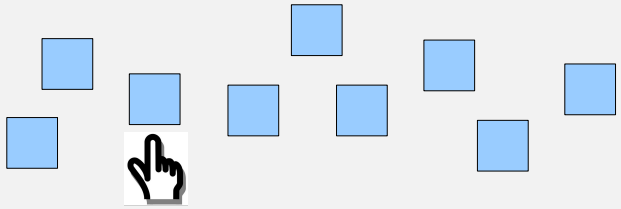
Iterators

To go through a collection of items one at a time, you might imagine a finger moving through the collection, pointing at each item in turn.



This is an iterator – the current position of the finger plus the ability to move the finger to the next thing.

Question	Java syntax for accessing each element of a collection one by one	an abstraction that provides a way to access each element of a collection one by one	the process of accessing each element of a collection one by one
for-each loop	✓ 6	1	0
iterator	1	✓ 5	1
traversal	0	1	✓ 6

"iterator" in general refers to the abstraction, but there is a specific Java type `Iterator`

Iterators

All iterators in Java are type `Iterator<E>`.

Using an iterator –

```
for ( Iterator<E> iter = mycollection.iterator() ;
      iter.hasNext() ; ) {
  E elt = iter.next();
  …
}
```

- all Collections classes have an `iterator()` method which returns an iterator initialized so the finger is pointing just before the first thing
- `hasNext()` asks whether there are any more elements after where the finger is pointing
- `next()` advances the finger to the next element and returns it
- *E* is replaced by the actual type of whatever is in the collection – don't actually type E!
  - when used alone and not as a type parameter, can use the corresponding primitive type e.g. `int`

Iterators

There's also a special form of `for` loop ("for-each loop") which is often more convenient –

```
for ( E elt : mycollection ) {
  …
}
```

- `elt` takes on the values of each element in the collection in turn

for-each loops can also be used with arrays –

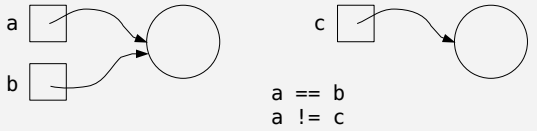
```
int[] numbers = { … };
for ( int num : numbers ) {
  …
}
```

Example

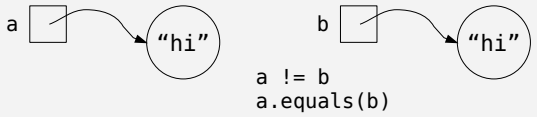
- `Reverser`
- `ListDemo`

Equality Comparisons – Objects

- *referential equality* refers to whether two things are the same object i.e. at the some location in memory
  - the references are the same

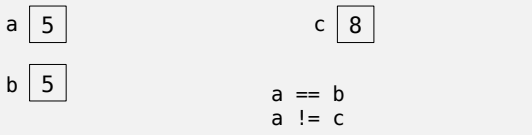


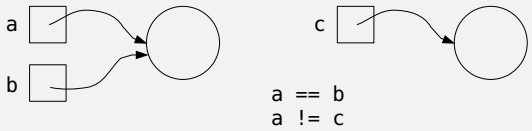
- *logical equality* refers to whether two things represent the same value i.e. are considered equivalent
  - "equivalent" depends on the specific class, so `equals()` must be implemented when appropriate – the default is to do the same thing as ==



Equality Comparisons – ==

- regardless of type, == compares what is in the box
  - for primitive types, this compares values
  - for objects, this means referential equality because references are in the box instead of values





Which of the following statements about == and `.equals()` are true? Choose all that apply.

	Answer	Respondents	Percentage
✓	for objects, a == b checks for reference equality, that is, whether a and b point to the same memory location	5	25%
✓	for primitive types, a == b checks whether a and b have the same value	6	30%
✓	for objects, a.equals(b) checks for logical equality, that is, whether a and b represent the same value	5	25%

Which of the following statements about == and `.equals()` are true?
Choose all that apply.

✗	== can only be used with primitive types - for example, if a and b are Strings, a == b would be illegal	3	15%
✓	.equals() can only be used with primitive types - for example, if a and b are ints, a.equals(b) would be illegal	1	5%
✗	you should never implement equals() for a class because it is already appropriately defined for all classes	0	0%

== is valid for objects such as strings, it just checks referential equality instead of logical equality

should read "`.equals()` can only be used with *object* types" – `a.equals(b)` is illegal if a, b are primitive types like `ints`

Collections ADTs

Containers –
- List
- Stack
- Queue

characterized by the idea of elements arranged in a line (ordered, not necessarily sorted)

elements accessed by position

Ordered containers –
- PriorityQueue

elements are ordered by priority

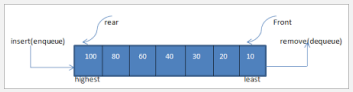
Lookup and membership –
- Dictionary (Map)
- Set

given a piece of information (key), find an associated piece of information (value)

Question	a collection of items without duplicates	a collection where items are removed in order of their precedence or importance, rather than their order of insertion	an associative array, that is a collection storing key-value pairs, where values are accessed by their key	sequence of items arranged in a linear order	a collection where elements are removed in the opposite order they were inserted (LIFO)	a collection where items are removed in the same order in which they were inserted (FIFO)
Set	✓ 6	0	1	0	0	0
PriorityQueue	0	✓ 7	0	0	0	0
Map	0	0	✓ 6	1	0	0
List	0	0	1	✓ 6	0	0

Java Collections – Ordered Containers

- `java.util.PriorityQueue<E>`
  - similar to a queue, but elements are ordered – smallest element is at the front



Key operations –
- add(e)
- peek()
- remove(), poll()
  - `remove()` throws an exception if the queue is empty, `poll()` returns `null` instead
- contains(e)
- clear()
- size()
- iterator()

Comparing Elements

- option #1 – the elements stored in the priority queue implement `Comparable<E>`
  - appropriate when elements of type E have a *natural ordering*
  - `Integer`, `String`, etc implement `Comparable`

```
public class MyClass implements Comparable<MyClass> {
  …
  public int compareTo ( MyClass other ) {
    // return negative value if 'this' comes first,
    // positive value if 'other' comes first, and 0
    // if equal
    …
  }
}


PriorityQueue<MyClass> pq = new PriorityQueue<MyClass>();
```


Comparing Elements

- option #2 – specify a `Comparator<E>`
  - appropriate when there are multiple ways elements of type E can be compared, or to compare objects not defined as `Comparable`

```
public class MyClassComparator implements
  Comparator<MyClass> {
  public int compare ( MyClass obj1, MyClass obj2 ) {
    // return negative value if 'obj1' comes first,
    // positive value if 'obj2' comes first, and 0
    // if equal
    …
  }
}

PriorityQueue<MyClass> pq =
 new PriorityQueue<MyClass>(new MyClassComparator());
```


Example

- `Sorter`
- `LengthSorter`

Java Collections – Lookup and Membership

Lookup tasks involve finding the value associated with a particular key.

- indexing in an array a[i] is a form of lookup – the index i is the key, the value stored in the array is the value
- an *associative array* is a generalization of an array where the index can be anything, not just an integer 0..n-1

A *set* is an unordered collection of elements.

- "unordered" means that there isn't a notion of 1st, 2nd, 3rd, …

Duplicates are not allowed.

- key operation is *contains* – does an element belong to the set?

Map in the Java Collections Framework

Map<K,V>

	`void`	**`clear()`** Removes all of the mappings from this map (optional operation).
★	`boolean`	**`containsKey(Object key)`** Returns `true` if this map contains a mapping for the specified key.
	`boolean`	**`containsValue(Object value)`** Returns `true` if this map maps one or more keys to the specified value.
	`Set<Map.Entry<K,V>>`	**`entrySet()`** Returns a **`Set`** view of the mappings contained in this map.
	`boolean`	**`equals(Object o)`** Compares the specified object with this map for equality.
★	`V`	**`get(Object key)`** Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
	`int`	**`hashCode()`** Returns the hash code value for this map.
	`boolean`	**`isEmpty()`** Returns `true` if this map contains no key-value mappings.
★	`Set<K>`	**`keySet()`** Returns a **`Set`** view of the keys contained in this map.
★	`V`	**`put(K key, V value)`** Associates the specified value with the specified key in this map (optional operation).
	`void`	**`putAll(Map<? extends K,? extends V> m)`** Copies all of the mappings from the specified map to this map (optional operation).
	`V`	**`remove(Object key)`** Removes the mapping for a key from this map if it is present (optional operation).
★	`int`	**`size()`** Returns the number of key-value mappings in this map.
	`Collection<V>`	**`values()`** Returns a **`Collection`** view of the values contained in this map.

Set in the Java Collections Framework

Set<E>

	`boolean`	**`add(E e)`**	Adds the specified element to this set if it is not already present (optional operation).
	`boolean`	**`addAll(Collection<? extends E> c)`**	Adds all of the elements in the specified collection to this set if they're not already present (optional operation).
	`void`	**`clear()`**	Removes all of the elements from this set (optional operation).
★	`boolean`	**`contains(Object o)`**	Returns `true` if this set contains the specified element.
	`boolean`	**`containsAll(Collection<?> c)`**	Returns `true` if this set contains all of the elements of the specified collection.
	`boolean`	**`isEmpty()`**	Returns `true` if this set contains no elements.
★	`Iterator<E>`	**`iterator()`**	Returns an iterator over the elements in this set.
★	`boolean`	**`remove(Object o)`**	Removes the specified element from this set if it is present (optional operation).
	`boolean`	**`removeAll(Collection<?> c)`**	Removes from this set all of its elements that are contained in the specified collection (optional operation).
	`boolean`	**`retainAll(Collection<?> c)`**	Retains only the elements in this set that are contained in the specified collection (optional operation).
★	`int`	**`size()`**	Returns the number of elements in this set (its cardinality).

Java Collections – Lookup and Membership

- interface `Map<K,V>`
- concrete classes `HashMap<K,V>`, `TreeMap<K,V>`
  - `HashMap` is usually what you want – use `TreeMap` only if you need to access the keys in sorted order

- interface `Set<E>`
- concrete classes `HashSet<E>`, `TreeSet<E>`
  - `HashSet` is usually what you want – use `TreeSet` only if you need to access the keys in sorted order

Map in the Java Collections Framework

- concrete classes – for creating new objects
  - `HashMap<K,V>`
    - elements are stored in a hashtable – key is converted to an array index using a *hash function*; the value is stored in that slot
    - essentially O(1) `put`, `get`, `remove`, `containsKey` ← very fast no matter how many things are in the Map
    - `equals()` is used to test for equality
  - `TreeMap<K,V>`
    - elements are stored in a balanced binary search tree – keys can be accessed in sorted order
      - keys must implement `Comparable` or else a `Comparator` must be supplied to the `TreeMap` constructor
    - O(log n) `put`, `get`, `remove`, `containsKey` ← pretty fast, but takes longer with more things in the Map
    - `compareTo()` is used to test for equality

`HashMap` is usually what you want – use `TreeMap` only if you need to access the keys in sorted order.

Example

- `LuckyNumbers`