L3 SRI - COO

Cours 3 : Généricité

Les différentes version de Java,
Les classes génériques,
Les itérateurs

Auteur : CHAUDET Christelle

Les différentes versions de JAVA

- Java a beaucoup évolué, et les différentes versions sont présentes dans l'industrie. Il est donc important de connaitre ce que contiennent ou non les différentes versions.
- Dans ce cours nous nous intéresserons à :
  - La version 1.2 (collections)
  - La version 1.4 (collections version plus aboutie)
  - La version 1.5 (introduction des génériques)
  - La version 1.7 (simplification de la syntaxe)

Classes génériques

- Depuis Java 5, la bibliothèque JAVA a intégré la généricité.
- Les classes génériques
  - Le constructeur d'une classe générique
- Les méthodes génériques
  - Les méthodes génériques
  - Paramétrage contraint

Classes génériques

- Une classe générique est paramétrée par des types

```
public class Paire<T,U> {
 private final T premier;
 private final U second;

 public Paire(T premier, U second) {
  this.premier = premier;
  this.second = second;
 }

  public T getPremier() {
   return premier;
  }

  public U getSecond() {
   return second;
  }
```

Attention définition du constructeur **non paramétré**

- Les paramètres de type sont passés au constructeur lors de l'invocation : **version 1.5 & 1.6**

```
Paire<String, Integer> paire1 =
                    new Paire<String, Integer>("un",1);
Paire<Integer, String> paire2 =
                    new Paire<Integer, String>(1,"un");
```

Classes génériques

- Une classe générique est paramétrée par des types

```
public class Paire<T,U> {
  private final T premier;
  private final U second;

  public Paire(T premier, U second) {
    this.premier = premier;
    this.second = second;
  }

  public T getPremier() {
    return premier;
  }

  public U getSecond() {
    return second;
  }
}
```

- Simplification d'écriture

version 1.5 & 1.6

```
Paire<String, Integer> paire1 = new Paire<String, Integer>("un",1);
```

version 1.7 & supérieur

```
Paire<String, Integer> paire1 = new Paire<>("un",1);
```


Classes génériques

- Une classe générique est paramétrée par des types

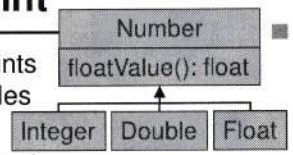
```
public class Paire<T,U> {
  private final T premier;
  private final U second;

  public Paire(T premier, U second) {
    this.premier = premier;
    this.second = second;
  }

  public T getPremier() {
    return premier;
  }

  public U getSecond() {
    return second;
  }
}
```

- Mais getClass() ne prend pas en compte les paramètres de généricité, la généricité étant absente de la JVM.

```
Paire<String, Integer> paire1 = new Paire<>("un",1);
Paire<Integer, String> paire2 = new Paire<>(1,"un");
System.out.println(paire1.getClass().equals(paire2.getClass()));
```

⇨ true

Paramétrage contraint



- Les types génériques peuvent être contraints grâce à des bornes afin, par exemple, de les obliger à fournir un certain service.
- **public class** Addition<T **extends** Number> {
  public float addition(T chiffre1, T chiffre2) {
  return (chiffre1.floatValue() + chiffre2.floatValue());
  } }

```
Addition<Integer> calculateurEntier = new Addition<>();
System.out.println(calculateurEntier.addition(3, 4));

Addition<Double> calculateurDouble = new Addition<>();
System.out.println(calculateurDouble.addition(-3.2, 4.9));

Addition<Float> calculateurFloat = new Addition<>();
System.out.println(calculateurFloat.addition(-3.2f, -4.9f));
```


Méthodes génériques

- Méthode générique : son profil est paramétré par des types.
- Exemple : sélection d'un élément d'un tableau.

```
public class Selection {
  public <T> T choix(int indice, T[] tableau) {
    if (indice >= tableau.length)
        return null;
    return tableau[indice];
  } }
```

Affichage

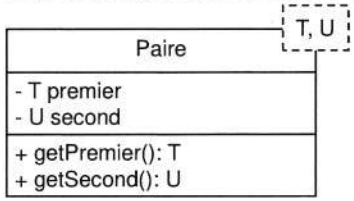
```
1
trois
```

- Appel à la méthode :

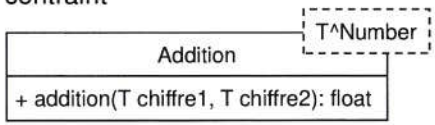
```
Selection selection = new Selection();
Integer valeur = selection.choix(0, new Integer[] {1,2,3});
String chaine = selection.choix(2,
        new String[] {"un","deux","trois"});
```

Représentation UML

- Représentation d'une classe Générique



- Représentation d'une classe générique avec paramétrage contraint



Les itérateurs

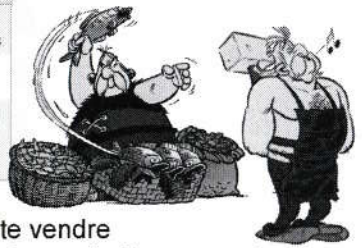
- L'interface **Iterator<E>** a trois méthodes qui permettent de parcourir l'ensemble des éléments d'une classe

Method Summary

java.util
Interface Iterator<E>

Methods

Modifier and Type	Method and Description
boolean	`hasNext()` Returns `true` if the iteration has more elements
E	`next()` Returns the next element in the iteration.
void	`remove()` Removes from the underlying collection the last element returned by this iterator (optional operation).



- Par exemple **Ordralfabétix** souhaite vendre ses poissons les moins frais d'abord pour limiter ses pertes. Son panier qui contient l'ensemble de ses poissons implémente l'interface Iterator<E>.

Les itérateurs

- Nous avons pour cela besoin
  - de comparer des dates:

```
public class Date {
  private int jour, mois, annee;
  public Date(int jour, int mois, int annee) {
    this.jour = jour; this.mois = mois;
    this.annee = annee;
  }
  public int getAnnee() { return annee;}
  public int getJour() { return jour; }
  public int getMois() { return mois; }
  public boolean precede(Date date) {
    int annee = date.getAnnee(); int mois = date.getMois(); int jour = date.getJour();
    if (this.annee < annee) return true;
    else if (this.annee == annee && this.mois < mois) return true;
    else if (this.annee == annee && this.mois == mois && this.jour < jour)
        return true;
    return false;
} }
```

  - de poissons

```
public class Poisson {
  private Date datePeche;
  public Poisson(Date datePeche) {
    this.datePeche = datePeche;
  }
  public Date getDatePeche() {
    return datePeche;
  }
}
```

Les itérateurs

- La classe « Panier » implémente l'interface Iterator.

```
public class Panier implements Iterator<Poisson> {
  private Poisson[] panier = new Poisson[4];
  private int nombrePoisson = 0;
  private int indiceIterateur = 0;

  public void ajouterPoisson(Poisson poisson) {
    panier[nombrePoisson] = poisson;
    nombrePoisson++;
  }

  public String afficherPanier() {
    String chaine = "Le panier est vide !";
    if (nombrePoisson > 0) {
      chaine = "Le panier contient les poissons : \n";
      for (int i = 0; i < nombrePoisson; i++) {
        chaine += "- " + panier[i] + "\n";
      }
    }
    return chaine;
  }
```

Les itérateurs

```
public class Panier implements Iterator<Poisson> {
  private Poisson[] panier = new Poisson[4];
  private int nombrePoisson = 0;
  private int indiceIterateur = 0;
    ...
  public boolean hasNext() {
    return nombrePoisson != 0
        && indiceIterateur<nombrePoisson;
  }
```

java.util

Interface Iterator<E>

Modifier and Type	Method and Description
boolean	`hasNext()` Returns `true` if the iteration has more elements.
E	`next()` Returns the next element in the iteration.
void	`remove()` Removes from the underlying collection the last element returned by this iterator (optional operation).

Les itérateurs

```
public class Panier implements Iterator<Poisson> {
 private Poisson[] panier = new Poisson[4];
 private int nombrePoisson = 0; private int indiceIterateur = 0;
  ...
 public Poisson next()
                 throws NoSuchElementException {
  if (hasNext()) {
    Date date = panier[indiceIterateur].getDatePeche();
    int indicePoisson = indiceIterateur;
    for (int i = indiceIterateur+1; i < nombrePoisson; i++) {
      Date datePeche = panier[i].ge
      if (datePeche
                .precede(date)) {
        date = datePeche;
        indicePoisson = i;
    } }
```

Modifier and Type	Method and Description
E	`next()` Returns the next element in the iteration.

```
    Poisson poisson = panier[indicePoisson];
    panier[indicePoisson] = panier[indiceIterateur];
    panier[indiceIterateur] = poisson;
    indiceIterateur++;
    return poisson;
    }
    throw new NoSuchElementException();
}
```

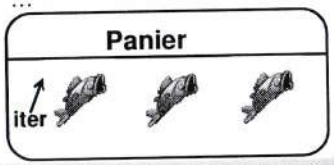
Les itérateurs

```
public class Panier implements Iterator<Poisson> {
 private Poisson[] panier = new Poisson[4];
 private int nombrePoisson = 0;
 private int indiceIterateur = 0;
  ...
 public void remove() throws IllegalStateException{
  if (nombrePoisson < 1)
   throw new IllegalStateException();

  for (int i = indiceIterateur - 1; i < nombrePoisson - 1; i++) {
    panier[i] = panier[i + 1];
  }
  indiceIterateur--;
  nombrePoisson--;
}
```

Modifier and Type	Method and Description
boolean	`hasNext()` Returns `true` if the iteration has more elements.
E	`next()` Returns the next element in the iteration.
void	`remove()` Removes from the underlying collection the last element returned by this iterator (optional operation).

Les itérateurs

```
public class Panier
implements Iterator<Poisson> {
  ...
}
```



java.util

Interface Iterator<E>

Methods

Modifier and Type	Method and Description
boolean	`hasNext()` Returns `true` if the iteration has more elements.
E	`next()` Returns the next element in the iteration.
void	`remove()` Removes from the underlying collection the last element returned by this iterator (optional operation).

```
Panier panier = new Panier();
panier.ajouterPoisson(new Poisson(new Date(12, 5, 2017)));
panier.ajouterPoisson(new Poisson(new Date(14, 5, 2017)));
panier.ajouterPoisson(new Poisson(new Date(10, 5, 2017)));
for (;panier.hasNext();) {
  System.out.println("hasNext() : " + panier.hasNext());
  System.out.println("next() : " + panier.next());
  panier.remove();
}
System.out.println("hasNext() : "
                   + panier.hasNext());
```

```
hasNext() : true
next() : poisson péché le 10/5/2017
hasNext() : true
next() : poisson péché le 12/5/2017
hasNext() : true
next() : poisson péché le 14/5/2017
hasNext() : false
```

Les itérateurs

```
public class Panier
implements Iterator<Poisson> {
   ...
}
```

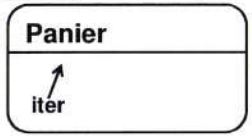
java.util
Interface Iterator<E>

Methods

Modifier and Type	Method and Description
boolean	hasNext() Returns true if the iteration has more elements.
E	next() ...in the iteration
	...ying collection the last ...erator (optional

```
hasNext() : false
java.util.NoSuchElementException
        at marche.PanierIterator.next(PanierIterator.java:46)
        at marche.PanierIterator.main(PanierIterator.java:105)
```

```
Panier panier = new Panier();
panier.ajouterPoisson(new Poisson(new Date(12, 5, 2017)));
panier.ajouterPoisson(new Poisson(new Date(14, 5, 2017)));
panier.ajouterPoisson(new Poisson(new Date(10, 5, 2017)));
for (;panier.hasNext();) {⇨ while
  System.out.println("hasNext() : " + panier.hasNext());
  System.out.println("next() : " + panier.next());
  panier.remove();
}
System.out.println("panier.hasNext() : " + panier.hasNext());
System.out.println("panier.next() : " + panier.next());
```



Les itérateurs

```
public class Panier
implements Iterator<Poisson> {
   ...
}
```

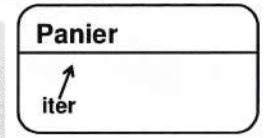
java.util
Interface Iterator<E>

Methods

Modifier and Type	Method and Description
boolean	hasNext() Returns true if the iteration has more elements.
E	next() ...the iteration
	...ng collection the last ...rator (optional

```
hasNext() : false
Exception in thread "main" java.lang.IllegalStateException
        at marche.PanierIterator.remove(PanierIterator.java:51)
        at marche.PanierIterator.main(PanierIterator.java:107)
```

```
Panier panier = new Panier();
panier.ajouterPoisson(new Poisson(new Date(12, 5, 2017)));
panier.ajouterPoisson(new Poisson(new Date(14, 5, 2017)));
panier.ajouterPoisson(new Poisson(new Date(10, 5, 2017)));
for (;panier.hasNext();) {
  System.out.println("hasNext() : " + panier.hasNext());
  System.out.println("next() : " + panier.next());
  panier.remove();
}
System.out.println("panier.hasNext() : " + panier.hasNext());
panier.remove();
```



Classe Iterable

- Pour pouvoir manipuler plus facilement les objets qui implémentent l'interface « Iterator », Java propose une interface qui récupère l'itérateur de la classe.

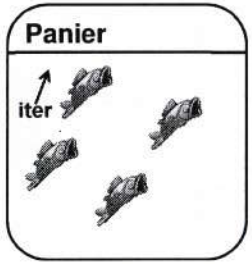
java.lang
Interface Iterable<T>

Method Summary

Iterator<T>	iterator() Returns an iterator over a set of elements of type T.

- L'interface **Iterable** ne possède qu'une seule méthode qui retourne un itérateur. Iterator<T> iterator()

  Un itérateur permet de parcourir l'ensemble des éléments d'une classe implémentant l'interface « Iterator » grâce aux méthodes hasNext, next, remove.



Classe Iterable

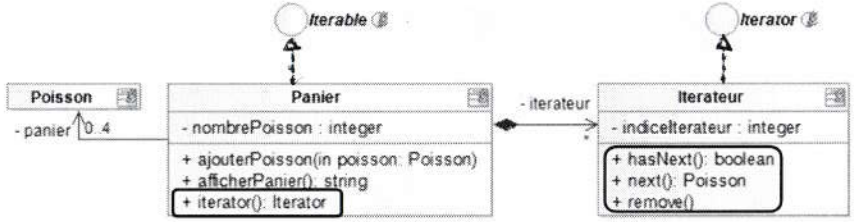
java.lang
Interface Iterable<T>

Method Summary

Iterator<T>	iterator() Returns an iterator over a set of elements of type T.

- Notre classe « Panier » implémente l'interface « Iterator », mais nous pouvons le parcourir qu'une seule fois. Pour pouvoir le parcourir autant de fois que nécessaire il faut qu'elle soit itérable.

- Pour cela nous allons créer une classe interne qui implémentera l'interface « Iterator », et rendrons la classe « Panier » itérable, c'est-à-dire qu'elle implémentera la méthode iterator() retournant un objet de type Iterator.



Classe Iterable

```
public class Panier implements Iterable<Poisson> {
  private Poisson[] panier = new Poisson[4];
  private int nombrePoisson = 0;
  public void ajouterPoisson(Poisson poisson) {...}
  public String afficherPanier() {...}
  public Iterator<Poisson> iterator() {
    return new Iterateur();
  }
  private class Iterateur implements Iterator<Poisson> {
    private int indiceIterateur = 0;
    public boolean hasNext() {...}
    public Poisson next() {...}
    public void remove() {...}
  }
}
```

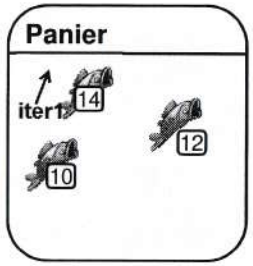
java.lang
Interface Iterable<T>

Method Summary

Iterator<T>	iterator() Returns an iterator over a set of elements of type T.

Classe Iterable

- Prenons en considération le problème précédent en respectant les contraintes suivantes :
  - Le panier doit pouvoir être parcouru plusieurs fois,
  - Le panier doit pouvoir être parcouru par plusieurs itérateurs en même temps (à condition qu'il ne modifie pas le panier).




```
Panier panier = new Panier();
panier.ajouterPoisson(new Poisson(new Date(12, 5, 2017)));
panier.ajouterPoisson(new Poisson(new Date(14, 5, 2017)));
panier.ajouterPoisson(new Poisson(new Date(10, 5, 2017)));
Iterator<Poisson> iter1 = panier.iterator();
iter1.hasNext(); iter1.next();
iter1.hasNext(); iter1.next();
Iterator<Poisson> iter2 = panier.iterator();
iter2.hasNext(); iter2.next();
iter2.hasNext(); iter2.next();
iter1.hasNext(); iter1.next();
iter2.hasNext(); iter2.next();
iter2.remove();
```


Classe Iterable

- Pour ne pas avoir le genre de problème que l'on vient de voir précédemment, on énonce la règle suivante : on peut avoir :
  - autant d'itérateur en lecture du panier que l'on souhaite,
  - un seul itérateur attaché au panier dans le cas de suppression ou d'ajout de poisson.
- Nous allons donc utiliser un nouvel attribut *nombrePoissonReference* dans la classe interne « Iterateur » qui sera initialisée avec le nombre de poissons du panier. Lorsqu'un objet de la classe itérateur supprimera un poisson ce nombre sera mis à jour mais seulement pour la référence en cours.
- Avant chaque méthode on vérifiera que le nombre de poissons du panier et *nombrePoissonReference* sont égaux sinon on lèvera l'exception ConcurrentModificationException

Classe Iterable

- Ajoutons le traitement pour la cohérence des paniers.

```
private class Iterateur implements Iterator<Poisson> {
  private int indiceIterateur = 0;
  private int nombrePoissonReference = nombrePoisson;

  public boolean hasNext() throws ... { verificationConcurrence(); ...}
  public Poisson next() throws ... {...}
  public void remove() throws ... {
    verificationConcurrence();
    ...
    nombrePoissonReference--;
  }
  private void verificationConcurrence() throws ConcurrentModificationException {
    if (nombrePoisson != nombrePoissonReference)
      throw new ConcurrentModificationException();
  }
}
```

```
Exception in thread "main" java.util.ConcurrentModificationException
        at testCours2.Panier$Iterateur.verificationConcurrence(Panier.java:76)
        at testCours2.Panier$Iterateur.hasNext(Panier.java:39)
        at testCours2.TestPoisson.main(TestPoisson.java:30)
```

Boucle foreach

- Toute classe implémentant l'interface **Iterator** peut être utilisée dans une instruction **foreach** ainsi que les tableaux
  - Parcours d'un tableau

```
int[] tableau = {1,2,3};
for (int chiffre : tableau) {
   System.out.println(chiffre);
}
```

  - Parcours d'une classe itérable

```
Panier panier =  new Panier();
panier.ajouterPoisson(new Poisson(new Date(12, 5, 2017)));
panier.ajouterPoisson(new Poisson(new Date(14, 5, 2017)));
panier.ajouterPoisson(new Poisson(new Date(10, 5, 2017)));

for (Poisson poisson : panier) {
  System.out.println(poisson);
}
```


Boucle foreach

- Mais attention : vous ne devez pas utiliser une boucle foreach si vous modifiez une classe itérable.

```
for (Poisson poisson : panier) {
  System.out.println(poisson);
  if (poisson.perime()) panier.remove(poisson);
}
```

OU

```
for (Poisson poisson : panier) {
  System.out.println(poisson);
  if (poisson.perime()) panier.add (new Poisson(new Date(15, 5, 2017)));
}
```

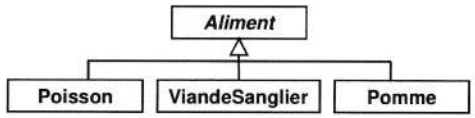
⇩

```
Exception in thread "main" java.util.ConcurrentModificationException
        at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:819)
        at java.util.ArrayList$Itr.next(ArrayList.java:791)
        at test.Cours3.main(Cours3.java:33)
```


Classe Iterable

- Notre classe Panier nous semble maintenant correcte, mais au fait, comment faire pour que l'on puisse avoir soit un panier de poissons, soit un panier de viande de sanglier, soit un panier de pommes …

  Mais attention pas de panier contenant tout à la fois !



Eh oui utilisons la généricité !

```
public class Panier<A extends Aliment> implements Iterable<A> {
  private A[] panier;
  private int nombreAliment = 0;

  public Panier(A[] panier) {  this.panier = panier; }
  public void ajouterAliment(A aliment) {…}
  public String afficherPanier() {…}
  public Iterator<A> iterator() {
    return new Iterateur();
  }

  private class Iterateur implements Iterator<A> {
    private int nombreAlimentReference = nombreAliment;
    private int indiceIterateur = 0;

    public boolean hasNext() throws … { …}
    public A next() throws … {…}
    public void remove() throws … {…}
    private void verificationConcurrence() throws … {…}
  }
}
```