NPTEL ONLINE CERTIFICATION COURSES

Data Structures and Algorithms Using Java

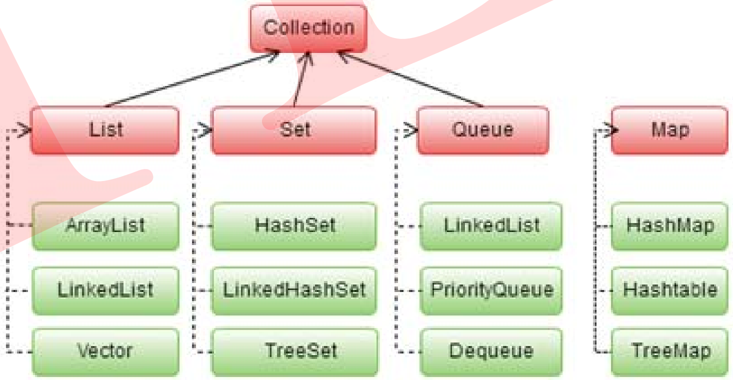
Debasis Samanta
Department of Computer Science & Engineering, IIT Kharagpur

Module 03: Java Collection Framework

Lecture 06 : Basics of the JCF

CONCEPTS COVERED

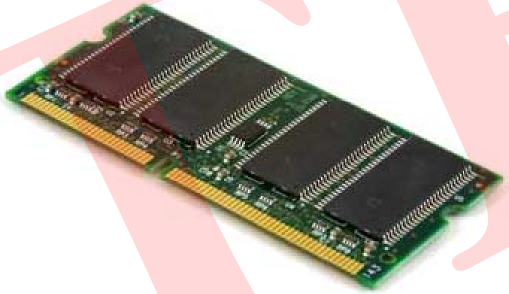
- About Data Structures
- Java Supports for Data Structures
  - Java Collection
  - Java Map
- Collection Framework
- Map Framework
- Java Legacy Classes

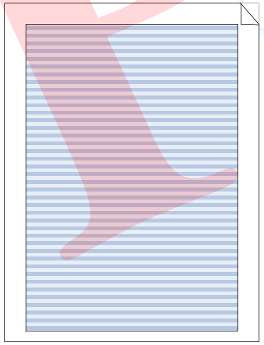


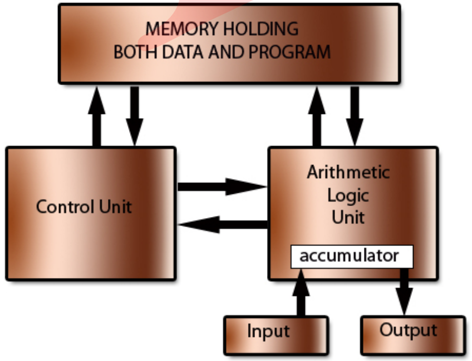
About Data Structures

Different data structures

- Computing is to manipulate data

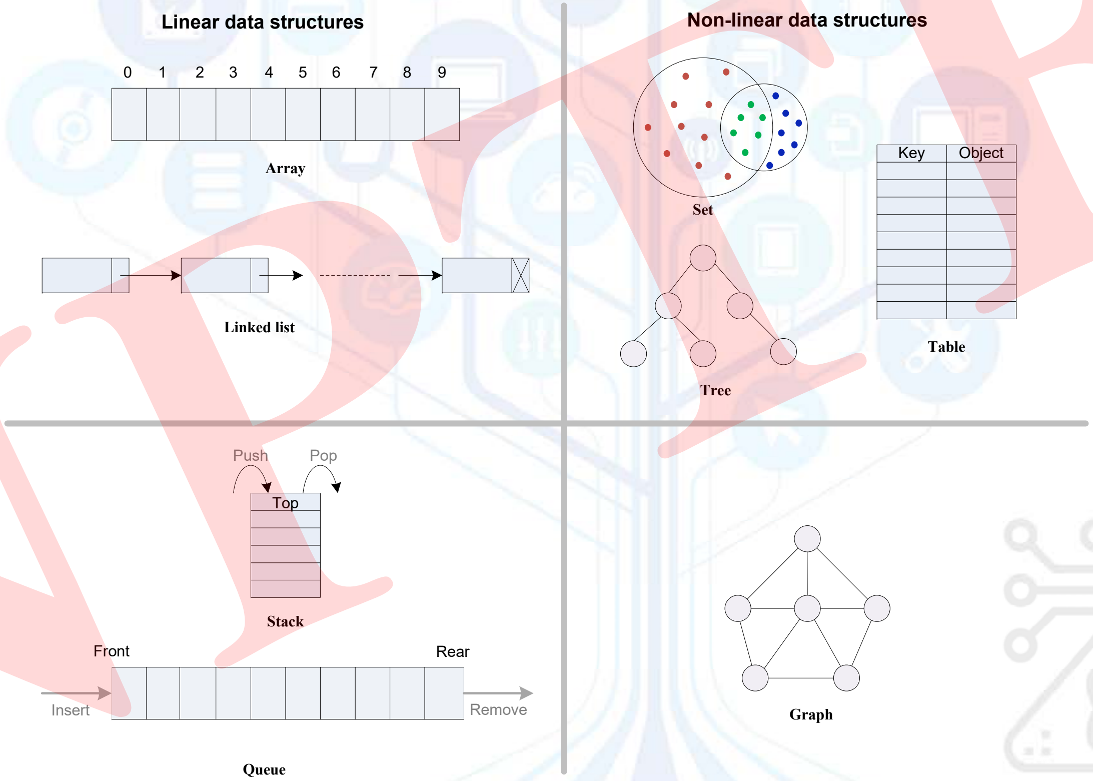






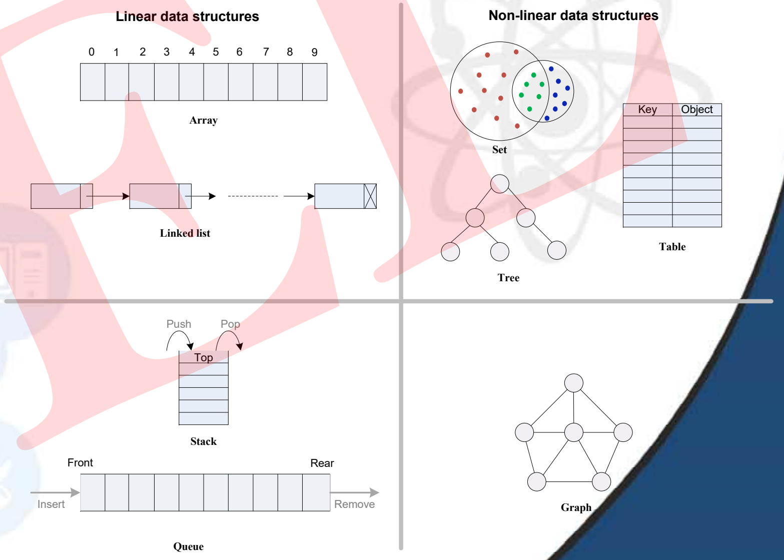
Different data structures

- There are several data structures known in the field of Computer Science.



Different data structures

- There are several data structures known in the field of Computer Science.
- All the data structures can be broadly classified into two categories:
  - Linear data structures
    - array, linked list, stack and queue
    - Linear data structures can be classified as indexed or sequential
    - Indexed: For example, array is an indexed data structures
    - Sequential: linked list is a sequential data structures
    - Stack and queue can be realized as indexed and as well as sequential data structures.
  - Non-linear data structures
    - For example, set, tree, table, graph, etc.



Java Supports for Data Structures

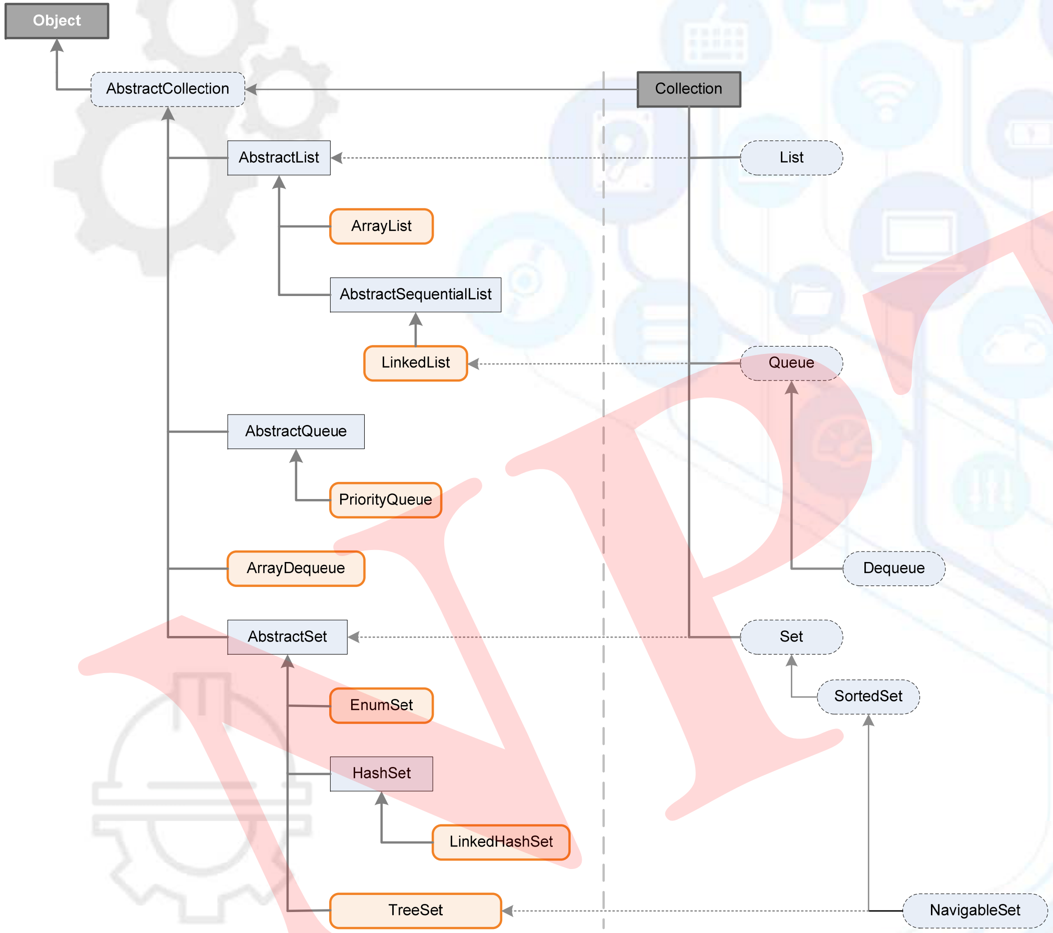
Java supports for data structures

- All the data structures as mentioned are called basic data structures
- Other any complex data structures can be realized with them.
- Since, data structures are important to build any software system (because together algorithm and data structures are used to develop programs), Java developer elegantly supports a good library of built-in data structures utilities.
- In Java, a concept has been introduced called **collection**.

What is a collection?

- A collection in Java is a group of objects (of any type).
- The java.util package contains one of Java's most powerful sub systems called collections framework.
- It is defined in `java.utl` package.
  - The package is a huge collection of interfaces and classes that provide state-of-the-art technology for managing groups of objects.
  - It is very popular among the programmers and software practitioners.

Java Collection Framework (JCF)



- Popularly abbreviated as JCF.
  - The java.util package was first time introduced in Java 2 release.
  - Prior to the release of Java 2, Java supported ad hoc classes such as `Dictionary, Vector, Stack, and Properties` to manipulate collection of objects.

Why collection framework?

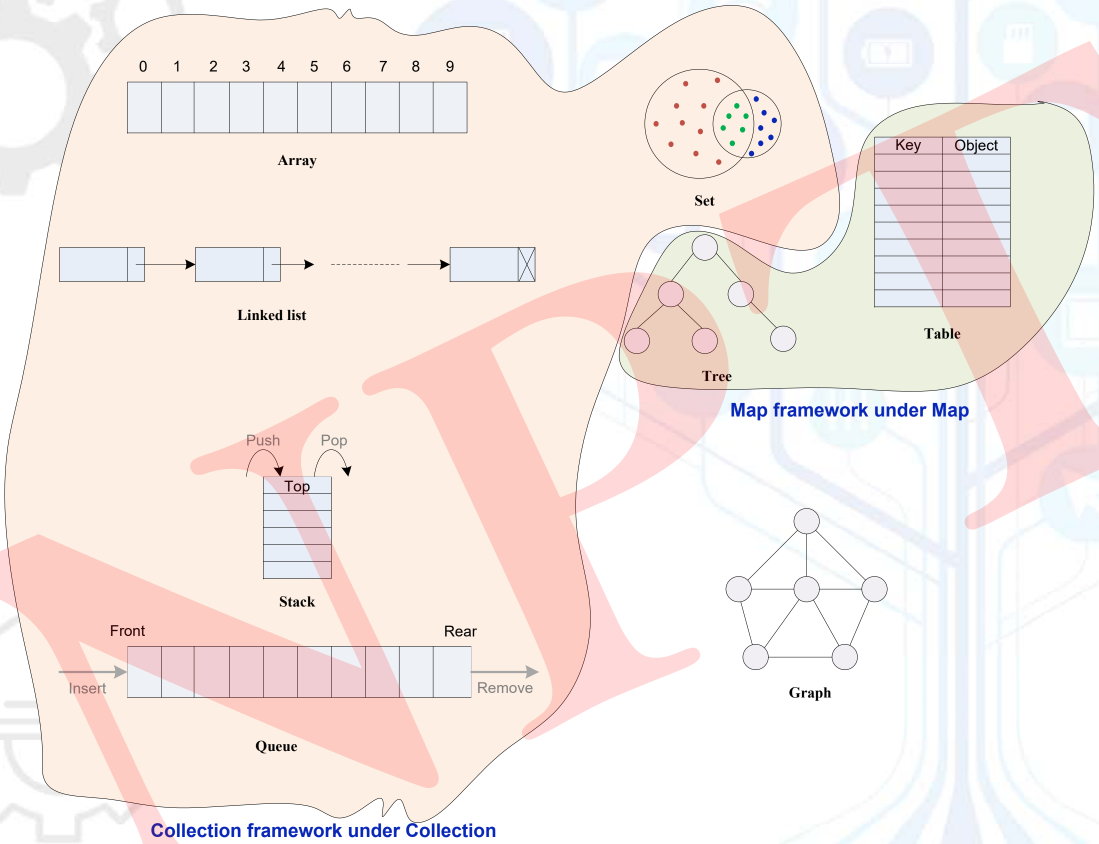
- The JCF has been introduced to meet several goals. Some of the major goals are listed in the following.

1. The framework provides high-performance software coding.
   - The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hash tables) are highly efficient. You seldom, if ever, need to code one of these "data engines" manually.
2. The framework allows different types of collections to work in a similar manner and with a high degree of interoperability.
3. Extending and/or adapting a collection is easy and flexible.

The framework

- The entire JCF consists of two parts:
  1. Collections are under `Collection`
  2. Facilities under `Map`

The framework

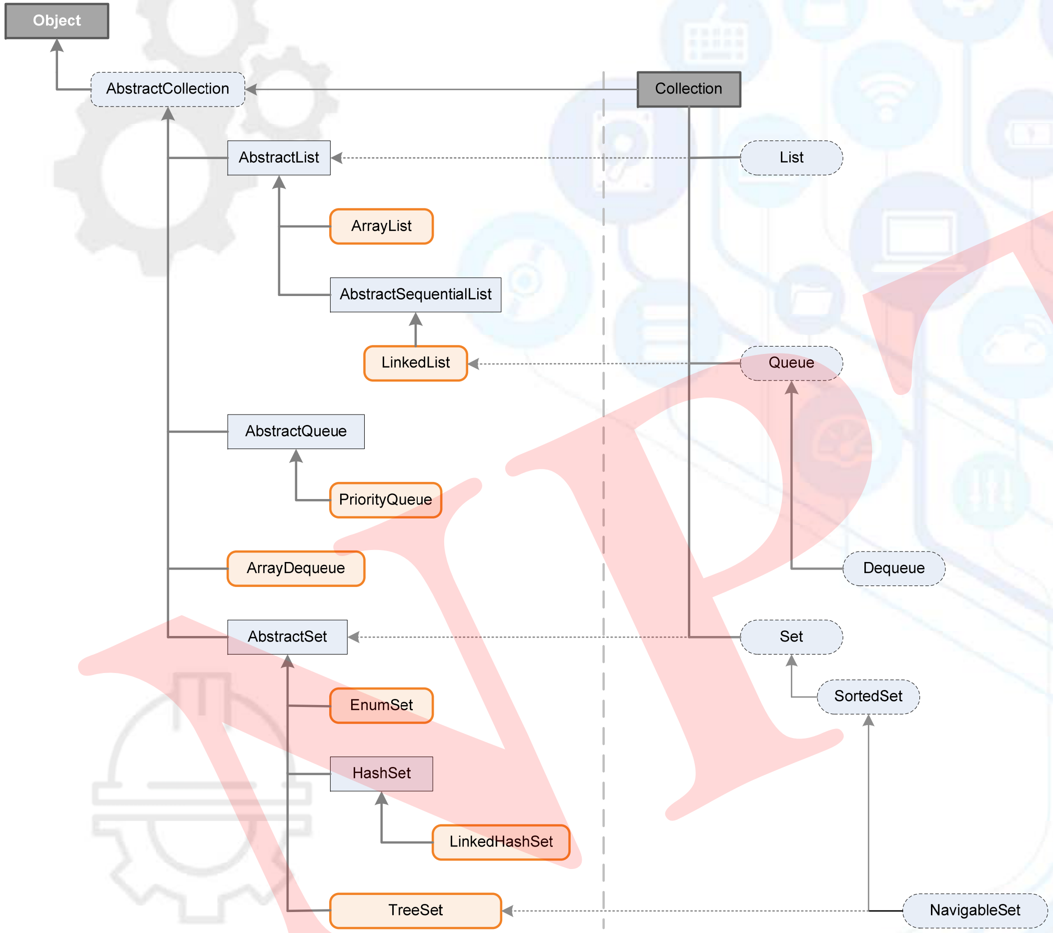


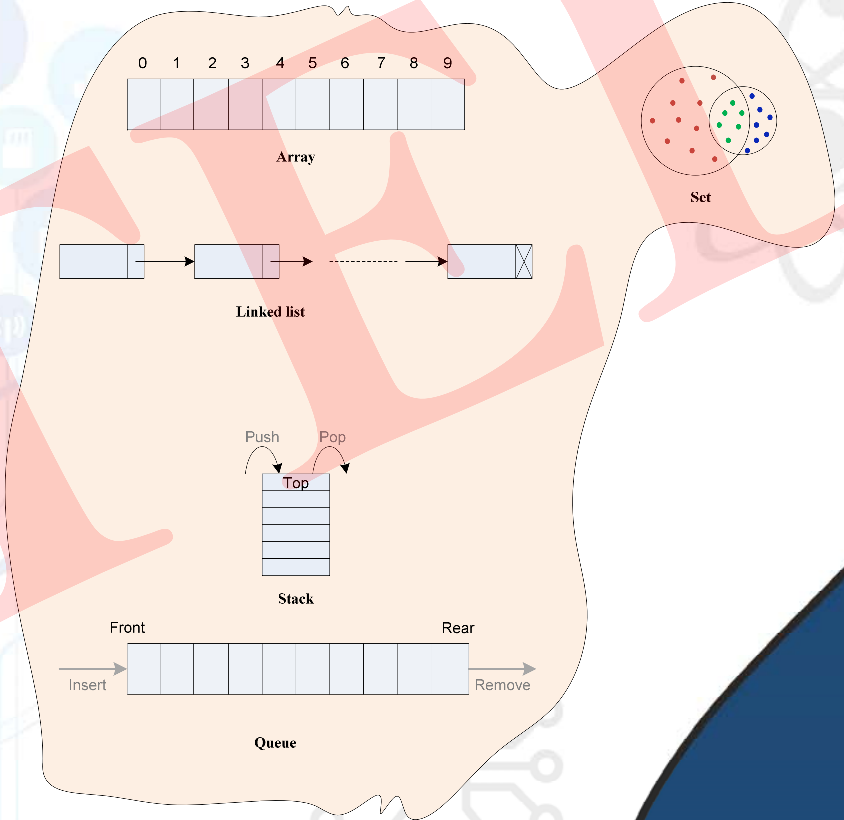
Note:

- There is no explicit facility for graph data structure.

Collection Framework

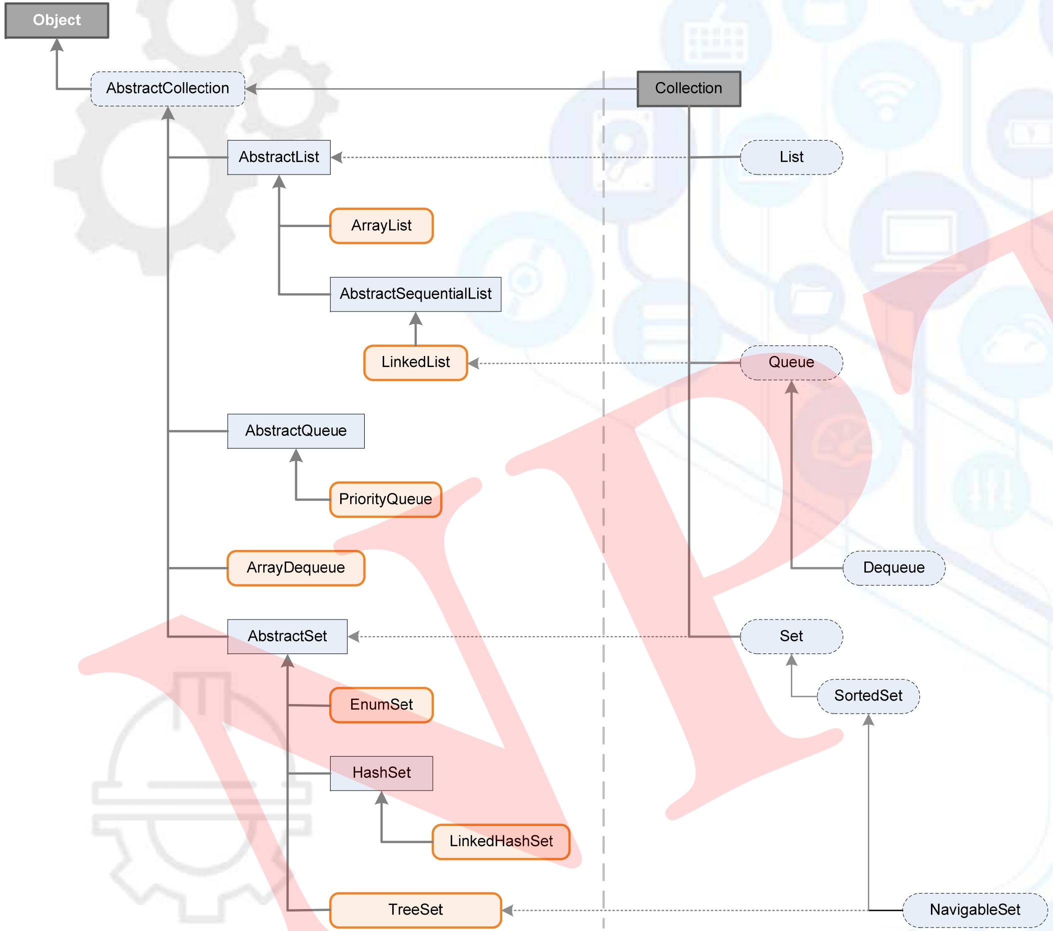
The framework : Collection





Collection framework under Collection

The framework : Collection



↑ extends

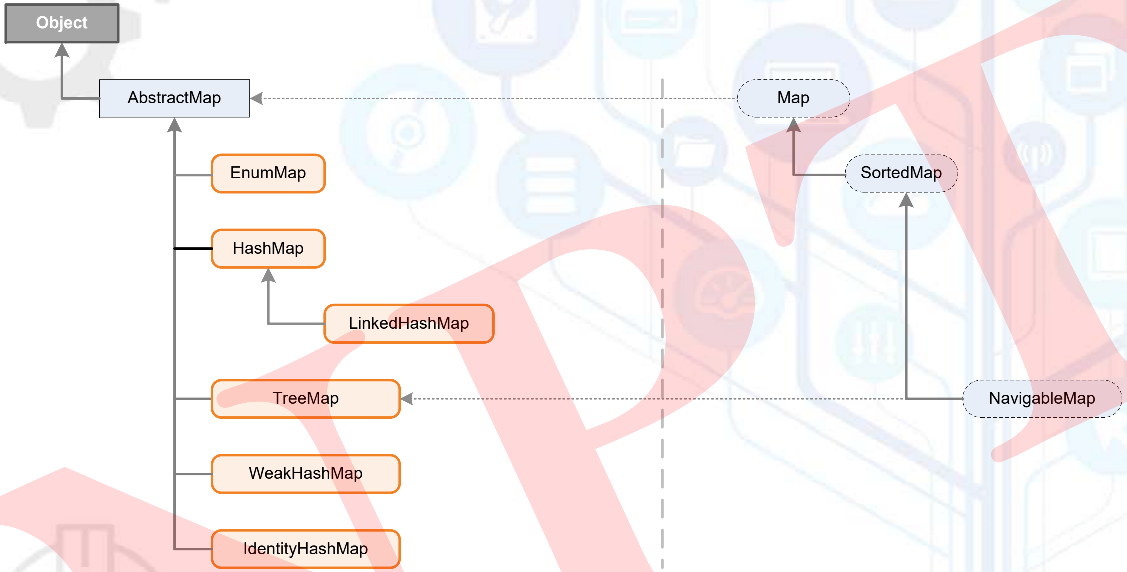
⇡ implements

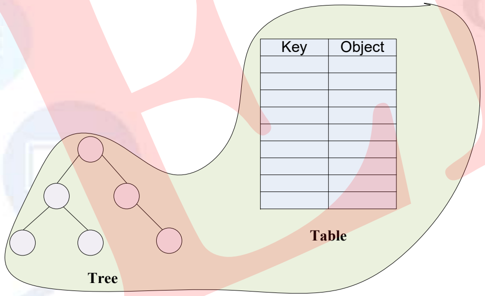
interface

class

Map Framework

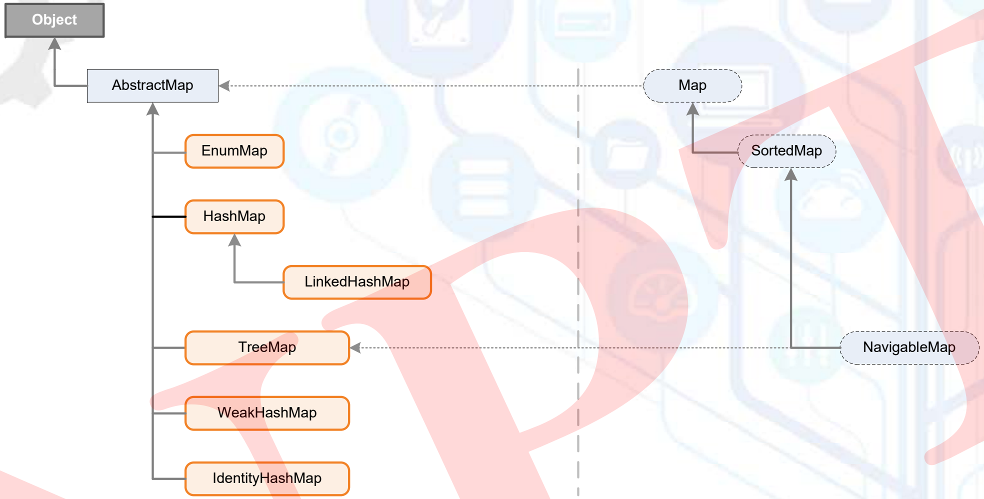
The framework : Map

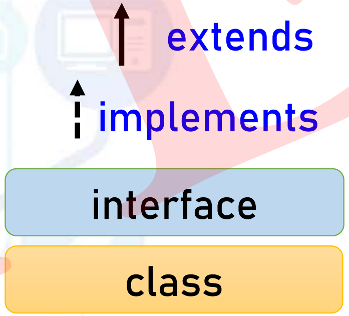




Map framework under Map

The framework : Map





Java Legacy Classes

Java legacy classes and interfaces

- The `java.util` package was first time introduced in Java 2 release and becomes a more powerful subsystem for a programmer today.

- Prior to the release of Java 2, Java supported ad hoc classes to manipulate collection of objects :

  - Dictionary, Hashtable, Vector, Stack, and Properties

Java legacy classes and interfaces

- With the inclusion of the Java collection framework, several of the original classes were reengineered to support the collection interface.
- In other words, none of the old classes have been deprecated, rather, they are still fully compatible with the Java Collection framework and there is still code that use them.
- Such classes are called legacy classes.

Dictionary | Hashtable | Properties | Stack | Vector

- There is one legacy interface called Enumeration.

REFERENCES

- https://cse.iitkgp.ac.in/~dsamanta/javads/index.html
- https://docs.oracle.com/javase/tutorial/

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

1951

योगः कर्मसु कौशलम

THANK YOU !

NPTEL ONLINE CERTIFICATION COURSES

Data Structures and Algorithms Using Java

Debasis Samanta

Department of Computer Science & Engineering, IIT Kharagpur

Module 03: Java Collection Framework

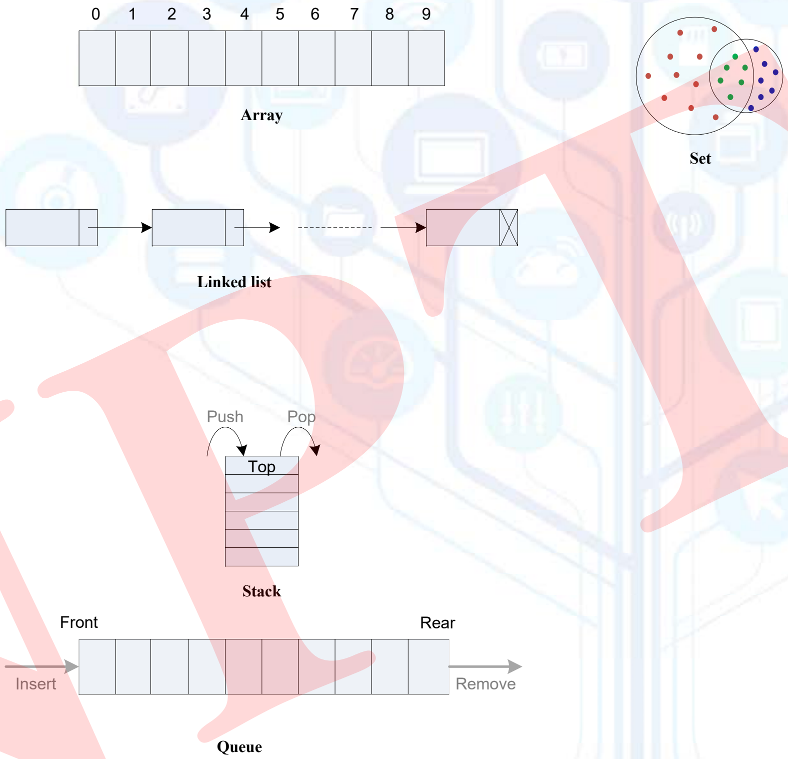
Lecture 07 : Collection in JCF

CONCEPTS COVERED

- Constituents of Collection of JCF
- Interfaces
- Classes
  - Constructors
  - Methods

Constituents of Collection

Collections of JCF

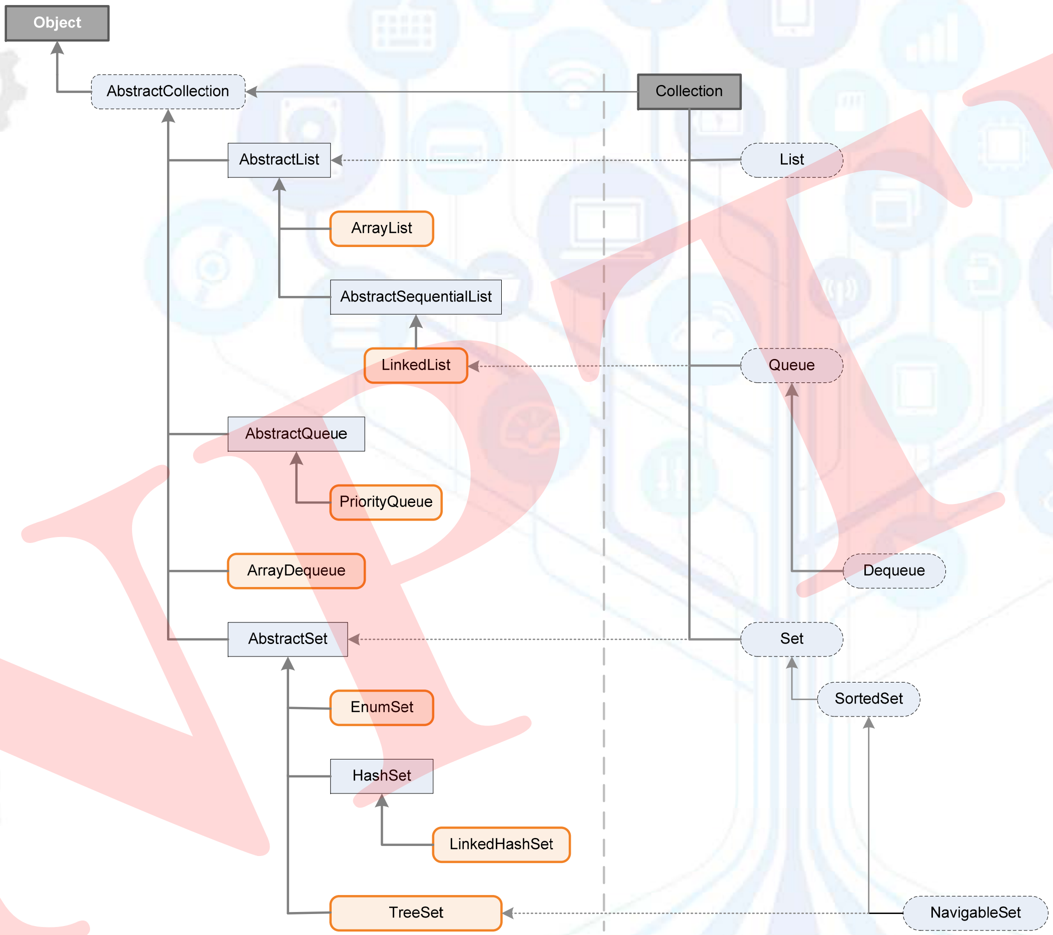


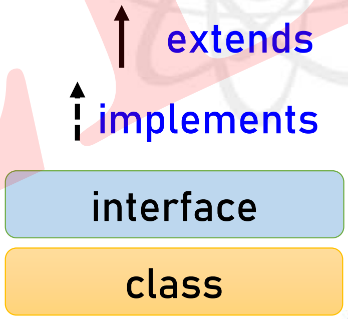
Collections under Collection

Collections of JCF

- **A collection** 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.
- The hierarchy of the classes and interfaces in JCF is quite complex.
- The entire Java Collections Framework (JCF) is built upon a set of standard interfaces, classes and algorithms.
  - **Interfaces:**
    Set, List, Queue, Deque
  - **Classes:**
    ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet

Java collection hierarchy





Interfaces Collection

Interfaces of collections

Interface	Description
Collection	Enables you to work with groups of objects; it is at the top of the collections hierarchy.
List	List extends **Collection** to handle sequences (lists of objects).
Queue	Queue extends **Collection** to handle special types of lists in which elements are removed only from the head.
Deque	Deque extends **Queue** to handle a double-ended queue
Set	Extends **Collection** to handle sets, which must contain unique elements.
SortedSet	Extends **Set** to handle sorted sets.
NavigableSet	NavigableSet extends **SortedSet** to handle retrieval of elements based on closest-match

Table 7.1: Interfaces in collections framework

Interfaces in Collection

Collection interface

- The Collection interface is the foundation upon which the collections framework is built because it must be implemented by any class that defines a collection.

- Collection is a generic interface that has this declaration:

```
interface Collection<T>
```

Here, T specifies the type of objects that the collection will hold.

Methods declared in Collection

- Collection declares the core methods that all collections will have.
- Because all collections implement Collection, familiarity with its methods is necessary for a clear understanding of the framework.
- These methods are summarized in Table 7.2.

Method	Description
boolean add(T *obj*)	Adds *obj* to the invoking collection. Returns **true** if *obj* was added to the collection. Returns **false** if *obj* is already a member of the collection and the collection does not allow duplicates.
boolean addAll(Collection<? extends T> *c*)	Adds all the elements of *c* to the invoking collection. Returns **true** if the collection changed (i.e., the elements were added). Otherwise, returns **false**.
void clear( )	Removes all elements from the invoking collection.
boolean contains(Object *obj*)	Returns **true** if *obj* is an element of the invoking collection. Otherwise, returns **false**.
boolean containsAll(Collection<?> *c*)	Returns **true** if the invoking collection contains all elements of *c*. Otherwise, returns **false**.

Table 7.2: The methods declared in Collection interface (*continued…*)

Methods declared in **Collection**

Method	Description
boolean equals(Object *obj*)	Returns **true** if the invoking collection and *obj* are equal. Otherwise, returns **false**.
int hashCode( )	Returns the hash code for the invoking collection.
boolean isEmpty( )	Returns **true** if the invoking collection is empty. Otherwise, returns **false**.
Iterator<T> iterator( )	Returns an iterator for the invoking collection.
default Stream<E> parallelStream( )	Returns a stream that uses the invoking collection as its source for elements. If possible, the stream supports parallel operations.
boolean remove(Object *obj*)	Removes one instance of *obj* from the invoking collection. Returns **true** if the element was removed. Otherwise, returns **false**.
boolean removeAll(Collection<?> *c*)	Removes all elements of *c* from the invoking collection. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
default boolean removeIf(Predicate <? super T> *p*)	Removes from the invoking collection those elements that satisfy the condition specified by *predicate*.
boolean retainAll(Collection<?> *c*)	Removes all elements from the invoking collection except those in *c*. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.

Table 7.2: The methods declared in Collection interface (*continued…*)

Methods declared in Collection

Method	Description
int size( )	Returns the number of elements held in the invoking collection.
default Spliterator<E> spliterator( )	Returns a spliterator to the invoking collections.
default Stream<E> stream( )	Returns a stream that uses the invoking collection as its source for elements. The stream is sequential.
Object[ ] toArray( )	Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.
<T> T[ ] toArray(T *array*[ ])	Returns an array that contains the elements of the invoking collection. The array elements are copies of the collection elements. If the size of *array* equals the number of elements, these are returned in *array*. If the size of *array* is less than the number of elements, a new array of the necessary size is allocated and returned. If the size of *array* is greater than the number of elements, the array element following the last collection element is set to **null** and an error is reported.

Table 7.2: The methods declared in Collection interface

Interface List

Interfaces of collections

Interface	Description
Collection	Enables you to work with groups of objects; it is at the top of the collections hierarchy.
List	List extends **Collection** to handle sequences (lists of objects).
Queue	Queue extends **Collection** to handle special types of lists in which elements are removed only from the head.
Deque	Deque extends **Queue** to handle a double-ended queue
Set	Extends **Collection** to handle sets, which must contain unique elements.
SortedSet	Extends **Set** to handle sorted sets.
NavigableSet	NavigableSet extends **SortedSet** to handle retrieval of elements based on closest-match

Interface List

- The List interface extends Collection and declares the behavior of a collection that stores a sequence of elements. Elements can be inserted or accessed by their position in the list, using a zero-based index.
- A list may contain duplicate elements.
- List is a generic interface that has this declaration:

```
interface List<T>
```

Here, T specifies the type of objects that the list will hold.

- In addition to the methods defined by Collection, List defines some of its own, which are summarized in Table 7.3.

Methods declared in List

Method	Description
void add(int *index*, E *obj*)	Inserts *obj* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
boolean addAll(int *index*, Collection<? extends E> *c*)	Inserts all elements of *c* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns **true** if the invoking list changes and returns **false** otherwise.
E get(int *index*)	Returns the object stored at the specified index within the invoking collection.
int indexOf(Object *obj*)	Returns the index of the first instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.
int lastIndexOf(Object *obj*)	Returns the index of the last instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.
ListIterator<E> listIterator( )	Returns an iterator to the start of the invoking list.

Table 7.3: The methods declared in List interface (*continued…*)

Methods declared in List

Method	Description
ListIterator<E> listIterator(int *index*)	Returns an iterator to the invoking list that begins at the specified *index*.
E remove(int *index*)	Removes the element at position *index* from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.
default void replaceAll(UnaryOperator<E> *opToApply*)	Updates each element in the list with the value obtained from the *opToApply* function.
E set(int *index*, E *obj*)	Assigns *obj* to the location specified by *index* within the invoking list. Returns the old value.
default void sort(Comparator<? super E> *comp*)	Sorts the list using the comparator specified by *comp*.
List<E> subList(int *start*, int *end*)	Returns a list that includes elements from *start* to *end*–1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

Table 7.3: The methods declared in List interface

Interface Queue

Interfaces of collections

Interface	Description
Collection	Enables you to work with groups of objects; it is at the top of the collections hierarchy.
List	List extends **Collection** to handle sequences (lists of objects).
Queue	Queue extends **Collection** to handle special types of lists in which elements are removed only from the head.
Deque	Deque extends **Queue** to handle a double-ended queue
Set	Extends **Collection** to handle sets, which must contain unique elements.
SortedSet	Extends **Set** to handle sorted sets.
NavigableSet	NavigableSet extends **SortedSet** to handle retrieval of elements based on closest-match

Interface Queue

- The Queue interface extends Collection and declares the behavior of a queue, which is often a first-in, first-out list.
- However, there are types of queues in which the ordering is based upon other criteria.
- Queue is a generic interface that has this declaration:

```
interface Queue<T>
```

Here, T specifies the type of objects that the queue will hold.

- The methods declared by Queue are shown in Table 7.4.

Methods declared in Queue

Method	Description
element( )	Returns the element at the head of the queue. The element is not removed. It throws **NoSuchElementException** if the queue is empty.
boolean offer(T *obj*)	Attempts to add *obj* to the queue. Returns **true** if *obj* was added and **false** otherwise.
T peek( )	Returns the element at the head of the queue. It returns **null** if the queue is empty. The element is not removed.
T poll( )	Returns the element at the head of the queue, removing the element in the process. It returns **null** if the queue is empty.
T remove( )	Removes the element at the head of the queue, returning the element in the process. It throws **NoSuchElementException** if the queue is empty.

Table 7.4: The methods declared in Queue interface

Interface Dequeue

Interfaces of collections

Interface	Description
Collection	Enables you to work with groups of objects; it is at the top of the collections hierarchy.
List	List extends **Collection** to handle sequences (lists of objects).
Queue	Queue extends **Collection** to handle special types of lists in which elements are removed only from the head.
Deque	Deque extends **Queue** to handle a double-ended queue
Set	Extends **Collection** to handle sets, which must contain unique elements.
SortedSet	Extends **Set** to handle sorted sets.
NavigableSet	NavigableSet extends **SortedSet** to handle retrieval of elements based on closest-match

Interface Dequeue

- The Deque interface extends Queue and declares the behavior of a double-ended queue.

- Double-ended queues can function as standard, first-in, first-out queues or as last-in, first-out stacks.

- Deque is a generic interface that has this declaration:

```
interface Deque<T>
```

Here, T specifies the type of objects that the deque will hold.

- In addition to the methods that it inherits from Queue, Deque adds those methods summarized in Table 7.5.

Methods declared in **Deueue**

Method	Description
void addFirst(E *obj*)	Adds *obj* to the head of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
void addLast(E *obj*)	Adds *obj* to the tail of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
Iterator<E> descendingIterator( )	Returns an iterator that moves from the tail to the head of the deque. In other words, it returns a reverse iterator.
E getFirst( )	Returns the first element in the deque. The object is not removed from the deque. It throws **NoSuchElementException** if the deque is empty.
E getLast( )	Returns the last element in the deque. The object is not removed from the deque. It throws **NoSuchElementException** if the deque is empty.
boolean offerFirst(E *obj*)	Attempts to add *obj* to the head of the deque. Returns **true** if *obj* was added and **false** otherwise. Therefore, this method returns **false** when an attempt is made to add *obj* to a full, capacity-restricted deque.
boolean offerLast(E *obj*)	Attempts to add *obj* to the tail of the deque. Returns **true** if *obj* was added and **false** otherwise.
E peekFirst( )	Returns the element at the head of the deque. It returns **null** if the deque is empty. The object is not removed.

Table 7.5: The methods declared in Dequeue interface (*continued…*)

Methods declared in Deueue

Method	Description
E peekLast( )	Returns the element at the tail of the deque. It returns **null** if the deque is empty. The object is not removed.
E pollFirst( )	Returns the element at the head of the deque, removing the element in the process. It returns **null** if the deque is empty.
E pollLast( )	Returns the element at the tail of the deque, removing the element in the process. It returns **null** if the deque is empty.
E pop( )	Returns the element at the head of the deque, removing it in the process. It throws **NoSuchElementException** if the deque is empty.
void push(E *obj*)	Adds *obj* to the head of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
E removeFirst( )	Returns the element at the head of the deque, removing the element in the process. It throws **NoSuchElementException** if the deque is empty.
boolean removeFirstOccurrence(Object *obj*)	Removes the first occurrence of *obj* from the deque. Returns **true** if successful and **false** if the deque did not contain *obj*.
E removeLast( )	Returns the element at the tail of the deque, removing the element in the process. It throws **NoSuchElementException** if the deque is empty.
boolean removeLastOccurrence(Object *obj*)	Removes the last occurrence of *obj* from the deque. Returns **true** if successful and **false** if the deque did not contain *obj*.

Table 7.5: The methods declared in Dequeue interface

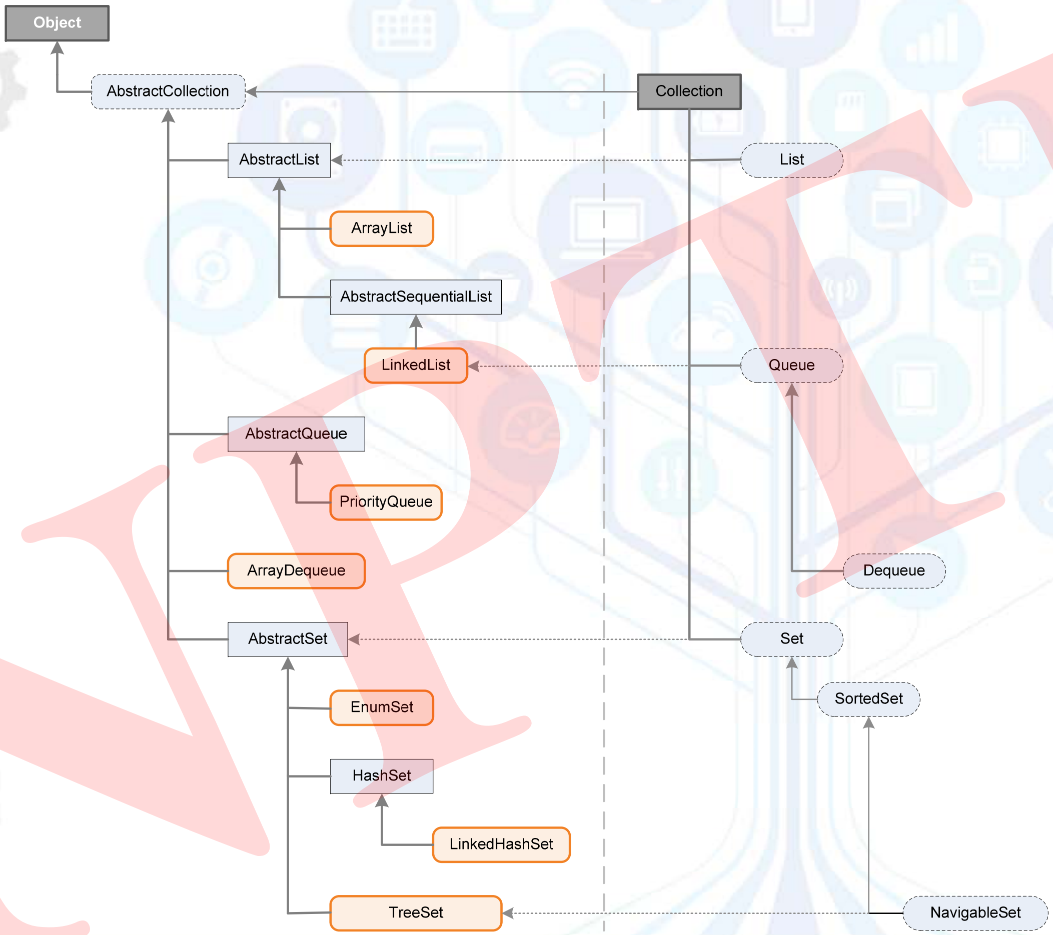
Interface Set

Interfaces of collections

Interface	Description
Collection	Enables you to work with groups of objects; it is at the top of the collections hierarchy.
List	List extends **Collection** to handle sequences (lists of objects).
Queue	Queue extends **Collection** to handle special types of lists in which elements are removed only from the head.
Deque	Deque extends **Queue** to handle a double-ended queue
Set	Extends **Collection** to handle sets, which must contain unique elements.
SortedSet	Extends **Set** to handle sorted sets.
NavigableSet	NavigableSet extends **SortedSet** to handle retrieval of elements based on closest-match

Classes in Collection

Class Collection



Classed in collection

- Interfaces are design rule, that is, it is the programmer task to have the implementations of each and every interfaces.
- It seems, then how the `java.util` package is useful. The `Collection` class take care this.
- The `Collection` class is the collection of classes which implements the interfaces we have discussed.
- In addition, the collection classes include many abstract classes as well. Anyway, a programmer has full liberty to adopt the implemented collection classes in their programs or they can implement of their own.
- The core collection classes are listed in Table 7.8.

Classes in `collection`

Class	Description
AbstractCollection	Implements most of the **Collection** interface.
AbstractList	Extends **AbstractCollection** and implements most of the **List** interface.
AbstractQueue	Extends **AbstractCollection** and implements parts of the **Queue** interface.
AbstractSequentialList	Extends **AbstractList** for use by a collection that uses sequential rather than random access of its elements.
LinkedList	Implements a linked list by extending **AbstractSequentialList**.
ArrayList	Implements a dynamic array by extending **AbstractList**.
ArrayDeque	Implements a dynamic double-ended queue by extending **AbstractCollection** and implementing the **Deque** interface.
AbstractSet	Extends **AbstractCollection** and implements most of the **Set** interface.
EnumSet	Extends **AbstractSet** for use with **enum** elements.
HashSet	Extends **AbstractSet** for use with a hash table.
LinkedHashSet	Extends **HashSet** to allow insertion-order iterations.
PriorityQueue	Extends **AbstractQueue** to support a priority-based queue.
TreeSet	Implements a set stored in a tree. Extends **AbstractSet**.

Table 7.6: The classes derived `Collection` class

Java Data Structures with Collection

Java data structures with collection

- You will learn how the different data structures that you can implement in your programs using the utilty available in java.uti package.

- Overall, all the data structures can be broadly classified into four categories. The broad data structures classification is shown in Table 7.9.

Data Structures	List	Queue	Set	Map
Indexed	ArrayList	ArrayDeque	HashSet	HashMap
Sequential	LinkedList	PriorityQueue	TreeSet	TreeMap
Indexed with links			LinkedHashSet	LinkedHashMap
Bit string			EnumSet	EnuMap

Table 7.7: Java Supports to data structures

REFERENCES

- https://cse.iitkgp.ac.in/~dsamanta/javads/index.html
- https://docs.oracle.com/javase/tutorial/

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

1951

योगः कर्मसु कौशलम

THANK YOU !

NPTEL ONLINE CERTIFICATION COURSES

Data Structures and Algorithms Using Java

Debasis Samanta

Department of Computer Science & Engineering, IIT Kharagpur

Module 03: Java Collection Framework

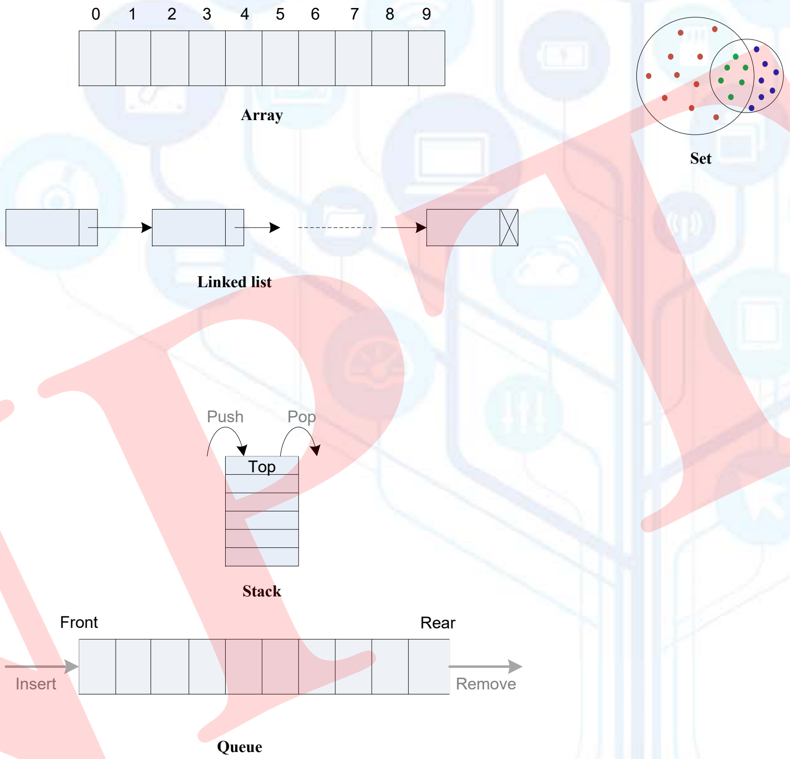
Lecture 08 : Set of JCF

CONCEPTS COVERED

- Constituents of Set in JCF
- Interfaces
- Classes
    - Constructors
    - Methods

Constituents of Set

Collections of JCF



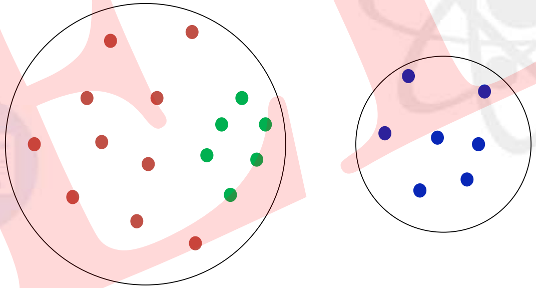
Collections under Collection

Set collections of JCF

- Set is a very useful concept in mathematics.

- Basically, Set is a type of collection that does not allow duplicate elements. That means an element can only exist once in a Set.

- Unlike other collection type such as array, list, linked list, set collection has the following distinctive characteristics.
  1. Duplicate elements are not allowed.
  2. Elements are not stored in order. That means you cannot expect elements sorted in any order when iterating over elements of a Set.



Collections of JCF

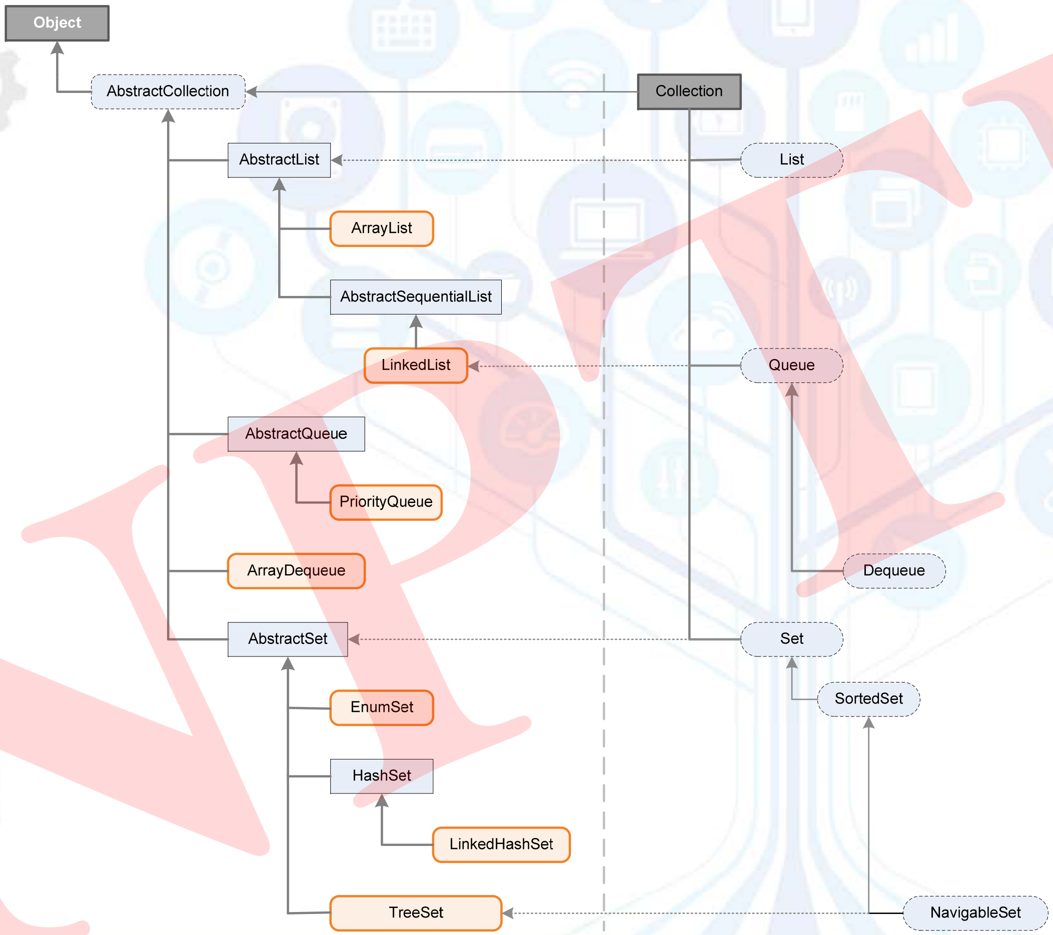
- Following are the interfaces and classes for managing set objects in Java
  - **Interfaces:**

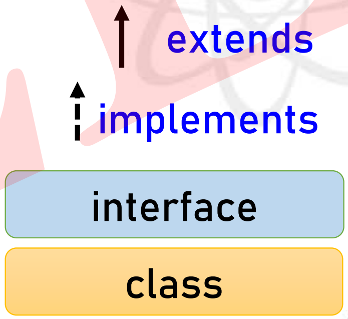
    Set, SortedSet, NavigableSet

  - **Classes:**

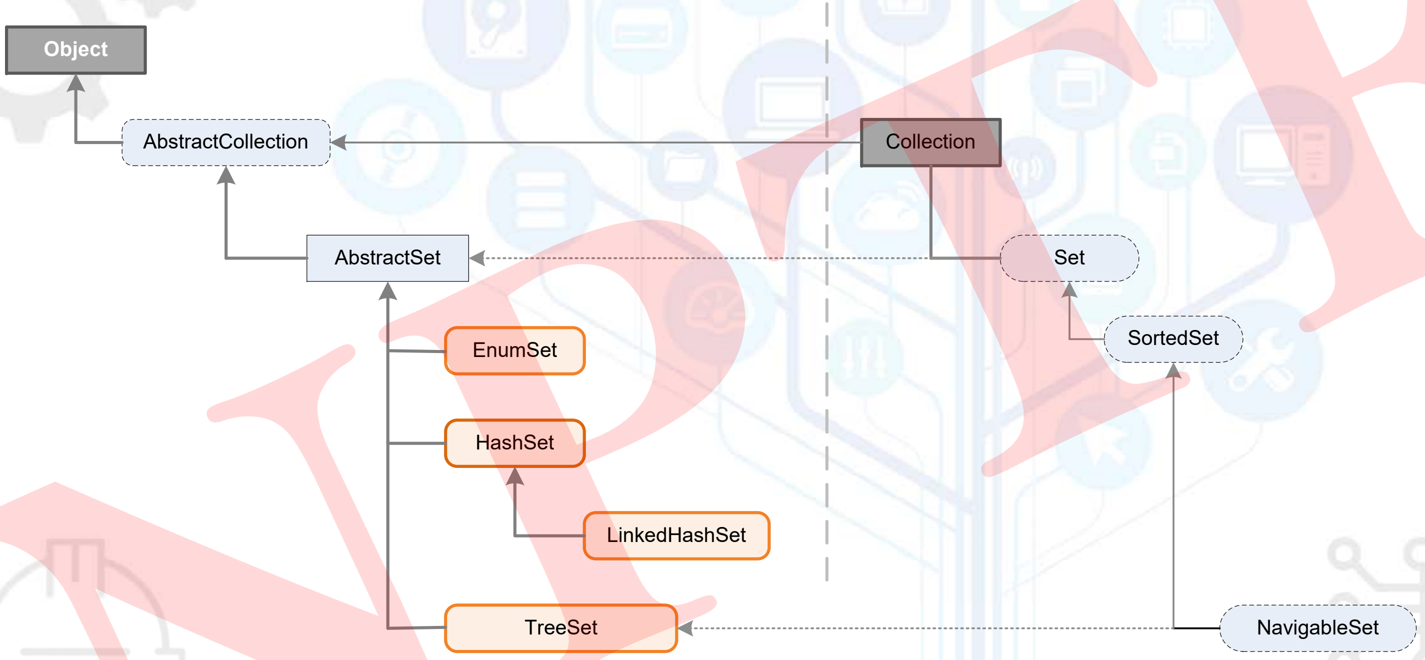
    EnumSet, HashSet, LinkedHashSet, TreeSet

Java collection hierarchy





Java collection hierarchy



extends

implements

interface

class

Interfaces for Set

Interfaces of collections

Interface	Description
Collection	Enables you to work with groups of objects; it is at the top of the collections hierarchy.
List	List extends **Collection** to handle sequences (lists of objects).
Queue	Queue extends **Collection** to handle special types of lists in which elements are removed only from the head.
Deque	Deque extends **Queue** to handle a double-ended queue
Set	Extends **Collection** to handle sets, which must contain unique elements.
SortedSet	Extends **Set** to handle sorted sets.
NavigableSet	NavigableSet extends **SortedSet** to handle retrieval of elements based on closest-match

Table 8.1: Interfaces for Set

Interface Set

- The Set interface defines a set. It extends Collection and specifies the behavior of a collection that does not allow duplicate elements.

- Therefore, the add( ) method returns false if an attempt is made to add duplicate elements to a set.

- Set is a generic interface that has this declaration:

```
interface Set<T>
```

Here, T specifies the type of objects that the set will hold.

- It does not specify any additional methods of its own.

Interface SortedSet

Interfaces of collections

Interface	Description
Collection	Enables you to work with groups of objects; it is at the top of the collections hierarchy.
List	List extends **Collection** to handle sequences (lists of objects).
Queue	Queue extends **Collection** to handle special types of lists in which elements are removed only from the head.
Deque	Deque extends **Queue** to handle a double-ended queue
Set	Extends **Collection** to handle sets, which must contain unique elements.
SortedSet	Extends **Set** to handle sorted sets.
NavigableSet	NavigableSet extends **SortedSet** to handle retrieval of elements based on closest-match

Interface SortedSet

- The SortedSet interface extends Set and declares the behavior of a set sorted in ascending order.

- SortedSet is a generic interface that has this declaration:

```
interface SortedSet<T>
```

Here, T specifies the type of objects that the set will hold.

- In addition to those methods provided by `Set`, the `SortedSet` interface declares the methods summarized in Table 7.6.

Methods declared in `SortedSet`

Method	Description
Comparator<? super E> comparator( )	Returns the invoking sorted set's comparator. If the natural ordering is used for this set, **null** is returned.
E first( )	Returns the first element in the invoking sorted set.
SortedSet<E> headSet(E *end*)	Returns a **SortedSet** containing those elements less than *end* that are contained in the invoking sorted set. Elements in the returned sorted set are also referenced by the invoking sorted set.
E last( )	Returns the last element in the invoking sorted set.
SortedSet<E> subSet(E *start*, E *end*)	Returns a **SortedSet** that includes those elements between *start* and *end*–1. Elements in the returned collection are also referenced by the invoking object.
SortedSet<E> tailSet(E *start*)	Returns a **SortedSet** that contains those elements greater than or equal to *start* that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object.

Table 8.2: The methods declared in `SortedSet` interface

Interface NavigableSet

Interfaces of collections

Interface	Description
Collection	Enables you to work with groups of objects; it is at the top of the collections hierarchy.
List	List extends **Collection** to handle sequences (lists of objects).
Queue	Queue extends **Collection** to handle special types of lists in which elements are removed only from the head.
Deque	Deque extends **Queue** to handle a double-ended queue
Set	Extends **Collection** to handle sets, which must contain unique elements.
SortedSet	Extends **Set** to handle sorted sets.
NavigableSet	NavigableSet extends **SortedSet** to handle retrieval of elements based on closest-match

Interface NavigableSet

- The `NavigableSet` interface extends `SortedSet` and declares the behavior of a collection that supports the retrieval of elements based on the closest match to a given value or values.
- `NavigableSet` is a generic interface that has this declaration:

```
interface NavigableSet<T>
```

Here, T specifies the type of objects that the set will hold.

- In addition to the methods that it inherits from `SortedSet, NavigableSet` adds those are summarized in Table 8.3.

Methods declared in SortedSet

Method	Description
E ceiling(E *obj*)	Searches the set for the smallest element *e* such that *e* >= *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
Iterator<E> descendingIterator( )	Returns an iterator that moves from the greatest to least. In other words, it returns a reverse iterator.
NavigableSet<E> descendingSet( )	Returns a **NavigableSet** that is the reverse of the invoking set. The resulting set is backed by the invoking set.
E floor(E *obj*)	Searches the set for the largest element *e* such that *e* <= *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
NavigableSet<E> headSet(E *upperBound*, boolean *incl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are less than *upperBound*. If *incl* is **true**, then an element equal to *upperBound* is included. The resulting set is backed by the invoking set.
E higher(E *obj*)	Searches the set for the largest element *e* such that *e* > *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
E lower(E *obj*)	Searches the set for the largest element *e* such that *e* < *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.

Table 8.3: The methods declared in `NavigableSet` interface (*continued*)

Methods declared in SortedSet

Method	Description
E pollFirst( )	Returns the first element, removing the element in the process. Because the set is sorted, this is the element with the least value. **null** is returned if the set is empty.
E pollLast( )	Returns the last element, removing the element in the process. Because the set is sorted, this is the element with the greatest value. **null** is returned if the set is empty.
NavigableSet<E> subSet(E *lowerBound*, boolean *lowIncl*, E *upperBound*, boolean *highIncl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are greater than *lowerBound* and less than *upperBound*. If *lowIncl* is **true**, then an element equal to *lowerBound* is included. If *highIncl* is **true**, then an element equal to *upperBound* is included. The resulting set is backed by the invoking set.
NavigableSet<E> tailSet(E *lowerBound*, boolean *incl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are greater than *lowerBound*. If *incl* is **true**, then an element equal to *lowerBound* is included. The resulting set is backed by the invoking set.

Table 8.4: The methods declared in `NavigableSet` interface

Class EnumSet

Class EnumSet

- EnumSet extends AbstractSet and implements Set. It is specifically for use with elements of an enum type.
- It is a generic class that has this declaration:

```
class EnumSet<E extends Enum<E>>
```

Here, E specifies the elements. Notice that E must extend Enum<E>, which enforces the requirement that the elements must be of the specified enum type.

- EnumSet defines no constructors. Instead, it uses the factory methods shown in Table 8.5 to create objects.

- The copyOf( ) and range( ) methods can also throw IllegalArgumentException. Notice that the of( ) method is overloaded a number of times. This is in the interest of efficiency. Passing a known number of arguments can be faster than using a vararg parameter when the number of arguments is small.

Methods declared in EnumSet

Method	Description
static <E extends Enum<E>> EnumSet<E> allOf(Class<E> *t*)	Creates an EnumSet that contains the elements in the enumeration specified by t..
static <E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> e)	Creates an EnumSet that is comprised of those elements not stored in e.
static <E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> c)	Creates an EnumSet from the elements stored in c.
static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> c)	Creates an EnumSet from the elements stored in c.
static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> t)	Creates an EnumSet that contains the elements that are not in the enumeration specified by t, which is an empty set by definition.
static <E extends Enum<E>> EnumSet<E> of(E v, E ... varargs)	Creates an EnumSet that contains v and zero or more additional enumeration values.

Table 8.5: The methods declared in `EnumSet class` (*continued*)

Methods declared in EnumSet

Method	Description
static <E extends Enum<E>> EnumSet<E> of(E v)	Creates an EnumSet that contains v.
static <E extends Enum<E>> EnumSet<E> of(E v1, E v2)	Creates an EnumSet that contains v1 and v2.
static <E extends Enum<E>> EnumSet<E> of(E v1, E v2, E v3)	Creates an EnumSet that contains v1 through v3.
static <E extends Enum<E>> EnumSet<E> of(E v1, E v2, E v3, E v4)	Creates an EnumSet that contains v1 through v4.
static <E extends Enum<E>> EnumSet<E> of(E v1, E v2, E v3, E v4, E v5)	Creates an EnumSet that contains v1 through v5.
static <E extends Enum<E>> EnumSet<E> range(E start, E end)	Creates an EnumSet that contains the elements in the range specified by start and end.

Table 8.5: The methods declared in `EnumSet class`

Class HashSet

Class HashSet

- HashSet extends AbstractSet and implements the Set interface. It creates a collection that uses a hash table for storage.
- HashSet is a generic class that has this declaration:

```
class HashSet<E>
```

Here, E specifies the type of objects that the set will hold.

- A hash table stores information by using a mechanism called hashing. In hashing, the informational content of a key is used to determine a unique value, called its hash code. The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically—you never see the hash code itself. Also, your code can't directly index the hash table. The advantage of hashing is that it allows the execution time of add( ), contains( ), remove( ), and size( ) to remain constant even for large sets.

Methods declared in HashSet

Constructor	Description
HashSet( )	It is a default constructor to create a hash set.
HashSet(Collection<? extends E> c)	It initializes the hash set by using the elements of c.
HashSet(int capacity)	It initializes the capacity of the hash set to capacity.
HashSet(int capacity, float fillRatio)	It initializes both the capacity and the fill ratio (also called load capacity) of the hash set from its arguments. The fill ratio must be between 0.0 and 1.0, and it determines how full the hash set can be before it is resized upward.

Table 8.6: The methods declared in `HashSet class`

Class LinkedHashSet

Class LinkedHashSet

- A LinkedHashSet is an ordered version of HashSet that maintains a doubly-linked list across all elements. When the iteration order is needed to be maintained this class is used. When iterating through a HashSet the order is unpredictable, while a LinkedHashSet lets us iterate through the elements in the order in which they were inserted. When cycling through LinkedHashSet using an iterator, the elements will be returned in the order in which they were inserted.

- The LinkedHashSet class extends HashSet and adds no members of its own. It is a generic class that has this declaration:

```
class LinkedHashSet<E>
```

Here, E specifies the type of objects that the set will hold.

- The constructors in the LinkedHashSet are shown in Table 8.7.

Constructors of LinkedHashSet

Constructor	Description
LinkedHashSet( )	It is a default constructor to create a hash set.
LinkedHashSet(Collection< ? extends E> c)	It initializes the hash set by using the elements of c.
LinkedHashSet(int capacity)	It initializes the capacity of the hash set to capacity.
LinkedHashSet(int capacity, float fillRatio)	It initializes both the capacity and the fill ratio (also called load capacity) of the linked hash set from its arguments. The fill ratio must be between 0.0 and 1.0, and it determines how full the linked hash set can be before it is resized upward.

Table 8.7: The constructors declared in `LinkedHashSet class`

Constructors and methods of LinkedHashSet

- The constructors in the LinkedHashSet class are in the similar form that of the constructor in Hashset class.
- The LinkedHashSet class extends HashSet class and implements Set interface.

- The LinkedHashSet class does not define any exclusive methods of its own. All methods are same as the methods as in HashSet class. This implies that whatever the operations we can perform with HashSet collections are also possible with the LinkedHashSet class. Hence, the manipulation of LinkedHashSet collections are not illustrated explicitly.

Class TreeSet

Class TreeSet

- TreeSet extends AbstractSet and implements the NavigableSet interface, which in turns successively extends SortedSet and Set interfaces.

- This implies all the methods defined in NavigableSet are implemented by the SortedSet class.
  - It may be noted that this class like LinkedHashSet class does not have its own method defined.

- The TreeSet It creates a collection that uses a tree for storage and hence its name.

- Further, in this type of set, elements are stored in ascending order of sorting.

- Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.

Constructors of TreeSet

Constructor	Description
TreeSet( )	It is a default constructor to create an empty set that will be sorted in ascending order according to the natural order of its elements.
TreeSet(Collection<? extends E> c)	It builds a tree set that contains the elements of c, where c is any collection.
TreeSet(Comparator<? super E> comp)	It creates an empty tree set that will be sorted according to the comparator specified by comp.
TreeSet(SortedSet<E> ss)	It builds a tree set that contains the elements of ss.

Table 8.7: The constructors declared in TreeSet class

Java data structures with `collection`

- You will learn how the different data structures that you can implement in your programs using the utility available in `java.util` package.

- Overall, all the data structures can be broadly classified into four categories. The broad data structures classification is shown in Table 8.8.

Data Structures	List	Queue	Set	Map
Indexed	ArrayList	ArrayDeque	HashSet	HashMap
Sequential	LinkedList	PriorityQueue	TreeSet	TreeMap
Indexed with links			LinkedHashSet	LinkedHashMap
Bit string			EnumSet	EnuMap

Table 8.8: Java Supports to data structures

REFERENCES

- https://cse.iitkgp.ac.in/~dsamanta/javads/index.html
- https://docs.oracle.com/javase/tutorial/

THANK YOU !

NPTEL ONLINE CERTIFICATION COURSES

Data Structures and Algorithms Using Java

Debasis Samanta

Department of Computer Science & Engineering, IIT Kharagpur

Module 03: Java Collection Framework

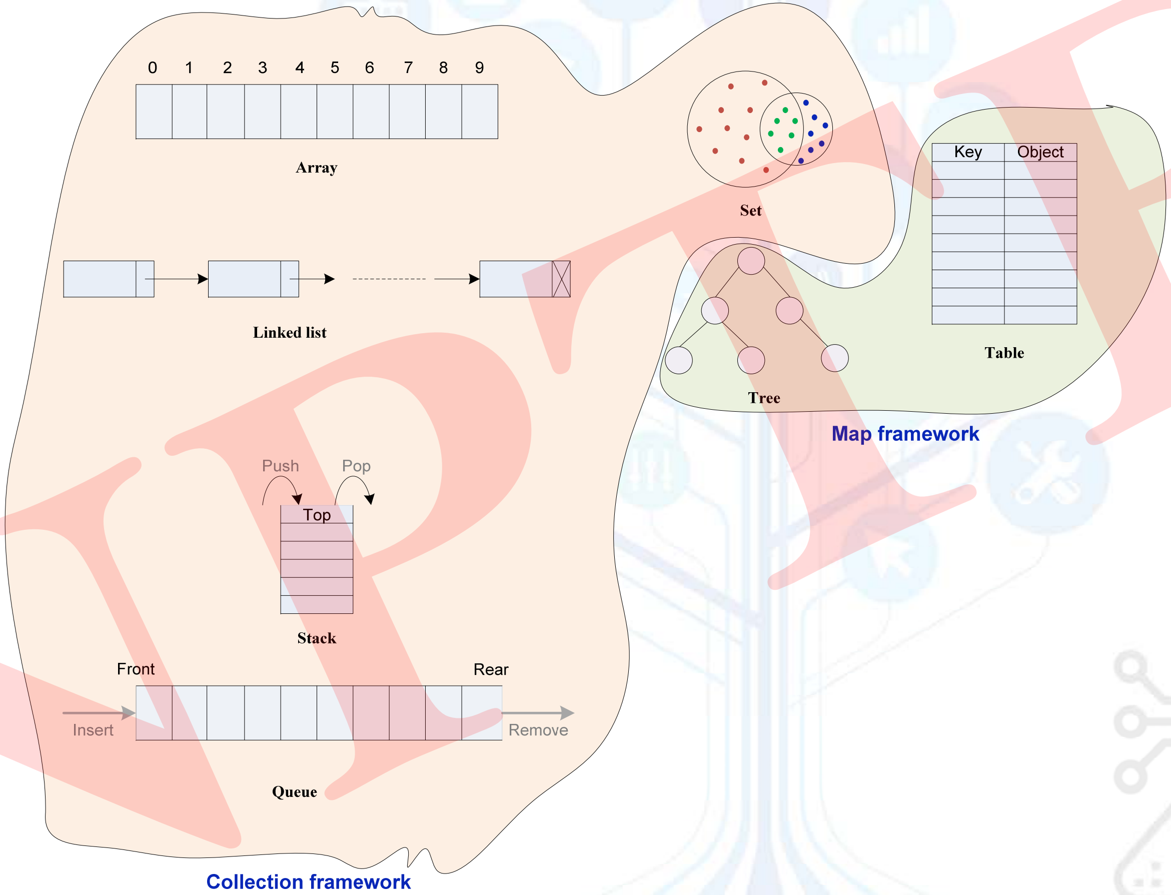
Lecture 09 : Map Framework

CONCEPTS COVERED

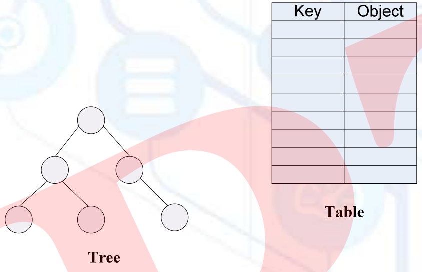
- Constituents of Map of JCF
- Interfaces
- Classes
  - Constructors
  - Methods

Constituents of Map

Java collection framework

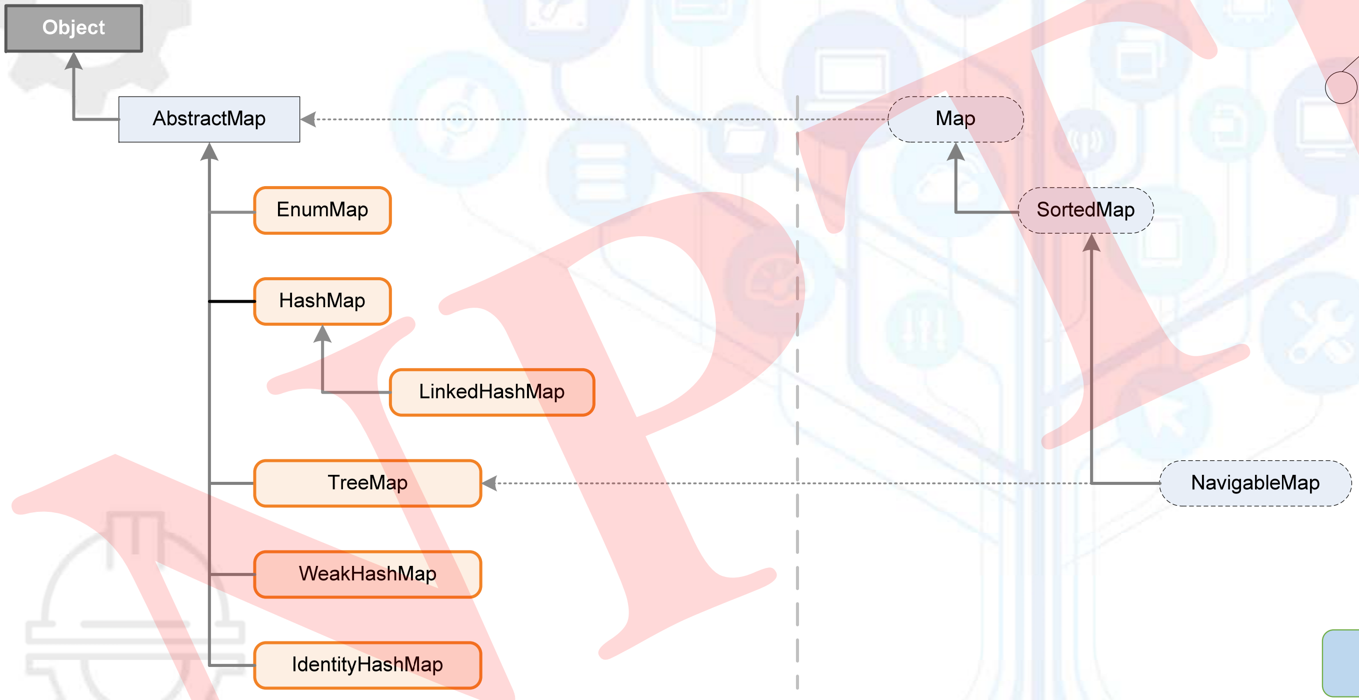


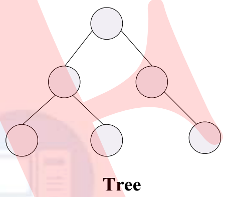
Map : Collection as tree and table

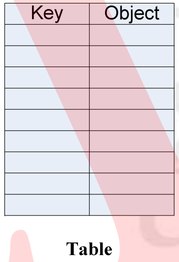


Map Framework

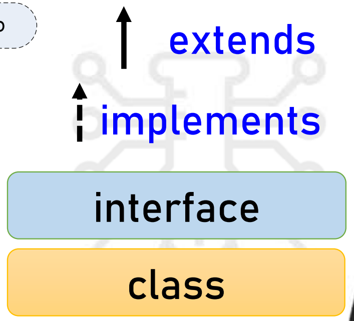
Map hierarchy







Map Framework



Map framework

- Java introduces the concept of Map, which is another member of the Java Collection Framework.
- In Java, a Map is an object that maps keys to values, or is a collection of key-value pairs. It models the function abstraction in mathematics.
-
- In `java.util` package, a number of interfaces and classes are defined and declared to support map objects in Java program.
- The hierarchy of the classes and interfaces of Map is quite complex like Collection framework.
- The entire Map framework is built upon a set of standard interfaces, classes and algorithms.
  - **Interfaces:**
    `Map, Map.Entry, StoredMap, NavigableMap`
  - **Classes:**
    `EnumMap, HashMap, TreeMap, LinkedHashMap, IdentityHashMap`

Interfaces in Map

Interfaces of map framework

Interface	Description
Map	Maps unique keys to values. The interface is generic and it is defined as interface Map<K, V>, where K specifies the type of keys, and V specifies the type of values.
Map.Entry	Describes an element (a key/value pair) in a map. This is an inner class of **Map**.
NavigableMap	Extends **SortedMap** to handle the retrieval of entries based on closest-match searches.
SortedMap	Extends **Map** so that the keys are maintained in ascending order.

Table 9.1: Interfaces in Map framework

Map Interface

Map interface

- The Map interface is another foundation like Collection framework. It must be implemented by any class that defines a map.

- Map is a generic interface that has this declaration:

```
interface Map<T>
```

Here, T specifies the type of objects that the map will hold.

Methods declared in Map

- Map declares the core methods that all maps will have.
- Because all maps implement Map, familiarity with its methods is necessary for a clear understanding of the framework.
- These methods are summarized in Table 9.2.

Method	Description
void clear( )	Removes all key/value pairs from the invoking map.
default V compute(K *k*, BiFunction<? super K, ? super V, ? extends V> *func*)	Calls *func* to construct a new value. If func returns non-**null**, the new key/value pair is added to the map, any preexisting pairing is removed, and the new value is returned. If *func* returns **null**, any preexisting pairing is removed, and **null** is returned.
default V computeIfAbsent(K *k*, Function<? super K, ? extends V> *func*)	Returns the value associated with the key *k*. Otherwise, the value is constructed through a call to *func* and the pairing is entered into the map and the constructed value is returned. If no value can be constructed, **null** is returned.
default V computeIfPresent(K *k*, BiFunction<? super K, ? super V, ? extends V> *func*)	If *k* is in the map, a new value is constructed through a call to *func* and the new value replaces the old value in the map. In this case, the new value is returned. If the value returned by *func* is **null**, the existing key and value are removed from the map and **null** is returned.

Table 9.2: The methods declared in Map interface (*continued…*)

Methods declared in Map

Method	Description
boolean containsKey(Object *k*)	Returns **true** if the invoking map contains *k* as a key. Otherwise, returns **false**.
boolean containsValue(Object *v*)	Returns **true** if the map contains *v* as a value. Otherwise, returns **false**.
Set<Map.Entry<K, V>> entrySet( )	Returns a **Set** that contains the entries in the map. The set contains objects of type **Map.Entry**. Thus, this method provides a set-view of the invoking map.
boolean equals(Object *obj*)	Returns **true** if *obj* is a **Map** and contains the same entries. Otherwise, returns **false**.
default void forEach(BiConsumer< ? super K, ? super V> *action*)	Executes *action* on each element in the invoking map. A **ConcurrentModificationException** will be thrown if an element is removed during the process.
V get(Object *k*)	Returns the value associated with the key *k*. Returns **null** if the key is not found.
default V getOrDefault(Object *k*, V *defVal*)	Returns the value associated with *k* if it is in the map. Otherwise, *defVal* is returned.
int hashCode( )	Returns the hash code for the invoking map.

Table 9.2: The methods declared in Map interface (*continued…*)

Methods declared in Map

Method	Description
boolean isEmpty( )	Returns **true** if the invoking map is empty. Otherwise, returns **false**.
Set<K> keySet( )	Returns a **Set** that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map.
default V merge(K *k*, V *v*, BiFunction<? super V, ? super V, ? extends V> *func*)	If *k* is not in the map, the pairing *k*,*v* is added to the map. In this case, *v* is returned. Otherwise, *func* returns a new value based on the old value, the key is updated to use this value, and **merge( )** returns this value. If the value returned by *func* is **null**, the existing key and value are removed from the map and **null** is returned.
V put(K *k*, V *v*)	Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are *k* and *v*, respectively. Returns **null** if the key did not already exist. Otherwise, the previous value linked to the key is returned.
void putAll(Map<? extends K, ? extends V> *m*)	Puts all the entries from *m* into this map.
default V putIfAbsent(K *k*, V *v*)	Inserts the key/value pair into the invoking map if this pairing is not already present or if the existing value is **null**. Returns the old value. The **null** value is returned when no previous mapping exists, or the value is **null**.
V remove(Object *k*)	Removes the entry whose key equals *k*.

Table 9.2: The methods declared in Map interface (*continued…*)

Methods declared in **Map**

Method	Description
default boolean remove(Object *k*, Object *v*)	If the key/value pair specified by *k* and *v* is in the invoking map, it is removed and **true** is returned. Otherwise, **false** is returned.
default boolean replace(K *k*, V *oldV*, V *newV*)	If the key/value pair specified by *k* and *oldV* is in the invoking map, the value is replaced by *newV* and **true** is returned. Otherwise **false** is returned.
default V replace(K *k*, V *v*)	If the key specified by *k* is in the invoking map, its value is set to *v* and the previous value is returned. Otherwise, **null** is returned.
default void replaceAll(BiFunction< ? super K, ? super V, ? extends V> *func*)	Executes *func* on each element of the invoking map, replacing the element with the result returned by *func*. A **ConcurrentModificationException** will be thrown if an element is removed during the process.
int size( )	Returns the number of key/value pairs in the map.
Collection<V> values( )	Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.

Table 9.2: The methods declared in Map interface

Interface SortedMap

Interfaces of Map

Interface	Description
Map	Maps unique keys to values. The interface is generic and it is defined as interface Map<K, V>, where K specifies the type of keys, and V specifies the type of values.
SortedMap	Extends **Map** so that the keys are maintained in ascending order.
NavigableMap	Extends **SortedMap** to handle the retrieval of entries based on closest-match searches.
Map.Entry	Describes an element (a key/value pair) in a map. This is an inner class of **Map**.

Interface StoredMap

- The SortedMap interface extends Map.
- It ensures that the entries are maintained in ascending order based on the keys.
- SortedMap is generic like the interface Map. The methods defined in the StoredMap interface are listed in Table 9.3.

Methods declared in StoredMap

Method	Description
Comparator<? super K> comparator( )	Returns the invoking sorted map's comparator. If natural ordering is used for the invoking map, **null** is returned.
K firstKey( )	Returns the first key in the invoking map.
SortedMap<K, V> headMap(K *end*)	Returns a sorted map for those map entries with keys that are less than *end*.
K lastKey( )	Returns the last key in the invoking map.
SortedMap<K, V> subMap(K *start*, K *end*)	Returns a map containing those entries with keys that are greater than or equal to *start* and less than *end*.
SortedMap<K, V> tailMap(K *start*)	Returns a map containing those entries with keys that are greater than or equal to *start*.

Table 9.3: The methods declared in StoredMap interface

Interface NavigableMap

Interfaces of Map

Interface	Description
Map	Maps unique keys to values. The interface is generic and it is defined as interface Map<K, V>, where K specifies the type of keys, and V specifies the type of values.
SortedMap	Extends **Map** so that the keys are maintained in ascending order.
NavigableMap	Extends **SortedMap** to handle the retrieval of entries based on closest-match searches.
Map.Entry	Describes an element (a key/value pair) in a map. This is an inner class of **Map**.

Interface NavigableMap

- The NavigableMap interface extends SortedMap and declares the behavior of a map that supports the retrieval of entries based on the closest match to a given key or keys.
- The NavigableMap is also a generic interface like the SortedMap and Map interfaces.
- The methods defined in the NavigableMap interface are listed in Table 9.4.

Methods declared in NavigableMap

Method	Description
Map.Entry<K,V> ceilingEntry(K *obj*)	Searches the map for the smallest key *k* such that *k >= obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K ceilingKey(K *obj*)	Searches the map for the smallest key *k* such that *k >= obj*. If such a key is found, it is returned. Otherwise, **null** is returned.
NavigableSet<K> descendingKeySet( )	Returns a **NavigableSet** that contains the keys in the invoking map in reverse order. Thus, it returns a reverse set-view of the keys. The resulting set is backed by the map.
NavigableMap<K,V> descendingMap( )	Returns a **NavigableMap** that is the reverse of the invoking map. The resulting map is backed by the invoking map.
Map.Entry<K,V> firstEntry( )	Returns the first entry in the map. This is the entry with the least key.
Map.Entry<K,V> floorEntry(K *obj*)	Searches the map for the largest key *k* such that *k <= obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K floorKey(K *obj*)	Searches the map for the largest key *k* such that *k <= obj*. If such a key is found, it is returned. Otherwise, **null** is returned.

Table 9.4: The methods declared in NavigableMap interface (*continued*)

Methods declared in NavigableMap

Method	Description
NavigableMap<K,V> headMap(K *upperBound*, boolean *incl*)	Returns a **NavigableMap** that includes all entries from the invoking map that have keys that are less than *upperBound*. If *incl* is **true**, then an element equal to *upperBound* is included. The resulting map is backed by the invoking map.
Map.Entry<K,V> higherEntry(K *obj*)	Searches the set for the largest key *k* such that *k* > *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K higherKey(K *obj*)	Searches the set for the largest key *k* such that *k* > *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.
Map.Entry<K,V> lastEntry( )	Returns the last entry in the map. This is the entry with the largest key.
Map.Entry<K,V> lowerEntry(K obj)	Searches the set for the largest key *k* such that *k* < *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K lowerKey(K *obj*)	Searches the set for the largest key *k* such that *k* < *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.
NavigableSet<K> navigableKeySet( )	Returns a **NavigableSet** that contains the keys in the invoking map. The resulting set is backed by the invoking map.

Table 9.4: The methods declared in NavigableMap interface (*continued*)

Methods declared in NavigableMap

Method	Description
Map.Entry<K,V> pollFirstEntry( )	Returns the first entry, removing the entry in the process. Because the map is sorted, this is the entry with the least key value. **null** is returned if the map is empty.
Map.Entry<K,V> pollLastEntry( )	Returns the last entry, removing the entry in the process. Because the map is sorted, this is the entry with the greatest key value. **null** is returned if the map is empty.
NavigableMap<K,V> subMap(K *lowerBound*, boolean *lowIncl*, K *upperBound* boolean *highIncl*)	Returns a **NavigableMap** that includes all entries from the invoking map that have keys that are greater than *lowerBound* and less than *upperBound*. If *lowIncl* is **true**, then an element equal to *lowerBound* is included. If *highIncl* is **true**, then an element equal to *highIncl* is included. The resulting map is backed by the invoking map.
NavigableMap<K,V> tailMap(K *lowerBound*, boolean *incl*)	Returns a **NavigableMap** that includes all entries from the invoking map that have keys that are greater than *lowerBound*. If *incl* is **true**, then an element equal to *lowerBound* is included. The resulting map is backed by the invoking map.

Table 9.4: The methods declared in NavigableMap interface

Interface Map.Entry

Interfaces of Map

Interface	Description
Map	Maps unique keys to values. The interface is generic and it is defined as interface Map<K, V>, where K specifies the type of keys, and V specifies the type of values.
SortedMap	Extends **Map** so that the keys are maintained in ascending order.
NavigableMap	Extends **SortedMap** to handle the retrieval of entries based on closest-match searches.
Map.Entry	Describes an element (a key/value pair) in a map. This is an inner class of **Map**.

Interface Map.Entry

- The Map.Entry interface enables you to work with a map entry.

- Recall that the entrySet( ) method declared by the Map interface returns a Set containing the map entries. Each of these set elements is a Map.Entry object.

- Map.Entry is generic and is declared like this: interface Map.Entry<K, V> Here, K specifies the type of keys, and V specifies the type of values.

- Table 9.5 summarizes the non-static methods declared by Map.Entry.

Methods declared in **Map.Entry**

Method	**Description**
boolean equals(Object *obj*)	Returns **true** if *obj* is a **Map.Entry** whose key and value are equal to that of the invoking object.
K getKey( )	Returns the key for this map entry.
V getValue( )	Returns the value for this map entry.
int hashCode( )	Returns the hash code for this map entry.
V setValue(V *v*)	Sets the value for this map entry to *v*. A **ClassCastException** is thrown if *v* is not the correct type for the map. An **IllegalArgumentException** is thrown if there is a problem with *v*. A **NullPointerException** is thrown if *v* is **null** and the map does not permit **null** keys. An **UnsupportedOperationException** is thrown if the map cannot be changed.

Table 9.5: The methods declared in Map.Entry interface

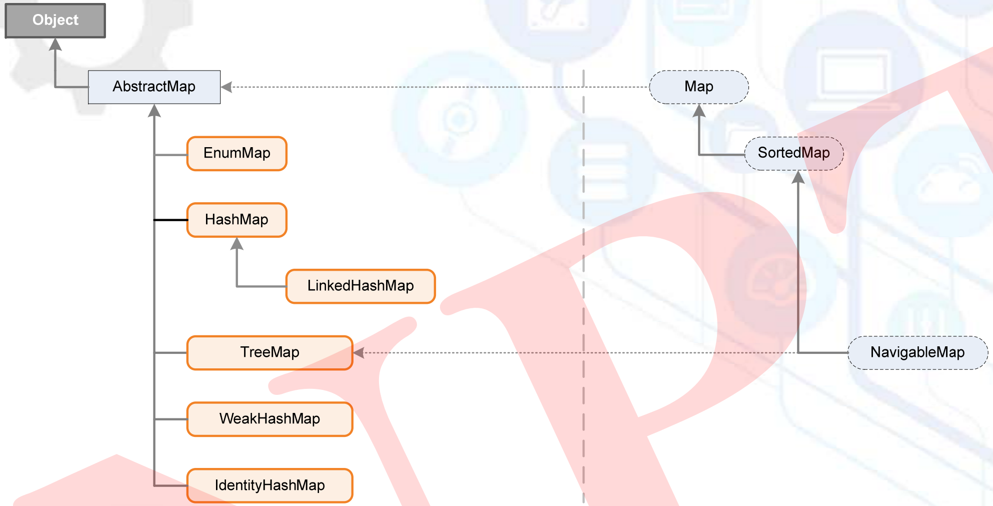
Map Classes

Map classes

Class	Description
AbstractMap	Implements most of the **Map** interface.
EnumMap	Extends **AbstractMap** for use with **enum** keys.
HashMap	Extends **AbstractMap** to use a hash table.
TreeMap	Extends **AbstractMap** to use a tree.
LinkedHashMap	Extends **HashMap** to allow insertion-order iterations.
IdentityHashMap	Extends **AbstractMap** and uses reference equality when comparing documents.
WeakHashMap	Extends **AbstractMap** to use a hash table with weak keys.

Table 9.6: The classes for map in Java Collection Framework

Class Map



- There are several classes (Table 9.6) to implement the map interfaces.
- All classes extends the AbstractMap class, which in turns implements the Map interface.
- This implies that all the methods in the Map interface are mostly defined in them in addition to some of their own methods.

EnumMap Class

Map classes

Class	Description
AbstractMap	Implements most of the **Map** interface.
EnumMap	Extends **AbstractMap** for use with **enum** keys.
HashMap	Extends **AbstractMap** to use a hash table.
TreeMap	Extends **AbstractMap** to use a tree.
LinkedHashMap	Extends **HashMap** to allow insertion-order iterations.
IdentityHashMap	Extends **AbstractMap** and uses reference equality when comparing documents.
WeakHashMap	Extends **AbstractMap** to use a hash table with weak keys.

Class EnumMap

- This class defines keys of an enum type. It is a generic class that has this declaration:

```
class EnumMap<K extends Enum<K>, V>
```

  Here, K specifies the type of key, and V specifies the type of value.

- Notice that K must extend Enum<K>, which enforces the requirement that the keys must be of an enum type.

- EnumMap defines the following constructors which is shown in Table 9.7.

Class EnumMap

Constructor	Description
EnumMap(Class<K> kType)	This constructor creates an empty EnumMap of type kType.
EnumMap(Map<K, ? extends V> m)	This constructor creates an EnumMap map that contains the same entries as m.
EnumMap(EnumMap<K, ? extends V> em)	To create an EnumMap initialized with the values in em.

Table 9.7: The constructors defined in EnumMap class

Note:

There is no method of its own defined in EnumMap class.

HashMap Class

Map classes

Class	Description
AbstractMap	Implements most of the **Map** interface.
EnumMap	Extends **AbstractMap** for use with **enum** keys.
HashMap	Extends **AbstractMap** to use a hash table.
TreeMap	Extends **AbstractMap** to use a tree.
LinkedHashMap	Extends **HashMap** to allow insertion-order iterations.
IdentityHashMap	Extends **AbstractMap** and uses reference equality when comparing documents.
WeakHashMap	Extends **AbstractMap** to use a hash table with weak keys.

Class HashMap

- This class is used to create a hash table to store the map. The execution of get( ) and put() method can be done in a constant time irrespective of the size of the table because of the use of hash value of key.

- HashMap is a generic class and it has the following declaration:

```
class HashMap<K, V>
```

  Here, K specifies the type of keys, and V specifies the type of values.

- HashMap defines the following constructors which is shown in Table 9.8.

Class HashMap : Constructors

Constructor	Description
HashMap( )	This constructor creates a default hash map.
HashMap(Map<? extends K, ? extends V> m)	The form is to initialize the hash map using the elements of m.
HashMap(int capacity)	The third form initializes the capacity of the hash map to capacity.
HashMap(int capacity, float fillRatio)	The fourth form initializes both the capacity and fill ratio of the hash map by using its arguments. The meaning of capacity and fill ratio is the same as for HashSet, described earlier. The default capacity is 16. The default fill ratio is 0.75.

Table 9.8: The constructors defined in HashMap class

Note:

There is no method of its own defined in HashMap class.

TreeMap Class

Map classes

Class	Description
AbstractMap	Implements most of the **Map** interface.
EnumMap	Extends **AbstractMap** for use with **enum** keys.
HashMap	Extends **AbstractMap** to use a hash table.
TreeMap	Extends **AbstractMap** to use a tree.
LinkedHashMap	Extends **HashMap** to allow insertion-order iterations.
IdentityHashMap	Extends **AbstractMap** and uses reference equality when comparing documents.
WeakHashMap	Extends **AbstractMap** to use a hash table with weak keys.

Class TreeMap

- The TreeMap creates maps stored in a tree structure.

- A TreeMap provides an efficient means of storing key/value pairs in sorted order and allows rapid retrieval. You should note that, unlike a hash map, a tree map guarantees that its elements will be sorted in ascending key order.

- The TreeMap class extends AbstractMap and implements the NavigableMap interface.

- TreeMap is a generic class that has this declaration:

```
class TreeMap<K, V>
```

Here, K specifies the type of keys, and V specifies the type of values.

- TreeMap defines the constructors which is shown in Table 9.9

Class TreeMap: Constructors

Constructor	Description
TreeMap( )	The first form constructs an empty tree map that will be sorted by using the natural order of its keys.
TreeMap(Comparator<? super K> comp)	The second form constructs an empty tree-based map that will be sorted by using the Comparator comp. (Comparators are discussed later in this chapter.)
TreeMap(Map<? extends K, ? extends V> m)	The third form initializes a tree map with the entries from m, which will be sorted by using the natural order of the keys..
TreeMap(SortedMap<K, ? extends V> sm)	The fourth form initializes a tree map with the entries from sm, which will be sorted in the same order as sm.

Table 9.9: The constructors defined in HashMap class

Note:

TreeMap has no map methods beyond those specified by the NavigableMap interface and the AbstractMap class.

LinkedHashMap Class

Map classes

Class	Description
AbstractMap	Implements most of the **Map** interface.
EnumMap	Extends **AbstractMap** for use with **enum** keys.
HashMap	Extends **AbstractMap** to use a hash table.
TreeMap	Extends **AbstractMap** to use a tree.
LinkedHashMap	Extends **HashMap** to allow insertion-order iterations.
IdentityHashMap	Extends **AbstractMap** and uses reference equality when comparing documents.
WeakHashMap	Extends **AbstractMap** to use a hash table with weak keys.

Class LinkedHashMap

- It maintains a linked list of the entries in the map, in the order in which they were inserted. This allows insertion-order iteration over the map. That is, when iterating through a collection-view of a LinkedHashMap, the elements will be returned in the order in which they were inserted.

- You can also create a LinkedHashMap that returns its elements in the order in which they were last accessed.

- LinkedHashMap is a generic class that has this declaration:

```
class LinkedHashMap<K, V>
```

Here, K specifies the type of keys, and V specifies the type of values.

- LinkedHashMap extends HashMap. LinkedHashMap defines constructors which is shown in Table 9.10.

Class LinkedHashMap

Constructor	Description
LinkedHashMap( )	It is a default constructor.
LinkedHashMap(Map<? extends K, ? extends V> m)	This constructor initializes the LinkedHashMap with the elements from m.
LinkedHashMap(int capacity)	The third form initializes the capacity.
LinkedHashMap(int capacity, float fillRatio)	initializes both capacity and fill ratio. The default capacity is 16. The default ratio is 0.75.
LinkedHashMap(int capacity, float fillRatio, boolean Order)	It allows you to specify whether the elements will be stored in the linked list by insertion order, or by order of last access. If Order is true, then access order is used. If Order is false, then insertion order is used.

Table 9.10: The constructors defined in LinkedHashMap class

Class LinkedHashMap

- LinkedHashMap adds only one method to those defined by HashMap. This method is removeEldestEntry(), and it is shown here:

```
protected boolean removeEldestEntry(Map.Entry<K, V> e)
```

- This method is used keep a track of whether the map removes any eldest entry from the map. So each time a new element is added to the LinkedHashMap, the eldest entry is removed from the map.

- This method is generally invoked after the addition of the elements into the map by the use of put() and putall() method. The oldest entry is passed in e. By default, this method returns false and does nothing. However, if you override this method, then you can have the LinkedHashMap remove the oldest entry in the map. To do this, have your override return true. To keep the oldest entry, return false.

Map classes

Class	Description
AbstractMap	Implements most of the **Map** interface.
EnumMap	Extends **AbstractMap** for use with **enum** keys.
HashMap	Extends **AbstractMap** to use a hash table.
TreeMap	Extends **AbstractMap** to use a tree.
LinkedHashMap	Extends **HashMap** to allow insertion-order iterations.
IdentityHashMap	Extends **AbstractMap** and uses reference equality when comparing documents.
WeakHashMap	Extends **AbstractMap** to use a hash table with weak keys.

Map classes

The IdentityHashMap class

The API documentation explicitly states that IdentityHashMap is not for general use and hence its discussion is ignored.

The IdentityHashMap class

WeakHashMap implements a map that uses “weak keys,” which allows an element in a map to be garbage-collected when its key is otherwise unused. This class is also not used for general use and is not discussed.

Java data structures with Collection

- You will learn how the different data structures that you can implement in your programs using the utilty available in `java.util` package.

- Overall, all the data structures can be broadly classified into four categories. The broad data structures classification is shown in Table 7.9.

Data Structures	List	Queue	Set	Map
Indexed	ArrayList	ArrayDeque	HashSet	HashMap
Sequential	LinkedList	PriorityQueue	TreeSet	TreeMap
Indexed with links			LinkedHashSet	LinkedHashMap
Bit string			EnumSet	EnuMap

Table 9.1: Java Supports to data structures

REFERENCES

- https://cse.iitkgp.ac.in/~dsamanta/javads/index.html
- https://docs.oracle.com/javase/tutorial/

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

1951

योगः कर्मसु कौशलम्

THANK YOU !

NPTEL ONLINE CERTIFICATION COURSES

Data Structures and Algorithms Using Java

Debasis Samanta

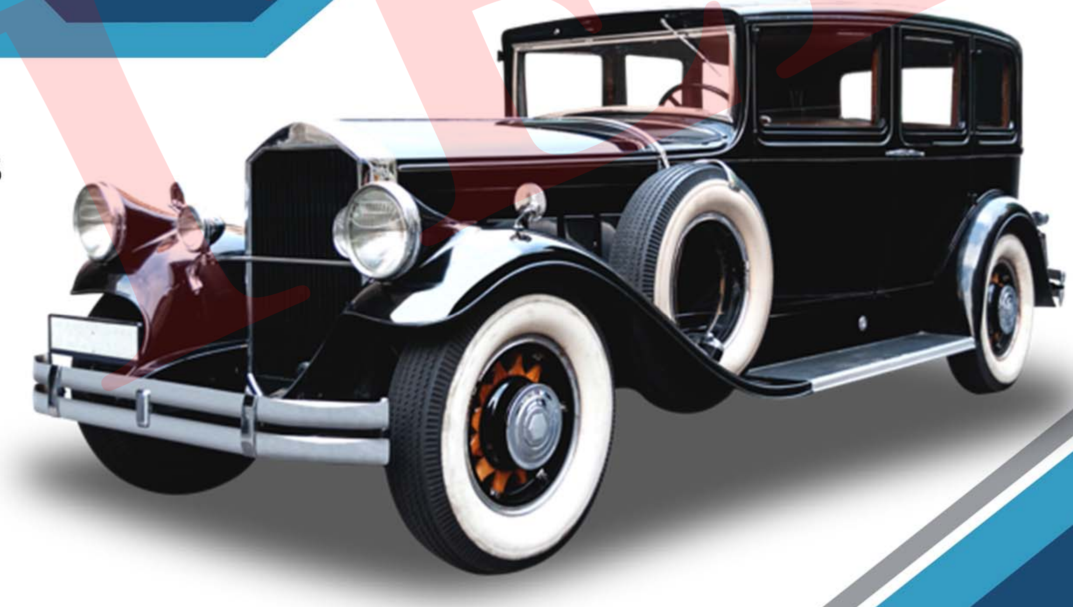
Department of Computer Science & Engineering, IIT Kharagpur

Module 03: Java Collection Framework

Lecture 10 : Java Legacy Classes

CONCEPTS COVERED

- Constituents of Java Legacy Classes
- Interfaces
- Classes
  - Constructors
  - Methods

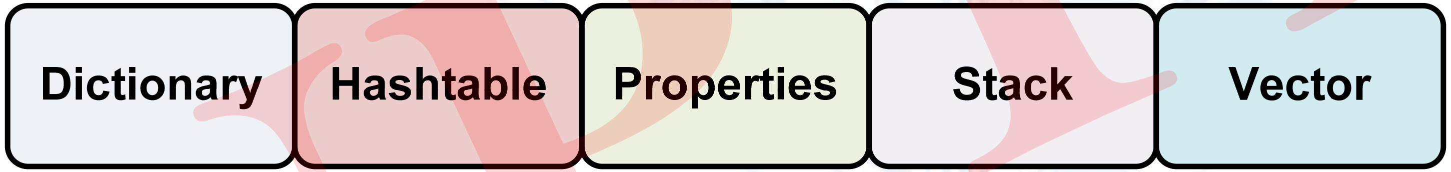


Java Legacy Classes: Background

The background

- Prior to the JCF (Java 2 and onward), the classes were known to meet the need as the JCF do for us are termed as Java legacy classes.

- The Java legacy classes are mentioned in the following.



Dictionary

Hashtable

Properties

Stack

Vector

- In addition, there is one legacy interface called Enumeration.

The background

- In fact, Java legacy classes include the classes and an interface that provided an ad hoc method of storing objects.
- Further, when Java Collections Framework were added in J2SE 1.2, the original classes were reengineered to support the collection interface.
- Furthermore, all legacy classes and interface were redesign in JDK 5 to support **Generics**.
- The Java legacy classes are not deprecated till this time and interestingly there are still codes that uses them.
- Last but not least, none of the JCF classes are synchronized, but all the legacy classes are synchronized. This may be a reason that the Java legacy classes are still in use.

Java Legacy Classes

Enumeration Interface

Interface in Java legacy classes

- `Enumeration` interface defines method to enumerate (obtain one at a time) through collection of objects.
- This interface is superseded (replaced) by Iterator interface.
- However, some legacy classes, such as `Vector` and `Properties` define several methods in which `Enumeration` interface is used.
- It has the following declaration:

```
interface Enumeration<E>
```

where E specifies the type of element being enumerated.

Methods declared in Enumeration Interface

Method	Description
boolean hasMoreElements()	It returns true while there are still more elements to extract, and returns false when all the elements have been enumerated.
Object nextElement()	It returns the next object in the enumeration i.e. each call to nextElement() method obtains the next object in the enumeration. It throws NoSuchElementException when the enumeration is complete.

Table 10.1: The methods declared by `Enumeration` interface

Class Vector

Class Vector

- `Vector` is similar to `ArrayList` which represents a dynamic array.
- There are two differences between `Vector` and `ArrayList`.
  1. Vector is synchronized while ArrayList is not.
  2. It contains many legacy methods that are not part of the JCF.
- With the release of JDK 5, `Vector` also implements `Iterable`.
  - This means that `Vector` is fully compatible with collections, and a `Vector` can have its contents iterated by the for-each loop.
- Vector is declared like this:

```
class Vector<E>
```

Here, E specifies the type of element that will be stored.

Constructors declared in Vector class

Constructor	Description
`Vector()`	This creates a default vector, which has an initial size of 10.
`Vector(int size)`	This creates a vector whose initial capacity is specified by size.
`Vector(int size, int incr)`	This creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time when a vector is resized for addition of objects.
`Vector(Collection c)`	This creates a vector that contains the elements of collection c.

Table 10.2: The constructors defined by `Vector` class

Methods defined in Vector class

Method	Description
`void addElement(E element)`	The object specified by *element* is added to the vector.
`int capacity( )`	Returns the capacity of the vector.
`Object clone( )`	Returns a duplicate of the invoking vector.
`boolean contains(Object element)`	Returns **true** if *element* is contained by the vector, and returns **false** if it is not.
`void copyInto(Object array[ ])`	The elements contained in the invoking vector are copied into the array specified by *array*.
`E elementAt(int index)`	Returns the element at the location specified by *index*.
`Enumeration<E> elements( )`	Returns an enumeration of the elements in the vector.
`void ensureCapacity(int size)`	Sets the minimum capacity of the vector to *size*.
`E firstElement( )`	Returns the first element in the vector.
`int indexOf(Object element)`	Returns the index of the first occurrence of *element*. If the object is not in the vector, –1 is returned.
`int indexOf(Object element, int start)`	Returns the index of the first occurrence of *element* at or after *start*. If the object is not in that portion of the vector, –1 is returned.
`void insertElementAt(E element, int index)`	Adds *element* to the vector at the location specified by *index*.
`boolean isEmpty( )`	Returns **true** if the vector is empty, and returns **false** if it contains one or more elements.

Table 10.3: The methods defined by `Vector` class (continued…)

Methods defined in Vector class

Method	Description
`E lastElement( )`	Returns the last element in the vector.
`int lastIndexOf(Object element)`	Returns the index of the last occurrence of *element*. If the object is not in the vector, –1 is returned.
`int lastIndexOf(Object element, int start)`	Returns the index of the last occurrence of *element* before *start*. If the object is not in that portion of the vector, –1 is returned.
`void removeAllElements( )`	Empties the vector. After this method executes, the size of the vector is zero.
`boolean removeElement(Object element)`	Removes *element* from the vector. If more than one instance of the specified object exists in the vector, then it is the first one that is removed. Returns **true** if successful and **false** if the object is not found.
`void removeElementAt(int index)`	Removes the element at the location specified by *index*.
`void setElementAt(E element, int index)`	The location specified by *index* is assigned *element*.
`void setSize(int size)`	Sets the number of elements in the vector to *size*. If the new size is less than the old size, elements are lost. If the new size is larger than the old size, **null** elements are added.
`int size( )`	Returns the number of elements currently in the vector.
`String toString( )`	Returns the string equivalent of the vector.
`void trimToSize( )`	Sets the vector's capacity equal to the number of elements that it currently holds.

Table 10.3: The methods defined by `Vector` class

Class Stack

Class Stack

- Stack is a subclass of Vector that implements a standard last-in, first-out stack.
- Stack only defines the default constructor, which creates an empty stack.
- It follows last-in, first-out principle for the stack elements.
-
- With the release of JDK 5, Stack was retrofitted for generics and is declared as shown here:

```
class Stack<E>
```

Here, E specifies the type of element stored in the stack.

Constructors declared in Stack class

Constructor	Description
`Stack()`	This creates an empty stack

Table 10.4: The constructors defined by `Stack` class

Methods defined in `Stack` class

Method	Description
`boolean empty( )`	Returns **true** if the stack is empty, and returns **false** if the stack contains elements.
`E peek( )`	Returns the element on the top of the stack, but does not remove it.
`E pop( )`	Returns the element on the top of the stack, removing it in the process.
`E push(E element)`	Pushes *element* onto the stack. *element* is also returned.
`int search(Object element)`	Searches for *element* in the stack. If found, its offset from the top of the stack is returned. Otherwise, –1 is returned.

Table 10.4: The methods defined by `Stack` class

Note:

Stack includes all the methods defined by `Vector` and adds several of its own, shown in Table 10.4 below.

Class Hashtable

Class Hashtable

- Like HashMap, Hashtable also stores key/value pair. However neither keys nor values can be null.
- There is one more difference between HashMap and Hashtable that is Hashtable is synchronized while HashMap is not.
-
- Hashtable was made generic by JDK 5. It is declared like this:

```
class Hashtable<K, V>
```

Here, K specifies the type of keys, and **V** specifies the type of values.

Constructors declared in Hashtable class

Constructor	Description
`Hashtable( )`	This is the default constructor. The default size is 11.
`Hashtable(int size)`	This creates a hash table that has an initial size specified by size.
`Hashtable(int size, float fillRatio)`	This creates a hash table that has an initial size specified by size and a fill ratio specified by fillRatio. This ratio must be between 0.0 and 1.0, and it determines how full the hash table can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hash table multiplied by its fill ratio, the hash table is expanded. If you do not specify a fill ratio, then 0.75 is used.
`Hashtable(Map<? extends K, ? extends V> m)`	This creates a hash table that is initialized with the elements in m. The capacity of the hash table is set to twice the number of elements in m. The default load factor of 0.75 is used.

Table 10.5: The constructors defined by `Hashtable` class

Methods defined in Hashtable class

Method	Description
`void clear( )`	Resets and empties the hash table.
`Object clone( )`	Returns a duplicate of the invoking object.
`boolean contains(Object value)`	Returns **true** if some value equal to *value* exists within the hash table. Returns **false** if the value isn't found.
`boolean containsKey(Object key)`	Returns **true** if some key equal to *key* exists within the hash table. Returns **false** if the key isn't found.
`boolean containsValue(Object value)`	Returns **true** if some value equal to *value* exists within the hash table. Returns **false** if the value isn't found.
`Enumeration<V> elements( )`	Returns an enumeration of the values contained in the hash table.
`V get(Object key)`	Returns the object that contains the value associated with *key*. If *key* is not in the hash table, a **null** object is returned.

Table 10.6: The methods defined by `Hashtable` class (continued…)

Methods defined in Hashtable class

Method	Description
`boolean isEmpty( )`	Returns **true** if the hash table is empty; returns **false** if it contains at least one key.
`Enumeration<K> keys( )`	Returns an enumeration of the keys contained in the hash table.
`V put(K key, V value)`	Inserts a key and a value into the hash table. Returns **null** if *key* isn't already in the hash table; returns the previous value associated with *key* if *key* is already in the hash table.
`void rehash( )`	Increases the size of the hash table and rehashes all of its keys.
`V remove(Object key)`	Removes *key* and its value. Returns the value associated with *key*. If *key* is not in the hash table, a **null** object is returned.
`int size( )`	Returns the number of entries in the hash table.
`String toString( )`	Returns the string equivalent of a hash table.

Table 10.6: The methods defined by `Hashtable` class

Class Properties

Class Properties

- `Properties` class extends `Hashtable` class.
- It is used to maintain list of values in which both key and value are `String`.
- One advantage of `Properties` over `Hashtable` is that we can specify a default property that will be useful when no value is associated with a certain key.
- In `Properties` class, you can specify a default property that will be returned if no value is associated with a certain key.
- `Properties` defines the following instance variable:

```
Properties defaults;
```

Constructors declared in Properties class

Constructor	Description
`Properties( )`	This creates a Properties object that has no default values
`Properties(Properties propDefault)`	This creates an object that uses propdefault for its default values.

Table 10.7: The constructors defined by `Properties` class

Methods defined in Properties class

Method	Description
`String getProperty(String key)`	Returns the value associated with *key*. A **null** object is returned if *key* is neither in the list nor in the default property list.
`String getProperty(String key, String defaultProperty)`	Returns the value associated with *key*. *defaultProperty* is returned if *key* is neither in the list nor in the default property list.
`void list(PrintStream streamOut)`	Sends the property list to the output stream linked to *streamOut*.
`void list(PrintWriter streamOut)`	Sends the property list to the output stream linked to *streamOut*.
`void load(InputStream streamIn) throws IOException`	Inputs a property list from the input stream linked to *streamIn*.
`void load(Reader streamIn) throws IOException`	Inputs a property list from the input stream linked to *streamIn*.
`void loadFromXML(InputStream streamIn) throws IOException, InvalidPropertiesFormatException`	Inputs a property list from an XML document linked to *streamIn*.
`Enumeration<?> propertyNames( )`	Returns an enumeration of the keys. This includes those keys found in the default property list, too.

Table 10.8: The methods defined by `Properties` class (continued…)

Methods defined in Properties class

Method	Description
`Object setProperty(String key, String value)`	Associates *value* with *key*. Returns the previous value associated with *key*, or returns **null** if no such association exists.
`void store(OutputStream streamOut, String description) throws IOException`	After writing the string specified by *description*, the property list is written to the output stream linked to *streamOut*.
`void store(Writer streamOut, String description) throws IOException`	After writing the string specified by *description*, the property list is written to the output stream linked to *streamOut*.
`void storeToXML(OutputStream streamOut, String description) throws IOException`	After writing the string specified by *description*, the property list is written to the XML document linked to *streamOut*.
`void storeToXML(OutputStream streamOut, String description,String enc)`	The property list and the string specified by *description* is written to the XML document linked to *streamOut* using the specified character encoding.
`Set<String> stringPropertyNames( )`	Returns a set of keys.

Table 10.8: The methods defined by `Properties` class

`store()` and `load()` methods

Note:

1. One of the most useful aspects of `Properties` is that the information contained in a `Properties` object can be easily stored to or loaded from disk with the `store( )` and `load( )` methods.

2. At any time, you can write a `Properties` object to a stream or read it back. This makes property lists especially convenient for implementing simple databases.

Class Dictionary

Class Dictionary

- `Dictionary` is an abstract class.
- It represents a key/value pair and operates much like Map.
- Although it is not currently deprecated, **Dictionary** is classified as obsolete, because it is fully superseded by Map class.

Methods declared in Dictionary class

Method	Description
`Enumeration<V> elements( )`	Returns an enumeration of the values contained in the dictionary.
`V get(Object key)`	Returns the object that contains the value associated with *key*. If *key* is not in the dictionary, a **null** object is returned.
`boolean isEmpty( )`	Returns **true** if the dictionary is empty, and returns **false** if it contains at least one key.
`Enumeration<K> keys( )`	Returns an enumeration of the keys contained in the dictionary.
`V put(K key, V value)`	Inserts a key and its value into the dictionary. Returns **null** if *key* is not already in the dictionary; returns the previous value associated with *key* if *key* is already in the dictionary.
`V remove(Object key)`	Removes *key* and its value. Returns the value associated with *key*. If *key* is not in the dictionary, a **null** is returned.
`int size( )`	Returns the number of entries in the dictionary.

Table 10.9: The methods declared by `Dictionary` class

REFERENCES

- https://cse.iitkgp.ac.in/~dsamanta/javads/index.html
- https://docs.oracle.com/javase/tutorial/

THANK YOU !