java.util and CollectionFramework

Prof. Ashish Bhatia

- This presentation is for teaching purpose only.
- This presentation may contain some material from books / api documentation / internet.
- No intention of breaking any rights or what so ever.

Date and Time Related Classes

- Date
- TimeZone
- Calendar
- GregorianCalendar

- Epoch : 1st Jan 1970 0:0:0.0 GMT
  - 1st January 1970 5:30:0.0 IST

Simple Program

```
1  import java.util.*;
2  class DateTest
3  {
4      public static void main(String args[])
5      {
6          Date d = new Date();
7          System.out.println(d);
8      }
9  }
```

????????? toString()

Constructor

Method	Description
`Date()`	Creates an object based on the current time of your computer clock to the nearest millisecond
`Date(long time)`	Creates an object based on the time value in milliseconds since 00:00:00 GMT on January 1, 1970 that is passed as an argument

Comparing Date

`after(Date earlier)`	Returns `true` if the current object represents a date that's later than the date represented by the argument `earlier`, and `false` otherwise.
`before(Date later)`	Returns `true` if the current object represents a date that's earlier than the date represented by the argument `later`, and `false` otherwise.
`equals(Object aDate)`	Returns `true` if the current object and the argument represent the same date and time, and `false` otherwise. This implies that they would both return the same value from `getTime()`.
`compareTo(Date date)`	This method is the result of the `Date` class implementing the `Comparable<Date>` interface. As you've seen in other contexts, this method returns a negative integer, zero, or a positive integer depending on whether the current object is less than, equal to, or greater than the argument. The presence of this method in the class means that you can use the `sort()` method in the `Arrays` class to sort an array of `Date` objects, or the `sort()` method in the `Collections` class to sort a collection of dates.

Calendar

- Abstract Class
- No public constructor
- day, month, year, hour, minutes , seconds, era
- Many int constants
- Many methods

AM
AM_PM
APRIL
AUGUST
DATE
DAY_OF_MONTH
DAY_OF_WEEK
DAY_OF_WEEK_IN_MONTH
DAY_OF_YEAR
DECEMBER
DST_OFFSET
ERA
FEBRUARY
FIELD_COUNT
FRIDAY
HOUR
HOUR_OF_DAY
JANUARY
JULY
JUNE
MARCH
MAY
MILLISECOND
MINUTE
MONDAY
MONTH
NOVEMBER
OCTOBER
PM
SATURDAY
SECOND
SEPTEMBER
SUNDAY
THURSDAY
TUESDAY
UNDECIMBER
WEDNESDAY
WEEK_OF_MONTH
WEEK_OF_YEAR
YEAR
ZONE_OFFSET

Some Methods

static Calendar getInstance( )	Returns a **Calendar** object for the default locale and time zone.
static Calendar getInstance(TimeZone *tz*)	Returns a **Calendar** object for the time zone specified by *tz*. The default locale is used.
static Calendar getInstance(Locale *locale*)	Returns a **Calendar** object for the locale specified by *locale*. The default time zone is used.
static Calendar getInstance(TimeZone *tz*, Locale *locale*)	Returns a **Calendar** object for the time zone specified by *tz* and the locale specified by *locale*.

TimeZone

- Class
- Maintains offset value
- ID for instances

- public static TimeZone getDefault()
- public static void setDefault(TimeZone zone)

static String[ ] getAvailableIDs( )	Returns an array of **String** objects representing the names of all time zones.
static String[ ] getAvailableIDs(int *timeDelta*)	Returns an array of **String** objects representing the names of all time zones that are *timeDelta* offset from GMT.

- public String getId()
- public String getDisplayName()
- public int getOffset(long date)
- public int getRawOffset
- public void setRawOffset(int offmilliseconds)
- public static TimeZone getTimeZone(String ID)

```
9        Calendar c = Calendar.getInstance();
10       System.out.println("Date = " + c.get(Calendar.DATE));
11       System.out.println("Month = " + c.get(Calendar.MONTH));
12       System.out.println("Year = " + c.get(Calendar.YEAR));
13
14       TimeZone z = TimeZone.getDefault();
15       System.out.println(z.getDisplayName());
```

```
F:\>java DateTest
Mon Oct 18 00:37:26 IST 2010
Date = 18
Month = 9
Year = 2010
India Standard Time
```

```
13
14        System.out.println("Hour " + c.get(Calendar.HOUR));
15        System.out.println("Minute " + c.get(Calendar.MINUTE));
16        System.out.println("Second " + c.get(Calendar.SECOND));
17
```

```
F:\>java DateTest
Mon Oct 18 00:40:33 IST 2010
Date = 18
Month = 9
Year = 2010
Hour 0
Minute 40
Second 33
India Standard Time
```

GregorianCalendar

- Concrete class of Calendar
- Two fields AD and BC for era
- 7 constructors
- GreogrianCalendar gc = new GregorianCalendar();
  - Current date and time in default locale and time zone

GregorianCalendar

GregorianCalendar(int *year*, int *month*, int *dayOfMonth*)
GregorianCalendar(int *year*, int *month*, int *dayOfMonth*, int *hours*, int *minutes*)
GregorianCalendar(int *year*, int *month*, int *dayOfMonth*, int *hours*, int *minutes*, int *seconds*)

Year = Number of year elapsed from 1900
Month = 0 indicating Jan

GregorianCalendar

GregorianCalendar(Locale *locale*)
GregorianCalendar(TimeZone *timeZone*)
GregorianCalendar(TimeZone *timeZone*, Locale *locale*)

- boolean isLeapYear(int year)
  - Test for leap year

DateFormat

```
java.lang.Object
  └ java.text.Format
      └ java.text.DateFormat
```

```
java.lang.Object
  └ java.text.Format
      └ java.text.DateFormat
          └ java.text.SimpleDateFormat
```

```java
DateFormat df = new SimpleDateFormat("dd-MMM-yyyy");
String s = df.format(d);

System.out.println("Formated date " + s);
```

```
Formated date 18-Oct-2010
```

Letter	Date or Time Component	Presentation	Examples
G	Era designator	Text	AD
y	Year	Year	1996; 96
M	Month in year	Month	July; Jul; 07
w	Week in year	Number	27
W	Week in month	Number	2
D	Day in year	Number	189
d	Day in month	Number	10
F	Day of week in month	Number	2
E	Day in week	Text	Tuesday; Tue
a	Am/pm marker	Text	PM
H	Hour in day (0-23)	Number	0
k	Hour in day (1-24)	Number	24
K	Hour in am/pm (0-11)	Number	0
h	Hour in am/pm (1-12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
z	Time zone	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	RFC 822 time zone	-0800

Array Class

```
java.lang.Object
  └ java.util.Arrays
```

- Contains various methods for manipulating arrays - mostly static methods
- Contains a static factory that allows arrays to be viewed as lists.
- fill, equals, sort, binarysearch, toString

fill method

- public static void fill(datatype[] array, datatype value)
- public static void fill(datatype[] array, int fromindex , int toindex, datatype value)

- datatype = Primitive Datatype + Object

Program

```
1   import java.util.*;
2   class ArrayTest
3   {
4       public static void main(String args[])
5       {
6           int arr[] = new int[5];
7
8           Arrays.fill(arr,45);
9           for(int i=0; i<arr.length;i++)
10          {
11              System.out.println(arr[i]);
12          }
13      }
14  }
```

Output

```
F:\>javac ArrayTest.java

F:\>java ArrayTest
45
45
45
45
45
```

fill()

static void fill(boolean *array*[ ], boolean *value*)
static void fill(byte *array*[ ], byte *value*)
static void fill(char *array*[ ], char *value*)
static void fill(double *array*[ ], double *value*)
static void fill(float *array*[ ], float *value*)
static void fill(int *array*[ ], int *value*)
static void fill(long *array*[ ], long *value*)
static void fill(short *array*[ ], short *value*)
static void fill(Object *array*[ ], Object *value*)

fill()

static void fill(boolean *array*[ ], int *start*, int *end*, boolean *value*)
static void fill(byte *array*[ ], int *start*, int *end*, byte *value*)
static void fill(char *array*[ ], int *start*, int *end*, char *value*)
static void fill(double *array*[ ], int *start*, int *end*, double *value*)
static void fill(float *array*[ ], int *start*, int *end*, float *value*)
static void fill(int *array*[ ], int *start*, int *end*, int *value*)
static void fill(long *array*[ ], int *start*, int *end*, long *value*)
static void fill(short *array*[ ], int *start*, int *end*, short *value*)
static void fill(Object *array*[ ], int *start*, int *end*, Object *value*)

equals()

- Return Boolean
- Take two arguments
- public static boolean equals(datatype [] a, datatype[] b)

- Compare each element and size of array

equals()

static boolean equals(byte *array1*[ ], byte *array2*[ ])
static boolean equals(char *array1*[ ], char *array2*[ ])
static boolean equals(double *array1*[ ], double *array2*[ ])
static boolean equals(float *array1*[ ], float *array2*[ ])
static boolean equals(int *array1*[ ], int *array2*[ ])
static boolean equals(long *array1*[ ], long *array2*[ ])
static boolean equals(short *array1*[ ], short *array2*[ ])
static boolean equals(Object *array1*[ ], Object *array2*[ ])

sort()

- Sort the array
- Ascending Order

```
static void sort(byte array[ ])
static void sort(char array[ ])
static void sort(double array[ ])
static void sort(float array[ ])
static void sort(int array[ ])
static void sort(long array[ ])
static void sort(short array[ ])
static void sort(Object array[ ])
static void sort(Object array[ ], Comparator c)
```

sort()

static void sort(byte *array*[ ], int *start*, int *end*)
static void sort(char *array*[ ], int *start*, int *end*)
static void sort(double *array*[ ], int *start*, int *end*)
static void sort(float *array*[ ], int *start*, int *end*)
static void sort(int *array*[ ], int *start*, int *end*)
static void sort(long *array*[ ], int *start*, int *end*)
static void sort(short *array*[ ], int *start*, int *end*)
static void sort(Object *array*[ ], int *start*, int *end*)
static void sort(Object *array*[ ], int *start*, int *end*, Comparator *c*)

binarySearch()

- Uses binary search method to find the index of element
- Prerequisites : Sorted Array

binarySearch()

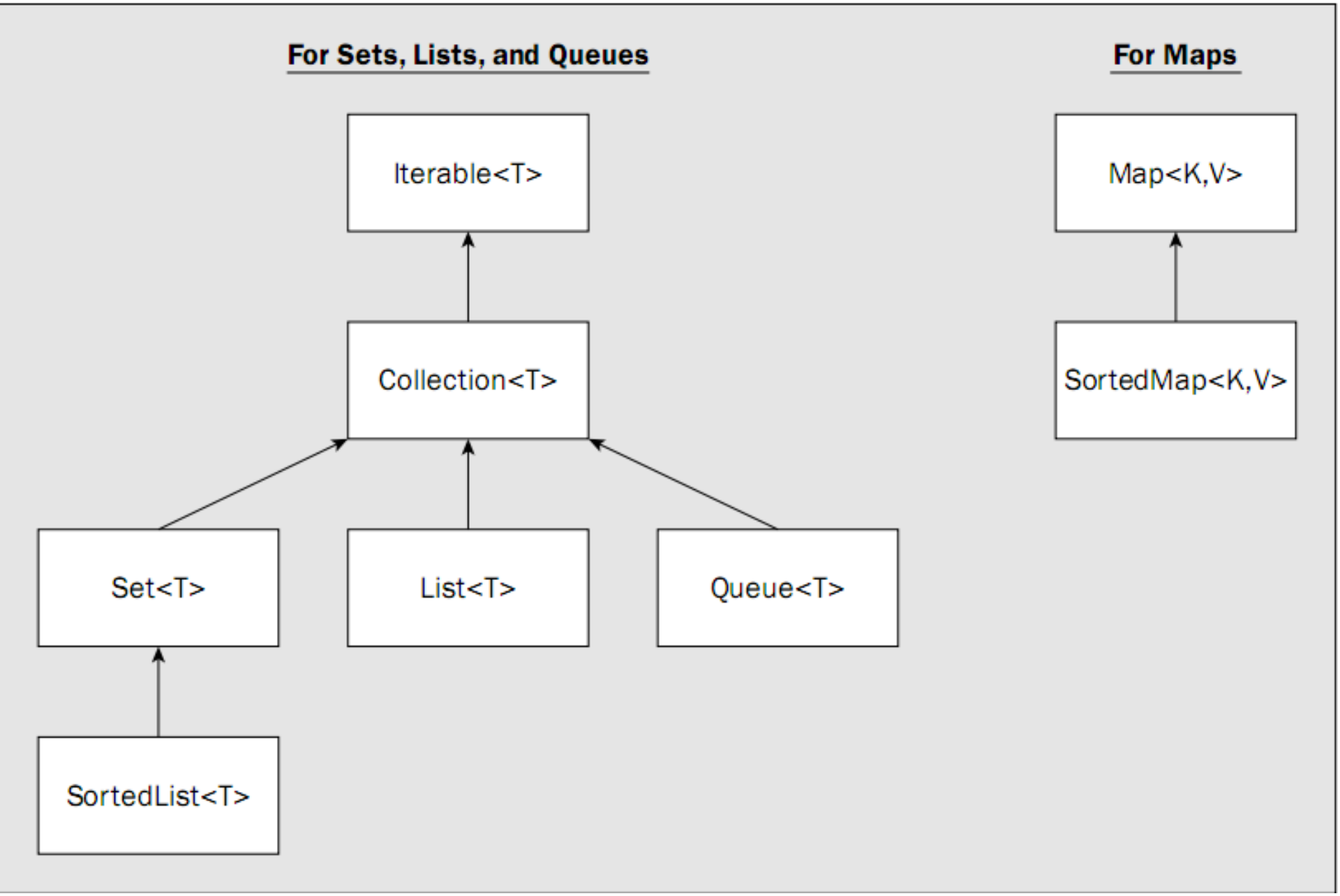
static int binarySearch(char[ ] *array*, char *value*)
static int binarySearch(double[ ] *array*, double *value*)
static int binarySearch(float[ ] *array*, float *value*)
static int binarySearch(int[ ] *array*, int *value*)
static int binarySearch(long[ ] *array*, long *value*)
static int binarySearch(short[ ] *array*, short *value*)
static int binarySearch(Object[ ] *array*, Object *value*)
static int binarySearch(Object[ ] *array*, Object *value*, Comparator *c*)

Collection Framework

- Standardizing and having interoperability between various data structures
- Introduced in Java 1.2
- Relies on Standard interfaces

Collection Framework

- Eight generics interfaces type
  - Determined the common methods for all collection class
- Operations
  - Basic Operations
  - Array Operations
  - Bulk Operations


For Sets, Lists, and Queues
For Maps
Iterable<T>
Collection<T>
Set<T>
List<T>
Queue<T>
SortedList<T>
Map<K,V>
SortedMap<K,V>

Basic Operations

boolean add(Object *obj*)	Adds *obj* to the invoking collection. Returns **true** if *obj* was added to the collection. Returns **false** if *obj* is already a member of the collection, or if the collection does not allow duplicates.
boolean remove(Object *obj*)	Removes one instance of *obj* from the invoking collection. Returns **true** if the element was removed. Otherwise, returns **false**.

Basic Operations

int size( )	Returns the number of elements held in the invoking collection.
void clear( )	Removes all elements from the invoking collection.
boolean contains(Object *obj*)	Returns **true** if *obj* is an element of the invoking collection. Otherwise, returns **false**.
boolean isEmpty( )	Returns **true** if the invoking collection is empty. Otherwise, returns **false**.
Iterator iterator( )	Returns an iterator for the invoking collection.

Array Operations

Object[ ] toArray( )	Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.
Object[ ] toArray(Object *array*[ ])	Returns an array containing only those collection elements whose type matches that of *array*. The array elements are copies of the collection elements. If the size of *array* equals the number of matching elements, these are returned in *array*. If the size of *array* is less than the number of matching elements, a new array of the necessary size is allocated and returned. If the size of *array* is greater than the number of matching elements, the array element following the last collection element is set to **null**. An **ArrayStoreException** is thrown if any collection element has a type that is not a subtype of *array*.

Bulk Operation

boolean addAll(Collection *c*)	Adds all the elements of *c* to the invoking collection. Returns **true** if the operation succeeded (i.e., the elements were added). Otherwise, returns **false**.
boolean removeAll(Collection *c*)	Removes all elements of *c* from the invoking collection. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
boolean retainAll(Collection *c*)	Removes all elements from the invoking collection except those in *c*. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
boolean containsAll(Collection *c*)	Returns **true** if the invoking collection contains all elements of *c*. Otherwise, returns **false**.

Iterator

- Lets you cycle through collection

Method	Description
boolean hasNext( )	Returns **true** if there are more elements. Otherwise, returns **false**.
Object next( )	Returns the next element. Throws **NoSuchElementException** if there is not a next element.
void remove( )	Removes the current element. Throws **IllegalStateException** if an attempt is made to call **remove( )** that is not preceded by a call to **next( )**.

Example

```
Collection collection = ...;
Iterator iterator = collection.iterator();
while (iterator.hasNext()) {
  Object element = iterator.next();
  if (removalCheck(element)) {
    iterator.remove();
  }
}
```

Set Interface

- Defines a set
- Collection of unique element [no duplication]
- Extends the Collection
  - All the methods
- HashSet and TreeSet are concrete classes
- AbstractSet : base abstract class

List Interface

- Extends Collection
- Insertion, updating, all are index based
- Index starts from 0
- It can store duplicates value
- It adds its own methods

Methods

void add(int *index*, Object *obj*)	Inserts *obj* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
boolean addAll(int *index*, Collection *c*)	Inserts all elements of *c* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns **true** if the invoking list changes and returns **false** otherwise.

Methods

Object get(int *index*)	Returns the object stored at the specified index within the invoking collection.
int indexOf(Object *obj*)	Returns the index of the first instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.
int lastIndexOf(Object *obj*)	Returns the index of the last instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.
ListIterator listIterator( )	Returns an iterator to the start of the invoking list.
ListIterator listIterator(int *index*)	Returns an iterator to the invoking list that begins at the specified index.

Methods

Object remove(int *index*)	Removes the element at position *index* from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.
Object set(int *index*, Object *obj*)	Assigns *obj* to the location specified by *index* within the invoking list.
List subList(int *start*, int *end*)	Returns a list that includes elements from *start* to *end*–1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

List Iterator

- public interface **ListIterator<E>** extends Iterator<E>
- A ListIterator has **no current element;** its ***cursor position*** **always lies between the element** that would be returned by a call **toprevious()** and the element that would be returned by a call to **next().**
- An iterator for a list of length n has n+1 possible cursor positions,

List Iterator

```
Element(0)   Element(1)   Element(2)   ... Element(n-1)
           ^            ^            ^                 ^
```

-

Method	Description
boolean hasNext( )	Returns **true** if there are more elements. Otherwise, returns **false**.
Object next( )	Returns the next element. Throws **NoSuchElementException** if there is not a next element.
void remove( )	Removes the current element. Throws **IllegalStateException** if an attempt is made to call **remove( )** that is not preceded by a call to **next( )**.

Method	Description
void add(Object *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**.
Object 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.
Object 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(Object *obj*)	Assigns *obj* to the current element. This is the element last returned by a call to either **next( )** or **previous( )**.

List Iterator

- In general, to use an iterator to cycle through thecontents of a collection, follow these steps:

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( ).

Vector Class

```
java.lang.Object
  └ java.util.AbstractCollection<E>
      └ java.util.AbstractList<E>
          └ java.util.Vector<E>
```

```
public class Vector<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Vector Class

- Dynamic Array
  - The size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created

- Synchronized
- Each vector tries to optimize storage management by maintaining a **capacity** and a **capacityIncrement**

Vector Class

- Modified from Java 1.2 ***[IS IT SAME AS JAVA 2]***
- To implement List and become part of Collection Framework

```
java.lang.Object
  └ java.util.AbstractCollection<E>
      └ java.util.AbstractList<E>
          └ java.util.Vector<E>
              └ java.util.Stack<E>
```

Vector Class Constructors

Vector( )

Initial Size 10

Vector(int *size*)

Vector(int *size*, int *incr*)

Vector(Collection *c*)

- The increment specifies the number of elements to allocate each time that a vector is resized upward.

Vector Class

- Starts with initial capacity
- Allocates more space for new objects once the capacity is reached
- It allocates more space that required so allocation is done minimal time
- Double size if nothing is specified
- What if incr size = 0 [Default Value]

Vector Class

Vector defines these protected data members:

```
int capacityIncrement;
int elementCount;
Object elementData[ ];
```

Vector Class

void addElement(Object *element*)	The object specified by *element* is added to the vector.
void insertElementAt(Object *element*, int *index*)	Adds *element* to the vector at the location specified by *index*.
boolean removeElement(Object *element*)	Removes *element* from the vector. If more than one instance of the specified object exists in the vector, then it is the first one that is removed. Returns **true** if successful and **false** if the object is not found.
void removeElementAt(int *index*)	Removes the element at the location specified by *index*.

Vector Class

void removeAllElements( )	Empties the vector. After this method executes, the size of the vector is zero.
void copyInto(Object *array*[ ])	The elements contained in the invoking vector are copied into the array specified by *array*.
void setElementAt(Object *element*, int *index*)	The location specified by *index* is assigned *element*.
Object elementAt(int *index*)	Returns the element at the location specified by *index*.

Vector Class

Object firstElement( )	Returns the first element in the vector.
int indexOf(Object *element*)	Returns the index of the first occurrence of *element*. If the object is not in the vector, –1 is returned.
int indexOf(Object *element*, int *start*)	Returns the index of the first occurrence of *element* at or after *start*. If the object is not in that portion of the vector, –1 is returned.

Vector Class

Object lastElement( )	Returns the last element in the vector.
int lastIndexOf(Object *element*)	Returns the index of the last occurrence of *element*. If the object is not in the vector, –1 is returned.
int lastIndexOf(Object *element*, int *start*)	Returns the index of the last occurrence of *element* before *start*. If the object is not in that portion of the vector, –1 is returned.

Vector Class

int capacity( )	Returns the capacity of the vector.
void ensureCapacity(int *size*)	Sets the minimum capacity of the vector to *size*.
int size( )	Returns the number of elements currently in the vector.
String toString( )	Returns the string equivalent of the vector.
void trimToSize( )	Sets the vector's capacity equal to the number of elements that it currently holds.
void setSize(int *size*)	Sets the number of elements in the vector to *size*. If the new size is less than the old size, elements are lost. If the new size is larger than the old size, **null** elements are added.

Vector Class

Enumeration elements( )	Returns an enumeration of the elements in the vector.
boolean isEmpty( )	Returns **true** if the vector is empty and returns **false** if it contains one or more elements.

```java
import java.util.*;

public class TestVector
{
    public static void main(String args[])
    {
        Vector v = new Vector(3,2);
        // size = 3 , increment = 2

        System.out.println("Initial Size " + v.size());
        System.out.println("Initial Capacity " + v.capacity());
    }
}
```

Now Lets add elements

```
v.addElement(new Integer(1));
v.addElement(new Integer(2));
v.addElement(new Integer(3));
v.addElement(new Integer(4));
```

- addElement(Object o)
- Size = ? Capacity = ?

- Size = 4 Capacity = 5
  - **WHY**

```
17          v.addElement(new Double(5.45));
```

- Size = ? Capacity = ?
- 5,5
- 5,8
- Any Other Option

```java
import java.util.*;

public class TestVector
{
    public static void main(String args[])
    {
        Vector v = new Vector(3,2);
        // size = 3 , increment = 2

        System.out.println("Initial Size " + v.size());
        System.out.println("Initial Capacity " + v.capacity());
        v.addElement(new Integer(1));
        v.addElement(new Integer(2));
        v.addElement(new Integer(3));
        v.addElement(new Integer(4));
        System.out.println(v.capacity() + " " + v.size());
        v.addElement(new Double(5.45));
        System.out.println( v.capacity() + " " + v.size());
    }
}
```

```
F:\>java -version
java version "1.7.0-ea"
Java(TM) SE Runtime Environment (build 1.7.0-ea-b59)
Java HotSpot(TM) Client VM (build 16.0-b03, mixed mode, sharing)

F:\>javac TestVector.java
Note: TestVector.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

F:\>
```

```
F:\>javac -Xlint TestVector.java
TestVector.java:7: warning: [rawtypes] found raw type: java.util.Vector
        Vector v = new Vector(3,2);
        ^
  missing type parameters for generic class java.util.Vector<E>
TestVector.java:7: warning: [rawtypes] found raw type: java.util.Vector
        Vector v = new Vector(3,2);
                       ^
  missing type parameters for generic class java.util.Vector<E>
TestVector.java:12: warning: [unchecked] unchecked call to addElement(E) as a me
mber of the raw type java.util.Vector
        v.addElement(new Integer(1));
```

Solution : Generics

```
Vector<Object> v = new Vector<Object>(3,2);
```

- <Object>
  - Specifies the data type

- Vector<String> v = new Vector<String> ();
- Vector<Rectangle> v = new Vector<Rectangle> ();

Printing Vector

```
Enumeration vEnum = v.elements();
while(vEnum.hasMoreElements())
    System.out.print(vEnum.nextElement() + " ");
System.out.println();
```

```
Iterator vItr = v.iterator();
System.out.println("\nElements in vector:");
while(vItr.hasNext())
    System.out.print(vItr.next() + " ");
System.out.println();
```

One Example ☺

```
3   abstract class Figure
4   {
5       int length,width;
6       Figure(int l, int w)
7       {
8           length = l;
9           width = w;
10      }
11      abstract int area();
12
13  }
```

```
14  class Rectangle extends Figure
15  {
16      Rectangle(int l, int w)
17      {
18          super(l,w);
19      }
20      int area()
21      {
22          return length*width;
23      }
24  }
```

Available at : http://www.a a

Lets Create Vector

```
Vector<Figure> v1 = new Vector<Figure>();

System.out.println("Initial Size " + v1.size());
System.out.println("Initial Capacity " + v1.capacity());
```

- Size = ?? Capacity = ??
- **DEFAULT = 10**

Will This Work !!!

```
Figure r1 = new Rectangle(40,50);
v1.add(r1);

Figure s1 = new Square(5);
v1.add(s1);

Rectangle r2 = new Rectangle(50,60);
v1.add(r2);

Square s2 = new Square(6);
v1.add(s2);
```

How to Print area ?

```
Iterator<Figure> vItr1 = v1.iterator();
System.out.println("\nElements in vector:");
while(vItr1.hasNext())
    System.out.print(vItr1.next().area() + " ");
System.out.println();
```

Stack

Method	Description
boolean empty( )	Returns **true** if the stack is empty, and returns **false** if the stack contains elements.
Object peek( )	Returns the element on the top of the stack, but does not remove it.
Object pop( )	Returns the element on the top of the stack, removing it in the process.
Object push(Object *element*)	Pushes *element* onto the stack. *element* is also returned.
int search(Object *element*)	Searches for *element* in the stack. If found, its offset from the top of the stack is returned. Otherwise, –1 is returned.

ArrayList

```
java.lang.Object
  └ java.util.AbstractCollection<E>
      └ java.util.AbstractList<E>
          └ java.util.ArrayList<E>
```

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

ArrayList

- Resizable-array implementation of the List interface.
- Implements all optional list operations, and permits all elements, including null.
- This class is roughly equivalent to Vector, except that it is unsynchronized.

Map Interface

- Continued to Next Presentation