Introduction : classes et objets

Programmation Orientée Objet

Jean-Christophe Routier
Licence mention Informatique
Université Lille 1

Préliminaire

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

Objectifs :

- présenter les concepts de base de l'approche objet de la programmation
  - adopter le "penser objet"
  - connaître et savoir mettre en œuvre les concepts fondamentaux
- préparer au cours de **Conception Orientée Objet** du S5

Programmer

1. analyse, étude du problème à modéliser : documentation et spécifications
2. conception : mise en place des solutions techniques
3. codage (construction du logiciel)
4. tests (recette)

Eléments à considérer :

maintenance – évolution – 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**.*

Paradigme de programmation

*Un **paradigme** de programmation est un style fondamental de programmation qui traite de la manière dont les solutions aux problèmes doivent être formulées dans un langage de programmation.* source Wikipédia

On peut programmer la même chose avec tous les langages. Ils ont tous le même pouvoir d'expressivité : **machines de Turing**

mais un langage donné est généralement plus particulièrement dédié à un paradigme de programmation

et certains problèmes sont plus faciles à programmer dans un langage que dans un autre

Programmation impérative et modulaire

Paradigme impératif

Un programme est une séquence d'instructions exécutées par l'ordinateur pour modifier son état.

instructions : *affectation, séquence, structures conditionnelles et itératives.*

Programmation modulaire

Un programme est décomposé en éléments plus simples dans le but de faciliter son développement et de permettre la réutilisabilité
Principe du *diviser pour régner*

éléments : *procédures, fonctions, modules, unités.*

Nous sommes entourés d'objets

- des voitures, des livres, des portes, des chaises, des ordinateurs, des thermomètres, des téléviseurs,...
- des chats, des personnes, des facteurs, des éléphants,...
- des comptes en banque, des dossiers étudiant, des connexions réseau,...

Ces objets

- ont des ***caractéristiques*** : la couleur d'une voiture, l'âge ou le nom d'une personne, le solde d'un compte en banque,...
- ont un ***comportement*** : ouvrir la porte, le chat miaule, créditer le compte en banque,...

Programmation objet

Paradigme objet

Un programme est un ensemble d'objets qui interagissent.

- reprend et prolonge la démarche modulaire : décomposition d'un problème en parties simples,
- la programmation des traitements reste impérative,
- plus intuitive car s'inspire du monde réel pour une modélisation "plus naturelle"
- facilite la réutilisation et la conception de "grandes" applications

L'approche objet s'est désormais imposée dans la construction logicielle avec de nombreux langages : Java, C#, Smalltalk, Python, php5, ...

Exemples

Un thermomètre mesure une température (un nombre), c'est une donnée/caractéristique de ce thermomètre, son **état**.

On peut envisager certaines opérations :

- connaître la température en degrés Celsius, en Farenheit
- modifier la température
- associer à la température une couleur (*bleu*, *vert*, *rouge*, ...) ou un symbôle (*froid*, *normal*, *chaud*,...)

A chaque opération correspond un traitement.

Quelques questions en suspens :

- la température est-elle un entier ou un flottant ?
- en degrés Celsius ou un en Farenheit ?
- à quelles températures correspondent *froid*, *chaud*, etc. ?

Un premier thermomètre

On peut donc disposer d'un objet thermomètre ⟶ soit *thermo1* son **identité**.

Son **état** est caractérisé par une température mesurée *temp* (22.5°C par exemple).

On peut exploiter son **comportement** :
`temperatureCelsius`, `temperatureFahrenheit`, `modifierTemperature`, `couleurTemperature`

```
thermo1.temperatureCelsius()        →  22.5
thermo1.temperatureFarenheit()      →  72.5
thermo1.modifierTemperature(25.8)   →  -
thermo1.temperatureCelsius()        →  25.8
thermo1.couleurTemperature()        →  Couleur.VERT
```

Le comportement dépend de l'état et agit sur l'état.

Le comportement correspond à un ensemble de traitements programmés en "style" impératif.

Par exemple :

```
temperatureCelsius
 données : ∅
 résultat : un flottant
 traitement :
     renvoyer temp

temperatureFahrenheit
 données : ∅
 résultat : un flottant
 traitement :
     renvoyer (9/5)* temp + 32
```

```
modifierTemperature
 données : flottant : nouvelleTemp
 résultat : ∅
 traitement :
     temp = nouvelleTemp

couleurTemperature
 données : ∅
 résultat : une couleur
 traitement :
     si (temp < 0) alors
        renvoyer Couleur.BLEU
     sinon si (temp < 30) alors
        renvoyer Couleur.VERT
     sinon
        renvoyer Couleur.ROUGE
```


Un autre thermomètre

On peut disposer d'un **autre** thermomètre ⟶ soit *thermo2* son **identité**.

Son **état** correspond également à sa température (-4°C par exemple).
La "structure" de l'état est la même mais la valeur est **différente**.

Il a le même **comportement**.

```
thermo2.temperatureCelsius()      →  -4
thermo2.temperatureFarenheit()    →  24.8
thermo2.couleurTemperature()      →  Couleur.bleu
```

Les résultats sont différents mais **les traitements** exécutés **sont les mêmes**.
Dans le traitement *temp* signifie "*la température du thermomètre impliqué*".

Il en serait de même pour **tous** les autres thermomètres.

Chaudière

Une chaudière est un objet dont

- l'état est caractérisé par le statut allumée/éteinte (un booléen)
- le comportement est défini par
  - connaître l'état de la chaudière : `estAllumee()`, `estEteinte()`
  - allumer la chaudière : `allumer()`
  - éteindre la chaudière : `eteindre()`

Le comportement `temperatureCelsius` n'a pas de sens pour un objet chaudière.

Thermostat

L'**état** d'un thermostat est caractérisé par

- la chaudière qu'il contrôle, *chaudiere*
- la température de déclenchement, *tempDeclenchement*
- un thermomètre pour mesurer la température ambiante, *thermo*.

Cet état **est composé d'autres objets** : *chaudiere*, *thermo*.

(et tempDeclenchement alors ?)

Son **comportement** est :

- mettre en marche ou arrêter la chaudière selon la température de déclenchement et celle mesurée par le thermomètre
- modifier la température de déclenchement,
- etc.

par exemple :

```
changerTempDeclenchement
  données : flottant : nouvelleTemp
  résultat : ∅
  traitement :
    tempDeclenchement = nouvelleTemp
    si (chaudiere.estAllumee() et
          tempDeclenchement < thermo.temperatureCelsius()) alors
       chaudiere.eteindre()
    sinon si (chaudiere.estEteinte() et
          tempDeclenchement ≥ thermo.temperatureCelsius()) alors
          chaudiere.allumer()
```

Le traitement de `changerTempDeclenchement` fait appel aux comportements des objets *chaudiere* et *thermo*.
L'objet thermostat **envoie des messages** à ces objets.

C'est ainsi que se crée la dynamique du programme.

Langage à objets (purs)

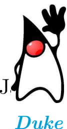
Alan Kay - SmallTalk

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

Java

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

- langage orienté objet (pas 100% objet), langage de classes
- langage compilé, fortement typé
- indépendance OS/architecture : multi plate-forme
  ↪ utilisation d'une machine virtuelle (la JVM) - bytecode J
  "*compile once, run everywhere*"



Duke

- gestion dynamique de la mémoire
  ↪ utilisation d'un GC (garbage collector = ramasse-miettes)
- gestion des erreurs par exceptions
- nombreuses bibliothèques/API (gratuites) (réseau, RMI, JDBC, etc.)

depuis 1995, libre depuis ~ 2007... actuellement "Java 6" – JDK, JRE, SDK

En Java

```java
public class Thermometre {

   private float temp;

   public Thermometre(float tempInit) {
      this.temp = tempInit;
   }

   public float temperatureCelsius() {
      return this.temp;
   }
   public float temperatureFahrenheit() {
      return (9.0/5.0)*this.temp+32;
   }
   public void modifierTemperature(float nouvelleTemp) {
      this.temp = nouvelleTemp;
   }
   public Color couleurTemperature() {
      if (this.temp < 0) {
         return Color.BLUE;
      }
      else if (this.temp < 30) {
         return Color.GREEN;
      }
      else return Color.RED;
   }
}
```


Objet

Objet

Objet = identité + état + comportement
attributs méthodes

avec

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

Une identité

Un objet forme un tout.

- permet de s'adresser à l'objet
- chaque identité est unique
  ↪ deux objets différents ont des identités différentes
- on peut faire référence à l'objet (à son identité), la nommer
- on peut avoir plusieurs références pour une seule identité (un seul objet)

Un état

- ensemble de propriétés ou caractéristiques définies par des valeurs
- valeurs propres (personnelles) à chaque objet
- l'état d'un objet (les valeurs des propriétés) 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 dit que l'on **invoque une méthode sur un objet**.

On ne peut utiliser une méthode qu'en l'invoquant sur un objet.

Envoi de message

- on s'adresse à l'objet par **envoi de messages**
  ↪ 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 attributs
    ↪ même interface de comportement

exemple :

- les thermomètres `thermo1` et `thermo2`
- des livres

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

sont caractérisés par les mêmes *attributs*
↪ auteur, titre, année, texte
et ont la même interface de *comportement*
↪ on peut leur faire accomplir les mêmes actions ↪ on peut les lire, les imprimer, etc.

il en serait de même pour (∼) tous les livres

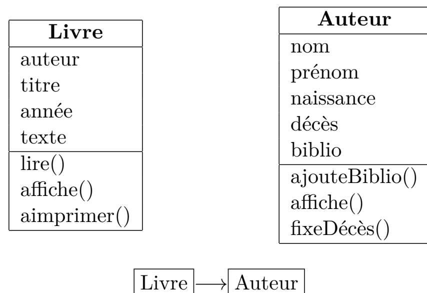
Tous les livres obéissent à un même schéma

⟹ on peut en abstraire un patron (abstrait), un "moule", un patron, un modèle, etc

Le moule définit
- les **attributs** qui caractérisent l'état
- l'interface et sa réaction = le comportement ⟹ les **méthodes**

de tous les moulages qui en seront issus

"moulages = objets"



Auteur
nom
prénom
naissance
décès
biblio
ajouteBiblio()
affiche()
fixeDécès()

Livre ⟶ Auteur

Classes et instances

Classe

Un moule est appelé classe. Une classe est un type.

Instance

Les moulages sont les objets appelés instances de la classe

Une fois qu'on a la classe, on peut potentiellement **créer** autant d'objets/instances conformes à la classe que l'on veut :

- ont des identités différentes
- ont des états définis par les **mêmes attributs**, mais avec des **valeurs différentes**
- auront le **même comportement** (mêmes **méthodes**)

classe = patron

- 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
  ↪ association de valeurs aux attributs
- ne répond qu'aux envois de messages autorisés par la classe
  ↪ interface = ensemble des messages acceptés par l'objet

classe : *abstrait*
la notion/le type "`Chien`"
personne n'a jamais vu "le 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)

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 :");
  }
}
```

Analyse (objet) d'un problème

- Quels sont les objets nécessaires à la résulution du problème ?
  ⟹ décomposition du problème en objets
- A quels modèles ces objets correspondent ils ?
  et donc quelles sont les classes ?
- Quelles sont les fonctionnalités dont on veut pouvoir disposer ?
  ⟹ Quel comportement ? càd quels messages doit/veut on pouvoir envoyer aux objets ?
- Quelle est la structure de l'état des objets

Constructeurs

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

utilisation de **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

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 ";"

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
    ⟹ pas nécessaire de libérer la mémoire "à la main" le GC s'en charge

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

Référence

identificateur d'objet = référence

Référence = un pointeur vers l'identité de l'objet

```
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());
```


Identificateur

- 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 correspond à un seul objet
- deux identificateurs peuvent faire référence au même objet

La notation "."

- si 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 :

`livre.auteur.nom` (sous `livre.auteur` : un objet `Auteur`)

`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 ?
    ↪ 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`

this

`this` = référence vers l'objet qui invoque la méthode

- `this` ne peut être utilisé que dans une méthode (n'a aucun sens ailleurs)
- exemple : on se place dans le corps d'une méthode de la classe `Livre`
  ⟹ 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()` ≡ `this.imprime()`

  `auteur` ≡ `this.auteur`

Un peu de syntaxe ?

```
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
- 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" (∼ interface de la méthode)

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 ;
   }
}
```


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 ?