Chapter-13 : Building Projects.

→ Modules : Haskell programs are organized into modules. Modules contain the datatypes, type synonyms, typeclasses, typeclass instances, and values we define at the top level.

↓

They offer a means to import other modules into the scope of our program, and they also contain values that can be exported to other modules.

↓

Primary focus for this chapter is to understand how to setup a project in Haskell, use the pkg manager known as Cabal, build the project with Stack, and work with Haskell modules as they are.

→ Making pkgs with stack : The Haskell Cabal, or Common Architecture for Building Applications & libraries, is a pkg mgr. A pkg. is a program we're building, including all its modules & dependencies.

↓

A pkg. has dependencies which are the interlinked elements of that program, i.e. the other libraries and pkgs. it may depend on alongwith any tests and documentation associated with the project.

↓

Cabal exists to help organize all this and make sure all dependencies are properly in scope.

↓

Stack is a cross platform program for developing Haskell projects, as it helps us manage projects with multiple as well as individual pkgs, whereas Cabal exists primarily to describe a single pkg. with a Cabal file that has .cabal file extension.

↓

Stack is built on top of Cabal in some imp. senses, hence we would still be working with .cabal files. However, stack simplifies the process somewhat, especially in large projects with multiple dependencies, by allowing us to build those large libraries only once and use them across projects.

↓

Stack also relies on LTS snapshot of Haskell pkgs from Stackage, that are guaranteed to work together, unlike pkgs from Hackage which may have conflicting dependencies.

↓

The basic structure embodied in Stack templates is the recommended project layout.

→ Working with a basic project:

*) Building the project: i) stack build

→ Making our program interactive

*) sayHello :: String → IO ( )
```
sayHello name =
    putStrLn ("Hi ++ name ++ "!")
```

*) main :: IO ( ) → main fn. to get user's name
```
main = do
  name ← getLine
  sayHello name
  dogs
```

do → Syntactic sugar → used inside fns. that return IO in order to sequence side effects in a convenient syntax
↓
begins the block

← bind

name → The result of binding over the IO String to String, bound to name'.

getLine → has type IO String, because it must perform I/O in order to obtained the String, and then returns that String

dogs → dogs expect nothing and is an IO action of type IO ( ), which fits the overall type of main.

getLine :: IO String
name :: String