# CIS 500

# Software Foundations

# Fall 2002

# 25 November



## Administrivia

- There will be class as usual on Wednesday.



## Where we are...



## The (an) essence of objects

- Multiple representations
- Encapsulation of state with behavior
- Subtyping
- Inheritance (incremental definition of behaviors)
- "Open recursion" through `self`

## What's missing

The peculiar status of classes (which are both run-time and compile-time things)

Named types with declared subtyping

Recursive types

Run-time type analysis (casting, etc.)

(Lots of other stuff — e.g., ...?)



## Modeling Java



## Quick Check

How many non-Java-hackers in the room?...



## Models in General

No such thing as a "perfect model" — The nature of a model is to abstract away from details!

So models are never just "good": they are always "good for some specific purpose or set of purposes."

## Models of Java

Lots of different purposes ⟶ lots of different kinds of models

- Source-level vs. bytecode level
- Large (inclusive) vs. small (simple) models
- Models of type system vs. models of run-time features (not entirely separate issues)
- Models of specific features (exceptions, concurrency, reflection, class loading, ...)
- Models designed for extension



## Featherweight Java

Purpose: model the "core OO features" and their types and nothing else.



## Featherweight Java

Purpose: model the "core OO features" and their types and nothing else.

Things left out...

- Reflection, concurrency, class loading, inner classes, ...



## Featherweight Java

Purpose: model the "core OO features" and their types and nothing else.

Things left out...

- Reflection, concurrency, class loading, inner classes, ...
- Exceptions, loops, ...

## Featherweight Java

Purpose: model the "core OO features" and their types and nothing else.

Things left out...

- Reflection, concurrency, class loading, inner classes, ...
- Exceptions, loops, ...
- Interfaces, overloading, ...



## Featherweight Java

Purpose: model the "core OO features" and their types and nothing else.

Things left out...

- Reflection, concurrency, class loading, inner classes, ...
- Exceptions, loops, ...
- Interfaces, overloading, ...
- Assignment (!!)



## Things left in

- Classes and objects
- Methods and method invocation
- Fields and field access
- Inheritance (including open recursion through `this`)
- Casting



## Example

```
class A extends Object { A() { super(); } }

class B extends Object { B() { super(); } }

class Pair extends Object {
  Object fst;
  Object snd;

  Pair(Object fst, Object snd) {
    super(); this.fst=fst; this.snd=snd; }

  Pair setfst(Object newfst) {
    return new Pair(newfst, this.snd); }
}
```

## Conventions

For syntactic regularity...

- Always include superclass (even when it is `Object`)
- Always write out constructor (even when trivial)
- Always call `super` from constructor (even when *no* arguments are passed)
- Always explicitly name receiver object in method invocation or field access (even when it is `this`)
- Methods always consist of a single `return` expression
- Constructors always
  - Take same number (and types) of parameters as fields of the class
  - Assign constructor parameters to "local fields"
  - Call `super` constructor to assign remaining fields
  - Do nothing else



## Formalizing FJ



## Nominal type systems

Big dichotomy in the world of programming languages:

- Structural type systems:
  - What matters about a type (for typing, subtyping, etc.) is just its structure.
  - Names are just convenient (but inessential) abbreviations.
- Nominal type systems:
  - Types are always named.
  - Typechecker mostly manipulates names, not structures.
  - Subtyping is declared explicitly by programmer (and checked for consistency by compiler).



## Advantages of Structural Systems

Somewhat simpler, cleaner, and more elegant (no need to always work wrt. a set of "name definitions")

Easier to extend (e.g. with parametric polymorphism)

Caveat: when recursive types are considered, some of this simplicity and elegance slips away...

## Advantages of Nominal Systems

Recursive types fall out easily

Using names everywhere makes typechecking (and subtyping, etc.) easy and efficient

Type names are also useful at run-time (for casting, type testing, reflection, ...).

Java (like most other mainstream languages) is a nominal system.



## Representing objects

Our decision to omit assignment has a nice side effect...

The only ways in which two objects can differ are (1) their classes and (2) the parameters passed to their constructor when they were created.

All this information is available in the `new` expression that creates an object. So we can identify the created object with the `new` expression.

Formally: object values have the form `new C(v̄)`



## Syntax (terms and values)

| | | |
|---|---|---|
| `t ::=` | | terms |
| | `x` | variable |
| | `t.f` | field access |
| | `t.m(t̄)` | method invocation |
| | `new C(t̄)` | object creation |
| | `(C) t` | cast |
| `v ::=` | | values |
| | `new C(v̄)` | object creation |



## Syntax (methods and classes)

| | | |
|---|---|---|
| `K ::=` | | constructor declarations |
| | `C(C̄ f̄) {super(f̄); this.f̄=f̄;}` | |
| `M ::=` | | method declarations |
| | `C m(C̄ x̄) {return t;}` | |
| `CL ::=` | | class declarations |
| | `class C extends C {C̄ f̄; K M̄}` | |

## Subtyping

As in Java, subtyping in FJ is declared.

Assume we have a (global, fixed) class table CT mapping class names to definitions.

$$\frac{\mathsf{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{...\}}}{\mathtt{C} <: \mathtt{D}}$$

$$\mathtt{C} <: \mathtt{C}$$

$$\frac{\mathtt{C} <: \mathtt{D} \qquad \mathtt{D} <: \mathtt{E}}{\mathtt{C} <: \mathtt{E}}$$



## More auxiliary definitions

From the class table, we can read off a number of other useful properties of the definitions (which we will need later for typechecking and operational semantics)...



## Fields lookup

$$\mathit{fields}(\mathtt{Object}) = \emptyset$$

$$\frac{\mathsf{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\}} \qquad \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}}{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}},\ \overline{\mathtt{C}}\ \overline{\mathtt{f}}}$$



## Method type lookup

$$\frac{\mathsf{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\}} \qquad \mathtt{B\ m\ (\overline{B}\ \overline{x})\ \{return\ t;\}} \in \overline{\mathtt{M}}}{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{B}} \to \mathtt{B}}$$

$$\frac{\mathsf{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\}} \qquad \mathtt{m} \text{ is not defined in } \overline{\mathtt{M}}}{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \mathit{mtype}(\mathtt{m}, \mathtt{D})}$$

## Method body lookup

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\}} \qquad \mathtt{B\ m\ (\overline{B}\ \overline{x})\ \{return\ t;\}} \in \overline{\mathtt{M}}}{mbody(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{t})}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\}} \qquad \mathtt{m} \text{ is not defined in } \overline{\mathtt{M}}}{mbody(\mathtt{m}, \mathtt{C}) = mbody(\mathtt{m}, \mathtt{D})}$$



## Valid method overriding

$$\frac{mtype(\mathtt{m}, \mathtt{D}) = \overline{\mathtt{D}} \to \mathtt{D}_0 \text{ implies } \overline{\mathtt{C}} = \overline{\mathtt{D}} \text{ and } \mathtt{C}_0 = \mathtt{D}_0}{override(\mathtt{m},\ \mathtt{D},\ \overline{\mathtt{C}} \to \mathtt{C}_0)}$$