


# The Go Programming Language

Simplicity - Design goal of Go	3
Trading features for simplicity	3
Simpler type system	4
Object-oriented programming	4
Introduction	4
++ / --	4
for	4
<b>2. Program structure</b>	<b>4</b>
Built-ins	4
Names	5
2.2 Declarations	5
2.3 Variables	5
Zero values	6
2.3.2 POINTERS	6
2.3.3 THE NEW FUNCTION	7
2.3.4 LIFETIME OF VARIABLES	7
2.4 Assignments	8
Operators with additional return values	8
2.5 Type Declarations	8
2.6 Packages and Files	9
2.6.2 PACKAGE INITIALIZATION	9
2.7 Scope	10
Code blocks	10
Lexical scope	10
<b>3. Basic Data Types</b>	<b>12</b>
3.1 Integers	12
Booleans	13
3.5 Strings	13
Unicode	14
String literals	14
Byte slices	15
Constants	15
THE CONSTANT GENERATOR IOTA	15
UNTYPED CONSTANTS	16

<b>4. Composite Types</b>	<b>16</b>
4.1 Arrays	16
4.2 Slices	18
Slice Comparison	19
Built-in functions	19
4.3 Maps	19
4.4 Structs	21
struct literal	22
Comparability	22
STRUCT EMBEDDING AND ANONYMOUS FIELDS	22
4.5 JSON	23
4.6 Text and HTML Templates	24
<b>5. Functions</b>	<b>25</b>
5.4 Errors	26
5.5 Function Values	27
5.6 Anonymous Functions	27
5.7 Variadic Functions	28
5.8 Deferred Function Calls	29
5.9 Panic	29
5.10 Recover	30
<b>6. Methods (OOP)</b>	<b>31</b>
6.1 Method Declarations	31
6.2 Methods with a Pointer Receiver	32
Receiver Type	33
NIL IS A VALID RECEIVER VALUE	33
6.3 Composing Types by Struct Embedding	33
6.4 Method Values and Expressions	35
6.6 Encapsulation	35
<b>7. Interfaces</b>	<b>37</b>
7.1 Interfaces as Contracts	38
7.3 Interface Satisfaction	38
7.5 Interface Values	40
7.10 Type Assertions	42
7.13 Type Switches	43
7.15 A Few Words of Advice 	45
<b>8. Goroutines and Channels</b>	<b>46</b>
8.1 Goroutines	46

GOROUTINES HAVE NO IDENTITY	47
8.4 Channels	47
8.4.1 UNBUFFERED CHANNELS	49
8.4.2 PIPELINES	50
8.4.3 UNIDIRECTIONAL CHANNEL TYPES	50
8.4.4 BUFFERED CHANNELS	50
Buffered or Unbuffered	51
8.5 Looping in Parallel	52
8.7 Multiplexing with select	53
8.9 Cancellation	54
<b>9. Concurrency with Shared Variables</b>	<b>55</b>
9.1 Race Conditions	55
9.2 Mutual Exclusion: sync.Mutex	56
9.3 Read/Write Mutexes: sync.RWMutex	57
9.4 Memory Synchronization	57
9.5 Lazy Initialization: sync.Once	58
9.6 The Race Detector	58

## Simplicity - Design goal of Go

“Simplicity requires more work at the beginning of a project to reduce an idea to its essence and more discipline over the lifetime of a project to distinguish good changes from bad or pernicious ones. With sufficient effort, a good change can be accommodated without compromising what Fred Brooks called the “conceptual integrity” of the design but a bad change cannot, and a pernicious change trades simplicity for its shallow cousin, convenience. Only through simplicity of design can a system remain stable, secure, and coherent as it grows.”

## Trading features for simplicity

“it has garbage collection, a package system, first-class functions, lexical scope, a system call interface, and immutable strings in which text is generally encoded in UTF-8. But it has comparatively few features and is unlikely to add more. For instance, it has no implicit numeric conversions, no constructors or destructors, no operator overloading, no default parameter values, no inheritance, no generics, no exceptions, no macros, no function annotations, and no thread-local storage.”

## Simpler type system

“Go has enough of a type system to avoid most of the careless mistakes that plague programmers in dynamic languages, but it has a simpler type system than comparable typed languages.”

## Object-oriented programming

“Go has an unusual approach to object-oriented programming. There are no class hierarchies, or indeed any classes; complex object behaviors are created from simpler ones by composition, not inheritance. Methods may be associated with any user-defined type, not just structures, and the relationship between concrete types and abstract types (interfaces) is implicit, so a concrete type may satisfy an interface that the type’s designer was unaware of.”

## Introduction

`++ / --`

“These are statements, not expressions as they are in most languages in the C family, so `j = i++` is illegal, and they are postfix only, so `--i` is not legal either.”

`for`

- “The for loop is the only loop statement in Go.”
- “Any of these parts may be omitted. If there is no initialization and no post, the semicolons may also be omitted”

## 2. Program structure

### Built-ins

- Constants:
  - `true false iota nil`
- Types:
  - `int int8 int16 int32 int64`
  - `uint uint8 uint16 uint32 uint64 uintptr`
  - `float32 float64 complex128 complex64`

- bool byte rune string error
- Functions:
  - make len cap new append copy close delete
  - complex real imag
  - panic recover

## Names

- Length of variable name proportional to its scope
  - There is no limit on name length, but convention and style in Go programs lean toward short names, especially for local variables with small scopes.
  - Generally, the larger the scope of a name, the longer and more meaningful it should be.
- Visibility scope
  - If an entity is declared within a function, it is local to that function.
  - If declared outside of a function, however, it is visible in all files of the package to which it belongs.
- CamelCase
  - The letters of acronyms and initialisms like ASCII and HTML are always rendered in the same case, so a function might be called `htmlEscape`, `HTMLEscape`, or `escapeHTML`, but not `escapeHtml`.

## 2.2 Declarations

- There are four major kinds of declarations:
  - var, const, type, and func.
- Structure of a file
  - Each file begins with a package declaration that says what package the file is part of.
  - The package declaration is followed by any import declarations,
  - and then a sequence of package-level declarations of types, variables, constants, and functions, in any order.

## 2.3 Variables

- Each declaration has the general form
  - `var name type = expression`
- Either the type or the `= expression` part may be omitted, but not both.
  - If the type is omitted, it is determined by the initializer expression.
- Omitting the type allows declaration of multiple variables of different types:
  - `var b, f, s = true, 2.3, "four" // bool, float64, string`
- *Within a function*, an alternate form called a short variable declaration may be used to declare and initialize local variables.

- It takes the form `name := expression`
- the type of name is determined by the type of expression.
- A var declaration tends to be reserved for local variables that need an explicit type that differs from that of the initializer expression, or for when the variable will be assigned a value later and its initial value is unimportant.
  - `var boiling float64 = 100 // a float64`
- Keep in mind that `:=` is a declaration, whereas `=` is an assignment.
- One subtle but important point: a short variable declaration does not necessarily declare all the variables on its left-hand side.
  - If some of them were already declared in the same lexical block, then the short variable declaration acts like an assignment to those variables.
    - declarations in an outer block are ignored
  - A short variable declaration must declare at least one new variable

## Zero values

- If the expression is omitted, the initial value is the zero value for the type
  - which is 0 for numbers,
  - false for booleans,
  - "" for strings, and
  - nil for interfaces and reference types (pointer, slice, map, channel, function).
  - aggregate type like an array or a struct has the zero value of all of its elements or fields.
- in Go there is no such thing as an uninitialized variable
  - Package-level variables are initialized before main begins
  - local variables are initialized as their declarations are encountered during function execution.

## 2.3.2 POINTERS

- Not every value has an address (e.g. constants),
  - but every variable does.
- The zero value for a pointer of any type is nil.
  - The test `p != nil` is true if p points to a variable.
- Pointers are comparable;
  - two pointers are equal if and only if they point to the same variable
  - or both are nil.
- It is perfectly safe for a function to return the address of a local variable.
  - the local variable created by this particular call will remain in existence even after the call has returned
- Each time we take the address of a variable or copy a pointer, we create new aliases or ways to identify the same variable.

- aliasing also occurs when we copy values of other reference types like slices, maps, and channels, and even structs, arrays, and interfaces that contain these types.

### 2.3.3 THE NEW FUNCTION

- The expression `new(T)`
  - creates an unnamed variable of type `T`,
  - initializes it to the zero value of `T`, and
  - returns its address, which is a value of type `*T`.
- new is only a syntactic convenience, not a fundamental notion
  - A variable created with `new` is no different from an ordinary local variable whose address is taken,
  - except that there's no need to invent (and declare) a dummy name,
  - and we can use `new(T)` in an expression.
  - The `new` function is relatively rarely used
    - because the most common unnamed variables are of struct types, for which the struct literal syntax is more flexible.
- Each call to `new` returns a distinct variable with a unique address
  - `new(int) != new(int)`
- Since `new` is a predeclared function, not a keyword,
  - it's possible to redefine the name for something else within a function

### 2.3.4 LIFETIME OF VARIABLES

- The lifetime of a package-level variable
  - is the entire execution of the program
- local variables have dynamic lifetimes:
  - a new instance is created each time the declaration statement is executed, and
  - the variable lives on until it becomes unreachable,
    - at which point its storage may be recycled.
  - Function parameters and results are local variables too;
    - they are created each time their enclosing function is called.
- keeping unnecessary pointers to short-lived objects within long-lived objects,
  - especially global variables,
  - will prevent the garbage collector from reclaiming the short-lived objects.
- Go's garbage collector recycles unused memory,
  - but do not assume it will release unused operating system resources
    - like open files and network connections.
  - They should be closed explicitly.

## 2.4 Assignments

- An assignment, explicit or implicit, is always legal if the left-hand side (the variable) and the right-hand side (the value) have the same type.
  - nil may be assigned to any variable of interface or reference type
- tuple assignment,
  - allows several variables to be assigned at once.
  - All of the right-hand side expressions are evaluated before any of the variables are updated
- as a matter of style,
  - avoid the tuple form if the expressions are complex;
  - a sequence of separate statements is easier to read.

## Operators with additional return values

- there are three operators that return additional results to indicate some kind of error.
- If these appear in an assignment in which two results are expected, each produces an additional boolean result
  - a map lookup
    - `v, ok = m[key]` `// map lookup`
  - type assertion
    - `v, ok = x.(T)` `// type assertion`
  - channel receive
    - `v, ok = <-ch` `// channel receive`

## 2.5 Type Declarations

- A named type may provide notational convenience
  - if it helps avoid writing out complex types over and over again.
  - The advantage is small when the underlying type is simple like float64,
  - but big for complicated types e.g structs.
- The named type provides a way to separate different and perhaps incompatible uses of the underlying type
  - so that they can't be mixed unintentionally.
    - E.g Celsius and Fahrenheit with underlying type of float64
  - Distinguishing the types makes it possible to avoid errors like inadvertently combining the types
- For every type T, there is a corresponding conversion operation T(x) that converts the value x to type T.
  - A conversion from one type to another is allowed if both have the same underlying type, or



- if both are unnamed pointer types that point to variables of the same underlying type;
  - these conversions change the type but not the representation of the value
    - For instance, converting a floating-point number to an integer discards any fractional part, and
    - converting a string to a []byte slice allocates a copy of the string data.
  - In any case, a conversion never fails at run time.
  - If x is assignable to T, a conversion is permitted but is usually redundant
- The underlying type of a named type determines its structure and representation,
  - and also the set of intrinsic operations it supports,
  - which are the same as if the underlying type had been used directly.
- Comparison operators (like == and <) can also be used to compare a value of a named type to another of the same type,
  - or to a value of an unnamed type (e.g. constants?) with the same underlying type.
  - But two values of different named types cannot be compared directly

## 2.6 Packages and Files

- Each package serves as a separate name space for its declarations.
  - By convention, a package's name matches the last segment of its import path
- The doc comment immediately preceding the package declaration documents the package as a whole.
  - Only one file in each package should have a package doc comment.
  - Extensive doc comments are often placed in a file of their own, conventionally called doc.go
- the go build tool treats a package specially if its import path contains a path segment named `internal`.
  - Such packages are called internal packages.
  - An internal package may be imported only by another package that is inside the tree rooted at the parent of the internal directory.
  - For example, given the packages below, `net/http/internal/chunked` can be imported from `net/http/httputil` or `net/http`, but not from `net/url`.
    - However, `net/url` may import `net/http/httputil`

### 2.6.2 PACKAGE INITIALIZATION

- Package initialization begins by initializing package-level variables in the order in which they are declared,
  - except that dependencies are resolved first
  - If the package has multiple `.go` files,
    - they are initialized in the order in which the files are given to the compiler;
    - the go tool sorts `.go` files by name before invoking the compiler.

- Within each file, `func init() { /* ... */ }` functions are automatically executed when the program starts,
  - Any file may contain any number of init functions
    - They are executed in the order in which they are declared.
  - init functions can't be called explicitly or referenced
- One package is initialized at a time,
  - in the order of imports in the program,
    - dependencies first,
  - so a package p importing q can be sure that q is fully initialized before p's initialization begins.
  - Initialization proceeds from the bottom up;
    - the main package is the last to be initialized.
    - In this manner, all packages are fully initialized before the application's main function begins.

## 2.7 Scope

- A declaration associates a name with a program entity, such as a function or a variable.
- The scope of a declaration is the part of the source code where a use of the declared name refers to that declaration.
- Don't confuse scope with lifetime.
  - The scope of a declaration is a region of the program text;
    - it is a compile-time property.
  - The lifetime of a variable is the range of time during execution when the variable can be referred to by other parts of the program;
    - it is a runtime property.

## Code blocks

- A *syntactic block* is a sequence of statements enclosed in braces like those that surround the body of a function or loop.
- A name declared inside a syntactic block is not visible outside that block.

## Lexical scope

- Lexical block
  - groupings of declarations that are not explicitly surrounded by braces in the source code;
  - There is a lexical block
    - for the entire source code,
      - called the universe block;
    - for each package;
    - for each file;
    - for each for, if, and switch statement;

- for each case in a switch or select statement; and, of course,
  - for each explicit syntactic block.
- A program may contain multiple declarations of the same name so long as each declaration is in a different lexical block.
  - For example, you can declare a local variable with the same name as a package-level variable.
  - Don't overdo it, though; the larger the scope of the redeclaration, the more likely you are to surprise the reader.
- When the compiler encounters a reference to a name,
  - it looks for a declaration, starting with the innermost enclosing lexical block and working up to the universe block.
  - If the compiler finds no declaration,
    - it reports an "undeclared name" error.
  - If a name is declared in both an outer block and an inner block,
    - the inner declaration will be found first.
    - In that case, the inner declaration is said to shadow or hide the outer one, making it inaccessible
- Within a function, lexical blocks may be nested to arbitrary depth,
  - so one local declaration can shadow another.
    - Care should be taken when assigning to package-level variables in nested scopes (e.g. init())
      - E.g. using short declaration could create a locally shadowed variable instead of referencing to the package-level variable
- Universe block
  - The declarations of built-in types, functions, and constants like int, len, and true are in the universe block and
  - can be referred to throughout the entire program.
- package level
  - Declarations outside any function can be referred to from any file in the same package
- File level
  - Imported packages are declared at the file level,
    - so they can be referred to from the same file,
    - but not from another file in the same package without another import.
- Local
  - the variables in the function, are local, so they can be referred to only from within the same function or perhaps just a part of it.
  - The scope of a control-flow label, as used by break, continue, and goto statements, is the entire enclosing function.
- normal practice in Go is to deal with the error in the if block and then return,
  - so that the successful execution path is not indented.

## 3. Basic Data Types

- Go's types fall into four categories:
  - basic types,
    - include numbers, strings, and booleans.
  - aggregate types,
    - arrays and structs
  - reference types,
    - a diverse group that includes
      - pointers
      - slices
      - maps
      - functions
      - channels
  - interface types

### 3.1 Integers

- The type *rune* is a synonym for `int32`
  - conventionally indicates that a value is a Unicode code point.
- the type *byte* is an synonym for `uint8`
  - emphasizes that the value is a piece of raw data rather than a small numeric quantity.
- `int` is not the same type as `int32`,
  - even if the natural size of integers is 32 bits
  - an explicit conversion is required to use an `int` value where an `int32` is needed, and vice versa.
  - Similarly for `uint`
- If the result of an arithmetic operation, whether signed or unsigned, has more bits than can be represented in the result type, it is said to overflow.
  - The high-order bits that do not fit are silently discarded.
  - If the original number is a signed type, the result could be negative if the leftmost bit is a 1
  - E.g.
    - ```
var u uint8 = 255
```
    - ```
fmt.Println(u, u+1, u*u) // "255 0 1"
```
    - ```
var i int8 = 127
```
    - ```
fmt.Println(i, i+1, i*i) // "127 -128 1"
```
- all values of basic type—booleans, numbers, and strings—are comparable,

- meaning that two values of the same type may be compared using the `==` and `!=` operators.
  - Furthermore, integers, floating-point numbers, and strings are ordered by the comparison operators.
  - The values of many other types are not comparable, and
    - no other types are ordered.
- unsigned numbers tend to be used only when their bitwise operators or peculiar arithmetic operators are required,
  - as when implementing bit sets, parsing binary file formats, or for hashing and cryptography.
  - They are typically not used for merely non-negative quantities.
    - E.g. using them in loop indices could result in infinite loops as they overflow and reset
- Integer literals of any size and type can be written as ordinary decimal numbers,
  - or as octal numbers if they begin with 0, as in 0666,
  - or as hexadecimal if they begin with 0x or 0X, as in 0xdeadbeef.

## Booleans


- Boolean values can be combined with the `&&` (AND) and `||` (OR) operators, which have short-circuit behavior:
  - if the answer is already determined by the value of the left operand, the right operand is not evaluated
  - making it safe to write expressions like this:
    - `s != "" && s[0] == 'x'`
- `&&` has higher precedence than `||`
- There is no implicit conversion from a boolean value to a numeric value like 0 or 1, or vice versa.

## 3.5 Strings

- A string is an immutable sequence of bytes.
  - the byte sequence contained in a string value can never be changed,
    - constructions that try to modify a string's data in place are not allowed
  - though of course we can assign a new value to a string variable.
  - Immutability means that it is safe for two copies of a string to share the same underlying memory,
    - making it cheap to copy strings of any length
    - the substring operation is also cheap.
    - No new memory is allocated in either case.
  - 💡 building up strings incrementally can involve a lot of allocation and copying.
    - In such cases, it's more efficient to use the `bytes.Buffer` type
- Strings may be compared with comparison operators like `==` and `<`;

- the comparison is done byte by byte,
  - so the result is the natural lexicographic ordering.
- Four standard packages are particularly important for manipulating strings:
  - bytes
  - strings,
    - provides many functions for searching, replacing, comparing, trimming, splitting, and joining strings
  - strconv
    - provides functions for converting boolean, integer, and floating-point values to and from their string representations, and functions for quoting and unquoting strings.
      - The `fmt.Printf` verbs `%b`, `%d`, `%u`, and `%x` are often more convenient than `Format` functions, especially if we want to include additional information besides the number
  - unicode
    - provides functions like `IsDigit`, `IsLetter`, `IsUpper`, and `IsLower` for classifying runes

## Unicode

- Go's range loop, when applied to a string, performs UTF-8 decoding implicitly.
  - Each time a UTF-8 decoder consumes an unexpected input byte
    - whether explicit in a call to `utf8.DecodeRuneInString` or implicit in a range loop
    - it generates a special Unicode replacement character, `'\uFFFD'`, which is usually printed as a white question mark inside a black hexagonal or diamond-like shape .
    - When a program encounters this rune value, it's often a sign that some upstream part of the system that generated the string data has been careless in its treatment of text encodings.

## String literals

- A string value can be written as a string literal, a sequence of bytes enclosed in double quotes:
  - Because Go source files are always encoded in UTF-8 and
  - Go text strings are conventionally interpreted as UTF-8,
  - we can include Unicode code points in string literals.
  - Within a double-quoted string literal, escape sequences that begin with a backslash `\` can be used to insert arbitrary byte values into the string.
- A raw string literal is written ``...``, using backquotes instead of double quotes.
  - Within a raw string literal, no escape sequences are processed;
  - the contents are taken literally, including backslashes and newlines,

- so a raw string literal may spread over several lines in the program source.

## Byte slices

- A string contains an array of bytes that, once created, is immutable.
- By contrast, the elements of a byte slice can be freely modified.
- Strings can be converted to byte slices and back again
  - the `[]byte(s)` conversion allocates a new byte array holding a copy of the bytes of `s`, and yields a slice that references the entirety of that array
  - The conversion from byte slice back to string with `string(b)` also makes a copy, to ensure immutability of the resulting string
- To avoid conversions and unnecessary memory allocation, many of the utility functions in the `bytes` package directly parallel their counterparts in the `strings` package
- The `bytes` package provides the `Buffer` type for efficient manipulation of byte slices.
  - A `Buffer` starts out empty but grows as data of types like `string`, `byte`, and `[]byte` are written to it.
  - a `bytes.Buffer` variable requires no initialization because its zero value is usable
  - The `bytes.Buffer` type is extremely versatile
  - it may be used as a replacement for a file whenever an I/O function requires
    - a sink for bytes (`io.Writer`) as `Fprintf` does above,
    - or a source of bytes (`io.Reader`).

## Constants

- When a sequence of constants is declared as a group,
  - the right-hand side expression may be omitted for all but the first of the group,
  - implying that the previous expression and its type should be used again.

```
const (
    a = 1
    b
    c = 2
    d
)

fmt.Println(a, b, c, d) // "1 1 2 2"
```

## THE CONSTANT GENERATOR IOTA

- the constant generator `iota` is used to create a sequence of related values without spelling out each one explicitly
  - In a `const` declaration, the value of `iota` begins at zero and increments by one for each item in the sequence

- Can also be used for complex increments e.g. using <<

## UNTYPED CONSTANTS

- Constants in Go are a bit unusual.
  - Although a constant can have any of the basic data types
  - ⚠ many constants are not committed to a particular type.
- There are six flavors of these uncommitted constants, called
  - untyped boolean,
  - untyped integer,
  - untyped rune,
  - untyped floating-point,
  - untyped complex, and
  - untyped string.
- ⚠ Only constants can be untyped.
  - When an untyped constant is assigned to a variable,
    - or appears on the right-hand side of a variable declaration with an explicit type,
    - the constant is implicitly converted to the type of that variable if possible.
  - In a variable declaration without an explicit type (including short variable declarations),
    - the flavor of the untyped constant implicitly determines the default type of the variable

## 4. Composite Types

- four such types
  - arrays,
  - slices,
  - maps, and
  - structs
- Arrays and structs are *aggregate* types;
  - their values are concatenations of other values in memory.
  - Arrays are homogeneous
    - their elements all have the same type
  - whereas structs are heterogeneous.
- Both arrays and structs are fixed size.
  - In contrast, slices and maps are *dynamic* data structures that grow as values are added.

### 4.1 Arrays

- An array is a *fixed-length* sequence of zero or more elements of a *particular type*.



- The size must be a *constant* expression,
  - that is, an expression whose value can be computed as the program is being compiled.
- Because of their fixed length, arrays are rarely used directly in Go.
- Slices, which can grow and shrink, are much more versatile
- The built-in function `len` returns the number of elements in the array.
- We can use an array literal to initialize an array with a list of values
  - In an array literal, if an ellipsis “...” appears in place of the length, the array length is determined by the number of initializers
    - `[...]int{1, 2, 3}`
- ⚠ The size of an array is part of its type,
  - so `[3]int` and `[4]int` are different types
- If an array’s element type is comparable then the array type is comparable too,
  - so we may directly compare two arrays of that type using the `==` operator,
  - which reports whether all corresponding elements are equal.
- 💡 it is also possible to specify a list of index and value pairs

```
type Currency int
```

```
const (
    USD Currency = iota
    EUR
    GBP
    RMB
)
```

```
symbol := [...]string{USD: "$", EUR: "€", GBP: "£", RMB:
"¥"}
```

```
fmt.Println(RMB, symbol[RMB]) // "3 ¥"
```

- In this form, indices can appear in any order and some may be omitted;
  - unspecified values take on the zero value for the element type. For instance,
    - `r := [...]int{99: -1}`
    - defines an array `r` with 100 elements,
      - all zero except for the last, which has value `-1`.
- ⚠ Arrays are not a reference type
  - When a function is called, a copy of each argument value is assigned to the corresponding parameter variable,
  - so the function receives a copy, not the original.
  - Passing large arrays in this way can be inefficient,
    - and any changes that the function makes to array elements affect only the copy, not the original.

- 💡 we can explicitly pass a pointer to an array so that any modifications the function makes to array elements will be visible to the caller

## 4.2 Slices

- Slices represent variable-length sequences whose elements all have the same type.
  - A slice type is written `[]T`, where the elements have type `T`;
  - it looks like an array type without a size
- Arrays and slices are intimately connected.
  - A slice is a lightweight data structure that gives access to a subsequence (or perhaps all) of the elements of an array,
    - which is known as the slice's underlying array.
  - Multiple slices can share the same underlying array
    - and may refer to overlapping parts of that array.
- A slice has three components:
  - a pointer,
    - points to the first element of the array that is reachable through the slice,
      - which is not necessarily the array's first element.
  - a length,
    - is the number of slice elements; it can't exceed the capacity,
  - a capacity
    - usually the number of elements between the start of the slice and the end of the underlying array.
    - The built-in functions `len` and `cap` return those values.
- Slicing beyond `cap(slice)` causes a panic,
  - ⚠ but slicing beyond `len(slice)` extends the slice
    - From the existing first element to the new length
- Since a slice contains a pointer to an element of an array,
  - passing a slice to a function permits the function to modify the underlying array elements.
  - In other words, copying a slice creates an alias for the underlying array.
- A slice literal looks like an array literal,
  - a sequence of values separated by commas and surrounded by braces,
  - but the size is not given.
  - This implicitly creates an array variable of the right size and yields a slice that points to it.
  - As with array literals, slice literals may specify the values in order, or give their indices explicitly, or use a mix of the two styles.
- The built-in function `make` creates a slice of a specified element type, length, and capacity.
  - The capacity argument may be omitted,
    - in which case the capacity equals the length.

- Under the hood, `make` creates an unnamed array variable and returns a slice of it;
  - the array is accessible only through the returned slice.

## Slice Comparison

- ⚠ Unlike arrays, slices are not comparable,
  - so we cannot use `==` to test whether two slices contain the same elements.
  - The standard library provides the highly optimized `bytes.Equal` function for comparing two slices of bytes (`[]byte`),
    - but for other types of slice, we must do the comparison ourselves
- The only legal slice comparison is against nil
  - A nil slice has no underlying array
  - But an empty slice literal `[]int{}`
    - Is not nil
    - But has `len 0`
  - ⚠ So, if you need to test whether a slice is empty, use `len(s) == 0`, not `s == nil`

## Built-in functions

- The built-in `append` function appends items to slices
  - ⚠ `append` returns the updated slice.
    - It is therefore necessary to store the result of `append`
  - The underlying array is doubled in capacity when there is no room for appending
- the built-in function `copy`
  - copies elements from one slice to another of the same type
  - The slices may refer to the same underlying array;
    - they may even overlap

## 4.3 Maps

- a map is a *reference* to a hash table
- an unordered collection of key/value pairs
  - in which all the keys are distinct, and
  - the value associated with a given key can be retrieved, updated, or removed
    - using a constant number of key comparisons on the average,
    - no matter how large the hash table.
  - Go does not provide a set type,
    - but since the keys of a map are distinct, a map can serve this purpose
- a map type is written `map[K]V`,
  - where `K` and `V` are the types of its keys and values.
- All of the keys in a given map are of the same type, and
  - all of the values are of the same type,

- but the keys need not be of the same type as the values.
- ⚠ The key type `K` must be comparable using `==`,
  - so that the map can test whether a given key is equal to one already within it.
  - There are no restrictions on the value type `V`.
- The built-in function `make` can be used to create a map
  - We can also use a map literal to create a new map populated with some initial key/value pairs
  - so an alternative expression for a new empty map is `map[string]int{}`
  - The zero value for a map type is `nil`,
    - that is, a reference to no hash table at all.
    - Most operations on maps, including lookup, delete, len, and range loops, are safe to perform on a `nil` map reference,
      - since it behaves like an empty map.
      - ⚠ But storing to a `nil` map causes a panic
        - You must allocate the map before you can store into it.
- Map elements
  - are accessed through the usual subscript notation
  - and removed with the built-in function `delete`
  - All of these operations are safe even if the element isn't in the map;
  - ⚠ a map lookup using a key that isn't present returns the zero value for its type
    - Accessing a map element by subscripting always yields a value.
    - If you need to know whether the element was really there or not use
      - `value, ok := ages[key]`
      - `if !ok { /* key is not a key in this map }`
  - ⚠ a map element is not a variable, and we cannot take its address
    - One reason that we can't take the address of a map element is that growing a map might cause rehashing of existing elements into new storage locations, thus potentially invalidating the address.
- To enumerate all the key/value pairs in the map, we use a range-based for loop
  - The order of map iteration is unspecified, and
  - different implementations might use a different hash function,
    - leading to a different ordering.
  - ⚠ In practice, the order is random,
    - varying from one execution to the next.
    - This is intentional
  - To enumerate the key/value pairs in order,
    - we must sort the keys explicitly,
      - for instance, using the `Strings` function from the `sort` package if the keys are strings.
- ⚠ As with slices, maps cannot be compared to each other;
  - the only legal comparison is with `nil`.
  - To test whether two maps contain the same keys and the same associated values, we must write a loop

- Sometimes we need a map or set whose keys are slices,
  - but because a map's keys must be comparable, this cannot be expressed directly.
  - However, it can be done in two steps.
    - First we define a helper function  $k$  that maps each key to a string, with the property that  $k(x) == k(y)$  if and only if we consider  $x$  and  $y$  equivalent.
    - Then we create a map whose keys are strings, applying the helper function to each key before we access the map.
  - And the type of  $k(x)$  needn't be a string; any comparable type with the desired equivalence property will do, such as integers, arrays, or structs.

## 4.4 Structs

- A struct is an *aggregate* data type
  - that groups together zero or more named values of arbitrary types as a single entity.
  - Each value is called a field.
  - All of these fields are collected into a single entity that can be
    - copied as a unit,
    - passed to functions and returned by them,
      - 💡 For efficiency, larger struct types are usually passed to or returned from functions indirectly using a pointer
    - stored in arrays, and so on.
  - Because structs are so commonly dealt with through pointers,
    - it's possible to use this shorthand notation to create and initialize a struct variable and obtain its address:
      - `pp := &Point{1, 2}`
- The zero value for a struct is composed of the zero values of each of its fields.
  - It is usually desirable that the zero value be a natural or sensible default.
    - For example, in `bytes.Buffer`, the initial value of the struct is a ready-to-use empty buffer, and
    - the zero value of `sync.Mutex` is a ready-to-use unlocked mutex.
  - Sometimes this sensible initial behavior happens for free,
    - but sometimes the type designer has to work at it.
- ⚠ Field order is significant to type identity
  - Changing the order of fields would result in defining a different struct type
- A named struct type  $S$  can't declare a field of the same type  $S$ :
  - an aggregate value cannot contain itself
  - But  $S$  may declare a field of the pointer type  $*S$ ,
    - which lets us create recursive data structures like linked lists and trees
- The individual fields are accessed using dot notation
  - The name of a struct field is exported if it begins with a capital letter

- The dot notation also works with a pointer to a struct
  - Without having to explicitly dereference the pointer before accessing the fields

## struct literal

- There are two forms of *struct literal*.
- The first form requires that a value be specified for every field, in the right order.
  - `Point{1, 2}`
  - It burdens the writer (and reader) with remembering exactly what the fields are, and
  - ⚠ it makes the code fragile should the set of fields later grow or be reordered.
  - Accordingly, this form tends to be used only within the package that defines the struct type,
    - or with smaller struct types for which there is an obvious field ordering convention, like `image.Point{x, y}` or `color.RGBA{red, green, blue, alpha}`.
- More often, the second form is used,
  - in which a struct value is initialized by listing some
  - or all of the field names and their corresponding values,
    - E.g. `anim := gif.GIF{LoopCount: nframes}`
  - If a field is omitted in this kind of literal, it is set to the zero value for its type.
  - Because names are provided, the order of fields doesn't matter.
- The two forms cannot be mixed in the same literal.
  - Nor can you use the (order-based) first form of literal to sneak around the rule that unexported identifiers may not be referred to from another package.

## Comparability

- If all the fields of a struct are comparable, the struct itself is comparable,
  - so two expressions of that type may be compared using `==` or `!=`.
  - The `==` operation compares the corresponding fields of the two structs in order
- 💡 Comparable struct types, like other comparable types, may be used as the key type of a map

## STRUCT EMBEDDING AND ANONYMOUS FIELDS

- Go's unusual struct embedding mechanism lets us use one named struct type as an anonymous field of another struct type,
  - providing a convenient syntactic shortcut so that a simple dot expression like `x.f` can stand for a chain of fields like `x.d.e.f`.
- Go lets us declare a field with a type but no name;
  - such fields are called anonymous fields.
  - The type of the field must be a named type
    - or a pointer to a named type.

- we can refer to the names at the leaves of the implicit tree without giving the intervening names
- Unfortunately, there's no corresponding shorthand for the struct literal syntax
  - The struct literal must follow the shape of the type declaration
- Because "anonymous" fields do have implicit names,
  - you can't have two anonymous fields of the same type since their names would conflict.

## 4.5 JSON

- The basic JSON types are
  - numbers (in decimal or scientific notation),
  - booleans (true or false), and
  - strings,
    - which are sequences of Unicode code points enclosed in double quotes,
    - with backslash escapes using a similar notation to Go
- These basic types may be combined recursively using JSON arrays and objects.
  - A JSON array is an ordered sequence of values,
    - written as a comma-separated list enclosed in square brackets;
    - JSON arrays are used to encode Go arrays and slices.
  - A JSON object is a mapping from strings to values,
    - written as a sequence of name:value pairs separated by commas and surrounded by braces;
    - JSON objects are used to encode Go maps (with string keys) and structs.
- Converting a Go data structure to JSON is called marshaling.
  - Marshaling is done by `json.Marshal`
  - ⚠ Only exported fields are marshaled, which is why we chose capitalized names for all the Go field names.
  - `Marshal` produces a byte slice containing a very long string with no extraneous white space
  - For human consumption, a variant called `json.MarshalIndent` produces neatly indented output.
- A field tag is a string of metadata associated at compile time with the field of a struct
  - A field tag may be any literal string,
    - but it is conventionally interpreted as a space-separated list of key:"value" pairs;
    - since they contain double quotation marks, field tags are usually written with raw string literals.
  - The `json` key controls the behavior of the `encoding/json` package, and other `encoding/...` packages follow this convention.
    - The first part of the `json` field tag specifies an alternative JSON name for the Go field.

- The second part of the field `omitempty` indicates that no JSON output should be produced if the field has the zero value for its type or is otherwise empty.
- The inverse operation to marshaling, decoding JSON and populating a Go data structure, is called unmarshaling,
  - and it is done by `json.Unmarshal`
  - the matching process that associates JSON names with Go struct names during unmarshaling is case-insensitive,
    - so it's only necessary to use a field tag when there's an underscore in the JSON name but not in the Go name

## 4.6 Text and HTML Templates

- sometimes formatting must be more elaborate
  - it's desirable to separate the format from the code more completely.
  - This can be done with the `text/template` and `html/template` packages,
    - which provide a mechanism for substituting the values of variables into a text or HTML template.
- A template is a string or file containing one or more portions enclosed in double braces, `{{...}}`, called actions.
  - Most of the string is printed literally,
  - but the actions trigger other behaviors.
  - Each action contains an expression in the template language,
    - a simple but powerful notation for printing values,
    - selecting struct fields,
    - calling functions and methods,
    - expressing control flow such as if-else statements and range loops, and
    - instantiating other templates.
- Within an action, there is a notion of the current value, referred to as “dot” and written as “.”, a period.
  - The dot initially refers to the template's parameter
- The `{{range ...}}` and `{{end}}` actions create a loop,
  - so the text between them is expanded multiple times, with dot bound to successive elements of `Items`.
- Within an action, the `|` notation makes the result of one operation the argument of another, analogous to a Unix shell pipeline.
  - Used to pipe values to functions to process values into a suitable format for output
- Producing output with a template is a two-step process.
  - First we must parse the template into a suitable internal representation, and
  - then execute it on specific inputs.
  - Parsing need be done only once.
- Because templates are usually fixed at compile time,



- failure to parse a template indicates a fatal bug in the program.
- The `template.Must` helper function makes error handling more convenient:
  - it accepts a template and an error,
  - checks that the error is nil (and panics otherwise), and
  - then returns the template.
- the `html/template` package uses the same API and expression language as `text/template`
  - but adds features for automatic and context-appropriate escaping of strings appearing within HTML, JavaScript, CSS, or URLs.
  - These features can help avoid a perennial security problem of HTML generation, an injection attack

## 5. Functions

- The parameter list specifies the names and types of the function's parameters,
  - which are the local variables whose values or arguments are supplied by the caller.
- Like parameters, results may be named.
  - In that case, each name declares a local variable initialized to the zero value for its type.
  - Well-chosen names can document the significance of a function's results.
  - Names are particularly valuable when a function returns multiple results of the same type
    - but it's not always necessary to name multiple results solely for documentation.
    - For instance, convention dictates that a final `bool` result indicates success; an error result often needs no explanation.
  - In a function with named results, the operands of a return statement may be omitted.
    - This is called a bare return.
    - In functions with many return statements and several results,
      - bare returns can reduce code duplication,
      - but they rarely make code easier to understand.
    - For this reason, bare returns are best used sparingly.
- Parameters are local variables within the body of the function,
  - with their initial values set to the arguments supplied by the caller.
  - Function parameters and named results are variables in the same lexical block as the function's outermost local variables.
- a sequence of parameters or results of the same type can be factored
  - so that the type itself is written only once
- The type of a function is sometimes called its signature.
  - Two functions have the same type or signature
    - if they have the same sequence of parameter types and

- the same sequence of result types.
  - The names of parameters and results don't affect the type,
    - nor does whether or not they were declared using the factored form.
- Every function call must provide an argument for each parameter,
  - in the order in which the parameters were declared.
  - ⚠ Go has no concept of default parameter values,
    - nor any way to specify arguments by name,
    - so the names of parameters and results don't matter to the caller
      - except as documentation.
- Arguments are passed by *value*,
  - so the function receives a copy of each argument;
    - modifications to the copy do not affect the caller.
  - ⚠ However, if the argument contains some kind of reference,
    - like a pointer, slice, map, function, or channel,
    - then the caller may be affected by any modifications the function makes to variables indirectly referred to by the argument.
- You may occasionally encounter a function declaration without a body,
  - indicating that the function is implemented in another language (e.g. asm).
  - Such a declaration defines the function signature.

## 5.4 Errors

- A function for which failure is an expected behavior returns an additional result,
  - conventionally the last one.
  - If the failure has only one possible cause, the result is a boolean,
    - usually called ok
- The built-in type error is an interface type.
  - a non-nil error has an error message string
    - which we can obtain by calling its Error method
      - or print by calling `fmt.Println(err)`
- Go's approach sets it apart from many other languages in which failures are reported using exceptions, not ordinary values.
  - Although Go does have an exception mechanism of sorts
    - it is used only for reporting truly unexpected errors that indicate a bug,
    - not the routine errors that a robust program should be built to expect.
- The `fmt.Errorf` function formats an error message using `fmt.Sprintf` and returns a new error value.
  - We use it to build descriptive errors by successively prefixing additional context information to the original error message.
  - When the error is ultimately handled by the program's main function,
    - it should provide a clear causal chain from the root problem to the overall failure,
    - reminiscent of a NASA accident investigation, e.g.:

- genesis: crashed: no parachute: G-switch failed: bad relay orientation
  - ⚠ Because error messages are frequently chained together, message strings should not be capitalized and newlines should be avoided.
- When designing error messages,
  - be deliberate,
    - so that each one is a meaningful description of the problem with sufficient and relevant detail, and
  - be consistent,
    - so that errors returned by the same function or by a group of functions in the same package are similar in form and
    - can be dealt with in the same way.
- Get into the habit of handling errors after every function call, and
  - when you deliberately ignore one, document your intention clearly.
- Error handling in Go has a particular rhythm.
  - After checking an error, failure is usually dealt with before success.
  - Functions tend to exhibit a common structure,
    - with a series of initial checks to reject errors,
    - followed by the substance of the function at the end, minimally indented.
- On occasion, however, a program must take different actions depending on the kind of error that has occurred.
  - Such specific conditions can be reported using distinguished errors
    - E.g. `var EOF = errors.New("EOF")` // in package io

## 5.5 Function Values

- Functions are first-class values in Go:
  - like other values, function values have types, and
  - they may be assigned to variables or passed to or returned from functions.
  - A function value may be called like any other function.
  - The zero value of a function type is nil.
    - Calling a nil function value causes a panic
- Functions are reference types
  - Function values may be compared with nil
  - but they are not comparable,
    - so they may not be compared against each other
    - or used as keys in a map.

## 5.6 Anonymous Functions

- Named functions can be declared only at the package level,
  - but we can use a function literal to denote a function value within any expression.
- A function literal is written like a function declaration,

- but without a name following the func keyword.
  - It is an expression, and its value is called an anonymous function.
- Function literals let us define a function at its point of use.
- ⚠ More importantly, functions defined in this way have access to the entire lexical environment,
  - so the inner function can refer to variables from the enclosing function
- Closures: function values are not just code but can have state.
  - The anonymous inner function can access and update the local variables of the enclosing function.
    - ⚠ Care must be taken when using local variables originate from loops
      - Because the enclosing function
        - Might get just the last value of the loop variable
        - if the call to the anon function is deferred to after the loop
      - Workaround is to create a local variable inside the loop
        - That references the loop variable
        - And use that in the anon function
  - These hidden variable references are why we classify functions as reference types
    - and why function values are not comparable.
  - Function values like these are implemented using a technique called closures,
    - and Go programmers often use this term for function values.
  - Here again we see an example where the lifetime of a variable is not determined by its scope
- Recursion: When an anonymous function requires recursion
  - we must first declare a variable,
  - and then assign the anonymous function to that variable.

## 5.7 Variadic Functions

- A variadic function is one that can be called with varying numbers of arguments.
  - The most familiar examples are `fmt.Printf` and its variants.
- To declare a variadic function,
  - the type of the final parameter is preceded by an ellipsis, "...",
  - which indicates that the function may be called with any number of arguments of this type.
- to invoke a variadic function when the arguments are already in a slice:
  - place an ellipsis after the final argument.
- Although the ... parameter behaves like a slice within the function body,
  - the type of a variadic function is distinct from the type of a function with an ordinary slice parameter.

## 5.8 Deferred Function Calls

- a defer statement is an ordinary function or method call prefixed by the keyword defer.
  - $\triangle$  The function and argument expressions are evaluated when the statement is executed,
  - but the actual call is deferred until the function that contains the defer statement has finished,
    - whether normally,
      - by executing a return statement or falling off the end, or
    - abnormally, by panicking.
  - Any number of calls may be deferred;
    - they are executed in the reverse of the order in which they were deferred.
- A defer statement is often used with paired operations
  - like open and close, connect and disconnect, or lock and unlock
  - to ensure that resources are released in all cases,
    - no matter how complex the control flow.
  - The right place for a defer statement that releases a resource is immediately after the resource has been successfully acquired
- The defer statement can also be used to pair “on entry” and “on exit” actions when debugging a complex function.
  - By deferring a call to the returned function in this way,
    - we can instrument the entry point and all exit points of a function in a single statement
    - and even pass values, like the start time, between the two actions.
- Deferred functions run after return statements have updated the function’s result variables.
  - Because an anonymous function can access its enclosing function’s variables, including named results,
  - a deferred anonymous function can observe the function’s results.
  - A deferred anonymous function can even change the values that the enclosing function returns to its caller
- If deferred function returns a error
  - We should check and return the error

## 5.9 Panic

- Go’s type system catches many mistakes at compile time,
- but others, like an out-of-bounds array access or nil pointer dereference, require checks at run time.
- When the Go runtime detects these mistakes, it panics.
- During a typical panic,
  - normal execution stops,

- all deferred function calls in that goroutine are executed,
  - all deferred functions are run in reverse order,
  - starting with those of the topmost function on the stack
    - and proceeding up to main
- and the program crashes with a log message.
  - This log message includes the panic value,
  - which is usually an error message of some sort,
  - and, for each goroutine, a stack trace showing the stack of function calls that were active at the time of the panic.
- Not all panics come from the runtime.
  - The built-in panic function may be called directly;
    - it accepts any value as an argument.
  - A panic is often the best thing to do when some “impossible” situation happens,
    - for instance, execution reaches a case that logically can’t happen:
- Although Go’s panic mechanism resembles exceptions in other languages,
  - the situations in which panic is used are quite different.
  - Since a panic causes the program to crash,
    - it is generally used for grave errors,
      - such as a logical inconsistency in the program;
  - diligent programmers consider any crash to be proof of a bug in their code.
  - In a robust program, “expected” errors,
    - the kind that arise from incorrect input, misconfiguration, or failing I/O,
    - should be handled gracefully;
    - they are best dealt with using error values.

## 5.10 Recover

- Giving up is usually the right response to a panic, but not always.
  - It might be possible to recover in some way,
  - or at least clean up the mess before quitting.
    - For example, a web server that encounters an unexpected problem could close the connection rather than leave the client hanging,
    - and during development, it might report the error to the client too.
- If the built-in recover function is called within a deferred function
  - and the function containing the defer statement is panicking,
  - recover ends the current state of panic and returns the panic value.
  - The function that was panicking does not continue where it left off
    - but returns normally.
    - If recover is called at any other time, it has no effect and returns nil.
- Recovering indiscriminately from panics is a dubious practice
  - because the state of a package’s variables after a panic is rarely well defined or documented.

- Perhaps a critical update to a data structure was incomplete, a file or network connection was opened but not closed, or a lock was acquired but not released.
- Furthermore, by replacing a crash with, say, a line in a log file, indiscriminate recovery may cause bugs to go unnoticed.
- Recovering from a panic within the same package
  - can help simplify the handling of complex or unexpected errors,
  - but as a general rule, you should not attempt to recover from another package's panic.
  - Public APIs should report failures as errors.
  - Similarly, you should not recover from a panic that may pass through a function you do not maintain,
    - such as a caller-provided callback,
    - since you cannot reason about its safety.

## 6. Methods (OOP)

- an object is simply a value or variable that has methods,
  - and a method is a function associated with a particular type.
- An object-oriented program is one that uses methods to express the properties and operations of each data structure
  - so that clients need not access the object's representation directly.

### 6.1 Method Declarations

- A method is declared with a variant of the ordinary function declaration in which an extra parameter appears before the function name.
  - The parameter attaches the function to the type of that parameter.
  - The extra parameter `p` is called the method's receiver,
    - a legacy from early object-oriented languages that described calling a method as "sending a message to an object."
- In Go, we don't use a special name like `this` or `self` for the receiver;
  - we choose receiver names just as we would for any other parameter.
  - Since the receiver name will be frequently used,
    - it's a good idea to choose something short and
    - to be consistent across methods.
- Selectors are also used to select fields of struct types, as in `p.X`.
  - Since methods and fields inhabit the same namespace,
    - declaring a method `X` on the struct type `Point` would be ambiguous and the compiler will reject it.
- It is often convenient to define additional behaviors for simple types
  - such as numbers, strings, slices, maps, and sometimes even functions.
  - Methods may be declared on any named type defined in the same package,

- so long as its underlying type is neither a pointer nor an interface.
- In allowing methods to be associated with any type, Go is unlike many other object-oriented languages.

## 6.2 Methods with a Pointer Receiver

- Because calling a function makes a copy of each argument value,
  - if a function needs to update a variable,
  - or if an argument is so large that we wish to avoid copying it,
  - we must pass the address of the variable using a pointer.
  - The same goes for methods that need to update the receiver variable:
    - we attach them to the pointer type
- If the receiver `p` is a variable of type `T` but the method requires a `*T` receiver,
  - we can use this shorthand: `p.foo()`
  - and the compiler will perform an implicit `&p` on the variable.
  - This works only for variables,
    - including struct fields like `p.X` and array or slice elements.
    - We cannot call a `*T` method on a non-addressable `T` receiver,
      - because there's no way to obtain the address of a temporary value.
      - `T{1, 2}.foo()` // compile error: can't take address of literal
  - Or the receiver argument has type `*T` and the receiver parameter has type `T`.
    - The compiler implicitly dereferences the receiver,
      - in other words, loads the value
- 💡 convention dictates that if any method of a type has a pointer receiver,
  - then all methods of the type should have a pointer receiver,
  - even ones that don't strictly need it.
- If all the methods of a named type `T` have a receiver type of `T` itself (not `*T`),
  - it is safe to copy instances of that type;
  - calling any of its methods necessarily makes a copy.
  - ⚠ But if any method has a pointer receiver,
    - you should avoid copying instances of `T`
      - because doing so may violate internal invariants.
- Named types (`T`) and pointers to them (`*T`) are the only types that may appear in a receiver declaration.
- ⚠ Furthermore, to avoid ambiguities, method declarations are not permitted on named types that are themselves pointer types



## Receiver Type

If in doubt, use a pointer, but there are times when a value receiver makes sense, usually for reasons of efficiency, such as for small unchanging structs or values of basic type. Some useful guidelines:

- If the receiver is a map, func or chan, don't use a pointer to them. If the receiver is a slice and the method doesn't reslice or reallocate the slice, don't use a pointer to it.
- If the method needs to mutate the receiver, the receiver must be a pointer.
- If the receiver is a struct that contains a `sync.Mutex` or similar synchronizing field, the receiver must be a pointer to avoid copying.
- If the receiver is a large struct or array, a pointer receiver is more efficient. How large is large? Assume it's equivalent to passing all its elements as arguments to the method. If that feels too large, it's also too large for the receiver.
- Can function or methods, either concurrently or when called from this method, be mutating the receiver? A value type creates a copy of the receiver when the method is invoked, so outside updates will not be applied to this receiver. If changes must be visible in the original receiver, the receiver must be a pointer.
- If the receiver is a struct, array or slice and any of its elements is a pointer to something that might be mutating, prefer a pointer receiver, as it will make the intention more clear to the reader.
- If the receiver is a small array or struct that is naturally a value type (for instance, something like the `time.Time` type), with no mutable fields and no pointers, or is just a simple basic type such as `int` or `string`, a value receiver makes sense. A value receiver can reduce the amount of garbage that can be generated; if a value is passed to a value method, an on-stack copy can be used instead of allocating on the heap. (The compiler tries to be smart about avoiding this allocation, but it can't always succeed.) Don't choose a value receiver type for this reason without profiling first.
- Finally, when in doubt, use a pointer receiver.

## NIL IS A VALID RECEIVER VALUE

- Just as some functions allow nil pointers as arguments,
  - so do some methods for their receiver,
  - especially if nil is a meaningful zero value of the type, as with maps and slices
- 💡 When you define a type whose methods allow nil as a receiver value,
  - it's worth pointing this out explicitly in its documentation comment

## 6.3 Composing Types by Struct Embedding

- embedding lets us take a syntactic shortcut to defining a `ColoredPoint` that contains all the fields of `Point`, plus some more.

- `type Point struct{ X, Y float64 }`
- `type ColoredPoint struct {`
  - `Point`
  - `Color color.RGBA``}`
- If we want, we can select the fields of `ColoredPoint` that were contributed by the embedded `Point` without mentioning `Point`
- A similar mechanism applies to the methods of `Point`.
  - We can call methods of the embedded `Point` field using a receiver of type `ColoredPoint`,
  - even though `ColoredPoint` has no declared methods
  - In this way, embedding allows complex types with many methods to be built up by the composition of several fields, each providing a few methods.
- Readers familiar with class-based object-oriented languages may be tempted to view `Point` as a base class and `ColoredPoint` as a subclass or derived class,
  - or to interpret the relationship between these types as if a `ColoredPoint` “is a” `Point`.
  - But that would be a mistake.
  - `Distance` has a parameter of type `Point`, and `q` is not a `Point`, so although `q` does have an embedded field of that type, we must explicitly select it. Attempting to pass `q` would be an error:
- A `ColoredPoint` is not a `Point`,
  - but it “has a” `Point`,
  - and it has two additional methods `Distance` and `ScaleBy` promoted from `Point`.
  - If you prefer to think in terms of implementation,
    - the embedded field instructs the compiler to generate additional wrapper methods that delegate to the declared methods
- The type of an anonymous field may be a pointer to a named type,
  - in which case fields and methods are promoted indirectly from the pointed-to object.
  - 💡 Adding another level of indirection lets us share common structures and vary the relationships between objects dynamically
- A struct type may have more than one anonymous field.
  - `type ColoredPoint struct {`
    - `Point`
    - `color.RGBA``}`
  - then a value of this type would have all the methods of `Point`,
    - all the methods of `RGBA`,
    - and any additional methods declared on `ColoredPoint` directly.
  - When the compiler resolves a selector such as `p.ScaleBy` to a method,
    - it first looks for a directly declared method named `ScaleBy`,

- then for methods promoted once from ColoredPoint's embedded fields,
  - then for methods promoted twice from embedded fields within Point and RGBA, and so on.
  - The compiler reports an error if the selector was ambiguous because two methods were promoted from the same rank.
- Methods can be declared only on
  - named types (like Point) and
  - pointers to them (\*Point),
  - but thanks to embedding, it's possible and sometimes useful for unnamed struct types to have methods too.


## 6.4 Method Values and Expressions

- Usually we select and call a method in the same expression, as in `p.Distance()`,
  - but it's possible to separate these two operations.
  - The selector `p.Distance` yields a method value,
    - a function that binds a method (`Point.Distance`) to a specific receiver value `p`.
  - This function can then be invoked without a receiver value;
    - it needs only the non-receiver arguments.
- Method values are useful when a package's API calls for a function value,
  - and the client's desired behavior for that function is to call a method on a specific receiver.
- Related to the method value is the method expression.
  - When calling a method, as opposed to an ordinary function,
    - we must supply the receiver in a special way using the selector syntax.
  - A method expression, written `T.f` or `(*T).f` where `T` is a type,
    - yields a function value with a regular first parameter taking the place of the receiver, so it can be called in the usual way.
- Method expressions can be helpful
  - when you need a value to represent a choice among several methods belonging to the same type
  - so that you can call the chosen method with many different receivers.

## 6.6 Encapsulation

- A variable or method of an object is said to be encapsulated if it is inaccessible to clients of the object.
  - Encapsulation, sometimes called information hiding, is a key aspect of object-oriented programming.
- Go has only one mechanism to control the visibility of names:
  - capitalized identifiers are exported from the package in which they are defined,
  - and uncapitalized names are not.

- The same mechanism that limits access to members of a package also limits access to the fields of a struct or the methods of a type.
  - As a consequence, to encapsulate an object, we must make it a struct.
- That's the reason the `IntSet` type from the previous section was declared as a struct type
  - even though it has only a single field
    - ```
type IntSet struct {
    words []uint64
}
```
  - We could instead define `IntSet` as a slice type as follows
    - ```
type IntSet []uint64
```
  - Although this version of `IntSet` would be essentially equivalent,
    - ⚠ it would allow clients from other packages to read and modify the slice directly.
- Another consequence of this name-based mechanism is that the unit of encapsulation is the package,
  - not the type as in many other languages.
  - The fields of a struct type are visible to all code within the same package.
  - Whether the code appears in a function or a method makes no difference.
- Encapsulation provides three benefits.
  - First, because clients cannot directly modify the object's variables,
    - one need inspect fewer statements to understand the possible values of those variables.
  - Second, hiding implementation details prevents clients from depending on things that might change,
    - which gives the designer greater freedom to evolve the implementation without breaking API compatibility.
    - once exported, a field cannot be unexported without an incompatible change to the API,
      - so the initial choice should be deliberate and should consider the complexity of the invariants that must be maintained,
      - the likelihood of future changes, and
      - the quantity of client code that would be affected by a change.
  - The third benefit of encapsulation, and in many cases the most important, is that it prevents clients from setting an object's variables arbitrarily.
    - Because the object's variables can be set only by functions in the same package,
    - the author of that package can ensure that all those functions maintain the object's internal invariants.
- Functions that merely access or modify internal values of a type,
  - such as the methods of the `Logger` type from `log` package,
    - are called getters and setters.
  - However, when naming a getter method, we usually omit the `Get` prefix.
    - This preference for brevity extends to all methods,

- not just field accessors,
  - and to other redundant prefixes as well, such as Fetch, Find, and Lookup.
-  Encapsulation is not always desirable.
  - E.g. By revealing its representation as an int64 number of nanoseconds,
  - time.Duration lets us use all the usual arithmetic and comparison operations with durations,
    - and even to define constants of this type
  - As another example, contrast IntSet with the geometry.Path type from the beginning of this chapter.
    - Path was defined as a slice type, allowing its clients to construct instances using the slice literal syntax,
    - to iterate over its points using a range loop, and so on,
    - whereas these operations are denied to clients of IntSet.
  - Here's the crucial difference:
    - geometry.Path is intrinsically a sequence of points, no more and no less,
      - and we don't foresee adding new fields to it,
      - so it makes sense for the geometry package to reveal that Path is a slice.
    - In contrast, an IntSet merely happens to be represented as a []uint64 slice.
      - It could have been represented using []uint, or something completely different for sets that are sparse or very small,
      - and it might perhaps benefit from additional features like an extra field to record the number of elements in the set.
      - For these reasons, it makes sense for IntSet to be opaque.

## 7. Interfaces

- Interface types express generalizations or abstractions about the behaviors of other types.
  - By generalizing, interfaces let us write functions that are more flexible and adaptable
    - because they are not tied to the details of one particular implementation.
- Many object-oriented languages have some notion of interfaces,
  - but what makes Go's interfaces so distinctive is that they are satisfied implicitly.
  - In other words, there's no need to declare all the interfaces that a given concrete type satisfies;
  - simply possessing the necessary methods is enough.
  - This design lets you create new interfaces that are satisfied by existing concrete types without changing the existing types,
    - which is particularly useful for types defined in packages that you don't control.

## 7.1 Interfaces as Contracts

- All the types we've looked at so far have been concrete types.
  - A concrete type specifies the exact representation of its values and exposes the intrinsic operations of that representation, such as arithmetic for numbers, or indexing, append, and range for slices.
  - A concrete type may also provide additional behaviors through its methods.
  - When you have a value of a concrete type, you know exactly what it is and what you can do with it.
- There is another kind of type in Go called an interface type.
  - An interface is an abstract type.
  - It doesn't expose the representation or internal structure of its values, or the set of basic operations they support;
  - it reveals only some of their methods.
  - When you have a value of an interface type, you know nothing about what it is;
  - *you know only what it can do*,
    - or more precisely, what behaviors are provided by its methods.
  - An interface type specifies a set of methods that a concrete type must possess to be considered an instance of that interface.
  - Similar to struct embedding, interfaces can be composed of other interfaces
- This freedom to substitute one type for another
  - that satisfies the same interface is called substitutability,
  - and is a hallmark of object-oriented programming.

## 7.3 Interface Satisfaction

- A type satisfies an interface if it possesses all the methods the interface requires.
- As a shorthand, Go programmers often say that a concrete type "is a" particular interface type, meaning that it satisfies the interface.
- The assignability rule for interfaces is very simple:
  - an expression may be assigned to an interface only if its type satisfies the interface.
  - ```
var w io.Writer
w = os.Stdout      // OK: *os.File has Write method
w = time.Second    // compile error: time.Duration lacks
                    Write method
```
  - This rule applies even when the right-hand side is itself an interface
- ⚠ one subtlety in what it means for a type to have a method.
  - Recall for each named concrete type T,
    - some of its methods have a receiver of type T itself
    - whereas others require a \*T pointer.
  - Recall also that it is legal to call a \*T method on an argument of type T

- so long as the argument is a variable;
  - the compiler implicitly takes its address.
  - But this is mere syntactic sugar:
- ⚠ a value of type T does not possess all the methods that a \*T pointer does,
  - and as a result it might satisfy fewer interfaces.
- ```
type IntSet struct { /* ... */ }
func (*IntSet) String() string

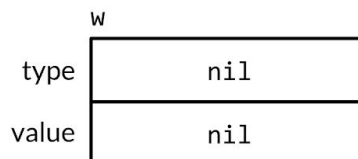
var _ = IntSet{}.String()
// compile error: String requires *IntSet receiver

var s IntSet
var _ = s.String()
// OK: s is a variable and &s has a String method
```
- However, since only \*IntSet has a String method, only \*IntSet satisfies the `fmt.Stringer` interface
  - ```
var _ fmt.Stringer = &s // OK
var _ fmt.Stringer = s // compile error: IntSet
lacks String method
```
- 💡 the `godoc -analysis=type` tool displays the methods of each type
  - and the relationship between interfaces and concrete types.
  - `-analysis=type` and `-analysis=pointer` flags augment the documentation and the source code with the results of advanced static analysis.
- Like an envelope that wraps and conceals the letter it holds,
  - an interface wraps and conceals the concrete type and value that it holds.
  - ⚠ Only the methods revealed by the interface type may be called,
    - even if the concrete type has others
    - ```
var w io.Writer
w = os.Stdout
w.Write([]byte("hello")) // OK: io.Writer has Write
method
w.Close() // compile error: io.Writer
lacks Close method
```
- Because the empty interface type places no demands on the types that satisfy it,
  - we can assign any value to the empty interface `interface{}`.
  - it is what allows functions like `fmt.Println`, or `errorf` to accept arguments of any type.
  - Of course, having created an `interface{}` value containing a boolean, float, string, map, pointer, or any other type,
    - we can do nothing directly to the value it holds since the interface has no methods.
    - We need a way to get the value back out again.
      - using a type assertion

- Since interface satisfaction depends only on the methods of the two types involved,
  - there is no need to declare the relationship between a concrete type and the interfaces it satisfies.
  - That said, it is occasionally useful to document and assert the relationship at compile time when it is intended but not otherwise enforced by the program.
    - ```
// *bytes.Buffer must satisfy io.Writer
var w io.Writer = new(bytes.Buffer)
// Or more efficiently
var _ io.Writer = (*bytes.Buffer)(nil)
```
    - We needn't allocate a new variable since any value of type \*bytes.Buffer will do, even nil, which we write as (\*bytes.Buffer)(nil) using an explicit conversion
- 
- Because duration-valued flags are so useful, this feature is built into the flag package,
  - but it's easy to define new flag notations for our own data types.
  - We need only define a type that satisfies the flag.Value interface

## 7.5 Interface Values

- Conceptually, a value of an interface type, or interface value, has two components,
  - a concrete type and
  - a value of that type.
  - These are called the interface's dynamic type and dynamic value.
- For a statically typed language like Go, types are a compile-time concept,
  - so a type is not a value.
  - In our conceptual model, a set of values called type descriptors provide information about each type,
    - such as its name and methods.
  - In an interface value, the type component is represented by the appropriate type descriptor.
- In Go, variables are always initialized to a well-defined value, and interfaces are no exception.
  - The zero value for an interface has both its type and value components set to nil
  - ```
var w io.Writer
```



- An interface value is described as nil or non-nil based on its dynamic type,
  - so this is a nil interface value.
  - You can test whether an interface value is nil using `w == nil` or `w != nil`.



- Calling any method of a nil interface value causes a panic
  - `w.Write([]byte("hello"))` // panic: nil pointer dereference
- `w = os.Stdout`
  - assigns a value of type `*os.File` to `w`
  - This assignment involves an implicit conversion from a concrete type to an interface type,
    - and is equivalent to the explicit conversion `io.Writer(os.Stdout)`.
  - A conversion of this kind, whether explicit or implicit,
    - captures the type and the value of its operand.
  - The interface value's dynamic type is set to the type descriptor for the pointer type `*os.File`,
    - and its dynamic value holds a copy of `os.Stdout`,
    - which is a pointer to the `os.File` variable representing the standard output of the process



- Calling the `Write` method on an interface value containing an `*os.File` pointer causes the `(*os.File).Write` method to be called.
  - The call prints "hello"
- In general, we cannot know at compile time what the dynamic type of an interface value will be,
  - so a call through an interface must use dynamic dispatch.
  - Instead of a direct call, the compiler must generate code to obtain the address of the method named `Write` from the type descriptor,
  - then make an indirect call to that address.
  - The receiver argument for the call is a copy of the interface's dynamic value, `os.Stdout`.
- The third statement assigns a value of type `*bytes.Buffer` to the interface value
  - `w = new(bytes.Buffer)`
  - The dynamic type is now `*bytes.Buffer` and the dynamic value is a pointer to the newly allocated buffer



- A call to the `Write` method uses the same mechanism as before
  - This time, the type descriptor is `*bytes.Buffer`,
    - so the `(*bytes.Buffer).Write` method is called,

- with the address of the buffer as the value of the receiver parameter.
  - The call appends "hello" to the buffer
- Interface values may be compared using == and !=.
  - Two interface values are equal if both are nil,
    - or if their dynamic types are identical and their dynamic values are equal according to the usual behavior of == for that type.
  - Because interface values are comparable, they may be used as the keys of a map or as the operand of a switch statement.
  - ⚠ However, if two interface values are compared and have the same dynamic type,
    - but that type is not comparable (a slice, for instance),
    - then the comparison fails with a panic
  - ⚠ In this respect, interface types are unusual.
    - Other types are either safely comparable (like basic types and pointers)
      - or not comparable at all (like slices, maps, and functions),
    - but when comparing interface values or aggregate types that contain interface values,
      - we must be aware of the potential for a panic.
    - A similar risk exists when using interfaces as map keys or switch operands.
    - Only compare interface values if you are certain that they contain dynamic values of comparable types.
- ⚠ A nil interface value, which contains no value at all,
  - is not the same as an interface value containing a pointer that happens to be nil.
  - This subtle distinction creates a trap into which every Go programmer has stumbled.

## 7.10 Type Assertions

- A type assertion is an operation applied to an interface value.
- Syntactically, it looks like `x.(T)`,
  - where x is an expression of an interface type and
  - T is a type, called the “asserted” type.
- A type assertion checks that the dynamic type of its operand matches the asserted type.
- There are two possibilities.
  - First, if the asserted type T is a concrete type,
    - then the type assertion checks whether x’s dynamic type is identical to T.
    - If this check succeeds, the result of the type assertion is x’s dynamic value, whose type is of course T.
    - In other words, a type assertion to a concrete type extracts the concrete value from its operand.
    - If the check fails, then the operation panics.

- `var w io.Writer`  
`w = os.Stdout`  
`f := w.(*os.File) // success: f == os.Stdout`  
`c := w.(*bytes.Buffer) // panic: interface holds`  
`*os.File, not *bytes.Buffer`
- Second, if instead the asserted type T is an interface type,
  - then the type assertion checks whether x's dynamic type satisfies T.
  - If this check succeeds, the dynamic value is not extracted;
  - the result is still an interface value with the same type and value components,
    - but the result has the interface type T.
  - In other words, a type assertion to an interface type
    - changes the type of the expression,
    - making a different (and usually larger) set of methods accessible,
    - but it preserves the dynamic type and value components inside the interface value.
- `var w io.Writer`  
`w = os.Stdout`  
`rw := w.(io.ReadWriter) // success: *os.File has both`  
`Read and Write`
  - `w = new(ByteCounter)`  
`rw = w.(io.ReadWriter) // panic: *ByteCounter has no`  
`Read method`
- No matter what type was asserted,
  - if the operand is a nil interface value, the type assertion fails.
  - A type assertion to a less restrictive interface type (one with fewer methods) is rarely needed,
    - as it behaves just like an assignment, except in the nil case.
- 💡 If the type assertion appears in an assignment in which two results are expected,
  - the operation does not panic on failure
  - but instead returns an additional second result, a boolean indicating success

## 7.13 Type Switches

- Interfaces are used in two distinct styles.
  - In the first style,
    - exemplified by `io.Reader`, `io.Writer`, `fmt.Stringer`, `sort.Interface`, `http.Handler`, and `error`,
    - an interface's methods express the similarities of the concrete types that satisfy the interface
    - but hide the representation details and intrinsic operations of those concrete types.

- The *emphasis is on the methods*, not on the concrete types.
- The second style exploits the ability of an interface value to hold values of a variety of concrete types
  - and considers the interface to be the union of those types.
  - Type assertions are used to discriminate among these types dynamically and treat each case differently.
  - In this style, *the emphasis is on the concrete types* that satisfy the interface,
    - not on the interface's methods (if indeed it has any),
    - and there is no hiding of information.
  - We'll describe interfaces used this way as discriminated unions.
- If you're familiar with object-oriented programming, you may recognize these two styles as
  - subtype polymorphism and
  - ad hoc polymorphism
- In its simplest form, a type switch looks like an ordinary switch statement
  - in which the operand is `x.(type)`
    - that's literally the keyword type
    - and each case has one or more types.
  - A type switch enables a multi-way branch
    - based on the interface value's dynamic type.
  - The nil case matches if `x == nil`,
  - and the default case matches if no other case does.
- Case order becomes significant when one or more case types are interfaces,
  - since then there is a possibility of two cases matching.
- cases needs access to the value extracted by the type assertion.
  - Since this is typical, the type switch statement has an extended form
  - that binds the extracted value to a new variable within each case:
    - `switch x := x.(type) { /* ... */ }`
      - Here we've called the new variables x too;
      - as with type assertions, reuse of variable names is common.
      - Like a switch statement, a type switch implicitly creates a lexical block,
      - so the declaration of the new variable called x does not conflict with a variable x in an outer block.
      - Each case also implicitly creates a separate lexical block.
- `switch x := x.(type) {`
  - `case nil:`
  - `..`
  - `case int, uint:`
  - `..`
  - `case bool:`
  - `..`

```

case string:
    ..
default:
    panic(fmt.Sprintf("unexpected type %T: %v", x, x))
}

```

- In this version, within the block of each single-type case, the variable `x` has the same type as the case.
- For instance, `x` has type `bool` within the boolean case and `string` within the string case.
- In all other cases, `x` has the (interface) type of the switch operand, which is `interface{}` in this example.
- When the same action is required for multiple cases, like `int` and `uint`, the type switch makes it easy to combine them.
- the function runs to completion only if the argument's type matches one of the cases in the type switch;
  - otherwise it panics with an "unexpected type" message.
- Although the type of `x` is `interface{}`, we consider it a *discriminated union* of `int`, `uint`, `bool`, `string`, and `nil`.
- The purpose of a traditional interface like `io.Reader`
  - is to hide details of the concrete types that satisfy it
  - so that new implementations can be created;
  - each concrete type is treated uniformly.
- By contrast, the set of concrete types that satisfy a discriminated union
  - is fixed by the design and exposed, not hidden.
  - Discriminated union types have few methods;
  - functions that operate on them are expressed as a set of cases
    - using a type switch, with different logic in each case.

## 7.15 A Few Words of Advice 💡

- When designing a new package,
  - novice Go programmers often start by creating a set of interfaces and
  - only later define the concrete types that satisfy them.
  - This approach results in many interfaces,
    - each of which has only a single implementation.
  - Don't do that.
  - Such interfaces are unnecessary abstractions; they also have a run-time cost.
  - You can restrict which methods of a type or fields of a struct are visible outside a package using the export mechanism
  - 💡 Interfaces are only needed when there are two or more concrete types that must be dealt with in a uniform way.
- We make an exception to this rule
  - when an interface is satisfied by a single concrete type

- but that type cannot live in the same package as the interface
    - because of its dependencies.
  - In that case, an interface is a good way to decouple two packages.
- Because interfaces are used in Go only when they are satisfied by two or more types,
  - they necessarily abstract away from the details of any particular implementation.
  - The result is smaller interfaces with fewer, simpler methods,
    - often just one as with `io.Writer` or `fmt.Stringer`.
  - Small interfaces are easier to satisfy when new types come along.
  - 💡 *A good rule of thumb for interface design is ask only for what you need.*
- Go has great support for the object-oriented style of programming,
  - but this does not mean you need to use it exclusively.
  - Not everything need be an object;
    - standalone functions have their place,
    - as do unencapsulated data types.

## 8. Goroutines and Channels

- Go enables two styles of concurrent programming.
  - goroutines and channels,
    - which support communicating sequential processes or CSP,
    - a model of concurrency in which values are passed between independent activities (goroutines)
    - but variables are for the most part confined to a single activity.
  - more traditional model of shared memory multithreading

### 8.1 Goroutines

- In Go, each concurrently executing activity is called a goroutine.
- When a program starts, its only goroutine is the one that calls the main function,
  - so we call it the main goroutine.
- New goroutines are created by the `go` statement.
- Syntactically, a `go` statement is an ordinary function or method call prefixed by the keyword `go`.
- A `go` statement causes the function to be called in a newly created goroutine.
  - The `go` statement itself completes immediately
- When main function returns,
  - all goroutines are abruptly terminated and the program exits.
- Other than by returning from main or exiting the program,
  - there is no programmatic way for one goroutine to stop another,
  - but there are ways to communicate with a goroutine to request that it stop itself.
- The Go runtime contains its own scheduler that uses a technique known as m:n scheduling, because it multiplexes (or schedules) m goroutines on n OS threads.

- The job of the Go scheduler is analogous to that of the kernel scheduler, but it is concerned only with the goroutines of a single Go program.
- Unlike the operating system's thread scheduler, the Go scheduler is not invoked periodically by a hardware timer, but implicitly by certain Go language constructs.
  - For example, when a goroutine calls `time.Sleep` or blocks in a channel or mutex operation, the scheduler puts it to sleep and runs another goroutine until it is time to wake the first one up.
  - Because it doesn't need a switch to kernel context, rescheduling a goroutine is much cheaper than rescheduling a thread.
- The Go scheduler uses a parameter called `GOMAXPROCS` to determine how many OS threads may be actively executing Go code simultaneously.
  - Its default value is the number of CPUs on the machine
  - Goroutines that are sleeping or blocked in a communication do not need a thread at all.
    - Goroutines that are blocked in I/O or other system calls or are calling non-Go functions, do need an OS thread, but `GOMAXPROCS` need not account for them.
  - You can explicitly control this parameter using the `GOMAXPROCS` environment variable or the `runtime.GOMAXPROCS` function.

## GOROUTINES HAVE NO IDENTITY

- In most operating systems and programming languages that support multithreading, the current thread has a distinct identity that can be easily obtained as an ordinary value,
  - typically an integer or pointer.
- This makes it easy to build an abstraction called thread-local storage,
  - which is essentially a global map keyed by thread identity,
  - so that each thread can store and retrieve values independent of other threads.
- Goroutines have no notion of identity that is accessible to the programmer.
  - This is by design, since thread-local storage tends to be abused.
- Go encourages a simpler style of programming in which parameters that affect the behavior of a function are explicit.
  - Not only does this make programs easier to read,
  - but it lets us freely assign subtasks of a given function to many different goroutines without worrying about their identity.

## 8.4 Channels

- If goroutines are the activities of a concurrent Go program,
  - channels are the connections between them.
- A channel is a communication mechanism
  - that lets one goroutine send values to another goroutine.
- Each channel is a conduit for values of a particular type,

- called the channel's element type.
  - E.g. The type of a channel whose elements have type `int` is written `chan int`.
- To create a channel, we use the built-in `make` function
  - `ch := make(chan int) // ch has type 'chan int'`
- As with maps, a channel is a *reference* to the data structure created by `make`.
  - When we copy a channel or pass one as an argument to a function,
  - we are copying a reference,
  - so caller and callee refer to the same data structure.
  - As with other reference types, the zero value of a channel is `nil`.
- Two channels of the same type may be compared using `==`.
  - The comparison is true if both are references to the same channel data structure.
  - A channel may also be compared to `nil`.
- A channel has two principal operations,
  - send and receive, collectively known as communications.
  - A send statement transmits a value from one goroutine,
    - through the channel, to another goroutine executing a corresponding receive expression.
  - Both operations are written using the `<-` operator.
  - In a send statement, the `<-` separates the channel and value operands.
  - In a receive expression, `<-` precedes the channel operand.
  - A receive expression whose result is not used is a valid statement.
  - `ch <- x // a send statement`

```
x = <-ch // a receive expression in an assignment
statement
```

```
<-ch // a receive statement; result is discarded
```

- Channels support a third operation, `close`,
  - which sets a flag indicating that no more values will ever be sent on this channel;
  - subsequent attempts to send will panic.
  - Receive operations on a closed channel yield the values that have been sent until no more values are left;
  - any receive operations thereafter complete immediately
    - and yield the zero value of the channel's element type.
    - There is no way to test directly whether a channel has been closed,
      - but there is a variant of the receive operation that produces two results: the received channel element, plus a boolean value,
      - conventionally called `ok`, which is true for a successful receive and false for a receive on a closed and drained channel.
  - Because the syntax above is clumsy and this pattern is common,
    - the language lets us use a range loop to iterate over channels too.
    - This is a more convenient syntax for receiving all the values sent on a channel and terminating the loop after the last one.
  - To close a channel, we call the built-in `close` function



- `close(ch)`
  - You needn't close every channel when you've finished with it.
    - It's only necessary to close a channel when it is important to tell the receiving goroutines that all data have been sent.
    - A channel that the garbage collector determines to be unreachable will have its resources reclaimed whether or not it is closed.
      - Don't confuse this with the close operation for open files.
      - It is important to call the Close method on every file when you've finished with it.
  - Attempting to close an already-closed channel causes a panic,
    - as does closing a nil channel.
- A channel created with a simple call to `make` is called an unbuffered channel,
  - but `make` accepts an optional second argument,
  - an integer called the channel's capacity.
  - If the capacity is non-zero, `make` creates a buffered channel.

### 8.4.1 UNBUFFERED CHANNELS

- A send operation on an unbuffered channel
  - blocks the sending goroutine until another goroutine executes a corresponding receive on the same channel,
    - at which point the value is transmitted and both goroutines may continue.
  - Conversely, if the receive operation was attempted first,
    - the receiving goroutine is blocked until another goroutine performs a send on the same channel.
- Communication over an unbuffered channel
  - causes the sending and receiving goroutines to synchronize.
  - Because of this, unbuffered channels are sometimes called synchronous channels.
  - When a value is sent on an unbuffered channel,
    - the receipt of the value happens before the reawakening of the sending goroutine.
- Messages sent over channels have two important aspects.
  - Each message has a value,
  - but sometimes the fact of communication and the moment at which it occurs are just as important.
  - We call messages events when we wish to stress this aspect.
  - When the event carries no additional information,
    - that is, its sole purpose is synchronization,
    - we'll emphasize this by using a channel whose element type is `struct{}`,
    - though it's common to use a channel of `bool` or `int` for the same purpose
      - since `done <- 1` is shorter than `done <- struct{}`.

## 8.4.2 PIPELINES

- Channels can be used to connect goroutines together
  - so that the output of one is the input to another.
  - This is called a pipeline.

## 8.4.3 UNIDIRECTIONAL CHANNEL TYPES

- the Go type system provides unidirectional channel types
  - that expose only one or the other of the send and receive operations.
  - The type `chan<- int`, a send-only channel of `int`, allows sends but not receives.
  - Conversely, the type `<-chan int`, a receive-only channel of `int`, allows receives but not sends.
  - The position of the `<-` arrow relative to the `chan` keyword is a mnemonic.
  - Violations of this discipline are detected at compile time.
- Since the `close` operation asserts that no more sends will occur on a channel,
  - only the sending goroutine is in a position to call it,
  - and for this reason it is a compile-time error to attempt to close a receive-only channel.
- Conversions from bidirectional to unidirectional channel types are permitted in any assignment.
  - They are done implicitly in function calls
    - when the function has unidirectional channels declared as args
    - And a bidirectional channel is passed to it
  - There is no going back, however:
    - once you have a value of a unidirectional type such as `chan<- int`,
    - there is no way to obtain from it a value of type `chan int` that refers to the same channel data structure.

## 8.4.4 BUFFERED CHANNELS

- A buffered channel has a queue of elements.
- The queue's maximum size is determined when it is created,
  - by the capacity argument to `make`.
  - `ch = make(chan string, 3) // creates a buffered channel capable of holding three string values`
  - In the unlikely event that a program needs to know the channel's buffer capacity, it can be obtained by calling the built-in `cap` function
- A send operation on a buffered channel inserts an element at the back of the queue,
- and a receive operation removes an element from the front.
- If the channel is full, the send operation blocks its goroutine until space is made available by another goroutine's receive.

- Conversely, if the channel is empty, a receive operation blocks until a value is sent by another goroutine.
  - We can send up to  $n$  values on a channel without the goroutine blocking
    - Where  $n$  is the capacity of the channel
  - In this way, the channel's buffer decouples the sending and receiving goroutines.
- When applied to a channel, the built-in `len` function returns the number of elements currently buffered.
  - Since in a concurrent program this information is likely to be stale as soon as it is retrieved, its value is limited,
  - but it could conceivably be useful during fault diagnosis or performance optimization.
- Novices are sometimes tempted to use buffered channels within a single goroutine as a queue,
  - lured by their pleasingly simple syntax,
  - but this is a mistake.
  - Channels are deeply connected to goroutine scheduling,
    - and without another goroutine receiving from the channel,
    - a sender—and perhaps the whole program—risks becoming blocked forever.
  - If all you need is a simple queue, make one using a slice.

## Buffered or Unbuffered

- in an unbuffered channel,
  - the slower goroutines sometimes could get stuck trying to send their responses on a channel from which no goroutine will ever receive.
  - This situation, called a goroutine leak, would be a bug.
  - Unlike garbage variables, leaked goroutines are not automatically collected,
    - so it is important to make sure that goroutines terminate themselves when no longer needed.
- The choice between unbuffered and buffered channels,
  - and the choice of a buffered channel's capacity,
  - may both affect the correctness of a program.
- Unbuffered channels give stronger synchronization guarantees
  - because every send operation is synchronized with its corresponding receive;
- with buffered channels, these operations are decoupled.
- Also, when we know an upper bound on the number of values that will be sent on a channel,
  - it's not unusual to create a buffered channel of that size
  - and perform all the sends before the first value is received.
  - Failure to allocate sufficient buffer capacity would cause the program to deadlock.
- Channel buffering may also affect program performance.

- The assembly line metaphor is a useful one for channels and goroutines.
  - So long as the workers work at about the same rate on average,
    - most of these handovers proceed quickly,
    - smoothing out transient differences in their respective rates.
  - More space between workers—larger buffers—
    - can smooth out bigger transient variations in their rates without stalling the assembly line,
    - such as happens when one worker takes a short break, then later rushes to catch up.
  - On the other hand, if an earlier stage of the assembly line is consistently faster than the following stage,
    - the buffer between them will spend most of its time full.
    - Conversely, if the later stage is faster, the buffer will usually be empty.
    - A buffer provides no benefit in this case.
  - For example, if the second stage is more elaborate,
    - a single worker may not be able to keep up with the supply from the first worker or meet the demand from the third.
    - To solve the problem, we could hire another worker to help the second, performing the same task but working independently.
    - This is analogous to creating another goroutine communicating over the same channels.

## 8.5 Looping in Parallel

- Problems that consist entirely of subproblems that are completely independent of each other are described as embarrassingly parallel.
- Embarrassingly parallel problems are the easiest kind to implement concurrently and enjoy performance that scales linearly with the amount of parallelism.
- when we cannot predict the number of loop iterations.
  - To know when the last goroutine has finished (which may not be the last one to start),
  - we need to increment a counter before each goroutine starts and decrement it as each goroutine finishes.
  - This demands a special kind of counter,
    - one that can be safely manipulated from multiple goroutines
    - and that provides a way to wait until it becomes zero.
    - This counter type is known as `sync.WaitGroup`
      - `Add // increments counter`
      - `Done // decrements counter`
      - `Wait // Waits till counter reaches 0`
  - Add increments the counter,
    - must be called before the worker goroutine starts,
    - not within it;

- otherwise we would not be sure that the Add happens before the” goroutine calls Wait.
  - Also, Add takes a parameter,
    - but Done does not;
      - it’s equivalent to Add(-1).
    - We use defer with Done to ensure that the counter is decremented even in the error case.
  - The structure of the code above is a common and idiomatic pattern for looping in parallel when we don’t know the number of iterations.
- We can limit parallelism using a buffered channel of capacity n to model a concurrency primitive called a counting semaphore.
  - Conceptually, each of the n vacant slots in the channel buffer represents a token entitling the holder to proceed.
  - Sending a value into the channel acquires a token, and receiving a value from the channel releases a token, creating a new vacant slot.
  - This ensures that at most n sends can occur without an intervening receive.
  - Although it might be more intuitive to treat filled slots in the channel buffer as tokens, using vacant slots avoids the need to fill the channel buffer after creating it.
  - Since the channel element type is not important, we’ll use struct{}, which has size zero.

## 8.7 Multiplexing with select

- When we need to receive from multiple channels in a goroutine
  - We can’t just receive from each channel
    - because whichever operation we try first will block until completion.
  - We need to multiplex these operations,
    - and to do that, we need a select statement.
  - ```
select {
    case <-ch1:
        // ...
    case x := <-ch2:
        // ...use x...
    case ch3 <- y:
        // ...
    default:
        // ...
}
```
- Like a switch statement, it has a number of cases and an optional default.
- Each case specifies a communication (a send or receive operation on some channel)
  - and an associated block of statements.
  - A receive expression may appear on its own, as in the first case,

- or within a short variable declaration, as in the second case;
  - the second form lets you refer to the received value.
- A select waits until a communication for some case is ready to proceed.
  - It then performs that communication and executes the case's associated statements;
  - the other communications do not happen.
  - A select with no cases, `select{}`, waits forever.
- If multiple cases are ready, select picks one at random,
  - which ensures that every channel has an equal chance of being selected.
- Sometimes we want to try to send or receive on a channel but avoid blocking if the channel is not ready—a non-blocking communication.
  - A select statement can do that too.
  - A select may have a default, which specifies what to do when none of the other communications can proceed immediately.
- The zero value for a channel is nil.
  - Perhaps surprisingly, nil channels are sometimes useful.
  - Because send and receive operations on a nil channel block forever,
    - a case in a select statement whose channel is nil is never selected.
  - This lets us use nil to enable or disable cases that correspond to features like handling timeouts or cancellation, responding to other input events, or emitting output.

## 8.9 Cancellation

- Sometimes we need to instruct a goroutine to stop what it is doing,
  - for example, in a web server performing a computation on behalf of a client that has disconnected.
- There is no way for one goroutine to terminate another directly,
  - since that would leave all its shared variables in undefined states.
- In general, it's hard to know how many goroutines are working on our behalf at any given moment.
- For cancellation, what we need is a reliable mechanism
  - to broadcast an event over a channel
  - so that many goroutines can see it as it occurs
  - and can later see that it has occurred.
- after a channel has been closed and drained of all sent values, subsequent receive operations proceed immediately, yielding zero values.
  - We can exploit this to create a broadcast mechanism:
    - don't send a value on the channel, close it.
  - we create a cancellation channel on which no values are ever sent,
    - but whose closure indicates that it is time for the program to stop what it is doing.

- We also define a utility function, `cancelled`, that checks or polls the cancellation state at the instant it is called.
- `var done = make(chan struct{})`

```
func cancelled() bool {
    select {
    case <-done:
        return true
    default:
        return false
    }
}
```

- Debugging : when main returns, a program exits, it can be hard to tell a main function that cleans up after itself from one that does not.
  - There's a handy trick we can use during testing:
  - if instead of returning from main in the event of cancellation,
    - we execute a call to `panic`, then the runtime will dump the stack of every goroutine in the program.
  - If the main goroutine is the only one left, then it has cleaned up after itself.
  - But if other goroutines remain, they may not have been properly cancelled, or perhaps they have been cancelled but the cancellation takes time;
    - a little investigation may be worthwhile.
  - The panic dump often contains sufficient information to distinguish these cases.

## 9. Concurrency with Shared Variables

### 9.1 Race Conditions

- concurrency-safe types are the exception rather than the rule,
  - so you should access a variable concurrently only if the documentation for its type says that this is safe.
- We avoid concurrent access to most variables either by
  - confining them to a single goroutine or
  - by maintaining a higher-level invariant of mutual exclusion.
- exported package-level functions are generally expected to be concurrency-safe.
  - Since package-level variables cannot be confined to a single goroutine,
  - functions that modify them must enforce mutual exclusion.
- There are many reasons a function might not work when called concurrently,
  - including deadlock, livelock, and resource starvation.
- A race condition is a situation in which the program does not give the correct result for some interleavings of the operations of multiple goroutines.

- Race conditions are pernicious because they may remain latent in a program and appear infrequently,
  - perhaps only under heavy load or when using certain compilers, platforms, or architectures.
- This makes them hard to reproduce and diagnose.
- A data race occurs whenever two goroutines access the same variable concurrently
  - and at least one of the accesses is a write.
- there are three ways to avoid a data race.
  - The first way is not to write the variable.
    - Data structures that are never modified or are immutable are inherently concurrency-safe and need no synchronization.
    - But obviously we can't use this approach if updates are essential
  - The second way to avoid a data race is to avoid accessing the variable from multiple goroutines.
    - These variables are confined to a single goroutine.
    - Since other goroutines cannot access the variable directly,
      - they must use a channel to send the confining goroutine a request to query or update the variable.
    - This is what is meant by the Go mantra "Do not communicate by sharing memory; instead, share memory by communicating."
    - A goroutine that brokers access to a confined variable using channel requests is called a monitor goroutine for that variable.
    - Even when a variable cannot be confined to a single goroutine for its entire lifetime, confinement may still be a solution to the problem of concurrent access.
      - For example, it's common to share a variable between goroutines in a pipeline by passing its address from one stage to the next over a channel.
      - If each stage of the pipeline refrains from accessing the variable after sending it to the next stage, then all accesses to the variable are sequential.
      - In effect, the variable is confined to one stage of the pipeline, then confined to the next, and so on.
      - This discipline is sometimes called serial confinement.
  - The third way to avoid a data race is to allow many goroutines to access the variable, but only one at a time.
    - This approach is known as mutual exclusion

## 9.2 Mutual Exclusion: sync.Mutex

- Its Lock method acquires the token (called a lock) and
- its Unlock method releases it



- It is essential that the goroutine release the lock once it is finished, on all paths through the function, including error paths.
- by deferring a call to `Unlock`, the critical section implicitly extends to the end of the current function,
  - freeing us from having to remember to insert `Unlock` calls in one or more places far from the call to `Lock`
- The mutex guards the shared variables.
- By convention, the variables guarded by a mutex are declared immediately after the declaration of the mutex itself.
  - If you deviate from this, be sure to document it.
- When you use a mutex, make sure that both it and the variables it guards are not exported,
  - whether they are package-level variables or the fields of a struct.

### 9.3 Read/Write Mutexes: `sync.RWMutex`

- This lock is called a multiple readers, single writer lock,
  - and in Go it's provided by `sync.RWMutex`
- allows read-only operations to proceed in parallel with each other,
  - but write operations to have fully exclusive access
- `RLock` can be used only if there are no writes to shared variables in the critical section.
- It's only profitable to use an `RWMutex` when most of the goroutines that acquire the lock are readers,
  - and the lock is under contention, that is, goroutines routinely have to wait to acquire it.
  - An `RWMutex` requires more complex internal bookkeeping, making it slower than a regular mutex for uncontended locks.
  - If in doubt, use an exclusive `Lock`.

### 9.4 Memory Synchronization

- In a modern computer there may be dozens of processors, each with its own local cache of the main memory.
- For efficiency, writes to memory are buffered within each processor and flushed out to main memory only when necessary.
- They may even be committed to main memory in a different order than they were written by the writing goroutine.
- Synchronization primitives like channel communications and mutex operations cause the processor to flush out and commit all its accumulated writes
  - so that the effects of goroutine execution up to that point are guaranteed to be visible to goroutines running on other processors.
- It is tempting to try to understand concurrency as if it corresponds to some interleaving of the statements of each goroutine,

- but this is not how a modern compiler or CPU works.
- All these concurrency problems can be avoided by the consistent use of simple, established patterns.
  - Where possible, confine variables to a single goroutine;
  - for all other variables, use mutual exclusion.

## 9.5 Lazy Initialization: `sync.Once`

- the `sync` package provides a specialized solution to the problem of one-time initialization: `sync.Once`.
- Conceptually, a `Once` consists of a mutex and a boolean variable that records whether initialization has taken place;
  - the mutex guards both the boolean and the client's data structures.
  - The sole method, `Do`, accepts the initialization function as its argument.
- Each call to `Do()` locks the mutex and checks the boolean variable.
  - In the first call, in which the variable is false, `Do` calls the function and sets the variable to true.
  - Subsequent calls do nothing, but the mutex synchronization ensures that the effects of `Do` on memory become visible to all goroutines.
- Using `sync.Once` in this way, we can avoid sharing variables with other goroutines until they have been properly constructed.

## 9.6 The Race Detector

- the Go runtime and toolchain are equipped with a sophisticated and easy-to-use dynamic analysis tool, the race detector.
- Just add the `-race` flag to your `go build`, `go run`, or `go test` command.
- This causes the compiler to build a modified version of your application or test with additional instrumentation
  - that effectively records all accesses to shared variables that occurred during execution,
  - along with the identity of the goroutine that read or wrote the variable.
  - In addition, the modified program records all synchronization events, such as `go` statements, channel operations, and calls to `(*sync.Mutex).Lock`, `(*sync.WaitGroup).Wait`, and so on.
- The race detector studies this stream of events,
  - looking for cases in which one goroutine reads or writes a shared variable that was most recently written by a different goroutine without an intervening synchronization operation.
  - This indicates a concurrent access to the shared variable, and thus a data race.
  - The tool prints a report that includes the identity of the variable, and the stacks of active function calls in the reading goroutine and the writing goroutine.
  - This is usually sufficient to pinpoint the problem.

- The race detector reports all data races that were actually executed.
  - However, it can only detect race conditions that occur during a run;
  - it cannot prove that none will ever occur.
  - For best results, make sure that your tests exercise your packages using concurrency.
- Due to extra bookkeeping, a program built with race detection needs more time and memory to run,
  - but the overhead is tolerable even for many production jobs.
  - For infrequently occurring race conditions, letting the race detector do its job can save hours or days of debugging.