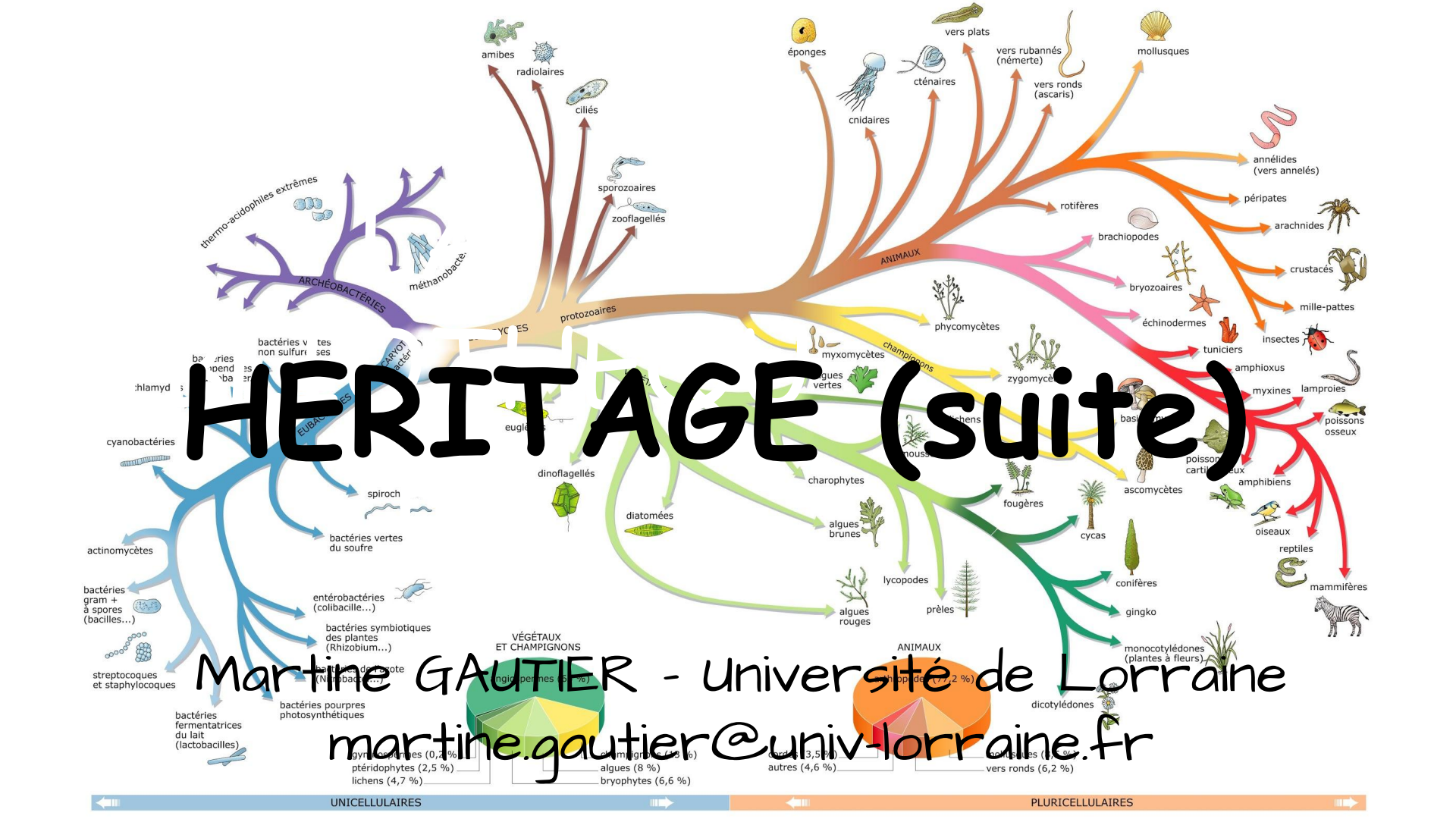
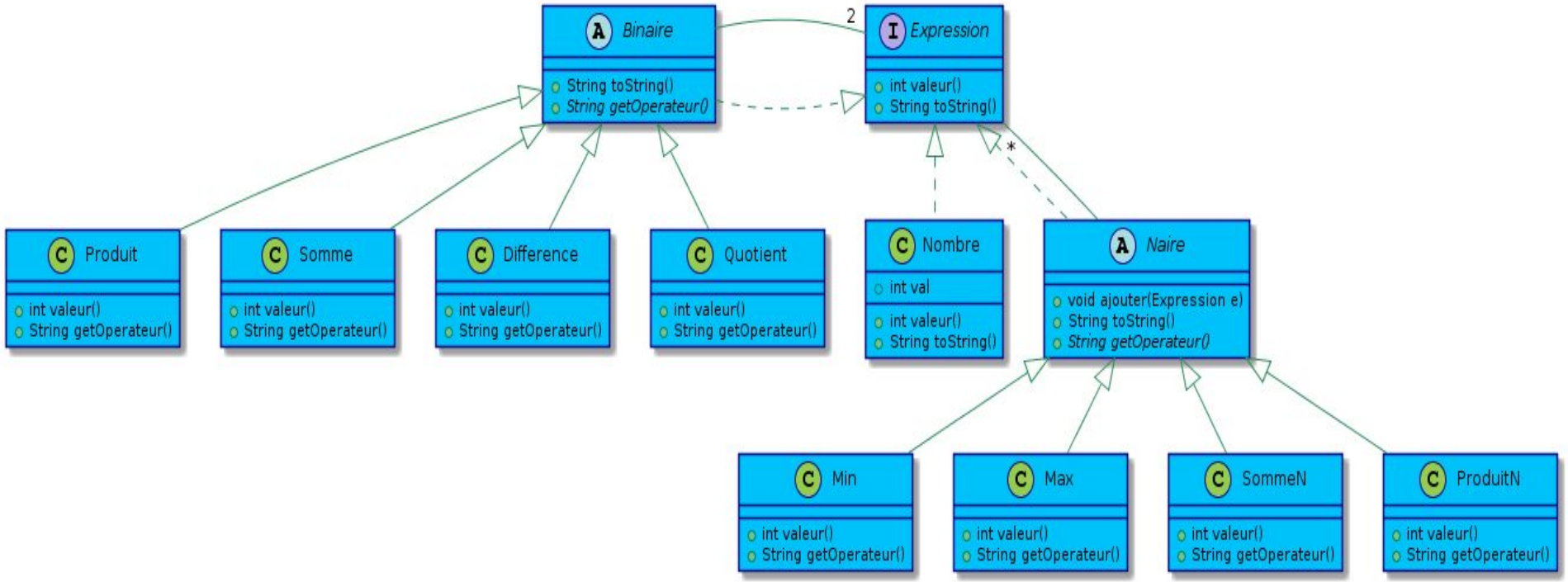

HERITAGE (suite)
Martine GAUTIER - Université de Lorraine
martine.gautier@univ-lorraine.fr
amibes
radiolaires
ciliés
sporozoaires
zooflagellés
protozoaires
thermo-acidophiles extrêmes
ARCHÉOBACTÉRIES
méthanobacté
éponges
cnidaires
cténaires
vers plats
vers rubannés (némerte)
vers ronds (ascaris)
mollusques
annélides (vers annelés)
péripates
arachnides
crustacés
mille-pattes
insectes
rotifères
brachiopodes
bryozoaires
échinodermes
tuniciers
amphioxus
myxines
lamproies
poissons osseux
ANIMAUX
phycomycètes
myxomycètes
champignons
zygomycètes
lichens
mousses
ascomycètes
poissons cartilagineux
amphibiens
oiseaux
reptiles
mammifères
cyanobactéries
spiroch
dinoflagellés
diatomées
charophytes
algues brunes
algues rouges
fougères
cycas
conifères
gingko
lycopodes
prêles
monocotylédones (plantes à fleurs)
dicotylédones
actinomycètes
bactéries gram + à spores (bacilles...)
streptocoques et staphylocoques
bactéries fermentatrices du lait (lactobacilles)
bactéries vertes du soufre
entérobactéries (colibacille...)
bactéries symbiotiques des plantes (Rhizobium...)
bactéries pourpres photosynthétiques
VÉGÉTAUX ET CHAMPIGNONS
ptéridophytes (2,5 %)
lichens (4,7 %)
algues (8 %)
bryophytes (6,6 %)
ANIMAUX
autres (4,6 %)
vers ronds (6,2 %)
UNICELLULAIRES
PLURICELLULAIRES

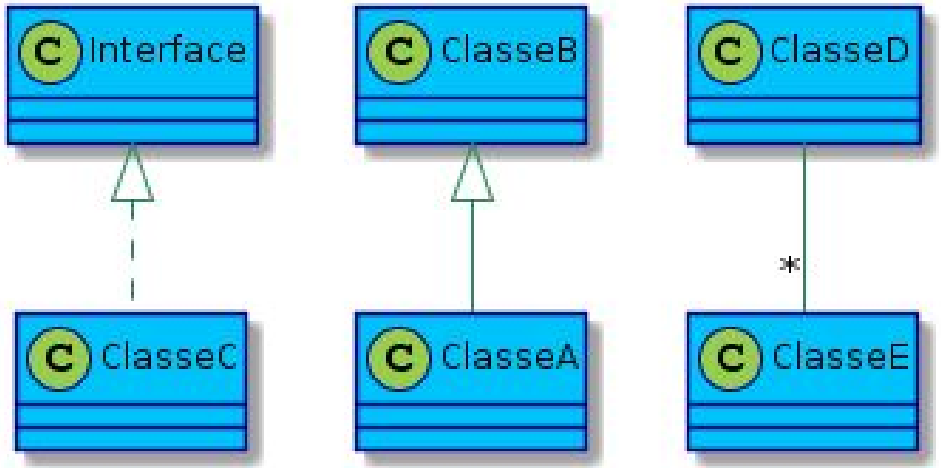
Menu du jour

- Le diagramme de `calc`
- Reprendre et approfondir les notions survolées dans le CM précédent et le Tp Calculette
- Vocabulaire

Diagramme de classes du package `calc`



Résumé - relations entre classes



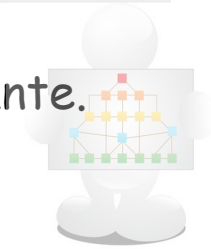
`ClasseC` implémente `Interface`.

`ClasseA` hérite de `ClasseB`.

Une instance de `ClasseD` connaît un nombre quelconque d'instances de `ClasseE`.

Attention au sens des flèches !

Hiérarchies de classes



- Ordonner les classes au sein d'arborescences, d'abstraction croissante.

- **Spécialisation** = Démarche descendante : séparer des objets suivant des caractéristiques plus spécifiques

  → on construit une sous-classe pour chaque sous-ensemble identifié.

- **Généralisation/ Factorisation** = Démarche ascendante : identifier des caractéristiques communes à des objets issus de classes différentes

  → on construit une super-classe pour regrouper ces caractéristiques (attributs et/ou fonctions)

Spécialisation

Adresse, AdresseMac

Client, Employe, Directeur, Gardien, *etc.*

Personne, Enseignant, Etudiant, Salarié, *etc.*

Généralisation

Expression, Binaire, Naire, Somme, Produit, Nombre, *etc.*

Polygone2D, Triangle, Carre, Rectangle, *etc.*

ElementDeJeu, ElementFixe, ElementQuiSeDeplace, Pingouin, Iceberg, Ours, *etc.*

Vehicule, Marin, AvecMoteur, Velo, SansMoteur,Trottinette, Moto, Voiture, *etc.*

Interface	Classe abstraite	Classe
Profils de fonctions publiques	Profils de fonctions (abstraites) Champs Constructeurs Textes complets de fonctions	Champs Constructeurs Textes complets de fonctions
Non instanciables	Non instanciables	Instanciables

Spécialisation

En général, toutes les classes de la hiérarchie décrivent des objets du monde.

→ elles sont toutes instanciables

Généralisation

En général, seules les classes feuilles de l'arborescence décrivent des objets du monde.

→ les classes feuilles sont instanciables

→ les autres ne sont pas instanciables (interfaces ou classes abstraites)

Les relations entre classes/interfaces

- Une classe hérite d'une classe, abstraite ou non (`extends`).
- Une classe implémente une interface (`implements`).
- Une interface hérite d'une interface (`extends`).

Le lien **`extends`** est unique.

Le lien **`implements`** peut être multiple.

Les relations entre classes/interfaces

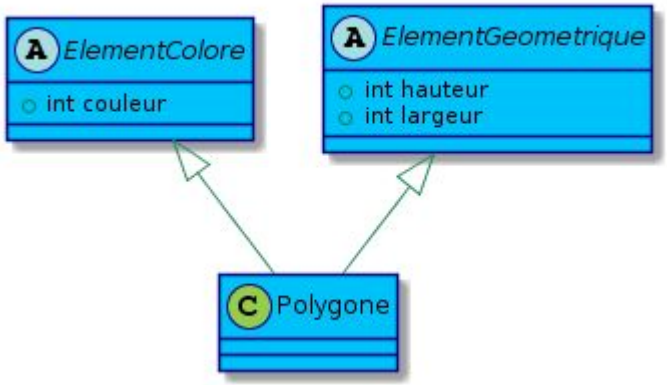
- Une classe hérite d'une classe, abstraites ou non (`extends`).
- Une classe implémente une interface (`implements`).
- Une interface hérite d'une interface (`extends`).

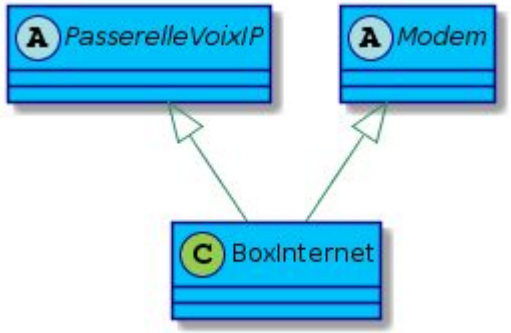
Le lien **`extends`** est unique.

Le lien **`implements`** peut être multiple.

Héritage multiple

C++ OCaml Eiffel





Ouvre la porte aux conflits entre les fonctions héritées des super-classes

La racine de l'arbre d'héritage

- Par défaut, toute classe hérite de la classe `java.lang.Object`.
- Fonctions de la classe Object

```
String toString()
boolean equals(Object o)
int hashCode()
```

La racine de l'arbre d'héritage

- Par défaut, toute classe hérite de la classe `java.lang.Object`.
- Fonctions de la classe Object

`String toString()` → adresse mémoire de l'objet

`boolean equals(Object o)` → même effet que ==

`int hashCode()` → adresse mémoire de l'objet

Les règles de l'héritage

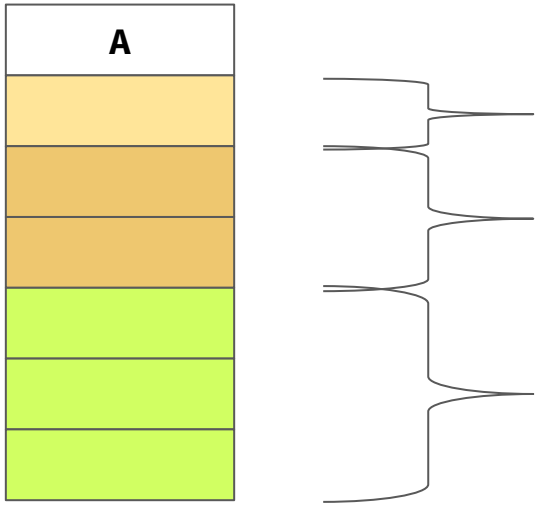
- Les champs/fonctions `public` et `protected` de la super-classe sont utilisables dans la sous-classe.
- Les constructeurs de la super-classe peuvent être appelés dans la sous-classe par l'intermédiaire de `super(...)`.
- En général le constructeur d'une classe abstraite est déclaré protected.
- La première instruction du constructeur d'une sous-classe doit être un appel à `super(...)`. En l'absence de celui-çi, le compilateur l'ajoute automatiquement.

Les règles de l'héritage ….

- Un client d'une classe ne peut utiliser que les constructeurs de la classe (et jamais ceux de la super-classe).

- Un client d'une classe peut utiliser toutes les fonctions `public` de la classe, ainsi que toutes celles des super-classes.

Les conséquences sur la représentation mémoire



Champs de C,
super-classe de B

Champs de B,
super-classe de A

Champs propres de A

La redéfinition

- Il arrive que la fonction héritée ne convienne pas aux instances de la sous-classes.

  → exemples de `toString`, `equals` et `hashCode` héritées de `Object`

- Il est possible de redéfinir la fonction dans la sous-classe, en conservant le même profil.

- Dans la sous-classe, il est possible d'exécuter la fonction héritée, en la préfixant par `super`.

La redéfinition

```
class Client {
  protected String nom ;
  public String toString() {
    return nom ;
  }
}
```

```
class Employe extends Client {
  private Date demb ;
  public String toString() {
    return nom + demb;
  }
}
```

```
class Employe extends Client {
  private Date demb ;
  public String toString() {
    return super.toString() + demb;
  }
}
```

Compatibilité de types

Affectation	Passage de paramètre
`x = y ;`	`o.f(y) ;`
Le type de déclaration de la variable y est identique à celui de la variable x.	Le type de déclaration du paramètre effectif y est identique à celui du paramètre formel x.

En dehors des types primitifs

Compatibilit

Affec

paramètre

```
x =
```

```
) ;
```

Le type de déc
variable y est ide
la varic

ion du paramètre
ntique à celui du
formel x.

Compatibilité de types (en présence d'héritage)

Affectation	Passage de paramètre
`x = y ;`	`o.f(y) ;`
Le type de déclaration de la variable y est identique à celui de la variable x.	Le type de déclaration du paramètre effectif y est identique à celui du paramètre formel x.
Le type de déclaration de la variable y est une sous-classe de celui de la variable x.	**Le type de déclaration du paramètre effectif y est une sous-classe de celui du paramètre formel x.**

Compatibilité de types …. exemples

```
Nombre n1 = new Nombre(177) ;
Nombre n2 = new Nombre(32) ;
Somme s1 = new Somme(n1, n2) ;
Binaire b1 = s1 ;
```

```
Expression e1 = new Nombre(177) ;
Expression e2 = new Nombre(32) ;
Expression e3 = new Somme(e1, e2) ;
e3 = n2 :
e2 = b1 ;
```

Compatibilité de types …. exemples

```
Nombre n1 = new Nombre(177) ;
Nombre n2 = new Nombre(32) ;
Nombre s1 = new Somme(n1, n2) ;
Expression e1 = new Nombre(177) ;
Binaire b1 = e1 ;
Somme s = new Nombre(-888) ;
```

Polymorphisme de variable

- A des moments différents de l'exécution, une variable peut contenir des instances de types différents.

```
Expression e ;
e = new Nombre(177) ;
e = new Somme(new Nombre(44), new Nombre(21)) ;
e = new Produit(new Nombre(18), e) ;
```

Polymorphisme de variable

- Type ***statique*** de la variable = type de déclaration

  → Unique

- Type ***dynamique*** de la variable (à un moment donné de l'exécution) = le type de l'instance dont l'adresse est dans la variable

  → Autant de types dynamiques possibles que de sous-classes du type statique

```
Binaire b ;
Expression e ;
```

Type statique ?
Types dynamiques possibles ?

Liaison dynamique

- Lors de l'exécution d'un appel de fonction de la forme `o.f(..)`, la fonction `f` exécutée est celle qui est définie dans le type dynamique du receveur `o`.

```
Expression e ;
…
int val = e.valeur() ;
```

- A opposer à la liaison statique (en C par exemple) : le compilateur sait exactement quelle fonction sera exécutée.

Opérateur cast

Pour forcer le type d'une expression

```
Polygone p ;
....
Carre c = (Carre) p ;
double cc = c.cote() ;
```

Opérateur instanceOf

Pour tester le type dynamique d'une variable

```
Expression e ;
....
if (e instanceOf Nombre) ….
```

```
class Point {
  protected double abs, ord ;
  double getAbscisse() { return abs ;}
  boolean equals(Object o) {
    if (o instanceOf Point) {
       Point p = (Point) o;
       return abs==p.getAbscisse() && ord == p.getOrdonnee();
    else return false ;
}}
```

Pour conclure

- Héritage, polymorphisme, redéfinition et liaison dynamique étroitement liés
- Mécanismes fondamentaux qui apportent toute leur puissance aux langages objets
- Apprendre à les utiliser

  → S'appuyer sur les Design Pattern