Names, Bindings & Scope

Names

- Identifiers
- Used for constants, variables, subprograms, classes ...
- Meaning?
  - Binding
  - Scope
- Design
  - Case sensitivity
  - Reserved vs keywords
  - _, camel notation, special forms (e.g. php, ruby)

Variables

- Primarily imperative but also in others (e.g. parameters)
- Attributes
  - Name
  - Anonymous variables
  - Address
  - l-value
  - Alias
  - Type
  - Value
  - r-value
### Binding

- Choice of attribute
  - Any entity in program
- Binding time
  - Time choice made
  - Language design time (e.g. \* means multiplication)
  - Language implementation time (e.g. `int` in C)
  - Compile time (e.g. type of variable in Java)
  - Link time (e.g. address of external reference)
  - Load time (e.g. address of static variable in C)
  - Run-time (e.g. value of variable)
- E.g. `count = count + 10;`
- Static vs dynamic binding

### Type Binding

- Must occur before a variable can be referenced
- Static
  - Occurs before execution, usually at compile time
  - Explicit vs implicit declaration
  - Type inference
    - Type deduced from context
    - E.g. from initial value
    - Common in functional languages
- Dynamic
  - Cannot/is not determined at compile time
  - Must occur before value used
    - On assignment
  - Flexible but costly
  - Scripting & pure OO languages
  - Interpretation

### Address (Storage) Binding

- Allocation/deallocation
  - Lifetime
    - Static, stack-dynamic, explicit heap-dynamic, implicit heap-dynamic
- Static
  - Bound before execution
  - Access speed vs flexibility & storage efficiency
- Stack-dynamic
  - Bound at elaboration of declaration
    - Run-time – method call/return
- Explicit heap-dynamic
  - Allocated by operator or method call
  - Implicit or explicit deallocation
- Implicit heap-dynamic
  - Supports dynamic type binding
Scope

- Scope
- Visibility
- Local, nonlocal, global
- Also modules, packages, classes, namespaces

Static Scope

- Determined at compile time
- Nesting
  - Of blocks
  - Of subprograms
  - Of other units (e.g., classes)
- Determining declaration
  - Check local scope
    - Not found – move to innermost encompassing scope
    - E.g.
- Hidden declarations
  - "hole-in-scope"
  - Referencing

```javascript
function big1() {
    function sub1() {
        var x = 9;
        sub2();
    }
    function sub2() {
        var y = x;
    }
    var x = 3;
    sub1();
}
```
• Blocks
  - Declarations in compound statements
    * E.g.
  - Nested blocks
    * E.g.
  - Redefinition in inner block illegal in Java and C#
• Let in functional languages
  - Bind names to values for a single expression
    * E.g.
• Declaration order
  - Declarations before statements
  - Defined before use
  - Anywhere
    * Scope whole block or from declaration
  - Java for
• Global scope
  - Outside any unit - visible unless hidden
  - Declaration is definition - C
• Block structuring alone not sufficient hence encapsulation

```c
if (list[i] < list[j]) {
    int temp;
    temp = list[i];
    list[i] = list[j];
    list[j] = temp;
}
```

```c
void sub() {
    int count;
    ...
    while (...) {
        int count;
        count++;
        ...
    }
    ...
}
```
Dynamic Scope

- Determined at execution time
- Declarations are "executable"
- Meaning of name is the most recent declaration executed
  - Dependent on order of execution
  - Static analysis not generally possible
- E.g.
  - Requires interpretation
  - Cannot tell the meaning of a statement until it is executed
- Not commonly used
### Scope vs Lifetime

- **Lifetime**
  - Period during execution that a variable is bound to a particular address
- **Related to scope**
  - E.g. from method entry to method exit
- **Exceptions**
  - Static variables
  - Subprogram calls
  - Hiding
- **Lifetime must cover scope**

### Referencing Environment

- **The set of variables visible to a statement**
- **Static scope**
  - Locals plus non-hidden in static ancestors plus globals
  - Known at compile time
- **Dynamic scope**
  - Locals plus non-hidden in all active subprograms plus globals
  - Known only at execution time

### Named Constants

- "variables" whose values are set once and cannot be changed
- **Variations**
  - Value computable at compile time
  - Value computable at execution time from reference environment
- **Initialization**
  - Value binding at allocation time
  - For static variables value must be constant
  - For dynamic value must be computable at allocation time