Généricité

(un autre outil de réutilisation)

- code paramétré par le type de données manipulées.
- construction de structures homogènes, avec extraction ne nécessitant pas de "cast".

Exemple:

```
public class Pair <U, V> {
    private U first;
    private V second;
    public Pair () { }
    public Pair (U u, V v) { first = u; second = v; }
    public U getFirst() { return first; }
    public void setFirst(U u) { first = u; }
    ...
}
```

- U et V représentent les types de données manipulées.
- Une classe (ou interface) peut être paramétrée par un ou plusieurs paramètres de type.
- Un paramètre de type est utilisé dans le corps de la classe comme un type habituel.
- Une classe générique est instanciée en donnant une valeur (ie: un type) aux paramètres.
  La valeur doit être un type référence (ie: ne peut être primitif)

  Pair < String, Integer > est une classe "habituelle"
  avec les constructeurs: . Pair < String, Integer > ()
  . Pair < String, Integer > (s, i)
  et les méthodes: . String getFirst()
  . void setFirst (s) ...

- Une classe générique peut être vue comme un modèle ("patron") servant à la construction de classes ordinaires.

. les classes et les interfaces dans "java.util" sont génériques:
List, ArrayList, Iterator, Comparator, ---
donc: les types de données sont contrôlés à la compilation par le paramètre de la structure.

```
List < String > l = new ArrayList <> (); ---
String s = l.get(0); // pas de "cast" nécessaire.
l.add (10); // refusé à la compilation
            // ( type non conforme)
```

Généricité contrainte.

. Les appels de caractéristique dont la cible est une variable de type paramètre, ne peut concerner que les méthodes de la classe "Object".
. On peut elargir le choix des méthodes possibles en imposant des contraintes au paramètre type:

E extends C1 & I1 & I2 --

- Un parametre peut être borné par une classe (qui doit être la première de la liste) et plusieurs interfaces.
- L'argument type doit être alors un sous-type de la classe et implémenter les différents interfaces.

Ex:
si class A < E extends Comparable < E > > { .. }
alors . pour qu'une instanciation A < B > soit valide, il faut que: B implémente l'interface Comparable < B >
. "compareTo" est utilisable dans le corps de la classe générique A.

Méthodes génériques.

A l'instar des classes, on peut paramétrer une méthode par le type de données qu'elle manipule.
Une méthode peut être générique dans une classe générique ou dans une classe non générique.
Un paramètre de méthode, comme pour une classe, peut être borné.

```
public class xx {
    public static <T> T mediane (T[] a){
        return a [a.length /2]; }
    ...
}
```

appel:
```
String noms = {...}
String mil = xx.<String> mediane (noms);
```

Rq: On peut, en général, omettre l'argument devant le nom de la méthode, s'il peut être déduit du contexte.

Code générique et machine virtuelle ("Erasure")

. La machine virtuelle n'a pas de type générique.
. Quand on définit un type générique, il lui est automatiquement associé un type dépouillé ("raw type"):
  . le nom du type dépouillé est celui du type générique sans les paramètres.
  . dans le corps, les variables types sont effacées et remplacées par leur borne (la première s'il y en a plusieurs, "Object" s'elles ne sont pas bornées) avec l'insertion de "cast" là où c'est nécessaire.

Conséquences:

. il y a un code unique pour chaque classe générique.
(et non pas un code par valeur de paramètre)

- le code défini avant la généricité reste valide en considérant le type dépouillé (mais avertissement).
- les tests dynamiques de type (instanceof, getClass) sont relatifs aux types dépouillés.

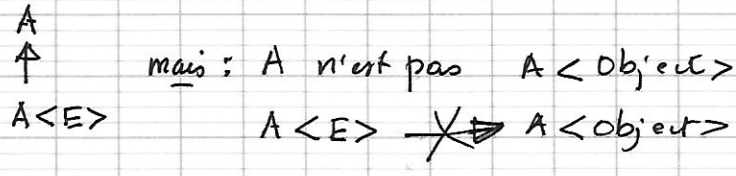
Restrictions:

- La valeur passée à un paramètre de type ne peut être d'un type primitif.
- On ne peut instancier un paramètre générique (new T() // illégal)
- On ne peut référencer un paramètre d'une classe générique dans un contexte "statique" (méthode, class imbriquée ...)
- On ne peut créer de tableaux de types paramétrés.

Généricité et Héritage.

- 

- Un type paramétré est un sous type de son type dépouillé.



- Une classe générique peut etendre (implémenter) d'autres classes (interfaces).

Jockers ("WildCards")

Pour définir une méthode permettant d'afficher les éléments d'une liste d'un type quelconque, on peut penser à définir la méthode :

```
class xx {
    public static void printList(PrintWriter out,
                                 List<Object> l){
        for(int i=0; i< l.size(); i++)
            out.println(l.get(i).toString());}
}
```

alors : "printList" ne peut être appelée avec un argument "List<Integer>" ou "List<String>" puisqu'ils ne sont pas des sous-types de "List<Object>".

Remède :

Déclarer "l" comme étant une liste d'"inconnus" :

```
List<?>
```

et alors

. Tout type List est compatible avec List<?>

mais

. Un objet de type "List<?>" est un objet en "lecture seule" : les méthodes telles que add, set... ne leur sont pas applicables (sauf pour 'null').

. l'extraction retourne "Object".

. On ne peut instancier un type inconnu :

```
List<?> l = new ArrayList<?>(); //illégal
List<?> l = new ArrayList<Integer>(); //légal.
```

Jockers bornés.

Un jocker peut être borné :

- supérieurement : ? extends A

  et alors C<? extends A>

  ↑

  C<B>

  si B ∈ [NullType, A] (covariance)

- inférieurement : ? super A

  C<? super A>

  ↑

  C<B>

  si B ∈ [A, Objet] (contravariance)

Rq ? ⟺ ? extends Objet.

Les ~~jockers~~ structures avec des jockers bornés :

- supérieurement sont des structures en lecture seule.
- inférieurement sont des structures en écriture seule.

Principe Get/Set :

. Utiliser les "jocker_extends" quand la structure est utilisée en lecture seule.

. Utiliser les "jocker_super" quand la structure est utilisée en écriture seule.

. Ne pas utiliser de jocker quand la structure est utilisée dans les deux modes.