# Classes et Objets

Maria Virginia Aponte

CNAM-Paris

27 février 2018

# Concepts fondamentaux en POO

# Les données en programmation

- Données élémentaires : donnée simple.
  Ex : un entier, nombre à virgule, caractère, booléen, etc ;

- Données structurées : plusieurs données mises ensemble (types hétérogènes).
  - tableaux, Strings (même type) ;
  - enregistrements, objets (types $\neq$).

  Utilité :
  - regrouper plusieurs variables reliées ;
  - structurer/faciliter la programmation et la réutilisation !

# Les objets « en 1 transparent »

Ils possèdent :

- des données locales (état interne) ;
- des opérations agissant sur ces données locales (méthodes) ;

Ils sont crées à partir :

- d'une classe ou type ;
- via l'opération `new`

**Appliquer les méthodes**

La méthode `m` d'un objet `o` s'applique **sur** celui-ci par `o.m()`, et non pas par `m(o)`;

# 1er exemple : objets de type String

```
String s = "Bonjour";
```

- données locales (état interne) $\Rightarrow$ caractères de la chaîne accessibles par indice : 'B' ,'o' ,'n' , 'j', 'o', 'u', 'r'
- opérations sur données locales (méthodes) $\Rightarrow$
  - `charAt(int)`, `equals(String)`, `equalsIgnoreCase(String)`, `substring(int)`, etc.

## Appliquer les méthodes

```
char c = s.charAt(0); // 1er carcatere
bool b = s.equalsIgnoreCase("BONjour"); // true
```

# Des objets comptes bancaires

Données (locales) d'un compte :

- nom titulaire, numéro du compte, solde courant ;
- historique dernières opérations.

$\Rightarrow$ plusieurs données de types $\neq$.

Opérations sur (les données locales) d'un compte :

- initialiser : nom titulaire, numéro compte, solde ;
- obtenir solde courant, réaliser retrait, dépôt ;
- consulter historique des opérations . . .

# Objets : données structurées (ou complexes)

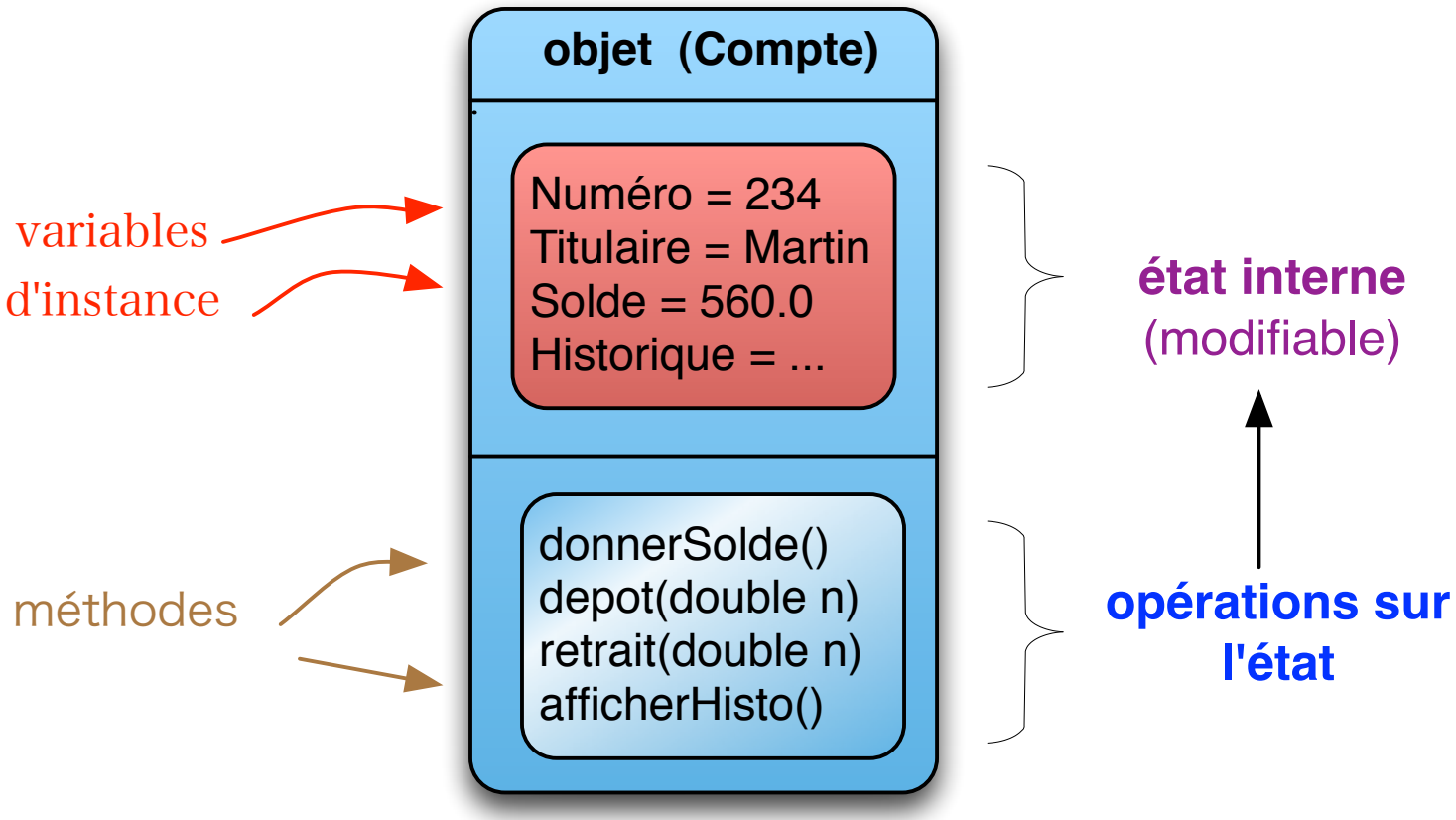
**Objet**

collection de variables internes et d' opérations *agissant localement* sur les variables internes.

Un objet compte bancaire contient

- état interne : (**données locales**) : variables nom du titulaire, solde courant, historique d'opérations .
- opérations : ( **méthodes**) applicables **sur** l'état interne de l'objet. Ex : retrait, depôt, etc.

# Un objet Compte




**Objet = état interne + opérations sur l'état**

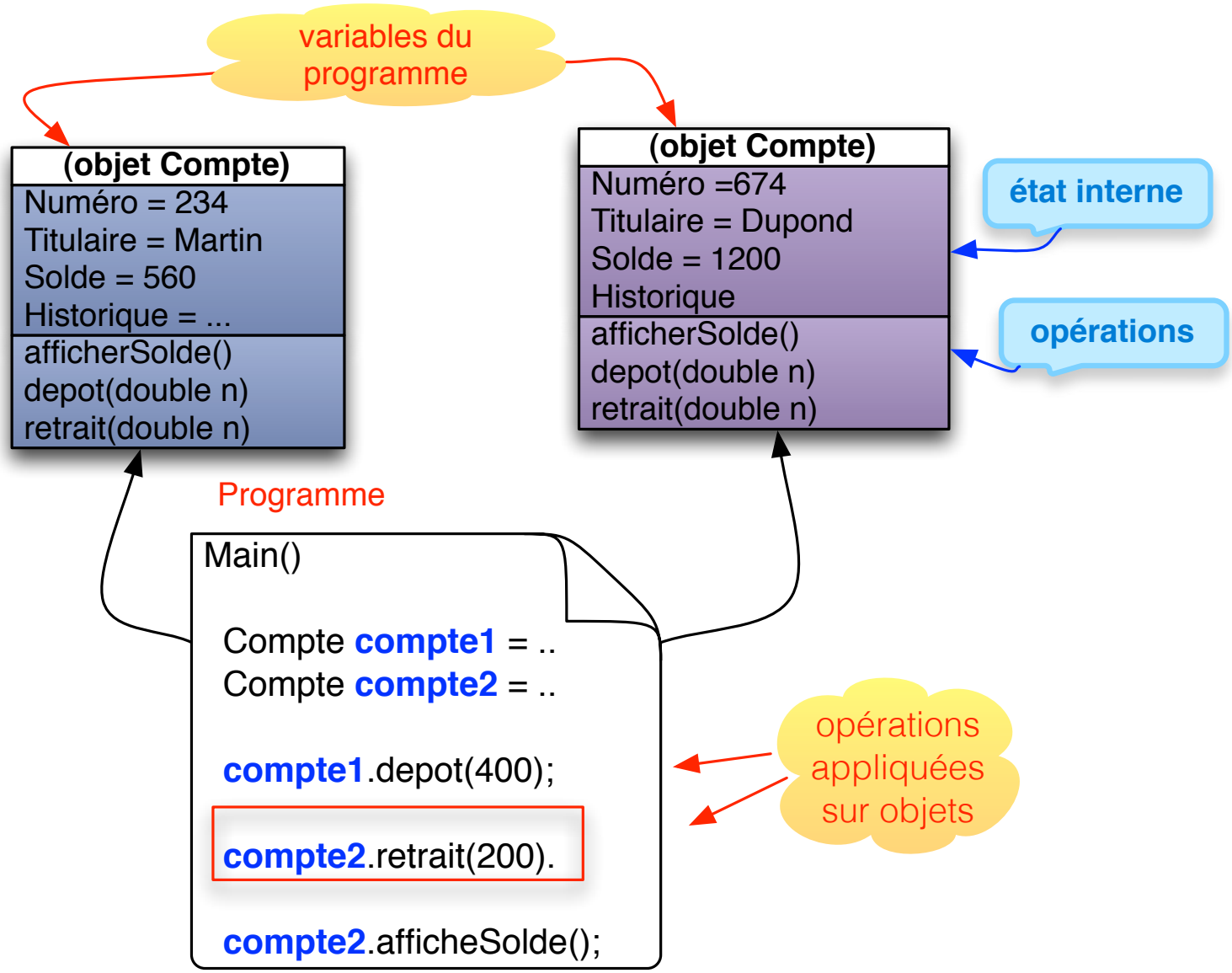
# Gestion de comptes en POO

## Programme OO

les objets sont les données du programme

- le programme manipule plusieurs objets Compte,
  - chaque objet possède un état interne (numéro de compte, solde) propre.
- tous les objets ont :
  - une structure commune (titulaire, numéro, solde).
  - un ensemble commun d'opérations (retrait, depôt, ..)

# Un programme OO



# Classe = Moule à objets + Type

Un objet est crée à partir d'une classe (même moule) :

```
Compte c = new Compte(1002, 60.0);
```

On dira : *c' est une instance de la classe Compte*.

`Compte` est :

- le nom de la classe (constructeur) pour créer l'objet c ;
- le type pour déclarer la variable c ;

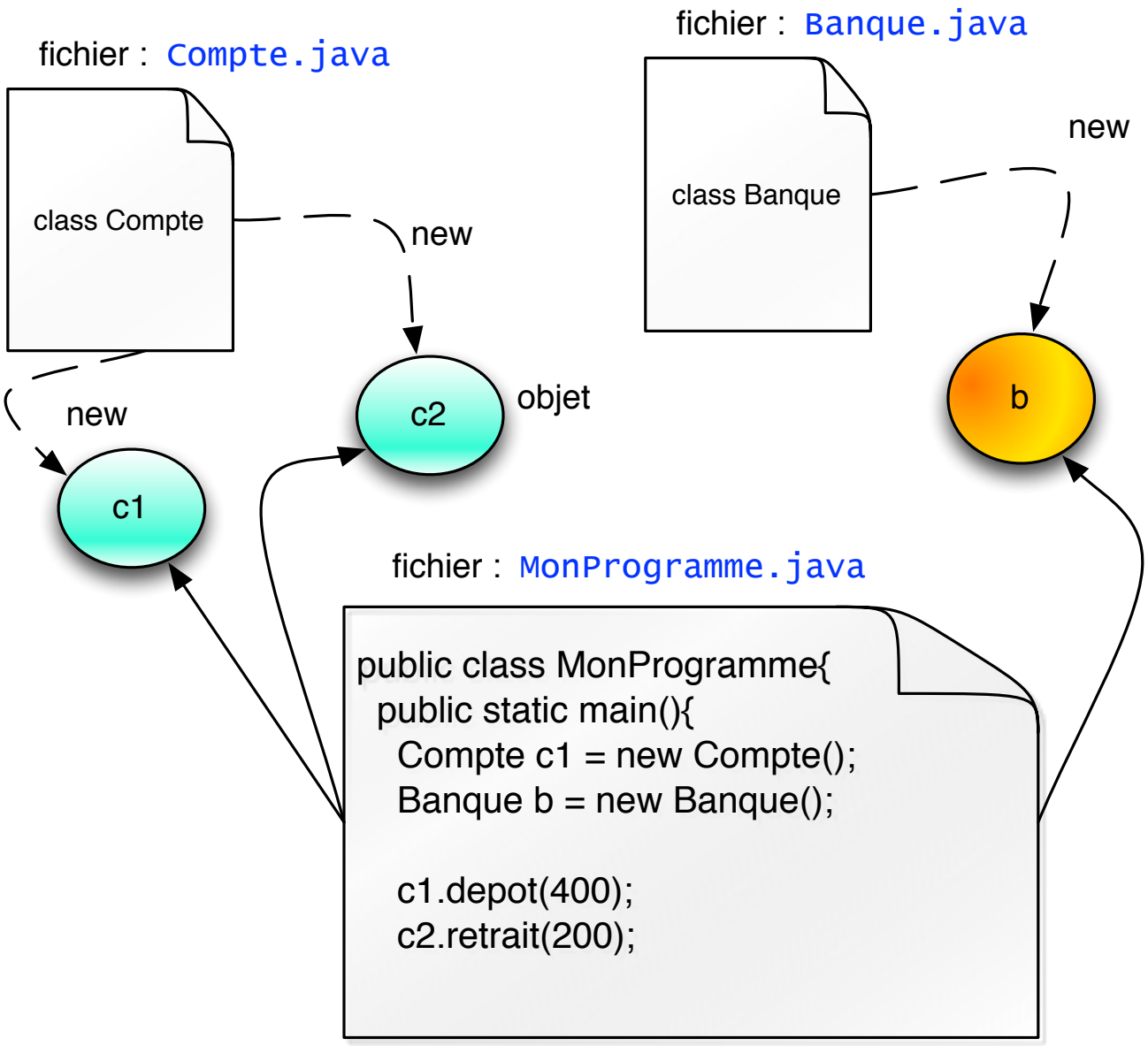
**Classe ⇒ "moule à objets" + type**

- décrit structure interne + opérations des objets à créer ;
- devient le type des objets crées ;

# Deux sortes de classes

- Classes "types" ou "moules à d'objets" :
  - servent à créer et initialiser des objets.
- Classes "programmes" :
  - possèdent une méthode `main()` avec le programme à exécuter et des variables objet ;
  - utilisent les classes "types" pour déclarer les types des variables objet,
  - et pour créer et initialiser les objets dans ces variables.

# Un programme OO



# Classes et programmes

Il faut les deux sortes pour écrire un programme orienté objet :

- **une ou plusieurs classes "type des données"** $\rightarrow$ modéliser chaque sorte d'objet : comptes, personnes titulaires de comptes, banque..

- **une unique classe "programme"** $\rightarrow$ main() création/utilisation d'objets à partir des classes "types des données".

# Exemple de classe (type) Compte

```
class Compte {
    int numero;      // Etat interne
    double solde;
    String titulaire;

    double getSolde() { return solde;  }
    void depot(double n){ solde = solde+n; }
    void retrait(double m) { solde = solde-m;  }
    void affiche(){
        System.out.println("Numero: "+ numero);
        System.out.println("Titulaire: "+titulaire);
        System.out.println("Solde: "+ solde);
    }
}
```

# Exemple (classe-programme) utilisant classe Compte

```
public class TestComptes{
  public static void main(String[] args){
    // Declaration
    Compte c1, c2, c3;
    // Creation et initialisation
    c1 = new Compte();
    c2 = new Compte();

    // Modification variables internes
    c1.numero =  123456,
    c1.titulaire = "Paul Durand";
    c1.solde = 1000.00;

   // Utilisation
    c1.depot(100.00);
    c1.affiche();
   }}
```

# Création d'objets instance d'une classe

## Déclarationd'une variable de type objet :

```
Compte c1;  // l'objet n'existe pas
```

## Création, initialisation (**en mémoire**)

de l'objet "mis" dans la variable :

```
c1 = new Compte();  // l'objet existe
```

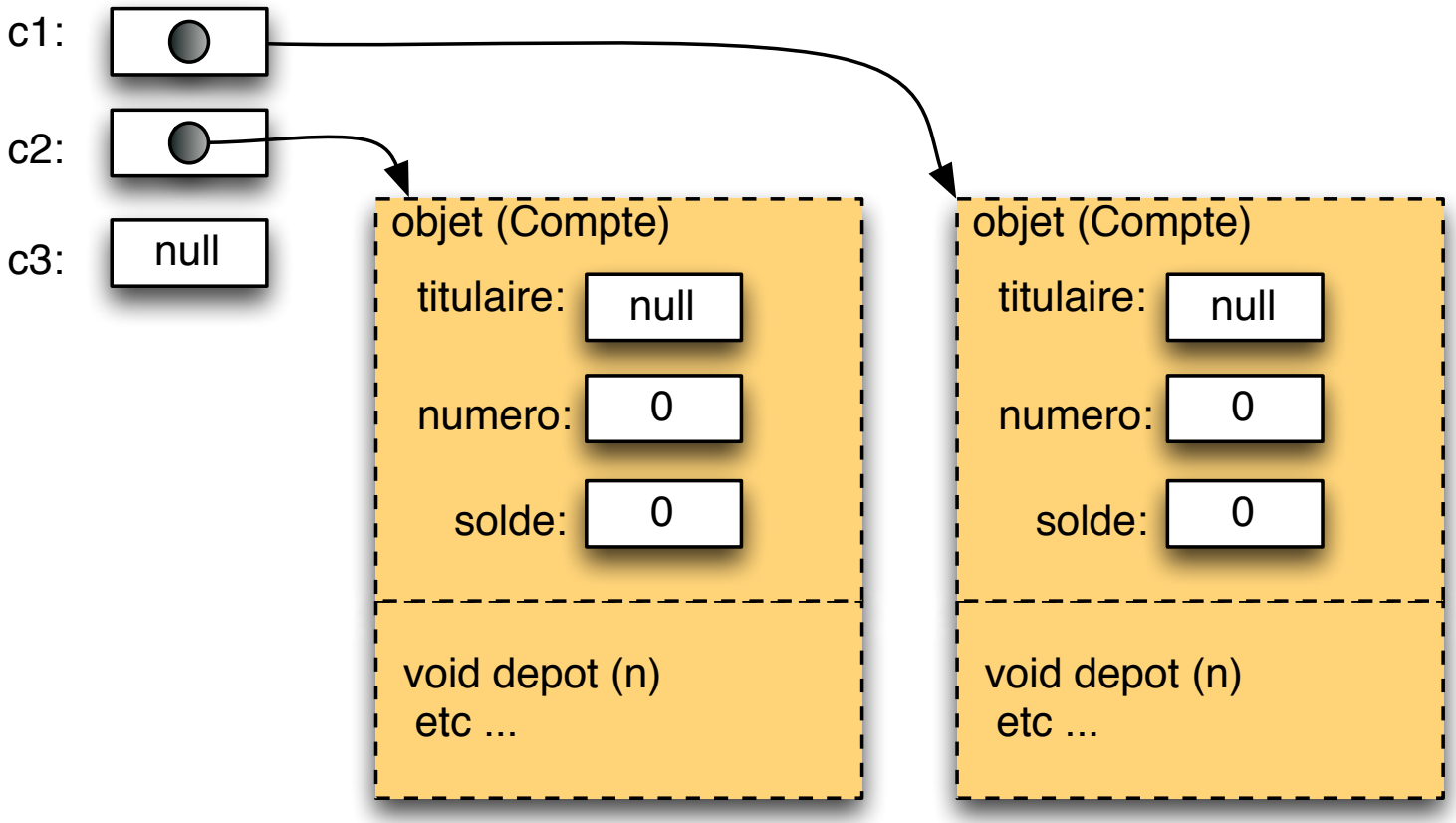
## Après création

variables internes de c1 $\Rightarrow$ initialisées (défaut ou constructeur).

# Aperçu des objets en mémoire

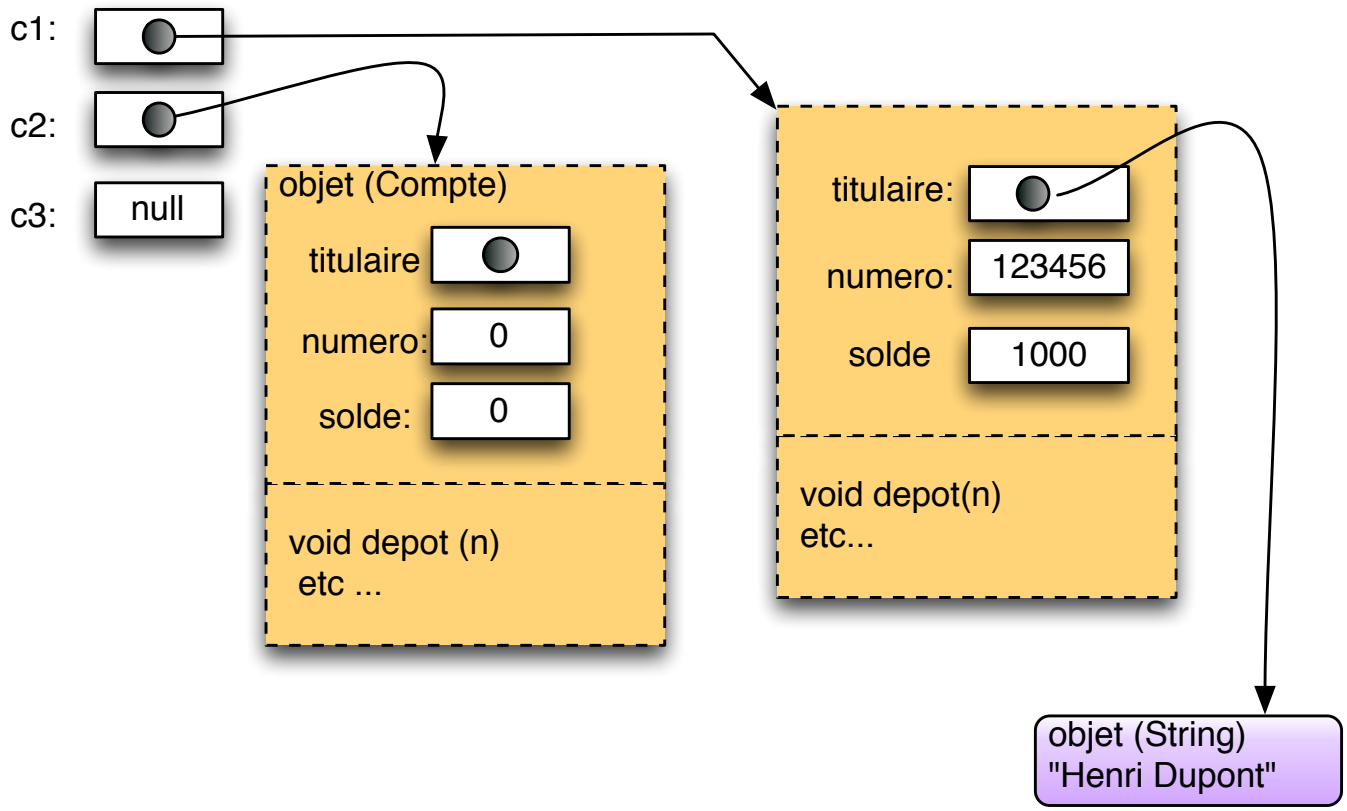
# Les objets du programme après création

```
Compte c1, c2, c3;
c1 = new Compte();
c2 = new Compte();
```



# Après modification état de c1

```
c1.numero =  123456,
c1.titulaire = "Henri Dupont";
c1.solde = 1000.00;
```



# Définir une classe (type)

# Classes : quelles composantes définir ?

1. Variables d'instance :
   $\rightarrow$ **données locales (état interne)** de l'objet ;

2. Méthodes :
   $\rightarrow$ **opérations** disponibles sur l'objet
   - $\Rightarrow$ exécutés sur l'état interne d'un objet (état possiblement modifié).

3. Constructeurs : pour créer les objets (via `new`) ;

# Exemple : définir une classe Compte

```
public class Compte{
  /* Variables d'instance */
  String titutaire ;
  int numero ;
  double solde;

  /* Méthodes */
  public double getSolde (){ return this.solde;}
  public void depot(double montant){
    this.solde = this.solde + montant ;
  }
  public void retrait(double montant)
    this.solde = this.solde - montant ;
  }
}
```

# Déclarer les variables d'instance

## Syntaxe

*Accessibilite Type nomVar = valeurInitiale;*

Exemple :

```
private String nomTitulaire;
private int numero;
private double solde = 0;
```

- création : si aucune valeur donnée ⇒ initialisées par défaut ;
- visibles par toutes les méthodes non statiques de la classe via la notation `this.nom-variable-instance`.
- `private` est un « modificateur d'accéssibilité » pour ces variables.

# Variables d'instance = état interne

- Définissent l'état interne de chaque objet :
  - initialisées à la création (défaut ou constructeur),
  - visibles par toutes les **méthodes de la classe** par notation `this.nom-variable-instance`.

- chaque objet peut lire/modifier ses variables (état) :

```
c1.depot(50); // +50 sur variable c1.solde
c2.depot(30); // +30 sur variable c2.solde
```

  - `c1.depot(50)` change variables instance de l'objet de l'appel (ici c1) ;
  - l'objet de l'appel est « l'objet courant »

- visibilité possible depuis l'extérieur de la classe (fortement déconseillée).

# Méthodes dans les classes à objets : statiques ou non

- non statiques : (ou « d'objet »)

```
public void depot(double montant){
    this.solde =  this.solde + montant ;
 }
```

  - pas de mot clé `static`
  - modélisent les comportements possibles de l'objet ;
  - agissent sur l'état (données) de l'objet courant.
  - invoquées sur un objet particulier : `c.depot(montant)`

- statiques : (ou « de classe »)
  - n'ont pas accès à l'état interne.
  - invoquées sur le nom de la classe qui la déclare : `Compte.nomMethodeStatique(...)`

# Invoquer méthodes d'objets (non statiques)

Toujours appelées sur un objet :

```
c1.depot(50); //  appel depot sur c1
```

- uniquement des méthodes de la classe ayant servi à créer l'objet ;
- agissent sur l'état interne (variables internes) de cet objet

```
c1.depot(50); // ajoute 50 dans c1.solde
c2.depot(30); // ajoute 30 dans c2.solde
c1.affiche(); // le solde courant dans c1
```

# Méthodes d'objet : « accesseurs » et « modificateurs »

- Accesseurs : fonctions ne modifiant pas l'état interne : utilisées pour l'observer, se contentent de renvoyer un résultat.

```
public double donneSolde(){
    return solde ;
}
```

- Modificateurs : modifient l'état interne.

```
public void depot(double montant){
    this.solde += montant ;
}
```

# Programme main : exemple + démo 1

Démo sous Eclipse :

```
public static void main(String[] args) {
        Compte c1 = new Compte();
        System.out.print("c1 creation: ");
        c1.affiche();
        c1.numero =  123456;
        c1.titulaire = "Henri Dupont";
        c1.depot(100);
        System.out.println("c1 depot etc -->");
        c1.affiche();
}
```

```
c1 creation: Numero: 0, Titulaire: null, Solde: 0.0
c1 depot etc -->
Numero: 123456, Titulaire: Henri Dupont, Solde: 100.0
```

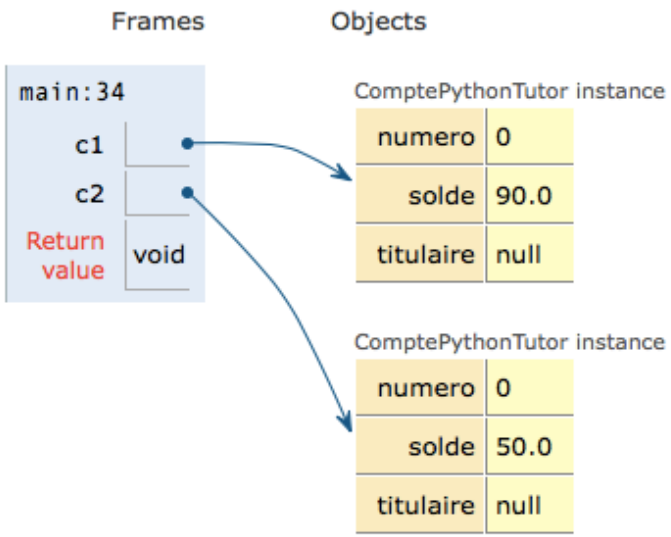
# Programme main : exemple + démo 2

**Démo sous PythonTutor** : nous ajoutons un main **à l'intérieur** de la classe Compte !

# Après exécution sous pythonTutor

```
    return this.solde;
}
public void affiche(){
    System.out.print("Numero: "+this.numero);
    System.out.print(", Titulaire: "+this.titulaire)
    System.out.println(", Solde: "+this.getSolde());
}

public static void main(String[] args) {
    ComptePythonTutor c1= new ComptePythonTutor();
    System.out.println("Après création c1"); // poin
    c1.affiche();
    ComptePythonTutor c2=new ComptePythonTutor();
    System.out.println("Après création c2"); // ici
    c2.affiche();
    c1.depot(100);
    System.out.println("Après dépot c1");  // ici
    c1.affiche();
    c2.depot(50);
    System.out.println("Après dépot c2"); // et ici
    c1.retrait(10);
}
```

```
Après création c1
Numero: 0, Titulaire: null, Solde: 0.0
Après création c2
Numero: 0, Titulaire: null, Solde: 0.0
Après dépot c1
Numero: 0, Titulaire: null, Solde: 100.0
Après dépot c2
```



# Les constructeurs

# Retour sur la construction d'objets

- Un objet de type `Compte` est crée en mémoire par
  `new Compte()`
- `Compte()` est un constructeur d'objets pour la classe `Compte`.

**Constructeur**

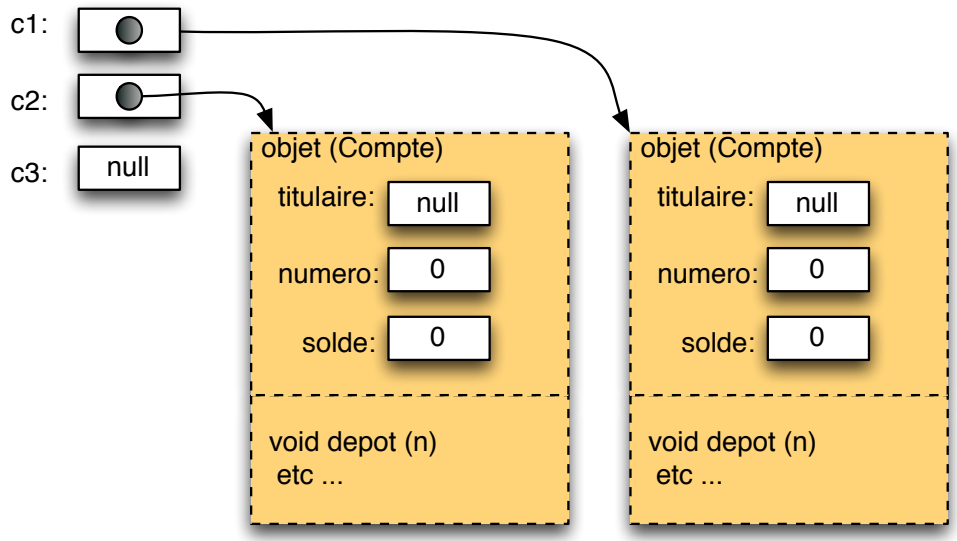
Sorte de méthode chargée de réserver de la place pour un nouvel objet, d'y stocker et initialiser ses variables d'instance.
Il retourne l'adresse de l'objet crée.
On ne peut l'invoquer que via l'opérateur `new`

# Exécution de `new`

Exécution de `new Compte()` :

1. allocation d'espace mémoire pour un objet de type `Compte`.
2. Initialisation de chaque variable d'instance définie dans la classe (valeur pas défaut si aucune valeur initiale).
3. retourne en résultat l'adresse de l'objet construit.

Les valeurs par défaut conviennent rarement : compte avec titulaire à null, solde à 0, ect !



# Constructeur par défaut

- donné par le nom de la classe sans paramètres : `Compte()`
- existe si aucun constructeur n'est déclaré explicitement ;
- initialise les variables d'instance à des valeurs par défaut
- il ne peut y avoir qu'un seul constructeur par défaut.
- n'est plus disponible dès que l'on déclare un constructeur explicitement.

**Principe no. 1 (Initialisation par constructeurs déclarés)**

Définition quasi-sytématique de constructeurs pour chaque classe.
But : tout objet crée est initialisé à des valeurs cohérentes.

# Les constructeurs déclarés

- On peut en définir plusieurs par classe : chacun correspond à un mode d'initialisation différent.
- leur nom est celui de la classe (`Compte`) et peuvent avoir des paramètres ;
- ils n'ont pas de type de retour ;
- on ne peut les invoquer que via `new` ;
- Si plusieurs constructeurs, le nombre (ou le type) de leurs arguments doivent être distinctes (leur signature).

**Attention**

Dès qu'un constructeur est déclaré, le constructeur par défaut de cette classe cese d'exister. Pour le rendre disponible, on doit l'introduire explicitement.

# Constructeurs pour la classe Compte

Un avec trois arguments et un sans le solde qui est alors mis par défaut à 25 euros (solde minimal autorisé).

```
public class Compte {
  String titulaire;
  int numero;
  double solde;
  public Compte(int num, String tit, double init){
      numero = num; titulaire=tit; solde = init;
  }
  public Compte(int num, String tit){
      numero = num; titulaire=tit; solde = 25;
}
```

# Exécution sous pythonTutor

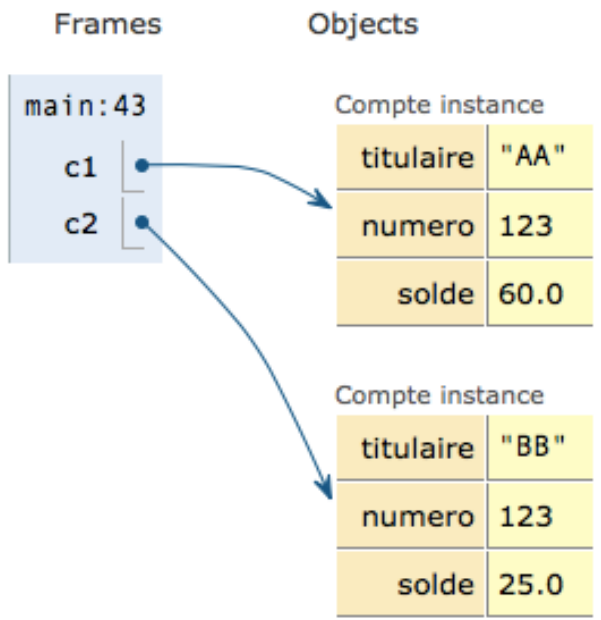
```
}

public void afficher(){
   System.out.println(toString());
}
public static void main(String[] args) {
    Compte c1 = new Compte(123, "AA",60);
    Compte c2 = new Compte(123, "BB");
    c1.afficher();
    c2.afficher();
```

Edit code

has just executed

to execute

code to set a breakpoint; use the Back and Forward buttons to jump there.



# Un constructeur peut échouer

Si les arguments d'initialisation sont invalides on peut choisir de faire échouer le constructeur ⇒ pas de création d'objets incohérents !

Exemple : le solde d'initialisation est négatif.

```
public class Compte {
  String titulaire;
  int numero;
  double solde;  // 25 euros minimum

  public Compte(int num, String tit, double init){
      if (init <25.0)
        throw new IllegalArgumentException();
      numero = num; titulaire=tit; solde = init;
  }
```

# Contrôler l'accès aux données, encapsulation.

# Modifier l'accès aux données

L'état d'un objet est à priori accessibles depuis l'extérieur de celui-ci :

```
Compte c1 = new Compte("Martin",1, 100);
Compte c2 = new Compte("Dupond", 2,  50);
c2.solde = c2.solde + c1.solde;
c1.solde = c1.solde - 10;
```

Quel est le problème avec ce code ?

- les soldes de `c2` et `c1` changent sans que cela corresponde à des opérations de compte ;
- on n'en garde pas trace dans l'historique...

**Conclusion**

il est important de protéger l'état des objets de modifications incontrôlées.

# Encapsulation en Java : modificateurs d'accès

Java : protection par accès restreint aux composantes d'une classe ou paquetage, via modificateurs d'accès :

## Modificateur d'accès

Spécifie la visibilité d'une composante depuis tout autre composante :

- `private` : visible uniquement par les méthodes de la classe ;
- `protected` : visible uniquement par les méthodes de la classe et celles des classes dérivées ;
- `public` : visible par tous ;
- pas de modificateur : visible par toutes les composantes appartenant au même paquetage (ou même répertoire, si paquetage par défaut).

# Protéger l'état interne des `Comptes`

```
public class Compte{
  private double numero;
  private String Titulaire;

  public double getSolde(){...}
    ....
  }
```

**Principe no. 2 : Privatisation des données + accéssibilité de méthodes**

1. les variables d'instance sont déclarées `private` : seuls les objets de cette classe pourront y accéder ;
2. les méthodes de la classe sont déclarées `public` : elles seront accessibles par tout autre objet depuis tout paquetage.
3. la classe elle-même est (biensûr) déclarée publique . . . .

# Conséquences de la protection des données

1. Initialiser variables protégées ⇒ *définir des constructeurs*.
2. Obtenir valeur variables protégées ⇒ définir des *accesseurs*
3. Attention : la défintion de modificateurs ne doit pas être systématique !

```
public class Compte{
  private int numero;
  private double solde;
  /* Constructeur */
  public Compte(int n, double init){
     this.numero=n; this.solde=init;}
  /* Acceseurs */
  public double getSolde() { return this.solde; }
  public double getNumero(){ return this.numero;}
  ...
 }
```

# Objets dans autres objets et tableaux

# Objets dans tableaux et objets

On peut enregistrer les objets dans toute structure de données : dans les cases d'un tableau ou dans les champs d'un autre objet.

Exemples :

- tableau avec case objets ( `Compte` )
- objet `Banque`, contenant la liste de tous les (objets) comptes.
- `Personne` contient variables nom, adresse, et date de naissance (tous des objets)
- la date de naissance modéliseé par un objet `Date` ;

# Tableau de comptes (1)

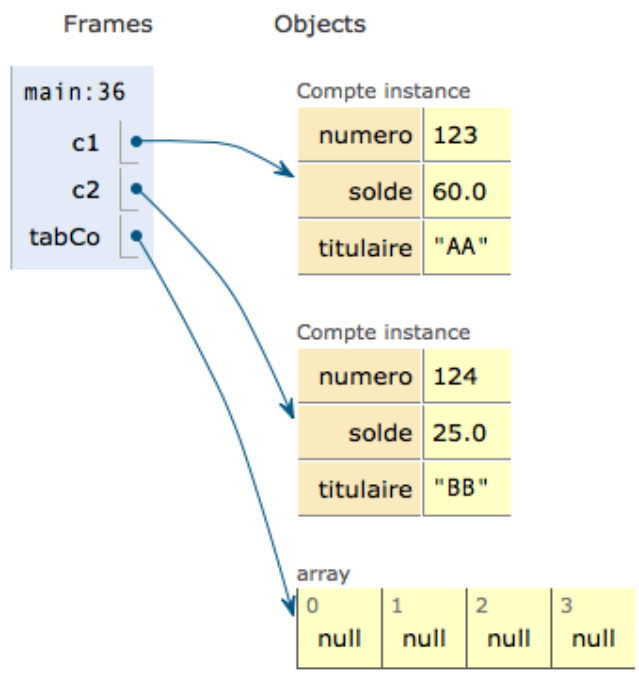
`new` initialise les cases du tableau à null !

```
Compte[] tabCo = new Compte [4];
```

```
25      }
26       // main pour Python Tutor
27      public static void main(String[] args) {
28          /* 2 comptes crées avec constructeurs différent
29          Compte c1 = new Compte(123, "AA",60);
30          Compte c2 = new Compte(124, "BB");
31          c1.affiche();
32          c2.affiche();
33          /* 1 tableau de comptes: ses cases sont à null
34          Compte[] tabCo = new Compte [4];
35          /*  chaque case devra être créee */
36          tabCo[0] = new Compte(125, "CC");
37          tabCo[1] = c1;
38          tabCo[2] = c2;
39          tabCo[3] = new Compte(126, "DD");
40      }
```

Edit code

line that has just executed



# Tableau de comptes (2)

**Danger !**

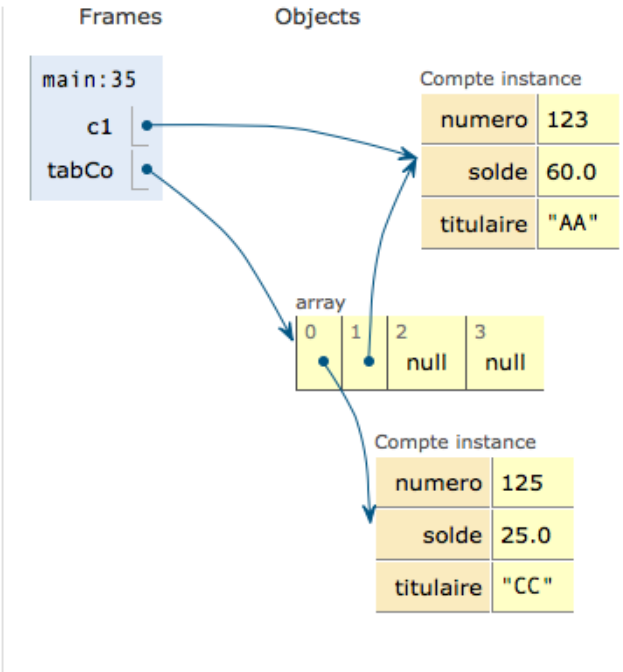
Si un tableau/objet contient des variables ou cases objets, ces dernières sont initialisées par défaut à null. Un accès à la case ou variable provoque un échec (NullPointerException).

Addendum au principe no. 1 : Les constructeurs devront initilaiser les variables objets internes.

```
Compte c1 = new Compte(123, "AA",60);
tabCo[0] = new Compte(125, "CC");
tabCo[1] = c1;
tabCo[2] = new Compte(124, "BB");
tabCo[3] = new Compte(126, "DD");
```

# Tableau de comptes (3) : intialisation partielle

```
Compte c1 = new Compte(123, "AA",60);
tabCo[0] = new Compte(125, "CC");
tabCo[1] = c1;
```



# objet Banque : exemple d'objets dans objet

- variables : arrayList de tous les comptes de la banque ;
- méthodes : prennent en paramètre un numéro de compte et **non pas un compte**.
- nouvelle méthode : ouverture d'un compte (crée + ajouté dans la liste des comptes).

```
public class Banque {
   int numComptes = 1; // Numero pour nouveau compte
   ArrayList<Compte> tous = new ArrayList<Compte>();

   public int ouvrirCompte(String nomTit, double init){
      Compte c = new Compte(numComptes,nomTit, init);
      tous.add(c);
      numComptes++;  // compteur de no's de compte.
      return n;  // Retourne numero du compte ouvert
   }  ...
```

# Banque : exemple d'objets dans objet (2)

Les méthodes prennent en paramètre un numéro de compte !

```
  ....
public double getSolde(int num){
    Compte c = tous.get(num-1); // simplification!
    return c.getSolde();        // delegation
}
public boolean depot(int num, double m) {
     if (!isNumCompte(num)) return false;
     Compte c = tous.get(num-1);
     c.depot(m);
     return true;
}
```

- `isNumCompte` teste si son argument est un numéro de compte valide de la banque ;
- `getSolde` cherche un compte de ce numéro et lui applique la méthode `getSolde` de la classe Compte.

# Utilisation d'un objet Banque

```
Banque bnp = new Banque(); // objet Banque
int n1 = bnp.ouvrirCompte("titi",100);
int n2 = bnp.ouvrirCompte("toto",350);
// solde du compte numero 1
System.out.println("Compte no."+n1 +
        "= "+bnp.getSolde(n1));
bnp.depot(n2, 300);  // depot sur compte numero 2
```

- `bnp.ouvrirCompte("titi",100)` : créé un nouvel objet `Compte` et l'ajoute dans la liste de la banque. Retourne le numéro attribué à ce compte.
- `bnp.getSolde(n1)` : solde du compte no. n1 de `bnp`.
- `bnp.depot(n2,300)` : ajoute 300 au compte no. n2 de `bnp`.

# Delégation

Pour implanter la méthode M d'un objet A, on délègue les actions à la méthode M d'un objet B s'il fait partie des données de A. Autrement dit, le code M() de A invoque M() de B.

```
public class B {
    int x;
    // Affiche les donnees de B
    public void print(){
        System.out.print(x);
    }
}
public class A {  // A contient un objet B
    B b;
    public void print(){ // afiche donnes de A
        b.print(); // delegation!
    }
}
```

# Délégation : exemple 1

- `A.print()` doit afficher les données de A :
  - données de A : un objet B ;
  - `A.print()` $\Rightarrow$ invoque `B.print()`.

```
public class B {
    int x;
    // Affiche les donnees de B
    public void print(){
        System.out.print(x);
    }
}
public class A {  // A contient un objet B
    B b;
    public void print(){
        b.print(); // delegation!
    }
}
```

# Exemple (2) de delégation : afficher une Personne

- `Personne` contient un objet `Date` ;
- `Date` contient : jour, mois, année + méthode `affiche()` .
- `affiche` de `Personne` : délègue l'affichage de la date de naissance à l'objet `Date`.

```
public class Date {
   int jour; int mois; int annee;
   public void affiche(){System.out.print(jour+"/"+mois+"
}
public class Personne { // contient Date
   String nom;
   Date naissance;  // ici
   public void affiche(){
      System.out.print("Nom: "+nom);
      naissance.affiche(); // delegation
   }
}
```

# Principe no.3

## Principe no. 3 : Delégation

Si un objet A contient un objet B, et si l'on doit définir la méthode M de A devant necéssairement agir sur les données de B :

- alors, c'est à la classe B de fournir une méthode M qui réalise les actions sur ses données propres (à B) ;
- la méthode M de A doit être définie par delégation sur la méthode M de B.
- En aucun cas la méthode M de A doit accèder directement aux données de B.

**Important** : Ce principe garantît que seul B accède à ses données.

# Exemple de delégation : méthode `getSolde()`

- `getSolde` dans `Banque` :
  1. prend en paramètre le numéro du compte,
  2. cherche un compte de ce numéro dans la liste de la banque
  3. et delègue le travail de renvoyer le solde à la méthode `getSolde` du compte trouvé.

```
public class Banque {
    ....
   public double getSolde(int num){
      Compte c = tous.get(num-1);
      return c.getSolde();          // delegation
   }
```

S'applique également pour écrire `retrait` et `depot` dans `Banque`.

# La variable `this` + demo avec PythonTutor

# Variables d'une méthode non statique

```
public class Compte{
   String titutaire;  int numero;  double solde ;

   public void depot(double montant){
      this.solde =  this.solde +  montant ;
   }
```

3 sortes de variables :

- paramètres (`montant`),
- locales (aucune ici) ;
- `this` = objet courant : pour désigner les variables de l'objet courant
  `this.solde` ≈ *la variable solde de l'objet courant*.

# this = objet courant

```
public void depot(double montant){
  this.solde = this.solde + montant ;
}
```

**`this` = objet courant**

C'est l'objet sur lequel la méthode qui utilise `this` est invoquée..

Dans la méthode `depot` :

- `this` est l'objet sur lequel on est en train d'opérer un dépot ;
- `this.solde` $\Rightarrow$ variable solde de cet objet courant.

La valeur de `this` change à chaque exécution de la méthode `depot`.

# Exécution d'une méthode utilisant this

```
public void depot(double montant){
  this.solde = this.solde + montant ;
}
```

Exécution de l'appel `c1.depot(150)` $\Rightarrow$

1. Valeur des variables pour l'appel à la méthode `depot` :
   - `montant` $\mapsto$ 150 ;
   - `this` $\mapsto$ `c1`, donc `this.solde = c1.solde`
2. Exécution du corps de la méthode `depot` :

```
this.solde = this.solde + montant ⇒
c1.solde = c1.solde + 150
```

Que se passe-t-il si on exécute `c2.depot(70)` ?

# Règles à suivre dans ce cours

## Suivre les 3 principes

1. Initialisation variables d'instance (principe 1)
   - préférer les constructeurs,
   - où donner valeurs initiales à la déclaration des variables :
2. Protéger les données via modificateurs d'accès (principe no. 2)
   - y accéder uniquement via les méthodes (déclarer accesseurs et modificateurs nécessaires).
3. Utiliser la délégation pour définir les méthodes agissant sur des objets internes (principe no. 3)

# Variables et méthodes statiques

# Variables et méthodes statiques

En plus de variables d'instance et méthodes non statiques, une classe peut avoir :

- Variables statiques $\Rightarrow$ variables partagées par tous les objets d'une même classe.
- Méthodes statiques $\Rightarrow$ actions ne touchant pas à l'état interne des objets, ne connaissant `this`, ni les variables d'instance.

# Variables statiques

- Déclarées au même niveau que les variables d'instance,
- mais précédées du mot clef **static**.
- Les variables statiques déclarées dans une classe **Nom-classe** sont accessibles de deux manières :
  - **depuis l'extérieur de la classe** :
    `Nom-classe.nom-variable-statique`
    ou `nom-objet.nom-variable-statique`
  - **depuis la classe** :
    `nom-variable-statique`

# Exemple 1 : joueurs en ligne

On veut modéliser les joueurs d'un jeu en ligne afin de compter :

- le nombre de joueurs connectés,
- le nombre de fois que chaque joueur a joué

Quelles sont les variables d'instance, variables statiques et méthodes de la classe Joueur ?

# Exemple 1(2) : modéliser un joueur

**Variables d'instance** (état interne)

- nom du joueur (propre au joueur) ;
- nombre de fois qu'il a joué (propre au joueur) ;

**Méthodes non statiques** (agissant sur l'état interne)

- un constructeur qui initialise ces variables,
- une méthode pour joueur,
- une méthode qui affiche le nombre de fois qu'il a joué.

**Variable statiques** (état partagé)

- compteur de tous les joueurs connectés sur le jeu

# Exemple 1 (3) : La classe Joueur

```
public class Joueur{
    String nom;
    int aJoue = 0;
    static int nbEnJeu=0;

    public Joueur(String n){  // constructeur
        this.nom = n; this.aJoue=0; nbEnJeu++;
    }
    public void jouer(){
        this.aJoue++;
    }
    public void afficheNbeFoisJoue(){
      Terminal.ecrireStringln
        (this.nom+ " a joue "+this.aJoue+" fois.");
   }
```

# Exemple 1 (4) : Un programme de test

```
public class TestJoueur1{
 public static void  main (String [] arguments){
   Joueur j1 = new Joueur("Pierre");
   Joueur j2 = new Joueur("Anne");
   j1.jouer();  j1.jouer();  j1.jouer();
   j1.afficheNbeFoisJoue();
   j2.afficheNbeFoisJoue();
   Terminal.ecrireStringln
     ("Nombre de joueurs connectes: "+ Joueur.nbEnJeu);
   Terminal.ecrireStringln
     ("Nombre de joueurs connectes: "+ j1.nbEnJeu);
  }
```

```
> java TestJoueur1
Pierre a joue 3 fois.
Anne a joue 0 fois.
Nombre de joueurs connectes: 2
Nombre de joueurs connectes: 2
```

# Méthodes statiques

1. Ne peuvent pas agir sur l'état interne ne peuvent pas faire référence aux variables d'instances ni à l'objet courant (`this`).
2. Elles sont communes à tous les objets d'une classe $\Rightarrow$ elles appartiennent à une classe.
3. On les invoque via un nom de classe : `Joueur.afficheEnLigne();` et non pas via un objet.

# Exemple 1(5) : Méthode afficheEnJeu()

But : afficher le nombre de joueurs connectés. Ne touche pas aux objets : c'est une méthode statique.

```
public class Joueur{
    String nom; int aJoue = 0;  static int nbEnJeu=0;

    public Joueur(String n){  // constructeur
        this.nom = n; this.aJoue=0; nbEnJeu++;
    }
    public void jouer(){
        this.aJoue++;
    }
    public void afficheNbeFoisJoue(){ ...
    }
    public static void afficheEnJeu(){
      Terminal.ecrireStringln
       ("Nombre de joueurs connectes: "+ Joueur.nbEnJeu);
   }
```

# Exemple 1(6) : Nouveau programme de test

```
    Joueur.afficheEnJeu();
    Joueur j1 = new Joueur("Pierre");
    Joueur j2 = new Joueur("Anne");
    j1.jouer();  j1.jouer();  j1.jouer();
    j1.afficheNbeFoisJoue(); j2.afficheNbeFoisJoue();
    Joueur.afficheEnJeu();
```

```
> java TestJoueur1
Nombre de joueurs connectes: 0
Pierre a joue 3 fois.
Anne a joue 0 fois.
Nombre de joueurs connectes: 2
```

La variable `NbEnJeu` est consultée alors qu'aucun objet de la classe n'est encore crée....

Comment expliquer cela ?

# Les paquetages

## Rôle des paquetages

- grouper sous un même nom et dans un même répertoire des classes et interfaces issues de plusieurs fichiers ;
- limiter l'accès aux composantes du paquetage.

## Syntaxe

- pas de construction syntaxique englobant toutes les composantes ;
- déclaration d'appartenance au paquetage par composante dans 1ère ligne de son fichier :
  `package` *nom-du-paquetage* ;

# Exemple 1 : paquetage pour les comptes

Toutes les classes et interfaces d'une application de gestion de comptes bancaires mis dans un paquetage *comptes*,

- chaque fichier de classe ou interface spécifie dans sa 1ère ligne :

```
package comptes;
```

- tous les fichiers sont (très souvent) réunis dans un sous-répertoire `comptes` (du nom du paquetage).

# Exemple : paquetage pour les comptes (2)

```
/* dans le fichier Compte.java */
package comptes;
public class Compte{
   .....

/* dans le fichier Banque.java */
package comptes;
public class Banque {
  ...

/* dans le fichier CarteCredit.java */
package comptes;
public interface CarteCredit {
...
```

Sans déclaration de paquetage, les composantes font partie du *paquetage par défaut*, regroupant toutes les classes *d'un même répertoire*.

# Utiliser une composante de paquetage : noms qualifiés

Utiliser une composante de paquetage depuis l'extérieur de celui-ci ?

- $\Rightarrow$ elle doit être déclarée `public`.
- une solution : utiliser son nom qualifié :

```
comptes.Banque b = new comptes.Banque();
```

`comptes.Banque` est le nom qualifié de la classe Banque.

# Utiliser une composante de paquetage : import

- importer la composante :

```
import comptes.Banque;
...
Banque b =  new Banque();
```

- importer toutes les composantes contenues dans un paquetage :

```
import comptes.*;
Banque b =  new Banque();
Compte c = new Compte();
```

$\Rightarrow$ pas besoin des noms qualifiés par la suite.

Importations en début de fichier (juste après `package`).

# 8. Résumé.

# Résumé

- Classe = patron d'objets + nom type, contenant :
  - variables d'instance : données locales, de préférence protegées ;
  - constructeurs (pour initialiser var. instance) ;
  - méthodes d'instance (+ éventuellement statiques).
- Objet = instance d'une classe :
  - créee via `new` + constructeur
  - possèdent données locales + méthodes d'instance.
- méthodes dans une classe :
  - d'instance (non statiques) : À invoquer sur les objets. Accèdent aux var. d'instance.
  - statiques : n'ont pas accès aux var. d'instance.

Demo : comptes et banque sous Eclipse.

# Parenthèse : la surcharge

# Parenthèse : la surcharge

**Surcharge** Un *opérateur* ou un *nom de méthode* est surchargé, si chaque utilisation/invocation peut correspondre à une sémantiques différente.

# Surcharge d'opérateurs

Exemples d'opérateurs surchargés :

- tous les opérateurs arithmétiques :
  - `1/2` $\Rightarrow$ sémantique : division sur les entiers ;
  - `1.5/2` $\Rightarrow$ sémantique : division sur les nombres flottant à double précision ;
- tous les opérateurs `+` :
  - `1+2` $\Rightarrow$ sémantique : addition sur les entiers ;
  - `1.5+2` $\Rightarrow$ sémantique : addition sur les nombres flottant à double précision ;
  - `"ab"+2` $\Rightarrow$ sémantique : concaténation de chaînes de caractères

Surcharge = même syntaxe pour plusieurs sémantiques.

# Surcharge de méthodes

```
// Cherche un entier dans un tableau d'entiers
 static boolean cherche(int x, int [] t){
   for (int i=0; i<t.length; i++){
          if (x == t[i]) return true;
   }return false;}

// Cherche un caractere dans un tableau de caracteres
 static boolean cherche(char x, char [] t){
   for (int i = t.length-1; i >=0;  i--){
          if (x == t[i]) return true;
   }return false;}
```

- deux méthodes de même nom
- avec des comportements différents (sémantique)

# Surcharge de méthodes (2)

On peut surcharger un nom de méthode à condition de ne pas introduire d"ambigüité lors d'un appel :

- le nombre et ou le type des paramètres doivent être à chaque fois différents, ou données dans un ordre différent ;
- on parle de signature de méthodes :

**Signature de méthode**

nom de la méthode + suite de types de ses arguments.

# Exemple d'utilisation de la surcharge

```
public static void main(String[] args) {
  int [] ti = {1,3, 7};
  char [] tc = {'e', 'f', 'k', 'r'};
  boolean a = cherche('a', tc);
  boolean b = cherche(3, ti);
```

- Deux méthodes `cherche` : un pour les tableaux de caractères, l'autre pour les tableaux d'entiers.
- Comment le compilateur sait lequel exécuter lors d'un appel ?

$\Rightarrow$ On peut toujours determiner quelle méthode exécuter en examinant le type des arguments **effectifs** de l'appel.