Les interfaces

NFA032

Présentation n°5

Présentation générale

- Motivations
  - Comment garantir qu'un objet saura répondre à une liste de messages
    - Inspiré de la *liste des opérations* des *types abstraits*
      - Définition implicite d'une notion de **contrat** : tout objet s'engage à disposer d'un comportement adapté pour ces messages
    - Langages à objets : il suffit a priori qu'il soit instance d'une classe $\Rightarrow$ catalogue des méthodes
  - Comment éviter les problèmes liés à l'héritage multiple
- Réponse en Java : les interfaces
  - Interface = catalogue de signatures de méthodes
  - Remarque : *interface* est féminin

Les interfaces – définition (1/3)

- Définition
  - Une interface est comme une classe, excepté que :
    - Elle ne peut pas être *instanciée* (pas de `new` ni de constructeurs)
    - Elle ne peut pas définir de *membres* (*variables* ou *méthodes*) *statiques*
    - Elle ne peut pas définir d'*attributs* (variables non statiques)
    - Elle ne peut pas spécifier le *corps* des méthodes : seules leurs *signatures* sont spécifiées
    - Toutes les méthodes spécifiées sont *publiques* (rappels juste après)
  - Relations entre classes et interfaces :
    - Une interface peut *hériter* d'autres interfaces
      - Étudié dans NFA032 mais plus tard
    - Une classe peut *hériter* d'interfaces : on dit que la classe *implémente* ces interfaces

Les interfaces – définition (2/3)

- Syntaxe d'une définition d'interface :

```
interface nom {
    signature1 ;
    …
    signaturen ;
}
```

  - Remarques :
    - Le qualificatif de protection **`public`** est optionnel pour chaque signature ; il est donc implicite si non précisé
    - Gestion des fichiers similaire à celle des classes :
      - Soit l'interface `I` est définie dans un fichier source `I.java`
      - Soit l'interface `I` est définie dans le fichier source d'une autre entité `E` (donc fichier `E.java`), définition placée après `E`

Les interfaces – définition (3/3)

- Exemple : tout animal porte un nom et pousse un cri ; il est possible d'en connaître le genre

```
interface unAnimal {
    String sonNom();
    String sonCri();
    String sonGenre();
}
```

  - Tout objet `a` qui répond à cette interface est donc susceptible de savoir répondre à :
    - `a.sonNom()`, qui est un texte indiquant le nom de l'animal
    - `a.sonCri()`, qui est un texte indiquant son cri
    - `a.sonGenre()`, qui est un texte indiquant son genre : chat, chien, …
- Exemple : toute fonction réelle *f* se caractérise par le fait qu'elle s'applique sur un *x* réel :

```
interface uneFonction {
    double de(double x);
}
```

  - Tout objet `f` qui répond à cette interface est donc susceptible de savoir répondre à :
    - `f.de(x)`, qui est un réel valant *f*(*x*) (notation mathématique), qui se lit « *f* de *x* »

A propos des protections d'accès

- En Java, tout membre `m` d'une classe `C` placée dans un paquetage `P` dispose d'une protection d'accès, spécifiée au début de la définition du membre.
- Par ordre décroissant de restriction :
  - Protection **`public`**
    - `m` exploitable par toute méthode de toute classe de tout paquetage
  - Protection non spécifiée = protection **`package`**
    - `m` exploitable par toute méthode de toute classe du paquetage `P`
  - Protection **`protected`**
    - `m` exploitable par toute méthode de toute classe héritant de `C` au sens large
  - Protection **`private`**
    - `m` exploitable par toute méthode de la classe `C`

Relation d'implémentation

- Une classe `C` peut *hériter* (on dit *implémenter*) d'autant d'interfaces que souhaité

```
class C implements I1, I2, …, In {
    …
}
```

- Le principe :
  - Toute méthode `m` dont la signature est précisée dans une interface I_k a son corps défini dans `C`
    - Protection **`public`** impérative
    - Annotation `@Override` facultative, mais conseillée
      - Permet au compilateur de s'assurer qu'il s'agit bien d'une méthode issue d'une interface
      - Permet au développeur de s'assurer qu'il ne s'est pas trompé dans la signature

Polymorphisme des variables et liaison dynamique de code

- Soient :
  - `I` une interface, spécifiant une méthode `m(…)`
  - `v` une variable de type `I`
  - `C` une classe qui implémente `I`
- Polymorphisme des variables
  - `v` peut être initialisée ou affectée avec tout objet de type `C`
- Liaison dynamique de code
  - Un appel `v.m(…)` appelle la méthode `m(…)` définie dans `C` si `v` référence un objet `o` de classe `C`
    - Le code appelé à l'exécution ne peut être prédit par le compilateur, en général ; il est déterminé juste avant l'appel, d'où la notion de *liaison dynamique de code*.

Polymorphisme des variables et liaison dynamique de code

- Quand une variable est définie :

  `I v`

  Il faut distinguer :
  - Le type déclaré de la variable `v` (ici `I`)
    - Aussi appelé son *type statique*
    - Utile à la compilation (vérification des accès ou appels)
      - Polymorphisme des variables
  - Le type réel de l'objet référencé par `v`, qui est nécessairement une classe `C` qui implémente `I`
    - Aussi appelé son *type dynamique*
    - Utile à l'exécution (détermination du bon corps de la méthode à appeler)
      - Liaison dynamique de code

Exemple – les animaux

- Exemple des animaux

```
interface unAnimal {
   String sonNom();
   String sonCri();
   String sonGenre();
}
```

- Quelques cris :
  - Pour un chien : *oua oua*
  - Pour un chat : *miaou*
  - Pour une souris : *couic*

Exemple – les animaux

- La classe des chiens (idem pour les autres) :

Légende :

Éléments à adapter pour chaque genre d'animal

```
class unChien implements unAnimal {
  String _nom;
  static int dernier = 0;
  static String genre = "un chien";

  unChien() { this(genre+" #"+ ++dernier); }

  unChien(String nom) { this._nom = nom; }

  @Override public String sonNom() {
    return this._nom;
  }

  @Override public String sonCri() {
    return "oua oua";
  }

  @Override public String sonGenre() {
    return genre;
  }
}
```

Exemple – les animaux

- Le programme principal crée un tableau de différents animaux, et demande à chacun qu'il *s'exprime* :

```
un chien #1 : oua oua
'Ros Minet (un chat) : miaou
Rintintin (un chien) : oua oua
Jerry (une souris) : couic
un chien #2 : oua oua
Sylvestre (un chat) : miaou
Speedy Gonzales (une souris) : couic
une souris #1 : couic
```

Exemple – les animaux

- Polymorphisme
- Liaison dynamique

```
class testAnimaux {
  public static void main(String[] args) {
    unAnimal[] mesAnimaux = new unAnimal[] {
      new unChien(),
      new unChat("'Ros Minet"),
      new unChien("Rintintin"),
      new uneSouris("Jerry"),
      new unChien(),
      new unChat("Sylvestre"),
      new uneSouris("Speedy Gonzales"),
      new uneSouris()
    };

    for (unAnimal animal : mesAnimaux)
      exprimeToi(animal);
  }

  static void exprimeToi(unAnimal animal) {
    Terminal.ecrireString(animal.sonNom()+" ");
    if (! animal.sonNom().startsWith(animal.sonGenre()))
      Terminal.ecrireString("("+animal.sonGenre()+") ");
    Terminal.ecrireStringln(": "+animal.sonCri());
  }
}
```

A propos du polymorphisme des variables

- Soient :
  - `I` une interface
  - `v` une variable de type `I`
  - `C` et `D` deux classes indépendantes qui implémentent `I`
  - Deux objets définis par `C oC` et `D oD`
- Le polymorphisme des variables :
  - Fonctionne des classes vers les interfaces

    ```
    I v = oC ; //
    ```
    est licite, par application du polymorphisme
  - Ne fonctionne pas dans l'autre sens :

    ```
    I v = oC ;
    oC = v ; // est incorrect
    oC = (C)v ; // est correct cette fois
    ```
    - Il faut écrire explicitement la conversion (opérateur (…) dit de *cast*)
  - Il y a vérification de type à l'exécution d'une conversion :

    ```
    I v = oC ;
    oD = (D)v; // compilable, mais lève l'exception ClassCastException
    ```

Relation d'implémentation

- Que se passe-t-il si une classe C n'implémente pas toutes les méthodes d'une interface ?
  - La classe devient *abstraite*
    - Elle doit être explicitement être définie comme telle
      - Mot-clé **`abstract`** écrit devant **`class C`**
    - Elle est non instanciable
      - Plus de **`new`** `C(…)` possible, mais constructeurs appelables dans les sous-classes
  - Nous verrons qu'elle peut être complétée grâce à l'héritage : certaines de ses sous-classes seront alors *concrètes* (soit non abstraites),

Classes abstraites (pas au programme)

- Une classe abstraite :
  - Est déclarée comme telle :
    - Mot-clé **`abstract`** écrit devant **`class`** …
  - Peut ne pas définir le corps d'une méthode de signature `s`
    - Mot-clé **`abstract`** écrit devant la signature de la méthode

```
abstract class … {
  …
 abstract s ;
  …
}
```

- Interface = classe abstraite dont toutes les méthodes seraient abstraites ? Non :
  - Pas de membres statiques, de constructeurs, de variables d'instances dans une interface, alors que possible dans une classe

Exemple non trivial – la programmation fonctionnelle

- Données :
  - nous disposons d'une fonction *minimum* capable de trouver le minimum d'une fonction réelle *f* sur un intervalle [*a*, *b*]
  - Nous disposons d'un catalogue de fonctions f_1, f_2, …, f_n dont nous voudrions pour chacune afficher la valeur minimale sur un intervalle [*inf*, *sup*] défini au départ
- Question : comment écrire un tel programme

Solution 1 : écrire autant de versions de *minimum* qu'il y a de fonctions à tester

- Duplication du code *n* fois
  - À la fois pour les fonctions à traiter

```
static double f1(double x) { ... }
...
static double fn(double x) { ... }

static double minimum_f1(double inf, double sup, double pas) {
  double min = Double.POSITIVE_INFINITY;
  for (double x=inf; x<=sup; x+=pas) {
    double y = f1(x);
    if (y<min) min = y;
  }
  return min;
}

static double minimum_f2(double inf, double sup, double pas) {
  ...
```

  - Eventuellement dans le programme principal

Solution 2 : repérer chaque fonction par un entier

```
static double f1(double x) { ... }
...
static double fn(double x) { ... }

static double minimum(int i, double inf, double sup, double pas) {
  double min = Double.POSITIVE_INFINITY;
  for (double x=inf; x<=sup; x+=pas) {
    double y;
    switch (i) {
      case 1: y = f1(x); break;
      ...
      case n:
      default: y = fn(x);
    }
    if (y<min) min = y;
  }
  return min;
}
```

- Problème : cette version de minimum ne sait traiter que les fonctions spécifiées ; comment intégrer une nouvelle fonction sans tout récrire et recompiler ?

Vers les interfaces

- Dans cet exemple, nous aimerions pouvoir :
  - Passer la fonction à traiter en paramètre de *minimum*
  - Disposer d'un tableau des fonctions à traiter dans le programme principal
- Constat :
  - En Java, les fonctions ne sont pas des objets
    - Une fonction ne peut pas être associée à une variable
- Solution :
  - Transformer les fonctions en des objets
    - Le procédé est appelé *réification* (réifier = définir sous forme d'objet)
    - Définir une interface qui spécifie ce qu'un objet fonction sait faire
    - Définir une classe qui l'implémente pour chaque fonction de l'exemple
    - Définir un objet de chaque classe fonction, référençables par des variables

Solution idéale

- Nous aimerions trouver le minimum des trois fonctions suivantes :
  - $\cos(x)$ sur l'intervalle $[-10,10]$
  - $(x-2)^2+3$ sur l'intervalle $[-5,5]$
  - $3-e^{-x^2}$ sur l'intervalle $[-10,10]$
- Trace du programme :

```
recherche du minimum sur [-10.0 , 10.0] de cos(x) :
  trouvé : -0.9999987317275397 ; attendu = -1.0
recherche du minimum sur [-5.0 , 5.0] de (x-2)^2+3 :
  trouvé : 3.0 ; attendu = 3.0
recherche du minimum sur [-10.0 , 10.0] de 3-exp(-x*x) :
  trouvé : 2.0 ; attendu = 2.0
```

Solution idéale

- Etape 1 : définir ce que sait faire un objet fonction :

```
interface uneFonction {
  double de(double x);
}
```

Si *f* est un objet `uneFonction`, alors *f*`.de(`*x*`)` calcule la valeur de *f*(*x*)

Solution idéale

- Etape 2 : définir la fonction *minimum*, la fonction à traiter *f* étant de type `uneFonction` :

```
static double minimum(uneFonction f, double inf, double sup, double pas) {
  double min = Double.POSITIVE_INFINITY;
  for (double x=inf; x<=sup; x+=pas) {
    double y = f.de(x);
    if (y<min) min = y;
  }
  return min;
}
```

Solution idéale

- Etape 3 : définir une classe qui implémente `uneFonction` par fonction à traiter :

```
class fonc1 implements uneFonction {
  @Override public double de(double x) { return Math.cos(x); };
}

class fonc2 implements uneFonction {
  @Override public double de(double x) { return exo1.sqr(x-2)+3; };
}

class fonc3 implements uneFonction {
  @Override public double de(double x) { return 3-Math.exp(-x*x); };
}
```

Solution idéale

- Etape 4 : nos objets fonctions sont désormais disponibles pour le programme principal :

```
class exo1 {

  public static void main(String[] args) {
    tester(new fonc1(), "cos(x)", -10, 10, -1);
    tester(new fonc2(), "(x-2)^2+3", -5, 5, 3);
    tester(new fonc3(), "3-exp(-x*x)", -10, 10, 2);
  }

  static void tester(uneFonction f, String exp, double inf, double sup,
                     double attendu) {
    Terminal.ecrireStringln("  recherche du minimum sur ["+inf+" , "+sup
                            +"] de "+exp+" :");
    Terminal.ecrireStringln("    trouvé : "+minimum(f, inf, sup, (sup-inf)/1000)
                            +" ; attendu = "+attendu);
  }

  static double sqr(double x) { return x*x; }
}
```

Solution idéale

- Etape 4 bis : réifions la notion de test

```
class unTest {
  uneFonction fonction;
  String      expression;
  double      inf,
              sup,
              attendu;

  unTest(uneFonction f, String e, double inf, double sup, double attendu) {
    this.fonction = f;
    this.expression = e;
    this.inf = inf;
    this.sup = sup;
    this.attendu = attendu;
  }

  unTest effectuer() {
    Terminal.ecrireStringln("  recherche du minimum sur ["+this.inf+" , "
                            +this.sup+"] de "+this.expression+" :");
    Terminal.ecrireStringln("    trouvé : "+exo1bis.minimum(this.fonction, this.inf, th
                            +" ; attendu = "+this.attendu);
    return this;
  }
}
```

Solution idéale

- Etape 4 bis (suite) : d'où le programme principal

```
class exo1bis {

  static double minimum(uneFonction f, double inf, double sup, double pas) {
    double min = Double.POSITIVE_INFINITY;
    for (double x=inf; x<=sup; x+=pas) {
      double y = f.de(x);
      if (y<min) min = y;
    }
    return min;
  }

  public static void main(String[] args) {
    unTest[] lesTests = new unTest[] {
      new unTest(new fonc1(), "cos(x)", -10, 10, -1),
      new unTest(new fonc2(), "(x-2)^2+3", -5, 5, 3),
      new unTest(new fonc3(), "3-exp(-x*x)", -10, 10, 2)
    };
    for (unTest test : lesTests)
      test.effectuer();
  }

  static double sqr(double x) { return x*x; }
}
```

Amélioration 1

- Toute classe peut redéfinir la méthode :

  `public` `String toString()`

  qui transforme automatiquement tout objet `o` en sa version textuelle (`String`) quand le contexte le nécessite

  – Conversion automatique vers `String`

- Méthode redéfinie pour chaque fonction :

```
class fonc1 implements uneFonction {
  @Override public double de(double x) { return Math.cos(x); };
  @Override public String toString() { return "cos(x)"; }
}
```

Amélioration 1

- Le texte de l'expression dans un test est inutile :

```
class unTest {
  uneFonction fonction;
  String      expression;
  double      inf,
              sup,
              attendu;

  unTest(uneFonction f, String e, double inf, double sup, double attendu) {
    this.fonction = f;
    this.expression = e;
    this.inf = inf;
    this.sup = sup;
    this.attendu = attendu;
  }

  unTest effectuer() {
    Terminal.ecrireStringln("  recherche du minimum sur ["+this.inf+" , "
                            +this.sup+"] de "+this.expression+" :");
    Terminal.ecrireStringln("    trouvé : "+exo1bis.minimum(this.fonction, this.inf, th
                            +" ; attendu = "+this.attendu);
    return this;
  }
}
```

fonction

Amélioration 1

- Idem pour le programme principal

```
class exo1bis {

  static double minimum(uneFonction f, double inf, double sup, double pas) {
    double min = Double.POSITIVE_INFINITY;
    for (double x=inf; x<=sup; x+=pas) {
      double y = f.de(x);
      if (y<min) min = y;
    }
    return min;
  }

  public static void main(String[] args) {
    unTest[] lesTests = new unTest[] {
      new unTest(new fonc1(), "cos(x)", -10, 10, -1),
      new unTest(new fonc2(), "(x-2)^2+3", -5, 5, 3),
      new unTest(new fonc3(), "3-exp(-x*x)", -10, 10, 2)
    };
    for (unTest test : lesTests)
      test.effectuer();
  }

  static double sqr(double x) { return x*x; }
}
```

Amélioration 2 (pas au programme)

- Constat :
  - Il faut définir autant de classes implémentant `uneFonction` qu'il y a de fonctions à traiter
- Il est possible, à partir d'une interface
  - De définir une classe qui l'implémente sans la nommer (classe *anonyme locale*)
  - Dans la foulée, d'en créer une instance
    - Très utilisé dans le code des interfaces graphiques (`Listener`) $\Rightarrow$ NFA035 ?

Amélioration 2

- Pour le programme principal :
  - Avant :

```
public static void main(String[] args) {
  unTest[] lesTests = new unTest[] {
    new unTest(new fonc1(), -10, 10, -1),
    new unTest(new fonc2(), -5, 5, 3),
    new unTest(new fonc3(), -10, 10, 2)
  };
  for (unTest test : lesTests)
    test.effectuer();
```

  - Après :
    - Supprimer toutes les classes $\texttt{fonc}_i$

Amélioration 2

- Programme principal (suite *Après*) :

```
public static void main(String[] args) {
  unTest[] lesTests = new unTest[] {
    new unTest(new uneFonction () {
                  @Override public double de(double x) { return Math.cos(x); };
                  @Override public String toString() { return "cos(x)"; }
               },
               -10, 10, -1),
    new unTest(new uneFonction () {
                  @Override public double de(double x) { return exo1bis.sqr(x-2)+3; };
                  @Override public String toString() { return "(x-2)^2+3"; }
               },
               -5, 5, 3),
    new unTest(new uneFonction () {
                  @Override public double de(double x) { return 3-Math.exp(-x*x); };
                  @Override public String toString() { return "3-exp(-x*x)"; }
               },
               -10, 10, 2)
  };
  for (unTest test : lesTests)
    test.effectuer();
```

Exercices

- Exercice 1
  - Composition de fonctions
- Exercice 2
  - Programmation fonctionnelle : transformations de tableaux via des fonctions