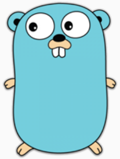
Programming in Go

Matt Holiday
Christmas 2020



Parametric Polymorphism

Generics in Go

"Generics" is shorthand for *parametric polymorphism*

That means we have a **type parameter** on a type or function

```
type MyType[T any] struct {
    v T                       // can be any valid Go type
    n int
}
```

Generics are a powerful feature for abstraction

And a possible source of *unnecessary* abstraction and complexity

Generics in Go

Use type parameters to **replace dynamic typing** with static typing

```
interface{}  +  v.(T)
```

$$\Downarrow$$

```
type MyType[T any] struct{ . . . }
```

If it runs faster, consider that a bonus

Continue to use (non-empty) interfaces wherever possible

Performance should not be your principal reason for generics (in most cases)

Generic type & function

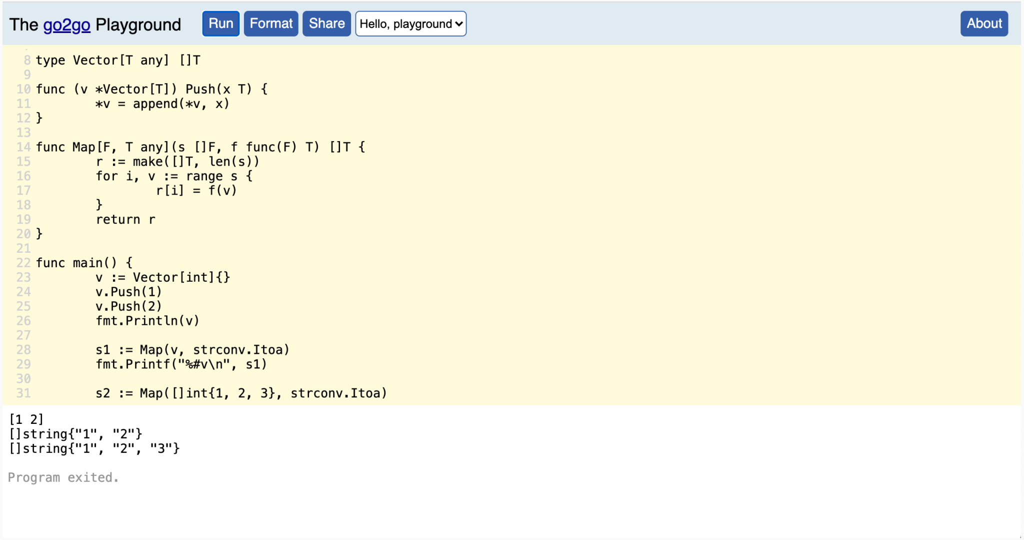
```go
type Vector[T any] []T

func (v *Vector[T]) Push(x T) {
    *v = append(*v, x)             // may reallocate
}

// note: F and T are both used in the parameter list

func Map[F, T any](s []F, f func(F) T) []T {
    r := make([]T, len(s))

    for i, v := range s {
        r[i] = f(v)
    }

    return r
}
```

Generic type & function

```go
func main() {
    v := Vector[int]{}

    v.Push(1)
    v.Push(2)

    s1 := Map(v, strconv.Itoa)
    s2 := Map([]int{1, 2, 3}, strconv.Itoa)

    fmt.Println(v)
    fmt.Printf("%#v\n", s1)
    fmt.Printf("%#v\n", s2)
}
```

Note: `Map` is a textbook example and not necessarily a good idea

Go 2 playground



```go
type Vector[T any] []T

func (v *Vector[T]) Push(x T) {
	*v = append(*v, x)
}

func Map[F, T any](s []F, f func(F) T) []T {
	r := make([]T, len(s))
	for i, v := range s {
		r[i] = f(v)
	}
	return r
}

func main() {
	v := Vector[int]{}
	v.Push(1)
	v.Push(2)
	fmt.Println(v)

	s1 := Map(v, strconv.Itoa)
	fmt.Printf("%#v\n", s1)

	s2 := Map([]int{1, 2, 3}, strconv.Itoa)
```

```
[1 2]
[]string{"1", "2"}
[]string{"1", "2", "3"}

Program exited.
```

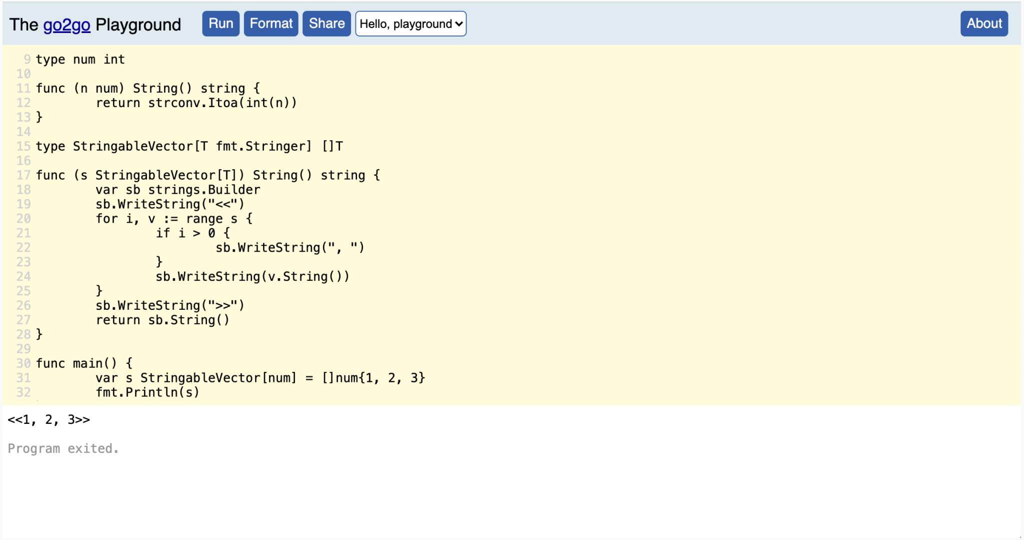
Generic type & method

```go
type num int

func (n num) String() string {
    return strconv.Itoa(int(n))
}

// type constraint: T must have String() method

type StringableVector[T fmt.Stringer] []T
```

Generic type & method

```
func (s StringableVector[T]) String() string {
    var sb strings.Builder
    sb.WriteString("<<")
    for i, v := range s {
        if i > 0 {
            sb.WriteString(", ")
        }
        sb.WriteString(v.String())
    }
    sb.WriteString(">>")
    return sb.String()
}

func main() {
    var s StringableVector[num] = []num{1, 2, 3}   // [num] required on type
    fmt.Println(s)
}
```

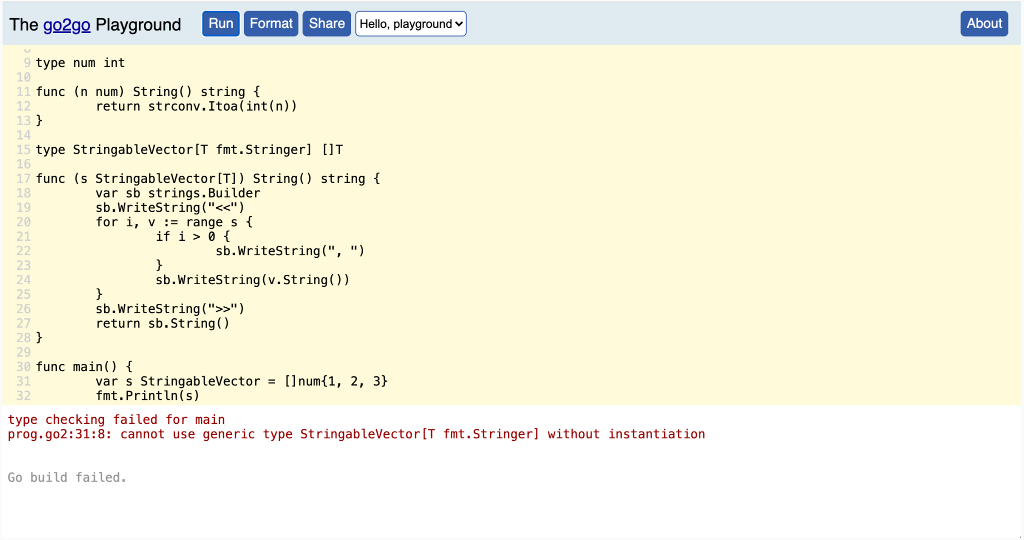
Go 2 playground



```go
type num int

func (n num) String() string {
	return strconv.Itoa(int(n))
}

type StringableVector[T fmt.Stringer] []T

func (s StringableVector[T]) String() string {
	var sb strings.Builder
	sb.WriteString("<<")
	for i, v := range s {
		if i > 0 {
			sb.WriteString(", ")
		}
		sb.WriteString(v.String())
	}
	sb.WriteString(">>")
	return sb.String()
}

func main() {
	var s StringableVector[num] = []num{1, 2, 3}
	fmt.Println(s)
```

```
<<1, 2, 3>>

Program exited.
```

Go 2 instantiation error



```
type num int

func (n num) String() string {
        return strconv.Itoa(int(n))
}

type StringableVector[T fmt.Stringer] []T

func (s StringableVector[T]) String() string {
        var sb strings.Builder
        sb.WriteString("<<")
        for i, v := range s {
                if i > 0 {
                        sb.WriteString(", ")
                }
                sb.WriteString(v.String())
        }
        sb.WriteString(">>")
        return sb.String()
}

func main() {
        var s StringableVector = []num{1, 2, 3}
        fmt.Println(s)
```

```
type checking failed for main
prog.go2:31:8: cannot use generic type StringableVector[T fmt.Stringer] without instantiation


Go build failed.
```