Object-Oriented Programming

Second Checkpoint

Pr. Olivier Gruber

(olivier.gruber@univ-grenoble-alpes.fr)

Laboratoire d'Informatique de Grenoble

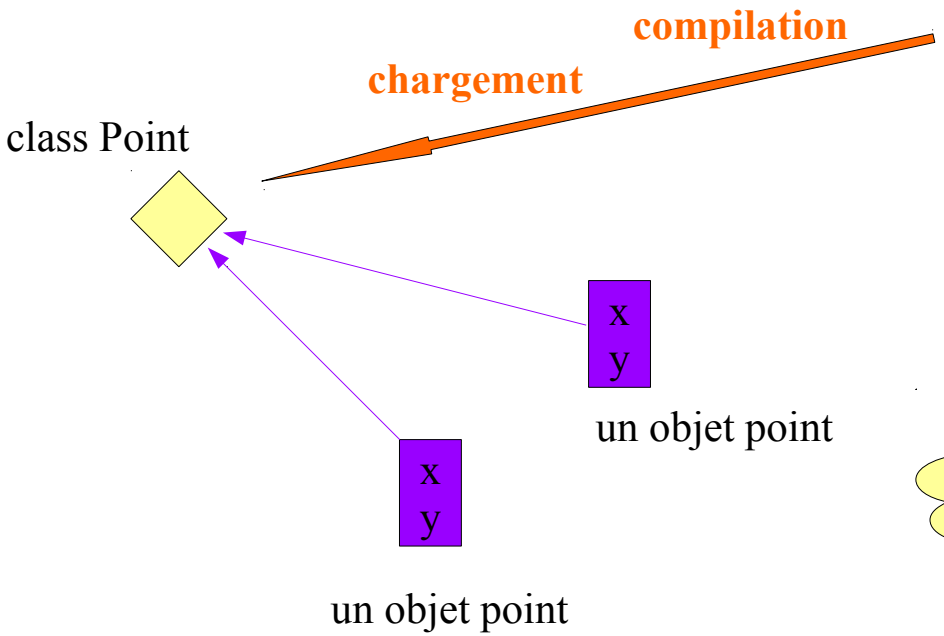
Université de Grenoble-Alpes

Aujourd'hui

- Un retour sur les concepts
  - La class Class…
  - Le ramasse-miette (garbage collector)
  -
- S'approprier les concepts

Le Paradigme Objet

- Chaque classe décrit ses instances
  - En termes de structure avec des champs – noms symboliques et types
  - En termes de comportement avec des méthodes – signatures
- Chaque objet est une instance d'une classe
  - Chaque objet connaît sa classe
  - Chaque objet est uniquement identifié




```
class Point{
  int x,y ;
  Point(int x, int y ) {
     this.x = x ;
     this.y = y ;
  }
  void translate(int dx, int dy ) {
     this.x += dx ;
     this.y += dy ;
  }
}
```

```
Point p = new Point();
Class c = p.getClass();
```

Le Paradigme Objet

- Exécution d'une méthode sur un objet (this)
  - Exécute les instructions
  - Utilise le contexte d'appel sur la pile
  - S'appuie sur les classes des objets
    - Pour accéder aux champs
    - Pour invoquer des méthodes

```
void translate(int dx, int dy ) {
   this.x += dx ;
   this.y += dy ;
}

Point p = new Point() ;
p.translate(2,4) ;
```

```
class Point{
   int x ;
   int y ;
}
```

2
3

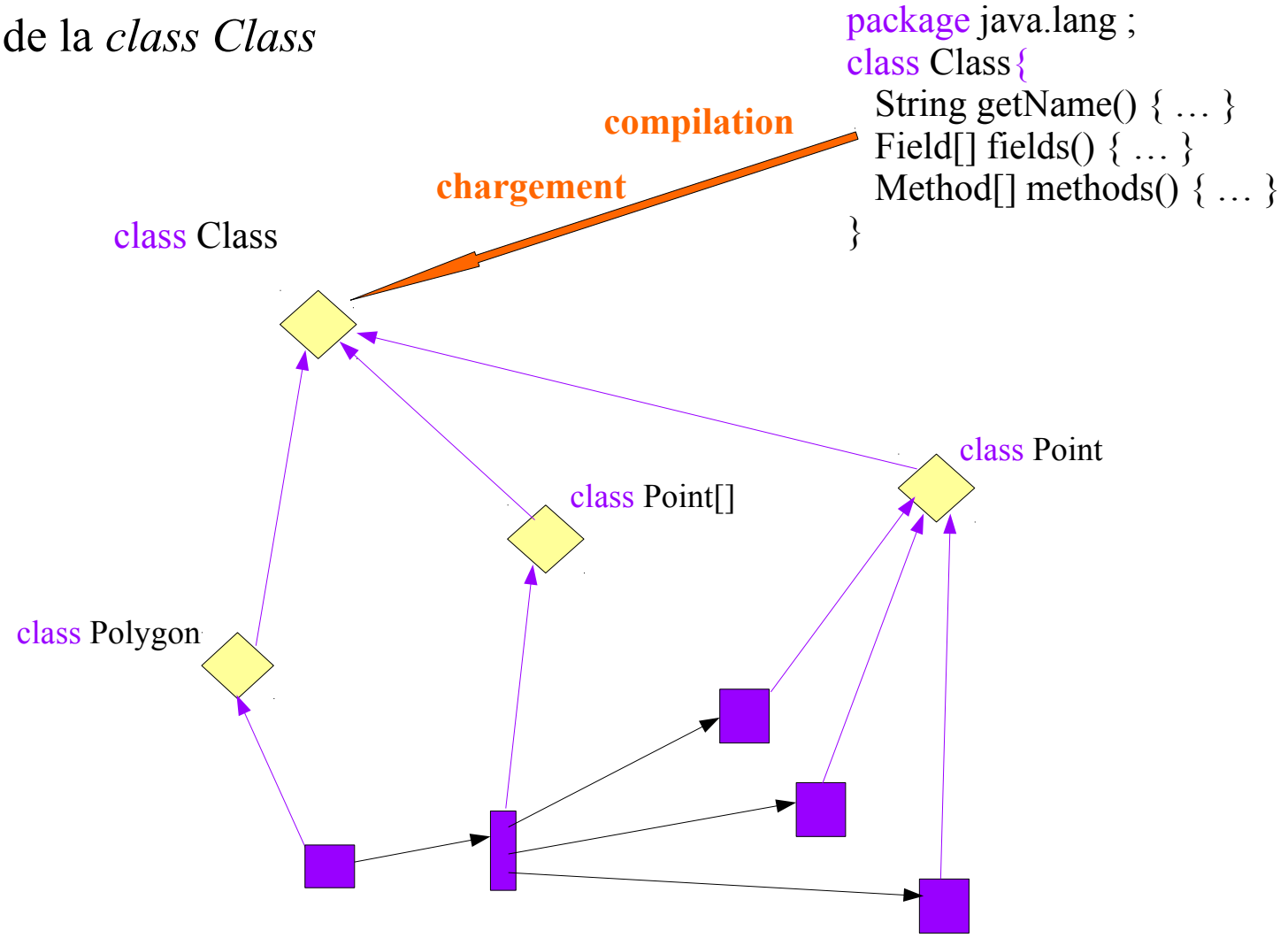
translate
this
dx=2
dy=4
???

Call Stack :

Le Paradigme Objet

- Tout est objet
  - Les classes sont donc des objets
  - Elles sont instances de la *class Class*

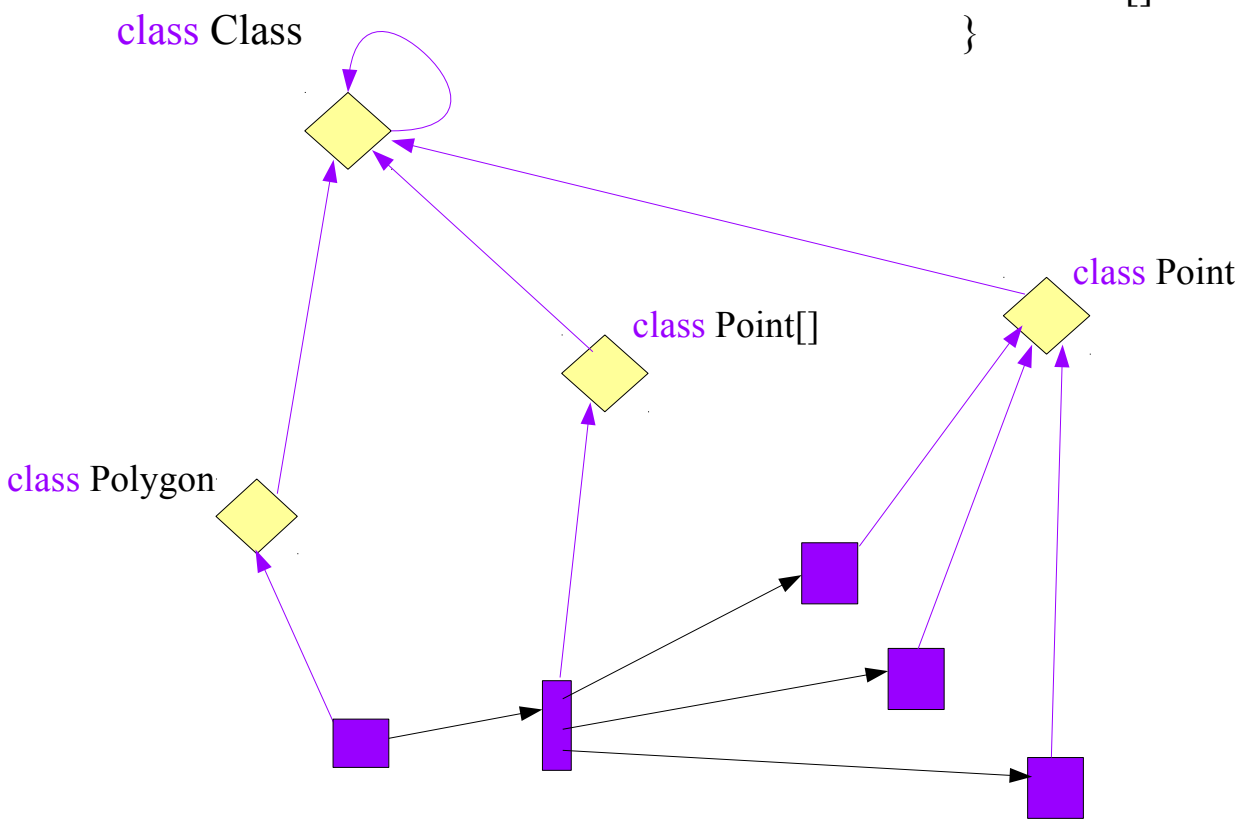
```
Point p = new Point();

Class c = p.getClass();

assert(c = Point.class);

Class cc = c.getClass();

assert(cc = cc.getClass());
```



Le Paradigme Objet

- Tout est objet
  - La *class Class* décrit donc ce qu'est une classe
  - Elle se décrit donc elle-même

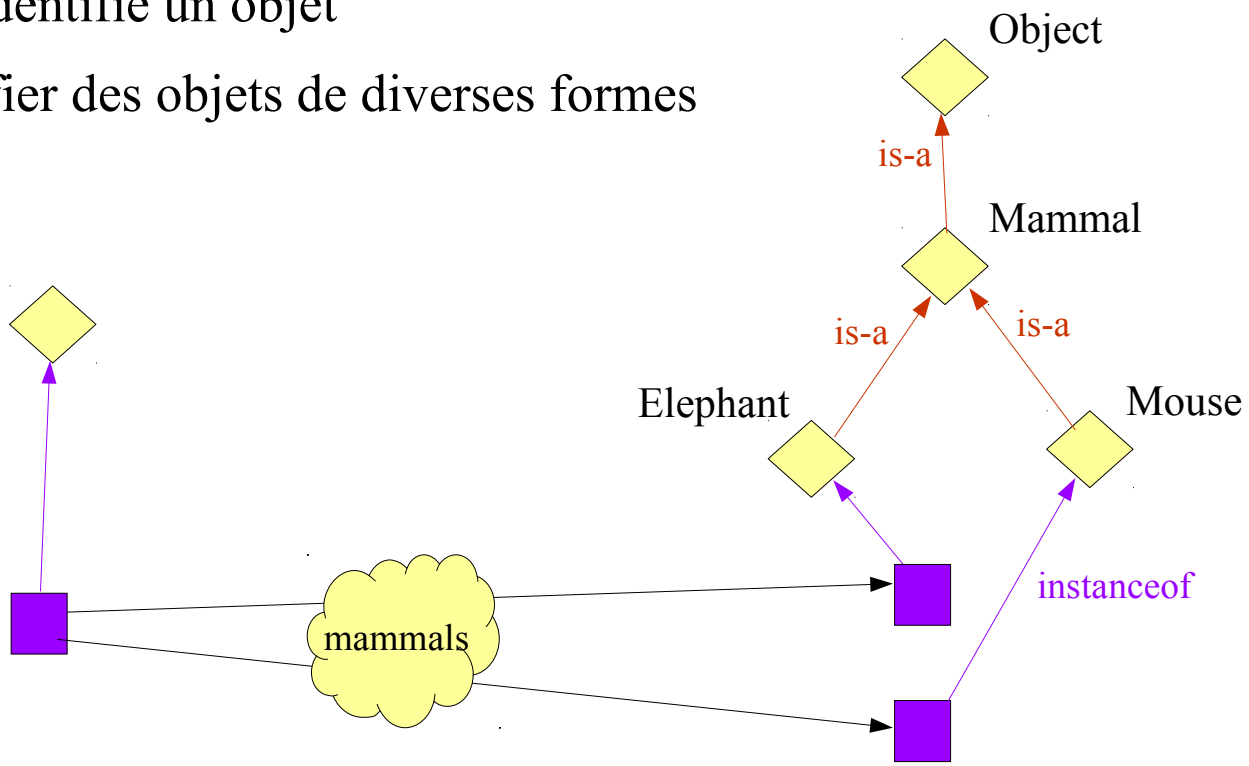
```
package java.lang ;
class Class{
   String getName() { … }
   Field[] fields() { … }
   Method[] methods() { … }
}
```



Polymorphisme & Objet

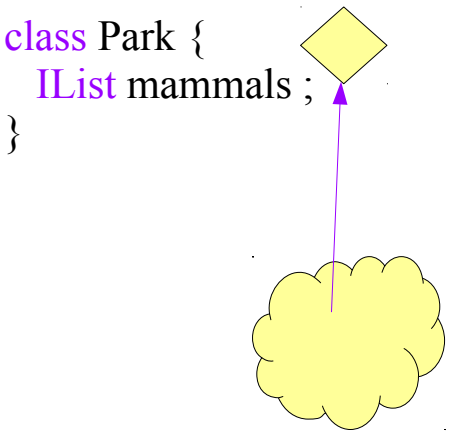
- Les objets sont-ils polymorphique ? **Non !**
  - Un objet n'a qu'une seule forme
  - Un objet ne se transforme pas d'une forme en une autre
- La programmation est-elle polymorphique ? **Oui !**
  - Une référence identifie un objet
  - Elle peut identifier des objets de diverses formes

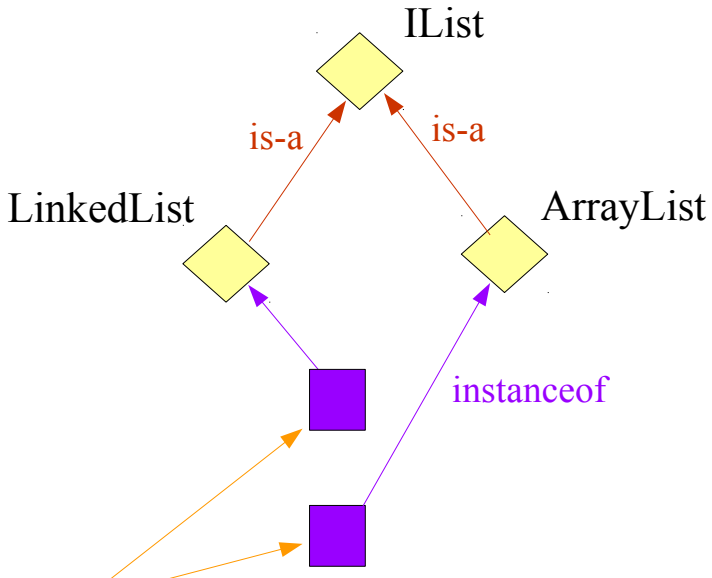
```
class Pair {
   Mammal x ;
   Mammal y ;
}
```



Polymorphisme & Objet

- La programmation est polymorphique
  - Une référence est typée – classe ou interface
  - Le type offre ***une vue*** de l'objet référencé
    - Un état avec des champs visible
    - Un comportement avec des méthodes

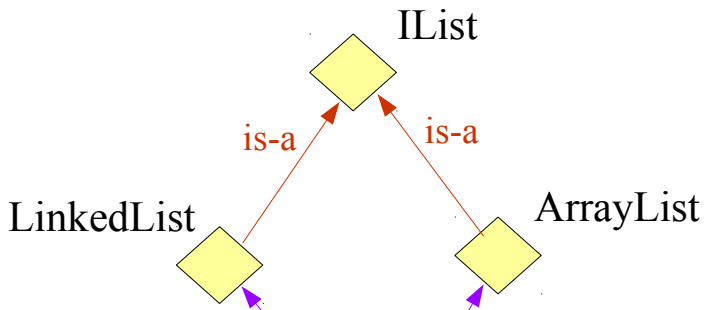




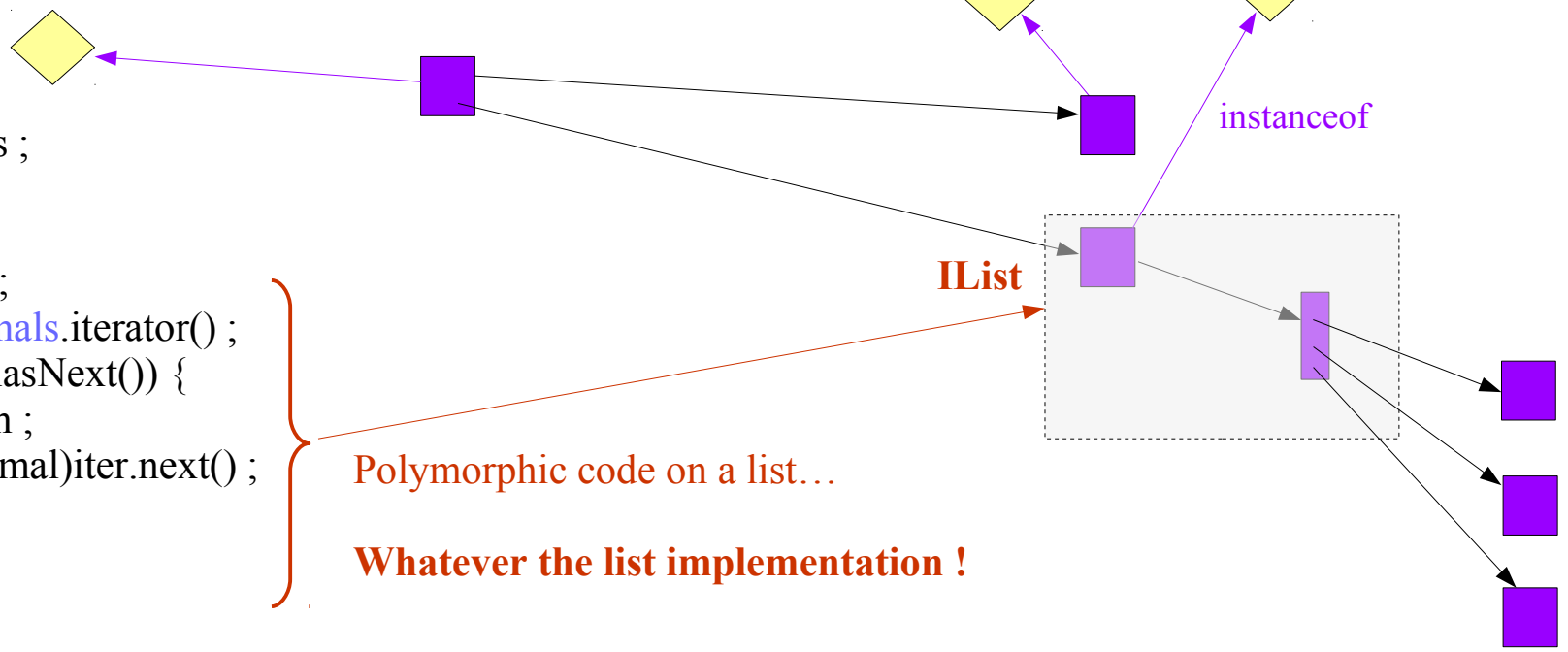
On sait que c'est une liste
Mais pas quelle implémentation ?

Polymorphisme & Objet

- La programmation est polymorphique
  - Permet d'interagir de façon générique
  - Avec des objets de *« formes différentes »*
  - Mais avec *« une vue commune »*




```
class Park {

  IList mammals ;

  void feed() {
     Iterator iter ;
     iter = mammals.iterator() ;
     while (iter.hasNext()) {
       Mammal m ;
       m = (Mammal)iter.next() ;
       ...
     }
  }
}
```



Polymorphic code on a list…

Whatever the list implementation !

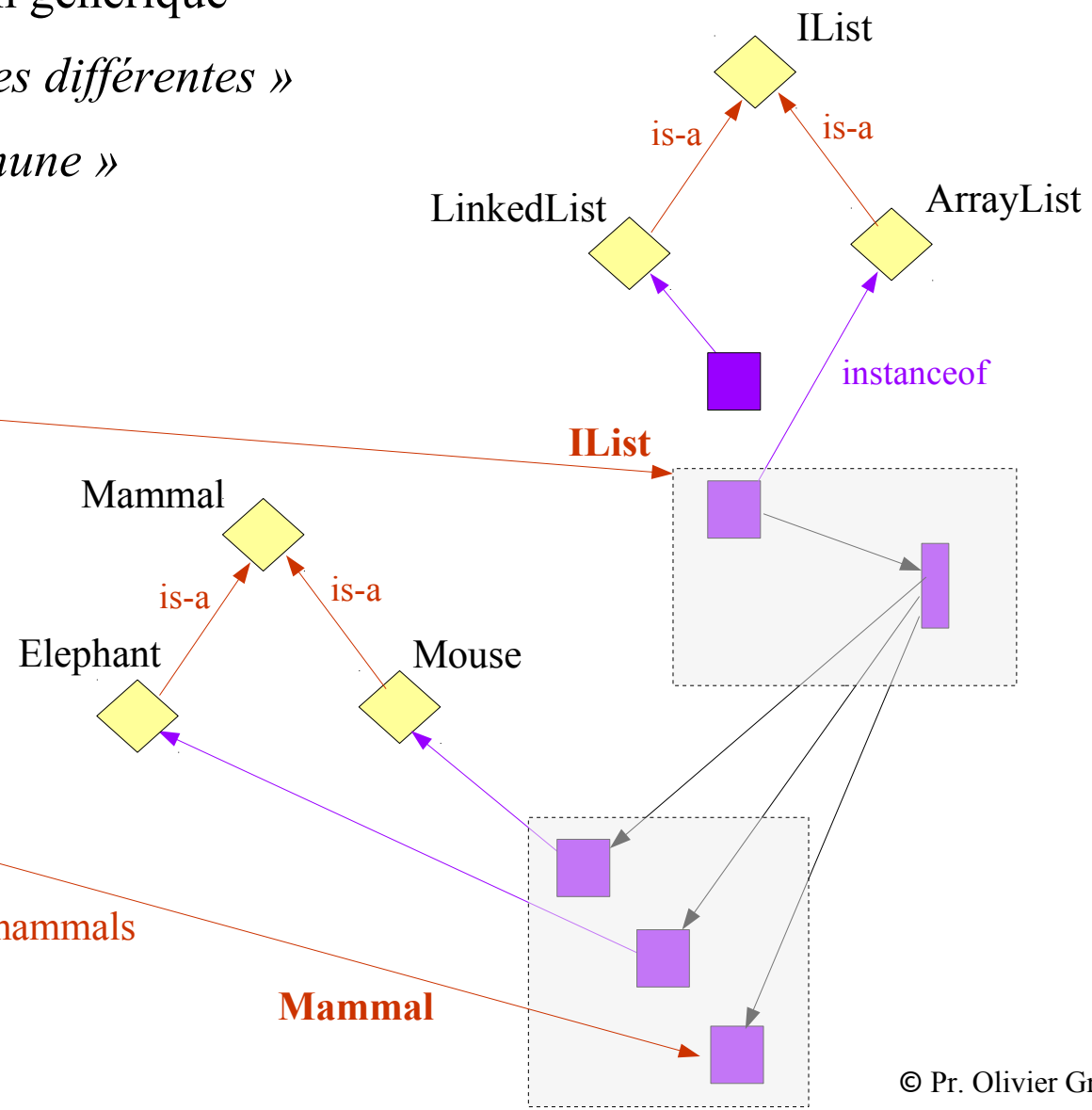
Polymorphisme & Objet

- La programmation est polymorphique
  - Permet d'interagir de façon générique
  - Avec des objets de *« formes différentes »*
  - Mais avec *« une vue commune »*

```
class Park {

  IList mammals ;

  void feed() {
     Iterator iter ;
     iter = mammals.iterator() ;
     while (iter.hasNext()) {
       Mammal m ;
       m = (Mammal)iter.next() ;
     }
  }
}
```

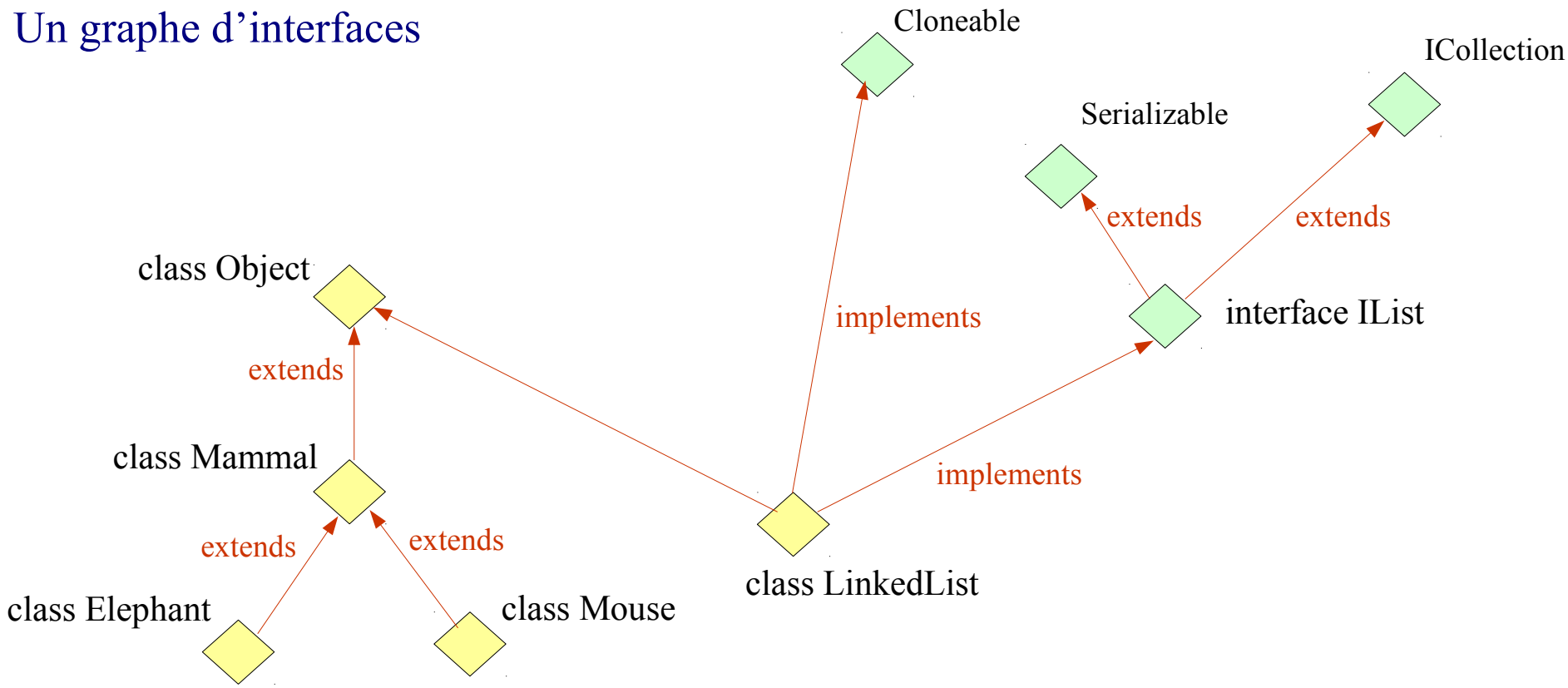
Polymorphic code on a list of mammals

Whatever the mammals !



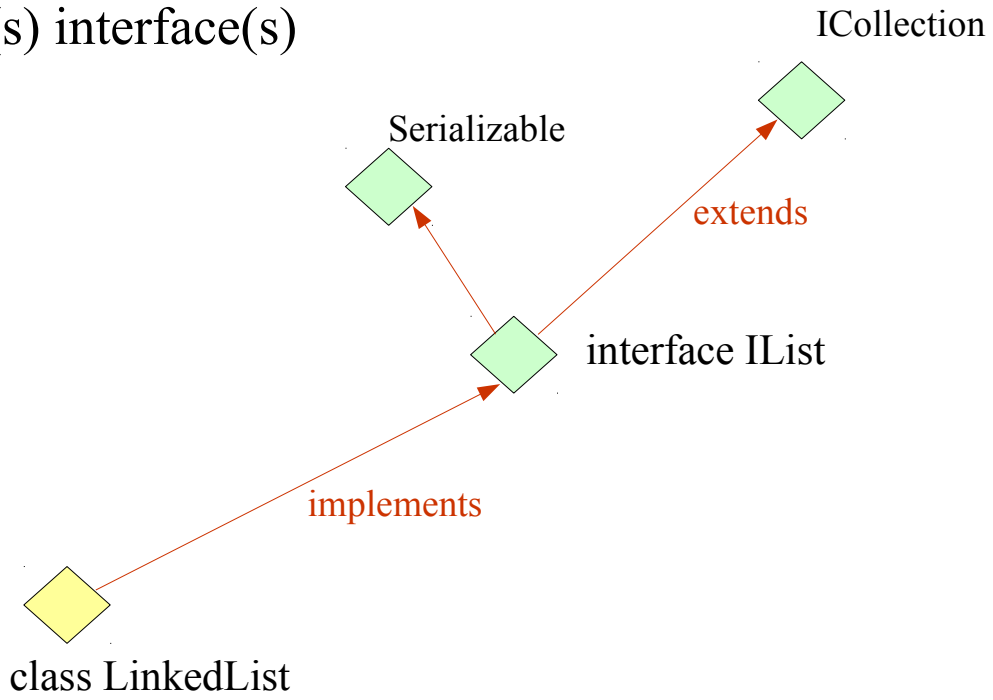
Classes versus Interfaces

- Un arbre de classes
- Un graphe d'interfaces



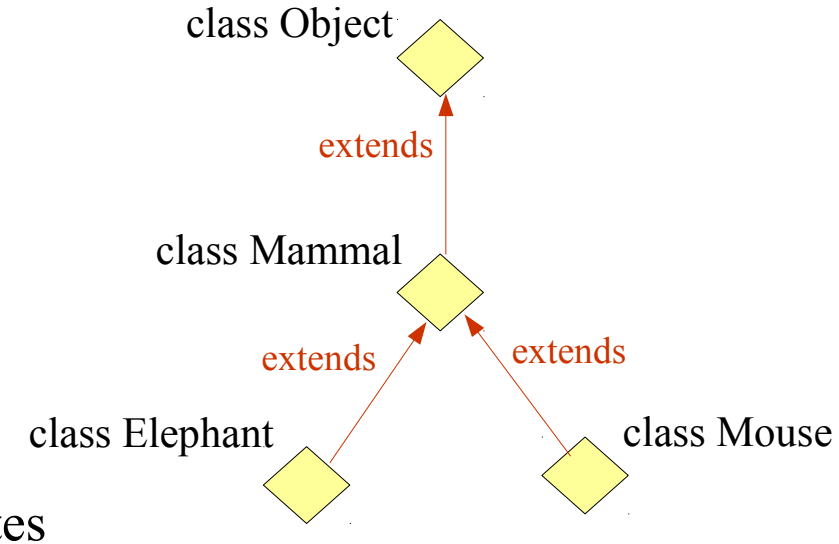
Classes versus Interfaces

- Une interface est abstraite
  - Elle ne fabrique pas d'objet
  - Elle définie seulement des *méthodes abstraites*
- Héritage comportementale seulement
  - Une classe implémente une ou plusieurs interfaces
  - Elle implémente toutes les méthodes de(s) interface(s)



Classes versus Interfaces

- Une classe – *comportement et structure*
  - Une classe est une fabrique d'objet
  - Elle définie champs et méthodes
  - Les méthodes ont du code associé
- Une classe peut-être abstraite
  - Auquel cas elle ne fabrique pas d'objets
  - Certaines de ses méthodes peuvent être abstraites



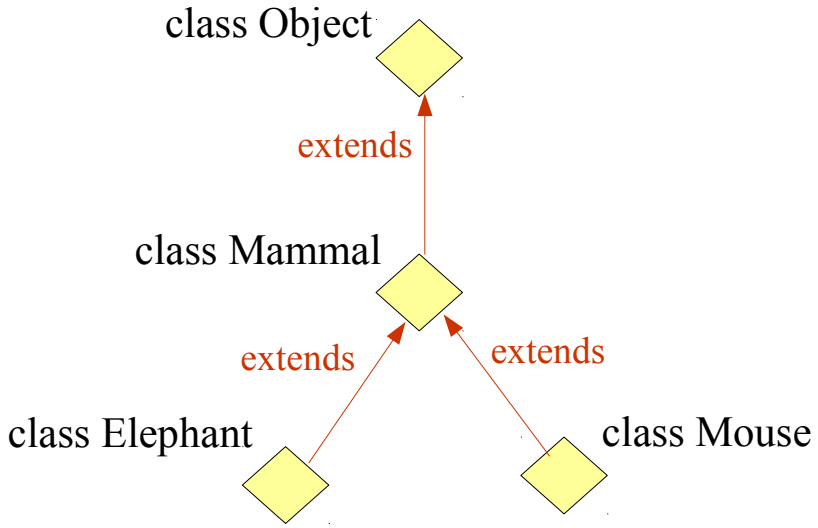
Exemple : le concept de mammifère existe mais en classe abstraite

cela n'aurait pas de sens d'avoir un objet instance de la classe Mammal

Aucun mammifère dans la nature n'est juste qu'un mammifère

Classes & Héritage

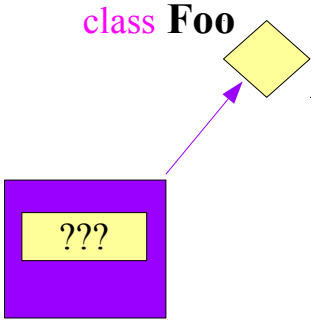
- Héritage structurel
  - Une classe définie des champs
  - Ils sont hérités par les sous classes
- Héritage comportementale
  - Une classe définie des méthodes
  - Elles sont héritées pas les sous classes
  - *Surcharges* et *redéfinitions* sont possibles



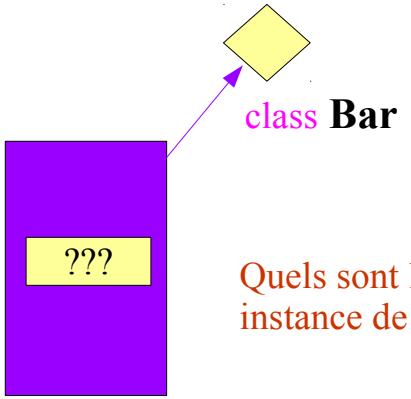
Héritage Structurel

```
package edu.ricm3;

public class Foo {
   int a;
   int b;
}
```

```
package edu.ricm5;

import edu.ricm3.Foo;

public class Bar extends Foo
   int b;
   String c;
}
```



Quels sont les champs dans un objet, instance de la classe Foo ?

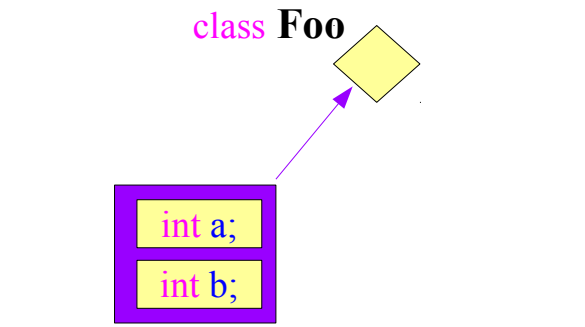


Quels sont les champs dans un objet, instance de la classe Bar ?

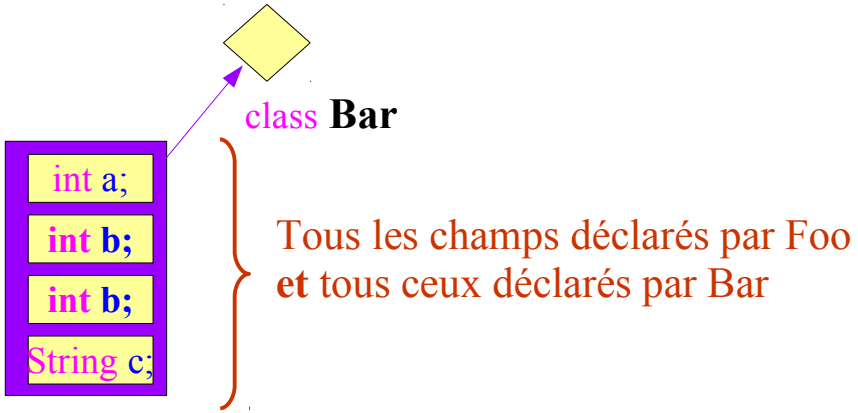
Héritage Structurel

```
package edu.ricm3;

public class Foo {
   int a;
   int b;
}
```

```
package edu.ricm5;

import edu.ricm3.Foo;

public class Bar extends Foo {
   int b;
   String c;
}
```



Tous les champs déclarés par Foo



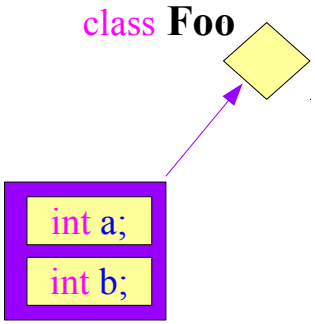
Tous les champs déclarés par Foo
et tous ceux déclarés par Bar

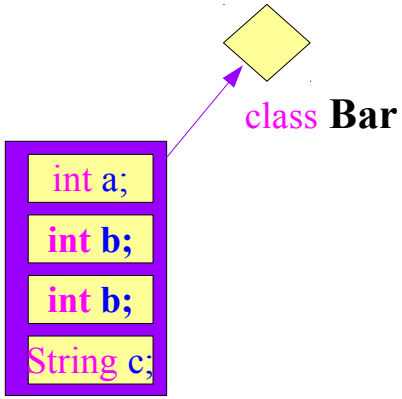
Héritage Structurel – Constructeurs

```
package edu.ricm3;

public class Foo {
  int a;
  int b;

  Foo() { }

  Foo(int x, int y) {
    a = x ; b = y ;
  }
}
```

```
package edu.ricm5;
import edu.ricm3.Foo;

public class Bar extends Foo {
  int b;
  String c;

  Bar(int x) {
    super(x,2*x) ;
  }
  Bar(int x, int y) {
    a = x ;
    b = y ;
    super.b = 2*x ;
  }
}
```

L'utilisation du mot clé super
Pour accéder aux champs cachés





Héritage Comportementale

```
package edu.ricm3;

public class Foo {
  int a;
  int b;

  Foo(int a, int b) {...}

  int foo(int x) { 1 }

}
```

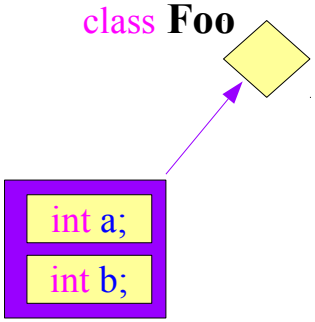
```
package edu.ricm5;

import edu.ricm3.Foo;

public class Bar extends Foo {
  int b;
  String c;

  Bar(String c) { ... }

  int bar(int x, int y) { 4 }
}
```

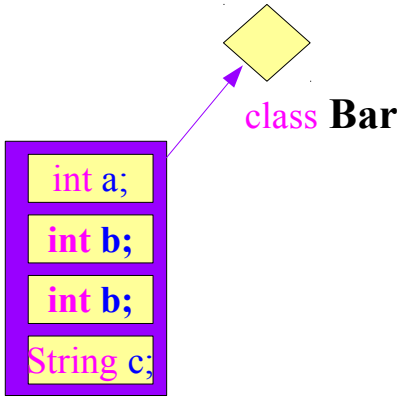
```
Foo f = new Foo(2,3);

f.foo(9) ;

Bar b = new Bar("toto") ;

b.bar(2,3) ;
b.foo(9) ;
```

inherited

```
f = b;
f.foo(9) ;
```

polymorphism

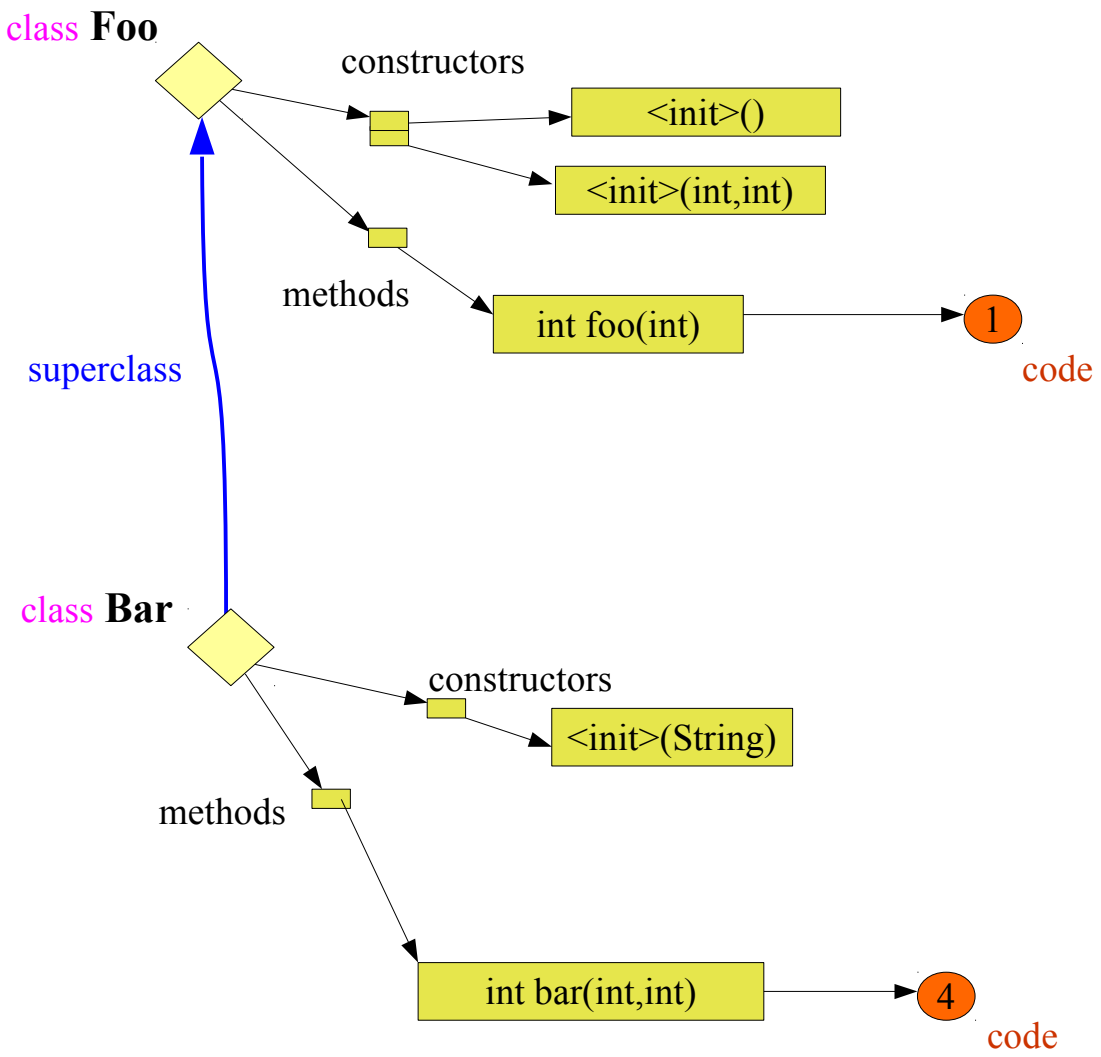




Liaison Tardive

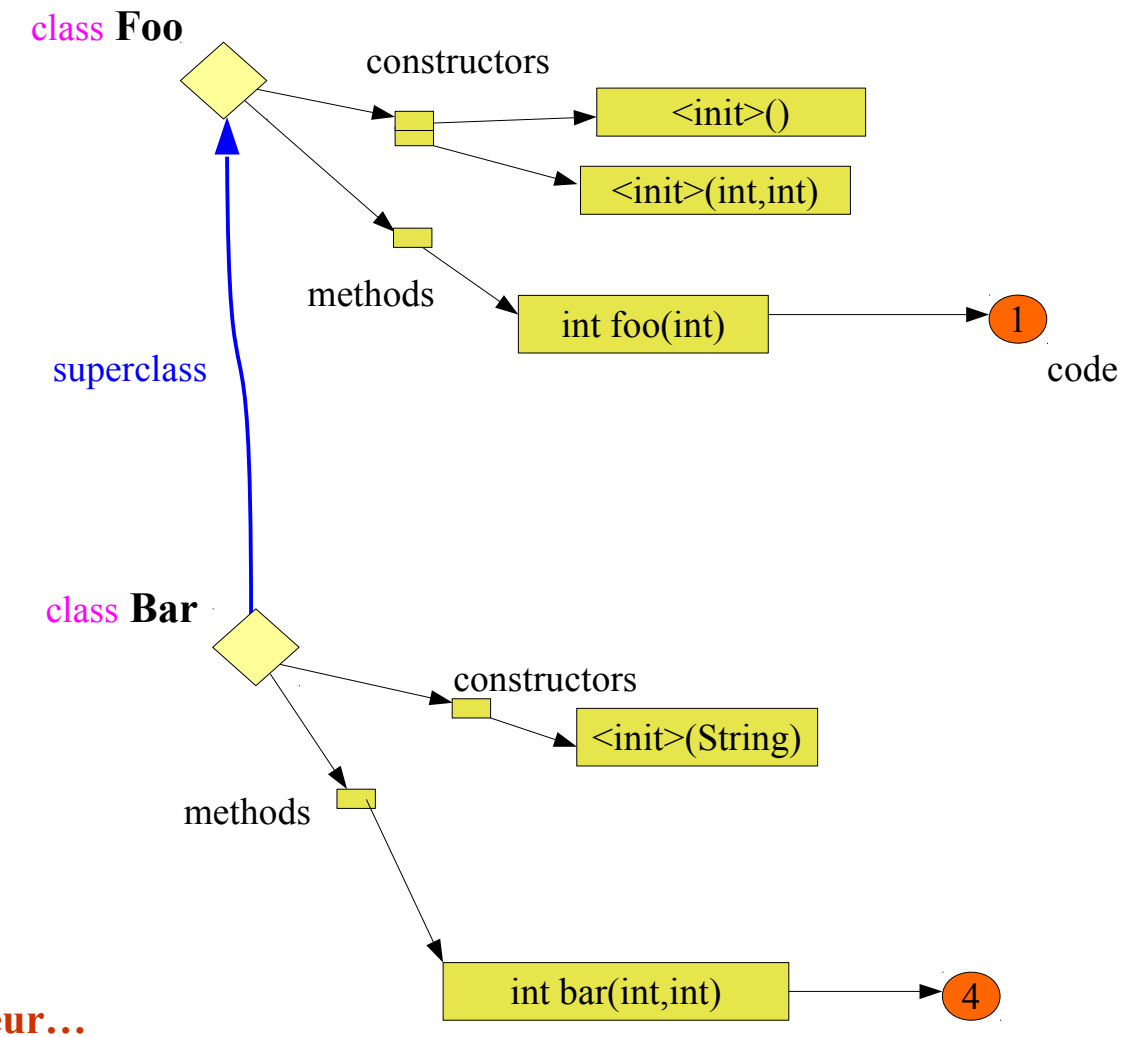
```
class Foo {
   int a;
   int b;
   Foo() {...}
   Foo(int a, int b) {...}

   int foo(int x) { 1 }
}
```

```
class Bar extends Foo {
   int b;
   String c;

   Bar(String c) { ... }


   int bar(int x, int y) { 4 }
}
```



Liaison Tardive

```
Foo f = new Foo(2,3);
f.foo(9) ;                    → 1

Bar b = new Bar("toto") ;
b.bar(2,3) ;                  → 4
b.foo(9) ;                    → 1

f = b;
f.foo(9) ;                    → 1
```



Quelle méthode ?

→ demandez à la classe du receveur…
→ recherche recursive sur les super-classes

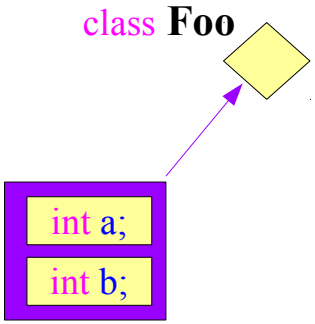
Héritage Comportementale

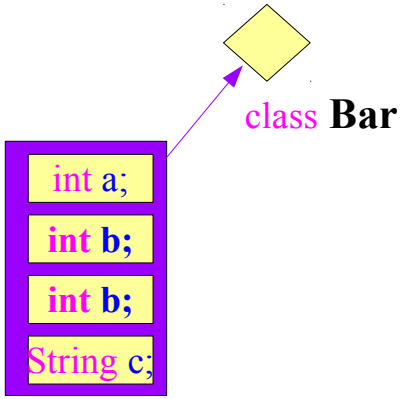
```
package edu.ricm3;

public class Foo {
  int a;
  int b;

  Foo(int a, int b) {...}

  int foo(int x) {...}

}
```

```
package edu.ricm5;

import edu.ricm3.Foo;

public class Bar extends Foo {
  int b;
  String c;

  Bar(String c) { ... }

  int foo(int x) {... }

  void foo(int x, int y) {... }

  int bar(int x, int y) { ... }
}
```

Override

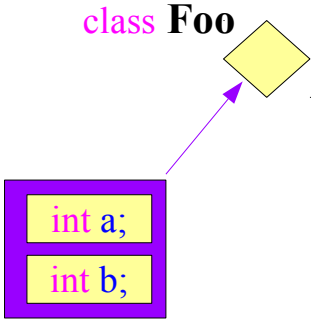
Overload



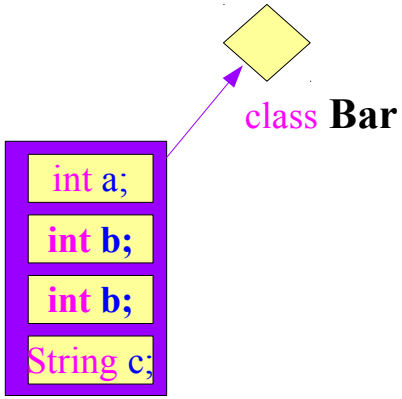


Héritage Comportementale

```
package edu.ricm3;

public class Foo {
  int a;
  int b;

  Foo(int a, int b) {...}

  int foo(int x) {...}

}
```




```
package edu.ricm5;

import edu.ricm3.Foo;

public class Bar extends Foo {
  int b;
  String c;

  Bar(String c) { ... }

  int foo(int x) {... }

  void foo(int x, int y) {... }

  int bar(int x, int y) { ... }
}
```




```
int a;
Foo foo;

foo = new Bar("toto") ;
```

```
a = foo.foo(3) ;
```

Quelle méthode ?

a) variable foo → typed as Foo

b) referenced object → an instance of Bar

c) method signature:

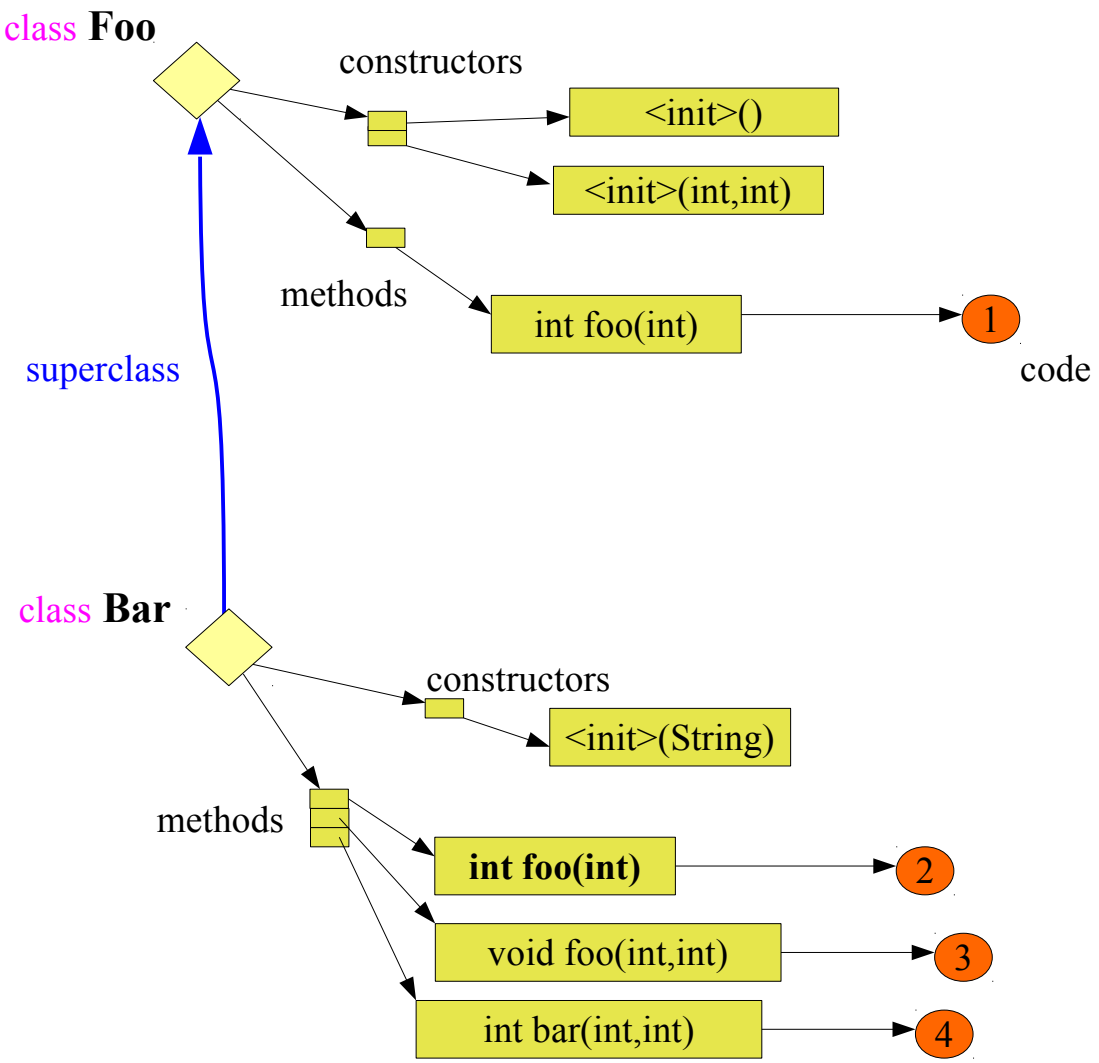
int foo(int)

→ demander à la classe du receveur

Liaison Tardive

```
class Foo {
   int a;
   int b;
   Foo() {...}
   Foo(int a, int b) {...}

   int foo(int x) { 1 }
}
```

```
class Bar extends Foo {
   int b;
   String c;

   Bar(String c) { ... }

   int foo(int x) { 2 }

   void foo(int x, int y) { 3 }

   int bar(int x, int y) { 4 }
}
```

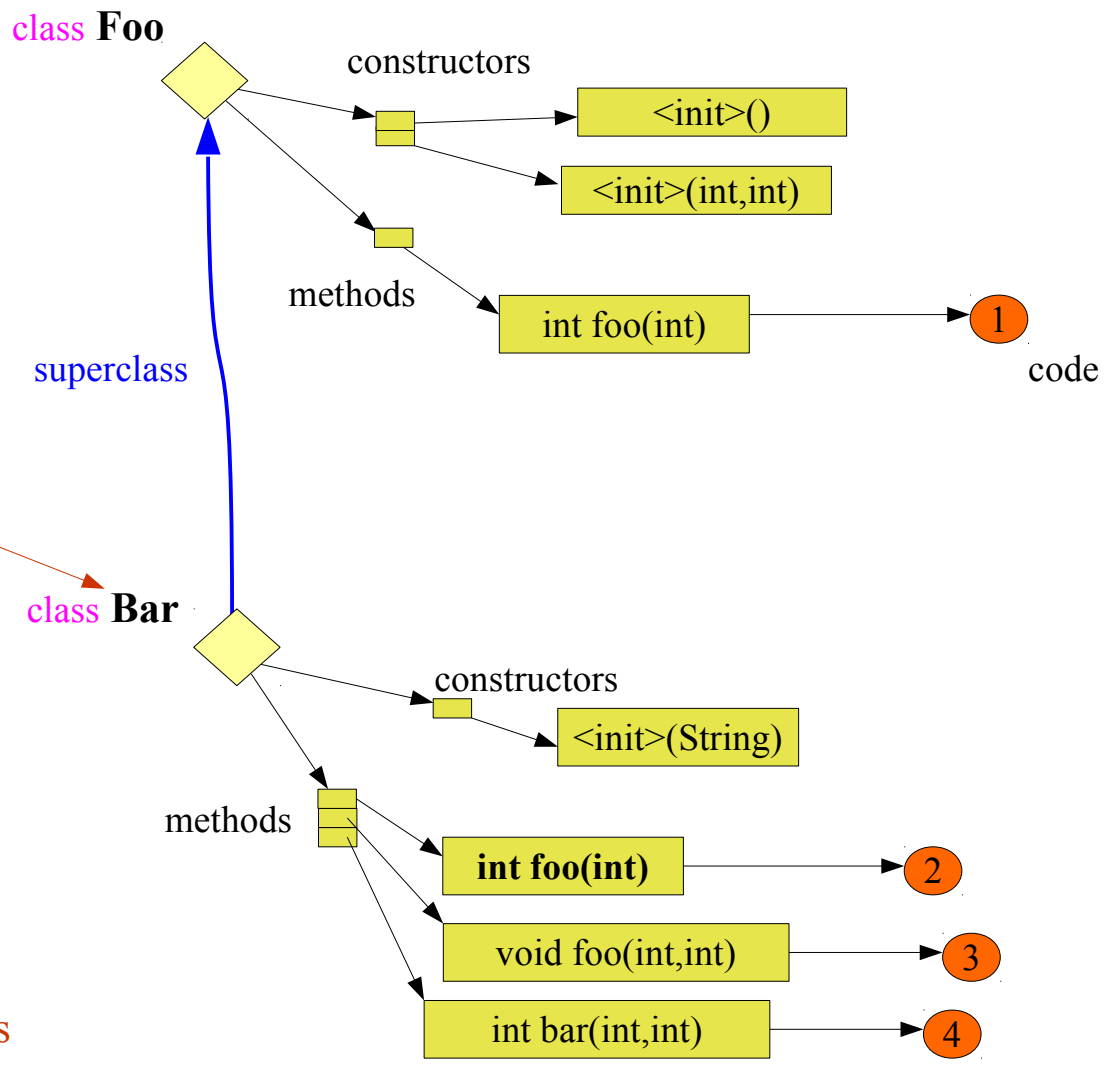


Liaison Tardive

```
int a;
Foo foo;

foo = new Bar("toto") ;
```

a = foo.foo(3) ;

Quelle méthode ?



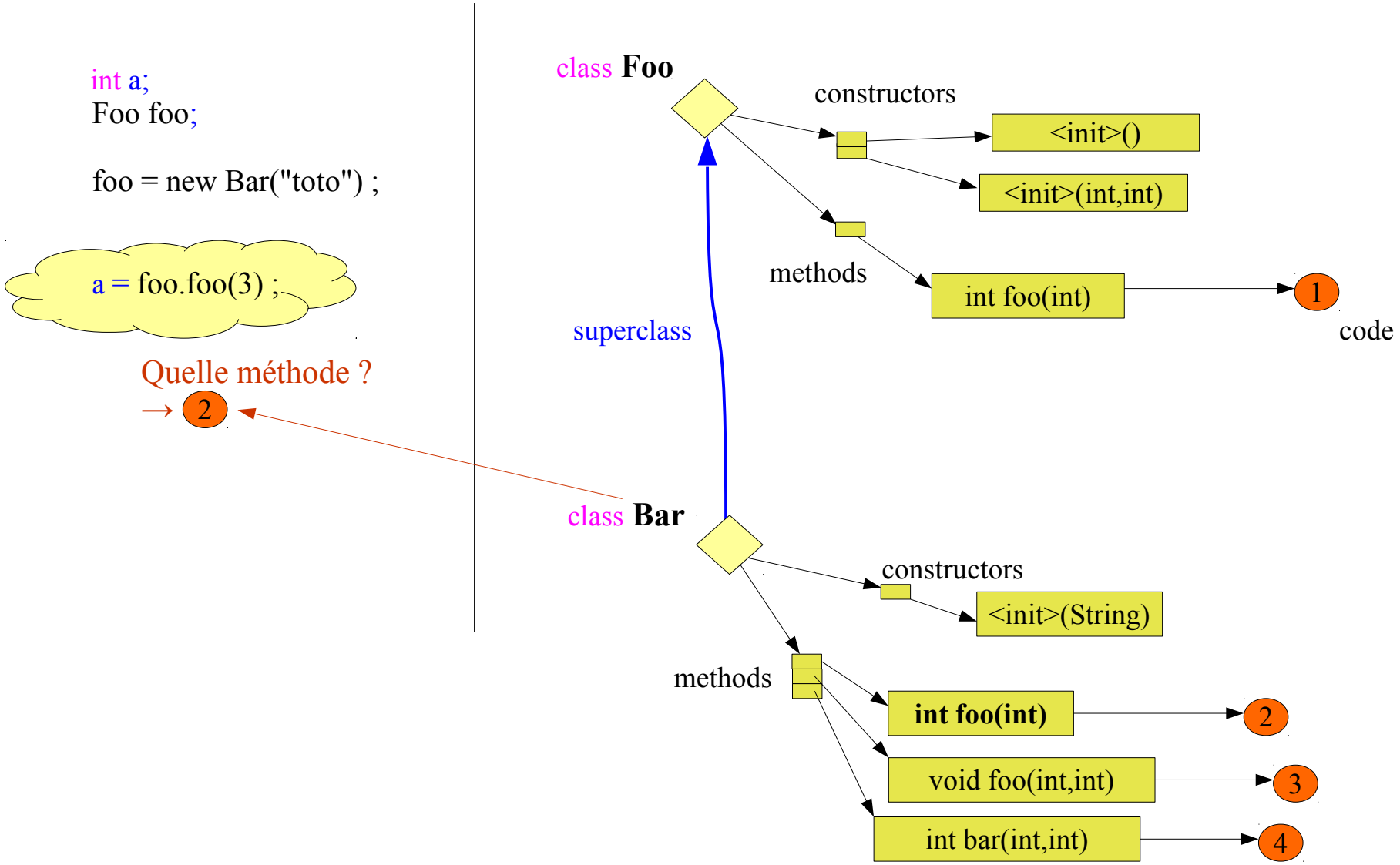
Liaison tardive :

Demander à la classe du receveur

Signature de la méthode : ***int foo(int)***

Recherche recursive sur les super-classes

Liaison Tardive



Liaison Tardive Polymorphique

- Processus pour sélection la méthode à invoquer
  - Invocation via une référence
    - Variable "***obj***"
  - La variable est typée
    - La variable "***obj***" est typée par la classe **Foo**
  - Le type permet de savoir que l'invocation est légale
    - La classes Foo a bien une méthode avec la signature "***int foo(int)***"
    - Donc le compilateur accepte : *obj.foo(3);*
  - *Mais le compilateur ne peut choisir un code statiquement*
    - A l'exécution, le code demandera à la classe du receveur "**this**"
    - Le receveur est l'objet référencé par la variable "***obj***"
  - Polymorphisme
    - Si le receveur est une instance de la classe Foo
      - Exécution du code 1
    - Si le receveur est une instance de la classe Bar
      - Exécution du code 2

```
public class Main {
  static Foo obj;
  static int test() {
    return obj.foo(3) ;
  }
}

public class Foo {
  public int foo(int x) { 1 }
}

public class Bar  extends Foo {
  public int foo(int x) { 2 }
}

public class Zorg  extends Foo {
  public int foo(int x) { 3 }
}
```

Liaison Tardive Polymorphique

- Sans liaison tardive...

```
Foo obj;
int a ;

    if (obj instanceof Zorg) {
      Zorg z = (Zorg)obj ;
      a = z.foo(3) ;
    } else  if (obj instanceof Bar) {
      Bar b = (Bar)obj ;
      a = b.foo(3) ;
    } else
       a = obj.foo(3) ;
```

```
public class Foo {
   public int foo(int x);
}

public class Bar  extends Foo {
   public int foo(int x);
}

public class Zorg  extends Foo {
   public int foo(int x);
}
```

Liaison Tardive Polymorphique

- Le langage C
  - Sans liaison tardive…
  - Sans type à l'exécution (pas de instanceof)
  - Sans surcharge pour les fonctions

```
static struct any* obj;
int a ;
  switch (obj→kind) {
  case ZORG :
      a = zorg_foo(&obj→u.z,3) ;
  case BAR :
      a = bar_foo(&obj→u.b,3) ;
  case FOO :
      a = foo_foo(&obj→u.f,3) ;
  }
```

```
struct foo { … }
struct bar { … }
struct zorg { … }
struct any {
 int kind ;
 union {
    struct foo f ;
    struct bar b ;
    struct zorg z ;
 } u ;
} ;

int foo_foo(struct foo* f, int x) ;

int bar_foo(struct bar* b, int x) ;

int zorg_foo(struct zorg* z, int x) ;
```

Choisissez le bon outil pour le travail à faire !
Et sachez bien l'utiliser !

Classes versus Interfaces

- Utiliser des interfaces
  - Quand on veut définir un comportement abstrait
  - Quand il n'y a pas d'héritage d'implémentation
  - Quand l'héritage multiple est nécessaire
  - Quand les classes qui implémenteront l'interface étendent déjà d'autres classes
- Utiliser des classes
  - Quand on veut de l'héritage d'implémentation
  - Ne pas oublier de qualifier les classes incomplètes comme abstraites
  - Quand l'héritage simple est suffisant