Singleton class

1. The global variable `singleInstance` plays a crucial role in implementing the Singleton pattern. Its purpose is to hold the single instance of the `single` struct so that it can be accessed and reused whenever needed. Here's how it works:

   Initialization: When the program starts running, the `singleInstance` is initially `nil`, indicating that no instance of the `single` struct has been created yet.
   Instance Creation: When the `getInstance()` function is called for the first time, it checks if `singleInstance` is `nil`. If it is, the function creates a new instance of the `single` struct and assigns it to the `singleInstance` variable.
   Subsequent Calls: On subsequent calls to `getInstance()`, since `singleInstance` is no longer `nil`, the function simply returns the existing instance that was created earlier. This ensures that only one instance of the `single` struct is ever created throughout the program's execution.
   Global Accessibility: Because `singleInstance` is a global variable, it can be accessed from any part of the program, allowing you to easily obtain the singleton instance from different functions or methods.

The global variable essentially serves as a "memory" for the instance, allowing the Singleton pattern to maintain its single-instance guarantee. It helps to keep track of whether the instance has been created and holds a reference to that instance so that it can be reused as needed.

2. In the provided code, there is no explicit constructor declared because Go does not have traditional constructors like some other programming languages do. Instead, the construction and initialization of the `single` struct happen directly when an instance is created.

Here's how it works:

```
if singleInstance == nil {
    singleInstance = &single{name: "somesh", age: 23}
}
```

In this part of the `getInstance()` function, when the `singleInstance` is `nil`, it means that no instance has been created yet. So, the code creates a new instance of the `single` struct directly and initializes its fields (`name` and `age`) in the same line. The address of this new instance is assigned to the `singleInstance` variable, effectively initializing the singleton instance.

In Go, this kind of initialization when creating a new instance is quite common. While it's not called a constructor in the traditional sense, it achieves the same goal of initializing the fields of a newly created object.