The GO Programming Language

Patrik Schweigl
Supervisor: Dr. Dohan Kim
University of Innsbruck

Outline

1. Introduction to Go - History
2. The Language
3. Conclusion

Why a new language

- Things are taking too long.

Why a new language

- Things are taking too long.
- No new system language in years, but much has changed.
    - Focus on networking
    - Focus on Client / Server architecture
    - Dependencies are growing
    - Focus on Cluster / multicore CPUs

Why a new language

- Things are taking too long.
- No new system language in years, but much has changed.
    - Focus on networking
    - Focus on Client / Server architecture
    - Dependencies are growing
    - Focus on Cluster / multicore CPUs
- Type system is too rigid in statically-typed compiled languages.

Why a new language

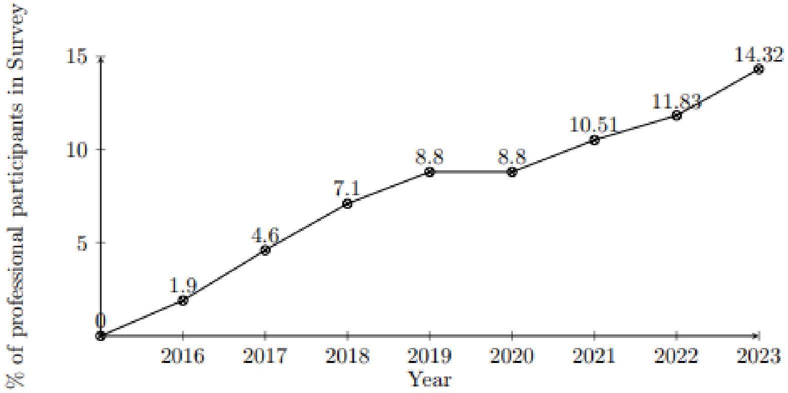
- Things are taking too long.
- No new system language in years, but much has changed.
  - Focus on networking
  - Focus on Client / Server architecture
  - Dependencies are growing
  - Focus on Cluster / multicore CPUs
- Type system is too rigid in statically-typed compiled languages.
- These problems are language endemic.

History

- Started 2007 at Google by Robert Griesemer, Rob Pike and Ken Thompson.
- Publicly announced in November 2009 and went Open Source.
- Go 1.0 was released in March 2012.
- Current Version: go 1.20 was released in February 2023.

Stack Overflow Survey

Figure 1: Stack Overflow Survey from 2015 to 2023



Facts

- Designed for efficiency, simplicity, and scalability

Facts

- Designed for efficiency, simplicity, and scalability
- Compiled language

Facts

- Designed for efficiency, simplicity, and scalability
- Compiled language
- System Language

Facts

- Designed for efficiency, simplicity, and scalability
- Compiled language
- System Language
- Strongly typed with static type checking

Facts

- Designed for efficiency, simplicity, and scalability
- Compiled language
- System Language
- Strongly typed with static type checking
- Built-In Concurrency support

Facts

- Designed for efficiency, simplicity, and scalability
- Compiled language
- System Language
- Strongly typed with static type checking
- Built-In Concurrency support
- Garbage Collector

Facts

- Designed for efficiency, simplicity, and scalability
- Compiled language
- System Language
- Strongly typed with static type checking
- Built-In Concurrency support
- Garbage Collector
- Fast compilation time through better dependency handling

Hello, World! Program in Go

```
// simple Hello World program
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
```

Variables and Types

- Go supports various data types, including:
  - `int`, `float`, `bool` for basic types
  - `string` for text
  - Example: `var value int32 = 42`

Variables and Types

- Go supports various data types, including:
  - `int`, `float`, `bool` for basic types
  - `string` for text
  - Example: `var value int32 = 42`
- Type inference allows omitting explicit type declarations.
  - Example: `var value = 42`
  - Or even better: `value := 42`

Variables and Types

- Go supports various data types, including:
  - `int`, `float`, `bool` for basic types
  - `string` for text
  - Example: `var value int32 = 42`
- Type inference allows omitting explicit type declarations.
  - Example: `var value = 42`
  - Or even better: `value := 42`
- Go supports Constants for numeric and boolean types, strings and runes.
  - `const Pi = 3.141592653589793238462643383279502884197169399...`
  - `const OneOverPi = 1 / Pi`

Functions

- Multiple return values:

```
func divide(a, b int) (int, int){
  return a/b, a%b
}
```

Functions

- Multiple return values:

```
func divide(a, b int) (int, int){
  return a/b, a%b
}
```

- Named return values:

```
func divide(a, b int) (res int, rem int){
  res, rem = a/b, a%b
  return
}
```

Functions

- Multiple return values:

```
func divide(a, b int) (int, int){
  return a/b, a%b
}
```

- Named return values:

```
func divide(a, b int) (res int, rem int){
  res, rem = a/b, a%b
  return
}
```

- Anonymous functions (closures)

Concurrency

- **Concurrency is not Parallelism.**

Concurrency

- **Concurrency is not Parallelism.**
- Concurrency is about dealing with multiple things at once.

Concurrency

- **Concurrency is not Parallelism.**
- Concurrency is about dealing with multiple things at once.
- Parallelism is about doing lots of things at once.

Concurrency

- **Concurrency is not Parallelism.**
- Concurrency is about dealing with multiple things at once.
- Parallelism is about doing lots of things at once.
- Concurrency is more about structure.

Concurrency

- **Concurrency is not Parallelism.**
- Concurrency is about dealing with multiple things at once.
- Parallelism is about doing lots of things at once.
- Concurrency is more about structure.
- Parallelism is about execution.

Concurrency: Goroutines

- Lightweight concurrent functions.
- Executed independently and concurrently.
- Enable efficient utilization of resources.
- Created using the go keyword.

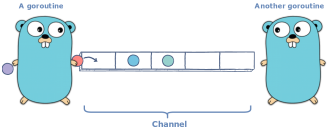


Figure: Goroutines in action

Concurrency: Goroutines Example

```go
func LongCalculation() int {
  time.Sleep(2 * time.Second) // simulation
  return 42
}
func main() {
  res := 0
  go func() {
    res = LongCalculation()
  }()
  for res == 0 {
  }
  fmt.Println(res)
}
```

Concurrency: Channels

- Communication mechanism between goroutines.
- Enable safe data exchange and synchronization.
- Prevent race conditions and data races.
- Sending and receiving data using the <− operator.

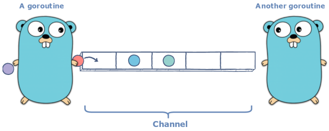


Figure: Goroutines in action

Concurrency: Channel Example

```
func LongCalculation() int {
    time.Sleep(2 * time.Second)
    return 42
}
func main() {
    channel := make(chan int)
    go func() {
        channel <- LongCalculation()
    }()
    fmt.Println(<-channel)
}
```

Structs

- Go uses struct for defining custom types.

```
type Person struct {
  First string
  Last string
}
person := Person{"Patrik", "Schweigl"}
```

Structs

- Go uses `struct` for defining custom types.

```go
type Person struct {
  First string
  Last string
}
person := Person{"Patrik", "Schweigl"}
```

- Methods can be implemented outside of struct

```go
func (p Person) FullName() string {
  return p.First + " " + p.Last
}
n := person.FullName()
```

Struct Embedding

- Promote composition over inheritance through struct embedding

```
type Employee struct {
  Person // embedded struct
  Email string
}
emp := Employee{Person{"Patrik", "Schweigl"}, "email"}
n := emp.FullName()
```

Interfaces

- Go supports `interface` for defining contracts.

```
type Magnitude interface {
  Abs() float64
}
type Point2D struct{ X, Y float64 }
func (p Point2D) Abs() float64 { return math.Sqrt(p.X*p.
    X + p.Y*p.Y) }
var x Magnitude = Point2D{}
```

Interfaces

- Go supports `interface` for defining contracts.

```
type Magnitude interface {
  Abs() float64
}
type Point2D struct{ X, Y float64 }
func (p Point2D) Abs() float64 { return math.Sqrt(p.X*p.
    X + p.Y*p.Y) }
var x Magnitude = Point2D{}
```

- Implicitly satisfied, when all methods are implemented

Interfaces

- Go supports `interface` for defining contracts.

```
type Magnitude interface {
  Abs() float64
}
type Point2D struct{ X, Y float64 }
func (p Point2D) Abs() float64 { return math.Sqrt(p.X*p.
    X + p.Y*p.Y) }
var x Magnitude = Point2D{}
```

- Implicitly satisfied, when all methods are implemented
- Enable polymorphism

Visibility

- Go manages code in packages

Visibility

- Go manages code in packages
- Start with an uppercase letter, to make struct / fields / variables accessible outside package.

Visibility

- Go manages code in packages
- Start with an uppercase letter, to make struct / fields / variables accessible outside package.
- Start with a lowercase letter, to make struct / field / variables not accessible outside package.

Visibility

- Go manages code in packages
- Start with an uppercase letter, to make struct / fields / variables accessible outside package.
- Start with a lowercase letter, to make struct / field / variables not accessible outside package.

```
package point3Dlib
type Point3D struct{ x, Y, Z float32 }
func (p Point3D) GetX() float32 { return p.x}
func (p *Point3D) SetX(x float32) {p.x = x}
```

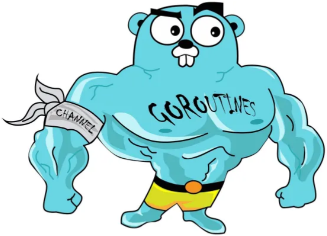
Idiomatic Error Handling

Go follows an idiomatic error handling pattern:

```go
func divide(a, b float64) (float64, error) {
  if b == 0 {return 0, errors.New("Divide by 0!")}
  return a / b, nil
}
func main() {
  if res, err := divide(12, 2); err != nil {
    fmt.Println(err.Error())
  } else {fmt.Println(res)}
}
```

Go’s strength

- Simple, yet powerful syntax
- Designed for a modern era
- Built-in support for concurrency
- Rich built-in tooling
- Rich standard library
- Open-Source



Go’s weakness

- Go lacks Enum types
- Method / Function overloading is missing
- Error handling is error-prone
- Garbage Collector

Sources

- https://gobyexample.com/
- https://en.wikipedia.org/wiki/Go_(programming_language)
- https://go.dev/doc/
- https://go.dev/talks/
- https://www.geeksforgeeks.org/golang/

Sources Images

- https://xkcd.com/303/
- https://www.educative.io/answers/what-are-channels-in-golang
- https://articles.wesionary.team/understanding-go-routine-and-channel-b09d7d60e575
- https://www.educative.io/answers/what-are-channels-in-golang
- https://www.golinuxcloud.com/golang-gopher/

Sources Images Cont

- https://play.google.com/store/apps/details?id=eu.ydns.chernish2_go_free
- https://de.wikipedia.org/wiki/Robert_Griesemer
- https://de.wikipedia.org/wiki/Rob_Pike
- https://en.wikipedia.org/wiki/Ken_Thompson

Thank you for the attention