Implementing OOP

Mike Spivey
Michaelmas Term 2023

UNIVERSITY OF OXFORD

Department of COMPUTER SCIENCE

Object-oriented programming

- *Encapsulation*: the implementation of a class should be hidden from its users.
- *Object identity*: each instance of a class should have a distinct identity, so that multiple instances can co-exist.
- *Polymorphism*: if several classes have the same interface, instances of them can be used interchangeably.

Inheritance is not so important.

About Oberon

Niklaus Wirth's language Oberon (and its successor Oberon–2) are interesting because OOP ideas emerge from a combination of other language features.

- We'll use Oberon syntax in this lecture, but the mechanisms apply to other OO languages too.

Object identity

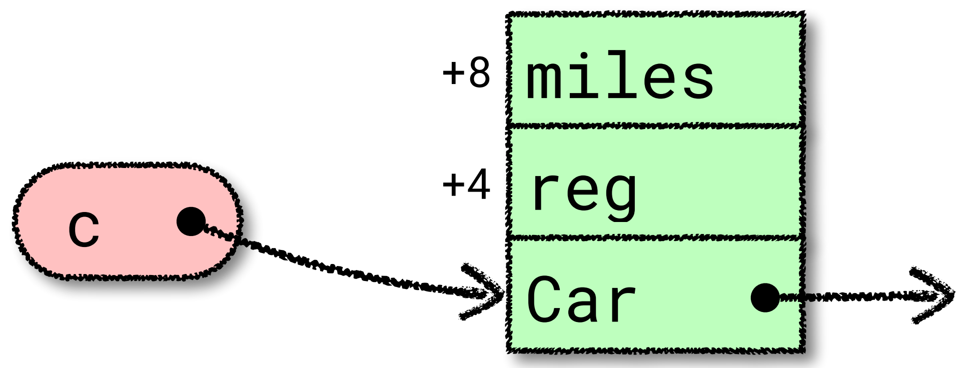
Each object can be stored as a heap-allocated record, and we can use the address of the record as its identity.

```
type Car = pointer to CarRec;
  CarRec =
    record reg, miles: integer end;

var c: Car;

new(c)
```

In many languages, the pointers are implicit.

Instance variables

Each object has fields for its instance variables, and also knows what class it belongs to: a pointer to a *class descriptor*, shared among all instances.



an object

Access to instance variables

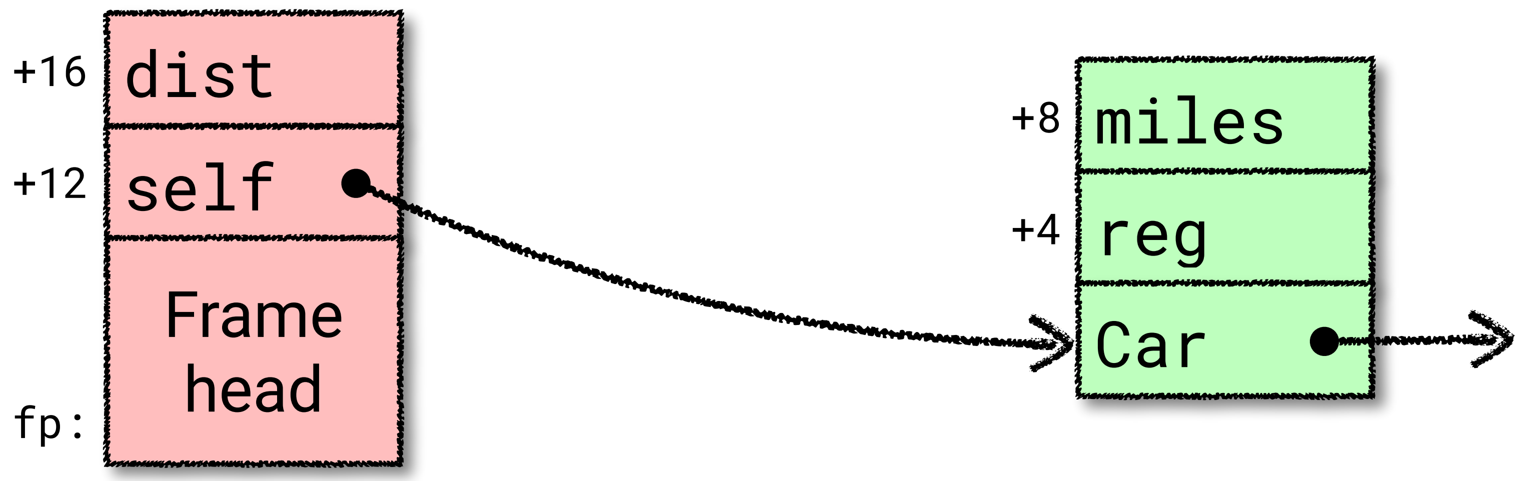
Methods can refer to instance variables of the object.

```
proc (self: Car) drive(dist: integer);
begin
   self.miles := self.miles + dist
end;
```

Many languages make the name `self` or `this` implicit, and allow the assignment to be written `miles := miles + dist`.

self.miles := self.miles + dist

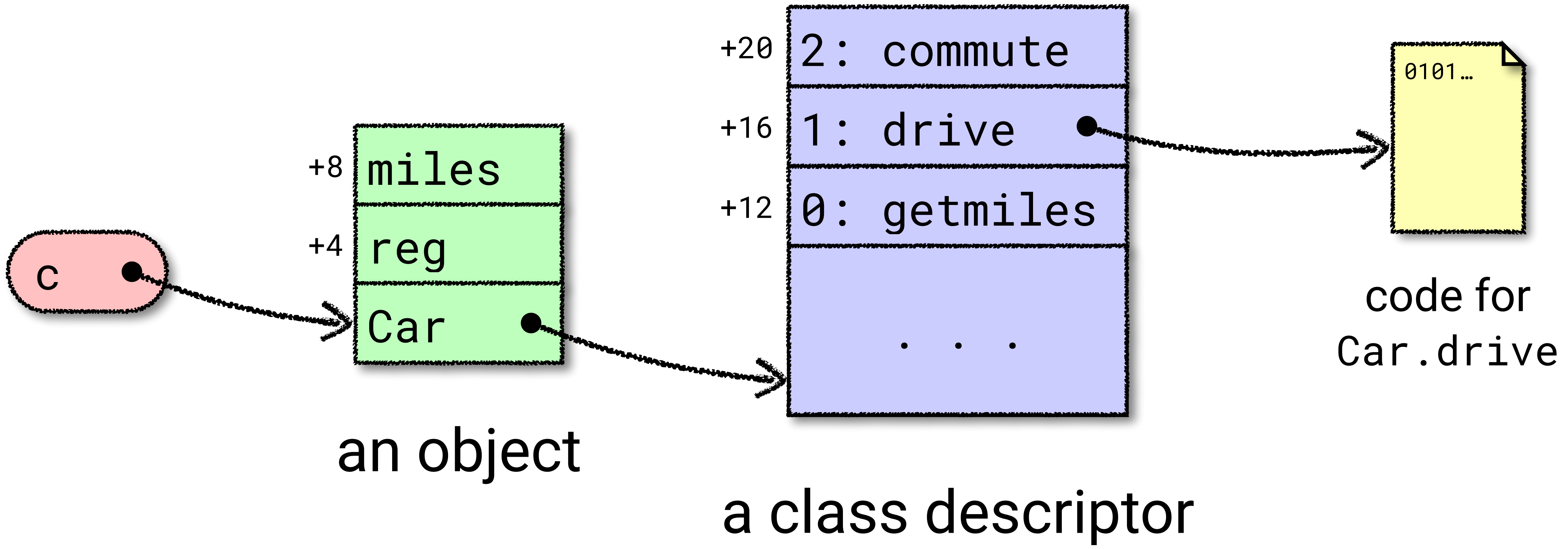
```
LDLW 12      ! self
LDNW 8       ! .miles
LDLW 16      ! dist
PLUS         ! +
LDLW 12      ! self
STNW 8       ! .miles :=
```



Virtual method tables

Each class has a *vtable* showing methods it supports, with a pointer to code for each of them.

Each method has a known offset in the vtable.



Method invocation

`c.drive(100)` is implemented like

`c.class.vtable[1](c,100)`:

```
CONST 100      ! 100
LDGW _c        ! c
DUP            ! duplicate
LOADW          ! .class
LDNW 16        ! .vtable[1]
CALL 2         ! (c,100)
```

(If methods are not nested, there is no need for static links.)

A subtlety

If a car collector should write

```
    garage[i++].drive(5)
```

we must avoid evaluating `garage[i++]` twice.

Hence the DUP: or if compiling for a register machine, we could put the object pointer in a temp.

Using a temp

```
c.drive(100)
```

```
⟨AFTER,
    ⟨DEFTEMP 1, ⟨LOADW, ⟨GLOBAL _c⟩⟩⟩,
    ⟨CALL 2,
        ⟨LOADW,
            ⟨OFFSET, ⟨LOADW, ⟨TEMP 1⟩⟩, ⟨CONST 16⟩⟩⟩,
        ⟨ARG 0, ⟨TEMP 1⟩⟩,
        ⟨ARG 1, ⟨CONST 100⟩⟩⟩⟩
```

As machine code

⟨*DEFTEMP* 1, ⟨*LOADW*, ⟨*GLOBAL* _v⟩⟩⟩

```
    ldr r0, =_c
    ldr r4, [r0]
```

⟨*ARG* 1, ⟨*CONST* 100⟩⟩

```
    mov r1, #100
```

⟨*ARG* 0, ⟨*TEMP* 1⟩⟩

```
    mov r0, r4
```

⟨*CALL* 2, ⟨*LOADW*,
⟨*OFFSET*, ⟨*LOADW*, ⟨*TEMP* 1⟩⟩, ⟨*CONST* 16⟩⟩⟩⟩

```
    ldr r3, [r4]
    ldr r3, [r3, #16]
    blx r3
```

Encapsulation

For languages compiled to machine code, encapsulation can be enforced as part of semantic analysis.

A class has a small table of instance variables and methods, each marked as public or private, so the rules can be checked at each use.

But generally there is no protection mechanism at the machine level.

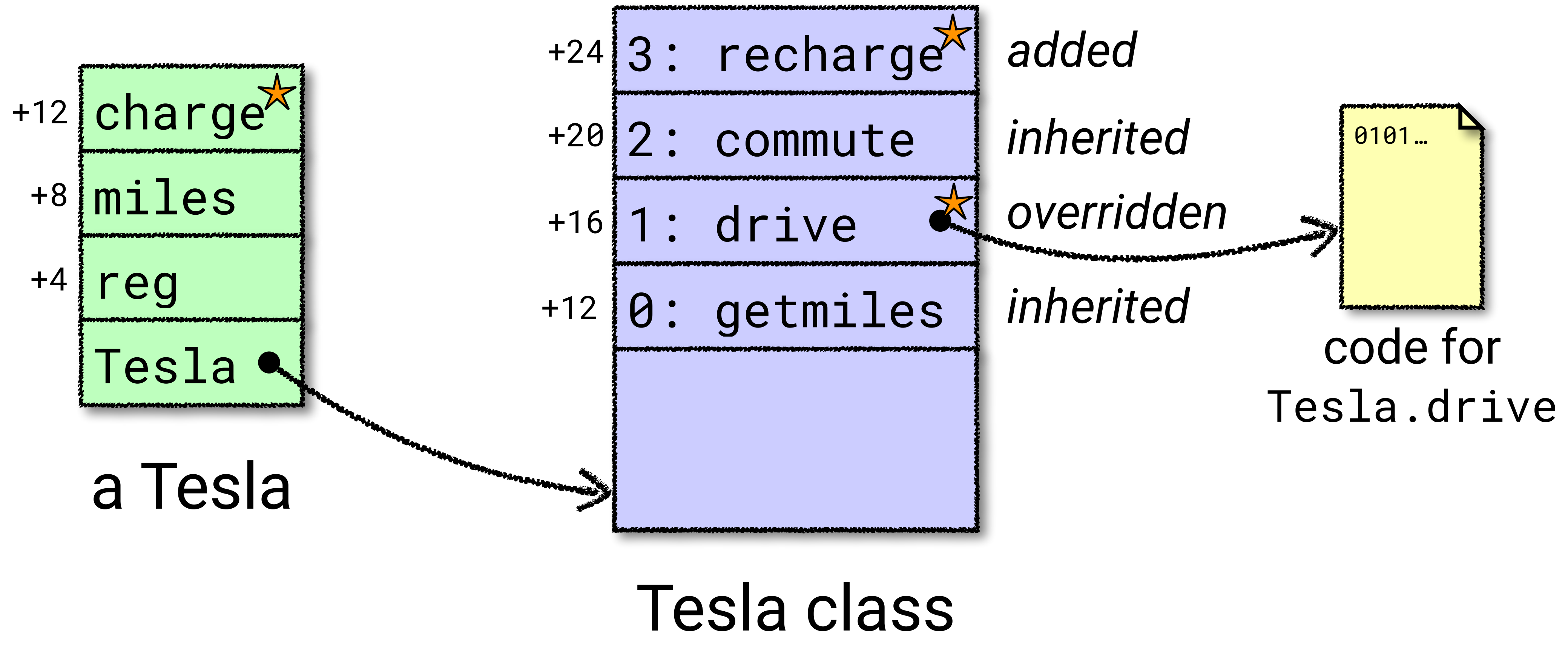
Subclasses

```
type Tesla = pointer to TeslaRec;
   TeslaRec =
      record (CarRec) charge: integer end;

proc (self: Tesla) drive(dist: int); …

proc (self: Tesla) recharge();
   begin self.charge := 100 end;
```

- Instance variables inherited and more added.
- Some methods inherited, some overridden, and some added

Implementing subclasses

Instances of a subclass add new instance variables at the end, and new methods go at the end of the vtable.



Dynamic dispatch

All vehicles use a consistent vtable index 1 for the `drive` method.

```
var c: Car; t: Tesla;

new(t);

c := t
```

The call `c.drive(10)` will correctly invoke the `Tesla` version of `drive`.

And `c.getmiles()` will use the `Car` version of `getmiles`.

Access to instance variables

However it is invoked, the `getmiles()` method in `Car` correctly returns the value of `miles`, even if the receiver is actually an instance of `Tesla`.

```
proc (self: Car) getmiles();
  begin return self.miles end;
```

This works because the `Car` part of a `Tesla`’s instance variables is laid out in the same way as an ordinary `Car`.

Late method binding

If `Car.commute()` is defined as

```
proc (self: Car) commute();
begin
   for i := 1 to 10 do
      self.drive(50)
   end
end
```

then invoking `commute` with a `Tesla` as receiver will use the `Tesla` version of the `drive` method: this is *late method binding*.

Fragile base class problem

The net effect of `commute()` invoked on a `Car` object is to increase `miles` by 500.

So what if we, or a compiler, ‘optimised’ it to

```
   self.miles := self.miles + 500?
```

What does that do when `commute()` is invoked on a `Tesla` object?

Conclusion: inheritance breaks encapsulation.

Super calls

Writing `Tesla.drive` as

```
proc (self: Tesla) drive(dist: integer);
begin
   self.drive↑(dist);
   self.charge := self.charge - 5*dist
end;
```

uses a *super call*, implemented as a static call to `Car.drive`.

- Still pass `self` as a parameter, so calls in the superclass method can use dynamic dispatch.

Fragile binary interface problem

Changes to `Car`, even if they don't affect its public interface (e.g., adding a new private method) will require all subclasses like `Tesla` to be recompiled, because the vtable layout can change.

This is unacceptable for languages like Java where code is collected from all over the web. That's why the JVM delays laying out vtables until the classes are loaded.

Type tests and casts

`v is Car`: true if `v` is a `Car` or `Tesla`, false if any other kind of `Vehicle`.

Implementation: each class knows its level and its list of ancestors.

`Vehicle`	0	`Vehicle`
`Car`	1	`Vehicle`, `Car`
`Tesla`	2	`Vehicle`, `Car`, `Tesla`

Use `(level >= 1 & ancestor[1] = Car)` without any need to search the ancestor chain.

In conclusion

With single inheritance: method dispatch, type tests and access to instance variables can all be compiled with fixed cost. But …

- dynamic dispatch has a hidden cost in mispredicted branches.
- multiple inheritance, as in C++, makes the picture (much) more complicated.

Scala's *traits* are implemented by flattening the program at compile time, then using single inheritance.