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)

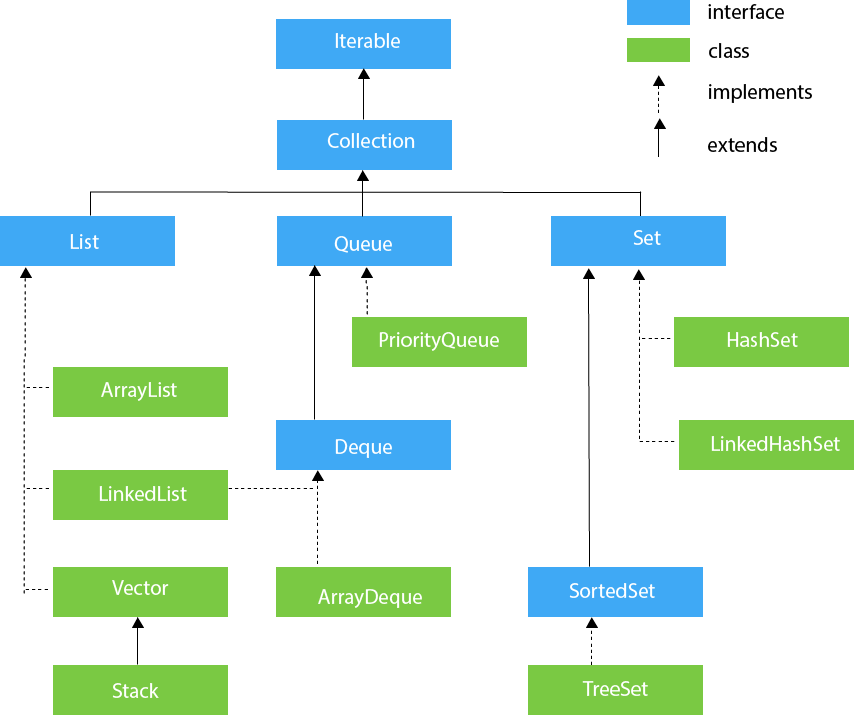
The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

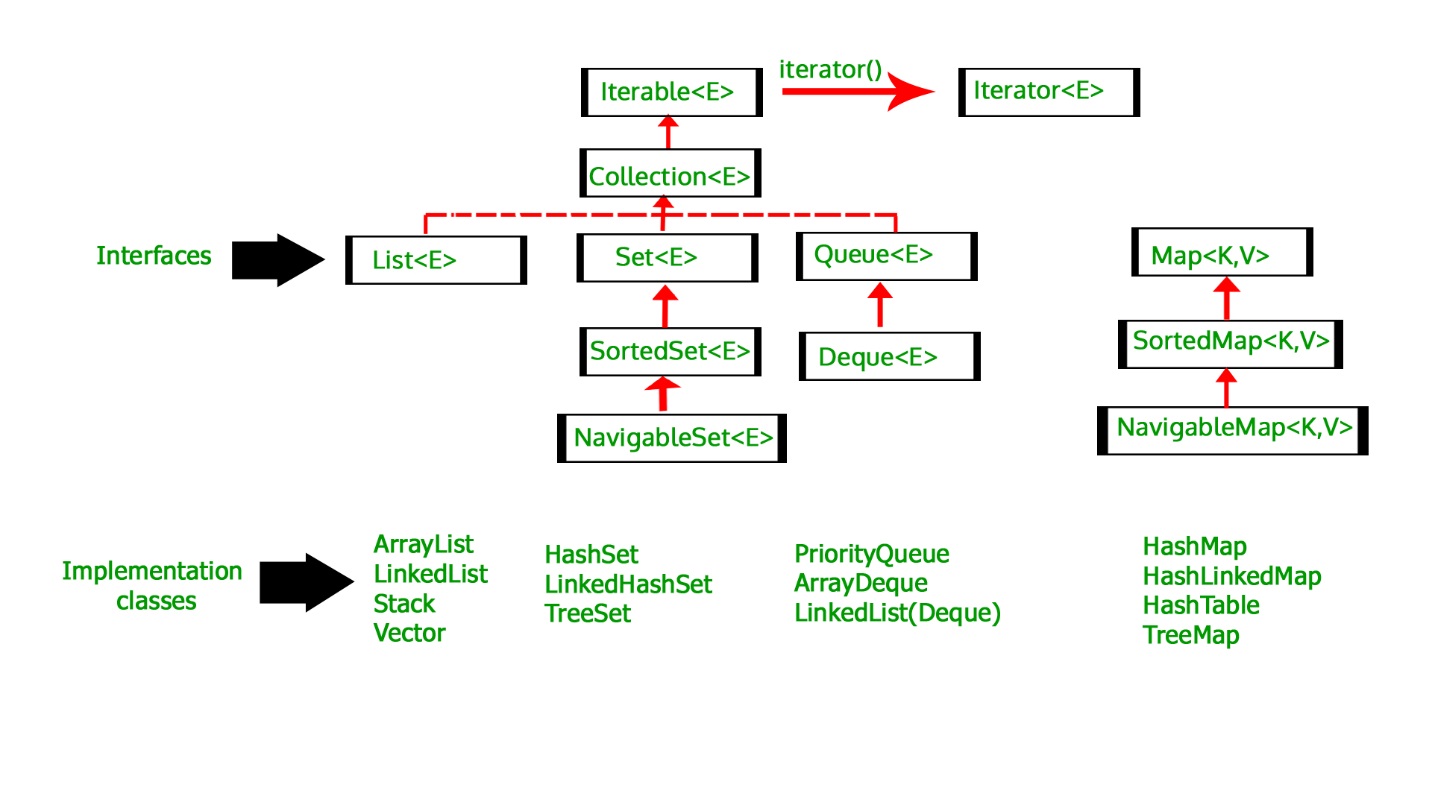
Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

#### **What is Collection in Java**

A Collection represents a single unit of objects, i.e., a group.

### **Hierarchy of Collection Framework**

Let us see the hierarchy of Collection framework. The **java.util** package contains all the classes and interfaces for the Collection framework.



**Hierarchy of Collection Framework**

Collection Map

/ / \ \ |

/ / \ \ |

Set List Queue Dequeue SortedMap

/

/

SortedSet

**Core Interfaces in Collections**

Note that this diagram only shows core interfaces.

**Collection :** Root interface with basic methods like add(), remove(),

contains(), isEmpty(), addAll(), ... etc.

[**Set**](https://www.geeksforgeeks.org/set-in-java/) **:** Doesn't allow duplicates. Example implementations of Set

interface are HashSet (Hashing based) and TreeSet (balanced

BST based). Note that TreeSet implements **SortedSet**.

[**List**](https://www.geeksforgeeks.org/list-interface-java-examples/) **:** Can contain duplicates and elements are ordered. Example

implementations are LinkedList (linked list based) and

[ArrayList](https://www.geeksforgeeks.org/array-vs-arraylist-in-java/) (dynamic array based)

[**Queue**](https://www.geeksforgeeks.org/queue-interface-java/) **:** Typically order elements in FIFO order except exceptions

like PriorityQueue.

[**Deque**](https://www.geeksforgeeks.org/deque-interface-java-example/) **:** Elements can be inserted and removed at both ends. Allows

both LIFO and FIFO.

[**Map**](https://www.geeksforgeeks.org/map-interface-java-examples/) **:** Contains Key value pairs. Doesn't allow duplicates. Example

implementation are [HashMap](http://www.geeksforgeeks.org/java-util-hashmap-in-java/) and TreeMap.

[TreeMap](https://www.geeksforgeeks.org/treemap-in-java/) implements **SortedMap**.

The difference between Set and Map interface is that in Set we

have only keys, whereas in Map, we have key, value pairs.

### **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.

## **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

# When to use ArrayList vs LinkedList in Java

ArrayList and LinkedList are two popular concrete implementations of List interface from Java's popular Collection framework. Being List implementation both ArrayList and LinkedList are ordered, the index based and allows duplicate. Despite being from same type hierarchy there are a lot of differences between these two classes which makes them popular among Java interviewers. The main difference between ArrayList vs LinkedList is that former is backed by an array while later is based upon linked list data structure, which makes the performance of add(), remove(), contains() and iterator() different for both ArrayList and LinkedList.

## **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

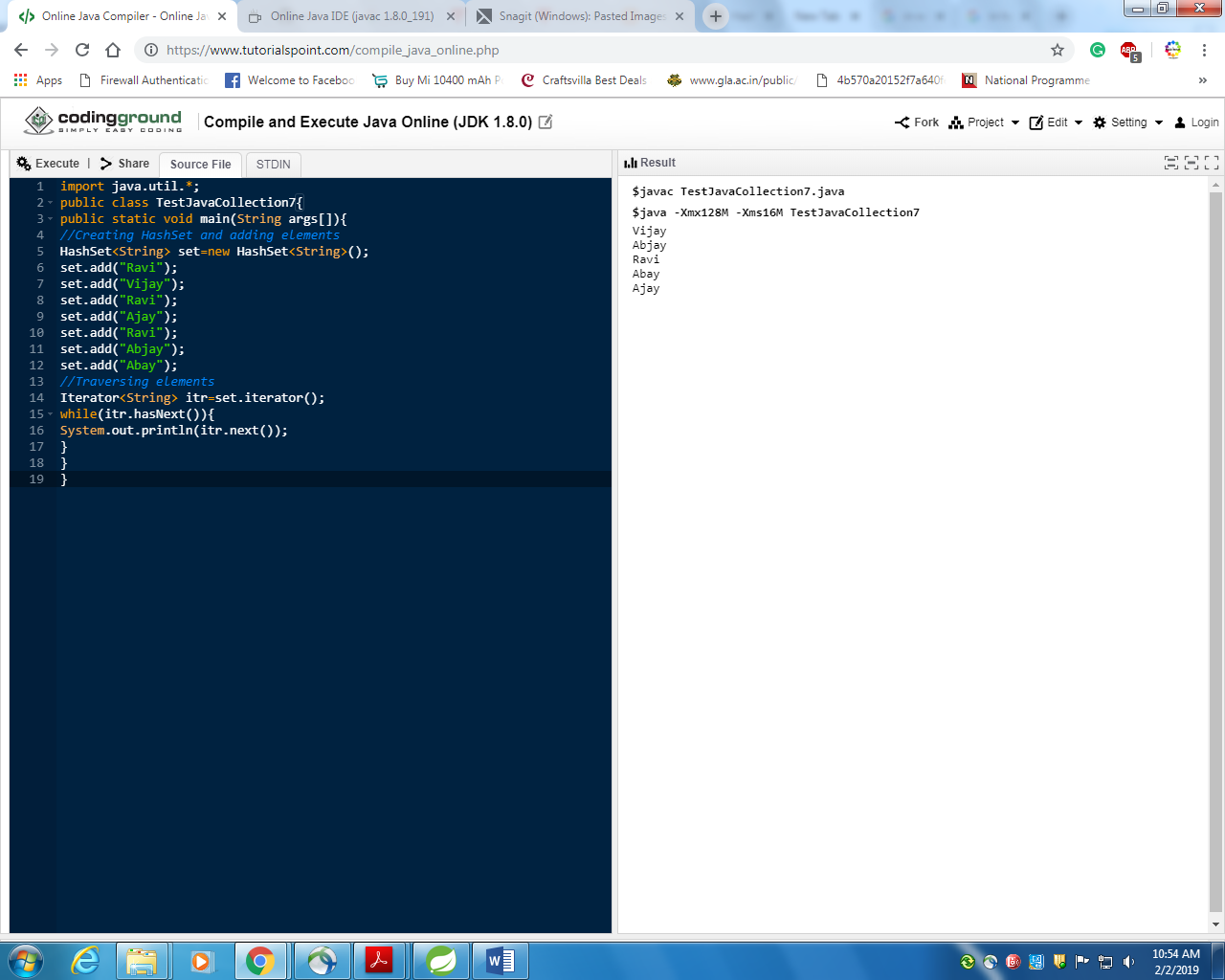


Fig compiler 1

## 

Fig compiler 2

## as it said in hashset order doesnot maintain but in above different compiler order comes to be same ? so my question is why does it contradicts ?

Ans is not maintain order, means insertion order is not preserve, there may be chances that output may be come in insertion order but that is only coincidence .

## **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

## **HashSet Vs TreeSet Vs LinkedHashSet In Java :**

Even though, **HashSet**, **LinkedHashSet** and **TreeSet** are all implementations of Set interface, there are some differences exist between them. In this article, I have tried to list out the differences between HashSet, LinkedHashSet and TreeSet in java. They also have some similarities between them. We will also discuss them at the end.

## **Differences Between HashSet, LinkedHashSet and TreeSet In Java :**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **HashSet** | **LinkedHashSet** | **TreeSet** |
|  |  |  |  |
| **How they work internally**? | HashSet uses HashMap internally to store it’s elements. | LinkedHashSet uses  LinkedHashMap internally to store it’s elements. | TreeSet uses TreeMap internally to store it’s elements. |
| **Order Of Elements** | HashSet doesn’t maintain any order of elements. | LinkedHashSet maintains insertion order of elements. i.e elements are placed as they are inserted. | TreeSet orders the elements according to supplied Comparator. If no comparator is supplied, elements will be placed in their natural ascending order. |
| **Performance** | HashSet gives better performance than the LinkedHashSet and TreeSet. | The performance of LinkedHashSet is between HashSet and TreeSet. It’s performance is almost similar to HashSet. But slightly in the slower side as it also maintains LinkedList internally to maintain the insertion order of elements. | TreeSet gives less performance than the HashSet and LinkedHashSet as it has to sort the elements after each insertion and removal operations. |
| **Insertion, Removal And Retrieval Operations** | HashSet gives performance of order O(1) for insertion, removal and retrieval operations. | LinkedHashSet also gives performance of order O(1) for insertion, removal and retrieval operations. | TreeSet gives performance of order O(log(n)) for insertion, removal and retrieval operations. |
| **How they compare the elements?** | HashSet uses equals() and hashCode() methods to compare the elements and thus removing the possible duplicate elements. | LinkedHashSet also uses equals() and hashCode() methods to compare the elements. | TreeSet uses compare() or compareTo() methods to compare the elements and thus removing the possible duplicate elements. It doesn’t use equals() and hashCode() methods for comparision of elements. |
| **Null elements** | HashSet allows maximum one null element. | LinkedHashSet also allows maximum one null element. | TreeSet doesn’t allow even a single null element. If you try to insert null element into TreeSet, it throws NullPointerException. |
| **Memory Occupation** | HashSet requires less memory than LinkedHashSet and TreeSet as it uses only HashMap internally to store its elements. | LinkedHashSet requires more memory than HashSet as it also maintains LinkedList along with HashMap to store its elements. | TreeSet also requires more memory than HashSet as it also maintains Comparator to sort the elements along with the TreeMap. |
| **When To Use?** | Use HashSet if you don’t want to maintain any order of elements. | Use LinkedHashSet if you want to maintain insertion order of elements. | Use TreeSet if you want to sort the elements according to some Comparator. |

# How to print in collections except hashmap

There are three common ways to iterate through a Collection in Java using either while(), for() or for-each().

## **While**

Java



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | import java.util.ArrayList;  import java.util.Collection;  import java.util.Iterator;    public class WhileIteration {        public static void main(String[] args) {            Collection<String> collection = new ArrayList<String>();            collection.add("zero");          collection.add("one");          collection.add("two");            Iterator<string> iterator = collection.iterator();            // while loop          while (iterator.hasNext()) {          System.out.println("value= " + iterator.next());          }      }  } |

## **For**

Java



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | import java.util.ArrayList;  import java.util.Collection;  import java.util.Iterator;    public class ForIteration {        public static void main(String[] args) {            Collection<String> collection = new ArrayList<String>();            collection.add("zero");          collection.add("one");          collection.add("two");            // for loop          for (Iterator<String> iterator = collection.iterator(); iterator.hasNext();) {          System.out.println("value= " + iterator.next());          }      }  } |

## **For-Each**

Java



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | import java.util.ArrayList;  import java.util.Collection;    public class ForEachInteration {        public static void main(String[] args) {            Collection<String> collection = new ArrayList<String>();            collection.add("zero");          collection.add("one");          collection.add("two");            // for-each loop          for (String s : collection) {          System.out.println("value= " + s);          }      }  } |

## **Result**

The result for each method should look like this:

|  |  |
| --- | --- |
| 1  2  3 | value= zero  value= one  value= two |

# Printing key and value in hash map

private HashMap<TypeKey,TypeValue > example= new HashMap<TypeKey, TypeValue>();

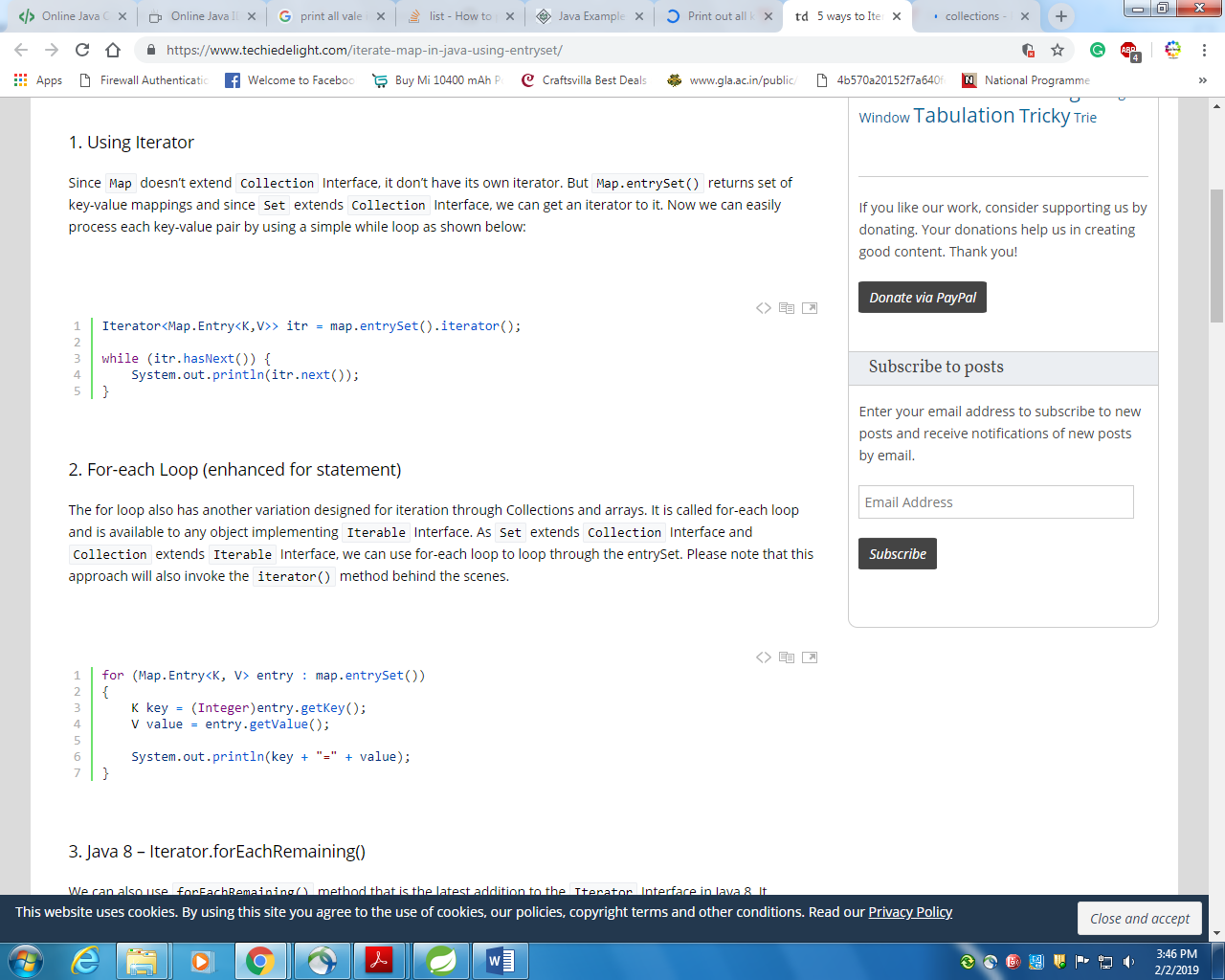
for (TypeKey name: example.keySet()){

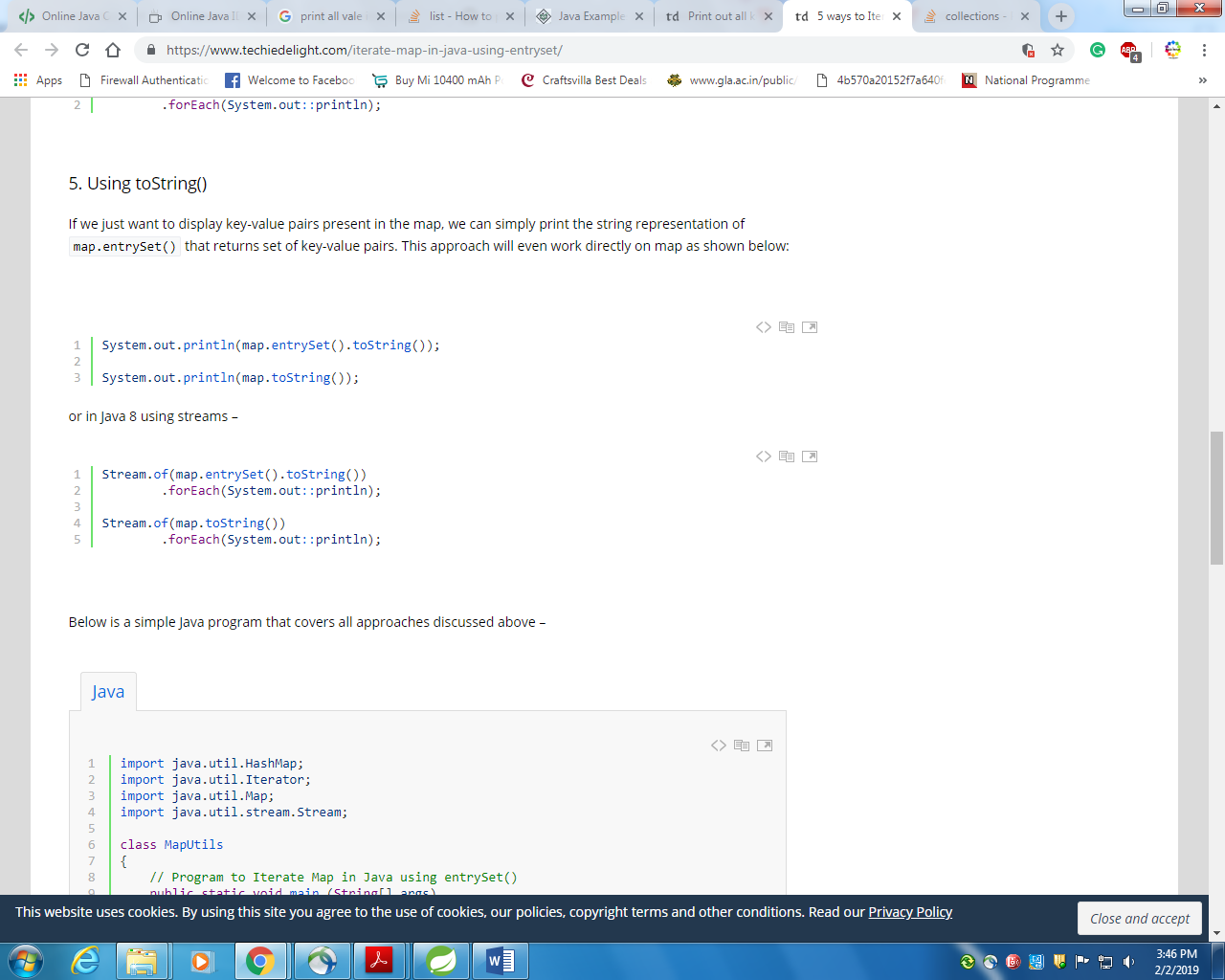
String key =name.toString();

String value = example.get(name).toString();

System.out.println(key + " " + value);

}



Gg

GENERICS

here are mainly 3 advantages of generics. They are as follows:

**1) Type-safety :** We can hold only a single type of objects in generics. It doesn’t allow to store other objects.

**2) Type casting is not required:** There is no need to typecast the object.

Before Generics, we need to type cast.

1. List list = new ArrayList();
2. list.add("hello");
3. String s = (String) list.get(0);//typecasting

After Generics, we don't need to typecast the object.

1. List<String> list = new ArrayList<String>(); // List<T>list
2. list.add("hello");
3. String s = list.get(0);

# What is the difference between a reference variable and an object in Java? How can I create both?

[Answer](https://www.quora.com/What-is-the-difference-between-a-reference-variable-and-an-object-in-Java-How-can-I-create-both)

[Follow· 1415](https://www.quora.com/What-is-the-difference-between-a-reference-variable-and-an-object-in-Java-How-can-I-create-both)

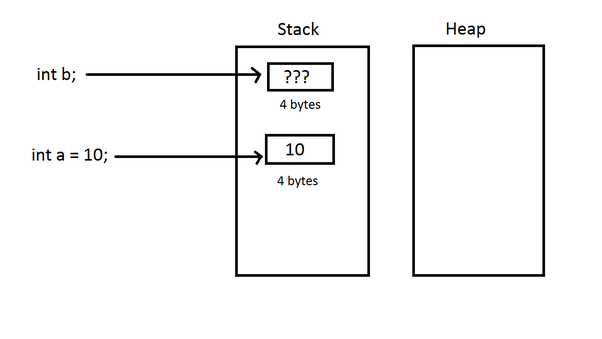
[Request](https://www.quora.com/What-is-the-difference-between-a-reference-variable-and-an-object-in-Java-How-can-I-create-both)

6 Answers

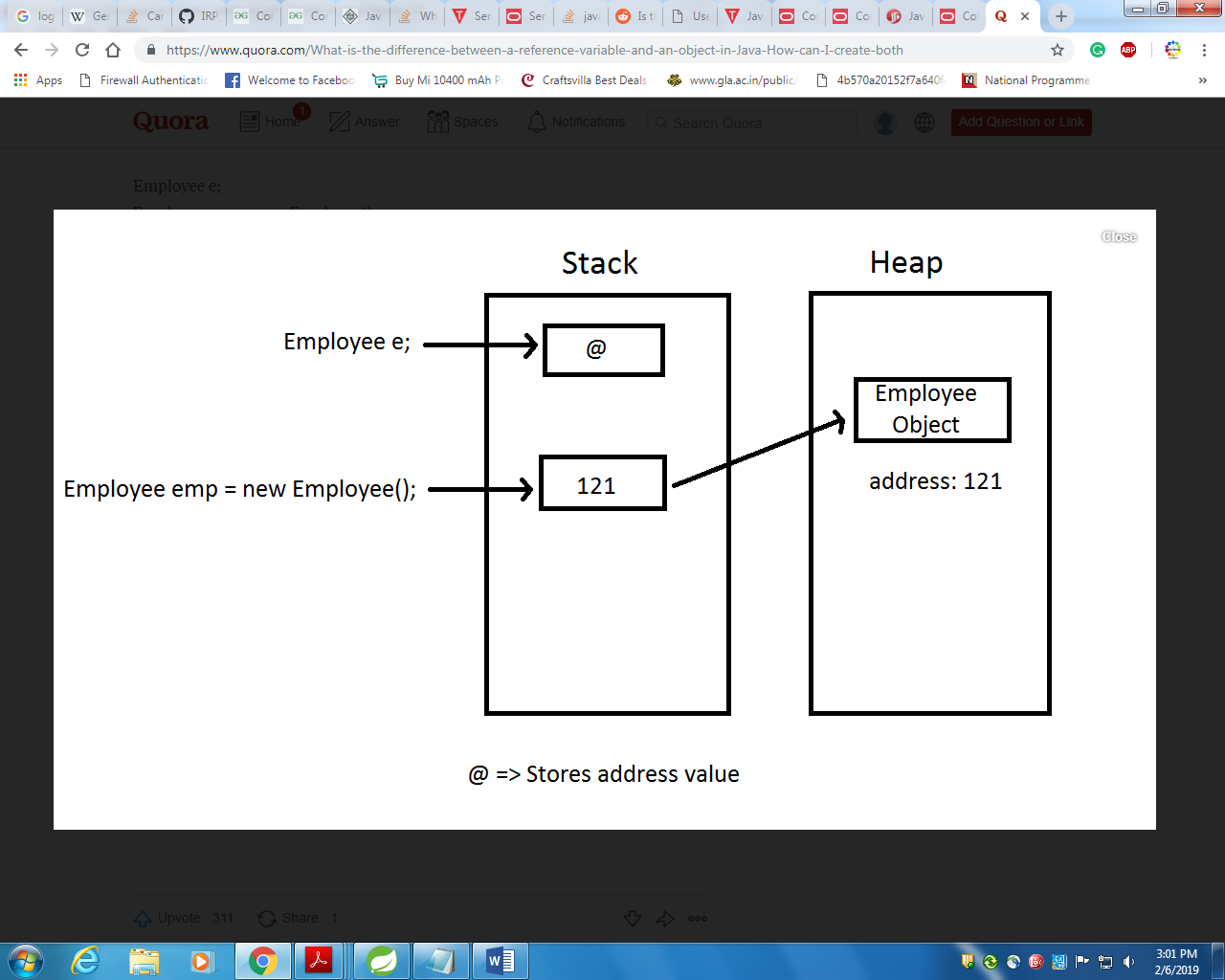
[Rajiv Sharma](https://www.quora.com/profile/Rajiv-Sharma-39), Software Developer at Microsoft

[Answered Jun 20, 2015](https://www.quora.com/What-is-the-difference-between-a-reference-variable-and-an-object-in-Java-How-can-I-create-both/answer/Rajiv-Sharma-39)

Lets assume that we have a class called Employee in Java. This is how you would create a reference of the Employee class:  
  
*Create a reference*  
**Employee e;**  
  
Now if you wanted to create an instance or object of the Employee class, you would do so:  
  
*Create an reference and an object*  
**Employee e = new Employee();**  
  
This gives a direct answer to your question, but to understand the real difference between references and objects, you need to understand a little about how memory is managed in Java  
  
In java, all types are divided into two categories,  *primitive types*which include int, long, char, boolean and so on, and *reference types* like String, Date, BufferedReader and other classes. This difference is based on the manner in which memory is allocated to the variables of these types. For example,  
  
int b;  
int a = 10;  
  
Both these statements cause the same effect, memory wise. A block of memory from the stack, big enough to hold an int (4 bytes) is allocated to these variables. This memory is allocated statically, since the size of int and other primitive types is fixed. (The ??? in the diagram indicate some unknown value since the integer has not been initialized yet)



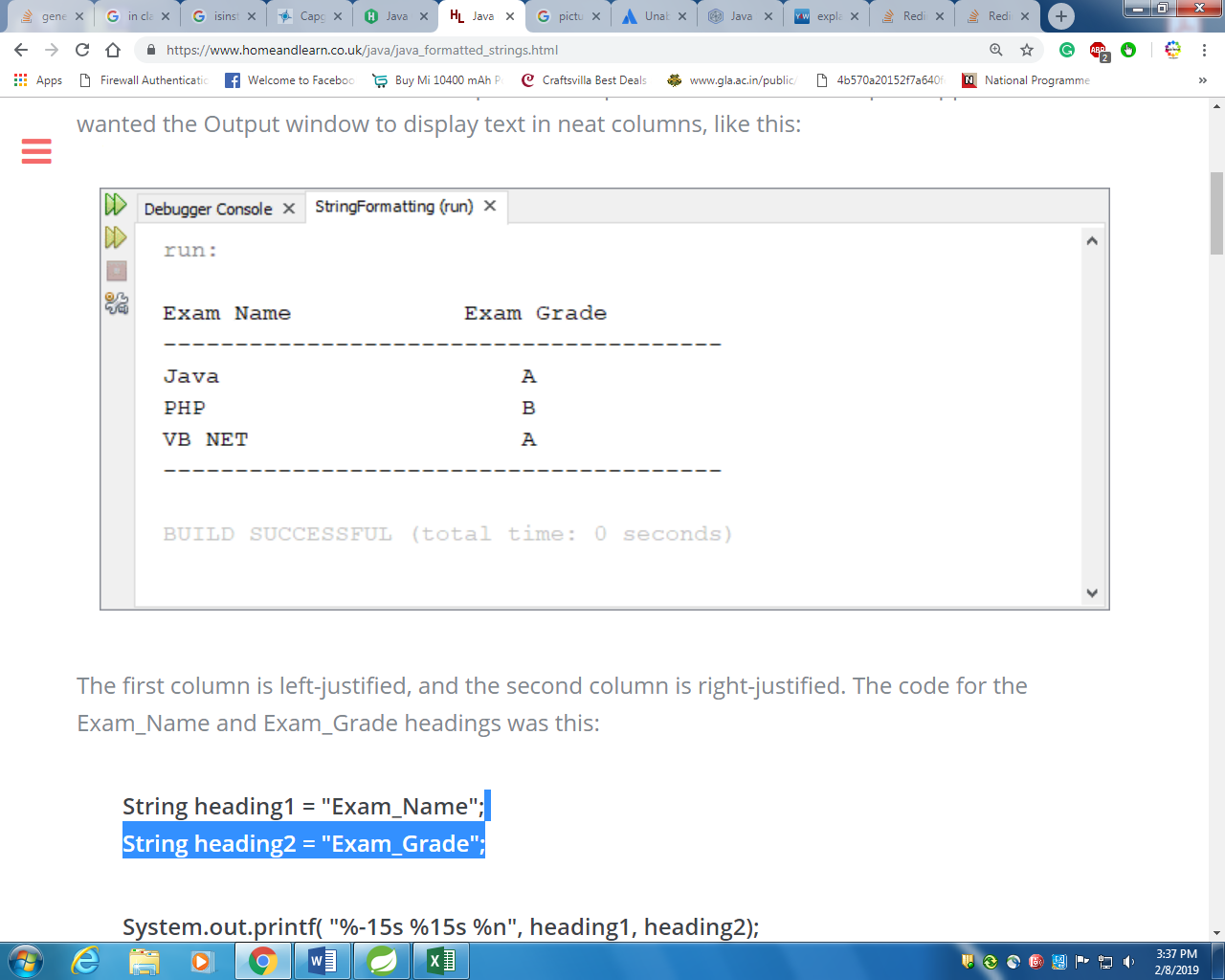
Consider the employee class.  
  
Employee e;  
Employee emp = new Employee();  
  
These two statements have different effect memory-wise since they are reference types. The first statement just creates a reference, or a pointer if you will, to an instance of type Employee. This essentially means, the statement tells the compiler**"e" is going to point (refer) to an Employee object, but right now is pointing to nothing (null)**. The interesting part is, the reference itself, that is "e" is stored on the stack statically. So, here you create *just the reference*.  
  
The second statement however, does more than this. "emp" is allotted memory as a reference as in the previous case, but the use of *new* keyword creates an object and allots memory to it on the heap, at runtime, i.e dynamically. This statement tells the compiler **"emp" is going to refer to the  Employee object that will be created as a result of the *new*keyword**. So, here you create a *reference and an object that the reference variable refers to*.

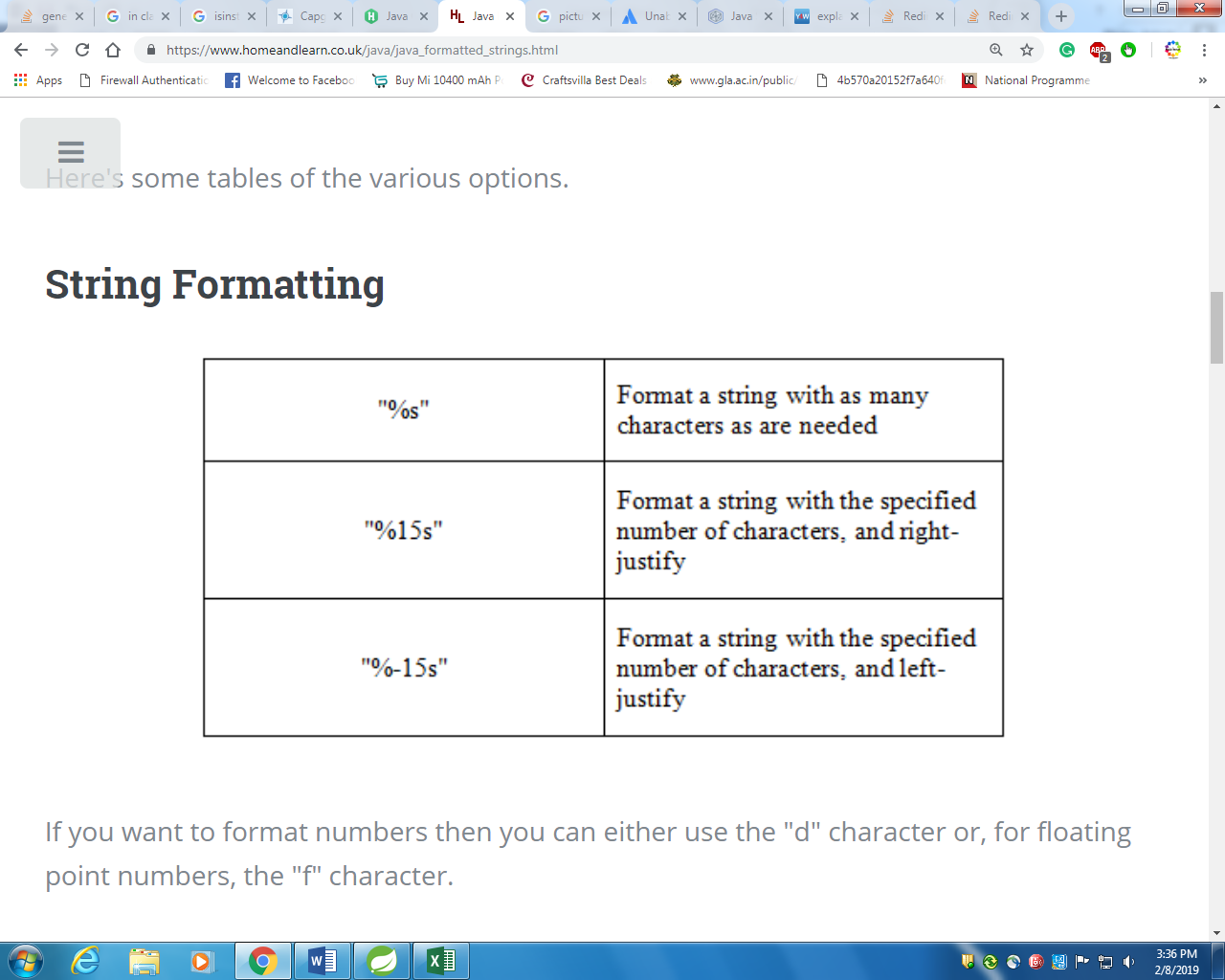


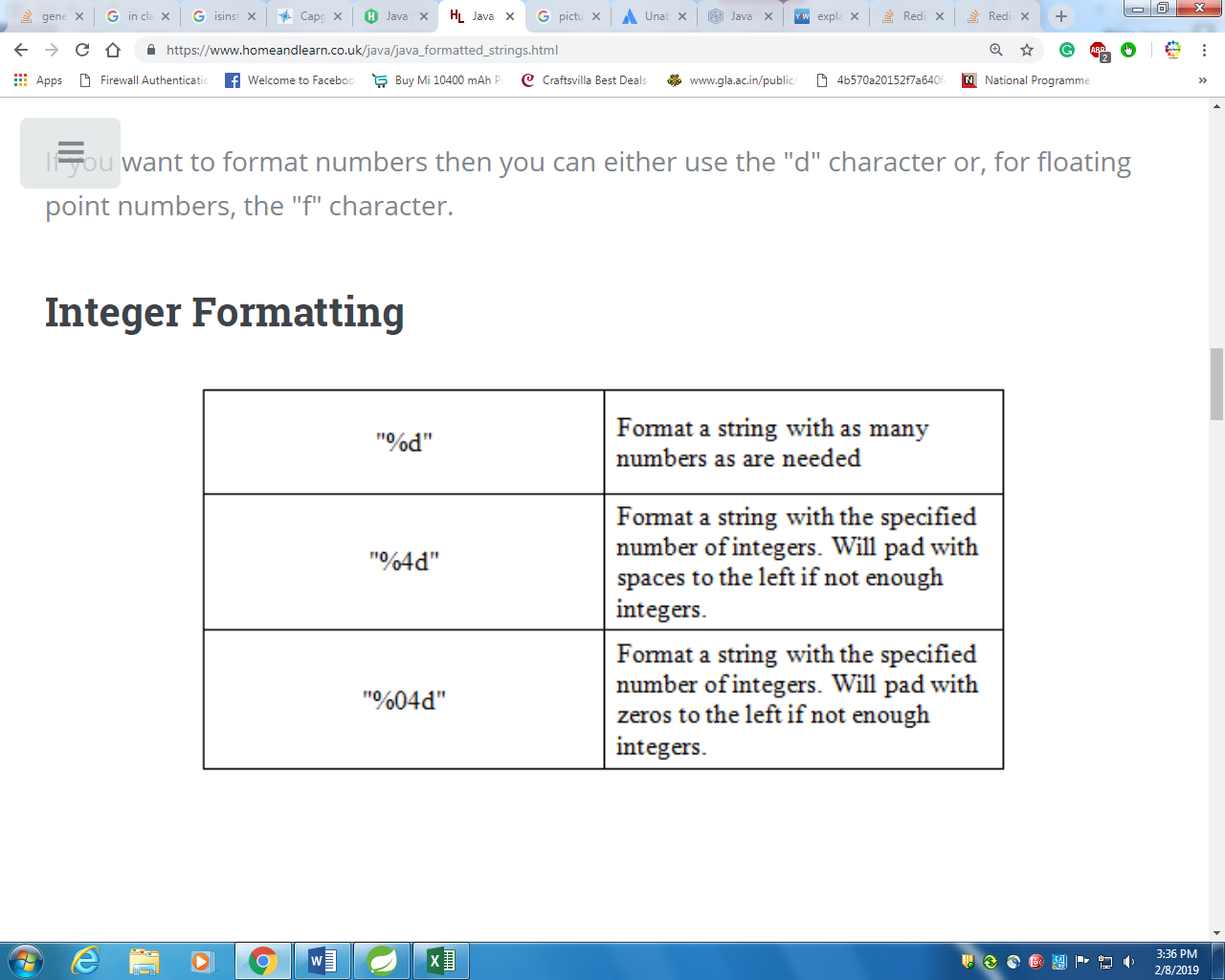
String formatting

**String heading1 = "Exam\_Name";  
String heading2 = "Exam\_Grade";**

**System.out.printf( "%-15s %15s %n", heading1, heading2)**







# [Redirect console output to string in java](https://stackoverflow.com/questions/8708342/redirect-console-output-to-string-in-java)

[Ask Question](https://stackoverflow.com/questions/ask)

51

28

I have one function whose **return type is VOID** and it prints directly on console.

However I need that output in string so that I can work on it.

As I cant make any changes with function with return type is VOID so I have to redirect that output to string.

How can I redirect it in JAVA?

There are many questions regarding redirecting stdout to string but they redirect only input taken from user and not output of some function...

[java](https://stackoverflow.com/questions/tagged/java) [string](https://stackoverflow.com/questions/tagged/string) [redirect](https://stackoverflow.com/questions/tagged/redirect) [console](https://stackoverflow.com/questions/tagged/console) [stdout](https://stackoverflow.com/questions/tagged/stdout)

[share](https://stackoverflow.com/q/8708342)[improve this question](https://stackoverflow.com/posts/8708342/edit)

asked Jan 3 '12 at 5:34

[votes](https://stackoverflow.com/questions/8708342/redirect-console-output-to-string-in-java?answertab=votes#tab-top)

95

If the function is printing to System.out, you can capture that output by using the System.setOutmethod to change System.out to go to a PrintStream provided by you. If you create a PrintStream connected to a ByteArrayOutputStream, then you can capture the output as a String.

Example:

// Create a stream to hold the output

ByteArrayOutputStream baos = new ByteArrayOutputStream();

PrintStream ps = new PrintStream(baos);

// IMPORTANT: Save the old System.out!

PrintStream old = System.out;

// Tell Java to use your special stream

System.setOut(ps);

// Print some output: goes to your special stream

System.out.println("Foofoofoo!");

// Put things back

System.out.flush();

System.setOut(old);

// Show what happened

System.out.println("Here: " + baos.toString());

Just copy above code and use

It is a code of

string baos = System.out.println();

# Generics

public class Main<T>

{T a;

public Main(T data)

{

a= data;

}

public static void main(String[] args) {

Main<String> obj = new Main<String>("asdf");

System.out.println(obj.a);

}

}

Output

Asdf

# Junit

# JUnit tutorial provides basic and advanced concepts of unit testing in java .it is an *open-source testing framework* for java programmers. The java programmer can create test cases and test his/her own code.

# To perform unit testing, we need to create test cases. The unit test case is a code which ensures that the program logic works as expected.

## **types of unit testing**

There are two ways to perform unit testing: 1) manual testing 2) automated testing.

#### **1) Manual Testing**

If you execute the test cases manually without any tool support, it is known as manual testing. It is time consuming and less reliable.

#### **2) Automated Testing**

If you execute the test cases by tool support, it is known as automated testing. It is fast and more reliable.

#### **Annotations for Junit testing**

The Junit 4.x framework is annotation based, so let's see the annotations that can be used while writing the test cases.

**@Test** annotation specifies that method is the test method.

**@Test(timeout=1000)** annotation specifies that method will be failed if it takes longer than 1000 milliseconds (1 second).

**@BeforeClass** annotation specifies that method will be invoked only once, before starting all the tests.

**@Before** annotation specifies that method will be invoked before each test.

**@After** annotation specifies that method will be invoked after each test.

**@AfterClass** annotation specifies that method will be invoked only once, after finishing all the tests.

## **Assert class**

The org.junit.Assert class provides methods to assert the program logic.

#### Methods of Assert class

The common methods of Assert class are as follows:

1. **void assertEquals(boolean expected,boolean actual)**: checks that two primitives/objects are equal. It is overloaded.
2. **void assertTrue(boolean condition)**: checks that a condition is true.
3. **void assertFalse(boolean condition)**: checks that a condition is false.
4. **void assertNull(Object obj)**: checks that object is null.
5. **void assertNotNull(Object obj)**: checks that object is not null.

**assertEquals**

**If the two objects are equal according to their implementation of their equals() method, the assertEquals() method will return normally. Otherwise the assertEquals() method will throw an exception, and the test will stop there.**