Lexical elements, literals, primitives, variables, constants, declarations, and their scope.

Session 03

Golang course by Exadel

11 Oct 2022

Sergio Kovtunenko
Lead backend developer, Exadel

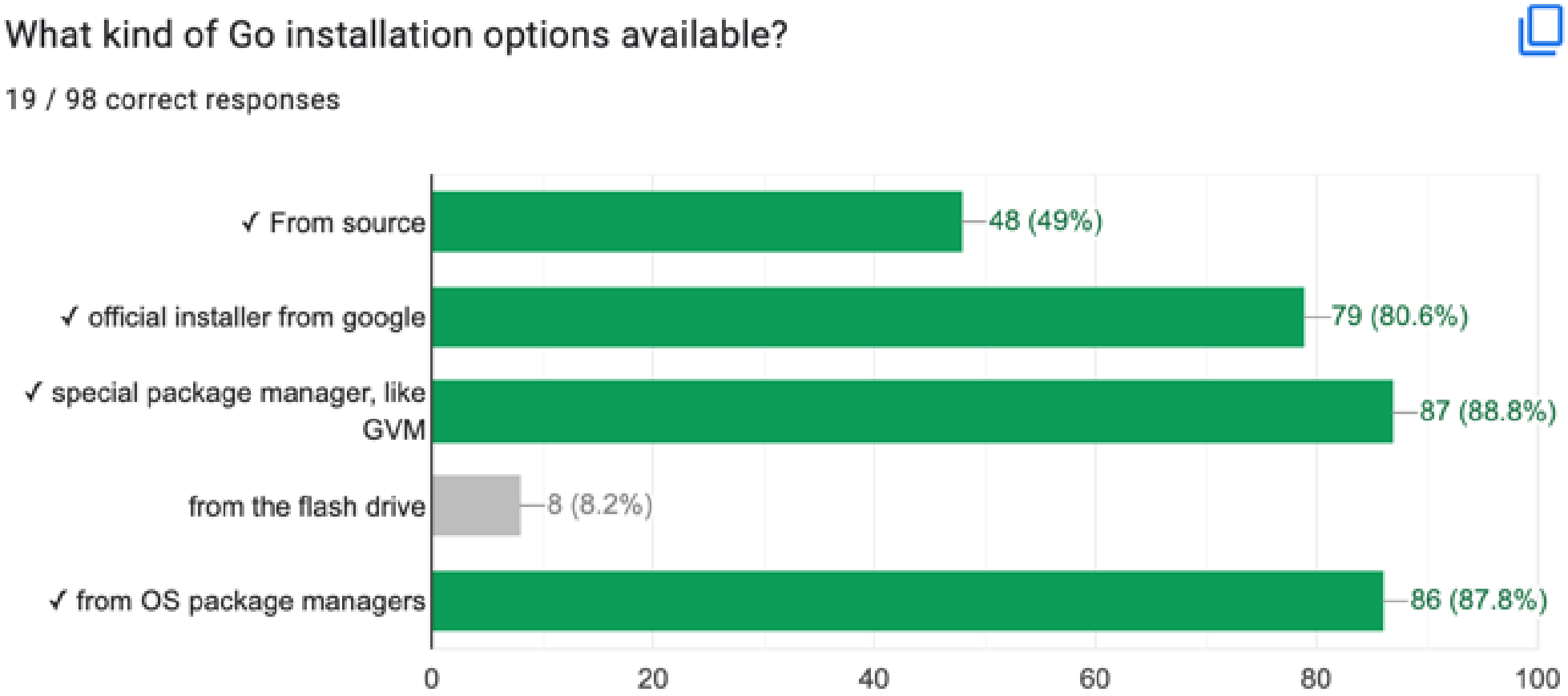
Agenda

- Revisit assessment results from the past session
- **Comments**
- **Keywords**, operators, and identifiers
- **Literals**
- Typed and Untyped **constants**
- Lack of enums
- **Variables**
- Conversions
- Variable type inference
- Next time...

Revisit assessment results from the past session

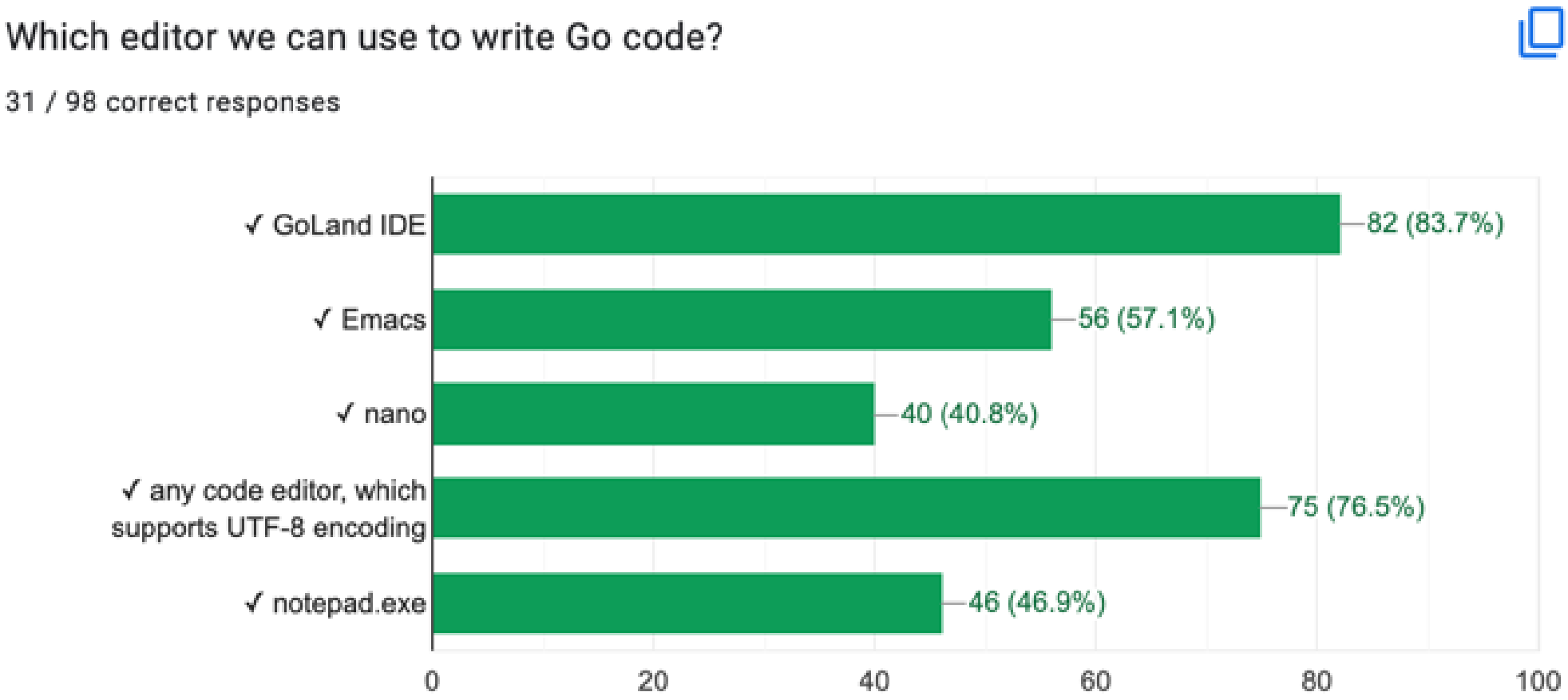
Frequently missed questions (1/2)

Question: What kind of Go installation options available?



Frequently missed questions (2/2)

Question: Which editor we can use to write Go code?



To learn more: "Editors and IDEs for Go" by Golang official Wiki (https://github.com/golang/go/wiki/IDEsAndTextEditorPlugins)

Comments

Comments

- **Line comments** start with the character sequence `//` and stop at the end of the line.
- **General comments** start with the character sequence `/*` and stop with the first subsequent character sequence `*/`.
- A **comment cannot start** inside a rune or string literal, or inside a comment.
- No nested `/* ... */` comments!

Keywords, operators, and identifiers

Semicolons, Tokens, Identifiers

▶️ A **newline** or **end of file** may trigger the insertion of a **semicolon** (;).

▶️ Semicolons:

- The formal grammar uses semicolons " ; " as terminators in a number of productions.
- **Compiler will insert** semicolons in many situations.

▶️ Identifiers:

- An **identifier** is a sequence of **one or more letters and digits**.

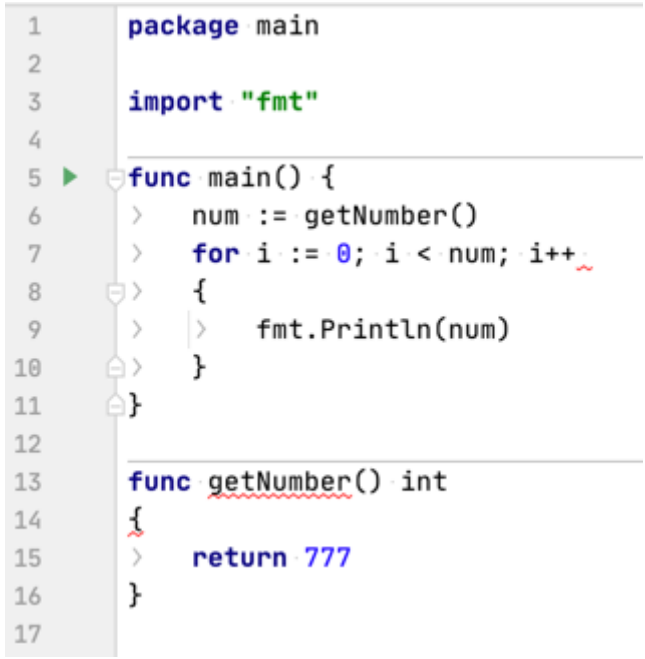
- The first character in an identifier **must be a letter**.

- Example:

```
a
_x9
ThisVariableIsExported
αβ
```

Compiler inserts semicolons (;)

▶️ Wrong code:



```
1  package main
2
3  import "fmt"
4
5  func main() {
6      num := getNumber()
7      for i := 0; i < num; i++
8      {
9          fmt.Println(num)
10     }
11 }
12
13 func getNumber() int
14 {
15     return 777
16 }
17
```

▶️ **Compiler will insert** semicolons in many situations.

Keywords, operator and delimeters

▶️ **Keywords** are reserved and may not be used as identifiers.

▶️ All keywords:

```
break        default      func         interface    select
case         defer        go           map          struct
chan         else         goto         package      switch
const        fallthrough  if           range        type
continue     for          import       return       var
```

▶️ The following character sequences represent **operators** (including assignment operators) and **punctuation**:

```
+    &    +=   &=   &&   ==   !=   (    )
-    |    -=   |=   ||   <    <=   [    ]
*    ^    *=   ^=   <-   >    >=   {    }
/    <<   /=   <<=  ++   =    :=   ,    ;
%    >>   %=   >>=  --   !    ...  .    :
&^   &^=
```

Literals

Integer literals

▶️ An **optional prefix** sets a non-decimal base

▶️ **0** for octal

▶️ **0x** or **0X** for hexadecimal

- letters **a-f** and **A-F** represent values **10** through **15**

▶️ Example of integer literals:

- 42
- 0600
- 0xBadFace
- 170141183460469231731687303715884105727

Floating point literals

▶️ Example of floating-point literals:

- `0.`
- `72.40`
- `072.40 // == 72.40 2.71828`
- `1.e+0`
- `6.67428e-11`
- `1E6`
- `.25`
- `.12345E+5`

Imaginary (complex) literals

▶️ Can be created by adding **“i”** postfix to constants.

▶️ Example:

- `0i`
- `011i // == 11i`
- `0.i`
- `2.71828i`
- `1.e+0i`
- `6.67428e-11i`
- `1E6i`
- `.25i`
- `.12345E+5i`

Rune literals

▶️ A **rune** means exactly the same as **"code point”**.

▶️ A **rune literal** represents a **rune constant**, an *integer value identifying a Unicode code point*.

- The Go language defines the word12 rune as an **alias** for the type **int32**, so programs can be clear when an integer value represents a code point.

▶️ A rune literal is expressed as one or more characters enclosed in single quotes, as in **'x'** or **'\n'**.

- Within the quotes, any character may appear **except newline and unescaped single quote**.

Rune Special values

▶️ Special values:

```
\a      U+0007 alert or bell
\b      U+0008 backspace
\f      U+000C form feed
\n      U+000A line feed or newline \r U+000D carriage return
\t      U+0009 horizontal tab
\v      U+000b vertical tab
\\      U+005c backslash
\'      U+0027 single quote (valid escape only within rune literals)
\"      U+0022 double quote (valid escape only within string literals)
```

▶️ Normal rune constants:

```
'a'
'ä'
'\t'
'\007'

'\''            // rune literal containing single quote character
'aa'            // illegal: too many characters
'\xa'           // illegal: too few hexadecimal digits
'\0'            // illegal: too few octal digits
'\uDFFF'        // illegal: surrogate half
'\U00110000'    // illegal: invalid Unicode code point
```

String literals

▶️ There are two **string** forms:

- **raw string** literals

- **interpreted string** literals

▶️ **Raw string literals** are character sequences between back quotes, as in **foo**.

- Within the quotes, any character may appear **except back quote**:

```
fmt.Println((`back ` + "`" + ` quote`)) // the way to overcome this limitation
```

String examples:

```
`abc`             // same as "abc"

`\n
\n`               // same as "\\n\n\\n"

"\n"

"\""              // same as `"`

"Hello, world!\n"

"\xff\u00FF"

"\uD800"          // illegal: surrogate half

"\U00110000"      // illegal: invalid Unicode code point
```

Typed and Untyped constants

Constants (1/2)

▶️ Constant expressions may contain *only constant operands* and are **evaluated at compile time**.

▶️ Constants **cannot** be declared using the **:=** syntax.

▶️ There are only:

- **boolean** constants

- **rune** constants

- **integer** constants

- **floating-point** constants (there are **no constants denoting the IEEE-754 negative zero, infinity, and not-a-number values**.)

- **complex** constants

- **string** constants

Constants (2/2)

▶️ **Floating-point constants** may have very high precision, so that arithmetic involving them is more accurate.

- The constants defined in the **math** package are given with many more digits than are available in a **float64**.

▶️ Here is the definition of **math.Pi** constant:

```
Pi = 3.14159265358979323846264338327950288419716939937510582097494459
```

▶️ When that value is assigned to a variable, some **precision will be lost**; the assignment will create the **float64** (or **float32**) value *closest to the high-precision value*.

- Having so many digits available means that calculations like **Pi/2** or other more intricate evaluations can carry more precision until the result is assigned, *making calculations involving constants easier to write without losing precision*.

Typed and untyped constants

▶️ Constants can be:

- typed
- untyped

▶️ Error example with **typed constants**:

```go
package main

import "fmt"

func main() {
    const typedHello string = "Hello, 世界"
    type MyString string
    var m MyString
    m = typedHello // Type error
    fmt.Println(m)
}
```

Default Type for a constants

▶️ **Untyped constant** has a **default type**, an implicit type that it transfers to a value if a type is needed where none is provided.

▶️ The **default type** of an untyped constant is **bool**, **rune**, **int**, **float64**, **complex128** or **string** respectively.

▶️ One way to think about untyped constants is that *they live in a kind of ideal space of values, a space less restrictive than Go's full type system*. But **to do anything with them**, we need to assign them to **variables**, and when that happens the variable (not the constant itself) needs a type, and the constant can tell the variable what type it should have.

▶️ In summary, a **typed constant** obeys all the rules of **typed values** in Go.

- On the other hand, an **untyped constant** does not carry a Go type in the same way and can be mixed and matched more freely.

- It does, however, have a **default type** that is exposed when, and only when, no other type information is available.

Examples of different constant types

```
const a = 2 + 3.0          // a == 5.0   (untyped floating-point constant)
const b = 15 / 4           // b == 3     (untyped integer constant)
const c = 15 / 4.0         // c == 3.75  (untyped floating-point constant)
const Θ float64 = 3/2      // Θ == 1.0   (type float64, 3/2 is integer division)
const Π float64 = 3/2.     // Π == 1.5   (type float64, 3/2. is float division)
const d = 1 << 3.0         // d == 8     (untyped integer constant)
const e = 1.0 << 3         // e == 8     (untyped integer constant)
const f = int32(1) << 33   // illegal    (constant 8589934592 overflows int32)
const g = float64(2) >> 1  // illegal    (float64(2) is a typed floating-point constant)
const h = "foo" > "bar"    // h == true  (untyped boolean constant)
const j = true             // j == true  (untyped boolean constant)
const k = 'w' + 1          // k == 'x'   (untyped rune constant)
const l = "hi"             // l == "hi"  (untyped string constant)
const m = string(k)        // m == "x"   (type string)
const Σ = 1 - 0.707i       //            (untyped complex constant)
const Δ = Σ + 2.0e-4       //            (untyped complex constant)
const Φ = iota*1i - 1/1i   //            (untyped complex constant)
```

Iota Constants

▶️ Within a **constant declaration**, the predeclared identifier **iota** represents *successive untyped integer constants*.

▶️ Its value is the **index** of the respective **constant declaration** in that constant declaration, **starting at zero**.

▶️ It can be used to construct a **set of related constants**.

```
const ( // iota is reset to 0
    c0 = iota  // c0 == 0
    c1 = iota  // c1 == 1
    c2 = iota  // c2 == 2
)

const ( // iota is reset to 0
    a = 1 << iota  // a == 1  (iota == 0)
    b = 1 << iota  // b == 2  (iota == 1)
    c = 3          // c == 3  (iota == 2, unused)
    d = 1 << iota  // d == 8  (iota == 3)
)

const ( // iota is reset to 0
    u         = iota * 42  // u == 0     (untyped integer constant)
    v float64 = iota * 42  // v == 42.0  (float64 constant)
    w         = iota * 42  // w == 84    (untyped integer constant)
)

const x = iota  // x == 0 (iota has been reset)
const y = iota  // y == 0 (iota has been reset)
```

Using iota in the middle

▶️ Example:

```
const (
        One   = 1
        Two   = 2
        // iota in the middle
        // Three = 2 + 1 => 3
        Three = iota + 1
        // Four  = 3 + 1 => 4
        Four
)
```

Source: "Ultimate Visual Guide to Go Enums and iota" by Inanc Gumus

(https://blog.learngoprogramming.com/golang-const-type-enums-iota-bc4befd096d3)

Iota values in the same line

▶️ In the **same line**, all constants will get the **same iota values**:

```
const (
    Active, Running = iota, iota + 100 // Active = 0, Running = 100  (iota == 0)
    Passive, Stopped                   // Passive = 1, Stopped = 101 (iota == 1)
    _, _                               // (iota == 2, unused)

    // You can't declare like this
    // Because, the last expression will be repeated
    //
    // When you uncomment this, the program will give you an error:
    //
    // CantDeclare

    // But, you can reset the last expression
    Reset = iota // here iota == 3

    // You can use any other expression even without iota
    AnyOther = 10 // here iota == 4, even if it is unused!
)
```

Iota last used expressions

▶️ The last used expression and type **will be repeated**:

```
const (
    /* 1st expression group */
    One int64 = iota + 1   // iota: 0, One  : 1 (type: int64)
    Two                    // iota: 1, Two  : 2 (type: int64)

    /* 2nd expression group */
    Four int32 = iota + 2   // iota: 2, Four : 4 (type: int32)
    Five                    // iota: 3, Five : 5 (type: int32)

    /* 3rd expression group */
    Six    = 6     // iota: 4, Six  : 6 (type: int)
    NotSeven       // iota: 5, Seven: 6 (type: int) -> The last used expression and type will be repeated.
    NotEight       // iota: 6, Eight: 6 (type: int) -> The last used expression and type will be repeated.
    NotNine        // iota: 7, Nine: 6 (type: int) -> The last used expression and type will be repeated.
)
```

We can count backwards with Iota

▶️ Counting backwards:

```
const (
      max = 10
)
const (
      a = (max - iota) // 10
      b                // 9
      c                // 8
)
```

(iota+1) trick

▶️ `iota + 1` trick:

```go
type Activity int

const (
	Sleeping = iota + 1
	Walking
	Running
)

func main() {
	// activity initialized to zero which is Sleeping
	var activity Activity

	// now you know about its initialization
	fmt.Println(activity == Sleeping)
	fmt.Println(activity == 0) // not initialized => true
}
```

▶️ This is very important for **serialization logic** (because of default values)

Lack of enums

▶️ **Enums** are types that contain only a **limited number of fixed values**, as opposed to types like **int** or **string** which can have a wide range of values.

▶️ We can see **enums** used throughout the standard library. Some common examples are:

- Prefix flags in the log package (https://pkg.go.dev/log#pkg-constants)
- Status codes and methods in the http package (https://pkg.go.dev/net/http#pkg-constants)
- Standard grayscale colors in the color package (https://pkg.go.dev/image/color#pkg-variables)

▶️ Simulate them using code generation approach, for example:

- Stringer (https://pkg.go.dev/golang.org/x/tools/cmd/stringer)
- abice/go-enum (https://github.com/abice/go-enum)
- etc.

Hand-made Enum example:

```go
package main

type Season int64

const (
    Summer Season = iota + 1
    Autumn
    Winter
    Spring
)

func (s Season) String() string {
    switch s {
    case Summer:
        return "summer"
    case Autumn:
        return "autumn"
    case Winter:
        return "winter"
    case Spring:
        return "spring"
    }
    return "unknown"
}
```

Variables

Basic info about Variables (1/2)

▶️ A variable is a **storage location** for holding a value.

- All value assignments, including parameter passings etc, are **shallow value copying**: the value pointed by pointers will not be copied

▶️ The set of permissible values is determined by the **variable's type**.

▶️ There are **structured variables** of types:

- **array**
- **slice**
- **map**
- **struct**

Basic info about Variables (2/2)

▶️ There are:

- **static types** (or *just types*): type of variable **is given in variable declaration**.
- **dynamic types**:
  - Variables of **interface type** also have a *distinct dynamic type*;
  - The **dynamic type** may vary during execution but values stored in **interface variables** are always assignable to the static type of the variable.

▶️ Example:

```
var x interface{}  // x is nil and has static type interface{}
var v *T           // v has value nil, static type *T
x = 42             // x has value 42 and dynamic type int
x = v              // x has value (*T)(nil) and dynamic type *T
```

Source: "There Are No Reference Types in Go" by Tapir Games (http://www.tapirgames.com/blog/golang-has-no-reference-values)

Types of variable declaration (1/3)

▶️ **Composite** declarations:

```
var (
    name     string
    age      int
    location string
)

var (
    name, location  string
    age             int
)
```

▶️ **Full** declaration:

```
var name     string
var age      int
var location string
var x        interface{}
```

Types of variable declaration (2/3)

▶️ Declaration can include initializers, **one per variable**:

- Variables could also be **initialized with expressions computed at runtime**, like calls to functions.

```
var (
    name     string = "Prince Oberyn"
    age      int    =  32
    location string = "Dorne"
)
```

▶️ Declaration **without** type specification:

```
var (
    name, location, age = "Prince Oberyn", "Dorne", 32
)

var (
    name     = "Prince Oberyn"
    age      =  32
    location = "Dorne"
)
```

Types of variable declaration (3/3)

- Short declaration **only inside functions**.
- Outside a function, every construct begins with a keyword (**var**, **func**, and so on) and the **:= construct is not available**:

```
func main() {
    name, location := "Prince Oberyn", "Dorne"
    age := 32
    fmt.Printf("%s (%d) of %s", name, age, location)
}
```

- If a variable has not yet been assigned a value, its value is the "**zero value**" for its type.
- There are **no concept** of "*uninitialized variable*" in GO.

Short variable declarations

Short variable declarations

▶️ A **short variable declaration** uses the `:=` syntax.

▶️ Short variable declarations may appear **only inside functions**.

- In some contexts such as the initializers for "**if**", "**for**", or "**switch**" statements, they can be used to **declare local temporary variables**.

▶️ It is *shorthand* for a regular variable declaration with **initializer expressions but no types**:

```
i, j := 0, 10
f := func() int { return 7 }
ch := make(chan int)
r, w := os.Pipe(fd)  // os.Pipe() returns two values
_, y, _ := coord(p)  // coord() returns three values; only interested in y coordinate
```

Short variable RE-declaration

▶️ Unlike regular variable declarations, a short variable declaration **may redeclare** (with the same type):

- variables declared earlier in the same block **OR**
- or the parameter lists if the block is the function body.

▶️ Redeclaration does not introduce a new variable; it just assigns a new value to the original.

```
field1, offset := nextField(str, 0)
field2, offset := nextField(str, offset)  // redeclares offset
a, a := 1, 2                              // illegal: double declaration of a or no new variable if a was
declared elsewhere
```

Short variable RE-declaration GOTCHA

▶️ Variable hiding gotcha:

```
func main() {
    v := 1
    {
        v := 2  // short variable declaration
        fmt.Println(v) // OUTPUT: 2
    }
    fmt.Println(v) // OUTPUT: 1
}
```

Variable Type Conversions

Variable Type Conversions

▶️ The expression `T(v)` converts the value v to the type T.

▶️ In Golang assignment between items of **different type** requires an **explicit conversion**

- which means that you **manually** need to convert types if you are passing a variable to a function expecting another type.

▶️ Very important: if the type starts with the operators/keyword it **must be parenthesized** when necessary (to avoid ambiguity):

- *****, **<-**, **func** and *has no result list*

```
*Point(p)        // same as *(Point(p))
(*Point)(p)      // p is converted to *Point
<-chan int(c)    // same as <-(chan int(c))
(<-chan int)(c)  // c is converted to <-chan int
func()(x)        // function signature func() x
(func())(x)      // x is converted to func()
(func() int)(x)  // x is converted to func() int
func() int(x)    // x is converted to func() int (unambiguous)
```

Conversion example:

```
type AwsRegion string

func main() {
    var region AwsRegion = "us-west-2"
    someLegacyFunctionNotUnderYourControl((*string)(&region)) // parentheses required!
}

func someLegacyFunctionNotUnderYourControl(s *string) {
    // do something with parameter
}
```

▶️ There is **no** linguistic mechanism to convert **between pointers and integers**.

Conversions `to` and `from` a string type

▶️ Converting a **signed or unsigned integer value** to a **string** type yields a string containing the UTF-8 representation of the integer.

- Values outside the range of valid Unicode code points are converted to "\uFFFD".

```
string('a')         // "a"
string(-1)          // "\ufffd" == "\xef\xbf\xbd"
string(0xf8)        // "\u00f8" == "ø" == "\xc3\xb8"
type MyString string

MyString(0x65e5)  // "\u65e5" == "日" == "\xe6\x97\xa5"
```

▶️ Converting a **slice of bytes** to a **string** type yields a string whose successive bytes are the elements of the slice.

```
string([]byte{'h', 'e', 'l', 'l', '\xc3', '\xb8'})   // "hellø"
string([]byte{})                                     // ""
string([]byte(nil))                                  // ""

type MyBytes []byte
string(MyBytes{'h', 'e', 'l', 'l', '\xc3', '\xb8'})  // "hellø"
```

Important note about memory allocations:

▶️ converting “to”/“from” string **usually does make a copy of content**, as string values are **immutable**

- For example if converting a **string** to **[]byte** would not make a copy, you could change the content of the string by changing elements of the resulting byte slice.

▶️ Sometimes **compiler can omit** new memory allocation as optimisation.

Conversions between numeric types

▶️ For the conversion of non-constant numeric values, the following rules apply:

- When converting **between integer types**, if the value is a signed integer, it is sign extended to implicit infinite precision; otherwise it is zero extended.
- When converting a **floating-point number to an integer**, the fraction is **discarded** (truncation towards zero).
- When converting an **integer or floating-point number to a floating-point type**, or a **complex number to another complex type**, the result value is **rounded** to the precision specified by the destination type.

Next time...

▶️ Session04:

Expressions and Statements

- Expressions
- Statements

Thank you

Golang course by Exadel

11 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)