The Name Game

- The ability to refer to objects in programs with human-readable *names* is fundamental to all programming languages
- A *name* is a mnemonic character string used to represent something else — variables, constants, operations, types, subroutines, modules…
- Provides a simple way to reference *abstractions* of more complex objects in a program: subroutines are *control abstractions*, classes are *data abstractions* — and we give them names so that we can use them easily

- *Binding*: An association between two things, such as a name and the thing that it names
- *Lifetime (of a binding)*: The period of time from the creation to the destruction of the binding
- *Scope (of a binding)*: The textual region of a program in which the binding is *active*
- *Referencing environment (of a statement or expression)*: The set of active bindings; corresponds to a collection of scopes that are examined, in order, to find a particular binding
- *Scope rules*: Rules that determine this collection and its order

Binding Time

- Binding time is the point at which a binding is created
  - More generally, binding time can be viewed as the point at which an implementation decision is made
  - In a way, a binding answers a question ("what is this called?" or "what thing has this name?")
- *Static* vs. *dynamic* times — usually refers to whether something occurs before or after run-time, respectively, although distinction is not set in stone (in Scott's words, these are "coarse" terms)

- *Language design*: literals, keywords
- *Language implementation*: arithmetic precision, I/O
- *Program creation*: developer decisions and designations
- *Compile time*: variable, function, data type resolution
- *Link time*: references to pre-existing code/libraries — *separate compilation*
- *Load time*: virtual-to-physical memory mapping; currently available code/libraries
- *Run time*: values depending on user input, external factors — includes *start-up time*, *module entry time*, *elaboration time*, *subroutine call time*, *block entry time*, and *statement execution time*

Generalizations About Binding Time

Not set in stone, but in general…

- Early binding times are associated with greater efficiency, while later binding times are associated with greater flexibility
- Languages that do a lot of early binding tend to be compiled
- Languages that do a lot of late binding tend to be interpreted

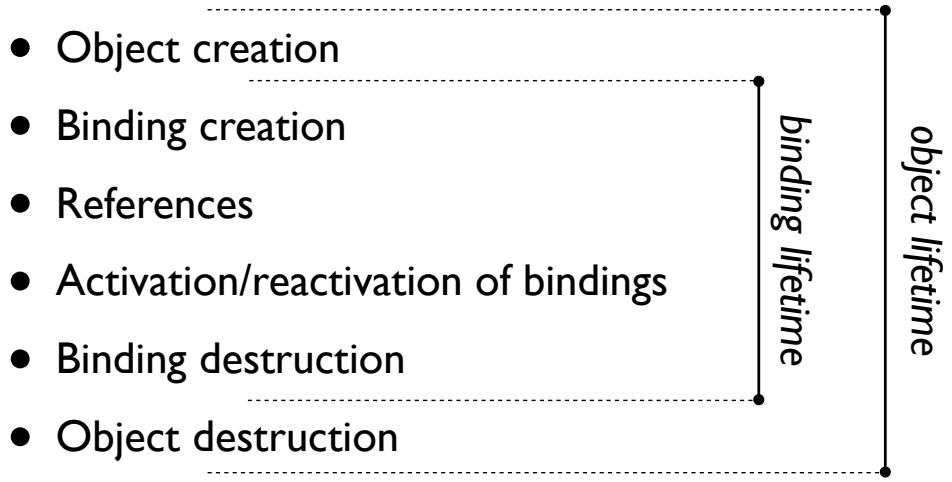
Binding Identifiers to Whatever They Identify

- A programming language's *scope rules* control the way identifiers are bound to their "identifiees"
- We have said that the *scope of a binding* is the textual region of a program in which the binding is active; we also use the word *scope* by itself as a program section of maximal size *in which no bindings change*, or *in which no re-declarations are permitted*
- For example, most languages *open* a new scope on subroutine entry; the term *elaboration* refers to the process of creating bindings when entering a scope

Object and Binding Lifetimes

- In this context, an "object" is just any "thing" that may have a name — not specific to object-orientation
- Creation and destruction of objects is distinct from the creation and destruction of their bindings
  - If an object outlives all of its bindings, then it's *garbage*
  - If a binding outlives its object, then it's a *dangling reference*
- Object lifetime typically corresponds to some *storage allocation* mechanism — *static*, *stack*, or *heap*
- Objects may be bound many times in its lifetime

Key Events in an Object's Lifetime



Static Allocation

- Absolute address (location), retained throughout a program's execution
  - Global variables
  - A program's machine-language translation
  - Local variables with persistent values — variables that retain values between subroutine invocations
  - Constant literals: numbers, strings
  - Compiler bookkeeping: debugging routines, type checking, garbage collection, exception handling

- In programming languages without recursion, subroutine constructs (a.k.a. *frames* or *activation records*) may be statically allocated
  - Includes local variables, elaboration-time constants (see below), arguments and return values, temporaries, and other bookkeeping information
- Constants — if their values are known at compile time (*compile-time constants*), they can be allocated statically at all times, regardless of recursion
  - Scalar compile-time constants may be stored directly in the target instructions themselves
  - There are also *elaboration-time constants* — constants whose values are not known until runtime: implemented as variables whose values may not be changed, but since their value is unknown until runtime, they cannot be statically allocated when local to a recursive subroutine

Stack-Based Allocation

- Benefits/motivations:
  - Recursive subroutine calls — impossible with static allocation
  - Allows memory reuse (always a "good thing")
- An individual subroutine invocation is represented by a *frame* or *activation record*
  - Arguments and return values usually reside at the bottom, for easy access by the caller, then you have parameters, local variables, bookkeeping information (implementation dependent)
  - Frames also refer to the correct instance of the routine in which it was declared — follow these links to find non-local variables
- Stack maintenance is handled by the subroutine *calling sequence* (the code that precedes and succeeds an actual subroutine call) and a subroutine's *prologue* and *epilogue* (code that executes at the beginning and end of the subroutine, respectively)
  - Sometimes the term "calling sequence" is applied to the whole shebang — code before the call, code at the beginning of a subroutine, code at the end, and finally code after the call returns
  - We save space by doing more work in the prologue and epilogue than the calling sequence (calling sequence: repeated per subroutine called; prologue/epilogue: written out only once per subroutine)
- *Relative* to the current stack frame, offsets to objects can be statically determined
  - Implement by dereferencing a *frame pointer* which tells you where you are in the stack

Heap-Based Allocation

- In this context, a *heap* is a region of storage in which blocks of memory can be allocated and deallocated at arbitrary times
  - Same term as, but completely unrelated to, the tree data structure of the same name
- Useful for anything that needs to be dynamically allocated: linked data structures, resizable objects
- Primary space issue: heap *fragmentation* (*internal* == more space per block than needed; *external* == scattered blocks eliminate large contiguous regions)

- Heap implementation:
  - Maintain one or more *free lists* — linked lists of unused heap blocks
  - Allocate blocks by *first fit* or *best fit* — no clear "better" algorithm, as it depends on the distribution of size requests
  - Multiple free lists provide *pools* that for different block size requests
    - Pools may be statically or dynamically allocated; common dynamic algorithms include *buddy system* and *Fibonacci heap*
  - Ability to satisfy requests degrades over time due to external fragmentation; may need to *compact* the heap
- Garbage collection: for objects that outlive bindings
  - *Explicit deallocation* — leave it to the programmer: language implementation simplicity and execution speed, but costly if the programmer screws up (dangling references, memory leaks)
  - *Implicit deallocation* — "unreachable" objects qualify for deallocation by a *garbage collection* mechanism: makes language implementation much tougher, but viewed as essential (i.e., worth the trouble) these days

Modules

- Scott: "A major challenge — perhaps *the* major challenge — in the construction of any large body of software is how to divide the effort among programmers in such a way that work can proceed on multiple fronts simultaneously."
- a.k.a. "Programming in the large" — and these days, what programming *isn't* in the large?
- a.k.a. "Modularization of effort" — thus the concept of *modules* in programming languages

- Key concepts in modules:
  - *Information hiding*: keep objects and algorithms invisible when appropriate, particularly with design or implementation decisions that are most likely to change in the future
  - *Cognitive load on programmer*: minimize the amount of information needed to understand the code
  - *Narrow interfaces*: changes to interfaces affect the most code
- Specific to naming in programming languages, information hiding reduces the risk of name conflicts, prevents violation of data abstractions, and may help to isolate bugs better
- First cut at information hiding: nested subroutines
  - Hidden items only live as long as the subroutine
  - *save*/*own*/*static* variables were an initial solution

Module Design Elements

- Objects inside a module are visible to each other
- *Exporting* makes objects inside a module visible to code outside the module
- *Importing* allows code inside a module to see objects outside (which may need to be exported)
- Bindings to variables are inactive outside the module, but not destroyed
- Some languages (Euclid, Turing) prohibit *aliases* — simultaneous multiple bindings to the same object (e.g., Pascal's variant records, C's unions, references)

Module Terms in Selected Languages

cluster	Clu
module (interface/implementation)	Modula (1, 2, 3)
package	Turing, Ada, Perl, Java
namespace	C++
signature/structure	ML

- Note how a "module" is not the same as a "compilation unit" — they are similar but distinct concepts (more on this later)
- Ditto with classes in object-oriented languages (i.e, also similar but distinct): note how they are orthogonal in some languages (Java, C++)

Module-as-Manager Paradigm

- In the *module-as-manager* paradigm, modules exist as a single abstraction: at run-time, there is only one "instance" of a module, and by extension, there is therefore only one instance of that module's variables
- For module code to be reused across multiple objects, a separate *type* needs to be defined, which the module subroutines receive as an argument
  - Since these types go hand-in-hand with the module (usually as an abstraction that it exports), that module also provides subroutines for creating and possibly destroying instances of this type — thus the "manager" term

Module-as-Type Paradigm

- In other languages, a module *is* a type — its internal variables are "local" to its subroutines, or subroutines may be viewed as "belonging" to a specific instance of the module
- The *class* in object-oriented languages may be thought of as module-as-type with *inheritance*, which allows new classes to refine or extend existing ones
- Module-as-type allows module subroutines to appear like binary or *n*-ary operations on the module type (e.g., *a.equals(b)* as opposed to *equals(a, b)*)

Modules and Separate Compilation

- The separation of concerns provided by modules gives them a natural correspondence with *separate compilation* — the ability to construct a program in distinct fragments or *compilation units*
- As mentioned, they are actually distinct ideas, though the natural fit makes them appear together a lot:
  - Modula-3, Ada: modules explicitly intended for separate compilation
  - C: files are primarily compilation units, with "conventions" allowing module-like behavior
  - Java: compilation unit (*class*, *interface*) distinct from module (*package*)

Evolution of Data Abstraction Facilities

none	ForTran, BASIC
subroutine nesting	Algol 60, Pascal, et. al.
persistent values in local variables	Algol 68, ForTran, C, et. al. (*own*, *save*, *static*, respectively)
module as manager	Modula, C files (in a way)
module as type	Simula, Euclid
classes, with inheritance	Simula, Smalltalk, C++, Eiffel, Java, et. al.

Scope Rules

- Two categories of scope rules: *static* or *lexical* scope rules, and *dynamic* scope rules
- *Static* or *lexical* scope rules define a scope in terms of the physical (a.k.a. lexical) structure of a program
  - Scopes can be determined by the compiler
  - All bindings for identifiers can be resolved by examining the program
  - Most compiled languages employ static scope rules

- *Dynamic* scope rules depend on the current state of program execution to determine bindings
  - Bindings cannot always be resolved by examining the program because they depend on calling sequences
  - To resolve a reference, dynamic scope rules use the most recent, active binding made at runtime
  - Dynamic scope rules are usually encountered in interpreted languages
  - Dynamic scope rules typically imply no type checking at compile time, since you can't always determine a reference's type under dynamic scope rules

Static Scope Rules

- *Most closely nested rule* (introduced by *block-structured* languages): an identifier is known in the scope in which it is declared and in each enclosed scope, unless it is redeclared in an enclosed scope
  - To resolve a reference to an identifier, examine the local scope and statically enclosing scopes until a binding is found
  - Applies to nested subroutines, as subroutines typically define a scope — permitted in many languages, though not in C and its descendants
  - Redeclared identifiers hide the binding for that identifier in the enclosing scope — called a *hole* in the scope — which some languages may "plug" by allowing a *qualifier* or *scope resolution operator* (e.g., *super* or *this* in Java; :: delimiter in C++)

- *Module import/export rule* (introduced by *modular* languages — languages with, duh, modules): an identifier declared within a module may be referenced in the enclosing scope only if it is *exported*; an identifier outside a module may be referenced within that module only if it is *imported*
  - The most closely nested rule applies *within* a module
  - Import/export of identifiers may be *implicit* or *explicit*
  - A module that requires explicit import is said to be a *closed* scope
  - A module for which identifiers that are not redeclared are visible/ inherited from the enclosing scope is said to be an *open* scope
- *Older, simpler rules*: single global scope (early versions of Basic), explicit *common* blocks (ForTran 77)

Static Scope Rules from Specific Languages

- Algol 60, Pascal: "classical" most closely nested rule
- Modula-2: explicit export from the defining module *and* explicit import into the using module
- C++ (namespaces): explicit import only; everything in the namespace is implicitly exported
- Euclid: *all* scopes are closed
- Java, other object-oriented languages: object-oriented *classes* represent a generalization of modules, and have more sophisticated static scope rules

Static Scope and Module Observations

- Closed scopes make it less likely to use a variable by mistake — explicit importing can be seen as "forced documentation"
- Subroutines are also closed scopes, but they always destroy bindings upon subroutine exit; modules are closed scopes *without* this limited lifetime
  - Bindings declared in a module are only *inactive* when outside the module, not destroyed
- *save* (ForTran), *own* (Algol 68), or *static* (C) variables provide similar facilities to bindings inside subroutines

Declaration Scope Issues

- A binding is typically formed through a *declaration* or *definition* in the source code
- Language design issue: does/should the scope of this binding include the portion of the scope (in the standalone sense of the word) before its declaration?
- Pascal says that the scope of an identifier is the entire block in which it is declared, excluding sub-blocks in which the identifier is redeclared; however, identifiers must be declared before they are used…

```
...
const
  A = 10;
...
procedure P;
  const
      B = A;
      ...
      A = 15;
...
```

```
...
const
   foo = 10;
...
procedure P1;
   ...
   procedure P2;
       var A: integer;
   begin
       ...
       A = foo;
       ...
   end {P2};
   ...
   procedure foo;
...
```

Pascal errors: *A* on the left and *foo* on the right are both used before they are declared — but what do you think the programmer meant? (particularly in the case of *foo*!)

Declaration Rules in Other Languages

- *Ada, C, C++, Java*: identifier scope ranges from declaration to end-of-block/scope
- *C++, Java*: no declare-before-use for members, but not locals
- *Perl, JavaScript*: declaration not necessary; identifiers have a default initial value…convenient, but you generally don't want to do this anyway
- *Lisp, ML*: explicit scope delimiters (*let*, *let**, *letrec*, *local*)

Separating Declaration from Definition

- Helps with recursive declarations
- Facilitates information hiding (interface vs. implementation)
- Accomplished in various ways: *forward* constructs, separate header or interface files
- …but when combined with the need to divide programs into separate sections that can be worked on in parallel, we get the overall concept of *modules*

Static Scope Rule Implementation

- Maintain a *symbol table* at compile time — maps names to information associated with them
  - Basic operations: *insert* and *lookup*
  - Functionality wrinkles (or, why a symbol table isn't just a plain old dictionary): nested scopes, forward declarations, saving for use in a symbolic debugger
- Established/published symbol table approaches: Graham-Joy-Rubine (1979), LeBlanc-Cook (1983, detailed in Scott), and many more

LeBlanc-Cook Symbol Table

- Each scope gets a unique serial number
  - Outermost scope (predefined identifiers) == 0
  - Programmer-declared globals == 1…etc.
- Single hash table for all names
  - Individual name records include name category (variable, constant, type, procedure, etc.), scope number, type (pointer to another symbol table entry), and more
- Scope stack for the current referencing environment
  - Entries contain scope number, whether the scope is closed, etc.
- Traverse name entries and scope stack for lookup

Dynamic Scope Rules

- Main rule: the current binding for a given name is the one encountered most recently during execution and not yet destroyed by returning from its scope
- Dynamic scope → dynamic semantic checks → interpreted languages
  - Because bindings depend on calling sequence, many language rules are dynamic (after run-time) instead of static (before run-time): type checking in expressions, argument checking in subroutine calls
- Sample languages: APL, Snobol, early Lisp, Perl (explicit in versions ≥ 5: *local* variable declarations)

```
#
# Dynamic scope sample: subroutine customization
#
our @listToPrint = qw/<nothing>/;

sub printList {
    foreach $item (@listToPrint) {
        print "$item; ";
    }
    print "end\n";
}

# Compare using my, our, local, and no modifier on @listToPrint.
sub printHis {
    local @listToPrint = qw/Tom Dick Harry/;
    printList();
}

sub printHers {
    local @listToPrint = qw/Jane Mary Muffet/;
    printList();
}

printHis();
printHers();
printList();
```

```
{Tom; Dick; Harry; } end
{Jane; Mary; Muffet; } end
{<nothing>; } end
```

Dynamic Scope Rule Implementation

- Option 1: maintain a stack (*association list*, or *A-list*) of active variables; resolve a binding by searching from the top of the stack — slow access, fast calls
- Option 2: maintain a lookup table for variable names; may need hash function for lookup, and subroutines need to manipulate table entries for local variables — fast access, slow calls
- Corresponds to symbol table lookup in statically-scoped languages: while static scope rules are more complicated, we resolve them at compile time only

Dynamic Scope Pros & Cons

- Pros
  - Simple implementation for interpreted languages
  - Implicit modification of program behavior (e.g., subroutine customization)
  - Lack of static structure (e.g., environment variables in scripts, default handlers for exceptions instead of fixed try/catch blocks)
- Cons
  - High run-time cost
  - Programs are more confusing, harder to understand
- Alternatives: static variables, default parameters

Binding Rules

- Core issue: in a programming language that can create *references* to subroutines (subroutines in variables, subroutines passed as parameters), what referencing environment applies when that subroutine reference is called? — these are the *binding rules* of the language
- *Shallow binding*: called subroutine sees the referencing environment at the time it is *invoked*
- *Deep binding*: called subroutine sees the names that were active at the time it was *defined*

- Terminology check: *first-class* values are those that can be passed as parameters, returned from subroutines, or assigned to variables; *second-class* values can only be passed as parameters; *third-class* values can do none of these
  - Binding rules are therefore irrelevant to languages where subroutines are *third-class* values (PL/I, Ada 83)
- The binding rule issue also applies only to identifiers that are neither global nor local
  - So, languages without nested subroutines (C, C++, Java), or where nested subroutines aren't first- or second-class (Modula-2), don't care
- Deep binding implementation: include the subroutine's referencing environment (e.g., A-list entry, stack static link) with its code pointer — this is called a *closure*

Overloading

- A name is *overloaded* when the name alone may refer to more than one object in a given scope
  - Conceptual opposite of *aliases*, where an object may be bound to more than one name in a given scope
- Solution: context of a name's use must contain sufficient information to disambiguate the overload
  - Subroutine signatures — arguments determine context
  - Arithmetic operators — "+" chooses between integer and floating-point implementations (and string concatenation, in some languages)

Related to Overloading, but not the Same

- *Coercion*: automatic conversion of an object of one type to an object of another type
- *Generics*: subroutines or modules with *polymorphic parameters* — parameters can take on more than one type, and the generic definition can serve as a *template* for one or more concrete versions
- *Polymorphism*: literally means "having many forms," so in this sense overloading is sometimes called "ad-hoc polymorphism" — but there's a lot more to this term than "mere" overloading…

Polymorphism

Apropos of the name, there are "many forms" or variants of polymorphism

- *Parametric polymorphism*: two kinds
  - *Explicit* variant is foundation of generics and templates — a single subroutine may act on parameters of multiple types, with explicit restrictions made by the programmer when needed
  - *Implicit* variant lets the programmer do whatever "makes sense" to the language — the language determines which types are appropriate in which contexts and enforces them for you (found primarily in functional languages — Lisp et. al. do it at run-time, while ML does "compile-as-you-go")

- *Subtype polymorphism*: bread and butter of object-oriented programming — calls the correct method or virtual function based on the specific subclass, without explicitly knowing what that subclass is
  - In short, subtype polymorphism is largely responsible for why *these* objects (in the object-oriented sense of the word) "do the right thing"