LEC 09

CSE 122

Maps

Questions during Class?

Raise hand or send here

sli.do #cse122

BEFORE WE START

Slido vote & chat with your neighbors:

What is your favorite form of potato?

Music: 122 25wi Lecture Tunes

Instructor: Elba Garza

TAs:

Anya	Daniel Ryan	Ken	Nicole
Ashley	Diya	Kuhu	Nicole
Cady	Elizabeth	Kyle	Niyati
Caleb	Hannah	Leo	Sai
Carson	Harshitha	Logan	Steven
Chaafen	Ivory	Maggie	Yang
Colin	Izak	Mahima	Zach
Connor	Jack	Marcus	
Dalton	Jacob	Minh	

Lecture Outline

- **Announcements**
- Sets, Iterators, For-each Loop Review
- Map Review
- *Debrief PCM: Count Words*
- *Practice: joinRosters*
- *Practice: mostFrequentStart*

Announcements

- Reminder: Quiz 1 is Tuesday, February 18th
  - Quiz 0 results coming soon!
- Resubmission Cycle 2 (R2) form open now
  - Due Tuesday, Feb 11 by 11:59 PM
  - Eligible Assignments: **C0**, P0, C1
- Programming Assignment 2 (P2) releasing later today!
  - Due Thursday, **Feb 20** by 11:59pm

Lecture Outline

- Announcements
- **Sets, Iterators, For-each Loop Review**
- Map Review
- *Debrief PCM: Count Words*
- *Practice: joinRosters*
- *Practice: mostFrequentStart*

Sets (ADT)

- A collection of unique values (no duplicates allowed) that can perform the following operations efficiently:
  - add
  - remove
  - search (contains)

- We don't think of a set as having indices; we just add things to the set in general and don't worry about order

Sets in Java

- `Set` is an interface in Java
  - In `java.util`

- `HashSet` and `TreeSet` are classes that implement the `Set` interface in Java
  - `HashSet`: Very fast! Implemented using a "hash table" array
    - *Elements are stored in an unpredictable order*
  - `TreeSet`: Pretty fast! Implemented using a "binary search tree"
    - *Elements are stored in sorted order*

Set Methods

Method	Description
`add(value)`	Adds the given value to the set, returns whether or not the given value was added successfully
`contains(value)`	Returns `true` if the given value is found in this set
`remove(value)`	Removes the given value from the set; returns `true` if the set contained the value, `false` if not
`clear()`	Removes all elements from the set
`size()`	Returns the number of elements in list
`isEmpty()`	Returns `true` if the set's size is 0; `false` otherwise
`toString()`	Returns a `String` representation of the set such as `"[3, 42, -7, 15]"`

For-Each Loop

A new kind of loop! ✨

```
Set<String> words = new HashSet<>();
for (String s : words) {
    System.out.println(s);
}
```

- BUT, you cannot modify the data structure inside a for-each loop
  - You will get a `ConcurrentModificationException`
  - They are “read-only”

Iterators

A new object that has access to all of the elements of a given collection and gives them to you one at a time—and it lets you modify the collection!

```
Set<String> lyrics = new HashSet<>();
Iterator<String> itr = lyrics.iterator();
while (itr.hasNext()) {
    String next = itr.next();
    if (next.contains("woo")) {
        itr.remove();
    }
}
```

Iterators

- Returned by the `iterator()` method

Methods	Description
`hasNext()`	Returns `true` if there are more elements for the iterator to return
`next()`	Returns the next element in the iteration
`remove()`	Removes and returns the element that was last returned by `next()`

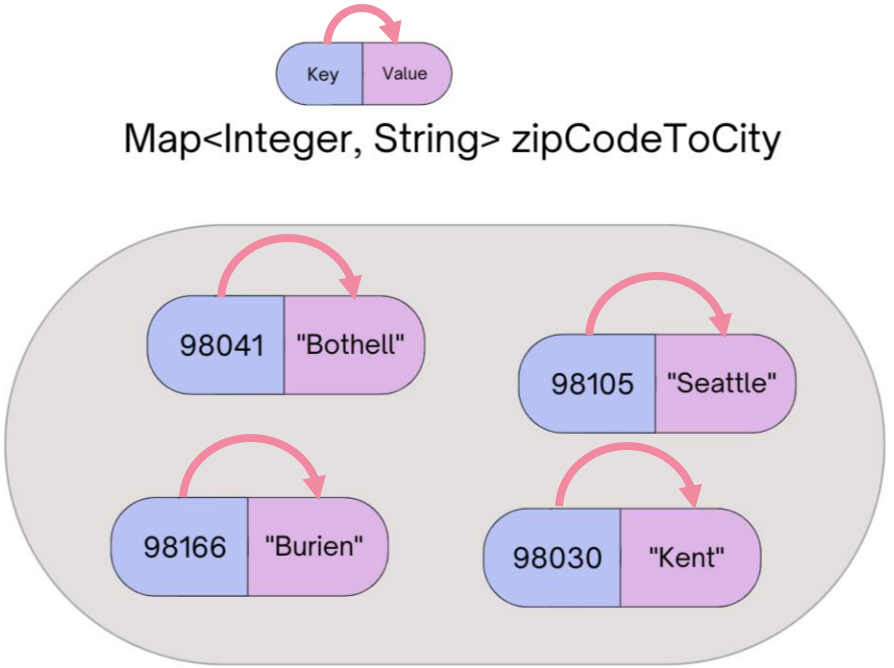
- You must use the iterator's `remove()` method to remove things from what you're iterating over – otherwise you will get a `ConcurrentModificationException`

Lecture Outline

- Announcements
- Sets, Iterators, For-each Loop Review
- **Map Review**
- *Debrief PCM: Count Words*
- *Practice: joinRosters*
- *Practice: mostFrequentStart*

Map ADT

- Data structure to map keys to values
  - Keys can be any* type; Keys must be unique
  - Values can be any type
- Operations
  - `put(key, value)`: Associate key to value
    - Overwrites duplicate keys
  - `get(key)`: Get value for key
  - `remove(key)`: Remove key/value pair



*Same as Python's `dict`

Programming with Maps in Java

- Interface: `Map`
- Implementations: `TreeMap`, `HashMap`

```
// Making a Map
Map<String, String> favArtistToSong = new TreeMap<>();

// adding elements to the above Map
favArtistToSong.put(”Stromae", " Ma Meilleure Ennemie");
favArtistToSong.put("Aitana", "Segundo Intento");
favArtistToSong.put("Laufey", "Promise");

// Getting a value for a key
String song = favArtistToSong.get("Aitana");
System.out.println(song);
```

Programming with Maps in Java

Methods	Description
`put(`**key**`, `**value**`)`	adds a mapping from the given key to the given value; if the key already exists, replaces its value with the given one
`get(`**key**`)`	returns the value mapped to the given key (`null` if not found)
`containsKey(`**key**`)`	returns `true` if the map contains a mapping for the given key
`remove(`**key**`)`	removes any existing mapping for the given key
`clear()`	removes **all** key/value pairs from the map
`size()`	returns the number of key/value pairs in the map
`isEmpty()`	returns `true` if the map's size is 0
`toString()`	returns a string such as `"{a=90, d=60, c=70}"`
`keySet()`	returns a set of all keys in the map
`values()`	returns a collection of all values in the map

Map Implementations

- Our first data structures with marked differences in how their implementations behave
- One `Map` ADT / Interface
- Two `Map` implementations
  - `TreeMap` – Pretty fast, and sorted keys
  - `HashMap` – Extremely fast, unsorted keys

```
Map<String, Integer> map1 = new TreeMap<>();
Map<String, Integer> map2 = new HashMap<>();
...
```

Practice : Think

sli.do #cse122

Select the method calls required to modify the given map `m` as follows:

Assume `m`'s contents are

```
98030="Kent"
98178="Seattle"
98166="Burien"
98041="Bothell"
```

We want to modify `m` so that its contents are

```
98030="Kent"
98178="Tukwila"
98166="Burien"
98041="Bothell"
98101="Seattle"
98126="Seattle"
```

A. `m.put(98178, "Tukwila");`

B. `m.remove(98178);`

C. `m.put(98126, "Seattle");`

D. `m.get(98178, "Seattle");`

E. `m.put(98101, "Seattle");`

Practice : Pair

sli.do #cse122

Select the method calls required to modify the given map m as follows:

Assume m's contents are

```
98030="Kent"
98178="Seattle"
98166="Burien"
98041="Bothell"
```

We want to modify m so that its contents are

```
98030="Kent"
98178="Tukwila"
98166="Burien"
98041="Bothell"
98101="Seattle"
98126="Seattle"
```

A. `m.put(98178, "Tukwila");`

B. `m.remove(98178);`

C. `m.put(98126, "Seattle");`

D. `m.get(98178, "Seattle");`

E. `m.put(98101, "Seattle");`

Lecture Outline

- Announcements
- Sets, Iterators, For-each Loop Review
- Map Review
- ***Debrief PCM: Count Words***
- *Practice: joinRosters*
- *Practice: mostFrequentStart*

Lecture Outline

- Announcements
- Sets, Iterators, For-each Loop Review
- Map Review
- *Debrief PCM: Count Words*
- ***Practice: joinRosters***
- *Practice: mostFrequentStart*

joinRosters

Write a method `joinRosters` that combines a `Map` from student name to quiz section, and a `Map` from TA name to quiz section and prints all pairs of students/TAs.

For example, if `studentSections` stores the following map:

```
{Alan=AC, Jerry=AB, Yueying=AA, Sharon=AB, Steven=AB, Zewditu=BA}
```

And `taSections` stores the following map

```
{Ayush=BA, Marcus=AA, Aishah=AB, Chaafen=AC}
```

```
AC: Alan - Chaafen
AB: Jerry - Aishah
AB: Sharon - Aishah
AB: Steven - Aishah
AA: Yueying - Marcus
BA: Zewditu - Ayush
```

Lecture Outline

- Announcements
- Map Review
- Debrief PCM: Sets, Iterators, For-each Loop
- Debrief PCM: Count Words
- *Practice: joinRosters*
- ***Practice: mostFrequentStart*** 

mostFrequentStart

Write a method called mostFrequentStart that takes a Set of words and does the following steps:

- Organizes words into “word families” based on which letter they start with
- Selects the largest “word family” as defined as the family with the most words in it
- Returns the starting letter of the largest word family (and if time, should update the Set of words to only have words from the selected family).

mostFrequentStart

For example, if the `Set` words stored the values

```
["hello", "goodbye", "library", "literary", "little", "repel"]
```

The word families produced would be

```
'h' -> 1 word ("hello")
'g' -> 1 word ("goodbye")
'l' -> 3 words ("library", "literary", "little")
'r' -> 1 word ("repel")
```

Since 'l' has the largest word family, we return `'l'` and modify the `Set` to only contain `Strings` starting with 'l'.