Structures de Données, Collections et généricité (Java)

Les structures de données acquises

Les Collections 1.2

Les apports 1.5 : la généricité

Conserver les acquis !

- Java 1.0 en 1995
- Java 1.4 en 2004
- Java 1.5 en 2005
- Java 1.6 en 2006

- Une même machine virtuelle
- Une compatibilité du code généré

- Des conséquences sur les choix d'aujourd'hui !

Les structures de données en héritage

- Array

- Vector
- Stack
- Hashtable

En héritage : Les tableaux

- Object[] to;
- String[] ts;
- ts = new String[10];
- to=ts;
- String s = (String) to[0]; // contrôle de type à l'exécution

- Si C1 est une sous-classe de C0
- C1[] est un sous type de C0[]
- On appelle cela 'covariance' des tableaux

Les difficultés de ce choix

```
class T {

   //@ requires a != null && 0 <= i & i < a.length;

  static void storeObject(Object[] t, int i, Object x) {
          t[i] = x; //  ArrayStoreException possible
}
  ...
}
```

//@ requires typeof(x) <: typeof(t);

java.util.Vector

- Cette classe gère une collection d'objets dans un tableau dynamique ou liste chaînée

  - **Vector** v = new **Vector**();

  - **v.addElement**("une chaine");
  - **v.addElement**(new date());
  - **v.addElement**(new String[]);
  - **v.addElement**(new Vector());

  - **v.setElementAt**("abcde", 2);
  - System.out.println(**v.elementAt**(2)); // --> abcde

java.util.Hashtable

- Cette classe gère une collection d'objets au travers d'une table de hachage dont les clés sont des String et les valeurs associées des Object.

```
– Hashtable ht = new Hashtable();

– ht.put("noel", new Date("25 Dec 1997"));

– ht.put("un vecteur", new Vector());

– Vector v = (Vector)ht.get("un vecteur");

– for(Enumeration e = ht.keys(); e.hasMoreElements();){
– String key = (String)e.nextElement;
–    ...
– }
```

Collections 1.2: Motivations

Les applications manipulent de nombreuses données en mémoire
Ces programmes doivent les conserver, les retrouver, les ordonner,…
Les tableaux existent déjà mais sont de « bas niveau »
Par analogie avec les mathématiques on peut souhaiter disposer

- d'ensembles d'objets (multi-set ou non)
- de n-uplets d'objets (listes)
- de relations d'ordre sur ces objets
- de listes ordonnées
- de fonctions qui a un objet fassent correspondre un autre

On peut vouloir considérer toutes ces organisations de données comme des Collections d'objets
On peut également vouloir itérer sur ces collections élément après élément sans tenir compte de l'organisation particulière
On doit pouvoir disposer de moyens de passer des tableaux à ces structures. Il nous faut aussi pouvoir choisir différentes stratégies pour ordonner les structures (sort).

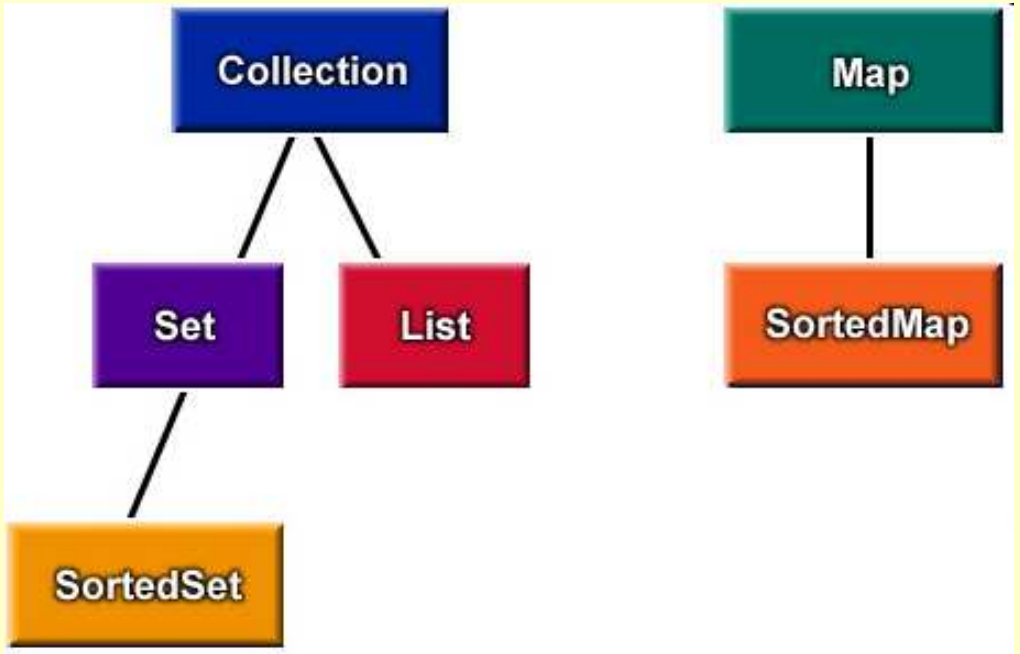
java.util

java

util

zip

jar



Collections et Maps

Collection: **un groupe d'éléments**

Set: **un ensemble d'éléments (sans duplications)**

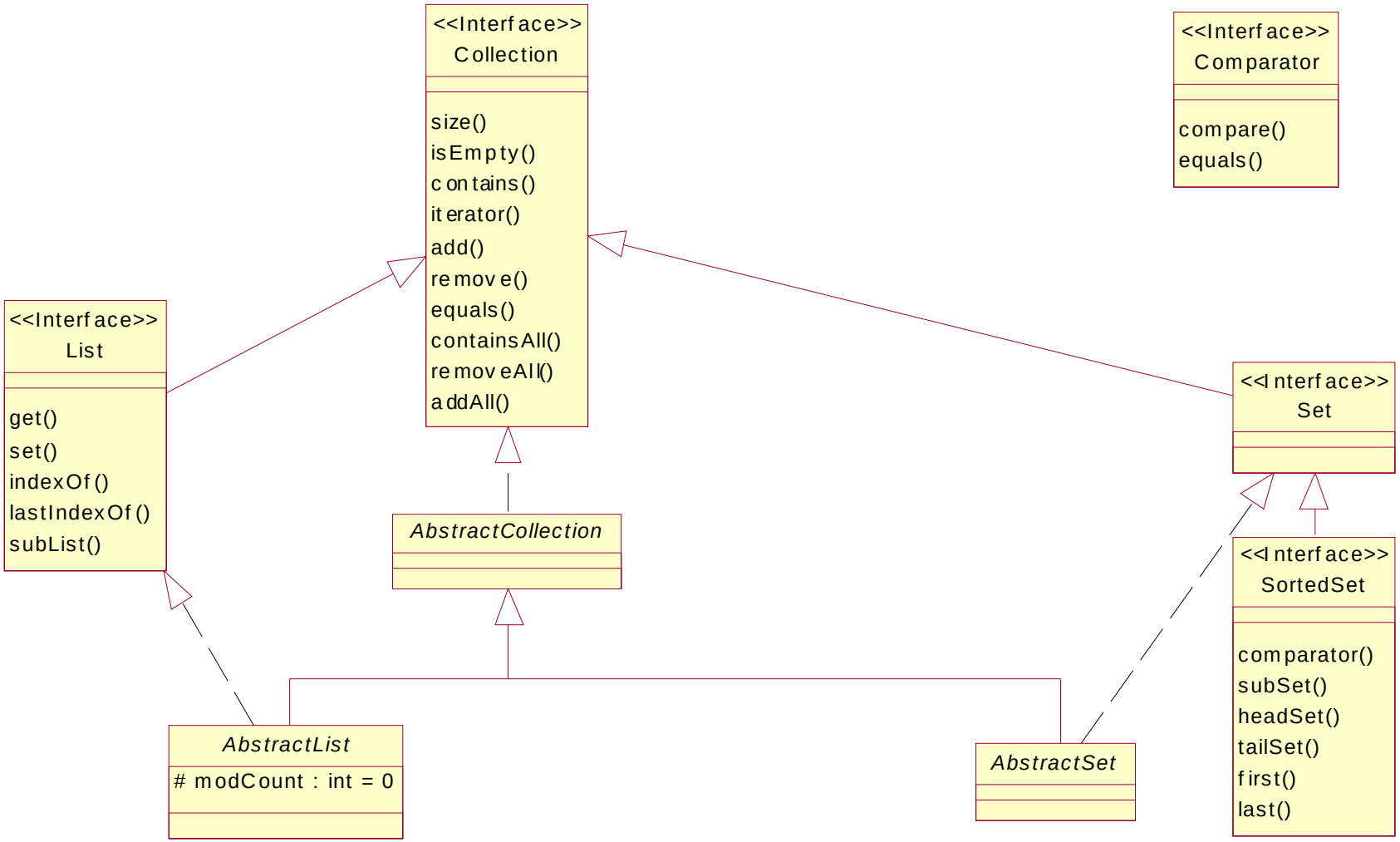
SortedSet: **ensemble ordonné**

List: **séquence d'éléments**

Map: **fonction des Clés vers les valeurs (pas de clés en doubles)**

SortedMap: **Comme Map mais avec clés ordonnées**

Graphe des interfaces



Interface Collection

```
public interface Collection {
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element);                    // Optional
    boolean remove(Object element);     // Optional
    Iterator iterator();
    boolean containsAll(Collection c);
    boolean addAll(Collection c);                   // Optional
    boolean removeAll(Collection c);                // Optional
    boolean retainAll(Collection c);                // Optional
    void clear();                               // Optional
    // Array Operations
    Object[] toArray();
    Object[] toArray(Object a[]);
}
```

Interface Iterator

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

Filtrer une collection

```
static void filter(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); ){
        if (!cond(i.next())) i.remove();
    }
}
```

Interface Set

```
public interface Set {
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element);                    // Optional
    boolean remove(Object element);                 // Optional
    Iterator iterator();

    boolean containsAll(Collection c);
    boolean addAll(Collection c);                   // Optional
    boolean removeAll(Collection c);                // Optional
    boolean retainAll(Collection c);                // Optional
    void clear();                                   // Optional

    // Array Operations
    Object[] toArray();
    Object[] toArray(Object a[]);
}
//java.lang.UnsupportedOperationException
```

Exemples

```
Set union = new HashSet(s1);
union.addAll(s2);

Set intersection = new HashSet(s1);
intersection.retainAll(s2);

Set difference = new HashSet(s1);
difference.removeAll(s2);
```

Interface List

```
public interface List extends Collection {
   // Positional Access
   Object get(int index);
   Object set(int index, Object element);                    // Optional
   void add(int index, Object element);                      // Optional
   Object remove(int index);                                  // Optional
   abstract boolean addAll(int index, Collection c);         // Optional

   // Search
   int indexOf(Object o);
   int lastIndexOf(Object o);

   // Iteration
   ListIterator listIterator();
   ListIterator listIterator(int index);

   // Range-view
   List subList(int from, int to);
}
```

Exemple

```
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);
}// indépendant du type de liste
```

interface ListIterator

```
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
}
```

Exemple

```
for (ListIterator i=l.listIterator(l.size())  ;  i.hasPrevious()  ; ) {

        Foo f = (Foo) i.previous();

        ...

}
```

Interface Map

```
public interface Map {
   // Basic Operations
   Object put(Object key, Object value);
   Object get(Object key);
   Object remove(Object key);
   boolean containsKey(Object key);
   boolean containsValue(Object value);
   int size();
   boolean isEmpty();

   // Bulk Operations
   void putAll(Map t);
   void clear();

   // Collection Views
   public Set keySet();
   public Collection values();
   public Set entrySet();

   // Interface for entrySet elements
   public interface Entry {
      Object getKey();
      Object getValue();
      Object setValue(Object value);
   }
}
```

Itérer sur une Map

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

The idiom for iterating over values is analogous. Here's the idiom for iterating over key-value pairs:

```
for (Iterator i=m.entrySet().iterator(); i.hasNext(); ) {
        Map.Entry e = (Map.Entry) i.next();
        System.out.println(e.getKey() + ": " + e.getValue());
}
```

Relation d'ordre

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

Exemple

```
import java.util.*;
public class Name implements Comparable {
   private String  firstName, lastName;

   public Name(String firstName, String lastName) {
      if (firstName==null || lastName==null)      throw new NullPointerException();
      this.firstName = firstName;
      this.lastName = lastName;
   }
   public String firstName()    {return firstName;}
   public String lastName()     {return lastName;}

   public boolean equals(Object o) {
      ….
   }
   …
public int compareTo(Object o) {
      Name n = (Name)o;
      int lastCmp = lastName.compareTo(n.lastName);
      return (lastCmp!=0 ? lastCmp :
            firstName.compareTo(n.firstName));
   }
}
```

Interface Comparator

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

Interface Queue

```
public interface Queue<E> extends Collection<E> {
E element();
boolean offer(E e);
E peek();
E poll();
E remove();
}
```

SortedSet

```
public interface SortedSet extends Set {
    // Range-view
    SortedSet subSet(Object fromElement, Object toElement);
    SortedSet headSet(Object toElement);
    SortedSet tailSet(Object fromElement);

    Object first();
    Object last();

     Comparator comparator();
}
```

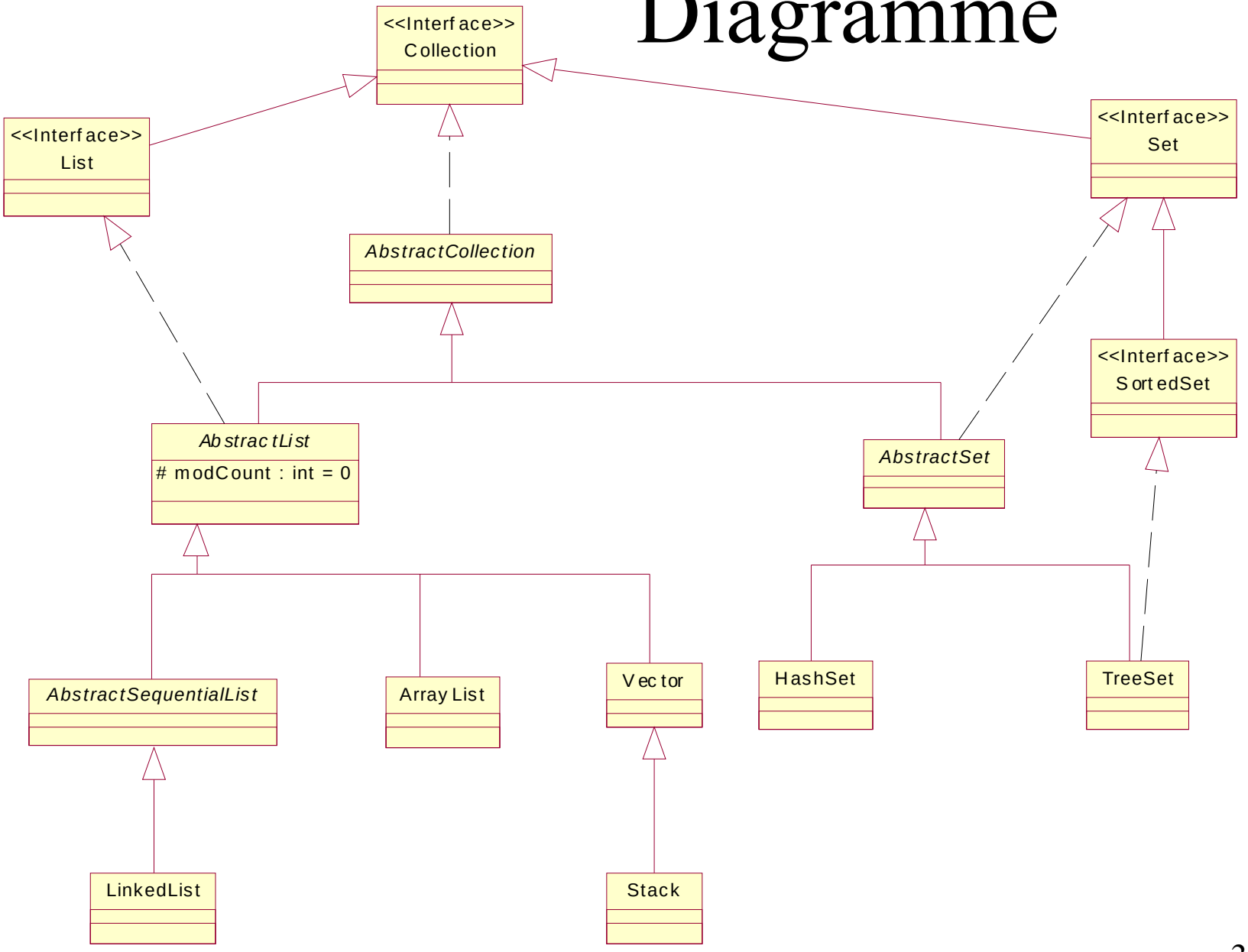
SortedMap

```
public interface SortedMap extends Map {
    Comparator comparator();

    SortedMap subMap(Object fromKey, Object toKey);
    SortedMap headMap(Object toKey);
    SortedMap tailMap(Object fromKey);

    Object first();
    Object last();
}
```

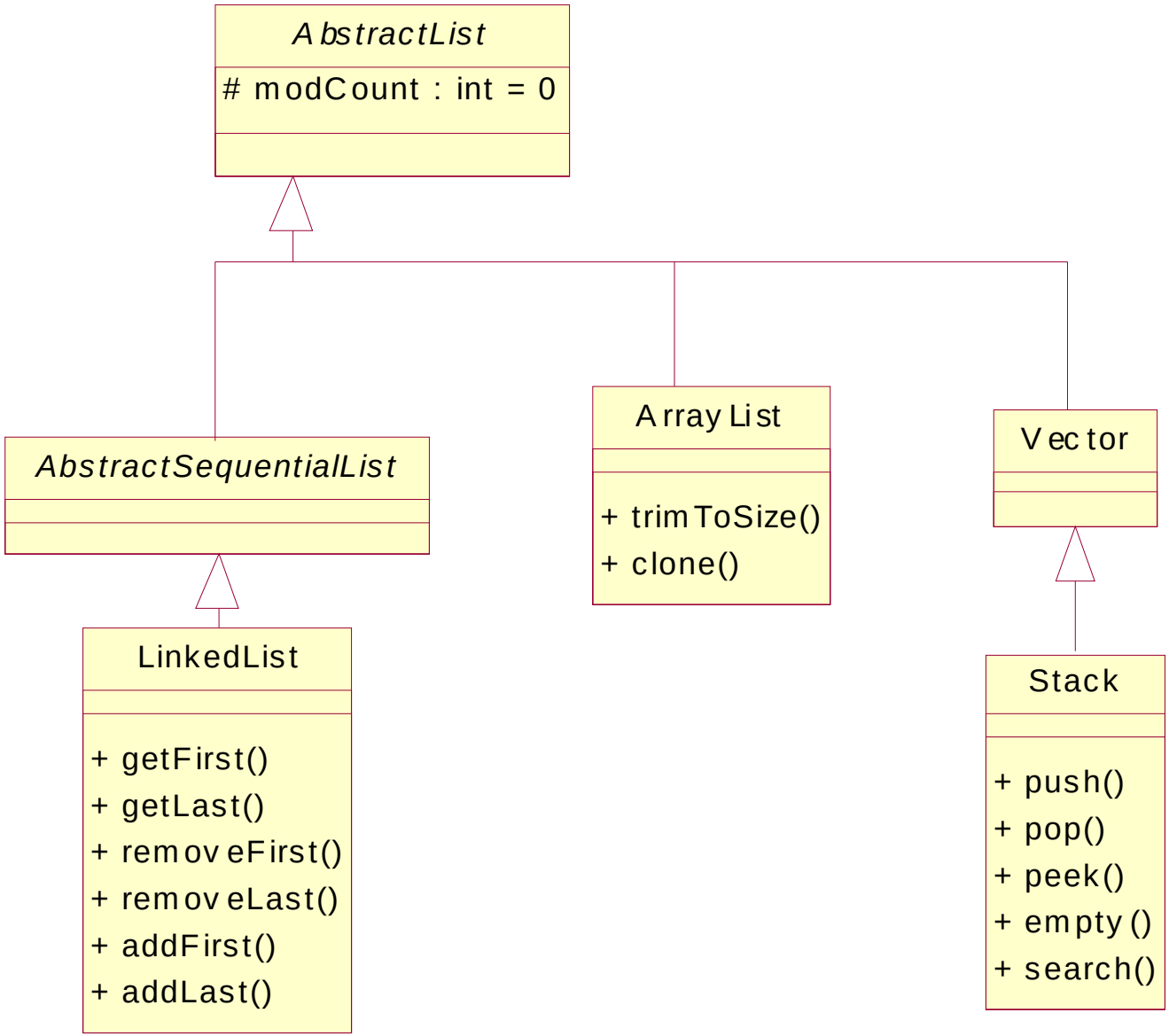
Implémentations

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

Diagramme


<<Interface>>
Collection
<<Interface>>
List
<<Interface>>
Set
AbstractCollection
<<Interface>>
SortedSet
AbstractList
modCount : int = 0
AbstractSet
AbstractSequentialList
ArrayList
Vector
HashSet
TreeSet
LinkedList
Stack

listes


AbstractList
modCount : int = 0
AbstractSequentialList
ArrayList
+ trimToSize()
+ clone()
Vector
LinkedList
+ getFirst()
+ getLast()
+ removeFirst()
+ removeLast()
+ addFirst()
+ addLast()
Stack
+ push()
+ pop()
+ peek()
+ empty()
+ search()

Conseils pour le typage des méthodes

Paramètres des méthodes

Toujours préférer typer par une interface plutôt que par une implémentation

Choisir l'interface la moins contrainte possible (ex Collection)

Paramètres de retour

Au contraire retourner le type d'implémentation le plus spécifique

Généricité en Java 1.5

Reproche souvent fait à Java

Absence des templates de C++,

De la généricité de Ada

Du polymorphisme paramétrique de ML.

La généricité est introduite en Java 1.5

Elle s'inspire d'une proposition dite GJ (Generic Java)

Pourquoi la Généricité

Les collections d'objets cohérents doivent être gérées par l'utilisateur qui doit

- Déclarer des collections d'Object
- Ne mettre dans ces collections que des éléments du bon type
- Les récupérer en introduisant un cast du bon type

- Le code est donc alourdi de cast
- Une erreur de cast produit une exception à l'exécution.

Un exemple sans généricité

```
interface Collection {
        public void add (Object x);
        public Iterator iterator ();
}

interface Iterator {
        public Object next ();
        public boolean hasNext ();
}

class NoSuchElementException extends RuntimeException {}
```

Une implémentation de Collection

```
class LinkedList implements Collection {
    protected class Node {
        Object elt;
        Node next = null;
        Node (Object elt) { this.elt=elt; }
    }
    protected Node head = null, tail = null;
    public LinkedList () {}
    public void add (Object elt) {
        if (head==null) { head=new Node(elt); tail=head; }
        else { tail.next=new Node(elt); tail=tail.next; }
    }
    …
```

et son itérateur

```
public Iterator iterator () {
    return new Iterator () {
        protected Node ptr=head;
        public boolean hasNext () { return ptr!=null; }
        public Object next () {
            if (ptr!=null) {
            Object elt=ptr.elt; ptr=ptr.next; return elt;
            } else throw new NoSuchElementException ();
        }
    };
}
}
```

Du bon usage du "cast"

```
class Test {
    public static void main (String[] args) {
        // byte list
            LinkedList xs = new LinkedList();
            xs.add(new Byte(0)); xs.add(new Byte(1));
            Byte x = (Byte)xs.iterator().next();
        // string list
            LinkedList ys = new LinkedList();
            ys.add("zero"); ys.add("one");
            String y = (String)ys.iterator().next();
        // string list treated as byte list
            Byte w = (Byte)ys.iterator().next(); // run-time exception
    }}
```

Cast de cast de …

```
// string list list

    LinkedList zss = new LinkedList();

    zss.add(ys);

    String z =(String)((LinkedList)zss.iterator().next()).iterator().next();
```

Collection et généricité

```
interface Collection<A> {
    public void add(A x);
    public Iterator<A> iterator();
}
interface Iterator<A> {
    public A next();
    public boolean hasNext();
}
class NoSuchElementException extends RuntimeException {}
```

Liste générique

```
class LinkedList<A> implements Collection<A> {
    protected class Node {
        A elt;
        Node next = null;
        Node (A elt) { this.elt=elt; }
    }
    protected Node head = null, tail = null;
    public LinkedList () {}
    public void add (A elt) {
        if (head==null) { head=new Node(elt); tail=head; }
        else { tail.next=new Node(elt); tail=tail.next; }
    }
```

Itérateur générique

```
public Iterator<A> iterator () {
    return new Iterator<A> () {
        protected Node ptr=head;
        public boolean hasNext () { return ptr!=null; }
        public A next () {
            if (ptr!=null) {
            A elt=ptr.elt; ptr=ptr.next; return elt;
            } else throw new NoSuchElementException ();
        }
    };
}
```

Erreurs à la compilation !

```
class Test {
    public static void main (String[] args) {
    // byte list
        LinkedList<Byte> xs = new LinkedList<Byte>();
        xs.add(new Byte(0)); xs.add(new Byte(1));
        Byte x = xs.iterator().next();
    // string list
        LinkedList<String> ys = new LinkedList<String>();
        ys.add("zero"); ys.add("one");
        String y = ys.iterator().next();
    // string list treated as byte list
        Byte w = ys.iterator().next(); // compile-time error
    }}
```

Moins de casts

```
// string list list

LinkedList<LinkedList<String>> zss = new LinkedList<LinkedList<String>>();

zss.add(ys);

String z = zss.iterator().next().iterator().next();
```

généricité avancée

- Contraindre les variables de type
- Méthodes génériques
- implémentation des génériques
- joker de type (wilcard)
- collections 1.5
- Exceptions et générique

Polymorphisme Contraint 1

```
interface Priority {
        int getPriority();
}
class Pimpl implements Priority {
        public int getPriority() {…}
}

class PriorityQueue<E      extends Priority        > {
    E queue[];
    void insert(E e) {
        …
        if(e.getPriority() < queue[i].getPriority()) {…}
        …
    }
}
PriorityQueue< Pimpl > p = new PriorityQueue< Pimpl >();
p.insert(new Pimpl ());
```

Polymorphisme Contraint 2

```
interface Comparable<I> {
        boolean lessThan(I o);
}


class SortedList<E   extends Comparable<E>  > {
    E list[];
    void insert(E e) {
        …
        if(e.lessThan(list[i])) {…}
        …
    }
}
```

Composition : &

- TypeVariable keyword $Bound_1$ & $Bound_2$ & … &$Bound_n$
- pour l ’exemple : une liste ordonnée, « sérialisable » et « clonable »

```
public class SortedList<T extends Object & Comparable<? super T>
                                          & Cloneable & Serializable>
   extends AbstractCollection<T>
   implements Iterable<T>, Cloneable, Serializable{

  private List<T> list = new ArrayList<T>();
  private Comparator<T> comparator = null;

  public int size(){return list.size();}

  public Iterator<T> iterator(){
    if( this.comparator!= null) Collections.sort(list, comparator);
    else Collections.sort(list);
    return list.iterator();
  }

  public boolean add(T t){ return list.add(t);}
...
```

Résumé

In class declaration:

```
public class Foo<T, U, V> { … }
```

Using bounds:

```
public class BizObj { … }
interface Service { … }
public class Foo<T, U extends Number, V extends BizObj & Service>
   { … }
```

Multiple interface bounds:

```
public interface Async { … }
public class Foo<T, U extends Number, V extends BizObj & Service
   & Async> { … }
```

Bounds with parameters:

```
public class Foo<T, U extends Number, V extends List<T>> { … }
```

1 : Méthodes génériques

```
…
public static  <T> void permute(T[] tab, int i,int j){
        T temp=tab[i] ;
        tab[i]=tab[j] ;
        tab[j]=temp ;
}
...
String[] tabnoms;
Float[] tabflott;
...
C.permute(tabnoms,3,4); // inutile d'en dire plus
C.permute(tabflott,0,2);
...
.
```

2 : Méthode générique

```
public final class Std {
   public static <T extends Comparable<T>> T min(T a, T b) {
            if(a.compareTo(b) <= 0) return a;
            else return b;
   }
}

public class Main {
   public static void main(String[] args) {
      Integer b1 = new Integer(2); Integer b2 = new Integer(5);
      Integer bMin = Std.min(b1, b2);
      Date d1 = new Date(101, 05, 10); Date d2 = new Date(101, 03, 8);
      Date dMin = Std.min(d1, d2);
      System.out.println("The minimums are: " + bMin + " and " + dMin);
   }
}
```

3 : méthode générique

```
public class Foo
{
 public <U extends List<? extends Number>> void iter(U c)
 {
  for (Number n : c)
  {
   System.out.println("A value of " + n.toString());
  }
 }

 public void caller()
 {
  ArrayList<Integer> list = new ArrayList<Integer> ();
  list.add(new Number(42));
  iter(list);

  Vector<BigDecimal> vector= new Vector<BigDecimal> ();
  vector.addElement(new BigDecimal(42));
  iter(vector);
 }
}
```

Implémentation des génériques

Le Compilateur Java supprime (erasure) toutes les infos génériques

<T> est remplacé par Object

<T extends Serializable> est remplacé par Serializable

Du code générique peut donc fonctionner avec de l'ancien code

List<String> et List<Integer> sont tous deux compilés en List

On ne peut pas surcharger des méthodes avec variable de type

```
class Pair<F,S>{
        void m(F f){..}
        void m(S s){...}
}
```

Les problèmes !

- La généricité n'existe qu'à la compilation
- Le code des classes qui instancient une même déclaration générique est unique à l'exécution
- Les variables de type n'existent plus à l'exécution

Conséquences: Sous Typage

- Ceci est légal

  class Collection<A> {…}

  class Set<A> **extends** Collection<A> {…}

  Collection<X> x = new Set<X>();

- Mais pas ceci !

  class Collection<A> {…}

  class Y **extends** X {…}

  Collection<Y> y = **new** Collection<Y>;

  Collection<X> x = y; // compile-time error

  x.insert(**new** X()); // car ceci violerait alors le typage

1 : Conséquences sur les génériques

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls; // attention !erreur de compilation
// illégal car…
lo.add(new Object()); // 3
String s = ls.get(0); // 4:
```

Du fait du ‘type erasure’ même si

String <:Object

Pourtant on n’a pas

List<String> <: List<Object>

Et ceci est vrai aussi bien sûr pour le passage de paramètres !

L’héritage des acquis !

On hérite d’une situation différente pour les tableaux

// Y <: X

```
Y[] y = new Y[10];
X[] x = y; // legal
x[0] = new X(); //ArrayStoreException.
```

2 : Conséquences sur les tableaux de génériques

```
List<String>[] lsa = new List<String>[10]; // pas autorisé !

Object o = lsa;
Object[] oa = (Object[]) o;

List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));

oa[1] = li; // compatible avec la règle sur les tableaux
String s = lsa[1].get(0); // run-time error - ClassCastException
```

3 : Conséquences

Une méthode:

```
void afficheCollection(Collection c){
    Iterator it=c.iterator();
    while (it.hasNext())
    System.out.println(it.next());
}
```

Comment réécrire cette méthode avec des collections génériques ?

Premier essai:

```
void afficheCollection(Collection<Object> c){
    Iterator<Object> it=c.iterator();
    while (it.hasNext())
    System.out.println(it.next());
}
```

Ne convient pas: on ne peut pas passer un objet de type Collection<Etudiant> en paramètre.

4 : Conséquences

On utilise un wildcard pour indiquer que le type du paramètre est n'importe quelle instance de Collection<T>:

```
void afficheCollection(Collection<?> c){
    for(Object o : c){                          // nouvelle boucle for
            System.out.println(o);
    }
}
```

Remarque: seules les méthodes « sûres » pour tous les types de paramètres peuvent être invoquées dans le corps de la méthode:

```
void insereObjet(Collection<?> c){
    c.add(new Object()); // rejeté à la compilation
}
```

Joker contraint

Bounded Wildcards:

```
void manipCollection(Collection<? extends Etudiant > c){
...
}
```

Accepte tout paramètre de type Collection<T> où T étend Etudiant.

Mêmes restrictions concernant les méthodes invoquées:

```
void insereObjet(Collection<? extends Etudiant> c){
    c.add(new Etudiant()); // rejeté à la compilation (c read-only)
}
```

Wildcards suite

```
List<?> list = createList();     // List de “Unknown”


List<Integer> intList = new ArrayList<Integer>();
List<? extends Number> numList = intList;


public void iterList(List<? extends Number> list)
{
 …
}
```

1: wilcard résumé

- **Collection<? extends Number> c = new ArrayList<Integer>(); // Read-only,**
- ***// c.add(new Integer(3)); // ? Aucune connaissance des sous-classes***

- **Collection<? super Integer> c = new ArrayList<Number>();**
- ***c.add(new Integer(3)); // ok***
- ***Integer i = c.iterator().next(); // erreur ? Comme Object***

- **Collection<?> c = new ArrayList<Integer>();**
- **System.out.println(" c.size() : " + c.size());**
- **c.add(new Integer(5)); // erreur de compil**

2: wilcard résumé
Méthodes génériques, syntaxe

```
<T> void ajouter(Collection<T> c,T t){
        c.add(t);
}

<T> void ajouter2(Collection<? extends T> c,T t){
        //c.add(t); // erreur  de compilation read_only ....
}

<T> void ajouter3(Collection<? super T> c,T t){
        c.add(t);
}
```

Exception et généricité

- Et si l'exception était générique (enfin, presque)

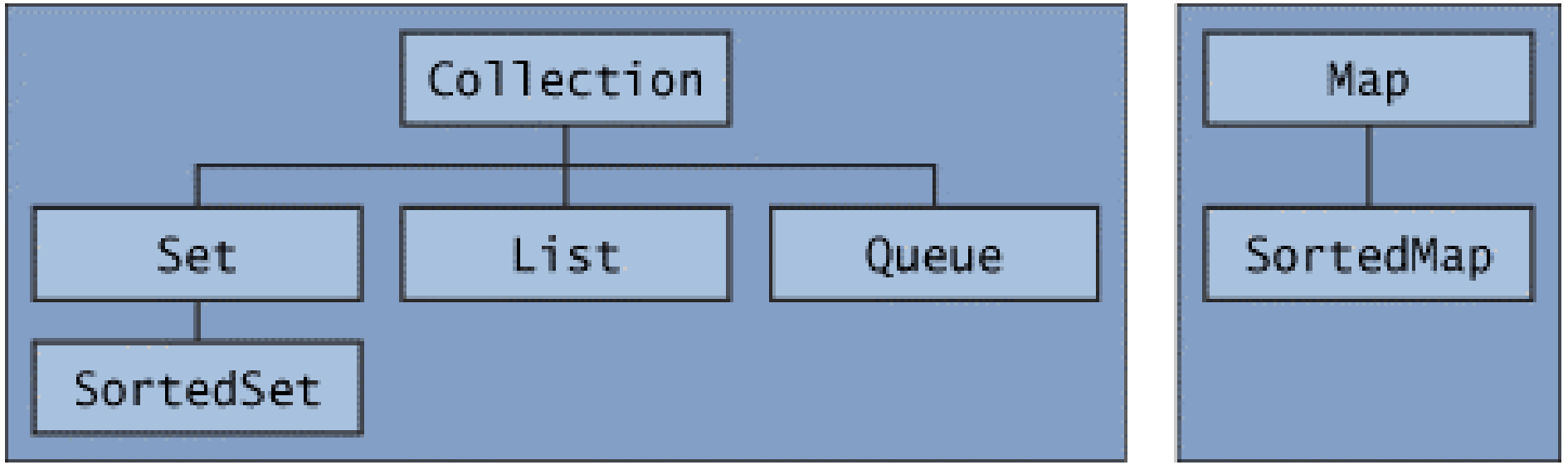
```
public interface PileI<T, PileVide extends Exception,
                          PilePleine extends Exception>{

      public void empiler(T elt) throws PilePleine;
      public T depiler() throws PileVide;
}
```

à l 'aide de deux implémentations concrètes PileVideException et PilePleineException

```
public class Pile<T> implements
                            PileI<T,PileVideException,PilePleineException>{
      private Object[] zone;
      private int index;
      public Pile(int taille){…}

      public void empiler( T elt) throws PilePleineException{
               if(estPleine()) throw new PilePleineException();
               ....
    }

    }
```

Collections 1.5

Les interfaces



Une interface Queue en plus

Les interfaces… de java.util

- Collection<E>
- Comparator<T>
- Enumeration<E>
- Iterator<E>
- List<E>
- ListIterator<E>
- Map<K,V>
- Map.Entry<K,V>
- Queue<E>
- Set<E>
- SortedMap<K,V>
- SortedSet<E>

Les interfaces sont génériques

```
public interface Collection<E> extends Iterable<E> {
    //Basic operations int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element); //optional
    boolean addAll(Collection<? extends E> c)
    Iterator<E> iterator(); //
    boolean containsAll(Collection<?> c);
    …
    //Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
    }
```

interface Map

```
public interface Map<K,V> {
        V put(K key, V value);
        V get(Object key);
        V remove(Object key);
        boolean containsKey(Object key);
        …..
        void putAll(Map<? extends K,? extends V> t);
        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,V> {
                K getKey(); V getValue(); V setValue(V value); }
} }
```