

# Introduction to Programming Languages and Techniques



Midterm II  
Review

# Topics

- String processing idioms
- I/O streams
- Generic classes (and methods)
- Reading and writing JUnit tests
- Collection classes
  - Standard collection interfaces
  - Using collections in programs
  - Implementing collections
    - Make sure you can reproduce the `ArraySimpleCollection` and `LinkedSimpleCollection` implementations from scratch

# Non-Topics

- Details of exceptions, e.g.
  - What it means that exceptions are objects
  - try...finally...
- Wildcard generics

# Some Recommended Reading

## ■ Generics:

- Niño & Hosch, chapter 12

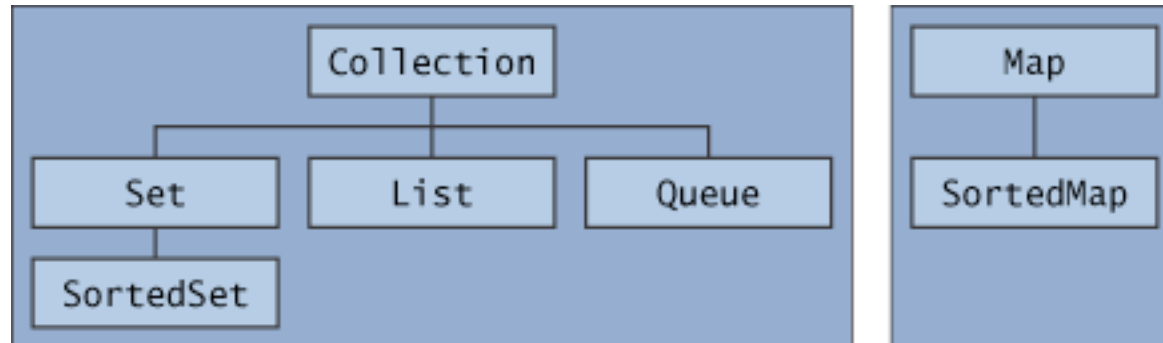
## ■ I/O Streams:

- <http://java.sun.com/docs/books/tutorial/essential/io/streams.html>
- <http://java.sun.com/docs/books/tutorial/essential/io/bytestreams.html>
- <http://java.sun.com/docs/books/tutorial/essential/io/charstreams.html>
- <http://java.sun.com/docs/books/tutorial/essential/io/buffers.html>

## ■ Collections:

- <http://java.sun.com/docs/books/tutorial/collections/intro/index.html>
- <http://java.sun.com/docs/books/tutorial/collections/interfaces/index.html>
- <http://java.sun.com/docs/books/tutorial/collections/implementations/set.html>

# Collection Interfaces



# Collection Interfaces

- Collection
  - Generic interface with common operations
- Set<T>
  - No duplicates
  - iterator returns elements in unspecified order
- SortedSet<T>
  - Like Set<T>, but iterator returns elements in increasing order
- List<T>
  - Duplicates allowed
  - iterator returns elements in order added
- Queue<T>
  - (see docs if you're interested)
- Map<K,V>
  - finite functions from K to V

# JUnit

- It is pretty likely that the midterm will include a question of this form:
  - “Here are a correct and a buggy version of a method for doing <some task>. Write three different JUnit tests that will succeed on the good one and fail on the bad one.”

# Linked Structures

- It is very likely that the midterm will include a question where you are asked to write a method that adds some functionality to the `SimpleLinkedListCollection` class.
  - I.e., you should understand how to build, traverse, and manipulate linked lists.

# SOME EXERCISES

# Exercise: Using collections

For each of the following scenarios, write the full type of the data structure you'd use to store the data. Use some combination of the generic `Map`, `Set`, and `List`. Remember to include their generic types (eg. `List<String>`, not `List`) and that `Maps`, `Sets`, and `Lists` can be nested. You may assume that classes representing each of the objects in the scenario exist, and that all of the classes implement `equals` and `hashCode` appropriately so that they can be used in sets and as keys in maps. However do not assume that these objects store any particular data. Remember to use fast data structures when you don't need the features of the slower ones. Finally, you can call the classes anything reasonable (`Title`, `Artist`, `Price`,...).

**Example:** You want to organize the current roster of each NBA team so that players can be found by searching for their team. The answer is

```
Map<Team, Set<Player>>
```

(do not assume, for instance, that the `Team` object contains the set of players on that team)

(a) An online music store needs to keep track for each artist of the artist's titles and respective prices.

(a) An online music store needs to keep track for each artist of the artist's titles and respective prices.

ANSWER: `Map<Artist, Map<Title, Price>>`

The wish list data structure for the music store keeps track for each customer of the set of titles the customer would like to have.

The wish list data structure for the music store keeps track for each customer of the set of titles the customer would like to have.

ANSWER:

`Map<Customer, Set<Title>>`

`Map<Customer, List<Title>>`

- (c) We keep track of the popularity of titles in the music store by maintaining an ordered table of titles “graded on the curve”: which titles are in the top 10% of sales, which are in the top 20% but not in the top 10%, which are in the top 30% but not in the top 20%, . . . . In other words, we have an ordered table of bins where each contains the titles in the appropriate sales bracket.

- (c) We keep track of the popularity of titles in the music store by maintaining an ordered table of titles “graded on the curve”: which titles are in the top 10% of sales, which are in the top 20% but not in the top 10%, which are in the top 30% but not in the top 20%, . . . . In other words, we have an ordered table of bins where each contains the titles in the appropriate sales bracket.

ANSWER:

```
List< Set<Title>>
```

```
Map<Integer, Set<Title>>
```

- (d) The music store is getting into socially-based recommendations, and so it wants a data structure that stores for each customer what other customers have music interests in common with that customer.

ANSWER:

`Map<Customer, Set<Customer>>`

- (e) The store's new recommendation algorithm generates automatically for each customer some playlists, each containing some titles in a play order, based on information about the customer's tastes. These recommendations are stored in a data structure that can be efficiently accessed when a customer logs in to retrieve the playlists generated for that customer.

ANSWER:

```
Map<Customer, Map<Playlist, List <Title>>>
```

```
Map<Customer, Map<String, List <Title>>>
```

# Exercise: max

- The Java standard libraries include a class `Comparator<T>` whose instances can be used for ordering two objects (of type `T`).

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

- The integer returned by the `compare` method will be negative if the first argument is less than the second, zero if they are equal, and positive if the second is greater than the first.

- Complete the definition of a new method 'max' in the `LinkedSimpleCollection` class, taking a comparator and returning the largest element of the collection (according to the comparator).

```
public class LinkedSimpleCollection<E>
    implements SimpleCollection<E> {

    public E max (Comparator<E> order) {
        // Fill in here:

    }
}
```

# Answer

```
public E max (Comparator<E> order) {
    E current = null;
    for (SimpleIterator<E> i = iterator(); i.hasNext(); ) {
        E e = i.next();
        if (current == null) current = e;
        else if (order.compare(current, e) < 0) current = e;
    }
    return current;
}
```

# Exercise: get

```
class LinkedSimpleCollection<E> {  
    ...  
    public E get(int i) {  
        // implement me!  
        // throws IndexOutOfBoundsException if  
        // i < 0 or i >= length of list.  
    }  
}
```

# Answer

```
public E get (int i) {
    Node<E> current = first;
    for (int j = 0;
        j < i && current != null; j++) {
        current = current.next;
    }
    if (current == null)
        throw new IndexOutOfBoundsException();
    else return current.element;
}
```

# More Exercises...

- Many more exercises can be found in the sample exams from previous offerings of 120, available on the course Exams web page
- Particularly recommended:
  - Spring 07, #3
  - Fall 07, #3 and #4
  - Spring 08, #1 and #2
  - Fall 08, #2 and #3
  - Spring 09, #1