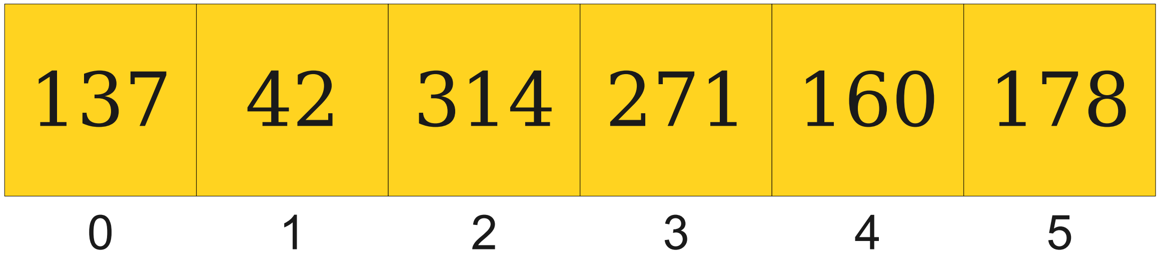
Introducing Arrays



137	42	314	271	160	178
0	1	2	3	4	5

- An array stores a **sequence** of multiple objects.
  - Can access objects by index using [].
- All stored objects have the same type.
  - You get to choose the type!
- Can store *any* type, even primitive types.
- Size is fixed; cannot grow once created.

Default Values in Arrays

- When creating an array:
  - `int`, `double`, `char`, etc. default to 0,
  - `boolean` defaults to `false`, and
  - Objects default to `null`.

Basic Array Operations

- To create a new array, specify the type of the array and the size in the call to `new`:

```
Type[] arr = new Type[size]
```

- To access an element of the array, use the square brackets to choose the index:

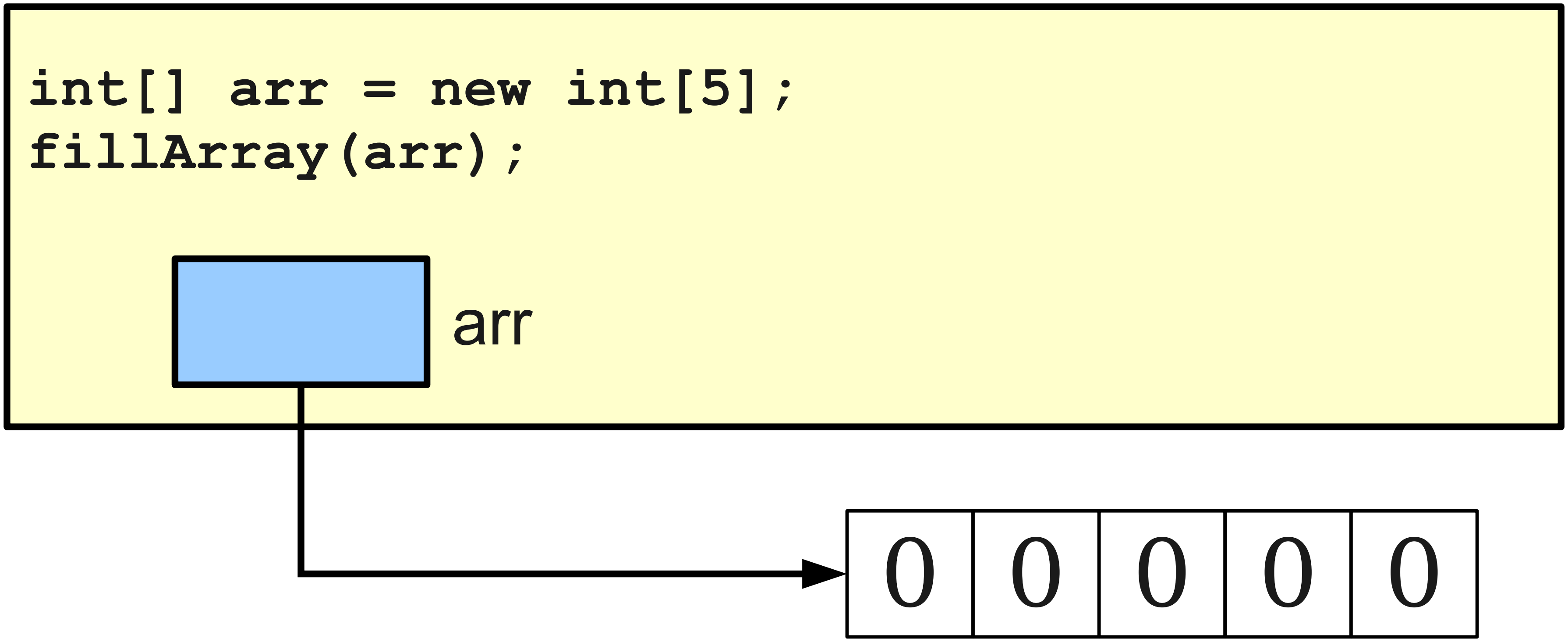
```
arr[index]
```

- To read the length of an array, you can read the `length` field:

```
arr.length
```

A Nuance of Pass-by-Reference

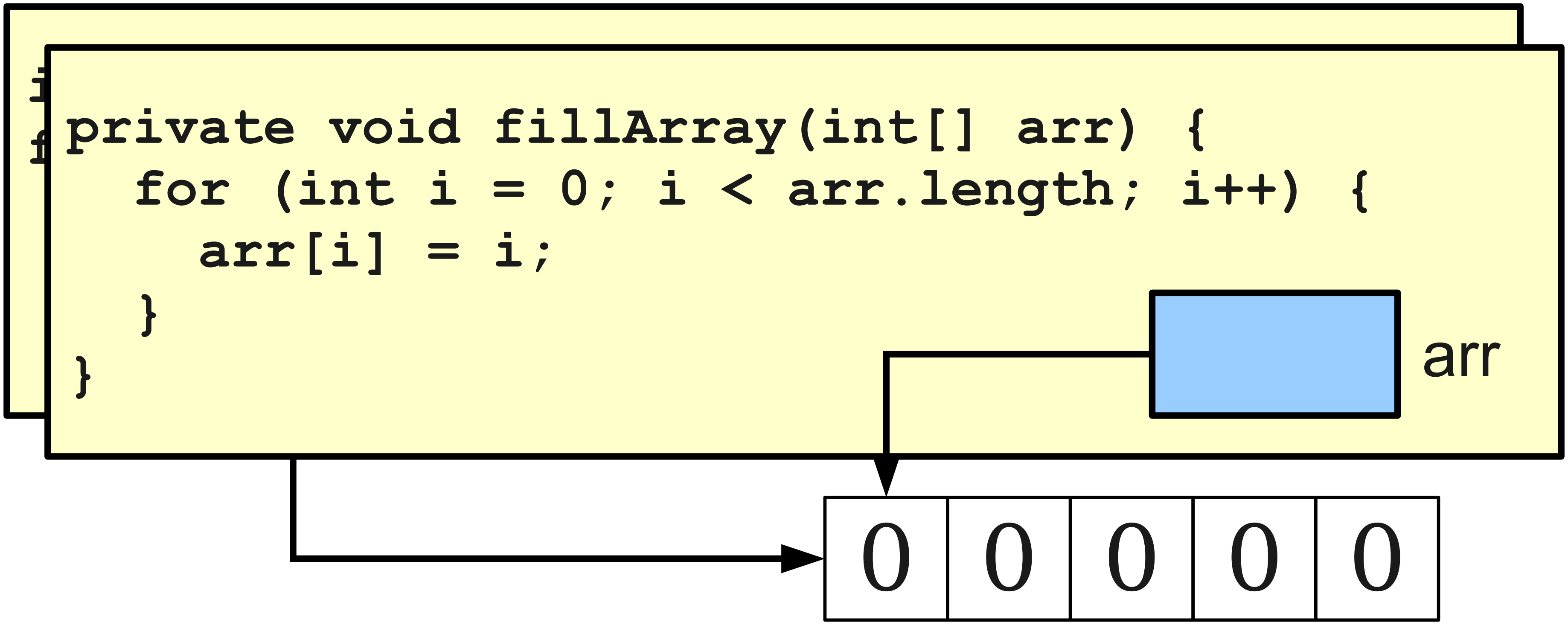
- Arrays are objects, so they are passed by reference.
- The **elements** of an array, like the fields of an object, can be modified inside of a method.



```
int[] arr = new int[5];
fillArray(arr);
```

A Nuance of Pass-by-Reference

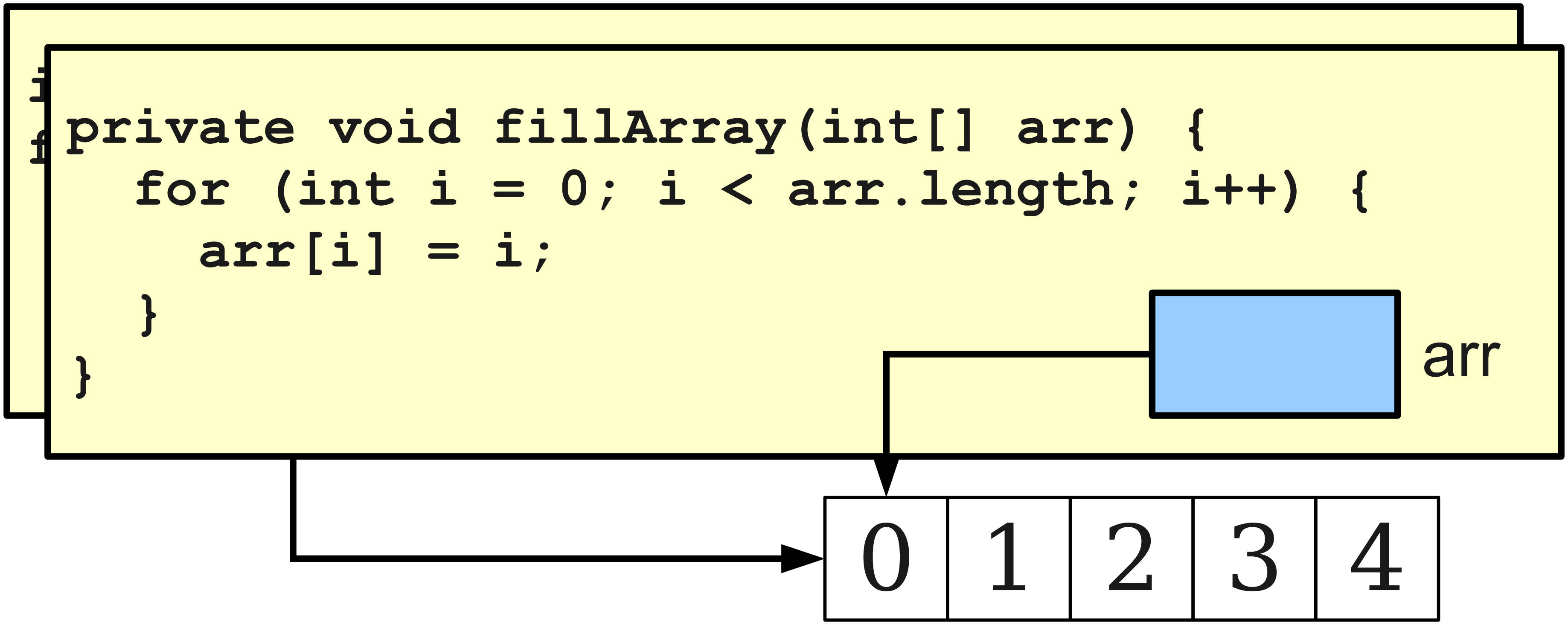
- Arrays are objects, so they are passed by reference.
- The **elements** of an array, like the fields of an object, can be modified inside of a method.



```
private void fillArray(int[] arr) {
  for (int i = 0; i < arr.length; i++) {
    arr[i] = i;
  }
}
```

A Nuance of Pass-by-Reference

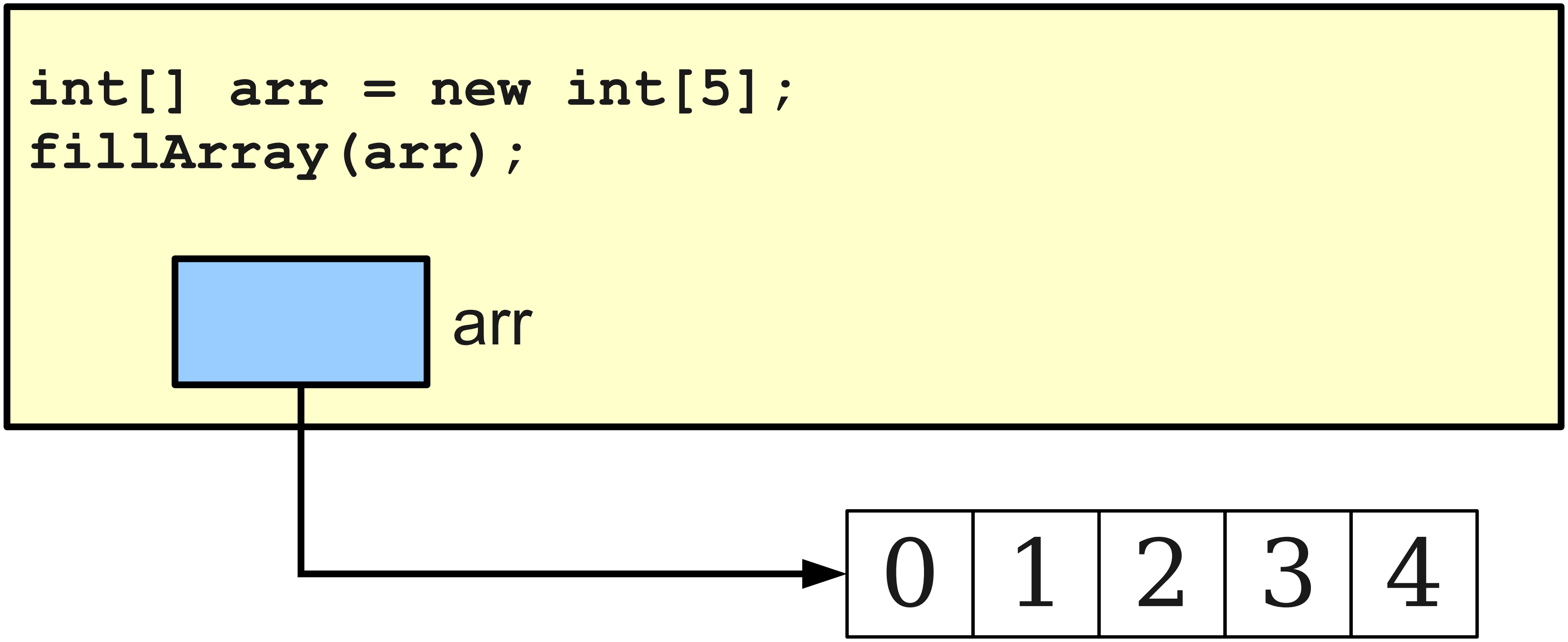
- Arrays are objects, so they are passed by reference.
- The **elements** of an array, like the fields of an object, can be modified inside of a method.



```
private void fillArray(int[] arr) {
   for (int i = 0; i < arr.length; i++) {
      arr[i] = i;
   }
}
```

A Nuance of Pass-by-Reference

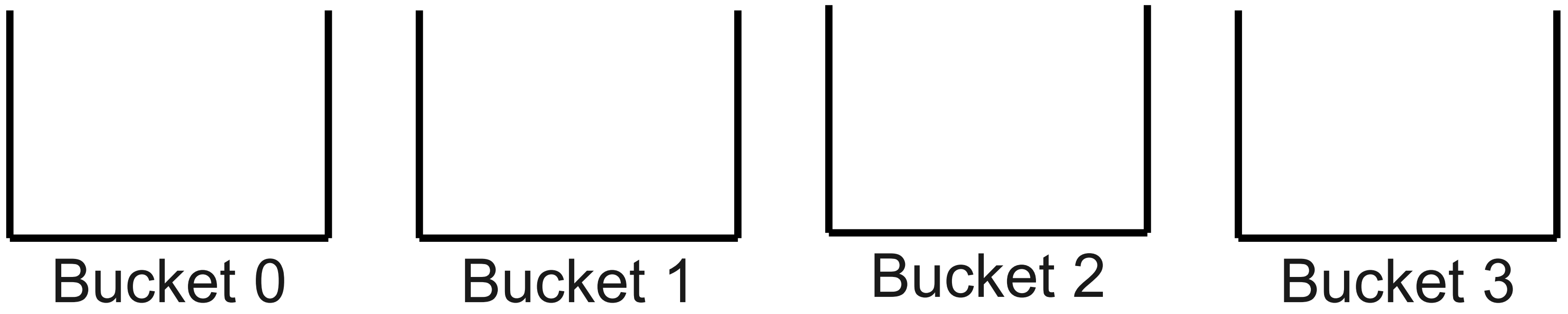
- Arrays are objects, so they are passed by reference.
- The **elements** of an array, like the fields of an object, can be modified inside of a method.



```
int[] arr = new int[5];
fillArray(arr);
```

Why Arrays?

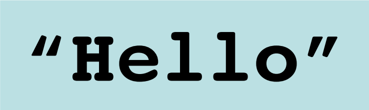
- Arrays are excellent for representing a fixed-size list of **buckets**.
- We can store values in the appropriate bucket by looking up the bucket by index.



Our First ArrayList

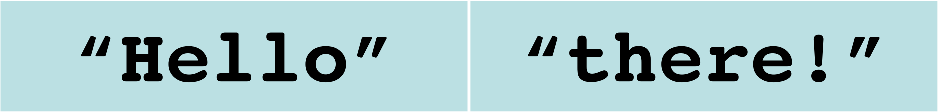
```
// Create an (initially empty) list
ArrayList<String> list = new ArrayList<>();
```

```
// Add an element to the back
list.add("Hello");    // now size 1
```



“Hello”

```
list.add("there!");   // now size 2
```



“Hello” “there!”

ArrayList Methods

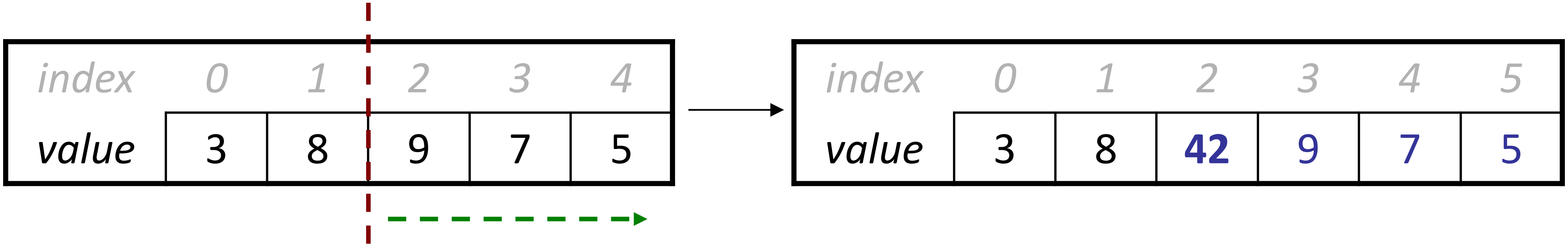
list.add(***value***);	appends value at end of list
list.add(***index***, ***value***);	inserts given value just before the given index, shifting subsequent values to the right
list.clear();	removes all elements of the list
list.get(***index***)	returns the value at given index
list.indexOf(***value***)	returns first index where given value is found in list (-1 if not found)
list.isEmpty()	returns `true` if the list contains no elements
list.remove(***index***);	removes/returns value at given index, shifting subsequent values to the left
list.remove(***value***);	removes the first occurrence of the value, if any
list.set(***index***, ***value***);	replaces value at given index with given value
list.size()	returns the number of elements in the list
list.toString()	returns a string representation of the list such as `"[3, 42, -7, 15]"`

Insert/remove

- If you insert/remove in the front or middle of a list, elements **shift** to fit.

```
list.add(2, 42);
```

  - shift elements right to make room for the new element

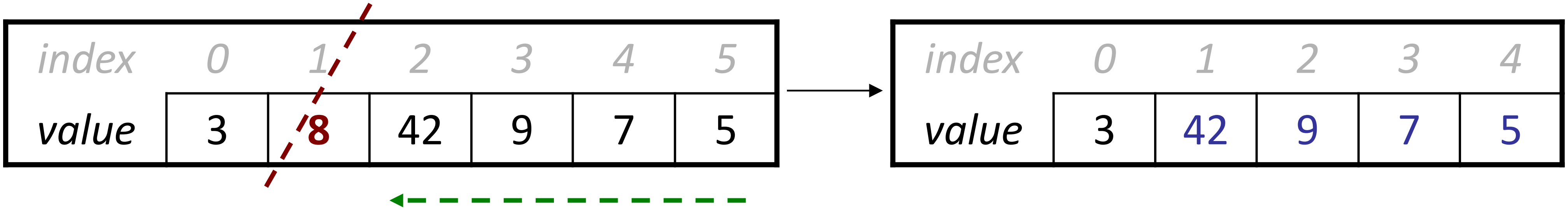


index	0	1	2	3	4
value	3	8	9	7	5

index	0	1	2	3	4	5
value	3	8	**42**	9	7	5

```
list.remove(1);
```

  - shift elements left to cover the space left by the removed element



index	0	1	2	3	4	5
value	3	**8**	42	9	7	5

index	0	1	2	3	4
value	3	42	9	7	5

ArrayLists + Primitives = 💔

Primitive	“Wrapper” Class
`int`	`Integer`
`double`	`Double`
`boolean`	`Boolean`
`char`	`Character`

ArrayLists + Wrappers = ❤️

```
// Use wrapper classes when making an ArrayList
ArrayList<Integer> list = new ArrayList<>();

// Java converts Integer <-> int automatically!
int num = 123;
list.add(num);

int first = list.get(0);   // 123
```

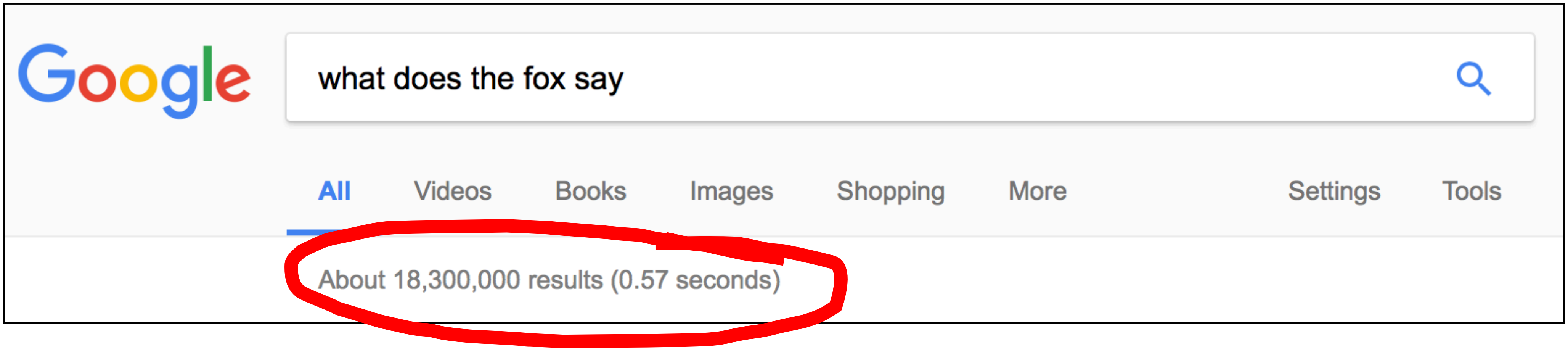
Conversion happens automatically!

Limitations of Lists

- Can only look up by *index* (int), not by String, etc.
- Cumbersome for preventing duplicate information
- Slow for lookup

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	12	49	-2	26	5	17	-6	84	72	3

How Is Webpage Lookup So Fast?



what does the fox say

About 18,300,000 results (0.57 seconds)

Introducing… HashMaps!

- A variable type that represents a collection of **key-value pairs**
- You access values by *key*
- Keys and values can be any type of **object**
- Resizable – can add and remove pairs
- Has helpful methods for searching for keys

HashMap Examples

- **Phone book:** name -> phone number
- **Search engine:** URL -> webpage
- **Dictionary**: word -> definition
- **Bank**: account # -> balance
- **Social Network**: name -> profile
- **Counter**: text -> # occurrences
- And many more…

Our First HashMap

```
import java.util.*;
```

```
HashMap<String, String> myHashMap = new HashMap<>();
```

Our First HashMap

```
HashMap<String, String> myHashMap = new HashMap<>();
```

Our First HashMap

Type of keys your HashMap will store.

```
HashMap<String, String> myHashMap = new HashMap<>();
```

Our First HashMap

Type of values your HashMap will store.

```
HashMap<String, String> myHashMap = new HashMap<>();
```

Our First HashMap

```
HashMap<String, String> myHashMap = new HashMap<>();
```

Our First HashMap

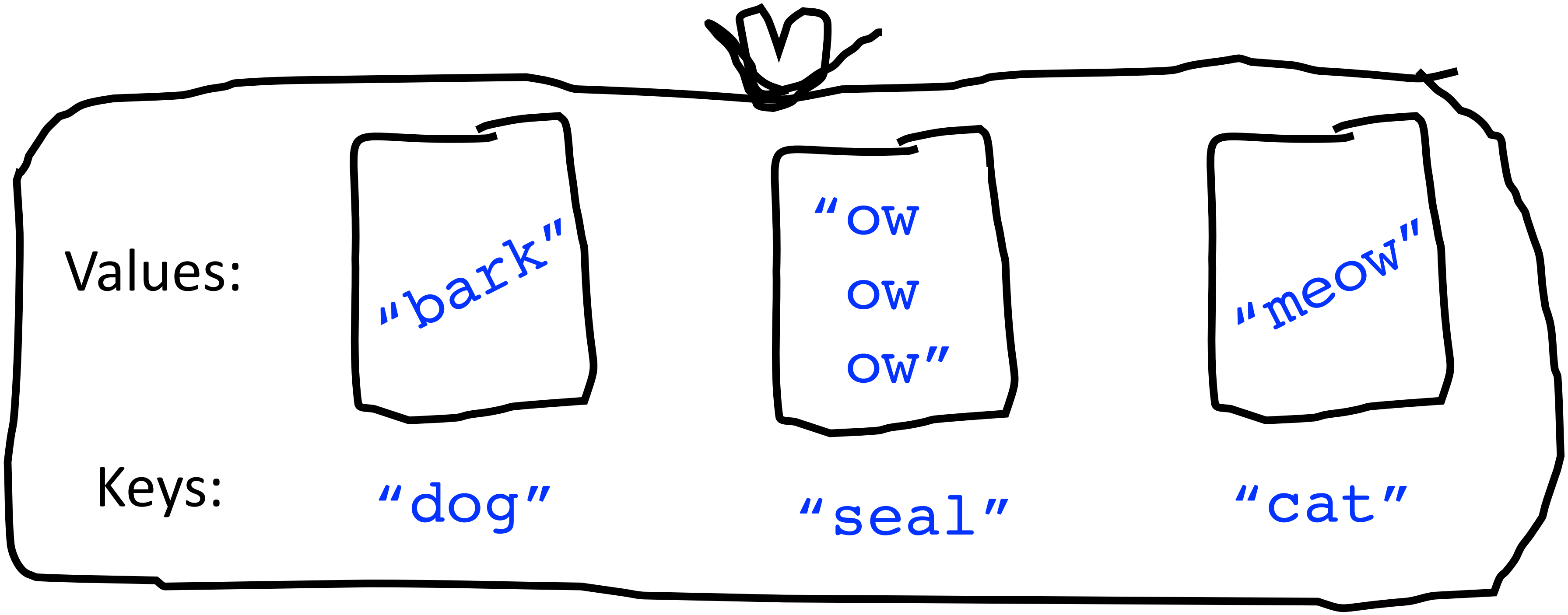
```
HashMap<String, String> myHashMap = new HashMap<>();
```

Our First HashMap

```
HashMap<String, String> myHashMap = new HashMap<>();
```

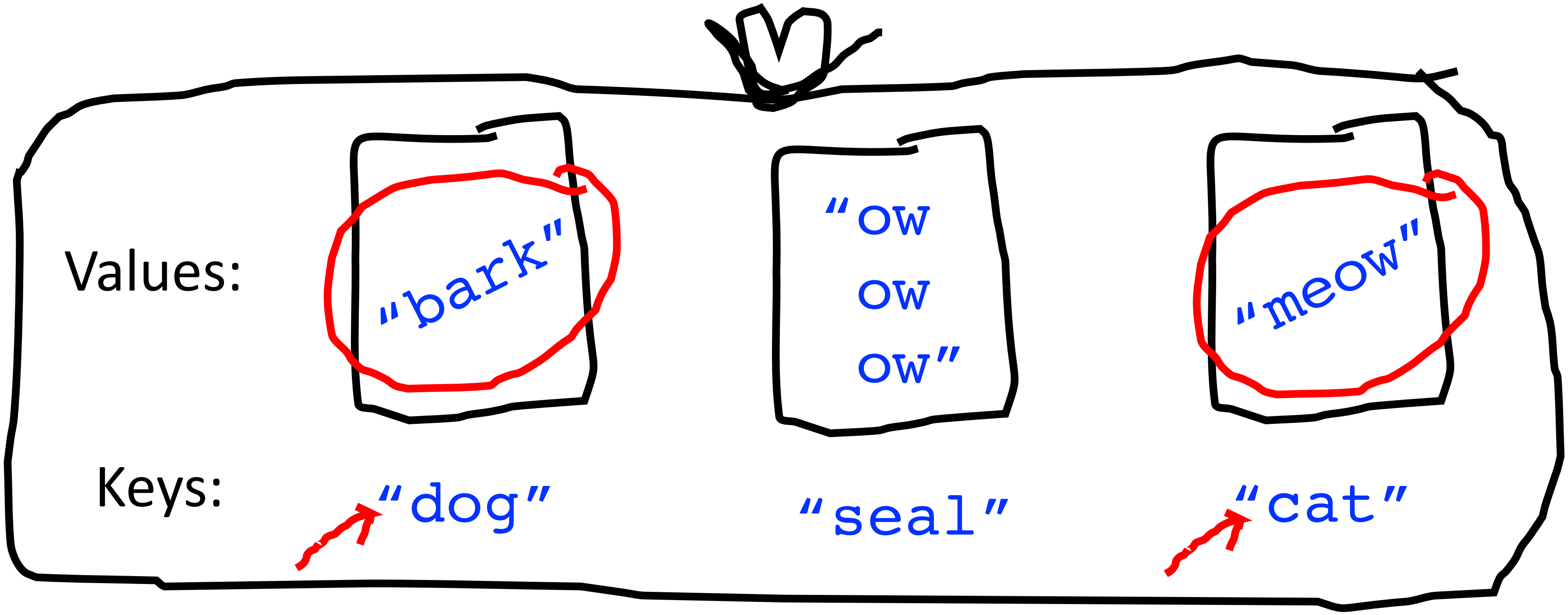
Our First HashMap - Put

```
// Create an (initially empty) HashMap
HashMap<String, String> map = new HashMap<>();
map.put("dog", "bark"); // Add a key-value pair
map.put("cat", "meow"); // Add another pair
map.put("seal", "ow ow"); // Add another pair
map.put("seal", "ow ow ow"); // Overwrites!
```



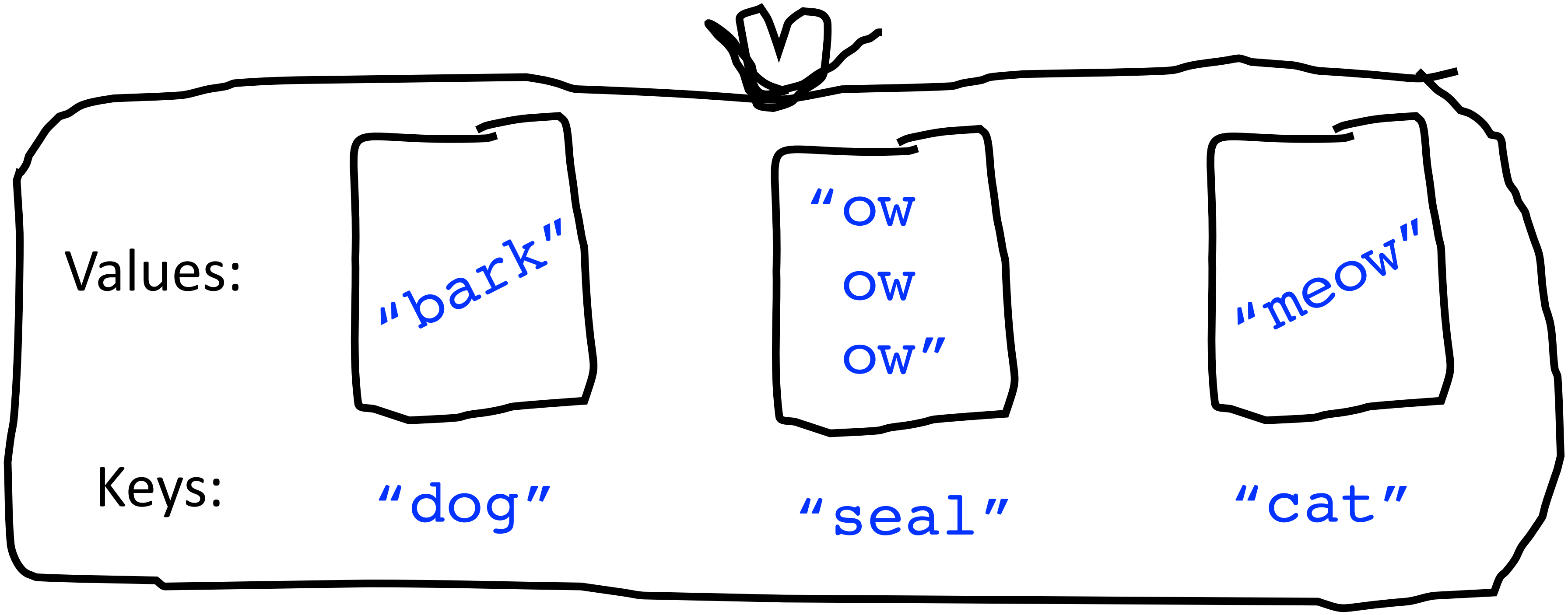
Our First HashMap - Get

```
...
String s = map.get("dog"); // Get a value for a key
String s = map.get("cat"); // Get a value for a key
String s = map.get("fox"); // null
```



Our First HashMap - Remove

```
...
map.remove("dog"); // Remove pair from map
map.remove("seal"); // Remove pair from map
map.remove("fox"); // Does nothing if not in map
```



Review: HashMap Operations

- `m.put(key, value);` Adds a key/value pair to the map.

  ```
  m.put("Eric", "650-123-4567");
  ```

  - Replaces any previous value for that key.

- `m.get(key)` Returns the value paired with the given key.

  ```
  String phoneNum = m.get("Jenny");   // "867-5309"
  ```

  - Returns null if the key is not found.

- `m.remove(key);` Removes the given key and its paired value.

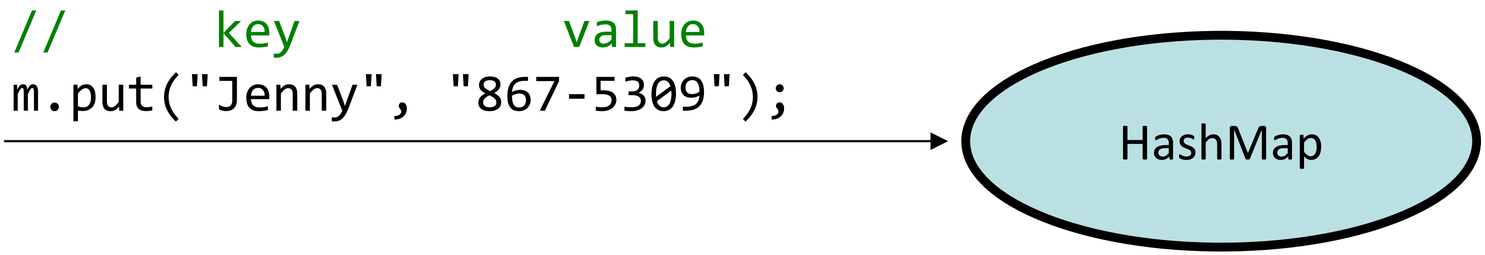
  ```
  m.remove("Rishi");
  ```

  - Has no effect if the key is not in the map.

key	value
"Jenny”	→ "867-5309"
"Mehran"	→ "123-4567"
"Marty"	→ "685-2181"
"Chris”	→ "947-2176"

Using HashMaps

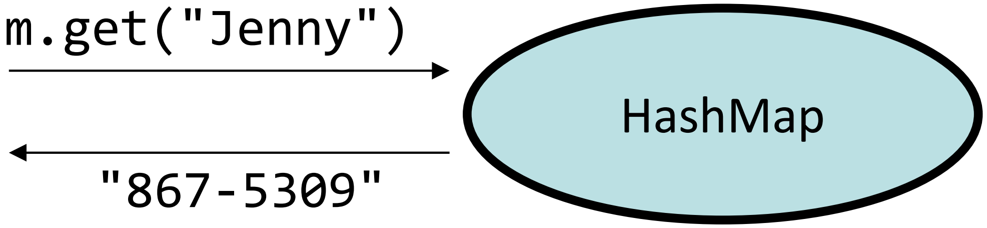
- A HashMap allows you to get from one half of a pair to the other.
  - Remembers one piece of information about every key.



```
//      key          value
m.put("Jenny", "867-5309");
```

  - Later, we can supply only the key and get back the related value:

    Allows us to ask: *What is Jenny’s phone number?*



```
m.get("Jenny")
```

```
"867-5309"
```

Practice: Map Mystery

Q: What are the correct map contents after the following code?

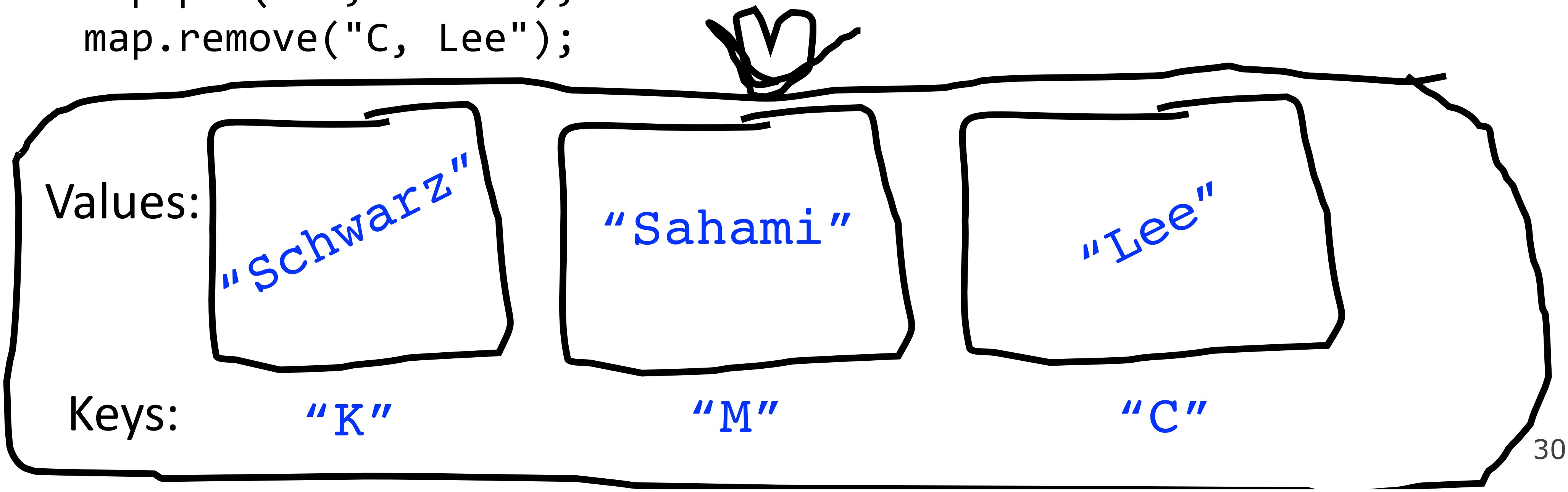
```
HashMap<String, String> map = new HashMap<>();
map.put("K", "Schwarz");
map.put("C", "Lee");
map.put("M", "Sahami");
map.put("M", "Stepp");
map.remove("Stepp");
map.remove("K");
map.put("J", "Cain");
map.remove("C, Lee");
```

```
A.  {C=Lee, J=Cain, M=Stepp, M=Sahami}
B.  {C=Lee, J=Cain, M=Stepp}
C.  {J=Cain M=Sahami, M=Stepp}
D.  {J=Cain, K=Schwarz, M=Sahami}
E.  other
```

Practice: Map Mystery

Q: What are the correct map contents after the following code?

```
HashMap<String, String> map = new HashMap<>();
map.put("K", "Schwarz");
map.put("C", "Lee");
map.put("M", "Sahami");
map.put("M", "Stepp");
map.remove("Stepp");
map.remove("K");
map.put("J", "Cain");
map.remove("C, Lee");
```



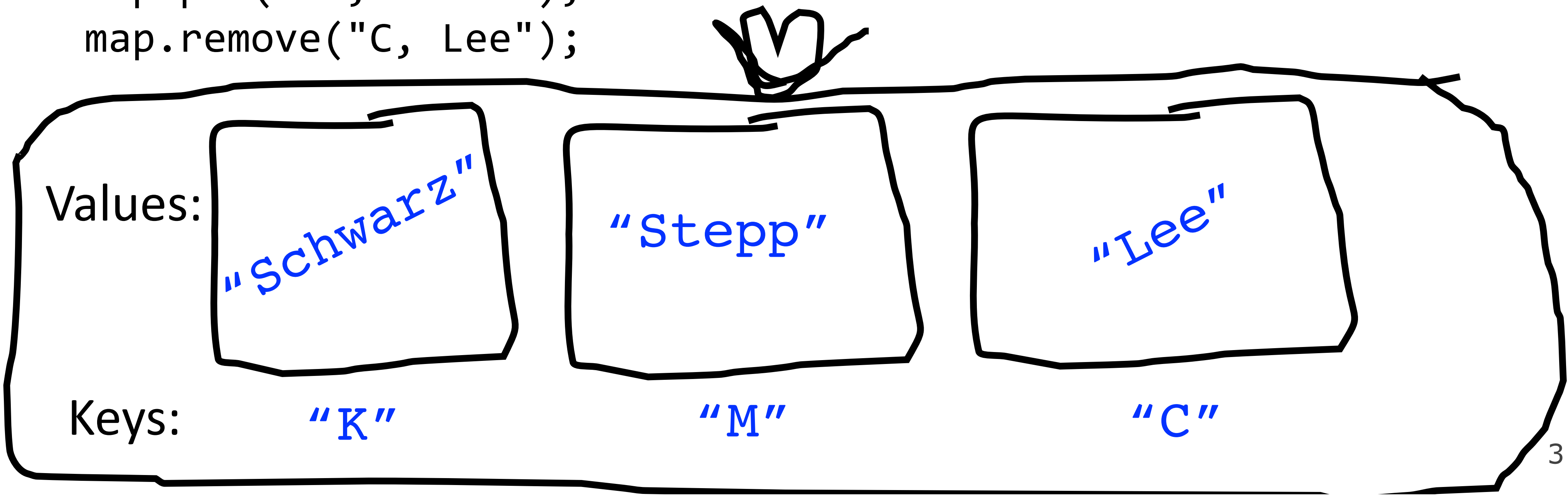
Values: “Schwarz” “Sahami” “Lee”

Keys: “K” “M” “C”

Practice: Map Mystery

Q: What are the correct map contents after the following code?

```
HashMap<String, String> map = new HashMap<>();
map.put("K", "Schwarz");
map.put("C", "Lee");
map.put("M", "Sahami");
map.put("M", "Stepp");
map.remove("Stepp");
map.remove("K");
map.put("J", "Cain");
map.remove("C, Lee");
```



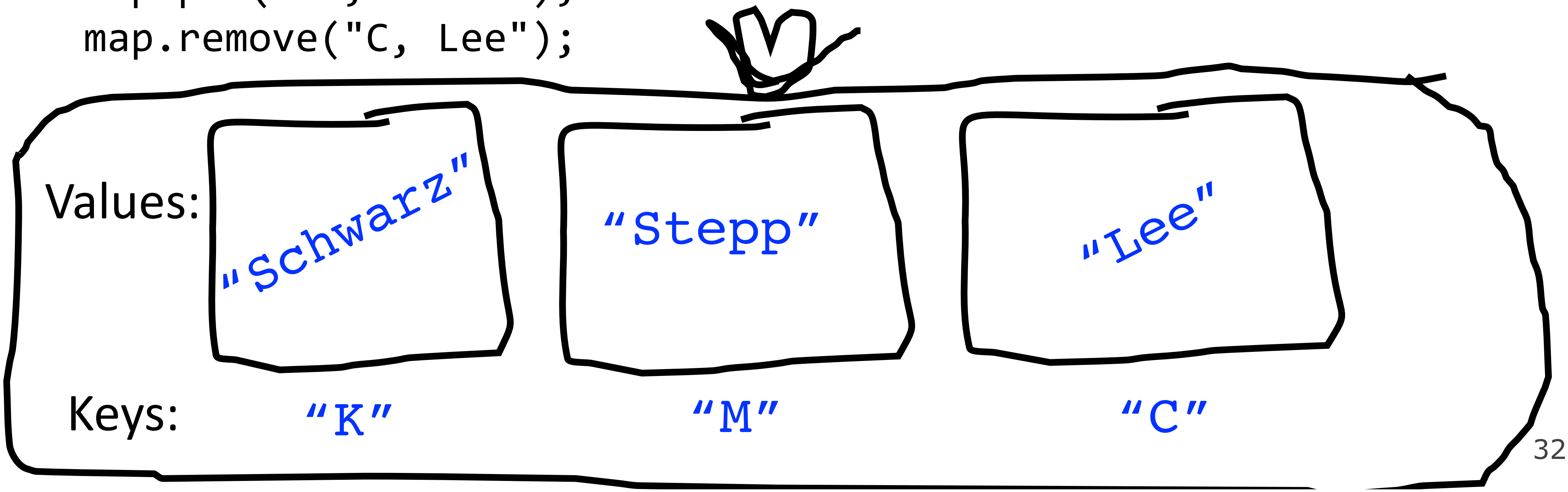
Values: "Schwarz" “Stepp” "Lee"

Keys: “K” “M” “C”

Practice: Map Mystery

Q: What are the correct map contents after the following code?

```
HashMap<String, String> map = new HashMap<>();
map.put("K", "Schwarz");
map.put("C", "Lee");
map.put("M", "Sahami");
map.put("M", "Stepp");
map.remove("Stepp");
map.remove("K");
map.put("J", "Cain");
map.remove("C, Lee");
```



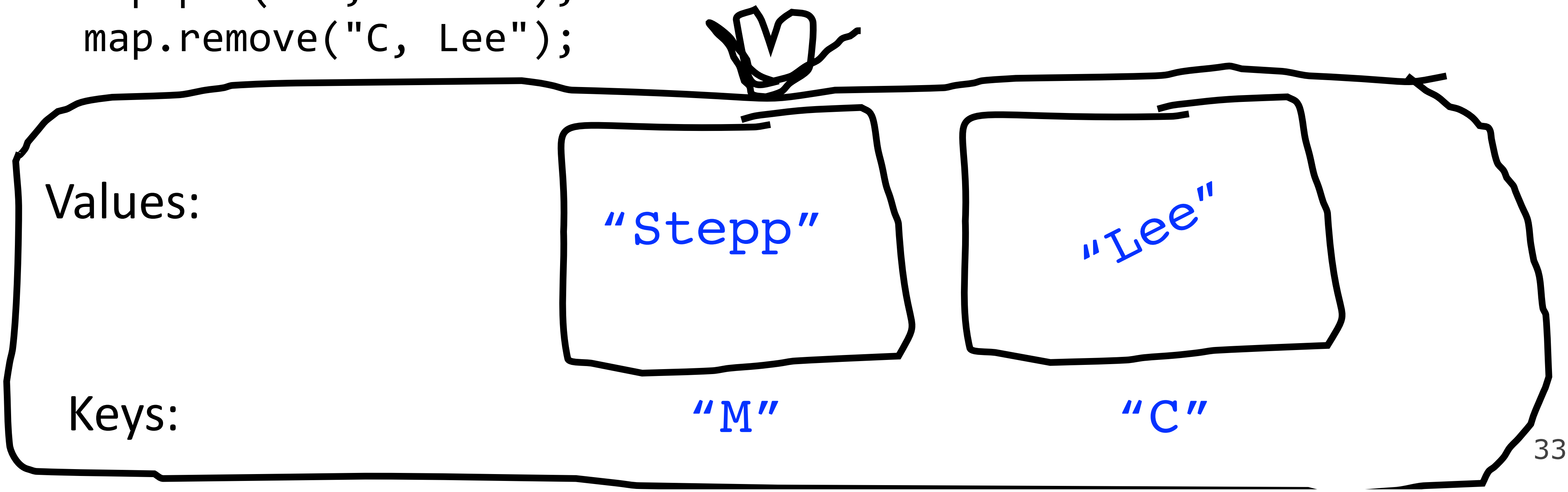
Values: "Schwarz" “Stepp” "Lee"

Keys: “K” “M” “C”

Practice: Map Mystery

Q: What are the correct map contents after the following code?

```
HashMap<String, String> map = new HashMap<>();
map.put("K", "Schwarz");
map.put("C", "Lee");
map.put("M", "Sahami");
map.put("M", "Stepp");
map.remove("Stepp");
map.remove("K");
map.put("J", "Cain");
map.remove("C, Lee");
```



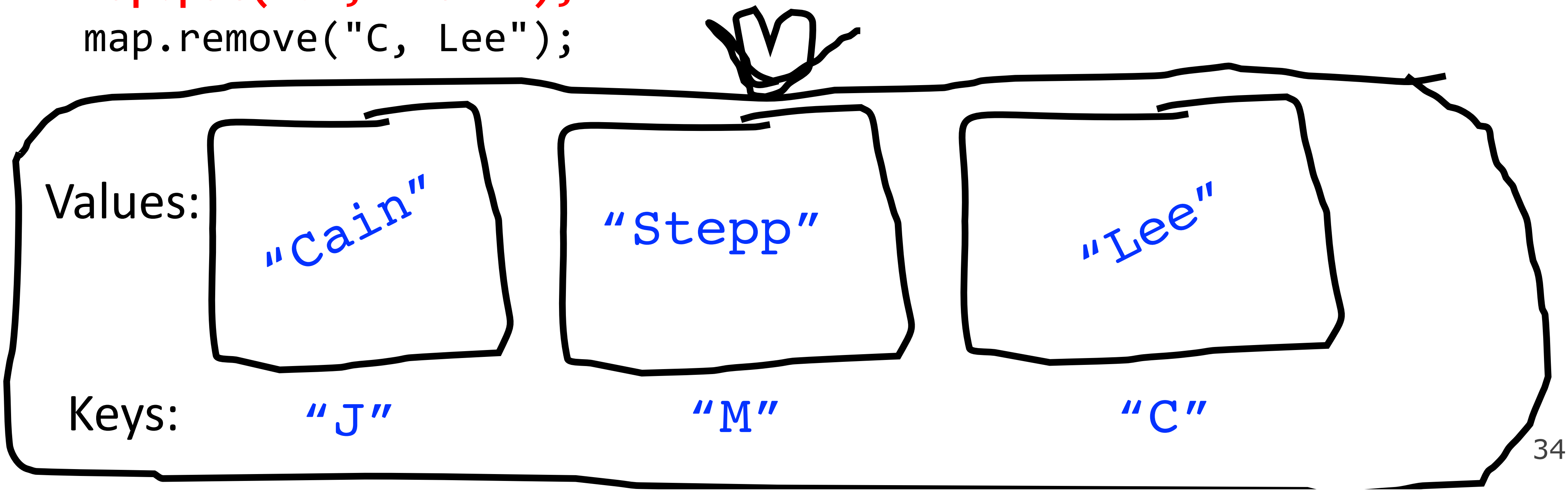
Values: “Stepp” “Lee”

Keys: “M” “C”

Practice: Map Mystery

Q: What are the correct map contents after the following code?

```
HashMap<String, String> map = new HashMap<>();
map.put("K", "Schwarz");
map.put("C", "Lee");
map.put("M", "Sahami");
map.put("M", "Stepp");
map.remove("Stepp");
map.remove("K");
map.put("J", "Cain");
map.remove("C, Lee");
```



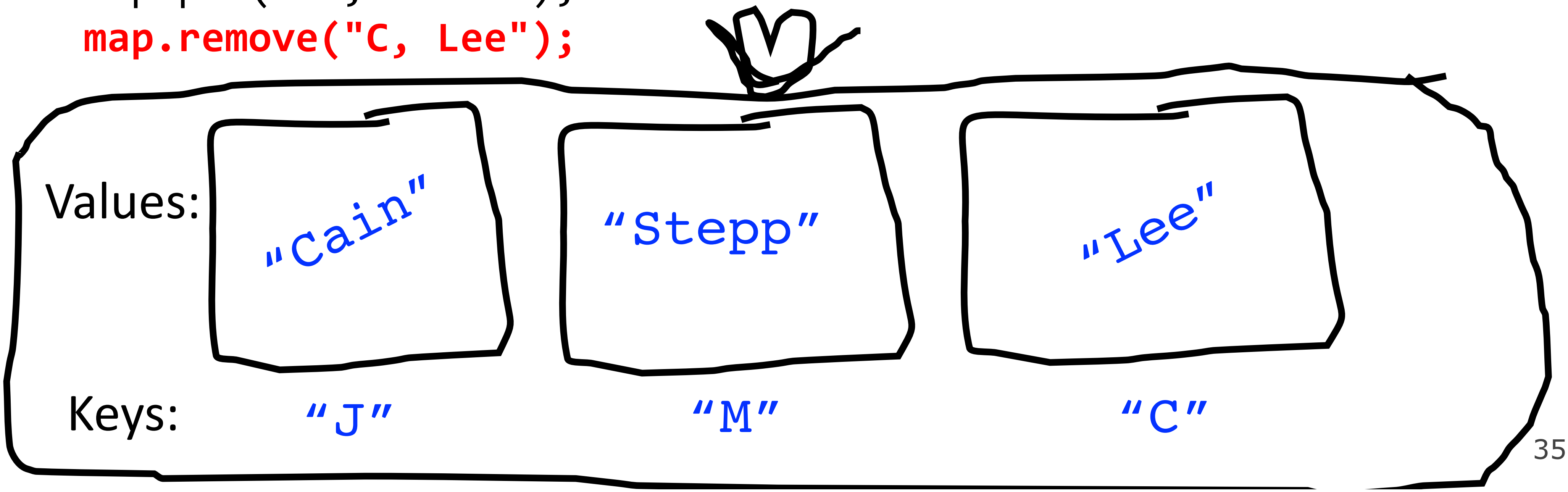
Values: "Cain" "Stepp" "Lee"

Keys: "J" "M" "C"

Practice: Map Mystery

Q: What are the correct map contents after the following code?

```
HashMap<String, String> map = new HashMap<>();
map.put("K", "Schwarz");
map.put("C", "Lee");
map.put("M", "Sahami");
map.put("M", "Stepp");
map.remove("Stepp");
map.remove("K");
map.put("J", "Cain");
map.remove("C, Lee");
```

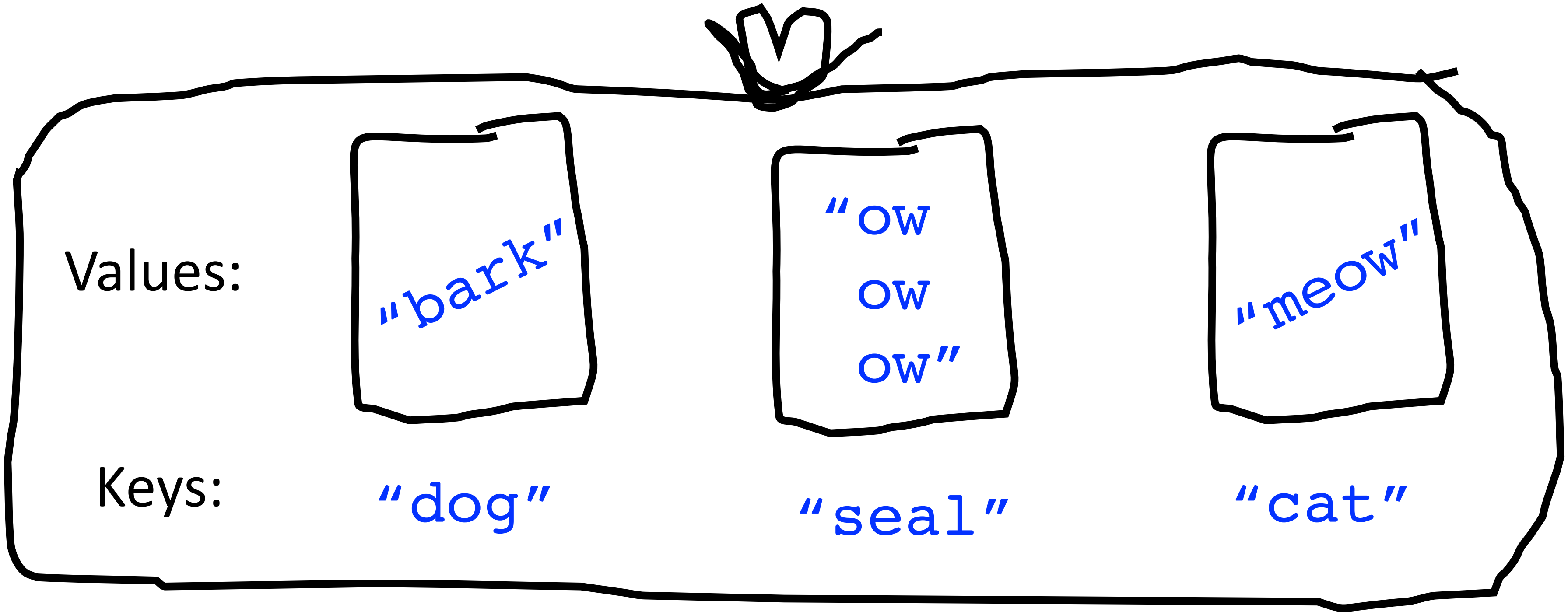


Values: "Cain" "Stepp" "Lee"

Keys: "J" "M" "C"

Iterating Over HashMaps

```
...
for (String key : map.keySet()) {
  String value = map.get(key);
  // do something with key/value pair...
}
// Keys occur in an unpredictable order!
```



Counting Exercise

- Write a program to count the number of occurrences of each unique word in a large text file (e.g. *Moby Dick*).
  - Allow the user to type a word and report how many times that word appeared in the book.
  - Report all words that appeared in the book at least 500 times.

- How can a **map** help us solve this problem?
  - Think about scanning over a file containing this input data:

```
To be or not to be or to be a bee not two bees ...
^
```

Practice: What's Trending?

- Social media can be used to monitor popular conversation topics.
- Write a program to count the frequency of **#hashtags** in tweets:
  - Read saved tweets from a large text file.
  - Report hashtags that occur at least 15 times.

- How can a **map** help us solve this problem?

Given these hashtags...

```
#stanford
#summer
#california
#stanford
```

We want to store...

```
"#stanford"   → 2
"#summer"     → 1
"#california" → 1
```