

Héritage et Polymorphisme

Par Nicolas Hurtubise

Inspiré et dérivé des notes de Jian-Yun Nie et Pascal Vincent

IFT1025 - Programmation 2

Au programme...

- Héritage
- Existence et Accessibilité
- Types et casts d'objets
- Surcharge
- Classes `abstract` et `final`
- Interfaces

Objets redondants

Reprenons la classe Etudiant :

```
public class Etudiant {  
    private String prenom, nom, courriel;  
    private int matricule;  
  
    public Etudiant(String prenom, String nom, int matricule) { ... }  
  
    public String nomComplet() { ... }  
  
    public void etudier() { ... }  
  
    public String getPrenom() { ... }  
    // Autres getters/setters ...  
}
```

Objets redondants

Si on a également besoin de définir une classe Professeur :

```
public class Professeur {  
    private String prenom, nom, courriel;  
    private String domaineDeRecherche;  
  
    public Professeur(String prenom, String nom, String domaine) { ... }  
  
    public String nomComplet() { ... }  
  
    public void enseigner() { ... }  
    public void rechercher() { ... }  
  
    public String getPrenom() { ... }  
    // Autres getters/setters ...  
}
```

Objets redondants

On remarque...

- Nos classes Etudiant et Professeur ont des attributs communs et des méthodes communes...
- Mais ça n'aurait pas de sens de les regrouper dans une même classe

Objets redondants

Selon les besoins de notre programme, on pourrait avoir besoin d'être encore plus précis (Étudiant Bac/Gradué) :

```
public class EtudiantBac {  
    private int matricule;  
    private String prenom, nom, courriel;  
  
    public EtudiantBac(int matricule, String prenom, String nom) {...}  
  
    public String nomComplet() { ... }  
  
    public void etudier() { ... }  
  
    public String getPrenom() { ... }  
    // Autres getters/setters ...  
}
```

Objets redondants

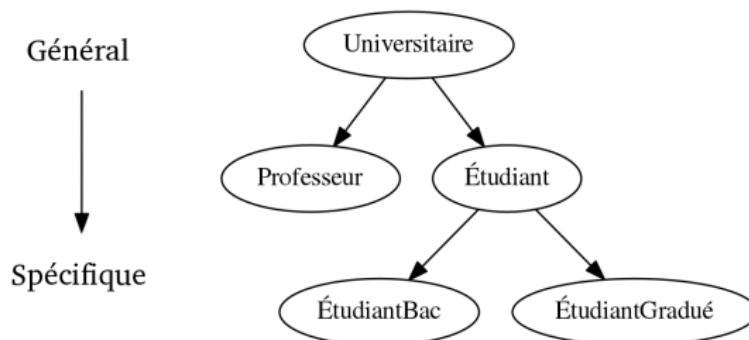
Selon les besoins de notre programme, on pourrait avoir besoin d'être encore plus précis (Étudiant Bac/Gradué) :

```
public class EtudiantGradue {  
    private int matricule;  
    private String prenom, nom, courriel;  
    private Professeur superviseur;  
    private String sujet;  
  
    public EtudiantGradue(int matricule, String prenom, String nom) ...  
  
    public String nomComplet() { ... }  
  
    public void etudier() { ... }  
    public void rechercher() { ... }  
  
    public String getPrenom() { ... }  
    // Autres getters/setters ...  
}
```

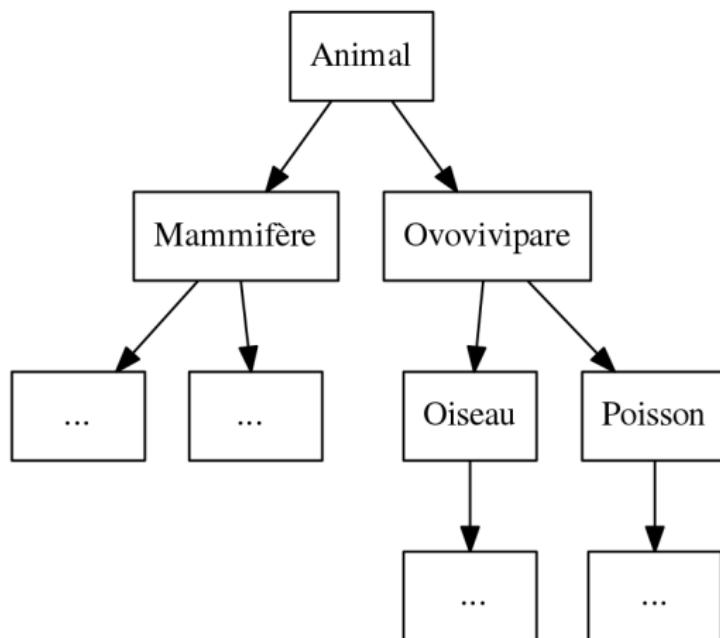
Objets redondants

Observation :

- Les *hiérarchies* entre les concepts sont communes dans le monde qu'on modélise



Objets redondants



Objets redondants

Peut-on trouver des hiérarchies ici ?

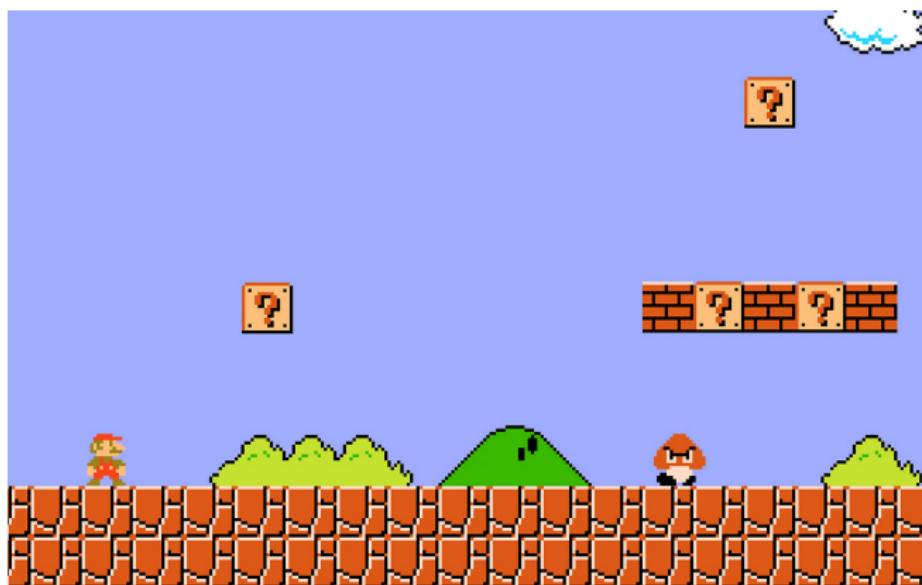


Figure 1:

<https://www.nintendo.fr/Jeu/NES/Super-Mario-Bros--803853.html>

Objets redondants

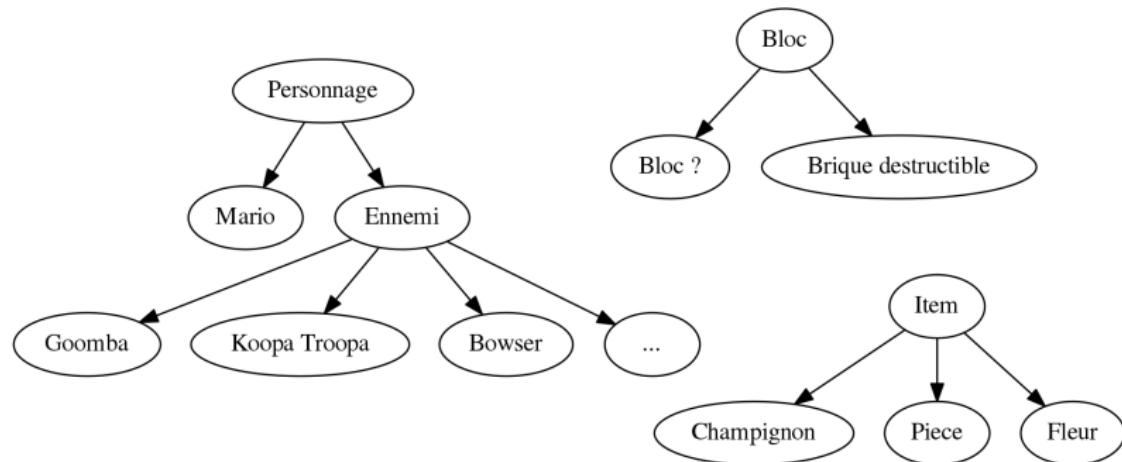


Figure 2: Hiérarchie possible entre les objets de Super Mario Bros

Objets redondants

- On veut éviter d'avoir à réécrire le même code partout...
 - On voudrait traduire la *hiérarchie naturelle entre les objets* dans la structure de notre code
-
- Mécanisme en programmation orientée objet : l'**héritage**
 - On peut écrire des classes *spécialisées* dérivées d'autres classes plus *génériques*

Héritage

- On déclare qu'une classe *hérite* d'une autre classe avec le mot-clé **extends**
- Une *sous-classe* (ou *classe fille*) hérite des méthodes et attributs de sa *super-classe* (ou *classe mère*)

```
public class Universitaire {  
    // Définition d'un universitaire simple  
    private String prenom, nom, courriel;  
  
    public String nomComplet() {  
        return prenom + " " + nom;  
    }  
}
```

Héritage

```
public class Professeur extends Universitaire {  
    // Contient tous les attributs d'un Universitaire  
  
    // + Attributs supplémentaires :  
    private String domaineDeRecherche;  
  
    // Contient toutes les méthodes d'un Universitaire  
  
    // + Méthodes supplémentaires :  
    public void enseigner() { ... }  
    public void rechercher() { ... }  
}
```

Héritage

```
public class Etudiant extends Universitaire {  
    // Contient tous les attributs d'un Universitaire  
  
    // + Attributs supplémentaires :  
    private int matricule;  
  
    // Contient toutes les méthodes d'un Universitaire  
  
    // + Méthodes supplémentaires :  
    public void etudier() { ... }  
}
```

Héritage

```
public class EtudiantGradue extends Etudiant {  
    /* Un étudiant gradué contient tous les attributs  
       d'un Etudiant et d'un Universitaire, plus d'autres : */  
  
    private Professeur superviseur;  
    private String sujet;  
  
    /* Contient toutes les méthodes d'un Etudiant  
       et d'un Universitaire, plus d'autres */  
    public void rechercher() { ... }  
}
```

Héritage

- On se sert de l'héritage pour définir des types de plus en plus spécifiques
- Un EtudiantGradue est une spécialisation d'un Etudiant, qui est une spécialisation d'un Universitaire

Universitaire
String prenom
String nom
String courriel
String nomComplet()

Étudiant
String prenom
String nom
String courriel
+int matricule
String nomComplet()
+void etudier()

ÉtudiantGradué
String prenom
String nom
String courriel
int matricule
+Professeur superviseur
+String sujet
String nomComplet()
void etudier()
+void rechercher()

Héritage

- Spécialisation : les objets de types enfants peuvent réaliser tout ce que le type parent peut faire + d'autres choses au besoin
- Toutes les méthodes définies dans une classe mère sont accessibles depuis les classes filles

Héritage

```
Universitaire universitaire = new Universitaire("Victor", "Hugo");
Etudiant etudiant = new Etudiant("Jimmy", "Whooper");
Professeur professeur = new Professeur("Johnny", "Tremblay");

universitaire.nomComplet() // => "Victor Hugo"
etudiant.nomComplet() // => "Jimmy Whooper"
professeur.nomComplet() // => Johnny Tremblay

professeur.rechercher() // OK
etudiant.etudier() // OK

// Invalide, méthode non définie pour un universitaire !
universitaire.rechercher()
```

Héritage : redéfinition de méthodes (Override)

- Spécialisation : les objets de types enfants peuvent *redéfinir leur implémentation des méthodes de la classe mère*

```
public class Professeur extends Universitaire {  
  
    @Override  
    public String nomComplet() {  
        return "Dr. " + prenom + " " + nom;  
    }  
  
    // ...  
}
```

Héritage : redéfinition de méthodes (Override)

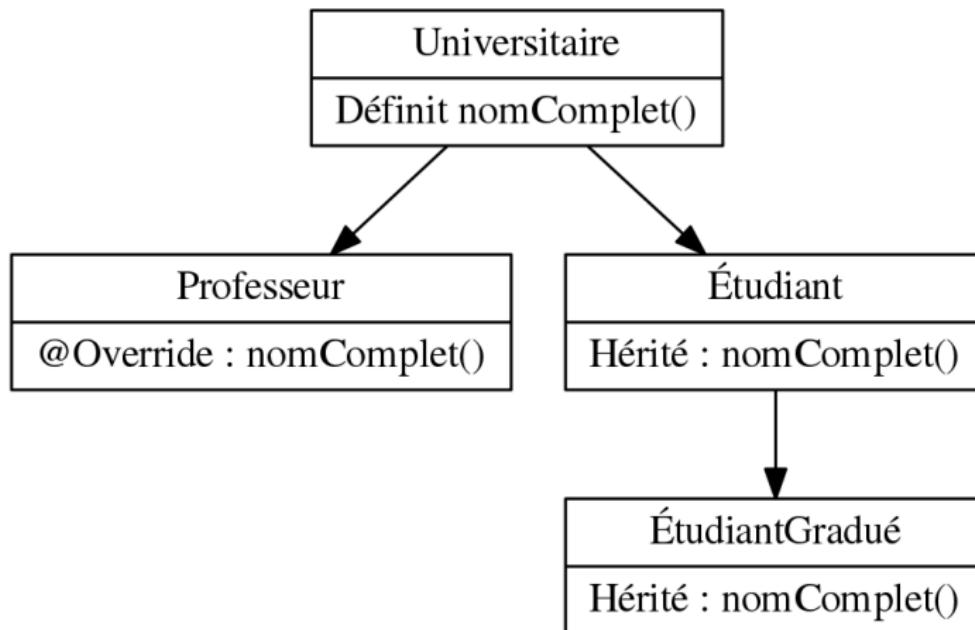
```
Universitaire universitaire = new Universitaire("Jimmy", "Whooper");
Etudiant etudiant = new Etudiant("Jimmy", "Whooper");
Professeur professeur = new Professeur("Jimmy", "Whooper");

System.out.println(universitaire.nomComplet());
// => Jimmy Whooper

System.out.println(etudiant.nomComplet());
// => Jimmy Whooper

System.out.println(professeur.nomComplet());
// => Dr. Jimmy Whooper
```

Héritage : redéfinition de méthodes (Override)



Exercice : Échecs

Est-ce qu'on peut appliquer ce concept ici ?

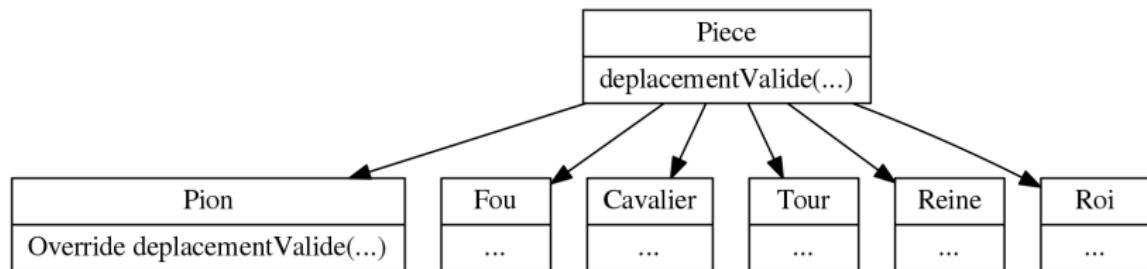


Figure 3:

<https://commons.wikimedia.org/wiki/File:ChessStartingPosition.jpg>

Exercice : Échecs

Chaque pièce a une façon différente de valider si un mouvement est possible elle selon son type :



Héritage : redéfinition de méthodes (Override)

Il y a quelques restrictions lors de la redéfinition de méthodes

La méthode redéfinie doit avoir :

- Le même nom
- Le même type de retour
- Les mêmes types de paramètres
- La même visibilité (public/private/...)

Par rapport à la méthode de la classe mère

Existence et Accessibilité

- Le *Principe d'encapsulation* demande que tous les attributs définis `private` ne soient utilisables/modifiables *que* dans le code source de la classe qui les a définis...

```
public class Parent {  
    public int a;  
    private int b;  
}  
  
public class Enfant extends Parent {  
  
    public void test() {  
        this.a = 10; // OK  
        // Invalide, l'attribut est private et  
        // n'est pas défini dans la classe Enfant !  
        this.b = 20;  
    }  
}
```

Existence et Accessibilité

- Comme n'importe quel utilisateur de la classe parent, on doit donc utiliser les getters/setters pour modifier les propriétés déclarées private

```
public class Parent {  
    public int a;  
    private int b;  
  
    // Accesseur  
    public int getB() {  
        return b;  
    }  
  
    // Mutateur  
    public void setB(int b) {  
        this.b = b;  
    }  
}
```

Existence et Accessibilité

- Comme n'importe quel utilisateur de la classe parent, on doit donc utiliser les getters/setters pour modifier les propriétés déclarées private

```
public class Enfant extends Parent {  
  
    public void test() {  
        // OK : a est héritée et est public  
        this.a = 10;  
  
        // INCORRECT : b est héritée mais est private  
        this.b = 20;  
  
        // OK : la méthode setB() est héritée et est public  
        this.setB(20); // OK  
    }  
}
```

Existence et Accessibilité

Pour `nomComplet()` dans notre classe `Professeur`, une définition valide serait plutôt :

```
public class Professeur extends Universitaire {  
  
    @Override  
    public String nomComplet() {  
  
        return "Dr. " + getPrenom() + " " + getNom();  
    }  
  
    // ...  
}
```

Existence et Accessibilité

Et si on voulait avoir accès la version originale de nomComplet ?

```
public class Professeur extends Universitaire {  
  
    @Override  
    public String nomComplet() {  
        // Utiliser Dr. + nomComplet() de Universitaire  
        return "Dr. " + /* ??? */;  
    }  
  
    // ...  
}
```

Mot-clé super

- Comme le mot-clé `this` qui permet de spécifier explicitement qu'on veut accéder à un attribut de l'instance actuelle, on peut accéder explicitement à la "version parent" d'un attribut ou d'une méthode en utilisant le mot-clé `super`

```
public class Professeur extends Universitaire {

    @Override
    public String nomComplet() {
        /* On redéfinis la méthode nomComplet() pour "Professeur",
           mais à l'intérieur de la définition de classe, on a
           encore accès à la version "Universitaire" si on
           utilise explicitement 'super' */

        return "Dr. " + super.nomComplet();
    }
    // ...
}
```

Mot-clé protected

- Le principe d'encapsulation n'est pas toujours pratique...
- Parfois, on aimerait gérer les attributs `private` de la super-classe directement dans la sous-classe, sans devoir les mettre `public` à tout le monde pour autant

```
public class Cours {  
    private String sigle, nom, description;  
    private Professeur prof;  
  
    public String getSigle() { ... }  
    // ...  
}  
  
public class Ift1025 extends Cours {  
    /* La classe cours est délibérément faite pour être étendue  
       par des sous-classes... Utiliser les getters() et  
       setters() partout va donner du code lourd à lire et écrire */  
}
```

Mot-clé protected

- Autre solution : déclarer l'accès `protected`
- Entre `public` et `private` :
 - Les classes du même package peuvent y accéder
 - Les *sous-classes* des autres packages peuvent y accéder
 - Les autres classes n'y ont simplement pas accès

```
public class Cours {  
    protected String sigle, nom, description;  
    protected Professeur prof;  
  
    public String getSigle() { ... }  
    // ...  
}
```

Mot-clé protected

Résumé

Accessibilité	public	private	protected	Rien (défaut) ¹
Dans la classe	oui	oui	oui	oui
Dans le package	oui	non	oui	oui
Sous-classes	oui	non	oui	non
Tout le monde	oui	non	non	non

<https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html>

¹Déclarer un attribut/une méthode sans spécifier l'accès signifie 'package private'

Héritage et Constructeurs

- Le *constructeur* d'une classe n'est jamais hérité
- On peut utiliser `super()` dans le constructeur de la sous-classe (avec des paramètres au besoin) pour appeler le constructeur parent
- L'appel à `super(...)` doit être *la toute première ligne du constructeur enfant*

Héritage et Constructeurs

```
public class Parent {
    public Parent() {
        System.out.println("Parent se fait construire");
    }
}

public class Enfant extends Parent {
    public Enfant() {
        super(); // Appelle le constructeur parent
        System.out.println("Enfant se fait construire");
    }
}
```

Héritage et Constructeurs

```
public static void main(String[] args) {
    System.out.println("-- Objet parent --");
    Parent p = new Parent();

    System.out.println("-- Objet enfant --");
    Enfant e = new Enfant();
}
```

```
-- Objet parent --
Parent se fait construire
-- Objet enfant --
Parent se fait construire
Enfant se fait construire
```

Héritage et Constructeurs

- Si un constructeur de la sous-classe n'appelle pas super(...) explicitement, super() (sans argument) est implicitement appelée comme première instruction

```
public class Parent {  
    public Parent() {  
        System.out.println("Parent se fait construire");  
    }  
}  
  
public class Enfant extends Parent {  
    public Enfant() {  
        System.out.println("Enfant se fait construire");  
    }  
}
```

Héritage et Constructeurs

Même résultat :

```
public static void main(String[] args) {
    System.out.println("-- Objet parent --");
    Parent p = new Parent();

    System.out.println("-- Objet enfant --");
    Enfant e = new Enfant();
}
```

```
-- Objet parent --
Parent se fait construire
-- Objet enfant --
Parent se fait construire
Enfant se fait construire
```

Héritage et Constructeurs

Implication :

- Si on n'appelle pas `super(...)` explicitement, le constructeur parent sans arguments doit être défini

```
public class Parent {  
    public Parent(String nom) {  
        System.out.println("Parent '" + nom + "' se fait construire");  
    }  
}  
  
/* Erreur !  
Impossible d'appeler le constructeur 'super()' automatiquement ! */  
public class Enfant extends Parent {  
    public Enfant() {  
        System.out.println("Enfant se fait construire");  
    }  
}
```

Héritage et Constructeurs

Implication :

- Si on n'appelle pas `super(...)` explicitement, le constructeur parent sans arguments doit être défini

```
public class Parent {  
    public Parent(String nom) {  
        System.out.println("Parent '" + nom + "' se fait construire");  
    }  
}  
  
public class Enfant extends Parent {  
    public Enfant() {  
        super("Bob"); // Ok, on appelle explicitement  
        System.out.println("Enfant se fait construire");  
    }  
}
```

Regroupier des types différents ensemble

- On veut écrire une newsletter qui s'adresse aux Étudiants autant qu'aux Professeurs

```
// Tableau pour les étudiants abonnés
Etudiant[] etudiantsAbonnes = new Etudiant[nbrEtudiantsAbonnes];

// Tableau pour les professeurs abonnés
Professeur[] profsAbonnes = new Professeur[nbrProfsAbonnes];

// Initialisation des new Etudiant()
// ...

// Initialisation des new Professeur()
// ...
```

Regroupier des types différents ensemble

```
// On se crée une instance de la Newsletter du mois
Newsletter lettre = new Newsletter(...);

String[] courriels = new String[nbrEtudiantsAbonnes + nbrProfAbonnes];

// On ajoute les courriels des étudiants
for(int i=0; i<etudiantsAbonnes.length; i++) {
    courriels[i] = etudiantsAbonnes[i].getCourriel();
}

// On ajoute les courriels des profs
for(int i=0; i<profAbonnes.length; i++) {
    courriels[nbrEtudiantsAbonnes + i] = profAbonnes[i].getCourriel();
}

lettre.envoyer(courriels);
```

Regroupier des types différents ensemble

- On fait exactement le même traitement sur les Professeurs que sur les Etudiants...
- Ce qui nous intéresse réellement, c'est le `.getCourriel()`, qui est défini pour la classe Universitaire et hérité dans toutes les sous-classes
- Un Etudiant et un Professeur sont tous deux dérivés du type Universitaire

Regroupier des types différents ensemble

Solution : on peut créer un tableau d'Universitaires qui contient des Etudiants et des Professeurs

```
Universitaire[] abonnes = new Universitaire[nbrAbonnes];  
  
// Un Etudiant est un Universitaire, mais plus spécifique  
abonnes[0] = new Etudiant("Jimmy", "Whooper");  
  
// Un Professeur est un Universitaire, mais plus spécifique  
abonnes[1] = new Professeur("Jean-Philippe", "Whooper");  
  
// ... Autres abonnés ici ...
```

Regroupier des types différents ensemble

On peut alors envoyer notre Newsletter à tous les Universitaires de la même façon :

```
Newsletter lettre = new Newsletter(...);

String[] courriels = new String[abonnes.length];

// On ajoute les courriels des universitaires abonnés (prof/étudiants)
for(int i=0; i<abonnes.length; i++) {
    courriels[i] = abonnes[i].getCourriel();
}

lettre.envoyer(courriels);
```

Regroupier différents types ensemble

Flashback du premier chapitre :

Tableaux

- En Java, les tableaux, comme les autres variables, ne peuvent contenir qu'un seul type de donnée.
- Si on veut un tableau contenant des entiers, on doit déclarer la variable comme étant de type "tableau de int"

Figure 4: Mensonge !

Regroupier différents types ensemble

Est-ce que ça a changé depuis ?

- Non !
- On utilise effectivement qu'un seul type de donnée ici...
Universitaire

```
Universitaire universitaire = new Etudiant("Jimmy", "Whooper");
```

- Un Etudiant compte comme un Universitaire

Quelle méthode est appelée ?

```
Universitaire professeur = new Professeur("Jimmy", "Whooper");  
professeur.nomComplet()
```

- Quelle version de la méthode devrait être appelée ?

```
// Version définie dans Universitaire  
public String nomComplet() {  
    return prenom + " " + nom;  
}
```

```
// Version redéfinie dans Professeur  
@Override  
public String nomComplet() {  
    return "Dr." + super.nomComplet();  
}
```

Quelle méthode est appelée ?

Règle :

- Le *type de la variable* qui réfère à l'objet définit quelles méthodes sont **disponibles**
- Vérification à la *compilation*

```
Universitaire u = ...;
```

```
/* Peu importe le type de l'objet dans u, on peut seulement appeler  
les méthodes disponibles dans la classe Universitaire */
```

Quelle méthode est appelée ?

Règle :

- Le *type de la variable* qui réfère à l'objet définit quelles méthodes sont **disponibles**
- Vérification à la *compilation*

```
Universitaire u = ...;
```

/ Peu importe le type de l'objet dans u, on peut seulement appeler les méthodes disponibles dans la classe Universitaire */*

- Le *type de l'instance* définit quelle *version de la méthode* est **appelée**
- Vérification à *l'exécution*

```
... e = new Etudiant();
```

/ Les méthodes appelées sur e sont celles du type Etudiant, peu importe le type de la référence */*

Quelle méthode est appelée ?

Exemple :

```
Universitaire x = new Professeur("Jimmy", "Whooper");  
  
// OK, car "x" est de type Universitaire  
x.nomComplet()  
/* => Affiche "Dr. Jimmy Whooper", car l'instance  
que référence x est de type Professeur */  
  
// INCORRECT : un Universitaire ne définit pas de méthode rechercher  
x.rechercher();
```

Quelle méthode est appelée ?

Pour utiliser les méthodes d'une classe donnée, la référence doit permettre de le faire

```
Universitaire x = new Professeur("Jimmy", "Whooper");
```

```
// Erreur de compilation  
x.rechercher();
```

On sait en lisant le code que x fait référence à un objet de type Professeur et devrait donc être capable de rechercher(), mais le compilateur ne le sait pas

Cast d'objets

Solution : on peut (caster) pour forcer une référence à être considérée comme celle d'un type précis

```
Universitaire x = new Professeur("Jimmy", "Whooper");
```

```
Professeur jimmy = (Professeur) x;
```

/ On prend une copie de la référence sur l'objet
en utilisant un type plus spécifique*

*Notez: x et jimmy réfèrent à la même instance */*

```
jimmy.rechercher(); // Ok !
```

On dit au compilateur :

Ne t'inquiète pas, je sais ce que je fais

Je suis 100% sûr que x est un Professeur valide

Cast d'objets

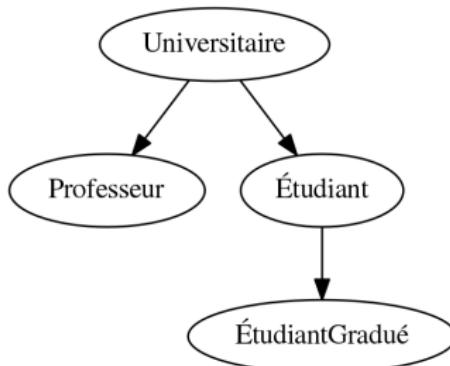
On peut caster dans les deux sens de la hiérarchie de classes :

- *Upcast* : caster la référence vers un type plus générique
- *Downcast* : caster la référence vers un type plus spécifique

Cast d'objets

```
Professeur professeur = new Professeur("Jean-Philippe", "Whooper");
Universitaire etudiantGradue = new EtudiantGradue("Xavier", "Whooper");
```

```
Universitaire u = (Universitaire) professeur; // Upcast
EtudiantGradue e = (EtudiantGradue) etudiantGradue; // Downcast
```



Upcast

- Une classe est automatiquement castée vers un type plus générique lorsque nécessaire
- Doit être une classe parent dans la hiérarchie

```
// Invalide, pas dans la même branche de la hiérarchie !
Professeur p1 = new Etudiant("Jimmy", "Whooper");

Universitaire u = new Etudiant("Xavier", "Whooper"); // OK

/* Également invalide, u contient une référence vers une instance
   qui n'est pas dans la branche de Professeur */
Professeur p2 = (Professeur) u;
```

Upcast

- Une classe est automatiquement castée vers un type plus générique lorsque nécessaire

```
public static void main(String[] args) {
    EtudiantGradue jimmy = new EtudiantGradue("Jimmy", "Whooper");

    sendEmail(jimmy); // Cast vers (Universitaire) automatique

}

public static void sendEmail(Universitaire u) {
    // ...
}
```

Upcast

- Une classe est automatiquement castée vers un type plus générique lorsque nécessaire

```
public static void main(String[] args) {
    Universitaire u = bestUniversitaireEver();

    System.out.println(u.nomComplet());
    // => Dr. Jimmy Whooper
}

public static Universitaire bestUniversitaireEver() {
    return new Professeur("Jimmy", "Whooper");
}
```

Downcast

- Si on veut utiliser les méthodes/attributs d'une classe plus spécifique que ce que la référence indique, on doit (caster) manuellement

```
Universitaire jimmy = new Etudiant("Jimmy", "Whooper");  
  
jimmy.etudier() // Invalide, erreur de compilation !  
  
Etudiant jimmyEtudiant = (Etudiant) jimmy;  
  
jimmyEtudiant.etudier(); // Ok
```

Downcast

- Doit absolument être un cast valide (doit correspondre à la vraie classe de l'instance ou à une classe parmi ses ancêtres)

```
Universitaire jimmy = new Universitaire("Jimmy", "Whooper");
Professeur prof = new Professeur("Jean-Philippe", "Whooper");
EtudiantGradue xavier = new EtudiantGradue("Xavier", "Whooper");

// Ok
Etudiant xavierEtudiant = (Etudiant) xavier;

// Invalide !
// Un 'Etudiant' est un 'Universitaire', mais l'inverse ne marche pas
Etudiant jimmyEtudiant = (Etudiant) jimmy;

// Invalide ! Pas dans la bonne branche
// Un 'Professeur' n'est pas un étudiant
Etudiant jeanPhilippe = (Etudiant) prof;
```

Cast d'objets

- Un (cast) ne modifie pas l'objet
- Permet seulement de regarder l'objet *comme s'il était une instance d'un autre type compatible*

```
EtudiantGradue xavier = new EtudiantGradue("Xavier", "Whooper");

/* La mémoire contient encore un EtudiantGradue
   mais on le regarde comme si c'était un Etudiant */
Etudiant xavierEtudiant = (Etudiant) xavier;
```

Question

- Quel est le type de la référence ?
- Quel est le type de l'instance ?

```
Universitaire u = new EtudiantGradue("Jimmy", "Whooper");
```

Réponse (rot13) : eéséerapr = Havirefvgnver, vafgnapr = RghqvnagTenqhr

Retrouver le type d'une instance : instanceof

- Puisqu'une *référence* d'un certain type peut contenir une *instance* d'un sous-type, on peut être en droit de se demander à quoi on a affaire exactement quand on manipule une référence...

```
Universitaire jimmy = ...
```

```
/* jimmy est un Universitaire, un Professeur,  
un Etudiant, un EtudiantGradue... ? */
```

Retrouver le type d'une instance : instanceof

- Le mot-clé `instanceof` permet de tester si une instance est d'une certaine classe :

```
Universitaire jimmy = ...
```

```
System.out.println(jimmy instanceof Etudiant);  
// => Affiche true ou false, selon le cas
```

Retrouver le type d'une instance : instanceof

Une instance d'un type est également considérée instance de tous ses super-types :

```
Universitaire universitaire = new Universitaire("Victor", "Hugo");
Professeur professeur = new Professeur("Jean-Philippe", "Whooper");
Etudiant etudiant = new Etudiant("Jimmy", "Whooper");
EtudiantGradue etudiantGradue = new EtudiantGradue("Xavier", "Whooper")

universitaire instanceof Universitaire    // => true
etudiant instanceof Universitaire          // => true

etudiantGradue instanceof EtudiantGradue // => true
etudiantGradue instanceof Etudiant       // => true
etudiantGradue instanceof Universitaire // => true

universitaire instanceof Etudiant         // => false
professeur instanceof Etudiant            // => false
```

Retrouver le type d'une instance : instanceof

On peut s'en servir pour valider le type avant de faire un downcast

```
Universitaire[] universitaires = new Universitaire[5];  
  
// ... initialisation ...  
  
for(int i=0; i<5; i++) {  
  
    // Si l'Universitaire est un étudiant, on le fait étudier  
    if(universitaires[i] instanceof Etudiant) {  
  
        ((Etudiant) universitaires[i]).etudier();  
  
    }  
}
```

Retrouver le type d'une instance : instanceof

Attention !

- Même si instanceof est pratique de temps en temps, préférer la *redéfinition* de méthodes pour spécialiser les traitements, plutôt que l'appel conditionnel avec instanceof + (caster)

Retrouver le type d'une instance : instanceof

Mauvais usage :

```
public class Chat extends Animal {  
    public void miauler() {  
        System.out.println("Miaou !");  
    }  
}  
  
public class Chien extends Animal {  
    public void aboyer() {  
        System.out.println("Arf !");  
    }  
}
```

Retrouver le type d'une instance : instanceof

Mauvais usage :

```
Animal animaux = new Animal[...];  
  
// ...  
  
for(int i=0; i<animaux.length; i++) {  
  
    if(animaux[i] instanceof Chat) {  
        ((Chat) animaux[i]).miauler();  
    } else if(animaux[i] instanceof Chien) {  
        ((Chien) animaux[i]).abooyer();  
    }  
}
```

Retrouver le type d'une instance : instanceof

Meilleure version :

```
public class Animal {  
    public void crier() { ... }  
}  
  
public class Chat extends Animal {  
    @Override  
    public void crier() {  
        System.out.println("Miaou !");  
    }  
}  
  
public class Chien extends Animal {  
    @Override  
    public void crier() {  
        System.out.println("Arf !");  
    }  
}
```

Retrouver le type d'une instance : instanceof

Meilleure version :

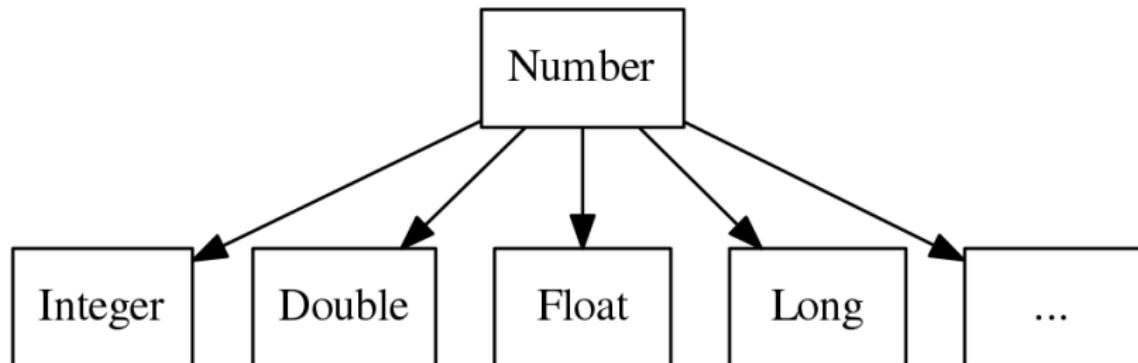
```
Animal animaux = new Animal[...];  
// ...  
  
for(int i=0; i<animaux.length; i++) {  
  
    /* Beaucoup plus gracieux, la bonne  
       méthode est appelée automatiquement */  
    animaux[i].crier();  
  
}
```

Regroupier des types primitifs ensemble

- Et est-ce qu'on pourrait avoir un tableau qui contient des `int` et des `double` ?
 - A priori => Non
 - `int` et `double` sont des primitives (ce ne sont pas des objets)
-
- On peut cependant utiliser l'orienté objet et l'*héritage* pour y arriver

Regroupier des types primitifs ensemble

- Java met à notre disposition des *Classes Wrapper* pour les cas où on voudrait profiter des mécanismes de l'orienté objet sur des types primitifs :
 - Integer, Double, Float, Long, Byte, Character, ...
- Chaque classe ne contient qu'un seul attribut, une valeur d'un certain type
- Toutes ces classes héritent de la classe Number



Regroupier des types primitifs ensemble

Tous les types primitifs ont une classe wrapper correspondante

Primitive type	Wrapper class
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

```
Integer i = new Integer(10);
```

```
Float f = new Float(10.5f);
```

Regroupier des types primitifs ensemble

- Autoboxing : les valeurs primitives sont automatiquement converties en leur équivalent Objet pour simplifier l'utilisation
- Valeur Autoboxée : convertie avec `new Integer(...)`, `new Float(...)`, ... automatiquement

```
Integer i = 10; // Valide

Float f = 10.5f; // Valide aussi

Number[] valeurs = new Number[3];

// Valide
valeurs[0] = 10;
valeurs[1] = 55.5f;
valeurs[2] = 1.3e100;
```

Surcharge (Overload)

En Java, on peut définir plusieurs fonctions qui portent le même nom dans la même classe :

```
public class Vecteur {  
    // Attributs ...  
    public void add(Vecteur v) {  
        // Fait l'addition des éléments terme à terme  
        // index 0 = this.elements[0] + v.elements[0]  
        // index 1 = ...  
    }  
  
    public void add(int d) {  
        // Additionne d à tous les éléments du vecteur  
        // index 0 = this.elements[0] + d  
        // index 1 = ...  
    }  
}
```

Pour autant que les *signatures* soient différentes

Surcharge (Overload)

Rappel :

- La **signature d'une fonction** (ou méthode) est définie comme étant son nom + le type de ses arguments

Java

```
public int carre(int x) {  
    return x * x;  
}
```

Signature

```
carre(int)
```

Surcharge (Overload)

- Java décide quelle version de la fonction appeler, de façon à choisir la plus *spécifique* selon le type de l'argument passé

```
public class Vecteur {  
    public void add(Vecteur v) { ... }  
    public void add(int d) { ... }  
}
```

```
Vecteur v = new Vecteur(new int[]{1, 2, 3});  
Vecteur w = new Vecteur(new int[]{10, 20, 30});
```

v // => Contient {1, 2, 3}

v.add(100); // Ajoute un int
// => v contient {101, 202, 303}

v.add(w); // Ajoute un Vecteur
// => v contient {111, 122, 133}

Surcharge (Overload)

Un bon exemple de fonction *surchargée* est la fonction
System.out.println :

```
System.out.println(10);
System.out.println(10.0);
System.out.println("10");
System.out.println(10.0f);
```

Multiples définitions de la même fonction, la bonne version est
appelée selon le type de l'argument qui est passé

Surcharge (Overload)

Question : est-ce qu'on pourrait faire quelque chose du genre ?

```
public class Question {  
  
    private int i;  
    public Question() { ... }  
    public void incrementer() {  
        i++;  
    }  
  
    public String getI() {  
        return "" + i;  
    }  
  
    public int getI() {  
        return i;  
    }  
}
```

Surcharge (Overload)

Quelle version de la fonction serait appelée ici ?

```
public class Question {  
    public String getI() { ... }  
    public int getI() { ... }  
}
```

```
Question q = new Question();
```

```
String valeurStr = q.getI();
```

```
int valeurInt = q.getI();
```

Surcharge (Overload)

Et ici ?

```
Question q = new Question();  
System.out.println(q.getI());
```

Surcharge (Overload)

- Impossible de trouver quelle fonction est appelée dans certains cas...
- La **signature** (nom de fonction + type des paramètres) de chaque méthode doit être unique, excluant le type de retour

```
// OK
public class Question {

    // ...

    public String getStringI() {
        return "" + i;
    }

    public int getIntI() {
        return i;
    }
}
```

Autre Question

- Qu'est-ce qui se passe si on surcharge en utilisant des types de la même hiérarchie ?

```
public class Question {  
  
    public void saluer(Universitaire u) {  
        System.out.println("Bonjour, universitaire !");  
    }  
  
    public void saluer(Etudiant e) {  
        System.out.println("Bonjour, étudiant !");  
    }  
  
    public void saluer(EtudiantGradue eg) {  
        System.out.println("Bonjour, étudiant gradué !");  
    }  
}
```

Autre Question

```
Question q = new Question();
Universitaire u = new Etudiant("Jimmy", "Whooper");

q.saluer(u);
// => Affiche ?
```

Autre Question

Règle :

- Java décide quelle version de la fonction appeler, de façon à choisir la plus *spécifique* pour le *type de l'argument passé* (le **type de la référence**, pas le type de l'instance)

```
Question q = new Question();
Universitaire u = new Etudiant("Jimmy", "Whooper");

q.saluer(u);
// => Affiche Bonjour, universitaire !
```

Autre Question

- La méthode est appelée suivant le type de la variable passée en paramètre
- Si on veut appeler la fonction qui reçoit en paramètre un Etudiant, on doit donc (caster) la référence

```
Question q = new Question();
Universitaire u = new Etudiant("Jimmy", "Whooper");

q.saluer((Etudiant) u);
// => Affiche Bonjour, étudiant !
```

Surcharge d'opérateurs ?

On peut surcharger des fonctions...

- Est-ce qu'on peut faire la même chose avec les opérateurs ?

En Java, l'opérateur + est *surchargé* :

```
1 + 1 // => 2  
1 + "1" // => "11"
```

Selon le type de ses opérandes, on fait soit une addition mathématique, soit une concaténation de Strings

Surcharge d'opérateurs ?

- Est-ce qu'on peut redéfinir le comportement des opérateurs sur nos objets nous-mêmes ?

```
Point3D p1 = new Point3D(1, 2, 3);  
Point3D p2 = new Point3D(10, 20, 30);
```

p1 + p2 // => donnerait un Point3D(11, 22, 33)

Surcharge d'opérateurs ?

- Pas en Java
- La surcharge du + est built-in dans le langage, on ne peut pas l'étendre à d'autres types
- D'autres langages permettent de le faire (ex.: Python, C++)

Surcharge d'opérateurs ?

```
#!/usr/bin/python3
import numpy as np

matrice1 = np.array([[1,2,3],
                     [4,5,6],
                     [7,8,9]])

matrice2 = np.array([[10,20,30],
                     [40,50,60],
                     [70,80,90]])

# Surcharge du '+' : addition définie pour des matrices
print(matrice1 + matrice2)

# Affiche :
#
# [[11 22 33]
#  [44 55 66]
#  [77 88 99]]
```

Surcharge d'opérateurs ?

- Java a été conçu avec la philosophie de ne pas permettre de faire des choses qui pourraient être trop dangereuses...
- Facile pour quelqu'un d'inexpérimenté d'abuser de la surcharge d'opérateurs et créer des objets dont le comportement est imprévisible
 - Multiplier deux dates ?
 - Additionner deux Etudiant ?
 - Le code pourrait vite devenir incompréhensible si on utilisait mal cette fonctionnalité

Surcharge de constructeurs

À noter qu'on peut surcharger le constructeur d'une classe au besoin :

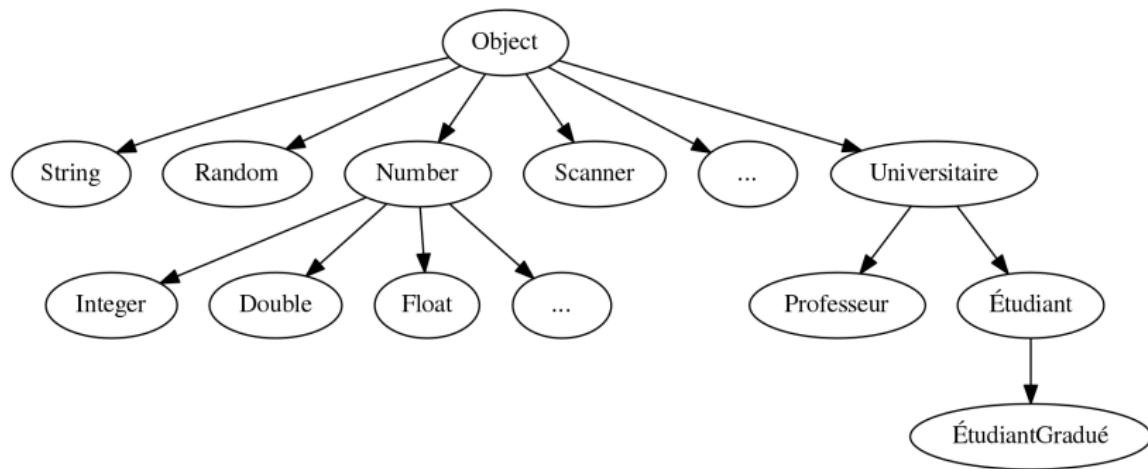
```
public class Universitaire {  
    // ...  
  
    public Universitaire(String prenom, String nom, String courriel) {  
        this.prenom = prenom;  
        this.nom = nom;  
        this.courriel = courriel;  
    }  
  
    public Universitaire(String prenom, String nom) {  
        this.prenom = prenom;  
        this.nom = nom;  
        this.courriel = prenom + "." + nom + "@umontreal.ca";  
    }  
}
```

Hiérarchie *ultime* de classes en Java

- Une classe définie sans extends est automatiquement une classe fille de la classe Object
- **Toutes** les classes héritent de la classe Object, directement ou indirectement

```
public class Universitaire { // Implicite : extends Object  
}
```

Hiérarchie *ultime* de classes en Java



Hiérarchie *ultime* de classes en Java

La classe Object définit quelques méthodes, entre autres :

- `String toString()`
- `boolean equals(Object otherObject)`
- `int hashCode()`
- ...

Tout objet possède ces méthodes, minimalement l'implémentation par défaut

- `toString` retourne une représentation textuelle de l'objet

toString()

- C'est assez commun de redéfinir `toString()` dans une classe :

```
public class Universitaire {  
    // ...  
  
    @Override  
    public String toString() {  
        return "[Universitaire : " + nomComplet() + "]";  
    }  
}
```

toString()

- C'est assez commun de redéfinir `toString()` dans une classe :

```
Universitaire universitaire = new Universitaire("Jimmy", "Whooper");  
  
System.out.println(universitaire);  
// => Affiche [Universitaire : Jimmy Whooper]  
//     plutôt que la référence Universitaire@7852e922
```

- La fonction `System.out.println(Object o);` va afficher le résultat de `o.toString()`
- Pratique lors du débogage

Classes abstraites et mot-clé abstract

- Parfois, une classe parente n'a pas de sens en elle-même
 - Pièce d'échec sans sous-type ?
 - La classe `Number`, parente de `Integer`, `Double`, ...
- On peut vouloir interdire l'instanciation de certains types d'objets s'ils n'ont un sens que dans le contexte où ils sont sous-classés

Classes abstraites et mot-clé abstract

Exemple : une Forme possède une aire

```
public class Forme {  
    public double aire() {  
        return 0; // ??  
    }  
}
```

Classes abstraites et mot-clé abstract

Exemple : une Forme possède une aire

```
public class Cercle extends Forme {
    private double rayon;

    @Override
    public double aire() {
        return Math.PI * rayon * rayon;
    }
}

public class Rectangle extends Forme {
    private double largeur, hauteur;

    @Override
    public double aire() {
        return largeur * hauteur;
    }
}
```

Classes abstraites et mot-clé abstract

Une Forme en elle-même n'a pas de sens, seules les sous-classes devraient être instanciables :

```
Forme forme = new Forme();  
  
System.out.println(forme.aire()); // Pas logique...
```

Classes abstraites et mot-clé abstract

Solution : définir la classe comme étant *abstraite* :

- La classe peut avoir des méthodes normales
- Elle peut également avoir des méthodes abstraites
 - Pas de code, seulement une signature et un type de retour à respecter

```
public abstract class Forme {  
    private String couleur;  
  
    public Forme() { ... }  
  
    public String getColor() {  
        return couleur;  
    }  
  
    // Méthode abstraite : pas de code  
    public abstract double aire();  
}
```

Classes abstraites et mot-clé abstract

- On ne peut pas instancier directement une classe abstraite
 - Il manque des définitions de méthodes
- On peut cependant utiliser le type abstrait pour une référence

```
Forme forme = new Forme(); // INCORRECT
```

```
Cercle cercle = new Cercle(); // OK
```

```
Forme triangle = new Triangle(); // OK
```

Classes abstraites et mot-clé abstract

- On ne peut pas instancier directement une classe abstraite
 - Il manque des définitions de méthodes
- On peut cependant utiliser le type abstrait pour une référence

```
// OK, crée un tableau de 3 références vers des formes
```

```
Forme[] formes = new Forme[3];
```

```
// OK
```

```
formes[0] = new Triangle();
```

```
formes[1] = new Cercle();
```

```
formes[2] = new Rectangle();
```

```
// Aire d'un Triangle
```

```
System.out.println(formes[0].aire());
```

```
// Aire d'un Cercle
```

```
System.out.println(formes[1].aire());
```

Classes abstraites et mot-clé abstract

- Seules les `abstract class` peuvent contenir des méthodes `abstract`
- Une classe fille d'une classe abstraite a deux possibilités :
 - 1 Implémenter toutes les méthodes abstraites définies par la super-classe
 - 2 Être elle-même abstraite

Classes abstraites et mot-clé abstract

Les classes enfants *concrètes* (qui ne sont pas `abstract`) doivent absolument redéfinir toutes les *méthodes abstraites* définies par la classe mère

```
public class Cercle extends Forme {  
    @Override  
    public double aire() {  
        return Math.PI * rayon * rayon;  
    }  
}
```

Classes abstraites et mot-clé abstract

- On peut également avoir une classe *abstraite* qui hérite d'une classe abstraite
- Pas obligatoire d'implémenter les méthodes définies *abstract* dans le parent

```
public abstract class Quadrilatere extends Forme {  
    // ... pas obligatoire (mais possible) d'@Override aire()  
}  
  
public class Rectangle extends Quadrilatere {  
    @Override  
    public double aire() {  
        return Math.PI * rayon * rayon;  
    }  
}
```

Classe final

- Interdire le fait d'extend une classe

```
public final class HeritageImpossible {  
}  
  
// Cause une erreur de compilation !  
public class Erreur extends HeritageImpossible {  
}
```

Classe final

- Pourquoi déclarer une classe final ?
 - 1 Quand ça ne fait pas de sens de pouvoir en hériter
 - 2 Plus généralement... Dès qu'elles ne sont pas *conçues avec la possibilité d'héritage en tête*

Classe final

Exemple : on a une classe Vector qui permet de manipuler un vecteur et de lui faire des additions :

```
public static void main(String[] args) {
    Vector x = new Vector(3);
    System.out.println(x);
    // => (0.0, 0.0, 0.0)

    x.add(0, 5.5);
    System.out.println(x);
    // => (5.5, 0.0, 0.0)

    x.add(1, 10);
    System.out.println(x);
    // => (5.5, 10.0, 0.0)

    x.addToIntAll(100);
    System.out.println(x);
    // => (105.5, 110.0, 100.0)
}
```

Classe final

On pourrait s'écrire une sous-classe CountVector qui garde le compte du nombre d'additions effectuées :

```
public class CountVector extends Vector {  
  
    private int nbEdits;  
  
    public CountVector(int n) {  
        super(n);  
        nbEdits = 0;  
    }  
  
    public int getNbEdits() {  
        return nbEdits;  
    }  
  
    // ...
```

Classe final

On pourrait s'écrire une sous-classe CountVector qui garde le compte du nombre d'additions effectuées :

```
// ...

@Override
public void add(int i, double x) {
    this.nbEdits++; // +1 chaque fois qu'on fait une addition
    super.add(i, x);
}

@Override
public void addToAll(double x) {
    super.addToAll(x);

    // +N si on incrémenté tous les éléments du vecteur
    this.nbEdits += this.size();
}

}
```

Classe final

Si on teste notre classe CountVector...

```
public static void main(String[] args) {
    CountVector x = new CountVector(3);
    // Vecteur de taille 3

    x.add(0, 5.5);
    System.out.println(x.getNbEdits());
    // => 1

    x.add(1, 10);
    System.out.println(x.getNbEdits());
    // => 2

    x.addToIntAll(100);
    System.out.println(x.getNbEdits());
    // => 8... 1 + 1 + 3 == 8 ?
}
```

Classe final

Notre classe CountVector est erronée... Si on regarde le code de la classe Vector, on peut découvrir pourquoi :

```
public class Vector {  
    // ...  
  
    public void add(int i, double x) {  
        this.values[i] += x;  
    }  
  
    public void addToAll(double x) {  
        for(int i=0; i < this.values.length; i++) {  
            /* Utilise la méthode add(), qui a été  
               redéfinie dans CountVector ! */  
            this.add(i, x);  
        }  
    }  
}
```

Classe final

Puisqu'à l'interne, `addAll` utilise la méthode `add`, c'est une erreur d'ajouter le nombre d'éléments en plus :

```
this.nbEdits += this.size();
```

Le code dans notre sous-classe dépend des *mécanismes internes* de la classe parente pour fonctionner correctement

Classe final

Puisqu'à l'interne, `addAll` utilise la méthode `add`, c'est une erreur d'ajouter le nombre d'éléments en plus :

```
this.nbEdits += this.size();
```

Le code dans notre sous-classe dépend des *mécanismes internes* de la classe parente pour fonctionner correctement

- **Encapsulation** : un objet est une *boîte noire*
 - On ne veut pas avoir à connaître son implantation pour l'utiliser
 - Les détails d'implantation à l'interne doivent pouvoir changer sans que ça affecte son interface publique
- Le principe d'encapsulation peut être brisé par l'héritage si on ne fait pas attention

Classe final

Conclusion : déclarer les classes `final` lorsqu'elles ne sont pas conçues pour l'héritage peut éviter des problèmes plus tard

Ça ne nous limite pas pour le futur ?

- Si on se rend compte plus tard qu'on a besoin de pouvoir hériter de la classe, rien ne nous empêche de simplement retirer le mot-clé `final`
- Pas besoin de réécrire le code existant

Question

Dans quel ordre devrait-on écrire les mots-clés ?

```
public abstract final class Question {  
    // ...  
}
```

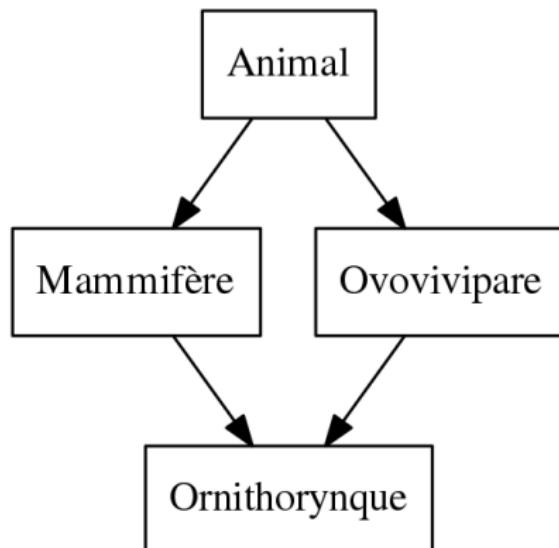
Ou plutôt...

```
public final abstract class Question {  
    // ...  
}
```

Réponse (rot13) : p'rfg har dhrgvba cvètr... nofgenpg + svany a'n cnf qr fraf

Héritage simple vs Héritage multiple

- Est-ce qu'on pourrait hériter de plus qu'une seule classe ?

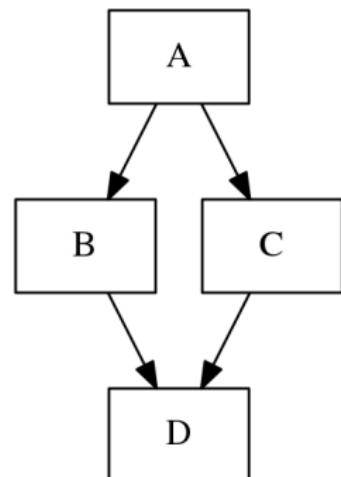


Héritage simple vs Héritage multiple

C'est un problème qu'on appelle le **Deadly Diamond of Death**

Classe "grand-mère" A en double... Comment on gère ça ?

- Attributs de la classe A dupliqués ?
- Si une méthode est redéfinie à la fois par B et par C (mais pas par D), quelle version utiliser ?

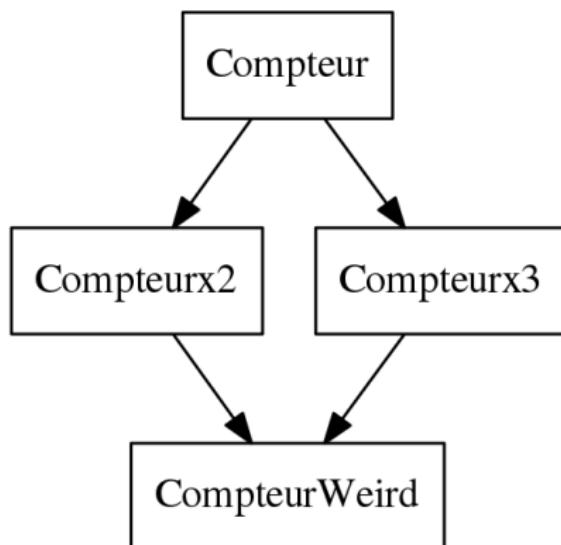


Héritage simple vs Héritage multiple

```
public class Compteur {  
    protected int i;  
    public Compteur() { i = 0; }  
    public void ajouter() { i++; }  
}  
  
public class Compteux2 extends Compteur {  
    @Override  
    public void ajouter() { i += 2; }  
}  
  
public class Compteux3 extends Compteur {  
    @Override  
    public void ajouter() { i += 3; }  
}
```

Héritage simple vs Héritage multiple

```
public class CompteurWeird extends Compteurx2, Compteurx3 {  
    ?  
}
```



Héritage simple vs Héritage multiple

- Pose plusieurs problèmes
- Pas possible en Java², une classe extends une seule autre classe
- Certains langages (ex.: C++) le permettent, mais c'est généralement vu comme une mauvaise pratique
 - Généralement une meilleure idée de préférer la composition

²Partiellement possible depuis Java 8, mais ça reste une mauvaise pratique

Héritage simple vs Héritage multiple

Une solution simple au problème d'héritage multiple (souvent la bonne) est de préférer la composition :

```
public class CompteurWeird {  
    private Compteurx2 compteur2x;  
    private Compteurx3 compteurx3;  
  
    // ...  
}
```

Héritage simple vs Héritage multiple

- Un design qui peut sembler avoir besoin d'héritage multiple n'en a souvent pas réellement besoin.
- Par exemple :
 - Une voiture n'hérite pas d'un moteur et de roues...
 - Une voiture est *composée* d'un moteur et de roues

Héritage simple vs Héritage multiple

```
public class Voiture extends Moteur, Roues {  
    ...  
}  
  
// Préférable :  
public class Voiture {  
  
    private Moteur moteur;  
    private Roues[] roues;  
  
    // ...  
}
```

Héritage simple vs Héritage multiple

- Mais pourtant... C'est pratique par moments de pouvoir regrouper des objets ensemble

```
Etudiant jimmy = new Etudiant(...);  
Musicien xavier = new Musicien(...);  
Cuisinier ricardo = new Cuisinier(...);  
  
NewsletterAbonnes[] abonnes = new NewsletterAbonnes[3];  
  
abonnes[0] = jimmy;  
abonnes[1] = xavier;  
abonnes[2] = ricardo;
```

- Est-ce qu'on redesign tous nos objets pour avoir une super-super-...classe qui a un courriel par-dessus tous les objets qui pourraient s'abonner à une Newsletter ?

Les Interfaces

- Les *interfaces* permettent de régler certains problèmes du genre sans avoir recours à l'héritage multiple
- Permet de définir des objets qui respectent une certaine interface (qui possède certaines méthodes publiques) sans pour autant hériter d'une classe

- Hiérarchie de classes => Familles d'objets
- Interface => Contrat à respecter dans une classe

Les Interfaces

Usage :

```
public class Truc extends Parent implements Interface1, Interface2, ...
    // code ici
}
```

- Une classe peut `extends` une seule classe, mais peut implémenter plusieurs interfaces
- Il n'y a pas de code dans une interface, seulement des définitions de fonctions à planter

Exemple d'interface : Contactable

```
public interface Contactable {  
    public String getCourriel(); // Pas de code dans la méthode  
}  
  
public class Etudiant implements Contactable {  
    // ...  
    @Override  
    public String getCourriel() { ... }  
}  
  
public class Musicien implements Contactable {  
    // ... doit redéfinir public String getCourriel()  
}  
  
public class Cuisinier implements Contactable {  
    // ... doit redéfinir public String getCourriel()  
}
```

Exemple d'interface : Contactable

On peut utiliser une interface comme un type :

```
Etudiant jimmy = new Etudiant(...);
Musicien xavier = new Musicien(...);
Cuisinier ricardo = new Cuisinier(...);

Contactable[] abonnes = new Contactable[3];

// Ok ! Les trois classes implémentent Contactable
abonnes[0] = jimmy;
abonnes[1] = xavier;
abonnes[2] = ricardo;

// Envoyer la newsletter à tout le monde
// ...
```

Exemple d'interface : Contactable

Re-Flashback de la première semaine :

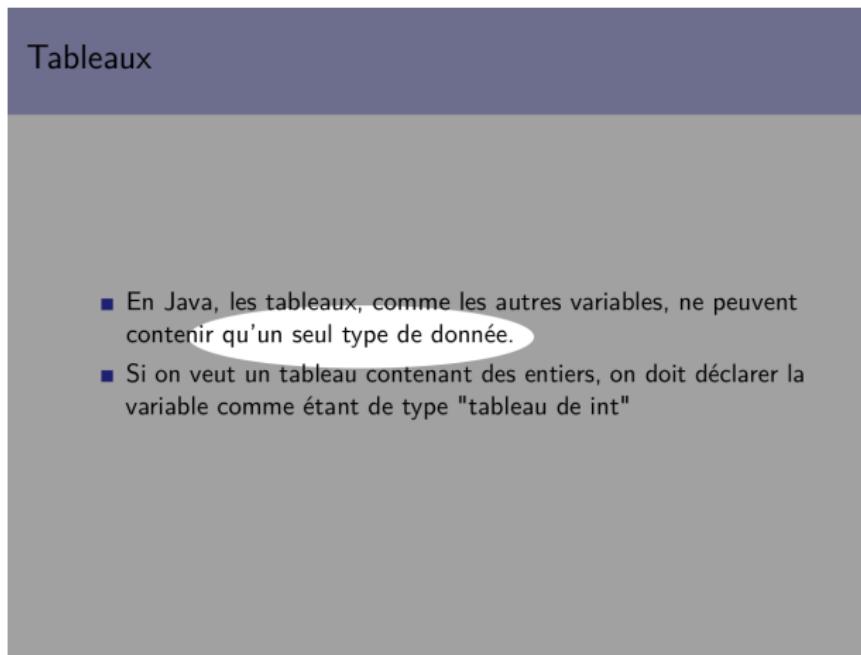


Figure 5: Re-Mensonge !

Exemple d'interface : Contactable

Toutes les interfaces implémentées comptent dans le type de l'objet :

```
Etudiant jimmy = new Etudiant(...);
Contactable[] abonnes = new Contactable[3];
abonnes[0] = jimmy;

// ...

jimmy instanceof Etudiant; // => true
jimmy instanceof Contactable; // => true
abonnes[0] instanceof Etudiant; // => true
abonnes[0] instanceof Contactable; // => true
```

Exemple d'interface : Contactable

Si on déclare une référence du type d'une interface, on peut seulement accéder aux méthodes définies pour l'interface :

```
Etudiant jimmy = new Etudiant(...);
Contactable[] abonnes = new Contactable[3];
abonnes[0] = jimmy;

// Erreur à la compilation !
// Type de la référence = Contactable
// Un "Contactable" ne définit pas la méthode etudier()
abonnes[0].etudier();

Etudiant abonne0 = (Etudiant) abonnes[0]; // Cast valide
abonne0.etudier(); // Ok
```

Exemple d'interface : Evaluable

Autre exemple : on pourrait déclarer une interface qui permet de trouver la "valeur" d'un objet :

```
public interface Evaluable {  
    public int valeurPourcents();  
}
```

Exemple d'interface : Evaluable

Un Etudiant serait évalué selon sa moyenne générale :

```
public class Etudiant extends Universitaire implements Evaluable {  
    // ...  
    @Override  
    public int valeurPourcents() {  
        return this.moyenne;  
    }  
}
```

Exemple d'interface : Evaluable

Un Professeur serait évalué selon ses évaluations de cours :

```
public class Professeur extends Universitaire implements Evaluable {  
    // ...  
    @Override  
    public int valeurPourcents() {  
        // Moyenne sur 100 des évaluations  
        // de cours du professeur  
        return this.moyenneEvaluationCours;  
    }  
}
```

Exemple d'interface : Evaluable

Un Cuisinier pourrait être évalué selon ses critiques de restaurant sur Yelp :

```
public class Cuisinier implements Evaluable {
    // ...
    @Override
    public int valeurPourcents() {
        // Moyenne sur 100 des évaluations
        // de son restaurant sur Yelp
        return this.reviewsYelp;
    }
}
```

Exemple d'interface : Evaluable

On pourrait alors écrire un algorithme général pour trier des objets de types différents en utilisant leur `valeurPourcents()` commune :

```
public static Evaluable[] trier(Evaluable[] elements) {  
    // Trier les éléments selon elements[i].valeurPourcents()  
}  
  
public static void main(String[] args) {  
  
    Etudiant jimmy = new Etudiant(...);  
    Professeur xavier = new Professeur(...);  
    Cuisinier ricardo = new Cuisinier(...);  
  
    Evaluable[] humains = new Evaluable[3];  
  
    Evaluable[] ordonnes = trier(humains);  
    System.out.println(ordonnes);  
    // => Affiche le meilleur humain  
}
```

Exemple : Super Mario Bros

On souhaite ajouter un bazooka comme item dans Mario³ :

- On peut faire exploser les items, les ennemis, les blocs...
- Mario ne peut pas exploser

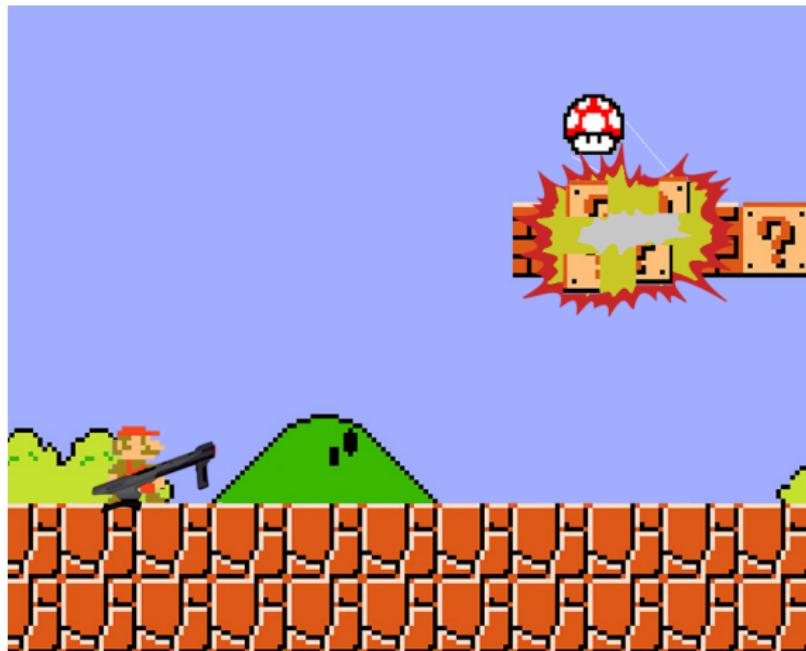


³source de l'image originale :

<https://www.nintendo.fr/Jeux/NES/Super-Mario-Bros-803853.html>

Exemple : Super Mario Bros

- Un bloc [?] qui explose libère son item



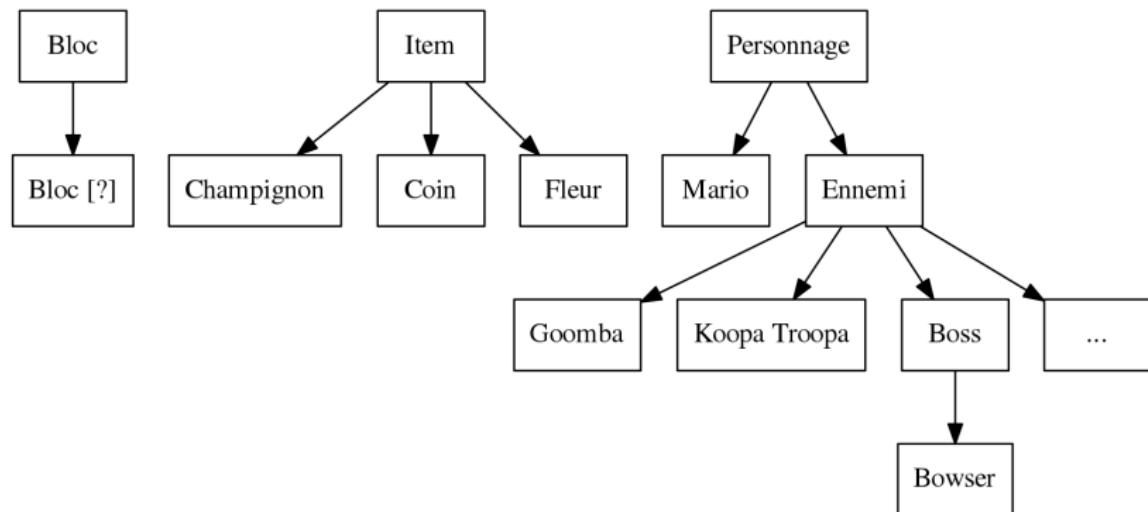
Exemple : Super Mario Bros

- Faire exploser un item doit faire monter le score de Mario

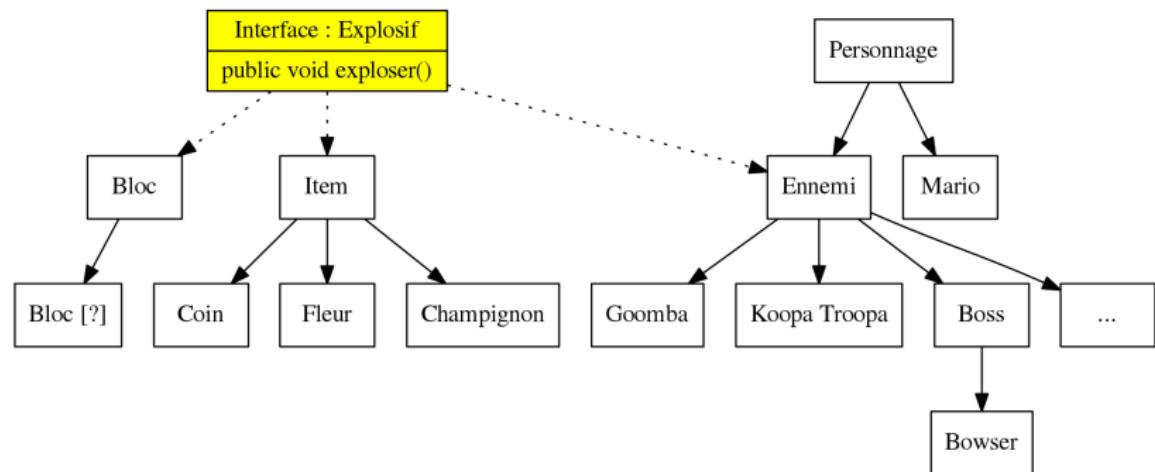


Exemple : Super Mario Bros

Comment adapter les objets existants ?



Exemple : Super Mario Bros



Polymorphisme

Un des concepts clés de l'Orienté Objet

Définition formelle (tirée de Wikipédia)

*Du grec ancien *polús* (plusieurs) et *morphê* (forme)*

Le concept consistant à fournir une interface unique à des entités pouvant avoir différents types

Polymorphisme

Appliqué de plusieurs façons différentes :

- Héritage
 - Redéfinition de méthodes (Override)
- Surcharge
 - Une même fonction va supporter plusieurs types d'entités (Overload)
- Programmation Générique
 - Types paramétrés (qu'on verra plus tard...)