IPRO

PROGRAMMATION ORIENTÉE OBJET & JAVA

Guillaume Bouyer

ENSIIE

www.ensiie.fr/~bouyer

DÉROULEMENT DU COURS

OBJECTIFS

Connaître les concepts clés et maitriser le vocabulaire de la programmation orientée objet

Savoir mener une conception logicielle avec méthode

Appliquer ces connaissances dans un petit projet de gestion dans le langage Java

PLAN DU COURS N°1

INTERVENANTS

Guillaume Bouyer

- ENSIIE, IBISC (Univ. Evry, Univ. Paris-Saclay)
- Enseignant-Chercheur en Réalité Virtuelle

Vitera Y

- Ethereal World (C-19)
- Développeur, game designer

PLANNING PRÉVISIONNEL

Séance	Date	Description
1	29 mars	Cours POO/Java
2	4 avril	TP 1
3	12 avril	TP 1
4	18 avril	Cours Exceptions + TP 2
	vacances	
5	3 mai	Cours Generics/Collections + Interro + TP 3
6	10 mai	Cours UML/Conception + TP 3 + Sujet projet
7	16 mai	Cours Design patterns + Projet
8	24 mai	Projet
	pont	
9	7 juin	Projet + Interro
10	14 juin	Projet
11	21 juin	Projet
12	27 juin	Soutenances

NOTATION

2 Interros

Projet

NOTIONS DE DÉVELOPPEMENT LOGICIEL

CRITÈRES DE QUALITÉ D'UN LOGICIEL

Un programme doit réunir un ensemble de qualités vis-à-vis de ses utilisateurs et de ses développeurs

- Correction, validité, vérifiabilité, efficacité
  - résoud effectivement le problème posé (par rapport à une spécification)
- Robustesse, fiabilité
  - devrait fonctionner même dans des conditions anormales
- Ergonomie, lisibilité, facilité d'utilisation et d'apprentissage
- Intégrité, sécurité
  - protection des données et du code interne

CRITÈRES DE QUALITÉ D'UN LOGICIEL

- Cycle de développement
  - Spécifications, implémentation, équipe, tests…
- Maintenance
- Réutilisabilité, extensibilité
  - Amortissement
- Portabilité
  - pourrait fonctionner sur plusieurs systèmes
- Compatibilité, interopérabilité
  - pour combiner les logiciels entre eux
  - pour durer dans le temps

=> *Compromis*

EX : COÛT DE MAINTENANCE

1. corrections de bugs
2. modifications des exigences des utilisateurs
3. modification de formats de données
4. modification des bibliothèques que l'on utilise

MOYENS POUR LA QUALITÉ LOGICIELLE

Il existe des concepts, des méthodes et des outils pour aider à atteindre ces critères de qualité

La programmation orientée objet rassemble plusieurs de ces moyens

- Concepts d'**abstraction, encapsulation, modularité** ou **généricité** => Réutilisabilité, compatibilité, extensibilité
- Phases de **spécification** et de conception bien identifiées, voire formelles, **tests** => Correction
- Techniques de programmation comme les **exceptions** => Robustesse

ABSTRACTION

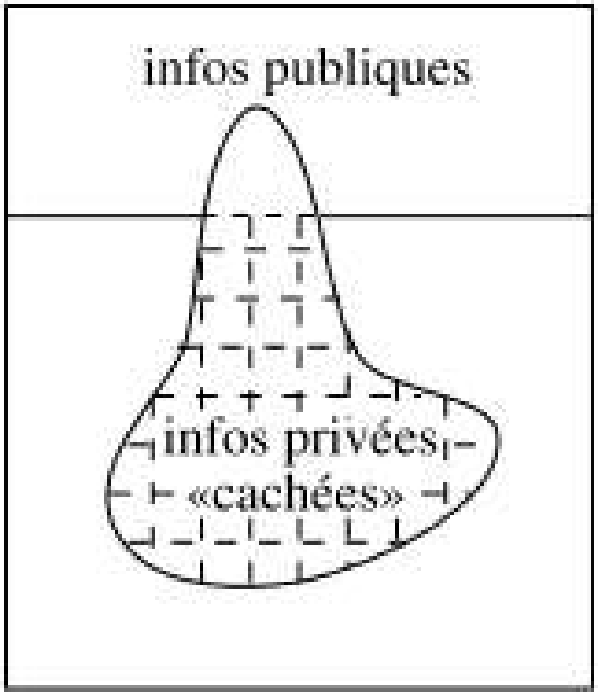
> *“Une abstraction désigne les caractéristiques essentielles d'un objet qui le distingue de tous les autres types d'objets et fournit ainsi des limites conceptuelles nettement définies par rapport à la perspective du spectateur.” (Grady Booch)*

Ces caractéristiques sont relatives au contexte dans lequel l'entité est utilisée

ENCAPSULATION

Regrouper au sein d'une même entité des données et les méthodes qui les manipulent

- Cacher les détails internes de l'entité de l'extérieur
- Les interactions sont effectuées via les méthodes fournies



MODULARITÉ

Décomposer un problème (programme) afin de réduire sa complexité globale, en modules connectés entre eux mais aussi indépendants que possible

- Module = ensemble de ressources (données, types, opérations) liées sémantiquement
- Connection par des interfaces bien définies = mode d'emploi du module, type abstrait... (partie *publique*)
- Corps du module = implémentation des ressources (partie *privée*)

MODULARITÉ

Permet entre autres de :

- programmer de manière plus sûre (utiliser uniquement les fonctions disponibles)
- limiter les impacts d'une modification
- documenter, travailler en équipe plus facilement (combiné à un système de gestion de versions)

LE LANGAGE JAVA

GÉNÉRALITÉS

- Un langage objet parmi d'autres (C++, C#, Python...)
- 1995 : Présentation (Gosling & Naughton, Sun Microsystems)
- 2010 : Rachat par Oracle
- Avril 2018 : version 10
- Utilisé par Android
- Largement présent en back-end dans de nombreuses entreprises / sites internet

PRINCIPES

- *Compile once, execute anywhere*
  - Un programme Java est compilé en **bytecode**

```
javac Programme.java
```

  - le bytecode `Programme.class` est interprétable par une **machine virtuelle** qui est installable sur toutes les plateformes

```
java Programme
```

- Langage fortement orienté objet
- Langage typé
- Syntaxe *C-like*
- Pas de pointeurs (mais des références)

COMPOSANTS

- JRE : Java Runtime Environment (= machine virtuelle)
  - “Covers most end-users needs. Contains everything required to run Java applications on your system.”
- JDK : Java Development Kit
  - “For Java Developers. Includes a complete JRE plus tools for developing, debugging, and monitoring Java applications.”

DISTRIBUTIONS

- Java SE : Standard Edition
  - “lets you develop and deploy Java applications on desktops and servers. Java offers the rich user interface, performance, versatility, portability, and security that today's applications require.”
- Java EE : Enterprise Edition
  - “the standard in community-driven enterprise software (...). Each release integrates new features that align with industry needs, improves application portability, and increases developer productivity.”
- Java ME, Card, TV
  - systèmes embarqués
- ...

INSTALLATION POUR IPRO

Java SE 8.172

Eclipse ou autre IDE

LES OBJETS

En POO, les objets sont la base de l'abstraction, l'encapsulation et la modularité.

NOTION D'OBJET

Un “objet” sert à modéliser les entités manipulées par le programme

- entités inanimées ou vivantes : *ville, véhicule, étudiant, fenêtre sur l’écran…*
- concepts : *date, réunion, planning de réservation…*

Un objet possède un état et un comportement, caractérisé par des réactions à des messages qu'on peut lui envoyer

NOTION D'OBJET

Un objet possède :

- une **identité** : unique, permet de distinguer un objet d'un autre
- des **données** (**attributs**, **variables d'instance**)
- des **méthodes** (**opérations**, services)

ATTRIBUTS

- les valeurs des attributs forment l'**état** interne de l'objet à un instant donné
  - les attributs sont nommés et typés
  - *Rq : de manière interne, on peut voir un objet comme un enregistrement avec des champs contenant des valeurs*

MÉTHODES

- fonctions ou procédures qui accèdent et/ou modifient les attributs (et d'autres variables)
- définissent le **comportement** de l'objet (ce qu'il peut faire, comment il peut le faire…) et ses réactions aux stimulations externes
  - En général, plusieurs objets de la même catégorie (ex. plusieurs fenêtres) ont le même comportement
- implémentent les algorithmes, les **traitements** associés à cet objet

NOTION D'OBJET : EXEMPLES

Compte bancaire

données : solde, client...

opérations : débiter, créditer, virer...

Fenêtre graphique

données : dimensions, position sur l'écran, présence ou non d'une barre d'ascenseur, couleur de fond...

opérations : fermer, réduire, modifier la taille, déplacer...

Patient

données : âge, taille, poids...

opérations : peser, mesurer, passer un examen...

Rq : Certains attributs font référence à d'autres objets

APPROCHE PROCÉDURALE (RAPPELS)

“Que doit faire mon programme ?”

L'approche procédurale nécessite de

- définir les structures de données
- définir l'ensemble des traitements sur ces données (analyse descendante)
  - qui retournent une valeur après leur invocation (fonction)
  - ou non (procédure)

Le programme principal enchaine les traitements sur les données

APPROCHE OBJET

“De quoi doit être composé mon programme ?”

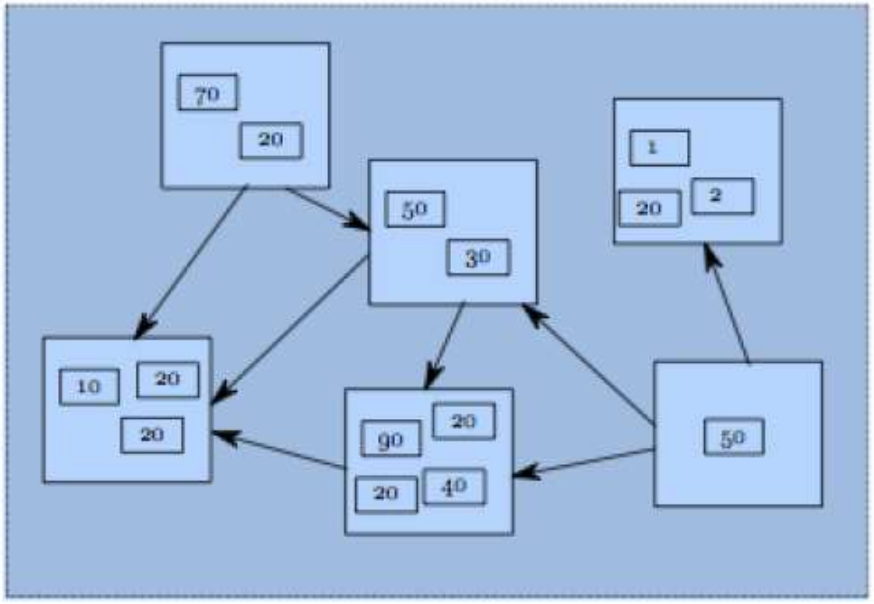
L'approche objet nécessite de

- identifier les “types” d'objets
- identifier les relations entre les objets et les communications possibles
- pour chaque objet
  - définir son interface publique (signature des méthodes)
  - définir son implémentation (attributs, corps des méthodes)

APPROCHE OBJET

Le programme principal :

- crée (instancie) les objets en mémoire avec leurs attributs
- utilise les méthodes des objets créés
- ces méthodes peuvent provoquer d'autres appels de méthodes et/ou la création d'autres objets…



CLASSES ET INSTANCES

NOTION DE CLASSE

- schéma, moule, modèle pour créer des objets qui partageront tous la même structure et le même comportement :
- abstraction pour décrire des objets similaires
  - même ensemble d'attributs
  - mêmes méthodes de traitement

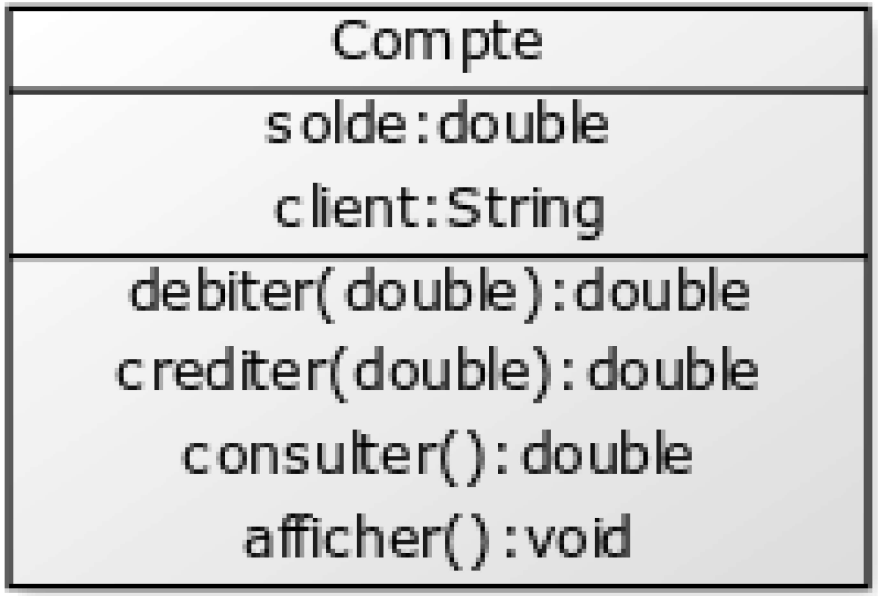
Exemples : livre, journal, employé, lecteur

CLASSE

Une classe :

- définit un **type**
- est identifiée par un nom
- est composée de **membres** qui peuvent être
  - des attributs, caractérisés par :
    - nom
    - type (si le langage est typé)
    - éventuellement valeur initiale
  - des méthodes : ensembles d'instructions de traitement

EXEMPLE DE CLASSE



Notation UML (Unified Modeling Language)

EXEMPLE DE CLASSE EN JAVA

Dans un fichier `Compte.java` (NB : nom **identique** à celui de la classe)

```
class Compte {
  //attributs (fields)
  double solde;
  Client client;

  //méthodes
  void debiter (double n) { solde = solde - n; }
  void crediter (double n) { solde = solde + n; }
  double consulter () { return solde; }
  void afficher () { System.out.println(solde); }
}
```

EXEMPLE DE CLASSE EN C++

Dans un fichier `Compte.h`

```
class Compte {
    double solde;
    Client client;

    void debiter (double n);
    void crediter (double n);
    double consulter ();
    void afficher ();
}
```

Dans un fichier `Compte.cpp`

```
#include "Compte.h"

void Compte::debiter (double n) {solde = solde - n;}
void Compte::crediter (double n) {solde = solde + n;}
double Compte::consulter () {return solde;}
void afficher () {out << solde << endl;}
```

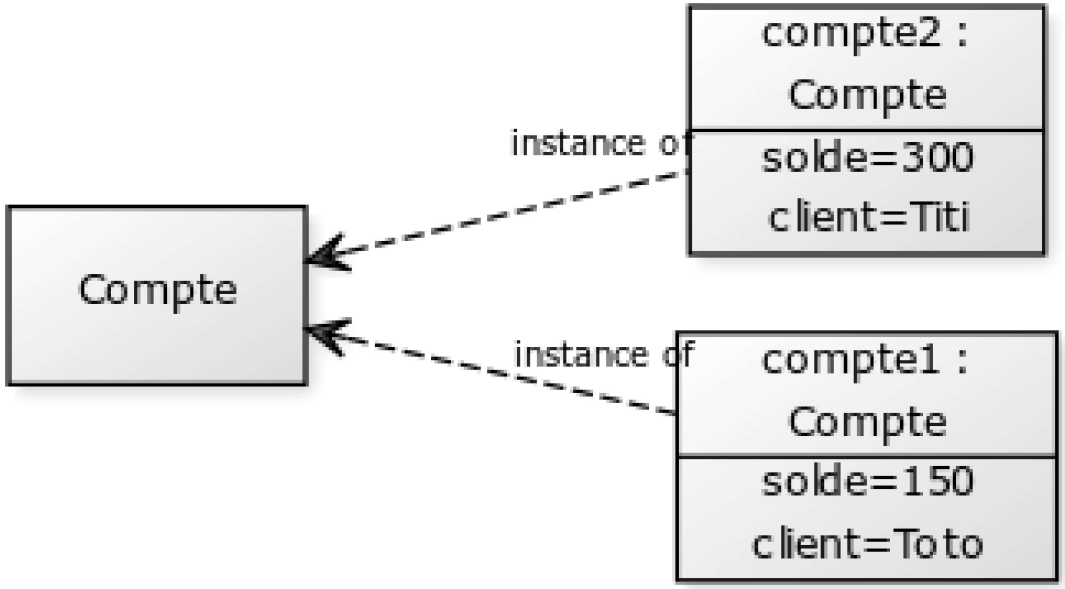
INSTANCES D'UNE CLASSE

Les objets utilisés dans les programmes sont des représentations dynamiques créées à partir du modèle donné par la classe. Chaque **instance** de la classe :

- se conforme à la description que celle-ci fournit
- possède une valeur propre pour chaque attribut (qui caractérisent l'état de l'objet)
- peut se voir appliquer toute méthode définie dans la classe
- est référencée par une variable

Une classe peut être **instanciée** plusieurs fois

INSTANCES D'UNE CLASSE : EXEMPLE



RÉFÉRENCES

Les références permettent d'identifier les objets

Une référence contient l'**adresse** d'un objet en mémoire = permet d'accéder à la structure de données correspondante aux attributs propres à l'objet

- Comme un pointeur, contient l'adresse d'une structure
- Mais contrairement aux pointeurs la seule opération autorisée sur les références est l'affectation d'une référence de même type

RÉFÉRENCES EN JAVA

```
// déclaration de 3 références dans une méthode
Banque bnp;          // identificateur de classe (= type)
Compte martin;
Compte dupond;
```

Rappel : il n'y a pas de pointeurs en Java

CRÉATION D'INSTANCE EN JAVA

Déclarer une référence ne suffit pas à créer un objet, c'est seulement un nom pour accéder à un objet

- Mot clé **new** puis appel à une méthode particulière qui porte le nom de la classe : le **constructeur**
- new renvoie une **référence** vers l'instance créée, qui peut être affectée à une variable x du type correspondant

```
// dans une méthode d'une classe
Compte martin;          // déclaration de deux références
Compte dupond;          // NB aucun objet n'est créé
martin = new Compte();  // création de 2 instances avec constructeur
dupond = new Compte();  // affectation aux références
```

Rq : l'objet créé contient les valeurs des attributs, mais pas le code des méthodes

CONSTRUCTEURS

- Méthodes écrites dans la classe qui portent le **même nom** que la classe
- Ils **ne renvoient pas de valeur** (pas de type de retour)
- Ils peuvent avoir ou non des arguments
- Il peut y en avoir plusieurs (doivent se distinguer par le nombre et le type des arguments)

  => **surcharge** des constructeurs (comme pour les méthodes)
- Il en faut au moins 1

CONSTRUCTEURS EN JAVA

Il existe un constructeur par défaut (utilisé dans les exemples précédents)

- sans argument
- ne fait rien d'autre que l'allocation mémoire
- n'existe que si aucun autre constructeur existe

CONSTRUCTEURS EN JAVA : EXEMPLE DES COMPTES

```
Compte(){   // Remplace le constructeur par défaut
  solde = 0.0;
  client = new Client();  // Appel au constructeur par défaut de Client
}

public Compte(Client c){  // Constructeur valué à 1 argument
  solde = 0.0;
  client = c;
}

Compte(double s, String nomClient){ // Constructeur valué à 2 arguments
  solde = s;
  client = new Client(nomClient);  // Suppose qu'il existe le constructeur correspondant
                                   // dans la classe Client
}

Compte(Compte c){ // Constructeur de copie
  solde = c.solde;
  client = c.client;
}
```

CONSTRUCTEURS EN JAVA : EXEMPLE D'UTILISATION

```java
Compte dupont = new Compte();
Compte martin = new Compte(100.0, "Martin");
Compte durand = new Compte(martin);
```

MÉTHODES ET ATTRIBUTS

Notions de base

UTILISATION DES MÉTHODES ET ACCÈS AUX DONNÉES D'UN OBJET

Manipuler et modifier l'état d'un objet se fait en lui appliquant ses méthodes (celles définies dans sa classe) avec l'opérateur "point" : .

```
objet.methode(arg1, arg2, ... )
```

NB : l'objet qui possède la méthode n'est pas pris en argument, contrairement au C par ex.

NB : si l'objet n'est pas correctement initialisé on ne peut appeler ses méthodes (erreur de compilation ou d'exécution)

Rq : certains auteurs considèrent que les objets interagissent et communiquent entre eux par l'envoi de messages, l'objet étant le "receveur" et la méthode le "sélecteur"

UTILISATION DES MÉTHODES EN JAVA

```
Compte martin = new Compte();
Compte dupond = new Compte();

martin.afficher();          // 0
dupont.crediter(10);
dupont.afficher();          // 10
martin.afficher();          // 0

x = dupont.consulter();     // x = 10

Compte.debiter(5);          // Erreur de compilation : Compte n'est pas un objet
```

SURCHARGE

La **surcharge** d'une méthode (ou d'un constructeur) permet de définir plusieurs fois une même méthode/constructeur avec des **arguments différents**.

Le compilateur choisit la méthode qui doit être appelée en fonction du nombre et du type des arguments

NB : le type de retour ne compte pas, 2 méthodes ne peuvent pas se différencier uniquement avec lui

```
class Compte {
  ...

  void crediter (double n) { solde = solde + n; }
  void crediter (float f) { ... }
  void afficher () { System.out.println(solde); }
  void afficher (String format) { ... }

}
```

PASSAGE DES PARAMÈTRES

La règle générale de Java est que les arguments sont passés **par valeur** : l'appel de méthode se fait par copie des valeurs passées en argument.

Mais à cause des références ce n'est pas si simple...

MEMBRES DE CLASSE : STATIC

Membres qui ne dépendent pas d'une instance de la classe

- attribut
  - partagé, accessible et identique pour toutes les instances de la classe
- méthode
  - utilisable sans instance de la classe

Un attribut statique doit être accédé par des méthodes statiques

Ex. valeur de précision commune à tous les `Point2D`

```
public static double epsilon = 1E-5;
```

MEMBRES DE CLASSE : ACCÈS

Les membres de classe sont accédées au travers du nom de la classe

`NomDeClasse.nomDeVariable`

NB : à l'intérieur du corps d'une méthode statique il n'est possible d'accéder qu'aux membres statiques de la classe (`this` n'existe pas)

Ex.

```
System.out.println("Hello World");
//méthode "println"" de l'attribut static "out"" de la classe "System""
```

MEMBRES DE CLASSE : INITIALISATION

S'il y a besoin d'une initialisation autre qu'une valeur, c'est impossible dans un constructeur, puisque le membre doit exister indépendamment des objets

=> Bloc spécifique qui sera exécuté lors de la création de la classe

```
public class C {
  static int T[];
  static {
    T = new int[10];
    for (int i = 0 ; i<10 ; i++)
      T[i]=i;
  }
}
```

MODIFICATEUR `FINAL`

Indique que l'élément considéré ne pourra pas être modifié : on ne pourra lui donner une valeur une seule fois dans le programme

- Pour une classe : héritage interdit
- Pour une méthode : redéfinition interdite dans les classes filles
- Pour un attribut : constant pour chaque instance (mais peut avoir 2 valeurs différentes pour 2 instances)

MODIFICATEUR `FINAL` : VARIABLE

- Pour une variable locale : sa valeur ne pourra être donnée qu'une seule fois
  - type primitif : valeur ne peut changer
  - type référence vers objet : ne pourra référencer un autre objet (mais l'état de l'objet pourra être modifié)

```
final Employe e = new Employe("Bibi");
e.nom = "Toto";          // Autorisé
e.setSalaire(12000);     // Autorisé
e = new Employe("Bob"); // INTERDIT
```

MODIFICATEUR `FINAL` : VARIABLE DE CLASSE

- Pour une variable de classe : constante dans tout le programme
  - peut ne pas être initialisée à sa déclaration mais doit avoir une valeur à la sortie de tous les constructeurs ou doit recevoir sa valeur dans un bloc d'initialisation `static`
  - ex. `static final double PI = 3.14;`

EX. HELLO WORD EN JAVA

```java
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello World !");
    }
}
```

Quelle sont les propriétés du `main` ?

RÉFÉRENCES ET GESTION MÉMOIRE

RÉFÉRENCES : EXEMPLES

Attention aux conséquences

```
Compte dupont = new Compte();
Compte martin = new Compte();
Compte titi;
martin.afficher();
dupont.crediter(10);
dupont.afficher();
titi = dupont;
titi.afficher();
titi.debiter(5);
titi.afficher();
dupont.afficher();
dupont.crediter(100);
titi.afficher();
```

Quelles sont les valeurs imprimées ?

RÉFÉRENCE NULL

`null` signifie que la référence ne pointe vers aucune instance

L'utilisation d'une méthode (ou d'une donnée) à partir d'une variable de type référence à `null` provoque une **erreur à l'exécution**

```
Compte c = null; // équivalent à Compte c;
c.crediter(5);
```

```
=> Exception in thread main java.lang.NullPointerException at
ProgrammePrincipal.main(ProgrammePrincipal.java:4)
```

Cause classique de bugs

AUTO-RÉFÉRENCEMENT

Une référence particulière est celle à l'objet courant : mot clé **`this`**

Permet d'utiliser les attributs et les autres méthodes de l'objet “courant” dans une méthode

Permet de lever les ambiguités : quand un nom d'attribut est utilisé dans le corps d’une méthode, c'est implicitement celui de l'objet courant

Permet aussi à un objet de se passer lui-même en paramètre d'une méthode : `sender.message(this, content)`

AUTO-RÉFÉRENCEMENT EN JAVA

```
class Compte {
  double solde = 0.0;
  Client client;

  public Compte(Client client){  // Constructeur valué à 1 argument
    solde = 0.0;
    this.client = client; // this *obligatoire* ici
  }

  public Compte(){
    this(new Client()); //appel au constructeur précédent
  }
  //equivalent à : public Compte(){ solde = 0; client = new Client(); }

  void afficher () { System.out.println(solde); }

  void debiter (double n) {
    this.solde = this.solde - n;  // this optionnel ici
  }

  void modifierSolde (double solde) {
    this.solde = solde;           // this *obligatoire* ici
    this.afficher();              // this optionnel ici
  }
}
```

RÉFÉRENCES : AUTRES EXEMPLES

```
class Ecole {
    Compte compte;
    //...
}
```

```
Compte bnp = new Compte();
Compte bnp2 = bnp ;
Ecole ensiie;
ensiie = new Ecole();
ensiie.compte = bnp;
```

COMPARAISON DE RÉFÉRENCES

Comparer 2 références (avec ==) revient à comparer 2 adresses

```
class ProgPrincipal {
  public static void main (String arg[]){
    Compte c = new Compte();
    Compte c2 = c;
    Compte c3;

    if (c==c2) System.out.println("c=c2");
    else System.out.println("c!=c2");

    c3 = new Compte;

    if (c==c3) System.out.println("c=c3");
    else System.out.println("C!=c3");
  }
}
```

DESTRUCTION DES INSTANCES

Certains langages ont des fonctionnalités de destruction des instances (ex. c++)

En Java, la gestion mémoire est assurée automatiquement par le **Garbage Collector** (“ramasse-miettes”)

Lorsqu'une instance n'est plus référencée, c'est-à-dire plus accessible à partir des variables du programme, la mémoire qu'elle occupait est libérée

```
Compte dupont = new Compte();
Compte martin = new Compte();
Compte durant = new Compte();
...
dupont = null;  // l'instance précédemment référencée par dupont pourra étre détruite
durant = martin; // l'instance précédemment référencée par durant pourra étre détruite
```

NB : destruction asynchrone (thread) donc temporalité non garantie

RELATIONS ENTRE CLASSES

DÉLÉGATION

- Un objet o1 instance de la classe C1 utilise les services d'un objet o2 instance de la classe C2 (o1 délègue une partie de son activité à o2)
- Pour que la classe C1 utilise les services de la classe C2, elle possède une référence de type de la classe C2

```
public class Cercle {
    private Point centre; //Cercle "possède" un centre
    private double r;

    public void translater(double dx, double dy) {
        centre.translater(dx,dy);
    }
}
```

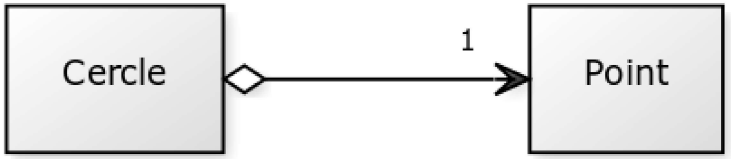
DÉLÉGATION : ASSOCIATION

L'attribut a une existence autonome (“Has-a relationship”)

Il peut être partagé (à un même moment il peut être lié à plusieurs instances d'objets)

Il peut être utilisé en dehors de la classe (NB : attention aux effets de bord)

```
public class Cercle {
  ...
  public Cercle( Point centre, double r) {
    this.centre = centre;
    this.r = r;
  }
  ...
}
```

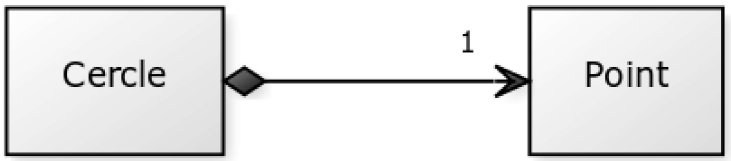


DÉLÉGATION : COMPOSITION

L'attribut n'est pas partagé (“Contains relationship”")

Les cycles de vie de l'objet et de son attribut sont liés

```
public class Cercle {
  ...
  public Cercle( Point centre, double r) {
    this.centre = new Point(centre);
    this.r = r;
  }
  ...
}
```



HÉRITAGE

Caractéristique fondamentale des langages objets (avec l'encapsulation)

Permet de construire une classe à partir d'une classe existante pour la rendre plus spécifique, plus spécialisée ou étendre ses fonctionnalités

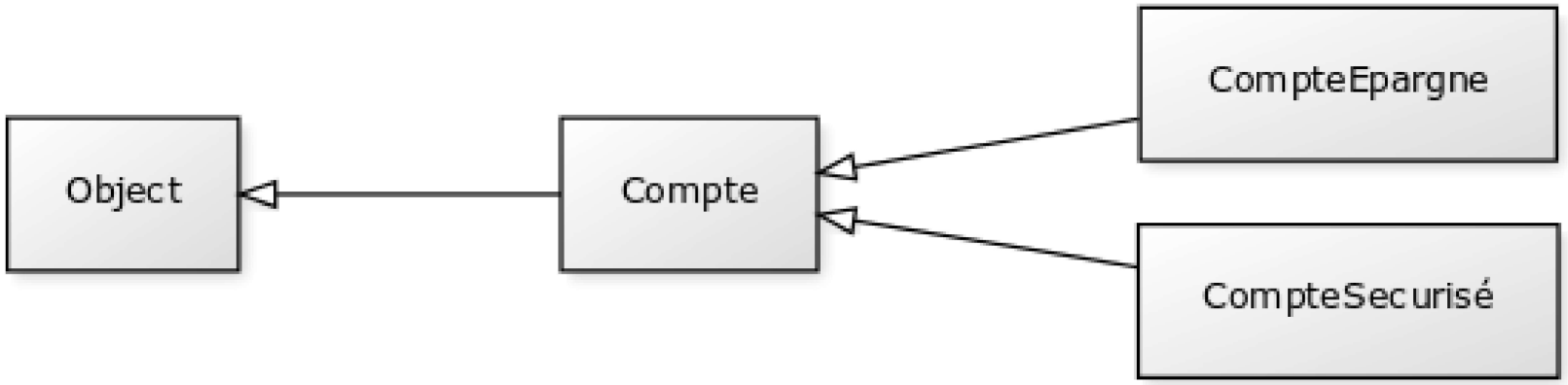
Une classe générale définit un ensemble d'attributs et/ou méthodes qui sont partagés par d'autres classes, qui "héritent de" (ou spécialisent) cette classe générale

- Consiste à factoriser certains attributs et/ou méthodes communs à plusieurs classes (définit différent niveaux d'abstraction)

HIÉRARCHIE

Cette relation d'héritage entre classes définit une **hiérarchie**

En Java la classe `Object` est la racine de l'arbre : toute classe hérite implicitement de cette classe existante



HÉRITAGE : VOCABULAIRE

- “*B* hérite de *A*”
- *A* = classe mère, ancêtre, super-classe de *B*
- *B* = classe héritière, classe fille, classe dérivée, sous-classe de *A*
- Notation : $A \leftarrow B$

HÉRITAGE : SOUS-TYPAGE

L'héritage dans les langages objets suit la règle suivante (polymorphisme d'inclusion) :

> ***Si C' est une sous-classe de C et o' un objet instance de C' alors o' a aussi le type C***

Cette règle suffit pour pouvoir utiliser une méthode qui attend des objets instances de la classe C avec des objets instances de la classe C'

=> Un code écrit pour des classes parentes peut être utilisé avec les classes filles

- *Ex* : la méthode `boolean estCrediteur (Compte x)` est utilisable sur les `CompteEpargne` et les `CompteSecurise`

HÉRITAGE : MEMBRES HÉRITÉS

Chaque classe héritière :

- hérite (possède) les attributs et méthodes de ses ancêtres (les connaissances les plus générales sont mises en commun)
- peut ajouter de nouveaux attributs
- peut ajouter de nouvelles méthodes
  - si même nom et paramètres différents = “surcharge” (“overloading”)
- peut modifier les méthodes héritées (sous certaines contraintes selon les langages)
  - même nom et mêmes paramètres = “redéfinition” (“overriding”)

EXEMPLES : AJOUT D'ATTRIBUTS

```
class Station {
  String nom;
  int nbHotel;
  int categorie;
  ...
}
```

```
class StationHiver extends Station {
  int nbRemontee;
  int nbPiste;
  Date ouverture;
  ...
}
```

```
class StationBalneaire extends Station {
  int nbPlage;
  Date ouverture;
  Color drapeau;
  ...
}
```

EXEMPLES : AJOUT DE MÉTHODES

```
class Personne {
  ...
  void setDomicile (String adresse) { ...}
}
```

```
class Employe extends Personne {
  void setBureau (String adresse) { ...}
  double calculerSalaire (...) { ... }
}
```

EXEMPLES : REDÉFINITION DE MÉTHODES

```
class Station {
  boolean reserverSejour(int n, Date d){ ... }
  ...
}
```

```
class StationHiver extends Station {
  boolean reserverSejour(int n, Date d){ ... } // redéfinie pour inclure la location
                                               // de skis et l'achat du forfait
  ...
}
```

EXERCICES

1. Définir la nouvelle classe des comptes d'épargne, qui sont des comptes qui possèdent un `taux` en % et une méthode `appliquerInterets()` qui permet d'augmenter le solde selon ce taux

EXERCICES

2. Définir la classe des comptes sécurisés pour lesquels un retrait n'est possible que si le solde est suffisant. Dans le cas contraire le solde n'est pas modifié et un message est affiché.

HÉRITAGE ET MÉTHODES

L'appel à une méthode déclenche un mécanisme de recherche dans le graphe d'héritage jusqu'à trouver une méthode ayant un nom et une signature correspondant à l'appel

- Surcharge = résolu à la compilation
- Redéfinition (la méthode a le même profil dans la classe fille)
  - résolu à l'exécution : la méthode réellement appelée dépend de la classe **réelle** de l'objet qui exécute cette méthode
  - **"polymorphisme"** d'héritage

Appel explicite de la méthode de la classe mère : `super.methode()`

HÉRITAGE ET REDÉFINITION

```
class CompteSecurise extends Compte{
    void debiter(double d, boolean depassement){
        if (solde - d < 0)
            if (depassement)
                solde = solde - d ;
        else solde = solde - d;
    }

    double consulter(){
        afficher();
        return solde;
    }
}

CompteSecurise dupont = new CompteSecurise();
dupont.debiter(10,false);
dupont.afficher();
dupont.debiter(5);
dupont.afficher();
```

`débiter` et `consulter` sont redéfinies ou surchargées ?

HÉRITAGE SIMPLE ET MULTIPLE

Héritage simple

on hérite d'**une seule classe**, chaque classe a un seul père dans le graphe d'héritage (hiérarchie = arbre)

Ex : Java

Héritage multiple

on hérite de **plusieurs classes simultanément** (hiérarchie = graphe orienté sans circuit)

Ex : C++, OCaml, Eiffel

Possibilité de conflit (“deadly diamond”)

B et *C* héritent de *A* et rédéfinissent sa méthode *m*, si *D* hérite de *B* et *C* quelle est la méthode *m* dont hérite *D* ?

HÉRITAGE ET CONSTRUCTEURS

Chaque fois qu'un objet est créé les constructeurs sont invoqués en remontant en séquence de classe en classe dans la hiérarchie jusqu'à la classe `Object`

- le constructeur de la classe `Object` est toujours exécuté en premier, suivi des constructeurs des différentes classes en redescendant dans la hiérarchie
- Garantit qu'un constructeur d'une classe est toujours appelé lorsqu'une instance de l'une de ses sous-classes est créée

HÉRITAGE ET CONSTRUCTEURS

Appel explicite des constructeurs de la classe mère : `super(paramètres constructeur)`

- utilisation analogue à celle de `this()`
- permet de réutiliser le code
- doit toujours être la **1ère instruction** dans le corps du constructeur

```
public class PointCouleur extends Point {
    Color c;

    public PointCouleur(double x, double y, Color c){
        super(x,y);
        this.c = c;
    }
}
```

Rq : S'il n'y a pas d'appel explicite, appel implicite à `super()` mais ce constructeur par défaut de la classe mère peut ne plus exister...

MÉTHODES ET CLASSES ABSTRAITES

Méthode “abstraite” ou “virtuelle” = méthode dont il manque le corps (seul son type et ses paramètres sont donnés)

Classe abstraite = classe qui possède une ou plusieurs méthodes abstraites (ou qui est définie explicitement comme telle)

- Une classe abstraite **ne peut pas être instanciée**
- Une classe abstraite peut être héritée
- Une méthode abstraite peut être utilisée dans une méthode concrète

En Java, mot clé `abstract`

CLASSES ABSTRAITES ET HÉRITAGE

Une sous-classe peut concrétiser tout ou partie des méthodes abstraites. S'il en reste elle sera elle-même abstraite.

```
abstract class Figure {
    abstract double aire();
}
```

```
class Carre extends Figure{
    double cote;
    double aire (){
         return cote*cote;
    }
}
```

```
class Cercle extends Figure{
    double rayon;
    private double pi = 3.14;
    double aire (){
        return pi*rayon*rayon;
    }
}
```

CLASSES ABSTRAITES ET HÉRITAGE

```
abstract class Aliment{
    String nom;
    abstract String modeDeConsommation();
    void info(){
        System.out.println(this.modeDeConsommation());
    }
}

class Pain extends Aliment{
    String modeDeConsommation(){
        return "cuit";
    }
}

class FoieGras extends Aliment{
    String modeDeConsommation(){
    return "cru ou cuit, avec un verre de Sauternes";
    }
}
```

INTERFACE : DÉFINITION ET DÉCLARATION

Une interface est comparable à une classe abstraite dont toutes les méthodes seraient abstraites

Une interface est un **contrat** entre une classe qui implémente l'interface et une classe qui utilise l'interface

En Java mot-clé `interface`

Ex : l'interface `Runnable` implémentée par les threads

```
public interface Runnable {
  public void run();
}
```

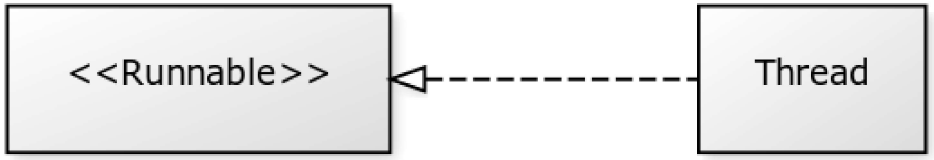
Rq : Depuis Java 8 il est possible d'implémenter des méthodes dans une interface (avec le mot-clé `default`) ce qui rend la différence moins claire avec les classes abstraites

INTERFACE : IMPLÉMENTATION

Fonctionne un peu à la manière d'un héritage avec mot-clé `implements`

Ex : la classe `Thread` qui implémente `Runnable`

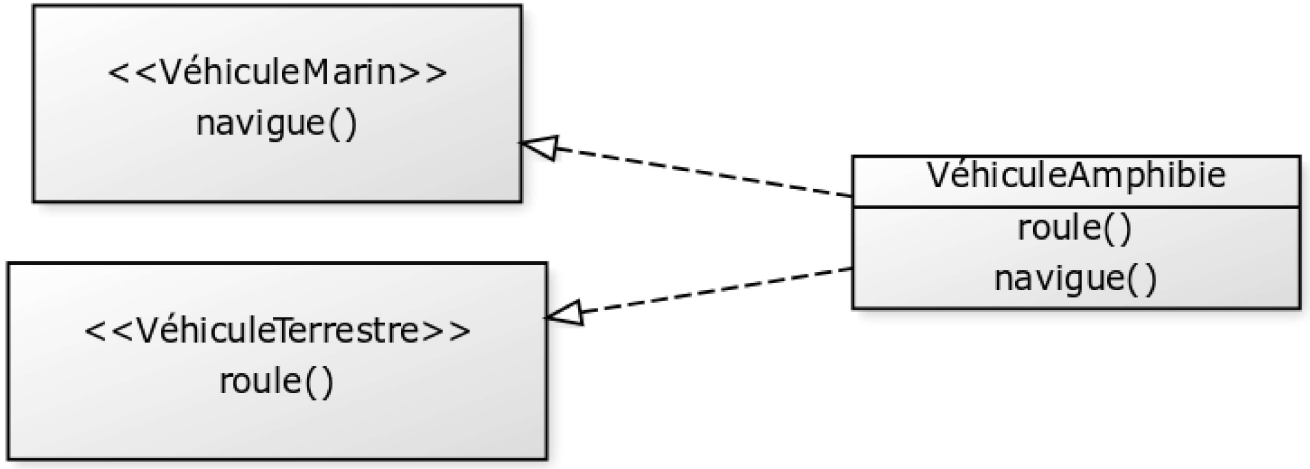
```
public class Thread implements Runnable {
  public Thread()  target= null; ...
  public Thread(Runnable t)  target = t ; ...
  ...
  public void run() {
    if (target != null) target.run();
  }
  ...
  private Runnable target;
  ...
}
```



INTERFACES ET HÉRITAGE

L'héritage multiple n'existe pas en Java mais les interfaces sont un moyen de s'en approcher

- Il est possible d'implémenter plusieurs interfaces
- Chaque interface donne un comportement
- Une classe peut donc hériter de plusieurs comportement en implémentant plusieurs interfaces



L'ENCAPSULATION

NOTION D'ENCAPSULATION

Consiste à protéger les données d'un objet de modifications extérieures (autres classes).

Une classe décrit

- une partie privée, cachée
  - structure de données interne (attributs)
  - corps des méthodes (algorithmes)
- une partie publique, visible (interface)
  - noms et paramètres des méthodes

NOTION D'ENCAPSULATION

Conséquences

- Les attributs d'un objet ne sont souvent visibles/accessibles (et surtout modifiables) qu'au travers de méthodes prévues à cet effet
- Un objet ne peut être utilisé que de la manière prévue à la conception de sa classe, sans risque d'utilisation incohérente

=> Robustesse du code

- Permet de masquer l'implémentation : la modification interne des objets ne doit pas affecter les programmes qui les utilisent

=> Facilite l'évolution du logiciel

VISIBILITÉ

Outil pour l'encapsulation

- Modificateur (mot clé) lors de la définition

Plusieurs niveaux de visibilité peuvent être définis pour les classes, les attributs et les méthodes, en général 3 (dépendant du langage) :

- public
- protégé
- privé

VISIBILITÉ EN JAVA : PUBLIC

- élément visible de partout
- classe
  - peut être utilisée dans n'importe quelle autre classe
- attribut
  - accessible directement depuis le code de n'importe quelle classe
- méthode
  - peut être invoquée depuis code de n'importe quelle classe

```
public class MaClasse {
  public int monEntier;
  public void afficher() { ... }
}
```

VISIBILITÉ EN JAVA : PRIVATE

- élément visible uniquement pour les méthodes de la classe
- classes
  - interdit
- attribut :
  - accessible directement uniquement depuis le code de la classe qui le définit
  - depuis l'extérieur devra passer par une méthode visible
- méthode :
  - peut être invoquée uniquement depuis code de la classe qui le définit (usage interne)

```
public class MaClasse {
    private int monEntier;
    public int getEntier() { return monEntier; }
```

VISIBILITÉ EN JAVA : NOTION DE PACKAGE

Les classes peuvent être organisées en répertoires particuliers, les packages (voir TP)

```
package Comptes;
public class Compte{
  ...
}
```

Sans instruction explicite, le package sera `default`

VISIBILITÉ EN JAVA : PROTECTED

- classe :
  - peut être utilisée uniquement dans une autre classe du même package
- attribut/méthode : visible uniquement
  - dans la classe où il est défini
  - dans toutes ses sous-classes
  - dans toutes les classes du même package

VISIBILITÉ EN JAVA : PAR DÉFAUT

En cas d'absence de modificateur

- classe :
  - équivalent au mode `protected` (peut être utilisée dans une autre classe du même package)
- attribut/méthode :
  - visible dans les classes du même package que celui de la classe où il est défini

Donc, pas d'encapsulation de données par défaut => **DANGER !**

- Tout utilisateur peut accéder et modifier les attributs de l'objet

VISIBILITÉ EN JAVA : EXEMPLE

```
public class Compte {
  ????? double solde;
  ????? Client client;

  public Compte(Client client){
    solde = 0.0;
    this.client = client;
  }

  public void debiter (double n) { solde = solde - n; }
  public void crediter (double n) { solde = solde + n; }
  public double consulter () { return solde; }
  public void afficher () { System.out.println(solde); }
  private double calculInterne() {...}
}
```

VISIBILITÉ EN JAVA : EXEMPLE

- `public int solde`
  - `titi.solde = -100` est possible, même si un compte sécurisé devrait toujours avoir un solde positif ou nul
- `private int solde`
  - `solde` n'est pas visible dans `CompteEpargne` et `CompteSecurisé` : il faut utiliser la méthode `consulter` pour y accéder et `titi.solde = -100` est interdit
- `protected int solde`
  - `solde` est visible dans `CompteEpargne` et `CompteSecurisé` mais pas à l'extérieur

ENCAPSULATION : RÈGLES CLASSIQUES

- Attributs `private` ou `protected` (si les sous-classes doivent les utiliser directement)
- Pour chaque attribut, définir les méthodes d'accès `public` (les “accesseurs”)
  - *Ex.* en lecture : `getX`, `getY` pour `Point`, `consulter` pour `Compte`
  - *Ex.* en écriture : `setX`, `setY` pour `Point`, `débiter` et `créditer` pour `Compte`
- Limiter au strict nécessaire les autres méthodes publiques

Les méthodes redéfinies dans les classes filles doivent fournir au moins les mêmes droits d'accès que dans la classe mère

CLASSES INTERNES/IMBRIQUÉES

Classes déclarées à l'intérieur d'une classe

- Classe intérieure considérée comme un membre de la classe contenante
- Classe englobante a accès à tous les membres de la classe intérieure
- Classe intérieure peut être `private, public, protected` ou par défaut “package private”
- Classe intérieure peut être `abstract` ou `final`

CLASSES INTERNES/IMBRIQUÉES

On distingue

- Classe interne (*inner class* ou *non static nested class*)
  - accès aux autres membres, même privés
  - instances liées aux instances de la classe englobante
- Classe imbriquée (*static nested class*)
  - accès aux autres membres `static`, même privés
  - instances pas liées à une instance de la classe englobante

CLASSES INTERNES/IMBRIQUÉES

```
class EnclosingClass {
    ...
    static class StaticNestedClass {
        ...
    }
    class InnerClass {
        ...
    }
}
```

La classe englobante fournit un espace de noms pour les classes internes : son nom est de la forme `EnclosingClass.InnerClass`

CLASSE LOCALE (ANONYME)

(Sous-)classe déclarée dans un bloc (ex. une méthode)

```
class Hello {
  public void read() {
    System.out.println("Hello!");
  }
}

class Website {
  Hello helloInstance = new Hello() {
    public void read() {
      System.out.println("Bonjour");
    }
  }; //obligatoire !
}
```

Utile si

- implémentation courte
- une seule instance est nécessaire
- utilisée directement après la définition

REMERCIEMENTS

Ressources et conseils

- C. Dubois, *ENSIIE*
- J-Y. Didier, *Univ. Evry*
- R. Décamps, *Google*

Cours en ligne

- G. Picard, *ENS Mines de St Etienne*
- P. Genoud, *IMAG*