CS558 Programming Languages Fall 2023 Lecture 9a

Andrew Tolmach Portland State University

© 1994-2023

Programming "in the large"

Language features for modularity are crucial for writing large programs

Enable programmers to work on small part of a large system without understanding every detail of the whole

Idea: related definitions can be grouped together into a named unit

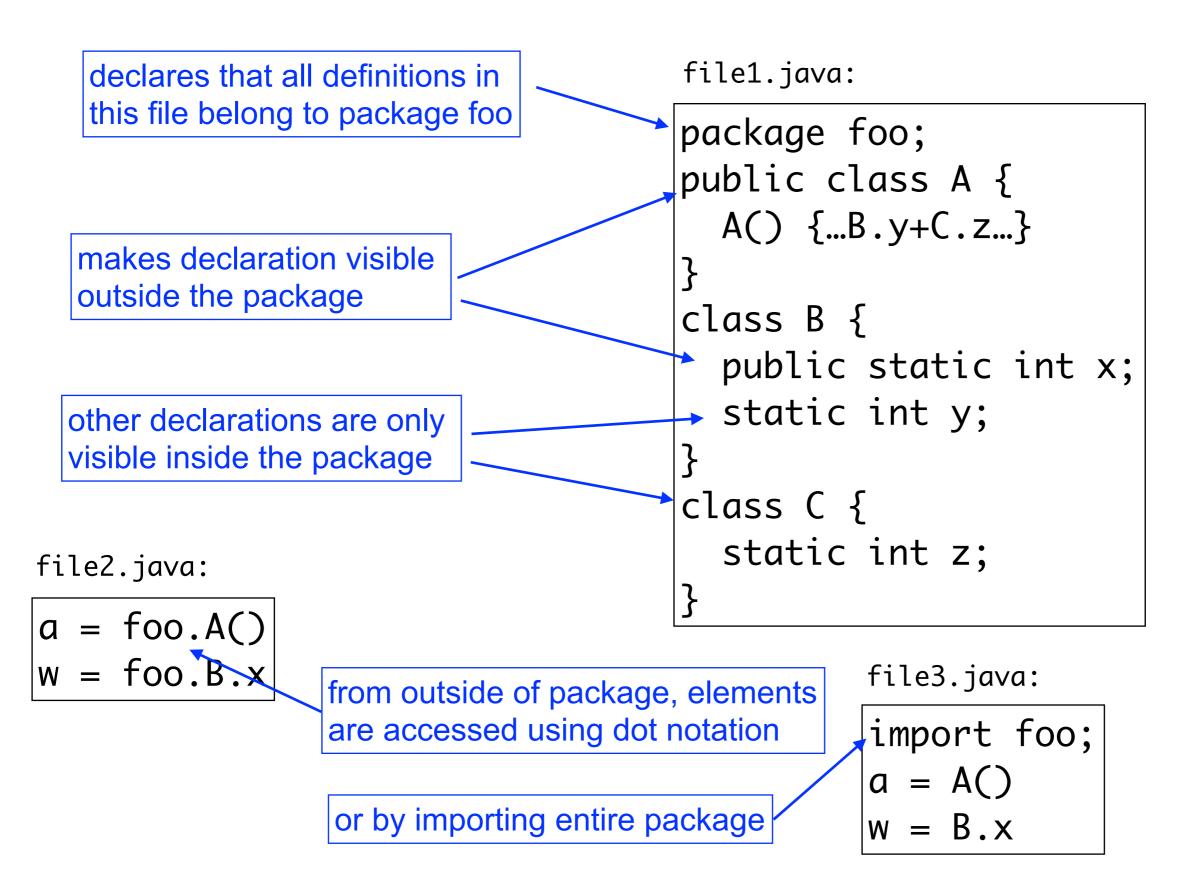
"module", "package", "class", "namespace", etc.

defining functions, types, variables, constants, exceptions, submodules, etc.

Often possible to export just a subset of definitions, specified by an interface, for use outside the module

Uses of definitions outside the module are qualified by module name

Example: Java packages



Hiding and Abstraction

A module typically encapsulates a set of services or a facility for use by other parts of the program

Sometimes we just want a sane way to manage global name space

Osually we also want to abstract over these services, by keeping some implementation info (internal functions, type definitions, etc.) hidden behind an interface

Allows independent development of service and client

Allows implementation to be changed without affecting client code

Improves clarity, maintainability, etc. of the code base

Interfaces

In many languages, interface to a module is given implicitly by using privacy modifiers on individual definitions

Some languages allow interfaces to be specified explicitly, maybe in a separate file from the implementation

An explicit interface is usually a set of top-level identifiers each with its type signature

Makes it possible to write, type-check and (maybe) compile client code based just on the interface

 (In a sense, whole interface is type signature of whole module)

Would also be nice to specify what interface elements do

but this is hard...

Abstract Data Types (ADT's)

Can we apply same kind of abstraction to user-defined types to make client code independent of type representation and operator implementation?

Ideally, user-defined types should have an associated set of operators, and clients should only be able to manipulate values via these operators

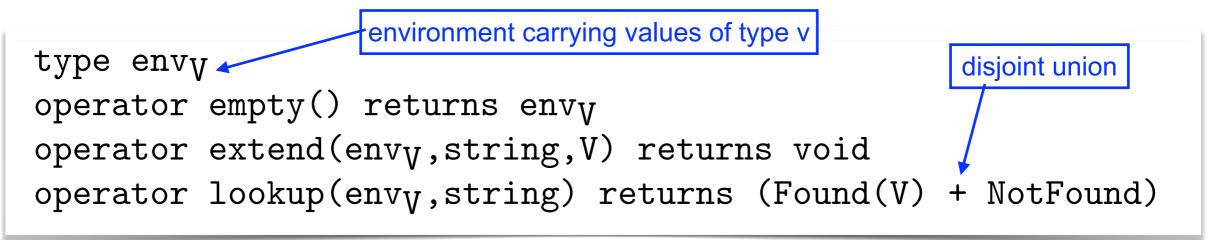
 (and maybe a few generic operators such as assignment or equality testing)

Clients should not be able to inspect or change the fields of the value representation

Idea: try to implement ADTs using modules and interfaces ability to do this is an important test for language's module facilities

ADT Example: Environments

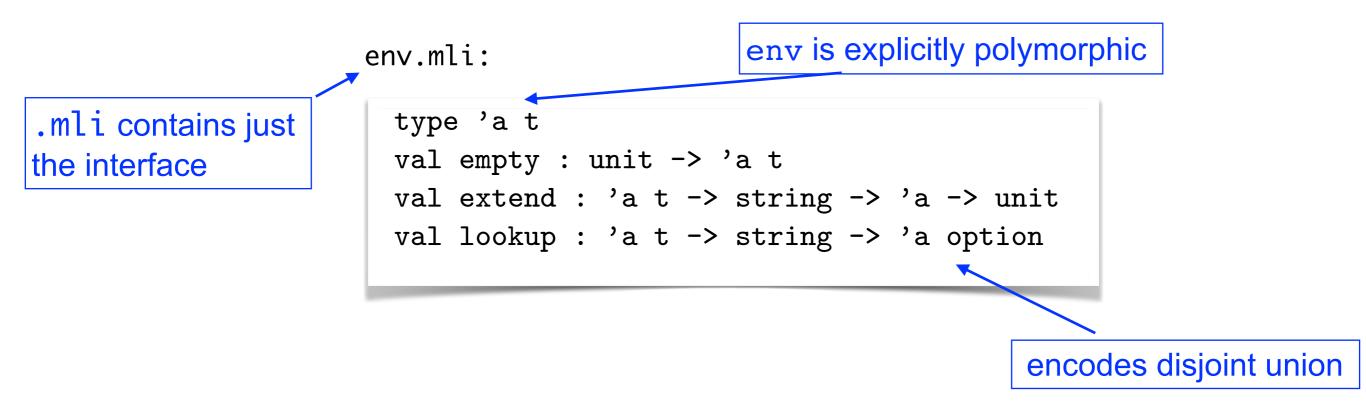
Consider an ADT for mutable environments mapping strings to values (of some arbitrary type), with this interface (in a made-up language):



One way to give a formal description of the desired ADT behavior is to state some laws that we want the operators to obey:

{ stmts } → P means "P is true after execution of stmts"

OCaml version: Interface and Client



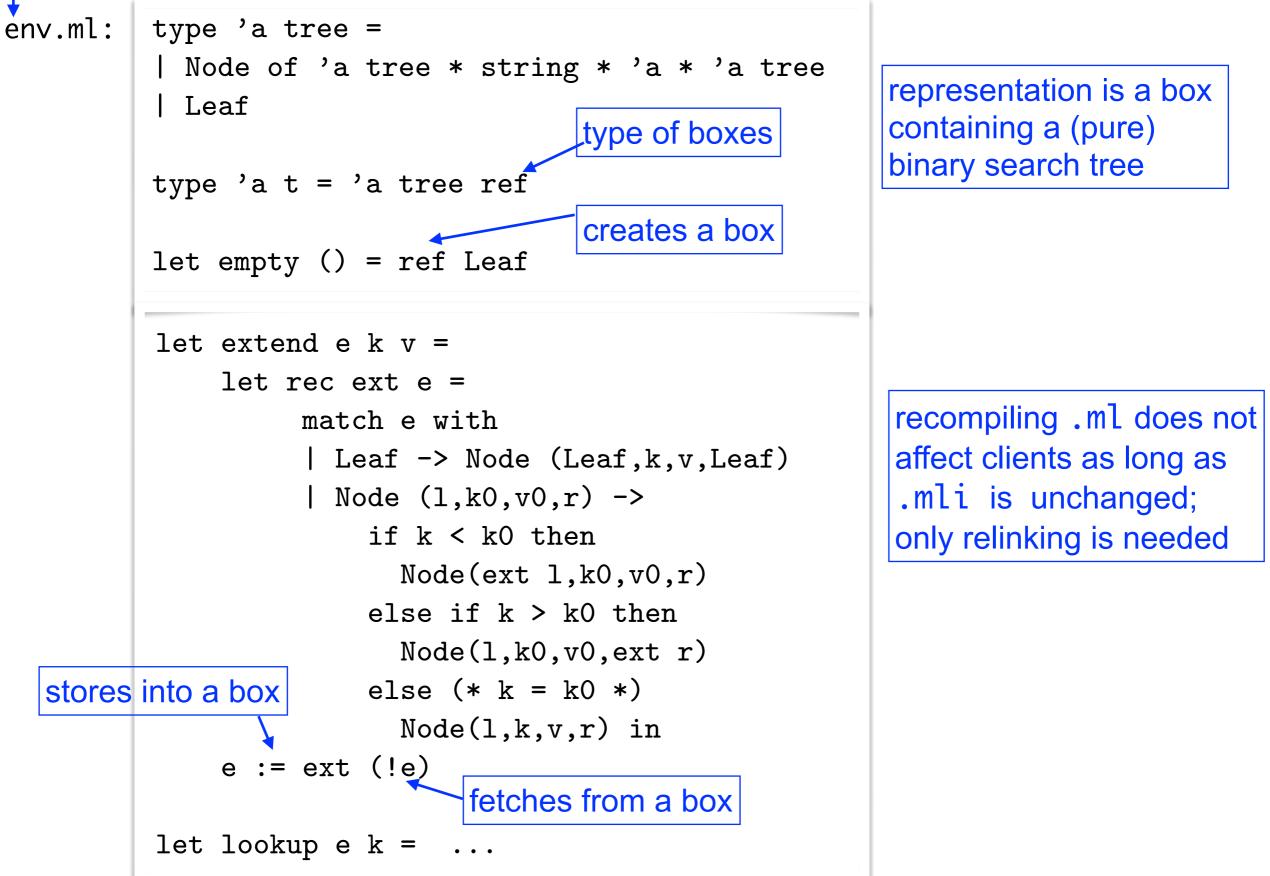
example-client.ml:

clients can be compiled using just the information in .mli even if implementation doesn't exist yet!

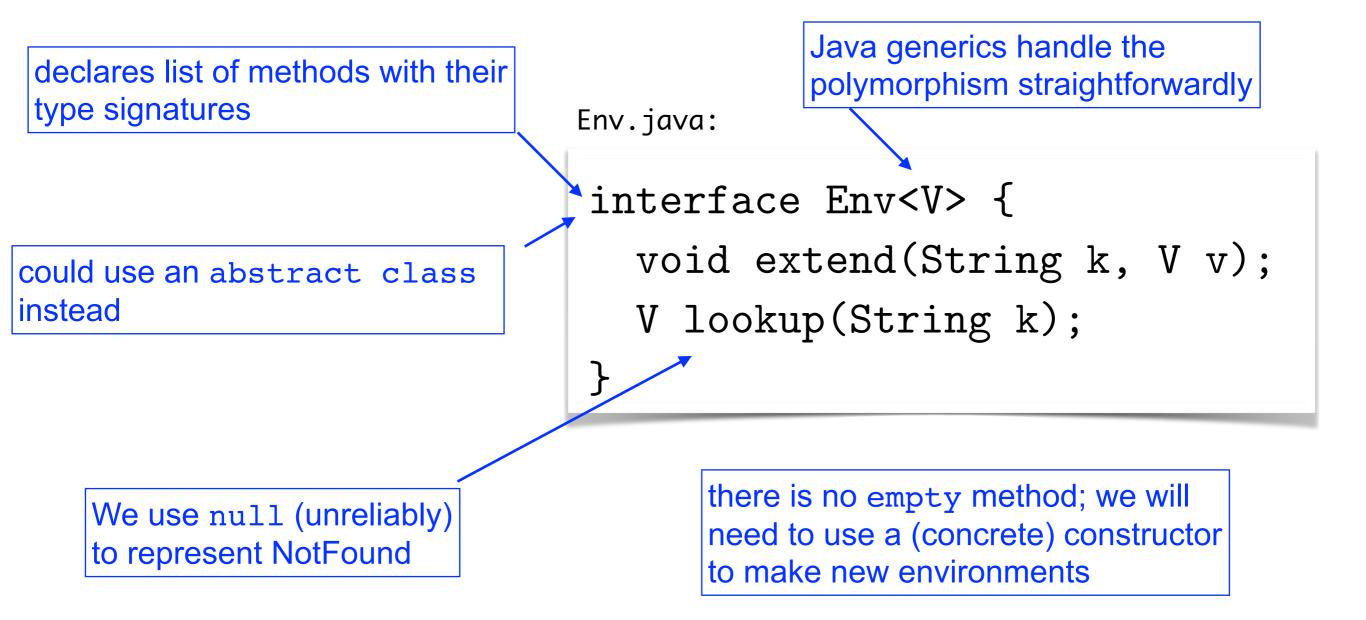
```
let main =
  let e = Env.empty() in
  Env.extend e "a" "alpha";
  Env.extend e "b" "beta";
  Env.extend e "a" "gamma";
  assert (Env.lookup e "a" = Some "gamma");
  assert (Env.lookup e "c" = None);
```

.ml contains just the implementation

OCaml version: Implementation

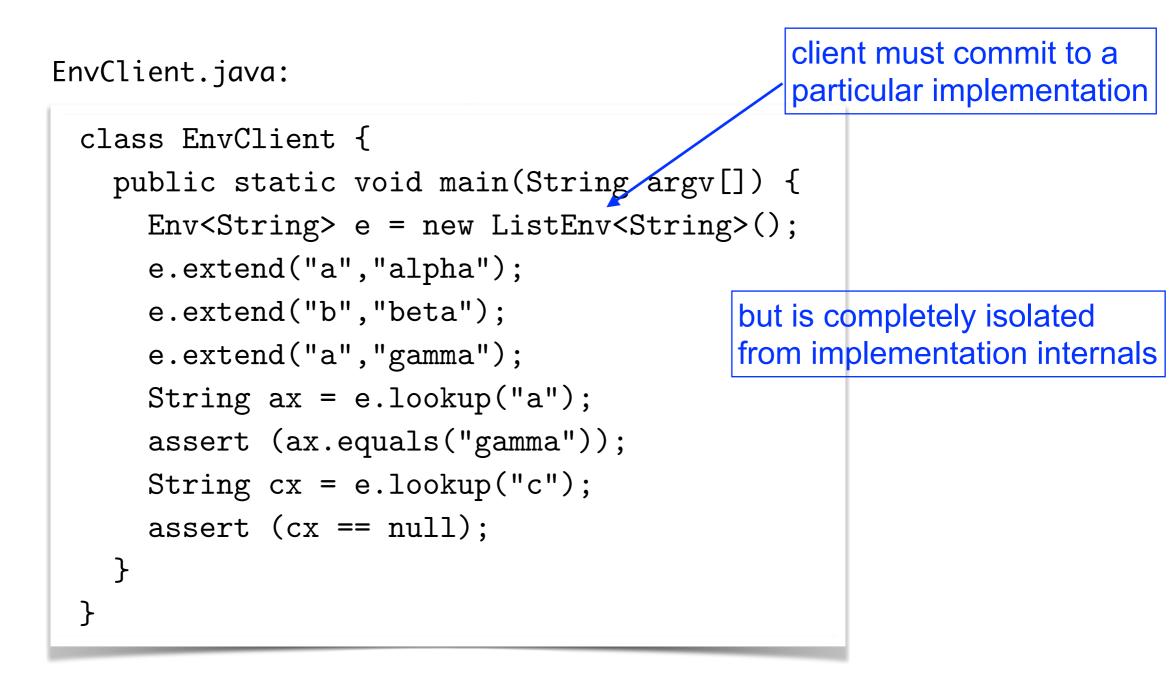


Java version: Interface



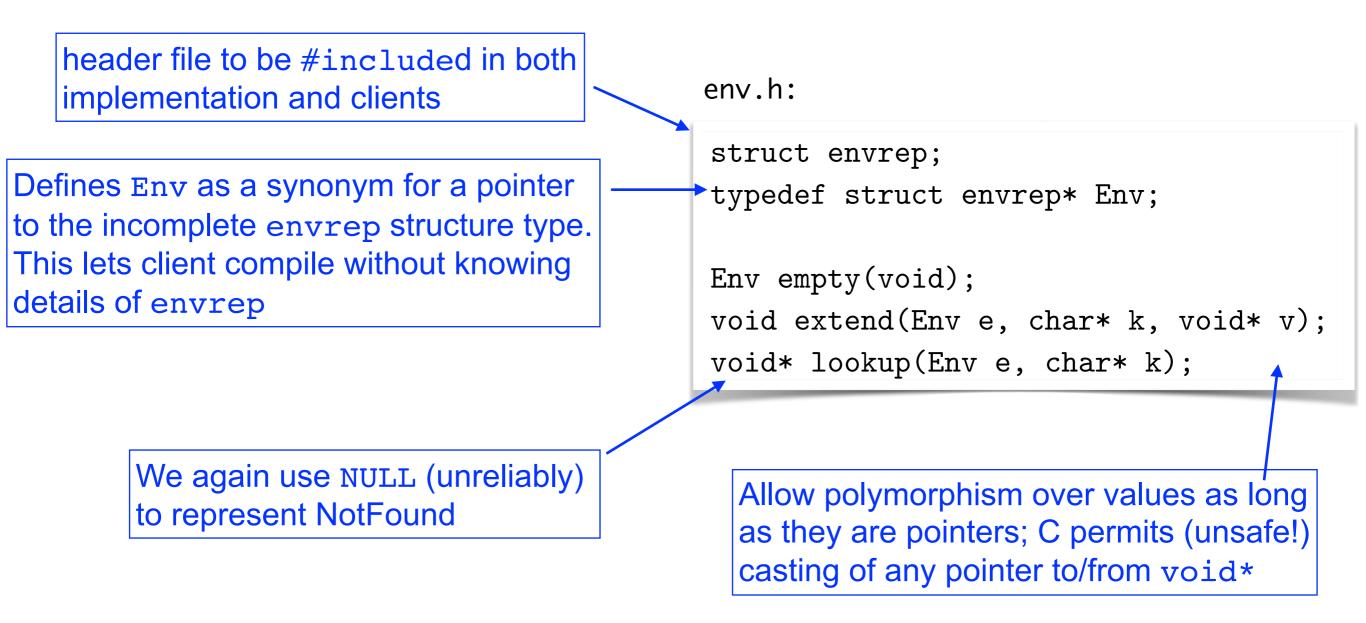
Java version: implementation ListEnv.java: class ListEnv<V> implements Env<V> { private class Node { String key; V value; // terminate with null Node next: uses private Node(String key, V value, Node next) { linked list this.key = key; this.value = value; this.next = next; representation } } private Node e; creates new public ListEnv() { empty env this.e = null; } public void extend(String k, V v) { can only run out of e = new Node(k, v, e);space if entire Java program heap is full public V lookup(String k) { for (Node u = e; u != null; u = u.next) if (k.equals(u.key)) Java garbage collector return u.value; takes care of freeing return null; } environments when they } are no longer accessible

Java version: client



C version: Interface

C doesn't have explicit module constructs, but separate compilation and header files can be used to simulate them (unsafely)



C Version: Implementation

one possible implementation

env.c:

```
envrep is private to this
                                                  void extend(Env e, char* k, void* v) {
#include "env.h"
                    file only by convention
#define SIZE 100
                                                     int i = find(e,k);
                    (not enforced by C)
                                                    if (i \ge 0)
struct envrep {
                                                       e->values[i] = v;
  int count;
                                                    else {
  char *keys[SIZE];
                                                       if (e->count >= SIZE)
                       uses a pair of arrays
  char *values[SIZE]:
                                                         exit(EXIT_FAILURE);
                       to represent env
};
                                                       e->keys[e->count] = k;
                                                       e->values[e->count] = v;
Env empty(void) {
                                                      e->count += 1;
all non-static functions
  Env e = malloc(sizeof(struct envrep));
                                                    }
                                                             are globally visible and must
  if (e == NULL) exit(EXIT_FAILURE);
                                                   }
                                                             be unique across entire program
  e \rightarrow count = 0;
                           can run out of room
                                                  void* lookup(Env e, char* k) {
  return e;
                           (unlike our abstract
                                                     int i = find(e,k);
}
                           ADT description)
                                                    if (i \ge 0)
static_int find(Env e, char* k) {
                                                      return e->values[i];
  int i;
                                                     else
  for (i = 0; i < e->count; i++)
                                                      return NULL;
    if (e \rightarrow keys[i] \rightarrow k)
                                                  }
                                                               can leak storage: ADT has
                          C enforces that
      return i;
                                                               no operator to delete an
                          static function is
  return -1;
                                                               environment, but C does not
                          invisible outside
}
                                                               have garbage collection
                          this file
```

C version: Client

example-client.c:

```
#include <assert.h>
#include "env.h"
int main(void) {
  Env e = empty();
  extend(e,"a","alpha");
                             we must "downcast" the void*
  extend(e,"b","beta");
                             values returned by lookup
  extend(e,"a","gamma");
  char* ax = (char*) lookup(e,"a");
  assert (strcmp(ax,"gamma") == 0);
  char* cx = (char*) lookup(e,"c");
  assert (cx == NULL);
}
```

Nothing in the C language prevents a bad client from including the concrete definition of envrep (or using a different definition altogether) and corrupting the representation arrays.

Universal operations

Although the idea of defining all operators for an ADT explicitly seems sensible, it can get quite tedious for the ADT author!

For every types, we will need a way to assign values or pass them as arguments. We may also expect to be able to compare them (at least for equality).

So many languages that support ADTs have built-in support for these basic operations, defined in an uniform way across all types — and sometimes also mechanisms for ADT authors to customize them.

Too much abstraction?

It is impossible for a compiler to generate client code for operations that move or compare data without knowing the size and layout of that data.

But these are characteristics of the type's implementation, not its interface!

So these "universal" operations break the abstraction barrier around the type and prevent separate compilation

A common fix (seen in our examples) is to require all abstract values to be boxed, giving a simple universal implementation for assignment and equality comparison