Data Structures

“A data structure is a specialized format for organizing, processing, retrieving and storing data. There are several basic and advanced types of data structures, all designed to arrange data to suit a specific purpose. Data structures make it easy for users to access and work with the data they need in appropriate ways.” ~David Loshin & Sarah Lewis

Collection

A data structure which allows a variety of like objects to be dealt with together. Examples we will use in Java:

- Array
- List
- Stack
- Queue
- Map
- Set

Array (may be duplicates, access by index)

A sequence of contiguous objects in a fixed size, can be accessed by an index.

```
String[] names = new String[] { "Jed", "Amy", "Rafael", "Lin", "Oskar", "Rafael" };
names[0] = “Jedi”;
System.out.println(“First element is “ + names[0]);
System.out.println(“Last element is “ + names[names.length-1]);

for (int i = 0; i < names.length; i++) {
        // act on element at index i, and can also use i
}

for (String name : names) {
        // act on each name element, but no access to index
}
```

List (may be duplicates, access by index or value)

A sequence of objects which can be accessed by an index.

```
List<String> names = new ArrayList<String>(Arrays.asList( "Jed", "Amy",
"Rafael", "Lin", "Oskar"));
names.set(0, "Jedi");
System.out.println(“First element is “ + names.get(0));
System.out.println(“Last element is “ + names.get(names.size()-1));

if (names.indexOf("Oskar") >= 0) {
        // Do stuff
}

for (int i = 0; i < names.size(); i++) {
        // act on element at index i using get, and can also use i
}

for (String name : names) {
        // act on each name, but no access to index
}

Collections.sort(names);
Collections.reverse(names);
```

Stack (values may be duplicates)

```
Stack<String> names = new Stack<String>();
names.push("Jed");
System.out.println(“Peek at top element: “ + names.peek());
System.out.println(“Pop top element off stack + names.pop());

while (names.size() > 0) {
        System.out.println(names.pop());
}

// Looping through the stack should seldom be used, as it violates the intent
for (String name : names) {
        // act on each name, but no access to index
}
```

Queue (values may be duplicates)

```
Queue<String> names = new LinkedList<String>();
names.offer("Jed");
names.add(“Harry”);
System.out.println(“Get oldest element off queue + names.poll());

while (names.size() > 0) {
        System.out.println("Next: "+names.poll());
}

// Looping through the queue should seldom be used, as it violates the intent
for (String name : names) {
        // act on each name, but no access to index
}
```

Map (key-value pairs - keys are case-sensitive)

```
Map<String, Integer> pets = new HashSet<String, Integer>();
pets.put("Ben Langhinrichs", 1);
pets.put("Roger Rabbit", 0);

System.out.println(“Ben’s # of pets: “ + pets.get("Ben Langhinrichs"));

// if Ben gets a new pet
pets.put("Ben Langhinrichs", 2);

// Finds out if the unique key exists
if (pets.containsKey("Ben Langhinrichs")) {
        // Do stuff
}

// Finds out if there is at least one key-value pair with the value specified
if (pets.containsValue(1)) {
        // Do stuff
}


// Looping through the map requires using the Map.Entry interface
for (Map.Entry<String, Integer> pet : pets.entrySet()) {
        System.out.println(pet.getKey() + " has " + pet.getValue() + “ pets”);
}
```

Set (unique values - values are case-sensitive)

```
Set<String> desserts = new HashSet<Strings>();
desserts.add("Blueberry Pie");
desserts.add("Cassata Cake");
if (dessert.contains("Blueberry Pie")) {
        // Do stuff
}

for (String dessert : desserts) {
        System.out.println("Dessert: "+dessert);
}

Set<String> keys = pets.keySet(); // all of the keys in the Map
for (String name : keys) {
        System.out.println(name + " has " + pets.get(name) + " pets");
}
```