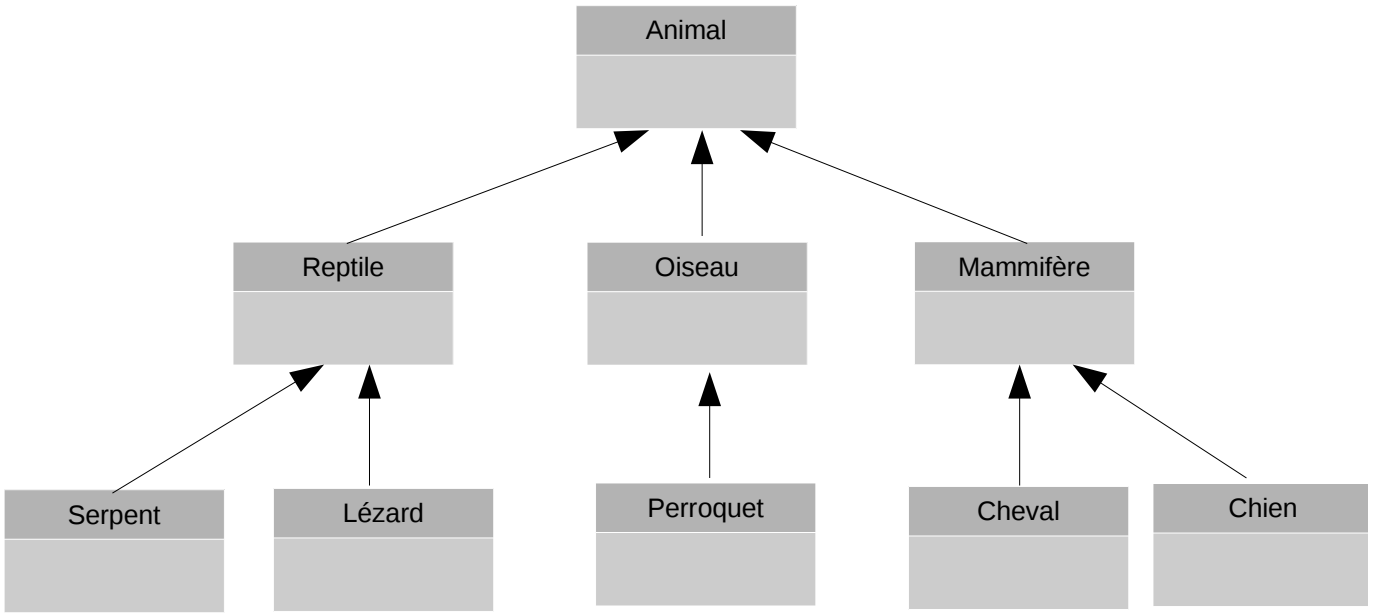
R308 : Consolidation de la programmation
Chapitre 2 : Héritage et Polymorphisme

1.1 Présentation :

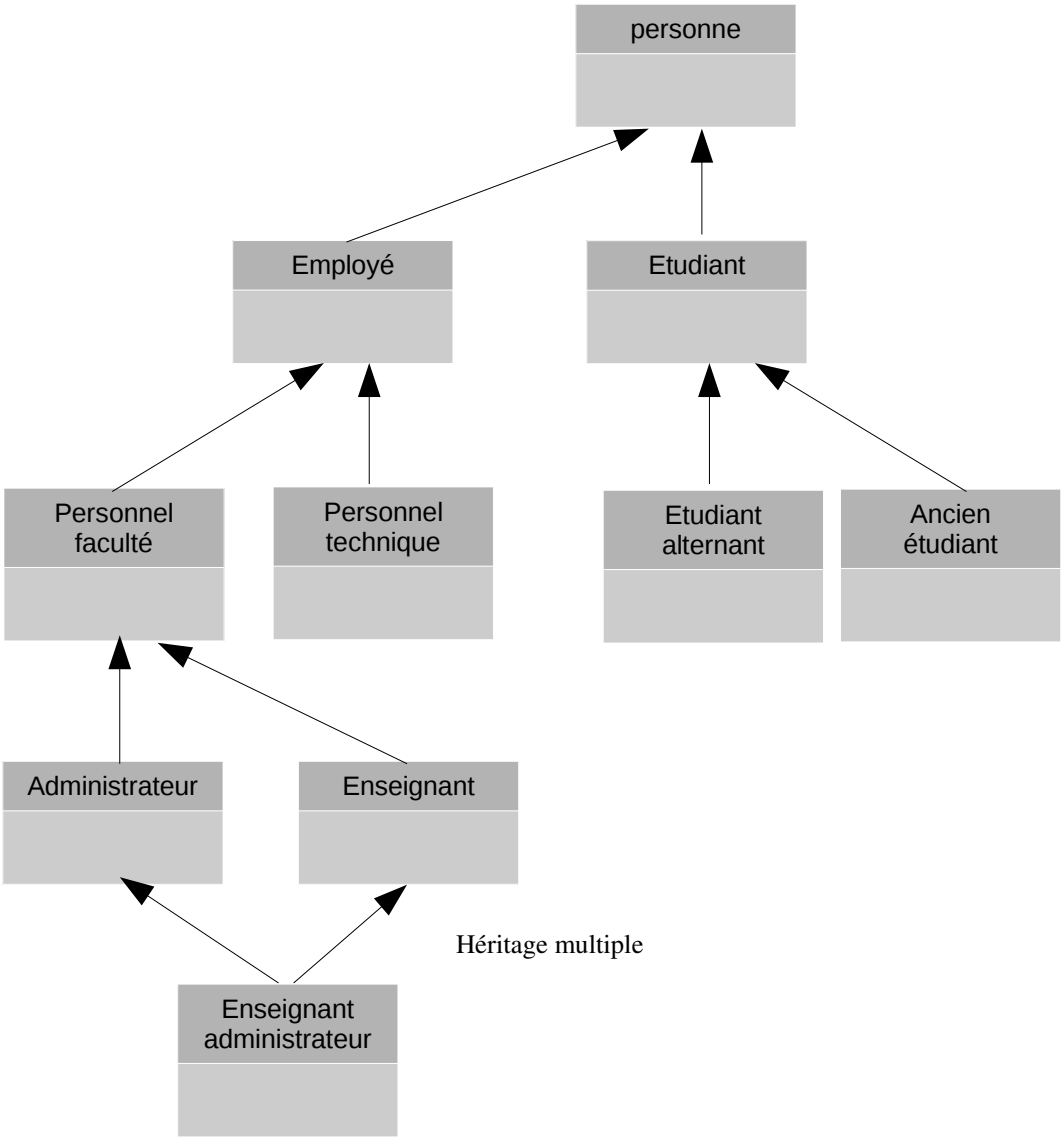
- L'héritage permet de définir des classes dérivées à partir de classes de base.
- Une classe dérivée hérite d'une classe mère et pourra rajouter ses propres membres (ou les modifier) sans retoucher la classe de base.
- Plusieurs classes pourront hériter d'une même classe de base.
- L'héritage peut s'effectuer à plusieurs niveaux.
- Certains langages autorisent l'héritage multiple (Python, C++)

1.2 Exemples de graphe d'héritage :

1. Le monde animal :



2. La communauté universitaire



Héritage multiple

1.3 Différence entre héritage et composition :

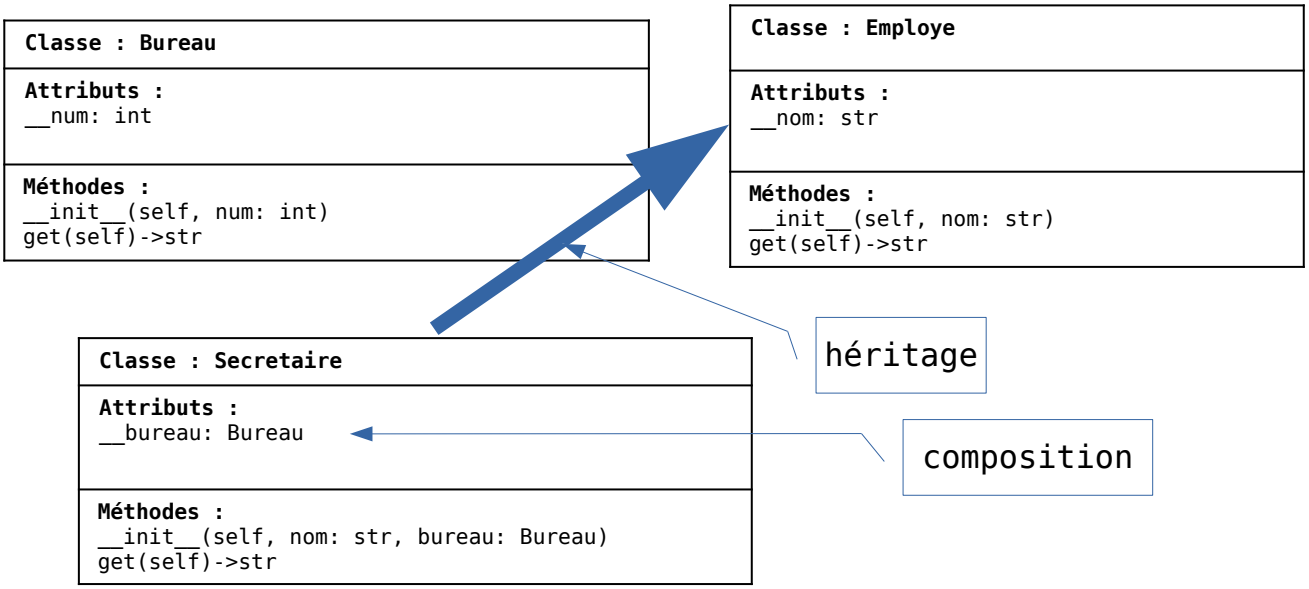
Savoir faire la distinction entre "est un" et "possède un".

"possède un" => Composition

"est un" => Héritage

1. Exemple : un employé d'une entreprise :

On peut définir les trois classes suivantes :



Un Employé est une Personne => Héritage

Un Employé possède un Bureau => Composition
Un Employé n'est pas un Bureau ni une ancienneté.

1.4 Mise en œuvre de l'héritage en python à partir de l'exemple précédent

```
class Bureau:
	# constructeur
	def __init__(self, num: int):
		self.__num: int = num
	def get(self)-> str:
		return f" numéro de bureau :  {self.__num}"
```

```
class Employe:
	# constructeur
	def __init__(self, nom: str):
		# attributs
		self.__nom: str = nom
	# observateur
	def get(self)-> str:
		return self.__nom
```

1. La classe Secrétaire hérite de la classe Employé :

```
class Secretaire(Employe):                          # la classe mère entre parenthèse (1)
	# constructeur
	def __init__(self, nom: str, bureau: Bureau):
		Employe.__init__(self, nom)                 # appel au constructeur classe mère (2)
		self.__bureau: Bureau = bureau              # attribut de type Bureau (3)
	# observateur
	def get(self)-> str:                            # surdéfinition de la méthode get() (4)
		chaine: str = None
		chaine = Employe.get(self)                  # appel méthode get() classe mère (5)
		chaine += self.__bureau.get()               # appel méthode get() attribut __bureau (6)
		return chaine
```

(1) pour préciser l'héritage, la classe mère est placer entre parenthèse. En cas d'héritage multiple, les différentes classes mères sont séparées par des virgules.
(2) l'appel au constructeur de la classe mère doit être la première instruction du constructeur.
l'attribut self est indispensable.
En cas d'héritage simple, une autre écriture est possible :

```
	super().__init__(nom)
```

(3) __bureau est un nouvel attribut de la classe Secrétaire
(4) sur-définition de la méthode get() de la classe mère.
(5) appel de la méthode get() de la classe mère.
Autre appel possible :

```
	chaine = super().get()
```

(6) # appel méthode get() de l'attribut __bureau

2. Exemple de main :

```
if __name__ == "__main__":
	# declaration d'une reference de type Secretaire
	secretaire: Secretaire = None
	# instanciation des variables
	secretaire = Secretaire("claudine", Bureau(101))
	# affichage des informations de la secrétaire
	print(secretaire.get())
	# affichage du nom de la classe
	print(secretaire.__class__.__name__)
```

```
$ claudine numéro de bureau : 101
$ Secretaire
```

1.5 Méthode abstraite :

Il est parfois intéressant de définir des méthodes abstraites dans un graphe d'héritage.
Une méthode abstraite est une méthode commune à un ensemble de classes filles mais qui sera spécifique à chaque classe fille.
Exemple : Le monde animal :
Tous les animaux se déplacent, chaque animal à une façon propre de se déplacer.

```python
from abc import abstractmethod

class Animal:
	def __init__(self, nom: str):
		self.__nom: str = nom

	@abstractmethod
	def deplacement(self)->None: # méthode abstraite
		pass

	def get(self)-> str:
		"""observateur qui fait appel à la méthode abstraite
		"""
		return f" {self.__nom} se deplace {self.deplacement()} "

class Mammifere(Animal):
	"""
	classe Mammifere qui hérite de la classe Animal
	"""
	def __init__(self, nom: str):
		Animal.__init__(self, nom)

	def deplacement(self)-> str:
		"""definition du mode de déplacement des mammiferes
		"""
		return "à quatre pates"

if __name__ == "__main__":
	# declaration des references d'objets
	mammifere1: Mammifere = None
	# instanciation des objets
	mammifere1 = Mammifere("vache")
	# afficher les carateristiques de l'animal
	print(mammifere1.get())
```

1. Résultat DE L'AFFICHAGE :

```
vache se deplace à quatre pates
```

1.6 Héritage multiple :

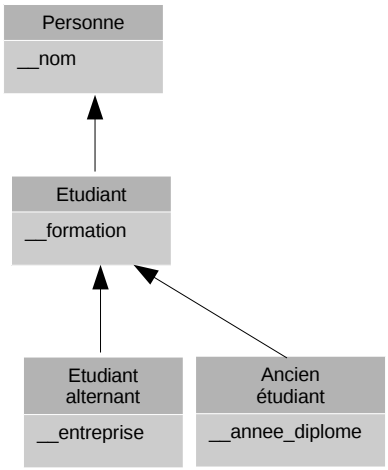
L'héritage multiple doit être utilisé avec précaution.

```python
class MereA:
	def __init__(self):
		self.__msg = "classe MereA"
	def get(self)-> str:
		return self.__msg

class MereB:
	def __init__(self):
		self.__msg = "classe MereB"
	def get(self)-> str:
		return self.__msg

class Fille(MereA, MereB):
	def __init__(self):
		MereA.__init__(self) # appel constructeur classe MereA
		MereB.__init__(self) # appel constructeur de la classe MereB

	def get(self)-> str: # sur definition de la methode get()
		chaine: f"{MereA.get(self)} " " {MereB.get(self)}"
		return chaine
if __name__ == "__main__":
	# declaration de la variable filles de type Fille
	fille: Fille = None
	# instanciation de la variable fille
	fille = Fille()
	# appel de la méthode get() le l'objet fille
	print(fille.get())
```

2 Polymorphisme :

Définition simple : nom masculin, caractère de ce qui peut avoir ou adopter plusieurs formes différentes.
Ce concept complète celui de l'héritage.
Le polymorphisme permet de manipuler des objets sans connaître leur type exact, à condition que les objets appartiennent au même graphe d'héritage.

2.1 Exemple à partir du graphe d'héritage de la communauté universitaire :



Chaque classe possédera une méthode **get()** qui retournera l'ensemble des attributs de l'objet concerné.

```
class Personne:
	def __init__(self, nom: str):
		self.__nom = nom
	def get(self)-> str:
		return self.__nom
	def get_classe(self)-> str: # retourne le nom de la classe
		return type(self).__name__

class Etudiant(Personne):
	def __init__(self, nom: str, formation: str):
		Personne.__init__(self, nom)
		self.__formation = formation
	def get(self)-> str:
		return f"{Personne.get(self)} formation : {self.__formation}"

class Alternant(Etudiant):
	def __init__(self, nom: str, formation: str, entreprise: str):
		Etudiant.__init__(self, nom, formation)
		self.__entreprise = entreprise
	def get(self)-> str:
		return f"{Etudiant.get(self)} entreprise : {self.__entreprise}"

class Ancien(Etudiant):
	def __init__(self, nom: str, formation: str, annee_diplome: str):
		Etudiant.__init__(self, nom, formation)
		self.__annee_diplome = annee_diplome
	def get(self)-> str:
		return f"{Etudiant.get(self)} dilopme en : {self.__annee_diplome}"

if __name__ == "__main__":
	liste_etudiants: list[Personne] = list()
	liste_etudiants.append(Etudiant("julia", "DUT R&T"))
	liste_etudiants.append(Alternant("kim", "DUT INFO", "OBS"))
	liste_etudiants.append(Ancien("perrine", "DUT MP", 2018))

	# afficher les caractéristique de chaque étudiant
	for etu in liste_etudiants:
		print(etu.get() + " ( " + etu.get_classe() + " )") (1)
	# afficher uniquement les alternant (3 possibilités)
	for etu in liste_etudiants:
		if (etu.get_classe() == "Alternant"):
			print(etu.get())
	for etu in liste_etudiants:
		if (etu.__class.__name == "Alternant"):
			print(etu.get())
	for etu in liste_etudiants:
		if (isinstance(etu, Alternant):
			print(etu.get())
```

```
----------toutes les personnes----------------
julia formation : DUT R&T ( Etudiant )
kim formation : DUT INFO entreprise : OBS ( Alternant )
perrine formation : DUT MP dilopme en : 2018 ( Ancien )
----------toutes les alternants----------------
kim formation : DUT INFO entreprise : OBS
```

(1) : la méthodes get() appelée sera celle de la classe de l'objet concerné => polymorphisme