Object-Oriented and Prototype-based programming

Prelude

This guide/tutorial is about Object-oriented and prototype-based programming. This guide includes general knowledge about object-oriented paradigm and tries to clear out some misunderstandings or misconceptions programmers may have about OOP.

This guide was originally called : "You don't know OOP", but it seems pretentious since this guide was roughly made without any correction and possibly many spelling mistakes. This is not really pollish, but it is an honest effort to share my view/knowledge on the subject and I hope people learn from it. This guide attempts to propose a good intuition about OOP without getting into fancy words.

Object-oriented in general

As the name suggest, object-oriented developpement use objects to create programs. You may ask yourself what is an object? It may seems like a stupid question but it is not. I'm sure many programmers (even experienced one) don't have a clear definition in their head.

Well, I will give you a simple one : objects are the runtime entities.

- They are the living things when your program runs (a.k.a instances)
- They have an internal state (instance variables)
- They can respond to certain message (their behavior)
- Their state can be mutated via message sending.

You might ask : *what is message sending?*

Well it is simple : method invokation.

```
var x = new Foo();
x.bar();    # We send the message `bar` to the object `x`.
```

An important notion to grasp is to differentiate function call (*à la Lisp*) and message sending.

In non-object oriented language like C, we have function call, like so:

```
#include <stdio.h>
void bar(void) { printf("calling `bar`\n"); }

int main() {
    bar(); // function call.
    return 0;
}
```

In non-OOP languages, function definition provides two things :

1. A signature;
2. an implementation.

In the C program above, when we call `bar`, we know the behavior beforehand.

However, in OOP languages, *methods* split the signature from the implementation, ie given a signature we can't know for sure which implementation (the body of the method) will be executed.

This is possible since object-oriented provides subtype polymorphism which grant us the ability to delayed the implementation resolution at runtime.

Here's an example of message sending :

```java
class Main {

  static class A {
    void foo() { System.out.println("A::foo"); }
  }

  static class B extends A {
    @Override
    void foo() { System.out.println("B::foo"); }
  }

  public static void main(String[] args) {
    A b1 = new B(); // b1 isa `B` at runtime but statically typed as an `A`.
    b1.foo();   // Will print `B::foo`.
  }
}
```

Here, given the method `foo` introduced by `A`, when we send the message `foo` to an instance `b1` typed by `A`, `B::foo` will be executed. In other words, even though we know the signature of `foo` we can't be sure which code will be executed. The signature `foo` isn't bound to a particular implementation. How Java knows which function to call? Well, this is done at runtime (late-binding) through an indirection table called the virtual function table (vft).

late-binding : the binding or the "function retrieval" is done late, a.k.a at runtime, a.k.a during the program execution.

NOTE: virtual function table are irrelevant to understand OOP concepts.

Wtf is subtype polymorphism?

In any bad school in the universe, we will give you the following half-ass definition:

Polymorphism : "*poly*" is for "*many*" and morphism for "*forms*", thus "many-forms".

However, the definition is pretty f@!cking useless considering there's many kind of polymorphism. But at school and in OOP, when we talk about "polymorphism" we mainly refer to subtype polymorphism.

Subtype polymorphism is the late-binding I previously talked about, ie method overriding. Furthermore, this kind of polymorphism is done at *runtime* (a.k.a which implementation to invoke?).

If you limit the definition of subtype polymorphism to methods, it becomes simpler to manage (IMO). Otherwise, we would need to talk about classes : given a class `A` and its set of direct and undirect children `Cs`, the type of `A` might refer to any child (direct or indirect) of `A`. However, I don't like to talk about classes since at the end of the day, the unpredictable behavior of method invokation is done by overriding.

Here's a non-exhaustive list of other kinds of polymorphism :

- parametric polymorphism ~ a.k.a generics

- ad hoc polymorphism ~ a.k.a overloading (the most missed used and badly understood among the community).

An interesting fact about parametric and ad hoc polymorphism is they are both *static* polymorphism which mean they are resolved at compile time.

IMPORTANT : subtyping doesn't equal inheritance.
IMPORTANT2: subtyping ~ specialization (it means the same thing)

Wtf is inheritance?

Inheritance is only the mechanism by which we reuse code. Many non-OOP languages have that feature, like Haskell (through Typeclasses) or Rust (through Traits). Even some old language use *implicit* inheritence where two classes who lookalike are implicitly related to each other.

This distinction is blurred since in OOP, in order to achieve a subtyping relationship between two classes, we must use inheritance. ie the inheritence of methods and attributes is the consequence of subtyping.

Conclusion

To conclude this chapter, here's the main idea to keep in mind :

Object-oriented paradigm is **all about objects and message sending**.

- Objects : living entities when the program runs, a.k.a your instances.
- Message sending : method invokation where the signature isn't bound to a particular implementation.

Metamodel of Class based OOP

Here I will present the metamodel described in this article : DucourPrivat This part is hard to understand and to explain clearly. I recommand to read the original article and this guide a couple of times to fully absorb these new concepts.

Global and local properties

```
class A
    fun foo: Int do return 1
end

class B
    super A
    redef fun foo do return 2
end

class C
    super A
    redef fun foo do return 3
end
```

This code snipet is written and NIT, it is fairly easy to understand.

The thing I want you to focus on is the method `foo`. We see in class `B` that `foo` is redefined over its ancestor `A::foo`. Same thing for class `C::foo`. In an informal way, we can say that `B::foo`, `C::foo` and `A::foo` are related to each other in some way. In addition, we can say that `A::foo` act as the "*parent*" of `B::foo` and `C::foo`, ie they are equivalentish.

Well, that's because the class `A` introduces a global property called `foo` where `B::foo` and `C::foo` are linked to. It is the first occurence of this property in the hierarchy, thus the global one.

Here are some useful definitions :

- Property – anything declared or defined inside a class, this includes attributes, methods definitions and redefinitions (overriding), etc.
- Global property – a property *introduced* by a class for the first time.
- Local property – any property defined in a class. They are *linked* to the class where they are defined and they *belong to* a global property. Every global property are local to the class they are defined in.

Given the definition of *local property*, we can say that `B::foo` and `C::foo` are local property of `B` and `C` respectively. Finally, these local properties belong to `A::foo`.

Here's a recap:

```
class A
    # Global + Local property of `A`.
    fun foo: Int do return 1
end

class B
    super A
    # Local property of `B`, belongs to `B::foo`
    redef fun foo do return 2
end

class C
    super A
    # Local property of `C`, belongs to `A::foo`
    redef fun foo do return 3
end
```

The metamodel of properties

If we were to create a class model (a.k.a meta-model) of all of this :

```
class Class
    var specilizes: Array[Class]    # Its parents
    var introduces: Array[GlobalProperty]
    var defines: Array[LocalProperty]
end

class GlobalProperty
end

class LocalProperty
    var belongs_to: GlobalProperty
    var redefines: nullable LocalProperty
end
```

Here, `LocalProperty::redefines` is a nullable field since the first introduction of a property do not redefine any other local properties.

Why overloading is badly understood

Overloading isn't bad per see, but most people doesn't know about the meta model I described earlier and about variance.

Variance

The variance in OOP-language describe the behavior of subtyping between types. We can find three types of variance :

1. Invariance : no variance at all
2. Covariance : subtyping follows the same direction of inheritance
3. Contravariance : subtyping goes in the opposite direction of inheritance.

Variance is usefull to understand generic subtyping rules and method redefinition rules. **NOTE**: We will not talk the variance on generic.

Covariance

```
class Food
end

class Grass
    super Food
end

class Animal
    fun eat(food: Food) do ...
end

class Cow
    super Animal
    super Food
    fun eat(food: Grass) do ...
end

var c1 = new Cow
c1.eat(new Grass) # OK!
# c1.eat(new Cow) # Compile Error : eat needs Grass not Cows
((Animal)c1).eat(new Cow) # COMPILE OK but will fail at runtime : TypeCast error.
```

If our language supports covariant method redefinition, then the global property `Animal::eat` can be redefine with subtype of `Food`.

Covariant method redefinition is useful the modelize real-world scenarios like the animals on a farm.

Covariant breaks the subtyping safety since we can upcast any class to one of its parent until we hit the global property with losser argument requirements.

This is caused by the fact `Cow::eat` is linked to the global `Animal::eat` which accepts any kind of `Food` instance.

Contravariant

```
class C
end

class D
    super C

    fun didi do ...
end

class A
    fun foo(d: D) do ...

    # returns an instance of D
    fun bar: D do ...
end

class B
    super A

    redef fun foo(d: C)
    do
        # Here we can't call `super` since
        # it would break the signature if `B`'s
        # superclass.
        ..
    end

    # Returns an instance of C
    redef fun bar: C do ...
end

var b1 = new B
var a1 = new A

a1.foo(new D)   # OK
b1.foo(new D)   # OK since D is subtype of C
b1.foo(new C)   # OK. So far so good..., but not for so long
var d1 = ((A)b1).bar, # Statically correct
d1.didi # ERROR ABORT : `d1` isn't an instance of `D`.
```

Here contravariance is safe over input type argument, however, as return type they are not safe.

Invariance

We've seen that covariance and contravariance are not safe. They may be useful, but most programming languages choose to keep invariant method redefinition.

So what is the problem with overloading?

Well, since most programming language are invariant over their redefinition AND they support overloading, it means this code works but in unexpected way :

```
class Food
```

```
end

class Grass
    super Food
end

class Animal
    fun eat(food: Food)
    do
        print "eating {food}..."
    end
end

class Cow
    super Animal
    super Food

    fun eat(food: Grass)
    do
        print "eating grass happily, because I'm a cow"
    end
end

var c1 = new Cow
c1.eat(new Grass) # OK, as expected
c1.eat(new Cow) # OK, What???
```

Since overloading is supported, `eat(Grass)` and `eat(Food)` does not belong to the same global property. They are two distinct global property with the same name.

Morever, the dispatch is done statically at compile time, ie the eat is disambiguated at compile time, a.k.a this is not subtype polymorphism.

NOTE: let me be clear on something… This is not unexpected behavior, however, it is unintuitive for most students and programmers. Most people forget about those rules and relearn it 30 times.

Conclusion

Overloading is mainly shit and not useful. What you really want from a class based language is covariant redefinition/multimethod like CLOS has. Some languages offer virtual type which emulate covariant redefinition.

Different kind of OOP

Among the mainstream programming language, we find two types of OOP: class-based and prototype-based.

Class based

Class based languages uses classes to describe the content and behavior of your objects. In most statically typed languages, classes do not live outside the compilation process.

Well, this is a lie. To be more precise, classe may live during the runtime. However, they are mostly used to store the virtual function table and other implementation detail information.

What I'm trying to say is the concept of a "class" only exist during your development and compilation process. At the end of the day, only objects live during the process.

This is kinda another lie since we could argue that dynamic language like Ruby unify class with objects (for dynamic metaprogramming). But let things be simple for a while. Metaprogramming in Ruby will be addressed in another guide.

So! Class are static beings. Onced you compile your program they disappear. They are blueprints for your objets. This is the case many languages like C++, NIT, Eiffel and OCaml (maybe) if I'm not mistaken.

Prototype-based

Remember at the beginning when I talked about OOP? Well I wrote OOP is only two things:

1. objects;
2. message sending.

Well, this is all about prototype-based language.

Just kinding, but seriously.... Prototype-based languages are only living objects.

SO....

For the rest of this guide we will talk exclusively about prototype-based language, because they are simple and beautiful IMO.

Prototype-based language for OOP

In this section I will talk about prototype-based language, how it works and how it relates to javascript.

Characteristics

Before with talk about javascript, we must talk about some characteristics of proto-based language.

1. They are mostly dynamic, in fact I think they are all dynamic (well I'm not aware of any kind of prototype-based language which is static). This is due to the fact that we have no class at all. Thus, we can't infer how our object will be layout or their behavior at compile time (it's really dynamic).
2. Prototype-based language must provide a way to create object from *ex nihilo*.
3. Most proto-based language adopts a "everything is an object" philosophy.
4. Method-lookup is implemented with delegation or some kind of concatenation.
5. Every object has a prototype-chain.
6. Every object can be the prototype of another object.

And I have no more characterics of proto-based that come to my mind.

Creating objects from *ex nihilo*

The expression *ex nihilo* means "*out of nothing*", without a constructor or a class. Prototype-based languages always provide a way to create them via a keyword or special syntax. In javascript, this is done via object literal construction.

```
var object1 = {
    "attr1" : 1,
    "attr2" : 2,
    ...
}
```

Here `object1` is a living object created at runtime from the curly braces object literal syntax.

NOTE: This is a similar syntax for the Lua language.

Why prototype-based?

Personnally, I would prefer calling it instance-based since everything must be created from living object or from *ex nihilo*. That being said, prototyping is somewhat related to the prototype design pattern where an initial object is created as "default" schema so that new instance can be cloned from it and customized to their needs.

Both cloned objects and the prototypes live their life and may be mutated.

In a language where don't have any notion of class, you must use some of your objects as "reference" which you derive other objects.

Here's an example in javascript without the `new` keyword:

```javascript
// my "reference" ~= prototype ~= class
var person = {
    this.age = 0,
    this.name = "default",
}

var p1 = {}
p1.__proto__ = person
console.log(p1.name)    // prints "default"
p1.name = "louis"
console.log(p1.name)    // prints "louis"
console.log(person.name) // prints "default"
```

As you can see I created two objects :

1. `person` serves as a "reference", a.k.a a prototype;
2. `p1` is an "instance" of `person` and "inherits" its behavior.

In other words, prototype is how you reuse behavior amongs objects in an object oriented language which is has no classes.

It's just stupid simple, it is literally the only natural way to achieve some classification and reusability with objects only.

Prototype chain

With the `__proto__` field we can build a chain of prototype the simulate class inheritance, like so :

```javascript
var A = {
    foo: function() {
        console.log("A::foo")
    },
}
var B = {
    bar: function() {
        console.log("B::bar")
    },
}
B.__proto__ = A
var C = {
    x: 0
}
C.__proto__ = C

// We create a new "instance" of `C`
var c1 = {}
c1.__proto__ = C

var class = c1.__proto__

// This loop will print out "{ x: 0 } , { bar: [Function] }, { foo: [Function] }, {}, "
// ie, it prints the content of each prototype (parent) in the prototype chain.
while(class != null) {
    console.log(class, ", ")
    class = class.__proto__
}
```

The `__proto__` field used in the previous example is used in Javascript as a way to do method lookup. When you send a message to an object, like `age`, the method/attribute is looked inside the targeted object for a property called `age`, if it fails to find the property, the method lookup goes up in prototype chain.

"Class" in javascript

Newer version of javascript comes with the keyword `class`, but keep in mind that javascript is a prototype-based language, thus it has no class.

As the specification of Mozilla says, the notion of `class` is only syntactic sugar for "simpler" code.

`this` keyword in javascript

The keyword `this` refers to the context and not the current object instance like in Java or any other OOP language.

By default javascript comes with a global object, called `window` in the browser or `global` in node. This global object is also known as the global *context*.

By default if you enter a function without binding the context, like in a top level function, then `this` refers to the global object :

```
function foo() {
    return this;
}

foo() == window;    // true
```

By default every function call must have a receiver, if no receiver is prefixed to the call like `object.foo`, then javascript implicitly add `this` before calling a function : `this.foo()`.

This is pretty standard in most OO language to have a global object and the implicit `this` binded to every function call, this is the case for Ruby and NIT too.

Due to the nature of object oriented nature, function call is replaced with message sending. By doing so, every message needs to be sent to an object which we call the receiver.

However, what is less commun is that any function can be bound to any other object at runtime:

```
function foo() {
    return this;
}

var x = {}
// `call` invoke a function with an extra argument: the value of `this`.
foo.call(x) == x; // true
```

Furthermore, you can derive any function by binding any object :

```
function foo() {
    return this;
}

var x = {}
// `call` invoke a function with an extra argument: the value of `this`.
var foo2 = foo.bind(x);
foo2 == foo // false
foo2() == x // true
```

The metamodel in prototype-based

The metamodel I previously presented is somewhat not really useful in the context of prototype-based language, since property lookup is done by traversing the entire prototype chain. In addition, objects in dynamic language can add, remove and override any inherited properties without respecting any contract or semantics previously established by its ancestors. Having a clean meta model to describe the relation between properties becomes less important but it is still important to understand.

References

TODO