Modules

information hiding: Makes objects and algorithms invisible to portions of the system that do not need them.

Information hiding is crucial for software maintenance. Om addition to reducing cognitive load, hiding reduces the risk of name conflicts. It also safeguards the integrity of data abstractions: any attempt to access an object outside of the module to which it belongs will cause the compiler to issue an "undefined symbol" error message. Finally, it helps to compartmentalize run-time errors: if a variable takes on an unexpected value, we can generally be sure that the code that modified it is in the variable's scope.

Encapsulating Data and Subroutines

The information hiding provided by nested subroutines is limited to objects whose lifetime is the same as that of the subroutine in which they are hidden. When control returns from a subroutine, its local variables will no longer be live: their values will be discarded.

Static variables allow a subroutine to have "memory"— to retain information from one invocation to the next— while protecting that memory from accidental access or modification by other parts of the program. Static variables allow programmers to build single-subroutine abstractions. Unfortunately, they do not allow the construction of abstractions whose interface needs to consist of more than one subroutine.

Modules as Abstractions

A module allows a collection of objects—subroutines, variables, types, and so on— to be encapsulated in such a way that:

(1) objects inside are visible to each other, but

(2) objects on the inside may not be visible on the outside unless they are **exported**, and

(3) objects on the outside may not be visible on the inside unless they are **imported**.

Only the **visibility** of objects is affected with import and export statements; modules do not affect the lifetime of the objects they contain.

Ada, Java, and Perl use the term **package**.

C++,C#, and PHP use the term **namespace**.

Modules can be emulated to some degree through use of the separate *compilation* facilities of C.

Bindings of names made inside the namespace may be partially or totally hidden (inactive) on the outside— but no destroyed. In C++, where namespaces can appear only at the outermost level of lexical nesting.

Imports and Exports

Modules into which names must be explicitly imported are said to be **closed scopes**.

Modules that do not require imports are said to be **open scopes**.

C++ is representative of an increasingly common option, in which *names* are automatically exported, but are available on the outside only when *qualified* with the module name—unless they are explicitly "imported" by another scope, at which point they are available unqualified. This option, which we call **selectively open** modules, also appear in Ada, Java, C#, and Python.

Modules as Managers

Modules facilitate the construction of abstractions by allowing data to be made private to the subroutines that use them.

In more complex programs, it may make sense for a module to export several related types, instances of which can then be passed to its subroutines.

See Mediator (Design Pattern)