Dynamic Type Binding

- Dynamic Type Binding (JavaScript and PHP)
- Specified through an assignment statement. For e.g., in JavaScript
  
  ```javascript
  list = [2, 4.33, 6, 8];
  list = 17.3;
  ```

- Advantage: flexibility (generic program units)
- Disadvantages:
  - High cost (dynamic type checking and interpretation)
  - Type error detection by the compiler is difficult
Variable Attributes (cont.)

• Type Inferencing (ML, Miranda, and Haskell)
  o Rather than by assignment statement, types are determined (by the compiler) from the context of the reference
  o fun circumference(r) = 3.1415 * r * r;

• Storage Bindings & Lifetime
  o Allocation – getting a cell from some pool of available cells
  o Deallocation – putting a cell back into the pool
Variable Attributes (cont.)

- The **lifetime** of a variable is the time during which it is bound to a particular memory cell.
Categories of Variables by Lifetimes

• Static
• Stack–dynamic
• Explicit heap–dynamic
• Implicit heap–dynamic

• In–class activity
Categories of Variables by Lifetimes

• **Static**—bound to memory cells **before execution begins** and remains bound to the same memory cell throughout execution, e.g., C and C++ **static** variables

  - **Advantages**: efficiency (direct addressing), history–sensitive subprogram support
  - **Disadvantage**: lack of flexibility (no recursion)
Categories of Variables by Lifetimes

- **Stack–dynamic**—Storage bindings are created for variables when their declaration statements are *elaborated*.
  (A declaration is elaborated when the executable code associated with it is executed)

- If scalar, all attributes except address are statically bound (i.e. before run time)
  - local variables in C subprograms/functions and Java methods
Categories of Variables by Lifetimes

- **Advantage**: allows recursion; conserves storage
- **Disadvantages**:
  - Overhead of allocation and deallocation
  - Subprograms cannot be history sensitive
  - Inefficient references (indirect addressing)

```c
int f() {
    static int a = 0;
    a += 2;
    printf("%d\n",a);
}

f() --> 2  f() ---> 4
```
Categories of Variables by Lifetimes

- **Explicit heap–dynamic** — Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution
- Referenced only through pointers or references, e.g. dynamic objects in C++ (via `new` and `delete`), all objects in Java
- **Advantage**: provides for dynamic storage management
- **Disadvantage**: inefficient and unreliable
Categories of Variables by Lifetimes

• *Implicit heap–dynamic*—Allocation and deallocation caused by assignment statements
  - all variables in APL; all strings and arrays in Perl, JavaScript, and PHP

• **Advantage:** flexibility (generic code)

• **Disadvantages:**
  - Inefficient, because all attributes are dynamic
  - Loss of error detection
Variable Attributes: Scope

• The *scope* of a variable is the *range of statements* over which it is visible
• The *nonlocal variables* of a program unit are those that are visible but not declared there
• The *scope rules of a language* determine how references to names are associated with variables
Static Scope

• Based on program text
• Scope of a variable can be statically determined (prior to execution)
• Find the variable declaration to connect a name reference
• *Search process*: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name
Static Scope

• Enclosing static scopes (to a specific scope) are called its *static ancestors*; the nearest static ancestor is called a *static parent*

• Some languages *allow nested subprogram definitions*, which create nested static scopes (e.g., Ada, JavaScript, Fortran 2003, and PHP)
Static Scope

Javascript nested subprograms

```javascript
function hypotenuse(a, b) {
    function square(x) { return x*x; }
    return Math.sqrt(square(a) + square(b));
}
```

Example credit: http://www.tutorialspoint.com/javascript/javascript_nested_functions.htm
Scope (cont.)

• Variables can be hidden from a unit by having a "closer" variable with the same name

    procedure Big is:
        X: Integer;
        procedure Sub1 is
            X: Integer; ...

• Ada allows access to these "hidden" variables
  o E.g., Big.X, unit.name
Blocks

• A method of creating static scopes inside program units—-from ALGOL 60

• Example in C:

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

• Legal in C and C++, but not in Java and C#— too error-prone
Declaration Order

- C99, C++, Java, and C# allow variable declarations to appear anywhere a statement can appear
  - In C99, C++, and Java, the scope of all local variables is from the declaration to the end of the block
  - In C#, the scope of any variable declared in a block is the whole block, regardless of the position of the declaration in the block
    - However, a variable still must be declared before it can be used
In C#,

```c
void F() {
    int x = 1, y, z = x * 2;
}
```

is the same as

```c
void F() {
    int x; x = 1;
    int y;
    int z; z = x * 2;
}
```

- In C++, Java, and C#, variables can be declared in `for` statements
  - The scope of such variables is restricted to the `for` construct
Global Scope

- C, C++, PHP, and Python support a program structure that consists of a sequence of function definitions in a file.
  - These languages allow variable declarations to appear outside function definitions.

- C and C++ have both declarations (just attributes) and definitions (attributes and storage).
  - A declaration outside a function definition specifies that it is defined in another file.
Global Scope (continued)

- PHP
  - Programs are embedded in XHTML markup documents, in any number of fragments, some statements and some function definitions
  - The scope of a variable declared in a function is **local to the function**
  - The scope of a variable implicitly declared outside functions is from the declaration to the end of the program, but skips over any intervening functions
    - Global variables can be accessed in a function through the `$GLOBALS` array or by declaring it `global`
Global Scope

- PHP

```php
$fruit = "apple";
function basket() {
    $fruit = "orange";
    print $fruit
    $gfruit = $GLOBALS['fruit']
    print $gfruit
}
```

Prints **orange** first and then **apple**
Global Scope (continued)

- Python
  - A global variable can be referenced in functions, but can be assigned in a function only if it has been declared to be `global` in the function

```python
>>> x=10
>>> def foobar():
...    global x
...    print x
...    x+=1
...  
>>> foobar()
10
>>> def foobar2():
...    x+=2
...  
>>> foobar2()
Traceback (most recent call last):
  File "<stdin>" , line 1, in <module>
  File "<stdin>" , line 2, in foobar2
UnboundLocalError: local variable 'x' referenced before assignment
```
Evaluation of Static Scoping

• Works well in many situations

• Problems:
  o Allows more access to variables than is necessary
  o Program evolution – As a program evolves, the initial structure is destroyed and local variables often become global; subprograms also gravitate toward becoming global, rather than nested
Dynamic Scope

• Based on calling sequences of program units, not their textual layout (temporal versus spatial)

• References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point
Scope Example

Big
- declaration of X
  Sub1
  - declaration of X -
  ...
  call Sub2
  ...
  Sub2
  ...
  - reference to X -
  ...
  ...
  call Sub1
  ...

Big calls Sub1
Sub1 calls Sub2
Sub2 uses X
Ada Scope Example

- **Static scoping**
  - Reference to X is to Big's X

- **Dynamic scoping**
  - Reference to X is to Sub1's X
Evaluation of Dynamic Scoping:

• Advantage: convenience

• Disadvantages:
  1. While a subprogram is executing, its variables are visible to all subprograms it calls
  2. Impossible to statically type check
  3. Poor readability- it is not possible to statically determine the type of a variable
Scope and Lifetime

• Scope and lifetime are sometimes closely related, but are different concepts
• For example: A static variable in a C/C++ function
Referencing Environments

• The *referencing environment* of a statement is the collection of all names that are visible in the statement.

• In a static-scoped language, it is the **local variables** plus all of the visible **variables** in all of the enclosing scopes.
Referencing Environments

- A subprogram is **active** if its execution has begun but has not yet terminated.
- In a dynamic-scoped language, the referencing environment is the **local variables** plus all **visible variables in all active subprograms**.
Named Constants

• A named constant is a variable that is bound to a value only when it is bound to storage

  • private static final int DAYS_IN_WEEK = 7;
  • private static final int COST_PER_MOVE = 25;
  • private static final double PI = 3.14159;
Named Constants

• Advantages:
  o readability and modifiability
  o Used to parameterize programs
  o The binding of values to named constants can be either static (called *manifest constants*) or dynamic
Named Constants

• Languages:
  o FORTRAN 95: constant-valued expressions
  o Ada, C++, and Java: expressions of any kind
  o C# has two kinds, `readonly` and `const`

  • the values of `const` named constants are bound at compile time
  • The values of `readonly` named constants are dynamically bound
Summary

• Naming conventions
• Variables are characterized by the attributes: name, address, value, type, lifetime, scope
• Binding type
• Variable lifetimes: static, stack dynamic, explicit heap dynamic, implicit heap dynamic
• Strong typing vs weak typing

• Update your website with relevant information about your language