Modules & Classes

- development of large systems
  - decomposition into loosely-dependent units (modules)
  - separate compilation
- two views
  - module as physical decomposition
    - Modula-2, Ada
  - module as logical decomposition
    - abstract data types
    - CLU, Alphard
    - module as class with objects implementing ADTs
      - O-O languages

- Ada
  - for DoD, by competition
  - too many languages & dialects
  - mission-critical, embedded systems
  - Pascal based
  - packages for decomposition
    - also as libraries
  - strongly-typed
  - tasking for parallelism
  - interrupt and exception handling
  - compiler validation
  - large and complex
  - O-O added in 1995
- Modula-2
  - Pascal based
  - modules as decomposition mechanism
  - much simpler than Ada

- Smalltalk
  - Dynabook project
  - integration of language and development environment
  - GUIs
  - influenced by LISP
  - pure O-O, everything is a class
- Eiffel
  - for large robust systems
  - Modula-2 based
  - strongly-typed
  - pre-, post-conditions and invariants
- C++
  - O-O extension to C
  - class is extension of struct
  - stronger type checking
  - all problems of C plus problems of hybrid

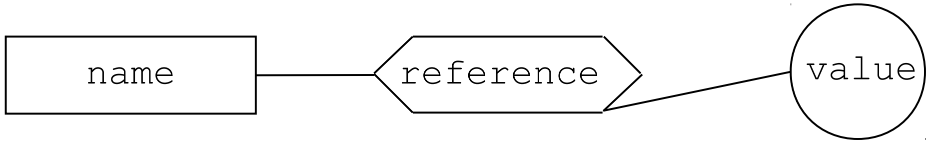
- Delphi
  - O-O extension to Pascal
  - visual development environment
    - RAD
  - Windows development
- Java
  - originally for embedded consumer electronics
  - retooled for WWW
  - comprehensive standard library
    - language extension
    - Java APIs
    - semi-independent
  - interpreted
    - platform independence
    - java bytecode and JVM
  - pure O-O
  - simpler and less error-prone than C++

Functional and Logic Languages

- functional
  - programming by functional composition
  - no side-effects
  - FP, ML, pure-LISP, Haskell, Gofer
- logic
  - Prolog
    - designed for AI
    - set of facts and rules and performs induction
    - little explicit control over program flow
    - hard to control efficiency
  - expert systems

Names, References, Values & Types

- data items have a value and a type
  - type determines set of operations
- variables store values
  - variables have a name, a storage location and attributes
- binding diagram (Barron)



- identifiers
  - allowable characters
  - length
  - case sensitivity
  - naming conventions

Binding

- association of one property with another
  - choice of a property from a set of possible properties
- name-declaration binding
  - connection between identifier and declaration
- name-type binding
  - choose type for a variable identified by a name
- declaration-reference binding
  - storage allocation
- reference-value binding
  - storing value of a variable

Binding Time

- time at which binding occurs
- alternatives
  - compile-time (e.g. name-type binding in Java)
  - load-time (e.g. declaration-reference binding for non-local variables in C)
  - run-time
    - at block entry (e.g. declaration-reference binding for local variables in Java)
    - at statement execution (e.g. reference-value binding for variables)
- early → early error detection and efficient execution
- late → flexibility

Name-Declaration Binding

- declaration
  - explicit vs implicit
- binding
  - connection between name and its declaration
  - scope rules
- at compile-time
  - can be determined from static program text
  - static scope
  - block structure
    - block, scope, hole-in-scope
    - local variable, non-local variable, global variable
    - e.g. Pascal, Java
- at run-time
  - dynamic scope

Static Scope Example - Pascal

```
program Example(input, output)
    var x, y : Real;
    procedure op1;
        var y, z : Integer;
    begin
        ...
        y := 34;
        x := 27.4;
        ...
    end{op1};
begin
    ...
    x := 3.768;
    y := x;
    ...
end{Example}.
```

Static Scope Example - Java

```
class Example {
    private double x, y;
    public void op1() {
        int y, z;
        y := 34;
        x := 27.4;
        ...
    } // op1
    public void op2() {
        x := 3.768;
        y := x;
        ...
    } // op2
} // Example
```

Dynamic Scope Example (Pascal syntax)

(that is, Pascal-*like*)

```
     program Dynamic(input, output);
         var x : Integer;
 3,5     procedure a;
         begin
             ... write(x); ...
         end{a};
 4       procedure b;
             var x : Real;
         begin
             ... x := 2.0; ... a; ...
         end{b};
 1   begin
 2,6     ... x := 1; ... a; ... b; ... a; ...
     end{Dynamic}.
```

1)The main program starts here.

2)Since we're using dynamic scoping, we have a binding stack for x.
Our global x starts as 1.

3)When a is first called, it looks at the top binding for x on the stack and writes 1.

4)When b is called, it puts a new x onto the stack (2.0).

5)When b calls a, since we haven't left b's control flow, a looks at the top of the binding stack and sees 2.0.

6)Now that b is finished, we leave its context and remove that x from the stack. When we invoke a again, it once again reports x as being 1.

Note that means you can't examine procedures in a vacuum.
You can't immediately tell the scope of x in a just by looking at a.

Name-Type Binding

- specification of type for variable
- statically typed
  - compile-time
  - advantages
    - reliable, efficient, understandable
  - strongly-typed vs firmly-typed vs weakly-typed vs typeless
- dynamically typed
  - run-time
    - variables don't have types, values do
  - advantage
    - flexibility
  - type discovery
- polymorphic variables in object-oriented languages
  - only subclass values assignable
    - e.g. `Stack s=new LnkStack();`
  - static type checking with flexibility of dynamic typing
- type inference (ML, Haskell)

Declaration-Reference Binding

- storage allocation (address=reference=pointer)
- lifetime or extent
- scope vs extent
  - name-declaration binding and declaration-reference binding
- local variables
  - allocated at block entry, freed at exit
- instance variables
  - allocated at statement execution (new) as part of object
- global variables
  - allocated at load time
- `static` and `own` variables
  - procedure memory
  - allocated at load time, but local scope
  - vs instance variables
- class variables (`static` in Java)
  - allocated at load time, one occurrence per class

Reference-Value Binding

- storing a value
- run-time at input, assignment or parameter passing
- assignment
  - 3 bindings involved:
    - name-declaration, declaration-reference, reference-value
  - dereferencing
  - l-value vs r-value
- language without an assignment statement ⇒ it may have immediate name-value binding
  - pure functional languages
- uninitialized variables
- constants
  - name-value binding at compile time
  - Pascal vs Ada vs Java

Types

- specify set of values and set of operations
- everything represented as bit strings, type gives interpretation
- kinds: scalar, structured and reference
- scalar
  - single values
  - numeric, logical, character
  - discrete (ordinal), e.g. `int`
    - unique predecessor
  - non-discrete, e.g. `double`
  - built-in or primitive types
    - defined in language
    - in Java not objects
  - programmer-defined types
    - type declarations (e.g. Pascal, Ada)
    - class declarations (e.g. Java)

Specifying Types

- typically in variable declarations
- type declarations
  - associate a name with a set of type attributes
  - modifiability, type-checking
- derived types vs subtypes
  - same representation but different meaning (derived)
  - subset of the values, but same meaning (subtype)
  - type checking
  - name equivalence (e.g. Ada) vs structural equivalence (e.g. C)
    - type compatibility: name equivalence requires explicit type conversion
    - To Ada, a rose by any other name... stops being a rose.
    - C is more forgiving of structures that line up well
  - abstract data types
    - modules contain type declaration
    - class is type declaration

Numeric Types

- machine representation vs standard model
  - efficiency vs compatibility
- operators
  - overloading: effect depends on type of operands
    - `2+3` integer addition
    - `2.3+4.5` floating-point addition
  - exponentiation?
  - relational operators: `<`, `>=`, etc.
  - equality operators: `==` or `=`
  - inequality operators: `!=` or `<>` or `/=`
- integer vs fixed-point vs floating-point
  - impact on equality

Integer Types

- uses
  - as models of Integer in Mathematics vs counting
- two's complement
  - range
- larger and smaller ranges
  - C – `char`, `short`, `int`, `long`
  - Java – `byte`, `short`, `int`, `long`
  - Pascal – subranges
  - Ada – `Short_Integer`, `Integer`, `Long_Integer` and subranges
- fixed-point types
  - COBOL & PL/I
    - for currency values

Floating-Point Types

- mathematical and scientific calculations
- approximations: range, precision determined by implementation
  - equality
- representation: sign, exponent, mantissa
  - real number = sign * mantissa * 2^exponent
- double precision
- Ada floating-point type declarations
- complex numbers
  - pair of floats
  - class definition

Logical Types

- truth values
- often called Boolean
- representation
  - bit vs byte
- literals
- C & C++
  - use `int` for logical values
  - 0 = false, non-zero = true
  - assignment vs equality test problem
    - e.g. `if(a=3)` is always true!
- operations
  - and, or, not
  - short circuit

Character Types

- operations
  - relational and equality
- coding scheme
  - unspecified, e.g. Pascal
  - ASCII, e.g. Ada
  - Unicode, e.g. Java
- `char` in C
  - 1 byte integer
  - signed & unsigned
- strings
  - array of char, e.g. Pascal, C, Ada
  - library class, e.g. Java
  - built-in type, e.g. Perl, SNOBOL

Enumeration Types

- named values
  - mapped to integers but not integers
- e.g. days of the week
  - Pascal:

```
type Days = (Sunday, Monday, Tuesday,
    Wednesday, Thursday, Friday, Saturday)
```

  - C/C++:

```
enum Days {Sunday, Monday, Tuesday, Wednesday,
          Thursday, Friday, Saturday};
```

- operations
  - relational operators
  - `pred`, `succ`

- overloading enumerands
  - e.g. Ada

```
type Light is (red, amber, green);
type Flag is (red, white);
```

  - not allowed in Pascal, C, C++
- boolean as enumerated type (Pascal, Ada)
- char as enumerated type (Ada)
  - allows other character sets to be defined, e.g.

```
type Hex is ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A',
  'B', 'C', 'D', 'E', 'F');
```

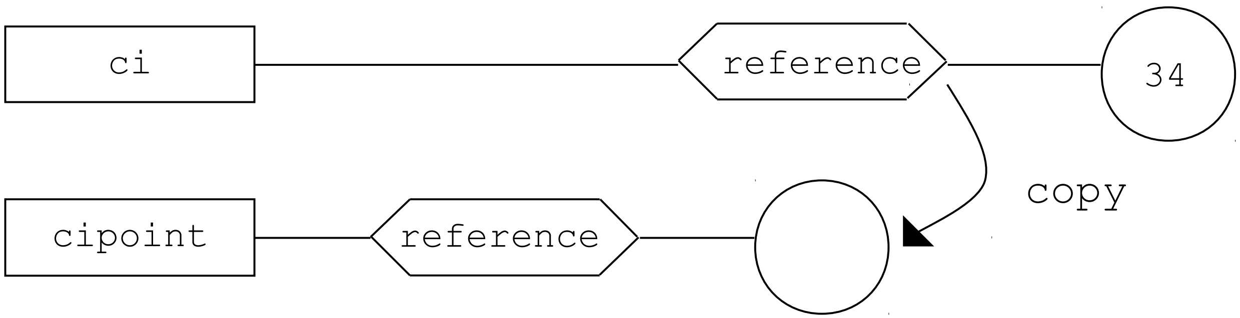
- enumeration type added to Java in 1.5, e.g.

```
public enum MainMenu {FILE, EDIT, FORMAT, VIEW};
```

  - is typesafe
  - `MainMenu` is a class implementing `Comparable<MainMenu>` and `Serializable`
  - `values()` returns an array containing the values of the enumeration
  - `valueOf(String)` returns the enumeration value with that name
  - `equals`, `hashCode`, `toString` and `compateTo` are implemented

Reference Types

- value is reference (pointer, address)
- referenced type is specified
  - type checking
  - e.g. C/C++ : `int ci, *cipoint;`
- addressof operator `cipoint = &ci;`
  - danger - dangling pointer



- dereferencing operator `x = *cipoint;`

- pointer arithmetic
    - C, C++ - yes   e.g. `cipoint++;`
    - Pascal, Ada, Java - no
- null pointer - `null`, `nil`
- low level programming
- aliases
    - addressof operator

      e.g. `cipoint = &ci;`

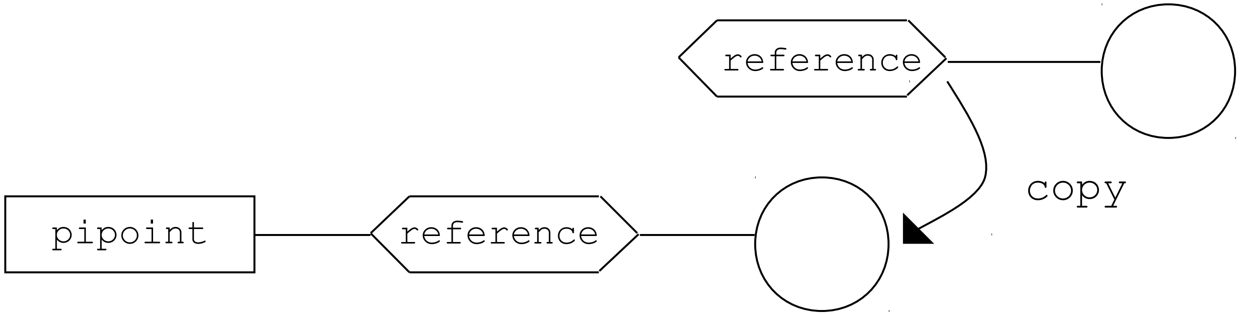
      `*cipoint` can be used in the same way as `ci`

      e.g. `x = ci+27` is equivalent to `x = *cipoint+27`
    - parameter passing
    - danger

Dynamic Variables

- allocation at statement execution
- Pascal

```
type Integerpt = ^Integer;
var pipoint, another: Integerpt;
    pi: Integer;
new(pipoint);
```




```
pipoint^ = 17;
```

Dynamic Variables

- C

```
int *cipoint;
cipoint = (int *) malloc(sizeof(int));
```

- C++

```
cipoint = new int;
*cipoint = 17;
```

- C++ vs Java

C++: `Stype *spoint = new Stype;`

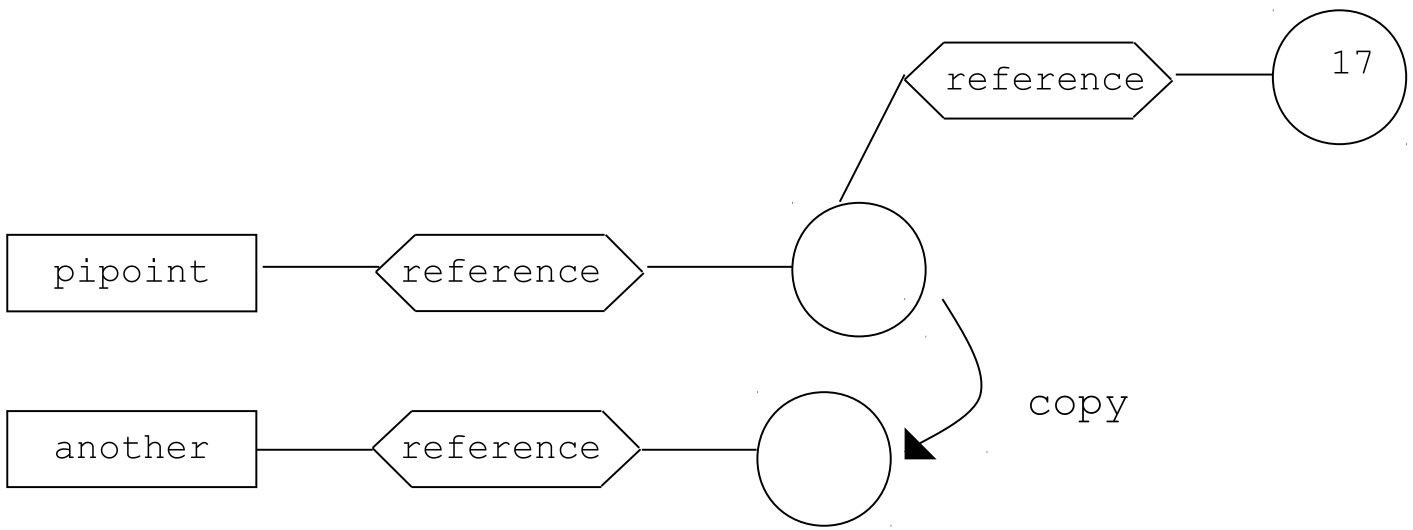
Java: `Stype spoint = new Stype();`

```
    Stype s;
```

- explicit pointer variable declaration and explicit dereferencing operator
  - C, Pascal - yes
  - Java - no

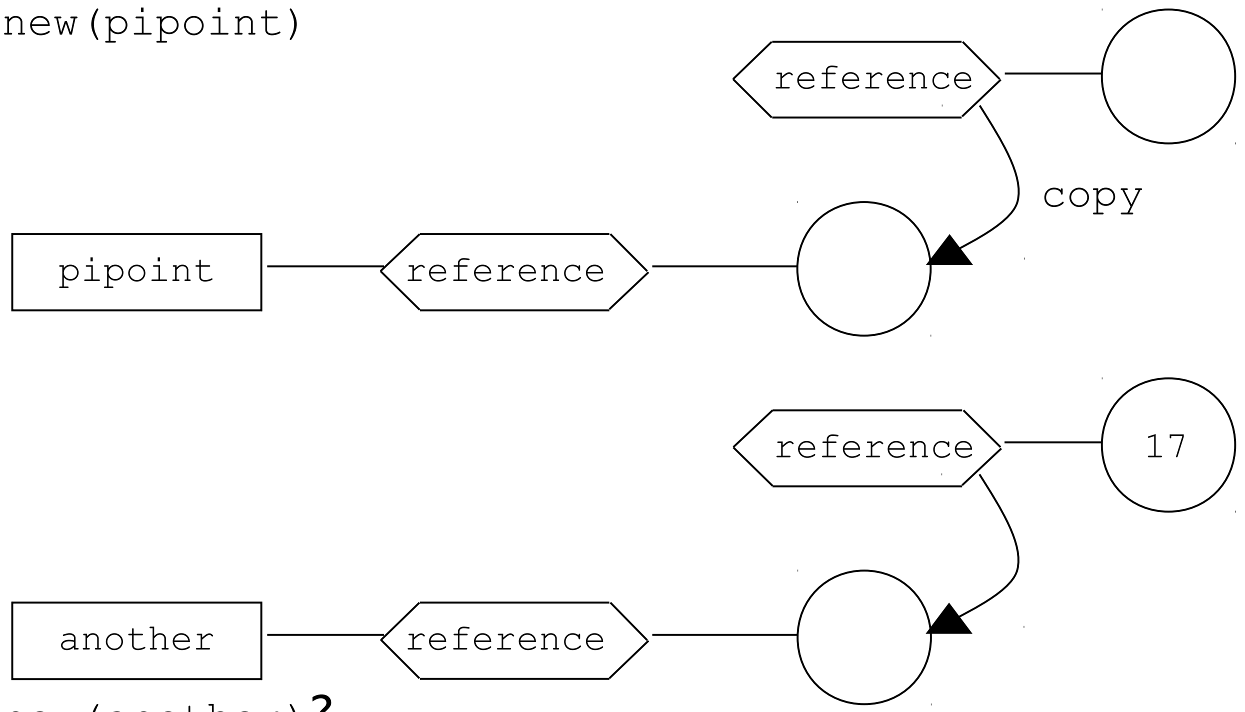
Aliasing of References

```
another := pipoint;
```



Garbage Creation

after `new(pipoint)`



after `new(another)`?

Garbage

- explicit deallocation
  - `dispose` (Pascal), `delete` (C++)

    e.g. `dispose(pipoint);`
  - knowing what to dispose
    - memory leaks
  - dangling reference
- garbage collection
  - Java
  - efficiency