TYPES FOR OBJECTS
• What a type system for Java would look like?

• Some points of order:
  • Java is a large language and providing type rules for all of its constructs is very time consuming.
  • First we'll carve out a small core of Java that captures most of the essential OO features. We'll call this small language Java Jr:
    • This should include at least classes, subclasses, objects, methods and field accesses.
    • Other features like exceptions, concurrency etc can be added later.
  • We'll formalise the type rules in the style that we have seen already.
    • We’ll use nominal typing to define the subtyping relation.
At top level, a Java Jr program is made up of a sequence of class declarations:

\[ P ::= \overline{C} \]

Assume a class ‘Main’ with a nullary constructor and a method ‘main’.

\[ C ::= \text{class } c \text{ extends } t \{ K \overline{F} \overline{M} \} \]

A class contains a constructor; its field types and method definitions.

\[ K ::= c(\overline{t} \overline{f}, \overline{u} \overline{g})\{\text{super}(\overline{f}); \text{this.} \overline{g} = \overline{g}; \} \]

The constructor methods are default and the only code they contain is to initialise the fields.

\[ F ::= t \overline{f}; \]

Classes have sequences of these field declarations.

\[ M ::= \text{public } t \overline{m}(\overline{t} \overline{x})\{\text{return } E; \} \]

Methods always have a return type (no void), they return the evaluation of a expression.

\[ E ::= v | x | E.m(\overline{E}) | E.f | E.f = E | \]

new \( t(\overline{E}) | (E == E ? E : E) \)

Expressions include base values, variables, method call, field accesses, new objects and conditionals.

Types \( t \) include Object, int and bool, Values \( v \) include integer and boolean literals, Variables \( x \) include the keyword ‘this’
JAVA JR IS A SUBSET OF JAVA

• If you look carefully, you can see that Java Jr is actually a valid subset of Java.
• This means one can write a Java Jr program and compile it using javac.
• We’ll assume the following Java execution harness:

```java
class JavaJRExec {
    public static void main(String[] args) {
        Main m = new Main();
        System.out.println(m.main().toString());
    }
}
```

• This will let us run Java Jr programs using the command java JavaJRExec
• It will print out the result of evaluating a program.
SOME USEFUL FUNCTIONS

- To give formal type rules for Java Jr it is useful to define a number of lookup functions on the source code:
- \( \text{name ( class } c \text{ extends } t \{ \ldots \} ) = c \)
- \( P . t = C \) if \( \text{name}(C) = t \)
- \( P . \text{Object.methods} = \{ \} \)
- \( P . t . \text{methods} = P . u . \text{methods} + M \) where \( P . t = \text{class } t \text{ extends } u \{ K \bar{F} \bar{M} \} \)
- Here \( \bar{M} + \bar{M}' \) is defined to replace occurrences of methods named \( m \) in \( \bar{M} \) where they also appear in \( \bar{M}' \)
- \( P . \text{Object.fields} = \{ \} \)
- \( P . t . \text{fields} = P . u . \text{fields} + F \) where \( P . t = \text{class } t \text{ extends } u \{ K \bar{F} \bar{M} \} \)
- Similar for \( F + F' \): fields named \( f \) in \( F \) are replaced by any fields declared in \( F' \)
- \( P . \text{Object.sigs} = \{ \} \)
- \( P . t . \text{sigs} = P . u . \text{sigs} + \bar{M} . \text{sigs} \) where \( P . t = \text{class } t \text{ extends } u \{ K \bar{F} \bar{M} \} \)
- \((\text{public } t \text{ m} (t x) \{ \ldots \} ) . \text{sigs} = t \text{ m}(t x)\)
Let's start by considering what the well-formed types of the language are. This is a relation between the Program P and the type names ranged over by t:

\[
P \vdash \text{Object} : \text{type} \quad P \vdash \text{int} : \text{type} \quad P \vdash \text{bool} : \text{type}
\]

\[
P \cdot t = \text{class } c \text{ extends } u \{ K \overline{F} \overline{M} \}
\]

\[
P \vdash t : \text{type}
\]

Now define the subtyping relation:

\[
P \vdash t : \text{type} \quad P \vdash t : \text{type} \quad P \vdash t <: u \quad P \vdash u <: s
\]

\[
P \cdot t = \text{class } c \text{ extends } u \{ K \overline{F} \overline{M} \}
\]

\[
P \vdash u : \text{type}
\]

\[
P \vdash t <: u
\]
We'll work through the Java Jr grammar from the top down:

\[ \overline{C} \vdash C_i : \text{class} \quad \text{for each } C_i \in \overline{C} \]
\[ \vdash \overline{C} : \text{program} \]

This says that for a whole program to be well-typed, then each separate class definition must be well-typed (with respect to the other defined types in the program).

We'll make an abbreviation in the above rule as follows:

\[ \overline{C} \vdash \overline{C} : \text{class} \]
\[ \vdash \overline{C} : \text{program} \]

We intend it to mean exactly the same as the above rule we are just writing it more concisely. We'll use this notation a lot.
This rule is a bit of a monster:

\[
P \vdash s : \text{type} \quad P \vdash \bar{u} : \text{type} \quad P \vdash \bar{M} : \text{method in } c \\
P.s.sigs \subseteq P.c.sigs \quad P.s.fields = \bar{t} \bar{f};
\]

\[
P \vdash \text{class } c \text{ extends } s \{ c(t \bar{f}, \bar{u} \bar{g})\{ \text{super} (f) ; \text{this.} \bar{g} = \bar{g} ; \} \bar{u} \bar{g} ; \bar{M} \} : \text{class}
\]

It captures what it means to be a valid class definition. In words:

- The declared supertype must actually be a type.
- The types of the fields in this class must actually be types.
- The methods declared in the class must be well-formed methods in that class.
- The signature of methods declared in the class must be compatible with the signature of any overridden methods in the superclass.
- The fields of the superclass must be properly listed in the parameters of the default constructor.
TYPE RULE FOR WELL-FORMED METHODS

\[
\begin{align*}
P \vdash t : \text{type} & \quad P \vdash \overline{t} : \text{type} & \quad P \mid \overline{x} : \overline{t}, \text{this} : c \vdash E : t \\
\hline
P \vdash \text{public } t \ m(t \ \overline{x})\{\text{return } E; \} : \text{method in } c
\end{align*}
\]

In this case we see that in order to be a well-formed method we need:

The return type of the method to be a well-formed type.

The types of the formal parameters to the method must be well-formed types.

The method body must be a well-typed expression.

n.b. Method bodies may contain free occurrences of the formal parameter variables, and hence we need a type environment for these. Also note that the 'this' keyword is a free variable in the body E and we need a type environment for that also.
We’ll start off with the simple expressions: values and variables. Note that type judgements for expressions also carry the type environment $\Gamma$ for the free variables in the expression.

\[
\begin{align*}
P | \Gamma \vdash n : \text{int} & \quad P | \Gamma \vdash b : \text{bool} \\
\frac{x : t \in \Gamma}{P | \Gamma \vdash x : t}
\end{align*}
\]

Now consider the type rule for creating new objects.

\[
\begin{align*}
P \vdash c : \text{class} & \quad P.c.\text{fields} = \bar{t} \bar{f} \\
\frac{P | \Gamma \vdash \bar{E} : \bar{t}}{P | \Gamma \vdash \text{new } c(\bar{E}) : c}
\end{align*}
\]

So $c$ must be a valid class in $P$ and the fields of this class (including inherited fields) must match the types of the arguments to the constructor.
MORE TYPE RULES FOR EXPRESSIONS

The type rule for conditional expressions should be straightforward:

\[
P \mid \Gamma \vdash E_1 : u \quad P \mid \Gamma \vdash E_2 : u \quad P \mid \Gamma \vdash E_T : t \quad P \mid \Gamma \vdash E_F : t
\]

\[
P \mid \Gamma \vdash (E_1 == E_2?E_T : E_F) : t
\]

We can compare objects of the same type, and the branches must have the same type. By subsumption (subtyping) we will be able to compare things of any two, possibly unrelated reference types.

Rules for field accesses are as follows:

\[
P \mid \Gamma \vdash E : u \quad t \ f ; \in P.u.fields
\]

\[
P \mid \Gamma \vdash E.f : t
\]

\[
P \mid \Gamma \vdash E : u \quad t \ f ; \in P.u.fields \quad P \mid \Gamma \vdash E' : t
\]

\[
P \mid \Gamma \vdash E.f = E' : t
\]

We look up the fields in the type of E and check that the type of their usage is matched.
The type rule for method calls is similar to field access:

\[
\frac{P \mid \Gamma \vdash E : u \quad t \ m(t \ f) ; \in P.u.sigs}{P \mid \Gamma \vdash \overline{E} : \overline{t}} \quad \frac{P \mid \Gamma \vdash E.m(\overline{E}) : t}{P \mid \Gamma \vdash E.m(\overline{E}) : t}
\]

We look at the type of the receiver.
Then look up the method signature for method m.
The return type from the method call is the return type declared in the signature.
The arguments to the method must match the types of the formal parameters.

Finally, we need to include the subsumption rule to allow subtyping:

\[
\frac{P \mid \Gamma \vdash E : t \quad P \vdash t <: u}{P \mid \Gamma \vdash E : u}
\]
ANY OTHER BUSINESS?

• That’s about it for Java Jr in terms of specifying the static type system.
• There are a number of other things that we could consider:

• How about term representations of the run time states of Java Jr programs?
  • To do this we would need to model the heap: use a mapping \( H \) from locations to “objects”. An object here is simply a record of values for the fields along with a class name field that the object was constructed with.
  • We would introduce a class of values \( o, o' \) etc into Java Jr to represent locations in the heap. These would be the result of evaluating \( \text{new } c( v_1, v_2, \ldots, v_n ) \).
  • The type for a location \( o \) would simply need to look up \( o \) in the heap and access the class name field.

• How about null pointers?
  • Easy, just add an extra value \textbf{null} to the grammar and give the type rule that \textbf{null} has any reference type.
CASTS

• We could consider adding the (downwards) casting operation: \((c)E\)
• But what is a good type rule for this?

\[
P \mid \Gamma \vdash E : t \\
\frac{}{P \mid \Gamma \vdash (c)E : c}^{\text{TCast}}
\]

• This rule forces the type checker to ignore the actual type of \(E\) and just treat it as if it has type \(c\). This seems okay.
• We need \(E\) to be well-typed still in case the run time check passes!
• In some formalisations of Java we see this variant of the rule:

\[
P \mid \Gamma \vdash E : t \quad P \vdash t <: c \quad \text{or} \quad P \vdash c <: t \\
\frac{}{P \mid \Gamma \vdash (c)E : c}^{\text{TSafeCast}}
\]

• This is problematic because it allows “stupid” casts. That is, for any classes \(A\) and \(B\), the expression \((A)\;\text{new}\;B(\;\text{)}\;)\) is well-typed. This follows from using subsumption to consider the \(B\) object at type \(\text{Object}\) and then the cast rule with \(A <: \text{Object}\).
• After one step of evaluation, the program would reach an ill-typed term (which would then throw an exception in the next step)
• The problem is resolved by adding another rule for “stupid” casts (same as TSafeCast but with unrelated types). Any use of this rule act as a static warning that a run time cast will fail.
NEXT LECTURE: LAMBDA CALCULUS AND THE CURRY HOWARD CORRESPONDENCE