Chapter 5
Names, Binding, Type Checking and Scopes
Names

- We discuss all user-defined names here

- **Design issues for names:**
  - Maximum length?
  - Are connector characters allowed?
  - Are names case sensitive?
  - Are special words reserved words or keywords?

- **Length**

  - If too short, they cannot be connotative

- Language examples:
  - FORTRAN I: maximum 6
  - COBOL: maximum 30
  - FORTRAN 90 and ANSI C: maximum 31
  - Ada and Java: no limit, and all are significant
  - C++: no limit, but implementors often impose one
Names (Continued)

- Connector Characters
  - Pascal, Modula-2, and FORTRAN 77 don't allow
  - Others do

- Case sensitivity

  - Disadvantage: readability (names that look alike are different)
    - Consider challenge in languages like where predefined names are mixed case - camel style (ex IndexOutOfBoundsException)
    - Easy typo problems vs. IDE autocomplete solves it

- CS Examples: Plenty - C, C++, JavaScript, and Java
- Non-CS Example: Eiffel
- Mixed up at times: PHP (user variables vs. functions)
Names (Continued)

**Special words**

- An aid to readability; used to delimit or separate statement clauses

  Def: A *keyword* is a word that is special only in certain contexts

  - Disadvantage: poor readability

  Def: A *reserved word* is a special word that cannot be used as a user-defined name
Variables

- A variable is an abstraction of a memory cell

- Variables can be characterized as a sextuple of attributes:
  
  \[ \text{name, address, value, type, lifetime, and scope} \]

  - name– considerations of length, case, character, etc.
  - Address – the memory address with which it is associated (not always easily knowable from code but it obviously implemented regardless)
Variables (Continued)

- A variable may have different addresses at different times during execution

- A variable may have different addresses at different places in a program

- If two variable names can be used to access the same memory location, they are called aliases (reminder: JS alias issue with reference types)

- Aliases are harmful to readability (program readers must remember all of them)
Variables (continued)

- **Type** - determines the range of values of variables and the set of operations that are defined for values of that type; in the case of floating point, type also determines the precision

- **Value** - the contents of the location with which the variable is associated

- **Abstract memory cell** - the physical cell or collection of cells associated with a variable
The Concept of Binding

- **The l-value** of a variable is its address
- **The r-value** of a variable is its value

- **Definition**: A *binding* is an association, such as between an attribute and an entity, or between an operation and a symbol

- **Definition**: Binding time is the time at which a binding takes place.
The Concept of Binding (Continued)

- Possible binding times:

1. Language design time--e.g., bind operator symbols to operations

2. Language implementation time--e.g., bind floating point type to a representation

3. Compile time--e.g., bind a variable to a type in C or Java

4. Load time--e.g., bind a FORTRAN 77 variable to a memory cell

5. Runtime--e.g., bind a nonstatic local variable to a memory cell
The Concept of Binding (Continued)

- **Definition**: A binding is *static* if it first occurs before run time and remains unchanged throughout program execution.

- **Definition**: A binding is *dynamic* if it first occurs during execution or can change during execution of the program.

- **Type Bindings**
  1. How is a type specified?
  2. When does the binding take place?

- If static, the type may be specified by either an explicit or an implicit declaration.

- **Definition**: An *explicit declaration* is a program statement used for declaring the types of variables.

- **Definition**: An *implicit declaration* is a default mechanism for specifying types of variables (the first appearance of the variable in the program).

Note: Even if you don’t need explicit declarations, it is generally better style to use them if possible.
The Concept of Binding (Continued)

FORTRAN, PL/I, BASIC, and Perl provide implicit declarations

*Advantage:* writability
*Disadvantage:* reliability

- *Dynamic Type Binding* (APL, JavaScript, SNOBOL)

- Specified through an assignment statement
e.g. JavaScript
  
  ```javascript
  list = [2, 4.33, 6, 8];  // array
  list = 17.3;  // Number
  ```

- *Advantage:* flexibility (generic code), easy writing
- *Disadvantages:*
  1. High cost (dynamic type checking and interpretation)
  2. Type error detection by the compiler is difficult
The Concept of Binding (Continued)

- Type Inferencing (ML, Miranda, and Haskell)
  - Rather than by assignment statement, types are determined from the context of the reference

- Storage Bindings & Lifetime

  Allocation - getting a cell from some pool of available cells
  Deallocation - putting a cell back into the pool

- Definition: The lifetime of a variable is the time during which it is bound to a particular memory cell
The Concept of Binding (Continued)

- *Categories of variables by lifetimes*

  1. *Static*—bound to memory cells before execution begins and remains bound to the same memory cell throughout execution.

    e.g. all FORTRAN 77 variables, C static variables

    *Advantages*: efficiency (direct addressing), history-sensitive subprogram support

    *Disadvantage*: lack of flexibility (no recursion for example)
The Concept of Binding (Continued)

2. *Stack-dynamic*--Storage bindings are created for variables when their declaration statements are elaborated.

   e.g. local variables in C subprograms and Java methods

**Advantages:**
- Allows recursion; conserves storage

**Disadvantages:**
- Overhead of allocation and deallocation
- Subprograms cannot be history sensitive
- Inefficient references (indirect addressing)
The Concept of Binding (continued)

3. *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`) and all objects in Java

*Advantages:* provides for dynamic storage management

*Disadvantages:* inefficient and unreliable, also the ideas required to use it properly can confuse some programmers
The Concept of Binding (continued)

4. Implicit heap-dynamic--Allocation and deallocation caused by assignment statements
   
e.g. all variables in APL; all strings and arrays in Perl and of course pretty much everything JavaScript!

   Note: Can hint at deallocation in JavaScript using delete, otherwise wait for garbage collection.

   **Advantage:** flexibility
   **Disadvantages:**
   - Inefficient, because all attributes are dynamic
   - Loss of error detection
Type Checking

- Generalize the concept of operands and operators to include subprograms and assignments

- Definition: Type checking is the activity of ensuring that the operands of an operator are of compatible types

- Definition: A compatible type is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler-generated code, to a legal type. This automatic conversion is called a coercion.

- Definition: A type error is the application of an operator to an operand of an inappropriate type
Type Checking (Continued)

- If all type bindings are static, nearly all type checking can be static

- If type bindings are dynamic, type checking must be dynamic

- *Definition:* A programming language is *strongly typed* if type errors are always detected
Strong Typing

- **Advantage of strong typing**: allows the detection of the misuses of variables that result in type errors

- **Examples**:
  1. FORTRAN 77 is not: parameters, EQUIVALENCE
  2. Pascal is not: variant records
  3. C and C++ are not: parameter type checking can be avoided; unions are not type checked
  4. Ada is, almost (UNCHECKED CONVERSION is loophole) - Java similar (Java is similar)

- Coercion rules strongly affect strong typing—they can weaken it considerably (C++ versus Ada)

- Although Java has just half the assignment coercions of C++, its strong typing is still less effective than that of Ada
Type Compatibility

- Our concern is primarily for structured types

- **Definition**: *Type compatibility by name* means the two variables have compatible types if they are in either the same declaration or in declarations that use the same type name

  - Easy to implement but highly restrictive:
  
  - Subranges of integer types are not compatible with integer types
  
  - Formal parameters must be the same type as their corresponding actual parameters (Pascal)

- **Definition**: *Type compatibility by structure* means that two variables have compatible types if their types have identical structures

  - More flexible, but harder to implement
Scope

- **Definition**: The *scope* of a variable is the range of statements over which it is visible.

- **Definition**: 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.
Scope (Continued)

1. Static scope

- Based on program text

- To connect a name reference to a variable, you (or the compiler) must find the declaration

- Search process: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name

- Enclosing static scopes (to a specific scope) are called its static ancestors; the nearest static ancestor is called a static parent
Scope (continued)

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

- C++ and Ada allow access to these "hidden" variables
  - In Ada:  unit.name
  - In C++: class_name::name

- Blocks
  - A method of creating static scopes inside program units--from ALGOL 60
Scope (continued)

- Examples:

  C and C++:  
  ```
  for (...) {
    int index;
    ...
  }
  ```

  Ada:  
  ```
  declare LCL : FLOAT;
  begin
    ...
  end
  ```
Scope (continued)

- Evaluation of Static Scoping

Consider the example:
Assume MAIN calls A and B
  A calls C and D
  B calls A and E

Diagram:
```
   MAIN
    A
     C
     D
    B
     E

   MAIN
   A
   B
   C
   D
   E
```
Scope (continued)
Scope (continued)

- Suppose the spec is changed so that D must now access some data in B

- Solutions:

  1. Put D in B (but then C can no longer call it and D cannot access A's variables)

  2. Move the data from B that D needs to MAIN (but then all procedures can access them)

- Same problem for procedure access!

- Overall: static scoping often encourages many globals
Scope (continued)

2. 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 (continued)

Example:

MAIN
- declaration of x
SUB1
- declaration of x -
...  
call SUB2
...

SUB2
...
- reference to x -
...

...

call SUB1
...

MAIN calls SUB1
SUB1 calls SUB2
SUB2 uses x
Scope (continued)

Static scoping - reference to x is to MAIN's x

  Dynamic scoping - reference to x is to SUB1's x

- Evaluation of Dynamic Scoping:
  - Advantage: convenience
  - Disadvantage: poor readability

Scope and Lifetime

- Scope and lifetime are sometimes closely related, but actually are different concepts
  - Consider a static variable in a C or C++ function
Referencing Environments

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

- In a static scoped language, that is the local variables plus all of the visible variables in all of the enclosing scopes.

- 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.

- See examples in book 237-239.
Named Constants

*Definition:* A named constant is a variable that is bound to a value only when it is bound to storage

- **Advantages:** readability and modifiability
- Used to parameterize programs

- The binding of values to named constants can be either static (called manifest constants) or dynamic

- **Languages:**
  - *Pascal:* literals only
  - *FORTRAN 90:* constant-valued expressions
  - *Ada, C++, and Java:* expressions of any kind
Variable Initialization

- *Definition*: The binding of a variable to a value at the time it is bound to storage is called *initialization*

- Initialization is often done on the declaration statement

  e.g., Ada
  
  ```plaintext
  SUM : FLOAT := 0.0;
  ```

  e.g. JavaScript
  
  ```plaintext
  var sum = 0;
  ```