Taller de Lenguajes de Programación

Programación Orientada a Objetos y Channels en GO

Javier Villegas Lainas

30 de junio 2025

Introducción

Fundamentos de Programación Orientada a Objetos en GO - Interfaces

Go no es un lenguaje orientado a objetos en el sentido tradicional, pero soporta muchos conceptos de POO de manera única:

- **No hay clases**, pero tenemos **structs** con métodos
- **No hay herencia**, pero tenemos **composición** y **embedding**
- **Interfaces implícitas**: no necesitas declarar que implementas una interfaz

La programación orientada a objetos es un paradigma de programación que utiliza la idea de «objetos» para representar datos y métodos. Go no soporta estrictamente la orientación a objetos pero es un lenguaje ligero orientado a objetos. La programación orientada a objetos en Golang es diferente a la de otros lenguajes como C++ o Java debido a los factores que se mencionan a continuación:

1. Structs

Go no soporta tipos personalizados a través de clases sino de structs. Los structs en Golang son tipos definidos por el usuario que contienen sólo el estado y no el comportamiento. Los structs pueden usarse para representar un objeto complejo que contenga más de un par clave-valor. Podemos añadir funciones a la estructura que pueden añadir comportamiento a la misma como se muestra a continuación: Ejemplo:

```
Go

// Golang program to illustrate the

// concept of custom types

package main


import (

    "fmt"

)
```

```
// declaring a struct
type Book struct{

    // defining struct variables
    name string
    author string
    pages int
}

// function to print book details
func (book Book) print_details(){

    fmt.Printf("Book %s was written by %s.", book.name, book.author)
    fmt.Printf("\nIt contains %d pages.\n", book.pages)
}

// main function
func main() {

    // declaring a struct instance
    book1 := Book{"Monster Blood", "R.L.Stine", 131}

    // printing details of book1
    book1.print_details()
```

```
    // modifying book1 details
    book1.name = "Vampire Breath"
    book1.pages = 162

    // printing modified book1
    book1.print_details()

}
```

Al ejecutar dicho código obtenemos lo siguiente:

```
Shell

Book Monster Blood was written by R.L.Stine.
It contains 131 pages.
Book Vampire Breath was written by R.L.Stine.
It contains 162 pages.
```

2. Encapsulación

Significa ocultar datos sensibles a los usuarios. En Go, la encapsulación se implementa poniendo en mayúsculas los campos, métodos y funciones, lo que los hace públicos. Cuando los structs, campos o funciones se hacen públicos, se exportan a nivel de paquete. Algunos ejemplos de miembros públicos y privados son:

```Go
package gfg


// this function is public as
// it begins with a capital letter
func Print_this(){
        // implementation
}


// public struct
type Book struct{


        // public field
        Name string
        // private field, only
        // available in gfg package
        author string
}
```

3. Herencia

Cuando una clase adquiere las propiedades de su superclase, se habla de herencia. Aquí, subclase/clase hija son los términos utilizados para la clase que adquiere propiedades. Para ello, se debe utilizar una estructura para lograr la herencia en Golang. Aquí, los usuarios tienen que componer utilizando structs para formar los otros objetos. Go maneja un concepto **“Composition over Inheritance”**.

“**Composition over Inheritance**” (Composición sobre Herencia) es un **principio de diseño** que recomienda **preferir construir funcionalidades complejas uniendo (componiendo) múltiples estructuras o comportamientos más simples, en lugar de usar herencia de clases**.

En **lenguajes como Java o C#**, la herencia clásica permite que una clase hija herede métodos y propiedades de una clase padre.

Go no tiene herencia clásica; en su lugar, utiliza **composición de structs e interfaces para compartir y reutilizar comportamientos**.
En Go:

- Puedes **incluir (embeber) structs dentro de otros structs**, lo que permite al struct que contiene acceder a los métodos del struct embebido **como si fueran propios**.
- Puedes **combinar múltiples interfaces para que un struct las implemente sin necesidad de herencia**.
- Esto se ajusta al principio KISS de Go (mantener las cosas simples y explícitas).

Veamos esto mejor con un ejemplo:

```go
Go

package main


import (
    "fmt"
    "time"
)


// Logger con un método Log para registrar mensajes
type Logger struct{}
```

```go
func (l Logger) Log(message string) {
    fmt.Println(time.Now().Format("15:04:05"), "LOG:", message)
}

// Notifier define la interfaz con el método Send
type Notifier interface {
    Send(message string)
}

// EmailNotifier compone a Logger embebido
type EmailNotifier struct {
    Logger
    EmailAddress string
}

func (e EmailNotifier) Send(message string) {
    e.Log("Sending email to " + e.EmailAddress)
    fmt.Printf("Email sent to %s: %s\n", e.EmailAddress, message)
}

// SMSNotifier compone a Logger embebido
type SMSNotifier struct {
    Logger
    PhoneNumber string
```

```
}

func (s SMSNotifier) Send(message string) {
    s.Log("Sending SMS to " + s.PhoneNumber)
    fmt.Printf("SMS sent to %s: %s\n", s.PhoneNumber, message)
}

// Función que usa polimorfismo
func SendNotification(n Notifier, message string) {
    n.Send(message)
}

func main() {
    email := EmailNotifier{EmailAddress: "user@example.com"}
    sms := SMSNotifier{PhoneNumber: "+123456789"}

    // Usando polimorfismo
    SendNotification(email, "Your order has been shipped!")
    SendNotification(sms, "Your OTP is 456789")
}
```

4. Interfaces

Las interfaces son tipos que tienen varios métodos. Los objetos que implementan todos los métodos de la interfaz implementan automáticamente la interfaz, es decir, las interfaces se

satisfacen implícitamente. Golang implementa el polimorfismo al tratar objetos de distintos tipos de forma coherente, siempre que se ciñan a una interfaz.

Veamos este concepto con un ejemplo

```go
Go

// Golang program to illustrate the

// concept of interfaces

package main


import (

    "fmt"

)


// defining an interface

type Sport interface{


    // name of sport method

    sportName() string

}


// declaring a struct

type Human struct{


    // defining struct variables

    name string

    sport string
```

```
}

// function to print book details
func (h Human) sportName() string{

    // returning a string value
    return h.name + " plays " + h.sport + "."
}

// main function
func main() {

    // declaring a struct instance
    human1 := Human{"Rahul", "chess"}

    // printing details of human1
    fmt.Println(human1.sportName())

    // declaring another struct instance
    human2 := Human{"Riya", "carrom"}

    // printing details of human2
    fmt.Println(human2.sportName())
}
```

Polimorfismo en la Práctica

En Go, el polimorfismo se implementa principalmente a través de interfaces. Una interfaz define un conjunto de métodos, y cualquier tipo que implemente todos esos métodos se considera una implementación de esa interfaz, permitiendo el polimorfismo.

El polimorfismo, en términos generales, es la capacidad de un objeto de tomar muchas formas. En programación orientada a objetos, esto significa que diferentes tipos de objetos pueden responder al mismo mensaje de manera diferente. En Go, esto se logra a través de interfaces y no mediante herencia como en otros lenguajes.

Cómo funciona en Go:

1. Interfaces: Las interfaces en Go definen un conjunto de métodos que un tipo debe implementar para ser considerado parte de esa interfaz.

2. Implementación: Un tipo implementa una interfaz si define todos los métodos especificados en la interfaz.

3. Polimorfismo: Cuando se utiliza una interfaz como tipo de variable, se puede asignar cualquier tipo que implemente esa interfaz. Esto permite que se invoquen los métodos de la interfaz en la variable, y el comportamiento real dependerá del tipo concreto del objeto asignado.

Vamos viendo con un ejemplo: “Sistema de Procesamiento de Pagos”

```Go
package main


import (
    "fmt"
    "errors"
```

```
)

// Interfaz común para todos los procesadores de pago
type PaymentProcessor interface {
    Process(amount float64) error
    GetFee() float64
}

// Procesador de tarjeta de crédito
type CreditCardProcessor struct {
    CardNumber string
    FeeRate    float64
}

func (cc CreditCardProcessor) Process(amount float64) error {
    if amount <= 0 {
        return errors.New("monto inválido")
    }
    fmt.Printf("💳 Procesando $%.2f con tarjeta ****%s\n",
               amount, cc.CardNumber[len(cc.CardNumber)-4:])
    return nil
}

func (cc CreditCardProcessor) GetFee() float64 {
    return cc.FeeRate
```

```
}

// Procesador PayPal
type PayPalProcessor struct {
    Email string
}

func (pp PayPalProcessor) Process(amount float64) error {
    if amount <= 0 {
        return errors.New("monto inválido")
    }
    fmt.Printf("🅿️ Procesando $%.2f via PayPal (%s)\n", amount, pp.Email)
    return nil
}

func (pp PayPalProcessor) GetFee() float64 {
    return 0.029 // 2.9%
}

// Procesador de criptomonedas
type CryptoProcessor struct {
    WalletAddress string
    Currency      string
}
```

```go
func (cp CryptoProcessor) Process(amount float64) error {
    if amount <= 0 {
        return errors.New("monto inválido")
    }
    fmt.Printf("🪙 Procesando $%.2f en %s (wallet: %s...)\n",
               amount, cp.Currency, cp.WalletAddress[:10])
    return nil
}

func (cp CryptoProcessor) GetFee() float64 {
    return 0.01 // 1%
}

// Función polimórfica que funciona con cualquier procesador
func ProcessOrder(processor PaymentProcessor, amount float64) error {
    fee := amount * processor.GetFee()
    total := amount + fee

    fmt.Printf("Procesando orden por $%.2f (+ $%.2f fee) = $%.2f total\n",
               amount, fee, total)

    return processor.Process(total)
}

// Función que elige el mejor procesador automáticamente
```

```
func ProcessWithBestRate(amount float64, processors []PaymentProcessor) error {
    if len(processors) == 0 {
        return errors.New("no hay procesadores disponibles")
    }

    // Encontrar el procesador con menor comisión
    bestProcessor := processors[0]
    lowestFee := bestProcessor.GetFee()

    for _, processor := range processors[1:] {
        if processor.GetFee() < lowestFee {
            bestProcessor = processor
            lowestFee = processor.GetFee()
        }
    }

    fmt.Printf("Seleccionado procesador con %.1f%% de comisión\n",
lowestFee*100)
    return ProcessOrder(bestProcessor, amount)
}

func main() {
    // Creamos diferentes procesadores
    creditCard := CreditCardProcessor{
        CardNumber: "1234567890123456",
        FeeRate:    0.035, // 3.5%
```

```
    }

    paypal := PayPalProcessor{
        Email: "user@example.com",
    }

    crypto := CryptoProcessor{
        WalletAddress: "1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa",
        Currency:      "BTC",
    }

    // Polimorfismo en acción
    processors := []PaymentProcessor{creditCard, paypal, crypto}

    fmt.Println("=== Procesando con el mejor rate ===")
    ProcessWithBestRate(100.0, processors)

    fmt.Println("\n=== Procesando con cada uno ===")
    for _, processor := range processors {
        ProcessOrder(processor, 50.0)
        fmt.Println()
    }
}
```

Generics in GO

Los genéricos son esencialmente código de plantilla/boilerplate escrito de forma que se pueda utilizar en Go con tipos que se pueden añadir posteriormente. El objetivo principal de los genéricos es lograr una mayor flexibilidad en términos de escritura de código con la adición de menos líneas. El concepto de genéricos existe desde hace mucho tiempo y ha formado parte de muchos lenguajes de programación, como Java, Python y C#, por nombrar algunos.

En palabras de Alexander Stepanov, los genéricos son una forma de crear una fuente incremental de catálogos de código abstracto que pueden ser funciones, algoritmos o estructuras de datos. Como parte de la versión 1.18, Generics se introdujo en Go por primera vez.

Teniendo en cuenta los tipos y funciones, Go trae Generics usando un concepto llamado Parámetros de Tipo. Estos tipos de parámetros pueden ser usados tanto con Funciones como con Estructuras. Vamos a profundizar en la implementación con algunos ejemplos.

Empezaremos escribiendo algo de código repetitivo:

```
Go

package main


func main() {


    var radius1 int = 8

    var radius2 float = 9.5

}
```

El código anterior declara dos variables de tipo int y float en la función principal, que contienen los valores de dos radios de unas circunferencias. En el siguiente paso, declaramos una función genérica que tomará cualquiera de las entradas dependiendo de su tipo y nos devolverá la circunferencia del círculo.

```Go
func generic_circumference [r int | float ](radius r) {
c= 2*3.14*r
fmt.println("The circumference is: ",c)
}
```

Por lo tanto, todo nuestro código se verá algo como esto:

```Go
package main

import "fmt"

func generic_circumference[r int | float32](radius r) {

    c := 2 * 3 * radius
    fmt.Println("The circumference is: ", c)

}

func main() {
    var radius1 int = 8
    var radius2 float32 = 9.5

    generic_circumference(radius1)
    generic_circumference(radius2)
```

```
}
```

El principal factor diferenciador en el código anterior es cómo declaramos una variable r en una lista y proporcionamos la lista de tipos (int, flaot32) que puede tomar esta variable dependiendo de los argumentos que se pasen durante la llamada a la función.

Parameterized Types en Go

En el ejemplo anterior, vimos cómo el uso de funciones genéricas nos permite indicar qué tipos puede aceptar una función. A través de los tipos parametrizados, Go proporciona otra forma de describir los tipos que queremos que acepte nuestra función. Basándonos en el ejemplo anterior, así es como quedaría la parametrización:

```Go
// Parameterized Types
type Radius interface {
    int64 | int8 | float64
}


func generic_circumference[R Radius](radius R){
    var c R
    c = 2 * 3 * radius
    fmt.Println("The circumference is: ", c)
}
```

En este enfoque, primero declaramos una interfaz llamada Radius, que contiene los tipos que podemos pasar a la función, que en el ejemplo anterior son: int64, int8, y float64. El siguiente cambio es cómo declaramos nuestra función genérica. En la declaración de la función, utilizando los corchetes, pasamos una instancia de esta Interfaz que declaramos anteriormente.

R se utiliza en el ejemplo anterior para referirse a cualquiera de los tipos que soporta la interfaz Radius. Si invocamos generic_circumference con un valor float64, entonces R en el contexto de esta función es un valor con tipo float64, si invocamos la función con un int64, entonces R es int64, y así sucesivamente.

Estos dos ejemplos utilizando parámetros de tipo y tipos parametrizados esperamos que te den una visión general de lo que los genéricos pueden hacer en Go. Definitivamente puedes experimentar con diferentes implementaciones basadas en tus necesidades y requerimientos y puedes mejorar la legibilidad de tu código con la adición de esta nueva característica.

Los genéricos proporcionan herramientas eficaces para lograr mayores niveles de abstracción en tu código y también mejoran la reutilización. Dicho esto, es importante identificar los casos de uso adecuados en los que se pueden implementar de forma que se incremente la eficiencia. Los genéricos son todavía un concepto incipiente en el desarrollo de Go y sería interesante ver lo fructíferas que resultan sus implementaciones.

Constraints in GO

En Go, el término «restricciones» se refiere principalmente a las restricciones de tipo utilizadas con los genéricos, y también a las restricciones de construcción (o etiquetas de construcción). Para esta guía solo veremos las restricciones de Tipos de datos con genéricos

Restricciones de Tipo (con Genéricos): Las restricciones de tipo son una parte fundamental de la característica de genéricos de Go (introducida en Go 1.18). **Definen el conjunto de tipos que pueden usarse como argumentos de tipo para una función o tipo genérico**.

Las restricciones de tipo se expresan como tipos de interfaz. Estas interfaces pueden especificar:

- **Métodos:** Cualquier tipo utilizado como argumento de tipo debe implementar los métodos especificados.
- **Conjuntos de tipos:** Utilizando los operadores de unión (|) e intersección (&), puede definir un conjunto de tipos específicos o tipos que satisfagan varias interfaces. Por ejemplo, la interfaz{ int | float64 } obliga a que un parámetro de tipo sea int o float64

```Go
    type Number interface {
```

```
    int | float64 | complex64
}
func Add[T Number](a, b T) T {
    return a + b
}
```

En este ejemplo, Number es una restricción de tipo, que garantiza que Add sólo puede utilizarse con tipos numéricos.

Concurrencia en GO

La concurrencia es el término informático para dividir un único proceso en componentes independientes y especificar cómo estos componentes comparten datos de forma segura. La mayoría de los lenguajes proporcionan concurrencia a través de una librería que utiliza hilos a nivel de sistema operativo que comparten datos intentando adquirir bloqueos. Go es diferente. Su principal modelo de concurrencia, posiblemente la característica más famosa de Go, se basa en Procesos Secuenciales Comunicados (CSP). Este estilo de concurrencia fue descrito en 1978 en un artículo de Tony Hoare, el inventor del algoritmo Quicksort. Los patrones implementados con CSP son tan potentes como los estándar, pero son mucho más fáciles de entender. En este capítulo, va a revisar rápidamente las características que son la columna vertebral de la concurrencia en Go: goroutines, canales y la palabra clave select. Luego verás algunos patrones comunes de concurrencia en Go y aprenderás sobre las situaciones en las que las técnicas de bajo nivel son un mejor enfoque.

GoRoutines

La goroutine es el concepto central del modelo de concurrencia de Go. Para entender las goroutines, definamos un par de términos. El primero es el proceso. Un proceso es una instancia de un programa que está siendo ejecutado por el sistema operativo de un ordenador. El sistema operativo asocia algunos recursos, como la memoria, con el proceso y se asegura de que otros procesos no puedan acceder a ellos. Un proceso se compone de uno o más hilos. Un hilo es una unidad de ejecución a la que el sistema operativo concede un tiempo de ejecución. Los hilos de un proceso comparten el acceso a los recursos. Una CPU puede ejecutar instrucciones de uno o más hilos al mismo tiempo, dependiendo del número de núcleos. Uno de los trabajos de un sistema operativo es programar los hilos en la CPU para asegurarse de que cada proceso (y cada hilo dentro de un proceso) tenga la oportunidad de ejecutarse.

Creación y ejecución de goroutines: Para crear una goroutine, utilice la palabra clave *go* seguida de la llamada a la función que desea ejecutar concurrentemente. He aquí un ejemplo:

```Go
package main
```

```go
import (
    "fmt"
    "time"
)

func printNumbers() {
    for i := 1; i <= 5; i++ {
        fmt.Println(i)
    }
}

func main() {
    go printNumbers() // Creating and running the goroutine
    time.Sleep(time.Second) // Wait to allow goroutine to complete
    fmt.Println("Main function exiting...")
}
```

Sincronización de Goroutines

Veamos el siguiente ejemplo

```go
Go
package main

import (
    "fmt"
```

```
    "time"
)

// Función normal
func sayHello(name string) {
    for i := 0; i < 3; i++ {
        fmt.Printf("Hello from %s (iteration %d)\n", name, i+1)
        time.Sleep(500 * time.Millisecond)
    }
}

func main() {
    fmt.Println("=== Ejecución SECUENCIAL ===")
    sayHello("Alice")
    sayHello("Bob")

    fmt.Println("\n=== Ejecución CONCURRENTE ===")

    // Lanzar goroutines con la palabra clave 'go'
    go sayHello("Charlie")
    go sayHello("Diana")

    // Sin esto, el programa terminaría antes que las goroutines
    time.Sleep(2 * time.Second)
```

```
    fmt.Println("Main function finished")
}
```

En el ejemplo anterior observamos que lanzamos dos goroutines y utilizamos un timer (time.Sleep) para esperar a que ambas goroutines terminen, Sin embargo, esta no es la forma correcta de esperar que las goroutiens finalicen. A veces, es necesario esperar a que todas las goroutines terminen antes de continuar. Los WaitGroups ayudan con la sincronización. En conclusión, un WaitGroup es un contador que permite esperar a que terminen un grupo de goroutines.

```Go
package main

import (
    "fmt"
    "sync"
)

func printNumbers(wg *sync.WaitGroup) {
    defer wg.Done()
    for i := 1; i <= 5; i++ {
        fmt.Println(i)
    }
}

func main() {
    var wg sync.WaitGroup
```

```
    wg.Add(1) // Number of goroutines to wait for
    go printNumbers(&wg)
    wg.Wait() // Wait until all goroutines are done
    fmt.Println("Main function exiting...")
}
```

Veamos un par de ejemplos más

```
Go
package main

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

func worker(id int, wg *sync.WaitGroup) {
    // ¡IMPORTANTE! Marcar como terminado al salir
    defer wg.Done()

    fmt.Printf("Worker %d starting\n", id)

    // Simular trabajo
    time.Sleep(time.Duration(id) * time.Second)
```

```
    fmt.Printf("Worker %d finished\n", id)
}

func main() {
    var wg sync.WaitGroup

    numWorkers := 3

    for i := 1; i <= numWorkers; i++ {
        wg.Add(1) // Incrementar contador
        go worker(i, &wg) // Lanzar goroutine
    }

    fmt.Println("Waiting for workers to finish...")
    wg.Wait() // Esperar a que todas terminen
    fmt.Println("All workers finished!")
}
```

Veamos un ejemplo práctico sobre una descarga concurrente

```
Go

package main

import (
```

```
    "fmt"
    "io"
    "net/http"
    "sync"
    "time"
)

func downloadURL(url string, wg *sync.WaitGroup) {
    defer wg.Done()

    start := time.Now()

    resp, err := http.Get(url)
    if err != nil {
        fmt.Printf("Error downloading %s: %v\n", url, err)
        return
    }
    defer resp.Body.Close()

    // Leer todo el contenido (para simular descarga completa)
    _, err = io.ReadAll(resp.Body)
    if err != nil {
        fmt.Printf("Error reading %s: %v\n", url, err)
        return
    }
```

```
    duration := time.Since(start)
    fmt.Printf("Downloaded %s in %v (Status: %s)\n",
               url, duration, resp.Status)
}

func main() {
    urls := []string{
        "https://httpbin.org/delay/1",
        "https://httpbin.org/delay/2",
        "https://httpbin.org/delay/1",
        "https://httpbin.org/delay/3",
    }

    fmt.Println("=== Descarga SECUENCIAL ===")
    start := time.Now()

    for _, url := range urls {
        downloadURL(url, &sync.WaitGroup{}) // Dummy WaitGroup
    }

    sequentialTime := time.Since(start)
    fmt.Printf("Tiempo secuencial total: %v\n\n", sequentialTime)

    fmt.Println("=== Descarga CONCURRENTE ===")
```

```
    start = time.Now()

    var wg sync.WaitGroup

    for _, url := range urls {
        wg.Add(1)
        go downloadURL(url, &wg)
    }

    wg.Wait()

    concurrentTime := time.Since(start)
    fmt.Printf("Tiempo concurrente total: %v\n", concurrentTime)
    fmt.Printf("Mejora: %.2fx más rápido\n",
               float64(sequentialTime)/float64(concurrentTime))
}
```

Canales en Go

Los canales son una forma de comunicar y sincronizar datos entre goroutines. Proporcionan un medio para que las goroutines envíen y reciban valores de forma segura. Los canales son como tubos o líneas telefónicas que conectan goroutines:

- Una goroutine puede **enviar** datos por el canal
- Otra goroutine puede **recibir** esos datos
- **Comunicación segura** - no race conditions

> ***"Don't communicate by sharing memory; share memory by communicating"***
>
> ***No compartas memoria comunicándote; comunícate compartiendo memoria.***

No se permite el transporte de diferentes tipos de datos desde el mismo canal. **Sintaxis**:

```
var Channel_name chan Type
```

También puede crear un canal mediante la función make() utilizando una declaración abreviada.
Sintaxis:

```
channel_name:= make(chan Type)
```

Veamos un ejemplo

```Go
// Go program to illustrate
// how to create a channel
package main

import "fmt"

func main() {

    // Creating a channel
    // Using var keyword
    var mychannel chan int
    fmt.Println("Value of the channel: ", mychannel)
    fmt.Printf("Type of the channel: %T ", mychannel)

    // Creating a channel using make() function
    mychannel1 := make(chan int)
    fmt.Println("\nValue of the channel1: ", mychannel1)
    fmt.Printf("Type of the channel1: %T ", mychannel1)
}
```

Output

```
Value of the channel:
Type of the channel: chan int
Value of the channel1:  0x432080
Type of the channel1: chan int
```

Enviando y recibiendo data desde un canal

En el lenguaje Go, el canal trabaja con dos operaciones principales una es enviar y otra es recibir, ambas operaciones conocidas colectivamente como comunicación. Y la dirección del operador <- indica si los datos son recibidos o enviados. En el canal, las operaciones de envío y recepción se bloquean hasta que el otro lado no está listo por defecto. Permite a las goroutine sincronizarse entre ellas sin bloqueos explícitos o variables de condición.

1. **Operación de envío:** La operación de envío se utiliza para enviar datos de una goroutine a otra goroutine con la ayuda de un canal. Valores como int, float64, y bool pueden ser enviados fácil y seguramente a través de un canal porque son copiados y no hay riesgo de acceso concurrente accidental al mismo valor. Del mismo modo, las cadenas también son seguras de transferir porque son inmutables. Pero enviar punteros o referencias como un slice, map, etc. a través de un canal no es seguro porque el valor de los punteros o referencias puede cambiar por la goroutine que envía o por la goroutine que recibe al mismo tiempo y el resultado es impredecible. Por lo tanto, cuando uses punteros o referencias en el canal debes asegurarte de que sólo puedan ser accedidos por una goroutine a la vez.

   ```
   Mychannel <- element
   ```

   La declaración anterior indica que los datos (elemento) enviar al canal (Mychannel) con la ayuda de un operador <-.

2. **Operación de recepción:** La operación de recepción se utiliza para recibir los datos enviados por el operador de envío.

   ```
   element := <-Mychannel
   ```

   La declaración anterior indica que el elemento recibe datos del canal(Mychannel). Si el resultado de la sentencia receive no se va a utilizar también es una sentencia válida. También puede escribir una sentencia receive como:

   ```
   <-Mychannel
   ```

```Go
// Go program to illustrate send
// and receive operation
package main

import "fmt"

func myfunc(ch chan int) {

    fmt.Println(234 + <-ch)
}
func main() {
    fmt.Println("start Main method")
    // Creating a channel
    ch := make(chan int)

    go myfunc(ch)
    ch <- 23
    fmt.Println("End Main method")
}
```

Salida

```
start Main method
257
End Main method
```

3. **Cerrar un canal:** También puede cerrar un canal con la ayuda de la función close(). Esta es una función incorporada y establece una bandera que indica que no se enviará más valor a este canal. **Sintaxis:**

```
close()
```

También se puede cerrar el canal utilizando el bucle for range. Aquí, la goroutine receptor puede comprobar el canal está abierto o cerrado con la ayuda de la sintaxis dada:

```
ele, ok:= <- Mychannel
```

Aquí, si el valor de ok es true lo que significa que el canal está abierto entonces, las operaciones de lectura pueden ser realizadas. Y si el valor de es false, lo que significa que el canal está cerrado, las operaciones de lectura no se realizarán.

Veamos esto con un ejemplo:

```
Go

// Go program to illustrate how
// to close a channel using for
// range loop and close function
package main


import "fmt"


// Function
func myfun(mychnl chan string) {


    for v := 0; v < 4; v++ {
```

```
        mychnl <- "GeeksforGeeks"
    }
    close(mychnl)
}

// Main function
func main() {

    // Creating a channel
    c := make(chan string)

    // calling Goroutine
    go myfun(c)

    // When the value of ok is
    // set to true means the
    // channel is open and it
    // can send or receive data
    // When the value of ok is set to
    // false means the channel is closed
    for {
        res, ok := <-c
        if ok == false {
            fmt.Println("Channel Close ", ok)
            break
```

```
        }
        fmt.Println("Channel Open ", res, ok)
    }
}
```

Al ejecutar el programa tendríamos el siguiente resultado

```
Channel Open  GeeksforGeeks true
Channel Open  GeeksforGeeks true
Channel Open  GeeksforGeeks true
Channel Open  GeeksforGeeks true
Channel Close  false
```

Tipos de Canales en GO

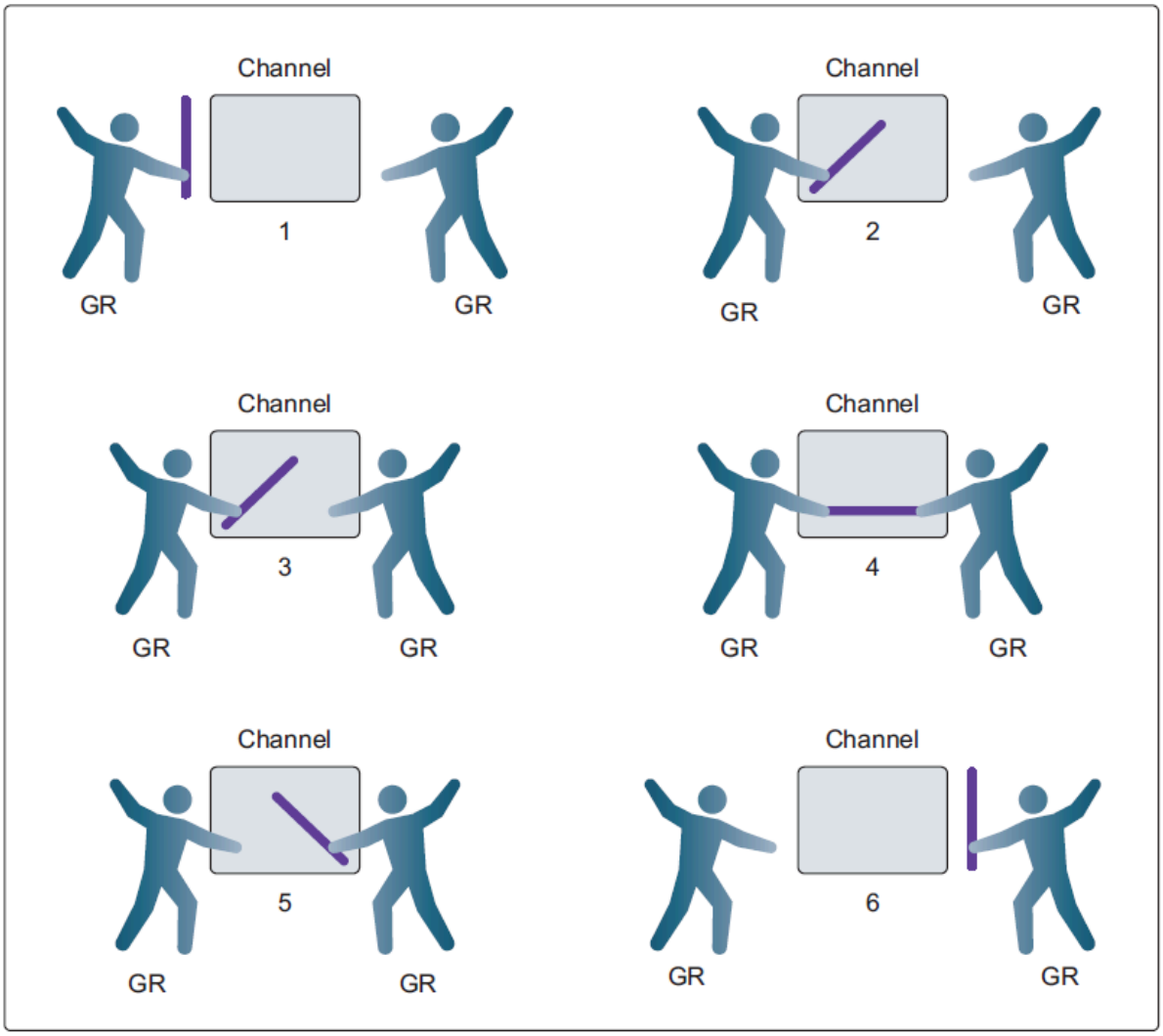
En Go, los canales se clasifican en canales sin búfer y canales con búfer. Los canales sin búfer tienen una capacidad de cero, lo que requiere que el emisor y el receptor estén listos simultáneamente para una operación de envío o recepción exitosa. Los canales con buffer, por otro lado, tienen una capacidad que permite almacenar un número limitado de valores sin bloquear al emisor inmediatamente.

Canales sin Buffer (Unbuffered Channels)

Un canal sin búfer es un tipo de canal por defecto que creamos utilizando el comando make(chan T). T indica el tipo de datos que se pasan a través del canal y el tamaño por defecto del canal es cero. Los canales sin buffer requieren que tanto la rutina go emisora como la receptora estén listas al mismo tiempo para la comunicación. Si una rutina go intenta enviar un valor a un canal sin buffer y no hay ninguna rutina go lista para recibir el valor, la rutina go emisora bloqueará la ejecución hasta que un receptor esté listo.

Un canal sin búfer es un canal sin capacidad para retener ningún valor antes de ser recibido. Estos tipos de canales requieren que tanto la goroutina emisora como la receptora estén listas en el mismo instante antes de que cualquier operación de envío o recepción pueda completarse. Si las dos goroutinas no están listas en el mismo instante, el canal hace esperar primero a la

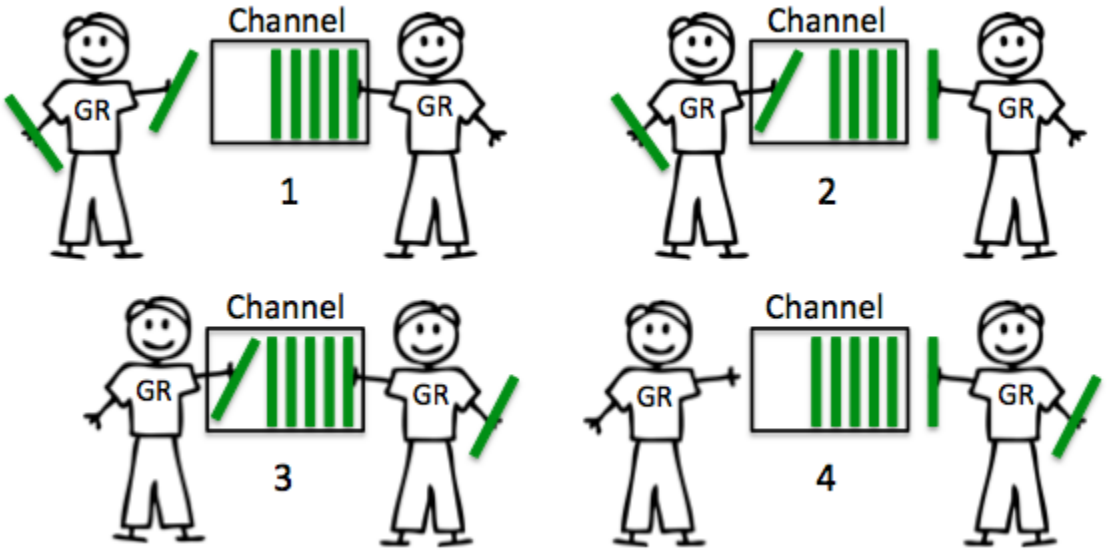
goroutina que realiza su respectiva operación de envío o recepción. La sincronización es inherente a la interacción entre el envío y la recepción en el canal. Uno no puede ocurrir sin el otro.



Canales con Buffer (Buffered Channel)

Los canales con búfer tienen capacidad y, por lo tanto, pueden comportarse de forma un poco diferente. Cuando una goroutine intenta enviar un recurso a un canal con buffer y el canal está lleno, el canal bloqueará la goroutine y la hará esperar hasta que haya un buffer disponible. Si hay espacio en el canal, el envío puede tener lugar inmediatamente y la goroutina puede seguir

adelante. Cuando una goroutina intenta recibir de un canal con buffer y el canal con buffer está vacío, el canal bloqueará la goroutina y la hará esperar hasta que un recurso haya sido enviado.



En el diagrama anterior, vemos un ejemplo de dos goroutines que añaden y eliminan elementos de un canal con buffer de forma independiente. En el paso 1, la goroutina de la derecha está eliminando un recurso del canal o realizando una recepción. En el paso 2, la goroutina de la derecha puede eliminar el recurso independientemente de que la goroutina de la izquierda añada un nuevo recurso al canal. En el paso 3, ambas goroutinas están añadiendo y eliminando un recurso del canal al mismo tiempo y en el paso 4 ambas goroutinas han terminado.

La sincronización aún ocurre dentro de las interacciones de recepciones y envíos, sin embargo cuando la cola tiene disponibilidad de buffer, los envíos no se bloquearán. Las recepciones no se bloquearán cuando haya algo que recibir del canal. Consecuentemente, si el buffer está lleno o si no hay nada que recibir, un canal con buffer se comportará como un canal sin buffer.

A continuación, algunas características importantes:

- **Asíncronos:** Permiten el envío de datos sin disponibilidad inmediata del receptor, hasta la capacidad del canal.
- **Capacidad:** Los canales con búfer tienen una capacidad definida que determina cuántos valores pueden contener antes de bloquearse.
- **Envío sin bloqueo (hasta capacidad):** Si el búfer no está lleno, el envío de datos no bloqueará al emisor.
- **Comportamiento de bloqueo (búfer lleno):** Si el búfer está lleno, el envío de datos bloqueará al emisor hasta que un receptor lea un valor.

Veamos con un ejemplo

```go
package main

import (
    "fmt"
    "time"
)

func demonstrateBuffered() {
    fmt.Println("=== Canal BUFFERED ===")
    buffered := make(chan int, 3)

    // Puedo enviar 3 valores sin bloquear
    buffered <- 1
    buffered <- 2
    buffered <- 3
    fmt.Println("Sent 3 values to buffered channel")

    // Recibir valores
    for i := 0; i < 3; i++ {
        value := <-buffered
        fmt.Printf("Received: %d\n", value)
    }
}
```

```go
func demonstrateUnbuffered() {
    fmt.Println("\n=== Canal UNBUFFERED ===")
    unbuffered := make(chan int)

    // Enviar en goroutine (sino deadlock)
    go func() {
        for i := 1; i <= 3; i++ {
            fmt.Printf("Sending: %d\n", i)
            unbuffered <- i
            time.Sleep(500 * time.Millisecond)
        }
        close(unbuffered)
    }()

    // Recibir en main
    for value := range unbuffered {
        fmt.Printf("Received: %d\n", value)
    }
}

func main() {
    demonstrateBuffered()
    demonstrateUnbuffered()
}
```

Ejemplo Práctico: Canales en Go - Sistema de Notificaciones de Restaurante

El ejemplo incluye:

- Canales SIN buffer → Comunicación crítica (pedidos urgentes, platos listos)
- Canales CON buffer → Comunicación que puede esperar (pedidos normales, notificaciones)

Escenarios desarrollados:

- Pedidos urgentes → Canal sin buffer (mesero DEBE atender inmediatamente)
- Pedidos normales → Canal con buffer (pueden hacer cola)
- Platos listos → Canal sin buffer (mesero DEBE recoger YA)
- Notificaciones → Canal con buffer (pueden acumularse)

```go
Go

package main


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


// ========================================
// MODELOS Y ESTRUCTURAS
// ========================================
```

```
type Order struct {
	ID        int
	Table     int
	Items     []string
	Priority string // "normal", "urgent"
	Timestamp time.Time
}

type Notification struct {
	Type      string // "kitchen", "waiter", "customer"
	Message   string
	Order     Order
	Timestamp time.Time
}

// ========================================
// SIMULADOR DEL RESTAURANTE
// ========================================

type Restaurant struct {
	// CANALES SIN BUFFER (Síncronos)
	// Para comunicación crítica que requiere confirmación inmediata
	urgentOrders   chan Order        // Pedidos urgentes - DEBE ser procesado inmediatamente
	kitchenReady   chan Order        // Plato listo - el mesero DEBE recogerlo ahora
```

```
	// CANALES CON BUFFER (Asíncronos)
	// Para comunicación que puede tolerar demora
	normalOrders   chan Order        // Pedidos normales - pueden esperar en cola
	notifications  chan Notification // Notificaciones generales - pueden acumularse
	customerAlerts chan Notification // Alertas a clientes - pueden agruparse

	// Estadísticas
	metrics struct {
		sync.Mutex
		ordersProcessed   int
		urgentProcessed   int
		normalProcessed   int
		avgProcessingTime time.Duration
		blockedOperations int
	}
}

func NewRestaurant() *Restaurant {
	return &Restaurant{
		// SIN BUFFER - Comunicación síncrona crítica
		urgentOrders: make(chan Order),        // Sin buffer = bloqueo hasta recepción
		kitchenReady: make(chan Order),        // Sin buffer = mesero debe recoger YA
```

```
            // CON BUFFER - Comunicación asíncrona con cola
            normalOrders:   make(chan Order, 10),        // Buffer 10 = hasta
10 pedidos en espera
            notifications:  make(chan Notification, 50), // Buffer 50 = muchas
notificaciones pendientes
            customerAlerts: make(chan Notification, 20), // Buffer 20 =
alertas agrupadas
      }
}

// ==========================================
// GENERADOR DE PEDIDOS
// ==========================================

func (r *Restaurant) orderGenerator(numOrders int, wg *sync.WaitGroup) {
      defer wg.Done()

      dishes := []string{"Pizza Margherita", "Pasta Carbonara", "Caesar Salad",
"Grilled Salmon", "Tiramisu"}

      for i := 1; i <= numOrders; i++ {
            order := Order{
                  ID:       i,
                  Table:    rand.Intn(20) + 1,
                  Items:    []string{dishes[rand.Intn(len(dishes))]},
                  Priority: "normal",
```

```
                Timestamp: time.Now(),
        }

        // 20% de probabilidad de ser urgente
        if rand.Float32() < 0.2 {
                order.Priority = "urgent"
        }

        fmt.Printf("📝 Nuevo pedido #%d (Mesa %d): %s [%s]\n",
                order.ID, order.Table, order.Items[0], order.Priority)

        // DEMOSTRACIÓN DE DIFERENCIA ENTRE CANALES
        if order.Priority == "urgent" {
                fmt.Printf("⚡ Enviando pedido urgente #%d por canal SIN
BUFFER...\n", order.ID)
                start := time.Now()

                // CANAL SIN BUFFER - Esta operación BLOQUEA hasta que
alguien reciba
                r.urgentOrders <- order

                duration := time.Since(start)
                fmt.Printf("✅ Pedido urgente #%d enviado (bloqueó por
%v)\n", order.ID, duration)

        } else {
```

```
                    fmt.Printf("📋 Enviando pedido normal #%d por canal CON
BUFFER...\n", order.ID)
                    start := time.Now()

                    // CANAL CON BUFFER - Esta operación NO bloquea si hay
espacio
                    select {
                    case r.normalOrders <- order:
                        duration := time.Since(start)
                        fmt.Printf("✅ Pedido normal #%d enviado (no bloqueó
- %v)\n", order.ID, duration)
                    default:
                        // Si el buffer está lleno, registramos como
operación bloqueada
                        r.metrics.Lock()
                        r.metrics.blockedOperations++
                        r.metrics.Unlock()
                        fmt.Printf("⚠️ Canal de pedidos normales lleno -
operación bloqueada!\n")
                    }
                }

                // Pausa entre pedidos
                time.Sleep(time.Duration(rand.Intn(300)+100) * time.Millisecond)
        }

        fmt.Println("🏁 Generador de pedidos terminado")
```

```
}

// ========================================
// COCINA (PROCESA PEDIDOS)
// ========================================

func (r *Restaurant) kitchen(id int, wg *sync.WaitGroup) {
	defer wg.Done()
	fmt.Printf("👨‍🍳 Cocinero %d iniciado\n", id)

	for {
		select {
		// ALTA PRIORIDAD: Pedidos urgentes (canal sin buffer)
		case order := <-r.urgentOrders:
			r.processOrder(order, id, "🔥URGENTE🔥")

		// BAJA PRIORIDAD: Pedidos normales (canal con buffer)
		case order := <-r.normalOrders:
			r.processOrder(order, id, "normal")

		// Timeout para salir del worker
		case <-time.After(2 * time.Second):
			fmt.Printf("👨‍🍳 Cocinero %d: No hay más pedidos,
terminando\n", id)
			return
		}
```

```
        }
    }


func (r *Restaurant) processOrder(order Order, cookID int, priority string) {
        start := time.Now()

        fmt.Printf("👨‍🍳 Cocinero %d procesando pedido %s #%d...\n", cookID,
priority, order.ID)

        // Simular tiempo de cocción (urgentes más rápido)
        cookingTime := time.Duration(rand.Intn(1000)+500) * time.Millisecond
        if order.Priority == "urgent" {
                cookingTime = cookingTime / 2 // Urgentes se cocinan más rápido
        }
        time.Sleep(cookingTime)

        processingTime := time.Since(start)
        fmt.Printf("✅ Cocinero %d terminó pedido #%d en %v\n", cookID, order.ID,
processingTime)

        // Actualizar métricas
        r.metrics.Lock()
        r.metrics.ordersProcessed++
        if order.Priority == "urgent" {
                r.metrics.urgentProcessed++
        } else {
```

```
                r.metrics.normalProcessed++
        }
        r.metrics.avgProcessingTime = (r.metrics.avgProcessingTime +
processingTime) / 2
        r.metrics.Unlock()

        // CANAL SIN BUFFER: Plato listo - mesero DEBE recogerlo inmediatamente
        fmt.Printf("🔔 Plato #%d listo - enviando por canal SIN BUFFER (mesero
debe recoger YA)\n", order.ID)
        r.kitchenReady <- order // Esta línea BLOQUEA hasta que un mesero lo
recoja
        fmt.Printf("📤 Plato #%d entregado a mesero\n", order.ID)

        // CANAL CON BUFFER: Notificaciones generales - pueden acumularse
        notification := Notification{
                Type:      "kitchen",
                Message:   fmt.Sprintf("Pedido #%d completado por cocinero %d",
order.ID, cookID),
                Order:     order,
                Timestamp: time.Now(),
        }

        // Esta operación NO bloquea gracias al buffer
        select {
        case r.notifications <- notification:
                // Enviado exitosamente
        default:
```

```
                fmt.Printf("⚠️ Buffer de notificaciones lleno - notificación
perdida\n")
        }
}


// ==========================================
// MESEROS (RECOGEN PLATOS Y SIRVEN)
// ==========================================

func (r *Restaurant) waiter(id int, wg *sync.WaitGroup) {
        defer wg.Done()
        fmt.Printf("🤵 Mesero %d iniciado\n", id)

        for {
                select {
                // CANAL SIN BUFFER: Recoger platos listos (crítico - no puede
esperar)
                case order := <-r.kitchenReady:
                        r.serveOrder(order, id)

                // Timeout para salir
                case <-time.After(3 * time.Second):
                        fmt.Printf("🤵 Mesero %d: No hay más platos, terminando\n",
id)
                        return
                }
```

```
        }
}

func (r *Restaurant) serveOrder(order Order, waiterID int) {
        fmt.Printf("🤵 Mesero %d recogió plato #%d - llevando a mesa %d\n",
                waiterID, order.ID, order.Table)

        // Simular tiempo de servicio
        serviceTime := time.Duration(rand.Intn(500)+200) * time.Millisecond
        time.Sleep(serviceTime)

        fmt.Printf("🍽️ Mesero %d sirvió pedido #%d en mesa %d\n",
                waiterID, order.ID, order.Table)

        // CANAL CON BUFFER: Notificar al cliente (puede esperar)
        customerAlert := Notification{
                Type:      "customer",
                Message:   fmt.Sprintf("Su pedido #%d ha sido servido en mesa %d",
order.ID, order.Table),
                Order:     order,
                Timestamp: time.Now(),
        }

        // Esta operación normalmente NO bloquea gracias al buffer
        select {
        case r.customerAlerts <- customerAlert:
```

```
                fmt.Printf("📱 Notificación enviada a cliente de mesa %d\n",
order.Table)
        default:
                fmt.Printf("⚠️ Buffer de alertas al cliente lleno\n")
        }
}

// ==========================================
// SISTEMA DE NOTIFICACIONES
// ==========================================

func (r *Restaurant) notificationSystem(wg *sync.WaitGroup) {
        defer wg.Done()
        fmt.Printf("📢 Sistema de notificaciones iniciado\n")

        for {
                select {
                // Procesar notificaciones generales
                case notification := <-r.notifications:
                        fmt.Printf("📋 [NOTIFICACIÓN] %s: %s\n",
                                notification.Type, notification.Message)

                // Procesar alertas a clientes
                case alert := <-r.customerAlerts:
                        fmt.Printf("📱 [ALERTA CLIENTE] %s\n", alert.Message)
```

```
                // Timeout para salir
                case <-time.After(5 * time.Second):
                        fmt.Printf("📢 Sistema de notificaciones terminando\n")
                        return
                }
        }
}


// ==========================================
// DEMOSTRACIÓN DE COMPORTAMIENTO DE CANALES
// ==========================================

func demonstrateChannelBehavior() {
        fmt.Println("\n" + "="*60)
        fmt.Println("🧪 DEMOSTRACIÓN: Comportamiento de Canales")
        fmt.Println("="*60)

        // CANAL SIN BUFFER
        fmt.Println("\n1️⃣ CANAL SIN BUFFER (make(chan int)):")
        unbuffered := make(chan int)

        fmt.Println("   - Las operaciones de envío BLOQUEAN hasta que alguien
reciba")
        fmt.Println("   - Las operaciones de recepción BLOQUEAN hasta que alguien
envíe")
        fmt.Println("   - Garantiza sincronización perfecta")
```

```
	go func() {
		fmt.Println("   📤 Goroutine: Enviando valor 42...")
		unbuffered <- 42
		fmt.Println("   ✅ Goroutine: Valor enviado (se desbloqueó cuando main lo recibió)")
	}()

	time.Sleep(100 * time.Millisecond) // Dar tiempo para ver el bloqueo
	fmt.Println("   📥 Main: Recibiendo valor...")
	value := <-unbuffered
	fmt.Printf("   ✅ Main: Recibido %d\n", value)

	// CANAL CON BUFFER
	fmt.Println("\n2️⃣ CANAL CON BUFFER (make(chan int, 3)):")
	buffered := make(chan int, 3)

	fmt.Println("   - Envío NO bloquea si hay espacio en buffer")
	fmt.Println("   - Recepción NO bloquea si hay datos en buffer")
	fmt.Println("   - Permite desacoplar sender y receiver")

	fmt.Println("   📤 Enviando 3 valores sin bloquear...")
	buffered <- 1
	fmt.Println("   ✅ Enviado: 1")
	buffered <- 2
	fmt.Println("   ✅ Enviado: 2")
```

```
	buffered <- 3
	fmt.Println("   ✅ Enviado: 3")

	fmt.Println("   📥 Recibiendo valores...")
	for i := 0; i < 3; i++ {
		val := <-buffered
		fmt.Printf("   ✅ Recibido: %d\n", val)
	}

	// DEMOSTRAR BLOQUEO EN BUFFER LLENO
	fmt.Println("\n3️⃣ BUFFER LLENO - Demostración de bloqueo:")
	fmt.Println("   📤 Llenando buffer (3 elementos)...")
	buffered <- 10
	buffered <- 20
	buffered <- 30
	fmt.Println("   ✅ Buffer lleno (3/3)")

	fmt.Println("   📤 Intentando enviar cuarto elemento (esto
bloqueará)...")
	go func() {
		fmt.Println("   ⏳ Goroutine: Enviando cuarto elemento
(bloqueado)...")
		buffered <- 40
		fmt.Println("   ✅ Goroutine: Cuarto elemento enviado (se
desbloqueó)")
	}()
```

```
        time.Sleep(100 * time.Millisecond) // Mostrar el bloqueo
        fmt.Println("   📥 Main: Recibiendo un elemento para hacer espacio...")
        val := <-buffered
        fmt.Printf("   ✅ Main: Recibido %d (goroutine se desbloqueó)\n", val)

        time.Sleep(50 * time.Millisecond) // Dar tiempo para que termine la goroutine
}

// ==========================================
// FUNCIÓN PRINCIPAL
// ==========================================

func main() {
        fmt.Println("🏪 SISTEMA DE NOTIFICACIONES DEL RESTAURANTE")
        fmt.Println("Demostración práctica de canales con y sin buffer\n")

        // Primero, demostrar comportamiento básico
        demonstrateChannelBehavior()

        fmt.Println("\n" + "="*60)
        fmt.Println("🏪 SIMULACIÓN DEL RESTAURANTE")
        fmt.Println("="*60)

        restaurant := NewRestaurant()
```

```
var wg sync.WaitGroup

// Configuración de la simulación
numOrders := 8
numCooks := 2
numWaiters := 2

fmt.Printf("Configuración: %d pedidos, %d cocineros, %d meseros\n\n",
	numOrders, numCooks, numWaiters)

// Iniciar cocineros
for i := 1; i <= numCooks; i++ {
	wg.Add(1)
	go restaurant.kitchen(i, &wg)
}

// Iniciar meseros
for i := 1; i <= numWaiters; i++ {
	wg.Add(1)
	go restaurant.waiter(i, &wg)
}

// Iniciar sistema de notificaciones
wg.Add(1)
go restaurant.notificationSystem(&wg)
```

```
        // Generar pedidos
        wg.Add(1)
        go restaurant.orderGenerator(numOrders, &wg)

        // Esperar a que termine todo
        wg.Wait()

        // Mostrar estadísticas finales
        fmt.Println("\n" + "="*60)
        fmt.Println("📊 ESTADÍSTICAS FINALES")
        fmt.Println("="*60)
        restaurant.metrics.Lock()
        fmt.Printf("Pedidos procesados: %d\n",
restaurant.metrics.ordersProcessed)
        fmt.Printf("  - Urgentes: %d\n", restaurant.metrics.urgentProcessed)
        fmt.Printf("  - Normales: %d\n", restaurant.metrics.normalProcessed)
        fmt.Printf("Tiempo promedio de procesamiento: %v\n",
restaurant.metrics.avgProcessingTime)
        fmt.Printf("Operaciones bloqueadas por buffer lleno: %d\n",
restaurant.metrics.blockedOperations)
        restaurant.metrics.Unlock()

        // Mostrar estado de los canales
        fmt.Println("\n📈 ESTADO DE LOS CANALES:")
        fmt.Printf("Canal pedidos normales (con buffer): %d/%d elementos\n",
```

```
            len(restaurant.normalOrders), cap(restaurant.normalOrders))
        fmt.Printf("Canal notificaciones (con buffer): %d/%d elementos\n",
            len(restaurant.notifications), cap(restaurant.notifications))
        fmt.Printf("Canal alertas cliente (con buffer): %d/%d elementos\n",
            len(restaurant.customerAlerts), cap(restaurant.customerAlerts))

        fmt.Println("\n🎉 Simulación completada exitosamente!")

        // Resumen educativo
        fmt.Println("\n" + "="*60)
        fmt.Println("📚 RESUMEN EDUCATIVO")
        fmt.Println("="*60)
        fmt.Println("🔹 CANALES SIN BUFFER (Síncronos):")
        fmt.Println("   • Usados para: urgentOrders, kitchenReady")
        fmt.Println("   • Garantizan: Sincronización perfecta")
        fmt.Println("   • Bloquean: Hasta que hay sender Y receiver")
        fmt.Println("   • Ideal para: Comunicación crítica")

        fmt.Println("\n🔸 CANALES CON BUFFER (Asíncronos):")
        fmt.Println("   • Usados para: normalOrders, notifications,
customerAlerts")
        fmt.Println("   • Permiten: Desacoplar sender/receiver")
        fmt.Println("   • No bloquean: Si hay espacio/datos en buffer")
        fmt.Println("   • Ideal para: Comunicación no crítica, mejorar
throughput")
}
```