Data Structure

Basic Data Structure

- Scalar Data Structure
  - Integer, Character, Boolean, Float, Double, etc.
- Vector or Linear Data Structure
  - Array, List, Queue, Stack, Priority Queue, Set, etc.
- Non-linear Data Structure
  - Tree, Table, Graph, Hash Table, etc.

Scalar Data Structure

- A *scalar* is the simplest kind of data that Java programming language manipulates. A scalar is either a number (like 4 or 3.25e20) or a character. (Integer, Character, Boolean, Float, Double, etc.)
- A scalar value can be acted upon with operators (like plus or concatenate), generally yielding a scalar result. A scalar value can be stored into a scalar variable. Scalars can be read from files and devices and written out as well.

List – A Basic Data Structure

- By definition, a **list** is a finite set of entries with a certain order.
- The entries in the list, depending on the language, may be constrained to the same type.
- An array is a related data structure that also stores its entries sequentially. However, the items in an array usually must be of the same type.
- Nearly all kinds of tree structures can also be stored as lists

Queue / Stack as Basic Data Structures

- Queue is a list with First-In Fist-Out manipulation
- EnQueue (Insert queue)
- DeQueue (Delete queue)

- Stack is a list with Last-In First-Out manipulation
- Push (Insert stack)
- Pop (Delete stack)

Object-Oriented Concept

- Everything is object
- Class (Class object)
- Instance (Instance object)
- Class
  - Encapsulation (Information Hiding, Data, Methods)
  - Polymorphism (Overriding, Overloading)
  - Visibility Mode (public, protected, private)
- Association
  - Composition / Aggregation

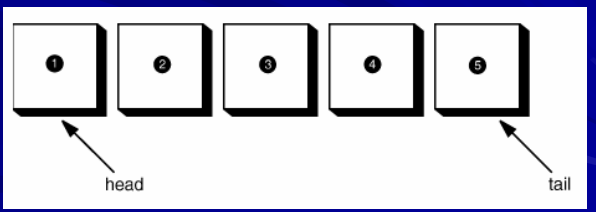
Separating Collection Interfaces and Implementation

- As is common for modern data structure libraries, the Java collection library separates interfaces and implementations.
- Let us look at that separation with a familiar data structure, the queue.
- The Java library does not supply a queue, but it is nevertheless a good example to introduce the basic concepts.

Queue Interface

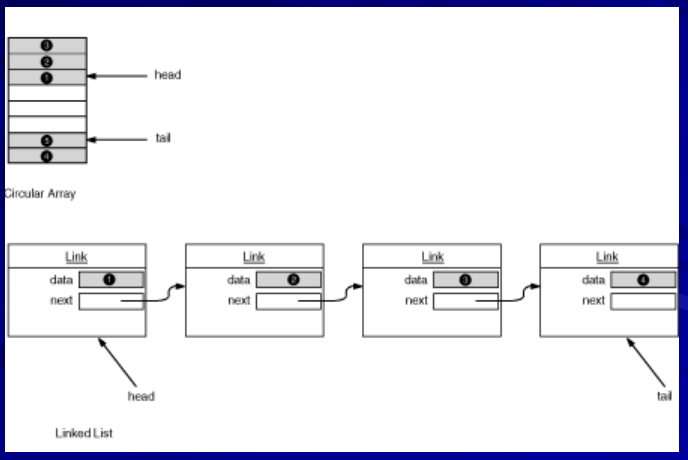
- A *queue interface* specifies that you can add elements at the tail end of the queue, remove them at the head, and find out how many elements are in the queue. You use a queue when you need to collect objects and retrieve them in a "first in, first out" fashion.

```
interface Queue
   { void add(Object obj);
     Object remove();
     int size();
   }
```



Implementation of Queue Interface

- The interface tells you nothing about how the queue is implemented. There are two common implementations of a queue, one that uses a "circular array" and one that uses a linked list



Implementation of Queue Interface - Code

```
class CircularArrayQueue implements Queue
{
CircularArrayQueue( int capacity) { . . . } public
void add(Object obj) { . . . }
public Object remove() { . . . }
public int size ( ) { . . . }
private Object [ ] elements;
private int head;
private int tail;
}

Queue expressLane = new
CircularArrayQueue(100);
expressLane.add(new Customer("Harry"));
```

1

```
class CircularArrayQueue
{
public void add(Object obj)
throws CollectionFullException
. . .
}
```

```
class LinkedListQueue
implements Queue
{
LinkedListQueue() { . . . }
public void add(Object obj) { . . . }
public Object remove() { . . . }
public int size() { . . . }
private Link head;
private Link tail;
}

Queue expressLane = new
LinkedListQueue(); expressLane.add(
new Customer("Harry"));
```

2

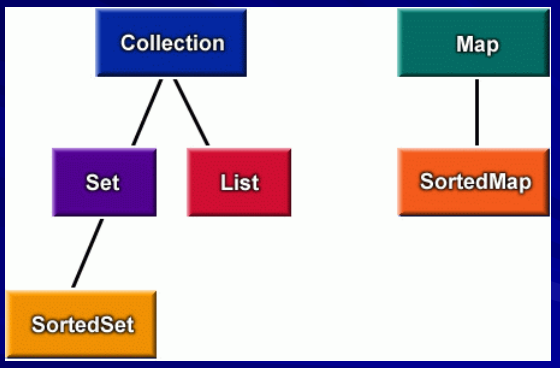
Collections

- A collection (sometimes called a *container*) is an object that groups multiple elements into a single unit.
- Collections are used to store, retrieve and manipulate data, and to transmit data from one method to another.
- Collections typically represent data items that form a natural group, a card hand, a mail folder, a telephone directory…

The Java Collections Framework

- The Java collections framework is made up of a set of interfaces and classes for working with groups of objects
- The Java Collections Framework provides
  - **Interfaces**: abstract data types representing collections. **Implementations**: concrete implementations of the collection interfaces.
  - **Algorithms**: methods that perform useful computations, like searching and sorting, on objects that implement collection interfaces.

The Interfaces



Note: Some of the material on these slides was taken from the Java Tutorial at http://www.java.sun.com/docs/books/tutorial

Sets

- A group of unique items, meaning that the group contains no duplicates
- Some examples
  - The set of uppercase letters ‘A’ through ‘Z’
  - The set of nonnegative integers { 0, 1, 2, … }
  - The empty set {}
- The basic properties of sets
  - Contain only one instance of each item
  - May be finite or infinite
  - Can define abstract concepts

Maps

- A map is a special kind of set.
- A map is a set of pairs, each pair representing a one-directional “mapping” from one set to another
  - An object that maps keys to values
- Some examples
  - A map of keys to database records
  - A dictionary (words mapped to meanings)
  - The conversion from base 2 to base 10

What Is The Real Difference?

- Collections
  - You can add, remove, lookup *isolated* items in the collection
- Maps
  - The collection operations are available but they work with a *key-value* pair instead of an isolated element
  - The typical use of a `Map` is to provide access to values stored by key

Another Way to Look At It

- The `Collection` interface is a group of objects, with duplicates allowed
- `Set` extends `Collection` but forbids duplicates
- `List` extends `Collection` and allows duplicates and positional indexing
- `Map` extends neither `Set` nor `Collection`

The `Collection` Interface

Found in the `java.util` package

Optional methods throw `UnsupportedOperationException` if the implementing class does not support the operation.

Bulk operations perform some operation on an entire `Collection` in a single shot

The `toArray` methods allow the contents of a `Collection` to be translated into An array.

Collection

```
// Basic Operations
size():int;
isEmpty():boolean;
contains(Object):boolean;
add(Object):boolean;          // Optional
remove(Object):boolean;       // Optional
iterator():Iterator;

// Bulk Operations
containsAll(Collection):boolean;
addAll(Collection):boolean;   // Optional
removeAll(Collection):boolean;// Optional
retainAll(Collecton):boolean; // Optional
clear():void;                 // Optional

// Array Operations
toArray():Object[];
toArray(Object[]):Object[];
```

`Set` Interface

- A `Set` is a `Collection` that cannot contain duplicate elements.
  - Set models the mathematical *set* abstraction.
- The `Set` interface extends `Collection` and contains *no* methods other than those inherited from `Collection`
- It adds the restriction that duplicate elements are prohibited.
- Two `Set` objects are equal if they contain the same elements.

`Set` Bulk Operations

- The bulk operations perform standard set-algebraic operations. Suppose s1 and s2 are Sets.
  - s1.containsAll(s2): Returns true if s2 is a **subset** of s1.
  - s1.addAll(s2): Transforms s1 into the **union** of s1 and s2. (The union of two sets is the set containing all the elements contained in either set.)
  - s1.retainAll(s2): Transforms s1 into the **intersection** of s1 and s2. (The intersection of two sets is the set containing only the elements that are common in both sets.)

Java `Lists`

- A `List` is an ordered `Collection` (sometimes called a *sequence*).
- Lists may contain duplicate elements.
- In addition to the operations inherited from `Collection`, the `List` interface includes operations for:
  - Positional Access
  - Search
  - List Iteration
  - Range-view

`List` Interface

Think of the Vector class

List	
// Positional Access	
get(int):Object;	
set(int,Object):Object;	// Optional
add(int, Object):void;	// Optional
remove(int index):Object;	// Optional
addAll(int, Collection):boolean;	// Optional
// Search	
int indexOf(Object);	
int lastIndexOf(Object);	
// Iteration	
listIterator():ListIterator;	
listIterator(int):ListIterator;	
// Range-view List	
subList(int, int):List;	

`Map` Interface

Map

```
// Basic Operations
put(Object, Object):Object;
get(Object):Object;
remove(Object):Object;
containsKey(Object):boolean;
containsValue(Object):boolean;
size():int;
isEmpty():boolean;

// Bulk Operations
void putAll(Map t):void;
void clear():void;

// Collection Views
keySet():Set;
values():Collection;
entrySet():Set;
```

EntrySet

```
getKey():Object;
getValue():Object;
setValue(Object):Object;
```

Implementation Classes

Interface	Implementation				Historical
`Set`	`HashSet`		`TreeSet`		
`List`		`ArrayList`		`LinkedList`	`Vector` `Stack`
`Map`	`HashMap`		`TreeMap`		`HashTable` `Properties`

Note: When writing programs think about interfaces and not implementations. This way the program does not become dependent on any added methods in a given implementation, leaving the programmer with the freedom to change implementations.

Iterator

- Represents a loop
- Created by Collection.iterator()
- Similar to Enumeration
  - Improved method names
  - Allows a remove() operation on the current item

Iterator Methods

- boolean hasNext()
  - Returns `true` if the iteration has more elements
- Object next()
  - Returns next element in the iteration
- void remove()
  - Removes the current element from the underlying Collection

Iterator

- An object that implements the `Iterator` interface generates a series of elements, one at a time
  - Successive calls to the `next()` method return successive elements of the series.
- The `remove()` method removes from the underlying Collection the last element that was returned by next.

Iterator

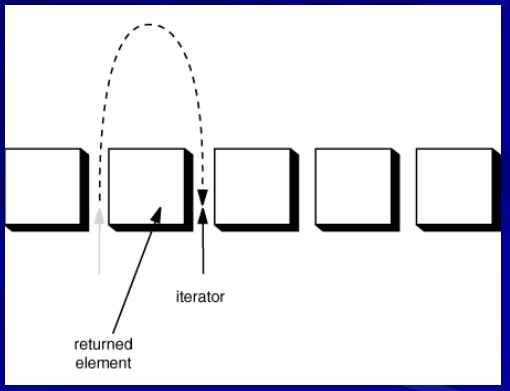
hasNext():boolean;
next():Object;
remove():voidt;

Iterator as pointer

Traverse

```
Iterator iter = c.iterator();
while (iter.hasNext())
{
Object obj = iter.next();
do something with obj
}
```

```
public static void print(Collection c)
{
System.out.print("[ ");
Iterator iter = c.iterator();
while (iter.hasNext())
   System.out.print( iter.next() + " ");
System.out.println("]");
}
```



Remove

```
Iterator it = c.iterator();
it.next();   // skip over the
             //first element
it.remove(); // now remove it
```

```
it.remove();
it.remove(); // Error!
```

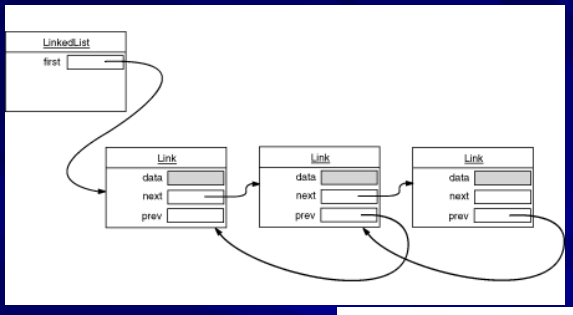
```
it.remove();
it.next();
it.remove(); // Ok
```

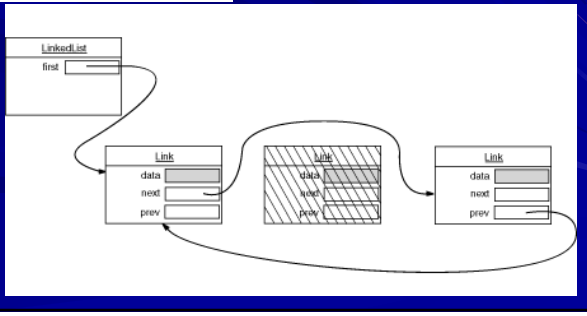
Implementation of method ‘addAll’

```
int size();
boolean isEmpty();
boolean contains(Object obj);
boolean containsAll(Collection c);
boolean equals(Object other);
boolean addAll(Collection from);
boolean remove(Object obj);
boolean removeAll(Collection c);
void clear();
boolean retainAll(Collection c);
Object[ ] toArray();
```

```
public static boolean addAll( Collection to, Collection from)
{
Iterator iter = from.iterator();
boolean modified = false;
while (iter.hasNext())
   if (to.add(iter.next()))
       modified = true;
return modified;
}
```

Link-Listed





Remove a Element

ListIterator is extended from Iterator

```
LinkedList staff = new LinkedList();
staff.add("Angela");
staff.add("Bob");
staff.add("Carl");
Iterator iter = staff.iterator();
for (int i = 0; i < 3; i++)
   System.out.println(iter.next());
iter.remove(); // remove last visited element
```

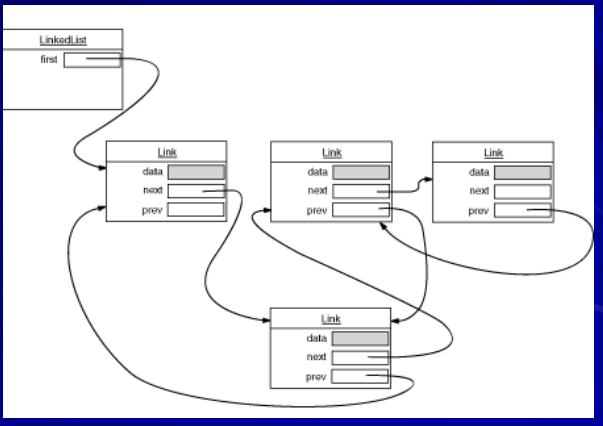
```
interface ListIterator extends Iterator
{ void add(Object);
. . .
}
```

ListIterator

- interface ListIterator extends Iterator
- Created by List.listIterator()
- Adds methods to
  - traverse the List in either direction
  - modify the List during iteration
- Methods added:
  - hasPrevious(), previous()
  - nextIndex(), previousIndex()
  - set(Object), add(Object)

Using ListIterator to Add an Element

```
ListIterator iter = staff.listIterator();
iter.next();
iter.add("Juliet");
```




```
ListIterator iter = list.listIterator();
Object oldValue = iter.next(); // returns first element
iter.set(newValue); // sets first element to newValue
```

```
LinkedList list = . . .;
ListIterator iter1 = list.listIterator();
ListIterator iter2 = list.listIterator();
iter1.next();
iter1.remove();
iter2.next(); // throws ConcurrentModificationException
```

LinkedListTest.java

```
import java.util.*;

public class LinkedListTest
{
  public static void main( String[] args)
  {
     List a = new LinkedList();
     a.add("Angela");
     a.add("Carl");
     a.add("Erica");

     List b = new LinkedList();
     b.add("Bob");
     b.add("Doug");
     b.add("Frances");
     b.add("Gloria"); // merge the words from b into a

     ListIterator aIter = a.listIterator();
     Iterator bIter = b.iterator();

     while (bIter.hasNext())
        {
           if (aIter.hasNext()) aIter.next();
           aIter.add(bIter.next());
        }

     System.out.println(a); // remove every second word
from b
     bIter = b.iterator();
     while (bIter.hasNext())
     {
        bIter.next(); // skip one element
        if (bIter.hasNext())
        {
           bIter.next(); // skip next element
           bIter.remove(); // remove that element
        }
     }

     System.out.println(b); // bulk operation: remove all
words in b from a
     a.removeAll(b);
     System.out.println(a);
  }
}
```