ĐẠI NAM
UNIVERSITY

LẬP TRÌNH TỔNG QUÁT

Thời gian: 4 tiết

Nội dung

- ✓ Lập trình tổng quát
- ✓ Java template
- ✓ Java collection framework
- ✓ Wildcard

Mục tiêu bài học

- Giới thiệu về lập trình tổng quát và cách thực hiện trong các ngôn ngữ lập trình
- Giới thiệu về collection framework với các cấu trúc tổng quát: List, HashMap, Tree, Set, Vector,…
- Định nghĩa và sử dụng Template và ký tự đại diện (wildcard)
- Ví dụ và bài tập về các vấn đề trên với ngôn ngữ lập trình Java

Giới thiệu về lập trình tổng quát (Generic programming)

Giới thiệu về lập trình tổng quát

- Tổng quát hóa chương trình để có thể hoạt động với các kiểu dữ liệu khác nhau, kể cả kiểu dữ liệu trong tương lai
  - Thuật toán đã xác định

Tổng quát hoá chương trình

- Ví dụ:

Phương thức **sort()**

- Số nguyên int
- Xâu ký tự String
- Đối tượng số phức Complex object…

Thuật toán giống nhau, chỉ khác về kiểu dữ liệu

Lớp lưu trữ kiểu ngăn xếp (Stack)

- Lớp IntegerStack → đối tượng Integer
- Lớp StringStack → đối tượng String
- Lớp AnimalStack → đối tượng Animal

Các lớp có cấu trúc tương tự, khác nhau về kiểu đối tượng xử lý

Giới thiệu về lập trình tổng quát

- Lập trình **Generic** có nghĩa là lập trình mà có thể tái sử dụng cho nhiều kiểu dữ liệu
  - Cho phép trừu tượng hóa kiểu dữ liệu
- Giải pháp trong các ngôn ngữ lập trình:
  - C: dùng con trỏ không định kiểu (con trỏ void)
  - C++: dùng template
  - Java 1.5 trở về trước: lợi dụng upcasting và kiểu tổng quát object
  - Java 1.5: đưa ra khái niệm về template

Giới thiệu về lập trình tổng quát

- Ví dụ C: hàm memcpy() trong thư viện string.h

```
void* memcpy(void* region1, const void* region2, size_t n);
```

  - Hàm memcpy() bên trên được khai báo tổng quát bằng cách sử dụng các con trỏ void*
  - Điều này giúp cho hàm có thể sử dụng với nhiều kiểu dữ liệu khác nhau
    - Dữ liệu được truyền vào một cách tổng quát thông qua địa chỉ và kích thước kiểu dữ liệu
    - Hay nói cách khác, để sao chép dữ liệu, ta chỉ cần địa chỉ và kích cỡ của chúng

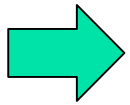
Giới thiệu về lập trình tổng quát

- Ví dụ: Lập trình Generic từ trước Java 1.5

```
public class ArrayList {
        public Object get(int i) { . . . }
        public void add(Object o) { . . . }
        . . .
        private Object[] elementData;
}
```

- Lớp Object là lớp cha tổng quát nhất → có thể chấp nhận các đối tượng thuộc lớp con của nó

```
List myList = new ArrayList();
myList.add("Fred");
myList.add(new Dog());
myList.add(new Integer(42));
```



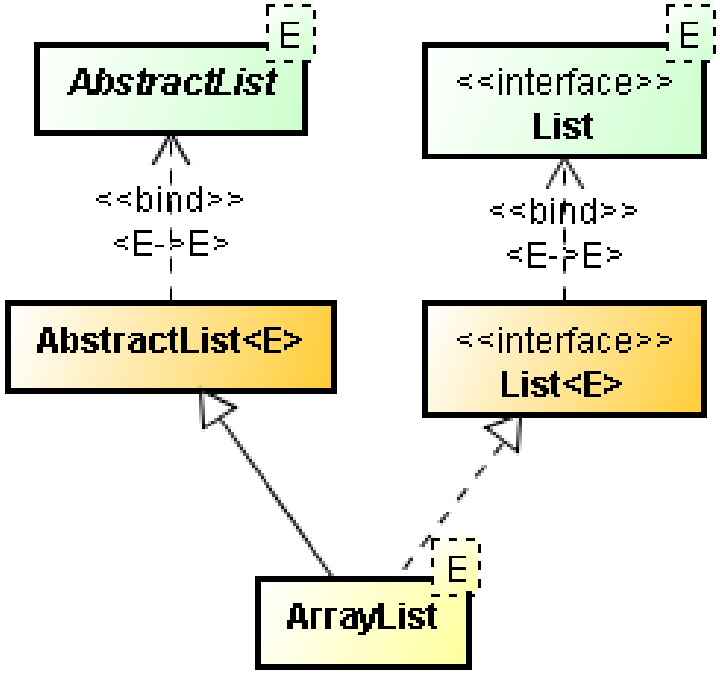
Các đối tượng trong một danh sách khác hẳn nhau

- Hạn chế: Phải ép kiểu ➔ có thể ép sai kiểu (run-time error)

```
String name = (String) myList.get(1); //Dog!!!
```

Giới thiệu về lập trình tổng quát

- Ví dụ: Lập trình Generic từ Java 1.5
  - Java 1.5 Template



Danh sách chỉ chấp nhận các đối tượng có kiểu là Integer

```
List<Integer> myList =
        new LinkedList<Integer>();
myList.add(new Integer(0));
Integer x = myList.iterator().next(); //Không cần ép kiểu

myList.add(new String("Hello")); //Compile Error
```

Định nghĩa và sử dụng Template

Lớp tổng quát

- 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

Lớp tổng quát

Tên kiểu, sẽ được thay thế bằng một kiểu cụ thể khi sử dụng

- Ví dụ:

```
public class Information<T> {
  private T value;
  public Information(T value) {
    this.value = value;
  }
  public T getValue() {
    return value;
  } }
Information<String> mystring =
      new Information<String>("hello");
Information<Circle> circle =
      new Information <Circle>(new Circle());
Information<2DShape> shape =
      new Information<>(new 2DShape());
```

Lớp tổng quát

- Quy ước đặt tên kiểu

Tên kiểu	Mục đích
E	Các thành phần trong một collection
K	Kiểu khóa trong Map
V	Kiểu giá trị trong Map
T	Các kiểu thông thường
S, U	Các kiểu thông thường khác

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

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

Phương thức tổng quát

- 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 static <E> void print(E[] a) { … }
```

Phương thức tổng quát

- 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();
  }
}
```

Phương thức tổng quát

- Ví dụ:

```
...
String[] str = new String[5];
Point[] p = new Point[3];
int[] intnum = new int[2];

ArrayTool.print(str);
ArrayTool.print(p);

// 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 & ..>
```

Giới hạn kiểu dữ liệu tổng quát

Chấp nhận các kiểu là lớp con của 2DShape

- 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
```

Lập trình tổng quát trong Java

(Collection framework)

Lập trình tổng quát trong Java

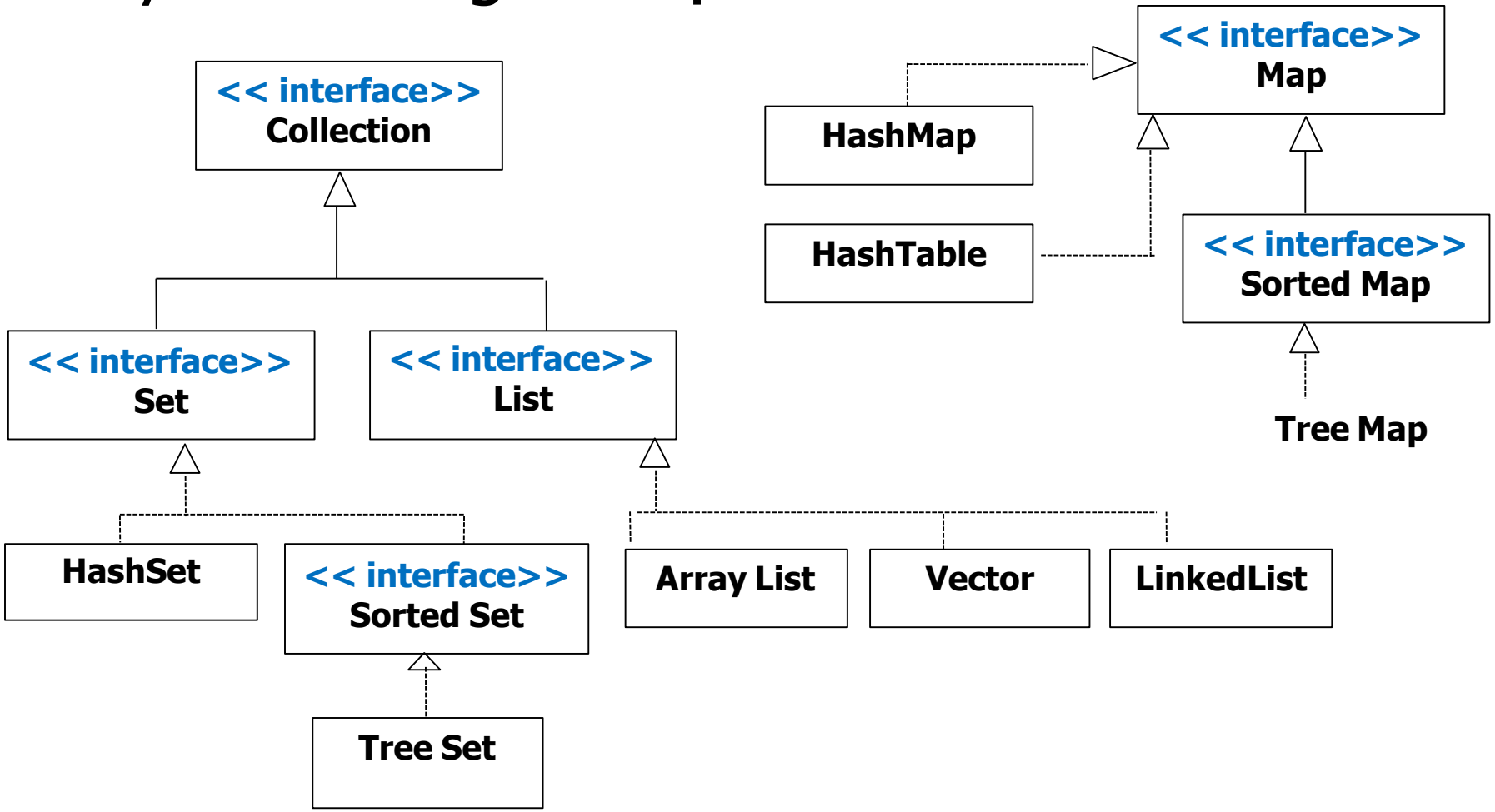
- Collection – tập hợp: Nhóm các đối tượng lại thành một đơn vị duy nhất
- Java Collections Framework:
  - Biểu diễn các tập hợp
  - Cung cấp giao diện tiêu chuẩn cho hầu hết các tập hợp cơ bản
  - Xây dựng dựa trên
    - Interface: thể hiện tính chất của các kiểu tập hợp khác nhau như List, Set, Map
    - Class: các lớp cụ thể thực thi các giao diện
    - Thuật toán: cài đặt một số thao tác đơn giản, là các phương thức tĩnh để xử lý trên collection như tìm kiếm, sắp xếp…

Lập trình tổng quát trong Java

- Java Collections Framework:
  - List: Tập các đối tượng tuần tự, kế tiếp nhau, có thể lặp lại
  - Set: Tập các đối tượng không lặp lại
  - Map: Tập các cặp khóa-giá trị (key-value) và không cho phép khóa lặp lại
    - Liên kết các đối tượng trong tập này với đối các đối tượng trong tập khác như tra từ điển/danh bạ điện thoại.

Lập trình tổng quát trong Java

- Cây cấu trúc giao diện Collection



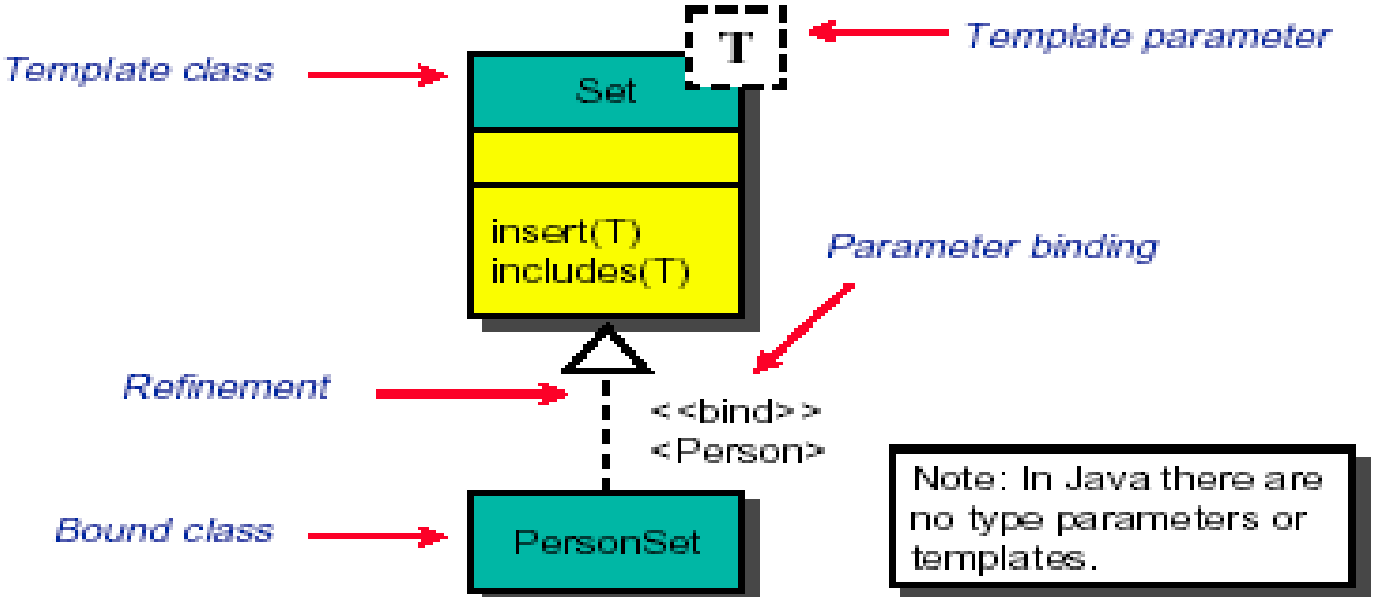
Lập trình tổng quát trong Java

- So sánh Tập hợp và mảng

Tập hợp	Mảng
Tập hợp (có thể) truy xuất theo dạng ngẫu nhiên	Mảng truy xuất 1 cách tuần tự
Tập hợp có thể chứa nhiều loại đối tượng/dữ liệu khác nhau	Mảng chứa 1 loại đối tượng/dữ liệu nhất định
Dùng theo kiểu tập hợp xây dựng sẵn của Java chỉ khai báo và gọi những phương thức đã được định nghĩa	Dùng tổ chức dữ liệu theo mảng phải lập trình hoàn toàn
Duyệt các phần tử tập hợp thông qua Iterator	Duyệt các phần tử mảng tuần tự thông qua chỉ số mảng

Lập trình tổng quát trong Java

- Các giao diện và lớp thực thi trong Collection framework của Java đều được xây dựng theo template
  → cho phép xác định tập hợp các phần tử cùng kiểu nào đó bất kỳ
  - Cho phép chỉ định kiểu dữ liệu của các Collection ➔ hạn chế việc thao tác sai kiểu dữ liệu



Lập trình tổng quát trong Java

- **Ví dụ**

```
public interface List<E> {
  void add(E x);
  Iterator<E> iterator();
}
```

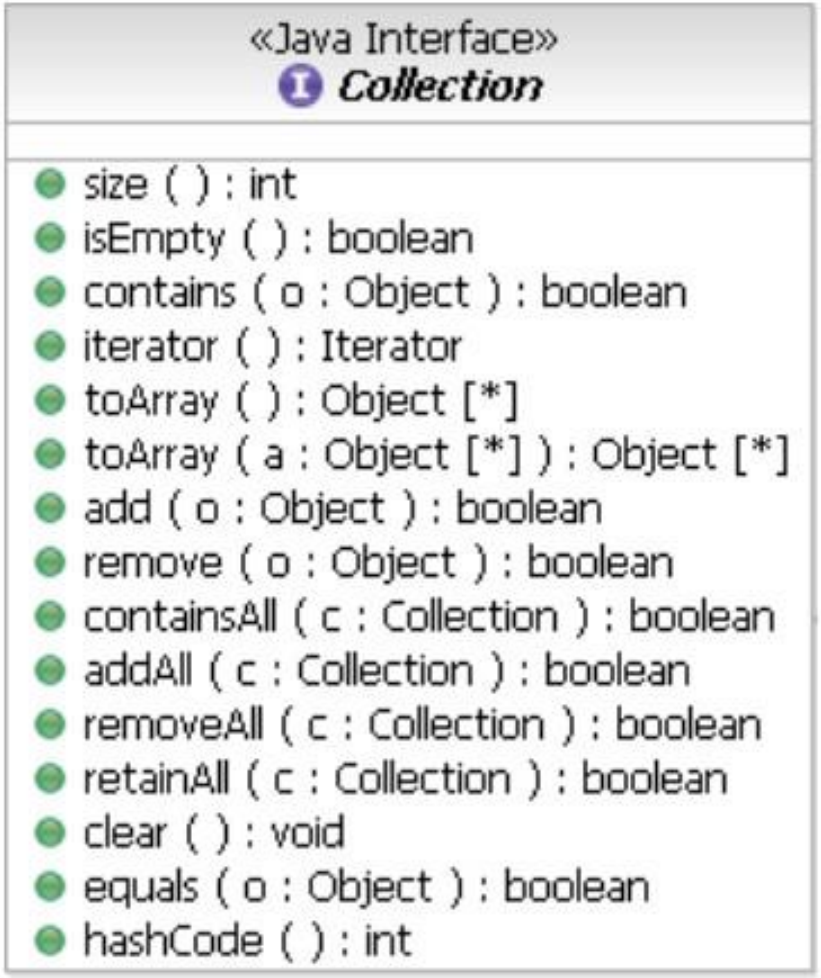
- **Ví dụ**

```
List<String> myList = new ArrayList<String>();
myList.add("Fred");        // OK
myList.add(new Dog());     //Compile error!

String s = myList.get(0);
```

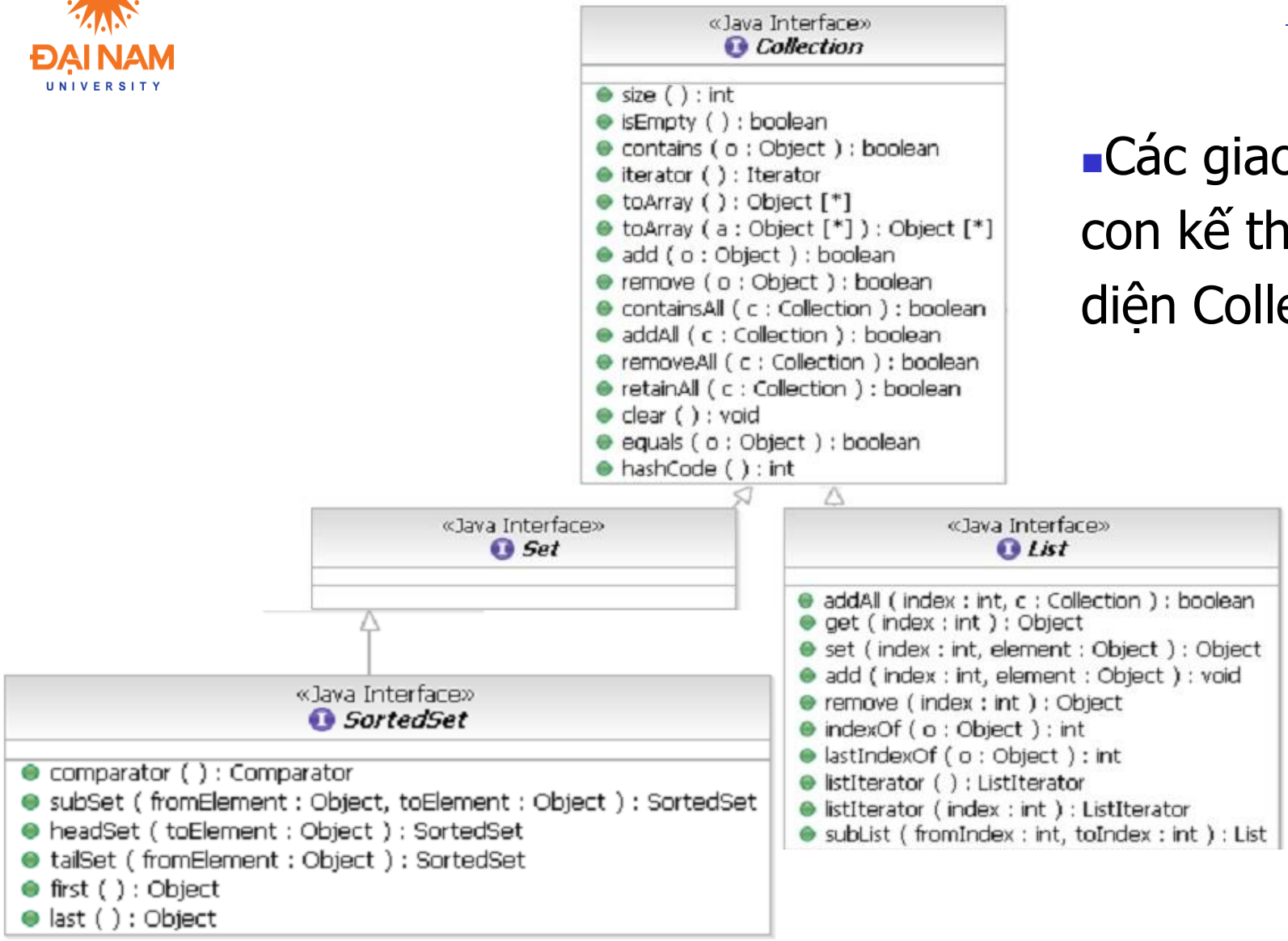
Giao diện Collection

- Xác định giao diện cơ bản cho các thao tác với một tập các đối tượng
  - Thêm vào tập hợp
  - Xóa khỏi tập hợp
  - Kiểm tra có là thành viên
- Chứa các phương thức thao tác trên các phần tử riêng lẻ hoặc theo khối
- Cung cấp các phương thức cho phép thực hiện duyệt qua các phần tử trên tập hợp (lặp) và chuyển tập hợp sang mảng



Giao diện Collection

```
public interface Collection {
    // Basic Operations
    int  size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element);
    boolean remove(Object element);
    Iterator iterator();
    // Bulk Operations
    boolean addAll(Collection c);
    boolean removeAll(Collection c);
    boolean retainAll(Collection c);
    ….
    // Array Operations
    Object[] toArray();
    Object[] toArray(Object a[]);
 }
```



- Các giao diện con kế thừa giao diện Collection

Giao diện Set

- Set kế thừa từ Collection, hỗ trợ các thao tác xử lý trên collection kiểu tập hợp
- Ví dụ:
  - Set of cars:
    - {BMW, Ford, Jeep, Chevrolet, Nissan, Toyota, VW}
  - Nationalities in the class
    - {Chinese, American, Canadian, Indian}
- Một tập hợp các phần tử không được trùng lặp.
- Set không có thêm phương thức riêng ngoài các phương thức kế thừa từ Collection.

Giao diện List

- List kế thừa từ Collection, nó cung cấp thêm các phương thức để xử lý collection kiểu danh sách
  - Danh sách là một collection với các phần tử được xếp theo chỉ số
- Một số phương thức của List
  - Object get(int index);
  - Object set(int index, Object o);
  - void add(int index, Object o);
  - Object remove(int index);
  - int indexOf(Object o);
  - int lastIndexOf(Object o);

Giao diện Map

- Xác định giao diện cơ bản để thao tác với một tập hợp bao gồm cặp khóa-giá trị
  - Thêm một cặp khóa-giá trị
  - Xóa một cặp khóa-giá trị
  - Lấy về giá trị với khóa đã có
  - Kiểm tra có phải là thành viên (khóa hoặc giá trị)
- Cung cấp 3 cách nhìn cho nội dung của tập hợp:
  - Tập các khóa
  - Tập các giá trị
  - Tập các ánh xạ khóa-giá trị

Giao diện Map

- Giao diện Map cung cấp các thao tác xử lý trên các bảng ánh xạ
  - Bảng ánh xạ lưu các phần tử theo khoá và không được có 2 khoá trùng nhau
- Một số phương thức của Map
  - Object put(Object key, Object value);
  - Object get(Object key);
  - Object remove(Object key);
  - boolean containsKey(Object key);
  - boolean containsValue(Object value);
  - ...

Giao diện SortedMap

- Giao diện SortedMap
  - thừa kế giao diện Map
  - các phần tử được sắp xếp theo thứ tự
  - tương tự SortedSet, tuy nhiên việc sắp xếp được thực hiện với các khóa
- Phương thức: Tương tự Map, bổ sung thêm:
  - firstKey( ): returns the first (lowest) value currently in the map
  - lastKey( ): returns the last (highest) value currently in the map

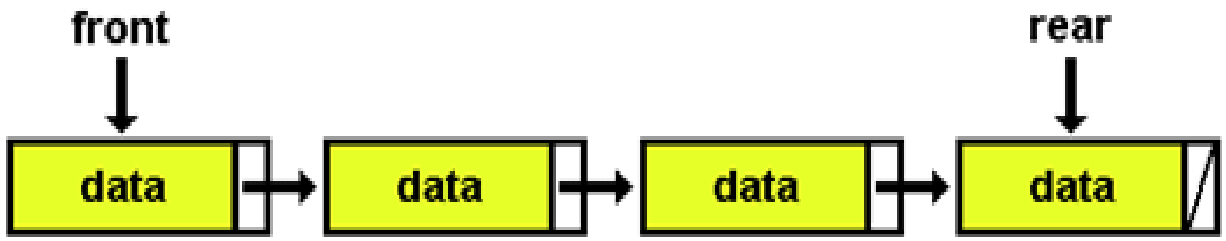
Các lớp thực thi giao diện Collection

- Java đã xây dựng sẵn một số lớp thực thi các giao diện Set, List và Map và cài đặt các phương thức tương ứng

		IMPLEMENTATIONS				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Legacy
INTERFACES	Set	HashSet		TreeSet		
	List		ArrayList		LinkedList	Vector, Stack
	Map	HashMap		TreeMap		HashTable, Properties

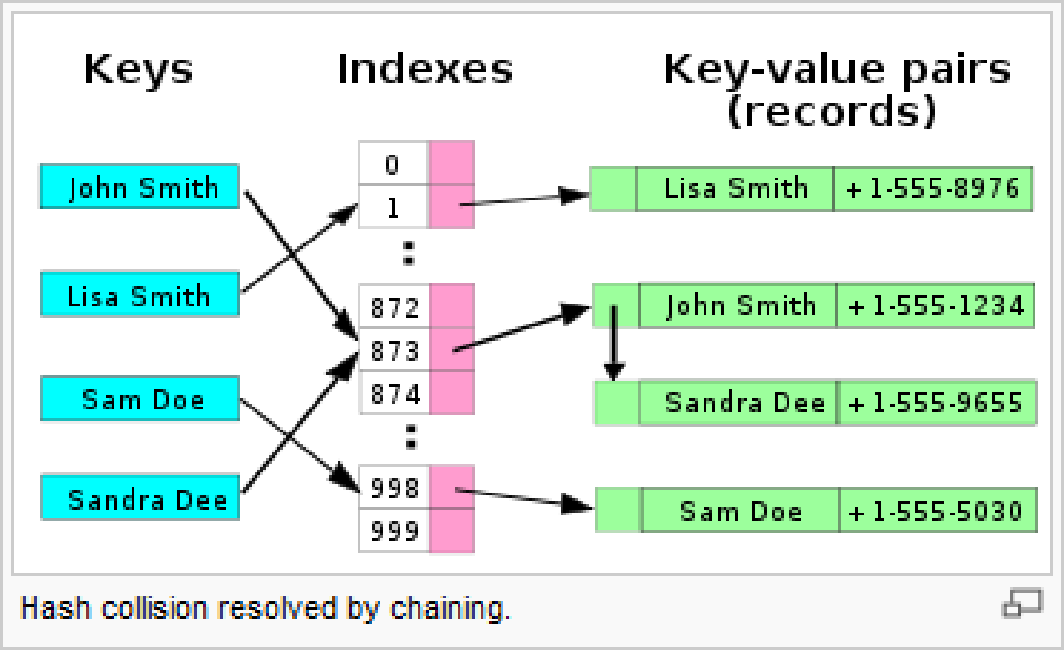
Các lớp thực thi giao diện Collection

- ArrayList: Mảng động, nếu các phần tử thêm vào vượt quá kích cỡ mảng, mảng sẽ tự động tăng kích cỡ
- LinkedList: Danh sách liên kết
  - Hỗ trợ thao tác trên đầu và cuối danh sách
  - Được sử dụng để tạo ngăn xếp, hàng đợi, cây…



Các lớp thực thi giao diện Collection

- HashSet: Bảng băm
  - Lưu các phần tử trong một bảng băm
  - Không cho phép lưu trùng lặp
  - Cho phép phần tử null



Hash collision resolved by chaining.

Các lớp thực thi giao diện Collection

- LinkedHashSet: Bảng băm kết hợp với linked list nhằm đảm bảo thứ tự các phần tử
  - Thừa kế HashSet và thực thi giao diện Set
  - Khác HashSet ở chỗ nó lưu trữ trong một danh sách móc nối đôi
  - Thứ tự các phần tử được sắp xếp theo thứ tự được insert vào tập hợp
- TreeSet: Cho phép lấy các phần tử trong tập hợp theo thứ tự đã sắp xếp
  - Các phần tử được thêm vào TreeSet tự động được sắp xếp
  - Thông thường, ta có thể thêm các phần tử vào HashSet, sau đó convert về TreeSet để duyệt theo thứ tự nhanh hơn

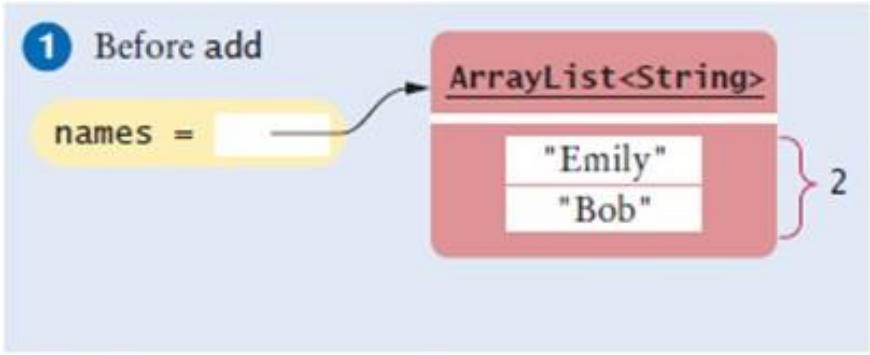
Các lớp thực thi giao diện Collection

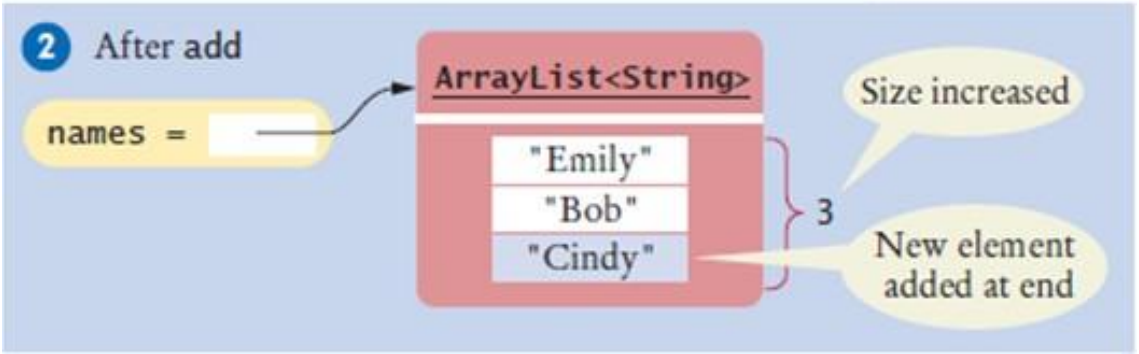
- HashMap: Bảng băm (cài đặt của Map)
- LinkedHashMap: Bảng băm kết hợp với linked list nhằm đảm bảo thứ tự các phần tử (cài đặt của Map)
- TreeMap: Cây (cài đặt của Map)
- Legacy Implementations
  - Là các lớp cũ được cài đặt bổ sung thêm các collection interface.
  - Vector: Có thể thay bằng ArrayList
  - Hastable: Có thể thay bằng HashMap

Các lớp thực thi giao diện Collection

- Ví dụ:

```
ArrayList<String> names =
            new ArrayList<String>();
names.add("Emily");
names.add("Bob");
names.add("Cindy");
```

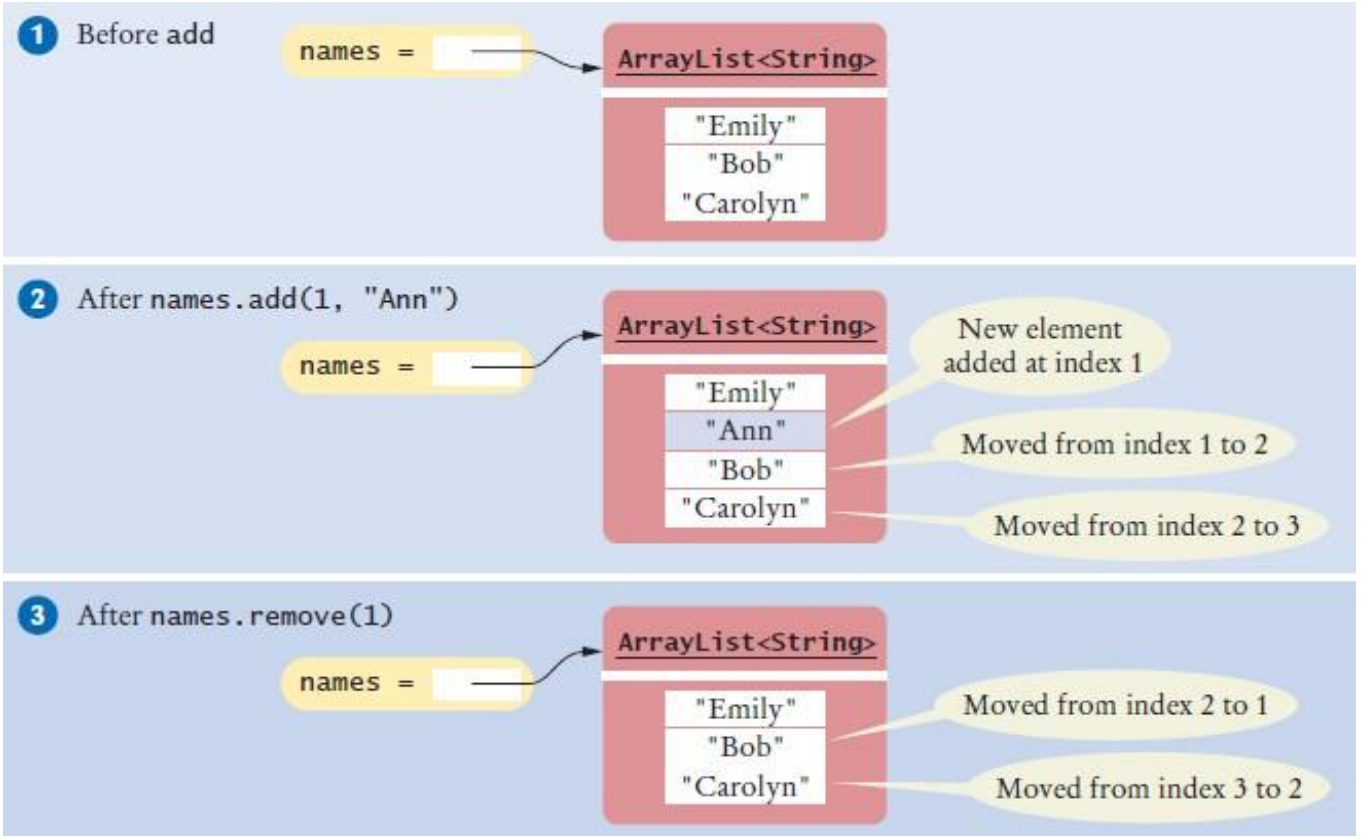




Các lớp thực thi giao diện Collection

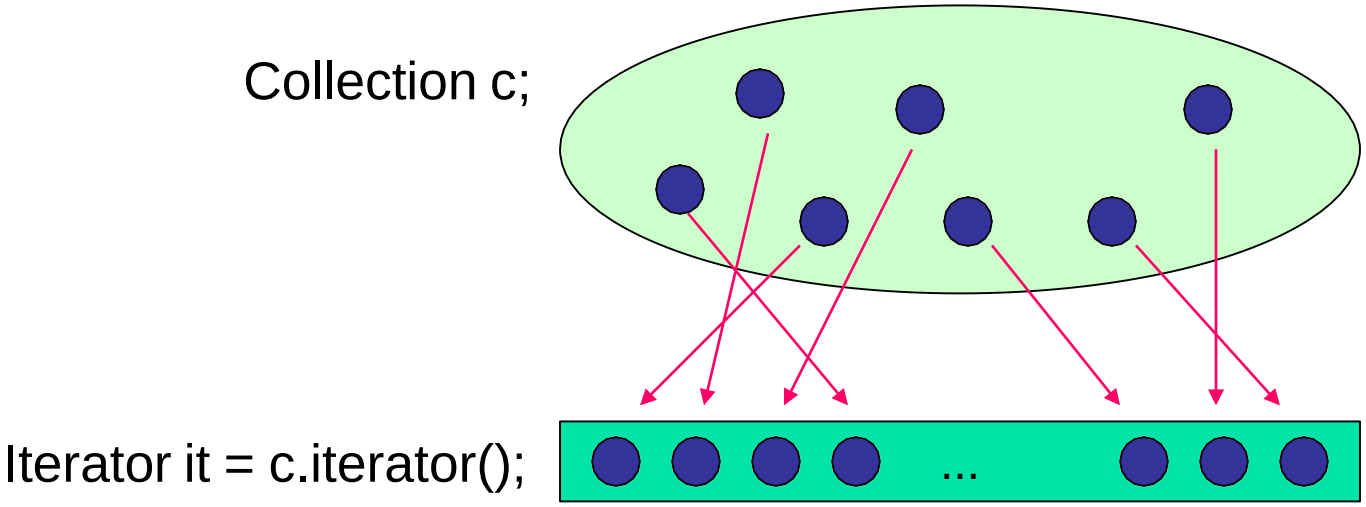
- Ví dụ:

```
String name = names.get(0);
names.add(1, "Ann");
names.remove(1);
```



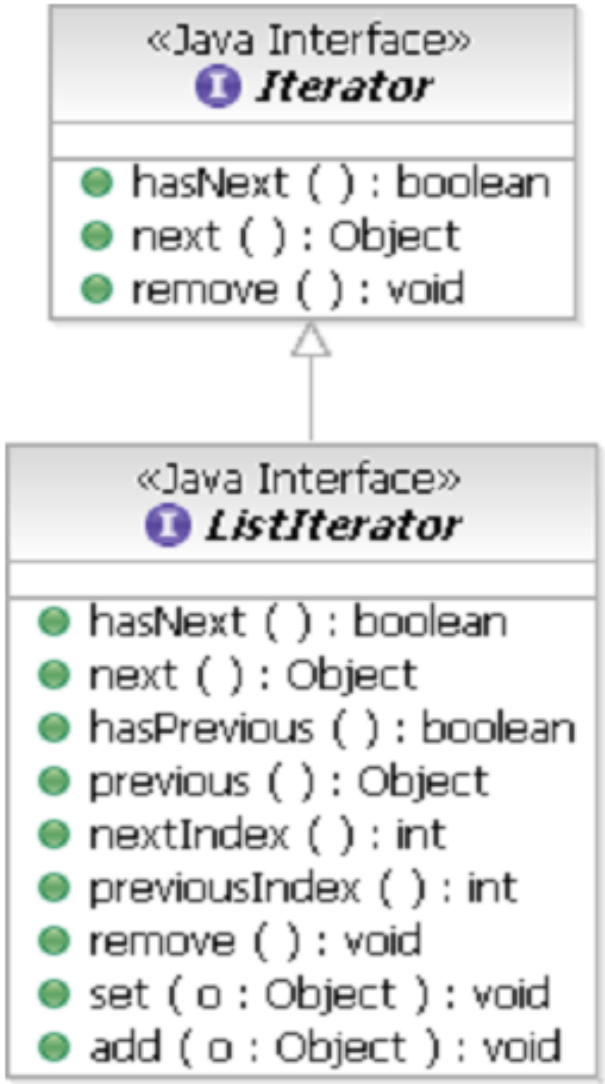
Giao diện Iterator và Comparator

- Sử dụng để duyệt và so sánh trên các Collection
- Iterator
  - Các phần tử trong collection có thể được duyệt thông qua Iterator



Giao diện Iterator và Comparator

- Iterator
  - Cung cấp cơ chế thuận tiện để duyệt (lặp) qua toàn bộ nội dung của tập hợp, mỗi lần là một đối tượng trong tập hợp
    - Giống như SQL cursor
  - Iterator của các tập hợp đã sắp xếp duyệt theo thứ tự tập hợp
  - ListIterator thêm các phương thức đưa ra bản chất tuần tự của danh sách cơ sở



Giao diện Iterator và Comparator

- Iterator : Các phương thức
  - iterator( ): yêu cầu container trả về một iterator
  - next( ): trả về phần tử tiếp theo
  - hasNext( ): kiểm tra có tồn tại phần tử tiếp theo hay không
  - remove( ): xóa phần tử gần nhất của iterator

Giao diện Iterator và Comparator

- Iterator: Ví dụ
  - Định nghĩa iterator

```
public interface Iterator {
        boolean hasNext();
        Object next();
        void remove();
}
```

  - Sử dụng iterator

```
Collection c;

Iterator i = c.iterator();
while (i.hasNext()) {
        Object o = i.next();
        // Process this object
}
```

Tương tự vòng lặp **for**

```
for (String name : names){
    System.out.println(name);
}
```

Giao diện Iterator và Comparator

- Giao diện Comparator được sử dụng để cho phép so sánh hai đối tượng trong tập hợp
- Một Comparator phải định nghĩa một phương thức compare( ) lấy 2 tham số Object và trả về -1, 0 hoặc 1
- Không cần thiết nếu tập hợp đã có khả năng so sánh tự nhiên (vd. String, Integer…)

Giao diện Iterator và Comparator

- Ví dụ lớp Person:

```
class Person {
    private int age;
    private String name;

    public void setAge(int age){
        this.age=age;
    }
    public int getAge(){
        return this.age;
    }
    public void setName(String name){
        this.name=name;
    }
    public String getName(){
        return this.name;
    }
}
```

Giao diện Iterator và Comparator

- Ví dụ Cài đặt AgeComparator :

```
class AgeComparator implements Comparator {
   public int compare(Object ob1, Object ob2) {
      int ob1Age = ((Person)ob1).getAge();
      int ob2Age = ((Person)ob2).getAge();

     if(ob1Age > ob2Age)
        return 1;
      else if(ob1Age < ob2Age)
        return -1;
      else
        return 0;
      }
   }
```

Giao diện Iterator và Comparator

- Ví dụ Sử dụng AgeComparator :

```
public class ComparatorExample {
   public static void main(String args[]) {
      ArrayList<Person> lst = new
         ArrayList<Person>();
      Person p = new Person();
      p.setAge(35); p.setName("A");
      lst.add(p);
      p = new Person();
      p.setAge(30); p.setName("B");
      lst.add(p);
      p = new Person();
      p.setAge(32); p.setName("C");
      lst.add(p);
}
```

Giao diện Iterator và Comparator

- Ví dụ Sử dụng AgeComparator :

```
System.out.println("Order before sorting");
    for (Person person : lst) {
        System.out.println(person.getName() +
                           "\t" + person.getAge());
    }
    Collections.sort(lst, new AgeComparator());
    System.out.println("\n\nOrder of person" +
                       "after sorting by age");
    for (Iterator<Person> i = lst.iterator();
         i.hasNext();) {
      Person person = i.next();
      System.out.println(person.getName() + "\t" +
                         person.getAge());
    } //End of for
  } //End of main
} //End of class
```

Ký tự đại diện (Wildcard)

Ký tự đại diện (Wildcard)

- Quan hệ thừa kế giữa hai lớp không có ảnh hưởng gì đến quan hệ giữa các cấu trúc tổng quát dùng cho hai lớp đó.
- Ví dụ:
  - Dog và Cat là các lớp con của Animal

  → Có thể đưa các đối tượng Dog và Cat vào một ArrayList<Animal>
  - Tuy nhiên, ArrayList<Dog>, ArrayList<Cat> lại không có quan hệ gì với ArrayList<Animal>

Ký tự đại diện (Wildcard)

- **Generic**
  - Kiểu khai báo trong lớp tổng quát (template) khi khởi tạo phải cùng với kiểu của các đối tượng thực sự.
  - Nếu khai báo **List<Foo>** ➔ Danh sách chỉ chấp nhận các đối tượng lớp Foo, các đối tượng là cha hoặc con của lớp **Foo** sẽ không được chấp nhận.

```
class Parent { }
class Child extends Parent { }
List<Parent> myList = new ArrayList<Child>();
```

Ký tự đại diện (Wildcard)

- Làm thế nào để xây dựng các tập hợp dành cho kiểu bất kì là lớp con của lớp cụ thể nào đó?

  → Giải pháp là sử dụng kí tự đại diện (wildcard)

- Ký tự đại diện: **?** dùng để hiển thị cho một kiểu dữ liệu chưa biết trong **`collection`**

```
void printCollection(Collection<?> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

- Khi biên dịch, dấu `?` có thể được thay thế bởi bất kì kiểu dữ liệu nào.

Ký tự đại diện (Wildcard)

- Tuy nhiên viết như thế này là không hợp lệ:

```
Collection<?> c = new ArrayList<String>();
c.add("a1"); //compile error, null
```

- Vì không biết c đại diện cho tập hợp kiểu dữ liệu nào c không thể thêm phần tử vào c

Ký tự đại diện (Wildcard)

- "? extends Type": Xác định một tập các kiểu con của Type. Đây là wildcard hữu ích
- "? super Type": Xác định một tập các kiểu cha của Type
- "?": Xác định tập tất cả các kiểu hoặc bất kỳ kiểu nào

Ký tự đại diện (Wildcard)

- Ví dụ:
  - `? extends Animal` có nghĩa là kiểu gì đó thuộc loại Animal
- Hai cú pháp sau là tương đương:

```
public void foo(ArrayList<? extends Animal> a)
public <T extends Animal> void foo( ArrayList<T> a)
```

  - Dùng "T", thường được sử dụng khi còn muốn T xuất hiện ở các vị trí khác

XIN CẢM ƠN!