We address the various typing rules for Featherweight Java, each in turn (see Figure 19-4, p. 259).

1. T-Var

\[ x : C \in \Gamma \]
\[ \Gamma \vdash x : C \]

Note that (as established earlier) the only types in the language are class names, so \( \Gamma \) is merely a mapping from (variable) names to (class) names. Also note that the only variables that will be typed in this manner are method parameters and \textit{this}. Unlike in Java, fields and methods in Featherweight Java must be referenced using an explicit object (for example, \textit{this}.somemethod() is valid; somemethod() is not).

2. T-Field

\[ \Gamma \vdash t_0 : C_0 \]
\[ \text{fields}(C_0) = \bar{C} \bar{f} \]
\[ \bar{f} \Gamma \vdash t_0.f_i : C_i \]

All the work of traversing the class inheritance hierarchy in order to find the appropriate field is handled by \textit{fields}(C_0).

It was pointed out that a field name could be understood within the type system as a function that maps from a class to class. Given some class, the field name’s corresponding function provides the field’s class. This is similar to the higher-order concept of intersection types, which we should be discussing later in the semester.

3. T-Invk and T-New

\[ \Gamma \vdash t_0 : C_0 \]
\[ \text{mtype}(m, C_0) = \bar{D} \rightarrow C \]
\[ \Gamma \vdash \bar{f} : \bar{C} \quad \bar{C} <: \bar{D} \]
\[ \Gamma \vdash t_0.m(\bar{t}) : C \]

\[ \text{fields}(C) = \bar{D} \bar{f} \]
\[ \Gamma \vdash \bar{f} : \bar{C} \quad \bar{C} <: \bar{D} \]
\[ \Gamma \vdash \text{new} \ C(\bar{t}) : C \]

The primary interesting feature of these rules is the subtyping test for each of the method or constructor parameters. Without this subtyping rule, the language would be just as powerful, but each method invocation might require an upcast for each of its arguments.
For example, with a subtyping check, classes \texttt{Number} extends \texttt{Object} and \texttt{Integer} extends \texttt{Number}, and some method \texttt{m(Number n)}, we can say \texttt{this.m(new Integer())}. Without the subtyping check, we would have to say \texttt{this.m((Number) new Integer())}.

(Note (not discussed in class): If we did change the language in this way, we would have to make some changes to the evaluation rules. Under the current evaluation rules, \texttt{new X().m((Number) new Integer()) → new X().m(new Integer())}. The result of this evaluation is no longer a well-typed term under the modified typing rules, and preservation is violated.)

4 T-Ucast, T-Dcast, and T-SCast

\[
\begin{align*}
\Gamma \vdash t_0 : D & \quad D <: C \\
\Gamma \vdash (C)t_0 : C \\
\Gamma \vdash t_0 : D & \quad C <: D \quad C \neq D \\
\Gamma \vdash (C)t_0 : C \\
\Gamma \vdash t_0 : D & \quad C \nleq; D \quad D \nleq; C \quad \text{stupid warning} \\
\Gamma \vdash (C)t_0 : C
\end{align*}
\]

We allow all types of casts here, but a “stupid cast” results in a stupid warning.

Given that a stupid cast can always be expressed as an upcast to \texttt{Object} followed by a downcast to the result \texttt{((Foo) new Bar())} is equivalent to \texttt{(Foo) (Object) new Bar()}, why should we give stupid casts any special treatment at all?

First, we must be clear about what we mean. Take, for example, the classes \texttt{Foo} extends \texttt{Object} and \texttt{Bar} extends \texttt{Object}. According to E-CastNew, the term \texttt{((Foo) new Bar())} is stuck, because \texttt{Bar \nleq; Foo}. Similarly, \texttt{(Foo) (Object) new Bar()} evaluates in one step to \texttt{(Foo) new Bar()} because \texttt{Bar \nleq; Object}; and the result is \texttt{(Foo) new Bar()}, which is stuck. In this way, the two terms are equivalent—both get stuck and, what’s more, the stuck expressions are syntactically equal. Eventually, any downcast term (a term typed with T-DCast) will yield a stuck term or an upcast term (a term typed with T-UCast); any stupid-cast term will get stuck; and any upcast term will yield a value.

So the T-SCast rule simply warns the programmer that a term will always get stuck. If the cast is expressed using up– and downcasts, the fact that the term will always get stuck is not as simple to prove. In general, of course, a downcast cannot be proven to always get stuck.

One could envision a language in which E-CastNew simply let anything go. In this language, a \texttt{Bar} might be cast to a \texttt{Foo}, and evaluation would proceed normally. If one of \texttt{Foo}’s methods, say \texttt{getFoo()} were invoked after the cast, then one of two things would happen:

2
1. If Bar had a method `getFoo()`, then that method would be invoked instead. Evaluation would happily proceed. If Bar's `getFoo()` method happened to return a value with a different type than Foo's `getFoo()` method, the typing discrepancy would be propagated.

2. If Bar did not have a method `getFoo()`, then the term would be stuck.

Under this scheme, we gain the flexibility to refer to arbitrary methods and fields in a value, regardless of the value’s actual type. But we lose the efficiency of being able to assume certain properties belong to a value of some type (for example, we would need to test, for each method invocation, that the method exists; we would no longer know in an implementation at what offset the field \( x \) may be found). And in such a language, typing rules would be much less informative (typing “suggestions”, we might call them), because there is no guarantee that a term with some type will actually produce a value of that type.

However, the situation in Featherweight Java is not much better, because we still can’t guarantee that a term of type \( T \) will produce a value of type \( T \). As theorem 19.5.4 states, Progress in FJ means that a term either produces a value or gets stuck.

*(Note (Not discussed in class): The difference here, though, is that we can pin down exactly what the term will look like when it gets stuck, whereas without the subtype-check in E-CastNew, a term could get stuck at a variety of points. Under FJ, we can still assert the Preservation property: if a term of type \( T \) produces something, the result will be of type \( T \).*

Still, the weakness of the guarantees we can make about a well-typed FJ term is troubling. This weakness stems from an inherent problem in languages that support non-symmetric subtyping (where \( A <: B \not\Rightarrow B <: A \)): if we allow a variety of values to be described in more general terms, we’re probably going to want at some point to talk about them in specific ways again. Put another way, supporting upcasts (even implicitly) is fundamentally limited without providing a downcast mechanism, and downcasts introduce problems for type safety.

*(Note (Not discussed in class): In making this point, we fail to note that downcasts can occur implicitly and safely: when I invoke a method on an object of some general type \( A \), that method, via this, has access to the specific type \( B \) (assuming \( B \) overrides the method in \( A \)). We’ve gone from general to specific without relying on an unsafe cast. Visitors, applied to a class hierarchy that supports them, allow arbitrary code to be executed based on the specific type of a general object, while avoiding unsafe casts. A subtyping language could support more sophisticated, but similar, constructs (such as match statements) to support the execution of some code based on an arbitrary object’s type.*

Here’s an example:

```java
class Integer { ... }

class A {
```
```java
Integer x;
A(Integer x) { super(); this.x = x; }
Integer apply(Visitor v) { return v.forA(this); }
}

class B extends A {
    Integer y;
    B(Integer x, Integer y) { super(x); this.y = y; }
    Integer apply(Visitor v) { return v.forB(this); }
}

class C extends A {
    Integer z;
    C(Integer x, Integer z) { super(x); this.z = z; }
    Integer apply(Visitor v) { return v.forC(this); }
}

class Visitor {
    Visitor() { super(); }
    Integer forA(A that) { return ... } // 0, as an Integer
    Integer forB(B that) { return ... } // 0, as an Integer
    Integer forC(C that) { return ... } // 0, as an Integer
}

class GetMagicNumber extends Visitor {
    GetMagicNumber() { super(); }
    Integer forA(A that) { return that.x; }
    Integer forB(B that) { return that.x.plus(that.y); }
    Integer forC(C that) { return that.x.times(that.z); }
}

// Now that we have this framework (whew!), we can write a class
// that can access the appropriate B and C fields of an A without
// downcasting to B or C.

class Foo {
    A a;
    Foo(A a) { super(); this.a = a; }
    Integer getMagicNumber() { return a.apply(new GetMagicNumber()); }
}

It's not pretty, but that's not the point. The point is that downcasts aren't
essential to subtyping languages—even one as simple as FJ.)
```