Unit-3: Collection Classes

22UCSC401 – OBJECT ORIENTED PROGRAMMING

4TH SEMESTER B DIVISION ACADEMIC YEAR: 2023 – 24

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, SDMCET, DHARWAD-2

The Collections Framework

CHAPTER-18: JAVA THE COMPLETE REFERENCE NINTH EDITION, HERBERT SCHILDT

Preamble & Introduction

- ✓ The *Collections Framework* is a sophisticated hierarchy of *interfaces and classes* that provide state-of-the-art technology for managing groups of objects
- ✓ It standardizes handling groups of objects by the programs
- ✓ It was added by J2SE 1.2
- ✓ Prior to J2SE 1.2, Java provided ad hoc classes such as **Dictionary**, **Vector**, **Stack**, and **Properties** to store and manipulate groups of objects
- ✓ The Collections Framework was designed to meet following goals:
  1) The framework had to be high-performance
  2) The framework had to allow different types of collections to work in similar manner and with high degree of interoperability
  3) Extending and/or adapting a collection had to be easy

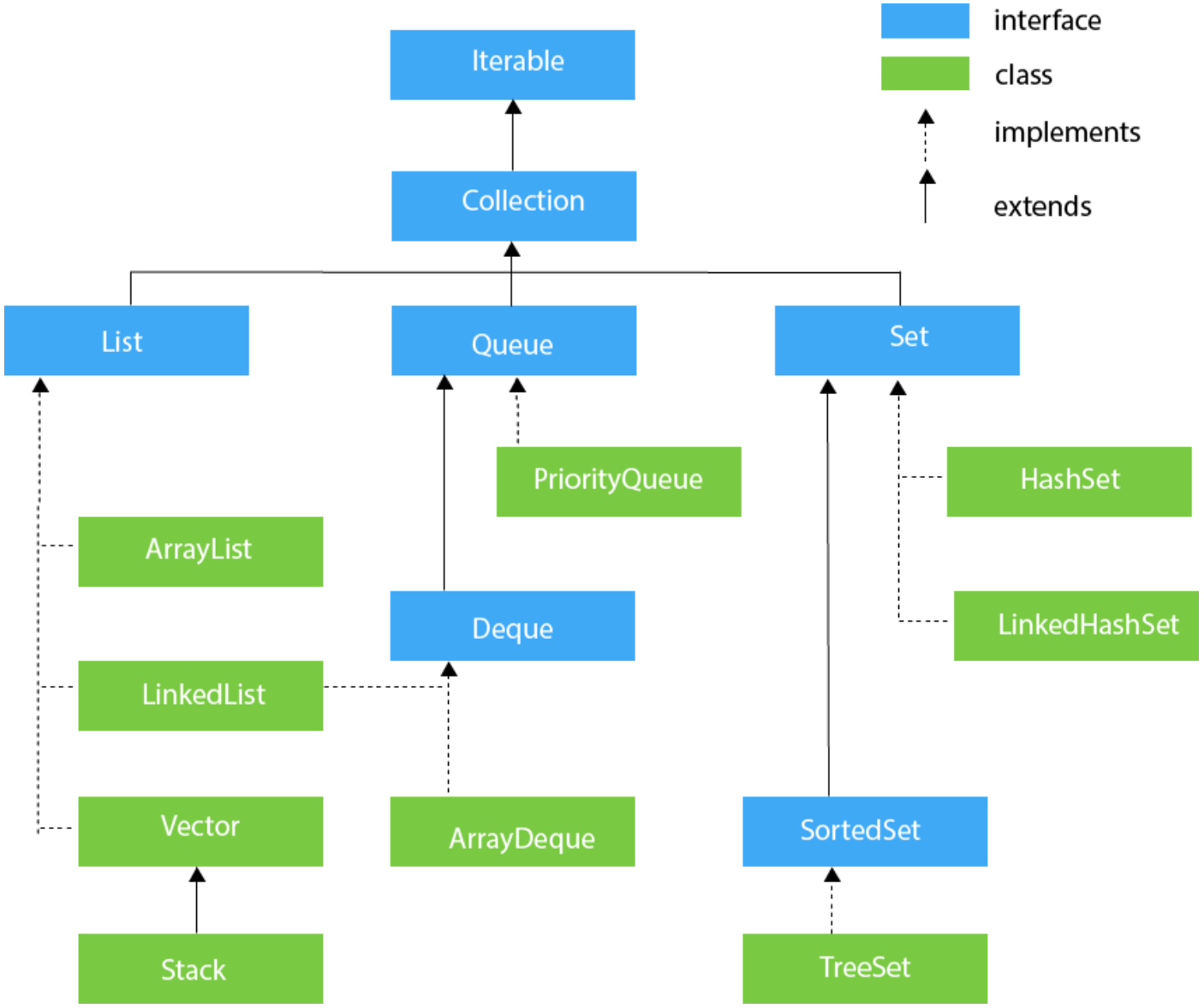
Introduction

- ✓ ***Algorithms*** are important part of collection mechanism
- ✓ Algorithms operate on collections and are defined as static methods within **Collection** class
- ✓ They are available for all collections
- ✓ The algorithms provide standard means of manipulating collections
- ✓ An ***iterator*** offers general-purpose, standardized way of accessing elements within a collection, one at a time
- ✓ Iterator provides means of *enumerating the contents of a collection*
- ✓ JDK 8 adds another type of iterator called a *spliterator*
- ✓ Spliterators are iterators that provide support for parallel iteration

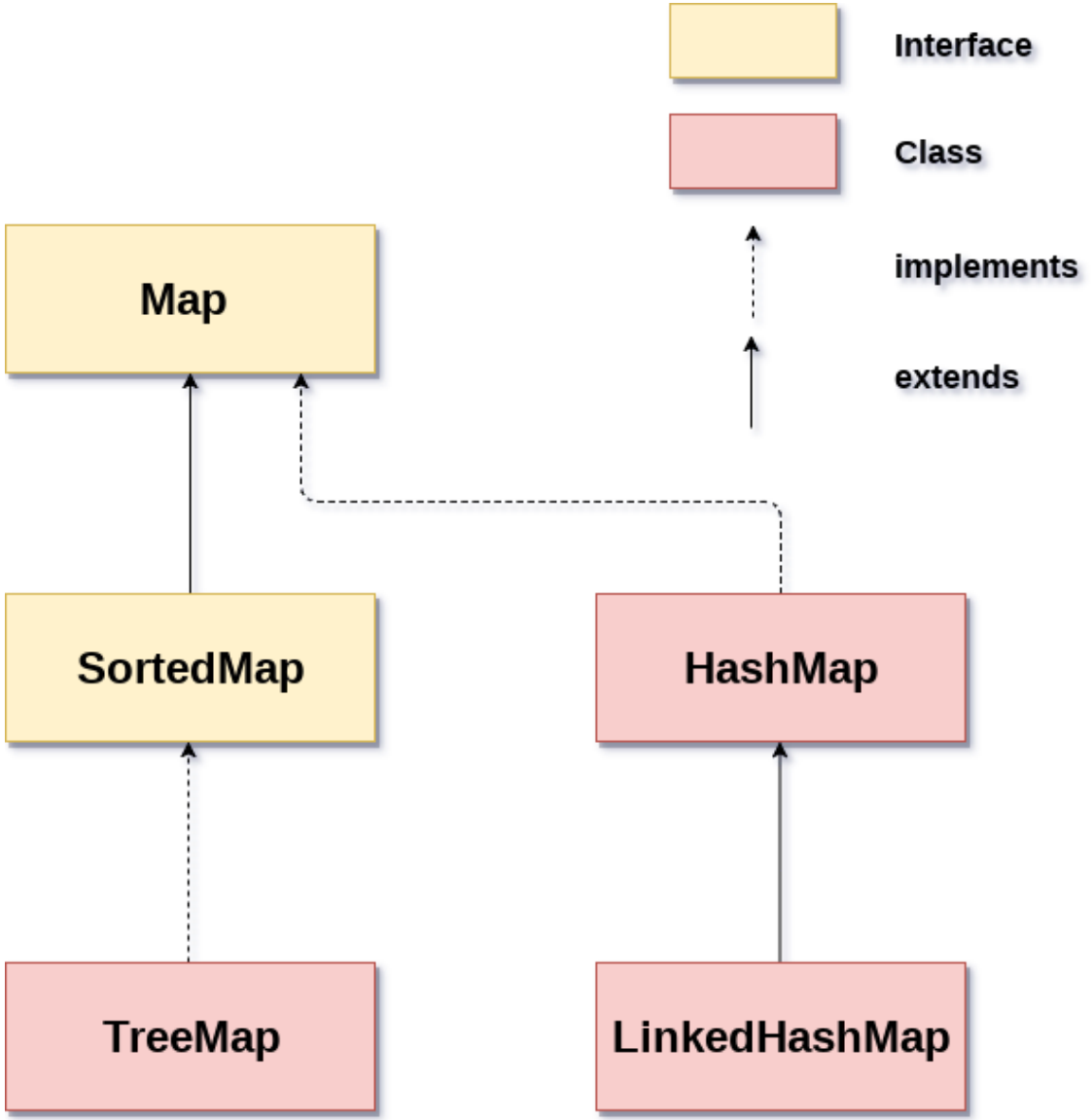
Introduction (contd…)

- ✓ **java.util** package contains all the classes and interfaces for Collection framework
- ✓ The collection framework defines several map interfaces and classes
- ✓ Maps store key/value pairs
- ✓ JDK5 changed collections framework by incorporating features like ***generics, autoboxing/ unboxing***, *and* ***for-each*** *style for loop*
- ✓ Generics fundamentally changed Collections Framework
- ✓ Generics added type safety feature to Collections Framework
- ✓ Autoboxing/ unboxing facilitates the use of primitive types in collections
- ✓ All collection classes in Collections Framework were retrofitted to implement **Iterable** interface
  - ✓ It means that a collection can be cycled through the use of **for-each style** for loop

Collection Framework Hierarchy



Collection Framework Hierarchy



The Collection Interfaces

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

interface Collection<E>

Here, **E** specifies the type of objects that the collection will hold. **Collection** extends the **Iterable** interface. This means that all collections can be cycled through by use of the for-each style **for** loop.

Method	Description
boolean add(E *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 E> *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**.
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<E> 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. (Added by JDK 8.)
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 E> *predicate*)	Removes from the invoking collection those elements that satisfy the condition specified by *predicate*. (Added by JDK 8.)

Table 18-1 The Methods Declared by **Collection**

Method	Description
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**.
int size( )	Returns the number of elements held in the invoking collection.
default Spliterator<E> spliterator( )	Returns a spliterator to the invoking collections. (Added by JDK 8.)
default Stream<E> stream( )	Returns a stream that uses the invoking collection as its source for elements. The stream is sequential. (Added by JDK 8.)
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**. An **ArrayStoreException** is thrown if any collection element has a type that is not a subtype of *array*.

Table 18-1 The Methods Declared by **Collection** *(continued)*

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.
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. (Added by JDK 8.)
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*. (Added by JDK 8.)
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 18-2 The Methods Declared by **List**

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 18-3 The Methods Declared by **SortedSet**

Method	Description
E 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(E *obj*)	Attempts to add *obj* to the queue. Returns **true** if *obj* was added and **false** otherwise.
E peek( )	Returns the element at the head of the queue. It returns **null** if the queue is empty. The element is not removed.
E poll( )	Returns the element at the head of the queue, removing the element in the process. It returns **null** if the queue is empty.
E 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 18-5 The Methods Declared by **Queue**

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.
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.

Table 18-6 The Methods Declared by **Deque**

Method	Description
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 18-6 The Methods Declared by **Deque** *(continued)*

The Collection Classes

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**.

The ArrayList Class

- **ArrayList** class extends **AbstractList**, and implements **List** interface
- **ArrayList** is a generic class
- **class ArrayList<E>**
  - **E** → type of objects that the list will hold
- **ArrayList** supports dynamic arrays that can grow as needed
- Standard arrays are of a fixed length
  - i.e., after arrays are created, they cannot grow or shrink
  - we must know in advance how many elements an array will hold
  - Sometimes, you may not know until run time precisely how large an array needs to be
- To handle this situation, the Collections Framework defines **ArrayList**

The ArrayList Class (contd…)

- ✓ In essence, an **ArrayList** is a variable-length array of object references
- ✓ **ArrayList** can dynamically increase or decrease in size
- ✓ Array lists are created with an initial size
- ✓ When this size is exceeded, the collection is automatically enlarged
- ✓ When objects are removed, the array can be shrunk

The ArrayList Class (contd…)

- **ArrayList** has three constructors:
- **ArrayList()**
  - Builds an empty array list
- **ArrayList(Collection<? extends E> *c*)**
  - Builds an array list that is initialized with elements of collection ***c***
- **ArrayList(int *capacity*)**
  - Builds an array list that has the specified initial ***capacity***
  - The capacity is the size of the array that is used to store elements
  - The capacity grows automatically as elements are added to an array list

The ArrayList Class (contd…)

- **ensureCapacity()** method is used to manually increase the capacity of array list
- **void ensureCapacity(int *cap*)**
  - ***cap*** specifies the new minimum capacity of the collection
- By increasing its capacity once, at the start, we can prevent several re-allocations later
  - Because re-allocations are costly in terms of time
  - Preventing unnecessary re-allocations improves performance
- If we want to reduce the size of an **ArrayList** object so that it is precisely as large as the no. of items that it is currently holding, call **trimToSize()** method
- **void trimToSize()**

Array vs ArrayList

Array	ArrayList
Static. i.e., their size cannot be changed once they are created	**Dynamic**. i.e., its size is automatically increased if elements are added beyond its capacity
Can hold both primitive and object types	Can hold only object types
Can be iterated only through **for loop or for-each loop**	**Iterator** is used to iterate through the elements
Does not support Generics	Supports Generics
Are not type safe	Are type safe
Can be multi-dimensional	Can only be singe dimensional

The LinkedList class

- **LinkedList** class extends **AbstractSequentialList** and implements **List**, **Deque**, and **Queue** interfaces
- **class LinkedList<E>**
  - **E** → type of objects that list will hold
- Constructors of **LinkedList** class:
- **LinkedList()**
  - Builds an empty linked list
- **LinkedList(Collection<? extends E> *c*)**
  - builds a linked list that is initialized with elements of collection ***c***

Vector

- **Vector** implements a dynamic array
- It is similar to **ArrayList** with two differences:
  - It is synchronized
  - It contains many legacy methods that duplicate the functionality of methods defined by Collections Framework
- **Vector** was reengineered to extend **AbstractList** and to implement **List** interface
- With JDK5, it was retrofitted for generics and reengineered to implement **Iterable**
- Currently **Vector** is fully compatible with collections, and can have its contents iterated by the enhanced for loop
- **class Vector<E>**
  - **E** → type of element that will be stored

Vector (contd..)

- **Vector()**
  - creates a default vector with an initial value of 10
- **Vector(int *size*)**
  - creates a vector whose initial capacity is specified by ***size***
- **Vector(int *size*, int *incr*)**
  - creates a vector whose initial capacity is specified by ***size*** and whose increment is specified by ***incr***
  - increment specifies the no. of elements to allocate each time a vector is resized upward
- **Vector(Collection<? extends E> *c*)**
  - creates a vector that contains elements of collection ***c***
- If ***incr*** is not specified, the vector's size **is doubled** by each allocation cycle

Vector (contd..)

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.

Table 18-16 The Legacy Methods Defined by **Vector**

Method	Description
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.
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 18-16 The Legacy Methods Defined by **Vector** *(continued)*

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
- ✓ With JDK5, **Stack** was retrofitted for generics
- ✓ **class Stack<E>**
  - ✓ **E** → type of element stored in the stack
- ✓ **Stack** includes all the methods defined by **Vector** and adds several of its own
- ✓ Although **Stack** is not deprecated, **ArrayDeque** is better choice

Stack (contd…)

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 18-17 The Methods Defined by **Stack**

Practice Programs

- ✓ Write a Java program to convert a given infix expression to postfix expression using Stack collection class
- ✓ Postgraduate course of the CSE department can accommodate a maximum of 20 students. Each PG student in the department is identified by his/her Roll No, USN, Name, Semester and Mobile Number. Using appropriate Collection class, write a Java Program to simulate the following scenarios:
  1) On request, the capacity was increased from 20 to 25
  2) Only 3 students enroll for the course
  3) Two students bearing Roll Numbers 2 and 3 voluntarily unroll from the course
  4) After a month, two students get enrolled in the course
  5) Print the information of all the students currently enrolled in the course

Accessing a Collection via an Iterator

- ✓ **Iterator** can be used to cycle through a collection, obtaining or removing elements
- ✓ **ListIterator** extends **Iterator** to allow bidirectional traversal of a list, and for modification of elements
- ✓ **interface Iterator<E>**
- ✓ **interface ListIterator<E>**
- ✓ **E** → type of objects being iterated

Accessing a Collection via an Iterator (contd…)

Method	Description
default void forEachRemaining(Consumer<? super E> *action*)	The action specified by *action* is executed on each unprocessed element in the collection. (Added by JDK 8.)
boolean hasNext( )	Returns **true** if there are more elements. Otherwise, returns **false**.
E next( )	Returns the next element. Throws **NoSuchElementException** if there is not a next element.
default void remove( )	Removes the current element. Throws **IllegalStateException** if an attempt is made to call **remove( )** that is not preceded by a call to **next( )**. The default version throws an **UnsupportedOperationException**.

Table 18-8 The Methods Declared by **Iterator**

Using an Iterator

✓ To use an iterator to cycle through contents of a collection, follow these steps:

1) Obtain an iterator to the start of the collection by calling the collection's **iterator()** method
2) Set up a loop that makes a call to **hasNext()**. Have the loop iterate as long as **hasNext()** returns **true**
3) Within the loop, obtain each element by calling **next()**

Spliterator

- ✓ JDK 8 adds new type of iterator called *spliterator*, which is defined by **Spliterator** interface
- ✓ It offers more functionality than **Iterator** or **ListIterator**
- ✓ It supports *parallel programming*
- ✓ **interface Spliterator<T>**
  - ✓ **T** → type of elements being iterated

Method	Description
int characteristics( )	Returns the characteristics of the invoking spliterator, encoded into an integer.
long estimateSize( )	Estimates the number of elements left to iterate and returns the result. Returns **Long.MAX_VALUE** if the count cannot be obtained for any reason.
default void forEachRemaining(Consumer<? super T> *action*)	Applies *action* to each unprocessed element in the data source.
default Comparator<? super T> getComparator( )	Returns the comparator used by the invoking spliterator or **null** if natural ordering is used. If the sequence is unordered, **IllegalStateException** is thrown.
default long getExactSizeIfKnown( )	If the invoking spliterator is sized, returns the number of elements left to iterate. Returns –1 otherwise.
default boolean hasCharacteristics(int *val*)	Returns **true** if the invoking spliterator has the characteristics passed in *val*. Returns **false** otherwise.
boolean tryAdvance(Consumer<? super T> *action*)	Executes *action* on the next element in the iteration. Returns **true** if there is a next element. Returns **false** if no elements remain.
Spliterator<T> trySplit( )	If possible, splits the invoking spliterator, returning a reference to a new spliterator for the partition. Otherwise, returns **null**. Thus, if successful, the original spliterator iterates over one portion of the sequence and the returned spliterator iterates over the other portion.

Table 18-10 The Methods Declared by **Spliterator**

The HashSet class

- ✓ **HashSet** extends **AbstractSet** and implements **Set** interface
- ✓ It creates a collection that uses *hash table* for storage
- ✓ **class HashSet<E>**
  - ✓ **E** → type of objects that set will hold
- ✓ A hash table stores information by using a mechanism called *hashing*
- ✓ Hashing is an algorithm to map object data to some representative integer values
- ✓ In hashing, informational content of a key is used to determine a unique value, called its *hash code*
- ✓ The hash code is then used as an index at which data associated with key is stored
- ✓ The transformation of key into its hash code is performed automatically
- ✓ The code can't directly index the hash table

The HashSet class (contd…)

- **HashSet()**
  - Constructs a default hash set
- **HashSet(Collection<? extends E> *c*)**
  - initializes hash set by using elements of ***c***
- **HashSet(int *capacity*)**
  - initializes capacity of hash set to ***capacity***
- **HashSet(int *capacity*, float *fillRatio*)**
  - initializes both ***capacity*** and *fill ratio* (also called load capacity) of hash set from its arguments
  - fill ratio must be between 0.0 and 1.0. It determines how full hash set can be before it is resized upward
  - when no. of elements is greater than capacity of hash set multiplied by its fill ratio, then hash set is expanded
  - For constructors that do not take fill ratio, 0.75 is used

The HashSet class (contd...)

- ✓ **HashSet** does not guarantee order of its elements
- ✓ If we need sorted storage, then collection, such as **TreeSet**, is a better choice

Working with Maps

- A *map* is an object that stores associations between keys and values, or *key/value* pairs
- Given a key, you can find its value
- Both keys and values are objects
- The keys must be unique, but values may be duplicated
- **Maps don't implement Iterable interface**
- However, we can obtain a *collection-view* of a map
  - This allows the use of either *for* loop or an *iterator*

Working with Maps (contd...)

Interface	Description
Map	Maps unique keys to 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.

The Map Interface

- **Map** interface maps unique keys to values
- A *key* is an object that we use to retrieve a value
- Given a key and a value, we can store the value in a **Map** object
- After the value is stored, we can retrieve it by using its key
- **interface Map<K, V>**
  - **K** → type of keys; **V** → type of values
- Two basics operations on maps: **get()** and **put()**
- **put()** is used to put value into map by specifying key and value
- **get()** is used to obtain value by passing key as argument

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. (Added by JDK 8.)
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. (Added by JDK 8.)
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. (Added by JDK 8.)
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. (Added by JDK 8.)
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. (Added by JDK 8.)
int hashCode( )	Returns the hash code for the invoking map.
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.

Table 18-11 The Methods Declared by **Map**

Method	Description
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. (Added by JDK 8.)
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**. (Added by JDK 8.)
V remove(Object *k*)	Removes the entry whose key equals *k*.
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. (Added by JDK 8.)
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. (Added by JDK 8.)
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. (Added by JDK 8.)
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. (Added by JDK 8.)
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 18-11 The Methods Declared by **Map** *(continued)*

The 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.
WeakHashMap	Extends **AbstractMap** to use a hash table with weak keys.
LinkedHashMap	Extends **HashMap** to allow insertion-order iterations.
IdentityHashMap	Extends **AbstractMap** and uses reference equality when comparing documents.

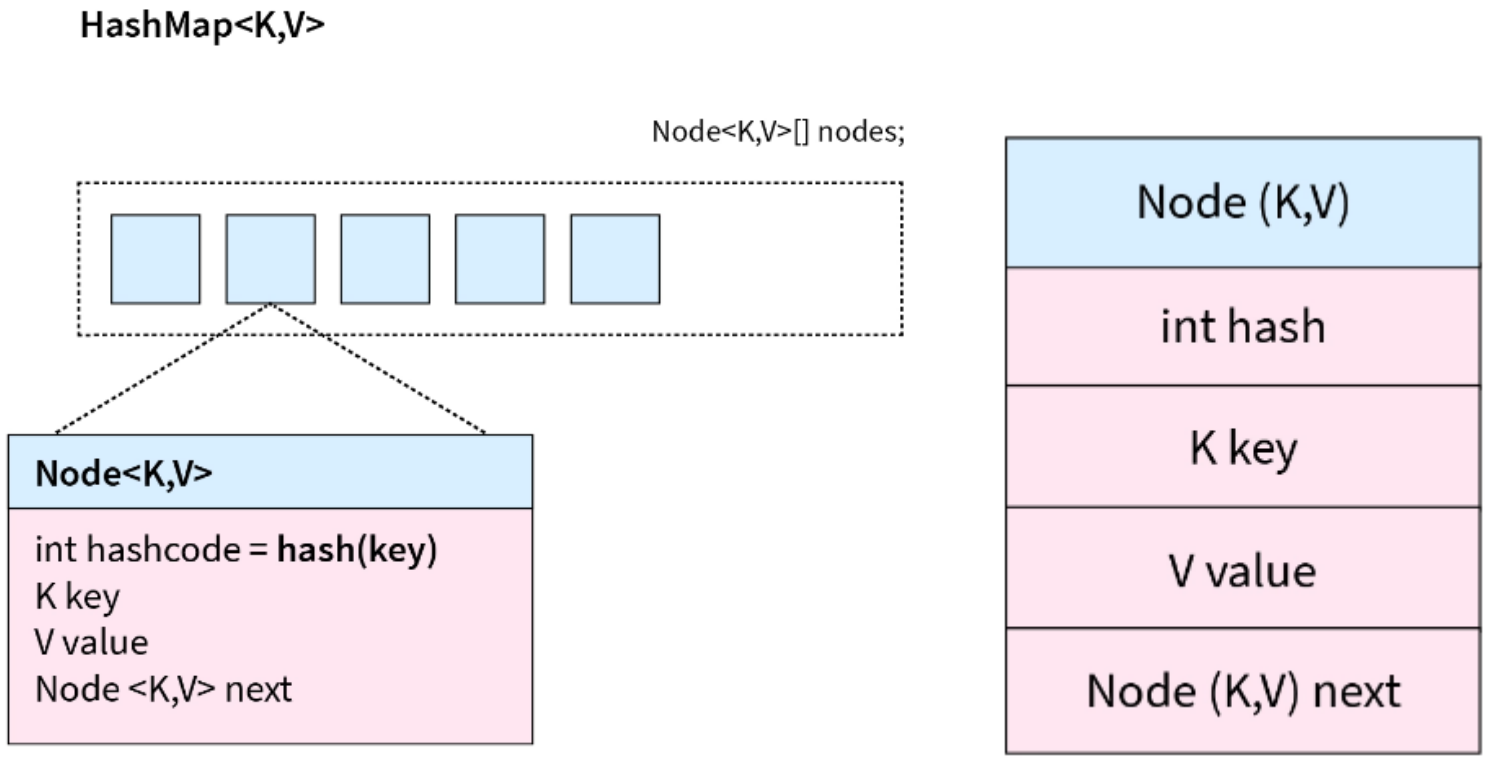
The HashMap Class

- **HashMap** class extends **AbstractMap** and implements **Map** interface
- It uses a hash table to store the map
- **class HashMap<K, V>**
  - **K** → type of keys; **V** → type of values
- **HashMap()**
  - constructs a default hash map
- **HashMap(Map<? extends K, ? extends V> *m*)**
  - initializes hash map by using elements of *m*
- **HashMap(int *capacity*)**
  - Initializes capacity of hash map to *capacity*
- **HashMap(int *capacity*, float *fillRatio*)**
  - initializes both capacity and fill ratio of hash map by using its arguments
  - Default capacity is 16
  - Default fill ratio is 0.75

Features of HashMap in Java

- ✓ **HashMap** in Java uses the technique of Hashing
  - ✓ Hashing function maps a big number or string to a small integer that can be used as an index
  - ✓ A shorter value helps in indexing and faster searches
- ✓ In **HashMap**, the keys should be unique but values can be duplicated
  - ✓ i.e., X key can't contain more than one value (X mapped to only one value)
  - ✓ However, other Y, Z keys can contain duplicate or the same values already mapped to X key
- ✓ **HashMap** in Java can also have **null** key and values
  - ✓ There could be only one **null** key
  - ✓ There could be any number of **null** values corresponding to different keys
- ✓ Unlike **HashTable**, **HashMap** is *not synchronized*. It is an unordered collection that doesn't guarantee any specific order of the elements
- ✓ To retrieve any value from **HashMap**, one must know the associated key

The HashMap Class (contd...)



Comparators

- By default, **TreeSet** and **TreeMap** store elements by using *natural ordering*
- If we want to order elements in different way, then we must specify a **Comparator** when *set* or *map* are constructed
- Use of **Comparator** gives ability to precisely govern how elements are stored within sorted collections and maps
- A **Comparator** in Java is a generic interface
- **interface Comparator<T>**
  - **T** → type of objects being compared

Methods of Comparator

Method Name	Description
int compare(T *obj1*, T *obj2*)	Compares two elements for order. Returns zero if the objects are equal. returns a positive value if *obj1* is greater than *obj2*. Otherwise, a negative value is returned
boolean equals(object *obj*)	Tests whether an object equals the invoking Comparator. *obj* is the object to be tested for equality. Returns **true** if *obj* and the invoking object are both **Comparator** objects and use the same ordering. Otherwise, it returns **false**
default Comparator<T> reversed()	(JDK8) reverses the ordering of the comparator on which it is called.

Methods of Comparator (contd…)

Method Name	Description
static <T> Comparator<T> nullsFirst(Comparator<? super T> *comp*) **static <T> Comparator<T> nullsLast(Comparator<? super T> *comp*)**	(JDK8) Comparator to handle **null** values
default Comparator<T> thenComparing(Comparator<? super T> *thenByComp*)	(JDK8) Returns a comparator that performs a second comparison when the outcome of the first comparison indicates that the objects being compared are equal. i.e., **compare by X then compare by Y**

Thank You