Programming Lanaugages (2)
Object-Oriented Programming Basics

Kenjiro Taura

Classes and objects

- a *class* ≈ a data type definition + functions (*methods*) for it
- an *object* is a data instance created from a class definition

```
# define a class named rect
class rect:
  def __init__(self, x, y, width, height):
    self.x = x
    self.y = y
    self.width = width
    self.height = height

r = rect(10,20,30,40) # create an instance (or an object) of rect
```

Methods

- ≈ functions
- unlike ordinary functions, a method of the same name can be defined for multiple classes (i.e., implemented differently)

```
class rect:
  ...
  # define a method named area
  def area(self):
    return self.width * self.height

class ellipse:
  ...
  # define another method named area
  def area(self):
    return self.radius * self.radius * math.pi

```

Dynamic dispatch

- when you call a method, which method gets called among many implementations is determined by the class argument(s) belong to

```
# shapes may have both rect and ellipse instances
for s in shapes:
    ... s.area() ...
```

Language design points

```
# shapes may have both rect and ellipse instances
for s in shapes:
    ... s.area() ...
```

- in a code like the above, a variable `s` may take a value of different classes (types) over time *(polymorphism)*
- for languages that require type declarations, *how to declare/specify the type of* `s` *or* `shapes`*?*
- *does Go/Julia/OCaml/Rust require type declarations?*

Language design points

```
# shapes may have both rect and ellipse instances
for s in shapes:
    ... s.area() ...
```

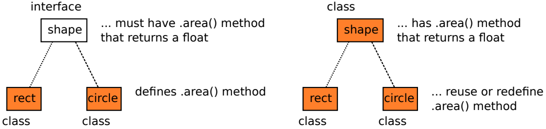
- more fundamentally, how can we guarantee, prior to execution, that *type errors (≈ application of non-existing methods) do not happen at runtime?*
- such property is called *type safety*
- an algorithm that checks type safety prior to execution is often called *static type checking*
- *does Go/Julia/OCaml/Rust guarantee type safety?*

Different approaches

- forgo static type checking and thus type safety (e.g., Python, javascript, Lisp, Smalltalk, ...)
- disallow polymorphism altogether and make it (trivially) type-safe (e.g., Pascal)
- do some (loose) static type checking but allow polymorphism via unsafe casts between pointers (e.g., C/C++)
- allow polymorphism yet guarantee type safety via *subtypes*
  - *C is a subtype of P ($C \leq P$)* $\equiv$ a value of C can be safely used wherever P is expected
  - allow $P \leftarrow C$ (assign a variable of type P a value of type C)

Different approaches to subtyping

- *subclass* vs. *interface*
  - a *subclass* that *inherits, extends or derives from* an existing class to make a subtype
  - an *interface* (or *trait*, *abstract class*, etc.) and a *(concrete) class* that *implements or conforms to* it



- *nominal (explicit)* vs. *structural* subtyping
  - nominal : subtype relation admitted only when so declared
  - structural : subtype relation admitted whenever appropriate (based on the structure)

How/if they guarantee type safety?

- following slides briefly explain how Go/Rust/OCaml guarantee *type safety*
- *type safety* $\equiv$ "no such methods" error never happens at runtime $\equiv$ when a program containing $o.m(\ldots)$ passes static type check, o always has method m at runtime
- recall that this is not the case for some languages (including Python, Julia, C++, etc.)

A common framework

- we (i.e., static type checker) like to guarantee that,
  - for any expression E whose *static type* is S,
  - any value E could take at runtime can be *safely put* in anywhere S is expected ($\approx$ any such value implements all the methods S specifies)
- for which we have to guarantee that, for any *assignment-like operations* $o = p$, any value p could take at runtime can be *safely put* in anywhere S is expected
- we want to check it by comparing p's static type (T) and o's static type (S)
- this is precisely what we like to capture by *subtype* relationship ($T \leq S$)

Note: assignment-like operations

- = any operation in which a value is stored to a location of potentially different static type
  - assignment to a variable/structure/array element
  - function calls (passing values to parameters)
  - function return (returning a value)

Subtype relationship

- T is a subtype of S ($T \leq S$)
- $\approx$ any value of T can be safely put anywhere S is expected
- $\approx$ T has all methods S has
- (this is not exactly correct, but suffices for now)

Go

- details on Assignability section of Go reference
- Q: when is a type T a subtype of another type S ($T \leq S$)?

Go

- details on Assignability section of Go reference
- Q: when is a type T a subtype of another type S ($T \leq S$)?
- A: one of the following
  1. S and T are identical type
  2. S is an interface and T is a `struct/interface` that *implements* S or a pointer to it

Go

- details on Assignability section of Go reference
- Q: when is a type T a subtype of another type S ($T \leq S$)?
- A: one of the following
  1. S and T are identical type
  2. S is an interface and T is a `struct/interface` that *implements* S or a pointer to it
- Q: so when is T said to *implement* an interface S?

Go

- details on Assignability section of Go reference
- Q: when is a type T a subtype of another type S ($T \leq S$)?
- A: one of the following
  1. S and T are identical type
  2. S is an interface and T is a `struct/interface` that *implements* S or a pointer to it
- Q: so when is T said to *implement* an interface S?
- A:
  - T has all the methods specified in S, and
  - each method in T has the same type as the method of the same name in S

Rust

- Q: when is a type T a subtype of another type S ($T \leq S$)?

Rust

- Q: when is a type T a subtype of another type S ($T \leq S$)?
- A: one of the following
  1. S and T are identical type
  2. S is a reference to a trait and T is a reference to a `struct` that *implements* S

Rust

- Q: when is a type T a subtype of another type S ($T \leq S$)?
- A: one of the following
  1. S and T are identical type
  2. S is a reference to a trait and T is a reference to a `struct` that *implements* S
- Q: so when does T *implement* an interface S?

Rust

- Q: when is a type T a subtype of another type S ($T \leq S$)?
- A: one of the following
  1. S and T are identical type
  2. S is a reference to a trait and T is a reference to a `struct` that *implements* S
- Q: so when does T *implement* an interface S?
- A:
  - T has all the methods specified in S, and
  - each method in T has the same type as the method of the same name in S

OCaml

- Q: when is a type T a subtype of another type S ($T \leq S$)?

OCaml

- Q: when is a type T a subtype of another type S ($T \leq S$)?
- A: one of the following
  1. S and T are identical type

OCaml

- Q: when is a type T a subtype of another type S ($T \leq S$)?
- A: one of the following
  1. S and T are identical type
  2. when each of S and T is a function type ($S = a \to b$ and $T = a' \to b'$), then $b' \leq b$ and $a \leq a'$

OCaml

- Q: when is a type T a subtype of another type S $(T \leq S)$?
- A: one of the following
  1. S and T are identical type
  2. when each of S and T is a function type ($S = a \rightarrow b$ and $T = a' \rightarrow b'$), then $b' \leq b$ and $a \leq a'$
  3. when each of S and T is an object type ($S = <m_0 : t_0, \ldots>$, $T = <m'_0 : t'_0, \ldots>$), then
     - $\{m_0, \ldots\} \subset \{m'_0, \ldots\}$ and
     - for each $m_i = m'_j$, $t'_j \leq t_i$