



Le langage Java

Approche orientée objet

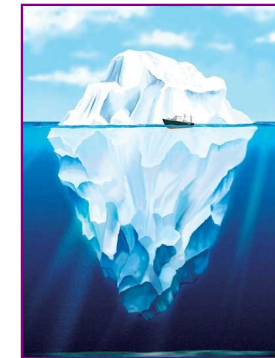
Sujets abordés

- L'encapsulation
- L'association
- L'héritage
- Le polymorphisme
- Les interfaces Java

Encapsulation

Problématique

- Une classe fournit un ensemble de services
 - Traitements
 - Transport de données
- Une classe doit être simple d'utilisation
 - Interface d'utilisation clair et simple
 - Complexité masquée à l'intérieur
- But : masquer les traitements internes
 - Dans une classe, possibilité de restreindre la visibilité des attributs et méthodes



Partie visible

*Traitements
internes*

Exemple (1/3)

- Calcul de la recette d'un aéroport

```
public class RecetteAeroport {  
    public long calculerRecetteTotale() {  
        //calcul recette des ventes billets  
        //calcul recette restaurants  
        //calcul taxes aéroport  
        //...  
    }  
}
```

Classe de calcul de la recette

```
public class Test {  
    public static void main(String[] args) {  
        RecetteAeroport r = new RecetteAeroport();  
        float recette = r.calculerRecetteTotale();  
        //...  
    }  
}
```

Appelant

- La méthode calculerRecetteTotale() est très longue. Comment la diviser en sous-méthodes ?

Exemple (2/3)

```
public class RecetteAeroport {  
    public long calculerRecetteTotale() {  
        //appel à toutes les autres méthodes  
    }  
  
    public long calculerVentesBilletsAvion() { //...}  
    public long calculerTaxesAeroport() { //... }  
    public long calculerRecetteRestaurants() { //...}  
}
```

Classe de calcul de la recette

```
public class Test {  
    public static void main(String[] args) {  
        RecetteAeroport r = new RecetteAeroport();  
        float recette = r.calculerRecetteTotale();  
        //...  
    }  
}
```

Appelant

- calculerRecetteTotale() fait appel aux autres méthodes de la classe
 - Les autres méthodes sont à usage interne de la classe.
 - Problème : la classe Test a accès à ces autres méthodes...

Exemple (3/3)

- Solution : les méthodes à usage interne sont privées
 - Elles ne sont pas visibles de la classe Test

```
public class RecetteAeroport {  
    public long calculerRecetteTotale() {  
        //appel à toutes les autres méthodes  
    }  
  
    private long calculerVentesBilletsAvion() { //...}  
    private long calculerTaxesAeroport() { //... }  
    private long calculerRecetteRestaurants() { //...}  
}
```

Classe de calcul de la recette

```
public class Test {  
    public static void main(String[] args) {  
        RecetteAeroport r = new RecetteAeroport();  
        float recette = r.calculerRecetteTotale();  
        //...  
    }  
}
```

Appelant

Règles de visibilité (1/2)

- Attributs et méthodes : plusieurs niveaux de visibilité
 - **public** : élément visible partout (mot-clé 'public').
 - **private** : élément à usage interne de la classe. Invisible de l'extérieur (mot-clé 'private').
 - **package** : élément visible partout dans le même package (pas de mot-clé, cas par défaut).

```
public class Test {  
    public int var1; // attribut public  
    private int var2; // attribut privé  
    int var3; // attribut de visibilité 'package'  
  
    private int method1() {} // méthode privée  
  
    public Test() {} // constructeur public  
}
```



Le quatrième niveau de visibilité (protected) sera vu plus loin dans les concepts objets (avec l'Héritage)

Règles de visibilité (2/2)

- Classes : deux niveaux de visibilité
 - **public** : classe visible partout (mot-clé 'public').
 - **package** : classe visible dans son package uniquement

```
public class Test {  
    // classe publique  
}
```

```
class Test {  
    // classe de visibilité package  
}
```

Encapsulation des variables (1/2)

- Méthodes getter et setter
 - Bonne pratique : déclarer les attributs d'une classe comme privés.
 - Utiliser des méthodes getter et setter pour accéder aux attributs.

```
public class Avion {  
    private long matricule;  
  
    public long getMatricule() {  
        return matricule;  
    }  
  
    public void setMatricule(long l) {  
        matricule = l;  
    }  
}
```



Tous les environnements de développement proposent des assistants pour générer les getters-setters

Encapsulation des variables (2/2)

- Intérêt des getters et setters ?
 - Il est possible de jouer sur la visibilité des méthodes d'accès.
 - Ex : variable en lecture seule en dehors du package.

Visibilité package (

```
public class Avion {  
    private long matricule;  
  
    public long getMatricule() {  
        return matricule;  
    }  
  
    void setMatricule(long l) {  
        matricule = l;  
    }  
}
```

- Un traitement spécifique peut être placé à chaque accès à une variable.
 - Message de log...

Association

- Il a été vu qu'une classe peut contenir des attributs de type primitif

```
public class Avion {  
    private long matricule;  
    //...  
}
```

- Une classe peut également porter des attributs typés par d'autres classes.

```
public class Avion {  
    private long matricule;  
    private Moteur moteur;  
  
    public Moteur getMoteur() { return moteur; }  
    public void setMoteur(Moteur moteur) {  
        this.moteur = moteur; }  
    //...  
}
```

```
public class Moteur {  
    private int identifiant;  
    private String dateRevision;  
    //...  
}
```

Références multiples

- Que produit le code suivant ?

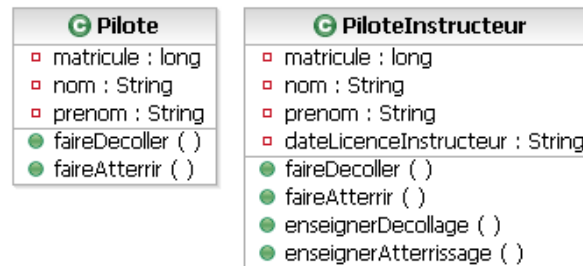
```
Avion monAvion = new Avion();  
Moteur monMoteur = new Moteur();  
monAvion.setMoteur(monMoteur);  
  
monMoteur.setDateRevision("21/02/2007");  
  
System.out.println(monAvion.getMoteur().getDateRevision());
```

- Les deux références pointent vers la même zone mémoire.

Héritage

Problématique

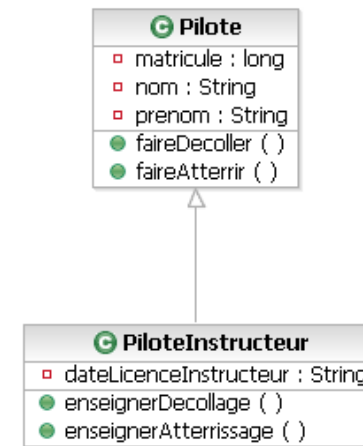
- Exemple
 - un pilote instructeur a un nom, un prénom, un matricule, une licence d'instructeur. Il sait piloter un avion (le faire décoller et atterrir); il peut également enseigner à des apprentis
- Modélisation basique



- Duplication de code \Rightarrow difficile à maintenir

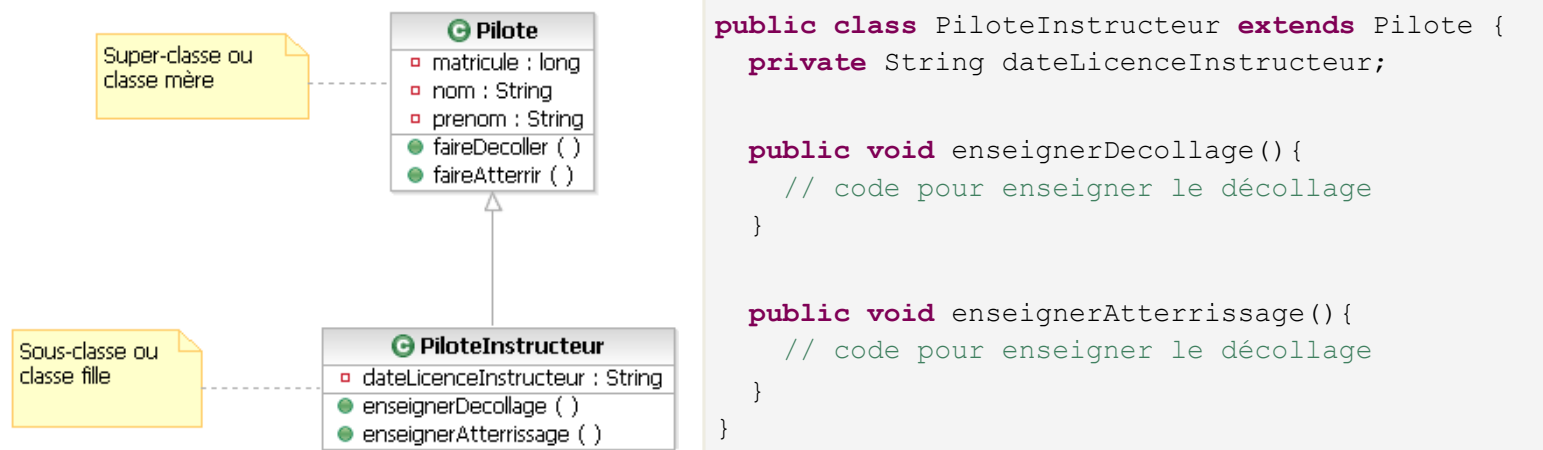
Solution : l'héritage

- Autre modélisation
 - un pilote instructeur **est un type particulier de pilote** qui a une licence d'instructeur. Il sait faire tout ce qu'un pilote sait faire, et de plus, il peut enseigner à des apprentis
- Relation d'héritage
 - Pilote définit les comportements et attributs communs
 - PiloteInstructeur ne définit que ce qui lui est propre
- Pas de duplication de code



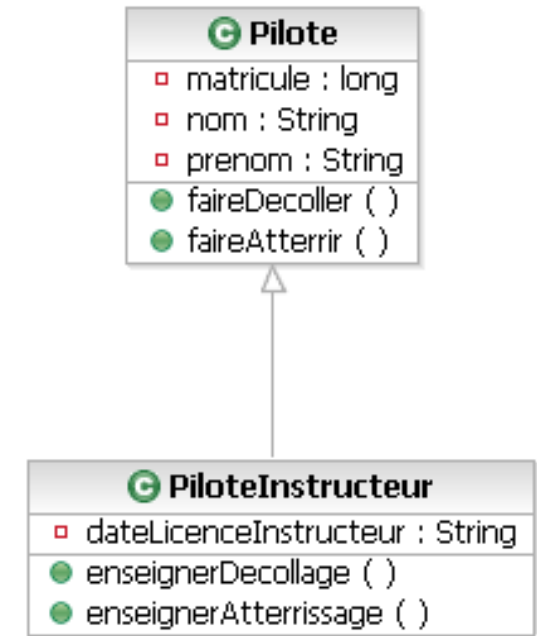
Relation d'héritage en Java

- Pas de code spécial dans la super-classe
- Mot-clé `extends` dans la déclaration de la sous-classe
 - `public class PiloteInstructeur extends Pilote`



Comportement (compilation)

- L'état et le comportement d'un objet sont définis le long de la hiérarchie
 - Structure interne (attributs)
 - matricule, nom, prénom, dateLicenseInstructeur
 - Comportement (méthodes)
 - enseignerDecollage(), enseignerAtterrissage(), faireDecoller(), faireAtterrir()



Comportement (exécution)

- Création

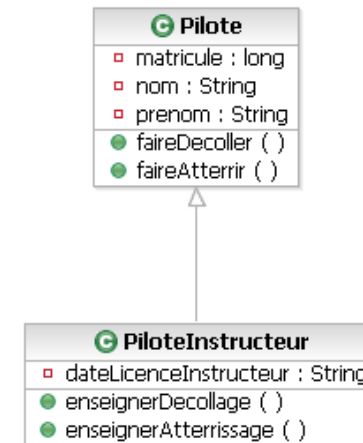
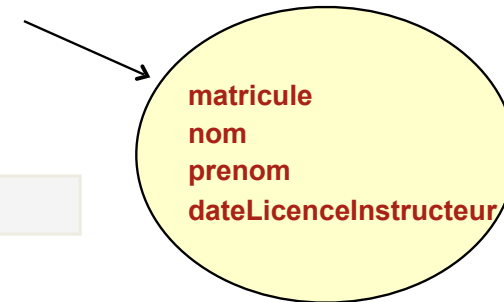
```
PiloteInstructeur instructeur = new PiloteInstructeur();
```

- 1 seul objet créé en mémoire avec les attributs de toutes les super-classes

- Appels de méthode

- Parcours de la hiérarchie de classes
 - à partir de la classe instanciée vers les super-classes
 - Arrêt dès la première implémentation trouvée (et exécution)

instructeur



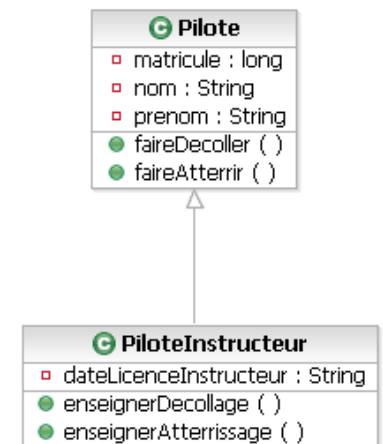
Héritage et visibilité

- accès aux attributs et méthodes des super-classes soumis aux contraintes de visibilité
 - public : oui, package : dépend du package de la classe
 - private : non

– Conséquence: le code suivant ne compile pas

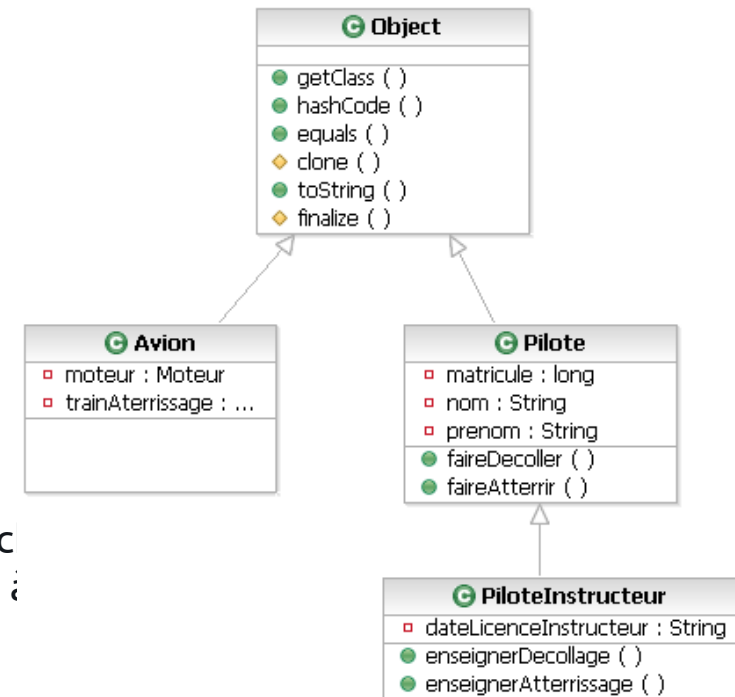
```
PiloteInstructeur instructeur = new PiloteInstructeur();  
instructeur.matricule = "1234567"; // attribut invisible
```

- protected autorise l'accès depuis les sous-classes :
 - donne aussi l'accès aux classes du package



Hiérarchie des classes

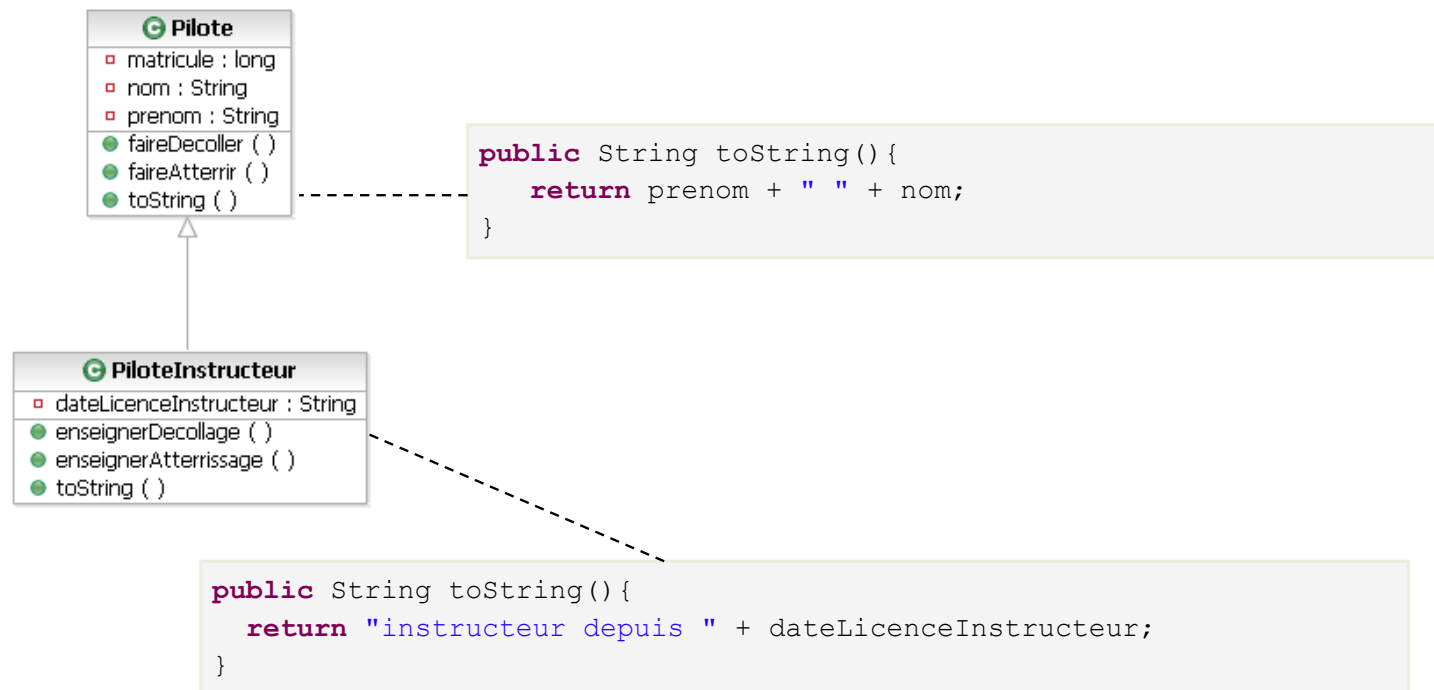
- Arborescence
 - Une classe a 1 seule super-classe
- Racine : `java.lang.Object`
 - Super-classe implicite
 - méthodes communes à tous les objets
 - `String toString()` : retourne la forme textuelle de l'objet, ici le nom long de la classe + un identifiant.
 - `int hashCode()` : renvoie un code pour les tables de hachage
 - `boolean equals(Object)` : indique si un objet est égal à le même objet en mémoire
 - ...



- Définir dans une sous-classe des méthodes avec la même signature que dans la super-classe
- Plusieurs sortes de redéfinitions
 - la définition de la super-classe est ignorée
 - extension de la définition de la super-classe
 - appel de la méthode de la super-classe
 - traitements spécifiques

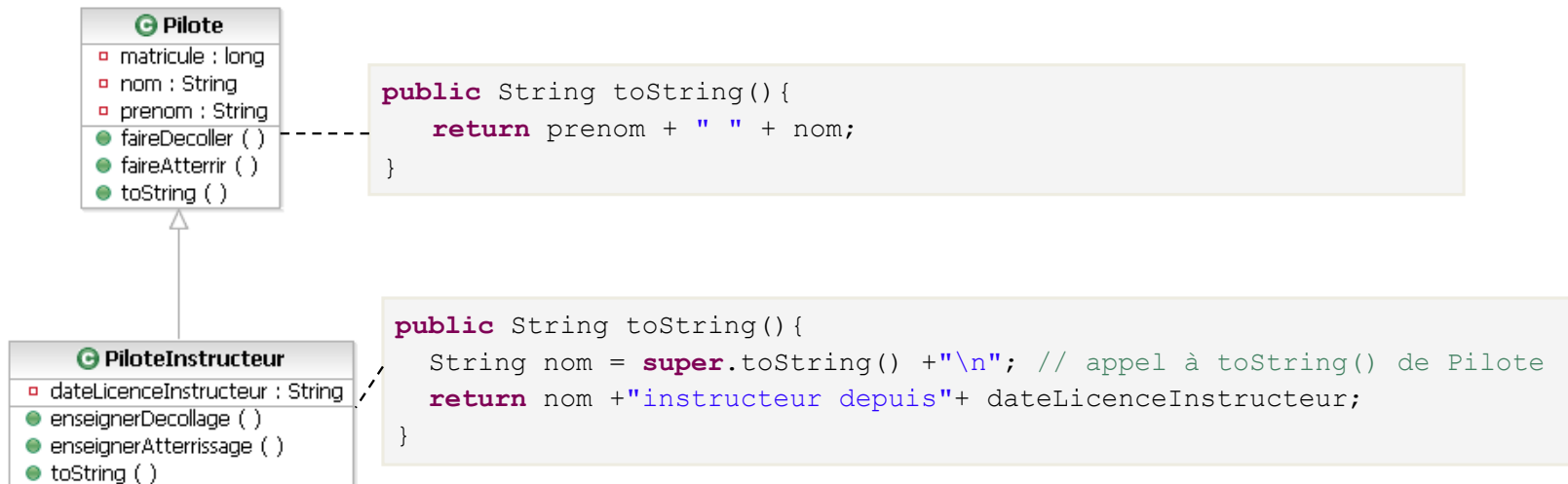
Redéfinition de méthodes

- 1^{er} cas : la définition de la super-classe est ignorée



Redéfinition de méthodes

- 2^{ème} cas : la définition de la super-classe est reprise
 - Comment référencer les attributs/méthode de la super-classe ?
 - Si appel au mot-clé **this** : boucle infinie !!!
 - Mot-clé **super**
 - commence la recherche de méthode ou d'attribut dans la super-classe

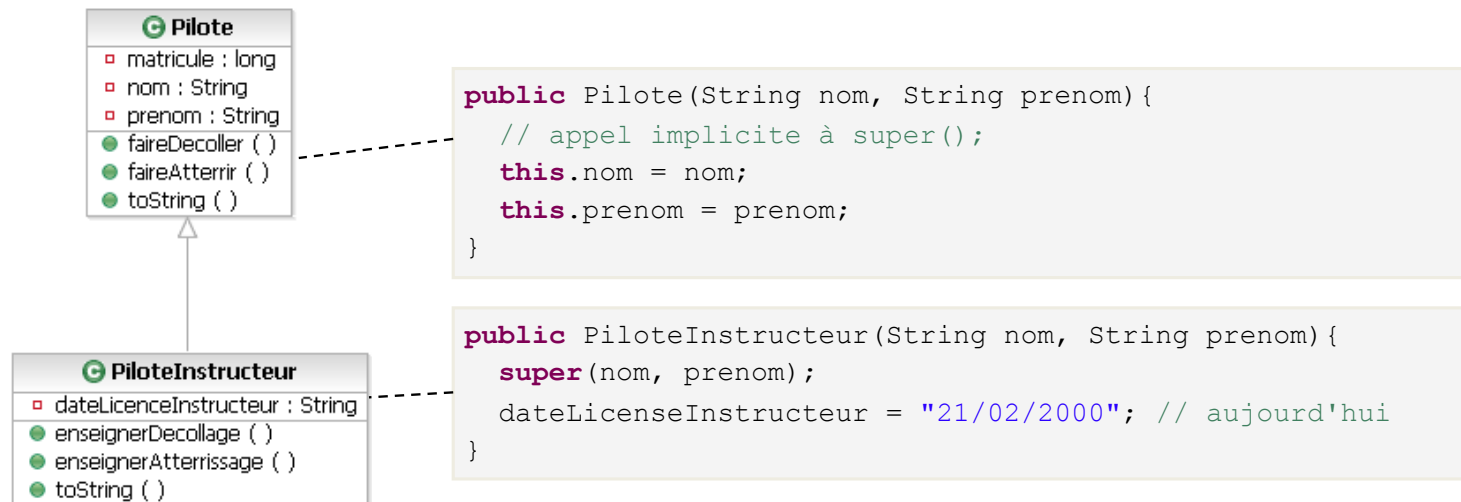


Redéfinition de méthodes

- Contraintes
 - le type de retour doit être strictement identique
 - la visibilité de la méthode de la super-classe ne peut pas être restreinte dans la sous-classe
 - par exemple : une méthode définie comme publique dans la super-classe ne peut pas être redéfinie comme privée dans la sous-classe
 - contrainte avec le mécanisme d'exception
 - sera abordée plus loin dans ce cours
- Ne pas confondre surcharge et redéfinition
 - Redéfinition : exactement la même signature
 - Surcharge : paramètres différents

Héritage et constructeurs

- Utilisation d'un constructeur hérité
 - appel d'un constructeur de la super-classe avec `super(...)`
 - Similitude avec utilisation de `this(...)`
 - puis traitements spécifiques
- ou appel implicite à `super()`



`super(...)` doit être la première instruction du constructeur

Interdire l'héritage

- Classe avec le mot-clé final
 - interdit la création de sous-classes

```
public final class PiloteInstructeur {  
    // attributs, constructeurs, méthodes  
}
```

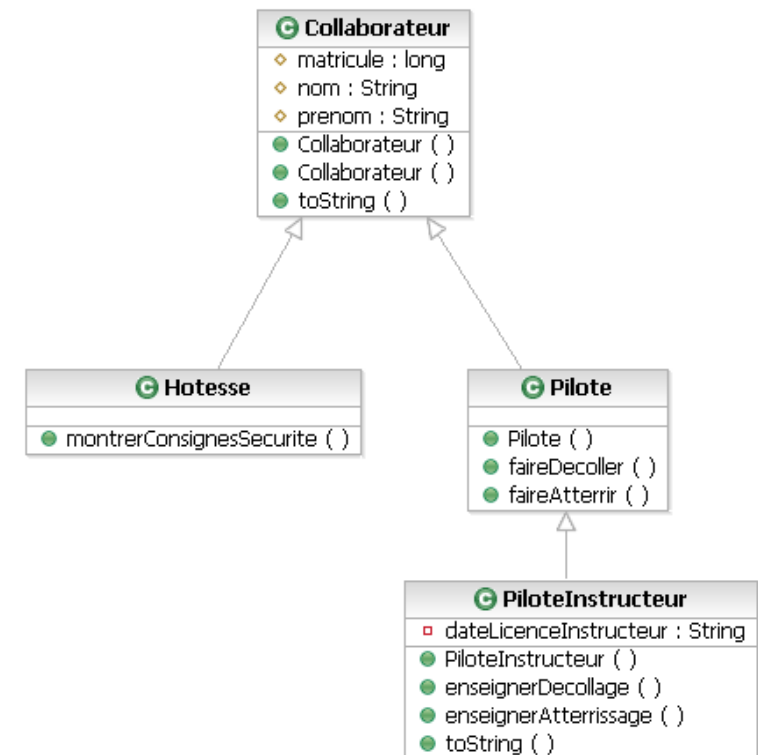
- Méthode avec le mot-clé final
 - interdit de redéfinir la méthode dans une sous-classe
 - A utiliser avec précaution

```
public final void faireDecoller(Avion unAvion) {  
    // code pour faire décoller l'avion  
}
```

- Rappel
 - pour un attribut, **final** signifie « constant »

Classe abstraite : Problématique

- Classe Collaborateur
 - Créée pour factoriser attributs et méthodes
 - Un collaborateur est une notion abstraite
 - `new Collaborateur()` n'a pas de sens
 - Comment l'interdire ?



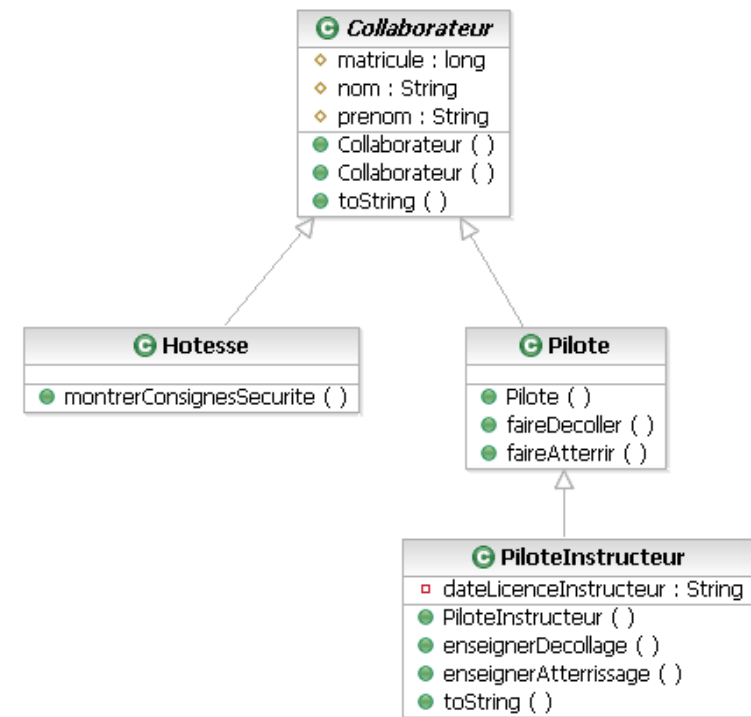
Classe abstraite

- Classe qui ne possède pas d'instance
- Propose une interface pour des sous-classes
 - fournit une interface commune
 - attributs
 - méthodes
 - les spécificités sont dans les sous-classes
 - ajout d'attributs
 - redéfinition/ajout de méthodes
- Utilisée dans le cadre d'une généralisation
 - factorisation des attributs et des méthodes
 - super-classe d'une hiérarchie

Déclaration de classe abstraite

- Collaborateur est abstraite
 - Notée en italique
- Mot-clé `abstract`

```
public abstract class Collaborateur {  
    protected long matricule;  
    protected String nom;  
    protected String prenom;  
  
    public Collaborateur(String nom, String prenom){  
        this.nom = nom;  
        this.prenom = prenom;  
    }  
    public String toString() {  
        return prenom + " " + nom;  
    }  
}
```



Classe abstraite et constructeurs

- Une classe abstraite peut avoir des constructeurs
 - Factorisation de l'initialisation
 - Appelés par les constructeurs des sous-classes
 - Mais appel direct (new) interdit

```
public abstract class Collaborateur {  
    protected String nom;  
    protected String prenom;  
  
    public Collaborateur(String nom, String prenom){  
        this.nom = nom;  
        this.prenom = prenom;  
    }  
}
```

```
public class Pilote extends Collaborateur{  
    public Pilote(String nom, String prenom){  
        super(nom, prenom);  
    }  
}
```


Méthode abstraite : Problématique

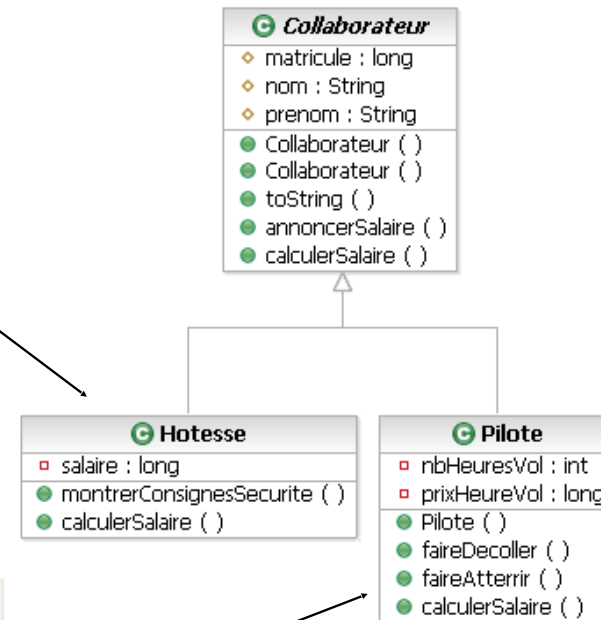
- Exemple salaires des collaborateurs
 - calculerSalaire renvoie le salaire
 - annoncerSalaire écrit le salaire dans la console
 - comment factoriser le code ?

Hotesse

```
public long calculerSalaire() {  
    return salaire;  
}  
  
public void annoncerSalaire() {  
    System.out.println("Salaire : " + calculerSalaire());  
}
```

Pilote

```
public long calculerSalaire() {  
    return nbHeuresVol*prixHeureVol;  
}  
  
public void annoncerSalaire() {  
    System.out.println("Salaire : " + calculerSalaire());  
}
```



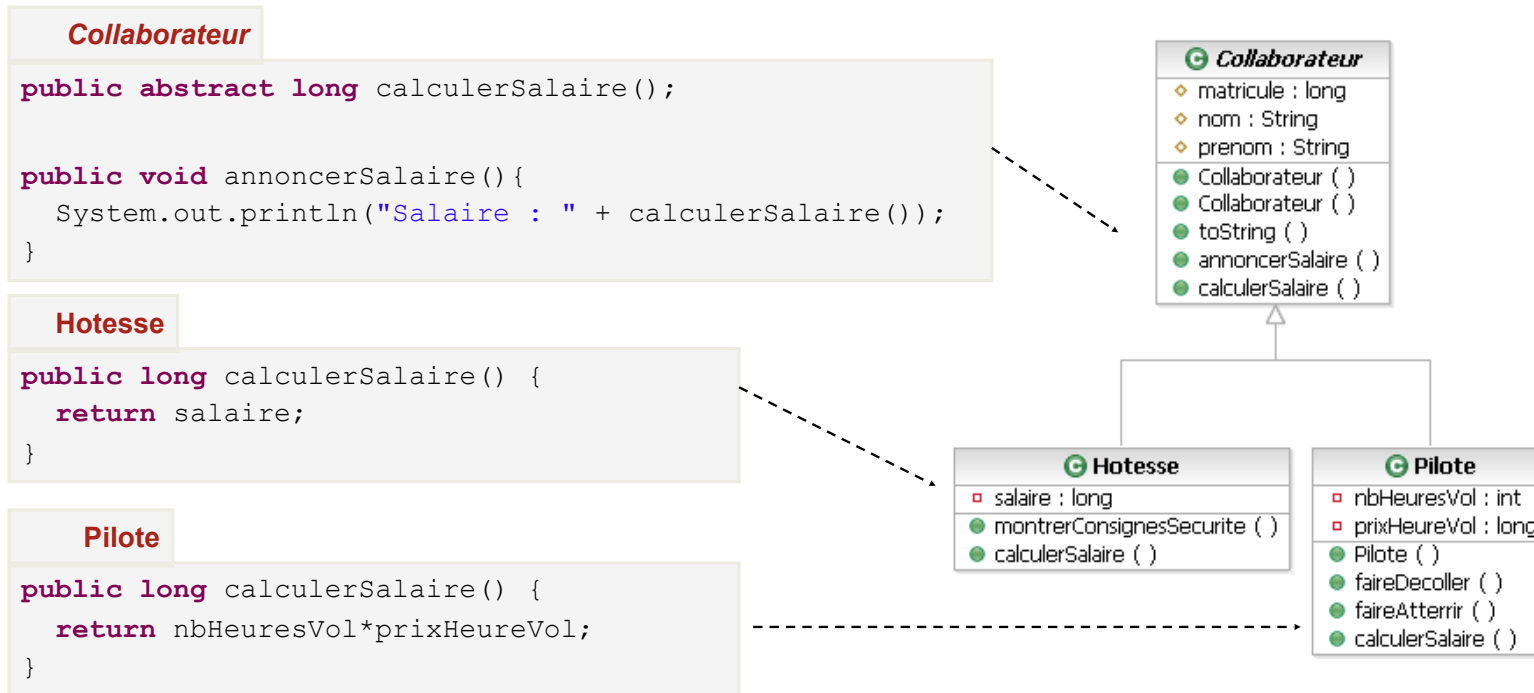
Méthode abstraite

- Mot-clé `abstract`
- Non complète
 - signature uniquement
 - Pas d'accolades `{ }`. Se termine par `;`
 - pas d'algorithme
 - spécifiée dans une classe abstraite

```
public abstract long calculerSalaire();
```

- Redéfinition obligatoire dans les sous-classes concrètes
 - définir l'algorithme
- Ne peut pas être privée, ni finale

Exemple abstraction + contraintes



Rappel : on recherche toujours les méthodes à partir de la classe courante

Contraintes liées à l'abstraction

- La redéfinition d'une méthode abstraite
 - est obligatoire dans une sous-classe concrète
 - sinon la sous-classe doit elle-même être abstraite
- Une classe abstraite
 - ne peut pas avoir d'instance
 - peut contenir des méthodes concrètes
 - peut contenir des constructeurs

Polymorphisme

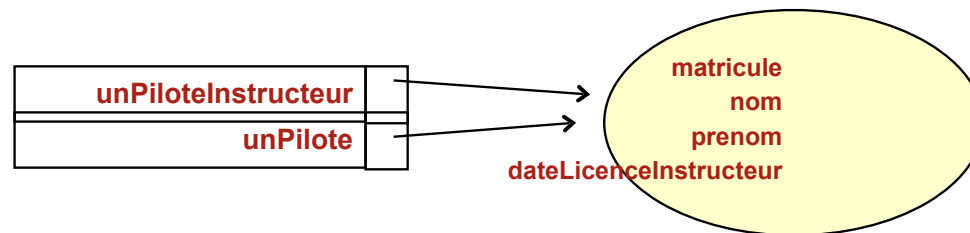
Polymorphisme : définition

- Un objet peut être vu sous plusieurs formes
 - Ex : un pilote instructeur peut être vu comme PiloteInstructeur, Pilote, Collaborateur ou Object
- Plusieurs objets différents peuvent être vus sous la même forme
 - Ex : un pilote instructeur, une hôtesse et un mécanicien peuvent être vus comme des collaborateurs

Un objet vu sous plusieurs formes

- Définitions
 - type réel d'un objet : classe dont le constructeur est appelé pour créer l'objet
 - type déclaré : classe utilisée pour manipuler l'objet
- Upcasting : type déclaré = superclasse du type réel
 - utilisation d'une variable du type de la super-classe à la place d'une variable du type réel de l'objet.

```
PiloteInstructeur unPiloteInstructeur = new PiloteInstructeur();  
Pilote unPilote = unPiloteInstructeur;
```



- Comportement
 - Compilation
 - vérification de la compatibilité type déclaré/type réel
 - si pas compatible, erreur de compilation
 - les méthodes/attributs accessibles sont ceux de la classe **déclarée**
 - Exécution
 - «binding dynamique» : détermine le type réel de l'objet à l'exécution, comportement de la classe réelle

Une forme pour plusieurs objets

- Plusieurs objets différents peuvent être vus sous la même forme
- Permet d'avoir des traitements génériques
- Exemple
 - Une gestionnaire de paie veut calculer le salaire d'un collaborateur
 - un collaborateur est un pilote, une hôtesse, un mécanicien ...
 - On peut manipuler directement le type Collaborateur

Exemple

- Implémentation

```
public class GestionnairePaie {  
    public void editerAttestationSalaire(Collaborateur collab){  
        long salaire = collab.calculerSalaire();  
        String texte = "attestation officielle. Montant du salaire : "+ salaire;  
        System.out.println(texte);  
    }  
}
```

```
public static void main(String[] args) {  
    GestionnairePaie paie = new GestionnairePaie();  
  
    Hotesse uneHotesse = new Hotesse();  
    paie.editerAttestationSalaire(uneHotesse);  
  
    Pilote unPilote = new Pilote();  
    paie.editerAttestationSalaire(unPilote);  
}
```

Avantages du polymorphisme

- Code plus lisible et plus maintenable

Procédural (pseudo code)

```
si collab = 'Pilote'  
    calculerSalairePilote  
si collab = 'Hotesse'  
    calculerSalaireHotesse  
si collab = 'Mecanicien'  
    calculerSalaireMecanicien
```

Objet

```
collab.calculerSalaire()
```

(=> mise en œuvre de
calculerSalaire dans
toutes les classes concernées)

Polymorphisme et redéfinition

- On associe souvent polymorphisme et redéfinition
- Cas pratique
 - `System.out.print(Object unObjet)`

```
public void print(Object obj) {  
    String texte = (obj == null) ? "null" : obj.toString();  
    write(texte);  
}
```

- méthode générique d'affichage
- mais le texte affiché dépend de la classe réelle de l'objet

Downcasting

- Besoin de convertir dans une autre classe, pour avoir accès aux méthodes spécifiques
- opérateur `cast ()`
 - exécution \Rightarrow essaie de convertir dans la classe
 - si impossible, erreur : `ClassCastException`
- Exemple

```
Collaborateur unCollaborateur = new Hotesse();  
/* je peux appliquer à unCollaborateur seulement les méthodes  
déclarées dans Collaborateur (et éventuellement redéfinies dans  
Hotesse)  
si je veux pouvoir appeler les méthodes propres à Hotesse */  
Hotesse uneHotesse = (Hotesse) unCollaborateur ;
```

Interfaces

- Exemple
 - Apprentissage du pilotage : utilisation d'un simulateur (puis d'un avion)
 - Mêmes boutons, mêmes comportements, mêmes "contrats"
⇒ mêmes signatures de méthode
 - Mais pas la même implémentation
 - Utiliser l'héritage ?
 - Inadapté : un simulateur et un avion sont très différents, pas d'attribut ou de méthode à factoriser
 - Un simulateur n'est pas une spécialisation d'avion.

Interface


- Interface : contrat que respecte une classe
 - Signature de méthodes, documentation
 - Pas d'implémentation
 - Des constantes
- Mise en œuvre
 - Déclaration : `interface` au lieu de `class`
 - Tous les attributs sont implicitement `public static final`
 - Toutes les méthodes sont implicitement `public` **et** `abstract`



Nom de l'interface : souvent suffixe en "able" : Comparable, Pilotable...

Exemple

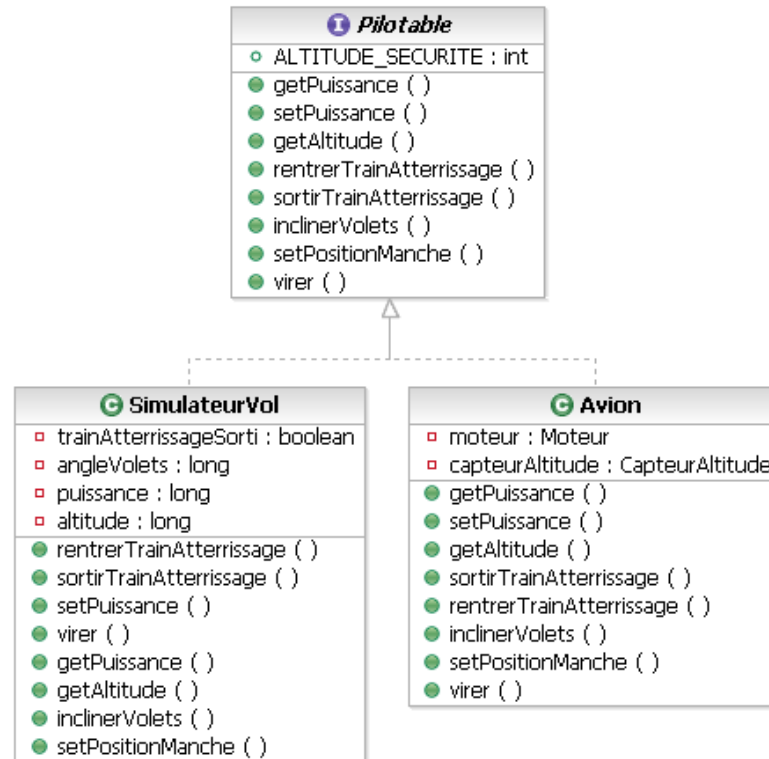
- Interface Pilotable

 Pilotable
◦ ALTITUDE_SECURITE : int
● getPuissance ()
● setPuissance ()
● getAltitude ()
● rentrerTrainAtterrissage ()
● sortirTrainAtterrissage ()
● inclinerVolets ()
● setPositionManche ()
● virer ()

```
public interface Pilotable {  
    public static final int ALTITUDE_SECURITE = 100;  
  
    public long getPuissance();  
    public void setPuissance(long puissanceCible);  
    public long getAltitude();  
    public void rentrerTrainAtterrissage();  
    public void sortirTrainAtterrissage();  
    public void inclinerVolets(long angle);  
    public void setPositionManche(long position);  
    public void virer(long angle);  
}
```

Respect de l'interface

- Avion et SimulateurVol respectent l'interface Pilotable



Classes d'implémentation

- Mot-clé `implements`
 - La classe implémente toutes les méthodes de l'interface
 - Sinon erreur compilation
 - doivent être déclarées `public`
 - + éventuellement d'autres méthodes

```
public class Avion implements Pilotable {  
    private Moteur moteur;  
    private CapteurAltitude capteurAltitude;  
  
    public long getPuissance() { return moteur.getPuissance(); }  
    public void setPuissance(long puissance) { moteur.setPuissance(puissance); }  
    public long getAltitude() { return capteurAltitude.getAltitude(); }  
    public void sortirTrainAtterrissage() { /* code */ }  
    public void rentrerTrainAtterrissage() { /* code */ }  
    public void inclinerVolets(long angle) { /* code */ }  
    public void setPositionManche(long position) { /* code */ }  
    public void virer(long angle) { /* code */ }  
}
```

- Interface = groupe de services rendus par la classe
 - ex : Pilotable, GestionnaireRadio, ...
- Utilisation comme type
 - accès seulement aux méthodes et attributs de l'interface (sinon erreur à la compilation)
 - permet d'appliquer des méthodes sans connaître la nature réelle de l'objet ⇒ indépendance vis-à-vis de l'implémentation
 - Implémentation de test
 - Implémentation "réelle"
 - Autre implémentation (autre vendeur par exemple)

- **Pilotable** : simulateur ou avion

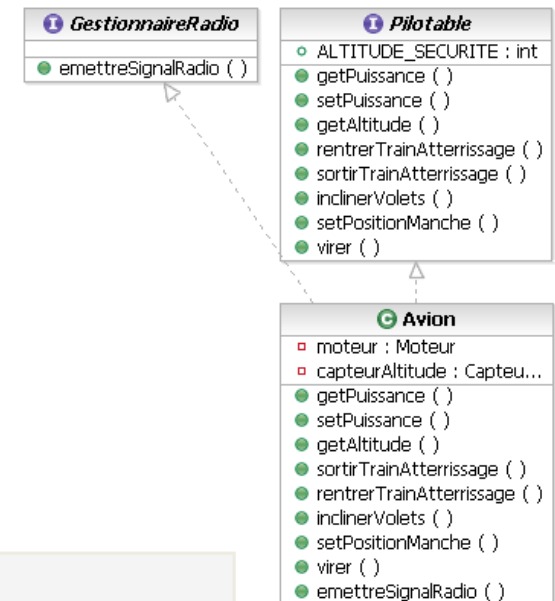
```
public void faireDecoller(Pilotable objetVolant) {  
    // code simpliste pour faire décoller l'avion  
    objetVolant.setPuissance(400);  
    objetVolant.inclinerVolets(10);  
    objetVolant.setPositionManche(3);  
    if (objetVolant.getAltitude() >= Pilotable.ALTITUDE_SECURITE) {  
        objetVolant.inclinerVolets(0);  
        objetVolant.rentreTrainAtterrissage();  
    }  
}
```

```
public static void main(String[] args) {  
    Pilote unPilote = new Pilote();  
    Pilotable pilotable = new SimulateurVol();  
    // autre possibilité : pilotable = new Avion();  
    unPilote.faireDecoller(pilotable);  
}
```

Héritage multiple

- N'existe pas en Java
 - une classe ne peut hériter que d'une **seule** classe
- Une classe peut implémenter plusieurs interfaces
 - Plusieurs "casquettes"
 - Objet pilotable
 - Objet avec GestionnaireRadio

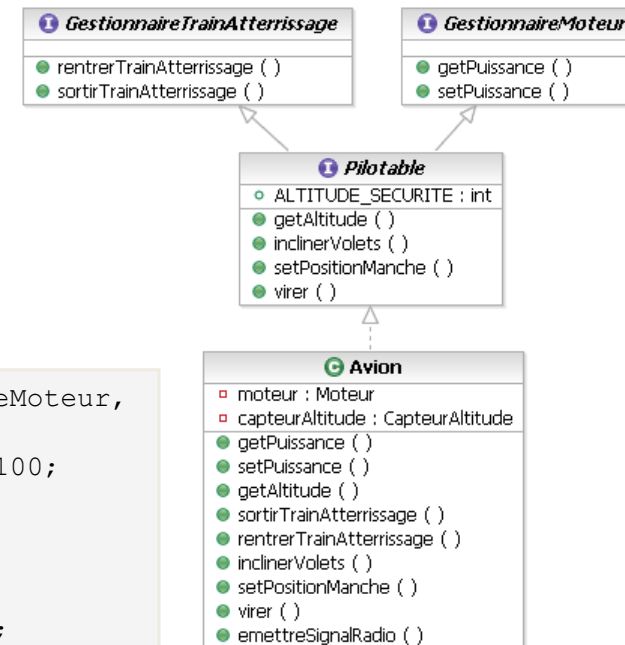
```
public class Avion implements Pilotable, GestionnaireRadio {  
    // attributs  
    // toutes les méthodes de l'interface Pilotable  
    public long getPuissance() { return moteur.getPuissance(); }  
    public void setPuissance(long puissance) { moteur.setPuissance(puissance); }  
  
    // toutes les méthodes de l'interface GestionnaireRadio  
    public void emettreSignalRadio(String message) { /* code */ }  
}
```



Héritage d'interface

- Une interface peut hériter de plusieurs interfaces
 - Ensemble des services des interfaces héritées

```
public interface Pilotable extends GestionnaireMoteur,  
GestionnaireTrainAtterrissage {  
    public static final int ALTITUDE_SECURITE = 100;  
  
    public long getAltitude();  
    public void inclinerVolets(long angle);  
    public void setPositionManche(long position);  
    public void virer(long angle);  
}
```



- Redéfinition de toutes les méthodes dans les classes d'implémentation