The Type Checker

- The type checker has several tasks:
  - determine the types of all expressions
  - check that values and variables are used correctly
  - resolve certain ambiguities by transformations
- Some languages have no type checker
  - no compile-time guarantees
  - runtime tagging and checking become necessary
The Joos types are:

- void (the empty set of values)
- byte
- short
- int
- char
- boolean
- C (objects of class C or any subclass)
- null (the polymorphic null constant)
- σ[] (for any type σ)
Type Annotations

- Type annotations:
  ```
  int x;
  Foo y;
  ```

  specify **invariants** about the runtime behavior:
  - `x` will always contain an integer value
  - `y` will always contain `null` or an object of class `Foo` or any subclass thereof

- Java-like Type annotations are (arguably) not very expressive viewed as invariants
  - Other type systems are more expressive
A program is type correct if the type annotations are valid invariants.

Type correctness is trivially undecidable:

```java
int x = 0;
TM(j);
x = false;
```

The program is type correct iff the \( j \)'th Turing machine does not halt on empty input.
A program is **statically type correct** if it satisfies some type rules.

The rules are chosen to be:
- simple to understand
- efficient to decide
- conservative with respect to type correctness

Type systems are not canonical

They are designed like the rest of the language
Static type systems are necessarily flawed:

- There is always **slack**: programs that are unfairly rejected by the type checker.
Examples of Slack

- **Trivial example:**
  ```java
  x = 87;
  if (false) x = true;
  ```

- **Useful example:**
  ```java
  Map m = new Hashmap();
  m.put("bar","foo");
  String x = m.get("bar");
  ```

- **Java 1.5 was designed to pick up some slack**
Specifying Type Rules

- Prose:

  "The argument of the sqrt function must be of type int, the result is of type double"

- Logical rules:

  \[
  \text{C} \vdash x : \text{int} \\
  \quad \text{C} \vdash \sqrt{x} : \text{double}
  \]
The first expression must be of type boolean, or a compile-time error occurs. The conditional operator may be used to choose between second and third operands of numeric type, or second and third operands of type boolean, or second and third operands that are each of either reference type or the null type. All other cases result in a compile-time error. Note that it is not permitted for either the second or the third operand expression to be an invocation of a void method. In fact, it is not permitted for a conditional expression to appear in any context where an invocation of a void method could appear. The type of a conditional expression is determined as follows: If the second and third operands have the same type (which may be the null type), then that is the type of the conditional expression. Otherwise, if the second and third operands have numeric type, then there are several cases: If one of the operands is of type byte and the other is of type short, then the type of the conditional expression is short. If one of the operands is of type $T$ where $T$ is byte, short, or char, and the other operand is a constant expression of type int whose value is representable in type $T$, then the type of the conditional expression is $T$. Otherwise, binary numeric promotion is applied to the operand types, and the type of the conditional expression is the promoted type of the second and third operands. Note that binary numeric promotion performs value set conversion. If one of the second and third operands is of the null type and the type of the other is a reference type, then the type of the conditional expression is that reference type. If the second and third operands are of different reference types, then it must be possible to convert one of the types to the other type (call this latter type $T$) by assignment conversion; the type of the conditional expression is $T$. It is a compile-time error if neither type is assignment compatible with the other type.
Three Kinds of Type Rules

- Declarations:  
  *this variable is declared to have this type*

- Propagations:  
  *if the argument is of this type, then the result is of that type*

- Restrictions:  
  *the argument must be of this type or that type*

- The logical notation handles all three uniformly
Judging Expressions

- The judgment:

\[ C, L, X, \sigma \vdash E : \tau \]

means that the expression E is statically type correct and has type \( \tau \) in a context where:

- C is the current class
- L is the current local environment
- X is the current set of exceptions
- \( \sigma \) is the return type of the current method
Judging Statements

- The judgment:

\[ C, L, X, \sigma |- S \]

means that the statement \( S \) is statically type correct in a context where:

- \( C \) is the current class
- \( L \) is the current local environment
- \( X \) is the current set of exceptions
- \( \sigma \) is the return type of the current method
The class environment is implicitly available

The notation $D \leq C$ means that $D$ is a subclass of $C$

We can query a class $C$:

- $C |- \text{extends } D$ (super class)
- $C |- \sigma f$ (fields)
- $C |- \text{static } \sigma f$ (static fields)
- $C |- (\sigma_1, ..., \sigma_k) X$ (constructors)
- $C |- \sigma m (\sigma_1, ..., \sigma_k) X$ (methods)
- $C |- \text{static } \sigma m (\sigma_1, ..., \sigma_k) X$ (static methods)
### Assignment Compatibility

- The relation $\sigma := \tau$ means that values of type $\tau$ may be assigned to variables of type $\sigma$:

\[
\begin{align*}
\sigma := \sigma & \quad \text{short := byte} \\
\text{int := byte} & \quad \text{int := short} \\
\text{int := char} & \quad \text{C := null} \\
\sigma[\cdot] := \text{null} & \quad \text{Object := } \sigma[\cdot] \\
\text{java.lang.Cloneable := } \sigma[\cdot] & \\
\text{java.io.Serializable := } \sigma[\cdot] \\
\sigma[\cdot] := \tau[\cdot], \text{ if } \sigma := \tau & \\
\text{C := D, if } D \leq C
\end{align*}
\]

- It is reflexive, transitive, and anti-symmetric
Statements

\[ \text{C, L, X, } \sigma \vdash S_1 \quad \text{C, L, X, } \sigma \vdash S_2 \]

\[ \text{C, L, X, } \sigma \vdash S_1 \ S_2 \]

\[ \text{C, L, X, } \sigma \vdash \{ \tau \ n; \ S \} \]

\[ \text{C, L, X, } \sigma \vdash E: \tau \]

\[ \text{C, L, X, } \sigma \vdash E; \]
Control Statements

\[ C, L, X, \sigma |- E : boolean \quad C, L, X, \sigma |- S \]

\[ C, L, X, \sigma |- \text{if} (E) \quad \text{S} \]

\[ C, L, X, \sigma |- E : boolean \quad C, L, X, \sigma |- S_i \]

\[ C, L, X, \sigma |- \text{if} (E) \quad S_1 \quad \text{else} \quad S_2 \]

\[ C, L, X, \sigma |- E : boolean \quad C, L, X, \sigma |- S \]

\[ C, L, X, \sigma |- \text{while} (E) \quad \text{S} \]
Return Statements

\[ \sigma = \text{void} \]

\[ C, L, X, \sigma |- \text{return} \]

\[ C, L, X, \sigma |- E : \tau \quad \sigma := \tau \]

\[ C, L, X, \sigma |- \text{return} \ E \]
Throw Statement

\[ C, L, X, \sigma \vdash E : \tau \quad \text{throwok}(\{\tau\}, X) \]

\[ C, L, X, \sigma \vdash \text{throw } E \]

\[
\text{throwok}(A, B) \equiv \\
\forall \alpha \in A: \text{Error} := \alpha \lor \\
\text{RuntimeException} := \alpha \lor \\
\exists \beta \in B: \beta := \alpha
\]
Locals

\[ L(n) = \tau \]

\[
C, L, X, \sigma \vdash n : \tau
\]
Fields

\[
\begin{align*}
C, L, X, \sigma & \vdash \text{this: } C \\
D & \vdash \text{static } \tau \ f \\
C, L, X, \sigma & \vdash D . f : \tau \\
\end{align*}
\]

\[
\begin{align*}
C, L, X, \sigma & \vdash E : D \\
D & \vdash \tau \ f \\
C, L, X, \sigma & \vdash E . f : \tau \\
\end{align*}
\]
Assignments

\[
\begin{align*}
\text{C,L,X,σ} & \vdash E : \tau_2 \quad \text{L(n)} = \tau_1 \quad \tau_1 := \tau_2 \\
\text{C, L, X, σ} & \vdash n = E : \tau_1
\end{align*}
\]

\[
\begin{align*}
\text{C,L,X,σ} & \vdash E : \tau_2 \quad \text{D} \vdash \text{static } \tau_1 f \quad \tau_1 := \tau_2 \\
\text{C, L, X, σ} & \vdash D \cdot f = E : \tau_1
\end{align*}
\]

\[
\begin{align*}
\text{C,L,X,σ} & \vdash E_2 : \tau_2 \quad \text{C,L,X,σ} \vdash E_1 : D \quad \text{D} \vdash \tau_1 f \quad \tau_1 := \tau_2 \\
\text{C, L, X, σ} & \vdash E_1 \cdot f = E_2 : \tau_1
\end{align*}
\]
Arrays

\[ C, L, X, \sigma |- E_1 : \tau_1 [ ] \quad C, L, X, \sigma |- E_2 : \tau_2 \quad \text{num}(\tau_2) \]

\[ C, L, X, \sigma |- E_1 [E_2] : \tau_1 \]

\[ C, L, X, \sigma |- E_1 : \tau_1 [ ] \quad C, L, X, \sigma |- E_2 : \tau_2 \]

\[ C, L, X, \sigma |- E_3 : \tau_3 \quad \text{num}(\tau_2) \quad \tau_1 := \tau_3 \]

\[ C, L, X, \sigma |- E_1 [E_2] = E_3 : \tau_1 \]

\[ \text{num}(\sigma) \equiv \sigma \in \{ \text{byte, short, int, char} \} \]
Array Operations

\[ C, L, X, \sigma \vdash E : \tau[\cdot] \]

\[ C, L, X, \sigma \vdash E . \text{clone}() : \text{Object} \]

\[ C, L, X, \sigma \vdash E : \tau[\cdot] \]

\[ C, L, X, \sigma \vdash E . \text{length} : \text{int} \]

\[ C, L, X, \sigma \vdash E : \tau_2 \]

\[ C, L, X, \sigma \vdash \text{new } \tau_1[E : \tau_1[\cdot]] \]
Operators

\[ C, L, X, \sigma |- E : boolean \]
\[ C, L, X, \sigma |- ! E : boolean \]

\[ C, L, X, \sigma |- E_1 : \tau_1 \quad C, L, X, \sigma |- E_2 : \tau_2 \quad \text{num}(\tau_1) \quad \text{num}(\tau_2) \]
\[ C, L, X, \sigma |- E_1 * E_2 : int \]
Literals

C,L,X,σ |- true : boolean

C,L,X,σ |- 42 : int

C,L,X,σ |- "abc" : String

C,L,X,σ |- null : null

C,L,X,σ |- '@' : char
### Plus

| C, L, X, σ |- E₁: τ₁  | C, L, X, σ |- E₂: τ₂  | num(τ₁)  | num(τ₂)  |
|---------------|----------------|----------------|-----------|-----------|
|               |                |                |           |           |
| C, L, X, σ |- E₁+E₂: int |

<table>
<thead>
<tr>
<th>C, L, X, σ</th>
<th>- E₁: String</th>
<th>C, L, X, σ</th>
<th>- E₂: τ₂</th>
<th>τ₂ ≠ void</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C, L, X, σ</td>
<td>- E₁+E₂: String</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C, L, X, σ</th>
<th>- E₁: τ₁</th>
<th>C, L, X, σ</th>
<th>- E₂: String</th>
<th>τ₁ ≠ void</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C, L, X, σ</td>
<td>- E₁+E₂: String</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Equality

\[
\begin{align*}
C, L, X, \sigma \vdash E_1 : \tau_1 & \quad C, L, X, \sigma \vdash E_2 : \tau_2 \\
(num(\tau_1) \land num(\tau_2)) \lor \tau_1 := \tau_2 \lor \tau_2 := \tau_1 \\
\hline \\
C, L, X, \sigma \vdash E_1 = E_2 : \text{boolean}
\end{align*}
\]
C, L, X, σ |- E: \( \tau_1 \) \( (\text{num}(\tau_1) \land \text{num}(\tau_2)) \lor \tau_1 := \tau_2 \lor \tau_2 := \tau_1 \)

C, L, X, σ |- (\( \tau_2 \)) E: \( \tau_2 \)

C, L, X, σ |- E: \( \tau \) \( \tau := D \lor D := \tau \)

C, L, X, σ |- E instanceof D: boolean
Why Restrict Casts?

- Always succeeds, but upcasting is useful for selecting different methods
  \[ T_2 := T_1 \]

- Really useful, the object may or may not be a subclass of \( T_2 \)
  \[ T_1 := T_2 \]

- Always fails, don't bother to execute
  \[ T_1 \neq T_2 \land T_1 \neq T_2 \]
Method Invocation in Joos1

\[
C, L, X, \sigma \mid - \ E : D \quad C, L, X, \sigma \mid - \ E_i : \tau_i \\
D \mid - \ \tau \ m(\tau_1, ..., \tau_k) \ Y \\
\hline \\
C, L, X, \sigma \mid - \ E \ . \ m \ (E_1, ..., E_k) : \tau
\]

\[
C, L, X, \sigma \mid - \ E_i : \tau_i \\
D \mid - \ \text{static } \tau \ m(\tau_1, ..., \tau_k) \ Y \\
\hline \\
C, L, X, \sigma \mid - \ D \ . \ m \ (E_1, ..., E_k) : \tau
\]

(we add exception handling machinery later)
**Constructor Invocation**

\[
\begin{align*}
C, L, X, \sigma & \mid - E_i : \tau_i \\
D & \mid - (\tau_1, \ldots, \tau_k) \ Y \quad \text{throwok}(Y, X) \\
\hline
C, L, X, \sigma & \mid - \text{new } D (E_1, \ldots, E_k) : D 
\end{align*}
\]

\[
\begin{align*}
C, L, X, \sigma & \mid - E_i : \tau_i \quad C \mid - \text{extends } D \\
D & \mid - (\tau_1, \ldots, \tau_k) \ Y \quad \text{throwok}(Y, X) \\
\hline
C, L, X, \sigma & \mid - \text{super} (E_1, \ldots, E_k) 
\end{align*}
\]
Method Invocation in Joos2

\[
\begin{align*}
\forall D |- y \ m(y_1, \ldots, y_k) \ Z : (\forall i : y_i := \sigma_i) & \Rightarrow (\forall i : y_i := \tau_i) \\
C, L, X, \sigma |- E \cdot m(E_1, \ldots, E_k) : \tau
\end{align*}
\]

- This rule does not describe access modifiers
- It supports overloading (closest match) and subtyping in the arguments
Checking Exceptions in Joos1

\[
C, L, X, \sigma \vdash E : D \quad C, L, X, \sigma \vdash E_i : \tau_i \\
D \vdash \tau m(\tau_1, \ldots, \tau_k) \ Y \\
\text{throwok}(\bigotimes\{Y_i | D \vdash \tau m(\tau_1, \ldots, \tau_k) \ Y_i \}, X) \\
\hline
C, L, X, \sigma \vdash E . m(E_1, \ldots, E_k) : \tau
\]

where

\[
Y \bigotimes Z = Y \downarrow Z \cup Z \downarrow Y \\
Y \downarrow Z = \{ y \in Y | \text{throwok}(\{y\}, Z) \}
\]

Intuitively: check “the intersection” of all possible exceptions
class X extends Throwable {}
class Y extends Throwable {} 

abstract class A {
    public abstract void m() throws X;
}
interface B {
    void m() throws Y;
}
class Test {
    void foo(C c) {
        // no throws clause required!
        c.m();
    }
}
class X extends Throwable {}
class Y extends X {}

abstract class A {
    public abstract void m() throws X;
}
interface B {
    void m() throws Y;
}
class Test {
    void foo(C c) throws Y { // no throws X clause required!
        c.m();
    }
}
Judging Methods and Constructors

\[
\begin{align*}
C, [a_i \rightarrow \sigma_i], \{ X_i \}, \sigma |- S & \quad \text{Throwable} := X_i \\
C |- \sigma \ m(\sigma_1 \ a_1, \ldots, \sigma_k \ a_k) & \quad \text{throws} X_1, \ldots, X_n \ \{ S \}
\end{align*}
\]

\[
\begin{align*}
C, [a_i \rightarrow \sigma_i], \{ X_i \}, \text{void} |- S & \quad \text{Throwable} := X_i \\
C |- C(\sigma_1 \ a_1, \ldots, \sigma_k \ a_k) & \quad \text{throws} X_1, \ldots, X_n \ \{ S \}
\end{align*}
\]
Different Kinds of Type Rules

- **Axioms:**

  \[ C, L, X, \sigma \vdash \text{this}: C \]

- **Predicates:**

  \[ \tau_1 := \tau_2 \lor \tau_2 := \tau_1 \]

- **Inferences:**

  \[ C, L, X, \sigma \vdash E : D \quad D \vdash \tau f \]

  \[ C, L, X, \sigma \vdash E.f : \tau \]
Type Proofs

- A type **proof** is a tree in which:
  - nodes are inferences
  - leaves are axioms or true predicates

- A judgment is **valid**
  \[ \equiv \]
  it is the root of some type proof
A Type Proof

<table>
<thead>
<tr>
<th>x→A, y→B</th>
<th>x:A</th>
<th>A:=B ∨ B:=A</th>
</tr>
</thead>
<tbody>
<tr>
<td>x:A, y:B</td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="x">x→A, y→B</a>=A</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
C, [x→A, y→B], X, \sigma |- x:A \quad A := B ∨ B := A
\]

\[
C, [x→A, y→B], X, \sigma |- (B) x : B \quad B := B
\]

\[
C, [x→A, y→B], X, \sigma |- y = (B) x : B
\]

\[
C, [x→A, y→B], X, \sigma |- y = (B) x ;
\]

\[
C, [x→A], X, \sigma |- \{ B y ; y = (B) x ; \}
\]

\[
C, [], X, \sigma |- \{ A x ; \{ B y ; y = (B) x ; \} \}
\]

(Assuming that B is a subclass of A)

- Equivalent to a trace of a successful type checker run
Static Type Correctness

- A program is **statically type correct**
  \[
  \equiv \quad \text{all method and constructor judgments are valid}
  \]

- A type system is **sound**
  \[
  \equiv \quad \text{static type correctness implies type correctness}
  \]
Java's Type System is Unsound

- The following is a valid judgement:

\[
C, [b \rightarrow B, x \rightarrow A[], y \rightarrow B[], c \rightarrow C], X, \sigma \mid x=y; x[0]=c; b=y[0];
\]

where \( B \leq A \), \( C \leq A \), and \( B \) and \( C \) are incomparable

- Afterward, \( b \) contains an object of class \( C \)
- But its declared type only allows \( B \) objects
Java's type system is too greedy:

- Runtime type checks catch the unsound cases
By design, the rule for assignment of arrays has been chosen to be convenient but unsound.

It should have looked like:

- $\sigma := \sigma$
- int := byte
- int := char
- $\sigma[] := \text{null}$
- java.lang.Cloneable := $\sigma[]$
- java.io.Serializable := $\sigma[]$
- $\sigma[] := \tau[]$, if $\sigma := \tau$
- $C := \text{null}$
- Object := $\sigma[]$
- java.lang.Cloneable := $\sigma[]$
- java.io.Serializable := $\sigma[]$
- $C := D$, if $D \leq C$

But that would then introduce annoying slack...
Deciding valid judgments of an inference system depends on its rules and axioms and may be:

- undecidable
- (hyper)exponential time (backtracking)
- linear time

The Java type system is designed to be simple

A type checker performs a single traversal of the parse tree in linear time
Type Checking in Java

- Type checking is performed bottom-up
- The rule to apply is uniquely determined by the types of subexpressions and the syntax

- The type proof also specifies how to transform the program with respect to:
  - Autoboxing/unboxing
  - implicit string conversions
  - `concat(....)` code
Implicit Coercions

- Suppose we add a coercion type rule:

\[
\begin{array}{c}
C,L,X,\sigma |- E: \text{int} \\
C,L,X,\sigma |- E: \text{float}
\end{array}
\]

corresponding to the JVM bytecode \( \text{i2f} \)

- Type proofs are now no longer unique!
Ambiguous Type Proofs

\[
\begin{align*}
L(i) &= \text{int} \\
C, L, X, \sigma |- i : \text{int} \\
C, L, X, \sigma |- i : \text{float} \\
C, L, X, \sigma |- i + j : \text{float} \\
C, L, X, \sigma |- i + j = f : \text{boolean} \\
\end{align*}
\]

\[
\begin{align*}
L(j) &= \text{int} \\
C, L, X, \sigma |- j : \text{int} \\
C, L, X, \sigma |- j : \text{float} \\
C, L, X, \sigma |- i + j : \text{float} \\
C, L, X, \sigma |- i + j = f : \text{boolean} \\
\end{align*}
\]

\[
\begin{align*}
L(f) &= \text{float} \\
C, L, X, \sigma |- f : \text{float} \\
C, L, X, \sigma |- f : \text{float} \\
C, L, X, \sigma |- i + j = f : \text{boolean} \\
\end{align*}
\]
Semantic Coherence

- Different type proofs generate different transformed programs
- The type system is **semantically coherent** if they always have the same semantics
- The previous hypothetical type system is **not**:
  - $i = 2147400000$
  - $j = 83647$
  - $f = 2.14748365E9$
- Java chooses the proof with coercions as near the root as possible