Please Complete The Socrative Pulse Survey

URL: gosocrative.com

Room Name: JAVAGOLD

House Keeping:

- Start Thinking of some topics/questions for our review day on Friday!
  - Formatting and pretty printing for a better command line experience for users.
  - Quick overview of Big O notation for a deeper understanding of data structures and algorithms
- Pair project assignment on Friday
- Zoom policy recap--- Recordings and on-camera attendance
- PowerPoint PDF availability
- Changing how I work with the LMS/Gitlab

Intro to Collections 2: Maps and Sets

Objectives

- Review
- Maps
- Sets
- Collections: Compare & Contrast

Review

- List: ordered data structure of varying size
  - Uses methods, not square brackets []
  - Not compatible with primitives
  - Usually implemented as an ArrayList

```
List<String> stringList = new
ArrayList<String>();

stringList.add(“string”);
stringList.add(“list”);
stringList.add(“elements”);
```

- There are classes for each primitive type
  - boolean -> Boolean, int -> Integer, etc.
  - Converting between the two is called AutoBoxing

```
Double doubleValue = 42.42;
double newValue = doubleValue;
```

- Use a stack or queue if ordering of elements is important
  - Stack ordering is LIFO, **L**ast **I**n **F**irst **O**ut
  - Queue ordering is FIFO, **F**irst **I**n **F**irst **O**ut

What Loop to use

Loop	**Reason**
for	Need an index or count. Need to be able to change a Collection or Array Need to be able to move through a Collection or Array in an arbitrary manner Used for Collections, Arrays, or to loop a set number of times
forEach	Need to loop from the first item to the last of an Array or Collection Don’t need an index or count Only Used with Collections or Arrays
while	Have only a boolean condition that determines when the loop should stop Used when have a condition unrelated to a count or index that determines when the loop should stop.

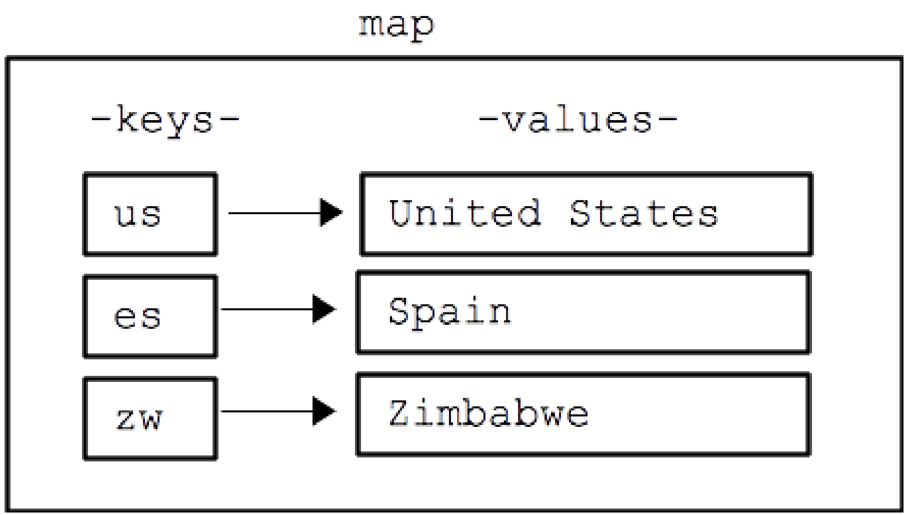
Loop Case Scenarios

```
class Car {
     public void drive(){ // code }
          public String toString(){ // code }
}

List<Car> cars;
// Code to initialize cars list with 10 Car objects
```

Scenario	**for**	**forEach**	**while**
Call the `drive` method on all `Car` objects in the `cars` list		✓	
Iterate through the `cars` list in reverse order	✓		
Change the third car in the `cars` list to a different `Car` object	✓		
Menu to continually ask the user to add `Car` objects to the `cars` list			✓
Remove the fifth `Car` object in the `cars` list	✓		
Display information on each `Car` object in the `cars` list using the `toString` method		✓	

Key Value Pairs

A set of 2 pieces of data, where the *value* is associated by a unique *key*, allowing the *value* to be retrieved by providing the *key*.



Key Value Pairs in real life

1. City lookup by zip code
   43220 → Columbus, OH
   90210 → Beverly Hills, CA
1. Phone book
   867-5209 → Jenny
   719-266-2837 → Callin Oats
1. Vending Machine
   A1 → Snickers Bar
   B2 → Potato Chips

Map<K, V>

A **map** is a collection that utilizes Key Value Pairs, allowing ***values*** to be assigned and then located using *user-defined* ***keys***.

- Keys are unique, i.e. there are no duplicate keys.
- If a key-value pair is added with a key that already exists, it will overwrite the existing one!

Map Keys

1. Can be any reference type
2. Must be unique
3. Stored as a Set

Map Values

1. Can be any reference type
2. Can have duplicates
3. Can be null

Map<K, V>: Declaring

Maps follow this declaration pattern:

```
import java.util.HashMap;
import java.util.Map;

public class MyClass {

                public static void main(String args[ ]) {

                                Map <Integer, String> myMap =
                                        new HashMap<Integer, String>();

                }
}
```

We will need these 2 imports for a hash map.

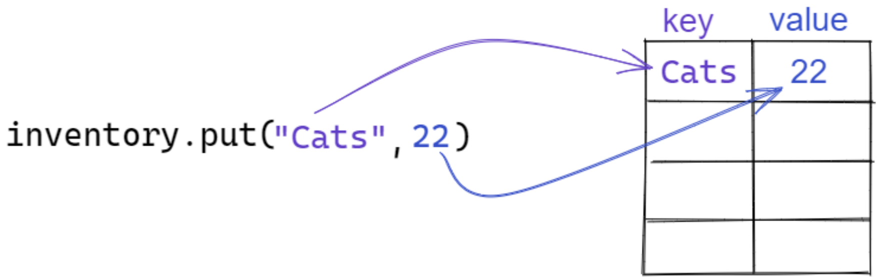
We are creating a type of Map called a HashMap

We have specified that the key will be an integer and the value will be the String

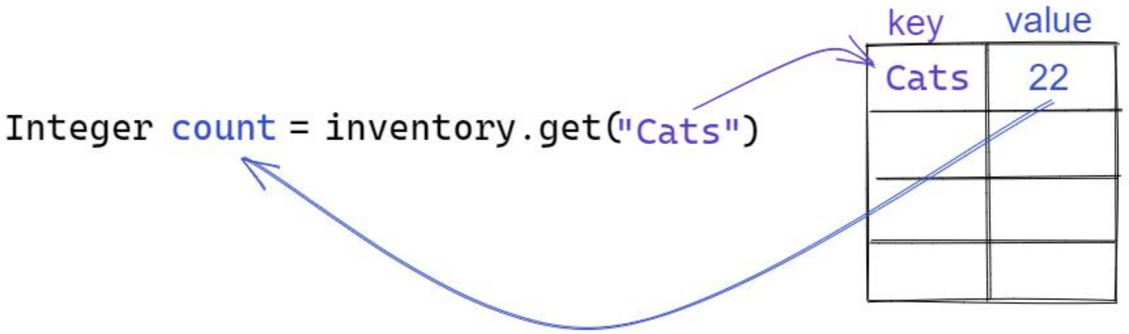
Note the “**new**” keyword which **instantiates** the map.

Add and Getting Map Elements

```
Map<String, Integer> inventory = new HashMap<String, Integer>();
```



```
inventory.put("Cats", 22)
```



```
Integer count = inventory.get("Cats")
```

Map<K, V> Operations

Declare	`Map<K, V> myMap;`
Initialize	`Map<K, V> myMap= new HashMap<K, V>;`
Get value	`myMap.`**`get`**`( key );`
Set element	

Iterate (loop)	`// Can only use a for-each loop`
	`for(K eachElement : myMap.`**`keySet`**`()){ ... }`

Add	`myMap.`**`put`**`( key, value );`
Remove	`myMap.`**`remove`**`( key );`
Insert	

Contains	`myMap.`**`containsKey`**`( key );`
	`myMap.`**`containsValue`**`( value );`

Map<K, V> Methods Cont.

.put(key, value)	***Adds or Updates*** the value in the Map. If the key does not exist it adds the key and the value.
.get(key)	Returns the value associated with the given key. If the key does not exist null is returned.
.remove(key)	Removes a key/value pair from the map. If the key exists the value is returned, otherwise null is returned
.containsKey(key)	Returns true if the key exists in the map
.containsValue(value)	Returns true if the value exists in the map
.keySet()	Returns all the keys in the map as a Set<T> collection
.entrySet()	Returns all Key/Value pairs as Entry<T, T> objects

Visual Explanation

Looping over a Map with keySet()

keySet() returns the keys in the map as a Set<T>, which can be used in a for-each loop and then used to get the value.

```
Map<String, Integer> inventory = new HashMap<String, Integer>();

for ( String key : inventory.keySet() ) {

    Integer value = inventory.get( key );

    }
```

Looping over a Map with entrySet()

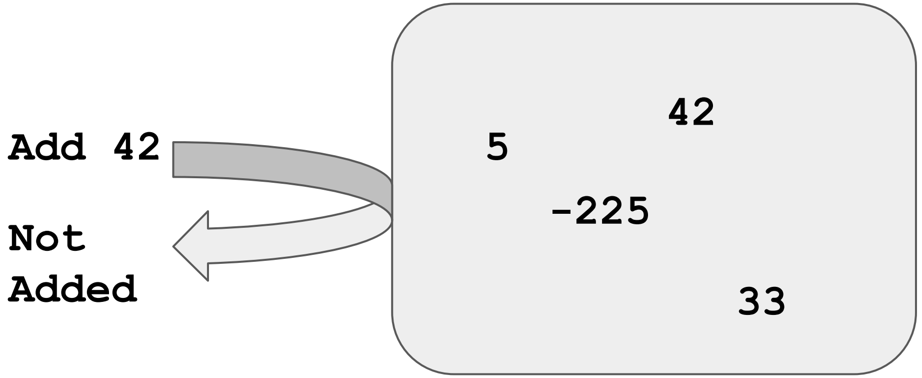
entrySet() returns the key value pairs in the map as a Set<Entry<T, T><, which can be used in a foreach loop.

```
Map<String, Integer> inventory = new HashMap<String, Integer>();

for ( Entry<String,Integer> nextEntry : inventory.entrySet() ) {

    String key = nextEntry.getKey();
    Integer value = nextEntry.getValue();

    }
```

Sets: Introduction

A set is also a collection of data.

- No duplicate elements are allowed.
  - Adding an existing element DOES NOT change the size of the set.
- It is also **unordered**
  - There is no *index* into a set.



Set<T>: Declaring

The following pattern is used in declaring a set.

```
import java.util.HashSet;
import java.util.Set;

public class MyClass {

		public static void main(String args[]) {

				Set<Integer> primeNumbersLessThan10 = new
HashSet<Integer>();
		}
}
```

Note the we will need these 2 imports for a hash map.

We are creating a type of Set called a HashSet

We have specified that the set will contain only integers.

Note the "**new**" keyword which **instantiates** the set.

Set<T> Operations

Declare	`Set<T> mySet;`
Initialize	`Set<T> mySet = new HashSet<T>;`
Get element	
Set element	

Iterate (loop)	`// Can only use a for-each loop`
	`for(T eachElement : mySet ){ ... }`

Add	mySet.**add**(element);
Remove	mySet.**remove**(element);
Insert	
Contains	mySet.**contains**(element);

Arrays vs Lists vs Maps vs Sets vs Stacks vs Queues

- Use **Arrays** when you know the maximum number of elements, and you know you will primarily be **working with primitive data types**.
- Use **Lists** when you want something that works like an array, but you don’t know the exact number of elements.
- Use **Maps** when you have key value pairs, where the keys are unique.
  - No duplicate elements
  - Fast element search
- Use **Sets** when you know your data does not contain repeating elements.
  - Can be used to remove duplicate elements.
  - All the keys on a given map comprise a set.
- Use **Stacks** and **Queues** when you want to enforce ordering of elements.
  - Stacks LIFO ordering
  - Queues FIFO ordering

Collection Complexity

Each collection time has a complexity associated with

1. Insert (at end, at beginning, at end)
2. Searching
3. Retrieval
4. Removal (from end, from beginning, from end)

Table of Collection Complexities

Complexities Chart Overview

It is ALWAYS more important to find a correct solution first. Only after a problem is solved should it be looked at for performance improvements.

Never Optimize First, only at the end

When to use each Collection

Collection	Usage	Use Case	Rarity
List	To hold a group of unknown items in a set order.	Shopping List	Very common
Map	To hold a group of key/value pairs where the value can be looked up by the key.	Inventory where a SKU is used to look up a product	Common
Set	To hold a group of unique items	Removing duplicate entries from a list of names	Common
Queue	To organize a group of items in a First In First Out ordering for processing	Email/Print queuing	Rare
Stack	To organize a group of items in a Last In First Out ordering for processing.	Document Undo functionality	Rare