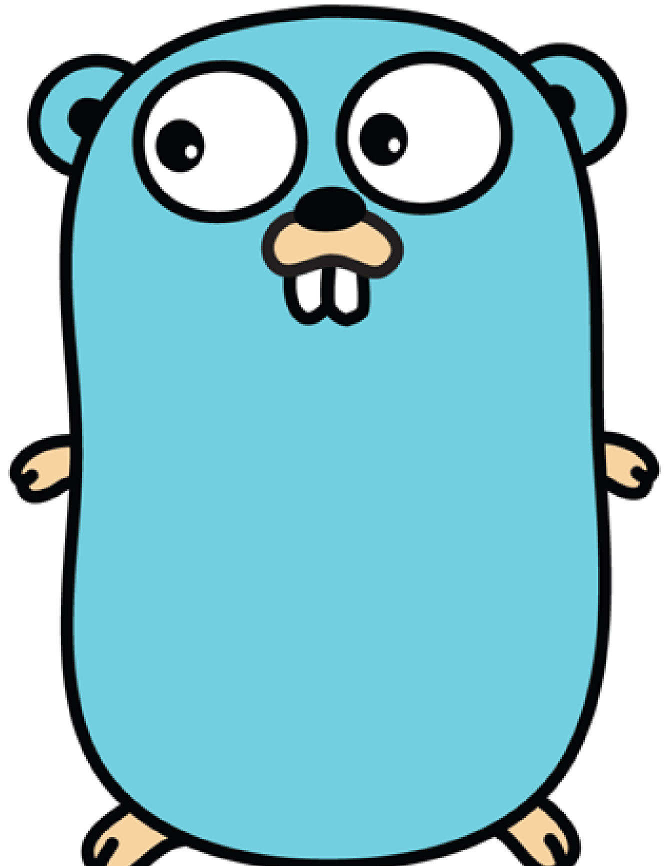
Generics in Go



Brief overview of Go

- Developed by Google in 2007, open-sourced in 2009.
- Designed for simplicity, performance, and scalability.
- **Efficient Concurrency:** Built-in support for lightweight goroutines and channels.
- **Simple Syntax:** Easy to learn and use; inspired by C but with a modern approach.
- **Fast Performance:** Compiles to machine code; no virtual machine required.
- **Garbage Collection:** Automatic memory management.
- **Standard Library:** Extensive, with built-in support for web servers, I/O, and more.
- Installation guide: https://go.dev/dl/

Introduction

- What are Generics?
  - Generics enable you to write reusable code by using type parameters in functions and data structures.
- Why use Generics?
  - Type safety without boilerplate
  - Code reusability
  - Improved performance by avoiding type casting

Problem Without Generics

```
func SumInts(numbers []int) int {  no usages
    total := 0
    for _, num := range numbers {
        total += num
    }
    return total
}

func SumFloats(numbers []float64) float64 {
    total := 0.0
    for _, num := range numbers {
        total += num
    }
    return total
}
```

- Example: Sum function
- Issues:
  - Code duplication
  - Prone to errors when scaling for new types

Solution With Generics

- Example: Generic Sum function

```go
func SumGeneric[T int | float64](numbers []T) T {
    var total T
    for _, num := range numbers {
        total += num
    }
    return total
}
```

- Benefits:
  - One function for multiple types
  - Type-safe operations

Anatomy of a Generic Function

- Syntax: func FunctionName[T TypeConstraint](parameters) ReturnType
- Key Parts:
  - T: Type parameter
  - TypeConstraint: Restricts T to specific types or interfaces

```go
func PrintSlice[T any](items []T) {
    for _, item := range items {
        fmt.Println(item)
    }
}
```

- T any: any means any type is allowed

Using Constraints

- Constraints: Define what operations are valid for a type parameter.
- Built-in Constraints: any, comparable

```go
func Find[T comparable](slice []T, value T) bool {
    for _, v := range slice {
        if v == value {
            return true
        }
    }
    return false
}
```

- Use Case: Searching in slices of any comparable type (e.g., int, string)

Custom Constraints

- Define Your Own Constraint

```
type Number interface {  2 usages  alexovidiupopa
    int | float64
}

func SumWithCustomInterface[T Number](numbers []T) T {
    var total T
    for _, num := range numbers {
        total += num
    }
    return total
}

func Multiply[T Number](a, b T) T {  1 usage  alexovidiupop
    return a * b
}
```

Practical Example: Generic Stack

- Generic Stack Implementation

```go
type Stack[T any] struct {   2 usages   alexovidiupopa
    elements []T
}

func (s *Stack[T]) Push(value T) {   no usages   alexovidiupopa
    s.elements = append(s.elements, value)
}

func (s *Stack[T]) Pop() (T, bool) {   no usages   alexovidiupopa
    if len(s.elements) == 0 {
        var zeroValue T
        return zeroValue, false
    }
    value := s.elements[len(s.elements)-1]
    s.elements = s.elements[:len(s.elements)-1]
    return value, true
}
```

Migration Tips

- Start Small: Convert repetitive functions to generics
- Use Built-in Constraints: Utilize *any* and *comparable*
- Test Thoroughly: Ensure your generic code works with all expected types
- Leverage Tools: Use linters to catch type constraint issues

Conclusion

- Why Use Generics?
  - Write cleaner, reusable, and type-safe code
- What’s Next?
  - Explore generic libraries in Go
  - Refactor existing projects with generics

Questions? Thank you!