Classes abstraites et interfaces en Java

Programmation orientée objet

IG2

RAPPELS SUR L'ORIENTÉ OBJET CLASSIQUE

Retour aux bases après un passage
par l'orienté objet prototypal de Javascript…

Rappels sur l'OO classique

L'orienté objet classique repose sur 3 concepts fondamentaux :

- **Encapsulation**
  On regroupe données et actions sur ces données.

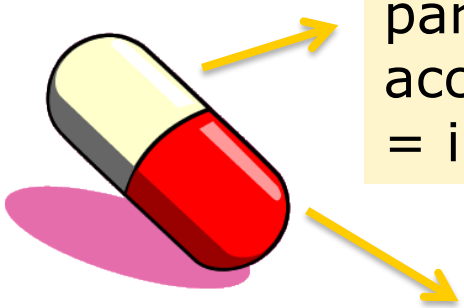
- **Héritage**
  On structure les classes selon une arborescence.

- **Polymorphisme (et liaison dynamique)**
  Le code exécuté est choisi selon le type à l'exécution.

Rappels sur l'OO classique

Qu'est-ce que l'**encapsulation** ?

- On regroupe données (attributs) et actions (méthodes).
- On gère la visibilité de chacune des propriétés.



partie « transparente » à laquelle on peut accéder depuis l'extérieur
= interface de l'objet, propriétés publiques

partie « opaque » non accessible depuis l'extérieur
= partie privée

Rappels sur l'OO classique

Vocabulaire lié à l'**encapsulation** :

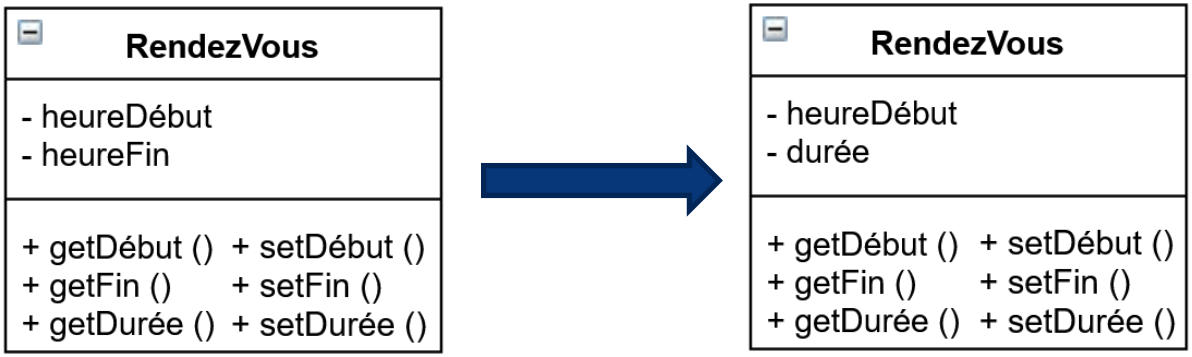
- **Getter / sélecteur / accesseur** : méthode qui permet d'obtenir la valeur d'un attribut logique
  - *Attribut logique : une donnée qui « pourrait » être un attribut (mais ne l'est pas forcément).*
  - *On peut donc avoir un getter qui renvoie autre chose que la valeur d'un attribut.*
  - *Chaque attribut n'a pas forcément besoin d'un getter !*
- **Setter / modificateur / mutateur** : méthode qui permet de modifier la valeur d'un attribut logique
  - *Mêmes remarques que ci-dessus.*
  - *Peut refuser la modification en cas de nouvelle valeur invalide.*

Rappels sur l'OO classique

Que permet l'**encapsulation** ?

- Découpe du code (et du travail)
- Découplage (= indépendance) du code

  Pour accéder aux éléments privés, il faut passer par des getters/setters. Cela permet de modifier l'implémentation interne de la classe SANS devoir réécrire le code qui utilise la classe.

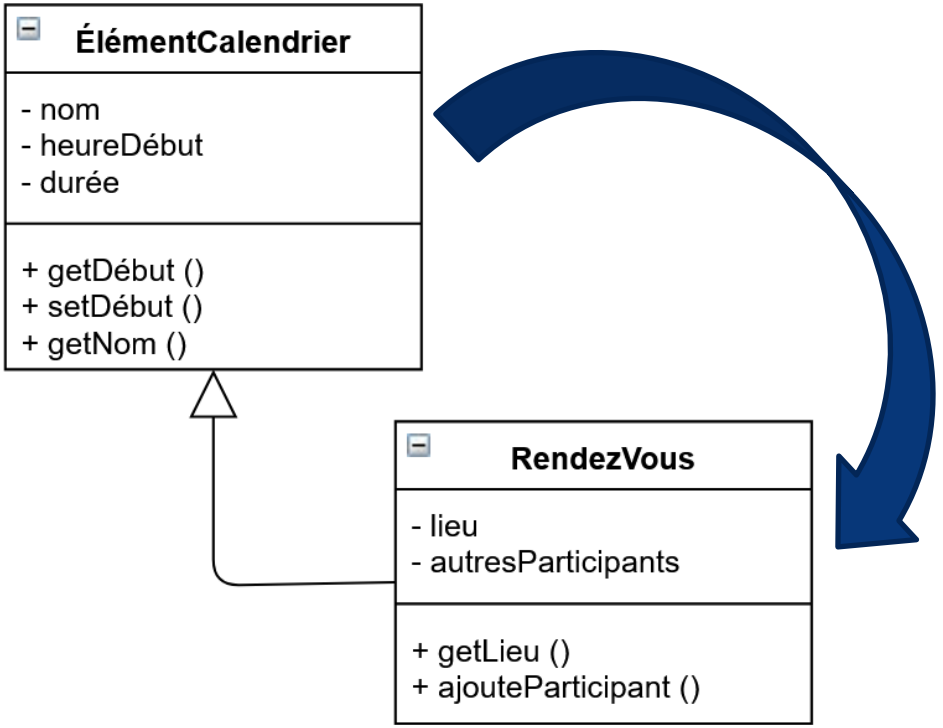


- Mettre en place des vérifications lors des modifications de données

Rappels sur l'OO classique

Qu'est-ce que l'**héritage** ?

- On organise les classes en arborescence.
- Chaque classe hérite de son parent (et de ses ancêtres).



RendezVous hérite des attributs et des méthodes de ÉlémentCalendrier

Rappels sur l'OO classique

Vocabulaire lié à l'**héritage** :

- Une sous-classe **étend** sa classe mère.
- La sous-classe hérite des méthodes de la classe mère mais peut aussi les **redéfinir** (overwrite).
- Processus de conception des classes :
  - du haut vers le bas = **spécialisation** (ajout des spécificités)
  - du bas vers le haut = **généralisation** (mise en commun)

- Voir aussi le concept d'**héritage multiple** (pas autorisé en Java).

Rappels sur l'OO classique

Que permet l'**héritage** ?

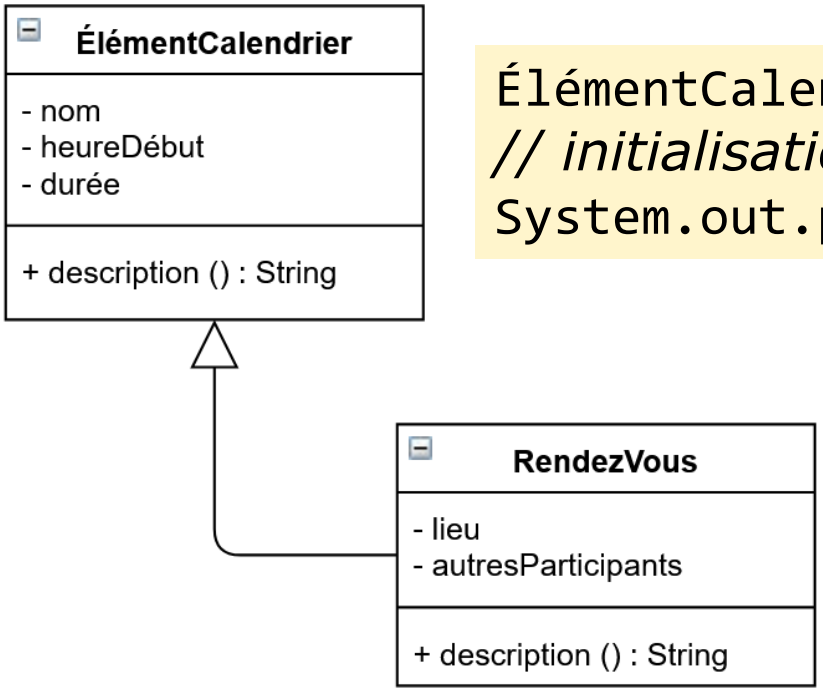
- Réutilisation du code (et point de modification unique)
  automatique ou via redéfinition (mot-clef super en Java)

- **Principe de substitution** : on peut utiliser un objet de la classe-fille partout où on attend un objet de la classe-mère
  - *en tant que paramètre de méthode :*
    `agenda.ajouter(élémentCalendrier)`
    peut être appelée avec un rendez-vous :
    `agenda.ajouter(rendezVous)`
  - *dans une variable typée par la classe mère :*
    `ÉlémentCalendrier ec = new RendezVous (…);`

Rappels sur l'OO classique

Qu'est-ce que le **polymorphisme** ?

- Une même syntaxe mais des sémantiques différentes.
- On utilise le code le plus approprié d'après l'héritage.




```
ÉlémentCalendrier ec;
// initialisation de ec
System.out.println(ec.description());
```

On utilisera le code correspondant au type du contenu de ec, déterminé lors de l'exécution.
= **liaison dynamique**

Rappels sur l'OO classique

Vocabulaire du **polymorphisme** / de la **liaison dynamique**

- Le mot **liaison** se réfère au lien entre l'appel de méthode écrit et le code à exécuter (voir « linkage » en C).
- Le mot **dynamique** indique que le lien est établi à la volée, au moment de l'exécution.

- <> **liaison statique** : on choisit l'implémentation lors de la compilation, une fois pour toutes !

Note : certains appels de méthodes sont résolus de manière statique en Java : les méthodes « private », « final » ou « static ».

Rappels sur l'OO classique

Que permet le **polymorphisme** et la **liaison dynamique** ?

- Traitement uniforme de collections (tableaux) d'objets :

```
ÉlémentCalendrier [] activités = agenda.…;
// peut mélanger des ÉlémentCalendrier
// et des RendezVous

for (ÉlémentCalendrier ec : activités)
  System.out.println(ec.description());
// appelle la bonne méthode pour chaque activité
```

EXEMPLE INTRODUCTIF

Exemple introductif

Une borne de renseignement interactive permet d'obtenir l'adresse des personnes encodées.

La borne est reliée à plusieurs périphériques de sortie : une imprimante laser, une imprimante jet d'encre, un terminal Braille, un écran et une fiche permettant de connecter une clef USB pour récupérer les informations dans un fichier.

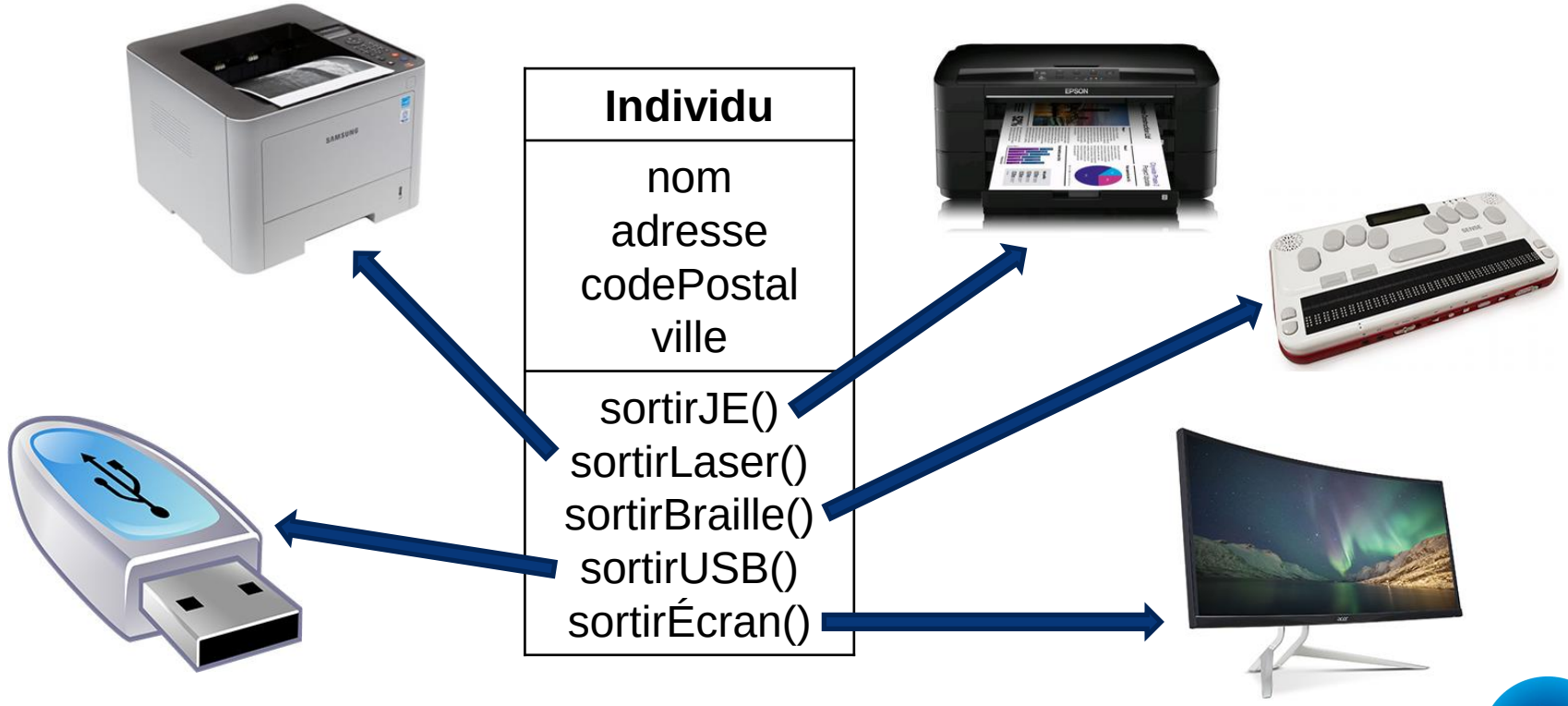
Individu
nom adresse codePostal ville

Les informations propres à un individu peuvent être sorties sur n'importe lequel de ces périphériques, au choix de l'utilisateur.

Exemple introductif

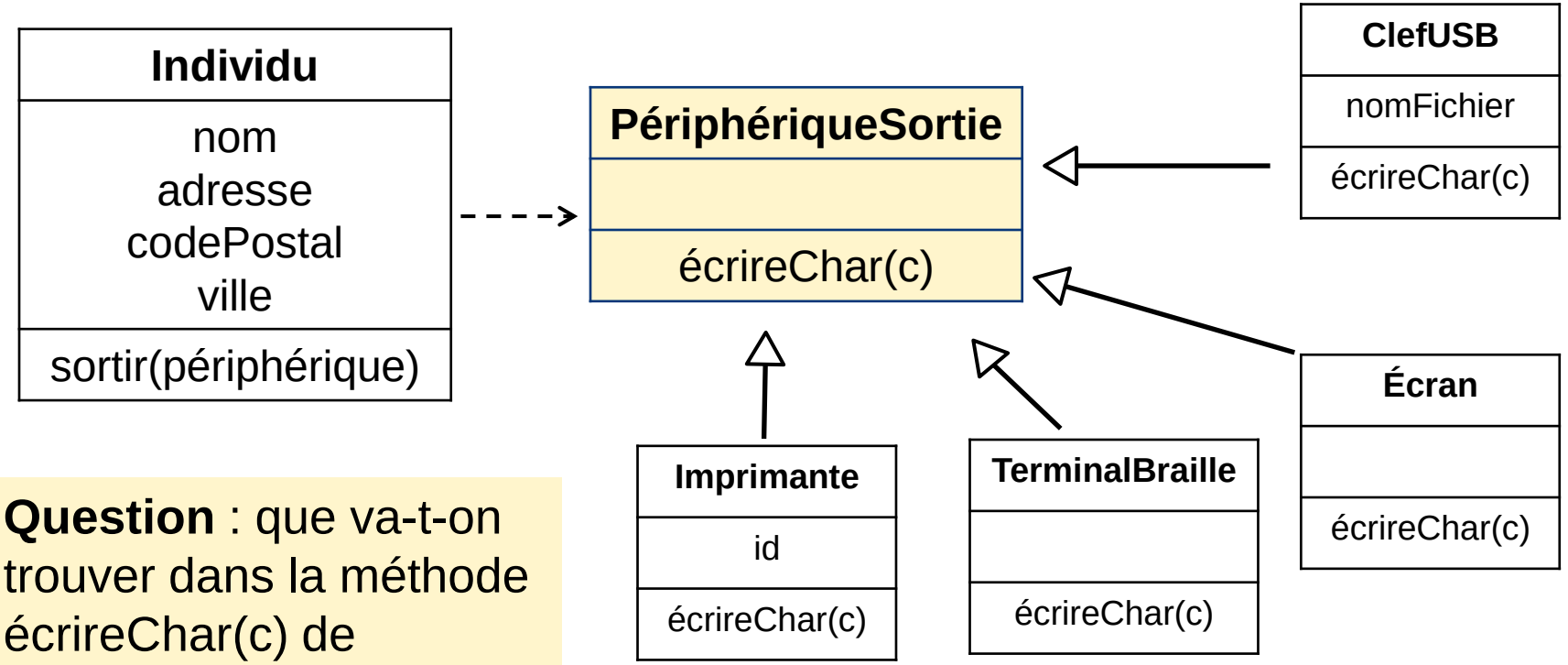
Que penser de la solution suivante ?

Que faire si on décide de changer le format de sortie ?



Exemple introductif

Idée : créer une classe PériphériqueSortie
et utiliser l'héritage



Question : que va-t-on trouver dans la méthode écrireChar(c) de PériphériqueSortie ?

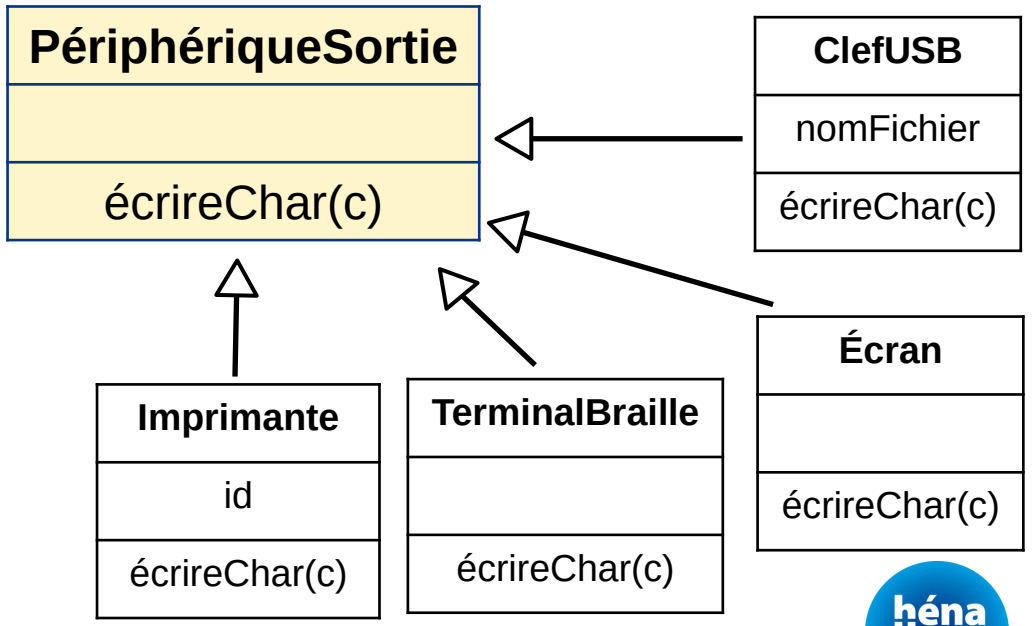
Exemple introductif

La méthode de PériphériqueSortie ne contient rien.

Le seul objectif de son existence est de s'assurer que chaque classe descendant de PériphériqueSortie possède bien une méthode écrireChar.

→ notion d'**interface**

```
class PériphériqueSortie {
  public void écrireChar (char c) {
    ...
  }
}
```



LES INTERFACES

Citer les méthodes que les classes
d'un groupe donné devront implémenter

Interfaces

Une **interface** = une classe...

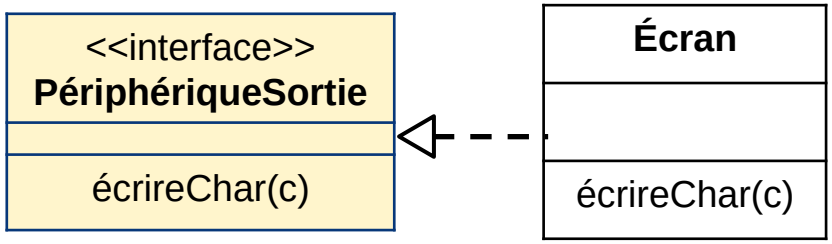
- où on se contente de préciser la signature des méthodes
- dont aucune méthode n'est implémentée
  (= toutes ses méthodes sont **abstraites**)
- sans instance (propre)
- sans constructeur
- sans attribut d'instance

Interface = modèle / gabarit / contrat pour des classes

Exemple : *« Pour être un périphérique de sortie, voici les méthodes que vous devez posséder… »*

Interfaces

Notations UML



- On dit que Écran **implémente** ou **réalise** l'interface PériphériqueSortie.

Interfaces

Syntaxe Java

```
interface PériphériqueSortie {
  public abstract void écrireChar (char c);
}
```

On peut omettre public abstract vu que les méthodes d'une interface le sont forcément !

```
class Écran implements PériphériqueSortie {
  public void écrireChar (char c) { … code … }
}
```

Chacune des méthodes annoncées dans l'interface doit être implémentée ici !

Interfaces

Note : une interface peut également posséder

- des méthodes de classe / statiques, et
- des attributs (qui sont automatiquement public, static et final), c'est-à-dire des constantes.

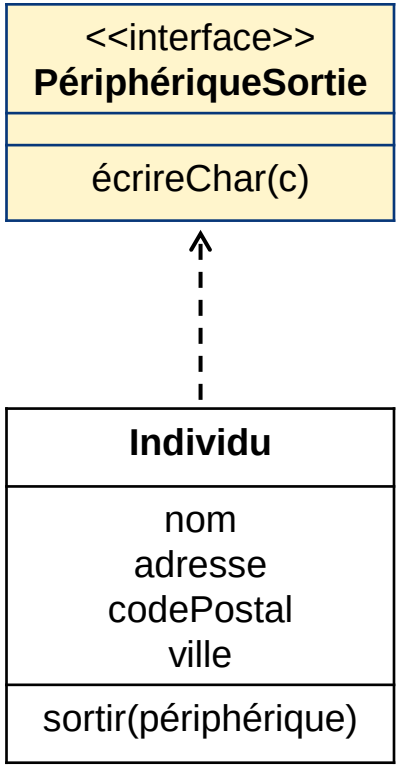
Interfaces

Utilisation d'une interface comme type de déclaration

```
interface PériphériqueSortie {
  void écrireChar (char c) {
    …
  }
}

class Individu {
  …
  public void sortir(PériphériqueSortie p) {
    …
    p.écrireChar(…)
    …
  }
}
```

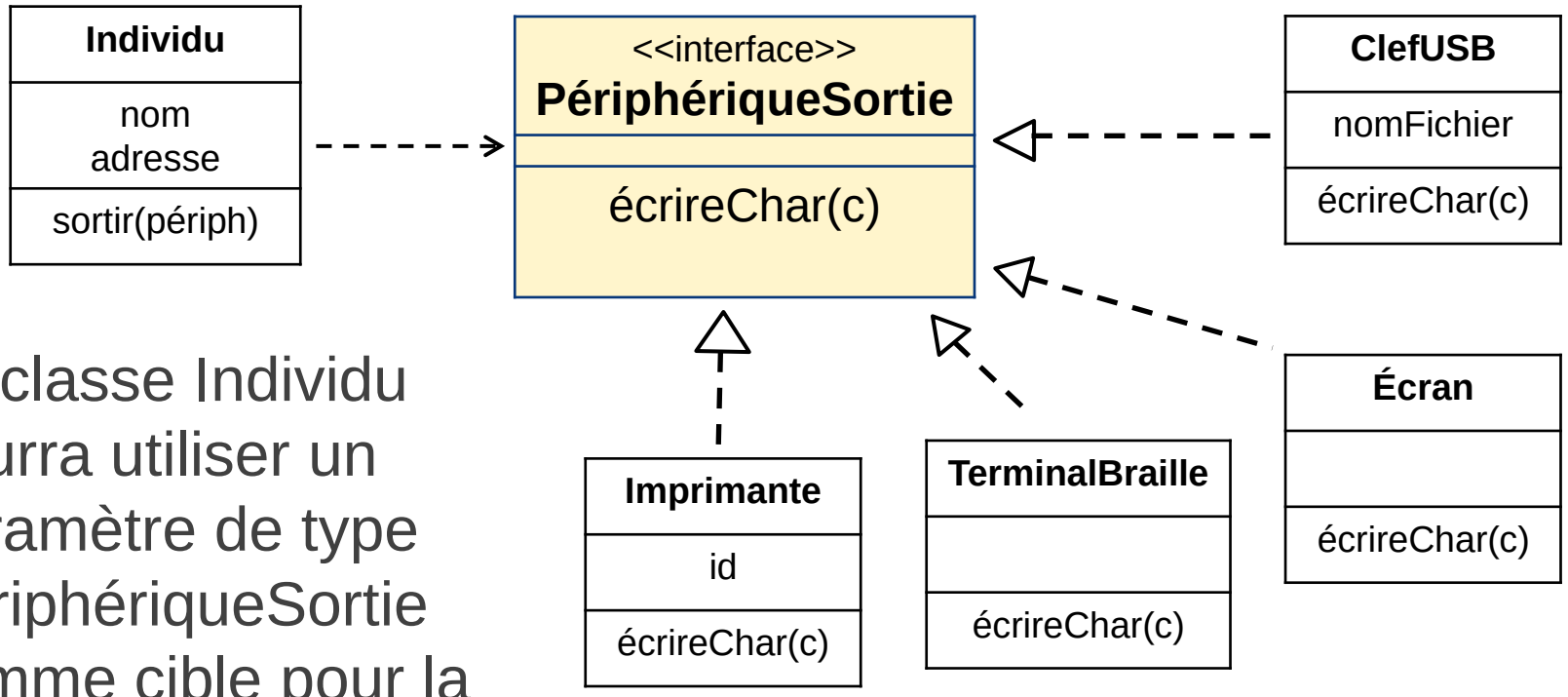
Il s'agira d'une instance d'une classe implémentant l'interface PériphériqueSortie et possédant donc une méthode écrireChar !



RETOUR SUR L'EXEMPLE

Exemple introductif

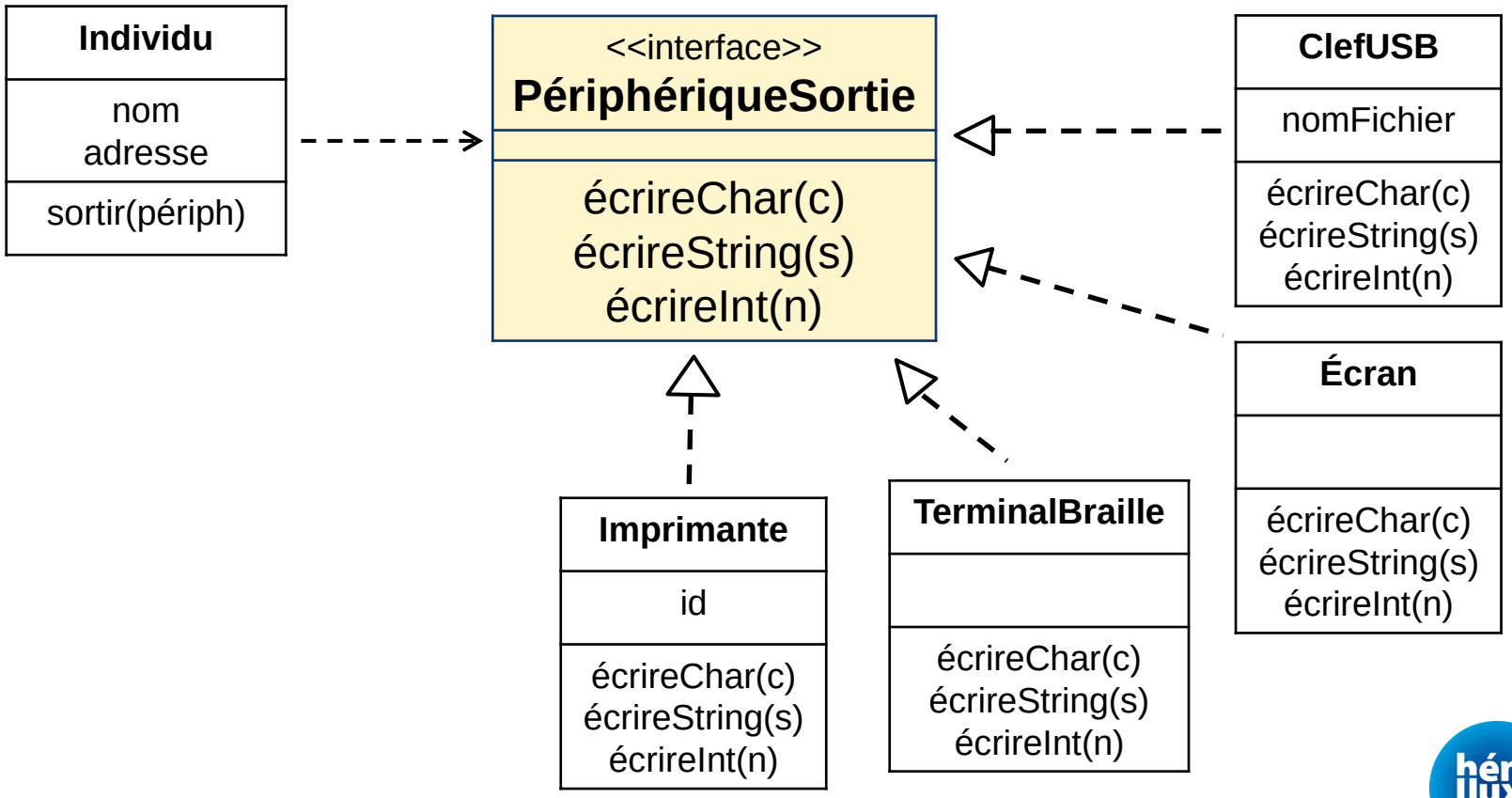
Que se passe-t-il si, pour faciliter les sorties, on ajoute des méthodes telles que écrireString, écrireInt, etc. ?



La classe Individu pourra utiliser un paramètre de type PériphériqueSortie comme cible pour la sortie des informations.

Exemple introductif

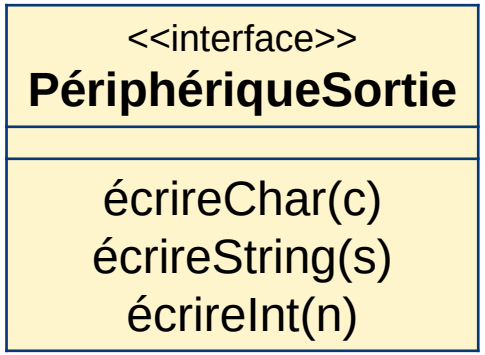
On dispose d'une interface PériphériqueSortie que vont implémenter tous les périphériques.



Exemple introductif

Dans l'interface PériphériqueSortie, toutes les méthodes doivent être abstraites. Les implémentations des 3 méthodes seront donc à charge de chacun des périphériques de sortie.

```
class Écran implements PériphériqueSortie {
  public void écrireChar (char c) {
    … code …
  }
  public void écrireString (String s) {
    for (int i = 0 ; i < s.length() ; i++)
      écrireChar(s.charAt(i));
  }
  public void écrireInt (int n) { … }
}
```



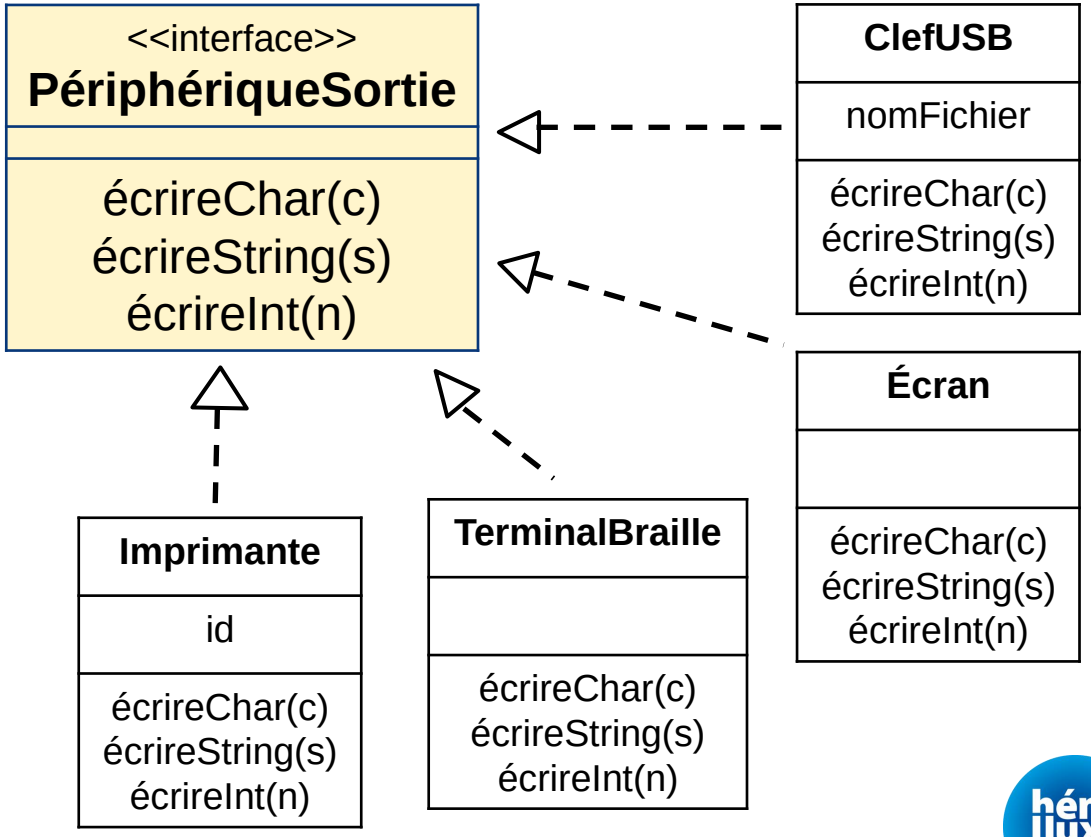
Implémentations identiques pour tous les périphériques de sortie !

Exemple introductif

Plutôt que de répéter les mêmes implémentations dans tous les périphériques de sortie, autant l'indiquer une seule fois dans PériphériqueSortie.

C'est impossible avec une interface (qui ne peut contenir que des méthodes abstraites).

→ **classe abstraite**



LES CLASSES ABSTRAITES

Hybrides à mi-chemin entre interfaces et classes (concrètes)

Classes abstraites

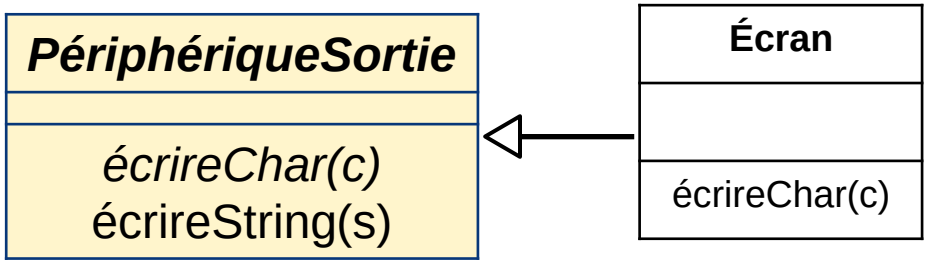
Une **classe abstraite** = une classe...

- où certaines méthodes sont abstraites
- sans instance (propre)
- qui peut avoir des attributs d'instance
- qui peut avoir des constructeurs

Classe abstraite = solution intermédiaire entre une classe « normale » (concrète) et une interface

Classes abstraites

Notations UML



- On dit que Écran **étend** ou **hérite** de la classe abstraite PériphériqueSortie.
- L'italique marque les éléments abstraits (le nom de la classe et la méthode abstraite ici).

Classes abstraites

Syntaxe Java

```
abstract class PériphériqueSortie {
  public abstract void écrireChar (char c);
  public void écrireString (String s) {
    for (int i = 0 ; i < s.length() ; i++)
      écrireChar(s.charAt(i));
  }
}

class Écran extends PériphériqueSortie {
  public void écrireChar (char c) { … }
  …
}
```

Ici, la mention « abstract » est obligatoire (contrairement aux interfaces).

Chacune des méthodes abstraites de la classe abstraite doit être implémentée ici !
Les méthodes concrètes, elles, sont héritées normalement.

REMARQUES FINALES

Compléments sur les classes abstraites et interfaces

Remarques finales

Les interfaces résolvent le problème de l'**héritage multiple**.

- *Problème de l'héritage multiple* :
  si les classes A et B possèdent une méthode en commun et C hérite de A et de B, de quelle méthode hérite C ?
- Interface = aucune implémentation, donc plus de souci !
- Une classe peut implémenter plusieurs interfaces !

```
class Écran implements PériphériqueSortie, PériphériqueEntrée {
  …
}
```

Remarques finales

- Héritages possibles

peut hériter de ↱	Classe concrète	Classe abstraite	Interface
Classe concrète	OUI (héritage standard)	OUI (implémenter toutes les méthodes abstraites)	
Classe abstraite	OUI (ajouter de nv méthodes abstraites)	OUI (ajouter de nouvelles méthodes abstraites ou ne pas les implémenter toutes)	
Interface	NON	NON	OUI (ajouter de nv méthodes abstraites) Mot-clé : extends !