An in-depth look at Slices and Maps

Session 07

Golang course by Exadel

24 Oct 2022

Sergio Kovtunenko
Lead backend developer, Exadel

Agenda

- ▶️ Questions from the past session
- ▶️ State of collections in other languages
- ▶️ Maps in Go
- ▶️ Slices in Go
- ▶️ Next time...

Questions from the past session

Explain signed/unsigned integer type differences

✓ List of all available primitive types:

```
bool        // values are: 'true' or 'false'
string

// Numeric types:

uint        // either 32 or 64 bits depends on host platform
int         // same size as uint
uintptr     // an unsigned integer large enough to store the uninterpreted bits of a pointer value
uint8       // the set of all unsigned  8-bit integers (0 to 255)
uint16      // the set of all unsigned 16-bit integers (0 to 65535)
uint32      // the set of all unsigned 32-bit integers (0 to 4294967295)
uint64      // the set of all unsigned 64-bit integers (0 to 18446744073709551615)

int8        // the set of all signed  8-bit integers (-128 to 127)
int16       // the set of all signed 16-bit integers (-32768 to 32767)
int32       // the set of all signed 32-bit integers (-2147483648 to 2147483647)
int64       // the set of all signed 64-bit integers (-9223372036854775808 to 9223372036854775807)

float32     // the set of all IEEE-754 32-bit floating-point numbers
float64     // the set of all IEEE-754 64-bit floating-point numbers

complex64   // the set of all complex numbers with float32 real and imaginary parts
complex128  // the set of all complex numbers with float64 real and imaginary parts

byte        // alias for uint8
rune        // alias for int32 (represents a Unicode code point)
```

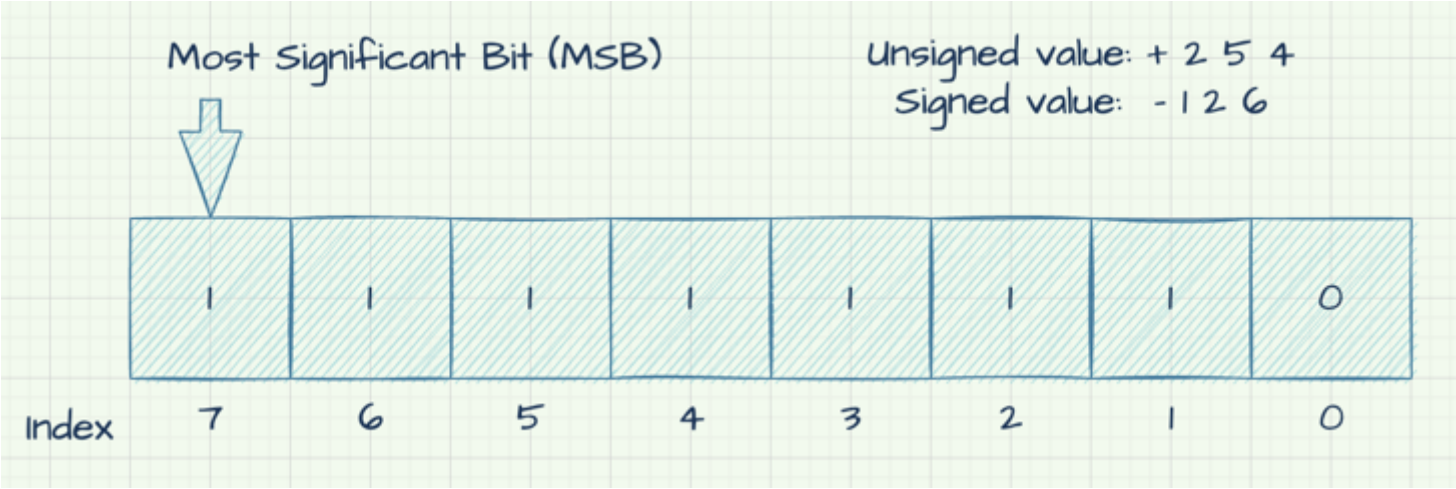
Binary representation (not only in Go)

```go
package main

import "fmt"

func main() {
    var signed int8 = -126
    fmt.Printf( format: "Decimal Value: %+d ; Binary form: %b\n", signed, signed)
    // output:
    // Decimal Value: -126 ; Binary form: -1111110

    // the Most Significant Bit (MSB) == 1 in value 0b11111110
    // if the number is positive, this has a value of 0, but if the number is negative, the bit is 1.
    var unsigned uint8 = 0b11111110
    fmt.Printf( format: "Decimal Value: %+d ; Binary form: %b\n", unsigned, unsigned)
    // output:
    // Decimal Value: +254 ; Binary form: 11111110
}
```

▶️ Check example at: `code/binary_representation_test.go`

▶️ To check: "proposal: Go 2: use unsigned integers for all lengths" (https://github.com/golang/go/issues/27460)

More info about integers



▶️ **Signed integers** are integers that can hold **positive and negative** numbers.

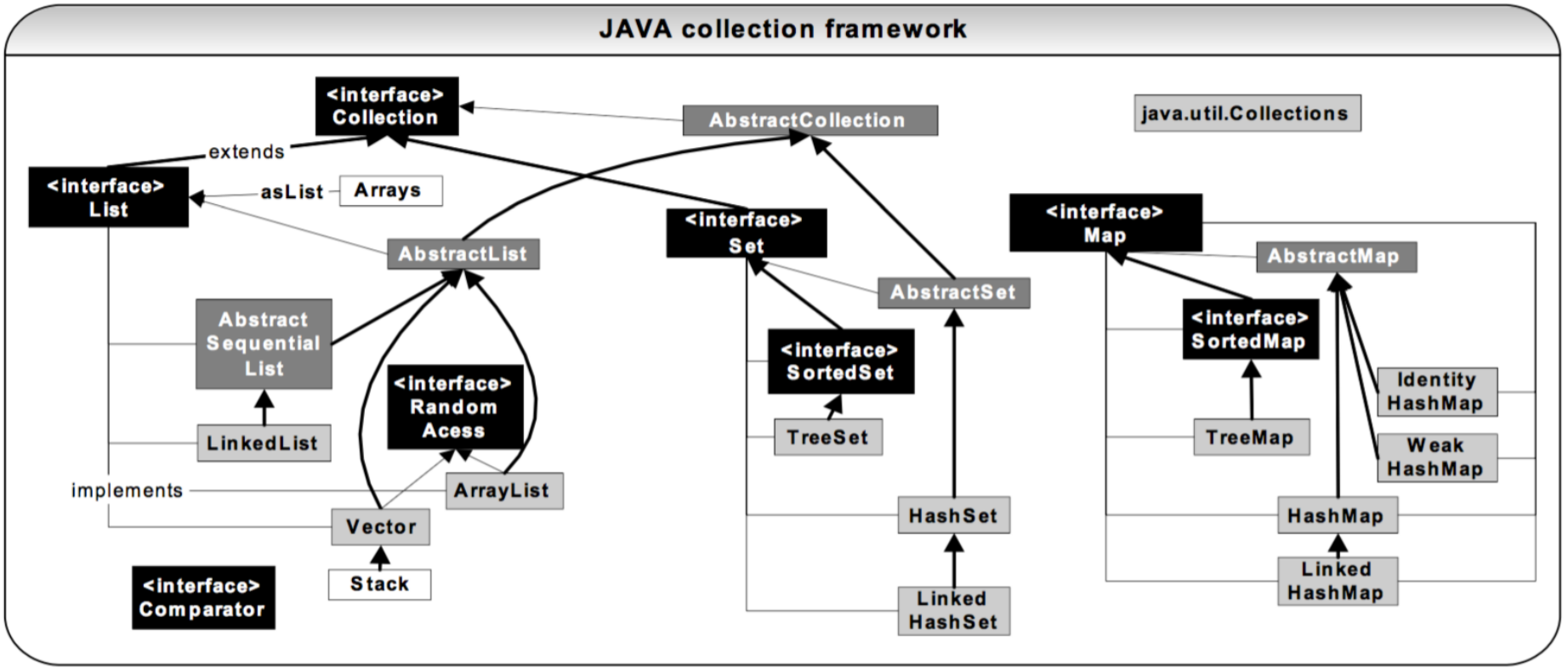
- for `int` on 32-bit CPU architecture it would be: from -2147483648 to 2147483647

▶️ **Unsigned integers** are integers that can only hold **non-negative** whole numbers.

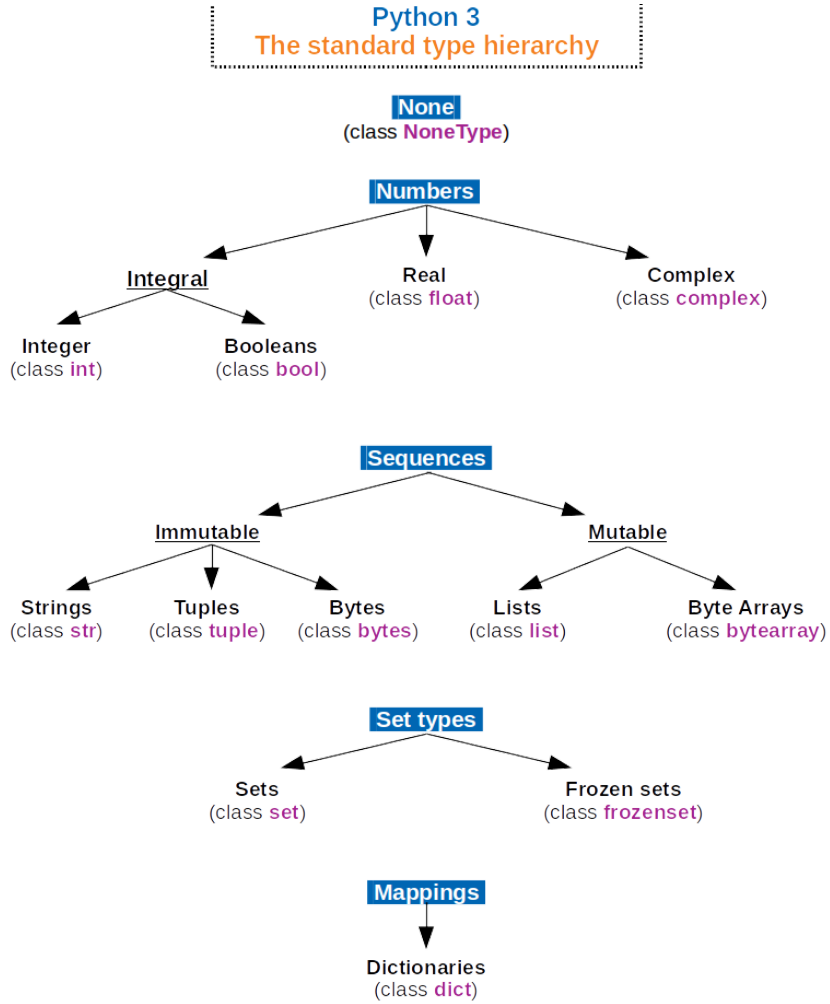
- for `uint` on 32-bit CPU architecture it would be: from 0 to 4294967295
- Consider the subtraction of two unsigned numbers, such as 3 and 5. 3 minus 5 is -2, but -2 can't be represented as an unsigned number.

State of collections in other languages

In Java (as of version 1.4)



In Python



Source: "Data type" by Wikipedia (https://en.wikipedia.org/wiki/Data_type)

Maps in Go

General information about Maps (1/2)

- ✓ Map is a language-supplied hash table.

```
map[KeyType]ValueType
```

- ✓ To create new map type we can use:
  - map literals;
  - **make()** function (as with slices).
    - The *initial capacity* does not bound its size:
      - maps grow to accommodate the number of items stored in them, with the exception of **nil** maps.
    - A **nil** map is equivalent to an empty map except that no elements may be added.
      - Might be used as "read-only" empty map (*safe to iterate over and retrieve values*).
    - Example:

```
make(map[string]int)
make(map[string]int, 100)
```

General information about Maps (2/2)

- ✓ **Keys** of map should support '*equality* ==' operator.
  - If <something> does not support 'equality' it can not be used as a key.
    - Slices, maps, and functions **can not** be used as a keys!
  - In short, comparable types are:
    - boolean;
    - numeric;
    - string;
    - pointer;
    - channel;
    - interface types;
    - structs or arrays that contain only those types.
- ✓ Map's value type may be any type at all, including another map.
- ✓ Uninitialised value **var m map[string]int** is **nil.**
- ✓ **Assigning** a map copies the map reference, **not** the ~~map contents~~.

Maps under the hood

▶️ As per Ian Taylor:

- "In the very early days what we call maps now were written as pointers, so you wrote `*map[int]int`. We moved away from that when we realized that no one ever wrote `map` without writing `*map`." (c) Ian Taylor
- More at: "email thread" (https://groups.google.com/g/golang-nuts/c/SjuhSYDITm4/m/jnrp7rRxDQAJ)

```
115  // A header for a Go map.
116  type hmap struct {
117      // Note: the format of the hmap is also encoded in cmd/compile/internal/reflectdata/reflect.go.
118      // Make sure this stays in sync with the compiler's definition.
119      count     int // # live cells == size of map.  Must be first (used by len() builtin)
120      flags     uint8
121      B         uint8  // log_2 of # of buckets (can hold up to loadFactor * 2^B items)
122      noverflow uint16 // approximate number of overflow buckets; see incrnoverflow for details
123      hash0     uint32 // hash seed
124
125      buckets    unsafe.Pointer // array of 2^B Buckets. may be nil if count==0.
126      oldbuckets unsafe.Pointer // previous bucket array of half the size, non-nil only when growing
127      nevacuate  uintptr        // progress counter for evacuation (buckets less than this have been evacuated)
128
129      extra *mapextra // optional fields
130  }
```

Map iteration order

✓ **Map iteration order is not specified:**

```
for key, value := range m {
    // order of key sequence different each time
}
```

- If you require a stable iteration order you must maintain a separate data structure that specifies that order.
  - This example uses a separate sorted slice of keys to print a *map[int]string* in key order:

```
package main
import (
    "fmt"
    "sort"
)

func main() {
    var m map[int]string
    var keys []int
    for k := range m {
        keys = append(keys, k)
    }
    sort.Ints(keys)
    for _, k := range keys {
        fmt.Println("Key:", k, "Value:", m[k])
    }
}
```

Map literals

✓ Map literals are *like struct literals*, but the keys are required:

```
type Vertex struct {
    Lat, Long float64
}

var m = map[string]Vertex {
    "Bell Labs": Vertex {
        40.68433, -74.39967,
    },
    "Google": Vertex {
        37.42202, -122.08408,
    },
}
```

✓ If the top-level type is just a type name, you can omit it from the elements of the literal:

```
type Vertex struct {
    Lat, Long float64
}

var m = map[string]Vertex {
    "Bell Labs": {40.68433, -74.39967},
    "Google":    {37.42202, -122.08408},
}
```

✓ Same syntax can be used to initialise an *empty map*.

- This functionally identical to using the **make()** function: `m := map[string]int{}`

Mutating maps

- ✓ **Insert** or **update** an element in map m: `m[key] = elem`
- ✓ **Retrieve** an element.
  - If the requested key doesn't exist, we get the value type's zero value: `elem := m[key]`
- ✓ **Delete** an element.
  - The **delete()** function doesn't return anything, and will do nothing if:
    - the specified key doesn't exist;
    - the map is `nil`.
  - Example: `delete(m, key)`
- ✓ **Test that a key is present** with a two-value assignment:
  - If key is in m, ok is true. If not, ok is false.
  - If key is not in the map, then elem is the zero value for the map's element type.
  - Example: `elem, ok := m[key]`
- ✓ **Check for number of items** inside map: `n := len(m)`

Common maps operations

✓ Common `map` operations:

```
make(map[K]V)
len(m)
m[k]
delete(m, k)
```

✓ Maps are not safe for concurrent use!

- One common way to protect maps is with **sync.RWMutex.**
  - This statement declares a 'counter' variable that is an anonymous struct containing a map and an embedded sync.RWMutex:

```
var counter = struct {
    sync.RWMutex
    m map[string]int
}{m: make(map[string]int)}
```

  - To read from the counter, take the read lock:

```
counter.RLock()
n := counter.m["some_key"]
counter.RUnlock()
fmt.Println("some_key:", n)
```

  - To write to the counter, take the write lock:

```
counter.Lock()
counter.m["some_key"]++
counter.Unlock()
```

Slices in Go

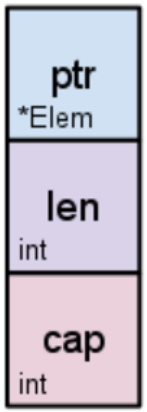
Slices general info

- ✓ A slice is a descriptor for a contiguous segment of an *underlying array* and *provides access* to a numbered sequence of elements **from that array**.
  - The elements can be addressed by *integer* indices **0** through **len(s)-1**.
  - A slice, once initialized, is always associated with an underlying array that holds its elements.
    - A slice therefore shares storage with its *array* and with *other slices of the same array*.
    - By contrast, *distinct arrays always represent distinct storage*.
- ✓ **Assigning** a slice copies the descriptor, **NOT** the underlying array!
- ✓ Slices may **grow** and **shrink**.
- ✓ The **zero value** of a slice is **nil**.
  - A **nil** slice has a *length* and *capacity* of **0**.

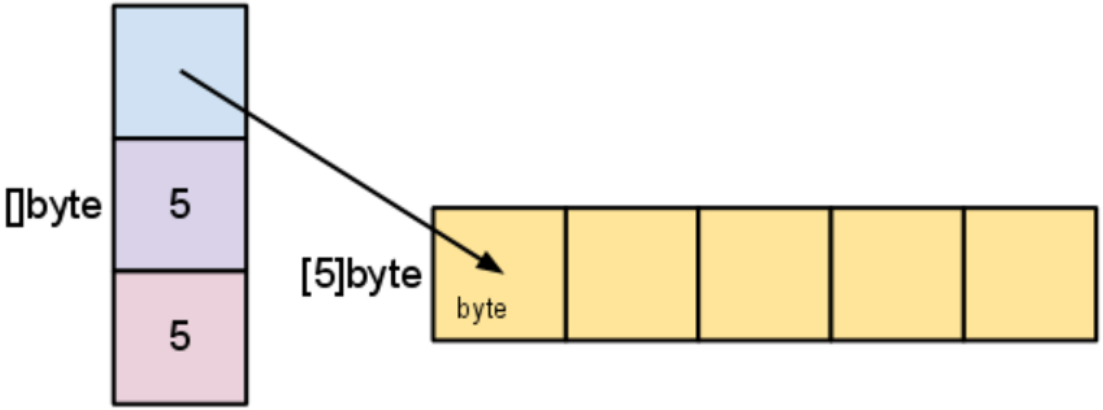
Slices internals

✓ Slices has ***length*** and ***capacity***.

- Internally, slices looks like this:



- For slice created by `make([]byte, 5)` is structured like this:



Slices under the hood

▶️ Check the `runtime.slice` struct definition.

```
15  type slice struct {
16      array unsafe.Pointer
17      len   int
18      cap   int
19  }
```

Slices creation

- ✓ Slices are created with the **make()** function.
  - It works by allocating a zeroed array and returning a slice that refers to that array.
    - A slice created with **make()** always allocates a *new, hidden array* to which the returned slice value refers.
  - Example:

```
a := make([]int, 5)  // len(a)=5, cap(a)=5
//To specify a capacity, pass a third argument to make:
b := make([]int, 0, 5) // len(b)=0, cap(b)=5

b = b[:cap(b)] // len(b)=5, cap(b)=5
b = b[1:]      // len(b)=4, cap(b)=4
```

  - This two forms are *equivalent*:

```
make([]int, 50, 100)
new([100]int)[0:50]
```

Multidimensional Slices (slices of slices)

✓ There is no multidimensional slices, instead **slice of slices** can be created:

- Example with *make()* function:

```
func main() {
    ss := make([][]uint8, 2) // ss variable is [][]uint8)
    fmt.Printf("ss:    %T %v %d/%d\n", ss, ss, len(ss), cap(ss))
    for i, s := range ss { // s is []uint8
        s = make([]uint8, 10, 20) // initialize internal slices!
        fmt.Printf("ss[%d]: %T %v %d/%d\n", i, s, s, len(s), cap(s))
    }
}
// OUTPUT:
//ss:    [][]uint8 [[] []] 2/2
//ss[0]: []uint8 [0 0 0 0 0 0 0 0 0 0] 10/20
//ss[1]: []uint8 [0 0 0 0 0 0 0 0 0 0] 10/20
```

Slices expression (short)

- ✓ Slice expressions construct a substring or slice from:
  - string
  - array
  - pointer to array
  - slice.

- ✓ For a string, array, pointer to array, or slice this expression constructs a substring or slice: `a[low : high]`
  - The result has *indices* starting at **0** and *length* equal to **high - low**.

- ✓ For convenience, any of the indices may be omitted.
  - A *missing low index* defaults to **zero**;
  - A *missing high index* defaults to the **length of the sliced operand**:

```
a[2:]  // same as a[2 : len(a)]
a[:3]  // same as a[0 : 3]
a[:]   // same as a[0 : len(a)]
```

- ✓ A *constant index* must be non-negative and representable by a value of type int.
  - If the indices are *out of range at run time*, a **run-time panic occurs.**

Slices expression (full)

- ✓ **Full** slice expressions (*rarely used*).
  - For an *array*, *pointer to array*, or *slice* a (but not a string), the primary expression `a[low : high : max]` constructs a slice of the *same type*, and with the *same length* and *elements* as the simple slice expression `a[low : high]`.
  - Additionally, it controls the resulting slice's **capacity** by setting it to `max - low`.
    - Only *the first index may be omitted* (it defaults to `0`).
  - Useful idea behind using full slice expression is to trigger a new buffer allocation instead of reusing data in original slice.
  - Example:

```
a := [5]int{1, 2, 3, 4, 5}
t := a[1:3:5]
// The slice t has type []int, length 2, capacity 4, and elements:
// t[0] == 2
// t[1] == 3
```

Iterating over Slices

- ✓ Slices can be iterated in **for** loop.
- ✓ When *ranging over a slice*, *two values* are returned for each iteration:
  - the first is the index
  - the second is a copy of the element at that index.
- We can *omit unnecessary variable* during iteration using underscore _ instead of variable name.

```go
package main
import "fmt"

var pow = []int{1, 2, 4, 8, 16, 32, 64, 128}

func main() {
    for i, v := range pow {
        fmt.Printf("2**%d = %d\n", i, v)
    }
}
// OUTPUT:
// 2**0 = 1
// 2**1 = 2
// 2**2 = 4
// 2**3 = 8
// 2**4 = 16
// 2**5 = 32
// 2**6 = 64
// 2**7 = 128
```

Built-in append() function

- ✓ Function **append()**:
  - The variadic function **append()** appends *zero or more values* x to s of type S, which must be a **slice** type, and returns the resulting **slice**, also of type S.
  - Remember that **append()** *may or may not allocate a new slice*!
  - As a special case, **append()** also accepts a *first argument assignable to type* ***[]byte*** with a second argument of **string** type followed by **...**
    - This form **appends** the **bytes** of the **string**.

```
append(s S, x ...T) S  // T is the element type of S
```

  - If the capacity of **s** is not large enough to fit the additional values, **append** allocates a *new*, sufficiently large underlying array that *fits both the existing slice elements and the additional values*.
    - Otherwise, **append** re-uses the underlying array.

```
s0 := []int{0, 0}
s1 := append(s0, 2)               // append a single element     s1 == []int{0, 0, 2}
s2 := append(s1, 3, 5, 7)         // append multiple elements    s2 == []int{0, 0, 2, 3, 5, 7}
s3 := append(s2, s0...)           // append a slice              s3 == []int{0, 0, 2, 3, 5, 7, 0, 0}
s4 := append(s3[3:6], s3[2:]...)  // append overlapping slice    s4 == []int{3, 5, 7, 2, 3, 5, 7, 0, 0}

var t []interface{}
t = append(t, 42, 3.1415, "foo")  //                            t == []interface{}{42, 3.1415, "foo"}

var b []byte
b = append(b, "bar"...)           // append string contents      b == []byte{'b', 'a', 'r' }
```

Built-in copy() function

- ✓ Function **copy()**:
  - The function **copy** copies slice elements from a source **src** to a destination **dst** and returns the number of elements copied.
  - Both arguments must have identical element type **T** and must be assignable to a slice of type **[]T**.
  - The number of elements copied is the minimum of **len(src)** and **len(dst)**.
  - As a special case, **copy** also accepts a destination argument assignable to type **[]byte** with a source argument of a **string** type.
    - This form copies the bytes from the string into the byte slice:

```
copy(dst, src []T) int
copy(dst []byte, src string) int
```

  - Examples:

```
var a = [...]int{0, 1, 2, 3, 4, 5, 6, 7}
var s = make([]int, 6)
var b = make([]byte, 5)
n1 := copy(s, a[0:])            // n1 == 6, s == []int{0, 1, 2, 3, 4, 5}
n2 := copy(s, s[2:])            // n2 == 4, s == []int{2, 3, 4, 5, 4, 5}
n3 := copy(b, "Hello, World!")  // n3 == 5, b == []byte("Hello")
```

To be reviewed on the next session

- ▶️ Common pitfalls with slices
- ▶️ Slicing slices pitfalls

Homework

- ▶️ Read: "Go maps in action" by Andrew Gerrand (https://go.dev/blog/maps)
- ▶️ Read: "If a map isn't a reference variable, what is it?" by Dave Cheney (https://dave.cheney.net/2017/04/30/if-a-map-isnt-a-reference-variable-what-is-it)
- ▶️ Read: "Go Slices: usage and internals" by Andrew Gerrand (https://blog.golang.org/go-slices-usage-and-internals)
- ▶️ Read: "Go slices are not dynamic arrays" by Applied Go (https://appliedgo.net/slices/)

Next time...

▶️ Session08: **Struct types and Interfaces in Go**

- Types vs. Interfaces: when and what to use?
- Struct types
  - overview of struct types
  - struct embedding
  - struct tags
  - pitfalls with struct types
- Interface types
  - Internals of interface types
  - Well-known pitfalls
  - Type assertions

Thank you

Golang course by Exadel

24 Oct 2022

Sergio Kovtunenko
Lead backend developer, Exadel
skovtunenko@exadel.com (mailto:skovtunenko@exadel.com)
https://github.com/skovtunenko (https://github.com/skovtunenko)
@realSKovtunenko (http://twitter.com/realSKovtunenko)