Basic Questions:

1. Can you explain the difference between var and := in Go?
2. How does Go handle memory management and garbage collection?
3. What are slices in Go, and how are they different from arrays?
4. What are Goroutines, and how do they differ from OS threads?
5. What is the purpose of the defer statement in Go?

Intermediate Questions:

6. Can you explain the use of interfaces in Go?
7. How does Go handle concurrency, and what are some common synchronization techniques?
8. What is a select statement, and how is it used with channels?
9. What are the advantages of using Go's standard library for HTTP server development?
10. How would you handle errors in Go effectively?

Advanced Questions:

11. Have you worked with Go modules? Can you explain how dependency management works in Go?
12. How would you optimize a Go program for performance?
13. Can you explain the difference between sync.Mutex and sync.RWMutex?
14. What are some common pitfalls when using Goroutines and channels?
15. Can you explain the context package and how it helps in request handling?

Answers:

Basic Questions

1. **Difference between var and := in Go?**
    - var is used for explicit variable declaration, allowing type specification.

go

CopyEdit

var x int = 10

    - := is a shorthand for variable declaration with type inference. It can only be used inside functions.

go

CopyEdit

x := 10 // Go infers x as int

2. **How does Go handle memory management and garbage collection?**

- Go has an automatic garbage collector that manages memory.
- It uses **tracing garbage collection** to detect unused memory and free it.
- The garbage collector is optimized to reduce pauses in program execution.

3. **What are slices, and how are they different from arrays?**
    - Arrays have fixed lengths:

go

CopyEdit

var arr [3]int = [3]int{1, 2, 3}

    - Slices are dynamic, internally backed by an array:

go

CopyEdit

slice := []int{1, 2, 3}

slice = append(slice, 4)

    - Slices have a length (len(slice)) and capacity (cap(slice)), while arrays don't grow.
4. **What are Goroutines, and how do they differ from OS threads?**
    - A Goroutine is a lightweight thread managed by the Go runtime.
    - Unlike OS threads, Goroutines are scheduled cooperatively, requiring less memory (~2KB per Goroutine).
    - Example:

go

CopyEdit

go func() {

  fmt.Println("Hello from Goroutine!")

}()

5. **Purpose of the defer statement?**
    - defer delays execution of a function until the surrounding function exits.
    - It's commonly used for cleanup tasks (e.g., closing files).
    - Example:

go

CopyEdit

func readFile() {

```
    f, _ := os.Open("file.txt")
    defer f.Close()  // Ensures file is closed at the end
}
```

Intermediate Questions

6. **Use of interfaces in Go?**
    - Interfaces define behavior without implementation.
    - Example:

```
go
CopyEdit
type Animal interface {
    Speak() string
}
type Dog struct{}
func (d Dog) Speak() string { return "Woof" }
var a Animal = Dog{}
fmt.Println(a.Speak())  // "Woof"
```

7. **Concurrency handling and synchronization techniques?**
    - Goroutines enable concurrency.
    - Synchronization techniques:
        - **Channels** for communication:

```
go
CopyEdit
ch := make(chan int)
go func() { ch <- 5 }()
fmt.Println(<-ch)  // 5
```

        - **sync.Mutex** for critical sections:

```
go
CopyEdit
var mu sync.Mutex
mu.Lock()
```

```
// Critical section
mu.Unlock()
```

- **sync.WaitGroup** for waiting on Goroutines.

8. **What is a select statement, and how is it used with channels?**
    - select is used to wait on multiple channel operations.
    - Example:

```
go
CopyEdit
ch1, ch2 := make(chan string), make(chan string)
go func() { ch1 <- "Hello" }()
go func() { ch2 <- "World" }()
select {
   case msg1 := <-ch1:
      fmt.Println(msg1)
   case msg2 := <-ch2:
      fmt.Println(msg2)
}
```

9. **Advantages of using Go's standard library for HTTP server development?**
    - Lightweight and efficient.
    - Built-in concurrency support.
    - Simple API:

```
go
CopyEdit
http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
   w.Write([]byte("Hello, World!"))
})
http.ListenAndServe(":8080", nil)
```

10. **Error handling in Go?**
    - Go uses explicit error handling via error type.
    - Example:

```
go
```

CopyEdit

```
func divide(a, b int) (int, error) {
   if b == 0 {
      return 0, errors.New("division by zero")
   }
   return a / b, nil
}
```

Advanced Questions

11. **Go Modules and dependency management?**
    - Go modules (go.mod) handle dependencies.
    - Commands:

sh

CopyEdit

```
go mod init myproject
go mod tidy  # Cleans up unused dependencies
go get github.com/some/package  # Adds dependency
```

12. **Optimizing Go programs for performance?**
    - **Use Goroutines** instead of OS threads.
    - **Avoid global variables** to prevent contention.
    - **Use sync.Pool** for object reuse.
    - **Optimize memory allocations** using make().
    - **Profile using pprof**:

sh

CopyEdit

```
go test -bench . -benchmem
```

13. **Difference between sync.Mutex and sync.RWMutex?**
    - sync.Mutex: Blocks both readers and writers.
    - sync.RWMutex: Allows multiple readers but only one writer.
    - Example:

go

CopyEdit

```
var mu sync.RWMutex
mu.RLock()  // Multiple readers can lock
mu.RUnlock()
mu.Lock()  // Only one writer can lock
mu.Unlock()
```

14. **Common pitfalls with Goroutines and channels?**
    - **Deadlocks** when Goroutines block indefinitely.
    - **Data races** when multiple Goroutines access shared data.
    - **Leaking Goroutines** if channels are not closed properly.
15. **Context package and request handling?**
    - context.Context helps control Goroutines via deadlines and cancellation.
    - Example:

go

CopyEdit

```
func handleRequest(w http.ResponseWriter, r *http.Request) {
   ctx, cancel := context.WithTimeout(r.Context(), 2*time.Second)
   defer cancel()

   select {
   case <-time.After(3 * time.Second):
      fmt.Fprintln(w, "Request completed")
   case <-ctx.Done():
      fmt.Fprintln(w, "Request cancelled")
   }
}
```