POO et langage JAVA

"Un langage de programmation est censé être une façon conventionnelle de donner des ordres à un ordinateur. Il n'est pas censé être obscur, bizarre et plein de pièges subtils (ça ce sont les attributs de la magie)." *Dave Small*

Programmation Orientée Objet

Programmation "classique"

- Objectif: écrire une séquence d'instructions pour qu'un ordinateur réponde à un problème donné
- Pour cela:
  - variables,
  - instructions: tests, boucles,…
  - décomposition en fonctions,
  - passages de paramètres,
  - structures,…

Programmation non objet

- Le programme est une fonction principale
- Cette fonction appelle des sous-fonctions, qui en appellent d'autres…
- Construction d'un programme
  - Réflexion sur les objectifs du programme
  - Découpage du programme en fonctions
  - Construction du programme par appels de ces fonctions
- + algo complexe, + décomposition importante

Programmation non objet

- Le +: factorisation des comportements
  - découpe fonctionnelle "intelligente" = factorisation de certains comportements
  - la prog de certaines fonctions réutilise des fonctions déjà définies (rendre ces fonctions génériques)

Exple: *bibliothèque avec Romans et Revues*

Programmation non objet

- Inconvénient: maintenance complexe si évolution
  - les fonctions deviennent interdépendantes
  - mise à jour à un endroit peu impacter beaucoup d'autres fonctions

Exple: *bibliothèque avec Romans et Revues*

On veut ajouter d'autres types de livres: BD, mangas,…
Les fonctions doivent être modifiées (lesquelles ?, être sûr qu'une modif ne change pas le fonctionnement pour les anciens types…)

7

Programmation non objet

- Variables
  - portée: à l'intérieur d'un bloc { }
- Variable globale
  - visible dans tout le programme, pour toute la durée du programme
  - Utilisation + simple mais inefficace et déstructurant
    - Inefficace: pas stockée dans la pile mais dans la mémoire globale -> accès plus long
    - Déstructurant: éparpillement des données et des fonctions

8

Programmation non objet

- Variables
  - portée: à l'intérieur d'un bloc { }
- Variable globale
  - visible dans tout le programme, pour toute la durée du programme
  - Utilisation + simple mais inefficace et déstructurant
  - tend à disparaître en POO

9

Prog non objet vers POO

- 1ère approche: vision « fonctionnalités »
  - Programmation non objet
    - Reçoit une donnée
    - Traite la donnée
    - Donne un résultat
- 2ème approche: vision « objet »
  - Programmation Orientée Objet (POO)
    - 1 donnée = 1 objet
    - Chaque objet a ses données propres, ses fonctions propres
    - 2 objets de même type ont les mêmes caractéristiques

10

Un objet, c'est quoi ?

- Regroupement d'info qui caractérisent l'objet
  - Les données (les **attributs**)
    - Ce qui définit l'objet
  - Les traitements (les **méthodes**)
    - Ce que peut faire l'objet (son comportement)

Date
- 1 Février 2023
- afficher: println(« 01/02/2023 »)
- dateEnSecondes: retourner 1675206000

Roman
- Titre: « Harry Potter »
- Ref: R234
- datePrêt: 1 Février 2023
- afficher: « Harry Potter, R234, en prêt »
- prêter: modifier la date de prêt
- rendre: annuler la date de prêt

11

Objet vs Classe

- Classe: modèle générique d'un objet
  - Définit les caractéristiques des objets
  - Tous les objets de cette classe ont les mêmes caractéristiques
- Objet = instance d'une classe

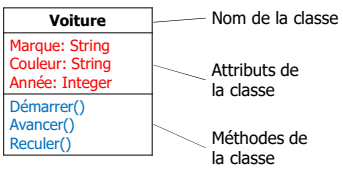
Classe **Date**
- jour, mois, année (entier)
- void afficher(){…}
- int dateEnSecondes(){…}

Classe **Roman**
- Titre (String)
- Ref (entier)
- datePrêt (Date)
- void afficher(){…}
- void prêter(){…}
- void rendre(){…}

12

Objet vs Classe: autre exemple

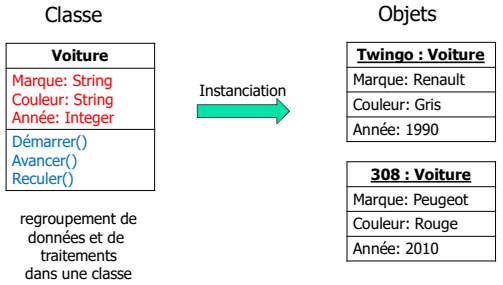
Classe



regroupement de données et de traitements dans une classe

13

Objet vs Classe: autre exemple



regroupement de données et de traitements dans une classe

14

Paradigme Objet

- La vision Objet de la programmation possède 3 grands principes:
  - **L'encapsulation**: principe d'abstraction de données et abstraction procédurale
  - **L'héritage**: principe de Généralisation/Spécification
  - **Le polymorphisme**: permet à une méthode de s'adapter à plusieurs classes

15

L'encapsulation

- Le principe d'encapsulation se base sur les idées suivantes
  - un objet rassemble en lui même ses données (les attributs) et le code capable d'agir dessus (les méthodes)
  - Abstraction de données: la structure d'un objet n'est pas visible de l'extérieur, son interface est constituée de messages invocables par l'utilisateur (= un objet ne change d'état que par l'utilisation de méthodes)

16

Parties publiques/privées

- Partie publique (≡ interface): un objet possède un certain nombre de méthodes lui permettant de répondre aux différentes interrogations de l'extérieur
- Partie privée: ce qui concerne l'implémentation interne de l'objet doit rester caché à l'utilisateur

17

Parties publiques/privées

- Pourquoi utiliser une partie privée?
  - Simplification pour l'utilisateur: certains calculs internes n'ont pas d'utilité pour l'utilisateur -> ne surcharge pas l'interface
  - Intérêt important pour le développeur: possibilité de changements (structure, calculs,…) sans perturber l'utilisateur

18

Parties publiques/privées

- Accessibilité, syntaxe
  - Les données accessibles depuis l'extérieur de la classe (données publiques) sont précédées du mot clé **public**
  - Les données non accessibles depuis l'extérieur de la classe (données privées) sont précédées du mot clé **private**

19

Encapsulation

- Le principe d'encapsulation n'est pas imposé par les langages OO, **c'est au programmeur de le respecter**.
- Exemple

Attributs en private

Méthodes:
- private pour les internes
- public pour l'interface avec l'utilisateur

```
public class Date {
   private int jour, mois, annee;

   public void affiche(){ … }

   public int compare(Date d){ return this.jours()-d.jours(); }

   private int jours(){ return … }
}
```

ici en private car calcul interne, non utile à l'utilisateur

20

Getters / Setters

- Encapsulation: attributs privés
  - Pas d'accès en lecture (int x=d.jour; *non autorisé*)
  - Pas d'accès en écriture (d.jour=12; *non autorisé*)
- Comment accéder aux attributs depuis l'extérieur de la classe ?
  - On crée des méthodes pour cela
    - **Getter**: accesseur à un attribut, retourne la valeur de l'attribut
      - Syntaxe: public type getNomAttribut(){ return attribut; }

21

Getters / Setters

- Exemple

```
public class Date {
   private int jour, mois, annee;          Attributs en private

   public int getJour() {
      return this.jour;
   }

   public int getMois() {
      return this.mois;
   }

   public int getAnnee() {
      return this.annee;
   }

}
```

Accesseurs (getters)

22

Getters / Setters

- Comment accéder aux attributs depuis l'extérieur de la classe ?
  - On crée des méthodes pour cela
    - Getter: accesseur à un attribut, retourne la valeur de l'attribut
      - Syntaxe: public type getNomAttribut(){ return attribut; }
    - **Setter**: mutateur de l'attribut, modifie la valeur de l'attribut
      - Syntaxe: public void setNomAttribut(type val){ attribut=val; }

23

Getters / Setters

- Exemple

```
public class Date {
   private int jour, mois, annee;          Attributs en private

   public int getJour() {return this.jour;}

   public int getMois() {return this.mois;}

   public int getAnnee() {return this.annee;}

   public void setJour(int jour) {
      this.jour = jour;
   }

   public void setMois(int mois) {
      this.mois = mois;
   }

   public void setAnnee(int annee) {
      this.annee = annee;
   }
}
```

Accesseurs (getters)

Mutateurs (setters)

24

Getters / Setters

- Pourquoi des getters/setters au lieu de mettre simplement l'attribut en *public* ?

```
public class Date {
   private int jour, mois, annee;

   public int getJour() {return this.jour;}
   public int getMois() {return this.mois;}
   public int getAnnee() {return this.annee;}

   public void setJour(int jour) {this.jour = jour;}
   public void setMois(int mois) {this.mois = mois;}
   public void setAnnee(int annee) {this.annee = annee;}
}
```

Maintenant l'utilisateur de la classe ne peut plus modifier l'attribut *jour*

Donne la liberté au développeur de laisser accès (ou non), en lecture et/ou en écriture aux attributs.

25

Vie d'un objet, de sa création à sa destruction

26

Cycle de vie d'un objet

- Avant d'utiliser un objet
  - Instanciation: création d'un objet
    - Allocation mémoire de l'espace nécessaire
    - Appel d'une méthode particulière: **constructeur**
- Utilisation de l'objet
  - Appel de ses méthodes
- L'objet n'est plus utilisé
  - Libération de la mémoire prise par l'objet
    - Appel d'une méthode particulière: **destructeur**

27

Constructeur/Destructeur

- Constructeur *Joue le rôle des fonctions init que nous faisions en Processing*
  - Méthode appelée lors de la création de l'objet
  - Rôle: initialiser les attributs de l'objet (rappels, ils sont privés et il n'existe pas forcément de setter, il faut quand même les initialiser)
- Destructeur
  - Méthode appelée lors de la destruction de l'objet
  - Rôle: libérer l'espace mémoire utilisé par les attributs de l'objet

En Java, pas de destructeur

28

Constructeur (1/3)

- Particularités
  - Méthode appelée automatiquement lors de la création de l'objet
  - Méthode portant le nom de la classe
  - Aucun type de retour
  - Si aucun constructeur défini: par défaut un constructeur vide est utilisé
  - Possibilité de surcharger un constructeur
    - différence sur le nombre et les types de paramètres (même nom, pas même prototype)

29

Constructeur (2/3)

- Exemple

```
public class Personne{
//attributs
private String nom, prenom;
private int age;

//constructeurs
public Personne(){
   nom=prenom=""; age=0;
  }

public Personne(String nom, String p, int a){
   this.nom=nom; prenom=p; age=a;
  }

}
```

Constructeur par défaut (pas de paramètres) On initialise les attributs avec des valeurs par défaut

Constructeur avec paramètres On initialise les attributs avec les valeurs reçues

30

Instanciation

- Créer des instances de classes ⇒ objets
- Instanciation (Java):
  - Déclaration d'un objet
  - Création de l'objet (opérateur **new**)
  - Exemple:

```
Personne x;
x=new Personne();
Personne y=new Personne("Croche","Sarah",22);
```

31

Constructeur (3/3)

- Exemple

```
public class Personne{
//attributs
private String nom, prenom;
private int age;

//constructeurs
public Personne(){
      nom=prenom=""; age=0;
  }

public Personne(String nom, String p, int a){
      this.nom=nom; prenom=p; age=a;
  }

}
```

Instanciations

```
Personne x = new Personne();

Personne y = new Personne("Croche" , "Sarah" , 22);
```

32

Références d'objets

Personne x,y; *// x et y ne sont pas des objets mais des références*

x=new Personne("Smith","Bob", 31); *// x référence une personne (Bob Smith 31ans)*

y=new Personne("Dupont","Alain",22); *// y référence une personne (Alain Dupont 22ans)*

x=y; *// x et y référencent la même personne (Alain Dupont)*

La personne "*Bob Smith*" n'est plus référencée

33

Garbage collector (Java)

- Détermine les objets inutilisés (non référencés)
- Objectif: récupérer leur espace mémoire
- Non maîtrise du processus: déclenchement automatique (mais possibilité de l'utiliser à la demande)
- Pb de ressource mémoire toujours possible
- Avantage: robustesse de l'application
- Inconvénient: coût

34

Création/destruction d'un objet

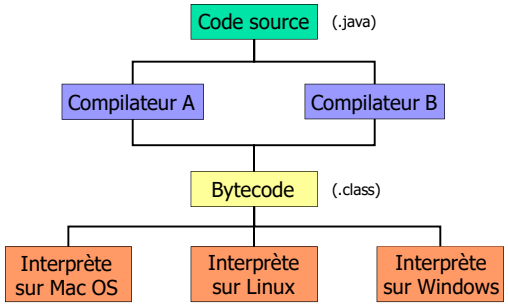
- Java
  - Création d'un objet:
    - Instanciation: opérateur **new**
    - Appel automatique du constructeur
  - Destruction de l'objet: garbage collector

35

Langage Java

36

Pourquoi un langage dit "portable" ?



37

Terminologie (1/2)

- JDK: Java Development Kit
  - Environnement dans lequel le code Java est compilé pour être transformé en *bytecode*
    (compilateur .class, débogueur, archiveur .jar, générateur documentation)
- Bytecode
  - code non directement exécutable
  - interprété par une machine virtuelle ⇒ portabilité

38

Terminologie (2/2)

- JRE: Java Runtime Environment (*Environnement d'exécution Java*)
  - Ensemble d'outils permettant l'exécution de programmes Java sur toutes les plateformes supportées
  - Constitué de
    - JVM: Java Virtual Machine.
      - Interprète le code Java et le convertit en code natif
    - bibliothèque standard à partir de laquelle doivent être développés tous les programmes en Java.

39

Téléchargements

- Java Development Kit (JDK)
  - Le JDK est disponible gratuitement en téléchargement sur le site d'*Oracle*
- IDE: JetBrains IntelliJ IDEA

40

Java: Tout est objet

- Tout est décrit sous forme de classes et d'objets
- Sauf: types primitifs: int, double, boolean,…
  - Par défaut: entiers -> int , réels -> double
  - Il existe une vision Orientée Objet de chaque type primitif: int->Integer, double->Double,…
- Les chaînes de caractères sont des objets
  - classe String

41

Le prog principal

- Il est dans une classe
- Le prototype est fixe

```
public class MainClass {
    public static void main(String[] args){
    }
}
```

42

Exemple

```
public class MainClass {

    public static void main(String[] args){

        Pion p=new Pion();
        Jeu j=new Jeu();

        p.setPosition(5);
        int x=p.getPosition();
        x=j.getRouge().getPosition();
        j.getBleu().setPosition(8);
        j.setBleu(p);
    }

}
```

43

static ?

- On a défini les attributs et méthodes d'instance
  - les attributs sont dupliqués dans chaque objet
  - Chaque objet possède une copie des méthodes

44

static : att/méth de **classe**

- **static** signifie: non rattaché à une instance de la classe
- Attribut static
  - commun à toutes les instances de la classe
  - existe même si aucune instance de la classe n'est créée
  - accès via une instance ou directement via la classe

45

static : att/méth de classe

```
public class Element {
    //pour l'exemple on utilise des attributs publics
    public static int valeurStatic; //attribut static: attribut de classe
    public int valeurClassique; //attribut classique: attribut d'instance

    //constructeur initialisant l'attribut d'instance
    public Element(int valeurInit) {
        valeurClassique = valeurInit;
    }
}
```

Element a=new Element(12);
Element b=new Element(34);

a.valeurStatic = 100;		**Element.valeurStatic** = 50;	
Affiche: a.valeurClassique	→ 12	Affiche: a.valeurClassique	→ 12
Affiche: a.valeurStatic	→ 100	Affiche: a.valeurStatic	→ 50
Affiche: b.valeurClassique	→ 34	Affiche: b.valeurClassique	→ 34
Affiche: b.valeurStatic	→ 100	Affiche: b.valeurStatic	→ 50

46

static : att/méth de classe

```
public class Element {
    //pour l'exemple on utilise des attributs publics
    public static int valeurStatic; //attribut static: attribut de classe
    public int valeurClassique; //attribut classique: attribut d'instance

    //constructeur initialisant l'attribut d'instance
    public Element(int valeurInit) {
        valeurClassique = valeurInit;
    }
}
```

Main:

Element.valeurStatic = 26;
Affiche: Element.valeurStatic → 26

Accès possible sans créer d'instance

47

static : att/méth de classe

- Méthode static
  - commune à toutes les instances de la classe
  - existe même si aucune instance de la classe n'est créée
  - accès via une instance ou directement via la classe
  - ne peut pas utiliser des attributs non static d'une classe (déduction du point n°2)

48

static : att/méth de classe

```
public class Element {
    //pour l'exemple on utilise des attributs publics
    public static int valeurStatic; //attribut static: attribut de classe
    public int valeurClassique; //attribut classique: attribut d'instance

    //constructeur initialisant l'attribut d'instance
    public Element(int valeurInit) {
        this.valeurClassique = valeurInit;
    }

    public static void increment(){
        valeurStatic++;
        //valeurClassique++; //interdit, n'a pas de sens
    }

}
```

49

final = constant

- **final** signifie: ne peut pas changer
- Attribut final
  - Défini à la déclaration (ex: final int x=3;)
    - compilateur inclut "en dur" la valeur de la constante là où elle intervient
  - Déclaré et défini dans le constructeur
    - défini dans tous les constructeurs de la classe
    - une fois défini, ne peut plus être modifié

50

final = constant

- Attribut final
  - exemple

```
public class MaClasse {
    public final int x=3;
    public final int y;

    public MaClasse(int value) {
        y = value;
        //x=value; //impossible car attribut déjà défini
        //y=2; //impossible car attribut déjà défini
    }
}
```

51

final = constant

- Attribut final
  - Attribut de type instance (ex: final MaClasse x;)
    - la référence n'est pas modifiable
    - l'accès aux attributs de l'instance est possible

52

final = constant

- Argument final
  - possibilité de créer des méthodes avec des arguments « final »

```
public class MaClasse {
    public final int x=3;
    public final int y;
    public int z;

    public MaClasse(int value) {...}

    public void f(final int value){
        z=value;
        //value++; //impossible car valeur non modifiable
    }
}
```

53

final = constant

- Méthode final: 2 intérêts
  - verrouiller la méthode: empêche la redéfinition de la méthode dans les classes dérivées
  - efficacité
    - le compilateur peut remplacer l'appel de la méthode directement par le code de celle-ci (afin d'éviter un appel de fct)
    - si le code est trop gros, le gain de temps devient nul et le compilateur ne fait pas ce changement
    - à réserver aux méthodes courtes

54

final = constant

- Classe final
  - interdit d'hériter de cette classe