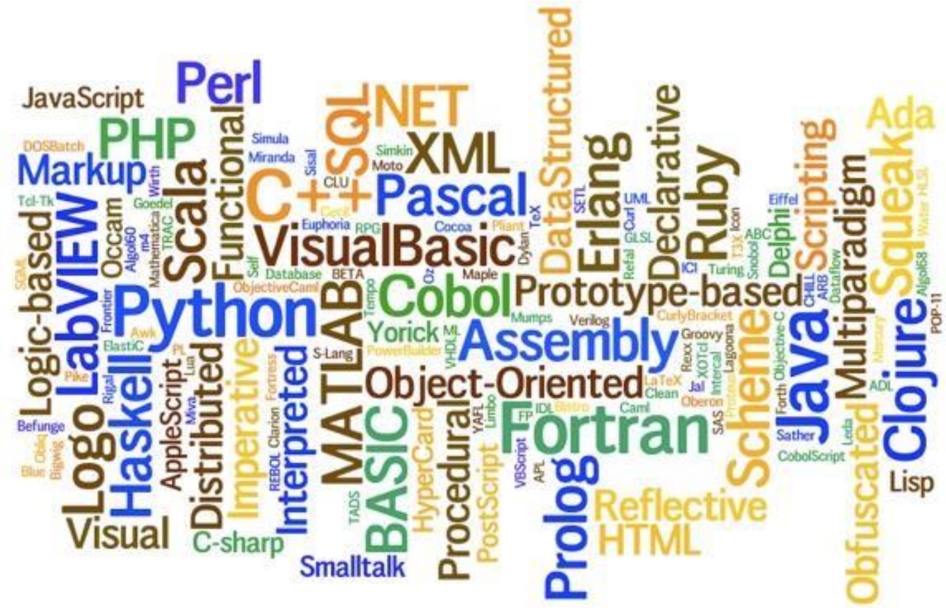


Programming Languages
Names, Bindings, and Scopes

Programming Languages
Module 5

Dr. Tamer ABUHMED
College of Computing

SUNG KYUN KWAN UNIVERSITY

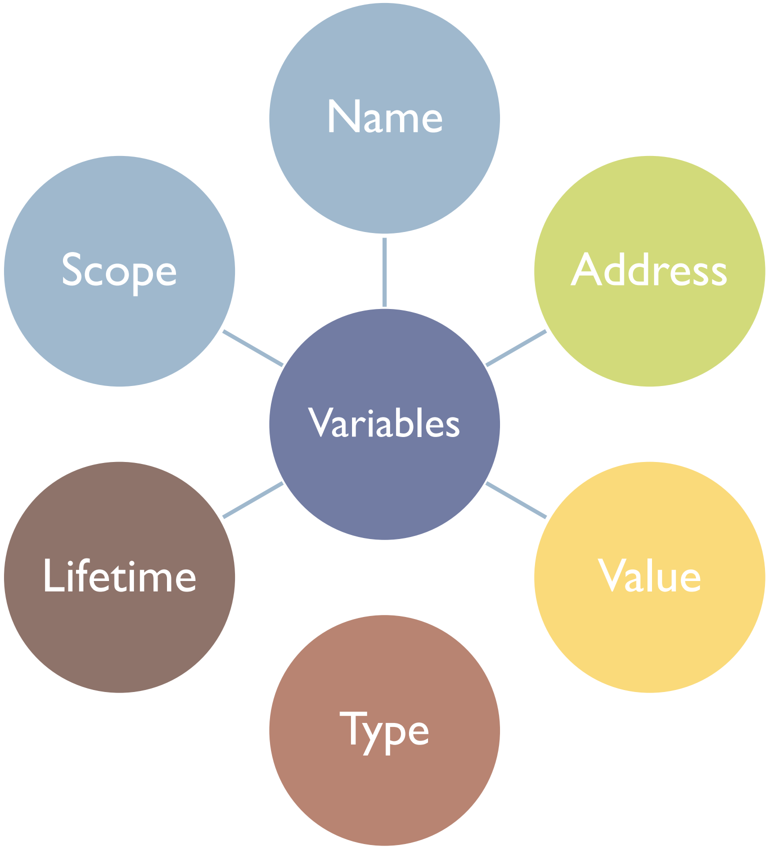
Chapter 5 Topics

- Introduction
- Names
- Variables
- The Concept of Binding
- Scope
- Scope and Lifetime
- Referencing Environments
- Named Constants

Introduction

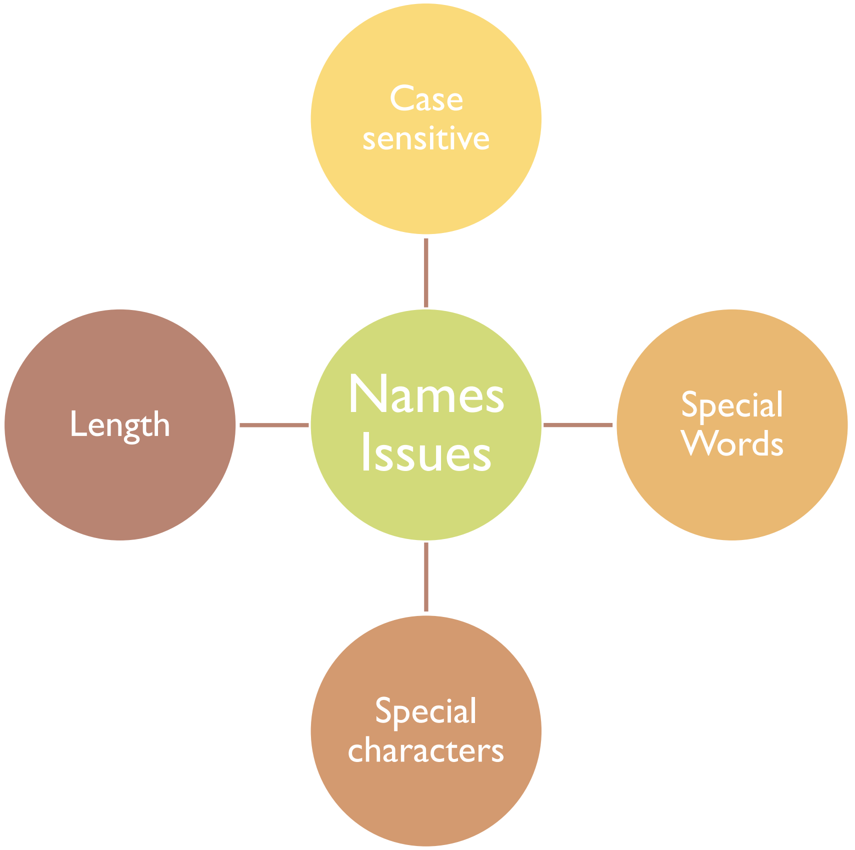
- Imperative languages are abstractions of von Neumann architecture
  - Memory
  - Processor

- Variables are characterized by attributes
  - To design a type, must consider scope, lifetime, type checking, initialization, and type compatibility



Names

- *name is a* fundamental attribute of variables
- The term *identifier* is often used interchangeably with *name*
- Design issues for names:
  - Are names case sensitive?
  - Are special words reserved words or keywords?



Names (continued)

- Length
  - earliest programming languages used single-character names
  - If too short, they cannot be connotative
  - Language examples:
    - FORTRAN 95: maximum of 31
    - C99: no limit but only the first 63 are significant; also, external names are limited to a maximum of 31
    - C#, Ada, and Java: no limit, and all are significant
    - C++: no limit, but implementers often impose one

Names (continued)

- Special characters
  - PHP: all variable names must begin with dollar signs
  - Perl: all variable names begin with special characters, which specify the variable's type

Types of Perl Variable

- Different types of variables start with a different symbol
  - Scalar variables start with $
  - Array variables start with @
  - Hash variables start with %

  - Ruby: variable names that begin with `@` are instance variables; those that begin with `@@` are class variables

Name Begins With	Variable Scope
$	A global variable
@	An instance variable
[a-z] or _	A local variable
[A-Z]	A constant
@@	A class variable

Names (continued)

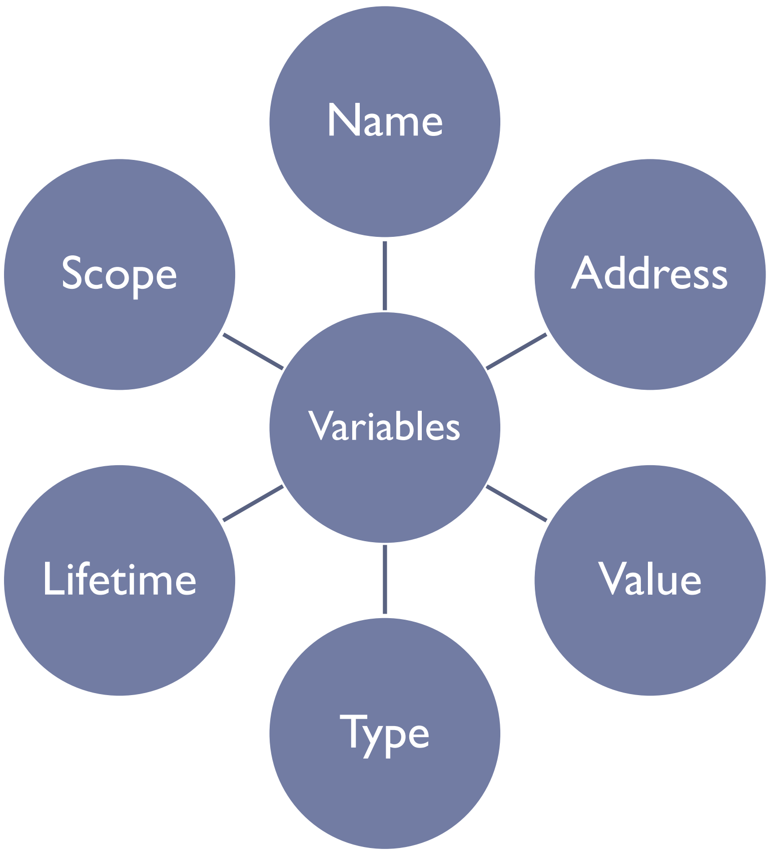
- Case sensitivity
  - Disadvantage: readability (names that look alike are different)
    - Names in the C-based languages are case sensitive
    - Names in others are not
    - Worse in C++, Java, and C# because predefined names are mixed case (e.g. `IndexOutOfBoundsException`)
  - "Case sensitive" is always better to reduce ambiguity

Names (continued)

- Special words
  - An aid to readability; used to delimit or separate statement clauses
    - A *keyword* is a word that is special only in certain contexts, e.g., in Fortran
      - `Real VarName` (`Real` *is a data type followed with a name, therefore* `Real` *is a keyword)*
      - `Real = 3.4` (*`Real` is a variable)*
  - A *reserved word* is a special word that cannot be used as a user-defined name
  - Potential problem with reserved words: If there are too many, many collisions occur (e.g., COBOL has 300 reserved words!)

Variables

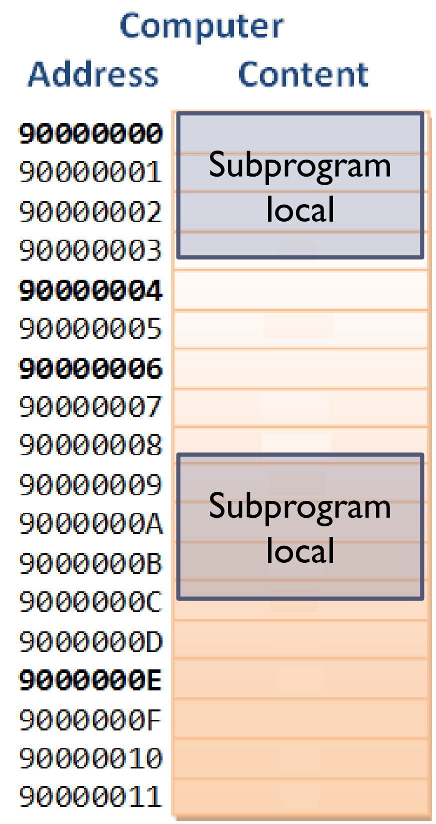
- A variable is an abstraction of a memory cell
- Variables can be characterized as a sextuple of attributes:
  - Name
  - Address
  - Value
  - Type
  - Lifetime
  - Scope



Variables Attributes

- **Name** - not all variables have them
- **Address** - the memory address with which it is associated
  - A variable may have different addresses at different times during execution

```
Subprogram(){
    int local;
}
Main (){
    .
    Subprogram();  1
    .
    Subprogram();  2
}
```



Variable Address

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

Student x = new Student ( );

.

.

.

.

x = new Student( );

Computer

	Address	Content
	90000000	
x	90000001	Student ( )
	90000002	
	90000003	
	90000004	
	90000005	
x	90000006	Student ( )
	90000007	
	90000008	

- If two variable names can be used to access the same memory location, they are called aliases
- Aliases are created via pointers, reference variables, C and C++ unions
- Aliases are harmful to readability (program readers must remember all of them)

Aliases

- Java (aliasing available with objects)

```
Rectangle box1 = new Rectangle (0, 0);
Rectangle box2 = box1;
```

- C++

```
int main() {
  int a = 0;
  int &b = a;      //alias
  int *c = &a;    //alias

  cout << a << b << *c << endl;
  a = 7;
  cout << a << b << *c;
  return 0;
}
```

- Python (aliasing applied directly)

```
# Assign a value to a new variable
a = 5
# Create an alias identifier for this variable
b = a
# Observe how they refer to the same variable!
print (id(a), id(b))
# Create another alias
c = b
# Now assign a new value to b!
b = 3
# And observe how a and c are still
#the same variable # But b is not
print (a,b,c)
print (id(a),id(b),id(c))
```

In python, assignment of a value to the alias identifier will break the alias, and create a separate variable by the same name instead!

Variables Attributes (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

  - The l-value of a variable is its address

  - The r-value of a variable is its value

```
x = 7
```

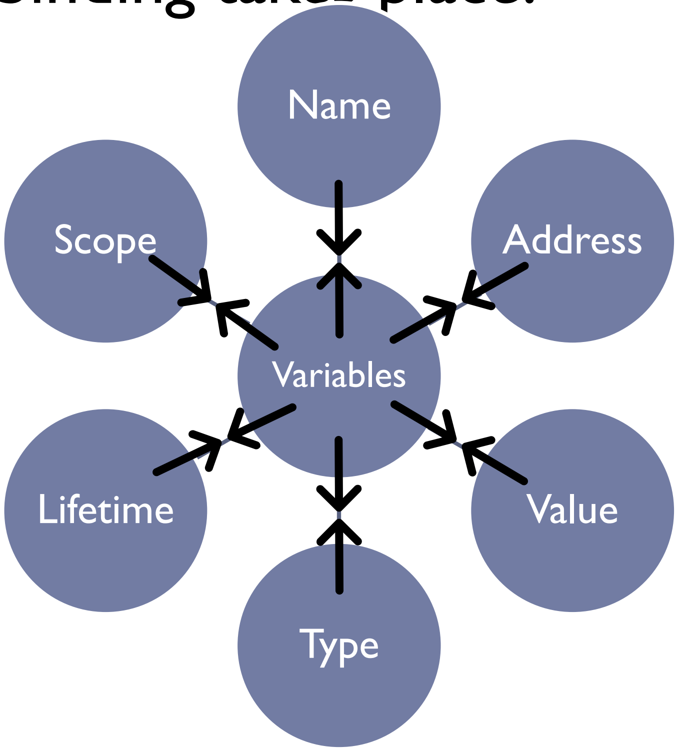
l-value   r-value

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

The Concept of Binding

A *binding* is an association between an entity and an attribute, such as between a variable and its type or value, or between an operation and a symbol

- *Binding time* is the time at which a binding takes place.



Possible Binding Times

- Language design time -- bind operator symbols to operations
- Language implementation time-- bind floating point type to a representation
- Compile time -- bind a variable to a type in C or Java
- Load time -- bind a C or C++ `static` variable to a memory cell)
- Runtime -- bind a nonstatic local variable to a memory cell

Example of bindings and their binding times

count = count + 5;

- The type of count is bound at compile time.
- The set of possible values of count is bound at compiler design time.
- The meaning of the operator symbol + is bound at compile time, when the types of its operands have been determined.
- The internal representation of the literal 5 is bound at compiler design time.
- The value of count is bound at execution time with this statement.

Binding Time: Static and Dynamic

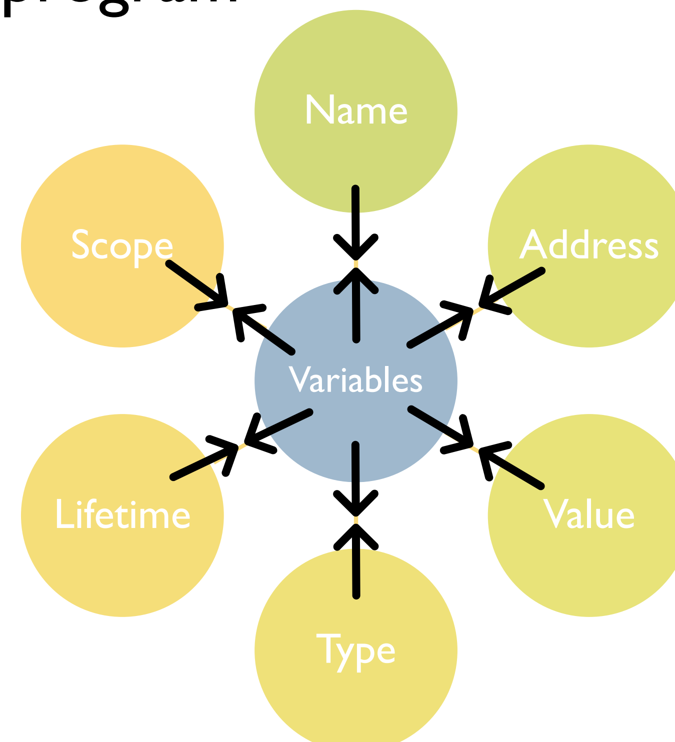
- A binding is *static* if it first occurs before run time and remains unchanged throughout program execution.
- A binding is *dynamic* if it first occurs during execution or can change during execution of the program

Static Binding

Before run time

dynamic Binding

After run time



Static and Dynamic Binding Variable Type

Overridden instance methods are bound at run time; and this kind of binding depends on the instance object type.

For example in java:

```
public class Parent {
public void writeName() {
   System.out.println("Parent");
     }
}

public class Child extends Parent {
   public void writeName() {
      System.out.println("Child");
   }

public static void main(String [] args) {
   Parent p = new Child();
   p.writeName();
    }
}
```

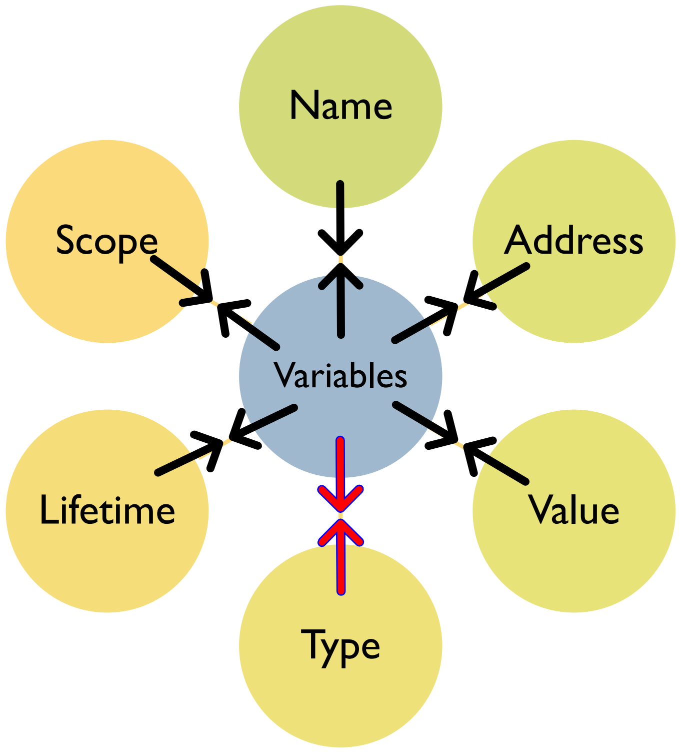
The instance variables, static variables, static overridden methods, and overloaded methods are all bound at compile time; and this kind of binding depends on the type of the reference variable and not on the object.

```
public class Parent {
public static String age = "50";
public String hairColor = "grey";
public void writeName() {
System.out.println("Parent"); }
   }

public class Child extends Parent {
  public static String age = "20";
  public String hairColor = "brown";
  public void writeName() {
     System.out.println("Child");  }
   public void writeName(String order) {
     System.out.println(order + " Child");  }

public static void main(String [] args) {
  Parent p = new Child();
  System.out.println("age: " + p.age);
  System.out.println("hairColor: " + p.hairColor);
  Child c = new Child();
   c.writeName("first");
} }
```

```
age: 50
hairColor: grey
first Child
```

Variable Type Binding

- How is a type specified?
- When does the binding take place?
- If static, the type may be specified by either an explicit or an implicit declaration



Explicit/Implicit Declaration

- An ***explicit*** *declaration* is a program statement used for declaring the types of variables
- An ***implicit*** *declaration* is a default mechanism for specifying types of variables through default conventions, rather than declaration statements
- Fortran, BASIC, Perl, Ruby, JavaScript, and PHP provide implicit declarations (Fortran has both explicit and implicit)
  - Advantage: writability (a minor convenience)
  - Disadvantage: reliability (less trouble with Perl)

Explicit/Implicit Declaration (continued)

- Some languages use type inferencing to determine types of variables (context)
  - C# - a variable can be declared with **`var`** and an initial value. The initial value sets the type

```
var sum = 0;
var total = 0.0;
var name = "Fred";
```

  - Visual BASIC 9.0+, ML, Haskell, F#, and Go use type inferencing. The context of the appearance of a variable determines its type

Dynamic Type Binding

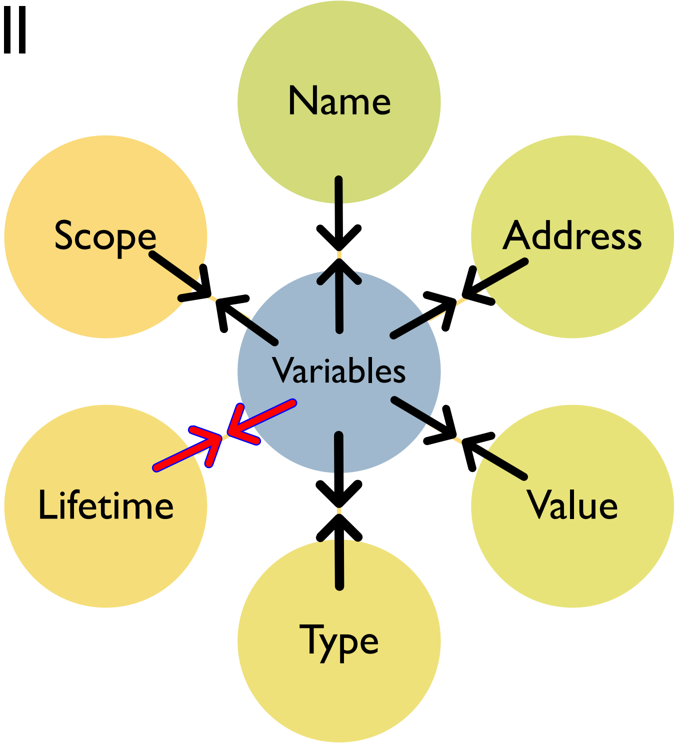
- Dynamic Type Binding (JavaScript, Python, Ruby, PHP, and C# (limited))
- Specified through an assignment statement
  e.g., JavaScript

```
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 (continued)

- Storage Bindings & Lifetime
  - Allocation - getting a cell from some pool of available cells
  - Deallocation - putting a cell back into the pool
- The lifetime of a variable is the time during which it is bound to a particular memory cell



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 in functions
  - 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
  - local variables in C subprograms (not declared `static`) and Java methods
- Advantage: allows recursion; conserves storage
- Disadvantages:
  - Overhead of allocation and deallocation
  - Subprograms cannot be history sensitive
  - Inefficient references (indirect addressing)

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

Summary

- Introduction
- Names
- Variables
- The Concept of Binding
- Scope
- Scope and Lifetime
- Referencing Environments
- Named Constants