Introduction à la programmation objet

Virginia Aponte

CNAM-Paris

March 2, 2017

1. Classes, objets et POO

Les données en programmation

- Données élémentaires : une variable contient une donnée simple. Ex: variables avec un entier, nombre à virgule, caractère, booléan, etc;

- Données structurées : une variable contient plusieurs données, pas forcément de même type.

  Ex: tableaux, Strings (même type);
  enregistrements, objets (types différents).

  Utilité:

  - regrouper plusieurs variables liées entre elles;
  - structurer/faciliter la programmation et la réutilisation!

Exemple: comptes bancaires

Données d'un compte:

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

$\Rightarrow$ données de types différents.

Opérations sur un compte:

- initialisation nom du titulaire, numéro du compte, solde courant;
- obtenir le solde courant, réaliser un retrait, un dépôt;
- garder une trace de chaque opération de retrait ou de dépôt dans l'historique.

$\Rightarrow$ opérations agissent *sur plusieurs données* du compte.

Objets: données structurées (ou complexes)

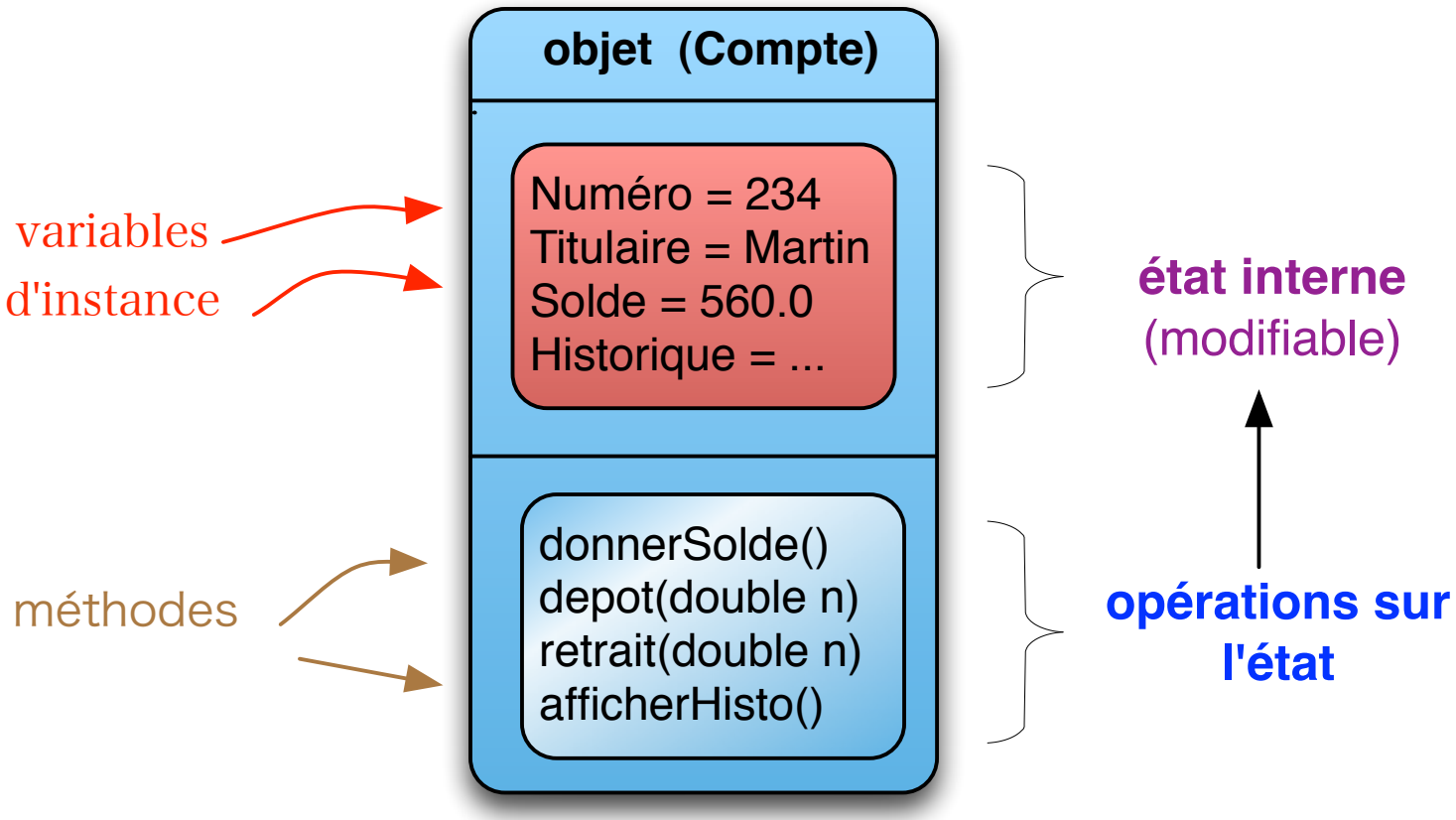
Objet

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

Exemple: un objet compte bancaire contient

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

Un objet Compte



Objet = état interne + opérations sur l'état

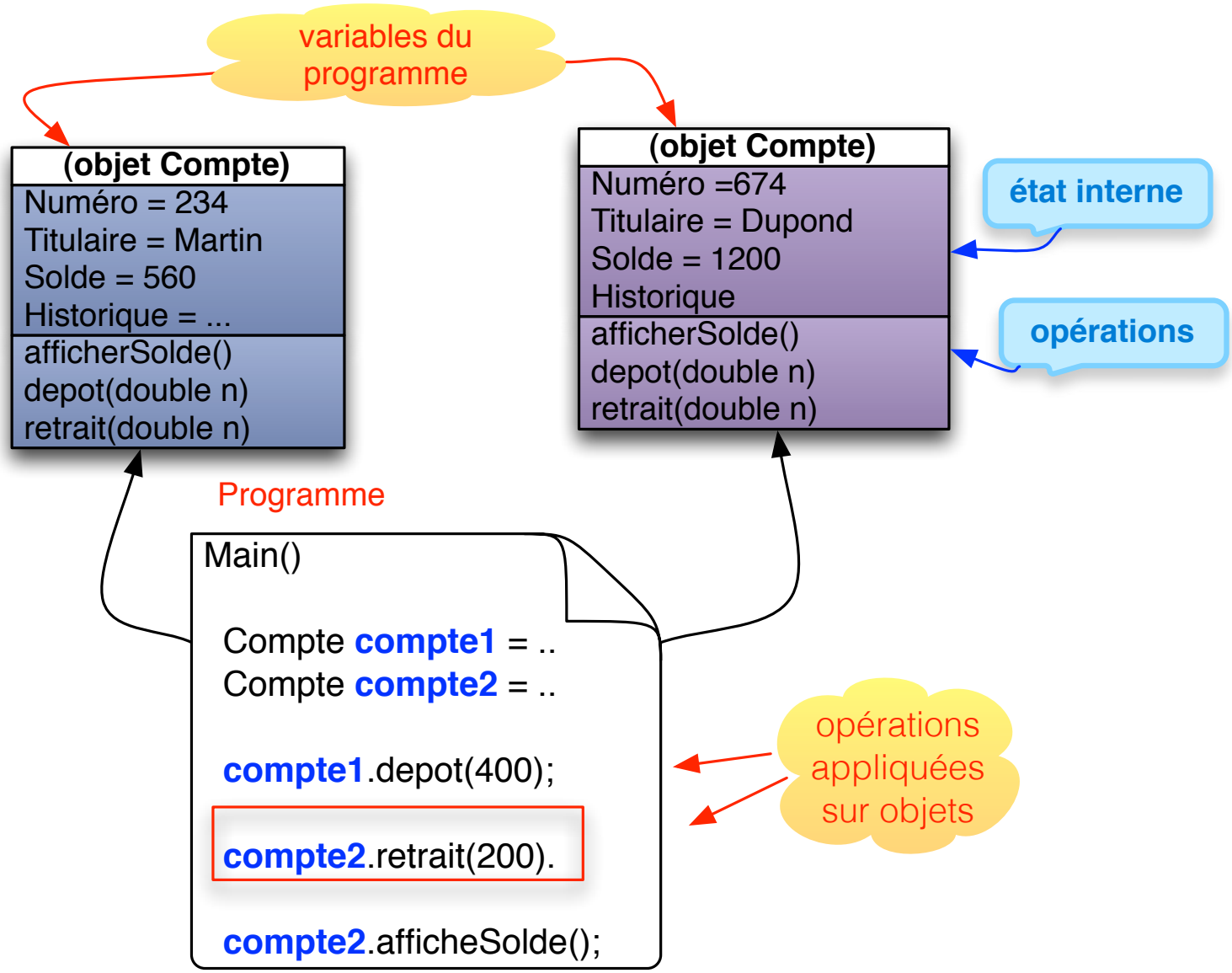
Gestion de comptes en POO

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 ⇒ utilise des objets



Classe = Moule à objets = Type

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

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

On dit: 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;

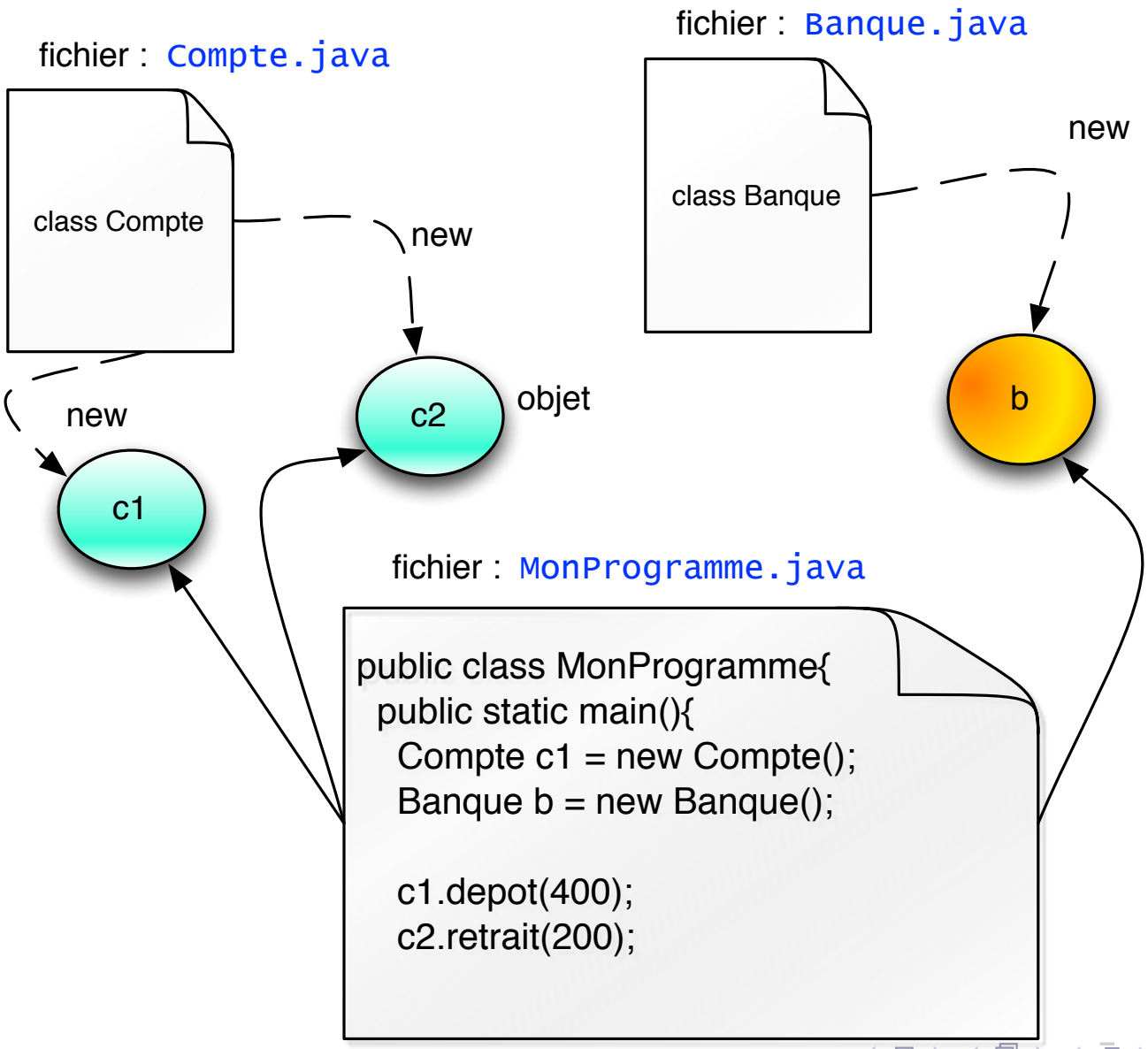
Classe

décrit structure interne et opérations des objets $\Rightarrow$ "moule à objets".

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.

Programme OO = Ensemble de classes (et fichiers)



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;
```

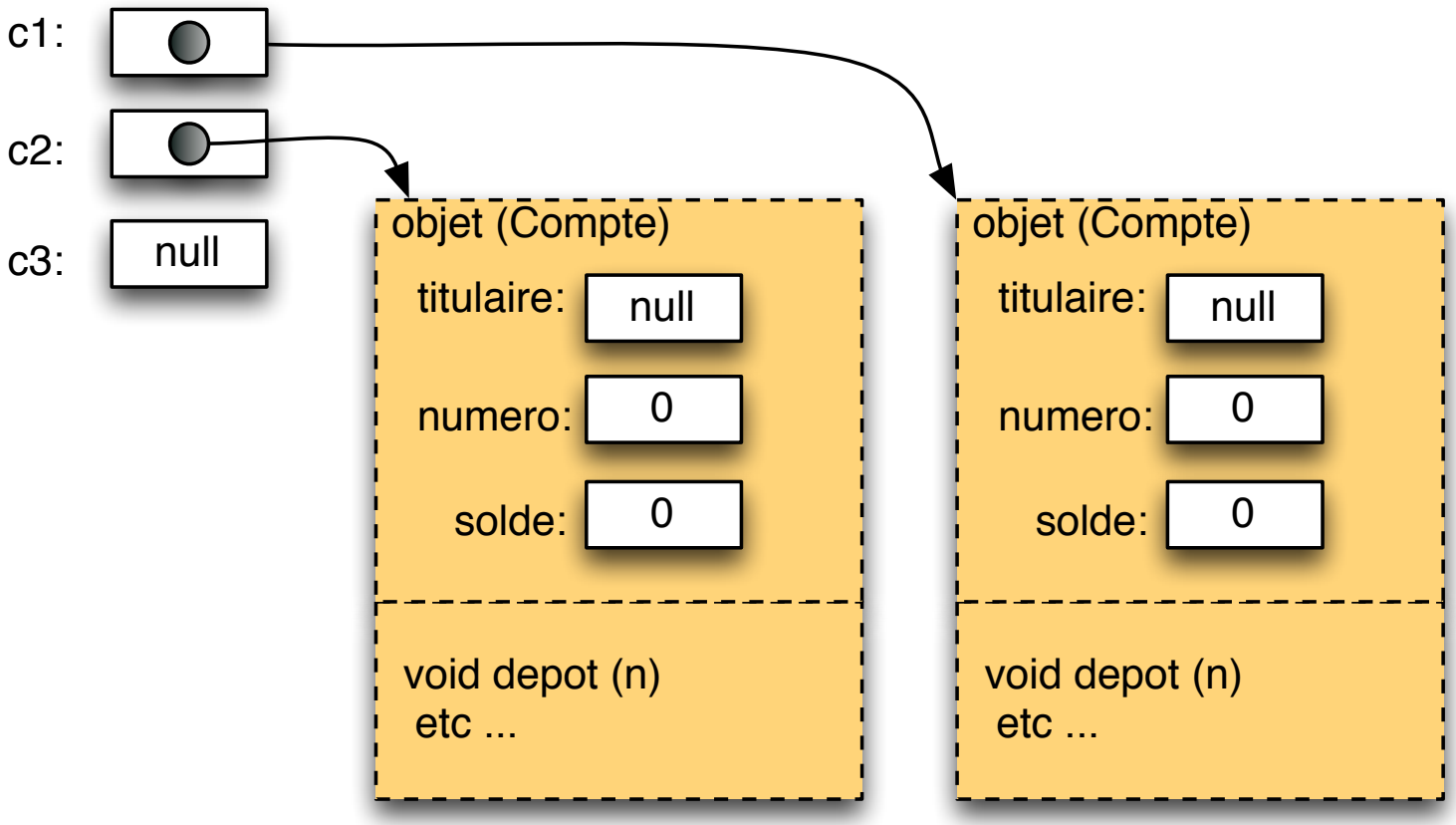
Création, initialisation de l'objet (**en mémoire**)

```
c1 = new Compte();
```

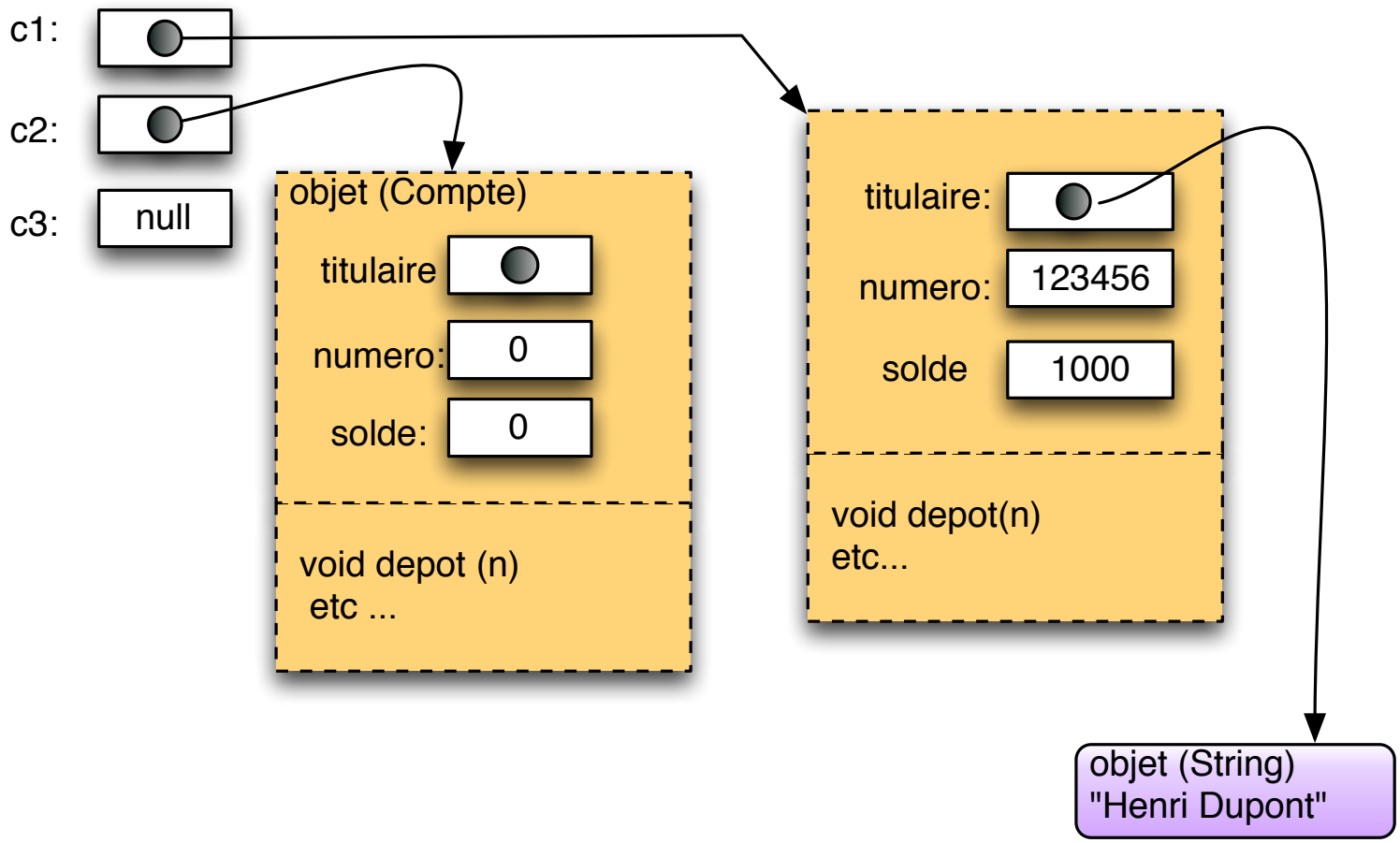
Après création

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

Les objets du programme après création



Les objets du programme après modification variables



Les méthodes dans les classes à objets

Deux sortes:

- non statiques :

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

  - pas de mot clé `static`
  - modélisent les comportéments possibles de l'objet;

- statiques : ne touchent pas aux variables internes.

Invoquer des méthodes non statiques

Méthodes non statiques:

- toujours appelées sur un objet;

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

- on ne peut appeler que les méthodes définies dans la classe ayant servi à créer l'objet;
- agissent sur l'état interne (variables internes) de cet objet

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

2. Les interfaces (outil de spécification)

Interfaces en Java: c'est quoi?

Interface

- Ensemble de profils de méthodes correspondant au minimum de fonctionnalités requises dans des classes.
- Mot-clé `interface` et contient:
  - un ensemble d'entêtes de méthodes;
  - aucune implantation pour les méthodes;
  - aucune variable;

Interfaces: à quoi cela sert?

- Type : spécifier le type d'un objet
  - en donant uniquement *la liste de métodes disponibles (+ leurs types)*;
- Contrat :
  - cahier de charges sur chaque méthode (javadoc);
- Documentation :
  - . . . et un contrat (javadoc) sert aussi de *documentation*;

En pratique

Les bibliothèques Java sont batîes sur diverses interfaces: on doit savoir les lire + utiliser.

Exemple: spécifier les objets déplaçables

Un objet *déplaçable* est caractérisé par:

- une position: des coordonnées (abscise et ordonnée);
- une opération de deplacement (de sa position).

```
public interface Displaceable {
 /** Retourne coordonne abscice (position). */
  public int getX();
 /** Retourne coordonne orodonnee (position). */
  public int getY();
 /** Deplacement avec differentiel (dx,dy) */
  public void move(int dx, int dy);
}
```

Syntaxe d'une interface

```
public interface Displaceable {
  public int getX();
  public int getY();
  public void move(int dx, int dy);
}
```

- Pas de variables : on ne dit pas comment un objet "deplaçable" est construit;
- Profils de méthodes : on donne la liste d'opérations de l'objet, avec description des types pour appels;
- Pas de corps de méthodes : on ne dit pas comment sont implantées les méthodes.

Une classe qui correspond à une interface

Une classe implante une interface

- si elle fournit au moins autant de méthodes (avec types compatibles) que celles requises par l'interface;
- syntaxe: signalé par le mot-clé `implements`

dans ce cas on dit que:

$\Rightarrow$ la classe satisfait le contrat établit par l'interface

c'est vérifié par le compilateur!

Erreur de compilation si dans la classe les méthodes ne correspondent pas à celles de l'interface.

Exemple d'implantation

```
public class Point implements Displaceable {
  private int x, y;

  public Point(int x0, int y0) {
    x = x0; y = y0;
  }
  public int getX() { return x; }
  public int getY() { return y; }
  public void move(int dx, int dy) {
    x = x + dx; y = y + dy;
  }
}
```

- en magenta : les méthodes requises par l'interface
- *le reste de l'implantation* est "laisse libre" par le contrat.

Suite exemple: autre implantation

- ici, l'état des objets a une structure différente (celle des cercles);
- on implante le mouvement par une déplacement du cercle;

```
public class Circle implements Displaceable {
  private Point center;
  private int radius;
  public Circle(Point initCenter, int initRadius) {
      center = initCenter; radius = initRadius;
  }
  public getRadius() { return radius; }
  public int getX() { return center.getX(); }
  public int getY() { return center.getY(); }
  public void move(int dx, int dy) {
    center.move(dx, dy);
  }
}
```

Les interfaces sont des types

- Type des variables: nous pouvons déclarer une variable avec une interface en lieu et place de son type;

```
Displaceable d;
```

- Implantation: cette variable pourra prendre la valeur de n'importe quel objet dont la classe implante l'interface

```
Displaceable d1 = new Point(1,2);
Displaceable d2 = new Circle(new Point(1,2), 3);
```

- Opérations restreintes par ce qui est décrit dans l'interface:

```
d2.move(-1,1);
d2.getX();    // donne 0.0
d2.getY();    // donne 3.0
d2.getRadius(); // erreur de compilation
```

Interfaces et abstraction

- *Nom unique* pour toutes les sortes d'objets déplaçables.
- on peut écrire du code générique, sur tous ces objets, sans s'occuper de leur "veritable nature";

```
class Bouger {
 public void moveItAll (ArrayList<Displaceable> s,
                        int dx, int dy) {
   for (int i=0; i < s.size(); i++) {
      s.get(i).move(dx,dy);
   }
 }
 public void exemple () {
   Displaceable s1 = new Point(5,5);
   Displaceable s2 = new Circle(new Point(0,0),100);
   ArrayList<Displaceable> l = new ArrayList<Displaceabl
   l.add(s1);  l.add(s2);
   moveItAll(l,5,10);
 }
}
```

Les interfaces sont des contrats

Une interface spécifie:

- les méthodes qu'un objet doit possèder;
- le comportement de ces méthodes (javadoc, pré et post conditions);
- éventuellement une *propriété invariante de l'état interne*: sans cela l'objet peut devenir incohérent.
  - interface pour les dates: la date interne est toujours correcte;
  - interface pour les comptes: un compte n'a jamais de solde débiteur;
- toute implantation *doit maintenir* la cohérence interne des objets;

Cohérence interne $\Rightarrow$ on parle d'invariant d'état.

Exemple: interface pour les comptes non débiteurs

- cohérence de l'état interne: `this.getSolde()` ≥ 0
- les méthodes sont spécifiées avec un contrat (javadoc);

```
public interface CompteNonDebiteur {
   /**
    * Retourne solde courant superieur ou egal a 0. */
   public double getSolde();

  /**
   * Retrait du montant m.
   * @throws IllegalArgumentException
        si avant retrait  getSolde()<m */
   public void retrait(double m);
   ....
}
```

Interfaces, abstraction et réutilisation

Les interfaces décrivent *de manière abstraite* les comportement des objets.

- des bibliothèques Java définissent des opérations très génériques sur des objets décrits par une interface;
- on peut les utiliser avec n'importe quelle classe qui les implante;
- ces bibliothèques deviennent alors hautement réutilisables.

Interfaces et bibliothèques Java

Il est indispensable de connaître les interfaces pour employer les bibliothèques!

Exemple: interface `Comparable` (bibliothèque Java)

En Java, les opérations de tri sont:

- définies sur les objets de n'importe quel type `T` (objet),
- à condition qu'ils soient comparables entre eux (selon un ordre).

Ces objets doivent implanter l'interface `Comparable<T>`:

```
interface Comparable<T> {
  /**
   * Retourne un entier:
        -  positif si this est plus grand que o,
        -  0 si egaux,
        -  negatif sinon
   */
  int compareTo(T o);
}
```

Interfaces, contrats et passage à l'échelle

Le développement de logiciels de grande taille

- se fait souvent par plusieurs équipes, en parallèle;
- les objets conçus par l'équipe A pourront utiliser des objets conçus par l'équipe B;
- comment s'accorder sur ce que doit faire chaque classe écrite par des équipes différentes?

Aide: établir des contrats précis pour chaque classe;
$\Rightarrow$ utiliser des interfaces.

3. Démo avec interfaces

4. Tester les classes

Que teste-t-on dans une classe?

Classe : état interne + méthodes lecture/modification état interne

- on teste *la cohérence* de l'état interne $\Rightarrow$
  - ex: pour un objet document qui utilise des fichiers images, les fichiers existent bien, et contiennent le bon format d'images.
- on teste comportement méthodes $\Rightarrow$ contrats de méthodes:
  - la méthode se comporte comme dit dans son contrat...
  - ...sauf que, certaines méthodes ne renvoient pas de résultat, mais modifient l'état interne $\Rightarrow$
    - tester que modification correspond au contrat
    - tester état interne reste cohérent

Quelle forme prend une spécification de classe?

- c'est une interface au sens Java
- décrit des contrats pour **les 2 parties** d'un objet:
  - contrat pour l'état interne :
    - décrit ce qu'est un état interne cohérent
  - contrats pour toutes les méthodes de la classe;
    - décrit le contrat de la méthode (arguments, préconditions, ce qu'elle retourne ou modifie)

Exemple: spécifier comptes non débiteurs

- côté données (état interne): le solde d'un compte (un double) n'est jamais négatif;
- côté opérations souhaitées:
  - `double getSolde();`
  - `void retrait montant(m);`
  - `void depot montant(m);`

Exemple (2): interfaces comptes non débiteurs

- cohérence de l'état interne: `this.getSolde()` ≥ 0
- les méthodes sont spécifiées avec un contrat (javadoc);

```
  /**
   * Comptes dont le solde n'est jamais negatif */
public interface CompteNonDebiteur {
  /**
   * Retourne solde courant superieur ou egal a 0. */
  public double getSolde();

 /**
  * Retrait du montant m.
  * @throws IllegalArgumentException
       si avant retait  getSolde()<m */
  public void retrait(double m);
  ....
}
```

Exemple (3): une implantation

```
public class CompteND implements CompteNonDebiteur {
      private double solde;
      public CompteND(double init){
        if (init<0)
           throw new IllegalArgumentException();
        this.solde = init;
      }
      public double getSolde() { return this.solde; }
      public void retrait(double m) {
          if (solde-m<0)
             throw new IllegalArgumentException();
          this.solde = this.solde-m;
      }
      public void depot(double n){
         if (solde+n<0)
            throw new IllegalArgumentException();
         this.solde = this.solde+n;
      }
}
```

Tester la méthode m d'une classe

Dans la méthode qui contient un cas de test (qui n'échoue pas):

1. créer un objet c (via le constructeur), en lui donnant des valeur initiales:
   - correspondant à un état interne cohérent,
   - pertinentes pour le cas à tester;
2. invoquer c.m(..) avec les argument pertinents pour le cas de test
3. tester en sortie que
   - le résultat ou état interne est bien ce qui était prévu d'après le contrat
   - l'état interne est toujours cohérent.

Tester le constructeur en cas d'échec

Le constructeur doit échouer à créer des objets d'état incohérent:

```
@Test(expected = IlegalArgumentException.class)
  public void testFailConstruction() {
      System.out.println("failToConstruct");
      new CompteND(-10.0);
}
```

Teste un cas d'échec du constructeur.

Tester le constructeur en cas de succès

Teste si le constructeur initialise correctement l'objet.

```
@Test
  public void testPostConstruction() {
      System.out.println("Postcondition␣Construct");
      CompteND c = new CompteND(10.0);
      assertEquals(10.0, c.getSolde(), 1e-15);
}
```

`retrait(n)` respecte-t-il son contrat?

```
@Test(expected = IllegalArgumentException.class)
  public void testRetraitEchec()  {
      System.out.println("retrait␣fails");
      CompteND c = new CompteND(80);
      c.retrait(100);
}
```

Teste un cas d'erreur.

`retrait(n)` en cas de succès

```
@Test
  public void testRetraitOK()  {
        CompteND c = new CompteND(100.0);
        c.retrait(30);
        assertEquals(70.0, c.getSolde(), 1e-15);
}
```

Tester l'invariant via une méthode dans la classe Compte

Parfois il est nécessaire d'ajouter une méthode qui teste si l'invariant est valide.

Elle est définie dans la classe CompteND,
car elle doit accéder aux variables privées de l'objet!

```
public invariantOK(){
        return this.solde >=0;
}
```

5. Demo de test de la classe CompteND