

L'appel de méthode

# Typage et Objet

Les langages « objet » peuvent être typés à la compilation et/ou à l'exécution

## 3 sortes de langages

- Typé à la compilation pas à l'exécution
  - C++ (sans RTTI), OCaml
- Typé à la compilation et à l'exécution
  - Java, C#
- Typé à l'exécution
  - PHP, Javascript, Python, Ruby

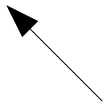
# Type et Objet

```
URI uri = new URI("http://www.playboy.com");
```

Type pour le compilateur



Classe pour la VM



Le **type** d'un objet correspond à son *interface* c-a-d l'ensemble des méthodes que l'on peut appeler sur cet objet

La **classe** d'un objet correspond à l'ensemble des propriétés + méthodes utilisées pour créer un objet

En Java, la classe est aussi un objet !

# 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 donné et utilisé avec un autre

# Principe de Substitution de Liskov

Barbara Liskov (1988) :

*« 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 paramétrés (generics)

Le sous-typage ne marche qu'avec des objets

# Sous-typage et héritage

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

```
class A {  
}  
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

```
interface I {  
    void m();  
}  
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-typage généralisé :

- Un tableau est un sous-type de Object, Serializable et Clonable
- Un tableau de U (U[ ]) est un sous-type d'un tableau de T (T[ ]) si U est un sous-type de T et T,U ne sont pas primitifs

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

# Tableau & ArrayStoreException

- Ce sous-typage sur les tableaux pose un problème :

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

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

# Appel virtuel & compilation

Le mécanisme d'appel polymorphe (appel virtuel ou « late binding ») 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 choisit la méthode en fonction de la **classe** du receveur (l'objet sur lequel on applique « .methode() »)

# 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 accessible)
- **final** ou la classe est **final** (pas le droit de redéfinir)
- Si l'appel se fait par **super**

dans les autres cas, l'appel est virtuel

# Même un appel avec this est polymorphe

```
class FixedSizeList {
    boolean isEmpty() {
        return this.size() == 0;
    }
    int size() {
        return 10;
    }
}
class EmptyList extends FixedSizeList {
    @Override
    int size() {
        return 0;
    }
}
public class PolymorphThis {
    public static void main(String[] args) {
        FixedSizeList list = new EmptyList();
        System.out.println(list.isEmpty()); // true
    }
}
```

# Site d'appel et implantations

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

méthode

Implantations

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

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

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

Site d'appel

# 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 < *default* < protected < public)

compilation

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

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

exécution

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

# Covariance du type de retour

Le type de retour de la méthode redéfinie peut-être un sous-type du type de retour 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 *pourraient* être des super-types du type de la méthode à redéfinir

```
private static void callAll(A[] array) {  
    String s = ...  
    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 levées 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 comptent pas

# Covariance des exceptions (2)

La méthode redéfinie peut ne pas lever 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 !

# Recherche de méthode à la compilation

L'algorithme de recherche de la méthode à appeler s'effectue en deux temps

A) on recherche les méthodes **applicables**  
(celles que l'on peut appeler)

B) parmi les méthodes applicables, on recherche s'il existe une méthode plus spécifique  
(dont les paramètres sont sous-types des paramètres des autres méthodes)

Cet algorithme est effectué par le **compilateur**

# A. méthodes applicables

Ordre dans la recherche des méthodes applicables :

- 1) Recherche des méthodes à nombre fixe d'arguments en fonction du sous-typage & conversions des types primitifs
- 2) Recherche des méthodes à nombre fixe d'arguments 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);  
}
```

**add(Object)** et **add(CharSequence)** sont applicables

## B. méthode la plus spécifique

Parmi les méthodes applicables, recherche de 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 paramètres sont sous-types des paramètres des autres méthodes

# Méthode la plus spécifique (2)

Si aucune méthode n'est plus spécifique que les autres, il y a alors ambiguïté (compile pas)

```
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  
    }
```

Les deux méthodes **add()** sont applicables, mais aucune n'est plus précise que l'autre

# Et lors de l'exécution ?

Le polymorphisme est implanté de la même façon quelque soit le langage objet typé (C++, Java, C#)

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

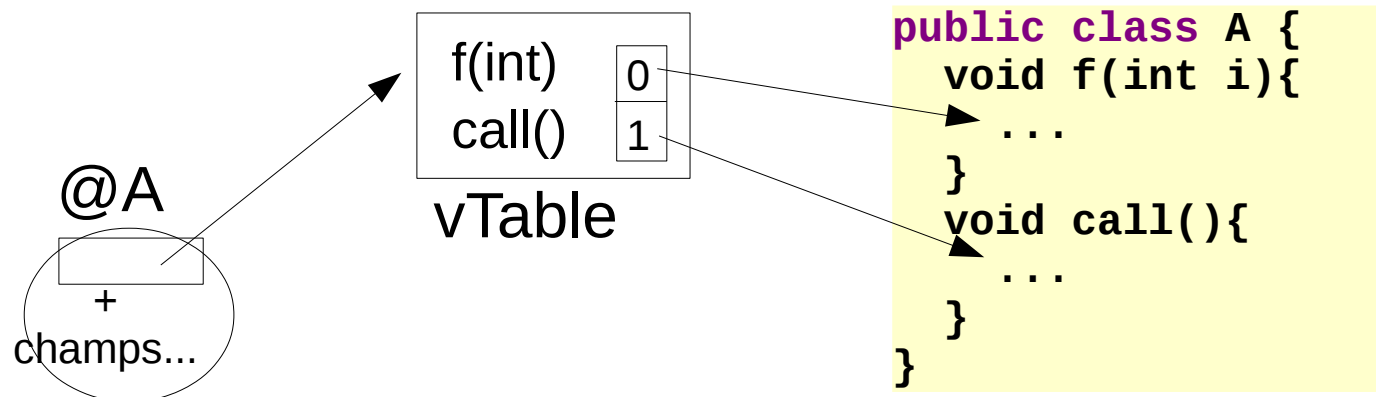
```
public class AnotherClass {  
    void callSite(){  
        A a=new B();  
        a.call();  
        a.f(7);  
    }  
}
```

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

# Implantation du polymorphisme

Chaque objet possède en plus de ses champs un pointeur sur une table de pointeurs de fonction

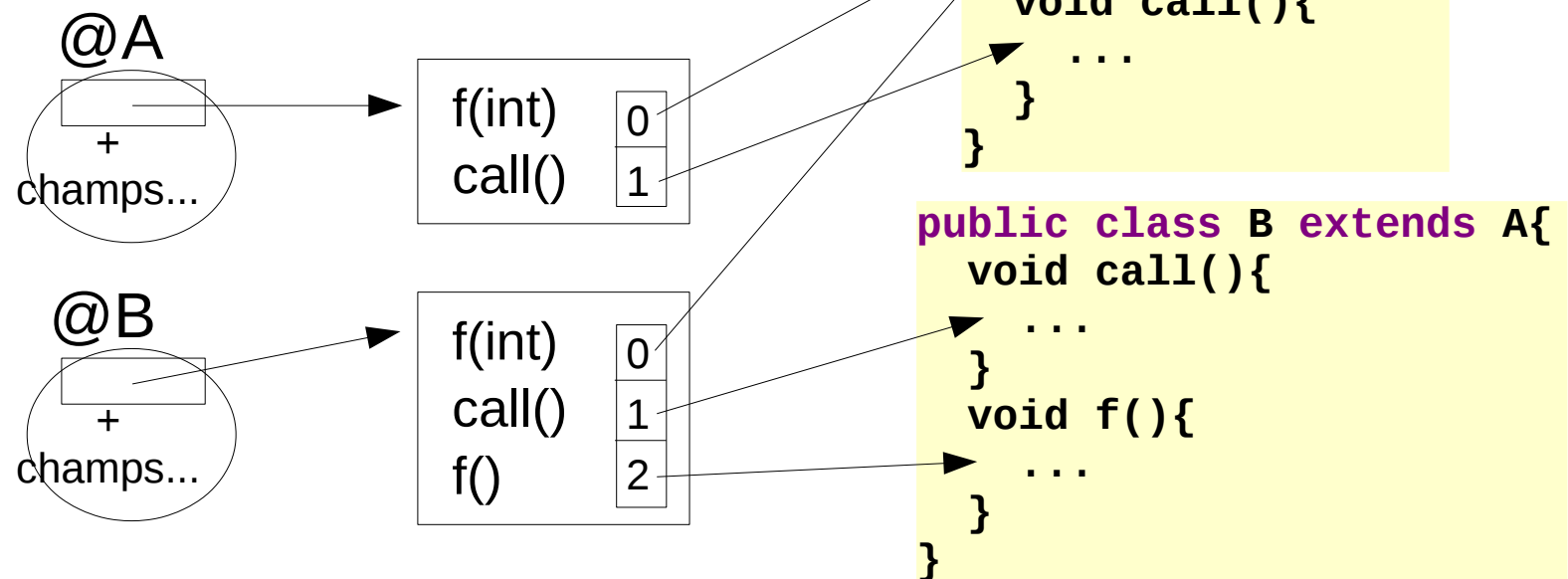
Le compilateur attribue à chaque méthode un index dans la table en commençant par numéroté les méthodes des classes de base



# vtable

Deux méthodes redéfinies ont **le même index**

L'appel polymorphe est alors :  
object.vtable[index](argument)



# vtable et interface

Le mécanisme de vtable ne marche pas bien avec l'héritage multiple car la numérotation n'est plus unique

Pour les implantations d'interfaces la classe possède une vtable par interface implantée

Sans optimisation, l'appel à travers une interface est plus lent que l'appel à travers une classe même abstraite