Programmation Avancée et Application

Bases de la POO en Java

Jean-Guy Mailly

jean-guy.mailly@u-paris.fr

LIPADE - Université de Paris
http://www.math-info.univ-paris5.fr/∽jmailly/

Bases de la POO en Java

1. Objets et classes

    Créations de classes et d'objets en Java

    Encapsulation

Objets et classes

Qu'est-ce qu'un objet ?

- Entité créée et manipulée par un programme correspondant à une entité (concrète ou abstraite) du monde réel

5 répertoires téléphoniques avec 100 personnes chacun	5 objets `Repertoire`
	500 objets `Personne`
Un cinéma qui diffuse 20 films avec 73 acteurs différents	1 objet `Cinema`
	20 objets `Film`
	73 objets `Acteur`
Une promo de licence avec 200 étudiants qui ont 5 notes pour chacune des 8 unités d'enseignement qu'ils étudient	1 objet `Promotion`
	200 objets `Etudiant`
	8 objets `UniteEnseignement`
	8000 objets `Note`

Qu'est-ce qu'une classe ?

- Certains objets représentent des entités du monde réel qui appartiennent à une même catégorie, sont des instances d'un même concept
- La *classe* est l'entité informatique qui correspond à ce concept

Qu'est-ce qu'une classe ?

- Certains objets représentent des entités du monde réel qui appartiennent à une même catégorie, sont des instances d'un même concept
- La *classe* est l'entité informatique qui correspond à ce concept

Monde réel	Objets	Classe
Marlon Brando	un objet `brando`	`Acteur`
Robert De Niro	un objet `deNiro`	
Al Pacino	un objet `pacino`	
Talia Shire	un objet `shire`	
...	...	
Programmation avancée	un objet `progAv`	`UniteEnseignement`
Génie logiciel	un objet `genieLog`	
Algorithmique avancée	un objet `algoAv`	
...	...	

Création de classes en Java

- Chaque fichier de code source porte l'extension `.java`
- Le préfixe du nom du fichier doit correspondre au nom de l'unique classe publique définie dans le fichier
- Exemple : fichier `MaClasse.java`

```
public class MaClasse {
  // ...
}
```

- On verra dans la suite qu'on peut avoir plusieurs classes dans le même fichier (celles qui ne correspondent pas au nom du fichier ne peuvent pas être déclarées `public`)
- Pour le début du cours, on fait l'hypothèse simplificatrice « 1 fichier = 1 classe »

Membres d'instance

- On a parlé des membres *de classe* précédemment (`static`), qui ne nécessitent pas de créer d'instances de la classe pour les utiliser
- Les membres (attributs ou méthodes) non `static` sont les membres *d'instance* : on doit créer un objet de cette classe pour les utiliser

Exemple :

```
public class Acteur {
  private String nom ;
  private String prenom ;
  private String nationalite ;

  // ...
}
```

- Les attributs `nom`, `prenom` et `nationalite` n'ont pas de sens s'ils ne sont pas associés à un acteur précis

Rappel : méthodes

- Une méthode est définie par une signature suivie d'un bloc d'instructions (appelé le *corps* de la méthode)
- Signature d'une méthode :
  - Liste de modificateurs
  - Type de retour
  - Nom de méthode
  - Liste de paramètres : type et nom de chaque paramètre (on parle aussi d'arguments)

Rappel : modificateurs

- Modificateurs d'accès :
  - `public` : on peut utiliser le membre depuis n'importe quel endroit du programme
  - `protected` : on peut utiliser le membre depuis les classes filles et les classes du même package
  - par défaut : sans modificateur d'accès, on peut utiliser le membre depuis les classes du même package
  - `private` : on peut utiliser le membre uniquement à l'intérieur de la classe où elle est définie

Méthodes et surcharge

- Plusieurs méthodes qui servent globalement à la même chose peuvent porter le même nom, à condition d'avoir des paramètres différents (types et/ou nombre de paramètres) : c'est la surcharge
- Exemple :

```
public class Film {
  public void ajoutActeur(Acteur acteur){
    // ...
  }

  public void ajoutActeur(String nom, String prenom,
                          String nationalite){
    // ...
  }
}
```

Constructeurs

- Les constructeurs sont des méthodes particulières qui servent à créer des instances de la classe
- La définition d'un constructeur est différente de la définition d'une méthode « classique »
  - Pas de type de retour
  - Même nom que la classe (y compris la majuscule au début !)
- Principale utilité : initialiser les attributs

Constructeurs : premier exemple

```
public class Acteur {
  private String nom ;
  private String prenom ;
  private String nationalite ;

  public Acteur(String n, String p, String nat){
    nom = n ;
    prenom = p ;
    nationalite = nat ;
  }
}
```

Constructeurs : surcharge

- Les constructeurs peuvent aussi être surchargés
- Exemple : on initialise un acteur sans connaître sa nationalité

```
public class Acteur {
  // ...

  public Acteur(String n, String p, String nat){
    nom = n ;
    prenom = p ;
    nationalite = nat ;
  }
  public Acteur(String n, String p){
    nom = n ;
    prenom = p ;
    nationalite = "" ;
  }
}
```

Le mot-clé this

- Le mot-clé **this** a plusieurs rôles, dont :
  - lever les ambiguités si un attribut porte le même nom qu'une variable locale
  - faire appel à un constructeur depuis un autre constructeur

Le mot-clé this pour lever l'ambiguité

- Distingue un attribut d'une variable locale (ou paramètre de méthode) qui porte le même nom

```
public class Acteur {
  // ...

  public Acteur(String nom, String prenom,
                          String nationalite){
    this.nom = nom ;
    this.prenom = prenom ;
    this.nationalite = nationalite ;
  }
}
```

Le mot-clé this et les constructeurs

```
public class Acteur {
  // ...

  public Acteur(String nom, String prenom,
                          String nationalite){
    this.nom = nom ;
    this.prenom = prenom ;
    this.nationalite = nationalite ;
  }

  public Acteur(String nom, String prenom){
    this(nom,prenom,"") ;
  }
}
```

Instanciation

- Instanciation : création d'un objet appartenant à une classe
- Appel au constructeur avec le mot-clé **new**
- Exemple : [1]

```
Acteur deNiro = new Acteur("De Niro", "Robert",
                                        "Americain");
```

- Création d'une zone de la mémoire dédiée aux données de l'objet créé, retourne une référence vers cet objet

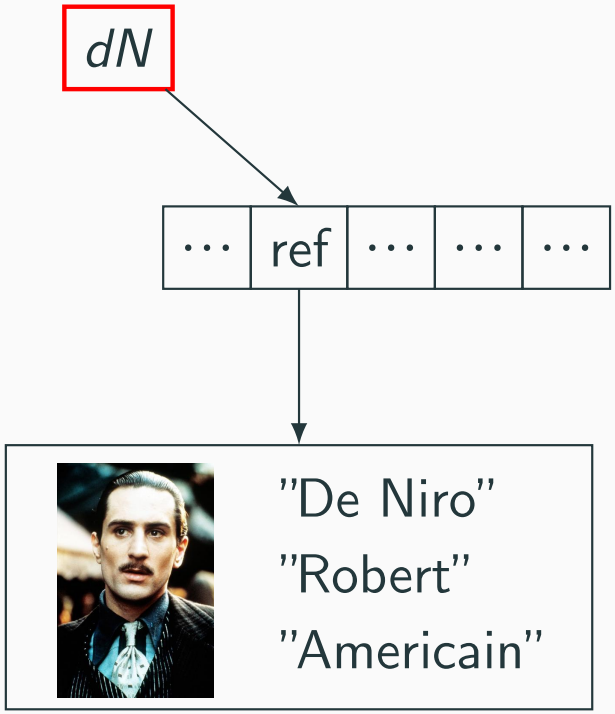
1. Je triche un peu, Robert De Niro a également la nationalité italienne depuis 2006...

Affectation et passage de paramètres

- Rappel : en Java, l'affectation se fait en recopie et le passage de paramètres se fait par valeur... pour les types simples
- En Java, on ne manipule jamais directement les objets, mais seulement des références vers ces objets ($\simeq$ des pointeurs)
- Lors d'une affectation ou d'un passage de paramètre, c'est donc la référence qui est copiée, pas l'objet lui même

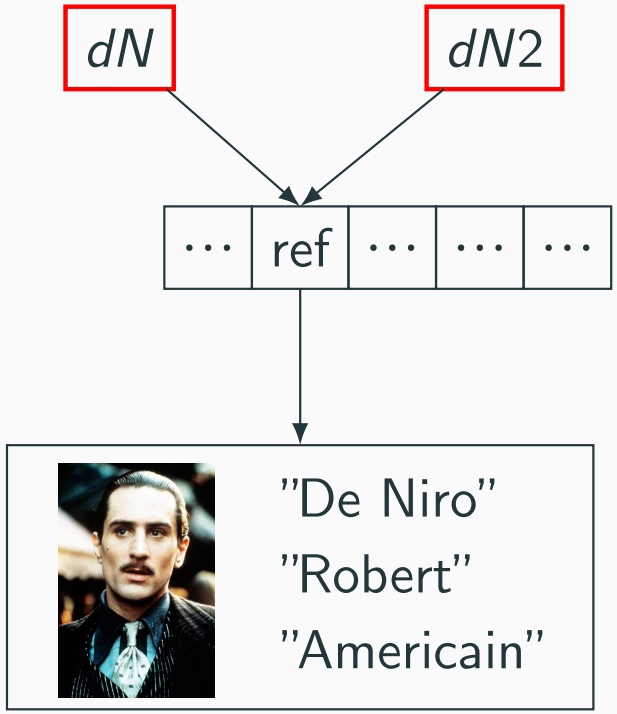
Exemple d'affectation d'objet

```
Acteur dN = new Acteur("De Niro",
        "Robert", "Americain");
```



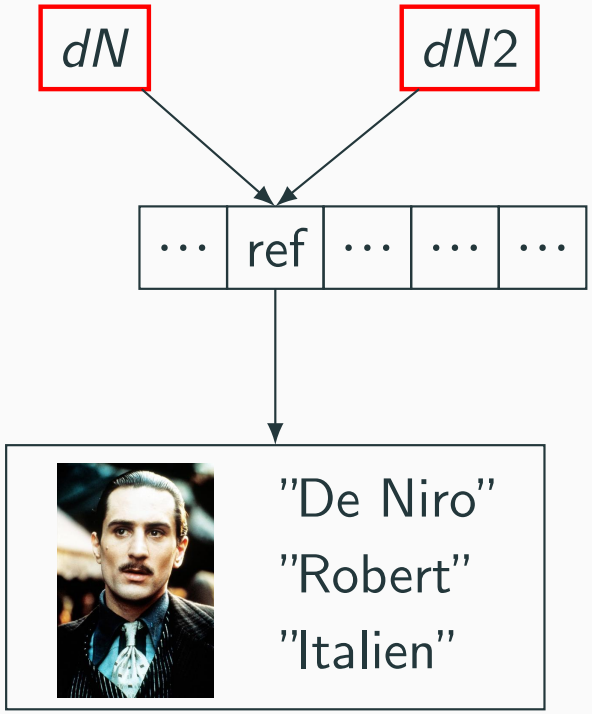
Exemple d'affectation d'objet

```
Acteur dN = new Acteur("De Niro",
        "Robert", "Americain");
Acteur dN2 = dN ;
```



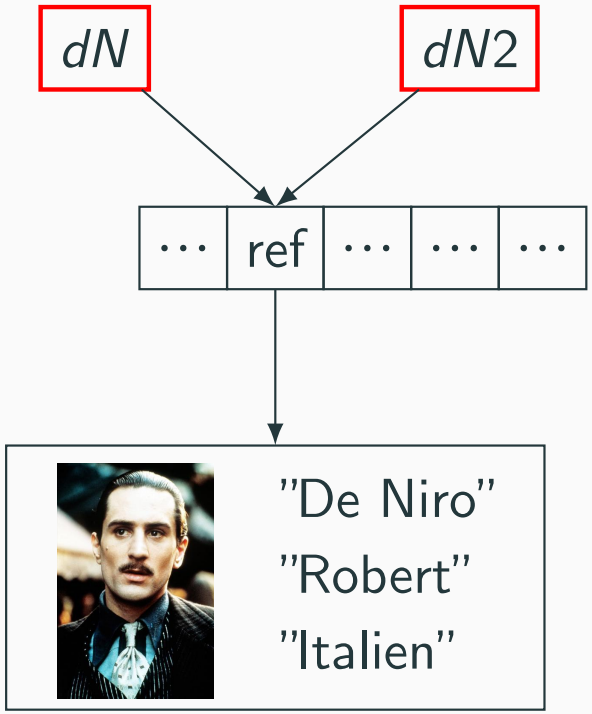
Exemple d'affectation d'objet

```
Acteur dN = new Acteur("De Niro",
        "Robert", "Americain");
Acteur dN2 = dN ;
dN2.setNationalite("Italien") ;
```



Exemple d'affectation d'objet

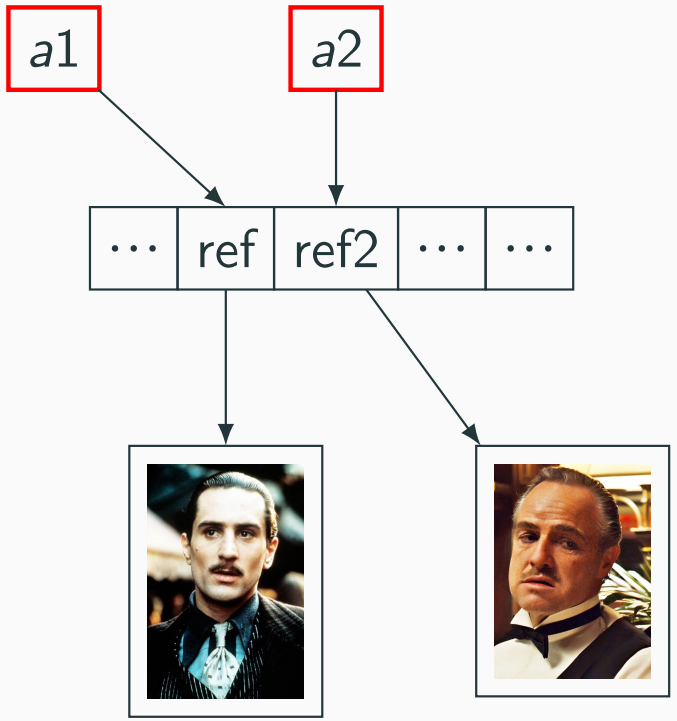
```
Acteur dN = new Acteur("De Niro",
        "Robert", "Americain");
Acteur dN2 = dN ;
dN2.setNationalite("Italien") ;
System.out.println(
            dN.getNationalite()) ;
```



Affichage : `"Italien"`

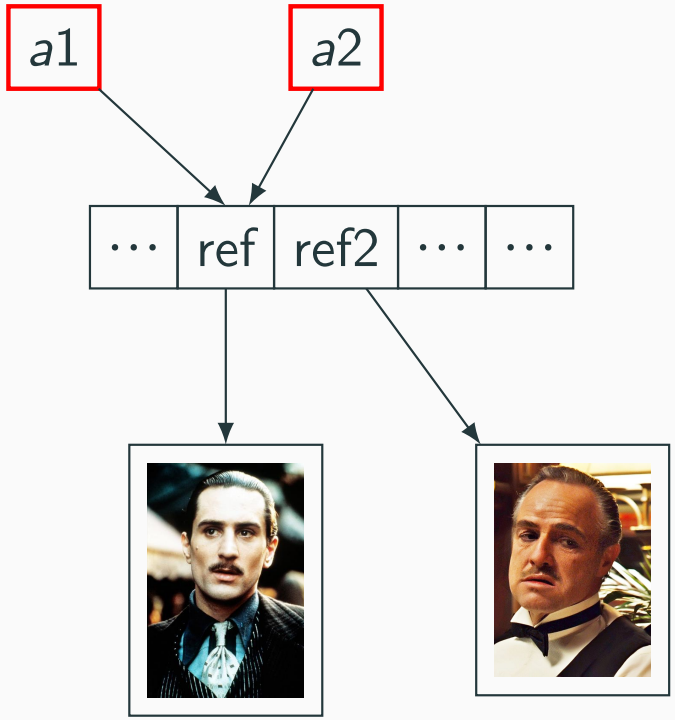
Exemple d'affectation d'objet

```
Acteur a1 = new Acteur("De Niro",
        "Robert", "Americain");
Acteur a2 = new Acteur("Brando",
        "Marlon", "Americain");
```



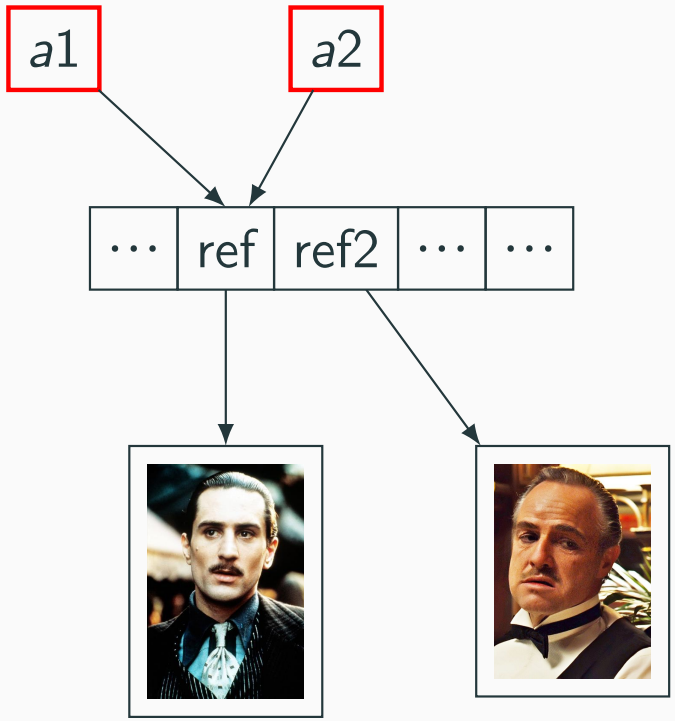
Exemple d'affectation d'objet

```
Acteur a1 = new Acteur("De Niro",
        "Robert", "Americain");
Acteur a2 = new Acteur("Brando",
        "Marlon", "Americain");
a2 = a1 ;
```



Exemple d'affectation d'objet

```
Acteur a1 = new Acteur("De Niro",
        "Robert", "Americain");
Acteur a2 = new Acteur("Brando",
        "Marlon", "Americain");
a2 = a1 ;
```

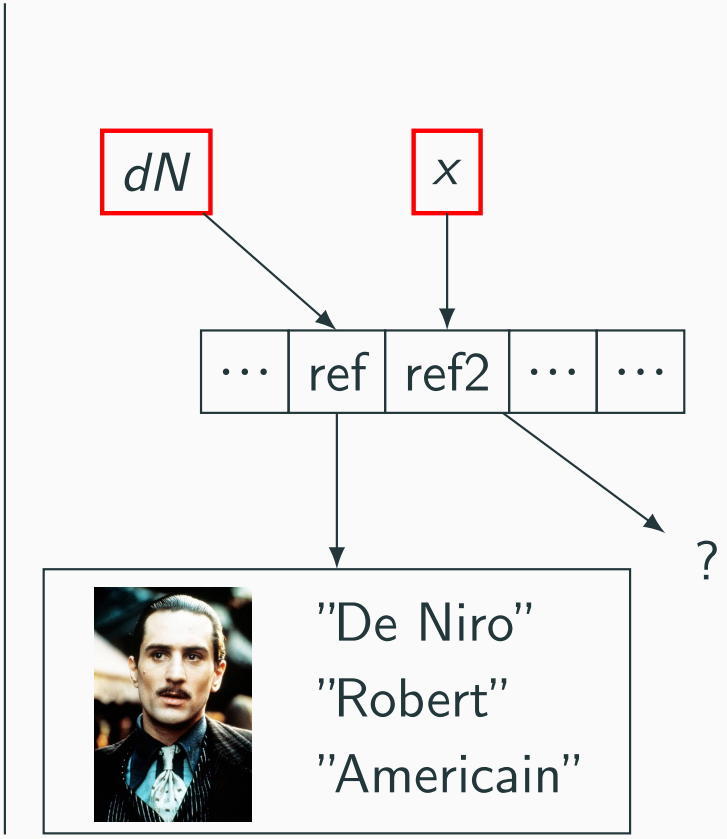


- Le *garbage collector* (GC, ou ramasse-miettes en français) est le sous-système de la JVM en charge de la mémoire
- Il libère l'espace correspondant à Marlon Brando dès que plus aucune référence à cet objet n'est faite dans le programme

Null

- **null** signifie qu'on ne fait référence à aucun objet

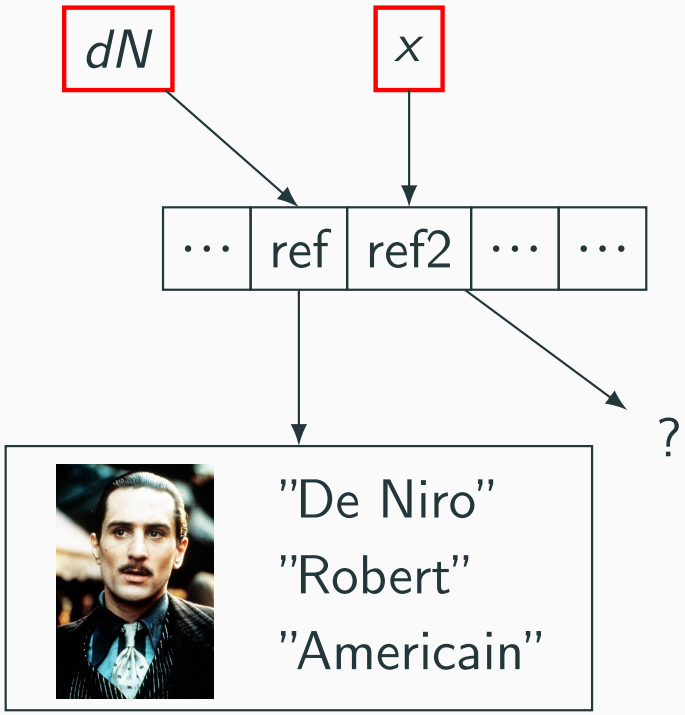
```
Acteur dN = new Acteur("De Niro",
        "Robert", "Americain");
Acteur x = null ;
```



Null

- **null** signifie qu'on ne fait référence à aucun objet

```
Acteur dN = new Acteur("De Niro",
        "Robert", "Americain");
Acteur x = null ;
System.out.println(dN.getNom());
```

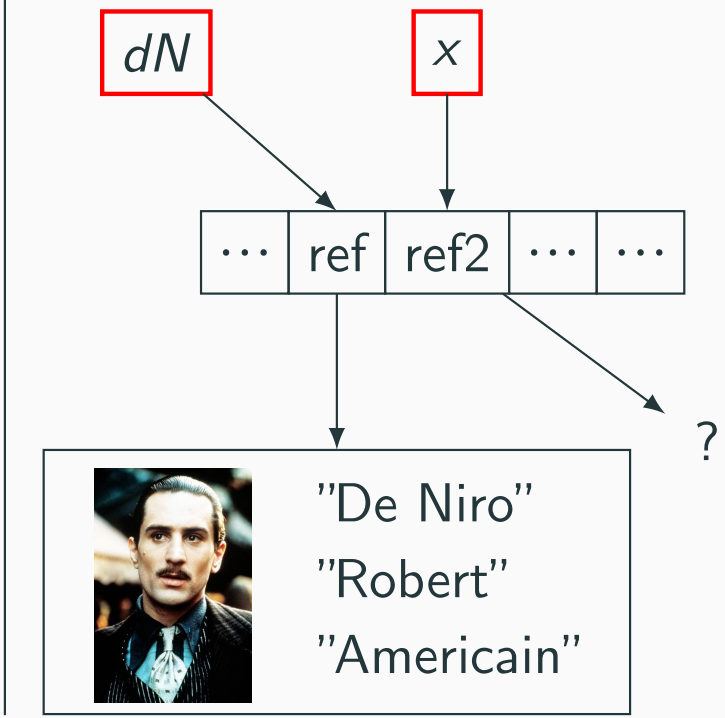


- Affichage : `De Niro`

Null

- **null** signifie qu'on ne fait référence à aucun objet

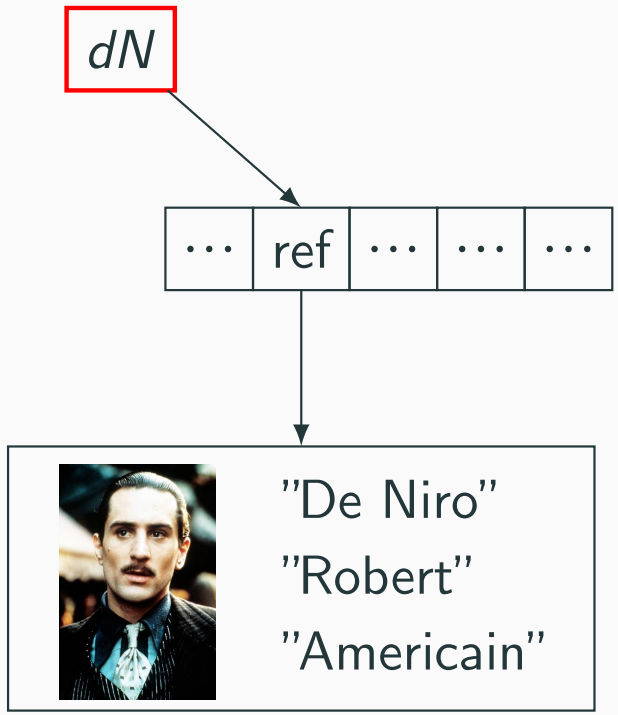
```
Acteur dN = new Acteur("De Niro",
        "Robert", "Americain");
Acteur x = null ;
System.out.println(dN.getNom());
System.out.println(x.getNom());
```



- Si on essaye d'accéder à un membre d'une référence **null**, on optient une erreur : java.lang.NullPointerException

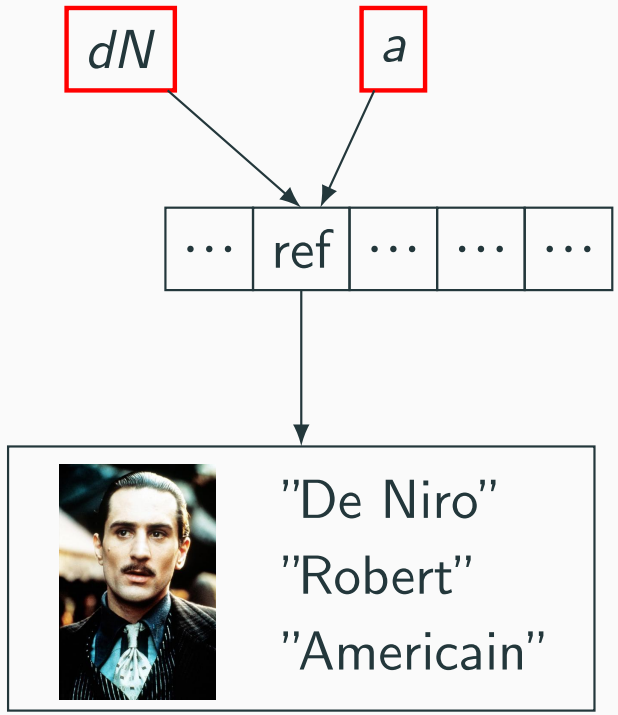
Passage de paramètres pour les objets

```
public class Test {
   private static void f(Acteur a){
    a.setNationalite("Italien") ;
   }
   public static void main(
                        String[] args){
     Acteur dN = new Acteur(
                "De Niro", "Robert",
                          "Americain");
     f(dN);
     System.out.println(
                dN.getNationalite());
   }
}
```



Passage de paramètres pour les objets

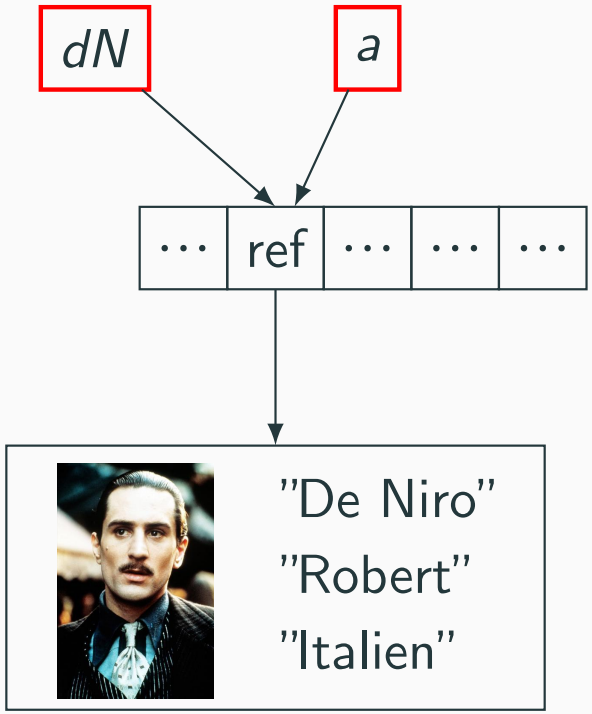
```
public class Test {
   private static void f(Acteur a){
    a.setNationalite("Italien") ;
   }
   public static void main(
                        String[] args){
     Acteur dN = new Acteur(
                 "De Niro", "Robert",
                          "Americain");
     f(dN);
     System.out.println(
                 dN.getNationalite());
   }
}
```



- `f` : variable locale a qui correspond à la même référence

Passage de paramètres pour les objets

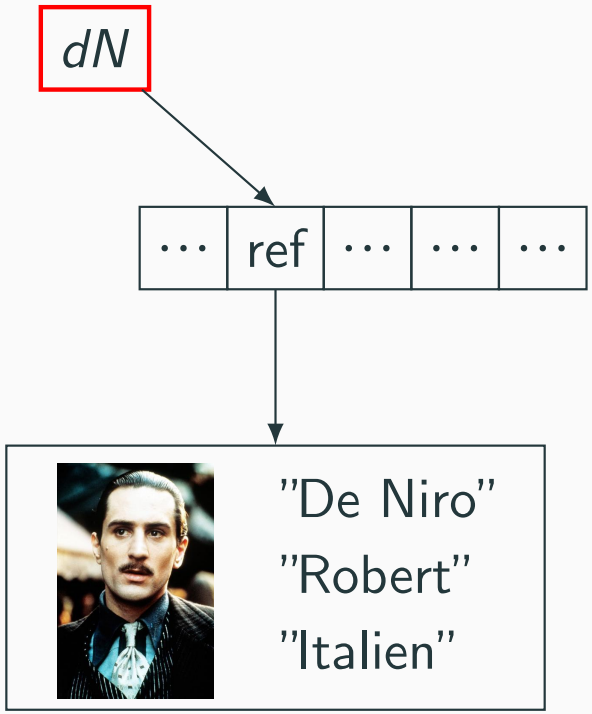
```
public class Test {
   private static void f(Acteur a){
    a.setNationalite("Italien") ;
   }
   public static void main(
                        String[] args){
     Acteur dN = new Acteur(
                 "De Niro", "Robert",
                          "Americain");
     f(dN);
     System.out.println(
                 dN.getNationalite());
   }
}
```



- `f` : modification de la nationalité de De Niro

Passage de paramètres pour les objets

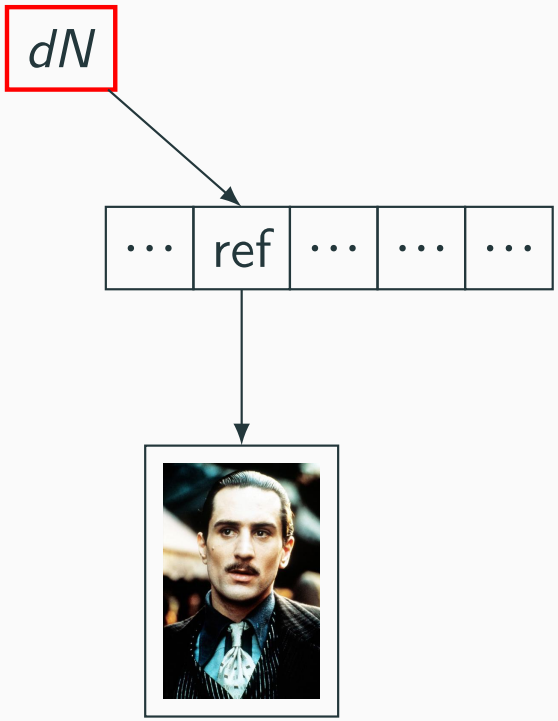
```
public class Test {
   private static void f(Acteur a){
    a.setNationalite("Italien") ;
   }
   public static void main(
                        String[] args){
     Acteur dN = new Acteur(
                 "De Niro", "Robert",
                          "Americain");
     f(dN);
     System.out.println(
                 dN.getNationalite());
   }
}
```



- Affichage : `"Italien"`

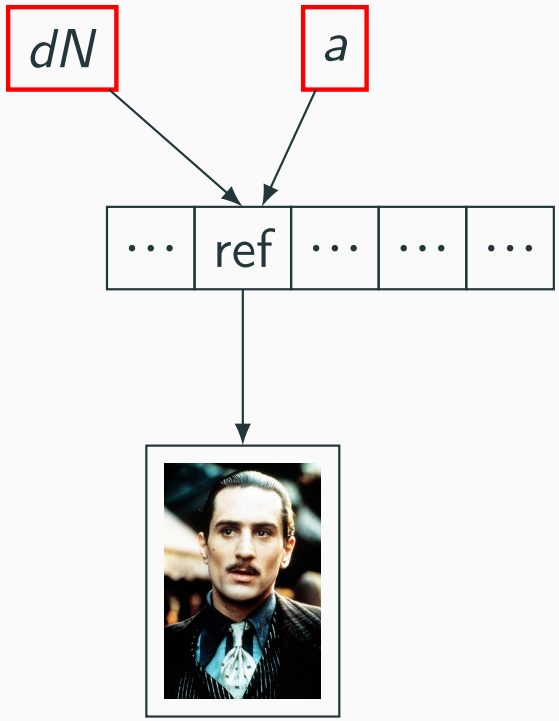
Passage de paramètres pour les objets

```
public class Test {
  private static void f(Acteur a){
    a = new Acteur("Brando","Marlon",
                          "Americain");
  }
  public static void main(
                      String[] args){
    Acteur dN = new Acteur(
              "De Niro", "Robert",
                          "Americain");
    f(dN);
    System.out.println(dN.getNom());
  }
}
```



Passage de paramètres pour les objets

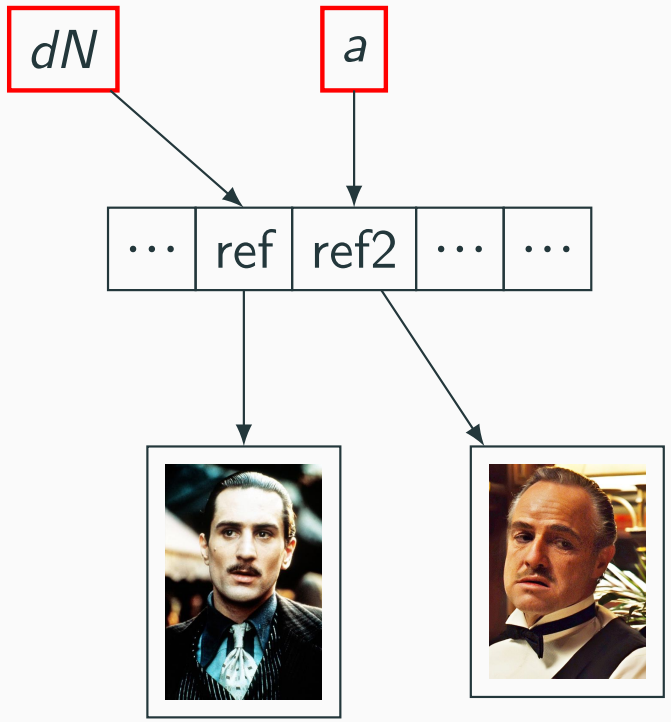
```
public class Test {
  private static void f(Acteur a){
    a = new Acteur("Brando","Marlon",
                          "Americain");
  }
  public static void main(
                      String[] args){
    Acteur dN = new Acteur(
              "De Niro", "Robert",
                          "Americain");
    f(dN);
    System.out.println(dN.getNom());
  }
}
```



- `f` : variable locale a qui correspond à la même référence

Passage de paramètres pour les objets

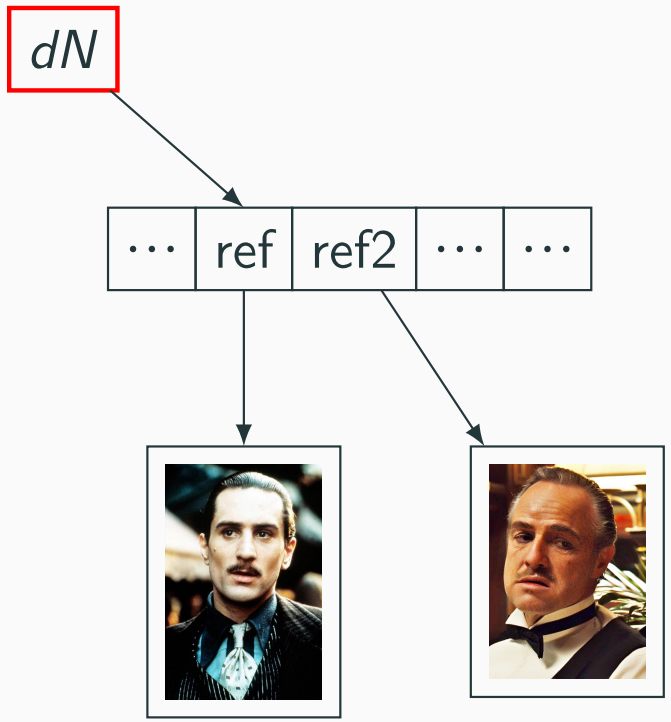
```
public class Test {
  private static void f(Acteur a){
    a = new Acteur("Brando","Marlon",
                          "Americain");
  }
  public static void main(
                       String[] args){
    Acteur dN = new Acteur(
              "De Niro", "Robert",
                          "Americain");
    f(dN);
    System.out.println(dN.getNom());
  }
}
```



- `f` : création nouvel objet et nouvelle référence pour Marlon Brando

Passage de paramètres pour les objets

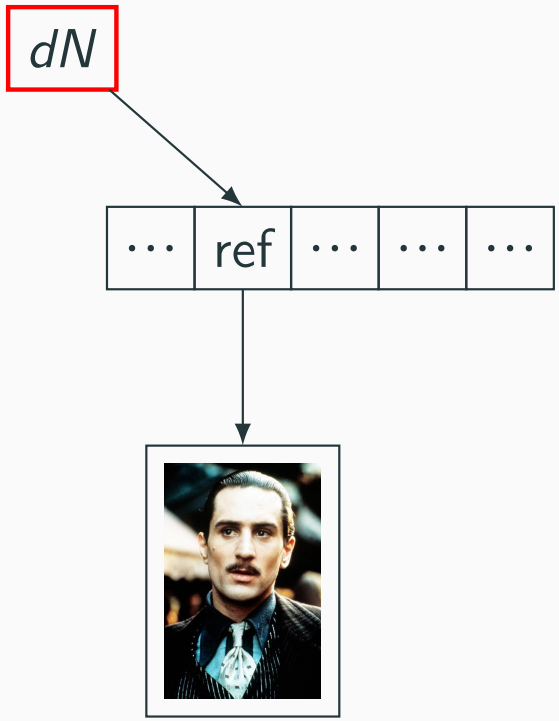
```
public class Test {
  private static void f(Acteur a){
    a = new Acteur("Brando","Marlon",
                   "Americain");
  }
  public static void main(
                     String[] args){
    Acteur dN = new Acteur(
              "De Niro", "Robert",
                   "Americain");
    f(dN);
    System.out.println(dN.getNom());
  }
}
```



- Sortie de `f` : suppression de la variable locale a

Passage de paramètres pour les objets

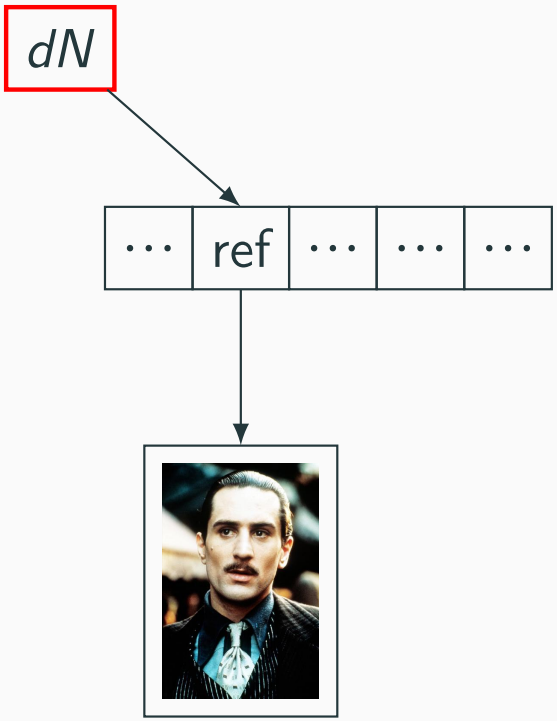
```
public class Test {
  private static void f(Acteur a){
    a = new Acteur("Brando","Marlon",
                          "Americain");
  }
  public static void main(
                      String[] args){
    Acteur dN = new Acteur(
              "De Niro", "Robert",
                          "Americain");
    f(dN);
    System.out.println(dN.getNom());
  }
}
```



- Sortie de `f` : le GC libère l'espace occupé par Marlon Brando

Passage de paramètres pour les objets

```
public class Test {
  private static void f(Acteur a){
    a = new Acteur("Brando","Marlon",
                          "Americain");
  }
  public static void main(
                      String[] args){
    Acteur dN = new Acteur(
              "De Niro", "Robert",
                          "Americain");
    f(dN);
    System.out.println(dN.getNom());
  }
}
```



- Affichage : `"De Niro"`

Complément d'information sur les expression « new »

- L'opérateur **new** retourne un objet qui peut être directement utilisé dans une expression plus complexe
- Exemple : appel d'une méthode sur le résultat du **new**

```
System.out.println(new Acteur("De␣Niro","Robert",
                              "Americain").getNom());
```

- Exemple 2 : utilisation comme paramètre d'une méthode

```
Film film = new Film();
film.ajoutActeur(new Acteur("De␣Niro","Robert",
                                      "Americain"));
```

Le Garbage Collector

- Le GC détecte les objets inutilisables et les supprime de la mémoire
- Il fonctionne en tâche de fond et ne supprime pas forcément les objets *immédiatement* après qu'ils soient devenus inutilisables
- Objet inutilisable : objet qui n'est associé à aucune référence
- Le programmeur n'a pas à gérer manuellement la mémoire (pas de `free()` comment en `C`)
- En cas de besoin, on peut activer le GC manuellement avec « System.gc() ; », même si ce n'est pas courant

Tableaux, objets et classes (1/2)

- Les tableaux sont des objets : pour chaque type d'éléments de tableaux, une classe correspondante est créée
- Exemple :

```
public class Test {
   public static void main(String[] args){
      String[] tab = {"Hello ", "World!"};
      System.out.println(tab[0].getClass());
      System.out.println(tab.getClass());
   }
}
```

- Affichage :

```
class java.lang.String
class [Ljava.lang.String;
```

Tableaux, objets et classes (2/2)

- Comme les tableaux sont des objets, les affectations et passages de paramètres se font par référence
- La longueur du tableau correspond donc à un attribut public et constant : tab.length
  - Nous verrons plus tard la définition d'attributs constants
- Les éléments d'un tableaux peuvent être de n'importe quel type, y compris des tableaux !
- Une matrice d'entiers de dimension 3×3

```
int[][] t = new int[3][];
t[0] = {1,2,3};
t[1] = {4,5,6};
t[2] = {7,8,9};
```

Principe d'encapsulation

- *Principe d'encapsulation* : les données (→ les attributs) ne doivent pas être accessibles et modifiables librement par n'importe qui
- Pour cela, il est recommandé que les attributs soient **private**, et ne puissent être utilisés ou modifiés que via des méthodes
    - Cela permet au développeur de s'assurer que ses classes seront utilisées d'une manière cohérente par la suite

Une classe sans encapsulation

```
public class Produit {
  public String nom ;
  public double prix ;
  public Produit(String nom, double prix){
    this.nom = nom ;
    this.prix = prix ;
  }
}
```

Une classe sans encapsulation

```
public class Produit {
  public String nom ;
  public double prix ;
  public Produit(String nom, double prix){
    this.nom = nom ;
    this.prix = prix ;
  }
}
```

- L'utilisation de cette classe peut mener à des incohérences :

```
Produit prod = new Produit(
        "CD Greatest Hits", 16.99);
// Ailleurs dans le code...
prod.prix = -10 ;
```

16,99 €
prime
Queen Greatest Hits I, II & III - Platinum Collection
357

La même classe avec encapsulation

```
public class Produit {
  private String nom ;
  private double prix ;

  public Produit(String nom, double prix){
    this.nom = nom ;
    this.prix = prix ;
  }
  public void setPrix(double prix){
    if(prix>=0){
      this.prix = prix ;
    }
  }
}
```

- Impossible de mettre un prix incohérent maintenant !
- Remarque : si nécessaire, on peut faire le même genre de contrôle de cohérence dans le constructeur

Les buts de l'encapsulation

- On assure dans les méthodes le respect de certaines contraintes (voir l'exemple du prix d'un produit)
- Les éventuelles modifications de la classe d'une version à l'autre sont transparentes pour les utilisateurs, qui ne voient que son interface (= les méthodes publiques) sans voir les détails d'implémentation

Les buts de l'encapsulation

- On assure dans les méthodes le respect de certaines contraintes (voir l'exemple du prix d'un produit)
- Les éventuelles modifications de la classe d'une version à l'autre sont transparentes pour les utilisateurs, qui ne voient que son interface (= les méthodes publiques) sans voir les détails d'implémentation

→ autant que possible, un attribut doit être **private**, et on y accède ou on le modifie via des méthodes
Seules exceptions : (certaines) constantes, et l'utilisation de **protected** pour l'héritage

Les constantes (1/2)

- Le modificateur **final** permet de définir des constantes : un attribut **final** ne peut plus être modifié après son initialisation
- Pour un attribut de type primitif, la valeur est fixée. Exemple : **final** **int** N = 5 ; → `N` vaudra toujours 5, il est impossible par la suite de lui affecter une autre valeur
- Pour un attribut de type objet, la référence est fixée, mais l'état de l'objet peut être modifié via des méthodes. Exemple :
  final Acteur DE_NIRO = **new** Acteur(...);
  et ailleurs dans le code : DE_NIRO.setNationalite(" Italien ") ;
- Même fonctionnement pour une variable locale ; on parlera plus tard de l'utilisation de **final** pour les méthodes et classes
- Convention : noms des constantes en majuscules, mots séparés par un underscore _

Les constantes (2/2)

- Par défaut, une constante est relative à une instance de la classe, elle peut (par exemple) être initialisée dans le constructeur

```
public class Personne {
  private final String NUM_SECU ;
  public Personne(String numSecu){
    NUM_SECU = numSecu ;
  }
}
```

Les constantes (2/2)

- Par défaut, une constante est relative à une instance de la classe, elle peut (par exemple) être initialisée dans le constructeur

```
public class Personne {
  private final String NUM_SECU ;
  public Personne(String numSecu){
    NUM_SECU = numSecu ;
  }
}
```

- On peut également définir une constante « absolue » :

```
public class Math {
  public static final double PI = 3.14 ;
}
```

→ `PI` est accessible depuis n'importe quel endroit du code (**public**), sans devoir créer d'instance de `Math` (**static**), et sans possibilité de modifier sa valeur (**final**)