GO PACKAGE HIERARCHY

In Go, files, packages, and modules form a hierarchical structure for organizing code:

Files:

Go source files are individual .go files that contain Go code. Each file must declare which package it belongs to using a package statement at the top. For example:

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
```

Packages:

A package is a collection of Go source files in the same directory that all declare the same package name. Packages are the basic unit of code organization and compilation in Go. All files in a directory must belong to the same package (with the exception of test files and the special main package).

Key points about packages:

- Package names should be short, lowercase, and descriptive
- The main package is special - it defines an executable program rather than a library
- Functions, types, and variables that start with a capital letter are exported (public), while those starting with lowercase are unexported (private to the package)
- You import other packages to use their exported functionality

Example package structure:

```
myproject/
  ├── main.go          (package main)
  ├── utils/
  │   ├── helper.go    (package utils)
  │   └── math.go      (package utils)
  └── models/
      └── user.go      (package models)
```

Modules:

Modules are collections of related packages that are versioned together. A module is defined by a go.mod file at its root directory. Modules were introduced in Go 1.11 to solve dependency management problems.

The go.mod file specifies:

- The module path (used as the import path prefix for packages within the module)
- The Go version the module requires
- Dependencies on other modules

Example go.mod:

```
module github.com/username/myproject

go 1.21

require (
    github.com/gorilla/mux v1.8.0
    github.com/lib/pq v1.10.7
)
```

Relationship Summary:

- **Files** contain the actual Go source code
- **Packages** group related files together and provide namespacing
- **Modules** group related packages together and handle versioning/dependencies

A typical workflow involves writing code in files, organizing those files into logical packages, and managing those packages (along with external dependencies) through modules. This three-tier system provides both local organization and broader ecosystem compatibility.