ENSAH

جامعة عبد المالك السعدي
Université Abdelmalek Essaadi

Programmation Orienté Objet

Langage Java

2ème Année Génie Informatique

Ecole Nationale des Sciences Appliquées – Al Hoceima

Prof A. Bahri
abahri@uae.ac.ma

A.U 2020/2021

Approches orientées objet (Rappel)

Approches classiques: Séparation nette entre les données et les procédures de traitement de ces données

➡ Au moindre changement des données, il faut modifier les procédures de manipulation de ces données

Approches orientées objet (Rappel)

Approches classiques : Difficulté de maintenance !!

➡ Solution : **Approche Orienté Objet**

Approches orientées objet (Rappel)

- Programme : ensemble d'objets indépendants communicant par envoi de messages
- Lorsqu'un objet désire une information d'un autre objet, il lui envoi un message
- Un objet regroupe des données et les procédures de manipulation de ses données
- Les messages sont le seul moyen de communication entre objets

Approches orientées objet (Rappel)

- Programme = ensemble d'objets
- Objet = données + procédures
- Communication = envoi de message

Approches orientées objet (Rappel)

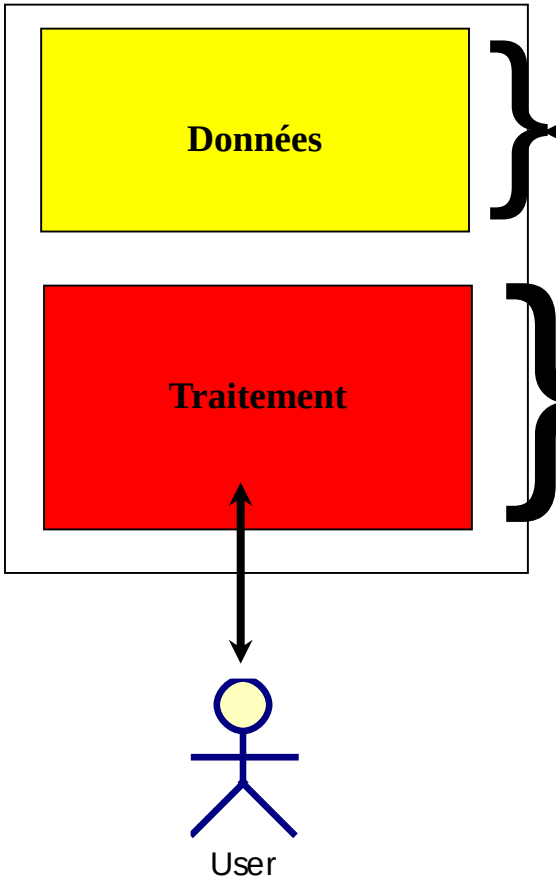
- L'orienté objet repose sur trois concepts de base :
  - Le concept de structuration en utilisant les objets et les classes
  - Le concept de communication par envoi de messages
  - Le concept de construction par affinage avec l'**héritage**.

Approches orientées objet (Rappel)

- Un objet est une entité
  - Autonome (ayant des limites bien définies)
  - Ayant une existence propre
  - Ayant un sens dans le domaine étudié

- Un objet regroupe à la fois
  - Des données (attributs)
  - Des procédures (opérations): Notion d'Encapsulation

Approches orientées objet (Rappel)



- Partie statique, passive
- Partie cachée, privée

- Partie dynamique, comportementale
- Partie visible, publique
- Interface avec l'extérieur

Approches orientées objet (Rappel)

- Plusieurs objets peuvent avoir des données et des comportements (opérations) en commun

➡ Il devient nécessaire de les regrouper dans un modèle général : **classe d'objets**

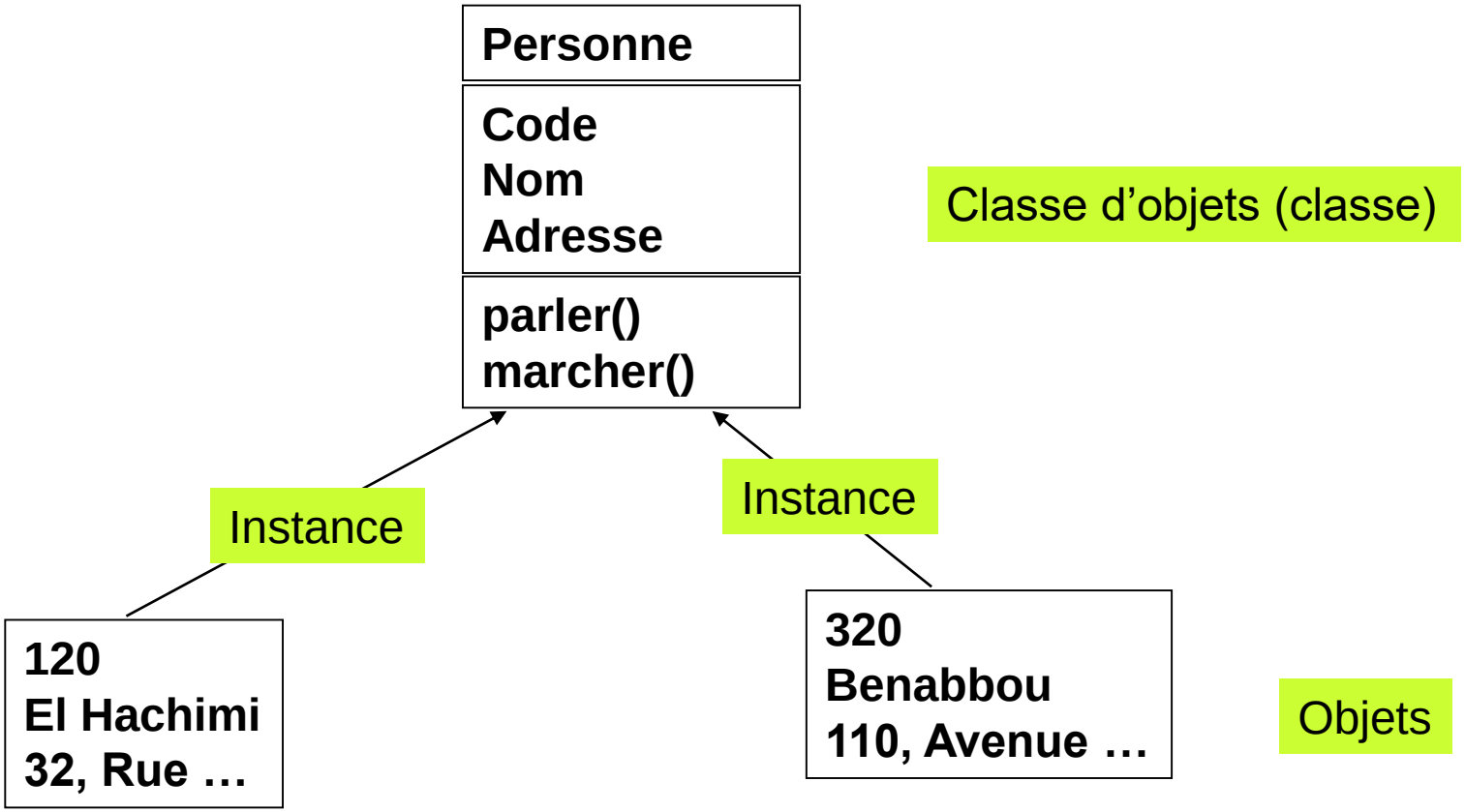
- Une classe est un ensemble d'objets ayant les mêmes données et les mêmes opérations

Approches orientées objet (Rappel)

- Une classe est un moule à partir de laquelle on crée (instancie) des objet

  - les objets: sont des instances d'une classe
  - L'instanciation : est la création d'un objet d'une classe

Approches orientées objet (Rappel)



Approches orientées objet (Rappel)

Remarques:

- Classe : modèle général
- Objets :
  - cas particuliers
  - Instances
  - occurrences

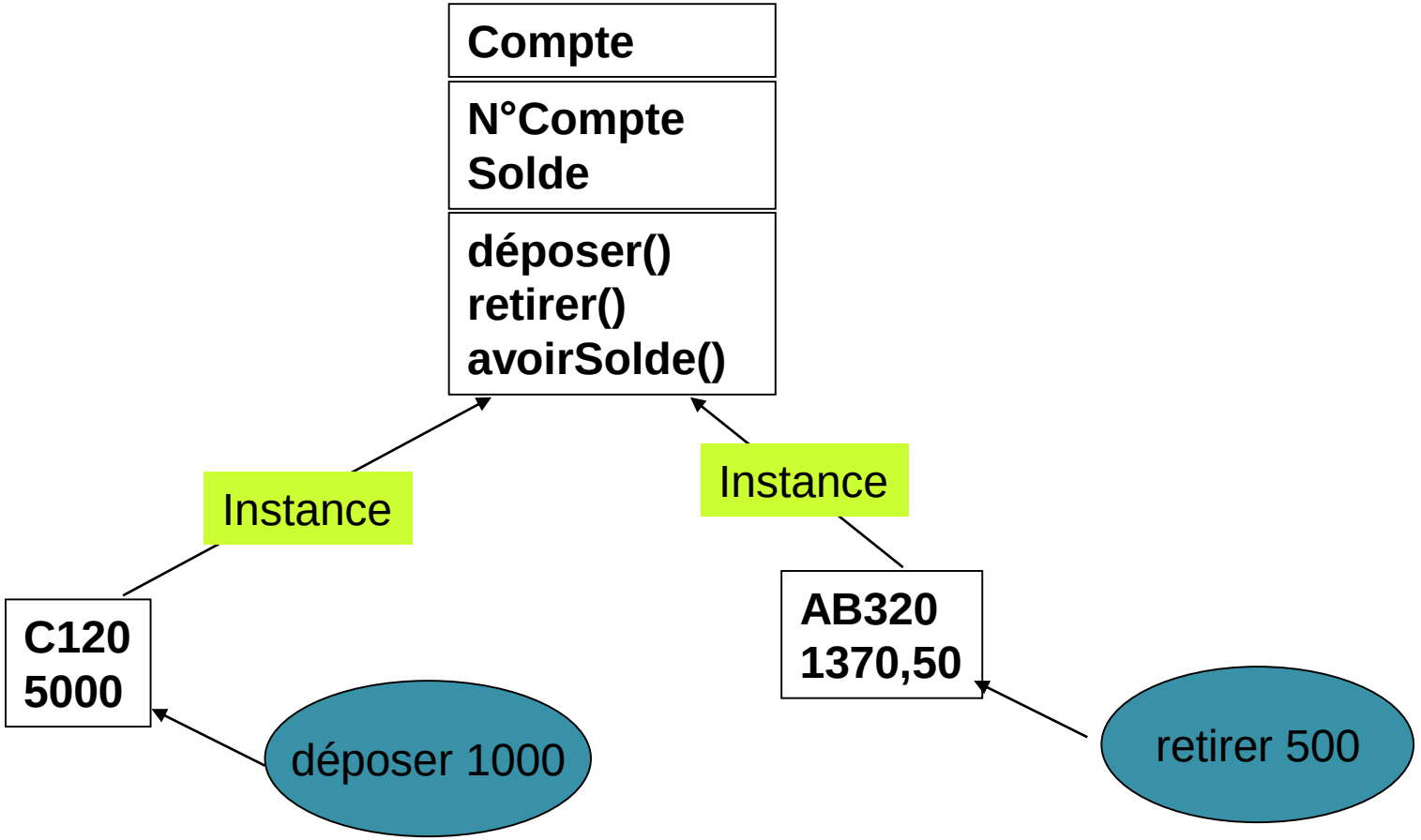
Approches orientées objet (Rappel)

- Les objets communiquent par envoi de messages

- Envoyer un message à un objet, c'est lui demander un service

- Lorsqu'un objet reçoit un message :
  - Soit le message correspond à un traitement défini dans la classe de l'objet auquel cas la méthode correspondante est exécutée.
  - Soit le message ne correspond pas, l'objet refuse le message et signale une erreur

Approches orientées objet (Rappel)



Généralisation / Spécialisation et héritage

- La généralisation est la relation entre une classe et une ou plusieurs de ses versions raffinées.

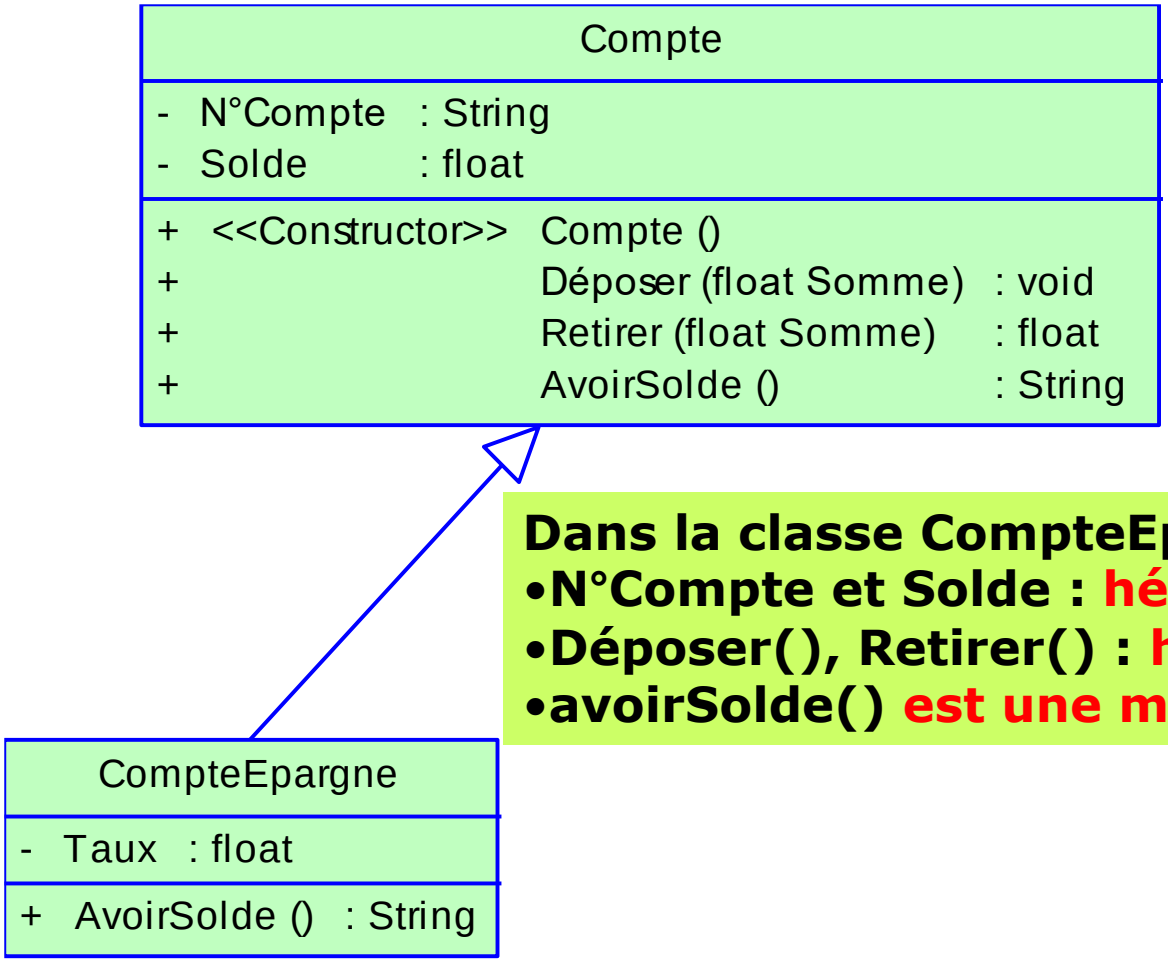
- On appelle la classe dont on tire les précisions la super-classe et les autres classes les sous-classes.

- C'est une relation de type « est un (is a) » ou « est une sorte de ».

Généralisation / Spécialisation et héritage

- La classe spécialisée (sous-classe):
  - hérite les méthodes et les attributs de la classe générale (super-classe)
  - peut ajouter ses propres attributs et méthodes.
  - peut redéfinir le comportement d'une méthode, mais pas des attributs.

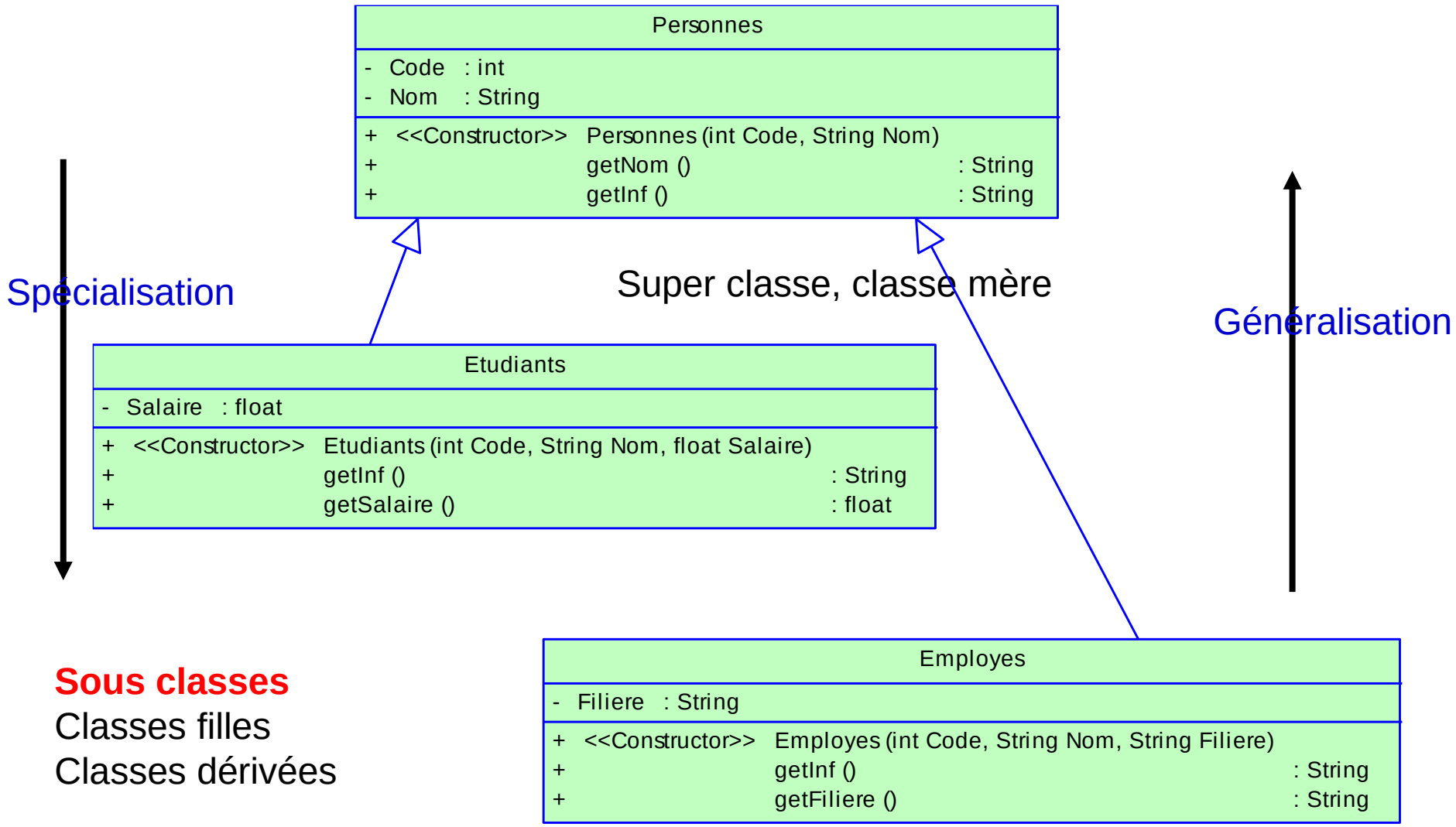
Généralisation / Spécialisation et héritage



Dans la classe CompteEpargne :

- **N°Compte et Solde : hérités**
- **Déposer(), Retirer() : héritées**
- **avoirSolde() est une méthode redéfinie**

Généralisation / Spécialisation et héritage



Sous classes
Classes filles
Classes dérivées

Interprétation des messages

- Lorsqu'un objet reçoit un message :
  - Si sa classe contient une méthode de même nom, il sera exécutée
  - Sinon, on remonte dans la hiérarchie à la recherche d'une méthode de même nom.

Interprétation des messages

Exemple:

- Soit CE une instance de la classe CompteEpargne :
  - CE.déposer() : fait appel à la méthode ‘déposer’ de la classe ‘Compte’
  - CE.avoirSolde() : fait appel à la méthode ‘avoirSolde’ de la classe ‘CompteEpargne’

Approches orientées objet (Rappel)

Remarques:

- La généralisation et la spécialisation sont deux façons pour voir la même relation, top-down (spécialisation) ou bottom-up (généralisation).
- L'héritage est l'implémentation de la relation de la généralisation/spécialisation.
- Une classe Java ne peut hériter que d'une seule classe.

Création d'une classe

```
Class Compte{
//déclaration des données (privées)
        private String NCompte;
        private float Solde;

//Déclaration des méthodes (publics)
public Compte(String n, float s){NCompte=n;Solde=s;}
public void deposer(float somme){Solde+=somme;}
public void retirer(float somme){
        if (Solde<somme)
           System.out.println(«Solde insuffisant ! »);
        else
            Solde-=somme;
}
public float avoirSolde(){return Solde;}
}
```

Création d'une classe

- Les données sont généralement privées.
- Les méthodes sont généralement publiques: (interface)
- Le constructeur est une méthode spéciale :
  - Porte le même nom que la classe
  - Utilisée pour initialiser les objets
  - Ne retourne aucune valeur, même void
  - Invoquée implicitement au moment de la création des objets
  - On peut avoir plusieurs constructeurs

Création d'objets (instances)

```
class Test{
    public static void main(String arg[]){
        Compte c1=new Compte("C120",5000);
        Compte c2;
        c2=new Compte("AB320",1370.50);
        float s=c1.Solde;        Erreur (solde donnée privée)
        float s1=c1.avoirSolde();
        System.out.println("Solde "+s1);
        c2.deposer(500);
        System.out.println("Solde "+c2.avoirSolde());
       …
    }
}
```

Champs final

- Il est possible de déclarer un champs constant (final)
- Les constantes sont généralement définies en plus public et static afin d'être accessibles depuis l'extérieur de la classe et directement par l'intermédiaire du nom de celle-ci sans avoir besoin de créer une instance de la classe.

```
class Constante {
    public static final int CONST1=50 ;
    public static final double PI=3.14 ;
}
```

Création d'objets (instances)

- Le mot clé **this** désigne l'objet courant (l'objet qui fait appel à la méthode)

- Exemple:

```
public void deposer(float somme){
/*augmenter le solde de l'objet faisant appel à la méthode déposer()*/
      this.Solde+=somme;
}
…
Compte c1=new Compte(100,3500)
c1.deposer(2000); //this indique l'objet c1
```

Création d'objets (instances)

- Remarque

  Le mot clé this est obligatoire lorsqu'il y a conflit entre un attribut de classe et un argument de la méthode utilisant cet attribut

- Exemple :

```
public Compte(String NCompte, float Solde){
   this.NCompte=NCompte;
   this.Solde=Solde;
}
```

Argument de la méthode

Attribut de la classe

Données de classes/d'instances

- Une donnée (attribut ou méthode) peut être :
  - Donnée d'instance, propre à l'objet
  - Donnée de classe, partagée entre tous les objets de la classe

Données de classes/d'instances

- Une donnée d'instances:
  - Attribut : chaque objet possède sa propre valeur.
  - Méthode : invoquée en faisant référence à l'objet.

Données de classes/d'instances

- Une donnée de classe (déclaration précédée du mot clé "static ")
  - Attribut : partagée entre tous les objets de la classe.
  - Méthode : invoquée en précisant la classe à la place de l'objet

Données statiques

```
class Personne{
//attributs d'instances
      private int Code;
      private String Nom;
//attribut de classe (statique)
      private static int NbrePers;
      …
}
```

Données statiques

```
class Personne{
        …
    public Personne(int code, String nom){
        NbrePers++;
        Code=code;Nom=nom;
    }
    public String getInf(){
        return "Code "+Code+"\n Nom "+Nom;
    }
    public void setCode(int code){
        this.code=code;
    }
    public static int getNbre(){
        return NbrePers;
    }
}
```

Données statiques

```
class Test{
    public static void main(String arg[]){
       Personne p=new Personne(10, "A. B");
       Personne q=new Personne(20, "I. I");
       System.out.println(p.getInf());
       System.out.println(Personne.getNbre());
       System.out.println(q.getInf());
       System.out.println(Personne.getNbre());
}
```

Différents types de méthodes

Classification des méthodes d'un objet

- Constructeur
  - Initialise l'objet
- Accesseur
  - Getter (getInf)
    - Export les données (souvent partiellement)
  - Setter (setCode)
    - Import les données (en les vérifiant)
- Méthode métier (business method)
  - Effectue des calculs en fonction des données (retirer)

Différents types de méthodes

❑ Accesseurs:

Les accesseurs sont des méthodes permettant l'accès à des attributs

❑ Intérêt des accesseurs:

Changer l'implantation sans changer le code de l'appel

```
Personne p= new Personne(25, "ali ");
System.out.println(p.getCode(), " ",p.getNom())
```

```
class Personne{
    . . .
    public int getCode(){
        return code;
    }
    public String getNom(){
        return nom;
    }

    public void setCode(int
        code){
        this.code=code;
    }
    public void setNom(String s){
        nom=s;
    }

    . . .
}
```

Notion d'héritage

L’héritage est le mécanisme de partage des informations entre objets tout en préservant leurs différences

Notion d'héritage

- En Java, toutes classes héritent de java.lang.Object, directement ou indirectement.
  - Directement : si l'on déclare une classe sans héritage, le compilateur rajoute extends Object.
  - Indirectement : si l'on hérite d'une classe celle-ci hérite de Object (directement ou indirectement)
- Quelques méthodes de la classe Object:
  - toString(), clone(), equals(java.lang.Object) , ….
  - **getClass()** : méthode de la classe Object retournant un objet de type Class qui représente la classe de l'objet appelant.
  - **getName():** méthode de la classe Class permet d'obtenir le nom de la classe sous forme d'une chaine de caractère.

```
String s= p1.getClass().getName();
System.out.println("Je suis un nouvel objet de la classe " +s);
```

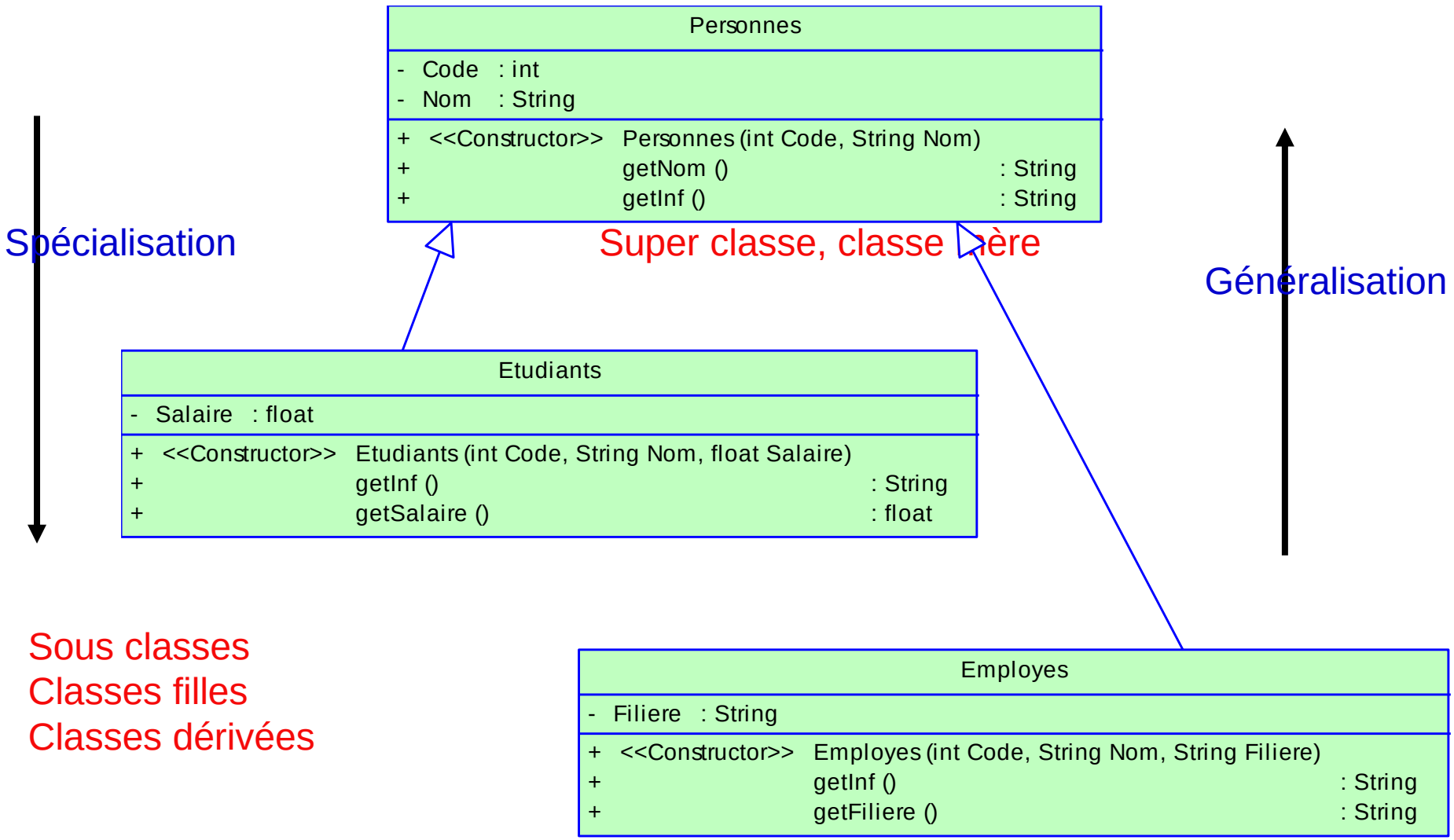
Notion d'héritage

Personnes
- Code : int - Nom : String
+ <<Constructor>> Personnes (int Code, String Nom) + getNom () : String + getInf () : String

Etudiants
- Code : int - Nom : String - Salaire : float
+ <<Constructor>> Etudiants (int Code, String Nom, float Salaire) + getNom () : String + getInf () : String + getSalaire () : float

Employes
- Code : int - Nom : String - Filiere : String
+ <<Constructor>> Employes (int Code, String Nom, String Filiere) + getNom () : String + getInf () : String + getFiliere () : String

Notion d'héritage



Notion d'héritage

```
class Personne{
	private int Code;
	private String Nom;
	public Personne(int code, String nom){
		Code=code; Nom=nom;
	}
	public String getNom(){
		return Nom;
	}
	public String getInf(){
		return "Code :"+Code +"\nNom : « + Nom;
	}
}
```

Notion d'héritage

```
class Employe extends Personne{
      private float Salaire;
      public Employe(int code, String nom, float salaire){
            super(code, nom);
            Salaire=salaire;
      }
      public float getSalaire(){
            return Salaire;
      }
      public String getInf(){
            String inf=super.getInf();
            inf=inf + "\n Salaire :"+Salaire;
            return inf;
      }
}
```

Notion d'héritage

```
class Etudiant extends Personne{
	private String filiere;
	public Etudiant(int code, String nom, String filiere){
		super(code, nom);
		filiere=filiere;
	}
	public String getFilere(){
		return filiere;
	}
	public String getInf(){String inf=super.getInf();
		inf=inf + "\n Filiere :"+filiere;
		return inf;
	}
}
```

Notion d'héritage

- Trois type de visibilités de données
  - Privée : donnée visible que par les méthodes de la classe.
  - Protégée : donnée visible que par les méthodes de la classes et des sous classes directes
  - Publique : donnée visible depuis n'importe quelle méthode

Constructeur vs Méthode

- La grosse différence entre un constructeur et une méthode, c'est que l'on n'hérite pas des constructeurs,

- Le constructeur de la classe hérité a pour mission de :
  – demander l'initialisation de la classe mère au travers de son constructeur
  – d'initialiser ses propres champs

- Une sous-classe hérite des méthodes de la superclasse

Assurer la redéfinition

- L'annotation @Override utilisée lorsqu'une méthode redéfinit la méthode de la super classe, tout en sachant que ceci n'est pas obligatoire !
- L'annotation @Override indique au compilateur de générer une erreur si une méthode ne redéfinit pas une autre

```
class Etudiant extends Personne{
    . . .
    @Override
    public String getInf(){String
        inf=super.getInf();
        Inf=inf + "\Filiere :"+Filiere;
        return Inf;
    }
    . . .
}
```

Redéfinition vs Surcharge

- La surcharge correspond à avoir des méthodes de même nom mais de profils différents dans une même classe
- Une méthode est redéfinie ssi elle a la même signature que l'originale.
- Si la signature est différente, il s'agit d'une simple surcharge : pas de mécanisme de liaison tardive.
- Pour éviter des erreurs , ajouter l'annotation @Override.
  - le compilateur vérifie qu'il s'agit bien d'une redéfinition,

Résumé

- Pour qu'une classe hérite d'une autre, on utilise le mot extends.
- Une classe peut hériter d'une autre tout sauf les constructeurs.
- Il faut toujours définir le constructeur de la nouvelle classe dérivée.
- Le constructeur de la nouvelle classe dérivée peut faire appel au constructeur de la classe parente en utilisant le mot super suivi des valeurs des arguments entre parenthèse.
- Lors de l'héritage, la nouvelle classe dérivée peut redéfinir les méthodes héritées de la classe parente.
- Dans une classe, on peut surcharger une méthode : On peut créer plusieurs méthodes qui ont le même nom, mais avec des signatures différentes.
- Dans une classe on peut également surcharger le constructeur : On peut créer plusieurs constructeurs avec des signatures différentes.
- Si vous ne définissez pas un constructeur dans votre classe, le compilateur java définit un constructeur par défaut.
- La notation super. permet d'avoir accès aux membres non static de la superclasse
- Si une méthode est déclarée final, celle-ci ne peut être redéfinie
- Si une classe est déclarée final, il est impossible de créer des sous-classes (mêmes raisons que pour les méthodes)

Exemples

- Quels résultats fournit ce programme ?

```
public class Test{
   public static void main (String args[]){
        double x1 = 1e200, x2 = 1e210 ;
        double y, z ;
        y = x1*x2 ;
        System.out.println ("valeur de y " + y) ;
        x2 = x1 ;
        z = y/(x2-x1) ;
        System.out.println (y + " divise par " + (x2-x1) + " = " + z) ;
        y = 15 ;
        z = y/(x2-x1) ;
        System.out.println (y + " divise par " + (x2-x1) + " = " + z) ;
        z = (x2-x1)/(x2-x1) ;
        System.out.println ((x2-x1) + " divise par " + (x2-x1) + " = " + z) ;
        System.out.println (z + "+1 = " + (z+1)) ;
        x1 = Float.POSITIVE_INFINITY ;
        x2 = Double.NEGATIVE_INFINITY ;
        z = x1/x2 ;
        System.out.println (x1 + "/" + x2 + " = " + z) ;
    }
}
```

Classes Abstraites et interfaces

En Java, il existe 3 types d'entités qu'on peut manipuler :

- Les classes (déjà vues)
- Les classes abstraites présentées par le mot clé abstract
- Les interfaces (Il ne s'agit pas ici d'interfaces graphiques!!!)

Classes Abstraites

- Dans une classe abstraite, le corps de quelques méthodes peut ne pas être défini (on déclare uniquement le prototype de la méthode). Ces méthodes sont dites des méthodes abstraites.

```
abstract class NomClasse{

    . . .

}
```

- Une méthode abstraite est aussi présentée par l'intermédiaire du mot clé abstract. C'est aux classes dérivées de redéfinir ces méthodes et de préciser leur comportement.

```
abstract class NomClasse{
    abstract type nomDeMéthode( paramètres) ;
    . . .
}
```

Classes Abstraites

- Une classe abstraite ne peut donc jamais être instanciée. Il s'agit d'une spécification devant être implémentée par l'intermédiaire d'une classe dérivée. Si cette dernière définit toutes les méthodes abstraites alors celle-ci est instanciable.
  - new C(...); // illégal (C : classe abstraite)
- manipulation de référence possible

```
public class D extends C { ... } // classe concrète
C refC = new D(); // légal
```

- Une classe abstraite (présentée par le mot clé abstract) peut ne pas contenir de méthodes abstraites.
- Une classe contenant une méthode abstraite doit obligatoirement être déclarée **abstract**.

Exemples

Le programme ci-dessous implémente une petite hiérarchie de 4 classes (A, B, C et D). Il y a quelques erreurs dans le programme. Toutes les erreurs sont dûmes à une utilisation erronée des modificateurs abstract et final. Expliquez ces erreurs.

```
class AbstractFinal {
    public static void main(String[] args) {
        A x = new A();
        B y = new B();
        C z = new C();
        y.b = 2;
        z.c = 3;
    }
}
abstract class A {
    int a;
  abstract int ma();
}
class B extends A {
    int b;
}
```

```
class C extends A {
    final double c = 1;
}

abstract class D extends A {
    double d;

    int operation(int a) {
        return (a * 2);
    }
    abstract int calcul(int b) {
    }
    abstract int machin();
}
```

La classe abstrait A est instanciable, Définir l'implementation de la fonction ma() dans B , C et dans D, dans D il ne faut pas définir le corps de la fonction abstraite calcul(in b),

Interfaces

- Les interfaces qui sont définies par l'intermédiaire du mot clé **interface** au lieu de **class** constituant un cas particulier des classes abstraites : d'autre part, ce sont des classes où aucune méthode n'est définie (uniquement le prototype de chaque méthode).

```
interface NomInterface{

        type1 methode1 (paramètres) ;

        type2 methode2 (paramètres) ;

           . . .

}
```

- L'extension d'une interface est appelée **implémentation** et elle est réalisée par l'intermédiaire du mot clé **implements** :

```
class NomDeClass implements NomInterface{
    ...
}
```

Interfaces

Remarques:

- Une classe qui implémente une interface doit définir toutes les méthodes de l'interface.
- Impossible de définir une méthode statique dans une interface
- Une interface peut aussi contenir des attributs.
- Toutes les attributs d'une interface doit obligatoirement être initialisés.
- Tous les attributs d'une interface sont **public**, **static** et **final**.

Héritage Multiple

- Problèmes de l'héritage multiple :
  - Si on hérite de deux méthodes ayant même signature dans deux super classes, quelle code choisir ?

- Solution :
  - Il n'y a pas d'héritage multiple en Java
  - Java définit des interfaces et permet à une classe d'implanter plusieurs interfaces

```
class I extends B implements C{

        . . .

}

Class k extends C implements E, F{

        . . .

}
```

L'ordre des déclarations est PRIMORDIAL. Vous DEVEZ mettre l'expression d'héritage AVANT l'expression d'implémentation, SINON votre code ne compilera pas !

Héritage Multiple

Héritage d'interface

- Une interface peut hériter d'une ou plusieurs interfaces
- Les méthodes de cette interface correspondent à l'union des méthodes des interfaces héritées

```
interface I implements A, B, C{

        . . .

}
```

Exemples

- Supposons que l'on a défini trois classes Film, LongMetrage, Documentaire. Dans une autre classe trois méthodes sont écrites avec un argument de chacune de ces classes :
  - int duree(Film a);
  - String aLAffricheDe(LongMetrage b);
  - String sujet(Documentaire c);
- Comment faire pour passer à duree un argument de type Film ou de type Documentaire?