Classes currently available in jlib

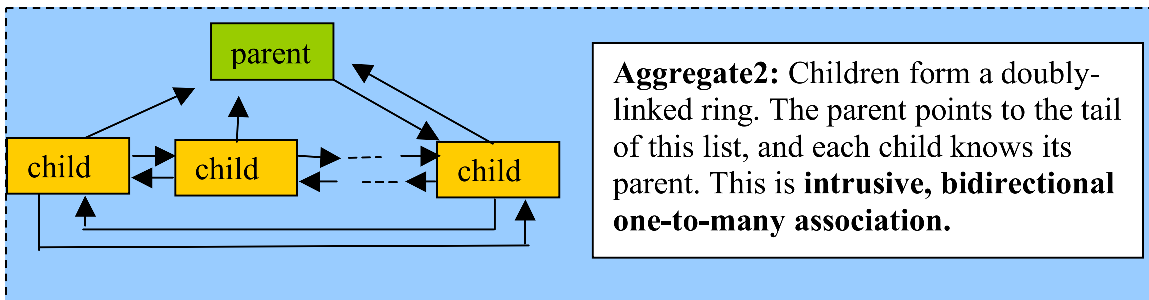
All the sets can be traversed in the same style. For example:

```
    Association Aggregate2<Parent,Child> pc;
    Parent p; Child c;
    Iterator it=new pc_Iterator();
    Forward traversal: for(c=it.fromHead(p); c!=null; c=it.next()){...}
    Reverse traversal: for(c=it.fromTail(p); c!=null; c=it.next()){...}
```

The reverse traversal is available only for doubly-linked sets.

You can remove an item from the set while traversing, for example

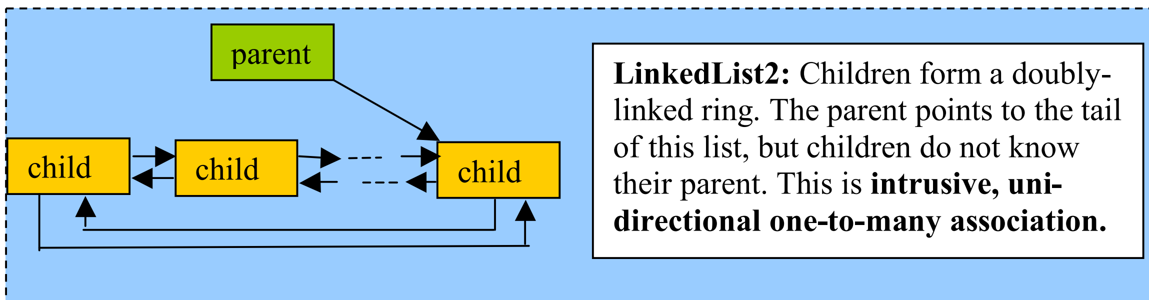
```
    for(c=it.fromHead(p); c!=null; c=it.next()){
        ...
        pc.remove(c);
    }
```




```
class Aggregate2 {      // default association Bi1toX
  public static Parent addHead(Parent p, Child c);
  public static Parent addTail(Parent p, Child c);
  public static Parent append(Child c1, Child c2); // append c2 after c1
  public static void insert(Child c1, Child c2); // insert c2 before c1
  public static Parent remove(Child c); // disconnects c but does not destroy it
  public static Parent parent(Child c);
  public static Child tail(Parent p);
  public static Child head(Parent p);
  public static Child next(Child c); // when c is tail, returns null
  public static Child prev(Child c); // when c is head, returns null
  public static Child nextRing(Child c);
  public static Child prevRing(Child c);
  public static Parent sort(Parent p);      // see note 1
  public static void merge(Child s,Child t,Parent p);     // see note 2
  public static void setTail(Parent p,Child c,int check); // see note 3
}

class Aggregate2Iterator {
  public Child fromHead(Parent p);
```

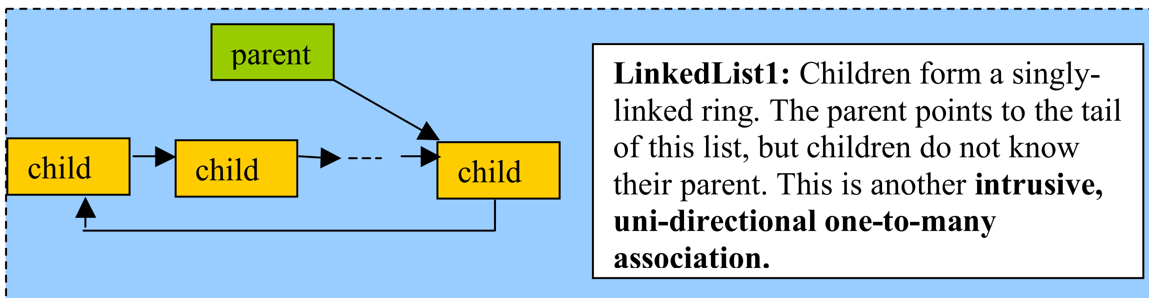
```
  public Child fromTail(Parent p);
  public Child next();
}
```

```
NOTES:
(1) In order to use this method, class Child must implement interface
    Comparable.compareTo(Object).
(2) merge() can either merge two rings, or to split one:
    If s and t have different parents, p must be the parent of either s or t,
    and the two rings merge under p. The other parent is left without children.
    If s and t have the same parent, p must be a parent without children, and
    the original ring splits at s and t, with the ring containing t under p.
(3) setTail() resets the tail within the same ring. For check=1, the method
    performs a consistency check which in this case is relatively expensive.
    For check=0, the operation is very fast, but potentially dangerous.
```




```
class LinkedList2 {
  public static Parent addHead(Parent p, Child c);
  public static Parent addTail(Parent p, Child c);
  public static Parent append(Parent p,Child c1, Child c2); // append c2 after c1
  public static void insert(Parent p,Child c1, Child c2); // insert c2 before c1
  public static Parent remove(Parent p,Child c); // disconnect c, not destroy it
  public static Child tail(Parent p);
  public static Child head(Parent p);
  public static Child next(Parent p,Child c); // null when c is tail
  public static Child prev(Parent p,Child c); // null when c is head
  public static Child nextRing(Child c);
  public static Child prevRing(Child c);
  public static Parent sort(Parent p);                    // see note 1
  public static void merge(Child s,Child t,Parent p);    // see note 2
  public static void setTail(Parent p,Child c,int check); // see note 3
}

class LinkedList2Iterator {
  public Child fromHead(Parent p);
  public Child fromTail(Parent p);
  public Child next();
}
```

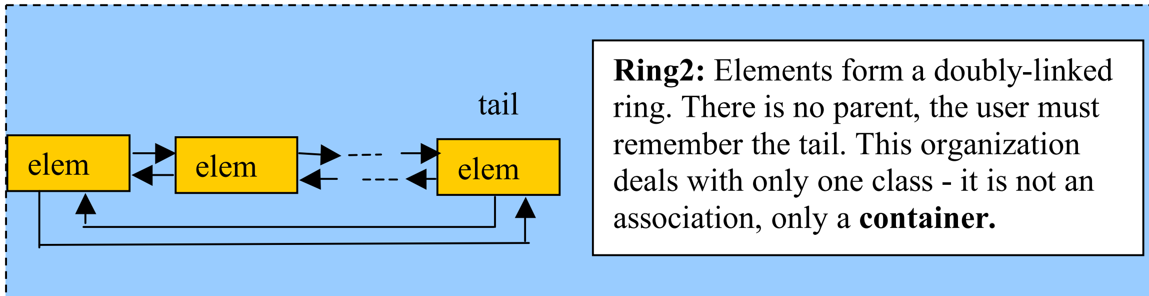
```
NOTES:
 (1) In order to use this method, class Child must implement interface
     Comparable.compareTo(Object).
 (2) merge() can either merge two rings, or to split one:
    If s and t have different parents, p must be the parent of either s or t,
    and the two rings merge under p. The other parent is left without children.
    If s and t have the same parent, p must be a parent without children, and
    the original ring splits at s and t, with the ring containing t under p.
(3) setTail() resets the tail within the same ring. For check=1, the method
    performs a consistency check which in this case is relatively expensive.
    For check=0, the operation is very fast, but potentially dangerous.
```




```
class LinkedList1 {
  public static Parent addHead(Parent p, Child c);
  public static Parent addTail(Parent p, Child c);
  public static Parent append(Parent p,Child c1, Child c2); // append c2 after c1
  public static Parent remove(Parent p,Child c); // disconnect c, not destroy it
  public static Child tail(Parent p);
  public static Child head(Parent p);
  public static Child next(Parent p,Child c); // null if c is tail
  public static Child nextRing(Child c);
  public static Parent sort(Parent p);                           // see note 1
  public static void merge(Child s,Child t,Parent p);      // see note 2
  public static void setTail(Parent p,Child c,int check);  // see note 3
}

class linkedList1Iterator {
  public Child fromHead(Parent p);
  public Child next();
}
```

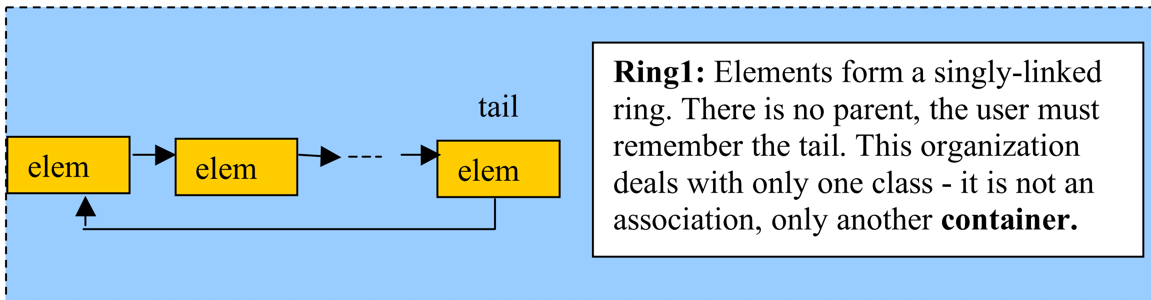
```
NOTES:
 (1) In order to use this method, class Child must implement interface
     Comparable.compareTo(Object).
 (2) merge() can either merge two rings, or to split one:
    If s and t have different parents, p must be the parent of either s or t,
    and the two rings merge under p. The other parent is left without children.
    If s and t have the same parent, p must be a parent without children, and
```

```
    the original ring splits at s and t, with the ring containing t under p.
(3) setTail() resets the tail within the same ring. For check=1, the method
    performs a consistency check which in this case is relatively expensive.
    For check=0, the operation is very fast, but potentially dangerous.
```



Ring2: Elements form a doubly-linked ring. There is no parent, the user must remember the tail. This organization deals with only one class - it is not an association, only a **container.**

```
class Ring2 {
  // unless otherwise stated, methods return the new tail
  public static Elem addHead(Elem tail, Elem e);
  public static Elem  addTail(Elem tail, Elem e);
  public static Elem  append(Elem tail,Elem c1, Elem c2); // append c2 after c1
  public static Elem  insert(Elem tail,Elem c1, Elem c2); // insert c2 before c1
  public static Elem  remove(Elem tail,Elem c); // disconnect c, not destroy it
  public static Elem next(Elem tail,Elem c); // null if c is tail
  public static Elem prev(Elem tail,Elem c); // null if c is head
  public static Elem nextRing(Elem c);
  public static Elem prevRing(Elem c);
  public static Elem  sort(Elem tail);                        // see note 1
  public static Elem  merge(Elem s,Elem t,Elem tail);    // see note 2
}

class Ring2Iterator {
  public fromHead(Elem tail);
  public fromTail(Elem tail);
  public next();
}

NOTES:
 (1) In order to use this method, class Child must implement interface
     Comparable.compareTo(Object).
 (2) merge() can either merge two rings, or to split one:
    If s and t have different parents, p must be the parent of either s or t,
    and the two rings merge under p. The other parent is left without children.
    If s and t have the same parent, p must be a parent without children, and
    the original ring splits at s and t, with the ring containing t under p.
```



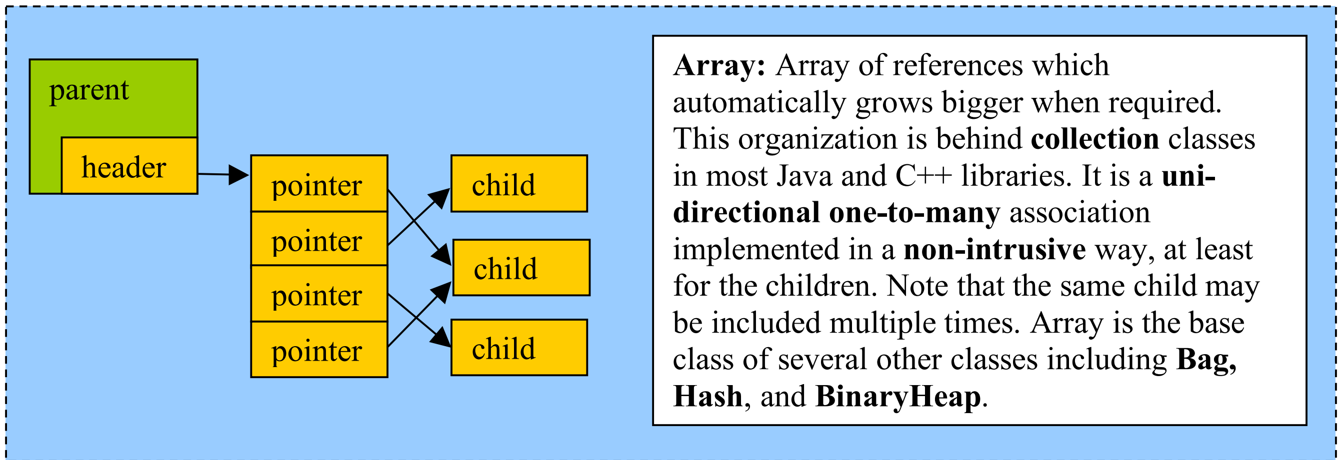

```
class Ring1 {
  // unless otherwise stated, methods return the new tail
  public static Elem addHead(Elem tail, Elem e);
  public static Elem  addTail(Elem tail, Elem e);
  public static Elem  append(Elem tail,Elem c1, Elem c2); // append c2 after c1
  public static Elem  remove(Elem tail,Elem c); // disconnect c, not destroy it
  public static Elem next(Elem tail,Elem c); // null if c is the tail
  public static Elem nextRing(Elem tail,Elem c);
  public static Elem  sort(Elem tail);                      // see note 1
  public static Elem  merge(Elem s,Elem t,Elem tail);    // see note 2
}

class Ring1Iterator {
    Elem fromHead(Elem tail);
    Elem next();
}

NOTES:
 (1) in order to use this method, class Child must implement interface
     Comparable.compareTo(Object).
 (2) merge() can either merge two rings, or to split one:
    If s and t have different parents, p must be the parent of either s or t,
    and the two rings merge under p. The other parent is left without children.
    If s and t have the same parent, p must be a parent without children, and
    the original ring splits at s and t, with the ring containing t under p.
```

DoubleLink: Mutual link between the source and target. This is **bi-directional one-to-one association.**

```
class DoubleLink {     // default association Bi1to1
  public static void add(Source s,Target t);
  public static void remove(Source s);
  public static Target next(Source s);
  public static Source prev(Target t);
}

// This class does not have any iterator
```

SingleLink: Single link from the source and target. This is **uni-directional one-to-one association.**

```
class SingleLink {     // default association Uni1to1
  public static void add(Source s,Target t);
  public static void remove(Source s);
  public static Target next(Source s);
}

// This class does not have any iterator
```



WARNINGS: This class and its interfaces are very similar to the C++ class Array from alib, but beware of the difference: The C++ version is an array of objects (not allowed in Java) and not an array of references. The C++ version has an array of references (LinkArray) which is identical with Java Array.

In order to simplify porting of programs coded with this library between Java and C++, the Java Array is also available under the name of **LinkArray** (array of references or *links*). LinkArrays match exactly between the two languages; Arrays match under normal circumstances. The difference to watch for is whether the C++ application code does not directly access the internal array of class Array. This is allowed to do but not recommended under C++, but it is impossible to do under Java. The reason for using this relatively-unsafe and hard-to-port technique would be the faster access and other operations on the array.

TERMINOLOGY:
For all arrays, the library uses the term **size** for the used part of the array, and the term **capacity** for the allocated size of the array. It always must be   size <= capacity.

class Array {   // also known as LinkArray

```
public static boolean form(Parent hp,int cap,int incr);        // see note 1
public static boolean formed(Parent hp); // checks whether the array is formed
public static boolean sizeChange(Parent hp,int newCap,boolean exact);
public static void extract(Parent hp, int k);                   // see note 2
public static void insert(Parent hp,int k,Child t);             // see note 3
public static void free(Parent hp);  // free the entire array
public static int capacity(Parent hp); // report current capacity
public static int size(Parent hp);     // report currently used size
public static int increment(Parent hp); // report the currently used increment
```

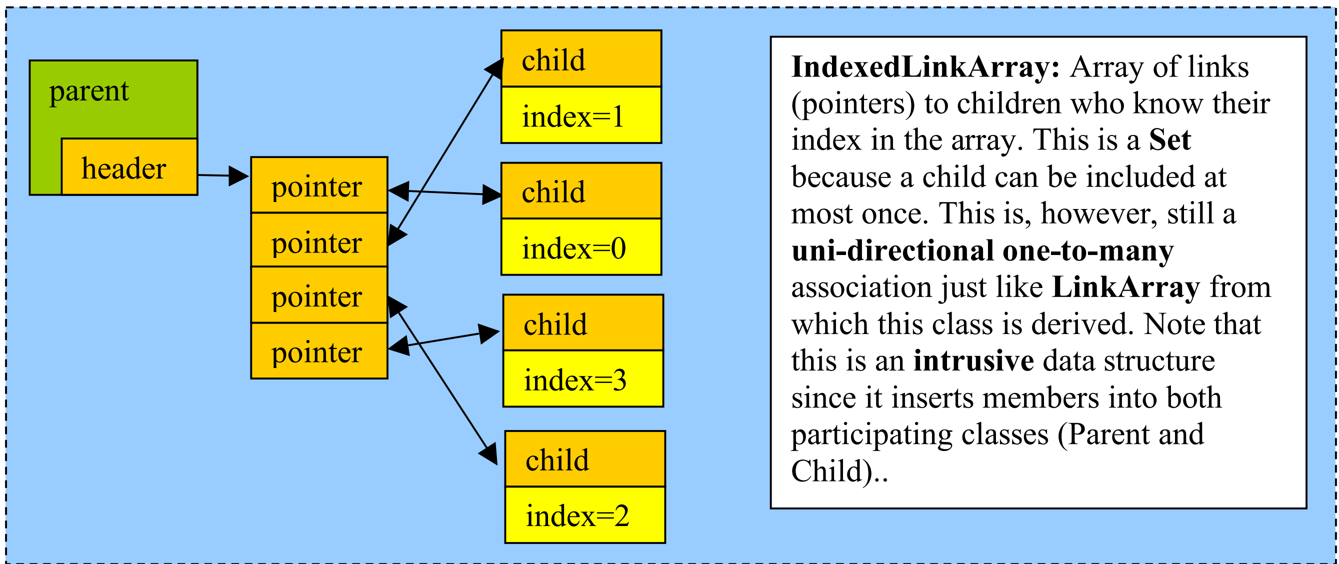
```
  public static Child get(Parent hp,int k); // a=array[k]
  public static void set(Parent hp,int k,Child a); // array[k]=a
  public static void remove(Parent hp,int k); // fast but order changed
  public static boolean reduce(Parent hp); // reduce the array to its used size
  public static boolean reduce(Parent hp,int newCap);            // see note 4
  public static boolean grow(Parent hp,int newCap); // grow capacity to newCap
  public static void reset(Parent hp,int newSz,int incr);
  public static void init(Parent hp);                            // note 7

  // the following methods assume that Child implements
  //              Comparable.compareTo(Object)
  static public void sort(Parent hp);
  static public void sortSubset(Parent hp,int i1,int i2);        // note 6
  public static void addOrd(Parent hp,Child op);
  public static void delOrd(Parent hp,Child obj);
  static public int binSearch(Parent hp,Child op);
}
```

There is no iterator, you traverse the array using integer index:

```
    Association Array<Parent,Child> array;

    int i,sz; Parent p; Child c;
    sz=array.size(p);
    for(i=0; i<sz; i++){
        c=array.get(p,i);
        ...
    }
```

NOTES:

(1) form() forms the array with the initial capacity cap. Value of incr controls how the array will grow: for incr>0 cap=cap+incr, for incr<0 cap=cap*(-incr), for incr=0 the array has a fixed capacity.
(2) extract() removes entry k and shrinks the array without changing the order, while remove() removes entry k and replaces it by they highest index entry. That is fast but changes the order of the array.
(3) Inserts the given object as entry k, and shifts the remaining part of the array.
(4) There are two forms of reduce(). This one forces reallocates the array to the given capacity (newSz) and throws away anything beyond it.
(5) binSearch() makes binary search and returns the index of the object with the same key as op or the first object with the higher key than that. In case of any error, e.g. when the array has not been formed yet, the function generates an error message and returns -1.
(6) Sort only the subset of the array between i1 and i2 inclusive.
(7) Initialize the array as empty with NULL at all entries. Typical use when reusing the array for something different, or after form() for improved reliability of the code.




```
class IndexedLinkArray {

    public static boolean form(Parent hp,int cap,int incr;            // note 1
    public static boolean formed(Parent hp);
    public static void free(Parent hp);
    public static int capacity(Parent hp);
    public static int increment(Parent hp;
    public static Child get(Parent hp,int k);                         // note 2
    public static void set(Parent hp,int k,Child a);                  // note 3
    public static void remove(Parent hp,int k);                       // note 4
    public static void insert(Parent hp,int k,Child t);
    public static boolean reduce(Parent hp);                          // note 5
    public static boolean reduce(Parent hp,int newCap);               // note 6
    public static boolean grow(Parent hp,int newCap);
    public static void sort(Parent hp);                               // note 7
    public static void sortSubset(Parent hpint i1,int i2);            // note 10
    public static void reset(Parent hp,int newSz,int incr);
    public static void addTail(Parent hp,Child a);
    public static int getIndex(Child e);                              // note 8
    public static int size(Parent hp);                                // note 9
    public static void init(Parent hp);                               // note 11
    public static void init(Child e);                                 // note 12

}
```

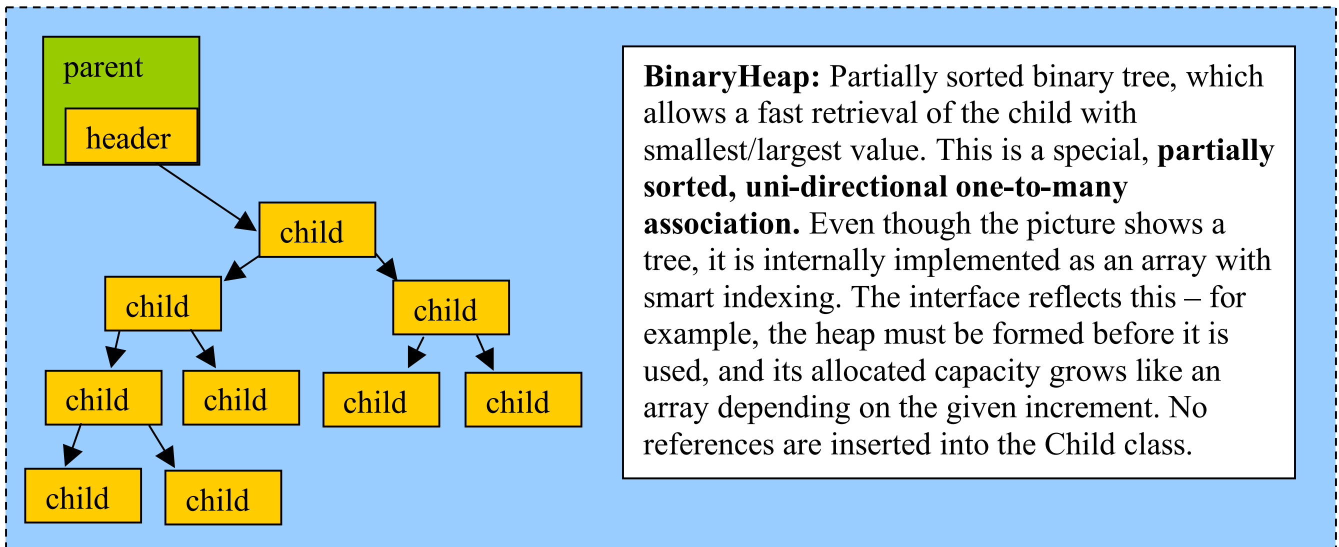
NOTES:

(1) Allocate and format array with the starting capacity cap and increment incr. Use negative incr when increasing by a multiple, incr=0 or -1 means a fix-sized array.

(2) This is the proper method to access an array entry as in

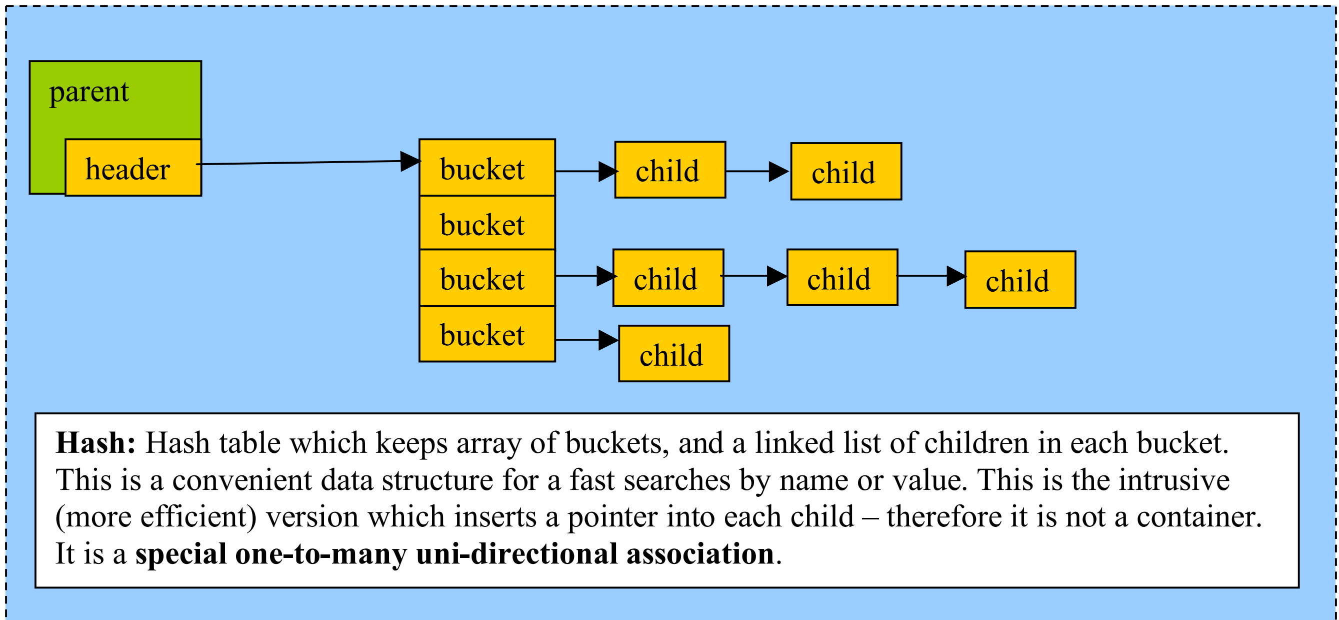
```
a=array[k]
```

(3) This is the proper method to set an array entry as in
    array[k]=a
(4) Fast method of removing an array entry, but the order of elements changes
    (the removed element is replaced by the last entry of the array).
(5) The capacity of the array is reduced to the currently used size.
(6) The capacity of the array is increased to newCap.
(7) The internal algorithm is qsort. The objects which are in the array must
    implement interface Comparable, for example see
    jlib/test/array_2/Employee.java.
(8) For a given Child, return its index in the array (returns -1 when the Child
    is not in the array).
(9) Return the actually used size of the array.
(10) Sort only the subset of the array between i1 and i2 inclusive.
(11) Initialize the array as empty with NULL at all entries. Typical use
    when reusing the array for something different, or after form() for
    improved reliability of the code.
(112 Additional initialization for each element is needed only if the array
    was previously used in a non-standard way.




```
When using BinaryHeap, class Element must implement interface
            Comparable.compareTo(Object).
Also, whenever an element changes its position in the heap,
its new position is reported by a call to Callback.callback(int).
This is handy in some advanced applications. You can bypass this reporting
by using null instead of callBck.

The internal implementation is based on this trick: If we number children row-by-
row 0,1,2,3,... then for entry i, i/2 is the index of its parent and
(2i+1),(2i+2) are indexes of its children.
```

```
class BinaryHeap {
  // Allocate heap for up to sz objects, using increment as for arrays
  public static Child form(Parent p, int sz,int incr);
  public static void free(Parent p); // Free (deallocate) the entire heap
  public static int size(Parent p);    // number of objects currently in the heap
  public static int capacity(Parent p); // currently allocated space for the heap

  // Insert e into the heap
  public static void inHeap(Parent h,Element e,Object callBck);

  // Return the top object and remove it from the heap.
  // Return 0 when the heap is empty.
  public static $2 outHeap(Parent h,Object callBck);

  // Re-sort the heap, assuming that the value for index n has changed.
  public static void updHeap(Parent h,int n,Object callBck);

  // Delete entry with index n, and re-sort the heap.
  public static $2 delHeap(Parent h,int n, Object callBck;
}
```



Hash: Hash table which keeps array of buckets, and a linked list of children in each bucket. This is a convenient data structure for a fast searches by name or value. This is the intrusive (more efficient) version which inserts a pointer into each child – therefore it is not a container. It is a **special one-to-many uni-directional association**.

```
When you use a hash table, you have to provide two following two methods for the
Child class (for examples, see the jlib tutorial or the jlib\test directory):
  public boolean id_equals(Child c);
which returns 'true' if the objects have the same key, and
  public int id_hash(int hashSz);
which converts key to a bucket number, assuming that hashSz is the number of
buckets. If you do not have your own favourite algorithm for this, use default
function id_Hash.hashString(String key,int hashSz)
or       id_Hash.hashInt(int key,int hashSz)
already provided with class Hash. For example:

EXAMPLES:
    class Child {
        private int key;
        private String strKey;
        public ZZ_Child ZZds;

        public boolean intHash_equals(Child c){return key==c.key;}
        public int intHash_hash(int hashSz){return intHash.hashInt(key,hashSz);}

        public boolean strHash_equals(Child c){return strKey.equals(c.strKey);}
        public int strHash_hash(int hashSz){
           return strHash.hashString(strKey,hashSz);
        }
    }

    // unusual situation: two hash tables, one using key the other keyStr
    Association Hash<Parent,Child> intHash;
    Association Hash<Parent,Child> strHash;
    Association Name<Child> cName;
```

Note that class Hash has an iterator which allows to traverse either a selected bucket or the entire hash table:

```
Child d; Parent p;
intHash_Iterator it;
for(c=it.first(p,i); c!=null; c=it.next()){...}
// where i is the bucket index, -1 will traverse the entire hash table
```

Since, internally, this class is derived from Array, its interface has a similar style. The hash table must be first formed in order to be used. However, the table does not re-allocate itself, and it is left to the user to monitor its loading and increase its size when needed. This usually does not take much code and results in a more intelligent and efficient code.

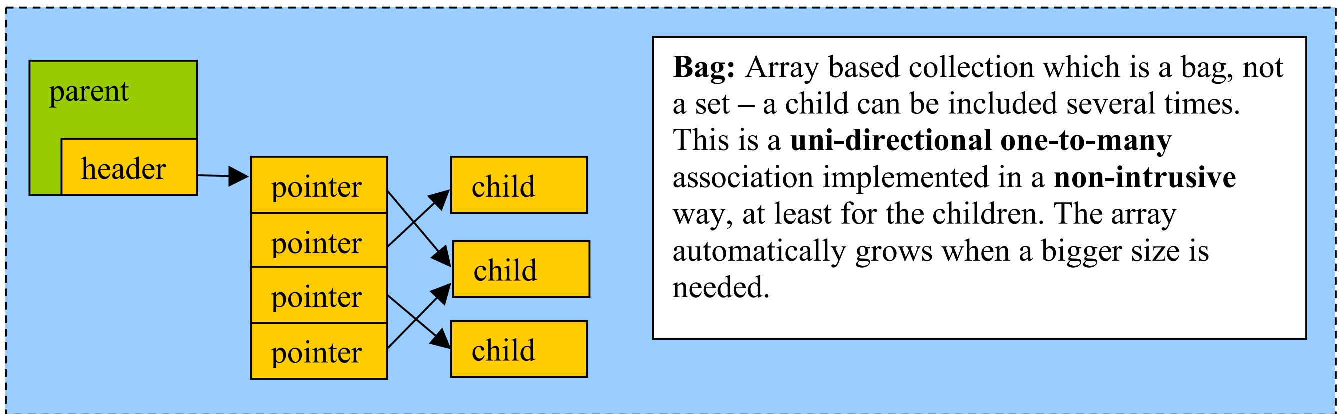
class Hash {

```
  public static void form(Holder h,int sz); // sz=number of buckets
  public static int formed(Holder ph   // number of buckets
  public static int popCount(Holder h); // count of objects in the hash table
  public static void free(Holder h); // free (deallocate) the entire hash table
  public static int resize(Holder h, int newSz); // resize to newSz buckets
  public static Child get(Holder h, Element e);    // see note 1
  public static int add(Holder h, Element e); // load e into the hash table
  public static Child remove(Holder h, Element e); // see note 2

  // convenient default functions to use when coding hash()
  // --------------------------------------------------
  public static int hashString(String val,int hashSz);
  public static int hashInt(int val,int hashSz);
}

class HashIterator {
    public Child first(Parent p,int i);  // traverse bucket i, i<0 entire table
    public Child next();
}
```

NOTES:

(1) get() searches the table and returns the object with the same key as the auxilliary object e. It returns null if such object is not found.

(2) remove() removes e from the hash table. If e is not in the table, it removes the object with the key identical to the e.




```
Bag {       // default association Uni1toX
  public static void form(Parent hp,int initCap,int incr); // see Notes 1,2
  public static int count(Parent hp);                       // see Note 3
  public static void free(Parent hp);                       // see Note 4
  public static void add(Parent hp, Child obj);             // see Note 5
  public static void remove(Parent hp, Child obj);          // see Note 6
}
class BagIterator {     // see Note 7
    public BagIterator();
    public Child first(Parent hp);
    public Child next();
};
```

NOTES:

(1) form() works exactly as the Array::form(), you provide the initial array size and increment which is applied when the array size must be increased. incr>0 implies an arithmentic increment, incr<0 implies an multiplicative increment. incr=0 or incr=1 defines a fixed-size array (its size cannot be changed).

(2) If you don't call formed(), the first call to add() automatically forms a default-sized array (8 entries, increment -3 which means 3x).

(3) count() returns the current number of links (number of array entries), not the number of children. When the collection is empty or not formed yet, this function returns 0.

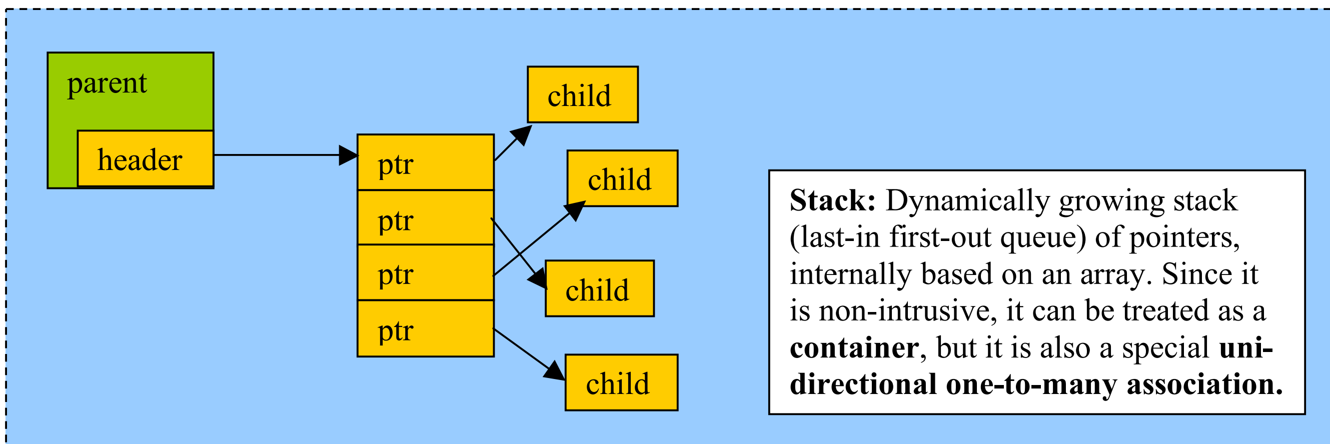
(4) The array is discarded but the children are not touched.

(5) add() can be called during an iteration loop without disturbing the expected behaviour (the new items are added at the end.

(6) remove() removes all(!) links to the given child. This is an expensive function because it traverses the entire collection. The order of the remaining children is not changed. Do not call remove() while running the iterator - the iterator may skip some children.

(7) Except for the behaviour listed under (5) and (6), the iterator works exactly like the iterators for Aggregate, LinkedList or Rings. For example:

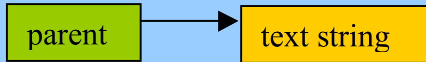
```
class Student;
class Course;
Association Bag<Student,Course> takes;
...
takes_Iterator it;
Student s; Course c;
...
for(c=it.first(s); c!=null; c=it.next()){...}
```




```
class Stack {
  public static Child form(Parent p, int sz,int incr); // note 1
  public static void push(Parent p,Child c);
  public static Child pop(Parent p);
  public static void free(Parent p); // free and deallocate the entire stack
  public static int size(Parent p);      // returns size of the stack
  public static int capacity(Parent p);  // returns allocated size

};
```

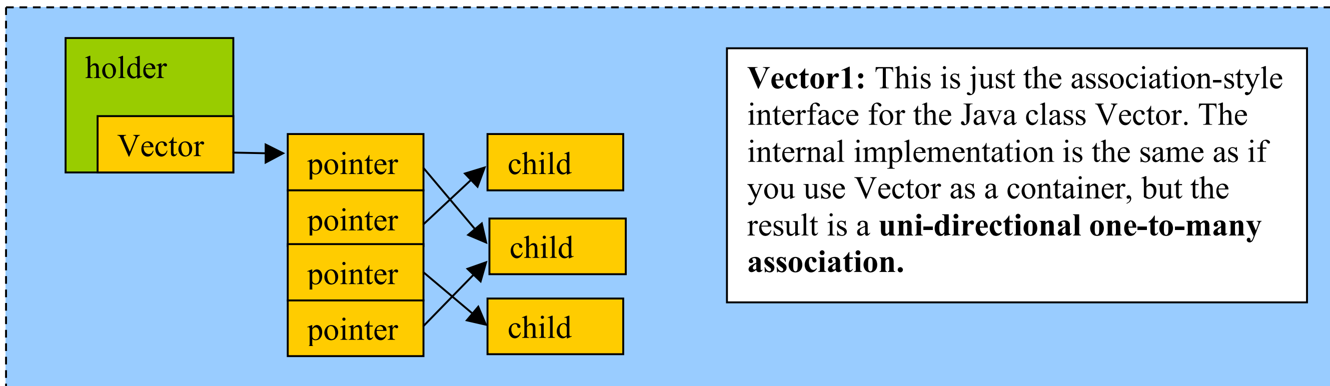
NOTES:

(1) Form the stack as you would form an array: sz=initial size, with the dynamic grow: for incr>0 sz=sz+incr, for incr<0 sz=sz*(-incr), for incr=0 the size is fixed.



Name: Text string attached via a reference. Since the alib strategy is to remove explicit pointers from all objects, we prefer this arrangement to the parent having a String member. This is a convenient data structure, neither a container nor an association.

```
class Name {
public:
  public static void add(Parent p, String c);
  public static String get(Parent p);
  public static char remove(Parent p);
  public static int compare(Parent p1, Parent p2);
};

NOTE: DataStructure Name<MyClass> mName;
      is essentially the same thing as using
      class MyClass {
          String mName;
          ...
      }
      Many programmers do not realize that the string is not inserted directly
      but only as a reference.
```



Classes Vector1 and Vector2 are an example how you can reuse container classes from any existing library and turn them into associations. Except for the interface, the Java class Vector is similar to our class Array, and is typically used as a member of the parent object:

```
class Department {
    Vector empl;
    ...
}

Department d; Employee e;
e=(Employee)(d->empl.at(7));
```

Note that when you read the last line, you think like is: *Go to Department d, get its vector empl, call its function at() with 7, and interpret the result as Employee.*

Class Vector1 does the same thing, but with a different interface:

```
class Department {
    ...
}

DataStructure Vector1<Department,Employee> empl;

Department d; Employee e;
e=empl.at(d,7);
```

Here you think with the emphasis on the data organization: *Go to the vector empl on Department d, and get Employee at position 7.* Note that you do not have to interpret (cast) the result, all the methods are type protected.

NOTES:
- Class Vector1 has the equivalent of all the methods of Java Vector.
- Since methods of Vector1 are static, various forms of method form() take on the role of the constructors.
- Method Vector.removeRange(int fromIndex,int toIndex) is protected and therefore has not been included in Vector1.

WARNING: Vector1 was obtained by a mechanical API conversion and has been only partially tested.

```
class Vector1 {

  // Construct an empty vector so that its internal data array has
  //  size 10 and its standard capacity increment is zero.
  // -----------------------------------------------------------------
  static public void form(Holder p);

  // Construct a vector containing the elements of the specified collection,
  // in the order they are returned by the collection's iterator.
  // -----------------------------------------------------------------
  static public void form(Holder p, Collection c);

  // Construct an empty vector with the specified initial capacity and
  // with its capacity increment equal to zero.
  // -----------------------------------------------------------------
  static public void form(Holder p, int initialCapacity);

  // Construct an empty vector with the specified initial capacity and
  // capacity increment.
  // -----------------------------------------------------------------
  static public void form(Holder p, int initialCapacity, int capacityIncrement);

  // Insert the specified element at the specified position in this Vector.
  // -----------------------------------------------------------------
  static public void add(Holder p, int index, Object element);

  // Append the specified element to the end of this Vector.
  // -----------------------------------------------------------------
  static public boolean add(Holder p, Object o);

  // Appends all of the elements in the specified Collection to the end
  // of this Vector, in the order that they are returned by the specified
  // Collection's Iterator.
  // -----------------------------------------------------------------
  static public boolean addAll(Holder p, Collection c);

  // Insert all of the elements in in the specified Collection into
  // this Vector at the specified position.
  // -----------------------------------------------------------------
  static public boolean addAll(Holder p, int index, Collection c);

  // Add the specified component to the end of this vector, increasing
  // its size by one.
  // -----------------------------------------------------------------
  static public void addElement(Holder p, Object obj);

  // Return the current capacity of this vector.
  // -----------------------------------------------------------------
  static public int capacity(Holder p);
```

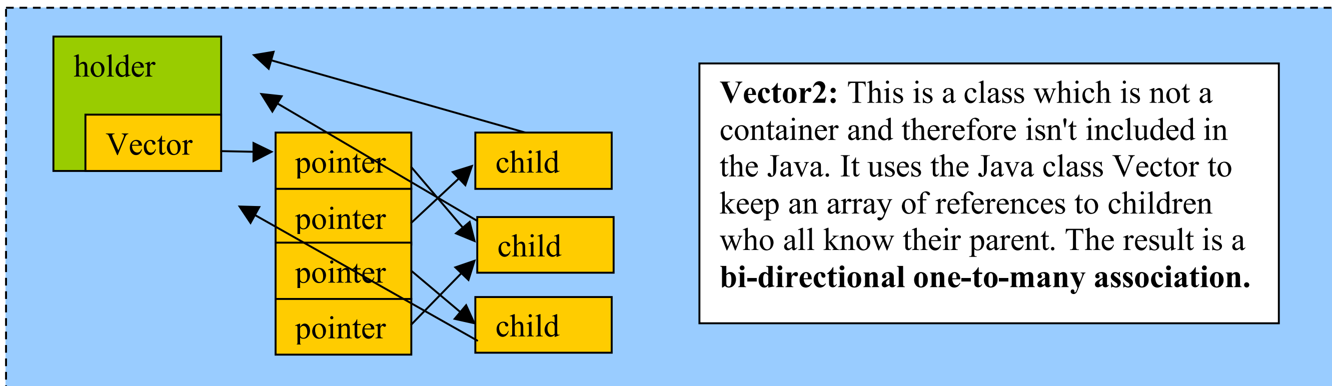
```
// Remove all of the elements from this Vector.
// ----------------------------------------------------------------
static public void clear(Holder p);

// Return a clone of this vector.
// ----------------------------------------------------------------
static public Object clone(Holder p);

// Test if the specified object is a component in this vector.
// ----------------------------------------------------------------
static public boolean contains(Holder p, Object elem);

// Return true if this Vector contains all of the elements
// in the specified Collection.
// ----------------------------------------------------------------
static public boolean containsAll(Holder p, Collection c);

// Copy the components of this vector into the specified array.
// ----------------------------------------------------------------
static public void copyInto(Holder p, Object[] anArray);

// Return the component at the specified index.
// ----------------------------------------------------------------
static public Object elementAt(Holder p, int index);

// Return an enumeration of the components of this vector.
// ----------------------------------------------------------------
static public Enumeration elements(Holder p);

// Increase the capacity of this vector, if necessary, to ensure that
// it can hold at least the number of components specified by
// the minimum capacity argument.
// ----------------------------------------------------------------
static public void ensureCapacity(Holder p, int minCapacity);

// Compare the specified Object with this Vector for equality.
// ----------------------------------------------------------------
static public boolean equals(Holder p, Object o);

// Return the first component (the item at index 0) of this vector.
// ----------------------------------------------------------------
static public Object firstElement(Holder p);

// Return the element at the specified position in this Vector.
// ----------------------------------------------------------------
static public Object get(Holder p, int index);

// Return the hash code value for this Vector.
// ----------------------------------------------------------------
static public int hashCode(Holder p);
```

```
// Search for the first occurence of the given argument, testing
// for equality using the equals() method.
// ------------------------------------------------------------------
static public int indexOf(Holder p, Object elem);

// Search for the first occurence of the given argument, beginning
// the search at index, and testing for equality using the equals() method.
// ------------------------------------------------------------------
static public int indexOf(Holder p, Object elem, int index);

// Insert the specified object as a component in this vector
// at the specified index.
// ------------------------------------------------------------------
static public void insertElementAt(Holder p, Object obj, int index);

// Test whether this vector has no components.
// ------------------------------------------------------------------
static public boolean isEmpty(Holder p);

// Return the last component of the vector.
// ------------------------------------------------------------------
static public Object lastElement(Holder p);

// Return the index of the last occurrence of the specified object
// in this vector.
// ------------------------------------------------------------------
static public int lastIndexOf(Holder p, Object elem);

// Search backwards for the specified object, starting from
// the specified index, and returns an index to it.
// ------------------------------------------------------------------
static public int lastIndexOf(Holder p, Object elem, int index);

// Remove the element at the specified position in this Vector.
// ------------------------------------------------------------------
static public Object remove(Holder p, int index);

// Remove the first occurrence of the specified element in this Vector
// ------------------------------------------------------------------
static public boolean remove(Holder p, Object o);

// Remove from this Vector all of its elements that are contained
// in the specified Collection.
// ------------------------------------------------------------------
static public boolean removeAll(Holder p, Collection c);

// Remove all components from this vector and sets its size to zero.
// ------------------------------------------------------------------
static public void removeAllElements(Holder p);

// Remove the first (lowest-indexed) occurrence of the argument
// from this vector.
```

```
// -----------------------------------------------------------------
static public boolean removeElement(Holder p, Object obj);

// Delete the component at the specified index.
// -----------------------------------------------------------------
static public void removeElementAt(Holder p, int index);

// Retain only the elements in this Vector that are contained
// in the specified Collection.
// -----------------------------------------------------------------
static public boolean retainAll(Holder p, Collection c);

// Replace the element at the specified position in this Vector with
// the specified element.
// -----------------------------------------------------------------
static public Object set(Holder p, int index, Object element);

// Set the component at the specified index of this vector to be the
// specified object.
// -----------------------------------------------------------------
static public void setElementAt(Holder p, Object obj, int index);

// Set the size of this vector.
// -----------------------------------------------------------------
static public void setSize(Holder p, int newSize);

// Return the number of components in this vector.
// -----------------------------------------------------------------
static public int size(Holder p);

// Return a view of the portion of this List between fromIndex,
// inclusive, and toIndex, exclusive.
// -----------------------------------------------------------------
static public List subList(Holder p, int fromIndex, int toIndex);

// Return an array containing all of the elements in this Vector
// in the correct order.
// -----------------------------------------------------------------
static public Object[] toArray(Holder p);

// Return an array containing all of the elements in this Vector
// in the correct order; the runtime type of the returned array is
// that of the specified array.
// -----------------------------------------------------------------
static public Object[] toArray(Holder p, Object[] a);

// Return a string representation of this Vector, containing
// the String representation of each element.
// -----------------------------------------------------------------
static public String toString(Holder p);

// Trim the capacity of this vector to be the vector's current size.
```

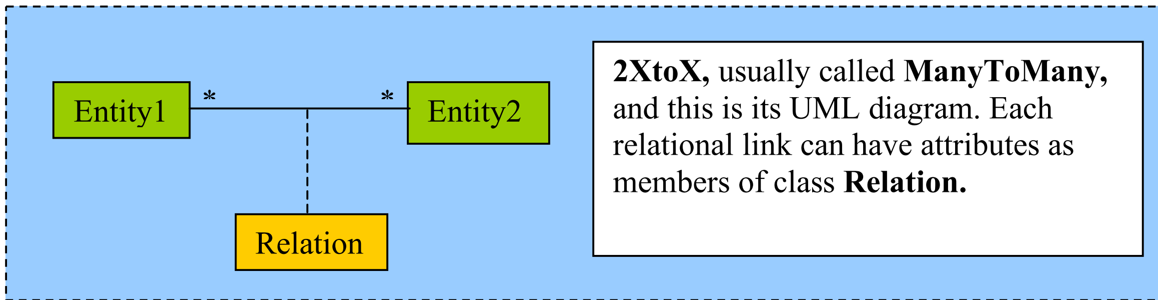
```
  // -------------------------------------------------------------
  static public void trimToSize(Holder p);
}
```

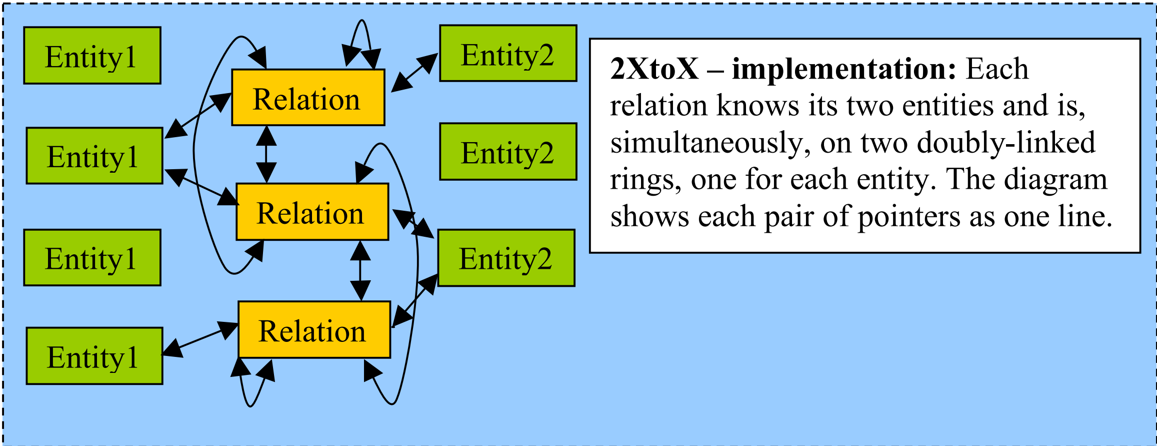
You may have noticed that Java Vector and Vector1 have many more methods than jlib Array and other jlib classes. This is a matter of design philosophy. Personally, I prefer data structures with a smaller number of methods which are easy to remember. Functionality of all Vector1 methods can be implemented with one or a few method calls from Array.




```
class Vector2 {
    public static Holder *getParent(Child c);
    // .. all other methods of Java Vector
}
```

NOTE: Internally, Vector2 is derived from Vector1, with all of its methods remaining without any change.






```
class 2XtoX {      // ManyToMany among 2 entities

    static void add(Relation r, Entity1 e1,Entity2 e2);
    static void remove(Relation r);
    static Entity1 entity1(Relation r);
    static Relation next1(Relation r);
    static Relation prev1(Relation r);
    static Entity2 entity2(Relation r);
    static Relation next2(Relation r);
    static Relation prev2(Relation r);
}

class 2XtoX_Iterator {
    // standard interface:      for(r=it.from1(e1); r!=null; r=it.next1()){...}
    // standard interface:      for(r=it.from2(e2); r!=null; r=it.next2()){...}

    Relation from1(Entity1 e);
    Relation next1();
    Relation from2(Entity2 e);
    Relation next2();
}
```

NOTES:

(1) Method add() adds the next relation as the tail on either ring, so the relations keep the order in which they were inserted.

(2) Remove is fast (it just re-links a few references), and it does not change the order of either list.

(2) Even though the lists are internally implemented as rings, when the relation is the tail, next1() or next2() return null. Similarly, when the relation is the head of the list, prev1() or prev2() return null.

(4) The iterator allows to traverse independently both rings. Use the appropriate pairs (from1 – next1) or (from2 – next2). Other combinations are not permitted.

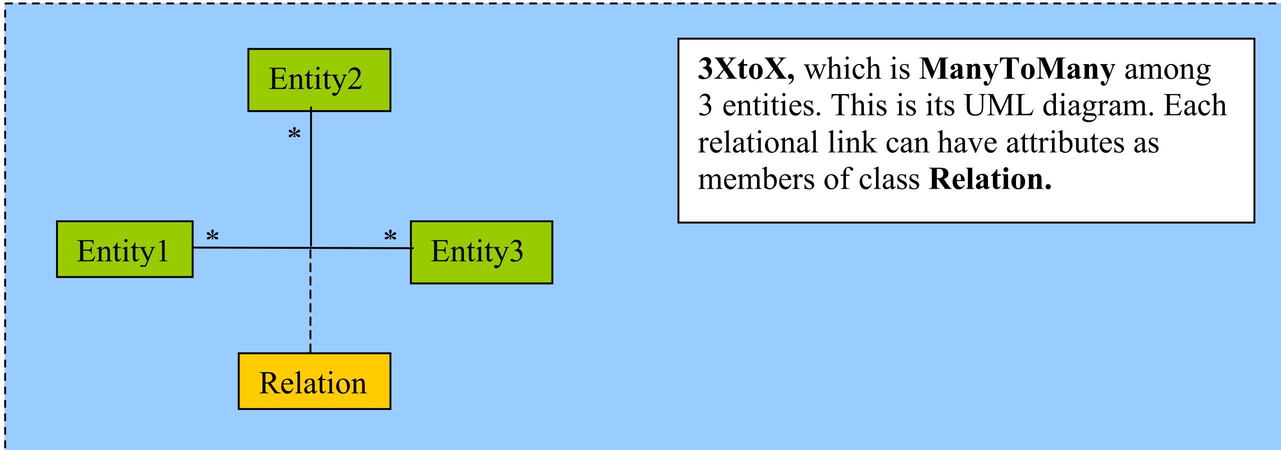
```
2XtoX_Iterator it=new 2ZtoX_Iterator();;
for(r=it.from1(e1); r!=null; r=it.next1()){...}
for(r=it.from2(e2); r!=null; r=it.next2()){...}
```

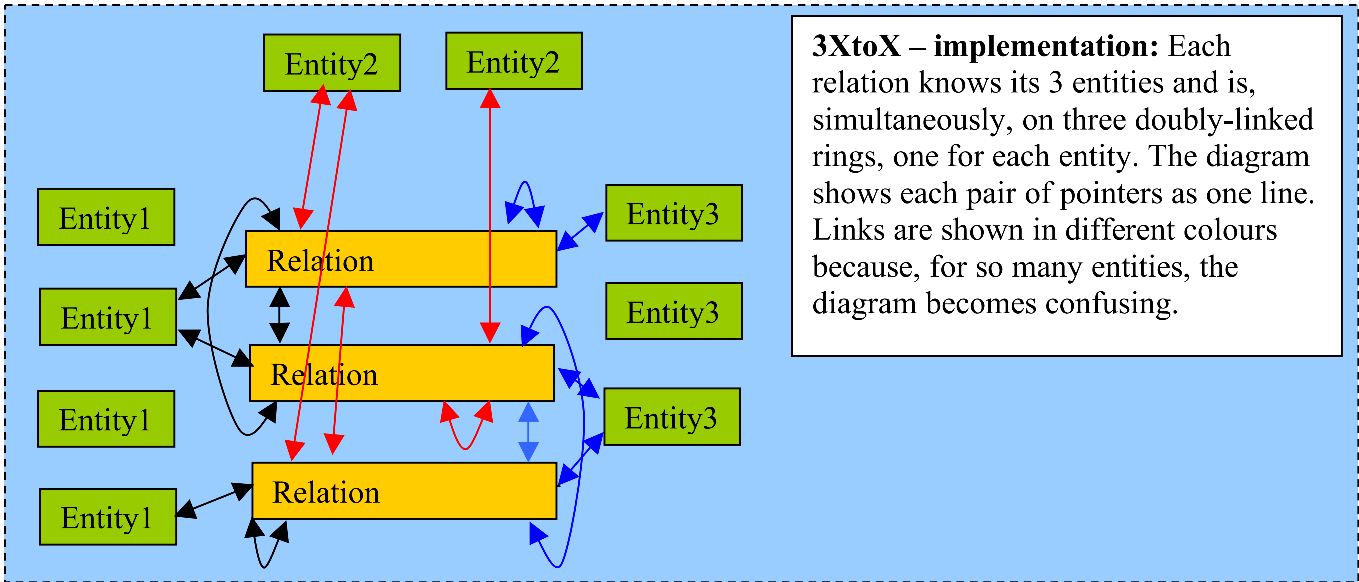
(5) The use of the same class for both the source and target is currently not allowed, e.g. `Association 2XtoX<R,E,E> xxx;`

If you use this construct, the compiler will give you error message

```
..\lib\xxx_2XtoXEntity12XtoXEntity2.jt (The system cannot find
the file specified)
```

Note that this construct really defines a graph, and a special class will be introduced for this purpose.






```
class 3XtoX {      // ManyToMany among 3 entities

    static void add(Relation r, Entity1 e1,Entity2 e2, Entity3 e3);
    static void remove(Relation r);
    static Entity1 entity1(Relation r);
    static Relation next1(Relation r);
    static Relation prev1(Relation r);
    static Entity2 entity2(Relation r);
    static Relation next2(Relation r);
    static Relation prev2(Relation r);
    static Entity3 entity3(Relation r);
    static Relation next3(Relation r);
    static Relation prev3(Relation r);
}
```

```
class 3XtoX_Iterator {
    // standard interface:      for(r=it.from1(e1); r!=null; r=it.next1()){...}
    // standard interface:      for(r=it.from2(e2); r!=null; r=it.next2()){...}
    // standard interface:      for(r=it.from3(e3); r!=null; r=it.next3()){...}

    Relation from1(Entity1 e);
    Relation next1();
    Relation from2(Entity2 e);
    Relation next2();
    Relation from3(Entity3 e);
    Relation next3();

}
```

NOTES:
(1) This class has identical interface with 2XtoX, except for the additional (and equivalent) methods related to the third Entity.
(2) Method add() adds the next relation as the tail on all three rings, so the relations keep the order in which they were inserted.
(3) Remove is fast (it just re-links a few references) and it does not change the order of either list.
(4) Even though the lists are internally implemented as rings, when the relation is the tail, next1(), next2() or next3() return null. Similarly, when the relation is the head of the list, prev1(), prev2() or prev3() return null.
(5) The iterator now allows to traverse independently the three rings. Use the appropriate pairs of *from* and *next*. Other combinations are not permitted.

```
            3XtoX_Iterator it=new 3XtoX_Iterator();
            for(r=it.from1(e1); r!=null; r=it.next1()){...}
            for(r=it.from2(e2); r!=null; r=it.next2()){...}
            for(r=it.from3(e3); r!=null; r=it.next3()){...}
```

(6) The implementation of both 2XtoX and 3XtoX provides examples of how ManyToMany relation of order n can be easily expanded to order (n+1).