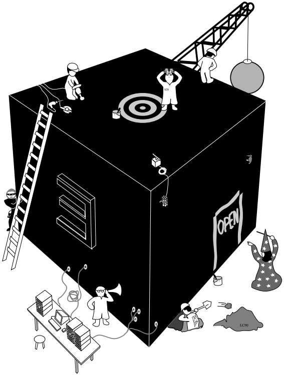

OPEN

Modules

> *Program modularization arose from the necessity of splitting large programs into fragments in order to compile them. ... It was soon realized that modularization had great advantages in terms of large-grain program structuring.*

Cardelli, 1996

Modularity

The maxims of modularity:

- minimize coupling — as independent as possible
- maximize cohesion — as focused as possible

D. Parnas. "A Technique for Software Module Specification," *Communications of the ACM*, volume 15, May 1972, pages 330–336.

D. Parnas. "On the Criteria To Be Used in Decomposing Systems Into Modules," *Communications of the ACM*, volume 15, number 12, December 1972, pages 1053–1058

Modules

A *module* is a construct for encapsulation, information hiding, and separate compilation, consisting of an interface specification and an implementation. It is a compile-time abstraction.

- encapsulation, aggregation – collecting pieces into a unit.
- information hiding – limiting access to the details
- representation independence – a change in the details does not affect the client
- separate compilation – capability to compile programs incrementally

Modules

In the shadow of many exciting development there has been a tendency to overlook the original purpose of modularization. Some language definitions specify what are to be the compilation units (e.g.: Ada), but others do not (e.g.: Standard ML). A paradoxical question then arises: when does a module system really support modularization (meant as separate compilation)?

Cardelli, 1996

Modularity

Each module should have well-defined *interface* or boundary. A *specification* is a description of the behavior of a module. Such as description may be informal as in English or formal as in ANNA, logic, etc.

In other languages such a construct is known as a *package* or a *structure*.

Ada, Modula 2, and Modula 3 interfaces are textually separate; Oberon's are not. Ada and Modula 2 allow nested modules; Modula 3 and Oberon do not.

C/C++ header files

Object-oriented classes

Modules

Mesa, Clu the first?

> *Modules provide a capability for partitioning a large system into manageable units. They can be used to encapsulate abstractions and to provide a degree of protection. In the design of Mesa, we were particularly influenced by the work of Parnas, who proposed information hiding as the appropriate criterion for modular decomposition, and by the concerns of Morris regarding protection in programming languages.*

Beschke, Morris Jr., and Satterthwaite,
"Early Experience with Mesa," *CACM*, 1977.


A.M.
TURING
AWARD
2008
BARBARA LISKOV
Developed the Liskov substitution principle
acm

Barbara Liskov (–)

When she was still a young professor at the Massachusetts Institute of Technology, she led the team that created the first programming language that did not rely on goto statements. The language, CLU (short for "cluster"), relied on an approach she invented — data abstraction — that organized code into modules. Every important programming language used today, including Java, C++ and C#, is a descendant of CLU.
In 2008, Liskov won the Turing Award.

CLU

```
complex = cluster is make_complex, real_part,
                     imaginary_part, plus, times
  rep = struct [ re, im : real ];
  make_complex = proc (x, y: real) returns (cvt)
    return (rep$(re:x, im:y));
  end make_complex;
  real_part = proc (z: cvt) returns (real)
    return (x.re);
  end real_part;
  imaginary_part = ...
  plus = proc (z, w: cvt) return (cvt)
    return (rep$(re:z.re+w.re, im:z.im+w.im));
  end plus;
  times = ...
end complex
```

cvt is reserved word calling for a conversion between the abstract type and its representation.

Modules

- interface specification – description of the items in the package
- implementation – programs, data structures, etc. to complete the facilities described in the interface
- client – some program that makes use of the facilities in the package

Modula 2, Modula 3, and Ada guarantee no violation of type security no matter the organization of the files or order of compilation.

C and C++ do not do inter module type checking!

```
link/main.c
link/aux.c
```

```
gcc -c aux.c                 # compile "aux.c", no errors

gcc aux.o main.c -o main     # link "main.o" and "aux.o", no

main                         # run main, !!!!
0.000000 -99771854684160876962015634666997225098865277688824
```

Modules in Ada

```
with List;          -- import other module
package Main is     -- this module
begin
   null;
end Main;
```

In the Ada language identifiers are *not* case sensitive, so `Main`, `mAiN`, and `main` are the same.
The Ada language does not specify how the computer retains modules, so, in general, an Ada implementation needs to know the correspondence between the name of the module and the operating system "handle" (the filename) of the module and its compiled artifacts.

Modules in Ada

GNAT requires the unit (module) name to be the same as the filename. It also uses a unit search path, cf., 3.3 Search Path of the GNAT user guide, as in the gcc.

2.3 File Naming Rules

> *The default file name is determined by the name of the unit that the file contains. The name is formed by taking the full expanded name of the unit and replacing the separating dots with hyphens and using lowercase for all letters.*

N.B. lowercase!

Aggregation in Ada

Note separation of specification and implementation. Note lack of cohesion (this is just an example, do not program this way).

```
package/junk.ads
package/junk.adb
package/use_junk.adb
```

Aggregation in Java

- Packages: aggregation of classes
- Classes: aggregation of values, classes

No clear separation; classes also can be used to aggregate classes.

```
class Aggregate {
  class B {/*...*/}
  class C {/*...*/}
}
```

Type Representation

- Integer: twos complement, sign/magnitude; cf `package/sign_integer_package.ads`
- Real: IEEE 754, ...
- Stack: array, linked list
- Graph: 2D array, adjacency list

The user does not care about the implementation (information hiding).

Type Representation

Consider first the well-known concepts of procedure specification and procedure implementation. Here is an example of a procedure specification:

```
//  Push an element into the stack
void push (int i);
```

Here is an example of a procedure implementation (body):

```
void push (int i) { data[size++] = i; }
```

What is a type specification? What is a type representation?

Type Representation

Often the only external interface to a type is its name.

```
type T;     type T is (Red, Green, Blue);
type T;     type T is array (1..3) of Positive;
type T;     type T is record
                Top: Natural;
                Data: array (1..20) of Character;
            end record;
```

Type Representation

But types in Ada can have (value) parameters. This is specified thusly:

```
type T (Size: Positive);
```

The implementation of the type might be:

```
type T is record
     Top: Natural;
      Data: array (1..Size) of Character;
 end record;
```

Abstract Data Types

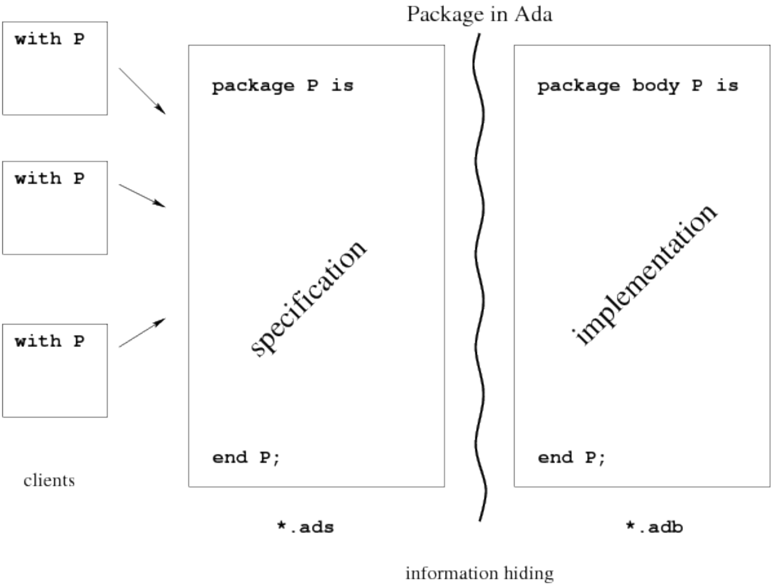
Modules usually focus, center around one type (cohesion).
Abstract data type. An *abstract data type* is a data type together with a complete set of relevant operations that form a new conceptual type of values. The representation of the type and the implementation of the operations are not important to the client and could be changed or improved.
Languages typically have two variations.
Opaque data type. An *opaque data type* is one whose structure or representation is hidden from the client.
Transparent data type. A *transparent data type* is one whose structure or representation is visible to the client.

Abstract Data Types

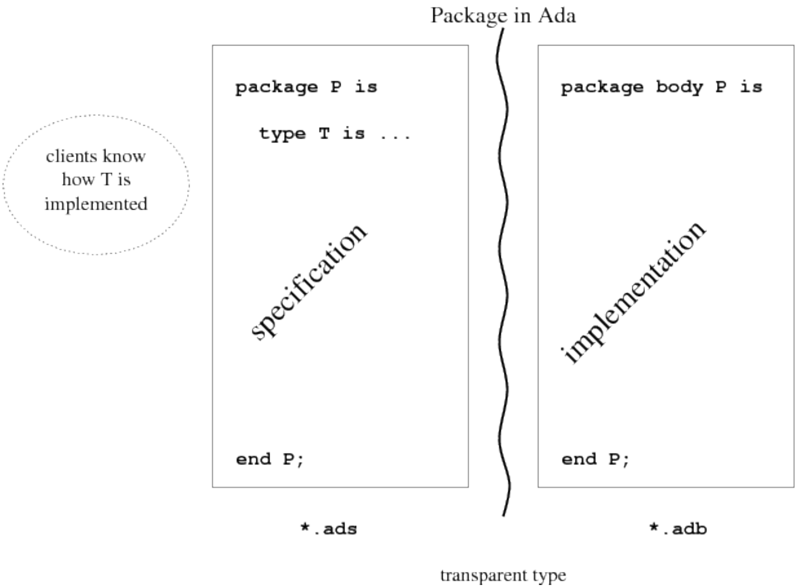
Some operations have special purposes, so they have special names.
Constructors. Procedures or functions that construct (allocate memory for) values out of their subparts are called *constructors*.
Predicates or *recognizers*. Boolean functions to distinguish the different ways values are constructed are called *predicates*.
Destructors or *selectors*. Functions that take constructors apart and return their constituent parts are called *destructors*.

And, finally, and less common:
Iterators. Procedures that access all the elements individually of data structure like a list are called *iterators*.

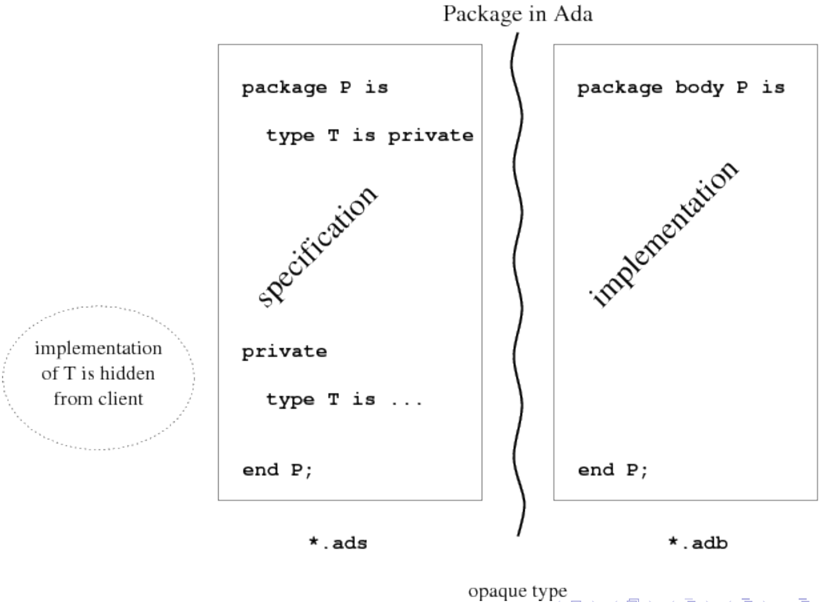
Specification/Implementation



Transparent Types



Opaque Types



Abstract Data Types

How to get an opaque type

- Ada: private section
- Modula-3: partial revelation
- SML: opaque type constraint
- Java, C++: declare members private
- Haskell: explicitly denying names to export

Haskell modules

In file `Main.hs`:

```
module Main (main) where
import Tree ( Tree(Leaf,Branch), fringe )

main = print (fringe (Branch (Leaf 1) (Leaf 2)))
```

In file `Tree.hs`:

```
module Tree ( Tree(Leaf,Branch), fringe ) where

data Tree a = Leaf a | Branch (Tree a) (Tree a)

fringe :: Tree a -> [a]
fringe (Leaf x)   = [x]
fringe (Branch left right) = fringe left ++ fring
```

Haskell modules

```haskell
module Stack (
   Stack, empty, isEmpty, push, top, pop) where
empty :: Stack a
isEmpty :: Stack a -> Bool
push :: a -> Stack a -> Stack a
top :: Stack a -> a
pop :: Stack a -> (a,Stack a)
newtype Stack a = StackImpl [a] -- opaque!
empty = StackImpl []
isEmpty (StackImpl s) = null s
push x (StackImpl s) = StackImpl (x:s)
top (StackImpl s) = head s
pop (StackImpl (s:ss)) = (s,StackImpl ss)
```

Abstract Data Types

```
ada/programs/fraction/fraction_package.ads
ada/programs/fraction/fraction_package.adb
ada/programs/fraction/opaque.ads
ada/programs/fraction/opaque.adb
m3/programs/rational/src/Rational.i3
m3/programs/rational/src/Rational.m3
m3/programs/opaque/src/Rational.i3
m3/programs/opaque/src/Rational.m3
sml/programs/rn.sml
```

Representation Independence

The compiled client does not know what the actual representation of the data type.

- Ada: any type in private section
- SML: any type
- Modula-3: REF types only
- Modula-2: REF types only

Pointers all have same size (regardless of what they point to), so clients may be isolated from changes in representation if the type is a pointer type.

Abstract Data Types

Abstract data type versus abstract state encapsulation.
A package in Ada may be like an abstract data type and provide (export) a single type with relevant operations on that type. Or, a package in Ada may encapsulate state by holding variables.

```
fraction/rational_adt.ads
fraction/rational_adt.adb
fraction/rational_number_package.ads
fraction/rational_number_package.adb
```

A stack package could be done either way.

Modules versus Classes

compile-time abstraction	run-time
statically instantiated	dynamically
collection	template
control visibility	extension
separate compilation	

See Clemens Szyperski, "Import is Not Inheritance."

Dynamic loading in Java gives rise to the possibility of “no such method” error (despite the type checking).

```
module/Main.java
module/version/C.java
module/bad/C.java
```

The following steps correctly compile and then fool Java.
Main.java is compiled only once.

```
$ javac -cp .:version Main.java
$ java Main
version 1
$ javac bad/C.java
$ java -cp bad:. Main
Exception in thread "main" java.lang.NoSuchMethodError: C.<
        at Main.main(Main.java:5)
```

The usual problem is a mistake in deployment of multiple versions of class files that is illustrated by something like this:

```
$ cp bad/C.class .
$ java Main
Exception in thread "main" java.lang.NoSuchMethodError: C.<
        at Main.main(Main.java:5)
```

The following steps correctly compile and then fool Java.

```
> rm *.class
> javac Main.java
> java Main
version 1
> cp version/C.class .
> java Main
version 2
> javac Main.java
> java Main
version 2
> cp bad/C.class .
> java Main
Exception in thread "main" java.lang.NoSuchMethodError: C.<
        at Main.main(Main.java:5)
```

Java Modules

Goals according to JSR 376.

- Reliable configuration—Explict dependence recognized at compile time
- Strong encapsulation—Explict export of module's capabilities
- Greater integrity—Prevent use of unintended classes

Python Encapsulation

No private attribute.

```
class Robot(object):
    def __init__(self):
        self.a = 123
        self._b = 123
        self.__c = 123

obj = Robot()
print(obj.a)
print(obj._b)
print(obj.__c)
```

Python Encapsulation

```
123
123
Traceback (most recent call last):
  File "test.py", line 10, in &lt;module&gt;
    print(obj.__c)
AttributeError: 'Robot' object has no attribute '
```