# Rusty Gophers

SCAN ME

Mateusz Szczyrzyca

https://devopsiarz.pl

# Important disclaimer!

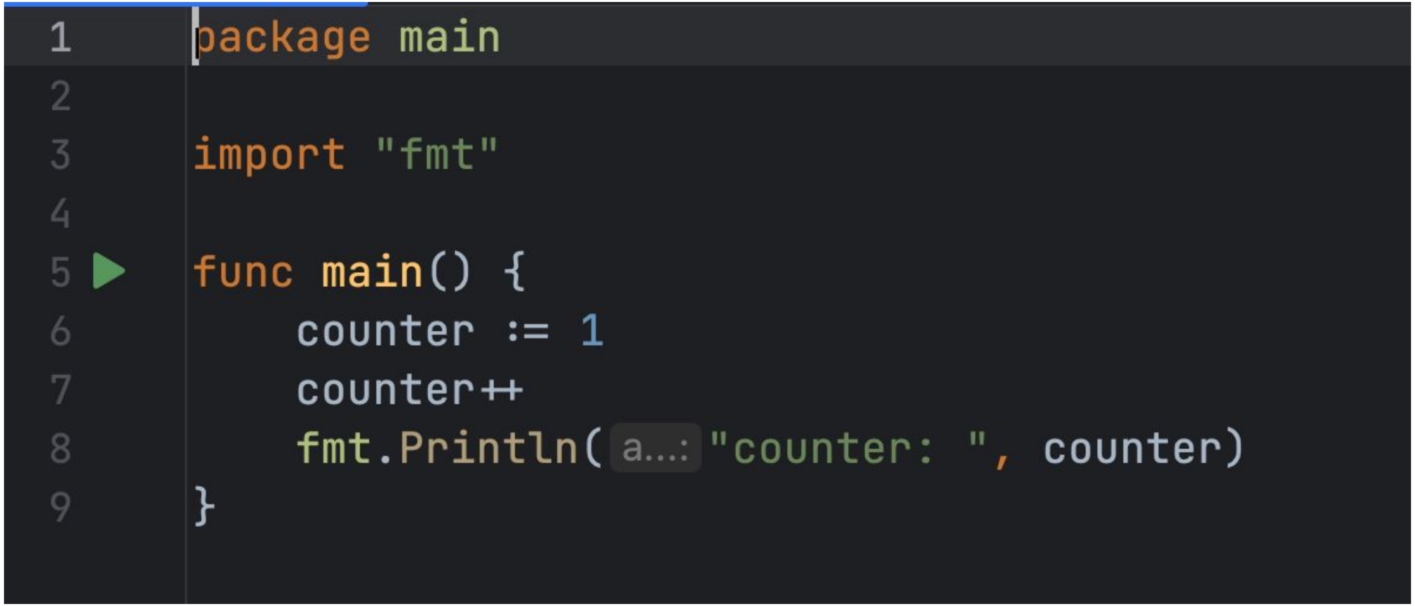
1) My opinions - may be wrong

2) Use styles and coding techniques approved by your team

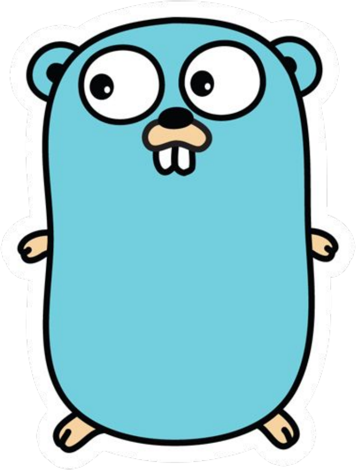
3) Examples may not be perfect

# Rule: use compiler to verify as much as possible

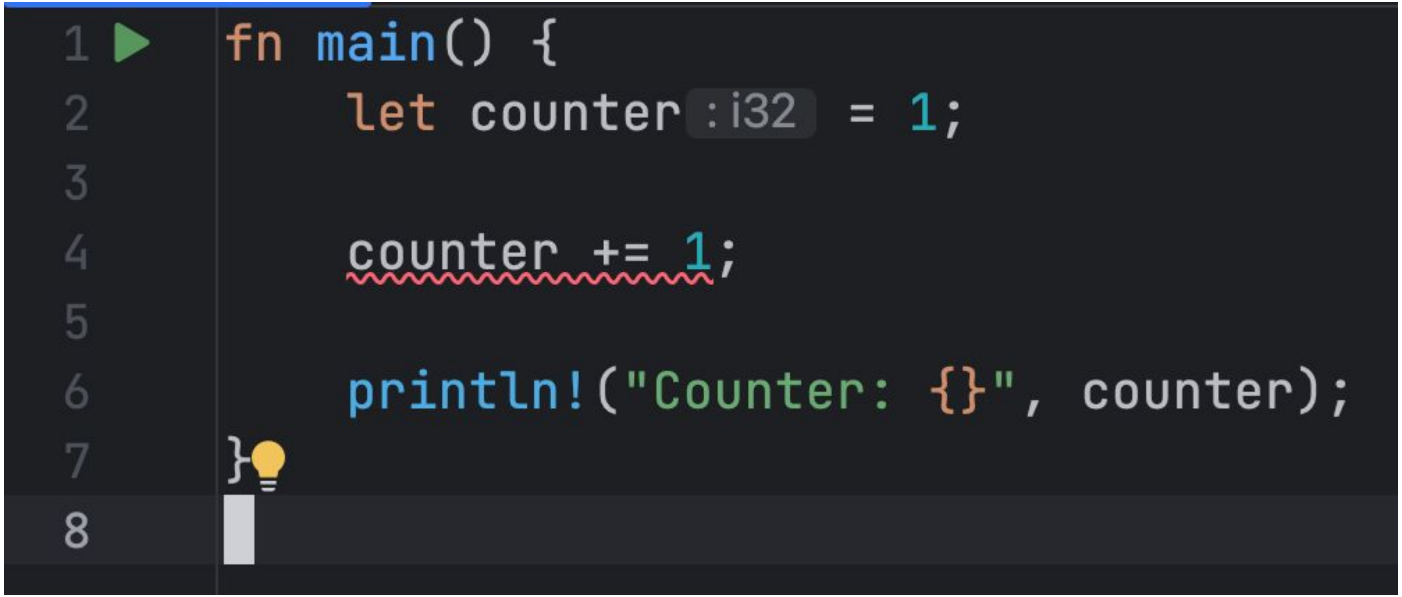
# 1. Immutability by default



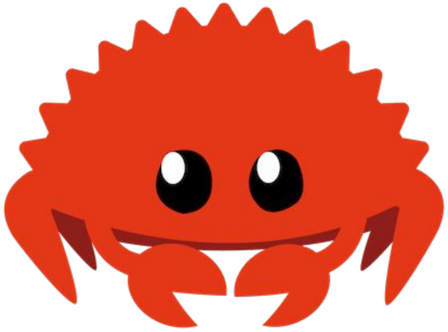
```go
package main

import "fmt"

func main() {
	counter := 1
	counter++
	fmt.Println(a...: "counter: ", counter)
}
```



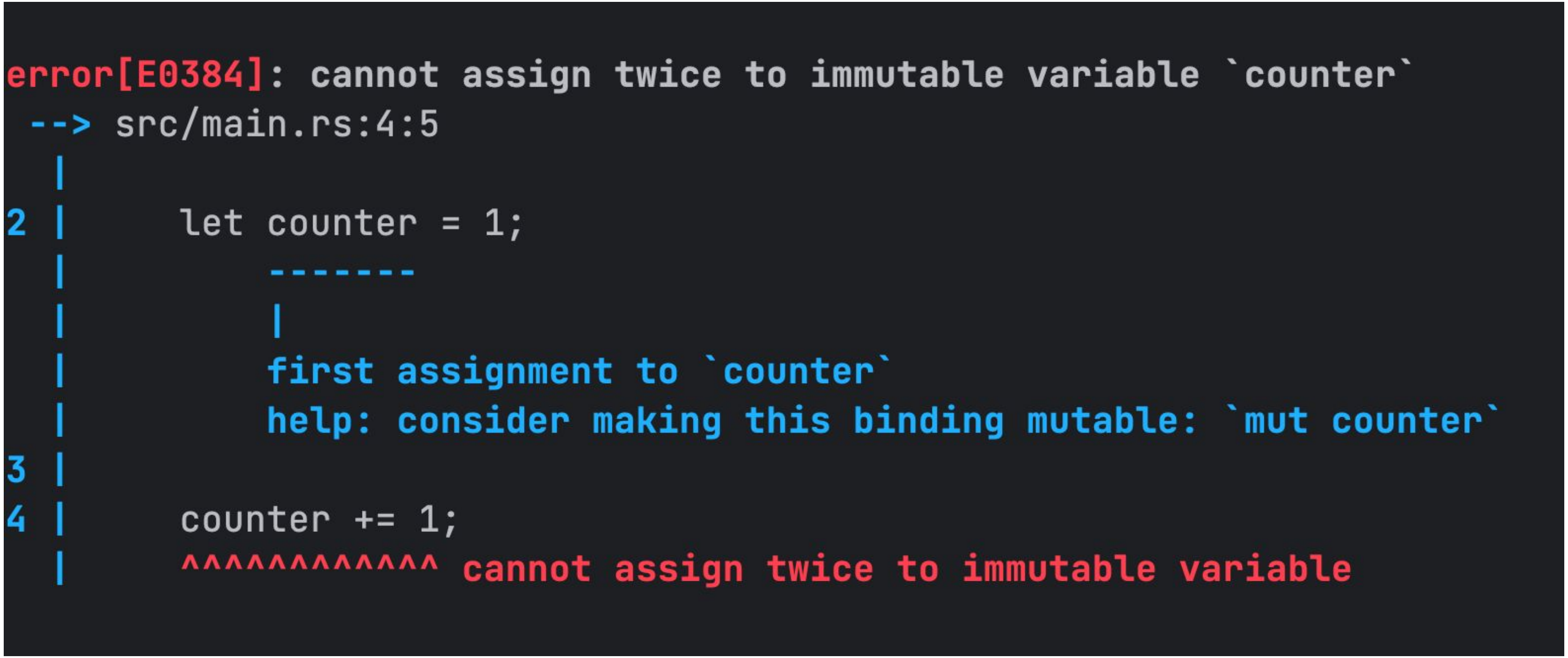
# 1. Immutability by default



```rust
fn main() {
    let counter: i32 = 1;

    counter += 1;

    println!("Counter: {}", counter);
}

```

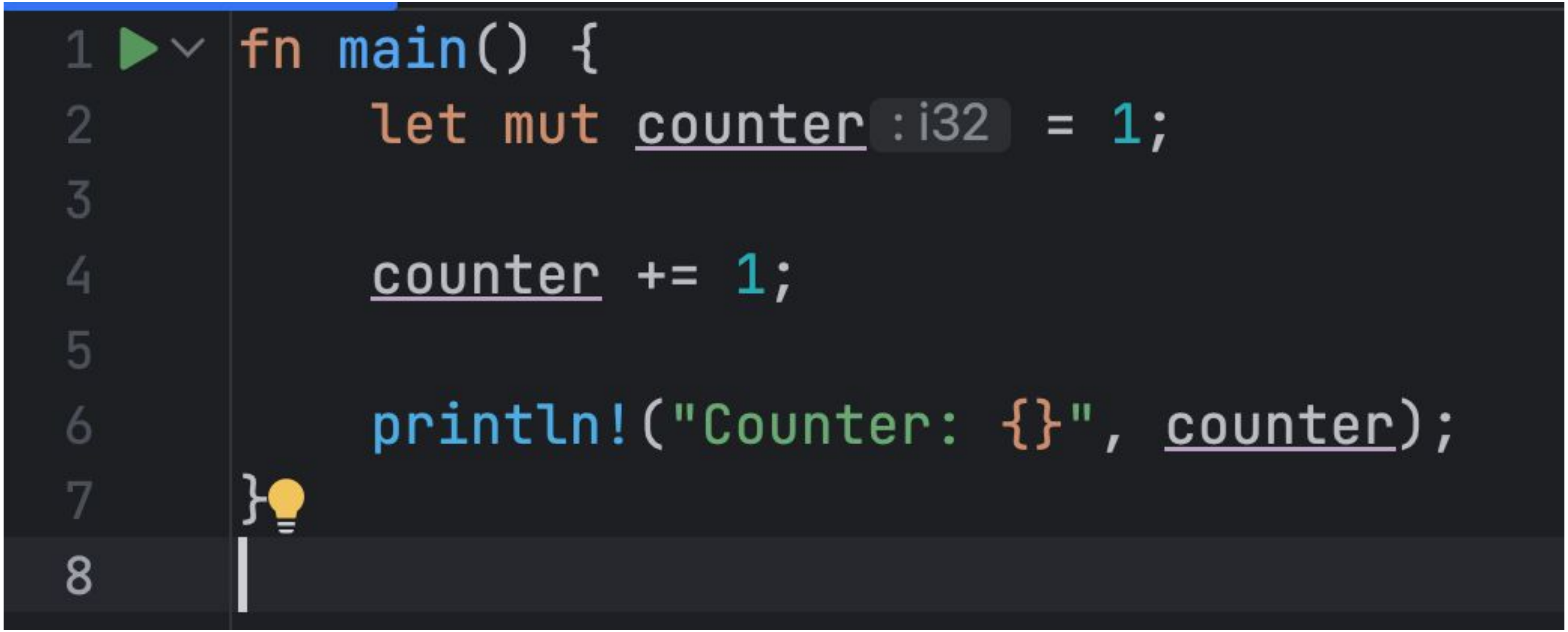


# 1. Immutability by default



```
error[E0384]: cannot assign twice to immutable variable `counter`
 --> src/main.rs:4:5
  |
2 |     let counter = 1;
  |         -------
  |         |
  |         first assignment to `counter`
  |         help: consider making this binding mutable: `mut counter`
3 |
4 |     counter += 1;
  |     ^^^^^^^^^^^^ cannot assign twice to immutable variable
```

# 1. Immutability by default



```rust
fn main() {
    let mut counter: i32 = 1;

    counter += 1;

    println!("Counter: {}", counter);
}

```

# 1. Immutability by default

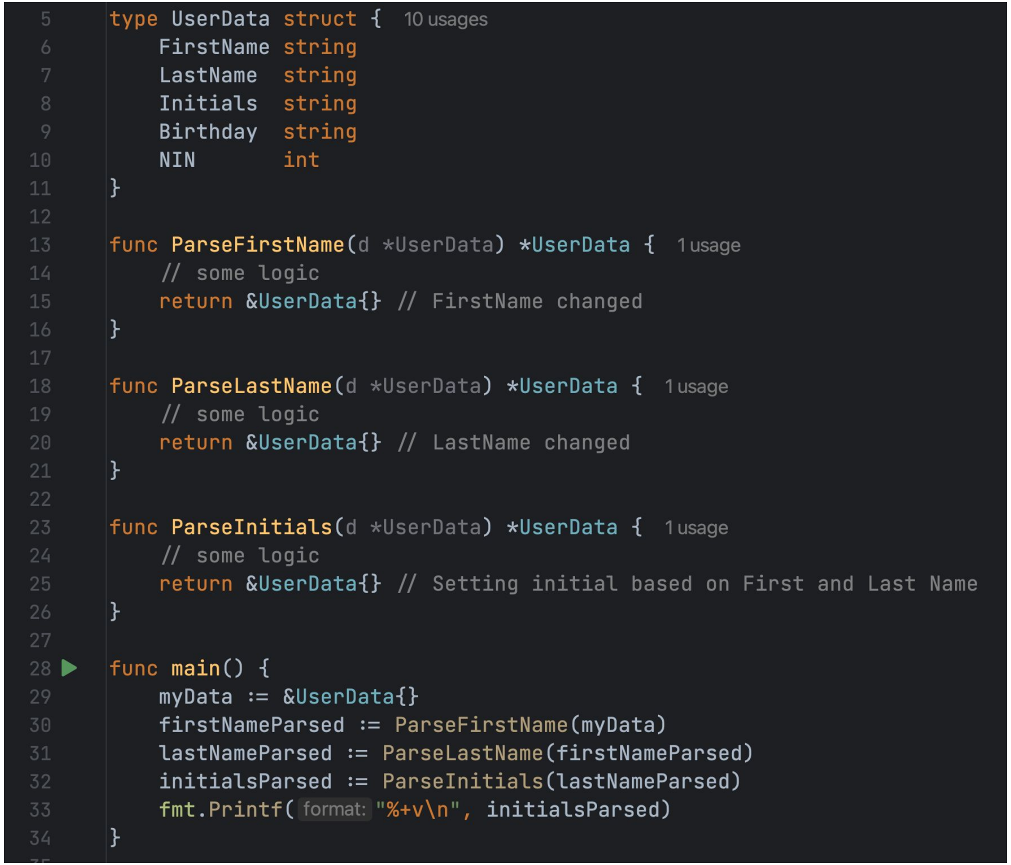
# 1. Immutability by default

https://tiny.pl/cxkvv - MIT OpenLearning (from Java course) (video version)
https://web.mit.edu/6.005/www/fa15/classes/09-immutability/ - text version, short link: https://tiny.pl/cxkbn

https://homes.cs.washington.edu/~mernst/pubs/immutability-aliasing-2013-lncs7850-abstract.html - "Immutability" by Alex Potanin, Johan Östlund, Yoav Zibin, and Michael D. Ernst. In Aliasing in Object-Oriented Programming - short link: https://tiny.pl/cxkb4

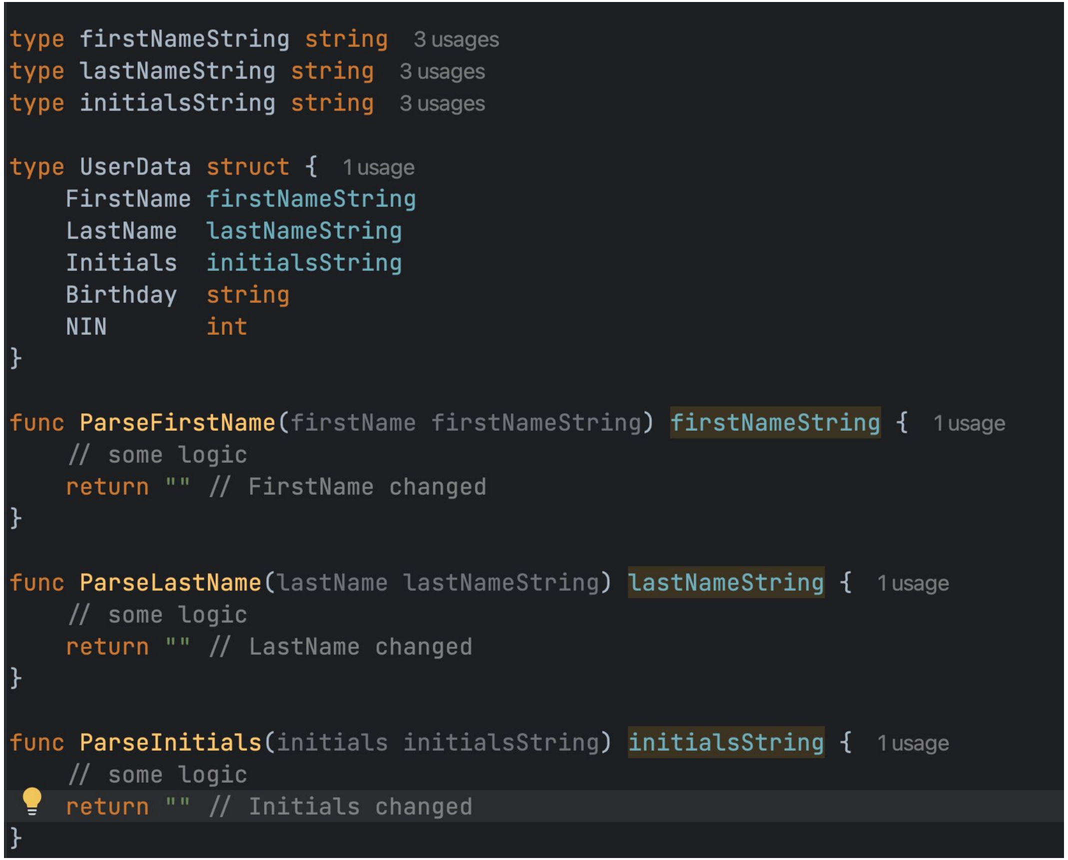
Programming Safety Tips: Why You Should Use Immutable Objects or How to create programs with bugs that can never be found or fixed - Charles W. Kann, Gettysburg College - short link: https://tiny.pl/cxkbl

# 1. Immutability by default

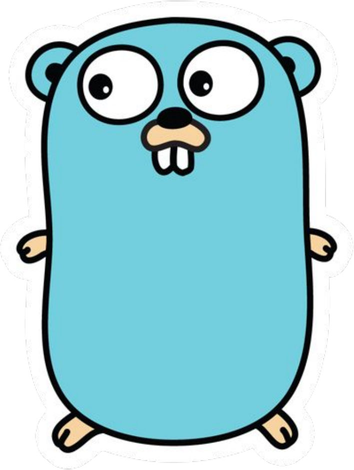


```
type UserData struct {  10 usages
    FirstName string
    LastName  string
    Initials  string
    Birthday  string
    NIN       int
}

func ParseFirstName(d *UserData) *UserData {  1 usage
    // some logic
    return &UserData{} // FirstName changed
}

func ParseLastName(d *UserData) *UserData {  1 usage
    // some logic
    return &UserData{} // LastName changed
}

func ParseInitials(d *UserData) *UserData {  1 usage
    // some logic
    return &UserData{} // Setting initial based on First and Last Name
}

func main() {
    myData := &UserData{}
    firstNameParsed := ParseFirstName(myData)
    lastNameParsed := ParseLastName(firstNameParsed)
    initialsParsed := ParseInitials(lastNameParsed)
    fmt.Printf( format: "%+v\n", initialsParsed)
}
```

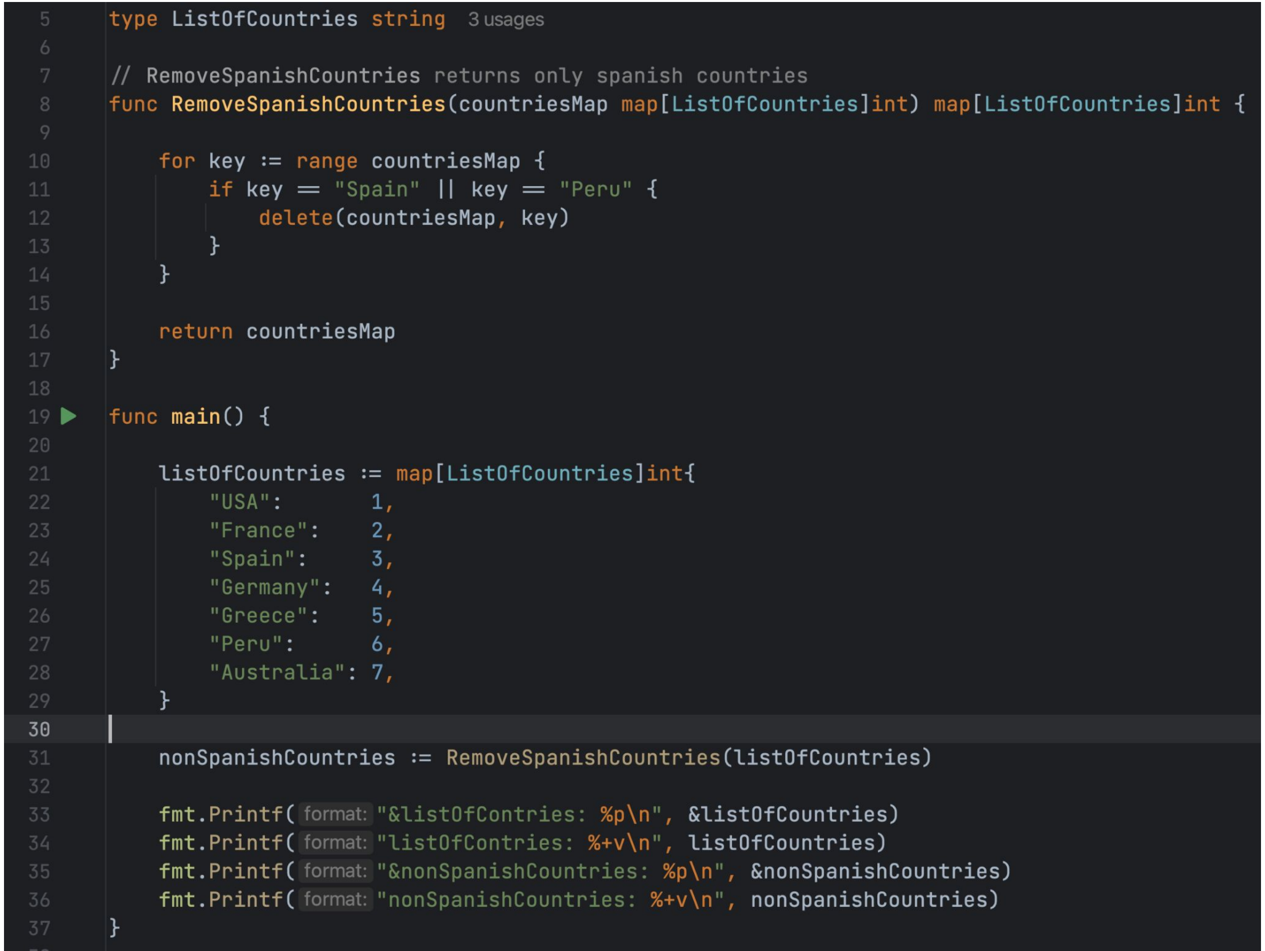
# 1. Immutability by default



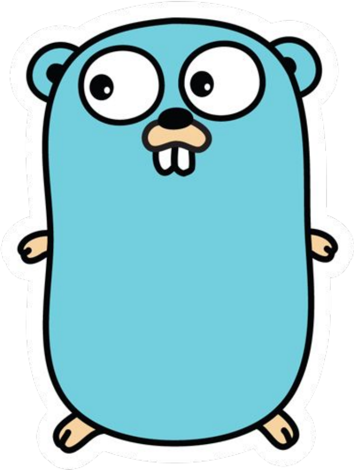
```
type firstNameString string  3 usages
type lastNameString string  3 usages
type initialsString string  3 usages

type UserData struct {  1 usage
    FirstName firstNameString
    LastName  lastNameString
    Initials  initialsString
    Birthday  string
    NIN       int
}

func ParseFirstName(firstName firstNameString) firstNameString {  1 usage
    // some logic
    return "" // FirstName changed
}

func ParseLastName(lastName lastNameString) lastNameString {  1 usage
    // some logic
    return "" // LastName changed
}

func ParseInitials(initials initialsString) initialsString {  1 usage
    // some logic
    return "" // Initials changed
}
```



# 1. Immutability by default (map problem)

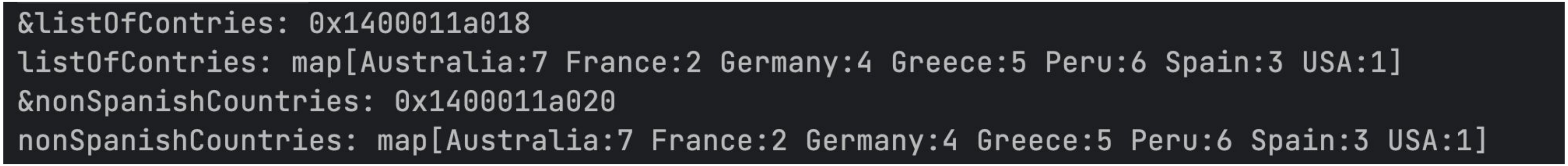


```go
type ListOfCountries string  3 usages

// RemoveSpanishCountries returns only spanish countries
func RemoveSpanishCountries(countriesMap map[ListOfCountries]int) map[ListOfCountries]int {

    for key := range countriesMap {
        if key == "Spain" || key == "Peru" {
            delete(countriesMap, key)
        }
    }

    return countriesMap
}

func main() {

    listOfCountries := map[ListOfCountries]int{
        "USA":       1,
        "France":    2,
        "Spain":     3,
        "Germany":   4,
        "Greece":    5,
        "Peru":      6,
        "Australia": 7,
    }

    nonSpanishCountries := RemoveSpanishCountries(listOfCountries)

    fmt.Printf( format: "&listOfContries: %p\n", &listOfCountries)
    fmt.Printf( format: "listOfContries: %+v\n", listOfCountries)
    fmt.Printf( format: "&nonSpanishCountries: %p\n", &nonSpanishCountries)
    fmt.Printf( format: "nonSpanishCountries: %+v\n", nonSpanishCountries)
}
```



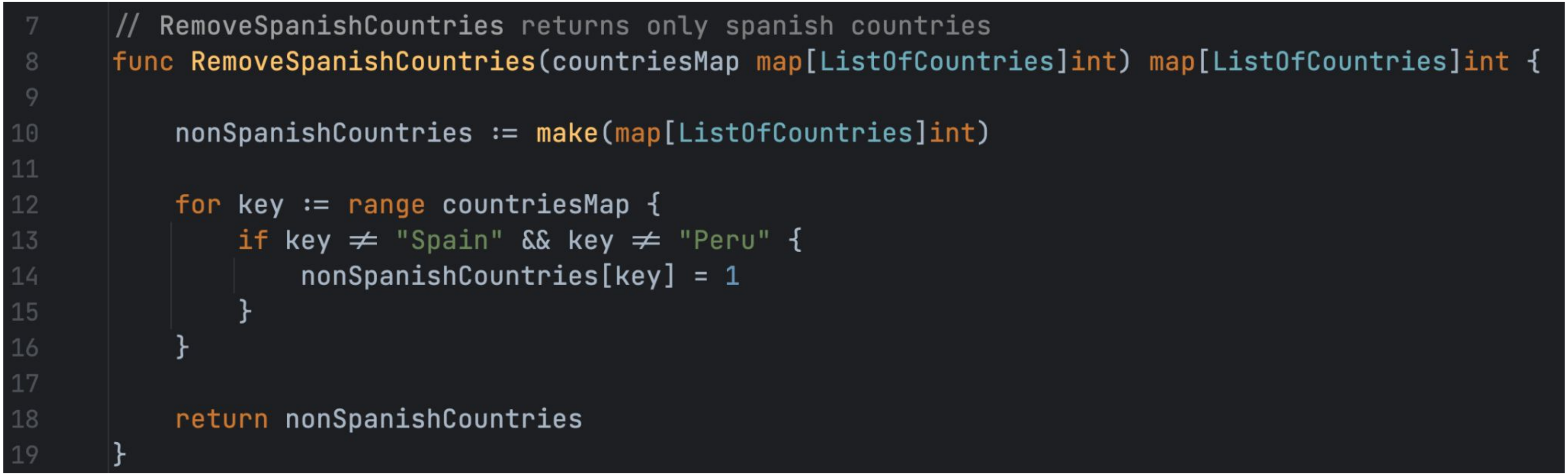
Rusty Gophers

# 1. Immutability by default (map problem)



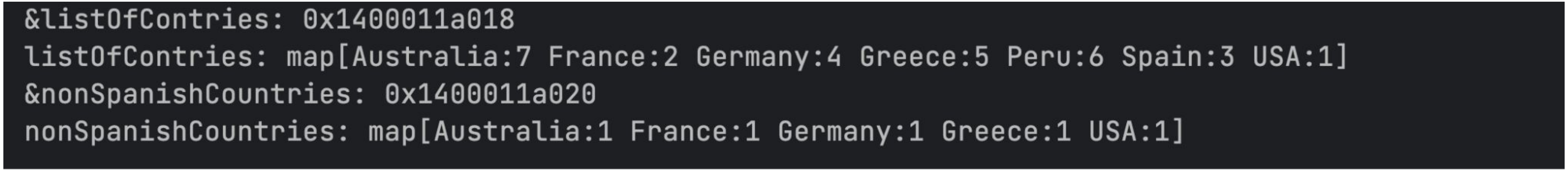
```
&listOfContries: 0x1400011a018
listOfContries: map[Australia:7 France:2 Germany:4 Greece:5 Peru:6 Spain:3 USA:1]
&nonSpanishCountries: 0x1400011a020
nonSpanishCountries: map[Australia:7 France:2 Germany:4 Greece:5 Peru:6 Spain:3 USA:1]
```

# 1. Immutability by default (map problem)



```go
// RemoveSpanishCountries returns only spanish countries
func RemoveSpanishCountries(countriesMap map[ListOfCountries]int) map[ListOfCountries]int {

    nonSpanishCountries := make(map[ListOfCountries]int)

    for key := range countriesMap {
        if key != "Spain" && key != "Peru" {
            nonSpanishCountries[key] = 1
        }
    }

    return nonSpanishCountries
}
```

# 1. Immutability by default (map problem)



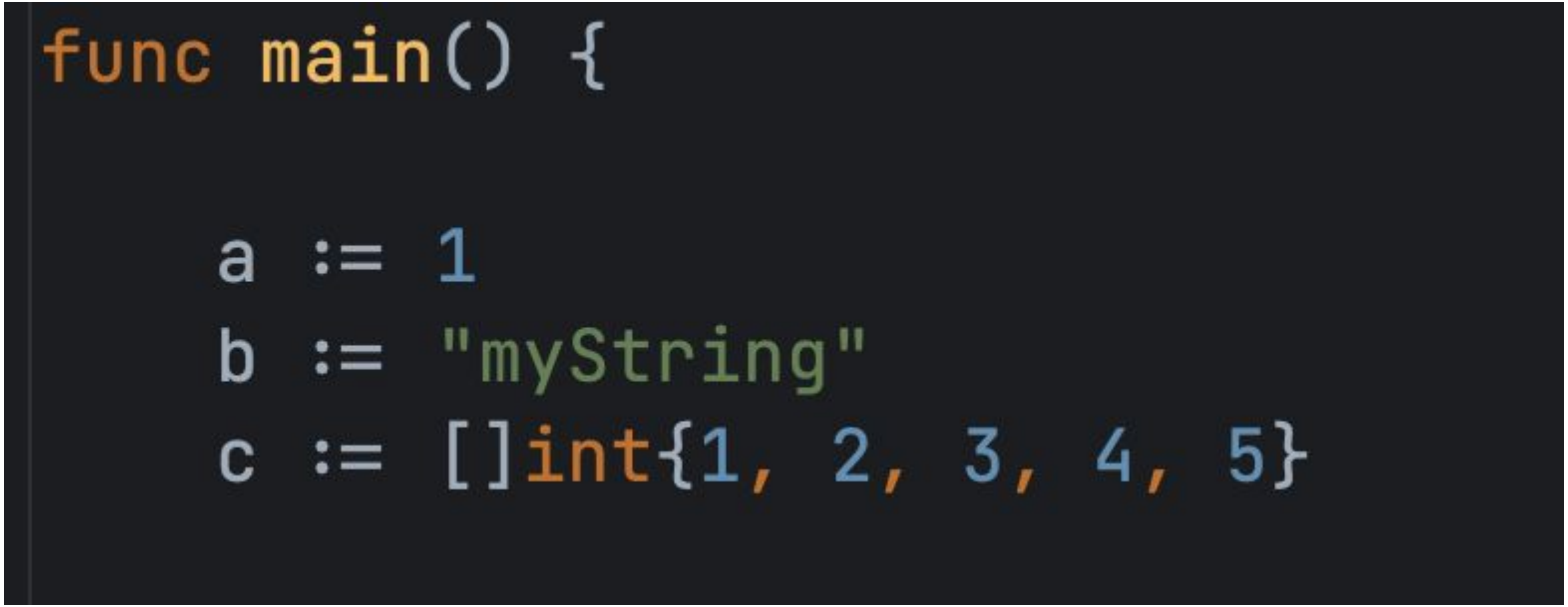
```
&listOfContries: 0x1400011a018
listOfContries: map[Australia:7 France:2 Germany:4 Greece:5 Peru:6 Spain:3 USA:1]
&nonSpanishCountries: 0x1400011a020
nonSpanishCountries: map[Australia:1 France:1 Germany:1 Greece:1 USA:1]
```

# 1. Immutability by default

Use `immutability` as often as you can

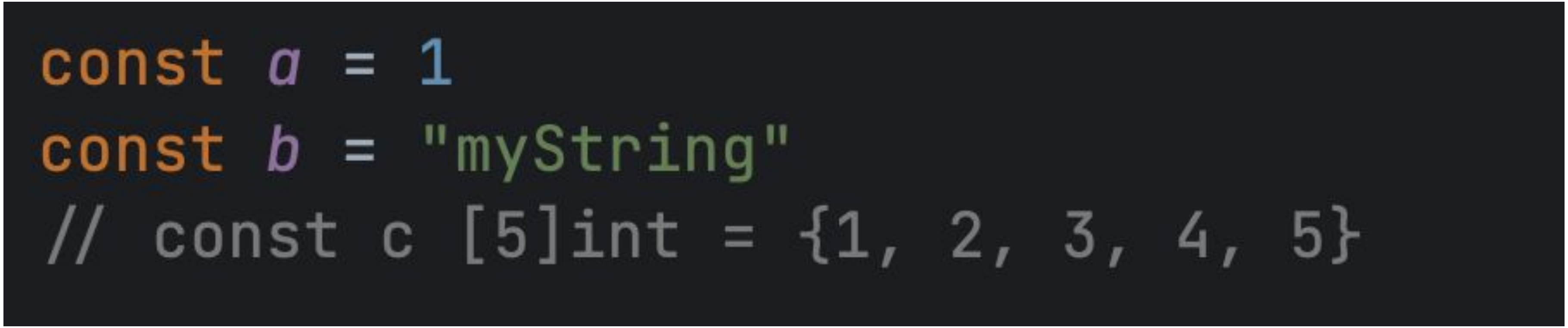
Use `const` as often as you can

# 1. Immutability by default



```
func main() {

    a := 1
    b := "myString"
    c := []int{1, 2, 3, 4, 5}
```

# 1. Immutability by default



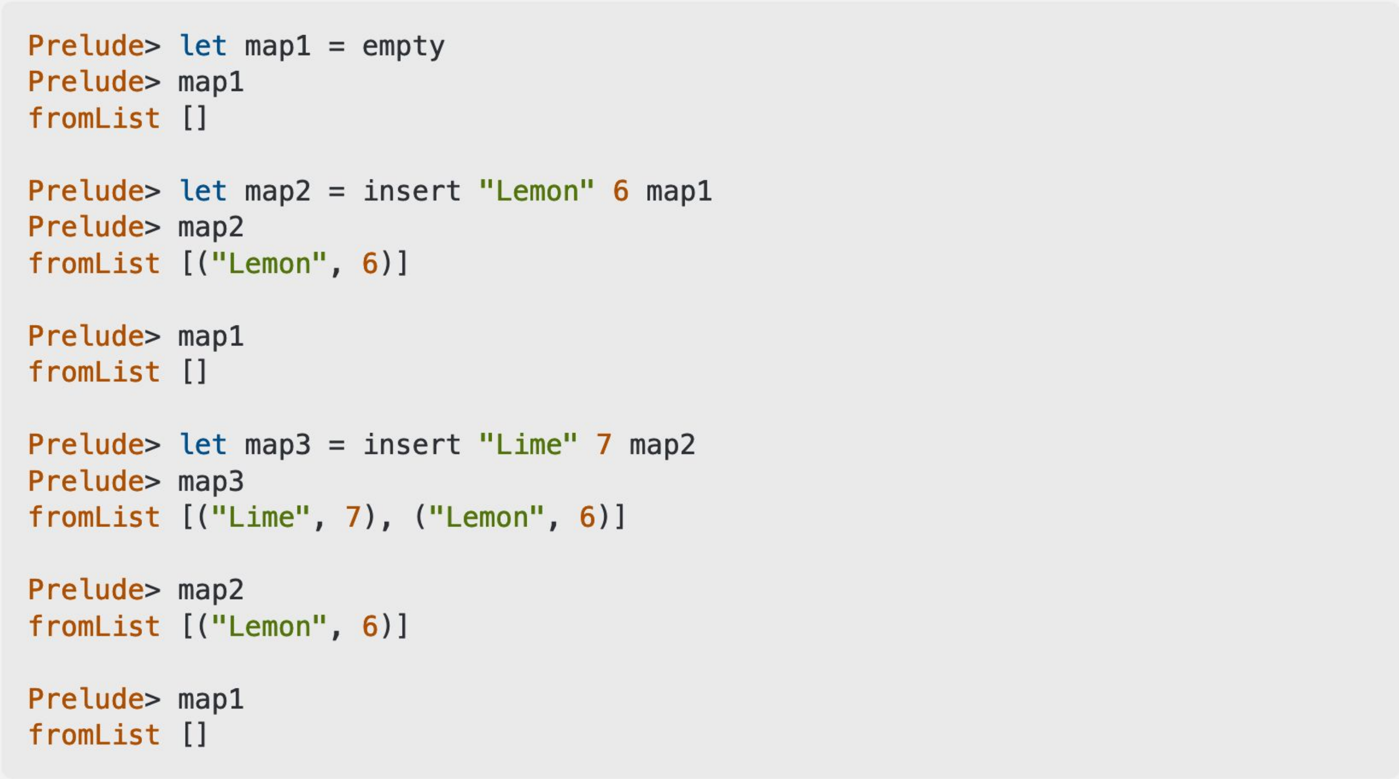
```
const a = 1
const b = "myString"
// const c [5]int = {1, 2, 3, 4, 5}
```

# 1. Immutability by default

slice?

# 1. Immutability by default

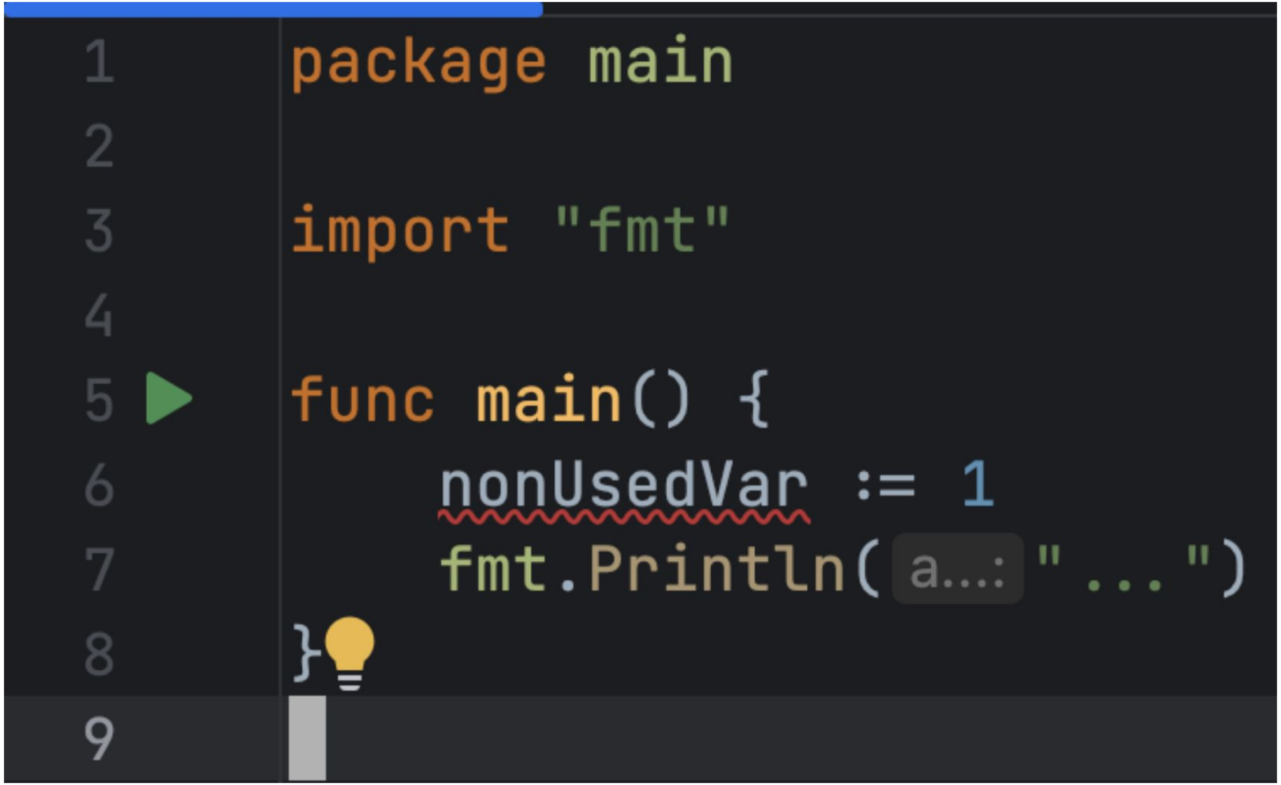
# 1. Immutability by default



```
Prelude> let map1 = empty
Prelude> map1
fromList []

Prelude> let map2 = insert "Lemon" 6 map1
Prelude> map2
fromList [("Lemon", 6)]

Prelude> map1
fromList []

Prelude> let map3 = insert "Lime" 7 map2
Prelude> map3
fromList [("Lime", 7), ("Lemon", 6)]

Prelude> map2
fromList [("Lemon", 6)]

Prelude> map1
fromList []
```

https://stackoverflow.com/a/49895382/9261336

# 2. Uninitialized variables

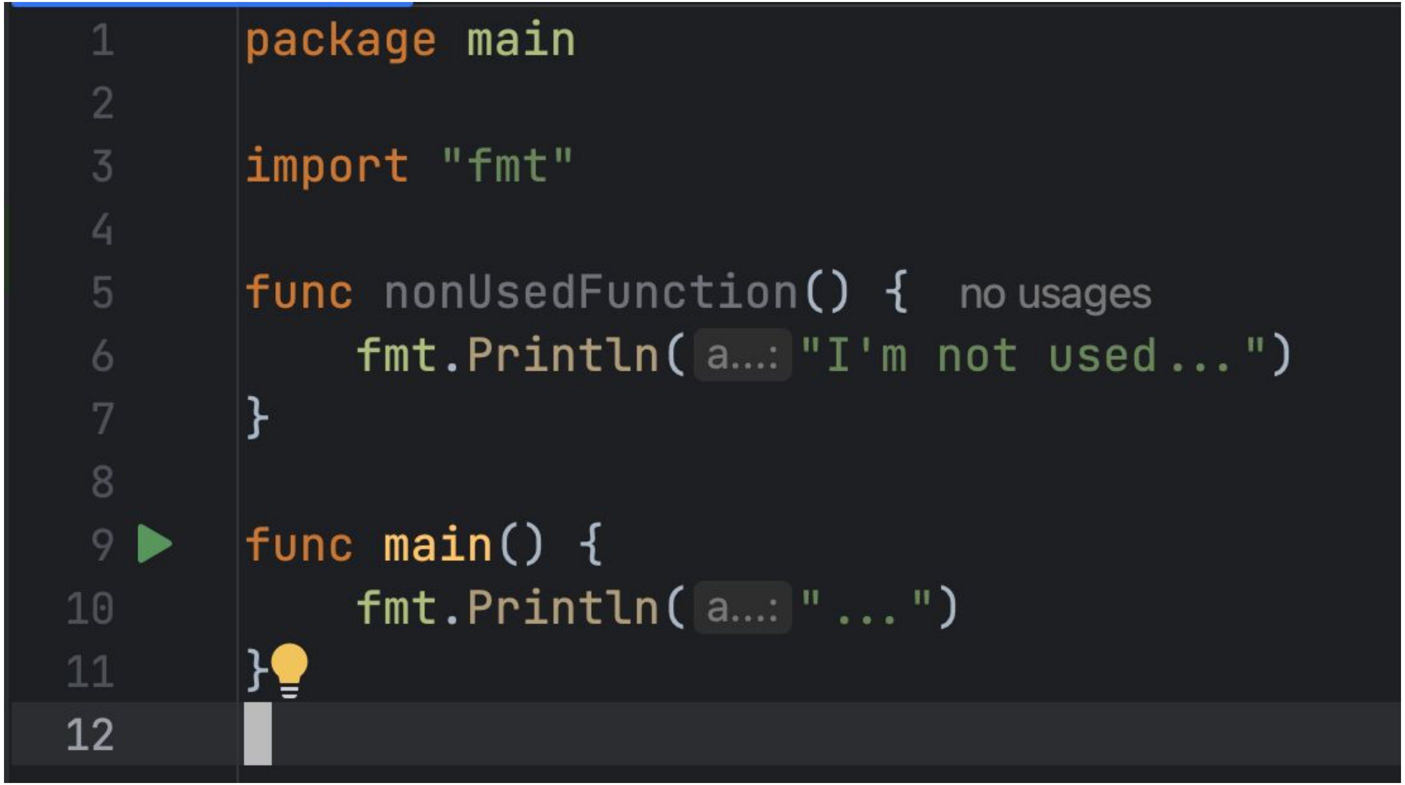


```go
package main

import "fmt"

func main() {
	nonUsedVar := 1
	fmt.Println(a...: "...")
}
```

https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.710.2018&rep=rep1&type=pdf -

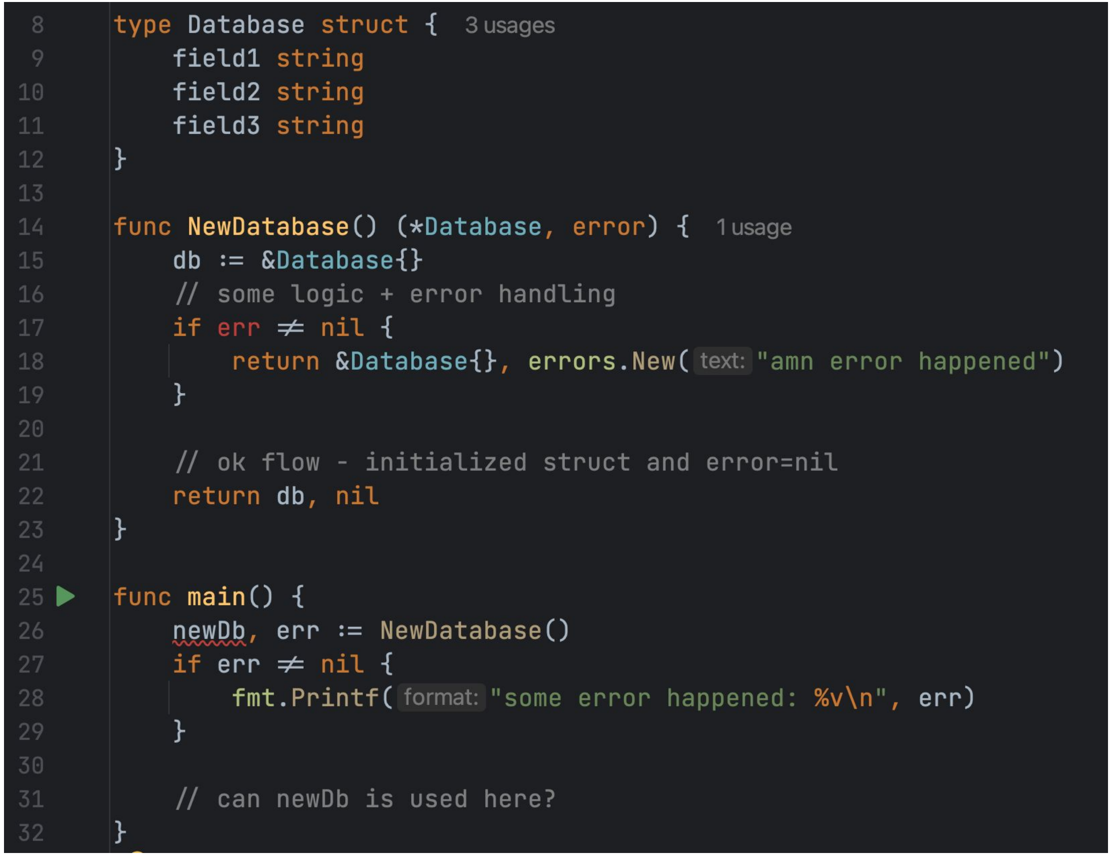
How Much Does Unused Code Matter for Maintenance

# 2. Uninitialized variables

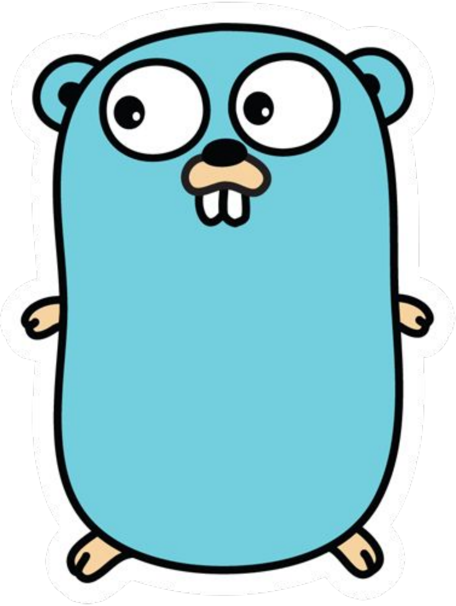


```go
package main

import "fmt"

func nonUsedFunction() {  no usages
	fmt.Println(a...: "I'm not used...")
}

func main() {
	fmt.Println(a...: "...")
}

```

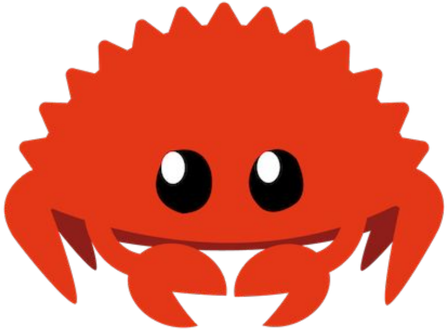
# 2. Uninitialized variables

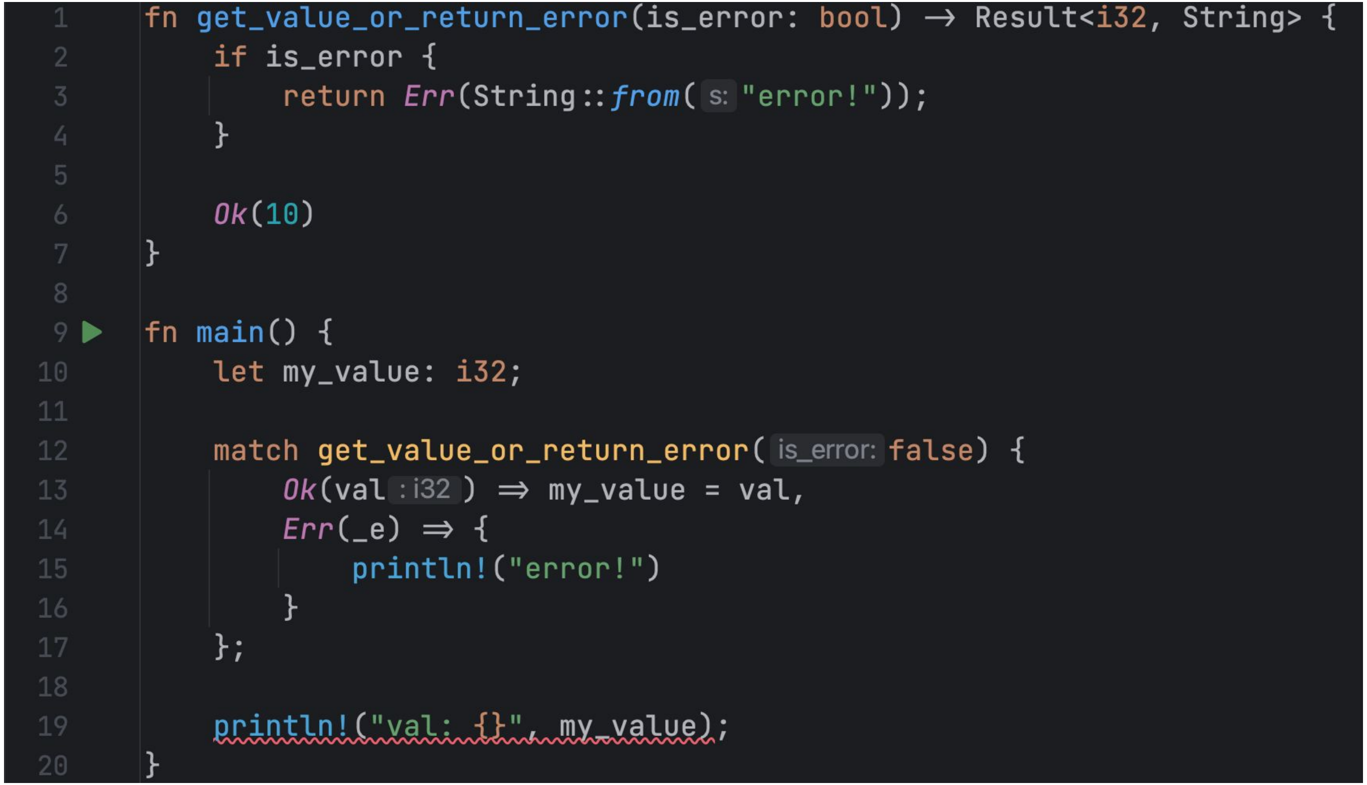


```
type Database struct {  3 usages
    field1 string
    field2 string
    field3 string
}

func NewDatabase() (*Database, error) {  1 usage
    db := &Database{}
    // some logic + error handling
    if err != nil {
        return &Database{}, errors.New( text: "amn error happened")
    }

    // ok flow - initialized struct and error=nil
    return db, nil
}

func main() {
    newDb, err := NewDatabase()
    if err != nil {
        fmt.Printf( format: "some error happened: %v\n", err)
    }

    // can newDb is used here?
}
```



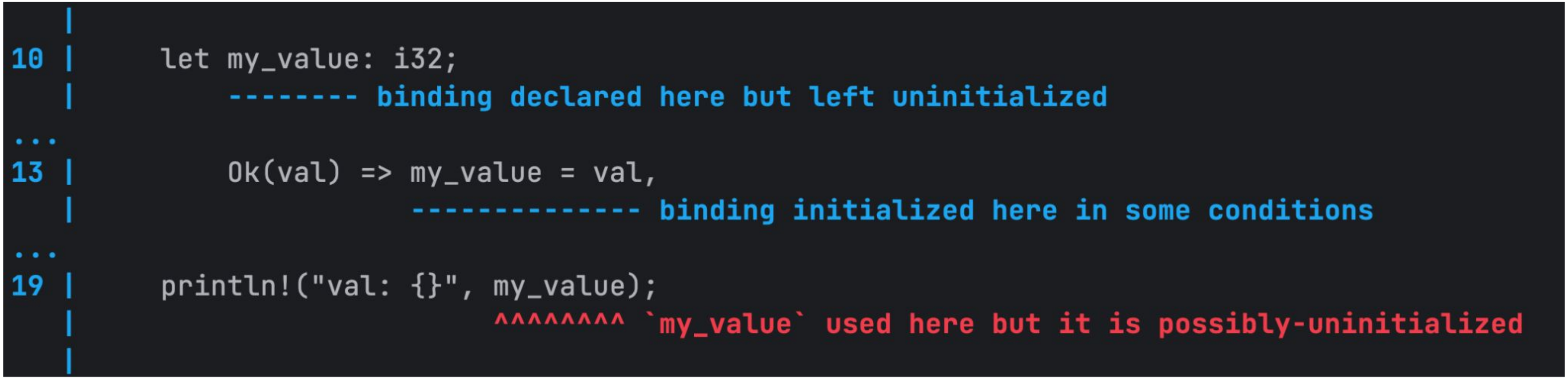
# 2. Uninitialized variables





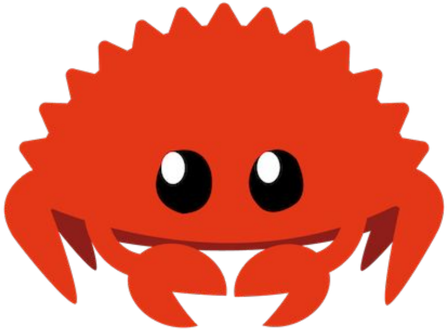
```rust
fn get_value_or_return_error(is_error: bool) -> Result<i32, String> {
    if is_error {
        return Err(String::from( s: "error!"));
    }

    Ok(10)
}

fn main() {
    let my_value: i32;

    match get_value_or_return_error( is_error: false) {
        Ok(val : i32 ) => my_value = val,
        Err(_e) => {
            println!("error!")
        }
    };

    println!("val: {}", my_value);
}
```

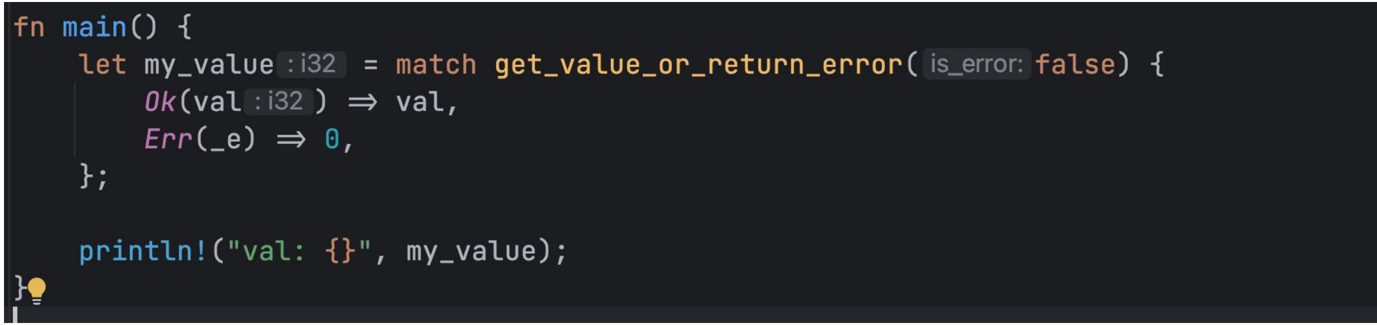
# 2. Uninitialized variables



```
   |
10 |     let my_value: i32;
   |         -------- binding declared here but left uninitialized
...
13 |         Ok(val) => my_value = val,
   |                    -------------- binding initialized here in some conditions
...
19 |     println!("val: {}", my_value);
   |                         ^^^^^^^^ `my_value` used here but it is possibly-uninitialized
   |
```

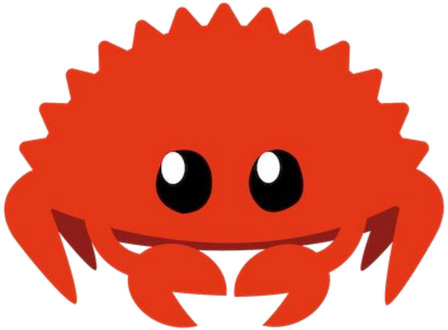
# 2. Uninitialized variables

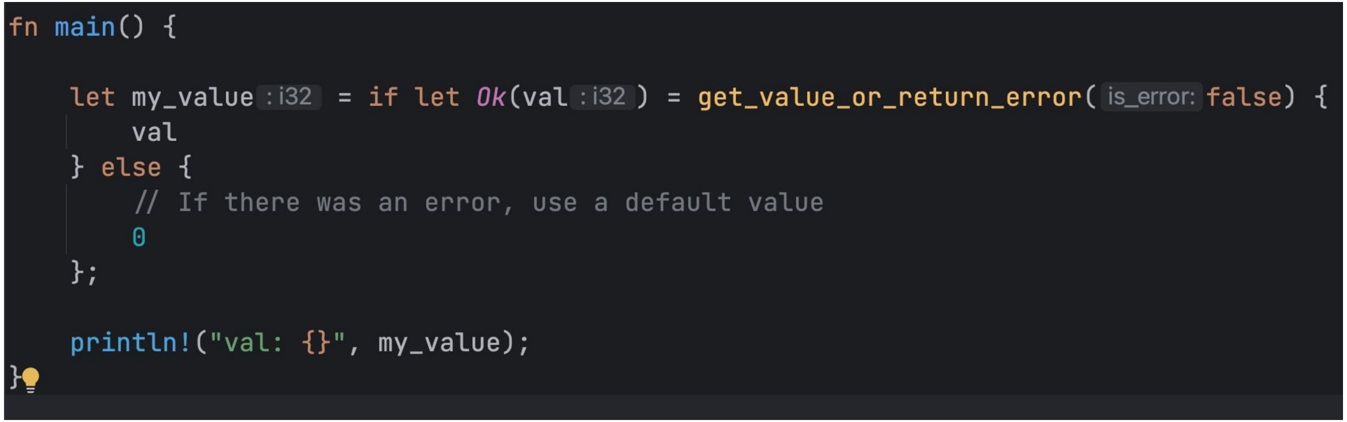




```
fn main() {
    let my_value: i32 = match get_value_or_return_error(is_error: false) {
        Ok(val: i32) => val,
        Err(_e) => 0,
    };

    println!("val: {}", my_value);
}
```

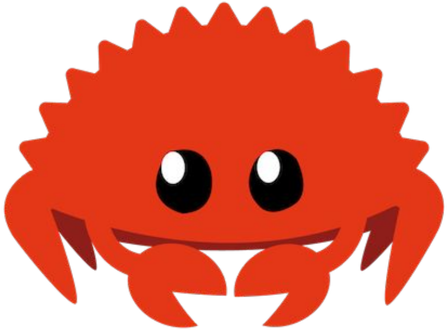
# 2. Uninitialized variables

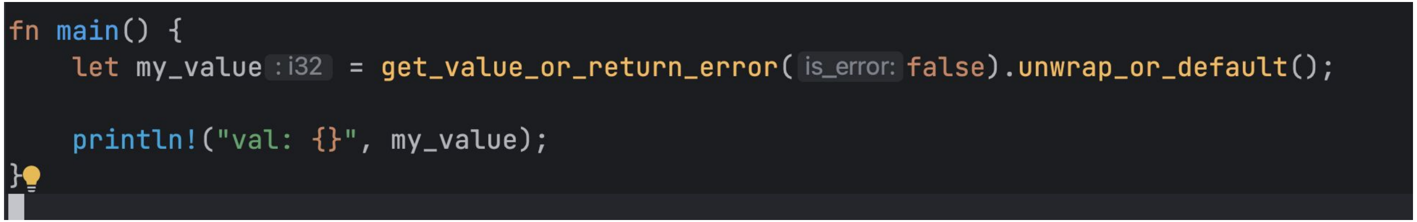




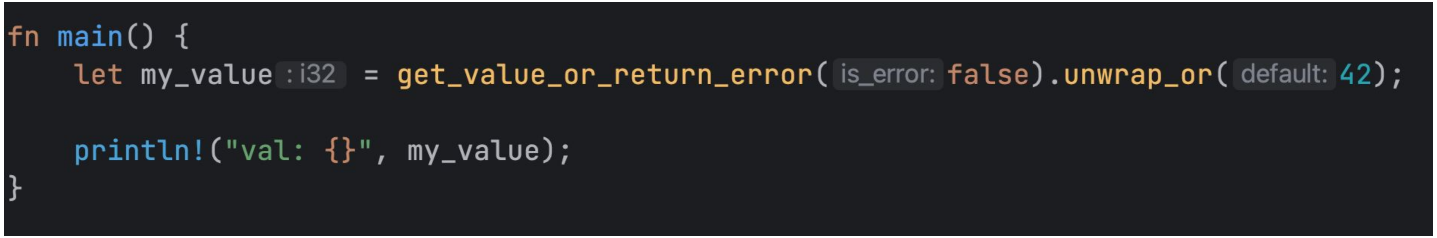
```
fn main() {

    let my_value: i32 = if let Ok(val: i32) = get_value_or_return_error(is_error: false) {
        val
    } else {
        // If there was an error, use a default value
        0
    };

    println!("val: {}", my_value);
}
```

# 2. Uninitialized variables



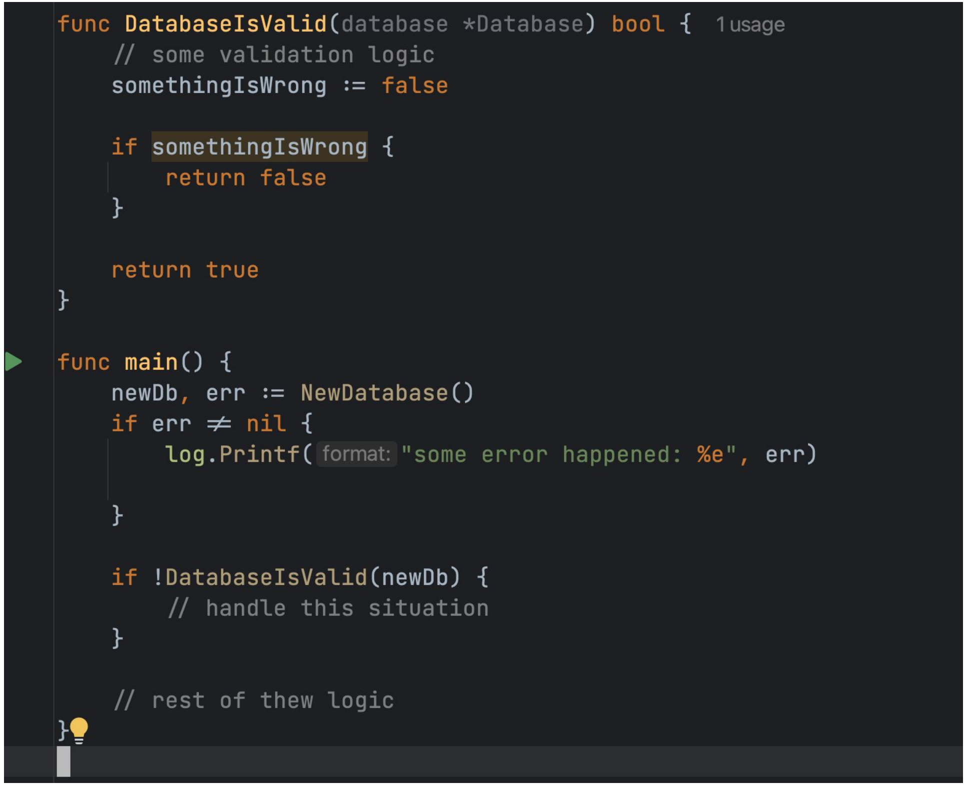


```
fn main() {
    let my_value : i32 = get_value_or_return_error( is_error: false).unwrap_or_default();

    println!("val: {}", my_value);
}
```

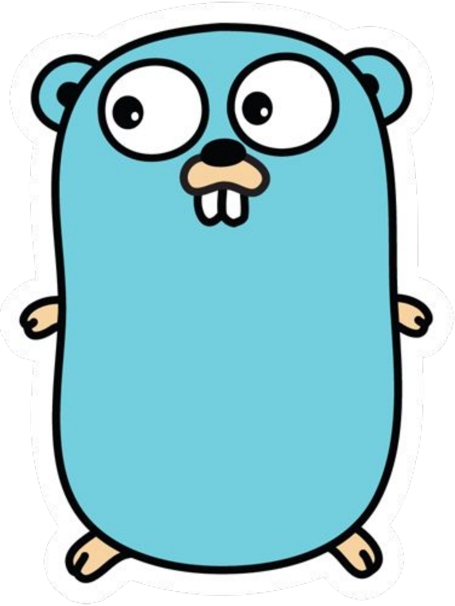


```
fn main() {
    let my_value : i32 = get_value_or_return_error( is_error: false).unwrap_or( default: 42);

    println!("val: {}", my_value);
}
```

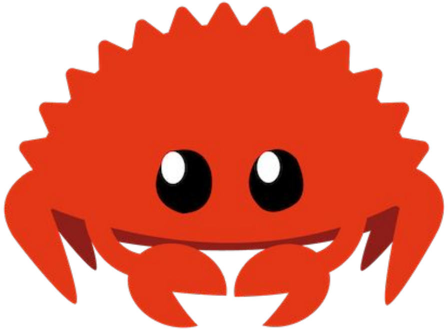
# 2. Uninitialized variables

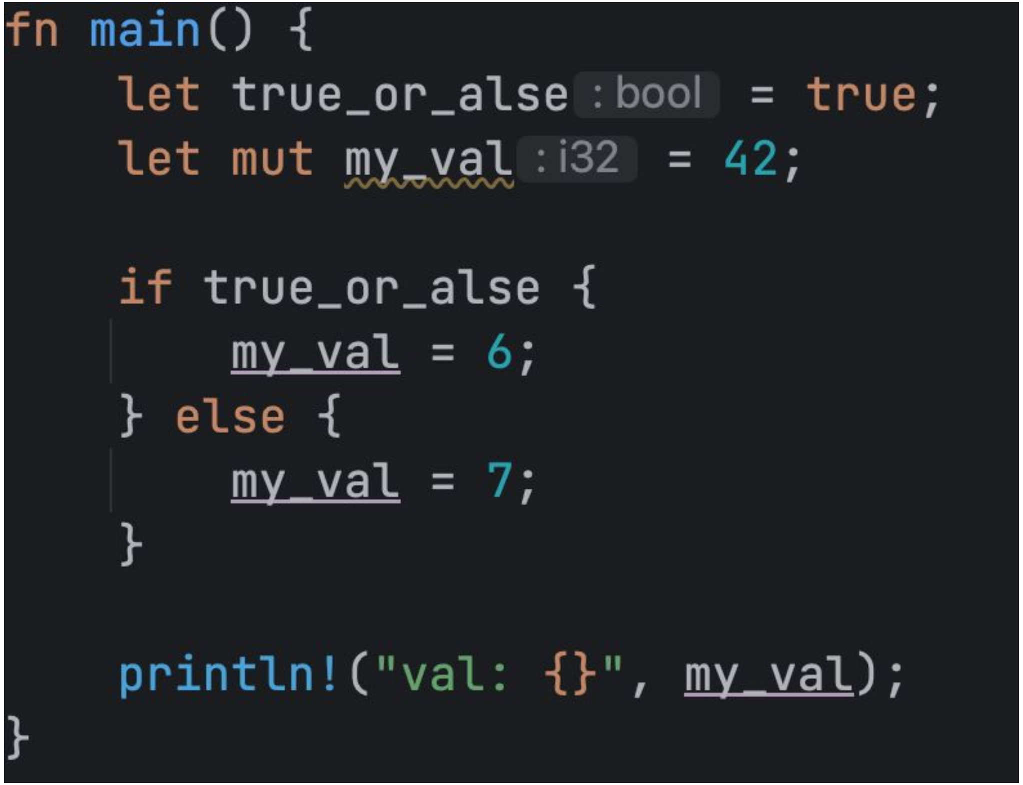


```
func DatabaseIsValid(database *Database) bool {  1 usage
    // some validation logic
    somethingIsWrong := false

    if somethingIsWrong {
        return false
    }

    return true
}

func main() {
    newDb, err := NewDatabase()
    if err ≠ nil {
        log.Printf( format: "some error happened: %e", err)

    }

    if !DatabaseIsValid(newDb) {
        // handle this situation
    }

    // rest of thew logic
}
```



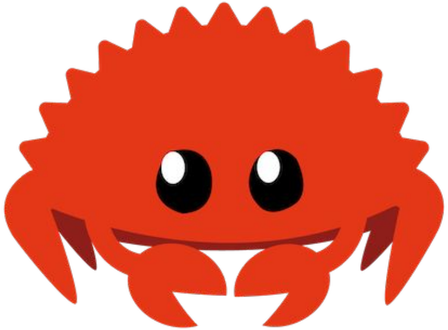
# 2. Uninitialized variables (or used)

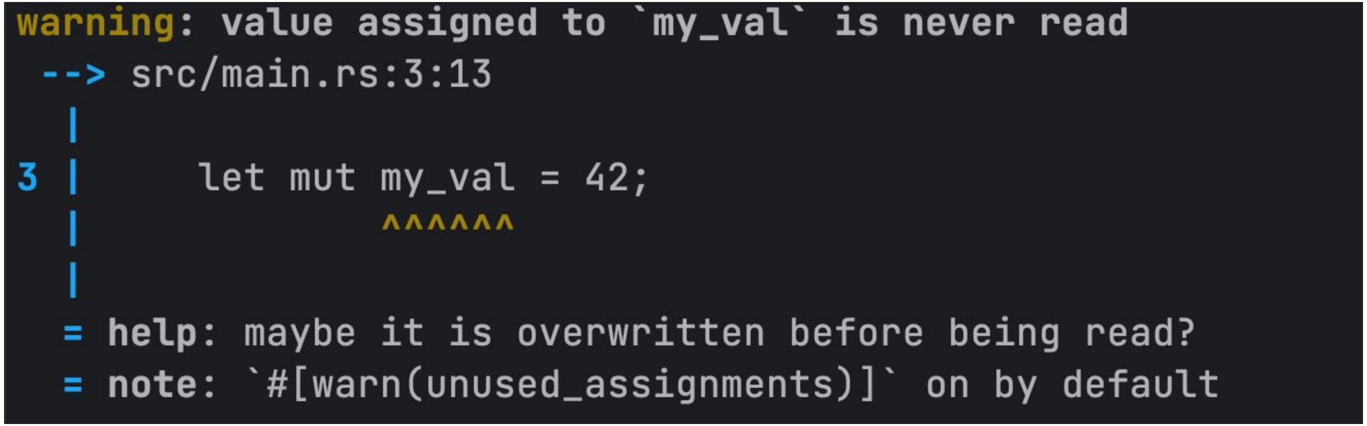




```
fn main() {
    let true_or_alse: bool = true;
    let mut my_val: i32 = 42;

    if true_or_alse {
        my_val = 6;
    } else {
        my_val = 7;
    }

    println!("val: {}", my_val);
}
```

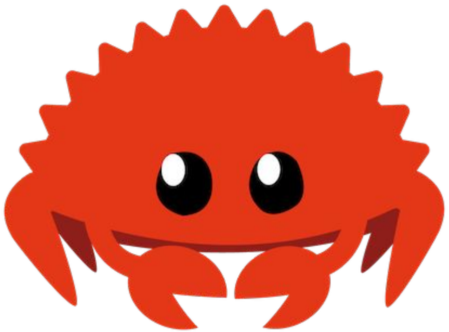
# 2. Uninitialized variables (or used)

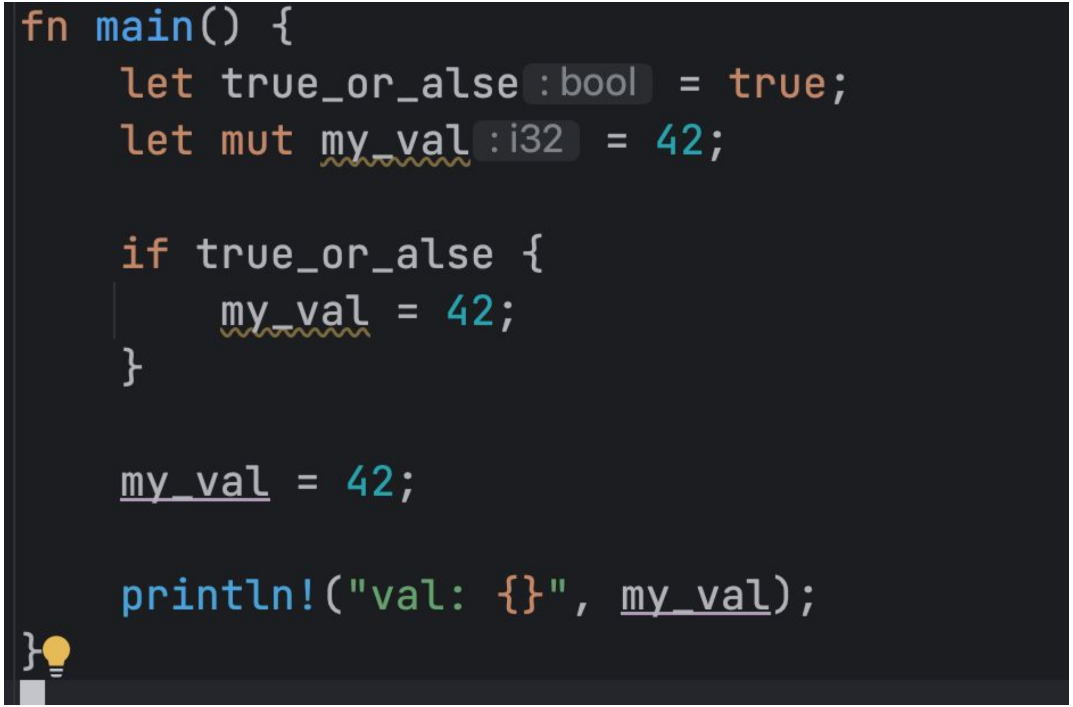




```
warning: value assigned to `my_val` is never read
 --> src/main.rs:3:13
  |
3 |     let mut my_val = 42;
  |             ^^^^^^
  |
  = help: maybe it is overwritten before being read?
  = note: `#[warn(unused_assignments)]` on by default
```

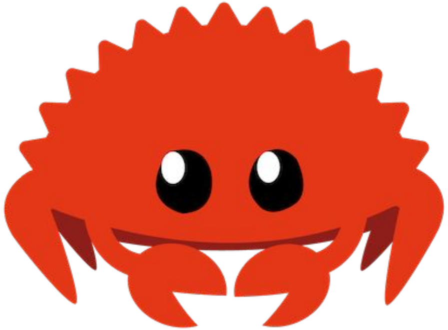
# 2. Uninitialized variables (or used)

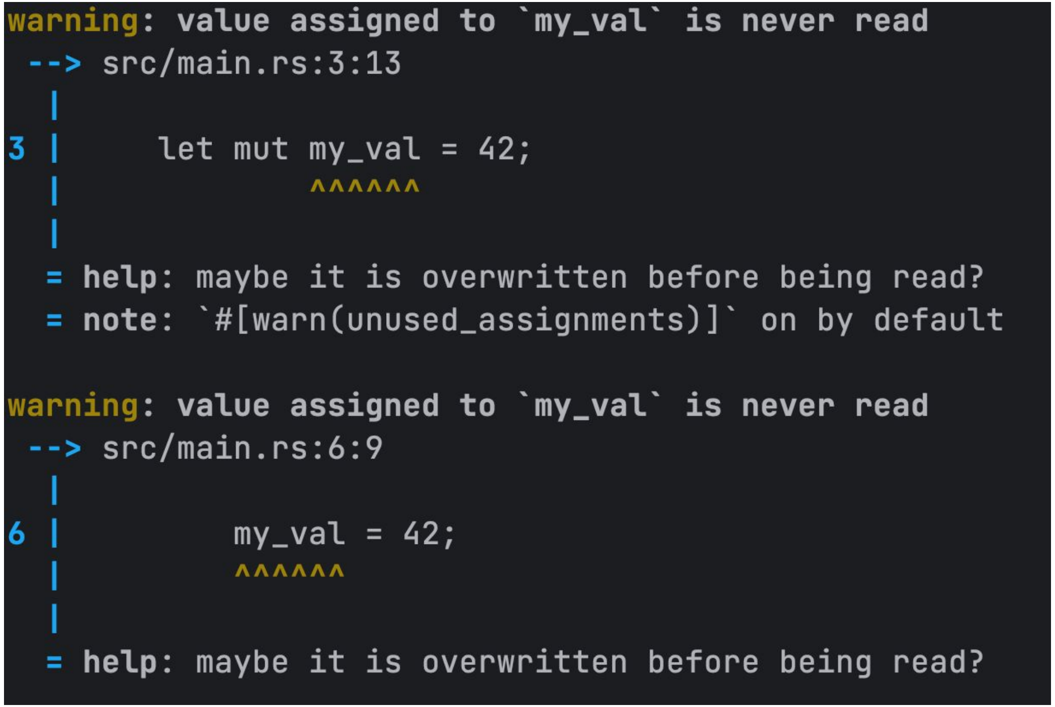




```
fn main() {
    let true_or_alse: bool = true;
    let mut my_val: i32 = 42;

    if true_or_alse {
        my_val = 42;
    }

    my_val = 42;

    println!("val: {}", my_val);
}
```

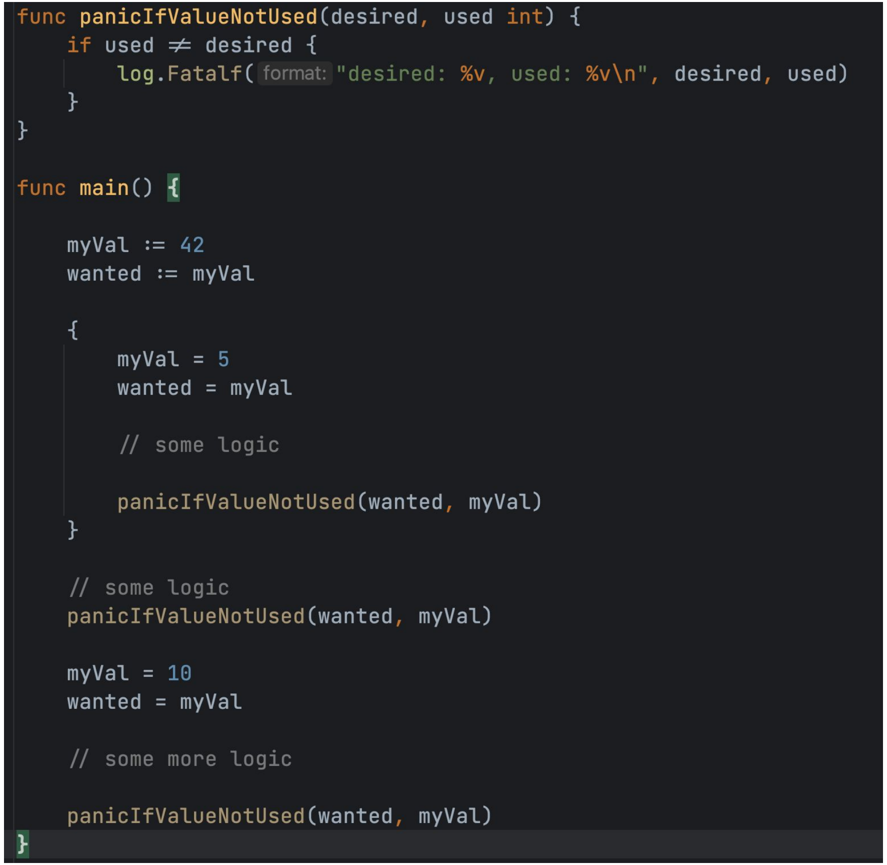
# 2. Uninitialized variables (or used)



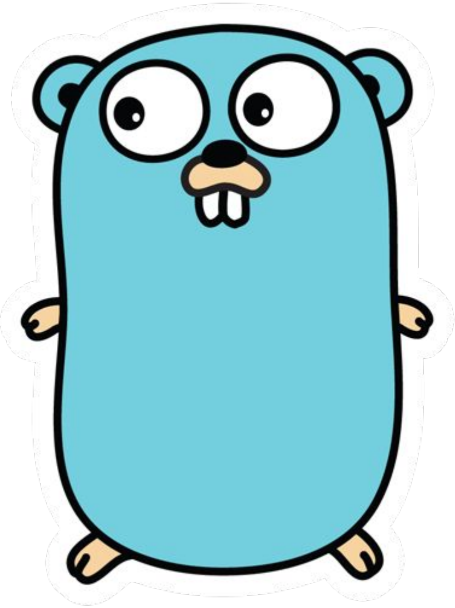


```
warning: value assigned to `my_val` is never read
 --> src/main.rs:3:13
  |
3 |     let mut my_val = 42;
  |             ^^^^^^
  |
  = help: maybe it is overwritten before being read?
  = note: `#[warn(unused_assignments)]` on by default

warning: value assigned to `my_val` is never read
 --> src/main.rs:6:9
  |
6 |         my_val = 42;
  |         ^^^^^^
  |
  = help: maybe it is overwritten before being read?
```

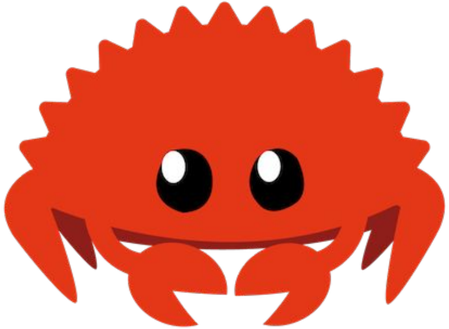
# 2. Uninitialized variables (or used)



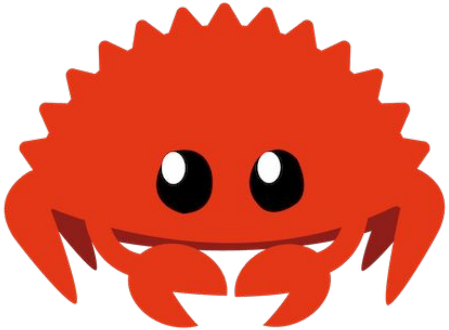
```go
func panicIfValueNotUsed(desired, used int) {
    if used ≠ desired {
        log.Fatalf( format: "desired: %v, used: %v\n", desired, used)
    }
}

func main() {

    myVal := 42
    wanted := myVal

    {
        myVal = 5
        wanted = myVal

        // some logic

        panicIfValueNotUsed(wanted, myVal)
    }

    // some logic
    panicIfValueNotUsed(wanted, myVal)

    myVal = 10
    wanted = myVal

    // some more logic

    panicIfValueNotUsed(wanted, myVal)
}
```

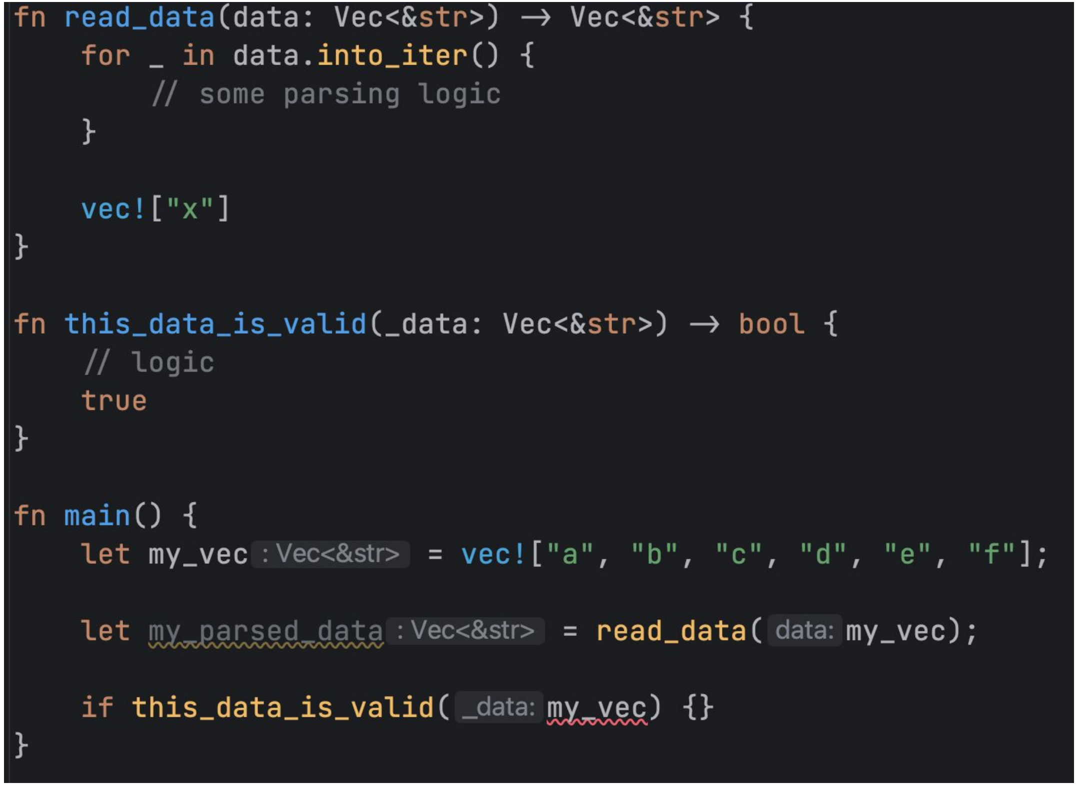


# 3. Consuming variables



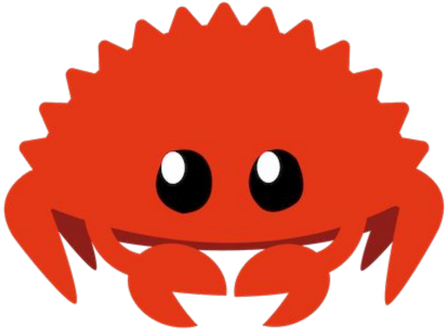
# 3. Consuming variables

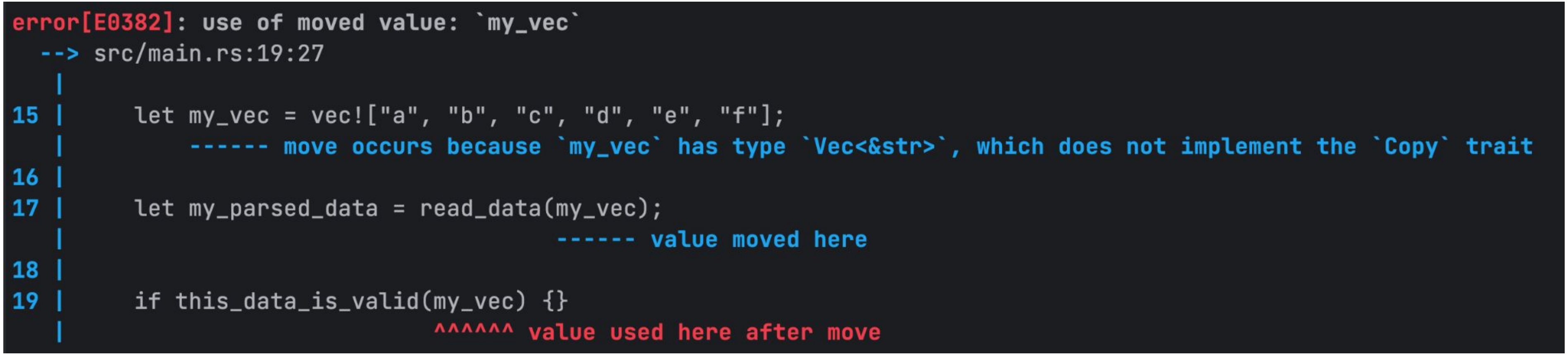




```
fn read_data(data: Vec<&str>) -> Vec<&str> {
    for _ in data.into_iter() {
        // some parsing logic
    }

    vec!["x"]
}

fn this_data_is_valid(_data: Vec<&str>) -> bool {
    // logic
    true
}

fn main() {
    let my_vec : Vec<&str> = vec!["a", "b", "c", "d", "e", "f"];

    let my_parsed_data : Vec<&str> = read_data( data: my_vec);

    if this_data_is_valid( _data: my_vec) {}
}
```

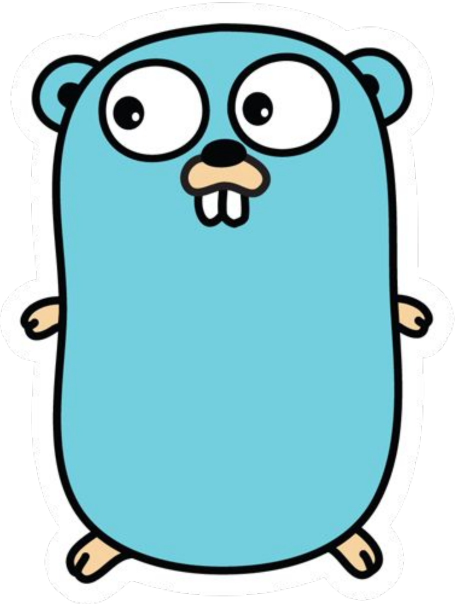
# 3. Consuming variables





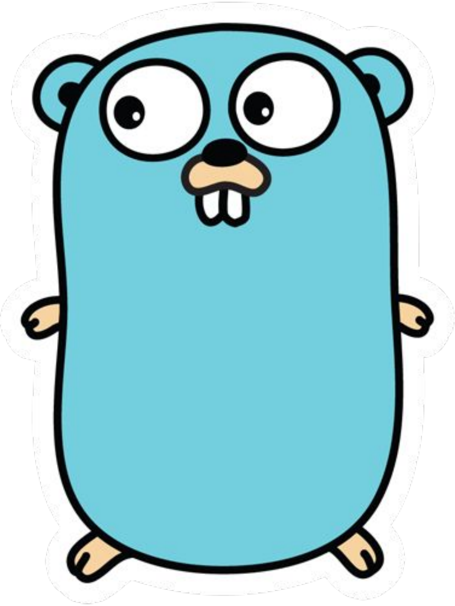
```
error[E0382]: use of moved value: `my_vec`
  --> src/main.rs:19:27
   |
15 |     let my_vec = vec!["a", "b", "c", "d", "e", "f"];
   |         ------ move occurs because `my_vec` has type `Vec<&str>`, which does not implement the `Copy` trait
16 |
17 |     let my_parsed_data = read_data(my_vec);
   |                                    ------ value moved here
18 |
19 |     if this_data_is_valid(my_vec) {}
   |                           ^^^^^^ value used here after move
```

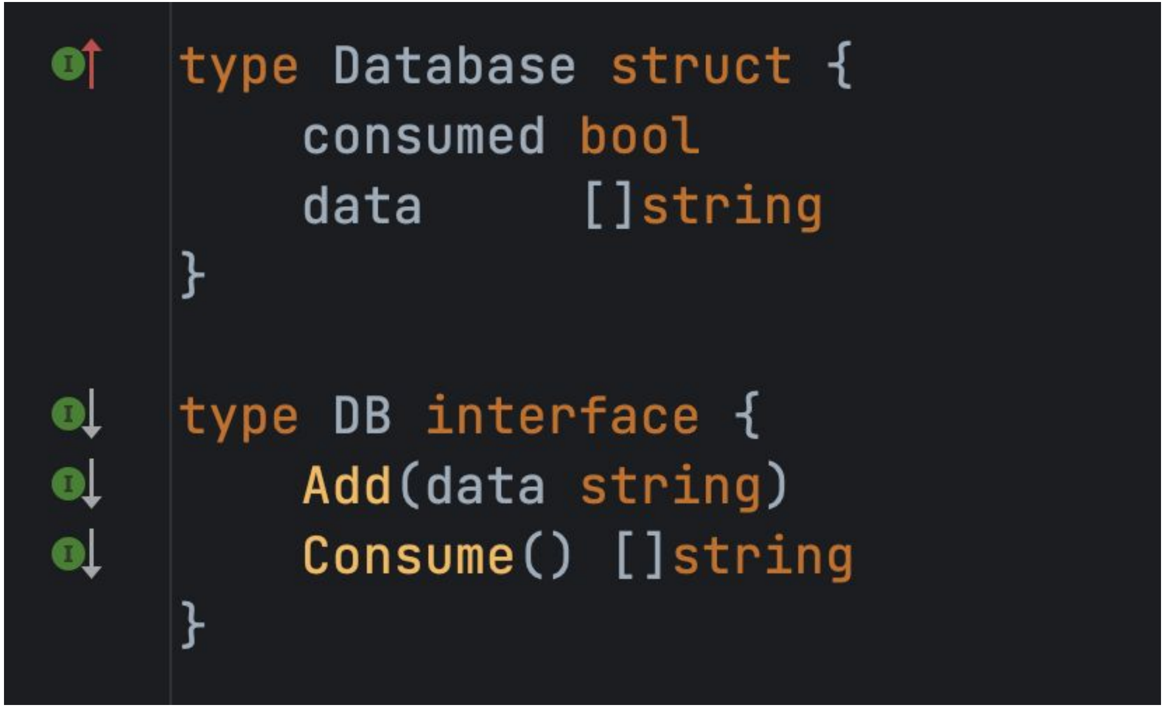
# 3. Consuming variables



?

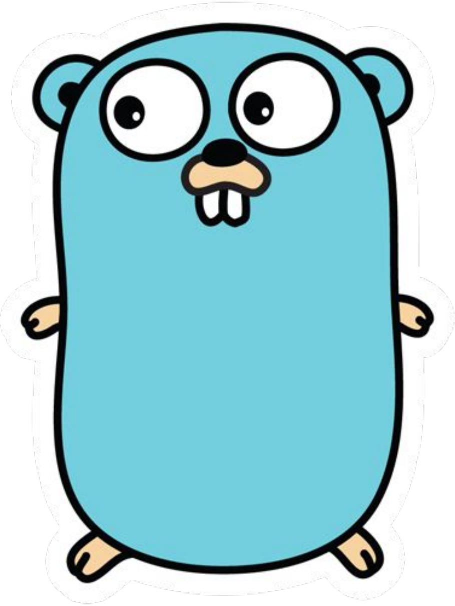
# 3. Consuming variables

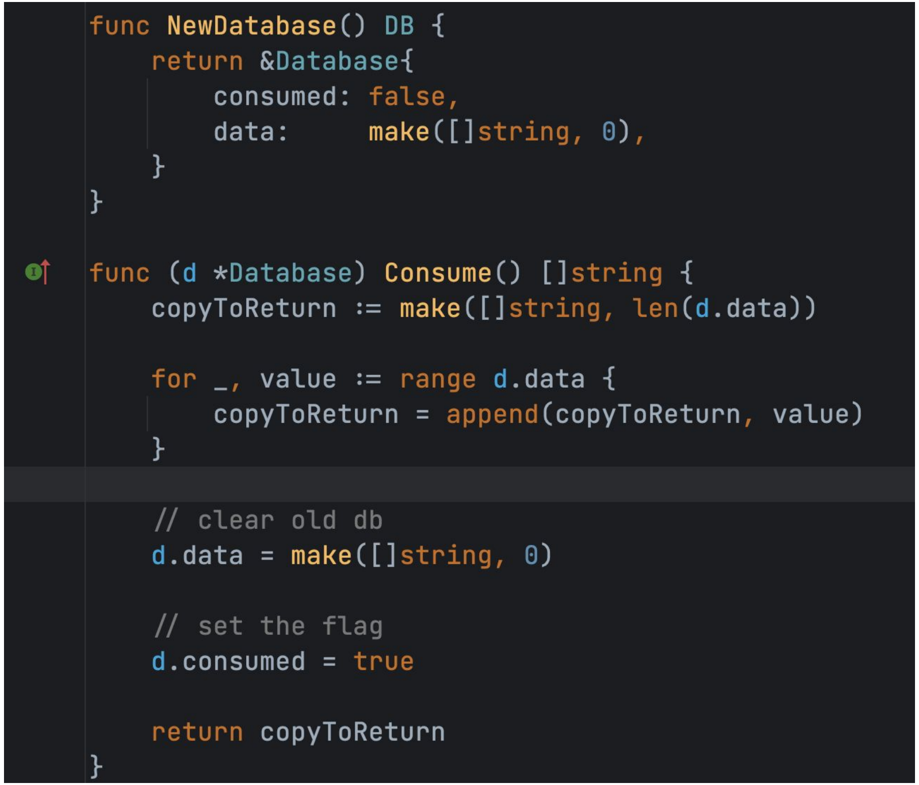




```
type Database struct {
    consumed bool
    data     []string
}

type DB interface {
    Add(data string)
    Consume() []string
}
```

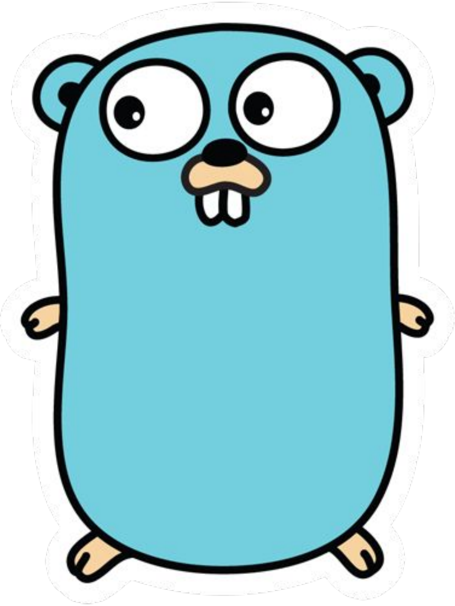
# 3. Consuming variables





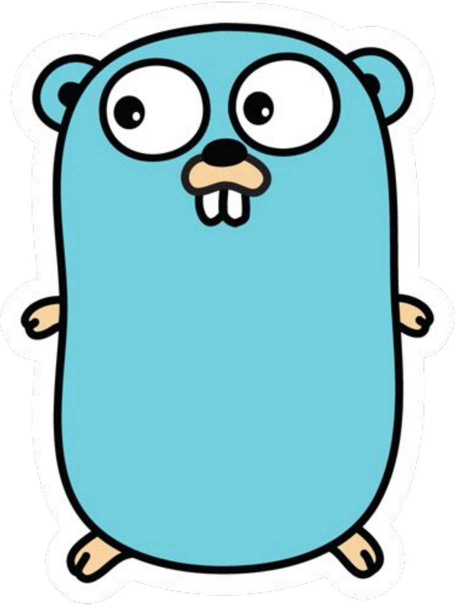
```go
func NewDatabase() DB {
    return &Database{
        consumed: false,
        data:     make([]string, 0),
    }
}

func (d *Database) Consume() []string {
    copyToReturn := make([]string, len(d.data))

    for _, value := range d.data {
        copyToReturn = append(copyToReturn, value)
    }

    // clear old db
    d.data = make([]string, 0)

    // set the flag
    d.consumed = true

    return copyToReturn
}
```

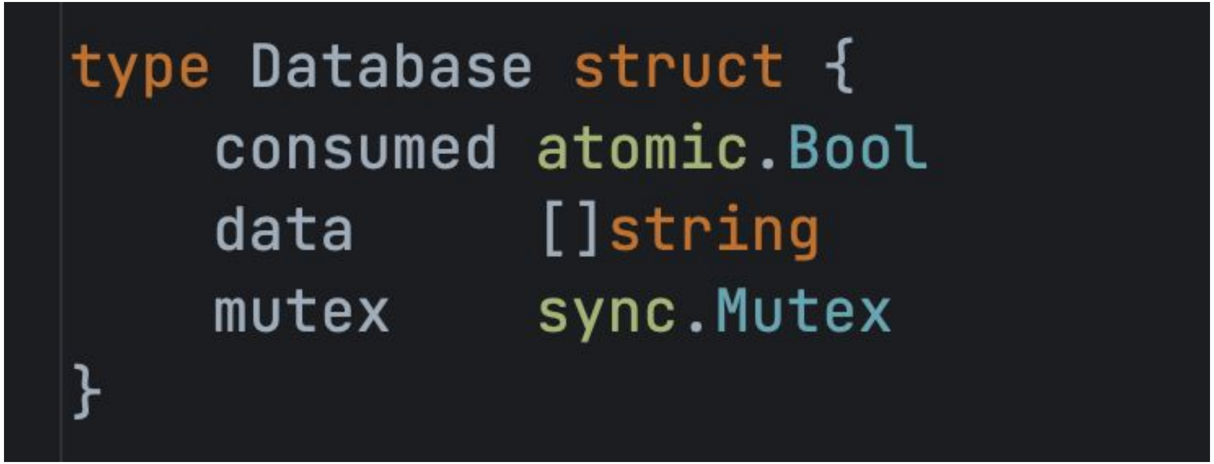
# 3. Consuming variables



thread safe version?

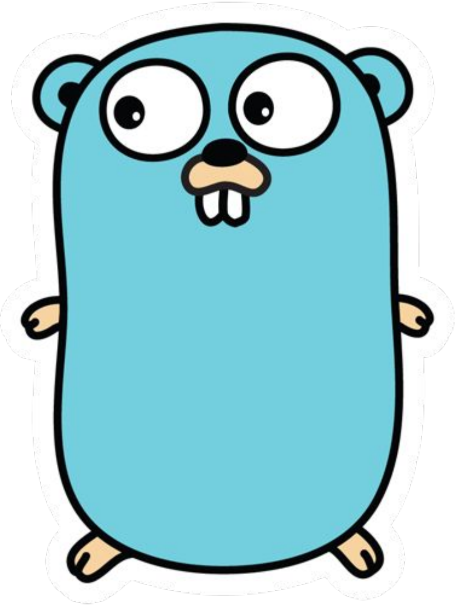
# 3. Consuming variables (thread safe)

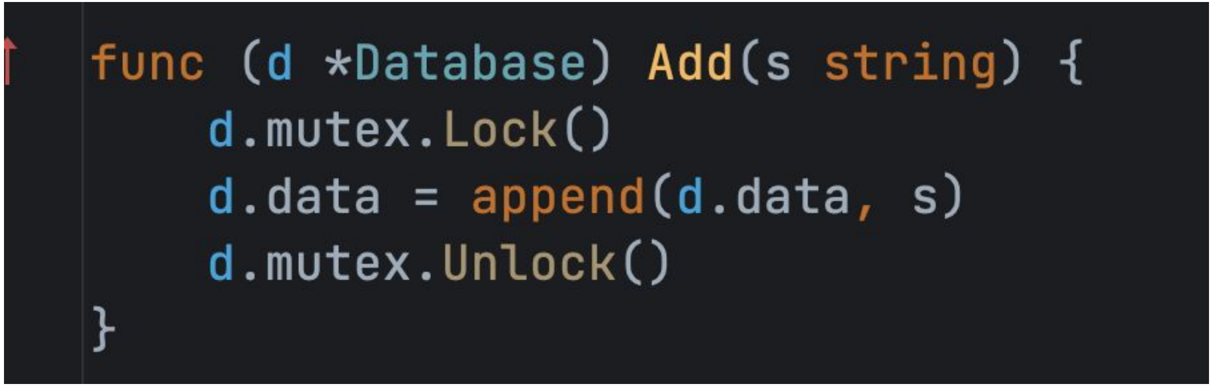




```go
type Database struct {
	consumed atomic.Bool
	data     []string
	mutex    sync.Mutex
}
```

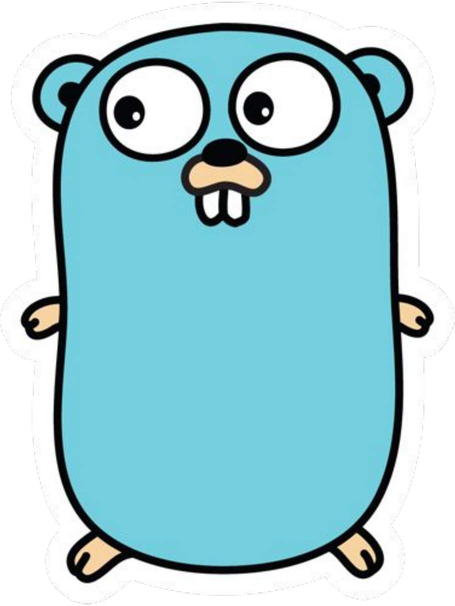
# 3. Consuming variables (thread safe)

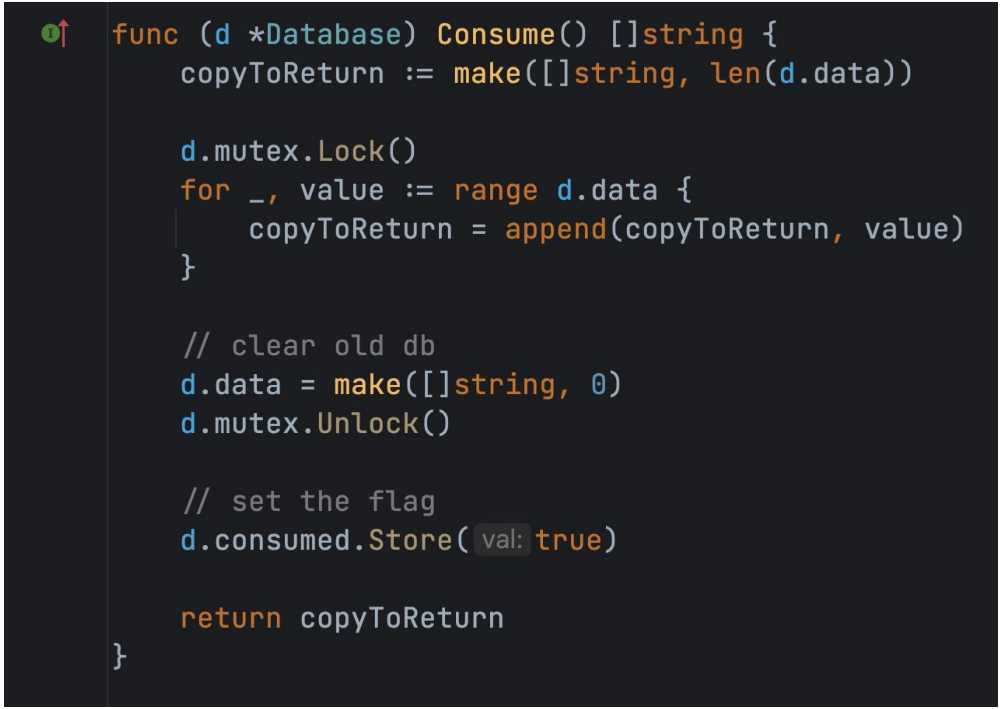




```go
func (d *Database) Add(s string) {
	d.mutex.Lock()
	d.data = append(d.data, s)
	d.mutex.Unlock()
}
```

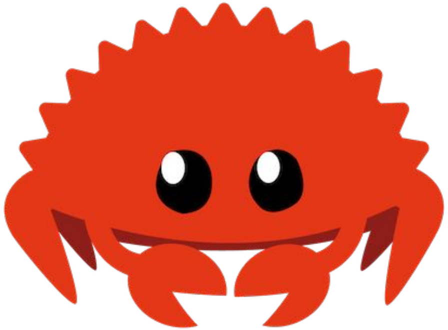
# 3. Consuming variables (thread safe)



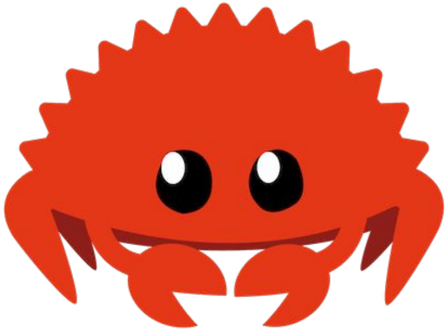


```go
func (d *Database) Consume() []string {
    copyToReturn := make([]string, len(d.data))

    d.mutex.Lock()
    for _, value := range d.data {
        copyToReturn = append(copyToReturn, value)
    }

    // clear old db
    d.data = make([]string, 0)
    d.mutex.Unlock()

    // set the flag
    d.consumed.Store( val: true)

    return copyToReturn
}
```

# 4. Scoping

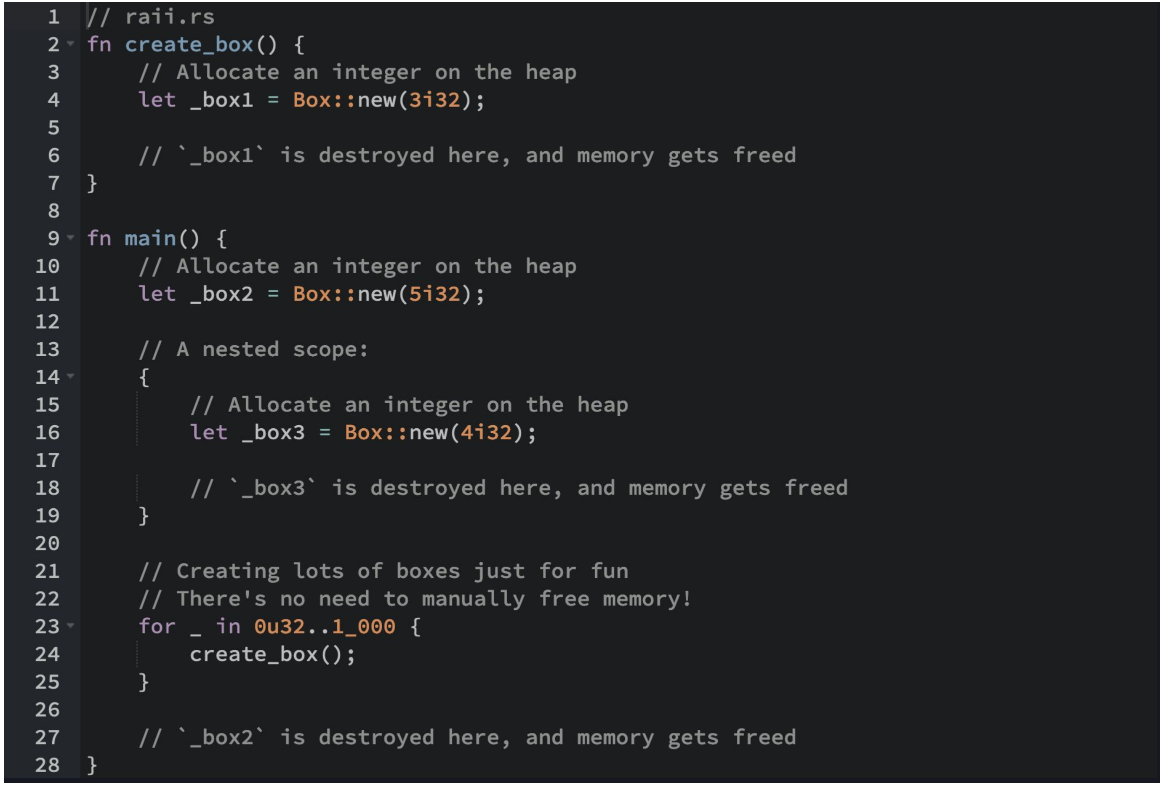


# 3. Scoping

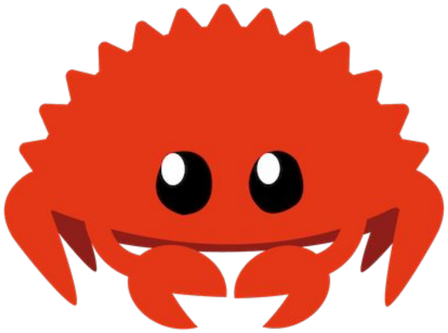


RAII

# 4. Scoping

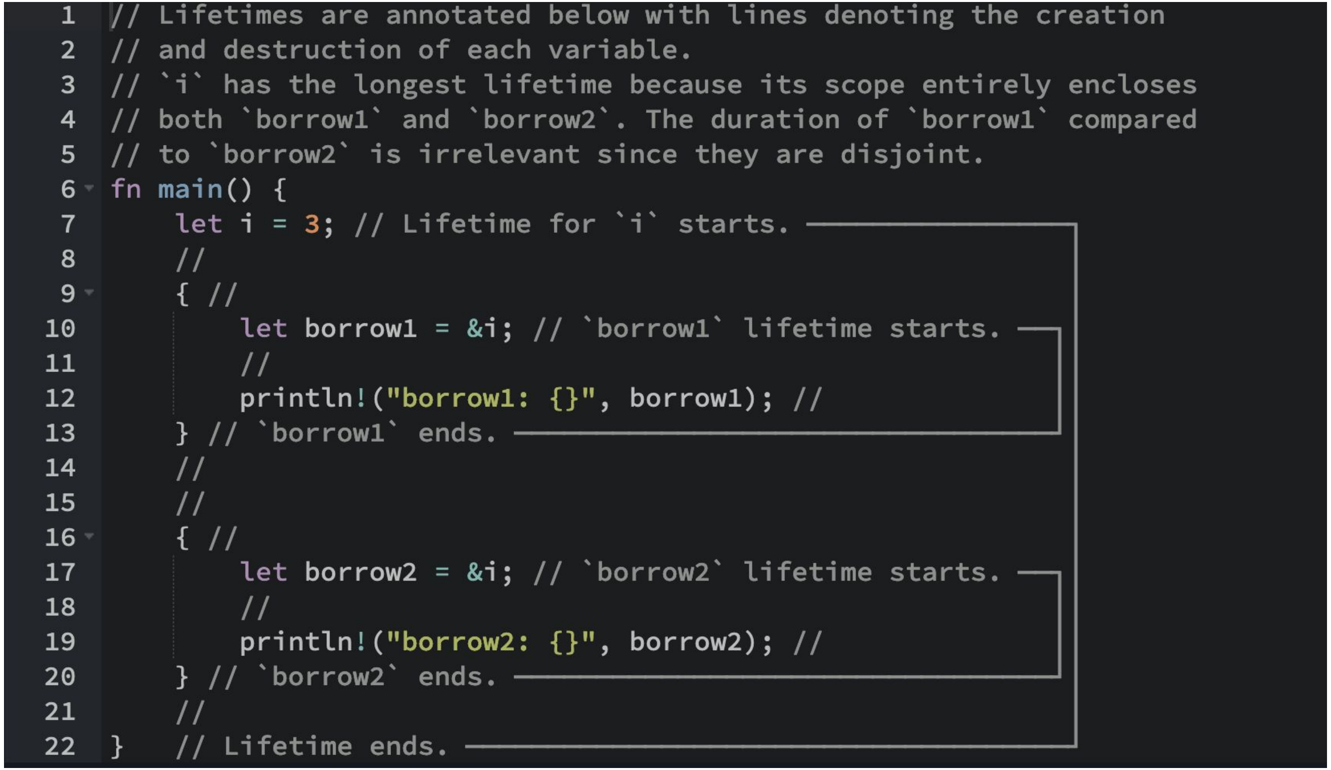


```rust
// raii.rs
fn create_box() {
    // Allocate an integer on the heap
    let _box1 = Box::new(3i32);

    // `_box1` is destroyed here, and memory gets freed
}

fn main() {
    // Allocate an integer on the heap
    let _box2 = Box::new(5i32);

    // A nested scope:
    {
        // Allocate an integer on the heap
        let _box3 = Box::new(4i32);

        // `_box3` is destroyed here, and memory gets freed
    }

    // Creating lots of boxes just for fun
    // There's no need to manually free memory!
    for _ in 0u32..1_000 {
        create_box();
    }

    // `_box2` is destroyed here, and memory gets freed
}
```

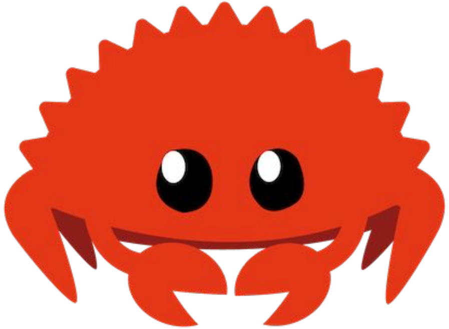


Source: https://doc.rust-lang.org/rust-by-example/scope/raii.html

# 4. Scoping

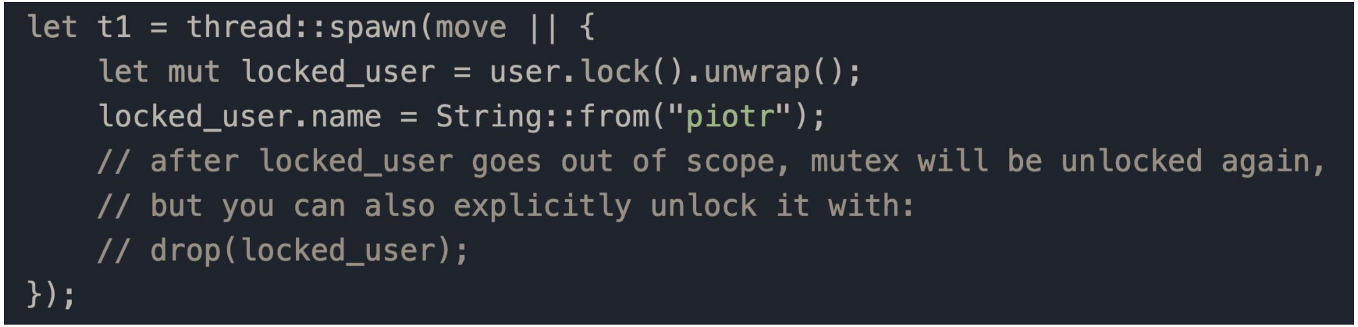


```
// Lifetimes are annotated below with lines denoting the creation
// and destruction of each variable.
// `i` has the longest lifetime because its scope entirely encloses
// both `borrow1` and `borrow2`. The duration of `borrow1` compared
// to `borrow2` is irrelevant since they are disjoint.
fn main() {
    let i = 3; // Lifetime for `i` starts.
    //
    { //
        let borrow1 = &i; // `borrow1` lifetime starts.
        //
        println!("borrow1: {}", borrow1); //
    } // `borrow1` ends.
    //
    //
    { //
        let borrow2 = &i; // `borrow2` lifetime starts.
        //
        println!("borrow2: {}", borrow2); //
    } // `borrow2` ends.
    //
}   // Lifetime ends.
```

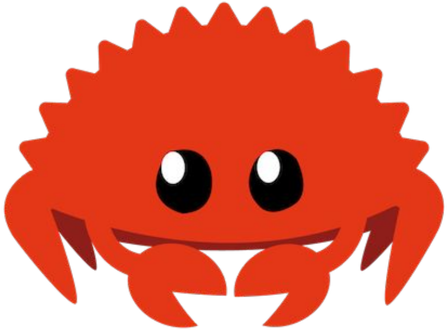


Source: https://doc.rust-lang.org/rust-by-example/scope/lifetime.html

# 4. Scoping

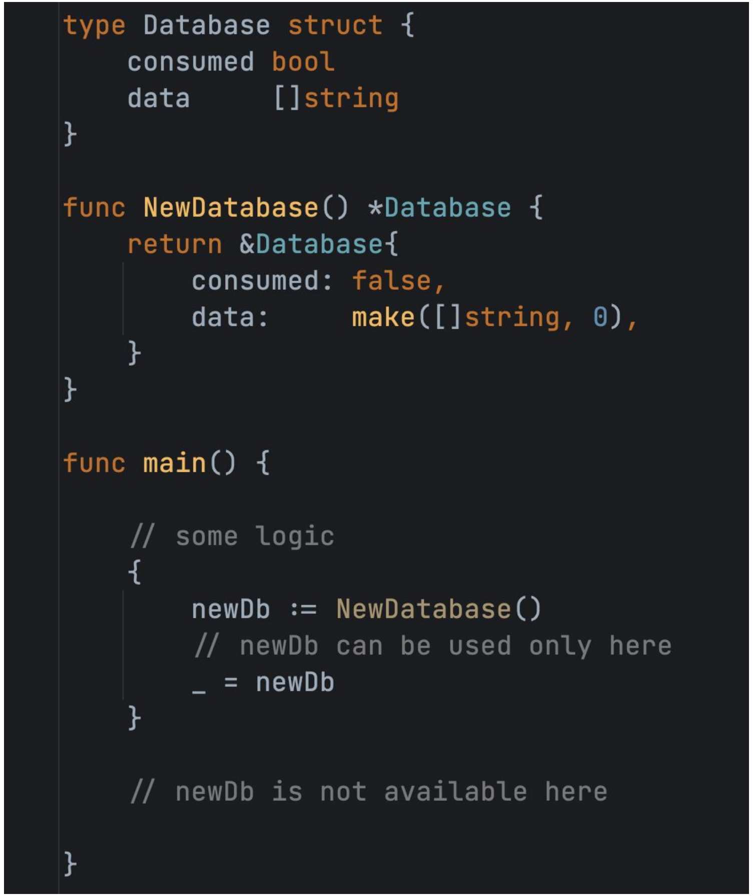


```
let t1 = thread::spawn(move || {
    let mut locked_user = user.lock().unwrap();
    locked_user.name = String::from("piotr");
    // after locked_user goes out of scope, mutex will be unlocked again,
    // but you can also explicitly unlock it with:
    // drop(locked_user);
});
```

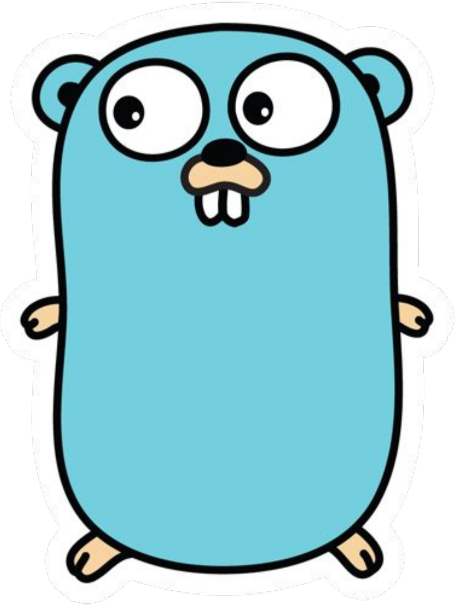


Source: https://itsallaboutthebit.com/arc-mutex/

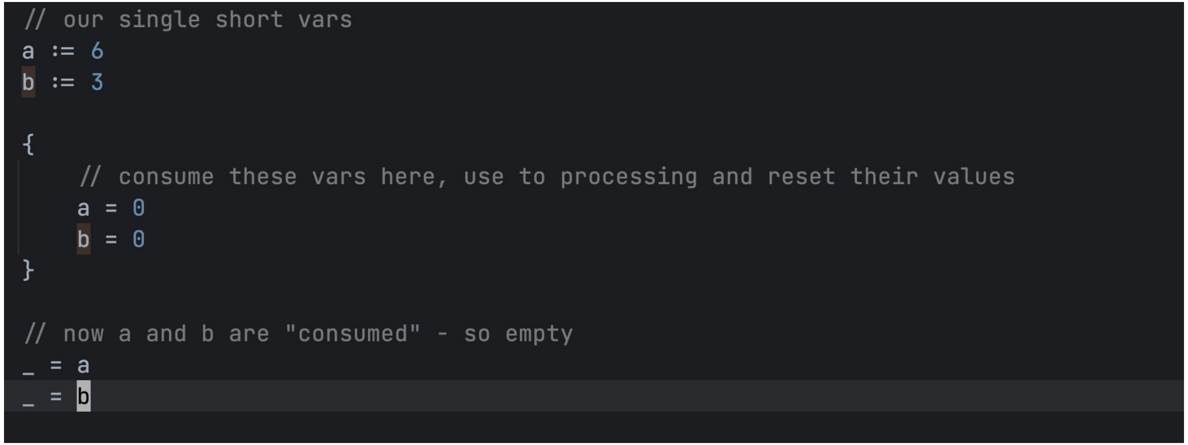
# 4. Scoping



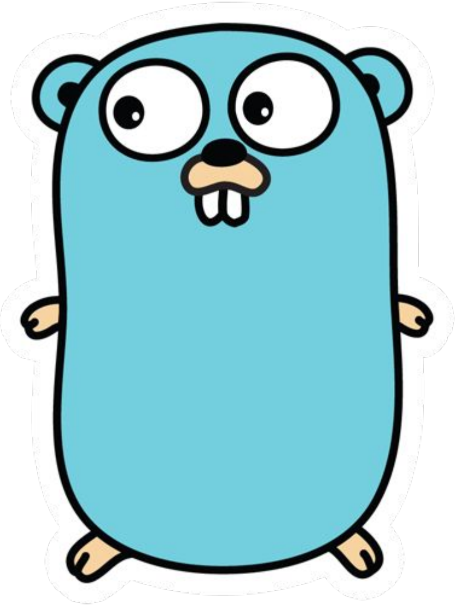
```go
type Database struct {
    consumed bool
    data     []string
}

func NewDatabase() *Database {
    return &Database{
        consumed: false,
        data:     make([]string, 0),
    }
}

func main() {

    // some logic
    {
        newDb := NewDatabase()
        // newDb can be used only here
        _ = newDb
    }

    // newDb is not available here

}
```



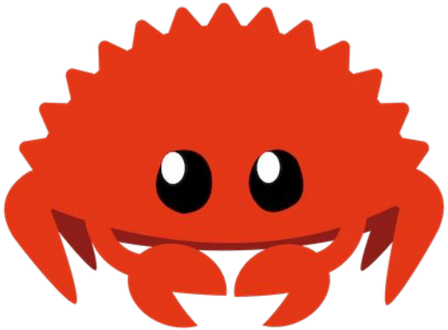
# 4. Scoping (consuming vars cdn)



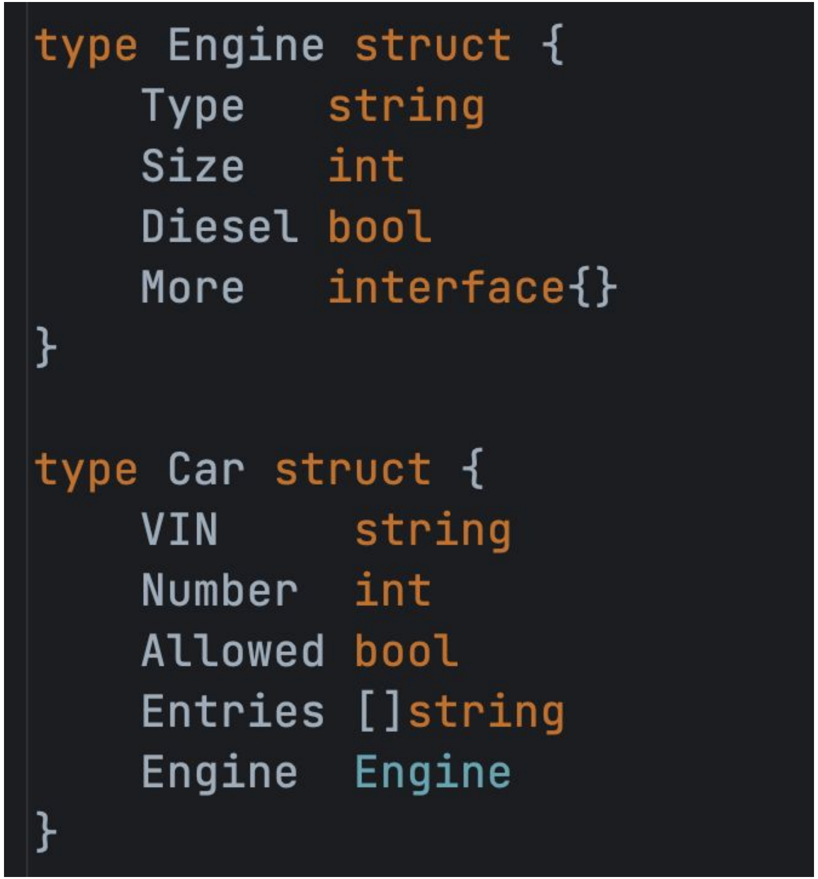
```
// our single short vars
a := 6
b := 3

{
    // consume these vars here, use to processing and reset their values
    a = 0
    b = 0
}

// now a and b are "consumed" - so empty
_ = a
_ = b
```



# 5. Structs

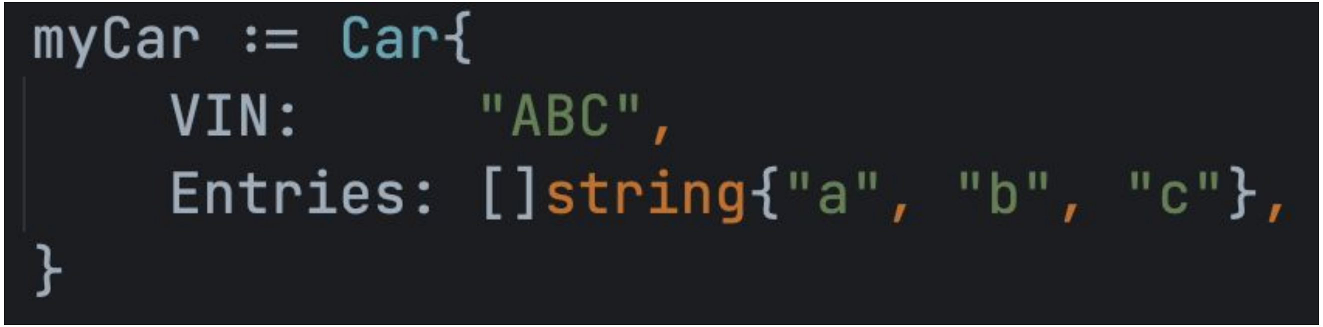


# 5. Structs



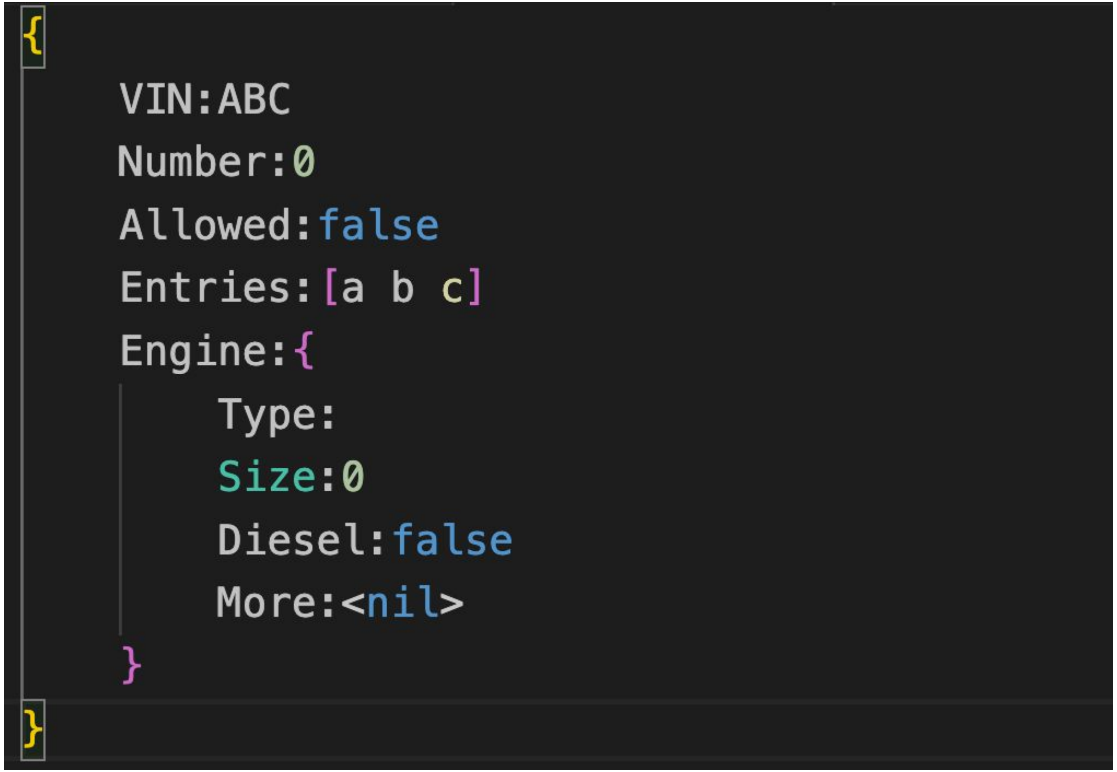
```go
type Engine struct {
	Type   string
	Size   int
	Diesel bool
	More   interface{}
}

type Car struct {
	VIN     string
	Number  int
	Allowed bool
	Entries []string
	Engine  Engine
}
```

# 4. Structs



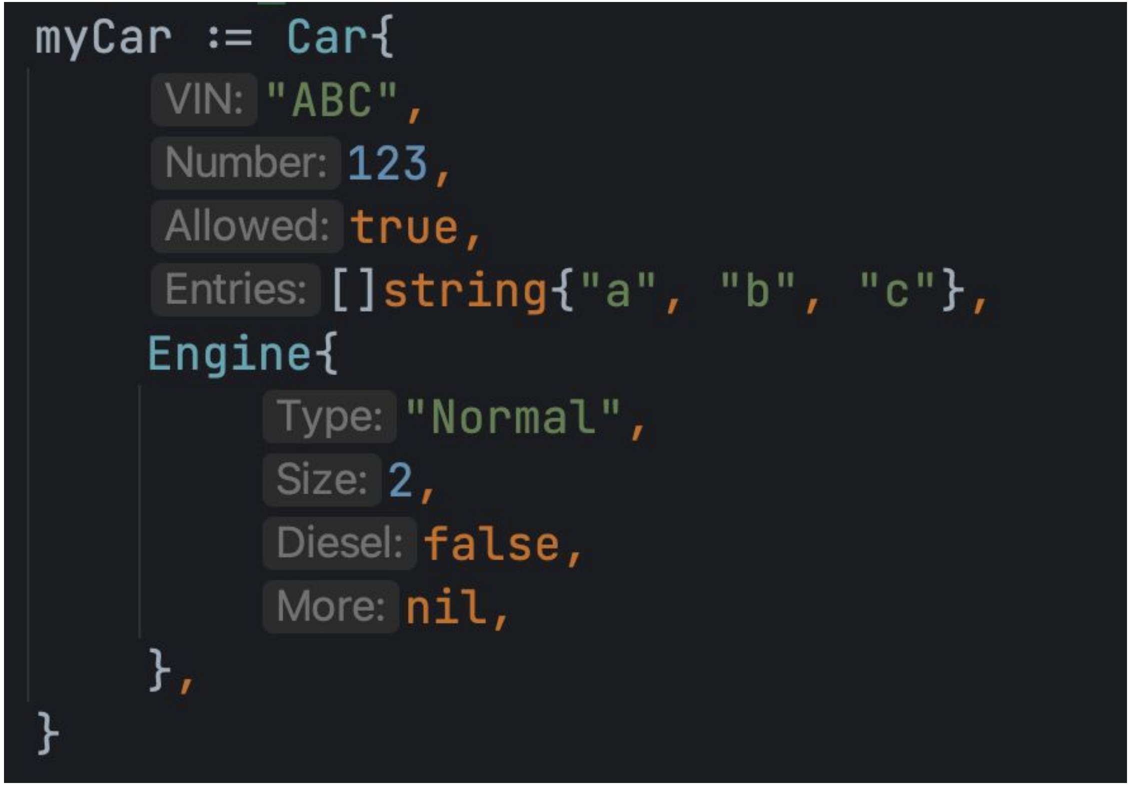
```go
myCar := Car{
	VIN:     "ABC",
	Entries: []string{"a", "b", "c"},
}
```

# 4. Structs

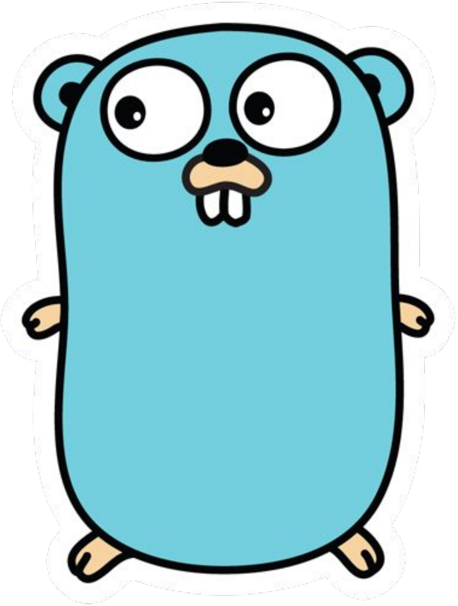


```
{
    VIN:ABC
    Number:0
    Allowed:false
    Entries:[a b c]
    Engine:{
        Type:
        Size:0
        Diesel:false
        More:<nil>
    }
}
```

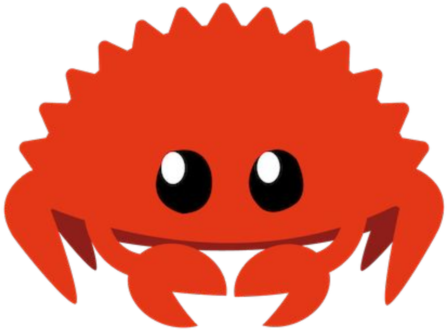
# 4. Structs



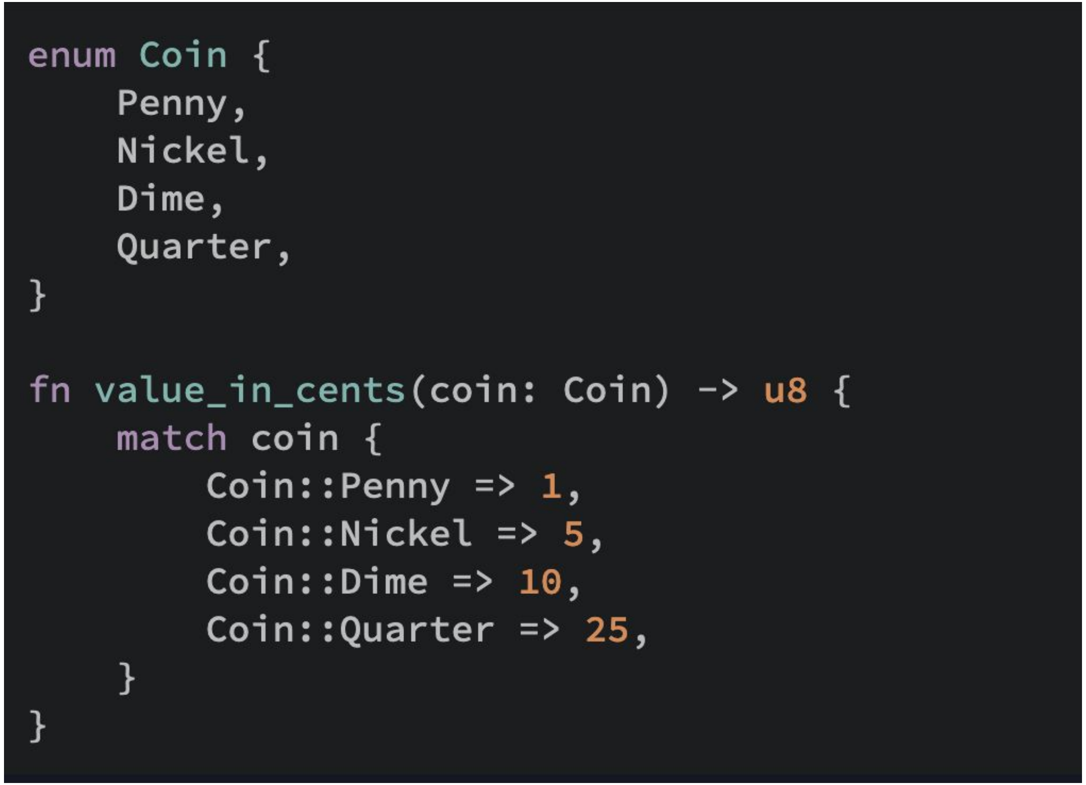
```
myCar := Car{
        VIN: "ABC",
        Number: 123,
        Allowed: true,
        Entries: []string{"a", "b", "c"},
        Engine{
                Type: "Normal",
                Size: 2,
                Diesel: false,
                More: nil,
        },
}
```



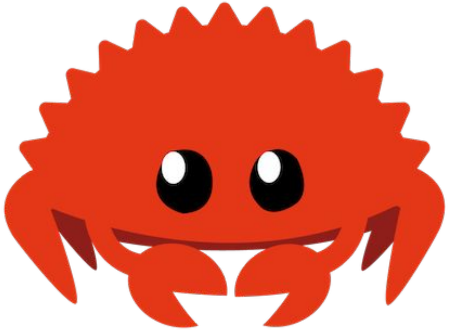
# 5. Enums



# 5. Enums

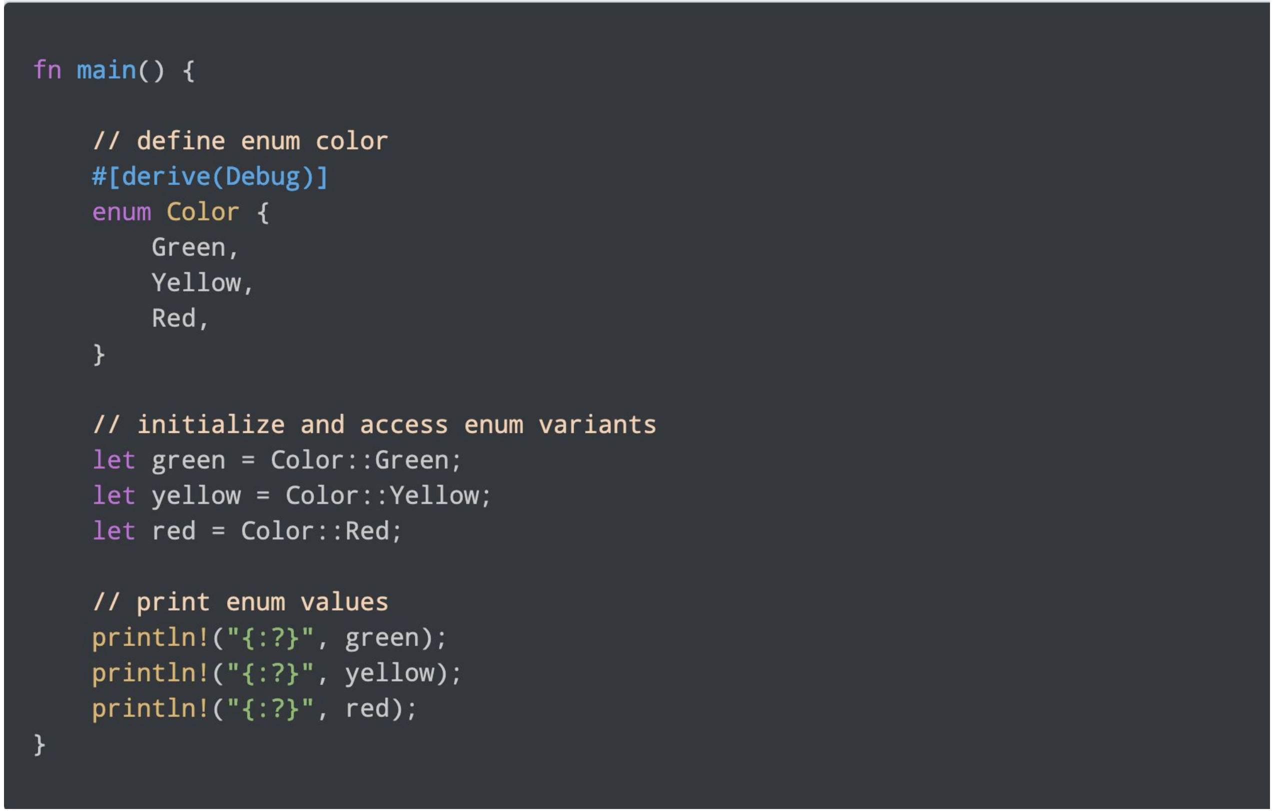


```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

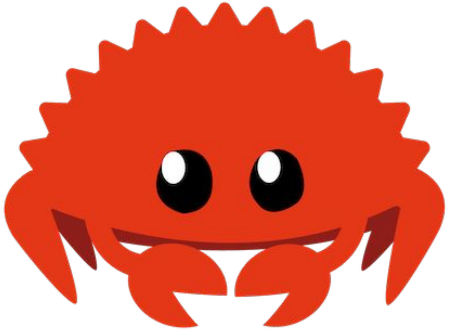


source: https://doc.rust-lang.org/book/ch06-02-match.html

# 5. Enums

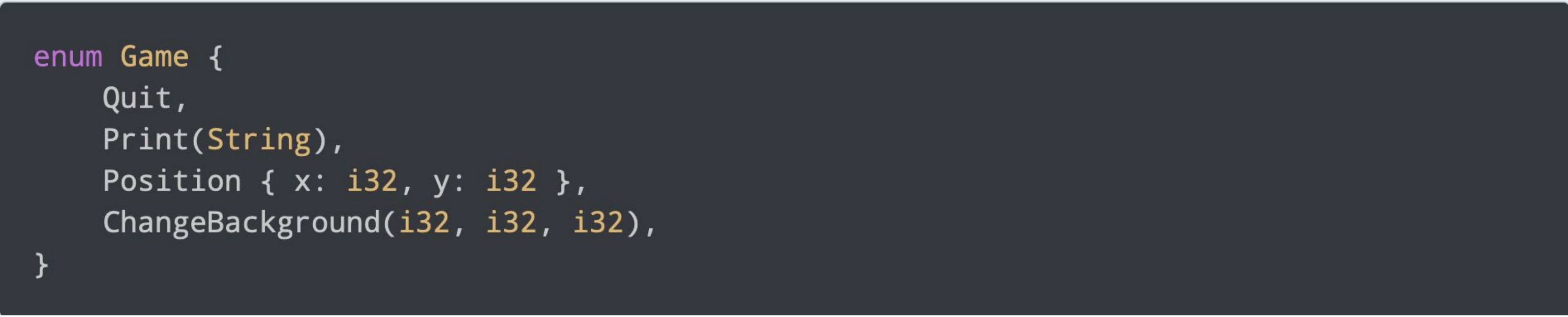


```
fn main() {

    // define enum color
    #[derive(Debug)]
    enum Color {
        Green,
        Yellow,
        Red,
    }

    // initialize and access enum variants
    let green = Color::Green;
    let yellow = Color::Yellow;
    let red = Color::Red;

    // print enum values
    println!("{:?}", green);
    println!("{:?}", yellow);
    println!("{:?}", red);
}
```

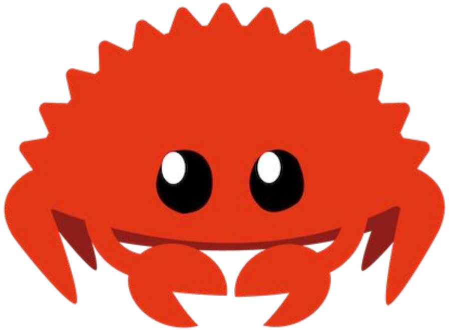


source: https://www.programiz.com/rust/enum

# 5. Enums

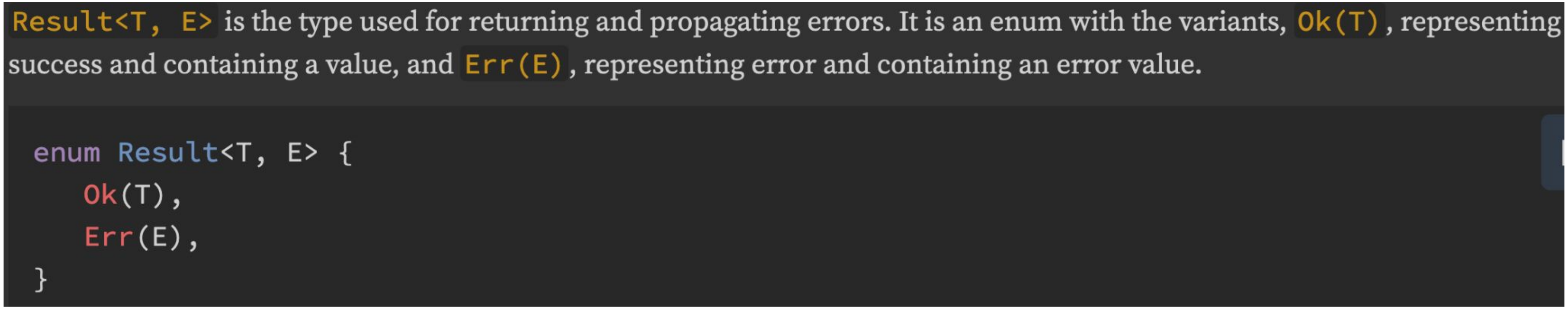


```
enum Game {
    Quit,
    Print(String),
    Position { x: i32, y: i32 },
    ChangeBackground(i32, i32, i32),
}
```



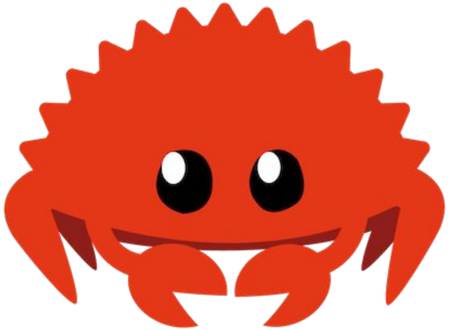
source: https://www.programiz.com/rust/enum

# 5. Enums

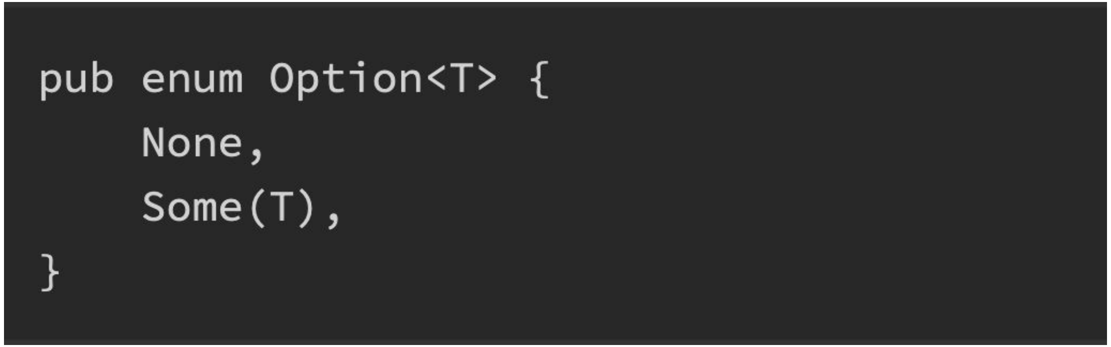


Result<T, E> is the type used for returning and propagating errors. It is an enum with the variants, Ok(T), representing success and containing a value, and Err(E), representing error and containing an error value.

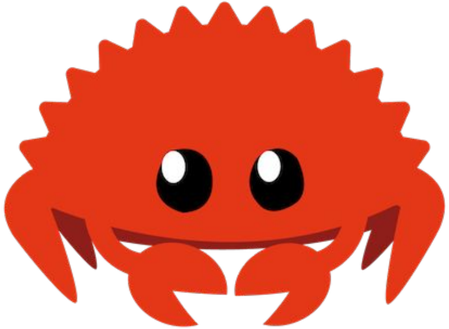
```rust
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```



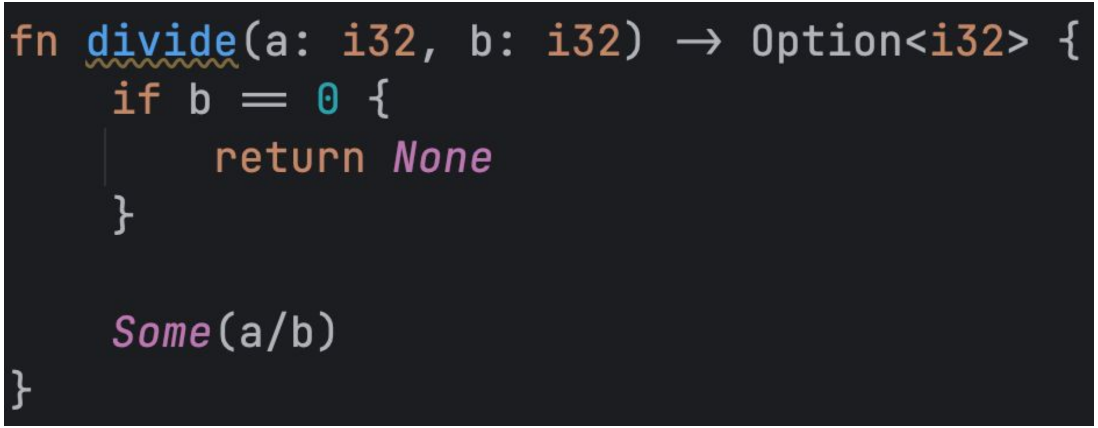
# 5. Enums



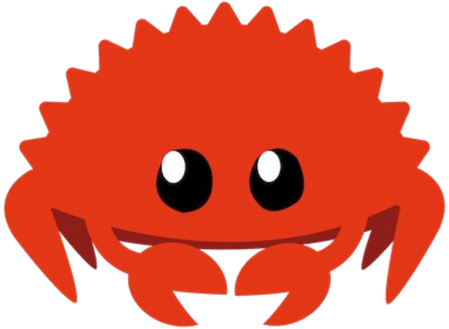
```
pub enum Option<T> {
    None,
    Some(T),
}
```



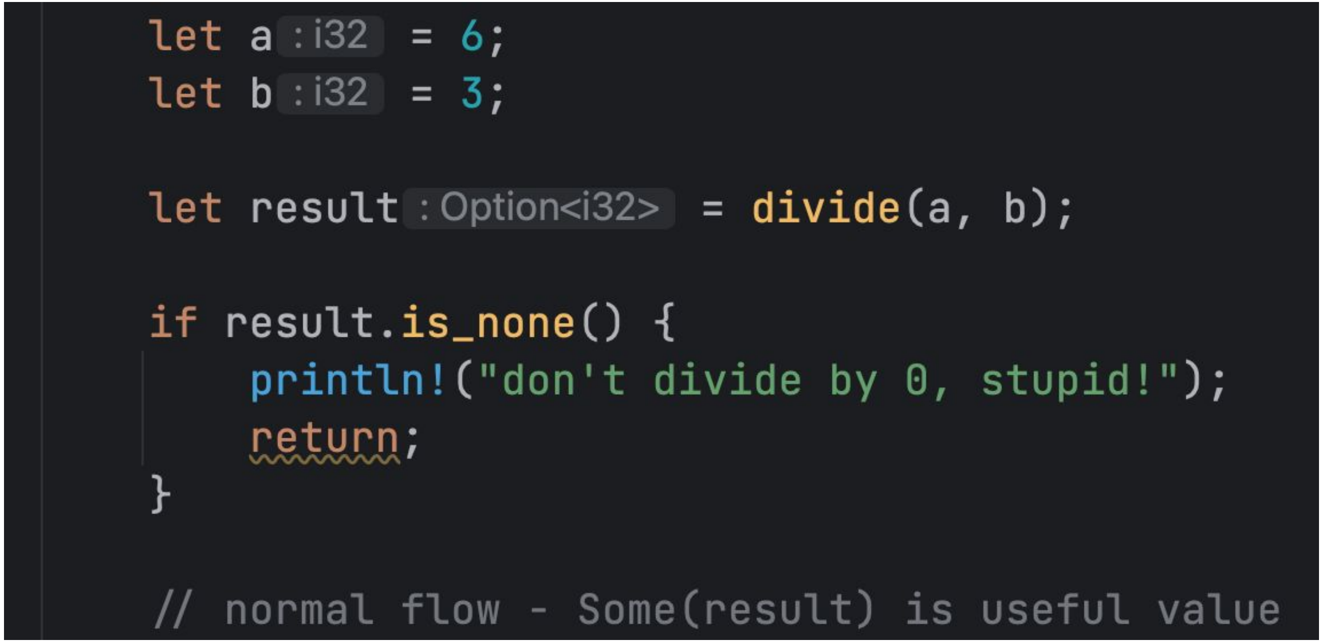
# 5. Enums



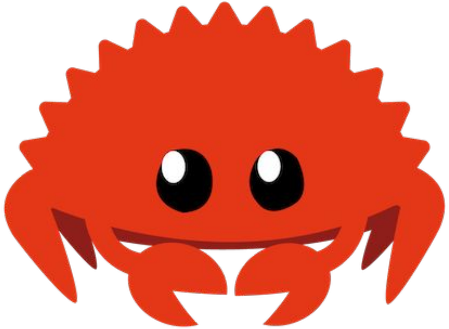
```
fn divide(a: i32, b: i32) -> Option<i32> {
    if b == 0 {
        return None
    }

    Some(a/b)
}
```



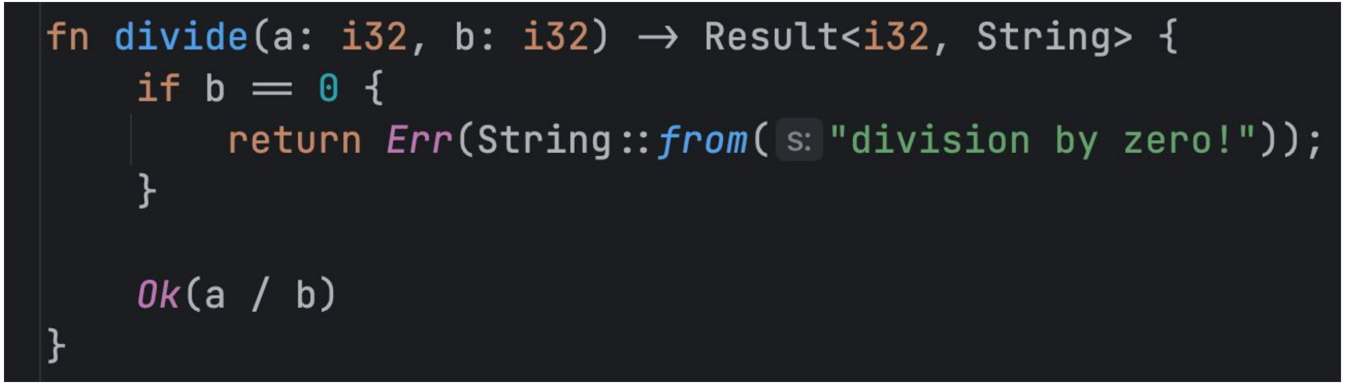
# 5. Enums



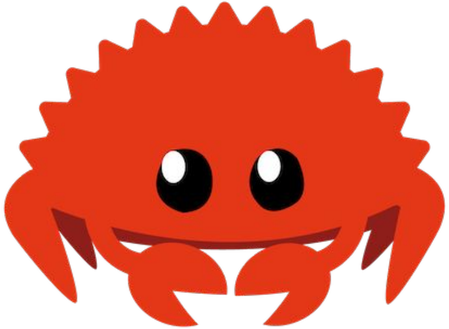
```
    let a : i32 = 6;
    let b : i32 = 3;

    let result : Option<i32> = divide(a, b);

    if result.is_none() {
        println!("don't divide by 0, stupid!");
        return;
    }

    // normal flow - Some(result) is useful value
```



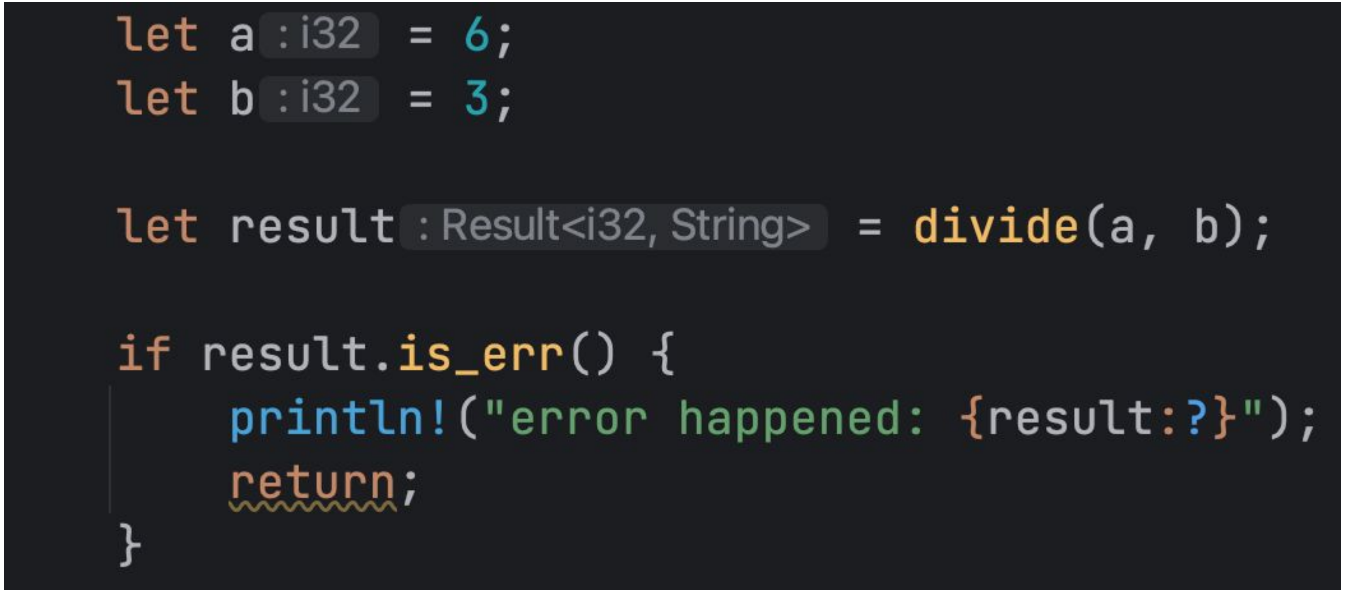
# 5. Enums



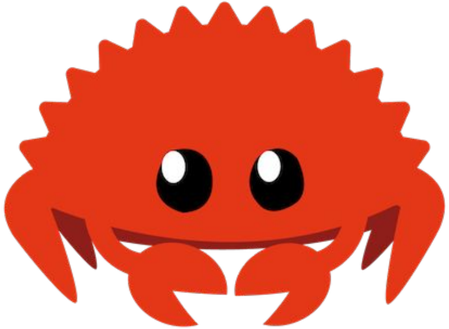
```
fn divide(a: i32, b: i32) -> Result<i32, String> {
    if b == 0 {
        return Err(String::from( s: "division by zero!"));
    }

    Ok(a / b)
}
```



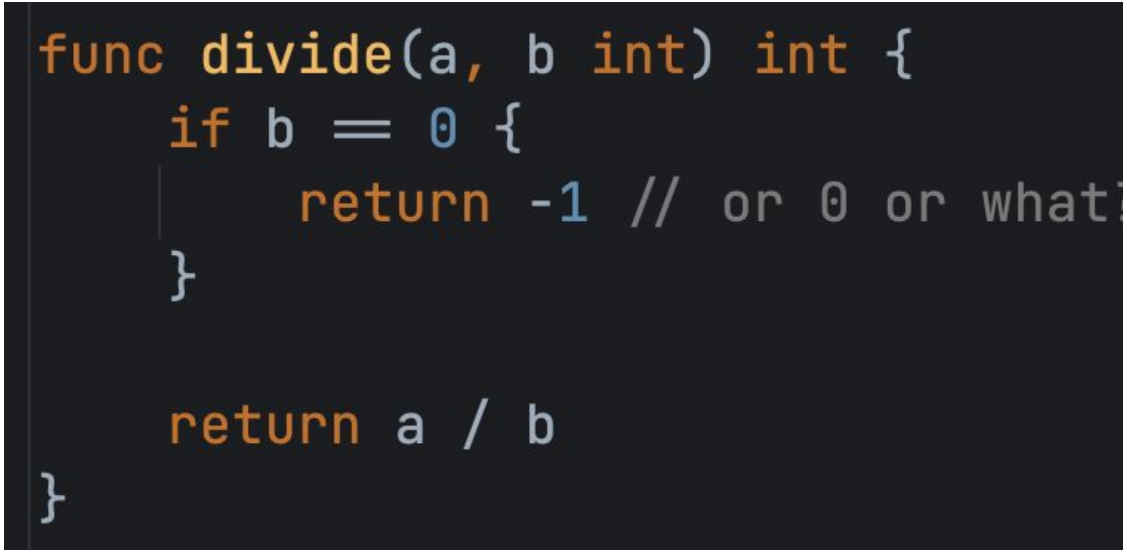
# 5. Enums



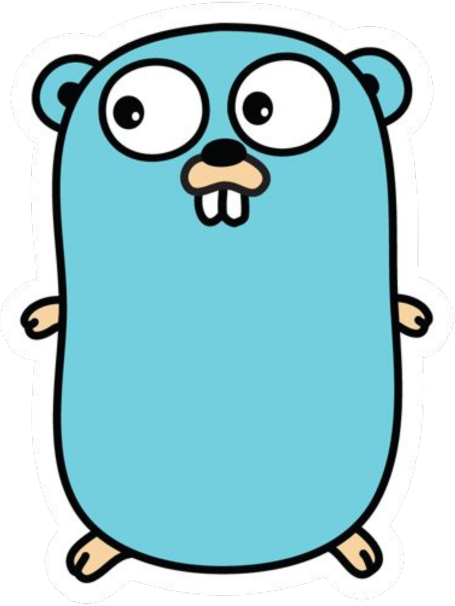
```
let a : i32 = 6;
let b : i32 = 3;

let result : Result<i32, String> = divide(a, b);

if result.is_err() {
    println!("error happened: {result:?}");
    return;
}
```



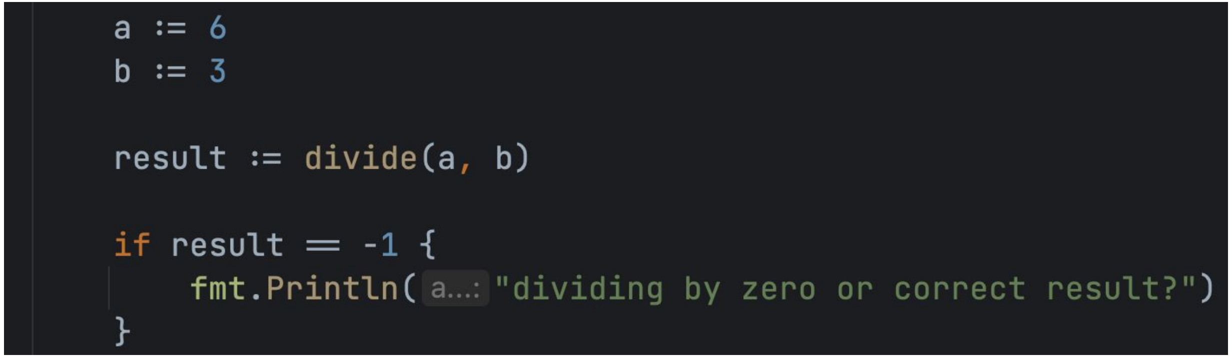
# 5. Enums



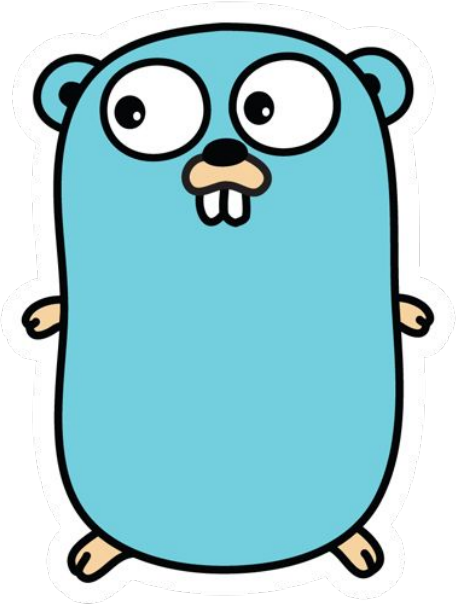
```go
func divide(a, b int) int {
	if b == 0 {
		return -1 // or 0 or what
	}

	return a / b
}
```



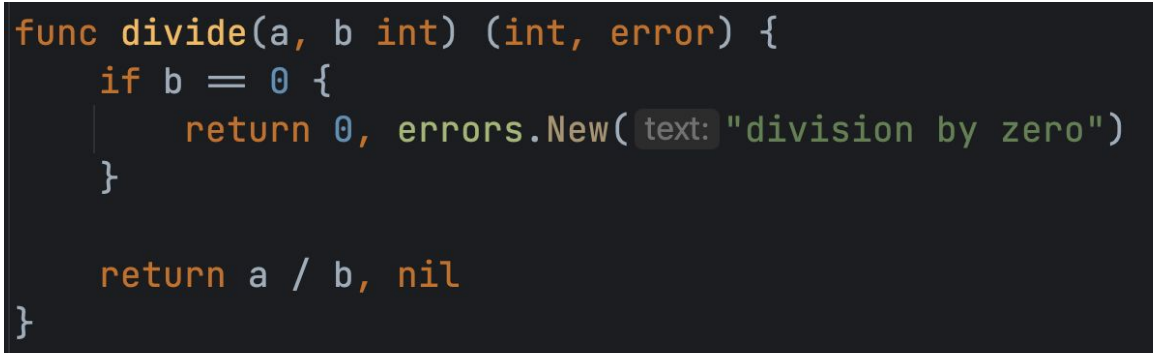
# 5. Enums



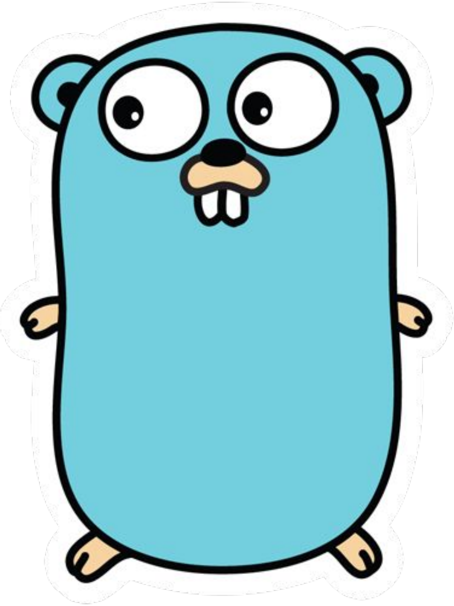
```go
	a := 6
	b := 3

	result := divide(a, b)

	if result == -1 {
		fmt.Println( a...: "dividing by zero or correct result?")
	}
```



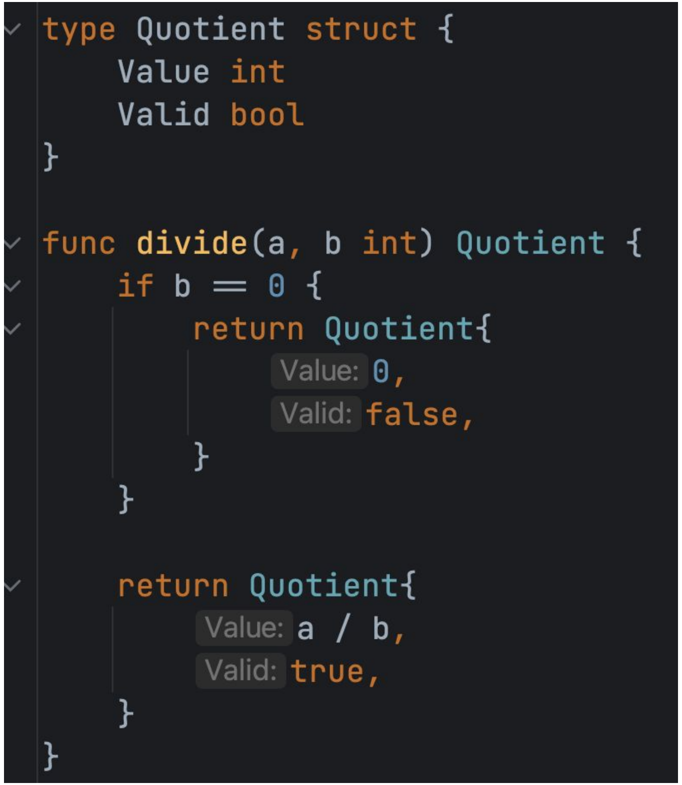
# 5. Enums



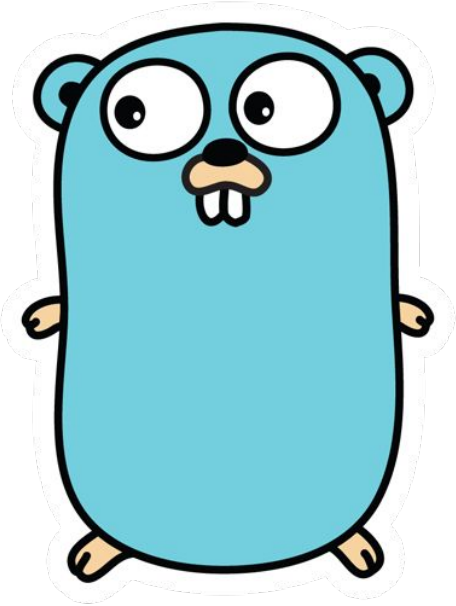
```go
func divide(a, b int) (int, error) {
    if b == 0 {
        return 0, errors.New( text: "division by zero")
    }

    return a / b, nil
}
```



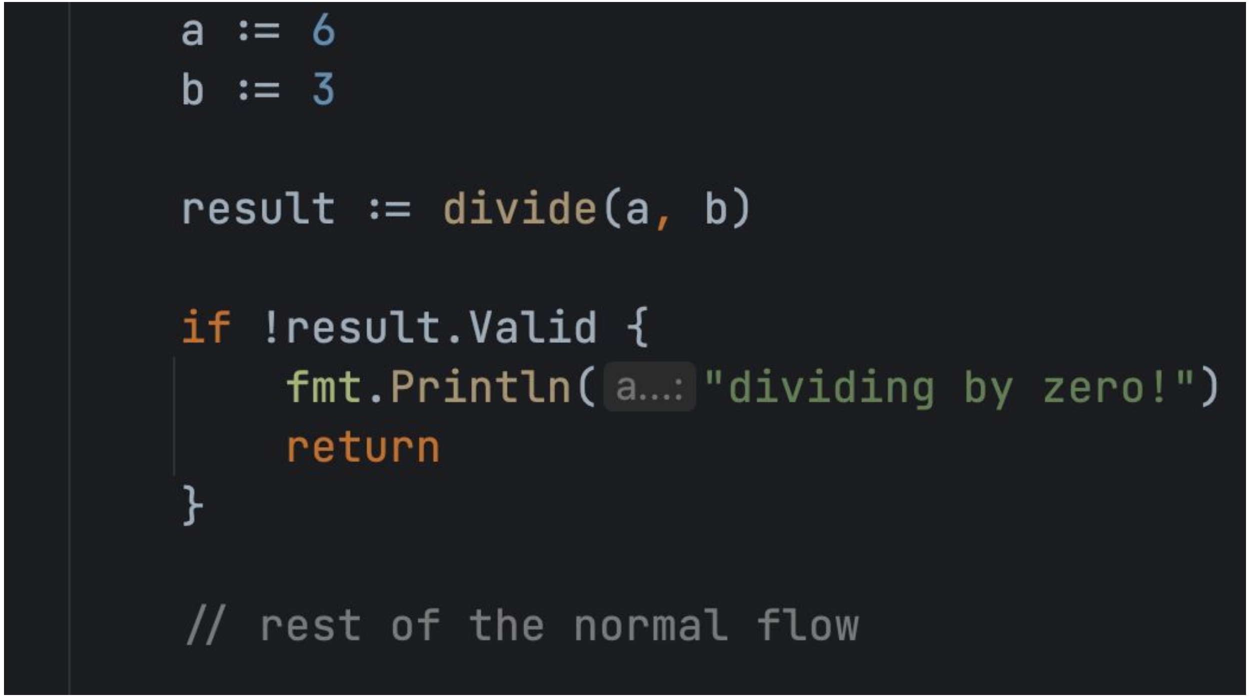
# 5. Enums



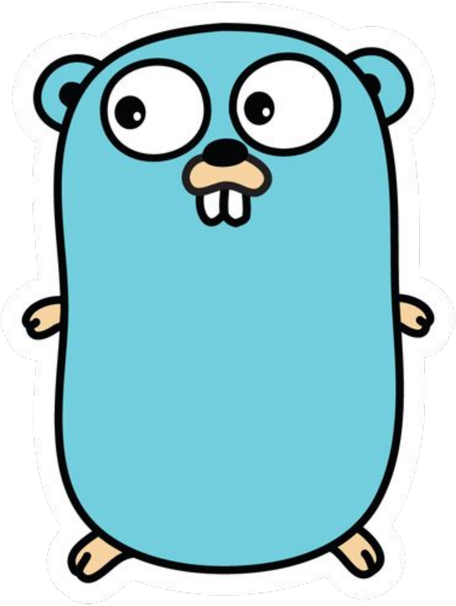
```go
type Quotient struct {
    Value int
    Valid bool
}

func divide(a, b int) Quotient {
    if b == 0 {
        return Quotient{
            Value: 0,
            Valid: false,
        }
    }

    return Quotient{
        Value: a / b,
        Valid: true,
    }
}
```



# 5. Enums

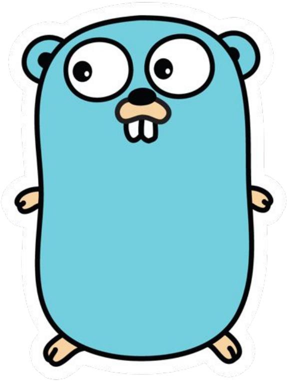


```
a := 6
b := 3

result := divide(a, b)

if !result.Valid {
    fmt.Println( a...: "dividing by zero!")
    return
}

// rest of the normal flow
```

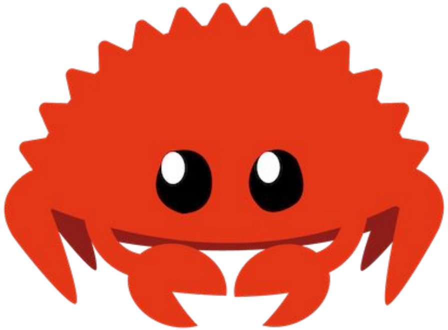


# 5. Enums

https://stackoverflow.com/questions/14426366/what-is-an-idiomatic-way-of-representing-enums-in-go

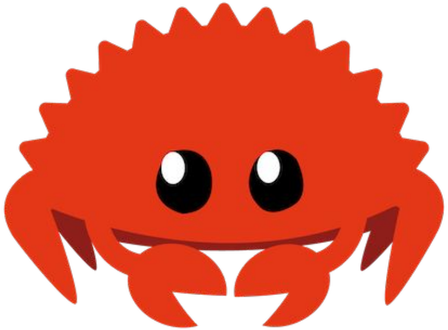


# 6. Concurrency & parallelism

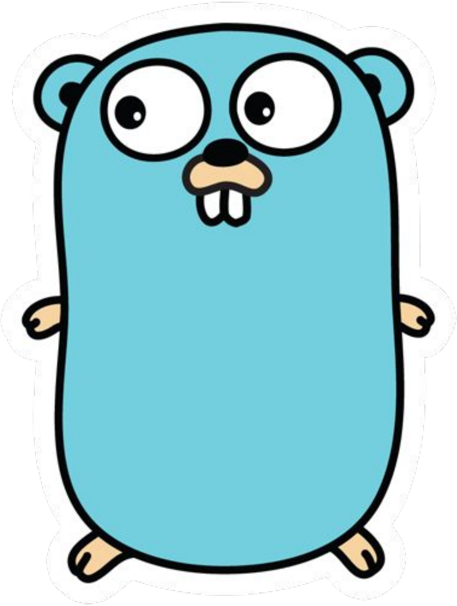


# 6. Concurrency & parallelism

https://doc.rust-lang.org/nomicon/concurrency.html

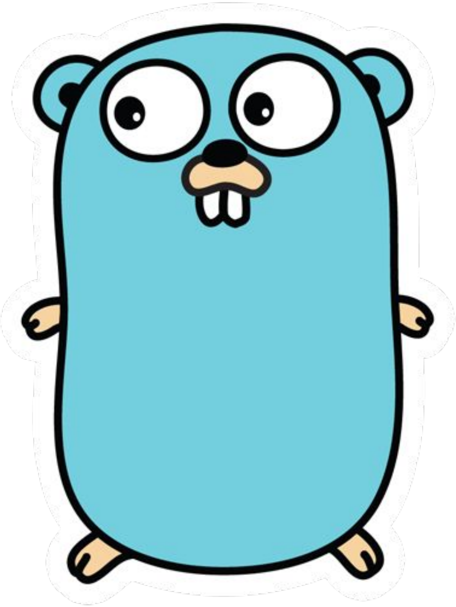


# 6. Concurrency & parallelism



# 6. Concurrency & parallelism

Use immutable data structures; copy (if possible) maps and slices using mutexes, use value receiver for structs (instead of pointers)



# 7. Useful links

[PL] https://tiny.pl/cgpnw - wzmacniamy system typów w Go

https://rosettacode.org/wiki/Category:Programming_Tasks - programming tasks in many languages

https://learnxinyminutes.com/docs/rust/ - if you want to learn Rust quickly

https://david-peter.de/cube-composer/ - learning FP on building blocks

https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.710.2018&rep=rep1&type=pdf -

How Much Does Unused Code Matter for Maintenance

# Q&A

# Thank You!

SCAN ME