Java Syntax Notes

Primitive Types

Array

One dimensional:

```
int[] myArr = new int[10];               // Integer array of 10 elements
```

Two dimensional:

```
int[][] myArr = new int[10][20];         // 10 rows, 20 columns
```

Array literals:

```
int[] myArr = new int[]{1, 2, 3};        // Length calculated during creation

int[] myArr = {1, 2, 3};

int[][] myArr = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} }   // 3 rows & cols
```

Accessing:

```
for (int i = 0; i < myArr.length; i++)   { System.out.print(myArr[i]; }
for (int i : myArr)                      { System.out.print(i); }
```

String

Creation:

```
String gopha = new String(“ok”);
```

String literal:

```
String gopha = “ok”;
```

Size:

```
gopha.length();
```

Accessing:

```
char[] chArr = gopha.toCharArray();
for (char c : chArr) { System.out.print(c); }

for (int i = 0; i < gopha.length(); i++) { System.out.print(gopha.charAt(i)); }
```

Collections

HashMap

A data structure that maps keys to values. A map cannot contain duplicate keys and each key can map to at most one value.

Import required:

```
import java.util.HashMap;
```

Creation:

```
HashMap<String, String> hm = new HashMap<>();
```

Add element:

```
hm.put(“gopha”, “ok”);          // Key is “gopha”, value is “ok”
```

Update element:

```
hm.put(“gopha”, hm.getOrDefault(“gopha”, “run”));
```

* Note: Attempts to retrieve the value for the key “*gopha*”. If not present, “*run*” will be used instead and saved for the respective key of “*gopha*”

Remove element:

```
hm.remove(“gopha”);             // Specify key to remove the entire entry
```

Size:

```
hm.size();
```

Accessing:

```
for (Map.Entry<String, String> entry : hm.entrySet()) {
        System.out.println(entry.getKey() + “ “ + entry.getValue());
}

for (String key : hm.keySet()) { System.out.println(key); }
for (String value : hm.values()) { System.out.println(value); }
```

Time Complexity:

- Access: O(1)
- Search: O(n)
- Insert: O(1)
- Remove: O(1)

HashSet

A collection that uses a Hash table for storage, only allowing unique elements to be added.

Import required: import java.util.HashSet;

Creation: HashSet<String> hs = new HashSet<>();

Add element: hs.add(“gopha ok”);

Remove element: hs.remove(“gopha ok”);

Search element: hs.contains(“gopha ok”);

Size: hs.size();

Accessing:

```
for (String s : hs) {
        System.out.println(s);
}
```

Time Complexity:

- Access: O(1)
- Search: O(1)
- Insert: O(1)
- Remove: O(1)

ArrayList

A collection of data elements sequentially ordered from 0 to length - 1. This means that we are able to access an element inside an ArrayList by its position (index).

Import required:

```
import java.util.ArrayList;
```

Creation:

```
ArrayList<Integer> list = new ArrayList<>();
List<Integer> list = new ArrayList<>();
```

Add element:

```
list.add(1);
```

Update element:

```
list.set(0, 100);        // Update index 0’s value to 100
```

Remove element:

```
list.remove(0);          // Remove index 0
list.clear();            // Remove all elements
```

Size:

```
list.size();
```

Accessing:

```
for (int i = 0; i < list.size(); i++) { System.out.println(list.get(i)); }
for (String s : list) { System.out.println(s); }
```

Sorting:

```
import java.util.Collections;
Collections.sort(list);                                  // Sort ascending
Collections.sort(list, Collections.reverseOrder());      // Sort descending
```

Time Complexity:

- Access: O(1)
- Search: O(n)
- Insert: O(1) (at the back of the ArrayList)
- Remove: O(n)

Heap

A specialized tree based structure data structure that satisfies the *heap property*: if A is a parent node of B, then the key (the value) of node A is ordered with respect to the key of node B with the same ordering applying across the entire heap.

A heap can be classified further as either a "max heap" or a "min heap". In a max heap, the keys of parent nodes are always greater than or equal to those of the children and the highest key is in the root node. In a min heap, the keys of parent nodes are less than or equal to those of the children and the lowest key is in the root node.

Import required:

```
import java.util.PriorityQueue;
```

Creation:

```
PriorityQueue<Integer> pq = new
        PriorityQueue<>(Collections.reverseOrder);         // Max heap
```

* Note: Omit "*Collections.reverseOrder*" for a min heap by default

```
PriorityQueue<Map.Entry<String, Integer>> pq = new PriorityQueue<>(
        (a, b) -> a.getValue().equals(b.getValue()) ?
                a.getKey().compareTo(b.getKey()) :
                a.getValue() - b.getValue()
);      // Max heap that contains pairs - if values for pairs are the same,
        // then they will be sorted ascending (a-z) according to key
```

Add element:

```
pq.add(10);
```

View top element:

```
pq.peek();          // Returns but does not remove the top element
```

Remove element:

```
pq.poll();          // Returns and removes the top element
```

Size:

```
pq.size();
```

Time Complexity:

- Access Max / Min: O(1)
- Insert: O(log(n))
- Remove Max / Min: O(log(n))

Queue

A collection of elements, supporting two principle operations: *enqueue*, which inserts an element into the queue, and *dequeue*, which removes an element from the queue.

Import required:	import java.util.Queue;	
Creation:	Queue<Integer> q = new LinkedList<>();	// Specify as a LinkedList!
Add element:	q.add(10);	
View top element:	q.peek();	// Returns head or null if empty
Remove element:	q.poll();	// Returns head or null if empty
Size:	q.size();	
	q.isEmpty();	// Returns true if the queue is empty

Time Complexity:

- Access: O(n)
- Search: O(n)
- Insert: O(1)
- Remove: O(1)

Stack

A collection of elements, with two principle operations: *push*, which adds to the collection, and *pop*, which removes the most recently added element.

Import required:	import java.util.Stack;	
Creation:	Stack<Integer> st = new Stack<>();	
Add element:	st.push(10);	
View top element:	st.peek();	// Returns but does not remove the top element
Remove element:	st.pop();	// Returns and removes the top element
Size:	st.size();	
	st.isEmpty();	// Returns true if the stack is empty

Time Complexity:

- Access: O(n)
- Search: O(n)
- Insert: O(1)
- Remove: O(1)

Linked List

A linear collection of data elements, called nodes, each pointing to the next node by means of a pointer. It is a data structure consisting of a group of nodes which together represent a sequence.

Import required:	import java.util.LinkedList;	
Creation:	LinkedList<Integer> list = new LinkedList<>();	
Add element:	list.add(1);	
Update element:	list.set(0, 100);	// Update index 0’s value to 100
Remove element:	list.remove(0);	// Remove index 0
	list.clear();	// Remove all elements
Size:	list.size();	
Accessing:	for (int i = 0; i < list.size(); i++) { System.out.println(list.get(i)); }	
	for (int i : list) { System.out.println(s); }	

Time Complexity:

- Access: O(n)
- Search: O(n)
- Insert: O(1)
- Remove: O(1)