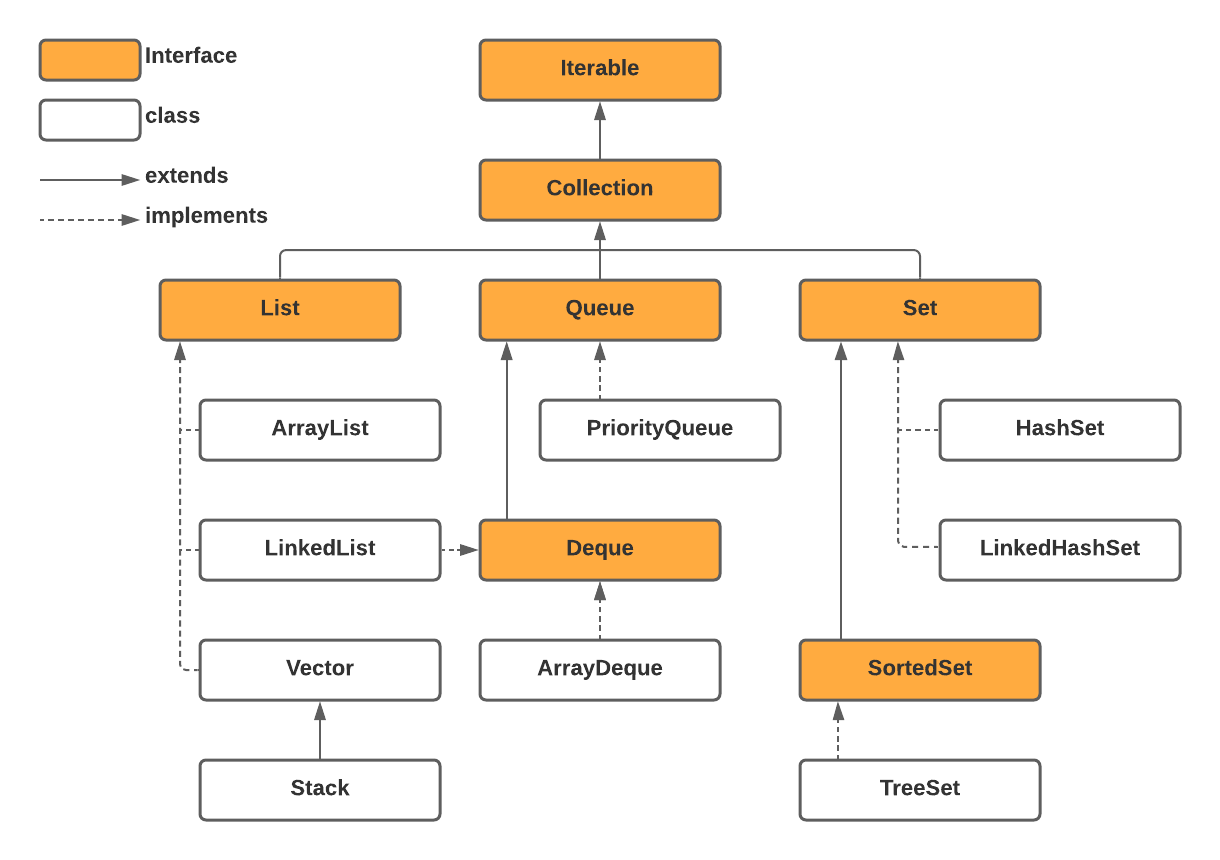
**Collection framework in Java**

The collection framework is a framework to work with a collection of data in Java.

If you are new to programming then you need to start with Array which is a core data structure to work with a collection of data.

The collection framework is a set of interfaces and concrete classes. We can use it to store and manipulate a group of objects. It supports built-in sorting and searching algorithms. We can say that the collection framework is ready APIS for various data structures. The collection framework is in core java libraries. We just need to import them from java.util package.



Before we get started to discuss each collection separately, there are a few common features you need to know:

* Collections can work(hold) only Objects, primitives are not allowed.
* Collections framework heavily utilize generics concept. For example, if you want to create ArrayList which can hold a collection of Strings, we can do it like this: *ArrayList<String> list = new ArrayList<>();*If you don’t specify a type it will assume as a collection of Objects.
* It’s common to use polymorphic way of creating instances from collections. ex: *List<String> list = new ArrayList<>();*

**List interface**

As you can see in the above scheme, List is an interface that has 3 concrete classes. The List is a dynamic size ordered data structure. It allows duplicate values.

Basically, collections allow us to work with a group of data. The same way as array does. Let’s see the differences in creating array and List.

// import statements and valid code above

int[] numberArray = new int[3];  
numberArray[0] = 1;  
numberArray[1] = 3;  
numberArray[2] = 7;  
System.out.println(Arrays.toString(numberArray)); // [1, 3, 7]

List<Integer> colorsList = new ArrayList<>();  
colorsList.add(1);  
colorsList.add(3);  
colorsList.add(4);  
System.out.println(colorsList); // [1, 3, 7]

In the above example, we created an array of int with values [1, 2, 3] and we created a List of Integer with values [1, 2, 3].

1. When we created an array of int we specify a size because an array is a fixed-size data structure, however, we don’t specify a size in the collections. We can think that collections have a dynamic size.
2. As you can see in the List example to specify the data type of it we need to use <> operator. It uses [generics](https://docs.oracle.com/javase/8/docs/technotes/guides/language/generics.html) in java.
3. An array can be created as a primitive data type but Collections do not allow primitives, they can hold only objects.

The list is an interface so it has many abstract methods and a List has three concrete classes that implement it. Why do we need three different implementations of the List? Because the way they implemented is different each implementation can be good for a specific scenario. Specifically ArrayList and LinkedList implementations.

[**ArrayList**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html) is the implementation of List. It’s based on an array internally and it can grow and shrink dynamically — resizable-array implementation. ArrayList not synchronized(it’s not threaded safe). If multiple threads will manipulate with the same ArrayList instance, java does not guarantee that it will behave as expected. It’s good to use when you don’t have to change(adding & removing) the number of elements inside significantly and it’s good for accessing elements by index. Why? Because it’s based on Array internally. Read this article on [Data Structures](https://beknazarsuranchiyev.medium.com/data-structures-with-java-b3d66a60ef2a) for more information.

[**LinkedList**](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html)implementation is based on a doubly-linked list. Because it’s based on a linked list internally, it has a dynamic size by nature of data structure. It’s not synchronized. It’s good to use when you don’t know the exact number of elements in advance and you don’t access different elements by index often. Why? Because it’s based doubly-linked list internally.

[**Vector**](https://docs.oracle.com/javase/8/docs/api/java/util/Vector.html). This class is roughly equivalent to ArrayList, except that it is synchronized.

Well, we discussed three main implementations of the [List interface](https://docs.oracle.com/javase/8/docs/api/java/util/List.html). Now, you just need to learn the methods(APIs) of List and you good to go.

import java.util.List;  
import java.util.ArrayList; public class Main {  
 public static void main(String[] args) {  
 // create list of String  
 List<String> cars = new ArrayList<>();

// add elements to back of the list  
 cars.add("bmw");  
 cars.add("audi");  
 cars.add("honda");  
 System.out.println(cars); // ["bmw", "audi", "honda"]  
   
 // insert element at specified position  
 cars.add(1, "tesla");  
 System.out.println(cars); // ["bmw", "tesla", "audi", "honda"]

// get specific element by index(index starts from 0)  
 String str = cars.get(2);  
 System.out.println(str); // audi

// find out number of elements in the list   
 int size = cars.size();   
 System.out.println("Size: " + size); // 4

// remove by index  
 cars.remove(0);  
 System.out.println(cars); // ["tesla", "audi", "honda"] // remove by value  
 cars.remove("honda");  
 System.out.println(cars); // ["tesla", "audi"]

// find index of specific element  
 int index = cars.indexOf("audi");  
 System.out.println(index); // 1

// check if list contains specified element  
 boolean isThere = cars.contains("bmw");  
 System.out.println(isThere); //false  
 }  
}

I used ArrayList implementation in the above example. It will be the same for all other implementations.

**Set Interface**

[Set](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html) does not allow duplicate values. There is no get(index) method in the Set interface. Overall, the performance of Set implementation classes is very high. It has three main implementations: HashSet, LinkedHashSet, TreeSet.

[**HashSet**](https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html)is the implementation of a Set interface. It based on hast table data structure. It does not guarantee insertion or iteration order. It allows one null value. HashSet has a constant-time performance. It doesn’t matter how large is collection, it will take constant time to perform basic operations as add, remove, contains, and size. It’s incredible! HashSet is good when you don’t care about ordering or sorting the data and If you don’t need to read by index.

[**LinkedHashSet**](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html)is exactly similar to HashSet except it will maintain insertion order. LinkedHashSet also guarantees constant-time performance. Performance is likely to be just slightly below that of HashSet, due to the added expense of maintaining the linked list.

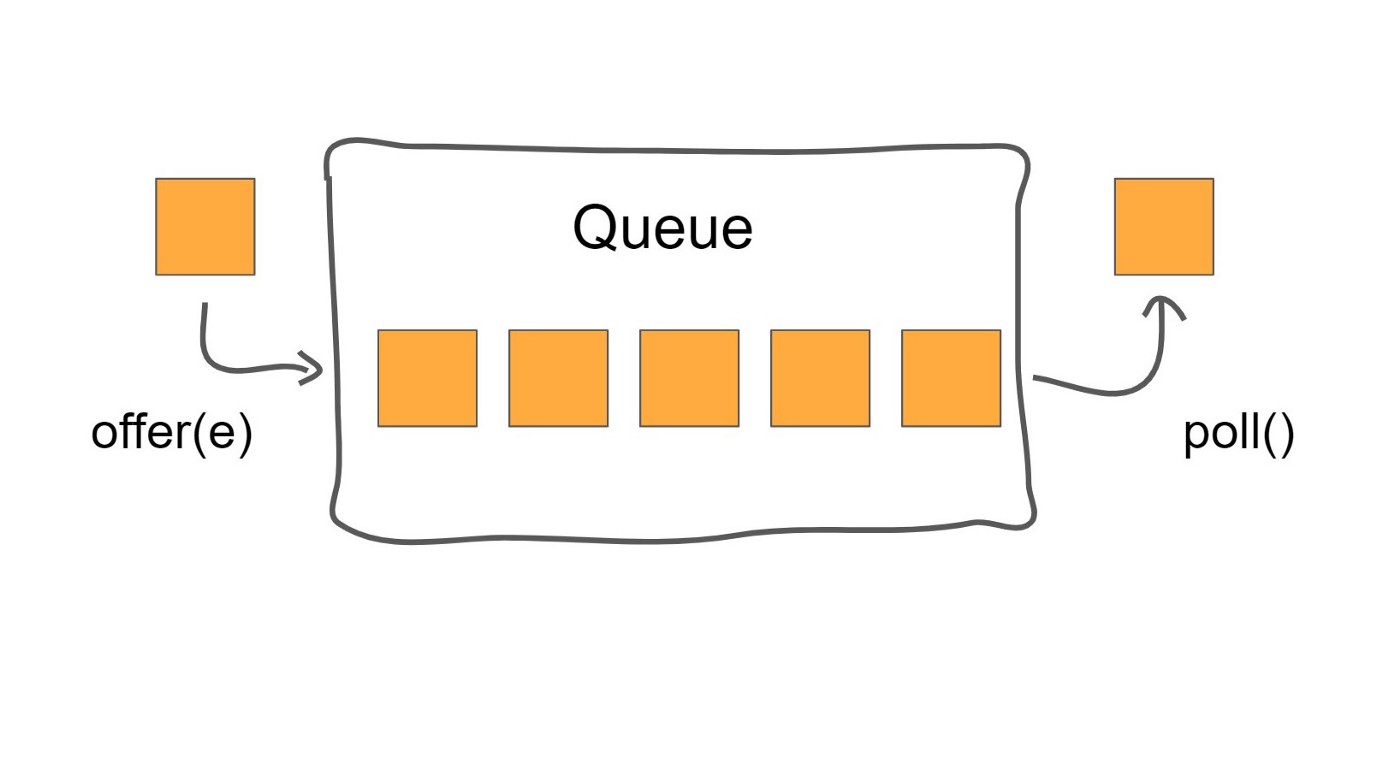
[**TreeSet**](https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html)is ordered set implementation. It follows the [*natural ordering*](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html)*.*What is natural ordering? Let’s first discuss the order itself. The data can be ordered differently. For example, it can be ordered alphabetically if it’s a collection of strings or it can be in ascending order if it’s a collection of numbers. And so on. If it’s a collection of custom objects, how it should be ordered? So in order to resolve this problem “How to order things?” Java has a [Comparable](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html) interface which has only one method *compareTo.*Every object that should be ordered can implement this interface and all other mechanisms of ordering in Java will be able to order your data. The same pattern for our TreeSet — it will order data based on the *compareTo* method of the set data type.

import java.util.\*;  
  
public class Main {  
 public static void main(String[] args) {  
 Set<String> cars = new HashSet<>();  
 cars.add("honda");  
 cars.add("audi");  
 cars.add("bmw");  
 cars.add("tesla");  
 cars.add("bmw");  
  
 // 1. We can see that there is no order  
 // 2. No duplicates  
 System.*out*.println(cars); // [audi, tesla, honda, bmw]  
  
  
 // find out number of elements in the set  
 System.*out*.println("Size: " + cars.size()); // Size: 4  
  
 // check if specific element is the set  
 System.*out*.println(cars.contains("honda")); // true  
  
 // check if set is empty  
 System.*out*.println(cars.isEmpty()); // false  
  
 // remove element from set if exist  
 // if it's removed return true otherwise false  
 System.*out*.println(cars.remove("honda")); // true  
 // there is no get(index) method in the Set  
 // We can iterate over the set with for each loop or Iterator<E>  
 for (String car : cars) {  
 // audi tesla bmw  
 System.*out*.print(car + " ");  
 }  
  
 System.*out*.println();  
 // Iterate over set with Iterator<E>  
 Iterator<String> iterator = cars.iterator();  
 while(iterator.hasNext()) {  
 // audi tesla bmw  
 System.*out*.print(iterator.next() + " ");  
 }  
 System.*out*.println();  
  
  
 // LinkedHashSet  
 Set<String> cars2 = new LinkedHashSet<>();  
 cars2.add("honda");  
 cars2.add("audi");  
 cars2.add("bmw");  
 cars2.add("tesla");  
 cars2.add("bmw");  
  
 // 1. LinkedHashSet implementation will keep insertion order  
 // 2. There is no duplicates allowed in set  
 System.*out*.println(cars2); // [honda, audi, bmw, tesla]  
  
  
 // TreeSet  
 Set<String> cars3 = new TreeSet<>();  
 cars3.add("honda");  
 cars3.add("audi");  
 cars3.add("bmw");  
 cars3.add("tesla");  
 cars3.add("bmw");  
  
 // 1. Data is ordered in natural order - for String is alphabetical order  
 // 2. There is no duplicates allowed in set  
 System.*out*.println(cars3); // [audi, bmw, honda, tesla]  
 }  
}

**Queue, Stack, and Dequeue**

These collections are designed for holding elements prior to processing.

[**Queue**](https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html) is First In First Out data structure. To understand a Queue data structure, you can always think about people waiting in the queue in the coffee shop. Who comes first will get served first and will be out of the queue first.



offer(e) adds to the back, poll() removes from the head

There are three main methods to work with the Queue interface:

1. *offer(e)* adds a new element to the back of the queue if possible otherwise. it will return false.
2. *poll()* method will remove the element from the head and return it.
3. *peek()* method returns, but do not remove, the head of the queue.

**Stack**is the Last In First Out data structure. Java does not have Stack interface, it has a [Stack](https://docs.oracle.com/javase/8/docs/api/java/util/Stack.html) class. Most of the time stack(LIFO) data structure is implemented by using Dequeue.

Think of a stack of plates. We always put a plate on top of the stack and when we need to take it from the top as well.

There are three main methods to work with the Stack:

1. *push(e)* Pushes an item onto the top of this stack.
2. *pop()* Removes the object at the top of this stack and returns that object as the value of this function.
3. *peek()* method returns an object from the top, but do not remove it.

[**Dequeue**](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html)is an interface that can behave as a Queue and as a Stack data structure as well.

import java.util.LinkedList;  
import java.util.Queue;  
import java.util.Stack;  
  
public class Main {  
 public static void main(String[] args) {  
 // Queue - FIFO (First In, First Out)  
 // LinkedList is one of the Queue implementations as well  
 Queue<String> queueInStore = new LinkedList<>();

queueInStore.add("Alex");  
 queueInStore.add("Misha");  
 queueInStore.add("Boris");  
 queueInStore.add("John");  
  
 // [Alex, Misha, Boris, John]  
 System.*out*.println(queueInStore);  
  
 // poll() method get element and remove it from queue  
 String currentCustomer = queueInStore.poll();  
  
 // Working with cutomer: Alex  
 System.*out*.println("Working with cutomer: " + currentCustomer);  
  
 // Alex is no longer in the queue  
 // Customers in line: [Misha, Boris, John]  
 System.*out*.println("Customers in line: " + queueInStore);  
 System.*out*.println("-------------");  
  
 currentCustomer = queueInStore.poll();  
 // Working with customer: Misha  
 System.*out*.println("Working with customer: " + currentCustomer);  
  
 // Customers in line: [Boris, John]  
 System.*out*.println("Customers in line: " + queueInStore);  
 System.*out*.println("==============");  
  
  
 // Stack  
 // LIFO - Last In First Out  
 Stack<String> messages = new Stack<>();  
 // push() will add element to the stack  
 // it will push an item onto the top of this stack  
 messages.push("Message from John");  
 messages.push("Message from Alex");  
 messages.push("Message from Smith");  
  
 //[Message from John, Message from Alex, Message from Smith]  
 System.*out*.println(messages);  
  
 //pop() will get elements from the top(last added)   
 // it will remove this element  
 String msg = messages.pop();  
  
 // Reading Message from Smith  
 System.*out*.println("Reading " + msg);  
 System.*out*.println("------");  
  
 msg = messages.pop();  
 // Reading Message from Alex  
 System.*out*.println("Reading " + msg);  
 System.*out*.println("------");  
 }  
}

**Map interface**

The [map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html) is a key value based data structure. In some programming languages, it’s known as a dictionary. Keys are unique in the map. The map is part of the collection framework(it does not extend Collection or Iterable).

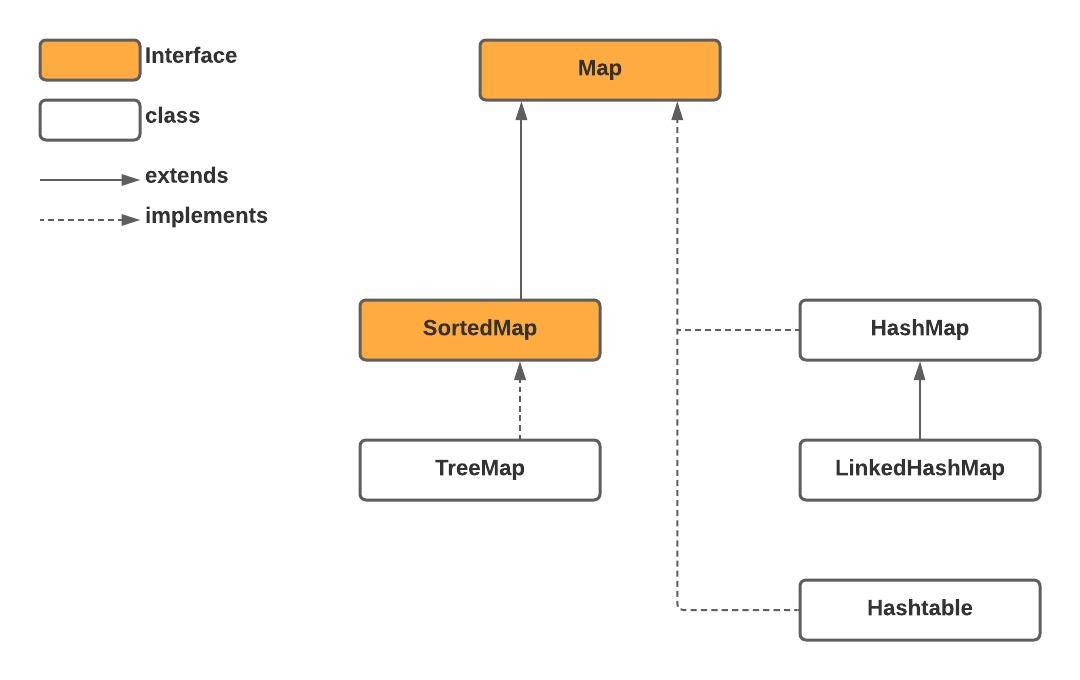


Diagram of the Map and some of its implementations

[**HashMap**](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) class is a hash table based implementation of the Map:  
1. Does not maintain any order.  
2. Allows one null as a key and any number of null as a value.  
3. Unsynchronized.

[**Hashtable**](https://docs.oracle.com/javase/8/docs/api/java/util/Hashtable.html). This class implements a hash table, which maps keys to values. Any non-null an object can be used as a key or as a value:  
1. Does not maintain any order.  
2. Does not allow null as key and value.  
3. Synchronized.

[**LinkedHashMap**](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html) class is almost the same as HashMap except it does maintain insertion order.

[**TreeMap**](https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html) class — Red-Black tree based [NavigableMap](https://docs.oracle.com/javase/8/docs/api/java/util/NavigableMap.html" \t "_blank) implementation. The map is sorted according to the [natural ordering](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html) of its keys, or by a [Comparator](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html) provided at map creation time, depending on which constructor is used.

import java.util.\*;  
  
public class Main {  
 public static void main(String[] args) {  
 // K , V  
 Map<String, Double> fruit = new HashMap<>();  
 fruit.put("Orange", 1.05);  
 fruit.put("Mango", 0.50);  
 fruit.put("Apple", 0.76);  
 fruit.put("Lemon", 1.47);  
 fruit.put("Banana", 1.39);  
 System.*out*.println(fruit);

// there is no indexes in the map  
 Double applePrice = fruit.get("Apple");  
 System.*out*.println("Apple price is " + applePrice);  
  
 System.*out*.println(fruit.get("Mango")); // 0.50  
 Double lemonPrice = fruit.get("abc");  
  
 // if key is not part of the map, it will return null  
 System.*out*.println(lemonPrice);  
  
 // Get sum of prices of Mango, Banana and Orange  
 Double sumPrice = fruit.get("Mango") + fruit.get("Banana") + fruit.get("Orange");  
 System.*out*.println("Price sum: " + sumPrice);  
  
 // put(K, V) - will put element into map  
 // get(K) - it will return V based on the key, if key is not in the map, it will return null  
 // containsKey(K) - returns true if key in the map otherwise false  
  
 boolean isThere = fruit.containsKey("Kiwi");  
 System.*out*.println("Kiwi is in the map: " + isThere);  
  
 String strKey = "Orange";  
 if (fruit.containsKey(strKey)) {  
 System.*out*.println("Price: " + fruit.get(strKey));  
 } else {  
 System.*out*.println("Key is not in the map: " + strKey);  
 }  
  
 // containsValue(V) true if value is there  
 System.*out*.println(fruit.containsValue(0.50));  
 System.*out*.println("-------");  
  
 // remove(K); - will remove entity from the map (K , V)  
 System.*out*.println(fruit);  
  
 // if key is not there nothing will happen  
 fruit.remove("Orange");  
 System.*out*.println(fruit);  
  
 // HashMap impl of Map does not maintain insertion order  
  
 // keySet() -> return all the keys of the as Set<K>  
 System.*out*.println("--- KEYS ---");  
 Set<String> keys = fruit.keySet();  
 System.*out*.println(keys);  
  
 // Mango=0.5  
 fruit.put("Mango", 1.50);  
 System.*out*.println(fruit);  
 }  
}