Scope

COMP 524: Programming Language Concepts
Björn B. Brandenburg

The University of North Carolina at Chapel Hill

Based in part on slides and notes by S. Olivier, A. Block, N. Fisher, F. Hernandez-Campos, and D. Stotts.

Referencing Environment

"All currently known names."

The set of active bindings.

➡At any given point in time during execution.

➡Can change: names **become valid** and **invalid** during execution in most programming languages.

➡Exception: early versions of **Basic** had only a single, global, fixed namespace.

How is the referencing environment defined?

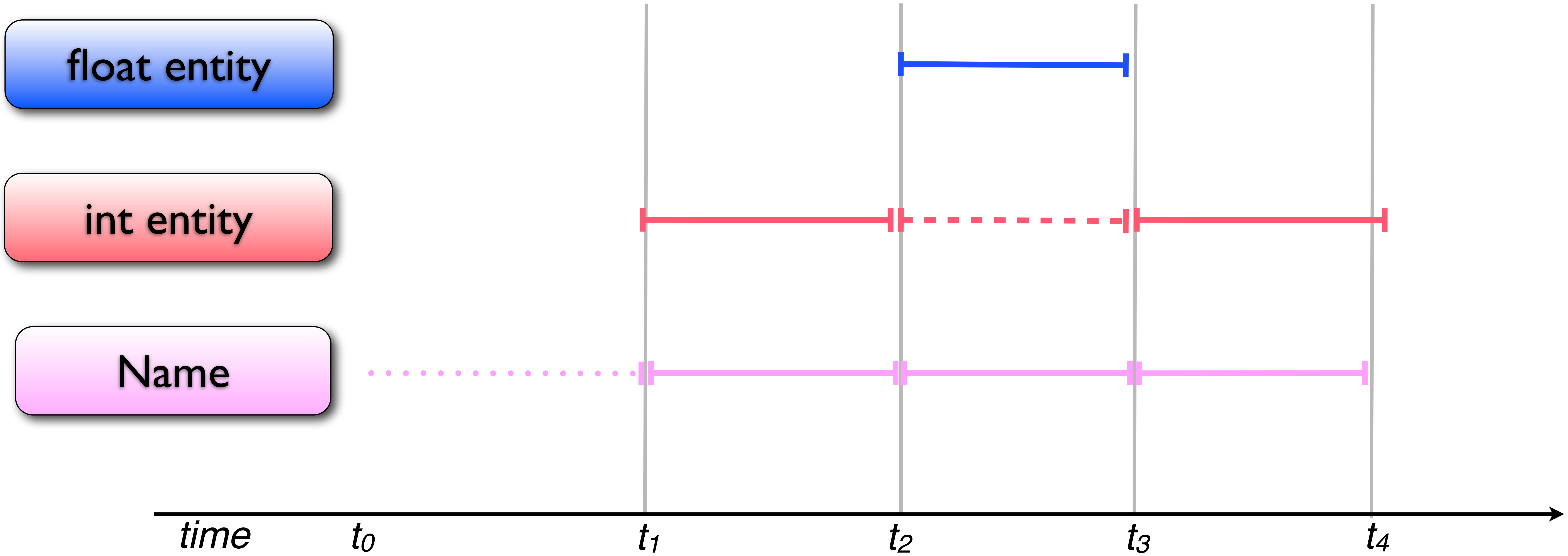
➡**Scope rules**.

➡The scope of a binding is its "lifetime."

➡I.e., the **textual region** of the program in which a binding is active.

Scope of a Binding

The (textual) region in which a binding is active.

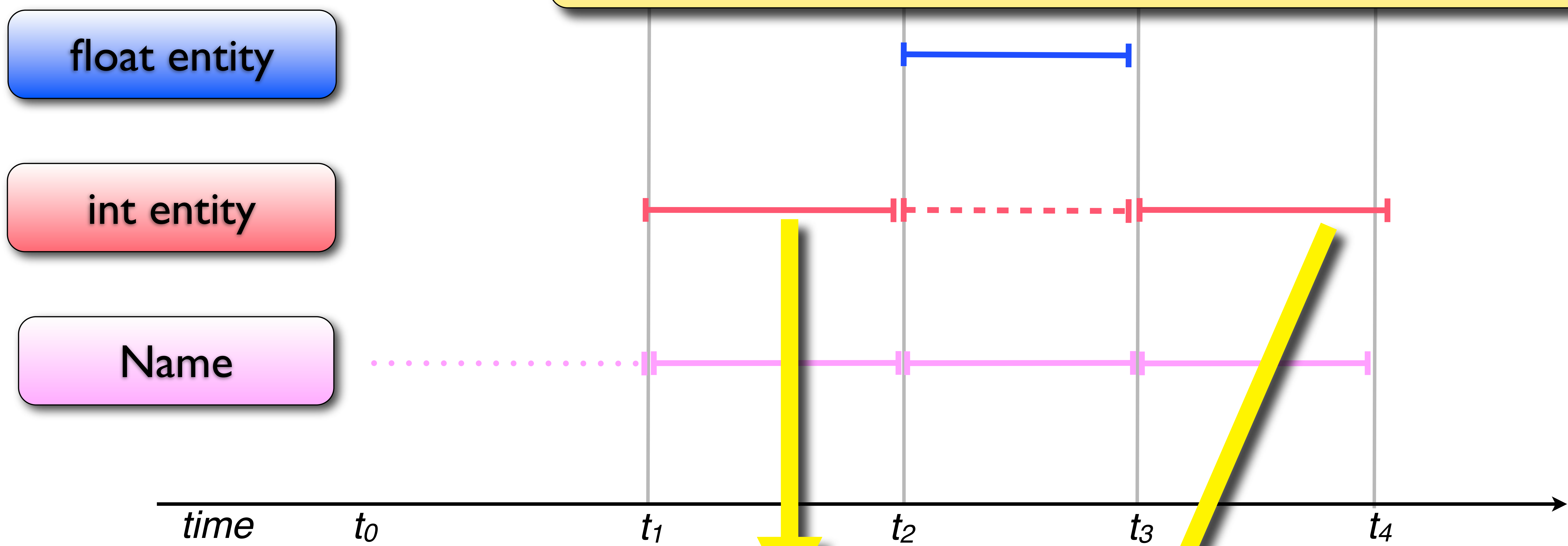



```
void method() {
  int name;
  // code executed in [t1-t2).
  {
    float name;
    // code executed in [t2-t3).
  }
  // code executed in [t3-t4).
}
```

Scope of a Binding

The (textual)

Scope of **name**-to-**int-entity** binding.

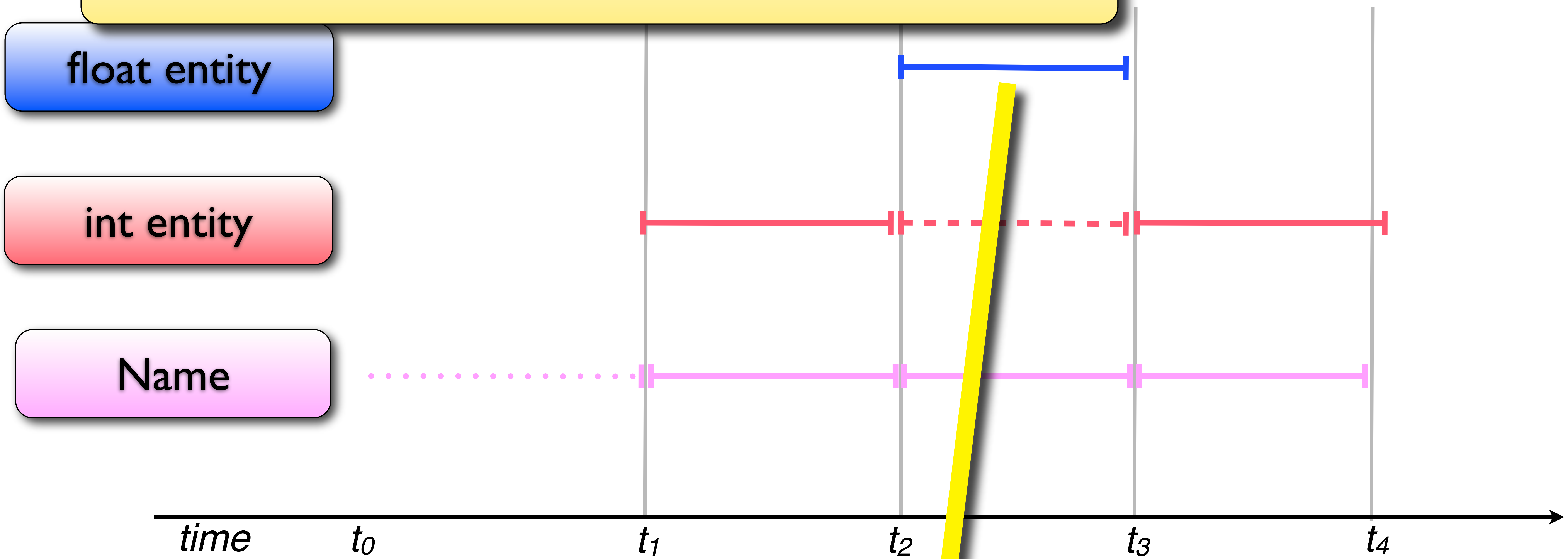



```
void method() {
  int name;
  // code executed in [t1-t2).
  {
    float name;
    // code executed in [t2-t3
  }
  // code executed in [t3-t4).
}
```

Scope of a Binding

Scope of name-to-float-entity binding.

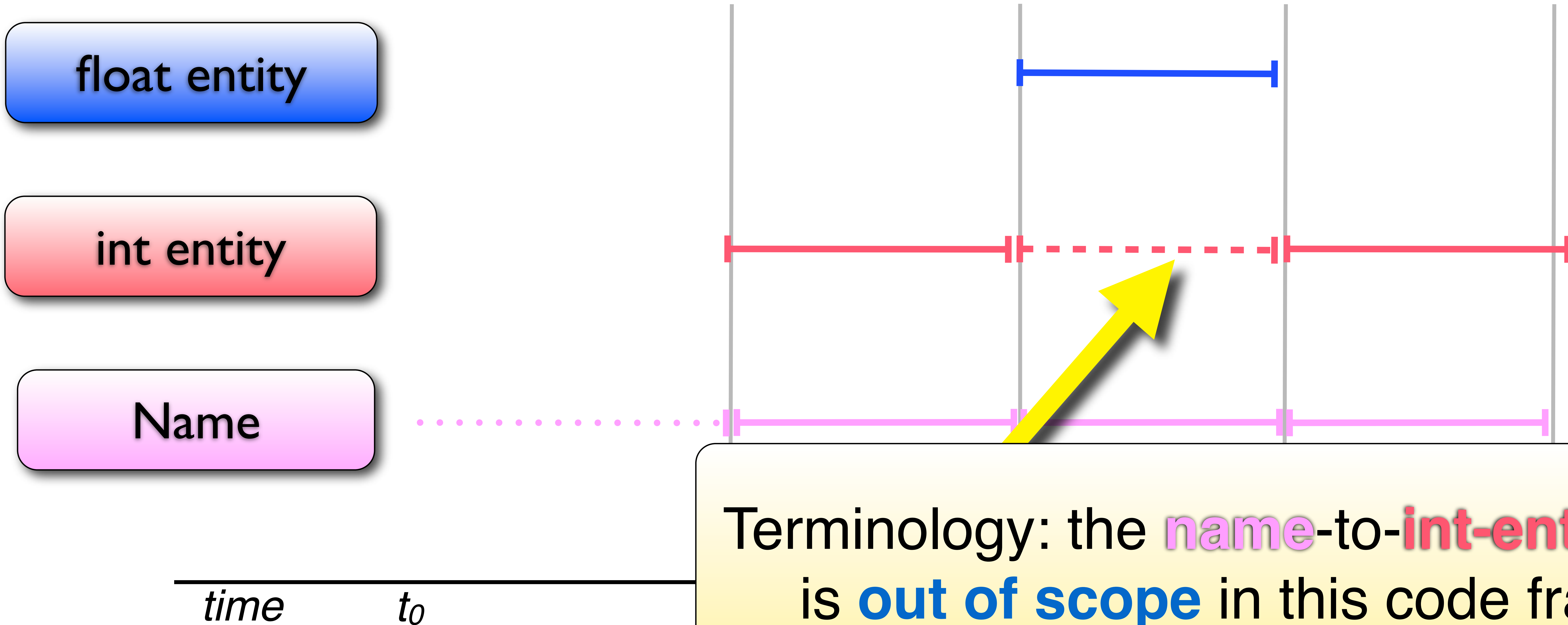
... is active.




```
void method() {
  int name;
  // code executed in [t1-t2).
  {
    float name;
    // code executed in [t2-t3).
  }
  // code executed in [t3-t4).
}
```

Scope of a Binding

The (textual) region in which a binding is active.



Terminology: the **name**-to-**int-entity** binding is **out of scope** in this code fragment. The scope is said to have a "**hole**."

```
void metho
  int name;
  // code executed in [  -t₂).
  {
   float name;
   // code executed in [t₂-t₃).
  }
  // code executed in [t₃-t₄).
}
```


Thursday, March 4, 2010

Language Scope Rules

a major language design choice

```
void printX() {
   printf("x = " + x);
}
```

what does x refer to?

Dynamically Scoped.

- ➡ Active bindings **depend on control flow**.
- ➡ Bindings are discovered during execution.
- ➡ E.g., meaning of a name depends on call stack.

Statically Scoped.

- ➡ All bindings **determined a compile time**.
- ➡ Bindings do not depend on call history.
- ➡ Also called **lexically scoped**.

Dynamically vs. Statically Scoped

Which bindings are active in subroutine body?

Dynamically Scoped:

Subroutine body is executed in the referencing environment of the **subroutine caller**.

Statically Scoped:

Subroutine body is executed in the referencing environment of the **subroutine definition**.

Dynamic Scope Example

```perl
# This is dynamically scoped Perl.
$x = 10;

sub printX {
  # $x is dynamically scoped.
  $from = $_[0];
  print "from $from: x = $x \n";
}

sub test0 {
  local $x; # binding of $x is shadowed.
  $x = 0;
  printX "test0"
}

sub test1 {
  local $x; # binding $x is shadowed.
  $x = 1;
  test0;
  printX "test1"
}

test1;
printX "main";
```

Dynamic Scope Example

```perl
# This is dynamically scoped Perl.
$x = 10;

sub printX {
  # $x is dynamically scoped.
  $from = $_[0];
  print "from $from: x = $x \n";
}

sub test0 {
  local $x; # binding of $x is shadowed.
  $x = 0;
  printX "test0"
}

sub test1 {
  local $x; # binding $x is shadowed.
  $x = 1;
  test0;
  printX "test1"
}

test1;
printX "main";
```

New binding created. Existing variable is **not overwritten**, rather, the existing binding (if any) is **shadowed**.

Dynamic Scope Example

```perl
# This is dynamically scoped Perl.
$x = 10;

sub printX {
  # $x is dynamically scoped.
  $from = $_[0];
  print "from $from: x = $x \n";
}

sub test0 {
  local $x; # binding of $x is shadowed.
  $x = 0;
  printX "test0"
}

sub test1 {
  local $x; # binding $x is shadowed.
  $x = 1;
  test0;
  printX "test1"
}

test1;
printX "main";
```

Dynamically scoped:
the **current binding** of `$x` is
the one encountered
most recently during execution
(that has not yet been destroyed).

Dynamic Scope Example

```perl
# This is dynamically scoped Perl.
$x = 10;

sub printX {
  # $x is dynamically scoped.
  $from = $_[0];
  print "from $from: x = $x \n";
}

sub test0 {
  local $x; # binding of $x is shadowed.
  $x = 0;
  printX "test0"
}

sub test1 {
  local $x; # binding $x is shadowed.
  $x = 1;
  test0;
  printX "test1"
}

test1;
printX "main";
```

Output:

```
from test0: x = 0
from test1: x = 1
from main: x = 10
```

Dynamic Scope Example

```
# This is dynamically scoped Perl.
$x = 10;

sub printX {
  # $x is dynamically scoped.
  $from = $_[0];
  print "from $from: x = $x \n";
}

sub test0 {
  local $x; # binding of $x is shadowed.
  $x = 0;
  printX "test0"
}

sub test1 {
  local $x; # binding $x is shadowed.
  $x = 1;
  test0;
  printX "test1"
}

test1;
printX "main";
```

Output:

```
from test0: x = 0
from test1: x = 1
from main: x = 10
```

Dynamic Scope Example

```
# This is dynamically scoped Perl.
$x = 10;

sub printX {
  # $x is dynamically scoped.
  $from = $_[0];
  print "from $from: x = $x \n";
}

sub test0 {
  local $x; # binding of $x is shadowed.
  $x = 0;
  printX "test0"
}

sub test1 {
  local $x; # binding $x is shadowed.
  $x = 1;
  test0;
  printX "test1"
}

test1;
printX "main";
```

Output:

```
from test0: x = 0
from test1: x = 1
from main: x = 10
```

Dynamic Scope Example

```
# This is dynamically scoped Perl.
$x = 10;

sub printX {
  # $x is dynamically scoped.
  $from = $_[0];
  print "from $from: x = $x \n";
}

sub test0 {
  local $x; # binding of $x is shadowed.
  $x = 0;
  printX "test0"
}

sub test1 {
  local $x; # binding $x is shadowed.
  $x = 1;
  test0;
  printX "test1"
}

test1;
printX "main";
```

Output:

```
from test0: x = 0
from test1: x = 1
from main: x = 10
```

Dynamic Scope

Origin.

- Most early **Lisp** versions were dynamically scoped.
- **Scheme is lexically scoped** and became highly influential; nowadays, dynamic scoping has fallen out of favor.

Possible use.

- **Customization** of "service routines." E.g., field width in output.
- As **output parameters** for methods (write to variables of caller).

Limitations.

- Hard to **reason about program**: names could be bound to "anything."
- **Accidentally overwrite** unrelated common variables (i, j, k, etc.).
- Scope management occurs at runtime; this **creates overheads** and thus limits implementation efficiency.

Static Scope Example

```
public class Scope {
  static int x = 10;

  static void printX(String from) {
    System.out.println("from " + from +
                       ": x = " + x);
  }

  static void test0() {
    int x = 0;
    printX("test0");
  }

  static void test1() {
    int x = 1;
    test0();
    printX("test1");
  }

  public static void main(String... args) {
    test1();
    printX("main");
  }
}
```

Static Scope Example

```java
public class Scope {
  static int x = 10;

  static void printX(String from) {
    System.out.println("from " + from +
                       ": x = " + x);
  }

  static void test0() {
    int x = 0;
    printX("test0");
  }

  static void test1() {
    int x = 1;
    test0();
    printX("test1");
  }

  public static void main(String... args) {
    test1();
    printX("main");
  }
}
```

New binding created. Existing variable is **not overwritten**, rather, the existing binding (if any) is **shadowed**.

Static Scope Example

```
public class Scope {
  static int x = 10;

  static void printX(String from) {
    System.out.println("from " + from +
                       ": x = " + x);
  }

  static void test0() {
    int x = 0;
    printX("test0");
  }

  static void test1() {
    int x = 1;
    test0();
    printX("test1");
  }

  public static void main(String... args) {
    test1();
    printX("main");
  }
}
```

Output:

```
from test0: x = 10
from test1: x = 10
from main: x = 10
```

Static Scope Example

```
public class Scope {
  static int x = 10;

  static void printX(String from) {
    System.out.println("from " + from +
                       ": x = " + x);
  }

  static void test0() {
    int x = 0;
    printX("test0");
  }

  static void test1() {
    int x = 1;
    test0();
    printX("test1");
  }

  public static void main(String... args) {
    test1();
    printX("main");
  }
}
```

Output:

```
from test0: x = 10
from test1: x = 10
from main: x = 10
```

Lexically scoped:
the **binding** of `x` is determined at compile time and based on the **enclosing scope of the method definition**.

Static Scope Example

```
public class Scope {
  static int x = 10;

  static void printX(String from) {
    System.out.println("from " + from +
                       ": x = " + x);
  }

  static void test0() {
    int x = 0;
    printX("test0");
  }

  static void test1() {
    int x = 1;
    test0();
    printX("test1");
  }

  public static void main(String... args) {
    test1();
    printX("main");
  }
}
```

shadowed

shadowed

Output:

```
from test0: x = 10
from test1: x = 10
from main: x = 10
```

Scope of the outermost binding of x.

Static/Lexical Scope

Variants.

➡**Single**, global scope: Early Basic.

➡Just **two**, global + local: Early Fortran.

➡**Nested** scopes: modern languages.

Advantages.

➡Names can be fully resolved at **compile time**.

➡Allows generation of **efficient code**; code generator can compute offsets.

➡Easier to reason about; there is **only one applicable enclosing referencing environment**.

Nested Scopes

If there are multiple bindings for a name to choose from, which one should be chosen?

```cpp
// this is C++
#include <iostream>
using namespace std;

int aName = 10;

class AClass {
private:
   int aName;

public:
   AClass();
   void aMethod();
   void bMethod();
};

AClass::AClass() {
   aName = 1;
}
```

```cpp
// continued…

void AClass::aMethod() {
   int aName = 2;
   cout << "a: " << aName << " "
        << ::aName << endl;
}

void AClass::bMethod() {
   cout << "b: " << aName << " "
        << ::aName << endl;
}

int main() {
  AClass obj;
  obj.aMethod();
  obj.bMethod();
  return 0;
}
```

Nested Scopes

If there are multiple bindings for a name to choose from, which one should be chosen?

```
// this is C++
#include <iostream>
using namespace std;

int aName = 10;

class AClass {
private:
   int aName;

public:
   AClass();
   void aMethod();
   void bMethod();
};

AClass::AClass() {
   aName = 1;
}
```

```
// continued…

void AClass::aMethod() {
   int aName = 2;
   cout << "a: " << aName << " "
        << ::aName << endl;
}

void AClass::bMethod() {
   cout << "b: " << aName << " "
        << ::aName << endl;
}

int main() {
  AClass obj;
  obj.aMethod();
  obj.bMethod();
  return 0;
}
```

```
Output:
a: 2 10
b: 1 10
```

Closest nested scope rule:

a binding is active in the scope in which it is declared and **in each nested scope**, unless it is shadowed by another binding (of the same name). This is the standard in **Algol descendants**.

```
#include <iostream>
using namespace std;

int aName = 10;

class AClass {
private:
   int aName;

public:
   AClass();
   void aMethod();
   void bMethod();
};

AClass::AClass() {
   aName = 1;
}
```

```
void AClass::aMethod() {
   int aName = 2;
   cout << "a: " << aName << " "
        << ::aName << endl;
}

void AClass::bMethod() {
   cout << "b: " << aName << " "
        << ::aName << endl;
}

int main() {
  AClass obj;
  obj.aMethod();
  obj.bMethod();
  return 0;
}
```

```
Output:
a: 2 10
b: 1 10
```

C++: Scope Resolution Operator ::

Some languages, such as C++, allow the closest-nested-scope rule to be overridden by explicitly referring to shadowed entities by “their full name.”

```
// this
#include <iostream>
using namespace std;

int aName = 10;

class AClass {
private:
   int aName;

public:
   AClass();
   void aMethod();
   void bMethod();
};

AClass::AClass() {
   aName = 1;
}
```

```
void AClass::aMethod() {
   int aName = 2;
   cout << "a: " << aName << " "
        << ::aName << endl;
}

void AClass::bMethod() {
   cout << "b: " << aName << " "
        << ::aName << endl;
}

int main() {
  AClass obj;
  obj.aMethod();
  obj.bMethod();
  return 0;
}
```

```
Output:
a: 2 10
b: 1 10
```

Implementing Scope

Symbol table.

- Map *Name* → (*Entity: Address, data type, extra info*)
- Keeps track of currently known names.
- One of two **central data structures** in compilers. (the other is the abstract syntax tree).

Implementation.

- Any **map-like abstract data type**. E.g.:
  - Association list.
  - Hash map.
  - Tree map.
- But how to keep track of scopes?
  - Constantly entering and removing table entries is **difficult and slow**.

Entering & Exiting a Scope

Idea: one table per scope/block.

- Called the "environment."

Referencing environment = stack of environments.

- **Push** a new environment onto the stack when **entering a nested scope**
- **Pop** environment off stack when **leaving a nested scope**.
- Enter **new declarations** into **top-most** environment.

Implementation.

- Can be implemented easily with a "enclosing scope" pointer.
- This is called the **static chain pointer**.
- The resulting data structure (a list-based stack of maps) is called the **static chain**.
- ***O(n)* lookup** time (n = nesting level).
  - Optimizations and alternate approaches exist, esp. for interpreters.

Entering & Exiting a Scope

Idea: one table per scope/block.

➡ Called the "environment."

Implementing the Closest Nested Scope Rule

```
To lookup a name aName:
    curEnv = top-most environment
    while curEnv does not contain aName:
        curEnv = curEnv.enclosingEnvirontment
        if curEnv == null:
            // reached top of stack
            throw new SymbolNotFoundException(aName)
    return curEnv.lookup(aName)
```

R

In

e

static chain.

➡ ***O(n)* lookup** time (n = nesting level).

‣ Optimizations and alternate approaches exist, esp. for interpreters.

Scoping & Binding Issues

Scoping & Binding: Name resolution.

- **Simple concepts**…
- …but surprisingly many design and implementation **difficulties arise**.

A few examples.

- Shadowing and type conflicts.
- Declaration order: where exactly does a scope begin?
- Aliasing.
  - An object by any other name…

```
int foo;
…
while (…) {
  float foo; // ok?
}
```

Declaration Order

Scope vs. Blocks.

➡ Many languages (esp. Algol descendants) are block-structured.

```
if {
  // a block
  while (…) {
    // a nested block
  }
}
```

```
BEGIN
  // a block
  REPEAT
    BEGIN
      // a nested block
    END
  UNTIL …
END
```

What is the scope of a declaration?

➡ Usually, the **scope of a declaration ends with the block** in which it was declared.

➡ But **where does it begin**?

➡ Does **declaration order** matter?

Declaration Order

Example: Algol 60

Declarations **must** appear at **beginning of block** and are valid from the point on where they are declared.

Thus, scope and block are ***almost*** the same thing.

But how do you declare a **recursive structure** like a linked list?

```
}
```

```
END
```

What is the scope of a declaration?

- ➡ Usually, the **scope of a declaration ends with the block** in which it was declared.
- ➡ But **where does it begin**?
- ➡ Does **declaration order** matter?

Declaration Order

Scop
➡ Ma

Example: Pascal
Names must be declared **before they are used**, but the scope is the **entire** surrounding block.

```
if
   // a block
   while (…) {
      // a nested block
   }
}
```

```
   REPEAT
      BEGIN
         // a nested block
      END
   UNTIL …
END
```

What is the scope of a declaration?
- ➡ Usually, the **scope of a declaration ends with the block** in which it was declared.
- ➡ But **where does it begin**?
- ➡ Does **declaration order** matter?

Declaration Order

Scop
➡ Ma

```
if
}
```

What
➡ Us ... h it
wa
➡ But
➡ Do

Example: Pascal
Names must be declared **before they are used**, but the scope is the **entire** surrounding block.

Surprising interaction…

```
const N = 10;
…
procedure foo; { procedure is new block }
const
    M = N;  { error; N used before decl. }
    …
    N = 20; { ok; outer N shadowed  }
…
```

Variable /Attribute Scope in Java

```
static int foo;

public static void test() {
    float foo;

    if (true) {
        char foo;
        int bar;
    }

    bar = 1;
}
```

Variable /Attribute Scope in Java

```java
static int foo;

public static void test() {
    float foo;

    if (true) {
        char foo;
        int bar;
    }

    bar = 1;
}
```

Error:

`bar` cannot be resolved

(Scope of `bar` ends with block.)

Scope in Java

Error:
Duplicate local variable foo
(local **`foo`**'s scope **not** shadowed!)

```
static int foo;

public static void test() {
    float foo;

    if (true) {
        char foo;
        int bar;
    }

    bar = 1;
}
```

Variable /Attribute Scope in Java

```java
static int foo;

public static void test() {
    float foo;

    if (true) {
        char foo;
        int bar;
    }

    bar = 1;
}
```

Ok:
local `foo` shadows attribute

Declaration Order in Java

```
static int foo = 3;

public static void test1() {
    float foo = bar;
    float bar = 2;
}

public static void test2() {
    float bar = foo;
    float foo = bar;
}
```

Declaration Order in Java

Error:
`bar` cannot be resolved
(Must be declared before use, like Pascal.)

```
static int foo = 3;

public static void test1() {
    float foo = bar;
    float bar = 2;
}

public static void test2() {
    float bar = foo;
    float foo = bar;
}
```

Declaration Order in Java

Ok: attribute `foo` not yet shadowed
(both `bar` and local `foo` initialized to 3.0; differs from Pascal)

```
static int foo = 3;

public static void test1() {
    float foo = bar;
    float bar = 2;
}

public static void test2() {
    float bar = foo;
    float foo = bar;
}
```

Declaration vs. Definition

```
void function1(void)
{
    function2();
}

void function2(void)
{
    function1();
}
```

C/C++: Name only valid after declaration.

- ➡ How to define a list type (**recursive type**)?
  - ‣ Next pointer is of the type that is being defined!
- ➡ How to implement **mutually-recursive functions**?
  - ‣ E.g., recursive-descent parser.

Implicit declaration.

- ➡ Compiler "**guesses**" signature of **unknown function**.
- ➡ **signature**: return value and arguments.
- ➡ Guesses wrong; this causes an error when actual declaration is encountered.

```
warning: conflicting types for 'function2'
warning: previous implicit declaration of 'function2'
```

Declaration vs. Definition

Solution: split declaration from definition.

C/C++: can declare name without defining it.

- ➡ Called a "**forward declaration**."
- ➡ A **promise**: "I'll shortly tell you what it means."

Declare before use; define later.

- ➡ **Recursive structures possible**.
- ➡ Also used to support **separate compilation** in C/C++.
  - ‣ Declaration in **header file**.
  - ‣ Definition not available until **linking**.

```
void function2();

void function1(void)
{
    function2();
}

void function2(void)
{
    function1();
}
```

Compiles without errors.

Declaration vs. Definition

Solution: split declaration from definition.

C/C++: can declare name without defining it.

➡ Called a "**forward declaration**."

➡ A **promise**: "I'll shortly tell you what it means."

Declare before use; define later.

➡ **Recursive structures possible**.

➡ Also used to support **separate compilation** in C/C++.

- Declaration in **header file**.
- Definition not available until **linking**.
- **If not defined:** linker reports "symbol not found" error.

```
Undefined symbols:
  "_main", referenced from:
      start in crt1.10.6.o
ld: symbol(s) not found
collect2: ld returned 1 exit status
```

Forward declaration without definition.

```
void function2();

void function1(void)
{
    function2();
}

void function2(void)
{
    function1();
}
```

Compiles without errors.

Aliasing

Objects with multiple names.

- Aliasing: seemingly independent variables refer to **same** object.
- Makes understanding programs more difficult (reduced readability).

Hinders optimization.

- In general, **compiler cannot decide** whether an object can become aliased in languages with **unrestricted pointers/references**.
- To avoid corner cases: **possible optimizations disabled**.

```
double sum, sum_of_squares;
void acc(double &x){
 sum += x;
 sum_of_squares += x * x;
}
acc(sum);
```

Aliasing

Objects with multiple names.

- Aliasing: seemingly independent variables refer to **same** object.
- Makes understanding programs more difficult (reduced readab

Hinders optimiza

- In general, **com** aliased in languages with **unrestricted pointers/references**.
- To avoid corner cases: **possible optimizations disabled**.

C++: x is passed by reference
(Function doesn't get a copy of the value, but the actual address of x).

```
double sum, sum_of_squares;
void acc(double &x){
 sum += x;
 sum_of_squares += x * x;
}
acc(sum);
```

Aliasing

Objects with multiple names.

- ➡ Aliasing: seemingly independent variables refer to **same** object.
- ➡ Makes understanding programs more difficult (reduced readability).

In this case, **x** and **sum** refer to the **same object**!

ecome aliased in languages with **unrestricted pointers/references**.

- ➡ To avoid corner cases: **possible optimizations disabled**.

```
double sum, sum_of_squares;
void acc(double &x){
 sum += x;
 sum_of_squares += x * x;
}
acc(sum);
```

Aliasing

Objects with multiple names.

- ➡ Aliasing: seemingly independent variables refer to **same** object.
- ➡ Makes understanding programs more difficult

Thus, **the value of x changes** between the two additions: not a proper “sum of squares.”

hether an object can become

pointers/references.

➡ To avoid corner cases: **possible optimizations disabled**.

```
double sum, sum_of_squares;
void acc(double &x){
 sum += x;
 sum_squares += x * x;
}
acc(sum);
```

Aliasing

Desirable optimization:
keep the value of x in a register between additions.
However, with aliasing, this is not a correct optimization:
semantics of program would be altered in corner case!

Hinders optimization.

- ➡ In general, **compiler cannot decide** whether an object can become aliased in languages with **unrestricted pointers/references**.
- ➡ To avoid corner cases: **possible optimizations disabled**.

```
double sum, sum_of_squares;
void acc(double &x){
 sum += x;
 sum_squares += x * x;
}
acc(sum);
```

Aliasing

Objects with multiple names.

- ➡ Aliasing: seemingly independent variables refer to **same** object.
- ➡ Makes understanding programs more difficult (reduced readability).

Hinders optimization.

- ➡ In general, **compiler cannot decide** whether an object can become aliased in languages with **unrestricted pointers/references**.
- ➡ To avoid corner cases: **possible optimizations disabled**.

When runtime efficiency is favored over language safety:
Some languages disallow or restrict aliasing,
e.g., Fortran (aliasing illegal) and C99 (type restrictions).

Bottom Line

- Languages designed for **efficient compilation** are usually **statically scoped**.
- Rules for **scopes**, **nested scopes**, and **shadowing** are crucial elements of language design.
- Seemingly simple rules can give rise difficult corner cases and inconsistent behavior.

Carefully read your language's ***specification****!*

The Need for Modules / Namespaces

Unstructured names.

- So far we have only considered “**flat**” **namespaces**.
  - Typical for language design before the mid ‘70ies.
- Sometimes **multiple** “flat” namespaces:
  - E.g., one each for subroutines, types, variables and constants.
  - No shadowing between **variable** `start` and a **subroutine** `start` in this case.

Too much complexity.

- Referencing environment often contains thousands of names.
  - OS APIs, libraries, the actual program, etc.
- Significant “**cognitive load**,” i.e., too many names confuse programmers.

The Need for Modules / Namespaces

Possibly including names for internal "helpers."
Programmer should not have to worry about these.

Thus, we'd like some way to **encapsulate** unnecessary details and expose only a **narrow interface**.

ts.

Too much complexity.

- Referencing environment often contains thousands of names.
  - OS APIs, libraries, the actual program, etc.
- Significant "**cognitive load**," i.e., too many names confuse programmers.

Name Clash Example in C

```
#include <fcntl.h> /* POSIX API for IO */

…

db_connection_t* open(db_settings_t *settings)
{
    /* …open a new data base connection… */
}
```

```
error: conflicting types for ‘open’
/usr/include/sys/fcntl.h:427:
   error: previous declaration of ‘open’ was here
```

Name already taken by POSIX API!
(as are thousands of other names)

```
#include <fcntl.h> /* POSIX API for IO */

…

db_connection_t* open(db_settings_t *settings)
{
    /* …open a new data base connection… */
}
```

```
error: conflicting types for 'open'
/usr/include/sys/fcntl.h:427:
   error: previous declaration of 'open' was here
```

Name already taken by POSIX API!
(as are thousands of other names)

```c
#include <fcntl.h> /* POSIX API for IO */

…

db_connection_t* open(db_settings_t *settings)
{
    /* …open a new data base connection… */
}
```

```
error: conflicting types for 'open'
/usr/include/sys/fcntl.h:427:
   error: previous declaration of 'open' was here
```

Common kludge: prefix all names with library name
E.g., use **`db_open`** instead of just **`open`**.

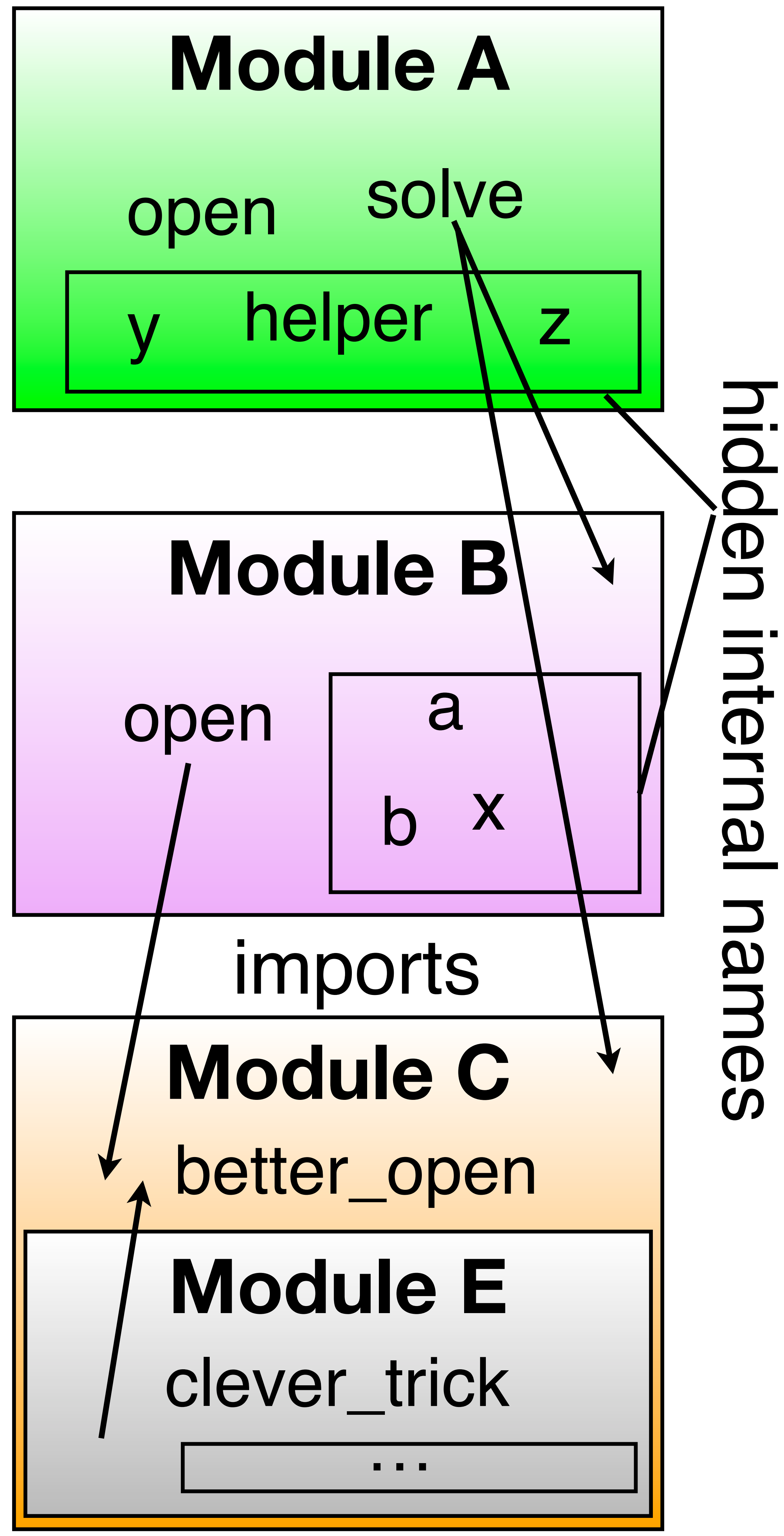
Module / Namespace / Package

*A means to **structure names** and enable **information hiding**.*

Collection of named objects and concepts.
➡ Subroutines, variables, constants, types, etc.

Encapsulation: constrained visibility.
➡ Objects in a module are visible to each other (i.e., all module-internal bindings are in scope).
➡ Outside objects (e.g., those defined in other modules) are not visible unless **explicitly imported**.
➡ Objects are only visible on the outside (i.e., their binding's scope can extend beyond the module) if they are **explicitly exported**.

Visibility vs. Lifetime.
➡ **Lifetime** of objects is **unaffected**.
➡ **Visiblity** just determines whether compiler will allow name to be used: **a scope a rule**.



Thursday, March 4, 2010

Module / Namespace / Package

*A means to **structure names** and enable **information hiding**.*

Collection of named objects and concepts.

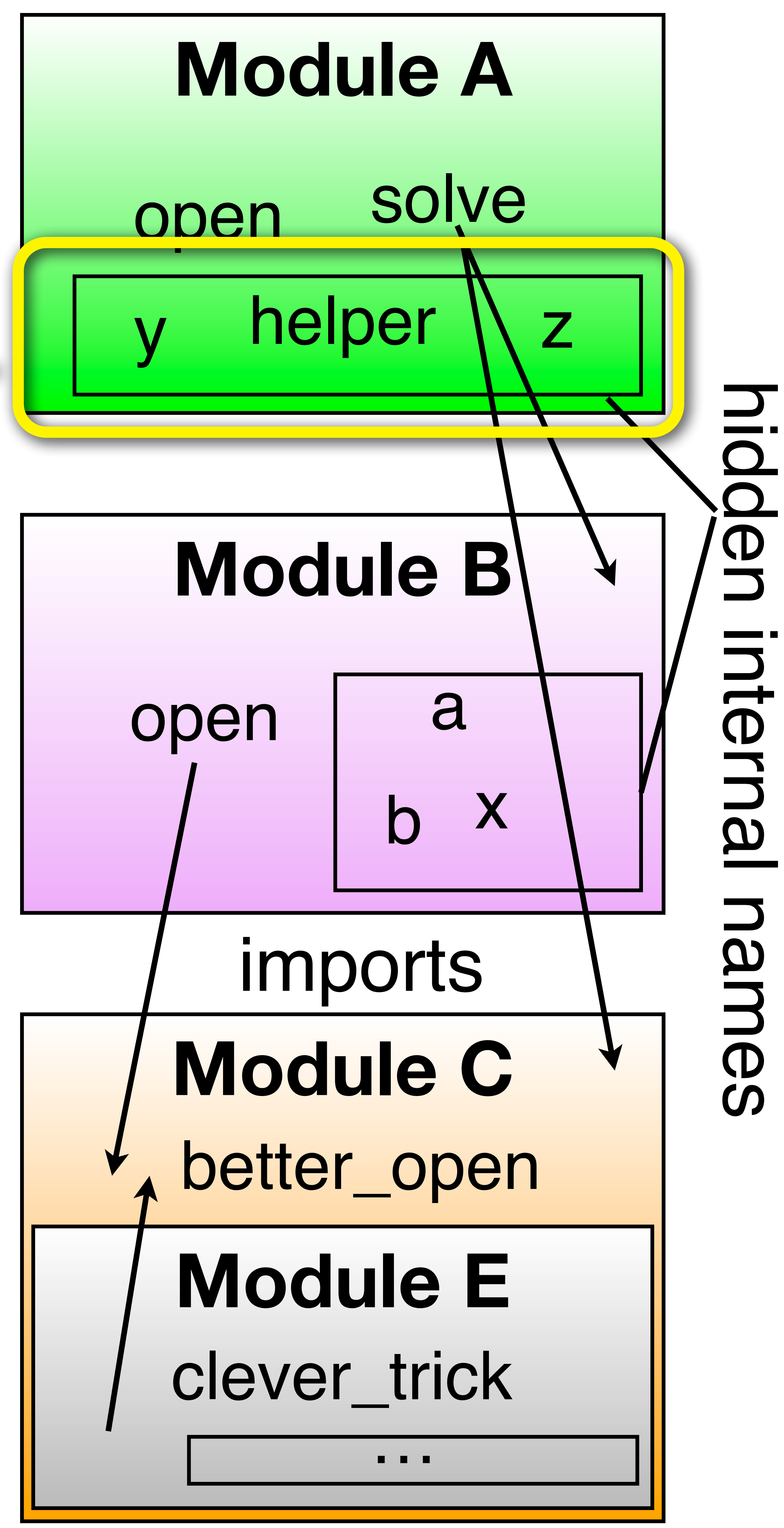
etc.

ther

cope).

Hide internal helper definitions: encourages decomposition of problems into simpler parts without "littering the global namespace."

- ➡ Outside objects (e.g., those defined in other modules) are not visible unless **explicitly imported**.
- ➡ Objects are only visible on the outside (i.e., their binding's scope can extend beyond the module) if they are **explicitly exported**.



Visibility vs. Lifetime.

- ➡ **Lifetime** of objects is **unaffected**.
- ➡ **Visiblity** just determines whether compiler will allow name to be used: **a scope a rule**.

Module / Namespace / Package

*A means to **structure names** and enable **information hiding**.*

Collection of named objects and concepts.

➡ Subroutines, variables, constants, types, etc.

Encapsulation: constrained visibility.

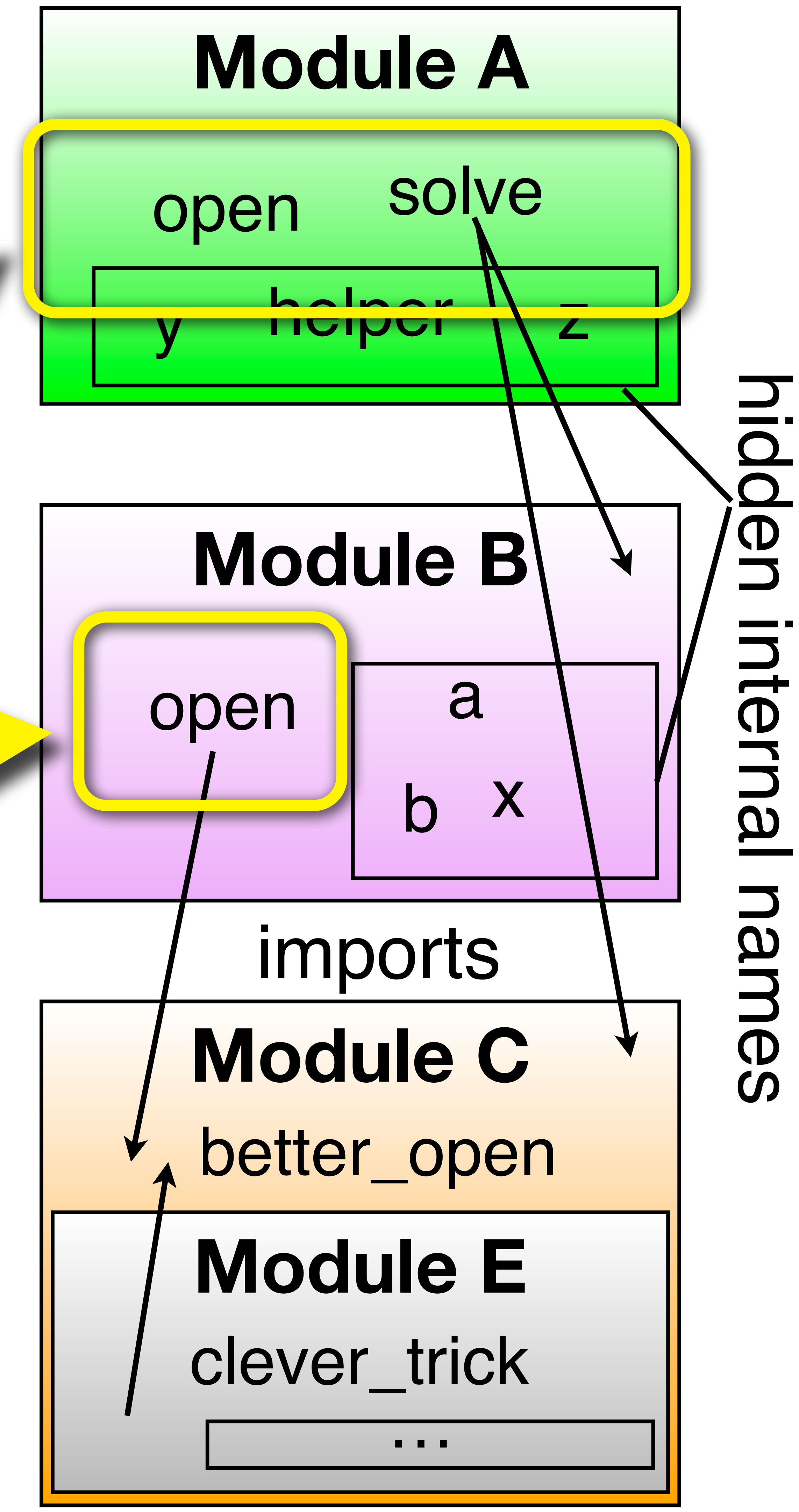
➡ Objects in a module are visible to each other

. orted. . e) if

they are **explicitly exported**.

Selectively import desired names

Avoid unintentional name clashes.



Visibility vs. Lifetime.

➡ **Lifetime** of objects is **unaffected**.

➡ **Visiblity** just determines whether compiler will allow name to be used: **a scope a rule**.

Imports & Exports

Scope "permeability."

- **closed**: names only become available via **imports**.
  - Anything not explicitly imported is not visible.
- **open**: exported names become **automatically visible**.
  - Can hide internals, but referencing environment can be large.
- **selectively open**: automatically visible with **fully-qualified name**; visible with "short name" only if imported.

```java
import java.io.IOException;
…
  try {
    …
  } catch (IOException ioe) {
    …
  }
```

```java
// no import!
…
  try {
    …
  } catch (java.io.IOException ioe) {
    …
  }
```

Java package scopes are selectively-open.

Imports & Exports

Scope “permeabi

- **closed**: names c
  - Anything not ex
- **open**: exported r
  - Can hide intern
- **selectively open**: automatically visible with **fully-qualified name**; visible with “short name” only if imported.

Closed wrt. “short names”:
`IOException` becomes only available after **explicit** import.

```
import java.io.IOException;
…
  try {
     …
  } catch (IOException ioe) {
     …
  }
```

```
// no import!
…
  try {
     …
  } catch (java.io.IOException ioe) {
     …
  }
```

Java package scopes are selectively-open.

Imports & Exports

Scope "permeability.

- ➡ **closed**: names only
  - Anything not explici
- ➡ **open**: exported nam
  - Can hide internals,
- ➡ **selectively open**: automatically visible with **fully-qualified name**; visible with "short name" only if imported.

Open wrt. fully-qualified names:
`java.io.IOException` is always visible and thus a valid name.

```
import java.io.IOException;
…
  try {
    …
  } catch (IOException ioe) {
    …
  }
```

```
// no import!
…
  try {
    …
  } catch (java.io.IOException ioe) {
    …
  }
```

Java package scopes are selectively-open.

Imports & Exports

Scope "permeability."

- ➡ **closed**: names only become available via **imports**.
  - Anything not explicitly imported is not visible.
- ➡ **open**: exported names become **automatically visible**.
  - Can hide internals, but referencing environment can be large.
- ➡ **selectively open**: automatically visible with **fully-qualified name**; visible with "short name" only if imported.

```
import j
…
  try {
    …
  } catc
    …
  }
```

In Algol-like languages, **subroutine scopes** are usually **open**, but module scopes are often **closed** or **selectively-open**.

```
ioe) {
    …
  }
```

Java package scopes are selectively-open.

Opaque Exports

Hide implementation detail.

- **Export type** without implementation detail.
  - A map ADT could be a hashmap, a tree, a list, etc.
- Want to **export the abstract concept**, but not the realization (which could change and should be encapsulated).

Opaque export.

- Compiler **disallows any references to structure internals**, including **construction**.
- Explicitly supported by many modern languages.
- Can be emulated.

```
public interface Thing {
  void doIt();
}

public class ThingFactory {

  static public Thing makeAThing() {
    return new ThingImpl(…);
  }

  private class ThingImpl implements Thing {
    …
  }

}
```

Emulating opaque exports in Java.

Module as a...

... **manager**.

- ➡Module **exists only once**.
- ➡Basically, a collection of subroutines and possibly types.
- ➡Possibly hidden, internal state.
- ➡Java: packages.

... **type**.

- ➡Module can be **instantiated** multiple times.
- ➡Can have references to modules.
- ➡Each instance has its private state.
- ➡Precursor to object-orientation.
- ➡Java: class.

Capturing Bindings / Scope

- Scope of a binding can be extended via **closures**.
- When a closure is defined, it **captures** all **active bindings**.
- We'll return to this when we look at nested subroutines and first-class functions.