



# Generics Collection Framework

Sisoft Technologies Pvt Ltd  
SRC E7, Shipra Riviera Bazar, Gyan Khand-3, Indirapuram, Ghaziabad  
Website: [www.sisoft.in](http://www.sisoft.in) Email: [info@sisoft.in](mailto:info@sisoft.in)  
Phone: +91-9999-283-283

# Generics



- At its core, the term *generics means parameterized types*.
- *Parameterized types are* important because they enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter.

# Generic Type

- A *generic type* is a generic class or interface that is parameterized over types
- A *generic class* is defined with the following format:

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

- The type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the *type parameters* (also called *type variables*) T1, T2, ..., and Tn.

# Generic Type: Class Declaration



```
public class Box<T> // T stands for "Type"  
{  
    private T t;  
    public void set(T t) { this.t = t; }  
    public T get() { return t; }  
}
```

# Generic Type: Invoking and Instantiating



- To instantiate this class, use the new keyword, as usual, but place `<Integer>` between the class name and the parenthesis

```
Box<Integer> integerBox = new Box<Integer>();
```

- In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (`<>`) as long as the compiler can determine, or infer, the type arguments from the context. This pair of angle brackets, `<>`, is informally called *the diamond*

# Bounded Type Parameter



- To restrict the types that can be used as type arguments in a parameterized type
- Upper bound: Declares the superclass from which all type arguments must be derived.  
*<T extends superclass>*

# Wild Card Parameter



- In generic code, the question mark (?), called the *wildcard*, represents an unknown type
- The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable; sometimes as a return type (though it is better programming practice to be more specific).
- The wildcard is never used as a type argument for a generic method invocation, a generic class instance creation, or a supertype.

# Type Erasure



- To implement generics, the Java compiler applies type erasure to:
  - Replace all type parameters in generic types with their bounds or Object if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.
  - Insert type casts if necessary to preserve type safety.
  - Generate bridge methods to preserve polymorphism in extended generic types.



# Generic - Example



- ◆ The following code snippet displays the use of non-generics collections:

```
ArrayList list = new ArrayList();  
list.add(0, new Integer(42));  
int total =  
((Integer)list.get(0)).intValue();
```

- ◆ The following code snippet displays the use of generics collections:

```
ArrayList<Integer> list = new  
ArrayList<Integer>();  
list.add(0, new Integer(42));  
int total =  
list.get(0).intValue();
```

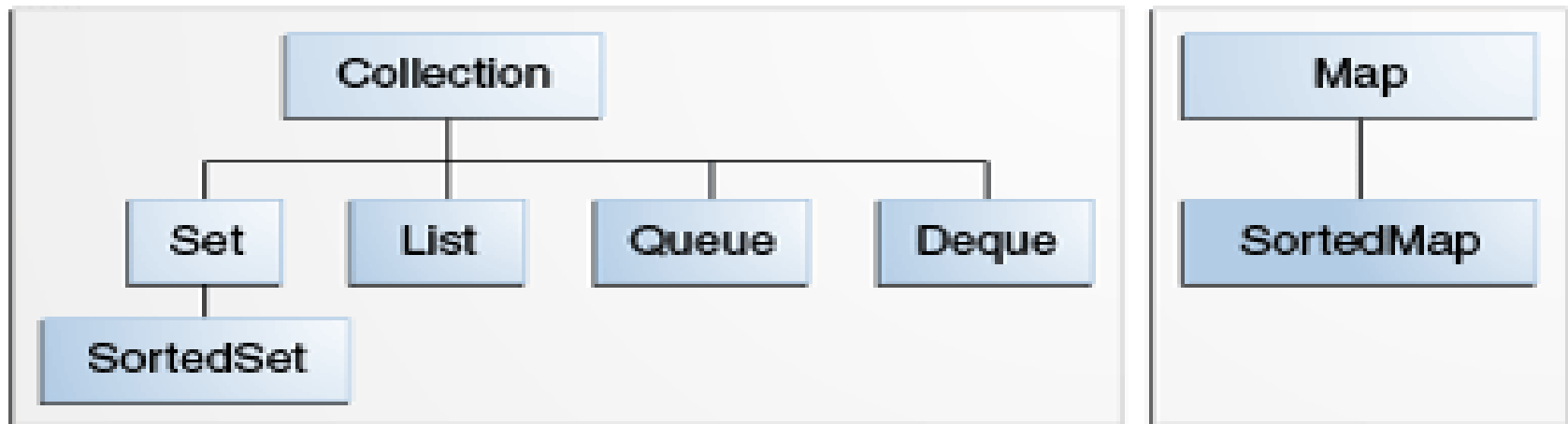
# Collections: What is it?

“The Collections Framework provides a well-designed set of interfaces and classes for storing and manipulating groups of data as a single unit, a collection.” -java.sun.com

# So what?

Well, The framework provides a convenient API to many of the abstract data types familiar to computer science: maps, sets, lists, trees, arrays, hashtables and other collections.

# Core Collection Interfaces

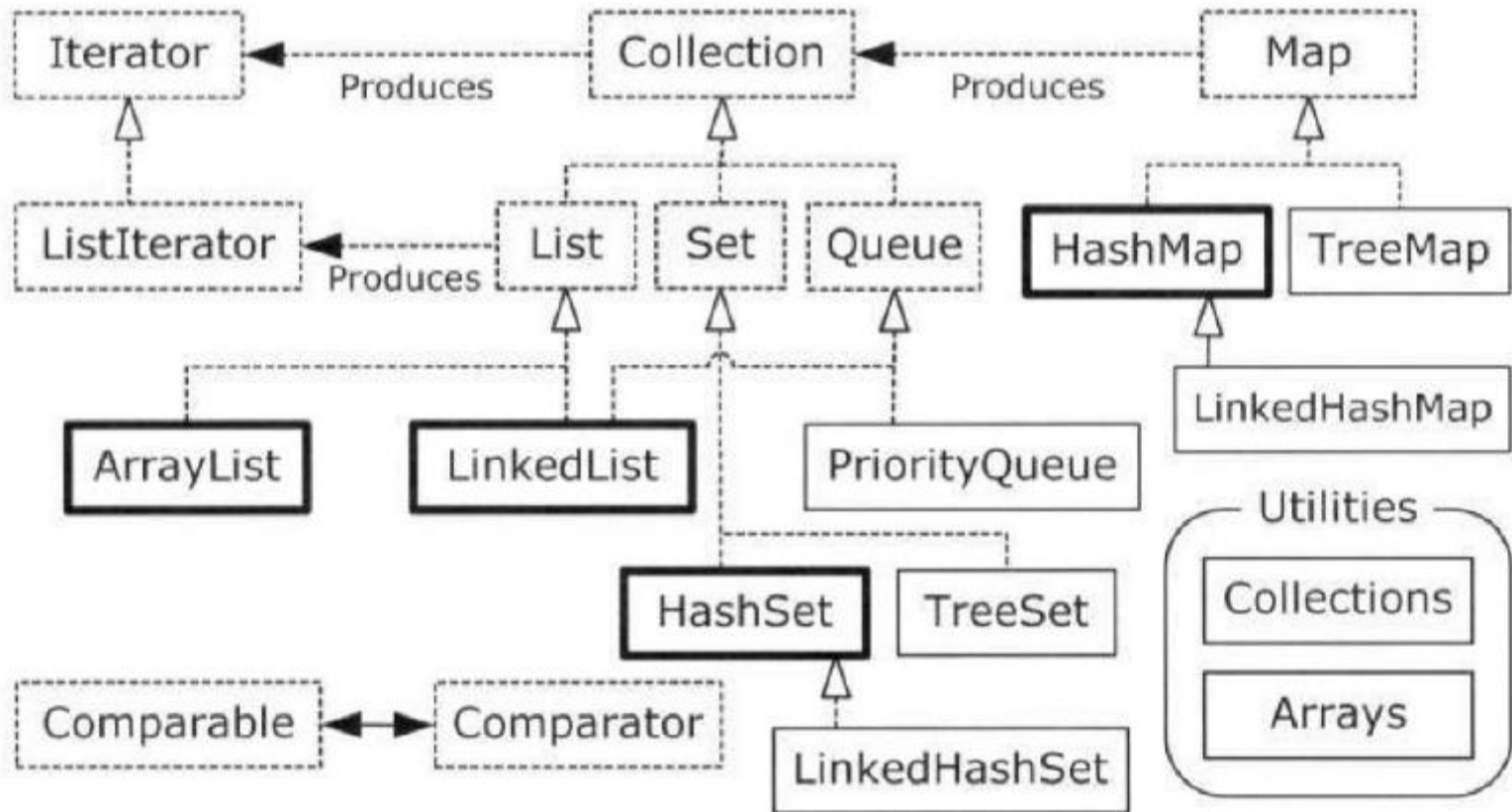


# Collection Implementations

- Classes that implement the collection interfaces typically have names in the form of *<Implementation-style><Interface>*
- Implementations are summarized in the table below:

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	<a href="#">HashSet</a>		<a href="#">TreeSet</a>		<a href="#">LinkedHashSet</a>
List		<a href="#">ArrayList</a>		<a href="#">LinkedList</a>	
Deque		<a href="#">ArrayDeque</a>		<a href="#">LinkedList</a>	
Map	<a href="#">HashMap</a>		<a href="#">TreeMap</a>		<a href="#">LinkedHashMap</a>

# What does it look like?





# public interface Collection<E> extends Iterable<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);  
}
```

# Traversing Collections



- The for-each construct allows you to concisely traverse a collection or array using a for loop
  - for (Object o : collection)  
System.out.println(o);
- Use Iterators as an object



# Iterators

- An Iterator is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired.
- You get an `Iterator` for a collection by calling its `iterator()` method.
- The following is the `Iterator` interface.

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

# Iterators: When to use



- Remove the current element. The for-each construct hides the iterator, so you cannot call remove. Therefore, the for-each construct is not usable for filtering.
- Iterate over multiple collections in parallel.

# Bulk Operations

- *Bulk operations* perform an operation on an entire Collection
- A Collection may provide “bulk” operations

```
boolean containsAll(Collection c);  
boolean addAll(Collection c); // Optional  
boolean removeAll(Collection c); // Optional  
boolean retainAll(Collection c); // Optional  
void clear(); // Optional  
Object[] toArray();  
Object[] toArray(Object a[]);
```

# Array Operations

- The *array operations* allow the contents of a Collection to be translated into an array
- The simple form with no arguments creates a new array of Object.
- The more complex form allows the caller to provide an array or to choose the runtime type of the output array.

```
Object[] toArray();
```

```
Object[] toArray(Object a[]);
```



# The Basics

simple, but not really.

- List
- Set
- Map



# public interface List<E> extends Collection<E>

```
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,  
  
<? 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);  
}
```

# What makes a List a List?

Duplicate elements are allowed  
and position-oriented operations  
are permitted.

# More on Lists

- The List Interface extends the Collection Interface.
- `AbstractList` implements `List` and extends `AbstractCollection`
- `AbstractList` is a convenience class that contains basic concrete implementation.



# ListIterators

- The three methods that ListIterator inherits from Iterator (hasNext, next, and remove) do exactly the same thing in both interfaces. The hasNext and the previous operations are exact analogues of hasNext and next. The former operations refer to the element before the (implicit) cursor, whereas the latter refer to the element after the cursor. The previous operation moves the cursor backward, whereas next moves it forward.
- The nextIndex method returns the index of the element that would be returned by a subsequent call to next, and previousIndex returns the index of the element that would be returned by a subsequent call to previous
- The set method overwrites the last element returned by next or previous with the specified element.
- The add method inserts a new element into the list immediately before the current cursor position.

```
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  
}
```





# Types of Lists

- ArrayList
- LinkedList

# ArrayList

- Extends AbstractList
- Auto-resizeable.
- Based on a simple array.
- Permits the null element.

# ArrayList overview

- Constant time positional access (it's an array)
- One tuning parameter, the initial capacity

```
public ArrayList(int initialCapacity) {  
    super();  
    if (initialCapacity < 0)  
        throw new IllegalArgumentException(  
            "Illegal Capacity: "+initialCapacity);  
    this.elementData = new Object[initialCapacity];  
}
```

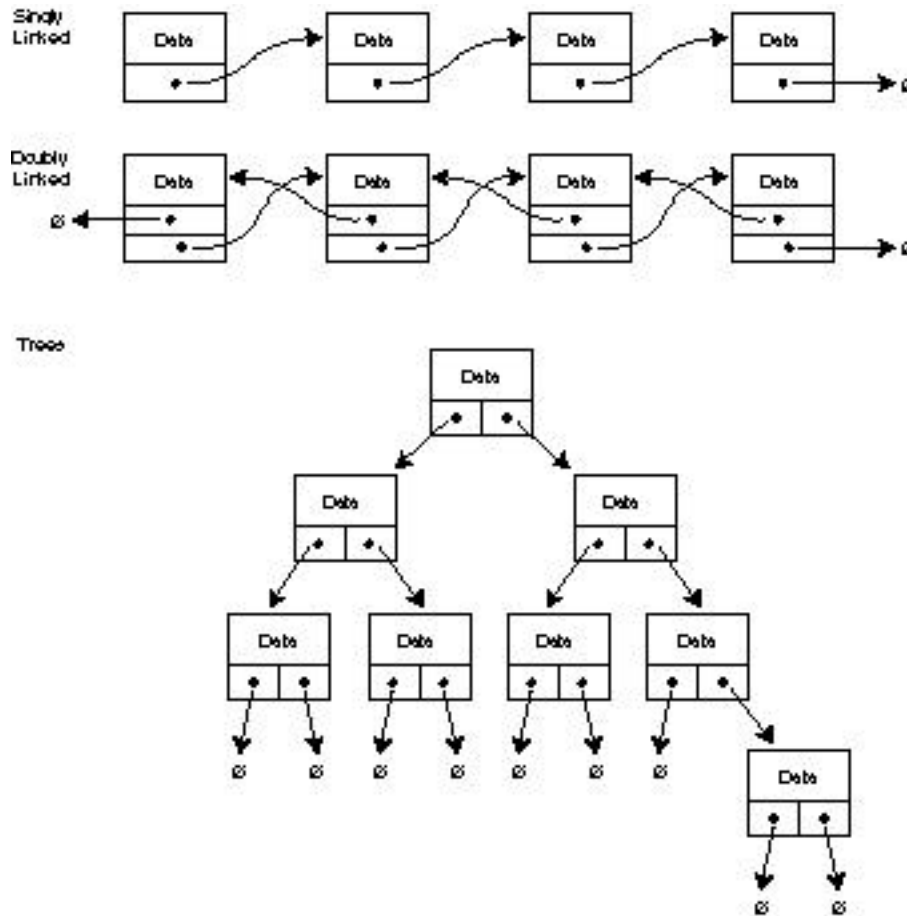
# ArrayList methods

- The indexed get and set methods of the List interface are appropriate to use since ArrayLists are backed by an array
  - `Object get(int index)`
  - `Object set(int index, Object element)`
- Indexed add and remove are provided, but can be costly if used frequently
  - `void add(int index, Object element)`
  - `Object remove(int index)`
- May want to resize in one shot if adding many elements
  - `void ensureCapacity(int minCapacity)`

# LinkedList

- Extends AbstractSequentialList, which extends AbstractList
- Similar to ArrayList
- Different implementation based on nodes that contain data and links to other nodes.

# Nodes In Action



# List Implementations

- ArrayList
  - low cost random access
  - high cost insert and delete
  - array that resizes if need be
- LinkedList
  - sequential access
  - low cost insert and delete
  - high cost random access



# LinkedList overview

- Stores each element in a node
- Each node stores a link to the next and previous nodes
- Insertion and removal are inexpensive
  - just update the links in the surrounding nodes
- Linear traversal is inexpensive
- Random access is expensive
  - Start from beginning or end and traverse each node while counting

# LinkedList entries

```
private static class Entry {
    Object element;
    Entry next;
    Entry previous;

    Entry(Object element, Entry next, Entry previous) {
        this.element = element;
        this.next = next;
        this.previous = previous;
    }
}

private Entry header = new Entry(null, null, null);

public LinkedList() {
    header.next = header.previous = header;
}
```

# LinkedList methods

- The list is sequential, so access it that way
  - `ListIterator listIterator()`
- Listlterator knows about position
  - use `add()` from Listlterator to add at a position
  - use `remove()` from Listlterator to remove at a position
- LinkedList knows a few things too
  - `void addFirst(Object o)`, `void addLast(Object o)`
  - `Object getFirst()`, `Object getLast()`
  - `Object removeFirst()`, `Object removeLast()`



Any questions?



# public interface **Set**<E> extends Collection<E>

```
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);  
}
```

Note: nothing added to Collection interface – except no duplicates allowed

# What makes a Set a Set?

No duplicate elements are allowed, such that

`e1.equals(e2)` is true.

# More on Sets

- The Set Interface extends the Collection Interface.
- AbstractSet implements Set and extends AbstractCollection
- AbstractSet is a convenience class that contains basic concrete implementation.

# Types of Sets

- HashSet
- TreeSet
- LinkedHashSet



# HashSet

- Implements the Set interface, backed by a hash table (actually a HashMap instance).
- Makes no guarantees as to the iteration order of the set.
- It does not guarantee that the order will remain constant over time.
- Permits the null element.

# TreeSet

- Extends AbstractSet, implements SortedSet
- Elements can be kept in ascending order according to compareTo() or compare()
- All elements must be comparable.

# LinkedHashSet

- Extends HashSet.
- Implements doubly-linked list.
- Can retrieve elements based on insertion-order.
- Less efficient than HashSet.



Any questions?

# What makes a Map a Map?

The collection is kept in key/value pairs. Any object can be a key or value. No duplicate keys allowed.

# public interface Map<K,V>



```
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);  
    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);  
    }  
}
```

# More on Maps

- The Map Interface does *not* extend the Collection Interface.
- AbstractMap implements Map and does *not* extend AbstractCollection
- AbstractMap is a convenience class that contains basic concrete implementation.

# More on Maps

- Maps revolve around two basic operations: **get( )** and **put( )**
- **To put a value** into a map, use **put( )**, **specifying the key** and the value
- To obtain a value, call **get( )**, **passing the key as an argument**. The value is returned.



# Map : Collection Views

- A means of iterating over the keys and values in a Map
- **Set keySet ()**
  - returns the Set of keys contained in the Map
- **Collection values ()**
  - returns the Collection of values contained in the Map.  
This Collection is not a Set, as multiple keys can map to the same value.
- **Set entrySet ()**
  - returns the Set of key-value pairs contained in the Map. The Map interface provides a small nested interface called Map.Entry that is the type of the elements in this Set.

# Types of Maps

- TreeMap
- HashMap
- LinkedHashMap

# HashMap and TreeMap

- HashMap
  - The keys are a set - unique, unordered
  - Fast
- TreeMap
  - The keys are a set - unique, ordered
  - Same options for ordering as a TreeSet
    - *Natural order (Comparable, compareTo(Object))*
    - *Special order (Comparator, compare(Object, Object))*

# HashMap

- Implements SortedMap, extends AbstractMap
- Permits the null element.
- Makes no guarantees as to the iteration order of the set.



# More HashMap

- It does not guarantee that the order will remain constant over time.
- Can retrieve elements based on insertion-order or access order.

# TreeMap

- Implements SortedMap, extends AbstractMap
- Keys can be kept in ascending order according to compareTo() or compare()



# LinkedHashMap

- Extends HashMap
- Implements doubly-linked list.



Any questions?



# Ordering Collections

- ◆ The `Comparable` and `Comparator` interfaces are useful for ordering and sorting the elements in a collection.
- ◆ The `Comparable` interface imparts natural ordering to classes that implement it.
- ◆ The `Comparator` interface is used to specify order relation. It can also be used to override natural ordering.

# The Comparable Interface

- ◆ The `Comparable` interface is a member of the `java.lang` package.
- ◆ By implementing the `Comparable` interface, an order to the objects of any class is provided.
- ◆ Collections that contain objects of classes can be sorted that implement the `Comparable` interface
- ◆ .The signature of the method is shown here:  

```
int compareTo(T obj)
```
- ◆ This method compares the invoking object with *obj*.
  - ◆ *It returns 0 if the values are equal.*
  - ◆ A negative value is returned if the invoking object has a lower value.
  - ◆ Otherwise, a positive value is returned.

# The Comparator Interface

- ◆ **Comparator is a generic interface** that has this declaration:  
interface Comparator<T>

Here, **T specifies the type of objects being** compared

- The **Comparator interface defines** two methods:  
**compare( )** and **equals()**.
- **The compare( ) method, shown here,** compares two elements for order:

```
int compare(T obj1, T obj2)
```

# Comparable vs Comparator Interface



Parameter	Comparable	Comparator
Sorting logic	Sorting logic must be in same class whose objects are being sorted. Hence this is called natural ordering of objects	Sorting logic is in separate class. Hence we can write different sorting based on different attributes of objects to be sorted. E.g. Sorting using id,name etc.
Implementation	Class whose objects to be sorted must implement this interface.e.g Country class needs to implement comparable to collection of country object by id	Class whose objects to be sorted do not need to implement this interface.Some other class can implement this interface. E.g.-CountrySortByIdComparator class can implement Comparator interface to sort collection of country object by id
Sorting method	<p><code>int compareTo(Object o1)</code>                      This method compares this object with o1 object and returns a integer.Its value has following meaning</p> <ol style="list-style-type: none"> <li>1. positive – this object is greater than o1</li> <li>2. zero – this object equals to o1</li> <li>3. negative – this object is less than o1</li> </ol>	<p><code>int compare(Object o1,Object o2)</code>                      This method compares o1 and o2 objects. and returns a integer.Its value has following meaning.</p> <ol style="list-style-type: none"> <li>1. positive – o1 is greater than o2</li> <li>2. zero – o1 equals to o2</li> <li>3. negative – o1 is less than o1</li> </ol>
Calling method	<p><code>Collections.sort(List)</code>                      Here objects will be sorted on the basis of <code>CompareTo</code> method</p>	<p><code>Collections.sort(List, Comparator)</code>                      Here objects will be sorted on the basis of <code>Compare</code> method in <code>Comparator</code></p>
Package	<code>Java.lang.Comparable</code>	<code>Java.util.Comparator</code>

# Utilities

- The Collections class provides a number of static methods that operate on or return collections
- Most operate on Lists, some on all Collections
  - Sort, Search, Shuffle
  - Reverse, fill, copy
  - Min, max
- Wrappers
  - synchronized Collections, Lists, Sets, etc
  - unmodifiable Collections, Lists, Sets, etc

# Collections Class



- Contains static methods for operating on collections and maps.
- Quite usefull :-)

# Collections Class: Few Methods



Method Signature	Description
<code>Collections.sort(List myList)</code>	Sort the myList (implementation of any List interface) provided in argument in natural ordering.
<code>Collections.sort(List, comparator c)</code>	Sort the myList(implementation of any List interface) as per comparator c ordering (c class should implement comparator interface)
<code>Collections.shuffle(List myList)</code>	Puts the elements of myList ((implementation of any List interface)in random order
<code>Collections.reverse(List myList)</code>	Reverses the elements of myList ((implementation of any List interface)
<code>Collections.binarySearch(List mlist, T key)</code>	Searches the mlist (implementation of any List interface) for the specified object using the binary search algorithm.
<code>Collections.copy(List dest, List src)</code>	Copy the source List into dest List.
<code>Collections.frequency(Collection c, Object o)</code>	Returns the number of elements in the specified collection class c (which implements Collection interface can be List, Set or Queue) equal to the specified object
<code>Collections.synchronizedCollection(Collection c)</code>	Returns a synchronized (thread-safe) collection backed by the specified collection.

# Arrays Class

- Contains static methods sorting and searching arrays, comparing arrays, and filling array elements.