

Object-Oriented Programming

Kenjiro Taura

2024/04/28

Contents

What is object-oriented programming?	2
Type Systems	17
Polymorphism and type safety	29
Establishing subtype relationship	42
Subtypes in actual languages : a taxonomy	59

What is object-oriented programming?

What is object-oriented programming?

*... Object-oriented programming (OOP) is a programming paradigm based on the concept of **objects**. Objects can contain data (called fields, attributes or properties) and have actions they can perform (called procedures or **methods** and implemented in code).*

— Wikipedia

Classes and objects : taxonomy

- **class-based** : in many languages, you first define a **class** (\approx template of objects)
 - an object is made from a class (object = **instance** of a class)
 - C++, Python, Go, Julia, Rust
- **prototype-based** or **classless** : in other languages, you can create an object with or without defining a class
 - an object can be made by a generic object expression or from a class
 - Javascript, OCaml

Classes and objects : examples

Python class definition

```
class point:  
    def __init__(self, x, y):  
        self.x = x;  
        self.y = y;
```

Object creation

```
a = point(1.2, 3.4)
```

Classless object creation : example

Javascript

```
let a = { "x" : 1.2, "y" : 3.4 }
```

OCaml (classless)

```
let a = object method x = 1.2 method y = 3.4 end
```

OCaml (with class)

```
class point (x : float) (y : float) =  
  object method x = x method y = y end;;  
let a = new point 1.2 3.4
```

Relevant keywords/syntax in our languages

language	class definition	object creation
Go	<code>type Point struct ...</code>	<code>Point(1.2, 3.4)</code>
Julia	<code>struct Point ...</code>	<code>Point(1.2, 3.4)</code>
Rust	<code>struct Point ...</code>	<code>Point(1.2, 3.4)</code>
OCaml	<code>class point ...</code>	<code>object ... end</code> or <code>new point ...</code>

Methods

- method \approx function or procedures in any other language
- so what is different?
 - *multiple definitions* of a method of the same name can exist
 - e.g., an `area` method for `rectangle`, `circle`, `triangle`, etc.
 - ***dynamic dispatch*** : when calling a method, which one gets called depends on which objects it is called for

Dynamic dispatch : taxonomy

- **single dispatch** : many languages determine which method gets called by the type of a *single* argument (“*receiver*” object)
 - C++, Python, Go, OCaml, Rust
- **multiple dispatch** : some languages determine which method gets called by the types of *multiple* arguments (objects)
 - Julia

Single dispatch : example

- multiple definitions of `area` method in Python

```
class circle:
    ...
    def area(self):
        r = self.r
        return pi * r * r

class rect:
    ...
    def area(self):
        return self.w * self.h
```

- dispatch, based on whether `s` is `circle` or `rect`

```
shapes = [circle(...), rect(...)]
for s in shapes:
    s.area() # method call (s is the receiver)
```

A single dispatch in Julia

- multiple definitions of `area` method in Julia

```
function area(c :: Circle)    function area(r :: Rect)
    pi * c.r * c.r            r.w * r.h
end                            end
```

- dispatch, based on whether `s` is `circle` or `rect`

```
shapes = [Circle(...), Rect(...)]
for s in shapes
    area(s)
end
```

Multiple dispatch in Julia

- let's say we define a method `contains(a, b)` that computes whether *a* contains *b*
- Julia allows you to define it based on *both* *a* and *b*

```
function contains(c0 :: Circle, c1 :: Circle) ...  
function contains(c0 :: Circle, r1 :: Rect) ...  
function contains(r0 :: Rect, c1 :: Circle) ...  
function contains(r0 :: Rect, r1 :: Rect) ...
```

Power of dynamic dispatch

- dynamic dispatch allows a single piece of code to work on many different kinds of data. e.g.,
- the following Python code

```
def sum(a, v0):  
    v = v0  
    for x in a:  
        v += x  
    return v
```

which is equivalent to

Power of dynamic dispatch

```
def sum(a, v0):  
    v = v0  
    it = a.__iter__()  
    try:  
        while True:                # = for x in a  
            x = it.__next__()  
            v = v.__iadd__(x)        # v += x  
    except StopIteration:  
        pass  
    return v
```

works for *any* `a` (and `v0`) satisfying the following

Power of dynamic dispatch

- `v0` has a method `__iadd__(x)`, which takes a parameter and returns anything that also has a method `__iadd__(x)`, which takes a parameter and returns anything that also has a method `__iadd__(x)`, which ...
- `a` has a method `__iter__()`, which
 - returns anything that has a method `__next__()`, which returns anything for which `v.__iadd__` works, ... (details omitted) ..., and
 - eventually raises `StopIteration`

Power of dynamic dispatch

- this is the reason why Python's for loop works for lots of data
 - lists, tuples, strings, dictionaries,
 - file handles,
 - numpy arrays
 - database query results,

and you can *define* your data structure for which the same code just works

Type Systems

Types

- **types** in programming languages \approx *kind* of data. e.g.,
 - integers, floating point numbers, array of integers, ...
 - there are user-defined types (e.g., `circle`, `rect`, etc.)
- the type of data generally determines what operations are valid on it, e.g.,
 - `s.area(...)` is valid if `s` is a `circle`, `rect`, or other type that defines an `area` method
 - `a[i] = x` is valid if `a` is an array, or other type that supports indexed assignment (`..[..] = ...`)

Type errors at runtime

- at runtime, each data naturally has its type (*dynamic type* or *runtime type*)
- when an operation not defined on the runtime type of data is applied, a *runtime type error* results.
- e.g., Python code below gets an error in the third iteration

```
shapes = [circle(...), rect(...), (3,4)]  
for s in shapes:  
    s.area()
```

Runtime vs. static type checking

- some languages perform type checking *during* execution (*runtime type checking*), which aborts the program with error messages when detected
 - Python, Javascript, Julia, ...
- some languages (*statically typed* languages) perform type checking *before* execution (*static* or *compile-time type checking*), which refuses to execute programs containing certain errors
 - C, C++, Java, Go, OCaml, Rust, ...

Static type checking and type safety

- some statically typed languages *guarantee* that no runtime type errors will happen for programs that pass static type checking (*type safe* languages)
 - Go, OCaml, Rust, ...
- it generally works by
 - calculating *the static or compile-time type of each expression*, and
 - judging the validity of each operation by static types,
 - before execution

Static type checking and type safety

- some languages do *not guarantee* no runtime type errors despite static type checking
 - some employ complementary runtime type checks, too (Java)
 - some forgo runtime type checks altogether; when a type error happens at runtime, it may cause *segmentation fault* or even worse, *data corruption* (C, C++)
 - you will see why later in the course (assembly languages and compilers)

A static type checking example (a hypothetical Python-like language)

```
l = [circle(..), circle(..)]  
for c in l:  
    c.area()
```

- static types (“*expr : type*” means *expr* has *type*)
 - ▶ `circle(..) : circle`
 - ▶ `[circle(..), circle(..)] : list of circle`
 - ▶ `l : list of circle`
 - ▶ `c : circle`
 - ▶ `c.area() : float`
- this program is **(well-)typed** and never causes a runtime error

An example containing an error

```
l = [(3,4), (5,6)]  
for p in l:  
    p.area()
```

- static types
 - ▶ (3,4) : pair of int
 - ▶ [(3,4), (5,6)] : list of pair of int
 - ▶ l : list of string
 - ▶ p : pair of int
 - ▶ p.area() : **error** (area on pair of int)

Is type safety difficult to achieve?

- in a simple case, no
- specifically, it is not difficult if the static type of an expression *uniquely* determines its runtime type
 - we call such a language *simply typed*
 - in simply typed languages, each expression or variable can take values of only a single runtime type
- Q : what's wrong with simply typed languages?

Why simply typed languages do not suffice?

- they are *inflexible* and hinder *code reusability*. e.g.,
- cannot put elements of different types in a single container

```
l = [rect(..), circle(..)]  
for s in l:  
    s.area() # what is the static type of s??
```

Why simply typed languages do not suffice?

- cannot have a single function definition of an array of different types, even when element type should not matter

```
def n_elems(l): # list of what?  
    n = 0  
    for x in l:  
        n += 1  
    return n
```

```
n_elems([1,2,3])  
n_elems(["a", "b", "c"])
```

Polymorphism

- in each of the examples, a single expression can take values of different types at runtime

```
l = [rect(..), circle(..)]    n_elems([1,2,3])
for s in l:                    n_elems(["a", "b", "c"])
    s.area()
```

- a variable or expression is said to be *polymorphic* when it can take values of different runtime types
- a language is said to support *polymorphism* when it allows polymorphic variables or expressions

Polymorphism and type safety

Polymorphism and type safety

- forget about type safety \Rightarrow polymorphism is easy to achieve
 - Julia, Python, Javascript, or many scripting languages
- forget about polymorphism (i.e., settle for simply typed languages) \Rightarrow type safety is easy to achieve
- achieving *both* polymorphism and type safety is difficult

Static type system for polymorphism

- informally, we need a static type that can represent multiple dynamic types
- two complementary approaches
 1. *subtype polymorphism* : allows a single static type that accommodates multiple types
 2. *parametric polymorphism* : allows a static type having *parameter(s)*, which can be instantiated into multiple types

Subtype polymorphism example

- `s` has a static type, like “shape”, that accommodates both `rect` and `circle`

```
l = [rect(..), circle(..)]  
for s in l:  
    s.area()
```

- in this example, we say `rect` (and `circle`) is a *subtype* of `shape`
- or, `shape` is a *supertype* of `rect` (and `circle`)
- more on this later

Parametric polymorphism

- `n_elems` has a static type (like “ $\forall \alpha. \text{array of } \alpha \rightarrow \text{int}$ ”), which can be instantiated into “array of `int` \rightarrow `int`” and “array of `string` \rightarrow `int`”

```
n_elems([1,2,3])
```

```
n_elems(["a", "b", "c"])
```

- we’ll cover this more in the next week

How static type checking works with subtyping

- consider the following program in a hypothetical Python-like language

```
def small(s : shape) -> bool:  
    return s0.area() < 10.0  
  
small(rect(..))  # or small(circle())
```

Type checking the function

```
def small(s : shape) -> bool:  
    return s.area() < 10.0
```

- $s : \text{shape}$
- $\Rightarrow s.\text{area}() : \text{float}$
- $\Rightarrow s.\text{area}() < 10.0 : \text{bool}$
- as straightforward as the simply-typed case

Type checking the function call

```
small(rect(..))
```

- `rect(..) : rect`
- `small : shape -> bool`
- must allow *passing argument of type rect (and circle) to a parameter of type shape*
 - `shape ← rect`
 - `shape ← circle`
- i.e., *treating a value of type rect as if it is of type shape*

In general ...

1. an operation (e.g., method call) is judged valid when *static type* of respective subexpression defines that operation
 - e.g., `s.area()` is valid as the static type of `s` is `shape`, which (we assume) defines `area` method
2. passing argument of T to formal argument of type S (or *treating T as if it is S* , which we denote as by $S \leftarrow T$) is judged valid **when doing so is safe**
 - e.g., `small(rect(...))` is judged valid as `shape` \leftarrow `rect` seems safe (but why?)

When is $S \leftarrow T$ safe?

- informally, $S \leftarrow T$ is safe when *any operation applicable to S is also applicable to T (*)* (Liskov substitution principle)
 - ex: “`shape` \leftarrow `rect`” is safe, because operation applicable to `shape` will be applicable to `rect` (note: whether it’s true depends on how they are actually defined, of course)
 - intuitively, safe when “ T is a kind of S ”
 - ex: `rect` (`circle`) is a kind of `shape`
 - but this intuitive reasoning breaks later (keep awake!)

When do we need to consider safety of $S \leftarrow T$?

- assignment:

```
x : shape = rect(...)
```

- passing argument:

```
def f(x : shape): ...  
f(rect(...))
```

- returning from a function

```
f (...) -> shape:  
  return rect(...)
```


When do we need to consider safety of $S \leftarrow T$?

- conditional expression

```
rect(...) if ... else circle(...) : shape
```

- heterogeneous list

```
[ rect(...), circle(...), ... ] : list of shape
```

- in all cases, we are treating an expression of type `rect (circle)` as `shape`

Subtype

- we write $T \leq S$ and say T is a **subtype** of S (and S is a **supertype** of T) when the condition (*) is the case
 - ex: $\text{rect} \leq \text{shape}$, $\text{circle} \leq \text{shape}$
- with this terminology, ***treating a value of T as S ($S \leftarrow T$) is safe simply when $T \leq S$***
- if we think of a type as a set, \leq represents a subset relation
 - this intuition breaks again later (keep awake!)

Establishing subtype relationship

Most general subtype relationship of record-like types

- when both S and T are record-like types (struct, class, etc.), $T \leq S$ holds if the following two conditions (†) are met
 1. T has all the (public) methods/fields of S
 2. for each (public) **immutable** field or method m ,
$$\text{type of } m \text{ in } T \leq \text{type of } m \text{ in } S$$
 3. for each (public) **mutable** field m ,
$$\text{type of } m \text{ in } T = \text{type of } m \text{ in } S$$

Most general subtype relationship of record-like types

- note: the exact definition can vary between languages and can be stronger (more restrictive)
- (*) is a *necessary* condition to achieve type safety

Subtype relationship example (1)

```
class shape:
    def area(self): ...

class rect:
    def area(self): ...
    def width(self): ...
    def height(self): ...
```

- Q: $\text{rect} \leq \text{shape}$ (is $\text{shape} \leftarrow \text{rect}$ safe)?

Subtype relationship example (2)

```
class shape:  
    def area(self): ...  
    def perimeter(self): ...
```

```
class rect:  
    def area(self): ...  
    def width(self): ...  
    def height(self): ...
```

- Q: $\text{rect} \leq \text{shape}$ (is $\text{shape} \leftarrow \text{rect}$ safe)?

Subtype relationship example (2)

- A: no

```
s : shape = rect(..)  
s.perimeter()
```


Subtype relationship example (3)

```
class node:
    def __init__(self):
        self.left : node
        self.right : node
```

```
class node_w_color:
    def __init__(self, col):
        self.left : node_w_color
        self.right : node_w_color
        self.color = col
```

- Q: $\text{node_w_color} \leq \text{node}$ (is $\text{node} \leftarrow \text{node_w_color}$ safe)?

Subtype relationship example (3)

- A: no

```
nc : node_w_color = node_w_color("red")  
no : node = nc  
no.left = node()  
nc.left.color      # calling .color on node!
```

- the assignment `no : node = nc` should have been disallowed

Subtype relationship example (4)

- Q : given $\text{rect} \leq \text{shape}$, is $\text{array rect} \leq \text{array shape}$?
- i.e., is the following assignment safe?

```
ar : array rect = [rect(), rect()]  
as : array shape = ar
```

Subtype relationship example (4)

- A : No

```
ar : array rect = [rect(), rect()]
as : array shape = ar
as[0] = circle()
ar[0].width()    # calling .width() on circle()!
```

- assignment `as : array shape = ar` should have been disallowed
- for reasoning, just consider each element of a mutable array is a mutable field

A side story

```
ar : array rect = [rect(), rect()]
as : array shape = ar
as[0] = circle()
ar[0].width()    # calling .width() on circle()!
```

- Java got it wrong and allows this assignment
 - a runtime exception occurs at `as[0] = circle()`
- there is another OOP language, called Eiffel, that got it wrong
 - anything (segfault or returning junk data) can happen at `ar[0].width()`

A side story

- Julia allows this assignment, too

```
ar :: Vector{Rect} = [Rect(..), Rect(..)]  
as :: Vector{Shape} = ar  
as[1] = circle()  
ar[1].width()    # calling .width() on circle()!
```

but `as : Vector{Shape} = ar` creates a copy of `ar` (wierd)

Subtype relationship tricky example (5)

```
class shape:  
    def area(self): ...  
    def eq(self, s : shape): ...
```

```
class rect:  
    def area(self): ...  
    def width(self): ...  
    def height(self): ...  
    def eq(self, s : rect): ...
```

- Q : $\text{rect} \leq \text{shape}$ (assignment $\text{shape} \leftarrow \text{rect}$ safe)?

Subtype relationship tricky example (5)

- A : No

```
s : shape = rect(..)  
s.eq(circle(..))
```

- would pass a `circle` to a formal argument of `eq` (`rect` type)

Subtype relationship tricky example (5)

- to reason more algorithmically,

$\text{rect} \leq \text{shape}$

$\Rightarrow (\text{type of eq in rect}) \leq (\text{type of eq in shape})$

$\Rightarrow \text{rect} \rightarrow \text{bool} \leq \text{shape} \rightarrow \text{bool}$

- in general, $a' \rightarrow b \leq a \rightarrow b$ holds when $a' \geq a$ (next slide)

$\Rightarrow \text{shape} \leq \text{rect}$ (false)

Subtype relationship between functions

- $a' \rightarrow b' \leq a \rightarrow b$ holds when

$$b' \leq b \text{ and } a' \geq a$$

- to see why, assume $f' : a' \rightarrow b'$ and $f : a \rightarrow b$,
 - and ask when $f \leftarrow f'$ is safe?
- recall substitution principle (*); it is when “ f' can take any data f can take (a)”. i.e.,
 - $a' \geq a$ (a' is a *supertype* of a)

Covariant and contravariant

- in general, a type $T(\alpha)$ parameterized by α , is said to be
 - **covariant on α** if replacing α with its subtype α' yields its subtype (i.e., $\alpha' \leq \alpha \Rightarrow T(\alpha') \leq T(\alpha)$)
 - **contravariant on α** if replacing α with its supertype α' yields its subtype (i.e., $\alpha' \geq \alpha \Rightarrow T(\alpha') \leq T(\alpha)$)
- in this terminology, a function type is
 - *covariant* on output type ($b' \leq b \Rightarrow a \rightarrow b \leq a \rightarrow b'$)
 - *contravariant* on input type ($a' \leq a \Rightarrow a' \rightarrow b \leq a \rightarrow b$)

Subtypes in actual languages : a taxonomy

Taxonomy of subtype relationships

interface subtyping vs. *concrete-type* subtyping

- concrete-type subtyping (C++, Java, OCaml)
 - \leq is introduced between ordinary (concrete) types
- interface subtyping (Go, Rust, Julia)
 - besides ordinary types, define *abstract types* (Julia), *interfaces* (Go), or *traits* (Rust)
 - \leq is introduced only between interfaces or between a concrete type and an interface

Taxonomy of subtype relationships

nominal subtyping vs. *structural* subtyping

- nominal (Julia, Rust)
 - \leq holds only when the programmer so specified explicitly
 - Julia: `struct T <: S`
 - Rust: `impl trait for struct`
- structural (Go, OCaml)
 - \leq is derived automatically from definitions

```
type Shape interface { area() float64 }  
type Rect struct { ... }  
func (r Rect) area() float64 { ... }
```

- with Go structural subtyping, $\text{Rect} \leq \text{Shape}$ is *automatically* established because `Rect` has an `area` method returning `float64`, allowing the following assignment

```
var s shape = rect{0, 0, 100, 100}
```

- does not need anything specific to define functions on `struct`

```
struct Rect ... end
struct Circle ... end
function area(r :: Rect) ... end
function area(c :: Circle) ... end
```


- if you want to define a single function body that works *both* on `Rect` and `Circle`, you can use an abstract type. e.g.,

```
abstract type Shape ... end
struct Rect <: Shape ... end
struct Circle <: Shape ... end
function dist(s :: Shape) ... end
```

Rust

```
trait Shape { fn area(&self) -> f64; }  
struct Rect { ... }  
impl Shape for Rect {  
    fn area(&self) -> f64 { ... }  
}
```

- with Rust (nominal subtyping between struct and trait), $\text{Rect} \leq \text{Shape}$ is established by explicitly stating `impl Shape for Rect`, allowing the assignment below

```
let s : &dyn Shape = &Rect{ ... };
```

OCaml

- OCaml requires no type (class) definitions to make objects
- type of an object expression is automatically derived from methods
- even if they have different names, they are the same if their structures are the same
- you may need a type annotation (actually, a cast) for
 - if expressions having different types in then/else clauses
 - `if ... new rect() else new circle()`
 - heterogeneous list/array expressions
 - `[new rect(); new circle(); ...]`

- you can use *type cast* $((e :> S))$ to treat e as if it is S
 - valid only when e 's (naturally derived) static type $T \leq S$
 - not the cast you may know in C
- e.g.,
 - `[(new rect() :> shape); (new circle() :> shape); ...]`
 - valid if `rect` \leq `shape` (determined by their method signatures)
 - note: unnecessary if `rect` and `circle` happen to have the same method signatures