Partie X : Méthodes de *dispatching* orientées objet

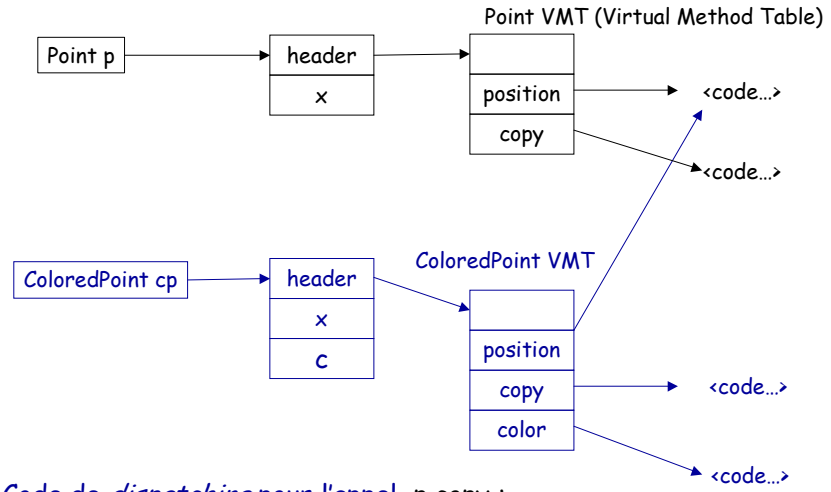
- Les langages OO supportent *la liaison dynamique* : un appel de méthode invoque un code qui dépend du type dynamique de l'objet receveur.
- A cause du sous-typage le type dynamique peut différer du type statique connu à la compilation.
- Problème : Comment réaliser le *dispatching* dynamique efficacement ?

Le cas de l'héritage simple

- Le *dispatching* dynamique est relativement simple à mettre en œuvre dans le cas de l'héritage simple, où chaque classe a au plus une super-classe.
- Exemple :

```
class Point {
  private int x;
  Point (int x) { this.x = x; }
  int position() { return x; }
  Point copy (int delta) {
    return new Point (position() + delta);
  }
}
class ColoredPoint extends Point {
  private Color c;
  ColoredPoint (int x, Color c) { super(x); this.c = c }
  Color color () { return c; }
  ColoredPoint copy (int delta) {
    return new ColoredPoint (x + delta, c);
  }
}
```

Graphique 1



Code de *dispatching* pour l'appel p.copy :
p.header.copy()

Dispatching de méthode

- Le schéma de *dispatching* par table de méthodes virtuelles (*virtual method table [VMT]* en anglais) est prédominant dans les situations d'héritage simple.
- Langages avec héritage simple :
  - Simula, Modula-3, Ada 95, Object Oberon, ...
- Langages avec héritage multiple :
  - Eiffel, Beta, C++, ...
- Langages avec sous-typage structurel :
  - Smalltalk, Cecil, Self, Pict, ...
- Hybrides – héritage + interfaces:
  - Java, Objective C

Techniques pour héritage multiple et hybrides

- Trampolines
- Tableaux de déplacement de lignes (*Row-displacement tables*)
- Antémémoire en ligne (*Inline caching*)

Exemple d'héritage multiple

```
class Point {
  private int x;
  Point (int x) { this.x = x; }
  int position () { return x; }
  Point copy (int delta) {
    return new Point (position() + delta);
  }
}

class Colored {
  private Color c;
  Colored (Color c) { this.c = c; }
  Color color () { return c; }
}

class ColoredPoint extends Colored, Point {
  ColoredPoint (int x, Color c) {
    Point.super (x); Colored.super (c);
  }
  ColoredPoint copy (int delta) {
    return new ColoredPoint (position() + delta, color());
  }
}
```

Trampolines

- Idée : Avoir des points d'entrée multiples pour les références, un par classe de base.
- Chaque point d'entrée a un champ entête (*header*) qui pointe vers une table de méthodes virtuelles.
- Quand on passe d'une sous-classe à une super-classe on met à jour le pointeur de l'objet pour qu'il pointe vers le point d'entrée correct.
- La redéfinition d'une méthode rend nécessaire de se déplacer d'un point d'entrée au début de l'objet englobant.
- Cela est réalisé par une méthode *trampoline* qui, une fois appelée, retourne la référence de l'objet englobant en soustrayant une valeur connue du point d'entrée.
- Cette technique a été utilisée pour Beta et C++.

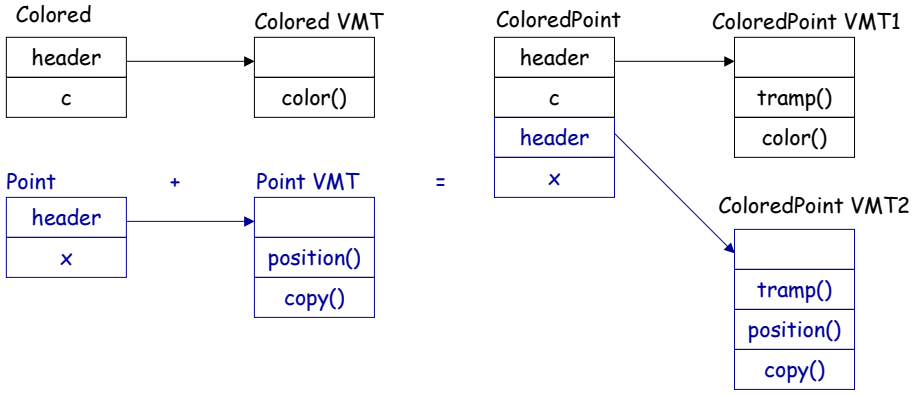
Trampolines (2)

- Avantages : des performances raisonnables même dans le pire des cas, les champs et les méthodes peuvent être hérités de façon multiple.
- Désavantages : surcoût des méthodes trampoline, les structures de données covariantes ne sont pas supportées. Par exemple en Java:

```
ColoredPoint[] <: Point[]
```

- Cela ne peut pas fonctionner avec les trampolines car il faudrait alors mettre à jour chaque pointeur dans le tableau.

Graphique 2



Point p; ColoredPoint cp;

p = cp ⇒ p = cp + 8;

cp = p ⇒ cp = p.tramp(p)

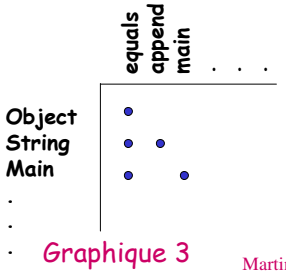
tramp(p) = p

tramp(p) = p - 8

Tableaux de déplacement de lignes *(row-displacement table)*

Conceptuellement, le problème du *dispatching* dynamique est le suivant :

- Étant donné un ensemble de classes et de méthodes , trouver le code correspondant à une classe et à une méthode données.
- Si l'on énumère les classes et les méthodes, cette tâche se réduit à une opération d'indexage simple dans un tableau à deux dimensions.



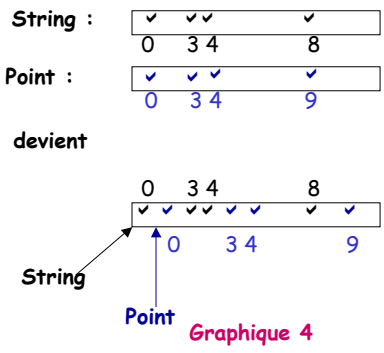
Graphique 3

- Problème : Cette table devient vite énorme. Application type : 500 classes, 2000 noms de méthodes uniques => 1'000'000 d'entrées.

Rendre le tableau plus compact

- Ce tableau à deux dimensions est occupé de façon clairsemée car chaque classe n'implante qu'un petit sous-ensemble de toutes les méthodes.
- On peut obtenir une meilleure utilisation de l'espace en imbriquant les lignes successives comme un ensemble de peignes.

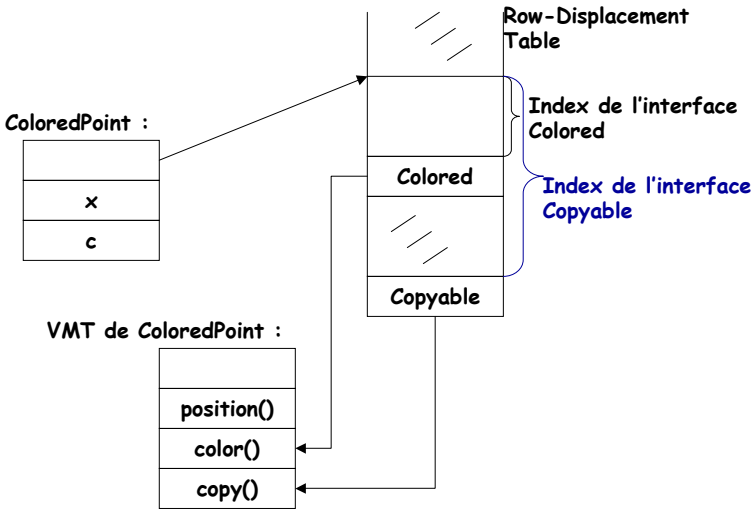
Rendre le tableau plus compact (2)



Graphique 4

- Variation pour Java : Indexer le tableau avec les classes et les interfaces plutôt qu'avec les classes et les méthodes. Une entrée du tableau pointe sur l'endroit dans une VMT où la méthode de l'interface est implantée.
- Cette technique a été utilisée dans certaines implantations très rapides de Java.

Graphique 5



Dispatching pour le tableau de lignes

- Code pour `I obj; ... obj.meth():`
  `obj.header[I.number].meth()`
- Comment savons-nous que l'entrée du tableau est utilisée pour la classe courante ?
- Nous n'avons pas besoin de le savoir, car Java est statiquement typé !
- Avantage du *dispatching* avec tableau à lignes (row table) : bonnes performances avec (cas moyen = pire des cas).

Considérations de *pipelining*

- Le *dispatching* par VMT et celui par tableau à lignes introduisent des bulles dans le pipeline (*pipeline bubbles).*
- On ne peut aller chercher de nouvelles instructions qu'après avoir calculé l'adresse dynamique de la méthode.
- Dans les processeurs modernes avec des pipelines profonds, cela peut s'avérer très coûteux.
- Exemple : pipeline de longueur 6 (+ writeback), 4 instructions en parallèle : 24 instructions manquées !
- Et les choses empirent avec des mots d'instructions très larges (processeurs VLIW).

Antémémoire (*Inline caching*)

- Observation : de nombreux appels vont toujours à la même classe.
- Idée : pour chaque instruction d'appel se rappeler le code qui a été utilisé à la dernière exécution de cette instruction.
- Immédiatement sauter vers ce code sans utiliser le *dispatching* dynamique.
- Au début du code cible il faut tester si l'on est bien dans la bonne classe.
- Si ce n'est pas le cas, retourner au schéma de *dispatching* dynamique classique, plus lent.
- Ce schéma peut offrir un gros gain si les appels vont toujours à la même classe.
- Sinon c'est une grosse perte !
- Exemples ?

Antémémoires et la JVM

- Le *inline caching* est utilisé pour implanter les appels aux méthodes d'interface dans l'interpréteur JVM de Sun.
- L'instruction `invoke_interface` a un champ qui contient la position relative de l'entrée de la méthode qui a été invoquée durant la dernière exécution de cette instruction (par rapport au début de la VMT).
- Quand `invoke_interface` est exécutée, il est d'abord vérifié qu'une entrée pour la méthode appelée se trouve bien à la position donnée.
- Sinon on recherche linéairement parmi toutes les méthodes de l'objet donné une méthode qui corresponde au nom et type de la méthode appelée.

Antémémoires polymorphes

- Le *inline caching* est une optimisation « tout ou rien » – c'est soit très rapide soit d'aucune aide (ralentit même les calculs).
- Meilleur compromis : garder un tableau des *n* dernières cibles.
- Si la cible courante est dans le tableau, sauter directement, sinon continuer avec le *dispatching* dynamique et ajouter la nouvelle cible dans le tableau.
- Si le tableau devient trop grand, revenir au *dispatching* dynamique total.
- Ce schéma est décrit dans la thèse de Urs Hölzle.
- Il est utilisé dans les implantations de Self, et Hotspot de Java.
- Avantage : on évite les bulles dans le pipeline ⇒ potentiellement de très bonnes performances, même meilleures que le dispatching simple avec VMT pour l'héritage simple.
- Désavantage : imprévisible : peut être (légèrement) pire que le dispatching avec VMT dans les mauvais cas.

Compilateurs JIT (Just In Time)

- L'interprétation du bytecode Java conduit à de faibles performances.
- La distribution des classes Java sous forme de code natif améliorerait les performances, mais au prix de la portabilité et de la sécurité.
- Les compilateurs JIT sont une façon de sortir de ce dilemme.
- Un compilateur JIT compile le bytecode en code natif soit au chargement, soit une fois que le code est exécuté un certain nombre de fois.
- En principe le code compilé JIT peut être plus rapide que du code natif compilé statiquement vu qu'il y a plus d'informations disponibles à l'exécution qu'à la compilation (P.ex. : Quelles méthodes sont appelées le plus souvent ? Combien de méthodes différentes sont invoquées par cette appel ?)

Compilateurs JIT (2)

- En pratique le code compilé JIT est généralement plus lent que du code natif car
  - le surcoût de la compilation ralentit l'exécution,
  - les optimisations des compilateurs JIT doivent aller vite et sont donc moins agressives que les optimisations des compilateurs de code natif.
- Compromis : compilateur lent et code généré rapide ou compilateur rapide et code généré lent ?
- Compromis lié : quand invoquer les compilateur JIT ?
  - Symantec: à la première exécution
  - Inprise : à la seconde exécution
  - Serveur Hotspot : après 10000 exécutions
  - Client Hotspot: après 1000 exécutions