Welcome To

DISRUPT YOUR INDUSTRY

Pointers

“Pointers is a variable that is used to store the memory address of another variable”

Pointers

“ Pointers is a variable that is used to store the memory address of another variable ”

```
var p *int64
```

Pointers

```
var value int = 10 //declaring a variable

var ptr *int // declaration of pointer

ptr = &value // initialization of pointer
```

Pointers

```
var value int = 10 //declaring a variable

var ptr *int // declaration of pointer

ptr = &value // initialization of pointer
```

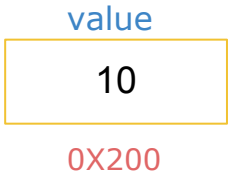
- * **Operator** also termed as the dereferencing operator used to declare pointer variable and access the value stored in the address.

Pointers

```
var value int = 10 //declaring a variable

var ptr *int // declaration of pointer

ptr = &value // initialization of pointer
```

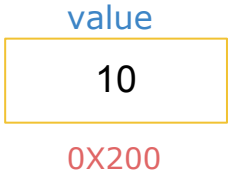
- * **Operator** also termed as the dereferencing operator used to declare pointer variable and access the value stored in the address.
- & **operator** termed as address operator used to returns the address of a variable.

Pointers



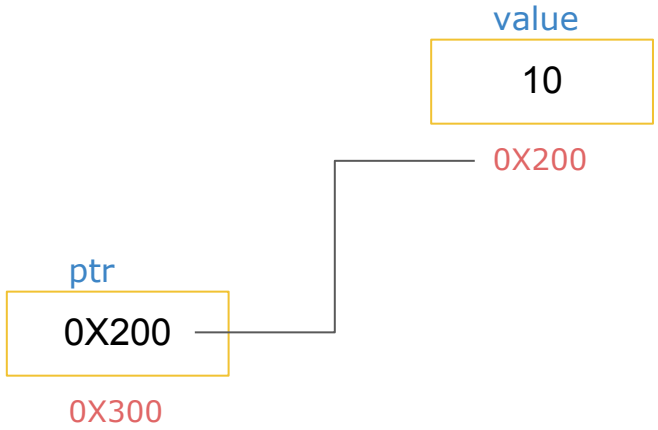

```
var value int = 10 //declaring a variable
```

Pointers



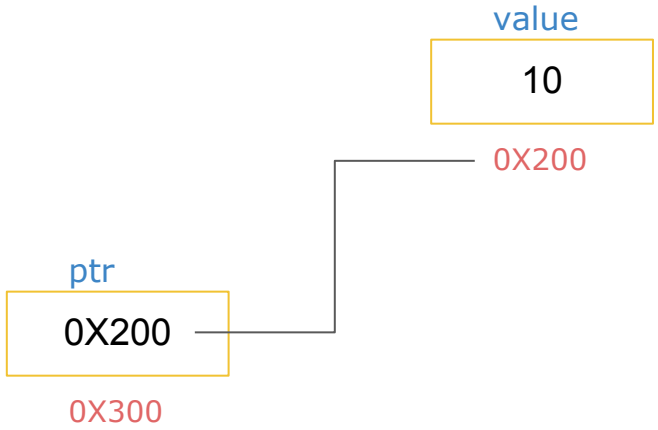

```
var value int = 10 //declaring a variable

var ptr *int // declaration of pointer
```

Pointers




```
var value int = 10 //declaring a variable

var ptr *int // declaration of pointer

ptr = &value // initialization of pointer
```

Pointers




```
var value int = 10 //declaring a variable

var ptr *int // declaration of pointer

ptr = &value // initialization of pointer

fmt.Println("Address of value = ", &value)
fmt.Println("Value stored in ptr = ", ptr)
fmt.Println("dereferencing ptr = ", *ptr)
```

```
type User struct {
    Name              string
    Email             string
    EncryptedPassword string
}
```

```
type User struct {
    Name              string
    Email             string
    EncryptedPassword string
}
```

```
func RedactUser(u User) {
    u.EncryptedPassword = ""
    return
}
```

```go
type User struct {
    Name              string
    Email             string
    EncryptedPassword string
}
```

```go
func RedactUser(u User) {
    u.EncryptedPassword = ""
    return
}
```

```go
func main() {
    u := User{
        Name:              "name",
        Email:             "name@example.com",
        EncryptedPassword: "AXNAUHEJWNXJANJKLSNIKSNXLKSN!",
    }

    RedactUser(u)

    fmt.Println(u)
}
```

```go
type User struct {
    Name              string
    Email             string
    EncryptedPassword string
}
```

```go
func RedactUser(u User) {
    u.EncryptedPassword = ""
    return
}
```

```go
func main() {
    u := User{
        Name:              "name",
        Email:             "name@example.com",
        EncryptedPassword: "AXNAUHEJWNXJANJKLSNIKSNXLKSN!",
    }

    RedactUser(u)

    fmt.Println(u)
}
```

```
{name name@example.com AXNAUHEJWNXJANJKLSNIKSNXLKSN!}
```

```go
type User struct {
    Name              string
    Email             string
    EncryptedPassword string
}
```

```go
func RedactUser(u *User) {
    u.EncryptedPassword = ""
    return
}
```

```go
func main() {
    u := User{
        Name:              "name",
        Email:             "name@example.com",
        EncryptedPassword: "AXNAUHEJWNXJANJKLSNIKSNXLKSN!",
    }

    RedactUser(&u)

    fmt.Println(u)
}
```

```go
type User struct {
    Name              string
    Email             string
    EncryptedPassword string
}
```

```go
func RedactUser(u *User) {
    u.EncryptedPassword = ""
    return
}
```

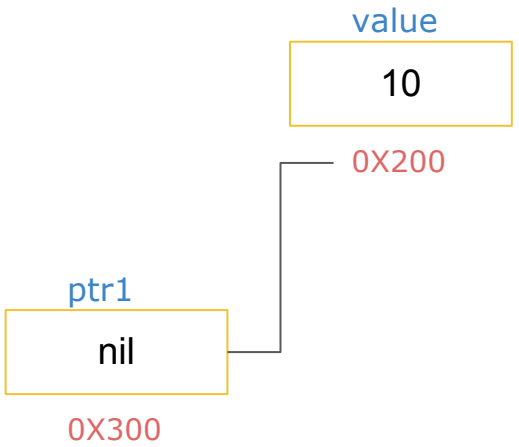
```go
func main() {
    u := User{
        Name:              "name",
        Email:             "name@example.com",
        EncryptedPassword: "AXNAUHEJWNXJANJKLSNIKSNXLKSN!",
    }

    RedactUser(&u)

    fmt.Println(u)
}
```

```
{name name@example.com }
```

Pointer to Pointer

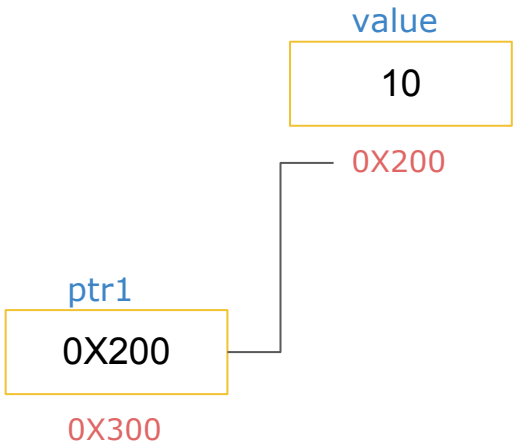
```
var value int = 10
```

Pointer to Pointer



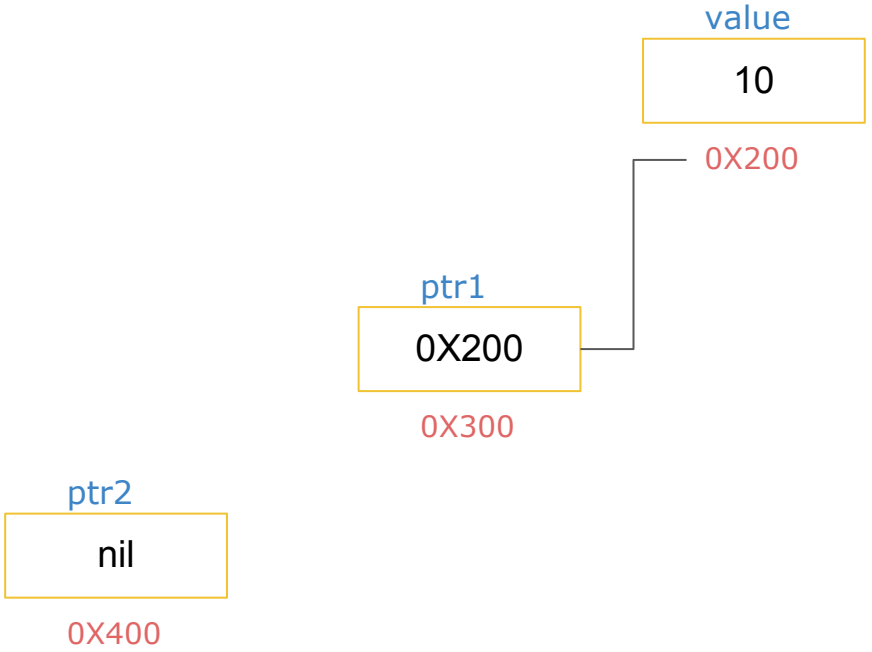

```
var value int = 10

var ptr1 *int
```

Pointer to Pointer



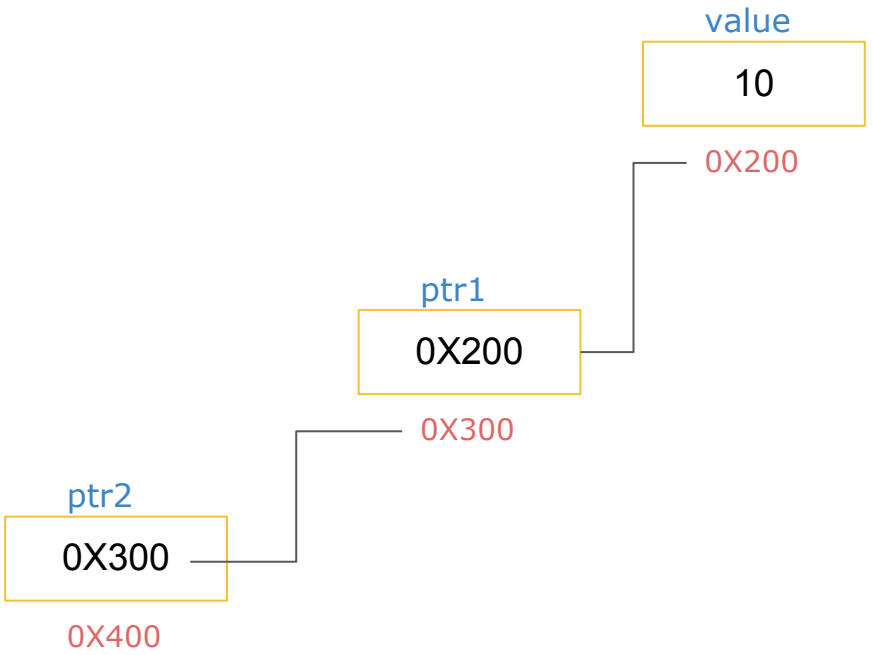

```
var value int = 10

var ptr1 *int
ptr1 = &value
```

Pointer to Pointer



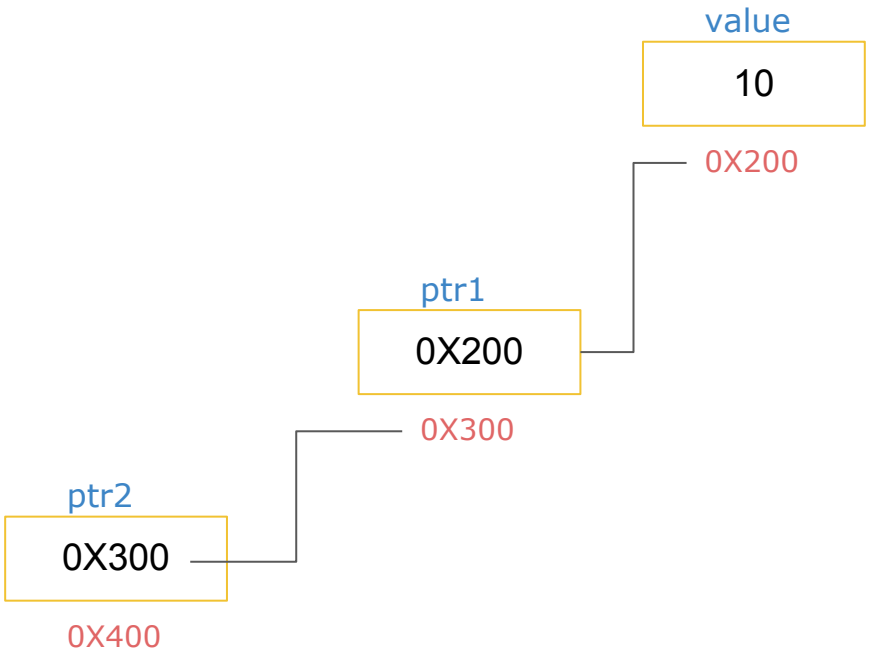

```
var value int = 10

var ptr1 *int
ptr1 = &value

var ptr2 **int
```

Pointer to Pointer




```
var value int = 10

var ptr1 *int
ptr1 = &value

var ptr2 **int
ptr2 = &ptr1
```

Pointer to Pointer




```
var value int = 10

var ptr1 *int
ptr1 = &value

var ptr2 **int
ptr2 = &ptr1

fmt.Println("The Value of variable value is = ", value)
fmt.Println("Address of variable value is = ", &value)

fmt.Println("The value of ptr1 is = ", ptr1)
fmt.Println("Address of ptr1 is = ", &ptr1)

fmt.Println("The value of ptr2 is = ", ptr2)
fmt.Println("**ptr2 = ", **ptr2)
```

Methods

- A method is a function with a special *receiver* argument.

Methods

- A method is a function with a special *receiver* argument.
- The receiver appears in its own argument list between the `func` keyword and the method name.

```
type User struct {
    Name              string
    Email             string
    EncryptedPassword string
}
```

```
func (u User) RedactUser() {
    // code
}
```

Methods

- A method is a function with a special *receiver* argument.
- The receiver appears in its own argument list between the `func` keyword and the method name.

```
type User struct {
    Name              string
    Email             string
    EncryptedPassword string
}
```

```
func (u User) RedactUser() {
    // code
}
```

```
var u User
u.RedactUser()
```

Methods

- A method is a function with a special *receiver* argument.
- The receiver appears in its own argument list between the `func` keyword and the method name.
- You can declare a method on non-struct types, too.

Methods

- A method is a function with a special *receiver* argument.
- The receiver appears in its own argument list between the `func` keyword and the method name.
- You can declare a method on non-struct types, too.

```
type MyFloat float64

func (f MyFloat) Abs() float64 {
    if f < 0 {
        return float64(-f)
    }
    return float64(f)
}

func main() {
    var m MyFloat
    m = math.Sqrt2
    fmt.Println(m.Abs())
}
```

Methods

- A method is a function with a special *receiver* argument.
- The receiver appears in its own argument list between the `func` keyword and the method name.
- You can declare a method on non-struct types, too.
- You can only declare a method with a receiver whose type is defined in the same package as the method. You cannot declare a method with a receiver whose type is defined in another package (which includes the built-in types such as `int`).

```
type MyFloat float64

func (f MyFloat) Abs() float64 {
  if f < 0 {
      return float64(-f)
  }
  return float64(f)
}

func main() {
  var m MyFloat
  m = math.Sqrt2
  fmt.Println(m.Abs())
}
```

Methods - Pointer receivers

- You can declare methods with pointer receivers.

.

Methods - Pointer receivers

- You can declare methods with pointer receivers.
- Receiver type itself should not be a pointer.

```
type MyFloat float64

func (f *MyFloat) Half() {
        *f = *f / 2.0
}
```

Methods - Pointer receivers

- You can declare methods with pointer receivers.
- Receiver type itself should not be a pointer.
- Methods with pointer receivers can modify the value to which the receiver points.

.

```
type User struct {
   Name              string
   Email             string
   EncryptedPassword string
}
```

```
func (u *User) RedactUser() {
    u.EncryptedPassword = ""
}
```

```
var u User
u.RedactUser()
```

Value as receivers vs values passed to a function

```
func RedactUser(u User) {
  u.EncryptedPassword = ""
  return
}
```

```
func (u User) RedactUser() {
    u.EncryptedPassword = ""
}
```

```
func main() {
  u := &User{
     Name:              "name",
     Email:             "name@example.com",
     EncryptedPassword: "AXNAUHEJWNXJANJKLSNIKSNXLKSN!",
  }

  RedactUser(*u)
  u.RedactUser()

  fmt.Println(u)
}
```

Pointer as receivers vs pointers passed to a function

```
func RedactUser(u *User) {
  u.EncryptedPassword = ""
  return
}
```

```
func (u *User) RedactUser() {
    u.EncryptedPassword = ""
}
```

```
func main() {
  u := User{
     Name:              "name",
     Email:             "name@example.com",
     EncryptedPassword: "AXNAUHEJWNXJANJKLSNIKSNXLKSN!",
  }

  RedactUser(&u)
  u.RedactUser()

  fmt.Println(u)
}
```

Stack overflow question

```
type student struct {
    name string
    age  int
}
```

```
func (s *student) update() {
    s.name = "unknown"
    s.age = 0
}
```

```
func main() {
    s := student{"hongseok", 13}
    fmt.Println(s)

    s.update()
    fmt.Println(s)
}
```

Stack overflow question

```
type student struct {
    name string
    age  int
}
```

```
func (s *student) update() {
    s.name = "unknown"
    s.age = 0
}
```

```
func main() {
    s := student{"hongseok", 13}
    fmt.Println(s)

    s.update()
    fmt.Println(s)
}
```

```
{hongseok 13}
{unknown 0}
```

Stack overflow question

```
type student struct {
    name string
    age  int
}
```

```
func (s *student) update() {
    s = &student{"unknown", 0}
}
```

```
func main() {
    s := student{"hongseok", 13}
    fmt.Println(s)

    s.update()
    fmt.Println(s)
}
```

Stack overflow question

```
type student struct {
    name string
    age  int
}
```

```
func (s *student) update() {
    s = &student{"unknown", 0}
}
```

```
func main() {
    s := student{"hongseok", 13}
    fmt.Println(s)

    s.update()
    fmt.Println(s)
}
```

```
{hongseok 13}
{hongseok 13}
```

Stack overflow question

```
type student struct {
    name string
    age  int
}
```

```
func (stu *student) update() {
    stu = &student{"unknown", 0}
}
```

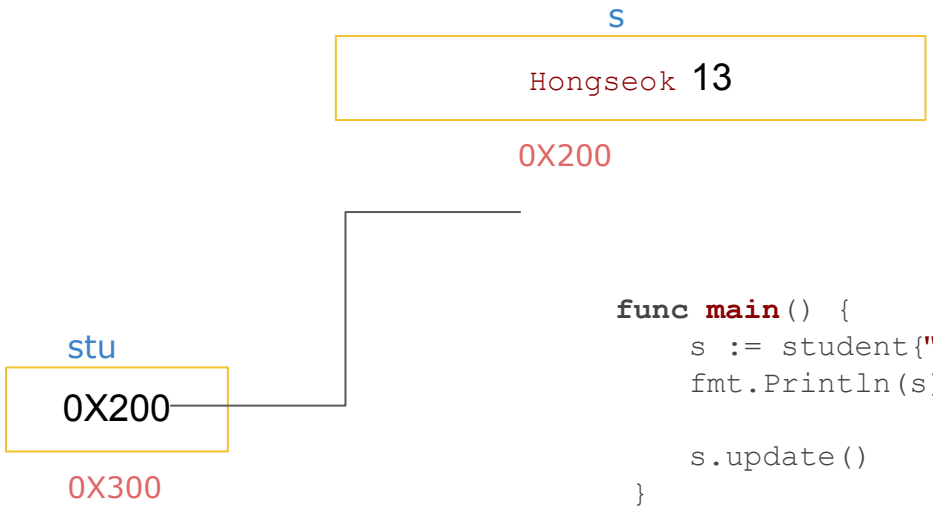
```
func main() {
    s := student{"hongseok", 13}
    fmt.Println(s)

    s.update()
    fmt.Println(s)
}
```

```
{hongseok 13}
{hongseok 13}
```

Stack overflow question


```
func main() {
    s := student{"hongseok", 13}
    fmt.Println(s)
}
```

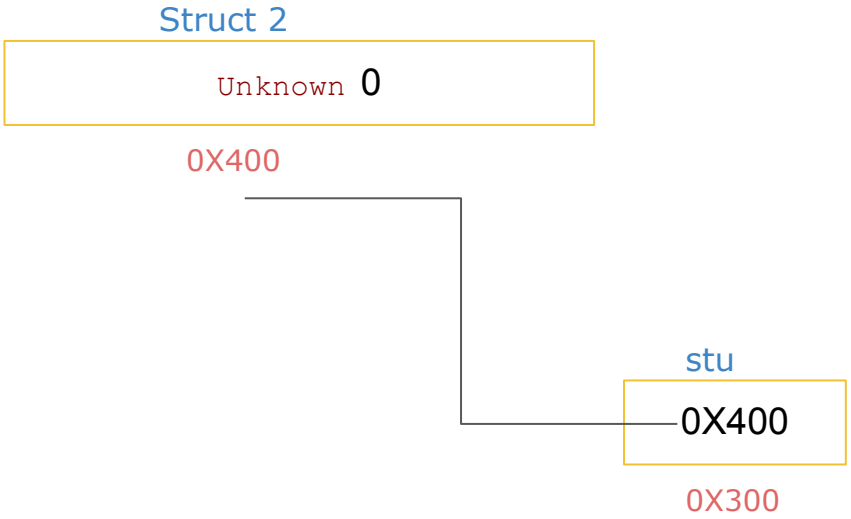
Stack overflow question




```go
func main() {
    s := student{"hongseok", 13}
    fmt.Println(s)

    s.update()
}
```

```go
func (stu *student) update() {

}
```

Stack overflow question




```
func main() {
    s := student{"hongseok", 13}
    fmt.Println(s)

    s.update()
}
```

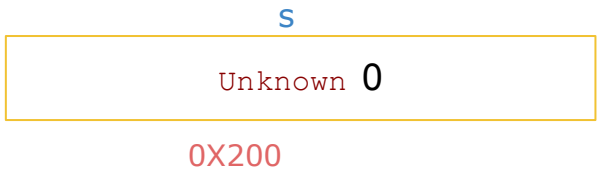
```
func (stu *student) update() {
    stu = &student{"unknown", 0}
}
```

Stack overflow question


```
func main() {
    s := student{"hongseok", 13}
    fmt.Println(s)

    s.update()
    fmt.Println(s)
}
```

```
func (s *student) update() {
    s = &student{"unknown", 0}
}
```

Stack overflow question




```
func main() {
    s := student{"hongseok", 13}
    fmt.Println(s)

    s.update()
    fmt.Println(s)
}
```

```
func (s *student) update() {
    *s = student{"unknown", 0}
}
```

Interfaces

- An interface type is defined as a set of method signatures.

Interfaces

- An interface type is defined as a set of method signatures.
- A value of interface type can hold any value that implements those methods.

Interfaces

- An interface type is defined as a set of method signatures.
- A value of interface type can hold any value that implements those methods.
- There is no explicit declaration of intent, no "implements" keyword.

```
type I interface {
      M()
 }
```

Interfaces

- An interface type is defined as a set of method signatures.
- A value of interface type can hold any value that implements those methods.
- There is no explicit declaration of intent, no "implements" keyword.

```
type I interface {
      M()
 }
```

```
type T struct {
      S string
}
```

Interfaces

- An interface type is defined as a set of method signatures.
- A value of interface type can hold any value that implements those methods.
- There is no explicit declaration of intent, no "implements" keyword.

```
type I interface {
      M()
 }
```

```
type T struct {
      S string
}
```

```
func (t T) M() {
      fmt.Println(t.S)
 }
```

Interfaces

- An interface type is defined as a set of method signatures.
- A value of interface type can hold any value that implements those methods.
- There is no explicit declaration of intent, no "implements" keyword.

```
type I interface {
      M()
 }
```

```
type T struct {
      S string
}
```

```
func (t T) M() {
      fmt.Println(t.S)
 }
```

```
func main() {
      var i I = T{"hello"}
      i.M()
 }
```

Interfaces

the Stringer interface is defined in the fmt package. Its String function is invoked when a type is passed to any of the print functions. We can customize the output message of our own types.

```
type Stringer interface {
    String() string
 }
```

```
func main() {
      u := &User{
            Name:  "name",
            Email: "name@example.com",
            EncryptedPassword: "AXNAUHE",
      }

      fmt.Println(u)

 }
```

```
&{name name@example.com AXNAUHE}
```

Interfaces

the Stringer interface is defined in the fmt package. Its String function is invoked when a type is passed to any of the print functions. We can customize the output message of our own types.

```
type Stringer interface {
    String() string
 }
```

```
func main() {
      u := &User{
            Name:  "name",
            Email: "name@example.com",
            EncryptedPassword: "AXNAUHE",
      }

      fmt.Println(u)

 }
```

Interfaces

the Stringer interface is defined in the fmt package. Its String function is invoked when a type is passed to any of the print functions. We can customize the output message of our own types.

```
type Stringer interface {
    String() string
 }
```

```
func (u User) String() string {
      return "Name = " + u.Name + " \nEmail = " + u.Email
 }
```

```
func main() {
      u := &User{
            Name:  "name",
            Email: "name@example.com",
            EncryptedPassword: "AXNAUHE",
      }

      fmt.Println(u)

 }
```

Interfaces

the Stringer interface is defined in the fmt package. Its String function is invoked when a type is passed to any of the print functions. We can customize the output message of our own types.

```
type Stringer interface {
    String() string
 }
```

```
func (u User) String() string {
      return "Name = " + u.Name + " \nEmail = " + u.Email
 }
```

```
func main() {
      u := &User{
            Name:  "name",
            Email: "name@example.com",
            EncryptedPassword: "AXNAUHE",
      }

      fmt.Println(u)

 }
```

```
Name = name
Email = name@example.com
```

Interfaces

The empty interface

- The interface type that specifies zero methods is known as the *empty interface*.
- An empty interface may hold values of any type.

Interfaces

The empty interface

- The interface type that specifies zero methods is known as the *empty interface*.
- An empty interface may hold values of any type.

```
func main() {
      var i interface{}
      describe(i)

      i = 42
      describe(i)

      i = "hello"
      describe(i)
 }
```

```
func describe(i interface{}) {
      fmt.Printf("(%v, %T)\n", i, i)
 }
```

Interfaces

The empty interface

- The interface type that specifies zero methods is known as the *empty interface*.
- An empty interface may hold values of any type.

```
func main() {
      var i interface{}
      describe(i)

      i = 42
      describe(i)

      i = "hello"
      describe(i)
 }
```

```
(<nil>, <nil>)
(42, int)
(hello, string)
```

```
func describe(i interface{}) {
      fmt.Printf("(%v, %T)\n", i, i)
 }
```

Interfaces

Type Assertion

- A type assertion provides access to an interface value's underlying concrete value.

```
t := i.(T)
```

Interfaces

Type Assertion

- A type assertion provides access to an interface value's underlying concrete value.

```
t := i.(T)
```

- If i does not hold a T, the statement will trigger a panic.

Interfaces

Type Assertion

- A type assertion provides access to an interface value's underlying concrete value.

```
t := i.(T)
```

- If i does not hold a T, the statement will trigger a panic.

```
t, ok := i.(T)
```

- To test whether an interface value holds a specific type, a type assertion can return two values.(does not trigger panic)

Interfaces

```
func main() {

	var x interface{} = "foo"

	var s string = x.(string)
	fmt.Println(s)

	s, ok := x.(string)
	fmt.Println(s, ok)

	n, ok := x.(int)
	fmt.Println(n, ok)

	n = x.(int)

}
```

Interfaces

```
func main() {

	var x interface{} = "foo"

	var s string = x.(string)
	fmt.Println(s)

	s, ok := x.(string)
	fmt.Println(s, ok)

	n, ok := x.(int)
	fmt.Println(n, ok)

	n = x.(int)

}
```

```
foo
foo true
0 false


panic: interface conversion: interface
{} is string, not int
```

Interfaces

Type Switches

A type switch is a construct that permits several type assertions in series.

```
switch v := i.(type) {
case T:
    // here v has type T
case S:
    // here v has type S
default:
    // no match; here v has the same
type as i
 }
```

Interfaces

```
func do(i interface{}) {
      switch v := i.(type) {
      case int:
            fmt.Printf("Twice %v is %v\n", v, v*2)
      case string:
            fmt.Printf("%q is %v bytes long\n", v, len(v))
      default:
            fmt.Printf("I don't know about type %T!\n", v)
      }
 }
```

```
func main() {
      do(21)
      do("hello")
      do(true)
 }
```

Interfaces

```
type Abser interface {
      Abs() float64
 }
```

Interfaces

```
type Abser interface {
      Abs() float64
 }
```

```
type MyFloat float64

func (f MyFloat) Abs() float64 {
      if f < 0 {
            return float64(-f)
      }
      return float64(f)
 }
```

```
type Vertex struct {
      X, Y float64
}

func (v *Vertex) Abs() float64 {
      return math.Sqrt(v.X*v.X +
v.Y*v.Y)
 }
```

Interfaces

```
type Abser interface {
      Abs() float64
 }
```

```
type MyFloat float64

func (f MyFloat) Abs() float64 {
      if f < 0 {
            return float64(-f)
      }
      return float64(f)
 }
```

```
type Vertex struct {
      X, Y float64
}

func (v *Vertex) Abs() float64 {
      return math.Sqrt(v.X*v.X +
v.Y*v.Y)
 }
```

```
func main() {
      var a Abser
      f := MyFloat(-math.Sqrt2)
      v := Vertex{3, 4}

      a = f  // a MyFloat implements Abser
      a = &v // a *Vertex implements Abser

      // In the following line, v is a Vertex (not *Vertex)
      // and does NOT implement Abser.
      a = v

      fmt.Println(a.Abs())
}
```

Why….

As the Go specification says, the method set of a type `T` consists of all methods with receiver type `T`, while that of the corresponding pointer type `*T` consists of all methods with receiver `*T` or `T`. That means the method set of `*T` includes that of `T`, but not the reverse.

This distinction arises because if an interface value contains a pointer `*T`, a method call can obtain a value by dereferencing the pointer, but if an interface value contains a value `T`, there is no safe way for a method call to obtain a pointer. (Doing so would allow a method to modify the contents of the value inside the interface, which is not permitted by the language specification.)

THANK YOU