Advanced Programming Language

MSc. Nguyen Cao Dat
dat@hcmut.edu.vn

Module III

DATA STRUCTURES IN JAVA

Content

- The limitations of arrays
- Java Collection Framework hierarchy
- Set interface, HashSet, and TreeSet
- List interface, ArrayList, and LinkedList
- Vector and Stack
- Map, HashMap, and TreeMap
- Collections and Arrays classes

Limitations of arrays

- Once an array is created, its size cannot be altered.
- Array provides inadequate support for inserting, deleting, sorting, and searching operations.

Content

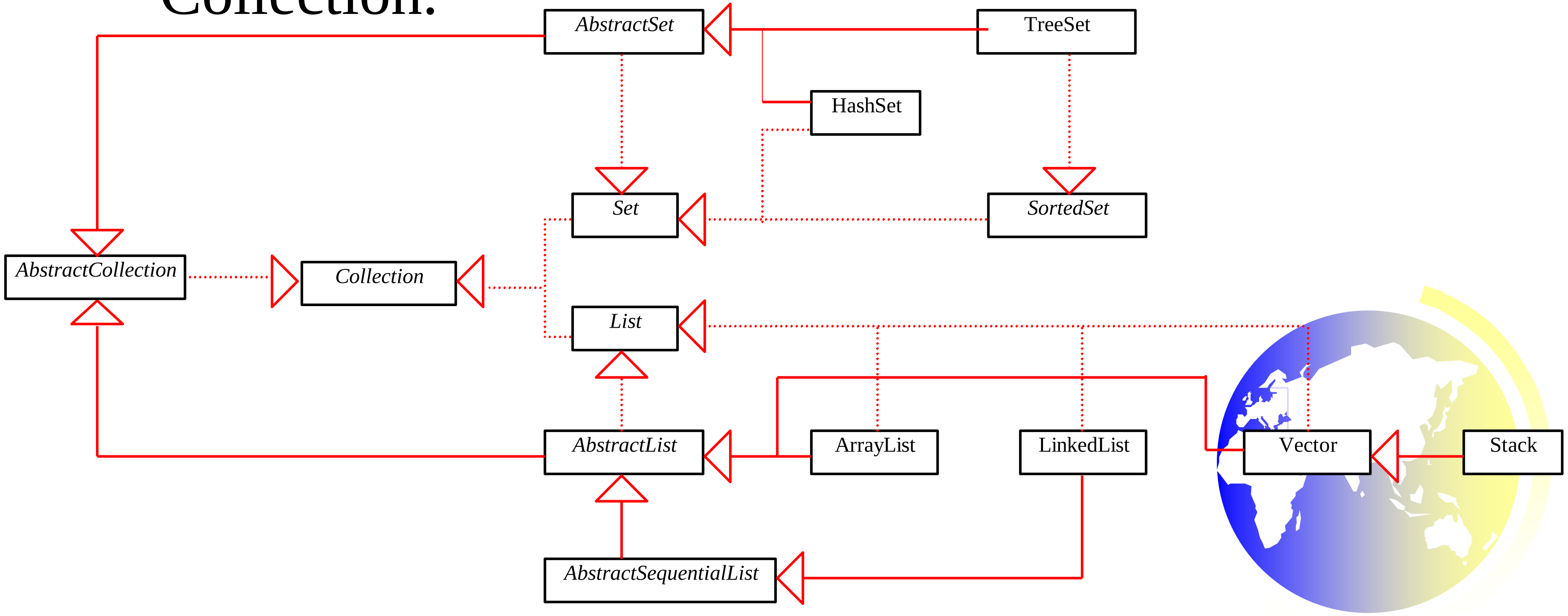
- The limitations of arrays
- Java Collection Framework hierarchy
- Set interface, HashSet, and TreeSet
- List interface, ArrayList, and LinkedList
- Vector and Stack
- Map, HashMap, and TreeMap
- Collections and Arrays classes

Java Collection Framework hierarchy

- A collection is an object that represents a group of objects, often referred to as elements.
- The Java Collections Framework supports two types of collections, named collections and maps.

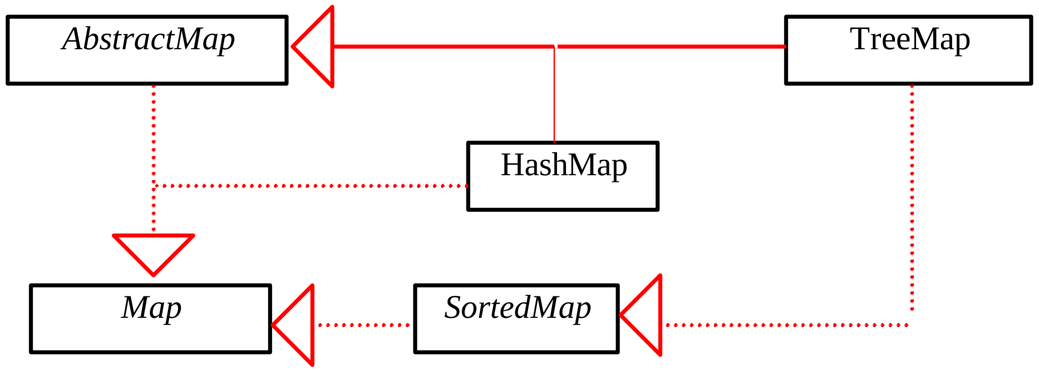
Java Collection Framework hierarchy, cont.

- A collection can be a set or a list, defined in the interfaces Set and List, which are subinterfaces of Collection.



Java Collection Framework hierarchy, cont.

- An instance of Map represents a group of objects, each of which is associated with a key.
- You can get the object from a map using a key, and you have to use a key to put the object into the map.



The Collection Interface

Collection

+add(element: Object): boolean
+addAll(collection: Collection): boolean
+clear(): void
+contains(elment: Object): boolean
+containsAll(collection: Collection):boolean
+equals(object: Object): boolean
+hashcode(): int
+iterator(): Iterator
+remove(element: Object): boolean
+removeAll(collection: Collection): boolean
+retainAll(collection: Collection): boolean
+size(): int
+toArray(): Object[]
+toArray(array: Object[]): Object[]

- The Collection interface is the root interface for storing and processing a collection of objects.

Questions?

Content

- The limitations of arrays
- Java Collection Framework hierarchy
- Set interface, HashSet, and TreeSet
- List interface, ArrayList, and LinkedList
- Vector and Stack
- Map, HashMap, and TreeMap
- Collections and Arrays classes

The Set Interface

- The Set interface extends the Collection interface.
- It does not introduce new methods or constants, but it stipulates that an instance of Set contains no duplicate elements.
- The concrete classes that implement Set must ensure that no duplicate elements can be added to the set.
- That is no two elements e1 and e2 can be in the set such that e1.equals(e2) is true.

The AbstractSet Class

- The AbstractSet class is a convenience class that extends AbstractCollection and implements Set.
- The AbstractSet class provides concrete implementations for the equals method and the hashCode method.
- The hash code of a set is the sum of the hash code of all the elements in the set.
- Since the size method and iterator method are not implemented in the AbstractSet class, AbstractSet is an abstract class.

The HashSet Class

- The HashSet class is a concrete class that implements Set.
- It can be used to store duplicate-free elements.
- For efficiency, objects added to a hash set need to implement the hashCode method in a manner that properly disperses the hash code.

Example: Using HashSet and Iterator

This example creates a hash set filled with strings, and uses an iterator to traverse the elements in the list.

```
 // Create a hash set
Set<String> set = new HashSet<>();

// Add strings to the set
set.add("London");
set.add("Paris");

// Display the elements in the hash set
for (String s: set) {
  System.out.print(s.toUpperCase() + " ");
}
```

The SortedSet Interface and the TreeSet Class

- SortedSet is a subinterface of Set, which guarantees that the elements in the set are sorted.
- TreeSet is a concrete class that implements the SortedSet interface.
- You can use an iterator to traverse the elements in the sorted order. The elements can be sorted in two ways.

The SortedSet Interface and the TreeSet Class, cont.

- One way is to use the <u>Comparable</u> interface.
- The other way is to specify a comparator for the elements in the set if the class for the elements does not implement the <u>Comparable</u> interface, or you don’t want to use the <u>compareTo</u> method in the class that implements the <u>Comparable</u> interface.

Example: Using TreeSet to Sort Elements in a Set

- This example creates a hash set filled with strings, and then creates a tree set for the same strings.
- The strings are sorted in the tree set using the compareTo method in the Comparable interface.
- The example also creates a tree set of geometric objects.
- The geometric objects are sorted using the compare method in the Comparator interface.

Questions?

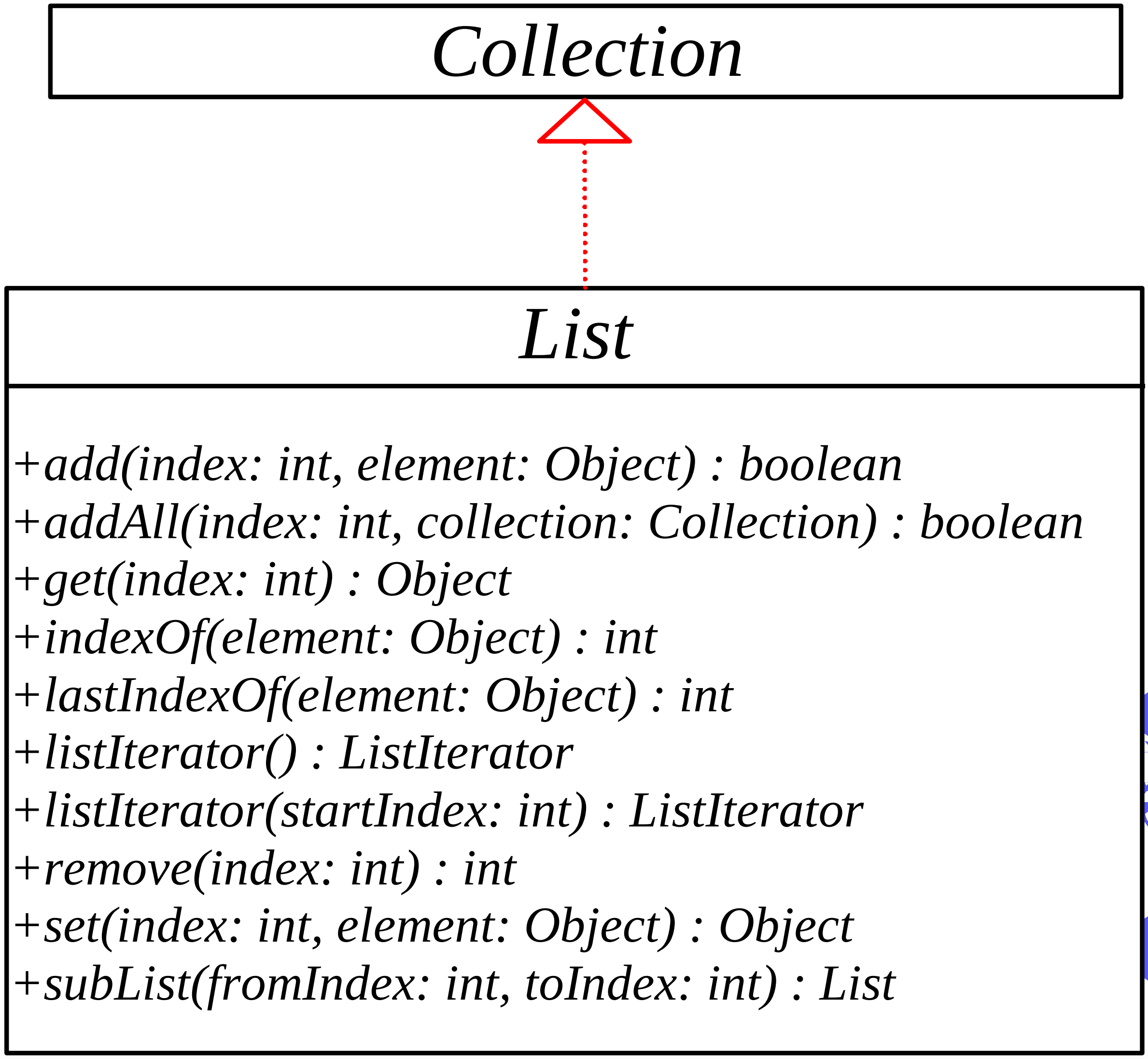
Content

- The limitations of arrays
- Java Collection Framework hierarchy
- Set interface, HashSet, and TreeSet
- List interface, ArrayList, and LinkedList
- Vector and Stack
- Map, HashMap, and TreeMap
- Collections and Arrays classes

The List Interface

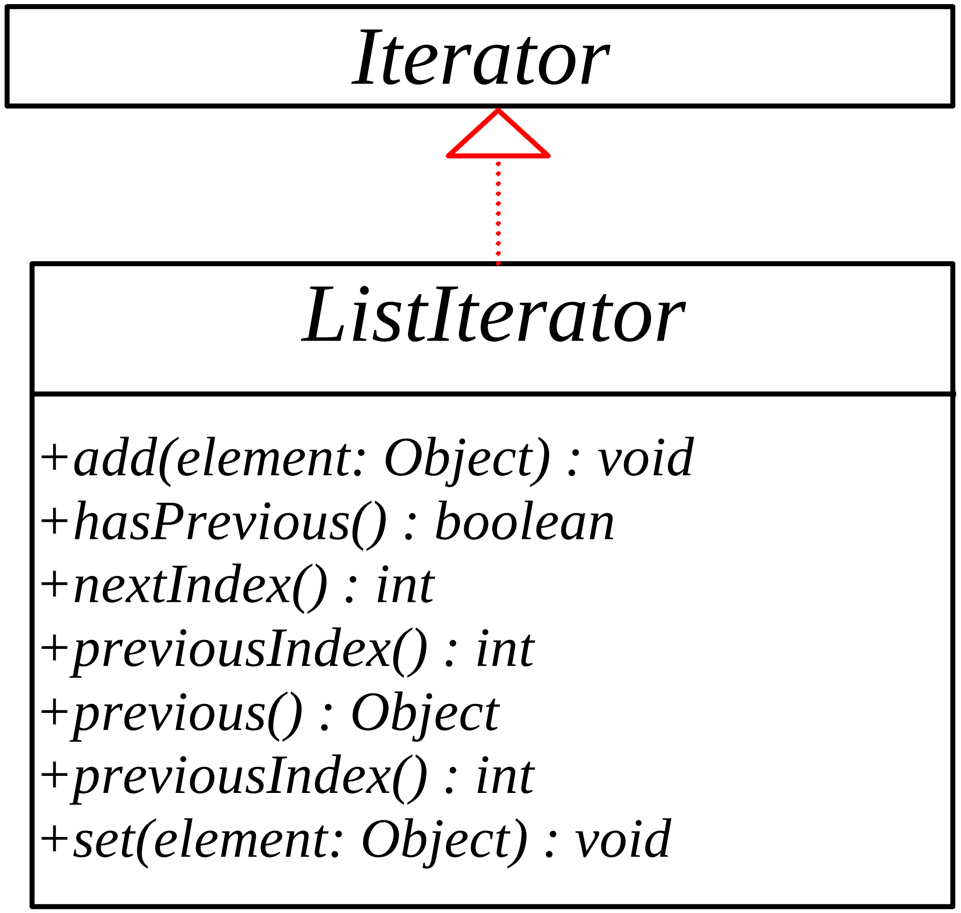
- A set stores non-duplicate elements.
- To allow duplicate elements to be stored in a collection, you need to use a list.
- A list can not only store duplicate elements, but can also allow the user to specify where the element is stored.
- The user can access the element by index.

The List Interface, cont.



+add(index: int, element: Object) : boolean
+addAll(index: int, collection: Collection) : boolean
+get(index: int) : Object
+indexOf(element: Object) : int
+lastIndexOf(element: Object) : int
+listIterator() : ListIterator
+listIterator(startIndex: int) : ListIterator
+remove(index: int) : int
+set(index: int, element: Object) : Object
+subList(fromIndex: int, toIndex: int) : List

The List Iterator



ArrayList and LinkedList

- The ArrayList class and the LinkedList class are concrete implementations of the List interface. Which of the two classes you use depends on your specific needs.
- If you need to support random access through an index without inserting or removing elements from any place other than the end, ArrayList offers the most efficient collection.
- However, your application requires the insertion or deletion of elements from any place in the list, you should choose LinkedList.
- A list can grow or shrink dynamically. An array is fixed once it is created. If your application does not require insertion or deletion of elements, the most efficient data structure is the array.

Example: Using ArrayList and LinkedList

- This example creates an array list, and inserts new elements into the specified location in the list, inserts and removes elements from the list.
- The example also creates a linked list from the array list, inserts and removes the elements from the list.

Questions?

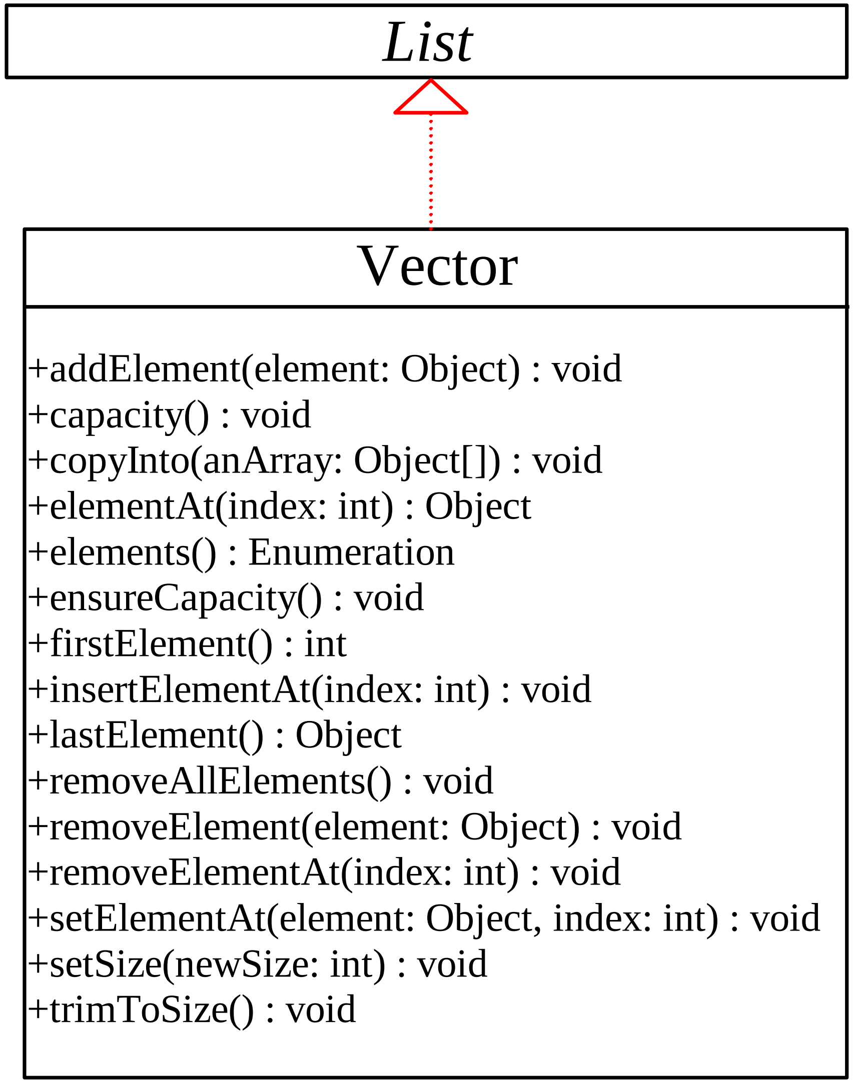
Content

- The limitations of arrays
- Java Collection Framework hierarchy
- Set interface, HashSet, and TreeSet
- List interface, ArrayList, and LinkedList
- Vector and Stack
- Map, HashMap, and TreeMap
- Collections and Arrays classes

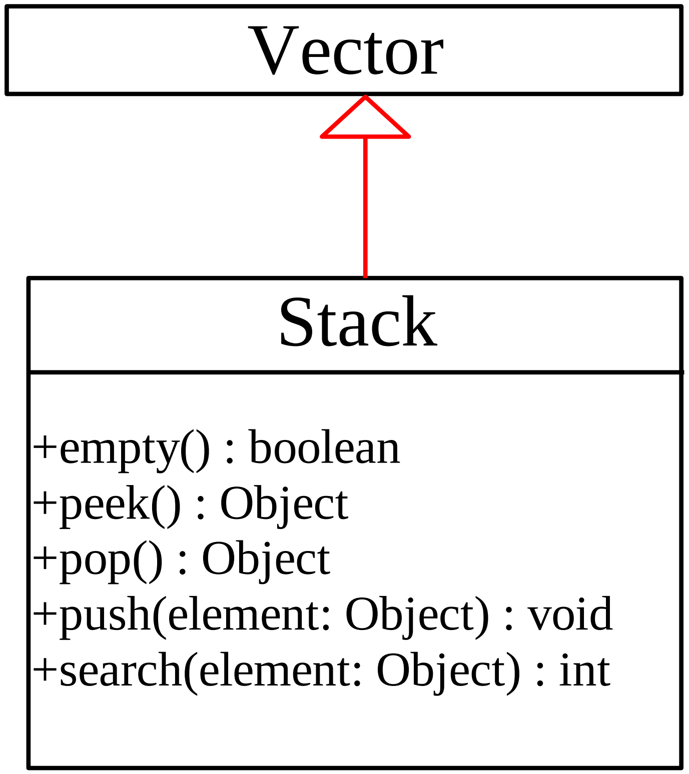
The Vector Class

- Vector is the same as ArrayList, except that Vector contains the synchronized methods for accessing and modifying the vector.
- If synchronization is required, you can use the synchronized versions of the collection classes.
- These classes are introduced later in the section, “The Collections Class.”

The Vector Class, cont.


List
Vector
+addElement(element: Object) : void
+capacity() : void
+copyInto(anArray: Object[]) : void
+elementAt(index: int) : Object
+elements() : Enumeration
+ensureCapacity() : void
+firstElement() : int
+insertElementAt(index: int) : void
+lastElement() : Object
+removeAllElements() : void
+removeElement(element: Object) : void
+removeElementAt(index: int) : void
+setElementAt(element: Object, index: int) : void
+setSize(newSize: int) : void
+trimToSize() : void

The Stack Class



The Stack class represents a last-in-first-out stack of objects. The elements are accessed only from the top of the stack. You can retrieve, insert, or remove an element from the top of the stack.

Example: Using Vector and Stack

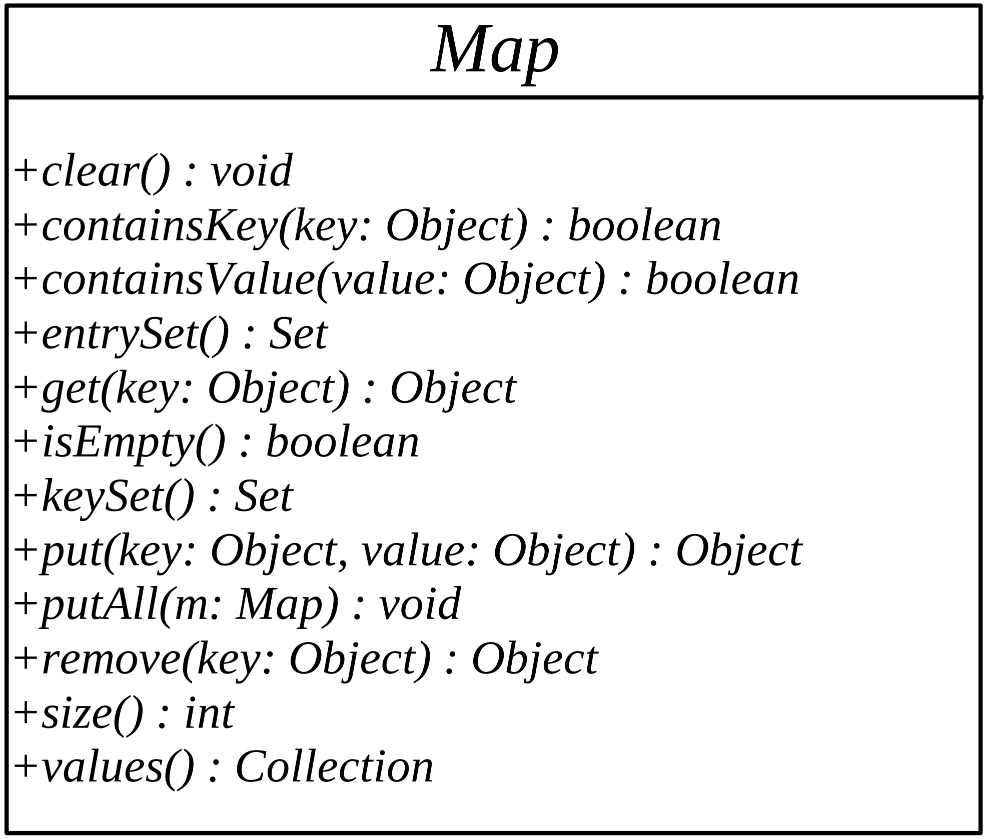
- This example presents two programs using a vector and a stack instead of an array, respectively.
- The program reads student scores from the keyboard, stores the scores in the vector, finds the best scores, and then assigns grades for all the students. A negative score signals the end of the input.

Questions?

Content

- The limitations of arrays
- Java Collection Framework hierarchy
- Set interface, HashSet, and TreeSet
- List interface, ArrayList, and LinkedList
- Vector and Stack
- Map, HashMap, and TreeMap
- Collections and Arrays classes

The Map Interface



Map

```
+clear() : void
+containsKey(key: Object) : boolean
+containsValue(value: Object) : boolean
+entrySet() : Set
+get(key: Object) : Object
+isEmpty() : boolean
+keySet() : Set
+put(key: Object, value: Object) : Object
+putAll(m: Map) : void
+remove(key: Object) : Object
+size() : int
+values() : Collection
```

The Map interface maps keys to the elements. The keys are like indexes. In List, the indexes are integer. In Map, the keys can be any objects.

HashMap and TreeMap

- The HashMap and TreeMap classes are two concrete implementations of the Map interface.
- The HashMap class is efficient for locating a value, inserting a mapping, and deleting a mapping.
- The TreeMap class, implementing SortedMap, is efficient for traversing the keys in a sorted order.

Example: Using HashMap and TreeMap

- This example creates a hash map that maps names to ages.
- The program first creates a hash map with name as its key and age as its value.
- The program then creates a tree map from the hash map, and displays the mappings in ascending order of the keys.

Content

- The limitations of arrays
- Java Collection Framework hierarchy
- Set interface, HashSet, and TreeSet
- List interface, ArrayList, and LinkedList
- Vector and Stack
- Map, HashMap, and TreeMap
- Collections and Arrays classes

The Collections Class

The Collections class contains various static methods for operating on collections and maps, for creating synchronized collection classes, and for creating read-only collection classes.

Collections

+binarySearch(list: List, key: Object) : int
+binarySearch(list: List, key: Object, c: Comparator) : int
+copy(src: List, des: List) : void
+enumeration(c: final Collection) : Enumeration
+fill(list: List, o: Object) : void
+max(c: Collection) : Object
+max(c: Collection, c: Comparator) : Object
+min(c: Collection) : Object
+min(c: Collection, c: Comparator) : Object
+nCopies(n: int, o: Object) : List
+reverse(list: List) : void
+reverseOrder() : Comparator
+shuffle(list: List) : void
+shuffle(list: List, rnd: Random) : void
+singleton(o: Object) : Set
+singletonList(o: Object) : List
+singletonMap(key: Object, value: Object) : Map
+sort(list: List) : void
+sort(list: List, c: Comparator) : void
+synchronizedCollection(c: Collection) : Collection
+synchronizedList(list: List) : List
+synchronizedMap(m: Map) : Map
+synchronizedSet(s: Set) : Set
+synchronizedSortedMap(s: SortedMap) : SortedMap
+synchronizedSortedSet(s: SortedSet) : SortedSet
+unmodifiedCollection(c: Collection) : Collection
+unmodifiedList(list: List) : List
+unmodifiedMap(m: Map) : Map
+unmodifiedSet(s: Set) : Set
+unmodifiedSortedMap(s: SortedMap) : SortedMap
+unmodifiedSortedSet(s: SortedSet) : SortedSet

Example: Using the Collections Class

- This example demonstrates using the methods in the Collections class.
- The example creates a list, sorts it, and searches for an element.
- The example wraps the list into a synchronized and read-only list.

The Arrays Class

The Arrays class contains various static methods for sorting and searching arrays, for comparing arrays, and for filling array elements. It also contains a method for converting an array to a list.

Arrays

+asList(a: Object[]) : List
+binarySearch(a: byte[],key: byte) : int
+binarySearch(a: char[], key: char) : int
+binarySearch(a: double[], key: double) : int
+binarySearch(a,: float[] key: float) : int
+binarySearch(a: int[], key: int) : int
+binarySearch(a: long[], key: long) : int
+binarySearch(a: Object[], key: Object) : int
+binarySearch(a: Object[], key: Object, c: Comparator) : int
+binarySearch(a: short[], key: short) : int
+equals(a: boolean[], a2: boolean[]) : boolean
+equals(a: byte[], a2: byte[]) : boolean
+equals(a: char[], a2: char[]) : boolean
+equals(a: double[], a2: double[]) : boolean
+equals(a: float[], a2: float[]) : boolean
+equals(a: int[], a2: int[]) : boolean
+equals(a: long[], a2: long[]) : boolean
+equals(a: Object[], a2: Object[]) : boolean
+equals(a: short[], a2: short[]) : boolean
+fill(a: boolean[], val: boolean) : void
+fill(a: boolean[], fromIndex: int, toIndex: int, val: boolean) : void

Overloaded fill method for char, byte, short, int, long, float, double, and Object.

+sort(a: byte[]) : void
+sort(a: byte[], fromIndex: int, toIndex: int) : void

Overloaded sort method for char, short, int, long, float, double, and Object.

Example: Using the Arrays Class

- This example demonstrates using the methods in the Arrays class.
- The example creates an array of int values, sorts it, searches for an element, and compares the array with another array.
- **https://www.geeksforgeeks.org/array-class-in-java/**

Questions?