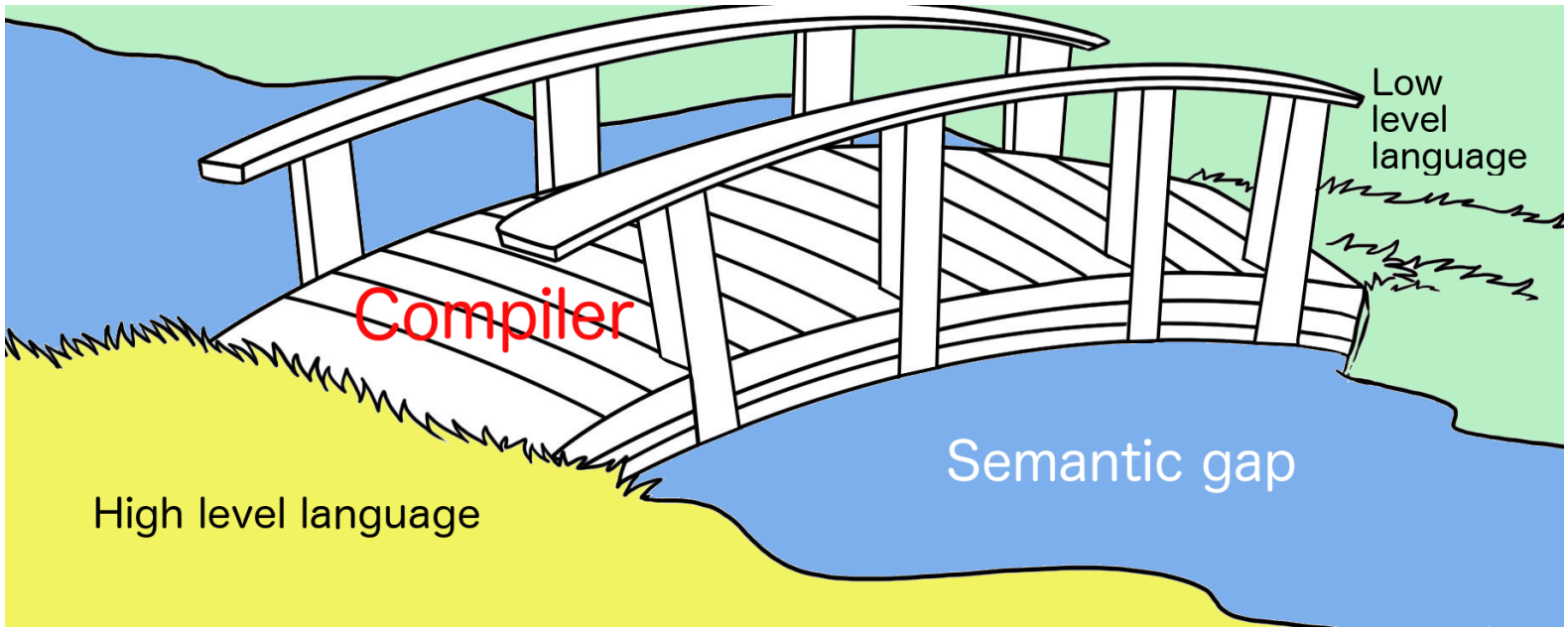
Compilers and computer architecture: Compiling OO languages

Martin Berger [1]

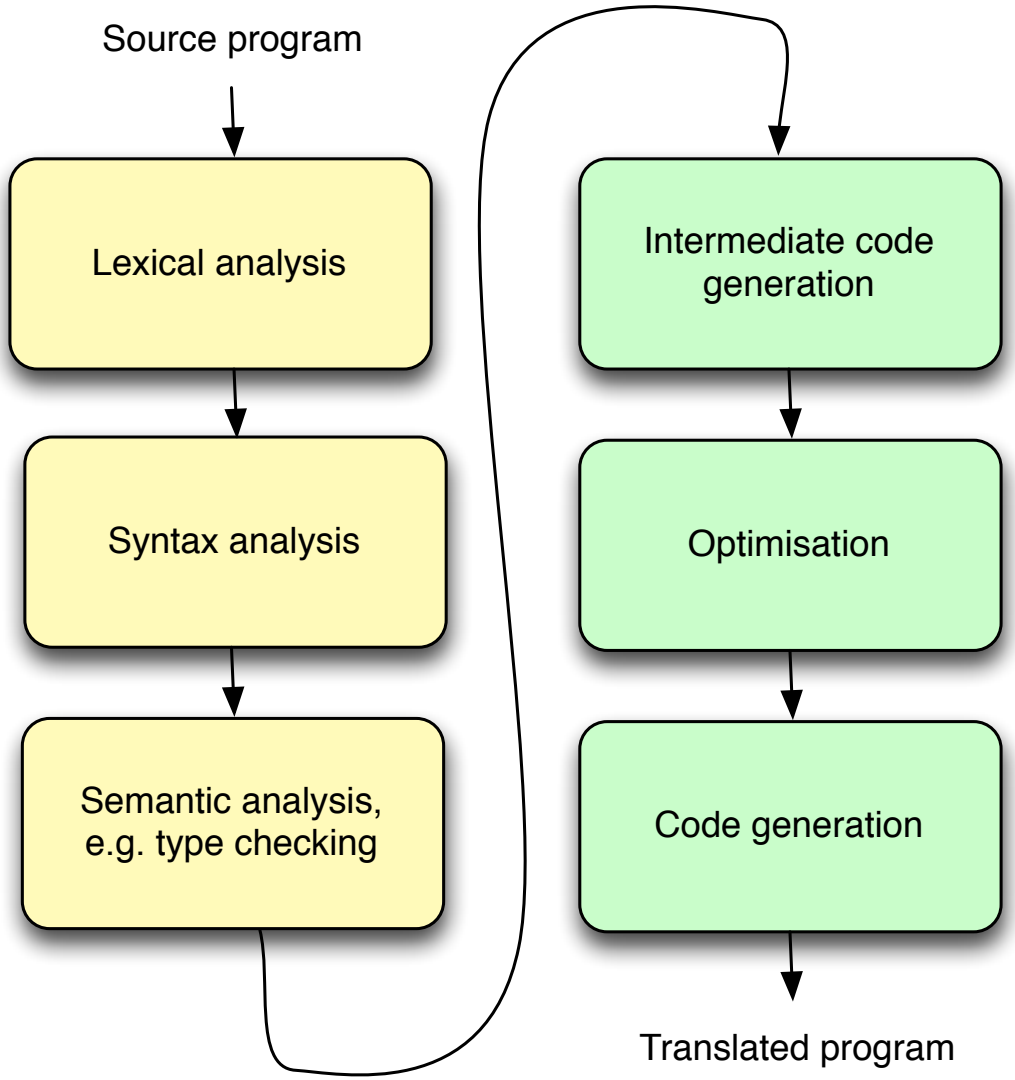
December 2019

[1] Email: M.F.Berger@sussex.ac.uk, Office hours: Wed 12-13 in Chi-2R312

Recall the function of compilers



Recall the structure of compilers



Introduction

The key ideas in object oriented programming are:

- Data (state) hiding through objects, objects carry access mechanisms (methods).
- Subtyping. If B is a subclass of A than any object that is an instance of B can be used whenever and wherever an instance of A is expected.

Let's look at an example.

A Java example

```
interface A { int f () { ... } }
class B implements A { int f () { ... } }
class C implements A { int f () { ... } }
...
public static void main ( String [] args ) {
   ...
   A a = if ( userInput == 0 ) {
      new B (); }
   else {
      new C (); }
   ...
   a.f() // Does the compiler know which f is used?
```

At compile time we don't know exactly what objects we have to invoke methods on.

Problem

The code generator must generate code such that access (methods and instance variables) to an object that is an instance of *A* must work for any subclass of *A*.

Indeed some subclasses of *A* might only become available at run-time.

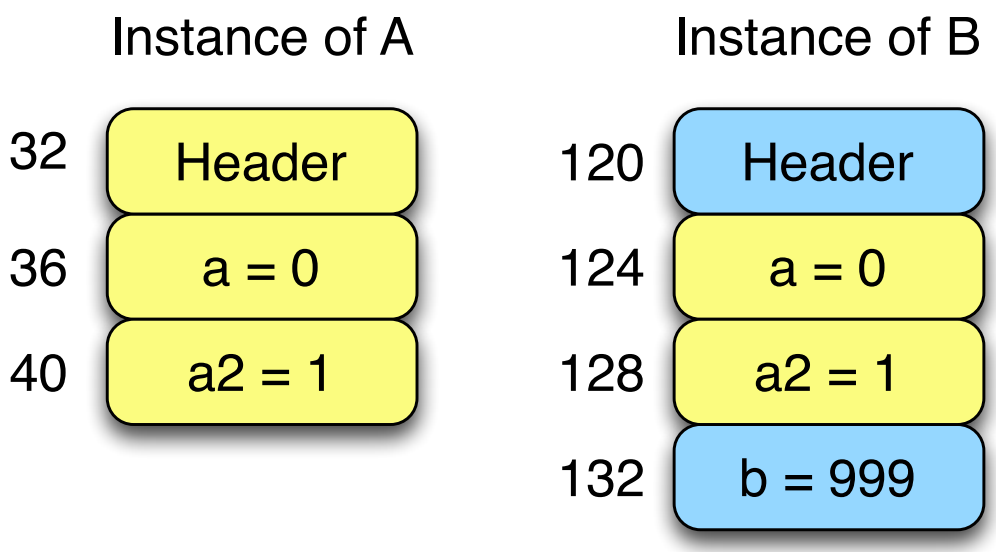
So we have two questions to ask:

- How are objects laid out in memory?
- How is method invocation implemented?

Object layout in memory

We solve these problems using the following ideas.

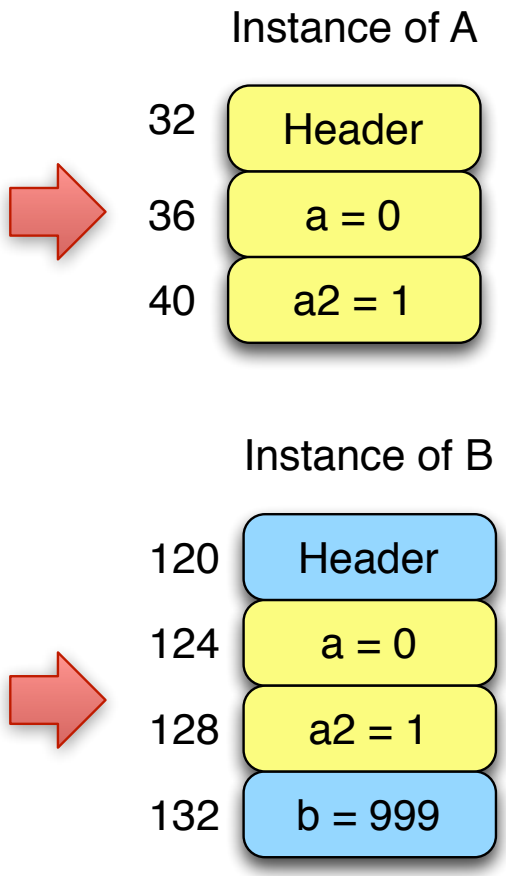
- Objects are laid out in contiguous memory, with pointer pointing to that memory giving us access to object.
- Each instance variable is at the same place in the contiguous memory representing an object, i.e. at a **fixed offset**, known at **compile-time**, from the top of the contiguous memory representing the offset.
- Subclass instance variables are added 'from below'.



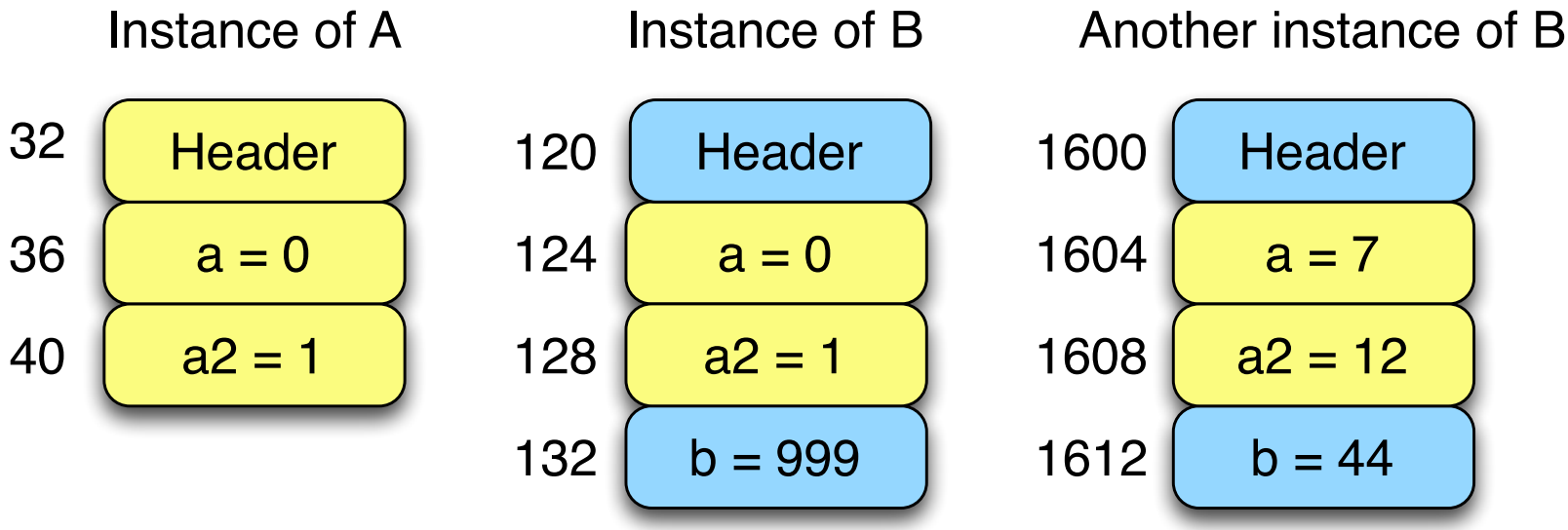
Object layout in memory

Note that the the number and types of instance variables/attributes (i.e. size in memory) are available to the compiler at compile time.

```
class A {
  int a = 0;
  int a2 = 1;
  int f () {
    a = a + a2;
    return a; } }

class B extends A {
  int b = 999;
  int f () { return a; }
  int g () {
    a = a - b + a2;
    return a; } }
```



Object layout in memory

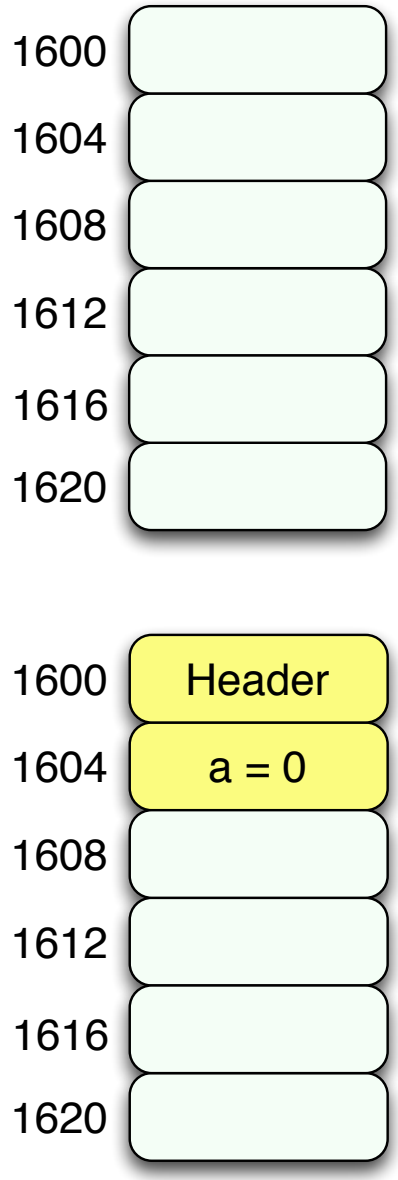


The compiler uses the same layout for every instance of a class. So if the size of the header is 4 bytes, and integers are 4 bytes, then `a` is always at offset 8 from the beginning of the object, and `a2` is always at offset 12, both in instances of `A` and `B`, and likewise for other subclasses of `A`, or other header and field sizes

This ensures that every instance of `B` can be used where an instance of `A` is expected.

Object layout in memory

This also works with deeper inheritance hierarchies.




```
class A {
    int a = 0; }
```

We've overlooked one subtle issue

In Java and other languages you can write this:

```
class A { public int a = 0; }
class B extends A { public int a = 1; }

class Main {
   public static void main ( String [] args ) {
      A a = new A ();
      B b = new B ();
      A ab = new B ();
      System.out.println ( "a.a = " + a.a );
      System.out.println ( "b.a = " + b.a );
      System.out.println ( "ab.a = " + ab.a ); } }
```

What do you think this program outputs? Why? (Example: prog/ex3.java)

Shadowing of instance variables/attributes

The solution is twofold:

- To determine what instance variable/attribute to access, the code generator looks at the **static** type of the variable (available at compile-time). Note that the type of the object at run-time might be different (e.g. `A ab = new B ();` in the example on the last slide).
- If there is more than one instance variable/attribute with the same name, we choose the one that is **closest** up the inheritance hierarchy.

Shadowing of instance variables/attributes (bigger example)

```
class A1 { a ...} // defines a
class A2 extends A1 { a ...} // defines a
class A3 extends A2 { a ...} // defines a
class A4 extends A3 {...} // doesn't define a
class A5 extends A4 { a ...} // defines a
class A6 extends A5 {...} // doesn't define a
class A7 extends A6 {...} // doesn't define a
class A8 extends A7 {...} // doesn't define a
class A9 extends A8 {...} // doesn't define a
class A10 extends A9 { a ...} // defines a
...
A7 x = new A10 ()
...
print ( x.a ) // prints A5's a
```

(Example: ex5.java)

Shadowing of instance variables/attributes

Do you think Java's shadowing is a good idea?

What alternative approaches would you recommend?

Multiple inheritance

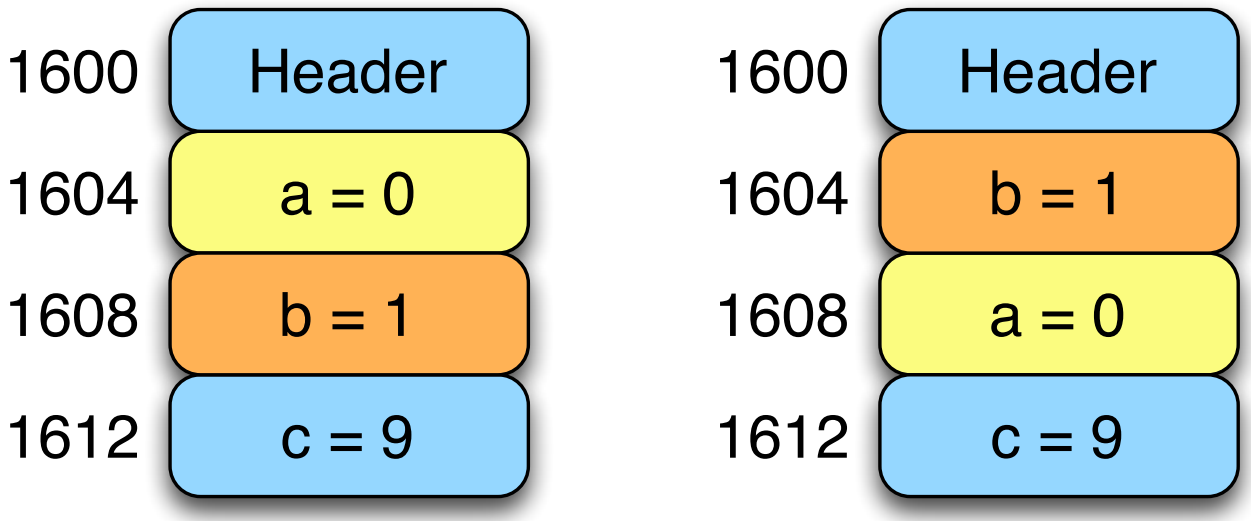
Some OO languages (e.g. C++, but not Java) allow **multiple inheritance**.

```
class A {
   int a = 0; }

class B {
   int b = 2; }

class C extends A, B {
   int c = 9; }
```

Now we have two possibilities for laying out objects that are instances of `C` in memory.

Multiple inheritance

Now we have two possibilities for laying out objects that are instances of `C` in memory.

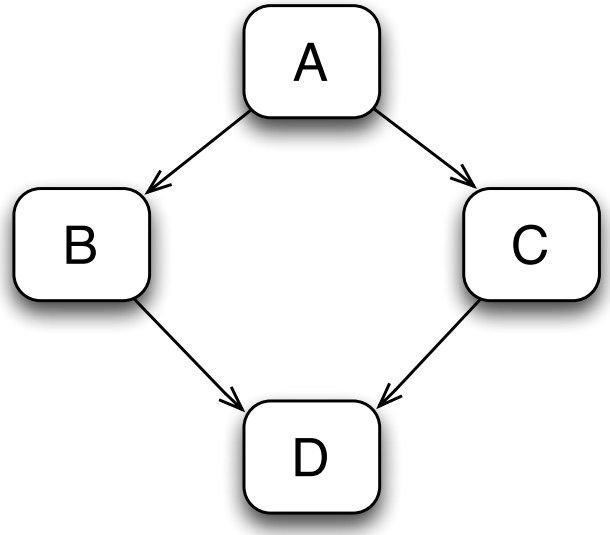


Either way is fine, as long as we always use the same choice!

Multiple inheritance: diamond inheritance

However with multiple inheritance the compiler must must be careful because attributes/instance variables and methods can be inherited more than once:

```
class A { int a = 0; }

class B extends A{
   int a = 2; }

class C extends A {
   int a = 9; }

class D extends B, C { int a = 11; ... }
```



Should `D` contain `a` once, twice, thrice, four or five times? To avoid such complications, Java and other languages prohibit multiple inheritance.

Quick question

Languages like Java have visibility restrictions (`private`, `protected`, `public`).

How does the code generator handle those?

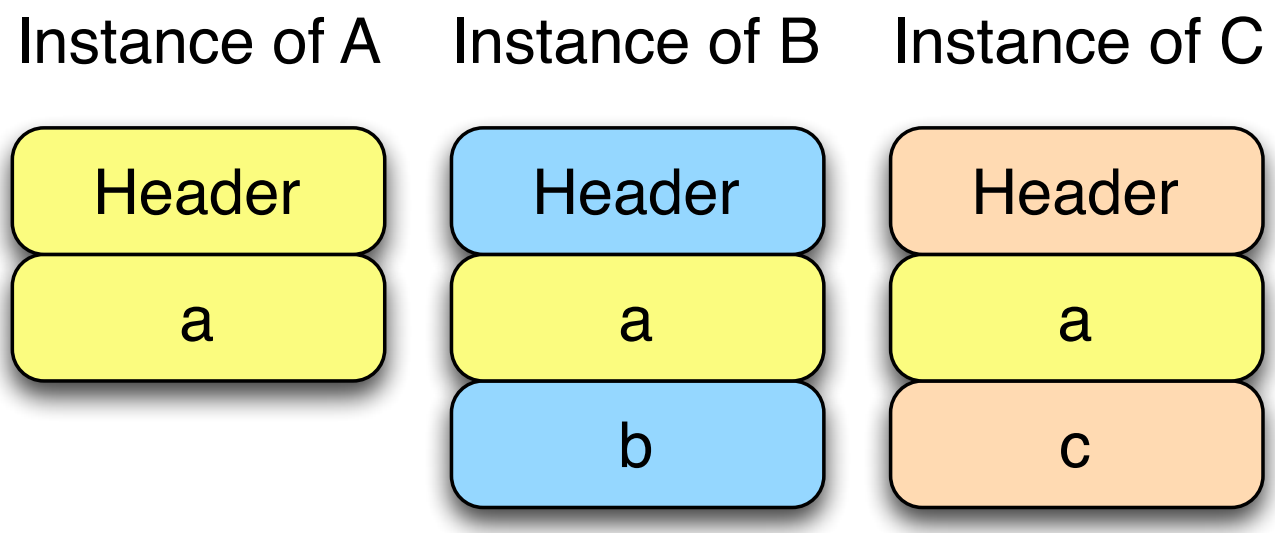
Answer: not at all, they are enforced by semantic analysis (type checking).

Summary

Inheritance relationships

```
class A { a ... }
class B extends A { b ... }
class C extends A { c ... }
```

give rise to the following object layouts.



Note that we can access `a` in the same way in instances of `A`, `B` and `C` just by using the offset from the top of the (contiguous memory region representing the) object.

Methods

We have now learned how to deal with object instance variables/attributes, what about methods? We need to deal with two questions:

- How to generate the code for the method body?
- Where/how to store method code to ensure dynamic dispatch works?

We begin with the former.

Compilation of method bodies

We have already learned how to generate code for procedures (static methods). Clearly (non-static) methods are very similar to procedures ... except:

Which method to invoke?

Can we reuse the code generator for methods?

Compilation of methods by reduction to procedures

Consider the following Java definition:

```
class A {
   int n = 10;
   int f ( int x ) = { n = n+1; return x+n; } }
```

What's the difference between

```
a.f(7)
```

and

```
f(a, 7)
```

Compilation of methods by reduction to procedures

We see a method invocation `a.f(7)` as a **normal procedure call** taking **two** arguments, with the additional argument being (a pointer to) the object `a` that we invoke the method on. The additional argument's name is hardcoded (to e.g. `this`).

```
int f_A ( A this, int x ) = {
   this.n = this.n + 1;
   return x + this.n }
```

So 'under the hood' the compiler generates a procedure `f_A` for each method `f` in each class `A`. The object (`this` in Java) becomes nothing but normal a procedure parameter in `f_A`. Each access to a instance variable `n` in the body of `f` is converted to an access `a.n` to the field holding `b` in the contiguous memory representing the object. Now we can **reuse** the code generator for procedures, with one caveat.

Where does the method body code go?

The only two issues left to resolve are

- How to find the actual method body?
- Where to store method bodies?

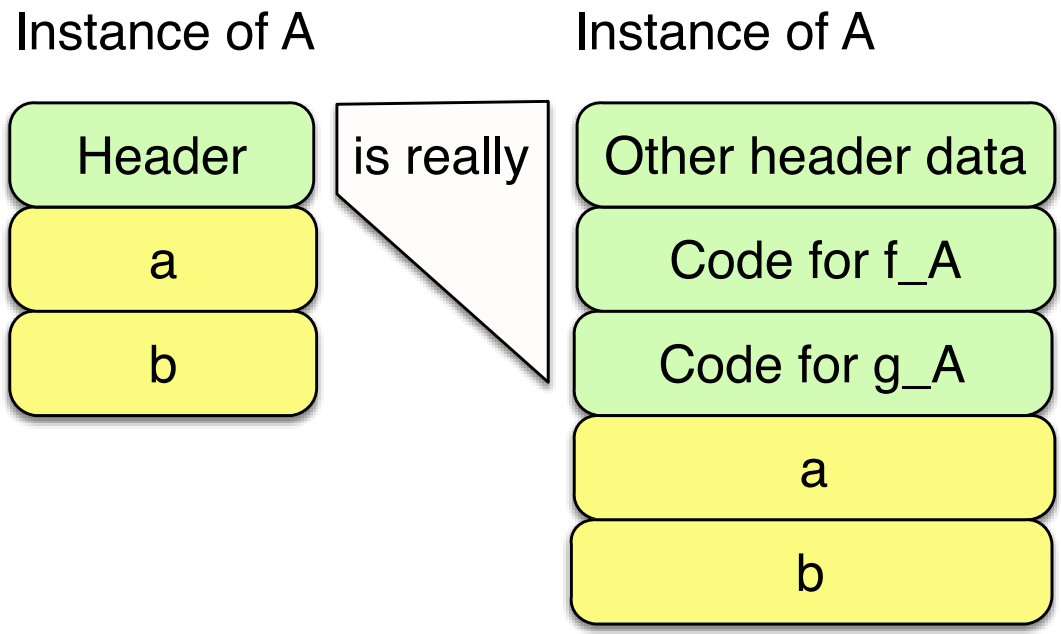
Any ideas?

Finding methods is easy: just access them (like fields) at *fixed offset* from the header, known at compile-time.

Where does the method body code go? First idea

Put them all in the contiguous memory with the instance variables/attributes.

```
class A {
 int a = 0;
 int b = 1;
 int f () = ...
 int g ( int x ) =
   ...
```



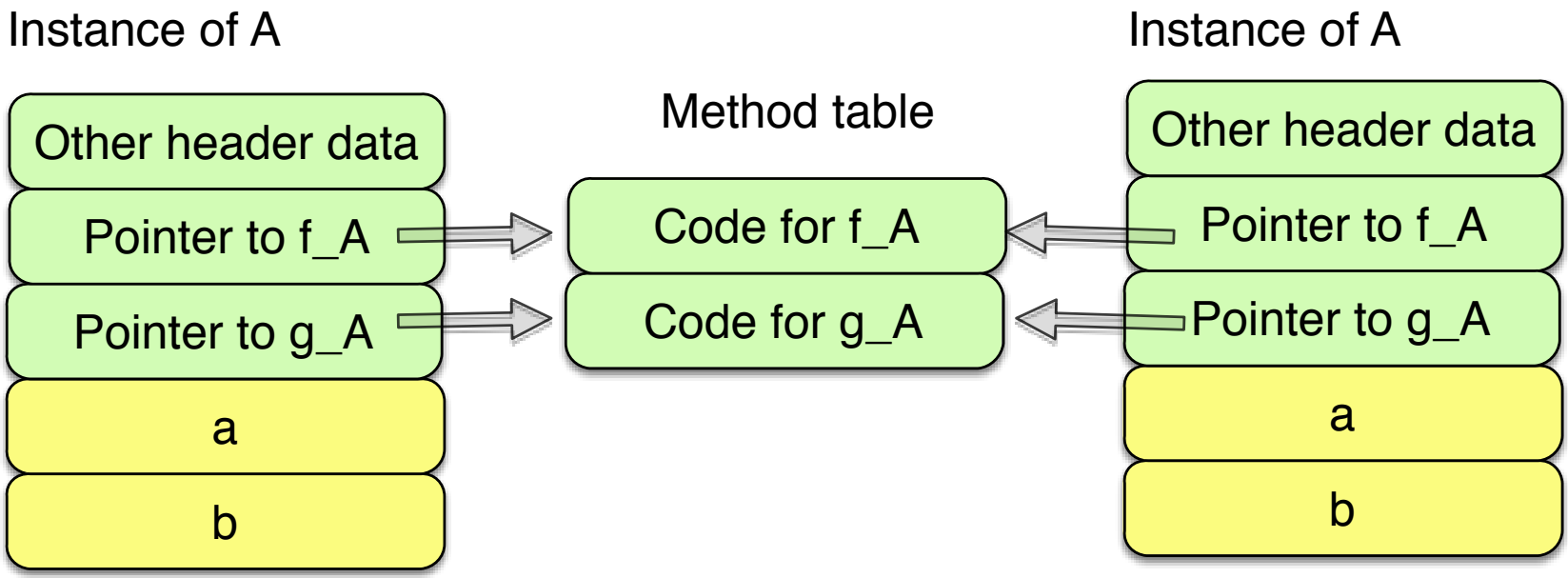
Note that `f_A` and `g_A` are normal procedures with an additional argument as described above.

Can you see the problem with this solution?

Where does the method body code go? Second idea

Problem: massive code duplication! But methods are the same for each object of the same class.

Instead we could share the method bodies between objects, and let the instances (at **fixed offset**) contain pointers the shared method bodies.

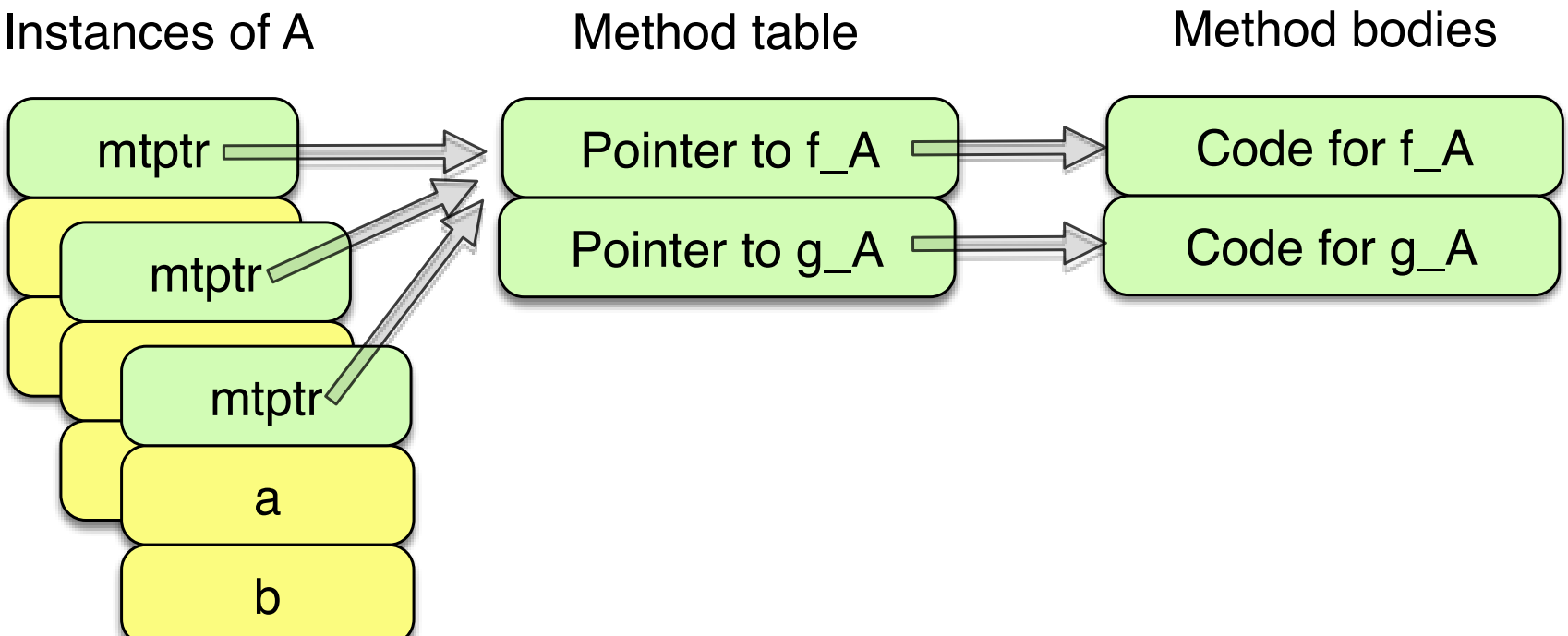


Can you see the problem with this solution?

Where does the method body code go? Third idea

The problem with the second approach is that we are still wasting memory: imagine a class with 100 methods, and 1.000.000 instances. With 64 bit pointers that's a whopping 762 MBytes! Wasteful.

A much better solution is to add a layer of indirection, and have just one pointer (at **fixed offset**) per object to a table that contains (at **fixed offset**) a pointer to each method's code (remaining headers omitted for readability).



Where does the method body code go? Third idea

This is the choice taken in practice as far as I know, because it is much more memory efficient.

The main disadvantage is that the 'pointer chasing' in the invocation of a method is (slightly) more time-consuming than with the first two ideas.

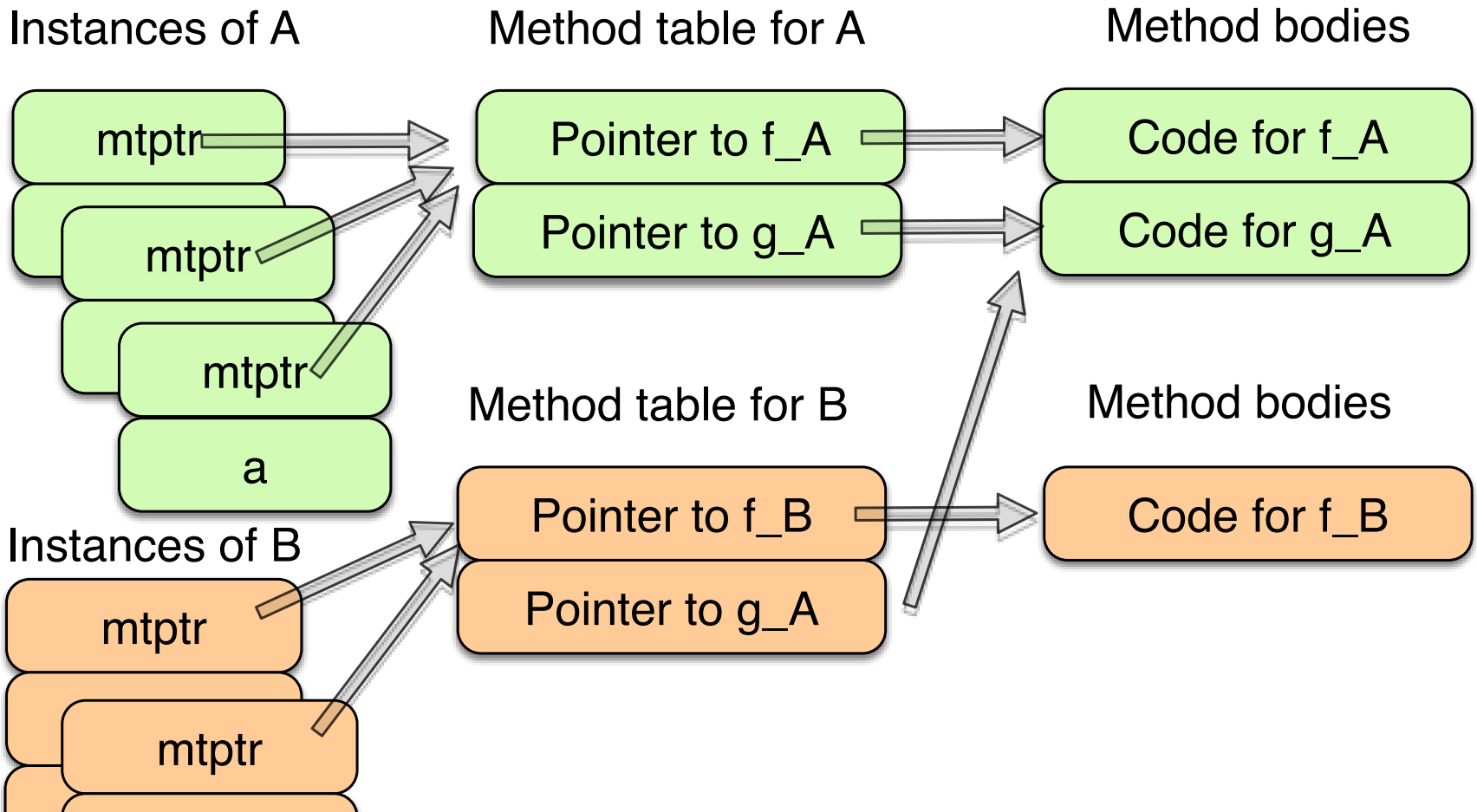
Just-in-time compilers (e.g. the JVM, Microsoft's CLR, Chrome's V8) employ clever tricks to ameliorate this shortcoming.

Inheritance and the third idea

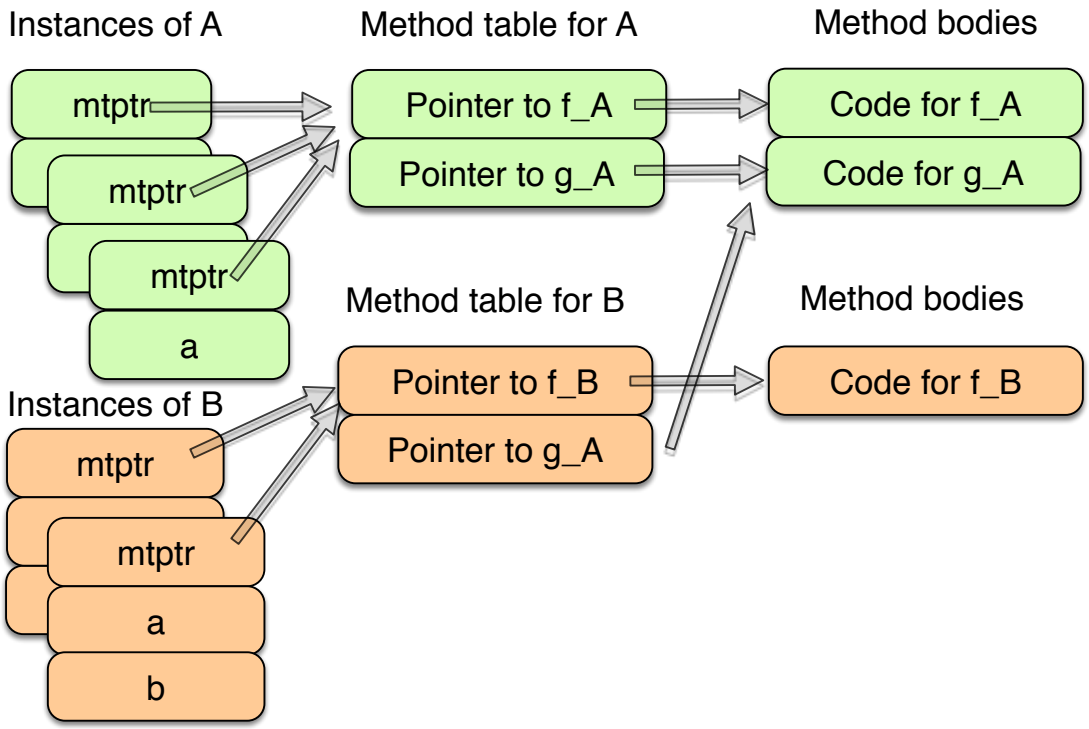
Does this also work when inheritance is involved?

```
class A {
  int a = 0;
  int f ( int x ) {...}
  int g ( int x ) {...} }
```

```
class B extends A {
  int b = 7;
  int f ( int x ) {...} }
```



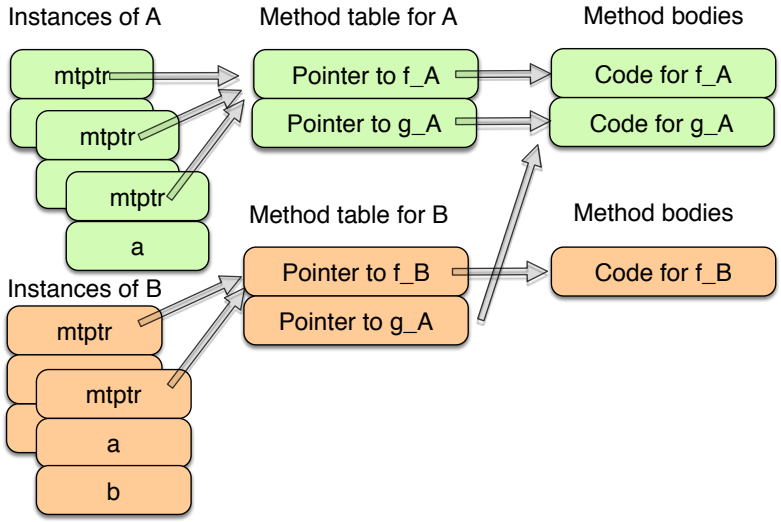
Inheritance and the third idea



The key insight (as with the layout of the object in contiguous memory) is that the pointer to the method table is always at fixed offset from the top of the object, and pointers to method bodies are always at fixed offset in the method table, e.g.:

- `f` is always at offset 0 in the method table.
- `g` is always at offset 4 in the method table (assuming 32 bit pointers.

Translation of method invocation



To translate `a.f(3)` where, for example, `a` has type `A` at compile time, but points to an instance of `B` at run-time, we do the following.

- Get the pointer `mptr` to the method table for the current instance of `a` (find by fixed offset available at compile time).
- In the method table find the pointer `p` to the body of `f_B` (also by fixed offset available at compile time).
- Create the AR just like for any other procedure, supplying a pointer to the object `a` as first argument.
- Jump to `p` the body of the procedure `f_B`.

Example

What does this program print out?

```
class A {
  void f () { System.out.println ( "A::f" ); }
  void g () { System.out.println ( "A::g" ); } }
class B extends A {
  void f () { System.out.println ( "B::f" ); } }
class Main {
  public static void main ( String [] args ) {
    A a = new A ();
    B b = new B ();
    A ab = new B ();
    a.f ();
    a.g ();
    b.f ();
    b.g ();
    ab.f ();
    ab.g (); } }
```

(Example: prog/ex4.java)

Reflection

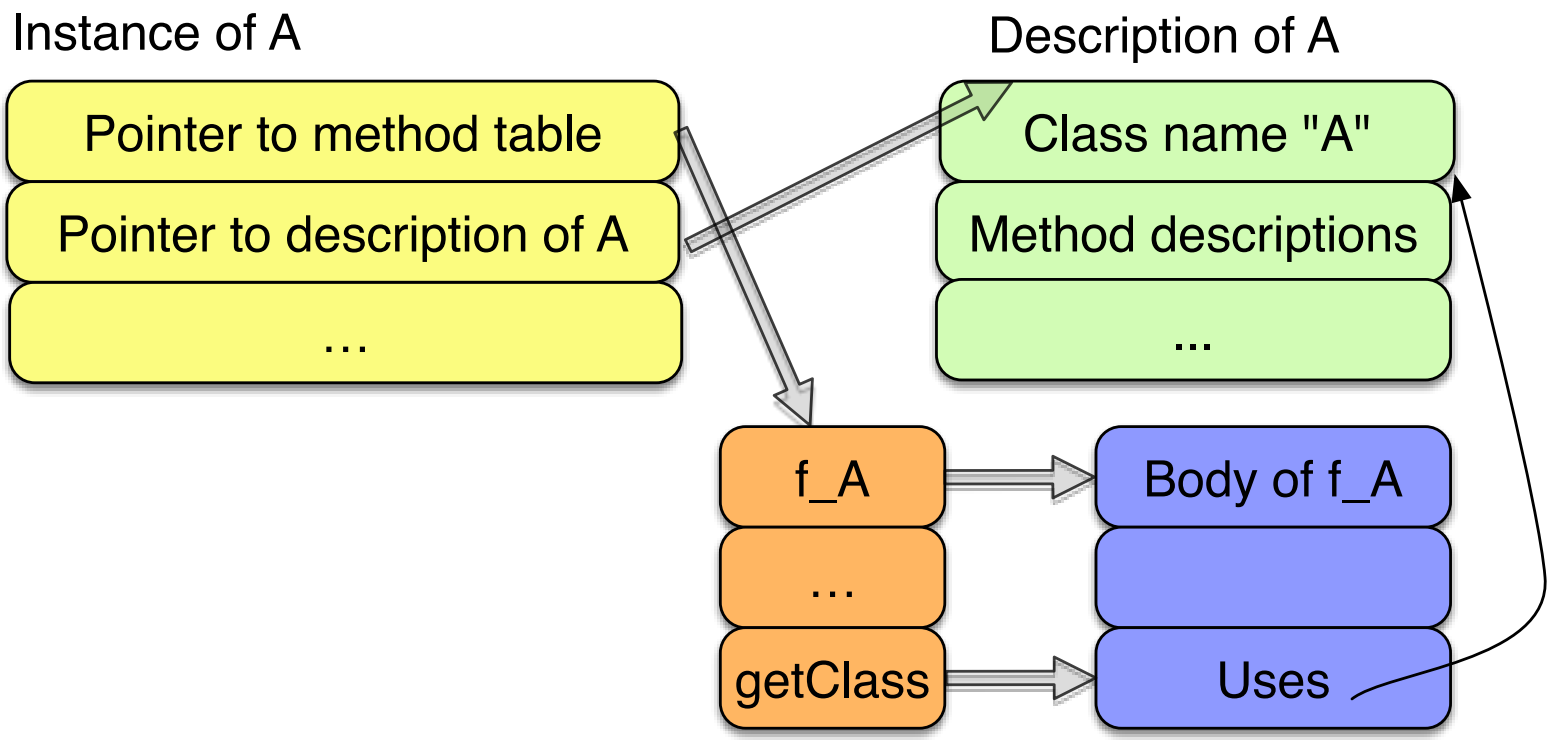
Languages like Java and Python enable **reflection**, that gives you information about the type of an object at run-time. You can do things like this:

```
import java.lang.reflect.Method;
class A {}
class B extends A {}
class Main {
    public static void main ( String [] args ) {
A a = new A ();
B b = new B ();
A ab = new B ();
try {
    System.out.println ( a.getClass().getName() );
    System.out.println ( b.getClass().getName() );
    System.out.println ( ab.getClass().getName() ); }
catch (Exception ioe){ System.out.println(ioe); } } }
```

What does this return? (Example: prog/refl.java)

Reflection

This can be implemented by creating a memory area at **fixed offset**, containing a description of each class, and each methods for reflection containing pointers to that description.



Alternatively, make all classes inherit from base class with suitable `getClassName` method or similar. Overwrite the `getClassName` method for each class (ideally automatically).

OO a la Javascript

We have learned how object oriented languages like Java are translated. They exhibit **class-based** object orientation.

There is also **prototype-based** object orientation. This is used in Javascript. The ideas seen here can be adapted to prototype-based object orientation.