Method call

Typing and objects

« object » languages could be typed at compile-time and/or at run-time

3 kinds of languages

- Typed at compile-time and not at run-time
  C++ (without RTTI), OCaml
- Typed at compile-time and at run-time
  - Java, C#
- Typed at run-time
  - PHP, Javascript, Python, Ruby

Types and objects

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

The **type** of an object correspond to its *interface,* that is the set of methods we could call on this object
The **class** of an object correspond to the set of properties and methods used to create an object

In Java, a class is also an object !

Sub-typing

Sub-typing correspond to the possibility of substituting a type by another type

This allows algorithms, wrote for a given type, to be **reused** with another (sub-)type

Liskov substitution principle

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 sub-type of T. »

Note: sub-typing is defined independently of the concept of class

Sub-typing in Java

Sub-typing exists for all “reference” types of Java :

- For inheritance between classes or interfaces
- For a class implanting an interface
- For arrays of objects
- For parametrized types (generics)

Sub-typing only works with objects (not with primitive types)

Sub-typing and inheritance

- If a class B (resp. interface) extends another class A (resp. interface), the sub-class B (resp. sub-interface) defines a sub-type of the type of base class A (resp. interface)

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

- Since B extends A, then B pick up all members of A

Sub-typing and array

In Java, arrays have a generalized sub-tying:

- An array is sub-type of Object, Serializable and Clonable
- An array of U (U[ ]) is sub-type of array of T (T[ ]) if U is sub-type of T and T,U aren't primitive types

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

Sub-typing and interface

- If a class A implements an interface I, then the type A is a sub-type of type I

```
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();
}
```

- Since A implements I, then A provides a code (implementation) for all methods declared in I

Array & ArrayStoreException

- Sub-typing of arrays pose a problem :

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

- You could consider an array of String as an array of Object, but you couldn’t store an Object in this array

Virtual call & compilation

The mechanism of polymorphic call (virtual call or « late biding ») is decomposed int two steps :

1) At **compile-time**, compiler chooses the **most specific method** given the **declared types** of the arguments

2) At **run-time**, the VM chooses the method to call with respect to the **class** of the receiving object (on which we call « .method() »)

Condition of virtual call

There is no virtual call if the method is :

- **static** (no receiver)
- **private** (no overriding possible because not accessible)
- **final** or class is **final** (overriding forbidden)
- called through **super**

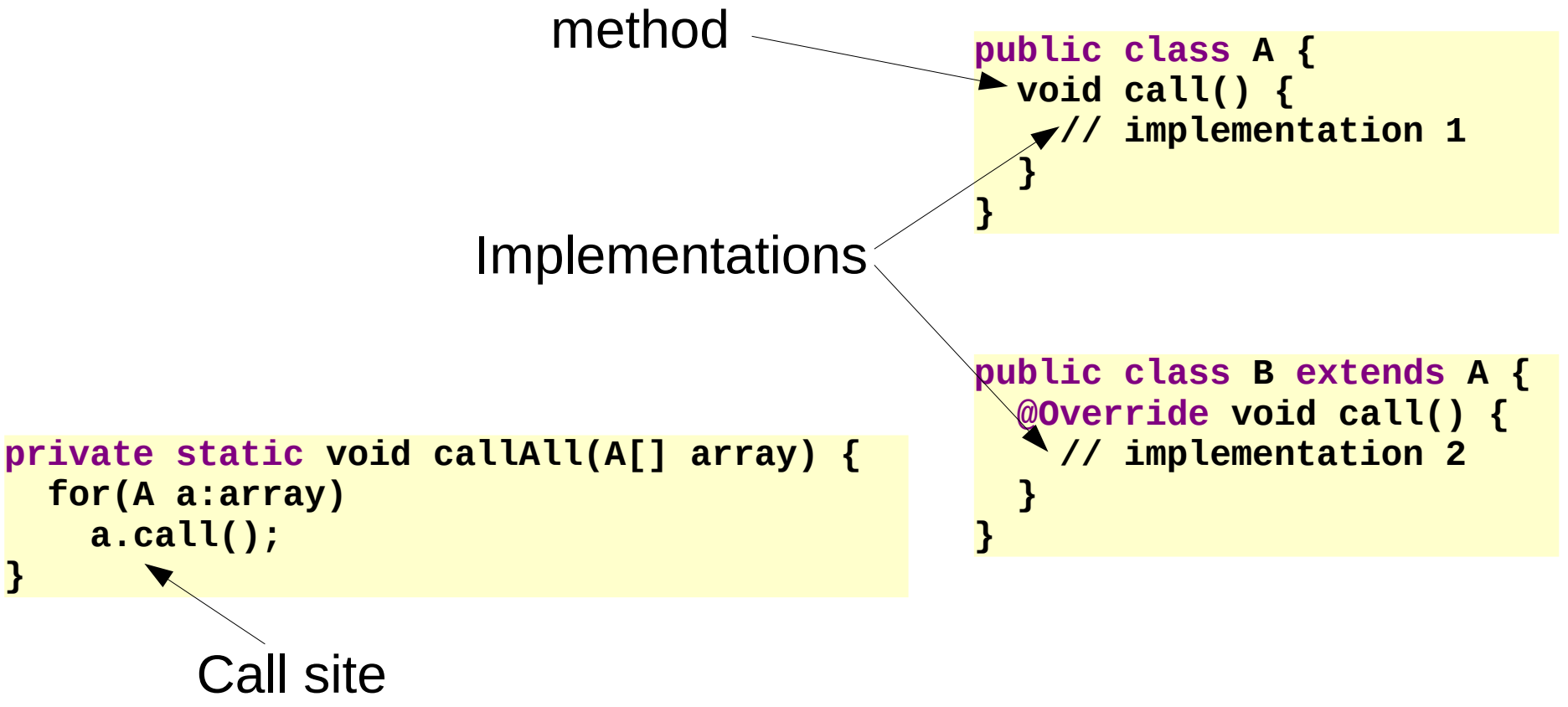
in other cases, the call is virtual (polymorphic)

Even a call on this is polymorphic

```
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
  }
}
```

Call site and implementations

We distinguish a method, its implementations and the call site of this method



Overriding conditions

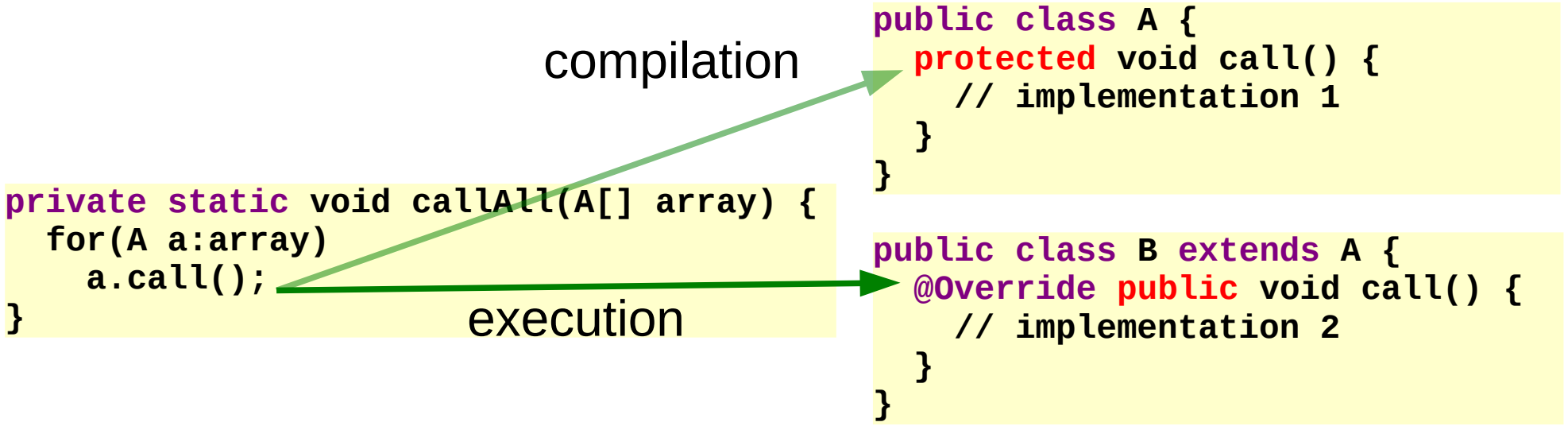
Method overriding occurs if it is possible, for a given call site, to invoke at run-time the overriding method instead of those retained at compile-time

If a method overrides another depends on methods :

- Names
- Accessibility modifiers
- Signatures
- Raised exceptions (throws)

Accessibility and overriding

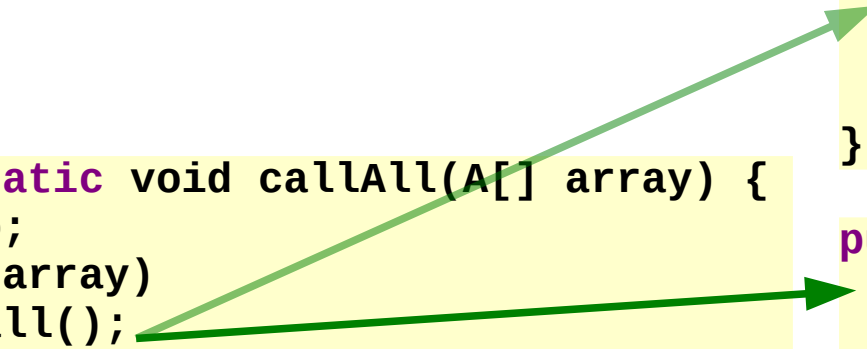
- Overriding method must have at least the same accessibility as those overridden (private < *default* < protected < public)



Covariance of return type

Return type of overriding method can be a sub-type of the return type of the overridden method

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



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

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

Since JDK 1.5

Contravariance of parameters

- Types of parameters of the overriding method ***could*** be super-types of parameters of overridden method

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

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

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

But it is not implemented in Java!
=> considered as two distinct methods

Covariance of exceptions

- Types of exceptions raised by overriding method can be sub-types of those declared by the overridden method

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

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

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

Only **checked** exceptions are considered

Covariance of exceptions (2)

Overriding method can raise no exception

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

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

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

Opposite doesn’t work !

Method resolution at compile-time

Resolution algorithm to find the “right” method to invoke works in two steps:

A) Look for **applicable methods**
(that can match with the call site / parameter types)

B) Among those applicable methods, check if one is “**most specific**” (its parameter types are sub-types of parameter types of other methods)

This two steps algorithm is processed by the **compiler**

A. applicable methods

Order in the the search of applicable methods :

1) Search among methods with fixed number of arguments, w.r.t sub-typing & primitive type conversions

2) Search among methods with fixed number of arguments allowing auto-[un]boxing

3) Search among methods allowing varargs (...)

As soon as one step (1, 2 or 3) finds one or several applicable methods, search stops.

Example of applicable methods

Compiler looks for applicable methods

```
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) and **add(CharSequence)** are applicable methods

B. most specific method

Among applicable methods, looks for the most specific one

```
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)
  }
```

Most specific method is that whose each parameter type is a sub-type of the parameter type of other methods

Most specific method (2)

If no method is most specific than the others the call is ambiguous (compilation error)

```
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
}
```

Both **add()** methods are applicable, but no one is more specific than the other

And what about run-time ?

Polymorphism is implemented in the same way whatever the typed object oriented language (C++, Java, C#)

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

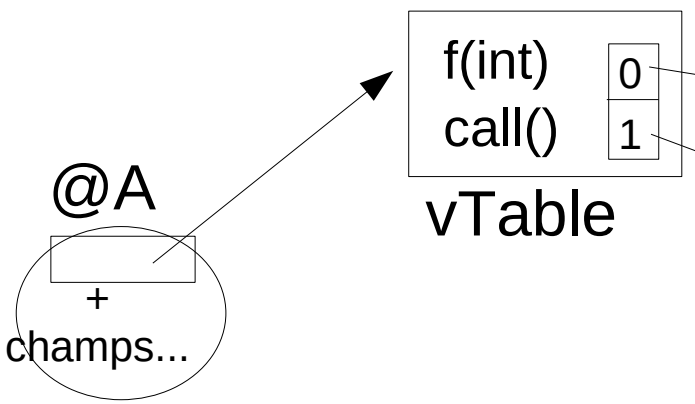
```
public class AnotherClass {
  void callSite(){
    A a=new B();
    a.call();
    a.f(7);
  }
}
```

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

Polymorphism implementation

Each object has (in addition to its fields) a pointer to a table of function pointers

The compiler assign an index in the table to each method, starting the numbering by methods of base-classes

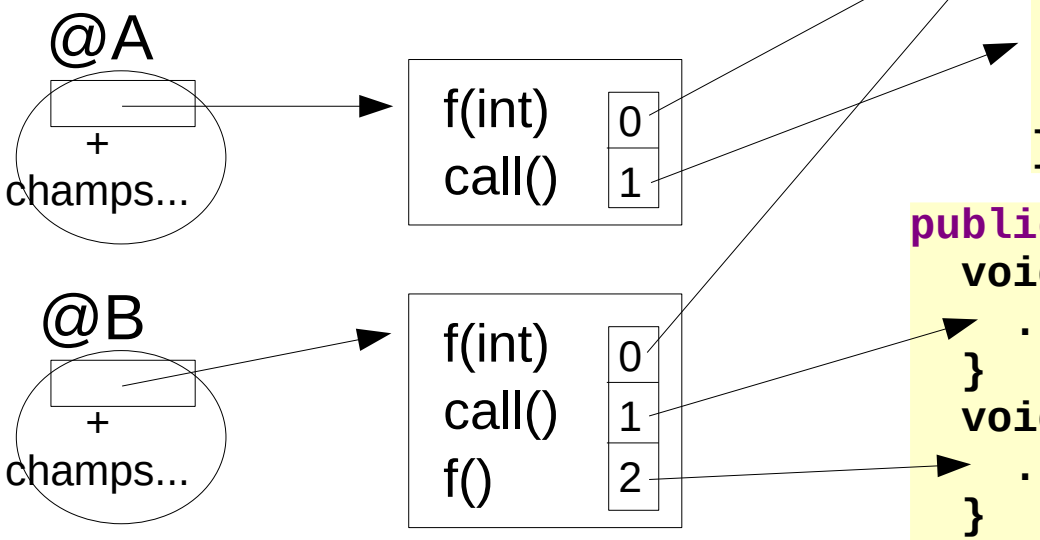



```
public class A {
  void f(int i){
    ...
  }
  void call(){
    ...
  }
}
```

vtable

An overriding method has the **same index** of the overridden one

Polymorphic (virtual) call is then : object.vtable[index](argument)




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

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

vtable and interface

vtable mechanism is not well suited for multiple inheritance, because the numbering is not unique

For interface implementations, the class owns one vtable per implemented interface

Without optimization, a method call through an interface is slower than through a class, even an abstract class.