Lab 02 Go Concurrency - Hands-on Learning (Simplified)

Duration: 2.5 hours | **Points:** 100

Lab Overview

Objective: Learn Go concurrency basics through simple, hands-on examples.

What You'll Build: Simple programs that run tasks concurrently using goroutines and channels.

Learning Style:
- 📖 **Read** sample code
- 🧠 **Understand** the concept
- ✍️ **Write** your own code based on examples
- 🚀 **Test** and see the results

Setup (5 minutes)

Create New Project

```
mkdir go-concurrency-lab
cd go-concurrency-lab
go mod init concurrency-lab
```

Create `main.go` file in your project directory.

Part 1: Your First Goroutine (In class practical)

📖 Sample Code: Understanding Goroutines

Goroutines allow functions to run concurrently (at the same time).

Create `main.go`:

```
package main

import (
    "fmt"
    "time"
)

// Regular function
func printNumbers() {
```

```
    for i := 1; i <= 5; i++ {
        fmt.Printf("Number: %d\n", i)
        time.Sleep(500 * time.Millisecond)
    }
}

func printLetters() {
    letters := []string{"A", "B", "C", "D", "E"}
    for _, letter := range letters {
        fmt.Printf("Letter: %s\n", letter)
        time.Sleep(500 * time.Millisecond)
    }
}

func main() {
    // WITHOUT goroutines (runs one after another)
    fmt.Println("=== Without Goroutines ===")
    start := time.Now()

    printNumbers()
    printLetters()

    fmt.Printf("Time: %s\n\n", time.Since(start))

    // WITH goroutines (runs at the same time)
    fmt.Println("=== With Goroutines ===")
    start = time.Now()

    go printNumbers()  // Add 'go' to run concurrently
    go printLetters()  // Add 'go' to run concurrently

    time.Sleep(3 * time.Second)  // Wait for them to finish
    fmt.Printf("Time: %s\n", time.Since(start))
}
```

Run: `go run main.go`

Expected Output:

```
=== Without Goroutines ===
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
Letter: A
Letter: B
Letter: C
Letter: D
```

```
Letter: E
Time: 5s

=== With Goroutines ===
Number: 1
Letter: A
Number: 2
Letter: B
Number: 3
Letter: C
Number: 4
Letter: D
Number: 5
Letter: E
Time: 2.5s
```

What happened? With goroutines, both functions run at the same time! ⚡

✍️ Your Task 1.1: Three Counters

Challenge: Make 3 counters count at the same time.

Instructions:

1. Create function `counter(name string, max int)` that:
    - Uses a for loop from 1 to max
    - Prints "Counter [name]: [number]" for each iteration
    - Waits 200ms between each number
    - Prints "Counter [name] finished!" at the end
2. In `main()`:
    - Start 3 counters concurrently using go:
        - Counter A counts to 3
        - Counter B counts to 4
        - Counter C counts to 5
    - Wait 2 seconds for them to finish

Starter Code:

```go
func counter(name string, max int) {
    // TODO: Implement counter function
}

func main() {
    fmt.Println("=== Three Counters ===")
```

```
    // TODO: Start 3 counters concurrently

    time.Sleep(2 * time.Second)
    fmt.Println("All done!")
}
```

Hints: - Use `for i := 1; i <= max; i++ { ... }`
- Use `fmt.Printf("Counter %s: %d\n", name, i)`
- Start goroutine: `go counter("A", 3)`
- Sleep: `time.Sleep(200 * time.Millisecond)`

Expected Output:

```
=== Three Counters ===
Counter A: 1
Counter B: 1
Counter C: 1
Counter A: 2
Counter B: 2
Counter C: 2
Counter A: 3
Counter B: 3
Counter C: 3
Counter A finished!
Counter B: 4
Counter C: 4
Counter B finished!
Counter C: 5
Counter C finished!
All done!
```

✍️ Your Task 1.2: Website Fetcher

Challenge: Simulate fetching data from 4 websites at the same time.

Instructions:

1. Create function `fetchWebsite(name string, delayMs int)` that:
    - Prints "Fetching [name]..."
    - Waits for `delayMs` milliseconds
    - Prints "✓ Got data from [name]"
2. In `main()`:
    - Fetch from 4 websites concurrently with different delays:
        - Google.com: 200ms
        - Facebook.com: 400ms
        - Amazon.com: 300ms

        - Twitter.com: 150ms
    - Measure and print total time taken

Starter Code:

```
func fetchWebsite(name string, delayMs int) {
    // TODO: Implement fetch simulation
}

func main() {
    fmt.Println("=== Fetching Websites ===")
    start := time.Now()

    // TODO: Fetch 4 websites concurrently

    time.Sleep(500 * time.Millisecond)
    fmt.Printf("\nCompleted in: %s\n", time.Since(start))
}
```

Hints:
- Use `fmt.Printf("Fetching %s...\n", name)`
- Convert ms to duration: `time.Duration(delayMs) * time.Millisecond`
- Start goroutine: `go fetchWebsite("Google.com", 200)`

Expected Output:

```
=== Fetching Websites ===
Fetching Google.com...
Fetching Facebook.com...
Fetching Amazon.com...
Fetching Twitter.com...
✓ Got data from Twitter.com
✓ Got data from Google.com
✓ Got data from Amazon.com
✓ Got data from Facebook.com

Completed in: ~400ms
(Sequential would take 1050ms!)
```

Part 2: Channels - Sending Data Between Goroutines (In class practical)

📖 Sample Code: Basic Channel

Channels let goroutines send data to each other.

```
package main
```

```go
import "fmt"

func sendMessage(ch chan string) {
    ch <- "Hello from goroutine!"  // Send to channel
}

func main() {
    // Create a channel
    messageChan := make(chan string)

    // Send message in goroutine
    go sendMessage(messageChan)

    // Receive message
    message := <-messageChan  // Receive from channel
    fmt.Println("Received:", message)
}
```

Output:

```
Received: Hello from goroutine!
```

📖 Sample Code: Multiple Messages

```go
func sendNumbers(ch chan int) {
    for i := 1; i <= 5; i++ {
        ch <- i  // Send number
    }
    close(ch)  // Important! Close when done
}

func main() {
    numberChan := make(chan int)

    go sendNumbers(numberChan)

    // Receive all numbers
    for num := range numberChan {
        fmt.Printf("Got: %d\n", num)
    }
}
```

Output:

```
Got: 1
Got: 2
Got: 3
```

```
Got: 4
Got: 5
```

✍️ Your Task 2.1: Calculator with Channels

Challenge: Calculate sum and average concurrently using channels.

Instructions:

1. Function `calculateSum(numbers []int, result chan int)`:
   - Loop through all numbers and add them together
   - Sleep 100ms each iteration (simulate processing time)
   - Send the sum to the `result` channel
2. Function `calculateAverage(numbers []int, result chan float64)`:
   - Loop through all numbers and calculate sum
   - Sleep 100ms each iteration
   - Calculate average by dividing sum by array length
   - Send the average to the `result` channel
3. In `main()`:
   - Create two channels (one for int, one for float64)
   - Run both functions concurrently using goroutines
   - Receive results from both channels
   - Print the results and time taken

Starter Code:

```go
func calculateSum(numbers []int, result chan int) {
    // TODO: Calculate sum and send to channel
}

func calculateAverage(numbers []int, result chan float64) {
    // TODO: Calculate average and send to channel
}

func main() {
    fmt.Println("=== Concurrent Calculator ===")
    numbers := []int{10, 20, 30, 40, 50}

    // TODO: Create channels, run calculations, receive results

    start := time.Now()
    fmt.Printf("Time: %s\n", time.Since(start))
}
```

Hints: - Create channel: `sumChan := make(chan int)` - Send to channel: `result <- sum` - Receive from channel: `sum := <-sumChan` - Average formula: `float64(sum) / float64(len(numbers))`

Expected Output:

```
=== Concurrent Calculator ===
Numbers: [10 20 30 40 50]
Sum: 150
Average: 30.0
Time: ~500ms (concurrent)
Sequential would take 1000ms
```

✍️ Your Task 2.2: Message Queue

Challenge: Multiple senders send messages to one receiver.

Instructions:

1. Function `sender(name string, messages chan string, count int)`:
    - Loop from 1 to count
    - Create a message string: "Message [i] from [name]"
    - Send the message to the channel
    - Sleep 150ms between messages
2. In `main()`:
    - Create a buffered channel with capacity 10
    - Start 3 senders concurrently:
        - Alice sends 3 messages
        - Bob sends 2 messages
        - Charlie sends 4 messages
    - Receive all 9 messages (3+2+4) and print them

Starter Code:

```go
func sender(name string, messages chan string, count int) {
    // TODO: Loop and send messages
}

func main() {
    fmt.Println("=== Message Queue ===")

    // TODO: Create buffered channel and start senders
    // TODO: Receive all messages
```

```
    fmt.Println("\nAll messages received!")
}
```

Hints:
- Buffered channel: `messages := make(chan string, 10)`
- Create message: `msg := fmt.Sprintf("Message %d from %s", i, name)`
- Send: `messages <- msg` - Total messages: 3 + 2 + 4 = 9

Expected Output:

```
=== Message Queue ===
Message 1 from Alice
Message 1 from Bob
Message 1 from Charlie
Message 2 from Alice
Message 2 from Bob
Message 2 from Charlie
Message 3 from Alice
Message 3 from Charlie
Message 4 from Charlie

All messages received!
```

Part 3: WaitGroup - Proper Synchronization (In class practical)

📖 Sample Code: Why We Need WaitGroup

Using `time.Sleep()` is not reliable. WaitGroup is better!

```go
package main

import (
    "fmt"
    "sync"
    "time"
)

func worker(id int, wg *sync.WaitGroup) {
    defer wg.Done()  // Tell WaitGroup we're done

    fmt.Printf("Worker %d starting\n", id)
    time.Sleep(time.Duration(id*100) * time.Millisecond)
    fmt.Printf("Worker %d done\n", id)
}

func main() {
    fmt.Println("=== WaitGroup Example ===")
```

```
    var wg sync.WaitGroup

    for i := 1; i <= 3; i++ {
        wg.Add(1)  // Add 1 to wait for
        go worker(i, &wg)
    }

    wg.Wait()  // Wait for all to finish
    fmt.Println("All workers completed!")
}
```

Output:

```
=== WaitGroup Example ===
Worker 1 starting
Worker 2 starting
Worker 3 starting
Worker 1 done
Worker 2 done
Worker 3 done
All workers completed!
```

✍ Your Task 3.1: File Downloader

Challenge: Download 5 files concurrently using WaitGroup.

Instructions:

1. Function `downloadFile(filename string, sizeMB int, wg *sync.WaitGroup)`:
    - Use `defer wg.Done()` at the start
    - Print "Downloading [filename] ([size]MB)..."
    - Sleep for (sizeMB * 100) milliseconds to simulate download
    - Print "✓ [filename] complete!"
2. In `main()`:
    - Create a WaitGroup variable
    - Loop through all files in the map
    - For each file: call `wg.Add(1)` and start download goroutine
    - Call `wg.Wait()` to wait for all downloads to complete

Starter Code:

```
func downloadFile(filename string, sizeMB int, wg *sync.WaitGroup) {
    // TODO: Implement download simulation with WaitGroup
}
```

```go
func main() {
    fmt.Println("=== File Downloader ===")
    start := time.Now()

    files := map[string]int{
        "video.mp4": 8, "song.mp3": 4, "photo.jpg": 2,
        "doc.pdf": 5, "archive.zip": 6,
    }

    // TODO: Use WaitGroup to download all files

    fmt.Printf("\n✓ All downloads complete! (%s)\n", time.Since(start))
}
```

Hints:

- Create WaitGroup: `var wg sync.WaitGroup`
- `defer wg.Done()` tells WaitGroup when function finishes
- Sleep duration: `time.Duration(sizeMB*100) * time.Millisecond`
- Loop through map: `for filename, size := range files { ... }`

Expected Output:

```
=== File Downloader ===
Downloading video.mp4 (8MB)...
Downloading song.mp3 (4MB)...
Downloading photo.jpg (2MB)...
Downloading doc.pdf (5MB)...
Downloading archive.zip (6MB)...
✓ photo.jpg complete!
✓ song.mp3 complete!
✓ doc.pdf complete!
✓ archive.zip complete!
✓ video.mp4 complete!

✓ All downloads complete! (~800ms)
Sequential would take 2500ms
```

✍️ Your Task 3.2: Number Processor (Bonus)

Challenge: Process numbers 3 ways concurrently: find evens, odds, and squares.

Instructions:

1. Function `findEvens(numbers []int, result chan []int, wg *sync.WaitGroup)`:
    - Use `defer wg.Done()`

    - Create empty slice for even numbers
    - Loop through numbers, if even (num % 2 == 0), add to slice
    - Sleep 50ms each iteration
    - Send slice to result channel
2. Function `findOdds(numbers []int, result chan []int, wg *sync.WaitGroup)`:
    - Similar to findEvens, but find odd numbers (num % 2 != 0)
3. Function `findSquares(numbers []int, result chan []int, wg *sync.WaitGroup)`:
    - Use `defer wg.Done()`
    - Create empty slice
    - Loop through numbers, calculate square (num * num), add to slice
    - Sleep 50ms each iteration
    - Send slice to result channel
4. In `main()`:
    - Create 3 channels and WaitGroup
    - Call `wg.Add(3)` and start all 3 processors concurrently
    - Create a goroutine that waits for all to finish, then closes channels
    - Receive results from all 3 channels

Starter Code:

```go
func findEvens(numbers []int, result chan []int, wg *sync.WaitGroup) {
    // TODO: Find even numbers
}

func findOdds(numbers []int, result chan []int, wg *sync.WaitGroup) {
    // TODO: Find odd numbers
}

func findSquares(numbers []int, result chan []int, wg *sync.WaitGroup) {
    // TODO: Calculate squares
}

func main() {
    fmt.Println("=== Number Processor ===")
    numbers := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

    // TODO: Create channels, WaitGroup, run processors, collect results

    fmt.Printf("\nTime: %s\n", time.Since(time.Now()))
}
```

Hints:
- Create empty slice: `var evens []int`
- Check even: `if num%2 == 0`

- Append to slice: `evens = append(evens, num)`
- Close channel: `close(evenChan)`

Expected Output:

```
=== Number Processor ===
Numbers: [1 2 3 4 5 6 7 8 9 10]
Evens: [2 4 6 8 10]
Odds: [1 3 5 7 9]
Squares: [1 4 9 16 25 36 49 64 81 100]

Time: ~500ms (concurrent)
Sequential would take ~1500ms
```

Part 4: Select - Handling Multiple Channels (Homework)

📖 Sample Code: Basic Select

Select lets you wait on multiple channels at once.

```go
package main

import (
    "fmt"
    "time"
)

func fast(ch chan string) {
    time.Sleep(100 * time.Millisecond)
    ch <- "Fast response!"
}

func slow(ch chan string) {
    time.Sleep(500 * time.Millisecond)
    ch <- "Slow response!"
}

func main() {
    fmt.Println("=== Select Example ===")

    ch1 := make(chan string)
    ch2 := make(chan string)

    go fast(ch1)
    go slow(ch2)

    // Wait for first response
```

```
    select {
    case msg := <-ch1:
        fmt.Println("Got:", msg)
    case msg := <-ch2:
        fmt.Println("Got:", msg)
    }
}
```

Output:

```
=== Select Example ===
Got: Fast response!
```

✍️ Your Task 4.1: Search Race

Challenge: Two search engines race. Return the fastest result using `select`.

Instructions:

1. Function `searchEngineA(query string, ch chan string)`:
    - Sleep 300ms (simulate search time)
    - Create result string: "Results from Engine A for '[query]'"
    - Send result to channel
2. Function `searchEngineB(query string, ch chan string)`:
    - Sleep 200ms (simulate search time)
    - Create result string: "Results from Engine B for '[query]'"
    - Send result to channel
3. In `main()`:
    - Create 2 channels
    - Start both search engines concurrently
    - Use `select` statement to get whichever result arrives first
    - Print which engine won and the result

Starter Code:

```
func searchEngineA(query string, ch chan string) {
    // TODO: Simulate Engine A search
}

func searchEngineB(query string, ch chan string) {
    // TODO: Simulate Engine B search
}

func main() {
    fmt.Println("=== Search Race ===")
```

```
    query := "golang concurrency"

    // TODO: Create channels, start searches, use select
}
```

Hints: - Create message: `fmt.Sprintf("Results from Engine A for '%s'", query)` - Use `select` with two `case` statements - Engine B should win (faster: 200ms vs 300ms)

Expected Output:

```
=== Search Race ===
🏆 Engine B won! (~200ms)
Results from Engine B for 'golang concurrency'
```

Part 5: Real-World Application - Library Simulation (Homework)

📖 Case Study: International University Library

Scenario:
- Library capacity: 30 students maximum at a time
- Daily visitors: 100 students
- Study duration: Each student studies for 1-4 hours (random)
- Rule: First come, first serve. If full, students wait outside

Simulation Rules:
- 1 second in simulation = 1 hour in real life
- Students arrive throughout the day
- Generate random reader ID for each student
- Track how long library needs to stay open

✍️ Your Task 5.1: Library Simulation (20 min)

Challenge: Simulate one day of library operations with concurrent students.

Instructions:

1. Create struct `Student` with fields:
    - `ID` (int) - reader ID
    - `StudyHours` (int) - how long they will study (1-4 hours)
2. Function `student(id int, studyHours int, library chan bool, wg *sync.WaitGroup)`:
    - Use `defer wg.Done()`
    - Try to enter library (send to channel)

- If library is full, wait
- Print “[Student ID] entered library, will study for [hours] hours”
- Sleep for studyHours seconds (simulate studying)
- Leave library (receive from channel)
- Print “[Student ID] left library after [hours] hours”

3. In `main()`:
   - Create buffered channel with capacity 30 (library seats)
   - Generate 100 students with random study hours (1-4)
   - Track start time and end time
   - Calculate total hours library was open

Starter Code:

```
package main

import (
    "fmt"
    "math/rand"
    "sync"
    "time"
)

type Student struct {
    ID         int
    StudyHours int
}

func student(id int, studyHours int, library chan bool, wg *sync.WaitGroup) {
    // TODO: Implement student behavior
    // - Enter library (library <- true)
    // - Print entry message
    // - Sleep for studyHours seconds
    // - Leave library (<-library)
    // - Print exit message
}

func main() {
    fmt.Println("=== Library Simulation ===")
    fmt.Println("Library capacity: 30 students")
    fmt.Println("Total students today: 100")
    fmt.Println("Simulation: 1 second = 1 hour\n")

    // TODO: Create buffered channel (capacity 30)
    // TODO: Create WaitGroup
    // TODO: Record start time

    // Generate 100 students
```

```
    students := make([]Student, 100)
    for i := 0; i < 100; i++ {
        students[i] = Student{
            ID:         i + 1,
            StudyHours: rand.Intn(4) + 1, // Random 1-4 hours
        }
    }

    // TODO: Start all students as goroutines
    // TODO: Wait for all students to finish
    // TODO: Calculate total time library was open

    fmt.Println("\n=== Simulation Complete ===")
    // TODO: Print results
}
```

Sample Input:

```
// 100 students with random study hours (1-4)
// Library capacity: 30 students
```

Expected Output:

```
=== Library Simulation ===
Library capacity: 30 students
Total students today: 100
Simulation: 1 second = 1 hour

Student 1 entered library, will study for 3 hours
Student 2 entered library, will study for 2 hours
Student 3 entered library, will study for 4 hours
...
Student 30 entered library, will study for 1 hours
Student 31 waiting... (library full)
Student 1 left library after 3 hours
Student 31 entered library, will study for 2 hours
...
Student 100 left library after 3 hours

=== Simulation Complete ===
Total students served: 100
Library was open for: 15 hours
Average wait time: 2.5 hours
Peak occupancy: 30 students
```

Hints: - Buffered channel as semaphore: `library := make(chan bool, 30)` - Enter library: `library <- true` (blocks if full) - Leave library: `<-library` (frees up a seat) - Sleep to simulate time: `time.Sleep(time.Duration(studyHours) * time.Second)` - Use `time.Now()` and `time.Since()` to track duration

✍️ Your Task 5.2: Enhanced Library Statistics (Optional Bonus, +5 points)

Challenge: Add detailed statistics tracking to the library simulation.

Additional Features to Implement:

1. Track waiting time for each student
2. Calculate average wait time
3. Track peak occupancy (maximum students at any moment)
4. Show hourly activity report

Code Structure:

```
type LibraryStats struct {
    TotalStudents    int
    TotalWaitTime    time.Duration
    PeakOccupancy    int
    CurrentOccupancy int
    mu               sync.Mutex
}

func (stats *LibraryStats) RecordEntry() {
    // TODO: Update statistics when student enters
}

func (stats *LibraryStats) RecordExit() {
    // TODO: Update statistics when student exits
}

func (stats *LibraryStats) PrintReport() {
    // TODO: Print detailed statistics
}
```

Enhanced Output:

```
=== Simulation Complete ===
Total students served: 100
Library was open for: 15 hours
Average wait time: 2.5 hours
Peak occupancy: 30 students (occurred at hour 3)
Quietest hour: Hour 12 (5 students)
Total student-hours: 250 hours
Average study duration: 2.5 hours per student
```

Submission Requirements (100 points)

Tasks (90 points)

☐ Task 1.1: Three Counters (10 points) ☐ Task 1.2: Website Fetcher (10 points) ☐ Task 2.1: Calculator with Channels (15 points) ☐ Task 2.2: Message Queue (15 points) ☐ Task 3.1: File Downloader (15 points) ☐ Task 3.2: Number Processor (15 points) ☐ Task 4.1: Search Race (10 points) ☐ **Task 5.1: Library Simulation (10 points) - REQUIRED**

Code Quality (10 points)

☐ Code compiles and runs (4 points) ☐ Uses goroutines correctly (2 points) ☐ Uses channels correctly (2 points) ☐ Clean, readable code (2 points)

Bonus (+5 points)

☐ Task 5.2: Enhanced Library Statistics (5 points)

Testing Checklist

Before submitting, make sure:

1. ✓ Code compiles: `go run main.go`
2. ✓ All goroutines finish
3. ✓ No program hangs (deadlock)
4. ✓ Output matches expected format
5. ✓ Time improvements are visible

Common Mistakes to Avoid

❌ Mistake 1: Not Waiting for Goroutines

```
// Wrong - program ends before goroutine finishes
go printNumbers()
// Program exits immediately!
```

☑ Solution: Use time.Sleep or WaitGroup

```
go printNumbers()
time.Sleep(2 * time.Second)  // Wait
// OR use WaitGroup
```

❌ Mistake 2: Forgetting to Close Channels

```
go sendNumbers(ch)
for num := range ch {  // Will hang forever!
```

```
    fmt.Println(num)
}
```

☑ Solution: Close channel when done

```
func sendNumbers(ch chan int) {
    for i := 1; i <= 5; i++ {
        ch <- i
    }
    close(ch)  // Important!
}
```

✗ Mistake 3: Forgetting defer wg.Done()

```
func worker(wg *sync.WaitGroup) {
    // ... work ...
    // Forgot wg.Done()!
}
```

☑ Solution: Use defer

```
func worker(wg *sync.WaitGroup) {
    defer wg.Done()  // Always runs
    // ... work ...
}
```

Quick Reference

Goroutines

```
go myFunction()          // Run concurrently
time.Sleep(1 * time.Second)  // Wait
```

Channels

```
ch := make(chan int)     // Create
ch <- 42                 // Send
value := <-ch            // Receive
close(ch)                // Close when done
```

WaitGroup

```
var wg sync.WaitGroup
wg.Add(1)                // Add
go worker(&wg)
wg.Wait()                // Wait

// In worker:
defer wg.Done()          // Mark done
```

Select

```
select {
case msg := <-ch1:
    // Handle ch1
case msg := <-ch2:
    // Handle ch2
}
```

Getting Help

During Lab: - 🤖 Ask AI: “How do channels work in Go?” - 👥 Ask classmates - 🙋 Ask instructor

Useful Prompts: - “Explain goroutines simply” - “Channel example in Go” - “How to use WaitGroup” - “Why is my Go program hanging?”

Congratulations! You’ve learned Go concurrency basics! 🎉

Remember: Concurrency makes programs faster by doing multiple things at once!