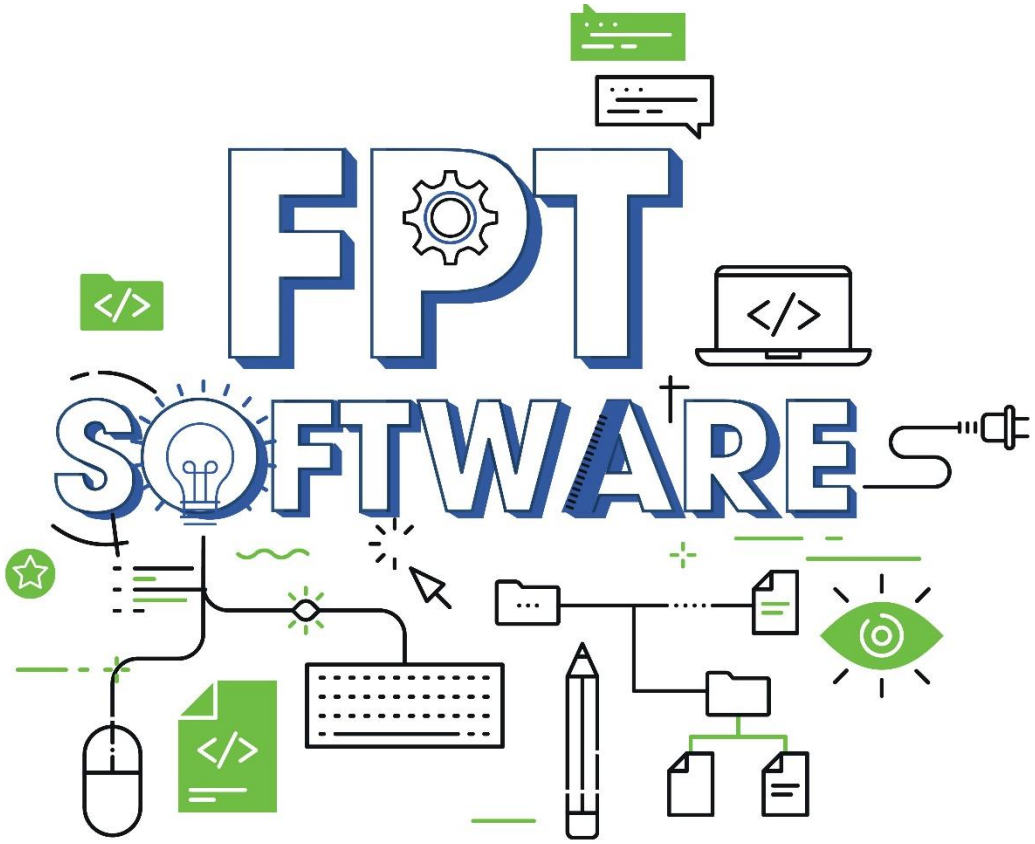
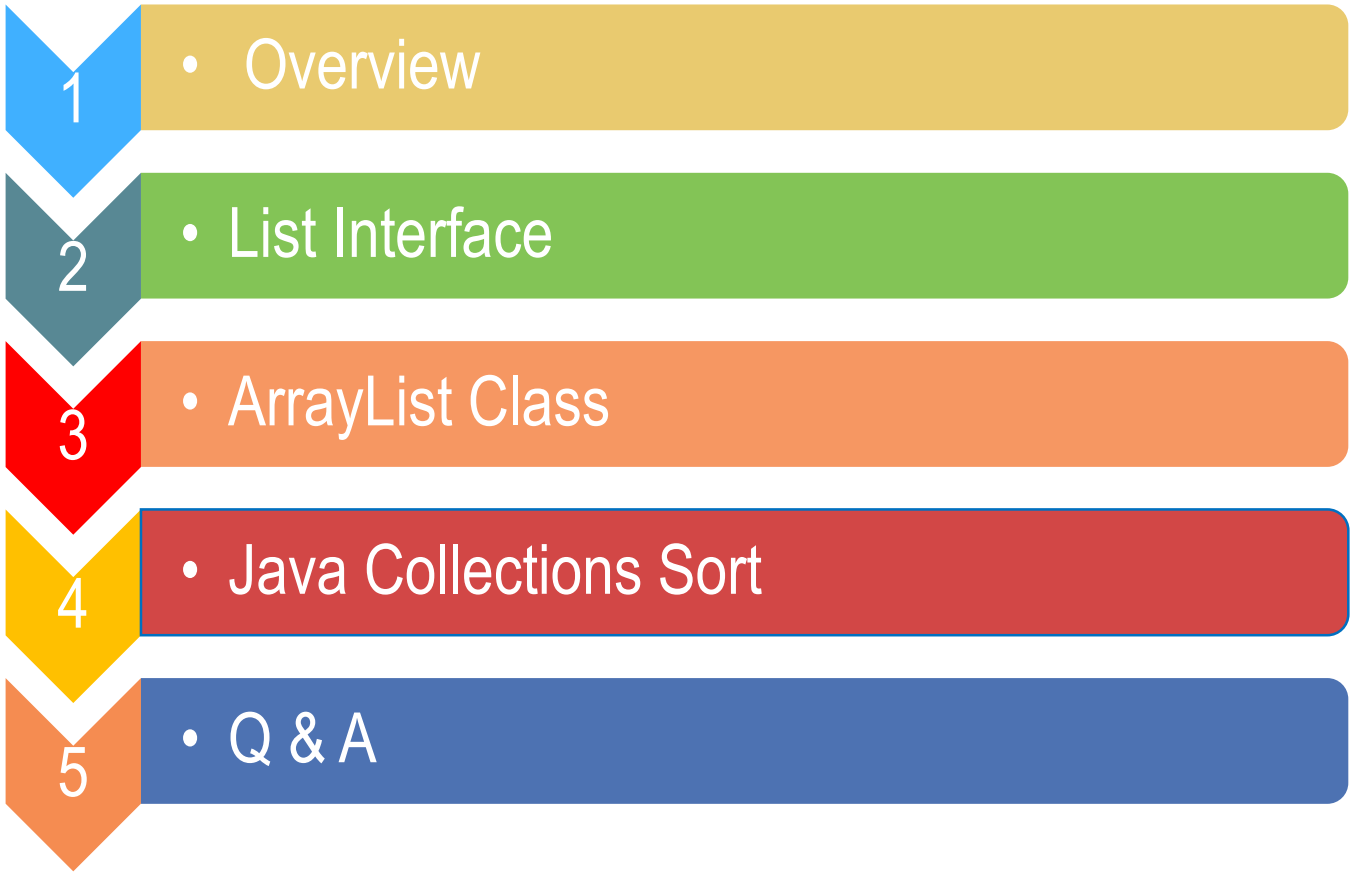
JAVA

COLLECTIONS FRAMEWORK

List Interface



Agenda


1
• Overview
2
• List Interface
3
• ArrayList Class
4
• Java Collections Sort
5
• Q & A

Lesson Objectives

- ***Understand*** *the hierarchy in the Collections Framework of Java.*
- ***Understand*** *the difference between List and other collection types.*
- ***Perform*** *common operations such as adding, removing, and accessing elements in an ArrayList.*
- ***Explore*** *sorting techniques for ArrayList elements.*
- ***Discuss*** *the use of Comparator and Comparable interfaces for custom sorting.*

Section 1

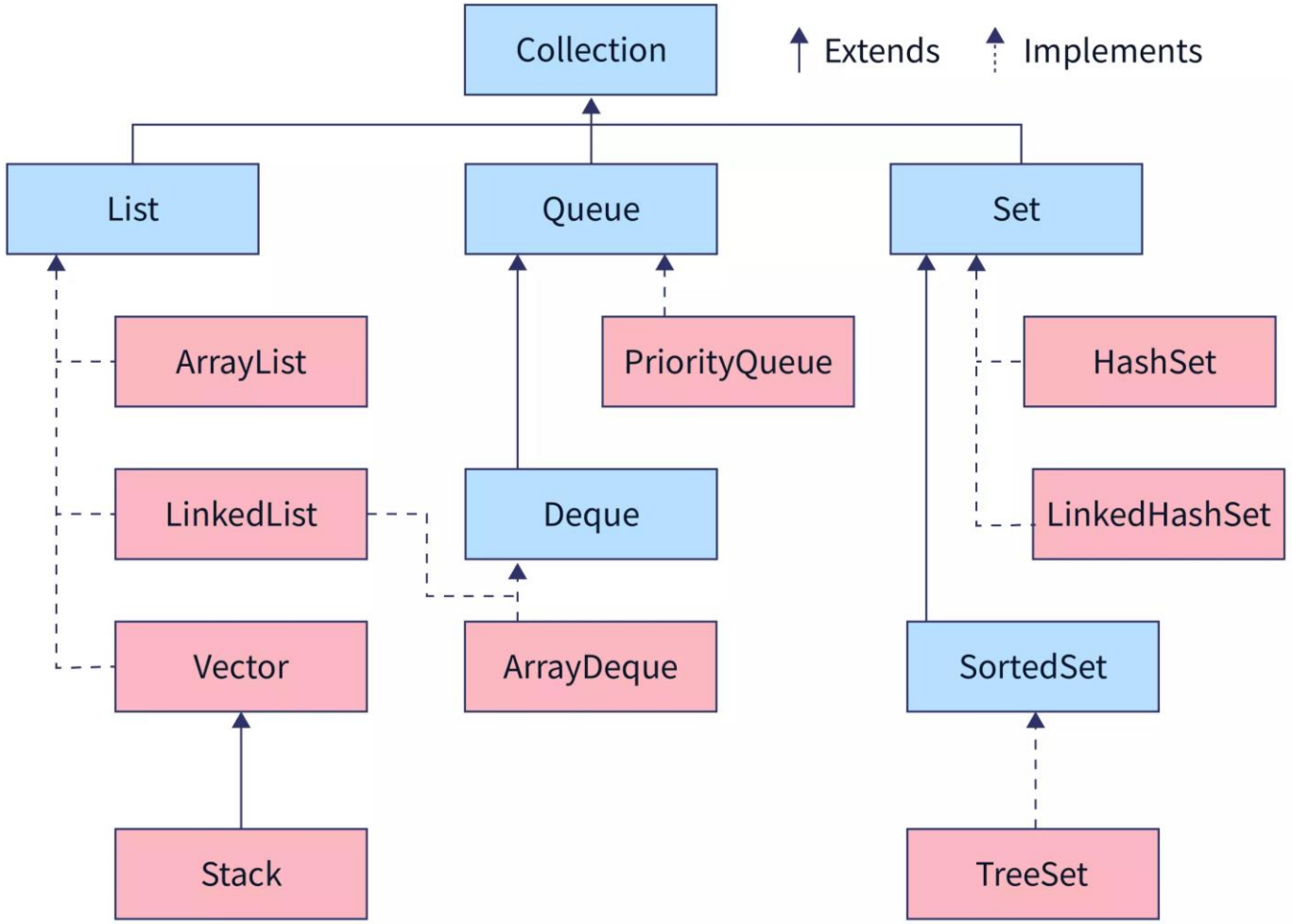
Overview

Overview

Collections in Java is a framework that **stores and manipulates a group of objects**.

- Collections framework is a **hierarchy of interfaces and classes** that provides easy management of a group of objects:
  - ✓Interfaces: **List, Queue, Deque, Set**
  - ✓Classes: **ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet**.

Hierarchy of Collections Framework



What is a Collection in Java?

A **Collection** in **Java is an object** which represents a **group of objects**, known as its elements.

- **Java Collection vs Collections Framework**

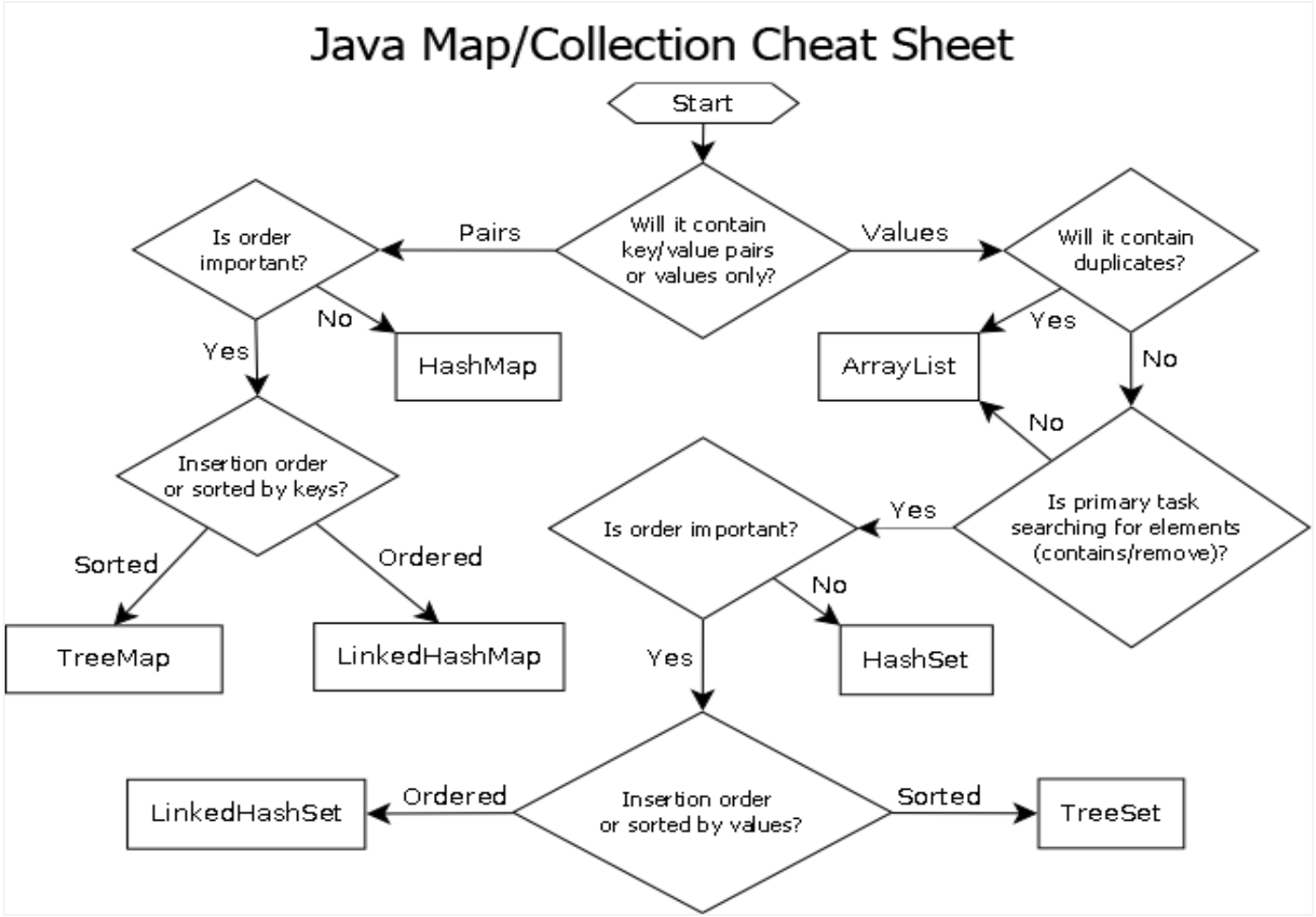
Collection in Java	Collections Framework
Collection in Java is a class.	Collections Framework is a framework.
It is a single unit that contains and **manipulates** a group of objects.	They are used to **manipulate collections**.

Methods of Collection interface

- **add(E e)**: It is used to insert an element in this collection.
- **addAll(Collection<? extends E> c)**: It is used to insert the specified collection elements in the invoking collection.
- **remove(Object element)**: It is used to delete an element from the collection.
- **size():** It returns the total number of elements in the collection.
- **clear():** It removes the total number of elements from the collection.
- **contains(Object element):** It is used to search an element.
- **toArray():** It converts collection into array.
- **isEmpty():** It checks if collection is empty.
- **stream():** It returns a sequential Stream with the collection as its source.

Java Collection Cheat Sheet

- **Collection Interfaces:**

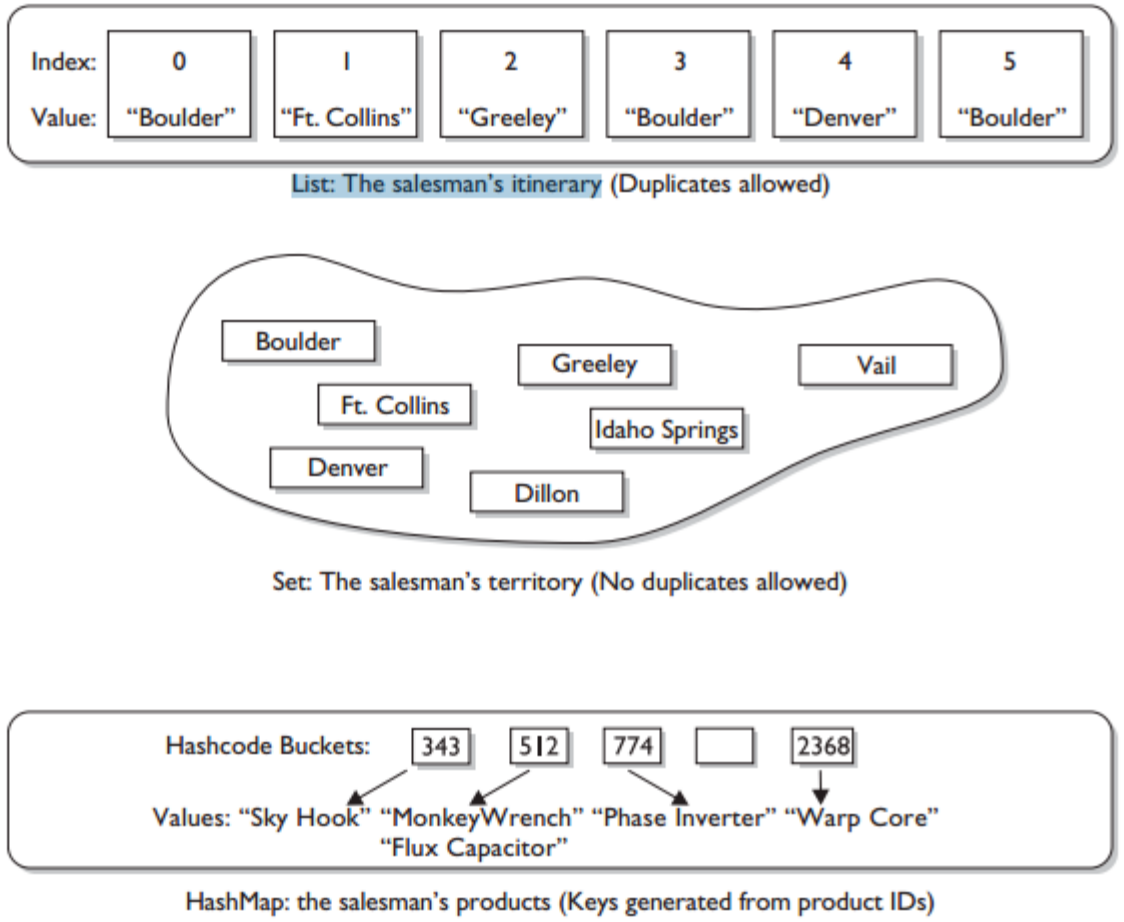


Java Collection

Figure 7-3 illustrates the structure of a List, a Set, and a Map.

FIGURE 7-3

The structure of a List, a Set, and a Map



Section 2

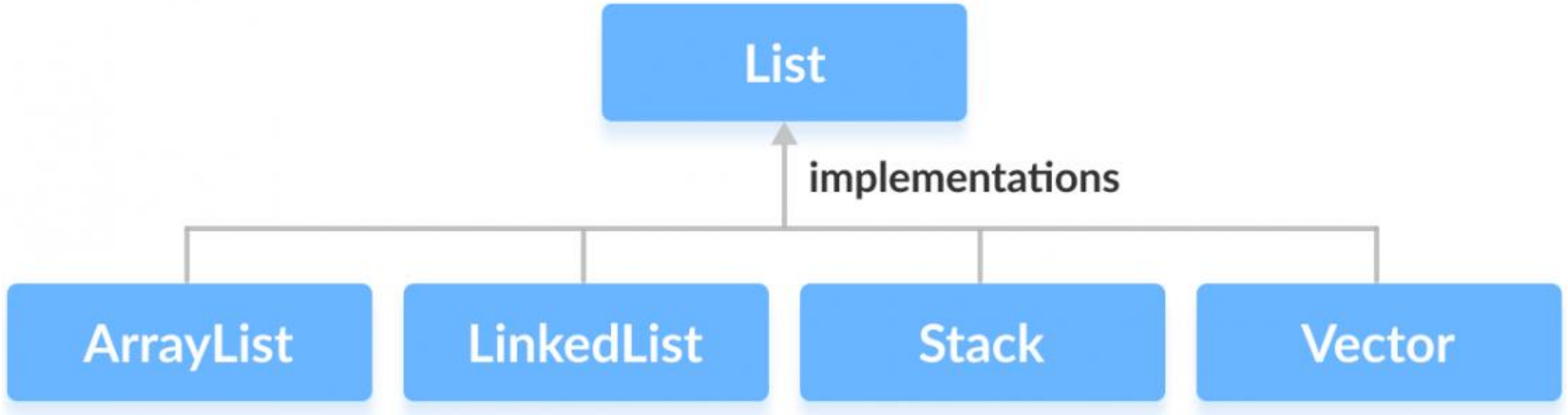
List Interface

List Interface

The **List** interface is an **ordered collection** that allows us to **store** and **access** elements sequentially. It extends the **Collection** interface.

- Since **List is an interface**, we cannot create objects from it.
- We can use these classes:
  - ✓ArrayList
  - ✓LinkedList
  - ✓Vector
  - ✓Stack

List Interface



- How to use List?

```
// ArrayList implementation of List
List<String> list1 = new ArrayList<>();
// LinkedList implementation of List
List<String> list2 = new LinkedList<>();
```

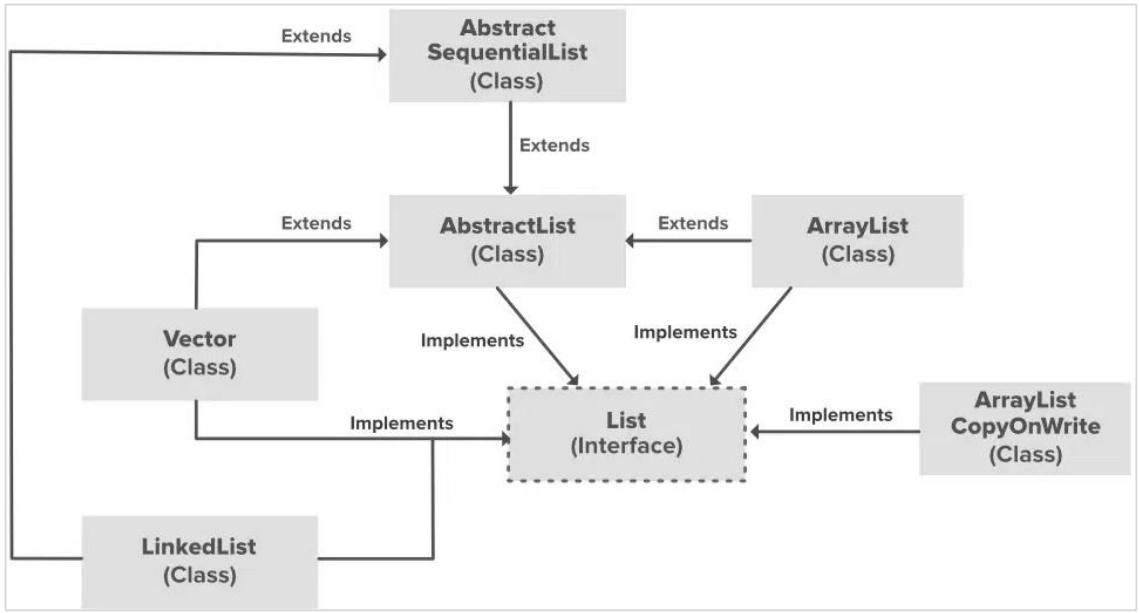
Methods of List interface

- **add()** - adds an element to a list
- **addAll()** - adds all elements of one list to another
- **get()** - helps to randomly access elements from lists
- **iterator()** - returns iterator object that can be used to sequentially access elements of lists
- **set()** - changes elements of lists
- **remove()** - removes an element from the list
- **removeAll()** - removes all the elements from the list
- **clear()** - removes all the elements from the list (more efficient than removeAll())
- **size()** - returns the length of lists
- **toArray()** - converts a list into an array
- **contains()** - returns true if a list contains specified element

ArrayList

- **ArrayList** supports dynamic arrays that can grow as needed.
  - ✓ Array lists are created with an initial size.
  - ✓ When this size is exceeded, the collection is automatically enlarged.
  - ✓ When objects are removed, the array may be shrunk
- **Syntax:**

```
List<DataType> arrName = new ArrayList<>();
```



ArrayList

- **ArrayList** is implemented as a resizable array. The important points about Java ArrayList class are:
  - ✓ Java ArrayList class can contain duplicate elements.
  - ✓ Java ArrayList class maintains insertion order.
  - ✓ Java ArrayList class is non synchronized.
  - ✓ Java ArrayList allows random access because array works at the index basis.
  - ✓ In ArrayList, manipulation is little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.
- **ArrayList class declaration:**

```
public class ArrayList<E> extends AbstractList<E>
                        implements List<E>, RandomAccess, Cloneable, Serializable
```

Main methods of ArrayList

Constructor	Description
ArrayList()	It is used to build an empty array list.
ArrayList(Collection<? extends E> c)	It is used to build an array list that is initialized with the elements of the collection c.
ArrayList(int capacity)	It is used to build an array list that has the specified initial capacity.
Method	**Description**
void add(int index, E element)	It is used to insert the specified element at the specified position in a list.
boolean add(E e)	It is used to append the specified element at the end of a list.
boolean addAll(Collection<? extends E> c)	It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
E get(int index)	It is used to fetch the element from the particular position of the list.
boolean isEmpty()	It returns true if the list is empty, otherwise false.
boolean contains(Object o)	It returns true if the list contains the specified element

Main methods of ArrayList

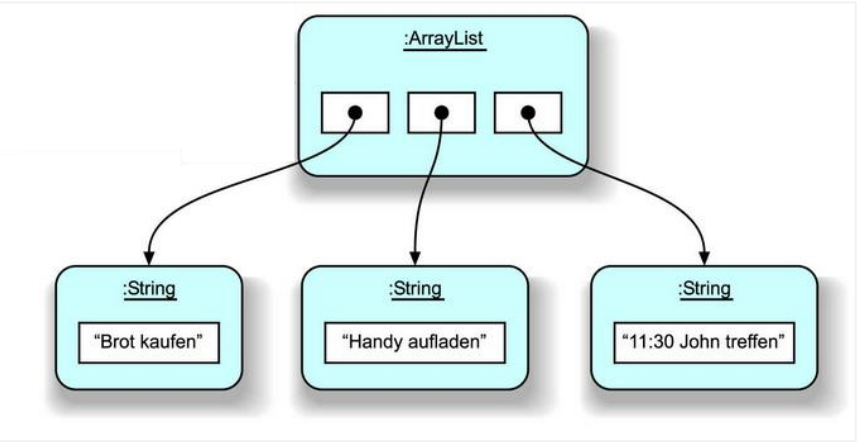
Method	Description
int indexOf(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
E remove(int index)	It is used to remove the element present at the specified position in the list.
boolean remove(Object o)	It is used to remove the first occurrence of the specified element.
boolean removeAll(Collection<?> c)	It is used to remove all the elements from the list.
boolean removeIf(Predicate<? super E> filter)	It is used to remove all the elements from the list that satisfies the given predicate.
protected void removeRange(int fromIndex, int toIndex)	It is used to remove all the elements lies within the given range.
void retainAll(Collection<?> c)	It is used to retain all the elements in the list that are present in the specified collection.
ist<E> subList(int fromIndex, int toIndex)	It is used to fetch all the elements lies within the given range.
int size()	It is used to return the number of elements present in the list.

ArrayList: Adding Elements

- **add(Object):** This method is used to add an element at the end of the ArrayList.
- **add(int index, Object):** This method is used to add an element at a specific index in the ArrayList.

```
public class ListExample {
   public static void main(String[] args) {
      // Creating an ArrayList of string type
      ArrayList<String> al = new ArrayList<>();
      // Adding elements to ArrayList Custom inputs
      al.add("Add");
      al.add("elements");
      al.add("an");
      al.add("ArrayList");
      // Here we are mentioning the index at which it is to be added
      al.add(2, "to");
      // Printing all the elements in an ArrayList
      System.out.println(al);

      }
   }
}
```



Output

```
[Add, elements, to, an, ArrayList]
```

ArrayList: Changing Elements

```java
public class ArrayListExample {
   // Main driver method
   public static void main(String args[]) {
      // Creating an ArrayList of string type
      ArrayList<String> al = new ArrayList<>();
      // Adding elements to ArrayList Custom inputs
      al.add("ArrayList");
      al.add("ArrayList");
      // Here we are mentioning the index at which it is to be added
      al.add(1, "in");
      // Printing all the elements in an ArrayList
      System.out.println(al);
      // Setting element at 1st index
      al.set(2, "Java");
      // Printing all the elements in an ArrayList
      System.out.println(al);
   }
}
```

- **Output:**

```
[ArrayList, in, ArrayList]
[ArrayList, in, Java]
```

ArrayList: Ways to iterate the elements

- There are **various ways to traverse** the collection elements:
  - ✓ By for loop.
  - ✓ By for-each loop.
  - ✓ By forEach() method.
  - ✓ By Iterator interface.
  - ✓ By ListIterator interface.
  - ✓ By forEachRemaining() method.

ArrayList: Get and Removing Elements

- **Can use ArrayList to store String, Number:**

ArrayListOfString.java

```
13 public class ArrayListOfString {
14     public static void main(String[] args) {
15
16         ArrayList<String> listOfNames = new ArrayList<String>();
17
18         listOfNames.add("Wartian Herkku");
19         listOfNames.add("Wellington Importadora");
20         listOfNames.add("White Clover Markets");
21         listOfNames.add("Wilman Kala");
22         listOfNames.add("Wolski");
23
24         System.out.println("Value at the first index: " + listOfNames.get(0));
25
26         // Use for loop to look up arraylist
27         System.out.println("Before array :");
28         int arrSize = listOfNames.size();
29         for (int i = 0; i < arrSize; i++) {
30             System.out.println(i + ": " + listOfNames.get(i));
31         }
32
33         listOfNames.remove("Wilman Kala");
34         listOfNames.remove(1);
35         listOfNames.add(2, "Matti Karttunen");
36
37         System.out.println("After array (Unsorted List):");
38
39         arrSize = listOfNames.size();
40         for (int i = 0; i < arrSize; i++) {
41             System.out.println(i + ": " + listOfNames.get(i));
42         }
```

Instance of ArrayList

Add value into ArrayList

Get value from ArrayList

Using for loop to lookup value

Remove by Value

Remove by Index

Add value by Index

ArrayList Example

- Iterating the elements using **for loop**.

```
43          /* Sort statement */
44          Collections.sort(listOfNames);
45
46          System.out.println("After Sorting:");
47
48          arrSize = listOfNames.size();
49          for (int i = 0; i < arrSize; i++) {
50              System.out.println(i + ": " + listOfNames.get(i));
51          }
52      }
53  }
```


```
Value at the first index: Wartian Herkku
Before array :
0: Wartian Herkku
1: Wellington Importadora
2: White Clover Markets
3: Wilman Kala
4: Wolski
After array (Unsorted List):
0: Wartian Herkku
1: White Clover Markets
2: Matti Karttunen
3: Wolski
After Sorting:
0: Matti Karttunen
1: Wartian Herkku
2: White Clover Markets
3: Wolski
```

ArrayList with Object

- Create an **Animal** class:

```
public class Animal {
  private String name;
  private float weight;

  public Animal() {
  }

  public Animal(String name, float weight) {
    super();
    this.name = name;
    this.weight = weight;
  }

  public String getName() {
    return name;
  }

public void setName(String name) {
    this.name = name;
  }

  public float getWeight() {
    return weight;
  }

  public void setWeight(float weight) {
    this.weight = weight;
  };

  // toString() method
}
```

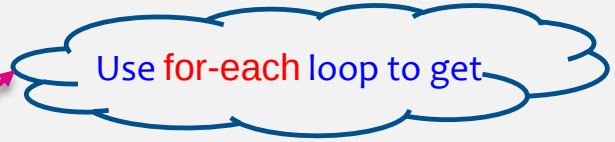
ArrayList with Object

- Adding elements to ArrayList
- Iterating the elements using for-each loop

```
public class ArrayListOfObject {
    public static void main(String[] args) {
      ArrayList<Animal> listOfAnimal = new ArrayList<>();

      listOfAnimal.add(new Animal("Cat", 2.0f));
      listOfAnimal.add(new Animal("Dog", 8.0f));
      listOfAnimal.add(new Animal("Turtle", 1.2f));
      listOfAnimal.add(new Animal("Bear", 60.0f));
      listOfAnimal.add(new Animal("Rabbit", 1.6f));
      listOfAnimal.add(new Animal("Bird", 0.6f));

      for (Animal animal : listOfAnimal) {
         System.out.println(animal);
      }
    }
}
```



ArrayList with Object

- Iterating ArrayList using Iterator:

```
Iterator<Animal> itr = listOfAnimal.iterator();
while (itr.hasNext()) {
   System.out.println(itr.next());
}
```

- Iterating ArrayList using forEach() method.

```
listOfAnimal.forEach(animal -> System.out.println(animal));
```

- **Output:**

```
Animal [name=Cat, weight=2.0]
Animal [name=Dog, weight=8.0]
Animal [name=Turtle, weight=1.2]
Animal [name=Bear, weight=60.0]
Animal [name=Rabbit, weight=1.6]
Animal [name=Bird, weight=0.6]
```

Section 3

Java Collections Sort

Overview

Collections sort in Java provides **in-built methods** to sort data faster and in an easier manner. *Collections* sort is a method of *Java Collections* class used to sort a list, which implements the *List* interface.

- All the elements in the list must be mutually comparable
  - ✓ If a list consists of *string elements*, then it will be sorted in alphabetical order.
  - ✓ If it consists of a *date element*, it will be sorted into chronological order.

How does it happen? String and date both implement the Comparable interface in Java. Comparable implementations provide a natural ordering for a class, which allows the object of that class to be sorted properly.

***ClassCastException:** If the list contains elements that are not mutually comparable using the specified comparator, it will throw ClassCastException.*

Collections Sort Method in Java

- The **Collections.sort** has two overloaded methods :
  - ✓**public static void sort(List list):** Sort the list into ascending order, the natural ordering of its element.
  - ✓**public static void sort(List list, Comparator c):** Sort the list according to the specified comparator.

Collections Sort Method in Java

- **Example:**

```
public class Main {
  public static void main(String[] args) {
    List<String> studentName = Arrays.asList("Tom","John","Harry","Philip","Max");
    Collections.sort(studentName);

    for(String name : studentName) {
       System.out.println(name);
    }
  }
}
```

- **Output:**

```
Harry
John
Max
Philip
Tom
```

For objects to have a natural order they must implement the interface java.lang.Comparable. The Comparable interface has a method compareTo():

- ✓ ***If both the objects are equal, returns 0***
- ✓ ***If the first object is greater than the second, returns a value > 0***
- ✓ ***If the second object is greater than the first, returns a value < 0***

Collections Sort Method in Java

- What if we have a list of **custom objects**: The class must implement a Comparable interface.
- **Example:**

```
public class Student implements Comparable<Student> {
   private Integer rollno;
   private String name;
   private String address;
   private Double gpa;
   // Constructor
   public Student(Integer rollno, String name, String address, Double gpa) {
     this.rollno = rollno;
     this.name = name;
     this.address = address;
     this.gpa = gpa;
   }

   // getter and setter methods
   // Used to print student details in main()
   public String toString() {
     return this.rollno + " " + this.name + " " + this.address + " "+ this.gpa;
   }

   @Override
   public int compareTo(Student student) {
     return this.name.compareTo(student.name);
   }
}
```

Collections Sort Method in Java

- **Example:**

```
public class StudentManagement {
  public static void main(String[] args) {

      List<Student> students = new ArrayList<>();

      Student student1 = new Student(12345, "Maria Anders", "Berlin", 8.0);
      Student student2 = new Student(23123, "Ana Trujillo", "México D.F.", 6.6);
      Student student3 = new Student(22334, "Antonio Moreno", "Berlin", 7.7);
      Student student4 = new Student(74231, "Thomas Hardy", "London", 9.2);
      Student student5 = new Student(89346, "Christina Berglund", "London", 8.5);

      students.add(student1);
      students.add(student2);
      students.add(student3);
      students.add(student4);
      students.add(student5);

      Collections.sort(students);

      students.forEach(s -> System.out.println(s));
  }
}
```

Output:

```
23123 Ana Trujillo México D.F.   6.6
22334 Antonio Moreno Berlin      7.7
89346 Christina Berglund London  8.5
12345 Maria Anders Berlin        8.0
74231 Thomas Hardy London        9.2
```

Java Collections sort(List list, Comparator c)

- We can implement the `java.util.Comparator` interface and pass an instance of it as the second argument of **sort().**
- Let's consider that we want to define the ordering based on the "*gpa*" field of the **Student**. We implement the `Comparator`, and in its `compare()` method, we need to write the logic for comparison:

```
public class SortByGpa implements Comparator<Student> {
   @Override
   public int compare(Student o1, Student o2) {
       return o1.getGpa().compareTo(o2.getGpa());
   }
}
```

- Now, we can sort it using this comparator:

```
Collections.sort(students, new SortByGpa());
```

- **Output:**

```
23123 Ana Trujillo México D.F.      6.6
22334 Antonio Moreno Berlin         7.7
12345 Maria Anders Berlin           8.0
89346 Christina Berglund London     8.5
74231 Thomas Hardy London           9.2
```

Java 8 Lambda : Comparator

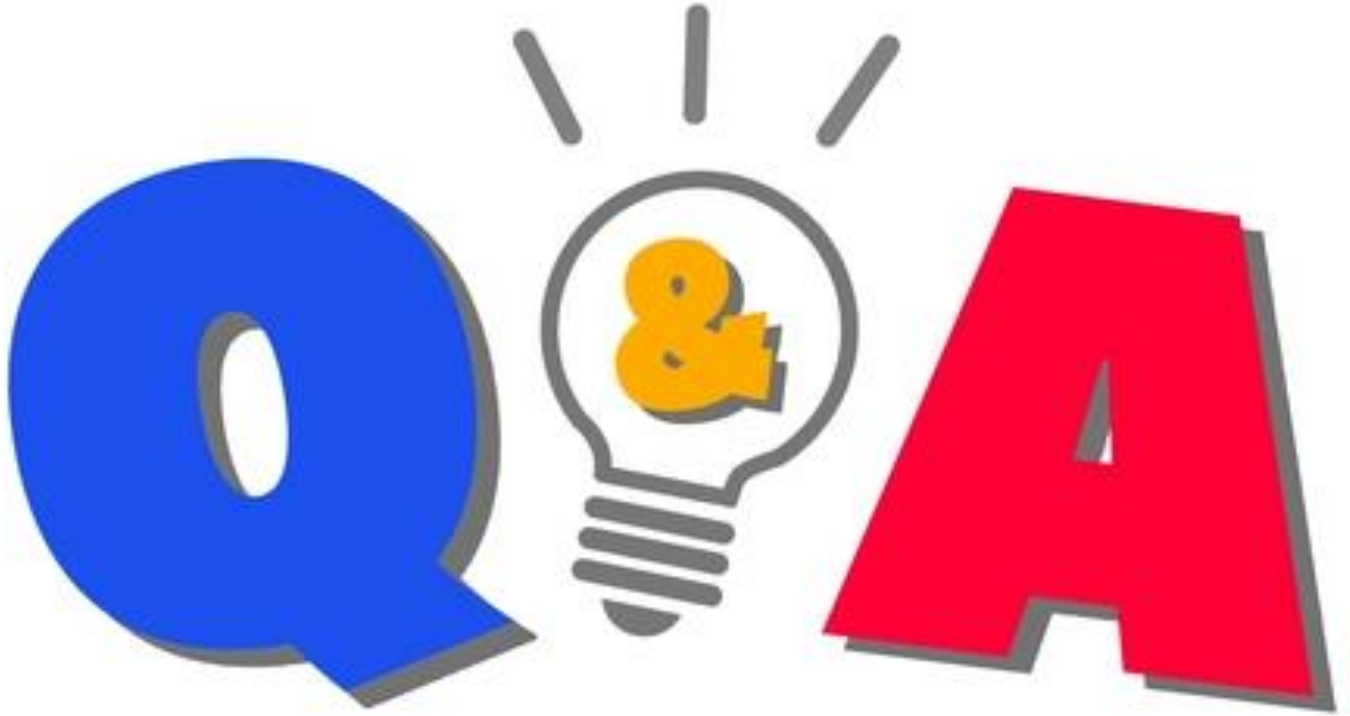
- Instead of writing new class for **Comparator**, using *lambda expression*, we can provide sorting logic at runtime as well:

```
Collections.sort(students, (s1,s2)->{
    return s1.getGpa().compareTo(s2.getGpa());
});
```

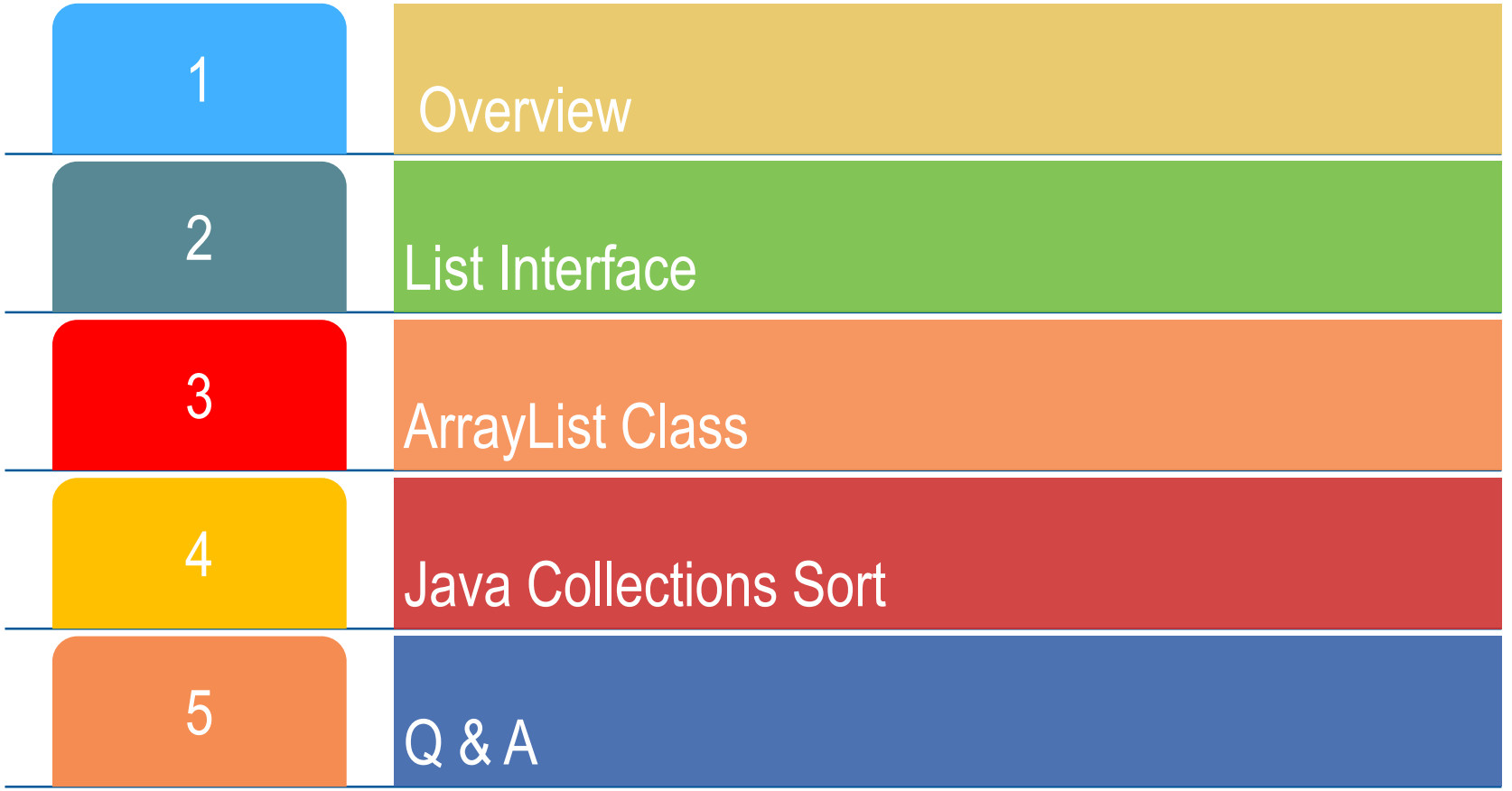
We will learn more about Lambda Expression and Function Interface in detail in the next session.

Comparator vs Comparable

- **The *Comparable* interface is a good choice to use for defining the default ordering,** or in other words, if it's the main way of comparing objects.
- So why use a *Comparator* if we already have *Comparable*? There are several reasons why:
  - ✓ Sometimes we can't modify the source code of the class whose objects we want to sort, thus making the use of *Comparable* impossible
  - ✓ Using *Comparators* allows us to avoid adding additional code to our domain classes
  - ✓ We can define multiple different comparison strategies, which isn't possible when using *Comparable*


Q
&
A

Lesson Summary



THANK YOU!