guide/style.md at master · uber-go/guide

Introduction

Styles are the conventions that govern our code. The term style is a bit of a misnomer, since these conventions cover far more than just source file formatting—gofmt handles that for us.

The goal of this guide is to manage this complexity by describing in detail the Dos and Don'ts of writing Go code at Uber. These rules exist to keep the code base manageable while still allowing engineers to use Go language features productively.

This guide was originally created by Prashant Varanasi and Simon Newton as a way to bring some colleagues up to speed with using Go. Over the years it has been amended based on feedback from others.

This documents idiomatic conventions in Go code that we follow at Uber. A lot of these are general guidelines for Go, while others extend upon external resources:

Effective Go

Go Common Mistakes

Go Code Review Comments

We aim for the code samples to be accurate for the two most recent minor versions of Go releases.

All code should be error-free when run through `golint` and `go vet`. We recommend setting up your editor to:

Run `goimports` on save

Run `golint` and `go vet` to check for errors

You can find information in editor support for Go tools here: https://github.com/golang/go/wiki/IDEsAndTextEditorPlugins

Guidelines

Pointers to Interfaces

You almost never need a pointer to an interface. You should be passing interfaces as values—the underlying data can still be a pointer.

An interface is two fields:

A pointer to some type-specific information. You can think of this as "type."

Data pointer. If the data stored is a pointer, it’s stored directly. If the data stored is a value, then a pointer to the value is stored.

If you want interface methods to modify the underlying data, you must use a pointer.

Verify Interface Compliance

Verify interface compliance at compile time where appropriate. This includes:

Exported types that are required to implement specific interfaces as part of their API contract

Exported or unexported types that are part of a collection of types implementing the same interface

Other cases where violating an interface would break users

Bad	Good

Bad	Good

type Handler struct {
  // ...
}
 	
func (h *Handler) ServeHTTP(
  w http.ResponseWriter,
  r *http.Request,
) {
  ...
}
	type Handler struct {
	  // ...
	}
	
	var _ http.Handler = (*Handler)(nil)
	
	func (h *Handler) ServeHTTP(
	  w http.ResponseWriter,
	  r *http.Request,
	) {
	  // ...
	}

The statement `var _ http.Handler = (*Handler)(nil)` will fail to compile if `*Handler` ever stops matching the `http.Handler` interface.

The right hand side of the assignment should be the zero value of the asserted type. This is `nil` for pointer types (like `*Handler`), slices, and maps, and an empty struct for struct types.

```
type LogHandler struct {
  h   http.Handler
  log *zap.Logger
}

var _ http.Handler = LogHandler{}

func (h LogHandler) ServeHTTP(
  w http.ResponseWriter,
  r *http.Request,
) {
  // ...
}
```

[Receivers and Interfaces]

Methods with value receivers can be called on pointers as well as values. Methods with pointer receivers can only be called on pointers or [addressable values].

For example,

```
type S struct {
  data string
```

```go
}

func (s S) Read() string {
  return s.data
}

func (s *S) Write(str string) {
  s.data = str
}

// We cannot get pointers to values stored in maps, because they are not
// addressable values.
sVals := map[int]S{1: {"A"}}

// We can call Read on values stored in the map because Read
// has a value receiver, which does not require the value to
// be addressable.
sVals[1].Read()

// We cannot call Write on values stored in the map because Write
// has a pointer receiver, and it's not possible to get a pointer
// to a value stored in a map.
//
//  sVals[1].Write("test")

sPtrs := map[int]*S{1: {"A"}}

// You can call both Read and Write if the map stores pointers,
// because pointers are intrinsically addressable.
sPtrs[1].Read()
sPtrs[1].Write("test")
```

Similarly, an interface can be satisfied by a pointer, even if the method has a value receiver.

```go
type F interface {
  f()
}
```

```
type S1 struct{}

func (s S1) f() {}

type S2 struct{}

func (s *S2) f() {}

s1Val := S1{}
s1Ptr := &S1{}
s2Val := S2{}
s2Ptr := &S2{}

var i F
i = s1Val
i = s1Ptr
i = s2Ptr

// The following doesn't compile, since s2Val is a value, and there is no value receiver for f.
//   i = s2Val
```

Effective Go has a good write up on [Pointers vs. Values](#).

Zero-value Mutexes are Valid

The zero-value of `sync.Mutex` and `sync.RWMutex` is valid, so you almost never need a pointer to a mutex.

Bad	Good
mu := new(sync.Mutex) mu.Lock()	var mu sync.Mutex mu.Lock()

If you use a struct by pointer, then the mutex should be a non-pointer field on it. Do not embed the mutex on the struct, even if the struct is not exported.

Bad	Good

type SMap struct {
  sync.Mutex
 	
  data map[string]string
}
	type SMap struct {
	  mu sync.Mutex
	
	  data map[string]string
	}

Bad	Good

func NewSMap() *SMap {
  return &SMap{
    data: make(map[string]string),
  }
}
 	
func (m *SMap) Get(k string) string {
  m.Lock()
  defer m.Unlock()
 	
  return m.data[k]
}
	func NewSMap() *SMap {
	  return &SMap{
	    data: make(map[string]string),
	  }
	}
	
	func (m *SMap) Get(k string) string {
	  m.mu.Lock()
	  defer m.mu.Unlock()
	
	  return m.data[k]
	}

The `Mutex` field, and the `Lock` and `Unlock` methods are unintentionally part of the exported API of `SMap`.	The mutex and its methods are implementation details of `SMap` hidden from its callers.

Copy Slices and Maps at Boundaries

Slices and maps contain pointers to the underlying data so be wary of scenarios when they need to be copied.

Receiving Slices and Maps

Keep in mind that users can modify a map or slice you received as an argument if you store a reference to it.

Bad	Good

func (d *Driver) SetTrips(trips []Trip) {
  d.trips = trips
}
 	
trips := ...
d1.SetTrips(trips)
 	
// Did you mean to modify d1.trips?
trips[0] = ...
	func (d *Driver) SetTrips(trips []Trip) {
	  d.trips = make([]Trip, len(trips))
	  copy(d.trips, trips)
	}
	
	trips := ...
	d1.SetTrips(trips)
	
	// We can now modify trips[0] without affecting d1.trips.
	trips[0] = ...

Returning Slices and Maps

Similarly, be wary of user modifications to maps or slices exposing internal state.

Bad	Good

type Stats struct {
  mu sync.Mutex
  counters map[string]int
	type Stats struct {
	  mu sync.Mutex
	  counters map[string]int

Bad	Good

}
 	
// Snapshot returns the current stats.
func (s *Stats) Snapshot() map[string]int {
  s.mu.Lock()
  defer s.mu.Unlock()
 	
  return s.counters
}
 	
// snapshot is no longer protected by the mutex, so any
// access to the snapshot is subject to data races.
snapshot := stats.Snapshot()
	}
	
	func (s *Stats) Snapshot() map[string]int {
	  s.mu.Lock()
	  defer s.mu.Unlock()
	
	  result := make(map[string]int, len(s.counters))
	  for k, v := range s.counters {
	    result[k] = v
	  }
	  return result
	}
	
	// Snapshot is now a copy.
	snapshot := stats.Snapshot()

Defer to Clean Up

Use defer to clean up resources such as files and locks.

Bad	Good

p.Lock()
if p.count < 10 {
  p.Unlock()
  return p.count
}
 	
p.count++
newCount := p.count
p.Unlock()
 	
return newCount
 	
// easy to miss unlocks due to multiple returns
	p.Lock()
	defer p.Unlock()
	
	if p.count < 10 {
	  return p.count
	}
	
	p.count++
	return p.count
	
	// more readable

Defer has an extremely small overhead and should be avoided only if you can prove that your function execution time is in the order of nanoseconds. The readability win of using defers is worth the miniscule cost of using them. This is especially true for larger methods that have more than simple memory accesses, where the other computations are more significant than the `defer`.

Channel Size is One or None

Channels should usually have a size of one or be unbuffered. By default, channels are unbuffered and have a size of zero. Any other size must be subject to a high level of scrutiny. Consider how the size is determined, what prevents the channel from filling up under load and blocking writers, and what happens when this occurs.

Bad	Good

// Ought to be enough for anybody!
c := make(chan int, 64)
	// Size of one
	c := make(chan int, 1) // or
	// Unbuffered channel, size of zero
	c := make(chan int)

[Start Enums at One]

The standard way of introducing enumerations in Go is to declare a custom type and a `const` group with `iota`. Since variables have a 0 default value, you should usually start your enums on a non-zero value.

Bad	Good

type Operation int
 	
const (
  Add Operation = iota
  Subtract
  Multiply
)
 	
// Add=0, Subtract=1, Multiply=2
	type Operation int
	
	const (
	  Add Operation = iota + 1
	  Subtract
	  Multiply
	)
	
	// Add=1, Subtract=2, Multiply=3

There are cases where using the zero value makes sense, for example when the zero value case is the desirable default behavior.

```
type LogOutput int

const (
  LogToStdout LogOutput = iota
  LogToFile
  LogToRemote
)

// LogToStdout=0, LogToFile=1, LogToRemote=2
```

[Use `"time"` to handle time]

Time is complicated. Incorrect assumptions often made about time include the following.

A day has 24 hours

An hour has 60 minutes

A week has 7 days

A year has 365 days

[And a lot more](#)

For example, *1* means that adding 24 hours to a time instant will not always yield a new calendar day.

Therefore, always use the `"time"` package when dealing with time because it helps deal with these incorrect assumptions in a safer, more accurate manner.

Use `time.Time` for instants of time

Use `time.Time` when dealing with instants of time, and the methods on `time.Time` when comparing, adding, or subtracting time.

Bad	Good

func isActive(now, start, stop int) bool {
  return start <= now && now < stop
}
	func isActive(now, start, stop time.Time) bool {
	  return (start.Before(now) || start.Equal(now)) && now.Before(stop)
	}

Use `time.Duration` for periods of time

Use `time.Duration` when dealing with periods of time.

Bad	Good

func poll(delay int) {
  for {
    // ...
    time.Sleep(time.Duration(delay) * time.Millisecond)
  }
}
 	
poll(10) // was it seconds or milliseconds?
	func poll(delay time.Duration) {
	  for {
	    // ...
	    time.Sleep(delay)
	  }
	}
	
	poll(10*time.Second)

Going back to the example of adding 24 hours to a time instant, the method we use to add time depends on intent. If we want the same time of the day, but on the next calendar day, we should use `Time.AddDate`. However, if we want an instant of time guaranteed to be 24 hours after the previous time, we should use `Time.Add`.

```
newDay := t.AddDate(0 /* years */, 0 /* months */, 1 /* days */)
```

maybeNewDay := t.Add(24 * time.Hour)

Use `time.Time` and `time.Duration` with external systems

Use `time.Duration` and `time.Time` in interactions with external systems when possible. For example:

Command-line flags: `flag` supports `time.Duration` via `time.ParseDuration`

JSON: `encoding/json` supports encoding `time.Time` as an RFC 3339 string via its `UnmarshalJSON` method

SQL: `database/sql` supports converting `DATETIME` or `TIMESTAMP` columns into `time.Time` and back if the underlying driver supports it

YAML: `gopkg.in/yaml.v2` supports `time.Time` as an RFC 3339 string, and `time.Duration` via `time.ParseDuration`.

When it is not possible to use `time.Duration` in these interactions, use `int` or `float64` and include the unit in the name of the field.

For example, since `encoding/json` does not support `time.Duration`, the unit is included in the name of the field.

Bad	Good

// {"interval": 2}
type Config struct {
  Interval int \`json:"interval"\`
}
	// {"intervalMillis": 2000}
	type Config struct {
	  IntervalMillis int \`json:"intervalMillis"\`
	}

When it is not possible to use `time.Time` in these interactions, unless an alternative is agreed upon, use `string` and format timestamps as defined in RFC 3339. This format is used by default by `Time.UnmarshalText` and is available for use in `Time.Format` and `time.Parse` via `time.RFC3339`.

Although this tends to not be a problem in practice, keep in mind that the `"time"` package does not support parsing timestamps with leap seconds (8728), nor does it account for leap seconds in calculations (15190). If you compare two instants of time, the difference will not include the leap seconds that may have occurred between those two instants.

Errors

Error Types

There are few options for declaring errors. Consider the following before picking the option best suited for your use case.

Does the caller need to match the error so that they can handle it? If yes, we must support the `errors.Is` or `errors.As` functions by declaring a top-level error variable or a custom type.

Is the error message a static string, or is it a dynamic string that requires contextual information? For the former, we can use `errors.New`, but for the latter we must use `fmt.Errorf` or a custom error type.

Are we propagating a new error returned by a downstream function? If so, see the section on error wrapping.

Error matching?	Error Message	Guidance
No	static	`errors.New`
No	dynamic	`fmt.Errorf`
Yes	static	top-level `var` with `errors.New`
Yes	dynamic	custom `error` type

For example, use `errors.New` for an error with a static string. Export this error as a variable to support matching it with `errors.Is` if the caller needs to match and handle this error.

<table>
<tr><th>No error matching</th><th>Error matching</th></tr>
<tr><td>

```
// package foo

func Open() error {
  return errors.New("could not open")
}

// package bar

if err := foo.Open(); err != nil {
  // Can't handle the error.
  panic("unknown error")
}
```

</td><td>

```
// package foo

var ErrCouldNotOpen = errors.New("could not open")

func Open() error {
  return ErrCouldNotOpen
}

// package bar

if err := foo.Open(); err != nil {
  if errors.Is(err, foo.ErrCouldNotOpen) {
    // handle the error
  } else {
    panic("unknown error")
  }
}
```

</td></tr>
</table>

For an error with a dynamic string, use `fmt.Errorf` if the caller does not need to match it, and a custom `error` if the caller does need to match it.

<table>
<tr><th>No error matching</th><th>Error matching</th></tr>
<tr><td>

```
// package foo
```

</td><td>

```
// package foo
```

</td></tr>
</table>

No error matching	Error matching

func Open(file string) error {
  return fmt.Errorf("file %q not found", file)
}
 	
// package bar
 	
if err := foo.Open("testfile.txt"); err != nil {
  // Can't handle the error.
  panic("unknown error")
}
	type NotFoundError struct {
	  File string
	}
	
	func (e *NotFoundError) Error() string {
	  return fmt.Sprintf("file %q not found", e.File)
	}
	
	func Open(file string) error {
	  return &NotFoundError{File: file}
	}
	
	// package bar
	
	if err := foo.Open("testfile.txt"); err != nil {
	  var notFound *NotFoundError
	  if errors.As(err, &notFound) {
	    // handle the error
	  } else {
	    panic("unknown error")
	  }
	}

Note that if you export error variables or types from a package, they will become part of the public API of the package.

Error Wrapping

There are three main options for propagating errors if a call fails:

return the original error as-is

add context with `fmt.Errorf` and the `%w` verb

add context with `fmt.Errorf` and the `%v` verb

Return the original error as-is if there is no additional context to add. This maintains the original error type and message. This is well suited for cases when the underlying error message has sufficient information to track down where it came from.

Otherwise, add context to the error message where possible so that instead of a vague error such as "connection refused", you get more useful errors such as "call service foo: connection refused".

Use `fmt.Errorf` to add context to your errors, picking between the `%w` or `%v` verbs based on whether the caller should be able to match and extract the underlying cause.

Use `%w` if the caller should have access to the underlying error. This is a good default for most wrapped errors, but be aware that callers may begin to rely on this behavior. So for cases where the wrapped error is a known `var` or type, document and test it as part of your function's contract.

Use `%v` to obfuscate the underlying error. Callers will be unable to match it, but you can switch to `%w` in the future if needed.

When adding context to returned errors, keep the context succinct by avoiding phrases like "failed to", which state the obvious and pile up as the error percolates up through the stack:

Bad	Good

s, err := store.New()
if err != nil {
    return fmt.Errorf(
        "failed to create new store: %w", err)
}
	s, err := store.New()
	if err != nil {
	    return fmt.Errorf(
	        "new store: %w", err)
	}

`failed to x: failed to y: failed to create new store: the error`	`x: y: new store: the error`

However once the error is sent to another system, it should be clear the message is an error (e.g. an `err` tag or "Failed" prefix in logs).

See also [Don't just check errors, handle them gracefully].

[Error Naming]

For error values stored as global variables, use the prefix `Err` or `err` depending on whether they're exported. This guidance supersedes the [Prefix Unexported Globals with _].

```
var (
  // The following two errors are exported
  // so that users of this package can match them
  // with errors.Is.

  ErrBrokenLink = errors.New("link is broken")
  ErrCouldNotOpen = errors.New("could not open")

  // This error is not exported because
  // we don't want to make it part of our public API.
  // We may still use it inside the package
```

```
  // with errors.Is.

  errNotFound = errors.New("not found")
)
```

For custom error types, use the suffix `Error` instead.

```
// Similarly, this error is exported
// so that users of this package can match it
// with errors.As.

type NotFoundError struct {
  File string
}

func (e *NotFoundError) Error() string {
  return fmt.Sprintf("file %q not found", e.File)
}

// And this error is not exported because
// we don't want to make it part of the public API.
// We can still use it inside the package
// with errors.As.

type resolveError struct {
  Path string
}

func (e *resolveError) Error() string {
  return fmt.Sprintf("resolve %q", e.Path)
}
```

[Handle Errors Once]

When a caller receives an error from a callee, it can handle it in a variety of different ways depending on what it knows about the error.

These include, but not are limited to:

if the callee contract defines specific errors, matching the error with `errors.Is` or `errors.As` and handling the branches differently

if the error is recoverable, logging the error and degrading gracefully

if the error represents a domain-specific failure condition, returning a well-defined error

returning the error, either wrapped or verbatim

Regardless of how the caller handles the error, it should typically handle each error only once. The caller should not, for example, log the error and then return it, because *its* callers may handle the error as well.

For example, consider the following cases:

Description	Code

Bad: Log the error and return it
 	
Callers further up the stack will likely take a similar action with the error. Doing so causing a lot of noise in the application logs for little value.
	u, err := getUser(id)
	if err != nil {
	  // BAD: See description
	  log.Printf("Could not get user %q: %v", id, err)
	  return err
	}

Good: Wrap the error and return it
 	
Callers further up the stack will handle the error. Use of `%w` ensures they can match the error with `errors.Is` or `errors.As` if relevant.
	u, err := getUser(id)
	if err != nil {
	  return fmt.Errorf("get user %q: %w", id, err)
	}

Good: Log the error and degrade gracefully
 	
If the operation isn't strictly necessary, we can provide a degraded but unbroken experience by recovering from it.
	if err := emitMetrics(); err != nil {
	  // Failure to write metrics should not
	  // break the application.
	  log.Printf("Could not emit metrics: %v", err)
	}

Description	Code

Good: Match the error and degrade gracefully
 	
If the callee defines a specific error in its contract, and the failure is recoverable, match on that error case and degrade gracefully. For all other cases, wrap the error and return it.
 	
Callers further up the stack will handle other errors.
	tz, err := getUserTimeZone(id)
	if err != nil {
	  if errors.Is(err, ErrUserNotFound) {
	    // User doesn't exist. Use UTC.
	    tz = time.UTC
	  } else {
	    return fmt.Errorf("get user %q: %w", id, err)
	  }
	}

Handle Type Assertion Failures

The single return value form of a type assertion will panic on an incorrect type. Therefore, always use the "comma ok" idiom.

Bad	Good

t := i.(string)	t, ok := i.(string)
	if !ok {
	  // handle the error gracefully
	}

Don't Panic

Code running in production must avoid panics. Panics are a major source of cascading failures. If an error occurs, the function must return an error and allow the caller to decide how to handle it.

Bad	Good

Bad	Good

func run(args []string) {
  if len(args) == 0 {
    panic("an argument is required")
  }
  // ...
}
 	
func main() {
  run(os.Args[1:])
}
	func run(args []string) error {
	  if len(args) == 0 {
	    return errors.New("an argument is required")
	  }
	  // ...
	  return nil
	}
	
	func main() {
	  if err := run(os.Args[1:]); err != nil {
	    fmt.Fprintln(os.Stderr, err)
	    os.Exit(1)
	  }
	}

Panic/recover is not an error handling strategy. A program must panic only when something irrecoverable happens such as a nil dereference. An exception to this is program initialization: bad things at program startup that should abort the program may cause panic.

var _statusTemplate = template.Must(template.New("name").Parse("_statusHTML"))

Even in tests, prefer `t.Fatal` or `t.FailNow` over panics to ensure that the test is marked as failed.

Bad	Good

// func TestFoo(t *testing.T)
 	
f, err := os.CreateTemp("", "test")
if err != nil {
  panic("failed to set up test")
}
	// func TestFoo(t *testing.T)
	
	f, err := os.CreateTemp("", "test")
	if err != nil {
	  t.Fatal("failed to set up test")
	}

[Use go.uber.org/atomic]()

Atomic operations with the [sync/atomic]() package operate on the raw types (`int32`, `int64`, etc.) so it is easy to forget to use the atomic operation to read or modify the variables.

[go.uber.org/atomic]() adds type safety to these operations by hiding the underlying type. Additionally, it includes a convenient `atomic.Bool` type.

Bad	Good

type foo struct {
  running int32 // atomic
}
 	
func (f* foo) start() {
	type foo struct {
	  running atomic.Bool
	}
	
	func (f *foo) start() {

Bad	Good

if atomic.SwapInt32(&f.running, 1) == 1 {
    // already running…
    return
  }
  // start the Foo
}
 	
func (f *foo) isRunning() bool {
  return f.running == 1 // race!
}
	if f.running.Swap(true) {
	    // already running…
	    return
	  }
	  // start the Foo
	}
	
	func (f *foo) isRunning() bool {
	  return f.running.Load()
	}

[Avoid Mutable Globals]

Avoid mutating global variables, instead opting for dependency injection. This applies to function pointers as well as other kinds of values.

Bad	Good

// sign.go
 	
var _timeNow = time.Now
 	
func sign(msg string) string {
  now := _timeNow()
  return signWithTime(msg, now)
}
	// sign.go
	
	type signer struct {
	  now func() time.Time
	}
	
	func newSigner() *signer {
	  return &signer{
	    now: time.Now,
	  }
	}
	
	func (s *signer) Sign(msg string) string {
	  now := s.now()
	  return signWithTime(msg, now)
	}

// sign_test.go
 	
func TestSign(t *testing.T) {
  oldTimeNow := _timeNow
  _timeNow = func() time.Time {
    return someFixedTime
  }
  defer func() { _timeNow = oldTimeNow }()
 	
  assert.Equal(t, want, sign(give))
}
	// sign_test.go
	
	func TestSigner(t *testing.T) {
	  s := newSigner()
	  s.now = func() time.Time {
	    return someFixedTime
	  }
	
	  assert.Equal(t, want, s.Sign(give))
	}

Avoid Embedding Types in Public Structs

These embedded types leak implementation details, inhibit type evolution, and obscure documentation.

Assuming you have implemented a variety of list types using a shared `AbstractList`, avoid embedding the `AbstractList` in your concrete list implementations. Instead, hand-write only the methods to your concrete list that will delegate to the abstract list.

```
type AbstractList struct {}

// Add adds an entity to the list.
func (l *AbstractList) Add(e Entity) {
  // ...
}

// Remove removes an entity from the list.
func (l *AbstractList) Remove(e Entity) {
  // ...
}
```

Bad	Good

// ConcreteList is a list of entities.
type ConcreteList struct {
  *AbstractList
}
	// ConcreteList is a list of entities.
	type ConcreteList struct {
	  list *AbstractList
	}
	
	// Add adds an entity to the list.
	func (l *ConcreteList) Add(e Entity) {
	  l.list.Add(e)
	}
	
	// Remove removes an entity from the list.
	func (l *ConcreteList) Remove(e Entity) {
	  l.list.Remove(e)
	}

Go allows type embedding as a compromise between inheritance and composition. The outer type gets implicit copies of the embedded type's methods. These methods, by default, delegate to the same method of the embedded instance.

The struct also gains a field by the same name as the type. So, if the embedded type is public, the field is public. To maintain backward compatibility, every future version of the outer type must keep the embedded type.

An embedded type is rarely necessary. It is a convenience that helps you avoid writing tedious delegate methods.

Even embedding a compatible AbstractList *interface*, instead of the struct, would offer the developer more flexibility to change in the future, but still leak the detail that the concrete lists use an abstract implementation.

<table>
<tr><th>Bad</th><th>Good</th></tr>
<tr><td>

```
// AbstractList is a generalized implementation
// for various kinds of lists of entities.
type AbstractList interface {
  Add(Entity)
  Remove(Entity)
}

// ConcreteList is a list of entities.
type ConcreteList struct {
  AbstractList
}
```

</td><td>

```
// ConcreteList is a list of entities.
type ConcreteList struct {
  list AbstractList
}

// Add adds an entity to the list.
func (l *ConcreteList) Add(e Entity) {
  l.list.Add(e)
}

// Remove removes an entity from the list.
func (l *ConcreteList) Remove(e Entity) {
  l.list.Remove(e)
}
```

</td></tr>
</table>

Either with an embedded struct or an embedded interface, the embedded type places limits on the evolution of the type.

Adding methods to an embedded interface is a breaking change.

Removing methods from an embedded struct is a breaking change.

Removing the embedded type is a breaking change.

Replacing the embedded type, even with an alternative that satisfies the same interface, is a breaking change.

Although writing these delegate methods is tedious, the additional effort hides an implementation detail, leaves more opportunities for change, and also eliminates indirection for discovering the full List interface in documentation.

Avoid Using Built-In Names

The Go language specification outlines several built-in, predeclared identifiers that should not be used as names within Go programs.

Depending on context, reusing these identifiers as names will either shadow the original within the current lexical scope (and any nested scopes) or make affected code confusing. In the best case, the compiler will complain; in the worst case, such code may introduce latent, hard-to-grep bugs.

Bad	Good

Bad	Good

var error string
// \`error\` shadows the builtin
 	
// or
 	
func handleErrorMessage(error string) {
    // \`error\` shadows the builtin
}
	var errorMessage string
	// \`error\` refers to the builtin
	
	// or
	
	func handleErrorMessage(msg string) {
	    // \`error\` refers to the builtin
	}

type Foo struct {
    // While these fields technically don't
    // constitute shadowing, grepping for
    // \`error\` or \`string\` strings is now
    // ambiguous.
    error error
    string string
}
 	
func (f Foo) Error() error {
    // \`error\` and \`f.error\` are
    // visually similar
    return f.error
}
 	
func (f Foo) String() string {
    // \`string\` and \`f.string\` are
    // visually similar
    return f.string
}
	type Foo struct {
	    // \`error\` and \`string\` strings are
	    // now unambiguous.
	    err error
	    str string
	}
	
	func (f Foo) Error() error {
	    return f.err
	}
	
	func (f Foo) String() string {
	    return f.str
	}

Note that the compiler will not generate errors when using predeclared identifiers, but tools such as `go vet` should correctly point out these and other cases of shadowing.

Avoid `init()`

Avoid `init()` where possible. When `init()` is unavoidable or desirable, code should attempt to:

Be completely deterministic, regardless of program environment or invocation.

Avoid depending on the ordering or side-effects of other `init()` functions. While `init()` ordering is well-known, code can change, and thus relationships between `init()` functions can make code brittle and error-prone.

Avoid accessing or manipulating global or environment state, such as machine information, environment variables, working directory, program arguments/inputs, etc.

Avoid I/O, including both filesystem, network, and system calls.

Code that cannot satisfy these requirements likely belongs as a helper to be called as part of `main()` (or elsewhere in a program's lifecycle), or be written as part of `main()` itself. In particular, libraries that are intended to be used by other programs should take special care to be completely deterministic and not perform "init magic".

Bad	Good

type Foo struct {
    // ...
}
 	
var _defaultFoo Foo
 	
func init() {
    _defaultFoo = Foo{
        // ...
    }
}
	var _defaultFoo = Foo{
	    // ...
	}
	
	// or, better, for testability:
	
	var _defaultFoo = defaultFoo()
	
	func defaultFoo() Foo {
	    return Foo{
	        // ...
	    }
	}

type Config struct {
    // ...
}
 	
var _config Config
 	
func init() {
    // Bad: based on current directory
    cwd, _ := os.Getwd()
 	
    // Bad: I/O
    raw, _ := os.ReadFile(
        path.Join(cwd, "config", "config.yaml"),
)
 	
    yaml.Unmarshal(raw, &_config)
}
	type Config struct {
	    // ...
	}
	
	func loadConfig() Config {
	    cwd, err := os.Getwd()
	    // handle err
	
	    raw, err := os.ReadFile(
	        path.Join(cwd, "config", "config.yaml"),
	    )
	    // handle err
	
	    var config Config
	    yaml.Unmarshal(raw, &config)
	
	    return config
	}

Considering the above, some situations in which `init()` may be preferable or necessary might include:

Complex expressions that cannot be represented as single assignments.

Pluggable hooks, such as `database/sql` dialects, encoding type registries, etc.

Optimizations to [Google Cloud Functions] and other forms of deterministic precomputation.

[Exit in Main]

Go programs use `os.Exit` or `log.Fatal*` to exit immediately. (Panicking is not a good way to exit programs, please don't panic.)

Call one of `os.Exit` or `log.Fatal*` **only in `main()`**. All other functions should return errors to signal failure.

Bad	Good

func main() {
  body := readFile(path)
  fmt.Println(body)
}
 	
func readFile(path string) string {
  f, err := os.Open(path)
  if err != nil {
    log.Fatal(err)
  }
 	
  b, err := io.ReadAll(f)
  if err != nil {
    log.Fatal(err)
  }
 	
  return string(b)
}
	func main() {
	  body, err := readFile(path)
	  if err != nil {
	    log.Fatal(err)
	  }
	  fmt.Println(body)
	}
	
	func readFile(path string) (string, error) {
	  f, err := os.Open(path)
	  if err != nil {
	    return "", err
	  }
	
	  b, err := io.ReadAll(f)
	  if err != nil {
	    return "", err
	  }
	
	  return string(b), nil
	}

Rationale: Programs with multiple functions that exit present a few issues:

Non-obvious control flow: Any function can exit the program so it becomes difficult to reason about the control flow.

Difficult to test: A function that exits the program will also exit the test calling it. This makes the function difficult to test and introduces risk of skipping other tests that have not yet been run by `go test`.

Skipped cleanup: When a function exits the program, it skips function calls enqueued with `defer` statements. This adds risk of skipping important cleanup tasks.

Exit Once

If possible, prefer to call `os.Exit` or `log.Fatal` **at most once** in your `main()`. If there are multiple error scenarios that halt program execution, put that logic under a separate function and return errors from it.

This has the effect of shortening your `main()` function and putting all key business logic into a separate, testable function.

Bad	Good

package main
 	
func main() {
  args := os.Args[1:]
  if len(args) != 1 {
    log.Fatal("missing file")
  }
  name := args[0]
 	
  f, err := os.Open(name)
  if err != nil {
    log.Fatal(err)
  }
  defer f.Close()
 	
  // If we call log.Fatal after this line,
  // f.Close will not be called.
 	
  b, err := io.ReadAll(f)
  if err != nil {
    log.Fatal(err)
  }
 	
  // ...
}
	package main
	
	func main() {
	  if err := run(); err != nil {
	    log.Fatal(err)
	  }
	}
	
	func run() error {
	  args := os.Args[1:]
	  if len(args) != 1 {
	    return errors.New("missing file")
	  }
	  name := args[0]
	
	  f, err := os.Open(name)
	  if err != nil {
	    return err
	  }
	  defer f.Close()
	
	  b, err := io.ReadAll(f)
	  if err != nil {
	    return err
	  }
	
	  // ...
	}

The example above uses `log.Fatal`, but the guidance also applies to `os.Exit` or any library code that calls `os.Exit`.

```
func main() {
  if err := run(); err != nil {
    fmt.Fprintln(os.Stderr, err)
    os.Exit(1)
  }
}
```

You may alter the signature of `run()` to fit your needs. For example, if your program must exit with specific exit codes for failures, `run()` may return the exit code instead of an error. This allows unit tests to verify this behavior directly as well.

```
func main() {
```

```
  os.Exit(run(args))
}

func run() (exitCode int) {
  // ...
}
```

More generally, note that the `run()` function used in these examples is not intended to be prescriptive. There's flexibility in the name, signature, and setup of the `run()` function. Among other things, you may:

accept unparsed command line arguments (e.g., `run(os.Args[1:])`)

parse command line arguments in `main()` and pass them onto `run`

use a custom error type to carry the exit code back to `main()`

put business logic in a different layer of abstraction from `package main`

This guidance only requires that there's a single place in your `main()` responsible for actually exiting the process.

Use field tags in marshaled structs

Any struct field that is marshaled into JSON, YAML, or other formats that support tag-based field naming should be annotated with the relevant tag.

Bad	Good

<pre>type Stock struct {
  Price int
  Name string
}
 	
bytes, err := json.Marshal(Stock{
  Price: 137,
  Name: "UBER",
})</pre>
	<pre>type Stock struct {
	  Price int \`json:"price"\`
	  Name string \`json:"name"\`
	  // Safe to rename Name to Symbol.
	}
	
	bytes, err := json.Marshal(Stock{
	  Price: 137,
	  Name: "UBER",
	})</pre>

Rationale: The serialized form of the structure is a contract between different systems. Changes to the structure of the serialized form--including field names--break this contract. Specifying field names inside tags makes the contract explicit, and it guards against accidentally breaking the contract by refactoring or renaming fields.

Don't fire-and-forget goroutines

Goroutines are lightweight, but they're not free: at minimum, they cost memory for their stack and CPU to be scheduled. While these costs are small for typical uses of goroutines, they can cause significant performance issues when spawned in large numbers without controlled lifetimes. Goroutines with unmanaged lifetimes can also cause other issues like preventing unused objects from being garbage collected and holding onto resources that are otherwise no longer used.

Therefore, do not leak goroutines in production code. Use go.uber.org/goleak to test for goroutine leaks inside packages that may spawn goroutines.

In general, every goroutine:

must have a predictable time at which it will stop running; or

there must be a way to signal to the goroutine that it should stop

In both cases, there must be a way code to block and wait for the goroutine to finish.

For example:

<table>
<tr><th>Bad</th><th>Good</th></tr>
<tr><td><pre>go func() {
  for {
    flush()
    time.Sleep(delay)
  }
}()</pre></td><td><pre>var (
  stop = make(chan struct{}) // tells the goroutine to stop
  done = make(chan struct{}) // tells us that the goroutine exited
)
go func() {
  defer close(done)

  ticker := time.NewTicker(delay)
  defer ticker.Stop()
  for {
    select {
    case <-ticker.C:
      flush()
    case <-stop:
      return
    }
  }
}()

// Elsewhere...
close(stop) // signal the goroutine to stop
<-done   // and wait for it to exit</pre></td></tr>
<tr><td>There's no way to stop this goroutine. This will run until the application exits.</td><td>This goroutine can be stopped with <code>close(stop)</code>, and we can wait for it to exit with <code><-done</code>.</td></tr>
</table>

[Wait for goroutines to exit]()

Given a goroutine spawned by the system, there must be a way to wait for the goroutine to exit. There are two popular ways to do this:

Use a `sync.WaitGroup`. Do this if there are multiple goroutines that you want to wait for

```
var wg sync.WaitGroup
for i := 0; i < N; i++ {
  wg.Add(1)
  go func() {
    defer wg.Done()
    // ...
  }()
}

// To wait for all to finish:
wg.Wait()
```

Add another `chan struct{}` that the goroutine closes when it's done. Do this if there's only one goroutine.

```
done := make(chan struct{})
go func() {
  defer close(done)
  // ...
}()

// To wait for the goroutine to finish:
<-done
```

No goroutines in `init()`

`init()` functions should not spawn goroutines. See also Avoid init().

If a package has need of a background goroutine, it must expose an object that is responsible for managing a goroutine's lifetime. The object must provide a method (`Close`, `Stop`, `Shutdown`, etc) that signals the background goroutine to stop, and waits for it to exit.

Bad	Good

func init() {
  go doWork()
}
 	
func doWork() {
	type Worker struct{ /* ... */ }
	
	func NewWorker(...) *Worker {
	  w := &Worker{
	    stop: make(chan struct{}),

<table>
<tr><th>Bad</th><th>Good</th></tr>
<tr><td><pre>
  for {
    // ...
  }
}
</pre></td><td><pre>
    done: make(chan struct{}),
    // ...
  }
  go w.doWork()
}

func (w *Worker) doWork() {
  defer close(w.done)
  for {
    // ...
    case <-w.stop:
      return
  }
}

// Shutdown tells the worker to stop
// and waits until it has finished.
func (w *Worker) Shutdown() {
  close(w.stop)
  <-w.done
}
</pre></td></tr>
<tr><td>Spawns a background goroutine unconditionally when the user exports this package. The user has no control over the goroutine or a means of stopping it.</td><td>Spawns the worker only if the user requests it. Provides a means of shutting down the worker so that the user can free up resources used by the worker.

Note that you should use <code>WaitGroups</code> if the worker manages multiple goroutines. See Wait for goroutines to exit.</td></tr>
</table>

Performance

Performance-specific guidelines apply only to the hot path.

Prefer strconv over fmt

When converting primitives to/from strings, `strconv` is faster than `fmt`.

<table>
<tr><th>Bad</th><th>Good</th></tr>
<tr><td><pre>
for i := 0; i < b.N; i++ {
  s := fmt.Sprint(rand.Int())
}
</pre></td><td><pre>
for i := 0; i < b.N; i++ {
  s := strconv.Itoa(rand.Int())
}
</pre></td></tr>
<tr><td><pre>
BenchmarkFmtSprint-4   143 ns/op   2 allocs/op
</pre></td><td><pre>
BenchmarkStrconv-4   64.2 ns/op   1 allocs/op
</pre></td></tr>
</table>

Avoid string-to-byte conversion

Do not create byte slices from a fixed string repeatedly. Instead, perform the conversion once and capture the result.

Bad	Good

`for i := 0; i < b.N; i++ {`
`  w.Write([]byte("Hello world"))`
`}`
	`data := []byte("Hello world")`
	`for i := 0; i < b.N; i++ {`
	`  w.Write(data)`
	`}`

`BenchmarkBad-4    50000000    22.2 ns/op`	`BenchmarkGood-4   500000000   3.25 ns/op`

Prefer Specifying Container Capacity

Specify container capacity where possible in order to allocate memory for the container up front. This minimizes subsequent allocations (by copying and resizing of the container) as elements are added.

Specifying Map Capacity Hints

Where possible, provide capacity hints when initializing maps with `make()`.

Providing a capacity hint to `make()` tries to right-size the map at initialization time, which reduces the need for growing the map and allocations as elements are added to the map.

Note that, unlike slices, map capacity hints do not guarantee complete, preemptive allocation, but are used to approximate the number of hashmap buckets required. Consequently, allocations may still occur when adding elements to the map, even up to the specified capacity.

Bad	Good

`m := make(map[string]os.FileInfo)`
 	
`files, _ := os.ReadDir("./files")`
`for _, f := range files {`
`    m[f.Name()] = f`
`}`
	`files, _ := os.ReadDir("./files")`
	
	`m := make(map[string]os.DirEntry, len(files))`
	`for _, f := range files {`
	`    m[f.Name()] = f`
	`}`

`m` is created without a size hint; there may be more allocations at assignment time.	`m` is created with a size hint; there may be fewer allocations at assignment time.

Specifying Slice Capacity

Where possible, provide capacity hints when initializing slices with `make()`, particularly when appending.

make([]T, length, capacity)

Unlike maps, slice capacity is not a hint: the compiler will allocate enough memory for the capacity of the slice as provided to `make()`, which means that subsequent `append()` operations will incur zero allocations (until the length of the slice matches the capacity, after which any appends will require a resize to hold additional elements).

Bad	Good

for n := 0; n < b.N; n++ {
  data := make([]int, 0)
  for k := 0; k < size; k++{
    data = append(data, k)
  }
}
	for n := 0; n < b.N; n++ {
	  data := make([]int, 0, size)
	  for k := 0; k < size; k++{
	    data = append(data, k)
	  }
	}

`BenchmarkBad-4    100000000    2.48s`	`BenchmarkGood-4   100000000    0.21s`

Style

Avoid overly long lines

Avoid lines of code that require readers to scroll horizontally or turn their heads too much.

We recommend a soft line length limit of **99 characters**. Authors should aim to wrap lines before hitting this limit, but it is not a hard limit. Code is allowed to exceed this limit.

Be Consistent

Some of the guidelines outlined in this document can be evaluated objectively; others are situational, contextual, or subjective.

Above all else, **be consistent**.

Consistent code is easier to maintain, is easier to rationalize, requires less cognitive overhead, and is easier to migrate or update as new conventions emerge or classes of bugs are fixed.

Conversely, having multiple disparate or conflicting styles within a single codebase causes maintenance overhead, uncertainty, and cognitive dissonance, all of which can directly contribute to lower velocity, painful code reviews, and bugs.

When applying these guidelines to a codebase, it is recommended that changes are made at a package (or larger) level: application at a sub-package level violates the above concern by introducing multiple styles into the same code.

Group Similar Declarations

Go supports grouping similar declarations.

Bad	Good

import "a"
import "b"
	import (
	  "a"
	  "b"
	)

This also applies to constants, variables, and type declarations.

Bad	Good

const a = 1
const b = 2
 	
var a = 1
var b = 2
 	
type Area float64
type Volume float64
	const (
	  a = 1
	  b = 2
	)
	
	var (
	  a = 1
	  b = 2
	)
	
	type (
	  Area float64
	  Volume float64
	)

Only group related declarations. Do not group declarations that are unrelated.

Bad	Good

type Operation int
 	
const (
  Add Operation = iota + 1
  Subtract
  Multiply
  EnvVar = "MY_ENV"
)
	type Operation int
	
	const (
	  Add Operation = iota + 1
	  Subtract
	  Multiply
	)
	
	const EnvVar = "MY_ENV"

Groups are not limited in where they can be used. For example, you can use them inside of functions.

Bad	Good

func f() string {
  red := color.New(0xff0000)
	func f() string {
	  var (

Bad	Good
green := color.New(0x00ff00) blue := color.New(0x0000ff) // ... }	red = color.New(0xff0000) green = color.New(0x00ff00) blue = color.New(0x0000ff)) // ... }

Exception: Variable declarations, particularly inside functions, should be grouped together if declared adjacent to other variables. Do this for variables declared together even if they are unrelated.

Bad	Good

func (c *client) request() {
caller := c.name
format := "json"
timeout := 5*time.Second
var err error
 	
// ...
}
	func (c *client) request() {
	var (
	caller = c.name
	format = "json"
	timeout = 5*time.Second
	err error
	)
	
	// ...
	}

[Import Group Ordering]

There should be two import groups:

Standard library

Everything else

This is the grouping applied by goimports by default.

Bad	Good
import ("fmt" "os" "go.uber.org/atomic" "golang.org/x/sync/errgroup")	import ("fmt" "os" "go.uber.org/atomic" "golang.org/x/sync/errgroup")

[Package Names]

When naming packages, choose a name that is:

All lower-case. No capitals or underscores.

Does not need to be renamed using named imports at most call sites.

Short and succinct. Remember that the name is identified in full at every call site.

Not plural. For example, `net/url`, not `net/urls`.

Not "common", "util", "shared", or "lib". These are bad, uninformative names.

See also [Package Names] and [Style guideline for Go packages].

Function Names

We follow the Go community's convention of using MixedCaps for function names. An exception is made for test functions, which may contain underscores for the purpose of grouping related test cases, e.g., `TestMyFunction_WhatIsBeingTested`.

Import Aliasing

Import aliasing must be used if the package name does not match the last element of the import path.

```
import (
  "net/http"


  client "example.com/client-go"
  trace "example.com/trace/v2"
)
```

In all other scenarios, import aliases should be avoided unless there is a direct conflict between imports.

<table>
<tr><th>Bad</th><th>Good</th></tr>
<tr><td><pre>import (
  "fmt"
  "os"

  nettrace "golang.net/x/trace"
)</pre></td><td><pre>import (
  "fmt"
  "os"
  "runtime/trace"

  nettrace "golang.net/x/trace"
)</pre></td></tr>
</table>

Function Grouping and Ordering

Functions should be sorted in rough call order.

Functions in a file should be grouped by receiver.

Therefore, exported functions should appear first in a file, after `struct`, `const`, `var` definitions.

A `newXYZ()`/`NewXYZ()` may appear after the type is defined, but before the rest of the methods on the receiver.

Since functions are grouped by receiver, plain utility functions should appear towards the end of the file.

Bad	Good

func (s *something) Cost() {
  return calcCost(s.weights)
}
 	
type something struct{ ... }
 	
func calcCost(n []int) int {...}
 	
func (s *something) Stop() {...}
 	
func newSomething() *something {
    return &something{}
}
	type something struct{ ... }
	
	func newSomething() *something {
	    return &something{}
	}
	
	func (s *something) Cost() {
	  return calcCost(s.weights)
	}
	
	func (s *something) Stop() {...}
	
	func calcCost(n []int) int {...}

Reduce Nesting

Code should reduce nesting where possible by handling error cases/special conditions first and returning early or continuing the loop. Reduce the amount of code that is nested multiple levels.

Bad	Good

for _, v := range data {
  if v.F1 == 1 {
    v = process(v)
    if err := v.Call(); err == nil {
      v.Send()
    } else {
      return err
    }
  } else {
    log.Printf("Invalid v: %v", v)
  }
}
	for _, v := range data {
	  if v.F1 != 1 {
	    log.Printf("Invalid v: %v", v)
	    continue
	  }
	
	  v = process(v)
	  if err := v.Call(); err != nil {
	    return err
	  }
	  v.Send()
	}

Unnecessary Else

If a variable is set in both branches of an if, it can be replaced with a single if.

Bad	Good

Bad	Good

var a int
if b {
  a = 100
} else {
  a = 10
}
	a := 10
	if b {
	  a = 100
	}

Top-level Variable Declarations

At the top level, use the standard `var` keyword. Do not specify the type, unless it is not the same type as the expression.

Bad	Good

var _s string = F()
 	
func F() string { return "A" }
	var _s = F()
	// Since F already states that it returns a string, we don't need to specify
	// the type again.
	
	func F() string { return "A" }

Specify the type if the type of the expression does not match the desired type exactly.

```
type myError struct{}

func (myError) Error() string { return "error" }

func F() myError { return myError{} }

var _e error = F()
// F returns an object of type myError but we want error.
```

Prefix Unexported Globals with _

Prefix unexported top-level `var`s and `const`s with `_` to make it clear when they are used that they are global symbols.

Rationale: Top-level variables and constants have a package scope. Using a generic name makes it easy to accidentally use the wrong value in a different file.

Bad	Good
// foo.go const (	// foo.go const (

Bad	Good

`  defaultPort = 8080`
`  defaultUser = "user"`
`)`
 	
`// bar.go`
 	
`func Bar() {`
`  defaultPort := 9090`
`  ...`
`  fmt.Println("Default port", defaultPort)`
 	
`  // We will not see a compile error if the first line of`
`  // Bar() is deleted.`
`}`
	`  _defaultPort = 8080`
	`  _defaultUser = "user"`
	`)`

Exception: Unexported error values may use the prefix `err` without the underscore. See Error Naming.

Embedding in Structs

Embedded types should be at the top of the field list of a struct, and there must be an empty line separating embedded fields from regular fields.

Bad	Good

`type Client struct {`
`  version int`
`  http.Client`
`}`
	`type Client struct {`
	`  http.Client`
	
	`  version int`
	`}`

Embedding should provide tangible benefit, like adding or augmenting functionality in a semantically-appropriate way. It should do this with zero adverse user-facing effects (see also: Avoid Embedding Types in Public Structs).

Exception: Mutexes should not be embedded, even on unexported types. See also: Zero-value Mutexes are Valid.

Embedding **should not**:

Be purely cosmetic or convenience-oriented.

Make outer types more difficult to construct or use.

Affect outer types' zero values. If the outer type has a useful zero value, it should still have a useful zero value after embedding the inner type.

Expose unrelated functions or fields from the outer type as a side-effect of embedding the inner type.

Expose unexported types.

Affect outer types' copy semantics.

Change the outer type's API or type semantics.

Embed a non-canonical form of the inner type.

Expose implementation details of the outer type.

Allow users to observe or control type internals.

Change the general behavior of inner functions through wrapping in a way that would reasonably surprise users.

Simply put, embed consciously and intentionally. A good litmus test is, "would all of these exported inner methods/fields be added directly to the outer type"; if the answer is "some" or "no", don't embed the inner type - use a field instead.

<table>
<tr><th>Bad</th><th>Good</th></tr>
<tr><td>

```
type A struct {
    // Bad: A.Lock() and A.Unlock() are
    //      now available, provide no
    //      functional benefit, and allow
    //      users to control details about
    //      the internals of A.
    sync.Mutex
}
```

</td><td>

```
type countingWriteCloser struct {
    // Good: Write() is provided at this
    //       outer layer for a specific
    //       purpose, and delegates work
    //       to the inner type's Write().
    io.WriteCloser

    count int
}

func (w *countingWriteCloser) Write(bs []byte) (int, error)
{
    w.count += len(bs)
    return w.WriteCloser.Write(bs)
}
```

</td></tr>
</table>

Bad	Good

type Book struct {
  // Bad: pointer changes zero value usefulness
  io.ReadWriter
 	
  // other fields
}
 	
// later
 	
var b Book
b.Read(...) // panic: nil pointer
b.String()   // panic: nil pointer
b.Write(...) // panic: nil pointer
	type Book struct {
	  // Good: has useful zero value
	  bytes.Buffer
	
	  // other fields
	}
	
	// later
	
	var b Book
	b.Read(...) // ok
	b.String() // ok
	b.Write(...) // ok

type Client struct {
  sync.Mutex
  sync.WaitGroup
  bytes.Buffer
  url.URL
}
	type Client struct {
	  mtx sync.Mutex
	  wg sync.WaitGroup
	  buf bytes.Buffer
	  url url.URL
	}

Local Variable Declarations

Short variable declarations (`:=`) should be used if a variable is being set to some value explicitly.

Bad	Good
var s = "foo"	s := "foo"

However, there are cases where the default value is clearer when the `var` keyword is used. Declaring Empty Slices, for example.

Bad	Good

func f(list []int) {
  filtered := []int{}
  for _, v := range list {
    if v > 10 {
      filtered = append(filtered, v)
    }
  }
}
	func f(list []int) {
	  var filtered []int
	  for _, v := range list {
	    if v > 10 {
	      filtered = append(filtered, v)
	    }
	  }
	}

nil is a valid slice

`nil` is a valid slice of length 0. This means that,

You should not return a slice of length zero explicitly. Return `nil` instead.

Bad	Good

if x == "" {
  return []int{}
}
	if x == "" {
	  return nil
	}

To check if a slice is empty, always use `len(s) == 0`. Do not check for `nil`.

Bad	Good

func isEmpty(s []string) bool {
  return s == nil
}
	func isEmpty(s []string) bool {
	  return len(s) == 0
	}

The zero value (a slice declared with `var`) is usable immediately without `make()`.

Bad	Good

nums := []int{}
// or, nums := make([]int)
 	
if add1 {
  nums = append(nums, 1)
}
 	
if add2 {
  nums = append(nums, 2)
}
	var nums []int
	
	if add1 {
	  nums = append(nums, 1)
	}
	
	if add2 {
	  nums = append(nums, 2)
	}

Remember that, while it is a valid slice, a nil slice is not equivalent to an allocated slice of length 0 - one is nil and the other is not - and the two may be treated differently in different situations (such as serialization).

Reduce Scope of Variables

Where possible, reduce scope of variables. Do not reduce the scope if it conflicts with Reduce Nesting.

Bad	Good

err := os.WriteFile(name, data, 0644)
if err != nil {
 return err
}
	if err := os.WriteFile(name, data, 0644); err != nil {
	 return err
	}

If you need a result of a function call outside of the if, then you should not try to reduce the scope.

Bad	Good

if data, err := os.ReadFile(name); err == nil {
  err = cfg.Decode(data)
  if err != nil {
    return err
  }
 	
  fmt.Println(cfg)
  return nil
} else {
  return err
}
	data, err := os.ReadFile(name)
	if err != nil {
	   return err
	}
	
	if err := cfg.Decode(data); err != nil {
	  return err
	}
	
	fmt.Println(cfg)
	return nil

Avoid Naked Parameters

Naked parameters in function calls can hurt readability. Add C-style comments (`/* ... */`) for parameter names when their meaning is not obvious.

Bad	Good
// func printInfo(name string, isLocal, done bool) printInfo("foo", true, true)	// func printInfo(name string, isLocal, done bool) printInfo("foo", true /* isLocal */, true /* done */)

Better yet, replace naked `bool` types with custom types for more readable and type-safe code. This allows more than just two states (true/false) for that parameter in the future.

```
type Region int

const (
  UnknownRegion Region = iota
  Local
)

type Status int

const (
  StatusReady Status = iota + 1
  StatusDone
  // Maybe we will have a StatusInProgress in the future.
)
```

func printInfo(name string, region Region, status Status)

Use Raw String Literals to Avoid Escaping

Go supports raw string literals, which can span multiple lines and include quotes. Use these to avoid hand-escaped strings which are much harder to read.

Bad	Good
wantError := "unknown name:\"test\""	wantError := `unknown error:"test"`

Initializing Structs

Use Field Names to Initialize Structs

You should almost always specify field names when initializing structs. This is now enforced by `go vet`.

Bad	Good

k := User{"John", "Doe", true}	k := User{
	    FirstName: "John",
	    LastName: "Doe",
	    Admin: true,
	}

Exception: Field names *may* be omitted in test tables when there are 3 or fewer fields.

```
tests := []struct{
  op Operation
  want string
}{
  {Add, "add"},
  {Subtract, "subtract"},
}
```

Omit Zero Value Fields in Structs

When initializing structs with field names, omit fields that have zero values unless they provide meaningful context. Otherwise, let Go set these to zero values automatically.

Bad	Good

user := User{
  FirstName: "John",
  LastName: "Doe",
	user := User{
	  FirstName: "John",
	  LastName: "Doe",

Bad	Good
MiddleName: "", Admin: false, }	}

This helps reduce noise for readers by omitting values that are default in that context. Only meaningful values are specified.

Include zero values where field names provide meaningful context. For example, test cases in [Test Tables] can benefit from names of fields even when they are zero-valued.

```
tests := []struct{
  give string
  want int
}{
  {give: "0", want: 0},
  // ...
}
```

Use `var` for Zero Value Structs

When all the fields of a struct are omitted in a declaration, use the `var` form to declare the struct.

Bad	Good
user := User{}	var user User

This differentiates zero valued structs from those with non-zero fields similar to the distinction created for map initialization, and matches how we prefer to declare empty slices.

Initializing Struct References

Use `&T{}` instead of `new(T)` when initializing struct references so that it is consistent with the struct initialization.

Bad	Good
sval := T{Name: "foo"} // inconsistent sptr := new(T) sptr.Name = "bar"	sval := T{Name: "foo"} sptr := &T{Name: "bar"}

Initializing Maps

Prefer `make(..)` for empty maps, and maps populated programmatically. This makes map initialization visually distinct from declaration, and it makes it easy to add size hints later if available.

<table>
<tr><th>Bad</th><th>Good</th></tr>
<tr><td>

```
var (
  // m1 is safe to read and write;
  // m2 will panic on writes.
  m1 = map[T1]T2{}
  m2 map[T1]T2
)
```

</td><td>

```
var (
  // m1 is safe to read and write;
  // m2 will panic on writes.
  m1 = make(map[T1]T2)
  m2 map[T1]T2
)
```

</td></tr>
<tr><td>Declaration and initialization are visually similar.</td><td>Declaration and initialization are visually distinct.</td></tr>
</table>

Where possible, provide capacity hints when initializing maps with `make()`. See Specifying Map Capacity Hints for more information.

On the other hand, if the map holds a fixed list of elements, use map literals to initialize the map.

<table>
<tr><th>Bad</th><th>Good</th></tr>
<tr><td>

```
m := make(map[T1]T2, 3)
m[k1] = v1
m[k2] = v2
m[k3] = v3
```

</td><td>

```
m := map[T1]T2{
  k1: v1,
  k2: v2,
  k3: v3,
}
```

</td></tr>
</table>

The basic rule of thumb is to use map literals when adding a fixed set of elements at initialization time, otherwise use `make` (and specify a size hint if available).

Format Strings outside Printf

If you declare format strings for `Printf`-style functions outside a string literal, make them `const` values.

This helps `go vet` perform static analysis of the format string.

<table>
<tr><th>Bad</th><th>Good</th></tr>
<tr><td>

```
msg := "unexpected values %v, %v\n"
fmt.Printf(msg, 1, 2)
```

</td><td>

```
const msg = "unexpected values %v, %v\n"
fmt.Printf(msg, 1, 2)
```

</td></tr>
</table>

Naming Printf-style Functions

When you declare a `Printf`-style function, make sure that `go vet` can detect it and check the format string.

This means that you should use predefined `Printf`-style function names if possible. `go vet` will check these by default. See [Printf family] for more information.

If using the predefined names is not an option, end the name you choose with f: `Wrapf`, not `Wrap`. `go vet` can be asked to check specific `Printf`-style names but they must end with f.

go vet -printfuncs=wrapf,statusf

See also go vet: Printf family check.

Patterns

Test Tables

Table-driven tests with subtests can be a helpful pattern for writing tests to avoid duplicating code when the core test logic is repetitive.

If a system under test needs to be tested against *multiple conditions* where certain parts of the the inputs and outputs change, a table-driven test should be used to reduce redundancy and improve readability.

Bad	Good

// func TestSplitHostPort(t *testing.T)
 	
host, port, err := net.SplitHostPort("192.0.2.0:8000")
require.NoError(t, err)
assert.Equal(t, "192.0.2.0", host)
assert.Equal(t, "8000", port)
 	
host, port, err = net.SplitHostPort("192.0.2.0:http")
require.NoError(t, err)
assert.Equal(t, "192.0.2.0", host)
assert.Equal(t, "http", port)
 	
host, port, err = net.SplitHostPort(":8000")
require.NoError(t, err)
assert.Equal(t, "", host)
assert.Equal(t, "8000", port)
 	
host, port, err = net.SplitHostPort("1:8")
require.NoError(t, err)
assert.Equal(t, "1", host)
assert.Equal(t, "8", port)
	// func TestSplitHostPort(t *testing.T)
	
	tests := []struct{
	  give string
	  wantHost string
	  wantPort string
	}{
	  {
	    give: "192.0.2.0:8000",
	    wantHost: "192.0.2.0",
	    wantPort: "8000",
	  },
	  {
	    give: "192.0.2.0:http",
	    wantHost: "192.0.2.0",
	    wantPort: "http",
	  },
	  {
	    give: ":8000",
	    wantHost: "",
	    wantPort: "8000",
	  },
	  {
	    give: "1:8",
	    wantHost: "1",
	    wantPort: "8",
	  },

Bad	Good

	`}`
	
	`for _, tt := range tests {`
	`  t.Run(tt.give, func(t *testing.T) {`
	`    host, port, err := net.SplitHostPort(tt.give)`
	`    require.NoError(t, err)`
	`    assert.Equal(t, tt.wantHost, host)`
	`    assert.Equal(t, tt.wantPort, port)`
	`  })`
	`}`

Test tables make it easier to add context to error messages, reduce duplicate logic, and add new test cases.

We follow the convention that the slice of structs is referred to as `tests` and each test case `tt`. Further, we encourage explicating the input and output values for each test case with `give` and `want` prefixes.

```
tests := []struct{
  give     string
  wantHost string
  wantPort string
}{
  // ...
}

for _, tt := range tests {
  // ...
}
```

[Avoid Unnecessary Complexity in Table Tests]

Table tests can be difficult to read and maintain if the subtests contain conditional assertions or other branching logic. Table tests should **NOT** be used whenever there needs to be complex or conditional logic inside subtests (i.e. complex logic inside the `for` loop).

Large, complex table tests harm readability and maintainability because test readers may have difficulty debugging test failures that occur.

Table tests like this should be split into either multiple test tables or multiple individual `Test...` functions.

Some ideals to aim for are:

Focus on the narrowest unit of behavior

Minimize "test depth", and avoid conditional assertions (see below)

Ensure that all table fields are used in all tests

Ensure that all test logic runs for all table cases

In this context, "test depth" means "within a given test, the number of successive assertions that require previous assertions to hold" (similar to cyclomatic complexity). Having "shallower" tests means that there are fewer relationships between assertions and, more importantly, that those assertions are less likely to be conditional by default.

Concretely, table tests can become confusing and difficult to read if they use multiple branching pathways (e.g. `shouldError`, `expectCall`, etc.), use many `if` statements for specific mock expectations (e.g. `shouldCallFoo`), or place functions inside the table (e.g. `setupMocks func(*FooMock)`).

However, when testing behavior that only changes based on changed input, it may be preferable to group similar cases together in a table test to better illustrate how behavior changes across all inputs, rather than splitting otherwise comparable units into separate tests and making them harder to compare and contrast.

If the test body is short and straightforward, it's acceptable to have a single branching pathway for success versus failure cases with a table field like `shouldErr` to specify error expectations.

Bad	Good

func TestComplicatedTable(t *testing.T) {
  tests := []struct {
    give   string
    want   string
    wantErr   error
    shouldCallX   bool
    shouldCallY   bool
    giveXResponse string
    giveXErr   error
    giveYResponse string
    giveYErr   error
  }{
    // ...
  }
 	
  for _, tt := range tests {
    t.Run(tt.give, func(t *testing.T) {
      // setup mocks
      ctrl := gomock.NewController(t)
      xMock := xmock.NewMockX(ctrl)
      if tt.shouldCallX {
        xMock.EXPECT().Call().Return(
          tt.giveXResponse, tt.giveXErr,
)
	func TestShouldCallX(t *testing.T) {
	  // setup mocks
	  ctrl := gomock.NewController(t)
	  xMock := xmock.NewMockX(ctrl)
	  xMock.EXPECT().Call().Return("XResponse", nil)
	
	  yMock := ymock.NewMockY(ctrl)
	
	  got, err := DoComplexThing("inputX", xMock, yMock)
	
	  require.NoError(t, err)
	  assert.Equal(t, "want", got)
	}
	
	func TestShouldCallYAndFail(t *testing.T) {
	  // setup mocks
	  ctrl := gomock.NewController(t)
	  xMock := xmock.NewMockX(ctrl)
	
	  yMock := ymock.NewMockY(ctrl)
	  yMock.EXPECT().Call().Return("YResponse", nil)
	
	  _, err := DoComplexThing("inputY", xMock,

Bad	Good

}
yMock := ymock.NewMockY(ctrl)
if tt.shouldCallY {
yMock.EXPECT().Call().Return(
tt.giveYResponse, tt.giveYErr,
)
}
 	
got, err := DoComplexThing(tt.give, xMock, yMock)
 	
// verify results
if tt.wantErr != nil {
require.EqualError(t, err, tt.wantErr)
return
}
require.NoError(t, err)
assert.Equal(t, want, got)
})
}
}
	yMock)
	assert.EqualError(t, err, "Y failed")
	}

This complexity makes it more difficult to change, understand, and prove the correctness of the test.

While there are no strict guidelines, readability and maintainability should always be top-of-mind when deciding between Table Tests versus separate tests for multiple inputs/outputs to a system.

Parallel Tests

Parallel tests, like some specialized loops (for example, those that spawn goroutines or capture references as part of the loop body), must take care to explicitly assign loop variables within the loop's scope to ensure that they hold the expected values.

```
tests := []struct{
  give string
  // ...
}{
  // ...
}

for _, tt := range tests {
  tt := tt // for t.Parallel
  t.Run(tt.give, func(t *testing.T) {
```

```
    t.Parallel()
    // ...
  })
}
```

In the example above, we must declare a `tt` variable scoped to the loop iteration because of the use of `t.Parallel()` below. If we do not do that, most or all tests will receive an unexpected value for `tt`, or a value that changes as they're running.

Functional Options

Functional options is a pattern in which you declare an opaque `Option` type that records information in some internal struct. You accept a variadic number of these options and act upon the full information recorded by the options on the internal struct.

Use this pattern for optional arguments in constructors and other public APIs that you foresee needing to expand, especially if you already have three or more arguments on those functions.

<table>
<tr><th>Bad</th><th>Good</th></tr>
<tr><td>

```
// package db

func Open(
  addr string,
  cache bool,
  logger *zap.Logger
) (*Connection, error) {
  // ...
}
```

</td><td>

```
// package db

type Option interface {
  // ...
}

func WithCache(c bool) Option {
  // ...
}

func WithLogger(log *zap.Logger) Option {
  // ...
}

// Open creates a connection.
func Open(
  addr string,
  opts ...Option,
) (*Connection, error) {
  // ...
}
```

</td></tr>
<tr><td>

The cache and logger parameters must always be provided, even if the user wants to use the default.

```
db.Open(addr, db.DefaultCache, zap.NewNop())
db.Open(addr, db.DefaultCache, log)
```

</td><td>

Options are provided only if needed.

```
db.Open(addr)
db.Open(addr, db.WithLogger(log))
```

</td></tr>
</table>

Bad	Good

db.Open(addr, false /* cache */, zap.NewNop())
db.Open(addr, false /* cache */, log)
	db.Open(addr, db.WithCache(false))
	db.Open(
	  addr,
	  db.WithCache(false),
	  db.WithLogger(log),
	)

Our suggested way of implementing this pattern is with an `Option` interface that holds an unexported method, recording options on an unexported `options` struct.

```
type options struct {
  cache  bool
  logger *zap.Logger
}

type Option interface {
  apply(*options)
}

type cacheOption bool

func (c cacheOption) apply(opts *options) {
  opts.cache = bool(c)
}

func WithCache(c bool) Option {
  return cacheOption(c)
}

type loggerOption struct {
  Log *zap.Logger
}

func (l loggerOption) apply(opts *options) {
  opts.logger = l.Log
}

func WithLogger(log *zap.Logger) Option {
```

```
  return loggerOption{Log: log}
}

// Open creates a connection.
func Open(
  addr string,
  opts ...Option,
) (*Connection, error) {
  options := options{
    cache:  defaultCache,
    logger: zap.NewNop(),
  }

  for _, o := range opts {
    o.apply(&options)
  }

  // ...
}
```

Note that there's a method of implementing this pattern with closures but we believe that the pattern above provides more flexibility for authors and is easier to debug and test for users. In particular, it allows options to be compared against each other in tests and mocks, versus closures where this is impossible. Further, it lets options implement other interfaces, including `fmt.Stringer` which allows for user-readable string representations of the options.

See also,

[Self-referential functions and the design of options]

[Functional options for friendly APIs]

Linting

More importantly than any "blessed" set of linters, lint consistently across a codebase.

We recommend using the following linters at a minimum, because we feel that they help to catch the most common issues and also establish a high bar for code quality without being unnecessarily prescriptive:

errcheck to ensure that errors are handled

goimports to format code and manage imports golint to point out common style mistakes

govet to analyze code for common mistakes

staticcheck to do various static analysis checks

Lint Runners

We recommend golangci-lint as the go-to lint runner for Go code, largely due to its performance in larger codebases and ability to configure and use many canonical linters at once. This repo has an example .golangci.yml config file with recommended linters and settings.

golangci-lint has various linters available for use. The above linters are recommended as a base set, and we encourage teams to add any additional linters that make sense for their projects.