Chapitre 5 : Types avancés

- Enum
- Generics,
- Collections
- Stream

Les types énumérés

- Première solution : créer une classe contenant uniquement des attributs static :

```
public class Jour {
   public static final String LUNDI = « lundi »;
   …
   public static final String DIMANCHE=
     « dimanche »;

   }
```

  - Inconvénients :
    - Pas de contrôle du type
    - Pas d'itération possible

Les types énumérés

- Bonne solution : créer un type **enum**

```
public enum Jour{
  LUNDI, MARDI, …, DIMANCHE
}
```

- Utilisation :

```
public class MaClasse{
   private Jour leJour = Jour.LUNDI;
   …
}
```

Les types énumérés

- Iteration :

```
for (Jour j : Jour.values()){
   …
}
```

- Ajout de méthodes :

```
public enum Jour {
 LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI,
 DIMANCHE;

 public Jour lendemain(){
    Jour[] semaine = Jour.values();
    return (semaine[(this.ordinal()+1) %
              semaine.length]);
 }
}
```

Les types énumérés

```
public enum Devise {
   Livre(0.872), YEN(105.949), DOLLAR(1.378),  ROUBLE(42.092);
   private double taux;

   Devise(double d){
        this.taux=d;
   }

   public double tauxCourant(){
        return this.taux;
   }

   public void changeTaux(double t){
    this.taux=t;
   }

   public double convertInEuros(double val){
    return (val/this.taux);
   }
}
```

Les Generics

- Il arrive que l'on ait une classe dont un attribut (ou une méthode) puisse avoir différents types (qui n'ont pas nécessairement de lien hiérarchique).

Première approche : utilisation d'Object

```
public class MyClass {
    Object elt;

    public MyClass(Object o){
        this.elt=o;
    }

    public Object getElt(){
        return this.elt;
    }
}
```

Les Generics

- **Inconvénients** :
  - Cast obligatoire

```
MyClass obj1 = new MyClass (2);
MyClass obj2 = new MyClass ("abc");
Integer x = (Integer)obj1.getElt();
```

  - Pas de typage fort à la compilation

```
Integer y = (Integer)obj2.getElt();
//erreur à l'exécution
```

Les Generics

Seconde approche : classe générique

```
public class MyGenClass<E> {
    private E elt;

     public MyGenClass (E e){
         this.elt=e;
     }

    public E getElt(){
        return this.elt;
     }
}
```

Les Generics

- **Avantages** :
  - Inférence du type (plus besoin de cast)

```
MyGenClass<Integer> obj1 = new MyGenClass<>(2);
MyGenClass<String> obj2 = new MyGenClass<>("abc");
Integer x = obj1.getElt();
```

  - typage fort à la compilation

```
 Integer y = obj2.getElt();
//erreur à la compilation
```

Les Generics

- **Attention** : Carre hérite de Rectangle, MAIS List<Carre> n'hérite pas de List<Rectangle>.

```
Rectangle r = new Carre(4); // pas de pb

List<Rectangle> lr = new ArrayList<Carre>();
                        // error : incompatible types
```

Les Generics

- **Attention** : Carre hérite de Rectangle, MAIS List<Carre> n'hérite pas de List<Rectangle>.

```
public static void printList(List<GeometricShape> l){
   l.forEach(x -> x.print());
}

List<Rectangle> lr = new ArrayList<>();
printList(lr); // erreur à la compilation
```

Les Generics

- Solution : utilisation de **wildcard**, noté ?

```
public static void printList(List<? extends GeometriShape> l)
{
        l.forEach(x -> x.print());
}
```

`List<? extends FormeGeometrique> l`

Se lit : « Une liste d'objets qui héritent de FormeGeometrique »

Les Generics

- Un paramètre de type générique défini à l'aide d'une wildcard de type <? extends… > ne peut pas être modifié.

```
public static void modif(List<? extends FormeGeometrique> l){
                l.add(new Carre(2)); // error à la compilation

    }
```

Les Generics

- On peut aussi utiliser les wildcards pour restreindre la sous-classe d'objets admis dans une collection :

```
public static void modif(List<? super Carre>
l){
      l.add(new Carre(2)); // OK
       }
}
```

- `List<? super Carre>` Se lit « Une liste d'objets dont Carre hérite (directement ou non) »

  On pourra donc ajouter des carrés, mais pas des Cercles

Java Collections Framework

Une collection est un objet qui permet de manipuler un groupe d'objets.

Il existe différents types de collections, qui permettent :

- de manipuler des groupes d'objets ordonnés ou non,
- d'autoriser les doublons ou non.
- d'avoir l'objet null ou non.

Java Collections Framework

- Deux interfaces génériques :
  - Collection<E> :
    - List<E>
    - Set<E>
    - …
  - Map<K,V> :
    - HashMap<K,V>
    - Bindings<String,Object>
    - …

Java Collections Framework

- Implémentations (List, Set, Map) :
  - Dépend de l'utilisation.
  - Exemple : Arraylist et LinkedList
    - ArrayList : un tableau redimensionnable.
      + accès direct à un élément
      – ajout/suppression d'éléments
    - LinkedList : structure doublement chaînée
      + ajout/suppression d'éléments
      – Accès à un élément nécessite un parcours

Java Collections Framework

- Déclaration et instanciation :

```
– Set<Car> mySet = new HashSet<>();

–  List<Car> myList = new ArrayList<>(10);
–  List<Car> myList2 = new LinkedList<>();

– Map<Car,Integer> myMap = new HashMap<>();
```

Java Collections Framework

- Principales méthodes de l'interface Collection : http://download.oracle.com/javase/7/docs/api/

  - `boolean add(E e)`
  - `boolean addAll(Collection<? extends E> c)`

    Ajout d'un élément e (ou d'une collection c) à la collection. Le boolean indique si oui ou non la collection a été modifiée suite à l'appel de la méthode.

  - `void clear()`

    Vide la collection

  - `boolean  contains(Object o)`
  - `boolean  containsAll(Collection<E> c)`

    Retourne vrai si la collection contient o

  - `boolean isEmpty()`

    Retourne vrai si la collection est vide

Java Collections Framework

- Principales méthodes de l'interface Collection : http://download.oracle.com/javase/7/docs/api/

  - `Iterator<E>  iterator()`
    Retourne un itérateur sur la collection

  - `boolean remove(Object o)`
  - `boolean removeAll(Collection<? extends E> c)`
    Suppression d'un élément (ou d'un collection). La fonction retourne vrai si la collection a été modifiée suite à l'appel de la méthode.

  - `boolean retainAll(Collection<E> c)`
    Ne garde que les éléments qui font partie de la collection c

  - `int size()`
    Retourne le nombre d'éléments de la collection

  - `Object[] toArray`
    Retourne un tableau correspondant à la collection

Java Collections Framework

- Un mot sur les listes :
  - Méthodes spécifiques :

  - `void add(int index, E element)`
    Ajoute element en position index

  - `Boolean addAll(int index, Collection<? extends E> c)`
    Ajoute la collection c en commençant à la position index

  - `E get(int index)`
    Retourne l'élément en position index

  - `int indexOf(Object o)`
    Retourne l'index de l'élément o, si o n'appartient pas à la liste, alors la méthode retourne -1;

Java Collections Framework

- Un mot sur les listes :
  - Méthodes spécifiques :

  - `ListIterator<E> listIterator()`
    Retourne un ListIterator, qui permet de se déplacer dans une liste, dans les deux sens.

  - `ListIterator<E> listIterator(int index)`
    idem, mais l'itérateur est initialisé en position index

  - `E set(int index, E element)`
    Remplace l'element en position index

  - `List<E> subList(int fromIndex, int toIndex)`
    Retourne la sous-liste comprise entre les indices fromIndex et toIndex

Java Collections Framework

- Un mot sur les Map:
  - Structure qui permet de manipuler des couples (clé, valeur)

```
Map<Key,Values> m = new HashMap<Keys,Values>();
```

- `put(Object key, Object val)`
  ajout d'un couple (clé, valeur)
- `Object get(Object key)`
  retourne la valeur correspondant à la clé
- `boolean containsKey(Object key)`
- `boolean containsValue(Object value)`
- `Collection values()`
- …

Parcours de collections

- boucle **for each** :

```
List<GeometricShape> l = new ArrayList<>();
for (GeometricShape gs : l){
   …
   }
```

Parcours de collections

- **Iterator** :

```
List<GeometricShape> l = new ArrayList<>();
…
Iterator<GeometricShape> it = l.iterator();
```

  - `it.next()` : retourne le prochain élément de la collection. Le premier appel de la méthode next() retourne le premier élément.
  - `it.hasNext()` : retourne true si l'itérateur n'a pas parcouru toute la collection
  - `it.remove()` : supprime l'élément pointé par l'itérateur

Parcours de collections

- **Iterator** :

```
List<GeometricShape> l = new ArrayList<>();
…
Iterator<GeometricShape> it = l.iterator();
While (it.hasNext()){
   GeometricShape gs = it.next();
   if (gs instanceof Circle){
      it.remove();
   }
}
```

Parcours de collections

- **Iterator** :

```
List<GeometricShape> l = new ArrayList<>();
…
Iterator<GeometricShape> it = l.iterator();
While (it.hasNext()){
    GeometricShape gs = it.next();
    if (gs instanceof Circle){
        it.remove();
    }
}
```

L'utilisation d'un iterator est le seul moyen sûr permettant de supprimer un élément d'une collection lors d'un parcours.

Parcours de collections

- **Interfaces fonctionnelles :**

```
List<GeometricShape> l = new ArrayList<>();
…
l.forEach(x -> x.print());
l.removeIf(x -> x instanceof Circle);
```

Parcours de collections

- **Interfaces fonctionnelles :**

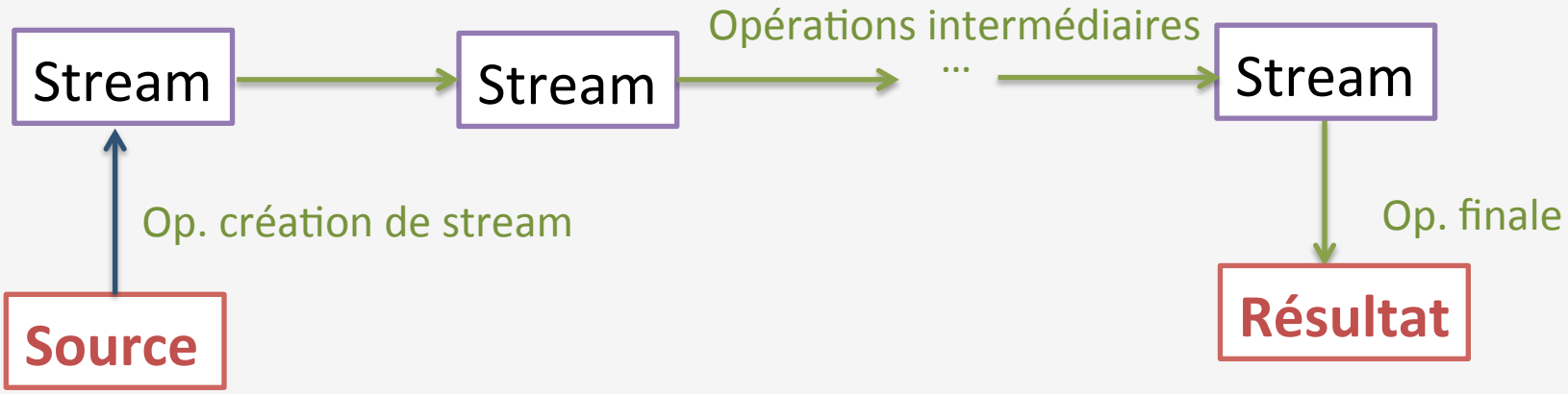
```
List<GeometricShape> l = new ArrayList<>();
…
l.forEach(x -> x.print());
l.removeIf(x -> x instanceof Circle);
```

lambdas

utilise un iterator

Streams

- Permet de définir un pipeline d'opérations à effectuer sur des données provenant d'une source (collection, tableau…)
- Ne modifie pas les données initiales
- Usage unique



Streams

Source

```
List<GeometricShape> l = new ArrayList<>();
```

Création

```
Stream<GeometricShape> s = l.stream();
```

Opérations Intermédiaires

```
filter, map, distinct,…
```

Opérations finales

```
reduce, forEach, collect, count…
```

Streams

Un exemple :

```
List<GeometricShape> l = new ArrayList<>();
Stream<GeometricShape> s = l.stream();


double result =
            s.filter( x -> (x instanceof Circle))
            .map( x -> x.perimeter())
            .reduce(0.0, (x,y) -> x+y );
```

Streams

Un exemple :

```
List<GeometricShape> l = new ArrayList<>();
Stream<GeometricShape> s = l.stream();


double result =
            s.filter( x -> (x instanceof Circle))
            .map( x -> x.perimeter())
            .reduce(0.0, (x,y) -> x+y );
```

Calcule la somme des aires des cercles