

Modules in SWI Prolog

- **Updated June 27, 2012** -

Creating and Populating Modules
Special Modules and Default Import
Predicate Lookup (Static and Dynamic Binding)
Using Modules like Objects

Modules

- Module = name space for predicate definitions
- Each file has an associated module
 - ◆ either the explicitly declared module

```
:- module('JTmeta', ← Module name
  Export list { [ ast_node_template/4 % (?Lang, ?NodeType, ?Arity, ?Template)
                , ast_arg_nr_name/4   % (?Lang, ?NodeType, ?ArgNumber, ?ArgName)
                , ast_parent/3       % (?Lang, ?Id, ?ParentId)
                , ...
                ]
  ).
...

```

Document the arguments and modes of exported predicates!

- ◆ or otherwise the implicit default module 'user'

Why Modules?

- Different Scopes / Namespaces
 - ◆ No name collisions
 - ◆ User-Defined Predicates
 - ◆ Library Predicates
- Structuring Large Applications
 - ◆ Separation of concerns
- Object-oriented Programming???

Creating and Populating Modules

module/2 directive
use_module/1
use_module/2
import/1

Exporting and Importing

- A module defines predicates that it exports
- Syntax
 - ◆ `:- module(name, [exported/1, ...]).`
 - ◆ This directive must precede any other directive or declaration in a file.
 - ◆ The module `name` does not need to be the same as the containing `filename` (but it is strongly recommended to keep them consistent!).
- Semantics
 - ◆ Exported predicates will be visible in the module that loads `name` via `consult(filename)` or `use_module(filename)`.
 - ◆ It is an error if two modules export the same predicates (name/arity) into the same importing module.
 - ◆ All non-exported predicates are not visible outside `name`
 - ⇒ ... unless some other module uses `import/2` to bypass the protection

“use_module” versus “consult”

- Think of “use_module/1” as defined by

```
use_module(File) :-  
    contains_module(File)  
    -> consult(File)  
    ; report_missing_module(File)  
    .
```

- Recommended way of loading module files
 - ◆ Better because it checks that there is actually a module and also makes the intention explicit → easier to understand code

```
:- module( pureImporter, [ myownstuff/4 ] ).  
  
:- use_module('JTmeta').
```

- This also works, but lacks the above advantages:

```
:- module( pureImporter, [ myownstuff/4 ] ).  
  
:- consult('JTmeta').
```

Implicit import

- Let **M** be a module declared in the file `filename`.
 - ◆ Exported predicates of **M** will be visible in any **Importer** module in whose context `consult(filename)` or `use_module(filename)` is executed
 - ◆ This is called “implicit import”.
- Importing modules can use the imported predicates internally ...

```
:- module( pureImporter, [ myownstuff/4 ] ).  
  
:- use_module('JTmeta').  
  
... internal use of predicates from JTmeta ...
```

- ... but can also re-export (some of) them

```
:- module( 're-exporter',    [ myownstuff/4  
                             , ast_node_template/4  
                             ] ).  
  
:- use_module('JTmeta').  
  
... internal use of predicates from JTmeta ...
```

← Reexported predicate
from JTmeta

Re-Export Shortcut

- Re-Exporting all predicates of an imported module

- ◆ `reexport(+Files)`

- ⇒ Import all predicates from the export list of each file in Files like `use_module/1`
- ⇒ ... but additionally re-export all imported predicates
- ⇒ ... without having to manually list them in the export list of the importing module!

```
:- module(cultivate_naming, []).  
  
:- reexport( [term_cache, term_metrics, term_graph] ).
```

- ◆ `reexport(+File, ImportList)`

- ⇒ Import only predicates from ImportList, like `use_module/2`
- ⇒ ... but additionally re-export all imported predicates

```
:- module(cultivate_naming, []).  
  
:- reexport( module_4, [ pred/1 ] ).  
:- reexport( module_5, except([do_not_use/1]).
```


Name Clashes

- Name Clash Rule: The same predicate may not be imported into the same module from two different modules!
 - ◆ Similar to exclusion of multiple inheritance in Java.
- Note: It is legal that different modules offer to export the same predicate:

```
% In file f1:
:- module( m1, [ p/1 ] ).
p(m1).
```

```
% In file f2:
:- module( m2, [ p/1 ] ).
p(m2).
```

- Only importing into the same module is illegal

```
% In file "ambiguous":
:- module( 'ambiguous', [...] ).
:- use_module(f1).           % implicitly imports p/1
:- use_module(f2).           % implicitly imports p/1
...
```

```
?- use_module( 'ambiguous' ).
```

Explicit Import: Avoiding Name Clashes (1)

`use_module(+File, +ImportList)`

- ◆ Consults File, importing only the predicates in ImportList
- ◆ Also allows for **renaming** or “**import-everything-except**”:

```
:- use_module(library(lists), [ member/2,  
                                , append/2 as list_concat  
                                ]).  
:- use_module(library(option), except([meta_options/3])).
```

New name in
importing module

- ◆ Warns if a predicate in ImportList is not in the ExportList of the module defined in File (but imports it nevertheless)
- ◆ Only works for static modules (declared in a file)

Explicit Import: Avoiding Name Clashes (2)

`import (+QualifiedHead)`

- ◆ QualifiedHead = Module:Head
- ◆ Imports definition of Head from any module Module
- ◆ ... also from dynamically created modules
- ◆ ... also for predicates that are not exported

Static use

```
% In file "noexport.pl":  
:- module(noexport, []).  
  
p(noexport).
```

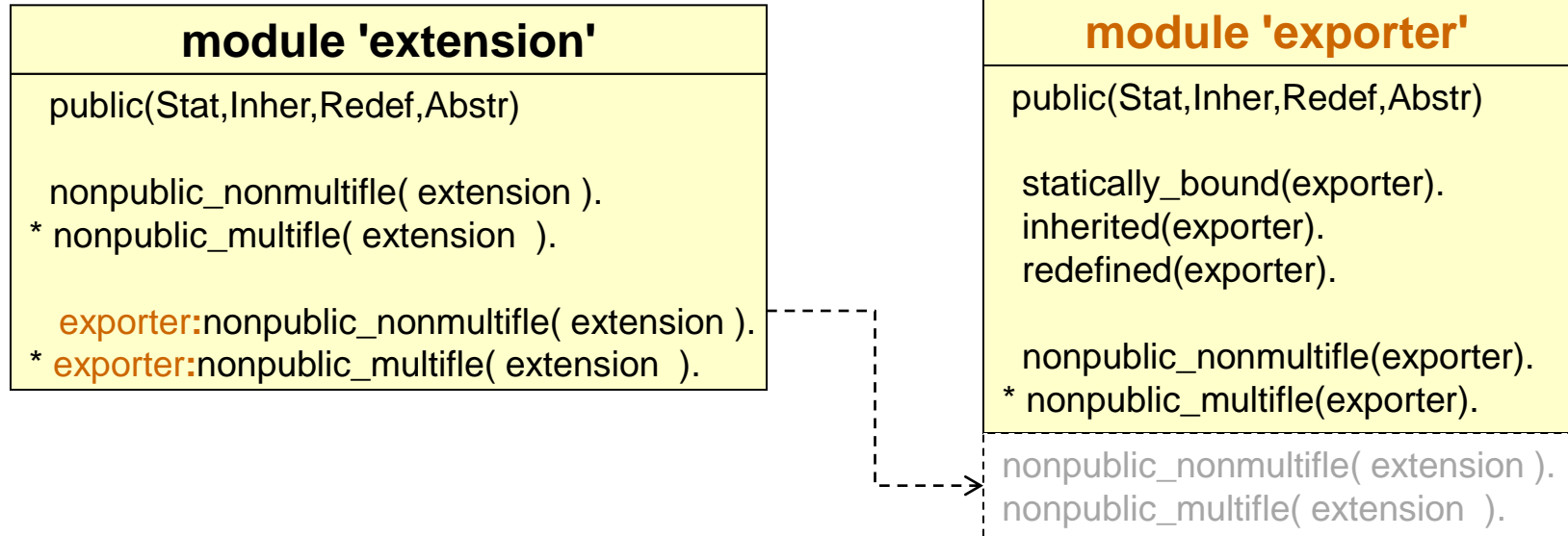
```
% In file "explicit_static_import":  
:- module(import, []).  
  
% Assumes noexport has been loaded:  
:- import(noexport:p/1).  
  
:- p(X), write(X).
```

Dynamic use

```
:- % anytime:  
   assert(newm:p(dynamic) ),  
   newm:import(noexport:p/1).
```

Creates a new module "newm" and imports into it the predicate "p" from module "noexport"

A module can declare clauses for others by prepending a module qualifier to a clause head



- One module “injects” definitions into another one
 - ◆ This is akin of “aspect-oriented introduction” mechanism
- May be as powerful and as confusing as aspect-orientation
 - ◆ Use it sparingly and document it thoroughly!!!

Summary: Populating Modules

- Predicate P is defined in module M if it is
 - ◆ declared in M 's file, without module qualifier in clause heads
 - ⇒ $P \text{ :- Body.}$
 - ◆ declared in another file with clause heads qualified with M
 - ⇒ $M:P \text{ :- Body.}$
 - ◆ asserted dynamically into M
 - ⇒ $M:\text{assert}((P \text{ :- Body}))$ or $\text{assert}(M:(P \text{ :- Body}))$
- Predicate P (that is not defined in M) is visible to M if it is imported
 - ◆ $\text{use_module}(+FileOfM)$
 - ⇒ consults $FileOfM$ and imports predicates from the export list of M
 - ◆ $\text{use_module}(+FileOfM, +ImportList)$
 - ⇒ consults $File$ and imports only the predicates in $ImportList$
 - ◆ $\text{import}(+QualifiedHead)$
 - ⇒ $QualifiedHead = M:P$
 - ⇒ imports definition of P from already created module M

Special Modules

Special modules and default import
Import to “user” ▶ When to do and when to avoid

Special Modules and Default Import

- SWI-Prolog has two special modules

- ◆ “system”

- ⇒ contains all built-in predicates

- ◆ “user”

- ⇒ module that the user addresses through the console or from files with no explicit module declaration

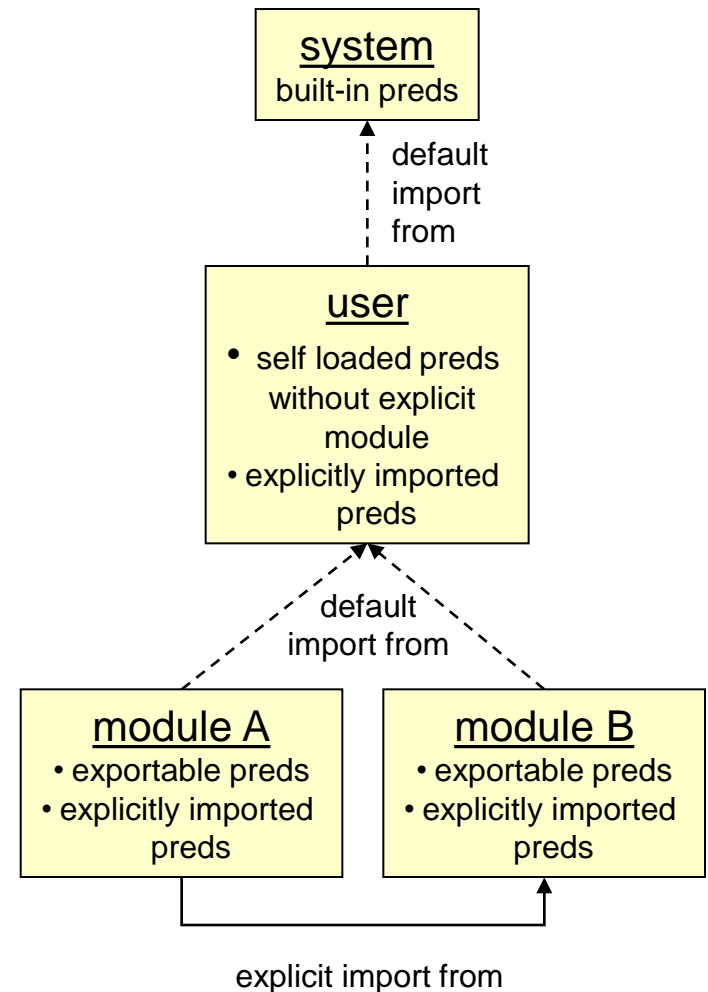
- Default import

- ◆ “user” automatically imports from “system”

- ◆ All other modules automatically import from “user”

- ⇒ Thus they can use all predicates imported into user without explicitly importing them

- ⇒ This includes the built-in predicates imported from “system” to “user”



Importing to „User“

- Intention

- ◆ distribute **common** definitions over all program modules
- ◆ ... without the need to import them each time explicitly

- Typical uses

- ◆ import your own global libraries / utilities
- ◆ import SWI-Prolog libraries
- ◆ redefine built-in predicates
- ◆ Typically, the load file of a large application looks like this:

```
:- use_module(compatibility).    % load XYZ-Prolog compatibility

:- use_module(
    [ error
      , goodies
      , debug
      , virtual_machine
      , ...
    ] ).
    % load generic parts
    % errors and warnings
    % general goodies (library extensions)
    % application specific debugging
    % virtual machine of application
    % other generic stuff
```


Importing to „User“:

```
:- module(a, []).  
q(N) :- p(N)
```

```
:- module(c, []).  
r(1).  
r(2).
```

```
:- module(b, [p/1]).  
p(N) :- c:r(N)
```

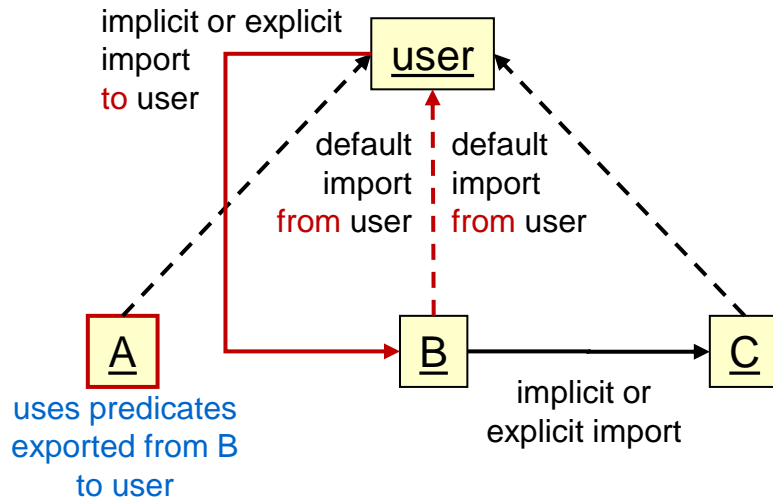
Question:

What is the issue?

Importing to „User“: Use it sparingly!

Import to “user”

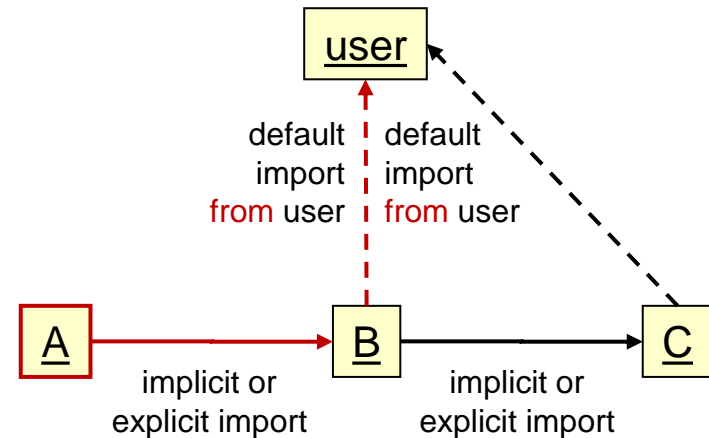
- Only use import to “user” for truly global library predicates!



- It creates bad dependencies
 - ◆ Hidden dependencies ☹️
 - ◆ Cyclic dependencies ☹️

Import to real clients

- Better architecture: Import from B to its real client A



- Better dependencies
 - ◆ explicit
 - ◆ acyclic

Predicate Lookup or “How to find my predicate?”

The context module ▶ `module_transparent/1` and `context_module/1`

Predicate lookup ▶ Meta-Predicates and the context module

Static and dynamic binding ▶ Dynamic binding idiom

Defining own meta-predicates

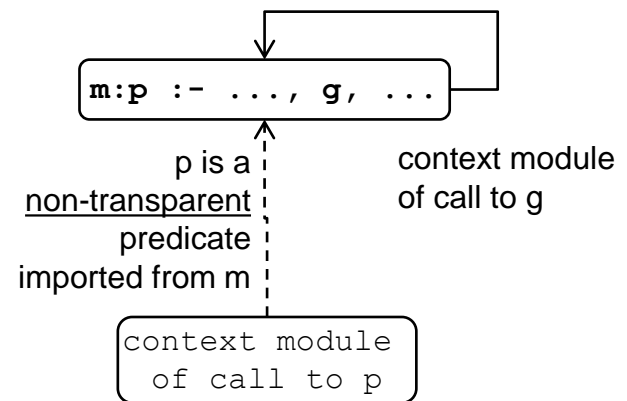
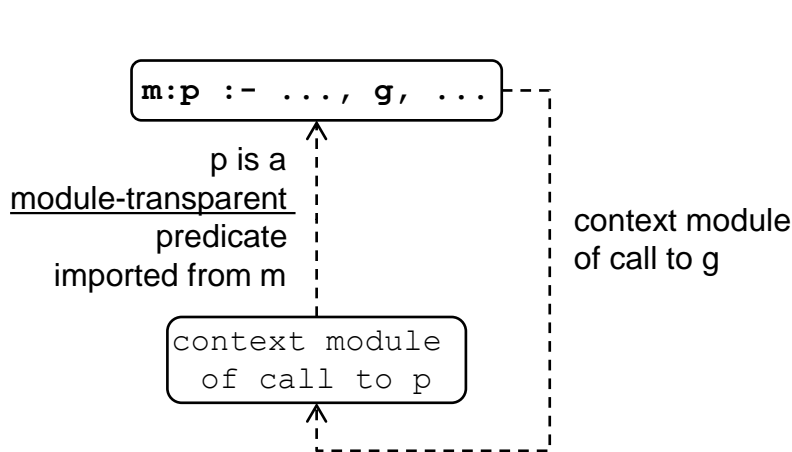
„Context Module“ of a Goal G

- Semantics

- ◆ Module that is used as the starting point for searching clauses for **dynamically bound** (see page 22 ff) predicate calls during execution of G

- Definition

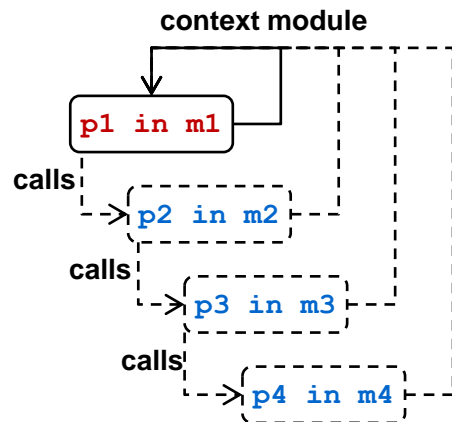
- ◆ The context module of a **module-qualified** goal “**Module:Goal**” is “**Module**”
- ◆ The context module of any **non-qualified** goal in the body of predicate P
 - ⇒ is the **textually containing module** if P is not “**module-transparent**”
 - ⇒ is the **context module of the call to P** if P is “**module-transparent**”



„Context Module“ of a Goal G ▶ Context Propagation

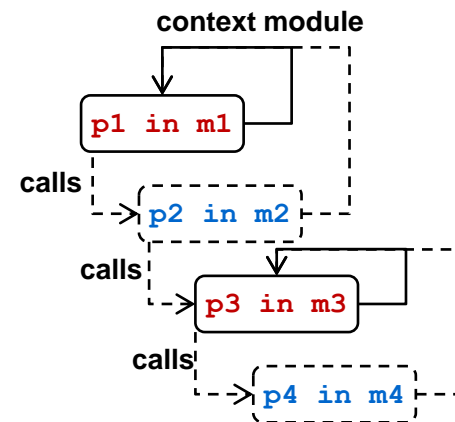
Propagation

- The context module propagates through an uninterrupted sequence of calls of **module-transparent** predicates



No Propagation

- The context module is reset by a **module-qualified call** or a call to a **non-transparent** predicate
- These interrupt the context propagation



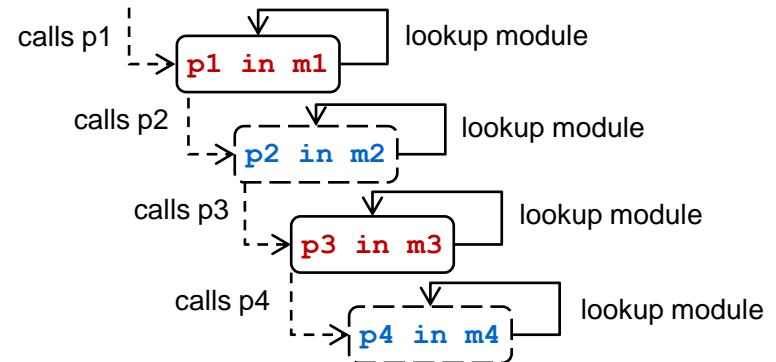
Legend: p2 in m2 module-transparent predicate

p1 in m1 non-transparent predicate

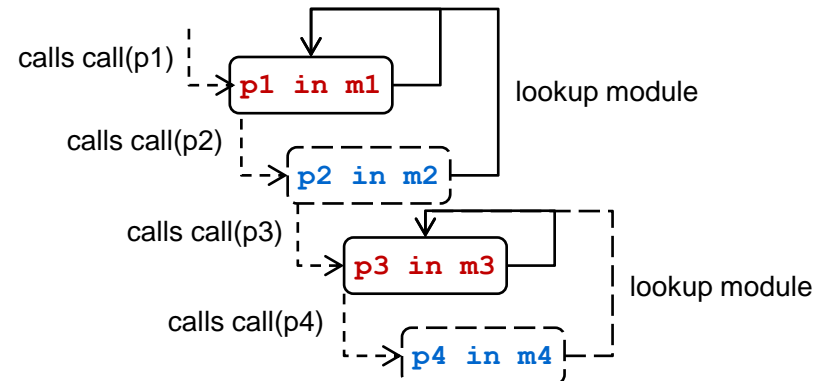
Predicate Lookup and Context Module

- Every direct predicate call
 - ◆ starts searching for predicate definitions in its **textually containing module**
 - ◆ ... hence is **statically bound** to the textually containing module
 - ◆ ... no matter whether the called predicate is module-transparent or not
- Every predicate call via a meta-predicate
 - ◆ starts searching for predicate definitions in the **context module**
 - ◆ ... hence is **dynamically bound** to the context module

Lookup module = textually containing



Lookup module = context module



Legend:

p2 in m2 module-transparent predicate

p1 in m1 non-transparent predicate

Context Module ▶ Example

SHOW

```
:- module(context_demo, [ is__transparent/1 , transp_calls_transp/1
                          , non_transparent/1 , nontransp_calls_transp/1
                          ]
).

:- module_transparent is__transparent/1, transp_calls_transp/1.

non_transparent(M) :- context_module(M).           % 1) textual context
is__transparent(M) :- context_module(M).           % 2) call context

transp_calls_transp(M) :- is__transparent(M).      % 3) transitive call context
nontransp_calls_transp(M) :- is__transparent(M).  % 4) non-trans. call context
```

```
?- non_transparent(Context).           % 1)
Context = context_demo.
```

```
?- is__transparent(Context).           % 2)
Context = user.
```

```
?- nontransp_calls_transp(Context). % 4)
Context = context_demo.
```

```
?- transp_calls_transp(Context). % 3)
Context = user.
```

Static versus Dynamic Binding

Case 1 and 2

- Every non-meta call is bound statically (to the textually containing module)

Case 1 and 3

- Every call in a non-transparent predicate is bound statically (to the textually containing module)

Case 4

- Every meta call in a transparent predicate is bound dynamically (to the context module)

Containing predicate	Kind of call	
	non-meta	meta
non-transparent	1) static binding	3) static binding
transparent	2) static binding	4) dynamic binding

Predicate Lookup ▶ Example

```
:- module(called, [ is__transparent/1 , transp_calls_transp/1
                  , non_transparent/1 , nontransp_calls_transp/1 ] ).

:- module_transparent is__transparent/1, transp_calls_transp/1.

non_transparent(M,B) :- context_module(M) ,      p(B) .
non_transparent(M,B) :- context_module(M) , call(p(B)) .

is__transparent(M,B) :- context_module(M) ,      p(B) .
is__transparent(M,B) :- context_module(M) , call(p(B)) .

transp_calls_transp(M,B) :- is__transparent(M,B) .
nontransp_calls_transp(M,B) :- is__transparent(M,B) .

p(called) .
```

```
:- module(caller, []).
p(caller) .
```

```
?- consult(called) .
?- consult(caller) .

?- caller:non_transparent(Context,Lookup) .
Context = called, Lookup = called ;
Context = called, Lookup = called.
```

Predicate Lookup ▶ Example

```
:- module(called, [ is__transparent/1 , transp_calls_transp/1
                  , non_transparent/1 , nontransp_calls_transp/1 ] ).

:- module_transparent is__transparent/1, transp_calls_transp/1.

non_transparent(M,B) :- context_module(M) ,      p(B) .
non_transparent(M,B) :- context_module(M) , call(p(B)) .

is__transparent(M,B) :- context_module(M) ,      p(B) .
is__transparent(M,B) :- context_module(M) , call(p(B)) .

transp_calls_transp(M,B) :- is__transparent(M,B) .
nontransp_calls_transp(M,B) :- is__transparent(M,B) .

p(called) .
```

```
:- module(caller, []).
p(caller) .
```

```
?- consult(called) .
?- consult(caller) .

?- caller:is__transparent(Context,Lookup) .
Context = caller , Lookup = called ;
Context = caller , Lookup = caller .
```

Predicate Lookup ▶ Example

```
:- module(called, [ is__transparent/1 , transp_calls_transp/1
                  , non_transparent/1 , nontransp_calls_transp/1 ] ).

:- module_transparent is__transparent/1, transp_calls_transp/1.

non_transparent(M,B) :- context_module(M) , p(B) .
non_transparent(M,B) :- context_module(M) , call(p(B)) .

is__transparent(M,B) :- context_module(M) , p(B) .
is__transparent(M,B) :- context_module(M) , call(p(B)) .

transp_calls_transp(M,B) :- is__transparent(M,B) .
nontransp_calls_transp(M,B) :- is__transparent(M,B) .

p(called) .
```

```
:- module(caller, []).
p(caller) .
```

```
?- consult(called) .
?- consult(caller) .

?- caller:transp_calls_transp(Context,Lookup) .
Context = caller , Lookup = called ;
Context = caller , Lookup = caller .
```

Predicate Lookup ▶ Example

```
:- module(called, [ is__transparent/1 , transp_calls_transp/1
                  , non_transparent/1 , nontransp_calls_transp/1 ] ).

:- module_transparent is__transparent/1, transp_calls_transp/1.

non_transparent(M,B) :- context_module(M) , p(B) .
non_transparent(M,B) :- context_module(M) , call(p(B)) .

is__transparent(M,B) :- context_module(M) , p(B) .
is__transparent(M,B) :- context_module(M) , call(p(B)) .

transp_calls_transp(M,B) :- is__transparent(M,B) .
nontransp_calls_transp(M,B) :- is__transparent(M,B) .

p(called) .
```

```
:- module(caller, []).
p(caller) .
```

```
?- consult(called) .
?- consult(caller) .

?- caller:nontransp_calls_transp(Context,Lookup) .
Context = called, Lookup = called ;
Context = called, Lookup = called.
```

Predicate Lookup ▶ Dynamic Binding Idiom

- Making a non-metapredicate call in a transparent predicate be dynamically bound is possible
 - ◆ Use the `context_module/1` predicate to get the current context module
 - ◆ ... then explicitly invoke the call in the context module

Containing predicate	non-meta	meta
non-transparent	static binding	static binding
transparent	dynamic binding	dynamic binding

```
:- module(imported_mod, [ transp/1]).  
:- module_transparent transp/1.  
  
transp(B) :-  
    context_module(M), M:p(B).  
p(imported).
```

```
:- module(importer_mod, []).  
:- use_module(imported_mod).  
p(importer).
```

```
?- consult(importer_mod).  
  
?- importer_mod:transp(Lookup).  
Lookup = importer.
```

Remember

- Module-sensitive metapredicates are documented by prepending “:” to the module-sensitive meta-arguments
 - ◆ `call(:Goal)`,
 - ◆ `assert(:Clause)`
 - ◆ `retract(:Head)`
 - ◆ `findall(Template, :Goal, List)`
 - ◆ `catch(:Goal,Exception,:Catcher)`
 - ◆ ...
- Any predicate, that calls a module-sensitive metapredicate must be declared `module_transparent`.
 - ◆ Otherwise, it won't work properly, trying to resolve meta-arguments in its own containing module instead of the invoking module.
 - ◆ This is awkward...

Defining own Meta-Predicates

meta_predicate +Head 1, +Head 2, ...

- Define the predicate referenced by Head_i as a meta-predicate
- Each argument of Head_i is a **meta argument specifier**
 - ◆ - / + / ?
 - ⇒ The argument is free / bound / unspecified at call time (and not module sensitive)
 - ◆ :
 - ⇒ The argument is module sensitive, but does not directly refer to a predicate.
 - For example: consult(:).
 - ◆ 0..9
 - ⇒ The argument term will reference a predicate with N more arguments than itself
 - For example: call(0) or maplist(1, +).
- Each argument marked “:” or “0” .. “9” is module-sensitive
 - ◆ It is qualified with the context module of the caller <module>:<term>, where <module> is a module name and <term> is not a :/2 term.

Home exercise

```
:- module( m2, [ ] ).
```

```
test(1).
```

```
test(2).
```

```
:- module( myMetaLibrary, [reportMetaArgument/1] ).
```

```
:- meta_predicate reportMetaArgument(0).
```

```
reportMetaArgument(Module:Term) :-  
    format('Module=~w, Term = ~q~n', [Module,Term]),  
    call(Module:Term).
```

```
?- reportMetaArgument(test).
```

```
Module = user, Term = test
```

```
?- reportMetaArgument(m1:test).
```

```
Module = , Term = test
```

```
?- m2:reportMetaArgument(test).
```

```
Module = , Term = test
```

```
?- m1:reportMetaArgument(m2:test).
```

```
Module = , Term = test
```

```
?- reportMetaArgument(m1:m2:test(X)).
```

```
Module = , Term = test(...)
```


Using Modules like Objects

Class instances versus Prototypes

Modules versus Prototypes

Inheritance and dynamic binding

Object-oriented Programming? ▶ Which one?

Comparing Prolog to objects without getting confused requires in the first place to know both families of object-oriented languages:

- **Class-based** languages ▶ Oranges
 - ◆ Java, C++, Smalltalk, Simula, Eiffel, ...
- **Prototype-based** languages ▶ Apples
 - ◆ Self, Newton Script, Java Script, ...

Schedule for this section

- 1) Class-based versus prototype-based objects
- 2) Prototype-based language concepts in Prolog
- 3) Object-oriented design for Prolog
- 4) Using UML for Prolog designs

Class Instances versus Prototypes

Class instances

- Contain fields and methods
- Are encapsulated
- Interact by dynamically bound invocations
- Are **defined by classes**
- Structure and behaviour are **fixed**
- Are **instantiated** from class
- **Share structure and behaviour** defined by their class
- **Cannot delegate** to other objects (inheritance only among classes)

Prototypes

- Contain fields and methods
- Are encapsulated
- Interact by dynamically bound invocations
- Are **self-contained**
- Structure and behaviour **can change** at any time
- Are **gradually modified** or cloned or both
- Are **unique**
- **Delegate** to other objects (= object-based inheritance)

→ Read about delegation: <http://roots.iai.uni-bonn.de/research/darwin/delegation>

SWI-Prolog Modules

- Contain dyn. and stat. predicates
- Are **not encapsulated**
- Interact by **statically or dynamically** bound invocations
- Are self-contained
- Structure and behaviour can change at any time (assert, ...)
- Are gradually modified or cloned or both
- Are unique
- **Import from** other modules
 - ◆ Local definitions **hide / override** those from other modules

Prototypes

- Contain fields and methods
- **Are encapsulated**
- **Interact by dynamically bound invocations**
- Are self-contained
- Structure and behaviour can change at any time
- Are gradually modified or cloned or both
- Are unique
- **Delegate** to other objects
 - ◆ Local definitions **override** those from other objects

SWI-Prolog Modules versus Prototypes ▶ The Similarities → Adapted UML

SWI-Prolog Modules

Module	←---- Name
dynPred(Arg1, ...)	←---- Data
statPred(Arg1, ...)	←---- Operations

Prototypes

Prototype	←---- Name
field	←---- Data
meth(Arg1, ...)	←---- Operations

- Are self-contained
- Structure and behaviour can change at any time (assert, ...)
- Are gradually modified or cloned or both
- Are unique
- Contain dyn. and stat. predicates

- Are self-contained
- Structure and behaviour can change at any time
- Are gradually modified or cloned or both
- Are unique
- Contain fields and methods

SWI-Prolog Modules versus Prototypes ▶

The Differences – No Encapsulation

SWI-Prolog Modules

Module	←---- Name
statPred(Arg1, ...)	←---- Data
dynPred(Arg1, ...)	←---- Operations

Prototypes

Prototype	←---- Name
field	←---- Data
meth(Arg1, ...)	←---- Operations

- Are **not** encapsulated
- Are encapsulated

Modules are **not** a protection mechanism!

- The notions “public”, “protected” and “private” make no sense.
- The export list is **not** the list of public predicates
- It is just the list of predicates that the module *recommends* others to import.
- Predicates in the export list can be excluded from import and predicates that are not in the list can be imported. The importing module decides!

Nevertheless, we are able to express the recommendation. It is still up to the clients to comply with it, but to comply they must at least know it!

Summary

- Modules are a namespace mechanism, not for protection
 - ◆ They avoid name clashes
 - ◆ They control visibility, not accessibility

- Modules are like non-encapsulated prototypes
 - ◆ Module name = object identity
 - ◆ Predicates = methods and fields

More on Modules

- Modules have very different semantics in different Prolog implementations
 - ◆ See the online documentation of Quintus, YAP, XSB, LPA, ...
- The SWI module system is one of the most advanced ones
 - ◆ Separate compilation is supported also for meta-predicates
 - ◆ “Module-Transparent” is supported only in SWI- and Eclipse-Prolog
- Fear of non-portability should not stop you from using modules
 - ◆ An unmanageable, badly structured program is unusable even on a single platform