CS 476 – Programming Language Design

William Mansky
Imperative Languages

• Arithmetic and boolean expressions
• Variables and assignment
• Control flow (conditionals, loops)
• Variable declarations
• Exceptions and exception handling
• Function declarations and calls
Object-Oriented Imperative Languages

• Arithmetic and boolean expressions
• Variables and assignment
• Control flow (conditionals, loops)
• Variable declarations
• Exceptions and exception handling
• Function declarations and calls
• Classes, methods, and objects
Object-Oriented Programming

• An object is a kind of value
• Objects have fields (object-specific variables) and methods (object-specific functions)

```java
ox := o.getx();
o.sety(ox);
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>3</td>
</tr>
<tr>
<td>y</td>
<td>5</td>
</tr>
<tr>
<td>getx()</td>
<td>return x;</td>
</tr>
<tr>
<td>sety(n)</td>
<td>y := n;</td>
</tr>
</tbody>
</table>
Object-Oriented Programming

• An *object* is a kind of value
• Objects have *fields* (object-specific variables) and *methods* (object-specific functions)

```
ox := getx(o);
sety(o, ox);
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>3</td>
</tr>
<tr>
<td>y</td>
<td>3</td>
</tr>
<tr>
<td>getx()</td>
<td>return x;</td>
</tr>
<tr>
<td>sety(n)</td>
<td>y := n;</td>
</tr>
</tbody>
</table>
Object-Oriented Programming

• An object is a kind of value

• Objects have fields (object-specific variables) and methods (object-specific functions)

• Different objects may provide the same method but have different code for it (dynamic dispatch)

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>3</td>
</tr>
<tr>
<td>y</td>
<td>3</td>
</tr>
<tr>
<td>getx()</td>
<td>return x;</td>
</tr>
<tr>
<td>sety(n)</td>
<td>y := n;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>4</td>
</tr>
<tr>
<td>theta</td>
<td>π/4</td>
</tr>
<tr>
<td>getx()</td>
<td>return r * cos(theta);</td>
</tr>
<tr>
<td>sety(n)</td>
<td>...</td>
</tr>
</tbody>
</table>

gtxt(o);
gtxt(o2);
Object-Oriented Programming

• An object is a kind of value
• Objects have fields (object-specific variables) and methods (object-specific functions)
• Different objects may provide the same method but have different code for it (dynamic dispatch)

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>3</td>
</tr>
<tr>
<td>y</td>
<td>3</td>
</tr>
<tr>
<td>getx()</td>
<td>return x;</td>
</tr>
<tr>
<td>sety(n)</td>
<td>y := n;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>4</td>
</tr>
<tr>
<td>theta</td>
<td>π/4</td>
</tr>
<tr>
<td>getx()</td>
<td>return r * cos(theta);</td>
</tr>
<tr>
<td>sety(n)</td>
<td>...</td>
</tr>
</tbody>
</table>
Object-Oriented Programming

• An object is a kind of value
• Objects have \textit{fields} (object-specific variables) and \textit{methods} (object-specific functions)
• Different objects may provide the same method but have different code for it (dynamic dispatch)
• Only an object’s methods can change its fields
• An object may belong to a \textit{class}, which describes a list of fields and methods the object contains
• Classes may have \textit{subclasses}, which extend them with more fields and methods
class Square extends Shape {
    int side;
    Square(int side){
        super();
        this.side = side;
    }
    int area(){
        return this.side * this.side;
    }
}

Square s = new Square(3);

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>side</td>
<td>3</td>
</tr>
</tbody>
</table>

int x = s.area();
// x will be 9
Java-Like Language: Syntax

\[ CL ::= \text{class } \text{id} \text{ extends } \text{id} \{ \ T \text{id}, \ldots, \ T \text{id}; \ K M \ldots M \} \]
\[ K ::= \text{id}(T \text{id}, \ldots, T \text{id})\{
 \quad \text{super}(\text{id}, \ldots, \text{id}); \ \text{this.} \text{id} = \text{id}; \ldots; \ \text{this.} \text{id} = \text{id}; \} \]
\[ M ::= T \text{id}(T \text{id}, \ldots, T \text{id})\{ \ C \} \]

\[ E ::= \# \ | \ E + E \ | \ \text{id} \ | \ E.\text{id} \ | \ \text{new } \text{id}(E, \ldots, E) \]
\[ C ::= \text{id} = E; \ | \ \text{id} = E.\text{id}(E, \ldots, E); \ | \ C C \]
\[ T ::= \text{int} \ | \ \text{id} \]
Java-Like Language: Subtyping

class A extends Object {
    int x;
    A(int x){
        super(); this.x = x;
    }
    int getx(){
        return this.x;
    }
}

class B extends A {
    int y;
    B(int x, int y){
        super(x); this.y = y;
    }
    int gety(){
        return this.y;
    }
}
Java-Like Language: Subtyping

(new A(5)).getx();
(new B(5, 6)).getx();

• Anywhere an object of class A is expected, an object of class B should work just as well!

\[ \Gamma \vdash e : \tau_1 \quad \tau_1 <: \tau_2 \]
\[ \frac{}{\Gamma \vdash e : \tau_2} \]
Java-Like Language: Subtyping

\[ \Gamma \vdash e : \tau_1 \quad \tau_1 <: \tau_2 \]
\[ \Gamma \vdash e : \tau_2 \]

\[ \Gamma(C) = \text{class } C \text{ extends } D \{ \ldots \} \]
\[ \Gamma \vdash C <: D \]

\[ \Gamma \vdash C <: D \quad \Gamma \vdash D <: E \]
\[ \Gamma \vdash C <: E \]
Java-Like Language: Types

- Types: int, any class name
- Rules:
  \[ \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \]
  \[ \Gamma \vdash e_1 + e_2 : \text{int} \]

  \[ \Gamma \vdash e : \tau_1 \quad \tau_1 \leq \tau_2 \]
  \[ \Gamma \vdash e : \tau_2 \]

  \[ \Gamma \vdash e \cdot f : \tau \]

  \[ \Gamma \vdash \text{new } C(e_1, \ldots, e_n) : \tau \]

  \[ \Gamma \vdash x = e \cdot m(e_1, \ldots, e_n) ; : \text{ok} \]
Java-Like Language: Types

• Types: int, any class name

\[
\Gamma \vdash e.f : \tau
\]

• \(e\) is an object of a class \(C\)
• \(C\) has a field \(f\) of type \(\tau\)

```java
class A extends Object {
    int x;
    ...
}
```

\(\text{objA.x : int}\)
Java-Like Language: Types

• Types: int, any class name

\[ \Gamma \vdash e : C \quad \text{fields} (\Gamma, C) = \cdots, \tau f \]
\[ \frac{}{\Gamma \vdash e.f : \tau} \]

• \( e \) is an object of a class \( C \)
• \( C \) has a field \( f \) of type \( \tau \)

```java
class A extends Object {
    int x;
    ...
}
objA.x : int
```
Java-Like Language: Fields

\[ \Gamma \vdash e : C \quad \text{fields}(\Gamma, C) = \cdots, \tau f \]

\[ \Gamma \vdash e.f : \tau \]

\[ \text{fields}(\Gamma, \text{Object}) = \text{nil} \]

\[ \Gamma(C) = \text{class } C \text{ extends } D \{ \tau_1 f_1; \cdots; \tau_n f_n; \cdots \} \]

\[ \text{fields}(\Gamma, C) = \text{fields}(\Gamma, D), \tau_1 f_1, \ldots, \tau_n f_n \]
Java-Like Language: Types

• Types: int, any class name

\[
\Gamma \vdash \text{new } C(e_1, \ldots, e_n) : \tau
\]

class A extends Object {
    int x;
    A(int x) {
        super();
        this.x = x;
    }
}

new A(5) : ?
Java-Like Language: Types

• Types: int, any class name

\[
\begin{align*}
\text{?} & \quad \Gamma \vdash \text{new } C (e_1, \ldots, e_n) : C \\
\end{align*}
\]

• \(e_1, \ldots, e_n\) have the right types

• The arguments to the constructor for \(C\) are exactly the fields of \(C\)

```java
class A extends Object {
    int x;
    A(int x) {
        super();
        this.x = x;
    }
}

new A(5) : A
```
Java-Like Language: Types

• Types: int, any class name

\[ \text{fields}(\Gamma, C) = \tau_1 f_1, \ldots, \tau_n f_n \quad \Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n \]
\[ \Gamma \vdash \text{new } C(e_1, \ldots e_n) : C \]

• \( e_1, \ldots e_n \) have the right types

• The arguments to the constructor for \( C \) are exactly the fields of \( C \)

class A extends Object {
    int x;
    A(int x){ super(); this.x = x; }
}

new A(5) : A
Java-Like Language: Types

• Types: int, any class name

\[ \Gamma \vdash x = e.m(e_1, \ldots, e_n) : \text{ok} \]

• \( e \) is an object of a class \( C \)
• \( C \) has a method \( m \)
• args and return of \( m \) are well typed

```java
class A extends Object {
    int x;
    int getx() { return this.x; }
}
```

```java
y = objA.getx();
```
Java-Like Language: Types

• Types: int, any class name

\[ \Gamma \vdash e : C \quad \text{methods}(\Gamma, C) = \cdots, \tau \ m(\tau_1 \ x_1, \ldots, \tau_n \ x_n) \]

\[ \begin{align*}
\Gamma & \vdash e_1 : \tau_1 & \cdots & \Gamma & \vdash e_n : \tau_n \\
\hline
\Gamma & \vdash x = e.m(e_1, \ldots, e_n); : \text{ok}
\end{align*} \]

• \( e \) is an object of a class \( C \)
• \( C \) has a method \( m \)
• \( \)args and return of \( m \) are well typed

\[
\begin{aligned}
\text{class A extends Object} & \{ \\
\text{int x;} & \text{int\ getx()} \{ \text{return this.x;} \} \\
\} & \\
\text{y = objA.getx();}
\end{aligned}
\]
Java-Like Language: Types

\[
\text{decls}(\Gamma) \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n, \_\text{ret} : \tau\} \vdash c : \text{ok}
\]

\[
\Gamma \vdash \tau m(\tau_1 x_1, \ldots, \tau_n x_n)\{c\} : \text{ok}
\]

• A method declaration is typed like a function declaration
Java-Like Language: Types

A method declaration is typed like a function declaration
Except that it is called on an object
Java-Like Language: Types

\[
\text{decls}(\Gamma) \cup \{\text{this} : C, x_1 : \tau_1, \ldots, x_n : \tau_n, \_\text{ret} : \tau\} \vdash c : \text{ok}
\]

\[
\Gamma, C \vdash \tau m(\tau_1 x_1, \ldots, \tau_n x_n)\{c\} : \text{ok}
\]

- A method declaration is typed like a function declaration
- Except that it is called on an object
- And it might be overriding a superclass’s method
Java-Like Language: Overriding

class A extends Object {
    int x;
    A(int x) {
        super(); this.x = x;
    }
    int getval() {
        return this.x;
    }
}

class B extends A {
    int y;
    B(int x, int y) {
        super(x); this.y = y;
    }
    int getval() {
        return this.x + this.y;
    }
}
Java-Like Language: Types

\[ \text{decls}(\Gamma) \uplus \{ \text{this} : C, x_1 : \tau_1, \ldots, x_n : \tau_n, \text{\_ret} : \tau \} \vdash c : \text{ok} \]

\[ \Gamma(C) = \text{class } C \text{ extends } D \{ \ldots \} \]

\[ \text{override\_safe}(\tau m(\tau_1 x_1, \ldots, \tau_n x_n), \Gamma, D) \]

\[ \Gamma, C \vdash \tau m(\tau_1 x_1, \ldots, \tau_n x_n)\{ c \} : \text{ok} \]

• A method declaration is typed like a function declaration
• Except that it is called on an object
• And it might be overriding a superclass’s method
Java-Like Language: Types

\[
\text{decls}(\Gamma) \cup \{\text{this} : C, x_1 : \tau_1, \ldots, x_n : \tau_n, \_\text{ret} : \tau\} \vdash c : \text{ok}
\]

\[
\Gamma(C) = \text{class } C \text{ extends } D \{\ldots\}
\]

\[
\text{override\_safe}(\tau m(\tau_1 x_1, \ldots, \tau_n x_n), \Gamma, D)
\]

\[
\Gamma, C \vdash \tau m(\tau_1 x_1, \ldots, \tau_n x_n)\{c\} : \text{ok}
\]

• \text{override\_safe}(\tau m(\tau_1 x_1, \ldots, \tau_n x_n), \Gamma, D) means that if a method named \(m\) is in \text{methods}(\Gamma, D), its type signature must be exactly \(\tau(\tau_1, \ldots, \tau_n)\)
Java-Like Language: Types

\[
\begin{align*}
\Gamma, C &\vdash K : \text{ok} \quad \Gamma, C \vdash M_1 : \text{ok} \quad \ldots \quad \Gamma, C \vdash M_j : \text{ok} \\
\Gamma &\vdash \text{class } C \text{ extends } D \{ \tau_1 f_1; \ldots; \tau_n f_n; K M_1 \ldots M_j \} : \text{ok}
\end{align*}
\]

\[
\begin{align*}
\text{fields}(\Gamma, D) &= \tau_1 x_1, \ldots, \tau_m x_m \\
\Gamma(C) &= \text{class } C \text{ extends } D \{ \tau_{(m+1)} x_{m+1}; \ldots; \tau_n x_n; \ldots \}
\end{align*}
\]

\[
\begin{align*}
\Gamma, C &\vdash C(\tau_1 x_1, \ldots, \tau_n x_n)\{ \text{super}(x_1, \ldots, x_m); \\
\text{this.} x_{m+1} = x_{m+1}; \ldots \text{ this.} x_n = x_n; \} : \text{ok}
\end{align*}
\]

• A class declaration is well typed if all of its methods are well typed and its constructor is correctly defined