Java Collections Framework

NATALIIA ROMANENKO

Agenda

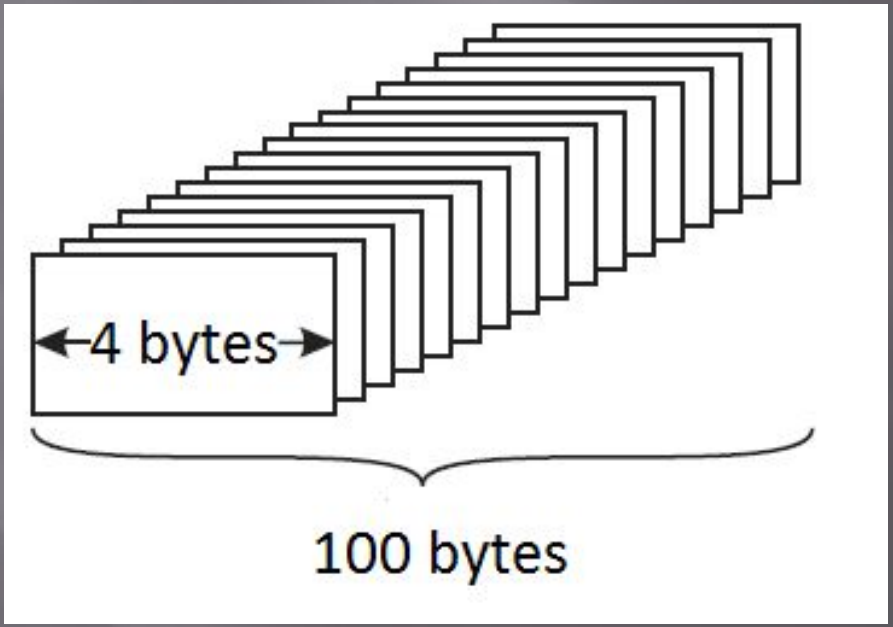
- Data Structure
- What is JCF?
- The Collection Interfaces
- Collections Implementations
- Ordering and Sorting
- The Legacy Collections Type
- The Collections Toolbox
- Other implementations in the API

Lecture Objectives

- To understand the concepts of Java collections Framework
- To be able to implement Java programs based on Collections

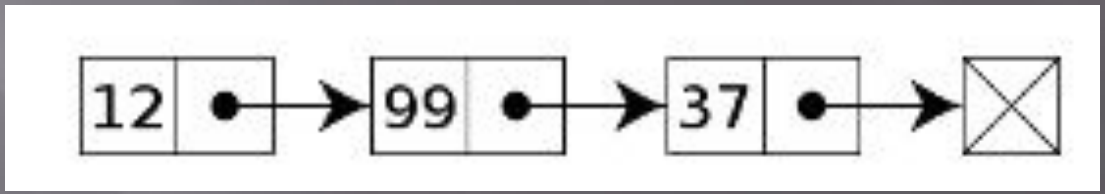
Store of Data Structure

Arrays - a linear data structure and it's mainly used to store similar data. An array is a particular method of storing **elements** of indexed data.



Store of Data Structure (cont.)

Linked list is a data structure consisting of a group of nodes. Each node is composed of a datum and a reference to the next node in the sequence. This structure allows for efficient insertion or removal of elements from any position in the sequence.

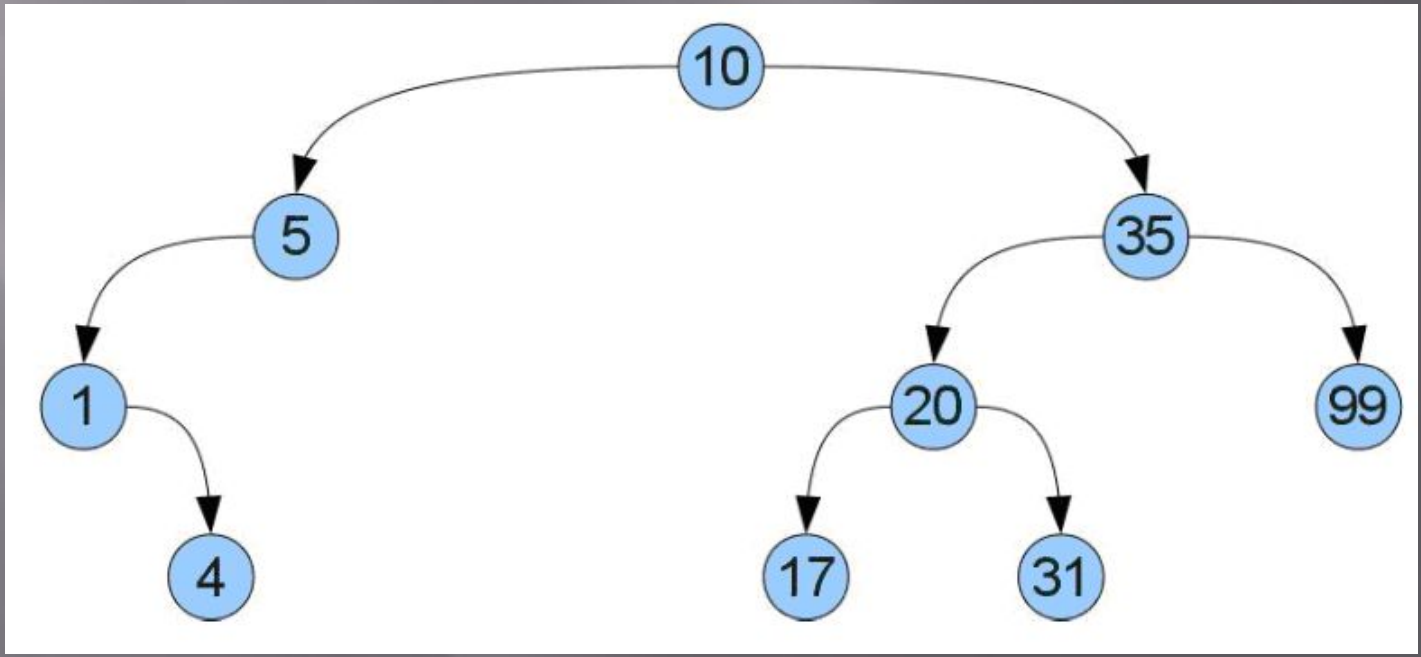


Store of Data Structure (cont.)

If each node has a reference to the next and previous nodes it's called **Doubly Linked List**.

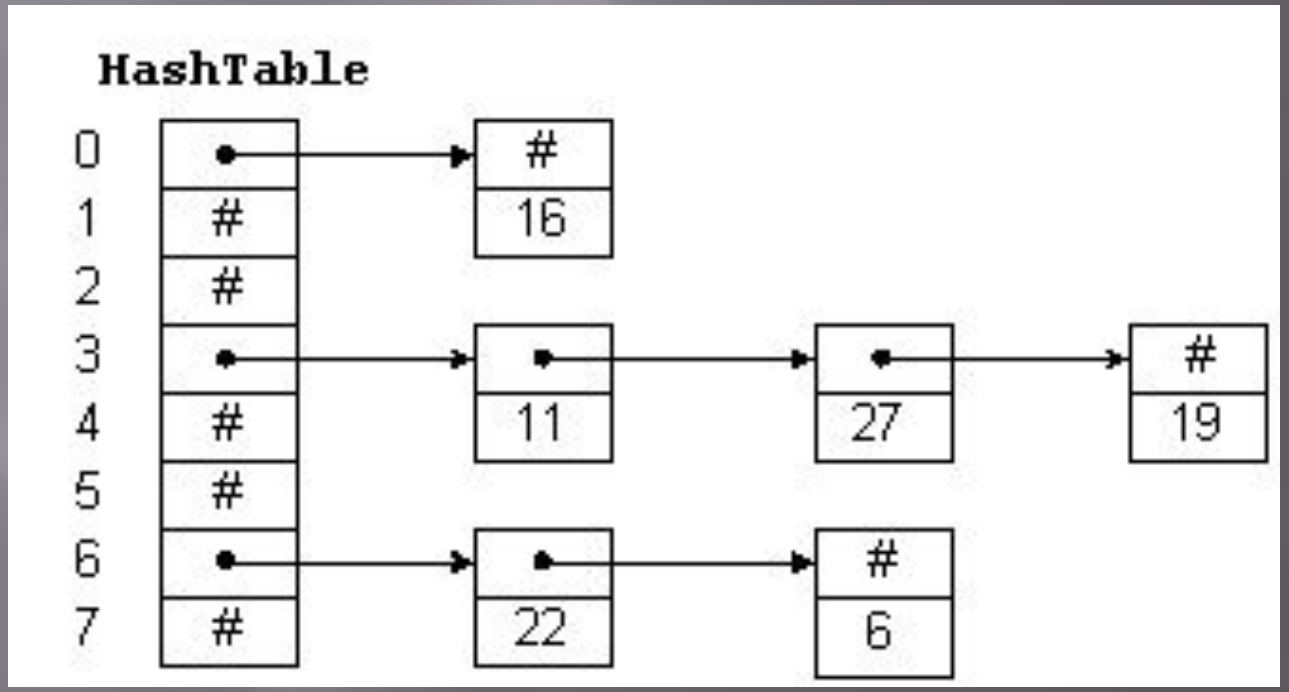
Store of Data Structure (cont.)

Binary tree is a tree data structure in which each node has at most two child nodes, usually distinguished as "left" and "right".



Store of Data Structure (cont.)

A hash table, or **a hash map**, is a data structure that associates *keys* with *values*.



The Limitation of Arrays

An array is a very useful type in Java but it has its restrictions:

- once an array is created it must be sized, and this size is fixed;
- it contains no useful pre-defined methods.

AND

SO

Java comes with a group of generic collection classes that grow as more elements are added to them, and these classes provide lots of useful methods.

- This group of collection classes are referred to as the **Java Collections Framework**.

What is a Collections Framework?

- A unified architecture for representing and manipulating collections.
- Includes:
  - **Interfaces**: A hierarchy of abstract data types.
  - **Implementations**
  - **Algorithms**: The methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces.

Hierarchy of interfaces

Exception Conventions

- UnsupportedOperationException
- ClassCastException
- IllegalArgumentException
- IllegalStateException
- NoSuchElementException
- NullPointerException
- IndexOutOfBoundsException

Iterators

Iterator is an object that enables a programmer to traverse a container, particularly lists.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

```
public void removeLongStrings (Collection<? extends CharSequence> coll, int maxLen) {
  Iterator<? extends CharSequence> it = coll.iterator();
  while (it.hasNext()) {
      CharSequence str = it.next();
      if (str.length() > maxLen) it.remove();
  }
}System.out.println(coll);
}
```

Collection<E>

```
public interface Collection<E> extends Iterable<E> {
  // Basic operations
  int size();
  boolean isEmpty();
  boolean contains(Object element);
  boolean add(E element);        //optional
  boolean remove(Object element); //optional
  Iterator<E> iterator();

  // Bulk operations
  boolean containsAll(Collection<?> c);
  boolean addAll(Collection<? extends E> c); //optional
  boolean removeAll(Collection<?> c);       //optional
  boolean retainAll(Collection<?> c);       //optional
  void clear();                             //optional

  // Array operations
  Object[] toArray();
  <T> T[] toArray(T[] a);
}
```

Set

- Set — a collection that cannot contain duplicate elements
- This interface models the mathematical set abstraction and is used to represent sets, such as the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine

Set<E> (methods)

```
public interface Set<E> extends Collection<E> {
   // Basic operations
   int size();
   boolean isEmpty();
   boolean contains(Object element);
   boolean add(E element);        //optional
   boolean remove(Object element); //optional
   Iterator<E> iterator();

   // Bulk operations
   boolean containsAll(Collection<?> c);
   boolean addAll(Collection<? extends E> c); //optional
   boolean removeAll(Collection<?> c);       //optional
   boolean retainAll(Collection<?> c);       //optional
   void clear();                             //optional

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

List

- List — an ordered collection (sometimes called a *sequence*)
- Lists can contain duplicate elements

List<E> (methods)

```
public interface List<E> extends Collection<E> {
   // Positional access
   E get(int index);
   E set(int index, E element);    //optional
   boolean add(E element);         //optional
   void add(int index, E element); //optional
   E remove(int index);            //optional
   boolean addAll(int index,
      Collection<? extends E> c); //optional

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

   // Iteration
   ListIterator<E> listIterator();
   ListIterator<E> listIterator(int index);

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

ListIterators

- A ListIterator extends Iterator to treat the collection as a list, allowing
  - access to the integer position (index) of elements
  - forward and backward traversal
  - modification and insertion of elements

ListIterator<E> (methods)

```
public interface ListIterator<E> extends Iterator<E> {
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void remove(); //optional
    void set(E e); //optional
    void add(E e); //optional
}
```

ListIterator

```
ListIterator<String> it =
    list.listIterator(list.size());
while (it.hasPrevious()) {
  String obj = it.previous();
    // … use obj  ….
}
```

Queue

- Queue — a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations
- Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner

Queue<E> (methods)

```
public interface Queue<E> extends Collection<E> {
    E element();        //throws
    E peek();           //null
     boolean add(E e);      //throws
     boolean offer(E e);    //null
    E remove();         //throws
    E poll();           //null
}
```

Queue

- In addition to the inherited core services offered by Collection, queue offers following methods in two flavors:

	Purpose	Throw Exception	Return Special Value
Insert	Inserts an elements to the queue	add(obj)	offer(obj)
Remove	Remove head of the queue and return it.	remove()	poll()
Examine	Return the head of the queue	element()	peek()

Deque

- A linear collection that supports element insertion and removal at both ends
- When a Deque is used as a queue, FIFO (First-In-First-Out) behavior results
- Deques can also be used as LIFO (Last-In-First-Out) stacks

Deque<E> (methods)

public interface Deque<E> extends Queue<E> {

element()

add(E e) (addLast(E e))

offer(E e) (offerLast(E e)), offerFirst(E e)

peek()* (peekFirst()), peekLast()

getFirst(), getLast()

remove() (removeFirst()), removeFirstOccurrence(Object o),
removeLast(), removeLastOccurrence(Object o)

poll() (pollFirst()), pollLast()

contains(Object o)

iterator()

descendingIterator()

size()

pop() (removeFirst())

push(E e) (addFirst(E e))

}

from Queue, from Stack

SortedSet

- SortedSet — a Set that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls

SortedSet<E> (methods)

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

    // Endpoints
    E first();
    E last();

    // Comparator access
    Comparator<? super E> comparator();
}
```

Map

- An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value
- The Map interface provides three *collection views*, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings

Map<K,V> (methods)

```
public interface Map<K,V> {

    // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk operations
    void putAll(Map<? extends K, ? extends V> m);
```

Map<K,V> (methods)

```
    void clear();

    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();

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

Map.Entry

```
public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
Map<String, String> map = new HashMap<String, String>();
map.put("1", "a");
map.put("2", "b");
map.put("3", "c");
for( Entry<String, String> entry : map.entrySet() ) {
        if( "2".equals( entry.getKey() ) )
            entry.setValue( "x" );
}
```

SortedMap

- SortedMap — a Map that maintains its mappings in ascending key order. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories

SortedMap<K,V> (methods)

```
public interface SortedMap<K, V> extends Map<K, V>{

    SortedMap<K, V> subMap(K fromKey, K toKey);
    SortedMap<K, V> headMap(K toKey);
    SortedMap<K, V> tailMap(K fromKey);
    K firstKey();
    K lastKey();

    Comparator<? super K> comparator();
}
```

Implementations

JDK provides implementations of each interface.

- All implementations permit null elements, keys and values
- All are Serializable, and all support a public clone method
- Each one is unsynchronized
- If you need a synchronized collection, the **synchronization wrappers** allow any collection to be transformed into a synchronized collection

HashSet, TreeSet, LinkedHashSet

- The two general purpose Set implementations are HashSet and TreeSet (and LinkedHashSet which is between them)
- HashSet is much faster but offers no ordering guarantees.
- If in-order iteration is important use TreeSet.
- Iteration in HashSet is linear in the sum of the number of entries and the capacity. It's important to choose an appropriate initial capacity if iteration performance is important. The default initial capacity is 101. The initial capacity may be specified using the int constructor. To allocate a HashSet whose initial capacity is 17:
  - *Set s= new HashSet(17);*

Set Implementation Comparisons

	HashSet	TreeSet	LinkedHashSet
Storage Type	Hash Table	Red-Black Tree	Hash Table with a Linked List
Performance	Best performance	Slower than HashSet	Little costly than HashSet
Order of Iteration	No guarantee of order of iteration	Order based	Orders elements based on insertion

ArrayList and LinkedList

- The two general purpose List implementations are ArrayList and LinkedList . **ArrayList** offers constant time positional access, and it's just plain fast, because it does not have to allocate a node object for each element in the List, and it can take advantage of the native method System.arraycopy when it has to move multiple elements at once
- If you frequently add elements to the beginning of the List, or iterate over the List deleting elements from its interior, you might want to consider **LinkedList**. These operations are constant time in a LinkedList but linear time in an ArrayList. Positional access is linear time in a LinkedList and constant time in an ArrayList

HashMap, TreeMap, LinkedHashMap

- The two general purpose Map implementations are HashMap and TreeMap.
- And LinkedHashMap (similar to LinkedHashSet)
- The situation for Map is exactly analogous to Set
- If you need SortedMap operations you should use TreeMap; otherwise, use HashMap

The Legacy Collection Types

- **Enumeration**
  - Analogous to Iterator.
- **Vector**
  - Analogous to ArrayList, maintains an ordered list of elements that are stored in an underlying array.
- **Stack**
  - Analogous of Vector that adds methods to push and pop elements.
- **Dictionary**
  - Analogous to the Map interface, although Dictionary is an abstract class, not an interface.
- **Hashtable**
  - Analogous HashMap.
- **Properties**
  - A subclass of Hashtable. Maintains a map of key/value pairs where the keys and values are strings. If a key is not found in a properties object a “default” properties object can be searched.

General-purpose Implementations					
Interfaces	**Implementations**				
	Hash table	**Resizable array**	**BalancedTree (sorted)**	**Linked list**	**Hash table + Linked list**
Set	HashSet		TreeSet (sorted)		LinkedHashSet
List		ArrayList		LinkedList	
Queue			PriorityQueue		
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap (sorted)		LinkedHashMap

Note the naming convention

LinkedList also implements queue and there is a PriorityQueue implementation (implemented with heap)

Implementations

- Each of the implementations offers the strengths and weaknesses of the underlying data structure.
- What does that mean for:
  - Hashtable
  - Resizable array
  - Tree
  - LinkedList
  - Hashtable plus LinkedList
- **Think about these tradeoffs when selecting the implementation!**

Ordering and Sorting

There are two ways to define orders on objects.

- Each class can define a *natural order* among its instances by implementing the Comparable interface.

- Arbitrary orders among different objects can be defined by *comparators*, classes that implement the Comparator interface.

The Comparable Interface

- The Comparable interface consists of a single method:
- *public interface Comparable<T> {*
- *public int compareTo(T o);*
- *}*
- The **compareTo** method compares the receiving object with the specified object, and returns a negative integer, zero, or a positive integer as the receiving object is less than, equal to, or greater than the specified Object.

Comparator

- Comparator is another interface (in addition to Comparable) provided by the Java API which can be used to order objects.
- You can use this interface to define an order that is different from the Comparable (natural) order.

Comparator

- A Comparator is an object that encapsulates an ordering. Like the Comparable interface, the Comparator interface consists of a single method:
- public interface Comparator<T> {
- int compare(T o1, T o2);
- }
- The compare method compares its two arguments, returning a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

Live Code 😊. Comparable

```
class HDTV implements Comparable<HDTV> {
        private int size;
        private String brand;
         public HDTV(int size, String brand) {
                this.size = size;
                this.brand = brand;
        }
        public int getSize() { return size;}
        public void setSize(int size) {this.size = size; }
        public String getBrand() { return brand;}
        public void setBrand(String brand) {this.brand = brand; }

        @Override
        public int compareTo(HDTV tv) {
                if (this.getSize() > tv.getSize()) return 1;
                else if (this.getSize() < tv.getSize()) return -1;
                else return 0;
        }
}
```

Live Code 😊. Comparable

```
public class Main {
	public static void main(String[] args) {
		HDTV tv1 = new HDTV(55, "Samsung");
		HDTV tv2 = new HDTV(60, "Sony");
		HDTV tv3 = new HDTV(42, "Panasonic");
		ArrayList<HDTV> al = new ArrayList<HDTV>();
		al.add(tv1);
		al.add(tv2);
		al.add(tv3);
		Collections.sort(al);
		for (HDTV a : al) {
			System.out.println(a.getBrand());
		}
	}
}
```

Live Code ☺️. Comparator

```
class HDTV {
        private int size;
        private String brand;
         public HDTV(int size, String brand) {
                this.size = size;
                this.brand = brand;
        }
        public int getSize() { return size;}
        public void setSize(int size) {this.size = size; }
        public String getBrand() { return brand;}
        public void setBrand(String brand) {this.brand = brand; }
}
```

Live Code 😊. Comparator

```
class SizeComparator implements Comparator<HDTV> {

@Override
	public int compare(HDTV tv1, HDTV tv2) {
		int tv1Size = tv1.getSize();
		int tv2Size = tv2.getSize();
		if (tv1Size > tv2Size) {
			return 1;
		} else if (tv1Size < tv2Size) {
			return -1;
		} else {
			return 0;
		}
	}
}
class BrandComparatorDescOrder implements Comparator<HDTV> {

	@Override
	public int compare(HDTV tv1, HDTV tv2) {
	return tv2.getBrand().compareTo(tv1.getBrand())
	}
}
```

Live Code 😊. Comparator

```
public class Main {
	public static void main(String[] args) {
		HDTV tv1 = new HDTV(55, "Samsung");
		HDTV tv2 = new HDTV(60, "Sony");
		HDTV tv3 = new HDTV(42, "Panasonic");

		ArrayList<HDTV> al = new ArrayList<HDTV>();
		al.add(tv1);
		al.add(tv2);
		al.add(tv3);

		Collections.sort(al, new SizeComparator());
		for (HDTV a : al) {
			System.out.println(a.getBrand());
		}

		Collections.sort(al, new BrandComparatorDescOrder());
		for (HDTV a : al) {
			System.out.println(a.getBrand());
		}
	}
}
```

Other implementations in the API

- Wrapper implementations delegate all their real work to a specified collection but add (or remove) extra functionality on top of what the collection offers.
  - Synchronization Wrappers
  - Unmodifiable Wrappers
- Convenience implementations are mini-implementations that can be more convenient and more efficient than general-purpose implementations when you don't need their full power
  - List View of an Array
  - Immutable Multiple-Copy List
  - Immutable Singleton Set
  - Empty Set, List, and Map Constants

Synchronization wrappers

The **synchronization wrappers** add automatic synchronization (thread-safety) to an arbitrary collection. There is one static factory method for each of the six core collection interfaces:

- *public static Collection synchronizedCollection(Collection c);*
- *public static Set synchronizedSet(Set s);*
- *public static List synchronizedList(List list);*
- *public static Map synchronizedMap(Map m);*
- *public static SortedSet synchronizedSortedSet(SortedSet s);*
- *public static SortedMap synchronizedSortedMap(SortedMap m);*

Each of these methods returns a synchronized (thread-safe) Collection backed by the specified collection.

Unmodifiable wrappers

- **Unmodifiable wrappers** take away the ability to modify the collection, by intercepting all of the operations that would modify the collection, and throwing an **UnsupportedOperationException.** The unmodifiable wrappers have two main uses:
- To make a collection immutable once it has been built.
- To allow "second-class citizens" read-only access to your data structures. You keep a reference to the backing collection, but hand out a reference to the wrapper. In this way, the second-class citizens can look but not touch, while you maintain full access.

Unmodifiable wrappers(cont.)

There is one static factory method for each of the six core collection interfaces:

- *public static Collection unmodifiableCollection(Collection c);*
- *public static Set unmodifiableSet(Set s);*
- *public static List unmodifiableList(List list);*
- *public static Map unmodifiableMap(Map m);*
- *public static SortedSet unmodifiableSortedSet(SortedSet s);*
- *public static SortedMap unmodifiableSortedMap(SortedMap m);*

Singleton

- static <T> Set<T>Collections.singleton(T e) returns an immutable set containing only the element e
  - This is handy when you have a single element but you would like to use a Set operation
- c.removeAll(Collections.singleton(e));

will remove all occurrences of e from the Collection c

The Collections Toolbox

The collections framework also provides polymorphic versions of algorithms you can run on collections.

- Sorting
- Shuffling
- Routine Data Manipulation
  - Reverse
  - Fill copy
  - etc.
- Searching
  - Binary Search
- Composition
  - Frequency
  - Disjoint
- Finding extreme values
  - Min
  - Max

Concurrent Collections

- ConcurrentReaderHashMap An analog of java.util.Hashtable that allows retrievals during updates.
- ConcurrentHashMap An analog of java.util.Hashtable that allows both concurrent retrievals and concurrent updates.
- CopyOnWriteArrayList A copy-on-write analog of java.util.ArrayList
- CopyOnWriteArraySet A java.util.Set based on CopyOnWriteArrayList.
- SyncCollection A wrapper class placing either Syncs or ReadWriteLocks around java.util.Collection
- SyncSet A wrapper around java.util.Set
- SyncSortedSet A wrapper around java.util.SortedSet
- SyncList A wrapper around java.util.List
- SyncMap A wrapper around java.util.Map
- SyncSortedMap A wrapper around java.util.SortedMap

How to choose which Java collection class to use?

Which Java List to use?

Class	Features/Implementation	When to use
ArrayList	Allows elements to be efficiently read by index. Adding/removing the last element is efficient. Not synchronized in any way.	In most cases.
LinkedList	First and last elements can be accessed efficiently; Other elements cannot be efficiently accessed by index; Not synchronized in any way.	Effectively, functions as a non-synchronized queue. In practice, rarely used: when you need a queue, you often need it to be concurrent or to provide other functionality; other implementations are often more useful.
CopyOnWriterArrayList	Allows safe concurrent access; Reads are efficient and non-blocking; Modifications are not efficient (since a brand new copy of the list is taken each time).	Where you need concurrent access and where frequency of reads far outweights frequency of modifications.

How to choose which Java collection class to use?

Which Java Set to use?

Ordering of keys	Non-concurrent	Concurrent
No particular order	HashSet	—
Sorted	TreeSet	ConcurrentSkipListMap
Fixed	LinkedHashSet	CopyOnWriteArraySet

How to choose which Java collection class to use?

Which Java Map to use?

Ordering of keys	Non-concurrent	Concurrent
No particular order	HashMap	ConcurrentHashMap
Sorted	TreeMap	ConcurrentSkipListMap
Fixed	LinkedHashMap	—

Open Source Collections Libraries in Java

- Apache Commons Collections Package
- Guava-libraries (Google Collections Library)
- Trove high performance collections for Java
- The Mango Library

Questions