A name is a string representing code or data

A binding is an association between a name and code or data

A referencing Environment is a complete set of active bindings at a point in a program

The scope of a binding is the region of a program or time interval in the program’s execution during which the binding is active

A scope is a maximal region of the program where no bindings are destroyed

Compile time: bindings are encoded in machine code

Link time: bindings are initiated by compiler but finalized by linker

Load time: Bindings are finalized when program is loaded

Run time: bindings become active during execution

Early binding: compile link load time. Faster code execution. More efficient. Typical in compiled languages. Example: Functions in C

VS

Late binding: run time. greater flexibility. Typical in interpreted languages. Example: Lambda expressions in Scheme

Entity lifetime: lifetime of code or data

Binding lifetime: lifetime of association between name and object

Storage Allocation

Stack: connection with subroutine/method/function call. Lifetime from invocation to return from subroutine. Allocation is automatic with each call.

Stack Based Allocation: when a subroutine called, stack frame is pushed on the stack.

Heap: stored on heap. Created and destroyed at arbitrary times. Eg: JAVA object

Heap-Based Allocation: region of memory from which entities can be dynamically allocated

Heap-Based De-allocation: Explicit deallocation by programmer. Eg C C++. Efficient Automatic de-allocation by garbage collector. Eg: JAVA. Bad for real-time systems. Safer if new user.

Data/Static: Stored at a fixed absolute address, lifetime is the whole program execution. Eg: global variables. Allocated at compile/link/load time

Code

Stack frame contains: Arguments passed to the subroutine. Space for return value(optional) return address of caller. Local variable. Temporary variables. Registers that must be saved across calls.

Before the Call

Caller push arguments on the stack, push dummy return value(optional) execute call instruction

During the Call

Saves base pointer, allocate local variable, allocate temporary variable, save registers, performs body of subroutine, restores registers, destroys local and temp variables, return.

After the Call

Pop arguments off the stack

* Lexical (static) scoping

Binding based on nesting of blocks. Can be determined at compile time

Idea: Current binding for a name is the one encountered in the smallest enclosing lexical unit.

You use this in C, Java, etc.

**Nested Classes:** supported in languages like Java

**Nested Functions:** supported in languages like Pascal

* Dynamic scoping
  + Binding depends on flow of execution at run time
  + Can be determined only at run time
  + More complex and difficult to use
  + Seen in Logo, (uncommon) Lisp, bash
  + Some languages permit programmers to choose static or dynamic scoping for a variable: Common Lisp, Perl

Current binding for a given name is the one

* Encountered most recently during execution
* Not hidden by another binding for the same name
* Not yet destroyed by exiting its scope
* This happens when a language implementation uses a single global reference environment instead of linked environments

Closures

* A subroutine is a general term for a procedure or a function
* Idea: Passing subroutines is allowed in many languages
  + Reference to a subroutine can be passed as a parameter
  + Subroutine has access to all active bindings in its scope
* Idea: *Referencing environment* of subroutine contains all the active bindings
  + **If deep binding is used, referencing environment is created when subroutine is passed**
  + If shallow binding is used, referencing environment is created when subroutine is called
* Closures are a construct found in languages with **deep binding**
* A *closure* consists of
  + A reference to a subroutine
  + A referencing environment
* Analogy: A program and its data.
* This is different from an object:
  + **object** = data + operations on the data
  + **closure** = subroutine (1 op) + data that it needs
* Idea: Closures can be used like objects
* Challenge: implementing closures when they can be returned by functions
  + When the subroutine is invoked, the scope it refers to (from where it was created) may no longer exist
  + Scopes must be preserved for use in closures

Scheme Referencing environments

* A ***frame***is a collection of variable-object bindings
* Frames can point to parent frames, resulting in a chain of frames
* A reference environment is a chain of frames starting with the most local frame
* Variable x in environment E is unbound if none of E’s frames binds x
* Otherwise x’s value is the value bound to it in the closest frame that contains a binding for x
* **Static variables** (in C) provide “private objects” to a single subroutine or compilation unit
* **Compilation units** (modules) provide the same set of “private objects” to a group of subroutines. (Essentially a single instance of a class.)
* **Packages** contain multiple compilation units, but do not always provide a single interface
* **Module types** can be instantiated, like classes but without inheritance
* **Classes** add inheritance to module types

Open VS closed scope

* **Open scope**: All public bindings are visible by default (do not have to be imported)
  + Example: C
* **Closed scope**: All public bindings must be explicitly imported before being used.
  + Closed scopes force the programmer to clearly specify the interface
  + Example: In Modula-2 visibility is explicitly specified using IMPORT and EXPORT
* **Selective Open scope**:
  + Default packages are visible by default
  + All other packages, modules, or classes have to be imported
  + Examples: Java, C#, Perl, Python, Ada, and Haskell provide selectively open scopes
* Packages are usually higher level units of organization comprising 1+ modules or classes