Classes et Objets

Maria Virginia Aponte

CNAM-Paris

28 février 2016

1. Concepts fondamentaux en POO

Les données en programmation

- Données élémentaires : donnée simple.
  Ex : un entier, nombre à virgule, caractère, booléen, etc ;
- Données structurées : plusieurs données mises ensemble (types hétérogènes).
  - tableaux, Strings (même type) ;
  - enregistrements, objets (types $\neq$).

  Utilité :
  - regrouper plusieurs variables reliées ;
  - structurer/faciliter la programmation et la réutilisation !

Les objets « en 1 transparent »

Ils possèdent :

- des données locales (état interne) ;
- des opérations agissant sur ces données locales (méthodes) ;

Ils sont crées à partir :

- d'une classe ou type ;
- via l'opération `new`

Appliquer les méthodes

La méthode `m` d'un objet `o` s'applique **sur** celui-ci par `o.m()`, et non pas par `m(o)`;

1er exemple : objets de type String

```
String s = "Bonjour";
```

- données locales (état interne) ⇒ caractères de la chaîne accessibles par indice : ’B’ ,’o’ ,’n’ , ’j’, ’o’, ’u’, ’r’
- opérations sur données locales (méthodes) ⇒
  - `charAt(int)`, `equals(String)`, `equalsIgnoreCase(String)`, `substring(int)`, etc.

Appliquer les méthodes

```
char c = s.charAt(0); // 1er carcatere
bool b = s.equalsIgnoreCase("BONjour"); // true
```

Des objets comptes bancaires

Données (locales) d'un compte :

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

$\Rightarrow$ plusieurs données de types $\neq$.

Opérations sur (les données locales) d'un compte :

- initialiser : nom titulaire, numéro compte, solde ;
- obtenir solde courant, réaliser retrait, dépôt ;
- consulter historique des opérations . . .

Objets : données structurées (ou complexes)

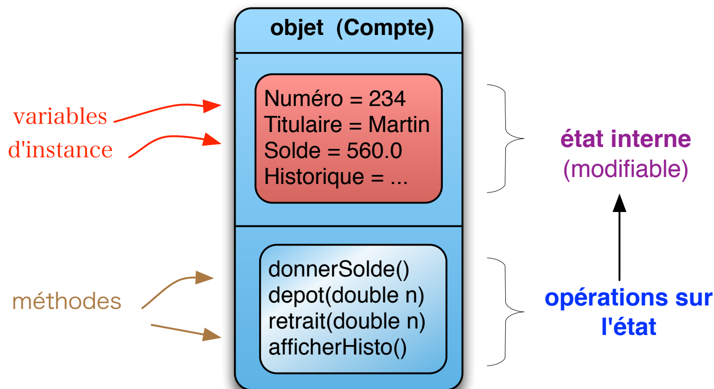
Objet

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

Un objet compte bancaire contient

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

Un objet Compte



Objet = état interne + opérations sur l'état

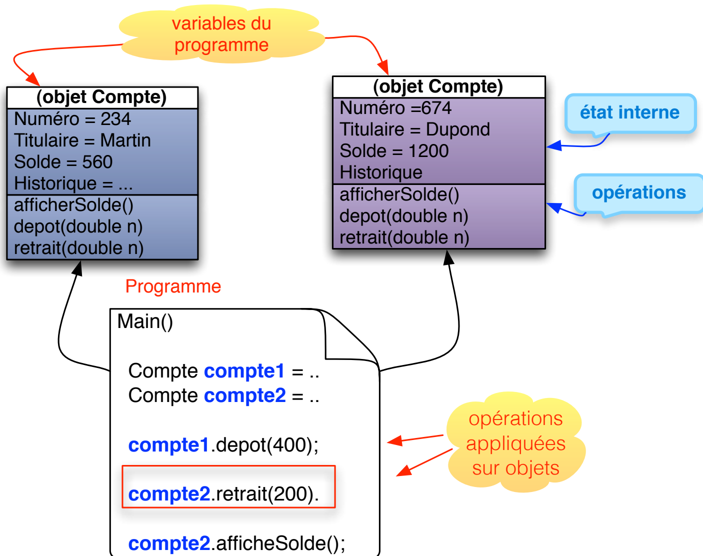
Gestion de comptes en POO

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



Classe = Moule à objets + Type

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

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

On dira : *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 ;

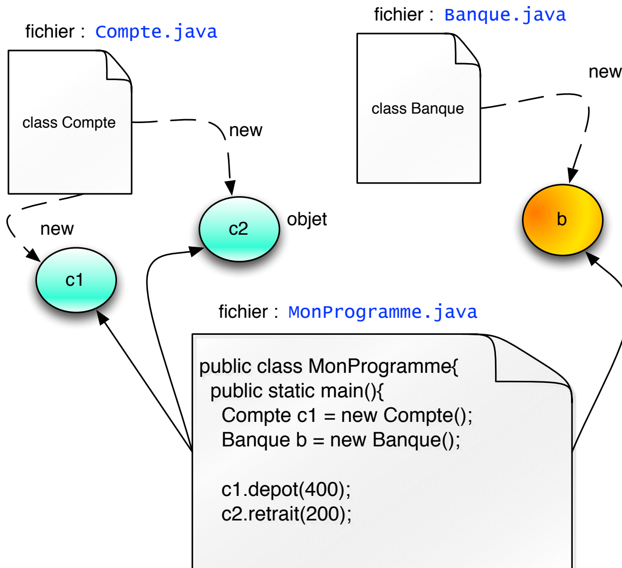
Classe ⇒ "moule à objets" + type

- décrit structure interne + opérations des objets à créer ;
- devient le type des objets crées ;

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.

Un programme OO



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

Déclaration d'une variable de type objet :

```
Compte c1;  // l'objet n'existe pas
```

Création, initialisation (en mémoire)

de l'objet "mis" dans la variable :

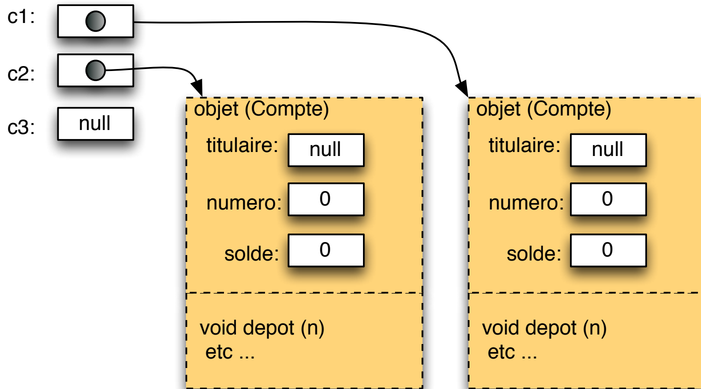
```
c1 = new Compte();  // l'objet existe
```

Après création

variables internes de c1 ⇒ initialisées (défaut ou constructeur).

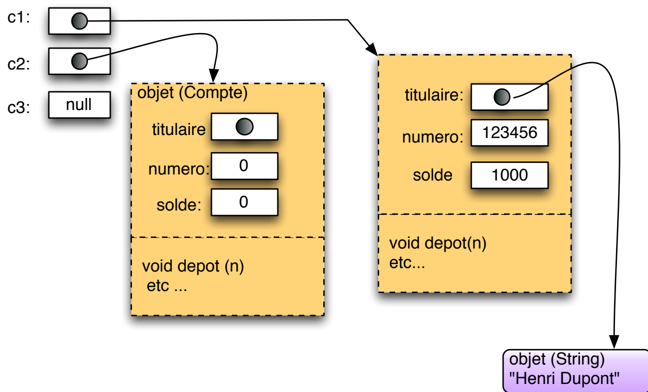
2. Aperçu des objets en mémoire

Les objets du programme après création



Après modification état de c1

```
c1.numero =  123456,
c1.titulaire = "Henri␣Dupont";
c1.solde = 1000.00;
```



3. Définir une classe (type)

Définir une classe : quelles composantes ?

1. Variables d'instance :
   → **données locales (état interne)** de l'objet ;
2. Méthodes :
   → **opérations** disponibles sur l'objet
   - ⇒ exécutés sur l'état interne d'un objet (état possiblement modifié).
3. Constructeurs : pour créer les objets (via `new`) ;

Exemple : définir une classe Compte

```
public class Compte{
  /* Variables d'instance */
  private String titutaire ;
  private int numero ;
  private double solde ;

  /* Méthodes */
  public double getSolde (){ return this.solde;}

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

  public void retrait(double montant)
    this.solde = this.solde - montant ;
  }
}
```

Déclarer des variables d'instance

Syntaxe

Accessibilite Type nomVar = valeurInitiale;

Exemple :

```
private String nomTitulaire;
private int numero;
private double solde = 0;
```

- création : si aucune valeur donnée ⇒ initialisées par défaut ;
- visibles par toutes les méthodes non statiques de la classe via la notation `this.nom-variable-instance`.

Comportement variables d'instance

- Définissent l'état interne de chaque objet :
  - initialisées à la création (défaut ou constructeur),
  - visibles par toutes les **méthodes de la classe** par notation `this.nom-variable-instance`.
- chaque objet peut lire/modifier ses variables (état) :

```
c1.depot(50); // +50 sur variable c1.solde
c2.depot(30); // +30 sur variable c2.solde
```

  - `c1.depot(50)` change variables instance de l'objet de l'appel (ici c1) ;
  - l'objet de l'appel est « l'objet courant »
- visibilité possible depuis l'extérieur de la classe (fortement déconseillée).

Méthodes dans les classes à objets : 2 sortes

- non statiques :

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

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

- statiques : n'ont pas accès aux variables internes.

Invoquer méthodes non statiques

Toujours appelées sur un objet :

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

- uniquement des méthodes de la classe ayant servi à créer l'objet ;
- agissent sur l'état interne (variables internes) de cet objet

```
c1.depot(50); // ajoute 50 dans c1.solde
c2.depot(30); // ajoute 30 dans c2.solde
c1.affiche(); // le solde courant dans c1
```

Variables d'une méthode non statique

```
public class Compte{
  String titutaire;  int numero;  double solde ;

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

3 sortes de variables :

- paramètres (`montant`),
- locales (aucune ici) ;
- `this` = objet courant : pour désigner les variables de l'objet courant
  `this.solde` ≈ *la variable solde de l'objet courant*.

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

this = objet courant

C'est l'objet sur lequel la méthode qui utilise `this` est invoquée..

Dans la méthode `depot` :

- `this` est l'objet sur lequel on est en train d'opérer un dépot ;
- `this.solde` $\Rightarrow$ variable solde de cet objet courant.

La valeur de `this` change à chaque exécution de la méthode `depot`.

Exécution d'une méthode utilisant this

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

Exécution de l'appel `c1.depot(150)` ⇒

1. Valeur des variables pour l'appel à la méthode `depot` :
   - `montant` ↦ 150 ;
   - `this` ↦ `c1`, donc `this.solde = c1.solde`
2. Exécution du corps de la méthode `depot` :

   `this.solde = this.solde + montant` ⇒
   `c1.solde = c1.solde + 150`

Que se passe-t-il si on exécute `c2.depot(70)` ?

Programme main : exemple

```
public class testBis  {
  public static void main (String [] arguments){
    Compte c1 = new Compte();
    Compte c2 = new Compte();
    c1.depot(100);
    c2.depot(60);
    c1.affiche();
    c2.affiche();
```

```
> java testBis
 Solde: 100
 Solde: 60
```

Via les appels aux méthodes non statiques, chaque objet agit, sur ses propres variables d'instance.

Méthodes accesseurs et modificateurs

- Accesseurs : fonctions ne modifiant pas l'état interne : utilisées pour l'observer, se contentent de renvoyer un résultat.

```
public double donneSolde(){
    return solde ;
}
```

- Modificateurs : modifient l'état interne.

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

Exemple modificateur : méthode depot

```
public class testBis  {
  public static void main (String [] arguments){
    Compte c1 = new Compte();
    c1.affiche();
    c1.depot(150);
    c1.affiche();
```

```
> java testBis
 Solde: 0
 Solde: 150
```

4. Objets dans les objets et appels inter objets

Objets dans les objets

L'état interne peut contenir toute sorte de variables, dont des objets.

Exemple : les personnes avec date de naissance.

- `Personne` contient variables nom, adresse, et date de naissance ;
- la date de naissance modélisé par un objet `Date` ;
- `Date` : variables jour, mois et année.

La classe Date

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

  public  void afficher(){
     Terminal.ecrireStringln(
      this.jour +"/"+ this.mois +"/"+this.annee);
  }
}

public class DateTest  {
  public static void main (String [] arguments){
    Date d2  = new Date();
    d2.annee=2000;  d2.jour =28;  d2.mois =2;
    d2.afficher();
  }}
```

```
> java Datetest
28 / 2 / 2000
```

La classe Personne

```
public class Personne{
  String nom;
  Date naissance;
  String numeSS;

  void afficher() { ... }
  boolean estMajeure() { ... }
  ...
}
```

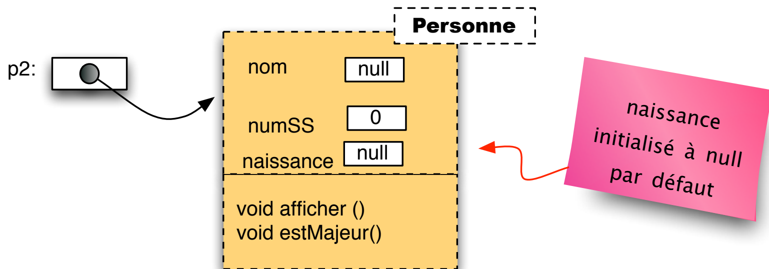
Variable d'instance `naissance` ⇒ **un objet de type** `Date`.

Initialisation en mémoire des objets "internes”

```
public static void main (String [] arguments){
    Personne p2 = new Personne();
    p2.nom="Martin";              // ok
    p2.naissance.jour = 18;   // ??
  }
}
```

Quelle est la valeur de `p2` en mémoire (dessin) ?
Peut-on exécuter `p2.jour = "Martin”;`
Quelle est la valeur de `p2.naissance.jour` ?
Peut-on exécuter `p2.naissance.jour = 18` ?

Initialisation par défaut des objets internes



Accès aux objets « internes »

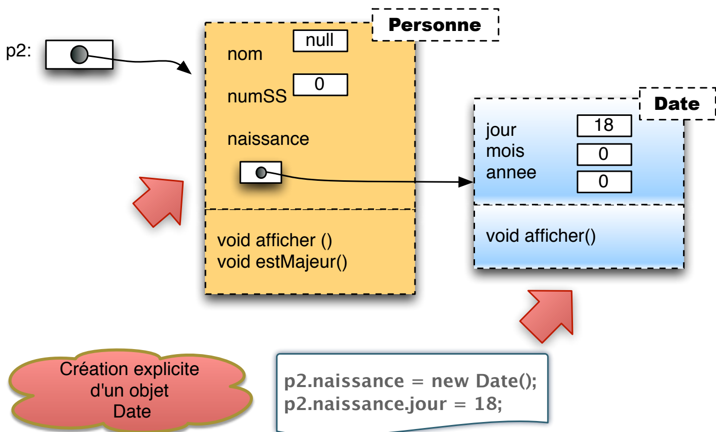
```
public class PersonneTest  {
  public static void main (String [] arguments){
    Personne p2 = new Personne();
    p2.nom="toto";              // ok
    p2.naissance.jour = 18;   // erreur
  }
}
```

- A la création de `p2`, `new` donne des valeurs par défaut (null) à `nom` et à `naissance` ;
- donc, `p2.naissance.jour` n'existe pas en mémoire !

```
> java DateTest2
Exception in thread "main" java.lang.NullPointerExceptio
        at DateTest2.main(DateTest2.java:99)
```

Initialisation explicite des objets internes

Elle est indispensable ! !



Initialiser les variables d'instance

```
public class PersonneTest  {
  public static void main (String [] arguments){
    Personne p2 = new Personne();
    p2.naissance= new Date();
    p2.naissance.jour = 18;
  }
}
```

Il faut initialiser p2.naissance :

```
    p2.naissance= new Date();
```

avant d'accéder aux champs jour, mois, et annee.

Appels sur objets internes

2 méthodes `afficher` : une dans `Personne`, l'autre dans `Date`.

```
public class PersonneTest  {
  public static void main (String [] arguments){
    Personne p2 = new Personne();
    p2.naissance= new Date();
    p2.naissance.jour = 18;
    Terminal.ecrireIntln(p2.naissance.jour);
    Terminal.ecrireString ( p2.nom + "␣␣date␣naissance:"
    p2.naissance.afficher();
  }
}
```

A quelle classe appartient la methode `afficher` invoquée ici ?

Appels sur objets internes (2)

La variable `p2.naissance` :

- est un objet de type `Date`, possédant une méthode `afficher`,
- c'est la méthode `afficher` de `Date` qui est invoquée.

```
Personne p2 = new Personne();
p2.naissance= new Date();
p2.naissance.afficher();
```

- l'appel `afficher` est effectué sur l' objet `p2.naissance`,
- la date dans `p2.naissance` est affichée ;
- comment s'exécute cet appel ?

Appels inter-objets

Considérons deux versions pour `afficher()` dans `Personne` :

```
class Personne {
 String nom;   int numSS;  Date naissance;

 void afficher1 (){...}

 void afficher2 (){...}
 }
```

Appels inter-objets (2)

```
void afficher1 (){
 Terminal.ecrireStringln("Nom:␣"+ this.nom);
 Terminal.ecrireStringln("NSS:␣"+ this.numSS);
 Terminal.ecrireStringln("Date␣naiss:␣"+this.naissance.
   +"/"+this.naissance.mois+"/"+this.naissance.annee);
}

void afficher2 (){
 Terminal.ecrireStringln("Nom:␣"+ this.nom);
 Terminal.ecrireStringln("NSS:␣"+ this.numSS);
 Terminal.ecrireStringln("Date␣naissance:␣");
 this.naissance.afficher();
}
```

Laquelle des 2 versions est préférable et pourquoi ?

Appels de inter-objets (3)

- `afficher2` utilise la méthode `afficher` déjà disponible dans la classe `Date` ;
- elle appelle cette méthode sur l'objet interne `this.naissance` ;
- `afficher1` "descend" dans les variables internes de l'objet `this.naissance` ;
- elle refait ce que ce que la méthode conçue pour les Dates fait déjà.

Appels inter-objets

La programmation objet se caractérise par les appels inter-objets ;
Il faut appeler des méthodes sur les objets internes, plutôt que descendre dans leurs variables !

5. Les constructeurs

Retour sur la construction d'objets

- Un objet de type `Date` est construit par `new Date()`
- `Date()` est un constructeur d'objets de type `Date`.

Constructeur

Sorte de méthode chargée de réserver de la place pour un nouvel objet, d'y stocker et initialiser les variables d'instance.
Il retourne l'adresse de l'objet crée.
On ne peut l'invoquer que via l'opérateur `new`

Exécution de `new`

Exécution de `new Date()` :

1. allocation d'espace mémoire pour un objet de type `Date`.
2. Initialisation de chaque variable d'instance définie dans la classe (valeur pas défaut si aucune valeur initiale).
3. retourne en résultat l'adresse de l'objet construit.

Les constructeurs d'une classe

- leur nom est celui de la classe (`Date`) ;
- ils n'ont pas de type de retour ;
- on ne peut les invoquer que via `new` ;
- si aucun constructeur est déclaré, Java crée un constructeur par défaut ;
- on peut définir ses propres constructeurs ou utiliser le constructeur par défaut.

Constructeurs par défaut

Si aucun constructeur n'est déclaré, un constructeur par défaut est fournit :

- sans paramètres ;
- initialise les variables d'instance aux valeurs initiales données par leurs déclarations ;
- ou à des valeurs par défaut si aucune valeur initiale est donnée.

Il ne peut y avoir qu'un seul constructeur par défaut.

Constructeurs déclarés

- On peut en définir plusieurs par classe.
- Utilisés pour décrire différente manières d'initialiser les objets construits.
- Ils peuvent ou non avoir des arguments.
- Si plusieurs constructeurs, le nombre (ou le type) de leurs arguments doivent être distinctes (leur signature).
- Autrement dit, on peut surcharger les constructeurs.

Attention : Dès qu'un constructeur est déclaré, le constructeur par défaut de cette classe cese d'exister.

Constructeurs pour la classe Date

Un sans argument, initialise la date à 1,1,1 ;
l'autre l'initialise à une date passée en argument.

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

  /* Initialise la date a 1/1/1 */
  public Date(){
    this.jour =1;  this.mois=1; this.annee =1;
  }
  /* Initialise selon les arguments */
  public  Date (int j, int m, int a){
      this.jour=j; this.mois = m; this.annee = a;
  }
}
```

```
public class datetest4  {
  public static void main (String [] arguments){
    Date  d1  = new Date ();
    Date  d2  = new Date (2,12,2000);
    d1.afficher();
    d2.afficher();
```

```
> java datetest4
  1 / 1 / 1
  2 / 12 / 2000
```

5.1 Parenthèse : la surcharge

Parenthèse : la surcharge

Surcharge Un *opérateur* ou un *nom de méthode* est surchargé, si chaque utilisation/invocation peut correspondre à une sémantiques différente.

Surcharge d'opérateurs

Exemples d'opérateurs surchargés :

- tous les opérateurs arithmétiques :
  - `1/2` ⇒ sémantique : division sur les entiers ;
  - `1.5/2` ⇒ sémantique : division sur les nombres flottant à double précision ;
- tous les opérateurs + :
  - `1+2` ⇒ sémantique : addition sur les entiers ;
  - `1.5+2` ⇒ sémantique : addition sur les nombres flottant à double précision ;
  - `"ab"+2` ⇒ sémantique : concaténation de chaînes de caractères

Surcharge = même syntaxe pour plusieurs sémantiques.

Surcharge de méthodes

```
// Cherche un entier dans un tableau d'entiers
 static boolean cherche(int x, int [] t){
   for (int i=0; i<t.length; i++){
          if (x == t[i]) return true;
   }return false;}

// Cherche un caractere dans un tableau de caracteres
 static boolean cherche(char x, char [] t){
   for (int i = t.length-1; i >=0;  i--){
          if (x == t[i]) return true;
   }return false;}
```

- deux méthodes de même nom
- avec des comportements différents (sémantique)

Surcharge de méthodes (2)

On peut surcharger un nom de méthode à condition de ne pas introduire d"ambigüité lors d'un appel :

- le nombre et ou le type des paramètres doivent être à chaque fois différents, ou données dans un ordre différent ;
- on parle de signature de méthodes :

Signature de méthode

nom de la méthode + suite de types de ses arguments.

Signature d'une méthode

Signature de méthode

nom de la méthode + suite de types de ses arguments.

Voici 4 signatures différentes pour le nom de méthode `cherche`

- `cherche(int, int [])`
- `cherche( int [], int)`
- `cherche(char, char [])`
- `cherche()`

Règles pour surcharger une méthode

La surcharge est autorisé si la nouvelle définition de la méthode possède une signature différente de toutes celles déjà existantes.

Exemple de surcharge autorisée

```
static boolean cherche(int x, int [] t){ ...

static boolean cherche(char x, char [] t){ ...
```

La surcharge pour les deux méthodes `cherche` est autorisée car leurs signatures sont distinctes :

```
cherche(int, int [])

cherche(char, char [])
```

Exemple d'utilisation de la surcharge

```
public static void main(String[] args) {
  int [] ti = {1,3, 7};
  char [] tc = {'e', 'f', 'k', 'r'};
  boolean a = cherche('a', tc);
  boolean b = cherche(3, ti);
```

- Deux méthodes `cherche` : un pour les tableaux de caractères, l'autre pour les tableaux d'entiers.
- Comment le compilateur sait lequel exécuter lors d'un appel ?

$\Rightarrow$ On peut toujours determiner quelle méthode exécuter en examinant le type des arguments **effectifs** de l'appel.

Exemple de surcharge (suite)

```
public static void main(String[] args) {
  int [] ti = {1,3, 7};
  char [] tc = {'e', 'f', 'k', 'r'};
  boolean a = cherche('a', tc);
  boolean b = cherche(3, ti);
```

Pour savoir laquelle des 2 méthodes exécuter, le compilateur examine le type des paramètres effectifs dans chaque appel :

- appel `cherche('a', tc)` ⇒ méthode de signature `cherche(char, char [])`.
- appel `cherche(3, ti)` ⇒ méthode de signature `cherche(int, int [])`.

Retour : utilisation de constructeurs

```
public class datetest4  {
  public static void main (String [] arguments){
    Date  d1  = new Date ();
    Date  d2  = new Date (2,12,2000);
    d1.afficher();
    d2.afficher();
```

Comment sait le compilateur quel constructeur appliquer dans chaque cas ?

Il suffit d'examiner les arguments de chaque appel.

```
> java datetest4
  1 / 1 / 1
  2 / 12 / 2000
```

Exemple 1 : constructeur pour Personne

```
public class Personne{
  Date naissance;  String nom;
  /* Constructeur qui prend une date d*/
  public Personne(String n, Date d){
     this.nom = n; this.naissance = d;
  }

  /* Creation dans le main */
 public static void main (String [] arguments){
    Date d = new Date(2,6,1978);
    Personne p2 = new Personne("Pierre␣Martin", d);
    ...
 }
```

L'initialisation de la date de naissance est réalisé par le constructeur des personnes.

Un constructeur peut échouer

Si la date d'initialisation est incorrecte, l'objet n'est pas crée :

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

  public Date(){ this.jour =1; this.mois=1;this.annee =1
  public  Date (int j, int m, int a){
    if (m >0 && m<13 && j <=Date.longueurMois(m,a)){
        this.jour=j; this.mois = m; this.annee = a;
    }
    else { throw new ErreurDate(); }
  }
```

Remarque : la méthode `longueurMois` est *statique*.

Suite de Date

```
public int getAnnee(){ return this.annee;  }
public void setAnnee(int aa){ this.annee=aa; }
public  static boolean bissextile(int annee ){
  ...
 }
public static int longueurMois(int m , int a ){
 if (m == 1 || m== 3 || m==5 ||m== 7  || m==8|| m==10
    return 31;   }
 else if (m==2) {if (bissextile(a)){return 29;}
                 else {return 28;}}
 else {return 30;}
 }
```

Remarque : les méthodes `longueurMois` et `bissextile` sont *statiques*.

Exemple 2 : constructeur pour comptes avec historique

Un compte tient l'historique des opérations réalisées ⇒ initialiser toutes les variables d'instance.

```
public class Compte{
  String titutaire ;
  int numero ;
  double solde ;
  ArrayList<String> historique;

  /* Constructeur */
  public Compte(String t, int n, double s){
     this.titutaire=t; this.numero=n; this.solde = s;
     historique = new ArrayList<String>();
  }
```

La variable `historique` est un objet, initialisé par le constructeur.

La méthode `toString` est souvent employée en Java
Convertit les données d'un objet en String.

```
public class Date {
  ....
  public toString(){
      return (this.jour +"/"+this.mois+"/"+this.annee);

  public  void afficher(){
    Terminal.ecrireStringln(this.toString());
  }
}
```

Commen fonctionne la méthode `afficher` ?

Exemple 2 (2) : mettre à jour l'historique

Supposons que la classe `Date` possède une méthode statique `aujourdhui` qui retourne la date du jour courant.

```
public class Compte{
    ...
  void depot(double m){
      solde = solde + m;
      String op =(Date.aujourdhui.toString())+":␣DEPOT="+
      historique.add(op);
  }
```

Chaque opération sur le compte est enregistrée au format :
`date : DEPOT= montant`

Pouvez-vous expliquer ce code ?

Exemple 2 (3) : Afficher l'historique

```
public class Compte{
    ...
  void afficheHistorique(){
    for (int i=0; i < historique.size(); i++){
      Terminal.ecrireStringln(historique.get(i));
    }
  }
```

On pourra également ajouter des méthodes de recherche d'opérations sur une date, sur une période, etc.

6. Variables et méthodes statiques

Variables et méthodes statiques

En plus de variables d'instance et méthodes non statiques, une classe peut avoir :

- Variables statiques ⇒ variables partagées par tous les objets d'une même classe.
- Méthodes statiques ⇒ actions ne touchant pas à l'état interne des objets, ne connaissant `this`, ni les variables d'instance.

Variables statiques

- Déclarées au même niveau que les variables d'instance,
- mais précédées du mot clef **static.**
- Les variables statiques déclarées dans une classe **Nom-classe** sont accessibles de deux manières :
  - **depuis l'extérieur de la classe** :
    `Nom-classe.nom-variable-statique`
    ou `nom-objet.nom-variable-statique`
  - **depuis la classe** :
    `nom-variable-statique`

Exemple 1 : joueurs en ligne

On veut modéliser les joueurs d'un jeu en ligne afin de compter :

- le nombre de joueurs connectés,
- le nombre de fois que chaque joueur a joué

Quelles sont les variables d'instance, variables statiques et méthodes de la classe Joueur ?

Exemple 1(2) : modéliser un joueur

Variables d'instance (état interne)

- nom du joueur (propre au joueur) ;
- nombre de fois qu'il a joué (propre au joueur) ;

Méthodes non statiques (agissant sur l'état interne)

- un constructeur qui initialise ces variables,
- une méthode pour joueur,
- une méthode qui affiche le nombre de fois qu'il a joué.

Variable statiques (état partagé)

- compteur de tous les joueurs connectés sur le jeu

Exemple 1 (3) : La classe Joueur

```
public class Joueur{
   String nom;
   int aJoue = 0;
   static int nbEnJeu=0;

   public Joueur(String n){   // constructeur
       this.nom = n; this.aJoue=0; nbEnJeu++;
   }
   public void jouer(){
       this.aJoue++;
   }
   public void afficheNbeFoisJoue(){
     Terminal.ecrireStringln
        (this.nom+ "␣a␣joue␣"+this.aJoue+"␣fois.");
  }
```

Exemple 1 (4) : Un programme de test

```
public class TestJoueur1{
 public static void  main (String [] arguments){
   Joueur j1 = new Joueur("Pierre");
   Joueur j2 = new Joueur("Anne");
   j1.jouer();  j1.jouer();  j1.jouer();
   j1.afficheNbeFoisJoue();
   j2.afficheNbeFoisJoue();
   Terminal.ecrireStringln
     ("Nombre␣de␣joueurs␣connectes:␣"+ Joueur.nbEnJeu);
   Terminal.ecrireStringln
     ("Nombre␣de␣joueurs␣connectes:␣"+ j1.nbEnJeu);
  }
```

```
> java TestJoueur1
Pierre a joue 3 fois.
Anne a joue 0 fois.
Nombre de joueurs connectes: 2
Nombre de joueurs connectes: 2
```

Méthodes statiques

1. Ne peuvent pas agir sur l'état interne ne peuvent pas faire référence aux variables d'instances ni à l'objet courant (`this`).

2. Elles sont communes à tous les objets d'une classe $\Rightarrow$ elles appartiennent à une classe.

3. On les invoque via un nom de classe : `Joueur.afficheEnLigne();` et non pas via un objet.

Exemple 1(5) : Méthode afficheEnJeu()

But : afficher le nombre de joueurs connectés. Ne touche pas aux objets : c'est une méthode statique.

```
public class Joueur{
   String nom; int aJoue = 0;  static int nbEnJeu=0;

   public Joueur(String n){  // constructeur
      this.nom = n; this.aJoue=0; nbEnJeu++;
   }
   public void jouer(){
      this.aJoue++;
   }
   public void afficheNbeFoisJoue(){ ...
   }
   public static void afficheEnJeu(){
     Terminal.ecrireStringln
      ("Nombre␣de␣joueurs␣connectes:␣"+ Joueur.nbEnJeu);
  }
```

Exemple 1(6) : Nouveau programme de test

```
Joueur.afficheEnJeu();
Joueur j1 = new Joueur("Pierre");
Joueur j2 = new Joueur("Anne");
j1.jouer();  j1.jouer();  j1.jouer();
j1.afficheNbeFoisJoue(); j2.afficheNbeFoisJoue();
Joueur.afficheEnJeu();
```

```
> java TestJoueur1
Nombre de joueurs connectes: 0
Pierre a joue 3 fois.
Anne a joue 0 fois.
Nombre de joueurs connectes: 2
```

La variable `NbEnJeu` est consultée alors qu'aucun objet de la classe n'est encore crée....

Comment expliquer cela ?

Exemple 2 : autres méthodes statiques

```
class Date {
 ...
 static boolean estBissextile(int a){
    return ( (a%4 ==0) && (!a%100 ==0) || (a%400==0));
 }

 /* Calcule le nombre de jours d'un mois pour une annee
 static int longueurMois(int m, int a) {
    if  (m == 4 || m== 6 || m==9 || m==11) {return(30);}
    else if (m==2) {
       if (bissextile(a)) {return(29); else return(28);
    } else {return(31);}
 }
```

Ces méthodes n'utilisent que leurs paramètres.
Elles n'utilisent ni this, ni aucune variable d'instance.

Exemple 2 (2)

On pourra employer ces méthodes en dehors de tout objet de type `Date` :

```
int lm = Date.longueurMois(2,2000);
```

Ou, pour définir une méthode non statique qui passe une date au lendemain :

```
class Date {
  ...
  void passerAuLendemain() {
    if  (this.jour < longueurMois(this.mois,this.annee))
        { this.jour++;}
    else if (this.mois < 12)
        { this.jour= 1; this.mois++;}
    else
        {this.jour= 1;this.mois= 1; this.annee++;}
  }
```

7. Contrôler l'accès aux données, encapsulation.

Modifier l'accès aux données

L'état d'un objet est à priori accessibles depuis l'extérieur de celui-ci :

```
Compte c1 = new Compte("Martin",1, 100);
Compte c2 = new Compte("Dupond", 2,  50);
c2.solde = c2.solde + c1.solde;
c1.solde = c1.solde - 10;
```

Quel est le problème avec ce code ?

- les soldes de `c2` et `c1` changent sans que cela corresponde à des opérations de compte ;
- on n'en garde pas trace dans l'historique...

Conclusion

il est important de protéger l'état des objets de modifications incontrôlées.

Autoriser ou interdire l'accès aux composantes d'une classe ou paquetage,

depuis toutes les autres classes ou paquetages via les *modificateurs d'accès* :

- `public`,
- `private`,
- `protected`

Avec la syntaxe

```
<modificateur-acces> <declaration-variable-ou-methode>
```

Droits d'accès : quelques principes

- une classe décrit une catégorie d'objets, offrant certaines fonctionnalités.
  - Exemple : liste, permettant ajout, suppression, recherche d'un élément, calcul longueur, etc.
- on donne accès aux fonctionnalités de la classe,
- mais on cache la manière dont ces fonctionnalités sont implantées.

Droits d’accès : quelques principes (2)

- *on donne accès aux fonctionnalités* de la classe,
- mais *on cache la manière dont ces fonctionnalités sont implémentées*.
- Concrètement :
  - fonctionnalités (accessibles) = les méthodes de la classe
  - leur implantation (cachée) = les variables d’instance et leur représentation.
- ainsi, un programme qui **utilise** les fonctionnalités d’une classe ne devra pas changer si seule l’implémentation des fonctionnalités change.
- Exemple : on change la représentation interne des Comptes : on ne doit changer que les méthodes de Compte qui utilise cette représentation, et pas tout le programme.

L'encapsulation en programmation

Notion de programmation au sens large, bâtie sur 2 idées :

Encapsulation

- englober dans un même "conteneur" : données + opérations sur ces données. Ex : *objets, modules, packages*.
- protéger les données à l'intérieur d'un conteneur, en laissant uniquement certaines opérations y accéder.

Selon les langages ⇒ une ou un mélange des 2 notions.

Encapsulation en Java : modificateurs d'accès

Java : protection par accès restreint aux composantes d'une classe ou paquetage, via modificateurs d'accès :

Modificateur d'accès

Spécifie la visibilité d'une composante depuis tout autre composante :

- `private` : visible uniquement par les méthodes de la classe ;
- `protected` : visible uniquement par les méthodes de la classe et celles des classes dérivées ;
- `public` : visible par tous ;
- pas de modificateur : visible par toutes les composantes appartenant au même paquetage (ou même répertoire, si paquetage par défaut).

Encapsulation = contention + protection

Résumé du principe :

- tout objet devient
  - non seulement conteneur,
  - mais aussi frontière à ne pas franchir par les attributs de l'objet.
- seules les méthodes restent accessibles depuis les autres classes.

Avantages de l'encapsulation

- programmes + clairs :
  - données + leurs opérations logiquement reliés se trouvent dans une même entité du langage ⇒ un objet, un paquetage, un module ;
- programmes + sûrs :
  - pas de modification malencontreuse des attributs ;
  - si méthodes satisfont leurs "contrats" ⇒ état interne reste toujours cohérent.
    Ex : *si les méthodes enregistrent correctement les opération dans l'historique, l'état du solde après opérations, correspond bien à ce que dit l'historique*.
- programmes + faciles à débogguer : si un objet ne se comporte pas comme souhaité, problème est à chercher du côté des méthodes de la classe (et pas ailleurs).

Protéger l'état interne des `Comptes`

```
public class Compte{
  private int numero;

     ....
  }
```

- les variables déclarées `private` ne sont visibles que par les méthodes de la classe où elle sont déclarées ;
- seuls les objets de la classe Compte pourront y accéder ;

Conséquences de la protection des données

1. Initialiser variables protégées ⇒ *définir des constructeurs*.
2. Obtenir valeur variables protégées ⇒ définir des *accesseurs*

```
public class Compte{
  private int numero;
  private double solde;
  /* Constructeur */
  public Compte(int n, double init){
     this.numero=n; this.solde=init;}
  /* Acceseurs */
  public double getSolde() { return this.solde; }
  public double getNumero(){ return this.numero;}
  ...
 }
```

Règles à suivre dans ce cours

Initialisation et accès aux attributs

- Initialisation variables d'instance :
  - préférer les constructeurs,
  - ou donner valeurs initiales à la déclaration des variables :
- Protéger les attributs via les modificateurs d'accès :
  - y accéder uniquement via les méthodes (déclarer accesseurs et modificateurs nécessaires).

Les paquetages

Rôle des paquetages

- grouper sous un même nom et dans un même répertoire des classes et interfaces issues de plusieurs fichiers ;
- limiter l'accès aux composantes du paquetage.

Syntaxe

- pas de construction syntaxique englobant toutes les composantes ;
- déclaration d'appartenance au paquetage par composante dans 1ère ligne de son fichier :
  `package` *nom-du-paquetage* ;

Exemple 1 : paquetage pour les comptes

Toutes les classes et interfaces d'une application de gestion de comptes bancaires mis dans un paquetage *comptes*,

- chaque fichier de classe ou interface spécifie dans sa 1ère ligne :

```
package comptes;
```

- tous les fichiers sont (très souvent) réunis dans un sous-répertoire `comptes` (du nom du paquetage).

Exemple : paquetage pour les comptes (2)

```
/* dans le fichier Compte.java */
package comptes;
public class Compte{
  .....

/* dans le fichier Banque.java */
package comptes;
public class Banque {
 ...

/* dans le fichier CarteCredit.java */
package comptes;
public interface CarteCredit {
...
```

Sans déclaration de paquetage, les composantes font partie du *paquetage par défaut*, regroupant toutes les classes *d'un même répertoire*.

Utiliser une composante de paquetage : noms qualifiés

Utiliser une composante de paquetage depuis l'extérieur de celui-ci ?

- $\Rightarrow$ elle doit être déclarée `public`.
- une solution : utiliser son nom qualifié :

```
comptes.Banque b = new comptes.Banque();
```

`comptes.Banque` est le nom qualifié de la classe Banque.

Utiliser une composante de paquetage : import

- importer la composante :

```
import comptes.Banque;
...
Banque b =  new Banque();
```

- importer toutes les composantes contenues dans un paquetage :

```
import comptes.*;
Banque b =  new Banque();
Compte c = new Compte();
```

⇒ pas besoin des noms qualifiés par la suite.

Importations en début de fichier (juste après `package`).

8. Résumé.

Résumé

- Classe = patron d'objets + nom type, contenant :
  - variables d'instance : données locales, de préférence protegées ;
  - constructeurs (pour initialiser var. instance) ;
  - méthodes d'instance (+ éventuellement statiques).
- Objet = instance d'une classe :
  - créee via `new` + constructeur
  - possèdent données locales + méthodes d'instance.
- méthodes dans une classe :
  - d'instance (non statiques) : À invoquer sur les objets. Accèdent aux var. d'instance.
  - statiques : n'ont pas accès aux var. d'instance.

Demo : comptes dans Eclipse.