

Lecture 12 — Objects

- Object-oriented programming in Java
- MP6
 - Representing objects
 - SOS rules for object, v. 1
 - Side effects and two-level state
 - SOS rules, v. 2

Exercise: Java programs

```
class C { int i;  
    int seti (int j) { i=j; return i; }  
    int geti () { return i; } }
```

```
class D { int f() { x = new C();  
    x.seti(10);  
    return x.geti(); // ?  
}
```

```
class E { int f() { x = new C();  
    y = x;  
    x.seti(10);  
    return y.geti(); // ?  
}
```

Exercise: Java programs (cont.)

```
class C { int i;  
    int seti (int j) { i=j; return i; }  
    int geti () { return i; } }
```

```
class D { int i; C c;  
    int seti (int j) { i=j; return i; }  
    C setc (C e) { c=e; return c; }  
    int geti () { return i; }  
    int getc () { return c; } }
```

```
class E { int f() { x = new D(); y = new C(); x.setc(y);  
    z = new D(); z.setc(y);  
    x.getc().seti(10);  
    return z.getc().geti(); // ?  
}
```

(Syntactic note: can write these in MiniJava; just make sure all expressions appear in assignment statements.)

Basics of object-oriented programming in Java

- An *object* is a heterogeneous collection of values, together with associated functions.
- The functions associated with an object depend solely on the class of the object. An object created by calling `new C()` contains the values given in `C`'s non-static fields, and the functions defined as methods in `C` (*ignoring inheritance*).
- Methods are called “on” an object, called the “receiver” of the method call: $e.f(e_1, \dots, e_n)$ — the value of e must be an object, and is the receiver of this call. The method called is the definition of f found in the class of the receiver.
- When executing a method, the receiver can be referred to by the name “this”. A field x of the receiver can be referred as `this.x` or `x`.

SOS rules for objects in MiniJava (v. 1)

- **Objects are just maps from variables to values, together with the name of the class:**

```
type value = IntV of int | StringV of string | BoolV of bool | NullV
           | ObjectV object
and varname = string and binding = (varname * value)
and object = id * (binding list) and state = binding list
```

- **SOS rules for new expressions:**

$(\text{NEW}) \text{ new } C(), \sigma, \pi \Downarrow$

$(\text{THIS}) \text{ this}, \sigma, \pi \Downarrow$

SOS rules for objects in MiniJava (v. 1), cont.

$$(\text{VAR}) \quad x, \sigma, \pi \Downarrow v$$

$$(\text{FIELD}) \quad x, \sigma, \pi \Downarrow v$$

$$(\text{VARASGN}) \quad x = e, \sigma, \pi \Rightarrow$$

$$(\text{FIELDASGN}) \quad x = e, \sigma, \pi \Rightarrow$$

SOS rules for objects in MiniJava (v. 1), cont.

(METHOD-CALL)

$e_0.f(e_1, \dots, e_n), \sigma, \pi \Rightarrow$

Which Java examples work?

Side effects

- *Side effect* = changes in state that results from a method call.
- With side effects, can evaluate the same expression twice and get different results:
 - Is this always true:
$$x = 5; y = a.f(x); y = a.f(x);$$
$$\equiv x = 5; y = a.f(x);$$
 - In Java?
 - In the MP5 version of MiniJava?

MP5 MJ has no side effects

THEOREM: In MP5 version of MiniJava: If $x \notin e$, $x=e; x=e \equiv x=e$. That is, for any states σ , σ' and program π ,

$$x=e; x=e, \sigma, \pi \Rightarrow \sigma' \quad \text{iff} \quad x=e, \sigma, \pi \Rightarrow \sigma'$$

● Can prove using SOS rules...

LEMMA: Let $Y = \{y_1, \dots, y_m\}$ be all variables in e , and σ and σ' two states that agree on Y ($\forall y \in Y, \sigma(y) = \sigma'(y)$). Then, for any v and π : $e, \sigma, \pi \Downarrow v$ iff $e, \sigma', \pi \Downarrow v$.

PROOF: By induction on the structure of the SOS “proof” of $e, \sigma, \pi \Downarrow v$. Ind. hyp. says it holds for every subcomputation; only interesting base case is variable reference, where the assumption about σ matching σ' is sufficient.

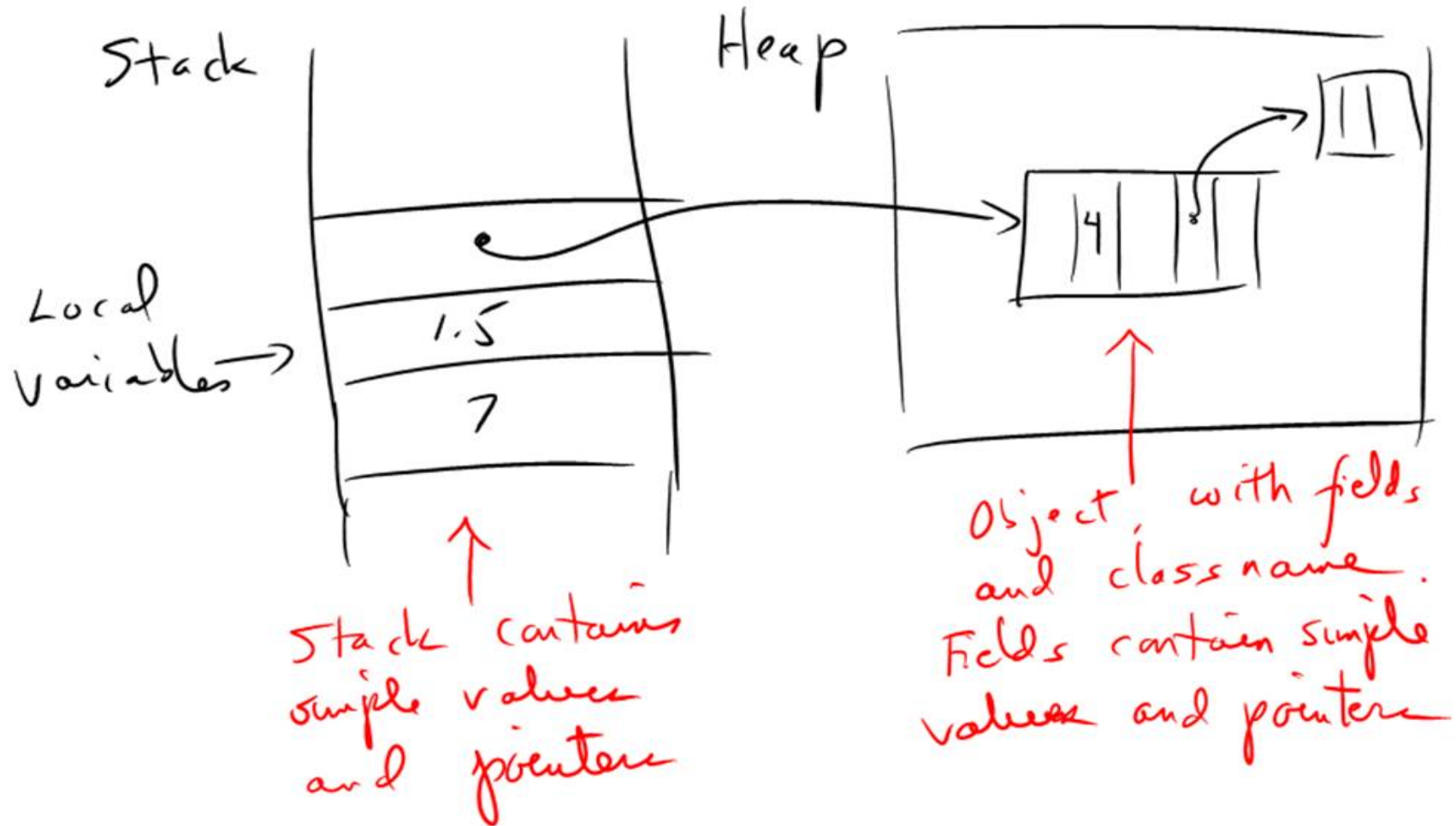
MP5 MJ has no side effects (cont.)

PROOF OF THEOREM: SOS rule for $x = e$; $x = e$; says to evaluate e in state σ , obtaining v , then evaluate e in $\sigma[v/x]$. The lemma tells us we will again get v , since σ and $\sigma[v/x]$ match on all variables occurring in e (since they differ only on x , and $x \notin e$). So we will produce $\sigma[v/x][v/x]$, which is obviously the same as $\sigma[v/x]$.

Side effects in Java

- If confined to primitive values (int, bool), Java also has no side effects. Objects have side effects (see earlier examples). (Note: strings are treated differently in Java and MJ.)
- On stack (in Java):
 - Primitive values are placed in stack. When passed to methods as arguments, value is copied to top of stack.
 - Objects are placed in the heap, with only a pointer on the stack. When passed to methods as arguments, or as the receiver, the *pointer* is copied to top of stack.
- Every reference to a field of an object goes through pointer to get to the object in the heap.
- The heap persists through the calls.

Stack and heap in Java



Two-level state in MJ

- Need to use same idea in implementation of MJ to implement objects correctly.
- Terminology: “stack” and “environment” synonyms; “heap” and “store” synonyms; “state” is a pair of an environment and a store.
- For MP6, have both environment and store. As in Java:
 - Environment contains simple value and pointers.
 - Store contains objects.
 - (*We will not implement arrays.*)

SOS rules (v. 2)

```
type varname = string
type classname = string
type stackvalue = IntV of int | StringV of string
                  | BoolV of bool | NullV | Location of location
and location = int
type environment = (varname * stackvalue) list
type heapvalue = Object of classname * environment
type store = heapvalue list
type state = environment * store
```

- In SOS rules, we still use σ for a state, but often write (ρ, η) instead, with ρ for environment and η for store.
- Consider evaluation of `new C`. Involves putting a new object in the store, so we must change the judgments for evaluation:

$$e, \sigma, \pi \Downarrow v, \eta$$

SOS rules (v. 2) (cont.)

```
type stackvalue = IntV of int | StringV of string
                  | BoolV of bool | NullV | Location of location
and location = int
type environment = (varname * stackvalue) list
type heapvalue = Object of classname * environment
type store = heapvalue list
type state = environment * store
```

- **Write an expression of type state for a state that contains variables x bound to 3 and y bound to an object of class C; C contains fields a and b , and in y these have integer values 4 and 5.**

SOS rules (v. 2) (cont.)

- Give the SOS rule for new:

$(\text{NEW}) \quad \text{new } C(), (\rho, \eta), \pi \Downarrow$

- New form of SOS rules reflected in new type of eval:

```
let rec eval (e:exp) ((env,sto) as sigma:state) (prog:program)
    : stackvalue * store =
```

- and the corresponding clause in eval (you can assume any auxiliary functions you think useful):

```
| NewId c ->
```

Threading the store

- The major new thing is that evaluation of expressions can have side effects, in particular, they can cause the store to change. (Evaluation of an expression *cannot* cause the environment to change.)
- Therefore, need to take the store from any expression evaluation and pass it along to the next expression evaluation. Can never discard any changes that occur in the store.

$$\begin{array}{l} \text{(BINOP-LESS)} \quad e_1 < e_2, (\rho, \eta), \pi \Downarrow \text{BoolV } (i_1 < i_2), \eta'' \\ \quad \quad \quad e_1, (\rho, \eta), \pi \Downarrow \text{IntV } (i_1), \eta' \\ \quad \quad \quad e_2, (\rho, \eta'), \pi \Downarrow \text{IntV } (i_2), \eta'' \end{array}$$

SOS rules (v. 2) (cont.)

(NOT) $!e, (\rho, \eta), \pi \Downarrow$

(INT-MULT) $e_1 * e_2, (\rho, \eta), \pi \Downarrow$

SOS rules (v. 2) (cont.)

(VAR) $x, (\rho, \eta), \pi \Downarrow$

(FIELD) $x, (\rho, \eta), \pi \Downarrow$

(METHOD-CALL)

$e_0.f(e_1, \dots, e_n), (\rho, \eta), \sigma, \pi \Downarrow v$

SOS rules (v. 2) (cont.)

- Rules for statements actually don't change — they always passed the state along from one to the next — except for assignment.

$(\text{VARASGN}) \quad x=e, (\rho, \eta), \pi \Rightarrow$

$(\text{FIELDASGN}) \quad x=e, (\rho, \eta), \pi \Rightarrow$