Vietnam National University of HCMC
International University
School of Computer Science and Engineering

Generic Collections

(IT069IU)

Nguyen Trung Ky

ntky@hcmiu.edu.vn

it.hcmiu.edu.vn/user/ntky

Agenda's today

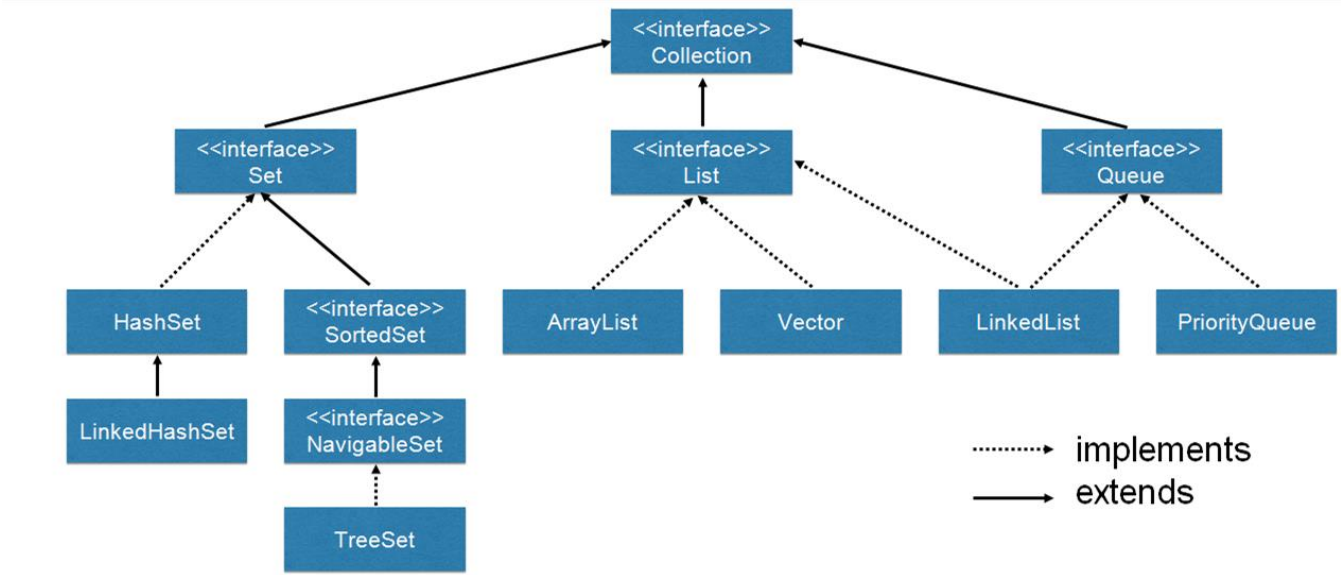
- **Java Generic Collections**
  - Type-Wrapper Classes for Primitive Types
    - Autoboxing vs Auto-unboxing
  - List
    - ArrayList
    - Vector
    - LinkedList

Collections!

- In this lecture, we are going to discuss of the **Java collections framework**, which contains many other **prebuilt generic data-structures**.
- Some **examples of collections** are **your favorite songs stored on your smartphone or media player**, your contacts list, **the cards you hold in a card game**, the members of your favorite sports team and the courses you take in school.

Collections

- Java **Collections** framework:
  - **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**.
  - **Prebuilt data structures.**
  - **A data structure is a particular way of organizing data so that it can be used effectively.**
  - **Interfaces** and **Methods** for manipulating those **data structures**.



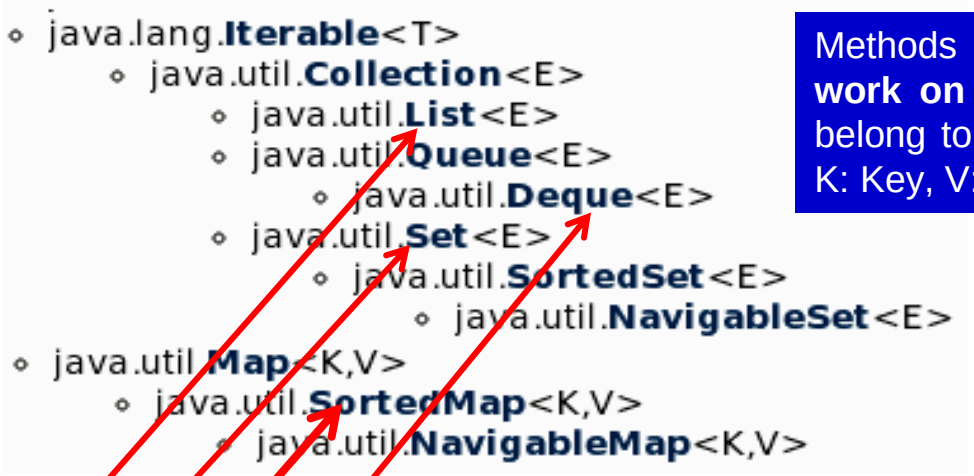
Interface	Description
`Collection`	The root interface in the collections hierarchy from which interfaces `Set`, `Queue` and `List` are derived.
`Set`	A collection that does not contain duplicates.
`List`	An ordered collection that can contain duplicate elements.
`Map`	A collection that associates keys to values and cannot contain duplicate keys.
`Queue`	Typically a first-in, first-out collection that models a waiting line; other orders can be specified.

Fig. 20.1 | Some collections-framework interfaces.

Collections Interface

- 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.**
- Interface **Collection** contains **bulk operations** for **adding**, **clearing** and **comparing** objects in a collection.
- A **Collection** can be converted to an array.
- Interface **Collection** provides a method that returns an **Iterator** object, which allows a program to **loop** the collection and modify elements from the collection during the iteration.

Collections Interface



Methods declared in these interfaces can **work on a list containing elements** which belong to arbitrary type. T: type, E: Element, K: Key, V: Value

Four types of group:

List can contain duplicate elements

Set can contain distinct elements only

Map can contain pairs <key, value>. Key of element is data for fast searching

Queue, Deque contains methods of restricted list.

Common methods on group are: Add, Remove, Search, Clear,…

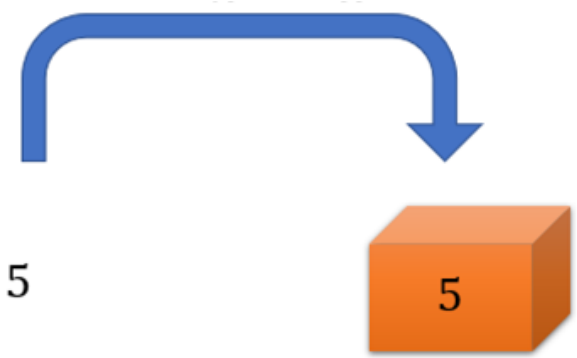
Common Methods of the interface Collection

Method	Description
add(Object x)	Adds x to this collection
addAll(Collection c)	Adds every element of c to this collection
clear()	Removes every element from this collection
contains(Object x)	Returns true if this collection contains x
containsAll(Collection c)	Returns true if this collection contains every element of c
isEmpty()	Returns true if this collection contains no elements
iterator()	Returns an Iterator over this collection (see below)
remove(Object x)	Removes x from this collection
removeAll(Collection c)	Removes every element in c from this collection
retainAll(Collection c)	Removes from this collection every element that is not in c
size()	Returns the number of elements in this collection
toArray()	Returns an array containing the elements in this collection

Elements can be stored using some ways such as an array, a tree, a hash table.
Sometimes, we want to traverse elements as a list → we need a list of references → Iterator

Collection is not 100% compatible with primitive data types

- But **Collections only stores reference to objects** so **it cannot store primitive data types.**
- So we need to figure out **how to store primitive data as an object of a class.**
  - Introducing **type-wrapper classes**!



int a = 5

Integer a = 5

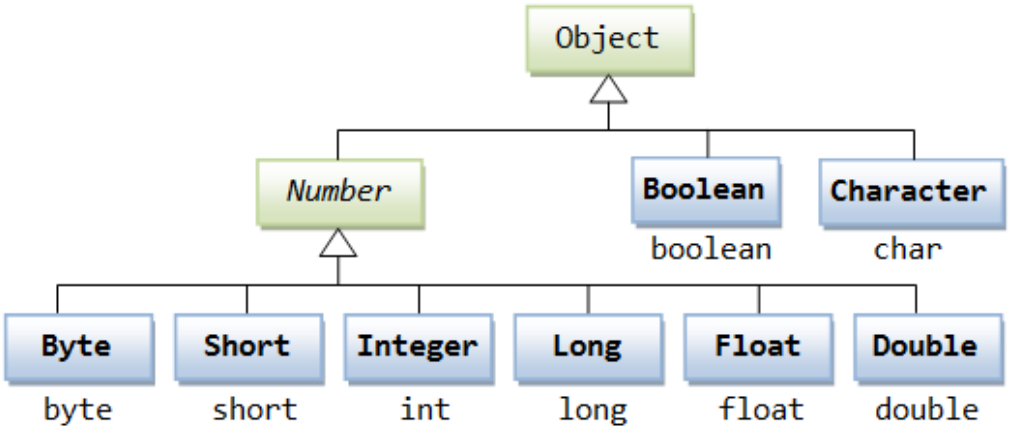
Type-Wrapper Classes for Primitive Types

- **Type-Wrapper classes provide a way to use primitive data types** (int, char, short, byte, etc) **as objects.**
- Each primitive type has a corresponding **type-wrapper class**:
  - **Boolean, Byte, Character, Double, Float, Integer, Long** and **Short.**
- Each type-wrapper class enables you to **manipulate primitive-type values** as **objects**.
- They are in **package java.lang.**

- Each of the numeric type-wrapper classes extends **superclass Number**.
- The type-wrapper classes are **final classes**, so you **cannot extend** them.

Wrapper Classes

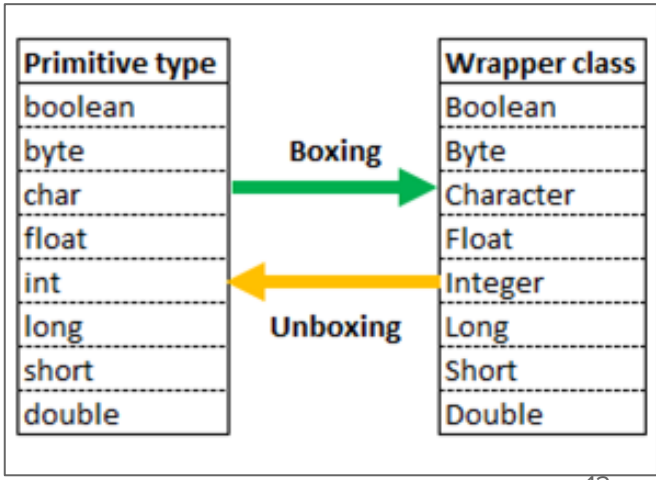
Primitive Data Type	Wrapper Class
double	*Double*
float	*Float*
long	*Long*
int	*Integer*
short	*Short*
byte	*Byte*
char	*Character*
boolean	*Boolean*



Autoboxing and Auto-Unboxing

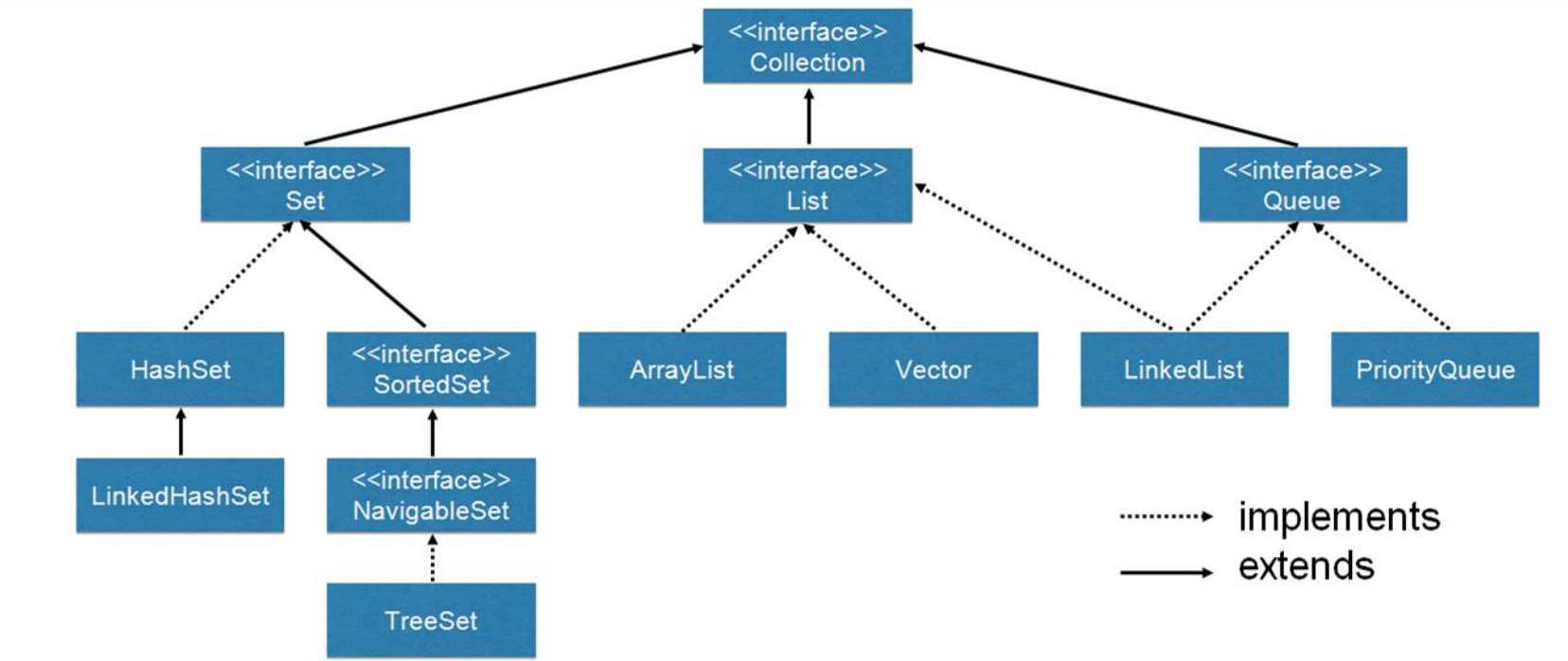
- A **boxing conversion** converts a value of a primitive type to an object of the corresponding type-wrapper class.
- An **unboxing conversion** converts an object of a type-wrapper class to a value of the corresponding primitive type.
- These **conversions** can be **performed automatically** (called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

```
Integer[] integerArray = new Integer[5]; // create integerArray
integerArray[0] = 10; // assign Integer 10 to integerArray[0]
int value = integerArray[0]; // get int value of Integer
```



Let's talk about List!

- **List interface**:
  - **ArrayList**, **Vector**, and **LinkedList**



List

- A **List** is a Collection that can contain **duplicate elements**.
- **Fist index** of a List is **zero**.
- A List keeps it elements in the order in which they were added.
- 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**, **Vector, and LinkedList**.

ArrayList

- The Java API provides several **predefined data structures**, called **collections**, used to **store groups of related objects**.
- ArrayList is a **resizable array** implementation in java. ArrayList **grows dynamically** and ensures that there is **always a space to add elements**.
- The collection class **ArrayList<T>** (package java.util) provides a **convenient solution** to this problem—it can **dynamically change its size** to accommodate **more elements**. The **T** (by convention) is a **placeholder**.

ArrayList to store String elements:

```
ArrayList<String> list;
```

ArrayList to store Integer elements:

```
ArrayList<Integer> integers;
```

ArrayList Methods

Method	Description
`add`	Adds an element to the *end* of the `ArrayList`.
`clear`	Removes all the elements from the `ArrayList`.
`contains`	Returns `true` if the `ArrayList` contains the specified element; otherwise, returns `false`.
`get`	Returns the element at the specified index.
`indexOf`	Returns the index of the first occurrence of the specified element in the `ArrayList`.
`remove`	Overloaded. Removes the first occurrence of the specified value or the element at the specified index.
`size`	Returns the number of elements stored in the `ArrayList`.
`trimToSize`	Trims the capacity of the `ArrayList` to the current number of elements.

Fig. 7.23 | Some methods and properties of class `ArrayList<T>`.

ArrayList Example

```
1  // Fig. 7.24: ArrayListCollection.java
2  // Generic ArrayList<T> collection demonstration.
3  import java.util.ArrayList;
4
5  public class ArrayListCollection
6  {
7     public static void main(String[] args)
8     {
9        // create a new ArrayList of Strings with an initial capacity of 10
10       ArrayList<String> items = new ArrayList<String>();
11
12       items.add("red"); // append an item to the list
13       items.add(0, "yellow"); // insert "yellow" at index 0
14
15       // header
16       System.out.print(
17          "Display list contents with counter-controlled loop:");
18
19       // display the colors in the list
20       for (int i = 0; i < items.size(); i++)
21          System.out.printf(" %s", items.get(i));
22
23       // display colors using enhanced for in the display method
24       display(items,
25          "%nDisplay list contents with enhanced for statement:");
26
27       items.add("green"); // add "green" to the end of the list
28       items.add("yellow"); // add "yellow" to the end of the list
29       display(items, "List with two new elements:");
30
31       items.remove("yellow"); // remove the first "yellow"
32       display(items, "Remove first instance of yellow:");
```

Output:

```
Display list contents with counter-controlled loop: yellow red
Display list contents with enhanced for statement: yellow red
List with two new elements: yellow red green yellow
Remove first instance of yellow: red green yellow
Remove second list element (green): red yellow
"red" is in the list
Size: 2
```

```
33
34          items.remove(1); // remove item at index 1
35          display(items, "Remove second list element (green):");
36
37          // check if a value is in the List
38          System.out.printf("\"red\" is %sin the list%n",
39             items.contains("red") ? "": "not ");
40
41          // display number of elements in the List
42          System.out.printf("Size: %s%n", items.size());
43       }
44
45       // display the ArrayList's elements on the console
46       public static void display(ArrayList<String> items, String header)
47       {
48          System.out.printf(header); // display header
49
50          // display each element in items
51          for (String item : items)
52             System.out.printf(" %s", item);
53
54          System.out.println();
55       }
56    } // end class ArrayListCollection
```

Output:

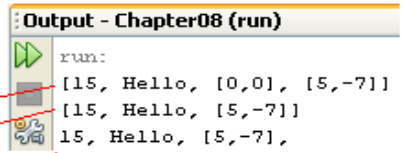
```
Display list contents with counter-controlled loop: yellow red
Display list contents with enhanced for statement: yellow red
List with two new elements: yellow red green yellow
Remove first instance of yellow: red green yellow
Remove second list element (green): red yellow
"red" is in the list
Size: 2
```

Vector Example

java.util.**Vector**<E> (implements java.lang.Cloneable, java.util.List<E>, java.util.RandomAccess, java.io.Serializable)

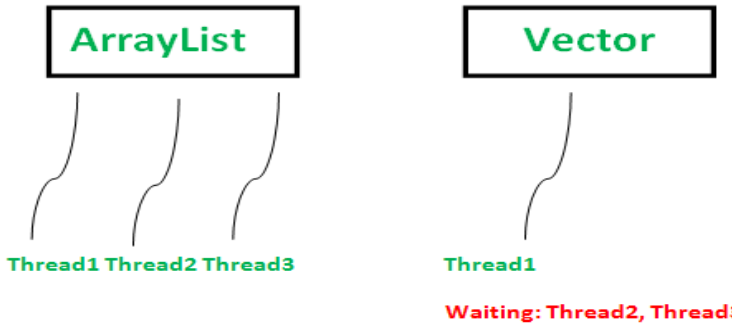
The Vector class is obsolete from Java 1.6 but it is still introduced because it is a parameter in the constructor of the javax.swing.JTable class, a class will be introduced in GUI programming.

```
import java.util.Vector;
class Point {
  int x,y;
  Point() { x=0; y=0; }
  Point(int xx, int yy){
      x=xx; y=yy;
  }
  public String toString() { return "[" + x + "," + y + "]";}
}
public class UseVector {
  public static void main(String[] args) {
      Vector v = new Vector();
      v.add(15);
      v.add("Hello");
      v.add(new Point());
      v.add(new Point(5,-7));
      System.out.println(v);
      v.remove(2);
      System.out.println(v);
      for (int i=0;i<v.size();i++) System.out.print(v.get(i) + ", ");
      System.out.println();
  }
}
```



ArrayList vs Vector

- Both are **resizable-array** implementations of List.
- The **primary difference** between **ArrayList** and **Vector:**
  - **Vectors** are **synchronized** by default.
  - **ArrayLists** are **not synchronized**.
- For example, if one thread is performing an add operation, then there can be another thread performing a remove operation in a multithreading environment.
- **Vector** is **synchronized**, which means only **one thread at a time can access the code**, while **ArrayList** is **not synchronized**, which means **multiple threads can work on ArrayList at the same time**. Therefore, **ArrayList is faster**.
- 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.



Common Methods of List

- List method ***add(Object x)*** adds an item to the end of a list.
- List method ***size()*** returns the number of elements.
- List method ***get(int index)*** 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(Object x)*** determine whether a Collection contains a specified element.
- Iterator method ***remove(Object x)*** removes the current element from a Collection.

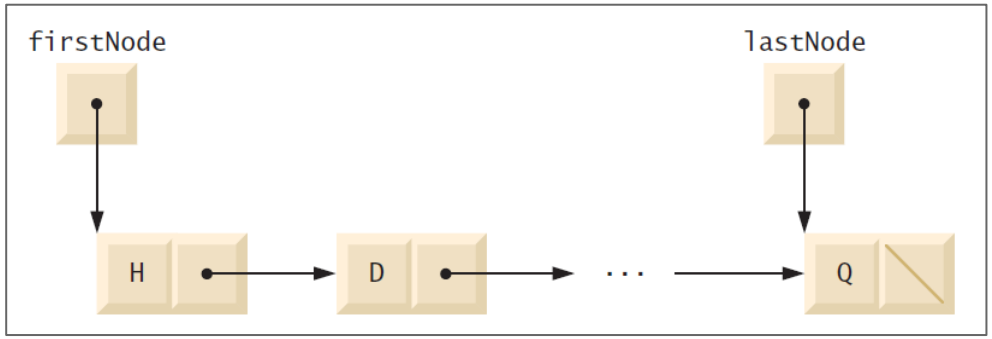
```java
// Fig. 20.2: CollectionTest.java
// Collection interface demonstrated via an ArrayList object.
import java.util.List;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

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

      // add elements in removeColors array to removeList
      String[] removeColors = { "RED", "WHITE", "BLUE" };
      List< String > removeList = new ArrayList< String >();

      for ( String color : removeColors )
         removeList.add( color );

      // output list contents
      System.out.println( "ArrayList: " );

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

      // remove from list the colors contained in removeList
      removeColors( list, removeList );

      // output list contents
      System.out.println( "\n\nArrayList after calling removeColors: " );

      for ( String color : list )
         System.out.printf( "%s ", color );
   } // end main

   // remove colors specified in collection2 from collection1
   private static void removeColors( Collection< String > collection1,
      Collection< String > collection2 )
   {
      // get iterator
      Iterator< String > iterator = collection1.iterator();

      // loop while collection has items
      while ( iterator.hasNext() )
      {
         if ( collection2.contains( iterator.next() ) )
            iterator.remove(); // remove current Color
      } // end while
   } // end method removeColors
} // end class CollectionTest
```

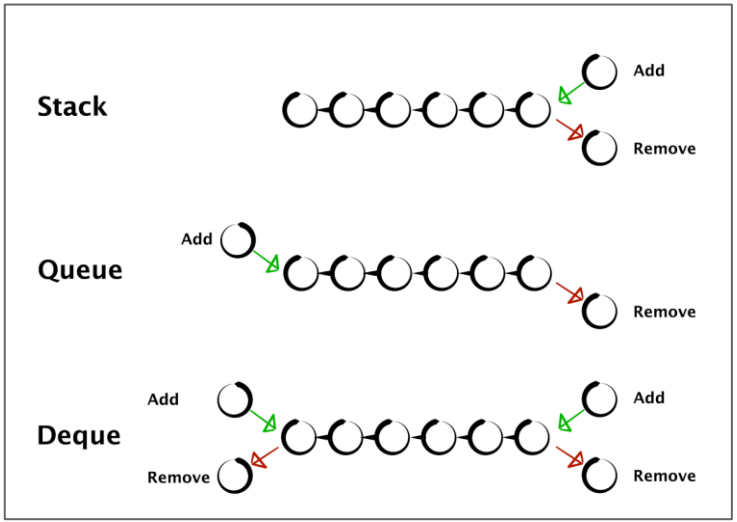
```
ArrayList:
MAGENTA RED WHITE BLUE CYAN

ArrayList after calling removeColors:
MAGENTA CYAN
```

Fig. 20.2 | `Collection` interface demonstrated via an `ArrayList` object. (Part 3 of 3.)

LinkedList

- **Inserting an element** between existing elements of an **ArrayList** or **Vector** is an **inefficient** operation.
- A **LinkedList** enables efficient **insertion** (or **removal**) of elements in the **middle** of a collection.
- A **LinkedList** can be used to create **stacks**, **queues** and **deques** (double-ended queues).





Other Common Methods of List

- List method **addAll(Object x)** appends all elements of a collection to the end of a List.
- List method **listIterator(int index)** gets A List's bidirectional iterator.
- String method **toUpperCase()** gets an uppercase version of a String.
- List method **subList(int start, int end)** obtains a portion of a List.
  - This is a so-called range-view method, which enables the program to view a portion of the list.
- List method **clear()** remove all the elements of a List.
- ListIterator method **hasPrevious()** determines whether there are more elements while traversing the list backward.
- ListIterator method **previous()** gets the previous element from the list.

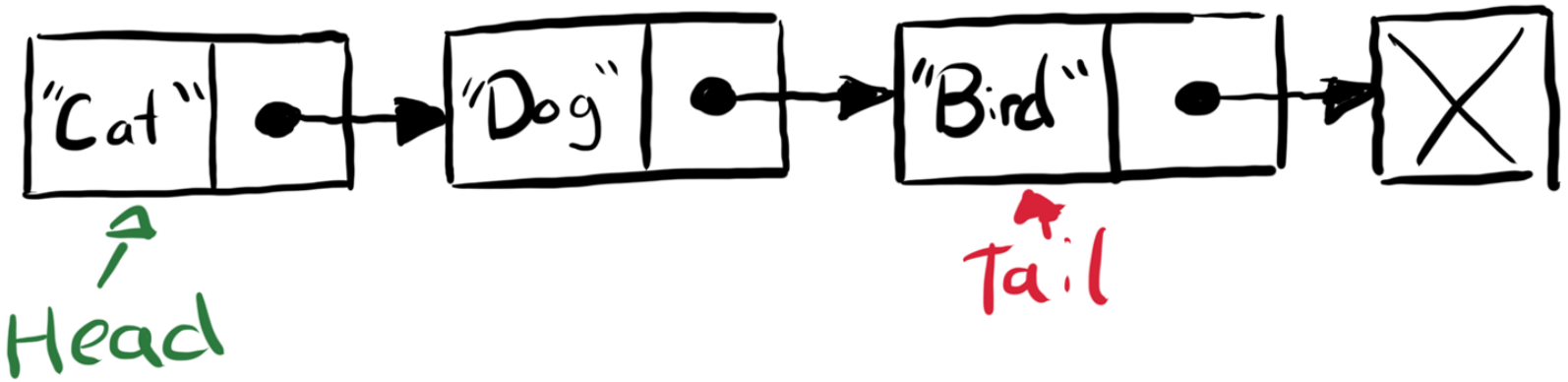
```java
// Fig. 20.3: ListTest.java
// Lists, LinkedLists and ListIterators.
import java.util.List;
import java.util.LinkedList;
import java.util.ListIterator;

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 string
      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
      printReversedList( list1 ); // print list in reverse order
   } // end main

   // 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

   // 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

   // print reversed list
   private static void printReversedList( List< String > list )
   {
      ListIterator< String > iterator = list.listIterator( list.size() );

      System.out.println( "\nReversed List:" );

      // print list in reverse order
      while ( iterator.hasPrevious() )
         System.out.printf( "%s ", iterator.previous() );
   } // end method printReversedList
} // end class ListTest
```

```
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

Reversed List:
SILVER GRAY BLUE BROWN WHITE BLUE GREEN YELLOW BLACK
```

Fig. 20.3 | `Lists`, `LinkedLists` and `ListIterators`. (Part 5 of 5.)

LinkedList Methods

- LinkedList method **addLast** adds an element to the end of a List.
- LinkedList method **add** also adds an element to the end of a List.
- LinkedList method **addFirst** adds an element to the beginning of a List.



Convert from Array to List, and List to Array

- Class Arrays provides static method **asList** to view an array as a List collection.
  - A List view allows you to manipulate the array as if it were a list.
  - This is useful for adding the elements in an array to a collection and for sorting array elements.
- Any modifications made through the List view change the array, and any modifications made to the array change the List view.
- List method **toArray** gets an array from a List collection.

LinkedList To Array

```
1  // Fig. 20.4: UsingToArray.java
2  // Viewing arrays as Lists and converting Lists to arrays.
3  import java.util.LinkedList;
4  import java.util.Arrays;
5
6  public class UsingToArray
7  {
8     // creates a LinkedList, adds elements and converts to array
9     public static void main( String[] args )
10    {
11       String[] colors = { "black", "blue", "yellow" };
12
13       LinkedList< String > links =
14          new LinkedList< String >( Arrays.asList( colors ) );
15
16       links.addLast( "red" ); // add as last item
17       links.add( "pink" ); // add to the end
18       links.add( 3, "green" ); // add at 3rd index
19       links.addFirst( "cyan" ); // add as first item
20
21       // get LinkedList elements as an array
22       colors = links.toArray( new String[ links.size() ] );
23
24       System.out.println( "colors: " );
25
26       for ( String color : colors )
27          System.out.println( color );
28    } // end main
29 } // end class UsingToArray
```

```
colors:
cyan
black
blue
yellow
green
red
pink
```

Collection Methods

Class Collections provides several high-performance algorithms for manipulating collection elements. The algorithms are implemented as static methods.

Method	Description
`sort`	Sorts the elements of a `List`.
`binarySearch`	Locates an object in a `List`, using the high-performance binary search algorithm which we introduced in Section 7.15 and discuss in detail in Section 19.4.
`reverse`	Reverses the elements of a `List`.
`shuffle`	Randomly orders a `List`'s elements.
`fill`	Sets every `List` element to refer to a specified object.
`copy`	Copies references from one `List` into another.
`min`	Returns the smallest element in a `Collection`.
`max`	Returns the largest element in a `Collection`.
`addAll`	Appends all elements in an array to a `Collection`.
`frequency`	Calculates how many collection elements are equal to the specified element.
`disjoint`	Determines whether two collections have no elements in common.

Method sort

- **Method sort** sorts the elements of a **List**
  - The elements must implement the **Comparable** interface.
  - The order is determined by the natural order of the elements' type as implemented by a **compareTo** method.
  - For example, the natural order for numeric values is ascending order, and the natural order for Strings is based on their lexicographical ordering.
  - Method compareTo is declared in interface Comparable and is sometimes called the **natural comparison method**.
  - The sort call may specify as a second argument a **Comparator** object that determines an alternative ordering of the elements.

Collection.sort()

```
1  // Fig. 20.6: Sort1.java
2  // Collections method sort.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  public class Sort1
8  {
9     public static void main( String[] args )
10    {
11       String[] suits = { "Hearts", "Diamonds", "Clubs", "Spades" };
12
13       // Create and display a list containing the suits array elements
14       List< String > list = Arrays.asList( suits ); // create List
15       System.out.printf( "Unsorted array elements: %s\n", list );
16
17       Collections.sort( list ); // sort ArrayList
18
19       // output list
20       System.out.printf( "Sorted array elements: %s\n", list );
21    } // end main
22 } // end class Sort1
```

```
Unsorted array elements: [Hearts, Diamonds, Clubs, Spades]
Sorted array elements: [Clubs, Diamonds, Hearts, Spades]
```

Collection.sort() in Reverse Order

- The Comparator interface is used for sorting a Collection’s elements in a different order.
- The static **Collections method reverseOrder** returns a Comparator object that orders the collection’s elements in reverse order.

```java
// Fig. 20.7: Sort2.java
// Using a Comparator object with method sort.
import java.util.List;
import java.util.Arrays;
import java.util.Collections;

public class Sort2
{
   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 );

      // sort in descending order using a comparator
      Collections.sort( list, Collections.reverseOrder() );

      // output List elements
      System.out.printf( "Sorted list elements: %s\n", list );
   } // end main
} // end class Sort2
```

```
Unsorted array elements: [Hearts, Diamonds, Clubs, Spades]
Sorted list elements: [Spades, Hearts, Diamonds, Clubs]
```

Method shuffle

- Method shuffle randomly orders a List's elements.

```java
// Fig. 20.10: DeckOfCards.java
// Card shuffling and dealing with Collections method shuffle.
import java.util.List;
import java.util.Arrays;
import java.util.Collections;

// class to represent a Card in a deck of cards
class Card
{
   public static enum Face { Ace, Deuce, Three, Four, Five, Six,
      Seven, Eight, Nine, Ten, Jack, Queen, King  };
   public static enum Suit { Clubs, Diamonds, Hearts, Spades };

   private final Face face; // face of card
   private final Suit suit; // suit of card

   // two-argument constructor
   public Card( Face cardFace, Suit cardSuit )
   {
       face = cardFace; // initialize face of card
       suit = cardSuit; // initialize suit of card
   } // end two-argument Card constructor

   // return face of the card
   public Face getFace()
   {
      return face;
   } // end method getFace

   // return suit of Card
   public Suit getSuit()
   {
      return suit;
   } // end method getSuit

   // return String representation of Card
   public String toString()
   {
      return String.format( "%s of %s", face, suit );
   } // end method toString
} // end class Card
```

```java
// class DeckOfCards declaration
public class DeckOfCards
{
   private List< Card > list; // declare List that will store Cards

   // set up deck of Cards and shuffle
   public DeckOfCards()
   {
      Card[] deck = new Card[ 52 ];
      int count = 0; // number of cards

      // populate deck with Card objects
      for ( Card.Suit suit : Card.Suit.values() )
      {
         for ( Card.Face face : Card.Face.values() )
         {
            deck[ count ] = new Card( face, suit );
            ++count;
         } // end for
      } // end for

      list = Arrays.asList( deck ); // get List
      Collections.shuffle( list );  // shuffle deck
   } // end DeckOfCards constructor

   // output deck
   public void printCards()
   {
      // display 52 cards in two columns
      for ( int i = 0; i < list.size(); i++ )
         System.out.printf( "%-19s%s", list.get( i ),
            ( ( i + 1 ) % 4 == 0 ) ? "\n" : "" );
   } // end method printCards

   public static void main( String[] args )
   {
      DeckOfCards cards = new DeckOfCards();
      cards.printCards();
   } // end main
} // end class DeckOfCards
```

```
Deuce of Clubs      Six of Spades       Nine of Diamonds    Ten of Hearts
Three of Diamonds   Five of Clubs       Deuce of Diamonds   Seven of Clubs
Three of Spades     Six of Diamonds     King of Clubs       Jack of Hearts
Ten of Spades       King of Diamonds    Eight of Spades     Six of Hearts
Nine of Clubs       Ten of Diamonds     Eight of Diamonds   Eight of Hearts
Ten of Clubs        Five of Hearts      Ace of Clubs        Deuce of Hearts
Queen of Diamonds   Ace of Diamonds     Four of Clubs       Nine of Hearts
Ace of Spades       Deuce of Spades     Ace of Hearts       Jack of Diamonds
Seven of Diamonds   Three of Hearts     Four of Spades      Four of Diamonds
Seven of Spades     King of Hearts      Seven of Hearts     Five of Diamonds
Eight of Clubs      Three of Clubs      Queen of Clubs      Queen of Spades
Six of Clubs        Nine of Spades      Four of Hearts      Jack of Clubs
Five of Spades      King of Spades      Jack of Spades      Queen of Hearts
```

Fig. 20.10 | Card shuffling and dealing with `Collections` method `shuffle`. (Part 5 of 5.)

Methods reverse, fill, copy, max and min

- Collections method **reverse** reverses the order of the elements in a List
- Method **fill** overwrites elements in a List with a specified value.
- Method **copy** takes two arguments—a destination List and a source List.
  - Each source List element is copied to the destination List.
  - The destination List must be at least as long as the source List; otherwise, an IndexOutOfBoundsException occurs.
  - If the destination List is longer, the elements not overwritten are unchanged.
- Methods **min** and **max** each operate on any Collection.
  - Method min returns the smallest element in a Collection, and method max returns the largest element in a Collection.

```java
// Fig. 20.11: Algorithms1.java
// Collections methods reverse, fill, copy, max and min.
import java.util.List;
import java.util.Arrays;
import java.util.Collections;

public class Algorithms1
{
   public static void main( String[] args )
   {
      // create and display a List< Character >
      Character[] letters = { 'P', 'C', 'M' };
      List< Character > list = Arrays.asList( letters ); // get List
      System.out.println( "list contains: " );
      output( list );

      // reverse and display the List< Character >
      Collections.reverse( list ); // reverse order the elements
      System.out.println( "\nAfter calling reverse, list contains: " );
      output( list );

      // create copyList from an array of 3 Characters
      Character[] lettersCopy = new Character[ 3 ];
      List< Character > copyList = Arrays.asList( lettersCopy );

      // copy the contents of list into copyList
      Collections.copy( copyList, list );
      System.out.println( "\nAfter copying, copyList contains: " );
      output( copyList );

      // fill list with Rs
      Collections.fill( list, 'R' );
      System.out.println( "\nAfter calling fill, list contains: " );
      output( list );
   } // end main

   // output List information
   private static void output( List< Character > listRef )
   {
      System.out.print( "The list is: " );

      for ( Character element : listRef )
         System.out.printf( "%s ", element );

      System.out.printf( "\nMax: %s", Collections.max( listRef ) );
      System.out.printf( "  Min: %s\n", Collections.min( listRef ) );
   } // end method output
} // end class Algorithms1
```

```
list contains:
The list is: P C M
Max: P  Min: C

After calling reverse, list contains:
The list is: M C P
Max: P  Min: C

After copying, copyList contains:
The list is: M C P
Max: P  Min: C

After calling fill, list contains:
The list is: R R R
Max: R  Min: R
```

Methods addAll, frequency and disjoint

- Collections method **addAll** takes two arguments—a Collection into which to insert the new element(s) and an array that provides elements to be inserted.
- Collections method **frequency** takes two arguments—a Collection to be searched and an Object to be searched for in the collection.
  - Method **frequency** returns the number of times that the second argument appears in the collection.
- Collections method **disjoint** takes two Collections and returns true if they have no elements in common.

```java
// Fig. 20.13: Algorithms2.java
// Collections methods addAll, frequency and disjoint.
import java.util.ArrayList;
import java.util.List;
import java.util.Arrays;
import java.util.Collections;

public class Algorithms2
{
   public static void main( String[] args )
   {
      // initialize list1 and list2
      String[] colors = { "red", "white", "yellow", "blue" };
      List< String > list1 = Arrays.asList( colors );
      ArrayList< String > list2 = new ArrayList< String >();

      list2.add( "black" ); // add "black" to the end of list2
      list2.add( "red" ); // add "red" to the end of list2
      list2.add( "green" ); // add "green" to the end of list2

      System.out.print( "Before addAll, list2 contains: " );

      // display elements in list2
      for ( String s : list2 )
         System.out.printf( "%s ", s );

      Collections.addAll( list2, colors ); // add colors Strings to list2

      System.out.print( "\nAfter addAll, list2 contains: " );

      // display elements in list2
      for ( String s : list2 )
         System.out.printf( "%s ", s );

      // get frequency of "red"
      int frequency = Collections.frequency( list2, "red" );
      System.out.printf(
         "\nFrequency of red in list2: %d\n", frequency );

      // check whether list1 and list2 have elements in common
      boolean disjoint = Collections.disjoint( list1, list2 );

      System.out.printf( "list1 and list2 %s elements in common\n",
         ( disjoint ? "do not have" : "have" ) );
   } // end main
```

```
Before addAll, list2 contains: black red green
After addAll, list2 contains: black red green red white yellow blue
Frequency of red in list2: 2
list1 and list2 have elements in common
```

Three ways to loop through a Collection

- The classic array index loop where you have the index of the element:

```
for (int i = 0; i < collection.length; i++) {
  type array_element = collection.get(index);
}
```

- Loop through with Iterator where you do not need any index of elements, and you can access them or remove or modify them:

```
for (Iterator iterator = collection.iterator(); iterator.hasNext();) {
  type type = (type) iterator.next();
}
```

- Loop through with Iterator where you do not need any index of elements, and to only access them without removing or modifying them:

```
for (iterable_type iterable_element : collection) {

}
```

Thank you for your listening!

“Motivation is what gets you started.
Habit is what keeps you going!”

Jim Ryun