CS 2100: Data Structures & Algorithms 1

Hash Tables

ADTs So Far; Sets and Maps in Java

Dr. Nada Basit // basit@virginia.edu

Spring 2022

Friendly Reminders

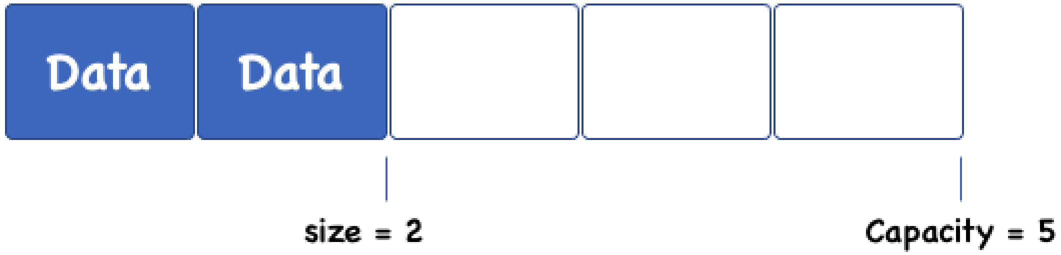
- The University updated the mask policy. As per my Request on Mar 28, 2022 (see Collab), I would greatly appreciate if you would do me a kind favor by **continuing to wear your masks** in CS 2100 (Ridley G008). I know it is a lot to ask, and it is **voluntary**, but I appreciate your understanding.
- If you forget your mask (or mask is lost/broken), I have a few available
  - Just come up to me at the start of class and ask!
- No eating or drinking in the classroom, please
- Our lectures will be **recorded** (see Collab) – please allow 24-48 hrs to post
- If you feel **unwell**, or think you are, please stay home
  - *We will work with you!*
  - At home: eye mask instead! Get some rest ☺

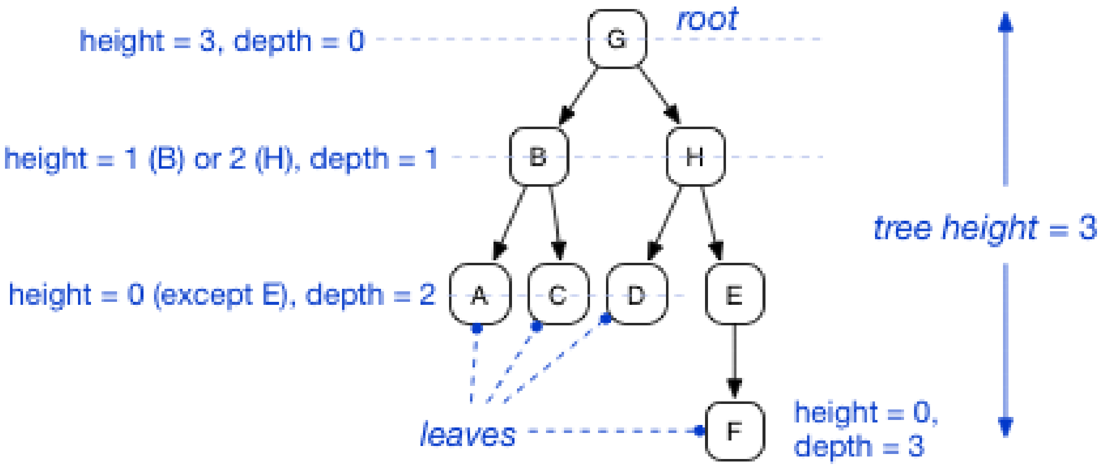
ADTs So Far

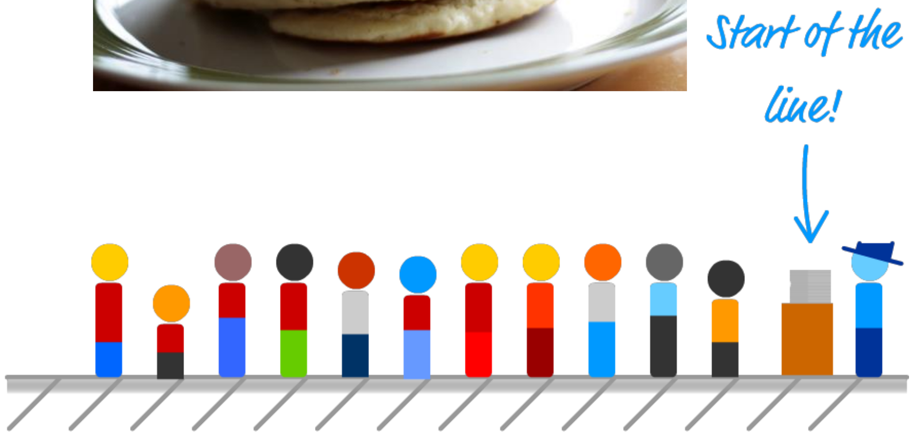
An overview of the Abstract Data Types we have seen so far

ADTs We Have Seen So Far

- Lists
- Stacks
- Queues
- Trees







Lists

- Operations:
  The operations are generally linear-time operations
  - `find`
  - `insert`
  - `remove`
  - `findKth`
- Implementations
  - Array (vector)
  - Linked list

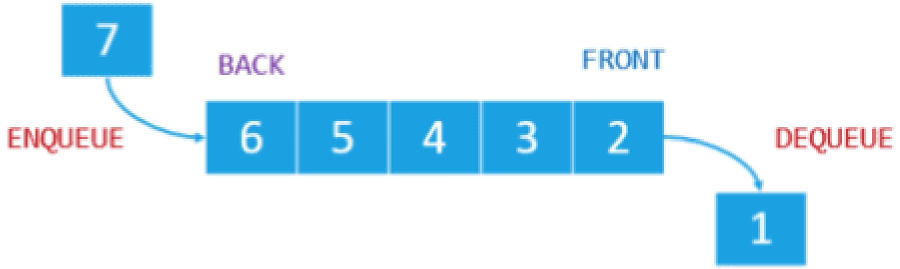
	Array (vector)	Linked List
find	$\Theta(n)$	$\Theta(n)$
insert	$\Theta(n)$ worst case, but often $\Theta(1)$	$\Theta(1)$
remove	$\Theta(n)$	$\Theta(n)$
findKth	$\Theta(1)$	$\Theta(n)$

Stacks

- List with data handled last-in first-out
- Operations:
  The operations are generally constant-time operations
  - `push`
  - `pop`
  - `top`
- Implementations
  - Array (vector)
  - Linked list

	Array (vector)	Linked List
push	$\Theta(n)$ worst case, but often $\Theta(1)$	$\Theta(1)$
pop	$\Theta(1)$	$\Theta(1)$
top	$\Theta(1)$	$\Theta(1)$

Queues



- First-in first-out list
- Operations:
  The operations are generally constant-time operations
  - enqueue
  - dequeue
- Implementations
  - Array (vector)
  - Linked lists

	Array (vector)	Linked List
enqueue	$\Theta(n)$ worst case, but often $\Theta(1)$	$\Theta(1)$
dequeue	$\Theta(1)$	$\Theta(1)$

Trees

- Goal is Θ(log n) runtime for most operations
  - Binary search trees
  - AVL Trees
  - Red-black trees
  - Splay trees – *a self-balancing BST (main idea: bring recently accessed items to the root of the tree, making recently searched items accessible in O(1) time if accessed again. In a typical application, **80%** of the access are to **20%** of the items*
- *Balanced trees are generally logarithmic-time operations*

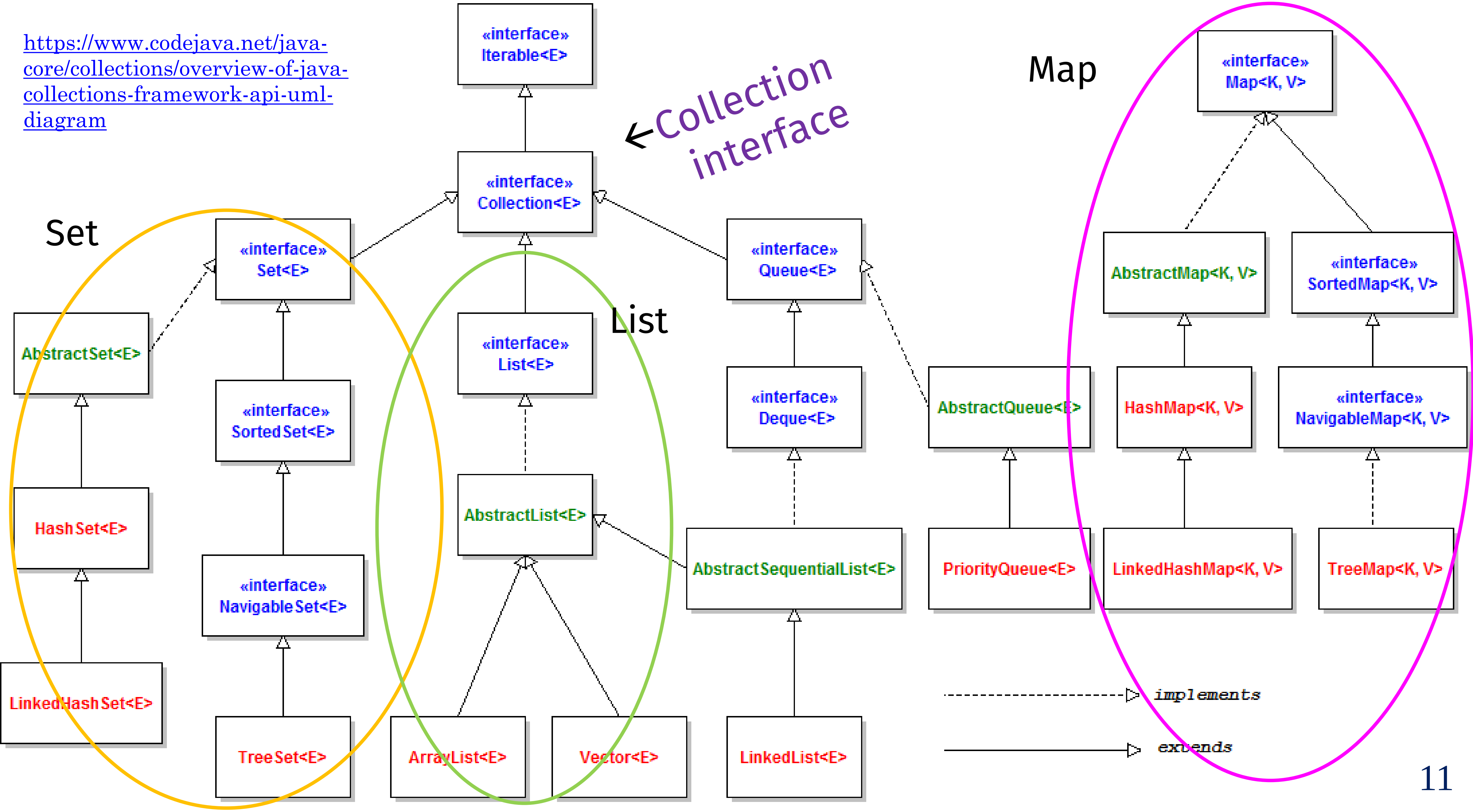
	BST	AVL	Red-black
find	worst case $\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$
insert	worst case $\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$
remove	worst case $\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$

Is There Anything Faster?

- Fastest possible search using binary comparison: $\Theta(\log n)$
- We can do better: (almost) constant ($\Theta(1)$) is possible with **hash tables!**
- **Hash tables** (lookup table)
  - Standard set of operations: find, insert, delete
  - **No** ordering property!
    - Thus, no `findMin` or `findMax`

Aside:
Sets and Maps

Introduction to the Set and Map data structures https://www.codejava.net/java-core/collections/overview-of-java-collections-framework-api-uml-diagram



Two New Abstract Data Types (ADTs)

- **Set**: Any data structure that stores a bunch of *unordered* elements
  - Insert/retrieve done using the **element** itself (e.g., insert(data))
  - ***No*** *duplicate values allowed in sets*

- **Map**: Any data structure that stores key-value pairs
  - insert and retrieve by **key**. e.g., insert("oranges", 2.95);
  - retrieve("oranges") returns 2.95
  - ***No*** *duplicate keys allowed*

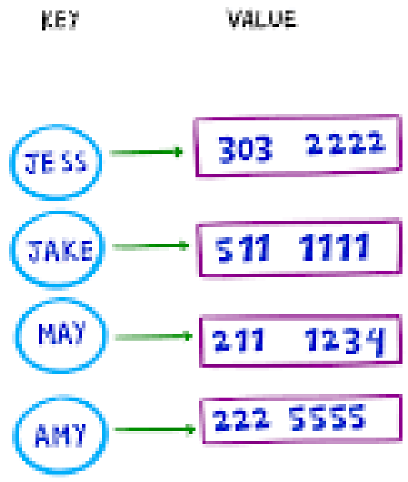
Two New Abstract Data Types (ADTs) :: SETs

- **Set**: **Methods** include:
  - `add(data)`, `find(data)`, `remove(data)`
  - *No real concept of indexing like a list*
- Set implementation examples:
  - **Trees**(BST, etc.); Java has a **`TreeSet`** class
    - Requires `.compareTo()` method
  - **Hash Tables**; Java has a **`HashSet`** class
    - Requires `.equals()` method
    - Also requires `.hashCode()` method

Think: marbles in a bag!
(unique, marbles!)

Two New Abstract Data Types (ADTs) :: MAPs

- **Map**: **Methods** include:
  - `put(key, T data)`, `T get(key)`, `T remove(key)`
  - *No real concept of indexing like a list*
- Map implementation examples:
  - **Trees**(BST, etc.); Java has a **`TreeMap`** class
    - Requires `.compareTo()` method
  - **Hash Tables**; Java has a **`HashMap`** class
    - Requires `.equals()` method
    - Also requires `.hashCode()` method



Keys with their associated values (e.g. name to phone #)

Which ADT is Hash Table?

- A **hash table** ***(we will see next lecture!)*** can be used to implement a **Map or a Set**
- In this class, we will usually use the latter (easier to show examples) but sometimes use either.

Aside:
Sets and Maps - Examples

Some Set and Map Java Examples

SETS

Looping over a Set (using for-each loop)

- Does **not** allow for positional access. There are no indices in a Set but you can still loop over each of the elements of a Set using a **for-each loop**:

```
// Create a set (a HashSet) called "mySet"
Set<String> mySet = new HashSet<String>();
```

```
// Assuming we populate mySet with String values…
// Loop through mySet and print out each of the elements:
for (String ele : mySet) {  // using a for-each loop!
      System.out.println(ele);
}
```

TreeSet and HashSet

TreeSet

- The “**tree**” refers to type of data structure used
- Bonus! Prints in “correct” (***sorted***) order
- Items maintained in order to avoid duplicates

HashSet

- Uses a “**hash**” or unique number for each item to avoid duplicates *(no order guarantee)*

Set - Methods

- Some **Set** behaviors
  - `boolean` **`add`**`(elem)` – returns *false* if already there
  - `boolean` **`remove`**`(elem)` – returns *false* if not there
- What's nice here: (returns *false* if can't, *true* otherwise)
  - Try to **add** something that's already there?
    Remove something that's not there? *No problem!*
  - It basically **ignores** that attempt! Doesn't throw error. Returns *false*.

Example using add()

```
TreeSet<String> aSet = new TreeSet<String>();
ArrayList<String> cities = new ArrayList<String>();

// assume contents:
{ "Paris", "Amsterdam", "London", "Lisbon", "Paris",
  "Vienna", "Prague", "Rome", "London" }

// What's a quick way to remove duplicates from "cities"?
for(String city : cities) {
      aSet.add(city); // duplicates will be removed! Done!
}
```

Sets use **generics** – they **type** in <>'s has to be an **object type**

TreeSet Example

```
// Create a TreeSet of Integers
TreeSet<Integer> tree = new
        TreeSet<Integer>();

// Add some elements
tree.add(12);
tree.add(63);
tree.add(34);
tree.add(45);
```

```
// Displaying the Tree set data
// Notice: elements are printed in
// SORTED order! (Not by accident!)
// It's a property of the "Tree" Set
System.out.print("Tree set data: ");

// for-each loop to print
for( Integer ele : tree ) {
 System.out.print(ele + " ");
}
```

Output:
Tree set data: 12 34 45 63

TreeSet Example – other handy methods

```
// Create a TreeSet of Integers
TreeSet<Integer> tree = new
       TreeSet<Integer>();

// Add some elements
tree.add(12);
tree.add(63);
tree.add(34);
tree.add(45);

int target = 34;
```

```
// What if I wanted to check if a
valued existed within the set?
if(tree.contains(target)) {
   System.out.print("Found!");
}
else
   System.out.print("Not Found!");
// You can use CONTAINS() method!
// No need for a loop (see above).
```

```
// To check if something is NOT
// contained (doesn't exist in set):
if(!tree.contains(target)) {...}
```

MAPS

Important Map Methods

- Keys and values
  - **put**(key, value), **get**(key), **remove**(key) – see next slide for details
  - **containsKey**(key), **containsValue**(value)
- Important / useful:
  - **keySet()** // returns a **Set** of keys
  - **values()** // returns a **Collection** of values
- Others methods too! See Java API for more.

More Details on Map Methods

- `put(key, value)` – stores new data for the key
  - If key is not in the map – makes new entry for it
  - If key is in the map – replaces the old data associated with the key
  - (is like “add” or “replace”)
- `get(key)` – retrieves the data (value) based on the key
- `remove(key)` – removes a key-value pair
  - Just call remove with the key (don’t have to pass the value)
- Remember – key is <u>not</u> a *position*!

More Details on Map Methods

- Remember map declarations need data types for **both the key and value**

- e.g. `HashMap<String, Cat> catsMap = new HashMap<String, Cat>();`

- **Add** to the map using `.put()`

```
Cat tiggerObj = new Cat();
catsMap.put("Tigger", tiggerObj);
```

- **Get** from the map using `.get()`

```
Cat tiggerObj = catsMap.get("Tigger"); //get on the key
```

Maps

- Map keys can be any object
  (as long as it meets the requirements for a type of map used)

```
HashMap<Dog, Person> dogsPerson = new HashMap<Dog,Person>();

Dog lucyObj = new Dog();
Person fred = new Person();
dogsPerson.put(lucyObj, fred);

Person p = dogsPerson.get(lucyObj);
```

HashMap Example

```
// Create a HashMap with a
// String Key and Integer Value
HashMap<String, Integer> vehicles = new
HashMap<String, Integer>();

// Add some vehicles
// (Key-Value pairs are:
// Vehicle and number of each vehicle)
vehicles.put("BMW", 5);
vehicles.put("Mercedes", 3);
vehicles.put("Audi", 4);
vehicles.put("Ford", 10);

// How many items in the HashMap? (4)
System.out.println("Total vehicles: " +
vehicles.size());
```

```
// Iterate over all vehicles,
// using the keySet method.
// for-each loop comes in handy!
for(String key: vehicles.keySet())
   System.out.println(
      key + " - " + vehicles.get(key));
System.out.println();

// Using get(), provide the Key,
// receive the associated Value (4 Audi cars)

String searchKey = "Audi";
if(vehicles.containsKey(searchKey))
  System.out.println("Found total " +
     vehicles.get(searchKey) + " " +
     searchKey + " cars!\n");
```

Output of previous code:

```
Total vehicles: 4
Audi - 4
Ford - 10
Mercedes - 3
BMW - 5

Found total 4 Audi cars!
```

Gives you the **value**:

```
vehicles.get(searchKey)
```