Chapitre 6
Hiérarchies de types

Voir Liskov&Guttag,
ch. 7 « Type Hierarchy », pp 147-187

Vue d'ensemble

- Une hiérarchie de types permet de définir des **familles de types** composées d'un **super-type** et de **sous-types**
  - représentant des hiérarchies du "monde réel" (mammifères, polygones...)
  - ou propres aux programmes (`BufferedReader` spécialisant `Reader`, `NullPointerException` spécialisant `Exception`...)
- Une famille peut avoir un ou **plusieurs niveaux**
- Le **principe de substitution** maintient la cohérence entre comportement du super-type et des sous-types

Objectif des hiérarchies de types

- **Étendre le comportement d'un ADT**
  - `IntSet` est étendu par deux sous-types :
    - `LogIntSet` (journalise chaque `int` inséré dans `this`)
    - `MaxIntSet` (retourne le plus grand `int` de `this`)

OU

- **Fournir de multiples implémentations à un ADT**
  - `Poly` peut être étendu par deux sous-types :
    - `DensePoly` (instancié lorsque le polynôme est dense)
    - `SparsePoly` (instancié lorsque la densité du polynôme est faible)

NB: aucun des sous-types n'ajoute de méthode à `Poly`

Avantages de la hiérarchie de type

- Permet d'affiner ou changer le comportement d'un ADT…
- … au bon niveau d'abstraction
- Minimise les changements dans les programmes clients
- Maximise la réutilisation (du code et des specs) des ADTs
- Minimise la complexité des ADTs

Affectation et dispatching (rappel)

- L'intérêt des hiérarchies de types repose sur la **flexibilité de l’affectation et du dispatching**
- Par exemple, comme on l'a vu, le code suivant est valide :

```
Poly p1 = new DensePoly(); // le poly nul
Poly p2 = new SparsePoly(3,20); // 3x^20
int i = p1.degree();
```

- Cet appel est licite puisque le type apparent (déclaré) de `p1` est `Poly`
- Mais, à l'exécution, c'est le code d'un sous-type (en l'occurrence `DensePoly`) qui peut être (ici, **sera**) invoqué
- Le compilateur s'assure que tous les sous-types de `Poly` ont toutes les méthodes de `Poly`

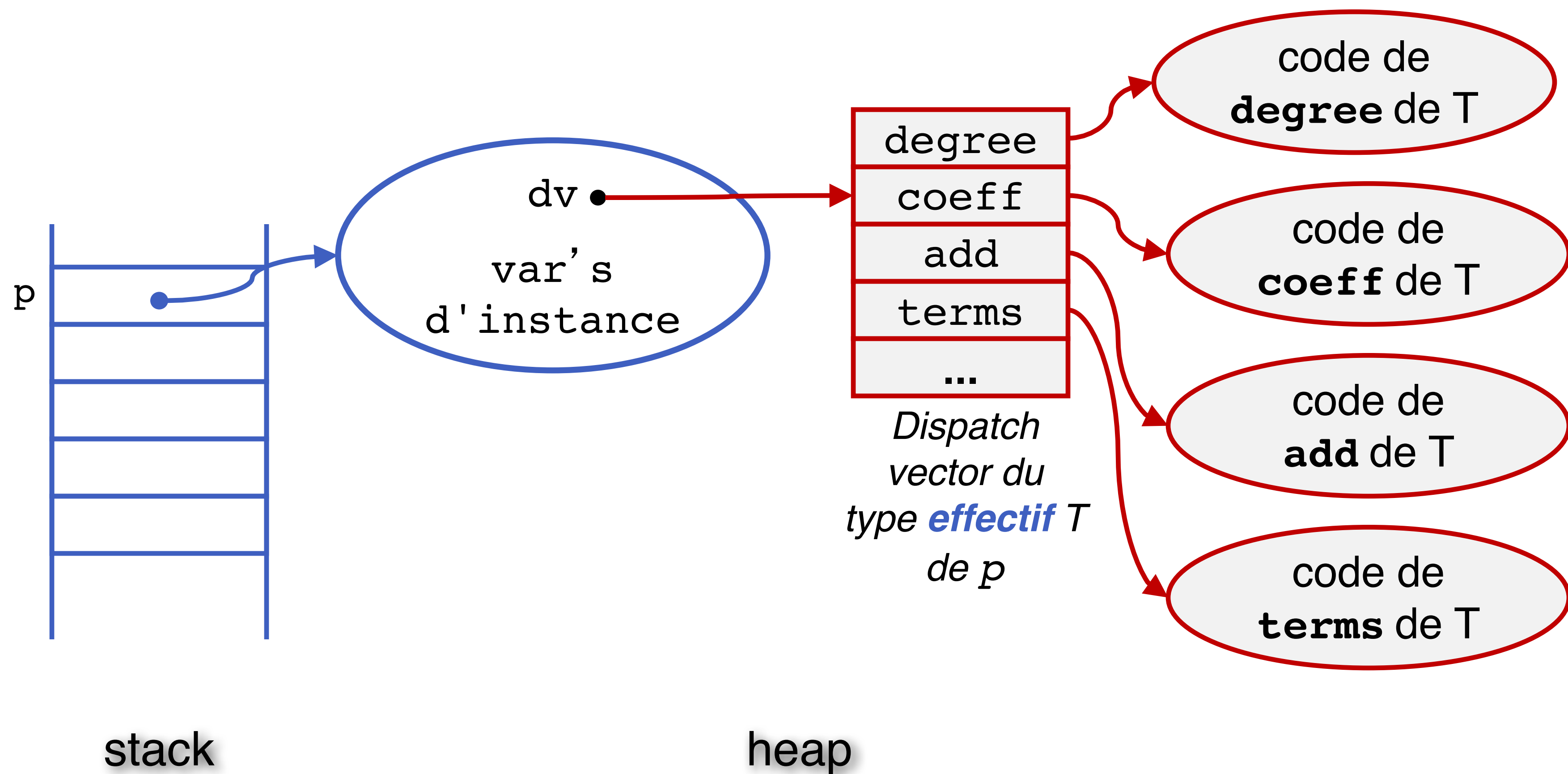
Affectation et dispatching (rappel)

- Le compilateur ne peut donc pas toujours déterminer le code qui sera appelé
- Par exemple

```
static Poly diff(Poly p) {
    Iterator<Integer> g = p.terms();
    …
}
```

- En conséquence, le compilateur ne génère pas le bytecode de l'appel directement mais génère un bytecode qui va chercher le code à exécuter dans le « **dispatch vector** » du type **effectif** de `p`

Affectation et dispatching (rappel)



Définition d'une hiérarchie de types

- On commence par définir le super-type
  - **peut** être **incomplet** (ex: manque des constructeurs, ne fournit pas d'implémentations de certaines méthodes...)
- Les sous-types sont définis relativement au super-type
  - les parties du super-types qui sont conservées ne doivent pas être précisées (réutilisation implicite)
  - on n'indique que ce qui est nouveau
    - exemples :
      - constructeurs
      - méthodes supplémentaires
      - (re-)implémentation de méthodes du super-type

Définition d'une hiérarchie de types en Java

- Un **super-type** est défini par une **classe** ou une **interface** qui fournit sa **spécification**, et si c'est une classe, éventuellement une **implémentation partielle** ou **complète**

- Une sous-classe **hérite** de la définition / implémentation d'**une (et une seule) classe** (`extends`) et de **0,1 ou plusieurs interfaces** (`implements`)

Définition d'une hiérarchie de types en Java

- Java distingue:
  - **classes concrètes** : fournissent une **implémentation complète** d'un type ⇒ peuvent être instanciées (et donc exécutées)
  - **classes abstraites** : fournissent **au plus une implémentation partielle** d'un type ⇒ **ne** peuvent **pas** être instanciées (ni exécutées)
    - peuvent avoir des **méthodes abstraites** (sans implémentation)
  - pour le code appelant, cette distinction n'a pas d'importance
    - sauf pour les appels aux constructeurs

Définition d'une hiérarchie de types en Java

- Une sous-classe déclare sa super-classe par une clause `extends` dans son en-tête
  - cela lui confère automatiquement les méthodes (non `private`) de sa super-classe (avec les mêmes signatures)
  - elle peut fournir des méthodes supplémentaires
- Exemple

```
public class DensePoly extends Poly {
    …
}
```

- Rappel : absence d'`extends` = `extends Object` implicite
- À part `Object`, toute classe a une et une seule **super-classe** directe (mais un type possède éventuellement plusieurs **super-types** directs : la super-classe directe + les interfaces implémentées)

Définition d'une hiérarchie de types en Java

- Une classe **concrète**
  - **doit fournir** une **implémentation** des **constructeurs**
  - **doit fournir** une **implémentation** des **méthodes abstraites** de sa super-classe
  - **peut réimplémenter** (overriding) les **méthodes concrètes** (=non `abstract`) non `final` de sa super-classe
    - NB: une méthode est déclarée `final` si on veut en interdire la ré-implémentation (pas utilisé dans le cours)
  - **hérite** des **méthodes concrètes non-réimplémentées** (dont les **méthodes** `final`) de sa super-classe
  - **peut fournir** des **méthodes supplémentaires**

Définition d'une hiérarchie de types en Java

- La **rep** d'un objet qui instancie une sous-classe est composée
  - des **variables d'instance** déclarées dans la **super-classe** (rep héritée)
  - **+** les **variables d'instance** déclarées dans la **sous-classe**

- Question de conception: comment le code d'une sous-classe va-t-il accéder à sa rep héritée (généralement `private`) ?
  - **indirectement**, c-à-d via les méthodes `public` de la super-classe ?
    - idéal mais peut s'avérer inefficace ou peu pratique
  - **directement** ? Dans ce cas, la rep de la super-classe peut être déclarée `protected`
    - Accessible aux sous-classes **mais** aussi au package
    - Attention: la rep est exposée !

Exemple : IntSet

```java
/**
 * @overview Les IntSets sont des ensembles non bornés d'entiers.
 * Ils sont mutables.
 * De manière générale, un IntSet est défini comme {x1, x2,...,xn}
 * où chaque xi est un int (pour 1 <= i <= n)
 */
public class IntSet {

     private List<Integer> els;

     // Constructeurs
     /**
      * @effects initialise this à l'ensemble vide
      */
     public IntSet() { els = new ArrayList<>(); }

     /**
      * @effects crée une copie de c
      */
     public IntSet(IntSet c) {
          els = new ArrayList<>();
          for(Integer i : c.els) { els.add(i); }
     }
```

Exemple : IntSet

```
// Méthodes
/**
 * @modifies this
 * @effects this_post = this U {x}
 */
public void insert (int x) { if (els.indexOf(x) < 0) els.add(x); }

/**
 * @modifies this
 * @effects this_post = this \ {x}
 */
public void remove (int x) { els.remove((Object)x); }

/**
 * @return true si x appartient à this; renvoie false sinon
 */
public boolean isIn (int x) { return els.contains(x); }

/**
 * @return la cardinalité de this
 */
public int size () { return els.size();  }
```

Exemple : IntSet

```
public String toString() {
    if (els.size()==0) { return "IntSet:{}"; }

    StringBuilder s = new StringBuilder("IntSet: {" + els.get(0));
    for (int i = 1; i < els.size(); i++) {
        s = s.append(" , "  + els.get(i));
    }
    return s.append("}").toString();
}

/**
 * @return un générateur qui produira tous les éléments de this
 * chacun exactement une fois, sans ordre particulier
 * @requires this ne doit pas être modifié pendant
 * que le générateur est utilisé
 */
public Iterator<Integer> elements() {
    return Collections.unmodifiableCollection(els).iterator();
}
```

Exemple : IntSet

```
public boolean repOK() {
    if (els == null) return false;
    for (int i = 0; i < els.size(); i++) {
    Integer x = els.get(i);
        for (int j = i+1; j < els.size(); j++) {
            if (x.equals(els.get(j))) return false;
        }
    }
    return true;
}

/**
 * @return true si this est un sous-ensemble de s; false sinon.
 */
public boolean subset (IntSet s) {
    if (s==null) { return false; }
    for (int i : els) {  if (!s.isIn(i)) { return false; } }
    return true;
 }
}
```

Exemple : `IntSet`

- Remarques :
  - `IntSet` est une classe concrète
    - instanciable et donc utilisable telle quelle
  - `IntSet` ne possède pas de variables, ni de méthodes `protected`
    - acceptable dans ce cas car l'itérateur `elements` fournit un accès aisé

Exemple : MaxIntSet (spec)

```
/**
 * @overview Un MaxIntSet est un sous-type d'IntSet
 * muni d'une méthode supplémentaire max
 */
public class MaxIntSet extends IntSet {

    // Constructeurs
    /**
     * @effects initialise this à l'ensemble vide
     */
    public MaxIntSet() { … }

    // Méthodes
    /**
     * @throws EmptyException si this est vide
     * @return le plus grand élément de this, sinon
     */
    public int max() throws EmptyException { … }
}
```

Exemple : `MaxIntSet` (spec)

- Remarques
  - `MaxIntSet` hérite entièrement du comportement de `IntSet` : **pas de re-spécification de méthodes existantes**
  - `MaxIntSet` ajoute la méthode `max`
  - **Seul ce qui est neuf ou ce qui change doit être spécifié**. Pour le reste, on se ramène à `IntSet`
    - Les méthodes redéfinies (overridées) sont préfixées de l'annotation `@Override`

Exemple : `MaxIntSet` (impl #1)

```
/**
 * @overview Un MaxIntSet est un sous-type d'IntSet
 * muni d'une méthode supplémentaire max
 */
public class MaxIntSet extends IntSet {

     /** … */
     public MaxIntSet() { super(); }

     /** … */
     public int max () throws EmptyException {
          Iterator<Integer> g = elements();
          if (!g.hasNext()) { throw new EmptyException("MaxIntSet.max()"); }
          int biggest = g.next();
          while (g.hasNext()){
               int z=g.next();
               if (z>biggest) {biggest = z;}
          }
          return biggest;
     }
}
```

Exemple : `MaxIntSet` (impl #1)

- Remarques
  - dans cette implémentation, seul `max` et le constructeur doivent être implémentés; les implémentations des méthodes héritées peuvent être gardées telles quelles
  - cette implémentation n'est pas très efficace: le `max` est recalculé à chaque appel même s’il n’a pas changé depuis l'appel précédent $\Rightarrow$ impl #2

Exemple : `MaxIntSet` (impl #2)

```
/**
 * @overview Un MaxIntSet est un sous-type d'IntSet
 * muni d'une méthode supplémentaire max
 */
public class MaxIntSet extends IntSet {

    // rep supplémentaire
    private int biggest; // le + grand élément si this n’est pas vide

    /** …  */
    public MaxIntSet() { super(); }

    @Override public void insert(int x) {
        if (size()==0 || x > biggest) { biggest = x; }
        super.insert(x);
    }

    …
```

Exemple : MaxIntSet (impl #2)

```
    @Override
    public void remove(int x) {
        super.remove(x);
        if (size()==0 || x < biggest) { return; }
        Iterator<Integer> g = elements();
        biggest = g.next();
        while (g.hasNext()) {
            int z=g.next();
            if (z>biggest) { biggest = z; }
        }
    }

    /** …  */
    public int max () throws EmptyException {
        if (size() == 0) { throw new EmptyException("MaxIntSet.max()"); }
        return biggest;
    }
}
```

Exemple: `MaxIntSet` (impl #2)

- Remarques
  - `MaxIntSet` possède une variable d'instance supplémentaire
  - En plus du constructeur et de `max`, `MaxIntSet` réimplémente `insert` et `remove`
    - l'implémentation des autres méthodes peut être héritée
  - Dans le constructeur, l'appel au constructeur de la super-classe (`super()`) peut être omis
    - donc, `public MaxIntSet() { }` aurait été également une implémentation correcte du constructeur
    - par défaut, `super()` est toujours exécuté comme première instruction du constructeur de la sous-classe
    - `els` est donc toujours initialisé avant que la moindre instruction de `MaxIntSet` soit exécutée

Exemple: `MaxIntSet` (impl #2)

- Remarques (suite)
  - ***dans le code de*** `MaxIntSet`, il faut pouvoir distinguer les appels aux méthodes réimplémentées des appels aux méthodes d'origine de la super-classe
    - `insert(x)` (équivalent à `this.insert(x)`) désigne un appel à la réimplémentation de `insert` par `MaxIntSet`
    - `super.insert(x)` désigne un appel à l'implémentation de `insert` par `IntSet`
  - les versions d'origine sont accessibles à la sous-classe mais **ne le sont pas pour les appels extérieurs** (si l'objet appelé est un `MaxIntSet`)
    - ex: `IntSet s = new MaxIntSet(); s.insert(3);`
      est toujours un appel à la version réimplémentée de `insert`

IR et FA revisités

- La FA d'une sous-classe est typiquement définie en utilisant la FA de la super-classe
- Exemple

```
// La FA est:
// AF_MaxIntSet(c) = AF_IntSet(c)
```

  - Ici, la fonction est la même, c-à-d que **les objets abstraits** représentés ne sont pas affectés par la variable `biggest`
  - Ceci indique aussi que `MaxIntSet` repose sur `IntSet` pour ce qui est du stockage des éléments de l'ensemble

IR et FA revisités

- Exemple

```
// L'IR est :
// I_MaxIntSet(c) :
// c.size > 0 => ( c.biggest in AF_IntSet(c)
//              &&   for all x in AF_IntSet(c): x <= c.biggest
//                 )
```

- NB: ∈ est noté `in`
- **L'IR d'une sous-classe (`I_sub`) ne doit inclure la vérification de l'IR de sa super-classe (`I_super`) que si la super-classe possède des variables `protected`**
  - ce n'est pas le cas ici
- Cependant, dans tous les cas, **`repOK` de la sous-classe doit toujours faire appel à `repOK` de la super-classe**

IR et FA revisités

- Exemple :

```
/** @overview …  */
public class MaxIntSet extends IntSet {
// comme avant +

    @Override public boolean repOK() {
        if (!super.repOK()) return false;
        if (size() == 0) return true;
        boolean found = false;
        Iterator<Integer> g = elements();
        while (g.hasNext()) {
            int z = g.next();
            if (z>biggest) { return false; }
            if (z == biggest) { found=true; }
        }
        return found;
    }
}
```

Exemple: `MaxIntSet` (impl #3)

- `remove` de `MaxIntSet` pourrait ne pas être jugée efficace car elle doit parfois parcourir `els` deux fois
  - pour supprimer `x`
  - pour recalculer `biggest`
- Une implémentation plus efficace peut être apportée en donnant à `els` la visibilité `protected`
- Dans ce cas, l'IR devient:

```
// L'IR est :
// I_MaxIntSet(c) =
// I_IntSet(c) && c.size() > 0 =>
//                      (  c.biggest in AF_IntSet(c)
//                     &&   for all x in AF_IntSet(c): x<=c.biggest
//                      )
```

Classes abstraites

- Une **classe abstraite** :
  - ne fournit qu'une **implémentation partielle** d'un type
  - **peut** avoir des **variables d'instances** (et dans ce cas, aussi des constructeurs)
    - ces **constructeurs ne peuvent cependant pas être appelés de l'extérieur**; uniquement par des sous-classes pour initialiser la rep héritée
  - typiquement, contient des méthodes normales et des méthodes abstraites
    - **les méthodes normales font cependant souvent usage de méthodes abstraites** (**« template pattern »**)
    - plus l'implémentation est « haut » dans la hiérarchie des types, plus la réutilisation est grande, plus les sous-classes sont simples et plus la correction est facile à établir

Exemple: `SortedIntSet` (spec)

```
/**
 * @overview Un SortedIntSet est un sous-type d'IntSet dont la  méthode
 * elements retourne les éléments de this en ordre croissant
 */
public class SortedIntSet extends IntSet {

    /**
     * @effects initialise this à l'ensemble vide
     */
    public SortedIntSet() { … }


    …
```

Exemple: SortedIntSet (spec)

```
/**
 * @return un générateur qui produira tous les éléments de this,
 * chacun exactement une fois, en ordre croissant
 * @requires this ne doit pas être modifié pendant que
 * le générateur est utilisé
 */
@Override
public Iterator<Integer> elements() { … }


/**
 * @see {@link IntSet#subset(Inset)}
 */
public boolean subset (SortedIntSet s) { … }

}
```

Exemple: `SortedIntSet` (spec)

- Remarques
  - la spec de `elements` a changé par rapport à `IntSet`
  - `SortedIntSet` possède deux méthodes `subset` surchargées (**overloading**):
    - `public boolean subset (IntSet s) // héritée`
    - `public boolean subset (SortedIntSet s) // overloading`
  - la méthode `subset` supplémentaire a (par défaut) la même spécification que la méthode `subset` héritée
    - ajout de l'annotation `@see {@link IntSet#subset(IntSet)}` pour créer un lien vers la spécification de `subset(IntSet)` définie dans `IntSet`
  - la méthode supplémentaire a été ajoutée pour bénéficier d'une plus grande efficacité quand les deux objets (`this` et le paramètre `s`) sont tous deux des `SortedIntSet`

Exemple: `SortedIntSet` (spec)

- Remarques (suite)
  - pour implémenter `SortedIntSet` on aimerait utiliser une `OrderedIntList` (cf. section 6.6 du bouquin de Liskov) comme rep
  - cependant, `IntSet` définit déjà une rep pour les éléments (l'`ArrayList els`) qui est héritée par ses sous-classes
  - de plus, l'overriding ne fonctionne que pour les méthodes et pas pour les variables
  - maintenir les deux rep dans une même classe est **inefficace**, **fastidieux** et **risqué**
  - une solution est de définir `IntSet` comme **classe abstraite dépourvue de rep**

Exemple : IntSet (abstract)

```
public abstract class IntSet {

     protected int sz; // la taille

     // Constructeurs
     public IntSet() { sz=0; }

     public IntSet(IntSet c) { this.sz = c.sz; }

     // Méthodes abstraites
     public abstract void insert(int x);
     public abstract void remove(int x);
     public abstract Iterator<Integer> elements();
     public abstract boolean repOK();

     // Méthodes
     public boolean isIn (int x) {
          Iterator<Integer> g = elements();
          while (g.hasNext()) {
               if (g.next().equals(x)) { return true; }
          }
          return false;
     }
     …
```

Exemple : IntSet (abstract)

```
    public int size ( ) { return sz; }

    @Override public String toString() {
        if (sz==0) { return "IntSet:{}"; }
        Iterator<Integer> g = elements();
        StringBuilder s = new StringBuilder("IntSet: {" + g.next());
        while (g.hasNext()) {
            s = s.append(" , "  + g.next());
        }
        return s.append("}").toString();
    }

    public boolean subset (IntSet s) {
        if (s==null) { return false; }
        Iterator<Integer> g = elements();
        while (g.hasNext()) {
            if (!s.isIn(g.next())) { return false; }
        }
        return true;
    }
}
```

Exemple : `IntSet` (abstract)

- Remarques
  - `insert`, `remove`, `elements` et `repOK` sont abstraites
  - `isIn`, `subset` et `toString` sont implémentées en utilisant la méthode abstraite `elements` (**« template pattern »**)
  - `size` pourrait être implémentée en utilisant `elements` mais ce serait inefficace, d'où la variable d'instance `sz`
  - visibilité de `sz`
    - `private` (+ accès via méthodes `protected`): `IntSet` pourra alors garantir l'IR (partiel) `sz >=0`. **Pas intéressant !**
    - ce qui nous intéresse, c'est de garantir que `sz` est la taille de l'ensemble ⇒ `protected`
  - comme `IntSet` n'a pas de rep propre, elle n'a pas non plus de FA, d'IR, ni de méthode `repOK`

Exemple : SortedIntSet (impl)

```
public class SortedIntSet extends IntSet {
    private OrderedIntList els;

    // FA(c) = {c.els[1],...,c.els[c.sz]}
    // I_SortedIntSet: c.els != null && c.sz = c.els.size

    public SortedIntSet() { els = new OrderedIntList(); sz = 0; }

    public SortedIntSet(SortedIntSet s) throws NullPointerException {
        if (s == null) throw new
            NullPointerException("SortedIntSet.SortedIntSet(SortedIntSet)");
            els = new OrderedIntList(s.els);
            sz = s.sz;
    }

    public Iterator<Integer> elements() { return els.smallToBig(); }


    @Override public boolean subset(IntSet s) {
        try {
            return subset((SortedIntSet) s);
        } catch (ClassCastException e) {
            return super.subset(s);
        }
    }
    …
```

Exemple : SortedIntSet (impl)

```
/**
 * @see {@link IntSet#subset(Inset)}
 */
public boolean subset(SortedIntSet s) {
     // l'implementation tire advantage du fait que l'itérateur smallToBig de
     // OrderedIntList retourne les éléments en ordre croissant
     if (s==null) { return false; }
     if (s.size() == 0 && size() > 0) return false;
     Iterator<Integer> g = elements();
     while (g.hasNext()) {
          int currentInt = g.next();
          try { if (currentInt < s.els.least()) return false;
                 if (currentInt > s.els.greatest()) return false;
          } catch (EmptyException e) {
               // normalement s devrait contenir au moins un élément
               throw new FailureException("SortedIntSet.subset(SortedIntSet)");
          }
          if (!s.isIn(currentInt)) { return false; }
     }
     return true;
}

…
```

Exemple : SortedIntSet (impl)

```
    @Override public boolean isIn(int x) {
        Iterator<Integer> g = elements();
        while (g.hasNext()) {
            int currentVal = g.next();
            if (currentVal > x) { return false;}
            if (currentVal == x) { return true; }
        }
        return false;
    }

    @Override public void insert(int x) {
        try { els.addEl(x); } catch (DuplicateException e) { // ne rien faire }
    }

    @Override public void remove(int x) {
        try { els.remEl(x); } catch (NotFoundException e) { // ne rien faire }
    }

    @Override public boolean repOK() {
        if (els == null) return false;
        if (sz != els.size()) return false;
        return true;
    }
}
```

Exemple : `SortedIntSet` (impl)

- Remarques
  - la méthode `subset` supplémentaire est ajoutée pour des questions d'efficacité
  - la méthode `subset` héritée est ré-implémentée pour invoquer la méthode `subset` supplémentaire dans le cas où le paramètre est de type `SortedIntSet`

Exemple : `SortedIntSet` (impl)

- Remarques (suite)
  - la FA établit la correspondance entre `OrderedIntList` (`els`) (rep) et les ensembles d'entiers (objets abstraits)
    - NB: les `OrderedIntList` sont traitées comme des séquences, d'où la liberté de la notation [ ]
  - l'IR contraint à la fois la variable d'instance `els` et la variable d'instance `sz`
    - le fait qu'`els` est ordonné et ne contient pas de doublons ne doit pas être mentionné car cela est vrai pour toute `OrderedIntList`

Utilisation de `protected`

- Il faut **éviter** l'utilisation de `protected` pour les membres de classes (variables d'instances et méthodes) pour 2 raisons :
  - si les membres demeurent `private`, la super-classe peut être ré-implémentée sans affecter les sous-classes
  - `protected` rend les membres **accessibles par tout le package**
- Les membres `protected` sont introduits pour permettre l'implémentation efficace des sous-classes
- 2 alternatives lors de l'utilisation de la visibilité `protected` :
  - variables `protected`
  - variables `private` et accès via méthodes `protected`
    - OK si cela permet à la super-classe de maintenir un invariant intéressant

Interfaces

- Une **classe** (d'abstraction de données) sert
  - à **définir** un type
  - **et** à en donner une **implémentation** complète ou partielle
- Une **interface** ne sert qu'à **définir** un type
  - elle ne fournit **pas d'implémentation**
  - elle ne contient que des **méthodes** à la fois **publiques, non statiques** et **abstraites**
    - ces 3 caractéristiques sont **implicites** pour les interfaces
    - on continuera, par convention, à expliciter le modificateur `public`
- Exemples d'interfaces : `Iterator`, `Cloneable`, `List`, `Iterable`

Interfaces

- Avantage principal : **une classe peut implémenter un nombre quelconque d'interfaces** (alors qu'elle ne peut étendre qu'une seule super-classe)
  - Les interfaces servent à fournir à une classe plusieurs super-types directs
  - Elles *simulent* ce qu'on appelle l'« **héritage multiple** »
  - Une interface est implémentée par une ou plusieurs classes
- Toute classe qui implémente une interface l'indique dans son en-tête par la clause `implements`
- Exemple:

```
public class SortedIntSet
    extends IntSet
    implements SortedSet, Iterable {
    ...
}
```

Implémentations multiples

- L'**implémentation multiple** est une manière de définir une **forme restreinte de famille de types** où
  - tous les « membres » (les types) de la famille
    - ont **exactement les mêmes méthodes** (sauf constructeurs)
    - et le **même comportement** tel que défini par la spécification du type implémenté (l'ancêtre commun)
  - le **type implémenté** (l'ancêtre commun) est soit une **classe abstraite**, soit une **interface**
  - l'**existence des sous-types** est dans une large mesure **invisible pour le code appelant** (hormis lors de la création des instances)
  - des **instances** de **différents sous-types coexistent** à l'exécution

Exemple : `Poly`

- **Idée**: fournir deux implémentations différentes de `Poly`, respectivement pour :
  - les polynômes « denses » (`DensePoly`)
  - les polynômes « creux » (`SparsePoly`)
- **Motivation**: souci d'**efficacité**

Exemple : `Poly` (impl)

```
public abstract class Poly {
     protected int deg; // le degré du polynôme

     /* constructeurs */
     protected Poly (int n) {deg = n;}

     /* méthodes abstraites */
     public abstract int coeff(int d);
     public abstract Poly add(Poly q) throws NullPointerException;
     public abstract Poly mul(Poly q) throws NullPointerException;
     public abstract Poly minus();
     public abstract Iterator<Integer> terms();
     public abstract boolean repOK();

     /* méthodes concrètes */
     public int degree () {return deg; }

     public boolean equals (Object o) {
          if (this == o) return true;
          try {return equals((Poly) o);}
          catch (ClassCastException e){return false;}
     }
     …
```

Exemple : Poly (impl)

```
    public boolean equals (Poly p) {
        if (p == null || deg != p.deg) return false;
        Iterator<Integer> tg = terms();
        Iterator<Integer> pg = p.terms();
        while (tg.hasNext()){
            int tx = tg.next();
            int px = pg.next();
            if (tx != px || coeff(tx) != p.coeff(px)) { return false; }
        }
        return true;
    }

    public Poly sub (Poly p) { return add(p.minus()); }

    public String toString() {
        if (deg == 0) return "Poly : 0";
        StringBuilder sb = new StringBuilder("Poly : ");
        Iterator<Integer> g = terms();
        while(g.hasNext()) {
            int d = g.next(); int coeff = coeff(d);
            sb.append(coeff + "x^" + d + " + " );
        }
        sb.delete(sb.length()-3, sb.length());
        return sb.toString();
    }
}
```

Exemple : `Poly` (impl)

- Remarques
  - `deg` est défini dans la classe abstraite car utile et efficace dans les deux implémentations
  - `deg` est `protected` car `Poly`, à elle seule, ne peut maintenir aucun invariant intéressant sur `deg`

Exemple : DensePoly (impl)

```
public final class DensePoly extends Poly {
     int [] trms; // coefficients des termes jusq'au terme de degre = deg

     public DensePoly(){ super(0);  trms = new int[1]; }

     public DensePoly (int c, int n) throws NegativeExponentException {
          super(n);
          if (n<0) throw new NegativeExponentException("Poly.Poly(int,int)");
          if (c==0) { trms=new int[1]; return; }
          trms = new int[n+1];
          for (int i=0; i<n; i++) { trms[i]=0; }
          trms[n]=c;
     }

     private DensePoly (int n) {
          super(n);
          trms = new int [n+1];
     }

     //implémentation de coeff, mul, minus, terms, hashCode et repOK
```

Exemple : DensePoly (impl)

```
    @Override
    public Poly add (Poly q) throws NullPointerException {
          if (q instanceof SparsePoly) return q.add(this);
          DensePoly la, sm;
          if (deg > q.deg) {la = this; sm = (DensePoly) q;}
          else {la = (DensePoly) q; sm = this;}
          int newdeg = la.deg; //le nouveau degré est plus grand...
          if (sm.deg == la.deg) {//...à moins que son coeff soit zéro
                for (int k = sm.deg; k>0; k--) {
                      if (sm.trms[k]+la.trms[k] !=0) { break; } else { newdeg--; }
                }
          }
          DensePoly r = new DensePoly(newdeg); // crée un nouveau DensePoly
          int i;
          for (i=0; i <= sm.deg && i <= newdeg; i++) {
                r.trms[i] = sm.trms[i] + la.trms[i];
          }
          for (int j = i; j <= newdeg; j++) {
                r.trms [j] = la.trms[j];
          }
          return r;
    }
}
```

Exemple : `DensePoly` (impl)

- Remarques
  - similaire à la version initiale de `Poly` (sauf non ré-implémentation des méthodes héritées du nouveau `Poly`)
  - comme prévu en cas d'implémentation multiple, **les sous-classes ne sont pas indépendantes** : dans les producteurs de `Poly`, il faut décider si on produit un `DensePoly` ou un `SparsePoly`

    ⇒ **complexification des méthodes**

    (voir slide suivant)

Exemple : `DensePoly` (impl)

- Remarques (suite)
  - Exemple: `add`
    - additionner un `DensePoly` et un `SparsePoly` est géré par `SparsePoly`
    - additionner deux `DensePoly` est géré par `DensePoly` et génère un `DensePoly`
      - mais peut-être pas la bonne décision si beaucoup de coefficients se sont annulés mutuellement
      - ⇒ on pourrait ajouter une vérification de la densité à la fin de `add` et, le cas échéant, générer un `SparsePoly`
      - pour cela, `SparsePoly` devrait fournir le constructeur suivant :

```
/**
 * @throws NullPointerException si trms == null
 * @effects sinon, initialise this de manière à représenter le même
 * Poly que celui représenté par trms dans DensePoly
 */
SparsePoly(int[] trms) { … }
```

Sémantique du sous-typage

- **Principe de substitution** (rappel)**: « Si S est un sous-type de T, les objets de type S doivent pouvoir être utilisés dans tout contexte qui requiert des objets de type T »**

- Plus précisément, **3 règles doivent être respectées** :
  - **règle des signatures**
    - « Le sous-type doit avoir toutes les méthodes du super-type et leurs signatures doivent être *compatibles* »
    - vérifiée par le **compilateur** Java
  - **règle des méthodes**
    - « Les appels aux méthodes du sous-type doivent *se comporter comme* des appels aux méthodes du super-type »
    - doit être vérifiée par le **programmeur**
  - **règle des propriétés**
    - « Le sous-type doit préserver toutes les propriétés qui sont vérifiées par le super-type »
    - doit être vérifiée par le **programmeur**

Règle des signatures

- **« Le sous-type doit avoir toutes les méthodes du super-type et leurs signatures doivent être *compatibles* »**
- But : garantir que chaque appel « type-correct » au super-type est un appel « type-correct » au sous-type
- Le **compilateur** garantit que :
  - **le sous-type possède toutes les méthodes du super-type**
  - et que **leurs signatures sont *compatibles***, c-à-d
    - qu'elles sont **identiques**,
    - **sauf pour les exceptions**: **le sous-type doit renvoyer** (`throws`) **les mêmes ou moins de types d'exceptions** que le super-type
      - logique: si l'appel fonctionne en prévoyant la survenance des exceptions A, B et C, il marchera également si seules B et C sont possibles

Règle des signatures

- Avant Java 1.5, la notion de **compatibilité** des signatures **de Java** était en fait **plus exigeante que nécessaire**
- En effet, la règle des signatures identiques (hormis exceptions), exigeait que **les types de retour soient identiques**
- Or, la méthode du sous-type peut se contenter de retourner un sous-type du type de retour de la méthode du super-type. Ceci est désormais autorisé en Java grâce aux **covariant return types**

Règle des signatures

- Exemple (non compatible avec les versions de Java < 1.5) :
  - il serait pratique de pouvoir définir

    ```
    IntSet clone()
    ```

  - qui permettrait d'éviter un `cast` lors de l'appel

    ```
    IntSet t = s.clone();
    ```

  - mais Java exige que `clone` possède la signature

    ```
    Object clone();
    ```

  - ce qui nécessite des appels du type

    ```
    IntSet t = (IntSet) s.clone();
    ```

Règle des méthodes

- **« Les appels aux méthodes du sous-type doivent *se comporter comme* des appels aux méthodes du super-type »**
- But : permettre de raisonner sur la signification des appels en se bornant à la spécification du super-type même si c'est le code d'un sous-type qui est exécuté
- Exemples:
  - pour tout `IntSet`, si on appelle `s.insert(x)`, on sait que `x` est dans l'ensemble au retour de l'appel
  - pour tout `Poly`, si un appel `p.coeff(3)` retourne 6, on sait que le degré de `p` est au moins de 3

Règle des méthodes

- Jusqu'ici, toutes les méthodes de super-types ré-implémentées par les sous-types vues au cours ont gardé la même spécification $\Rightarrow$ règle des méthodes vérifiée
- Une exception: la méthode `elements` de `SortedIntSet`
- Lever une partie de la sous-détermination d'une méthode au niveau du sous-type est fréquent et correct
- Mais toute re-spécification du sous-type n'est pas sans danger...

Règle des méthodes

- Formellement, la règle des méthodes s'exprime comme suit :
  - Règle des préconditions: un sous-type peut ***affaiblir* la précondition**
    - $\text{pré}_{super} \Rightarrow \text{pré}_{sub}$
  - Règles des postconditions: un sous-type peut ***renforcer* la postcondition**
    - $(\text{pré}_{super} \;\&\&\; \text{post}_{sub}) \Rightarrow \text{post}_{super}$

- Rappel :
  - précondition = clause **`@requires`**
  - postcondition = clauses **`@effects`** , **`@return`** , **`@throws`**

Règle des méthodes

- **Affaiblissement des préconditions**
  - idée : la méthode du sous-type *demande moins* de son appelant que la méthode du super-type
  - motivation : logique, puisque l'appelant suppose qu'il a à faire au super-type, alors son appel sera valable pour le sous-type aussi
- Exemple :
  - si `IntSet` possède la méthode suivante

```
/**
 * @requires this n'est pas vide
 * @modifies this
 * @effects ajoute 0 à this
 */
public void addZero() { … }
```

  - alors, un sous-type d'`IntSet` pourrait avoir

```
/**
 * @modifies this
 * @effects ajoute 0 à this
 */
public void addZero() { … }
```

Règle des méthodes

- **Renforcement des postconditions**
  - idée : la méthode du sous-type *fournit plus* à son appelant que la méthode du super-type
  - motivation: logique, puisque l'appelant suppose qu'il a à faire au super-type, alors son appel lui fournira le résultat qu'il attend (plus, éventuellement, un « bonus » qui ne l'intéresse pas). Mais, pour ça, l'appelant doit respecter la précondition (de la méthode du super-type, évidemment!), sinon la méthode peut faire n'importe quoi
- Exemple (suite) :
  - un sous-type d'`IntSet` pourrait aussi avoir

```
/**
 * @modifies this
 * @effects si this n'est pas vide, ajoute 0 à this; sinon, ajoute 1
 */
public void addZero() { … }
```

Règle des méthodes

- Exemples :
  - Soit deux sous-types de `IntSet`
    - l'un, `LogIntSet`, avec :

```
/**
 * @overview un LogIntSet est un IntSet muni d'un log de tous les entiers qui
 * ont fait partie de l'ensemble
 */

/**
 * @modifies this
 * @effects ajoute x à l'ensemble et au log
 */
public void insert (int x) { … }
```

    - l'autre avec :

```
/**
 * @modifies this
 * @effects si x est impair, ajoute x à l'ensemble;
 * sinon, ne fait rien
 */
public void insert (int x) { … }
```

Règle des propriétés

- **« Le sous-type doit préserver toutes les propriétés (abstraites) qui sont vérifiées par le super-type »**
- But : permettre de raisonner sur les objets comme s'ils étaient des instances du super-type même si leur type effectif est plus spécifique
- Attention : raisonnement au niveau de l'**abstraction**
  $\Rightarrow$ sur la **spécification** **!**
- Toutes les méthodes (celles du super-type + celles de tous les sous-types) doivent préserver ces propriétés

Règle des propriétés

- Typiquement (à indiquer dans l'`overview`)
  - Invariants abstraits (**`@invariant`**)
    - ex: la taille (cardinalité) d'un `IntSet` est toujours ≥ 0
  - Propriétés d’évolution du type abstrait
    - ex: le degré d’un `Poly` ne change jamais

- Par exemple, cette dernière propriété est à démontrer pour toutes les méthodes de `Poly`, de `DensePoly` et de `SparsePoly`

- Comment ? Comme d'habitude:
  - par induction sur les types de données
  - montrer que les créateurs et producteurs établissent la propriété
  - montrer que les autres méthodes la préservent

Règle des propriétés : exemple

- Soit `FatSet`,

```
/**
 * @overview Un FatSet est un ensemble mutable d’entiers
 * dont la cardinalité est ≥ 1
 */
```

le constructeur de `FatSet` construit toujours un ensemble contenant un entier et, de plus, `FatSet` n'a pas de `remove` mais

```
/**
 * @modifies this
 * @effects si x est dans this ET que this contient d’autres éléments
 * que x, retire x de this;
 * sinon, ne fait rien
 */
public void removeNonEmpty (int x) { ... }
```

Règle des propriétés : exemple

- Soit `ThinSet`, sous-classe de `FatSet` avec une méthode supplémentaire

```
/**
 * @modifies this
 * @effects si x est dans this, retire x de this;
 * sinon, ne fait rien
 */
public void remove (int x) { ... }
```

- La spécification de cette méthode viole la propriété : cardinalité ≥ 1

- `ThinSet`, **n'**est donc **pas** un sous-type légitime de `FatSet`

Règle des propriétés : exemple

- Soit `SimpleSet` tel que

```
/**
 * @overview Un SimpleSet est un ensemble mutable d'entiers.
 * La cardinalité d'un SimpleSet ne peut que croître.
 */
```

- `SimpleSet` ne possède donc qu'un constructeur et les méthodes `isIn` et `insert`
- `IntSet`, est-il un sous-type légitime de `SimpleSet` ?
- **Non**, car `IntSet` possède une méthode `remove` qui viole la propriété : un `SimpleSet` ne peut que croître

Résumé

- Le mécanisme d'héritage fourni par Java a été utilisé pour définir des familles de types
- 2 utilisations possibles :
  - **implémentations multiples**
    - `Poly` et ses sous-types `DensePoly` et `SparsePoly`
  - **extensions du comportement**
    - `IntSet` et ses sous-types `SortedIntSet`, `LogIntSet` et `MaxIntSet`

Résumé

- Avantages :
  - les similarités et différences entre les types sont clairement identifiées
    $\Rightarrow$ facilite la compréhension du programme
  - **réutilisation** : écrire du code dans le super-type qui fonctionne pour tous les sous-types
  - **extensibilité** : pour ajouter de nouvelles *abstractions* (sous-types) tout en réutilisant du code du super-type
    - avec l'encapsulation: on était prémunis contre les changements d'*implémentation* seulement
- Attention: ces avantages ne sont garantis que si on obéit au principe de substitution
  - étend l'abstraction par spécification aux hiérarchies de types
  - garant de la sémantique du sous-typage