Introduction à la programmation et au développement d'applications en Java

Héritage de classes

Introduction

- Héritage 101
- Protection et redéfinition de membres
- Retour sur la classe **`Object`**
- Accès à un parent : le mot-clé **`super`**
- Contrôle de l'héritage : **`final`** et **`abstract`**
- Constructeurs et héritage

Héritage de classes

- Une classe peut être déclarée comme **héritant** d'une autre classe
- On dit qu'elle **dérive** de cette classe **parente** (**classe de base**)
- Le mot-clé **`extends`** est utilisé pour indiqué l'héritage
- La classe dérivée (**sous-classe**) possède automatiquement les caractéristiques de sa classe de base (**superclasse**), et peut ajouter des fonctionnalités (**spécialisation**)

Héritage de fonctionnalités

```
VolCargo vc = new
volCargo();

vc.ajouter(1.0, 2.5, 3.0);

Passager tom = new
               Passager(0,
2);

// fonctionnalité héritée
vc.ajouterUnPassager(tom);
```

```
class VolCargo extends Vol {
  float maxVolume = 1000.0f;
  float volumeUtilise;

  public void ajouter(float h, float l, float p) {
    float volume = h * l * p;
    if (placeDispoPour(volume))
      volumeUtilise += volume;
    else
      gererTropVolumineux();
  }

  private boolean placeDispoPour(float volume) {
    return volumeUtilise + volume <= maxVolume;
  }

  private void gererTropVolumineux() { … }
}
```

Héritage et affectation

Un objet d'un type d'une classe dérivée peut être affecté à une référence de la classe de base

```
Vol v = new VolCargo();

Passager tom = new
               Passager(0,
2);

v.ajouterUnPassager(tom);

// Erreur compilation
// La référence v ne peut
// accéder aux membres
// de la classe dérivée
v.ajouter(1.0, 2.5, 3.0);
```

```
class VolCargo extends Vol {
  float maxVolume = 1000.0f;
  float volumeUtilise;

  public void ajouter(float h, float l, float p) {
    float volume = h * l * p;
    if (placeDispoPour(volume))
      volumeUtilise += volume;
    else
      gererTropVolumineux();
  }

  private boolean placeDispoPour(float volume) {
    return volumeUtilise + volume <= maxVolume;
  }

  private void gererTropVolumineux() { … }
}
```

Héritage et affectation

Cela permet d'appeler des services de la classe de base sans se soucier de savoir à quel type exact de Vol on a affaire

```
Vol[] escadron = new Vol[3];

escadron[0] = new Vol();
escadron[1] = new
VolCargo();
escadron[2] = new Vol();

for (Vol v : escadron) {
  v.ajouterUnPassager(...);
}
```

```
class VolCargo extends Vol {
  float maxVolume = 1000.0f;
  float volumeUtilise;

  public void ajouter(float h, float l, float p) {
    float volume = h * l * p;
    if (placeDispoPour(volume))
      volumeUtilise += volume;
    else
      gererTropVolumineux();
  }

  private boolean placeDispoPour(float volume) {
    return volumeUtilise + volume <= maxVolume;
  }

  private void gererTropVolumineux() { … }
}
```

Masquage de champ : hasardeux

Un champ de même nom dans la classe dérivée **cache le champ de la classe de base, y compris dans les méthodes parentes**

```
class VolCargo extends Vol {
  // autres membres...
  int nbPlaces = 12;
}
```

```
Vol v = new Vol();
System.out.print(v.nbPlaces);     // 150

VolCargo vc = new VolCargo();
System.out.print(vc.nbPlaces);    // 12

Vol v2 = new VolCargo();
System.out.print(v2.nbPlaces);    //  150

v2.ajouterUnPassager();// nbPlaces == 150 !
vc.ajouterUnPassager();// nbPlaces == 150 !!
```

```
class Vol {
  // autres membres...
  int nbPlaces = 150;

  public void ajouterUnPassager() {
    if (placeDispo())
      nbPassagers++;
    else
      gererTropNombreux();
  }
  private boolean placeDispo() {
    return nbPassagers < nbPlaces;
  }
}
```

Mêmes membres dérivés

- Le **masquage de champ** (attribut) de la classe de base dans la classe dérivée (même nom) est donc permis mais **à éviter**
- La **redéfinition de méthode** de la classe de base dans la classe dérivée (même signature) est en revanche **très utile** et constitue une fondation importante de la programmation orientée objet
- Redéfinir ≠ masquer

Redéfinition de méthodes

Le *runtime* Java vérifie toujours le **type exact** de l'instance pour appeler la méthode redéfinie si elle existe

```
class VolCargo extends Vol {

  int getNbPlaces() { return 12; }
}
```

```
Vol v = new Vol();
System.out.print(v.getNbPlaces); // 150

VolCargo vc = new VolCargo();
System.out.print(vc.getNbPlaces); // 12

Vol v2 = new VolCargo();
// appel de la méthode redéfinie
System.out.print(v2.getNbPlaces());   12

//le type réel de l'instance est utilisé
v2.ajouterUnPassager();// getNbPlaces==12
vc.ajouterUnPassager();// getNbPlaces==12
```

```
class Vol {

  int getNbPlaces() { return 150; }

  public void ajouterUnPassager() {
    if (placeDispo())
      nbPassagers++;
    else
      gererTropNombreux();
  }
  private boolean placeDispo() {
    return nbPassagers < getNbPlaces();
  }
}
```

L'annotation `@Override`

- Java ne fournit pas de moyen pour indiquer qu'une méthode est redéfinissable, contrairement à d'autres langages objet
- En Java, toutes les méthodes sont redéfinissables, sauf si vous spécifiez explicitement le contraire (voir plus loin)
- Pour indiquer qu'on redéfinit une méthode (et pour faire vérifier par le compilateur que vous avez bien la même signature), on utilise l'annotation **`@override`** juste au dessus de la redéfinition dans la classe dérivée (pas obligatoire mais c'est une bonne pratique)

L'annotation `@Override`

- Précisez toujours l'annotation **`@Override`** quand vous redéfinissez une méthode, pour indiquer que c'est bien votre intention
- Le lecteur du code saura immédiatement qu'il s'agit d'une méthode redéfinie
- Le compilateur vérifiera qu'il existe bien une méthode ayant la même signature dans la classe de base, et vous préviendra dans le cas contraire

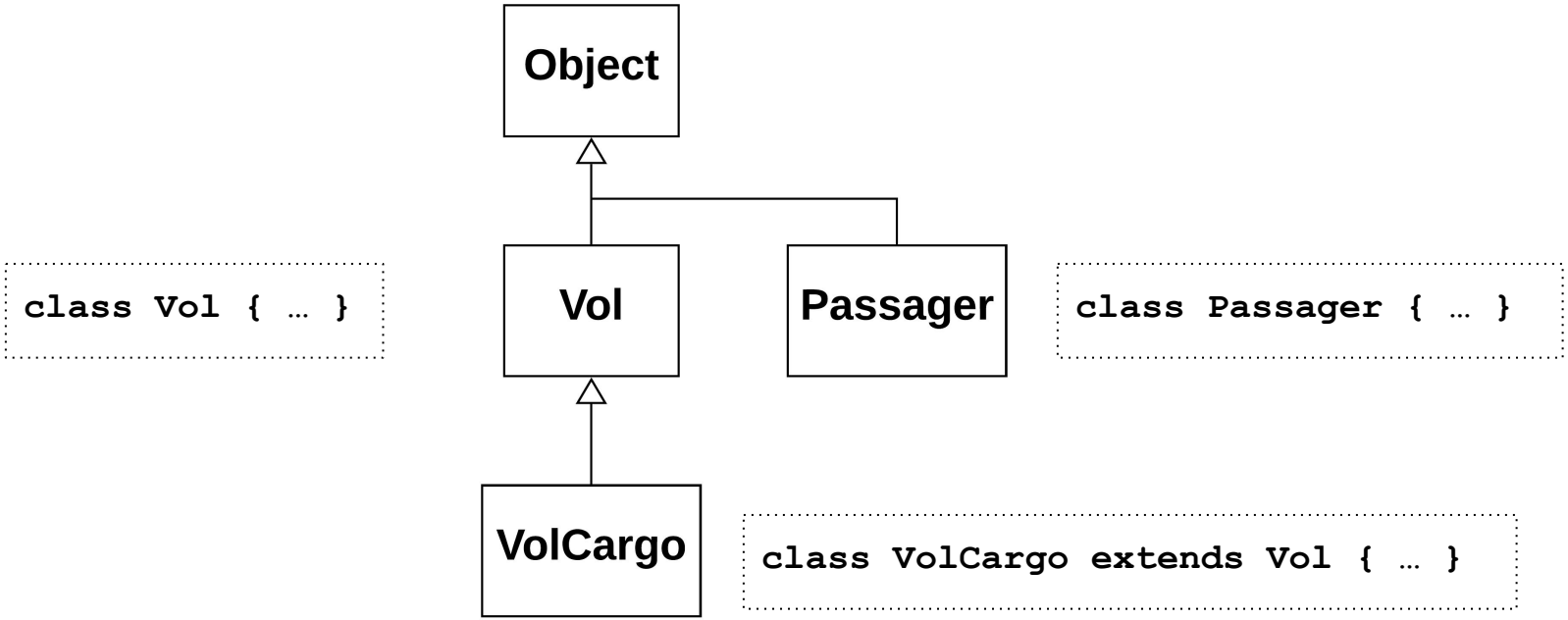
```
class VolCargo extends Vol {
  @Override
  int getNbPlaces() { return 12; }
}
```

```
class Vol {

  int getNbPlaces() { return 150; }

  public void ajouterUnPassager() {
    if (placeDispo())
      nbPassagers++;
    else
      gererTropNombreux();
  }
  private boolean placeDispo() {
    return nbPassagers < getNbPlaces();
  }
}
```

La classe `Object`

- La classe **`Object`** est la racine de la hiérarchie de classes Java
- Toute classe hérite donc automatiquement de **`Object`** et a ses caractéristiques
- Utile pour déclarer des variables/champs/paramètres qui peuvent référencer n'importe quelle objet
- Définit des méthodes qui sont ainsi héritées, disponibles et redéfinissables pour tous les objets

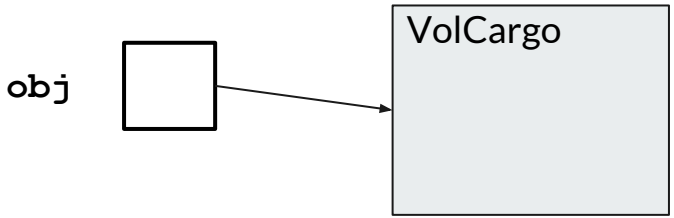
Hiérarchie de classes



Manipulation d'objets

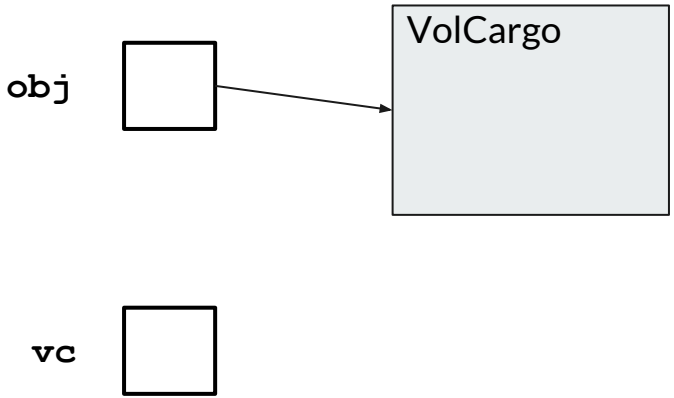
```
Object bidules = new Object[3];
bidules[0] = new Vol();
bidules[1] = new Passager();
bidules[2] = new VolCargo();

Object obj = new Passager();
obj = new Vol[20];
obj = new VolCargo();
obj.ajouter(1.0, 2.0, 3.5); // Erreur !
```

Manipulation d'objets



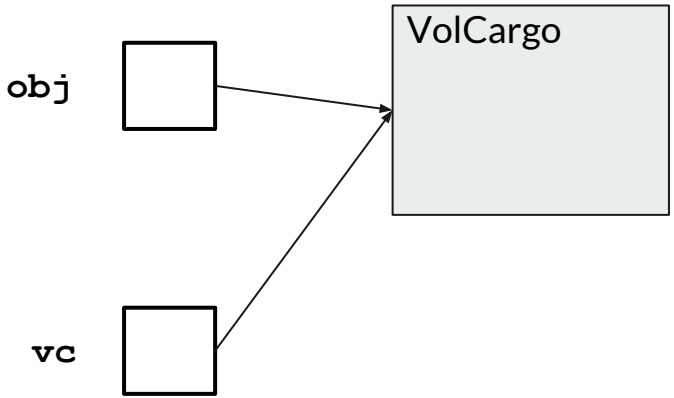

```
Object bidules = new Object[3];
bidules[0] = new Vol();
bidules[1] = new Passager();
bidules[2] = new VolCargo();

Object obj = new Passager();
obj = new Vol[20];
obj = new VolCargo();
obj.ajouter(1.0, 2.0, 3.5); // Erreur !
```

Manipulation d'objets



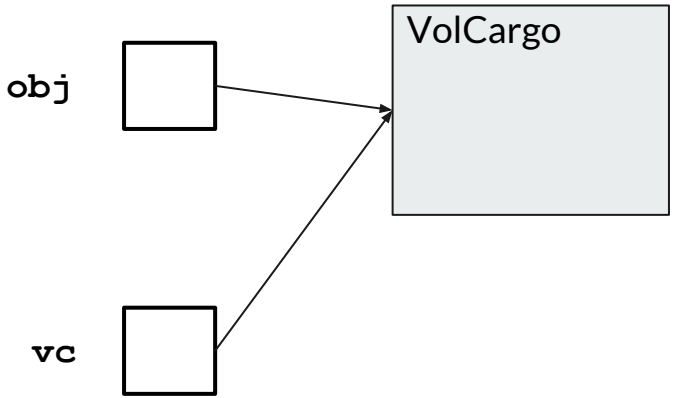

```
Object bidules = new Object[3];
bidules[0] = new Vol();
bidules[1] = new Passager();
bidules[2] = new VolCargo();

Object obj = new Passager();
obj = new Vol[20];
obj = new VolCargo();
obj.ajouter(1.0, 2.0, 3.5); // Erreur !
VolCargo vc = obj;  // Erreur !
```

Manipulation d'objets




```
Object bidules = new Object[3];
bidules[0] = new Vol();
bidules[1] = new Passager();
bidules[2] = new VolCargo();

Object obj = new Passager();
obj = new Vol[20];
obj = new VolCargo();
obj.ajouter(1.0, 2.0, 3.5); // Erreur !
VolCargo vc = obj;    // Erreur !


  VolCargo vc = (VolCargo) obj;
  vc.ajouter(1.0, 2.0, 3.5);
```

L'opérateur `instanceof`



À l'exécution, l'opérateur **`instanceof`** vérifie que l'objet donné (à gauche) est bien une instance de la classe donnée (à droite)

```
Object bidules = new Object[3];
bidules[0] = new Vol();
bidules[1] = new Passager();
bidules[2] = new VolCargo();

Object obj = new Passager();
obj = new Vol[20];
obj = new VolCargo();
obj.ajouter(1.0, 2.0, 3.5); // Erreur !
VolCargo vc = obj;    // Erreur !

if (obj instanceof VolCargo) {
  VolCargo vc = (VolCargo) obj;
  vc.ajouter(1.0, 2.0, 3.5);
}
```

Méthodes de la classe `Object`

Méthode	Description
`clone`	crée une nouvelle instance qui copie l'instance courante
`hashCode`	code qui permet d'identifier rapidement si deux objets sont potentiellement différents, à redéfinir si **`equals`** l'est aussi
`getClass`	information sur le type de l'instance courante
`finalize`	utilisé pour certains scénario de « nettoyage » de ressources
`toString`	chaîne de caractères décrivant l'objet courant
`equals`	compare l'objet courant à un autre objet

Égalité d'objets

Quand peut-on dire que deux objets sont égaux ? Réponse : ça dépend de vos besoins

```
Vol v1 = new Vol(175);
Vol v2 = new Vol(175);

if (v1 == v2) // égalité de
références
  qqchose();  // non exécuté

if (v1.equals(v2)) //références aussi
  qqchose();  // non exécuté
```

```
class Vol {
  // autres membres...
  private int numero;
  private char classe;









}
```

Égalité d'objets

Quand peut-on dire que deux objets sont égaux ? Réponse : ça dépend de vos besoins

```
Vol v1 = new Vol(175);
Vol v2 = new Vol(175);

if (v1 == v2) // égalité de
références
  qqchose();  // non exécuté

if (v1.equals(v2))
  qqchose();
```

```
class Vol {
  // autres membres...
  private int numero;
  private char classe;

  @Override
  public boolean equals(Object obj) {



    Vol autre = (Vol) obj;



  }
}
```

Égalité d'objets

Quand peut-on dire que deux objets sont égaux ? Réponse : ça dépend de vos besoins

```
Vol v1 = new Vol(175);
Vol v2 = new Vol(175);

if (v1 == v2) // égalité de références
  qqchose();  // non exécuté

if (v1.equals(v2)) //méthode redéfinie
  qqchose();  // exécuté
```

```
class Vol {
  // autres membres...
  private int numero;
  private char classe;

  @Override
  public boolean equals(Object obj) {



    Vol autre = (Vol) obj;

    return (numero == autre.numero
      && classe == autre.classe);
  }
}
```

Égalité d'objets

Quand peut-on dire que deux objets sont égaux ? Réponse : ça dépend de vos besoins

```
Vol v1 = new Vol(175);
Vol v2 = new Vol(175);

if (v1 == v2) // égalité de références
  qqchose();  // non exécuté

if (v1.equals(v2)) //méthode redéfinie
  qqchose();  // exécuté

if (v1.equals("toto"))   // erreur
  qqchose();             // d'exécution
```

```
class Vol {
  // autres membres...
  private int numero;
  private char classe;

  @Override
  public boolean equals(Object obj) {



    Vol autre = (Vol) obj;

    return (numero == autre.numero
      && classe == autre.classe);
  }
}
```

Égalité d'objets

Quand peut-on dire que deux objets sont égaux ? Réponse : ça dépend de vos besoins

```
Vol v1 = new Vol(175);
Vol v2 = new Vol(175);

if (v1 == v2) // égalité de références
  qqchose();  // non exécuté

if (v1.equals(v2)) //méthode redéfinie
  qqchose();  // exécuté

if (v1.equals("toto"))  // pas d'erreur
  qqchose();  // non exécuté
```

```
class Vol {
  // autres membres...
  private int numero;
  private char classe;

  @Override
  public boolean equals(Object obj) {
    if (!(obj instanceof Vol))
      return false;

    Vol autre = (Vol) obj;

    return (numero == autre.numero
      && classe == autre.classe);
  }
}
```

Référence spéciale : `super`

- **`super`** est similaire à **`this`**, c'est une **référence implicite à l'objet courant**
- Mais **`super`** traite l'objet **comme si c'était une instance de la classe de base**
- On l'utilise donc pour **accéder à des membres de la classe de base** qui ont été redéfinis
- Exemple : dans la méthode **`equals`**, on voudrait appeler la version de la classe parente (**`Object`**) pour tester rapidement si les deux références sont égales ; on peut y accéder dans la classe dérivée (**`Vol`**) en invoquant **`equals`** depuis la référence spéciale **`super`** : `if (super.equals) ...`

super : utilisation

```
Vol v1 = new Vol(175);
Vol v2 = v1;

if (v1 == v2) // égalité de références
  qqchose();  // exécuté

if (v1.equals(v2)) // méthode redéfinie
  qqchose();  // exécuté
// mais algorithme de test d'égalité
// potentiellement long alors qu'un
// simple test de références suffirait
```

```
class Vol {
  // autres membres...
  private int numero;
  private char classe;

  @Override
  public boolean equals(Object obj) {



    if (!(obj instanceof Vol))
      return false;

    Vol autre = (Vol) obj;

    return (numero == autre.numero
      && classe == autre.classe);
  }
}
```

super : utilisation

```
Vol v1 = new Vol(175);
Vol v2 = v1;

if (v1 == v2) // égalité de références
  qqchose();  // exécuté

if (v1.equals(v2)) // test rapide
  qqchose();  // exécuté
```

```
class Vol {
  // autres membres...
  private int numero;
  private char classe;

  @Override
  public boolean equals(Object obj) {
    if (super.equals(obj))
      return true;

    if (!(obj instanceof Vol))
      return false;

    Vol autre = (Vol) obj;

    return (numero == autre.numero
      && classe == autre.classe);
  }
}
```

Contrôler l'héritage

- Par défaut, toutes les classes peuvent être héritées, et les classes dérivées peuvent utiliser et/ou redéfinir les méthodes de base
- On peut modifier ce comportement dans la définition de la classe
- Le mot-clé **`final` interdit l'héritage de classe et/ou la redéfinition de méthodes** dans les classes dérivées

`final` : interdire héritage/redéfinition

```
// Sur classe
final public class Passager {

  // la classe ne peut plus
  // être héritée du tout

}
```

```
class VolCargo extends Vol {
  float maxVolume = 1000.0f;
  float volumeUtilise;

  // Sur méthode ; celle-ci ne pourra pas être
  // redéfinie dans une éventuelle sous-classe
  public final void ajouter(float h, float l,
                                        float p) {
    float volume = h * l * p;
    if (placeDispoPour(volume))
      volumeUtilise += volume;
    else
      gererTropVolumineux();
  }
  private boolean placeDispoPour(float volume) {
    return volumeUtilise + volume <= maxVolume;
  }
}
```

Contrôler l'héritage

- Par défaut, toutes les classes peuvent être héritées, et les classes dérivées peuvent utiliser et/ou redéfinir les méthodes de base
- On peut modifier ce comportement dans la définition de la classe
- Le mot-clé **`final` interdit l'héritage de classe et/ou la redéfinition de méthodes** dans les classes dérivées
- Le mot-clé **`abstract` force l'héritage de classe et/ou la redéfinition de méthodes** dans les classes dérivées

`abstract` : forcer héritage/redéfinition

```
public class Pilote {
  private Vol volCourant;

  public void assigner(Vol v) {
    if (peutAccepter(v))
      volCourant = v;
    else
      gererNonAcceptation();
  }

  public boolean
              peutAccepter(Vol v) {
    // problème : implémentation
    // dépend du type de pilote...
  }
  private void gererNonAcceptation()
  { ... }
}
```

`abstract` : forcer héritage/redéfinition

```
public abstract class Pilote {
  private Vol volCourant;

  public void assigner(Vol v) {
    if (peutAccepter(v))
      volCourant = v;
    else
      gererNonAcceptation();
  }

  public abstract boolean
              peutAccepter(Vol v) ;



  private void gererNonAcceptation()
  { ... }
}
```

`abstract` : forcer héritage/redéfinition

```
public abstract class Pilote {
  private Vol volCourant;

  public void assigner(Vol v) {
    if (peutAccepter(v))
      volCourant = v;
    else
      gererNonAcceptation();
  }

  public abstract boolean
                peutAccepter(Vol v) ;



  private void gererNonAcceptation()
  { ... }
}
```

```
public class PiloteDeCargo extends Pilote
{
  @Override
  public boolean peutAccepter(Vol v) {
    return v.getNbPassagers() == 0;
  }
}
```

```
public class PiloteToutesLicences
                                extends Pilote
{
  @Override
  public boolean peutAccepter(Vol v) {
    return true;
  }
}
```

Héritage et constructeurs

- En Java, **les constructeurs ne sont pas hérités** dans les sous-classes
- Un constructeur de la classe de base **doit toujours être appelé**
- Si vous ne le faites pas explicitement, **Java invoquera automatiquement le constructeur sans arguments** de la classe de base
- Pour appeler un constructeur de la classe de base, on utilise la notation `super(arguments)` pour préciser la signature du constructeur souhaité
- Cela doit alors être la **première ligne** du constructeur de la classe dérivée

Héritage et constructeurs

```
public class Vol {
  private int numero;

  public Vol() { }

  public Vol(int numero) {
    this.numero = numero;
  }
}
```

```
VolCargo vc = new VolCargo();
```

```
public class VolCargo extends Vol {
  float maxVolume = 1000.0f;

  // Java génère automatiquement
  // un constructeur par défaut sans args
  // qui va appeler le constructeur
  // sans args de la classe de base

}
```

Héritage et constructeurs

```
public class Vol {
  private int numero;

  public Vol() { }

  public Vol(int numero) {
    this.numero = numero;
  }
}
```

```
VolCargo vc = new VolCargo();
VolCargo vc294 =
      new VolCargo(294); // Erreur
```

```
public class VolCargo extends Vol {
  float maxVolume = 1000.0f;

  // Pas de constructeur avec un argument

}
```

Héritage et constructeurs

```
public class Vol {
  private int numero;

  public Vol() { }

  public Vol(int numero) {
    this.numero = numero;
  }
}
```

```
VolCargo vc = new VolCargo();
VolCargo vc294 =
      new VolCargo(294); // OK
```

```
public class VolCargo extends Vol {
  float maxVolume = 1000.0f;

  public VolCargo(int numero) {
    super(numero);
  }

}
```

Héritage et constructeurs

```
public class Vol {
  private int numero;

  public Vol() { }

  public Vol(int numero) {
    this.numero = numero;
  }
}
```

```
VolCargo vc = new VolCargo();
VolCargo vc294 =
      new VolCargo(294);
VolCargo vc295 =
      new VolCargo(295, 2000.0f);
```

```
public class VolCargo extends Vol {
  float maxVolume = 1000.0f;

  public VolCargo(int numero) {
    super(numero);
  }

  public VolCargo(int numero, float maxVol) {
    this(numero);
    maxVolume = maxVol;
  }

}
```

Héritage et constructeurs

```
public class Vol {
  private int numero;

  public Vol() { }

  public Vol(int numero) {
    this.numero = numero;
  }
}
```

```
VolCargo vc = new VolCargo();
VolCargo vc294 =
      new VolCargo(294);
VolCargo vc295 =
      new VolCargo(295, 2000.0f);
```

```
public class VolCargo extends Vol {
  float maxVolume = 1000.0f;

  public VolCargo(int numero) {
    super(numero);
  }

  public VolCargo(int numero, float maxVol) {
    this(numero);
    maxVolume = maxVol;
  }

}
```

Héritage et constructeurs

```
public class Vol {
  private int numero;

  public Vol() { }

  public Vol(int numero) {
    this.numero = numero;
  }
}
```

```
VolCargo vc = new VolCargo(); //OK
VolCargo vc294 =
      new VolCargo(294);
VolCargo vc295 =
      new VolCargo(295, 2000.0f);
```

```
public class VolCargo extends Vol {
  float maxVolume = 1000.0f;

  public VolCargo(int numero) {
    super(numero);
  }

  public VolCargo(int numero, float maxVol) {
    this(numero);
    maxVolume = maxVol;
  }

  public VolCargo() { }



}
```

Héritage et constructeurs

```
public class Vol {
  private int numero;

  public Vol() { }

  public Vol(int numero) {
    this.numero = numero;
  }
}
```

```
VolCargo vc = new VolCargo();
VolCargo vc294 =
               new VolCargo(294);
VolCargo vc295 =
      new VolCargo(295, 2000.0f);
VolCargo vcGros =
           new VolCargo(5000.0f);
```

```
public class VolCargo extends Vol {
  float maxVolume = 1000.0f;

  public VolCargo(int numero) {
    super(numero);
  }

  public VolCargo(int numero, float maxVol) {
    this(numero);
    maxVolume = maxVol;
  }

  public VolCargo() { }

  public VolCargo(float maxVol) {
    // appel implicite de Vol() dans base
    maxVolume = maxVol;
  }
```

Résumé

- L'héritage permet à une classe d'être définie avec les caractéristiques d'une **classe parente** (**classe de base**), grâce au mot-clé **`extends`**
- La **classe dérivée** (**sous-classe**) peut **redéfinir** les méthodes de son parent, et on devrait toujours utiliser l'annotation **`@Override`** pour indiquer la redéfinition
- Toutes les classes dérivent directement ou indirectement de la classe **`Object`**
- Par défaut, des variables-références sont **égales** seulement si les références qu'elles désignent sont les mêmes (**même objet pointé**)
- On peut redéfinir **`Object.equals`** pour modifier ce comportement
- **`super`** permet d'accéder à l'instance courante comme si c'était une instance de la classe de base
- **`final`** et **`abstract`** permettent de **contrôler l'héritage et la redéfinition**
- Les constructeurs ne sont pas hérités, on utilise super pour les appeler