Complete Go Interview Preparation Guide

Table of Contents

- 1. <u>Language Fundamentals</u>
- 2. <u>Data Types and Variables</u>
- 3. Control Flow
- 4. Functions
- 5. Arrays and Slices
- 6. <u>Maps</u>
- 7. Structs and Methods
- 8. Interfaces
- 9. Pointers
- 10. Goroutines and Channels
- 11. Error Handling
- 12. <u>Packages and Modules</u>
- 13. <u>Testing</u>
- 14. Common Patterns
- 15. Practice Problems
- 16. Interview Questions

Language Fundamentals

Basic Syntax

```
go
package main
import "fmt"

func main() {
   fmt.Println("Hello, World!")
}
```

Variable Declaration

```
go
// Different ways to declare variables
var name string = "John"
var age = 25
count := 10
var x, y int = 1, 2
// Zero values
var s string // ""
var i int // 0
var b bool // false
var p *int // nil
```

Constants

```
const Pi = 3.14159
const (
StatusOK = 200
StatusNotFound = 404
)

// iota for auto-incrementing constants
const (
Sunday = iota // 0
Monday // 1
Tuesday // 2
)
```

Data Types and Variables

Basic Types

```
// Numeric types
var i8 int8 = 127
var i16 int16 = 32767
var i32 int32 = 2147483647
var i64 int64 = 9223372036854775807
var ui8 uint8 = 255
var f32 float32 = 3.14
var f64 float64 = 3.141592653589793

// Other types
var b bool = true
var s string = "Hello"
var r rune = 'A' // Unicode code point
var bt byte = 65 // Alias for uint8
```

Type Conversion

```
var i int = 42
var f float64 = float64(i)
var u uint = uint(f)
var s string = string(rune(i))

// String conversions
import "strconv"
str := strconv.ltoa(123)  // int to string
num, err := strconv.Atoi("123") // string to int
```

Control Flow

If Statements

```
if x > 0 {
    fmt.Println("positive")
} else if x < 0 {
    fmt.Println("negative")
} else {
    fmt.Println("zero")
}

// If with short statement
if num := getNumber(); num > 0 {
    fmt.Println("positive:", num)
}
```

Switch Statements

```
go
switch day := time.Now().Weekday(); {
    case day == time.Saturday || day == time.Sunday:
        fmt.Println("Weekend")
    default:
        fmt.Println("Weekday")
}

// Type switch
switch v := interface{}(x).(type) {
    case int:
        fmt.Printf("Integer: %d\n", v)
    case string:
        fmt.Printf("String: %s\n", v)
    default:
        fmt.Printf("Unknown type: %T\n", v)
}
```

Loops

```
// For loop (only loop in Go)
for i := 0; i < 10; i++ {
  fmt.Println(i)
// While-style loop
for condition {
  // code
// Infinite loop
for {
 // code
  break
// Range loop
slice := []int{1, 2, 3, 4, 5}
for index, value := range slice {
  fmt.Printf("Index: %d, Value: %d\n", index, value)
// Range over map
m := map[string]int{"a": 1, "b": 2}
for key, value := range m {
  fmt.Printf("%s: %d\n", key, value)
```

Functions

Basic Function Declaration

```
func add(a, b int) int {
  return a + b
}

// Multiple return values
func divmod(a, b int) (int, int) {
  return a / b, a % b
}

// Named return values
func rectangle(width, height float64) (area, perimeter float64) {
  area = width * height
  perimeter = 2 * (width + height)
  return // naked return
}
```

Variadic Functions

```
go
func sum(numbers ...int) int {
  total := 0
  for _, num := range numbers {
    total += num
  }
  return total
}

// Usage
result := sum(1, 2, 3, 4, 5)
slice := []int{1, 2, 3}
result2 := sum(slice...) // spread operator
```

Function as Values

```
func main() {
    //Function as variable
    add := func(a, b int) int {
        return a + b
    }

    //Function as parameter
    result := calculate(10, 5, add)
}

func calculate(a, b int, operation func(int, int) int) int {
    return operation(a, b)
}
```

Closures

```
func counter() func() int {
  count := 0
  return func() int {
    count++
    return count
  }
}
func main() {
  c := counter()
  fmt.Println(c()) // 1
  fmt.Println(c()) // 2
}
```

Arrays and Slices

Arrays

```
// Fixed size
var arr [5]int
arr[0] = 1

// Initialize with values
numbers := [5]int{1, 2, 3, 4, 5}
auto := [...]int{1, 2, 3} // compiler counts elements
```

Slices

Slice Operations

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

// Append
slice = append(slice, 4, 5)
slice = append(slice, []int{6, 7}...)

// Copy
dest := make([]int, len(slice))
copy(dest, slice)

// Delete element at index i
slice = append(slice[:i], slice[i+1:]...)

// Insert element at index i
slice = append(slice[:i], append([]int{newElement}, slice[i:]...)...)
```

Maps

Basic Map Operations

```
go
// Create map
var m map[string]int
m = make(map[string]int)
// Map literal
scores := map[string]int{
  "Alice": 85,
  "Bob": 92,
  "Carol": 78,
// Operations
scores["David"] = 88  // Add/update
delete(scores, "Carol") // Delete
value, exists := scores["Alice"] // Check existence
// Iterate
for name, score := range scores {
  fmt.Printf("%s: %d\n", name, score)
```

Structs and Methods

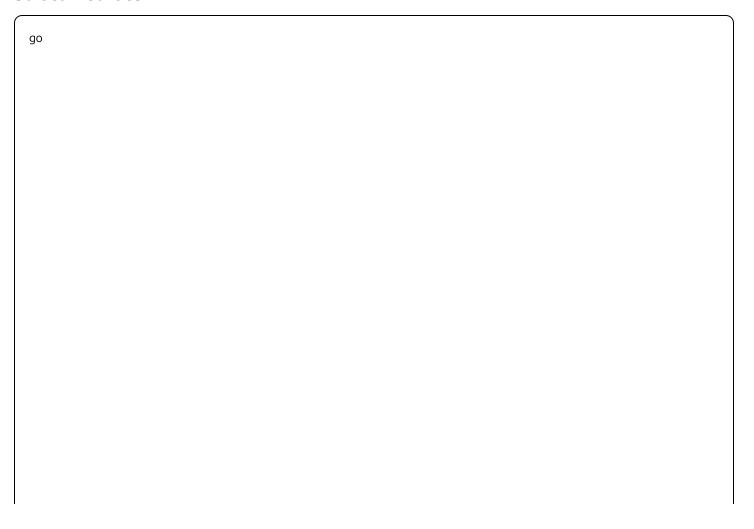
Struct Definition

```
go

type Person struct {
   Name string
   Age int
   City string
}

// Anonymous struct
var config = struct {
   Host string
   Port int
}{
   Host: "localhost",
   Port: 8080,
}
```

Struct Methods



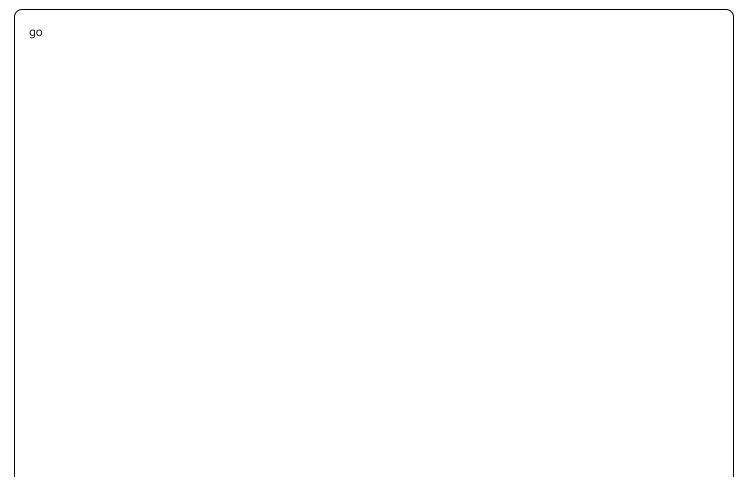
```
type Rectangle struct {
    Width, Height float64
}

// Value receiver
func (r Rectangle) Area() float64 {
    return r.Width * r.Height
}

// Pointer receiver
func (r *Rectangle) Scale(factor float64) {
    r.Width *= factor
    r.Height *= factor
}

// Usage
rect := Rectangle{Width: 10, Height: 5}
fmt.Println(rect.Area()) // 50
rect.Scale(2)
fmt.Println(rect.Width) // 20
```

Embedding (Composition)



```
type Animal struct {
  Name string
func (a Animal) Speak() {
  fmt.Printf("%s makes a sound\n", a.Name)
type Dog struct {
  Animal // embedded field
  Breed string
func (d Dog) Bark() {
  fmt.Printf("%s barks\n", d.Name)
// Usage
dog := Dog{
 Animal: Animal{Name: "Rex"},
 Breed: "Golden Retriever",
dog.Speak() // inherited method
dog.Bark() // own method
```

Interfaces

Interface Definition

```
type Writer interface {
    Write([]byte) (int, error)
}

type Closer interface {
    Close() error
}

type WriteCloser interface {
    Writer
    Closer
}
```

Interface Implementation

```
go

type File struct {
    name string
}

func (f *File) Write(data []byte) (int, error) {
    fmt.Printf("Writing %s to %s\n", string(data), f.name)
    return len(data), nil
}

func (f *File) Close() error {
    fmt.Printf("Closing %s\n", f.name)
    return nil
}

// File automatically implements WriteCloser
var wc WriteCloser = &File{name: "test.txt"}
```

Empty Interface and Type Assertions

```
var i interface{} = "hello"

// Type assertion
s := i.(string)
s, ok := i.(string) // safe assertion

// Type switch
switch v := i.(type) {
    case string:
        fmt.Printf("String: %s\n", v)
    case int:
        fmt.Printf("Integer: %d\n", v)
    default:
        fmt.Printf("Unknown type: %T\n", v)
}
```

Pointers

Pointer Basics

Pointer vs Value Receivers

```
go
```

```
type Counter struct {
    value int
}

// Value receiver - operates on copy
func (c Counter) ValueIncrement() {
    c.value++// doesn't affect original
}

// Pointer receiver - operates on original
func (c *Counter) PointerIncrement() {
    c.value++// affects original
}
```

Goroutines and Channels

Goroutines

```
func main() {

// Start goroutine
go sayHello("world")

// Anonymous goroutine
go func() {

fmt.Println("anonymous goroutine")
}()

// Wait for goroutines
time.Sleep(time.Second)
}

func sayHello(name string) {

fmt.Printf("Hello, %s!\n", name)
}
```

Channels

```
// Create channel
ch := make(chan int)
bufferedCh := make(chan int, 100)

// Send and receive
go func() {
   ch <- 42 // send
}()
   value := <-ch // receive

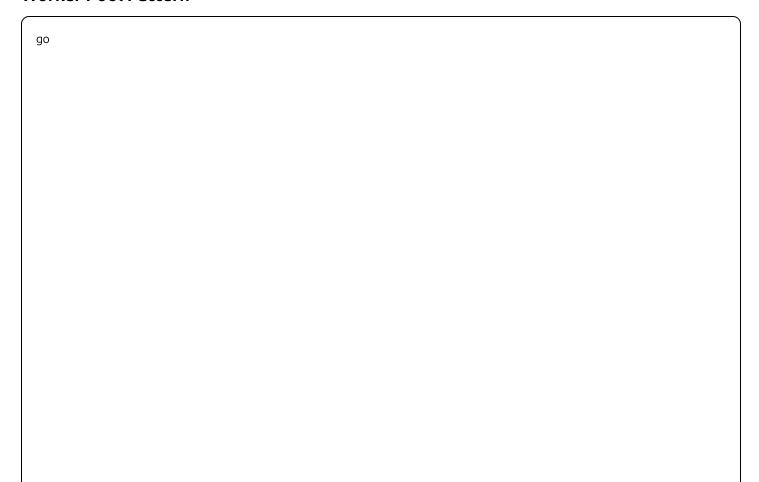
// Channel directions
func sender(ch chan<- int) { // send-only
   ch <- 42
}

func receiver(ch <-chan int) { // receive-only
   value := <-ch
   fmt.Println(value)
}</pre>
```

Channel Patterns

```
// Select statement
select {
case msg1 := <-ch1:
  fmt.Println("Received from ch1:", msg1)
case msg2 := <-ch2:
  fmt.Println("Received from ch2:", msg2)
case <-time.After(time.Second):</pre>
  fmt.Println("Timeout")
default:
  fmt.Println("No channels ready")
// Channel closing
close(ch)
value, ok := <-ch // ok is false if channel is closed
// Range over channel
for value := range ch {
  fmt.Println(value)
}// exits when channel is closed
```

Worker Pool Pattern



```
func workerPool(jobs <-chan int, results chan<- int, workerID int) {</pre>
 for job := range jobs {
   fmt.Printf("Worker %d processing job %d\n", workerID, job)
    results <- job * 2
func main() {
 jobs := make(chan int, 100)
 results := make(chan int, 100)
 // Start workers
 for w := 1; w <= 3; w++ {
   go workerPool(jobs, results, w)
 // Send jobs
 for j := 1; j <= 9; j++ {
   jobs <- j
  close(jobs)
 // Collect results
 for r := 1; r <= 9; r++ {
    <-results
```

Error Handling

Basic Error Handling

```
import "errors"

func divide(a, b float64) (float64, error) {
    if b == 0 {
        return 0, errors.New("division by zero")
    }
    return a / b, nil
}

// Usage
result, err := divide(10, 2)
if err != nil {
    fmt.Printf("Error: %v\n", err)
    return
}
fmt.Printf("Result: %f\n", result)
```

Custom Errors

```
type ValidationError struct {
Field string
Message string
}

func (e *ValidationError) Error() string {
return fmt.Sprintf("validation failed for %s: %s", e.Field, e.Message)
}

func validateAge(age int) error {
   if age < 0 || age > 150 {
      return &ValidationError{
      Field: "age",
      Message: "must be between 0 and 150",
   }
}
return nil
}
```

Error Wrapping (Go 1.13+)

```
import "fmt"

func processFile(filename string) error {
    file, err := os.Open(filename)
    if err != nil {
        return fmt.Errorf("failed to open file %s: %w", filename, err)
    }
    defer file.Close()

// process file...
    return nil
}

// Unwrap errors
if err := processFile("test.txt"); err != nil {
    var pathErr *os.PathError
    if errors.As(err, &pathErr) {
        fmt.Println("Path error:", pathErr.Path)
    }
}
```

Packages and Modules

Package Structure

```
go

// math/calculator.go
package math

import "errors"

// Exported function (starts with capital letter)
func Add(a, b int) int {
  return a + b
}

// Unexported function (starts with lowercase letter)
func multiply(a, b int) int {
  return a * b
}
```

Module Management

```
bash

# Initialize module
go mod init example.com/myproject

# Add dependency
go get github.com/gin-gonic/gin

# Update dependencies
go mod tidy

# Vendor dependencies
go mod vendor
```

Testing

Basic Testing

```
go
//math_test.go
package math
import "testing"

func TestAdd(t *testing.T) {
    result := Add(2, 3)
    expected := 5

    if result!= expected {
        t.Errorf("Add(2, 3) = %d; want %d", result, expected)
    }
}

func TestAddNegative(t *testing.T) {
    result := Add(-1, 1)
    if result!= 0 {
        t.Errorf("Add(-1, 1) = %d; want 0", result)
    }
}
```

Table-Driven Tests

```
go
func TestAddTable(t *testing.T) {
  tests := []struct {
    name string
   a, b int
   want int
 }{
   {"positive numbers", 2, 3, 5},
   {"negative numbers", -1, -1, -2},
   {"mixed", -1, 1, 0},
   {"zeros", 0, 0, 0},
  for _, tt := range tests {
   t.Run(tt.name, func(t *testing.T) {
      if got := Add(tt.a, tt.b); got != tt.want {
        t.Errorf("Add(%d, %d) = %d, want %d", tt.a, tt.b, got, tt.want)
   })
```

Benchmarks

```
go
func BenchmarkAdd(b *testing.B) {
    for i := 0; i < b.N; i++ {
        Add(2, 3)
    }
}</pre>
```

Common Patterns

Singleton Pattern

```
go
```

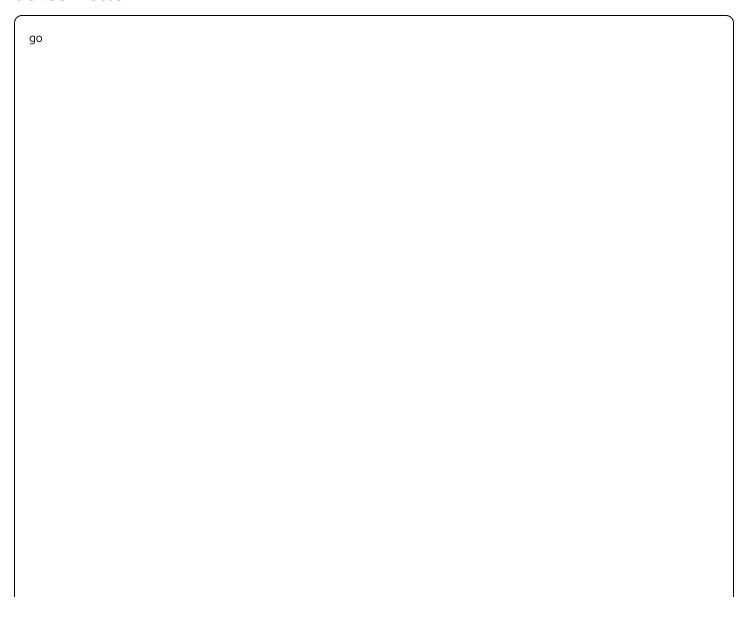
```
import "sync"

type singleton struct {
  value string
}

var instance *singleton
  var once sync.Once

func GetInstance() *singleton {
  once.Do(func() {
    instance = &singleton{value: "I'm singleton"}
  })
  return instance
}
```

Builder Pattern



```
type HTTPClient struct {
  timeout time. Duration
  retries int
  baseURL string
type HTTPClientBuilder struct {
  client *HTTPClient
func NewHTTPClientBuilder() *HTTPClientBuilder {
  return &HTTPClientBuilder{
    client: &HTTPClient{
      timeout: 30 * time.Second,
      retries: 3.
     baseURL: "",
   },
func (b *HTTPClientBuilder) Timeout(d time.Duration) *HTTPClientBuilder {
  b.client.timeout = d
  return b
func (b *HTTPClientBuilder) Retries(r int) *HTTPClientBuilder {
  b.client.retries = r
  return b
func (b *HTTPClientBuilder) Build() *HTTPClient {
  return b.client
// Usage
client := NewHTTPClientBuilder().
  Timeout(60*time.Second).
  Retries(5).
  Build()
```

Context Pattern

```
import "context"

func processWithTimeout(ctx context.Context, data string) error {
    // Create context with timeout
    ctx, cancel := context.WithTimeout(ctx, 5*time.Second)
    defer cancel()

select {
    case <-time.After(10 * time.Second):// simulate work
    return nil
    case <-ctx.Done():
        return ctx.Err() // timeout or cancellation
    }
}</pre>
```

Practice Problems

1. FizzBuzz

```
func fizzBuzz(n int) {
    for i := 1; i <= n; i++ {
        switch {
        case i%15 == 0:
            fmt.Println("FizzBuzz")
        case i%3 == 0:
            fmt.Println("Fizz")
        case i%5 == 0:
            fmt.Println("Buzz")
        default:
            fmt.Println(i)
        }
    }
}</pre>
```

2. Palindrome Check

```
func isPalindrome(s string) bool {
   runes := []rune(strings.ToLower(s))
   left, right := 0, len(runes)-1

for left < right {
    if runes[left] != runes[right] {
       return false
    }
   left++
    right--
   }
   return true
}</pre>
```

3. Fibonacci with Memoization

```
func fibonacci() func(int) int {
  cache := make(map[int]int)

var fib func(int) int
fib = func(n int) int {
  if n <= 1 {
    return n
  }

  if val, exists := cache[n]; exists {
    return val
  }

  cache[n] = fib(n-1) + fib(n-2)
  return cache[n]
}

return fib
}</pre>
```

4. Concurrent URL Fetcher

```
import (
  "fmt"
  "io"
  "net/http"
  "sync"
  "time"
type Result struct {
 URL string
  Response string
  Error error
func fetchURLs(urls []string) []Result {
  var wg sync.WaitGroup
  results := make([]Result, len(urls))
  for i, url := range urls {
   wg.Add(1)
   go func(index int, u string) {
     defer wg.Done()
     client := &http.Client{Timeout: 10 * time.Second}
     resp, err := client.Get(u)
     if err != nil {
       results[index] = Result{URL: u, Error: err}
       return
     defer resp.Body.Close()
     body, err := io.ReadAll(resp.Body)
     results[index] = Result{
       URL: u,
       Response: string(body)[:100], // first 100 chars
       Error: err,
     }
   }(i, url)
  wg.Wait()
```

```
return results
```

5. Generic Stack

```
go
type Stack[T any] struct {
  items []T
func (s *Stack[T]) Push(item T) {
 s.items = append(s.items, item)
func (s *Stack[T]) Pop() (T, bool) {
  if len(s.items) == 0 {
    var zero T
   return zero, false
  index := len(s.items) - 1
  item := s.items[index]
  s.items = s.items[:index]
  return item, true
func (s *Stack[T]) Peek() (T, bool) {
 if len(s.items) == 0 {
    var zero T
    return zero, false
  return s.items[len(s.items)-1], true
// Usage
stack := &Stack[int]{}
stack.Push(1)
stack.Push(2)
value, ok := stack.Pop() // 2, true
```

Interview Questions

Basic Questions

Q: What are the main features of Go? A: Go features include:

- Simple syntax and fast compilation
- Built-in concurrency with goroutines and channels
- Garbage collection
- Strong static typing with type inference
- Composition over inheritance
- Rich standard library
- Cross-platform compilation

Q: What's the difference between (make) and (new)? A:

- (new(T)) allocates zeroed storage for a new item of type T and returns a pointer (*T)
- (make(T, args)) creates slices, maps, and channels only, returns an initialized (not zeroed) value of type (make(T, args))

```
p := new([]int) // p is *[]int, *p is nil slice
s := make([]int, 0) // s is []int, initialized empty slice
```

Q: Explain goroutines vs threads A: Goroutines are lightweight user-space threads managed by the Go runtime. They have smaller stack size (2KB vs 1-8MB for OS threads), cheaper creation/destruction, and are multiplexed onto OS threads by the Go scheduler.

Intermediate Questions

Q: What's the difference between buffered and unbuffered channels? A:

- Unbuffered channels provide synchronous communication sender blocks until receiver is ready
- Buffered channels allow asynchronous communication up to buffer capacity
- Use buffered channels to improve performance when you know the capacity needed

Q: How does the Go scheduler work? A: Go uses an M:N scheduler where M goroutines are multiplexed onto N OS threads. The scheduler uses work-stealing queues and can preempt goroutines at function calls and channel operations.

Q: What are some common concurrency patterns? A:

- 1. Worker pools for processing jobs
- 2. Fan-out/fan-in for parallel processing
- 3. Pipeline pattern for data processing stages
- 4. Context for cancellation and timeouts

Advanced Questions

Q: How do you prevent race conditions? A: Use synchronization primitives:

- Mutex for exclusive access
- RWMutex for read-write scenarios
- Channels for communication
- Atomic operations for simple counters
- sync.Once for one-time initialization

Q: Explain interface satisfaction and duck typing A: Go uses structural typing - if a type implements all methods of an interface, it automatically satisfies that interface. No explicit declaration needed.

Q: What's the difference between composition and embedding? A: Embedding automatically promotes methods from embedded types, while composition requires explicit method calls through the composed field.

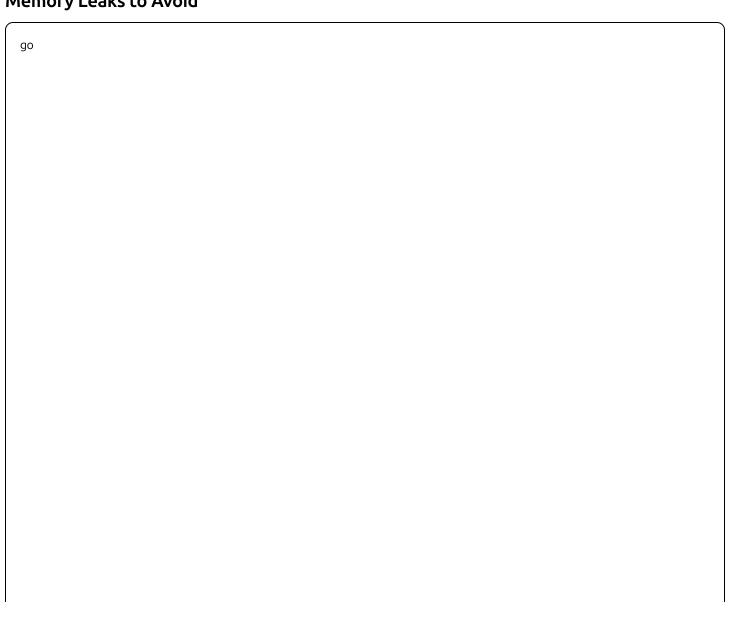
Memory Management

Understanding Pointers and Memory

```
// Stack vs heap allocation
func stackExample() {
    x := 42 // likely allocated on stack
    fmt.Println(x)
} // x goes out of scope, memory reclaimed

func heapExample() *int {
    x := 42 // escapes to heap because we return pointer
    return &x
}
```

Memory Leaks to Avoid



```
// Slice memory leak - keeping reference to large underlying array
func processLargeSlice() []int {
  large := make([]int, 1000000)
  //... fill large slice
  // BAD: keeps entire large array in memory
  return large[0:10]
  // GOOD: copy to new slice
  result := make([]int, 10)
  copy(result, large[0:10])
  return result
// Goroutine leak - goroutine never exits
func leakyGoroutine() {
  ch := make(chan int)
  go func() {
    for {
      select {
      case <-ch:
       // process
      // Missing case for shutdown!
  }()
  // Channel never closed, goroutine never exits
```

Performance Tips

Efficient String Building

```
import "strings"

// BAD: creates new string each iteration
func inefficientConcat(words []string) string {
    result := ""
    for _, word := range words {
        result += word + " "
    }
    return result
}

// GOOD: use strings.Builder
func efficientConcat(words []string) string {
    var builder strings.Builder
    for _, word := range words {
        builder.WriteString(word)
        builder.WriteString(" ")
    }
    return builder.String()
}
```

Slice Preallocation

```
go
// BAD: multiple allocations as slice grows
func inefficientAppend() []int {
    var result []int
    for i := 0; i < 1000; i++ {
        result = append(result, i)
    }
    return result
}

// GOOD: preallocate capacity
func efficientAppend() []int {
    result := make([]int, 0, 1000)
    for i := 0; i < 1000; i++ {
        result = append(result, i)
    }
    return result
}</pre>
```

Common Interview Coding Challenges

1. Two Sum

```
func twoSum(nums []int, target int) []int {
    numMap := make(map[int]int)

for i, num := range nums {
    complement := target - num
    if index, exists := numMap[complement]; exists {
        return []int(index, i)
      }
      numMap[num] = i
    }
    return nil
}
```

2. Reverse Linked List

```
type ListNode struct {
    Val int
    Next *ListNode
}

func reverseList(head *ListNode) *ListNode {
    var prev *ListNode
    current := head

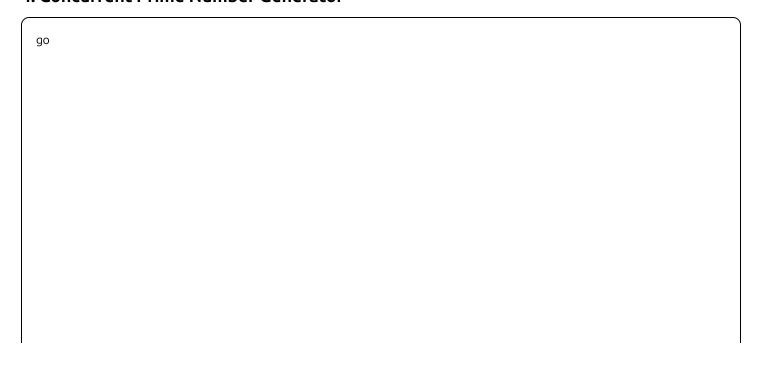
for current!= nil {
    next := current.Next
    current.Next = prev
    prev = current
    current = next
}

return prev
}
```

3. Valid Parentheses

```
go
func isValid(s string) bool {
 stack := []rune{}
 pairs := map[rune]rune{
   ')': '(',
   '}': '{',
   ']': '[',
 for _, char := range s {
   if opening, exists := pairs[char]; exists {
     // Closing bracket
     if len(stack) == 0 || stack[len(stack)-1] != opening {
        return false
      stack = stack[:len(stack)-1] // pop
   } else {
     // Opening bracket
     stack = append(stack, char) // push
 return len(stack) == 0
```

4. Concurrent Prime Number Generator



```
func generatePrimes(max int) <-chan int {</pre>
  primes := make(chan int)
  go func() {
    defer close(primes)
    isPrime := func(n int) bool {
      if n < 2 {
        return false
      for i := 2; i*i <= n; i++ {
       if n\%i == 0 {
          return false
      return true
    for i := 2; i <= max; i++ {
      if isPrime(i) {
        primes <- i
  }()
  return primes
// Usage
for prime := range generatePrimes(100) {
  fmt.Println(prime)
```

5. Rate Limiter

```
import "time"
type RateLimiter struct {
  tokens chan struct{}
func NewRateLimiter(rate int) *RateLimiter {
  rl := &RateLimiter{
    tokens: make(chan struct{}, rate),
 // Fill bucket initially
 for i := 0; i < rate; i++ {
   rl.tokens <- struct{}{}</pre>
 // Refill tokens
  go func() {
    ticker := time.NewTicker(time.Second / time.Duration(rate))
    defer ticker.Stop()
    for range ticker.C {
      select {
      case rl.tokens <- struct{}{}:</pre>
      default: // bucket full
  }()
  return rl
func (rl *RateLimiter) Allow() bool {
  select {
 case <-rl.tokens:
   return true
  default:
    return false
```