# Langages objets

héritage
(bloc 10)

M2 Pro CCI, Informatique
Emmanuel Waller, LRI, Orsay

# résumé des épisodes précédents

- découverte Java, prise en main environnement
- types primitifs, opérateurs et expressions, instructions de contrôle, débugage
- Tableaux, fonctions, objets (sans et avec fonctions)
- Imbrication des constructeurs tableau et objet
- Méthodes dynamiques

- Java sans objets (exactement comme C)
  - opérateurs et expressions, instructions de contrôle, fonctions (statiques)
  - types primitifs, tableaux, objets (struct + pointeur)
- concepts objets
  - méthodes dynamiques

# héritage

- But, principe, vocabulaire
- Fonctionnement
- Constructeur
- Réutilisation de code
- Subtilités, cas des méthodes statiques, subtilité fondamentale
- méthodologie de conception
- dérivations successives

# but

- éviter certaines duplications de code, au moyen de factorisations (structure et code) entre classes
  - en particulier : en conjonction avec redéfinition des méthodes dynamiques (prochain cours) : traitement plus simple et uniforme d'ensemble hétérogène
    - en particulier lors extension application
  - cas particulier : réutilisation de code
- exemple : on verra que le TD d'aujourd'hui, pour le même énoncé, fournit un code bien meilleur que TD précédent

# remarque

- l'héritage concerne exclusivement les membres des classes (champs et méthodes dynamiques)
- il ne concerne pas du tout les champs et méthodes statiques
  - Mais méthodes statiques fournissent mécanisme restreint présentant des similarités (vu ci-dessous)

# exemple

- Gestion très simple de points du plan : afficher, déplacer
- Version 0 : méthodes dynamiques, pas d'héritage (= matériel cours précédent) :
  - champs (dynamiques) : x, y
  - méthodes (dynamiques) : affiche (coordonnées), déplace
- Ex1.java
- Déroulement mémoire

# exemple

- on veut maintenant en plus gérer des points avce une couleur (colorés) :
  - champs : x, y, couleur
  - méthodes : affiche (coordonnées, sans la couleur), déplace, repeint
    - (on affichera la couleur plus tard)

# principe

- intuition : un point coloré est un cas particulier de point
- autrement dit : un point coloré est simplement un point, avec en plus une couleur et une fonction pour la modifier
- on va écrire cela en Java de la manière suivante :

- la classe Point comme ci-dessus :
  - x, y, affiche, déplace
- la classe PointColoré, en déclarant qu'elle possède :
  - tous les membres (champs et méthodes dynamiques) de Point (mais sans recopier leur définition)
  - Et en plus ses propres spécificités :
    - Champ : couleur
    - Méthode : repeint (en écrivant bien sûr sa définition entièrement)

# vocabulaire

- Point est la *classe de base* (ou sur-classe, ou super-classe) de PointColoré
- PointColoré *dérive* (ou étend, ou est la sous-classe) de Point
  - elle *hérite* des membres de Point (elle les a)
    - d'où le terme « héritage »
  - elle en possède en propre (qui n'existent pas dans Point)

# Notation graphique informelle

- Classes et signatures

Point | ...

|

PointColoré | ...

# exemple

- Points et points colorés : héritage : Ex2.java

# héritage

- But, principe, vocabulaire
- Fonctionnement
- Constructeur
- Réutilisation de code
- Subtilités, cas des méthodes statiques, subtilité fondamentale
- méthodologie de conception
- dérivations successives

# fonctionnement

- un objet PointColoré possède :
  - les champs (dynamiques) : x, y, couleur
    - tout objet de PointColoré possède ces champs, et on peut y accéder dans tout code
      - Évident pour couleur
      - Vrai pour x, y, même si pas explicites dans définition classe PointColoré
  - les méthodes (dynamiques) : affiche, déplace, repeint
    - dans tout code, on peut appeler ces méthodes sur tout objet PointColoré
      - Évident pour repeint
      - Vrai pour affiche, déplace, même si pas explicites dans définition classe PointColoré

- autrement dit il possède :
  - les champs
    - de sa super-classe : x, y
    - de sa classe : couleur
  - les méthodes
    - de sa super-classe : affiche, déplace
    - de sa classe : repeint
- autrement dit : il accède en outre aux membres de sa classe de base, exactement comme s'ils étaient définis dans la classe dérivée elle-même
- C'est ce mécanisme qu'on appelle l'héritage

## exemple

- revoir cela dans Ex2.java, fonction main

## pourquoi ça marche

- Parce que déplace n'utilise que les champs (mémoire) et méthodes (code) de Point
- or un PointColoré est en particulier un Point
- donc il possède (au moins) tous les champs et méthodes utilisés par déplace
- Donc déplace peut s'appliquer sur un PointColoré

## exemple

- Ex2.java : déroulement mémoire

cases

## héritage

- But, principe, vocabulaire
- Fonctionnement
- Constructeur
- Réutilisation de code
- Subtilités, cas des méthodes statiques, subtilité fondamentale
- méthodologie de conception
- dérivations successives

## constructeur

- constructeur ne s'hérite pas (rappel : méthode spéciale)
- mal (duplication) :

```
PointColore(int abs, int ord, int c) {
    this.x = abs; // duplique constructeur de Point
    this.y = ord; // duplique constructeur de Point
    this.couleur = c;    }
```

- parfait (aucune duplication)

```
PointColore(int abs, int ord, int c) {
    super(abs, ord); // appelle constructeur Point
                     // obligatoire : première instruction
    this.couleur = c;    }
```

## exemple

- Ex2.java : constructeur

## héritage

- But, principe, vocabulaire
- Fonctionnement
- Constructeur
- Réutilisation de code
- Subtilités, cas des méthodes statiques, subtilité fondamentale
- méthodologie de conception
- dérivations successives

# Rappel : un intérêt héritage

- Réutilisation de code (cf. but, début cours auj.)
  - si on a besoin de points colorés
  - et qu'on dispose déjà de la classe Point
  - alors :
    - au lieu de réécrire le code de Point dont on a besoin
    - on écrit juste PointColoré
  - avantage : inutile recoder méthodes de Point dont on a besoin
  - Utile même sans résolution dynamique surcharge
- Résolution dynamique surcharge (prochain cours, et cf. subtilité fondamentale, fin cours auj.)

- remarque :
  - pas concept objet : ancien, principe des bibliothèques
  - héritage permet cela très facilement et élégamment, donc code de qualité (maintenance efficace)

# exemple

- Réutilisation de code : affichage d'un point coloré
- Affichage de tous les champs d'un PointColoré :
- Naïf :
  - Print : x, y, couleur
  - Duplication avec affiche de Point
- mieux :
  - x, y : afficher de Point
  - + print couleur

- Remarque : on pourrait l'appeler affiche au lieu de affichePointColoré, mais aujourd'hui on ne veut pas gérer la résolution dynamique de la surcharge en présence d'héritage (prochain cours)
- Ex3.java

# héritage

- But, principe, vocabulaire
- Fonctionnement
- Constructeur
- Réutilisation de code
- Subtilités, cas des méthodes statiques, subtilité fondamentale
- méthodologie de conception
- dérivations successives

# plus généralement

- partout où un objet Point est attendu, un objet PointColoré convient (et est accepté), car il a « au moins » les membres d'un objet Point :

```
class Application/main
  Point p = new PointColore(3, 7, 7);
  System.out.println(p.x + " " + p.y);
```

- remarque : le compilateur ne peut être sûr en général que le contenu d'une variable Point est un PointColoré, d'où :

```
Point p = new PointColore(3, 7, 7);
System.out.println(p.couleur); // erreur compilation
```

- mais l'inverse n'est pas possible :

```
PointColore p2 = new Point(7, 4); // erreur compilation
```

  - en effet, new Point() ne crée pas le champ couleur qu'un PointColoré doit avoir, et qui est attendu par les codes qui manipulent un Point Coloré

# règle d'affectation d'une variable

- une variable Point, ou case de type Point d'un tableau, ou champ de type Point d'un objet, peut recevoir un objet :
  - de la classe Point (évidemment)
  - ou de n'importe quelle sous-classe de Point

## cas des méthodes statiques

- (Cf. remarque début cours aujourd'hui)
- une méthode statique n'est jamais héritée
- mais elle accepte des arguments d'une sous-classe

## exemple

- Méthode statique et sous-classe : Ex4.java

## Subtilité fondamentale

- Qu'a-t-on gagné avec l'héritage ?
- Champs dynamiques : factorisation de la déclaration
- Méthodes :
  - Héritage pur : rien gagné : tout peut se faire avce méthodes statiques
  - Résolution dynamique de la surcharge :
    - Impossible avec méthodes statiques
    - Or c'est ça le vrai but ultime visé (prochain cours)

## exemple

- Point et PointColoré sans méthodes dynamiques : Ex4bis.java

## héritage

- But, principe, vocabulaire
- Fonctionnement
- Constructeur
- Réutilisation de code
- Subtilités, cas des méthodes statiques, subtilité fondamentale
- méthodologie de conception
- dérivations successives

## méthodologie de conception

- on a vu la méthodologie pour l'extension d'une application existante : partir de Point et créer PointColoré
- Voyons maintenant la conception d'une application à partir de zéro
- énoncé (rappel) : on veut gérer :
  - des points simples avec leurs coordonnées x, y, les afficher et les déplacer
  - des points colorés avec leurs coordonnées x, y, et leur couleur, les afficher (avec leur couleur), les déplacer et les repeindre

## Méthodologie à partir de zéro

- écrire dans un premier temps toutes les classes sans héritage
- puis factoriser dans une sur-classe commune :
  - tous les membres communs
    - Champs
    - Parties de code communes
- on écrit une sous-classe pour chaque entité :
  - avec ses spécificités
- remarque : la sur-classe peut ne correspondre à aucune entité réelle (classe abstraite : vu bientôt)

## exemple

- 2 classes :
  - points simples : x, y, affiche(x,y), déplace
  - points colorés : x, y, couleur, affiche(x,y,couleur), déplace, repeint
- après factorisation :
  - sur-classe commune : x, y, affiche(x,y), déplace
  - sous-classe points simples : rien => on l'élimine
  - sous-classe points colorés : couleur, repeint, affichePointColoré = affiche de Point + print couleur
- résultat final :
  - Point : x, y, affiche, déplace
  - PointColoré sous-classe de Point : couleur, repeint, affichePointColoré

## exemple

- Rappel : Ex2.java

## héritage

- But, principe, vocabulaire
- Fonctionnement
- Constructeur
- Réutilisation de code
- Subtilités, cas des méthodes statiques, subtilité fondamentale
- méthodologie de conception
- dérivations successives

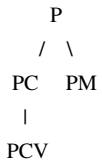
## dérivations successives

class Point { . . . }

class PointColoré extends Point { . . . }

class PointColoréValué extends PointColoré { . . . }

class PointMobile extends Point { . . . }




## héritage

- But, principe, vocabulaire
- Fonctionnement
- Constructeur
- Réutilisation de code
- Subtilités, cas des méthodes statiques, subtilité fondamentale
- méthodologie de conception
- dérivations successives

## héritage : ce qu'il faut se rappeler

- deux choses sont héritées :
  - champs (dynamiques)
  - méthodes (dynamiques)
- utilité : membres communs factorisés dans super-classe commune => évite duplication de code
  - Autre but : prochain cours, avec résolution dynamique surcharge
- une case déclarée de type Point accepte un objet Point ou de n'importe quelle sous-classe de Point

## héritage et redéfinition des méthodes dynamiques

- Delannoy chap. 8.1, 8.2 (?)