Introduction à la programmation objet en Java

Virginia Aponte

CNAM-Paris

9 septembre 2023

Contenu

Caractéristiques d'un objet (Java)

- Crée à partir d'une classe dont il est une instance.
- A un type dynamique : la classe ayant servi à le créer,
- C'est une donnée composite formé de :
  - état ou variables d'instance locales à l'objet
  - méthodes *agissant localement* sur celles-ci
- A une identité unique donnée par *son adresse dans le tas*,
- A une « interface » donnée par la liste de toutes les méthodes qu'on peut invoquer sur l'objet
- Appel à ses méthodes préfixés par l'objet.
  Ex : `mot.charAt(0)` $\Rightarrow$ appel de méthode de la classe `String` sur l'objet `mot`.

4 concepts principaux en POO

Nous les étudiérons dans ce cours et les prochains.

1. Encapsulation : on favorise la protection des variables d'instance ;
2. Polymorphisme : une *variable objet* a un type déclaré (statique) et un type dynamique (constaté à un moment de l'exécution). Ils ne sont pas forcément égaux, mais ils sont *compatibles par sous-typage*.
3. Héritage : mécanisme sytaxique permettant de créer de nouveaux types (classes, interfaces) par extension de types existants.
4. Abstraction : on prefère déclarer les variables du type le plus général possible parmi tous les types potentiellement compatibles.

Utiliser des objets (de la bibliothèque standard)

Les bibliothèques Java

- apprendre à utiliser des objets de la bibliothéque standard de Java :
  - 4240 classes dans l'API standard de java ;
  - très (très) nombreuses bibliothèques disponibles
- indispensable de consulter sa documentation . . .

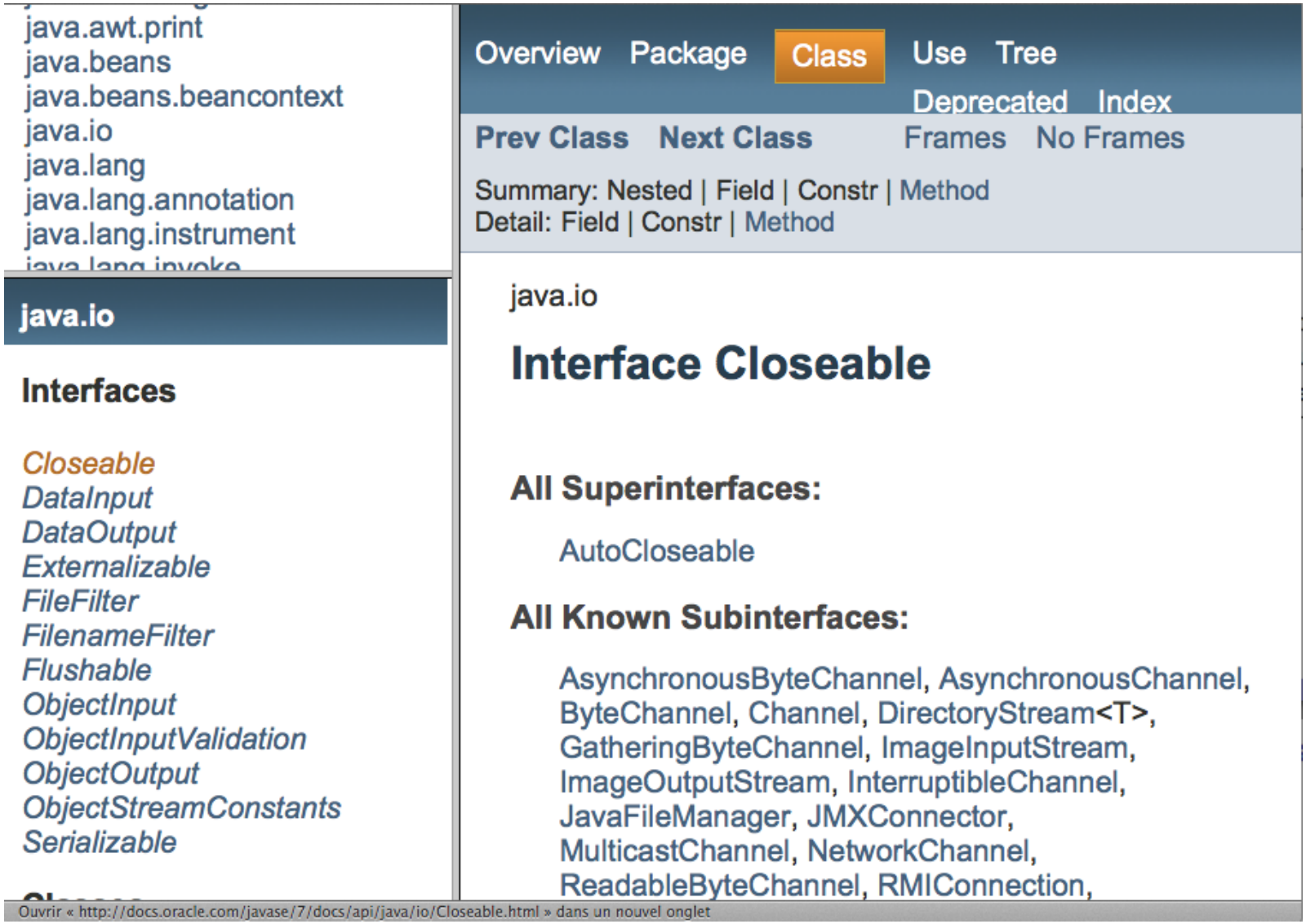
Documentation et Interface

La documentation d'une classe explique :

- ce que *représente* un objet de la classe ;
- comment le *créer* ;
- la liste des méthodes permettant de le *manipuler* (+ leur comportément)

On appelle tout cela, « interface » de l'objet, au sens où cette description nous permet de savoir comment l'utiliser.

La javadoc



1

1. Voir le cours sur la javadoc pour plus de détails.

Description d'une classe

La javadoc d'une classe explique :

- quels sont les *invariants* des objets : quelles propriétés sont vraies sur les objets une fois créés.
- comment *créer un objet* ;
- comment le modifier ;
- comment connaître son état.
- comment utiliser l'objet.

Documentation d'une méthode

- **Signature de la méthode** : nom de la méthode, type de retour, type des paramètres ;
- courte description de ce que fait la méthode ;
- spécification de la méthode ;
- description des paramètres ;
- description de la valeur retournée ;
- description des exceptions levées par la méthode.

Exemple avec String et ArrayList : création d'objets

- Créons l'objet directement avec `new` + un constructeur

```
String s1 = new String({'a', 'b'});
String s2 = "bonjour";
ArrayList<String> a1 = new ArrayList<String>();
```

  - s1, s2 : 2 instances de String,
  - a : instance d'ArrayList.

Question : trouvez dans la javadoc les constructeurs employés ici

Invocation de méthodes sur un objet

Deux sortes de méthodes dans une classe : statiques ou d'instance.

- Comme son nom l'indique, une méthode *d'instance* agit sur un objet ou instance ;
- Avant d'appliquer ces méthodes il faut donc avoir crée l'instance.
- Syntaxe d'une application :
  - *instance* "." *nom-méthode+arguments*
  - Ex : `s1.charAt(0);`
  - $\Rightarrow$ application de la méthode `charAt(0)` sur l'objet `s1`.

La POO

Problème : modéliser des données complexes

- Donnée complexe : constituée d'autres données complexes . . .
  - Ex : un système bancaire constitué de banques qui possèdent des clients qui possèdent des comptes.
  - Vous voulez implanter des opérations de banque à banque
- En programmation procédurale :
  - Données centralisées et hiérarchisées, structure complexe
  - En Java ce sera éclaté en plusieurs structures « parallèles ». Ex : un tableau d'identifiants de comptes de la banque + un tableau des soldes pour ces comptes, etc.
  - Code compliqué : on devra manipuler/modifier plusieurs tableaux en gardant leur cohérence.
  - La vue logique de l'application est offusquée par la gestion de la compléxité des données.

Derrière la notion d'objet

Découper une donnée complexe en données plus petites associées aux concepts de notre application : compte, client, banque, système bancaire.

- 1 concept $\approx$ un type d'objet (classe)

Derrière la notion d'objet

Découper une donnée complexe en données plus petites associées aux concepts de notre application : compte, client, banque, système bancaire.

- 1 concept $\approx$ un type d'objet (classe)
- chaque type d'objet regroupe localement

Derrière la notion d'objet

Découper une donnée complexe en données plus petites associées aux concepts de notre application : compte, client, banque, système bancaire.

- 1 concept $\approx$ un type d'objet (classe)
- chaque type d'objet regroupe localement
  1. ses données (dites variables d'instance)

Derrière la notion d'objet

Découper une donnée complexe en données plus petites associées aux concepts de notre application : compte, client, banque, système bancaire.

- 1 concept $\approx$ un type d'objet (classe)
- chaque type d'objet regroupe localement
  1. ses données (dites variables d'instance)
  2. méthodes qui opèrent sur celles-ci

Derrière la notion d'objet

Découper une donnée complexe en données plus petites associées aux concepts de notre application : compte, client, banque, système bancaire.

- 1 concept $\approx$ un type d'objet (classe)
- chaque type d'objet regroupe localement
  1. ses données (dites variables d'instance)
  2. méthodes qui opèrent sur celles-ci
- Exemple : objet Compte

Derrière la notion d'objet

Découper une donnée complexe en données plus petites associées aux concepts de notre application : compte, client, banque, système bancaire.

- 1 concept $\approx$ un type d'objet (classe)
- chaque type d'objet regroupe localement
  1. ses données (dites variables d'instance)
  2. méthodes qui opèrent sur celles-ci
- Exemple : objet Compte
  1. données du compte : identifiant, solde courant, titulaire.

Derrière la notion d'objet

Découper une donnée complexe en données plus petites associées aux concepts de notre application : compte, client, banque, système bancaire.

- 1 concept $\approx$ un type d'objet (classe)
- chaque type d'objet regroupe localement
  1. ses données (dites variables d'instance)
  2. méthodes qui opèrent sur celles-ci
- Exemple : objet Compte
  1. données du compte : identifiant, solde courant, titulaire.
  2. opérations sur ces données : dépôt, retrait, obtenir le solde

Derrière la notion d'objet

Découper une donnée complexe en données plus petites associées aux concepts de notre application : compte, client, banque, système bancaire.

- 1 concept $\approx$ un type d'objet (classe)
- chaque type d'objet regroupe localement
  1. ses données (dites variables d'instance)
  2. méthodes qui opèrent sur celles-ci
- Exemple : objet Compte
  1. données du compte : identifiant, solde courant, titulaire.
  2. opérations sur ces données : dépôt, retrait, obtenir le solde
- les données de types d'objet différents peuvent être interconnectées, mais chaque objet traite les siennes et « interdit » leur accès aux autre types d'objet.

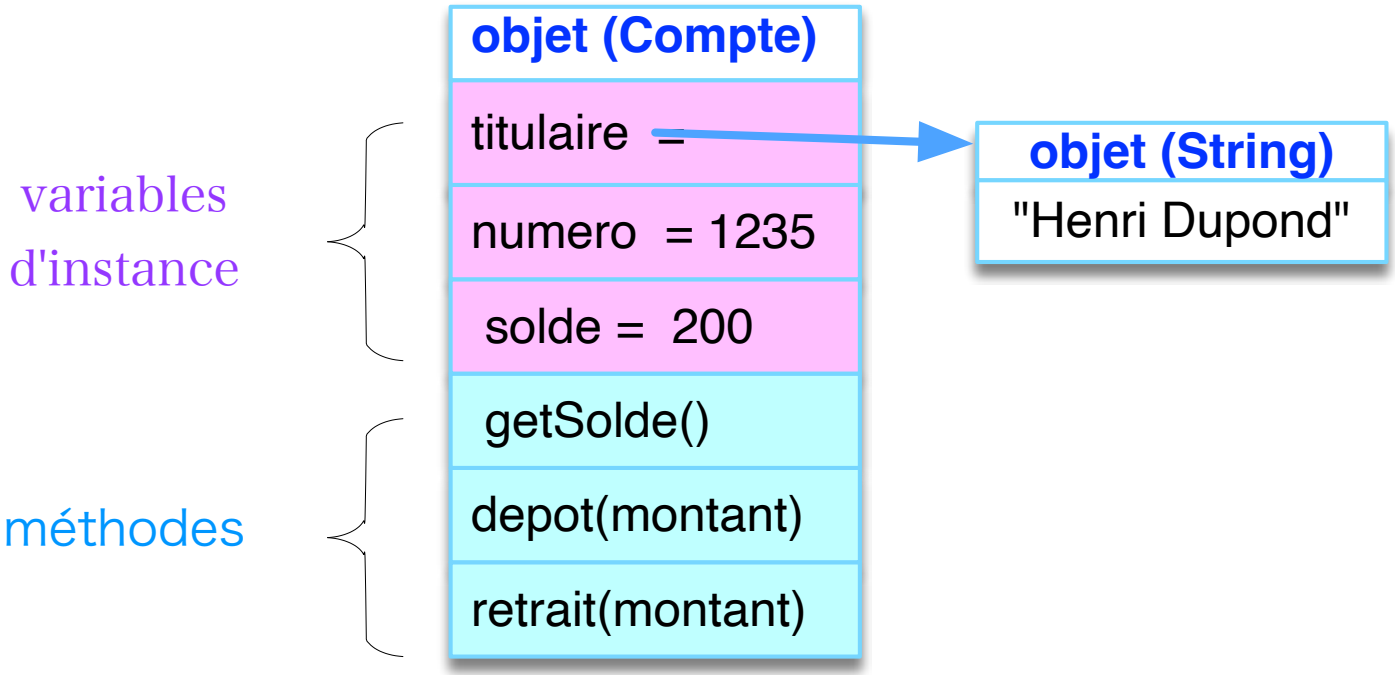
Exemple 1 : objet compte bancaire

Un objet `Compte` possède :

1. Données locales (variables d'instance) :
   - nom du titulaire, numéro du compte, solde courant ;
2. Opérations (méthodes) :
   - obtenir le numéro du compte, le solde courant ;
   - réaliser un retrait, un dépôt ;

$\Rightarrow$ agissent *sur les données* d'une instance.

Structure d'une instance de Compte



objet = état interne + opérations sur l'état

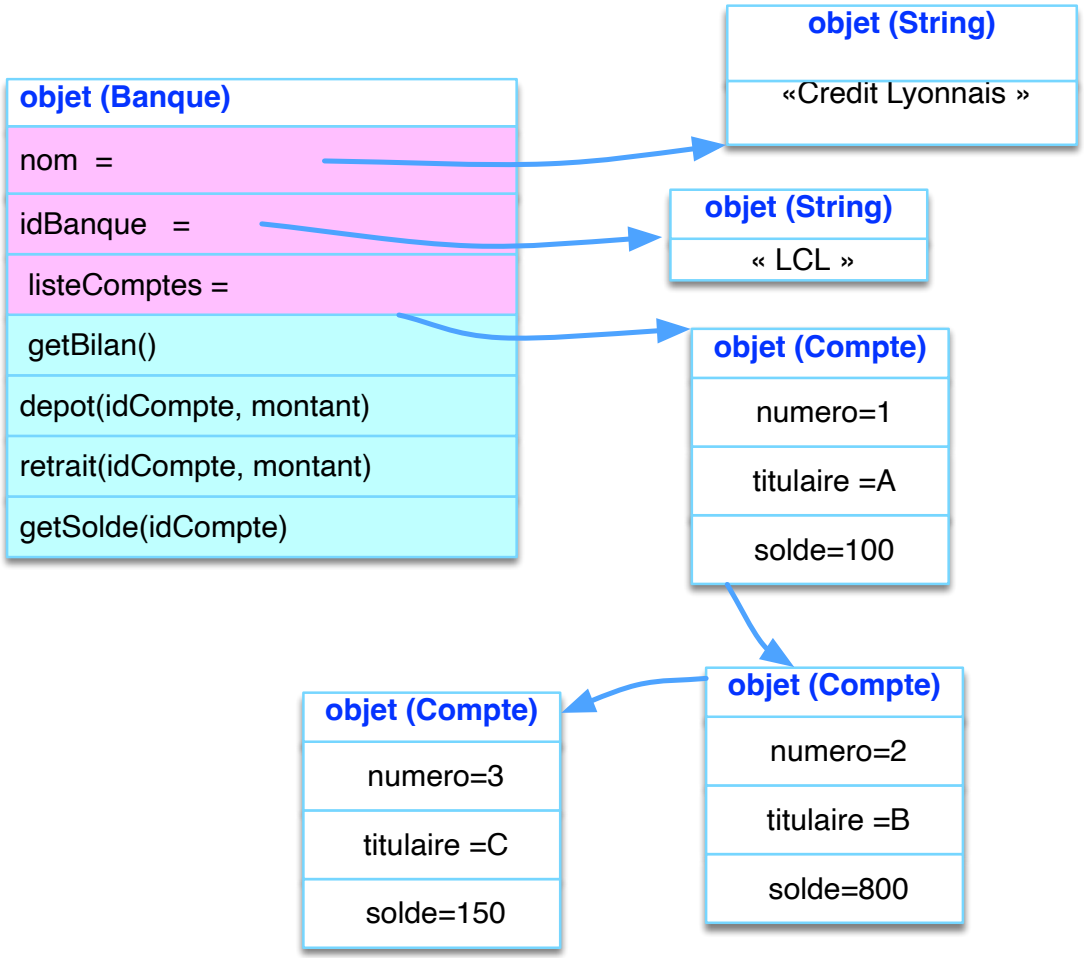
Exemple 2 : objet Banque

Un objet `Banque` possède :

1. Variables d'instance :
   - nom de la banque,
   - identtifiant (système IBAN),
   - liste de tous les comptes $\Rightarrow$ pointe vers d'autres objets !
2. Opérations (méthodes) :
   - tester si un numéro de compte existe
   - obtenir le solde d'un numéro de compte
   - réaliser un retrait, un dépôt pour un numéro de compte
   - calculer le bilan de la banque

Structure d'un objet Banque

Une banque avec 3 comptes bancaires. Notez que cet objet « est composé » (pointe) entre autres de 3 objets compte .



Application avec Compte et Banque

- Un objet `Banque` possède une liste d'objets `Compte` + méthodes pour gérer cette liste.
- Opération « nouveau client » :
  - création d'un nouvel objet `Compte` par client,
  - une méthode de `Banque` doit nous permettre de l'ajouter dans la liste de tous les comptes.
- Opération « solde pour un numéro de compte » :
  - chercher ce numéro dans la liste de tous les comptes
  - appliquer sur l'objet trouvé, la méthode qui permet d'obtenir le solde d'un compte.

Les classes pour fabriquer nos propres objets

Déclarer une classe qui servira à créer les objets

La classe `Compte` servira à créer des instances. Elle définit :

1. les attributs ou variables d'instance du futur objet ;

Déclarer une classe qui servira à créer les objets

La classe `Compte` servira à créer des instances. Elle définit :

1. les attributs ou variables d'instance du futur objet ;
2. les constructeurs pour initialiser les attributs

Déclarer une classe qui servira à créer les objets

La classe `Compte` servira à créer des instances. Elle définit :

1. les attributs ou variables d'instance du futur objet ;
2. les constructeurs pour initialiser les attributs
   - (si aucun → constructeur par défaut : `Compte()`)

Déclarer une classe qui servira à créer les objets

La classe `Compte` servira à créer des instances. Elle définit :

1. les attributs ou variables d'instance du futur objet ;
2. les constructeurs pour initialiser les attributs
   - (si aucun → constructeur par défaut : `Compte()`)
3. les méthodes utilisables sur ces attributs.

Déclarer une classe qui servira à créer les objets

La classe `Compte` servira à créer des instances. Elle définit :

1. les attributs ou variables d'instance du futur objet ;
2. les constructeurs pour initialiser les attributs
   - (si aucun → constructeur par défaut : `Compte()`)
3. les méthodes utilisables sur ces attributs.
4. En général, ne contient pas de méthode main !

La classe Compte

```
class Compte {
  private int numero;         // numero
  private double solde;       // solde courant
  private String titulaire; // nom titulaire
 // Constructeur
  public Compte(int n, String t, double s){
     numero=n; titulaire=t;solde=s; }
  public int getNumero() {return numero; }
  public String getTitulaire() { return titulaire;}
  public double getSolde(){ return solde; }
  public void depot(double n){
    solde = solde+n; }
  public void retrait(double m){
    solde = solde-m;  }
  public String toString() {
    return numero+", "+titulaire+", "+solde }
}
```

Les attributs de la classe Compte

```
private int numero;          // numero
private double solde;        // solde courant
private String titulaire;    // nom titulaire
```

Pour chaque objet crée :

- variables locales à cet objet $\Rightarrow$ variables d'instance ou attributs

Les attributs de la classe Compte

```
private int numero;          // numero
private double solde;        // solde courant
private String titulaire;    // nom titulaire
```

Pour chaque objet crée :

- variables locales à cet objet $\Rightarrow$ variables d'instance ou attributs
- ce sont les données propres à chaque objet Compte

Les attributs de la classe Compte

```
private int numero;          // numero
private double solde;        // solde courant
private String titulaire;    // nom titulaire
```

Pour chaque objet crée :

- variables locales à cet objet ⇒ variables d'instance ou attributs
- ce sont les données propres à chaque objet Compte
- protégées (`private`) ⇒ accessibles uniquement par les méthodes d'instance de la classe (encapsulation)

Le constructeur Compte sert à créer les instances

```
public Compte(int n,String t,double s){
   numero=n; titulaire=t;solde=s; }
```

Utilisé pour créer et initialiser les objets de type `Compte`

- sorte de méthode appelable uniquement via `new`
- sans type de retour ; son nom est celui de la classe
- peut prendre des arguments pour initialiser les attributs de l'objet que l'on crée.

Création du compte numéro 345689 de Mr. Dupont, avec 400 euros

```
public static void main (String [] arguments){
   Compte  c  = new Compte (345689,"Dupont", 400);
```

Le constructeur par défaut

- Si la classe ne déclare pas de constructeur, on utilise le constructeur par défaut, sans arguments. Il initialise les attributs à des valeurs par défaut ;

Le constructeur par défaut

- Si la classe ne déclare pas de constructeur, on utilise le constructeur par défaut, sans arguments. Il initialise les attributs à des valeurs par défaut ;
- Attention : Dès qu'un constructeur est déclaré, le constructeur par défaut de cette classe cese d'exister.

Le constructeur par défaut

- Si la classe ne déclare pas de constructeur, on utilise le constructeur par défaut, sans arguments. Il initialise les attributs à des valeurs par défaut ;
- Attention : Dès qu'un constructeur est déclaré, le constructeur par défaut de cette classe cese d'exister.
- Si plusieurs constructeurs, le nombre/type des arguments doivent être distincts (surcharge).

Création et initialisation d'objets (main)

- Déclaration : `Compte c;`

Création et initialisation d'objets (main)

- Déclaration : `Compte c;`
  - variables de méthodes ⇒non initialisées par défaut ;

Création et initialisation d'objets (main)

- Déclaration : `Compte c;`
  - variables de méthodes $\Rightarrow$non initialisées par défaut ;
  - attributs $\Rightarrow$ *toujours initialisés (via new)*.

Création et initialisation d'objets (main)

- Déclaration : `Compte c;`
  - variables de méthodes $\Rightarrow$non initialisées par défaut ;
  - attributs $\Rightarrow$ *toujours initialisés (via new)*.

  $\Rightarrow$ sans initialisation, c contient null.

Création et initialisation d'objets (main)

- Déclaration : `Compte c;`
  - variables de méthodes ⇒non initialisées par défaut ;
  - attributs ⇒ *toujours initialisés (via new)*.

  ⇒ sans initialisation, c contient null.
- Création et initialisation : `new` + appel constructeur(arguments)

```
Compte c1 = new Compte();//constructeur par defaut
Compte c2 = new Compte();
Compte c3;     // contient null
```

Création et initialisation d'objets (main)

- Déclaration : `Compte c;`
  - variables de méthodes ⇒non initialisées par défaut ;
  - attributs ⇒ *toujours initialisés (via new)*.

  ⇒ sans initialisation, c contient null.
- Création et initialisation : `new` + appel constructeur(arguments)

```
Compte c1 = new Compte();//constructeur par defaut
Compte c2 = new Compte();
Compte c3;     // contient null
```

- *c1 et c2 sont des instances de Compte*

Création et initialisation d'objets (main)

- Déclaration : `Compte c;`
  - variables de méthodes ⇒non initialisées par défaut ;
  - attributs ⇒ *toujours initialisés (via new)*.

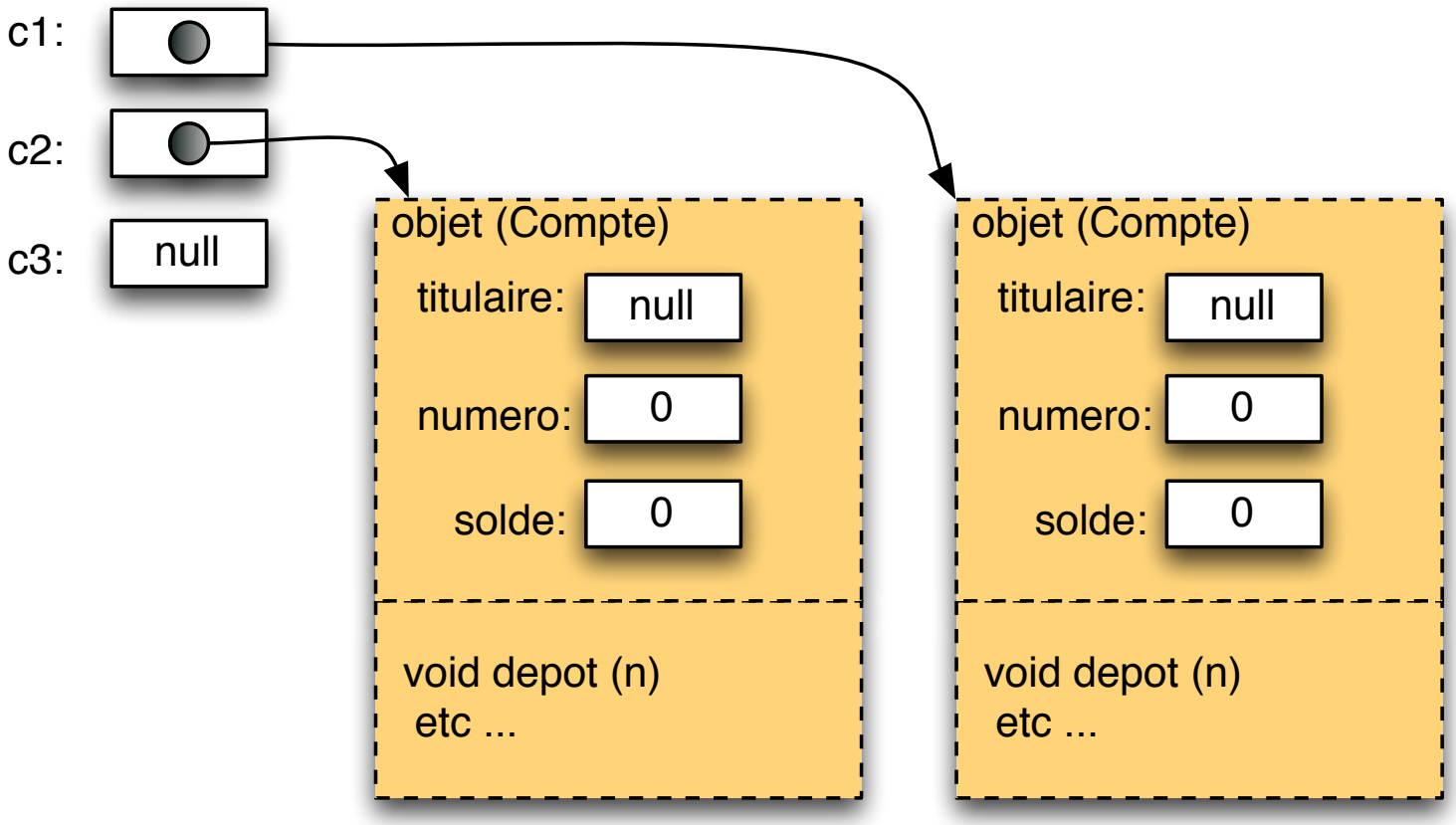
  ⇒ sans initialisation, c contient null.
- Création et initialisation : `new` + appel constructeur(arguments)

```
Compte c1 = new Compte();//constructeur par defaut
Compte c2 = new Compte();
Compte c3;     // contient null
```

- *c1 et c2 sont des instances de Compte*

  ⇒ ils pointent vers 2 objets de structure identique et adresses du tas différentes.

Les variables et objets après exécution



Deux sortes de méthodes

1. méthodes d'instance :

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

   - pas de mot clé `static`
   - accès aux variables d'instance via la variable `this`

2. méthodes statiques : pas d'accès aux variables d'instance ; `this` est interdit.

```
public static Compte creerCompte()){
    int num = Terminal.lireInt();
    double s = Terminal.lireDouble()
    return new Compte(num,s);
}
```

La variable `this` = objet courant

La variable `this`

- À l'exécution, `this` contient l'adresse de l'objet courant. C'est l'objet sur lequel la méthode en cours exécution a été invoquée.
- `this.var` donne accès à la variable d'instance `var` de l'objet courant.

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

- `depot` a été invoquée sur l'objet c1 ;
- pendant l'exécution de `depot` ⇒ `this == c1`
- le code `this.solde` dans `depot` correspond à `c1.solde` (le solde de c1).

Appel aux méthodes d'instance

- Toujours appelées sur un objet (l'appel est préfixé par l'objet sur lequel se fait l'appel) :

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

- agissent sur l'état (variables d'instance) de cet objet

```
c1.depot(50); // +50 dans c1.solde
c2.depot(30); // +30 dans c2.solde
c1.getSolde(); // renvoie le solde courant de c1
```

- à l'exécution, la variable `this` contient l'adresse de l'objet courant (ici, `c1`), i.e. l'adresse de l'objet sur lequel la méthode en train de s'exécuter (ici `depot`) a été invoquée ⇒ dans le code de depot : `this.solde = c1.solde`

Méthodes `accesseur` et `mutateur`

Deux termes pour parler de certaines catégories de méthodes d'instance :

- accesseur(*getter*) : méthode d'instance qui retourne la valeur d'un attribut.

```
public int getNumero() {
    return this. numero;
}
```

- mutateur (*setter*) : méthode d'instance qui modifie un attribut, et qui souvent ne retourne aucun résultat.

```
public void setPrixArticle(double p) {
    this.prixArticle = p;
}
```

Bien entendu, ces catégories ne couvrent pas toutes les méthodes.

Encapsulation

```
private int numero;          // numero
private double solde;        // solde courant
private String titulaire;    // nom titulaire
```

Encapsulation

- protection d'attributs : `private` ou `protected`
- Seules les méthodes d'instance de la classe y ont accès.
- Conséquence : ajout de méthodes *accesseur* pour obtenir leurs valeurs en dehors de l'objet. Ex : `getNumero()`, `getSolde()` ...

Composition et délégation

```
public calss Banque {
  private Compte [] cpts = new Compte[1000];
  private nb = 0;

  public nouveau(int n, String t, double s){
     cpts[nb] =  new Compte(n,t,s); nb++; }

  public getSolde(int numCpt) { ... }
  public retrait(numCpt, double montant){ ...}
```

- `Banque` est composé d'un tableau avec tous ses objets `Compte`
- c'est *un objet composite*
- opérations devront *agir sur les objets composants*
- Ex : `getSolde(num)` obtenir le solde de l'objet compte de numéro num.

Composition et délégation(suite)

```
public class Banque {
  private Compte[] cpts= new Compte[50];
 // Retourne le compte de numero n
  private Compte getCompteDeNum(int n) { ... }
  public double getSolde(int n) {
     Compte c = getCompteDeNum(n);
     return c.getSolde(); // DELEGATION
  }    ...
```

`getSolde(n)` : opération agissant sur les composants

- trouver l'objet c ayant le numéro de compte n ;
- y appliquer la méthode propre à cet objet : `c.getSolde()`

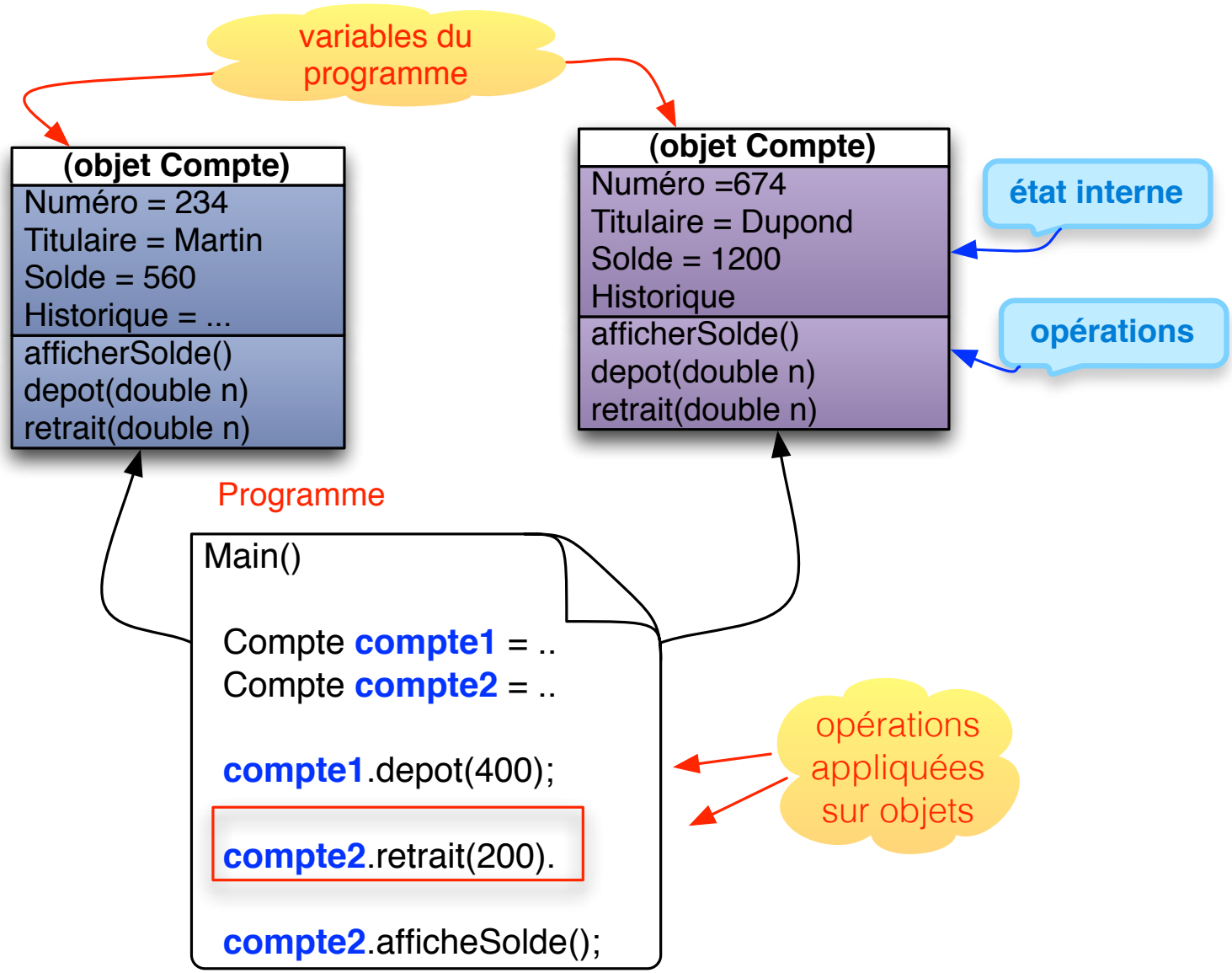
Méthode `getSolde(num)` $\Rightarrow$ « délègue » l'action à réaliser sur une méthode propre à l'objet composant concerné.

Qu'est-ce qu'un programme orienté objet ?

Programme orienté objet

- le programme manipule plusieurs objets Compte,
- le programme applique des méthodes (d'objet) sur certains de ces objets
- ces méthodes modifient l'état interne des objets concernés.

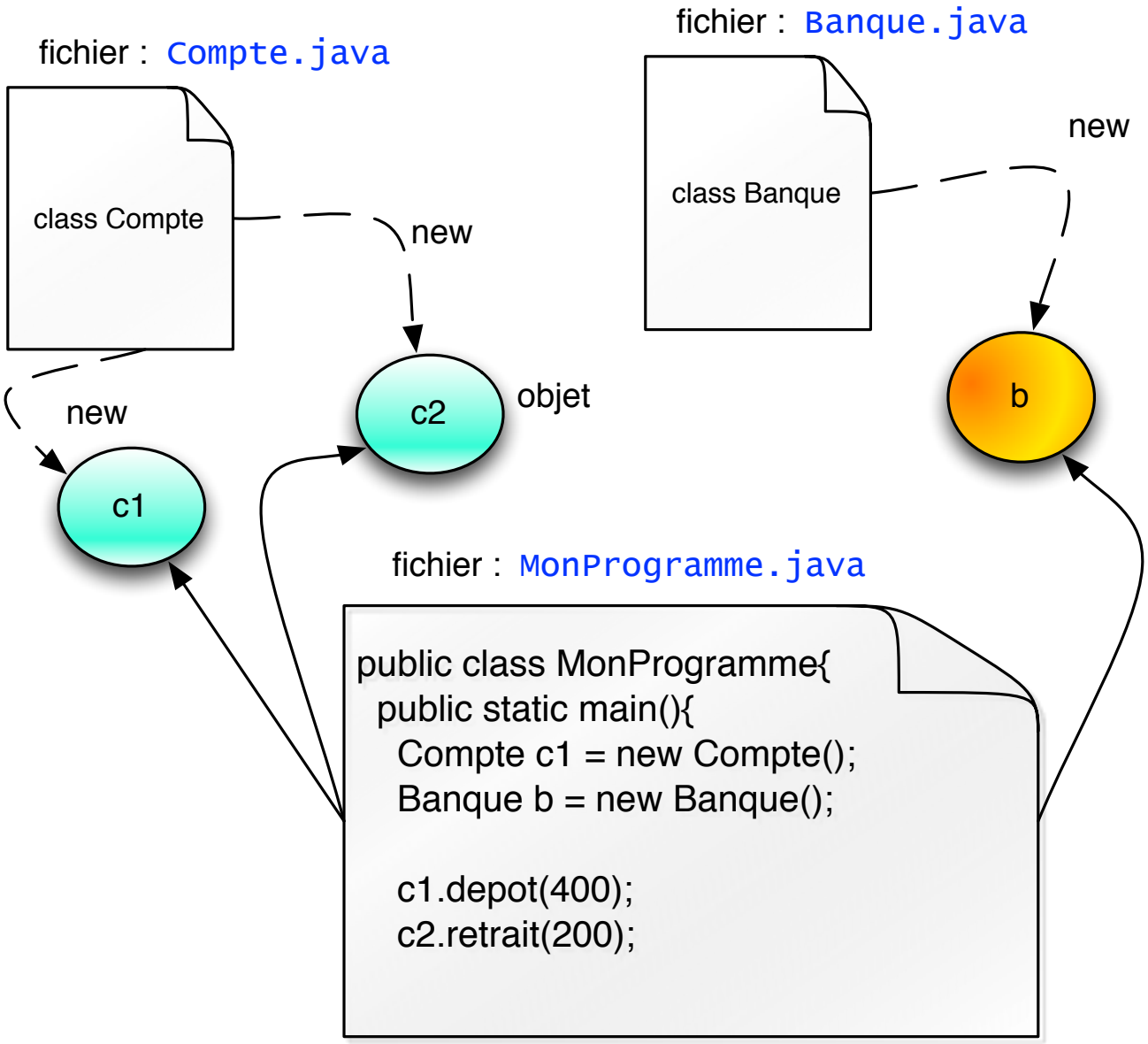
Un programme OO ⇒ utilise des objets



En Java : deux sortes de classes

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

Programme OO = Ensemble de classes (et fichiers)



Répresentation des objets en mémoire

Les objets sont répresentés par des références

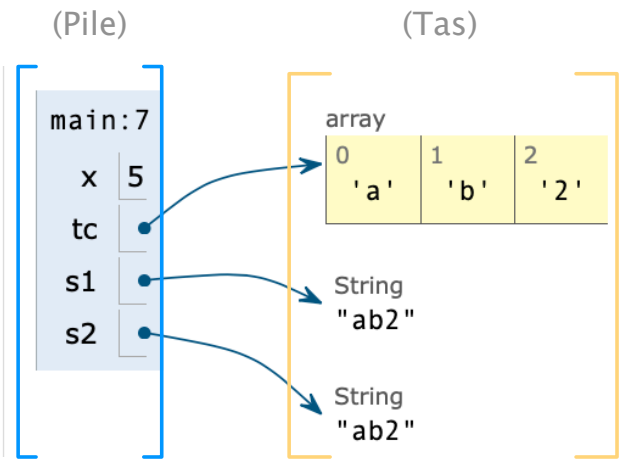
En PythonTutor :

- La Pile est nommée *Frames*
  - *frame* : mémoire d'exécution pour une méthode (ici, main) ;
- Le Tas est nommée *Objects*
- les variables du main sont dans la pile ;
- celles de type référence « pointent » vers le tas ;
- c.a.d., elles contiennent une adresse du tas.

Objets répresentés par des références (dessin)

On voit ici que les objet instance de String sont representés par des pointeurs.

```
public class YourClassNameHere {
    public static void main(String[] args) {
      int x = 5;
      char [] tc = {'a', 'b', '2'};
      String s1 = new String(tc);
      String s2 = "ab2";
    }
}
```



Affectation entre variables référence = partage de données

```
int [] t1, t2;
t1 = {1,2};
t2 = {10,2, 9, 7};
t1 = t2;    // <--- Affectation
```

Affectation entre variables référence (de types compatibles)

Comportement :

- copie du contenu (une adresse) d'une variable vers l'autre ;
- après coup, elles contiennent la même adresse.

t1, t2 partagent la mêmes donnée ! ⇒ si on change une composante dedans, les deux variables « voient » ce changement.

Egalité == sur objets

Qu'affiche ce programme ?

```
  char [] tc = {'a', 'b', '2'};
  String s1 = new String(tc);
  String s2 = s1;
  String s3 = "ab2";
   if (s1==s2){ System.out.println("s1==s2");
   } else {
        System.out.println("s1!=s2");  }
   if (s1==s3){
        System.out.println("s1==s3");
   } else {
        System.out.println("s1!=s3");  }
```

```
> java Chap12d
s1==s2
s1!=s3
```

Passer en paramètre un objet

Un objet Date contient 3 variables d'instance : jour, mois, annee.

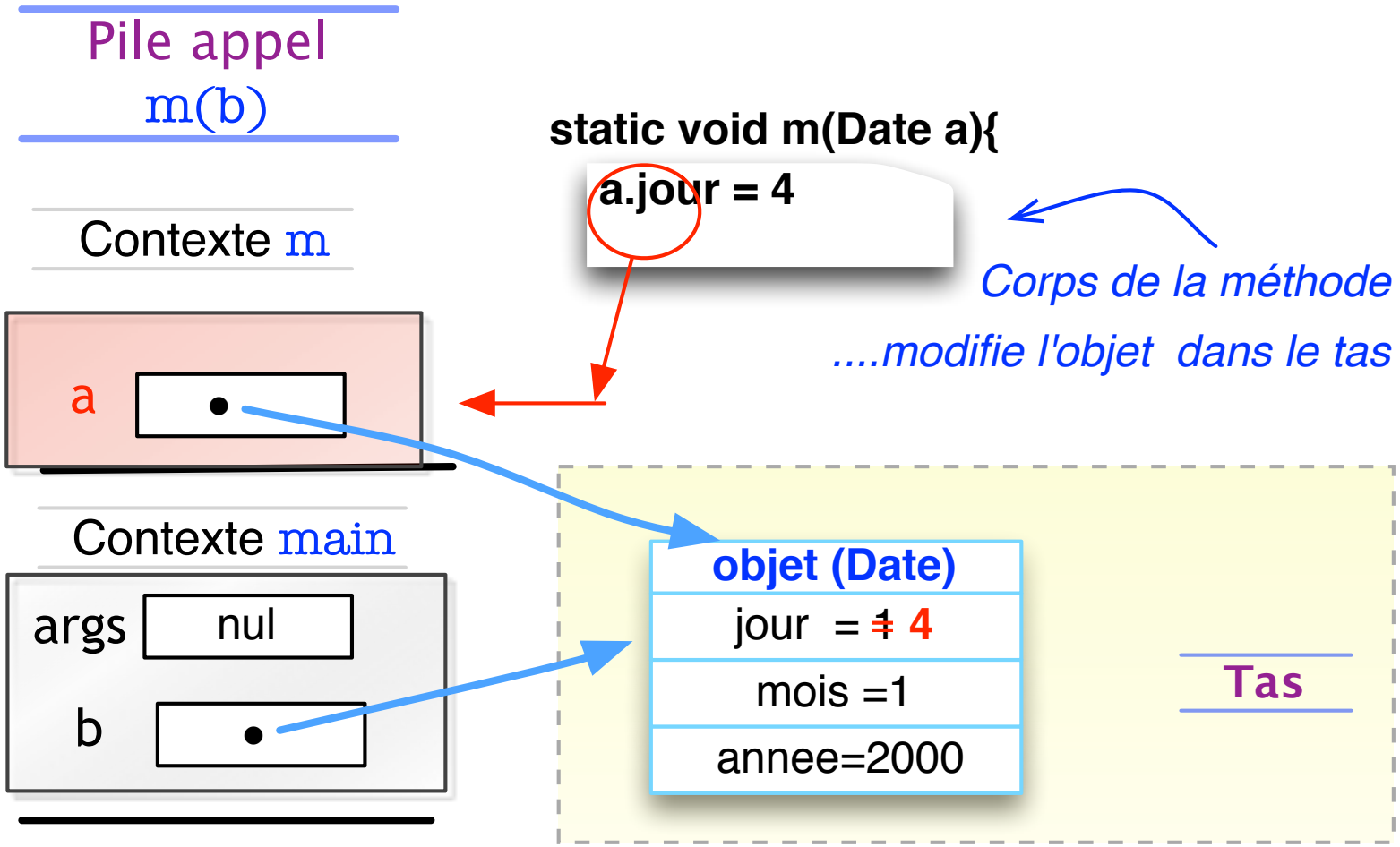
```
static void m(Date a){
    a.jour=4;
}
public static void main(String [] args){
    Date b = new Date(1,1,2000);
    m(b);
```

Exécution de `m(b)` :

1. `a.jour = 4` ⇒ modifie la variable `jour` dans le tas.
2. Retour au main. La variable `a` n'existe plus. Dans le tas, l'objet référencé par `b` **a été modifié**.

`b.afficherDate()` ⇒ affiche `4 / 1 / 2000`

Paramètre objet (dessin)



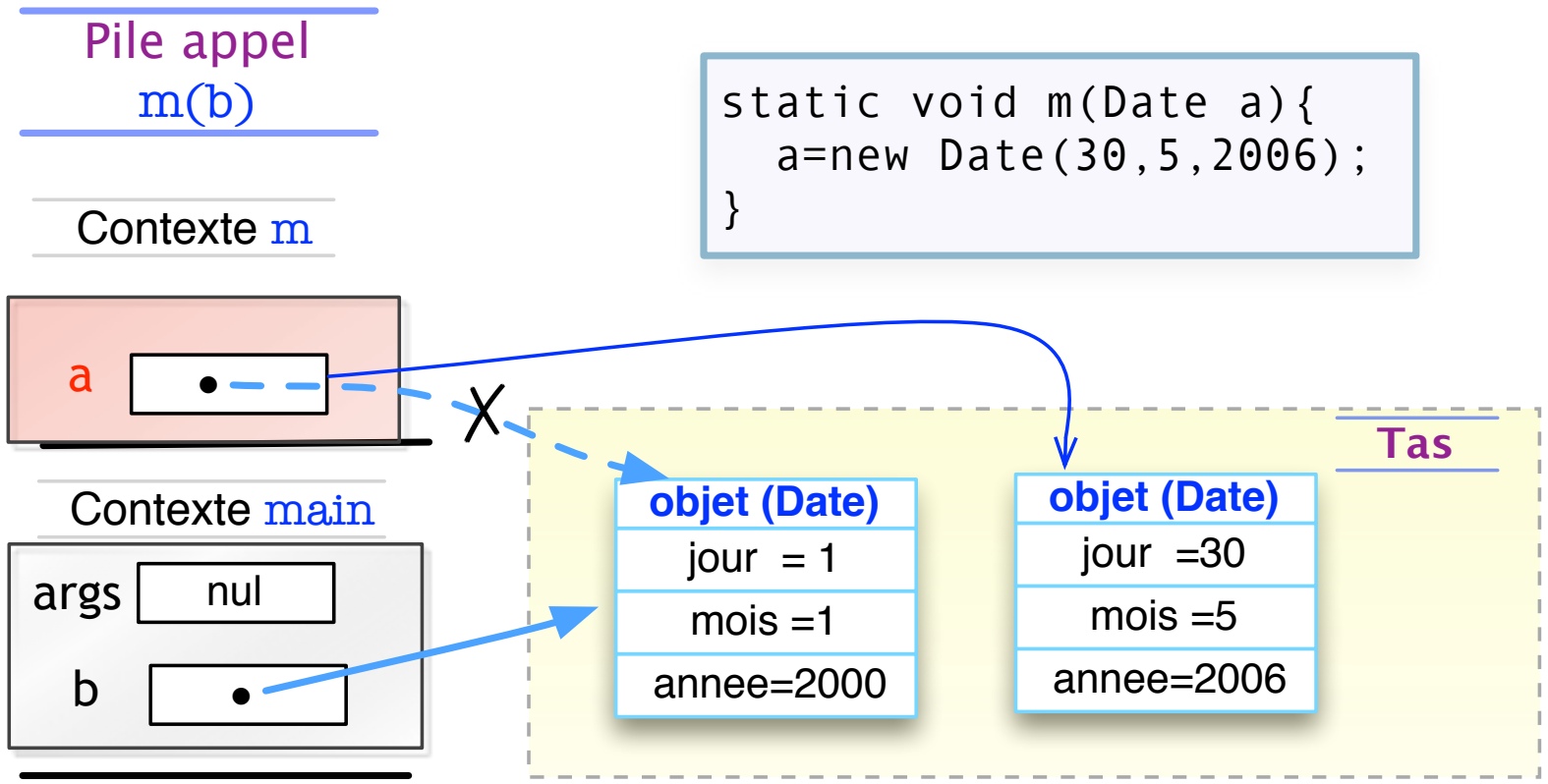
Passer en paramètre un objet (bis)

Un autre exemple de passage de paramètre d'un objet.

```
static void m(Date a){
    a =  new Date(30,5,2006);
}
public static void main(String [] args){
    Date b = new Date(1,1,2000);
    m(b);
    b.afficherDate();
```

Quelle est la date affichée ici ?

Exemple bis avec un dessin




```
static void m(Date a){
  a=new Date(30,5,2006);
}
```

la méthode ne modifie pas la référence passée mais un objet pointé localement.

Passage de paramètres

```
static void m(<un-type> p){...
 }

static void main... {
 <un-type> x = ...;
 m(x);    // appel de methode
```

- Lors d'un appel de méthode `m(x)`, on passe à m **la valeur contenue dans la variable** x.
- Cette valeur est recopiée dans la variable `p` du contexte de m. Elle est utilisée pendant l'exécution de m.
- Si m réalise une affectation sur p, cela **n'a aucune incidence** sur la valeur de x, et cela quelque soit le type de x (primitif ou référence).

Quelques principes de programmation à adopter

Principe 1 : pratiquer l'encapsulation

- Les variables d'instance déclarées `private`, les méthodes déclarées (pour la plupart) `public`
  - il faudra alors équiper la classe de méthodes *acceseur* pour retourner la valeur des attributs,
  - et parfois (mais pas toujours), des méthodes pour les modifier.
  - Avantage : les modifications des instances sont circonscrites aux méthodes de la classe ! (plus simple, moins de bugs).

Principe 2 : initialisation par constructeurs déclarés + état cohérent

- Déclaration systématique d'un ou plusieurs constructeurs pour initialiser intégralement toutes les variables d'instance ;
- Au passage : on refléchit aux conditions de cohérence de l'état interne ;
- si nécessaire : on interdit (via un échec) la création d'objets incohérents

Principe 3 : Réécrire la méthode `toString()`

- `toString()` est une méthode définie par défaut sur n'importe quel objet
- par défaut, cette méthode renvoie une chaîne correspondant à l'identité de l'objet (son adresse mémoire)
- on souhaite en général qu'elle retourne une chaîne qui correspond à l'état courant de l'objet

Exemple : méthode `toString()` de la classe Compte :

```
public String toString() {
  return "Numero: "+this.numero+
       ", Titulaire: "+ this.titulaire
        + ", Solde: "+this.getSolde();
}
```

Principe 4 : favoriser la délégation

- Si on a des objest composites
- concevoir les opérations du composite autant que possible par délégation.
- parfois il peut être necéssaire de repenser les opérations du composant
- pour en ajouter celles pouvant manquer !