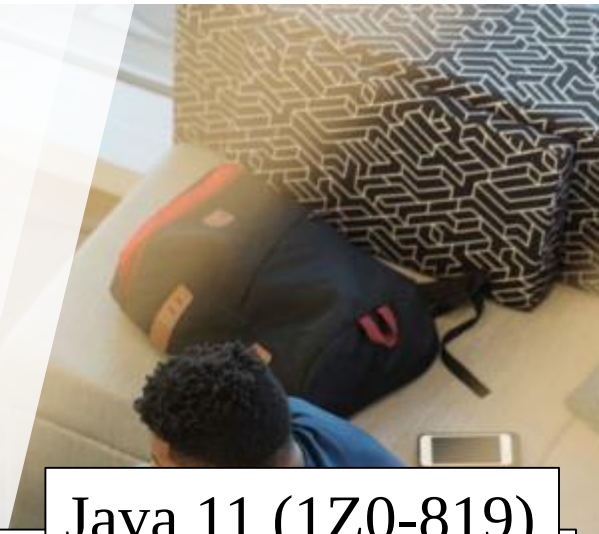
Collections

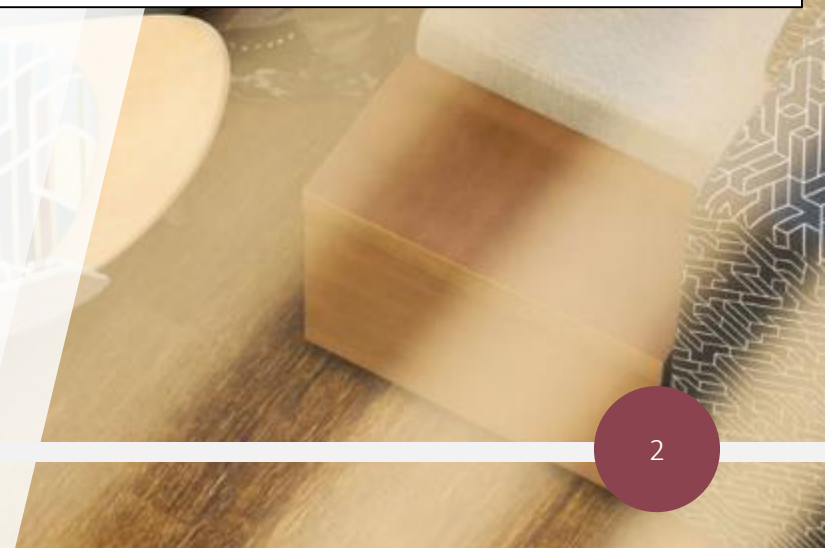
Collections



Java 11 (1Z0-819)

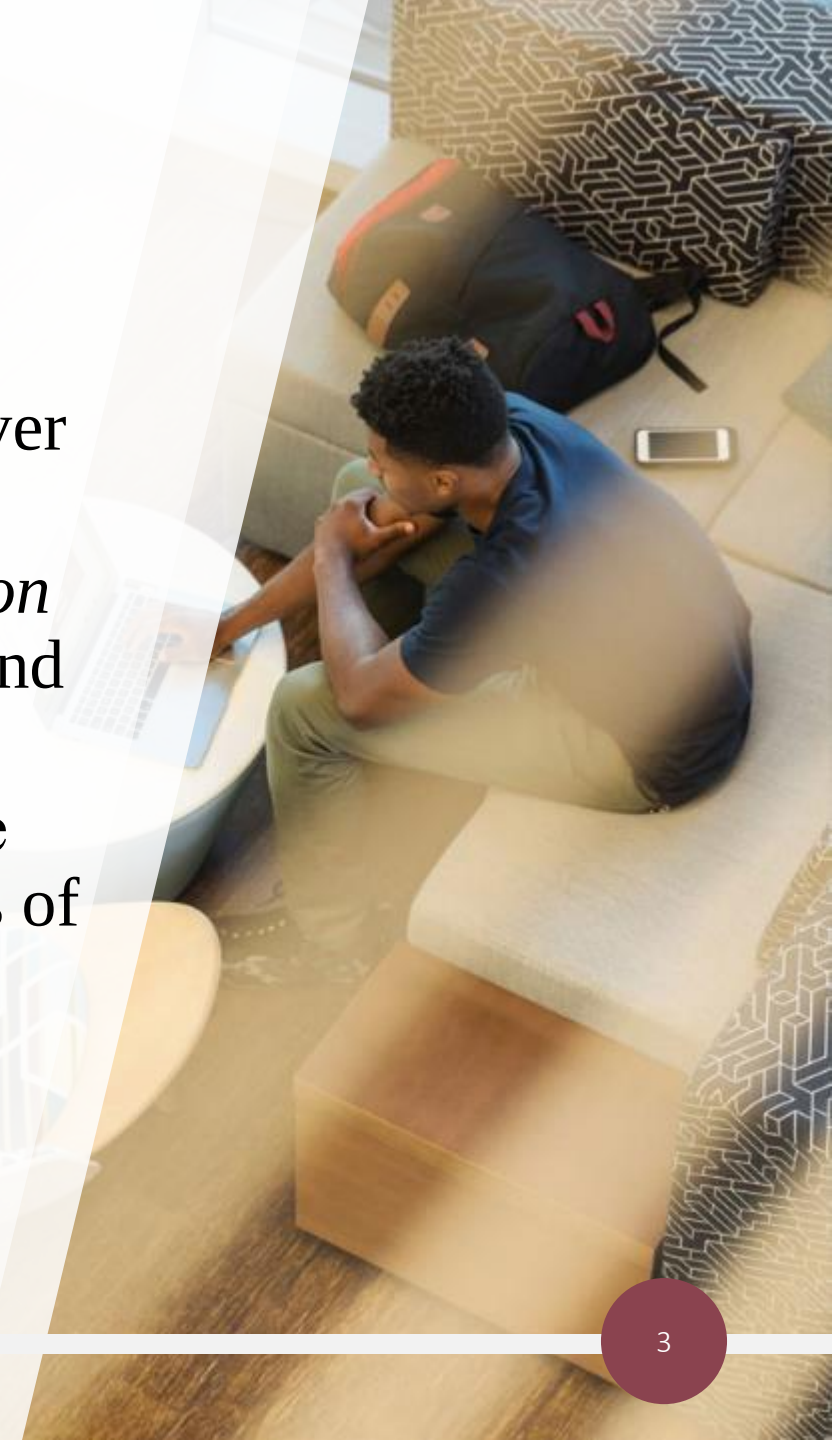
Working with Arrays and Collections

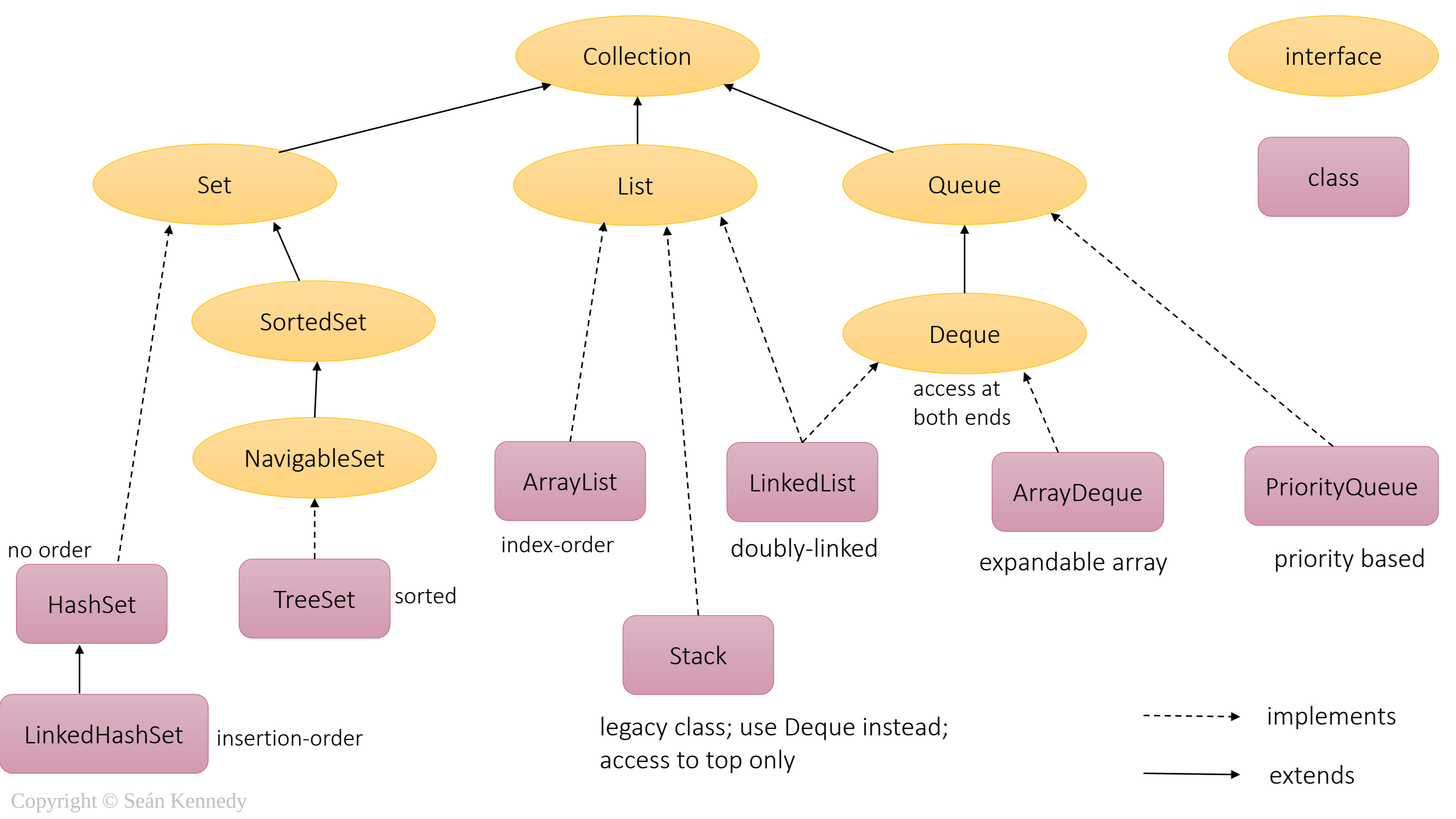
- ✓ Use generics, including wildcards
- ✓ Use a Java array and List, Set, Map and Deque collections, including convenience methods
- ✓ Sort collections and arrays using Comparator and Comparable interfaces

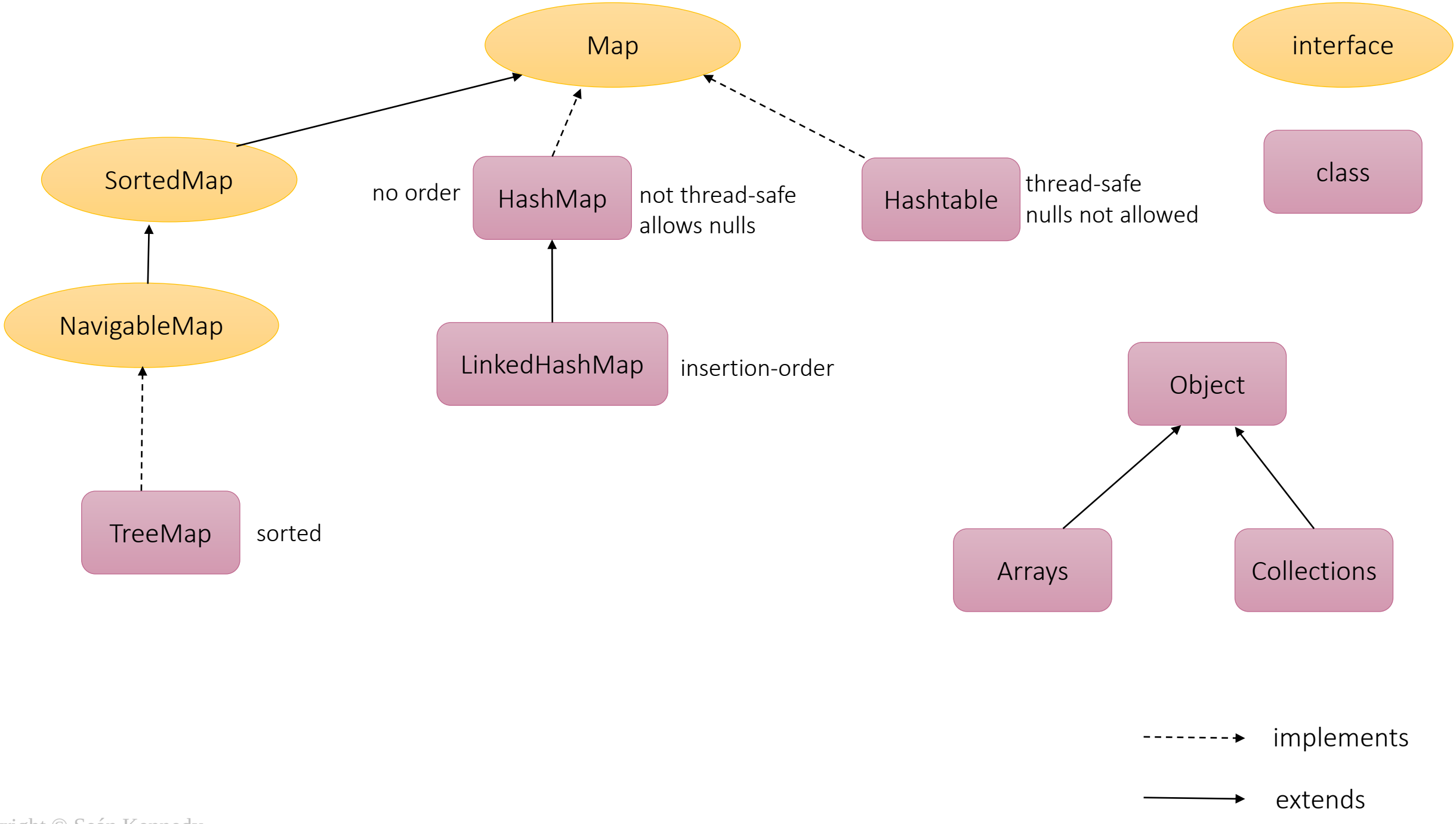


Collections

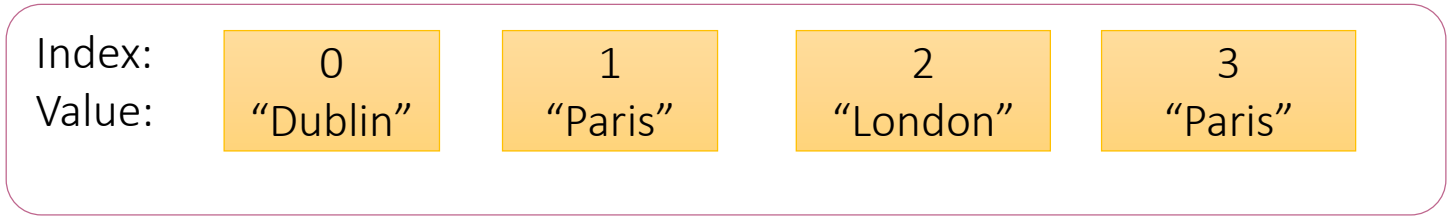
- Differentiate the following:
  - collection - lowercase ‘c’; represents any of the data structures in which objects are stored and iterated over
  - Collection - capital ‘C’; this is the *java.util.Collection* interface that the *Set*, *List* and *Queue* interfaces extend
  - Collections - capital ‘C’ and ends with ’s’; this is the *java.util.Collections* utility class which contains lots of *static* methods for use with collections





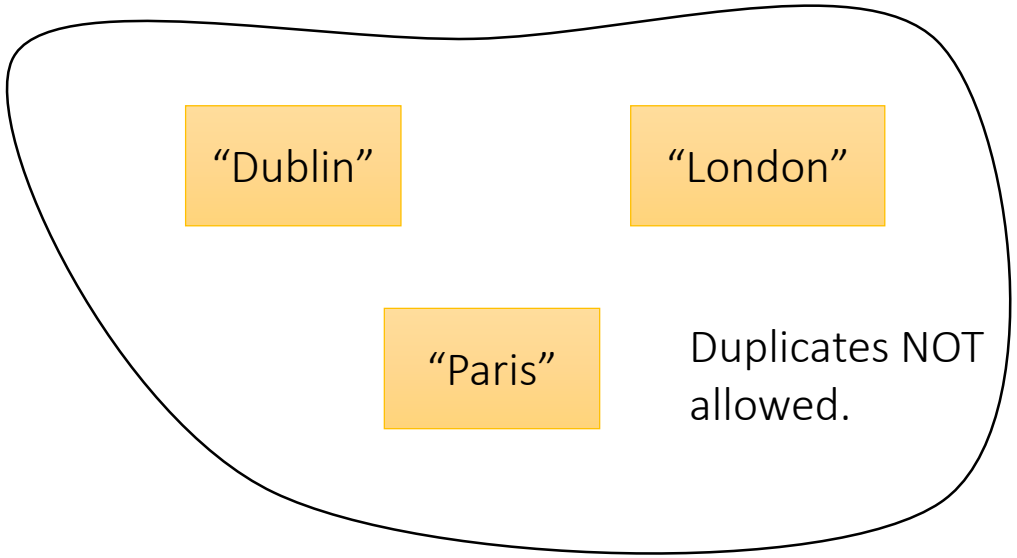


List:

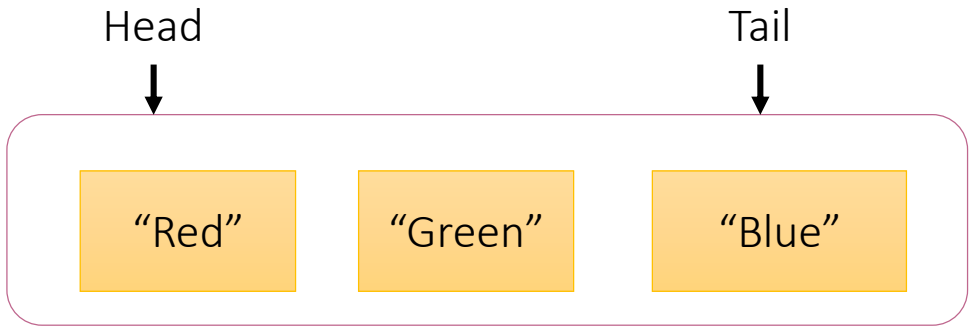


Duplicates allowed.

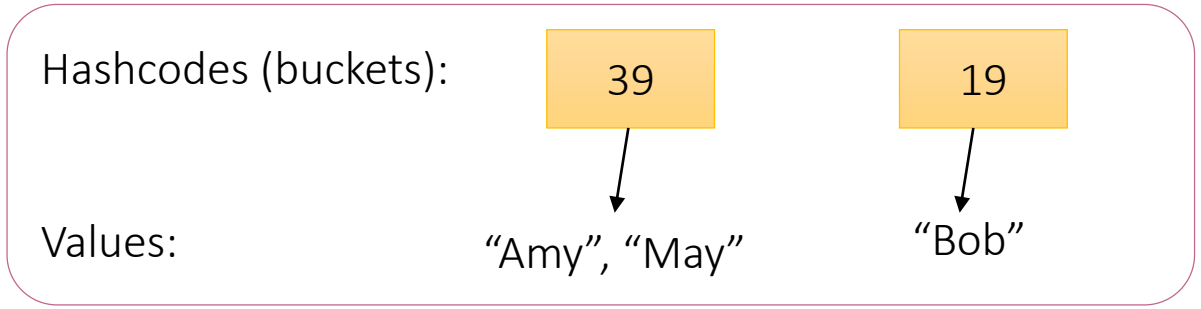
Set:



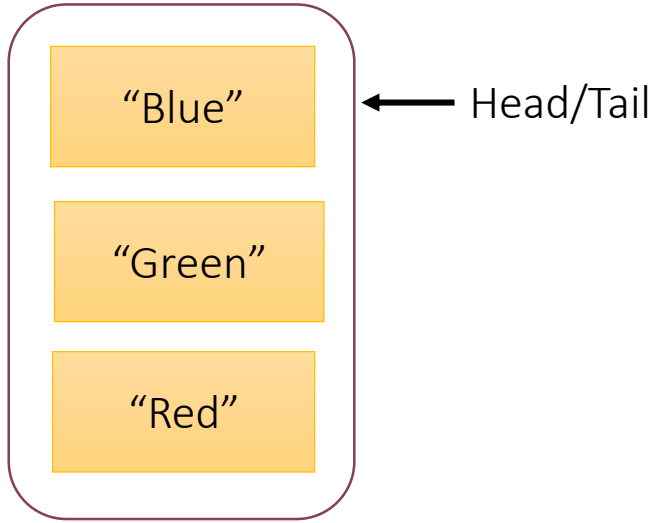
FIFO Queue:



Map:



LIFO Queue (Stack):



Popular *Collection* Methods

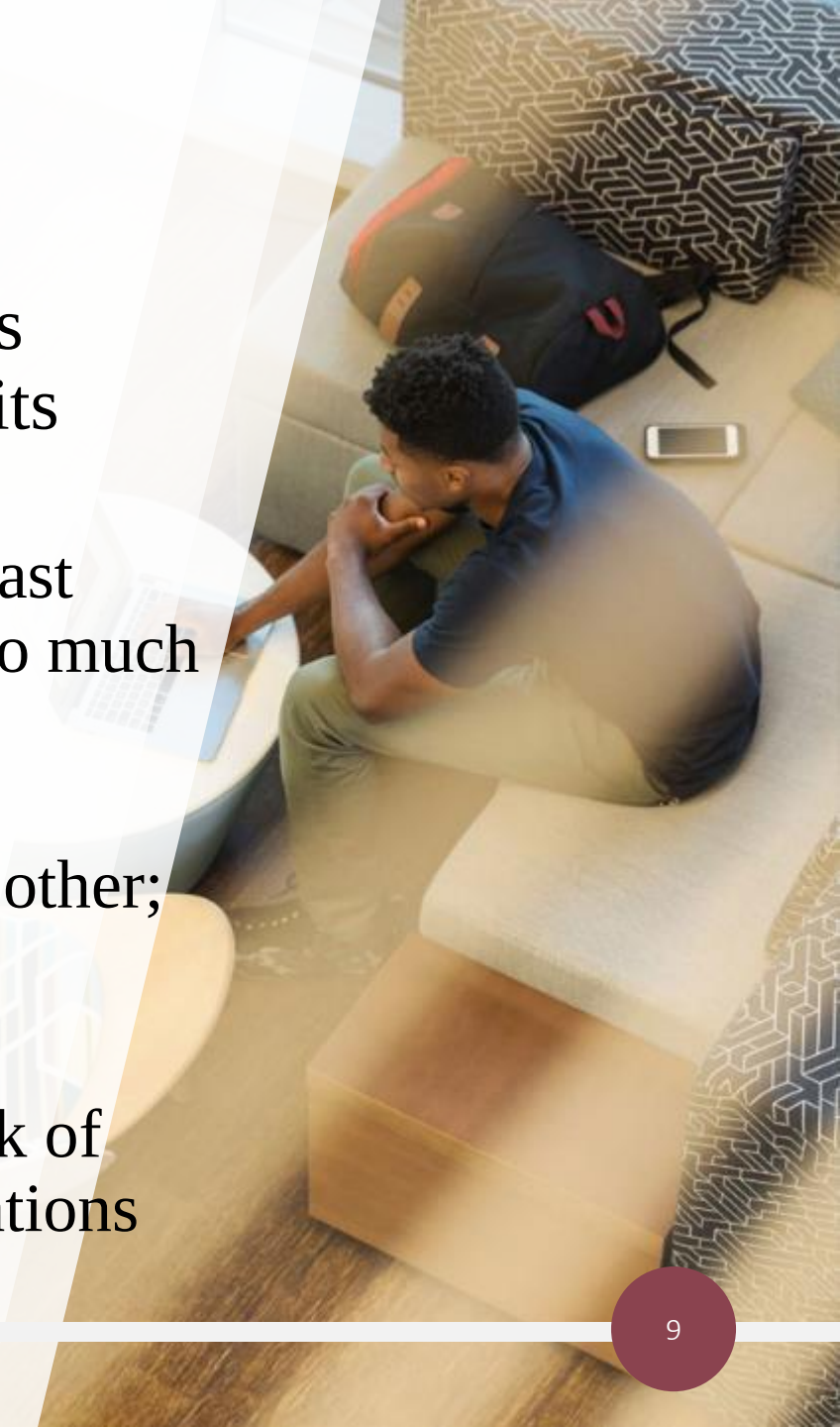
boolean	add(E element)	adds the element to the end
boolean	remove(Object o)	removes a single instance of the element specified
boolean	isEmpty()	returns true if the collection contains no elements
int	size()	returns the number of elements in the collection
void	clear()	removes all of the elements
boolean	contains(Object o)	does the collection contain the specified element
boolean	removeIf(Predicate<? super E> p)	removes all elements that match the condition
void	forEach(Consumer<? super T> c) Note: this method is a *default* method in the *Iterable* interface and *Collection* extends *Iterable.*	performs the given action on all elements in the collection

Code: CommonCollectionMethods.java

CommonCollectionMethods.java

Collections - *List*

- Collections have four basic flavours:
  - *List* – an ordered collection (sequence); provides precise control over access to an element using its integer index; duplicate elements are allowed
    - *ArrayList* - a growable array; fast iteration and fast random access; use when you are not likely to do much insertion/deletion (shuffling required).
    - *LinkedList* - elements are doubly-linked to each other; fast insertion/deletion.
    - *Stack* – represents a last-in-first-out (LIFO) stack of objects. The *Deque* interface and its implementations are more complete and should be used instead.



Popular Factory Methods

List<T>	Arrays.asList(T… a)	returns a fixed-size list “backed” by the array i.e. changes to the list write through to the array and vice versa; cannot add/delete elements but can replace elements
List<E>	List.of(E… elements)	returns an immutable list containing the elements specified
List<E>	List.copyOf(Collection<? extends E> coll)	returns an immutable list containing the elements of the given collection

Code: UsingLists.java

Popular *List* Methods

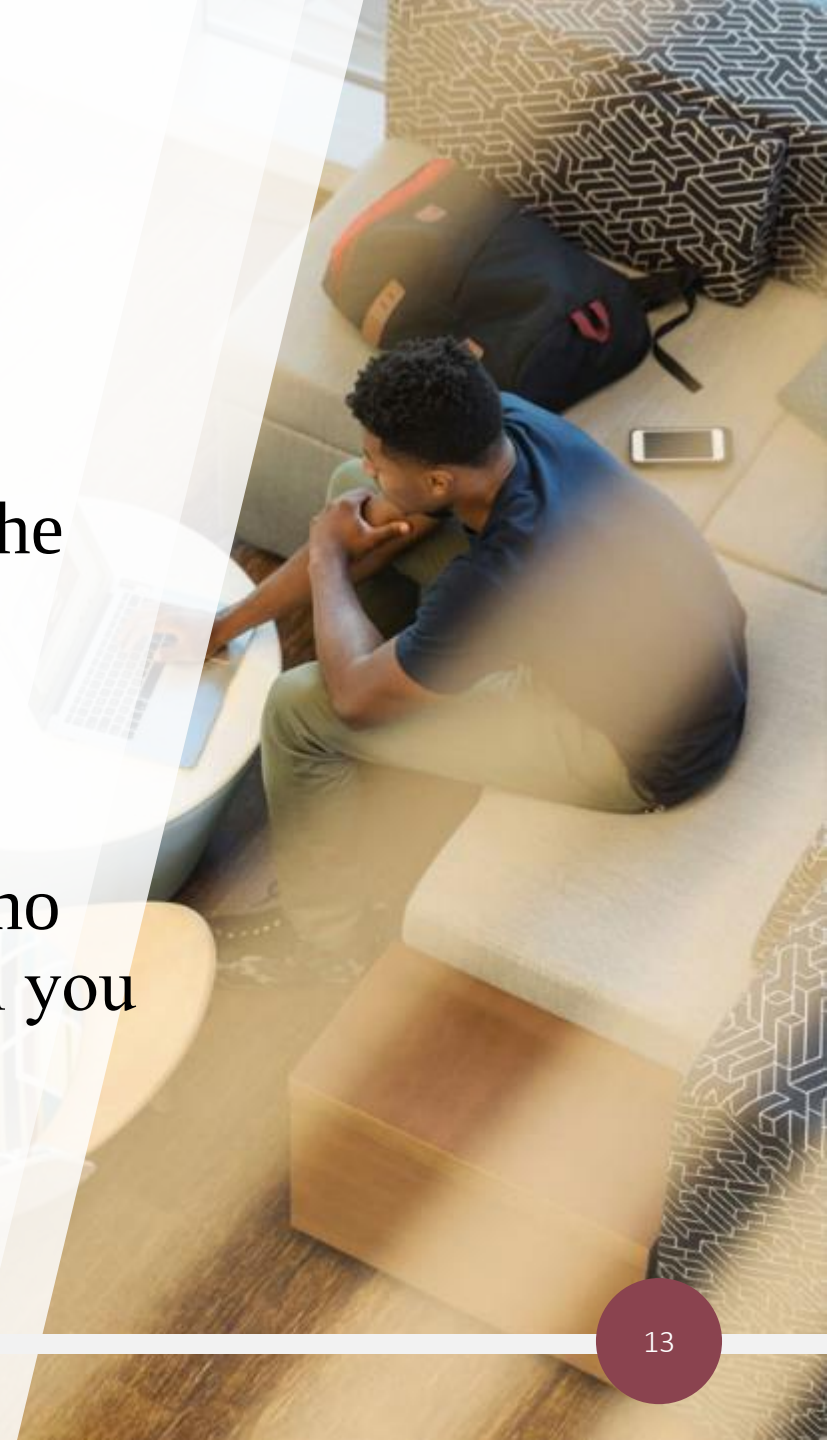
void	add(int index, E element)	adds the element at the index and moves the rest down one place
E	get(int index)	returns the element at that index
E	remove(int index)	removes the element at that index and moves the rest up one place
void	replaceAll(UnaryOperator<E> op)	replaces each element in the list by applying the operator
E	set(int index, E e)	replaces the element at that index with the specified element (the original is returned)

Code: UsingLists.java

UsingLists.java

Collections - *Set*

- *Set* – collections with no duplicate elements.
  - *HashSet*
    - unsorted, unordered *Set;* uses the hashcode of the object being inserted; the more efficient your *hashCode()* implementation, the better access performance you will get.
    - use this class when you want a collection with no duplicates and you don't care about order when you iterate through it.



Collections - *Set*

- *Set* – collections with no duplicate elements.
  - *LinkedHashSet*
    - an ordered version of *HashSet* (insertion order).
    - elements are doubly-linked to each other
    - use this class instead of *HashSet* when you care about the iteration order.

Collections - *Set*

- *Set* – collections with no duplicate elements.
  - *TreeSet*
    - a sorted collection (“Tree”)
    - elements can be sorted according to their “natural order” - for *String*’s, the natural order is alphabetic; for *Integer*’s, the natural order is numeric.
    - elements can also be sorted according to a custom order by providing a comparator at creation time.

Popular *Set* Methods

Set<E>	Set.of(E… elements)	returns an immutable set containing the elements specified
Set<E>	Set.copyOf(Collection<? extends E> coll)	returns an immutable set containing the elements of the given collection

Code: UsingSets.java

UsingSets.java

Collections - *Map*

- *Map* – maps keys to values; keys are unique; each key can map to at most one value.
  - *HashMap*
    - unsorted, unordered *Map*.
    - uses the hashcode of the object being inserted; the more efficient your *hashCode()* implementation, the better access performance you will get.
    - use this class when you want a *Map* and you don't care about order when you iterate through it.
    - allows one *null* key and multiple *null* values.

Collections - *Map*

- *Map* – (unique) keys maps to values; each key can map to at most one value.
  - *LinkedHashMap*
    - maintains insertion order
  - *TreeMap*
    - a sorted *Map*; sorted by natural order of it's keys or by a custom order (via a comparator).
  - *Hashtable*
    - similar to *HashMap* except *Hashtable* is thread-safe (slower) and nulls are not allowed

Popular *Map* Methods

void	clear()	removes all keys and values from the map
boolean	containsKey(Object key)	is the key in the map
boolean	containsValue(Object value)	is this value in the map
Set<Map.Entry<K,V>>	entrySet()	returns a Set view of the key/value pairs
void	forEach(BiConsumer(key, value)	perform the given BiConsumer on each entry in the map
V	get(Object key)	returns the value for the specified key or null if no mapping exists
boolean	isEmpty()	is the map empty

Code: UsingMaps.java

Popular *Map* Methods

Set<K>	keySet()	returns a Set view of all the keys in the map
V	put(K key, V value)	adds or replaces the key/value pair. Returns previous value or null.
V	putIfAbsent(K key, V value)	adds key/value pair if key not there already and returns null; otherwise, returns existing value.
V	remove(Object key)	removes the entry if the key exists and returns the value that was there; returns null if key not in map.
V	replace(K key, V value)	replaces the value for the key and returns the old value; returns null if key not in map
void	replaceAll(BiFunction<K,V,V> fn)	replaces each value with the results of the function.
int	size()	how many key/value pairs in the map
Collection<V>	values()	returns a Collection view of all the values

Code: UsingMaps.java

UsingMaps.java

Collections - *Queue*

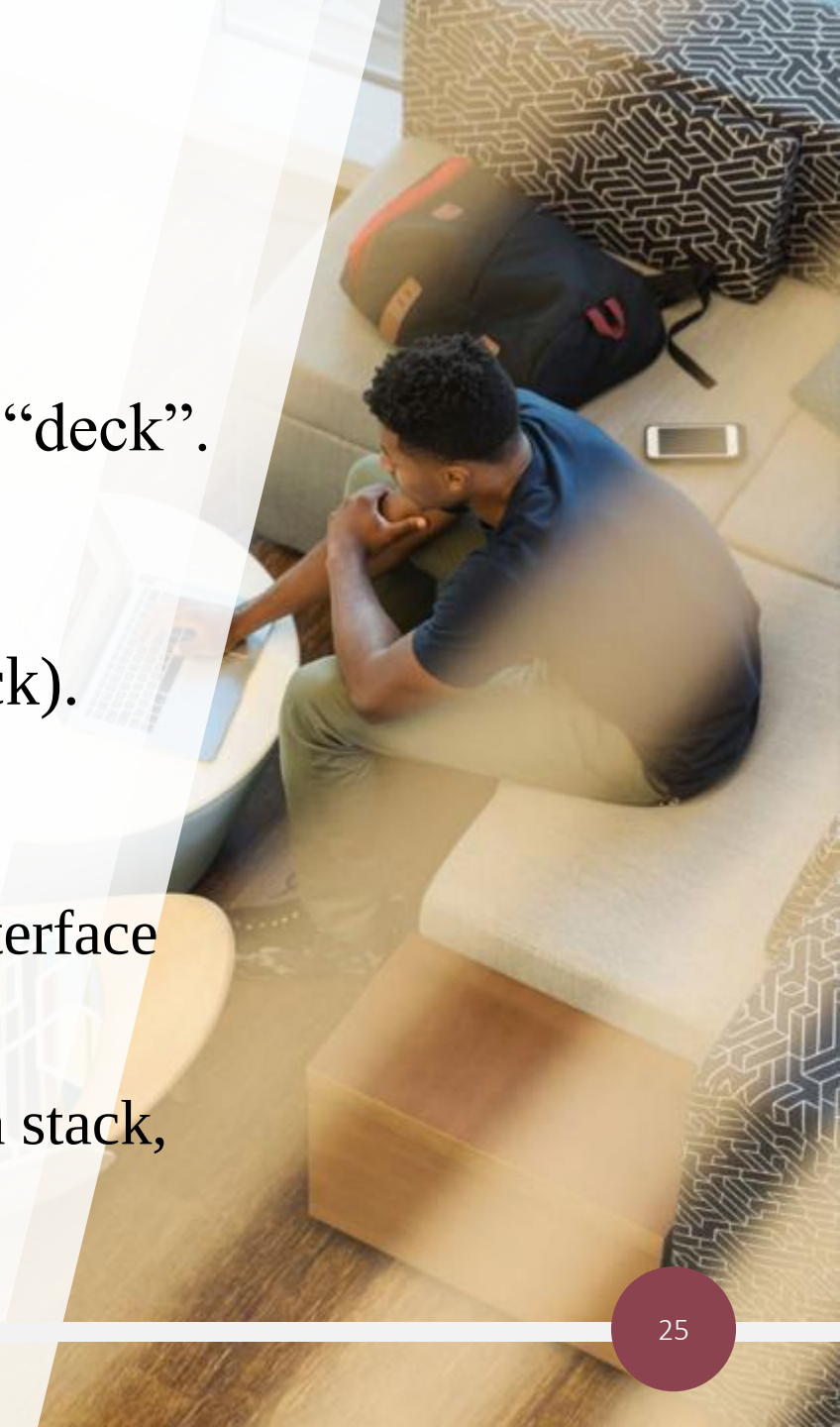
- *Queue* - a collection that specifies the order in which elements are to be processed.
  - Typically the order is FIFO (First In First Out).
  - Exceptions are priority queues (order is natural ordering or according to a supplied comparator) and LIFO (Last In First Out) queues (stacks).
  - *LinkedList*
    - as *LinkedList* implements *Queue;* basic queues can be handled with a *LinkedList*

Collections - *Queue*

- *PriorityQueue*
  - *PriorityQueue* orders the elements relative to each other such that “priority-in, priority-out” (as opposed to a FIFO or LIFO).
    - the elements are either ordered by natural order or by a custom order via a comparator.
    - elements that are sorted first will be accessed first.

Collections - *Queue*

- *Deque*
  - deque (“double ended queue”) and is pronounced “deck”.
  - access from both ends permitted.
  - can be used as both FIFO (queue) and LIFO (stack).
  - *ArrayDeque*
    - expandable-array implementation of the *Deque* interface (no capacity restrictions).
    - API: “likely to be faster than *Stack* when used as a stack, and faster than *LinkedList* when used as a queue”.



Popular *Queue* Methods

	Throws exception	Returns special value
Examine	element()	peek()
Insert	add(e)	offer(e)
Remove	remove()	poll()

The most common methods are *peek()*, *offer()* and *poll()* as they do not throw exceptions. POP is useful for remembering them.

Code: UsingQueues.java

Popular *Deque* Methods

	Head (First Element)		Tail (Last Element)	
	Throws exception	Returns special value	Throws exception	Returns special value
Examine	getFirst()	peekFirst()	getLast()	peekLast()
Insert	addFirst(e)	offerFirst(e)	addLast(e)	offerLast(e)
Remove	removeFirst()	pollFirst()	removeLast()	pollLast()

Code: UsingQueues.java

Using *Deque* as a queue

	Queue method	Equivalent Deque method
Examine	element()	getFirst()
	peek()	**peekFirst()**
Insert (end)	add(e)	addLast(e)
	offer(e)	**offerLast(e)**
Remove (front)	remove()	removeFirst()
	poll()	**pollFirst()**

Using *Deque* as a stack

Stack method	Equivalent Deque method
push(e)	addFirst(e)
pop()	removeFirst()
peek()	getFirst()

Beginning of deque is the "top" of the stack