Object-Oriented Programming

Collections

Dr. Seán Russell
sean.russell@ucd.ie

School of Computer Science,
University College Dublin

November 26, 2018

Learning outcomes

After this lecture and the related practical students should...

- understand the functionality of the Java Collections Framework
- be able to use data structures defined in collections

Table of Contents

Arrays

- Arrays are useful in many situations, but the main problem is that they have a fixed size
- If we do not know how many objects we want to store, it can be difficult to plan our programs using arrays
- Fortunately, dynamic data structures exist that can grow as we add more items
- You have studied some of these in Data Structures and Algorithms 1 and you will study more in the second part of the course
- Data Structures like, Lists, Stacks, Queues, Maps, Trees, Graphs and others

Creating Dynamic Data Structures

- These data structures are usually easier to use than an array and often more efficient
- The problem is that they are often complicated and difficult to implement
- Fortunately, Java provides most of these data structures in a framework called Collections
- The Java Collections framework is stored in the package `java.util`
- Additionally, all of the classes and interfaces are generic
- Meaning that we can store any type of data and not have to worry about typecasting

What is in Collections?

1. A set of interfaces
   - These tell us the operations available for the different types of data structures
   - For example there are interfaces for `List`, `Map`, `Queue`, `Deque` and `Set`
2. A set of implementation classes
   - For each Interface there may be many implementation classes
   - For example, `List` is implemented by the `ArrayList`, `LinkedList` and `Vector` classes
3. A set of Algorithms
   - There are implementations for sorting, searching and even randomising data

Table of Contents

List

- The list interface is a little different than the list that you learned about in Data Structures and Algorithms
- The methods in the interface are based on indexes rather than on positions
- This makes it less efficient in some situations than you learned
- The List has one type parameter (E) for the type that will be stored in the list

Adding to a List

- The list has many methods for adding data
- `public boolean add(E e)`
- `public void add(int index, E e)`
- `public boolean addAll(Collection<E> c)`
- `public boolean addAll(int index, Collection<E> c)`
- `public E set(int index, E e)`

Getting data from a List

- There is one main way of getting data from a list
- `public E get(int index)`
- The `List` interface also implements `Iterable`
- `public Iterator<E> iterator()`
- There are also methods for removing
- `public boolean remove(Object e)`
- `public E remove(int index)`
- A method for finding the index of an object
- `public int indexOf(Object o)`

List Utility Methods

- There are many utility functions in the List interface
- `public void clear()`
- `public int size()`
- `public void sort(Comparator<E> c)`
- `public boolean contains(Object e)`
- `public boolean isEmpty()`

List Example

```
1  List<String> strings = new ArrayList<String>();
2  strings.add("VIVEK");
3  strings.add("Sean");
4  strings.add("LINA");
5  strings.add("Anca");
6  strings.add("ABEY");
7  strings.sort(String.CASE_INSENSITIVE_ORDER);
8  Iterator<String> ns = strings.iterator();
9  while(ns.hasNext()) {
10   System.out.println(ns.next());
11 }
12 System.out.println();
13 strings.remove(1);
14 ns = strings.iterator();
15 while(ns.hasNext()) {
16   System.out.println(ns.next());
17 }
```

Table of Contents

Map

- The Map is an interface for an associative store
- This is useful where we will be storing objects that can be identified by some unique number (like student number or some other id)
- There are different implementations such as the `HashMap`, `TreeMap` and `EnumMap`
- The Map interface has two type parameters, one for the type of the Key (K) and one for the Type of the Value (V)

Adding Data to a Map

- There are three methods to insert data into a map
- `public V put(K key, V value)`
- `public void putAll(Map<K, V> m)`
- `public void putIfAbsent(K key, V value)`
- `public V replace(K key, V value)`

Getting Data from the Map

- There are multiple ways of getting data from a Map
- `public V get(Object key)`
- `public V getOrDefault(Object key, V defaultValue)`
- `public Collection<V> values()`
- `public Set<K> keySet()`
- `public Set<Map.Entry<K,V>> entrySet()`

Map Utility Methods

- There are many utility functions in the Map interface
- `public void clear()`
- `public int size()`
- `public boolean containsKey(Object key)`
- `public boolean containsValue(Object value)`
- `public boolean isEmpty()`

Map Example

```
1  Map<String, String> names = new
       HashMap<String, String>();
2  names.put("Vivek","Vivek Nallur");
3  names.put("Sean","Sean Russell");
4  names.put("Lina", "Lina Xu");
5  names.put("Abey", "Abraham Campbell");
6  names.put("Sean", "Dave Lillis");
7
8  System.out.println(names.get("Sean")+"\n\n");
9
10 Iterator<String> ns =
       names.values().iterator();
11 while(ns.hasNext()) {
12   System.out.println(ns.next());
13 }
```

Table of Contents

Algorithms

- There are algorithms provided by two different classes in Collections
- The first is `Arrays` and the second is `Collections`
- Arrays contains a number of static methods that are useful for dealing with arrays
- Collections contains a number of static methods that are useful for dealing with Lists and other Collections

Arrays

- Arrays provides a number of methods for sorting data
- There is one method for each primitive number type e.g. int, double, char...
- There is also a method for objects
- Arrays also provides methods for searching
- Data must be sorted first
- Arrays also provides a toString method that can be used to easily print arrays

Sorting Numbers

- `public static void sort(int[] a)`
- This will sort an array of integers into the correct order
- There is one method for each of the primitive number types

```
1 int[] arr = new int[]
    {6,7,4,3,2,3,2,3,4,23,5};
2 Arrays.sort(arr);
3 System.out.println(Arrays.toString(arr));
```

Sorting Objects

- Arrays also provides a method for sorting objects, but the objects must implement the Comparable interface

```
1  public class Student implements
       Comparable<Student> {
2    String name;
3    public Student(String n) {
4      name = n;
5    }
6
7    public int compareTo(Student o) {
8      return name.compareTo(o.name);
9    }
10 }
```

Sorting Objects

- Once we have objects that are Comparable, we can sort them

```
1 Student[] students = new Student[] {new
    Student("Sean"), new Student("Abey"), new
    Student("Vivek"), new Student("Anca")};
2 System.out.println(Arrays.toString(students));
3 Arrays.sort(students);
4 System.out.println(Arrays.toString(students));
```

Searching Arrays

- Arrays provides an implementation of binary search for each of the number types
- The data must be sorted first
- The method will tell us which index it first finds the number in

Searching Arrays

```
1 int[] arr = new int[] { 6, 7, 4, 3, 2, 3, 2,
     3, 4, 23, 5 };
2 Arrays.sort(arr);
3 System.out.println(Arrays.toString(arr));
4
5 int a = Arrays.binarySearch(arr, 23);
6 System.out.println(a);
```

Sorting Lists

- Collections provides a two methods to sort Lists
- One of the methods should be used when the objects in the list implements the Comparable interface
- `public static <T extends Comparable<T>> void sort(List<T> list)`
- For the other method, we must provide a Comparator to compare the objects
- `public static <T> void sort(List<T> list, Comparator<T> c)`

Sorting Comparable

```
1 List<Student> list = new
    ArrayList<Student>();
2 list.add(new Student("Sean"));
3 list.add(new Student("Abey"));
4 list.add(new Student("Vivek"));
5 list.add(new Student("Anca"));
6 Collections.sort(list);
7 System.out.println(list);
```

Sorting with Comparator 1

```
1  import java.util.Comparator;
2
3  public class Module {
4    private String code;
5    public Module(String c) {code = c;}
6    public String toString() {return code;}
7  }
8  class ModuleComp implements
       Comparator<Module>{
9    public int compare(Module m1, Module m2) {
10     return
       m1.toString().compareTo(m2.toString());
11   }
12 }
```

Sorting with Comparator 2

```
1 List<Module> mods = new ArrayList<Module>();
2 mods.add(new Module("OOP"));
3 mods.add(new Module("Prog 1"));
4 mods.add(new Module("OOD"));
5 Collections.sort(mods, new ModuleComp());
6 System.out.println(mods);
```