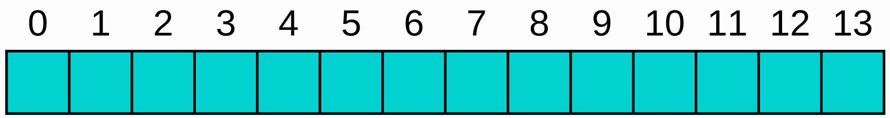
Data Structures

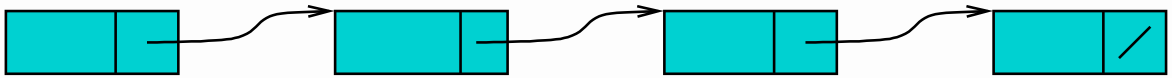
The basic data structures are:

- array
- list
- hashtable
- tree

Array: indexed access, can be resizable

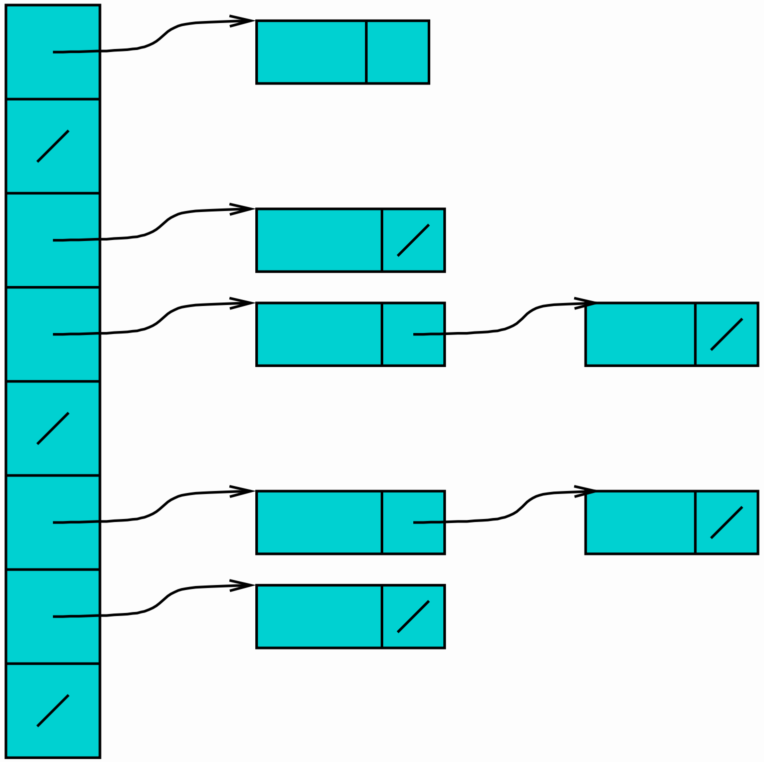


List: singly or doubly linked, can be circular

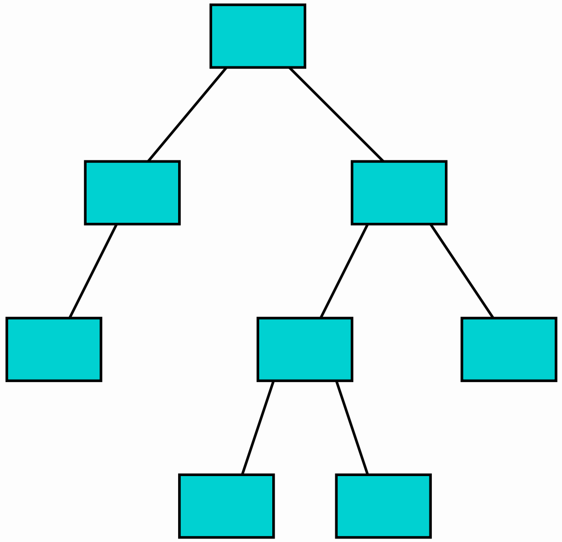


Data Structures (cont.)

Hashtable:



Tree:



Abstract Data Types

Data structures support different operations:

- insert an element
- remove an element
- search an element
- . . .

Abstract data types are interfaces specifying what operations are provided. Examples of ADTs:

- **Set**
- **Dictionary** – also called **Map**
- **Vector** – resizable array
- **Stack** – also called **LIFO**
- **Queue** – also called **FIFO**
- **Priority Queue**

Collections in Java

A **collection** (sometimes called **container**) is an objects that groups multiple elements into single unit.

Earlier versions of Java included the following collections:

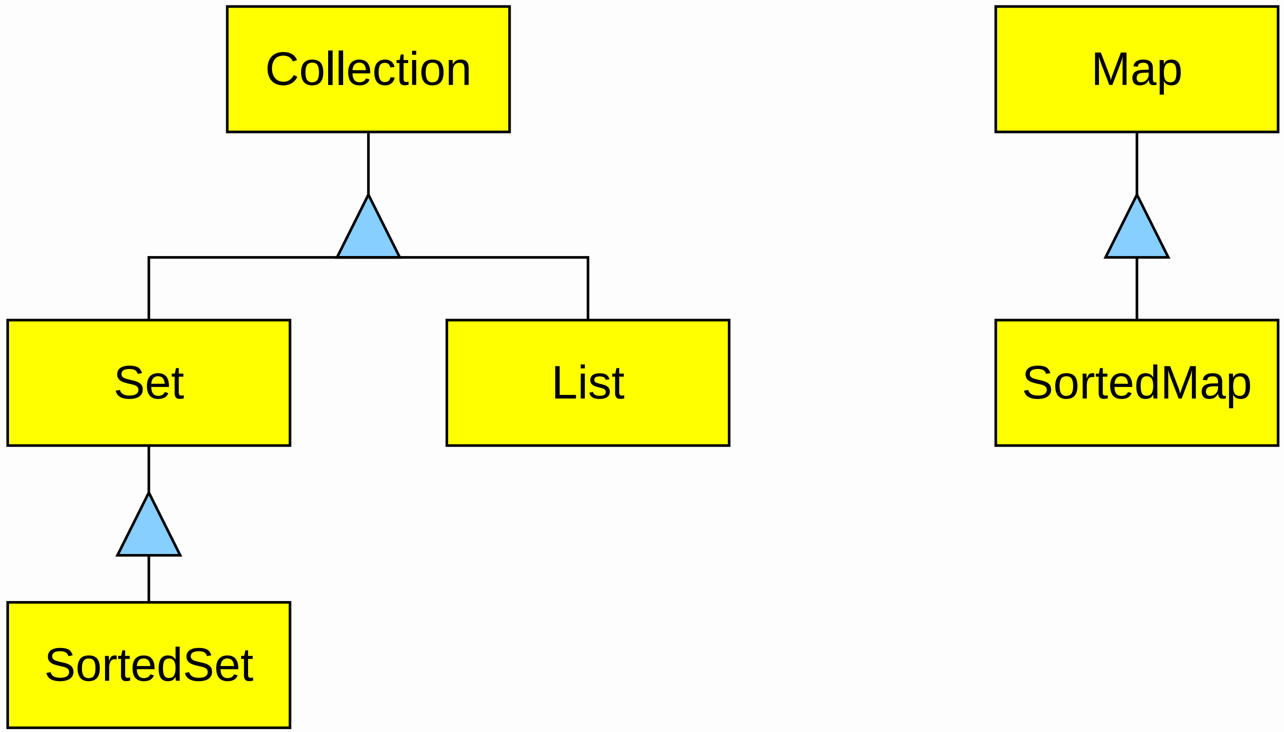
- java.util.Vector
- java.util.Hashtable
- array

Current versions of Java contain **collection framework** – a unified architecture for representing and manipulating collections. It consists of:

- **Interfaces** – abstract data types representing collections
- **Implementations** – concrete implementations of the interfaces
- **Algorithms** – methods that perform useful computations (searching and sorting)

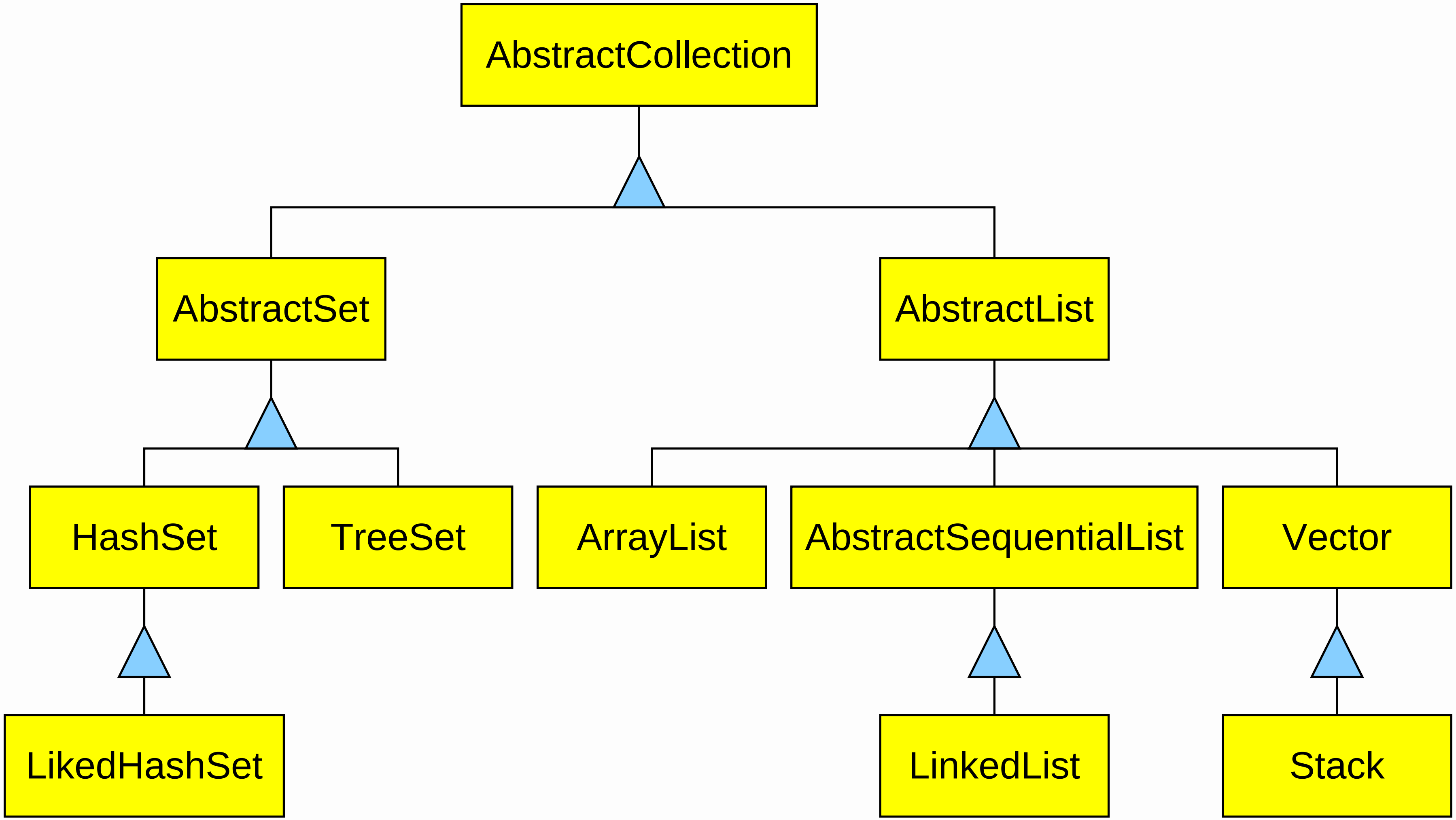
Interfaces

The collection interfaces in the package **java.util** form a hierarchy:



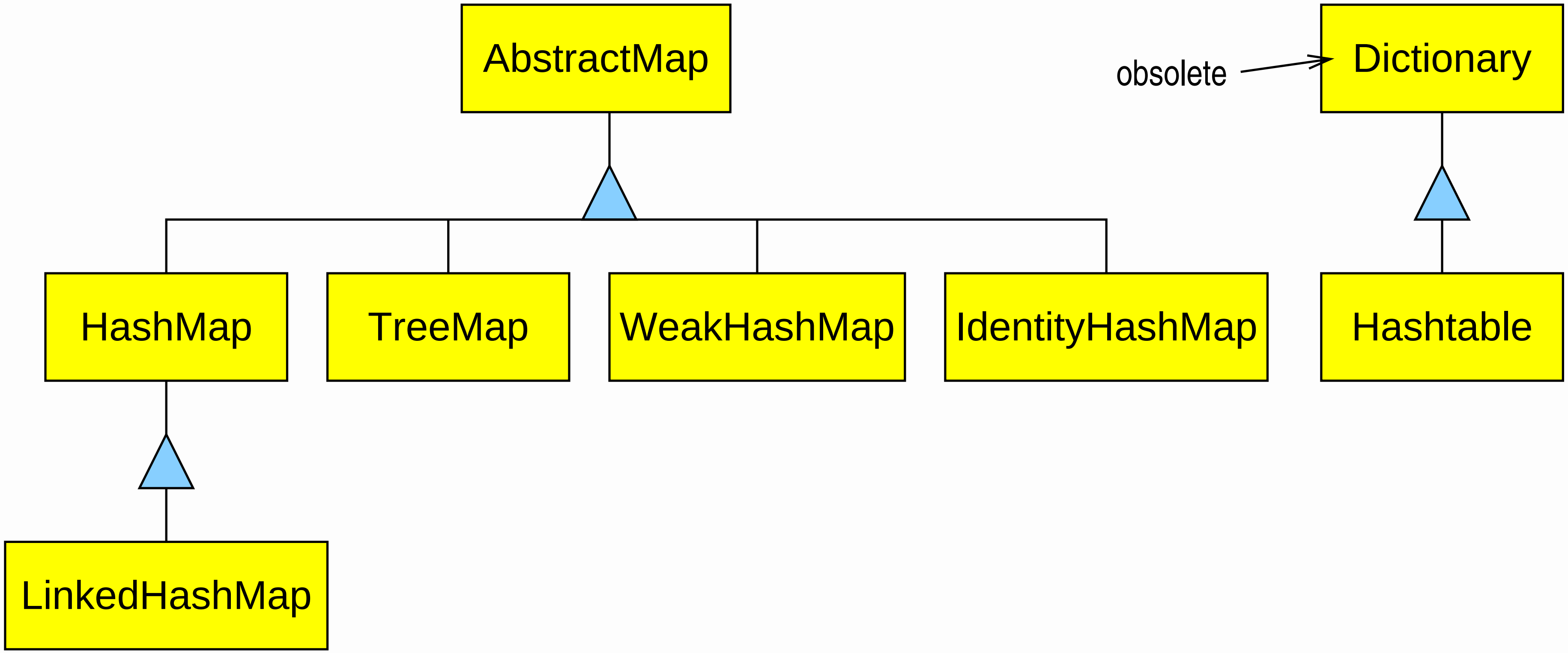
Implementations

The classes implementing collections in the package **java.util**:



Implementations (cont.)

The classes implementing maps:



The Collection Interface

The **Collection** is the root of the collection hierarchy.

A **Collection** represents a group of objects – its **elements.** (Some implementations allow duplicate elements and others do not.)

The primary use of the **Collection** interface is pass around collections of objects where maximum generality is desired.

The **Collection** interface declares the following basic operations:

- int size()
- boolean isEmpty()
- boolean contains(Object o)
- boolean add(Object o) – *optional*
- boolean remove(Object o) – *optional*
- Iterator iterator()

Note: Some operation are designated as **optional.** Implementations that do not implement them throw an **UnsupportedOperationException**.

Iterators

An **iterator** provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

The **java.util.Iterator** provides uniform interface for traversing different aggregate structures.

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

Example of use:

```
Collection c = new ArrayList();
   . . .    // fill the collection
for (Iterator i = c.iterator(); i.hasNext(); ) {
    Object o = i.next();
       . . .  // process the element
}
```

Enumerations

Earlier implementations of Java used the **java.util.Enumeration** interface instead of **iterator**:

```
public interface Enumeration {
    boolean hasMoreElements();
    Object nextElement();
}
```

The differences between them are:

- **Iterator** allows the caller to remove elements from the underlying collection.
- Method names have been improved in **Iterator**.

New implementations should use **Iterator** in preference to **Enumeration**.

Iterators (cont.)

The **Iterator** interface contains the optional method `remove()` that removes from the underlying collection the last element that was returned by `next()`:

- The `remove()` method may be called only once per call to `next()` – an exception is thrown if this condition is violated.
- The `remove()` method is the **only** safe way to modify a collection during iteration.
- The behavior is unspecified if the underlying collection is modified in any other way while iteration is in progress.

Bulk Operations

The **bulk operations** perform some operation on an entire **Collection** in a single shot:

- boolean containsAll(Collection c)
- boolean addAll(Collection c) – *optional*
- boolean removeAll(Collection c) – *optional*
- boolean retainAll(Collection c) – *optional*
- void clear() – *optional*

For example. to remove **all** instances of a specified element e from a collection c we can use:

```
c.removeAll(Collections.singleton(e));
```

Note: The class **Collections** contains many useful static methods that operate on collections. The `singleton()` method returns an immutable collection (set) containing only the specified object.

Array Operations

The `toArray()` allow the contents of a **Collection** to be translated into an array:

- Object[] toArray()
- Object[] toArray(Object[] a)

The following code dumps the contents of c into a newly allocated array:

```
Object[] a = c.toArray();
```

Suppose c is a collection known to contain only strings. The following code can be used to dump the contents of c into a newly allocated array of `String`:

```
String[] a = (String[])c.toArray(new String[0]);
```

Note: If the collection fits in the specified array, this array is used, otherwise a new array is allocated.

The Set Interface

A **Set** is a **Collection** that cannot contain duplicate elements. It models a mathematical **set** abstraction.

The **Set** interface contains **no** methods than those inherited from **Collection**.

There are two general-purpose **Set** implementations:

- **HashSet** – stores its elements in a hashtable, it is the best-performing implementation.
- **TreeSet** – stores its elements in a red-black tree, guarantees the order of iteration (the elements will be sorted).

The following code creates a new collection containing the same elements as the collection c, but with all duplicates eliminated:

```
Collection d = new HashSet(c);
```

The Set Interface (cont.)

Example of use of a **Set** that prints out any duplicate words, the number of distinct words, and a list of the words with duplicates eliminated:

```
import java.util.*;

public class FindDuplicates {
    public static void main(String[] args) {
        Set s = new HashSet();
        for (int i = 0; i < args.length; i++) {
            if (!s.add(args[i])) {
                System.out.println("Duplicate detected: "
                                   + args[i]);
            }
        }
        System.out.println(s.size() +
                           " distinct words detected: " + s);
    }
}
```

The Set Interface (cont.)

The bulk operations on sets correspond to standard set-algebraic operations:

- `s1.containsAll(s2)` – returns **true** if s2 is a **subset** of s1

$$s_2 \subseteq s_1$$

- `s1.addAll(s2)` – transforms s1 into the **union** of s1 and s2

$$s_1 \cup s_2$$

- `s1.retainAll(s2)` – transforms s1 into the **intersection** of s1 and s2

$$s_1 \cap s_2$$

- `s1.removeAll(s2)` – transforms s1 into the **set difference** of s1 and s2

$$s_1 - s_2$$

The List Interface

A **List** is an ordered **Collection** (sometimes called a **sequence**). Lists may contain duplicate elements.

There are two general-purpose **List** implementations:

- **ArrayList** – generally the best-performing implementation
- **LinkedList** – offers better performance under certain circumstances

The **List** contains methods for positional access that manipulate elements based on their numerical position in the list:

- Object get(int index)
- Object set(int index, Object element) – *optional*
- void add(int index, Object element) – *optional*
- Object remove(int index) – *optional*
- boolean addAll(int index, Collection c) – *optional*

The List Interface (cont.)

For example, the following method swaps two elements of a list:

```
private static void swap(List a, int i, int j) {
    Object tmp = a.get(i);
    a.set(i, a.get(j));
    a.set(j, tmp);
}
```

The following method randomly permutes the specified **List** using the specified source of randomness:

```
public static void shuffle(List a, Random rnd) {
    for (int i = a.size(); i > 1; i--) {
        swap(a, i-1, rnd.nextInt(i));
    }
}
```

Note: The class **Collections** contains such method shuffle().

The List Interface (cont.)

- The `remove()` operation always removes the **first** occurrence of the specified element.
- The `add()` and `addAll()` operations always append the new element(s) to the **end** of the list.
- To concatenate one list to another we can use:

```
list1.addAll(list2);
```

- The non-destructive version of concatenation:

```
List list3 = new ArrayList(list1);
list3.addAll(list2);
```

- The **List** interface contains two methods for searching:
  - int indexOf(Object o)
  - int lastIndexOf(Object o)

The ListIterator Interface

The **List** interface supports its own extended version of iterator:

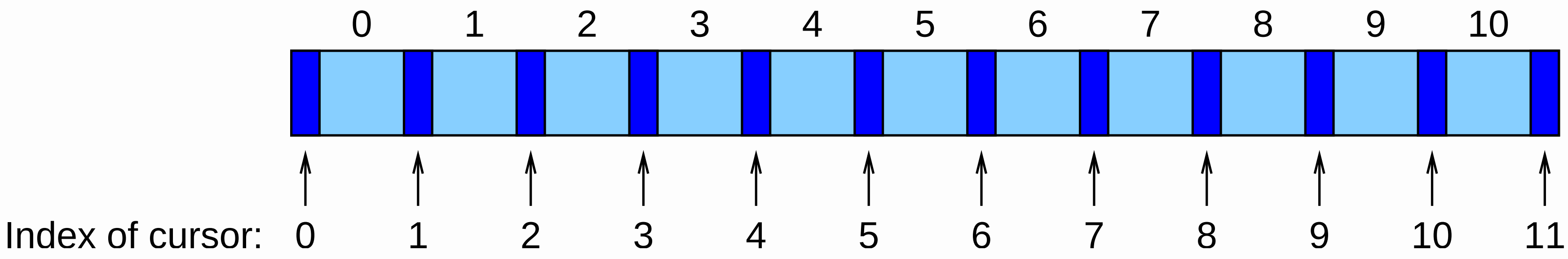
```
public interface ListIterator extends Iterator {
    boolean hasNext();
    Object next();

    boolean hasPrevious();
    Object previous();

    int nextIndex();
    int previousIndex();

    void remove();          // optional
    void set(Object o);     // optional
    void add(Object o);     // optional
}
```

The `ListIterator` Interface (cont.)

To obtain `ListIterator` we can use **List** methods:

- ListIterator listIterator()
- ListIterator listIterator(int index)

A list iterators has a cursor pointing **between** elements:



The ListIterator Interface (cont.)

Iterating backwards in a list:

```
for (ListIterator i = list.listIterator(list.size());
        i.hasPrevious(); ) {
    Object o = i.previous();
     . . .
}
```

A method that replaces all occurrences of one specified value with another:

```
public static void replace(List l, Object x, Object y) {
    for (ListIterator i = l.listIterator(); i.hasNext(); ) {
        if (x == null ? i.next() == null
                      : x.equals(i.next())) {
            i.set(y);
        }
    }
}
```

The List Interface (cont.)

The **List** interface contains a method returning a **range-view**:

- List subList(int fromIndex, int toIndex)

The returned **List** contains the portion of the original list whose indexes range from `fromIndex`, inclusive, to `toIndex`, exclusive.

Changes in the former **List** are reflected in the latter.

For example, to remove a range of elements from a list we can use:

```
list.subList(fromIndex, toIndex).clear();
```

Searching for an element in a range:

```
int i = list.subList(fromIndex, toIndex).indexOf(o);
int j = list.subList(fromIndex, toIndex).lastIndexOf(o);
```

The Collections Class

The **Collections** class contains static methods implementing different algorithms working on collections. Most of them apply specifically to **List**:

- void sort(List list)
- int binarySearch(List list, Object key)
- void reverse(List list)
- void shuffle(List list)
- void fill(List list, Object obj)
- void copy(List dest, List src)

There is a similar class called **Arrays** containing as static methods algorithms working on arrays.

The Map Interface

A **Map** is an object that maps keys to values.

A map cannot contain duplicate keys: Each key can map to at most one value.

The most important methods:

- Object put(Object key, Object value) – *optional*
- Object get(Object key)
- Object remove(Object key) – *optional*
- boolean containsKey(Object key)
- boolean containsValue(Object value)
- int size()
- boolean isEmpty()

The Map Interface (cont.)

Other methods:

- void putAll(Map t) – *optional*
- void clear() – *optional*
- Set keySet()
- Collection values()
- Set entrySet()

The **Collection**-view methods provide the **only** means to iterate over a **Map**:

```
for (Iterator i = m.keySet().iterator(); i.hasNext(); ) {
    System.out.println(i.next());
}
```

The Map Interface (cont.)

There are two general-purpose **Map** implementations:

- **HashMap** - stores its entries in a hash table, it is the best-performing implementation
- **TreeMap** - stores its entries in a red-black tree, guarantees the order of iteration

There is also an older class **Hashtable**.

Hashtable has been retrofitted to implement **Map.**

Object Ordering

Objects that implement the **java.lang.Comparable** interface can be ordered automatically. The **Comparable** interface provides **natural ordering** for a class:

```
public interface Comparable {
    public int compareTo(Object o);
}
```

The method `o1.compareTo(o2)` returns:

- a negative integer – if o1 is less than o2
- zero – if o1 is equal to o2
- a positive integer – if o1 is greater than o2

Many standard classes such as **String** and **Date** implement the **Comparable** interface.

Object Ordering (cont.)

```
import java.util.*;

public class Name implements Comparable {
    private String firstName, lastName;

      . . .

    public boolean equals(Object o) {
        if (!(o instanceof Name)) return false;
        Name n = (Name)o;
        return firstName.equals(n.firstName) &&
               lastName.equals(n.lastName);
    }

    public int hashCode() {
        return 31 * firstName.hashCode() +
               lastName.hashCode();
    }

      . . .
```

Object Ordering (cont.)

```
    . . .

    public int compareTo(Object o) {
        Name n = (Name)o;
        int cmp = lastName.compareTo(n.lastName);
        if (cmp != 0) return cmp;
        return firstName.compareTo(n.firstName);
    }
}
```

Note how methods `equals()` and `hashCode()` are redefined to be consistent with `compareTo()`.

Comparators

If we want to sort objects in some other order than natural ordering, we can use the **Comparator** interface:

```
public interface Comparator {
    int compare(Object o1, Object o2);
}
```

A **Comparator** is an object that encapsulates ordering.

The `compare()` method compares two its arguments, returning a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

Methods implementing different algorithms in classes **Collections** and **Arrays** allow to specify the comparator that should be used in these algorithms.

The SortedSet Interface

A **SortedSet** is a **Set** that maintains its elements in ascending order, sorted according to the elements' **natural order**, or according to a **Comparator** provided at **SortedSet** creation time.

The **SortedSet** adds the following methods to the methods declared in the **Set** interface:

- SortedSet subSet(Object fromElement, Object toElement)
- SortedSet headSet(Object toElement)
- SortedSet tailSet(Object fromElement)
- Object first()
- Object last()
- Comparator comparator()

The SortedSet Interface (cont.)

There are some differences on behavior of methods inherited from the **Set** interface:

- The iterator returned by the `iterator()` traverses the sorted set in order.
- The array returned by `toArray()` contains the sorted set's elements in order.

The **SortedSet** interface is implemented by the class:

- **TreeSet**

The SortedMap Interface

A **SortedMap** is a **Map** that maintains its entries in ascending order, sorted according to the keys' **natural order**, or according to a **Comparator** provided at **SortedMap** creation time.

The **SortedMap** adds the following methods to the methods declared in the **Map** interface:

- Comparator comparator()
- SortedMap subMap(Object fromKey, Object toKey)
- SortedMap headMap(Object toKey)
- SortedMap tailMap(Object fromKey)
- Object firstKey()
- Object lastKey()

There is one class implementing the **SortedMap** interface:

- **TreeMap**

Implementations

The general-purpose implementations are summarized in the table below:

	Implementations			
	Hash Table	**Resizable Array**	**Balanced Tree**	**Linked List**
Set	HashSet		TreeSet	
List		ArrayList		LinkedList
Map	HashMap		TreeMap	

The **SortedSet** and **SortedMap** interfaces are implemented by **TreeSet** and **TreeMap** classes.

The BitSet Class

The **java.util.BitSet** class implements a vector of bits that grows as needed.

Each component of the bit set has a boolean value. The bits of a BitSet are indexed by nonnegative integers. Individual indexed bits can be examined, set, or cleared.

One **BitSet** may be used to modify the contents of another **BitSet** through logical AND, logical inclusive OR, and logical exclusive OR operations.

The **BitSet** class can used as an efficient implementation of a set if the corresponding universe of possible values is finite and small.

The logical operations then correspond to the set operations.

Note: The **BitSet** class is not part of the collection framework.