

Le langage Java

- l'objet-

Construire des types de données

- Programme doivent manipuler des entités complexes
 - Ex: personnes, comptes en banque, pages web,...
- Les types primitifs **int**, **double**, **boolean**, etc. ne suffisent pas
 - Chacun ne capture qu'une partie d'une entité à représenté
 - Ex: taille en cm d'une personne peut être représentée par un **int**
- => Une **classe** Java nous permet de représenter une entité
 - Une classe a un **nom de classe** et des **attributs**
 - Attributs : ensembles de caractéristiques de l'entité
 - Mot clé **class**

Anatomie d'une classe simple

```
class Nom_de_classe {  
    type_attribut1 attribut1;  
    type_attribut2 attribut2;  
    ...  
}
```

- Déclaration

```
Nom_de_classe nom = new Nom_de_classe();
```

- Accès aux attributs

```
nom.attribut1 = ... ;  
... = nom.attribut2 ;
```

Exemples

...au tableau...

Constructeur

- Mécanisme d'**initialisation paramétrée** d'une classe
- Code inséré dans la classe, sous la forme :

```
class Nom_classe {  
    ...  
    public Nom_classe(...paramètres...) {  
        // code initialisation  
    }  
    ...  
}
```

Ex:

...au tableau...

- On peut avoir plusieurs constructeurs avec des paramètres différents
 - Il y a toujours un **constructeur par défaut**, sans paramètres (fourni par Java)
 - Appel : new `Nom_classe()`;

Que fait exactement **new** ?

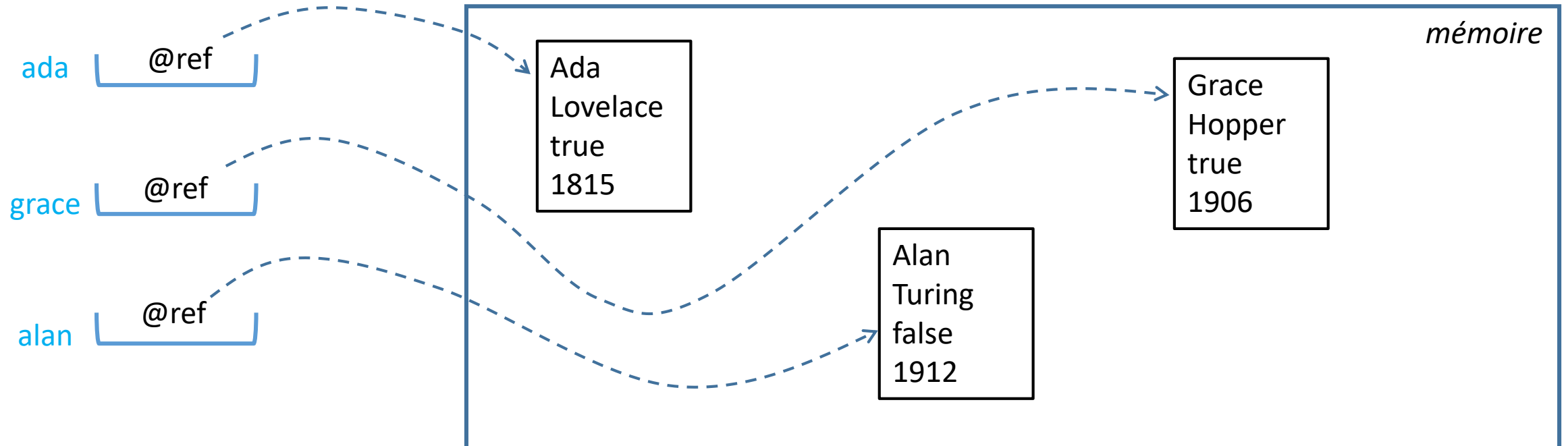
- Construit une **instance** de la classe en mémoire
 - Réserve une place de la bonne taille
- Exécute le constructeur pour initialiser l'instance
- Renvoie une **référence** vers l'instance construite
 - C'est-à-dire son adresse dans la mémoire

Illustration du fonctionnement de **new**

```
Humain ada = new Humain("Ada", "Lovelace", true, 1815);
```

```
Humain grace = new Humain("Grace", "Hopper", true, 1906);
```

```
Humain alan = new Humain("Alan", "Turing", false, 1912);
```



Programmation Orientée Objet : généralités

- Classes permettent de stocker des données relatives à des « entités » réelles ou virtuelles
- On leur rajoute des **méthodes** d'accès et de traitement de ces données
 - Ex: Classe Rectangle
 - Attributs : coordonnées coin supérieur gauche, largeur, longueur, couleur...
 - Méthodes : dessin du rectangle, calcul de périmètre, calcul d'aire,...
- D'où le nom de programmation par « objet »
 - Les attributs et les méthodes permettent de modéliser
 - Des caractéristiques de l'objet
 - Ainsi que son comportement

Pensons objet !

Quels pourraient être les attributs et les méthodes des classes suivantes ?

- Etudiant
- Date
- ApplicationSmartphone

Objets et génie logiciel

- Idée : faire des objets (=classes en java) **réutilisables**
- Concepts :
 - **Encapsulation**
 - L'utilisation d'une classe ne change pas même si les détails d'implémentation changent
 - **Héritage**
 - Étendre des classes existantes

Encapsulation

- Séparer le « client » (utilisateur de la classe) de l'implémentation de la classe
 - Ex: Voiture : démarrer, accélérer, freiner
Pas besoin de savoir comment le moteur marche...
 - Anti-ex: ZIP codes US, an 2000, IPv4/IPv6
- Principes
 - Cacher l'information
 - Ask, don't touch

Cacher l'information

Modifieurs pour les attributs et les méthodes de classe

- **private** : accès uniquement à l'intérieur de la classe
- **public** : tout le monde peut y accéder

Exemple public / private

```
class Prix {
```

```
    private double prixHT, prixTTC;  
    private double tauxTVA = 1.20;  
    private void calculeTTC() {  
        prixTTC = tauxTVA*prixHT;  
    }
```

Invisible de l'extérieur

```
    public Prix(double prixHT) {  
        this.prixHT = prixHT;  
        calculeTTC();  
    }  
    public double getPrixTTC() {  
        return prixTTC;  
    }
```

Visible de l'extérieur

```
}
```

Ask, don't touch

- Interdire l'accès à **tous les attributs de classe**
 - Tous les attributs sont private
- Modification assurée par des méthodes publiques getXXX et setXXX
 - Ex:

```
public void setPrixHT(double prixHT) {  
    if (prixHT >= 0.0)  
        this.prixHT = prixHT;  
}  
public double getPrixHT() {  
    return prixHT;  
}
```

Pensons encapsulation !

Pour les classes suivantes, proposez des méthodes publiques et privées, et des attributs privés:

- Etudiant
- CompteBancaire
- NombreComplexe

Héritage

- Héritage : définir une classe qui étend/spécialise une classe
 - Ex: Etudiant hérite de Humain
 - class **Etudiant** **extends** **Humain**
- La classe « dérivée » reçoit automatiquement de sa classe « parente »
 - Tous ses attributs
 - Toutes ses méthodes
- Elle peut ajouter ses propres attributs et méthodes

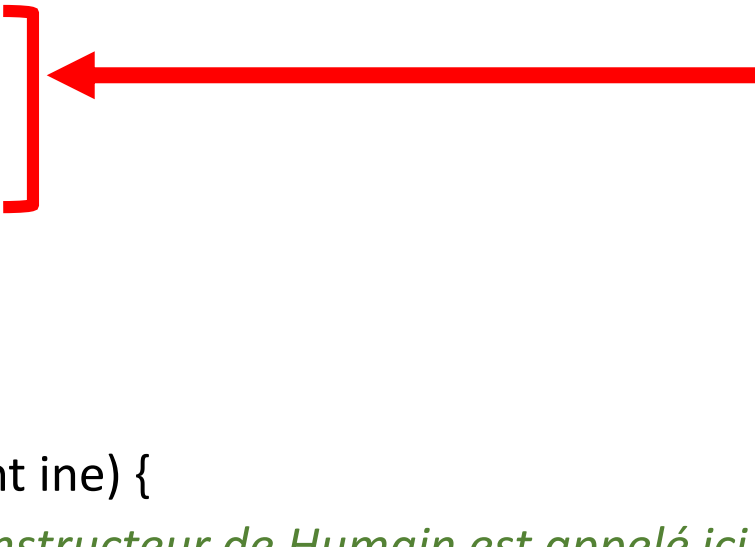
Exemple

- Humain et Etudiant : ...*au tableau*...

Constructeurs de classes filles

- On peut réutiliser le constructeur de la classe parente avec le mot clé **super(...paramètres...)**
- Exemple:

```
class Humain {  
    private String nom;  
    public Humain(String nom) {  
        this.nom = nom;  
    }  
}  
  
class Etudiant extends Humain {  
    private int ine;  
    public Etudiant(String nom, int ine) {  
        super(nom); // le constructeur de Humain est appelé ici  
        this.ine = ine;  
    }  
}
```



Surcharge

- On peut redéfinir des méthodes de la classe parente
- Mot clé **@Override** à écrire avant la méthode dans la classe fille
- Ex:

```
class Humain {  
    public String toString() { ... }  
}  
class Etudiant extends Humain {  
    @Override  
    public String toString() { ...nouveau code... }  
}
```

- `super.truc()` dans une méthode de la classe fille appelle la méthode *truc* de la classe parente

Classes et méthodes abstraites

- Une méthode est **abstraite** si elle n'a pas d'implémentation
 - Elle sera implémentée dans les classes filles
 - Cela permet de savoir à l'avance comment utiliser les classes filles
 - Ex: *...objets graphiques...au tableau...*
- En java: déclaration du type
abstract type_retour nom_methode(...parametres...);
- Une classe ayant des méthodes abstraites doit être déclarée abstraite
 - `abstract class Nom_classe { ... }`
- Une classe abstraite ne peut pas être instanciée (pas de new)
- Mais elle peut être avoir des sous-classes

Pensons héritage !

Imaginez les hiérarchies suivantes :

- Autres classes dérivées de Humain ?
- Classes dérivées de Etudiant ?
- Hiérarchie de formes ?

Polymorphisme (d'héritage)

- Technique pour abstraire les détails des spécialisations (classes filles) dans une famille d'objets
 - Une classe parente (parfois abstraite) définit les méthodes utilisables
 - Les classes filles surchargent ces méthodes en fonction de leurs spécificités

- Ex:

```
class Forme { ... void dessiner() {...} }
```

```
class Rectangle extends Forme { ... @Override void dessiner() {...} }
```

On peut écrire:

```
Forme f = new Rectangle();  
f.dessiner();
```

Attention:

Utiliser une méthode spécifique à Rectangle à partir de f ?
...au tableau...

Autre modificateurs importants

- **final** pour une classe : ne peut pas être étendue par héritage
- **static** pour un attribut : la valeur de l'attribut est partagée par toutes les instances de la classes
- **static** pour une méthode :
 - La méthode n'a pas besoin d'une instance pour être appelée (ex: Arrays.toString)
 - Elle ne peut pas opérer sur des attributs non statiques de la classe