Féidearthachtaí as Cuimse
Infinite Possibilities

Java Collections Framework

Object Oriented programming

OLLSCOIL TEICNEOLAÍOCHTA BHAILE ÁTHA CLIATH
TECHNOLOGICAL UNIVERSITY DUBLIN

A constant requirement in programming:

- To be able to hold a collection of “data “ to be used by your class(es)

  - An application to do find a bus route for your journey – and display it as points on a map
  - An application to read text files to find the number of matching words.
  - An application to display a list of countries and allow you to click on one to find out tourism details etc et c

A constant requirement in programming:

- To be able to hold a collection of “data “ to be used by your class(es)
- Lists, sets, maps, queues….
- So far, just ArrayLists
- Your data/ application guides you to the right one
  - Order, duplicates, links, key values etc

What type of collection?

- A student roll system for attendance monitoring
- [student 1, student 2, 3 etc.….]

? Duplicates

? Ordered

What type of collection?

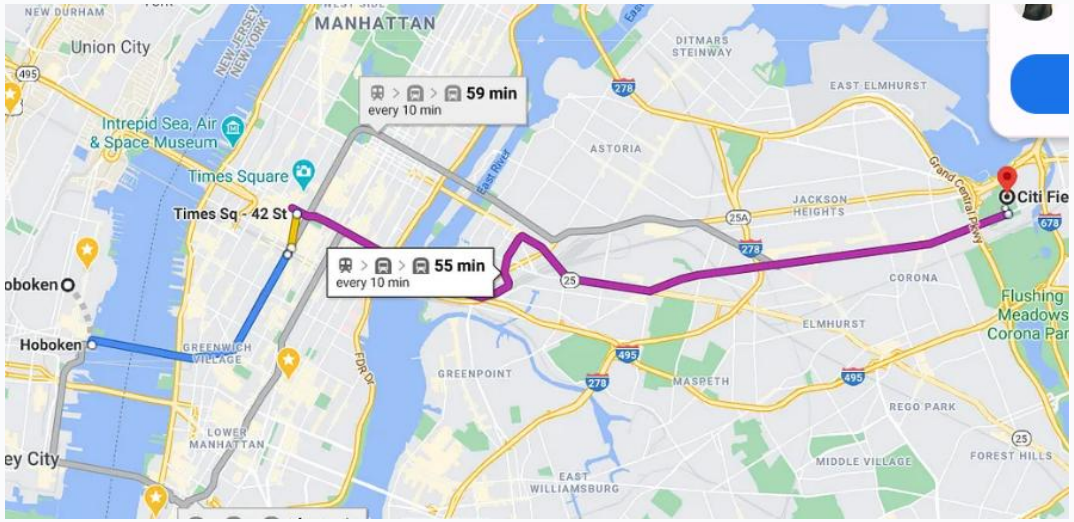
- An application to do find a bus route for your journey – and display it as bus stops on a map

- [stop 1, stop 2, stop 3….]

? Duplicates

? Ordered (mixing matters?)

? Anything else



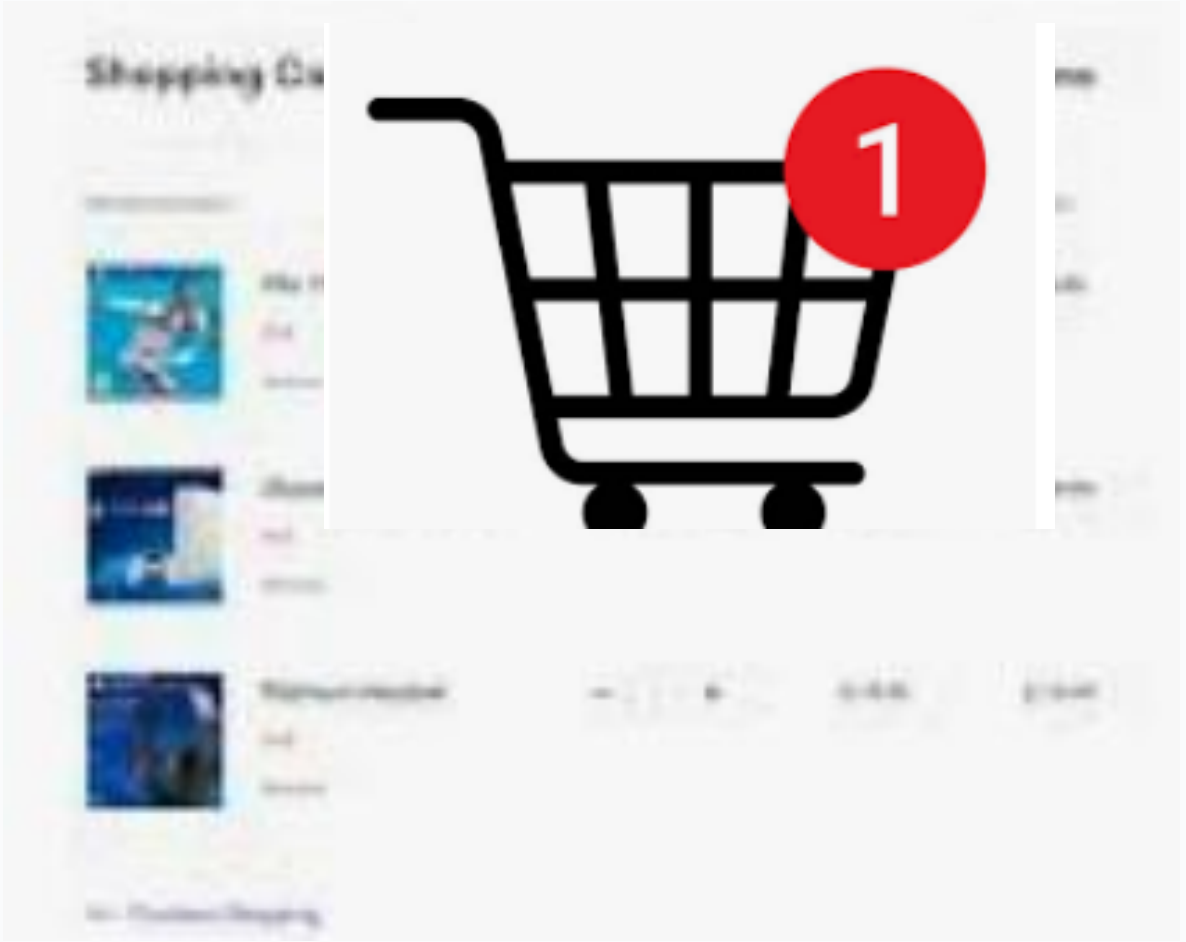
What type of collection?

- Shopping cart app
- Store items

? Duplicates

? Ordered

? Anything else



Using the Java Collections Framework

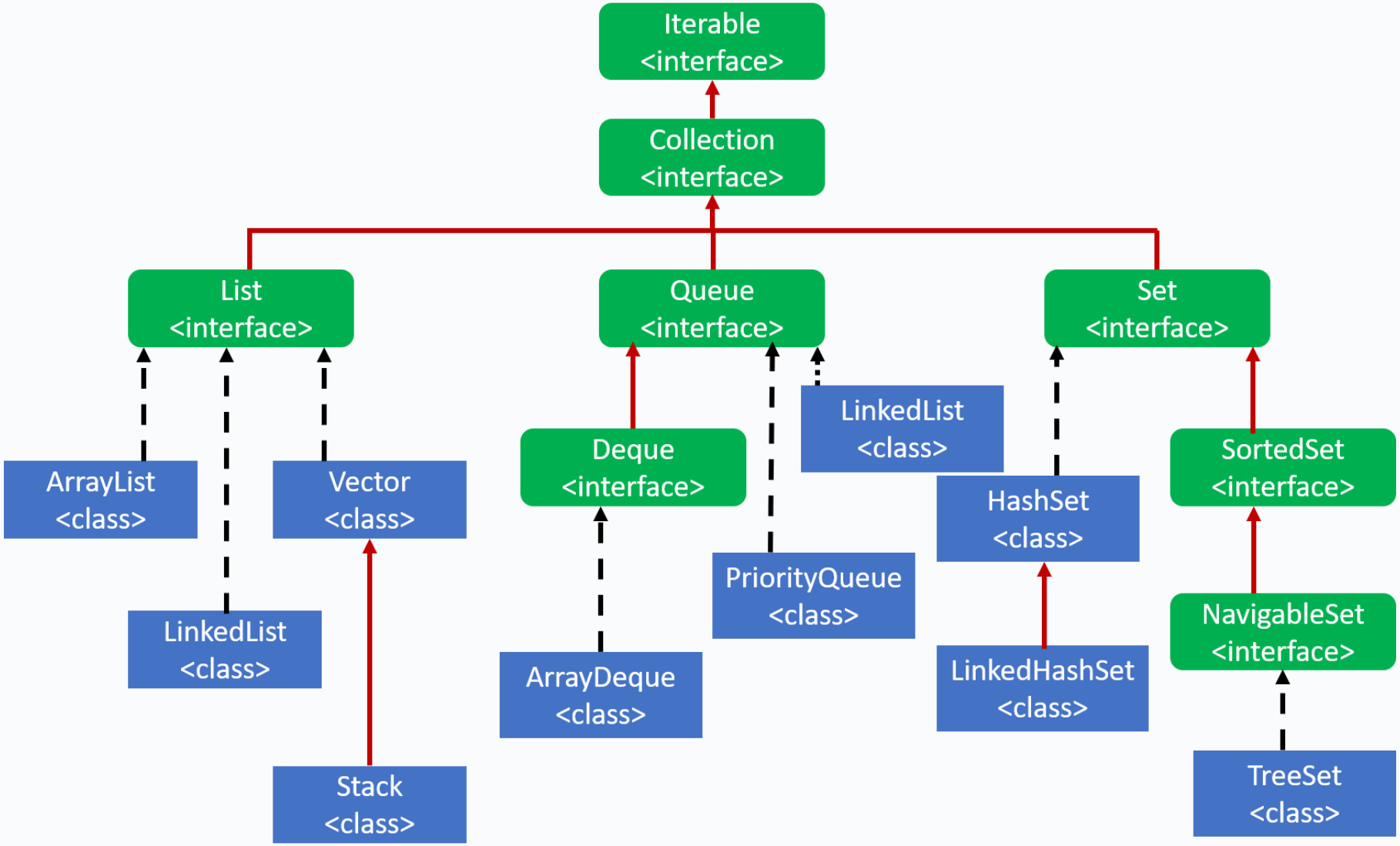
- **First** - Identifying the type of collection - important
  - List? Set? Map? Queue

- **Then – use the methods:** What type of things are done again and again on any of these collections?

Java Collections Framework

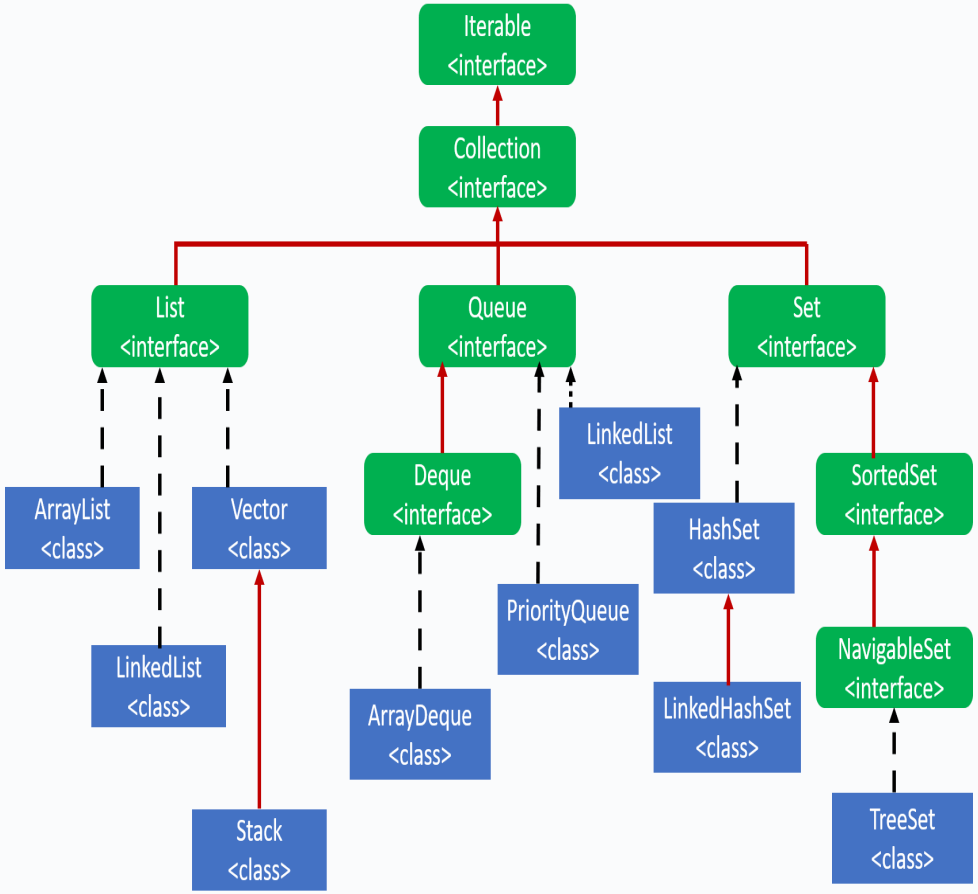
- **The Java Collections Framework** is a set of classes and interfaces implementing complex collection data structures.
- A **collection** represents a group of **object**s - The objects are known as its *elements*
- e.g. ..

Java Collections Framework (most of it)



Collections Intefaces

- The core collection interface act as blueprints for the behaviours of the various data structures.
- E.g.<List> contains the behaviour (methods) that list type data structures should be able to execute
- You can't instantiate these... they are just interfaces



Any class that implements me must provide these methods."

An ASIDE

```
interface Readable {
    void read();
}

interface Borrowable extends Readable
{
    void borrow(String name);
}
```

- Interfaces can extend other interfaces without implementing anything.
  This allows you to build a hierarchy of behaviours that classes implement later.

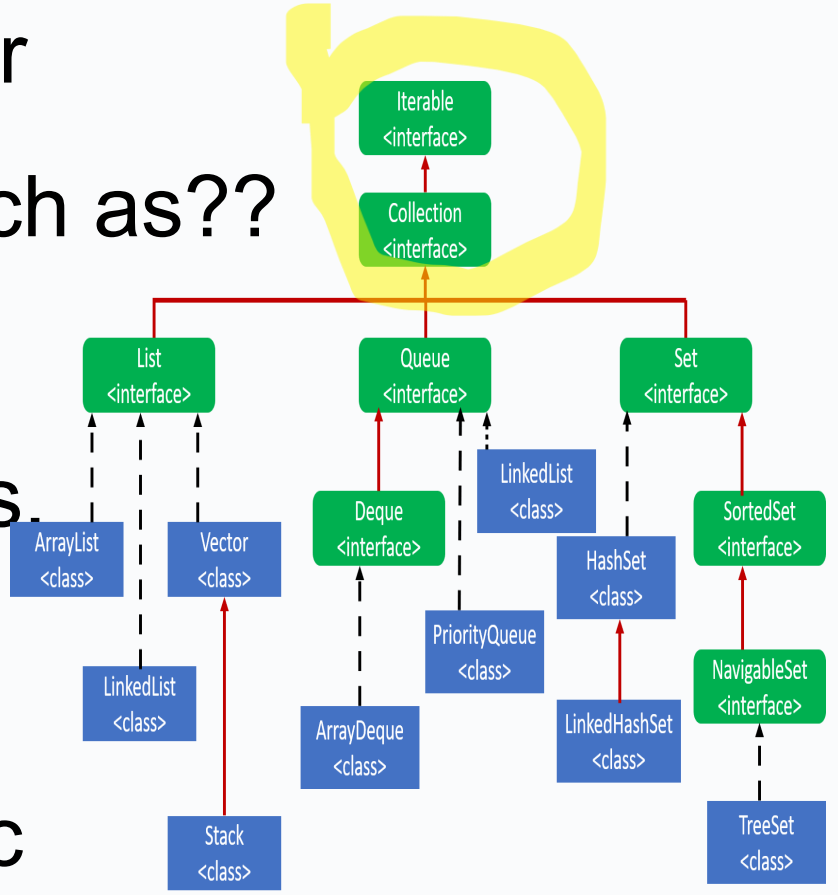
Iterator and Collection Interfaces

- At the top: i.e. the common denominator

  behaviour that all collections need – such as??

  - e.g. get the next element, add an element, remove an element, contains, size etc

  - Behaviour that becomes more specific relates to... e.g. duplicates, ordered .. etc



public interface Collection<E>
extends Iterable<E>

- Collection — the root of the collection hierarchy.
- **The Collection Interface describes the basic operations that *all collection classes should implement***
- i.e. The Collection interface *is the common denominator behaviour* that all collections implement
- e.g. add an element, check if the collection is empty
- Don’t forget.. interfaces don’t DO anything .. they just specify *what to do*, not how.

public interface Collection<E>

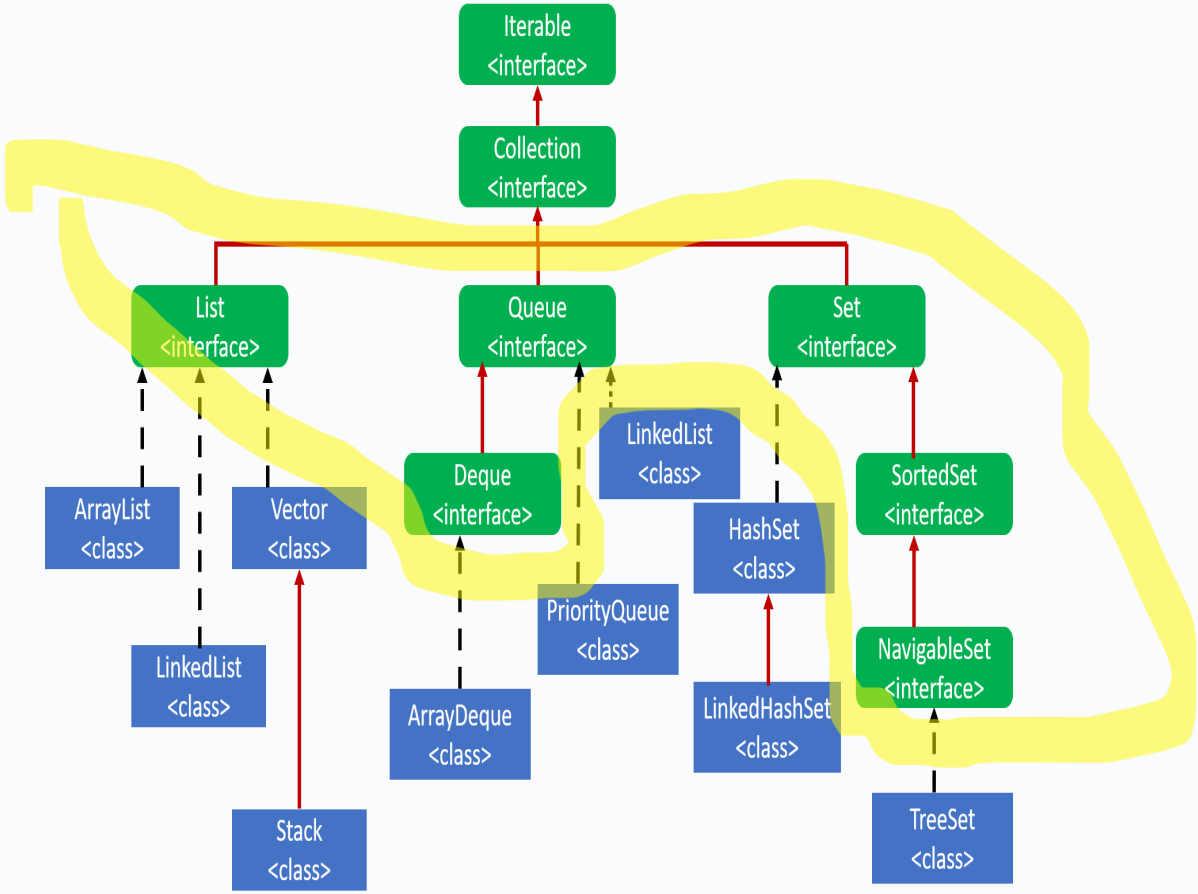
extends Iterable<E>

Actual code!

```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);
    boolean remove(Object element);
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

The next level of interfaces branches into specific behaviours

- Are the elements in the collection always ordered?
- Can they have duplicates?
- Can they have null values?
- Does it need to be thread safe ? (Vector vs array list)
- Double ended? (e.g. deque)
- etc



More Interfaces (that extend Collection Interface)

Lists. Ordered collections of values that allow the client to add and remove elements. ArrayList class falls into this category. Duplicate values allowed.

Sets. Unordered collections of values in which a particular object can appear at most once.

Queues – using FIFO,

Maps. Structures that create associations between keys and values. The HashMap class is in this category.

Summary List

- ✓ **Ordered**
- ✓ **Allows duplicates**
- ✓ **Indexed access**

- **Examples:**
- ArrayList
- LinkedList
- **Example operations**

Summary Queue

- **For processing elements in order**
- **✓ Usually FIFO (first in, first out)**
- **✓ Adds at back, removes from front**

```
Queue<String> queue = new LinkedList<>();

queue.offer("First");

queue.offer("Second");

queue.offer("Third");


System.out.println("\nQUEUE (FIFO):");

System.out.println("Poll: " + queue.poll()); // removes First

System.out.println("Poll: " + queue.poll()); // removes Second

System.out.println("Remaining: " + queue);
```

e.g. public interface Queue<E> extends Collection<E>

```
public interface Queue<E> extends
  Collection<E> {
     E element();                  //throws
     E peek();                  //null
     boolean offer(E e);  //add - bool
     E remove();                //throws
     E poll();                  //null
}
```

Not that obvious what they do… what would you need to do with a queue? E.g Program that manages “lift buttons” ?

Queue interface methods

`peek()` `element()` `poll()` `remove()`

- The **peek()** method retrieves the value of the first element of the queue without removing it from the queue.
- The **element()** method behaves like peek(), so it again retrieves the value of the first element without removing it. Unlike peek , however, if the list is empty element() throws a NoSuchElementException
- The **poll()** method retrieves the value of the first element of the queue by removing it from the queue. At each invocation it removes the first element of the list
- The **remove()** method behaves as the poll() method, so it removes the first element of the list and if the list is empty it throws a NoSuchElementException

Set Summary

- ✓ **No duplicates**
- ✓ **No index**
- ✓ **Order not guaranteed (except LinkedHashSet / TreeSet)**
- A Set is about **uniqueness**.
  It does **not** store things in positions.

```
// SET — no duplicates, no guaranteed order (HashSet)


Set<String> set = new HashSet<>();

set.add("Dog");

set.add("Cat");

set.add("Dog"); // duplicate ignored


System.out.println("\nSET (no duplicates):");

for (String item : set) {

System.out.println(item);

}
```

public interface Set<E> extends Collection<E>

```
public interface Set<E> extends Collection<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator();
    Object[] toArray();
    <T> T[] toArray(T[] a);

    // Modification operations
    boolean add(E e);
    boolean remove(Object o);

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean retainAll(Collection<?> c);
    boolean removeAll(Collection<?> c);
    void clear();

    // Comparison and hashing
    boolean equals(Object o);
    int hashCode();
}
```

"Set" related methods

Collection of objects , unordered, with no duplicates.

All required behaviour summed up in these methods (apart from Iteration and Collection methods which are already specified)

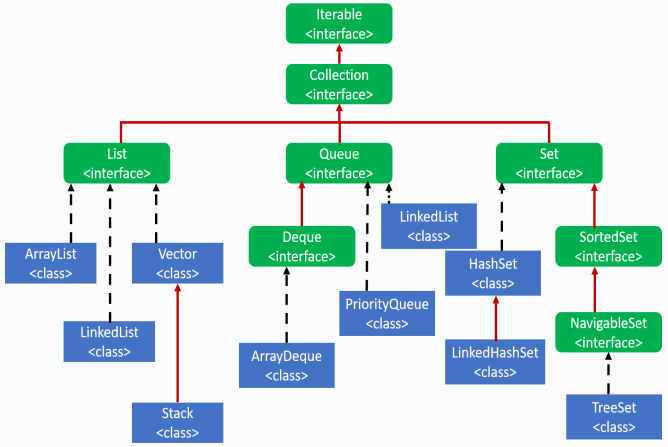
public interface **SortedSet<E>** extends Set<E>

- More specific behaviour than Set
- SortedSet — a Set that maintains its elements in ascending order. *Several additional operations are provided to take advantage of the ordering*. Sorted sets are used for naturally ordered sets, such as **word lists and membership rolls**.

public interface **SortedSet<E>** extends Set<E>

```
public interface SortedSet<E> extends Set<E> {
    // Range-view
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);

    // Endpoints
    E first();
    E last();

    // Comparator access
    Comparator<? super E> comparator();
}
```

Also have “Map”



Collection represents a group of (like a List of Strings). individual elements
Map<K, V> represents a mapping of **key-value pairs** — it’s fundamentally different to Collections. So has its own hierarchy

Map Summary

- **Map**
- **✓ Stores key → value pairs**
- **✓ Keys must be unique**
- **✓ Values can be duplicates**
- **❌ No index**
- Acts like a dictionary or lookup table.

```
Map<Integer, String> map = new HashMap<>();

map.put(1, "Red");

map.put(2, "Blue");

map.put(1, "Green"); // key already exists → value replaced
```

```
System.out.println("\nMAP (key → value):");

for (Map.Entry<Integer, String> entry : map.entrySet()) {

System.out.println("Key " + entry.getKey() + ": " + entry.getValue());

}
```

public interface **Map<K,V>**

- Map — an object that maps keys to values.
- A Map cannot contain duplicate keys;
- Each key can map to at most one value.

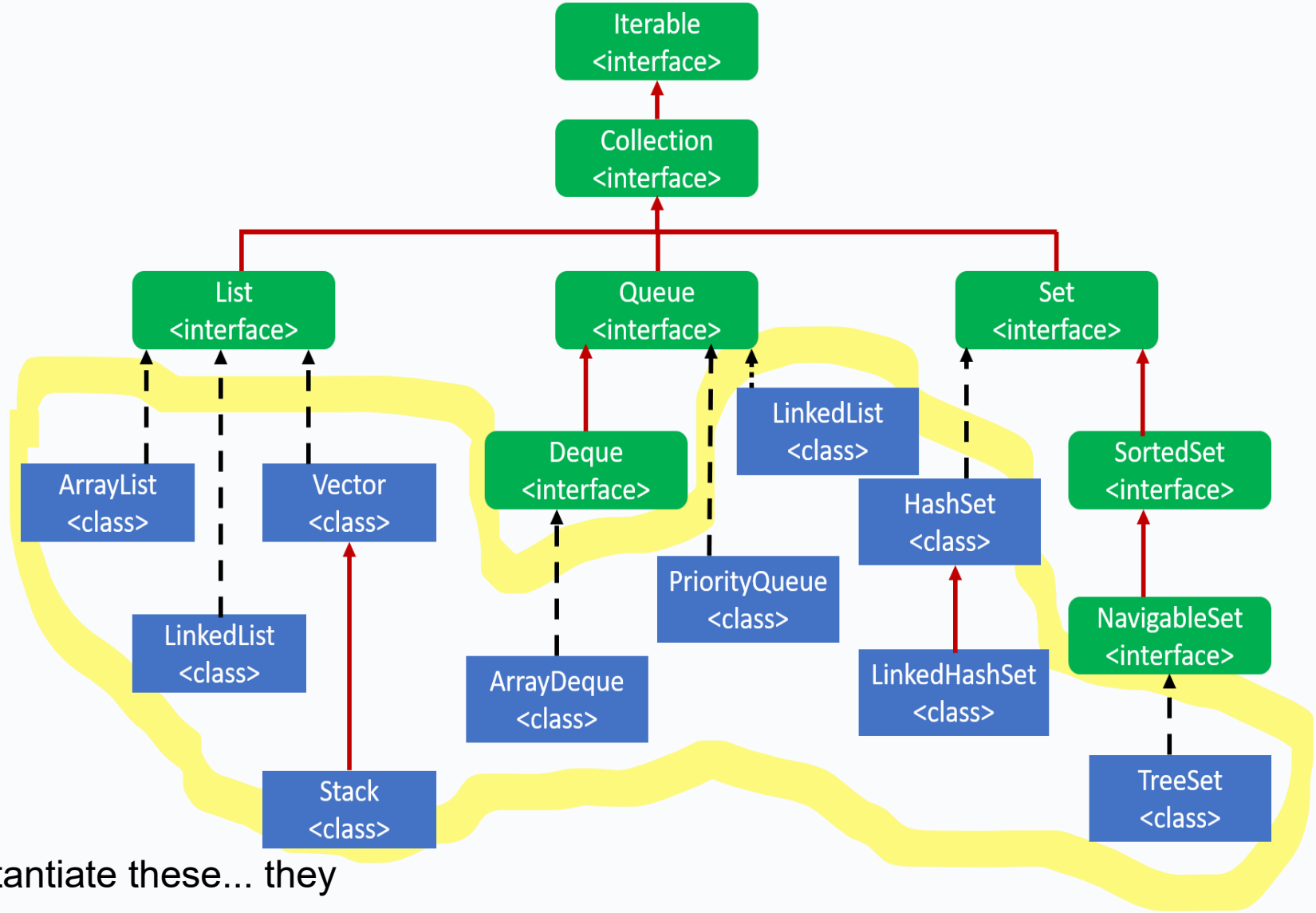
If you've used Hashtable, you're already familiar with the basics of Map.

public interface **Map<K,V>**

```
public interface Map<K,V> {

    // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk operations
    void putAll(Map<? extends K, ? extends V> m);
    void clear();

    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();

    // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

USING the collections framework..

These Java API concrete classes are provided in the API



You can't instantiate these... they are just interfaces

e.g. Using a List in your application

- Two most common concrete classes are `arrayList` and `LinkedList` (which also implements queue)
  - LinkedLists – every entry maintains pointers to neighbouring entries in the list
  - ArrayLists just have **index** for each entry
  - Results in changes to performance for insert, delete, memory usage etc (Big O)

```
ArrayList<String> myList = new ArrayList<String>
```

Big-O notation is a mathematical way to describe how the performance of an algorithm changes as the amount of data grows.

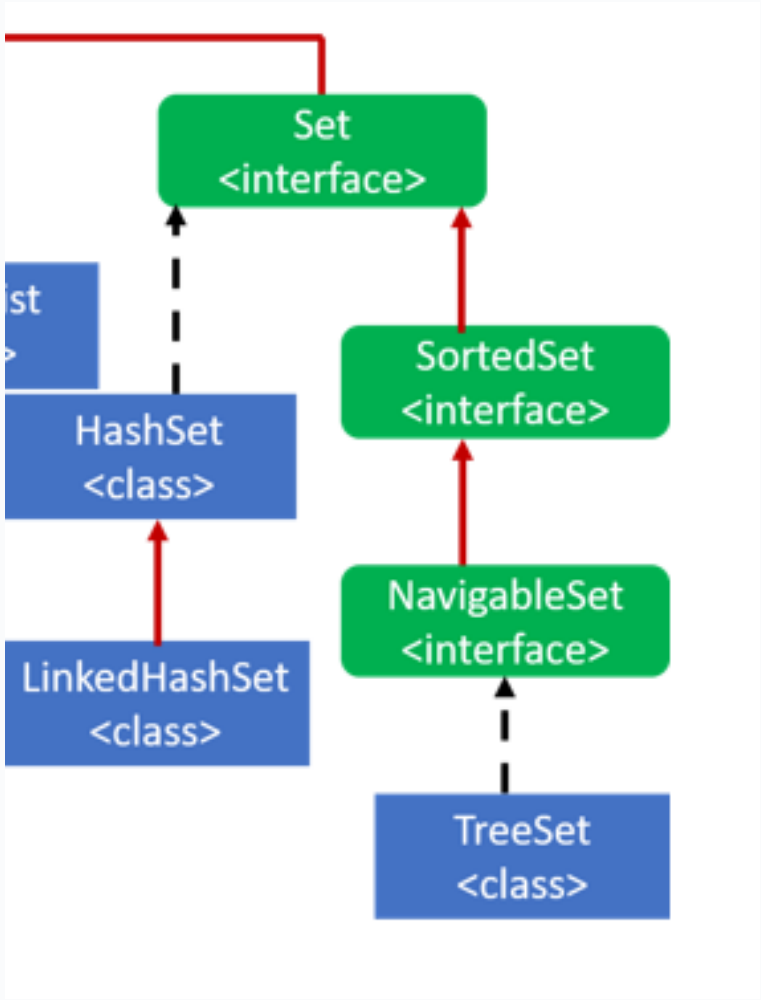
Because in programming:
10 elements → everything is fast
10,000 → mistakes start to show
10,000,000 → bad choices can make your program unusable

ArrayList is excellent when you mostly **access by index** or **append to end**, but poor for lots of insert/delete in the middle.

ArrayList
get(index) → **O(1)**
add at end → **O(1)**
insert at middle → **O(n)**
contains() → **O(n)**

LinkedList
•add/remove at ends → **O(1)**
•get(index) → **O(n)**
•contains() → **O(n)**

e.g. Set



- Three most common concrete classes are

`HashSet`,

`LinkedHashSet`

`TreeSet`

Code

- Write the code for creating
  - a collection of values (e.g. guessed numbers in a quiz to guess a number);
  - If a guess was made before, it should be detected .

HashSet (type of Set)

- HashSet creates a collection that uses a hash table for storage.
- A hash table stores information by using hashing (the informational content of a key is used to determine a unique value (hash code)
- The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically.
- HashSet can’t contain duplicates
- $O(1)$ seek time for an element – (i.e.access time is constant - independent of numbers of elements)

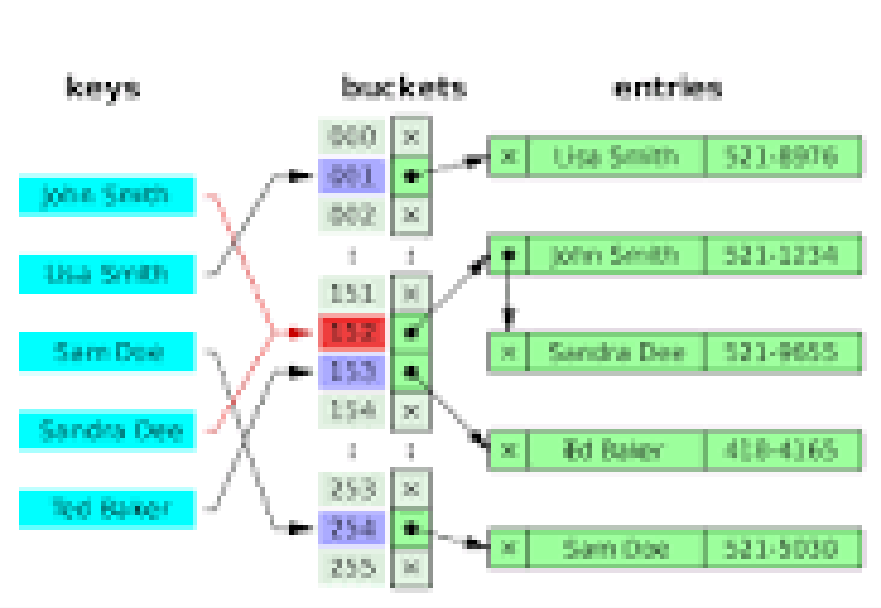
Hashing

- Takes data (e.g. "apple" or 42)
- Runs it through a **hash function**
- Produces a numeric result (hash code)
- That hash code maps the data to a **bucket** in a hash table

  (**bucket** is one storage location inside a hash table.)

HashSet is based on hash table

- Each entry in the hashtable has is put in a “bucket”. The bucket address is found by using a hashing algorithm to calculate it, from the contents of the entry.
- *Fast to retrieve!*



Why HashSet is so fast?

- Because it uses **hashing** → it jumps straight to the correct **bucket** instead of searching everything.
- This gives:
- **Average time: O(1)**
- Adding: O(1)
- Checking exists: O(1)

What collection to use in my program:

To store and manipulate:

- the names of all the people in this class
- a hand of playing cards
- a task scheduling app
- a deck of 52 playing cards, stored by suit/ number, not shuffled
- the names of all the countries in Europe.
- An organiser app that stores list of to-dos
- to write a lift simulation programme and store the set of lift requests waiting in the system
- An online polling app store names against votes

Complete Summary Table

Scenario	Best Collection	Reason
Class list	ArrayList	Ordered, duplicates allowed
Hand of cards	ArrayList	Small, ordered, easy updates
Task scheduler	PriorityQueue	Next task by priority
Sorted deck	ArrayList	Fixed order, index needed
Countries in Europe	HashSet	No duplicates, fast lookup
To-do organiser	ArrayList	List UI, ordered
Lift requests	Queue (ArrayDeque)	FIFO behaviour
Polling app (name → votes)	HashMap	Key/value pairs

What do we know so far about the “Java Collections Framework?

- We know there are a set of interfaces (Collection, Set, Queue, List, & Map) -

- These define “behaviour” or characteristics of particular types of collections

- The interfaces are implemented by a set of useful concrete classes that we use all the time – e.g. HashSet, ArrayList, HashMap etc

- All of this is shown diagrammatically (previously)

Questions

- What is the difference between Set, Queue, List re **duplicates** and **ordering**?
- What are they? (from a code point of view?)
- What is the BigO performance of Hashset – and why?
- Name 5 concrete classes in the Collections framework?

The remainder of the Collections Framework

- The rest of the Collections Framework consists of
  - A **Collections Class** that contains general useful implemented methods (and yes, the name is confusing!)
  - A useful **iterator** interface (up the top)

`Collections` class

- `Collections` **class is a utility class ,** which has specific methods to work with collections.
  https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html

- This class consists exclusively **of static methods that** operate on or return collections.
  You **do not create an object** of this class.
  You just call its **static methods**.

- Remember what `static` means ? How do you call static methods for a class?

Using the collections Class

- It (also) does a bunch of useful operations that you might want to do on a Collection
  - sort(), shuffle(), search(), reverse(),

To USE it, you call the methods you want, passing in the collection (i.e. data structure) you are working with..

e.g.

```
ArrayList<String> letters = new ArrayList<String>();
      letters.add("A");
      letters.add("B");
      letters.add("C");

Collections.shuffle(letters); // random reordering
```

Collections Class vs Collection Interface?

- Confusing .. Yes!!!.
- Collections class contains implemented behaviour to perform useful things.

  e.g. `Collections.shuffle(myCollection);`

- Collection Interface is just an Interface (which consists of empty methods) that sits at the top of the Collections hierarchy)

Benefits of the Java collections framework

- **Reduces programming effort:**
- **Increases program speed and quality:** This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms.
- **Allows interoperability among unrelated APIs:**
- **Reduces effort to learn and to use new APIs:**
- **Reduces effort to design new APIs:**
- **Fosters software reuse:** New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

In summary

- The Java Collection Framework package (java.util) contains:

1. A set of interfaces (i.e. Collection, Set, List etc),
2. Implementation classes (e.g. ArrayList), and
3. Algorithms (such as sorting and searching) (In the Collections class)

Note: Data structure operations Big-O complexity..

Big-O notation gives worst case scenario for the processing time of an algorithm

e.g.

O(1) = algorithm will always take the same amount of time, regardless of data size. e.g. find an entry on a hashtable

O(n) = time will increase linearly according to data set size (n)

e.g. find a character in a string

Data structure operations Big-O complexity..

Big-O notation gives worst case scenario

It is useful to be aware of these

http://bigocheatsheet.com/