


# Comparators and Iterators

---

## Exam-Level 04



# Announcements

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
	2/12 Project 1B Due Weekly Survey Due			2/15 Midterm 1 (7-9pm)		
		2/20 Lab 4 Due Project 1C Due				



# Content Review

---



# Comparables

**Comparables** are things that **can be compared with each other**.

Any class could implement this interface.

Defines the notion of being “less than” or “greater than”.

```
public class Dog implements Comparable<Dog> {  
    private String name;  
    private int size;  
    @Override  
    public int compareTo(Dog otherDog) {  
        return this.size - otherDog.size;  
    }  
}
```



# Comparables

Can't use `<` and `>` directly on dog objects - undefined for them!

Instead, use the `compareTo` method instead.

```
if (d1 < d2) {  
} else  
}
```

```
if (d1.compareTo(d2) < 0) {  
    // Dog 1 "less than" dog  
} else {  
  
}
```



# Comparators

**Comparators** are things that **can be used to compare two objects**. Think of it as a “seesaw”. Comparables are the things sitting on the seesaw. Not the seesaw itself!

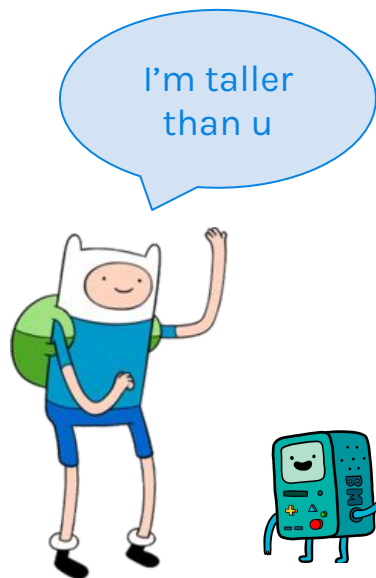
```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

```
public class DogComparator<Dog> implements Comparator<Dog> {  
    public int compare(Dog d1, Dog d2) {  
        return d1.size - d2.size;  
    }  
}
```



Credit to Austin for this slide

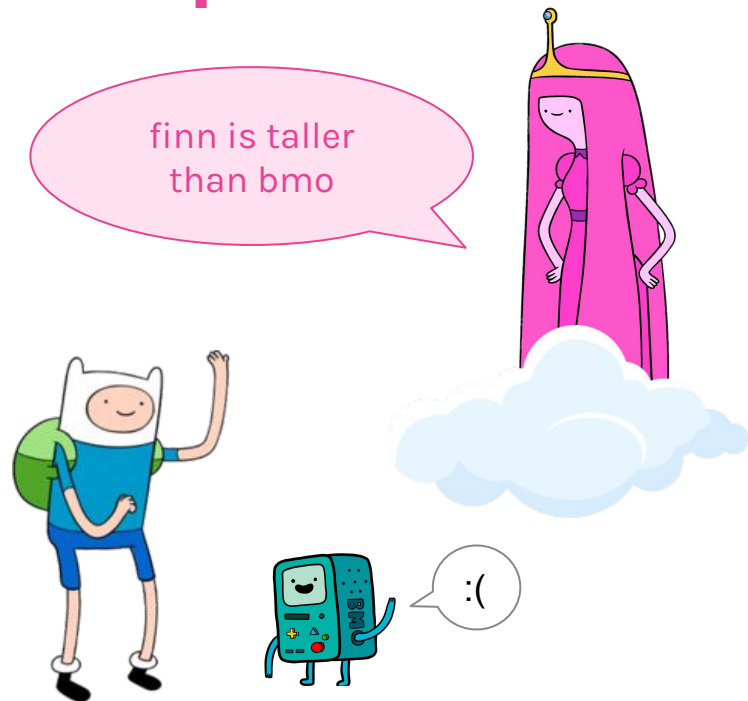
# Comparables



`finn.compareTo(bmo)`

VS

# Comparators



`bubblegum.compare(finn, bmo)`



# Why does compare/compareTo return an integer?

The `Comparator` interface's `compare` function takes in two objects of the same type and outputs:

- A negative integer if `o1` is “less than” `o2`
- A positive integer if `o1` is “greater than” `o2`
- Zero if `o1` is “equal to” `o2`

For `Comparable`, it is the same, except `o1` is `this`, and `o2` is the `other` object passed in.

Think of it as subtracting!

<code>compare(T o1, T o2) -&gt; o1 - o2</code>	<code>o1.compareTo(o2) -&gt; o1 - o2</code>
<code>o1 - o2 &lt; 0 -&gt; o1 &lt; o2</code>	<code>o1 - o2 &lt; 0 -&gt; o1 &lt; o2</code>
<code>o1 - o2 &gt; 0 -&gt; o1 &gt; o2</code>	<code>o1 - o2 &gt; 0 -&gt; o1 &gt; o2</code>
<code>o1 - o2 = 0 -&gt; o1 = o2</code>	<code>o1 - o2 = 0 -&gt; o1 = o2</code>





# The Iterator & Iterable Interfaces

**Iterators** are objects that can be iterated through in Java (in some sort of loop).

```
public interface Iterator<T> {  
    boolean hasNext();  
    T next();  
}
```

**Iterables** are objects that can produce an iterator.

```
public interface Iterable<T> {  
    Iterator<T> iterator();  
}
```



# The Iterator & Iterable Interfaces

The enhanced for loop

```
for (String x : lstOfStrings) // Lists, Sets, Arrays are all Iterable!
```

is shorthand for:

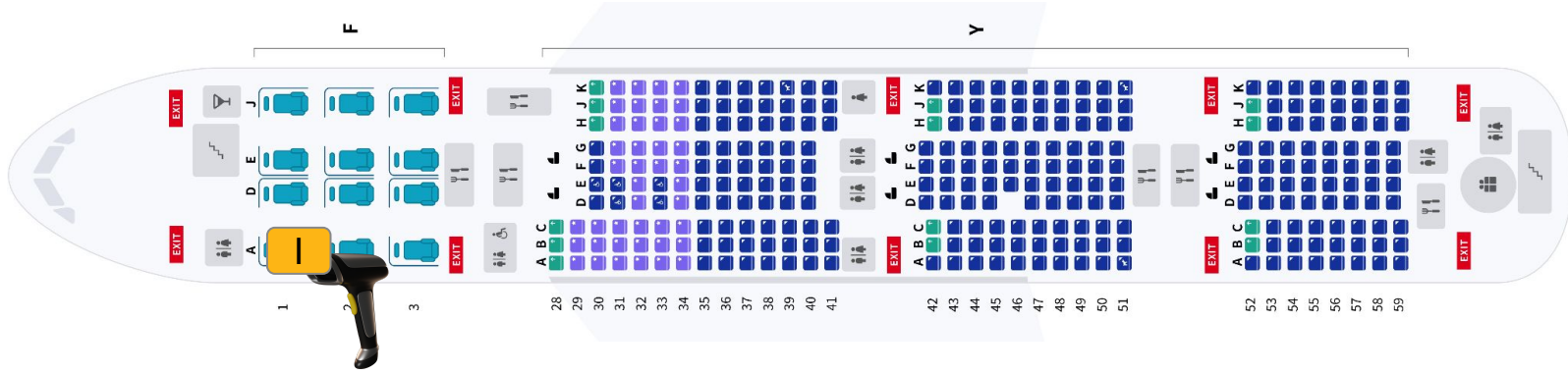
```
for (Iterator<String> iter = lstOfStrings.iterator(); iter.hasNext();) {  
    String x = iter.next();  
}
```



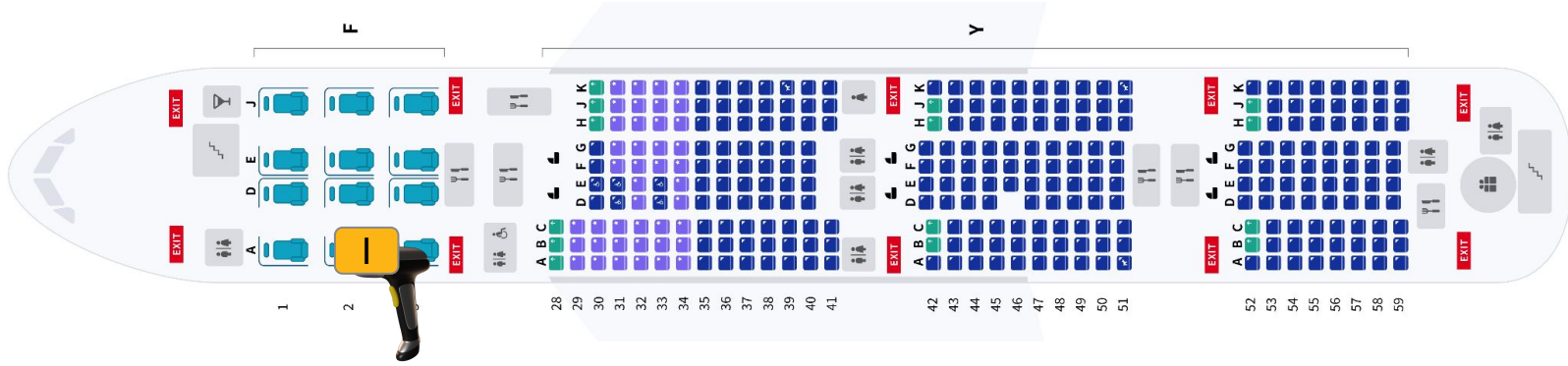
Credit to Ergun for this slide



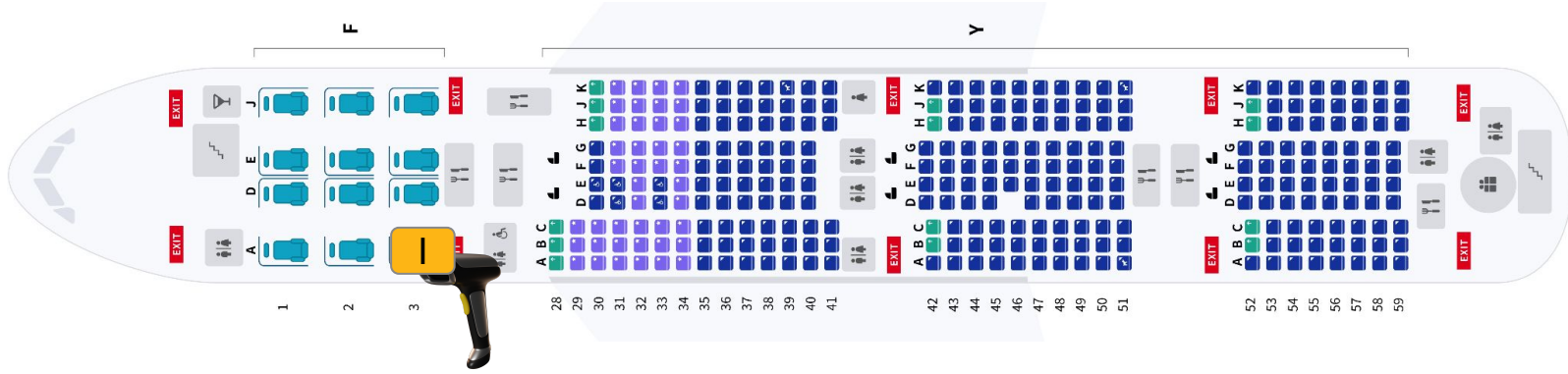
Credit to Ergun for this slide



Credit to Ergun for this slide



Credit to Ergun for this slide



# Check for Understanding

1. If we were to define a class that implements the interface `Iterable<Dog>`, what method(s) would this class need to define?
2. If we were to define a class that implements the interface `Iterator<Integer>`, what method(s) would this class need to define?
3. What's one difference between `Iterator` and `Iterable`?



# Check for Understanding

1. If we were to define a class that implements the interface `Iterable<Dog>`, what method(s) would this class need to define?

```
public Iterator<Dog> iterator()
```

2. If we were to define a class that implements the interface `Iterator<Integer>`, what method(s) would this class need to define?

```
public boolean hasNext()  
public Integer next()
```

3. What's one difference between `Iterator` and `Iterable`?

`Iterators` are the actual object we can iterate over, i.e., think a Python generator over a list.

`Iterables` are object that can produce an iterator, i.e., an array is iterable; an iterator over the array could go through the element at every index of the array).





# == vs. .equals()

- == compares if two variables point to the same object in memory.
  - null is compared with ==
- For reference types: `.equals()` (ex. `myDog.equals(yourDog)`)
  - Each class can provide own implementation by overriding
  - Defaults to `Object`'s `.equals()` (which is the same as ==)
  - Example: We make the `Dog` `.equals()` method return true if both Dogs have the same name
    - `Dog fido = new Dog("Fido"); Dog otherFido = new Dog("Fido");`
    - `fido == otherFido -> false, but fido.equals(otherFido) -> true`



# Exam Tips

Have a good night of sleep before the exam!

Take a few practice midterms and review lecture slides/discussions as needed. There's also a test question bank on the website in the resources section.

Remember our approach to understanding questions in order to solve them.

Flip through the exam as soon as you see it to get a sense of time allocation.



# Worksheet

---



## 1 Take Us to Your "Yrnqr"

You're a traveler who just landed on another planet. Luckily, the aliens there use the same alphabet as the English language, but in a different order.

Given the `AlienAlphabet` class below, fill in `AlienComparator` class so that it compares strings lexicographically, based on the order passed into the `AlienAlphabet` constructor. For simplicity, you may assume all words passed into `AlienComparator` have letters present in order.

For example, if the alien alphabet has the order "dba...", which means that d is the first letter, b is the second letter, etc., then `AlienComparator.compare("dab", "bad")` should return a negative value, since dab comes before bad.

If one word is an exact prefix of another, the longer word comes later. For example, "bad" comes before "badly". *Hint: indexOf might be helpful.*

Order of Completion:  
Top-down

```
1 public class AlienAlphabet {
2     private String order; ← tracks correct ordering of letters
3     public AlienAlphabet(String alphabetOrder) {
4         order = alphabetOrder;
5     }
6     public class AlienComparator implements Comparator<String> { ← generic
7         public int compare(String word1, String word2) {
8
9             int minLength = Math.min(word1.length(), word2.length());
10
11             for (int index = 0; index < minLength; index++) {
12
13                 int char1Rank = order.indexOf(word1.charAt(index));
14                 int char2Rank = order.indexOf(word2.charAt(index));
15                 if (char1Rank > char2Rank) {
16                     return -1;
17                 } else if (char1Rank < char2Rank) {
18                     return 1;
19                 }
20             }
21
22             return word1.length() - word2.length();
23         }
24     }
25 }
26
27
28 }
```

Smaller rank is better

## 2 Iterator of Iterators

Implement an `IteratorOfIterators` which takes in a `List` of `Iterators` of `Integers` as an argument. The first call to `next()` should return the first item from the first iterator in the list. The second call should return the first item from the second iterator in the list. If the list contained  $n$  iterators, the  $n+1$ th time that we call `next()`, we would return the second item of the first iterator in the list.

Note that if an iterator is empty in this process, we continue to the next iterator. Then, once all the iterators are empty, `hasNext` should return **false**. For example, if we had 3 `Iterators` A, B, and C such that A contained the values [1, 3, 4, 5], B was empty, and C contained the values [2], calls to `next()` for our `IteratorOfIterators` would return [1, 2, 3, 4, 5].

```
import java.util.*;
public class IteratorOfIterators implements Iterator<Integer> {
    private List<Iterator<Integer>> iterators;
    public IteratorOfIterators(List<Iterator<Integer>> a) {
        iterators = new ArrayList<>(a);
        for (Iterator<Integer> iterator : a) {
            if (iterator.hasNext()) {
                iterators.add(iterator);
            }
        }
    }

    @Override
    public boolean hasNext() {
        return iterators.length() > 0;
    }

    @Override
    public Integer next() {
        if (!hasNext()) {
            throw new UnsupportedOperationException();
        }
        Iterator<Integer> first = iterators.remove(0);
        Integer val = first.next();
        if (first.hasNext()) {
            iterators.add(first);
        }
        return val;
    }
}
```

Can't just make a new iterator - takes too long to load it fully; could also be nearly infinite

↳ soln uses LinkedList for access to remove First, addLast (Deque fn)

↳ makes sure iterators start with elements or more checks in next() needed

↳ equiv. to remove First