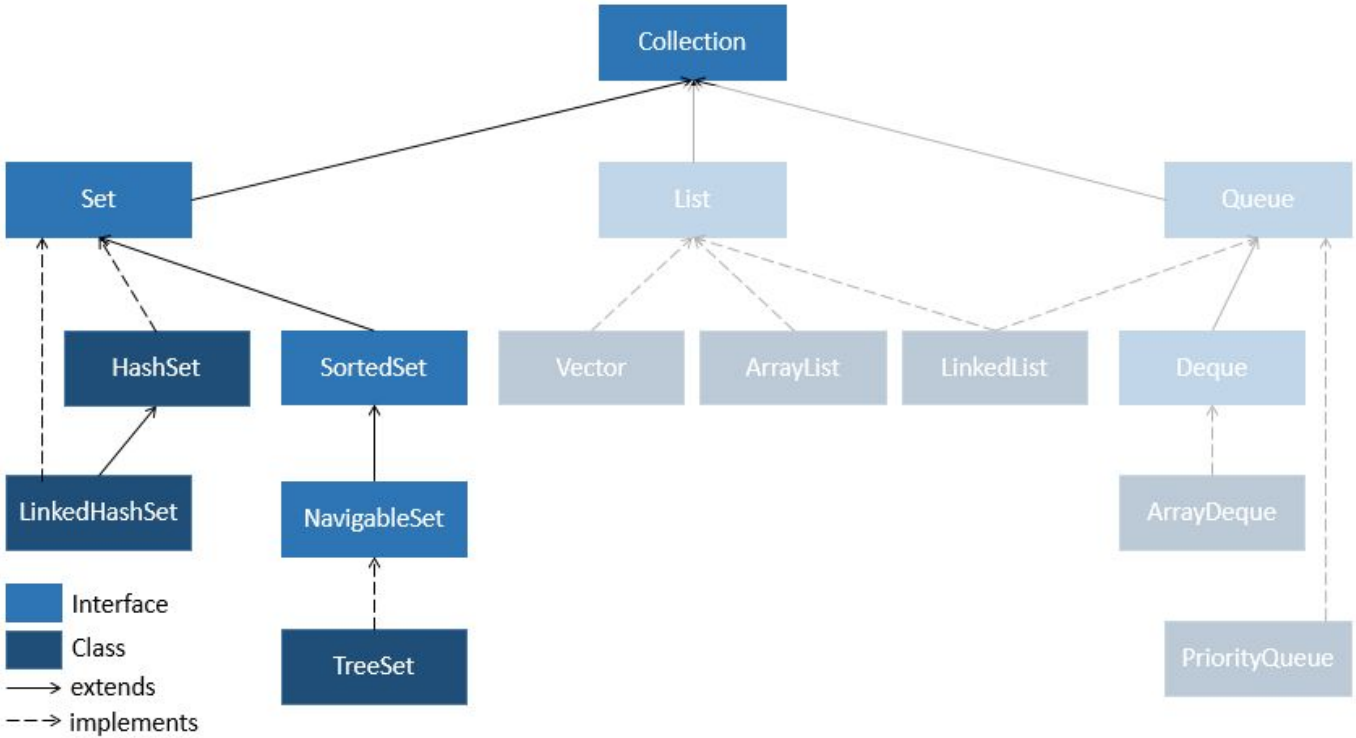
Java™

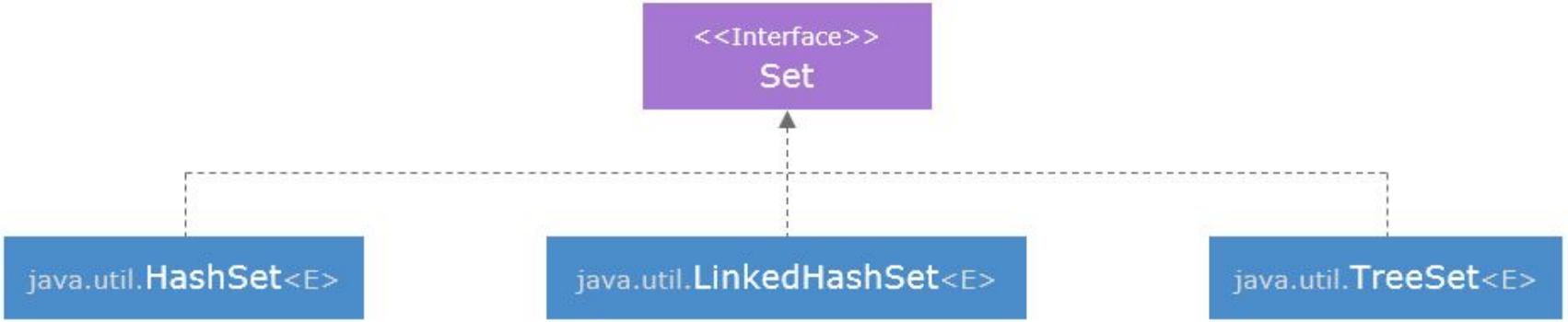
Collection Framework 2


Collection
Set
List
Queue
HashSet
SortedSet
Vector
ArrayList
LinkedList
Deque
LinkedHashSet
NavigableSet
ArrayDeque
Interface
Class
extends
implements
TreeSet
PriorityQueue

Set

- It is Interface present in the java.util package.
- Set represents an unordered collection with **unique** elements.
- Since sets are unordered, they can not be accessed using indexes
- It implementations allow a single null element.

Set Classes



- Uses hash tables for storing elements
- No guarantee in the order of the elements
- Depends on the equals() and hashCode() methods for detecting duplicates

- Uses a combination of hash tables and linked lists for storing elements
- Elements are arranged in the order of their insertion
- Depends on the equals() and hashCode() methods for detecting duplicates

- Uses tree-based data structure for storing elements
- Elements are arranged in their natural ordering
- Depends on the compareTo() method for ordering as well as for detecting duplicates

HashSet

- To store elements HashSet uses hashing mechanism. Hashing involves determining a unique value by using a key. This unique value is called as hashcode. Then this hashcode is used for indexing the data associated with the key.
- HashSet can contains only unique elements.
- When iterated elements in a HashSet are returned in a **random order**. Means HashSet does not guarantee any order.
- We can store Null values in a set.
- It usage equals() and hashCode() for detecting duplicate elements.


```
HashSet<DataTypeClass> set= new HashSet<DataTypeClass>(); // Creating HashSet
```

HashSet Operations

- Adding elements to a HashSet
- Looping over a HashSet
- Iterating over a set
- Converting list to a set

HashSet Methods

Method	Description
`boolean add(E e)`	Adds the specified element to the set if it is not already present.
`boolean remove(Object o)`	Removes the specified element from the set if it is present.
`boolean contains(Object o)`	Returns `true` if the set contains the specified element.
`int size()`	Returns the number of elements in the set.
`void clear()`	Removes all elements from the set.
`boolean isEmpty()`	Returns `true` if the set is empty.
`Iterator<E> iterator()`	Returns an iterator over the elements in the set.
`Object[] toArray()`	Returns an array containing all the elements in the set.
`<T> T[] toArray(T[] a)`	Returns an array containing all elements in the set, in the specified array type.
`boolean addAll(Collection<? extends E> c)`	Adds all elements of the specified collection to the set.
`boolean removeAll(Collection<?> c)`	Removes all elements of the specified collection from the set.

Need of overriding equals() and hashCode() in HashSet

- Set is a collection that automatically removes duplicates when added. What happens if the objects of user-defined are added to a Set?
- The answer lies in the implementation of equals() and hashCode() method of Object class. By default equals() method compares object reference to check if two objects are equal or not.
- We need to Override equals() and hashCode() in user-defined class that checks specific attribute is equal or not on that basis it will decide two Objects are equal or not.

LinkedHashSet

- LinkedHashSet maintains the insertion order, meaning elements are returned in the order they were inserted.
- LinkedHashSet has slightly slower performance compared to HashSet due to its ordering mechanism (doubly linked list).
- It inherit methods from HashSet class.
- It usage same methods that used by HashSet Class.
- It also usage equals() and hashCode() for detecting duplicate elements.

TreeSet

- It uses tree-based data structure for storing elements.
- It maintains elements in sorted (natural) order.
- For user defined class objects we need to provide custom **Comparable or Comparator.**
- It has slower performance compared to HashSet as it uses a Red-Black tree for maintaining sorted order.
- It depends on the compareTo() or compare() method for ordering as well as for detecting duplicates.
- TreeSet does not allow null elements, as it uses comparison operations.

TreeSet Methods

Some methods exclusive to the TreeSet class:

Method	Description
E first()	From the Set, it returns the first (lowest) element.
E last()	From the Set, it returns the last (highest) element.
E ceiling(E element)	The least element either >= given element is returned. Null is returned if empty.
E floor(E element)	Greatest element either <= given element is returned. Null is returned if empty.

Sorting using Comparable Interface

- Comparable Interface is from **java.util** package which has only one method compareTo(Object).

  public int compareTo(<T> object)

- String class and Wrapper class implements Comparable Interface by default.
- This method returns an integer and if it returns -
  - zero, it means the current object is equal to the passed object
  - positive integer, it means the current object is greater than the passed object
  - a negative integer means the current object is lesser than the passed object.

Sorting using Comparator Interface

- Comparator Interface is from java.util package which has only one method compareObject).

  int compare(T o1, T o2);

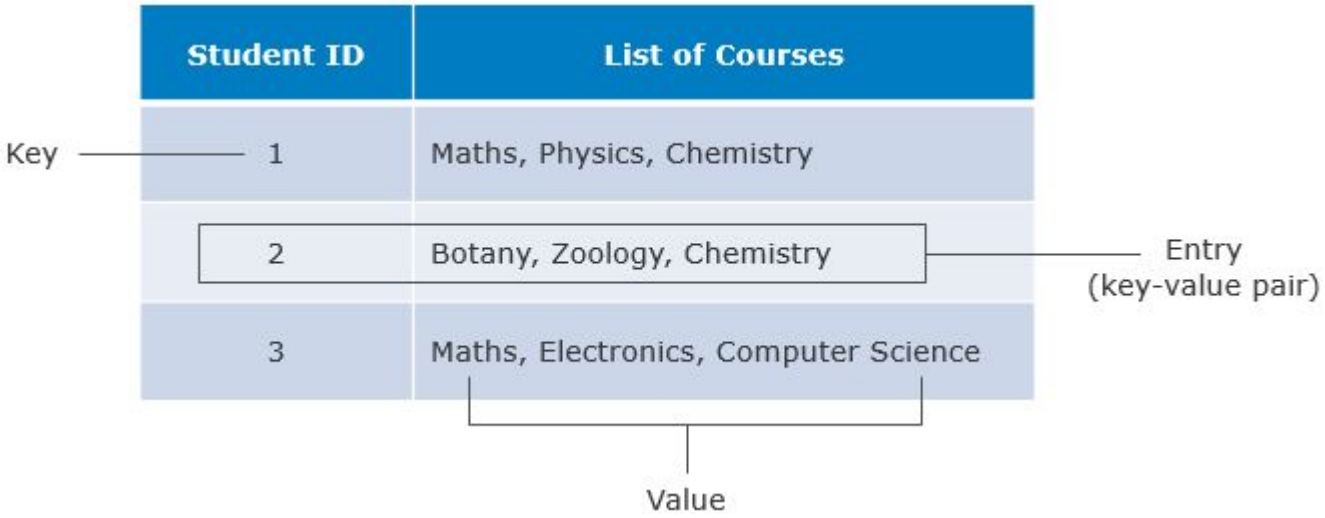
- This method returns an integer and if it returns -
  - zero, it means the current object is equal to the passed object
  - positive integer, it means the current object is greater than the passed object
  - a negative integer means the current object is lesser than the passed object.

HashSet vs LinkedHashSet vs TreeSet

Feature	HashSet	LinkedHashSet	TreeSet
Ordering	No guarantee of order.	Maintains insertion order.	Sorted in natural order (or custom comparator).
Implementation	Backed by a `HashMap`.	Backed by a `LinkedHashMap`.	Backed by a `TreeMap` (Red-Black Tree).
Duplicates	Does **not** allow duplicates.	Does **not** allow duplicates.	Does **not** allow duplicates.
Null Values	Allows a single `null` value.	Allows a single `null` value.	Does **not** allow `null` values.
Ordering Guarantee	No guarantee (random order).	Elements retain the order of insertion.	Sorted by natural/comparator order.
Memory Usage	Lower memory usage.	Higher memory usage due to maintaining order.	Higher due to tree structure.
Best Use Case	When order is not important and fast lookup is needed.	When order of insertion is required.	When elements need to be sorted.
Sorting Support	No sorting.	No sorting.	Sorted elements.

Map

Real life example



Map

- A Map represents a mapping between a key and a value, allowing data to be stored in key-value pairs.
- It is similar to a dictionary, where lookup is performed based on a key, and the respective value is returned.
- Example: In a bookstore, the key can be a book name, and the value can be a shelf number, allowing quick book searches.
- In Java, Map is an interface defined as:
- It belongs to the java.util package.

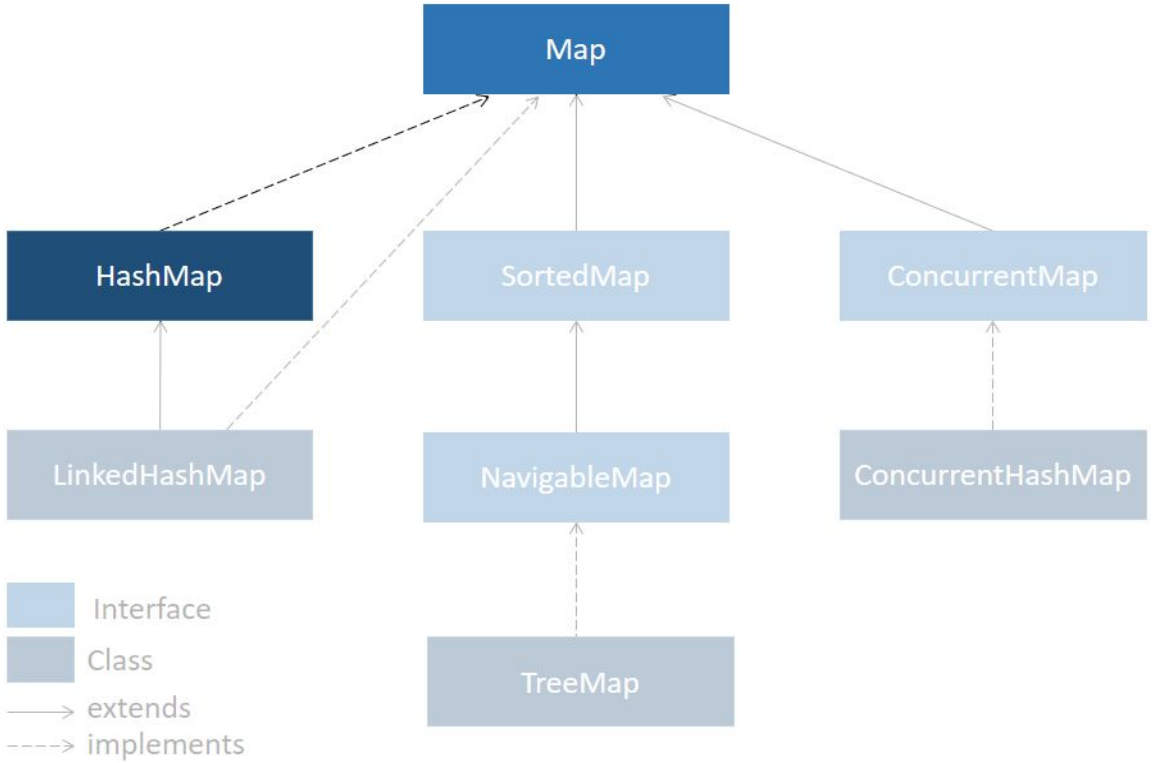
public interface Map<K, V>

K stands for the key type (must be unique).

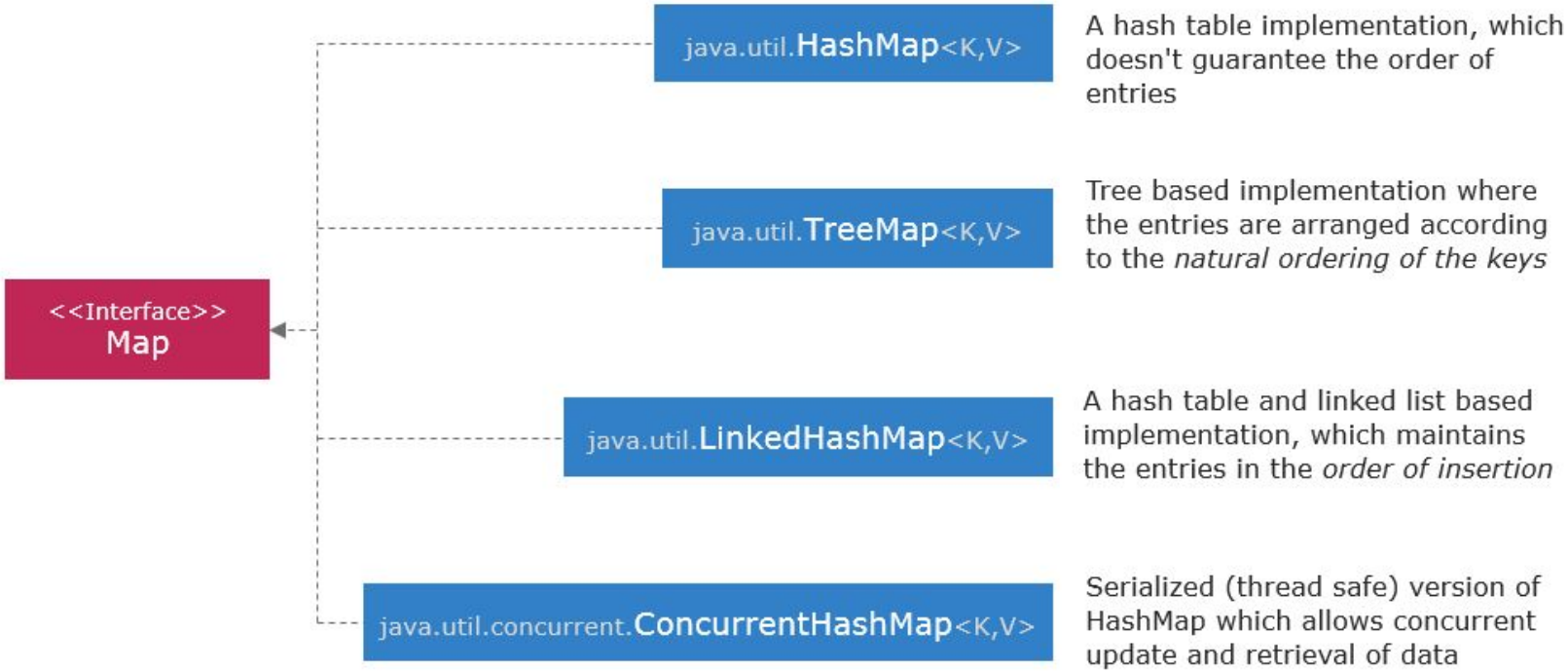
V stands for the value type (can have duplicates).

- Keys must be unique, but values can be duplicate.

Map Hierarchy



Map Classes


<<Interface>>
Map
java.util.HashMap<K,V>
A hash table implementation, which doesn't guarantee the order of entries
java.util.TreeMap<K,V>
Tree based implementation where the entries are arranged according to the *natural ordering of the keys*
java.util.LinkedHashMap<K,V>
A hash table and linked list based implementation, which maintains the entries in the *order of insertion*
java.util.concurrent.ConcurrentHashMap<K,V>
Serialized (thread safe) version of HashMap which allows concurrent update and retrieval of data

HashMap

- A HashMap stores data in key-value pairs, where the key must be unique, but values can be duplicated.
- **Since HashMap does not maintain any specific order, retrieving elements may return them in an random sequence.**
- Allows one null key and multiple null values.
- It Implements the Map interface and belongs to the java.util package.


```
HashMap<String, Integer> map1 = new HashMap<String, Integer>(); // String key, Integer value

Map<Integer, String> map2 = new HashMap<Integer, String>();    // Integer key, String value
```

HashMap Methods

Method	Description	Example Usage
`put(K key, V value)`	Adds a key-value pair to the map or updates the value for an existing key.	`map.put(1, "One");`
`get(Object key)`	Retrieves the value associated with the specified key.	`map.get(1);` // Returns `"One"`
`remove(Object key)`	Removes the key-value pair for the specified key.	`map.remove(1);`
`containsKey(Object key)`	Checks if the specified key exists in the map.	`map.containsKey(5);` // Returns `true/false`
`containsValue(Object value)`	Checks if the specified value exists in the map.	`map.containsValue("Five");`
`size()`	Returns the number of key-value mappings in the map.	`map.size();` // Returns number of entries
`keySet()`	Returns a set of all keys in the map.	`Set<Integer> keys = map.keySet();`
`values()`	Returns a collection of all values in the map.	`Collection<String> values = map.values();`
`clear()`	Removes all key-value mappings from the map.	`map.clear();`

Iterating over a HashMap

1. Using keySet()

```
Set<Integer> keys = books.keySet();

for(Integer key:keys) {

        String value = map.get(key);

        System.out.println(value);

}
```


2. Using values()

```
Collection<String> cvalues = books.values();

for(String temp:cvalues) {

        System.out.println(temp); }
```

Iterating over a HashMap Cont.

3. Using entrySet()

```
//Getting set of entries using entrySet()
Set<Entry<Integer,String>> valueset = books.entrySet();
//iterating over set
for(Entry<Integer,String> obj:valueset) {
        System.out.println("Key-"+obj.getKey()+" : Value-"+obj.getValue());
}
```

LinkedHashMap

- LinkedHashMap is a class that implements the Map interface, providing a hash table and linked list implementation for storing key-value pairs.
- **It maintains the order of insertion (or access order), unlike HashMap which does not guarantee any order.**
- Allows One Null Key and Multiple Null Values: Like HashMap.
- Internally uses a doubly linked list to maintain the order of the elements.
- Maintaining Insertion Order when the order of entries matters.


```
Map<Integer,String> lmap = new LinkedHashMap<Integer,String>();
```

TreeMap

- TreeMap is a class that implements the Map interface and stores key-value pairs in a sorted order according to the natural ordering of the keys or by a specified comparator or Comparable.
- Internally, a TreeMap uses a Red-Black Tree to store keys,
- Unlike HashMap or LinkedHashMap, TreeMap does not allow null keys (but it allows null values).
- Useful when you need keys to be automatically sorted, such as in implementing a navigation system or when the order of elements is significant.


```
Map<Integer,String> tmap = new TreeMap<Integer,String>();
```

TreeMap Methods

Method	Description	Example Usage
`firstKey()`	Returns the first (lowest) key in the map.	`K firstKey = map.firstKey();`
`lastKey()`	Returns the last (highest) key in the map.	`K lastKey = map.lastKey();`
`headMap(K toKey)`	Returns a view of the portion of the map whose keys are strictly less than `toKey`.	`map.headMap(5);`
`tailMap(K fromKey)`	Returns a view of the portion of the map whose keys are greater than or equal to `fromKey`.	`map.tailMap(5);`
`subMap(K fromKey, K toKey)`	Returns a view of the portion of the map whose keys range from `fromKey` to `toKey`.	`map.subMap(3, 7);`

Choosing Right Collection

- Choose the right collection based on the requirement to improve performance. Some of the points to be considered are:
- If duplicate elements are not allowed to choose Set otherwise choose List.
- ArrayList is quicker than LinkedList to randomly access elements.
- For quick removal and addition, LinkedList is better than ArrayList.
- Use collections such as TreeMap, TreeSet when elements in the collection are required to be sorted and ordered.
- Use concurrent collections to support concurrent access.

Generics

- Generics are used for creating **interfaces, methods, and classes** that specify the object type on which they work as a parameter.
- **Generics Class**

  class class-name<type-parameter-list> { }
- The type-parameter-list indicates the type parameters. Usually, the type parameter is defined by a single letter of capital and is usually one of the E (element), T (type), K (key), N(number) and V (value)

Generics Contd.

```
class Record<E> {
    private E[] record;
    public E add(E item) {
        // Code to add record item
    }
    public E get(int index) {
        // Code to get record at specified index
    }
}
```


Record class is a generic class.

```
Record<Integer> integerRecord = new Record<Integer>();
Record<String> stringRecord = new Record<String>();
Record<Professor> professorRecord = new Record<>();    // In Java 7 or later versions
```

Generics Contd.

Generics also have certain restrictions:

- We should not create an instance for the type parameter.
- E e = new E(); // Compilation error
- java.lang.Throwable class cannot be extended by the Generic class.
- A generic type with primitive type cannot be used. It must always be used with a reference type.

Assignments

1. Write a program to sort a list of employees. Each employee has the following attributes: id (Integer), name (String), age (Integer), salary (Double)
   a. Implement the Comparable interface to sort employees by id (natural ordering).

2. Write a program to managing student records using a HashSet. Each student has the following attributes: studentId (Integer), name (String), age (Integer), grade (String). Since a student should be considered unique based on their studentId, you need to override the equals() and hashCode() methods correctly to ensure uniqueness in the HashSet.

3. Write a program to create student class that has the following attributes: id (Integer), name (String), age (Integer), salary (Double). Now create HashSet, LinkedHashSet and TreeSet and Store student object in it and display all 3 sets data.

Assignments

4. Develop a system to manage bank accounts using LinkedHashSet. Each account should have the following attributes: accountNumber (Long), accountHolder (String), balance (Double), accountType (String). Implement the equals() and hashCode() methods to ensure unique accounts based on accountNumber. Implement methods to:

1. Add new accounts
2. Display accounts in the order they were inserted
3. Update account balance

Thanks