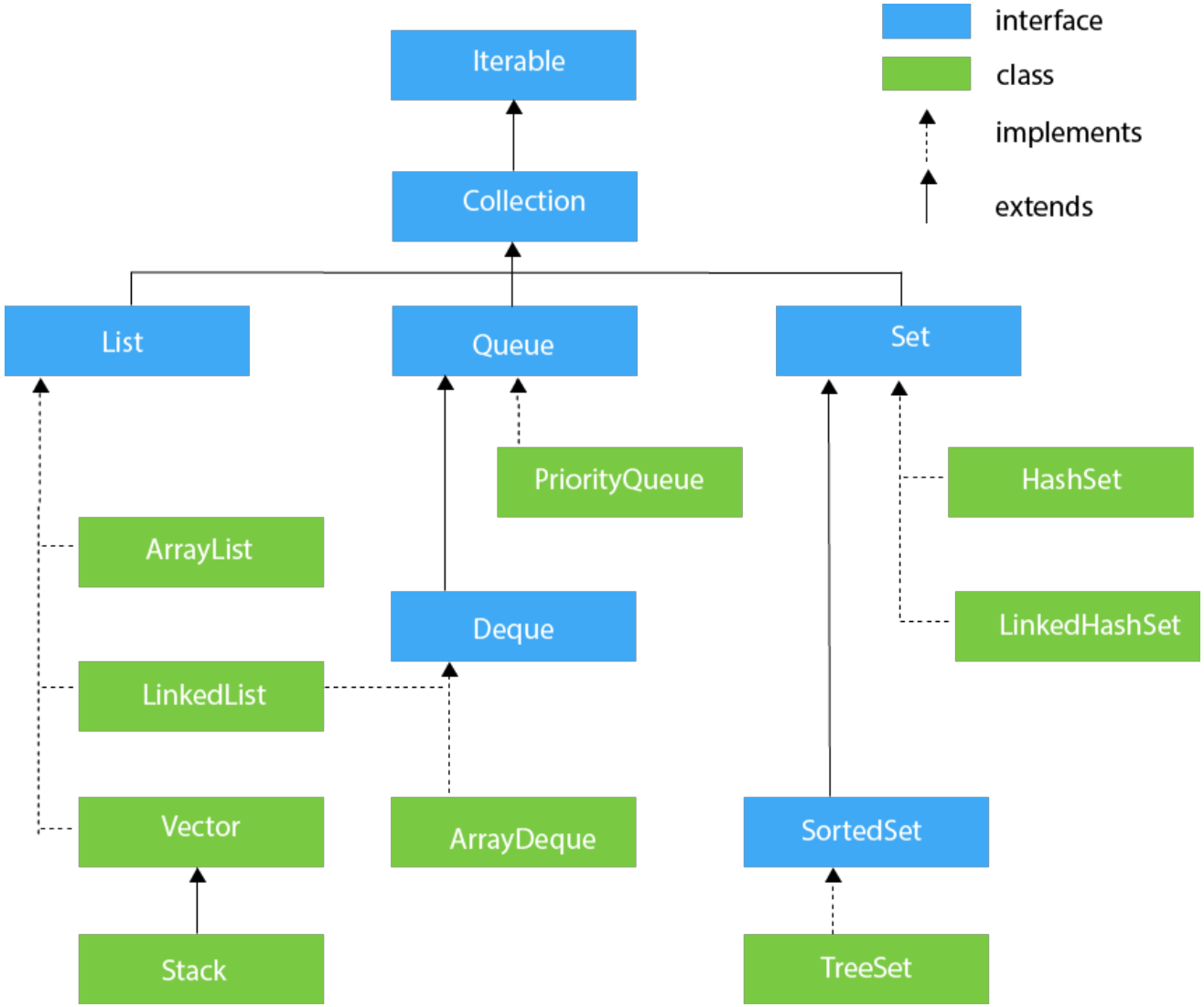
Object Oriented Programming with Java

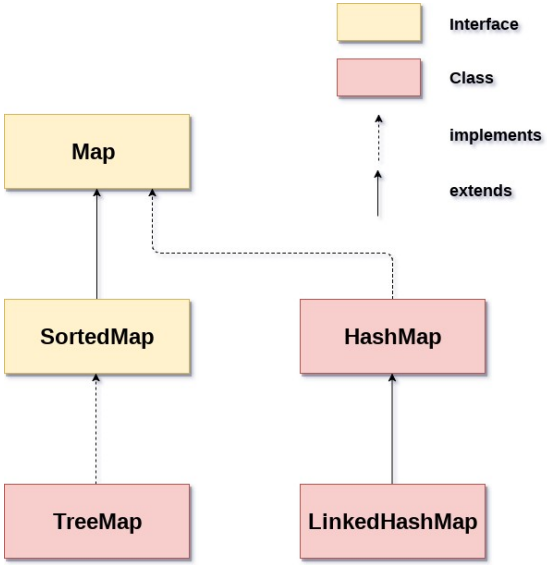
Java Collections

Outline

- Useful JCF classes and methods
- Iterators
- Comparators

Java Collections Class Hierarchy





Source: javatpoint.com

List interface (there are more methods!)

`boolean`	`add(E e)`	Appends the specified element to the end of this list (optional operation).
`void`	`add(int index, E element)`	Inserts the specified element at the specified position in this list (optional operation).
`boolean`	`addAll(Collection<? extends E> c)`	Appends all of the elements in the specified collection to the end of this list
`boolean`	`addAll(int index, Collection<? extends E> c)`	Inserts all of the elements in the specified collection into this list at the specified position (optional operation).
`boolean`	`contains(Object o)`	Returns `true` if this list contains the specified element.
`E`	`get(int index)`	Returns the element at the specified position in this list.
`int`	`indexOf(Object o)`	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
`boolean`	`isEmpty()`	Returns `true` if this list contains no elements.
`int`	`lastIndexOf(Object o)`	Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
`E`	`remove(int index)`	Removes the element at the specified position in this list (optional operation).
`boolean`	`remove(Object o)`	Removes the first occurrence of the specified element from this list, if it is present (optional operation).
`boolean`	`removeAll(Collection<?> c)`	Removes from this list all of its elements that are contained in the specified collection (optional operation).
`int`	`size()`	Returns the number of elements in this list.
`<T> T[]`	`toArray(T[] a)`	Returns an array containing all of the elements in this list in proper sequence; the runtime type of the returned array is that of the specified array.

ArrayList basics

```java
import java.util.ArrayList;

// non-parametric constructor
List<String> list = new ArrayList<>();

// constructor with initial capacity
List<Double> list = new ArrayList<>(20);

// constructor from collection
List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3));

// iteration with foreach loop
for(Integer in : list)
    System.out.println(in);

// iteration with iterator
ListIterator<Integer> it = list.listIterator(list.size());
while(it.hasNext())
    System.out.println(it.next());
```

List *vs* ArrayList *vs* LinkedList

- `List` is an interface; use it for declaration but not for instantiation
- `ArrayList` and `LinkedList` are classes; use it for both declaration and instantiation

```
public static void main (String args[])
{
    List<Integer> l1 = new ArrayList<>();
    List<Integer> l2 = new LinkedList<>();
    ArrayList<Integer> l3 = new ArrayList<>();
    LinkedList<Integer> l4 = new LinkedList<>();

    l1.add(3); l1.add(7); l1.add(-2);
    l2.addAll(l1); l3.addAll(l1); l4.addAll(l1);
    l4.addFirst(13);
    l2.addLast(34);  // ERROR
    l1.remove((Integer)(-2));

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

Just because the execution output looks the same, doesn't mean that the performance is the same too. Selecting the proper collection is critical for efficiency!

ArrayList *vs* LinkedList

	ArrayList	LinkedList
Internal Implementation	Dynamic array	Doubly linked list
Memory allocation	Slow (expand, shrink, shift)	Fast (as needed only)
implements interface	List	List, Queue
Data access	Fast (random access)	Slow (serial search)

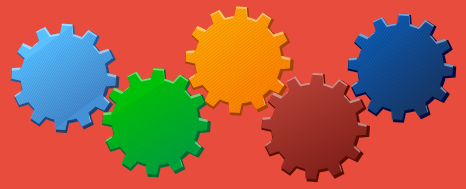
ArrayList *vs* Vector

	ArrayList	Vector
Multithreading	Non synchronized	Synchronized
Performance	Fast	Slow (all threads wait for the lock to be released)
Expansion (not a Java specification, just the trend)	Increase by 50%	Increase by 100%

Iterator for a user-defined collection

```
import java.util.Iterator;

class MyCollection<T> implements Iterable<T> {
    Iterator<T> iterator() {
        return new MyCollectionIterator<T>(this);
    }
}

class MyCollectionIterator<T> implements Iterator<T> {
    MyCollectionIterator(MyCollection ref) {
    }

    boolean hasNext() {
    }

    T next() {
    }

    void remove() {
    }
}
```

Practice Problems



Write a method that takes an ArrayList of Integers and an int and runs a foreach loop searching for that number. If it finds the number in the list, it removes it. The number can appear multiple times in the list.

```
void removeElement(List<Integer> xs, int num)
```

Removing items while iterating a collection

Removing items while iterating a collection will throw a `ConcurrentModificationException` because the for statement doesn't *communicate* with the remove call

```
List<Integer> a = new ArrayList<>(Arrays.asList(0, 40, 11,-6));

for (int item : a)
{
    if (item==11)
        a.remove(new Integer(11));    // throws exception
  System.out.println(item);
}
```

Removing items while iterating a collection

To avoid the exception use an iterator to loop through the collection and remove the iterator instead of the collection's item.

```
List<Integer> a = new ArrayList<>(Arrays.asList(0, 40, 11,-6));

Iterator iter = a.iterator();

while(iter.hasNext())
{
  Integer item = (Integer)iter.next();

  if (item==11)
      iter.remove();
  System.out.println(item);
}
```

Comparable *vs* Comparator interfaces

Comparable

- The `Comparable` interface provides only one method named **`compareTo(Object other)`**
- Comparison in this case has a predetermined logic (e.g. comparison based on student's GPA)
- We must edit a class if we want to implement the Comparable interface

Comparator

- The `Comparator` interface provides only one method named **`compare(Object one, Object two)`**
- We can define multiple comparators, each one implementing a comparison based on different criteria
- We do not need to edit a class itself because comparators are independent classes
- Comparators are the only option when we want to provide an alternative comparison for a built-in type. For example, if we want to compare Strings based on their length only, we can't use Comparable.

Comparable *vs* Comparator interfaces

```java
class Student implements Comparable<Student> {
    private double gpa;
    private String name;

    public int compareTo(Student other) {
        return this.gpa - other.gpa;
    }
}
```

```java
class StudentComparatorName implements Comparator<Student> {
    public int compare(Student s1, Student s2) {
        return s1.getName().compareTo(s2.getName());
    }
}
```

```java
class StudentComparatorGPA implements Comparator<Student> {
    public int compare(Student s1, Student s2) {
        if(s1.getGPA()-s2.getGPA() < 0.2) return 0;
          else return s1.getGPA()-s2.getGPA();
    }
}
```

```java
MyCollection<Student> xs = new MyCollection<>();
Collections.sort(xs); // uses comparable

StudentComparatorName com = new StudentComparatorName();
Collections.sort(xs, com); // uses comparator
```

Collections class (there are more methods!)

not to be confused with the **`Collection`** interface!

`static <T> boolean`	addAll(Collection<? super T> c, T... elements) Adds all of the specified elements to the specified collection.
`static <T> int`	binarySearch(List<? extends Comparable<? super T>> list, T key) Searches the specified list for the specified object using the binary search algorithm.
`static <T> int`	binarySearch(List<? extends T> list, T key, Comparator<? super T> c) Searches the specified list for the specified object using the binary search algorithm.
`static <T> void`	copy(List<? super T> dest, List<? extends T> src) Copies all of the elements from one list into another.
`static boolean`	disjoint(Collection<?> c1, Collection<?> c2) Returns `true` if the two specified collections have no elements in common.
`static <T> Enumeration<T>`	enumeration(Collection<T> c) Returns an enumeration over the specified collection.
`static <T> void`	fill(List<? super T> list, T obj) Replaces all of the elements of the specified list with the specified element.
`static int`	frequency(Collection<?> c, Object o) Returns the number of elements in the specified collection equal to the specified object.
`static int`	indexOfSubList(List<?> source, List<?> target) Returns the starting position of the first occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence.
`static int`	lastIndexOfSubList(List<?> source, List<?> target) Returns the starting position of the last occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence.
`static <T> ArrayList<T>`	list(Enumeration<T> e) Returns an array list containing the elements returned by the specified enumeration in the order they are returned by the enumeration.

Collections class (there are more methods!)

not to be confused with the **`Collection`** interface!

`static <T extends Object & Comparable<? super T>> T`	max(Collection<? extends T> coll) Returns the maximum element of the given collection, according to the *natural ordering* of its elements.
`static <T> T`	max(Collection<? extends T> coll, Comparator<? super T> comp) Returns the maximum element of the given collection, according to the order induced by the specified comparator.
`static <T extends Object & Comparable<? super T>> T`	min(Collection<? extends T> coll) Returns the minimum element of the given collection, according to the *natural ordering* of its elements.
`static <T> T`	min(Collection<? extends T> coll, Comparator<? super T> comp) Returns the minimum element of the given collection, according to the order induced by the specified comparator.
`static void`	reverse(List<?> list) Reverses the order of the elements in the specified list.
`static void`	shuffle(List<?> list) Randomly permutes the specified list using a default source of randomness.
`static void`	shuffle(List<?> list, Random rnd) Randomly permute the specified list using the specified source of randomness.
`static <T extends Comparable<? super T>> void`	sort(List<T> list) Sorts the specified list into ascending order, according to the natural ordering of its elements.
`static <T> void`	sort(List<T> list, Comparator<? super T> c) Sorts the specified list according to the order induced by the specified comparator.
`static void`	swap(List<?> list, int i, int j) Swaps the elements at the specified positions in the specified list.