



Cours IHM-1

JavaFX

2 - Compléments de programmation



- Ce chapitre introduit de manière sommaire quelques éléments de programmation qui, faute de temps, n'ont pas été vus durant le cours de programmation du premier semestre.
- Ces éléments sont cependant utiles, voire nécessaires dans le cadre des programmes *Java/JavaFX* comportant des interfaces graphiques (GUI).
- La présentation de ces concepts se limite aux éléments essentiels qui sont nécessaires dans le cadre du cours IHM-1.
- On ne peut donc pas prétendre maîtriser ces concepts après cette brève introduction. Elle devrait cependant suffire pour en comprendre les mécanismes principaux .
- Pour plus de détail, il faut consulter l'abondante documentation disponible (livres, web) sur ces sujets.



Énumérations

Énumérations [1]



- En *Java*, le mot-clé **enum** permet de définir un *type énuméré*.
- L'utilisation de **enum** définit une classe spéciale (appelée *enum type*) qui hérite implicitement de **java.lang.Enum** et qui permet de représenter un ensemble fixe de constantes (des champs qui sont implicitement **static** et **final**) que l'on écrit donc en majuscules.
- Une variable de type énuméré ne peut prendre qu'une des valeurs constantes définies pour le type considéré (ou **null** ⚠).
- Exemple de définition du type énuméré **EColor** :

```
public enum EColor {RED, BLUE, GREEN, YELLOW}
```
- **EColor** correspond au type énuméré. Les variables de ce type ne peuvent prendre que les valeurs des constantes **RED**, **BLUE**, **GREEN**, **YELLOW** ou la référence **null**.

Énumérations [2]



- Exemple d'utilisation du type `EColor` :

```
public class TestEColor {  
    public static void main(String[] args) {  
        EColor color = EColor.BLUE;  
        if (args.length > 0) color = EColor.RED;  
        System.out.println(color);  
    }  
}
```

- Le programme affichera `BLUE` (si aucun paramètre n'est passé au lancement) ou `RED` (si au moins un paramètre est transmis au lancement).

Énumérations [3]



- On peut s'épargner la notation préfixée (*type.constante*) en ajoutant une importation statique des éléments du type énuméré.
- Exemple d'utilisation avec importation statique des constantes :

```
import static ex.chap02.EColor.*;

public class TestEColor {

    public static void main(String[] args) {
        EColor color = BLUE;
        if (args.length > 0) color = RED;
        System.out.println(color);
    }
}
```

Énumérations [4]



- Pour tous les types énumérés, une méthode statique **values()** est implicitement déclarée.
- Elle retourne un tableau des constantes d'énumération dans leur ordre de déclaration.
- Utile pour parcourir tous les éléments d'un type énuméré :

```
public class TestEColor {  
    public static void main(String[] args) {  
        for (EColor color : EColor.values()) {  
            System.out.print(color + " ");  
        }  
    }  
}
```

- Le programme affichera :

RED BLUE GREEN YELLOW

Énumérations [5]



- Pour tous les types énumérés, une méthode statique **valueOf()** est implicitement déclarée.
- Cette méthode convertit une chaîne de caractères (passée en paramètre) en un élément du type énuméré ou lève l'exception **IllegalArgumentException** si la conversion n'est pas possible (attention, la méthode est sensible à la casse [minuscules/majuscules]).
- Exemple de conversion d'une chaîne de caractères en un élément de type énuméré :

```
public class TestEColor {  
    public static void main(String[] args) {  
        EColor yColor = EColor.valueOf("YELLOW");  
        System.out.println(yColor);  
    }  
}
```

Énumérations [6]



- La méthode d'instance **name()** retourne un **String** correspondant à la représentation textuelle d'une variable de type énuméré (elle se comporte de la même manière que la méthode **toString()**).
 - Exemple : `String colName = yColor.name();`
- La méthode **ordinal()** retourne un **int** correspondant à la position de l'élément dans la déclaration d'une variable de type énuméré (la numérotation commence à zéro).
 - Exemple : `int pos = yColor.ordinal(); // 3 for YELLOW`
- Une énumération peut être définie à l'intérieur d'une classe ou dans une unité de compilation séparée (comme toute classe *Java*).
- Une énumération ne peut pas être définie dans une méthode.
- Il est possible d'ajouter des champs, des constructeurs et des méthodes dans une classe de type **enum**.

Consulter la documentation pour plus de détails.



Généricité

Types génériques

Types génériques [1]



- Un **type générique** est une classe ou une interface qui prend en paramètre un type.
- Prenons l'exemple d'une classe (non-générique) qui permet de stocker et restituer un objet de type **Integer** :

```
public class IntegerBox {  
    private Integer anInt;  
  
    public Integer get() {  
        return anInt;  
    }  
  
    public void set(Integer value) {  
        anInt = value;  
    }  
}
```

- Que faudrait-il faire pour gérer de la même manière d'autres types d'éléments, par exemple **Float**, **String**, **Point**, **Rectangle**, ... ?

Types génériques [2]



- Créer les variantes `FloatBox`, `StringBox`, ... ?
- Créer une classe `Box` avec un champ de type `Object` ?
- Une solution : créer une **classe générique** :

Inconvénients ?

```
public class Box<T> {  
    private T obj;  
  
    public T get() {  
        return obj;  
    }  
  
    public void set(T obj) {  
        this.obj = obj;  
    }  
}
```

- La syntaxe `className<type>` permet de passer un type en paramètre à la classe. Dans l'exemple précédent `T` peut être considéré comme un paramètre formel qui représente un type (non-primitif) quelconque.

Types génériques [3]



- Lorsqu'on utilise la classe (comme type ou pour l'instancier), on passe, entre < et >, le type effectif souhaité :

```
public static void main(String[] args) {  
    Box<Integer> intBox = new Box<>(); // ou new Box<Integer>();  
    Box<String> strBox = new Box<>(); // ou new Box<String>();  
    Box<Point2D> p2dBox = new Box<>(); // ou new Box<Point2D>();  
  
    intBox.set(5);  
    strBox.set("Hello");  
    p2dBox.set(new Point2D(1.2, 3.4));  
    System.out.println(p2dBox.get());  
}
```

- On peut passer plus d'un type générique à une classe ou à une interface en les séparant par une virgule :

```
public class TwoBox<T, U> { ... }
```

Types génériques [4]



- Par convention, on utilise une seule lettre majuscule pour nommer les paramètres formels de type.
- La syntaxe `<>` (nommée *diamond*) peut être utilisée lorsque le compilateur est capable de déterminer (d'inférer) le type en fonction du contexte.
- Les interfaces génériques se basent exactement sur les mêmes conventions que celles des classes génériques.
- De nombreuses interfaces et classes de la plateforme *Java* (notamment les *Collections*) utilisent la généricité.

```
List<Point2D>    polygon = new ArrayList<>();  
Pair<String, Integer> p1 = new Pair<>("Even", 8);  
Pair<String, String>  p2 = new Pair<>("Hi", "World");
```

List / ArrayList ?

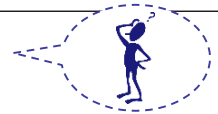


Types génériques [5]



- On peut aussi utiliser un type générique comme paramètre effectif d'un type paramétré.

```
List<Pair<String, Integer>> pList = new ArrayList<>();  
pList.add(new Pair<>("Small", 3));
```



- Pour assurer la compatibilité du code écrit avant le JDK 5.0, il est possible d'omettre le paramètre effectif et de déclarer ce que l'on nomme un *raw type*.
- Dans ce cas, le type du paramètre est implicitement **Object**, certaines limitations s'appliquent et des mises en garde peuvent apparaître à la compilation (*unchecked conversions*).

```
Box b = new Box();    // Raw type  
b.set(8);             // Autoboxing int => Integer => Object  
b.set("Hello");       // String => Object
```

Types génériques [6]



- Pour définir un paramètre générique qui inclut une classe donnée ainsi que toutes ses sous-classes, on utilise la syntaxe :

< T extends *type* >

- Par exemple :

```
public class ShapeBox<T extends Polygon> {  
    private T shape;    // Polygon or sub-class of Polygon  
    . . .  
}
```

- La formulation "**extends *type***" doit être prise ici dans un sens général (*extends or implements*). Elle doit être interprétée comme :
 - La classe *type* ainsi que toutes ses sous-classes *ou*
 - Toute classe qui implémente l'interface *type*
- Avec cette syntaxe, on indique ainsi la "*borne supérieure*" du type accepté comme paramètre (*upper bound*).

Types génériques [7]



- Le symbole '?' peut être utilisé comme caractère *joker* (*wildcard*) dans différentes situations en lien avec les classes et interfaces génériques : type d'un paramètre, d'un champ, d'une variable locale, etc.
- Il représente un type (inconnu) quelconque.
- Par exemple :

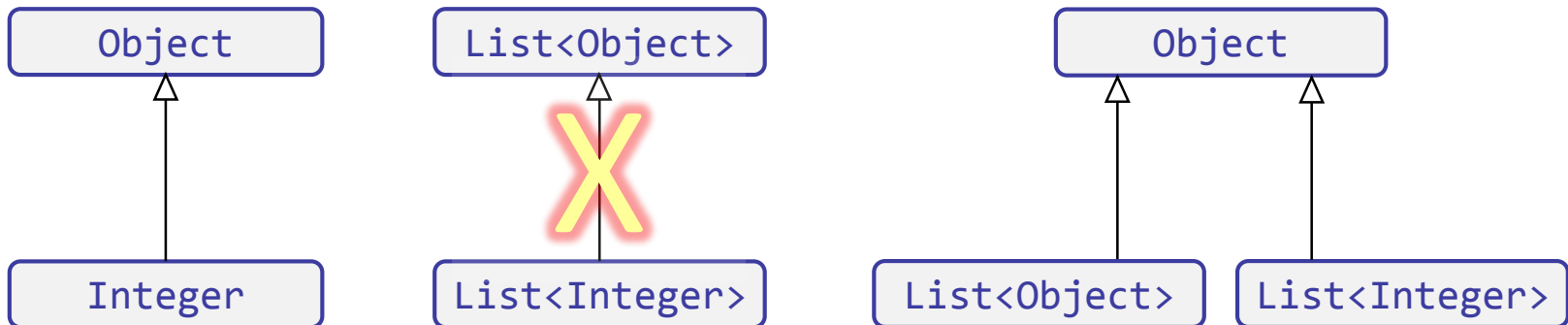
```
public static int countNonNull(List<?> list) {  
    int count = 0;  
    for (Object e : list) {  
        if (e != null) count++;  
    }  
    return count;  
}
```

- Dans cet exemple, seules les méthodes de l'interface **List** ou de **Object** (sur les éléments de la liste) peuvent être utilisées.

Types génériques [8]



- Le caractère joker **?** n'est pas équivalent à **Object**.
 - **List<?>** : une liste d'éléments de n'importe quel type
 - **List<Object>** : une liste d'éléments de type **Object**
- Attention : Même si (par ex.) **Integer** est une sous-classe de **Object**, **List<Integer>** n'est pas une sous-classe de **List<Object>**



Types génériques [9]



- Lors de l'introduction de la généricité dans le langage, la machine virtuelle n'a pas été modifiée. Une classe paramétrée est compilée en une classe non paramétrée.
- Les variables de types sont donc perdues (oubliées) après la compilation (*type erasure*). Certaines opérations qui utilisent les types génériques à l'exécution (par exemple certains transtypages, création d'objets, utilisation de `instanceof`, ...) ne sont donc pas autorisées et provoquent donc des erreurs (qui peuvent paraître surprenantes) lors de la compilation.

```
public class Box<T> {  
    . . .  
    T temp = new T(); // Erreur à la compilation  
    . . .  
}
```

Types génériques [10]



- Le caractère *joker* '?' (*unbounded wildcard*) peut également être combiné avec les mots-clés **extends** ou **super** pour définir des bornes supérieures (*upper bound*) respectivement des bornes inférieures (*lower bound*) pour les types acceptés.
 - `<? extends type>` acceptera la classe *type* ainsi que toutes ses sous-classes. Cette notation est également valable pour toutes les classes qui implémentent l'interface *type*. *[upper bounded]*
 - `<? super type>` acceptera la classe *type* ainsi que toutes les classes parentes de *type*. *[lower bounded]*
- Par exemple, pour ajouter des nombres à une liste d'**Integer**, de **Number** ou d'**Object** :

```
public static void addOneToTen(List<? super Integer> list) {  
    for (int i = 1; i <= 10; i++) {  
        list.add(i);  
    }  
}
```



Quelques règles et recommandations

- L'utilisation du caractère *joker* `<? ...>` dépendra du rôle que joue le paramètre générique dans la méthode/fonction :
 - Paramètre *'in'* (valeur utilisée en entrée de la méthode)
[Typiquement la méthode *consomme* les éléments d'une collection]
 - Paramètre *'out'* (valeur utilisée en sortie, fournie par la méthode)
[Typiquement la méthode *produit* les éléments d'une collection]
 - Paramètre *'in/out'* (valeur utilisée en entrée et modifié par la méthode)
- Il est recommandé d'appliquer les **règles** suivantes :
 - Tous les paramètres *'in'* seront définis avec **extends** (*upper-bound*)
 - Tous les paramètres *'out'* seront définis avec **super** (*lower-bound*)
 - Pour les paramètres *'in'* accédés exclusivement avec des méthodes de **Object**, on utilisera un *joker* non borné (*unbounded-wildcard* `<?>`)
 - Pour tous les paramètres *'in/out'* on n'utilisera pas de *joker*
 - On n'utilisera pas de *joker* comme type de retour

Types génériques [12]



- Dans une classe (générique ou non) il est possible d'écrire une **méthode générique** (statique ou non-statique).
- Syntaxiquement, le ou les types génériques `<...>` se placent, dans l'en-tête, après les modificateurs et avant le type de retour.
- Exemple :

```
public static <T> boolean contains(Box<T> box, T value) {  
    return box.get().equals(value);  
}
```

- En général, le paramètre générique n'est pas nécessaire lors de l'invocation de la méthode, il est inféré par le contexte.

```
Box.contains(fBox, 1.2f);
```

```
Box.<Float>contains(fBox, 1.2f); // Syntaxe rarement nécessaire
```



Propriétés

Notion de propriété [1]



- La notion de "propriété" (*property*) est très présente dans *JavaFX*.
- Une **propriété** est un élément d'une classe que l'on peut manipuler à l'aide de *getters* (lecture) et de *setters* (écriture).
- Les propriétés sont généralement représentées par des attributs de la classe mais elles pourraient aussi être stockées dans une base de données ou autre système d'information.
- En plus des méthodes `get...()` et `set...()`, les propriétés *JavaFX* possèdent une troisième méthode `...Property()` qui retourne un objet qui implémente l'interface `Property` [`...` : nom de la propriété].
- Intérêt des propriétés :
 - Elles peuvent être liées entre-elles (*Binding*), c-à-d que le changement d'une propriété entraîne automatiquement la mise à jour d'une autre.
 - Elles peuvent déclencher un événement lorsque leur valeur change et un gestionnaire d'événement (*Listener*) peut réagir en conséquence.

Notion de propriété [2]



- Exemple d'une classe définissant la propriété *balance* (solde) pour un compte bancaire.

```
public class BankAccount {  
    private DoubleProperty balance = new SimpleDoubleProperty();  
    public final double getBalance() {  
        return balance.get();  
    }  
    public final void setBalance(double amount) {  
        balance.set(amount);  
    }  
    public final DoubleProperty balanceProperty() {  
        return balance;  
    }  
    . . .  
}
```



Notion de propriété [3]



- La classe abstraite `DoubleProperty` permet d'emballer une valeur de type `double` et d'offrir des méthodes pour consulter et modifier la valeur mais également pour "observer" et "lier" les changements.
- `SimpleDoubleProperty` est une classe concrète prédéfinie.
- La plateforme *Java* offre des classes similaires pour la plupart des types primitifs, les chaînes de caractères, certaines collections ainsi que le type `Object` qui peut couvrir tous les autres types.
 - `IntegerProperty` / `SimpleIntegerProperty`
 - `StringProperty` / `SimpleStringProperty`
 - `ListProperty<E>` / `SimpleListProperty<E>`
 - `ObjectProperty<T>` / `SimpleObjectProperty<T>`
 - ...
- Des classes existent également pour définir des propriétés *read-only*
 - `ReadOnlyIntegerWrapper`, `ReadOnlyIntegerProperty`
 - ...

Observation des propriétés [1]



- Toutes les classes de propriétés implémentent l'interface `Observable` et offrent de ce fait, la possibilité d'enregistrer des *observateurs* (**Listener**) qui seront avertis lorsque la valeur de la propriété change.
- Une instance de l'interface fonctionnelle `ChangeListener<T>` pourra ainsi être créée pour réagir à un tel changement. La méthode `changed()` sera alors invoquée et recevra en paramètre la valeur observée ainsi que l'ancienne et la nouvelle valeur de la propriété.

```
DoubleProperty sum = account.balanceProperty();  
sum.addListener( (ObservableValue<? extends Number> obsVal,  
                 Number oldVal,  
                 Number newVal ) ->  
    {  
        //--- ChangeListener code  
        System.out.println(oldVal+" becomes "+ newVal);  
        . . .  
    } );
```

Observation des propriétés [2]



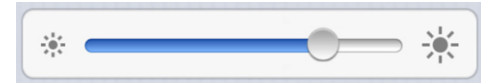
Remarque: Dans l'exemple précédent, une expression lambda est utilisée pour implémenter la méthode `changed()`. On pourrait également utiliser une classe anonyme ou créer l'instance d'une classe "ordinaire" qui implémente l'interface `ChangeListener<T>`.

- L'interface fonctionnelle `InvalidationListener<T>` permet également de réagir aux changements des valeurs de propriétés dans les situations où les propriétés sont calculées à partir d'autres et que l'on veut éviter d'effectuer les calculs à chaque changement.
- Avec cette interface, c'est la méthode `invalidated(Observable o)` qui est invoquée lorsqu'un changement potentiel de la valeur de la propriété est intervenu.
- Cette méthode peut cependant être invoquée alors que le résultat observé n'a finalement pas changé (cela dépend des opérations effectuées).

Propriétés des composants



- Les composants utilisés dans les interfaces graphiques (boutons, champs texte, cases à cocher, *sliders*, etc.) possèdent tous de nombreuses propriétés.



- Pour chacun des composants, la documentation (*Javadoc*) décrit dans une des rubriques (*Property Summary*) la liste des propriétés de la classe concernée ainsi que celles qui sont héritées.

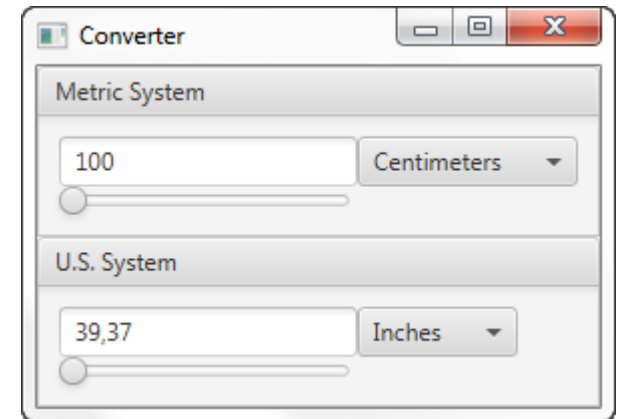
Property Summary

All Methods	Instance Methods	Concrete Methods
Type	Property and Description	
DoubleProperty	blockIncrement The amount by which to adjust the slider if the track of the slider is clicked.	
ObjectProperty<StringConverter<Double>>	labelFormatter A function for formatting the label for a major tick.	
DoubleProperty	majorTickUnit The unit distance between major tick marks.	
DoubleProperty	max The maximum value represented by this Slider.	
IntegerProperty	minorTickCount The number of minor ticks to place between any two major ticks.	

Lier des propriétés [1]



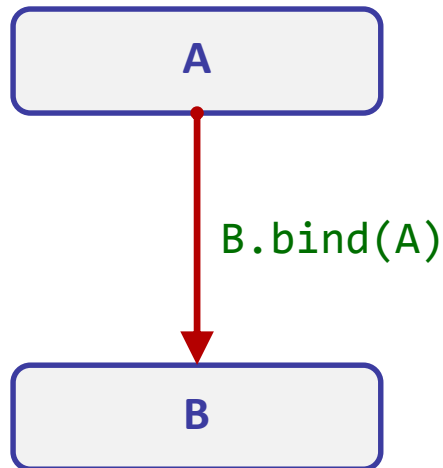
- Un des avantages des propriétés *JavaFX* est la possibilité de pouvoir les lier entre-elles. Ce mécanisme, appelé **binding**, permet de mettre à jour automatiquement une propriété en fonction d'une autre.
- Dans les interfaces utilisateurs, on a fréquemment ce type de liens.
 - Par exemple, lorsqu'on déplace le curseur d'un *slider*, la valeur d'un champ texte changera (ou la luminosité d'une image, la taille d'un graphique, le niveau sonore, etc.).
- Il est possible de lier deux propriétés *A* et *B* de manière
 - **Unidirectionnelle** : un changement de *A* entraînera un changement de *B* mais pas l'inverse (*B* non modifiable autrement)
 - **Bidirectionnelle** : un changement de *A* entraînera un changement de *B* et réciproquement (les deux sont modifiables)



Lier des propriétés [2]

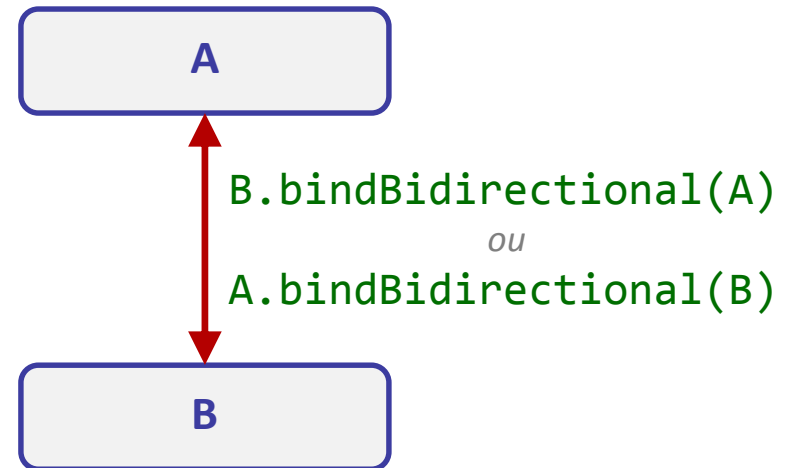


- Illustration des liens possibles entre deux propriétés **A** et **B** :



Liaison unidirectionnelle

(la valeur de la propriété B dépend de la valeur de la propriété A)



Liaison bidirectionnelle

(la valeur de la propriété B dépend de la valeur de la propriété A et inversement)

Lier des propriétés [3]



- La méthode **bind()** permet de créer un lien unidirectionnel.
- La méthode doit être appelée sur la propriété qui sera "soumise" à l'autre (celle qui est passée en paramètre).

```
BankAccount account = new BankAccount();  
Slider slider = new Slider();  
account.balanceProperty().bind(slider.valueProperty());
```

- Dans cet exemple la valeur du solde du compte bancaire est liée (asservie) à la position du curseur du *slider*. Si l'on tente de modifier le solde d'une autre manière (avec **setBalance()** par exemple) une exception sera générée.
- Une liaison bidirectionnelle s'effectue de manière similaire, mais en utilisant la méthode **bindBidirectional()**.
- Pour supprimer le lien : **unbind()** et **unbindBidirectional()**

Lier des propriétés [4]



- Une propriété ne peut être liée (asservie) qu'à une seule autre si le lien est unidirectionnel (`bind()`). Par contre, les liens bidirectionnels (`bindBidirectional()`) peuvent être multiples.
- Parfois, une propriété dépend d'une autre mais avec une relation plus complexe. Il est ainsi possible de créer des **propriétés calculées**.
- Deux techniques sont à disposition (elles peuvent être combinées) :
 - Utiliser la classe utilitaire `Bindings` qui possède de nombreuses méthodes statiques permettant d'effectuer des opérations.
 - Utiliser les méthodes disponibles dans les classes qui représentent les propriétés; ces méthodes peuvent être chaînées (*Fluent API*).
- Des opérations de conversions sont parfois nécessaires si le type des propriétés à lier n'est pas le même. Par exemple pour lier un champ texte (`StringProperty`) à un *slider* dont la valeur est numérique (`DoubleProperty`).

Lier des propriétés [5]

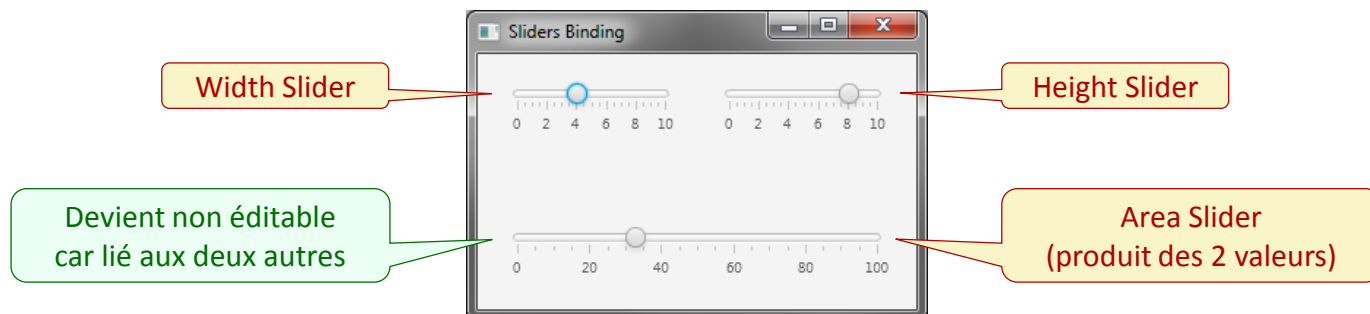


- Exemple de *binding* permettant de coupler la valeur d'un *slider* (*Area*) aux valeurs d'entrée de deux autres *sliders* (*Width* et *Height*)
- En utilisant les méthodes statiques de la classe Bindings :

```
Slider    widthSlider  = new Slider();    // Input
Slider    heightSlider = new Slider();    // Input
Slider    areaSlider   = new Slider();    // Output

DoubleProperty widthPty = widthSlider.valueProperty();
DoubleProperty heightPty = heightSlider.valueProperty();
DoubleProperty areaPty   = areaSlider.valueProperty();

areaPty.bind(Bindings.multiply(widthPty, heightPty));
```



Lier des propriétés [6]



- Même exemple, en utilisant des méthodes chaînées (*Fluent API*) :

```
Slider    widthSlider  = new Slider(); // Input
Slider    heightSlider = new Slider(); // Input
Slider    areaSlider   = new Slider(); // Output

DoubleProperty widthPty = widthSlider.valueProperty();
DoubleProperty heightPty = heightSlider.valueProperty();
DoubleProperty areaPty   = areaSlider.valueProperty();

areaPty.bind(widthPty.multiply(heightPty));
```

- Un jeu d'opérations est disponible aussi bien avec la classe `Bindings` qu'avec les méthodes chaînables.
 - `min()`, `max()`
 - `equal()`, `notEqual()`, `lessThan()`, `lessThanOrEqualTo()`, ...
 - `isNull()`, `isNotNull()`, `isEmpty()`, `isNotEmpty()`
 - `convert()`, `concat()`, `format()`, ...
 - *et beaucoup d'autres ...*

Lier des propriétés [7]



- Si les opérations disponibles dans les API, dites de haut-niveau, ne permettent pas d'exprimer la relation entre les propriétés, il est possible de définir une liaison de plus bas niveau (*low-level binding*) en redéfinissant la méthode abstraite `computeValue()` d'une des classes de *binding* (`DoubleBinding`, `BooleanBinding`, `StringBinding`, ...).

```
DoubleBinding complexBinding = new DoubleBinding() {
    {
        //--- Set listeners to 'in'-properties
        super.bind(aSlider.valueProperty(), bSlider.valueProperty());
    }

    @Override //--- Compute 'out' value
    protected double computeValue() {
        double w = aSlider.valueProperty().get();
        double h = bSlider.valueProperty().get();
        return Math.sqrt(h) * Math.pow(w, 3);
    }
};

outSlider.valueProperty().bind(complexBinding); // Do the binding
```



Annotations

Annotations [1]



- En Java, une annotation constitue une métadonnée que l'on peut appliquer à différents éléments du code source.
- Les annotations n'ont pas d'effet direct sur le fonctionnement des instructions annotées.
- Le caractère '@' indique au compilateur que l'identificateur qui suit est une annotation.
- Un exemple simple :

```
@Override
public String toString() {
    //--- A String representation of current object
    return "Circle: Radius=" + radius;
}
```

- L'annotation `@Override` indique au compilateur que la méthode redéfinit une méthode héritée. Il pourra ainsi générer une erreur si l'en-tête n'est pas correcte

Annotations [2]



- Sans l'annotation `@Override` le code fonctionne parfaitement (et de manière identique) mais le programmeur ne serait pas averti qu'il crée une nouvelle méthode d'instance (faute de frappe par ex.) alors qu'il voulait redéfinir une méthode d'une des classes parentes.
- Une annotation peut inclure un ou plusieurs paramètres :

```
@Author (name = "Tryphon Tournesol",  
         date = "06.06.2016"           ) // Metadata about the class  
public class MyClass {  
    . . .  
}
```

- Si l'annotation ne comporte qu'un seul paramètre dont le nom est `value`, le nom peut être omis :

```
@SuppressWarnings("unchecked")  
public class MyClass {  
    . . . // Warnings about unchecked generic operations are suppressed  
}
```

Annotations [3]



- Des annotations multiples peuvent être attribuées à une même déclaration (une même annotation peut également être répétée).

```
@Author (name = "Tryphon Tournesol") // Repeating annotations
@Author (name = "Archibald Haddock") // Repeating annotations
@EBook
public class MyClass {
    . . .
}
```

- Des annotations peuvent être appliquées à des déclarations (classe, champs, méthodes, etc.) mais également lors de l'utilisation des types (ces *Type Annotations* requièrent l'installation d'un *Checker Framework*, voir par exemple types.cs.washington.edu/checker-framework).

```
public void check(String name) {
    @NonNull String loginName = name; // Exception thrown if null
    . . .
}
```

Annotations [4]



- Il existe un certain nombre d'annotations prédéfinies, traitées par le compilateur ou par certains outils standard de la plateforme *Java*.
 - Exemples : `@Deprecated`, `@Override`, `@SuppressWarnings`, `@SafeVarargs`, `@FunctionalInterface`, ...
- Il est également possible de définir ses propres annotations en écrivant une interface particulière (`@interface ...`)

```
import java.lang.annotation.Documented;

@Documented           // Will appear in Javadoc-generated documentation
public @interface ClassPreamble {
    String    author();           // Parameter
    String    date();
    int       currentRevision() default 1; // Param. with default value
    String    lastModifiedBy() default "N/A";
    String[]  reviewers();       // Array parameter
}
```



Expressions Lambda

Références de méthodes

Expressions lambda [1]



- Une **expression lambda** peut être assimilée à une fonction anonyme qui a potentiellement accès au contexte du code englobant.
- Il s'agit essentiellement d'un bloc de code avec des paramètres et qui est destiné à être exécuté ultérieurement (une ou plusieurs fois).
- Une expression lambda peut être utilisée pour implémenter une *interface fonctionnelle* (interface ne comportant qu'une seule méthode d'instance abstraite).

- La syntaxe de base est :

ou

```
(parameters) -> expression  
(parameters) -> { statements; }
```

- Remarque : Les parenthèses autour des paramètres sont obligatoires s'il y a zéro (*burger arrow* : `() ->`) ou plusieurs paramètres, mais optionnelles s'il n'y en a qu'un seul et que le type est inféré (voir page suivante).

Expressions lambda [2]



- S'il y a plusieurs paramètres, il seront séparés par des virgules.
- Le type des paramètres peut être indiqué explicitement ou être automatiquement déterminé par le compilateur (inféré) en fonction du contexte.
- Le type de retour est toujours inféré du contexte.
- Exemples d'expressions lambda :

```
(int x, int y) -> { return x + y; }
```

```
// Types explicites, valeur de retour
```

```
(x, y) -> x + y
```

```
// Types inférés, valeur de retour
```

```
x -> x * x
```

```
// Type inféré, valeur de retour
```

```
() -> 123
```

```
// Pas de paramètre, valeur de retour
```

```
s -> System.out.println(s)
```

```
// Type inféré, pas de valeur de retour
```

```
() -> { for (int i=0; i<10; i++) doIt(); }
```

```
// Pas de valeur de retour
```

Expressions lambda [3]



- Par défaut le code *Java* s'exécute dans un *thread* appelé *main-thread* (créé automatiquement). Il est possible de créer d'autres *threads* qui sont des fils d'exécution (des processus légers) permettant l'exécution parallèle (ou pseudo-parallèle) du code.
- Une expression lambda peut être utilisée pour instancier l'interface fonctionnelle **Runnable** (définie dans le package `java.lang`) qui comporte la méthode abstraite `public void run();` permettant notamment d'exécuter son code dans un *thread* séparé.
- Pour lancer du code dans un *thread* séparé, il faut
 - Créer une instance de la classe **Thread** en passant l'objet **Runnable** à son constructeur : `Thread t = new Thread(myRunnable);`
 - Appeler la méthode `start()` de l'instance de **Thread** : `t.start();`
- *Une autre manière de faire* : créer une sous-classe de **Thread** et redéfinir la méthode `run()`. Créer une instance et invoquer `start()`.

Expressions lambda [4]



- Exemple : création de deux *threads* à l'aide d'expressions lambda.

```
public static void main(String[] args) {  
    Runnable rA = () -> { for (int i=0; i<100; i++)  
                           System.out.println("A"+i); };  
  
    Runnable rB = () -> { for (int i=0; i<100; i++)  
                           System.out.println("B"+i); };  
  
    Thread tA = new Thread(rA); // Création du thread  
    Thread tB = new Thread(rB);  
  
    tA.start(); // Lancement du thread : méthode run()  
    tB.start();  
}
```

A0
B0
B1
B2
A1
A2
B3
B4
B5
A3
B6
B7
B8
A4
B9
A5
A6
A7
A8
A9
A10
A11
B10
...

*Un des résultats
possibles à l'exécution*

Interfaces fonctionnelles [1]



- Les interfaces fonctionnelles jouent un rôle important en lien avec les expressions lambda. Plusieurs de ces interfaces sont prédéfinies (notamment dans le package `java.util.function`).
- Deux exemples représentatifs (utilisés par la suite) :

```
// Supplier : Represents a supplier of results
//-----
@FunctionalInterface
public interface Supplier<T> {
    T get(); // Gets a T type result
}
```

```
// Function : Represents a function that accepts one argument
//              and produces a result
//-----
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t); // Applies this function to the given argument
}
```

Interfaces fonctionnelles [2]



Quelques interfaces fonctionnelles prédéfinies

Interface fonctionnelle	Type Param	Type Retour	Méthode abstraite	Description	Autres Méthodes
Runnable	-	void	run	Exécute une action	
Supplier<T>	-	T	get	Fournit une valeur de type T	
Consumer<T>	T	void	accept	Consomme une valeur de type T	chain
BiConsumer<T, U>	T, U	void	accept	Consomme des valeurs de type T et U	chain
Function<T, R>	T	R	apply	Fonction avec un seul paramètre	compose, andThen, identity
BiFunction<T, U, R>	T, U	R	apply	Fonction avec deux paramètres	andThen
UnaryOperator<T>	T	T	apply	Opérateur unaire sur le type T	compose, andThen, identity
BinaryOperator<T>	T, T	T	apply	Opérateur binaire sur le type T	andThen

Interfaces fonctionnelles [3]



Quelques interfaces fonctionnelles prédéfinies

Interface fonctionnelle	Type Param	Type Retour	Méthode abstraite	Description	Autres Méthodes
Predicate<T>	T	boolean	test	Fonction booléenne (prédicat)	and, or, negate, isEqual
BiPredicate<T, U>	T, U	boolean	test	Fonction booléenne avec 2 paramètres	and, or, negate

- Il existe également un certain nombre d'interfaces fonctionnelles prédéfinies pour les types primitifs **int**, **long** et **double**. Exemples :

Interface fonctionnelle	Type Param	Type Retour	Méthode abstraite	Description	Autres Méthodes
IntFunction<T>	int	T	apply	Fonction avec paramètre int	
LongPredicate	long	boolean	test	Fonction booléenne avec paramètre long	and, or, negate
...	...	...	...	... et beaucoup d'autres ...	...

Références de méthodes [1]



- Une méthode existante peut être utilisée en tant qu'implémentation de la méthode abstraite d'une interface fonctionnelle.
- On définit pour cela une **référence de méthode** avec l'opérateur **::**
- Trois variantes existent :

Class :: staticMethodName
instance :: instanceMethodName
Class :: instanceMethodName

- Exemples avec équivalences en expressions lambda :

Variante	Exemple	Lambda équivalent
<i>Class :: staticMethodName</i>	<code>String::valueOf</code>	<code>(s) -> String.valueOf(s)</code>
<i>instance :: instanceMethodName</i>	<code>obj::toString</code>	<code>() -> obj.toString()</code>
<i>Class :: instanceMethodName</i>	<code>String::toString</code>	<code>(s) -> s.toString()</code>
<i>Class :: instanceMethodName</i>	<code>String::charAt</code>	<code>(s, i) -> s.charAt(i)</code>

Références de méthodes [2]



- Dans les deux premières variantes de syntaxe (*Class::staticMethodName* et *instance::instanceMethodName*), la référence de méthode est équivalente à une expression lambda qui fournit les paramètres de la méthode.
- Exemple variante 1 :

```
Supplier<Double> random = Math::random;  
double result = random.get(); // Math.random();
```

- Exemple variante 2 :

```
Random r = new Random();  
Supplier<Double> random = r::nextDouble;  
double result2 = random.get(); // r.nextDouble();
```

Ces deux exemples utilisent l'interface fonctionnelle prédéfinie Supplier<T> décrite dans les pages précédente.

Références de méthodes [3]



- Dans la troisième variante (`Class::instanceMethodName`), même si on mentionne le nom de la classe en préfixe, c'est bien une méthode d'instance qui sera invoquée. Le premier paramètre représente l'objet sur lequel la méthode sera invoquée.
- Exemple variante 3 :

```
Function<String, Integer> fLen = String::length;  
  
String s1 = "Hello";  
String s2 = "World !";  
  
int s1Len = fLen.apply(s1); // s1.length()  
int s2Len = fLen.apply(s2); // s2.length()
```

- Les références de méthodes représentent donc une alternative aux expressions lambda dans le cas où il n'y a qu'une seule et unique méthode à exécuter.
- Cela améliore généralement la lisibilité et la clarté du code.

Références de constructeurs [1]



- Les **références de constructeurs** sont semblables aux références de méthodes à la différence que le nom de la "méthode" est **new**.
- Les références de constructeurs servent à instancier une interface fonctionnelle avec un constructeur compatible avec le contexte d'utilisation.
- Syntaxe :

Class :: new

- Exemples avec équivalences en expressions lambda :

Syntaxe	Exemple	Lambda équivalent
<i>Class</i> ::new	String::new	() -> new String()
<i>Class</i> ::new	PointHolder::new	(p) -> new PointHolder(p)

Références de constructeurs [2]



- Exemple d'utilisation de références de constructeurs pour créer des objets de type `Button` (un des constructeurs de la classe `Button` permet de passer le libellé du bouton en paramètre) :

```
Function<String, Button> btFactory = Button::new;  
Button bOk    = btFactory.apply("Ok");    // new Button("Ok");  
Button bQuit  = btFactory.apply("Quit");  // new Button("Quit");
```

Accès aux variables locales [1]



- En plus des éventuels paramètres définis, des variables déclarées dans son corps et des champs statiques, une expression lambda à accès à toutes les variables locales qui sont *effectivement finales*.
- Une ***variable effectivement finale*** est une variable locale :
 - déclarée avec le modificateur **final**
 - ou*
 - non modifiée après sa première affectation
- Une expression lambda qui utilise (capture) des variables locales externes est parfois désignée par le terme de *capturing lambda*.
- On utilise également le terme *closure* pour désigner le bloc de code de l'expression lambda accompagné de l'ensemble des variables locales capturées.

Accès aux variables locales [2]



- Dans l'exemple qui suit, les variables `text` et `count` ne sont pas définies dans l'expression lambda. Ce sont des paramètres de la méthode englobante capturés par l'expression lambda (variables locales effectivement finales).

```
public static void repeatMessage(String text, int count) {  
    Runnable r = () -> { for (int i=0; i<count; i++) {  
        System.out.println(text);  
        Thread.yield(); // Gives a chance other threads to run  
    }  
};  
new Thread(r).start();  
}
```

- Ces variables restent accessibles malgré l'exécution différée du code.

```
public static void main(String[] args) {  
    repeatMessage("Hello", 100);  
    repeatMessage("Jane", 100);  
}
```



Stream

Stream API [1]



- Un **stream** représente une séquence d'éléments. Cette structure supporte différents types d'opérations sur ces éléments.
- Les *streams* constituent un concept important en lien avec les expressions lambda qui sont souvent utilisées pour les manipuler.
- L'essentiel des librairies (API) liées aux *streams* se trouve dans le package `java.util.stream` avec notamment l'interface `Stream<T>`
- Il est possible de créer un *stream* à partir d'un tableau, d'une collection, d'un générateur (`Stream.generate()`), d'un itérateur (`Stream.iterate()`), d'un canal d'entrée/sortie, etc.
- Des manières classiques de créer un *stream* à partir :
 - D'un tableau : `Arrays.stream(array)`
 - D'une collection : `collection.stream()`
 - D'une séquence de valeurs : `Stream.of(v1, v2, ...)`

Stream API [2]



- Un exemple d'utilisation à partir d'une liste de **String** sur laquelle on applique une série d'opérations :

```
public static void main(String[] args) {  
    List<String> myList =  
        Arrays.asList("a1", "a2", "b1", "c2", "d1", "c1");  
  
    myList.stream()  
        .filter(s -> s.startsWith("c"))  
        .map(String::toUpperCase)  
        .sorted()  
        .forEach(System.out::println);  
}
```

*Operation
pipeline*

- A l'exécution, ce programme affichera :

C1
C2



- Une expression lambda et deux références de méthodes sont utilisées dans cet exemple pour définir certaines des opérations effectuées sur le flux.



- Les opérations appliquées à un *stream* peuvent être :
 - *Intermédiaires* : elles retournent un *stream* ce qui permet un enchaînement d'opérations (*pipeline, fluent API*)
 - *Terminales* : elles ne retournent rien (effectuent une opération) ou retournent autre chose qu'un *stream*
- Les opérations intermédiaires ne sont exécutées que s'il y a une opération terminale (*laziness*).
- L'ordre dans lequel les opérations sont enchaînées peut jouer un rôle important sur le résultat et sur le nombre d'opérations à effectuer par le système (typiquement, un filtrage est à effectuer le plus tôt possible dans la chaîne des opérations).
- Après avoir exécuté une opération terminale, un *stream* est fermé et ne peut plus être réutilisé (si on veut le faire, il faut en créer un à nouveau).