Polytech Paris-Sud
Formation initiale 3e année
Spécialité Informatique
Année 2016-2017

UML

Interlude
De la conception à la programmation, exemples avec Java

Delphine Longuet
delphine.longuet@lri.fr

http://www.lri.fr/~longuet/Enseignements/16-17/Et3-UML

De la conception à la programmation

Objectif : Obtenir un squelette du programme à partir du diagramme de classes conception.

Peut être fait en deux étapes :

1. Raffiner le diagramme de classes en un ensemble de classes (plus d'associations)
2. Implémenter chaque classe dans le langage de programmation choisi

Séparation des problèmes :

1. Association concrétisée en attribut, classe, opération ?
2. Détails d'implémentation et contraintes du langage choisi

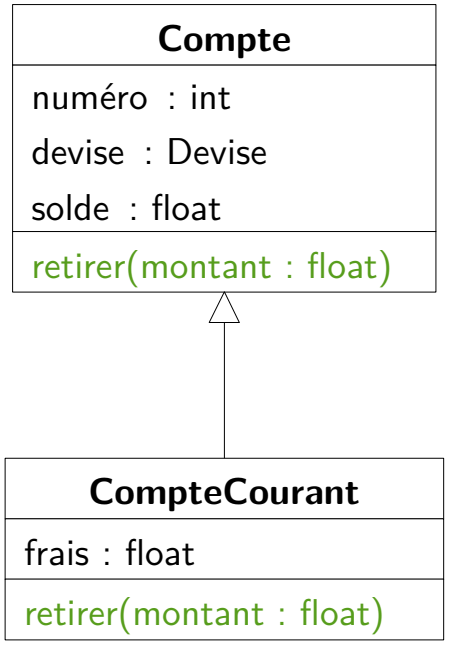
Correspondances UML/Java

UML	Java
Classe du diagramme	`class`
Héritage	`extends`
Classe/opération abstraite	`abstract`
Interface	`interface`
Redéfinition	`@Override`
Association (multiplicité, navigabilité...)	?
Classe-association	?
Composition	?
Agrégation	?

Dans la suite : aucune règle générale, seulement des exemples

En pratique : chaque cas demande réflexion et traitement ad-hoc

Héritage et redéfinition



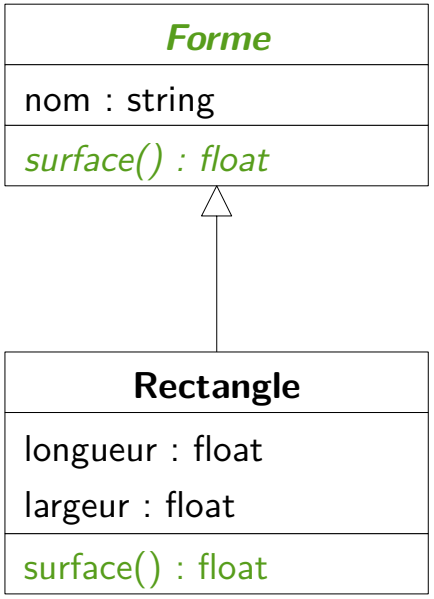

```
public class Compte {
    private int numero;
    private Devise devise;
    private float solde;

    public void retirer(float montant) {
        if(solde - montant >= 0) {
            solde = solde - montant;
        }
        return;
    }
}
```

```
public class CompteCourant extends Compte {
    private float frais;

    @Override
    public void retirer(float montant) {
        if(solde - montant >= 0) {
            solde = solde - montant;
        } else {
            solde = solde - montant - frais;
        }
        return;
    }
}
```

Classes et opérations abstraites



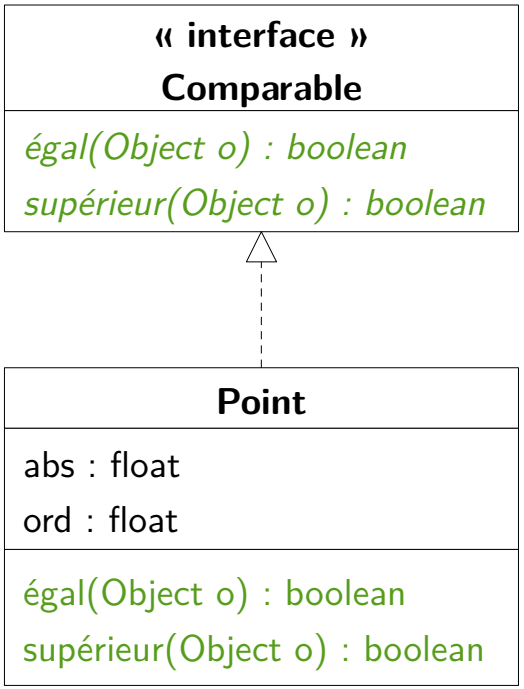

```
public abstract class Forme {
    private String nom;

    public abstract float surface();
}
```

```
public class Rectangle extends Forme {
    private float longueur;
    private float largeur;

    public float surface() {
        return (longueur * largeur);
    }
}
```

Interfaces



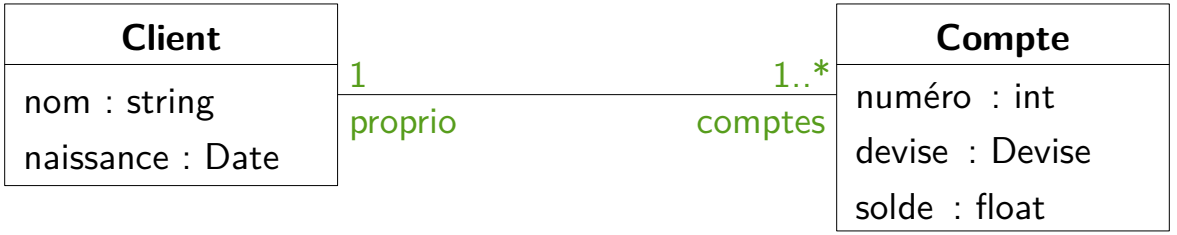

```
public interface Comparable {
    public boolean egal(Object o);
    public boolean superieur(Object o);
}
```

```
public class Point implements Comparable {
    private float abs;
    private float ord;

    @Override
    public boolean egal(Object o) {
        Point p = (Point) o;
        return (this.abs == p.abs
            && this.ord == p.ord);
    }

    @Override
    public boolean superieur(Object o) {
        Point p = (Point) o;
        return (this.abs >= p.abs
            && this.ord >= p.ord);
}
```

Associations

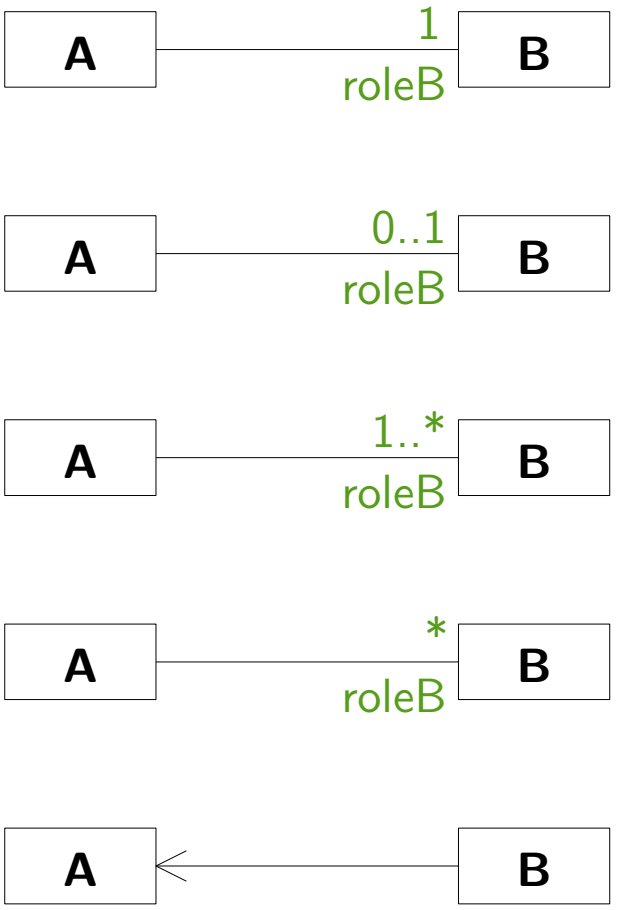



```
public class Client {
    private Sring nom;
    private Date naissance;
    private Set<Comptes> comptes;
}
```

```
public class Compte {
    private int numero;
    private Devise devise;
    private float solde;
    private Client proprio;
}
```

Set = tableau, ArrayList, HashSet... en fonction de l'utilisation prévue de cet attribut (complexité du parcours de la structure, de la recherche, de l'insertion, nombre d'éléments maximum...)

Associations, pseudo-règles




```
public class A {
    private B roleB;
}
```

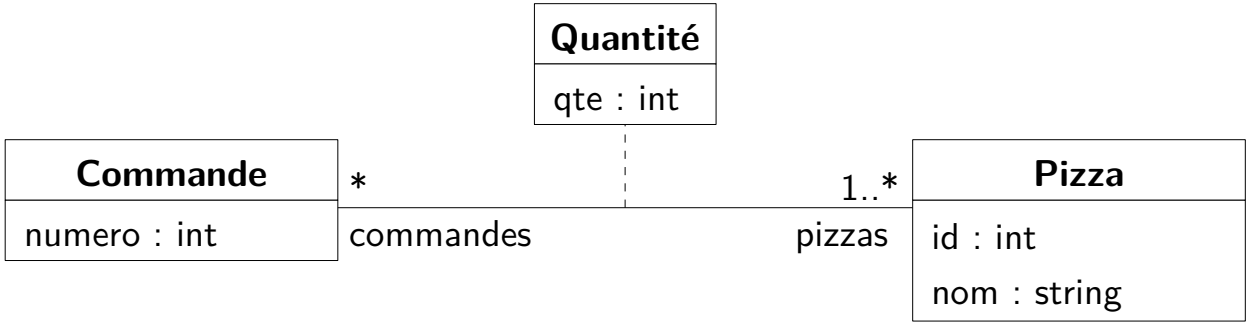
Si pas de B, `roleB = null`

```
public class A {
    private Set<B> roleB;
}
```

Si pas de B, `roleB.isEmpty() = true`

```
public class A {
    // pas d'accès possible à B
}
```

Classes-associations (1)




```
public class Commande {
    private int numero;
    private Map<Pizza,Integer> pizzas;
}
```

On peut choisir autre chose que Map

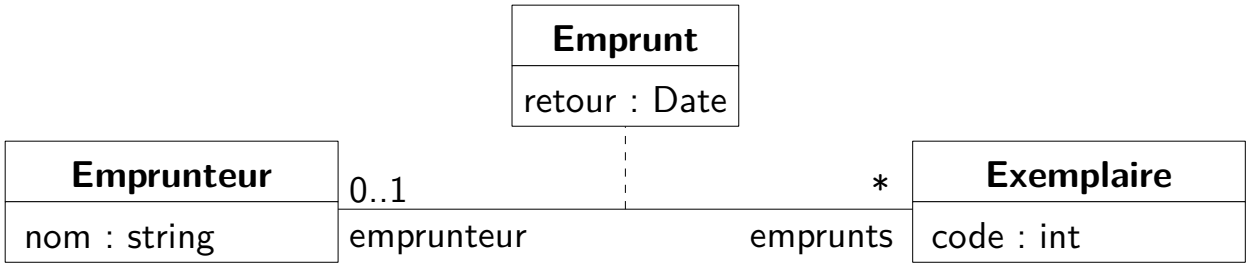
Contrainte d'unicité sur les pizzas pour en faire des clés

```
public class Pizza {
    private int id;
    private String nom;
    private Set<Commande> commandes;
// ou bien
    private Map<Commande,Integer> commandes;
}
```

Ou même, pas navigable dans ce sens-là

(De quelles informations a-t-on réellement besoin ?)

Classes-associations (2)




```
public class Emprunteur {
    private String nom;
    private Map<Exemplaire,Date> emprunts;
}
```

Contrainte d'unicité sur les exemplaires

pour pouvoir en faire une clé de la Map

```
public class Exemplaire {
    private int code;
    private Emprunteur emprunteur;
    private Date retour;
}
```

Contrainte `emprunteur == null`

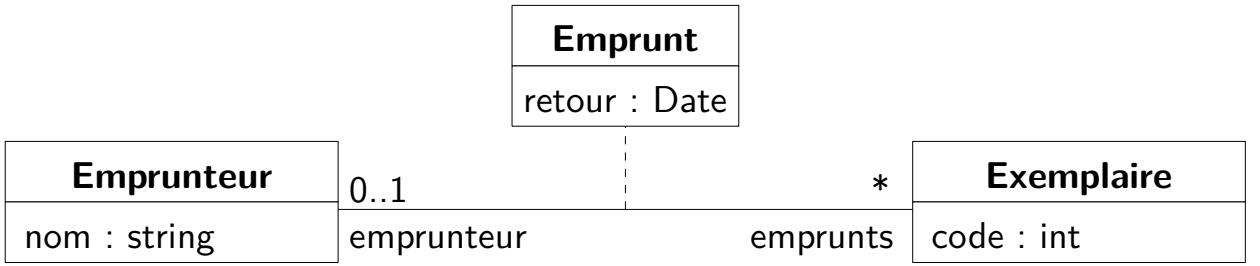
ssi `retour == null`

ou bien une classe `Pair`

Classes-associations (2)

Si peu de couples (Emprunteur,Exemplaire), recherche fastidieuse des exemplaires sortis, depuis Exemplaire comme depuis Emprunteur

Un solution : une classe Emprunt qui regroupe ces couples

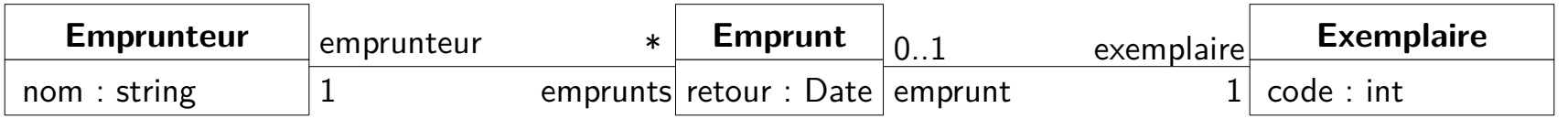



```
public class Emprunteur {
     private String nom;
     private Set<Emprunts> emprunts;
}
```

```
public class Emprunt {
     private Date retour;
     private Emprunteur emprunteur;
     private Exemplaire exemplaire;
}
```

```
public class Exemplaire {
     private int code;
     private Emprunt emprunt;
}
```

Ce qui correspond finalement à l'interprétation de la classe-association où Emprunt est une classe à part entière



Agrégation et composition


```
public class ListeLecture {
     private ArrayList<Morceau> morceaux;

     public ListeLecture() {
          morceaux = new ArrayList<Morceau>();
     }

     public void add(Morceau m) {
          this.morceaux.add(m);
     }
}
```

```
public class Album {
     private String nom;
     private String artiste;
     private ArrayList<Morceau> morceaux;

     public Album(String artiste,String nom,Morceau m) {
          ...
          this.morceaux = new ArrayList<Morceau>();
          this.add(m);
     }

     public void add(Morceau m) {
          this.morceaux.add(m);
     }
}
```

```
public class Morceau {
     private String titre;
     private Album album;
     private Set<ListeLecture> listes;

     public Morceau(String titre,String artiste,String nom_album) {
          ...
          if(...) { // si l'album existe
               this.album = a;
               a.add(this);
          } // sinon on le crée avec ce morceau
               this.album = new Album(artiste,nom_album,this);
          }
     }
}
```