LU2IN002
Introduction à la Programmation Orientée Objet

Responsable de l'UE et du cours du vendredi :
Christophe Marsala
(e-mail : Christophe.Marsala@lip6.fr)

Cours du lundi : Sabrina Tollari
(e-mail : Sabrina.Tollari@lip6.fr)

(support réalisé à partir de ceux de Christophe Marsala et de Vincent Guigue)

SCIENCES SORBONNE UNIVERSITÉ

Cours 11 – vendredi 29 novembre 2024

Programme du jour

1. Conteneurs génériques
   - Création d'une classe générique
   - Extension d'une classe générique
   - *wildcard*
2. Design Patterns
3. Exercices d'annales
   - Contrôle de novembre 2024 : questions des Quizzes
     - Quiz : égalité entre 2 vêtements
     - Quiz : redéfinition complexe
   - Examen de janvier 2020 (2019oct)
     - Exercice 1 : Combien d'instances, quelle méthode ?

Programme

Problématique générale (1/2)

Conteneurs génériques

- Construire une structure de données adaptée à différents types d'entrées. Exemples :
  - Liste d'entiers, liste de réels, liste de `String`, liste de `Point`...
- ... pour ne pas construire une classe pour chaque cas !!!

Solution avant les génériques (avant Java 1.5) :

```
1  public class ListeGenOld {
2      private final static int TAILLE_MAX = 500;
3      private Object[] liste;
4      private int size;
5      public ListeGenOld(){
6          liste = new Object[TAILLE_MAX];
7          size  = 0;
8      }
9      public void add(Object o){liste[size] = o; size ++;}
10     public Object get(int i){return liste[i];}
11     ...
12 }
13 // main : une liste de Point ?
14 ListeGenOld liste = new ListeGenOld();
15 liste.add(new Point()); // OK
16 liste.add("Bonjour"); // OK !!!
17 Point p = (Point)liste.get(0); // cast obligatoire
```


Problématique générale (2/2)

- Limites de la solution précédente :
  - Obligation de faire des `cast` à chaque récupération d'objet
  - Faible sécurisation : on peut mettre n'importe quoi dans la structure...
  - Difficilement compatible avec des algorithmes génériques (tri, min, max...)
- **Solution (depuis Java 1.5) : utiliser des génériques**

Classes, interfaces ou méthodes paramétrées par un ou plusieurs types, appelés paramètres de type (*type parameters*)

Exemple : type générique `ArrayList<E>` avec paramètre de type `E`

```
ArrayList<Integer> listeEntiers = new ArrayList<Integer>();
ArrayList<Double> listeReels = new ArrayList<Double>();
ArrayList<String> listeString = new ArrayList<String>();
ArrayList<Point> listePoints = new ArrayList<Point>();
```

- Pas besoin de caster : `Point p = listePoints.get(0);` OK
- Seuls les points sont acceptés dans la liste
- Une seule classe `ArrayList` ⇒ plus facilement compatibles avec différent algorithmes

Conteneurs génériques

Le package `java.util` propose de nombreux conteneurs génériques. Exemples :

- HashSet, HashMap
- ArrayList, ArrayDeque
- TreeMap, TreeSet
- LinkedList
- LinkedHashSet, LinkedHashMap

General-purpose Implementations

Interfaces	Hash table Implementations	Resizable array Implementations	Tree Implementations	Linked list Implementations	Hash table + Linked list Implementations
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Création d'une classe générique...

Exemple (très) utile : la `Paire`

La plupart des langages modernes gèrent des N-uplets... Mais pas Java. On peut créer une classe `Paire` pour retourner facilement plusieurs valeurs depuis une méthode.

```
public class Paire<A,B> { // 2 paramètres de type : A et B
    private A el1;
    private B el2;
    public Paire(A el1, B el2) {
        this.el1 = el1;
        this.el2 = el2;
    }
    public A getEl1() {
        return el1;
    }
    public B getEl2() {
        return el2;
    }
}
```


... Et usage client

La syntaxe est celle des ArrayList... Vous la connaissez déjà!

```
Paire<Integer,String> p1=new Paire<Integer,String>(10,"bonjour");
Integer x=p1.getEl1();
String s1=p1.getEl2();
```

- Le type est donc : `Paire<Integer, String>`
- Le type du contenant est passé en argument *spécial* entre chevrons <>

On peut utiliser la même classe paire, mais avec d'autres types entre chevrons

```
Paire<String,Point> p2
            = new Paire<String,Point>("toto",new Point(1,2));
String s2=p2.getEl1();
Point p=p2.getEl2();
```


Extension d'une classe générique (1)

Besoin d'une liste de *quelque chose*

- Aquarium = liste de poissons
- Train = liste de wagons
- Population = liste de personnes
- ...

```
public class Aquarium extends ArrayList<Poisson> {
   // bcp de méthodes héritées !!!
    ...
}
```

Usage client : la liste ne gère que des poissons

```
Aquarium aqua = new Aquarium();
aqua.add(new Thon());
aqua.add(new Requin());
```


Extension d'une classe générique (2)

Besoin d'une liste avec des méthodes spécifiques...
... mais toujours générique

Exemple : récupération de l'élément du milieu

```
public class MaListeMilieu<E> extends ArrayList<E> {
    // constructeur sans argument par défaut
    // Les méthodes sont héritées : add, get, size...

    // Méthode spécifique
    public E getMilieu(){
        return super.get(super.size()/2); // division entière
    }

}
```

Usage client :

```
MaListeMilieu<Double> li = new MaListeMilieu<Double>();
li.add(2.0);
li.add(1.4);
li.add(3.7);
Double x = li.getMilieu();
```


Cast sur les objets génériques <E>

1. Coté *contenu* : très agréable (et classique)
   - objets de types `E` et descendants de `E`
   - récupération d'objets dans des variables `E`

2. Coté *contenant* : non flexible

```
ArrayList<Personne> pop = new ArrayList<Personne>(); // OK

// Etudiant extends Personne
ArrayList<Personne> promo = new ArrayList<Etudiant>(); // KO !!!
```

⇒ Une seule issue (en cas de besoin) : la syntaxe *wildcard*

La syntaxe *wildcard*

Subsomption *contenu/contenant*

On a besoin de cette propriété pour définir des algorithmes génériques. Syntaxes :

- `ArrayList<?>` : n'importe quelle liste
- `ArrayList<? extends Poisson>` : n'importe quelle liste d'objets dérivés de poissons

```
ArrayList<? extends Poisson> li = new ArrayList<Thon>();
```

ATTENTION : ce type de syntaxe empêche toute modification sur l'objet passé

Exemple : comment proposer une technique de **recherche de minimum** dans une liste **sans connaître le type de contenu** ?

1. Définir une propriété (interface) : Comparable
2. Définir un algorithme acceptant n'importe quel conteneur d'objets comparables en utilisant la syntaxe *wildcard*

La syntaxe *wildcard* (préliminaires)

1 Définir une propriété : Comparable

```
public interface Comparable<E> {
    //retourne -1 si courant < obj, 0 si égalité, 1 sinon
    public int compareTo(E obj);
}
```

Avec par exemple un Poisson répondant à la spécification :

```
public class Poisson implements Comparable<Poisson> {
    private double taille;

    public Poisson(double taille) {
        this.taille = taille;
    }
    public double getTaille() {
        return taille;
    }
    public int compareTo(Poisson obj) {
        if(taille<obj.taille)
            return -1;
        else if(taille==obj.taille)
            return 0;
        else
            return 1;
    }
```


La syntaxe *wildcard*

2 Utiliser la propriété dans un algorithme générique :

```
public class GenericTools<E extends Comparable<E>> {
    ...

    public E getMinimum(ArrayList<? extends E> liste){
        E min = liste.get(0);
        for (int i=1; i<liste.size(); i++){
            // si : min > liste.get(i)
            if(min.compareTo(liste.get(i)) == 1)
                min = liste.get(i);
        }
        return min;
    }
}
```

Usage coté client :

```
ArrayList<Poisson> aquarium = new ArrayList<Poisson>();
GenericTools<Poisson> tool = new GenericTools<Poisson>();
Poisson lePlusPetit = tool.getMinimum(aquarium);
```


Algorithmes génériques

Classe algorithmique de gestion des listes générique

la classe Collections

- min, max, sort, shuffle, indexOf, frequency...

Quelques exemples d'outils disponibles :

Programme

De la Bidouille OO à la Programmation OO

Principes Objets :

- Encapsulation
- Héritage & abstraction
- Polymorphisme

Question (difficile)

Comment combiner ces différents outils pour écrire des programmes ou boîtes-à-outils flexibles, réutilisables, efficaces, etc... ?

Patrons de conception (design patterns)

Définition : design pattern

Élément de conception réutilisable permettant de résoudre un problème récurrent en programmation orientée objet.

- Historique :
  - analogie avec une méthode de conception d'immeuble en architecture [Alexander, 77]
  - introduites par le "GOF" dans le livre Design Patterns en 1999
    - dans le "GOF" 23 patterns standards

Pourquoi les patterns ?

Des recettes d'expert ayant fait leurs preuves.
Un vocabulaire commun pour les architectes logiciels.
Approche incontournable dans le monde de la P.O.O.

Remarque : certains patterns sont déjà inclus dans le langage...

Idiomes de la programmation

Portion de code Java que l'on utilise lorsque l'on a à résoudre un problème récurrent

- Parcours d'un tableau

```
for(int i=0;i<tableau.length;i++) {
   tableau[i] = ...
}
```

- Gestion d'exception

```
try {
  XYZ(...);
} catch(XYZException e) {
      e.printStackTrace(System.err);
}
```


Conception vs. Programmation

Un design pattern est un élément de conception (objet)
Remarque : ce n'est pas au niveau du langage de programmation (donc **pas spécifique à Java**, un pattern est aussi valable en C++, en Python...)

Citations

"Chaque patron décrit un problème qui se manifeste constamment dans notre environnement, et donc décrit le coeur de la solution à ce problème, d'une façon telle que l'on puisse réutiliser cette solution des millions de fois, sans jamais le faire deux fois de la même manière" Christopher Alexander - 1977.

"Les patrons offrent la possibilité de capitaliser un savoir précieux né du savoir-faire d'experts" [Buschmann] - 1996.

Coté implémentation : grands principes

Principe 1

Favoriser la composition (lien dynamique, flexible) sur l'héritage (lien statique, peu flexible)

Attention : favoriser ne veut pas dire remplacer systématiquement, l'héritage est largement utilisé aussi !

Principe 2

Les clients programment en priorité pour des interfaces (ou abstractions, classes abstraites, etc) plutôt qu'en lien direct avec les implémentations (classes concrètes)

Différents types de patterns

- Pattern de création : utilisé pour la création d'objets
  - par exemple : singleton, prototype, factory,...
- Pattern structurel : utilisé pour rajouter des fonctionnalités à une classe
  - par exemple : decorator, composite,...
- Pattern comportemental : utilisé pour mettre en œuvre des moyens de communication entre objets
  - par exemple : iterator, strategy,...

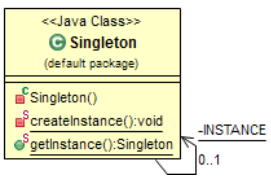
Description standard

Nom (classification)	singleton (création)
Intention	description générale et succinte
Alias	autres noms connus pour le pattern
Motivation	au moins 2 exemples/scénarios
Indications d'utilisation	liste des situations qui justifient de l'utilisation du pattern
Structure	diagramme de classes UML indépendant du langage
Constituants	explication des différentes classes intervenant dans le pattern
Implémentation	principes, pièges, astuces, techniques pour implanter le pattern dans un langage objet donné
Utilisations remarquables	programmes réels qui l'utilisent
Limites	limites concernant son utilisation

Singleton (Construction)

Garantir l'unicité d'une instance

- A utiliser quand il ne peut y avoir qu'une instance (eg Console)
- Fournir un accès à cette instance



Singleton (Construction)

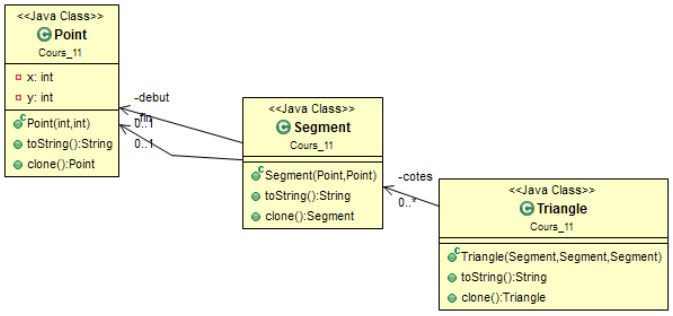
Dans cette version de `Singleton`, l'instance est créée seulement si la méthode `getInstance()` est appelée au moins une fois

```java
public class Singleton {
    private static Singleton INSTANCE = null;

    private Singleton() {}

    private static void createInstance() {
        if (INSTANCE == null) {
            INSTANCE = new Singleton();
        }
    }
    public static Singleton getInstance() {
        if (INSTANCE == null) createInstance();
        return INSTANCE;
    }
}
```


Prototype (Construction)

Créer un nouvel objet à partir d'un objet existant

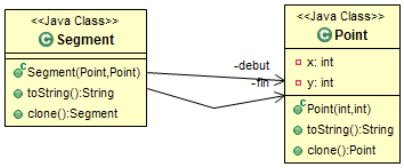
- Fournir une copie exacte de l'objet existant
- Fournir une interface commune aux objets



Composite (Structure)

Un objet E, composé d'objets E

- une sous-figure, composée de figure
- une stratégie, composée de sous-stratégie




```java
public class Segment {
        private Point debut, fin;
        public Segment(Point p1, Point p2) {
                debut = p1; fin = p2;
        }

        public String toString() {
                return "Segment [debut=" + debut + ", fin=" + fin
        }
        public Segment clone() {
                return new Segment(debut.clone(), fin.clone());
        }
}
```


Iterator (comportement)

Objectif

- Découpler le choix des structures de données des implémentations d'algorithmes
- Proposer une interface de parcours d'un ensemble d'objets quelle que soit la structure associée (liste, pile, arbre,...)

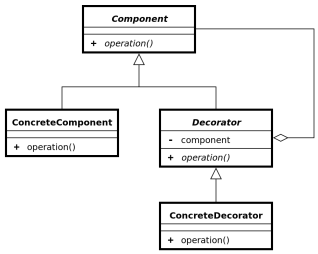
```java
// code d'utilisation spécifique
Iterator<MyType> iter = list.iterator();

// code commun à toute structure de données
while (iter.hasNext()) {
    System.out.print(iter.next());
    if (iter.hasNext())
        System.out.print(", ");
}
```


Decorator (Structure)

Ajouter une fonctionnalité...

... sur n'importe quel objet d'une arborescence de classe

- Permettre de lire des informations de haut niveau (String, double...) dans n'importe quel flux
- Rendre n'importe quelle stratégie prudente

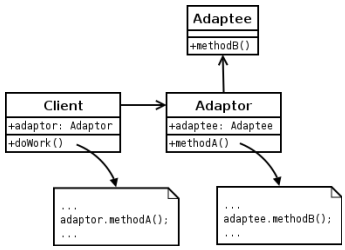



```java
public MonObjetDecore extends MonObjet{
  private MonObjet obj;
  public MonObjetDecore(MonObjet obj){ this.obj = obj;}

  public void mafonction(){
    if(cas 1) return obj.mafonction()
    else // code spécifique
  }
```


Adapter (Structurel)

Idée : réutiliser une fonction déjà implémentée...

... Mais dans une architecture contrainte
= Adapter la classe
Repose sur le principe de la délégation

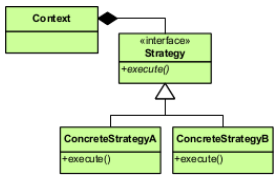


- Très très fréquent : réutiliser une classe sans la modifier... Alors que le client est déjà spécifié
  - on a un objet d'une classe `Client` qui veut utiliser un objet de classe B mais le client ne sait manipuler des objets de classe A. Il faut donc adapter la classe B à l'interface de A pour que le Client puisse finalement utiliser B.
- Technique d'optimisation, calcul de graphes...
  - Adapter les objets à passer en paramètres... Ou adapter les algorithmes.

Strategy (Comportement)

Gérer le comportement distinctement de l'objet

- Robot... Qui marche, rampe, vole... Ou une combinaison des 3
- Le client gère la stratégie
- On peut créer de nouvelles stratégies avec des robots existants




```java
public class Robot{
  private Strategy str;
  public Robot(Strategy str){this.str = str;}

  public void action(){
    str.action(); // ou str.action(this);
  }
```

⇒ Une alternative (beaucoup plus flexible) à l'héritage sur les robots

Note : DP compatible avec `Composite`

Et plein d'autres encore

- Visitor
  - vient éxecuter un algorithme dans un objet
- MVC : model view controler
  - pour les interfaces graphiques, séparation des éléments clés
- ...

Comme un second niveau de programmation (de la conception en fait !)
A continuer avec de l'UML et d'autres UE de génie logiciel

- UE L3 LU3IN002 "Programmation par objets"

Programme

Exo 1 Questions issues des Quizzes : Q1.1

```java
public class Vetement {
  private int taille ;
  public Vetement(int t){
    taille=t;
  }
}
```

```java
public class Chemise extends Vetement{
  public Chemise(int t) { super(t); }
}
public class Pantalon extends Vetement{
  public Pantalon(int t) { super(t); }
}
```

On suppose que deux vêtements sont égaux s'ils sont de même type et de même taille. Écrire la **redéfinition** de la méthode standard `equals` pour la classe `Vetement`.

```java
// Dans quelle classe ? Vetement
public boolean equals (Object o) {
    if (o == this) return true; // optionnel
    if (o == null) return false;
    if (getClass()!=o.getClass()) // même type de vêtement
        return false;
    Vetement v=(Vetement)o;
    return taille==v.taille; // même taille de vêtement
}
```

△ Redéfinition ⟹ même signature ⟹ le paramètre est obligatoirement `Object`

Exo 1 Questions issues des Quizzes : Q1.2 (1/5)

```java
public class X {}
public class Y extends X {}
public class Z extends Y {}
public class Mere {
   public void meth(X x) {System.out.println("Mere : meth(X)");}
   public void meth(Y y) {System.out.println("Mere : meth(Y)");}
}
public class Fille extends Mere {
   public void meth(Y y) { System.out.println("Fille : meth(Y)");}
   public void methDeFille(){System.out.println("meth de Fille");}
}
// main
Mere mf=new Fille(); // subsomption
```

- surcharge : `meth(X x)` et `meth(Y y)` dans `Mere`
  ⟹ `meth(X x)` et `meth(Y y)` accessibles dans `Mere` et `Fille`
- redéfinition : `meth(Y y)` dans `Mere`, redéfinie dans `Fille`
  ⟹ la méthode appelée dépend de l'objet, pas de la variable qui référence l'objet
- `methDeFille()` accessible que dans `Fille`

Exo 1 Questions issues des Quizzes : Q1.2 (2/5)

```java
public class X {}
public class Y extends X {}
public class Z extends Y {}
public class Mere {
   public void meth(X x) {System.out.println("Mere : meth(X)");}
   public void meth(Y y) {System.out.println("Mere : meth(Y)");}
}
public class Fille extends Mere {
   public void meth(Y y) { System.out.println("Fille : meth(Y)");}
   public void methDeFille(){System.out.println("meth de Fille");}
}
// main
Mere mf=new Fille(); Y y=new Y();
```

1) `mf.meth( y );`

- La variable `mf` est de type `Mere`
  ⟹ Compilation OK, car il existe une méthode de signature de `meth(Y)` dans `Mere`
- L'objet référencé par `mf` est de type `Fille`
  ⟹ Exécution : la méthode appelée est celle redéfinie dans `Fille`
- Affichage : `"Fille : meth(Y)"`

Exo 1 Questions issues des Quizzes : Q1.2 (3/5)

```java
public class X {}
public class Y extends X {}
public class Z extends Y {}
public class Mere {
   public void meth(X x) {System.out.println("Mere : meth(X)");}
   public void meth(Y y) {System.out.println("Mere : meth(Y)");}
}
public class Fille extends Mere {
   public void meth(Y y) { System.out.println("Fille : meth(Y)");}
   public void methDeFille(){System.out.println("meth de Fille");}
}
// main
Mere mf=new Fille(); Z z=new Z();
```

2) `mf.meth( z );`

- La variable `mf` est de type `Mere`
  - Il n'existe pas de méthode de signature `meth(Z)` dans `Mere`...
  - ... mais il existe 2 méthodes compatibles (grâce à l'héritage), on prend la méthode dont le paramètre est le plus proche de `Z`

  $\Longrightarrow$ Compilation OK, car `Z` est un `Y` et `meth(Y)` existe dans `Mere`
- L'objet référencé par `mf` est de type `Fille` (qui redéfinit `meth(Y)`)
- Affichage : `"Fille : meth(Y)"`

Exo 1 Questions issues des Quizzes : Q1.2 (4/5)

```java
public class X {}
public class Y extends X {}
public class Z extends Y {}
public class Mere {
   public void meth(X x) {System.out.println("Mere : meth(X)");}
   public void meth(Y y) {System.out.println("Mere : meth(Y)");}
}
public class Fille extends Mere {
   public void meth(Y y) { System.out.println("Fille : meth(Y)");}
   public void methDeFille(){System.out.println("meth de Fille");}
}
// main
Mere mf=new Fille(); X xy=new Y();
```

3) `mf.meth( xy );`

- La variable `mf` est de type `Mere`, le paramètre est de type `X`

  $\Longrightarrow$ Compilation OK, car `meth(X)` existe dans `Mere`
- L'objet référencé par `mf` est de type `Fille` qui ne redéfinit pas la méthode `meth(X)`
- Affichage : `"Mere : meth(X)"`

Exo 1 Questions issues des Quizzes : Q1.2 (5/5)

```java
public class X {}
public class Y extends X {}
public class Z extends Y {}
public class Mere {
   public void meth(X x) {System.out.println("Mere : meth(X)");}
   public void meth(Y y) {System.out.println("Mere : meth(Y)");}
}
public class Fille extends Mere {
   public void meth(Y y) { System.out.println("Fille : meth(Y)");}
   public void methDeFille(){System.out.println("meth de Fille");}
}
// main
Mere mf=new Fille();
```

3) `mf.methDeFille();`

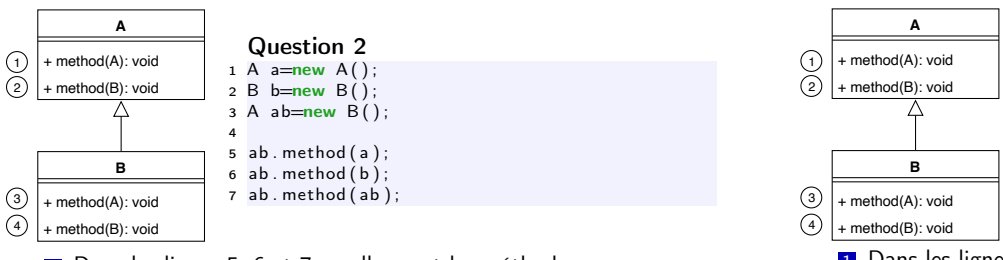
- La variable `mf` est de type `Mere`

  $\Longrightarrow$ Compilation **KO**, car pas d'accès aux méthodes des classes filles
- L'objet référencé par `mf` est de type `Fille` $\Longrightarrow$ on peut caster

```java
((Fille)mf).methDeFille(); // OK
```


Examen Janvier 2020 (2019oct) : Exercice 1

Exercice 1 : Combien d'instances, quelle méthode?



Question 1

```java
1 A[] tab = new A[5];
2 for(int i=0; i<4; i++) {
3   if(i==0)
4     tab[i] = new A();
5   else
6     tab[i] = new B();
7 }
8 tab[1] = tab[0];
9 A a = new A();
10 tab[2] = a;
```

1. Combien d'instances de A et de B ont été créées au total? Donner les 2 chiffres.
2. Combien en reste-t-il à l'issue de l'exécution du programme?

Solutions

1. **2 instances de A** (ligne 4 (pour i=0) et ligne 9)
   3 instances de B (lignes 6 (3 fois pour i=1,2,3))
2. Les instances référencées par `tab[1]` et `tab[2]` ont été déférencées (lignes 8 et 10), il reste donc **2 instances de A et 1 instance de B**

Examen Janvier 2020 (2019oct) : Exercice 1

Question 2

```java
1 A a=new A();
2 B b=new B();
3 A ab=new B();
4
5 ab.method(a);
6 ab.method(b);
7 ab.method(ab);
```

1. Dans les lignes 5, 6 et 7, quelles sont les méthodes pré-sélectionnées par le **compilateur**?
2. Quelles sont les méthodes exécutées par la **JVM**?

Solutions

1. Présélection : ① ② ① (la variable `ab` est de type `A`, elle n'a accès que aux méthodes de `A`)
2. Exécution : ③ ④ ③ (dépend de l'instance référencée par `ab` qui est de type `B`)

Examen Janvier 2020 (2019oct) : Exercice 1

Question 3 : dans le cas où la méthode ② n'existe plus

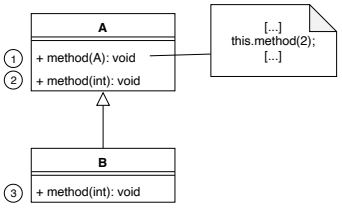
```java
1 A a=new A();
2 B b=new B();
3 A ab=new B();
4
5 ab.method(a);
6 ab.method(b);
7 ab.method(ab);
```

1. Dans les lignes 5, 6 et 7, quelles sont les méthodes pré-sélectionnées par le **compilateur**?
2. Quelles sont les méthodes exécutées par la **JVM**?

Solutions

1. Présélection : ① ① ① (une seule méthode disponible à partir de la variable `ab` de type `A`, celle de signature `method(A)` ; ligne 6, l'argument `b` de type `B` hérite de `A`, ① peut donc être utilisée)
2. Exécution : ③ ③ ③ (seule une méthode de signature `method(A)` peut être appelée. Ici, l'instance référencée par `ab` qui est de type `B`, c'est donc ③ qui est appelée)

Examen Janvier 2020 (2019oct) : Exercice 1



Question 4

```
1 A a=new A();
2 B b=new B();
3 A ab=new B();
4
5 a.method(a);
6 b.method(a);
7 ab.method(a);
```

1. En considérant la nouvelle architecture. Quelle méthode est invoquée à l'intérieur de ① dans les 3 appels?

Solutions

1. Exécution : ② ③ ③ (dépend de l'instance)