Les lambdas

Introduction

- Java est une langage orienté objet
  - tout est objet
  - pas de fonction en dehors d'un classe
- Le passage d'un traitement en tant que paramètre est souvent effectué par une classe anonyme

```
new Thread(new Runnable() {

    @Override
    public void run() {
        // TODO Auto-generated method stub

    }
}).start();
```


(cc) BY-NC-SA

Introduction

- Java 8 ajoute les expressions lambda au langage
  - JSR 335
  - une expression lambda est une fonction anonyme
    - c'est à dire qui n'a pas de nom
    - elle est écrite à l'endroit où elle est nécessaire
      - exemple : lambda comme paramètre d'une autre fonction/méthode
    - elle est exécutée dans le contexte où elle apparaît
    - peut être stockée dans une variable, retournée par une méthode
  - allège le passage de traitement en tant que paramètre

```
new Thread(()-> {}).start();
```


(cc) BY-NC-SA

Introduction

- L'intégration des lambdas en Java 8
  - s’appuie sur les interfaces
    - interface fonctionnelle, ne possédant qu'une seule méthode
  - n'ajoute pas de type au langage
  - une expression lambda n'est pas transformée en classe par le compilateur
    - pas de classe anonyme interne

(cc) BY-NC-SA

Introduction

- La définition d'une fonction lambda est effectuée sans déclaration explicite du type de retour
- C'est un raccourci syntaxique
  - simplifie l'écriture des traitements passés en tant que paramètres
  - l'évaluation de la lambda est effectuée par le compilateur par inférence de type avec l'interface fonctionnelle attendue

```
new Thread(()-> {}).start();
```

une implémentation de `Runnable` est attendue

(cc) BY-NC-SA

Syntaxe

- Syntaxe très simple
  - paramètres
  - opérateur flèche ->
    - sépare les paramètres et les traitements
  - corps de la fonction

- Deux formes principales

```
(paramètres) -> expression
```

```
(paramètres) -> { traitements ; }
```


(cc) BY-NC-SA

Syntaxe

- Règles d'écriture d'une fonction Lambda
  - paramètres
    - aucun ou plusieurs
    - le type peut être déclaré ou inféré par le compilateur
    - entouré par des parenthèses ()
      - si un seul paramètre dont le type est inféré, les parenthèses ne sont pas obligatoires
  - corps
    - aucune, une ou plusieurs instructions
    - si une seule instruction
      - les accolades {} sont facultattives
      - le type de retour correspond à celui de l'instruction

(cc) BY-NC-SA

Syntaxe

- Exemples

```
() -> System.out.println("hi")

(x,y) -> x+y

(Point p, int x) -> p.add(x)

Arrays.asList(tabString).sort((e1,e2)->e1.compareTo(e2));

(x,y) -> {if(x<y)
          return y;
        else
          return x;}

Operation op = (x,y) -> x % y;
```


(cc) BY-NC-SA

Portées des variables

- L'expression lambda se comporte syntaxiquement comme un bloc de code imbriqué
  - la lambda peut avoir accès à certaines variables dans le contexte englobant (closure)
  - il est donc possible d'utiliser dans le corps de la lambda
    - variables passées en paramètre de la lambda
    - variables définies dans le corps de la lambda
    - variables définies dans le contexte englobant
      - **seules les variables dont la valeur ne change pas peuvent être utilisées**
      - si ces variables ne sont pas déclarées `final`, une valeur leur sera assignée et jamais modifiée

(cc) BY-NC-SA

Langage - les interfaces fonctionnelles

- Interface qui possède une seule méthode abstraite
  - l'annotation `@FunctionnalInterface` permet au compilateur de vérifier que l'interface ne comporte qu'une seule méthode abstraite

```java
@FunctionalInterface
public interface Operation
{
    int compute(int x, int y);
}
```

  - peut posséder des méthodes par défaut
  - les méthodes de la classe `Object` ne sont pas prises en compte

(cc) BY-NC-SA

Les interfaces fonctionnelles

- Objectif : définir une signature de méthode qui pourra être utilisée pour passer en paramètre
  - une référence vers une méthode statique
  - une référence vers une méthode d'instance
  - une référence vers un constructeur
  - une expression lambda

(cc) BY-NC-SA

Les interfaces fonctionnelles

- Classiquement, cette interface est implémentée par une classe
  - ou par création d'une classe anonyme

```
public class Addition implements Operation
{
     @Override
     public int compute(int x, int y)
     {
          return x+y;
     }
}
```

- Une classe utilise l'interface

```
public class Calcul
{
     public static int compute(int x, int y, Operation op){
          return op.compute(x,y);
     }
}
```


(cc) BY-NC-SA

Les interfaces fonctionnelles

- Exemple

```
public class CalculLambda
{

     public static void main(String[] args)
     {
          int r = Calcul.compute(2,3, new Addition());

          r = Calcul.compute(5, 6, new Operation()
                    {
                         @Override
                         public int compute(int x, int y)
                         {
                              return x-y;
                         }
                    });

     }
}
```


(cc) BY-NC-SA

Références de méthode

- Syntaxe simplifiée pour invoquée une méthode dans une lambda
  - raccourci syntaxique
  - nouvel opérateur ::
- Quatre type de référence de méthode

Type de référence	Syntaxe	Exemple
méthode statique	clazz:method	String::valueOf
méthode d'instance	objet:method	contact::toString
méthode d'un objet d'un type compatible	clazz::method	Object::toString
constructeur	clazz:new	Contact::new

(cc) BY-NC-SA

Référence de méthode

(cc) BY-NC-SA

Référence de méthode

- Exemples

```
public class Voiture
{
     public static Voiture create(final Supplier<Voiture> supplier){
          return supplier.get();
     }

     public void rouler(){
          System.out.println("Rouler");
     }
}
```

```
public static void main(String[] args)
{
     List<Voiture> voitures = new ArrayList<>();
     voitures.add(Voiture.create(Voiture::new));
     voitures.add(Voiture.create(Voiture::new));
     voitures.add(Voiture.create(Voiture::new));

     voitures.forEach(Voiture::rouler);
}
```


(cc) BY-NC-SA