|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Collections in Java   1. [Java Collection Framework](https://www.javatpoint.com/collections-in-java) 2. [Hierarchy of Collection Framework](https://www.javatpoint.com/collections-in-java#collectionhierarchy) 3. [Collection interface](https://www.javatpoint.com/collections-in-java#collectionmethods) 4. [Iterator interface](https://www.javatpoint.com/collections-in-java#collectioniterator)   The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.  Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.  Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes ([ArrayList](https://www.javatpoint.com/java-arraylist), Vector, [LinkedList](https://www.javatpoint.com/java-linkedlist), [PriorityQueue](https://www.javatpoint.com/java-priorityqueue), HashSet, LinkedHashSet, TreeSet).  What is Collection in Java  A Collection represents a single unit of objects, i.e., a group.  What is a framework in Java   * It provides readymade architecture. * It represents a set of classes and interfaces. * It is optional.   What is Collection framework  The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:   1. Interfaces and its implementations, i.e., classes 2. Algorithm   Do You Know?   * What are the two ways to iterate the elements of a collection? * What is the difference between ArrayList and LinkedList classes in collection framework? * What is the difference between ArrayList and Vector classes in collection framework? * What is the difference between HashSet and HashMap classes in collection framework? * What is the difference between HashMap and Hashtable class? * What is the difference between Iterator and Enumeration interface in collection framework? * How can we sort the elements of an object? What is the difference between Comparable and Comparator interfaces? * What does the hashcode() method? * What is the difference between Java collection and Java collections?   Hierarchy of Collection Framework  Let us see the hierarchy of Collection framework. The **java.util** package contains all the [classes](https://www.javatpoint.com/object-and-class-in-java) and [interfaces](https://www.javatpoint.com/interface-in-java) for the Collection framework.  Hierarchy of Java Collection framework  Methods of Collection interface  There are many methods declared in the Collection interface. They are as follows:   |  |  |  | | --- | --- | --- | | **No.** | **Method** | **Description** | | 1 | public boolean add(E e) | It is used to insert an element in this collection. | | 2 | public boolean addAll(Collection<? extends E> c) | It is used to insert the specified collection elements in the invoking collection. | | 3 | public boolean remove(Object element) | It is used to delete an element from the collection. | | 4 | public boolean removeAll(Collection<?> c) | It is used to delete all the elements of the specified collection from the invoking collection. | | 5 | default boolean removeIf(Predicate<? super E> filter) | It is used to delete all the elements of the collection that satisfy the specified predicate. | | 6 | public boolean retainAll(Collection<?> c) | It is used to delete all the elements of invoking collection except the specified collection. | | 7 | public int size() | It returns the total number of elements in the collection. | | 8 | public void clear() | It removes the total number of elements from the collection. | | 9 | public boolean contains(Object element) | It is used to search an element. | | 10 | public boolean containsAll(Collection<?> c) | It is used to search the specified collection in the collection. | | 11 | public Iterator iterator() | It returns an iterator. | | 12 | public Object[] toArray() | It converts collection into array. | | 13 | public <T> T[] toArray(T[] a) | It converts collection into array. Here, the runtime type of the returned array is that of the specified array. | | 14 | public boolean isEmpty() | It checks if collection is empty. | | 15 | default Stream<E> parallelStream() | It returns a possibly parallel Stream with the collection as its source. | | 16 | default Stream<E> stream() | It returns a sequential Stream with the collection as its source. | | 17 | default Spliterator<E> spliterator() | It generates a Spliterator over the specified elements in the collection. | | 18 | public boolean equals(Object element) | It matches two collections. | | 19 | public int hashCode() | It returns the hash code number of the collection. |   Iterator interface   |  | | --- | | Iterator interface provides the facility of iterating the elements in a forward direction only. |   Methods of Iterator interface  There are only three methods in the Iterator interface. They are:   |  |  |  | | --- | --- | --- | | **No.** | **Method** | **Description** | | 1 | public boolean hasNext() | It returns true if the iterator has more elements otherwise it returns false. | | 2 | public Object next() | It returns the element and moves the cursor pointer to the next element. | | 3 | public void remove() | It removes the last elements returned by the iterator. It is less used. |   Iterable Interface  The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.  It contains only one abstract method. i.e.,   1. Iterator<T> iterator()   It returns the iterator over the elements of type T.  Collection Interface  The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.  Some of the methods of Collection interface are Boolean add ( Object obj), Boolean addAll ( Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.  List Interface  List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.  List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.  To instantiate the List interface, we must use :   1. List <data-type> list1= **new** ArrayList(); 2. List <data-type> list2 = **new** LinkedList(); 3. List <data-type> list3 = **new** Vector(); 4. List <data-type> list4 = **new** Stack();   There are various methods in List interface that can be used to insert, delete, and access the elements from the list.  The classes that implement the List interface are given below.  AD  ArrayList  The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.   1. **import** java.util.\*; 2. **class** TestJavaCollection1{ 3. **public** **static** **void** main(String args[]){ 4. ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist 5. list.add("Ravi");//Adding object in arraylist 6. list.add("Vijay"); 7. list.add("Ravi"); 8. list.add("Ajay"); 9. //Traversing list through Iterator 10. Iterator itr=list.iterator(); 11. **while**(itr.hasNext()){ 12. System.out.println(itr.next()); 13. } 14. } 15. }   Output:  Ravi  Vijay  Ravi  Ajay  LinkedList  LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.  Consider the following example.   1. **import** java.util.\*; 2. **public** **class** TestJavaCollection2{ 3. **public** **static** **void** main(String args[]){ 4. LinkedList<String> al=**new** LinkedList<String>(); 5. al.add("Ravi"); 6. al.add("Vijay"); 7. al.add("Ravi"); 8. al.add("Ajay"); 9. Iterator<String> itr=al.iterator(); 10. **while**(itr.hasNext()){ 11. System.out.println(itr.next()); 12. } 13. } 14. }   Output:  Ravi  Vijay  Ravi  Ajay  Vector  Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.  Consider the following example.   1. **import** java.util.\*; 2. **public** **class** TestJavaCollection3{ 3. **public** **static** **void** main(String args[]){ 4. Vector<String> v=**new** Vector<String>(); 5. v.add("Ayush"); 6. v.add("Amit"); 7. v.add("Ashish"); 8. v.add("Garima"); 9. Iterator<String> itr=v.iterator(); 10. **while**(itr.hasNext()){ 11. System.out.println(itr.next()); 12. } 13. } 14. }   Output:  Ayush  Amit  Ashish  Garima  Stack  The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.  Consider the following example.   1. **import** java.util.\*; 2. **public** **class** TestJavaCollection4{ 3. **public** **static** **void** main(String args[]){ 4. Stack<String> stack = **new** Stack<String>(); 5. stack.push("Ayush"); 6. stack.push("Garvit"); 7. stack.push("Amit"); 8. stack.push("Ashish"); 9. stack.push("Garima"); 10. stack.pop(); 11. Iterator<String> itr=stack.iterator(); 12. **while**(itr.hasNext()){ 13. System.out.println(itr.next()); 14. } 15. } 16. }   Output:  Ayush  Garvit  Amit  Ashish  Queue Interface  Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.  Queue interface can be instantiated as:   1. Queue<String> q1 = **new** PriorityQueue(); 2. Queue<String> q2 = **new** ArrayDeque();   There are various classes that implement the Queue interface, some of them are given below.  PriorityQueue  The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.  Consider the following example.   1. **import** java.util.\*; 2. **public** **class** TestJavaCollection5{ 3. **public** **static** **void** main(String args[]){ 4. PriorityQueue<String> queue=**new** PriorityQueue<String>(); 5. queue.add("Amit Sharma"); 6. queue.add("Vijay Raj"); 7. queue.add("JaiShankar"); 8. queue.add("Raj"); 9. System.out.println("head:"+queue.element()); 10. System.out.println("head:"+queue.peek()); 11. System.out.println("iterating the queue elements:"); 12. Iterator itr=queue.iterator(); 13. **while**(itr.hasNext()){ 14. System.out.println(itr.next()); 15. } 16. queue.remove(); 17. queue.poll(); 18. System.out.println("after removing two elements:"); 19. Iterator<String> itr2=queue.iterator(); 20. **while**(itr2.hasNext()){ 21. System.out.println(itr2.next()); 22. } 23. } 24. }   Output:  head:Amit Sharma  head:Amit Sharma  iterating the queue elements:  Amit Sharma  Raj  JaiShankar  Vijay Raj  after removing two elements:  Raj  Vijay Raj  Deque Interface  Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.  Deque can be instantiated as:   1. Deque d = **new** ArrayDeque();   ArrayDeque  ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.  ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.  Consider the following example.   1. **import** java.util.\*; 2. **public** **class** TestJavaCollection6{ 3. **public** **static** **void** main(String[] args) { 4. //Creating Deque and adding elements 5. Deque<String> deque = **new** ArrayDeque<String>(); 6. deque.add("Gautam"); 7. deque.add("Karan"); 8. deque.add("Ajay"); 9. //Traversing elements 10. **for** (String str : deque) { 11. System.out.println(str); 12. } 13. } 14. }   Output:  Gautam  Karan  Ajay  Set Interface  Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.  Set can be instantiated as:   1. Set<data-type> s1 = **new** HashSet<data-type>(); 2. Set<data-type> s2 = **new** LinkedHashSet<data-type>(); 3. Set<data-type> s3 = **new** TreeSet<data-type>();   HashSet  HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.  Consider the following example.   1. **import** java.util.\*; 2. **public** **class** TestJavaCollection7{ 3. **public** **static** **void** main(String args[]){ 4. //Creating HashSet and adding elements 5. HashSet<String> set=**new** HashSet<String>(); 6. set.add("Ravi"); 7. set.add("Vijay"); 8. set.add("Ravi"); 9. set.add("Ajay"); 10. //Traversing elements 11. Iterator<String> itr=set.iterator(); 12. **while**(itr.hasNext()){ 13. System.out.println(itr.next()); 14. } 15. } 16. }   Output:  Vijay  Ravi  Ajay  LinkedHashSet  LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.  Consider the following example.   1. **import** java.util.\*; 2. **public** **class** TestJavaCollection8{ 3. **public** **static** **void** main(String args[]){ 4. LinkedHashSet<String> set=**new** LinkedHashSet<String>(); 5. set.add("Ravi"); 6. set.add("Vijay"); 7. set.add("Ravi"); 8. set.add("Ajay"); 9. Iterator<String> itr=set.iterator(); 10. **while**(itr.hasNext()){ 11. System.out.println(itr.next()); 12. } 13. } 14. }   Output:  Ravi  Vijay  Ajay  SortedSet Interface  SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order. The SortedSet provides the additional methods that inhibit the natural ordering of the elements.  The SortedSet can be instantiated as:   1. SortedSet<data-type> set = **new** TreeSet();   TreeSet  Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.  Consider the following example:   1. **import** java.util.\*; 2. **public** **class** TestJavaCollection9{ 3. **public** **static** **void** main(String args[]){ 4. //Creating and adding elements 5. TreeSet<String> set=**new** TreeSet<String>(); 6. set.add("Ravi"); 7. set.add("Vijay"); 8. set.add("Ravi"); 9. set.add("Ajay"); 10. //traversing elements 11. Iterator<String> itr=set.iterator(); 12. **while**(itr.hasNext()){ 13. System.out.println(itr.next()); 14. } 15. } 16. }   Output:  Ajay  Ravi  Vijay  What are we going to learn in Java Collections Framework   1. [ArrayList class](https://www.javatpoint.com/java-arraylist) 2. [LinkedList class](https://www.javatpoint.com/java-linkedlist) 3. [List interface](https://www.javatpoint.com/java-list) 4. [HashSet class](https://www.javatpoint.com/java-hashset) 5. [LinkedHashSet class](https://www.javatpoint.com/java-linkedhashset) 6. [TreeSet class](https://www.javatpoint.com/java-treeset) 7. [PriorityQueue class](https://www.javatpoint.com/java-priorityqueue) 8. [Map interface](https://www.javatpoint.com/java-map) 9. [HashMap class](https://www.javatpoint.com/java-hashmap) 10. [LinkedHashMap class](https://www.javatpoint.com/java-linkedhashmap) 11. [TreeMap class](https://www.javatpoint.com/TreeMap-class-in-collection-framework) 12. [Hashtable class](https://www.javatpoint.com/Hashtable-class-in-collection-framework) 13. [Sorting](https://www.javatpoint.com/Sorting-in-collection-framework) 14. [Comparable interface](https://www.javatpoint.com/Comparable-interface-in-collection-framework) 15. [Comparator interface](https://www.javatpoint.com/Comparator-interface-in-collection-framework) 16. [Properties class in Java](https://www.javatpoint.com/properties-class-in-java)AD |

Java HashMap



Java **HashMap** class implements the Map interface which allows us *to store key and value pair*, where keys should be unique. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the java.util package.

HashMap in Java is like the legacy Hashtable class, but it is not synchronized. It allows us to store the null elements as well, but there should be only one null key. Since Java 5, it is denoted as HashMap<K,V>, where K stands for key and V for value. It inherits the AbstractMap class and implements the Map interface.

Points to remember

* Java HashMap contains values based on the key.
* Java HashMap contains only unique keys.
* Java HashMap may have one null key and multiple null values.
* Java HashMap is non synchronized.
* Java HashMap maintains no order.
* The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

Hierarchy of HashMap class

As shown in the above figure, HashMap class extends AbstractMap class and implements Map interface.

HashMap class declaration

Let's see the declaration for java.util.HashMap class.

1. **public** **class** HashMap<K,V> **extends** AbstractMap<K,V> **implements** Map<K,V>, Cloneable, Serializable

HashMap class Parameters

Let's see the Parameters for java.util.HashMap class.

* **K**: It is the type of keys maintained by this map.
* **V**: It is the type of mapped values.

AD

Constructors of Java HashMap class

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| HashMap() | It is used to construct a default HashMap. |
| HashMap(Map<? extends K,? extends V> m) | It is used to initialize the hash map by using the elements of the given Map object m. |
| HashMap(int capacity) | It is used to initializes the capacity of the hash map to the given integer value, capacity. |
| HashMap(int capacity, float loadFactor) | It is used to initialize both the capacity and load factor of the hash map by using its arguments. |

Methods of Java HashMap class

|  |  |
| --- | --- |
| **Method** | **Description** |
| void clear() | It is used to remove all of the mappings from this map. |
| boolean isEmpty() | It is used to return true if this map contains no key-value mappings. |
| Object clone() | It is used to return a shallow copy of this HashMap instance: the keys and values themselves are not cloned. |
| Set entrySet() | It is used to return a collection view of the mappings contained in this map. |
| Set keySet() | It is used to return a set view of the keys contained in this map. |
| V put(Object key, Object value) | It is used to insert an entry in the map. |
| void putAll(Map map) | It is used to insert the specified map in the map. |
| V putIfAbsent(K key, V value) | It inserts the specified value with the specified key in the map only if it is not already specified. |
| V remove(Object key) | It is used to delete an entry for the specified key. |
| boolean remove(Object key, Object value) | It removes the specified values with the associated specified keys from the map. |
| V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) | It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping). |
| V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction) | It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null. |
| V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) | It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null. |
| boolean containsValue(Object value) | This method returns true if some value equal to the value exists within the map, else return false. |
| boolean containsKey(Object key) | This method returns true if some key equal to the key exists within the map, else return false. |
| boolean equals(Object o) | It is used to compare the specified Object with the Map. |
| void forEach(BiConsumer<? super K,? super V> action) | It performs the given action for each entry in the map until all entries have been processed or the action throws an exception. |
| V get(Object key) | This method returns the object that contains the value associated with the key. |
| V getOrDefault(Object key, V defaultValue) | It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key. |
| boolean isEmpty() | This method returns true if the map is empty; returns false if it contains at least one key. |
| V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction) | If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value. |
| V replace(K key, V value) | It replaces the specified value for a specified key. |
| boolean replace(K key, V oldValue, V newValue) | It replaces the old value with the new value for a specified key. |
| void replaceAll(BiFunction<? super K,? super V,? extends V> function) | It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. |
| Collection<V> values() | It returns a collection view of the values contained in the map. |
| int size() | This method returns the number of entries in the map. |

Java HashMap Example

Let's see a simple example of HashMap to store key and value pair.

1. **import** java.util.\*;
2. **public** **class** HashMapExample1{
3. **public** **static** **void** main(String args[]){
4. HashMap<Integer,String> map=**new** HashMap<Integer,String>();//Creating HashMap
5. map.put(1,"Mango");  //Put elements in Map
6. map.put(2,"Apple");
7. map.put(3,"Banana");
8. map.put(4,"Grapes");
10. System.out.println("Iterating Hashmap...");
11. **for**(Map.Entry m : map.entrySet()){
12. System.out.println(m.getKey()+" "+m.getValue());
13. }
14. }
15. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=HashMapExample1)

Iterating Hashmap...

1 Mango

2 Apple

3 Banana

4 Grapes

In this example, we are storing Integer as the key and String as the value, so we are using HashMap<Integer,String> as the type. The put() method inserts the elements in the map.

To get the key and value elements, we should call the getKey() and getValue() methods. The Map.Entry interface contains the *getKey()* and *getValue()* methods. But, we should call the entrySet() method of Map interface to get the instance of Map.Entry.

No Duplicate Key on HashMap

You cannot store duplicate keys in HashMap. However, if you try to store duplicate key with another value, it will replace the value.

1. **import** java.util.\*;
2. **public** **class** HashMapExample2{
3. **public** **static** **void** main(String args[]){
4. HashMap<Integer,String> map=**new** HashMap<Integer,String>();//Creating HashMap
5. map.put(1,"Mango");  //Put elements in Map
6. map.put(2,"Apple");
7. map.put(3,"Banana");
8. map.put(1,"Grapes"); //trying duplicate key
10. System.out.println("Iterating Hashmap...");
11. **for**(Map.Entry m : map.entrySet()){
12. System.out.println(m.getKey()+" "+m.getValue());
13. }
14. }
15. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=HashMapExample2)

Iterating Hashmap...

1 Grapes

2 Apple

3 Banana

Java HashMap example to add() elements

Here, we see different ways to insert elements.

1. **import** java.util.\*;
2. **class** HashMap1{
3. **public** **static** **void** main(String args[]){
4. HashMap<Integer,String> hm=**new** HashMap<Integer,String>();
5. System.out.println("Initial list of elements: "+hm);
6. hm.put(100,"Amit");
7. hm.put(101,"Vijay");
8. hm.put(102,"Rahul");
10. System.out.println("After invoking put() method ");
11. **for**(Map.Entry m:hm.entrySet()){
12. System.out.println(m.getKey()+" "+m.getValue());
13. }
15. hm.putIfAbsent(103, "Gaurav");
16. System.out.println("After invoking putIfAbsent() method ");
17. **for**(Map.Entry m:hm.entrySet()){
18. System.out.println(m.getKey()+" "+m.getValue());
19. }
20. HashMap<Integer,String> map=**new** HashMap<Integer,String>();
21. map.put(104,"Ravi");
22. map.putAll(hm);
23. System.out.println("After invoking putAll() method ");
24. **for**(Map.Entry m:map.entrySet()){
25. System.out.println(m.getKey()+" "+m.getValue());
26. }
27. }
28. }

Initial list of elements: {}

After invoking put() method

100 Amit

101 Vijay

102 Rahul

After invoking putIfAbsent() method

100 Amit

101 Vijay

102 Rahul

103 Gaurav

After invoking putAll() method

100 Amit

101 Vijay

102 Rahul

103 Gaurav

104 Ravi

Java HashMap example to remove() elements

Here, we see different ways to remove elements.

1. **import** java.util.\*;
2. **public** **class** HashMap2 {
3. **public** **static** **void** main(String args[]) {
4. HashMap<Integer,String> map=**new** HashMap<Integer,String>();
5. map.put(100,"Amit");
6. map.put(101,"Vijay");
7. map.put(102,"Rahul");
8. map.put(103, "Gaurav");
9. System.out.println("Initial list of elements: "+map);
10. //key-based removal
11. map.remove(100);
12. System.out.println("Updated list of elements: "+map);
13. //value-based removal
14. map.remove(101);
15. System.out.println("Updated list of elements: "+map);
16. //key-value pair based removal
17. map.remove(102, "Rahul");
18. System.out.println("Updated list of elements: "+map);
19. }
20. }

Output:

Initial list of elements: {100=Amit, 101=Vijay, 102=Rahul, 103=Gaurav}

Updated list of elements: {101=Vijay, 102=Rahul, 103=Gaurav}

Updated list of elements: {102=Rahul, 103=Gaurav}

Updated list of elements: {103=Gaurav}

Java HashMap example to replace() elements

Here, we see different ways to replace elements.

1. **import** java.util.\*;
2. **class** HashMap3{
3. **public** **static** **void** main(String args[]){
4. HashMap<Integer,String> hm=**new** HashMap<Integer,String>();
5. hm.put(100,"Amit");
6. hm.put(101,"Vijay");
7. hm.put(102,"Rahul");
8. System.out.println("Initial list of elements:");
9. **for**(Map.Entry m:hm.entrySet())
10. {
11. System.out.println(m.getKey()+" "+m.getValue());
12. }
13. System.out.println("Updated list of elements:");
14. hm.replace(102, "Gaurav");
15. **for**(Map.Entry m:hm.entrySet())
16. {
17. System.out.println(m.getKey()+" "+m.getValue());
18. }
19. System.out.println("Updated list of elements:");
20. hm.replace(101, "Vijay", "Ravi");
21. **for**(Map.Entry m:hm.entrySet())
22. {
23. System.out.println(m.getKey()+" "+m.getValue());
24. }
25. System.out.println("Updated list of elements:");
26. hm.replaceAll((k,v) -> "Ajay");
27. **for**(Map.Entry m:hm.entrySet())
28. {
29. System.out.println(m.getKey()+" "+m.getValue());
30. }
31. }
32. }

Initial list of elements:

100 Amit

101 Vijay

102 Rahul

Updated list of elements:

100 Amit

101 Vijay

102 Gaurav

Updated list of elements:

100 Amit

101 Ravi

102 Gaurav

Updated list of elements:

100 Ajay

101 Ajay

102 Ajay

Difference between HashSet and HashMap

HashSet contains only values whereas HashMap contains an entry(key and value).

Java HashMap Example: Book

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** MapExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating map of Books
17. Map<Integer,Book> map=**new** HashMap<Integer,Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to map
23. map.put(1,b1);
24. map.put(2,b2);
25. map.put(3,b3);
27. //Traversing map
28. **for**(Map.Entry<Integer, Book> entry:map.entrySet()){
29. **int** key=entry.getKey();
30. Book b=entry.getValue();
31. System.out.println(key+" Details:");
32. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
33. }
34. }
35. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=MapExample)

Output:

1 Details:

101 Let us C Yashwant Kanetkar BPB 8

2 Details:

102 Data Communications and Networking Forouzan Mc Graw Hill 4

3 Details:

103 Operating System Galvin Wiley 6

Related Topics

[How to iterate Map in Java](https://www.javatpoint.com/how-to-iterate-map-in-java)

[How to Sort HashMap in Java](https://www.javatpoint.com/how-to-sort-hashmap-in-java)

[Load Factor in HashMap](https://www.javatpoint.com/load-factor-in-hashmap)

[Working of HashMap in Java | How HashMap Works](https://www.javatpoint.com/working-of-hashmap-in-java)

[Difference between HashMap and Hashtable](https://www.javatpoint.com/difference-between-hashmap-and-hashtable)

[How to Sort HashMap by Value](https://www.javatpoint.com/how-to-sort-hashmap-by-value)

[Difference between HashSet and HashMap](https://www.javatpoint.com/difference-between-hashset-and-hashmap)

[Difference between HashMap and TreeMap](https://www.javatpoint.com/difference-between-hashmap-and-treemap)

[Java Map Interface](https://www.javatpoint.com/java-map)