PROPIEDADES EN KOTLN

¿Qué es una propiedad en kotlin?

En la POO tradicional el término propiedad se refiere a lo que conocemos por "atributo" , "field" o "campo". En kotlin es un concepto un poco más amplio pues engloba: un campo, su función accesor get y su función mutador set. Los conceptos de get y set son idénticos en Kotlin que en c++ o Java, en kotlin simplemente se hace más automática sintácticamente la relación entre un campo y sus típicos get/set

accesores: getters y setters

Vimos en ejemplos previos que para declarar una propiedad simplemente usábamos var/val como para las variables locales de las funciones. Si una variable con var/val se define en el cuerpo de la clase fuera de toda función o en el constructor primario ya se constituye en propiedad de la clase. Pero realmente una propiedad kotlin es algo más. Esta es la sintáxis completa de declaración de una propiedad:

```
var <propertyName>[: <PropertyType>] [= <property_initialize
r>]
    [<getter>]
    [<setter>]
```

Fíjate que en la sintaxis anterior, que en la definición de la propiedad si queremos podemos incluir un get/set asociado a la propiedad de forma muy concisa y compacta Veamos un ejemplo que incluye los get y set

In [11]:
```
class Persona {
    var nombre:String ="chuly"

    // getter
    get() =field +  " Soy get()"

    // setter
    set(value) {
        field = value + " metido por set"
    }
}
val  p=  Persona()
println(p.nombre)
p.nombre="rosky"
println(p.nombre)
```

```
chuly Soy get()
rosky metido por set Soy get()
```

la keyword field y el concepto de back field en kotlin

Simplificadamente una propiedad podemos resumirla con la siguiente fórmula *propiedad* =

valor + set +get. Así que en kotlin una propiedad es algo más que un valor. Internamente, de alguna manera se tiene que almacenar este valor y esto se hace a través una variable *interna* que se llama *back field*, Al escribir los métodos get/set a menudo querremos acceder a este valor y esto se hace con la palabra reservada **field**. Es decir, al *back field* se accede con la palabra reservada *field*. Lógicamente esta palabra sólo tiene sentido dentro de la declaración de una propiedad.

Si en el ejemplo anterio en lugar de field hubieramos usado el nombre de la propiedad se habría producido un efecto recursivo indeseado

Accesores Por Defecto

Si al declarar una propiedad no especificamos accesores, kotlin crea unos por defecto. Por ejemplo:

```
class Persona{

    var nombre = "chosky"
}
```

Equivale a definir:

```
class Persona{

    var nombre = "chosky"
        get() = field
        set(value) {
            field = value
        }
}
```

A menudo los accesores por defecto son más que suficientes y no necesitamos por lo tanto escribirlos salvo que precisemos personalizarlos.

relación entre var/val y accesores

Recuerda que val genera variables no modificables, por tanto, cuando declaras una propiedad con val, solo vamos a poder personalizar el get ya que el set no es accesible. Por tanto:

val => campo + get

var => campo + get + set

Visibilidad de accesores

Más adelante estudiaremos los modificadores de visibilidad con más detenimiento. Por el momento observamos que uno de esos modificadores es *private*. Se puede aplicar *private* a un *set* para prohibir su uso fuera de la clase.

In [17]:
```
class Persona{

    var nombre = "chosky"
        private set
}
val p=Persona()
println(p.nombre)
p.nombre="Rusky" //error set es private
```

```
chosky
```

En el siguiente ejemplo vemos un caso de porqué puede resultar interesante hacer private el set de una propiedad.

Tenemos una clase que encapsula unas coordenadas x, y. No queremos definir como val las propiedades porque queremos poder cambiar su valor, pero por otro lado, no queremos poder cambiar directamente las coordenadas en una instrucción de asignación ya que queremos forzar a usar las funciones miembro de la clase para cambiar las coordenadas.

In [19]:
```
class Coordenadas {
    var x: Int = 0
        private set
    var y: Int = 0
        private set

    fun moveLeft() {
        x -= if (x == 0) 0 else 1
    }

    fun moveRight() {
        x += if (x == 300) 0 else 1
    }

    fun moveUp() {
        y -= if (y == 0) 0 else 1
    }

    fun moveDown() {
        y += if (y == 300) 0 else 1
    }
}
val c= Coordenadas()
c.moveLeft()
//c.x=77//error
```

campos calculados.

En lenguajes como java o c++ no solía quererse definir un atributo que su valor dependía de otros atributos. Por ejemplo el área de un rectangulo depende de el ancho y alto, si cambia el ancho y el alto cambia el área, por esta razón en estos lenguajes se prefiere usar un método/funcion area() que calcule el valor del área para evitar almacenar el valor del área, simplemente, cada vez que se requiera se calcula invocando al método/función.

En kotlin en cambio si que tiene sentido definir una propiedad área quer realmente

encapsula un método que hace el cálculo pero el resultado final es que obtenemos un objeto con una riqueza semántica al objeto que hace que se asemaje más a los objetos de la realidad, ya que, efectivamente en la realidad nos gusta ver el area como una propiedad de un rectángulo, no sólo como un cálculo.

In [3]:
```
class Rectangle(val width: Int, val height: Int) {
    val area: Int // property type is optional since it can be infer
        get() = this.width * this.height
}
val mirectangulo= Rectangle(2,3)
print(mirectangulo.area)
```

```
6
```

Backing properties

La keyword *field* sólo es posible usarla dentro de los accesores get/set. Por lo tanto, ya que solo los get/set son capaces de de usar field sólo los get/set son capaces de acceder directamente al valor de una propiedad. Ni siquiera otros métodos de la propia clase pueden acceder directamente al valor, se ven obligados a acceder a través de los get/set. Observa el siguiente ejemplo. Tienes que tener claro que *imprimirNombre()* no está usando el field de *nombre*, está usando el get() de *nombre*. Esto realmente ya fue visto más arriba en este libro.

In [4]:
```
class Persona {
    var nombre ="chuly"

        // getter
        get() =field +  " Soy get()"
    fun imprimirNombre() = println(nombre)

}
val  p=  Persona()
p.imprimirNombre()
```

```
chuly Soy get()
```

Usar una back property asociada a una property

Si dentro de una clase escribimos una serie de métodos, es habitual querer que dichos métodos puedan acceder al valor de una propiedad "saltándose el filtro" de set/get". Para conseguir esto, podemos usar una un segunda propiedad de respaldo de la primera que llamamos *Back property*. Usaremos está *back property* para trabajar de forma asociada con la propiedada a la que respalda. Por convenio a una *Back property* debemos declararla con un nombre igual que el de la propiedad a la que queremos respaldar pero comenzando con un guión bajo. El guión bajo es una norma de estilo que advierte que no se debe acceder a esta propiedad desde fuera de la clase pero no lo evita. Si queremos que el acceso no se produzca debemos además de añadir el modificador de visibilidad *private*.

In [4]:
```
class Persona {
    private var _nombre="chuly" //_nombre va a funcionar como propie
    var nombre:String
        // getter
        get() =_nombre +  " Soy get()"
        //setter
        set(value){
            _nombre=value.uppercase() //con set() obligamos a almace
        }
    fun imprimirNombre() = println(_nombre)

}
val  p=  Persona()
p.imprimirNombre()
println(p.nombre)
//println(p._nombre) //¡ERROR!
p.nombre="Zurky"
p.imprimirNombre()
println(p.nombre)
```

```
chuly
chuly Soy get()
ZURKY
ZURKY Soy get()
```

Si observas el ejemplo de arriba, cuando trabajamos con un propiedad de respaldo ¿Quién va a acceder a dicha propiedad?:

- los métodos de la clase que quieren a un valor "original "sin filtros get/set
- los set/get de la propiedad asociada para mantener la lógica de valor original y valor filtrado. En muchas situaciones al trabajar con back property los set/get usarán esta back property en lugar de *field*. Todo depende de la lógica deseada y recuerda que cuando "se mete la lógica por medio" suele haber muchas soluciones o enfoques equivalentes.