Mutabilité/Polymorphisme

Java Avancé

Rémi Forax

Plan

- Objet mutable/non mutable
- Sous-typage/Polymorphisme
- Règles de programmation

Mutabilité/Immutabilité

- Qu'affiche le code suivant ?

```
public class AnotherClass {
  public void anotherMethod() {
    Point p=new Point(2,3);
    System.out.println(
      p.toString());
    System.out.println(p.getX());
  }
}
```

```
public class Point {
  public Point(double x,double y) {
    this.x=x;
    this.y=y;
  }
  public int getX() {
    return x;
  }
  ...
  private double x;
  private double y;
}
```

Mutabilité/Immutabilité

- Vous etes sûr ?

```
public class AnotherClass {
  public void anotherMethod() {
    Point p=new Point(2,3);
    System.out.println(
      p.toString());
    System.out.println(p.getX());
  }
}
```

```
public class Point {
  public Point(double x,double y) {
    this.x=x;
    this.y=y;
  }
  public int getX() {
    return x;
  }
  public String toString() {
    String s=x+","+y;
    x=4;
    return s;
  }
  private double x;
  private double y;
}
```

Mutabilité/Immutabilité

- Quel est le problème ?
  - Lors de l'utilisation, on considère l'objet comme ne pouvant pas changer de valeur après création (on dit que l'objet est **immutable**)
  - Alors que le code de toString() modifie le champs x
- Choisir si est un objet est mutable où pas est une décision de design importante

Mutabilité/Immutabilité

- Java permet de garantir l'immutabilité en déclarant les champs **final**

```
public class Point {
  public Point(double x,double y) {
    this.x=x;
    this.y=y;
  }
  public String toString() {
    String s=x+","+y;
    x=4;  // erreur
    return s;
  }
  private final double x;
  private final double y;
}
```

- Rêgle: on met les champs final par défaut

Immutabilité

- final est pas suffisant !

```
public class Circle {
  public Circle(Point p,int radius) {
    ...
  }
  public void translate(int dx,int dy) {
    center.translate(dx,dy);
  }
  private final Point center;
  private final int radius;
}
```

- il faut que les objets référencés ne soit pas des objets mutables

Objets mutables/non mutables

- Un objet est mutable s'il est possible de modifier son état après sa création
- Sinon il est non mutable :
  - Champs **final**
  - Que des méthodes de consultation (getter), pas de méthode de modification (setter)
  - Certain objet peuvent être les deux, levant une exception ou non si l'on essaye de les modifier (cf les collections en Java)

Problème des objets mutables

- L'utilisation d'objets mutables peut casser l'encapsulation

```
public class Cat {
  private final StringBuilder name;
  public Cat(StringBuilder name) {
    this.name=name;
  }
  public StringBuilder getName() {
    return name;
  }
  public static void main(String[] args) {
    StringBuilder name=new StringBuilder("sylvestre");
    Cat cat=new Cat(name);
    name.reverse();
    System.out.println(cat.getName()); // ertsevlys
  }
}
```

La copie défensive

- Faire une copie lors de la création n'est pas suffisant !!!

```
public class Cat {
  private final StringBuilder name;
  public Cat(StringBuilder name) {
    this.name=new StringBuilder(name);
  }
  public StringBuilder getName() {
    return name;
  }
  public static void main(String[] args) {
    StringBuilder name=new StringBuilder("sylvestre");
    Cat cat=new Cat(name);
    cat.getName().reverse();
    System.out.println(cat.getName()); // ertsevlys
  }
}
```

La copie défensive (suite)

- La copie défensive doit être fait aussi lors de l'envoie de paramètre

```
public class Cat {
  private final StringBuilder name;
  public Cat(StringBuilder name) {
    this.name=new StringBuilder(name);
  }
  public StringBuilder getName() {
    return new StringBuilder(name);
  }
  public static void main(String[] args) {
    StringBuilder name=new StringBuilder("sylvestre");
    Cat cat=new Cat(name);
    name.reverse();
    cat.getName().reverse();
    System.out.println(cat.getName()); // sylvestre
  }
}
```

Utiliser un objet non-mutable

- Le plus simple est de souvent utiliser un objet non mutable (enfin quand on peut)

```
public class Cat {
  private final String name;
  public Cat(String name) {
    this.name=name;
  }
  public String getName() {
    return name;
  }
  public static void main(String[] args) {
    String name="sylvestre";
    Cat cat=new Cat(name);
    name.reverse();             // ne sert à rien
    cat.getName().reverse();    // d'ailleurs marche pas
    System.out.println(cat.getName()); // sylvestre
  }
}
```

Tableau toujours mutable

- La copie défensive doit être effectuée lorsque l'on retourne la valeur d'un champs

```
public class Stack {
  public Stack(int capacity) {
    array=new int[capacity];
  }
  public int[] asArray() {
    return array.clone();
  }
  private final int[] array;
  public static void main(String[] args) {
    Stack s=new Stack(3);
    s.asArray()[3]=-30;
  }
}
```

- En Java, les tableaux sont toujours mutables !

Mutabilité & création

- Si un objet est non mutable, toutes modifications entraînent la création d'un nouvel objet

```
public class Point { //immutable
  public Point(int x,int y) {
    this.x=x;
    this.y=y;
  }
  public Point translate(int dx,int dy) {
    return new Point(x+dx,y+dy);
  }
  private final int x;
  private final int y;
}
```

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

Mutabilité & création

- Il n'existe pas de façon de dire au compilateur qu'il faut absolument récupérer la valeur de retour

```
public class Point { //immutable
  public Point(int x,int y) {
    this.x=x;
    this.y=y;
  }
  public Point translate(int dx,int dy) {
    return new Point(x+dx,y+dy);
  }
  private final int x;
  private final int y;
}
```

```
public void anotherMethod {
  Point p=new Point(2,3);
  p.translate(1,1);      // oups
  System.out.println(p); // 2,3
}
```

Alors mutable ou pas ?

- En pratique
  - Les petits objets sont non-mutable,
    le GC les recycle facilement
  - Les gros (tableaux, list, table de hachage, etc) sont mutables pour des questions de performance
- Mais attention, pensez aux copies défensives
- On ne pool que les gros objets ayant un temps de création énorme (Connection DB, Thread, Socket)

Paramètre & type de retour

- Si cela dépend des cas d'utilisation
  - on écrit une version immutable et une mutable (String/StringBuilder)
  - les collections en Java peuvent être les deux, elle lève une exception ou non si l'on essaye de les modifier
- On envoie la version immutable aux méthodes **public**, dans une méthode ou vers les méthodes privées on peut utiliser la version immutable

Protection en cas d'objet mutable

- Pour une méthode, lorsque l’on envoie ou reçoit un objet mutable, on prend le risque que le code extérieur modifie l’objet pendant ou après l’appel de la méthode
- Pour palier cela
  - Passer une copie/effectuer une copie de l’argument
  - passer/accepter un objet non mutable

En prog. concurrente

- De plus, dans un environnement multi-thread, on préfère largement avoir des objets qui ne peuvent pas changer de valeur

- Cela évite tout les problèmes classiques de la programmation concurrente

Sous-typage

- Le sous-typage correspond au fait de pouvoir substituer un type par un autre

- Cela permet la réutilisation d'algorithme écrit pour un type et utiliser avec un autre

Principe de liskov

- Barbara Liskov(88) :
  If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.
- Le sous-typage est défini de façon générale sans notion de classe

Sous-typage en Java

- Le sous-typage existe pour les différents types de Java :
  - Pour l'héritage de classes ou d'interface
  - Pour une classe implantant l'interface
  - Pour les tableaux d'objets
  - Pour les types parametrés
- Le sous-typage ne marche qu'avec des objets

Sous-typage et héritage

- Si un classe (resp. interface) hérite d'une autre, la sous-classe (resp.interface) est un sous-type de la classe (resp.interface) de base

```
public class ClassTyping {
  private static class A {
  }
  private static class B extends A {
  }
  public static void main(String[] args) {
    A a=new B();
  }
}
```

- Comme B hérite de A, B récupère l'ensemble des membres de A

Sous-typage et interface

- Si une classe implante une interface, alors la classe est un sous-type de l'interface

```
public class InterfaceTyping {
  private interface I {
    void m();
  }
  private static class A implements I {
    public void m() {
      System.out.println("hello");
    }
  }
  public static void main(String[] args) {
    I i=new A();
  }
}
```

- Comme A implante I, A possède un code pour toutes les méthodes de I

Sous-typage et tableau

- En Java, les tableaux possèdent un sous-typages généralisés :
  - Un tableau est un sous-type de Object, Serializable et Clonable
  - Un tableau de U[] est un sous-type de T[] si U est un sous-type de T et T,U ne sont pas primitifs

```
public class InterfaceTyping {
  public static void main(String[] args) {
    Object[] o=args;
    double[] array=new int[3]; // illégal
  }
}
```

Tableau & ArrayStoreException

- Comme le sous-typage sur les tableaux existe, cela pose un problème :

```
public class InterfaceTyping {
  public static void main(String[] args) {
    Object[] o=args;
    o[0]=new Object();
    // ArrayStoreException à l'exécution
  }
}
```

- Il est possible de considérer un tableau de String comme un tableau d'objet mais il n'est possible d'ajouter un Object à ce tableau

Polymorphisme

- Le polymorphisme consiste à considérer les fonctionnalités suivant le **type réel** d'un objet et non suivant le type de la variable dans laquelle il est stocké

- Le sous-typage permet de stocker un objet comme une variable d'un super-type, le polymorphisme fait en sorte que les méthodes soient appelées en fonction du **type réel** de l'objet

A quoi ça sert ?

- Le polymorphisme fait en sorte que certaines parties de l'algorithme soit spécialisée en fonction du type réel de l'objet

```
public class Polymorphic {
  private static void print(Object[] array) {
    for(Object o:array)
      System.out.println(o.toString());
      // appel dynamiquement Integer.toString(), String.toString(),
      // Double.toString(), Boolean.toString()
  }
  public static void main(String[] args) {
    Object[] array=new Object[]{2,arg[0],3.4,false};
    print(array);
  }
}
```

- En Java, seul l'appel de méthode est polymorphe

Appel virtuel & compilation

- Le mécanisme d'appel polymorphe (appel virtuel) est décomposé en deux phases :

  1) Lors de la **compilation**, le compilateur choisit la méthode la **plus spécifique** en fonction du **type déclaré** des arguments

  2) Lors de **l'exécution**, la VM choisie la méthode en fonction du **type réel** du receveur (le type de l'objet sur lequel on applique '.')

Condition d'appel virtuel

- Il n'y a pas d'appel virtuel si la méthode est :
  - **statique** (pas de receveur)
  - **private** (pas de redéfinition possible, car pas visible)
  - **final** (pas le droit de redéfinir)
  - Si l'appel se fait par **super**
- Dans les autres cas, l'appel est virtuel

Exemple d'appel virtuel

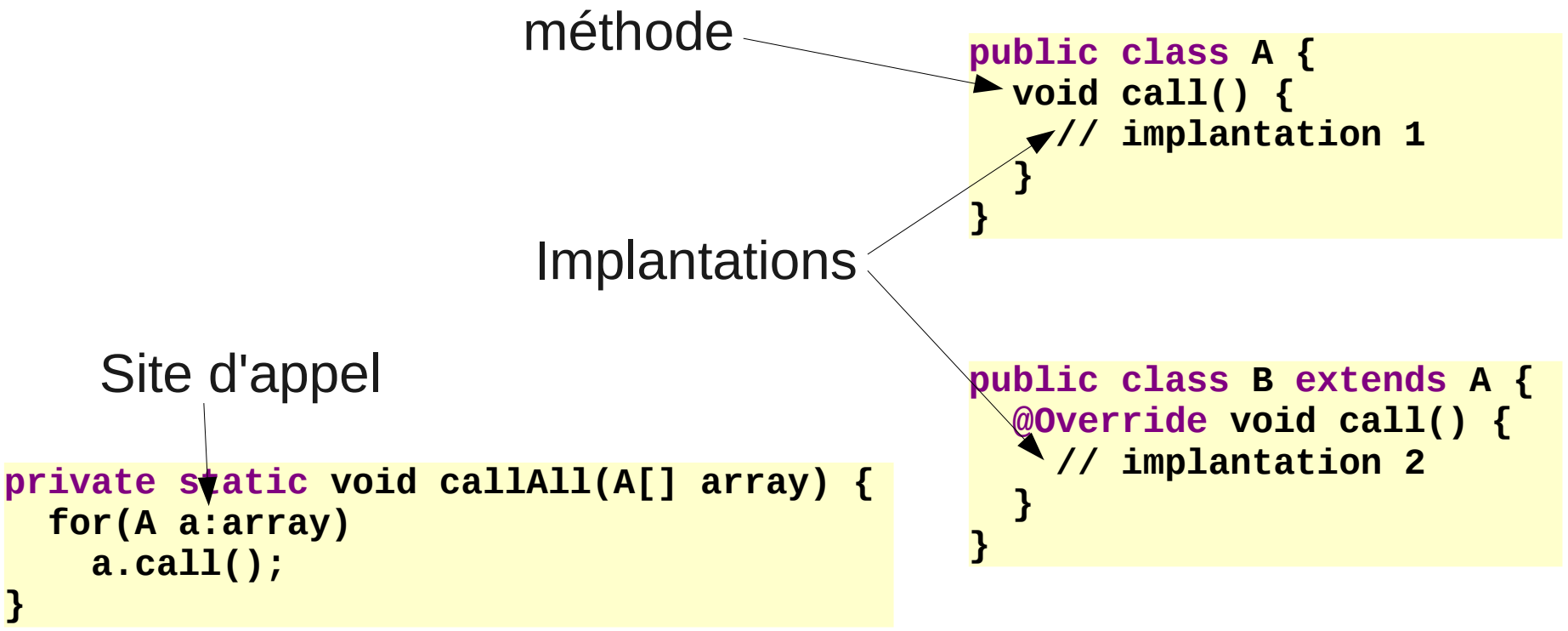
- Un appel, même avec “this.”, est polymorphe

```
public class FixedSizeList {
  boolean isEmpty() {
    return size()==0;
    // est équivalent à: return this.size()==0;
  }
  int size() {
    return 10;
  }
}

public class EmptyList extends FixedSizeList {
  int size() {
    return 0;
  }
}
public class void main(String[] args) {
  FixedSizeList list=new EmptyList();
  System.out.println(list.isEmpty()); // true
}
```

Implantation et Site d'appel

- On distingue la méthode, les implantations de cette méthode et le site d'appel à la méthode

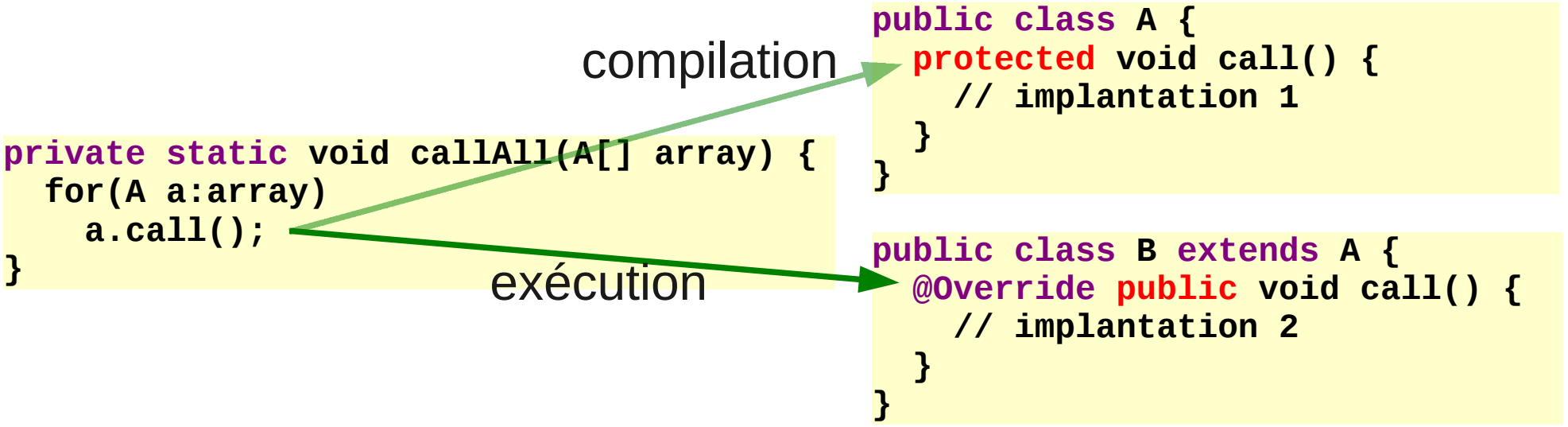


Condition de la redéfinition

- Il y a redéfinition de méthode s'il est possible pour un site d'appel donné d'appeler la méthode redéfinie en lieu et place de la méthode choisie à la compilation
- Le fait qu'une méthode redéfinisse une autre dépend :
  - Du nom de la méthode
  - Des modificateurs de visibilité des méthodes
  - De la signature des méthodes
  - Des exceptions levées (throws) par la méthode

Visibilité et redéfinition

- Il faut que la méthode redéfinie ait une visibilité au moins aussi grande
  (private< <protected<public)



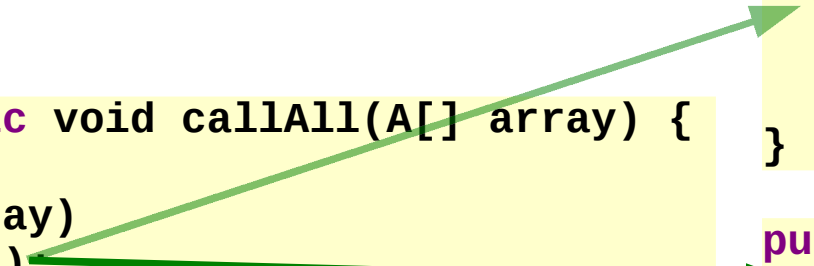
Covariance du type de retour

- Le type de retour de la méthode redéfinie peut-être un sous-type de la méthode à redéfinir

```
private static void callAll(A[] array) {
  Object o;
  for(A a:array)
    o=a.call();
}
```

```
public class A {
  Object call() {
    // implantation 1
  }
}
```

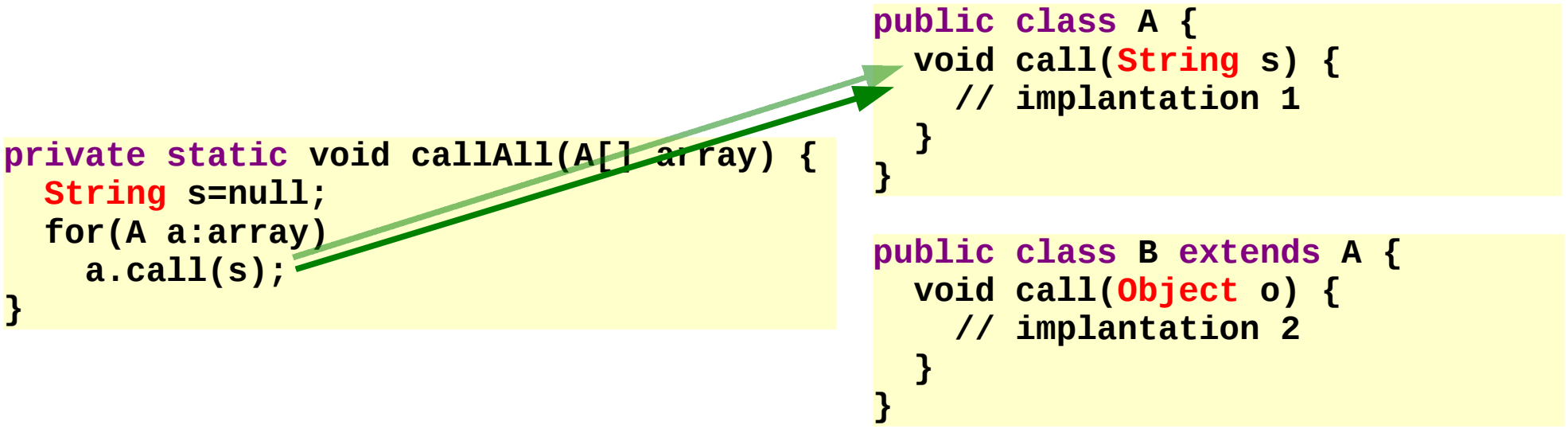
```
public class B extends A {
  @Override String call() {
    // implantation 2
  }
}
```



- Ne marche qu'à partir de la 1.5

Contravariance des paramètres

- Les types des paramètres peuvent être des super-types du type de la méthode à redéfinir



```
private static void callAll(A[] array) {
  String s=null;
  for(A a:array)
    a.call(s);
}
```

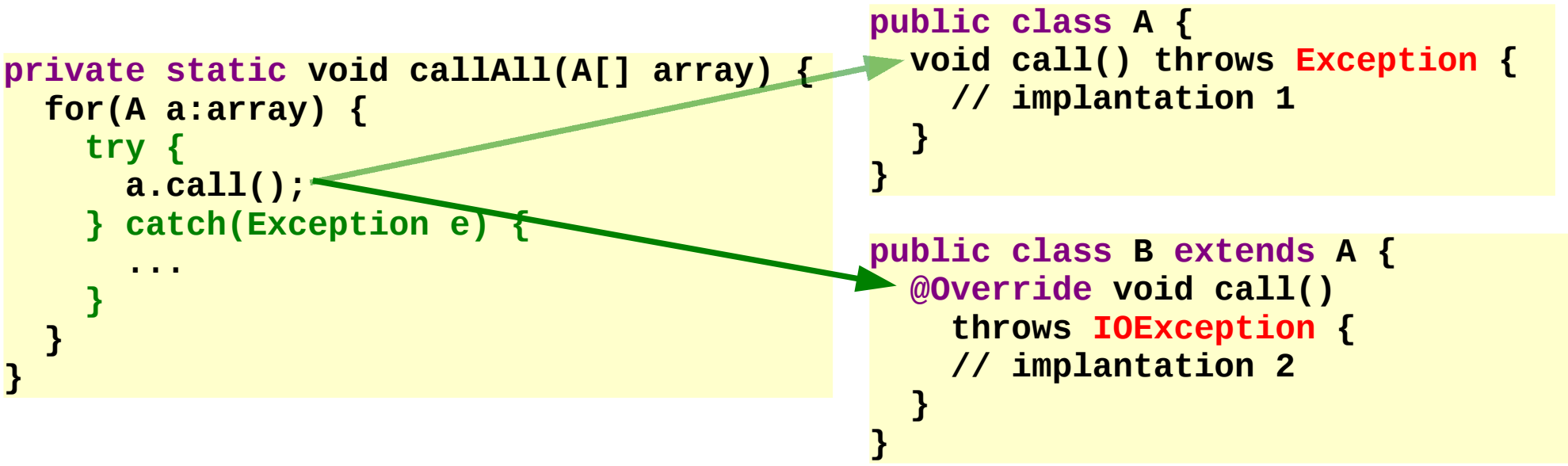
```
public class A {
  void call(String s) {
    // implantation 1
  }
}
```

```
public class B extends A {
  void call(Object o) {
    // implantation 2
  }
}
```

- Pas implanté en Java, dommage !

Covariance des exceptions

- Les exceptions levés peuvent être des sous-types de celles déclarées



```
private static void callAll(A[] array) {
  for(A a:array) {
    try {
      a.call();
    } catch(Exception e) {
      ...
    }
  }
}
```

```
public class A {
  void call() throws Exception {
    // implantation 1
  }
}
```

```
public class B extends A {
  @Override void call()
    throws IOException {
    // implantation 2
  }
}
```

- Les exceptions non *checked* ne compte pas

Covariance des exceptions (2)

- La méthode redéfinie peut ne pas levée d'exception

```
private static void callAll(A[] array) {
  for(A a:array) {
    try {
      a.call();
    } catch(Exception e) {
      ...
    }
  }
}
```

```
public class A {
  void call() throws Exception {
    // implantation 1
  }
}
```

```
public class B extends A {
  @Override void call() {
    // implantation 2
  }
}
```

- L'inverse ne marche pas !!!

vtable et interface

- Le mécanisme de vtable ne marche pas bien avec l'héritage multiple car la numérotation n'est plus unique
- Les implantations d'interface possèdent un mécanisme externe de fonctionnement utilisant une vtable pour chaque interface implantée
- L'appel à travers une interface est donc en général plus lent que l'appel à travers une classe même abstraite

Appel de méthode

- L'algorithme d'appel de méthode s'effectue en deux temps
  - On recherche les méthodes applicables (celles que l'on peut appeler)
  - Parmi les méthodes applicables, on recherche s'il existe une méthode plus spécifique (dont les paramètres serait sous-types des paramètres des autres méthodes)
- Cet algorithme est effectué par le compilateur

Méthodes applicables

- Ordre dans la recherche des méthodes applicables :

  1) Recherche des méthodes à nombre fixe d'argument en fonction du sous-typage

  2) Recherche des méthodes à nombre fixe d'argument en permettant l'auto-[un]boxing

  3) Recherche des méthodes en prenant en compte les varargs

- Dès qu'une des recherches trouve une ou plusieurs méthodes la recherche s'arrête

Exemple de méthodes applicables

- Le compilateur cherche les méthodes applicables

```
public class Example {
  public void add(Object value) {
  }
  public void add(CharSequence value) {
  }
}
```

```
  public static void main(String[] args) {
    Example example=new Example();
    for(String arg:args)
      example.add(arg);
  }
```

- Ici, **add(Object)** et **add(CharSequence)** sont applicables

Méthode la plus spécifique

- Recherche parmi les méthodes applicables, la méthode la plus spécifique

```
public class Example {
  public void add(Object value) {
  }
  public void add(CharSequence value) {
  }
}
```

```
  public static void main(String[] args) {
    Example example=new Example();
    for(String arg:args)
      example.add(arg); // appel add(CharSequence)
  }
```

- La méthode la plus spécifique est la méthode dont tous les paramêtres sont sous-type des paramètres des autres méthodes

Méthode la plus spécifique (2)

- Si aucune méthode n'est plus spécifique que les autres, il y a alors ambiguité

```
public class Example {
  public void add(Object v1, String v2) {
  }
  public void add(String v1, Object v2) {
  }
}       public static void main(String[] args) {
          Example example=new Example();
          for(String arg:args)
            example.add(arg,arg);
          // reference to add is ambiguous, both method
          // add(Object,String) and method add(String,Object) match
        }
```

- Dans l'exemple, les deux méthodes **add()** sont applicables, aucune n'est plus précise

Surcharge et auto-[un]boxing

- Le boxing/unboxing n'est pas prioritaire par rapport à la valeur actuelle

```
public static void main(String[] args) {
  List list=...
  int value=3;
  Integer box=value;

  list.remove(value); // appel remove(int)
  list.remove(box);   // appel remove(Object)
}
```

```
public class List {
  public void remove(Object value) {
  }
  public void remove(int index) {
  }
}
```

Surcharge et Varargs

- Une méthode à nombre variable d'arguments n'est pas prioritaire par rapport à une méthode à nombre fixe d'arguments

```
public class VarargsOverloading {
  private static int min(int... array) {
  }

  private static int min(double value) {
  }

  public static void main(String[] args) {
    min(2); // appel min(double)
  }
}
```

Surcharge et Varargs (2)

- Surcharge entre deux varargs :

```
public class VarargsOverloading {
  private static void add(Object... array) { }
  private static void add(String ... array) { }

  public static void main(String[] args) {
    add(args[0],args[1]); // appel add(String...)
  }
}
```

- Choix entre deux varargs ayant même wrapper :

```
public class VarargsOverloading {
  private static int min(int... array) { }
  private static int min(Integer... array) {
}
  public static void main(String[] args) {
    min(2);  // reference to min is ambiguous, both method
             // min(int...) and method min(Integer...) match
  }
}
```

Implantation du polymorphisme

- Le polymorphiqme est implanté de la même façon quelque soit le langage (enfin, vrai pour les language typé C++, Java, C#)

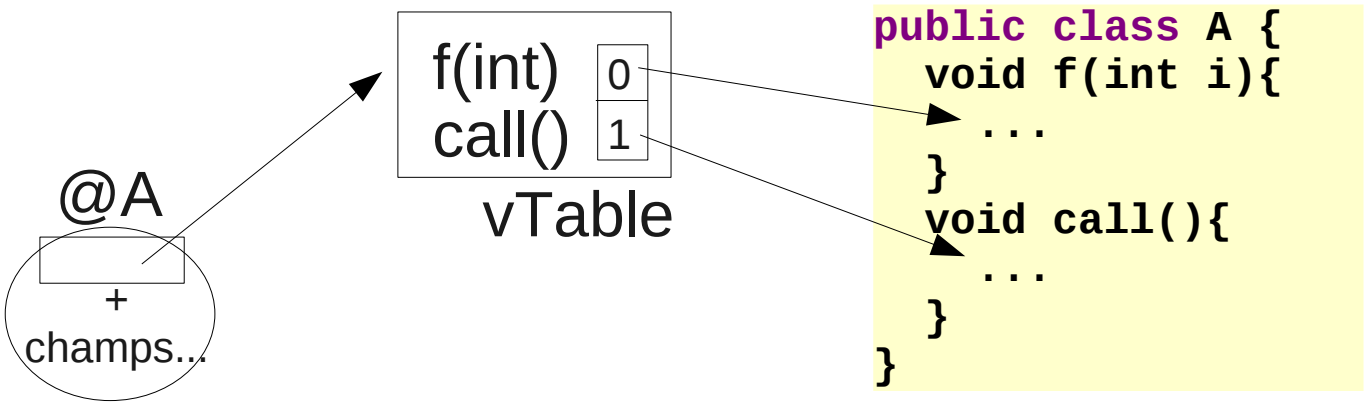
```
public class A {
  void f(int i){
    ...
  }
  void call(){
    ...
  }
}
```

```
public class B extends A{
  void call(){
    ...
  }
  void f(){
    ...
  }
}
```

```
public class AnotherClass {
  void callSite(){
    A a=new B();
    a.call();
    a.f(7);
  }
}
```

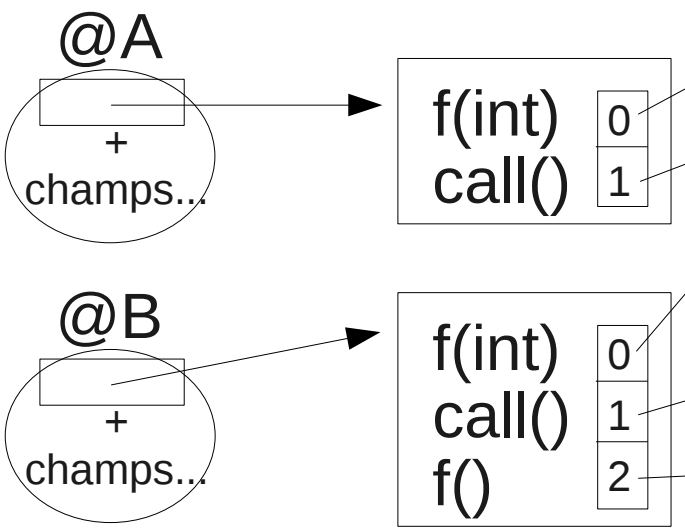
Implantation du polymorphisme

- Chaque objet possède en plus de ces champs un pointeur sur une table de pointeurs de fonction
- Le compilateur attribue à chaque méthode un index dans la table en commençant par numéroter les méthodes des classes de base



vtable

- Deux méthodes redéfinies ont **le même index**
- L'appel polymorphe est alors : object.vtable[index](argument)




```
public class A {
  void f(int i){
    ...
  }
  void call(){
    ...
  }
}
```

```
public class B extends A{
  void call(){
    ...
  }
  void f(){
    ...
  }
}
```

vtable et interface

- Le mécanisme de vtable ne marche pas bien avec l'héritage multiple car la numérotation n'est plus unique
- Les implantations d'interface possèdent un mécanisme externe de fonctionnement utilisant une vtable pour chaque interface implantée
- L'appel à travers une interface est donc en général plus lent que l'appel à travers une classe même abstraite

Coder en Java

- Convention de Code
- Declaration des variables au plus proche
- Pas d'initialisation inutile
- Champs private/final
- Exception ?
- Assert/check des paramètres
- Documentation
- Valeur par défaut des paramètres
- Import sans *
- Visibilité (type/méthode)