# Import Functionality

This lesson introduces a basic component of Go program, i.e., packages.

## Packages

A library, module, or namespace in any other language is called a **package**. Packages are a way to structure code. A program is constructed as a *package* which may use facilities from other packages. A package is often abbreviated as *'pkg'*.

Every Go file belongs to only *one* package whereas one package can comprise many different Go files. Hence, the filename(s) and the package name are generally *not* the same. The package to which the code-file belongs must be indicated on the *first* line. A package name is written in *lowercase* letters. For example, if your code-file belongs to a package called **main**, do the following:

```
package main
```

A standalone executable belongs to *main*. Each Go application contains one *main*.

An application can consist of different packages. But even if you use only package *main*, you don't have to stuff all code in 1 big file. You can make a number of smaller files, each having `package main` as the 1st line of code. If you compile a source file with a package name other than main, e.g., **pack1**, the object file is stored in **pack1.a**.

## Package dependencies

To build a program, the packages, and the files within them must be compiled in the correct order. Package dependencies determine the order in which to build the packages. Within a package, the source files must all be compiled together. The package is compiled as a unit, and by convention, each directory contains one package. If a package is changed and recompiled, all the client programs that use this package must be recompiled too!

## Import keyword

A Go program is created by *linking* set of packages together, with the **import** keyword. For example, if you want to import a package say **fmt**, then you do:

```
package main
import "fmt"
```

`import "fmt"` tells Go that this program needs functions, or other elements from the package `fmt`, which implements a functionality for formatted IO. The package names are enclosed within " "(double quotes).

Import loads the public declarations from the compiled package; it does not insert the source code. If multiple packages are needed, they can each be imported by a separate statement. For example, if you want to import two packages, `fmt` and `os` in one code file, there are some following ways to do so:

```
import "fmt"
import "os"
```

or you can do:

```
import "fmt"; import "os"
```

Go has provided us with a shorter and more elegant way of importing multiple packages known as *factoring the keyword*. It is stated as:

```
import (
  "fmt"
  "os"
)
```

Factoring means calling a *keyword* once on multiple instances. You may have noticed that we imported two packages using a single `import` keyword. It is also applicable to keywords like `const`, `var`, and `type`.

## Visibility #

Packages contain all other code objects apart from the blank identifier (`_`). Also, identifiers of code-objects in a package have to be unique which means that there can be no naming conflicts. However, the same identifier can be used in different packages. The package name qualifies a package to be different.

### Visibility rule #

Packages expose their code objects to code outside of the package according to the following rule enforced by the compiler:

When the identifier (of a constant, variable, type, function, struct field, …) starts with an uppercase letter, like, **Group1**, then the 'object' with this identifier is visible in code outside the package (thus available to client-programs, or 'importers' of the package), and it is said to be exported (like public identifiers/variables in OO languages). Identifiers that start with a lowercase letter are not visible outside the package, but they are visible and usable in the whole package (like private identifiers/variables).

> **Note:** Capital letters can come from the entire *Unicode-range*, like Greek; not only ASCII letters are allowed.

Importing a package gives access only to the exported objects in that package. Suppose we have an instance of a variable or a function called `Object` (starts with O so it is exported) in a package `pack1`. When `pack1` is imported in the current package, `Object` can be called with the usual dot-notation from OO-languages:

```
pack1.Object
```

Packages also serve as namespaces and can help us avoid name-conflicts. For

Packages also serve as namespaces and can help us avoid name conflicts. For example, variables with the *same* name in two packages are differentiated by

their package name, like `pack1.Object` and `pack2.Object`.

A package can also be given another name called an *alias*. If you name a package then its alias will be used throughout the code, rather than its original name. For example:

```go
import fm "fmt"
```

Now in the code, whenever you want to use `fmt`, use its alias (not `fmt`).

> **Note:** Go has a motto known as *“No unnecessary code!”*. So importing a package which is not used in the rest of the code is a *build-error*.

---

That’s how functionalities are imported in the Golang. In the next lesson, you’ll learn how to write a function in Go.