Wrapper Class

In Java programming language, there are eight primitive types and each of these has a corresponding library class of reference type. For example, there is a class **java.lang.Integer** that corresponds to primitive type int.

These kinds of classes are called wrappers. The wrapper classes are immutable; we cannot change a wrapped value after the wrapper has been constructed. They are also final, so we cannot subclass them.

Why do we have wrapper classes for primitive types?

- The wrapper classes are used whenever a primitive type needs to be treated as an Object. For example, the classes in the Collections API (like ArrayList) can only hold Object references.
- To provide an assortment of utility functions for primitives. Most of these functions are related to various conversions: converting primitives to and from String objects, and converting primitives and String objects to and from different bases (or radix), such as binary, octal, and hexadecimal.

Boxing and Unboxing in Java

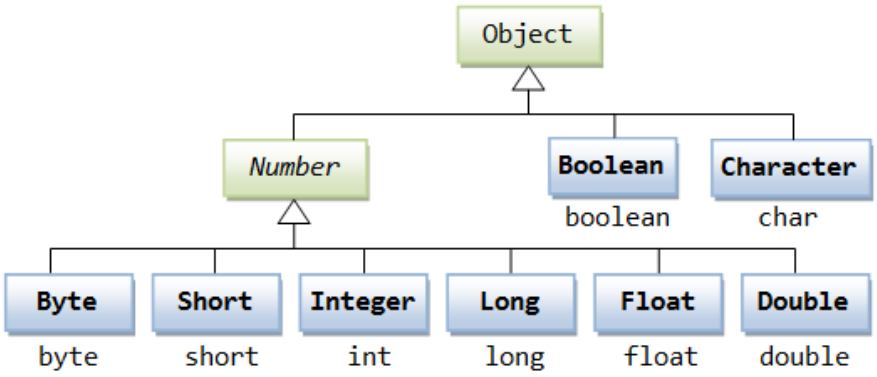
Conversion of a primitive type to the corresponding reference type is called **boxing**, such as an int to a java.lang.Integer.

Autoboxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an **`int`** to an **`Integer`**, a **`double`** to a **`Double`**, and so on.

Conversion of the reference type to the corresponding primitive type is called **unboxing**, such as Byte to byte.
Since JDK 1.5, Conversion from primitive types to corresponding wrapper objects and vice versa can happen automatically.

The Java compiler applies **autoboxing** when a primitive value is:

- Passed as a parameter to a method that expects an object of the corresponding wrapper class.
- Assigned to a variable of the corresponding wrapper class.



```
Integer intObj = 5566;    // autobox from int to Integer
int i = intObj;           // auto-unbox from Integer to int
Double doubleObj = 55.66; // autoboxing from double to Double
double d = doubleObj;     // atuo-unbox from Double to double
```

Collection Framework

Although we can use an array to store a group of elements of the same type (either primitives or objects). The array, however, does not support so-called *dynamic allocation* - it has a ***fixed length*** which cannot be changed once allocated. Furthermore, array is a simple linear structure. Many applications may require more complex data structure such as linked list, stack, hash table, sets, or trees.

In Java, dynamically allocated data structures (such as ArrayList, LinkedList, Vector, Stack, HashSet, HashMap, Hashtable) are supported in a unified architecture called the Collection Framework, which mandates the common behaviors of all the classes.

A collection, as its name implied, is simply an object that holds a collection (or a group, a container) of objects. Each item in a collection is called an element. A framework, by definition, is a set of interfaces that force you to adopt some design practices. A well-designed framework can improve your productivity and provide ease of maintenance.

In terms of programming, a collection is a data structure that holds a set of objects in a specific manner. It looks like arrays but collections are more advanced and more flexible. An array simply stores a fixed number of objects, whereas a collection stores objects dynamically, i.e. you can add or remove objects as you wish.

Collections in java is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections Framework is a set of reusable data structures and algorithms which are designed to free programmers from implementing data structures themselves so that they can focus on business logics.

The Java Collections Framework provides common data structures implementations which are enough for general-purpose such as list, set, map, queue, tree, etc. These collections are high-performance, high-quality, and easy to use with very good documentation.

In addition, the Java Collections Framework provides useful and robust algorithms such as searching and sorting on collections, and the interoperability between collections and arrays.

All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections.

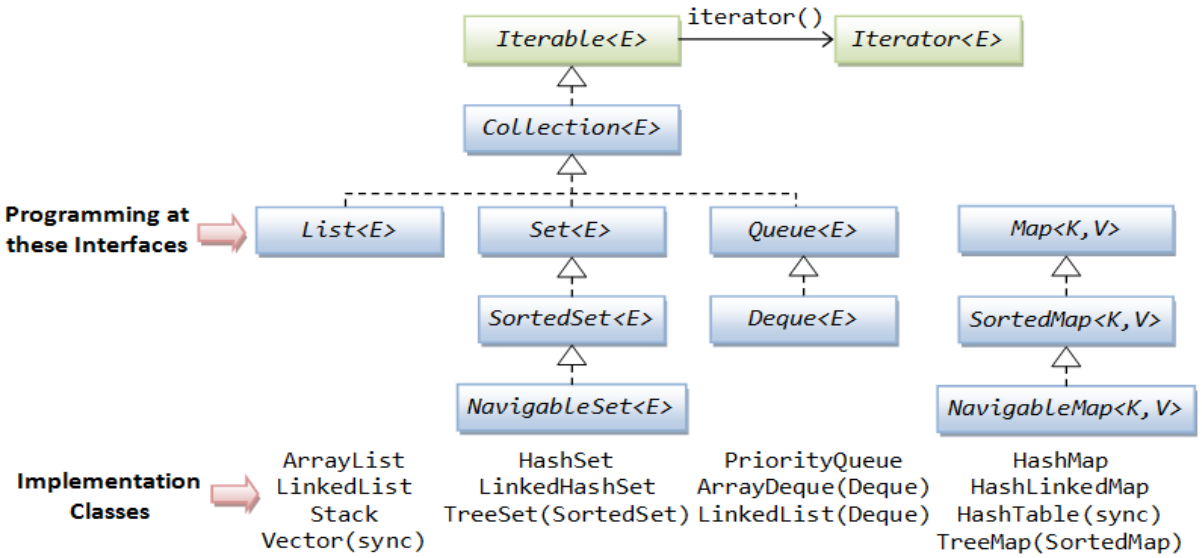
Collection Framework

The collection framework provides a unified interface to store, retrieve and manipulate the elements of a collection, regardless of the underlying and actual implementation. This allows the programmers to program at the interfaces, instead of the actual implementation.

Java Collection simply means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque etc.) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet etc)

Interfaces

The **core collection interfaces** encapsulate different types of collections, which are shown in the figure below. These interfaces allow collections to be manipulated independently of the details of their representation. Core collection interfaces are the foundation of the Java Collections Framework. As you can see in the following figure, the core collection interfaces form a hierarchy.



Iterable<E> Interface

The `Iterable<E>` interface, which takes a generic type `E` and read as `Iterable` of element of type `E`, declares one `abstract` method called `iterator()` to retrieve the `Iterator<E>` object associated with all the collections. This`Iterator` object can then be used to transverse through all the elements of the associated collection.

```
Iterator<E> iterator();  // Returns the associated Iterator instance
                         // that can be used to transverse thru all the elements of the collection
```

All implementations of the collection (e.g., `ArrayList`, `LinkedList`, `Vector`) must implement this method, which returns an object that implements `Iterator` interface.

Iterator<E> Interface

The `Iterator<E>` interface, declares the following three `abstract` methods:

```
boolean hasNext()    // Returns true if it has more elements
E next()             // Returns the next element of generic type E
void remove()        // Removes the last element returned by the iterator
```

As seen in the introductory example, you can use a while-loop to iterate through the elements with the `Iterator` as follows:

```
List<String> lst = new ArrayList<String>();
lst.add("alpha");
lst.add("beta");
lst.add("charlie");
// Retrieve the Iterator associated with this List via the iterator() method
Iterator<String> iter = lst.iterator();

// Transverse thru this List via the Iterator
while (iter.hasNext())    // Retrieve each element and process
{
   String str = iter.next();
   System.out.println(str);
}
```

Collection Interface

The `Collection<E>`, which takes a generic type `E` and read as Collection of element of type `E`, is the root interface of the Collection Framework. It defines the common behaviors expected of all classes, such as how to add or remove an element, via the following abstract methods:

The Collection interface is the least common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired. Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as `Set` and `List`.

List<E>

List Interface Models a resizable linear array, which allows indexed access. List can contain duplicate elements. Frequently-used implementations of List include `ArrayList`, `LinkedList`, `Vector` and `Stack`

Set<E>

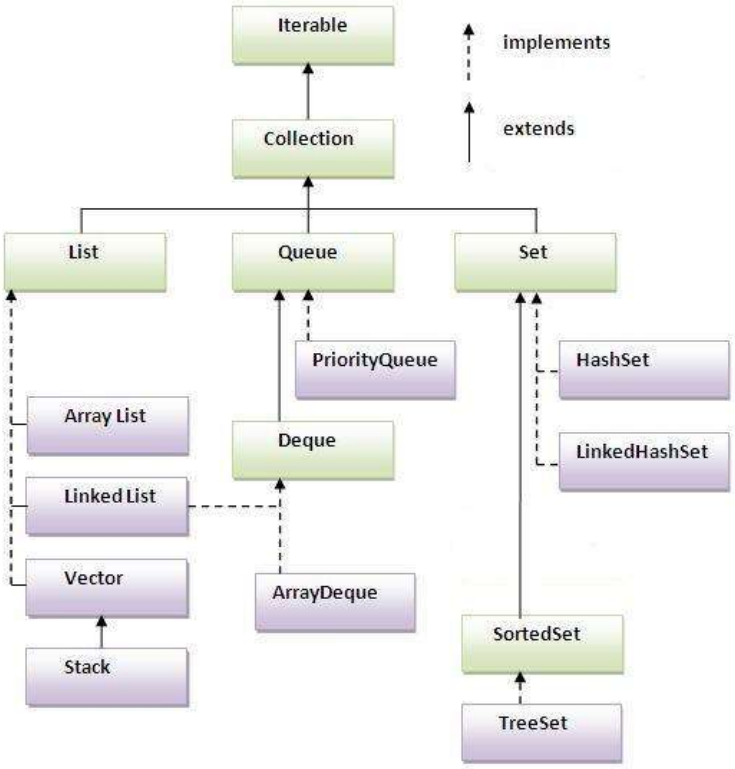
It models a mathematical set, where no duplicate elements are allowed. Frequently-used implementations of `Set` are `HashSet` and `LinkedHashSet`. The sub-interface `SortedSet<E>` models an ordered and sorted set of elements, implemented by `TreeSet`.

Queue<E>

`Queue<E>` interface models queues such as First-in-First-out (FIFO) queue and priority queue. It sub interface `Deque<E>` models queues that can be operated on both ends. Implementations include `PriorityQueue`, `ArrayDeque` and `LinkedList`.

Map<K,V>,

The interface `Map<K,V>`, which takes two generic types K and V and read as `Map` of Key type K and Value type V, is used as a collection of "key-value pairs". No duplicate key is allowed. Frequently-used implementations include `HashMap`, `Hashtable` and `LinkedHashMap`. Its sub-interface `SortedMap<K, V>` models an ordered and sorted map, based on its key, implemented in `TreeMap`.



Methods of Collection interface

There are many methods declared in the Collection interface. They are as follows:

No.	Method	Description
1	public boolean **add**(Object element)	is used to insert an element in this collection.
2	public boolean **addAll**(Collection c)	is used to insert the specified collection elements in the invoking collection.
3	public boolean **remove**(Object element)	is used to delete an element from this collection.
4	public boolean **removeAll**(Collection c)	is used to delete all the elements of specified collection from the invoking collection.
5	public boolean **retainAll**(Collection c)	is used to delete all the elements of invoking collection except the specified collection.
6	public int **size**()	return the total number of elements in the collection.
7	public void **clear**()	removes the total no of element from the collection.
8	public boolean **contains**(Object element)	is used to search an element.
9	public boolean **containsAll**(Collection c)	is used to search the specified collection in this collection.
10	public Iterator **iterator**()	returns an iterator.
11	public Object[] toArray()	converts collection into array.

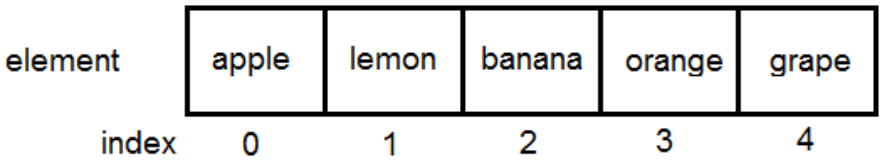
12	public boolean isEmpty()	checks if collection is empty.
13	public boolean equals(Object element)	matches two collection.
14	public int hashCode()	returns the hashcode number for collection.

What is a List?

A **`List`** is a kind of collections in the Java Collection Framework. It's used widely in Java programming and programmers love it.

A **`List`** collection maintains elements in form of index-based, meaning the first element is stored at 0-index, the second one is at 1-index, the third one is at 2-index, and so on.

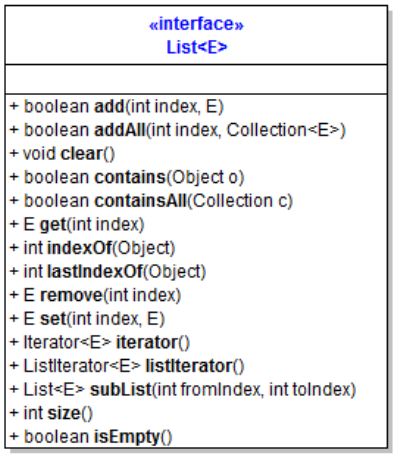
A list collection stores elements by insertion order (either at the end or at a specific position in the list). A list maintains indices of its elements so it allows adding, retrieving, modifying, removing elements by an integer index (zero-based index; the first element is at 0-index, the second at 1-index, the third at 2-index, and so on). The following picture illustrates a list that stores some String elements:



A List collection

A list can store objects of any types. Primitive types are automatically converted to corresponding wrapper types, e.g. integer numbers are converted to `Integer` objects. It allows null and duplicate elements, and orders them by their insertion order (index).

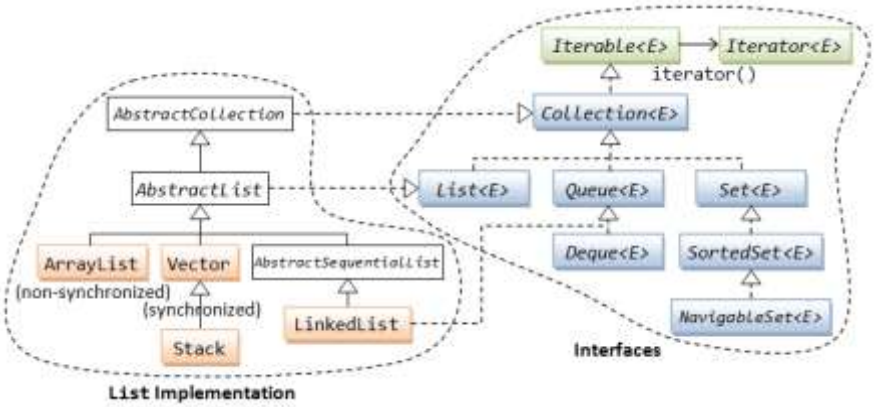
The following class diagram depicts the primary methods defined in the **`java.util.List`** interface:



Characteristics of Lists:

A **`List`** collection can store any objects. It maintains elements by insertion order, meaning that when you add an object to a list, the object will be placed at the end of the list.

A **`List`** allows null and duplicate elements, and orders them by their insertion order, hence most operations on list are based on the indices.



List Implementations:

The Java Collection Framework provides two major implementations of the `List` interface. They are **`ArrayList`** and **`LinkedList`**. That means `List` is the super interface, and **`ArrayList`** and **`LinkedList`** are two sub classes.

ArrayList

An implementation that stores elements in a backing array. The array's size will be automatically expanded if there isn't enough room when adding new elements into the list. It's possible to set the default size by specifying an initial capacity when creating a new **`ArrayList`**.

- Java ArrayList class uses a **dynamic array (Resizable Array Data Structure)** for storing the elements. It extends **AbstractList** class and implements List interface.
- 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 Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

Basically, an **`ArrayList`** offers constant time for the following operations: **`size`**, **`isEmpty`**, **`get`**, **`set`**, **`iterator`**, and **`listIterator`**; amortized constant time for the **`add`** operation; and linear time for other operations. Therefore, this implementation can be considered if we want fast, random access of the elements.

LinkedList

An implementation that stores elements in a doubly-linked list data structure. It offers constant time for adding and removing elements at the end of the list; and linear time for operations at other positions in the list. Therefore, we can consider using a **`LinkedList`** if fast adding and removing elements at the end of the list is required.

- Java LinkedList class uses **doubly linked list** to store the elements. It extends the AbstractList class and implements List and Deque interfaces.
- Java LinkedList class can contain duplicate elements.
- Java LinkedList class maintains insertion order.
- Java LinkedList class is non synchronized.
- In Java LinkedList class, manipulation is fast because no shifting needs to be occurred.
- Java LinkedList class can be used as list, stack or queue.

fig- doubly linked list

Besides **ArrayList** and **LinkedList**, `Vector` class is a legacy collection and later was retrofitted to implement the **List** interface. **Vector** is thread-safe, but **ArrayList** and **LinkedList** are not. The following class diagram depicts the inheritance tree of the **List** collections:

Vector

Vector is also a List implementation. However, **Vector** is an old collection which was created before the Java Collection Framework. Nowadays, **Vector** is obsolete, and it exists only for the purpose of backward compatibility with old APIs.

Why and When Use Lists?

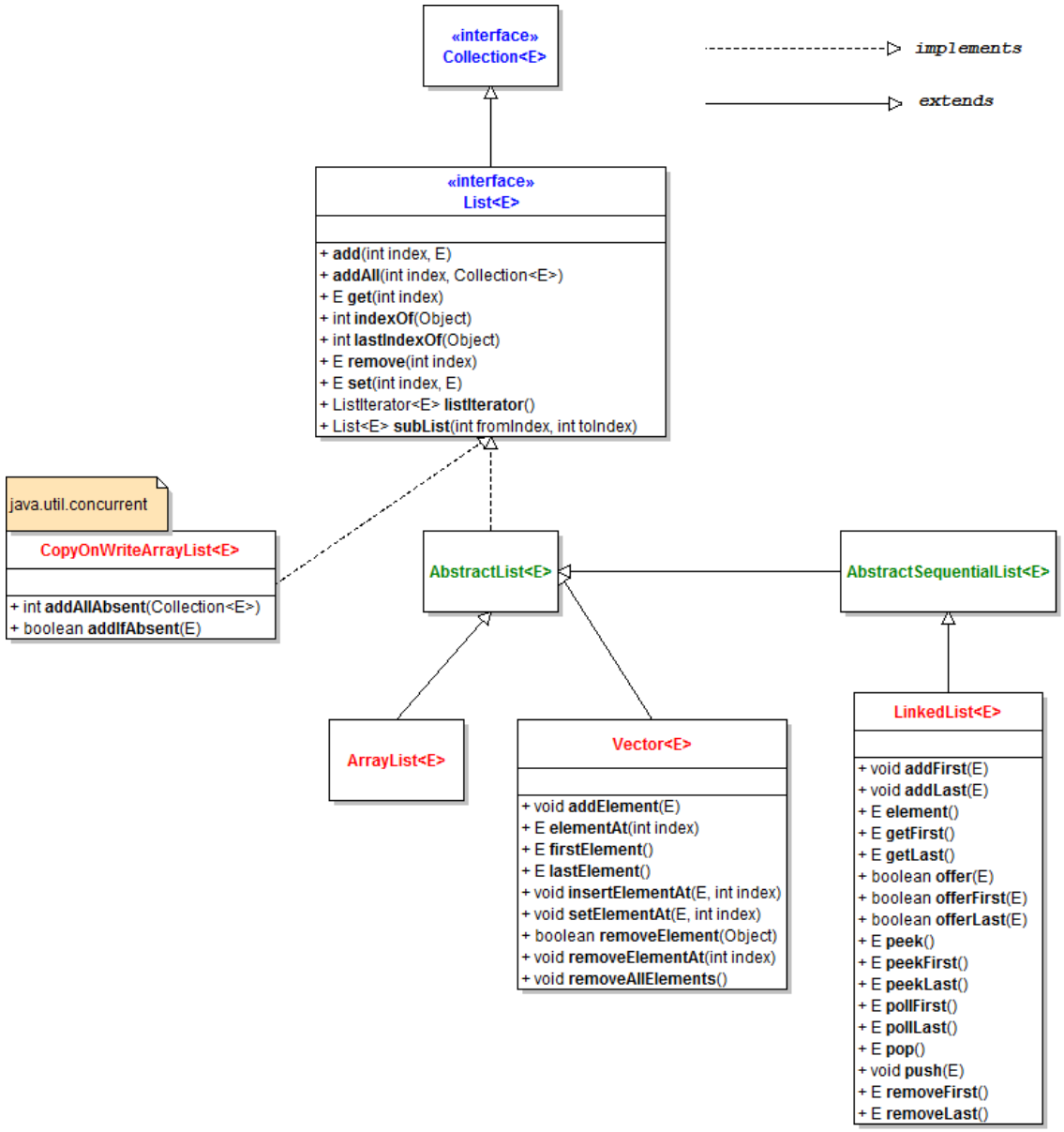
Consider using a `List` collection when you want to store and retrieve elements based on index.

More specifically, consider using an **ArrayList** when you want fast, random accessing of elements as an **ArrayList** provides constant time access to any elements in the list.

And consider using a **LinkedList** when you want fast adding and removing elements at the end of the list, as a **LinkedList** store elements in a doubly-linked data structure.

- **List**<E> is the base interface for all kinds of list. It defines general operations for a List type.
- Abstract subclasses: **AbstractList<E>** and **AbstractSequentialList**<E>
- Concrete implementation classes: **ArrayList<E>, Vector<E>, LinkedList<E>** and **CopyOnWriteArrayList<E>(**this class is under **java.util.concurrent** package).
- Legacy collection: **Vector<E>**
- Implementation classes in JDK which are not members of Java Collections Framework: **AttributeList**,**RoleList**, **RoleUnresolvedList** and **Stack**.

The following class diagram describes the hierarchy structure of List API in Java Collections Framework:



1. Creating a new list

It’s a good practice to declare a list instance with a generic type parameter, for example:

```
List<Object> listAnything = new ArrayList<Object>();

List<String> listWords = new ArrayList<String>();

List<Integer> listNumbers = new ArrayList<Integer>();

List<String> linkedWords = new LinkedList<String>();
```

Since Java 7, we can remove the type parameter on the right side as follows:

```
List<Integer> listNumbers = new ArrayList<>();

List<String> linkedWords = new LinkedList<>();
```

The compiler is able to infer the actual type parameter from the declaration on the left side.

When creating a new `ArrayList` using the empty constructor, the list is constructed with an initial capacity of ten. If you are sure how many elements will be added to the list, it's recommended to specify a capacity which is large enough. Let's say, if we know that a list contains around 1000 elements, declare the list as follows:

```
List<Integer> listNumbers = new ArrayList<>(1000);
```

It's also possible to construct a list that takes elements from an existing collection, for example:

```
List<Integer> listNumberOne;      // existing collection

List<Integer> listNumberTwo = new ArrayList<>(listNumberOne);
```

The `listNumberTwo` constructed with copies of all elements from the `listNumberOne`.

2. Basic operations: adding, retrieving, updating, removing elements

Adding elements

The methods **add(Object)**, **add(index, Object)** and **addAll(Collection)** are used to add elements to the list. It requires to add elements of the same type (or sub type) as the type parameter declared by the list. For example:

```
List<String> listStrings = new ArrayList<String>();

listStrings.add("One");

listStrings.add("Two");

listStrings.add("Three");

 // But this will cause compile error

listStrings.add(123);
```

Adding elements of sub types of the declared type:

```
List<Number> linkedNumbers = new LinkedList<>();

linkedNumbers.add(new Integer(123));

linkedNumbers.add(new Float(3.1415));

linkedNumbers.add(new Double(299.988));

linkedNumbers.add(new Long(67000));
```

We can insert an element into the list at a specified index, for example:

```
listStrings.add(1, "Four");
```

That inserts the String “Four” at the 2nd position in the list.

We can also add all elements of an existing collection to the end of the list:

```
listStrings.addAll(listWords);
```

Or add the elements to the list at a specified position:

```
listStrings.addAll(2, listWords);
```

That inserts all elements of the `listWords` collection at 3rd position of the `listStrings` collection.

Retrieving elements

The **get(index)** method is used to retrieve an element from the list at a specified index. For example, the following code gets an element at 2nd position in the array list and an element at 4th position in the linked list:

```
String element = listStrings.get(1);

Number number = linkedNumbers.get(3);
```

For a **`LinkedList`** implementation, we can get the first and the last elements like this:

```
LinkedList<Number> numbers = new LinkedList<Number>();

// add elements to the list...
// get the first and the last elements:

Number first = numbers.getFirst();

Number last = numbers.getLast();
```

Note that the **`getFirst`**`()` and **`getLast`**`()` methods are specific to the `LinkedList` class.

Updating elements

Use the **set(index, element)** method to replace the element at the specified index by the specified element. For example:

```
listStrings.set(2, "Hi");
```

That replaces the 3rd element in the list by the new String "Hi".

Removing elements

To remove an element from the list, use the **remove(index)** or **remove(Object)** method which removes the element at the specified index or by object reference. For example:

Remove the element at the 3rd position in the list:

```
listStrings.remove(2);
```

If the specified index is out of range (index < 0 or index >= list size), a java.lang.IndexOutOfBoundsException is thrown.

Remove the String element "Two" in the list:

```
listStrings.remove("Two");
```

Notes about the remove(Object) method:

- It compares the specified object with the elements in the list using their `equals()` method, so if you use your own defined object type, make sure it implements the `equals()` method correctly.
- It only removes the first occurrence of the specified element in the list (i.e. if a list contains duplicate elements, only the first element is removed).
- It returns true if the list contained the specified element, or false otherwise. Thus it's recommended to check return value of this method, for example:

```
if (listStrings.remove("Ten")) {
        System.out.println("Removed");
} else {
        System.out.println("There is no such element");
}
```

To remove all elements in the list, use the **clear()** method:

```
listStrings.clear();
```

3. Iterating over a list

Basically, we can use the enhanced for loop to iterate through all elements in the list, as follows:

```
for (String element : listStrings) {

        System.out.println(element);

}
```

Or use an iterator like this:

```
Iterator<String> iterator = listStrings.iterator();
while (iterator.hasNext()) {
        System.out.println(iterator.next());
}
```

For more list-specific, use a list iterator as shown below:

```
Iterator<Number> iterator = linkedNumbers.listIterator();
```

```
 while (iterator.hasNext()) {
        System.out.println(iterator.next());
}
```

Since Java 8, we can use the `forEach()` method like this:

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

4. Searching for an element in a list

To search for position of a specific element in the list or to know if the list contains the specified element, the following methods can be used:

- **boolean contains(Object)**: returns true if the list contains the specified element.
- **int indexOf(Object)**: returns the index of the first occurrence of the specified element in the list, or -1 if the element is not found.
- **int lastIndexOf(Object)**: returns the index of the last occurrence of the specified element in the list, or -1 if the element is not found.

Examples:

```
if (listStrings.contains("Hello")) {
        System.out.println("Found the element");
} else {
        System.out.println("There is no such element");
}
int firstIndex = linkedNumbers.indexOf(1234);
int lastIndex = listStrings.indexOf("Hello");
```

Note that the above methods compare the elements using their **`equals()`** method, so if you define your own type, make sure it implements the `equals()` method correctly.

5. Sorting a list

The simplest way to sort out elements in a list is using the **Collections.sort()** static method which sorts the specified list into ascending order, based on the natural ordering of its elements. Here's an example:

```
List<String> listStrings = new ArrayList<String>();
listStrings.add("D");
listStrings.add("C");
listStrings.add("E");
listStrings.add("A");
listStrings.add("B");
System.out.println("listStrings before sorting: " + listStrings);
Collections.sort(listStrings);
```

```
System.out.println("listStrings after sorting: " + listStrings);
Output:
listStrings before sorting: [D, C, E, A, B]
listStrings after sorting: [A, B, C, D, E]
```

Note that all elements in the list must implement the `Comparable` interface, so if you define your own type, make sure it implements that interface and its **`compareTo()`** method.

6. Copying one list into another

The **Collections.copy(dest, src)** static method allows us to copy all elements from the source list into the destination one. Note that the destination list must be large enough to contain the entire source list. Here's an example:

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

List<String> destList = new ArrayList<String>();
destList.add("V");
destList.add("W");
destList.add("X");
destList.add("Y");
destList.add("Z");
System.out.println("destList before copy: " + destList);
Collections.copy(destList, sourceList);
System.out.println("destList after copy: " + destList);
The output would be:
destList before copy: [V, W, X, Y, Z]
destList after copy: [A, B, C, D, Z]
```

7. Shuffling elements in a list

To randomly permute elements in a list, use the **Collections.shuffle()** static method. Here's a quick example:

```
List<Integer> numbers = new ArrayList<Integer>();
for (int i = 0; i <= 10; i++)
      numbers.add(i);
```

```
System.out.println("List before shuffling: " + numbers);


Collections.shuffle(numbers);
System.out.println("List after shuffling: " + numbers);
```

The output would be:

```
List before shuffling: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
List after shuffling: [6, 4, 5, 0, 1, 3, 9, 7, 2, 10, 8]
```

8. Reversing elements in a list

To reverse order of elements in a list, use the **Collections.reverse()** static method. Here's a quick example:

```
List<Integer> numbers = new ArrayList<Integer>();
for (int i = 0; i <= 10; i++) numbers.add(i);
System.out.println("List before reversing: " + numbers);
Collections.reverse(numbers);
System.out.println("List after reversing: " + numbers);
```

The output would be:

```
List before reversing: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
List after reversing: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

9. Extracting a portion of a list

The **subList(fromIndex**, **toIndex)** allows us to get a portion of the list between the specified **fromIndex** (inclusive) and **toIndex** (exclusive). Here's an example:

```
List<String> listNames = Arrays.asList("Tom", "John", "Mary", "Peter", "David",
"Alice");
System.out.println("Original list: " + listNames);
List<String> subList = listNames.subList(2, 5);
System.out.println("Sub list: " + subList);
```

Output:

```
Original list: [Tom, John, Mary, Peter, David, Alice]
Sub list: [Mary, Peter, David]
```

Note that the sub list is just a view of the original list, so any modifications made on the original list will reflect in the sub list.

10. Converting between Lists and arrays

The Java Collection Framework allows us to easily convert between lists and arrays.

The **Arrays.asList(T… a)** method converts an array of type **T** to a list of type **T**. Here's an example:

```
List<String> listNames = Arrays.asList("John", "Peter", "Tom", "Mary", "David",
"Sam");

List<Integer> listNumbers = Arrays.asList(1, 3, 5, 7, 9, 2, 4, 6, 8);

System.out.println(listNames);

System.out.println(listNumbers);
```

Output:

```
[John, Peter, Tom, Mary, David, Sam]

[1, 3, 5, 7, 9, 2, 4, 6, 8]
```

And the **List** interface provides the **toArray()** method that returns an array of Objects containing all of the elements in the list in proper sequence (from first to last element). Here's an example:

```
List<String> listWords = new ArrayList<String>();

// add elements to the list

Object[] arrayWords = listWords.toArray();
```

And the **toArray(T[] a)** method returns an array of type **T**, for example:

```
String[] words = listWords.toArray(new String[0]);

Integer[] numbers = listNumbers.toArray(new Integer[0]);
```

Note that the returned array contains copies of elements in the list, that means we can safely modify the array without affecting the list.

11. Concurrent lists

By default, **`ArrayList`** and **`LinkedList`** are not thread-safe, so if you want to use them in concurrent context, you have to synchronize them externally using the **Collections.synchronizedList()** static method which returns a synchronized list that wraps the specified list. For example:

```
List<Object> unsafeList = new ArrayList<Object>();

List<Object> safeList = Collections.synchronizedList(unsafeList);
```

Note that you must manually synchronize the returned list when iterating over it, for example:

```
synchronized (safeList) {
        Iterator<Object> it = safeList.iterator();
        while (it.hasNext()) {
                System.out.println(it.next());
        }
}
```

Difference between ArrayList and LinkedList

ArrayList and LinkedList both implements List interface and maintains insertion order. Both are non synchronized classes.

But there are many differences between ArrayList and LinkedList classes that are given below.

ArrayList	LinkedList
ArrayList internally uses **dynamic array** to store the elements.	LinkedList internally uses **doubly linked list** to store the elements.
Manipulation with ArrayList is **slow** because it internally uses array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses doubly linked list so no bit shifting is required in memory.
ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and **Deque interfaces**.
ArrayList is **better for storing and accessing** data.	LinkedList is **better for manipulating** data.

Java Non-generic Vs Generic Collection

Java collection framework was non-generic before JDK 1.5. Since 1.5, it is generic.

Java new generic collection allows you to have only one type of object in collection. Now it is type safe so typecasting is not required at run time.

Let's see the old non-generic example of creating java collection.

1. ArrayList al=**new** ArrayList(); //creating old non-generic arraylist

Let's see the new generic example of creating java collection.

2. ArrayList<String> al=**new** ArrayList<String>(); //creating new generic arraylist

In generic collection, we specify the type in angular braces. Now ArrayList is forced to have only specified type of objects in it. If you try to add another type of object, it gives *compile time error*.

Vector

- Vector implements a dynamic array. It is similar to ArrayList, but with two differences:
  - **Vector is synchronized**
  - **Vector contains many legacy methods that are not part of the collections framework**
- Constructors

```
Vector( )
Vector(int size)
Vector(int size, int incr)
Vector(Collection c)
```

- Vector Class allows duplicate values
- It is similar to ArrayList, internally follows Resizable Array Structure

Important Methods	
boolean add (Object o)	Add element to the Vector
void add (int index, Object element)	Add element to the Vector at specified position
void clear ()	Clears elements
Object get (int index)	Returns element at specified index
boolean remove (Object o)	Removes element
int size()	Returns size
Void addElement()	Add element to the Vector
Object elementAt(int index)	Return the element at specified index
Enumeration elements()	Return an enumeration of element in vector
Object firstElement()	Return first element in the Vector
Object lastElement()	Return last element in the Vector
boolean removeAllElement()	Remove all element of the Vector

Program N0. 104

Vector Demo

```
import java.util.*;

public class VectDemo_104
{
	public static void main(String[] args)
	{
		Vector v = new Vector();
		v.add(10);
		v.add(23.43);
		v.add("Hello World");
		v.add(10);

		System.out.println("\n\t Elements = " + v);
		System.out.println("\n\t Using For :");
		for(int i=0; i<v.size(); i++)
		{
			System.out.print("\t" + v.get(i));
		}

		System.out.println("\n\t Using Iterator :");
		Iterator it = v.iterator();
		while(it.hasNext())
		{
			System.out.print("\t" + it.next());
```

```
            }


            System.out.println("\n\t Using ListIterator :");
            ListIterator lit = v.listIterator();
            while(lit.hasNext())
            {
                  System.out.print("\t" + lit.next());
            }

            System.out.println("\n\tDisplay elements reverse :");
            while(lit.hasPrevious())
            {
                  System.out.print("\t" + lit.previous());
            }
      }
}
```

Stack

- Stack is a subclass of Vector that implements a standard **last-in, first-out** stack.
- Stack only defines the default constructor, which creates an empty stack.

```
Stack( )
```

- Stack includes all the methods defined by Vector, and adds several of its own
- Stack Class is similar to Vector [It is Synchronized]
- It follows Stack Data Structure

SN	Methods with Description
1	**boolean empty()** Tests if this stack is empty. Returns true if the stack is empty, and returns false if the stack contains elements.
2	**Object peek( )** Returns the element on the top of the stack, but does not remove it.
3	**Object pop( )** Returns the element on the top of the stack, removing it in the process.
4	**Object push(Object element)** Pushes element onto the stack. element is also returned.
5	**int search(Object element)** Searches for element in the stack. If found, its offset from the top of the stack is returned. Otherwise, .1 is returned.

Program N0. 105

Stack Demo

```
import java.util.*;

public class StackDemo_105
{
	public static void main(String[] args)
	{
		Stack st = new Stack();
		st.push(10);
		st.push(34.32);
		st.push("Hello World");

		System.out.println("\n\t Elements = " + st);

		System.out.print("\n\t Element at Top = " +
		st.pop());

		System.out.println("\n\t Elements = " + st);
	}
}
```

Summary of List Implementations

Property	ArrayList	LinkedList	Vector	Stack
Ordered	Ordered by index	Ordered by index	Ordered by index	Ordered by Index
Null Values	Allowed	Allowed	Allowed	Allowed
Duplicate	Allowed	Allowed	Allowed	Allowed
Synchronized	No	No	Yes	Yes
Initial Capacity	10	Not Applicable	10	10
Data Structure	Resizable Array	Doubly Linked List	Resizable Array	Resizable Array

4 Mechanisms of Collections Iteration

Iteration is one of the basic operations carried on a collection. Basically, an iteration takes elements from a collection one after another, from the first element to the last one.

For example, you may want to go through all students in a class to print their names or to find who have high scores in the recent exam. Or you may want to traverse through a list of numbers to calculate the sum and average. Such kinds of operations are very common in programming.

The Java programming language provides four mechanisms for iterating over collections, including for loops, iterator and forEach (since Java 8).

Before going to each kind of iteration, suppose that we have a `List` collection as follows:

```
List<String> listNames = new ArrayList<>();
listNames.add("Tom");
listNames.add("Mary");
listNames.add("Peter");
listNames.add("John");
listNames.add("Kim");
```

This list contains names of all students in a class. Note that the diamond operator <> used in the right side of the assignment:

```
ArrayList<>();
```

This syntax can be used from Java 7, which allows us to declare generics collections in a more compact way, as the compiler can infer the parameter type in the right side from the left side (thus the so-called ***type inference***).

1. The Classic For Loop:

This iteration mechanism is very familiar in programming in which a counter variable runs from the first element to the last one in the collection. Here's the code that iterates over the `listNames` above:

```
for (int i = 0; i < listNames.size(); i++) {
        String aName = listNames.get(i);
        System.out.println(aName);
}
```

Here are the pros of this mechanism:

- This is the most familiar construct in programming.
- Useful if we need to access and use the counter variable, e.g. print the numeric order of the students: 1, 2, 3, etc.

And here are the cons:

- Using a counter variable requires the collection must store elements in form of index-based like `ArrayList`, and we must know the collection's size beforehand.
- The collection must provide a method to access its elements by index-based, which not supported by all collections, e.g. a `Set` does not store elements as index-based. Thus this mechanism cannot be used with all collections.

2. The Iterator Mechanism:

Due to the limitations of the classic for loop, the **Iterator** mechanism is created to allow us to iterate all kinds of collections. Thus you can see the **`Collection`** interface defines that every collection must implement the **`iterator`**`()` method.

Concept of the Iterator

An iterator is an object that enables us to traverse a collection. There is an iterator (**`java.util.Iterator`**) in all the top level interfaces of the Java Collections Framework that inherits java.util.Collection interface. These interfaces are **`java.util.List`**, **`java.util.Queue`**, **`java.util.Deque`**, and **`java.util.Set`**. Furthermore, there is the **`java.util.Map`** interface that does not inherit **`java.util.Collection`**.

Lists also have a special iterator called a list iterator (**`java.util.ListIterator`**). What's the difference? The **`java.util.Iterator`** is forward looking only while the **`java.util.ListIterator`** is bidirectional (forward and backward). Furthermore, the **`java.util.ListIterator`** inherits **`java.util.Iterator`**. The result of using either iterator to loop through a list will be the same as we will see later.

The following example explains the concept of iterator:

```
Iterator<String> iterator = listNames.iterator();
while (iterator.hasNext()) {
        String aName = iterator.next();
        System.out.println(aName);
}
```

This code snippet does the same thing as the classic for loop example above. You may need some explanations:

- The `hasNext()` method returns true if the collection has more elements to traverse, otherwise return false.
- The `next()` method returns the current element. Note that we don't have to cast the returned object as we use generics.

Here's another example that illustrates how to iterate over a `Set` using an Iterator:

```
Set<Integer> numbers = new HashSet<>();
numbers.add(100);
numbers.add(35);
numbers.add(89);
numbers.add(71);

Iterator<Integer> iterator = numbers.iterator();
while (iterator.hasNext()) {
        Integer aNumber = iterator.next();
        System.out.println(aNumber);
}
```

And here's another example demonstrating how to iterate over a `Map` using an iterator:

```
Map<Integer, String> mapAscii = new HashMap<>();
mapAscii.put(65, "A");
mapAscii.put(66, "B");
mapAscii.put(67, "C");
mapAscii.put(68, "D");
Iterator<Integer> keyIterator = mapAscii.keySet().iterator();
while (keyIterator.hasNext()) {
       Integer key = keyIterator.next();
       String value = mapAscii.get(key);
       System.out.println(key + " -> " + value);
}
```

Because the map stores elements in form of key=value pairs, first we need to get the iterator of the keys (a Set collection), then use this iterator to get each key, and retrieve the value corresponds to that key.

3. The Enhanced For Loop:

Since Java 5, programmers can use a more succinct syntax to iterate over a collection - It's the ***enhanced for loop***.

For example, the following code uses the enhanced for loop to iterate over the `listNames` collection above:

```
for (String aName : listNames) {
       System.out.println(aName);
```

```
}
```

The code is more compact and more readable. That's why this construct is called enhanced for loop - an enhanced feature of the Java programming language.

NOTE:

The enhanced for loop actually uses an iterator behind the scenes. That means the Java compiler will convert the enhanced for loop syntax to iterator construct when compiling. The new syntax just gives the programmers a more convenient way for iterating over collections.

Using the enhanced for loop, we can re-write the code to iterate the `Set` collection above like this:

```
for (Integer aNumber : numbers) {
	System.out.println(aNumber);
}
```

Compare to the previous code (using iterator), this code is incredible simpler and more understandable right?

And the code that iterates over a Map can be re-written using the enhanced for loop like this:

```
for (Integer key : mapAscii.keySet()) {
	String value = mapAscii.get(key);
	System.out.println(key + " -> " + value);
}
```

This looks much simpler than the previous code using iterator, right? Thanks to the enhanced for loop - it helps programmers write code more quickly and more readable.

As the Java programming language evolves, we have a new mechanism which is describe below.

4. The forEach Mechanism:

Java 8 with Lambda expressions, introduces a totally new way for iterating over collections - it's the ***forEach*** mechanism.

What's the biggest difference between the **forEach** mechanism and the previous ones?

Well, in the previous mechanisms (classic for loop, iterator and enhanced for loop), the programmers control how the collection is iterated. The iteration code is not part of the collection itself - it's written by programmers - hence the term ***external iteration***.

In contrast, the new mechanism encapsulates the iteration code in the collection itself, thus the programmers do not have to write code for iterating collections. Instead, the programmers specify what-to-do in each iteration - this is the big difference! Hence the term ***internal iteration***: the collections handle the iteration itself, whereas the programmers pass the action - what needs to do in each iteration.

The following example helps you understand the concepts:

```
listNames.forEach(name -> System.out.println(name));
```

Amazing! This code looks even more compact and more readable than the enhanced for loop version. As we can read the above line like this: for each element in the list Names, print the name to the console.

Since Java 8, each collection has a `forEach()` method that implements the iteration internally. Note that this method takes a Lambda expression or in other words, the programmers can pass their code - or function - into this method. As shown in the above example, the code to print each element is passed into the method.

What is Set?

Basically, **Set** is a type of collection that does not allow duplicate elements. That means an element can only exist once in a **Set**. It models the set abstraction in mathematics.

Characteristics of a Set collection:

The following characteristics differentiate a `Set` collection from others in the Java Collections framework:

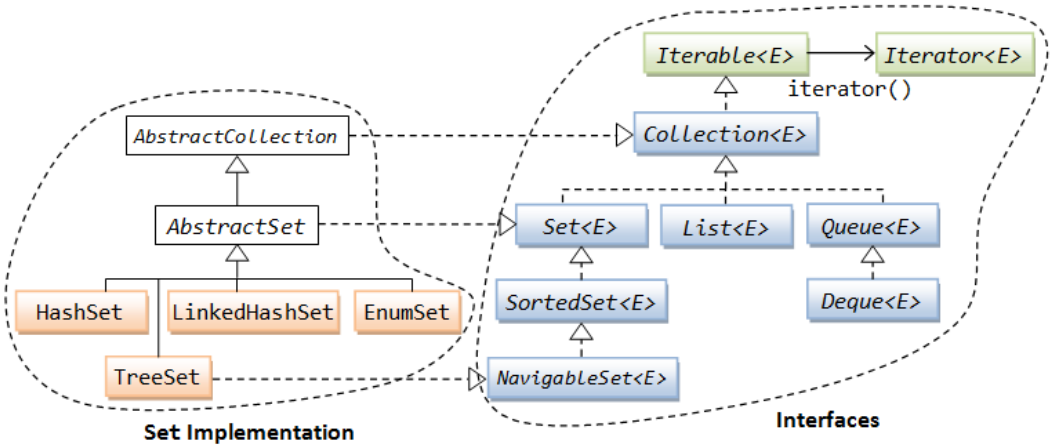
- Duplicate elements are not allowed.
- Elements are not stored in order. That means you cannot expect elements sorted in any order when iterating over elements of a **Set**.

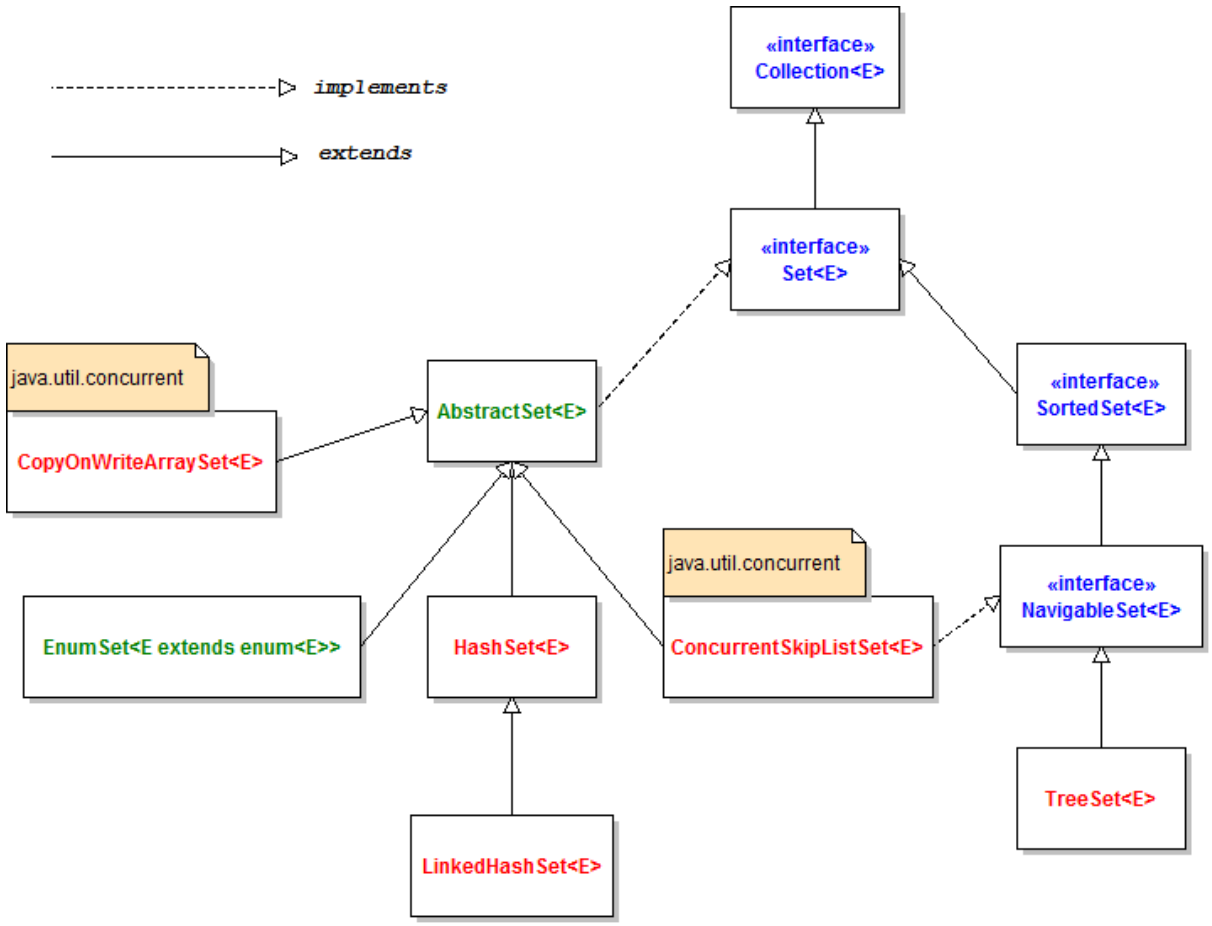
Why and When Use Sets?

Based on the characteristics, consider using a **Set** collection when:

- You want to store elements distinctly without duplication, or unique elements.
- You don't care about the order of elements.

For example, you can use a **Set** to store unique integer numbers; you can use a **Set** to store cards randomly in a card game; you can use a Set to store numbers in random order, etc.





Set Implementations:

The Java Collections Framework provides three major implementations of the Set interface: **`HashSet`**, **`LinkedHashSet`** and **`TreeSet`**.

HashSet

HashSet is the best-performing implementation and is a widely-used **`Set`** implementation. It represents the core characteristics of sets: no duplication and unordered.

- Uses hashtable to store the elements.
- Contains unique elements only.
- HashSet extends AbstractSet and implements the Set interface. It creates a collection that uses a hash table for storage.
- Elements are not stored in order. That means you cannot expect elements sorted in any order when iterating over elements of a **`Set`**
- A hash table stores information by using a mechanism called hashing. In hashing, the informational content of a key is used to determine a unique value, called its 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
- The first form constructs a default hash set:

```
HashSet( )
```

- The following constructor form initializes the hash set by using the elements of c

```
HashSet(Collection c)
```

- The following constructor form initializes the capacity of the hash set to capacity. The capacity grows automatically as elements are added to the Hash.

```
HashSet(int capacity)
```

SN	Methods with Description
1	**boolean add(Object o)** Adds the specified element to this set if it is not already present.
2	**void clear()** Removes all of the elements from this set.
4	**boolean contains(Object o)** Returns true if this set contains the specified element
5	**boolean isEmpty()** Returns true if this set contains no elements.
7	**boolean remove(Object o)** Removes the specified element from this set if it is present.
8	**int size()** Returns the number of elements in this set (its cardinality).

Program N0. 106

HashSet Demo

```
import java.util.*;

public class HashSetDemo_106
{
	public static void main(String[] args)
	{
		HashSet set1 = new HashSet();
		set1.add(99);
		set1.add(10);
		set1.add(543.34);
		set1.add("Hello World");
		set1.add(new Integer(23));
		//boolean result = set1.add(99);
		System.out.println(set1.add("Hello World"));
		System.out.println("\n\t Elements = " + set1);
	}
}
```

LinkedHashSet

This implementation orders its elements based on insertion order. So consider using a **`LinkedHashSet`** when you want to store unique elements in order.

- This class extends **HashSet**, but adds no members of its own.
- `LinkedHashSet` maintains a linked list of the entries in the set, in the order in which they were inserted
- 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.
- The LinkedHasSet internally follow hashing technique and DoubleLinkedList Structure
- The LinkedHashSet class supports following constructors

  `LinkedHashSet( )`

  `LinkedHashSet(Collection c)`
- Creation of LinkedHashSet

  `LinkedHashSet lset = new LinkedHashSet()`

  `LinkedHashSet<E> lset = new LinkedHashSet<E>();`

Program N0. 107

LinkedHashSet Demo

```
import java.util.*;

public class LinkedHashSetDemo_107
{
   public static void main(String[] args)
   {
       LinkedHashSet hs = new LinkedHashSet();
       hs.add(10);
       hs.add(43);
       hs.add("Hello World");
       hs.add(10);

       System.out.println("\n\t Elements = " + hs);
   }
}
```

TreeSet

This implementation orders its elements based on their values, either by their natural ordering, or by a **`Comparator`** provided at creation time.

- TreeSet provides an implementation of the Set interface that uses a tree for storage. Objects are stored in sorted, ascending order.
- Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.

Program N0. 108

Treeset Demo

```
import java.util.*;

public class TreeSetDemo_108
{
   public static void main(String[] args)
   {
       TreeSet ts = new TreeSet();
       ts.add(31);
```

```
        ts.add(10);
        ts.add(3);
        ts.add(21);

        System.out.println("\n\t Elements = " + ts);
    }
}
```

Therefore, besides the uniqueness of elements that a **Set** guarantees, consider using **HashSet** when ordering does not matter; using **LinkedHashSet** when you want to order elements by their insertion order; using **TreeSet** when you want to order elements by their values.

1. Creating a new Set

Always use generics to declare a **Set** of specific type, e.g. a **Set** of integer numbers:

```
Set<Integer> numbers = new HashSet<>();
```

Remember using the interface type (Set) on as the reference type, and concrete implementation (**HashSet**, **LinkedHashSet**, **TreeSet**, etc) as the actual object type:

```
Set<String> names = new LinkedHashSet<>();
```

We can create a Set from an existing collection. This is a trick to remove duplicate elements in non-Set collection. Consider the following code snippet:

```
List<Integer> listNumbers = Arrays.asList(3, 9, 1, 4, 7, 2, 5, 3, 8, 9, 1, 3, 8, 6);
System.out.println(listNumbers);
Set<Integer> uniqueNumbers = new HashSet<>(listNumbers);
System.out.println(uniqueNumbers);
```

Output:

```
[3, 9, 1, 4, 7, 2, 5, 3, 8, 9, 1, 3, 8, 6]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

You see, the list `listNumbers` contains duplicate numbers, and the set `uniqueNumbers` removes the duplicate ones.

As with Java 8, we cand use stream with filter and collection functions to return a `Set` from a collection. The following code collects only odd numbers to a `Set` from the `listNumbers` above:

```
Set<Integer> uniqueOddNumbers = listNumbers.stream().filter(number -> number
% 2 != 0).collect(Collectors.toSet());
System.out.println(uniqueOddNumbers);
```

Output:

```
[1, 3, 5, 7, 9]
```

Note that the default, initial capacity of a `HashSet` and `LinkedHashSet` is **16**, so if you are sure that your Set contains more than 16 elements, it's better to specify a capacity in the constructor. For example:

```
Set<String> bigNames = new HashSet<>(1000);
```

This creates a new `HashSet` with initial capacity is 1000 elements.

2. Performing Basic Operations on a Set

Adding elements to a Set:

The **add()** method returns true if the set does not contain the specified element, and returns false if the set already contains the specified element:

```
Set<String> names = new HashSet<>();
names.add("Tom");
names.add("Mary");
 if (names.add("Peter")) {
       System.out.println("Peter is added to the set");
}
if (!names.add("Tom")) {
       System.out.println("Tom is already added to the set");
}
```

Output:

```
Peter is added to the set
Tom is already added to the set
```

The Set can contain a null element:

```
names.add(null);
```

Removing an element from a Set:

The **remove(object)** method removes the specified element from the set if it is present (the method returns true, or false otherwise) :

```
if (names.remove("Mary")) {
          System.out.println("Marry is removed");
}
```

Note that the objects in the Set should implement the `equals()` and `hashCode()` methods correctly so the `Set` can find and remove the objects.

Check if a Set is empty:

The **isEmpty()** method returns true if the set contains no elements, otherwise returns false:

```
if (names.isEmpty()) {
          System.out.println("The set is empty");
} else {
          System.out.println("The set is not empty");
}
```

Remove all elements from a Set:

The **clear()** method removes all elements from the set. The set will be empty afterward:

```
names.clear();
if (names.isEmpty()) {
	System.out.println("The set is empty");
}
```

Get total number of elements in a Set:

The **size()** method returns the number of elements contained in the set:

```
Set<String> names = new HashSet<>();
names.add("Tom");
names.add("Mary");
names.add("Peter");
names.add("Alice");

System.out.printf("The set has %d elements", names.size());
```

Output:

```
The set has 4 elements
```

Note that the `Set` interface does not provide any API for retrieving a specific element due to its nature of unordered. Except the `TreeSet` implementation allows retrieving the first and the last elements.

3. Iterating over elements in a Set

Using an iterator:

```
Set<String> names = new HashSet<>();
names.add("Tom");
names.add("Mary");
names.add("Peter");
names.add("Alice");
Iterator<String> iterator = names.iterator();
while (iterator.hasNext()) {
		String name = iterator.next();
		System.out.println(name);
}
```

Output:

```
Tom
Alice
Peter
```

```
Mary
```

Using the enhanced for loop:

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

Using the **forEach()** method with Lambda expression in Java 8:

```
                names.forEach(System.out::println);
```

4. Searching for an element in a Set

The **contains(Object)** method returns true if the set contains the specified element, or return false otherwise. For example:

```
Set<String> names = new HashSet<>();
names.add("Tom");
names.add("Mary");
names.add("Peter");
names.add("Alice");
 if (names.contains("Mary")) {
          System.out.println("Found Mary");
}
```

Note that if the set contains custom objects of your own type, e.g. `Student` or `Employee`, the object should implement the `equals()` and `hashCode()` methods correctly so the `Set` can find the objects.

5. Performing Bulk Operations between two Sets

We can perform some mathematic-like operations between two sets such as subset, union, intersection and set difference. Suppose that we have two sets `s1` and `s2`.

Subset operation:

s1.containsAll(s2) returns true if s2 is a subset of s1 (s2 is a subset of s1 if s1 contains all of the elements in s2).

Example:

```
Set<Integer> s1 = new HashSet<>(Arrays.asList(20, 56, 89, 31, 8, 5));
Set<Integer> s2 = new HashSet<>(Arrays.asList(8, 89));

if (s1.containsAll(s2)) {
              System.out.println("s2 is a subset of s1");
}
```

Output:

```
s2 is a subset of s1
```

Union operation:

`s1.addAll(s2)` transforms `s1` into the **union** of `s1` and `s2`. (The union of two sets is the set containing all of the elements contained in either set.)

Example:

```
Set<Integer> s1 = new HashSet<>(Arrays.asList(1, 3, 5, 7, 9));
Set<Integer> s2 = new HashSet<>(Arrays.asList(2, 4, 6, 8));
System.out.println("s1 before union: " + s1);
s1.addAll(s2);
System.out.println("s1 after union: " + s1);
```

Output:

```
s1 before union: [1, 3, 5, 7, 9]
s1 after union: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Intersection operation:

`s1.retainAll(s2)` - transforms `s1` into the intersection of `s1` and `s2`. (The intersection of two sets is the set containing only the elements common to both sets.)

Example:

```
Set<Integer> s1 = new HashSet<>(Arrays.asList(1, 2, 3, 4, 5, 7, 9));
Set<Integer> s2 = new HashSet<>(Arrays.asList(2, 4, 6, 8));
System.out.println("s1 before intersection: " + s1);
s1.retainAll(s2);
System.out.println("s1 after intersection: " + s1);
```

Output:

```
s1 before intersection: [1, 2, 3, 4, 5, 7, 9]
s1 after intersection: [2, 4]
```

Set difference operation:

`s1.removeAll(s2)` — transforms `s1` into the (asymmetric) set difference of `s1` and `s2`. (For example, the set difference of `s1` minus `s2` is the set containing all of the elements found in `s1` but not in `s2`.)

Example:

```
Set<Integer> s1 = new HashSet<>(Arrays.asList(1, 2, 3, 4, 5, 7, 9));
Set<Integer> s2 = new HashSet<>(Arrays.asList(2, 4, 6, 8));
System.out.println("s1 before difference: " + s1);
s1.removeAll(s2);
System.out.println("s1 after difference: " + s1);
```

Output:

```
s1 before difference: [1, 2, 3, 4, 5, 7, 9]
s1 after difference: [1, 3, 5, 7, 9]
```

6. Concurrent Sets

All three implementations `HashSet`, `LinkedHashSet` and `TreeSet` are not synchronized. So if you use them in concurrent context (multi-threads), you have to synchronize them externally using **`Collections.synchronizedSet()`** static method. For example:

```
Set<Integer> numbers = Collections.synchronizedSet(new HashSet<Integer>());
```

The returned set is synchronized (thread-safe). And remember you must manually synchronize on the returned set when iterating over it:

```
synchronized (numbers) {
  Iterator<Integer> iterator = numbers.iterator();
  while (iterator.hasNext()) {
      Integer number = iterator.next();
      System.out.println(number);
      }
  }
```

Summary of Set Collection

Property	HashSet	LInkedHashSet	TreeSet
Ordered	Unordered	Ordered by insertion	Sorted order
Null Values	Allowed	Allowed	Allowed
Duplicate	Not Allowed	Not Allowed	Not Allowed
Synchronized	No	No	No
Initial Capacity	16	16	Not Applicable
Data Structure	HashTable	HashTable + Double Linked List	Balanced Tree

Equals() and hashCode()

When it comes to working with collections, we should override the `equals()`and `hashCode()` methods properly in the classes of the elements being added to the collections. Otherwise we will get unexpected behaviors or undesired results.

You know, the `Object` class (the super class of all classes in Java) defines two methods `equals()` and `hashCode()`. That means all classes in Java (including the ones you created) inherit these methods. Basically, the `Object` class implements these methods for general purpose so you may not see them frequently.

However, you will have to override them specifically for the classes whose objects are added to collections, especially the hashtable-based collections such as `HashSet` and `HashMap`.

Understanding the equals() method:

When comparing two objects together, Java calls their `equals()` method which returns true if the two objects are equal, or false otherwise. Note that this comparison using `equals()` method is very different than using the == operator.

Here's the difference:

The `equals()` method is designed to compare two objects semantically (by comparing the data members of the class), whereas the == operator compares two objects technically (by comparing their references i.e. memory addresses).

NOTE: The implementation of `equals()` method in the `Object` class compares references of two objects. That means we should override it in our classes for semantic comparison. Almost classes in the JDK override their own version of `equals()` method, such as `String`, `Date`, `Integer`, `Double`, etc.

A typical example is String comparison in Java. Let's see the following code:

```
String s1 = new String("This is a string");
String s2 = new String("This is a string");
boolean refEqual = (s1 == s2);
boolean secEqual = (s1.equals(s2));
System.out.println("s1 == s2: " + refEqual);
System.out.println("s1.equals(s2): " + secEqual);
```

Output:

```
s1 == s2: false
s1.equals(s2): true
```

You see, the reference comparison (== operator) returns false because `s1` and `s2`are two different objects which are stored in different locations in memory. Whereas the semantic comparison returns true because `s1` and `s2` has same value ("This is a string") which can be considered equal semantically.

Likewise, let say we have the `Student` class as following:

```
public class Student {
        private String id;
        private String name;
        private String email;
        private int age;

        public Student(String id, String name, String email, int age) {
                this.id = id;
                this.name = name;
                this.email = email;
                this.age = age;
        }
         public String toString() {
                String studentInfo = "Student " + id;
                studentInfo += ": " + name;
                studentInfo += " - " + email;
                studentInfo += " - " + age;

                return studentInfo;
        }
}
```

In practice, we can consider two `Student` objects are semantically equal if they have same attributes (id, name, email and age). Now, let's see how to override the `equals()` method in this class to confirm that two `Student` objects having identical attributes are considered to be equal:

```
public boolean equals(Object obj) {
        if (obj instanceof Student) {
```

```
                Student another = (Student) obj;
                if (this.id.equals(another.id) &&
                        this.name.equals(another.name) &&
                        this.email.equals(another.email) &&
                        this.age == another.age) {
                                return true;
                }
        }
         return false;
}
```

Here, this `equals()`method checks if the passed object is of type `Student` and if it has same attributes as the current object, they are considered to be equal (return true); otherwise they are not equal (return false). Let's test it out with the following code:

```
Student student1 = new Student("123", "Tom", "tom@gmail.com", 30);
Student student2 = new Student("123", "Tom", "tom@gmail.com", 30);
Student student3 = new Student("456", "Peter", "peter@gmail.com", 23);
System.out.println("student1 == student2: " + (student1 == student2));
System.out.println("student1.equals(student2): " + (student1.equals(student2)));
System.out.println("student2.equals(student3): " + (student2.equals(student3)));
```

And we have the following output:

```
student1 == student2: false
student1.equals(student2): true
student2.equals(student3): false
```

Let's see another example to understand how overriding the `equals()` method really helps. Suppose that we have a list of students like this:

```
List<Student> listStudents = new ArrayList<>();
This list contains the three Student objects above:
listStudents.add(student1);
listStudents.add(student2);
listStudents.add(student3);
```

Now we want to check whether the list contains a student with a given ID. I'll tell you an easy and interesting solution using `equals()` method.

Note that the `List` interface provides the `contains(Object)` method which can be used for checking if the specified object exists in the list. Behind the scene, the list invokes the `equals()` method on the search object to compare it with other objects in the collection.

And do you agree that two `Student` objects can be considered to be equal if they have same ID? So we update the `equals()` method in the `Student` class like this:

```
public boolean equals(Object obj) {
        if (obj instanceof Student) {
                Student another = (Student) obj;
```

```
                        if (this.id.equals(another.id)) {
                                        return true;
                        }
                }
                 return false;
        }
```

Here, this `equals()` method compares only the `id` attribute of two `Student` objects. And add another constructor to the `Student` class:

```
        public Student(String id) {
                this.id = id;
        }
```

Now, we can perform the checking like this:

```
        Student searchStudent1 = new Student("123");
        Student searchStudent4 = new Student("789");
        boolean found1 = listStudents.contains(searchStudent1);
        boolean found4 = listStudents.contains(searchStudent4);

        System.out.println("Found student1: " + found1);
        System.out.println("Found student4: " + found4);
```

Here's the result:

```
        Found student1: true
        Found student4: false
```

It's awesome, isn't it? Thanks to the `equals()` method which makes our code simple. Imagine if we don't use it, we would have implemented the search functionality more complex like this:

```
        public boolean searchStudent(List<Student> listStudents, String id) {
                for (Student student : listStudents) {
                        if (student.getId().equals(id)) {
                                return true;
                        }
                }
                return false;
        }
```

Understanding the hashCode() method:

The `Object` class defines the `hashCode()` method as follows:

```
        public int hashCode()
```

You can see this method returns an integer number. So where is it used?

Here's the secret:

This hash number is used by hashtable-based collections like `Hashtable`, `HashSet` and `HashMap` to store objects in small containers called "buckets". Each bucket is associated with a hash code, and each bucket contains only objects having identical hash code.

In other words, a hashtable groups its elements by their hash code values. This arrangement helps the hashtable locates an element quickly and efficiently by searching on small parts of the collection instead the whole collection.

Here are the steps to locate an element in a hashtable:

- Get hash code value of the specified element. This results in the `hashCode()` method to be invoked.
- Find the right bucket associated with that hash code.
- Inside the bucket, find the correct element by comparing the specified element with all the elements in the bucket. This results in the `equals()` method of the specified element to be invoked.

Having said that, when we add objects of a class to a hashtable-based collection (`HashSet`, `HashMap`), the class's `hashCode()` method is invoked to produce an integer number (which can be an arbitrary value). This number is used by the collection to store and locate the objects quickly and efficiently, as a hashtable-based collection does not maintain order of its elements.

NOTE: The default implementation of `hashCode()` in the `Object` class returns an integer number which is the memory address of the object. We should override it in our own classes. Almost classes in the JDK override their own version of `hashCode()` method, such as `String`, `Date`, `Integer`, `Double`, etc.

The Rules Between equals() and hashCode():

As explained above, as hashtable-based collection locates an element by invoking its `hashCode()` and `equals()` methods, so we must obey this contract with regard to the way we override these methods:

- When the `equals()` method is overridden, the `hashCode()` method must be overridden as well.
- If two objects are equal, their hash codes must be equal as well.
- If two objects are not equal, there's no constraint on their hash codes (their hash codes can be equal or not).
- If two objects have identical hash codes, there's no constraint on their equality (they can be equal or not).
- If two objects have different hash codes, they must not be equal.

By following these rules, we keep the collections consistent in maintaining its elements. If we violate these rules, the collections will behave unexpectedly such as the objects cannot be found, or wrong objects are returned instead of the correct ones.

Now, let's see how the `hashCode()` and `equals()` methods affect the behaviors of a `Set` by coming back to the student example.

Until now, we have the `Student` class written like this:

```
public class Student {
        private String id;
        private String name;
        private String email;
        private int age;

        public Student(String id) {
                this.id = id;
        }
        public Student(String id, String name, String email, int age) {
```

```
                this.id = id;
                this.name = name;
                this.email = email;
                this.age = age;
        }
        public String toString() {
                String studentInfo = "Student " + id;
                studentInfo += ": " + name;
                studentInfo += " - " + email;
                studentInfo += " - " + age;

                return studentInfo;
        }
        public boolean equals(Object obj) {
                if (obj instanceof Student) {
                        Student another = (Student) obj;
                        if (this.id.equals(another.id)) {
                                return true;
                        }
                }

                return false;
        }
}
```

Note that, there's only `equals()` method is overridden till now.

We add three `Student` objects to a `HashSet` as shown in the following code:

```
Student student1 = new Student("123", "Tom", "tom@gmail.com", 30);
Student student2 = new Student("123", "Tom", "tom@gmail.com", 30);
Student student3 = new Student("456", "Peter", "peter@gmail.com", 23);
Set<Student> setStudents = new HashSet<Student>();
setStudents.add(student1);
setStudents.add(student2);
setStudents.add(student3);
```

Now, let's print information of all students in this set using Lambda expressions:

```
setStudents.forEach(student -> System.out.println(student));
```

And we have the following output:

```
Student 456: Peter - peter@gmail.com - 23
```

```
Student 123: Tom - tom@gmail.com - 30
Student 123: Tom - tom@gmail.com - 30
```

Look, do you notice that there seems to be 2 duplicate students (ID: 123), right?

Oh, we would expect the set does not contain duplicate elements, why is this possible?

Here's the reason:

The set invokes the `equals()` and `hashCode()` methods on each object being added to ensure there's no duplication. In our case, the `Student` class overrides only the `equals()` method. And the `hashCode()` method inherited from the `Object` class returns memory addresses of each object which is not consistent with the `equals()` method (the contact is violated). Therefore the set treats the `student1` and `student2` object as two different elements.

Now, let's override the `hashCode()` method in the `Student` class to obey the contract of `equals()` and `hashCode()`. Here's the code needs to be added:

```
public int hashCode() {
        return 31 + id.hashCode();
}
```

This method returns an integer number based on the hash code of the id attribute (its `hashCode()` method is overridden by the `String` class). Run the code to print the set again and observe the result:

```
Student 123: Tom - tom@gmail.com - 30
Student 456: Peter - peter@gmail.com - 23
```

Awesome! The duplicate element is now removed, you see? That's exactly what we want.

With the `equals()` and `hashCode()` methods overridden properly, we can also perform search on the set like this:

```
Student searchStudent = new Student("456");
boolean found = setStudents.contains(searchStudent);
System.out.println("Found student: " + found);
```

Output:

```
Found student: true
```

For more experiments yourself, try to remove either the `equals()` or `hashCode()` method and observe the outcome.

Vishal Shah, I hope the above explanation and examples help you understand how the `equals()` and `hashCode()` methods work, and why they play important roles with regard to collections.

There's more tips about implementing `equals()` and `hashCode()` methods correctly and efficiently, which you can find in the following article:

http://www.javaranch.com/journal/2002/10/equalhash.html

Sorting Collection

To understand object ordering properly, let's see some examples where we use the utility class `Collections` to sort elements of a collection (or `Arrays` class to sort elements in an array):

- `Collections.sort(list)`: sorts a List collection.
- `Arrays.sort(array)`: sorts an array.

Example #1: Sorting a list of String objects

```
List<String> names = Arrays.asList(
                                "Tom", "Peter", "Alice", "Bob", "Sam",
                                "Mary", "Jane", "Bill", "Tim", "Kevin");
System.out.println("Before sorting: " + names);
Collections.sort(names);
System.out.println("After sorting: " + names);
```

Output:

```
Before sorting: [Tom, Peter, Alice, Bob, Sam, Mary, Jane, Bill, Tim, Kevin]
After sorting: [Alice, Bill, Bob, Jane, Kevin, Mary, Peter, Sam, Tim, Tom]
```

In this example, the list names is sorted by alphabetic order of `String`.

Example #2: Sorting a list of Integer objects

```
List<Integer> numbers = Arrays.asList(8, 2, 5, 1, 3, 4, 9, 6, 7, 10);
System.out.println("Before sorting: " + numbers);
Collections.sort(numbers);
System.out.println("After sorting: " + numbers);
```

Output:

```
Before sorting: [8, 2, 5, 1, 3, 4, 9, 6, 7, 10]
After sorting: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Here, the integer numbers in the list numbers are sorted by alphanumeric order.

Example #3: Sorting a list of Date objects

```
DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");

List<Date> birthdays = new ArrayList<>();
birthdays.add(dateFormat.parse("2016-01-20"));
birthdays.add(dateFormat.parse("1998-12-03"));
birthdays.add(dateFormat.parse("2009-07-15"));
birthdays.add(dateFormat.parse("2012-04-30"));
System.out.println("Before sorting: ");
for (Date date : birthdays) {
          System.out.println(dateFormat.format(date));
}

Collections.sort(birthdays);

System.out.println("After sorting: ");
```

```
for (Date date : birthdays) {
        System.out.println(dateFormat.format(date));
}
```

Output:

```
Before sorting:
2016-01-20
1998-12-03
2009-07-15
2012-04-30

After sorting:
1998-12-03
2009-07-15
2012-04-30
2016-01-20
```

Here, the list birthdays is sorted by chronological order of its elements - objects of type `Date`.

From the 3 examples above, the collections are sorted by natural ordering of its elements:

- The natural ordering of `String` objects is alphabetic order.
- The natural ordering of `Integer` objects is alphanumeric order.
- The natural ordering of `Date` objects is chronological order.

Understanding Natural Ordering

Natural ordering is the default ordering of objects of a specific type when they are sorted in an array or a collection. The Java language provides the **`Comparable`** interface that allows us define the natural ordering of a class. This interface is declared as follows:

```
public interface Comparable<T> {
        public int compareTo(T object);
}
```

As you can see, this interface is parameterized (generics) and it has a single method `compareTo()` that allows two objects of a same type to be compared with each other. The important point here is the value returned by this method: an integer number indicates the comparison result of two objects. Remember these rules:

- Compare value = 0: two objects are equal.
- Compare value > 0: the first object (the current object) is greater than the second one.
- Compare value < 0: the first object is less than the second one.

Imagine that, when the objects are being sorted, their `compareTo()` methods are invoked to compare with other objects. And based on the compare value returned, the objects are sorted by natural ordering.

Classes whose objects used in collections or arrays should implement the `Comparable` interface for providing the natural ordering of its objects when being sorted. Otherwise we will get an error at runtime.

A class that implements the `Comparable` interface is said to have **class natural ordering.** And the `compareTo()` method is called the **natural comparison method**.

In the above examples, we don't have to write code to implement the `Comparable` interface because the `String`, `Integer` and `Date` classes already implemented this interface. Hence we can sort a collection containing objects of these types.

Other wrapper types in Java are also comparable: `Long`, `Double`, `Float`, etc.

When we create our own type, we have to implement the `Comparable` interface in order to have objects of our type eligible to be sorted in collections or arrays. Let's see an example to understand how the `Comparable` interface is used.

Let's say we have the `Employee` class which is defined as shown below:

```
public class Employee {
	String firstName;
	String lastName;
	Date joinDate;

	public Employee(String firstName, String lastName) {
		this.firstName = firstName;
		this.lastName = lastName;
	}

	public String toString() {
		return firstName + " " + lastName;
	}
	// getters and setters
}
```

Add some employees to a list collection like this:

```
List<Employee> listEmployees = new ArrayList<>();

Employee employee1 = new Employee("Tom", "Eagar");
Employee employee2 = new Employee("Tom", "Smith");
Employee employee3 = new Employee("Bill", "Joy");
Employee employee4 = new Employee("Bill", "Gates");
Employee employee5 = new Employee("Alice", "Wooden");

listEmployees.add(employee1);
listEmployees.add(employee2);
listEmployees.add(employee3);
listEmployees.add(employee4);
listEmployees.add(employee5);
```

Try to sort this list:

```
Collections.sort(listEmployees);
```

We will get an error at runtime: **no suitable method found for sort(List<Emmployee>)…**

WHY?

It's because the `Employee` class doesn't implement the `Comparable` interface so the `sort()` method cannot compare the objects.

Now, let's have the `Employee` class implements the `Comparable` interface, and we define the natural ordering is first name - last name, meaning the employees are sorted by first name first, then by last name. Here's the updated version of the `Employee` class:

```
public class Employee implements Comparable<Employee> {
        // fields...
        // constructors...
        // getters...
        // setters...

        // implement the natural comparison method:

    public int compareTo(Employee another) {
        int compareValue = this.firstName.compareTo(another.firstName);
        if (compareValue == 0) {
            return this.lastName.compareTo(another.lastName);
        }
        return compareValue;
    }
}
```

Look at how the `compareTo()` method is implemented here:

- First, we compare the first name by using the String's `compareTo()` method. We can safely use this method of the built-in types in Java: `String`, `Date`, `Integer`, `Long`, etc.
- If two employees have same first name (compare value = 0), then we compare their last name. Finally the compare value is returned as per the contract of the `Comparable` interface.

Now, run this test code and observe the result:

```
System.out.println("Before sorting: " + listEmployees);
Collections.sort(listEmployees);
System.out.println("After sorting: " + listEmployees);
```

Output:

```
Before sorting: [Tom Eagar, Tom Smith, Bill Joy, Bill Gates, Alice Wooden]
After sorting: [Alice Wooden, Bill Gates, Bill Joy, Tom Eagar, Tom Smith]
```

Awesome! It works perfectly as we expected: the employees are sorted by their first name, and then last name.

Note #1:

We cannot compare objects of different types, e.g. a `String` object cannot be compared with an Integer object. As the `compareTo()` method enforces this rule, we can only compare objects of the same type. If we add objects of different types to a collection and sort it, we will get `ClassCastException`.

Note #2:

If we want to reverse the natural ordering, simply swap the objects being compared in the `compareTo()` method. For example, the following implementation sorts employees by their first name into descending order:

```
public int compareTo(Employee another) {
        return another.firstName.compareTo(this.firstName);
}
```

In case we use a sorted collection i.e. `TreeSet`, we don't have to use the `Collections.sort()` utility method, as a `TreeSet` sorts its elements by their natural ordering. The following example demonstrates how to use a `TreeSet` to sort Strings:

```
Set<String> setNames = new TreeSet<>();
setNames.addAll(Arrays.asList("Tom", "Peter", "Alice", "Bob", "Sam",
                                    "Mary", "Jane", "Bill", "Tim", "Kevin"));
System.out.println(setNames);
```

Output:

```
[Alice, Bill, Bob, Jane, Kevin, Mary, Peter, Sam, Tim, Tom]
```

Similarly, we can sort the `Employee` objects using a `TreeSet` like this:

```
Set<Employee> setEmployees = new TreeSet<>();

Employee employee1 = new Employee("Tom", "Eagar");
Employee employee2 = new Employee("Tom", "Smith");
Employee employee3 = new Employee("Bill", "Joy");
Employee employee4 = new Employee("Bill", "Gates");
Employee employee5 = new Employee("Alice", "Wooden");

setEmployees.add(employee1);
setEmployees.add(employee2);
setEmployees.add(employee3);
setEmployees.add(employee4);
setEmployees.add(employee5);

System.out.println(setEmployees);
```

Output:

```
[Alice Wooden, Bill Gates, Bill Joy, Tom Eagar, Tom Smith]
```

So far we have got understanding about the natural ordering of objects and how the `Comparable` interface defines the ordering.

What if we want to sort objects in an order which differs from the natural ordering? For example, sort the employees list above by seniority (based on their join dates)?

Understanding Comparator

The `Collections` utility class provides a method for sorting a list using an external comparator:

```
Collections.sort(list, comparator)
```

This overloaded version takes two parameters: a list collection and a comparator, which is any object that implements the `Comparator` interface. This interface declares this method:

```
public interface Comparator<T> {
        public int compare(T obj1, T obj2);
}
```

Like the `Comparable` interface, this interface is also parameterized for any specific type. The `compare()` method is similar except it takes both the objects to be compared as arguments. The return value is also evaluated similarly.

For example, the following class compares two `Employee` objects using the `Comparator` interface:

```
public class EmployeeComparator implements Comparator<Employee> {
          public int compare(Employee emp1, Employee emp2) {
            return emp1.getJoinDate().compareTo(emp2.getJoinDate());
          }
}
```

In this comparator, we compare two `Employee` objects by their join dates. And update the `Employee` class like this (add an overloaded constructor and update the `toString()` method):

```
public class Employee implements Comparable<Employee> {
          // fields...
          // getters & setters....
          // constructor
          public Employee(String firstName, String lastName, Date joinDate) {
                  this.firstName = firstName;
                  this.lastName = lastName;
                  this.joinDate = joinDate;
          }
           public String toString() {
             DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
                   return firstName + " " + lastName + " " +
                            dateFormat.format(joinDate);
             }
}
```

And here's the test code:

```
DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");


List<Employee> listEmployees = new ArrayList<>();
```

```
Employee employee1 = new Employee("Tom", "Eagar", dateFormat.parse("2007-12-03"));
Employee employee2 = new Employee("Tom", "Smith", dateFormat.parse("2005-06-20"));
Employee employee3 = new Employee("Bill", "Joy", dateFormat.parse("2009-01-31"));
Employee employee4 = new Employee("Bill", "Gates", dateFormat.parse("2005-05-12"));
Employee employee5 = new Employee("Alice", "Wooden", dateFormat.parse("2009-01-
22"));

listEmployees.add(employee1);
listEmployees.add(employee2);
listEmployees.add(employee3);
listEmployees.add(employee4);
listEmployees.add(employee5);

System.out.println("Before sorting: ");
System.out.println(listEmployees);

Collections.sort(listEmployees, new EmployeeComparator());

System.out.println("After sorting: ");
System.out.println(listEmployees);

Collections.sort(listEmployees, (emp1, emp2) ->
emp1.getJoinDate().compareTo(emp2.getJoinDate()));
```

Output:

```
Before sorting:
[Tom Eagar 2007-12-03, Tom Smith 2005-06-20, Bill Joy 2009-01-31, Bill Gates 2005-
05-12, Alice Wooden 2009-01-22]

After sorting:
[Bill Gates 2005-05-12, Tom Smith 2005-06-20, Tom Eagar 2007-12-03, Alice Wooden
2009-01-22, Bill Joy 2009-01-31]
```

Note #3:

Since Java 8, we can use Lambda expressions to create a comparator more easily like this:

```
Collections.sort(listEmployees,
            (emp1, emp2) -> emp1.getJoinDate().compareTo(emp2.getJoinDate()));
```

We can also pass a comparator when creating a new instance of a `TreeSet` like this:

```
Set<Employee> setEmployees = new TreeSet<>(new EmployeeComparator());
```

Then the TreeSet will sort its elements according to the order defined by the specified comparator.

Using a comparator is useful in the following scenarios:

- The class doesn't have natural ordering (or we don't have source code to update it).
- We want to sort objects in orders other than the natural ordering.
- We want to provide multiple ways for sorting the objects, e.g. one comparator for each sorting criteria.

The constraint between natural ordering and equals

Now, let's discuss about the constraint between natural ordering and equals() method.

You know, the documentation of both Comparable and Comparator states that the natural ordering and the ordering specified by a comparator should be consistent with the equals() method of the class. Let's say we have two objects obj1 and obj2 of class A, then:

```
If obj1.compareTo(obj2) = 0 then obj1.equals(obj2) = true
```

If this contract is violated, we will get strange behavior when using sorted collections such as TreeSet and TreeMap.

Let's examine an example to understand why this constraint really matters. Come back to the example of sorting a list of Employee objects I provided in the previous email.

We haven't overridden the equals() method yet. Now, let's override it for the Employee class:

```
public class Employee implements Comparable<Employee> {
            // fields, constructors, getters and setters and toString()...
      public int compareTo(Employee another) {
            int compareValue = this.firstName.compareTo(another.firstName);
            if (compareValue == 0) {
                  return this.lastName.compareTo(another.lastName);
            }
            return compareValue;
      }

      public boolean equals(Object obj) {
            if (obj instanceof Employee) {
                  Employee another = (Employee) obj;
                  if (this.firstName.equals(another.firstName)
                        && this.lastName.equals(another.lastName))
                  {
                       return true;
                  }
            }
            return false;
      }
}
```

Currently, it is compatible with the compareTo() method which also compares first name and then last name.

What if we need to change the compareTo() method for comparing two Employee objects by their seniority (join date) like this:

```
public int compareTo(Employee another) {
```

```
            return this.joinDate.compareTo(another.joinDate);
    }
```

Let's execute some test code to see the outcome:

```
    DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
    Set<Employee2> setEmployees = new TreeSet<>(new EmployeeComparator2());
    Employee2 employee1 = new Employee2("Tom", "Eagar", dateFormat.parse("2007-12-
    03"));
    Employee2 employee2 = new Employee2("Tom", "Smith", dateFormat.parse("2005-06-
    20"));
    Employee2 employee3 = new Employee2("Bill", "Joy", dateFormat.parse("2007-12-03"));
    Employee2 employee4 = new Employee2("Bill", "Gates", dateFormat.parse("2005-05-
    12"));
    Employee2 employee5 = new Employee2("Alice", "Wooden", dateFormat.parse("2005-06-
    20"));

    setEmployees.add(employee1);
    setEmployees.add(employee2);
    setEmployees.add(employee3);
    setEmployees.add(employee4);
    setEmployees.add(employee5);
    System.out.println(setEmployees);
```

Note that the `employee1` and `employe5` have same join date, so do the `employee3` and `employee4`. Add all of these 5 objects to the set:

```
    setEmployees.add(employee1);
    setEmployees.add(employee2);
    setEmployees.add(employee3);
    setEmployees.add(employee4);
    setEmployees.add(employee5);
```

And print the set:

```
    System.out.println(setEmployees);
```

Can you guess the output? Here is it:

```
    [Tom Smith 2005-06-20, Tom Eagar 2007-12-03, Bill Joy 2009-01-31]
```

Ouch! Why are there only 3 employees in the set?

It's because the set compares the objects using the `compareTo()` method which considers two employees are equal if they have same join date, whereas the set does not allow duplicate elements, hence the `employee4` and `employee5` objects are not added to the set.

Now, you understand the consequence if natural ordering and equals are not consistent, right?

So is there any solution or workaround?

Suppose that we still want to keep the natural ordering based on join date, while keep compatible with the `equals()` method, here's how we update the `compareTo()` method:

That's it! In this solution, we compare the `Employee` objects by their join dates first. If equal, continue comparing by their first names. And if equal, continue comparing their last names. This way we can keep the `compareTo()` method compatible with the `equals()` method.

Run the test code again and observe the output:

```
[Tom Smith 2005-06-20, Alice Wooden 2007-12-03, Tom Eagar 2007-12-03, Bill Gates
2009-01-31, Bill Joy 2009-01-31]
```

The same problem and solution applies for a comparator.