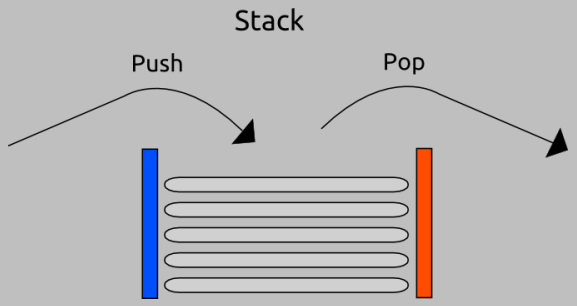
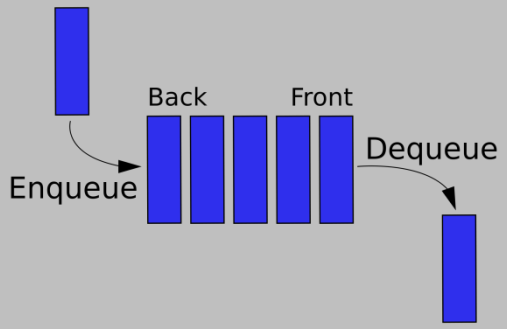
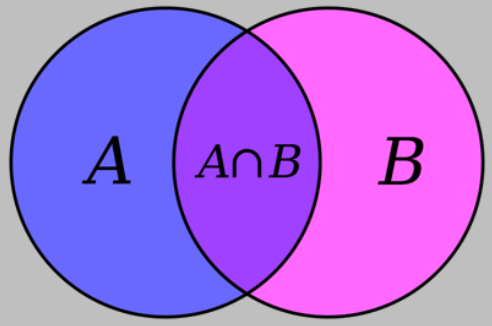


first in last out



first in first out



set

Generic Collections

Lecture 10

Department of Computer Engineering

INHA University

Dr. Tamer ABUHMED

Outline

- Introduction
- The collections framework
- Wrapper Classes
  - Autoboxing and Auto-Unboxing
- Lists
- ArrayList and Iterator
- LinkedList
- Collections framework Algorithms
- Stack
- PriorityQueue

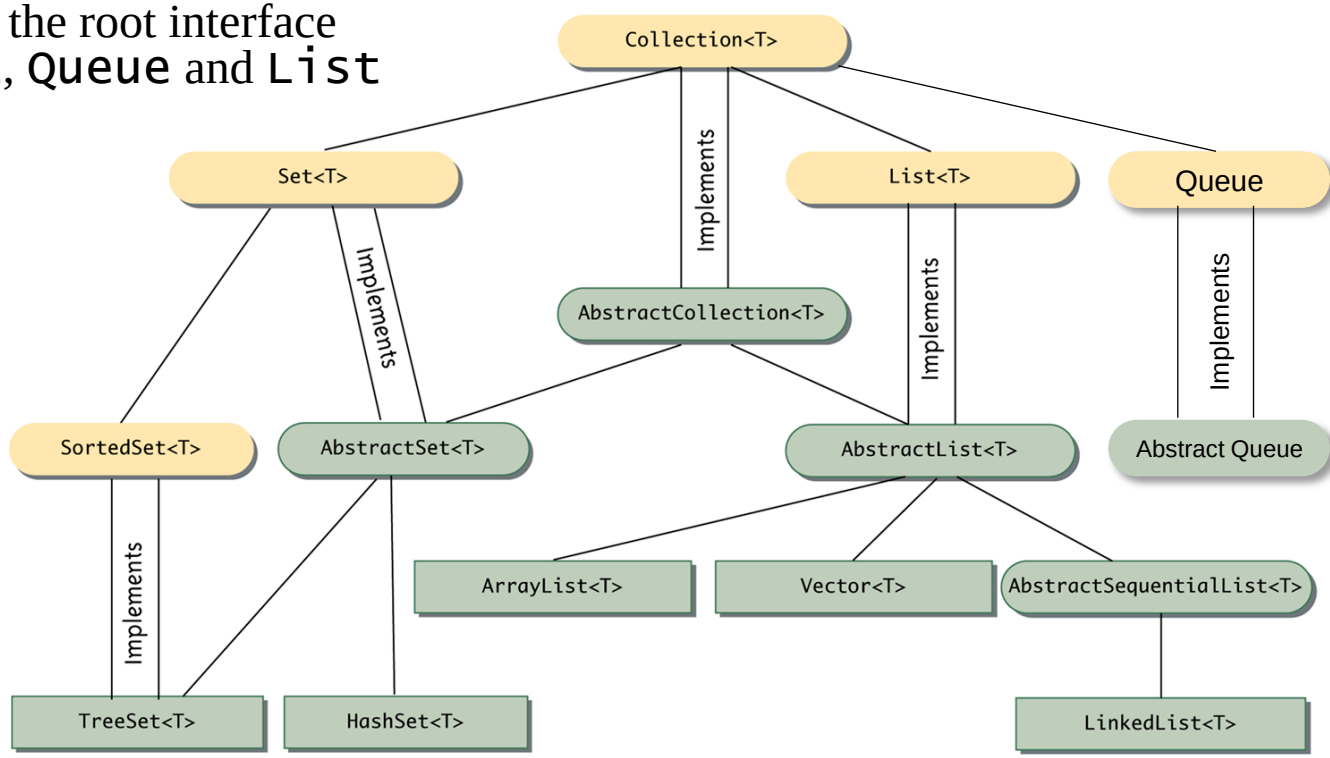
Introduction

- A collection is a data structure—actually, an object—that can hold references to other objects.
  - Usually, collections contain references to objects that are all of the same type.
- Java collection belongs to the collections framework
  - prebuilt data structures
  - interfaces and methods for manipulating those data structures
- Package `java.util`.

The collections framework

Interface `Collection` is the root interface from which interfaces `Set`, `Queue` and `List` are derived.

Interface `Set` defines a collection that does not contain duplicates.
Interface `Queue` defines a collection that represents a waiting line.



Wrapper Classes

- Can we convert the primitive data types into Objects?
  기본 자료형을 지원하지 않아 객체형태로 표현해야 함

YES

- Primitive data types have corresponding classes called "wrapper classes" which provide object versions of primitive data.
- Wrapper classes are used in situations where an object is required rather than primitive data values.
- Example:

Primitive Type	Wrapper Class
byte	Byte
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

The table lists the primitive data types and their corresponding wrappers.

모든 java 변수의 부모 클래스 = object
-> 오토박싱 가능

Autoboxing and Auto-Unboxing

- A boxing conversion converts a value of a primitive type to an object of the corresponding type-wrapper class.
- An unboxing conversion converts an object of a type-wrapper class to a value of the corresponding primitive type.
- These conversions can be performed automatically (called autoboxing and auto-unboxing).
- Example:


```
//Boxing
int a = 8;
Integer object = new Interger(a);
```

```
// create integerArray
Integer[] integerArray = new Integer[ 5 ];

// assign Integer 10 to integerArray[ 0 ]
integerArray[ 0 ] = 10;
```

```
//Auto boxing
Integer object = 8;
```

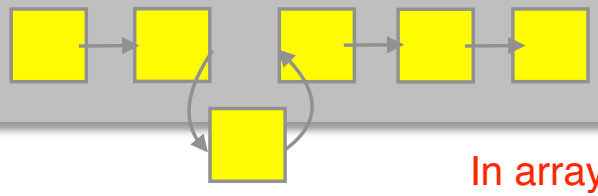
```
// get int value of Integer
int value = integerArray[ 0 ];
```

```
//Unboxing
int b = object;
```

Lists

- A `List` (sometimes called a sequence) is a `Collection` that *can contain duplicate elements*.
- `List` indices are *zero* based.
- In addition to the methods inherited from `Collection`, `List` provides methods for manipulating elements via their indices, manipulating a specified range of elements, searching for elements and obtaining a `ListIterator` to access the elements.
- Interface `List` is implemented by several classes, including `ArrayList`, `LinkedList` and `Vector`.
- Autoboxing occurs when you add primitive-type values to objects of these classes, because they store only references to objects.

Lists Cont.



linkedList = 추가 삽입 편리

In array, 한칸 씩 오른쪽으로 밀고 추가 해야함

- Class `ArrayList` and `Vector` are resizable-array implementations of `List`.
- Inserting an element between existing elements of an `ArrayList` or `Vector` is an inefficient operation. -> linkedList
- A `LinkedList` enables efficient insertion (or removal) of elements in the middle of a collection.
- The primary difference between `ArrayList` and `Vector` is that `Vector`s are synchronized by default, whereas `ArrayList`s are not. Mulithread -> vector

  Don,t use vector in normal software
  <- synchronized is slow <- have to check multithread
- Unsynchronized collections provide better performance than synchronized ones.
- For this reason, `ArrayList` is typically preferred over `Vector` in programs that do not share a collection among threads.

```
Vector<Integer> array = new Vector();
array.add(new Integer(12));
array.add(2);
array.add("hello");//error
array.add(2.3);//error
System.out.println(array.get(1));//print array[1]
array.clear(); //clear array
```

List<Object> array = new ArrayList();
->new에서 구체화 시키기

ArrayList and Iterator

- `List` method `add` adds an item to the end of a list.
- `List` method `size` returns the number of elements.
- `List` method `get` retrieves an individual element's value from the specified index.
- `Collection` method `iterator` gets an `Iterator` for a `Collection`.
- `Iterator`- method `hasNext` determines whether a `Collection` contains more elements.
  - Returns `true` if another element exists and `false` otherwise.
- `Iterator` method `next` obtains a reference to the next element.
- `Collection` method `contains` determine whether a `Collection` contains a specified element.
- `Iterator` method `remove` removes the current element from a `Collection`.

ArrayList Example

```
import java.util.List;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class CollectionTest
{
   public static void main( String[] args )
   {
      // add elements in colors array to list
      String[] colors = { "MAGENTA", "RED", "WHITE", "BLUE", "CYAN" };
      List< String > list = new ArrayList< String >();

      for ( String color : colors )
         list.add( color ); // adds color to end of list
      // output list contents
      System.out.println( "ArrayList: " );

      for ( int count = 0; count < list.size(); count++ )
         System.out.printf( "%s ", list.get( count ) );


   } // end main
}
```

Output

```
ArrayList:
MAGENTA RED WHITE
BLUE CYAN
```

Iterator = Pointer

- Sometimes you will want to cycle through the elements in a collection. For example, you might want to display each element in an ArrayList.
- The easiest way to do this is to employ an iterator, which is an object that implements either the Iterator or the ListIterator interface.
- An `Iterator` is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

Iterator with ArrayList Example

```
public class IteratorTest {
public static void main(String[] args) {
// add elements in colors array to list
String[] colors = { "MAGENTA", "RED", "WHITE", "BLUE",
"CYAN" };
List<String> list = new ArrayList<String>();

for (String color : colors)
     list.add(color); // adds color to end of list

// Modify objects being iterated
System.out.print("Modifying the contents of list ...\n ");
ListIterator litr = list.listIterator();
while (litr.hasNext()) {
   Object element = litr.next();
   litr.set(element + "+");
}

// Now, display the list backwards
Iterator itr = list.listIterator();
System.out.print("Modified list: ");
while (itr.hasNext()) {
Object element = itr.next();
System.out.print(element + " ");
}
```

```
// Now, display the list backward
System.out.print("\n Modified list
backwards: ");
while (litr.hasPrevious()) {
Object element = litr.previous();
System.out.print(element + " ");
}
System.out.println("\n");

  }
}
```

Output

```
Modifying the contents of list ...

Modified list: MAGENTA+ RED+ WHITE+ BLUE+
CYAN+

Modified list backwards: CYAN+ BLUE+
WHITE+ RED+ MAGENTA+
```

Iterator with LinkedList Example

```
public class ListTest
{
   public static void main( String[] args )
   {
      // add colors elements to list1
  String[] colors = { "black", "yellow", "green", "blue",
"violet", "silver" };
  List< String > list1 = new LinkedList< String >();

  for ( String color : colors )
       list1.add( color );

      // add colors2 elements to list2
 String[] colors2 =  { "gold", "white", "brown", "blue",
"gray", "silver" };
  List< String > list2 = new LinkedList< String >();

 for ( String color : colors2 )
        list2.add( color );

 list1.addAll( list2 ); // concatenate lists
 list2 = null; // release resources
 printList( list1 ); // print list1 elements

convertToUppercaseStrings( list1 ); // convert to uppercase
printList( list1 ); // print list1 elements

System.out.print( "\nDeleting elements 4 to 6..." );
removeItems( list1, 4, 7 ); // remove items 4-6 from list
printList( list1 ); // print list1 elements
// output List contents
private static void printList( List< String > list )
 {
    System.out.println( "\nlist: " );

      for ( String color : list )
         System.out.printf( "%s ", color );
```

```
System.out.println();
   } // end method printList
} // end main
// locate String objects and convert to uppercase
private static void convertToUppercaseStrings( List< String >
list )
 {
     ListIterator< String > iterator = list.listIterator();

 while ( iterator.hasNext() )
    {
 String color = iterator.next(); // get item
 iterator.set( color.toUpperCase() );
// convert to upper case
      } // end while
 } // end method convertToUppercaseStrings

// obtain sublist and use clear method to delete sublist
items
private static void removeItems( List< String > list, int
start, int end )
   {
      list.subList( start, end ).clear(); // remove items
   } // end method removeItems
```

Output

```
list:
black yellow green blue violet silver gold white brown blue gray
silver
list:
BLACK YELLOW GREEN BLUE VIOLET SILVER GOLD WHITE BROWN BLUE GRAY
SILVER
Deleting elements 4 to 6...
list:
BLACK YELLOW GREEN BLUE WHITE BROWN BLUE GRAY SILVER
```

Collections framework Algorithms

- The collections framework also provides polymorphic versions of algorithms you can run on collections.
  - Sorting: sorts the elements of a `List`
  - Shuffling: randomly orders a `List`'s elements.
  - Routine Data Manipulation
    - Reverse
    - Copy: takes two arguments—a destination `List` and a source `List`
    - Fill: overwrites elements in a `List` with a specified value
  - Searching
    - Binary Search: locates an object in a `List and returns the index`
  - Composition
    - Frequency: returns the number of times that the second argument appears in the collection
    - Disjoint: takes two `Collection`s and returns `true` if they have no elements in common
  - Finding extreme values
    - Min: returns the smallest element in a `Collection`
    - Max: returns the largest element in a `Collection`

Collections framework Algorithms

```
public static void main( String[] args )
   {
      String[] suits = { "Hearts", "Diamonds", "Clubs", "Spades" };

      // Create and display a list containing the suits array elements
      List< String > list = Arrays.asList( suits ); // create List
      System.out.printf( "Unsorted array elements: %s\n", list );
      Collections.sort( list ); // sort ArrayList
      // output list
      System.out.printf( "Sorted array elements: %s\n", list );
      Collections.shuffle(list);
      System.out.printf( "Shuffle array elements: %s\n", list );
      String[] suitsCopy = new String [4];
      List< String > listCopy = Arrays.asList(suitsCopy);
      Collections.copy(listCopy, list);
      System.out.printf( "List Copy elements: %s\n", listCopy );

      List< String > listFill = Arrays.asList(suitsCopy);
      Collections.fill(listFill, "R");
      System.out.printf( "List Fill elements: %s\n", listFill );

      int index = Collections.binarySearch(list, "Spades");
      System.out.printf( "Element Spades exists in the List at index: %d\n", index );

      int frequancy = Collections.frequency(listFill, "R");
      System.out.printf( "the repetition of R in the listFill is: %d\n", frequancy );


      boolean disjoint = Collections.disjoint(list, listFill);
      System.out.printf( "The disjoint of list and listFill is %b\n", disjoint );

   } // end main
} // end class
```

Output

```
Unsorted array elements: [Hearts, Diamonds, Clubs, Spades]
Sorted array elements: [Clubs, Diamonds, Hearts, Spades]
Shuffle array elements: [Clubs, Hearts, Diamonds, Spades]
List Copy elements: [Clubs, Hearts, Diamonds, Spades]
List Fill elements: [R, R, R, R]
Element Spades exists in the List at index: 3
the repetition of R in the listFill is: 4
The disjoint of list and listFill is true
```

Stack Class of Package java.util

- Class `Stack` in the Java utilities package (`java-.util`) extends class `Vector` to implement a stack data structure.
- `Stack` method `push` adds a `Number` object to the top of the stack.
- Any integer literal that has the suffix `L` is a `long` value.
- An integer literal without a suffix is an `int` value.
- Any floating-point literal that has the suffix `F` is a `float` value.
- A floating-point literal without a suffix is a `double` value.
- `Stack` method `pop` removes the top element of the stack.
  - If there are no elements in the `Stack`, method `pop` throws an `EmptyStackException`, which terminates the loop.
- Method `peek` returns the top element of the stack without popping the element off the stack.
- Method `isEmpty` determines whether the stack is empty.

Stack Example

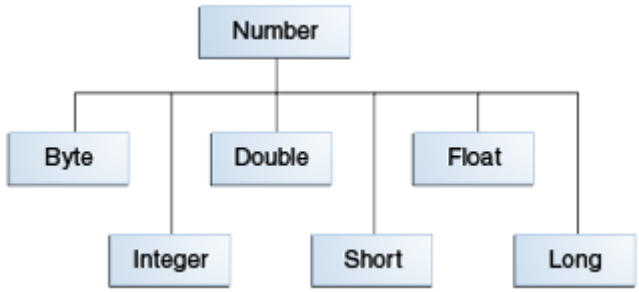
```
public class StackTest
{
   public static void main( String[] args )
   {
      Stack< Number > stack = new Stack< Number >(); // create a Stack

      // use push method
      stack.push( 12L ); // push long value 12L
      System.out.println( "Pushed 12L" );
      printStack( stack );
      stack.push( 34567 ); // push int value 34567
      System.out.println( "Pushed 34567" );
      printStack( stack );
      stack.push( 1.0F ); // push float value 1.0F
      System.out.println( "Pushed 1.0F" );
      printStack( stack );
      stack.push( 1234.5678 ); // push double value 1234.5678
      System.out.println( "Pushed 1234.5678 " );
      printStack( stack );

      // remove items from stack
      try
      {
         Number removedObject = null;

         // pop elements from stack
         while ( true )
         {
            removedObject = stack.pop(); // use pop method
            System.out.printf( "Popped %s\n", removedObject );
            printStack( stack );
         } // end while
      } // end try
      catch ( EmptyStackException emptyStackException )
      {
         emptyStackException.printStackTrace();
      } // end catch
   } // end main

   // display Stack contents
   private static void printStack( Stack< Number > stack )
   {
      if ( stack.isEmpty() )
         System.out.println( "stack is empty\n" ); // the stack is empty
      else // stack is not empty
         System.out.printf( "stack contains: %s (top)\n", stack );
   } // end method printStack
} // end class StackTest
```



Stack Example: Output

Output

```
Pushed 12L
stack contains: [12] (top)
Pushed 34567
stack contains: [12, 34567] (top)
Pushed 1.0F
stack contains: [12, 34567, 1.0] (top)
Pushed 1234.5678
stack contains: [12, 34567, 1.0, 1234.5678] (top)
Popped 1234.5678
stack contains: [12, 34567, 1.0] (top)
Popped 1.0
stack contains: [12, 34567] (top)
Popped 34567
stack contains: [12] (top)
Popped 12
stack is empty

java.util.EmptyStackException
at java.util.Stack.peek(Unknown Source)
at java.util.Stack.pop(Unknown Source)
at StackTest.main(StackTest.java:34)
```

Class PriorityQueue and Interface Queue

- Interface `Queue` extends interface `Collection` and provides additional operations for inserting, removing and inspecting elements in a queue.
- `PriorityQueue` orders elements by their natural ordering.
  - Elements are inserted in priority order such that the highest-priority element (i.e., the largest value) will be the first element removed from the `PriorityQueue`.
- Common `PriorityQueue` operations are
  - `offer` to insert an element at the appropriate location based on priority order
  - `poll` to remove the highest-priority element of the priority queue
  - `peek` to get a reference to the highest-priority element of the priority queue
  - `clear` to remove all elements in the priority queue
  - `size` to get the number of elements in the queue.

PriorityQueue Example

```
public class PriorityQueueTest
{
   public static void main( String[] args
)
   {
      // queue of capacity 11
      PriorityQueue< Double > queue = new
PriorityQueue< Double >();

      // insert elements to queue
      queue.offer( 3.2 );
      queue.offer( 9.8 );
      queue.offer( 5.4 );
      queue.offer( 5.4 );
      queue.offer( 5.2 );
      System.out.print( "Polling from
queue: " );

      // display elements in queue
      while ( queue.size() > 0 )
      {
         System.out.printf( "%.1f ",
queue.peek() ); // view top element
         queue.poll(); // remove top
element
      } // end while
   } // end main
} // end class PriorityQueueTest
```

Output

```
Polling from queue: 3.2 5.2 5.4
5.4 9.8
```

Summary

- Introduction
- The collections framework
- Wrapper Classes
  - Autoboxing and Auto-Unboxing
- Lists
- ArrayList and Iterator
- LinkedList
- Collections framework Algorithms
- Stack
- PriorityQueue