Unit 14: Maps

Anthony Estey

CSC 115: Fundamentals of Programming II

University of Victoria

Maps - Motivation

- With classes we have been able to create different types of objects to fit the needs of the problem we are trying to solve in our programs

- In the last assignment, we saw that sometimes it can be difficult to compare two objects so that they can be ordered
  - In the Passenger class we implemented a compareTo method which allowed us to order Passengers by their boarding gate, followed by check-in time.

- It would be nice if we could easily and effectively order elements within our data structures

Maps - Introduction

- A **Map** is also called an **Associative Array**, **Associative List**, or a **Dictionary**

- Main idea:
  - Assign a unique key to every element in the collection
  - Construct the keys such that they can be easily and efficiently be compared and ordered
  - Map each key to a data value

Maps - Introduction

- What do we end up with?
  - A collection of ***key-value*** pairs
  - Keys are unique
  - Each key **maps** to (or is **associated** with) a value

Maps - Example

- Student information
  - key: student id number
    - unique to every student; easy to compare and order;
  - value associated with each key: student data
    - transcript, program, schedule, contact information, etc.

- Dictionary:
  - key: word
    - easy to compare and order strings
  - value associated with each key: word information
    - definition, classification, pronunciation, etc.

The Map ADT

- The Map ADT specifies the following operations:
  - Insert an entry (key-value pair) into the map
  - Determine whether a key exists in the map
  - Get the value associated with an existing key in the map
  - Remove an entry from the map (specified by its key)
  - Update the value associated with a key
  - Output all entries in the map
  - Remove all entries from the map

Map Interface:

```
put(k, v)
containsKey(k)
get(k)
remove(k)
entries()
size()
clear()
```

Always using the key for all operations!

Maps

- Important takeaways:
  - Keys must be distinct
  - Each value is “mapped” to a unique key
  - Each element must have both a key and a value
  - We need the key to insert, find, update, and remove elements

Map Implementation

- We will use Java Generics for flexibility
  - this allows us to use any type for the key and value

- General structure:
  - `Map<KeyType, ValueType> myMap;`
- Specific examples:
  - `Map<Integer, BankAccount> accounts;`
    - Bank account data is accessed by the particular account's corresponding unique integer id
  - `Map<String, Student> classList;`
    - Student data is accessed by the student's unique student id

Operation Visualization

```
Map<Integer, Employee> staff = new MyMap<Integer, Employee>();
```

staff:

Map Interface:

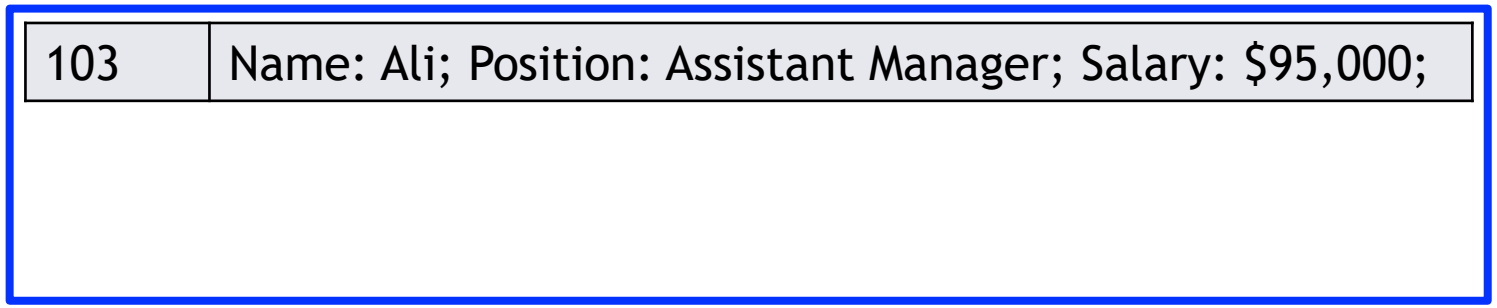
```
put(k, v)
containsKey(k)
get(k)
remove(k)
entries()
size()
clear()
```

Operation Visualization

```
Map<Integer, Employee> staff = new MyMap<Integer, Employee>();

Employee ali = new Employee("Ali", …);
staff.put(103, ali);
```

staff:



Map Interface:

```
put(k, v)
containsKey(k)
get(k)
remove(k)
entries()
size()
clear()
```

Operation Visualization

```
Map<Integer, Employee> staff = new MyMap<Integer, Employee>();
```

```
staff.size();                → 1
staff.containsKey(103);      → true
staff.get(103)               → Ali's employee object data
```

staff:

103	Name: Ali; Position: Assistant Manager; Salary: $95,000;

Map Interface:

```
put(k, v)
containsKey(k)
get(k)
remove(k)
entries()
size()
clear()
```

Operation Visualization

```
Map<Integer, Employee> staff = new MyMap<Integer, Employee>();

Employee kim = new Employee("Kim", …);
Employee alex = new Employee("Alex", …);
staff.put(204, kim);
staff.put(137, alex);
```

staff:

103	Name: Ali; Position: Assistant Manager; Salary: $95,000;
204	Name: Kim; Position: CEO; Salary: $185,000;
137	Name: Alex; Position: Junior Developer; Salary: $70,000

Map Interface:

```
put(k, v)
containsKey(k)
get(k)
remove(k)
entries()
size()
clear()
```

Operation Visualization

```
Map<Integer, Employee> staff = new MyMap<Integer, Employee>();
```

```
staff.size();                → 3
staff.containsKey(103);      → true
staff.containsKey(199);      → false
```

staff:

103	Name: Ali; Position: Assistant Manager; Salary: $95,000;
204	Name: Kim; Position: CEO; Salary: $185,000;
137	Name: Alex; Position: Junior Developer; Salary: $70,000

Map Interface:

```
put(k, v)
containsKey(k)
get(k)
remove(k)
entries()
size()
clear()
```

Operation Visualization

```
Map<Integer, Employee> staff = new MyMap<Integer, Employee>();
```

```
staff.remove(204);
Employee otherAlex = new new Employee("Alex", …);
staff.put(211, otherAlex);
```

keys are distinct. Values don't have to be

staff:

103	Name: Ali; Position: Assistant Manager; Salary: $95,000;
211	Name: Alex; Position: Junior Developer; Salary: $70,000
137	Name: Alex; Position: Junior Developer; Salary: $70,000

Map Interface:

```
put(k, v)
containsKey(k)
get(k)
remove(k)
entries()
size()
clear()
```

Operation Visualization

```
Map<Integer, Employee> staff = new MyMap<Integer, Employee>();
```

```
Employee sam = new new Employee("Sam", …);
staff.put(103, sam);
```

the put operation adds a new entry to the map if the key does not already exist, otherwise, it updates an existing entry

staff:

103	Name: Sam; Position: Accountant; Salary: $135,000;
211	Name: Alex; Position: Junior Developer; Salary: $70,000
137	Name: Alex; Position: Junior Developer; Salary: $70,000

Map Interface:

```
put(k, v)
containsKey(k)
get(k)
remove(k)
entries()
size()
clear()
```