perform as could be expected for a doubly-linked list.

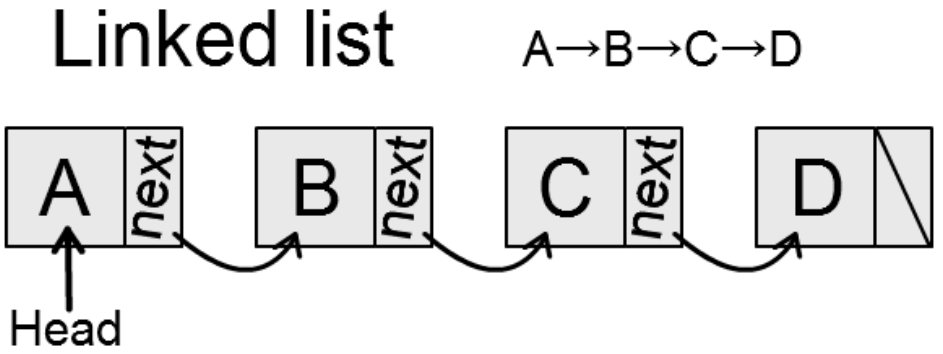
LinkedList

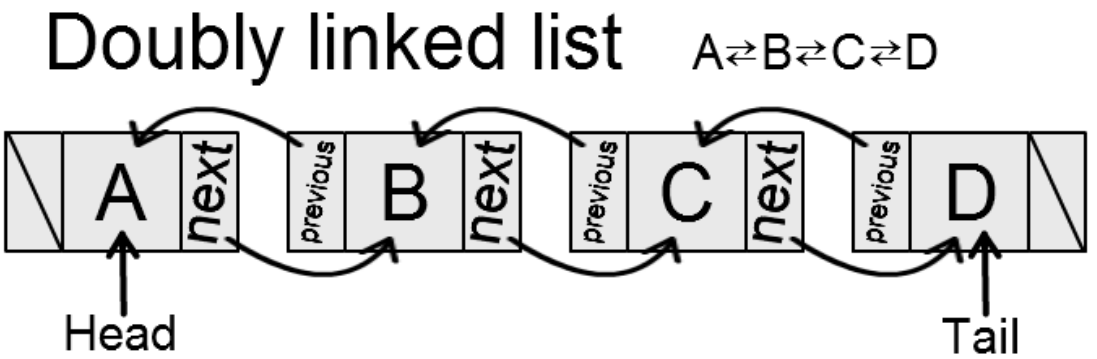
Representation of Java LinkedList Node



A array representation of linear Doubly LinkedList in Java

LinkedList

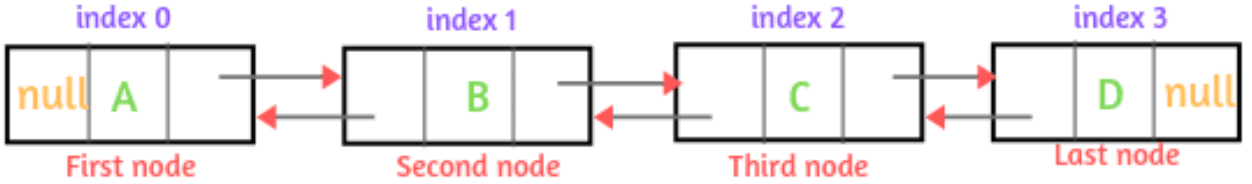




ListIterator Methods:
hasNext() - next()
hasPrevious() - previous()

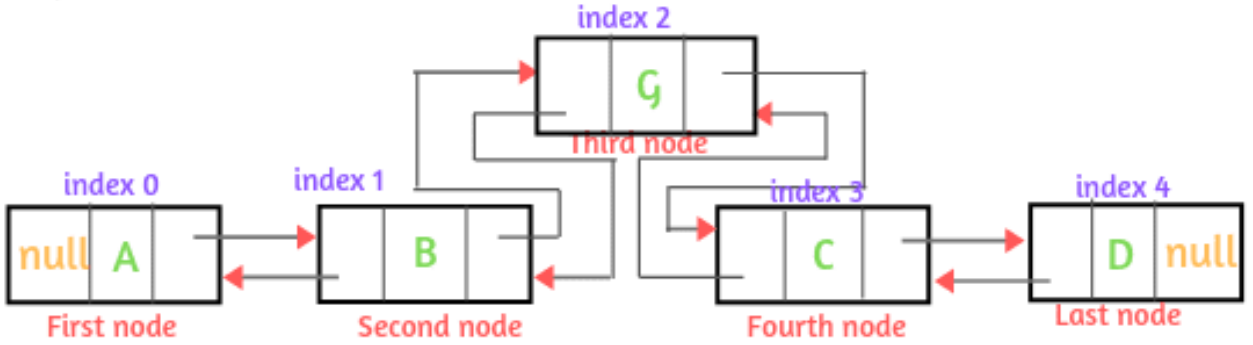
LinkedList

Initial LinkedList Data



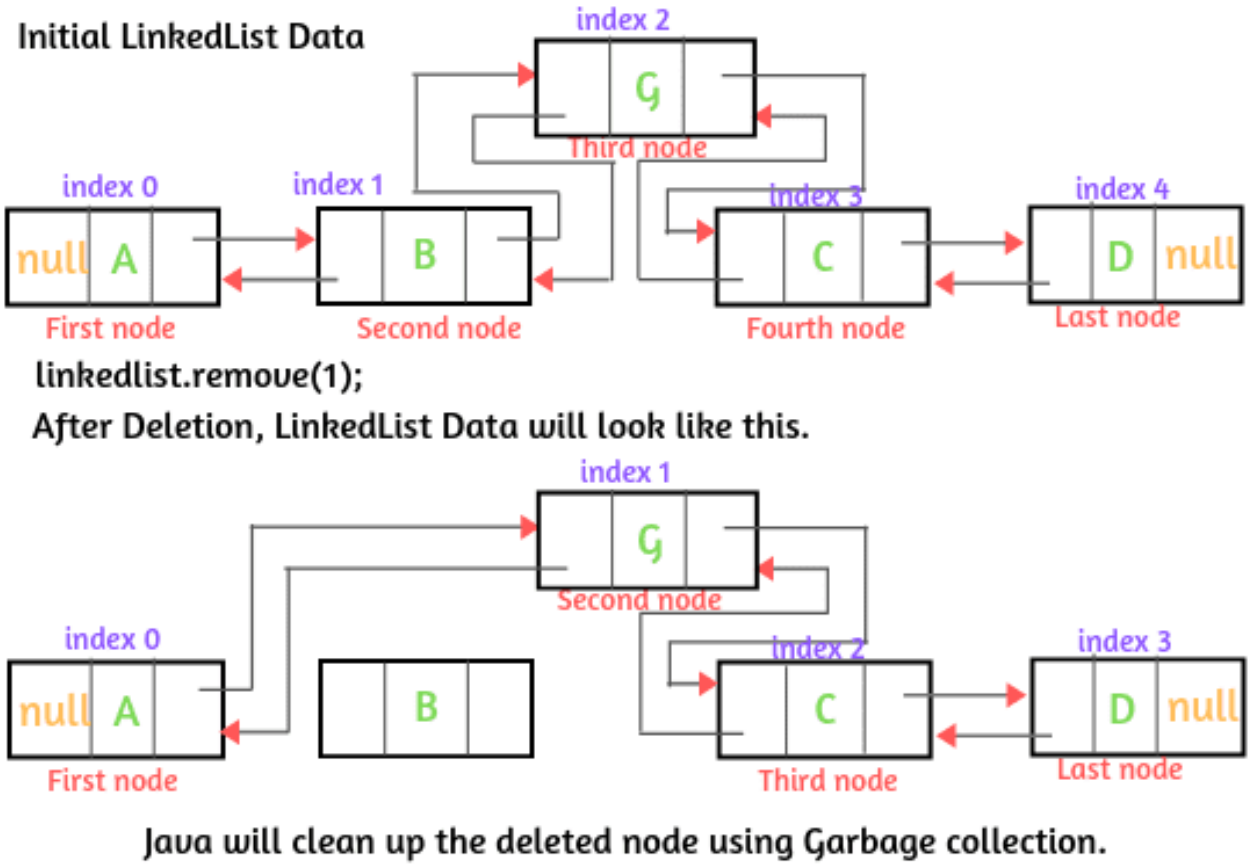
linkedlist.add(2,"G");

After Insertion, LinkedList Data will look like this.



You can see that one node is created with element G and simply changes the next and previous pointer only. No shift of operation has occurred.

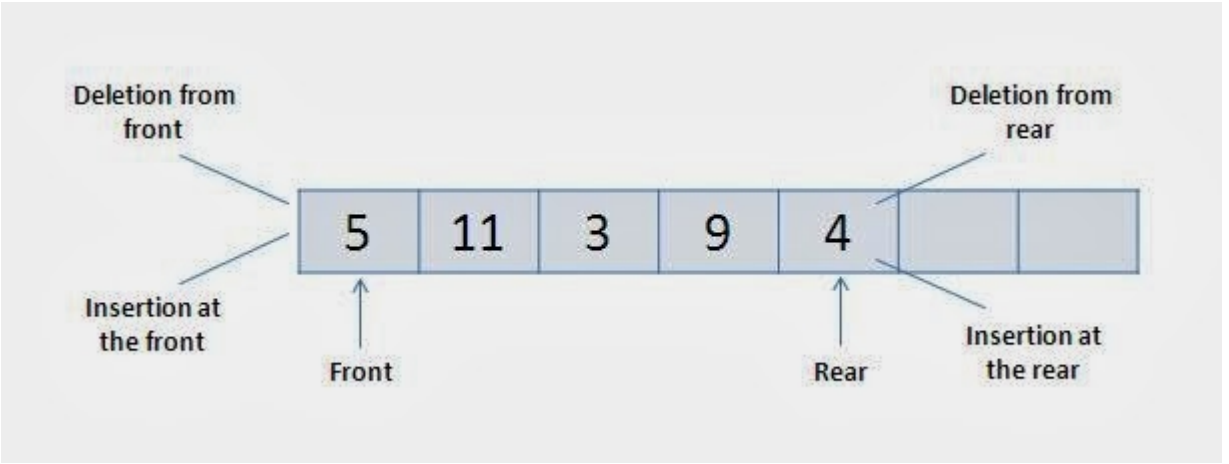
LinkedList


Initial LinkedList Data
index 2
G
Third node
index 0
index 1
index 3
index 4
null
A
B
C
D
null
First node
Second node
Fourth node
Last node
linkedlist.remove(1);
After Deletion, LinkedList Data will look like this.
index 1
G
Second node
index 0
index 2
index 3
null
A
B
C
D
null
First node
Third node
Last node
Java will clean up the deleted node using Garbage collection.

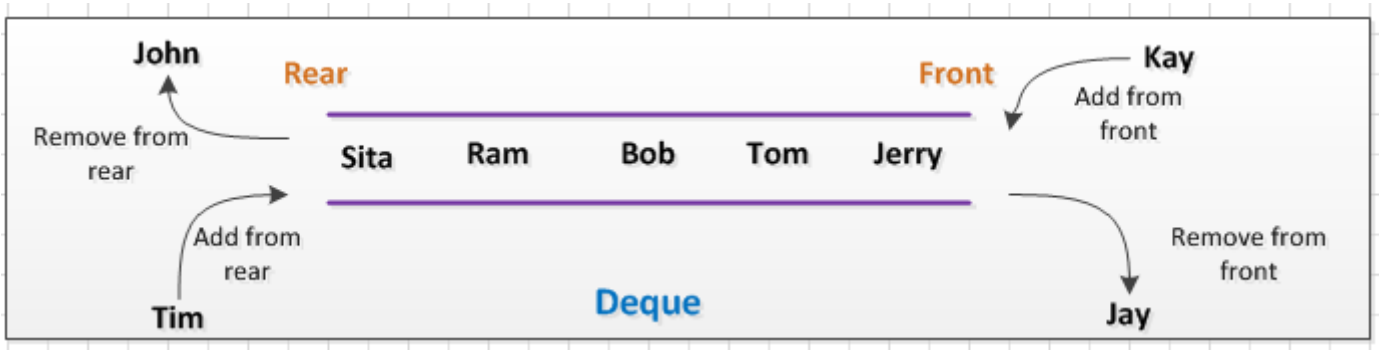
LinkedList

Simple Queue

Double-ended Queue (Deque)



LinkedList



Deque Methods

	First Element or Head		Last Element or Tail	
Operations	**Throws exception**	**Special Value**	**Throws exception**	**Special Value**
Insert	addFirst(element)	offerFirst(element)	addLast(element)	offerLast(element)
Remove	removeFirst()	pollFirst()	removeLast()	pollLast()
Examine	getFirst()	peekFirst()	getLast()	peekLast()

LinkedList

- To create a LinkedList of `String` type:

```
LinkedList<String> llist=new LinkedList<String>();
```

- To add an element to a list:

```
llist.add(2,"Apple");          llist.add("Apple");
llist.addFirst("Apple");       llist.addLast("Apple");
```

- To remove an element from the list:

```
llist.remove(2);          llist.remove("Apple");
llist.removeFirst();      llist.removeLast();
```

- To get the element from the list:

```
llist.get(3);  llist.getFirst();    llist.getLast();
```

LinkedList

- To insert the specified element to a list: return boolean

```
llist.offer("Mango");
llist.offerFirst("Mango");    llist.offerLast("Mango");
```

- To remove an element from the list: returns *spl. value*

```
llist.poll();
llist.pollFirst();      llist.pollLast();
```

- To search an element from the list: returns *spl. value*

```
llist.peek();
llist.peekFirst();      llist.peekLast();
```

LinkedList

```
import java.util.*;
class LinkTest
{
    public static void main(String args[])
    {
      LinkedList<String> llist = new LinkedList<String>();
      llist.add("Mango");
      llist.add("Grapes");
      llist.addLast("Orange");
      System.out.println("Original List: "+llist);
      llist.addFirst("Apple");
      System.out.println("After addFirst: "+llist);
      llist.add("Kiwi");
      System.out.println("After add: "+llist);
   System.out.println("Peek First: "+llist.peekFirst());
-------------------------------------------------
Original List: [Mango, Grapes, Orange]
After addFirst: [Apple, Mango, Grapes, Orange]
After add: [Apple, Mango, Grapes, Orange, Kiwi]
```

LinkedList

```
System.out.println("Poll Last: "+llist.pollLast());
System.out.println("Add Banana: "+
                                llist.offerFirst("Banana"));
System.out.println("offFir, pollLast: "+llist);
llist.removeLast();
System.out.println("After removeLast: "+llist);
ListIterator<String> litr = llist.listIterator(2);
while(litr.hasNext()) {
   String element = litr.next();
   litr.set(element + "+");
   }
System.out.println("Modified List: "+llist);
```

```
----------------------------------------------------
Peek First: Apple
Poll Last: Kiwi
Add Banana: true
offFir, pollLast: [Banana, Apple, Mango, Grapes, Orange]
After removeLast: [Banana, Apple, Mango, Grapes]
Modified List: [Banana, Apple, Mango+, Grapes+]
```

Object Ordering

- A List l may be sorted as follows.

```
Collections.sort(l);
```

- If the List consists of `String` elements, it will be sorted into alphabetical order.
- If it consists of `Date` elements, it will be sorted into chronological order.
- How does this happen?

Object Ordering

- How does this happen?
- String and Date both implement the `Comparable` interface.
- Comparable implementations provide a natural ordering for a class, which allows objects of that class to be *sorted automatically*.
- If you try to sort the elements of a list which do not implement `Comparable`, `Collections.sort(list)` will throw a `ClassCastException`.

Object Ordering

- Similarly, `Collections.sort(list,Comparator)` will throw a `ClassCastException` if you try to sort a list whose elements cannot be compared to one another using the comparator.
- Elements that can be compared to one another are called *mutually comparable*.
- Although elements of different types may be mutually comparable, none of the classes permit interclass comparison.

Writing your own Comparable

- The `Comparable` interface consists of the following method.

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

- The `compareTo` method compares the *receiving object* with the *specified object* and returns a *- integer, 0, or a + integer* depending on whether the receiving object is *less than, equal to, or greater than* the specified object.
- If the specified object cannot be compared to the receiving object, the method throws a `ClassCastException`.

Writing your own Comparable

```
import java.util.*;
import java.lang.*;
class Name implements Comparable<Name> {
    private final String firstName, lastName;
    public Name(String firstName, String lastName) {
        if (firstName == null || lastName == null)
            throw new NullPointerException();
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public String firstName() { return firstName; }
    public String lastName()  { return lastName;  }

    public String toString() {
         return firstName + " " + lastName;
    }
```

Writing your own Comparable

```
 public int compareTo(Name n) {
   int lastCmp = lastName.compareTo(n.lastName);
   return(lastCmp !=0 ? lastCmp :
                        firstName.compareTo(n.firstName));
    }
}
Class ComTest{
public static void main(String args[]){
        ArrayList<Name> names=new ArrayList<Name>();
        names.add(new Name("John", "Smith"));
        names.add(new Name("Karl", "Ng"));
        names.add(new Name("Jeff", "Smith"));
        names.add(new Name("Tom", "Rich"));
        Collections.sort(names);
        System.out.println(names);
    }
}
-----------------------------------------------------
[Karl Ng, Tom Rich, Jeff Smith, John Smith]
```

Collection Algorithms

- The polymorphic algorithms of Collections are pieces of *reusable functionality* provided by the Java platform.
- The Collections Framework defines several algorithms that can be applied to collections and maps.
- These algorithms are defined as static methods within the Collections class.

Collection Algorithms

- The great majority of the algorithms provided by the Java platform operate on `List` instances, but a few of them operate on arbitrary `Collection` instances.
- This section briefly describes the following algorithms:

  Sorting

  Shuffling

  Routine Data Manipulation

  Searching

  Composition

  Finding Extreme Values

Algorithm - Sort

- The sort algorithm reorders a List so that its elements are in ascending order according to an ordering relationship.
- The simple form takes a List and sorts it according to its *elements' natural ordering*.

```
import java.util.*;
public class Sort {
    public static void main(String[] args) {
        List<String> list = Arrays.asList(args);
        Collections.sort(list);
        System.out.println(list);    }
}
% java Sort i walk the line
[i, line, the, walk]
```

Algorithm – Data Manipulation

- The Collections class provides five algorithms for doing routine data manipulation on List objects:
  - ***reverse*** — reverses the order of the elements in a List.
  - ***fill*** — *overwrites* every element in a List with the specified value. This operation is useful for reinitializing a List.
  - ***copy*** — takes two arguments, a destination List and a source List, and copies the elements of the source into the destination, *overwriting* its contents.

Algorithm – Data Manipulation

- The destination List must be at least as long as the source. If it is longer, the remaining elements in the destination List are unaffected.
- ***swap*** — swaps the elements at the specified positions in a List.
- ***addAll*** — adds all the specified elements to a Collection. The elements to be added may be specified individually or as an array.

Algorithm – Searching

- The binarySearch algorithm searches for a specified element in a sorted List.
- This algorithm has two forms.
  - The first takes a List and an element to search for (the "search key").
  - This form assumes that the List is sorted in ascending order according to the natural ordering of its elements.
  - `static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)`

Algorithm – Searching

- The second form takes a *Comparator* in addition to the List and the search key, and assumes that the List is sorted into ascending order according to the specified Comparator.
- `static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)`
- Searches the specified list for the specified object using the binary search algorithm.

Algorithm – Composition

- The frequency and disjoint algorithms test some aspect of the composition of one or more Collections:
  - ***frequency*** — counts the number of times the specified element occurs in the specified collection.

    ```
    frequency(Collection<?> c, Object o)
    ```

  - ***disjoint*** — determines whether two Collections are disjoint; that is, whether they contain no elements in common.

    ```
    disjoint(Collection<?> c1, Collection<?> c2)
    ```

Algorithm – Finding Ext Values

- The min and the max algorithms return, respectively, the minimum and maximum element contained in a specified Collection.
- The simple form takes only a Collection and returns the minimum (or maximum) element according to the elements' natural ordering.
- `static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> coll)`

Algorithm – Finding Ext Values

- The second form *takes a Comparator* in addition to the Collection and returns the minimum (or maximum) element according to the specified Comparator.
- `static <T> T  min(Collection<? extends T> coll, Comparator<? super T> comp)`

Algorithms – Example

```
Class AlgoTest{
public static void main(String args[]){
   LinkedList<Integer> ll = new LinkedList<Integer>();
   ll.add(-8);
   ll.add(20);
   ll.add(-20);
   ll.add(8);
   Collections.sort(ll);
   System.out.println("List sorted: "+ll);
   Collections.reverse(ll);
   System.out.println("List in reverse: "+ll);
   Collections.shuffle(ll);
   System.out.println("Shuffled List: "+ll);
-------------------------------------------------
List sorted: [-20, -8, 8, 20]
List in reverse: [20, 8, -8, -20]
Shuffled List: [-8, 20, -20, 8]
```

Algorithms – Example

```
   Collections.swap(ll,1,3);
   System.out.println("Swapped List: "+ll);
   System.out.println("bin Search: "
               +Collections.binarySearch(ll,8));
   Collections.rotate(ll,2);
   System.out.println("Rotate List: "+ll);
   System.out.println("Minimum: "+Collections.min(ll));
   System.out.println("Maximum: "+Collections.max(ll));
```

```
-----------------------------------------------------

Swapped List: [-8, 8, -20, 20]
bin Search: 1
Rotate List: [-20, 20, -8, 8]
Minimum: -20
Maximum: 20
```

Algorithms – Example

```
    LinkedList<Integer> ll1 = new LinkedList<Integer>();
    ll1.add(5);
    ll1.add(7);
    Collections.copy(ll,ll1);
    System.out.println("New List: "+ll1);
    System.out.println("After copy: "+ll);
     }
}
```

```
New List: [5, 7]
After copy: [5, 7, -8, 8]
```

Reference

https://docs.oracle.com/javase/tutorial/collections/algorithms/index.html

Java – The Complete Reference by Herbert Schildt