Lesson 3.1 - Defining & Creating Structs

This lesson introduces structs in Go as user-defined types that group together named fields of different types. It explains how to define structs and various ways to create and initialize struct instances, including positional and named initialization, and discusses the convention of using constructor functions.

Key Takeaways

- Structs are composite data types used to group related data together with named fields.
- Struct instances can be created using struct literals, either by providing values in the order of field declaration or by explicitly naming the fields.
- When initializing structs by position, you must provide a value for all the fields in the order they are defined.
- Constructor functions (often named New...) are commonly used to create and initialize struct instances, potentially including validation or default values.

Key Concepts

- **Struct:** A user-defined type in Go that groups together zero or more named fields of arbitrary types. It is similar to classes in other languages but without inheritance.
- **Field:** A named member of a struct that holds a value of a specific type.
- **Struct Literal:** The syntax used to create an instance of a struct by providing values for its fields, either by position (item{10, 20}) or by name (item{x: 10, y: 20}).
- **Constructor Function:** A function that creates and returns an initialized instance of a struct. It can encapsulate initialization logic and enforce invariants.

Lesson 3.2 - Adding Methods to Structs

This lesson explains how to add methods to structs in Go, distinguishing between value receivers and pointer receivers and when to use each. It also introduces the concept of embedding structs to promote code composition and reuse.

Key Takeaways

- Methods in Go are functions associated with a specific type (like a struct) through a receiver.
- Value receivers operate on a copy of the struct, so changes made within the method do not affect the original struct instance.
- Pointer receivers operate on the actual struct instance, allowing modifications to its fields. You must use a pointer receiver if you need to mutate the struct.
- Embedding a struct within another allows the outer struct to directly access the fields and methods of the embedded struct, promoting composition over inheritance.

Key Concepts

- **Method:** A function that has a receiver argument. The receiver specifies the type the method is associated with.
- **Value Receiver:** The receiver parameter of a method is a value of the struct type itself (i item). Changes to the fields of i inside the method are not reflected in the original item.
- **Pointer Receiver:** The receiver parameter of a method is a pointer to the struct type (i *item). Changes to the fields of i inside the method do affect the original item. This is necessary for mutating the struct.
- **Embedding:** Including one struct type as an anonymous field within another struct type. This allows the outer struct to access the inner struct's fields and methods as if they were its own, facilitating code reuse through composition.

Lesson 3.3 - Creating & Implementing Interfaces

This lesson introduces interfaces in Go as sets of method signatures that define a contract of behavior. It highlights Go's principle of defining interfaces based on "what you need, not what you provide" and demonstrates how types implicitly implement interfaces by implementing the required methods. The lesson also covers method sets and using iota for creating enumerated constants.

Key Takeaways

- An interface in Go is a type that specifies a set of methods (function signatures).
- A type implicitly implements an interface if it provides all the methods defined in that interface. There is no explicit "implements" keyword.
- Go interfaces are typically small, focusing on the minimal set of methods needed to define a specific behavior.
- The **iota** keyword can be used to create sequences of related constant values, often used to simulate enums.
- The **fmt.Stringer** interface allows a type to define its own string representation.

Key Concepts

- **Interface:** A type that defines a set of methods (signatures). Any type that implements all the methods in an interface is said to satisfy that interface. Interfaces define behavior.
- **Implicit Implementation:** In Go, a type satisfies an interface simply by implementing all of its methods. There is no explicit declaration required.
- **Method Set:** The set of methods associated with a given type. When checking if a type implements an interface, Go considers the method set of the type (including both value and pointer receivers).
- **iota:** A predeclared identifier representing successive untyped integer constants in a constant declaration. It is reset to 0 whenever const appears and increments after each constant specification within a group.

Lesson 3.4 - The Empty Interface

This lesson focuses on the empty interface (interface{}). It explains that it has no methods, meaning that all types implement it. While it offers flexibility to hold any type, its use should be minimized and primarily reserved for scenarios like serialization or printing where the type is inherently unknown.

Key Takeaways

- The empty interface (interface{}) has zero methods.
- Every type in Go implements the empty interface because there are no method requirements to fulfill.
- The empty interface can be used as a placeholder for any type when the specific type is not known or needs to vary.
- Overuse of the empty interface can weaken type safety, and more specific interfaces should be preferred when possible.
- Type assertions can be used to retrieve the underlying concrete type of a value stored in an empty interface variable.

Key Concepts

- **Empty Interface (interface{}):** An interface with no methods specified. Because an interface specifies the methods a type must have, an interface with no methods imposes no requirements, so all types satisfy it.
- **Type Assertion:** An operation performed on an interface value that allows access to its underlying concrete type. The syntax is x.(T), where x is an interface value and T is the asserted type. It can panic if the underlying type is not T. The form v, ok := x.(T) provides a safer way to check the type.

Lesson 3.5 - Using Generics

This lesson introduces generics in Go, explaining how they enable writing code that can work with multiple types while maintaining type safety. It covers defining generic functions and generic data structures using type parameters declared with square brackets []. Generics help catch type-related errors at compile time.

Key Takeaways

- Generics allow you to write functions and types that are parameterized by types, enabling code reuse across different data types.
- Type parameters are declared in square brackets [] after the function or type name, e.g., func MyFunction[T any](arg T). The any keyword represents any type.
- Generics enhance type safety by ensuring that type-related errors are caught during compilation, similar to static typing but with more flexibility.
- Generics are particularly useful for implementing generic algorithms and data structures that can operate on various types without the need for type-specific implementations or relying on the empty interface.

Key Concepts

- **Type Parameter:** A placeholder for a type that is specified when a generic function or type is instantiated (used). Type parameters are declared within square brackets.
- **Generic Function:** A function that is parameterized by one or more type parameters, allowing it to operate on values of different types in a type-safe manner.
- **Generic Type:** A type (such as a struct or interface) that is parameterized by one or more type parameters, making it able to work with values of different types.