DIGITALHOUSE

Interfaces

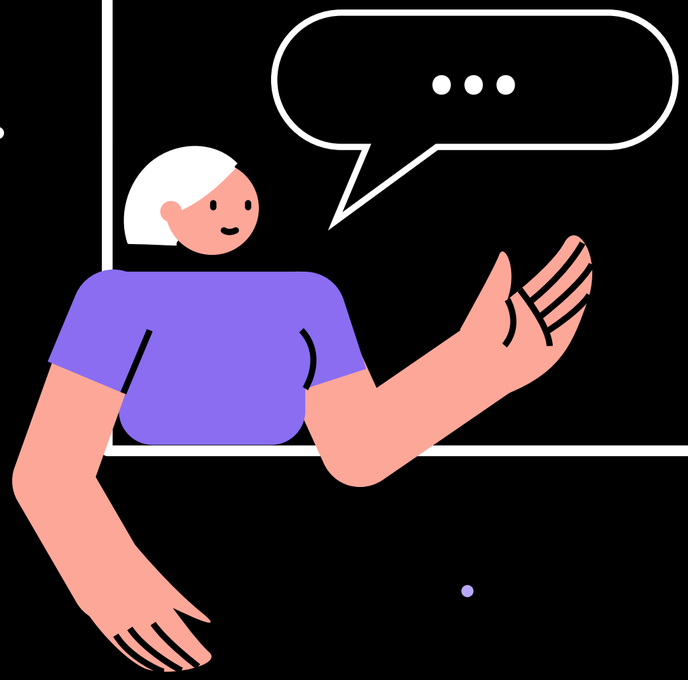
Certified Tech Developer
The Ultimate Degree

Índice

01

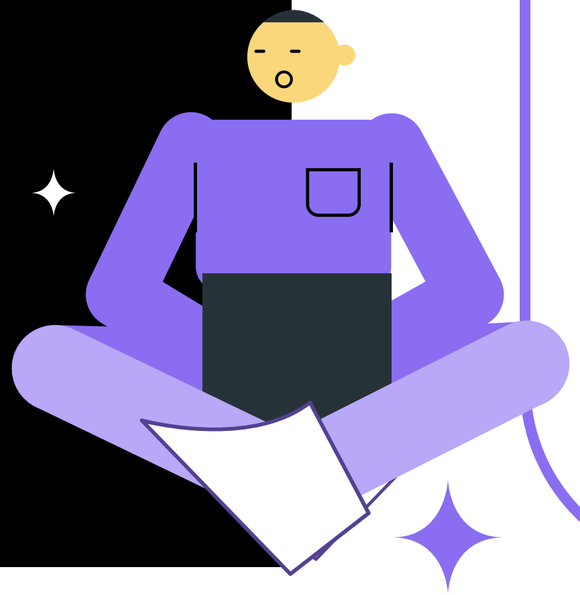
¿Qué son las interfaces?

Una **interfaz** es una forma de definir métodos que deben ser utilizados, pero sin definirlos.



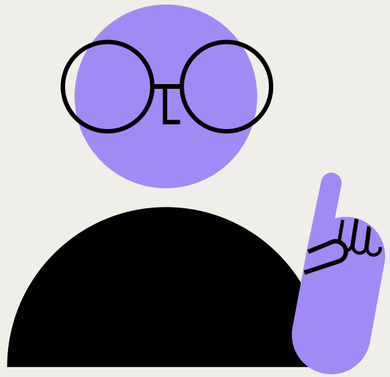
¿Para qué se utilizan?

Las interfaces son utilizadas para brindar modularidad al lenguaje.



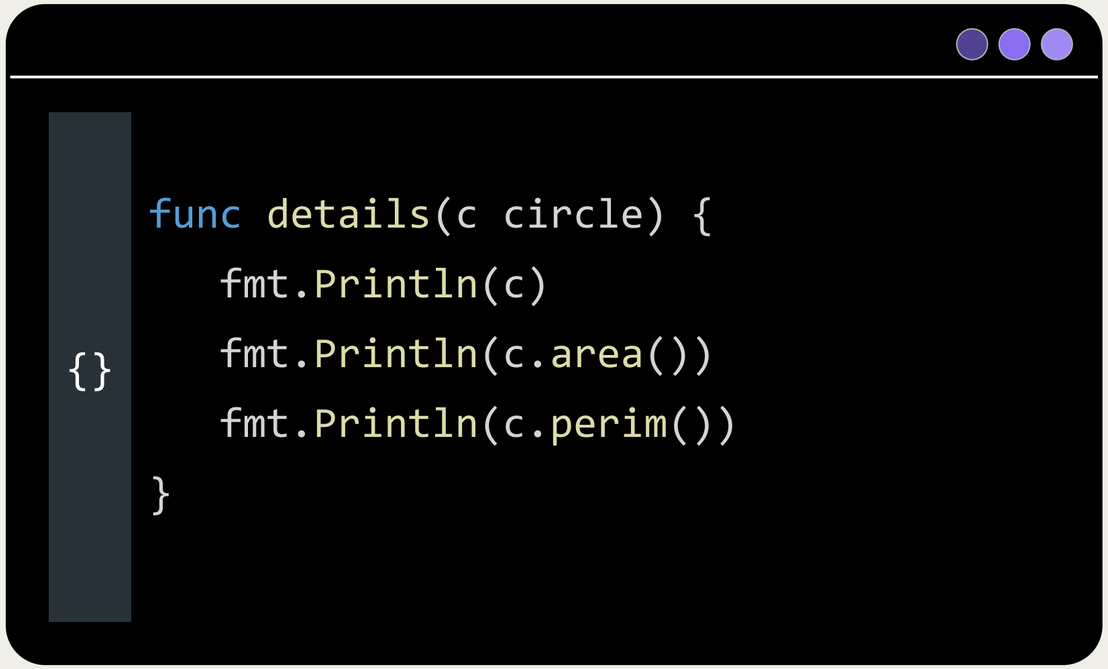
Ejemplo #1

Generamos una estructura **circle** y las funciones que mostrarán el área y el perímetro de la figura:



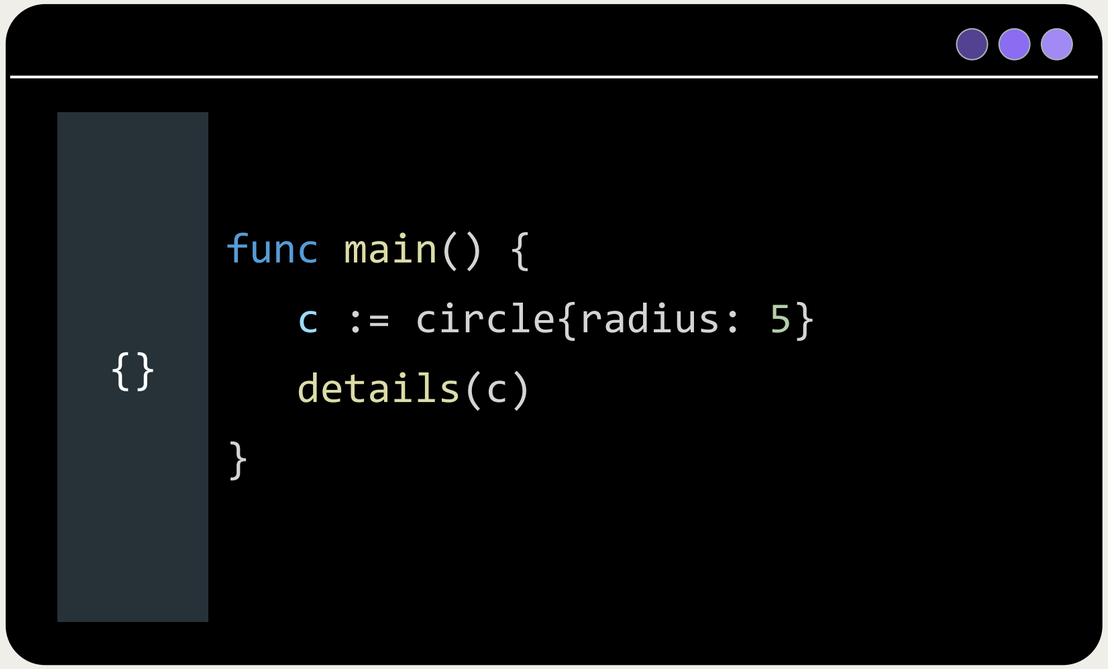
```
{}
type circle struct {
  radius float64
}

func (c circle) area() float64 {
  return math.Pi * c.radius * c.radius
}

func (c circle) perim() float64 {
  return 2 * math.Pi * c.radius
}
```

Creamos una función **details** que imprima el área y el perímetro que generamos para dicho objeto:



```
func details(c circle) {
    fmt.Println(c)
    fmt.Println(c.area())
    fmt.Println(c.perim())
}
```

Y ejecutaremos la función:



```
func main() {
    c := circle{radius: 5}
    details(c)
}
```

¿Qué pasará si queremos generar más figuras geométricas utilizando nuestra **función details**?

Aquí es donde entran en juego las interfaces. Estas nos permiten implementar el mismo comportamiento a diferentes objetos.

Ejemplo #2

A continuación, definimos nuestra interfaz **geometry** que contiene dos métodos que adoptarán nuestros objetos:

```
{}
type geometry interface {
    area() float64
    perim() float64
}
```

Generamos otro objeto geométrico. En este caso, un rectángulo que —lógicamente— tenga los mismos métodos:

```
{}
type rect struct {
    width, height float64
}
func (r rect) area() float64 {
    return r.width * r.height
}
func (r rect) perim() float64 {
    return 2*r.width + 2*r.height
}
```

Modificaremos nuestra función **details** para que, en lugar de recibir un círculo, reciba una figura geométrica:

```
{}
func details(g geometry) {
    fmt.Println(g)
    fmt.Println(g.area())
    fmt.Println(g.perim())
}
```

De esta forma podemos seguir agregando figuras geométricas sin necesidad de modificar nuestra función:

```
{}
func main() {
    c := newCircle(2)
    fmt.Println(c.area())
    fmt.Println(c.perim())
}
```

Ejemplo #3

En el siguiente ejemplo, creamos una función que nos genere el objeto:

```
{}
func newCircle(values float64)
circle {
    return circle{radius: values}
}
```

Ejecutamos el **main** del programa:

```
{}
func main() {
    c := newCircle(2)
    fmt.Println(c.area())
    fmt.Println(c.perim())
}
```

¿Qué pasará si queremos reutilizar nuestra función para poder implementar varias figuras geométricas?

En este caso tendremos que crear una función que retorne una interfaz que pueda implementar todos nuestros objetos geométricos.

Ejemplo #4

Vamos a reemplazar nuestra función **newCircle** por **newGeometry** y le pasaremos dos constantes que definimos para especificar cuál es el objeto que generamos:

```
{}
```

```
const (
    rectType   = "RECT"
    circleType = "CIRCLE"
)
func newGeometry(geoType string, values ...float64) geometry {
    switch geoType {
    case rectType:
        return rect{width: values[0], height: values[1]}
    case circleType:
        return circle{radius: values[0]}
    }
    return nil
}
```

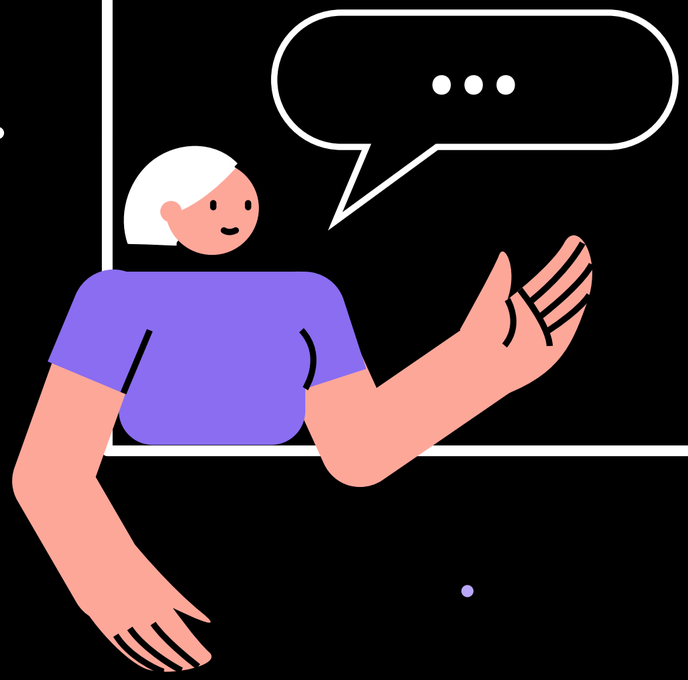
Implementamos en nuestro **main** y corremos el programa:

```
{}
```

```go
func main() {
    r := newGeometry(rectType, 2, 3)
    fmt.Println(r.area())
    fmt.Println(r.perim())
    c := newGeometry(circleType, 2)
    fmt.Println(c.area())
    fmt.Println(c.perim())
}
```

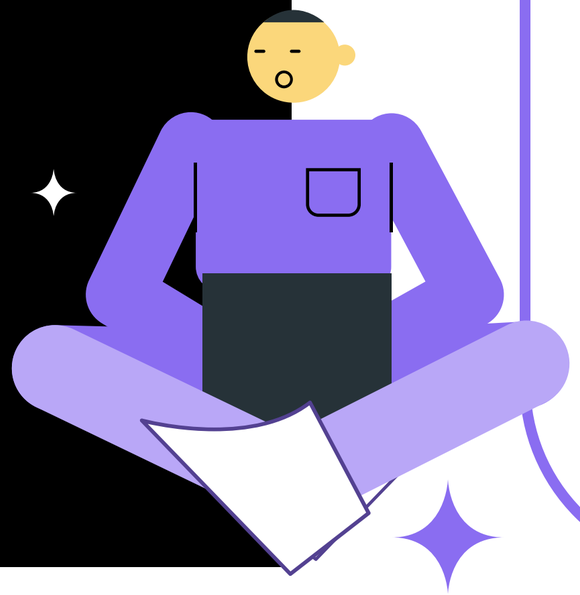
02 Interfaces vacías

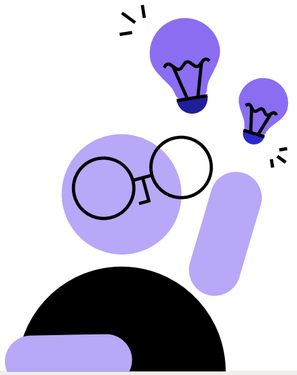
Son aquellas interfaces que no tienen métodos declarados.



¿Para qué se utilizan?

La utilidad de estas interfaces es proveernos un tipo de datos “comodín”. Es decir, almacenar valores que sean de un tipo de datos desconocido, o que pueda variar dependiendo el flujo del programa.





¿Cómo se declara una variable con este tipo?

```
{}  var miVariable interface{}
```

¿Cómo funciona?

Como vimos anteriormente, una interfaz define el conjunto mínimo de métodos que un tipo de datos debe implementar para poder ser considerado como implementador de dicha interfaz. Por lo tanto, todos los tipos de datos son considerados implementadores de la interfaz vacía, porque implementan al menos cero métodos.

Ejemplo de un uso de interfaz vacía

{}

```go
type ListaHeterogenea struct {
   Data []interface{}
}

func main() {
   l := ListaHeterogenea{}
   l.Data = append(l.Data, 1)
   l.Data = append(l.Data, "hola")
   l.Data = append(l.Data, true)

   fmt.Printf("%v\n", l.Data)

}
```

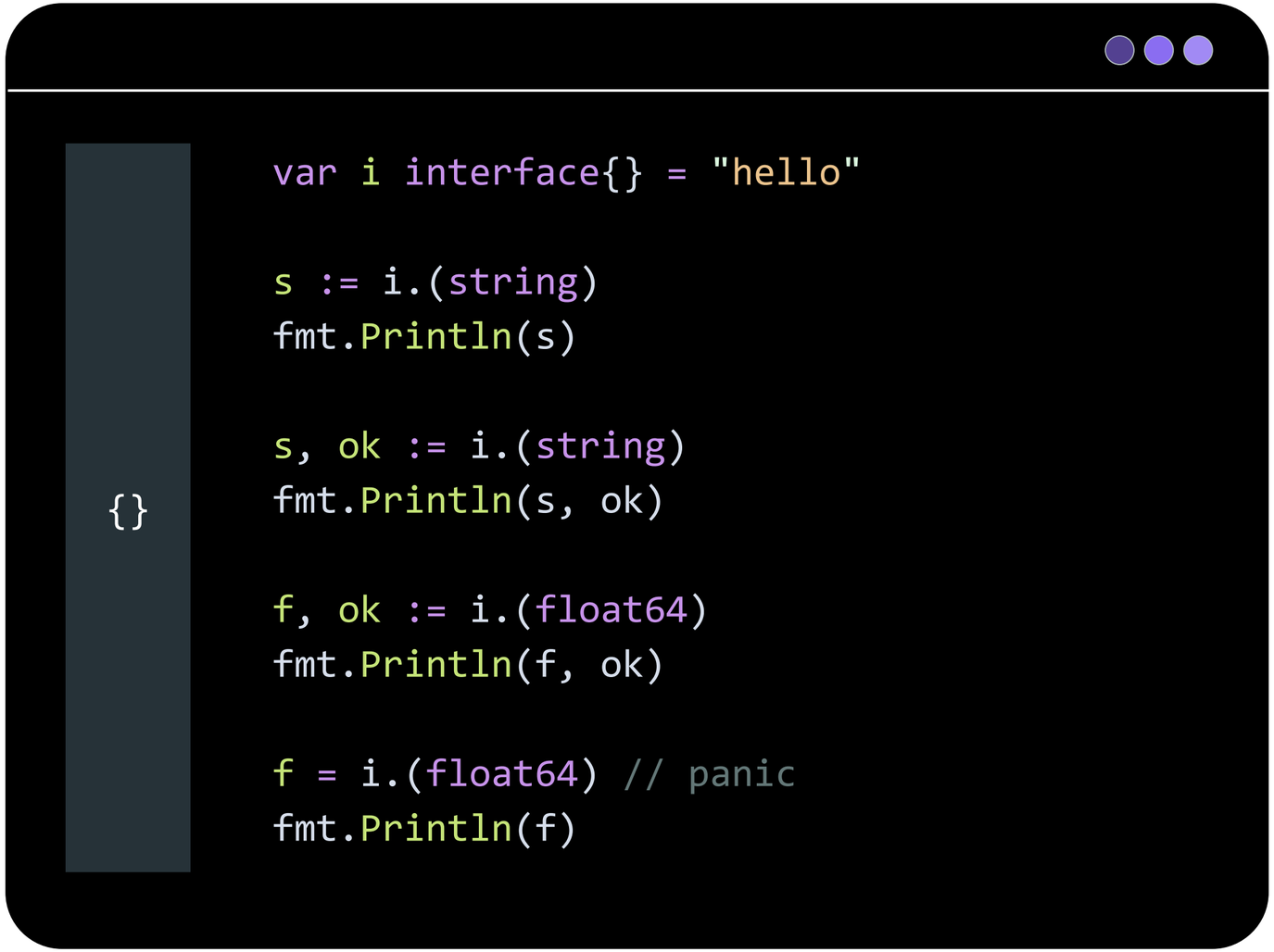
output

```
> $ go run interfaces_vacias.go
[1 hola true]
```

03 Type assertion

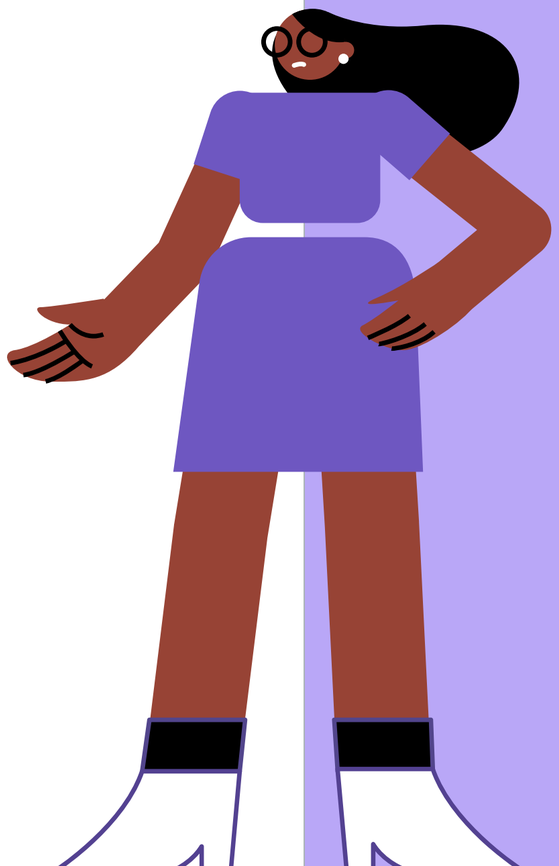
Type assertion (aserción de tipos)

La aserción de tipos provee acceso al tipo de datos exacto que está abstraído por una interfaz.



Conclusiones

En esta clase aprendimos el concepto de las interfaces y su aplicación dentro de Go. Estas nos proveen modularidad al momento de desarrollar nuestras aplicaciones.



DIGITALHOUSE

¡Muchas gracias!