Programming “in the large”

- Language features for modularity are crucial for writing large programs.

- Idea: related definitions can be grouped together into a named unit.
  - “module”, “package”, “class”, “namespace”, etc.
  - defining functions, types, variables, constants, exceptions, sub-modules, 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.
Why use modules?

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

- Manage name space: avoid conflicts in the choice of top-level names.

- Hide representation and implementation details: module internals are not visible outside the module.

- Independent development: often, module implementations can be changed without requiring its clients to be rewritten (or, ideally, even recompiled).
Example: Java packages

declares that all definitions in this file belong to package foo

makes declaration visible outside the package

other declarations are only visible inside the package

file1.java:
```java
package foo;
public class A {
    A() {
        ...B.y+C.z...
    }
}
class B {
    public static int x;
    static int y;
}
class C {
    static int z;
}
```

file2.java:
```java
a = foo.A()
w = foo.B.x
```

file3.java:
```java
import foo;
a = A()
w = B.x
```
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 **naming scheme**, e.g. putting all the trigonometric functions into a library called Math.

Usually 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…
Example: OCaml Interfaces

In OCaml, each module foo typically lives in a file foo.ml and its interface in a separate file foo.mli.

env.mli:

```ocaml
type env
val empty : env
val extend : env -> string -> int -> env
val lookup : env -> string -> int option
```

env.ml:

```ocaml
type env = (string * int) list
let empty = []
let extend e k v = (k,v)::e
let rec lookup e k =
  match e with
  | (k',v')::e' when k = k' -> Some v'
  | _::e' -> lookup e' k
  | _ -> None
```

Clients of env can be compiled using just the information in env.mli, even if env.ml doesn’t exist yet!

Recompiling env.ml does not affect the client, as long as env.mli does not change: only relinking is required.

Here interface exposes existence of type env but not its representation as a list. So client can only operate on the underlying list using the three operators in the interface. Hence env is an abstract type...
Classes as Modules

- In class-based OO languages like C++ and Java, classes themselves can play role of modules.

- For example, in Java it is common to group together related static methods and variables into a class. (Indeed, they must go in some class.)

  - e.g. Integer class defines static methods for string-to-integer conversion and the MAX_VALUE constant.

  - Same “dot” notation is used to reference these, e.g. Integer.MAX_VALUE

- Role of module interfaces can be played by abstract class or interface definitions, which give names and types of members, but no concrete implementations.
Review: Primitive Types

- Primitive types like `float`, `bool` and `->` (functions) are usually **abstract**
- Internals of a value **cannot be inspected**
- Values can only be created and manipulated using a **fixed set of operators**
- E.g. we can test the sign of a floating point number using a relational operator
  
  ```
  if (a < 0) print "a is negative"
  ```

  rather than by trying to inspect the sign bit in the IEEE754 floating point standard representation

  ```
  if (a & 0x80000000 != 0) print "a is negative"
  ```

- This helps make programs **portable** across language implementations and hardware
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 signature (in a made-up language):

- type env_\_V
- operator empty() returns env_\_V
- operator extend(env_\_V, string, V) returns void
- operator lookup(env_\_V, string) returns (Found(V) + NotFound)

Here env_\_V means an environment carrying values of type V, and Found(V) + NotFound is a disjoint union type.

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

{ e = empty(); v = lookup(e, k) } \implies v = NotFound
{ v0 = lookup(e, k’); extend(e, k, v); v1 = lookup(e, k’) } \implies v1 = if k = k’ then Found(v) else v0

\{ \text{stmts} \} \implies P \text{ means “P is true after execution of stmts”}
C version: Interface

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

- header file to be #included in both implementation and clients

```
struct envrep;
typedef struct envrep* Env;

Env empty(void);
void extend(Env e, char* k, void* v);
void* lookup(Env e, char* k);
```

- Defines Env as a synonym for a pointer to the incomplete envrep structure type. This lets client compile without knowing details of envrep

- We use NULL (unreliably) to represent NotFound

- Allow polymorphism over values as long as they are pointers; C permits (unsafe!) casting of any pointer to/from void*
C Version: Implementation

env.c:

```c
#include "env.h"
define SIZE 100

struct envrep {
    int count;
    char *keys[SIZE];
    char *values[SIZE];
};

Env empty(void) {
    Env e = malloc(sizeof(struct envrep));
    if (e == NULL) exit(EXIT_FAILURE);
    e->count = 0;
    return e;
}

static int find(Env e, char* k) {
    int i;
    for (i = 0; i < e->count; i++)
        if (e->keys[i] == k)
            return i;
    return -1;
}

void extend(Env e, char* k, void* v) {
    int i = find(e,k);
    if (i >= 0)
        e->values[i] = v;
    else {
        if (e->count >= SIZE)
            exit(EXIT_FAILURE);
        e->keys[e->count] = k;
        e->values[e->count] = v;
        e->count += 1;
    }
}

void* lookup(Env e, char* k) {
    int i = find(e,k);
    if (i >= 0)
        return e->values[i];
    else
        return NULL;
}
```

C enforces that static function is invisible outside this file

envrep is private to this file only by convention (not enforced by C)

uses a pair of arrays to represent env

void extend(Env e, char* k, void* v) {
    int i = find(e,k);
    if (i >= 0)
        e->values[i] = v;
    else {
        if (e->count >= SIZE)
            exit(EXIT_FAILURE);
        e->keys[e->count] = k;
        e->values[e->count] = v;
        e->count += 1;
    }
}

C enforces that static function is invisible outside this file

all non-static functions are globally visible and must be unique across entire program

C enforces that static function is invisible outside this file

void* lookup(Env e, char* k) {
    int i = find(e,k);
    if (i >= 0)
        return e->values[i];
    else
        return NULL;
}

can run out of room (unlike our abstract ADT description)

void extend(Env e, char* k, void* v) {
    int i = find(e,k);
    if (i >= 0)
        e->values[i] = v;
    else {
        if (e->count >= SIZE)
            exit(EXIT_FAILURE);
        e->keys[e->count] = k;
        e->values[e->count] = v;
        e->count += 1;
    }
}

can leak storage: ADT has no operator to delete an environment, but C does not have garbage collection
C version: Client

element-client.c:

```c
#include <assert.h>
#include "env.h"

int main(void) {
    Env e = empty();
    extend(e,"a","alpha");
    extend(e,"b","beta");
    extend(e,"a","gamma");
    char* ax = (char*) lookup(e,"a");
    assert (strcmp(ax,"gamma") == 0);
    char* cx = (char*) lookup(e,"c");
    assert (cx == NULL);
}
```

we must “downcast” the `void*` values returned by `lookup`.

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.
Java version: Interface

interface Env<V> {
    void extend(String k, V v);
    V lookup(String k);
}

- Like an abstract class but can only contain instance methods (no variables or static members).
- Java generics handle the polymorphism straightforwardly.
- There is no empty method; we will need to use a (concrete) constructor to make new environments.
- We again use null (unreliably) to represent NotFound.
ListEnv.java: class ListEnv<V> implements Env<V> {
    private class Node {
        String key;
        V value;
        Node next;    // terminate with null
        Node(String key, V value, Node next) {
            this.key = key; this.value = value; this.next = next;
        }
    }
    private Node e;
    public ListEnv() {
        this.e = null;
    }
    public void extend(String k, V v) {
        e = new Node(k, v, e);
    }
    public V lookup(String k) {
        for (Node u = e; u != null; u = u.next)
            if (k.equals(u.key))
                return u.value;
        return null;
    }
}
EnvClient.java:

class EnvClient {
    public static void main(String argv[]) {
        Env<String> e = new ListEnv<String>();
        e.extend("a","alpha");
        e.extend("b","beta");
        e.extend("a","gamma");
        String ax = e.lookup("a");
        assert (ax.equals("gamma"));
        String cx = e.lookup("c");
        assert (cx == null);
    }
}

client must commit to a particular implementation

but is completely isolated from implementation internals
OCaml version: Interface and Client

**env.mli:**

```ocaml
let type 'a t
val empty : unit -> 'a t
val extend : 'a t -> string -> 'a -> unit
val lookup : 'a t -> string -> 'a option
```

**example-client.ml:**

```ocaml
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);
```
env.ml:
```ocaml
type 'a tree = 
  | Node of 'a tree * string * 'a * 'a tree 
  | Leaf

type 'a t = 'a tree ref

let empty () = ref Leaf

let extend e k v = 
  let rec ext e = 
    match e with
    | Leaf -> Node (Leaf,k,v,Leaf) 
    | Node (l,k0,v0,r) ->
      if k < k0 then
        Node(ext l,k0,v0,r)
      else if k > k0 then
        Node(l,k0,v0,ext r)
      else (* k = k0 *)
        Node(l,k,v,r) in
    e := ext (!e)

let lookup e k = ...
```

- Compared to environment type in the previous OCaml example, this one is polymorphic ('a is a type variable) and mutable.
- The `ref` constructor creates a storage location (a box). The `!` operator fetches the contents of the location and the `:=` operator overwrites the contents.
- The environment is represented by a binary search tree.
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.