# Chapter 3:
# Data Structures in Java

## Title:

Data Structures in Java

## Key Words:

ArrayList, HashSet, TreeSet, HashMap, TreeMap

## Summary:

This unit provides an overview of different collections data structures used in Java.

## Outcomes:

Student will learn in this unit:

- Choosing the appropriate data structure.
- Using different available data structures in Java.

## Plan:

1 Learning Object

1. Data Structures in Java

# 1. Data Structures in Java

## Learning outcomes:

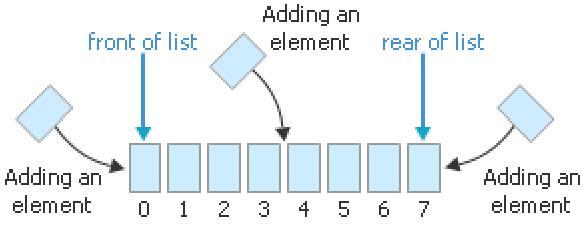
Learning main data structures in Java.

## Collections and lists:

- **collection**: an object that stores data ("**elements**")

```
    import java.util.*;   // to use Java's collections
```

- **list:** a collection of elements with 0–based **indexes**
  - elements can be added to the front, back, or elsewhere
  - a list has a **size** (number of elements that have been added)
  - in Java, a list can be represented as an `ArrayList` object



## Idea of a list:

- An ArrayList is like an array that resizes to fit its contents.
- When a list is created, it is initially empty.

  `[]`
- You can add items to the list. (By default, adds at end of list)

```
[hello, ABC, goodbye, okay]
```

The list object keeps track of the element values that have been added to it, their order, indexes, and its total size.

You can add, remove, get, set, ... any index at any time.

## Type parameters (generics):

```
ArrayList<Type> name = new ArrayList<Type>();
```

- When constructing an `ArrayList`, you must specify the type of its elements in < >
  - This is called a *type parameter* ; `ArrayList` is a *generic* class.
  - Allows the `ArrayList` class to store lists of different types.
- Example:

```
ArrayList<String> names = new ArrayList<String>();
names.add("Marty Stepp");
names.add("Stuart Reges");
```

## ArrayList methods:

| | |
|---|---|
| add(**value**) | appends value at end of list |
| add(**index, value**) | inserts given value just before the given index,<br>shifting subsequent values to the right |
| clear() | removes all elements of the list |
| indexOf(**value**) | returns first index where given value is found in list<br>(-1 if not found) |
| get(**index**) | returns the value at given index |
| remove(**index**) | removes/returns value at given index, shifting<br>subsequent values to the left |
| set(**index, value**) | replaces value at given index with given value |
| size() | returns the number of elements in list |
| toString() | returns a string representation of the list<br>such as "[3, 42, -7, 15]" |

## ArrayList vs. array:

```
String[] names = new String[5];     // construct
names[0] = "Jessica";               // store
String s = names[0];                // retrieve
for (int i = 0; i < names.length; i++) {
     if (names[i].startsWith("B")) { ... }
}                                   // iterate
```

```
ArrayList<String> list = new ArrayList<String>();
list.add("Jessica");                // store
String s = list.get(0);             // retrieve
for (int i = 0; i < list.size(); i++) {
     if (list.get(i).startsWith("B")) { ... }
}                                        // iterate
```

## ArrayList as param / return:

- Syntax:

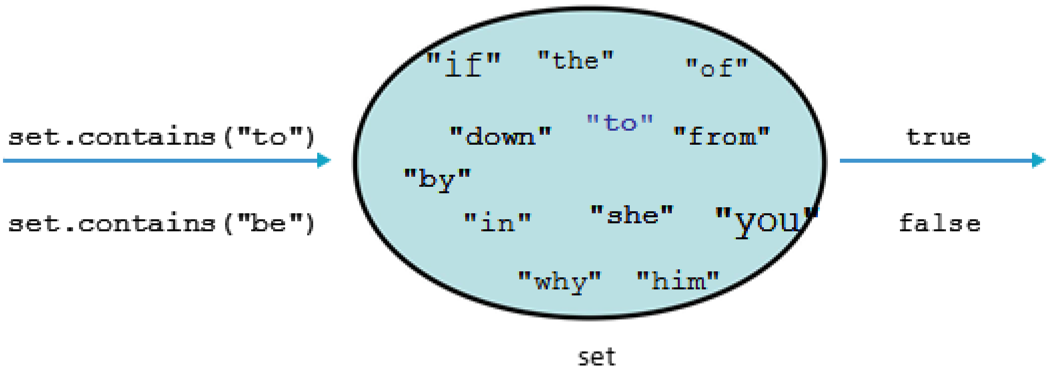
```
Public void name(ArrayList<Type> name) {          // param
Public ArrayList<Type> name(params)               // return
```

- Example:

```
   // Returns count of plural words in the given list.
   public int countPlural(ArrayList<String> list)
     { int count = 0;
     for    (int i = 0; i < list.size(); i++){
            String str = list.get(i);
                if (str.endsWith("s"))
                    { count++;
                }
          }
          return count;
   }
```

## Sets:

- **set**: A collection of unique values (no duplicates allowed) that can perform the following operations efficiently:
  - add, remove, search (contains)
  - We don't think of a set as having indexes; we just add things to the set in general and don't worry about order



## Set implementation:

- in Java, sets are represented by `Set` type in `java.util`
- `Set` is implemented by `HashSet` and `TreeSet` classes
  - `HashSet`: implemented using a "hash table" array;
  - very fast: **O(1)** for all operations
  - elements are stored in unpredictable order
  - `TreeSet`: implemented using a "binary search tree";
  - pretty fast: **O(log N)** for all operations
  - elements are stored in sorted order

## Set methods:

- Example:

```
List<String> list = new ArrayList<String>();
...
Set<Integer> set = new TreeSet<Integer>();      // empty
Set<String> set2 = new HashSet<String>(list);
```

Can construct an empty set, or one based on a given collection

| | |
|---|---|
| add(**value**) | adds the given value to the set |
| contains(**value**) | returns `true` if the given value is found in this set |
| remove(**value**) | removes the given value from the set |
| clear() | removes all elements of the set |
| size() | returns the number of elements in list |
| isEmpty() | returns `true` if the set's size is 0 |
| toString() | returns a string such as `"[3, 42, -7, 15]"` |

## The "for each" loop:

- Syntax:

```
for (type name:
        collection) {
        statements;
}
```

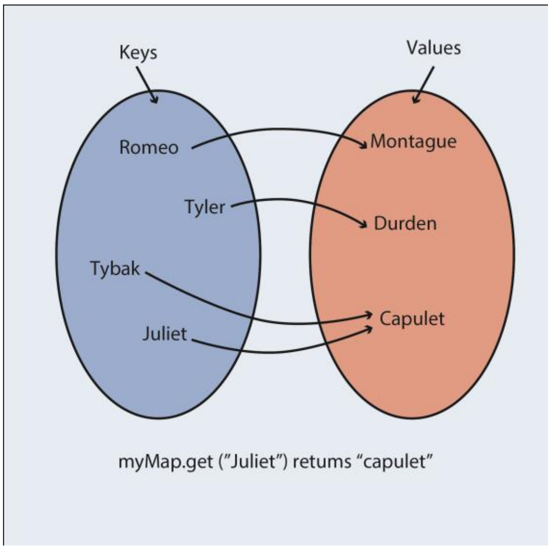
- Provides a clean syntax for looping over the elements of a `Set`, `List`, array, or other collection
- Example:

```
Set<Double> grades = new HashSet<Double>();
...
for (double grade: grades) {
  System.out.println("Student's grade: " + grade);
}
```

  - needed because sets have no indexes; can't `get` element `i`

## Maps:

- **map**: Holds a set of unique keys and a collection of *values*, where each key is associated with one value.
  - a.k.a. "dictionary", "associative array", "hash"
- basic map operations:
  - **put** (*key, value*): Adds a mapping from a key to a value.
  - **get** (*key*) : Retrieves the value mapped to the key.
  - **Remove** *(key)*: Removes the given key and its mapped value.



myMap.get ("Juliet") returns "capulet"

## Map implementation:

- in Java, maps are represented by `Map` type in `java.util`
- `Map` is implemented by the `HashMap` and `TreeMap` classes
  - `HashMap`: implemented using an array called a "hash table"; extremely fast: **O(1)**; keys are stored in unpredictable order
  - `TreeMap`: implemented as a linked "binary tree" structure; very fast: **O(log N)** ; keys are stored in sorted order
- A map requires 2 type params: one for keys, one for values.

- Example:

```
// maps from String keys to Integer values
Map<String, Integer> votes = new HashMap<String, Integer>();
```

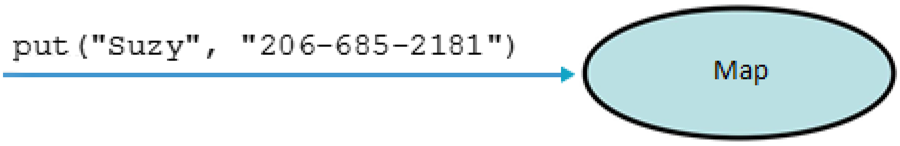
## Map methods:

| | |
|---|---|
| `put(key, value)` | adds a mapping from the given key to the given value;<br>if the key already exists, replaces its value with the given one |
| `get(key)` | returns the value mapped to the given key (`null` if not found) |
| `containsKey(key)` | returns `true` if the map contains a mapping for the given key |
| `remove(key)` | removes any existing mapping for the given key |
| `clear()` | removes all key/value pairs from the map |
| `size()` | returns the number of key/value pairs in the map |
| `isEmpty()` | returns `true` if the map's size is 0 |
| `toString()` | returns a string such as `"{a=90, d=60, c=70}"` |
| `keySet()` | returns a set of all keys in the map |
| `values()` | returns a collection of all values in the map |
| `putAll(map)` | adds all key/value pairs from the given map to this map |
| `equals(map)` | returns `true` if given map has the same mappings as this one |

## Using maps:

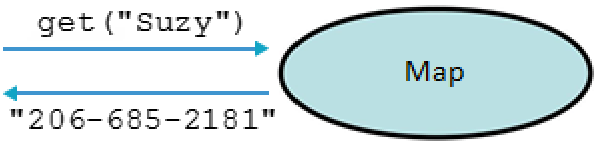
- A map allows you to get from one half of a pair to the other.
  - Remembers one piece of information about every index (key).

```
//     Keyvalue
```



Later, we can supply only the key and get back the related value:

Allows us to ask: `What is Suzy's phone number?`



## keySet and values:

- `keySet` method returns a `Set` of all keys in the map
  - can loop over the keys in a foreach loop
  - can get each key's associated value by calling `get` on the map

- Example:

```
Map<String, Integer> ages = new TreeMap<String, Integer>();
ages.put("Marty", 19);
ages.put("Geneva", 2);   // ages.keySet() returns Set<String>
ages.put("Vicki", 57);
for (String name: ages.keySet()) {       // Geneva -> 2
     int age = ages.get(name);                // Marty -> 19
     System.out.println(name + " -> " + age);  // Vicki -> 57
    }
```

- `values` method returns a collection of all values in the map
  - can loop over the values in a foreach loop
  - no easy way to get from a value to its associated key (s)

## The compareTo method:

- The standard way for a Java class to define a `comparison` function for its objects is to define a compareTo method.
  - Example: in the `String` class, there is a method:
    `public int compareTo(String other)`
- A call of **A**.`compareTo(`**B**`)` will return:

```
a value <   0   if A comes "before" B in the ordering,
a value >   0   if A comes "after" B in the ordering,
or               0   if A and B are considered "equal" in
the ordering
```

## compareTo:

- `compareTo` can be used as a test in an `if` statement.

| Primitives | Objects |
|---|---|
| `if (a < b) { ...` | `if (a.compareTo(b) < 0) { ...` |
| `if (a <= b) { ...` | `if (a.compareTo(b) <= 0) { ...` |
| `if (a == b) { ...` | `if (a.compareTo(b) == 0) { ...` |
| `if (a != b) { ...` | `if (a.compareTo(b) != 0) { ...` |
| `if (a >= b) { ...` | `if (a.compareTo(b) >= 0) { ...` |
| `if (a > b) { ...` | `if (a.compareTo(b) > 0) { ...` |

- Example:

```
String a = "alice";
String b = "bob";
if (a.compareTo(b) < 0) {      // true
        ...
}
```

## compareTo and collections:

- You can use an array or list of strings with Java's included binary search method because it calls compareTo internally.

  Example:

```
String[] a = {"al", "bob", "cari", "dan", "mike"};
int index = Arrays.binarySearch(a, "dan"); // 3
```

- Java's TreeSet / Map use compareTo internally for ordering.

  Example:

```
Set<String> set = new TreeSet<String>();
for (String s: a) {
     set.add(s);
 }
 System.out.println(s);
 // [al, bob, cari, dan, mike]
```

## Ordering our own types:

- We cannot binary search or make a TreeSet / Map of arbitrary types, because Java doesn't know how to order the elements.
  - The program compiles but crashes when we run it.

  Example:

```
Set<HtmlTag> tags = new TreeSet<HtmlTag>();
tags.add(new HtmlTag("body", true));
tags.add(new HtmlTag("b", false));
...
```

```
Exception in thread "main" java.lang.ClassCastException
at java.util.TreeSet.add(TreeSet.java:238
```

## Comparable:

```
public interface Comparable<E> {
    public int compareTo(E other);
}
```

- A class can implement the `Comparable` interface to define a natural ordering function for its objects.
- A call to your `compareTo` method should return:

```
a value <  0  if this object comes "before" the other
object,
a value >  0  if this object comes "before" the other
object,
Or 0          if this object is considered "equal" to the
other
```

## Comparable example:

```
public class Point implements Comparable<Point> {
     private int x;
      private int y;
...
     // sort by x and break ties by y
     public int compareTo(Point other) {
          if (x < other.x) {
               return -1;
          } else if (x > other.x) { return 1;
          } else if (y < other.y) {
          return -1;     // same x, smaller y
          } else if (y > other.y) {
          return 1; // same x, larger y
          } else {
          return 0; // same x and same y
          }
     }
```

## Collections class:

| Method name | Description |
|---|---|
| binarySearch(**list, value**) | returns the index of the given value in a sorted list (< 0 if not found) |
| copy(**listTo, listFrom**) | copies **listFrom**'s elements to **listTo** |
| emptyList(), emptyMap(), emptySet() | returns a read-only collection of the given type that has no elements |
| fill(**list, value**) | sets every element in the list to have the given value |
| max(**collection**),<br>min(**collection**) | returns largest/smallest element |
| replaceAll(**list, old, new**) | replaces an element value with another |
| reverse(**list**) | reverses the order of a list's elements |
| shuffle(**list**) | arranges elements into a random order |
| sort(**list**) | arranges elements into ascending order |

## Sorting methods in Java:

- The Arrays and Collections classes in java.util have a static method sort that sorts the elements of an array / list

Example

```
// sorting array elements
String[] words = {"foo", "bar", "baz", "ball"};
Arrays.sort(words);
System.out.println(Arrays.toString(words));
// [ball, bar, baz, foo]


List<String> words2 = new
ArrayList<String>();
for (String word: words) {
        words2.add(word);
}
Collections.sort(words2);
System.out.println(words2);
// [ball, bar, baz, foo]
```