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

Generic Collections

(IT069IU)

Le Duy Tan, Ph.D.

ldtan@hcmiu.edu.vn

leduytanit.com

Previously,

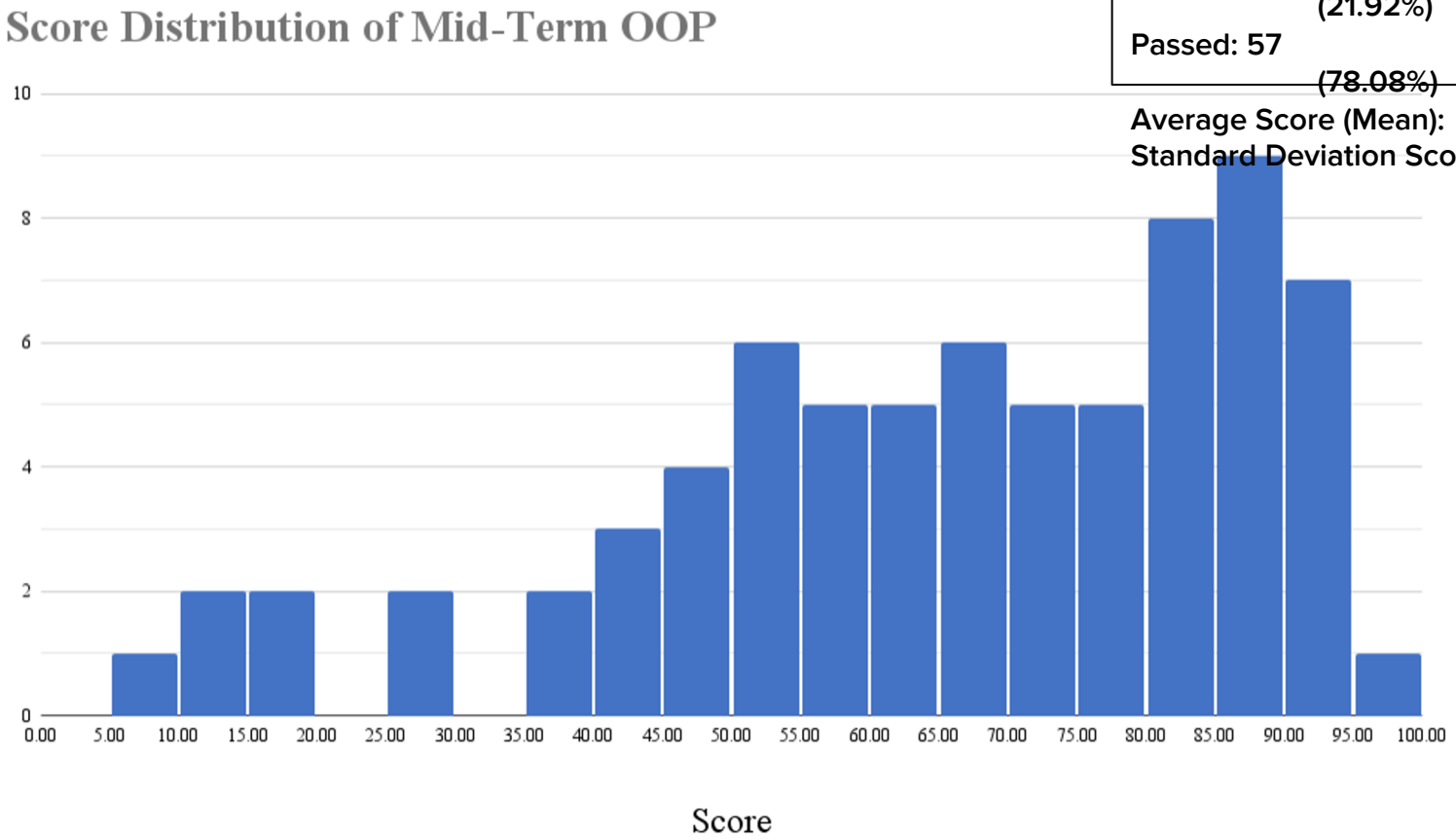
- **Mid-term Exam**
  - Possible Topics
  - Format
- **GUI** in Java
  - Swing Components
  - Swing vs JavaFX vs AWT
  - JOption-Panel (DialogBox)
  - Java 2D API
    - Java Coordinate System
    - Draw Lines
    - Draw Rectangles and Ovals
    - Draw Colors
    - Draw Polygons and Polylines
  - Text Fields, Buttons and Menu Nagivations.
  - Event Handler (Mouse & Keyboard)
  - Layout:
    - BorderLayout Frame & GridLayoutFrame

Agenda

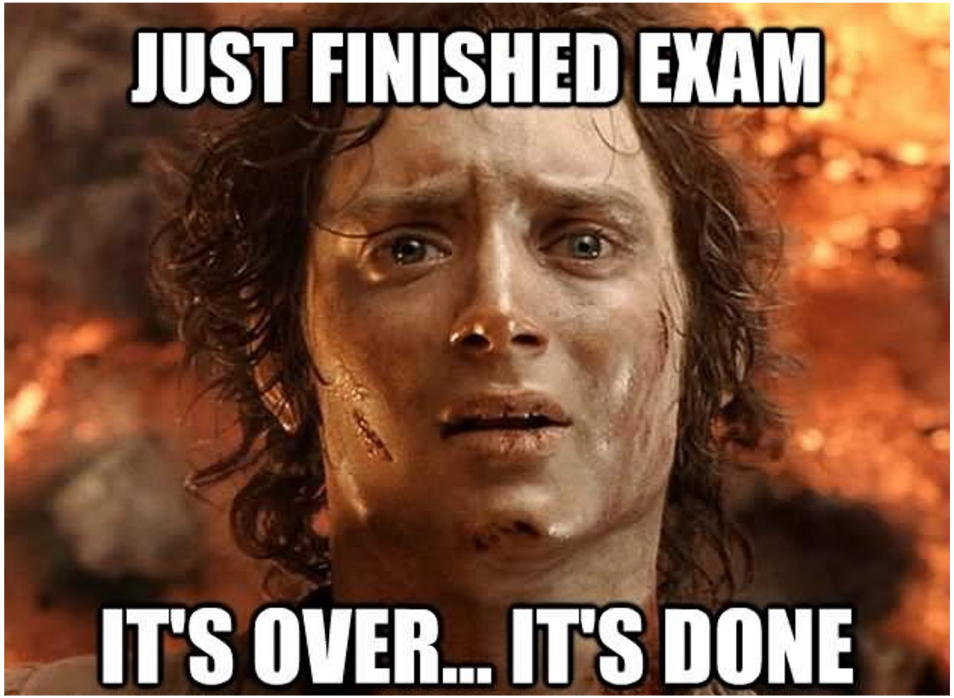
- **Mid-term Exam Result**
  - Score Distribution
  - Statistics
  - Solutions
- **Java Generic Collections**
  - Type-Wrapper Classes for Primitive Types
    - Autoboxing vs Auto-unboxing
  - List
    - ArrayList
    - Vector
    - LinkedList
  - Sets
    - HashSet
    - TreeSet
  - Maps
    - Hashtable
    - HashMap
    - TreeMaps
- HackerRank
  - Introduction
  - Coding Challenge

Mid-term Result

Attended: 73 (91.25%)
Missed: 7 (9.59%)
Failed: 16 (21.92%)
Passed: 57 (78.08%)
Average Score (Mean): 64.3
Standard Deviation Score: 22.63



Mid-Term Exam Coding Solution

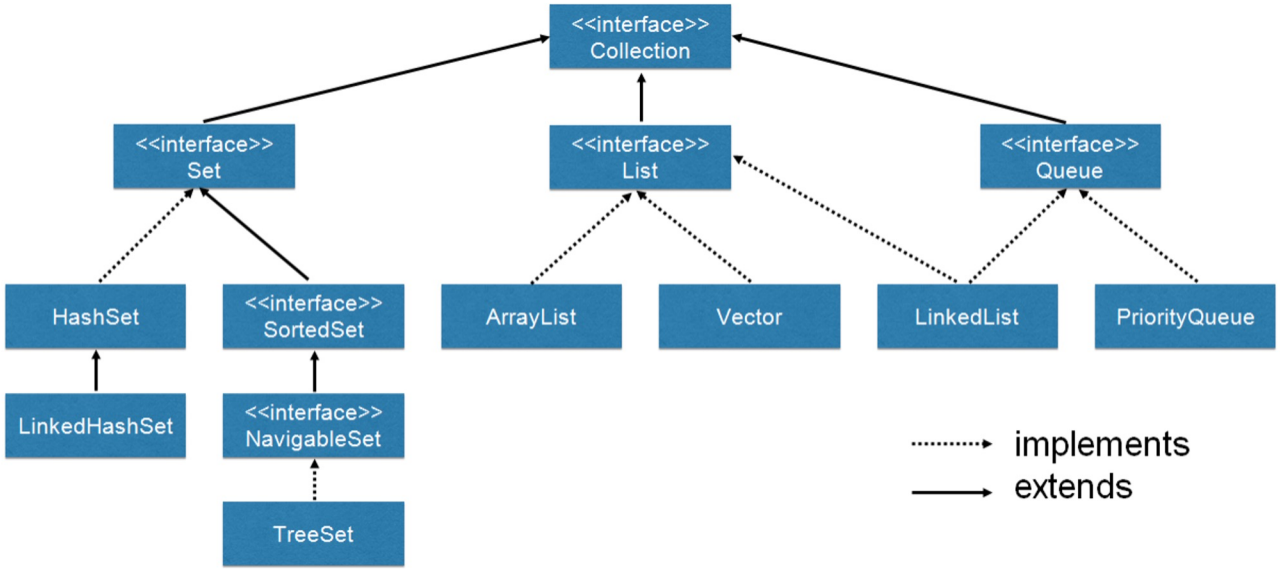


ArrayList and beyond!

- We introduced the generic **ArrayList** collection—a **dynamically resizable array**-like **data structure** that stores references to objects of a type.
- In this lecture, we continue our discussion 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:
  - **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.
  - 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.



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.

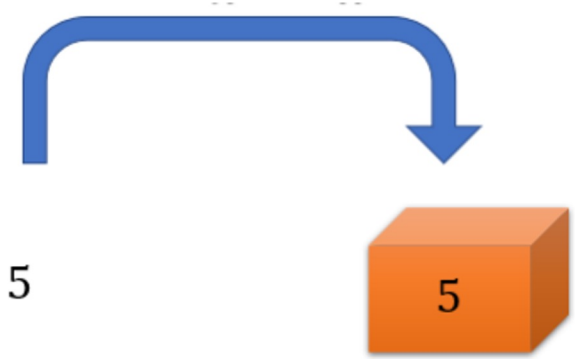
Fig. 20.1 | Some collections-framework interfaces.

Interface Collection and Class Collections

- 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.
- Class **Collections** provides **static methods** that **search**, **sort** and **perform** other operations on collections.

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

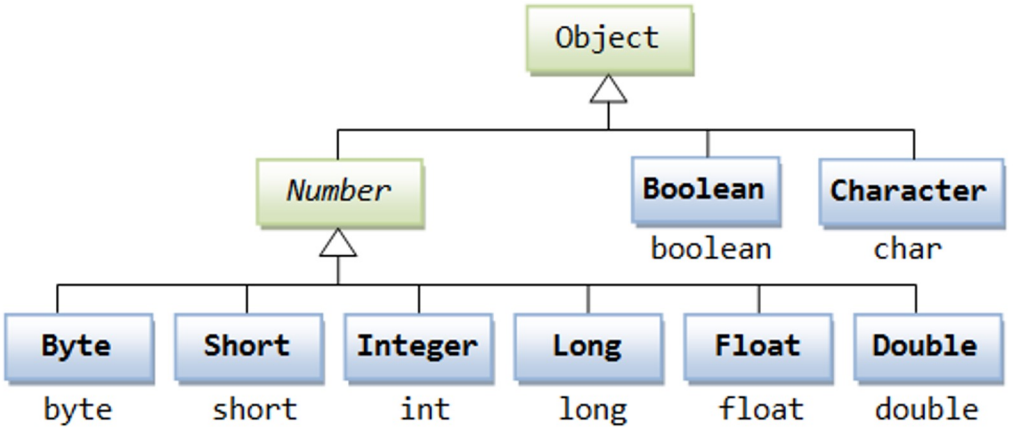
Type-Wrapper Classes for Primitive Types

- **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

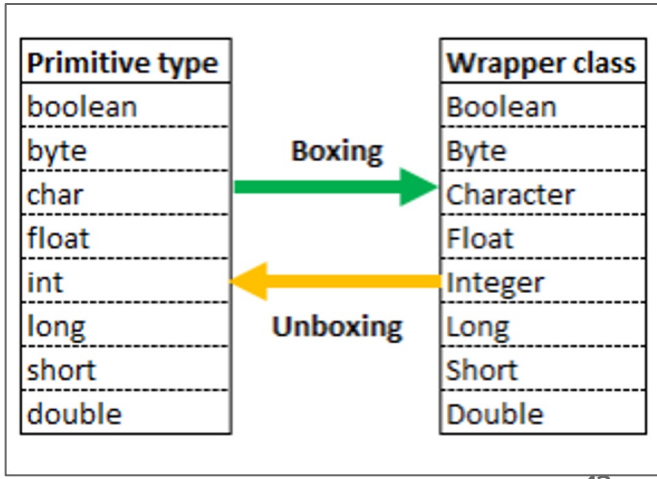
Primitive Data Type	Wrapper Class
double	*Double*
float	*Float*
long	*Long*
int	*Integer*
short	*Short*
byte	*Byte*
char	*Character*
boolean	*Boolean*



Autoboxing and Auto-Unboxing

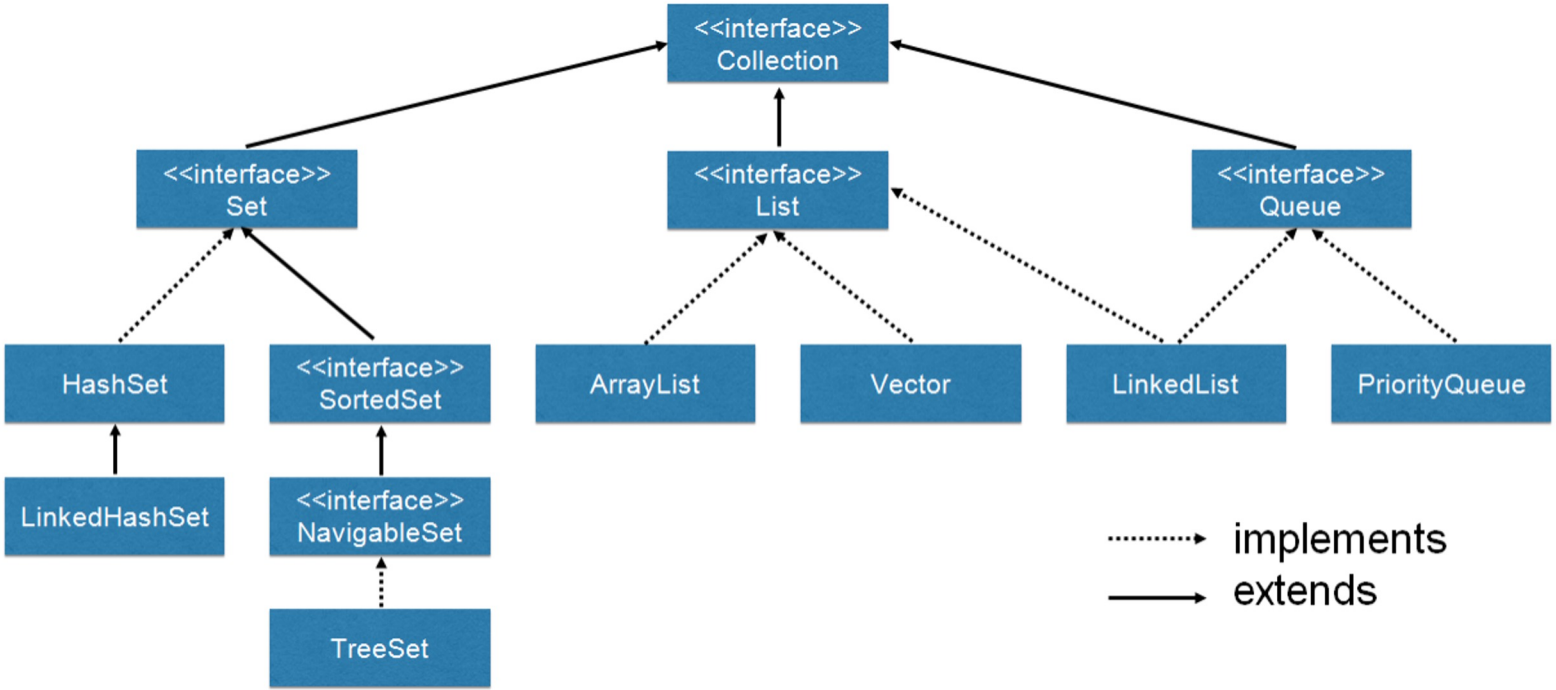
- A **boxing conversion** converts a value of a primitive type to an object of the corresponding type-wrapper class.
- An **unboxing conversion** converts an object of a type-wrapper class to a value of the corresponding primitive type.
- These **conversions** can be **performed automatically** (called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

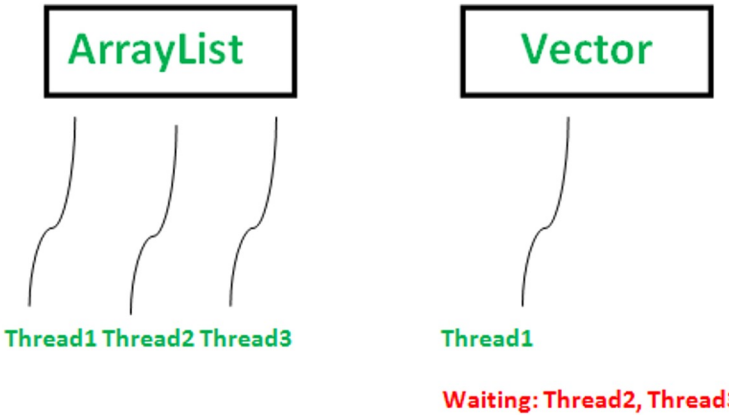


List

- A **List** is a Collection that can contain **duplicate elements**.
- **Fist index** of a List is **zero**.
- 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**.
- Autoboxing occurs when you add primitive-type values to objects of these classes, because they store only references to objects.

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.



List Methods

- 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.

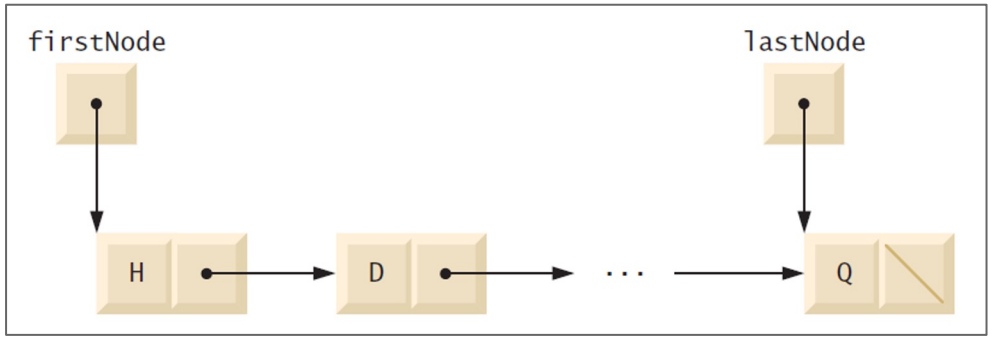
```java
// Fig. 20.2: CollectionTest.java
// Collection interface demonstrated via an ArrayList object.
import java.util.List;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

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
```

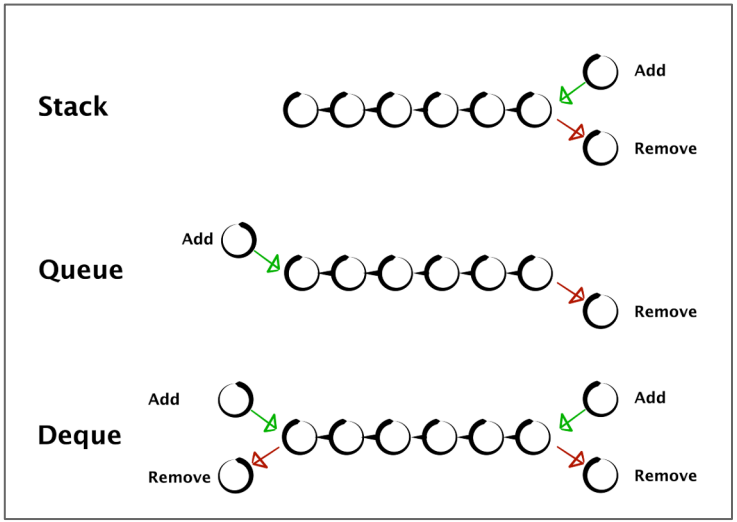
```
ArrayList:
MAGENTA RED WHITE BLUE CYAN

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).





List Methods

- List method **addAll** appends all elements of a collecton to the end of a List.
- List method **listIterator** gets A List's bidirectional iterator.
- String method **toUpperCase** gets an uppercase version of a String.
- List-Iterator method **set** replaces the current element to which the iterator refers with the specified object.
- List method **subList** 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 the elements of a List.
- List method **size** returns the number of items in the 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.

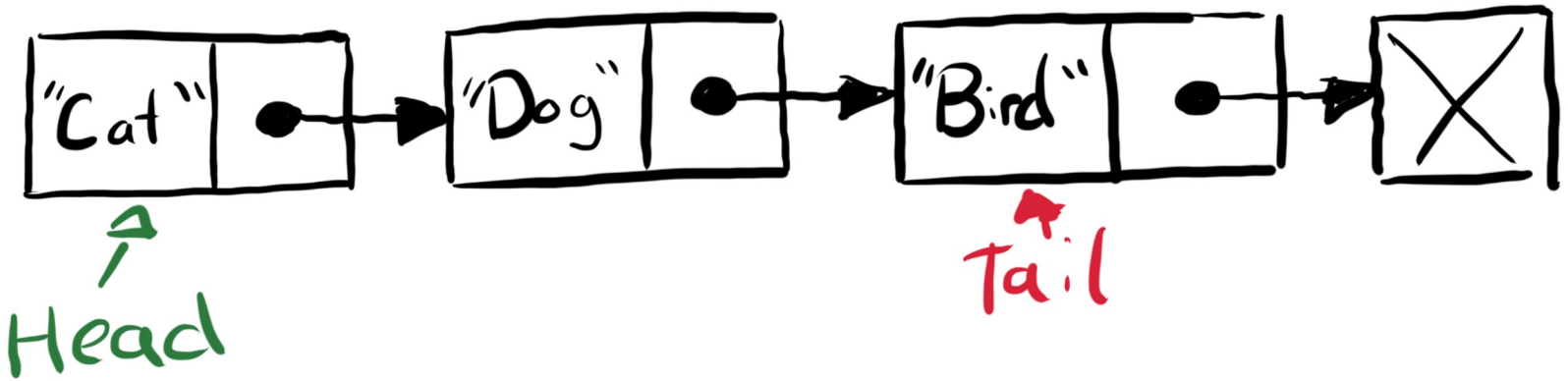
```java
// Fig. 20.3: ListTest.java
// Lists, LinkedLists and ListIterators.
import java.util.List;
import java.util.LinkedList;
import java.util.ListIterator;

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

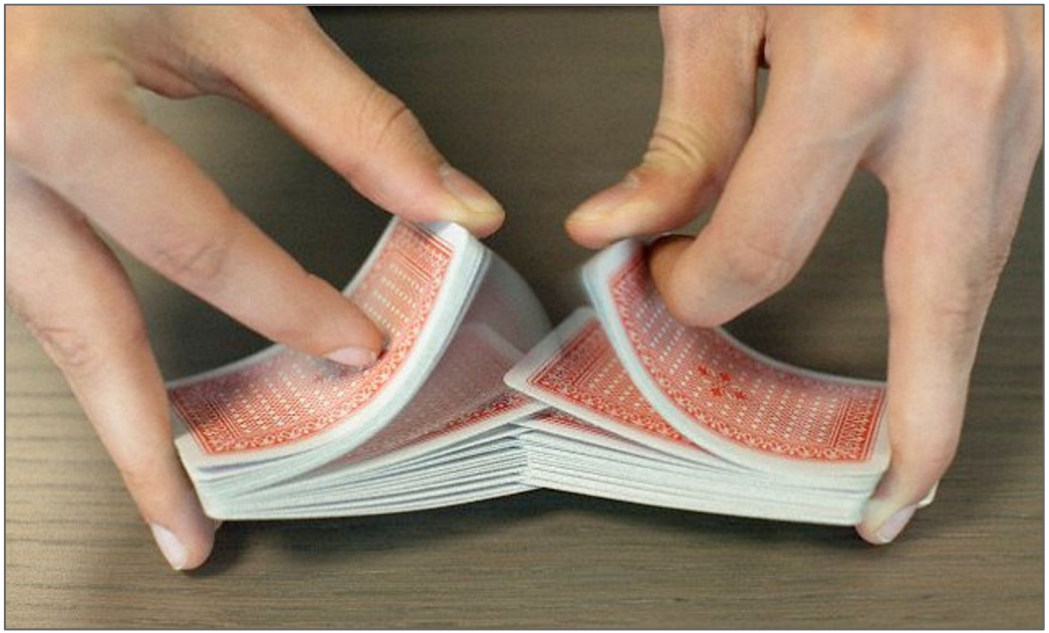
- The Comparator interface is used for sorting a Collection's elements in a different order.
- The static **Collections method reverseOrder** returns a Comparator object that orders the collection's elements in reverse order.

```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
```

Method binarySearch

- static **Collections method binarySearch** locates an object in a **List**.
  - If the object is found, its index is returned.
  - If the object is not found, **binarySearch** returns a negative value.
  - Method **binarySearch** determines this negative value by first calculating the insertion point and making its sign negative.
  - Then, **binarySearch** subtracts 1 from the insertion point to obtain the return value, which guarantees that method binarySearch returns positive numbers (>= 0) if and only if the object is found.

```java
// Fig. 20.12: BinarySearchTest.java
// Collections method binarySearch.
import java.util.List;
import java.util.Arrays;
import java.util.Collections;
import java.util.ArrayList;

public class BinarySearchTest
{
   public static void main( String[] args )
   {
      // create an ArrayList< String > from the contents of colors array
      String[] colors = { "red", "white", "blue", "black", "yellow",
         "purple", "tan", "pink" };
      List< String > list =
         new ArrayList< String >( Arrays.asList( colors ) );

      Collections.sort( list ); // sort the ArrayList
      System.out.printf( "Sorted ArrayList: %s\n", list );

      // search list for various values
      printSearchResults( list, colors[ 3 ] ); // first item
      printSearchResults( list, colors[ 0 ] ); // middle item
      printSearchResults( list, colors[ 7 ] ); // last item
      printSearchResults( list, "aqua" ); // below lowest
      printSearchResults( list, "gray" ); // does not exist
      printSearchResults( list, "teal" ); // does not exist
   } // end main

   // perform search and display result
   private static void printSearchResults(
      List< String > list, String key )
   {
      int result = 0;

      System.out.printf( "\nSearching for: %s\n", key );
      result = Collections.binarySearch( list, key );

      if ( result >= 0 )
         System.out.printf( "Found at index %d\n", result );
      else
         System.out.printf( "Not Found (%d)\n",result );
   } // end method printSearchResults
} // end class BinarySearchTest
```

```
Sorted ArrayList: [black, blue, pink, purple, red, tan, white, yellow]

Searching for: black
Found at index 0

Searching for: red
Found at index 4

Searching for: pink
Found at index 2

Searching for: aqua
Not Found (-1)

Searching for: gray
Not Found (-3)

Searching for: teal
Not Found (-7)
```

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
```

Sets

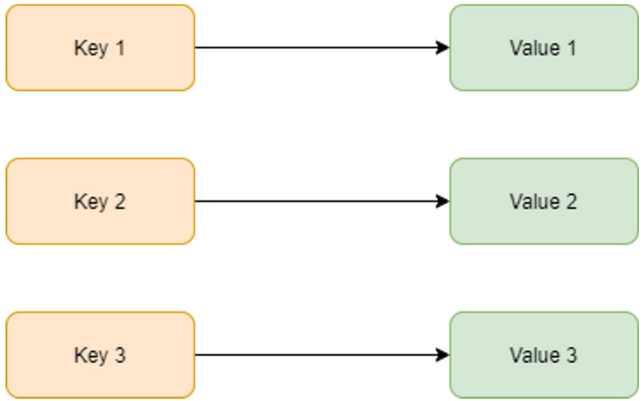
- A Set is an unordered Collection of **unique elements** (i.e., **no duplicate elements**).
- The collections framework contains several Set implementations, including **HashSet** and **TreeSet**.
- **HashSet** stores its elements in a **hash table**, and **TreeSet** stores its elements in a **tree**.

```java
// Fig. 20.16: SetTest.java
// HashSet used to remove duplicate values from array of strings.
import java.util.List;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;
import java.util.Collection;

public class SetTest
{
   public static void main( String[] args )
   {
      // create and display a List< String >
      String[] colors = { "red", "white", "blue", "green", "gray",
         "orange", "tan", "white", "cyan", "peach", "gray", "orange" };
      List< String > list = Arrays.asList( colors );
      System.out.printf( "List: %s\n", list );

      // eliminate duplicates then print the unique values
      printNonDuplicates( list );
   } // end main

   // create a Set from a Collection to eliminate duplicates
   private static void printNonDuplicates( Collection< String > values )
   {
      // create a HashSet
      Set< String > set = new HashSet< String >( values );

      System.out.print( "\nNonduplicates are: " );

      for ( String value : set )
         System.out.printf( "%s ", value );

      System.out.println();
   } // end method printNonDuplicates
} // end class SetTest
```

```
List: [red, white, blue, green, gray, orange, tan, white, cyan, peach, gray,
orange]

Nonduplicates are: orange green white peach gray cyan red blue tan
```

Maps

- **Maps** associate keys to values.
  - The keys in a Map must be unique, but the associated values need not be.
  - If a Map contains both unique keys and unique values, it is said to implement a **one-to-one mapping.**
  - If only the keys are unique, the Map is said to implement a **many-to-one mapping**—many keys can map to one value.
- Three of the several classes that implement interface Map are **Hashtable**, **HashMap** and **TreeMap**.
- **Hashtables** and **HashMaps** store elements in hash tables, and **TreeMaps** store elements in trees.
- **TreeMaps** keeps the key in sorted order.
- Terms:
  - *"Map"* is used by Java, C++
  - *"Dictionary"* is used by .Net, Python
  - *"Associative array"* is used by PHP



Maps

- Map method **containsKey** determines whether a key is in a map.
- Map method **put** creates a new entry in the map or replaces an existing entry's value.
  - Method put returns the key's prior associated value, or null if the key was not in the map.
- Map method **get obtain the specified key's associated value in the map.**
- HashMap method **keySet** returns a set of the keys.
- Map method **size** returns the number of key/value pairs in the Map.
- Map method **isEmpty** returns a boolean indicating whether the Map is empty.

```java
// Fig. 20.18: WordTypeCount.java
// Program counts the number of occurrences of each word in a String.
import java.util.Map;
import java.util.HashMap;
import java.util.Set;
import java.util.TreeSet;
import java.util.Scanner;

public class WordTypeCount
{
   public static void main( String[] args )
   {
      // create HashMap to store String keys and Integer values
      Map< String, Integer > myMap = new HashMap< String, Integer >();

      createMap( myMap ); // create map based on user input
      displayMap( myMap ); // display map content
   } // end main

   // create map from user input
   private static void createMap( Map< String, Integer > map )
   {
      Scanner scanner = new Scanner( System.in ); // create scar
      System.out.println( "Enter a string:" ); // prompt for use
      String input = scanner.nextLine();

      // tokenize the input
      String[] tokens = input.split( " " );

      // processing input text
      for ( String token : tokens )
      {
         String word = token.toLowerCase(); // get lowercase word

         // if the map contains the word
         if ( map.containsKey( word ) ) // is word in map
         {
            int count = map.get( word ); // get current count
            map.put( word, count + 1 ); // increment count
         } // end if
         else
            map.put( word, 1 ); // add new word with a count of 1 to map
      } // end for
   } // end method createMap

   // display map content
   private static void displayMap( Map< String, Integer > map )
   {
      Set< String > keys = map.keySet(); // get keys

      // sort keys
      TreeSet< String > sortedKeys = new TreeSet< String >( keys );

      System.out.println( "\nMap contains:\nKey\t\tValue" );

      // generate output for each key in map
      for ( String key : sortedKeys )
         System.out.printf( "%-10s%10s\n", key, map.get( key ) );

      System.out.printf(
         "\nsize: %d\nisEmpty: %b\n", map.size(), map.isEmpty() );
   } // end method displayMap
} // end class WordTypeCount
```

```
Enter a string:
this is a sample sentence with several words this is another sample
sentence with several different words

Map contains:
Key             Value
a                   1
another             1
different           1
is                  2
sample              2
sentence            2
several             2
this                2
with                2
words               2

size: 10
isEmpty: false
```

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:

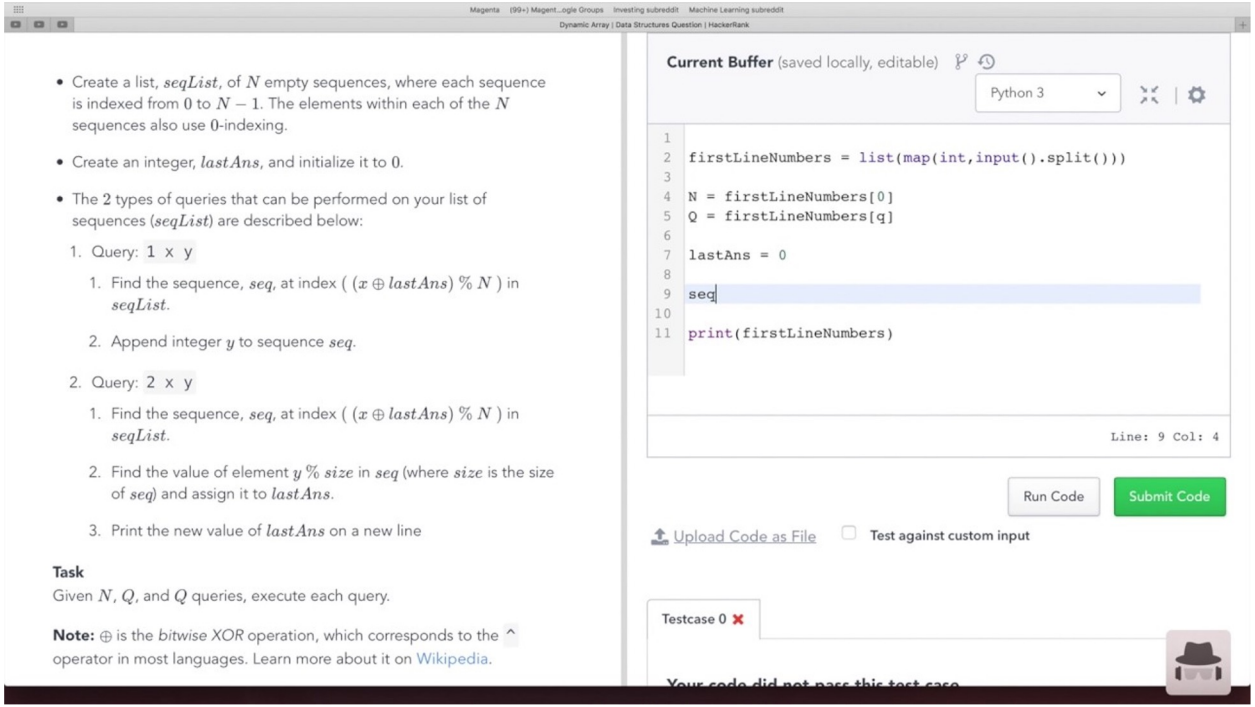
```
for (Iterator iterator = collection.iterator(); iterator.hasNext();) {
  type type = (type) iterator.next();
}
```

- 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) {

}
```

HackerRank

- HackerRank is a place where programmers from all over the world come together to solve coding problem to prepare for job interviews.
- https://www.hackerrank.com/interview/interview-preparation-kit



Get Used To HackerRank!

https://www.hackerrank.com/domains/tutorials/30-days-of-code

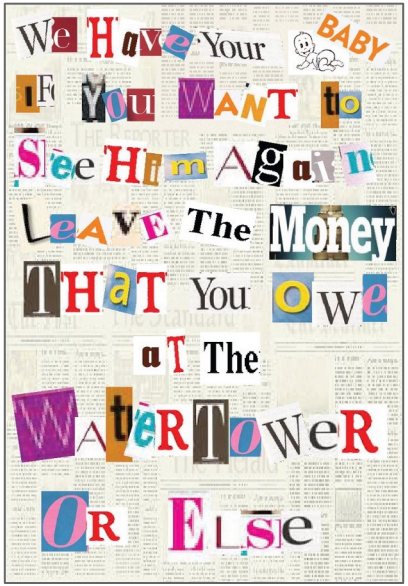
https://www.hackerrank.com/challenges/30-hello-world/problem

- Select your language to be Java 7.
- Type your code
- Press “Run Code” to test your code.
- When you are happy with your code, press “Submit Code” to let your code to be tested against different test cases. If you passed all the test cases, then you pass the problem.

Challenge Time!
Real Job Interview Question: Ransom Note

https://www.hackerrank.com/challenges/ctci-ransom-note/problem

Hint: you should use HashMap.



Recap

- **Mid-term Exam Result**
  - Score Distribution
  - Statistics
  - Solutions
- **Java Generic Collections**
  - Type-Wrapper Classes for Primitive Types
    - Autoboxing vs Auto-unboxing
  - List
    - ArrayList
    - Vector
    - LinkedList
  - Sets
    - HashSet
    - TreeSet
  - Maps
    - Hashtable
    - HashMap
    - TreeMaps
- HackerRank
  - Introduction
  - Coding Challenge

Thank you for your listening!

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

Jim Ryun