24/03/2023
Version 3

POO

JÉRÉMY PERROUAULT

INTRODUCTION

LES BASES DE L'OBJET

INTRODUCTION

- Un **objet**, c'est donc la modélisation
  - D'une chose tangible (Smartphone, Stylo, Voiture, Personne, …)
  - D'une chose conceptuelle (Réunion, Service, Idée, …)

- Une **classe**, c'est donc le modèle de l'objet
  - Un **objet**, c'est l'instance d'une **classe**

INTRODUCTION

- Les attributs et les méthodes peuvent être
  - Publiques Accessibles par tous les **objets**
  - Privées Accessibles uniquement par l'**objet** lui-même (fermées aux autres)
  - Protégées Accessibles par l'**objet** et ses **classes** « filles » (fermées aux autres)
  - Package Accessibles par les **objets** du même package (par défaut en **JAVA**)

INTRODUCTION

- Principe d'encapsulation permet de ne pas avoir besoin de savoir comment ça fonctionne
  - Une interface qui masque la complexité
- Un Smartphone ne présente qu'une interface Homme/Machine
  - Nous manipulons le Smartphone, et lui interagit avec les autres objets et éléments qui le composent
- Permet également de protéger l'information contenue dans l'objet
  - Ne présente que les actions possibles, ou les attributs

INTRODUCTION

- Chaque instance a une vie dans l'application
  - Des modifications sur l'un n'aura pas d'effet sur les autres objets de même type
- Distinction par son adresse en mémoire
  - Deux **objets** `Produit` qui auraient les mêmes caractéristiques
  - Ils sont identiques sur le papier, mais sont réellement deux **objets** bien différents
    - Ils ont chacun une adresse mémoire qui leur est propre
  - C'est un peu l'équivalent d'une « clé primaire » dans une base de données

INSTANCIER

LES CONSTRUCTEURS

INSTANCIER

- Un **objet** est une **classe** dont l'instance peut être créée et détruite
  - `Constructeur`
    - Méthode appelée à l'instanciation de l'objet en mémoire
  - `Destructeur`
    - Méthode appelée avant sa destruction en mémoire
    - **JAVA** ne prévoit pas de `destructeur` manuel
      - En revanche, la méthode `finalize()` peut être utilisée (appelée par le `GarbageCollector`)

- Un objet n'est détruit que d'une seule façon
- Un objet peut être construit de différentes façons
  - Plusieurs constructeurs
  - Plusieurs paramètres

INSTANCIER

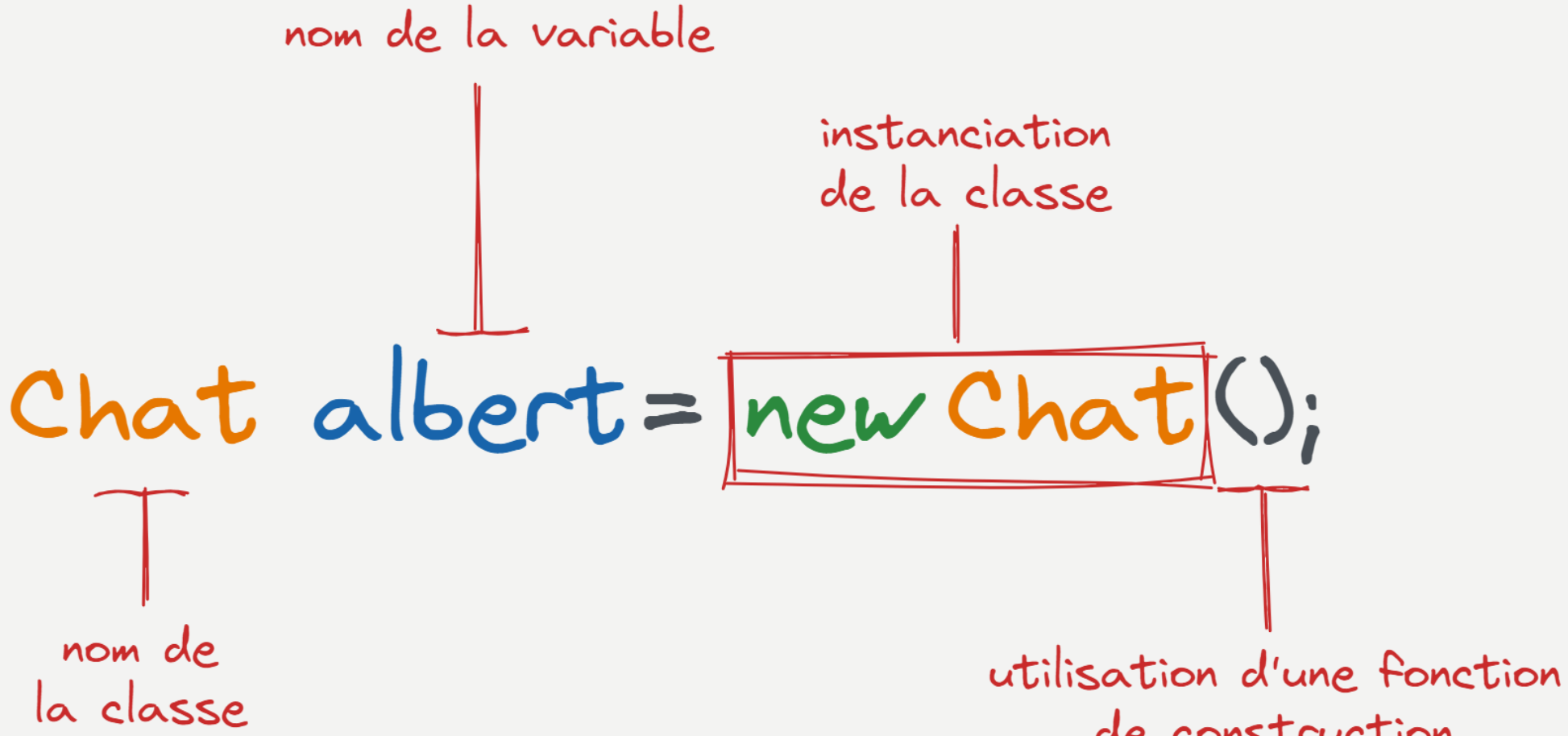
- Le `constructeur` est une méthode qui existe implicitement
  - Pas obligatoire de le décrire dans la **classe**
- Il est possible d'en ajouter un qui attend un ou plusieurs paramètres

```
public class Chat {
  private String attributPrive;

  public Chat() {
    // Constructeur
  }
}
```

```
/!\
Si un constructeur avec paramètres est défini,
le constructeur par défaut (implicite) n'existe plus !
Il faut le décrire à nouveau
/!\
```

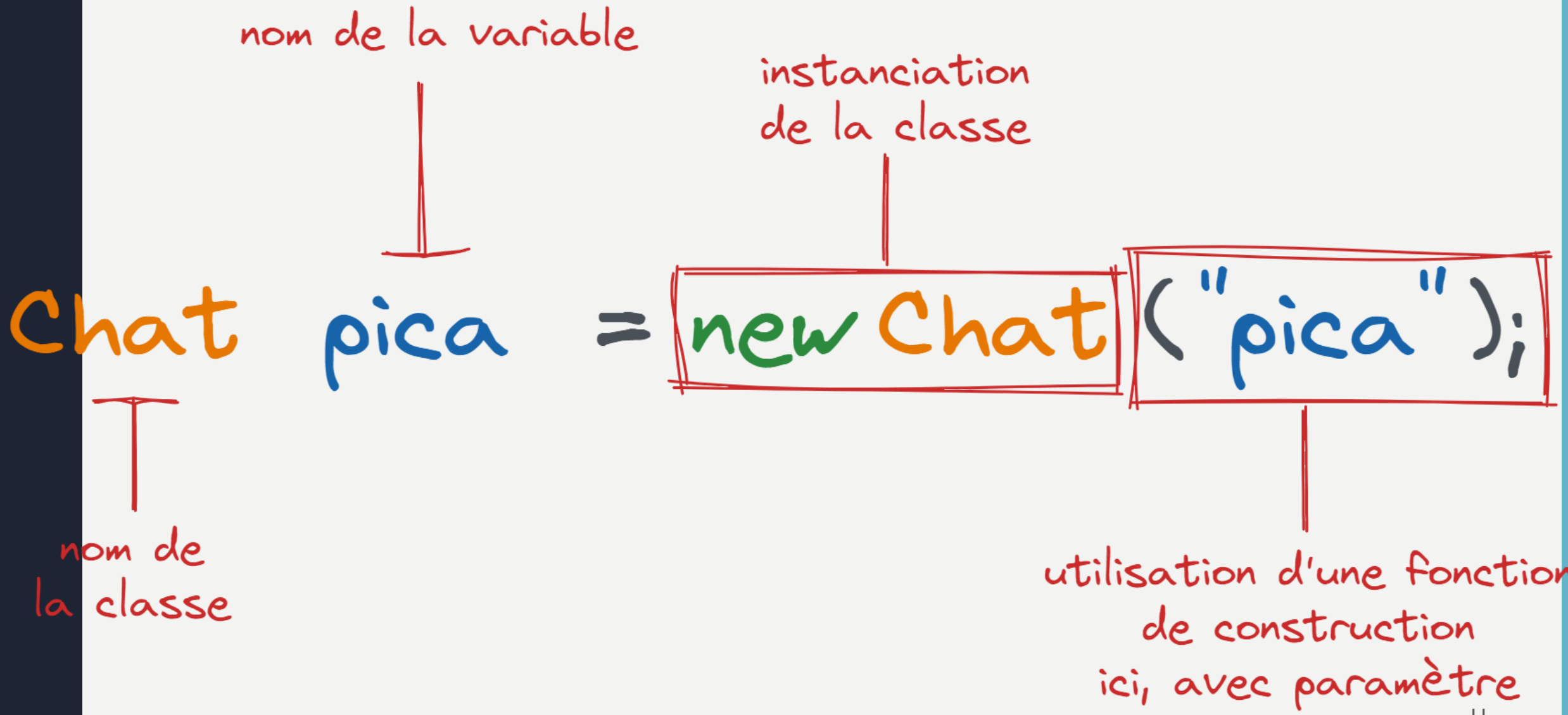
INSTANCIER



L'objet de type **Chat** sera sauvegardé dans la variable **albert**

La méthode **manger** de **albert** sera appelée, pas la méthode d'un autre **Chat**

INSTANCIER


nom de la variable
instanciation de la classe
Chat pica = new Chat("pica");
nom de la classe
utilisation d'une fonction de construction ici, avec paramètre

EXERCICE

- Créer une classe `Chat`
  - Avec un `constructeur`
    - Qui affiche dans la console « Création d'un chat ! »
    - Qui n'attend pas de paramètre
    - Puis qui attend le nom du chat en paramètre et qui affiche « Création du chat 'nom du chat' ! »
- Tester dans le programme principal

CONVENTIONS

LES RÈGLES COMMUNES

NOM DES CLASSES

- De manière générale, on va nommer les **classes**
  - En `CamelCase`
    - première lettre en majuscule
    - Si composée de plusieurs mots : première lettre de chaque mot en majuscule

- `Chat`
- `ChatSansGriffes`

ATTRIBUTS ET MÉTHODES

- En dehors des **classes** et **interfaces**, de manière générale, en **JAVA**
  - `lowerCamelCase`
    - Première lettre en minuscule
    - Si composée de plusieurs mots : première lettre de chaque mot en majuscule

- `chat`
- `monChatQuiDort`

NOM DES INTERFACES

- De manière générale, on va nommer les **interfaces**
  - En `CamelCase`
    - Avec en préfix « I » (comme **Intercace**)
    - première lettre en majuscule
    - Si composée de plusieurs mots : première lettre de chaque mot en majuscule

- `IChasseur`
- `IChasseurPuissant`

ATTRIBUTS

- Un attribut doit toujours être privé ou protégé, jamais publique
- Il faut des méthodes d'accès en lecture et en écriture publiques
  - Accesseurs (`getters`, `setters`)

  - `monAttribut`
  - `getMonAttribut()`
  - `setMonAttribut(valeur)`

HÉRITAGE

LES RELATIONS
L'HÉRITAGE

HÉRITAGE

- Principe visant à favoriser la réutilisabilité et l'adaptabilité des objets
- Très proche de la phylogénétique
  - Liens de parenté
    - Un chat est un animal, mais un animal n'est pas un chat

- **Classe** « fille »
  - **Classe** qui hérite d'une autre **classe** (on dit aussi qu'elle étend les fonctionnalités, ou spécifie)
- **Classe** « mère »
  - **Classe** directement parente de la **classe** « fille »

HÉRITAGE

- En **JAVA**, on manifeste la notion d'héritage via le mot-clé `extends`
  - Une **classe** « fille » ne peut hériter que d'une seule **classe** « mère »

- Dans un langage orienté objet, tout est **Object**, tout hérite implicitement de **Object**
  - **Object** est la plus « haute » **classe** en **POO**

HÉRITAGE

- « fille » possède tous les attributs et méthodes de sa « mère »
  - Publiques et protégés, elle n'aura pas accès aux éléments privés
  - Et comme tout est **Object**, tout **objet** a déjà des méthodes disponibles (`toString`, …)
- « fille » spécialise les attributs et les méthodes de sa « mère »
  - Un chat, c'est un mammifère plus spécifique, qui lui-même est moins abstrait qu'un animal
    - Spécialisation de la méthode `manger`
    - Un chat ne mange pas de la même façon qu'un chimpanzé
    - C'est donc la méthode `manger` du chat qui sera utilisée, et non la méthode de l'animal
      - Polymorphisme d'héritage
  - Un chat peut miauler, tandis qu'un chien peut aboyer

POLYMORPHISME

- Il existe plusieurs types de *polymorphismes*
  - Héritage (**overriding**)
    - C'est la réécriture – la spécialisation – d'une méthode
    - Un chat a sa propre façon de manger, mais tous les mammifères mangent
  - Ad hoc (**overloading**)
    - C'est la redéfinition d'une méthode dans la même **classe**, avec d'autres paramètres
    - Un chat court différemment selon le type de terrain (terre ou boue)

EXERCICE

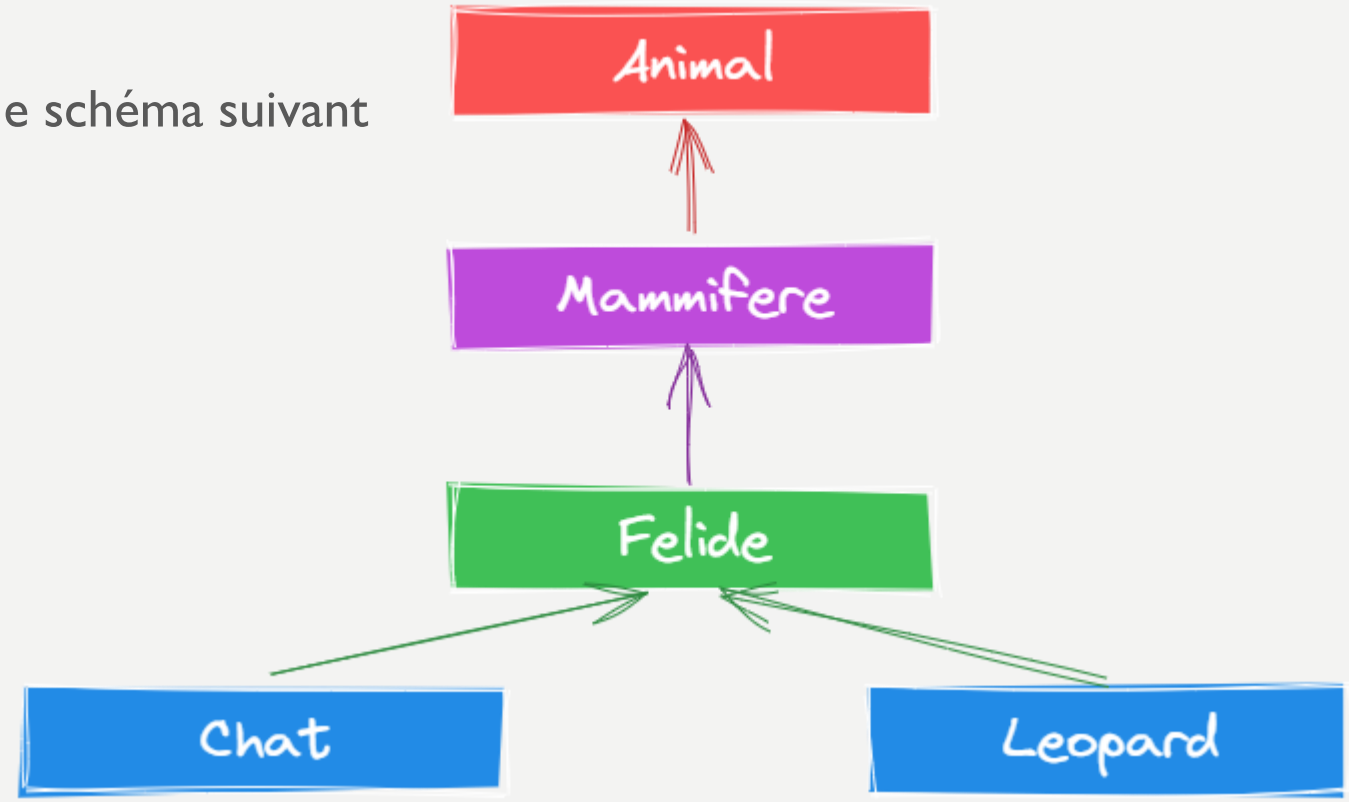
- Dans le programme principal, afficher l'instance du chat dans la console
- … Puis, modifier la méthode `toString` de la classe `Chat`
  - Rejouer le programme principal, noter la différence
- … Puis, retourner le nom du chat dans la méthode `toString`
  - Rejouer le programme principal, noter la différence

EXERCICE

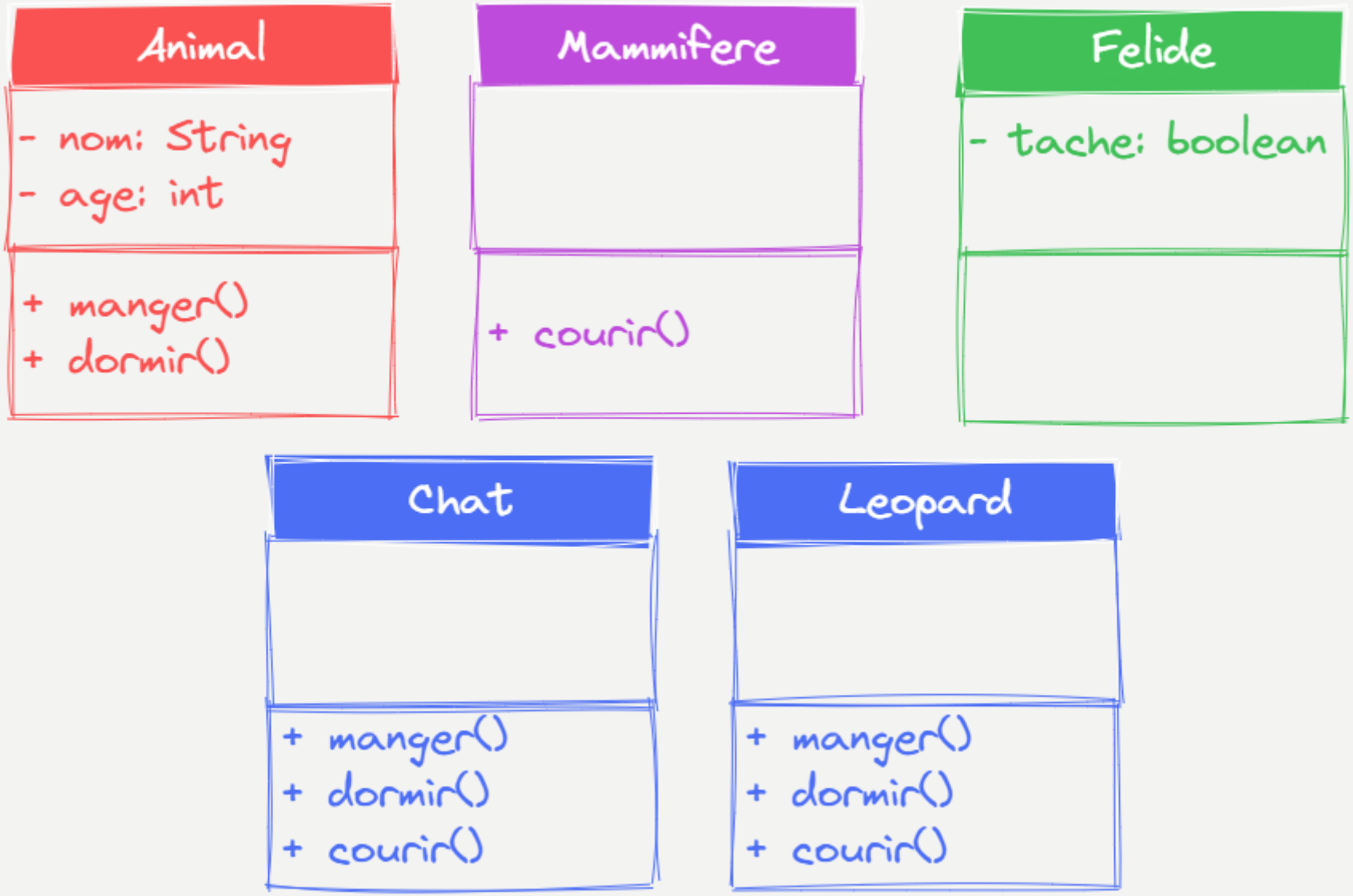
- Créer une classe `Felide`
    - La classe `Chat` doit en hériter
  - Ajouter un constructeur à `Felide`
    - Afficher « Création d'un Félidé ! »
    - Que se passe-t-il ?

EXERCICE

- Modéliser sous forme de classes le schéma suivant



EXERCICE


Animal
- nom: String
- age: int
+ manger()
+ dormir()
Mammifere
+ courir()
Felide
- tache: boolean
Chat
+ manger()
+ dormir()
+ courir()
Leopard
+ manger()
+ dormir()
+ courir()

LES CONTRATS

LES CLASSES ABSTRAITES
LES INTERFACES

CLASSE ABSTRAITE

- **Classe** qui ne peut pas être instanciée
  - On ne pourra pas avoir d'objet pour une classe abstraite
- Défini des attributs et des méthodes
  - Ils sont là, mais n'existent pas
- Une **classe** abstraite est nécessairement une **classe** « mère »
  - Ses « filles » pourront spécialiser les méthodes (*polymorphisme*)
- Les méthodes
  - Peuvent être décrites
  - Peuvent être abstraites

CLASSE ABSTRAITE

- Reprenons la classe `Animal`
  - On ne sait pas comment un animal mange
  - On sait comment un chat mange
  - On sait comment un léopard mange
- `Animal` est abstrait alors que `Chat` et `Leopard` sont concrets
  - `Animal` ne décrit pas la façon dont il mange
  - Seuls `Chat` et `Leopard` décrivent le comportement dans cette action

CLASSE ABSTRAITE

- En **JAVA**, on manifeste la notion d'abstraction via le mot-clé
  - `abstract`

- Une **classe** peut être abstraite, et certaines méthodes de cette **classe** peuvent l'être aussi
  - Certaines autres méthodes sont concrètes, avec une comportement établit
  - Les méthodes abstraites devront être implémentées par les premières **classes filles** concrètes
    - C'est dans leur contrat

EXERCICE

- Un animal sait dormir d'une manière générale
  - Mais on va respécifier ce comportement pour tous les Félidés
- Un mammifère sait courir, mais on ne sait pas encore comment il fait
  - Par contre, tous les félidés courent de la même façon

- Modifier la hiérarchie des **classes** pour correspondre à une meilleure réalité
  - Utiliser `abstract` aux endroits les plus cohérents

INTERFACE

- Une **classe fille** ne peut hériter que d'une seule **classe mère**
  - Un chat est un mammifère, mais tous les mammifères ne chassent pas
  - Le chat n'est pas le seul mammifère à chasser, il y a notamment les félins et les canins

- Une **interface**, c'est une **classe** 100% abstraite !
  - Impossible à instancier
  - Aucun attribut
  - Les méthodes ne peuvent pas être décrites – il n'y a aucun comportement qui est décrit
    - C'est donc aux **classes** qui l'implémentent de le faire, et elles en ont l'obligation
- On parle d' "implémentation" d'interface

INTERFACE

- En **JAVA**, on manifeste la notion d'implémentation via le mot-clé
  - `implements`
  - Les **interfaces** implémentées, si elles sont plusieurs, sont séparées par une virgule

- Une **classe**
  - Ne peut pas hériter de plus d'une **classe**
  - Peut implémenter plusieurs **interfaces**
- Une interface
  - Peut hériter de plusieurs **interfaces**

INTERFACE

- Pour que nos félins (et les autres chasseurs) puisse chasser
  - Définition d'une interface `IChasseur`
    - Méthode `chasser` qui attendra un `IChassable` en argument
  - Implémentation de `IChasseur` par `Felide` et `Canide`
    - Description du comportement `chasser`
    - Qui pourront être de nouveau spécialiser pour `Chat` et `Léopard`

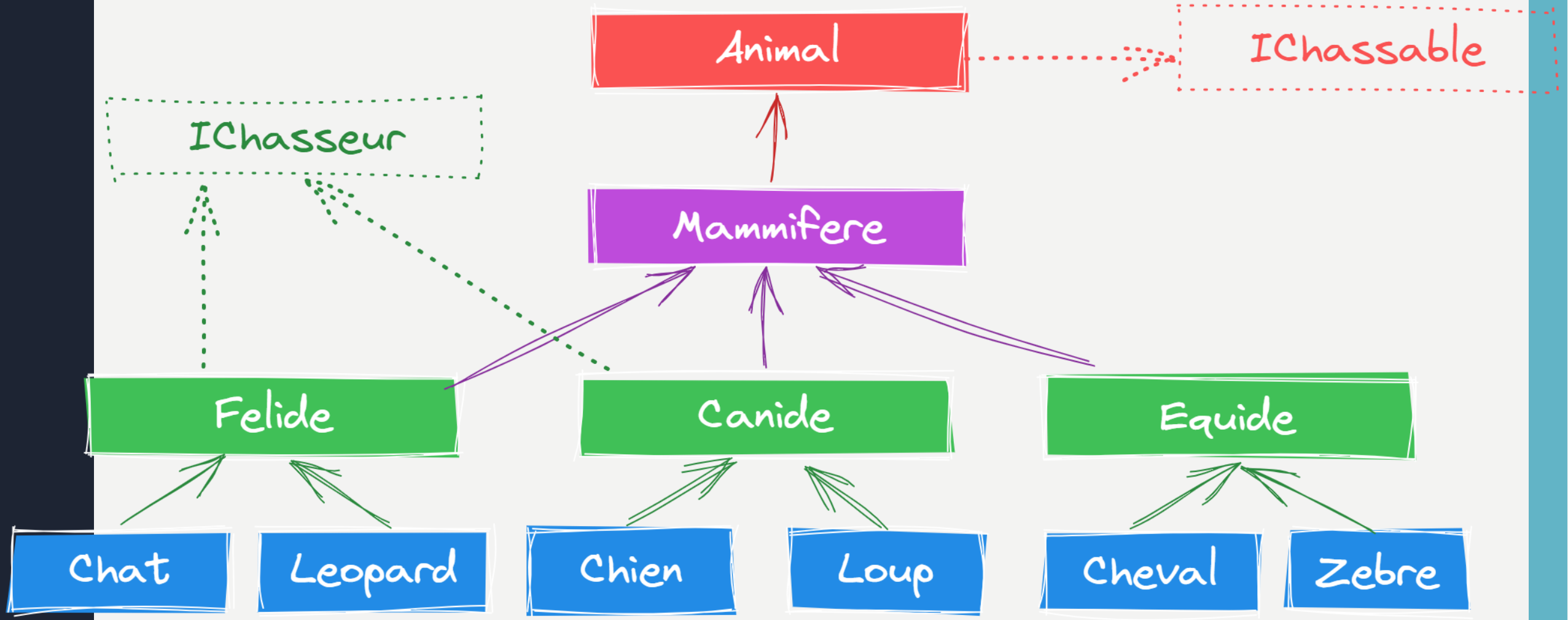
- On peut manipuler un `Chat`
  - Comme un mammifère, sans savoir qu'il peut chasser
  - Comme un chasseur, sans savoir qu'il peut manger
  - Comme un chat, en sachant qu'il peut manger et faire très mal

INTERFACE

- Pour que nos animaux puissent être chassés
  - Définition d'une interface `IChassable`
    - Méthode `repondre` qui attendra un `IChasseur` en argument
  - Implémentation de `IChassable` par tous les animaux
    - Description du comportement `defendre` général : « L'animal se sauve »

- On peut manipuler un `Animal`
  - Comme un animal, sans savoir qu'il peut être chassé
  - Comme un chassable, sans savoir qu'il peut manger

INTERFACE


Animal
IChassable
IChasseur
Mammifere
Felide
Canide
Equide
Chat
Leopard
Chien
Loup
Cheval
Zebre

EXERCICE

- Modéliser sous forme de **classes** et d'**interfaces** le schéma précédent
  - Implémenter les méthodes
    - `chasser(IChassable proie)`
    - `defendre(IChasseur predateur)`
  - Le cheval se laissera faire

LES CONTRATS

- Les **classes** abstraites et les **interfaces** sont comme des contrats
  - On garantit que leurs méthodes seront implémentées par la **classe** « fille »
    - Cependant, la méthode peut ne rien faire dans la **classe** « fille », mais elle existe

LES CONTRATS

- Couplage faible
  - Les composants (**classes**) deviennent indépendants les uns des autres
    - Plus facilement manipulable et réutilisables
    - Plus facilement évolutif
    - Plus facilement testables
  - Démonstration : `Repository`
- En conclusion
  - Les **interfaces** sont plutôt petites et indépendantes
  - Si plusieurs déclinaisons du comportement existent, créer une **classe** abstraite
  - Si la fonctionnalité peut être réutilisée dans plusieurs **classes**, créer une interface

LES CONTRATS

- Couplage faible
  - Les composants (**classes**) deviennent indépendants les uns des autres
    - Plus facilement manipulable et réutilisables
    - Plus facilement évolutif
    - Plus facilement testables
- En conclusion
  - Les **interfaces** sont plutôt petites et indépendantes
  - Si plusieurs déclinaisons du comportement existent, créer une **classe** abstraite
  - Si la fonctionnalité peut être réutilisée dans plusieurs **classes**, créer une interface

DÉMONSTRATION

- `ArrayList` et `List`

- Intérêt du couplage faible, via Pattern `Factory`

GÉNÉRIQUES

LES TEMPLATES GÉNÉRIQUES

DÉMONSTRATION

- Pattern `Repository`

TEMPLATE GÉNÉRIQUE

- C’est le cas notamment des listes qu’on utilise de la manière suivante

```
List<Chat> mesChats = new ArrayList<>();

mesChats.add(new Chat());
```

EXERCICE

- Créer une liste de 10 animaux (qui seront en fait des chevaux, des chats, …)
- Créer une liste de 5 chasseurs (qui seront des chats et des chiens)

- Parcourir ces 2 listes
  - Les chausseurs chassent les 10 animaux

ENUMS

LES ENUMÉRATIONS

LES ÉNUMÉRATEURS

- Sans les énumérateurs

```
public void testType(int type) {
  if (type == 0) {
    // ...
  }
}
```

  - (éventuellement avec des constantes)

Avec les énumérateurs

```
public enum TypePersonne {
    CLIENT, FOURNISSEUR;
}
```

```
public void testType(TypePersonne type) {
  if (type == TypePersonne.CLIENT) {
    // ...
  }
}
```

EXERCICE

- Créer un énumérateur `TypeAnimal`
  - `EAU`, `TERRE`, `AIR`, `FEU`
- Ajouter une méthode `getType()` à `Animal`
  - Dont le comportement est inconnu et qui sera spécifier dans les classes `Felide`, `Canide`, …
    - `Felide` est de type `TERRE`
    - `Canide` est de type `EAU`
    - `Equide` est de type `FEU`
- Dans le programme principal, créer (ou utiliser) une liste des 10 animaux
  - Pour chaque animal, afficher son type

LES MOTS CLÉS

LES MOTS CLÉS À RETENIR

LES MOTS CLÉS

Mot clé	Fonction
`class`	Définition d'une **classe**
`interface`	Définition d'une **interface**
`new`	Instancier une **classe**
`abstract`	Définition d'une **classe** ou d'une méthode abstraite
`public`	**Classe**, attribut ou méthode accessible par tous
`protected`	**Classe**, attribut ou méthode accessible par l'objet et ses filles
`private`	**Classe**, attribut ou méthode accessible par l'objet uniquement
`static`	Permet d'accéder à l'attribut ou méthode sans instance
`extends`	Héritage d'une **classe** mère par une **classe** fille
`implements`	Implémentation d'une **interface** par une **classe**
`this`	Dans l'objet, accède aux valeurs de ses attributs et à ses méthodes
`super`	Dans l'objet, accède aux valeurs des attributs et méthodes de son parent
`final`	Permet de « figer » une **classe**, une méthode, un attribut ou une variable