Object-Oriented Programming

Kenjiro Taura

2024/04/28

Contents

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	`type Point struct ...`	`Point(1.2, 3.4)`
Julia	`struct Point ...`	`Point(1.2, 3.4)`
Rust	`struct Point ...`	`Point(1.2, 3.4)`
OCaml	`class point ...`	`object ... end`

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 (object)
  - it is called a "*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:                 class rect:
  ...                           ...
  def area(self):               def area(self):
    r = self.r                    return self.w * self.h
    return pi * r * r
```

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

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

An equivalent Julia example

- multiple definitions of `area` method in Julia

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

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

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

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

Multiple dispatch

- 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) ...
```

Type Systems

Types

- ***types*** in programming languages ≈ *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:
  print(s.area())
```

Runtime type errors are disastrous

- some languages perform ***runtime type checking***, which detects runtime type errors and gracefully aborts the program with error messages
  - Python, Javascript, Julia
- some languages do not perform runtime type checking, which may cause *segmentation fault* or even worse, *data corruption*
  - C, C++

Static types and static type checking

- other languages guarantee, ***prior to execution***, that no runtime type errors will happen (***static type checking***)
- 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

An example (in 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

Another example

```
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)

(A bit pedantic) taxonomy

- languages performing static type checking are generally called ***statically typed***
- not all statically typed languages *guarantee* absense of runtime type errors
  - C, C++, Java
- statically typed languages that do are called ***type safe***
  - Go, OCaml, and Rust (without `unsafe`) are type safe

Is type safety difficult to achieve?

- in the 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
- then what's the matter?

Why simply typed languages do not suffice?

- they are *inflexible* and hider *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 representing multiple dynamic types
- two common 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

- `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 "$\forall\alpha.\ \mathrm{array\ of}\ \alpha \to \mathrm{int}$"), which can be instantiated into "array of int" and "array of string"

```
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

- in the hypothetical Python-like language

```
def smaller(s0 : shape, s1 : shape) -> shape:
  return (s0 if s0.area() < s1.area() else s1)

smaller(rect(..), circle(..))
smaller(circle(..), rect(..))
```

- `s0, s1` : shape
- $\Rightarrow$ `s0.area(), s1.area()` : float
- $\Rightarrow$ `s0.area() < s1.area()` : boolean
- $\Rightarrow$ `s0 if ... else s1` : shape

The key question

- in the example above,
  - `smaller(rect(..), circle(..))` is valid. i.e.,
  - passing a value of “rect” (or “circle”) type to a parameter of “shape” type is allowed
- the key question:

for two types S and T when is an *assignment-like operation* $S \leftarrow T$ valid (safe if allowed)?

Note: assignment-like operation

- intuitively, any operation that flows a value to another place
  - assignment (left hand side : $S \leftarrow$ right hand side T)
  - passing arguments (formal arg : $S \leftarrow$ actual arg : T)
- in general, any operation where a value whose static type is T becomes a value of another expression whose static type is S
  - returning a value (return type $S \leftarrow$ returned expression : T)
  - conditional expression (result type $S \leftarrow$ then/else expression : T)

When is $S \leftarrow T$ safe?

- informally, $S \leftarrow T$ is safe when ***any operation applicable to S is also applicable to T $(*)$*** (*Liskov substition principle*)
  - ex: "shape $\leftarrow$ rect" is safe, because operation applicable to (any) shape will be applicable to rect (whether it's true depends on how they are actually defined, of course)
- intuitively, T is a kind of S
  - ex: rect (circle) is a kind of shape

Subtype

- we write $T \leq S$ and say T is a ***subtype*** of S (and S is a ***supertype*** of T) when $(*)$ is the case
  - ex: rect $\leq$ shape, circle $\leq$ shape
- if we think of a type as a set, $\leq$ represents a subset relation
- the exact definition of $\leq$ varies between languages, but $(*)$ must hold to achieve type safety

Most generic subtype relationship

- if 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 method m,
     type of m in $T \leq$ type of m in S

Subtype relationship example (1)

- shape
  - has `area()` method returning float
- rect
  - has `area()` method returning float
  - has additional `width()` and `height()` methods
- rect $\leq$ shape holds

Subtype relationship example (2)

- shape
  - has `area()` method returning float and
  - `perimeter()` method returning float
- rect is the same as before
- rect $\leq$ shape does not (*should not*) hold
- to see why, consider

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

Subtype relationship tricky example (3)

- shape
  - has `area()` method returning float and
  - `eq(s : shape)` method returning bool
- rect
  - has `area()` method returning float,
  - has `width()` and `height()` method each returning float, and
  - `eq(r : rect)` method returning bool
- does rect $\leq$ shape hold?

Subtype relationship tricky example (3)

- no, it *should not* hold
- to see why not, consider

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

- which passes circle type to a formal argument of `eq` (rect type)

Subtype relationship tricky example (3)

- more algorithmically,

rect $\leq$ shape
$\Rightarrow$ type of `eq` in rect $\leq$ type of `eq` in shape
$\Rightarrow$ rect $\rightarrow$ bool $\leq$ shape $\rightarrow$ bool

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

$\Rightarrow$ shape $\leq$ rect (false)

Subtype relationship between functions

- $a' \to b' \leq a \to b$ holds when
  - $b' \leq b$ and $a' \geq a$
- recall substition principle $(*)$
  - assume $f' : a' \to b'$ and $f : a \to b$,
  - and ask when $f \leftarrow f'$ is safe?
- 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 ($b' \leq b \Rightarrow a \to b \leq a \to b'$)
  - *contravariant* on input type ($a' \leq a \Rightarrow a' \to b \leq a \to b$)

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)
    - besides ordinary types, define *abstract types*, *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 (Rust)
    - $\leq$ holds only when the programmer so specified explicitly (`impl` *trait* `for` *struct*)
  - structural (Go, OCaml)
    - $\leq$ is derived automatically from definitions

Go

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

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

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

Subtyping in Rust

```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), `Rect` $\leq$ `Shape` is established by explicitly stating `impl Shape for Rect`, allowing the assignment below

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

OCaml

- OCaml does not require type (class) definitions to make objects
- when you define class, subtype relationship is automatically derived
- nor does it require type of variables to be specified
- … everything just *naturally* happens (learn in the notebook)