



CHƯƠNG 6 LẬP TRÌNH TỔNG QUÁT

ThS. Phạm Văn Tiệp

Nội dung

- Generics
- Collections



ĐẠI NAM
UNIVERSITY

GENERIC

Generics là gì?

- Generics là **kiểu đại diện**, nó cho phép tạo mã nguồn code không phụ thuộc vào **kiểu dữ liệu cụ thể**, chỉ khi code thực thi thì kiểu cụ thể mới xác định.

Generics class

- Lớp tổng quát (**generic class**) là lớp có thể nhận kiểu dữ liệu là một lớp bất kỳ.
- **Cú pháp**
Tên Lớp <kiểu 1, kiểu 2, kiểu 3...> {

}
- Các phương thức hay thuộc tính của lớp tổng quát có thể sử dụng các kiểu được khai báo như mọi lớp bình thường khác.

Generics class

Ví dụ

```
public class Information<T> {  
    private T value;  
    public Information(T value) {  
        this.value = value;  
    }  
    public T getValue() {  
        return value;  
    }  
}
```

```
Information<String> string = new Information<String>("hello");  
Information<Circle> circle = new Information<Circle>(new Circle());  
Information<2DShape> shape = new Information<>(new 2DShape());
```

Generics class

- Chú ý: Không sử dụng các kiểu dữ liệu nguyên thủy cho các lớp tổng quát được
- Ví dụ

`Information<int> integer = new Information<int>(2012);`

`Information<Integer> integer = new Information<Integer>(2012); // OK`

Generics method

- Phương thức tổng quát (Generic method) là các phương thức tự định nghĩa kiểu tham số của nó.
- Có thể được viết trong lớp bất kỳ (tổng quát hoặc không)
- **Cú pháp**
(chỉ định truy cập) <kiểu1, kiểu 2...> (kiểu trả về) tên phương thức (danh sách tham số) {
...
}
• Ví dụ
`public <E> static void print(E[] a) { ... }`

Ví dụ

```
public class ArrayTool {  
    // Phương thức in các phần tử trong mảng String  
    public static void print(String[] a) {  
        for (String e : a)  
            System.out.print(e + " ");  
        System.out.println();  
    }  
    // Phương thức in các phần tử trong mảng với kiểu dữ liệu bất kỳ  
    public static <E> void print(E[] a) {  
        for (E e : a)  
            System.out.print(e + " ");  
        System.out.println();  
    }  
}
```

Ví dụ

...

```
Point[] p = new Point[3];
```

```
String[] str = new String[5];
```

```
int[] intnum = new int[2];
```

```
ArrayTool.print(p);
```

```
ArrayTool.print(str);
```

```
// Không dùng được với kiểu dữ liệu nguyên thủy
```

```
ArrayTool.print(intnum);
```

Giới hạn kiểu dữ liệu tổng quát

- Có thể giới hạn các kiểu dữ liệu tổng quát sử dụng phải là dẫn xuất của một hoặc nhiều lớp

- Giới hạn 1 lớp

`<type_param extends bound>`

- Giới hạn nhiều lớp

`<type_param extends bound_1 & bound_2 & ..>`

Ví dụ

```
public class Information<T extends 2DShape> {  
    private T value;  
    public Information(T value) {  
        this.value = value;  
    }  
    public T getValue() {  
        return value;  
    }  
}
```

...

```
Information<Point> pointInfo = new Information<Point>(new Point()); //OK
```

```
Information<String> stringInfo = new Information<String>(); // error
```



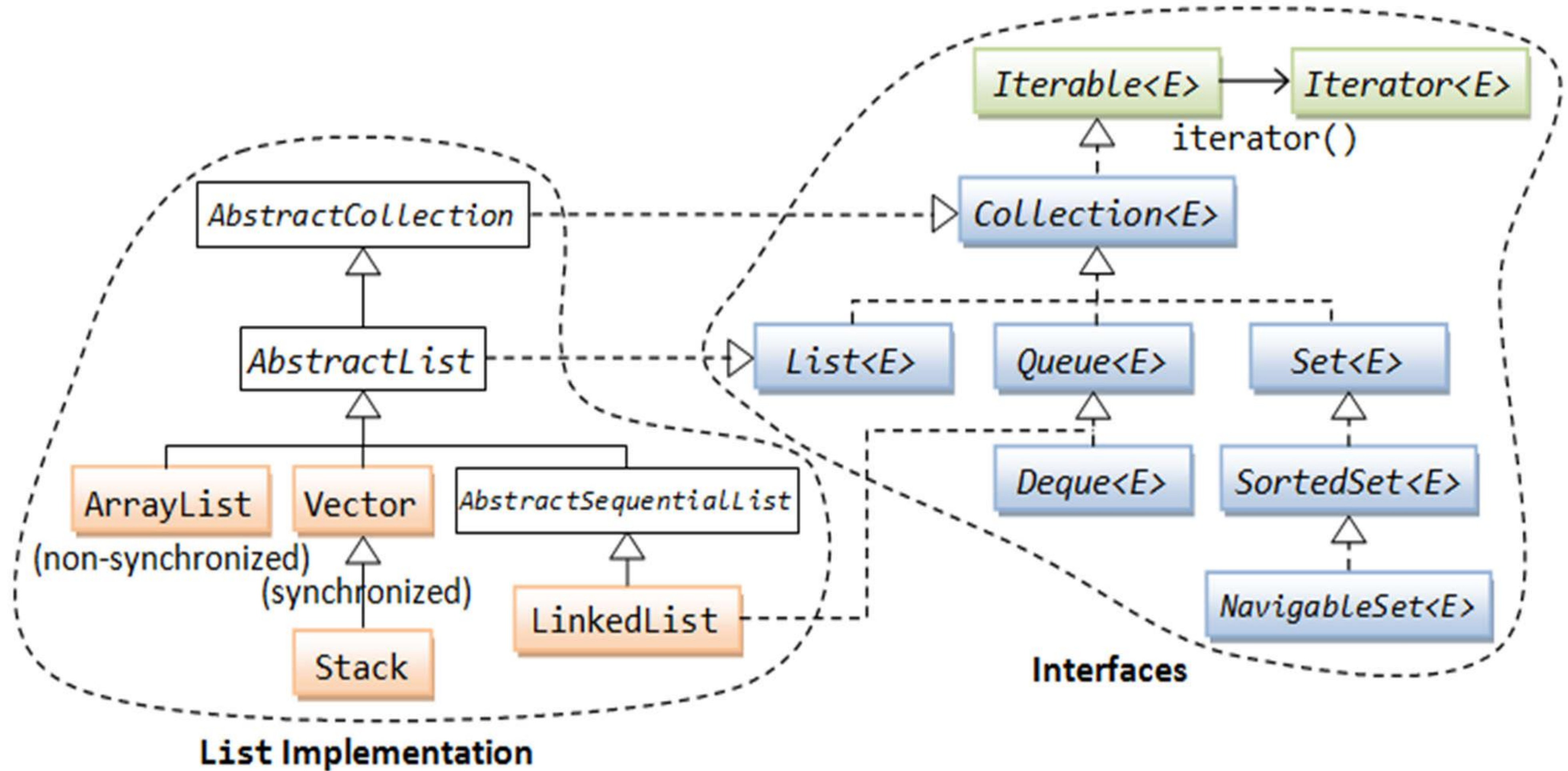
ĐẠI NAM
UNIVERSITY

COLLECTION

Collection là gì?

- **Collection** là một đối tượng mà nó nhóm các đối tượng khác thành phần tử và cung cấp các phương thức cơ bản để thêm, xóa, lấy, duyệt các phần tử...
 - Phần tử của Collection không được phép là **kiểu nguyên thủy**
- **Collections Framework** thống nhất cách thức sử dụng các collection, gồm 3 thành phần chính:
 - Interface
 - Lớp triển khai
 - Thuật toán
- Sử dụng đối tượng **Iterator** để duyệt qua tất cả các phần tử của collection
- Được xây dựng dựa trên kỹ thuật lập trình tổng quát
- Ưu điểm: tiện dụng, hiệu năng cao

Các interfaces và Lớp triển khai



Ví dụ

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

public class ArrayListDemo {
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        list.add("alpha"); // add an element
        list.add("beta");
        list.add("charlie");
        System.out.println(list); // [alpha, beta, charlie]
```

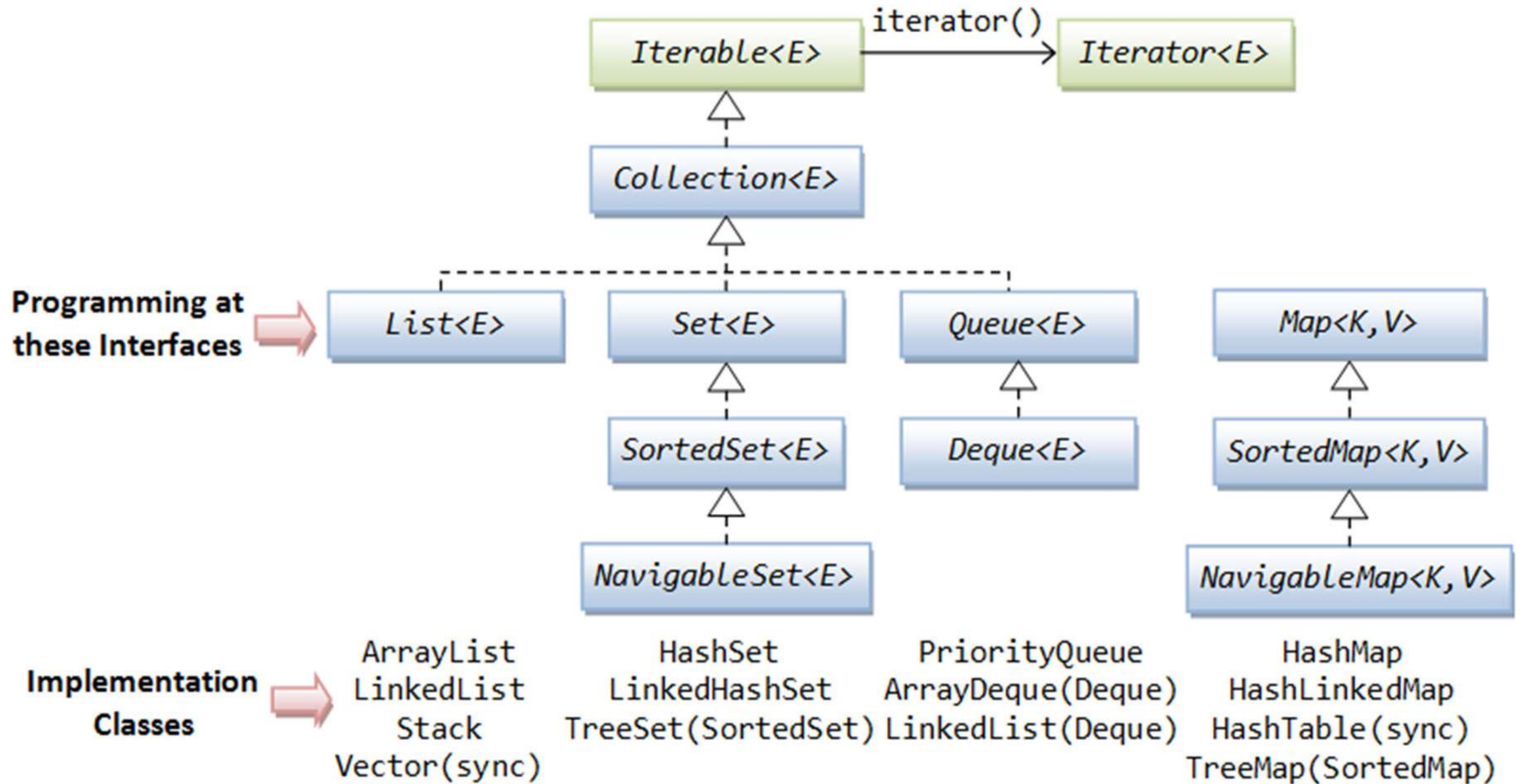
Ví dụ (tiếp)

```
// Iterator of Strings
Iterator<String> iter = list.iterator();
while (iter.hasNext()) {
    String str = iter.next();
    System.out.println(str);
}
for (String str : list) {
    System.out.println(str);
}
}
```




INTERFACES COLLECTION

Các interfaces



Các interfaces (tiếp)

▪ **Iterable<E>**

//Trả lại một đối tượng Iterator để duyệt qua các phần tử
Iterator<E> iterator()

▪ **Iterator<E>**

// Trả về true nếu còn phần tử chưa được duyệt
boolean hasNext()

// Trả về phần tử tiếp theo trong collection
E next()

// Xóa phần tử đang duyệt
void remove()

Các interfaces (tiếp)

▪ **Collection<E>**

// Trả về số phần tử của collection

int size()

// Xóa mọi phần tử trong collection

void clear()

// Trả về true nếu không có phần tử nào trong collection

boolean isEmpty()

// Thêm 1 phần tử vào collection, trả về true nếu thành công

boolean add(E element)

// Xóa 1 phần tử, trả về true nếu thành công

boolean remove(Object element)

// Trả về true nếu trong collection chứa phần tử element

boolean contains(Object element)

// Chuyển collection thành mảng

Object[] toArray()

Các interfaces (tiếp)

▪ **List<E>**, **Set<E>** và **Queue<E>**

Là các interface kế thừa từ interface Collection

- **List<E>**: lưu trữ các phần tử một cách tuần tự, các phần tử có thể giống nhau → mảng có kích thước thay đổi
 - Các lớp triển khai: [ArrayList](#), [LinkedList](#), [Vector](#), [Stack](#)
- **Set<E>**: mô hình hóa tập hợp trong Toán học, không chấp nhận có các phần tử giống nhau
 - Các lớp triển khai: [Set](#), [HashSet](#), [LinkedHashSet](#)
 - Các interface kế thừa: [SortedSet<E>](#) có triển khai là [TreeSet](#)
- **Queue<E>**: Hàng đợi FIFO
 - interface con: [Deque<E>](#)
 - Triển khai từ interface con: [PriorityQueue](#), [ArrayDeque](#) và [LinkedList](#).

Các interfaces (tiếp)

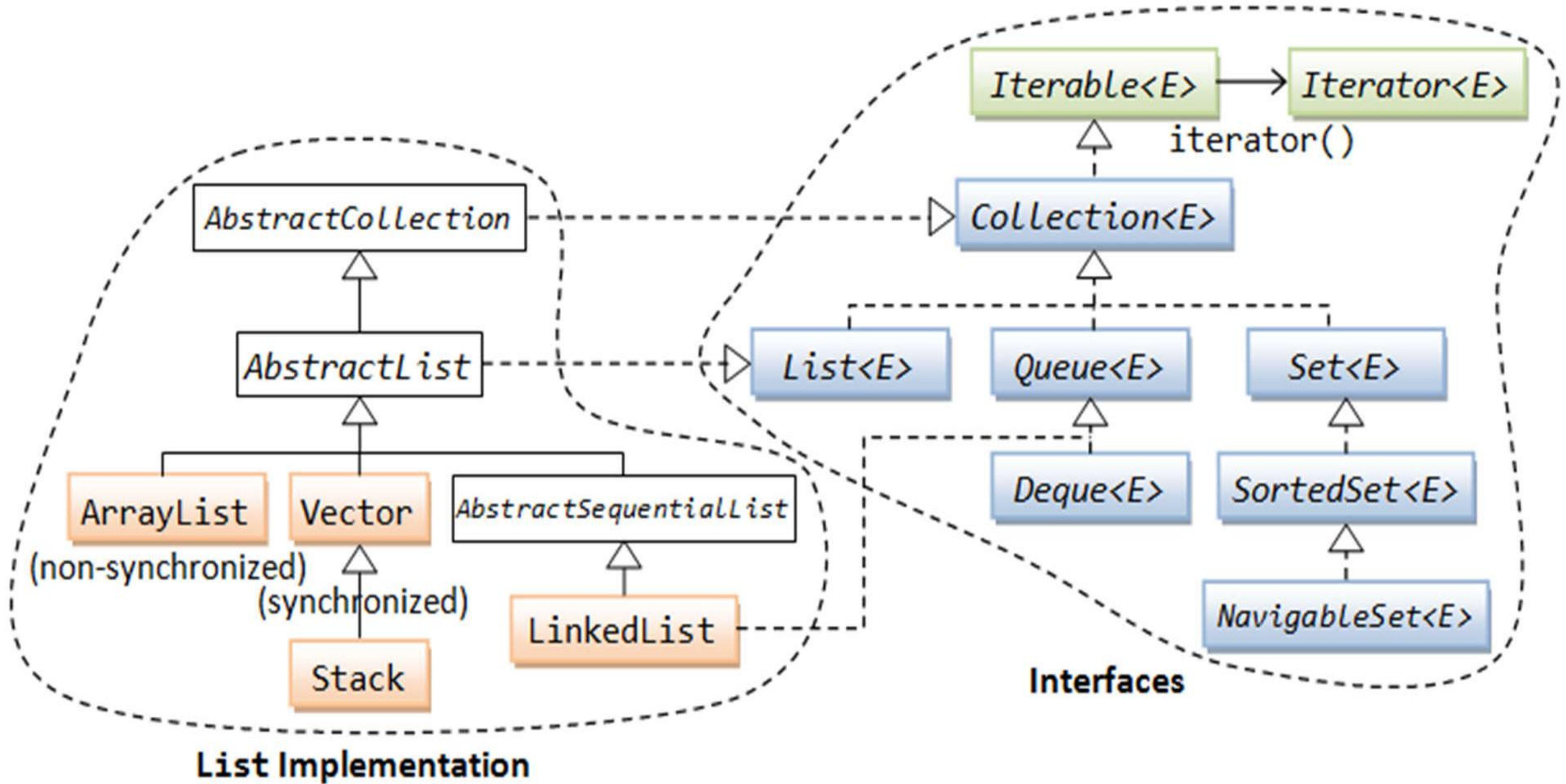
■ **Map<K,V>**

- Tổ chức các phần tử thành cặp $\langle K, V \rangle$
 - K: khóa. Không chấp nhận khóa có giá trị trùng nhau
 - V: giá trị
- Các triển khai: **HashMap, Hashtable, LinkedHashMap**
- Interface con: **SortedMap<K, V>**
 - Triển khai: **TreeMap**



LIST<E> VÀ CÁC LỚP TRIỂN KHAI

Cây kế thừa



Interface LIST<E>

//Một số phương thức truy cập danh sách qua chỉ số

void add(int index, E element) // thêm

E set(int index, E element) // thay thế

E get(int index) // lấy phần tử

E remove(int index) // xóa phần tử

//vị trí của phần tử đầu tiên giống với obj

int indexOf(Object obj)

//vị trí của phần tử cuối cùng giống obj

int lastIndexOf(Object obj)

//Lấy ra một danh sách con

List<E> subList(int fromIndex, int toIndex)

Interface LIST<E>

//Một số phương thức kế thừa từ Collection

int **size**()

boolean **isEmpty**()

boolean **add**(E element)

boolean **remove**(Object obj)

boolean **contains**(Object obj)

void **clear**()

Các lớp triển khai từ `LIST<E>`

- **ArrayList**: lớp triển khai tiện dụng nhất, được sử dụng thường xuyên để thay thế cho mảng
 - Các phương thức của ArrayList là không đồng bộ (non- synchronized)
- Khi có nhiều luồng truy cập tới ArrayList cần đồng bộ hóa các luồng bằng **Collections.synchronizedList(List<E> list)**
- **Vector**: được phát triển từ Java 1.0, hiện ít dùng
 - Các phương thức của Vector là đồng bộ (synchronized)
- hỗ trợ truy cập đa luồng
 - Hiệu năng chương trình khi sử dụng ArrayList tốt hơn
- **Stack**: kế thừa từ Vector
- **LinkedList**: danh sách liên kết

ArrayList – Ví dụ

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Iterator;
/**
 * The ArrayListExamples class illustrates the usage of the
 * ArrayList in the Collections Framework
 */
public class ArrayListExamples {
    public static void main(String args[]) {
        // Creating an empty array list
        ArrayList<String> list = new ArrayList<String>();
        // Adding items to arrayList
        list.add("Item1");
        list.add("Item3");
        list.add(1, "Item2");    // Add Item3 to the second position
                                //of array list. The other elements
                                //are pushed backward.
        list.add("Item4");
```

ArrayList – Ví dụ

// Display the contents of the array list

```
System.out.println("The arraylist contains the following elements: "+ list);
```

// Checking index of an item

```
int pos = list.indexOf("Item2");
```

```
System.out.println("The index of Item2 is: " + pos);
```

// Checking if array list is empty

```
boolean check = list.isEmpty();
```

```
System.out.println("Checking if the arraylist is empty: " + check);
```

// Getting the size of the list

```
int size = list.size();
```

```
System.out.println("The size of the list is: " + size);
```

ArrayList – Ví dụ

```
// Checking if an element is included to the list
boolean inList = list.contains("Item5");
System.out.println("Checking if the arraylist contains the object Item5: " +
inList);
// Getting the element in a specific position
String item = list.get(0);
System.out.println("The item is the index 0 is: " + item);
// Retrieve elements from the arraylist
// 1st way: loop using index and size list
System.out.println("Retrieving items with loop using index and size list");
for (int i = 0; i < list.size(); i++) {
    System.out.println("Index: " + i + " - Item: " + list.get(i));
}
```

ArrayList – Ví dụ

// 2nd way: using foreach loop

```
System.out.println("Retrieving items using foreach loop");  
for (String str : list) {  
    System.out.println("Item is: " + str);  
}
```

// 3rd way:using iterator

// hasNext(): returns true if there are more elements

// next(): returns the next element

```
System.out.println("Retrieving items using iterator");  
for (Iterator<String> i = list.iterator(); i.hasNext();) {  
    System.out.println("Item is: " + i.next());  
}
```

ArrayList – Ví dụ

```
// Replacing an element
list.set(1, "NewItem");
System.out.println("The arraylist after the
replacement is: " + list);
// Removing items
// removing the item in index 0
list.remove(0);
// removing the first occurrence of item "Item3"
list.remove("Item3");
System.out.println("The final contents of the  arraylist are: " + list);
}
```


Chuyển đổi LIST thành mảng

Object[] toArray() // Sử dụng mảng Object[]

<T> T[] toArray(T[] a) // Sử dụng kiểu tổng quát

- Ví dụ

```
List<String> list = new ArrayList<String>();
```

```
list.add("alpha");
```

```
list.add("beta");
```

```
list.add("charlie");
```

```
// Sử dụng mảng Object[]
```

```
Object[] strArray1 = list.toArray();
```

```
System.out.println(Arrays.toString(strArray1));
```

```
// Sử dụng kiểu tổng quát. Cần truyền tham số là kiểu dữ liệu
```

```
String[] strArray2 = list.toArray(new String[0]);
```

```
strArray2[0] = "delta"; // modify the returned array
```

```
System.out.println(Arrays.toString(strArray2));
```

```
System.out.println(list);
```

Chuyển đổi mảng thành LIST

```
public static <T> List<T> asList(T[] a)
```

- Ví dụ

```
String[] strs = {"alpha", "beta", "charlie"};
```

```
System.out.println(Arrays.toString(strs)); // [alpha, beta, charlie]
```

```
List<String> list = Arrays.asList(strs);
```

```
System.out.println(list); // [alpha, beta, charlie]
```

```
// Changes in array or list write thru
```

```
strs[0] += "88";
```

```
list.set(2, list.get(2) + "99");
```

```
System.out.println(Arrays.toString(strs)); // [alpha88, beta, charlie99]
```

```
System.out.println(list); // [alpha88, beta, charlie99]
```

```
// Initialize a list using an array
```

```
List<Integer> listInt = Arrays.asList(22, 44, 11, 33);
```

```
System.out.println(listInt); // [22, 44, 11, 33]
```



SẮP XẾP, SO SÁNH VÀ TÌM KIẾM

Sắp xếp trên Collection

- Một số lớp trong Collections Framework đã được sắp xếp sẵn, như **SortedSet**, **SortedMap**
- Các lớp khác cung cấp sẵn các phương thức sắp xếp, nhưng cần định nghĩa cách thức so sánh các phần tử
 - **Collections.sort()**
 - **Arrays.sort()**
- Định nghĩa cách thức so sánh các phần tử: 2 cách
 - Lớp của các phần tử phải triển khai từ interface **java.lang.Comparable<E>**
 - Viết bộ so sánh triển khai từ interface **java.util.Comparator<E>**

So sánh với Comparable<E>

- Lớp triển khai định nghĩa phương thức **int compareTo(E obj)** trả về:
 - 0 nếu bằng đối tượng so sánh obj
 - Nhỏ hơn 0 nếu nhỏ hơn đối tượng so sánh
 - Lớn hơn 0 nếu lớn hơn đối tượng so sánh
- Khi định nghĩa **compareTo()** cần thống nhất với kết quả trả về của **equals()**:
 - Nếu **compareTo()** trả về 0, **equals()** nên trả về true
- Các lớp bao và lớp String đều là các lớp triển khai từ **Comparable**

Comparable – Ví dụ

```
public class Student implements Comparable<Student>{  
    private String id;  
    private String name;  
    private int level;  
  
    @Override  
    public int compareTo(Student o) {  
        return this.id.compareToIgnoreCase(o.getID());  
    }  
    //Declare other methods  
}
```

Comparable – Ví dụ

```
public class StudentSortDemo{  
    Student[] arr = new Student[50];  
    List<Student> list = new ArrayList<Student>();  
    //enter data for arr and list...  
    //sort arr  
    Arrays.sort(arr);  
    //sort list  
    Collections.sort(list);  
    //Display results...  
}
```

Làm thế nào để sắp xếp theo nhiều tiêu chí?

So sánh với Comparable<E>

- Định nghĩa bộ so sánh triển khai từ Comparator<E>, định nghĩa phương thức `int compare(E o1, E o2)` trả về:
 - 0 nếu $o1 == o2$
 - Nhỏ hơn 0 nếu $o1 < o2$
 - Lớn hơn 0 nếu $o1 > o2$
- Không bắt buộc lớp của các phần tử phải kế thừa từ giao diện Comparable
 - Tiện dụng hơn
 - Mềm dẻo hơn: có thể viết nhiều bộ so sánh theo các tiêu chí khác nhau
- Cung cấp phương thức `thenComparing()` để so sánh đồng thời theo các tiêu chí khác nhau

Comparator – Ví dụ

```
public class Student{  
    private String id;  
    private String name;  
    private int level;  
    //Declare methods  
}
```


Comparator – Ví dụ

```
public class StudentSortDemo{
    Student[] arr = new Student[50];
    List<Student> list = new ArrayList<Student>();
    public static class StudentComparator implements Comparator<Student>{
        @Override
        public int compare(Student s1, Student s2){
            return s1.getName().compareToIgnoreCase(s2.getName());
        }
    }
    Comparator<Student> compStudent = new StudentComparator();
    //sort arr
    Arrays.sort(arr, compStudent);
    //sort list
    Collections.sort(list, compStudent);
    //Display results...
}
```

So sánh đồng thời nhiều bộ tiêu chí

```
public static class StudentComparatorByName implements Comparator<Student>{
    @Override
    public int compare(Student s1, Student s2){
        return s1.getName().compareToIgnoreCase(s2.getName());
    }
}

public static class StudentComparatorByLevel implements Comparator<Student>{
    @Override
    public int compare(Student s1, Student s2){
        return s1.getLevel()- s2.getLevel();
    }
}

Comparator<Student> c = (new StudentComparatorByLevel()).thenComparing(
    new StudentComparatorByName());
```

So sánh đồng thời nhiều bộ tiêu chí

Từ Java 8 cho phép tạo các bộ so sánh một cách đơn giản hơn

```
public static class StudentComparators {  
    public static final Comparator<Student> NAME =  
        (Student s1, Student s2) ->  
            s1.getName().compareToIgnoreCase(s2.getName());  
    public static final Comparator<Student> LEVEL =  
        (Student s1, Student s2) ->  
            Integer.compare(s1.getLevel(), s2.getLevel());  
    public static final Comparator<Student> NAME_THEN_LEVEL =  
        (Student s1, Student s2) ->  
            NAME.thenComparing(LEVEL).compare(s1, s2);  
}  
Arrays.sort(arr, StudentComparators.NAME_THEN_LEVEL);  
Collections.sort(list, StudentComparators.NAME_THEN_LEVEL);
```

■ Tìm kiếm tuần tự

- Mảng: duyệt và so sánh
- Collection: `boolean contains(Object o)`
- List:
 - `int indexOf(Object o)`
 - `int lastIndexOf(Object o)`

■ Tìm kiếm nhị phân: chỉ thực hiện khi mảng, collection đã được sắp xếp.

Sử dụng các phương thức tĩnh của lớp `Arrays` và `Collections`

- Trả về chỉ số của phần tử nếu tìm thấy
- Trả về -1 nếu không tìm thấy

Tìm kiếm nhị phân – Ví dụ

- Lớp của các phần tử triển khai từ Comparable<E>
 - Arrays.binarySearch(Object[] arr, Object key)
 - Arrays.binarySearch(Object[], int from, int to, Object key)
 - Collections.binarySearch(List<E>, E key)
- Sử dụng bộ so sánh triển khai từ Comparator<E>
 - Arrays.binarySearch(E[], E key, Comparator<E> c)
 - Arrays.binarySearch(E[], int from, int to, E key, Comparator<E> c)
 - Collections.binarySearch(List<E>, E key, Comparator<E> c)

Tìm kiếm nhị phân – Ví dụ

```
public class Student implements Comparable<Student> {  
    private String id;  
    private String name;  
    private int level;  
    @Override  
    public int compareTo(Student o) {  
        return this.id.compareToIgnoreCase(o.id);  
    }  
    //Declare other methods  
}
```

Tìm kiếm nhị phân – Ví dụ

```
public class StudentSortDemo{  
    Student[] arr = new Student[50];  
    List<Student> list = new ArrayList<Student>();  
  
    Arrays.sort(arr);  
    Student student = new Student("20123456");  
    System.out.println("Found this student at" +  
        Arrays.binarySearch(arr, student));  
  
    Collections.sort(list);  
    System.out.println("Found this student at" +  
        Collections.binarySearch(list, student));  
}
```

Tìm kiếm nhị phân – Ví dụ khác

```
public class Student{  
    private String id;  
    private String name;  
    private int level;  
    //Declare methods  
}
```


Tìm kiếm nhị phân – Ví dụ khác

```
public class StudentSortDemo{
    Student[] arr = new Student[50];
    List<Student> list = new ArrayList<Student>();
    public static class StudentComparator implements Comparator<Student>{
        @Override
        public int compare(Student s1, Student s2){
            return s1.id.compareToIgnoreCase(s2.id);
        }
    }
    Comparator<Student> compStudent = new StudentComparator();
    Arrays.sort(arr, compStudent);
    Student student = new Student("20123456");
    System.out.println("Found this student at" +
        Arrays.binarySearch(arr, student, compStudent));
    Collections.sort(list, compStudent);
    System.out.println("Found this student at" +
        Collections.binarySearch(list, student, compStudent));
}
```