Application Program Development

APD545

Instructor: Maryam Sepehrinour
Email: Maryam.Sepehrinour@SenecaPolytechnic.ca

Seneca

Outcomes

- ✓ Java Collection Framework
- ✓ Java Iterators

Outcomes

- ✓ **Java Collection Framework**
- ✓ Java Iterators

Collections Frameworks

- A *collection* is an object that represents a group of objects

- A collections framework is a unified architecture for representing and manipulating collections
  - the manipulation should be independent of the implementation details

- Advantages of having a collections framework
  - reduces *programming effort* by providing useful data structures and algorithms
  - increases *performance* by providing high-performance data structures and algorithms
  - *interoperability* between the different collections
  - Having a *common language* for manipulating the collections

The components of Java Collections Framework from the user's perspective

- Collection Interfaces: form the basis of the framework
  - such as sets, lists and maps

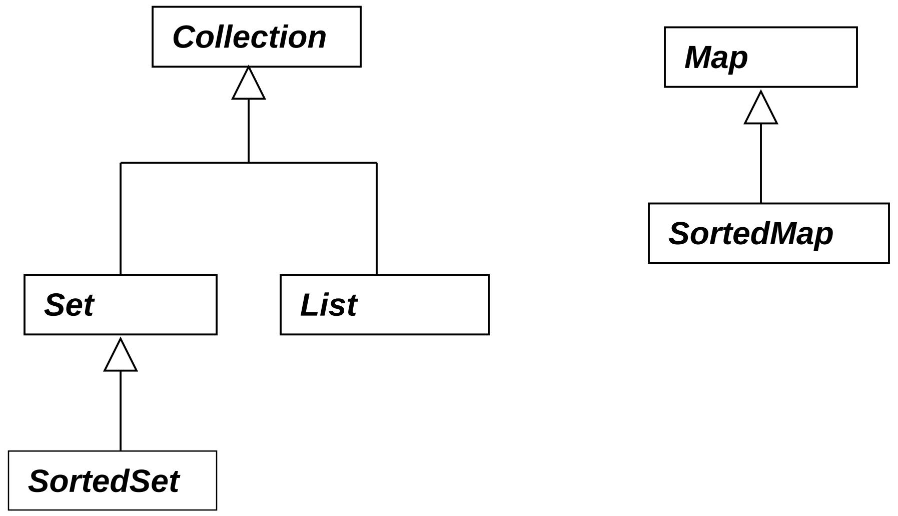
- Implementations Classes: implementations of the collection interfaces

- Utilities - Utility functions for manipulating collections such as sorting, searching…

- The Java Collections Framework also includes:
  - algorithms
  - *wrapper* implementations
    - add functionality to other implementations such as synchronization

Collections Interfaces

- Java allows using:
  - Lists
  - Sets
  - Hash tables (maps)



Interface: Collection

- The Collection interface is the root of the collection hierarchy
- Some Collection implementations allow
  - duplicate elements and others do not
  - the elements to be ordered or not
- JDK doesn't provide any direct implementations of this interface
  - It provides implementations of more specific sub interfaces like Set and List

```
public interface Collection {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element);
    boolean remove(Object element);
    Iterator iterator();

    // Bulk Operations
    boolean containsAll(Collection c);
    boolean addAll(Collection c);
    boolean removeAll(Collection c);
    boolean retainAll(Collection c);
    void clear();

    // Array Operations
    Object[] toArray();
    Object[] toArray(Object a[]);
```

Interface: Set

- A Set is a collection that cannot contain duplicate elements
- A set is not ordered
  - however, some subsets maintain order using extra data structures
- This is similar to the mathematical concept of *sets*

```
public interface Set {
   // Basic Operations
   int size();
   boolean isEmpty();
   boolean contains(Object element);
   boolean add(Object element);
   boolean remove(Object element);
   Iterator iterator();

   // Bulk Operations
   boolean containsAll(Collection c);
   boolean addAll(Collection c);
   boolean removeAll(Collection c);
   boolean retainAll(Collection c);
   void clear();

   // Array Operations
   Object[] toArray();
   Object[] toArray(Object a[]);
}
```

Interface: List

- A List <u>is ordered</u> (also called a *sequence*)
- Lists <u>can</u> contain duplicate elements
- The user can access elements by their integer index (position)

```
public interface List extends Collection {
   // Positional Access
   Object get(int index);
   Object set(int index, Object element);
   void add(int index, Object element);
   Object remove(int index);
   abstract boolean addAll(int index, Collection c);

   // Search
   int indexOf(Object o);
   int lastIndexOf(Object o);

   // Iteration
   ListIterator listIterator();
   ListIterator listIterator(int index);

   // Range-view
   List subList(int from, int to);
}
```

Interface: Map

- A Map is an object that maps keys to values
- Maps cannot contain duplicate keys
- Each key can map to at most one value
- Hashtables are an example of Maps

```
public interface Map {
   // Basic Operations
   Object put(Object key, Object value);
   Object get(Object key);
   Object remove(Object key);
   boolean containsKey(Object key);
   boolean containsValue(Object value);
   int size();
   boolean isEmpty();

   // Bulk Operations
   void putAll(Map t);
   void clear();

   // Collection Views
   public Set keySet();
   public Collection values();
   public Set entrySet();

   // Interface for entrySet elements
   public interface Entry {
      Object getKey();
      Object getValue();
      Object setValue(Object value);
   }
}
```

Interface: SortedSet

- A SortedSet is a Set that maintains its elements in an order
- Several additional operations are provided to take advantage of the ordering

```
public interface SortedSet extends Set {
   // Range-view
   SortedSet subSet(Object fromElement,
Object toElement);
   SortedSet headSet(Object toElement);
   SortedSet tailSet(Object fromElement);

   // Endpoints
   Object first();
   Object last();

   // Comparator access
   Comparator comparator();
}
```

Interface: SortedMap

- A SortedMap is a Map that maintains its mappings in ascending key order
- The SortedMap interface is used for apps like dictionaries and telephone directories

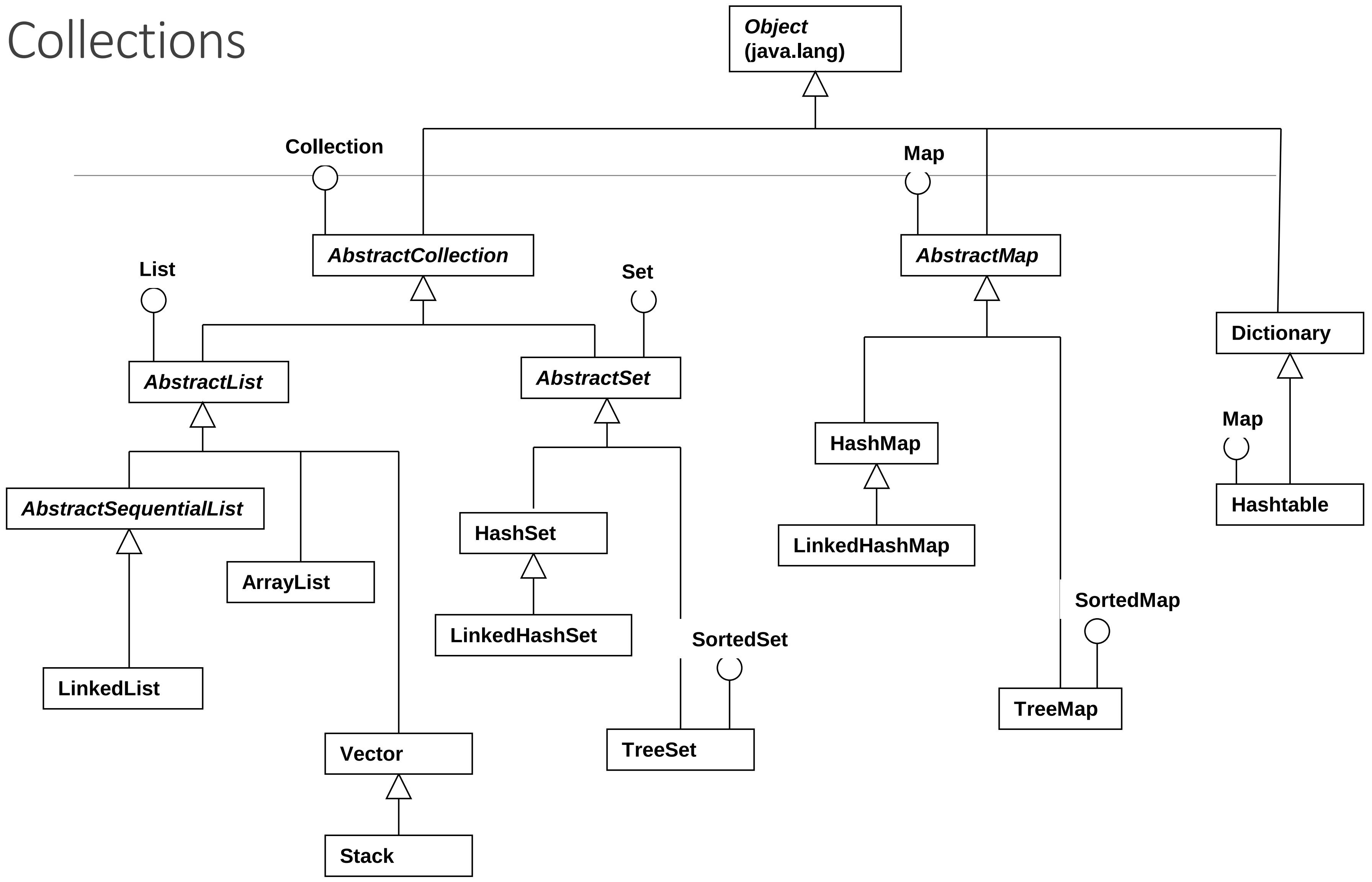
```
public interface SortedMap extends Map {
   Comparator comparator();

   SortedMap subMap(Object fromKey,
Object toKey);
   SortedMap headMap(Object toKey);
   SortedMap tailMap(Object fromKey);

   Object firstKey();
   Object lastKey();
}
```

Seneca

Java Collections

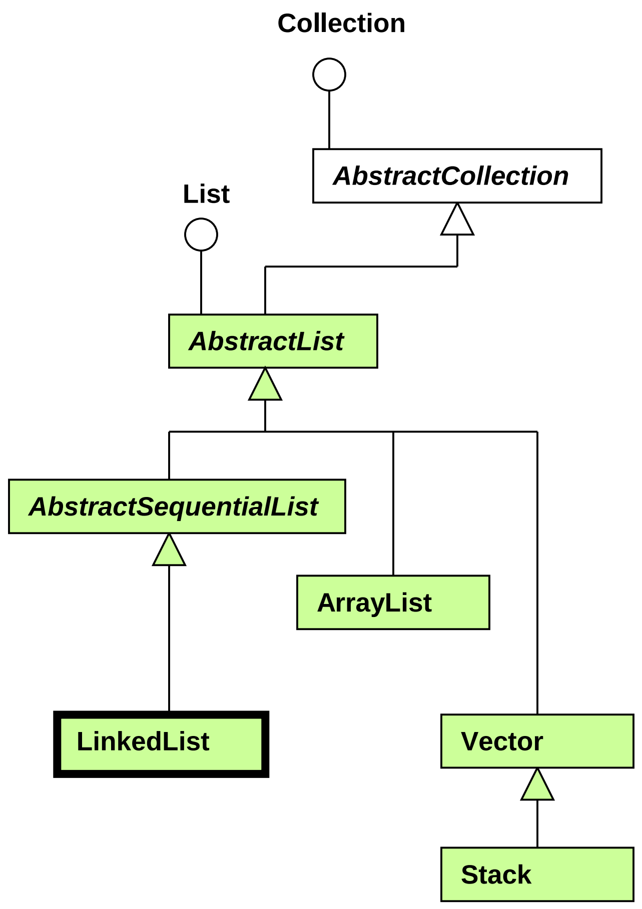


Java Lists: LinkedList

- Java uses a doubly-linked list
  - it can be traversed from the beginning or the end

- LinkedList provides methods to get, remove and insert an element at the beginning and end of the list
  - these operations allow a linked list to be used as a stack or a queue

- LinkedList is not synchronized
  - problems if multiple threads access a list concurrently
  - LinkedList *must* be synchronized externally



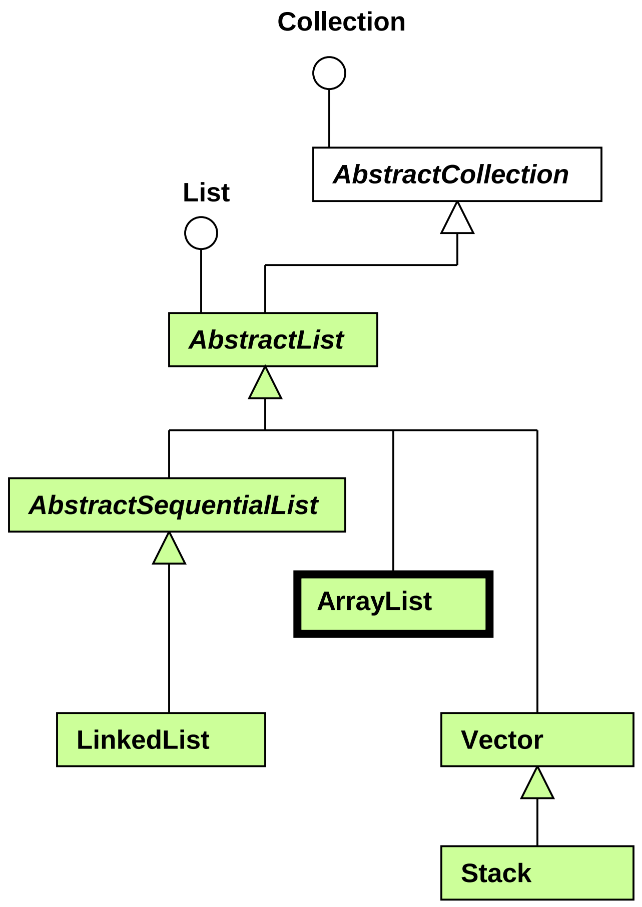
Java Lists: ArrayList

- Uses an array to store the elements

- In addition to the methods of the interface List
  - it provides methods to manipulate the size of the array (e.g. ensureCapacity)

- More efficient than LinkedList for methods involving indices – get(), set()

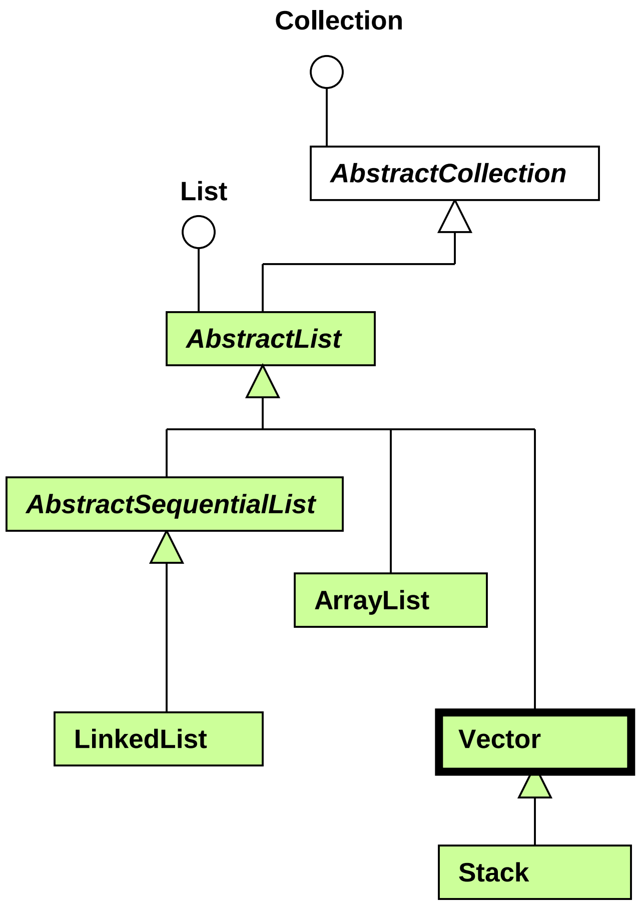
- It is not synchronized



Java Lists: Vector

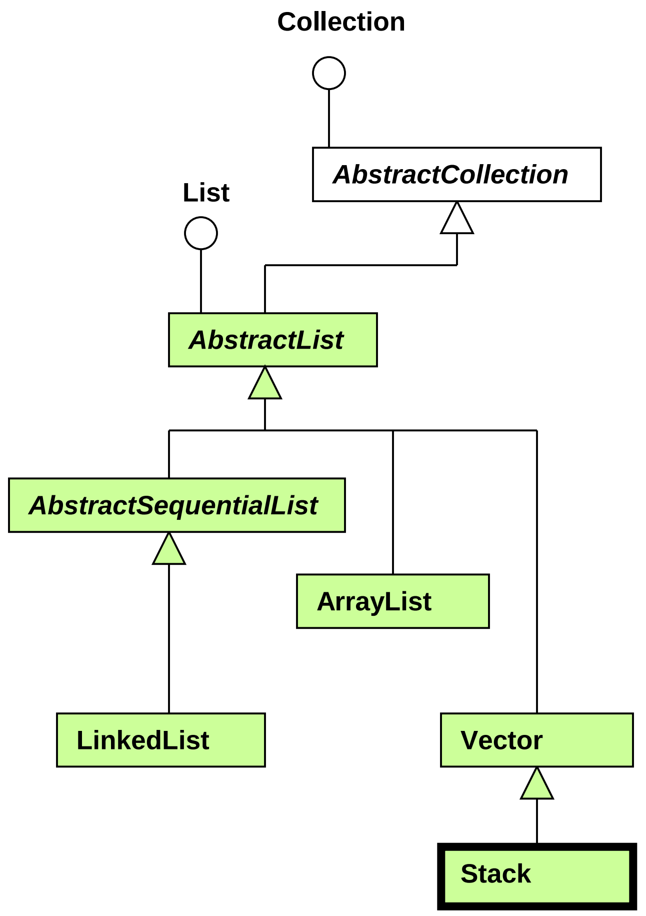
- Same as an the class ArrayList

- The main difference is that:
  - The methods of Vector are synchronized



Java Lists: Stack

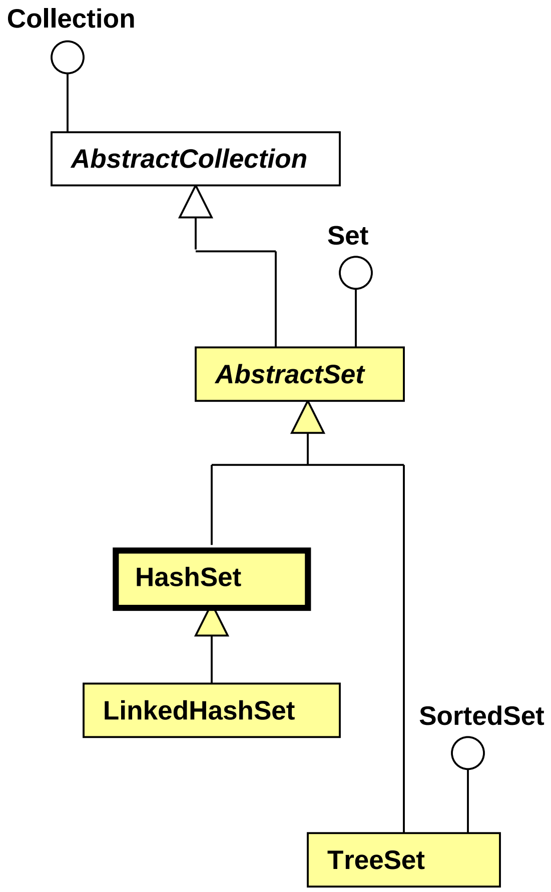
- The Stack class represents a last-in-first-out (LIFO) stack of objects
- The common push and pop operations are provided
- As well as a method to peek at the top item on the stack is also provided
- Note: It is considered bad design to make Stack a Subclass of vector
    - Vector operations should not be accessible in a Stack
    - It is designed this way because of a historical mistake



Seneca

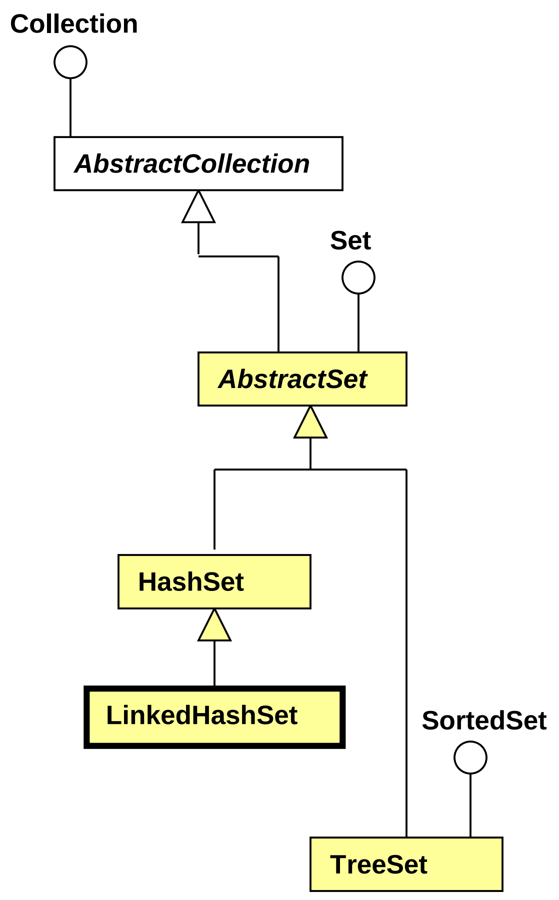
Java Sets: HashSet

- HashSet uses a hash table as the data structure to represent a set
- HashSet is a good choice for representing sets if order of the elements is not important
- HashSet methods <u>are not synchronized</u>



Java Sets: LinkedHashSet

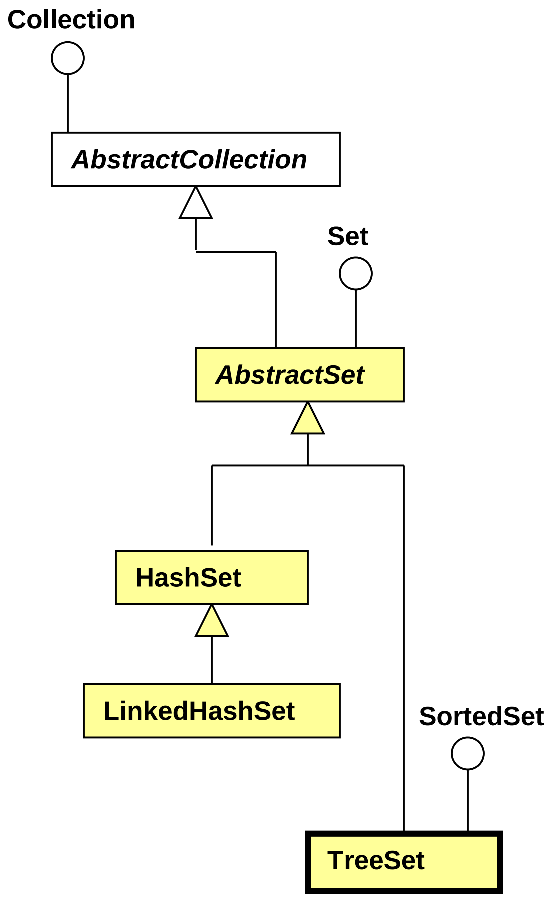
- Hash table and linked list implementation of the Set interface
- LinkedHashSet differs from HashSet in that the order is maintained
- Performance is below that of HashSet, due to the expense of maintaining the linked list
- Its methods are not synchronized



Seneca

Java Sets: TreeSet

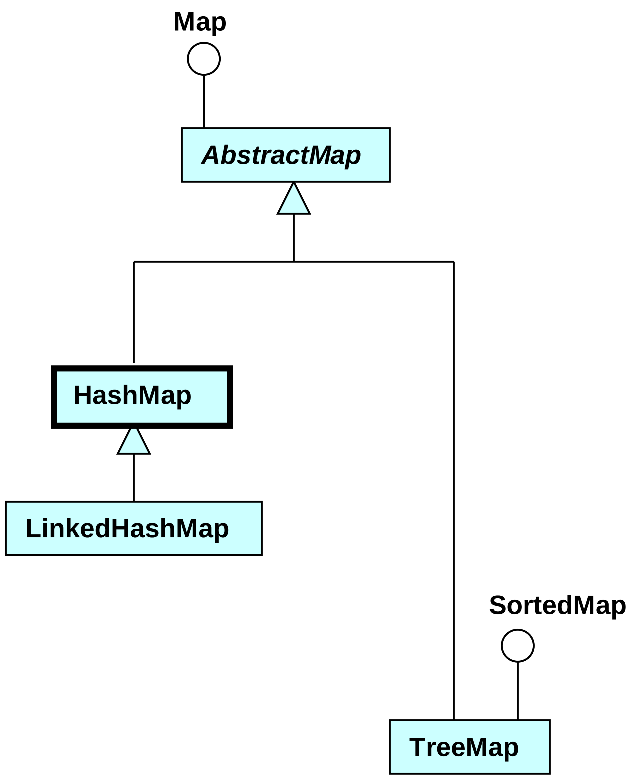
- Stores the elements in a balanced binary tree
  - A binary tree is a tree in which each node has at most two children
- TreeSet elements are sorted
- Less efficient than HashSet in insertion due to the use of a binary tree
- Its methods are not synchronized



Seneca

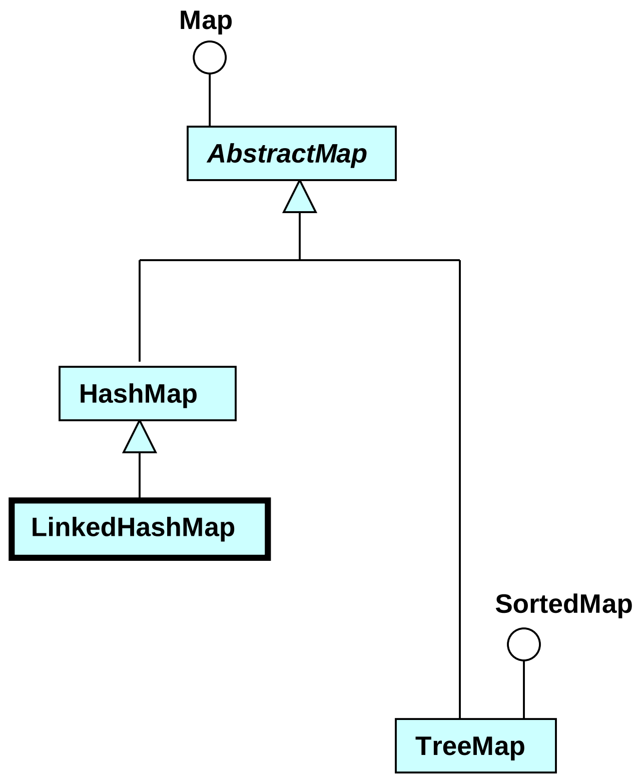
Java Maps: HashMap

- Stores the entries in a hash table
- Efficient in inserting (put()) and retrieving elements (get())
- The order is not maintained
- Unlike HashTable, HashMap's methods are not synchronized



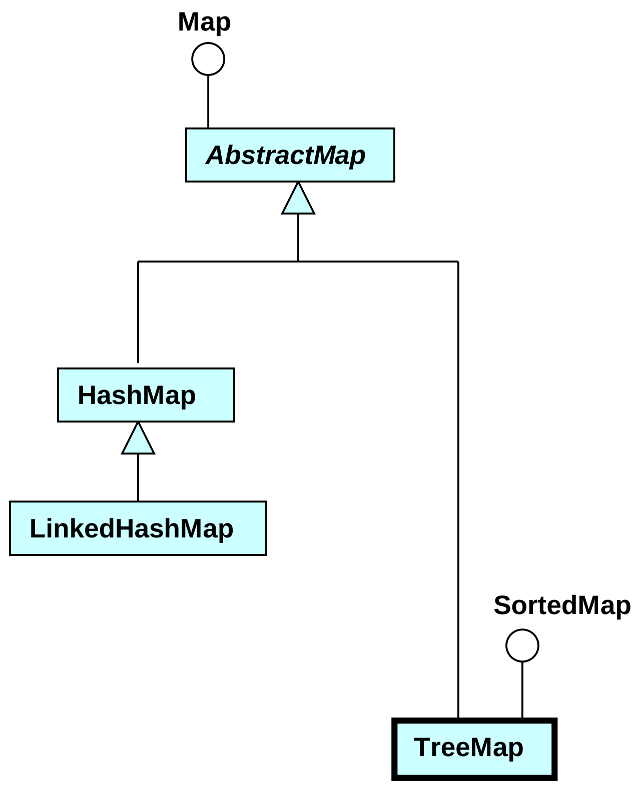
Java Maps: LinkedHashMap

- Hash table and linked list implementation of the Map interface
- LinkedHashMap differs from HashMap in that the order is maintained
- Performance is below that of HashMap, due to the expense of maintaining the linked list
- Its methods are not synchronized



Java Maps: TreeMap

- Red-Black tree based implementation of the SortedMap interface
- The keys are sorted according to their natural order
- Less efficient than HashMap for insertion and mapping
- Its methods are not synchronized



The Iterator Interface

- JCF provides a uniform way to iterate through the collection elements using the Iterator Interface

- The Iterator interface contains the following methods
  - boolean **hasNext**(): returns true if the iteration has more elements.
  - Object **next**(): returns the next element in the iteration.
  - void **remove**(): removes from the collection the last element returned by the iterator

- Iterator replaces Enumeration in the Java Collections Framework. Iterators differ from enumerations in two ways:
  - They allow the caller to remove elements from the collection during the iteration with well-defined semantics (a major drawback of Enumerations)
  - Method names have been improved

The class Arrays

- This class contains various static methods for manipulating arrays such as:
  - Sorting, comparing collections, binary searching…

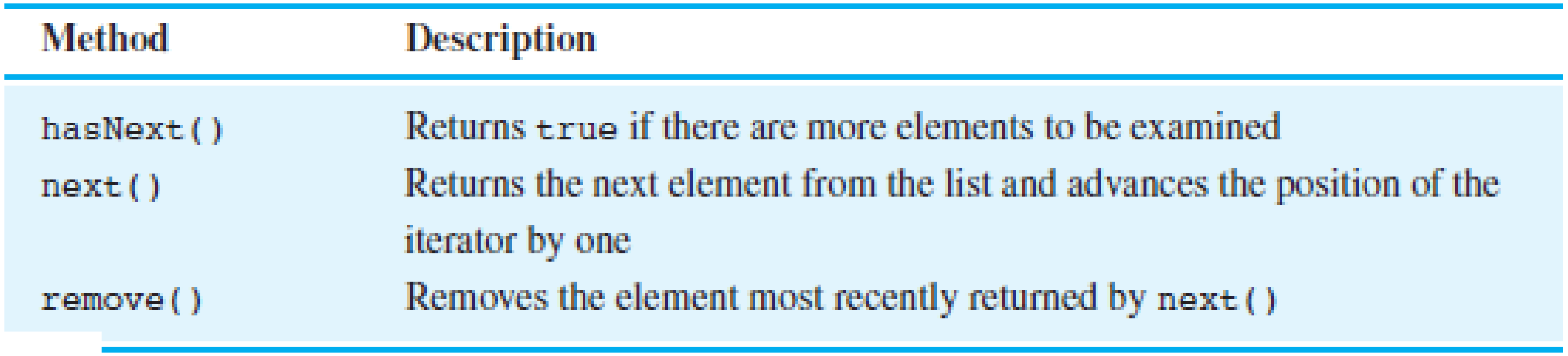
- Refer to: http://java.sun.com/j2se/1.4.2/docs/api/ for a complete list of utilities

Outcomes

- ✓ Java Collection Framework
- ✓ **Java Iterators**

Iterator Interface

- The Iterator interface is available from the Java 1.2 collection framework onwards.
- It traverses the collection of objects one by one.
- Popularly known as "Universal Java Cursor" as it works with all collections.
- This interface supports 'read' and 'remove' operations i.e. you can remove an element during an iteration using the iterator.



Method	Description
`hasNext()`	Returns `true` if there are more elements to be examined
`next()`	Returns the next element from the list and advances the position of the iterator by one
`remove()`	Removes the element most recently returned by `next()`

The next() method

Prototype: E next ()

Parameters: no parameters

Return type: E -> element

Description: Returns the next element in the collection.

If the iteration (collection) has no more elements, then it throws **NoSuchElementException**.

The hasNext() method

Prototype: boolean hasNext()

Parameters: NIL

Return type: true => there are elements in the collection.

False => no more elements

Description: The function hasNext() checks if there are more elements in the collection that is being accessed using an iterator. If there are no more elements, then you don't call the next() method. In other words, this function can be used to decide if the next() method is to be called.

Seneca

The remove() method

Prototype: void remove()

Parameters: NIL

Return type: NIL

Description: Removes the last element returned by the iterator iterating over the underlying collection. The remove () method can be called only once per next () call.

If the iterator does not support remove operation, then it throws **UnSupportedOperationException**. It throws **IllegalStateException** if the next method is not yet called.

Thank you!