Introduction : classes et objets

Programation Orientée Objet

Jean-Christophe Routier
Licence mention Informatique
Université des Sciences et Technologies de Lille

Préliminaire

- cours de **Programmation Orientée Objet**, pas de JAVA
  (même si certains aspects langage seront présentés)

- ne pas mélanger concepts/idées avec la technique du langage.
- il faut distinguer :
  - le concept de l'illustration
  - l'esprit de la concrétisation dans un langage
- la maîtrise d'un langage ne suffit pas à faire un bon programmeur

Préliminaire

- cours de **Programmation Orientée Objet**, pas de JAVA
  (même si certains aspects langage seront présentés)

- ne pas mélanger concepts/idées avec la technique du langage.
- il faut distinguer :
  - le concept de l'illustration
  - l'esprit de la concrétisation dans un langage
- la maîtrise d'un langage ne suffit pas à faire un bon programmeur

Préliminaire

- cours de **Programmation Orientée Objet**, pas de JAVA
  (même si certains aspects langage seront présentés)

- ne pas mélanger concepts/idées avec la technique du langage.
- il faut distinguer :
  - le concept de l'illustration
  - l'esprit de la concrétisation dans un langage
- la maîtrise d'un langage ne suffit pas à faire un bon programmeur

Préliminaire

- cours de **Programmation Orientée Objet**, pas de JAVA
  (même si certains aspects langage seront présentés)

- ne pas mélanger concepts/idées avec la technique du langage.
- il faut distinguer :
  - le concept de l'illustration
  - l'esprit de la concrétisation dans un langage
- la maîtrise d'un langage ne suffit pas à faire un bon programmeur

2 programmeurs

- le créateur de bibliothèques (API)
  ↪ "programmeur pour les programmeurs"
- l'utilisateur d'API ou "programmeur client"
  ↪ "programmeur pour les utilisateurs"

Il faut être les deux !

Il est plus difficile d'être le premier :

- voir plus loin que son pb (**abstraire**)
- anticiper sur une réutilisation

Adage

Un bon programmeur est un programmeur **paresseux** (pas fainéant ! : fait néant)
↪ travailler bien, et peut être plus, maintenant pour en faire moins plus tard.

*Programmer c'est **investir**.*

2 programmeurs

- le créateur de bibliothèques (API)
  ↪ "programmeur pour les programmeurs"
- l'utilisateur d'API ou "programmeur client"
  ↪ "programmeur pour les utilisateurs"

Il faut être les deux !

Il est plus difficile d'être le premier :

- voir plus loin que son pb (**abstraire**)
- anticiper sur une réutilisation

Adage

Un bon programmeur est un programmeur **paresseux** (pas fainéant ! : fait néant)
↪ travailler bien, et peut être plus, maintenant pour en faire moins plus tard.

*Programmer c'est **investir**.*

2 programmeurs

- le créateur de bibliothèques (API)
  ↪ "programmeur pour les programmeurs"
- l'utilisateur d'API ou "programmeur client"
  ↪ "programmeur pour les utilisateurs"

Il faut être les deux !

Il est plus difficile d'être le premier :

- voir plus loin que son pb (**abstraire**)
- anticiper sur une réutilisation

Adage

Un bon programmeur est un programmeur **paresseux** (pas fainéant ! : fait néant)
↪ travailler bien, et peut être plus, maintenant pour en faire moins plus tard.
*Programmer c'est **investir**.*

2 programmeurs

- le créateur de bibliothèques (API)
  ↪ "programmeur pour les programmeurs"
- l'utilisateur d'API ou "programmeur client"
  ↪ "programmeur pour les utilisateurs"

Il faut être les deux !

Il est plus difficile d'être le premier :

- voir plus loin que son pb (**abstraire**)
- anticiper sur une réutilisation

Adage

Un bon programmeur est un programmeur **paresseux** (pas fainéant ! : fait néant)
↪ travailler bien, et peut être plus, maintenant pour en faire moins plus tard.

*Programmer c'est **investir**.*

Conception

Programmation = algorithmique **ET** architecture (logicielle)

"conception/design"

maintenance/évolution/réutilisation

POO : importance de l'organisation des objets/classes entre eux.

- programmation $\Longrightarrow$ modélisation d'un problème
- programmer = représenter de manière abstraite le pb à résoudre.
- nécessité d'un langage pour la définition de l'abstraction

- On peut programmer dans (presque) tous les paradigmes avec (presque) tous les langages, mais un langage donné est mieux adapté à (supporte mieux) tel ou tel "style" de programmation.

Approche modulaire de la programmation

Programmation impérative et

- analyse descendante des problèmes, par décomposition
- approche modulaire pour une programmation propre
  - restreindre l'accès/protéger
  - proposer une interface
  - faciliter le travail à plusieurs et la réutilisabilité
- abstraction de données
  $\hookrightarrow$ types définis par le programmeur

Programmation objet

- Prolonger la logique de la programmation modulaire.
- Mieux coller à la réalité du problème pour une modélisation "plus naturelle"

- Nous réfléchissons en terme d'objets... (*Descartes*)
  - ... appréhendons la notion de *Livre* en général...
    ↪ caractéristiques : auteur, titre, année de parution, etc.
    ↪ actions : voir le titre, lire, connaître le nombre de pages, etc.
  - ... et distinguons des livres particuliers.
    ↪ obéissent à la notion générale "*Livre*" : "possèdent" les caractéristiques de la notion et peuvent "subir" les actions

Un problème est défini par un ensemble d'objets qu'il faut modéliser.

5 concepts clés

1. abstraction
2. modularisation
3. encapsulation
4. réutilisation (agrégation/composition)
5. polymorphisme (des objets)

Langage à objets (purs)

Alan Kay - SmallTalk

- "Tout est objet"
- "Un programme est un regroupement d'objets qui se disent quoi faire par envois de messages"
- "Chaque objet a sa propre mémoire constituée d'autres objets"
- "Chaque objet a un type"
- "Tous les objets d'un type donné peuvent recevoir le même type de messages"

Langage à objets (purs)

Alan Kay - SmallTalk

- "Tout est objet"
- "Un programme est un regroupement d'objets qui se disent quoi faire par envois de messages"
- "Chaque objet a sa propre mémoire constituée d'autres objets"
- "Chaque objet a un type"
- "Tous les objets d'un type donné peuvent recevoir le même type de messages"

Langage à objets (purs)

Alan Kay - SmallTalk

- "Tout est objet"
- "Un programme est un regroupement d'objets qui se disent quoi faire par envois de messages"
- "Chaque objet a sa propre mémoire constituée d'autres objets"
- "Chaque objet a un type"
- "Tous les objets d'un type donné peuvent recevoir le même type de messages"

Langage à objets (purs)

Alan Kay - SmallTalk

- "Tout est objet"
- "Un programme est un regroupement d'objets qui se disent quoi faire par envois de messages"
- "Chaque objet a sa propre mémoire constituée d'autres objets"
- "Chaque objet a un type"
- "Tous les objets d'un type donné peuvent recevoir le même type de messages"

Langage à objets (purs)

Alan Kay - SmallTalk

- "Tout est objet"
- "Un programme est un regroupement d'objets qui se disent quoi faire par envois de messages"
- "Chaque objet a sa propre mémoire constituée d'autres objets"
- "Chaque objet a un type"
- "Tous les objets d'un type donné peuvent recevoir le même type de messages"

JAVA

la VM de Lisp, les concepts de Smalltalk, le typage d'ADA, la syntaxe de C

développé par Sun, gratuit, libre et maintenant ouvert (licence GPL2)

- langage orienté objet (pas 100% objet)
- langage de classes
- indépendance OS/architecture : multi plate-forme
  $\hookrightarrow$ utilisation d'une (J)VM
  "*compile once, run everywhere*"
- nombreuses bibliothèques/API (gratuites) $\hookrightarrow$ la "plate-forme Java"
  $\Longrightarrow$ simplification de la programmation (pas de pb pour réutiliser)
- "vivant" (APIs en évolution) et utilisé
- permet/favorise la construction de gros projets

- langage compilé
- fortement typé (+ polymorphisme)
- gestion dynamique de la mémoire $\hookrightarrow$ utilisation d'un GC
- gestion des erreurs (exceptions)
- multi-threading
- Java et le réseau ("network is the computer")
- réflexivité (introspection)
- pas d'arithmétique de pointeur

Objet

Objet = identité + état + comportement

avec...

L'**identité** permet d'exploiter le **comportement** d'un objet. Le comportement agit sur et est influencé par l'**état**.

Une identité

Un objet forme un tout.

- deux objets différents ont des identités différentes
- on peut désigner l'objet (y faire référence)

Un état

- ensemble de propriétés/caractéristiques définies par des valeurs
- permet de le personnaliser/distinguer des autres objets
- peut évoluer dans le temps

attributs ("data member")

Un comportement

- ensemble des traitements que peut accomplir un objet (ou que l'on peut lui faire accomplir)

méthodes ("member functions")

- on s'adresse à l'objet par **envoi de messages**
  $\hookrightarrow$ on "demande" à l'objet de faire ceci ou cela

  envoi de message = accès à un attribut ou invocation de méthode

- le comportement définit l'ensemble des messages qu'un objet peut recevoir

- **interface** de l'objet
  - = "ensemble" des manières que l'on a pour interagir avec l'objet
  - = ensemble des messages reconnus par l'objet
- "interface de comportement"

- certains objets présentent les mêmes caractéristiques :
  - identités différentes mais
    ↪ états définis par les mêmes propriétés
    ↪ même interface de comportement

exemple :

"*Le Seigneur des Anneaux*"
de John Ronald Reuel Tolkien
paru en 1954

et

"Dune"
de Frank Herbert
paru en 1965

sont caractérisés par les mêmes *propriétés*
↪ auteur, titre, année, texte
(associées à des valeurs différentes)

et ont la même interface de *comportement*
↪ on peut leur faire accomplir les mêmes actions : emprunter, lire, les imprimer, etc.

mais ont des *identités* différentes

il en serait de même pour ($\sim$) tous les livres

- certains objets présentent les mêmes caractéristiques :
  - identités différentes mais
    ↪ états définis par les mêmes propriétés
    ↪ même interface de comportement

exemple :

"*Le Seigneur des Anneaux*" de John Ronald Reuel Tolkien paru en 1954	et	"Dune" de Frank Herbert paru en 1965

sont caractérisés par les mêmes *propriétés*
↪ auteur, titre, année, texte
(associées à des valeurs différentes)
et ont la même interface de *comportement*
↪ on peut leur faire accomplir les mêmes actions ↪ on peut les lire, les imprimer, etc.
mais ont des *valeurs* différentes

il en serait de même pour ($\sim$) tous les livres

- certains objets présentent les mêmes caractéristiques :
  - identités différentes mais
    ↪ états définis par les mêmes propriétés
    ↪ même interface de comportement

exemple :

"*Le Seigneur des Anneaux*"
de John Ronald Reuel Tolkien
paru en 1954

et

"Dune"
de Frank Herbert
paru en 1965

sont caractérisés par les mêmes *propriétés*
↪ auteur, titre, année, texte
(associées à des valeurs différentes)

et ont la même interface de *comportement*
↪ on peut leur faire accomplir les mêmes actions ↪ on peut les lire, les imprimer, etc.

mais ont des *identités* différentes

il en serait de même pour ($\sim$) tous les livres

- certains objets présentent les mêmes caractéristiques :
  - identités différentes mais
    ↪ états définis par les mêmes propriétés
    ↪ même interface de comportement

exemple :

"*Le Seigneur des Anneaux*"
de John Ronald Reuel Tolkien
paru en 1954

et

"Dune"
de Frank Herbert
paru en 1965

sont caractérisés par les mêmes *propriétés*
↪ auteur, titre, année, texte
(associées à des valeurs différentes)

et ont la même interface de *comportement*
↪ on peut leur faire accomplir les mêmes actions ↪ on peut les lire, les imprimer, etc.

mais ont des *identités* différentes

il en serait de même pour ($\sim$) tous les livres

- certains objets présentent les mêmes caractéristiques :
  - identités différentes mais
    ↪ états définis par les mêmes propriétés
    ↪ même interface de comportement

exemple :

"*Le Seigneur des Anneaux*"
de John Ronald Reuel Tolkien
paru en 1954

et

"Dune"
de Frank Herbert
paru en 1965

sont caractérisés par les mêmes *propriétés*
↪ auteur, titre, année, texte
(associées à des valeurs différentes)

et ont la même interface de *comportement*
↪ on peut leur faire accomplir les mêmes actions ↪ on peut les lire, les imprimer, etc.

mais ont des *identités* différentes

il en serait de même pour ($\sim$) tous les livres

- certains objets présentent les mêmes caractéristiques :
  - identités différentes mais
    ↪ états définis par les mêmes propriétés
    ↪ même interface de comportement

exemple :

"*Le Seigneur des Anneaux*"
de John Ronald Reuel Tolkien
paru en 1954

et

"Dune"
de Frank Herbert
paru en 1965

sont caractérisés par les mêmes *propriétés*
↪ auteur, titre, année, texte
(associées à des valeurs différentes)

et ont la même interface de *comportement*
↪ on peut leur faire accomplir les mêmes actions ↪ on peut les lire, les imprimer, etc.

mais ont des *identités* différentes

il en serait de même pour ($\sim$) tous les livres

Tous les livres obéissent à un même schéma

$\Longrightarrow$ on peut en abstraire un patron (abstrait), un "moule", un modèle, etc

Le moule définit

- les attributs qui caractérisent l'état
- l'interface et sa réaction = le comportement $\Longrightarrow$ les méthodes

de tous les moulages qui en seront issus

"moulages = objets"

Tous les livres obéissent à un même schéma

$\Longrightarrow$ on peut en abstraire un patron (abstrait), un "moule", un modèle, etc

Le moule définit

- les attributs qui caractérisent l'état
- l'interface et sa réaction = le comportement $\Longrightarrow$ les méthodes

de tous les moulages qui en seront issus

"moulages = objets"

Tous les livres obéissent à un même schéma

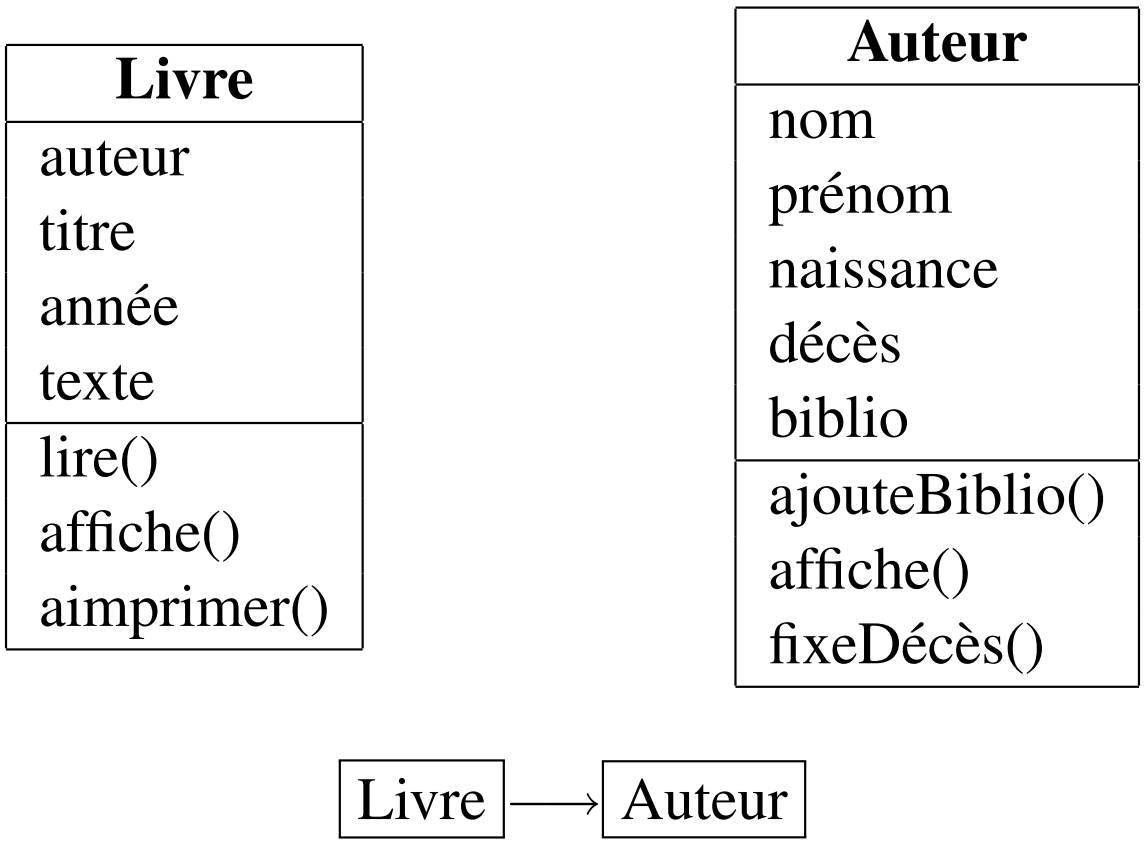
$\Longrightarrow$ on peut en abstraire un patron (abstrait), un "moule", un modèle, etc

Le moule définit

- les attributs qui caractérisent l'état
- l'interface et sa réaction = le comportement $\Longrightarrow$ les méthodes

de tous les moulages qui en seront issus

"moulages = objets"


Livre
auteur
titre
année
texte
lire()
affiche()
aimprimer()
Auteur
nom
prénom
naissance
décès
biblio
ajouteBiblio()
affiche()
fixeDécès()
Livre
Auteur

Classes et instances

Classe

un moule est appelé **classe** (= **type**)

Instance

les moulages sont les objets appelés **instances** de la classe

!!! moulage ne signifie pas copie (on peut les "décorer" différemment) !!!

Une fois qu'on a le moule/classe, on peut potentiellement **créer** autant de moulages/instances que l'on veut :

- identités différentes
- états définis par les mêmes attributs, mais valeurs différentes possibles
- toutes les instances auront le même comportement (mêmes méthodes)

Classes et instances

Classe

un moule est appelé **classe** (= **type**)

Instance

les moulages sont les objets appelés **instances** de la classe

!!! moulage ne signifie pas copie (on peut les "décorer" différemment) !!!

Une fois qu'on a le moule/classe, on peut potentiellement **créer** autant de moulages/instances que l'on veut :

- identités différentes
- états définis par les mêmes attributs, mais valeurs différentes possibles
- toutes les instances auront le même comportement (mêmes méthodes)

Classes et instances

Classe

un moule est appelé **classe** (= **type**)

Instance

les moulages sont les objets appelés **instances** de la classe

!!! moulage ne signifie pas copie (on peut les "décorer" différemment) !!!

Une fois qu'on a le moule/classe, on peut potentiellement **créer** autant de moulages/instances que l'on veut :

- identités différentes
- états définis par les mêmes attributs, mais valeurs différentes possibles
- toutes les instances auront le même comportement (mêmes méthodes)

Classes et instances

Classe

un moule est appelé **classe** (= **type**)

Instance

les moulages sont les objets appelés **instances** de la classe

!!! moulage ne signifie pas copie (on peut les "décorer" différemment) !!!

Une fois qu'on a le moule/classe, on peut potentiellement **créer** autant de moulages/instances que l'on veut :

- identités différentes
- états définis par les mêmes attributs, mais valeurs différentes possibles
- toutes les instances auront le même comportement (mêmes méthodes)

classe = patron/moule

- décrit la structure de l'état (les attributs et leurs types)
- définit les envois de messages acceptés par l'objet
  $\Longrightarrow$ "interface"

instance = objet obéissant à un patron

- état correspond à la structure
  $\hookrightarrow$ association de valeurs aux attributs
- ne répond qu'aux envois de messages autorisés par la classe
  $\hookrightarrow$ interface = ensemble des messages acceptés par l'objet

classe = patron/moule

- décrit la structure de l'état (les attributs et leurs types)
- définit les envois de messages acceptés par l'objet
  $\Longrightarrow$ "interface"

instance = objet obéissant à un patron

- état correspond à la structure
  $\hookrightarrow$ association de valeurs aux attributs
- ne répond qu'aux envois de messages autorisés par la classe
  $\hookrightarrow$ interface = ensemble des messages acceptés par l'objet

classe : *abstrait*

la notion/le type "`Chien`"

personne n'a jamais vu "le moule/type `Chien`"

instance : *concret*

"ce chien noir que je vois dans la rue", "le chien de mon voisin"

classe : *abstrait*

la notion/le type "`Chien`"

personne n'a jamais vu "le moule/type `Chien`"

instance : *concret*

"ce chien noir que je vois dans la rue", "le chien de mon voisin"

programmation définition des classes $\Longrightarrow$ abstraction

à l'exécution travail sur des objets/instances $\Longrightarrow$ concrétisation

- La classe définit le comportement de **toutes** ses instances
  Les instances ont des identités différentes et des valeurs d'attribut différentes.

Interface d'une classe

= ensemble des messages acceptés par les instances de la classe
≡ ensemble des signatures des méthodes publiques (généralement)

- il faut créer les instances selon le modèle de la classe pour concrétiser les entités permettant la résolution du problème

utilisation de **constructeurs**

programmation définition des classes $\Longrightarrow$ abstraction

à l'exécution travail sur des objets/instances $\Longrightarrow$ concrétisation

- La classe définit le comportement de **toutes** ses instances
  Les instances ont des identités différentes et des valeurs d'attribut différentes.

Interface d'une classe

= ensemble des messages acceptés par les instances de la classe
≡ ensemble des signatures des méthodes publiques (généralement)

- il faut créer les instances selon le modèle de la classe pour concrétiser les entités permettant la résolution du problème

utilisation de **constructeurs**

programmation définition des classes $\Longrightarrow$ abstraction

à l'exécution travail sur des objets/instances $\Longrightarrow$ concrétisation

- La classe définit le comportement de **toutes** ses instances
  Les instances ont des identités différentes et des valeurs d'attribut différentes.

Interface d'une classe

= ensemble des messages acceptés par les instances de la classe
$\equiv$ ensemble des signatures des méthodes publiques (généralement)

- il faut **créer** les instances selon le modèle de la classe pour concrétiser les entités permettant la résolution du problème

utilisation de **constructeurs**

programmation définition des classes $\Longrightarrow$ abstraction

à l'exécution travail sur des objets/instances $\Longrightarrow$ concrétisation

- La classe définit le comportement de **toutes** ses instances
  Les instances ont des identités différentes et des valeurs d'attribut différentes.

Interface d'une classe

= ensemble des messages acceptés par les instances de la classe
$\equiv$ ensemble des signatures des méthodes publiques (généralement)

- il faut **créer** les instances selon le modèle de la classe pour concrétiser les entités permettant la résolution du problème

utilisation de **constructeurs**

programmation définition des classes $\Longrightarrow$ abstraction

à l'exécution travail sur des objets/instances $\Longrightarrow$ concrétisation

- La classe définit le comportement de **toutes** ses instances
  Les instances ont des identités différentes et des valeurs d'attribut différentes.

Interface d'une classe

= ensemble des messages acceptés par les instances de la classe
$\equiv$ ensemble des signatures des méthodes publiques (généralement)

- il faut **créer** les instances selon le modèle de la classe pour concrétiser les entités permettant la résolution du problème

utilisation de **constructeurs**

Constructeurs

Chaque appel à un constructeur crée un nouvel objet (instance) qui obéit au patron défini par la classe :

- l'instance créée aura les attributs et le comportement définis dans la classe
  ↪ réservation d'un espace mémoire pour la mémorisation de l'état

- le constructeur est généralement l'occasion d'initialiser les attributs ("personnaliser" l'état de l'instance)
- il peut y avoir plusieurs constructeurs pour une même classe
  ↪ plusieurs initialisations

Constructeurs

Chaque appel à un constructeur crée un nouvel objet (instance) qui obéit au patron défini par la classe :

- l'instance créée aura les attributs et le comportement définis dans la classe
  ↪ réservation d'un espace mémoire pour la mémorisation de l'état

- le constructeur est généralement l'occasion d'initialiser les attributs ("personnaliser" l'état de l'instance)
- il peut y avoir plusieurs constructeurs pour une même classe
  ↪ plusieurs initialisations

en Java

Construction en JAVA

new + nom de la classe (+ param)

exemple : `new Livre()`

`new Livre("JRR Tolkien", "Le Seigneur des Anneaux", 1954)`

- il y a un constructeur par défaut (constructeur sans argument)
  ↪ il n'existe **que** si il n'y a pas d'autres constructeurs

en Java

Construction en JAVA

new + nom de la classe (+ param)

exemple : `new Livre()`

```
new Livre("JRR Tolkien", "Le Seigneur des Anneaux", 1954)
```

- il y a un constructeur par défaut (constructeur sans argument)
  ↪ il n'existe **que** si il n'y a pas d'autres constructeurs

en Java

Construction en JAVA

new + nom de la classe (+ param)

exemple : `new Livre()`

```
new Livre("JRR Tolkien", "Le Seigneur des Anneaux", 1954)
```

- il y a un constructeur par défaut (constructeur sans argument)
  ↪ il n'existe **que** si il n'y a pas d'autres constructeurs

Ok... mais ça donne quoi concrètement ?

```
public class Livre {
  // les attributs de la classe livre
  private String auteur;
  private String titre;
  private int annee;
  private String texte;
  // constructeur
  public Livre(String unAuteur, String titre, int annee, String texte) {
    this.auteur = unAuteur;
    this.titre = titre;
    this.annee = annee;
    this.texte = texte;
  }
  // les méthodes de la classe Livre
  public String getAuteur() {
    return this.auteur;
  }
  public void affiche(String msg) {
    System.out.println(msg+" -> "+this.titre+" de "+this.auteur+" paru en "+this.annee);
  }
  public void lit() {
    System.out.println(this.texte);
  }
  public void litEtAffiche() {
    this.lit();                    // invocation de lit() sur l'objet lui-même
    this.affiche("info :");
  }
}
```

Déclaration de classe

```
public class NomDeClasse {
    ...
    déclarations des constructeurs, attributs et méthodes
    ...
}
```

- un fichier/une classe (publique)
- le nom de la classe et du fichier doive correspondre
  ↪ classe `MachineChose` ⟹ fichier `MachineChose.java`
  ici `Livre.java`
- on définit la classe (= le moule)
- l'ordre des déclarations dans le fichier importe peu
- convention de nommage/d'écriture de code
- différenciation majuscules/minuscules

Déclaration de classe

```
public class NomDeClasse {
    ...
    déclarations des constructeurs, attributs et méthodes
    ...
}
```

- un fichier/une classe (publique)
- le nom de la classe et du fichier doive correspondre
  ↪ classe `MachineChose` ⟹ fichier `MachineChose.java`
  ici `Livre.java`
- on définit la classe (= le moule)
- l'ordre des déclarations dans le fichier importe peu
- convention de nommage/d'écriture de code
- différenciation majuscules/minuscules

Déclaration de classe

```
public class NomDeClasse {
    ...
    déclarations des constructeurs, attributs et méthodes
    ...
}
```

- un fichier/une classe (publique)
- le nom de la classe et du fichier doive correspondre
  ↪ classe `MachineChose` $\Longrightarrow$ fichier `MachineChose.java`
  ici `Livre.java`
- on définit la classe (= le moule)
- l'ordre des déclarations dans le fichier importe peu
- convention de nommage/d'écriture de code
- différenciation majuscules/minuscules

Déclaration de classe

```
public class NomDeClasse {
    ...
    déclarations des constructeurs, attributs et méthodes
    ...
}
```

- un fichier/une classe (publique)
- le nom de la classe et du fichier doive correspondre
  ↪ classe `MachineChose` $\Longrightarrow$ fichier `MachineChose.java`
  ici `Livre.java`
- on définit la classe (= le moule)
- l'ordre des déclarations dans le fichier importe peu
- convention de nommage/d'écriture de code
- différenciation majuscules/minuscules

Déclaration de classe

```
public class NomDeClasse {
    ...
    déclarations des constructeurs, attributs et méthodes
    ...
}
```

- un fichier/une classe (publique)
- le nom de la classe et du fichier doive correspondre
  ↪ classe `MachineChose` $\Longrightarrow$ fichier `MachineChose.java`
  ici `Livre.java`
- on définit la classe (= le moule)
- l'ordre des déclarations dans le fichier importe peu
- convention de nommage/d'écriture de code
- différenciation majuscules/minuscules

Déclaration de classe

```
public class NomDeClasse {
    ...
    déclarations des constructeurs, attributs et méthodes
    ...
}
```

- un fichier/une classe (publique)
- le nom de la classe et du fichier doive correspondre
  ↪ classe `MachineChose` $\Longrightarrow$ fichier `MachineChose.java`
  ici `Livre.java`
- on définit la classe (= le moule)
- l'ordre des déclarations dans le fichier importe peu
- convention de nommage/d'écriture de code
- différenciation majuscules/minuscules

Déclaration de classe

```
public class NomDeClasse {
    ...
    déclarations des constructeurs, attributs et méthodes
    ...
}
```

- un fichier/une classe (publique)
- le nom de la classe et du fichier doive correspondre
  ↪ classe `MachineChose` $\Longrightarrow$ fichier `MachineChose.java`
  ici `Livre.java`
- on définit la classe (= le moule)
- l'ordre des déclarations dans le fichier importe peu
- convention de nommage/d'écriture de code
- différenciation majuscules/minuscules

Déclaration de méthode

```
modificateur type nom([params*]) {
   ...  traitement ...
   [return expression;]              // si type ≠ void
}
```

- `type` = type primitif ou classe de la valeur de retour, si aucune : `void`
- la valeur de retour est précisée par `return` (fin de traitement)
- dans le code de la méthode, `this` désigne l'objet sur lequel la méthode a été invoquée.

Signature

Signature de la méthode = "entête" ($\sim$ interface de la méthode)

Déclaration de méthode

```
modificateur type nom([params*]) {
   ...  traitement ...
   [return expression;]              // si type ≠ void
}
```

- `type` = type primitif ou classe de la valeur de retour, si aucune : `void`
- la valeur de retour est précisée par `return` (fin de traitement)
- dans le code de la méthode, `this` désigne l'objet sur lequel la méthode a été invoquée.

Signature

Signature de la méthode = "entête" ($\sim$ interface de la méthode)

Déclaration de méthode

```
modificateur type nom([params*]) {
   ...  traitement ...
   [return expression;]              // si type ≠ void
}
```

- `type` = type primitif ou classe de la valeur de retour, si aucune : `void`
- la valeur de retour est précisée par `return` (fin de traitement)
- dans le code de la méthode, `this` désigne l'objet sur lequel la méthode a été invoquée.

Signature

Signature de la méthode = "entête" ($\sim$ interface de la méthode)

Déclaration de méthode

```
modificateur type nom([params*]) {
   ...  traitement ...
   [return expression;]              // si type ≠ void
}
```

- `type` = type primitif ou classe de la valeur de retour, si aucune : `void`
- la valeur de retour est précisée par `return` (fin de traitement)
- dans le code de la méthode, `this` désigne l'objet sur lequel la méthode a été invoquée.

Signature

Signature de la méthode = "entête" ($\sim$ interface de la méthode)

Déclaration de méthode

```
modificateur type nom([params*]) {
   ...  traitement ...
   [return expression;]              // si type ≠ void
}
```

- `type` = type primitif ou classe de la valeur de retour, si aucune : `void`
- la valeur de retour est précisée par `return` (fin de traitement)
- dans le code de la méthode, `this` désigne l'objet sur lequel la méthode a été invoquée.

Signature

Signature de la méthode = "entête" ($\sim$ interface de la méthode)

Analyse (objet) d'un problème

- Quels sont les objets ?
  $\Longrightarrow$ décomposition du pb en objets
- A quels modèles ces objets correspondent ils ?
  et donc quelles sont les classes ?
- Quelles sont les structures des états des objets
- Quelles sont les fonctionnalités dont on veut pouvoir disposer ?
  $\Longrightarrow$ Quels messages doit/veut on pouvoir envoyer aux objets ?

Analyse (objet) d'un problème

- Quels sont les objets ?
  $\Longrightarrow$ décomposition du pb en objets
- A quels modèles ces objets correspondent ils ?
  et donc quelles sont les classes ?
- Quelles sont les structures des états des objets
- Quelles sont les fonctionnalités dont on veut pouvoir disposer ?
  $\Longrightarrow$ Quels messages doit/veut on pouvoir envoyer aux objets ?

Analyse (objet) d'un problème

- Quels sont les objets ?
  $\Longrightarrow$ décomposition du pb en objets
- A quels modèles ces objets correspondent ils ?
  et donc quelles sont les classes ?
- Quelles sont les structures des états des objets
- Quelles sont les fonctionnalités dont on veut pouvoir disposer ?
  $\Longrightarrow$ Quels messages doit/veut on pouvoir envoyer aux objets ?

Analyse (objet) d'un problème

- Quels sont les objets ?
  $\Longrightarrow$ décomposition du pb en objets
- A quels modèles ces objets correspondent ils ?
  et donc quelles sont les classes ?
- Quelles sont les structures des états des objets
- Quelles sont les fonctionnalités dont on veut pouvoir disposer ?
  $\Longrightarrow$ Quels messages doit/veut on pouvoir envoyer aux objets ?

Déclaration

- Il est possible de nommer un objet créé pour pouvoir y faire **référence** par la suite.
  - on précise le type (classe) de la référence (donc de l'objet référencé)
  - on nomme la référence
  - on affecte une valeur (existante ou résultante d'une construction) = l'objet

- Destructeur d'objet ?
  - but : libérer l'espace mémoire occupé par l'objet
    en JAVA pas de destructeur d'objet explicite
    $\Longrightarrow$ pas nécessaire de libérer la mémoire "à la main" le GC s'en charge

(GC = Garbage Collector = "rammasse-miettes")

Déclaration

- Il est possible de nommer un objet créé pour pouvoir y faire **référence** par la suite.
  - on précise le type (classe) de la référence (donc de l'objet référencé)
  - on nomme la référence
  - on affecte une valeur (existante ou résultante d'une construction) = l'objet

- Destructeur d'objet ?
  - but : libérer l'espace mémoire occupé par l'objet
    en JAVA pas de destructeur d'objet explicite
    $\Longrightarrow$ pas nécessaire de libérer la mémoire "à la main" le GC s'en charge

(GC = Garbage Collector = "rammasse-miettes")

Déclaration

- Il est possible de nommer un objet créé pour pouvoir y faire **référence** par la suite.
  - on précise le type (classe) de la référence (donc de l'objet référencé)
  - on nomme la référence
  - on affecte une valeur (existante ou résultante d'une construction) = l'objet

- Destructeur d'objet ?
  - but : libérer l'espace mémoire occupé par l'objet
    en JAVA pas de destructeur d'objet explicite
    $\Longrightarrow$ pas nécessaire de libérer la mémoire "à la main" le GC s'en charge

(GC = Garbage Collector = "rammasse-miettes")

Exemples

en JAVA :

```
Auteur unAuteur = new Auteur();
Livre unLivre = new Livre("JRR Tolkien",
                          "Le Seigneur des Anneaux", 1954);
```

Remarques :

- conventions d'écriture (majuscules/minuscules)
- instructions se terminent par un ";"

Référence

- on fait **référence** à un objet en utilisant son identificateur

- tout identificateur doit être initialisé avant d'être utilisé
  ↪ il faut lier l'identificateur à une référence.
- un identificateur non initialisé a la valeur **null**

- un identificateur est unique (correspond à un seul objet)
- deux identificateurs peuvent faire référence au même objet

Référence

- on fait **référence** à un objet en utilisant son identificateur
- tout identificateur doit être initialisé avant d'être utilisé
  ↪ il faut lier l'identificateur à une référence.
- un identificateur non initialisé a la valeur **null**
- un identificateur est unique (correspond à un seul objet)
- deux identificateurs peuvent faire référence au même objet

Référence

- on fait **référence** à un objet en utilisant son identificateur
- tout identificateur doit être initialisé avant d'être utilisé
  ↪ il faut lier l'identificateur à une référence.
- un identificateur non initialisé a la valeur **null**
- un identificateur est unique (correspond à un seul objet)
- deux identificateurs peuvent faire référence au même objet

Référence

- on fait **référence** à un objet en utilisant son identificateur
- tout identificateur doit être initialisé avant d'être utilisé
  ↪ il faut lier l'identificateur à une référence.
- un identificateur non initialisé a la valeur **null**
- un identificateur est unique (correspond à un seul objet)
- deux identificateurs peuvent faire référence au même objet

Référence

- on fait **référence** à un objet en utilisant son identificateur

- tout identificateur doit être initialisé avant d'être utilisé
  ↪ il faut lier l'identificateur à une référence.
- un identificateur non initialisé a la valeur **null**

- un identificateur est unique (correspond à un seul objet)
- deux identificateurs peuvent faire référence au même objet

La notation "."

- dès que l'on possède une référence sur un objet, on peut envoyer un message à cet objet

notation "." (->)

`objet.message`

`objet.attribut` envoi du message "accès à l'attribut `attribut`" à `objet`

`objet.methode([params*])` envoi à `objet` le message "exécute la méthode `methode` avec les paramètres `params`"
↪ le traitement décrit dans le corps de la méthode est exécuté.

- il faut évidemment que ce message soit accepté/reconnu par l'objet
  $\Longrightarrow$ message appartient à l'interface de la classe de l'objet

```
Livre unLivre = new Livre(...)
new TV().on()
unLivre.auteur = "Tolkien"
unLivre.imprime()
```

les "cascades" sont possibles :

$\underbrace{\texttt{livre.auteur}}$`.nom`

un objet `Auteur`

$\overbrace{\texttt{livre.auteur}}$`.fixeDeces()`

```
Livre unLivre;
```

`unLivre.auteur` $\Longrightarrow$ **erreur** : référence non initialisée

- La dynamique d'un programme (le "*traitement*") se fait par une succession de constructions d'objets et d'envois de messages à ces objets.

- Un envoi de message se fait **toujours** sur un objet (instance).

- Toujours se poser les questions :
  - Quel est l'objet à qui ce message est envoyé ?
  - Ai-je le droit de lui envoyer ce message ?
    $\hookrightarrow$ sa définition (classe) accepte t-elle ce message ?

Cas de l'auto-référence

- Dans le traitement de l'une de ses méthodes un objet peut avoir à s'envoyer un message (pour accéder à un de ses attributs ou invoquer une des ses méthodes)
- utilisation de l'**auto-référence**, en JAVA : **this**

Cas de l'auto-référence

- Dans le traitement de l'une de ses méthodes un objet peut avoir à s'envoyer un message (pour accéder à un de ses attributs ou invoquer une des ses méthodes)
- utilisation de l'**auto-référence**, en JAVA : **this**
- exemple : on se place dans le corps d'une méthode de la classe `Livre`
  $\Longrightarrow$ lors du traitement l'objet invoquant est une instance de `Livre`

  `this.imprime()` signifie *"envoyer à `this` (= moi-même) le message `imprime()`"*
- Si pas d'ambigüité, `this` peut être omis :

  `imprime()` $\equiv$ `this.imprime()`

  `auteur` $\equiv$ `this.auteur`

  - `this` ne peut être utilisé que dans une méthode
  - `this` = référence de l'objet qui invoque la méthode

Cas de l'auto-référence

- Dans le traitement de l'une de ses méthodes un objet peut avoir à s'envoyer un message (pour accéder à un de ses attributs ou invoquer une des ses méthodes)
- utilisation de l'**auto-référence**, en JAVA : **this**

- exemple : on se place dans le corps d'une méthode de la classe `Livre` $\Longrightarrow$ lors du traitement l'objet invoquant est une instance de `Livre`

  `this.imprime()` signifie *"envoyer à `this` (= moi-même) le message `imprime()`"*

- Si pas d'ambigüité, `this` peut être omis :

  `imprime()` $\equiv$ `this.imprime()`

  `auteur` $\equiv$ `this.auteur`

  - `this` ne peut être utilisé que dans une méthode
  - `this` = référence de l'objet qui invoque la méthode

Cas de l'auto-référence

- Dans le traitement de l'une de ses méthodes un objet peut avoir à s'envoyer un message (pour accéder à un de ses attributs ou invoquer une des ses méthodes)
- utilisation de l'**auto-référence**, en JAVA : **this**

- exemple : on se place dans le corps d'une méthode de la classe `Livre`
  $\Longrightarrow$ lors du traitement l'objet invoquant est une instance de `Livre`

  `this.imprime()` signifie "*envoyer à* `this` *(= moi-même) le message* `imprime()`"

- Si pas d'ambigüité, `this` peut être omis :

  `imprime()` $\equiv$ `this.imprime()`

  `auteur` $\equiv$ `this.auteur`

  - `this` ne peut être utilisé que dans une méthode
  - `this` = référence de l'objet qui invoque la méthode

Cas de l'auto-référence

```
public class Livre {
 // les attributs de la classe livre
  private String auteur;
  private String titre;
  private int annee;
  private String texte ="";
 // constructeur
  public Livre(String unAuteur, String titre, int annee, String texte) {
   this.auteur = unAuteur;
   this.titre = titre;
   this.annee = annee;
   this.texte = texte;
  }
 // les méthodes de la classe Livre
  public String getAuteur() {
   return this.auteur;
  }
  public void affiche(String msg) {
   System.out.println(msg+" -> "+this.titre+" de "+this.auteur+" paru en "+this.annee);
  }
  public void lit() {
   System.out.println(this.texte);
  }
  public void litEtAffiche() {
   this.lit();                    envoi du messsage lit() à lui-même
   this.affiche("info :");        envoi du messsage affiche("info :") à lui-même

  }
}
```

Autres remarques

- conventions d'écriture et de nommage
  (cf. `http://java.sun.com/docs/codeconv/index.html`)

  > *Why Have Code Conventions ? Code conventions are important to programmers for a number of reasons: 80% of the lifetime cost of a piece of software goes to maintenance. Hardly any software is maintained for its whole life by the original author. Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly. If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.*

- les commentaires : `//...` et aussi `/* ... */`

Autres remarques

- conventions d'écriture et de nommage
  (cf. `http://java.sun.com/docs/codeconv/index.html`)

  > *Why Have Code Conventions ? Code conventions are important to programmers for a number of reasons: 80% of the lifetime cost of a piece of software goes to maintenance. Hardly any software is maintained for its whole life by the original author. Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly. If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.*

- les commentaires : `//...` et aussi `/* ... */`

Penser "objet"

- Les instances de la classe `Livre` possède un attribut `auteur`.
- Cet attribut ne devrait pas être du type `String`
  (pourquoi un *auteur* serait une chaîne de caractères ?),
  mais d'une classe (type) plus appropriée : `Auteur` ou `Personne`.
- construction d'objet par composition/agrégation d'objets

```
public class Auteur {              // dans Auteur.java
  private String nom;
  private String prenom;
  public Auteur(String nom, String prenom) { ...  }
  ...
}

public class Livre {               // dans Livre.java
  // les attributs de la classe livre
  private Auteur auteur;
  private String titre;
  // constructeur
  public Livre(Auteur unAuteur, String titre) {
    this.auteur = unAuteur;
    this.titre = titre;
  }
  // les méthodes de la classe Livre
  public Auteur getAuteur() { return this.auteur; }
  public void setAuteur(Auteur auteur) { this.auteur = auteur; }
}

-------------------------------------------------------------------------

Auteur unAuteur = new Auteur("Frank","Herbert");
Livre unLivre = new Livre(unAuteur,"Dune");
Livre leLivre = new Livre(new Auteur("J.R.R.","Tolkien"),"Le Seigneur...");

System.out.println(leLivre.getAuteur().getNom());
```

Classe existante :

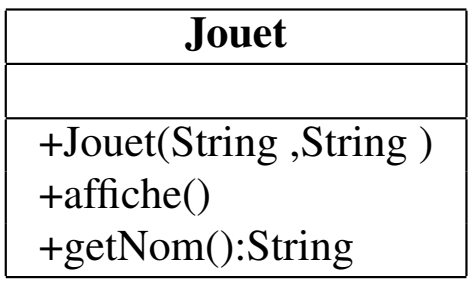
Jouet
+Jouet(String ,String) +affiche() +getNom():String

```
public class Usine {                    // dans Usine.java
   // constructeurs de la classe Usine
   public Usine() {
      this.marque = "generique";    // valeur par défaut
   }
   public Usine(String marque) {    // autre constructeur
      this.marque = marque;
   }

   // l'attribut de la classe Usine
   private String marque;

   // la méthode de la classe Usine
   public Jouet fabrique(String nom) {
      Jouet unJouet = new Jouet(this.marque, nom);
      unJouet.affiche();
      return unJouet;
   }
}
```

Classe existante :




```
public class Usine {                    // dans Usine.java
   // constructeurs de la classe Usine
   public Usine() {
      this.marque = "generique";    // valeur par défaut
   }
   public Usine(String marque) {    // autre constructeur
      this.marque = marque;
   }

   // l'attribut de la classe Usine
   private String marque;

   // la méthode de la classe Usine
   public Jouet fabrique(String nom) {
      Jouet unJouet = new Jouet(this.marque, nom);
      unJouet.affiche();
      return unJouet;
   }
}
```

Remarques

- plusieurs constructeurs
- utilisation du `this` (levée d'ambigüité)
- valeur par défaut possible pour les attributs
- méthode avec paramètre
- création d'un nouvel objet lors d'un traitement
  ↪ c'est comme cela que la dynamique du programme va se faire !
- dès que l'on possède la référence d'un objet on peut utiliser son interface publique (cf. `unJouet`)
- on peut "retourner" la référence d'un objet comme résultat (`fabrique`)
- on ne connaît que l'interface (publique) des instances de la classe `Jouet` (aucune idée de l'implémentation)
  $\Longrightarrow$ on peut l'utiliser sans en connaître l'implémentation
- `marque` de type `String` ? pourquoi ?

Remarques

- plusieurs constructeurs
- utilisation du `this` (levée d'ambigüité)
- valeur par défaut possible pour les attributs
- méthode avec paramètre
- création d'un nouvel objet lors d'un traitement
  $\hookrightarrow$ c'est comme cela que la dynamique du programme va se faire !
- dès que l'on possède la référence d'un objet on peut utiliser son interface publique (cf. `unJouet`)
- on peut "retourner" la référence d'un objet comme résultat (`fabrique`)
- on ne connaît que l'interface (publique) des instances de la classe `Jouet` (aucune idée de l'implémentation)
  $\Longrightarrow$ on peut l'utiliser sans en connaître l'implémentation
- `marque` de type `String` ? pourquoi ?

Remarques

- plusieurs constructeurs
- utilisation du `this` (levée d'ambigüité)
- valeur par défaut possible pour les attributs
- méthode avec paramètre
- création d'un nouvel objet lors d'un traitement
  ↪ c'est comme cela que la dynamique du programme va se faire !
- dès que l'on possède la référence d'un objet on peut utiliser son interface publique (cf. `unJouet`)
- on peut "retourner" la référence d'un objet comme résultat (`fabrique`)
- on ne connaît que l'interface (publique) des instances de la classe `Jouet` (aucune idée de l'implémentation)
  $\Longrightarrow$ on peut l'utiliser sans en connaître l'implémentation
- `marque` de type `String` ? pourquoi ?

Remarques

- plusieurs constructeurs
- utilisation du `this` (levée d'ambigüité)
- valeur par défaut possible pour les attributs
- méthode avec paramètre
- création d'un nouvel objet lors d'un traitement
  ↪ c'est comme cela que la dynamique du programme va se faire !
- dès que l'on possède la référence d'un objet on peut utiliser son interface publique (cf. `unJouet`)
- on peut "retourner" la référence d'un objet comme résultat (`fabrique`)
- on ne connaît que l'interface (publique) des instances de la classe `Jouet` (aucune idée de l'implémentation)
  ⟹ on peut l'utiliser sans en connaître l'implémentation
- `marque` de type `String` ? pourquoi ?

Remarques

- plusieurs constructeurs
- utilisation du `this` (levée d'ambigüité)
- valeur par défaut possible pour les attributs
- méthode avec paramètre
- création d'un nouvel objet lors d'un traitement
  ↪ c'est comme cela que la dynamique du programme va se faire !
- dès que l'on possède la référence d'un objet on peut utiliser son interface publique (cf. `unJouet`)
- on peut "retourner" la référence d'un objet comme résultat (`fabrique`)
- on ne connaît que l'interface (publique) des instances de la classe `Jouet` (aucune idée de l'implémentation)
  $\Longrightarrow$ on peut l'utiliser sans en connaître l'implémentation
- `marque` de type `String` ? pourquoi ?

Remarques

- plusieurs constructeurs
- utilisation du `this` (levée d'ambigüité)
- valeur par défaut possible pour les attributs
- méthode avec paramètre
- création d'un nouvel objet lors d'un traitement
  ↪ c'est comme cela que la dynamique du programme va se faire !
- dès que l'on possède la référence d'un objet on peut utiliser son interface publique (cf. `unJouet`)
- on peut "retourner" la référence d'un objet comme résultat (`fabrique`)
- on ne connaît que l'interface (publique) des instances de la classe `Jouet` (aucune idée de l'implémentation)
  $\Longrightarrow$ on peut l'utiliser sans en connaître l'implémentation
- `marque` de type `String` ? pourquoi ?

Remarques

- plusieurs constructeurs
- utilisation du `this` (levée d'ambigüité)
- valeur par défaut possible pour les attributs
- méthode avec paramètre
- création d'un nouvel objet lors d'un traitement
  $\hookrightarrow$ c'est comme cela que la dynamique du programme va se faire !
- dès que l'on possède la référence d'un objet on peut utiliser son interface publique (cf. `unJouet`)
- on peut "retourner" la référence d'un objet comme résultat (`fabrique`)
- on ne connaît que l'interface (publique) des instances de la classe `Jouet` (aucune idée de l'implémentation)
  $\Longrightarrow$ on peut l'utiliser sans en connaître l'implémentation
- `marque` de type `String` ? pourquoi ?

Remarques

- plusieurs constructeurs
- utilisation du `this` (levée d'ambigüité)
- valeur par défaut possible pour les attributs
- méthode avec paramètre
- création d'un nouvel objet lors d'un traitement
  $\hookrightarrow$ c'est comme cela que la dynamique du programme va se faire !
- dès que l'on possède la référence d'un objet on peut utiliser son interface publique (cf. `unJouet`)
- on peut "retourner" la référence d'un objet comme résultat (`fabrique`)
- on ne connaît que l'interface (publique) des instances de la classe `Jouet` (aucune idée de l'implémentation)
  $\Longrightarrow$ on peut l'utiliser sans en connaître l'implémentation
- `marque` de type `String` ? pourquoi ?

Remarques

- plusieurs constructeurs
- utilisation du `this` (levée d'ambigüité)
- valeur par défaut possible pour les attributs
- méthode avec paramètre
- création d'un nouvel objet lors d'un traitement
  $\hookrightarrow$ c'est comme cela que la dynamique du programme va se faire !
- dès que l'on possède la référence d'un objet on peut utiliser son interface publique (cf. `unJouet`)
- on peut "retourner" la référence d'un objet comme résultat (`fabrique`)
- on ne connaît que l'interface (publique) des instances de la classe `Jouet` (aucune idée de l'implémentation)
  $\Longrightarrow$ on peut l'utiliser sans en connaître l'implémentation
- `marque` de type `String` ? pourquoi ?

Référence

identificateur d'objet = référence

(référence = pointeur)

```
String chaine;                    // "chaine" référence "null"
chaine = new String("Le Seigneur des Anneaux");
Livre id1Livre = new Livre();
Livre id2Livre = id1Livre;
```

Important

La référence permet d'accéder à l'objet, mais n'est pas l'objet lui-même.
Une *variable référence d'objet* contient l'information sur comment trouver l'objet.

cf. télécommande d'un téléviseur

- **déclaration** `String chaine`
  ↪ réservation d'un nom pour potentiellement un futur objet `chaine`
- **création** `new String("Le Seigneur des Anneaux")`
  ↪ création de l'objet grâce à un constructeur
  Le système (la JVM) possède un moyen de repérer l'objet.

- **liaison**

```
String chaine  =  new String("Le Seigneur des Anneaux");
                  ↑
```

le moyen de retrouver l'objet est stocké dans `chaine`

Quand l'identifiant `chaine` est utilisé, l'information stockée dans `chaine` sert à "trouver" l'objet :

```
System.out.println(chaine.length());
```

Rappel : chaque appel à `new` crée un nouvel objet

```
String chaine;
chaine = new String("Le Seigneur des Anneaux");
chaine = new String("Dune");
```

← nouvel objet créé, la référence de cet objet est placé dans **`chaine`**

le premier objet référencé est "perdu"
$\Longrightarrow$ si plus aucune (autre) référence sur lui :"GC"

Exploitation

```
Livre leLivre = new Livre("Tolkien", "Le Seigneur des Anneaux", 1954);
leLivre.affiche();
Livre unLivre;
unLivre = new Livre("Frank Herbert","Dune",1965);
System.out.println(leLivre.getAuteur());
System.out.println(unLivre.getAuteur());
System.out.println(new Livre(...).getAuteur());
```