ECS 140A
Programming Languages

October 19, 2023

Administrative stuff

Your midterm exam is Tuesday. Only 60 minutes for your exam.

Scope is everything so far through Java:
lectures, discussions, Chapters 1, 2, 5, and 6 in your textbook plus some knowledge of Java (write with some competency, read with greater competency).

Closed book, but you may bring two (2) pages of notes (8.5 x 11 inches). Single-sided or double-sided, it's up to you, but no more than two pages. No electronics.

Don’t start writing until I say you may begin. Stop writing when I say you’re out of time. Students have failed the exam for continuing to write after time is called.

Why this stuff is very very cool*

Because an object of a class that implements an interface is also an object of that interface type. That concept is the basis of an important object-oriented programming principle called polymorphism.

*assuming you come from the planet Nerdtron

Why this stuff is very very cool*

Because an object of a class that implements an interface is also an object of that interface type. That concept is the basis of an important object-oriented programming principle called polymorphism.

Polymorphism is derived from the word fragment *poly* and the word *morpho* in Greek, and it literally means "multiple forms".

*assuming you come from the planet Nerdtron

Why this stuff is very very cool

Polymorphism simplifies the processing of various objects in the same class hierarchy by using the same method call for any object in the hierarchy. We make the method call using an object reference of the interface.

At run time, the Java Virtual Machine determines which class in the hierarchy the object actually belongs to and invokes the version of the method implemented for that class.

from Java 5 Illuminated by Anderson and Franceschi

Polymorphism in action

Disclaimer: this animation probably wasn't produced with Java…polymorphism isn't specific to Java

These aren't penguins, they're polymorphic objects!

(From the movie "Happy Feet")

Much less exciting example

```
public class SimVend2025
{
  public static void main (String[] args)
  {
    VendingMachine foo1 = new CokeMachine2025();
    VendingMachine foo2 = new FrenchFryMachine2025();

    foo1.vendItem();
    foo2.vendItem();
  }
}
```

Much less exciting example

```
public class SimVend2025
{
  public static void main (String[] args)
  {
    VendingMachine foo1 = new CokeMachine2025();
    VendingMachine foo2 = new FrenchFryMachine2025();

    foo1.vendItem();
    foo2.vendItem();
  }
}
```

```
Adding another CokeMachine to your empire
Adding another FrenchFryMachine to your empire
Have a Coke
9 cans remaining
Have a nice hot cup of french fries
9 cups of french fries remaining
```

Why this stuff is still very very cool

```
public class SimVend2025
{
  public static void main (String[] args)
  {
    VendingMachine foo1 = new CokeMachine2025();
    VendingMachine foo2 = new FrenchFryMachine2025();

    foo1.vendItem();
    foo2.vendItem();
  }
}
```

The little foos may look like VendingMachine objects to you and me, but Java knows the difference and finds the appropriate method for each foo. That makes our programming job a lot easier to do. Why?

Why this stuff is still very very cool

Because the alternative is to write lots of chunks of code that look like sort of like this (if they were written in English):

```
if we want to vend an item from foo1 and foo1 is a CokeMachine2025
then print "have a Coke" else
if we want to vend an item from foo1 and foo1 is a FrenchFryMachine2025
then print "have a hot cup of french fries" else
if we want to vend an item from foo1 and foo1 is a PizzaMachine2025
then print "have a previously-frozen pizza" else
if we want to vend an item from foo1 and foo1 is a SushiMachine2025
then print "have some raw fish that's been in this machine for weeks" else
if we want to vend an item from foo1 and foo1 is a BeerMachine2025...
```

As the number of classes within the same hierarchy grows, so does the size of the chunks of code represented above. Eeyow!

What have we just seen?

Two big principles of object-oriented programming:

- inheritance
- polymorphism

Two Java-specific features to implement these principles (though other languages have similar features):

- interfaces
- abstract methods

Two paths to the same place (sort of)

Interfaces (and the `implements` keyword) are one way of defining relationships between classes (more correctly, between a class and a specification for the interface of that class) and providing organizational structure to a collection of classes. That organizational structure is a hierarchy.

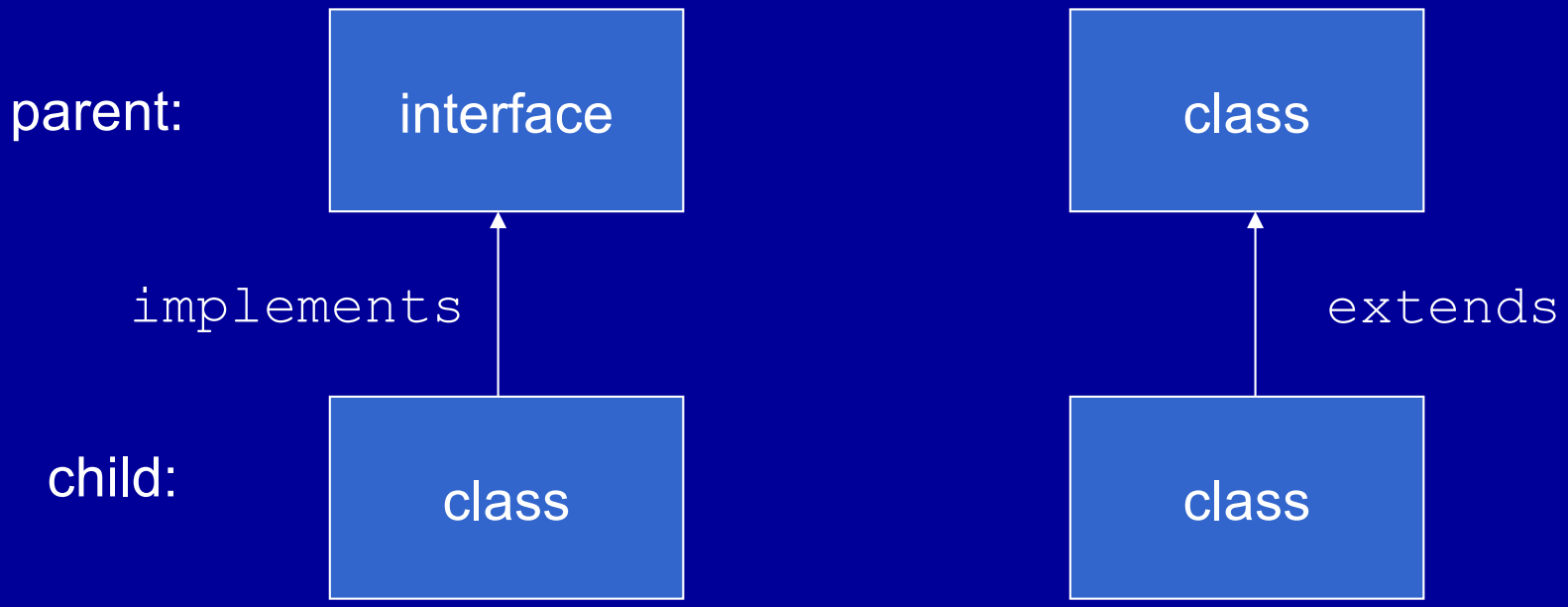
Inheritance (and the `extends` keyword) is another way of defining relationships between classes and imposing a hierarchical organization.

Two paths to the same place (sort of)

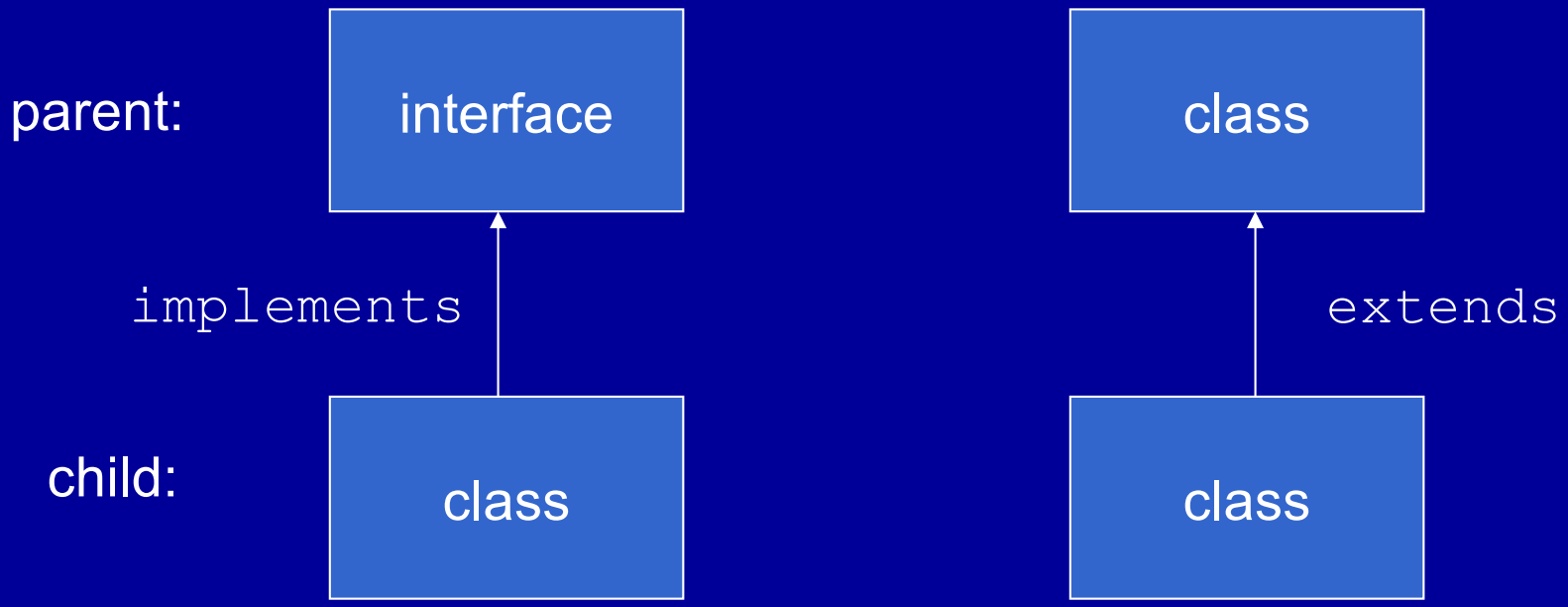
Interfaces (and the `implements` keyword) are one way of defining relationships between classes (more correctly, between a class and a specification for the interface of that class) and providing organizational structure to a collection of classes. That organizational structure is a hierarchy.

Inheritance (and the `extends` keyword) is another way of defining relationships between classes and imposing a hierarchical organization.

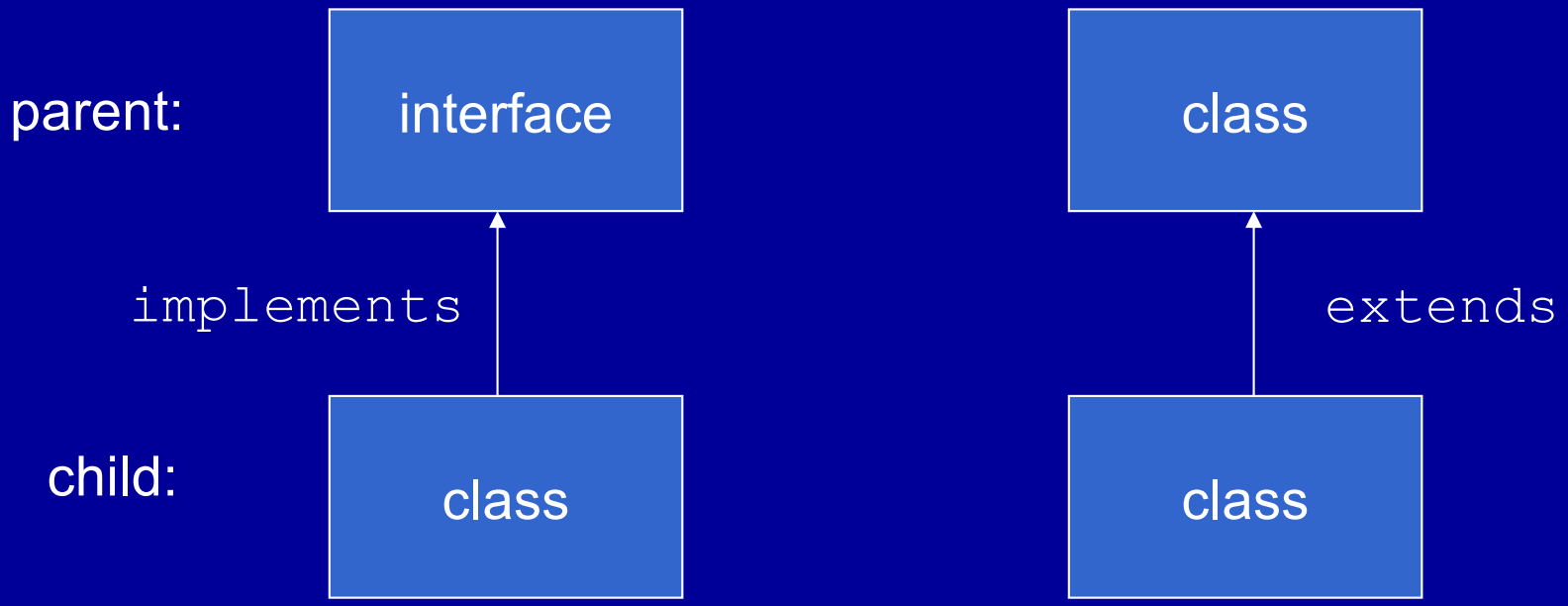
Why do we care about organization? Because when programs get real (where real == big), a well-organized program will make your programming life easier.



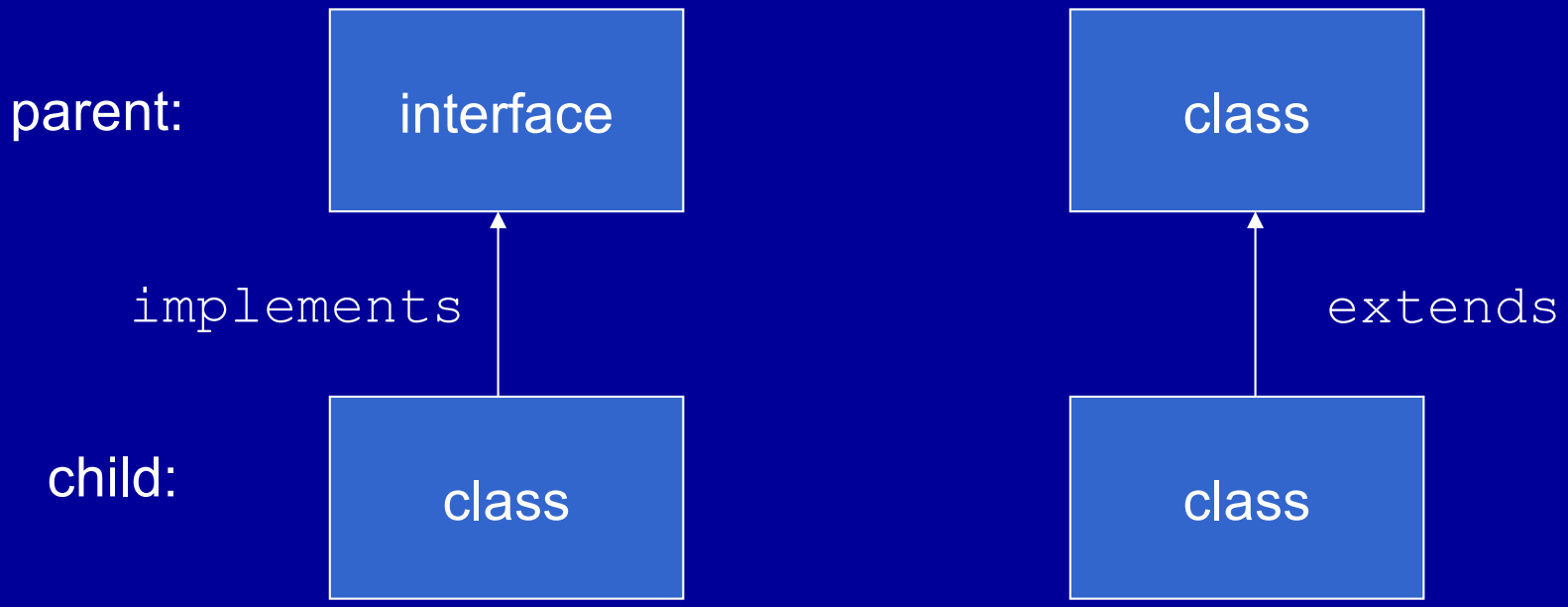
On the left, the parent interface serves as a specification for the child class. It tells Java what methods must be defined, and what the parameter lists must look like, before a class can be said to implement the interface. It does not tell Java what the methods must do or how they must do it. It only tells Java how the "outside world" interacts with objects of that class.



On the right, the parent class tells Java what methods and variables will be included in objects derived from the child class. This is lots more information than is provided by the interface, so the programmer has less to do when defining the child class. But the programmer may want to change the way the methods inherited by the child class behave, and that means some of the work comes back anyway.



In both cases, the parent prescribes "minimums" for the child. The child class definition isn't limited to what's shown in the parent -- in both cases we can add more variables and more methods to the child classes than what's prescribed by the parents.



This whirlwind introduction or reintroduction has left out more than a few details. Look at the Java reference we included in the course syllabus, or look online for other Java details as you need them. We'll count on the next homework assignment to motivate you to do your Java research as needed.

Questions?

Nothing past this point will be on the first midterm exam.

Semantics

Much less formality than syntax
Much more confusion than syntax

How to specify semantics

Language reference manuals

- The most commonly used way
- Imprecise due to inherent ambiguity in the natural language used to describe semantics

Reference implementations

- Specify semantics of a language by defining a translator (e.g., build the compiler)
- Can't answer questions about what a program means in advance - you have to run the program to know the answers

How to specify semantics

Formal definitions

- These are precise
- They are also complicated, abstract, not easy to understand
- Different methods are available
- Denotational semantics is probably best (what the book sort of uses…see chapter 7)

How to specify semantics

Denotational semantics

- *This is a formal, rigorous, and concise mechanism for describing language semantics. The idea is to create for each programming language entity (1) a rigorously defined corresponding math object and (2) a function that maps every instance of a programming language entity onto instances of math objects. These two tasks aren't very easy to do.*

How to specify semantics

Axiomatic semantics

- *Useful in proving the correctness of a program. It asks the question, "does the program compute what's specified?" but it doesn't ask how the computation is done. "Axiomatic semantics is a powerful tool for research into program correctness proofs, and it provides an excellent framework in which to reason about programs, both during their construction and later. Its usefulness in describing the meaning of programming languages to either language users or compiler writers is, however, highly limited."*
  (Concepts of Programming Languages by Robert W. Sebesta, 1999)

How to specify semantics

Operational semantics

- *The high-level language statements are described in terms of some known lower-level language for which a translator already exists, or is assumed can be built, or is at least understood and agreed upon. For example, Java is described in an intermediate language which is then translated by the Java Virtual Machine. But you have to assume that there is a Java Virtual Machine, and somebody will have had to describe the language understood by the Virtual Machine in terms of some lower-level language, and so on, until you get down to machine code.*

Names in programming languages

We tend to overlook this, but the use of names in PLs is a really big deal. It was the addition of names that pulled us out of the machine code primordial ooze and let us take our first steps with assembly languages. Then Fortran. And so on.

The use of names is a fundamental abstraction.

- We name variables, functions, parameters, classes, methods, types, ...
- "A fundamental step in describing the semantics of a language is to determine the meaning of each name used in a program." (your textbook)

Names in programming languages

How to form names (or identifiers)

- What can be used in a name?
- Usually some combination of letters, digits, and/or _
- C: (letter | _) {letter | digit | _}
- Dartmouth BASIC: Variable names were limited to A to Z, A0 to A9, B0 to B9, ..., Z0 to Z9, giving a maximum of 286 possible distinct variables!

Name length restrictions?

- Fortran: no more than 6 characters (names beginning with I,J,K,L,M,N were integer types by default)
- C, C++, Java: no restrictions

Names in programming languages

Are names case-sensitive?

- Most languages have case-sensitive names
- Some do not: Ada, Fortran, Pascal
- Enables hard-to-find errors

Are there special reserved words?

- Most PLs have reserved words (e.g., int, if, while, for...)
- Decisions here by language designers impact programmers
- COBOL has ~300 reserved words
- Programmers complain they can’t think of variable names

For example…Python identifiers

When we create an identifier, we follow two rules:

- The identifier contains only letters, numbers, and underscores.
- The identifier can't start with a number.

Numbers are 0 through 9.
Letters are the 26 characters in the English alphabet, both uppercase and lowercase.
The underscore is _.

Reserved words in Python

There are some identifiers that are reserved by Python. You can't use them as variable names. You should get familiar with these, but you don't need to memorize them.

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

You should also avoid using the name of previously-defined functions as variable names (e.g., `print, sum`).

Attributes

The meaning of a name is determined by the attributes or properties associated with the name.

A programming language enables us to define names and associate them with attributes.

For example, a program variable is a named abstraction of memory cells.

The compiler/interpreter translates between variable names and actual memory locations.

Attributes

Attributes of variables can be specified in a declaration (like in C or Java):

```
int x = 5;  // says x is of type integer and
            // the value of x is 5
```

On the other hand, in Python, the type of a variable is based on the value bound to the variable, so it can change during execution:

```
x = 100
...
x = "hello world"
```

Attributes

Typically, there are five kinds of attributes we are concerned with. What might they be?

Attributes

Typically, there are five kinds of attributes we are concerned with:

- Location: places values can be stored (e.g., where is this thing in memory?)
- Value: any storable quantity (1, 3.14, "hello")
- Type: determines possible values and operations (e.g., does addition work here?)
- Scope: the region of a program where a variable is accessible
- Lifetime: time duration where a variable is in valid state

Binding

The process of associating an attribute with a name is called binding. When the binding is done is called the binding time. There are different possible binding times for attributes of names, for example:

- language definition time
- language implementation time
- compile time
- link time
- load time
- execution time

Binding

- language definition time
- language implementation time
- compile time
- link time
- load time
- execution time

The first five all fall into the category of early binding...the binding is done before execution. The last one is in the category of late binding.

There's more late binding in (some) functional languages than in imperative languages. We also see it more often in interpreted languages vs compiled languages.

An example of bindings

Consider

```
int count;
...
count = count + 1;
```

Some binding questions that must be answered:

- possible types for `count`
- possible meanings for `+`
- possible values for `count`
- how is `1` represented
- does the meaning of `+` work with the type attributes of `count` and `1`?

An example of bindings

Consider

```
int count;
...
count = count + 1;
```

Some binding times:

- type of `count` is bound at compile time
- set of possible values for `count` is bound at language definition (or maybe language implementation) time
- meaning of the symbol `+` is bound at compile time, when the types of its operands have been determined (when are possible meanings of + bound?)
- internal representation of the literal `1` is bound at language implementation time (number of bits may vary by implementation)
- value of `count` is bound at execution time with this statement

Early vs. late binding

Early binding happens before runtime

- also called static binding

Late binding happens at runtime

- also called dynamic binding

Static binding is often associated with having to declare types with names (variables, functions, ...) when writing the program, before compilation.

Dynamic binding is often associated with not making those declarations in advance, and letting the binding be determined during execution.

Early vs. late binding

There are trade-offs between the two:

- Early binding usually leads to more efficient execution (because there's less work being done at runtime)
- Late binding usually leads to more flexibility for the programmer, but possibly at the expense of not seeing errors until runtime

Declarations

Declarations are a principal method for establishing bindings

Example 1: `int x = 0;  // C variable`

- Explicitly specifies the data type of x and its initial value
- Implicitly specifies the scope of x and its location in memory (we don't specify it, but the compiler knows what to do)

Example 2: `double f (int x); // C fcn prototype`

- Explicitly specifies f's type: int -> double
- Implicitly specifies nothing else
- Still needs another declaration to specify the code in the function body

Scope

The scope of a binding is the region of the program over which the binding applies

Example: In languages with nested blocks

```
int x;              // binds x to int
{
  double x; ... // binds x to double
}
```

Scope

The scope of a binding is the region of the program over which the binding applies

Example: In languages with subroutines/procedures/functions

- Upon entry to the subroutine
  - Create bindings for new local variables (declared for first time in subroutine)
  - Hide bindings for variables that are re-declared (called "scope holes")
  - Reference the variables
- Upon exit
  - Remove bindings for local variables
  - Re-activate bindings for variables that were hidden

Lexical scope

Also called static scope

- The scope of a binding is defined in terms of the lexical structure of programs
- The compiler can determine the scopes
- You can see the scopes because of the lexical structure
- All bindings for names can be resolved by examining the structure of the program
- Choose the most recent, active binding made at compile time
- Most compiled languages employ static scope rules (C, C++, Java)

Lexical scope

Classical example: the "most closely nested rule”

- A name is known in the scope where it is declared and in each enclosed scope, unless it is re-declared in an enclosed scope ("outside in")
- To resolve a reference to a name, examine the local scope and statically enclosing scopes until a binding is found ("inside out")

Examples…

Static or lexical scope example

```
begin
    integer x;
    procedure f;
        begin
            x = x + 3;
        end
    x = 20;

    begin
        integer x;
        x = 10;
        f;
        write x;
    end
    f;
    write x;
end
```

If static scope rules are in use, what is written?

Static or lexical scope example

```
begin
    integer x;
    procedure f;
        begin
            x = x + 3;
        end
    x = 20;

    begin
        integer x;
        x = 10;
        f;
        write x;
    end
    f;
    write x;
end
```

If static scope rules are in use, what is written? 10 and 26

More familiar syntax: static in C

```
#include <stdio.h>
int x;
void f () {x = x + 3;}
int main ()
{
  x = 20;
  {  int x;
     x = 10;
     f ();
     printf ("%d\n", x);
  }
  f ();
  printf ("%d\n", x);
  return 0;
}
```

Dynamic scope

Under dynamic scoping

- Bindings depend on program execution and calling sequences
- Thus, you can't always resolve scoping questions by just looking at the program (which makes the program more difficult to understand)
- Use the most recent, active binding made at run time to resolve a reference

Dynamic scope

Dynamic scoping isn't common, but is still found in some languages

- Early Lisp dialects assumed dynamic scope rules (there were many dialects)
- Common Lisp assumes lexical scope, but you can create variables with dynamic scope if you want to (not necessarily a good thing)
- Also optional in Perl. Found in bash and dash.
- Why still around? Easy to implement. Legacy in the case of Common Lisp.
- "The most horrible Lisp bugs may be those involving dynamic scope." [Paul Graham in ANSI Common Lisp]

Dynamic scope example

```
begin
    integer x;
    procedure f;
        begin
            x = x + 3;
        end
    x = 20;

    begin
        integer x;
        x = 10;
        f;
        write x;
    end
    f;
    write x;
end
```

If dynamic scope rules are in use, what is written?

Dynamic scope example

```
begin
    integer x;
    procedure f;
        begin
            x = x + 3;
        end
    x = 20;

    begin
        integer x;
        x = 10;
        f;
        write x;
    end
    f;
    write x;
end
```

If dynamic scope rules are in use, what is written? 13 and 23

More familiar syntax: dynamic in Bash

```
#!/bin/bash
X=0
function f () { let X=X+3 }
X=20

function dummy {
  local X
  X = 10
  f
  echo $X
}

dummy
f
echo $X
```