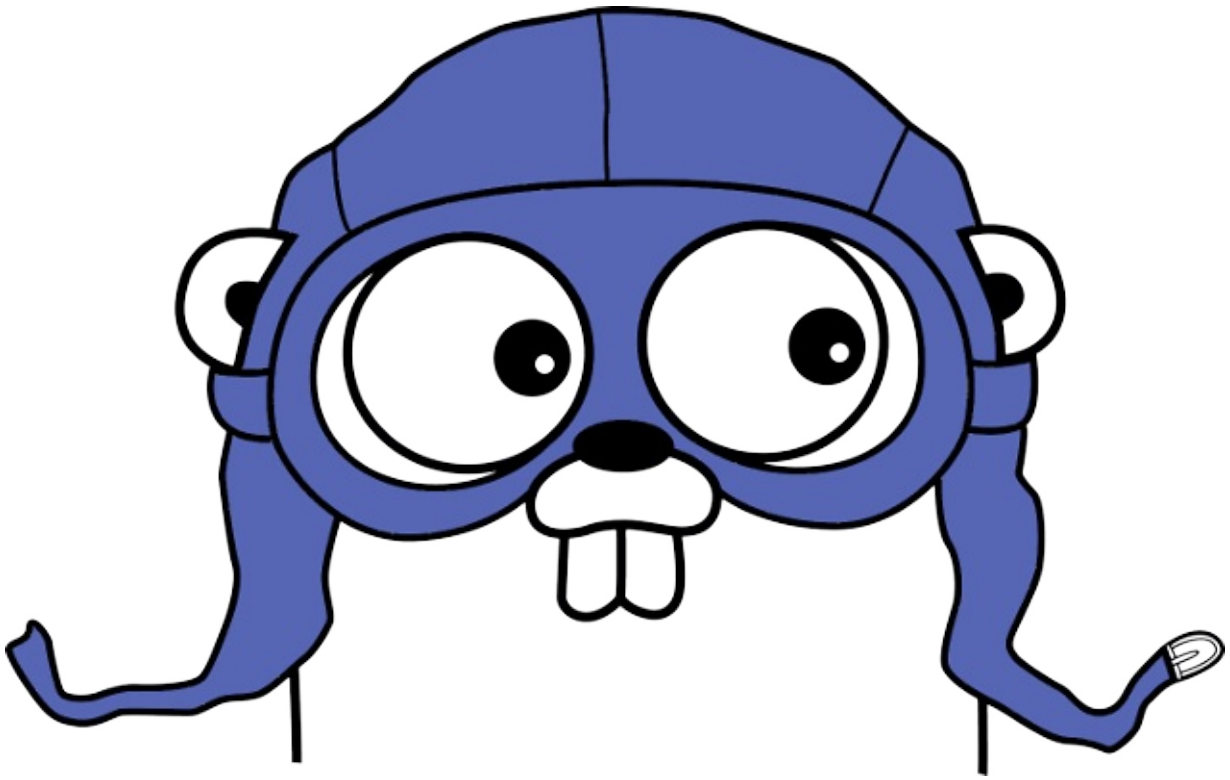
# Table of Contents



# Go Patterns


A curated collection of idiomatic design & application patterns for Go language.

## Creational Patterns

| Pattern | Description | Status |
|---|---|---|
| Abstract Factory | Provides an interface for creating families of releated objects | ✘ |
| Builder | Builds a complex object using simple objects | ✔ |
| Factory Method | Defers instantiation of an object to a specialized function for creating instances | ✔ |
| Object Pool | Instantiates and maintains a group of objects instances of the same type | ✔ |
| Singleton | Restricts instantiation of a type to one object | ✔ |

## Structural Patterns

| Pattern | Description | Status |
|---|---|---|
| Bridge | Decouples an interface from its implementation so that the two can vary independently | ✘ |
| Composite | Encapsulates and provides access to a number of different objects | ✘ |
| Decorator | Adds behavior to an object, statically or dynamically | ✔ |
| Facade | Uses one type as an API to a number of others | ✘ |
| Flyweight | Reuses existing instances of objects with similar/identical state to minimize resource usage | ✘ |
| Proxy | Provides a surrogate for an object to control it's actions | ✔ |

# Behavioral Patterns

| Pattern | Description | Status |
|---|---|---|
| Chain of Responsibility | Avoids coupling a sender to receiver by giving more than object a chance to handle the request | ✘ |
| Command | Bundles a command and arguments to call later | ✘ |
| Mediator | Connects objects and acts as a proxy | ✘ |
| Memento | Generate an opaque token that can be used to go back to a previous state | ✘ |
| Observer | Provide a callback for notification of events/changes to data | ✔ |
| Registry | Keep track of all subclasses of a given class | ✘ |
| State | Encapsulates varying behavior for the same object based on its internal state | ✘ |
| Strategy | Enables an algorithm's behavior to be selected at runtime | ✔ |
| Template | Defines a skeleton class which defers some methods to subclasses | ✘ |
| Visitor | Separates an algorithm from an object on which it operates | ✘ |

# Synchronization Patterns

| Pattern | Description | Status |
|---|---|---|
| Condition Variable | Provides a mechanism for threads to temporarily give up access in order to wait for some condition | ✘ |
| Lock/Mutex | Enforces mutual exclusion limit on a resource to gain exclusive access | ✘ |
| Monitor | Combination of mutex and condition variable patterns | ✘ |
| Read-Write Lock | Allows parallel read access, but only exclusive access on write operations to a resource | ✘ |
| Semaphore | Allows controlling access to a common resource | ✔ |

# Concurrency Patterns

| Pattern | Description | Status |
|---|---|---|
| N-Barrier | Prevents a process from proceeding until all N processes reach to the barrier | ✘ |
| Bounded Parallelism | Completes large number of independent tasks with resource limits | ✔ |
| Broadcast | Transfers a message to all recipients simultaneously | ✘ |
| Coroutines | Subroutines that allow suspending and resuming execution at certain locations | ✘ |
| Generators | Yields a sequence of values one at a time | ✔ |
| Reactor | Demultiplexes service requests delivered concurrently to a service handler and dispatches them syncronously to the associated request handlers | ✘ |
| Parallelism | Completes large number of independent tasks | ✔ |
| Producer Consumer | Separates tasks from task executions | ✘ |

# Messaging Patterns

| Pattern | Description | Status |
|---|---|---|
| Fan-In | Funnels tasks to a work sink (e.g. server) | ✔ |
| Fan-Out | Distributes tasks among workers (e.g. producer) | ✔ |
| Futures & Promises | Acts as a place-holder of a result that is initially unknown for synchronization purposes | ✘ |
| Publish/Subscribe | Passes information to a collection of recipients who subscribed to a topic | ✔ |
| Push & Pull | Distributes messages to multiple workers, arranged in a pipeline | ✘ |

# Stability Patterns

| Pattern | Description | Status |
|---|---|---|
| Bulkheads | Enforces a principle of failure containment (i.e. prevents cascading failures) | ✘ |
| Circuit-Breaker | Stops the flow of the requests when requests are likely to fail | ✔ |
| Deadline | Allows clients to stop waiting for a response once the probability of response becomes low (e.g. after waiting 10 seconds for a page refresh) | ✘ |
| Fail-Fast | Checks the availability of required resources at the start of a request and fails if the requirements are not satisfied | ✘ |
| Handshaking | Asks a component if it can take any more load, if it can't, the request is declined | ✘ |
| Steady-State | For every service that accumulates a resource, some other service must recycle that resource | ✘ |

# Profiling Patterns

| Pattern | Description | Status |
|---|---|---|
| Timing Functions | Wraps a function and logs the execution | ✔ |

# Idioms

| Pattern | Description | Status |
| --- | --- | --- |
| Functional Options | Allows creating clean APIs with sane defaults and idiomatic overrides | ✔ |

# Anti-Patterns

| Pattern | Description | Status |
| --- | --- | --- |
| Cascading Failures | A failure in a system of interconnected parts in which the failure of a part causes a domino effect | ✘ |

# Builder Pattern

Builder pattern separates the construction of a complex object from its representation so that the same construction process can create different representations.

In Go, normally a configuration struct is used to achieve the same behavior, however passing a struct to the builder method fills the code with boilerplate `if cfg.Field != nil {...}` checks.

# Implementation

```
package car

type Speed float64

const (
    MPH Speed = 1
    KPH       = 1.60934
)

type Color string

const (
    BlueColor  Color = "blue"
    GreenColor       = "green"
    RedColor         = "red"
)

type Wheels string

const (
    SportsWheels Wheels = "sports"
    SteelWheels         = "steel"
)

type Builder interface {
    Color(Color) Builder
    Wheels(Wheels) Builder
    TopSpeed(Speed) Builder
    Build() Interface
}

type Interface interface {
    Drive() error
    Stop() error
}
```

# Usage

```
assembly := car.NewBuilder().Paint(car.RedColor)

familyCar := assembly.Wheels(car.SportsWheels).TopSpeed(50 * car.MPH).Build()
familyCar.Drive()

sportsCar := assembly.Wheels(car.SteelWheels).TopSpeed(150 * car.MPH).Build()
sportsCar.Drive()
```

# Factory Method Pattern

Factory method creational design pattern allows creating objects without having to specify the exact type of the object that will be created.

## Implementation

The example implementation shows how to provide a data store with different backends such as in-memory, disk storage.

### Types

```
package data

import "io"

type Store interface {
    Open(string) (io.ReadWriteCloser, error)
}
```

### Different Implementations

```
package data

type StorageType int

const (
    DiskStorage StorageType = 1 << iota
    TempStorage
    MemoryStorage
)

func NewStore(t StorageType) Store {
    switch t {
    case MemoryStorage:
        return newMemoryStorage( /*...*/ )
    case DiskStorage:
        return newDiskStorage( /*...*/ )
    default:
        return newTempStorage( /*...*/ )
    }
}
```

# Usage

With the factory method, the user can specify the type of storage they want.

```
s, _ := data.NewStore(data.MemoryStorage)
f, _ := s.Open("file")

n, _ := f.Write([]byte("data"))
defer f.Close()
```

# Object Pool Pattern

The object pool creational design pattern is used to prepare and keep multiple instances according to the demand expectation.

## Implementation

```
package pool

type Pool chan *Object

func New(total int) *Pool {
    p := make(Pool, total)

    for i := 0; i < total; i++ {
        p <- new(Object)
    }

    return &p
}
```

## Usage

Given below is a simple lifecycle example on an object pool.

```
p := pool.New(2)

select {
case obj := <-p:
    obj.Do( /*...*/ )

    p <- obj
default:
    // No more objects left – retry later or fail
    return
}
```

## Rules of Thumb

- Object pool pattern is useful in cases where object initialization is more expensive than

the object maintenance.
- If there are spikes in demand as opposed to a steady demand, the maintenance overhead might overweigh the benefits of an object pool.
- It has positive effects on performance due to objects being initialized beforehand.

# Singleton Pattern

Singleton creational design pattern restricts the instantiation of a type to a single object.

## Implementation

```
package singleton

type singleton map[string]string

var (
    once sync.Once

    instance singleton
)

func New() singleton {
    once.Do(func() {
        instance = make(singleton)
    })

    return instance
}
```

## Usage

```
s := singleton.New()

s["this"] = "that"

s2 := singleton.New()

fmt.Println("This is ", s2["this"])
// This is that
```

## Rules of Thumb

- Singleton pattern represents a global state and most of the time reduces testability.

# Decorator Pattern

Decorator structural pattern allows extending the function of an existing object dynamically without altering its internals.

Decorators provide a flexible method to extend functionality of objects.

## Implementation

`LogDecorate` decorates a function with the signature `func(int) int` that manipulates integers and adds input/output logging capabilities.

```
type Object func(int) int

func LogDecorate(fn Object) Object {
    return func(n int) int {
        log.Println("Starting the execution with the integer", n)

        result := fn(n)

        log.Println("Execution is completed with the result", result)

        return result
    }
}
```

## Usage

```
func Double(n int) int {
    return n * 2
}

f := LogDecorate(Double)

f(5)
// Starting execution with the integer 5
// Execution is completed with the result 10
```

## Rules of Thumb

- Unlike Adapter pattern, the object to be decorated is obtained by **injection**.
- Decorators should not alter the interface of an object.

# Proxy Pattern

The proxy pattern provides an object that controls access to another object, intercepting all calls.

## Implementation

The proxy could interface to anything: a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate.

Short idea of implementation:

```
// To use proxy and to object they must implement same methods
type IObject interface {
    ObjDo(action string)
}

// Object represents real objects which proxy will delegate data
type Object struct {
    action string
}

// ObjDo implements IObject interface and handel's all logic
func (obj *Object) ObjDo(action string) {
    // Action behavior
    fmt.Printf("I can, %s", action)
}

// ProxyObject represents proxy object with intercepts actions
type ProxyObject struct {
    object *Object
}

// ObjDo are implemented IObject and intercept action before send in real Object
func (p *ProxyObject) ObjDo(action string) {
    if p.object == nil {
        p.object = new(Object)
    }
    if action == "Run" {
        p.object.ObjDo(action) // Prints: I can, Run
    }
}
```

## Usage

More complex usage of proxy as example: User creates "Terminal" authorizes and PROXY send execution command to real Terminal object See proxy/main.go or view in the Playground.

# Observer Pattern

The observer pattern allows a type instance to "publish" events to other type instances ("observers") who wish to be updated when a particular event occurs.

## Implementation

In long-running applications—such as webservers—instances can keep a collection of observers that will receive notification of triggered events.

Implementations vary, but interfaces can be used to make standard observers and notifiers:

```
type (
    // Event defines an indication of a point-in-time occurrence.
    Event struct {
        // Data in this case is a simple int, but the actual
        // implementation would depend on the application.
        Data int64
    }

    // Observer defines a standard interface for instances that wish to list for
    // the occurrence of a specific event.
    Observer interface {
        // OnNotify allows an event to be "published" to interface implementations.
        // In the "real world", error handling would likely be implemented.
        OnNotify(Event)
    }

    // Notifier is the instance being observed. Publisher is perhaps another decent
    // name, but naming things is hard.
    Notifier interface {
        // Register allows an instance to register itself to listen/observe
        // events.
        Register(Observer)
        // Deregister allows an instance to remove itself from the collection
        // of observers/listeners.
        Deregister(Observer)
        // Notify publishes new events to listeners. The method is not
        // absolutely necessary, as each implementation could define this itself
        // without losing functionality.
        Notify(Event)
    }
)
```

# Usage

For usage, see observer/main.go or view in the Playground.

# Strategy Pattern

Strategy behavioral design pattern enables an algorithm's behavior to be selected at runtime.

It defines algorithms, encapsulates them, and uses them interchangeably.

# Implementation

Implementation of an interchangeable operator object that operates on integers.

```
type Operator interface {
    Apply(int, int) int
}

type Operation struct {
    Operator Operator
}

func (o *Operation) Operate(leftValue, rightValue int) int {
    return o.Operator.Apply(leftValue, rightValue)
}
```

# Usage

## Addition Operator

```
type Addition struct{}

func (Addition) Apply(lval, rval int) int {
    return lval + rval
}
```

```
add := Operation{Addition{}}
add.Operate(3, 5) // 8
```

## Multiplication Operator

```
type Multiplication struct{}

func (Multiplication) Apply(lval, rval int) int {
    return lval * rval
}
```

```
mult := Operation{Multiplication{}}

mult.Operate(3, 5) // 15
```

# Rules of Thumb

- Strategy pattern is similar to Template pattern except in its granularity.
- Strategy pattern lets you change the guts of an object. Decorator pattern lets you change the skin.

# Semaphore Pattern

A semaphore is a synchronization pattern/primitive that imposes mutual exclusion on a limited number of resources.

## Implementation

```
package semaphore

var (
    ErrNoTickets      = errors.New("semaphore: could not aquire semaphore")
    ErrIllegalRelease = errors.New("semaphore: can't release the semaphore without acq
uiring it first")
)

// Interface contains the behavior of a semaphore that can be acquired and/or released.

type Interface interface {
    Acquire() error
    Release() error
}

type implementation struct {
    sem     chan struct{}
    timeout time.Duration
}

func (s *implementation) Acquire() error {
    select {
    case s.sem <- struct{}{}:
        return nil
    case <-time.After(s.timeout):
        return ErrNoTickets
    }
}

func (s *implementation) Release() error {
    select {
    case _ = <-s.sem:
        return nil
    case <-time.After(s.timeout):
        return ErrIllegalRelease
    }

    return nil
}

func New(tickets int, timeout time.Duration) Interface {
    return &implementation{
        sem:     make(chan struct{}, tickets),
        timeout: timeout,
    }
}
```

# Usage

## Semaphore with Timeouts

```
tickets, timeout := 1, 3*time.Second
s := semaphore.New(tickets, timeout)

if err := s.Acquire(); err != nil {
    panic(err)
}

// Do important work

if err := s.Release(); err != nil {
    panic(err)
}
```

## Semaphore without Timeouts (Non-Blocking)

```
tickets, timeout := 0, 0
s := semaphore.New(tickets, timeout)

if err := s.Acquire(); err != nil {
    if err != semaphore.ErrNoTickets {
        panic(err)
    }

    // No tickets left, can't work :(
    os.Exit(1)
}
```

# Bounded Parallelism Pattern

Bounded parallelism is similar to parallelism, but allows limits to be placed on allocation.

# Implementation and Example

An example showing implementation and usage can be found in bounded_parallelism.go.

# Generator Pattern

Generators) yields a sequence of values one at a time.

# Implementation

```
func Count(start int, end int) chan int {
    ch := make(chan int)

    go func(ch chan int) {
        for i := start; i <= end ; i++ {
            // Blocks on the operation
            ch <- i
        }

        close(ch)
    }(ch)

    return ch
}
```

# Usage

```
fmt.Println("No bottles of beer on the wall")

for i := range Count(1, 99) {
    fmt.Println("Pass it around, put one up,", i, "bottles of beer on the wall")
    // Pass it around, put one up, 1 bottles of beer on the wall
    // Pass it around, put one up, 2 bottles of beer on the wall
    // ...
    // Pass it around, put one up, 99 bottles of beer on the wall
}

fmt.Println(100, "bottles of beer on the wall")
```

# Parallelism Pattern

Parallelism allows multiple "jobs" or tasks to be run concurrently and asynchronously.

# Implementation and Example

An example showing implementation and usage can be found in parallelism.go.

# Fan-In Messaging Patterns

Fan-In is a messaging pattern used to create a funnel for work amongst workers (clients: source, server: destination).

We can model fan-in using the Go channels.

```
// Merge different channels in one channel
func Merge(cs ...<-chan int) <-chan int {
    var wg sync.WaitGroup

    out := make(chan int)

    // Start an send goroutine for each input channel in cs. send
    // copies values from c to out until c is closed, then calls wg.Done.
    send := func(c <-chan int) {
        for n := range c {
            out <- n
        }
        wg.Done()
    }

    wg.Add(len(cs))
    for _, c := range cs {
        go send(c)
    }

    // Start a goroutine to close out once all the send goroutines are
    // done.  This must start after the wg.Add call.
    go func() {
        wg.Wait()
        close(out)
    }()
    return out
}
```

The `Merge` function converts a list of channels to a single channel by starting a goroutine for each inbound channel that copies the values to the sole outbound channel.

Once all the output goroutines have been started, `Merge` a goroutine is started to close the main channel.

# Fan-Out Messaging Pattern

Fan-Out is a messaging pattern used for distributing work amongst workers (producer: source, consumers: destination).

We can model fan-out using the Go channels.

```
// Split a channel into n channels that receive messages in a round-robin fashion.
func Split(ch <-chan int, n int) []<-chan int {
    cs := make([]chan int)
    for i := 0; i < n; i++ {
        cs = append(cs, make(chan int))
    }

    // Distributes the work in a round robin fashion among the stated number
    // of channels until the main channel has been closed. In that case, close
    // all channels and return.
    distributeToChannels := func(ch <-chan int, cs []chan<- int) {
        // Close every channel when the execution ends.
        defer func(cs []chan<- int) {
            for _, c := range cs {
                close(c)
            }
        }(cs)

        for {
            for _, c := range cs {
                select {
                case val, ok := <-ch:
                    if !ok {
                        return
                    }

                    c <- val
                }
            }
        }
    }

    go distributeToChannels(ch, cs)

    return cs
}
```

The `Split` function converts a single channel into a list of channels by using a goroutine to copy received values to channels in the list in a round-robin fashion.

# Publish & Subscribe Messaging Pattern

Publish-Subscribe is a messaging pattern used to communicate messages between different components without these components knowing anything about each other's identity.

It is similar to the Observer behavioral design pattern. The fundamental design principals of both Observer and Publish-Subscribe is the decoupling of those interested in being informed about `Event Messages` from the informer (Observers or Publishers). Meaning that you don't have to program the messages to be sent directly to specific receivers.

To accomplish this, an intermediary, called a "message broker" or "event bus", receives published messages, and then routes them on to subscribers.

There are three components **messages**, **topics**, **users**.

```
type Message struct {
    // Contents
}


type Subscription struct {
    ch chan<- Message

    Inbox chan Message
}

func (s *Subscription) Publish(msg Message) error {
    if _, ok := <-s.ch; !ok {
        return errors.New("Topic has been closed")
    }

    s.ch <- msg

    return nil
}
```

```
type Topic struct {
    Subscribers    []Session
    MessageHistory []Message
}

func (t *Topic) Subscribe(uid uint64) (Subscription, error) {
    // Get session and create one if it's the first

    // Add session to the Topic & MessageHistory

    // Create a subscription
}

func (t *Topic) Unsubscribe(Subscription) error {
    // Implementation
}

func (t *Topic) Delete() error {
    // Implementation
}
```

```
type User struct {
    ID uint64
    Name string
}

type Session struct {
    User User
    Timestamp time.Time
}
```

# Improvements

Events can be published in a parallel fashion by utilizing stackless goroutines.

Performance can be improved by dealing with straggler subscribers by using a buffered inbox and you stop sending events once the inbox is full.

# Circuit Breaker Pattern

Similar to electrical fuses that prevent fires when a circuit that is connected to the electrical grid starts drawing a high amount of power which causes the wires to heat up and combust, the circuit breaker design pattern is a fail-first mechanism that shuts down the circuit, request/response relationship or a service in the case of software development, to prevent bigger failures.

**Note:** The words "circuit" and "service" are used synonymously throught this document.

# Implementation

Below is the implementation of a very simple circuit breaker to illustrate the purpose of the circuit breaker design pattern.

## Operation Counter

`circuit.Counter` is a simple counter that records success and failure states of a circuit along with a timestamp and calculates the consecutive number of failures.

```
package circuit

import (
    "time"
)

type State int

const (
    UnknownState State = iota
    FailureState
    SuccessState
)

type Counter interface {
    Count(State)
    ConsecutiveFailures() uint32
    LastActivity() time.Time
    Reset()
}
```

## Circuit Breaker

Circuit is wrapped using the `circuit.Breaker` closure that keeps an internal operation counter. It returns a fast error if the circuit has failed consecutively more than the specified threshold. After a while it retries the request and records it.

**Note:** Context type is used here to carry deadlines, cancelation signals, and other request-scoped values across API boundaries and between processes.

```
package circuit

import (
    "context"
    "time"
)

type Circuit func(context.Context) error

func Breaker(c Circuit, failureThreshold uint32) Circuit {
    cnt := NewCounter()

    return func(ctx context) error {
        if cnt.ConsecutiveFailures() >= failureThreshold {
            canRetry := func(cnt Counter) {
                backoffLevel := Cnt.ConsecutiveFailures() - failureThreshold

                // Calculates when should the circuit breaker resume propagating reque
sts
                // to the service
                shouldRetryAt := cnt.LastActivity().Add(time.Seconds * 2 << backoffLev
el)

                return time.Now().After(shouldRetryAt)
            }

            if !canRetry(cnt) {
                // Fails fast instead of propagating requests to the circuit since
                // not enough time has passed since the last failure to retry
                return ErrServiceUnavailable
            }
        }

        // Unless the failure threshold is exceeded the wrapped service mimics the
        // old behavior and the difference in behavior is seen after consecutive failu
res
        if err := c(ctx); err != nil {
            cnt.Count(FailureState)
            return err
        }

        cnt.Count(SuccessState)
        return nil
    }
}
```

# Related Works

- sony/gobreaker is a well-tested and intuitive circuit breaker implementation for real-

world use cases.

# Timing Functions

When optimizing code, sometimes a quick and dirty time measurement is required as opposed to utilizing profiler tools/frameworks to validate assumptions.

Time measurements can be performed by utilizing `time` package and `defer` statements.

## Implementation

```
package profile

import (
    "time"
    "log"
)

func Duration(invocation time.Time, name string) {
    elapsed := time.Since(invocation)

    log.Printf("%s lasted %s", name, elapsed)
}
```

## Usage

```
func BigIntFactorial(x big.Int) *big.Int {
    // Arguments to a defer statement is immediately evaluated and stored.
    // The deferred function receives the pre-evaluated values when its invoked.
    defer profile.Duration(time.Now(), "IntFactorial")

    y := big.NewInt(1)
    for one := big.NewInt(1); x.Sign() > 0; x.Sub(x, one) {
        y.Mul(y, x)
    }

    return x.Set(y)
}
```

# Functional Options

Functional options are a method of implementing clean/eloquent APIs in Go. Options implemented as a function set the state of that option.

# Implementation

## Options

```
package file

type Options struct {
    UID         int
    GID         int
    Flags       int
    Contents    string
    Permissions os.FileMode
}

type Option func(*Options)

func UID(userID int) Option {
    return func(args *Options) {
        args.UID = userID
    }
}

func GID(groupID int) Option {
    return func(args *Options) {
        args.GID = groupID
    }
}

func Contents(c string) Option {
    return func(args *Options) {
        args.Contents = c
    }
}

func Permissions(perms os.FileMode) Option {
    return func(args *Options) {
        args.Permissions = perms
    }
}
```

## Constructor

```
package file

func New(filepath string, setters ...Option) error {
    // Default Options
    args := &Options{
        UID:         os.Getuid(),
        GID:         os.Getgid(),
        Contents:    "",
        Permissions: 0666,
        Flags:       os.O_CREATE | os.O_EXCL | os.O_WRONLY,
    }

    for _, setter := range setters {
        setter(args)
    }

    f, err := os.OpenFile(filepath, args.Flags, args.Permissions)
    if err != nil {
        return err
    } else {
        defer f.Close()
    }

    if _, err := f.WriteString(args.Contents); err != nil {
        return err
    }

    return f.Chown(args.UID, args.GID)
}
```

# Usage

```
emptyFile, err := file.New("/tmp/empty.txt")
if err != nil {
    panic(err)
}

fillerFile, err := file.New("/tmp/file.txt", file.UID(1000), file.Contents("Lorem Ipsu
m Dolor Amet"))
if err != nil {
    panic(err)
}
```

# Contribution Guidelines

Please ensure your pull request adheres to the following guidelines:

- Make an individual pull request for each suggestion.
- Choose the corresponding patterns section for your suggestion.
- List, after your addition, should be in lexicographical order.

# Commit Messages Guidelines

- The message should be in imperative form and uncapitalized.
- If possible, please include an explanation in the commit message body
- Use the form `<pattern-section>/<pattern-name>: <message>` (e.g. `creational/singleton: refactor singleton constructor`)

# Pattern Template

Each pattern should have a single markdown file containing the important part of the implementation, the usage and the explanations for it. This is to ensure that the reader doesn't have to read bunch of boilerplate to understand what's going on and the code is as simple as possible and not simpler.

Please use the following template for adding new patterns:

```
# <Pattern-Name>
<Pattern description>

## Implementation

## Usage

// Optional
## Rules of Thumb
```