COSE212: Programming Languages

Lecture 19 — Typed Object-Oriented Language

Hakjoo Oh
2026 Fall

Review: Classes

P	$\rightarrow$	$D^*\ E$	program
D	$\rightarrow$	$\texttt{class}\ C\ \texttt{extends}\ C'\ \{F^*\ M^*\}$	class declaration
F	$\rightarrow$	$\texttt{field}\ x$	field declaration
M	$\rightarrow$	$\texttt{method}\ m(x_1, \ldots, x_n)\ E$	method declaration
E	$\rightarrow$	$n \mid x$	
	$\mid$	$E + E \mid E - E$	arithmetic
	$\mid$	$\texttt{iszero}\ E$	zero test
	$\mid$	$\texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E$	conditional
	$\mid$	$\texttt{let}\ x = E\ \texttt{in}\ E$	binding
	$\mid$	$\texttt{proc}\ x\ E \mid E\ E$	procedure / call
	$\mid$	$x := E \mid E; E$	assignment / sequence
	$\mid$	$\texttt{new}\ C(E_1, \ldots, E_n)$	object creation
	$\mid$	$E.m(E_1, \ldots, E_n)$	message send
	$\mid$	$\texttt{self}$	current object
	$\mid$	$\texttt{super}.m(E_1, \ldots, E_n)$	super call

What Can Go Wrong Without Types?

Imagine a `Counter` class with methods `incr`, `set(int)`, `get`:

```
let c = new Counter() in c.foo()
let c = new Counter() in c.set(iszero 0)
(new Counter()).incr() + 1
```

In CLASSES, semantic errors are caught only at runtime:

- *Method not found*: `Counter` does not declare `foo`.
- *Wrong argument type*: `c.set` expects an `int`, not a `bool`.
- *Wrong return-type use*: `c.incr()` returns `void`.

Goal: Develop a static type checker that catches these at compile-time

Example

The type checker takes a program in which every field and method declaration is annotated with a type.

```
class Counter extends object {
  field int count
  method void initialize ()       { count := 0 }
  method void incr ()             { count := count + 1 }
  method void set  (int n)        { count := n }
  method int  get ()              { count }
}
let c = new Counter() in c.incr(); c.get()
```

- The type checker concludes that the program does not have type errors and the whole expression has type `int`.
- The type checker rejects "`c.set(true)`" (wrong arg type), "`c.foo()`" (method not found), and "`c.incr() + 1`" (wrong return-type use).

Types of the Language

T	$::=$	int	integer
	$\mid$	bool	boolean
	$\mid$	`void`	no useful return value
	$\mid$	$T \to T$	function
	$\mid$	C	object type (class name)

- A class name C is also a type.
- `void` is the result type of a method whose value is discarded.
- Function types appear because we keep `proc`/call from Classes.

Syntax of Typed Classes

P	$\rightarrow$	$D^*\ E$	program
D	$\rightarrow$	$\texttt{class}\ C\ \texttt{extends}\ C'\ \{F^*\ M^*\}$	class declaration
F	$\rightarrow$	$\texttt{field}\ T\ x$	typed field
M	$\rightarrow$	$\texttt{method}\ T_r\ m(T_1\ x_1, \ldots, T_n\ x_n)\ E$	typed method
E	$\rightarrow$	$n \mid x$	
	$\mid$	$E + E \mid E - E$	arithmetic
	$\mid$	$\texttt{iszero}\ E$	zero test
	$\mid$	$\texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E$	conditional
	$\mid$	$\texttt{let}\ x = E\ \texttt{in}\ E$	binding
	$\mid$	$\texttt{proc}\ (x : T)\ E \mid E\ E$	procedure / call
	$\mid$	$x := E \mid E; E$	assignment / sequence
	$\mid$	$\texttt{new}\ C(E)$	object creation
	$\mid$	$E.m(E)$	method call
	$\mid$	$\texttt{self}$	current object
	$\mid$	$\texttt{super}.m(E)$	super call

Example: Point and ColorPoint

```
class Point extends object {
  field int x   field int y
  method void initialize (int initx, int inity) {
    x := initx; y := inity }
  method void move (int dx, int dy) { x := x+dx; y := y+dy }
  method int  getY ()  { y }
}
class ColorPoint extends Point {
  field int color
  method void initialize (int initx, int inity, int initc) {
    super.initialize(initx, inity);
    color := initc }
  method int get-color () { color }
}
let cp = new ColorPoint(10, 20, 87) in
let f  = proc (p : Point) p.getY() in
  begin cp.move(3, 4); f(cp) end
```

Note that `f(cp)` passes a `ColorPoint` where a `Point` is expected. Our type checker captures this *subclass polymorphism* with a *subtyping* relation.

Subtyping: $T_1 <: T_2$

"T_1 is a subtype of T_2" = "every value of type T_1 may safely be used where type T_2 is expected."

$$\frac{}{T <: T}\ \textsc{Sub-Refl} \qquad \frac{T_1 <: T_2 \quad T_2 <: T_3}{T_1 <: T_3}\ \textsc{Sub-Trans}$$

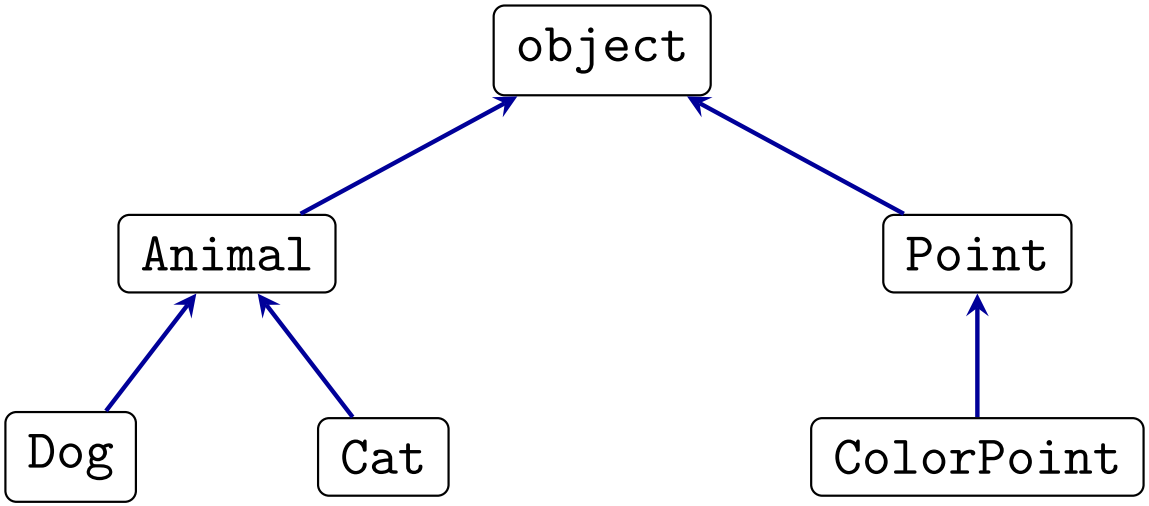
$$\frac{C\ \texttt{extends}\ C' \in P}{C <: C'}\ \textsc{Sub-Class}$$

$$\frac{T_1' <: T_1 \qquad T_2 <: T_2'}{T_1 \to T_2 <: T_1' \to T_2'}\ \textsc{Sub-Arrow}$$

- Sub-Class: every declared inheritance creates one subtype relation.
- Sub-Arrow: function types are *contravariant* in the argument and *covariant* in the result.
- Distinct base types (int, bool, `void`) are not related by subtyping; only Sub-Refl applies.

Subtyping in Pictures

The `extends` tree literally *is* the class subtype tree: inheritance edges become subtype edges (root downward).



The arrow $C \to C'$ reads "C `<:` C'". Transitive closure (Sub-Trans) gives e.g. `Dog <: object`. Unrelated classes such as `Dog` and `Point` are incomparable.

Subtyping for Function Types

Recall:

$$T_1 \to T_2 \ <: \ T_1' \to T_2' \quad \text{iff} \quad T_1' <: T_1 \text{ and } T_2 <: T_2'$$

Argument position is *contravariant*; result position is *covariant*.

Contravariant argument:

$$(\texttt{Point} \to \text{int}) \ <: \ (\texttt{ColorPoint} \to \text{int})$$

A function over *any* `Point` also handles a `ColorPoint`. The reverse fails: a `ColorPoint` $\to$ int function may use `ColorPoint`-only operations (e.g., `get-color`) and cannot be called on a plain `Point`.

Covariant result:

$$(\text{int} \to \texttt{ColorPoint}) \ <: \ (\text{int} \to \texttt{Point})$$

A function that returns a `ColorPoint` is safe to use where the caller only expects a `Point`: every returned `ColorPoint` is also a `Point`.

Type Judgment

We extend the typing judgment from $\Gamma \vdash E : T$ to

$$\zeta; \Gamma \vdash E : T$$

where

- $\Gamma : Id \to type$ is the type environment that maps each in-scope variable to its type.
- $\zeta : Class \to ClassType$ is the static class environment, where

  $$ClassType = Class \times (Field \to type) \times (MName \to type^* \times type \times Class)$$

  - For each declared class C, $\zeta(C) = (C', fT, mT)$ records the parent, field types, and method signatures.
  - A signature $mT(m) = ((T_1, \ldots, T_n), T_r, C_h)$ stores parameter types, return type, and the host class C_h where m was declared.

Typing Rules (Inherited)

$$\frac{}{\zeta; \Gamma \vdash n : \text{int}} \text{ T-Num} \qquad \frac{}{\zeta; \Gamma \vdash x : \Gamma(x)} \text{ T-Var}$$

$$\frac{\zeta; \Gamma \vdash E_1 : \text{int} \quad \zeta; \Gamma \vdash E_2 : \text{int}}{\zeta; \Gamma \vdash E_1 + E_2 : \text{int}} \text{ T-Plus}$$

$$\frac{\zeta; \Gamma \vdash E : \text{int}}{\zeta; \Gamma \vdash \texttt{iszero } E : \text{bool}} \text{ T-Iszero}$$

$$\frac{\zeta; \Gamma \vdash E_1 : \text{bool} \quad \zeta; \Gamma \vdash E_2 : T \quad \zeta; \Gamma \vdash E_3 : T}{\zeta; \Gamma \vdash \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 : T} \text{ T-If}$$

$$\frac{\zeta; \Gamma \vdash E_1 : T_1 \quad \zeta; \Gamma[x \mapsto T_1] \vdash E_2 : T_2}{\zeta; \Gamma \vdash \texttt{let } x = E_1 \texttt{ in } E_2 : T_2} \text{ T-Let}$$

$$\frac{\zeta; \Gamma[x \mapsto T_1] \vdash E : T_2}{\zeta; \Gamma \vdash \texttt{proc } (x : T_1)\ E : T_1 \to T_2} \text{ T-Proc}$$

$$\frac{\zeta; \Gamma \vdash E_1 : T_1 \to T_2 \quad \zeta; \Gamma \vdash E_2 : T_1}{\zeta; \Gamma \vdash E_1\ E_2 : T_2} \text{ T-Call}$$

$$\frac{\zeta; \Gamma \vdash E : \Gamma(x)}{\zeta; \Gamma \vdash x := E : \texttt{void}} \text{ T-Assign} \qquad \frac{\zeta; \Gamma \vdash E_1 : T_1 \quad \zeta; \Gamma \vdash E_2 : T}{\zeta; \Gamma \vdash E_1; E_2 : T} \text{ T-Seq}$$

Subsumption

$$\frac{\zeta; \Gamma \vdash E : T \qquad T <: T'}{\zeta; \Gamma \vdash E : T'} \text{ T-SUB}$$

- T-SUB lifts an expression's type from T up to any supertype T'. Semantically, this means that a value of a subtype can be used wherever a supertype is expected.
- Unlike the other rules, T-SUB is not syntax-directed: it can fire at any expression.

Example

Consider `f(cp)`, where `f` : `Point` $\rightarrow$ `Point` and `cp` : `ColorPoint`. Let $\Gamma = [f{:}\texttt{Point}\rightarrow\texttt{Point},\ cp{:}\texttt{ColorPoint}]$.

$$\dfrac{\dfrac{}{\zeta;\Gamma \vdash f : \texttt{Point}\rightarrow\texttt{Point}}\ \text{T-Var} \quad \dfrac{\dfrac{}{\zeta;\Gamma \vdash cp : \texttt{ColorPoint}}\ \text{T-Var} \quad \dfrac{\dfrac{\texttt{ColorPoint extends Point} \in P}{\texttt{ColorPoint} <: \texttt{Point}}\ \text{Sub-Class}}{\zeta;\Gamma \vdash cp : \texttt{Point}}\ \text{T-Sub}}{\zeta;\Gamma \vdash f\ cp : \texttt{Point}}\ \text{T-Call}$$

- T-Call's premise demands the argument type to match the parameter type exactly ($\Gamma \vdash cp : \texttt{Point}$).
- T-Sub is used to lift cp from `ColorPoint` to `Point` using `ColorPoint` <: `Point` (Sub-Class).
- This is how subsumption works in the inherited rules: T-Sub fires wherever subtype flexibility is needed.

Typing Rule: `new`

$$\frac{\begin{array}{c}\zeta(C) = (\_, \_, mT) \\ mT(\texttt{initialize}) = ((T_1, \ldots, T_n), \_, \_) \\ \zeta; \Gamma \vdash E_i : T_i' \text{ and } T_i' <: T_i \text{ for each } i\end{array}}{\zeta; \Gamma \vdash \texttt{new } C(E_1, \ldots, E_n) : C}\ \text{T-New}$$

- Look up the `initialize` signature for class C.
- Each actual argument E_i must have a subtype of the corresponding declared parameter type T_i.
- The new expression's type is C itself.

Typing Rule: $E_0.m(\ldots)$

$$\frac{\zeta;\Gamma \vdash E_0 : C \quad \zeta(C) = (\_, \_, mT) \quad mT(m) = ((T_1, \ldots, T_n), T_r, \_) \quad \zeta;\Gamma \vdash E_i : T_i' \text{ and } T_i' <: T_i \text{ for each } i}{\zeta;\Gamma \vdash E_0.m(E_1, \ldots, E_n) : T_r}\ \text{T-Send}$$

- The receiver must have an object type C.
- C's method table must declare m.
- Arguments are checked against m's parameter types with subtyping.
- The send's type is m's declared return type T_r.

Typing Rules: `self` and `super`

$$\frac{}{\zeta; \Gamma \vdash \texttt{self} : \Gamma(\texttt{self})}\ \text{T-Self}$$

$$\frac{\begin{array}{c}\zeta(\Gamma(\texttt{host})) = (C_p, \_, \_) \\ \zeta(C_p) = (\_, \_, mT) \\ mT(m) = ((T_1, \ldots, T_n), T_r, \_) \\ \zeta; \Gamma \vdash E_i : T_i' \ \text{ and } \ T_i' <: T_i \ \text{ for each } i\end{array}}{\zeta; \Gamma \vdash \texttt{super}.m(E_1, \ldots, E_n) : T_r}\ \text{T-Super}$$

- T-Self: `self`'s type is read off Γ; it is the receiver class fixed by step (2) of the algorithm.
- T-Super: walk to the parent of the host class. Hence super dispatches statically, matching its run-time behaviour.

The Type-Checking Algorithm

The checker runs in three steps:

1. **Building the class environment ζ.**
   - Scan the classes and build ζ, recording each one's parent, field types, and method signatures.
   - Inheritance is resolved here: the parent's fields and methods carry over to the child.
   - Verify that every override has a signature compatible with its parent's.
2. **Type-checking method bodies.**
   - Build a typing environment Γ from the class's fields, the method's parameters, and the pseudo-bindings `self`/`host` pointing to the enclosing class.
   - Check the body under this Γ; its type must fit the declared return type (a subtype is fine).
   - Parameters shadow fields with the same name.
3. **Type-checking the main expression** under an empty Γ. Its type is the program's overall type; any failed check rejects the program.

(1) Building the Class Environment ζ

Classes are processed top-down along the inheritance tree. Given the current ζ and $D = \texttt{class}\ C\ \texttt{extends}\ C'\ \{F^*\ M^*\}$:

1. **Look up the parent's entry:** $\zeta(C') = (\_,\ fT_{par},\ mT_{par})$. Root case: $\zeta(\texttt{object}) = (\bot,\ \emptyset,\ \emptyset)$, where $\bot$ means no parent.
2. **Define C's field table.** Let $F^* = \texttt{field}\ T_1\ x_1, \ldots, \texttt{field}\ T_k\ x_k$. Then extend fT_{par} with the declared bindings[1]:

$$fT_C = fT_{par}[x_1 \mapsto T_1, \ldots, x_k \mapsto T_k].$$

3. **Define C's method table.** Let $M^* = M_1, \ldots, M_l$, with $M_j = \texttt{method}\ T_r^j\ m_j(\ldots)\ E_j$, and let $\sigma_j = ((T_1^j, \ldots, T_{n_j}^j),\ T_r^j,\ C)$ be the signature of M_j. Extend (or override) mT_{par}:

$$mT_C = mT_{par}[\, m_1 \mapsto \sigma_1, \ldots, m_l \mapsto \sigma_l\,].$$

4. **Check override consistency** for each method in M^* that already appears in mT_{par} (next slide).

Result: $\zeta[\, C \mapsto (C',\ fT_C,\ mT_C)\,]$.

[1] Fields are renamed to unique names, so the extension never collides with fT_{par}.

Override Consistency

Suppose class C extends C' and both declare a method named m. Let

$$\begin{aligned} mT_{par}(m) &= ((T_1, \ldots, T_n),\ T_r,\ \_) \\ mT_C(m) &= ((T'_1, \ldots, T'_n),\ T'_r,\ \_) \end{aligned}$$

The override is acceptable iff

$$T_i <: T'_i \text{ for each } i, \qquad T'_r <: T_r.$$

- *Contravariant parameters, covariant return*: exactly the conditions for the child's method, viewed as a function type, to be a subtype of the parent's (Sub-Arrow).
- Many languages are stricter and require parameter types to match exactly and allow only a covariant return type. Our rule is the most permissive sound choice.
- Constructor exception: `initialize` is invoked by `new` $C(\ldots)$ at the receiver's own class, not via dynamic dispatch. Each class may declare its own `initialize` with an arbitrary signature, and the override check above does not apply to it.

Override Consistency: Examples

Assume `ColorPoint` $<:$ `Point` and the parent declares m.

Accepted:

- Same signature: parent int $m()$, child int $m()$.
- Covariant return: parent `Point` $m()$, child `ColorPoint` $m()$.
  (`ColorPoint` $<:$ `Point` ✓.)
- Contravariant parameter: parent int $m($`ColorPoint`$)$, child int $m($`Point`$)$.
  (`ColorPoint` $<:$ `Point` ✓.)

Rejected:

- Arity mismatch: parent int $m()$, child int $m($int$)$.
- Bad return: parent int $m()$, child bool $m()$. (bool $\not<:$ int.)
- Covariant parameter: parent int $m($`Point`$)$, child int $m($`ColorPoint`$)$.
  (Needs `Point` $<:$ `ColorPoint`, which fails.)

(2) Type-Checking Method Bodies

For each class C in the program and each method
`method` $T_r\ m(T_1\ x_1, \ldots, T_n\ x_n)\ E_b$ declared in C:

1. **Build the typing environment Γ_b** from three sources:
   - *fields*: every binding in fT_C,
   - *parameters*: $x_1 \mapsto T_1,\ \ldots,\ x_n \mapsto T_n$,
   - *pseudo-bindings*: $\texttt{self} \mapsto C,\ \texttt{host} \mapsto C$.

   Parameters are added after fields, so they shadow fields of the same name. Formally,

$$\Gamma_b\ =\ fT_C[\,x_1 \mapsto T_1, \ldots, x_n \mapsto T_n,\ \texttt{self} \mapsto C,\ \texttt{host} \mapsto C\,].$$

2. **Derive the body's type $\zeta; \Gamma_b \vdash E_b : T'$** using the typing rules.
3. **Check the declared return type:** require $T' <: T_r$

If every method body passes, the class declarations are internally consistent.

(3) Type-Checking the Main Expression

Given $P = D_1 \cdots D_n; E$ with ζ built from $D_1, \ldots, D_n$ by step (1):

1. **Derive** $\zeta; \emptyset \vdash E : T$.
2. **Report** T as the program's overall type.

If any of the derivation fails (in either step (2) or (3)), the program is rejected.

Example

```
class A extends object { field int v
  method void initialize () { v := 0 }
  method int  get ()        { v } }
class B extends A {
  method int  get ()        { v + 1 } }   % override
in (new B()).get()
```

(1) Build ζ (abbreviating `initialize` as `init`):

$$\zeta = \begin{bmatrix} \boldsymbol{A} \mapsto (\texttt{object},\ [\boldsymbol{v}{:}\mathsf{int}],\ [\texttt{init}{:}((), \texttt{void}, \boldsymbol{A}),\ \texttt{get}{:}((), \mathsf{int}, \boldsymbol{A})]) \\ \boldsymbol{B} \mapsto (\boldsymbol{A},\ [\boldsymbol{v}{:}\mathsf{int}],\ [\texttt{init}{:}((), \texttt{void}, \boldsymbol{A}),\ \texttt{get}{:}((), \mathsf{int}, \boldsymbol{B})]) \end{bmatrix}$$

(2) Method bodies, each well-typed against its declared return:

- $\boldsymbol{A}.\texttt{init}$ with $\boldsymbol{\Gamma_b} = [\boldsymbol{v}{:}\mathsf{int},\ \texttt{self}{:}\boldsymbol{A},\ \texttt{host}{:}\boldsymbol{A}]$: $\boldsymbol{v}{:=}\boldsymbol{0} : \texttt{void}$ ✓.
- $\boldsymbol{A}.\mathsf{get}$ with the same $\boldsymbol{\Gamma_b}$: $\boldsymbol{v} : \mathsf{int}$ ✓.
- $\boldsymbol{B}.\mathsf{get}$ with $\boldsymbol{\Gamma_b} = [\boldsymbol{v}{:}\mathsf{int},\ \texttt{self}{:}\boldsymbol{B},\ \texttt{host}{:}\boldsymbol{B}]$: $\boldsymbol{v}{+}\boldsymbol{1} : \mathsf{int}$ ✓.

(3) Main $\boldsymbol{E} = (\texttt{new}\ \boldsymbol{B}()).\mathsf{get}()$ under empty $\boldsymbol{\Gamma}$: T-New gives $\texttt{new}\ \boldsymbol{B}() : \boldsymbol{B}$; T-Send on $\boldsymbol{mT_B}(\mathsf{get}) = ((), \mathsf{int}, \boldsymbol{B})$ gives result int. Hence $\boldsymbol{\zeta}; \boldsymbol{\emptyset} \vdash \boldsymbol{E} : \mathsf{int}$.

Summary

We added a static type system to CLASSES:

- Class names as types
- Subtyping and subsumption
- Typing judgment $\boldsymbol{\zeta; \Gamma \vdash E : T}$
- Three-phase checking algorithm