Java Collections

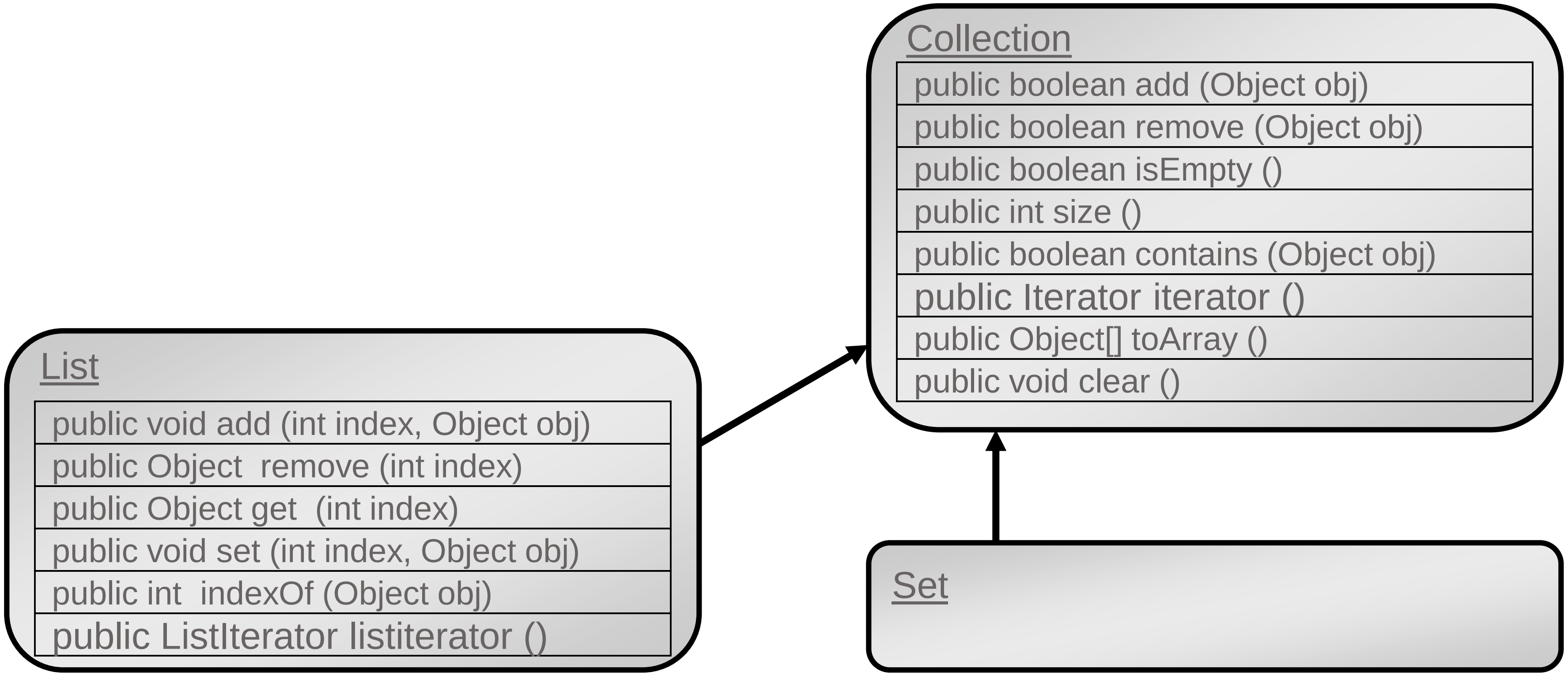
Java Collections

- Java objects that holds an Objects group
- Number of objects in the group is dynamic
- There are 4 types of collections:
  - **Collection** - unordered group, duplicates are permitted
  - **Set** - unordered group, duplicates are forbidden
  - **List** - ordered group, duplicates are permitted
  - **Map** - group of key–value pairs

Collections API

All types are interfaces implemented

by different kinds of classes:


Collection
public boolean add (Object obj)
public boolean remove (Object obj)
public boolean isEmpty ()
public int size ()
public boolean contains (Object obj)
public Iterator iterator ()
public Object[] toArray ()
public void clear ()
List
public void add (int index, Object obj)
public Object remove (int index)
public Object get (int index)
public void set (int index, Object obj)
public int indexOf (Object obj)
public ListIterator listiterator ()
Set

Collections API

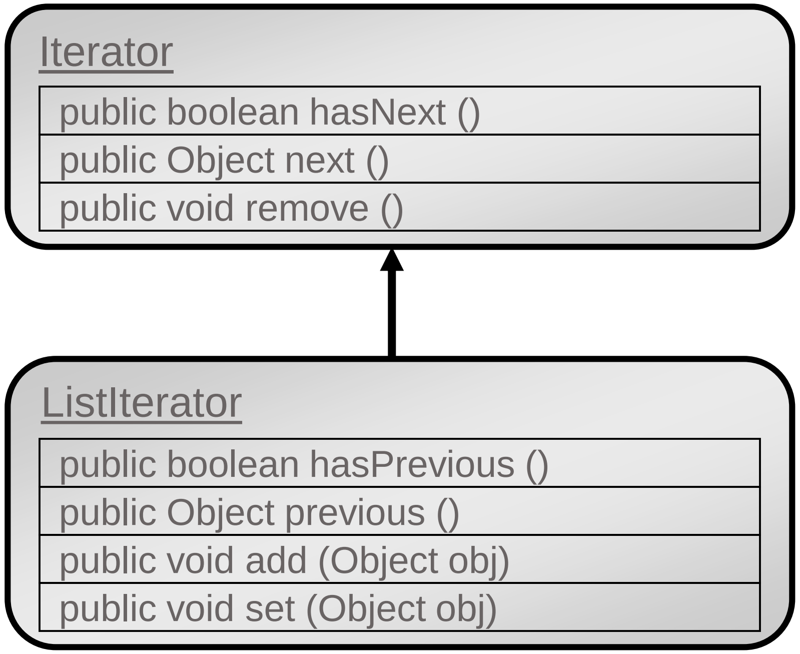
- All type are interfaces implemented by different kinds of classes:

Map

public Object put (Object key, Object value)
public Object get (Object key)
public Object remove (Object key)
public int size ()
public boolean containsKey (Object key)
public boolean isEmpty ()
public Set keySet ()
public void clear ()

Iterators

- Iterator retrieves any Object in a Collection
- Collection & Set Iterators are unordered
- List Iterator is ordered – therefore it has more options


Iterator
public boolean hasNext ()
public Object next ()
public void remove ()
ListIterator
public boolean hasPrevious ()
public Object previous ()
public void add (Object obj)
public void set (Object obj)

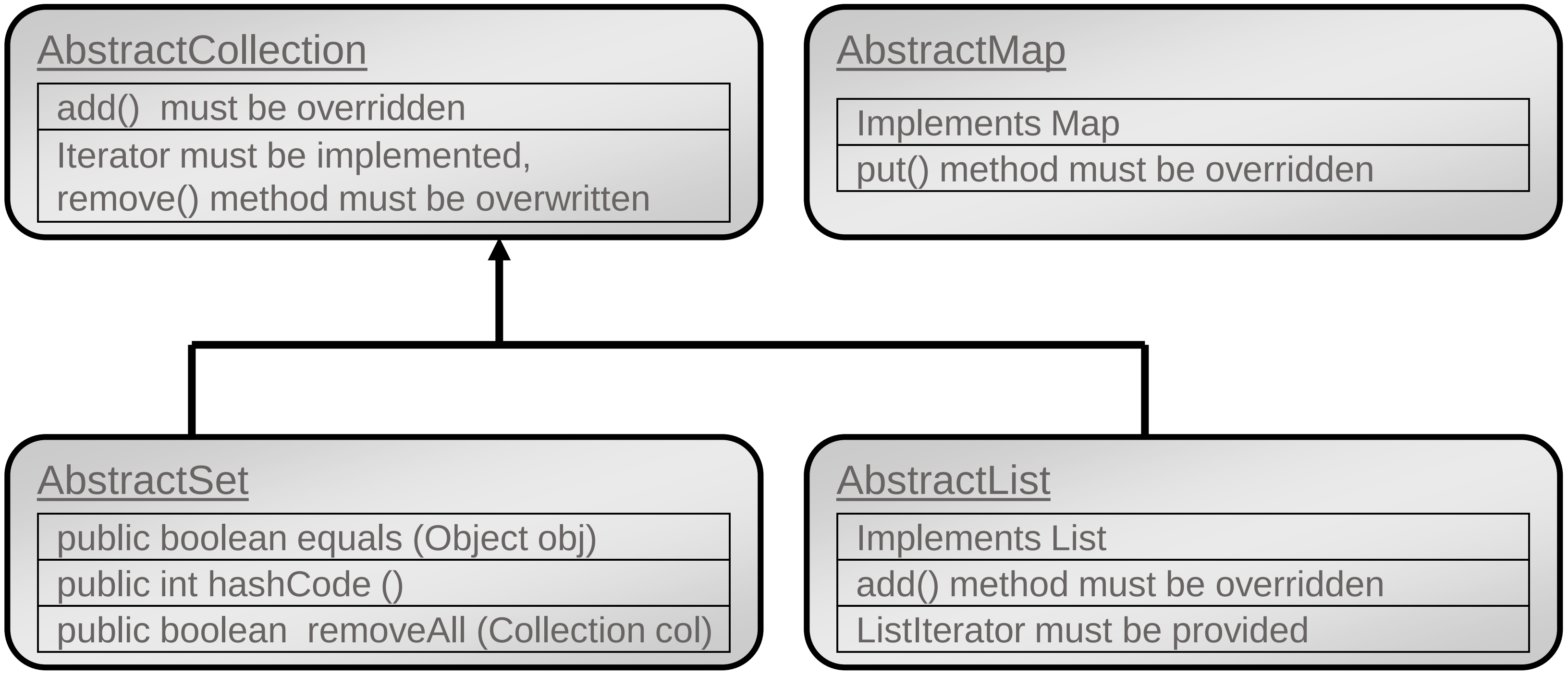
Iterators

- Example:

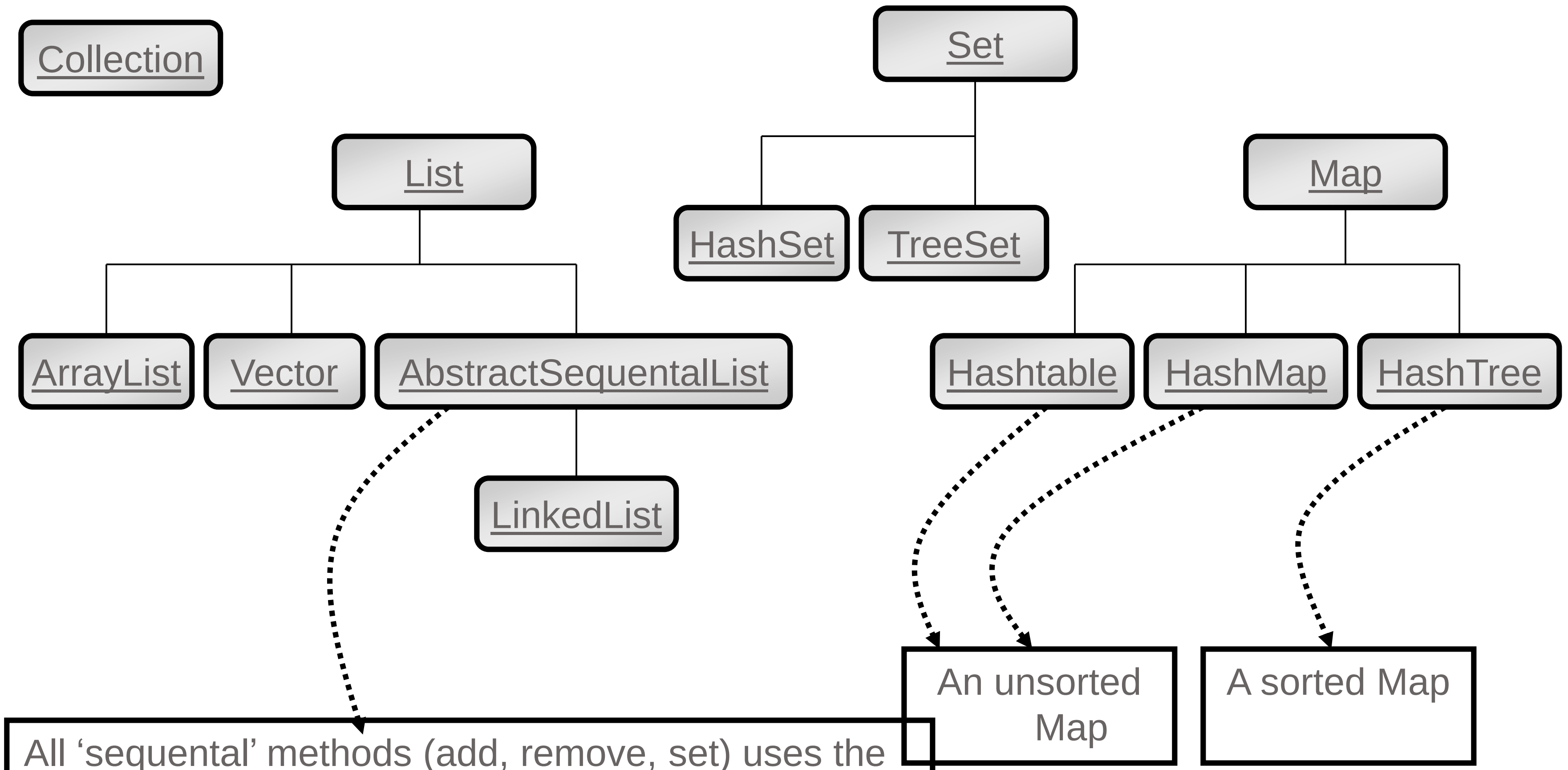
```
Collection col = new ArrayList ();
// add some elements..
Iterator elements = col.iterator();
while (elements.hasNext()){
      System.out.println(elements.next())
}
```

Abstract Adapters

- Abstract adapters used to create customized collections


AbstractCollection
add() must be overridden
Iterator must be implemented,
remove() method must be overwritten
AbstractMap
Implements Map
put() method must be overridden
AbstractSet
public boolean equals (Object obj)
public int hashCode ()
public boolean removeAll (Collection col)
AbstractList
Implements List
add() method must be overridden
ListIterator must be provided

Implementation Classes



All 'sequental' methods (add, remove, set) uses the ListIterator logic for tracking.
Usually, the ListIterator calls the sequental methods

List Implementations

- **Vector** - a synchronized List implementation
  - is thread-safe
- **ArrayList** - a light-weight collection
  - is not thread-safe
- **LinkedList** – A ‘natural’ sequence implementation

List Implementations

- Example:

```
public static void main (String [] args){
      List list=new ArrayList();
       list.add(new Integer(4));
       list.add(new Integer(3));
       list.add(“Hello”);
       list.add(new Integer(4));
       list.add(“Bye”);
      list.add(“Hello”);
      System.out.println(list);
}
```

Output:

```
                [4, 3, Hello, 4, Bye, Hello]
```

Set Implementations

- **HashSet** – a set that reflects a HashMap
- **TreeSet** – a set that reflect a TreeMap
- Both are not thread-safe

Set Implementations

- Example:

```
public static void main (String [] args){
      Set list=new HashSet();
       list.add(new Integer(4));
       list.add(new Integer(3));
       list.add(“Hello”);
       list.add(new Integer(4));
       list.add(“Bye”);
      list.add(“Hello”);
      System.out.println(list);
}
```

Duplicate, not added

Duplicate, not added

Possible Output 1:

```
[4, 3, Hello, Bye]
```

Possible Output 2:

```
[Hello, 4, Bye, 3]
```

Map Implementations

- **HashMap** - unsorted map
  - is not thread-safe

- **Hashtable** - like HashMap but:
  - is thread-safe
  - null values are not permitted

- **HashTree** - a sorted map
  - is not thread-safe

Map Implementations

- Example:

```
public static void main (String [] args){
      Map list=new HashMap();
       list.put(new Integer(1),”One”);
      list.put(new Integer(2),”Two”);
      list.put(new Integer(3),”Three”);
      list.put(new Integer(4),”Four”);
      list.put(new Integer(5),”Five”);
      list.put(new Integer(1),”Six”);
      System.out.println(list);
}
```

Exising key – value is replaced

Possible Output :

```
{1=six, 2=Two, 3=Three, 4=Four, 5=Five }
```

When & What to use ?

Some points to consider:

- When single thread is involved – no need in thread-safe collections
- Use List only when the collection must be ordered
- For object pool implementation use Set or HashTable/Map
- Use the initial-size and increment-size parameters of the collection’s constructor
- When there’s key-value pairs – prefer HashTable [for quicker search]
- List: Vector is synchronized & ArrayList is not
- To synchronize the work done on a non thread-safe collection use:

```
Set s = Collections.synchronizedSet(new HashSet(...));
```

Enumeration

- A primitive version of Iterator
- Can be used for any purpose – like in StringTokenizer
- Methods:
- Is returned by:

Enumeration

public boolean hasMoreElements ()
public Object nextElement ()

Hashtable.keys ()
Hashtable.elements ()
Vector.elements ()

Sorting Arrays & Collections

- Sorting arrays: Arrays.sort method:

java.util.Arrays

public void sort (<type> array)
public void sort (<type> array, int from, int to)

<type> - all primitives except boolean

- Sorting Lists: Collections.sort

java.util.Collections

public void sort (List list) [Comparable Objects]
public void sort (List list, Comparator comp)

Comparator is discussed in the next slide

- Sorting Sets: use SortedSet implementations like TreeSet

  (which also uses Comparator)

Sorting Arrays & Collections

Comparator

- Specifies how to compare between two Objects
- Is used to make a comparison according to the application logic
- When sorting Lists or Sets – several Comparators can be used

- compare() method will return :
  Positive value – if O1>O2
  Negative value – if O1<O2
  Zero – if O1 logically equals to O2

java.util.Comparator Interface

public int compare (Object o1, Object o2)
public boolean equals ()

- equals() check if two Comparable objects are equal – programmer can use the logic inherited from java.lang.Object

Sorting Arrays & Collections

- Comparable

- Specifies how an object is compared to another

- All wrapper classes are comparable – compared according to their wrapped values

- When sorting Lists without a Comparator, all objects must be Comparable otherwise a ClassCastException is thrown

- compareTo() method will return :
  - Positive value – if ‘this’>O
  - Negative value – if ‘this’<O
  - Zero – if ‘this’ logically equals to O2

java.lang.Comparable Interface

```
public int compareTo (Object o)
```

References

- http://java.sun.com/
- SUN Educational Services SL-275