Properties of an identifier (and the object it represents) may be set at

- **Compile-time**

  These are *static* properties as they do not change during execution. Examples include the type of a variable, the value of a constant, the initial value of a variable, or the body of a function.

- **Run-time**

  These are *dynamic* properties. Examples include the value of a variable, the lifetime of a heap object, the value of a function's parameter, the number of times a while loop iterates, etc.

Example:

In Fortran

- The scope of an identifier is the whole program or subprogram.
- Each identifier may be declared only once.
- Variable declarations may be implicit. (Using an identifier implicitly declares it as a variable.)
- The lifetime of data objects is the whole program.

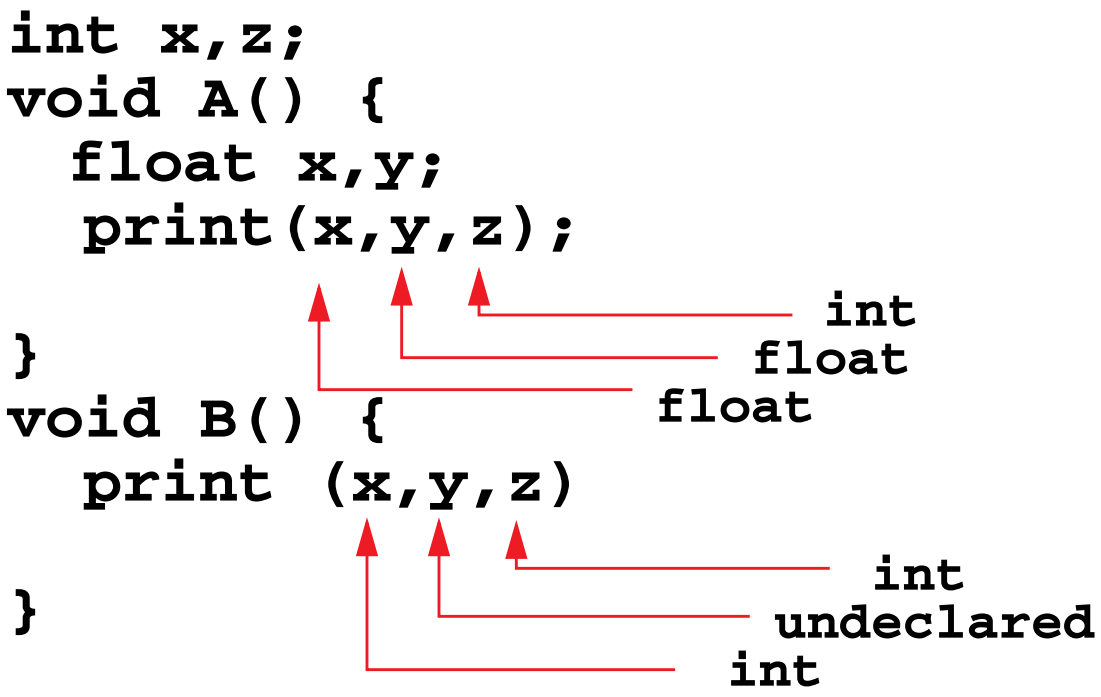
Block Structured Languages

- Include Algol 60, Pascal, C and Java.
- Identifiers may have a non-global scope. Declarations may be *local* to a class, subprogram or block.
- Scopes may *nest*, with declarations propagating to inner (contained) scopes.
- The lexically *nearest* declaration of an identifier is bound to uses of that identifier.

Binding of an identifier to its corresponding declaration is usually static (also called lexical), though dynamic binding is also possible.

Static binding is done prior to execution—at compile-time.

Example (drawn from C):



Block Structure Concepts

- Nested Visibility

  No access to identifiers outside their scope.

- Nearest Declaration Applies

  Static name scoping.

- Automatic Allocation and Deallocation of Locals

  Lifetime of data objects is bound to the scope of the Identifiers that denote them.

Variations in these rules of name scoping are possible.

For example, in Java, the lifetime of all class objects is from the time of their creation (via **`new`**) to the last visible reference to them.

Thus

```
... Object o;...
```

creates an *object reference* but does not allocate any memory space for `o`.

You need

```
... Object o = new Object(); ...
```

to actually create memory space for `o`.

Dynamic Scoping

An alternative to static scoping is *dynamic scoping*, which was used in early Lisp dialects (but not in Scheme, which is statically scoped).

Under dynamic scoping, identifiers are bound to the dynamically closest declaration of the identifier. Thus if an identifier is not locally declared, the call chain (sequence of callers) is examined to find a matching declaration.

Example:

```
int x;
void print() {
  write(x); }
main () {
  bool x;
  print();
}
```

Under static scoping the **x** written in **print** is the lexically closest declaration of **x**, which is as an **int**.

Under dynamic scoping, since **print** has no local declaration of **x**, **print**'s caller is examined. Since **main** calls **print**, and it has a declaration of **x** as a **bool**, that declaration is used.

Dynamic scoping makes type checking and variable access harder and more costly than static scoping. (Why?)

However, dynamic scoping does allow a notion of an "extended scope" in which declarations extend to subprograms called within that scope.

Though dynamic scoping may seen a bit bizarre, it is closely related to *virtual functions* used in C++ and Java.

Virtual Functions

A function declared in a class, C, may be redeclared in a class derived from C. Moreover, for uniformity of redeclaration, it is important that *all* calls, including those in methods within C, use the new declaration.

Example:

```
class C {
  void DoIt()(PrintIt();}
  void PrintIt()
     {println("C rules!");}
}
class D extends C {
  void PrintIt()
      {println("D rules!");}
  void TestIt() {DoIt();}
}
D dvar = new D();
dvar.TestIt();
D rules! is printed.
```

Scope vs. Lifetime

It is usually required that the lifetime of a run-time object at least cover the scope of the identifier. That is, whenever you can access an identifier, the run-time object it denotes better exist.

But,

it is possible to have a run-time object's lifetime *exceed* the scope of its identifier. An example of this is *static* or *own* variables.

In C:

```
void p() {
  static int i = 0;
  print(i++);
}
```

Each call to `p` prints a different value of `i` (0, 1, ...) Variable `i` retains its value across calls.

Some languages allow an explicit binding of an identifier for a fixed scope:

```
Let
 id = val
in
  statements
end;
```

```
{
  type id = val;
   statements
}
```

A declaration may appear wherever a statement or expression is allowed. Limited scopes enhance readability.

Structs vs. Blocks

Many programming languages, including C, C++, C#, Pascal and Ada, have a notion of grouping data together into *structs* or *records*.

For example:

```
struct complex { float re, im; }
```

There is also the notion of grouping statements and declarations into *blocks*:

```
{ float re, im;
  re = 0.0; im = 1.0;
}
```

Blocks and structs look similar, but there are significant differences:

Structs are *data*,

- As originally designed, structs contain only data (no functions or methods).
- Structs can be dynamically created, in any number, and included in other data structures (e.g., in an array of structs).
- All fields in a struct are visible outside the struct.

Blocks are *code*,

- They can contain both code and data.
- Blocks *can't* be dynamically created during execution; they are "built into" a program.
- Locals in a block *aren't* visible outside the block.

By adding functions and initialization code to structs, we get *classes*—a nice blend of structs and blocks.

For example:

```
class complex{
  float re, im;
  complex (float v1, float v2){
    re = v1; im = v2; }
 }
```

Classes

- Class objects can be created as needed, in any number, and included in other data structure.
- They include both data (fields) and functions (methods).
- They include mechanisms to initialize themselves (constructors) and to finalize themselves (destructors).
- They allow controlled access to members (private and public declarations).

Type Equivalence in Classes

In C, C++ and Java, instances of the same struct or class are type-equivalent, and mutually assignable.

For example:

```
class MyClass { ... }
MyClass v1, v2;
v1 = v2; // Assignment is OK
```

We expect to be able to assign values of the same type, including class objects.

However, sometimes a class models a data object whose size or shape is set upon creation (in a constructor).

Then we may *not* want assignment to be allowed.

```
class Point {
   int dimensions;
   float coordinates[];
   Point () {
     dimensions = 2;
     coordinates = new float[2];
   }
   Point (int d) {
     dimensions = d;
     coordinates = new float[d];
   }
 }
 Point plane = new Point();
 Point solid  = new Point(3);
 plane = solid; //OK in Java
```

This assignment is allowed, even though the two objects represent points in different dimensions.

Subtypes

In C++, C# and Java we can create *subclasses*—new classes derived from an existing class.

We can use subclasses to create new data objects that are similar (since they are based on a common parent), but still *type-inequivalent*.

Example:

```
class Point2 extends Point {
   Point2() {super(2); }
}
class Point3 extends Point {
   Point3() {super(3); }
}
Point2 plane  = new Point2();
Point3 solid  = new Point3();
plane = solid; //Illegal in Java
```

Parametric Polymorphism

We can create distinct subclasses based on the values passed to constructors. But sometimes we want to create subclasses based on distinct *types*, and types can't be passed as parameters. (Types are not values, but rather a **property** of values.)

We see this problem in Java, which tries to create general purpose data structures by basing them on the class `Object`. Since any object can be assigned to `Object` (all classes must be a subclass of `Object`), this works—at least partially.

```
class LinkedList {
  Object value;
  LinkedList next;
  Object head() {return value;}
  LinkedList tail(){return next;}
  LinkedList(Object O) {
     value = O; next = null;}
  LinkedList(Object O,
             LinkedList L){
     value = O; next = L;}
}
```

Using this class, we can create a linked list of any subtype of **`Object`**.

But,

- We can't guarantee that linked lists are *type homogeneous* (contain only a single type).
- We must cast **`Object`** types back into their "real" types when we extract list values.

- We must use wrapper classes like **`Integer`** rather than **`int`** (because primitive types like **`int`** aren't objects, and aren't subclass of **`Object`**).

For example, to use **`LinkedList`** to build a linked list of **`int`**s we do the following:

```
LinkedList l =
  new LinkedList(new Integer(123));
int i =
  ((Integer) l.head()).intValue();
```

This is pretty clumsy code. We'd prefer a mechanism that allows us to create a "custom version" of **`LinkedList`**, based on the type we want the list to contain.

We can't just call something like **`LinkedList(int)`** or **`LinkedList(Integer)`** because types can't be passed as parameters.

Parametric polymorphism is the solution. Using this mechanism, we *can* use type parameters to build a "custom version" of a class from a general purpose class.

C++ allows this using its template mechanism. Tiger Java also allows type parameters.

In both languages, type parameters are enclosed in "angle brackets" (e.g., **`LinkedList<T>`** passes **`T`**, a type, to the **`LinkedList`** class).

Thus we have

```
class LinkedList<T> {
  T value; LinkedList<T> next;
  T head() {return value;}
  LinkedList<T> tail() {
     return next;}
  LinkedList(T O) {
     value = O; next = null;}
  LinkedList(T O,LinkedList<T> L)
     {value = O; next = L;}
}
LinkedList<int> l =
   new LinkedList(123);
int i = l.head();
```