CIT
5940

Java Collections

The Java Collection Interface

- The root interface in the *collection hierarchy*
- A collection represents a group of objects, which are known as its elements
- Some of the many subinterfaces & implementations:
    - List: an ordered sequence of elements
        - ArrayList, LinkedList
    - Deque: a double-ended queue
        - ArrayDeque, LinkedList (useful for stack operations)
    - Set: an unordered collection with no duplicates
        - TreeSet, HashSet

Properties of Collections

Property	Definition	Example
Ordered	Elements have "positions" or indices; user can control where to insert or retrieve an element	`add(int index, E element)` in `List` ADT
Unordered	User cannot control where to insert or retrieve elements	`add(E element)` in `Set` ADT
Sorted	Collection elements are sorted using their natural ordering (when `Comparable`) or by a *comparator*.	`SortedSet` ADT, `TreeSet`
Allow duplicates	Multiple copies of two elements that are `equals()` to each other can be stored in the same data structure	`List` ADT
No duplicates	Only one copy of an element can be stored in the data structure	`Set` ADT

The `Collection` Interface: Key Operations

- `boolean add(E e)` adds the specified element to the collection
- `boolean contains(Object o)` returns `true` if this collection contains the specified element
- `boolean remove(Object o)` removes the specified element from this collection (if present)

All return `boolean` to indicate success or not. *Why?*

The `Collection` Interface: Operations

- Operations relying on comparing elements using `equals()` or `hashCode()` methods take an object as parameter instead of a generic type.
    - Both methods are defined in the `Object` class so **everything** in Java has them.
    - You provide implementations of these methods when you define your own classes.

Ordered Data Structures

Elements have "positions" or indices; operations for inserting or retrieving elements are defined in terms of the location at which that element lives.

- Arrays are ordered: use the `[]` notation to specify indices
- Lists of all persuations (`ArrayList`, `LinkedList`, `Vector`) are ordered

Affordances & Arrays

Arrays are simple, but they offer only three real operations: `set`, `get`, and `length`

Lists in Java are built to provide easier implementation of important ordered operations (`add`, `contains`, `remove`)

- `ArrayList` is your go-to about 90% of the time. Actually more like 99%.
- `LinkedList` also exists. You probably don't want to actually use it.

Ordered ⇏ Sorted

Here's an ordered array that's not sorted:

```
[3, 1, 4, 1, 5, 9, 2]
```

Here's an ordered array that IS sorted:

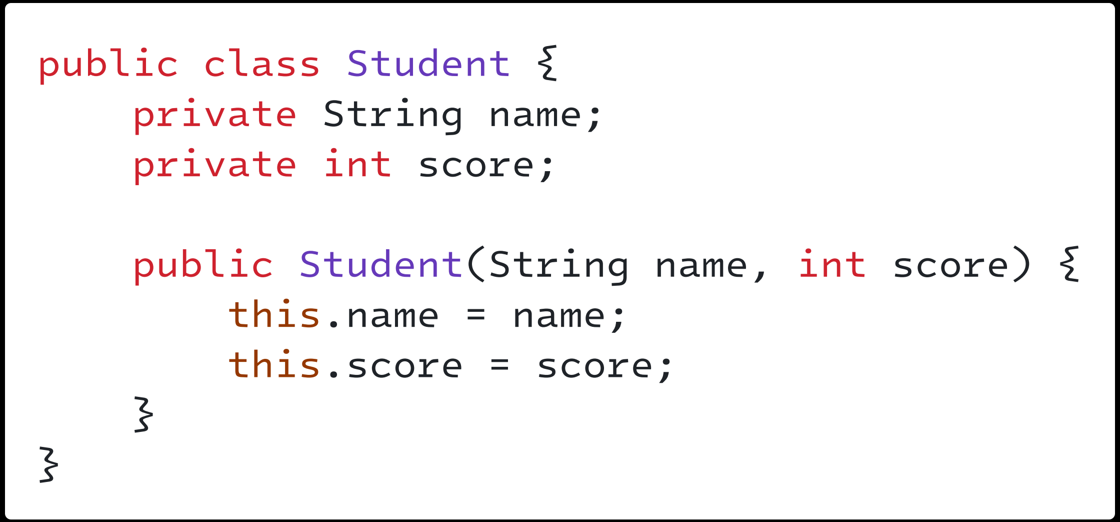
```
[1, 1, 2, 3, 4, 5, 9]
```

To turn some ordered array/list into a sorted array/list, use `Arrays.sort()`/`Collections.sort()`.

Sorting Yourself Out

What's the sorted version of this array of students?

```
[new Student("Harry", 54),
new Student("Voravich", 80),
new Student("Sid", 79)]
```



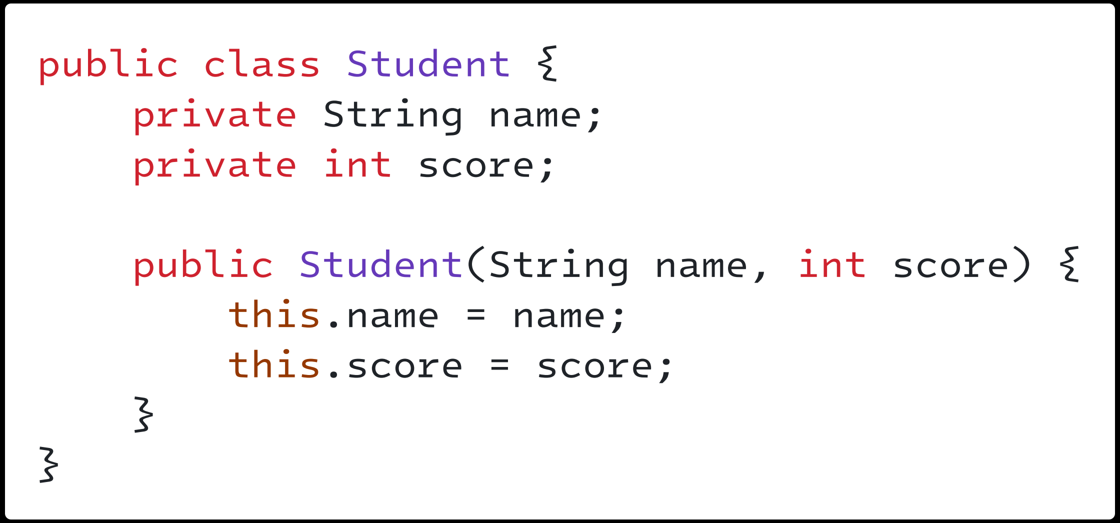
```
public class Student {
    private String name;
    private int score;

    public Student(String name, int score) {
        this.name = name;
        this.score = score;
    }
}
```

Sorting Yourself Out

What's the sorted version of this array of students?

```
[new Student("Harry", 54),
new Student("Voravich", 80),
new Student("Sid", 79)]
```



```java
public class Student {
    private String name;
    private int score;

    public Student(String name, int score) {
        this.name = name;
        this.score = score;
    }
}
```

There's no well-defined ordering! `.sort()` would fail to compile, actually, because Java can't figure out how to do the necessary comparisons between any two `Student` objects.

The `Comparable` Interface

- Built-in Java interface
- Includes a single abstract method to implement: `compareTo`
  - A class that implements `Comparable` must provide an implementation of `compareTo`
  - Objects of a class implementing the ADT are "sortable"
- Imposes a total ordering on the objects of each class that implements it
  - This ordering is referred to as the class' *natural ordering*, and
  - The class's `compareTo` method is referred to as its *natural comparison method*

The `Comparable` ADT: `compareTo`

- Compares two objects for order
- Returns:
  - a negative integer if the object on which the method is invoked is *less than* the object passed as an argument
  - zero if the object on which the method is invoked is *equal to* the object passed as an argument
  - a positive integer if the object on which the method is invoked is *greater than* the object passed as an argument

```
objInvokedOn.compareTo(objPassedAsArg);
```

Making an Object Sortable

`Comparable` is generically typed, so you have to specify the type in the class definition

```java
public class Student implements Comparable<Student> {
    private String name;
    private int score;

    public Student(String name, int score) {
        this.name = name;
        this.score = score;
    }

    @Override
    public int compareTo(Student other) {
        // TODO: implement so that Students are ordered by score, incr.
    }
}
```

Making an Object Sortable

`Comparable` is generically typed, so you have to specify the type in the class definition

```java
public class Student implements Comparable<Student> {
    private String name;
    private int score;

    public Student(String name, int score) {
        this.name = name;
        this.score = score;
    }

    @Override
    public int compareTo(Student other) {
        return this.score - other.score;
    }
}
```

Sorted Data Structures

- Use a Binary Search Tree to store records (more on these in a couple weeks)
- Records need to be compared in order to find where to insert a new record
- Implementations:
  - TreeSet
  - TreeMap

The `Comparator` ADT

- Defines a comparison function, which imposes a *total ordering* on some collection of objects
- Provides an ordering for collections of objects that don't have a natural ordering
    - i.e. those that don't implement `Comparable`

The `Comparator` ADT: `compare`

```
int compare(T o1, T o2);
```

Compares `o1` and `o2` for order. Returns:

- a negative integer if `o1` is *less than* `o2`
- zero if `o1` is *equal to* `o2`
- a positive integer if `o1` is *greater than* `o2`

  (same rules as `compareTo()`)

Example: Why?

Consider the following class:

```
public class Tuple<L, R> {
    private L left;
    private R right;

    public Tuple(L left, R right) {
        this.left = left;
        this.right = right;
    }
}
```

Example: Why?

The following code would throw an exception:

```
Tuple<Integer, Integer> t1 = new Tuple<>(7, 1);
TreeSet<Tuple<Integer, Integer>> s = new TreeSet<>();
s.add(t1);
```

```
Exception in thread "main" java.lang.ClassCastException: class Tuple cannot be cast to class java.lang.Comparable
```

Example: Using a Comparator

```
Comparator<Tuple<Integer, Integer>> cmp = new Comparator<>() {
    @Override
    public int compare(Tuple<Integer, Integer> t1, Tuple<Integer, Integer> t2) {
        return t1.left - t2.left;
    }
};

TreeSet<Tuple<Integer, Integer>> s1 = new TreeSet<>(cmp);
s1.add(t1);
```

This sorted collection of `Tuple` objects will be maintained in ascending order of their left values.

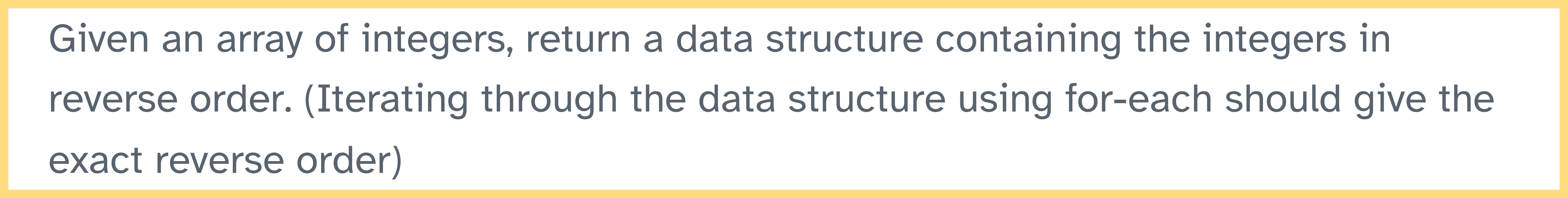
The Map Interface

- A map is an object that maps keys to values.
- A map cannot contain duplicate keys, but duplicate values are OK
    - Each key can only map to at most one value
- Subinterfaces and implementations:
    - SortedMap
        - TreeMap
    - HashMap

The Map Interface: Operations

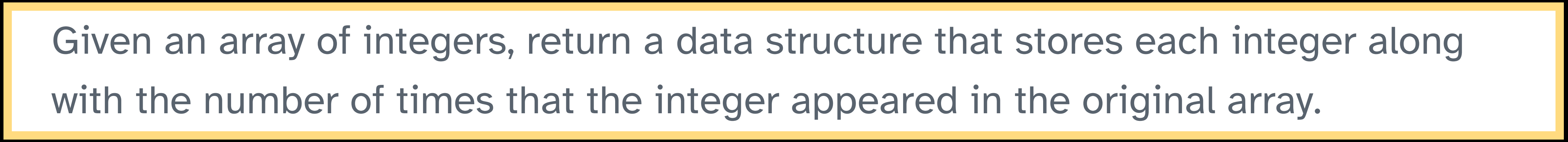
Method	Purpose
`V put(K key, V value)`	Associates the specified value with the specified key in this map
`V get(Object key)`	Returns the value to which the specified key is mapped, or null
`boolean containsKey(Object key)`	Returns `true` iff this map contains a mapping for the specified key
`V remove(Object key)`	Removes the mapping for the specified key from this map if present
`boolean remove(Object key, Object value)`	Removes the entry for the specified key only if it is currently mapped to the specified value

Collection Exercises



Given an array of integers, return a data structure containing the integers in reverse order. (Iterating through the data structure using for-each should give the exact reverse order)

Collection Exercises



Given an array of integers, return a data structure that stores each integer along with the number of times that the integer appeared in the original array.

Collection Exercises

Given an array of integers, return another array of integers with all duplicate integers removed.

Collection Exercises

Given an array of names, return the data structure that will be most efficient in looking up whether a name was contained in the original array.

Summary

- Different data structures, *even those with the same opersations*, have different trade-offs
- It's important to learn which data structures are appropriate for the problem at hand

Membership/containment: sets!

Ordering: lists!

Association/mapping: maps!