**Featherweight Java**

Classes
\[ C ::= \text{class } c \text{ extends } c \{ \_f ; \_d \} \]

Constructors
\[ k ::= c (\_x) \{ \text{super } (x); \text{this.} _f = x; \} \]

Methods
\[ d ::= c \text{ m} (\_x)\{\text{return } e \}; \]

Types
\[ \tau ::= c \]

Expressions
\[ e ::= x | e.f | e.m(e) | \text{new } c (e) | (c) e \]

Notation:
- \( f \): field name
- \( c \): class name
- \( m \): method name
- \( x \): variable name
- \( e \) is an abbreviation for \( e_1, e_2, \ldots, e_k \)
- \( \_x \) is an abbreviation for \( c_1 x_1; \ldots \)
Class expressions:

class $c$ extends $c'$

\[
\{ c_1 f_1; \\
    c_2 f_2; \\
    \vdots \\
    k \\
    d_1 \\
    d_2 \ldots \}
\]

declares class

$\rightarrow$ $c$ to be a subclass of $c'$

$\rightarrow$ with additional fields $c_i f_i$

$\rightarrow$ a single constructor $k$

$\rightarrow$ methods $d_i$
Constructor expressions:

\[
c (c_1' x_1', c_2' x_2', \ldots, c_1 x_1, c_2 x_2, \ldots)\{
\text{super}(x_1', \ldots)
\text{this} . f_1 = x_1;
\text{this} . f_2 = x_2;
\vdots
\}
\]

declares a constructor for class \( c \)

\( \rightarrow \) with arguments \( c_i' x_i' \) corresponding to fields of the superclass
\( \rightarrow \) and arguments \( c_i x_i \) corresponding to new fields of the subclass
\( \rightarrow \) \( x_i' \) are initialised via the superclass
\( \rightarrow \) \( \text{this} . f_i = x_i; \) fields initialised in the subclass
Method expressions:

\[ c \ m \ (c_1 \ x_1, c_2 \ x_2, \ldots) \{ \text{return} (e) \}; \]

denotes a method \( m \)

\( \rightarrow \) which returns a value of class \( c \)

\( \rightarrow \) with arguments \( x_i \) of class \( c_i \)

\( \rightarrow \) and a body returning the value of expression \( e \)
Field selection:

\[ e.f \]

select field \( f \) from instance \( e \)

Method invocation:

\[ e.m(e_1, e_2, \ldots) \]

invoke method \( m \) of instance \( e \) with arguments \( e_1, e_2 \)
Instance creation:

\[
\text{new } c \left( e_1, e_2, \ldots \right)
\]

creates a new instance of class \( c \) with arguments \( e_1, e_2 \)

Casting:

\[
(c) \ e
\]

casts value \( e \) to class \( c \)
the set of types is limited to the set of class names

- in examples, we may assume the presence of the types like `int`, but we will not discuss the semantics of operations on these types

there is a class `Object`

- all other classes are subclasses of `Object`

a special variable `this` referring to the instance itself
the subclass/superclass relationship is similar to the
class/supertype relationship with coercion interpretation
if \( c \) is a subclass of \( c' \), the \( c \) has at least as many fields as \( c' \)
objects of class \( c \) can be coerced (cast) to \( c' \) by deleting the
additional fields
we therefore write \( c <: c' \)
class Pt extends Object {
    int x;
    int y;
    Pt (int x, int y) {
        super (); this.x = x; this.y = y;
    }
}

class CPt extends Pt {
    color c;
    CPt (int x, int y, color c) {
        super (x,y);
        this.c = c;
    }
    color getc () {return this.c;}
}
The static semantics is defined by the following judgements:

\[
\begin{align*}
\tau <: \tau' & \quad \text{subtyping} \\
\Gamma \vdash e : \tau & \quad \text{expression typing} \\
d \text{ok in } c & \quad \text{well formed method} \\
C \text{ ok} & \quad \text{well formed class} \\
T \text{ ok} & \quad \text{well formed class table} \\
\text{fields}(c) = \{c_1 f_1, c_2 f_2 \ldots\} & \quad \text{field lookup} \\
\text{type} (m, c) = c \rightarrow c & \quad \text{method type}
\end{align*}
\]

A program consists of a class table \( T \) (sequence of class declarations) and an expression \( e \).

We use the class table as implicit parameter in the following. We only look at some interesting aspects of the static semantics.
Subclass relationship:

\[
\begin{align*}
\tau & <: \tau \\
\tau_1 & <: \tau_2 \\
\tau_2 & <: \tau_3
\end{align*}
\]

\[
\tau_1 <: \tau_3
\]

\[
T(c) = \text{class } c \text{ extends } c' \{ \ldots \}
\]

\[
c <: c'
\]
Every variable has to be declared

\[
\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}
\]

Types of fields are defined in the class table:

\[
\frac{\Gamma \vdash e_0 : c_0 \quad \textbf{fields}(c_0) = c_1 f_1 \ldots}{\Gamma \vdash e_0.f_i : c_i}
\]
Argument and result types of methods are defined in the class table:

\[ \Gamma \vdash e_0 : c_0 \quad \Gamma \vdash e : c \quad \text{type}(m, c_0) = c' \rightarrow c_1 \quad c <: c' \]

\[ \Gamma \vdash e_0.m(e) : c_1 \]
All casts are statically valid:

\[
\frac{\Gamma \vdash e_0 : c'}{
\Gamma \vdash (c)e_0 : c}
\]
Method overriding:

New subclass method must have

- the same argument types
- the same result type

as superclass method

\[
T(c) = \text{class } c \text{ extends } c' \{ \ldots \} \quad \text{type}(m, c') = c \rightarrow c_0 \quad x : c, \text{this} : c \vdash e_0 : c_0 \\
\begin{array}{c}
    c_0 m (c \ x) \{ \text{return } e_0; \} \
\end{array}
\quad \text{ok in } c
\]
Properties:

- casts may fail at run time: checks required
- method invocation is statically checked
- field selection is statically checked
We discuss the single step or structural operational semantics of Featherweight Java.
**Featherweight Java: Dynamic Semantics**

We discuss the single step or structural operational semantics of Featherweight Java.

**Values**: an instance is a value if all of it’s arguments are values

\[
\frac{\text{value}}{\text{new } c (v) \text{ value}}
\]
We discuss the single step or structural operational semantics of Featherweight Java.

**Values:** an instance is a value if all of its arguments are values.

\[
\frac{v \text{ value}}{\text{new } c(v) \text{ value}}
\]

In essence, a instance is just a collection of named fields, labeled with the class name.
Field Selection:

- $c'_i f'_i$: fields of the super class
- $c_j f_j$: new fields of the class
- retrieve value of field from either superclass or class itself

\[
\text{fields}(c) = c'_i f'_i, \ c f
\]
\[
\text{new } c \ (v'_i, v) \cdot f'_i \mapsto v'_i
\]

\[
\text{fields}(c) = c'_i f'_i, \ c f
\]
\[
\text{new } c \ (v'_i, v) \cdot f_i \mapsto v_i
\]
Field Selection:

- $c_i f'_i$: fields of the super class
- $c_j f_j$: new fields of the class
- retrieve value of field from either superclass or class itself

\[
\begin{align*}
\text{fields}(c) &= c'_f, c_f \\
\text{new } (v', v). f'_i &\mapsto v'_i
\end{align*}
\]

Note that in Featherweight Java, the content of fields cannot be changed after initialization.

We could extend the language using techniques described previously.
Method Invocation:

- Method invocation relies on an auxiliary predicate, \( \text{body} \), which provides the list of formal arguments and the body of a method \( m \) defined in a class \( c \).
- Replace all formal parameters \( x \) by actual parameters \( e' \).
- Replace every occurrence of \( \text{this} \) by the instance itself.

\[
\begin{align*}
\text{body}(m, c) &= x \rightarrow e_0 \\
\text{new } c(e) \cdot m(e') &\mapsto \{e'/x\}\{\text{new } c(e)/\text{this}\}e_0
\end{align*}
\]
Type Cast:

→ if $c'$ is supertype of $c$, then cast instance by “ignoring” fields that are not present in $c'$

→ otherwise, cause a checked run-time error

$$
\begin{align*}
\frac{c <: c'}{(c')\text{new } c (e) \mapsto \text{new } c (e)} \\
\frac{c \not<: c'}{(c')\text{new } c (e) \mapsto \text{error}}
\end{align*}
$$
Type Cast:

¬ if \(c'\) is supertype of \(c\), then cast instance by “ignoring” fields that are not present in \(c'\)

¬ otherwise, cause a checked run-time error

\[
\begin{align*}
\text{if } c' <: c & \Rightarrow (c')\text{new } c \ (e) \mapsto \text{new } c \ (e) \\
\text{if } c' \not<: c & \Rightarrow \text{error}
\end{align*}
\]

Note the interplay with the static semantic rules for type cast
Dynamic Dispatch:

To find the body $body(m, c)$ of a method $m$ of a class $c$, we have to search upwards through the class hierarchy for the first definition of $m$.

$$T(c) = \text{class } c \text{ extends } c'\{...d\} \quad d_i = c_i \ m(c_i \ x)\{\text{return}(e);\}$$

$$body(m, c) = x \rightarrow e$$

$$T(c) = \text{class } c \text{ extends } c'\{...d\} \quad m \notin d \quad body(m, c') = x \rightarrow e$$

$$body(m, c) = x \rightarrow e$$
Dynamic Dispatch:

To find the body $\text{body}(m, c)$ of a method $m$ of a class $c$, we have to search upwards through the class hierarchy for the first definition of $m$.

$$T(c) = \text{class } c \text{ extends } c'\{\ldots d\} \quad d_i = c_i \ m(c_i \ x)\{\text{return}(e);\}$$

$$\text{body}(m, c) = x \rightarrow e$$

$$T(c) = \text{class } c \text{ extends } c'\{\ldots d\} \quad m \notin d \quad \text{body}(m, c') = x \rightarrow e$$

$$\text{body}(m, c) = x \rightarrow e$$

And finally, we need rules to determine the evaluation order
For records (or n-tuples) two types of subtyping:

→ **Width Subtyping**

\[
\frac{m > n}{\{l_1 : \tau_1, \ldots l_m : \tau_m\} <: \{l_1 : \tau_1, \ldots l_n : \tau_n\}}
\]

Note: ordering of fields in records immaterial, rule allows us to “drop” arbitrary subset of fields

→ **Depth Subtyping**

\[
\frac{\sigma_1 <: \tau_1, \sigma_2 <: \tau_2, \ldots}{\{l_1 : \sigma_1, \ldots l_n : \sigma_n\} <: \{l_1 : \tau_1, \ldots l_n : \tau_n\}}
\]
For the implementation, we need to synthesize the coercion functions if we want to avoid runtime checks.
IMPLEMENTATION OF SUBTYPING

For the implementation, we need to synthesize the coercion functions if we want to avoid runtime checks.

We define a new relation $\tau_1 <: \tau_2 \rightsquigarrow f$

$\Rightarrow \tau_1$ is subclass of $\tau_2$, and $f$ is the coercion function of type $\tau_1 \rightarrow \tau_2$. 
\[
\text{Int} <: \text{Float} \leadsto \text{intToFloat}
\]

\[
\tau <: \tau \leadsto \text{id}
\]

\[
\begin{aligned}
\tau_1 &<: \tau_2 \leadsto f_1 \\
\tau_2 &<: \tau_3 \leadsto f_2 \\
\tau_1 &<: \tau_3 \leadsto
\end{aligned}
\]

\[
\begin{aligned}
\tau_1' &<: \tau_1 \leadsto f_1 \\
\tau_2 &<: \tau_2' \leadsto f_2 \\
\tau_1 &\rightarrow \tau_2 <: \tau_1' \rightarrow \tau_2' \leadsto
\end{aligned}
\]
\[ \text{Int} <: \text{Float} \rightsquigarrow \text{intToFloat} \]

\[ \tau <: \tau \rightsquigarrow \text{id} \]

\[ \begin{array}{c}
\tau_1 <: \tau_2 \rightsquigarrow f_1 \\
\tau_2 <: \tau_3 \rightsquigarrow f_2
\end{array} \]

\[ \tau_1 <: \tau_3 \rightsquigarrow f_2 \cdot f_1 \]

\[ \begin{array}{c}
\tau_1' <: \tau_1 \rightsquigarrow f_1 \\
\tau_2 <: \tau_2' \rightsquigarrow f_2
\end{array} \]

\[ \tau_1 \rightarrow \tau_2 <: \tau_1' \rightarrow \tau_2' \rightsquigarrow \]
\[
\begin{align*}
\text{Int} & \prec \text{Float} \rightsquigarrow \text{intToFloat} \\
\tau & \prec \tau \rightsquigarrow \text{id} \\
\tau_1 & \prec \tau_2 \rightsquigarrow f_1 \quad \tau_2 & \prec \tau_3 \rightsquigarrow f_2 \\
\tau_1 & \prec \tau_3 \rightsquigarrow f_2 \cdot f_1 \\
\tau_1' & \prec \tau_1 \rightsquigarrow f_1 \quad \tau_2 & \prec \tau_2' \rightsquigarrow f_2 \\
\tau_1 & \rightarrow \tau_2 & \prec \tau_1' \rightarrow \tau_2' \rightsquigarrow f, \quad \text{where } f \ g \ x = f_2(g(f_1 \ x))
\end{align*}
\]
Subtyping complicates efficient implementation of field selection
Record type no longer reveals fields and types of fields of record
Possible solution: coercion creates a new copy of the record with adjusted number of fields
  - drop: create a copy of record, delete fields
  - copy: create a copy of record, apply respective coercion function to all components
Problematic for mutable records
Possible solution: separation of view from contents