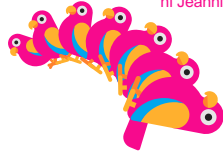


✦ No Laptop Monday! ✦

hi Jeannie

# Week07



- map (dictionary, table) interface

## WARMUP

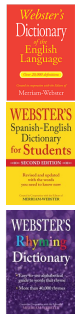
long, long, ago, many people owned a large, physical book called a ✦ dictionary ✦

- what was it used for?
- did you know there were different kinds of dictionaries?

[record lecture]

## real-world dictionaries

**note:** these are NOT hash maps  
(hash maps are NOT ordered)



```
Map<String, String> dictionary;  
// {Dog=A cute, four-legged mammal, ...}
```

```
Map<String, String> spanishToEnglishDictionary;  
// {Gato=Cat, Perro=Dog, ...}
```

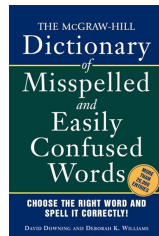
```
Map<String, List<String>> rhymingDictionary;  
// {Dog=[Bog, Cog, Frog], ...}
```

## common occurrence in ye olden times

- **child:** "how do you spell trebuchet?"
- **parent:** "look it up in the dictionary."

**note:** this advice made no sense

unless...



modern alternative



# maps

map interface

## map

- a **pair** is two things
- a **map** (**dictionary**, **table**) stores **key-value pairs**
  - in a map, you can **lookup** a key's value
  - a map in Java's standard library is `HashMap<KeyType, ValueType>`
    - `HashMap<String, ArrayList<String>> rhymingDictionary;`

analogy

let us imagine...



you have a three week long  
road trip planned to Bennington,  
but your prized parrot **Hans** is  
deathly afraid of driving

it's time to **put** **Hans** in the bird kennel



`map.put("Hans", 🦜)`

you return invigorated from a  
lovely trek to Bennington

it's time to **get** **Hans** from the bird kennel



here is  
Hans

squawk

Hans

`map.get("Hans")`

`HashMap<String, Bird>`



## map interface

- // **Put** (add, insert) a new key-value pair into the map.  
// **NOTE:** Can also be used to update a key's value.  
// **NOTE:** All keys are unique.  
`void put(KeyType key, ValueType value);`
- // **Get** (lookup) a key's value in the map.  
// **NOTE:** if key not in map, returns null  
`ValueType get(KeyType key);`
- // Get the (unordered) set of all keys.  
// **NOTE:** Use a for-each loop to iterate through this.  
// for (KeyType key : map.keySet()) { ... }  
`Set<KeyType> keySet();`

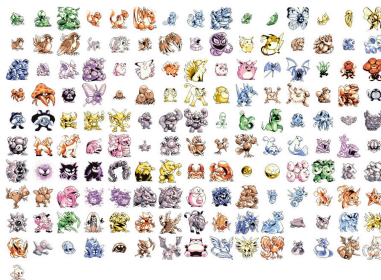
## simple example

```
HashMap<Character, Integer> map = new HashMap<>();  
PRINT(map); // {}  
  
map.put('a', 1);  
PRINT(map); // {a=1}  
  
map.put('b', 1);  
PRINT(map); // {a=1, b=1}  
  
map.put('b', 2);  
PRINT(map); // {a=1, b=2}
```

## motivating example

### Pokémon

- there are 151 **Pokémon**
- each has its own **number**
  1. Bulbasaur
  2. Ivysaur
  3. Venusaur
  4. Charmander
  5. ...



### number → name

```
1 → "Bulbasaur"  
2 → "Ivysaur"  
3 → "Venusaur"  
4 → "Charmander"  
...
```

what data structure  
should we use to  
implement this?

an array

the answer is always array

even during the map lecture

number → name (Option A)

```
String[] names = { "Bulbasaur", "Ivysaur", ... };
```

```
String name = names[number - 1]; // NOTE: Bulbasaur is #1
```

number → name (Option B)

```
String[] _names = { "Bulbasaur", "Ivysaur", ... };  
String getName(int number) {  
    return _names[number - 1]; // NOTE: Bulbasaur is #1  
}
```

```
String name = getName(number);
```

number → name (Option C)

```
// NOTE: Bulbasaur is #1  
String[] names = { "", "Bulbasaur", "Ivysaur", ... };
```

```
String name = names[number];
```

**lesson:** if you're mapping from  $A \rightarrow B$ ,  
and A is something like 0, 1, 2, 3, ...

...then you can just use an array 🤔👍  
(or an array list, if you don't know the length ahead of time)

buuut, if you're mapping from  $A \rightarrow B$ ,  
and A is *nothing* like 0, 1, 2, 3, ...

...then you're going to want a **map** 📖

name  $\rightarrow$  number

1. what **data structure** will you need? (hint: `HashMap<???, ???>`)
2. how will you **set it up** (how will you populate it)?
3. how will you **use it**?

```
"Bulbasaur"  -> 1
"Ivysaur"    -> 2
"Venusaur"   -> 3
"Charmander" -> 4
...
```

```
void put(KeyType key, ValueType value);
ValueType get(KeyType key);
```

name  $\rightarrow$  number

```
// data structures
HashMap<String, Integer> numberFromName;
```

```
// setup
String[] names = { "Bulbasaur", "Ivysaur", ... };
numberFromName = new HashMap<>();
for (int i = 0; i < names.length; ++i) {
    // Bulbasaur = 1
    numberFromName.put(names[i], i + 1);
}
```

```
// usage
String name = ...;
int number = numberFromName.get(name);
```

[let's implement `numberFromName`]

TODO (Jim): use `keySet()` method in test code

## PokemonExample


```
import java.util.*;
class Main extends Cow {
    public static void main(String[] args) {
        // setup
        HashMap<String, Integer> numberFromName = new HashMap<>();
        String[] names = { "Bulbasaur", "Ivysaur", "Venusaur", ... };
        for (int number = 0; number < names.length; ++number) {
            numberFromName.put(names[number], number + 1);
        }

        // usage
        for (String name : numberFromName.keySet()) {
            PRINT(name + " -> " + numberFromName.get(name));
        }
    }
}
```

```
Mewtwo -> 150
Raichu -> 26
Arbok -> 24
Kabuto -> 140
Charmander -> 4
Charmeleon -> 5
Shellder -> 90
Fearow -> 22
Kangaskhan -> 115
Mankey -> 56
Dodrio -> 85
Vaporeon -> 134
Golem -> 76
Marowak -> 105
Pikachu -> 25
Raticate -> 20
Kadabra -> 64
Primeape -> 57
Venomoth -> 49
Magikarp -> 129
Pidgeotto -> 17
Slowbro -> 80
Gastly -> 92
Magiketon -> 82
```


**reminder:** hash maps are NOT ordered  
(even though we put Pokemon into the hash map in order,  
the hash map did NOT store them in order)

# maps in other languages

 Python has a built-in map (dictionary)

```
numbers = {}
numbers["Bulbasaur"] = 1
numbers["Ivysaur"] = 2
print(numbers["Bulbasaur"]) # 1
print(numbers) # {"Bulbasaur": 1, "Ivysaur": 2}
```

```
# hmm...
map = {}
map[True] = False
map['Jim'] = 3
map[-1] = map
print(map) # {True: False, 'Jim': 3, -1: {...}}
```

 Lua's *only* data structure is a map (table)

- why?
- how?
  - what about like...arrays?

```
names = {}
names[1] = "Bulbasaur" -- okay fine, i guess Lua also has strings
names[2] = "Ivysaur"
print(names[1]) -- Bulbasaur
```

## 11 – Data Structures

Tables in Lua are not a data structure; they are *the* data structure. All structures that other languages offer---arrays, records, lists, queues, sets---are represented with tables in Lua. More to the point, tables implement all these structures efficiently.

In traditional languages, such as C and Pascal, we implement most data structures with arrays and lists (where lists = records + pointers). Although we can implement arrays and lists using Lua tables (and sometimes we do that), tables are more powerful than arrays and lists; many algorithms are simplified to the point of triviality with the use of tables. For instance, you seldom write a search in Lua, because tables offer direct access to any type.

It takes a while to learn how to use tables efficiently. Here, we will show how you can implement typical data structures with tables and will provide some examples of their use. We will start with arrays and lists, not because we need them for the other structures, but because most programmers are already familiar with them. We have already seen the basics of this material in our chapters about the language, but I will repeat it here for completeness.

💎 Wednesday! 💎

## Week07b

- hash functions
- hash map with separate chaining

**WARMUP:** what does this print?

```
HashMap<String, String> map = new HashMap<>();
map.put("hello", "world");
map.put(map.get("hello"), "on fire");
PRINT(map.get("world"));
PRINT(map.get("on fire"));
```



[record lecture]

## map interface

- `// Put (add, insert) a new key-value pair into the map.`  
`// NOTE: Can also be used to update a key's value.`  
`// NOTE: All keys are unique.`  
`void put(KeyType key, ValueType value);`
- `// Get (lookup) a key's value in the map.`  
`// NOTE: if key not in map, returns null`  
`ValueType get(KeyType key);`
- `// Get the (unordered) set of all keys.`  
`// NOTE: Use a for-each loop to iterate through this.`  
`// for (KeyType key : map.keySet()) { ... }`  
`Set<KeyType> keySet();`

## motivation

let's get motivated

### History [ edit ]

The idea of hashing arose independently in different places. In January 1953, [Hans Peter Luhn](#) wrote an internal IBM memorandum that used hashing with chaining. The first example of [open addressing](#) was proposed by A. D. Linh, building on Luhn's memorandum.<sup>[4]:547</sup> Around the same time, [Gene Amdahl](#), [Elaine M. McGraw](#), [Nathaniel Rochester](#), and [Arthur Samuel](#) of IBM Research implemented hashing for the [IBM 701 assembler](#).<sup>[7]:124</sup> Open addressing with linear probing is credited to Amdahl, although [Andrey Ershov](#) independently had the same idea.<sup>[7]:124–125</sup> The term "open addressing" was coined by [W. Wesley Peterson](#) on his article which discusses the problem of search in large files.<sup>[8]:15</sup>

The first [published](#) work on hashing with chaining is credited to [Arnold Dumey](#), who discussed the idea of using remainder modulo a prime as a hash function.<sup>[9]:15</sup> The word "hashing" was first published in an article by Robert Morris.<sup>[7]:126</sup> A [theoretical analysis](#) of linear probing was submitted originally by Konheim and Weiss.<sup>[8]:15</sup>

so basically...





in ~1953 someone realized that if you combined an array, the %, and something called a "hash function," you could implement a map that ran in  $O(1)$ ...



let's learn how this works 🤔👍

## review: % vs. MODULO (Math.floorMod)

$$\begin{array}{r} 7 \% 3 \\ 3 \overline{) 7} \\ \underline{-6} \\ 1 \end{array}$$

 $-7 \% 3$ $-2R-1$ $3 \overline{) -7}$ $\underline{-6}$ $-1$	 $-7 \% 3$  <code>Math.floorMod(-7, 3)</code>  <code>MODULO(-7, 3)</code> $-1 + 3 = 2$
---	---

*Handwritten notes in the first column:*  
A blue arrow points from  $-7 - (-6)$  to  $-1$ .  
A green arrow points from  $-1$  to  $-1 + 3 = 2$ .

%

- `x % y` returns the **remainder** of (`x / y`) and is read "x **modulo** y"
- `int foo = 17 % 5; // 2` ("17 divided by 5 is 3 remainder 2")
- ⚠️ % probably doesn't do what you expect for negative numbers;  
if `x` can be negative, use `Math.floorMod(x, y)` instead
- `5 % 3 // 2`
- `4 % 3 // 1`
- `3 % 3 // 0`
- `2 % 3 // 2`
- `1 % 3 // 1`
- `0 % 3 // 0`
- `-1 % 3 // -1` WAIT WHAT 😱
- `MODULO(-1, 3) // 2` 🤔👍

MODULO(x, y)

- `MODULO( 5, 3) // 2`
- `MODULO( 4, 3) // 1`
- `MODULO( 3, 3) // 0`
- `MODULO( 2, 3) // 2`
- `MODULO( 1, 3) // 1`
- `MODULO( 0, 3) // 0`
- `MODULO(-1, 3) // 2`
- `MODULO(-2, 3) // 1`
- `MODULO(-3, 3) // 0`
- `MODULO(-4, 3) // 2`
- `MODULO(-5, 3) // 1`
- `MODULO(-6, 3) // 0`
- `MODULO(-7, 3) // 2`

# hash function

# hash function

## hash function

- a **hash function** takes some data and returns an integer called the data's **hash code** (hash, hash value)
  - this is called "**hashing**" the data
  - 🚩 a hash function **MUST** be **deterministic**
    - given the same data, a hash function **MUST** return the same value every time it is called
- a **hash collision** (hash clash) happens when two different pieces of data have the same hash code
  - a *good* hash function has very few collisions

## hash function

- 
- 
- 🚩 a hash function **MUST** be **deterministic**
  -
- 
- a *good* hash function has very few collisions

## let's play...is it a hash function?

- `int foo(String a) { return a.length(); }`
  - **yes, though it's bad.**
    - **note:** "foo" and "bar" collide
- `int bar(String a) { return 0; }`
  - **yes, though it's so bad.**
    - **note:** everything collides
- `int baz(String a) { return random.nextInt(); }`
  - **no, function is not deterministic.**

## From the ASCII table...

Symbol	Decimal	Binary
A	65	01000001
B	66	01000010
C	67	01000011
D	68	01000100
E	69	01000101
F	70	01000110
G	71	01000111
H	72	01001000
I	73	01001001
J	74	01001010
K	75	01001011
L	76	01001100
M	77	01001101
N	78	01001110
O	79	01001111
P	80	01010000
Q	81	01010001
R	82	01010010
S	83	01010011
T	84	01010100
U	85	01010101
V	86	01010110
W	87	01010111
X	88	01011000
Y	89	01011001
Z	90	01011010

Symbol	Decimal	Binary
a	97	01100001
b	98	01100010
c	99	01100011
d	100	01100100
e	101	01100101
f	102	01100110
g	103	01100111
h	104	01101000
i	105	01101001
j	106	01101010
k	107	01101011
l	108	01101100
m	109	01101101
n	110	01101110
o	111	01101111
p	112	01110000
q	113	01110001
r	114	01110010
s	115	01110011
t	116	01110100
u	117	01110101
v	118	01110110
w	119	01110111
x	120	01111000
y	121	01111001
z	122	01111010

let's play...do they collide?

```
int hash(String string) {
    int result = 0;
    for (int i = 0; i < string.length(); ++i) {
        result += string.charAt(i); // NOTE: chars are integers
    }
    return result;
}
```

- "a" and "z"?
- "cat" and "dog"?
- "abc" and "ABC"?
- "iamlordvoldemort" and "tommarvoloriddle"?

## Java's built-in hashCode()

### hash function

- all Java objects have a built-in hash function called hashCode()
- hashCode() can be negative!

```
PRINT( "Hans".hashCode()); // 2241694
PRINT( "Gary".hashCode()); // 2212033
PRINT("Kahoot".hashCode()); // -2054990942
PRINT("Kahoot".hashCode()); // -2054990942
```

### hash function

- often, you want to use a hash code to index into an array
  - ✦ `MODULO(object.hashCode(), array.length)`

```
int N = 10;
PRINT(MODULO( "Hans".hashCode(), N)); // 4
PRINT(MODULO( "Gary".hashCode(), N)); // 3
PRINT(MODULO("Kahoot".hashCode(), N)); // 8
PRINT(MODULO("Kahoot".hashCode(), N)); // 8
```

```
/* String.java — immutable character sequences; the object of string literals
Copyright (C) 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005
Free Software Foundation, Inc.
```

```
/**
 * Computes the hashCode for this String. This is done with int arithmetic,
 * where ** represents exponentiation, by this formula:<br>
 * <code>s[0]*31**(n-1) + s[1]*31**(n-2) + ... + s[n-1]</code>.
 *
 * @return hashCode value of this String
 */
public int hashCode()
{
    if (cachedHashCode != 0)
        return cachedHashCode;

    // Compute the hash code using a local variable to be reentrant.
    int hashCode = 0;
    int limit = count + offset;
    for (int i = offset; i < limit; i++)
        hashCode = hashCode * 31 + value[i];
    return cachedHashCode = hashCode;
}
```

## hash maps

## hash map

- a **hash map (hash table)** is a great way to implement a map
- hash maps are implemented using an **array** and a **hash function**
- hash maps are FAST
  - `put(key, value)` is  $O(1)$  time (ish) 😊 FAST
  - `get(key)` is  $O(1)$  time (ish) 😊 FAST
- there are many different "flavors" of hash maps
  - separate chaining (today)
  - open addressing (friday)
  - and soo much more!

### Cuckoo hashing

Article Talk

From Wikipedia, the free encyclopedia

**Cuckoo hashing** is a scheme in computer programming for resolving hash collisions of values of hash functions in a table, with worst-case constant lookup time. The name derives from the behavior of some species of cuckoo, where the cuckoo chick pushes the other eggs or young out of the nest when it hatches in a variation of the behavior referred to as brood parasitism; analogously, inserting a new key into a cuckoo hashing table may push an older key to a different location in the table.



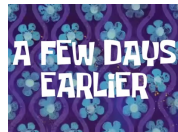
## separate chaining

### separate chaining

- **separate chaining** is one way of implementing a hash map
- key-value pairs are stored in an array of **buckets**
- one good choice of bucket is an array list
  - `ArrayList<KeyValuePair> bucket;`
- in practice, you don't have to use an array list; you can use linked lists, binary search trees, or some mixture of the two (this is actually what Java does)

## analogy

previously, on *Hans the Parrot*



you have a three week long  
road trip planned to Bennington,  
but your prized parrot **Hans** is  
deathly afraid of driving

it's time to **put** **Hans** in the bird kennel



`map.put("Hans", 🦜)`

you return invigorated from a  
lovely trek to Bennington

it's time to **get** **Hans** from the bird kennel



here is  
Hans

squack

Hans

`map.get("Hans")`

`HashMap<String, Bird>`



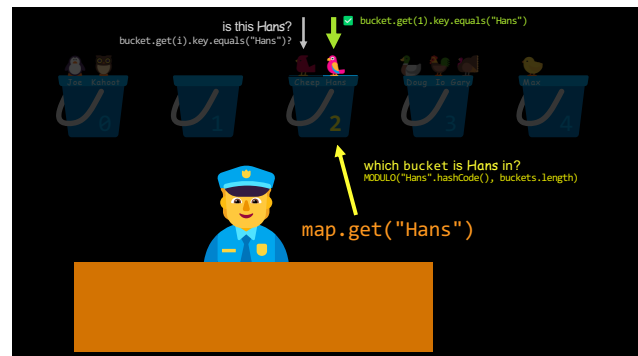
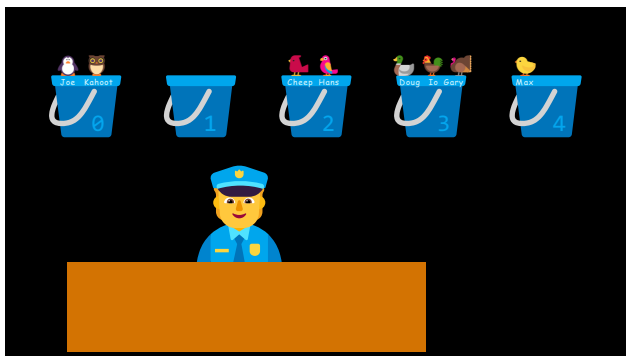
now, let us imagine again...



in a thrilling twist, it turns out  
that the Bennington trip was  
secretly cover for Agent **Hans**'s  
corporate espionage of  
the bird kennel

(rumor had it they were keeping the birds in *buckets* 🐦)

it's time to **get Hans** back to the hideout



```
int[] arrayOfInts;  
ArrayList<KeyValuePair>[] buckets;
```

```
// Bucket[] buckets  
ArrayList<KeyValuePair>[] buckets;
```

```
ArrayList<KeyValuePair>[] buckets;
```

- to **construct** the hash map
  - create a new buckets array
  - iterate through the array and create a new (empty) bucket in each slot

```
ArrayList<KeyValuePair>[] buckets;
```

- to **put** a key-value pair into the hash map (or update an existing key's value)
  - // hash key to get bucket index
    - // NOTE: hashCode() can be negative, so use MODULO
    - bucketIndex = MODULO(key.hashCode(), buckets.length)
  - // if key in the map, update corresponding value
  - iterate through that bucket
    - if you find a key-value pair with matching key...
      - update its value
      - return;
  - // otherwise, add key-value pair to map
  - if you did NOT find any key-value pair with matching key
    - add the key-value pair to the bucket

```
ArrayList<KeyValuePair>[] buckets;
```

- to **get** a key's value from the hash map
  - // hash key to get bucket index
    - bucketIndex = MODULO(key.hashCode(), buckets.length)
  - // if key in the map, return corresponding value
  - iterate through that bucket
    - if you find a key-value pair with matching key...
      - return its value
  - // otherwise, return null
  - if you did NOT find any key-value pair with matching key
    - return null;

## hash map questions

how big should we make the array?

why would need to resize (grow) a hash map?  
how do we resize a hash map?  
why might this be slow?