LẬP TRÌNH NÂNG CAO

Lecture 01: Generics, Collections

multicampus

Mục lục

multicampus

Generic programming (Generic Types)

multicampus

- Tổng quát hóa chương trình để có thể hoạt động với các kiểu dữ liệu khác nhau, kể cả các kiểu dữ liệu trong tương lai (thuật toán đã xác định). Ví dụ, kiểu ngăn xếp làm việc với các kiểu phần tử khác nhau.
- Các kỹ thuật lập trình:
  - C, C++, Java từ 1.4 về trước: dùng con trỏ *void*, *template, Object*
  - Java 1.5,1.6: Generics
  - DotNet (C#): Generics

Generic programming (Generic Types)

multicampus

- **Sử dụng lớp tập hợp và các phép toán trên tập hợp để minh hoạ:**
  - *aSet*.**Member**(*element*)
  - *aSet*.**Insert**(*element*)
  - *aSet*.**Delete**(*element*)
  - *aSet*.**Count**
  - *aSet*.**Subset** (*anotherSet*)
  - *aSet*.**GetEnumerator**()
  - *aSet*.**Intersection** (*anotherSet*)
  - *aSet*.**Union** (*anotherSet*)
  - *aSet*.**Diff** (*anotherSet*)

The classes IntSet

multicampus

```
public class IntSet {
  private int capacity;   private static int DefaultCapacity = 10;
  private int [] store;  private int next;
  public IntSet(int capacity)
        {
          this.capacity = capacity;     store = new int [capacity];    next = 0;
         }
  public IntSet() {this(DefaultCapacity);   }
  public IntSet( int[] elements)
        { this(elements.length);
          for  (int i=0;i<elements.length;i++) this.Insert(elements[i]);
         }  // Copy constructor
  public IntSet(IntSet s)
        {  this(s.capacity);
          for  (int i=0;i<s.next;i++)      this.Insert(s.store[i]);
        }
   public bool Member( int  element){   }
   public void Insert( int  element){ store[next] =element; next++;}
   public void Delete( int  element){   }
}
```

and StringSet

multicampus

```
public class StringSet{
  private int capacity;   private static int DefaultCapacity = 10;
  private String [] store;  private int next;
  public IntSet(int capacity)  {
          this.capacity = capacity;     store = new String [capacity];    next = 0;
        }
  public IntSet() {this(DefaultCapacity);   }
  public IntSet(String[] elements) {
          this(elements.length);
          for  (int i=0;i<elements.length;i++) this.Insert(elements[i]);
        }  // Copy constructor
  public IntSet(IntSet s) {
          this(s.capacity);
          for  (int i=0;i<s.next;i++)      this.Insert(s.store[i]);
        }
   //  public bool Member(String  element){   }
    public void Insert(String  element){ store[next] =element; next++;}
   //  public void Delete(String  element){   }
}
```

The class ObjectSet

multicampus

```
public class ObjectSet{
  private int capacity;   private static int DefaultCapacity = 10;
  private Object [] store;  private int next;
  public IntSet(int capacity)  {
          this.capacity = capacity;     store = new Object [capacity];    next = 0;
        }
  public IntSet() {this(DefaultCapacity);   }
  public IntSet(Object[] elements)
       { this(elements.length);
         for  (int i=0;i<elements.length;i++) this.Insert(elements[i]);
        }  // Copy constructor
  public IntSet(IntSet s)
       {  this(s.capacity);
         for  (int i=0;i<s.next;i++)      this.Insert(s.store[i]);
       }
   //  public bool Member(Object element){   }
    public void Insert(Object element){ store[next] =element; next++;}
   //  public void Delete(Object element){   }
}
```

Problems

multicampus

- Problems with **IntSet** and **StringSet**
  - Tedious to write both versions: *Copy and paste* programming.
  - Error prone to maintain both versions.

- Problems with **ObjectSet**
  - Elements of the set must be downcasted in case we need to use some of their specialized operations.

Generic Classes in Java

multicampus

```java
public class StudentDAO {
    public void save(Student student) {
        // code to save student details to database
    }

    public Student get(long id) {
        // code to get student details from database...
        // ...and return a Student object
    }
}
```

```java
public class ProfessorDAO {
    public void save(Professor professor) {
        // code to save professor details to database
    }

    public Professor get(long id) {
        // code to get professor details from database...
        // ...and return a Professor object
    }
}
```

Using generics, we can write a more general DAO class like the following:

```java
public class GeneralDAO<T> {

    public void save(T entity) {
        // code to save entity details to database
    }

    public T get(long id) {
        // code to get details from database...
        // ...and return a T object
    }
}
```

Generic Classes in Java

multicampus

Using generics, we can write a more general DAO class like the following:

```
1  public class GeneralDAO<T> {
2
3      public void save(T entity) {
4          // code to save entity details to database
5      }
6
7      public T get(long id) {
8          // code to get details from database...
9          // ...and return a T object
10     }
11 }
```

- Here, T is called ***type parameter*** of the GeneralDAO class. T stands for any type. The GeneralDAO class is said to be ***generified***. The following code illustrates how to use this generic class:

```
1  GeneralDAO<Student> studentDAO = new GeneralDAO<Student>();
2
3  Student newStudent = new Student();
4  studentDAO.save(newStudent);
5
6  Student oldStudent = studentDAO.get(250);
```

Generic classes with more than one type parameter

multicampus

```
1  public class Pair<T, U> {
2      T first;
3      U second;
4
5      public Pair(T first, U second) {
6          this.first = first;
7          this.second = second;
8      }
9
10     public T getFirst() {
11         return first;
12     }
13
14     public U getSecond() {
15         return second;
16     }
17 }
```

Bounded type parameters

multicampus

```java
public class GeneralDAO<T extends Entity> {

    public void save(T entity) {
        // code to save entity details to database
    }

    public T get(long id) {
        // code to get details from database...
        // ...and return a T object
    }
}
```

- Entity is called the upper bound, which can be any class or interface.
- The GeneralDAO class can be used only work with sub types of Entity, not with every type

Using multiple bounds

multicampus

- We can use the syntax **<T extends X & Y & Z>** to define a generic class whose type parameter can be sub types of multiple types. Here, X, Y, Z can be classes and interfaces
- For example, the following generic class is designed works with only types that are sub types of Runnable and JFrame:

```
1  public class WindowApp<T extends JFrame & Runnable> {
2      T theApp;
3
4      public WindowApp(T app) {
5          theApp = app;
6      }
7  }
```

Generic Methods

multicampus

- The following method counts number of occurrences of a String in an array of Strings:

```java
public static int count(String[] array, String item) {
    int count = 0;

    if (item == null) {
        for (String s : array) {
            if (s == null) count++;
        }
    } else {
        for (String s : array) {
            if (item.equals(s)) {
                count++;
            }
        }
    }

    return count;

}
```

Here's an example usage of this method:

```java
String[] helloWorld = {"h", "e", "l", "l", "o", "w", "o", "r", "l", "d"};
int count = count(helloWorld, "l");
System.out.println("#occurrences of l: " + count);
```

Generic Methods

multicampus

- Now, we need to count the occurrence of an element in an array of any type.

  The <T> is always placed before the return type of the method

```java
public static <T> int count(T[] array, T item) {
    int count = 0;

    if (item == null) {
        for (T element : array) {
            if (element == null) count++;
        }
    } else {
        for (T element : array) {
            if (item.equals(element)) {
                count++;
            }
        }
    }

    return count;

}
```

With this generic version, we can count occurrence of a number in an array of integers like this:

```java
Integer[] integers = {1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0};
int count = count(integers, 0);
System.out.println("#occurrences of zeros: " + count);
```

Generic Methods

multicampus

- Note that if the type parameter of a non-static generic method is same as the enclosing class, the indicator <T> is not required. The following class illustrates this point:

```
1  public class Util<T> {
2
3      public int count(T[] array, T item) {
4          // code...
5      }
6  }
```

Wildcards in generic methods

multicampus

- **extends wildcard**

```java
private static double sum(Collection<? extends Number> numbers) {
    double result = 0.0;

    for (Number num : numbers) result += num.doubleValue();

    return result;
}
```

- The compile only allows us to pass a collection of a subtype of Number

```java
List<Integer> integers = Arrays.asList(2, 4, 6);
double sum = sum(integers);
System.out.println("Sum of integers = " + sum);


List<Double> doubles = Arrays.asList(3.14, 1.68, 2.94);
sum = sum(doubles);
System.out.println("Sum of doubles = " + sum);

List<Number> numbers = Arrays.<Number>asList(2, 4, 6, 3.14, 1.68, 2.94);
sum = sum(numbers);
System.out.println("Sum of numbers = " + sum);
```

Wildcards in generic methods

multicampus

- **super wildcard (áp dụng với lớp tương ứng và các lớp cha)**

```
1  public static void append(Collection<? super Integer> integers, int n) {
2      for (int i = 1; i <= n; i++) {
3          integers.add(i);
4      }
5  }
```

Then it's legal to pass a list of numbers like this:

```
1  List<Number> numbers = new ArrayList<Number>();
2  append(numbers, 5);
3  numbers.add(6.789);
4
5  System.out.println(numbers);
```

- Do lớp Numer là lớp cha của tất các các lớp: BigDecimal, BigInteger, Byte, Double, Float, **Integer**, Long, Short

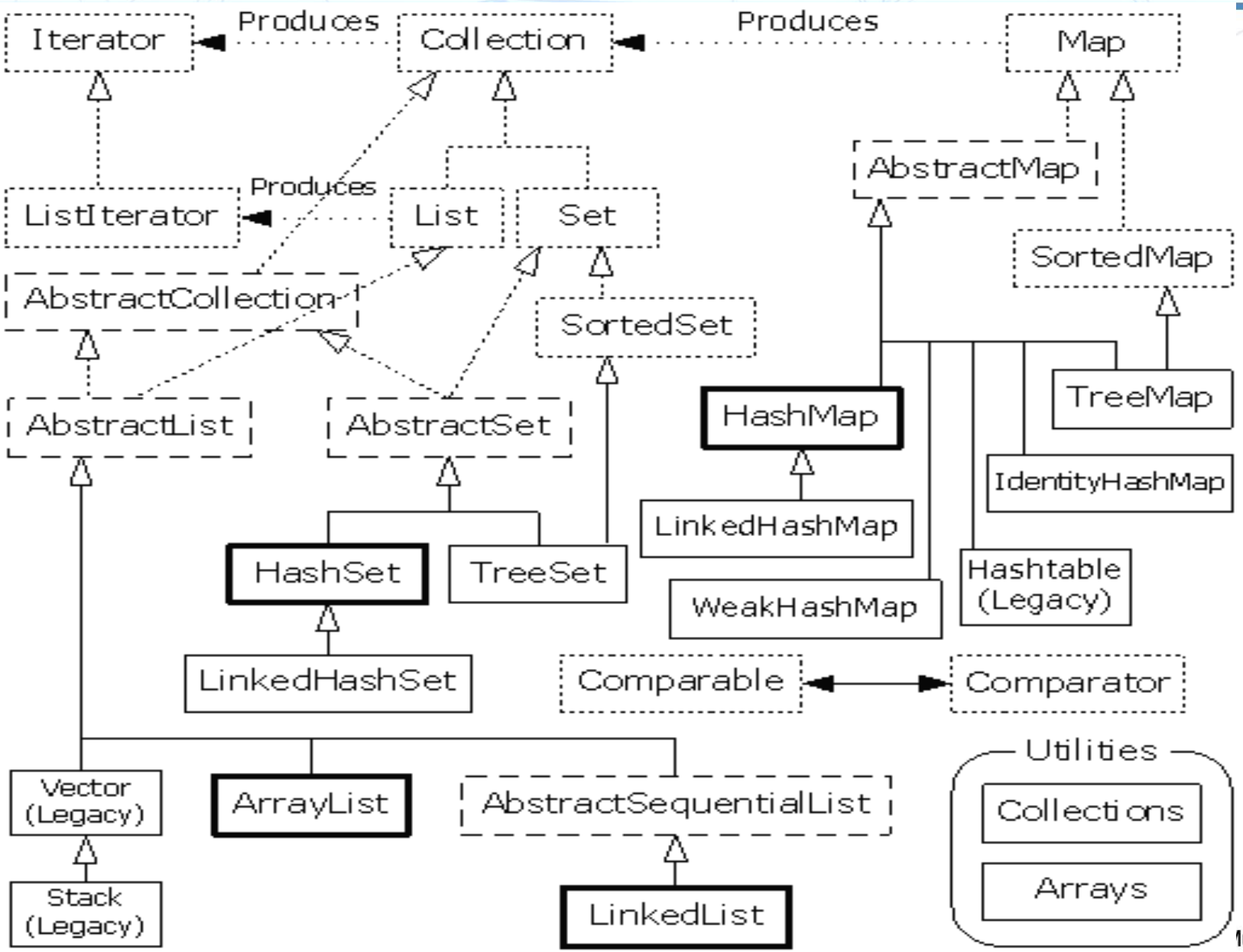
Java Collections Framework

multicampus

- A **collection** is a data structure that holds a set of objects. It looks like arrays but collections are more advanced and more flexible.
- **Java Collections Framework** is a set of reusable data structures and algorithms.
- **Why Use Java Collections Framework**
  - ***Reduce programming effort:*** with the reusable and useful data structures and algorithms, the programmers do not have to re-invent the wheel, thus they can devote their time on developing application's business.
  - ***Increase program speed and quality***
  - ***Software reuse:*** due to the Java Collections Framework is built into the JDK, code written using collections framework can be re-used every where among applications, libraries and APIs. That cuts development cost and increases interoperability among Java programs.

Java Collections Framework

multicampus



Java Collections Framework

multicampus

- Starting with Java 5, a collection holds objects of a specified type (tổng quát). A collection class's or interface's definition takes object type as a parameter:
  - *Collection*<***E***>
  - *List*<***E***>
  - Stack<***E***>
  - *Set*<***E***>
- A map takes two object type parameters:
  - Map<***K***,***V***>

Collection, Iterator Interface

- *Collection* interface represents any collection.
- An iterator is an object that helps to traverse the collection (process all its elements in sequence).
- A collection supplies its own iterator(s), (returned by collection's iterator method);

Collection<E> Methods

multicampus


```
boolean isEmpty ()
int size ()
boolean contains (E obj)
boolean add (E obj)
boolean remove (E obj)
Iterator<E> iterator ()
//   ... other methods
```

Supplies an iterator for this collection

Iterator<E> Methods

multicampus

boolean hasNext ()

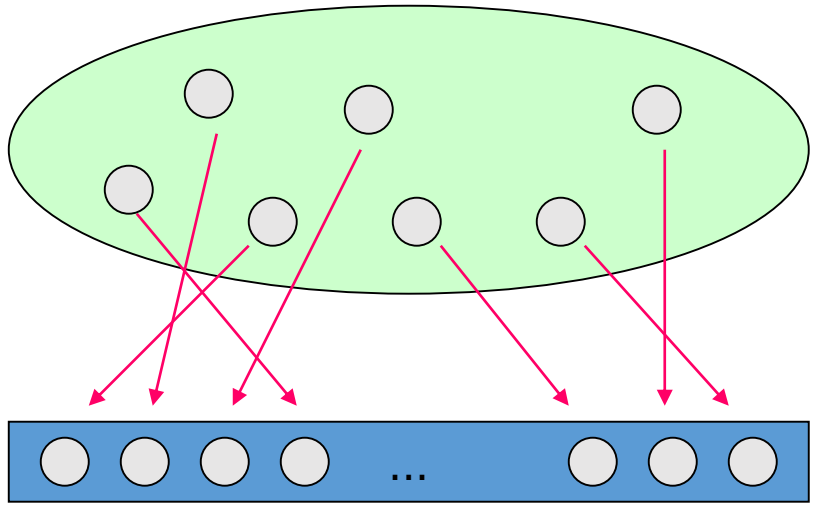
E next ()

void remove ()

What's "next" is determined by a particular collection

Removes the last visited element

Collection c;



Iterator it = c.iterator();

Iterator ⇔ “For Each” Loop

```
Collection<String> words = new ArrayList<String>();  ...
```

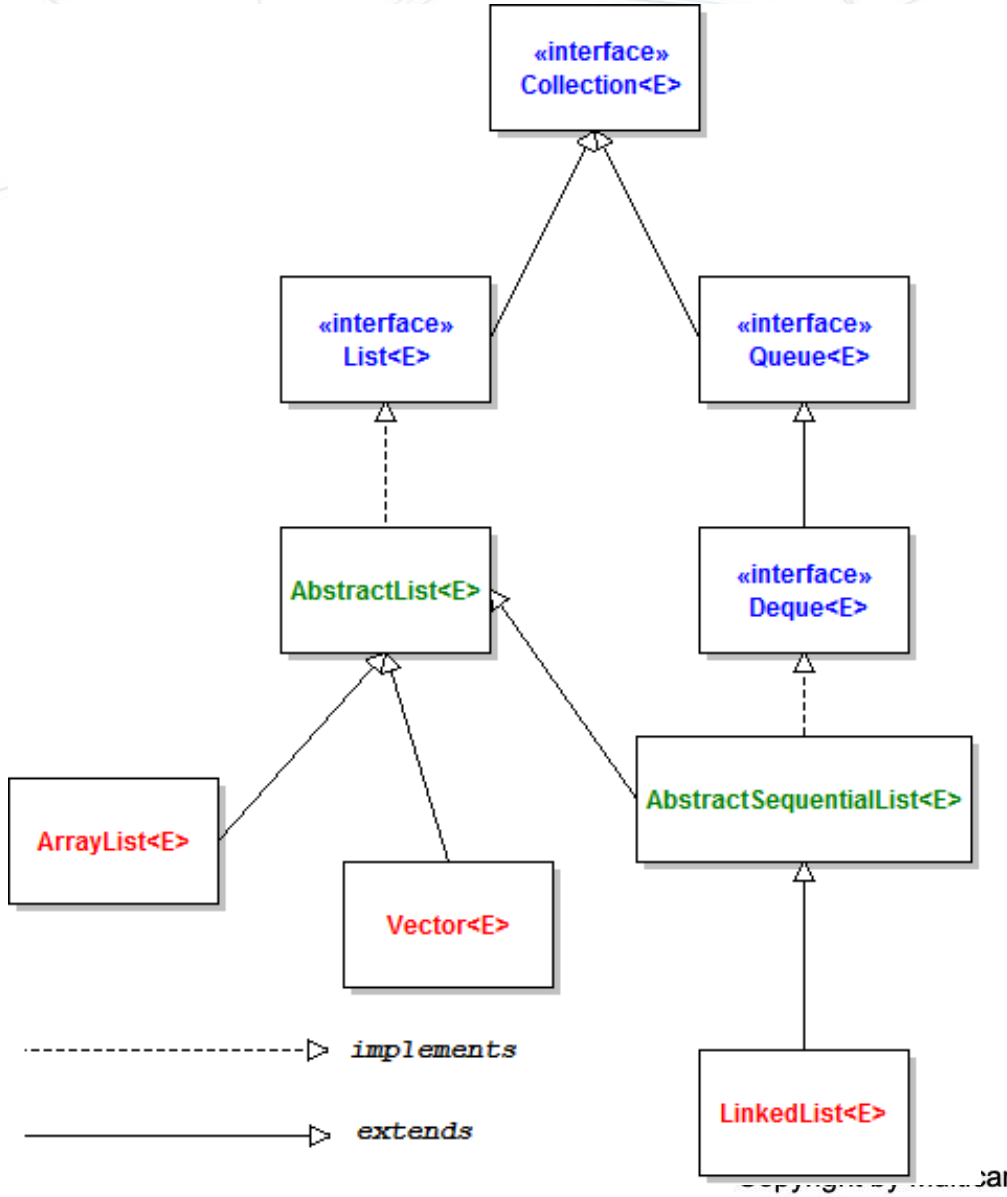
```
for (String word : words) {
    < ... process word >
}
```

```
Iterator<String> iter =
            words.iterator();
while (iter.hasNext ())  {
    String word = iter.next ();
    < ... process word >
}
```

A “for each” loop is a syntactic shortcut that replaces an iterator

List Collection

multicampus



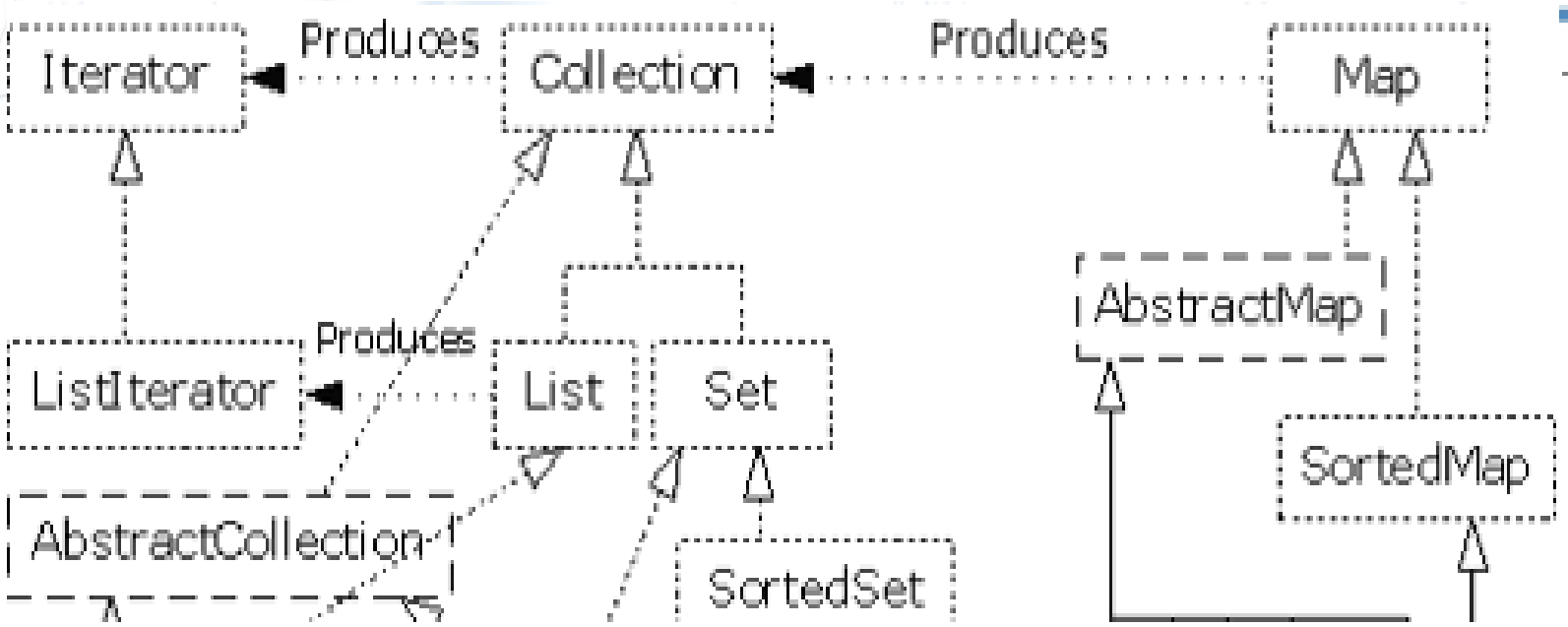
Lists, *ListIterator*

multicampus

- A list represents a collection in which all elements are numbered by indices:

$$a_0, a_1, ..., a_{n-1}$$

- java.util:
  - *List, ListIterator* interface
  - **ArrayList**
  - **LinkedList**



- *ListIterator* is an extended iterator, specific for lists (*ListIterator* is a subinterface of *Iterator*)

List<E> Methods

multicampus

```
// All Collection<E> methods, plus:
E get (int i)
E set (int i, E obj)
void add (int i, E obj)
E remove (int i)
int indexOf (Object obj)
ListIterator<E> listIterator ()
ListIterator<E> listIterator (int i)
```

These methods are familiar from ArrayList, which implements List

Returns a ListIterator that starts iterations at index i

ListIterator<E> Methods

multicampus

```
// The three Iterator<E> methods, plus:
int nextIndex ()
boolean hasPrevious ()
E previous ()
int previousIndex ()
void add (E obj)
void set (E obj)
```

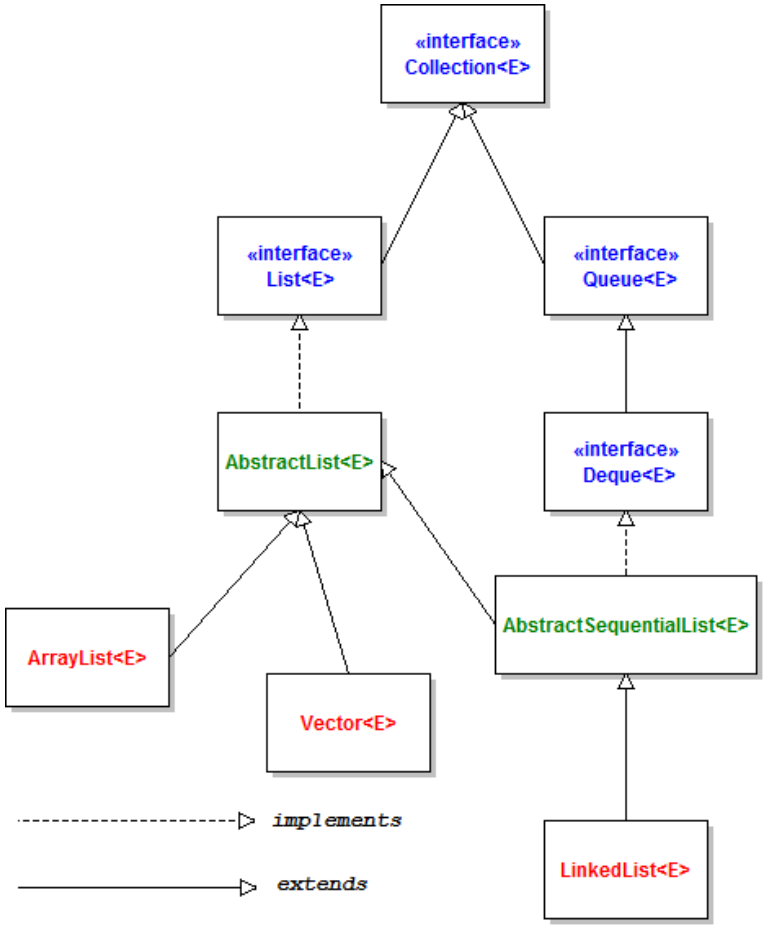
Can traverse the list backward

Can add elements to the list (inserts after the last visited element)

Can change elements (changes the last visited element)

ArrayList

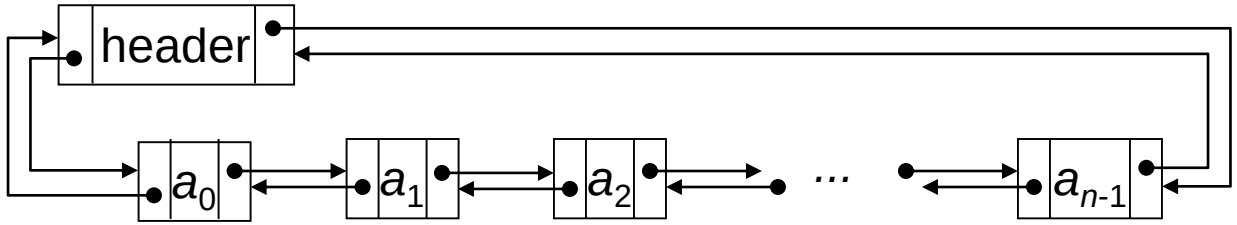
- Represents a list as a *dynamic **array*** (array that is resized when full)
- Static array?
- Provides *random access* to the elements
- Implements all the methods of *List<E>*

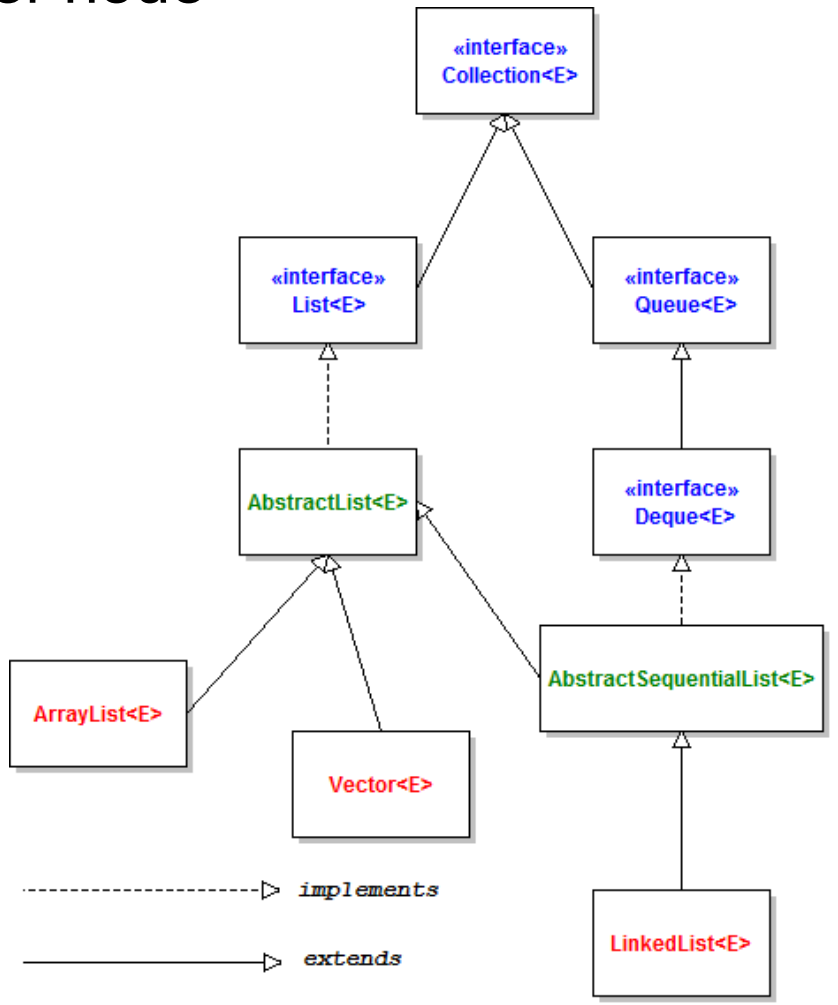


LinkedList

- Represents a list as a doubly-linked list with a header node

- Implements all the methods of *List<E>*





LinkedList (cont'd)

multicampus

- Additional methods specific to LinkedList:

```
void addFirst (E obj)
void addLast (E obj)
E getFirst ()
E getLast ()
E removeFirst ()
E removeLast ()
```

ArrayList vs. LinkedList

multicampus

- Implements a list as **an array**

\+ Provides ***random access*** to the elements

– Inserting and removing elements requires shifting of subsequent elements

– Needs to be resized when runs out of space

- Implements a list as a **doubly-linked list** with a header node

– No random access to the elements — needs to traverse the list to get to the *i*-th element

\+ Inserting and removing elements is done by rearranging the links — no shifting

\+ Nodes are allocated and released as necessary

ArrayList vs. LinkedList (cont'd)

multicampus

	ArrayList	LinkedList
get(i) and **set(i, obj)**	***O*(1)**	*O*(*n*)
add(i, obj) and **remove(i)**	*O*(*n*)	*O*(*n*)
add(0, obj)	*O*(*n*)	***O*(1)**
add(obj)	***O*(1)**	***O*(1)**
contains(obj)	*O*(*n*)	*O*(*n*)

ArrayList vs. LinkedList (cont'd)

multicampus

```
for (int i = 0;  i < list.size();  i++) {
   Object x = list.get (i);
   ...
}
```

Works well for an ArrayList — $O(n)$
inefficient for a LinkedList — $O(n^2)$

```
Iterator iter = list.iterator ( );
while (iter.hasNext ( )) {
   Object x = iter.next ( );
   ...
}
```

```
for (Object x : list) {
   ...
}
```

Work well for <u>both</u> an ArrayList and a LinkedList — $O(n)$

Contains method

multicampus

```
public static void main(String[] args) throws ParseException {
     DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
    List<Employee2> listEmployees = new LinkedList<>();
    Employee2 employee1 = new Employee2("Tom", "Eagar", dateFormat.parse("2007-12-03"));
    Employee2 employee2 = new Employee2("Tom", "Smith", dateFormat.parse("2005-06-20"));
    Employee2 employee3 = new Employee2("Bill", "Joy", dateFormat.parse("2009-01-31"));
    Employee2 employee4 = new Employee2("Bill", "Gates", dateFormat.parse("2005-05-12"));
    listEmployees.add(employee1);
    listEmployees.add(employee2);
    listEmployees.add(employee3);
    listEmployees.add(employee4);
      Employee2 empCantim = new Employee2("Bill", "Joy", dateFormat.parse("2009-01-31"));
    boolean exit=listEmployees.contains(empCantim);
    if (exit)
        System.out.println("Có");
    else
        System.out.println("Không");
}
```

Utility classes: Collections and Arrays

multicampus

- **Collections.sort(list):**
  - **sort (List<T> list)**
  - **binarySearch (List<? extends T> list,T key, Comparator<? super T> c)**
- **Arrays:**
  - **sort (T[] a, Comparator<? super T> c)**
  - **binarySearch (T[] a, T key, Comparator<? super T> c)**

Utility classes: Collections and Arrays

multicampus

- .

```
List<String> names = Arrays.asList(
        "Tom", "Peter", "Alice", "Bob", "Sam",
        "Mary", "Jane", "Bill", "Tim", "Kevin");
System.out.println("Before sorting: " + names);
Collections.sort(names);
System.out.println("After sorting: " + names);
int i= Collections.binarySearch(names, "Bob");
if (i>0) {
    System.out.println("Tìm thấy Bob ở vị trí:"+i);
} else System.out.println("Không tìm thấy Bob");

List<Integer> numbers = Arrays.asList(8, 2, 5, 1, 3, 4, 9, 6, 7, 10);
System.out.println("Before sorting: " + numbers);
Collections.sort(numbers);
System.out.println("After sorting: " + numbers);
```

Utility classes: Collections and Arrays

multicampus

- .

```
int[] arr = {13, 7, 6, 45, 21, 9, 101, 102};
Arrays.sort(arr);
System.out.printf("Modified arr[] : %s",
                  Arrays.toString(arr));
int i= Arrays.binarySearch(arr,45);
if (i>0) {
    System.out.println("Tìm thấy 45 ở vị trí:"+i);
} else System.out.println("Không tìm thấy 45");
```

Object Ordering in Java

multicampus

- Lỗi biên dịch:

```
public class Employee {
    String firstName;
    String lastName;
    Date joinDate;
    public Employee(String firstName, String lastName)
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public String toString() {
        return firstName + " " + lastName;
    }
    // getters and setters
}
```

```
public static void main(String[] args) {
    List<Employee> listEmployees = new ArrayList<>();
    Employee employee1 = new Employee("Tom", "Eagar");
    Employee employee2 = new Employee("Tom", "Smith");
    Employee employee3 = new Employee("Bill", "Joy");
    Employee employee4 = new Employee("Bill", "Gates");
    Employee employee5 = new Employee("Alice", "Wooden");
    listEmployees.add(employee1);
    listEmployees.add(employee2);
    listEmployees.add(employee3);
    listEmployees.add(employee4);
    listEmployees.add(employee5);
    Collections.sort(listEmployees);
}
```

Object Ordering in Java

multicampus

- Giao diện Comparable

```java
public interface Comparable<T> {
    public int compareTo(T object);
}
```

```java
public class Employee2 implements Comparable<Employee> {
     String firstName;
    String lastName;
    Date joinDate;
    public Employee2(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public String toString() {
        return firstName + " " + lastName;
    }
    public int compareTo(Employee another) {
        int compareValue = this.firstName.compareTo(another.firstName);
        if (compareValue == 0) {
            return this.lastName.compareTo(another.lastName);
        }
        return compareValue;
    }
     // getters...
    // setters...
}
```

Object Ordering in Java

multicampus

- Giao diện Comparator

```java
public interface Comparator<T> {
    public int compare(T obj1, T obj2);
}
```

```java
class EmployeeComparator implements Comparator<Employee2> {
    public int compare(Employee2 emp1, Employee2 emp2) {
        return emp1.getJoinDate().compareTo(emp2.getJoinDate());
    }
}
```

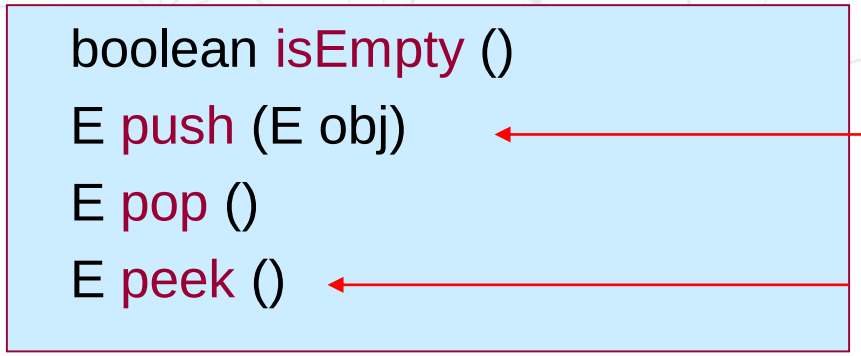
- Demo

Stacks

multicampus

- A stack provides temporary storage in the LIFO (Last-In-First-Out) manner.
- Stacks are useful for dealing with nested structures and branching processes:
  - pictures within pictures
  - folders within folders
  - methods calling other methods
- Controlled by two operations: *push* and *pop*.
- Implemented as java.util.Stack<*E*> class

Stack<*E*> Methods



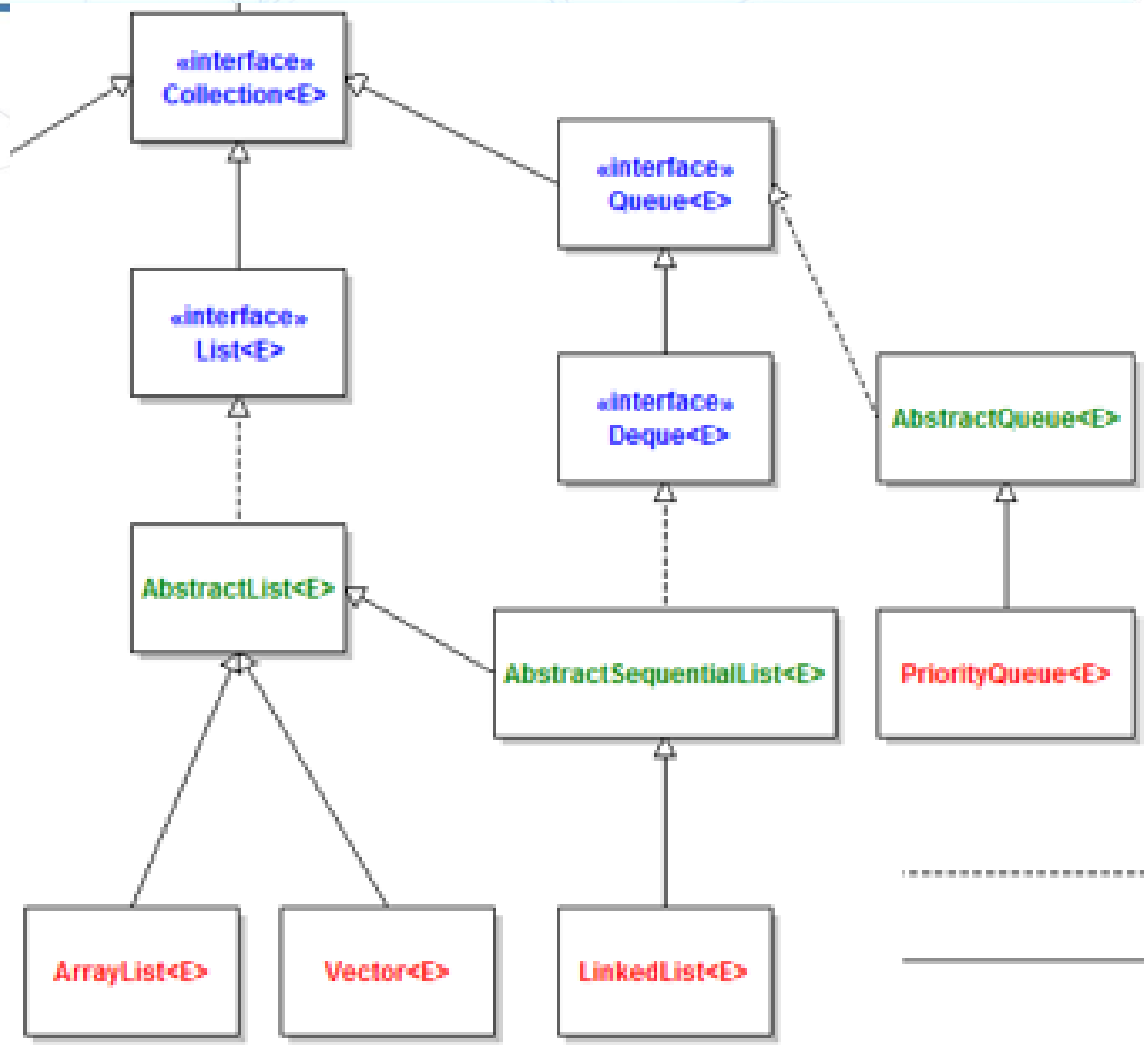
Returns obj; use as void

Returns the top element without removing it from the stack

Queue Interface

multicampus

- A queue provides temporary storage in the FIFO (First-In-First-Out) manner
- Useful for dealing with events that have to be processed in order of their arrival



Queue<E> Methods

multicampus

boolean isEmpty ()

boolean add (E obj)

E remove ()

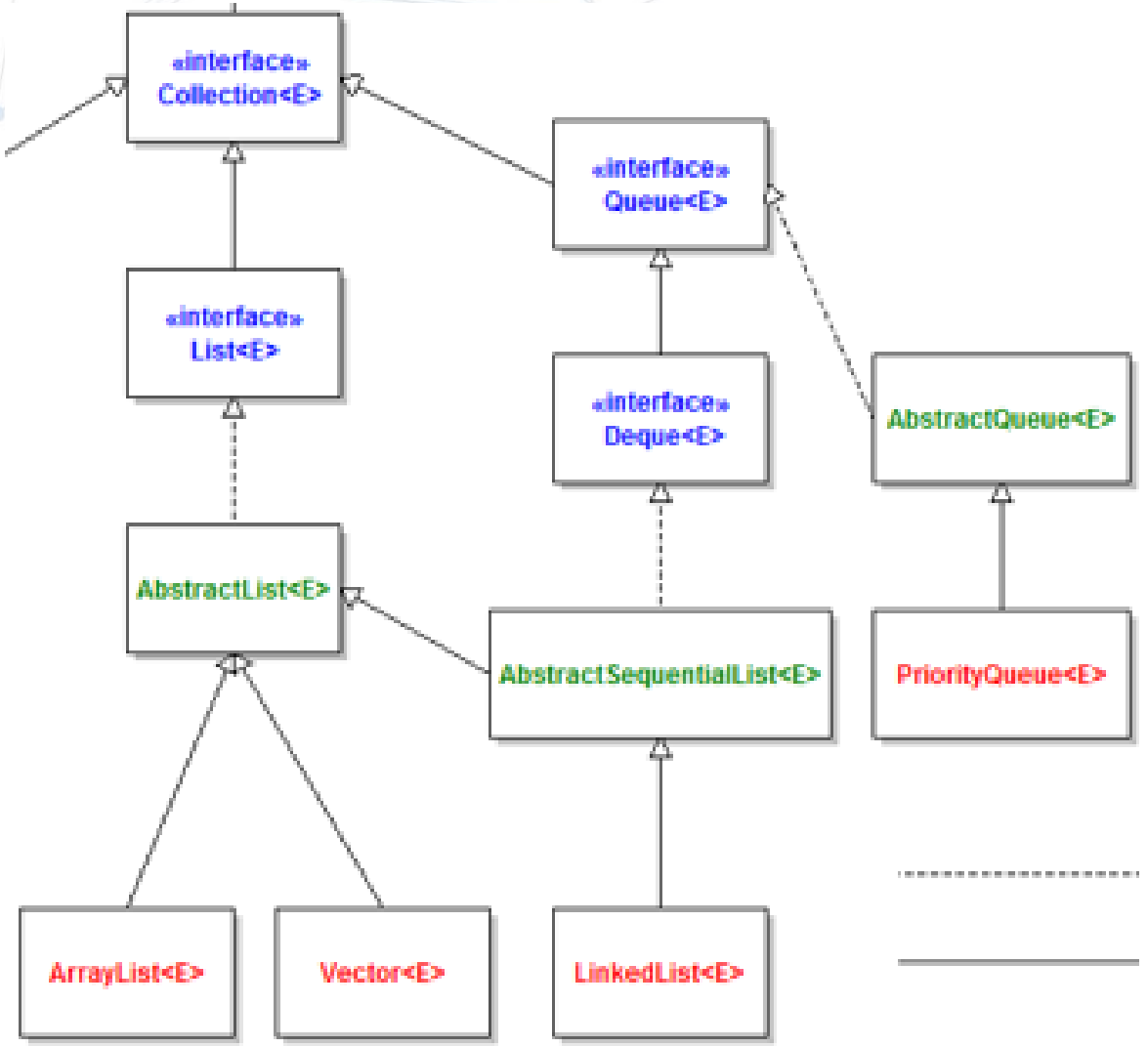
E peek ()

Returns the first element without removing it from the queue

Priority Queues

multicampus

- In a priority queue, items are processed NOT in order of arrival, but in order of priority.
- java.util:
  - *Queue* interface
  - PriorityQueue (implements *Queue*)



PriorityQueue<*E*> Class

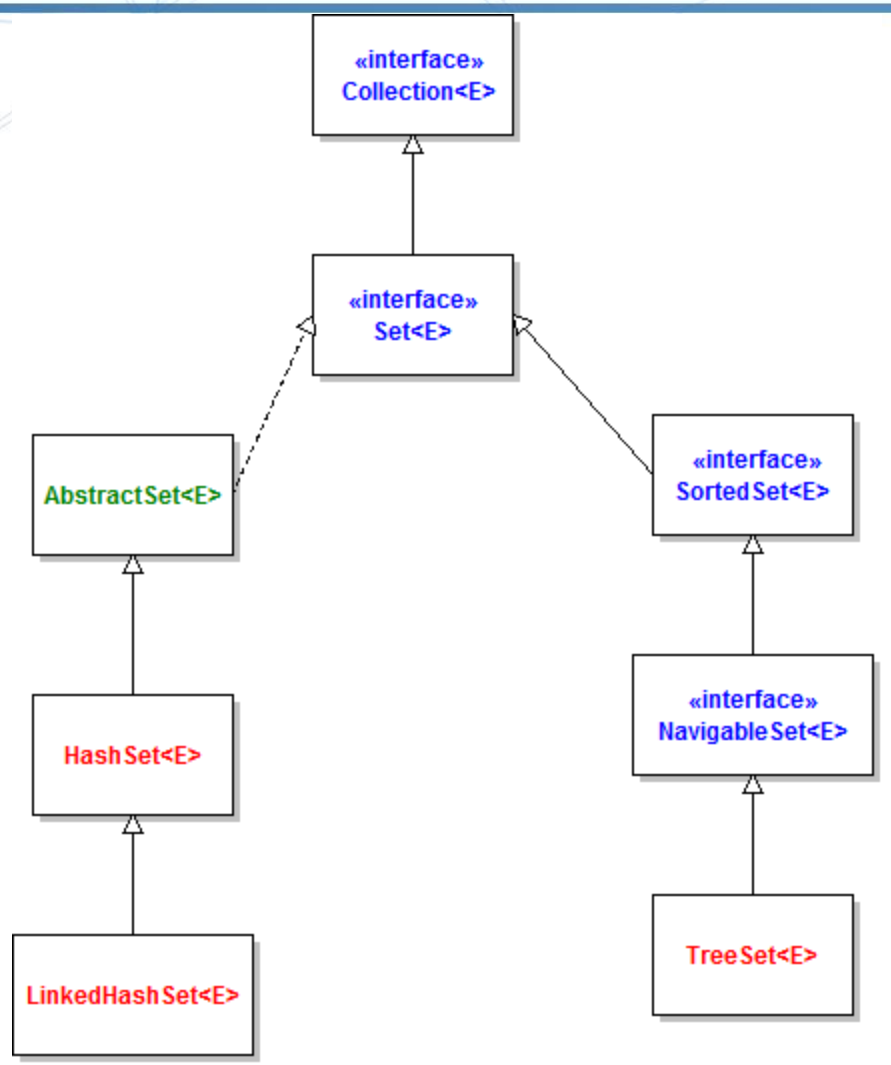
multicampus

- Works with Comparable objects (or takes a comparator as a parameter).
- The smallest item has the highest priority.
- Implements a priority queue as a *min-heap (Cấu trúc đống min – Heap Sort)*
- Both add and remove methods run in $O(\log n)$ time; peek runs in $O(1)$ time.
- Thực hành:

- Tạo ra một hàng đợi gồm các sinh viên ưu tiên (ai điểm cao nhất được ưu tiên trước).
- Tạo ra một hàng đợi gồm các nhân viên (hoten, ngaysinh,..) ưu tiên (tuổi cao nhất thì được ưu tiên trước).

Sets

- A set is a collection without duplicate values
- What is a “duplicate” depends on the implementation
- java.util:
  - *Set* interface
  - SortedSet interface
  - TreeSet
  - HashSet
-



Set<E> Methods

multicampus

Methods of *Set<E>* are the same as methods of *Collection<E>*

Set's semantics are different from *Collection* (no duplicates), but *Set* does not add any new methods.

SortedSet

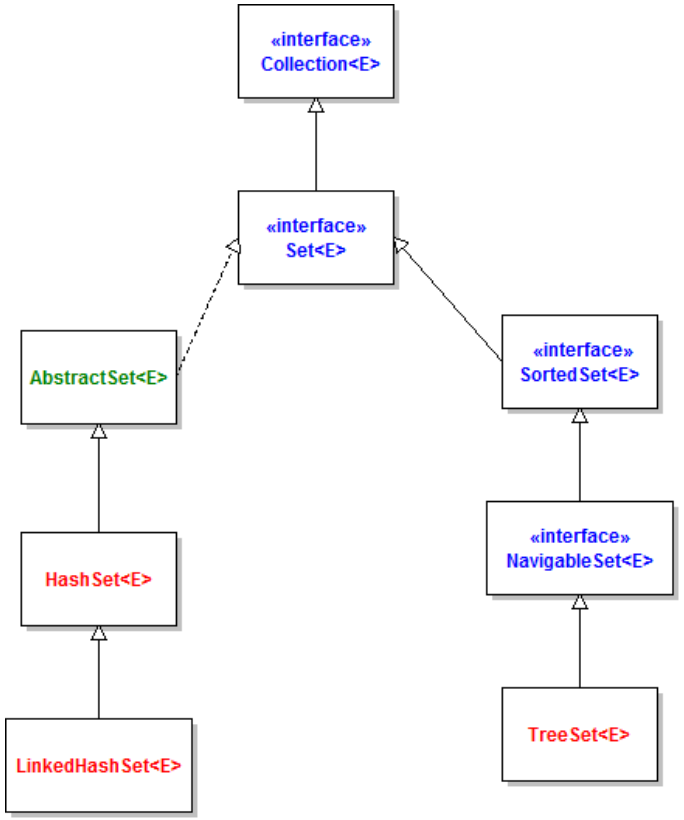
multicampus

- SortedSet kế thừa từ Set, nó hỗ trợ thao tác trên tập hợp các phần tử có thể so sánh được.
- Các đối tượng đưa vào trong một SortedSet phải cài đặt giao tiếp Comparable hoặc lớp cài đặt SortedSet phải nhận một Comparator trên kiểu của đối tượng đó.
- Một số phương thức của SortedSet<E>:
  - E first(); // lấy phần tử đầu tiên (nhỏ nhất)
  - E last(); // lấy phần tử cuối cùng (lớn nhất)
  - SortedSet subSet(E e1, E e2); // lấy một tập các phần tử nằm trong khoảng từ e1 tới e2.

TreeSet<*E*>

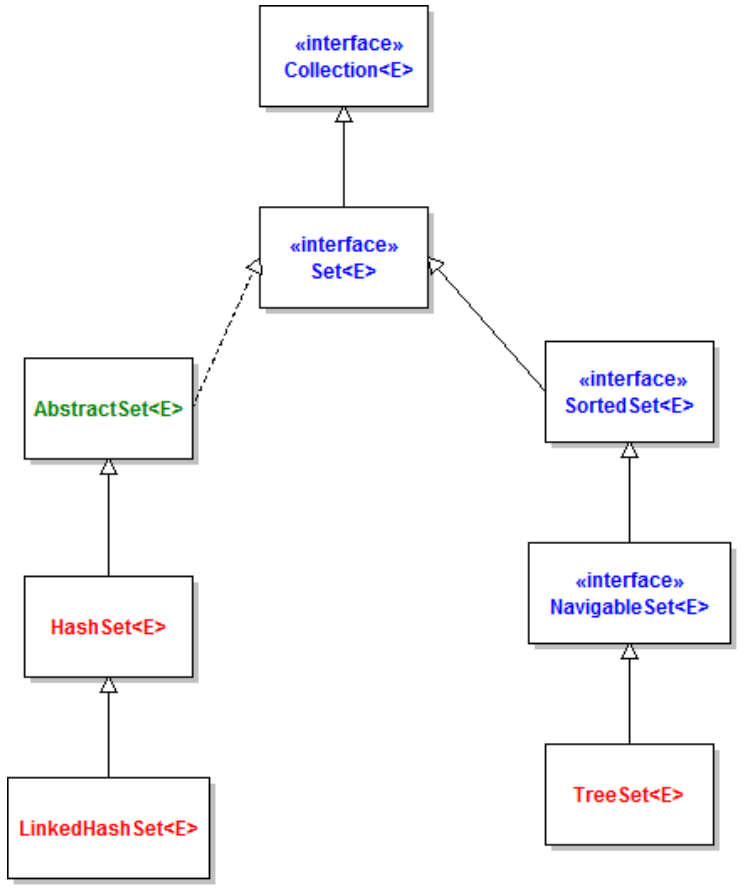
multicampus

- Works with Comparable objects (or takes a comparator as a parameter)
- Implements a set as a *Binary Search Tree (Red Black Tree)*
- contains, add, and remove methods run in *O*(log *n*) time
- Iterator returns elements in ascending order



HashSet<*E*>

- Works with objects for which **reasonable** hashCode and equals methods are defined; hashCode method = hash function; equals method is used for finding object.
- Implements a set as a *hash table*
- contains, add, and remove methods run in *O*(1) time
- Iterator returns elements in no particular order

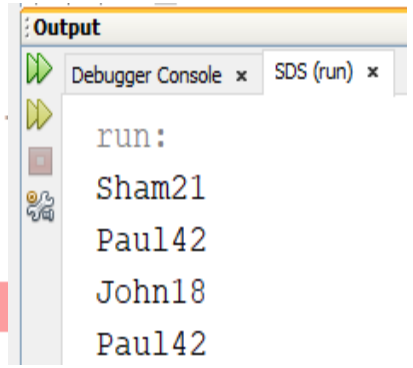


equals() & hashCode() with HashSet

multicampus

```
static class Human {
    Integer age;
    String name;
    Human(int age, String name) {
            this.age=age; this.name=name;
    }
    @Override
    public String toString() {
        return name+age.toString();
    }
}
```

```
Human human1 = new Human(21, "Sham");
Human human2 = new Human(42, "Paul");
Human human3 = new Human(18, "John");
Set<Human> hashSet = new HashSet<Human>();
hashSet.add(human1);
hashSet.add(human2);
hashSet.add(human3);
// Below code creates a new object  of 'Paul'.
Human human4 = new Human(42, "Paul");
hashSet.add(human4);
for (Human h:hashSet ){
    System.out.println(h.toString());
}
```

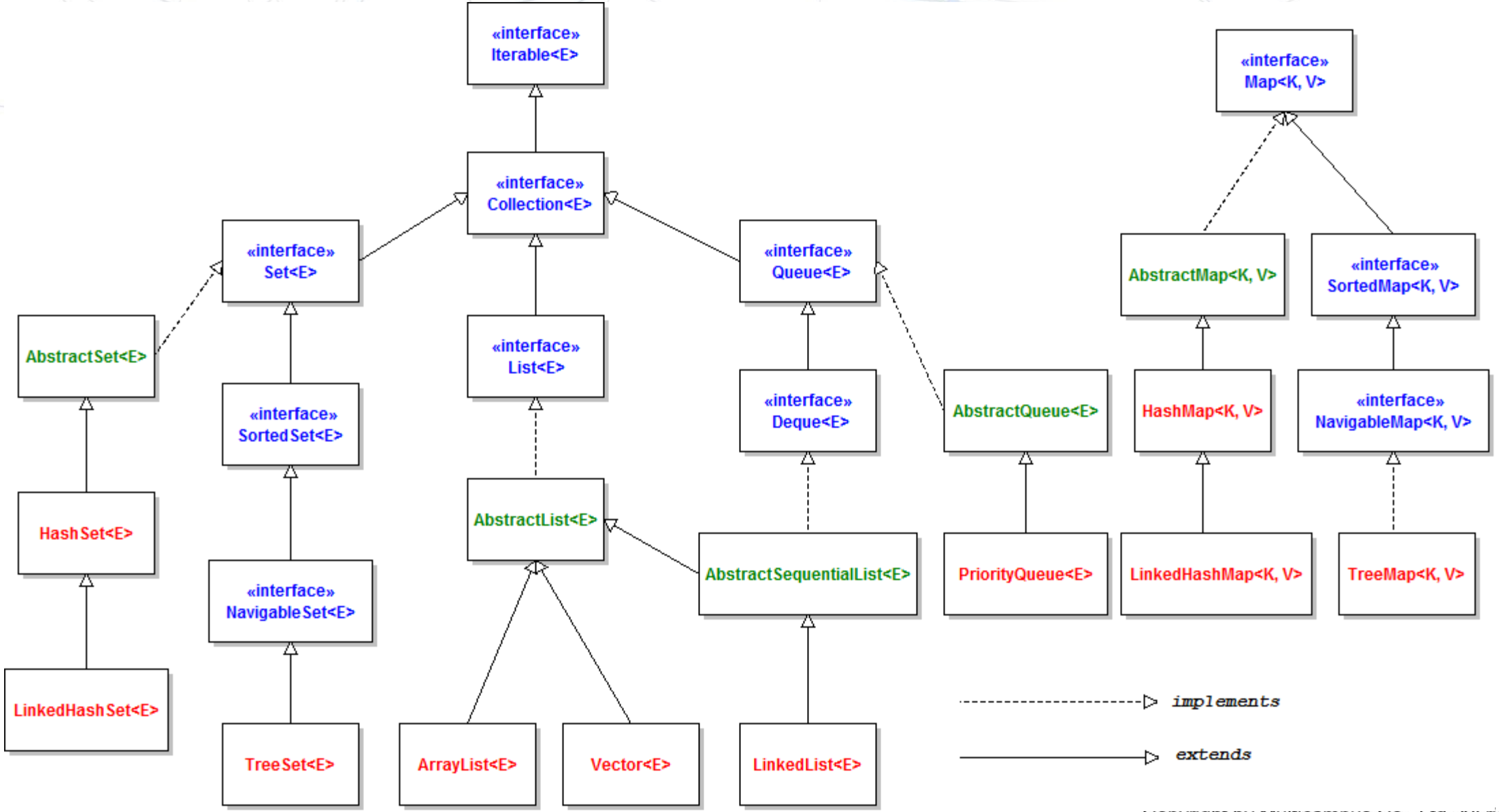


- **How does a Set prevents duplicate entries:**

The Solution is, we have to override hashCode() and equals() method in our class.

Map Collection

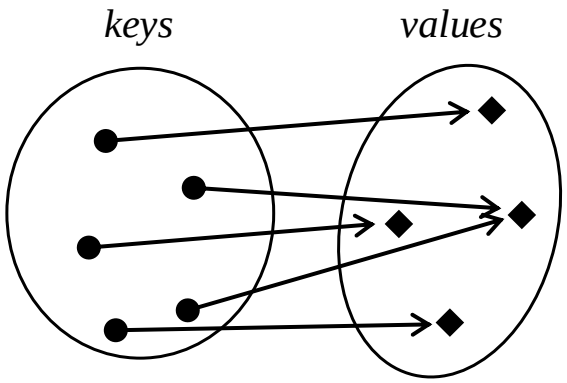
multicampus



Map Collection

multicampus

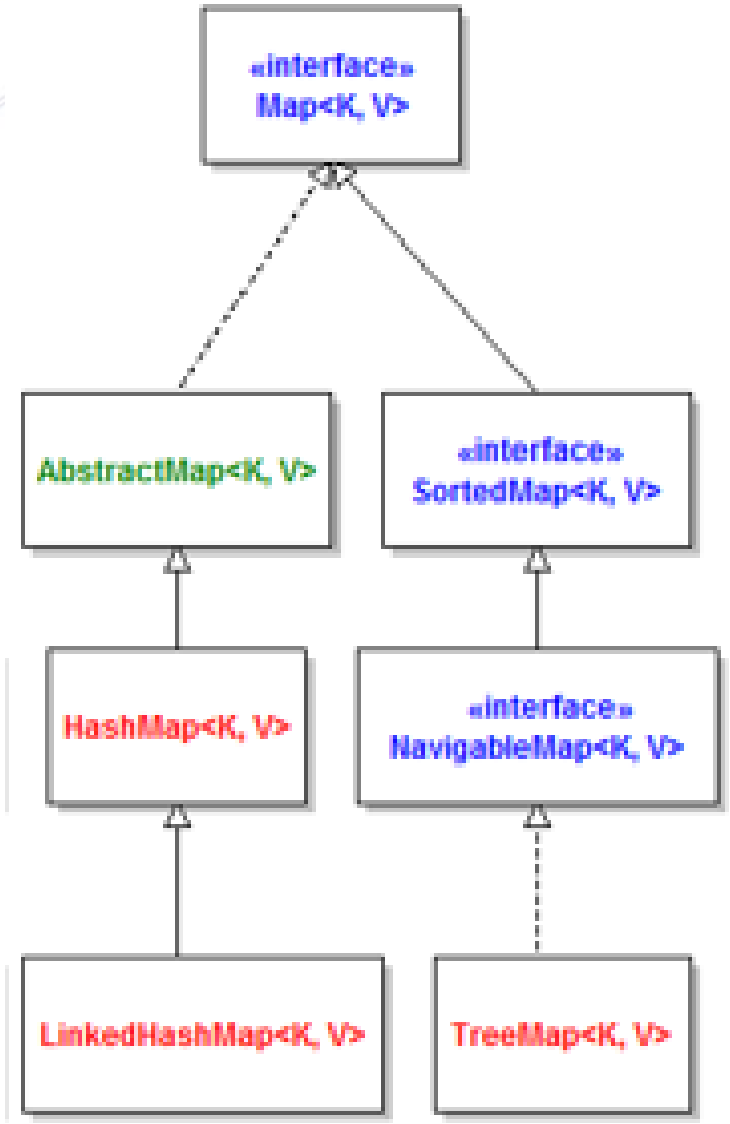
- A *map* is not a collection; it represents a correspondence between a set of keys and a set of values
- Only one value can correspond to a given key; several keys can be mapped onto the same value



Maps (cont'd)

multicampus

- java.util:
  - *Map* interface
  - SortedMap interface
  - TreeMap
  - HashMap

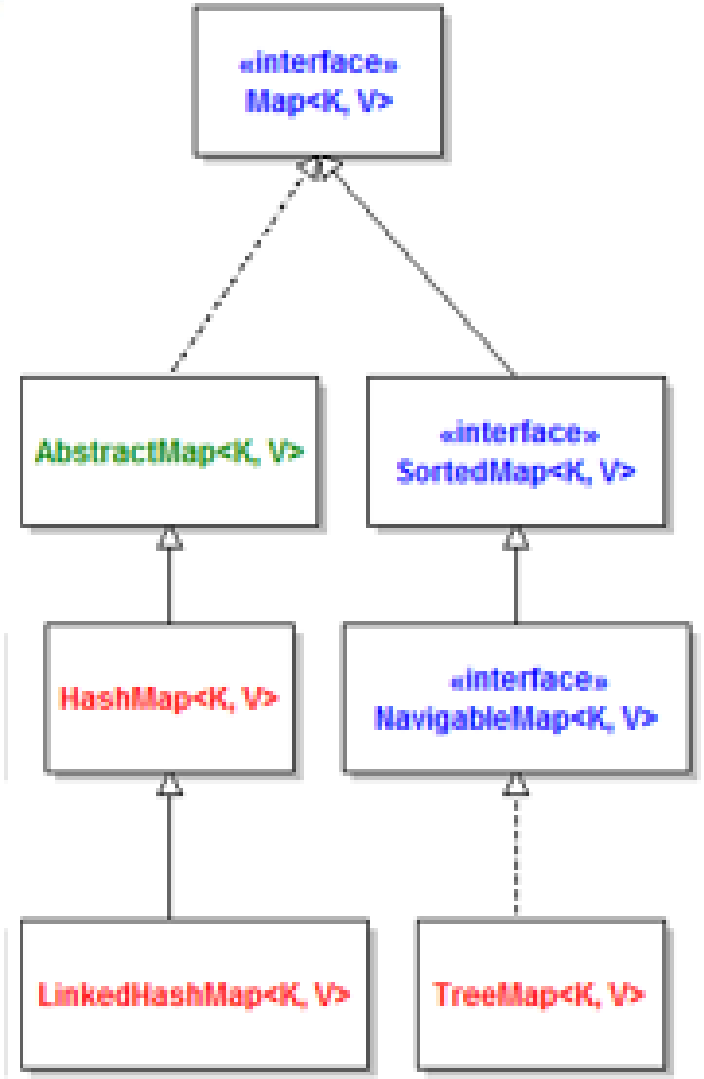


Map<K, V> Methods

multicampus

```
boolean isEmpty ()
int size ()
V get (K key)
V put (K key, V value)
V remove (K key)
boolean containsKey (Object key)
Set<K> keySet ()
```

Returns the set of all keys



TreeMap<*K*,*V*>

multicampus

- Works with Comparable keys (or takes a comparator as a parameter)
- Implements the key set as a *Binary Search Tree* (Red Black Tree)
- containsKey, get (tìm kiếm theo kiểu khoá-giá trị), and put methods run in *O*(log *n*) time

HashMap<*K*,*V*>

multicampus

- Works with keys for which reasonable hashCode and equals methods are defined
- Implements the key set as a *hash table*
- containsKey, get (key-value), and put methods run in *O*(1) time

Methods: hashCode and equals

multicampus

- HashMap and HashSet use the hashcode value of an object to find out how the object would be stored in the collection, and subsequently hashcode is used to help locate the object in the collection.
- HashMap and HashSet work as follows:
  - First, find out the right bucket using hashCode().
  - Secondly, search the bucket for the right element using equals()

Methods: hashCode and equals

multicampus

- **Case 1: Overriding both equals(Object) and hashCode() method**

```
Employee g1 = new Employee("aditya", 1);
Employee g2 = new Employee("aditya", 1);

Map<Employee, String> map = new HashMap<Employee, String>();
map.put(g1, "CSE");
map.put(g2, "IT");

for (Employee geek : map.keySet()) {
    System.out.println(map.get(geek).toString());
}
```

- **Output:**

IT

Methods: hashCode and equals

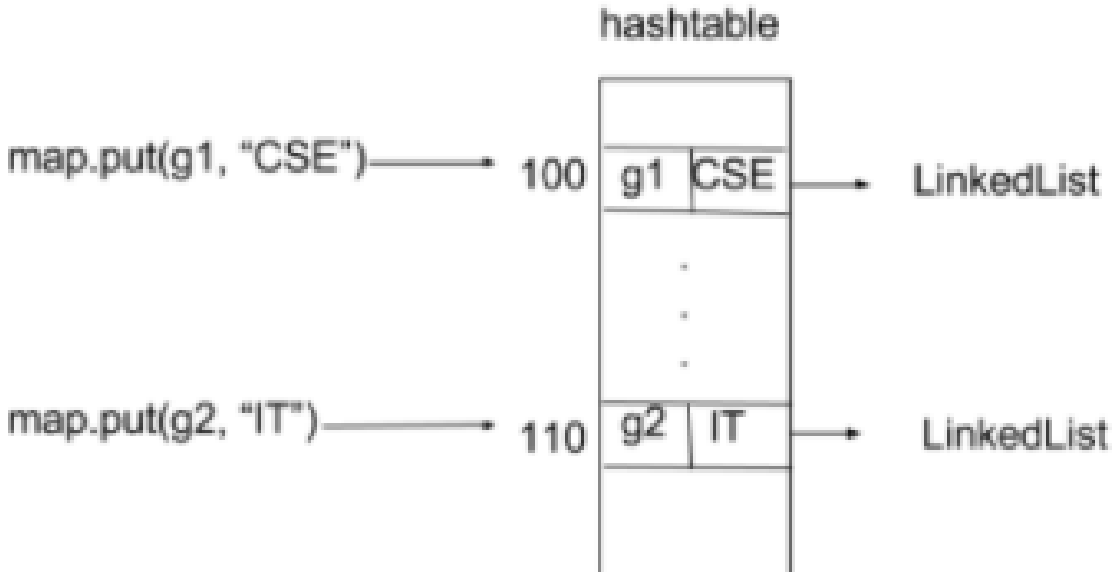
multicampus

- **Case 2 : Overriding only the equals(Object) method**

```
Employee g1 = new Employee("aditya", 1);
Employee g2 = new Employee("aditya", 1);

Map<Employee, String> map = new HashMap<Employee, String>();
map.put(g1, "CSE");
map.put(g2, "IT");

for (Employee geek : map.keySet()) {
    System.out.println(map.get(geek).toString());
}
```



- **Output:**

CSE

IT

Methods: hashCode and equals

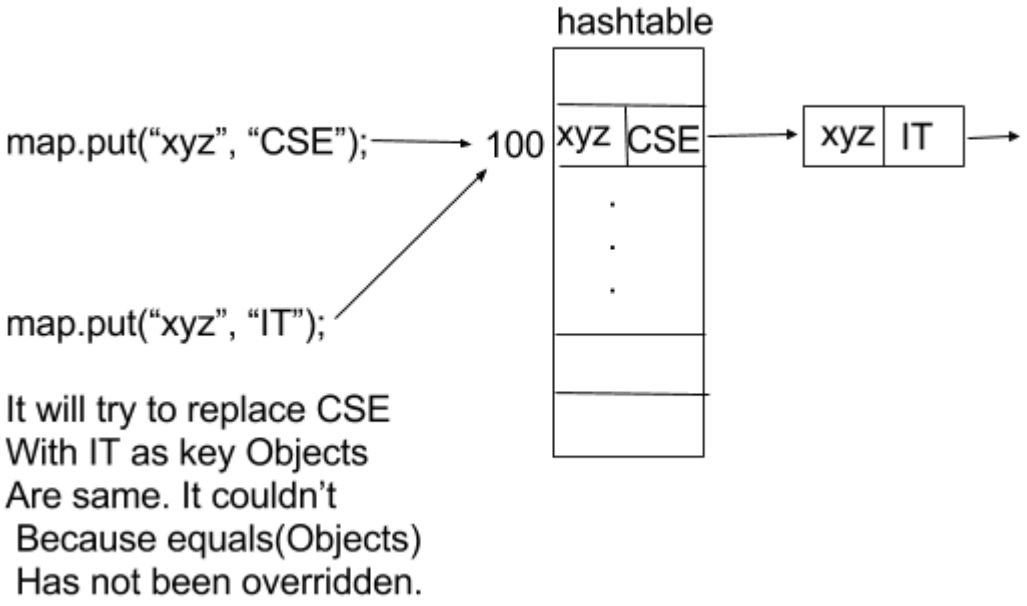
multicampus

- **Case 3: Overriding only hashCode() method**

```
Employee g1 = new Employee("aditya", 1);
Employee g2 = new Employee("aditya", 1);

Map<Employee, String> map = new HashMap<Employee, String>();
map.put(g1, "CSE");
map.put(g2, "IT");

for (Employee geek : map.keySet()) {
    System.out.println(map.get(geek).toString());
}
```



- **Output:**

CSE

IT

Case Study: Stock Exchange (giao dịch chứng khoán)

multicampus

- Implements a toy stock exchange (giao dịch chứng khoán nhỏ)
- Can be structured as a team development project
- Uses TreeSet, TreeMap, HashMap, Queue, and PriorityQueue classes

SafeTrade Structural Design

multicampus

Data	***interface*** **=> class**
Registered traders	*Map* => TreeMap<String, Trader>
Logged-in traders	*Set* => TreeSet<Trader>
Mailbox for each trader	*Queue* => LinkedList<String>
Listed stocks	*Map* => HashMap<String, Stock>
Sell orders for **each stock**	*Queue* => PriorityQueue<TradeOrder> (with ascending price comparator)
Buy orders for **each stock**	*Queue* => PriorityQueue<TradeOrder> (with descending price comparator)

Case Study: Stock Exchange (giao dịch chứng khoán)

multicampus

- Implements a toy stock exchange (giao dịch chứng khoán nhỏ)
- Can be structured as a team development project
- Uses TreeSet, TreeMap, HashMap, Queue, and PriorityQueue classes

Collections and Generics Best Practices

multicampus

1. Choosing the right collections

- To know which kind of collection (List, Set, Map, Queue, etc) is appropriate to solve the problem, you should figure out the characteristics and behaviors of each and the differences among them.
- Basically, you decide to choose a collection by answering the following questions:
  - Does it allow duplicate elements?
  - Does it allow accessing elements by index?
  - Does it offer fast adding and fast removing elements?

multicampus

2. Always using interface type when declaring a collection

```
1  List<String> listNames = new ArrayList<String>();   // (1)
```

instead of:

```
1  ArrayList<String> listNames = new ArrayList<String>();  // (2)
```

- What's the difference between (1) and (2)?

In (1), the type of the variable listNames is List, and in (2) listNames has type of ArrayList. By declaring a collection using an interface type, the code would be more flexible as you can change the concrete implementation easily when needed, for example: Có thể đổi từ ArrayList sang LinkedList khi cần.

multicampus

2. Always using interface type when declaring a collection

The flexibility of using interface type for a collection is more visible in case of method's parameters. Consider the following method:

```
1  public void foo(Set<Integer> numbers) {
2  }
```

Here, by declaring the parameter `numbers` as of type `Set`, the client code can pass any implementations of `Set` such as `HashSet` or `TreeSet`:

```
1  foo(treeSet);
2  foo(hashSet);
```

This makes your code more flexible and more abstract.

In contrast, if you declare the parameter `numbers` as of type `HashSet`, the method cannot accept anything except `HashSet` (and its subtypes), which makes the code less flexible.

Collections and Generics Best Practices

multicampus

2. Always using interface type when declaring a collection

It's also recommended to use interface as return type of a method that returns a collection, for example:

```
1  public Collection listStudents() {
2      List<Student> listStudents = new ArrayList<Student>();
3
4      // add students to the list
5
6      return listStudents;
7  }
```

This definitely increases the flexibility of the code, as you can change the real implementation inside the method without affecting its client code.

multicampus

3. Use generic type and diamond operator

The <> is informally called the diamond operator (Java 1.7). This operator is quite useful. Imagine if you have to declare a collection like this:

Without the diamond operator, you have to repeat the same declaration twice:

```
Map< Integer, Map<String, Student> > map
        = new HashMap< Integer, Map<String, Student> >();
```

So the diamond operator saves you:

```
Map<Integer, Map<String, Student>> map = new HashMap<>();
```

Collections and Generics Best Practices

multicampus

4. Specify initial capacity of a collection if possible

Ex: List<String> listNames = new ArrayList <String>(5000);

5. Prefer isEmpty() over size()

Avoid checking the emptiness of a collection like this:

```
if (listStudents.size() > 0) {
   // dos something if the list is not empty
}
```

Instead, you should use the isEmpty() method:

```
if (!listStudents.isEmpty()) {
   // dos something if the list is not empty
}
```

Collections and Generics Best Practices

multicampus

6. Do not return null in a method that returns a collection

A null value should not be used to indicate no result. The best practice is, returning an empty collection to indicate no result.

Ex:

```
1  public List<Student> findStudents(String className) {
2      List<Student> listStudents = null;
3
4      if (//students are found//) {
5          // add students to the lsit
6      }
7
8      return listStudents;
9  }
```

Dòng 2 cần thay bằng List<Student> listStudents = new ArrayList<>;

Collections and Generics Best Practices

multicampus

7. Do not use the classic for loop

Không nên sử dụng dạng classic:

```
1  for (int i = 0; i < listStudents.size(); i++) {
2      Student aStudent = listStudents.get(i);
3
4      // do something with aStudent
5  }
```

Có thể sử dụng dạng:

```
1  Iterator<Student> iterator = listStudents.iterator();
2
3  while (iterator.hasNext()) {
4      Student nextStudent = iterator.next();
5
6      // do something with nextStudent
7  }
```

Tốt nhất là dạng:

```
1  for (Student aStudent : listStudents) {
2      // do something with aStudent
3  }
```

THANK YOU

multicampus