COLLECTIONS

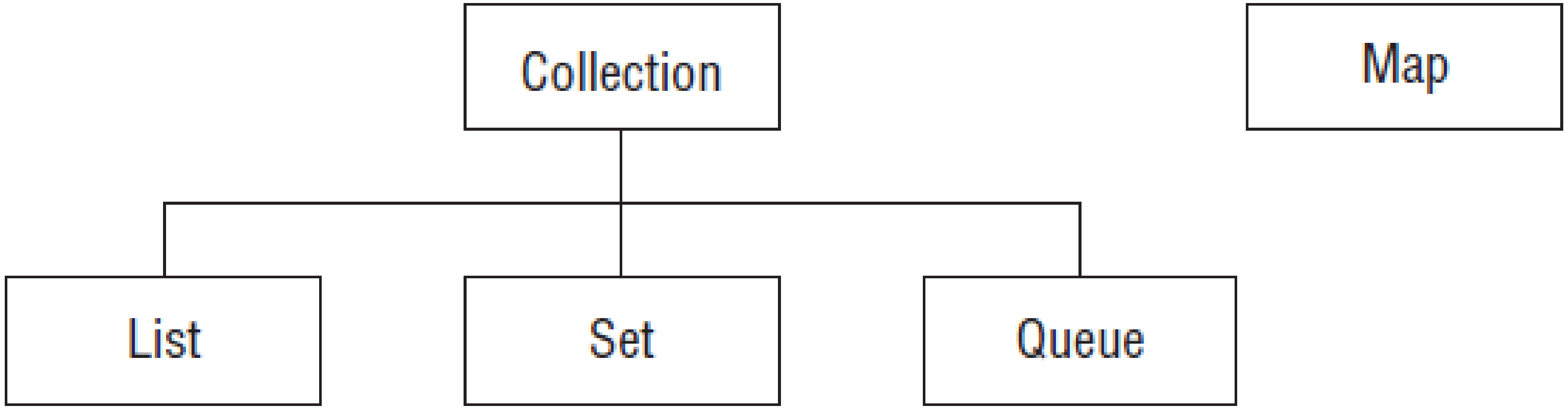
OUTLINE

1) JAVA COLLECTIONS FRAMEWORK

- **A collection is a group of objects contained in a single object.**
- The Java Collections Framework is a set of classes in java.util for storing collections.
- There are four main interfaces in the Java Collections Framework:
  - **List**: A list is an ordered collection of elements that allows duplicate entries. Elements in a list can be accessed by an int index.
  - **Set**: A set is a collection that does not allow duplicate entries.
  - **Queue**: A queue is a collection that orders its elements in a specific order for processing. A typical queue processes its elements in a first-in, first-out order, but other orderings are possible.
  - **Map**: A map is a collection that maps keys to values, with no duplicate keys allowed. The elements in a map are key/value pairs.

1) JAVA COLLECTIONS FRAMEWORK

- The figure shows the `Collection` interface and its core subinterfaces.
- Notice that `Map` doesn't implement the `Collection` interface. It is considered part of the Java Collections Framework, even though it isn't technically a `Collection`.
- It is a collection (note the lowercase), though, in that it contains a group of objects.
- The reason why maps are treated differently is that they need different methods due to being key/value pairs.

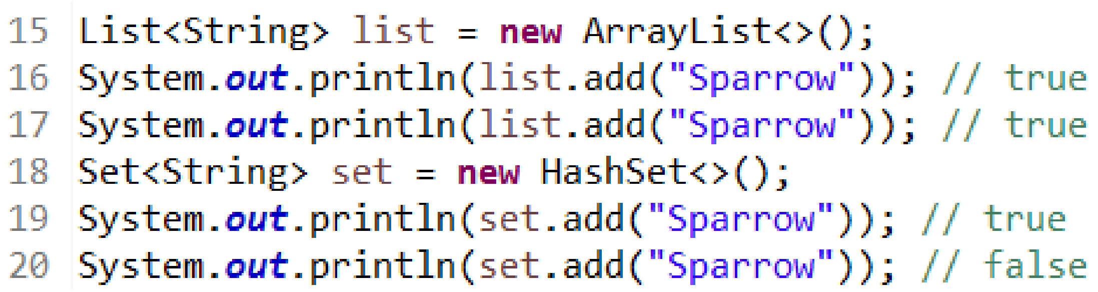


2) COMMON METHODS

ADD

- The `add()` method inserts a new element into the `Collection` and returns whether it was successful.
- The method signature is:
  - `boolean add(E element)`
- Remember that the Collections Framework uses generics. We will see E appear frequently. It means the generic type that was used to create the collection.
- For some collection types, add() always returns true. For other types, there is logic as to whether the add was successful.

- The following shows how to use this method.



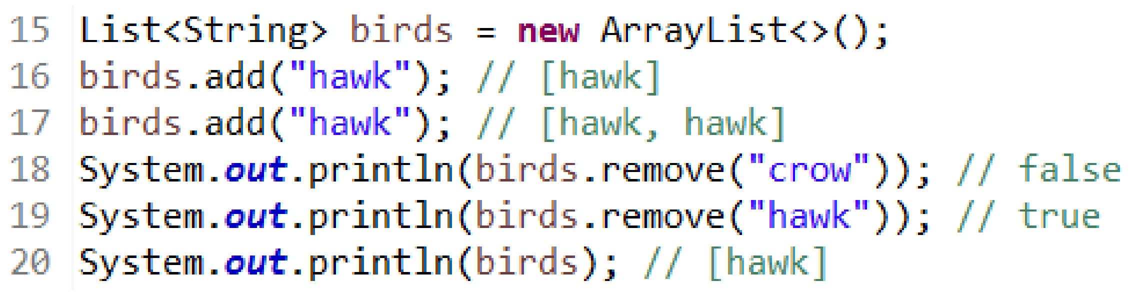
```
15 List<String> list = new ArrayList<>();
16 System.out.println(list.add("Sparrow")); // true
17 System.out.println(list.add("Sparrow")); // true
18 Set<String> set = new HashSet<>();
19 System.out.println(set.add("Sparrow")); // true
20 System.out.println(set.add("Sparrow")); // false
```

- A `List` allows duplicates, making the return value **`true`** each time.
- A `Set` does not allow duplicates. On line 20, we tried to add a duplicate so that Java returns **`false`** from the `add()` method.

2) COMMON METHODS

<u>REMOVE</u>

- The `remove()` method removes a single matching value in the `Collection` and returns whether it was successful.
- The method signature is:
  - `boolean remove(Object object)`
- This time, the **`boolean`** return value tells us whether a match was removed.
- The following shows how to use this method:



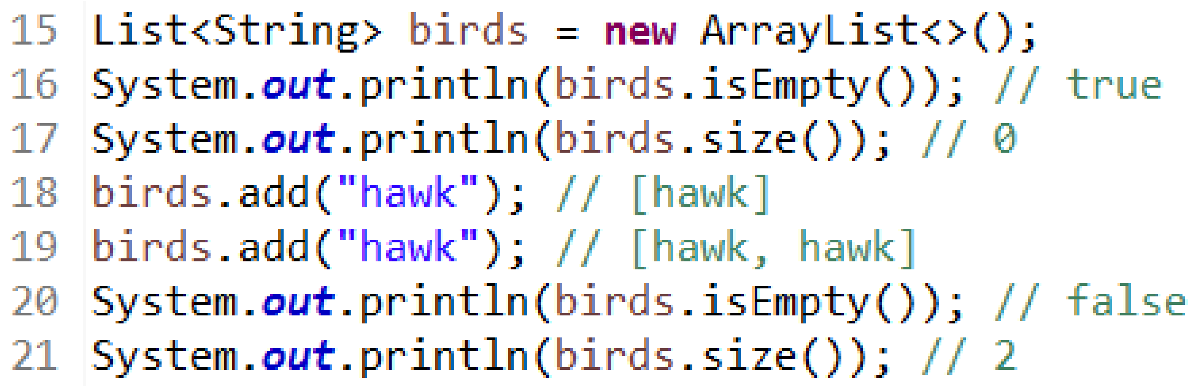
```
15 List<String> birds = new ArrayList<>();
16 birds.add("hawk"); // [hawk]
17 birds.add("hawk"); // [hawk, hawk]
18 System.out.println(birds.remove("crow")); // false
19 System.out.println(birds.remove("hawk")); // true
20 System.out.println(birds); // [hawk]
```

- Line 18 tries to remove an element that is not in `birds`.
- It returns **`false`** because no such element is found.
- Line 7 tries to remove an element that is in `birds`, so it returns **`true`**.
- **<u>Notice that it removes only one match.</u>**
- Since calling `remove()` with an **`int`** uses the index, an index that doesn't exist will throw an exception.
- For example, `birds.remove(100);` throws an `IndexOutOfBoundsException`.
- Remember that there are overloaded `remove()` methods.
- One takes the element to remove. The other takes the index of the element to remove.
- The latter is being called here.

2) COMMON METHODS

ISEMPTY() AND SIZE()

- The `isEmpty()` and `size()` methods look at how many elements are in the `Collection`.
- The method signatures are:
  - `boolean isEmpty()`
  - `int size()`
- The following shows how to use these methods:



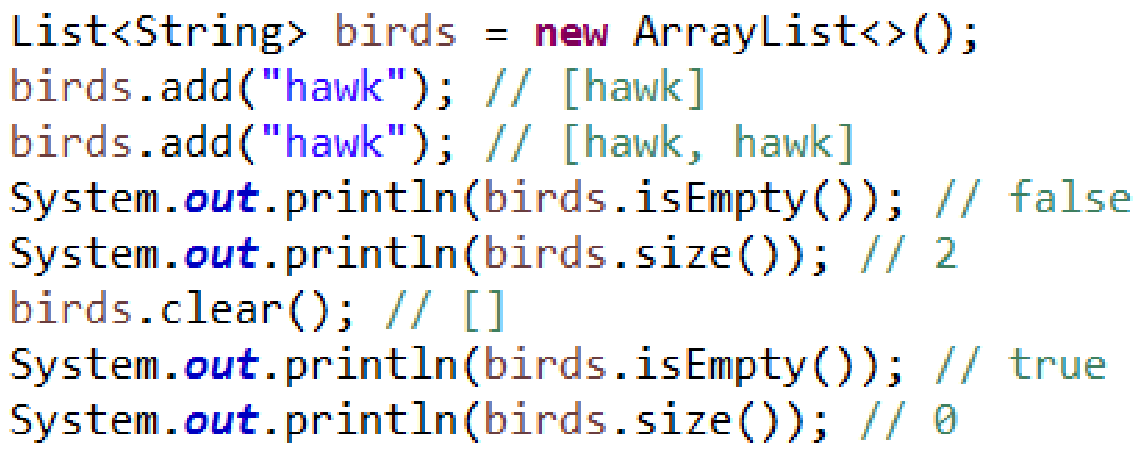
```
List<String> birds = new ArrayList<>();
System.out.println(birds.isEmpty()); // true
System.out.println(birds.size()); // 0
birds.add("hawk"); // [hawk]
birds.add("hawk"); // [hawk, hawk]
System.out.println(birds.isEmpty()); // false
System.out.println(birds.size()); // 2
```

- At the beginning, `birds` has a size of 0 and is empty.
- It has a capacity that is greater than 0.
- After we add elements, the size becomes positive and it is no longer empty.

2) COMMON METHODS

CLEAR()

- The `clear()` method provides an easy way to discard all elements of the Collection.
- The method signature is:
  - `void clear()`
- The following shows how to use this method:



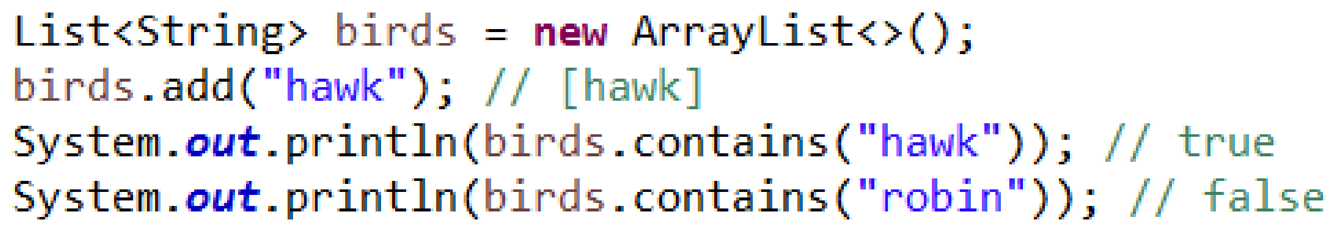
```
List<String> birds = new ArrayList<>();
birds.add("hawk"); // [hawk]
birds.add("hawk"); // [hawk, hawk]
System.out.println(birds.isEmpty()); // false
System.out.println(birds.size()); // 2
birds.clear(); // []
System.out.println(birds.isEmpty()); // true
System.out.println(birds.size()); // 0
```

- After calling clear(), `birds` is back to being an empty `ArrayList` of size 0.

2) COMMON METHODS

CONTAINS()

- The contains() method checks if a certain value is in the Collection.
- The method signature is:
  - `boolean contains(Object object)`
- The following shows how to use this method:



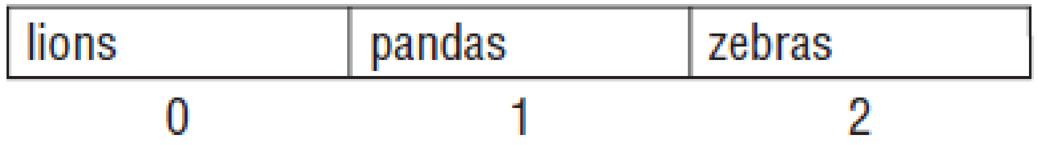
```
List<String> birds = new ArrayList<>();
birds.add("hawk"); // [hawk]
System.out.println(birds.contains("hawk")); // true
System.out.println(birds.contains("robin")); // false
```

- This method calls **equals()** on each element of the `ArrayList` to see if there are any matches.

3) LIST INTERFACE

- **We use a list when we want an ordered collection that can contain duplicate entries.**
- **Items can be retrieved and inserted at specific positions in the list based on an int index much like an array.**
- Lists are commonly used because there are many situations in programming where we need to keep track of a list of objects.
- For example, we might make a list of what we want to see at the zoo:
  - First, we see the lions because they go to sleep early.
  - Second, see the pandas because there is a long line later in the day. And so forth.

- The figure shows how we can envision a `List` .
- **Each element of the `List` has an index, and the indexes begin with zero**.



- Sometimes, we don’t actually care about the order of elements in a list.
- List is like the “go to” data type.
- When we make a shopping list before going to the store, the order of the list happens to be the order in which we thought of the items.
- We probably aren’t attached to that particular order, it isn’t hurting anything.

3) LIST INTERFACE
3.1) LIST IMPLEMENTATIONS

- The main thing that all `List` implementations have in common is that they are ordered and allow duplicates.
- An **`ArrayList`** is like a resizable array. When elements are added, the **`ArrayList`** automatically grows.
- **When you aren’t sure which collection to use, use an `ArrayList`.**
- **The main benefit of an `ArrayList` is that you can look up any element in constant time.**
- Adding or removing an element is slower than accessing an element.
- This makes an `ArrayList` a good choice when you are reading more often than (or the same amount as) writing to the `ArrayList` .

- In computer programming, we use **big O notation** to talk about the performance of algorithms.
- This is called an ***order of magnitude*** difference.
- Big O notation lets you compare the order of magnitude of performance rather than the exact performance.
- It also assumes the worst-case response time. If you write an algorithm that could take a while or be instantaneous, big O uses the longer one.
- It uses an **`n`** to reflect the number of elements or size of the data you are talking about.

3) LIST INTERFACE
3.1) LIST IMPLEMENTATIONS

- The following lists the most common big O notation values that you will see and what they mean:
- **<u>O(1) - constant time</u>**:
  - It doesn’t matter how large the collection is, the answer will always take the same time to return.
  - Returning the last element of an array has O(1) because we know the last index.
- **<u>O(log n) - logarithmic time</u>**:
  - A logarithm is a mathematical function that grows much more slowly than the data size.
  - Binary search runs in logarithmic time because it doesn’t look at the majority of the elements for large collections.
- **<u>O(n) - linear time:</u>**
  - The performance will grow linearly with respect to the size of the collection.
  - Looping through a list and returning the number of elements matching “Panda” will take linear time.
- **<u>O(n2) - n squared time:</u>**
  - Code that has nested loops where each loop goes through the data takes n squared time.
  - An example would be putting every pair of pandas together to see if they’ll share an exhibit.

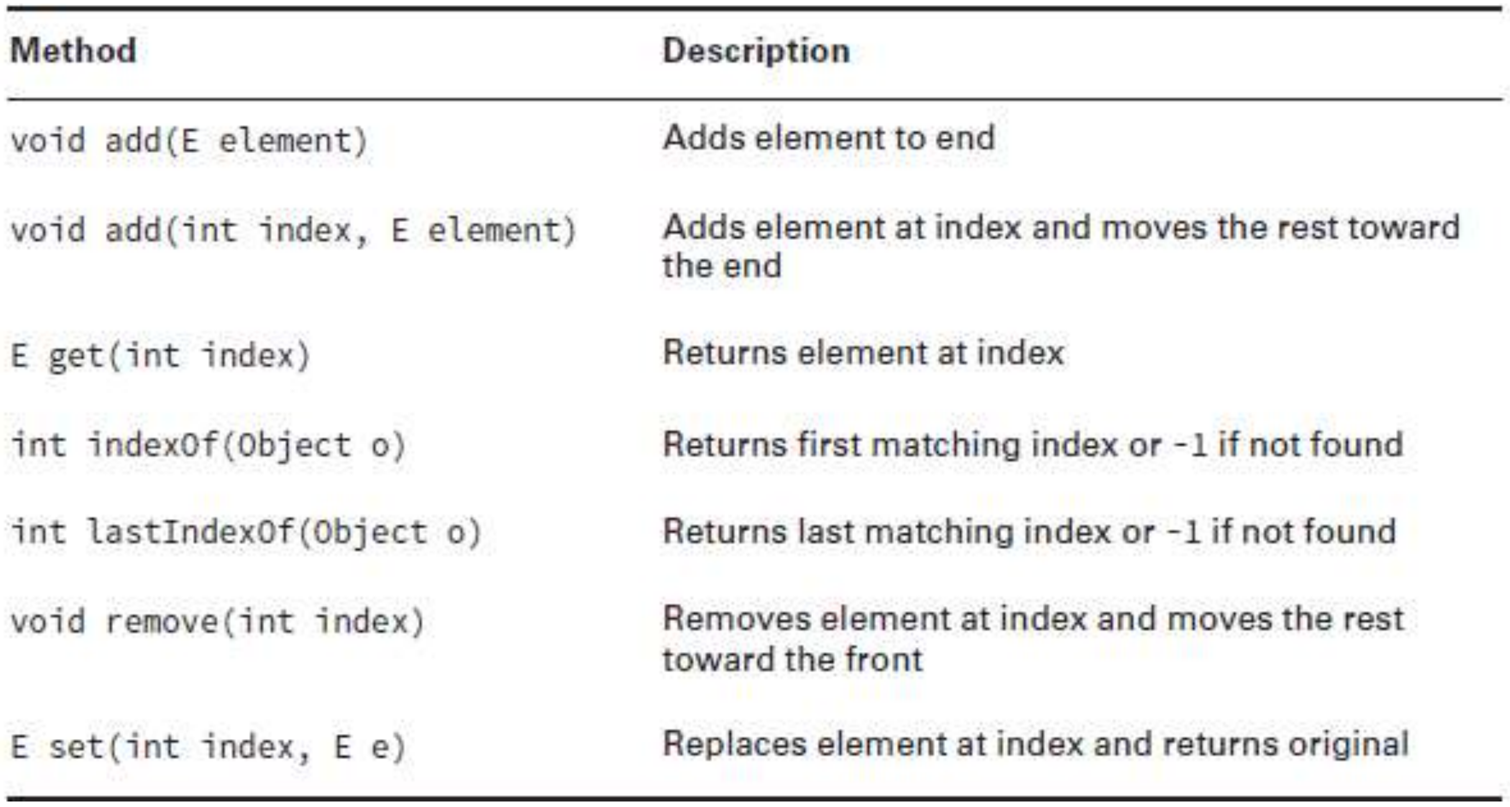
3) LIST INTERFACE
3.1) LIST IMPLEMENTATIONS

- A `LinkedList` is special because it implements both `List` and `Queue`.
- It has all of the methods of a `List`.
- It also has additional methods to facilitate adding or removing from the beginning and/or end of the list.
- The main benefits of a `LinkedList` are that you can access, add, and remove from the beginning and end of the list in constant time.
- **<u>The tradeoff is that dealing with an arbitrary index takes linear time.</u>**
- This makes a LinkedList a good choice when you'll be using it as Queue.

- A `Stack` is a data structure where you add and remove elements from the top of the stack.
- Think about a stack of paper as an example.
- Stack hasn't been used for new code in ages. If you need a stack, use an `ArrayDeque` instead.
- More on this when we get to the Queue section.

3) LIST INTERFACE
3.2) LIST METHODS

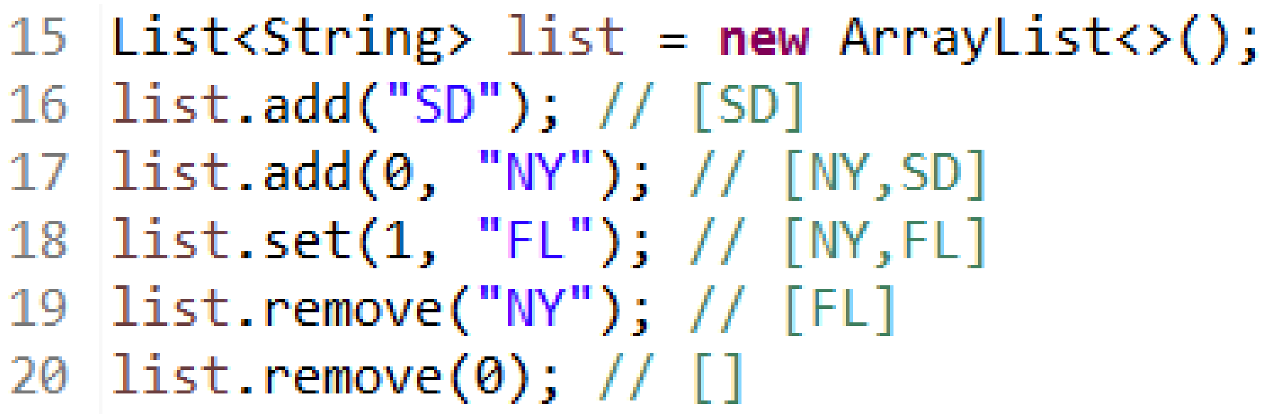
- The methods in the `List` interface are for working with indexes.
- In addition to the inherited Collection methods, the method signatures that you need to know are in the following table.



Method	Description
`void add(E element)`	Adds element to end
`void add(int index, E element)`	Adds element at index and moves the rest toward the end
`E get(int index)`	Returns element at index
`int indexOf(Object o)`	Returns first matching index or `-1` if not found
`int lastIndexOf(Object o)`	Returns last matching index or `-1` if not found
`void remove(int index)`	Removes element at index and moves the rest toward the front
`E set(int index, E e)`	Replaces element at index and returns original

3) LIST INTERFACE
3.2) LIST METHODS

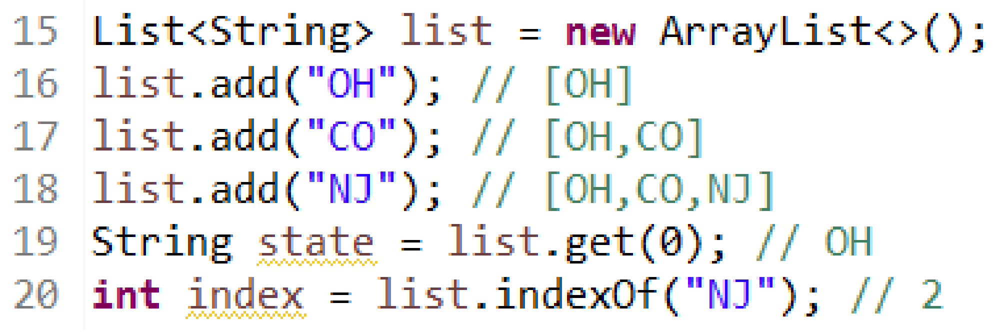
- The following statements demonstrate these basic methods for adding and removing items from a list:
- On line 15, `list` starts out empty.
- Line 16 adds an element to the end of the list.
- Line 17 adds an element at index 0 that bumps the original index 0 to index 1.
- Notice how the `ArrayList` is now automatically one larger.
- Line 18 replaces the element at index 1 with a new value.
- Line 19 removes the element matching “NY”.
- Finally, line 20 removes the element at index 0 and list is empty again



```java
15 List<String> list = new ArrayList<>();
16 list.add("SD"); // [SD]
17 list.add(0, "NY"); // [NY,SD]
18 list.set(1, "FL"); // [NY,FL]
19 list.remove("NY"); // [FL]
20 list.remove(0); // []
```

3) LIST INTERFACE
3.2) LIST METHODS

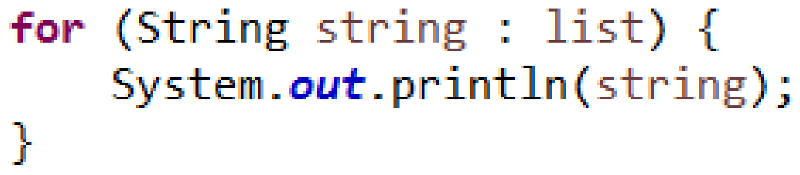
- Let’s look at one more example that queries the list:



```
15 List<String> list = new ArrayList<>();
16 list.add("OH"); // [OH]
17 list.add("CO"); // [OH,CO]
18 list.add("NJ"); // [OH,CO,NJ]
19 String state = list.get(0); // OH
20 int index = list.indexOf("NJ"); // 2
```

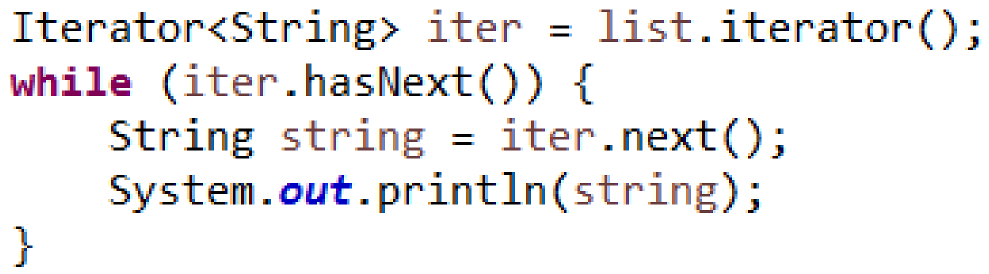
- Lines 16 through 18 add elements to list in order.
- Line 19 requests the element at index 0.
- Line 20 searches the list until it hits an element with “NJ”.
- The elements do not need to be in order for this to work because indexOf() looks through the whole list until it finds a match.

- Let’s look on how to iterate a collection with an enhanced for loop:



```
for (String string : list) {
    System.out.println(string);
}
```

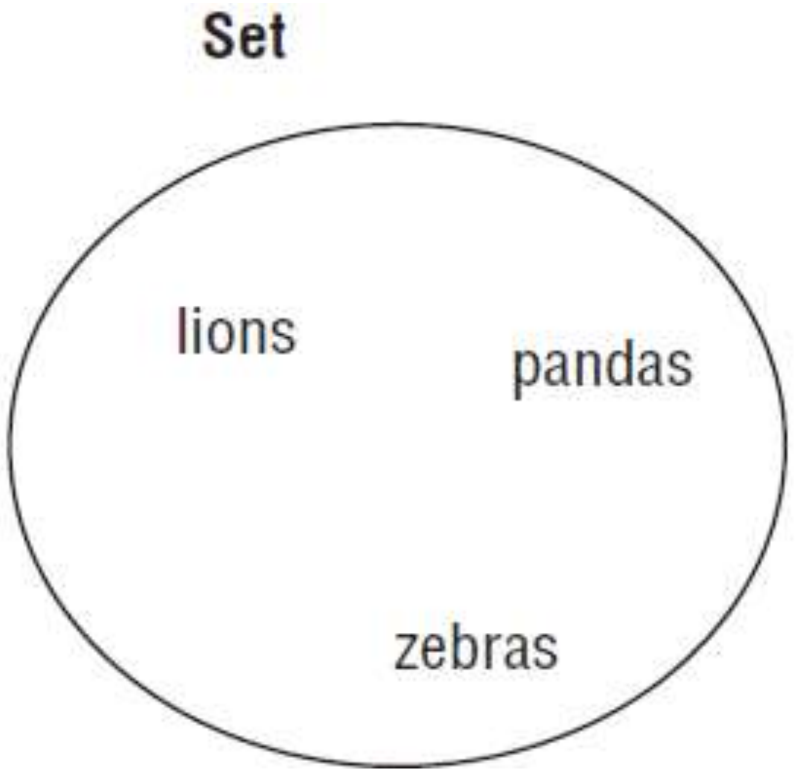
- There’s also an older way where we still use Iterator with generics.



```
Iterator<String> iter = list.iterator();
while (iter.hasNext()) {
    String string = iter.next();
    System.out.println(string);
}
```

4) SET INTERFACE

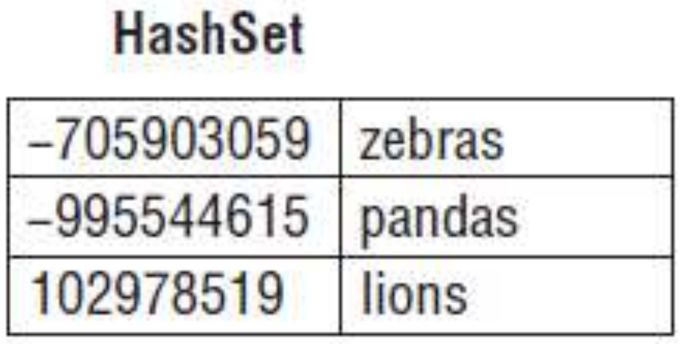
- **You use a set when you don’t want to allow duplicate entries.**
- For example, you might want to keep track of the unique animals that you want to see at the zoo.
- You aren’t concerned with the order in which you see these animals, but there isn’t time to see them more than once.
- You just want to make sure that you see the ones that are important to you and remove them from the set of outstanding animals to see after you see them.
- The figure shows how you can envision a Set.
- **The main thing that all Set implementations have in common is that they do not allow duplicates**.



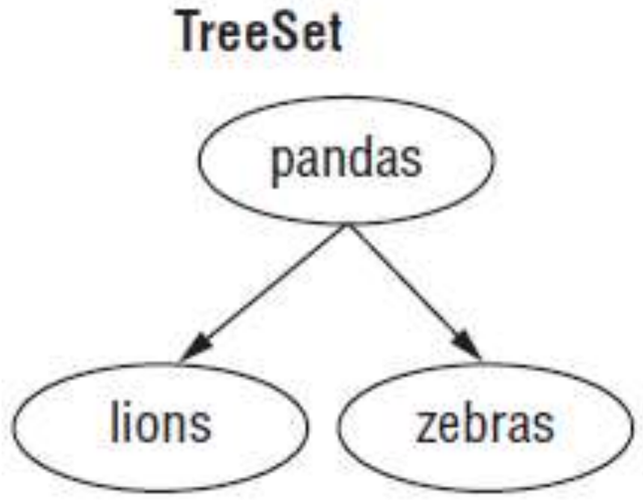
4) SET INTERFACE
4.1) SET IMPLEMENTATIONS

- A `HashSet` stores its elements in a hash table.
- **This means that it uses the hashCode() method of the objects to retrieve them more efficiently.**
- The main benefit is that adding elements and checking if an element is in the set both have constant time.
- **The tradeoff is that you lose the order in which you inserted the elements.**
- Most of the time, you aren't concerned with this in a set anyway, making HashSet the most common set.

- A `TreeSet` stores its elements in a sorted tree structure.
- **The main benefit is that the set is always in sorted order.**
- **The tradeoff is that adding and checking if an element is present are both O(log n).**
- TreeSet implements a special interface called NavigableSet

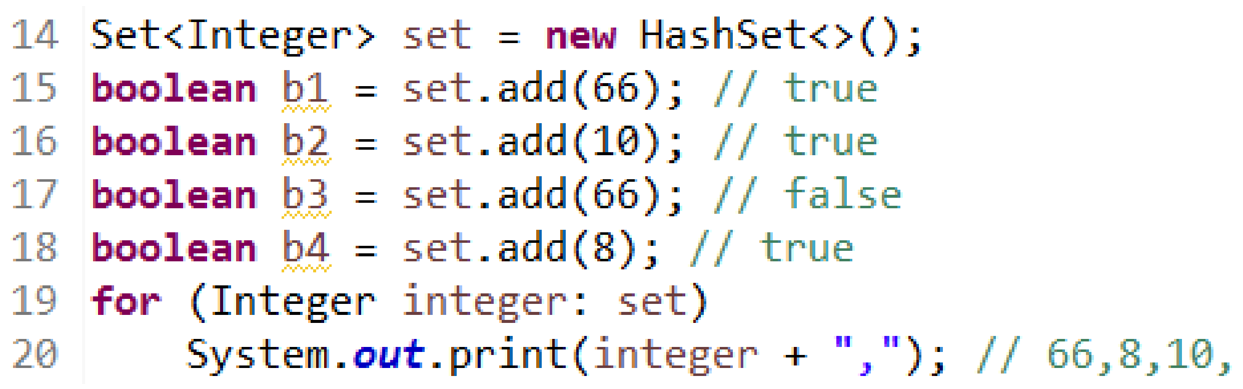

HashSet

–705903059	zebras
–995544615	pandas
102978519	lions



4) SET INTERFACE
4.2) SET METHODS

- The Set interface doesn’t add any extra methods that you need to know for the exam.
- You just have to know how sets behave with respect to the traditional Collection methods.
- You also have to know the differences between the types of sets.
- Let’s start with HashSet:

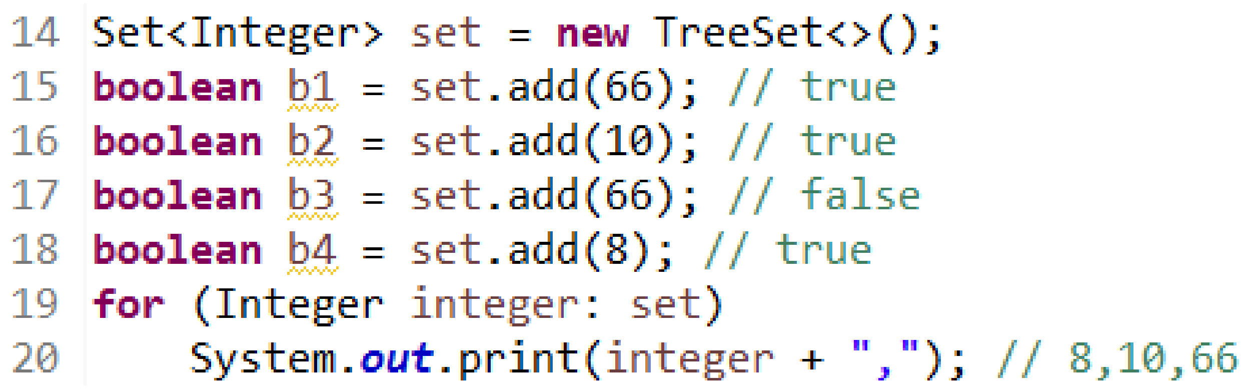


```
14 Set<Integer> set = new HashSet<>();
15 boolean b1 = set.add(66); // true
16 boolean b2 = set.add(10); // true
17 boolean b3 = set.add(66); // false
18 boolean b4 = set.add(8); // true
19 for (Integer integer: set)
20     System.out.print(integer + ","); // 66,8,10,
```

- The `add()` method returns **`true`** unless the `Integer` is already in the set.
- Line 17 returns **`false`**, because we already have 66 in the set and a set must preserve uniqueness.
- Line 19-20 prints the elements of the set in an arbitrary order. In this case, it happens not to be sorted order, or the order in which we added the elements.
- Remember that the **`equals()`** method is used to determine equality.
- The **`hashCode()`** method is used to know which bucket to look in so that Java doesn’t have to look through the whole set to find out if an object is there.
- The best case is that hash codes are unique, and Java has to call **`equals()`** on only one object.
- The worst case is that all implementations return the same **`hashCode()`**, and Java has to call **`equals()`** on every element of the set anyway.

4) SET INTERFACE
4.2) SET METHODS

- Now let’s look at the same example with `TreeSet`:

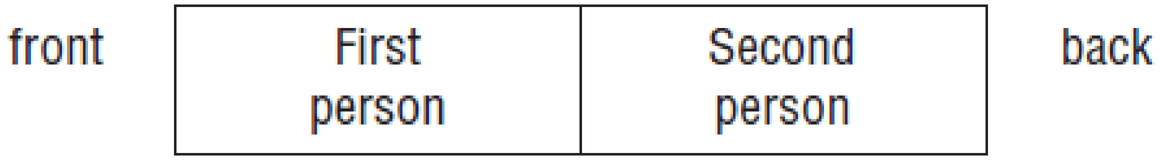


```
Set<Integer> set = new TreeSet<>();
boolean b1 = set.add(66); // true
boolean b2 = set.add(10); // true
boolean b3 = set.add(66); // false
boolean b4 = set.add(8); // true
for (Integer integer: set)
    System.out.print(integer + ","); // 8,10,66
```

- **This time, the elements are printed out in their natural sorted order.**
- Numbers implement the `Comparable` interface in Java, which is used for sorting.

5) QUEUE INTERFACE

- **You use a queue when elements are added and removed in a specific order.**
- Queues are typically used for sorting elements prior to processing them.
- For example, when you want to buy a ticket and someone is waiting in line, you get in line behind that person.
- A queue is assumed to be **FIFO (first-in, first-out)**.
- Some queue implementations change this to use a different order.
- You can envision a **FIFO** queue as shown in the figure.
- The other common format is **LIFO (last-in, first-out.)** used for stacks.



- All queues have specific requirements for adding and removing the next element.
- Beyond that, they each offer different functionality.
- We will look at the implementations that we need to know and the available methods.

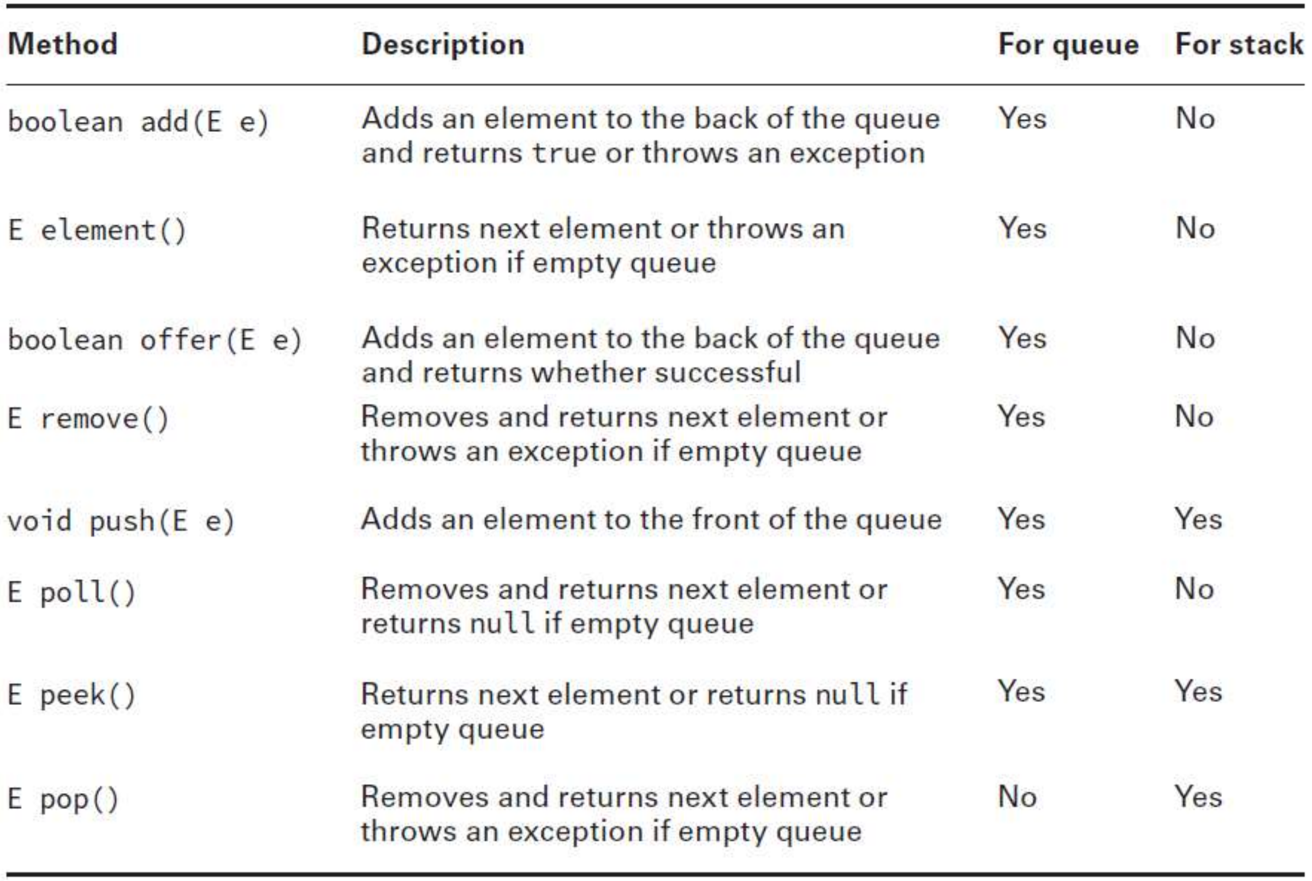
5) QUEUE INTERFACE
5.1) QUEUE IMPLEMENTATIONS

- We saw `LinkedList` earlier in the `List` section.
- In addition to being a list, it is a double-ended queue.
- A double-ended queue is different from a regular queue in that you can insert and remove elements from both the front and back of the queue.
- Think, “Mr. President, come right to the front. You are the only one who gets this special treatment. Everyone else will have to start at the back of the line.”
- **The main benefit of a `LinkedList` is that it implements both the `List` and `Queue` interfaces.**
- **The tradeoff is that it isn’t as efficient as a “pure” queue.**

- **An `ArrayDeque` is a “pure” double-ended queue. <u>It stores its elements in a resizable array</u>**.
- The main benefit of an `ArrayDeque` is that it is more efficient than a `LinkedList`.
- Deque is supposed to be pronounced “deck,” but many people, say it wrong as “d-queue.”

5) QUEUE INTERFACE
5.2) QUEUE METHODS

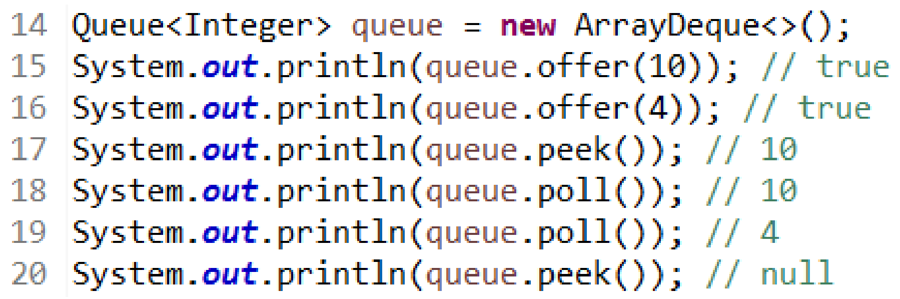
- The `ArrayDeque` contains many methods.
- The table lists the important ones.
- **`push`** is what makes it a double-ended queue.
- As you can see, there are basically two sets of methods.
- One set throws an exception when something goes wrong.
- The other uses a different return value when something goes wrong.
- The `offer/poll/peek` methods are more common.
- This is the standard language people use when working with queues.



Method	Description	For queue	For stack
`boolean add(E e)`	Adds an element to the back of the queue and returns `true` or throws an exception	Yes	No
`E element()`	Returns next element or throws an exception if empty queue	Yes	No
`boolean offer(E e)`	Adds an element to the back of the queue and returns whether successful	Yes	No
`E remove()`	Removes and returns next element or throws an exception if empty queue	Yes	No
`void push(E e)`	Adds an element to the front of the queue	Yes	Yes
`E poll()`	Removes and returns next element or returns `null` if empty queue	Yes	No
`E peek()`	Returns next element or returns `null` if empty queue	Yes	Yes
`E pop()`	Removes and returns next element or throws an exception if empty queue	No	Yes

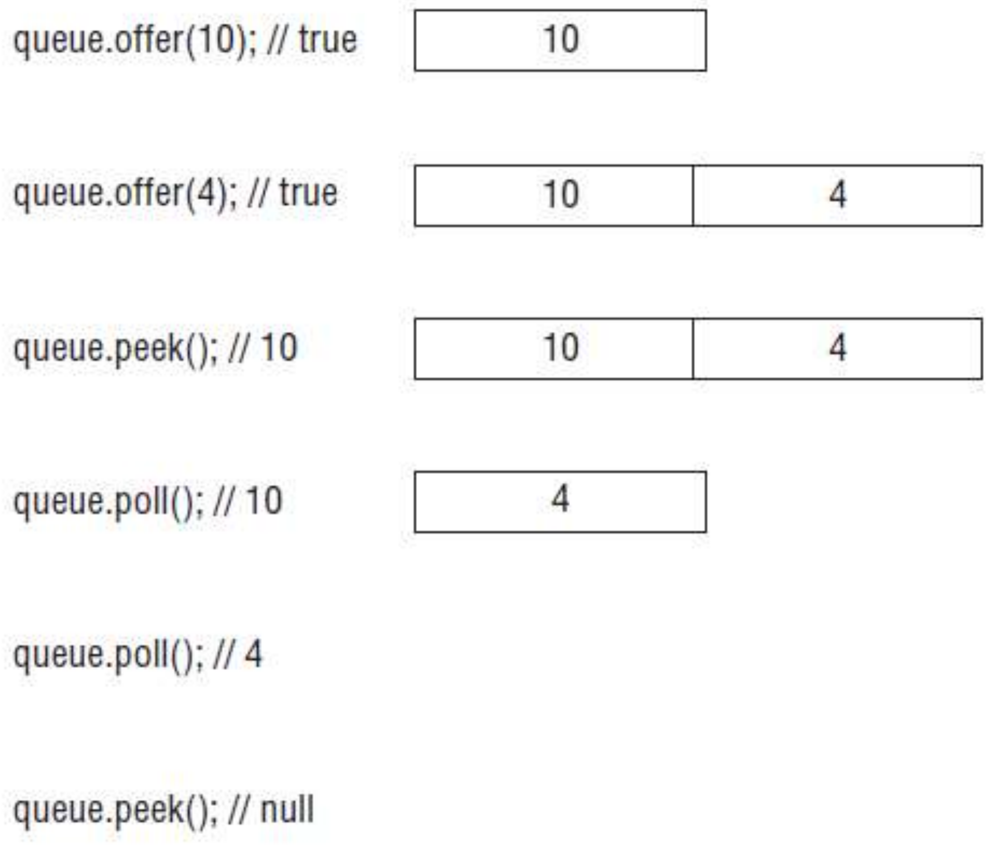
5) QUEUE INTERFACE
5.2) QUEUE METHODS

- Let’s look at an example that uses some of these methods:



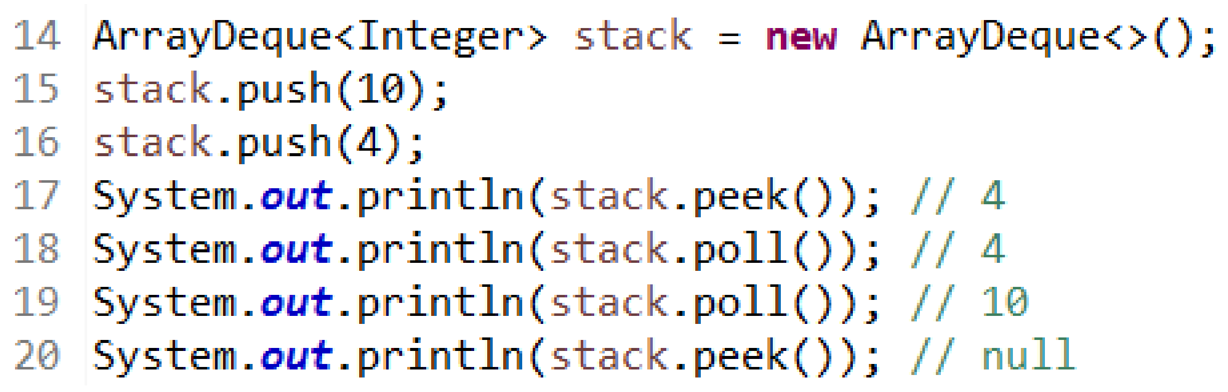
```
14 Queue<Integer> queue = new ArrayDeque<>();
15 System.out.println(queue.offer(10)); // true
16 System.out.println(queue.offer(4)); // true
17 System.out.println(queue.peek()); // 10
18 System.out.println(queue.poll()); // 10
19 System.out.println(queue.poll()); // 4
20 System.out.println(queue.peek()); // null
```

- The figure shows what the queue looks like at each step of the code.
- Lines 15 and 16 successfully add an element to the end of the queue.
- Line 17 looks at the first element in the queue, but it does not remove it.
- Lines 18 and 19 actually remove the elements from the queue, which results in an empty queue.
- Line 20 tries to look at the first element of a queue, which results in null.

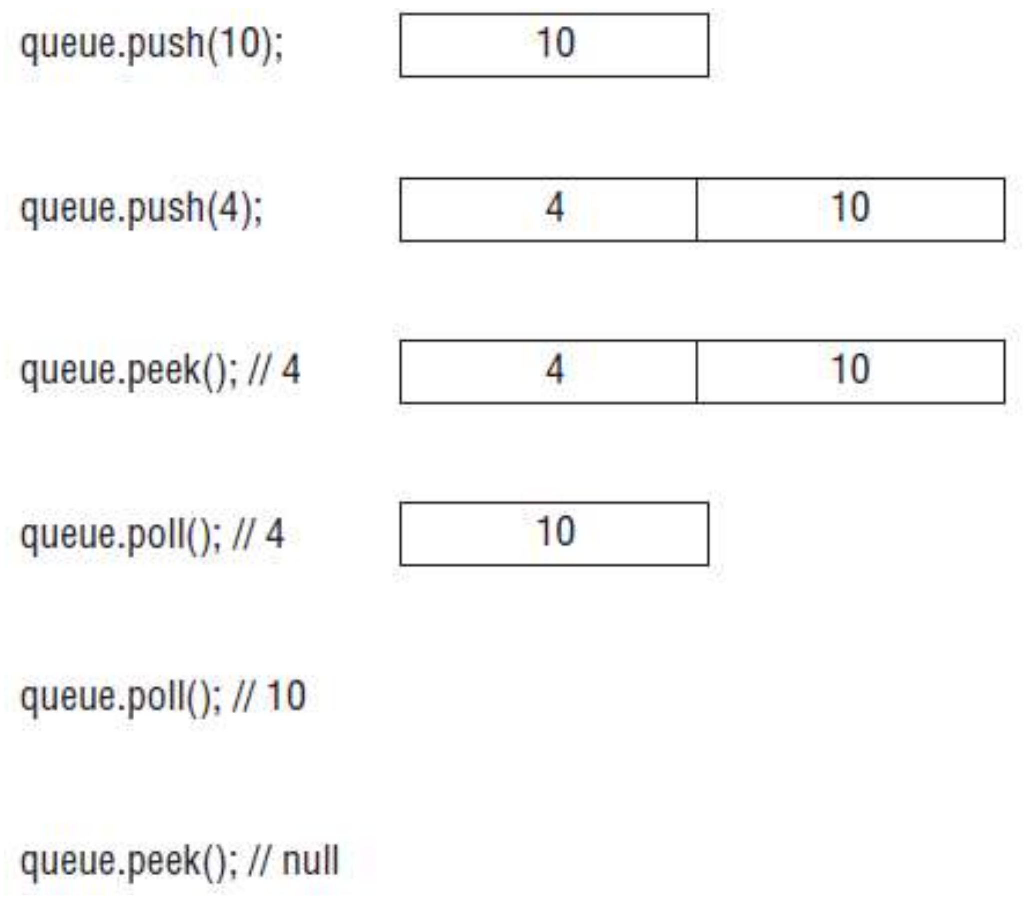


5) QUEUE INTERFACE
5.2) QUEUE METHODS

- What if we want to insert an element at the other end, just as we could with a Stack?
- No problem. We just call the `push()` method.
- It works just like offer() except at the other end of the queue.
- When talking about LIFO (stack), people say push/poll/peek.
- When talking about FIFO (single-ended queue), people say offer/poll/peek.
- Now let's rewrite that example using the stack functionality:

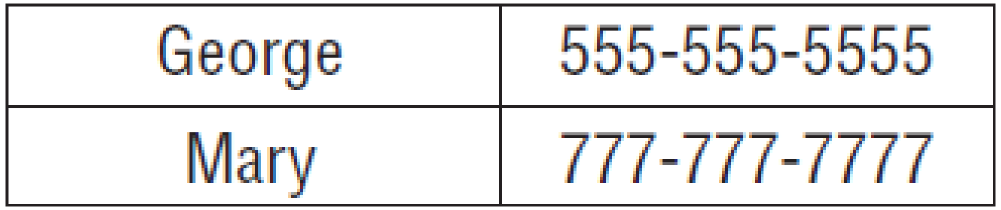


```
ArrayDeque<Integer> stack = new ArrayDeque<>();
stack.push(10);
stack.push(4);
System.out.println(stack.peek()); // 4
System.out.println(stack.poll()); // 4
System.out.println(stack.poll()); // 10
System.out.println(stack.peek()); // null
```



6) MAP INTERFACE

- **You use a map when you want to identify values by a key.**
- For example, when you use the contact list in your phone, you look up “George” rather than looking through each phone number in turn.
- You can envision a Map as shown in the figure.
- You don’t need to know the names of the specific interfaces that the different maps implement, but you do need to know that `TreeMap` is sorted and navigable.
- The main thing that all four classes have in common is that they all have keys and values.
- Beyond that, they each offer different functionality.
- We will look at the implementations that you need to know and the available methods.



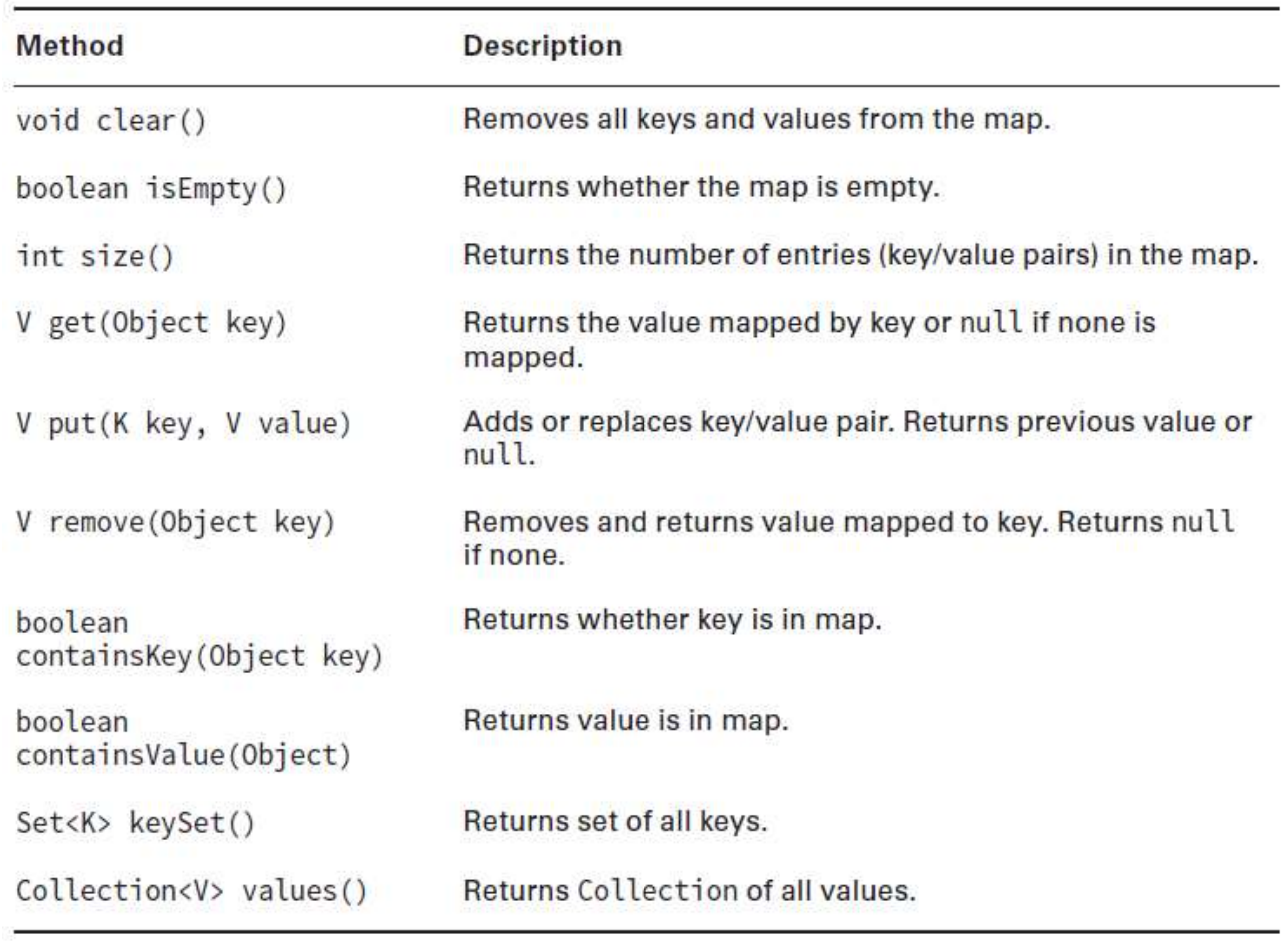
George	555-555-5555
Mary	777-777-7777

6) MAP INTERFACE
6.1) MAP IMPLEMENTATIONS

- A `HashMap` stores the keys in a hash table.
- **This means that it uses the hashCode() method of the keys to retrieve their values more efficiently.**
- **The main benefit is that adding elements and retrieving the element by key both have constant time.**
- **The tradeoff is that you lose the order in which you inserted the elements**.
- Most of the time, you aren't concerned with this in a map anyway.
- If you were, you could use `LinkedHashMap`.
- A `TreeMap` stores the keys in a sorted tree structure.
- **The main benefit is that the keys are always in sorted order.**
- **The tradeoff is that adding and checking if a key is present are both O(log n).**

6) MAP INTERFACE
6.2) MAP METHODS

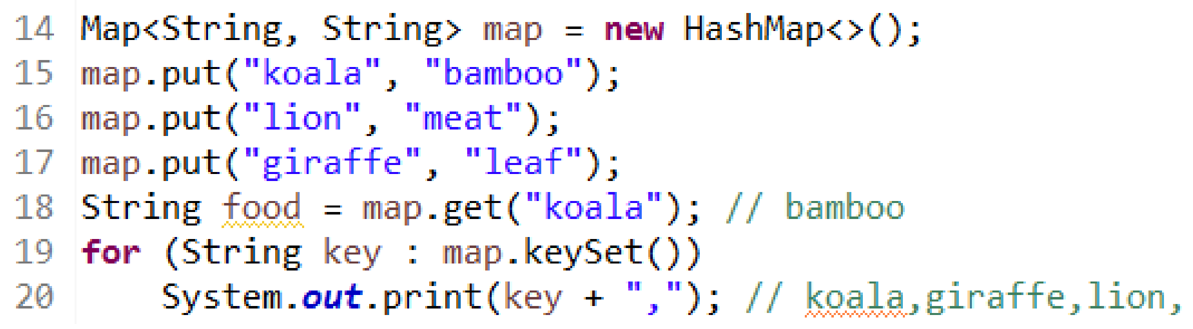
- Given that Map doesn't extend Collection, there are more methods specified on the Map interface.
- Since there are both keys and values, we need generic type parameters for both.
- The class uses K for key and V for value.
- Most of the method signatures that you need to know are in the following table.



Method	Description
`void clear()`	Removes all keys and values from the map.
`boolean isEmpty()`	Returns whether the map is empty.
`int size()`	Returns the number of entries (key/value pairs) in the map.
`V get(Object key)`	Returns the value mapped by key or `null` if none is mapped.
`V put(K key, V value)`	Adds or replaces key/value pair. Returns previous value or `null`.
`V remove(Object key)`	Removes and returns value mapped to key. Returns `null` if none.
`boolean containsKey(Object key)`	Returns whether key is in map.
`boolean containsValue(Object)`	Returns value is in map.
`Set<K> keySet()`	Returns set of all keys.
`Collection<V> values()`	Returns `Collection` of all values.

6) MAP INTERFACE
6.2) MAP IMPLEMENTATIONS

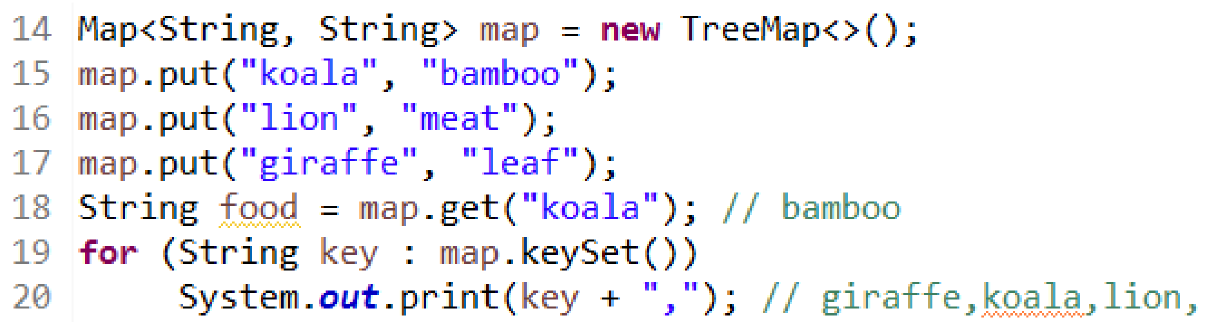
- First let’s look at **`HashMap`**:



```java
Map<String, String> map = new HashMap<>();
map.put("koala", "bamboo");
map.put("lion", "meat");
map.put("giraffe", "leaf");
String food = map.get("koala"); // bamboo
for (String key : map.keySet())
    System.out.print(key + ","); // koala,giraffe,lion,
```

- Java uses the **`hashCode()`** of the key to determine the order.

- Now let’s look at **`TreeMap`**:



```java
Map<String, String> map = new TreeMap<>();
map.put("koala", "bamboo");
map.put("lion", "meat");
map.put("giraffe", "leaf");
String food = map.get("koala"); // bamboo
for (String key : map.keySet())
    System.out.print(key + ","); // giraffe,koala,lion,
```

- **`TreeMap`** sorts the keys as we would expect.
- If we were to have called values() instead of keySet(), the order of the values would correspond to the order of the keys.

7) COMPARING COLLECTIONS

TYPE	CAN CONTAIN DUPLICATE ELEMENTS ?	ELEMENTS ORDERED ?	HAS KEYS AND VALUES ?	MUST ADD/REMOVE IN SPECIFIC ORDER ?
LIST	YES	YES (by index)	NO	NO
SET	NO	NO	NO	NO
QUEUE	YES	YES (retrieved in defined order)	NO	YES
MAP	YES	NO	YES	NO

7) COMPARING COLLECTIONS

- To pick the top book of a stack of books ?
  - **`ArrayDequeue`**
  - The description is of a **last-in first-out** data structure, so you need a **stack**, which is a type of `Queue`.
- To sell tickets to people in the order in which they appear in line and tell them their position in line ?
  - **`LinkedList`**
  - The description is of a **first-in first-ou**t data structure, so you need a **`queue`**.
  - You also needed **`indexes`**, and **`LinkedList`** is the only class to match both requirements.
- To write down the first names of all of the elephants so that you can tell them to your friend’s three-year-old every time she asks. (The elephants do not have unique first names.)
  - **`ArrayList`**
  - Since there are duplicates, you need a list rather than a set. You will be accessing the list more often than updating it, since three-year-olds ask the same question over and over, making an ArrayList better than a LinkedList.
- To list the unique animals that you want to see at the zoo today
  - **`HashSet`**
  - The keyword in the description is **unique**. When you see **“unique,” think “set.”**
  - Since there were no requirements to have a sorted order or to remember the insertion order, you use the most efficient set.

7) COMPARING COLLECTIONS

- To list the unique animals that you want to see at the zoo today in alphabetical order ?
  - **`TreeSet`**
  - Since it says **“unique”,** you need a **set**.
  - This time, you need to sort, so you cannot use a **`HashSet`**.
- To look up animals based on a unique identifier ?
  - **`HashMap`**
  - Looking up by key should make you think of a map.
  - Since you have no ordering or sorting requirements, you should use the most basic map.