Object-Oriented Languages

Parsing and type-checking an object-oriented language should be fairly straightforward. Just as with procedural languages like BPL, the symbol table follows the tree structure. If you have class Foo and a reference to one of the *properties* of Foo (either a method or a variable), the declaration of that property needs to be found either among the declarations of Foo or among the declarations of one of the ancestors of Foo in the class hierarchy.

If class Bar extends class Foo and we assign an object of class Bar to a variable declared to have type Foo:

```
Foo P = new Bar()
```

(which of course we can do since every object of class Bar is also an object of class Foo, just as all Students are also Persons)

then any property P.x we refer to needs to be a property of class Foo.

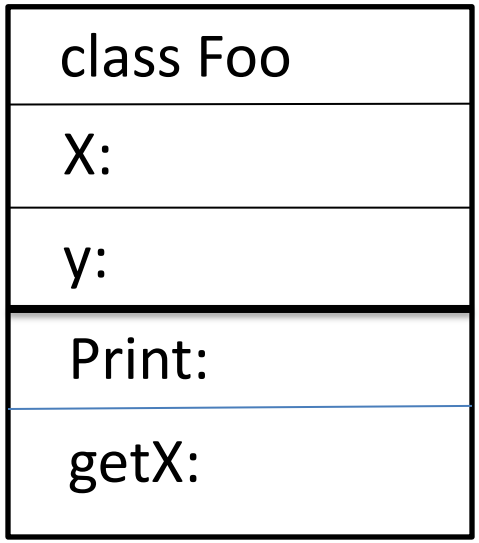
The interesting issues with compiling object-oriented languages occur in the code generation stage.

An object of a class is represented by an *Object-Record*, which has storage for each property of the class, both the instance variables and the methods. If the language does not allow the methods of a class to be modified the Object-Record might contain only the instance variables and a pointer to a static block of the methods, but for consistency I'll show records with both variables and methods.

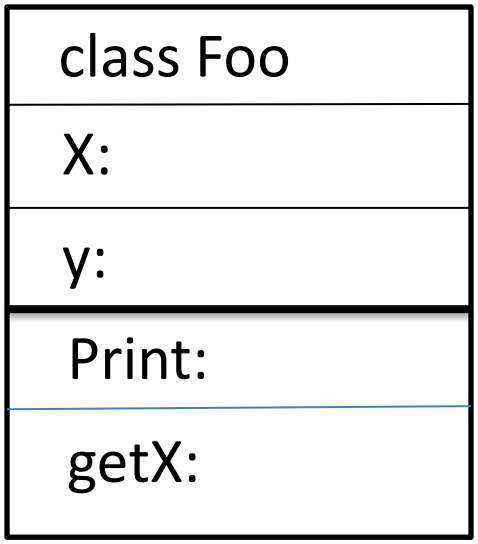
For example, with class declations

```
class Foo {
        int x, y;
        void Print() {
        }
        int getX() {
        }
}
```

the Object-Record might be



In this record "Foo" is an enumerated type so the object knows its native class, "x" and "y" are storage locations, and "Print"and "Foo" are the locations of those functions in the code file.



In this record the offsets of the instance variables from the start of the record are known at compile-time, so if the getX() method wants to return the value of instance variable x we know how to generate code to find that value.

The value of any object is just the address of its Object-Record.

Let's suppose class Foo also contains a setter method for instance variable y:

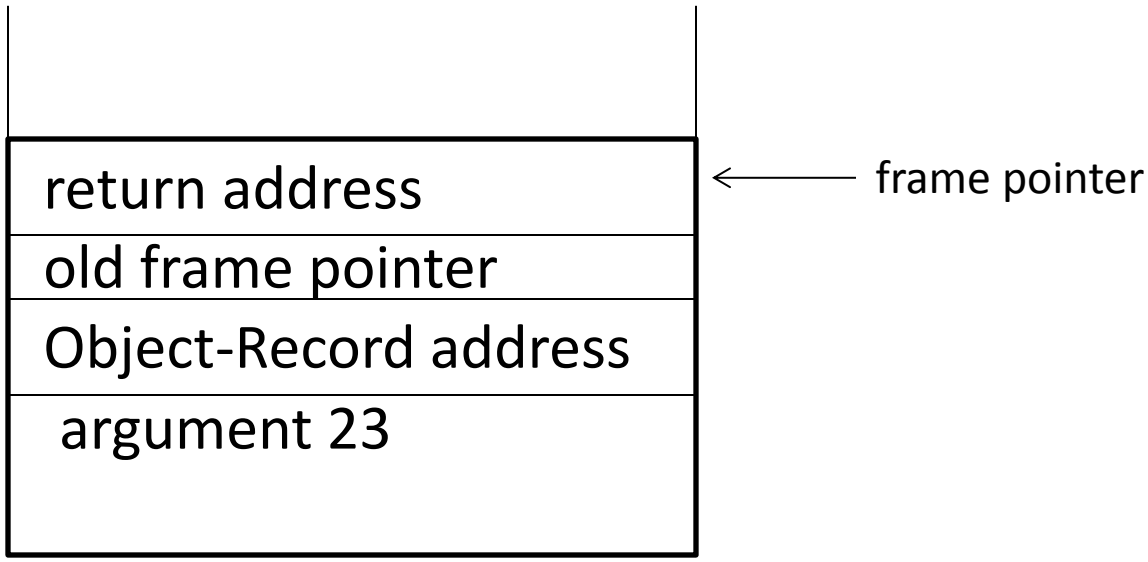
```
void setY(int a) {
        y = a;
}
```

and that we call this method:

```
Foo P = new Foo();
P.setY(23);
```

The code generated for the method call pushes the value of P (the address of its Object-Record) onto the stack as the unlisted first argument for the call.

This leaves the stack frame for the call looking like:



The code we generate puts the Object-Record address (a known offset from fp) into the accumulator, adds the offset of y from the start of the Object-Record, and uses this as the destination address for the assignment. The argument 23, also at a known offset from fp, is moved to this destination.

The code for the caller in P.setY(23) does the following:

- push 23 onto the stack
- push the value of P (the address of its Object-Record) onto the stack.
- push the current frame pointer onto the stack
- put the value of P into the accumulator
- add the appropriate increment, known at compile-time, to get the address of the setY method. Use this as the address of the call.

Python's notation for the method and its call reflects this runtime structure:

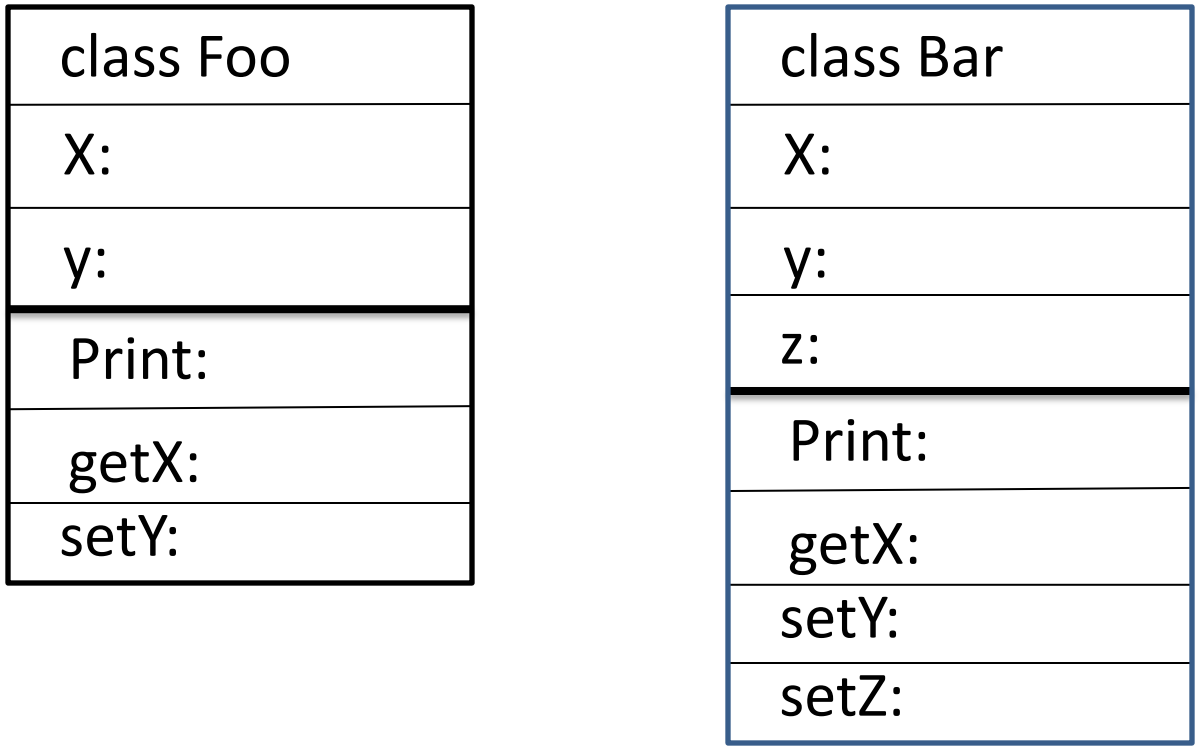
```
class Foo:
        def setY(self, a):
                self.y = a

def main( ):
        P = Foo()
        P.setY(23)
```

That mysterious argument *self* is Python's way of referring to the Object-Record for variable P.

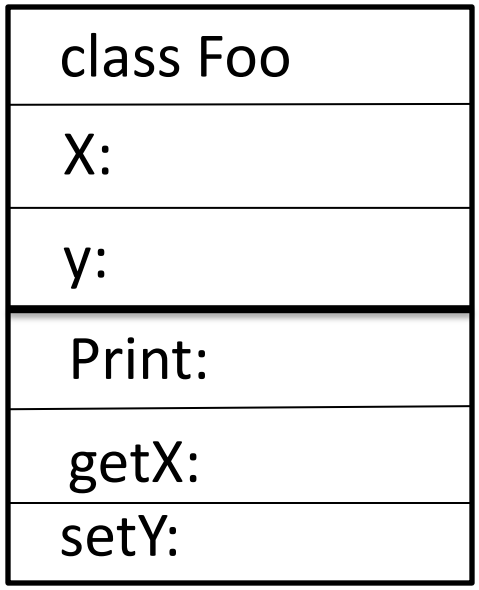
The Object-Record for an object of a subclass needs to include all of the inherited properties as well as those defined within the subclass. For example, if class Foo has instance variables x and y and subclass Bar adds instance variable z, an object of class Bar has all three variables. Since we are storing the values of these variables in the Object-Record, that record must have room for all 3.

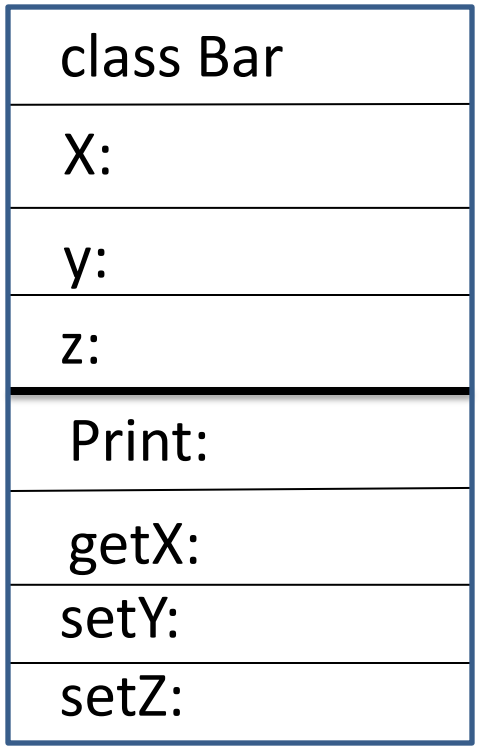
Since we need to know the offsets of each property in the Object-Record at compile time, the variables declared only in the subclass need to be listed after those declared in the parent class.

To extend our previous example, suppose class Foo has instance variables x and y, methods getX(), setX(), and Print(), and subclass Bar adds instance variable z and method setZ(). The Object-Records for objects of classes Foo and Bar are shown on the next slide:



This way an inherited method such as setY( ) can use the same offset for variable y for objects of both class Foo and class Bar.





We would use the same Object-Record structure for class Bar if it overrides the methods of class Foo: the method pointers would just point to the methods of class Bar rather than the methods of class Foo.