Java Data Structures

Tips and Tricks - Part 2

Micah Stairs

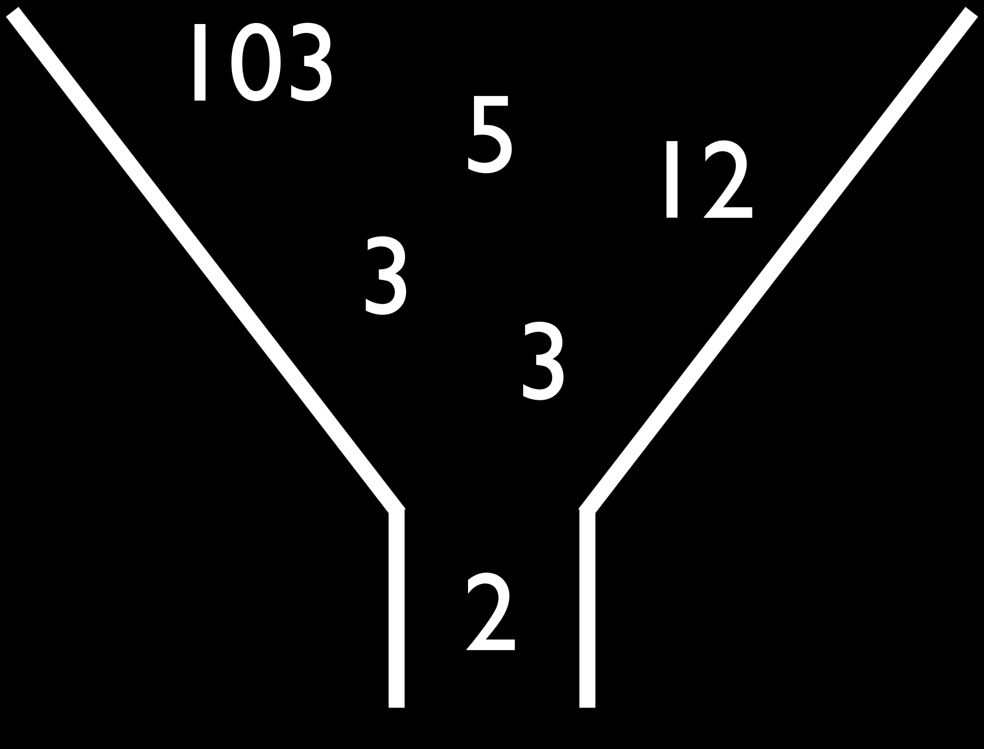
Outline

- Priority Queues
  - Comparators / Comparable
- Maps
  - Hashing
- Solution Sketch for a Kattis Problem

Priority Queues

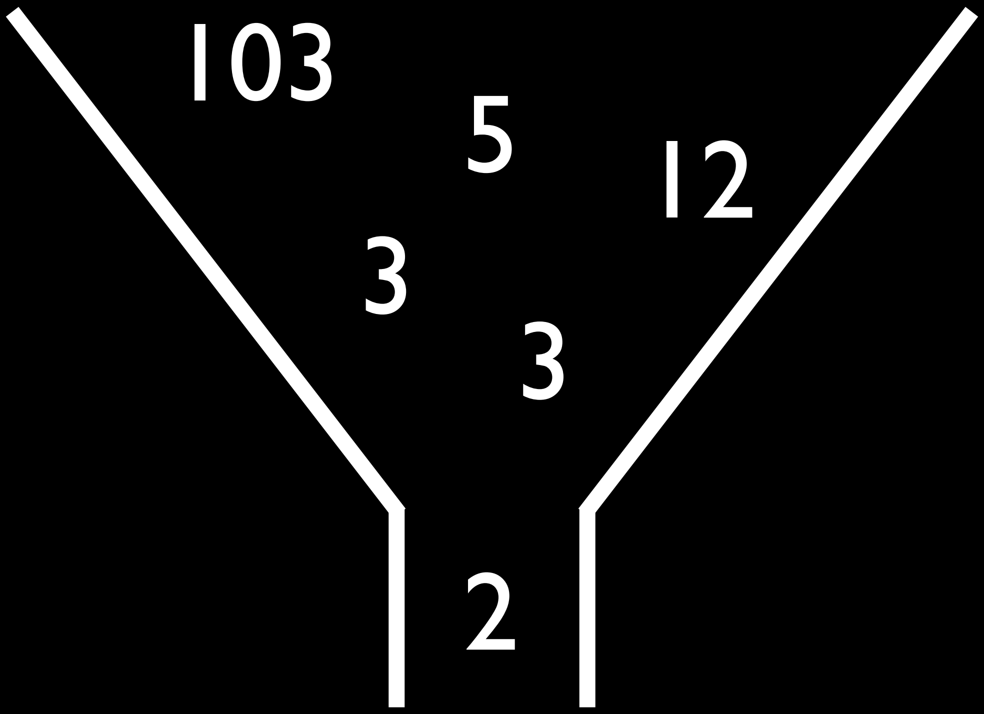
Priority Queues

A container in which elements with a higher priority are removed first.



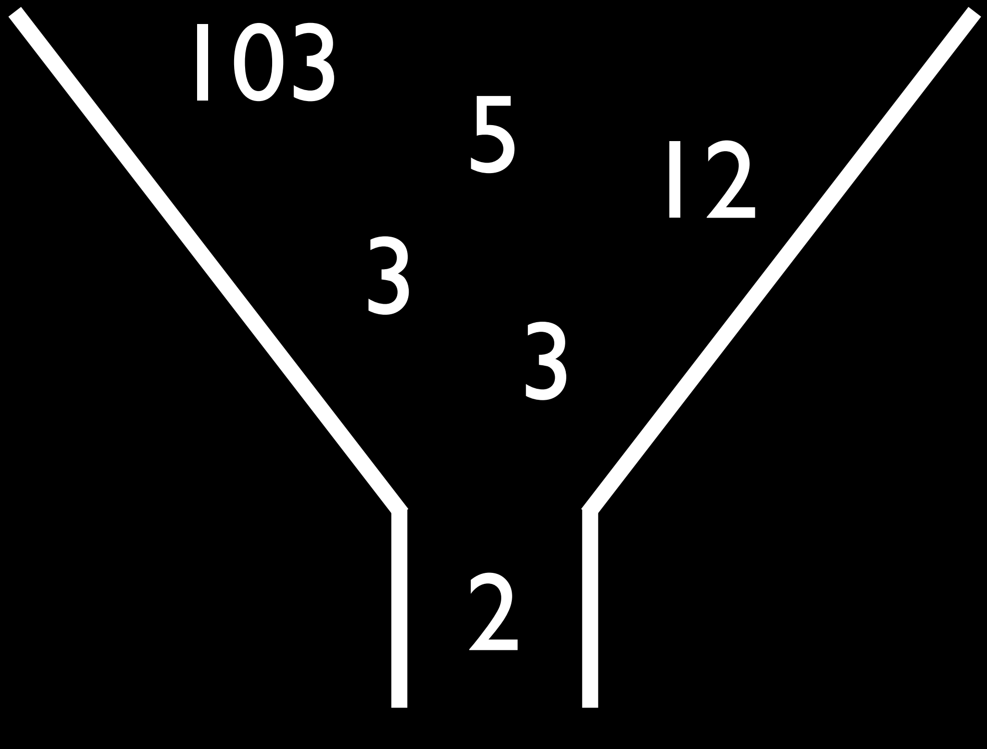
Priority Queues

By default, the elements are sorted by their natural ordering (where smaller elements have a higher priority).



Priority Queues

In competitive programming, this data structure is commonly used for Dijkstra's Shortest Path Algorithm, but has other uses as well.

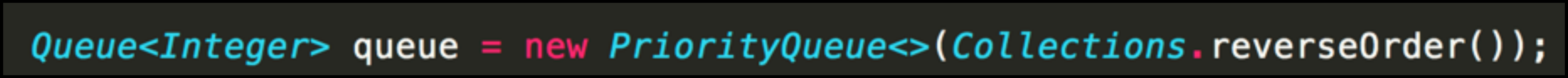


Priority Queues

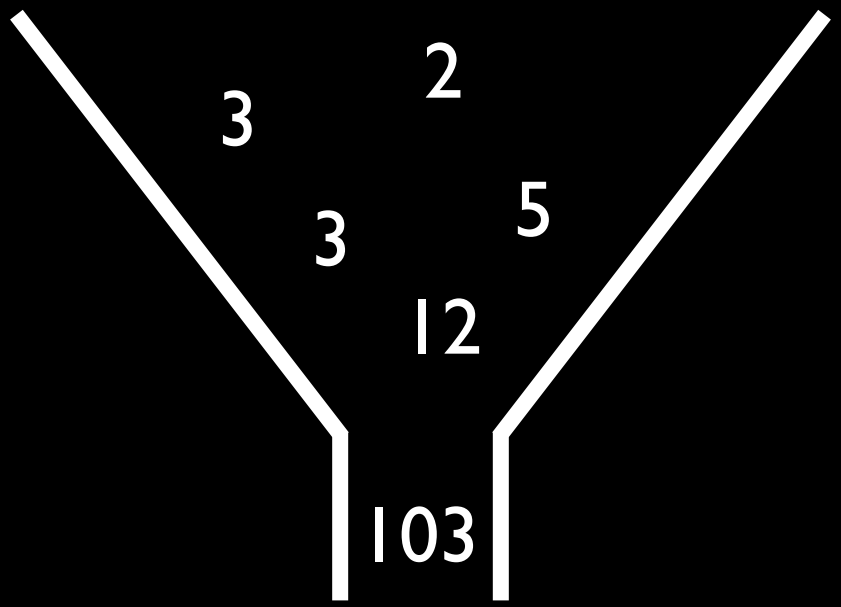
Operation	Time Complexity
peeking at top element	O(1)
removing top element	O(log n)
adding new element	O(log n)
checking for element	O(n)
removing specific element	O(n)

Priority Queues

If we want to reverse the natural ordering of our elements in the priority queue, this comparator is actually part of the Java API.

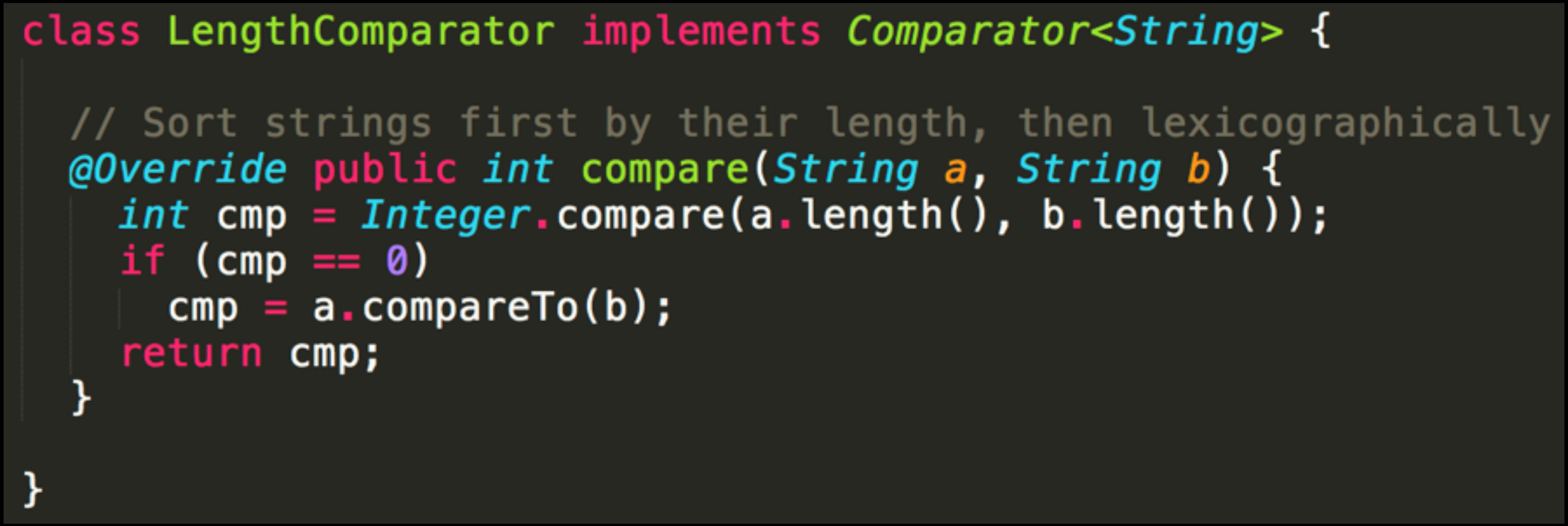


```
Queue<Integer> queue = new PriorityQueue<>(Collections.reverseOrder());
```



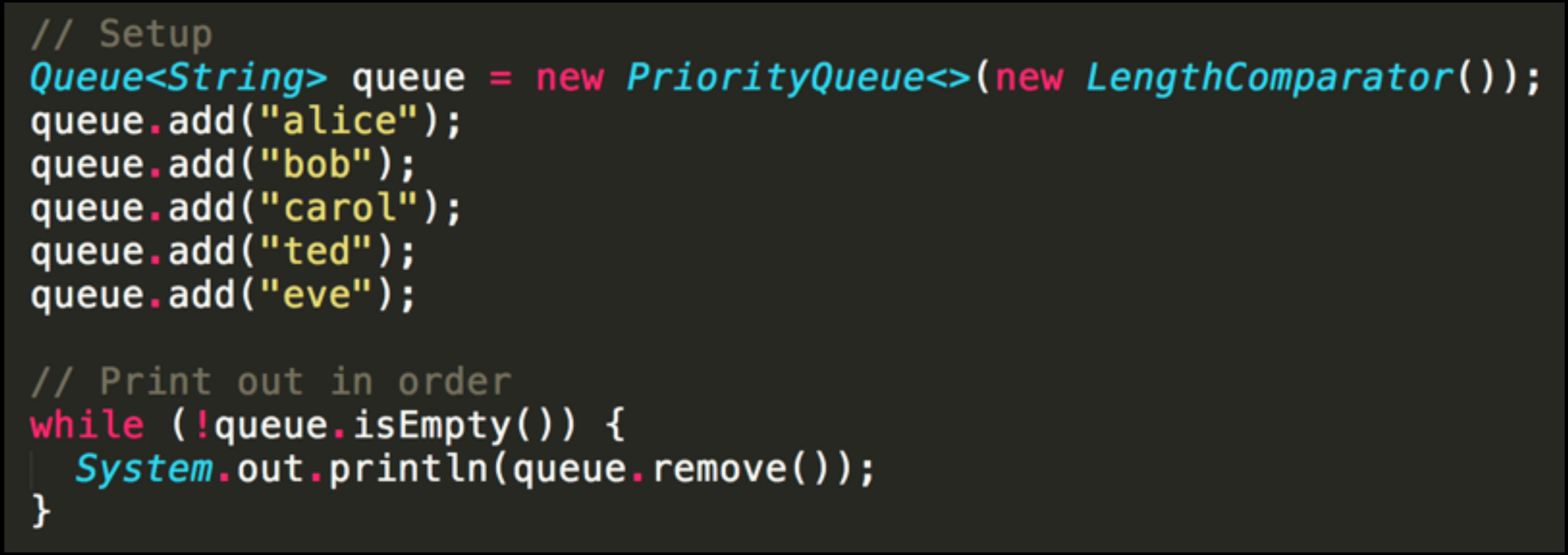
Priority Queues

Strings are normally sorted lexicographically, however, we can write a *Comparator* to define an alternate ordering for our Priority Queue.



```java
class LengthComparator implements Comparator<String> {

  // Sort strings first by their length, then lexicographically
  @Override public int compare(String a, String b) {
    int cmp = Integer.compare(a.length(), b.length());
    if (cmp == 0)
      cmp = a.compareTo(b);
    return cmp;
  }

}
```

Priority Queues



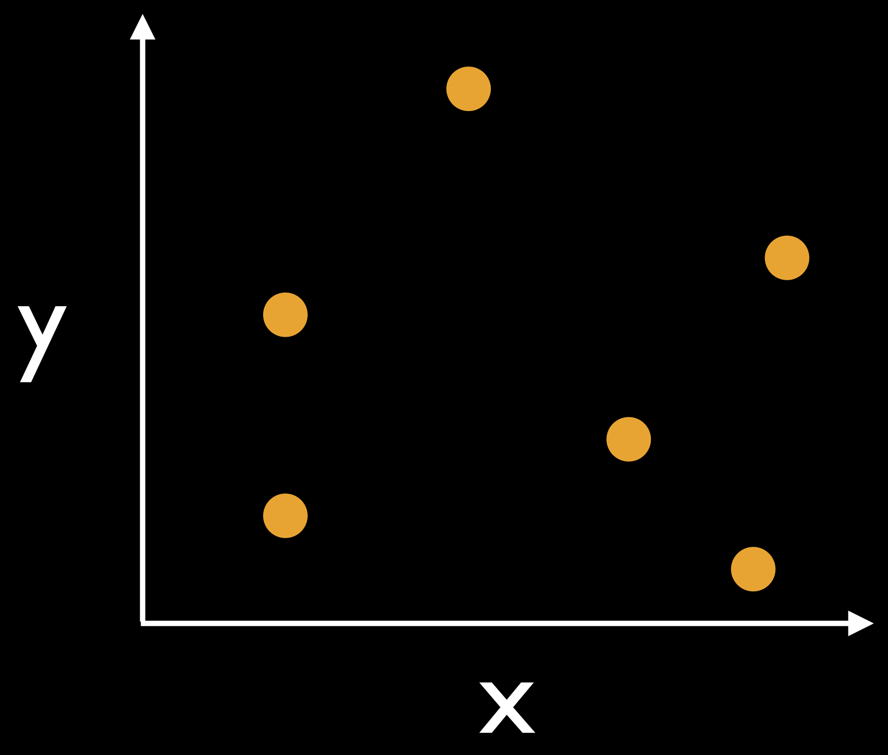
```java
// Setup
Queue<String> queue = new PriorityQueue<>(new LengthComparator());
queue.add("alice");
queue.add("bob");
queue.add("carol");
queue.add("ted");
queue.add("eve");

// Print out in order
while (!queue.isEmpty()) {
  System.out.println(queue.remove());
}
```

Outputs:

```
bob
eve
ted
alice
carol
```

Priority Queues

In computational geometry, it is common to sort (x,y) coordinates initially by one coordinate and then by the other. Writing a *Comparator* would allow you to achieve this.

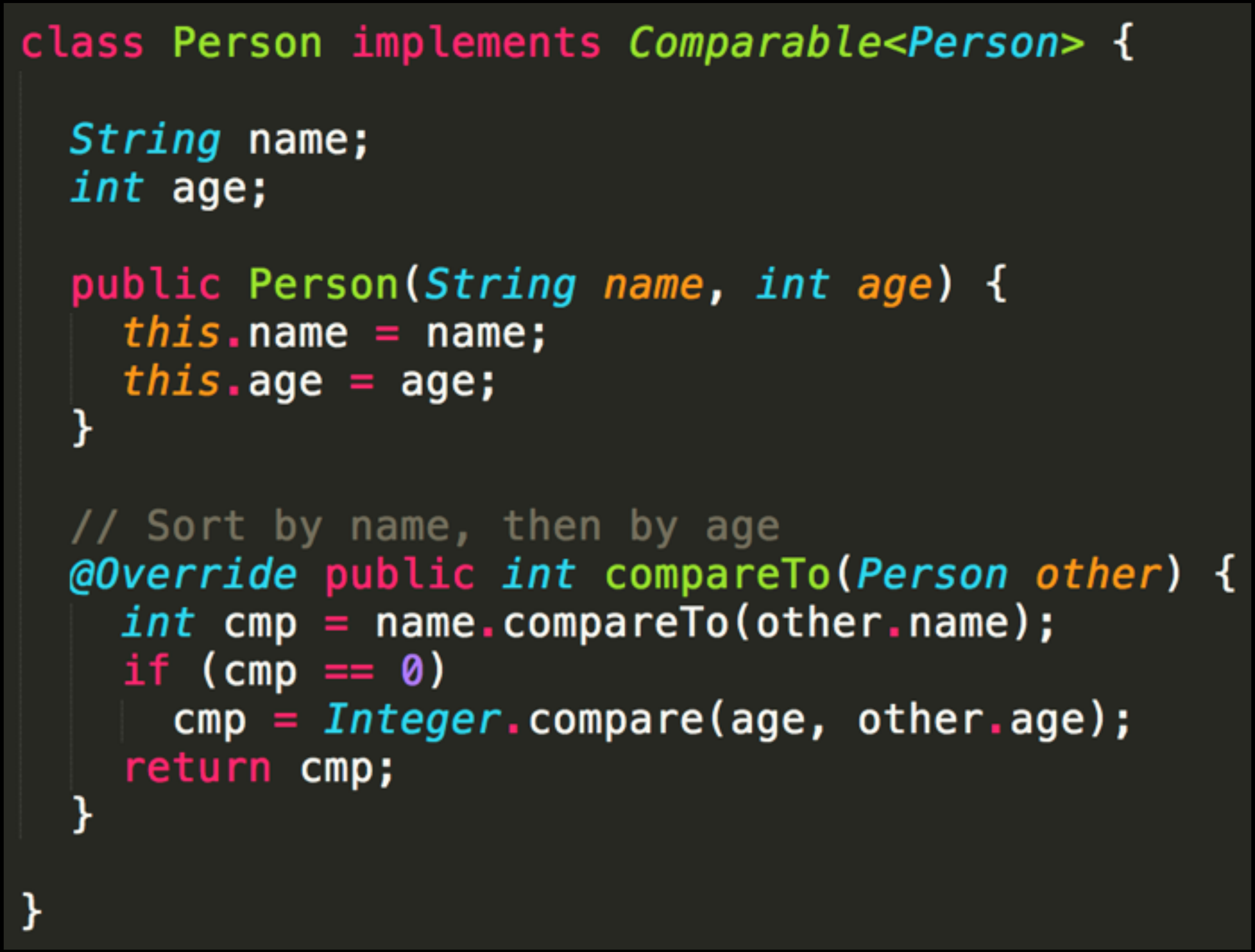


Priority Queues

If you write your own class you can also implement the Comparable interface, which defines the ordering of your class.

This ordering will be used by default in Priority Queues, Arrays.sort(), Collections.sort(), etc.

Priority Queues



```java
class Person implements Comparable<Person> {

  String name;
  int age;

  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }

  // Sort by name, then by age
  @Override public int compareTo(Person other) {
    int cmp = name.compareTo(other.name);
    if (cmp == 0)
      cmp = Integer.compare(age, other.age);
    return cmp;
  }

}
```

Priority Queues

Pros:

- Easily finds the top element.
- Can easily customize ordering of elements.

Cons:

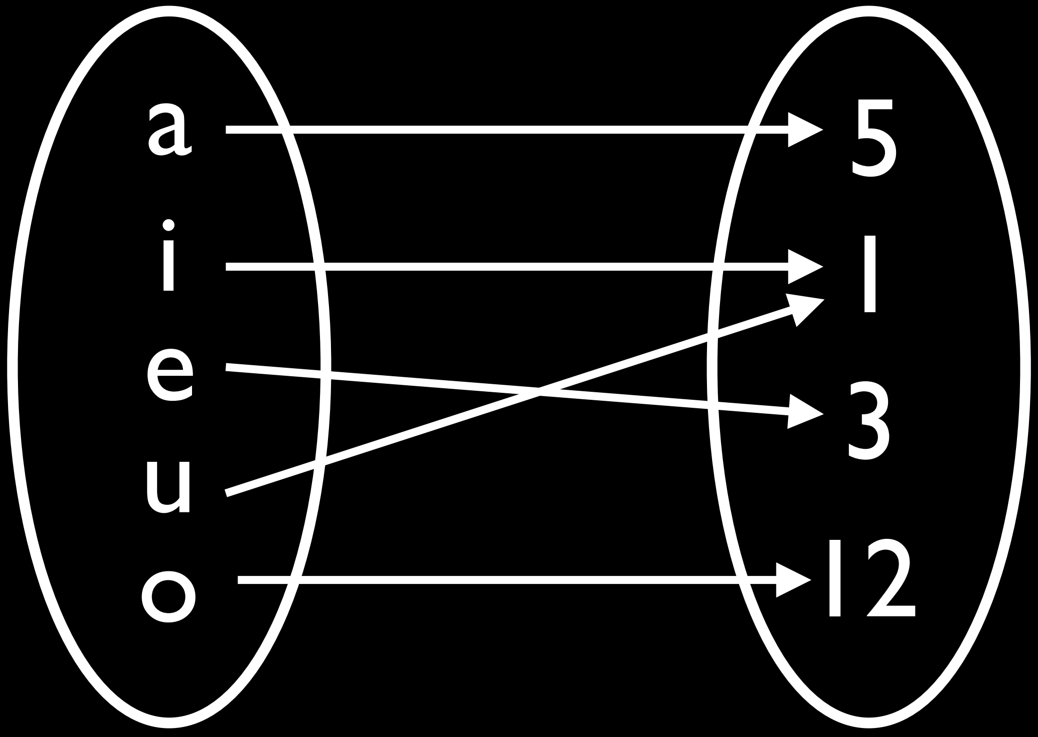
- Removing a specific element or checking to see if it contains a specific element is inefficient.

Maps

Maps

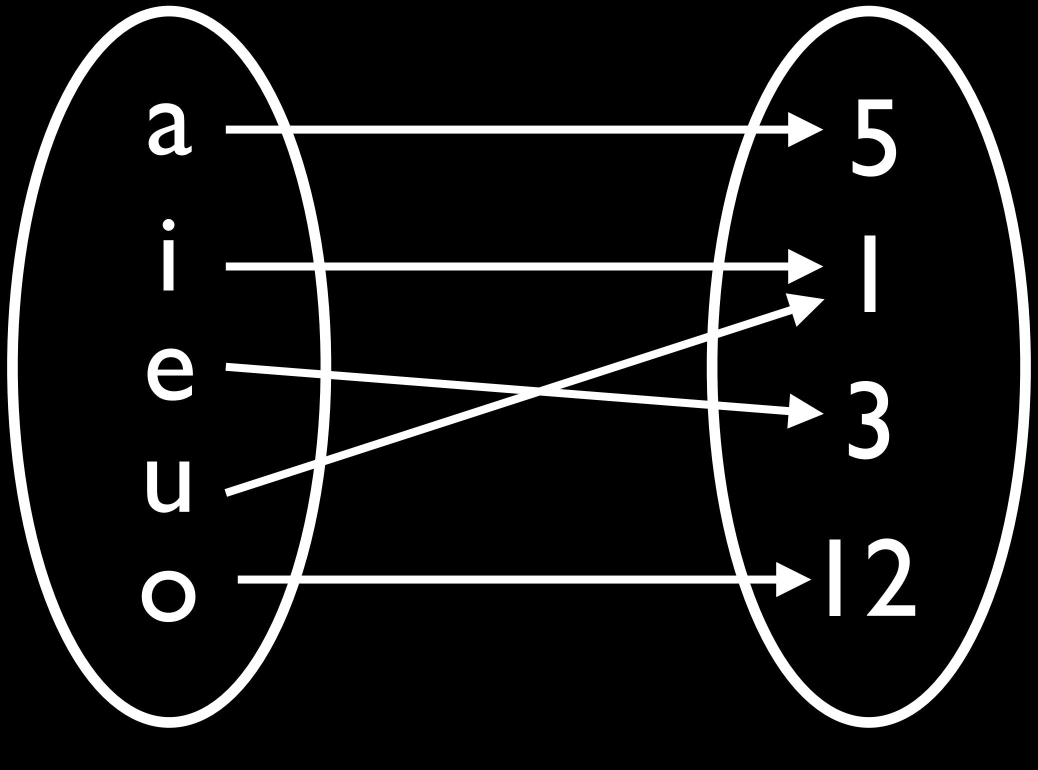
A container in which unique keys are mapped to values.

(**Note:** In Python, this is called a "dictionary")



Maps

There are two main implementations of the *Map* interface. *HashMap*'s keys are not sorted, while *TreeMap*'s keys are.

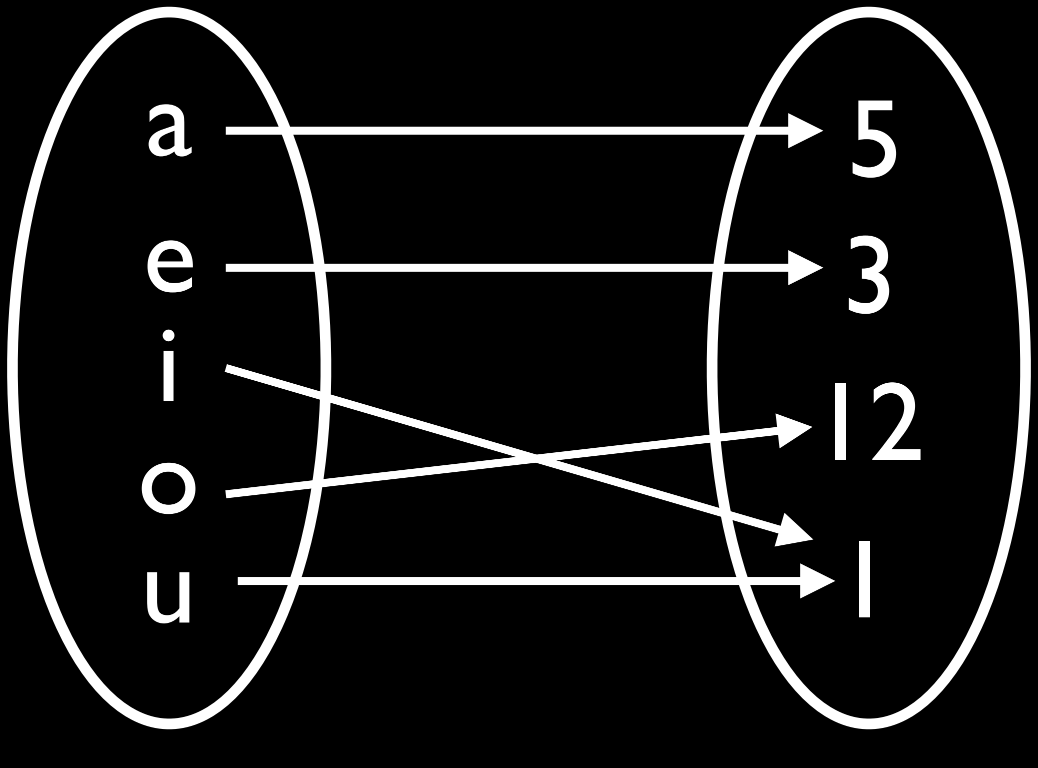


Maps

Operation	HashMap	TreeMap
put()	expected O(1)	O(log n)
containsKey()	expected O(1)	O(log n)
removeKey()	expected O(1)	O(log n)
containsValue()	O(n)	O(n)
removeValue()	O(n)	O(n)

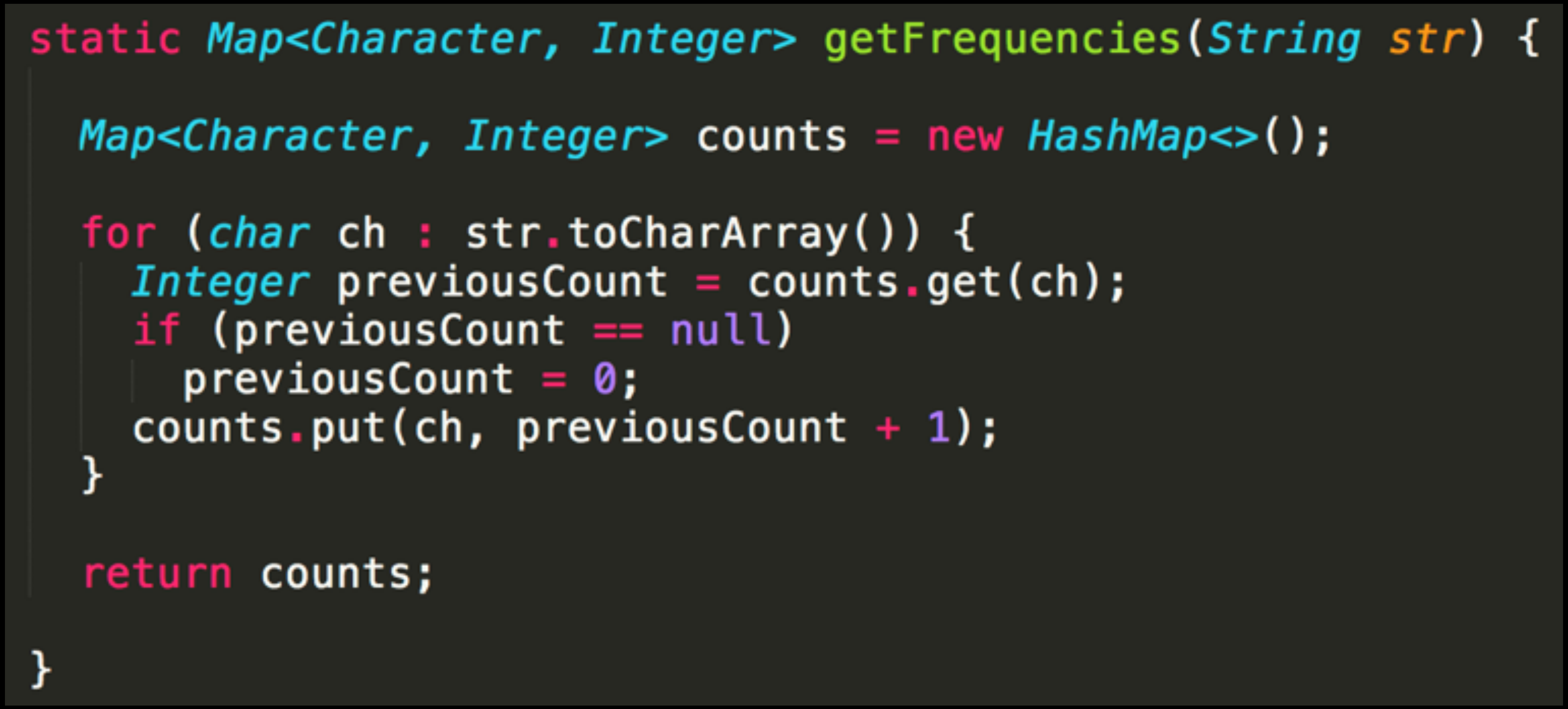
Maps

The *TreeMap* class has many other useful operations such as *firstKey()*, *lastKey()*, *lowerKey()*, *higherKey()*, *floorKey()*, and *ceilingKey()*. These operations are all O(log n).



Maps

For example, we could use maps to count the frequency of each letter in a string.



```java
static Map<Character, Integer> getFrequencies(String str) {

  Map<Character, Integer> counts = new HashMap<>();

  for (char ch : str.toCharArray()) {
    Integer previousCount = counts.get(ch);
    if (previousCount == null)
      previousCount = 0;
    counts.put(ch, previousCount + 1);
  }

  return counts;

}
```

getFrequencies("banana") gives {a=3, b=1, n=2}

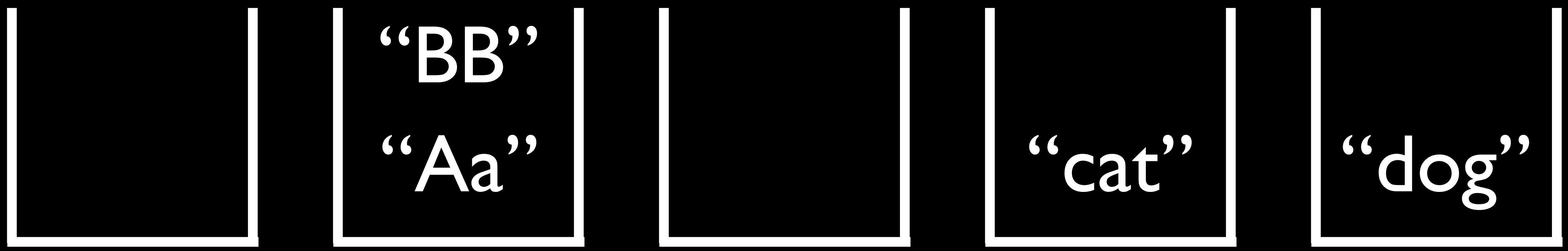
Maps

The *HashMap* class relies on the *hashCode()* method in order to efficiently store and retrieve its keys.

Equal objects have the same hashcode, but the fact that two objects have the same hashcode does not imply that they are equal.

Maps

At a simplified level, the *HashMap* class works by placing objects with the same hashcode into the same bucket.



“BB”

“Aa”

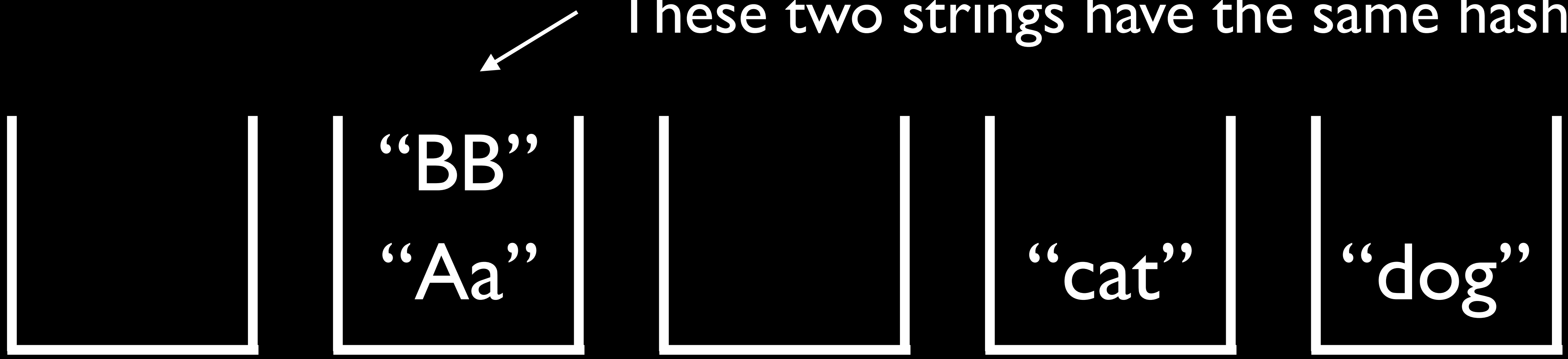
“cat”

“dog”

Maps

We have a “collision” if two keys hash to the same hashcode. If our hashing function is good, then we expect that there aren’t very many elements in each bucket, and that most buckets have something in them.

These two strings have the same hashcode



“BB”

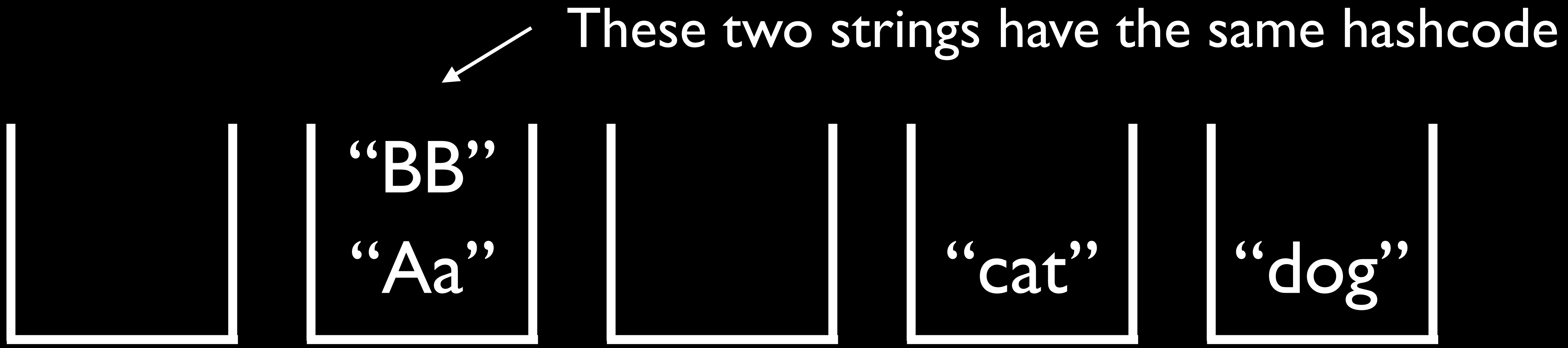
“Aa”

“cat”

“dog”

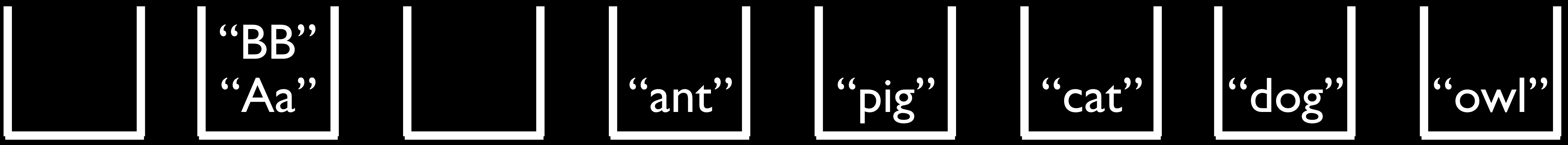
Maps

A collision does not affect our ability to correctly insert or remove from the map, however, it degrades performance.



Maps

As more keys get put into our *HashMap*, more buckets will be added to help distribute the load (but keys with the same hashcode will still be put in the same bucket).



“BB”
“Aa”

“ant”

“pig”

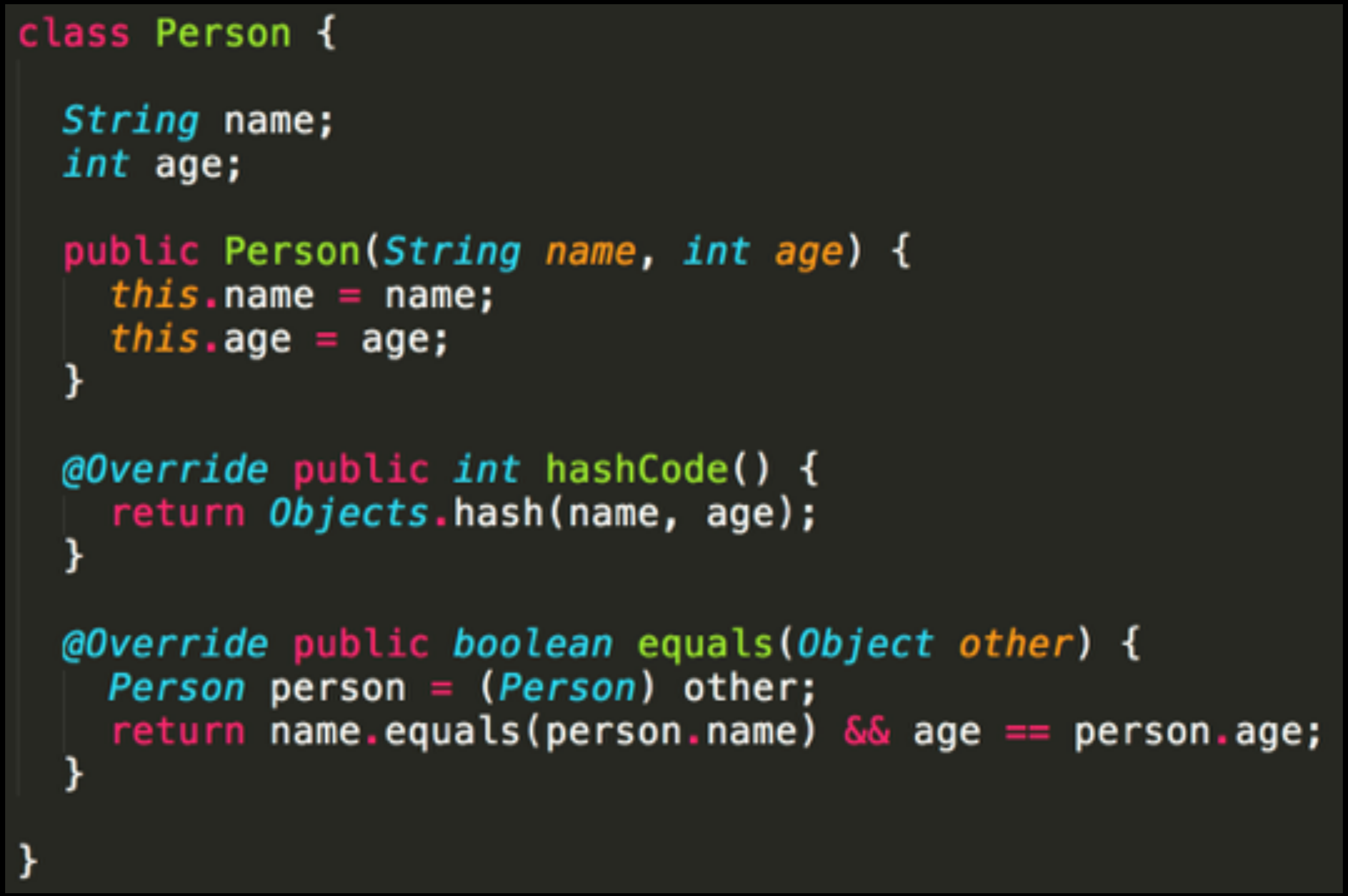
“cat”

“dog”

“owl”

Maps

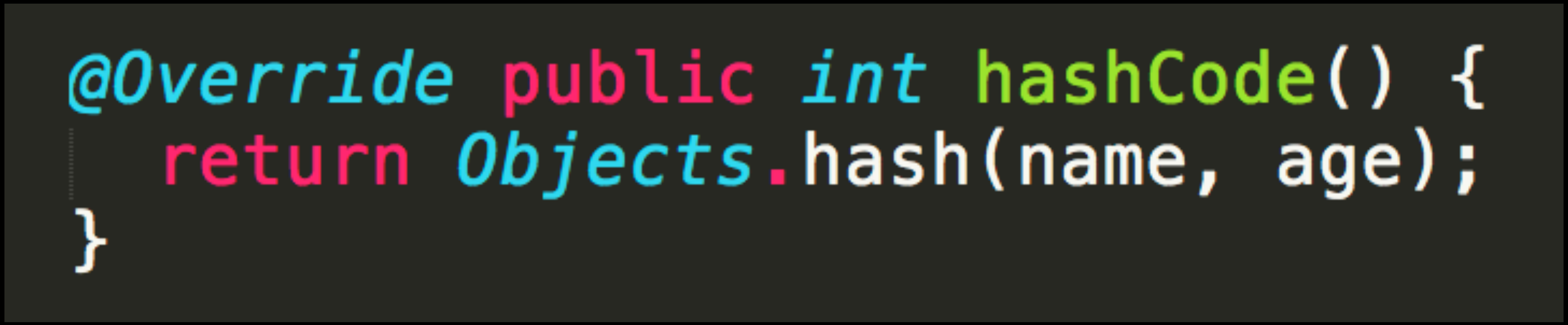
If we write our own class and want to use it as a key for a *HashMap*, then we need to override the *hashCode()* and *equals()* methods.



```java
class Person {

  String name;
  int age;

  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }

  @Override public int hashCode() {
    return Objects.hash(name, age);
  }

  @Override public boolean equals(Object other) {
    Person person = (Person) other;
    return name.equals(person.name) && age == person.age;
  }

}
```

Maps

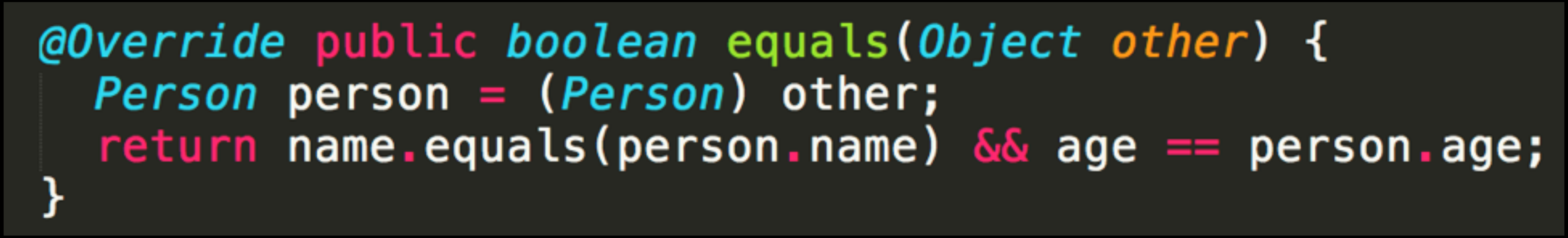
For the *hashCode()* method, I would definitely recommend just using *Objects.hash()*. This method takes any number of parameters and should do a good job at avoiding collisions.



```
@Override public int hashCode() {
  return Objects.hash(name, age);
}
```

Maps

The *equals()* method is also pretty straightforward. You can simply call the *equals()* methods of each property you want to check, or use == if comparing primitive types.



```
@Override public boolean equals(Object other) {
  Person person = (Person) other;
  return name.equals(person.name) && age == person.age;
}
```

Maps

Pros:

- Efficient data structure which allows you to pair two pieces of information together.

Cons:

- Each key can only have one associated value (although you could make that value be a list of elements).

Solution Sketch

Sort of Sorting

https://open.kattis.com/problems/sortofsorting

Summary: Given a list of names (2 to 20 characters each), sort them only by their first two letters, and output them. If two names start with the same two letters then their order should remain unchanged.

Sort of Sorting

Solution:

- Make a custom *Comparator* for the *String* class. In the *compareTo()* method, sort by the first character if they are different, otherwise sort by the second character.
- Store the list of names and use a *sort()* method from the Java API with your comparator (note that these sorts are stable).

Sources

- https://docs.oracle.com
- https://open.kattis.com/problems/sortofsorting