**Collections**

* A collection — sometimes called a container — is simply an object that groups multiple elements into a single unit.
* Collections are used to store, retrieve, manipulate, and communicate aggregate data
* A collections framework is a unified architecture for representing and manipulating collections.

All collections frameworks contain the following:

**Interfaces**: These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.

**Implementations**: These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.

**Algorithms**: These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

**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.



The core collection interfaces.

A Set is a special kind of Collection, a SortedSet is a special kind of Set, and so forth. Note also that the hierarchy consists of two distinct trees — a Map is not a true Collection.

Note that all the core collection interfaces are generic. For example, this is the declaration of the Collection interface.

public interface Collection<E>

The <E> syntax tells you that the interface is generic. When you declare a Collection instance you can and should specify the type of object contained in the collection.

The following code snippet without generics requires casting:

List list = new ArrayList();

list.add("hello");

String s = (String) list.get(0);

When re-written to use generics, the code does not require casting:

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

list.add("hello");

String s = list.get(0); // no cast

By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

The following list describes the core collection interfaces:

* Collection - the root of the collection hierarchy. A collection represents a group of objects known as its elements. 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.
* Set - a collection that cannot contain duplicate elements.
* List - an ordered collection (sometimes called a sequence). Lists can contain duplicate elements.
* Queue - a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations.
* Deque - a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Deque provides additional insertion, extraction, and inspection operations.
* Map - an object that maps keys to values. A Map cannot contain duplicate keys; each key can map to at most one value.

Collection — the root of the collection hierarchy. A collection represents a group of objects known as its elements. 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.

The Collection interface contains methods that perform basic operations, such as int size(), boolean isEmpty(), boolean contains(Object element), boolean add(E element), boolean remove(Object element), and Iterator<E> iterator().

It also contains methods that operate on entire collections, such as boolean containsAll(Collection<?> c), boolean addAll(Collection<? extends E> c), boolean removeAll(Collection<?> c), boolean retainAll(Collection<?> c), and void clear().

Traversing Collections

1) for-each construct

for-each construct allows you to concisely traverse a collection or array using a for loop. The following code uses for-each construct to print out each element of a collection on a separate line.

for (Object o : collection)

System.out.println(o);

for (datatype counter\_name: collection\_variable\_name)

System.out.println(counter\_name);

e.g.

**class** EnhancedForDemo {

**public** **static** **void** main(String[] args) {

**int**[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// using collection

**for** (**int** item : numbers) {

System.***out***.println("Count is: " + item);

}

System.***out***.println();

// using array length

**for** (**int** i = 0; i < numbers.length; i++) {

System.***out***.println("Count is: " + numbers[i]);

}

}

}

2) Using Iterators.

An Iterator is an object that enables you to traverse through a collection. You get an Iterator for a collection by calling its iterator method.

Module java.base

Package java.util

Interface Iterator<E>

Type Parameters:

E - the type of elements returned by this iterator

Use Iterator instead of the for-each construct when you need to:

* Remove the current element. The for-each construct hides the iterator, so you cannot call remove. Therefore, the for-each construct is not usable for filtering.
* Perform operations of Interface object like Set etc. Below is an example

**import** java.awt.AWTException;

**import** java.awt.event.KeyEvent;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.Set;

**import** java.util.SortedSet;

**import** org.openqa.selenium.By;

**import** org.openqa.selenium.Keys;

**import** org.openqa.selenium.WebDriver;

**import** org.openqa.selenium.chrome.ChromeDriver;

**import** org.openqa.selenium.firefox.FirefoxDriver;

**import** org.openqa.selenium.interactions.Actions;

**import** com.sun.glass.ui.Robot;

**public** **class** Multiple\_Windows {

**public** **static** **void** main(String[] args) **throws** InterruptedException, AWTException {

WebDriver driver = **new** FirefoxDriver();

Thread.*sleep*(3000);

// Using below two lines does not add new tab in browser as mentioned in some of

// google search pages

// driver.findElement(By.xpath("//body")).sendKeys(Keys.CONTROL+"t");

// driver.findElement(By.cssSelector("body")).sendKeys(Keys.CONTROL +"t");

// Using below two lines does not add new tab in browser as mentioned in some of

// google search pages

// String selectLinkOpeninNewTab = Keys.chord(Keys.CONTROL,"t");

// driver.findElement(By.linkText("urlLink")).sendKeys(selectLinkOpeninNewTab);

// Using below two lines does not add new tab in browser as mentioned in some of

// google search pages

// Actions a = new Actions(driver);

// a.keyDown(Keys.CONTROL).sendKeys("t").keyUp(Keys.CONTROL).build().perform();

// Robot class is used for Key press and Mouse movement

java.awt.Robot robot1 = **new** java.awt.Robot();

// press Ctrl + t

robot1.keyPress(KeyEvent.***VK\_CONTROL***);

robot1.keyPress(KeyEvent.***VK\_T***);

// Release t followed by control key

robot1.keyRelease(KeyEvent.***VK\_T***);

robot1.keyRelease(KeyEvent.***VK\_CONTROL***);

// Add sleep as it takes time to add a tab in browser

Thread.*sleep*(3000);

// Add another using Robot class

java.awt.Robot robot2 = **new** java.awt.Robot();

robot2.keyPress(KeyEvent.***VK\_CONTROL***);

robot2.keyPress(KeyEvent.***VK\_T***);

robot2.keyRelease(KeyEvent.***VK\_T***);

robot2.keyRelease(KeyEvent.***VK\_CONTROL***);

// Add sleep as it takes time to add a tab in browser

Thread.*sleep*(3000);

// Get all window handles i.e. handles for all available tabs

Set<String> s = driver.getWindowHandles();

// an Iterator object is created as we want to traverse through the Set object i.e. s

// We can also typecase Set s to ArrayList or SortedSet to avoid creating an iterator object and doing operations

// ArrayList s1 = (ArrayList) driver.getWindowHandles();

// SortedSet s2 = (SortedSet) driver.getWindowHandles();

Iterator<String> i = s.iterator();

**while** (i.hasNext()) {

String windowname = i.next();

// Use to move focus to a particular tab

driver.switchTo().window(windowname);

// Add sleep as it takes time to move focus to another tab in browser

Thread.*sleep*(2000);

driver.navigate().to("http://zeenews.india.com/");

// Add sleep as it may take time to load web site hence window title and other parameters could be blank

Thread.*sleep*(3000);

System.***out***.println("Window handle = " + windowname);

System.***out***.println(driver.getTitle());

System.***out***.println();

}

}

}

Collection Interface Bulk Operations

Bulk operations perform an operation on an entire Collection.

* containsAll — returns true if the target Collection contains all of the elements in the specified Collection.
* addAll — adds all of the elements in the specified Collection to the target Collection.
* removeAll — removes from the target Collection all of its elements that are also contained in the specified Collection.
* retainAll — removes from the target Collection all its elements that are not also contained in the specified Collection. That is, it retains only those elements in the target Collection that are also contained in the specified Collection.
* clear — removes all elements from the Collection.

The addAll, removeAll, and retainAll methods all return true if the target Collection was modified in the process of executing the operation.

Below example uses ArrayList to demonstrate functioning of above methods

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Simple {

**public** **static** **void** main(String args[]) {

List<String> list1 = **new** ArrayList<String>();

List<String> list2 = **new** ArrayList<String>();

list1.add("Hi1");

list1.add("Hi2");

list1.add("How are");

list1.add("you");

list2.add("Hi1");

list2.add("Hi2");

list2.add("How are");

list2.add("you");

**for** (**int** i = 0; i < list1.size(); i++) {

System.***out***.println(list1.get(i));

}

**for** (**int** i = 0; i < list2.size(); i++) {

System.***out***.println(list2.get(i));

}

System.***out***.println();

// returns true as Hi1 is present in list1

System.***out***.println(list1.contains("Hi1"));

// returns true as HI1 is not present in list1

System.***out***.println(list1.contains("HI1"));

// returns true as list1 has same objects as list2

System.***out***.println(list2.containsAll(list1));

// adds objects from list1 to list2

list2.addAll(list1);

System.***out***.println();

System.***out***.println("after addall");

System.***out***.println("List 2:");

**for** (**int** i = 0; i < list2.size(); i++) {

System.***out***.println(list2.get(i));

}

list2.add("i am doing good");

list2.removeAll(list1);

System.***out***.println();

System.***out***.println("after removeall");

System.***out***.println("List 2:");

**for** (**int** i = 0; i < list2.size(); i++) {

System.***out***.println(list2.get(i));

}

list1.clear();

list2.clear();

System.***out***.println();

System.***out***.println("After clearing");

System.***out***.println("List 1:");

**for** (**int** i = 0; i < list1.size(); i++) {

System.***out***.println(list1.get(i));

}

System.***out***.println("List 2:");

**for** (**int** i = 0; i < list2.size(); i++) {

System.***out***.println(list2.get(i));

}

list1.add("1");

list1.add("2");

list1.add("3");

list1.add("4");

list1.add("5");

list2.add("3");

list2.add("4");

list2.add("5");

list2.add("6");

list2.add("7");

//code for showing how to use toArray()

Object[] str\_array = list1.toArray();

**for** (**int** i = 0; i < str\_array.length; i++) {

System.***out***.println("str\_array " + i + str\_array[i].toString());

}

// it retains only those elements in the target Collection i.e. 'list1' that are also contained in the specified Collection i.e. 'list2'.

list1.retainAll(list2);

System.***out***.println();

System.***out***.println("After retainall");

System.***out***.println("List 1:");

**for** (**int** i = 0; i < list1.size(); i++) {

System.***out***.println(list1.get(i));

}

}

}

The toArray methods are provided as a bridge between collections and older APIs that expect arrays on input. The array operations allow the contents of a Collection to be translated into an array. The simple form with no arguments creates a new array of Object. The more complex form allows the caller to provide an array or to choose the runtime type of the output array.

For example, suppose that c is a Collection. The following snippet dumps the contents of c into a newly allocated array of Object whose length is identical to the number of elements in c.

Object[] a = c.toArray();

Suppose that c is known to contain only strings (perhaps because c is of type Collection<String>). The following snippet dumps the contents of c into a newly allocated array of String whose length is identical to the number of elements in c.

String[] a = c.toArray(new String[0]);

**Set — a collection that cannot contain duplicate elements.**

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.

The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited. Two Set instances are equal if they contain the same elements.

There are three general-purpose Set implementations:

1. HashSet - which stores its elements in a hash table, is the best-performing implementation; however it makes no guarantees concerning the order of iteration
2. TreeSet – which stores and orders its elements based on their values; it is substantially slower than HashSet
3. LinkedHashSet – stores and orders its elements based on the order in which they were inserted into the set (insertion-order). It is slower than Hashset

Suppose you have a Collection, c, and you want to create another Collection containing the same elements but with all duplicates eliminated. The following one-liner does the trick.

Collection<Type> noDups = new HashSet<Type>(c);

See example below

**import** java.util.ArrayList;

**import** java.util.Collection;

**import** java.util.Comparator;

**import** java.util.HashSet;

**import** java.util.Iterator;

**import** java.util.List;

**import** java.util.ListIterator;

**import** java.util.stream.Collectors;

**public** **class** Simple {

**public** **static** **void** main(String args[]) {

List<String> list1 = **new** ArrayList<String>();

List<String> list2 = **new** ArrayList<String>();

list1.add("1");

list1.add("200");

list1.add("13");

list1.add("32");

list1.add("4");

list1.add("4");

list1.add("5");

System.***out***.println("List1 before sort:");

System.***out***.println("List elements are:");

**for** (**int** i = 0; i < list1.size(); i++) {

System.***out***.println(list1.get(i));

}

System.***out***.println("List1 after sort:");

list1.sort(Comparator.*reverseOrder*());

System.***out***.println("List elements are:");

**for** (**int** i = 0; i < list1.size(); i++) {

System.***out***.println(list1.get(i));

}

System.***out***.println();

// One way of converting list to HashSet

System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*");

HashSet<String> h = **new** HashSet<>();

h.addAll(list1);

Iterator<String> ite1 = h.iterator();

**while**(ite1.hasNext()) {

System.***out***.println(ite1.next());

}

// Another way of converting list to HashSet

System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*");

Collection<String> noDups = **new** HashSet<String>(list1);

Iterator<String> ite2 = noDups.iterator();

**while**(ite2.hasNext()) {

System.***out***.println(ite2.next());

}

}

}

**Set Interface Basic Operations**

* size: operation returns the number of elements in the Set (its cardinality)
* isEmpty: checks whether the set is empty and returns a boolean value
* add: adds the specified element to the Set if it is not already present and returns a boolean indicating whether the element was added.
* remove: removes the specified element from the Set if it is present and returns a boolean indicating whether the element was present.
* Iterator: returns an Iterator over the Set

Set Interface Bulk Operations

Bulk operations are particularly well suited to Sets; when applied, they perform standard set-algebraic operations. Suppose s1 and s2 are sets. Here's what bulk operations do:

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

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.)

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.)

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.)

To calculate the union, intersection, or set difference of two sets nondestructively (without modifying either set), the caller must copy one set before calling the appropriate bulk operation. The following are the resulting idioms.

Set<Type> union = new HashSet<Type>(s1);

union.addAll(s2);

Set<Type> intersection = new HashSet<Type>(s1);

intersection.retainAll(s2);

Set<Type> difference = new HashSet<Type>(s1);

difference.removeAll(s2);

**Howework: Create 2 Set variables i.e. S1 with elements 1,2,3,4,5 and S2 with elements 3,4,5,6,7. Find Union, Intersection and Difference of these two sets**

List — an ordered collection (sometimes called a sequence). Lists can contain duplicate elements. The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position).

The List interface includes operations for the following:

**Positional access** — manipulates elements based on their numerical position in the list. This includes methods such as get, set, add, addAll, and remove.

**Search** — searches for a specified object in the list and returns its numerical position. Search methods include indexOf and lastIndexOf.

**Iteration** — extends Iterator semantics to take advantage of the list's sequential nature. The listIterator methods provide this behavior.

**Range-view** — The sublist method performs arbitrary range operations on the list.

The Java platform contains two general-purpose

* ArrayList, which is usually the better-performing implementation, and
* LinkedList which offers better performance under certain circumstances.

List Algorithms

* sort — sorts a List using a merge sort algorithm, which provides a fast, stable sort. (A stable sort is one that does not reorder equal elements.)
* shuffle — randomly permutes the elements in a List.
* reverse — reverses the order of the elements in a List.
* rotate — rotates all the elements in a List by a specified distance.
* swap — swaps the elements at specified positions in a List.
* replaceAll — replaces all occurrences of one specified value with another.
* fill — overwrites every element in a List with the specified value.
* copy — copies the source List into the destination List.
* binarySearch — searches for an element in an ordered List using the binary search algorithm.
* indexOfSubList — returns the index of the first sublist of one List that is equal to another.
* lastIndexOfSubList — returns the index of the last sublist of one List that is equal to another.

**Homework:**

**Create a set try play around with all the above methods and print the result**

Queue — a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations.

Queues typically, but do not necessarily, order elements in a FIFO (first-in, first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator or the elements' natural ordering. Whatever the ordering used, the head of the queue is the element that would be removed by a call to remove or poll. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every Queue implementation must specify its ordering properties.

Deque — a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Deque provides additional insertion, extraction, and inspection operations.

Deques can be used both as FIFO (first-in, first-out) and LIFO (last-in, first-out). In a deque all new elements can be inserted, retrieved and removed at both ends.

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

The Map interface includes methods for

* basic operations (such as put, get, remove, containsKey, containsValue, size, and empty)
* bulk operations (such as putAll and clear)
* collection views (such as keySet, entrySet, and values).

The Java platform contains three general-purpose Map implementations:

1. HashMap
2. TreeMap
3. LinkedHashMap.

Below are the basic operations of Map

* put
* get
* containsKey
* containsValue
* size
* isEmpty

Below is sample example using HashMap

**import** java.util.HashMap;

**import** java.util.Iterator;

**import** java.util.Set;

**public** **class** Simple {

**public** **static** **void** main(String args[]) {

HashMap<String, String> h1 = **new** HashMap<>();

HashMap<String, String> h2 = **new** HashMap<>();

// put - Associates the specified value with the specified key in this map. If

// the map previously contained a mapping for the key, the old value is

// replaced.

h1.put("MH", "Maharashtra");

h1.put("GJ", "Gujarat");

h1.put("UP", "Uttar Pradesh");

h1.put("AP", "Arunachal Pradesh");

h1.put("TN", "Tamil Nadu");

h2.put("MH", "MH0212");

h2.put("KL", "Kerela");

h2.putAll(h1);

Set<String> s2 = h2.keySet();

Iterator<String> i2 = s2.iterator();

System.***out***.println();

System.***out***.println("Below is the Keyset and corresponding values for h2: ");

**while** (i2.hasNext()) {

String key2 = i2.next();

System.***out***.println(key2 + " - " + h2.get(key2));

}

// get - Returns the value to which the specified key is mapped, or null if this

// map contains no mapping for the key.

System.***out***.println(h1.get("Gj"));

System.***out***.println(h1.get("MH"));

System.***out***.println(h1.get("TN"));

System.***out***.println(h1.get("AP"));

// Returns true if this map contains a mapping for the specified key.

System.***out***.println("Is MH available in Set: " + h1.containsKey("MH"));

// Returns a Set view of the keys contained in this map

Set<String> s1 = h1.keySet();

Iterator<String> i1 = s1.iterator();

System.***out***.println();

System.***out***.println("Below is the Keyset and corresponding values for h1: ");

**while** (i1.hasNext()) {

String key1 = i1.next();

System.***out***.println(key1 + " - " + h1.get(key1));

}

}

}

**Map Interface Bulk Operations**

* clear - removes all the mappings from the Map
* putAll – Copies all of the mappings from the specified map to this map. These mappings will replace any mappings that this map had for any of the keys currently in the specified map.

SortedMap — a Map that maintains its mappings in ascending key order. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories.

MultiMap Example