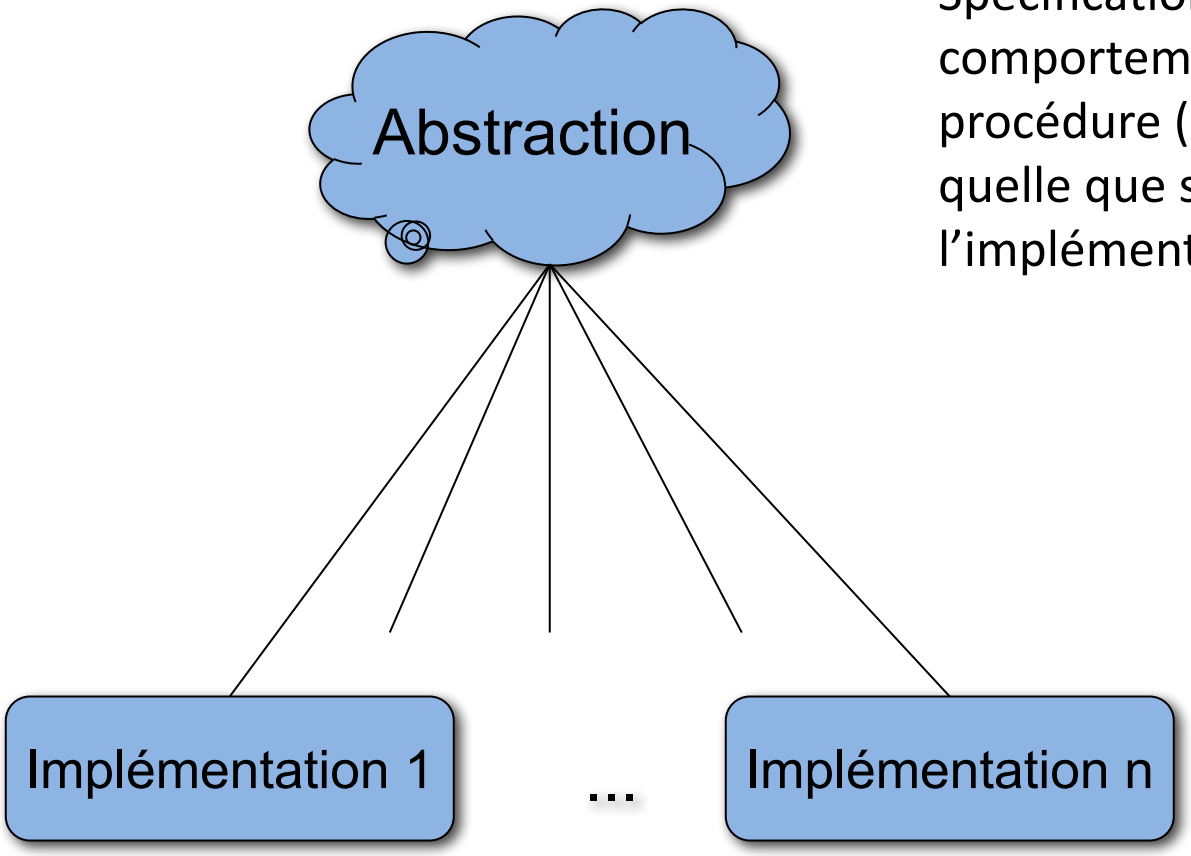
Chapitre 3
Abstraction procédurale

Voir Liskov&Guttag,
ch. 3 « Procedural Abstraction », pp 39-56

Procédures

- Java est un langage de programmation orienté-objet
  - ç-à-d qu'il incorpore des mécanismes facilitant l'application de ce paradigme
- Cependant, Java peut être aussi utilisé pour faire de la programmation impérative (procédurale) classique (comme en Pascal, C, Python…)
- La spécification et la programmation de procédures sont déjà abordées dans les autres cours de programmation
- Nous allons simplement
  - rappeler quelques principes
  - les adapter à Java
  - introduire la notion de spécification d'abstractions procédurales
  - introduire la notion d'exception

Abstraction par spécification



Spécification qui suffit à comprendre le comportement réalisé par une procédure (= le « quoi »)
quelle que soit la manière de l'implémenter (= le « comment »)

Ensemble de procédures (choix d'implémentation) réalisant le même comportement

Abstraction par spécification

- Avantages
  - permet de **rendre les choix d'implémentation indépendants les uns des autres** (une fois la spécification établie)
    - e.g. choix de l'algorithme, des types des variables locales, ... et même parfois du langage
    - NB : des variations de performance peuvent cependant apparaître d'une implémentation à l'autre
  - **compréhension rapide** du rôle d'une procédure (pas besoin de lire le code)
  - permet d'atteindre 2 propriétés : **localité** et **modifiabilité**

Localité

- **Localité** : l'implémentation d'une abstraction peut être lue/écrite sans qu'il n'y ait besoin de consulter l'implémentation d'aucune autre abstraction
  - i.e. pour écrire un programme qui utilise la procédure P, seule la spécification de P est nécessaire
- Permet l'indépendance des activités des programmeurs
- Dans les grands programmes, la quantité d'information qui peut ainsi être ignorée par un programmeur est **énorme**

Modifiabilité

- **Modifiabilité** : une abstraction peut être réimplémentée sans nécessiter la réimplémentation des parties de programme qui l'utilisent
  - pour peu que l'abstraction (la spécification) ne doive pas changer
  - par exemple pour améliorer les performances
  - permet de ne devoir réimplémenter que les parties où la performance est insuffisante (approche itérative)

Spécifications
d'abstraction procédurale

- Les abstractions sont décrites au moyen de spécifications
- Une spécification est écrite dans un langage de spécification qui peut être
  - **formel** i.e. syntaxe et sémantique définies mathématiquement
    - non sujet à ambiguïtés
    - nécessite des connaissances spécialisées
  - **informel**, par ex, le français
    - risques d'ambiguïtés (pouvant être réduits si approche rigoureuse)
    - compréhensible par un plus grand public

Spécifications
d'abstraction procédurale

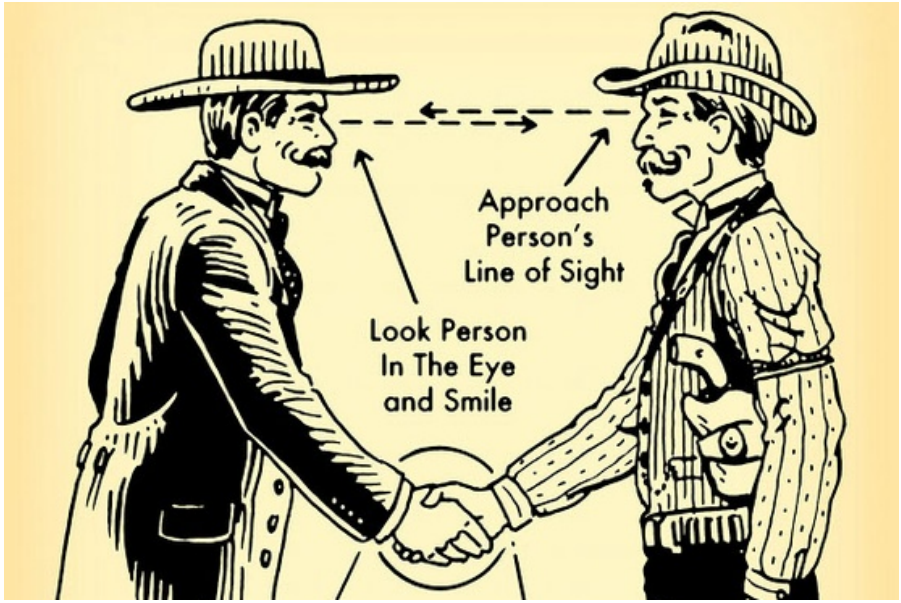
- La spécification d'une procédure comprend
  - sa **signature** (header)
    - son nom
    - nombre, ordre, noms et types des paramètres
    - type du résultat
    - les exceptions renvoyées (voir ch. 4)
    - **NB : il s'agit d'information syntaxique !**
  - une **description des effets** (comportement) de celle-ci
    - information **sémantique**

Spécifications
d'abstraction procédurale

- Exemple
  - `void removeDupls (List l);`
  - `float sqrt (float x);`
- La signature ne donne aucune information sur le comportement que l'on peut attendre de la procédure dans les différents cas de figure
- Comme la signature d'une fonction (par ex, $f : \Re \rightarrow \Re$), une signature ne décrit que la forme des appels à une procédure mais pas **ce que fait** la procédure

Spécifications = contrat

- Le **client** promet de satisfaire les **préconditions** de la procédure
- Le **programmeur** (implémenteur) promet que, si le client satisfait les préconditions, la valeur de retour et l'état lorsque la procédure se termine satisferont les **postconditions**



Contrat de spécification

```
f ()
        REQUIRES: préconditions
        EFFECTS:  postconditions
```

précondition
{ f (); }
postcondition

Si les préconditions sont vraies,
après l'appel à f (),
les postconditions seront vraies.

Canevas de spécifications

```
/**
 * @requires  préconditions
 * @modifies  liste des inputs modifiés par la procédure
 * @effects   ┐
 * @return    ├ postconditions
 * @throws    ┘
 */
type_de_retour nomProc (...) {
  ...
}
```

Canevas de spécifications

- Les 5 clauses ont pour but de décrire :
  - quels sont les inputs de la procédure
  - quelles sont les relations qui existent entre les **inputs** et les **outputs** de la procédure
  - quels sont les effets de la procédure
- inputs
  - **explicites** : les paramètres repris dans la signature
  - **implicites** : fichiers, sorties/entrées standard (`System.out, System.in`)

Canevas de spécifications

- `@requires` définit les conditions sous lesquelles la procédure (ou, du moins, l'abstraction) est définie
  - n'est utile que si la procédure est **partielle**
  - définit les classes d'inputs pour lesquels son comportement est défini
  - si la procédure est **totale**, `@requires` n'est pas nécessaire
  - son comportement est défini pour tous les inputs respectant la signature
- ≡ précondition

Canevas de spécifications

- `@modifies` donne les noms des inputs, explicites et implicites, qui sont modifiés par la procédure
- Si la clause n'est pas vide (i.e., si des inputs sont modifiés), on dira que la procédure a des **effets de bord** (side effects)

Canevas de spécifications

- Les postconditions consistent en 3 clauses:
  - **`@effects`**: les **effets** de la procédure **sur les inputs** n'ayant pas été éliminés par la clause **`requires`**
  - **`@return`**: **l'objet** ou la **valeur retournée** par la procédure
  - **`@throws`**: les **exceptions lancées** par la procédure ainsi que leur cas de déclenchement
- **`@effects`** indique donc
  - quels sont les outputs produits
  - quelles sont les modifications apportées aux inputs figurant dans la clause **`@modifies`**
- la **`postcondition`** n'indique rien de ce qui se passe lorsque la clause **`@requires`** n'est pas satisfaite
  - cela reste indéfini

3 clauses pour les postconditions ?

- Avantages de cette décomposition :
  - Séparation des préoccupations
  - Clarification
  - Compatibilité avec les tags de la Javadoc standard

Procédures en Java

- En Java, on programme les procédures en tant que **méthodes statiques** de classes
- Pour appeler une telle méthode, il faut connaître la classe (statique) dans laquelle elle se trouve mais pas l'objet (dynamique)
  - une méthode statique se comporte de la même façon quel que soit l'objet sur laquelle elle est exécutée
  - souvent, les méthodes statiques sont regroupées dans des **classes statiques** i.e. qui n'admettent aucune instance (même via leurs sous-classes)
- Les méthodes statiques sont parfois appelées des procédures *standalone* (indépendantes) car leur comportement ne dépend pas d'un contexte (objet) particulier

Procédures en Java

- On essayera de regrouper les procédures de manière cohérente plutôt que chaotique
- On s'efforcera de suivre le schéma suivant :

```
/**
 * @overview Description générale de l'utilité de la classe
 */
visibilité class nom_de_classe {
   /**
    * spécifications
    */
   visibilité static type_retour nomProc1 (...)

   /**
    * spécifications
    */
   visibilité static type_retour nomProc2 (...)

   ...
}
```

Procédures en Java

Exemple

```
/**
 * @overview Fournit des procédures servant à manipuler des tableaux d'entiers
 */
public class Arrays {

   /**
    * @return Si x appartient à a, renvoie l'index de x; sinon, renvoie -1.
    */
   public static int search (int[] a, int x) {...}

   /**
    * @requires a est trié par ordre croissant
    * @return Si x appartient à a, renvoie l'index de x; sinon, renvoie -1.
    */
   public static int searchSorted (int[] a, int x) {...}

   /**
    * @modifies a
    * @effects Réarrange les éléments de a par ordre croissant. Par exemple,
    *    si a = [3,1,6,1] avant l'appel, après l'appel, on aura a = [1,1,3,6]
    */
   public static void sort (int[] a) {...}
}
```

Valeurs avant et après l'exécution

- Pour tout input modifié, il faut pouvoir lier sa valeur avant l'appel à sa valeur après l'appel
- Convention :
  - soit x le nom d'un des paramètres formels
  - x dénotera sa valeur **avant** l'appel
  - x_post (parfois aussi noté x') dénotera sa valeur **après** l'appel
- Exemple

```
/**
 * @modifies a
 * @effects Réarrange les éléments de a par ordre croissant.
 *          Par exemple, si a = [3,1,6,1], a_post = [1,1,3,6]
 */
public static void sort (int[] a) {...}
```

Retour de nouveaux objets

- Parfois, on exige qu'une procédure retourne un **nouvel objet**
- Cela doit être indiqué dans la spécification
- Exemple :

```
/**
   * @return renvoie un *nouveau* tableau contenant les mêmes
   * éléments que a dans le même ordre mais où chaque élément
   * supérieur à n a été remplacé par n
  */
 public static int[] boundArray (int[] a, int n) {...}
```

- Il est essentiel de savoir si on doit renvoyer un nouvel objet ou un input modifié. En effet, si le type de celui-ci est mutable, il pourrait y avoir partage de référence.
  - NB : c'est important même si le tableau en input ne contient aucun entier supérieur à n

Inputs implicites
Exemple

- Exemple de procédure sans inputs explicites mais avec deux inputs implicites

```
/**
 * @requires System.in contient une ligne de texte
 * @modifies System.in et System.out
 * @effects lit une ligne dans System.in, positionne le curseur
 *          à la fin de System.in et copie la ligne dans System.out
 */
public static void copyLine () {
   ...
}
```

Approche incrémentale et itérative

- On commence par écrire la spécification des procédures
- Une fois celles-ci fixées, les programmeurs se répartissent les tâches d'implémentation
- Celles-ci peuvent nécessiter l'implémentation de procédures auxiliaires à une procédure donnée (ou à un ensemble de procédures contenues dans la même classe)
- Les procédures auxiliaires ne doivent pas être connues des autres programmeurs. On leur donnera la visibilité **`private`**

Approche incrémentale et itérative

- Exemple :

```
/**
 * @requires a != null
 * @modifies a
 * @effects Réarrange les éléments de a par ordre croissant.
 *          Par exemple, si a = [3,1,6,1], a_post = [1,1,3,6]
 */
public static void sort (int[] a) {
   // implémentation 1, « bubblesort »
   ...
}
```

Approche incrémentale et itérative

```
/**
  * @modifies a
  * @effects Réarrange les éléments de a par ordre croissant.
  *          Par exemple, si a = [3,1,6,1], a_post = [1,1,3,6]
  */
public static void sort (int[] a) {
   ... // implémentation 2, « quicksort »
}

/**
  * @requires a n'est pas null, 0 <= low et high < a.length()
  * @modifies a
  * @effects a_post[low],..., a_post[high] sont triés par ordre croissant
  */
private static void quicksort (int[] a, int low, int high) {
   ...
}

/**
  * @requires a n'est pas null, 0 <= i < j < a.length()
  * @modifies a
  * @effects Réarrange les éléments de a en 2 groupes contigus
  *   a_post[i],..., a_post[res] et a_post[res+1],..., a_post[j] t.q.
  *   chaque élément du 2ème groupe est >= à tous les éléments du 1er.
  *   a[x]=a_post[x] pour tout x t.q. 0<x<i ou j<x<a.length()
  * @return res
  */
private static int partition (int[] a, int i, int j) {
   ...
}
```

Exemple

```
/**
 * @overview Provides useful standalone procedures for manipulating ArrayList
 */
public class ArrayListHelper {
     /**
     * @requires All elements in al are not null.
     * @modifies al
     * @effects Removes all duplicate elements from al
     *          (but preserves the first occurrence).
     *  Uses equals to determine duplicates. The order of remaining elements may change.
     */
     public static <T> void removeDupls (ArrayList<T> al) {
          if (al==null) return;
          for (int i=0; i < al.size(); i++) {
               T x = al.get(i);
               int j = i + 1;
               // remove all duplicates of x from the rest of al
               while (j < al.size()) {
                    if (!x.equals(al.get(j))) { j++; }
                    else {
                         al.set(j, al.get(al.size()-1));
                         al.remove(al.size()-1);
                    }
               }
          }
     }
}
```

Conception d'abstractions procédurales

- Les procédures doivent être **modulaires**
  - Il faut décomposer mais pas trop
  - Ex : Quelle serait l'utilité de décomposer d'avantage `quicksort` ?

- Les procédures doivent être **simples**
  - elle doivent avoir une utilité bien définie
  - facilement exprimable
  - indépendante du contexte d'exécution
  - Test : Est-il facile de donner à la procédure un nom significatif?

Conception d'abstractions procédurales

- Les spécifications doivent être **minimales**
  - une spécification est plus minimale qu'une autre si elle contient moins de contraintes
  - idée : avoir des spécifications contraignantes mais pas trop
    - juste ce qu'il faut pour être sûr d'obtenir l'effet escompté
    - pas trop pour laisser la plus grande liberté au programmeur de trouver la meilleure solution
  - habituellement, l'obligation d'utiliser tel ou tel algorithme ne doit pas faire partie de la spécification (sauf s'il n'y a qu'une méthode numérique particulière qui peut résoudre le problème)

Conception d'abstractions procédurales

- Une spécification peut être **sous-déterminée** (underdetermined)
  - une spécification est sous-déterminée si, pour certains inputs, il y a plus d'un output autorisé
  - Exemple : `removeDupls`
- La sous-détermination peut être voulue ou non. Dans ce dernier cas, c'est une erreur de spécification.
- Ce qui importe c'est que ce qui est pertinent pour l'utilisateur de la procédure figure dans la spécification mais tout le reste ne doit pas être défini (et doit « ne pas être défini »!)
- Toutefois, une spécification sous-déterminée a généralement une implémentation *déterministe* (i.e. exécutée deux fois sur les mêmes inputs, elle fournit les mêmes outputs)

Conception d'abstractions procédurales

- Procédures **totales** vs procédures **partielles**
  - Une procédure est totale ssi sa spécification donne un comportement bien défini pour chaque input vérifiant sa signature
  - Une procédure est partielle dans le cas contraire

- **Les procédures partielles sont plus dangereuses** que les procédures totales
  - Lorsqu'on les appelle sans satisfaire la clause `@requires`, aucune garantie n'est fournie quant à leur comportement
    - elle peuvent boucler indéfiniment, « planter », renvoyer un output quelconque
    - sauf pour les plantages, il est possible qu'on ne se rende compte du problème que longtemps après...

Conception d'abstractions procédurales

- Néanmoins, les procédures partielles sont souvent plus efficaces que les procédures totales
- Exemple
  - si `searchSorted` devait pouvoir travailler sur des tableaux non triés, on n'aurait pas d'autres choix que le parcours exhaustif du tableau
  - Hormis pour de très petits tableaux, le parcours exhaustif est beaucoup moins efficace qu'une recherche dichotomique

Conception d'abstractions procédurales

- Comment choisir entre procédure totale ou partielle ?
- Il n'y a pas de règle générale
- Il s'agit de trouver le bon **compromis** entre **sûreté** et **efficacité** (et éventuellement encore d'autres qualités)
- Conseils
  - si le contexte d'utilisation de la procédure est très large (ex : bibliothèque de fonctions mathématiques réutilisables), la **sûreté** est très importante
  - si le contexte d'utilisation est plus restreint (ex : les procédures `quicksort` et `partition`), on peut plus facilement s'assurer que tous les appels vérifient **`@requires`** et donc la recherche d'**efficacité** devient moins risquée

Conception d'abstractions procédurales

- Remarque
  - ne pas devoir garantir le résultat quand le `@requires` n'est pas satisfait n'empêche pas de vérifier s'il est satisfait ou non
  - si le `@requires` n'est pas satisfait, on peut produire un message d'erreur, renvoyer une valeur conventionnelle ou, mieux, générer une exception (voir chapitre suivant)
  - si cette vérification est coûteuse, cela n'a pas de sens de la faire. Par contre, si elle n'est pas coûteuse, cela en vaut la peine
    - Quid des exemples:
      - `searchSorted`
      - `removeDupls`