Lecture 4

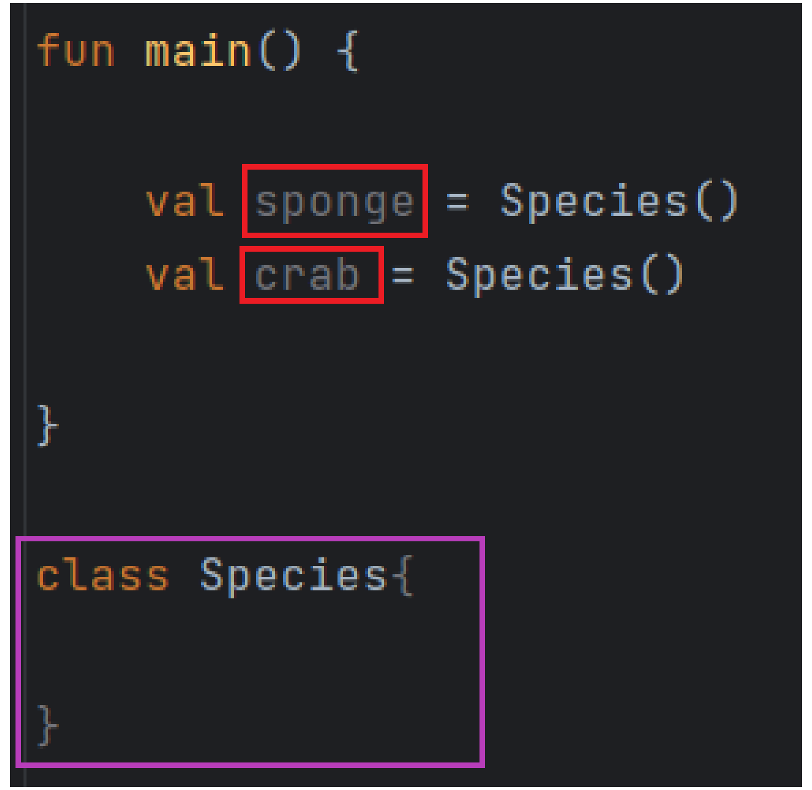
COMP 3717- Mobile Dev with Android Tech

第4讲

COMP 3717 - 使用Android技术进行移动开发

Classes & objects

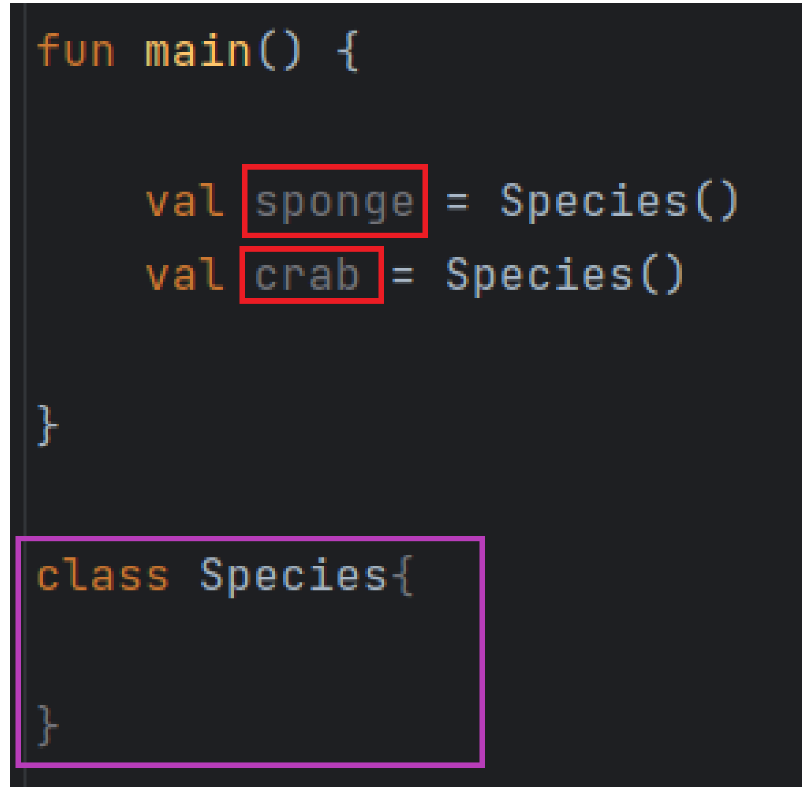
- Like other OOP languages, kotlin uses classes and objects



```
fun main() {

    val sponge = Species()
    val crab = Species()

}

class Species{

}
```

类与对象

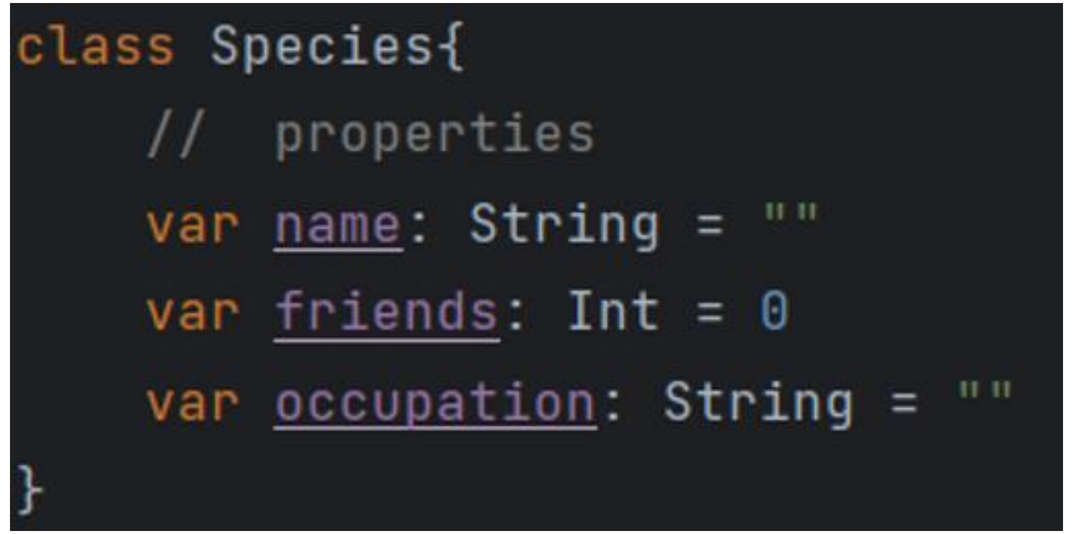
- 与其他面向对象编程语言一样，Kotlin 使用 类和 对象



```
fun main() {

    val sponge = Species()
    val crab = Species()

}

class Species{

}
```

Classes & objects (cont.)

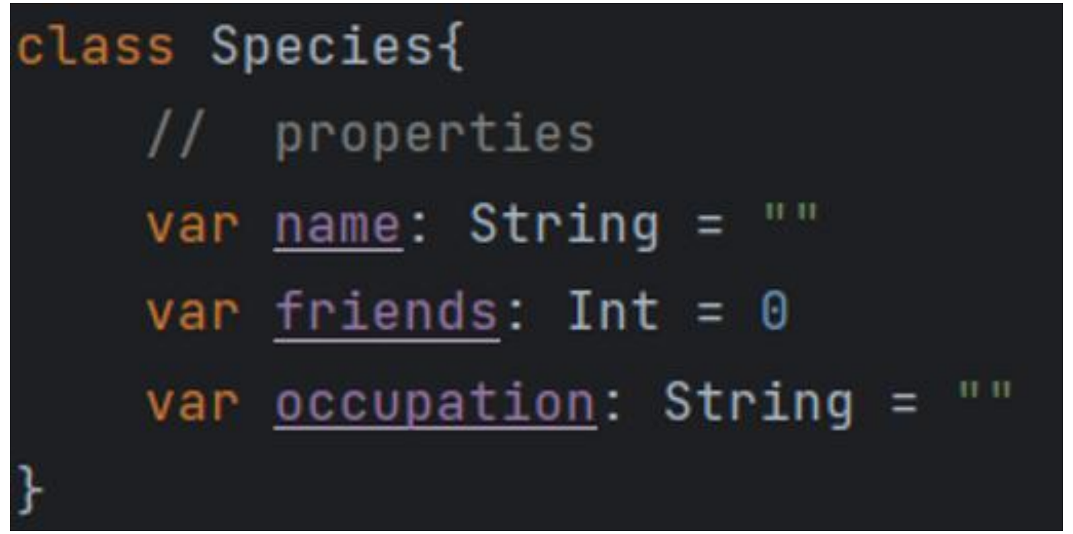
- We define properties (aka. state) within a class



```
class Species{
    //  properties
    var name: String = ""
    var friends: Int = 0
    var occupation: String = ""
}
```

类与对象（续）

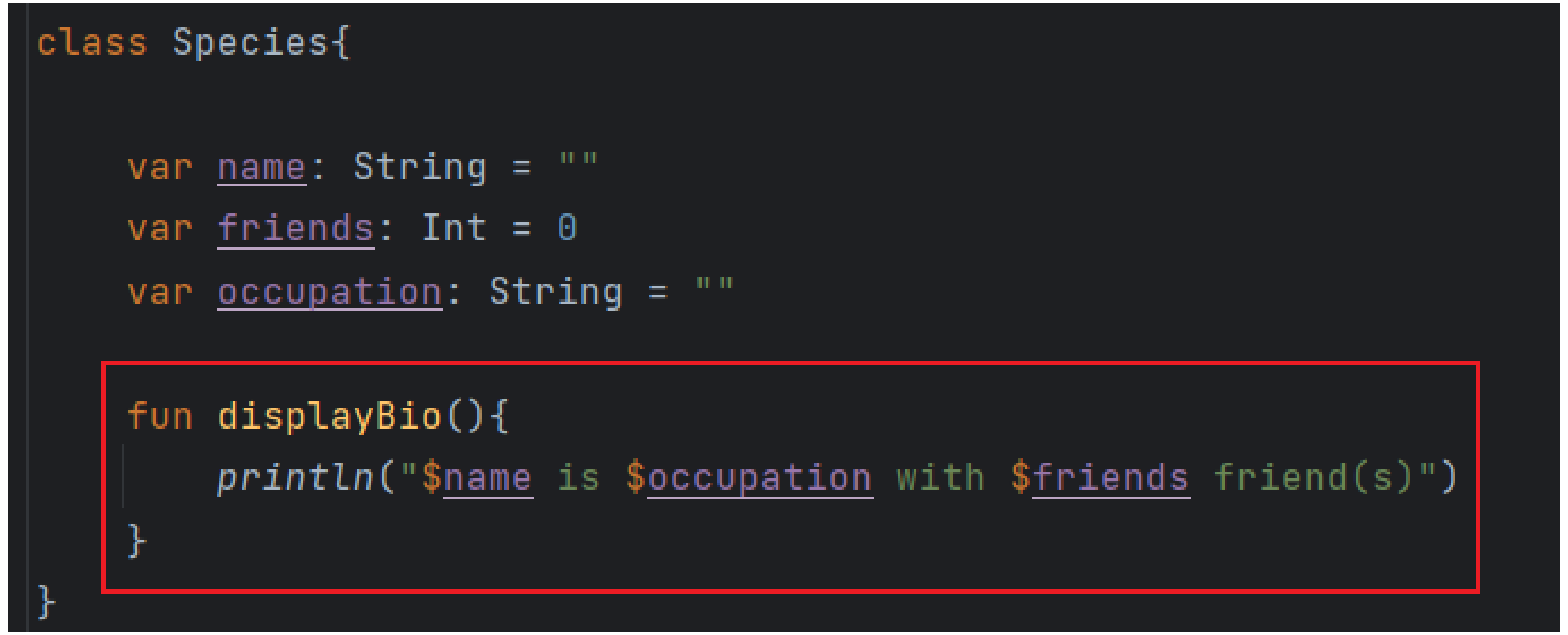
- 我们在类中定义属性（即状态）



```
class Species{
    //  properties
    var name: String = ""
    var friends: Int = 0
    var occupation: String = ""
}
```

Methods

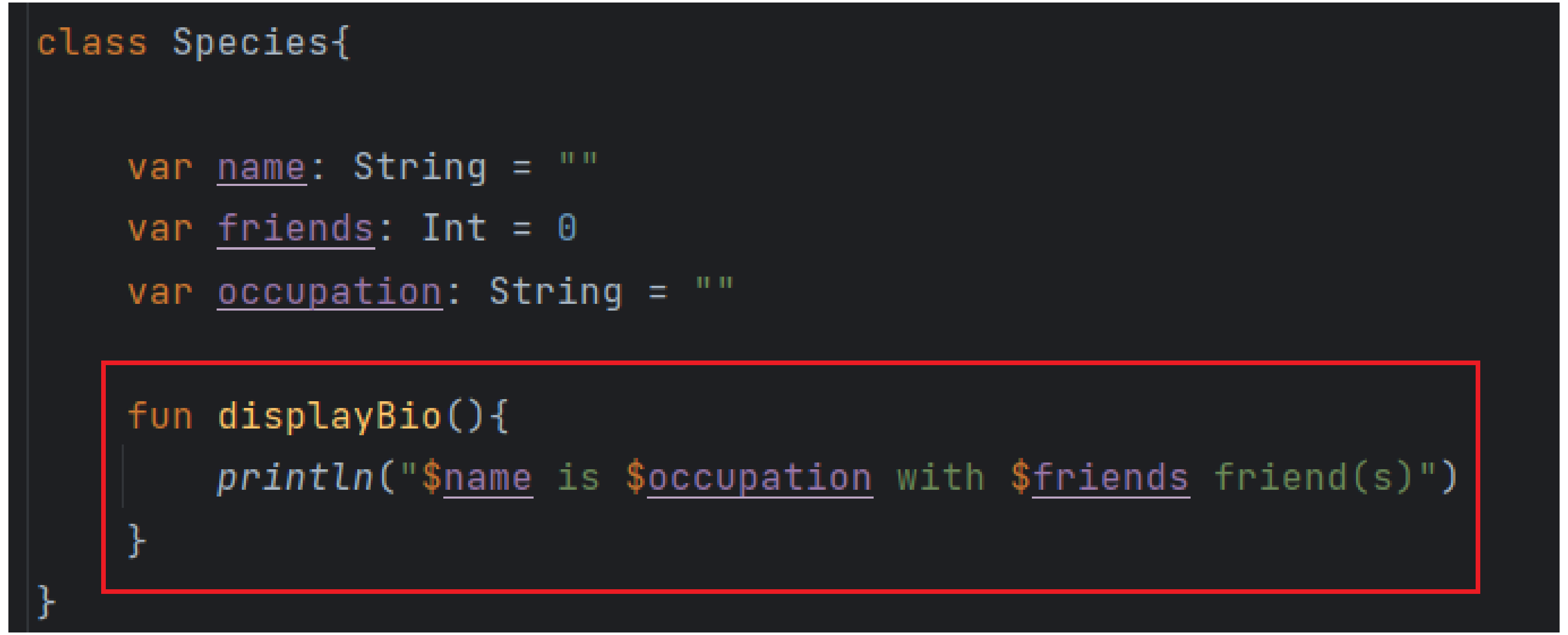
- Classes also have methods



```
class Species{

    var name: String = ""
    var friends: Int = 0
    var occupation: String = ""

    fun displayBio(){
        println("$name is $occupation with $friends friend(s)")
    }
}
```

方法

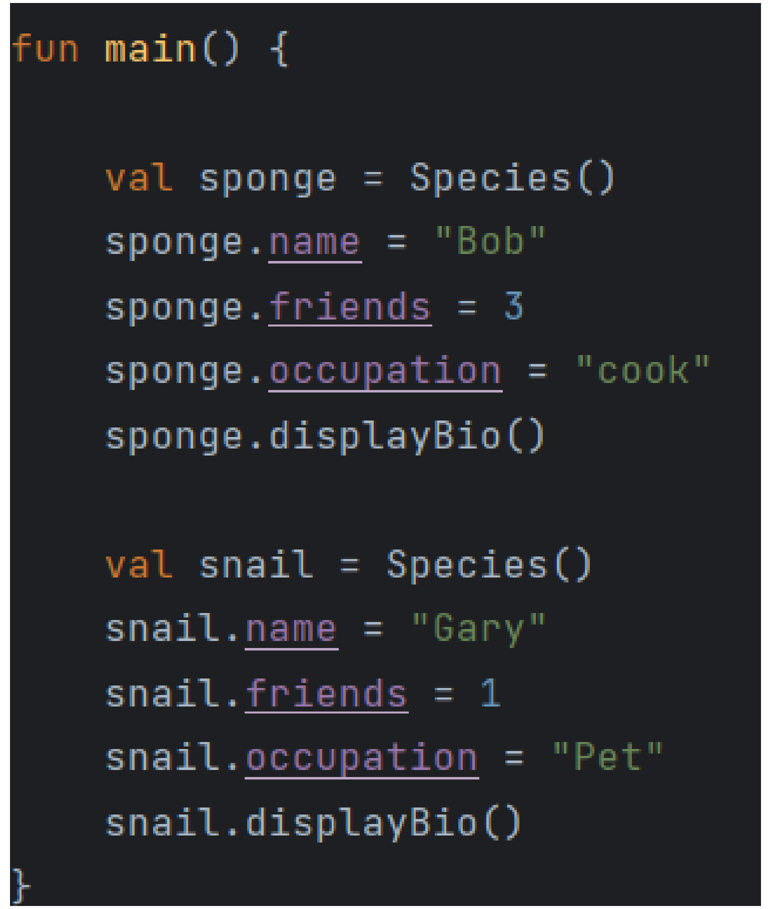
- 类还具有 方法



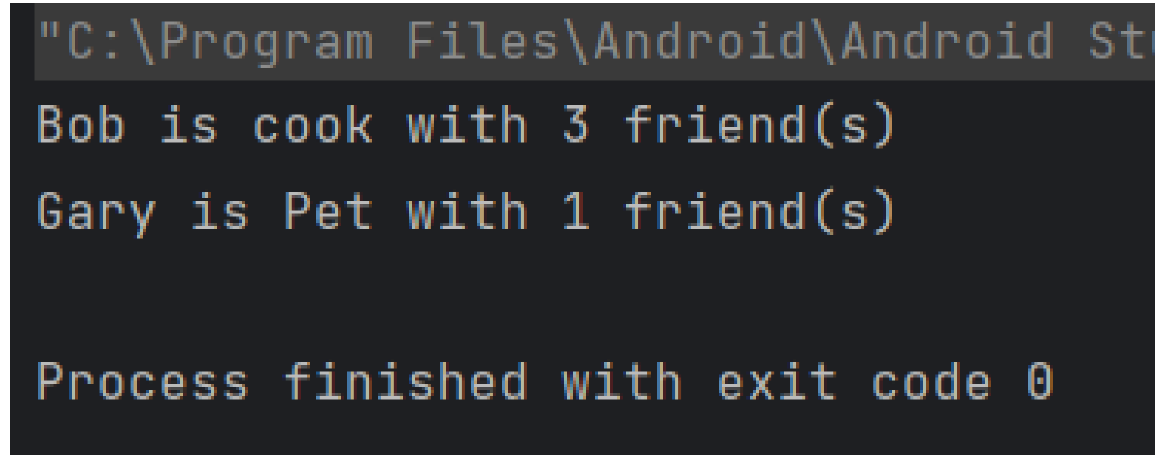
```
class Species{

    var name: String = ""
    var friends: Int = 0
    var occupation: String = ""

    fun displayBio(){
        println("$name is $occupation with $friends friend(s)")
    }
}
```

Methods (cont.)

- Properties and methods are specific to the object instance



```
fun main() {

    val sponge = Species()
    sponge.name = "Bob"
    sponge.friends = 3
    sponge.occupation = "cook"
    sponge.displayBio()

    val snail = Species()
    snail.name = "Gary"
    snail.friends = 1
    snail.occupation = "Pet"
    snail.displayBio()
}
```

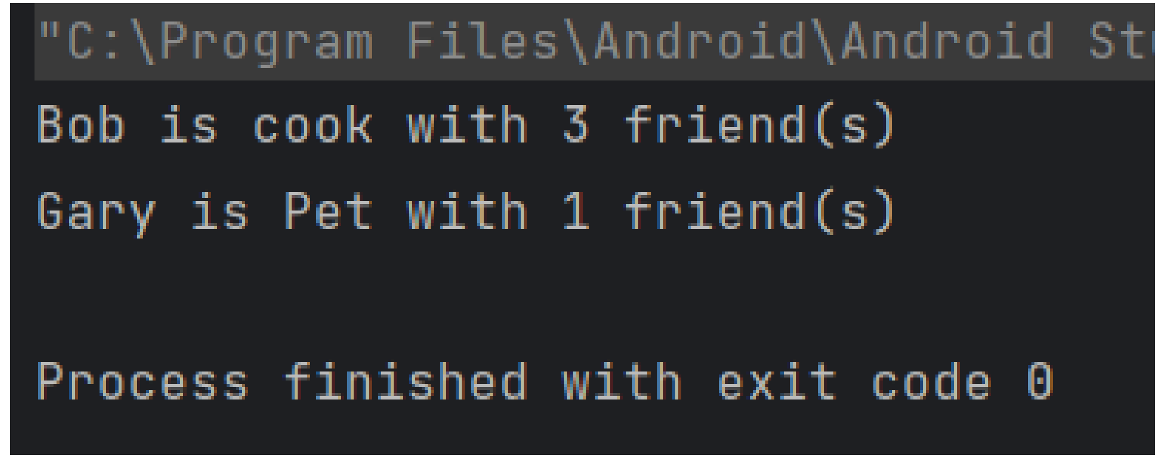


```
"C:\Program Files\Android\Android St
Bob is cook with 3 friend(s)
Gary is Pet with 1 friend(s)

Process finished with exit code 0
```

方法（续）

- 属性和方法特定于对象实例

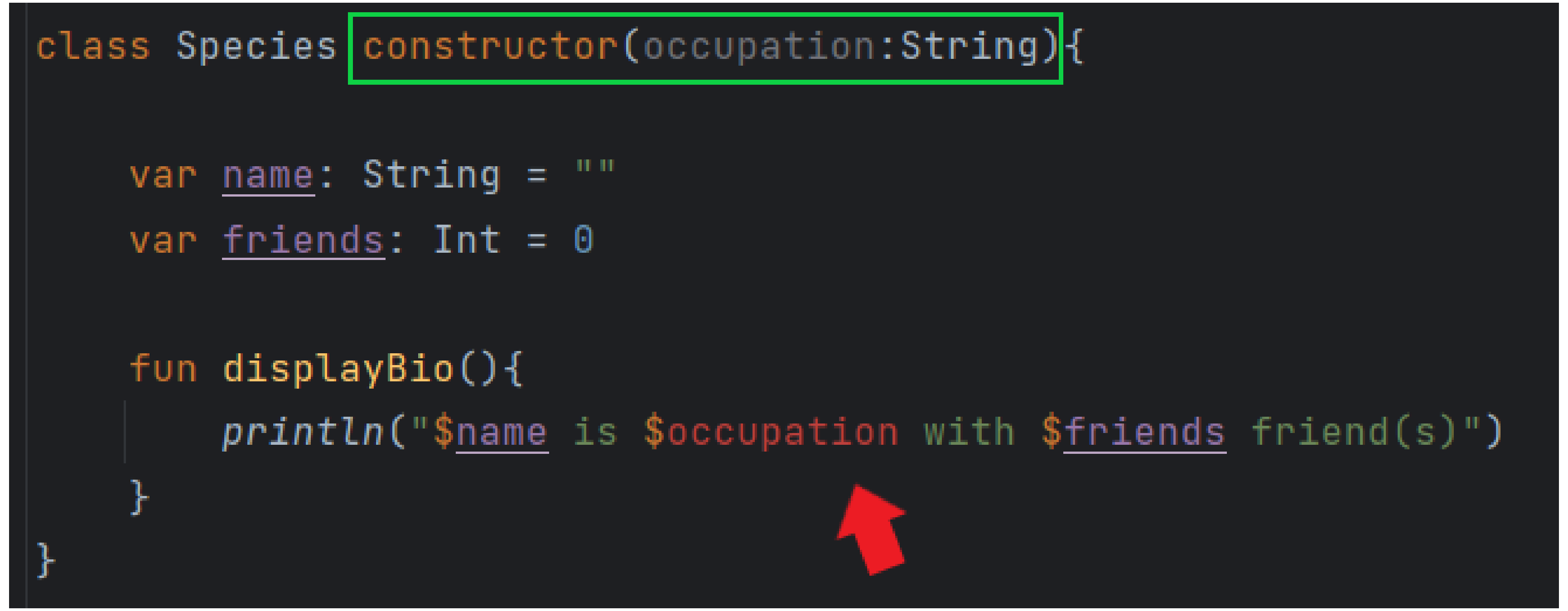
```
fun main() {

    val sponge = Species()
    sponge.name = "Bob"
    sponge.friends = 3
    sponge.occupation = "cook"
    sponge.displayBio()

    val snail = Species()
    snail.name = "Gary"
    snail.friends = 1
    snail.occupation = "Pet"
    snail.displayBio()
}
```



```
"C:\Program Files\Android\Android St
Bob is cook with 3 friend(s)
Gary is Pet with 1 friend(s)

Process finished with exit code 0
```

Constructors

- To pass arguments into our class we use parameters in a constructor

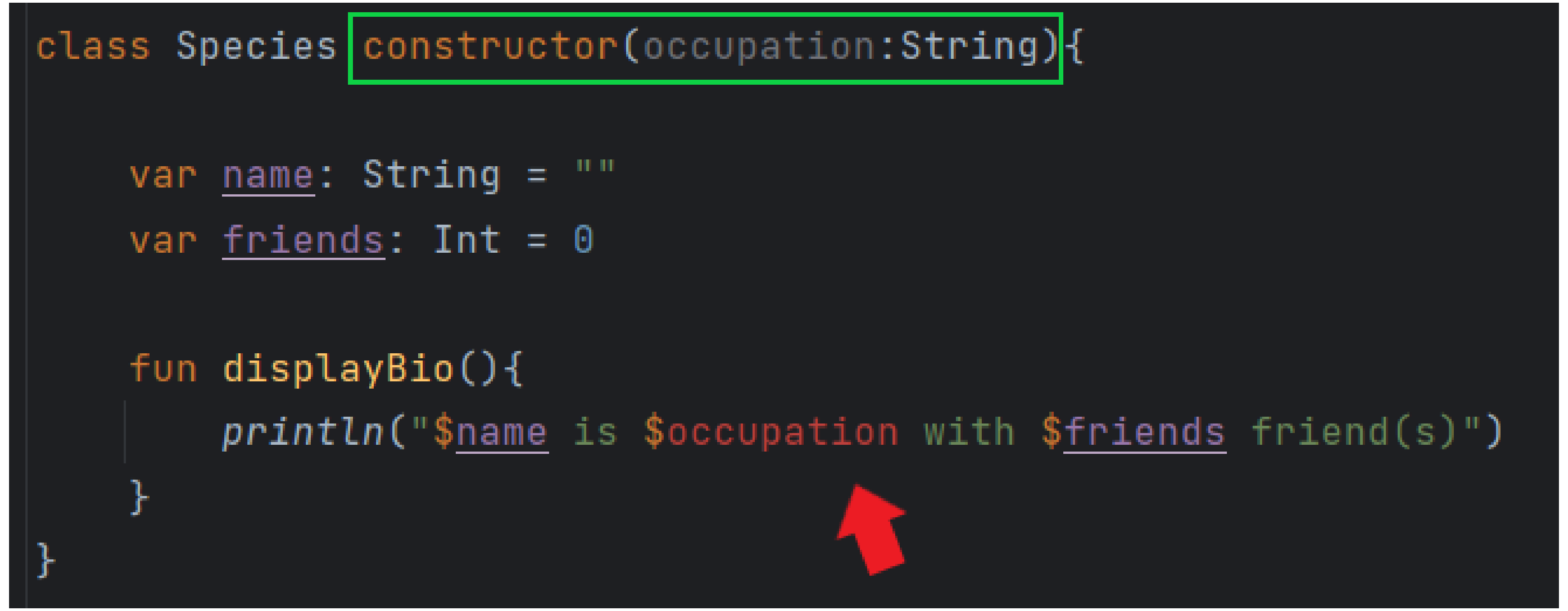


```
class Species constructor(occupation:String){

    var name: String = ""
    var friends: Int = 0

    fun displayBio(){
        println("$name is $occupation with $friends friend(s)")
    }
}
```

- Notice we can't use the constructor parameters in methods

构造函数

- 要将参数传递给我们的类，我们会在 构造函数 中使用参数

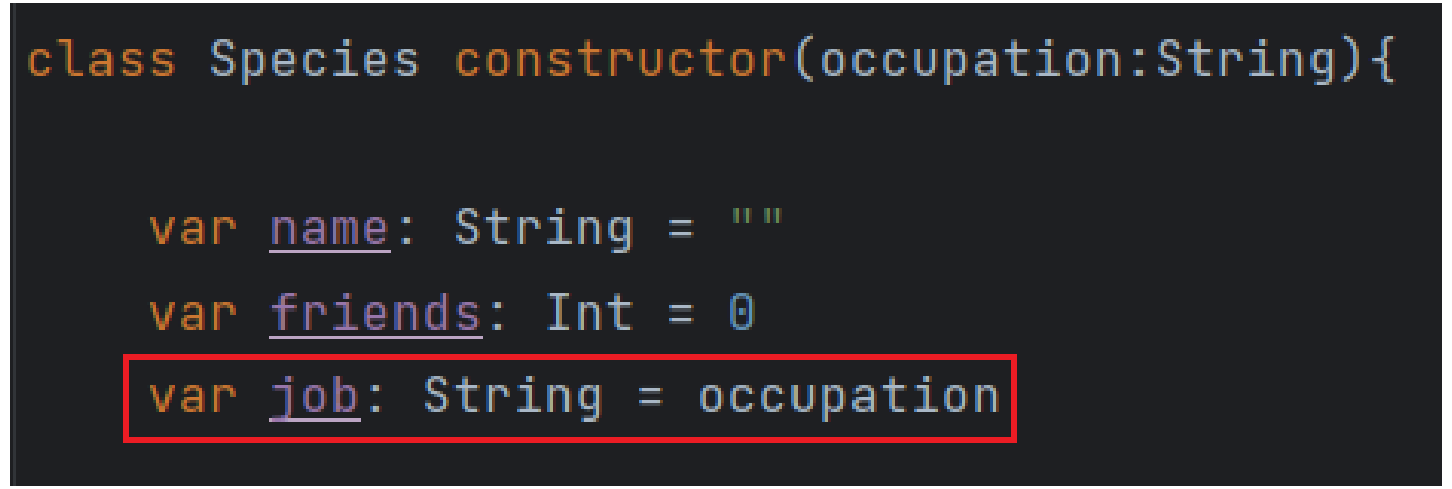


```
class Species constructor(occupation:String){

    var name: String = ""
    var friends: Int = 0

    fun displayBio(){
        println("$name is $occupation with $friends friend(s)")
    }
}
```

- 注意，我们不能在方法中使用 构造函数参数

Constructors (cont.)

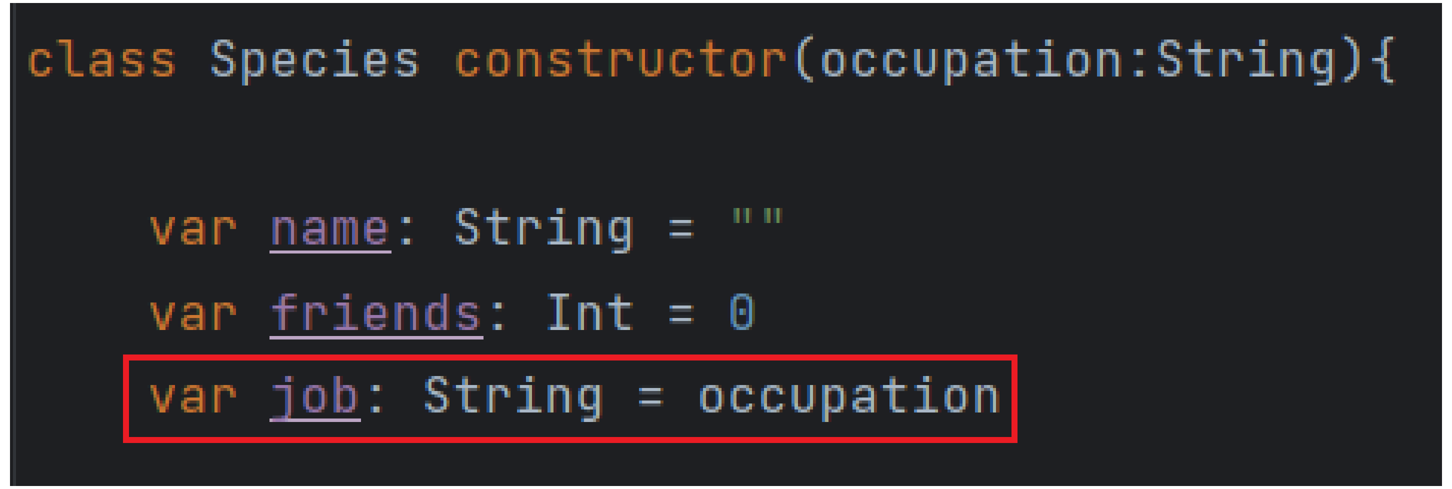
- Constructor parameters can either be used outside functions



```
class Species constructor(occupation:String){

    var name: String = ""
    var friends: Int = 0
    var job: String = occupation
```

构造函数（续）

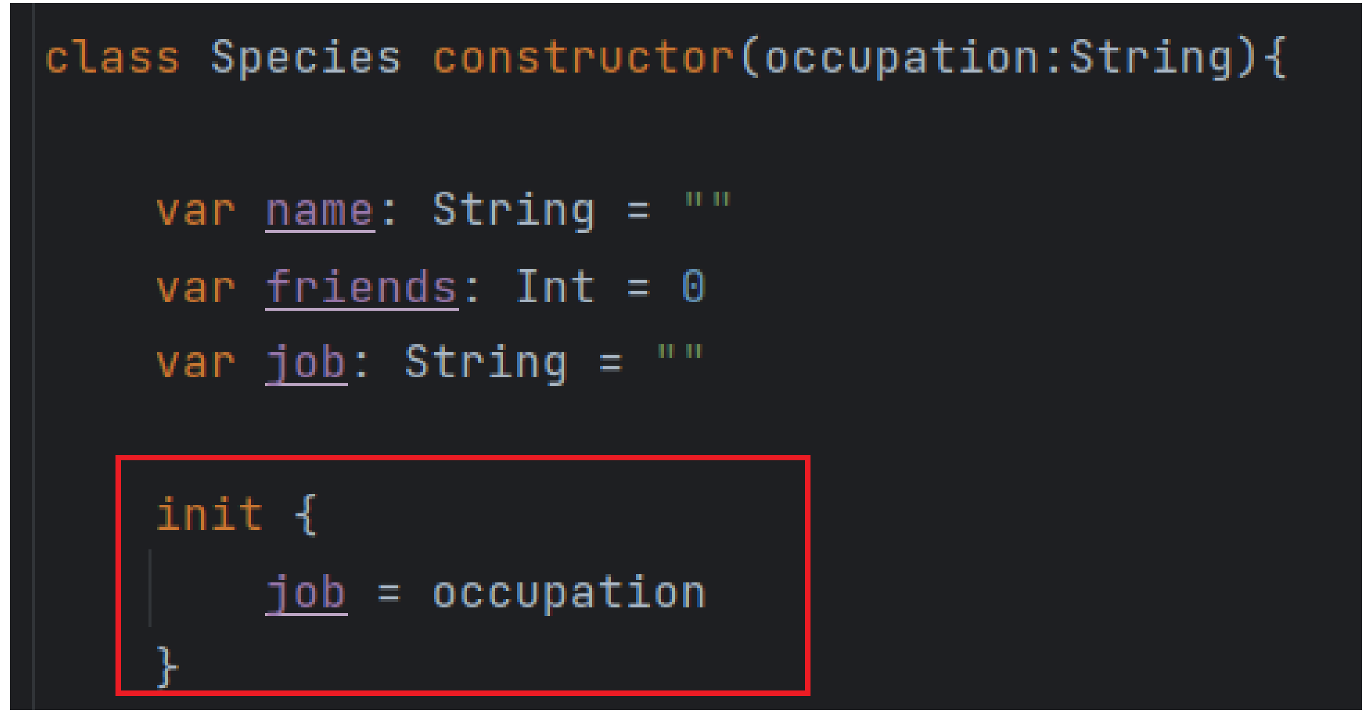
- 构造函数参数可以用于函数外部



```
class Species constructor(occupation:String){

    var name: String = ""
    var friends: Int = 0
    var job: String = occupation
```

Constructors (cont.)

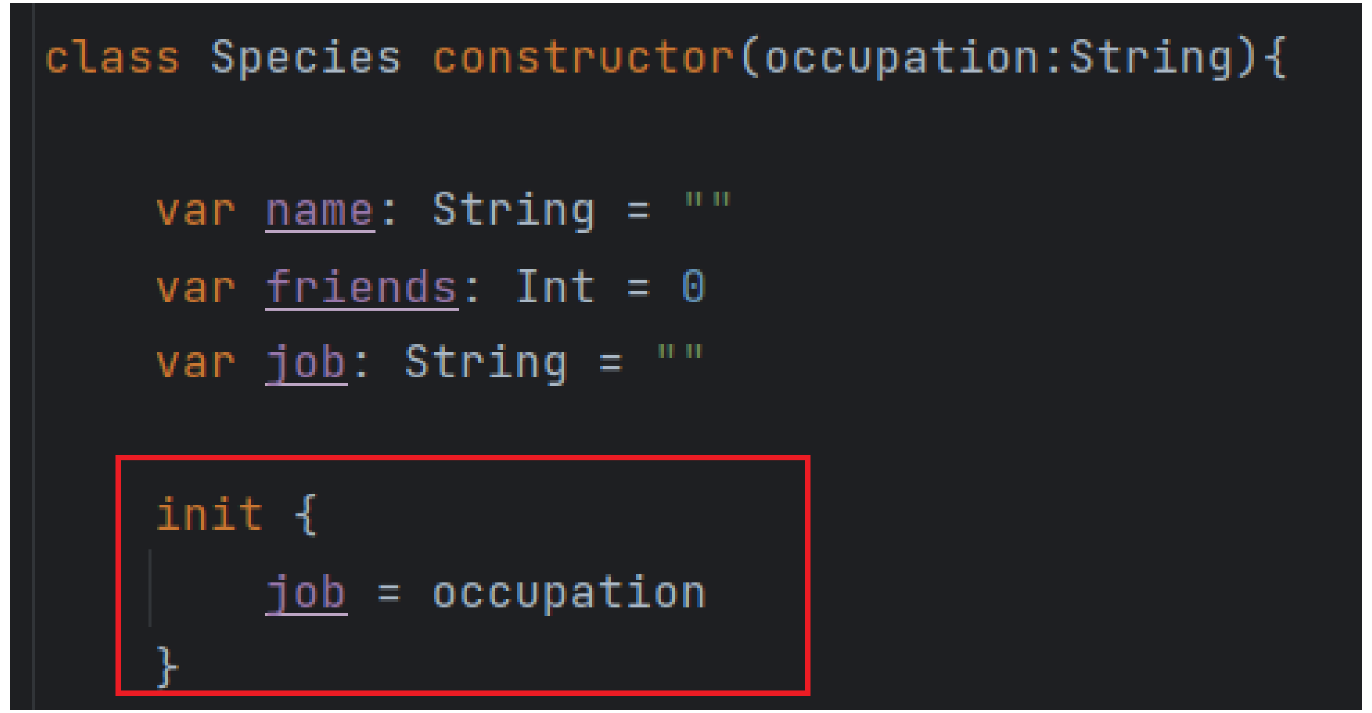
- Or used inside the *init* function
  - The init block is run whenever the class is instantiated



```
class Species constructor(occupation:String){

    var name: String = ""
    var friends: Int = 0
    var job: String = ""

    init {
        job = occupation
    }
```

构造函数（续）

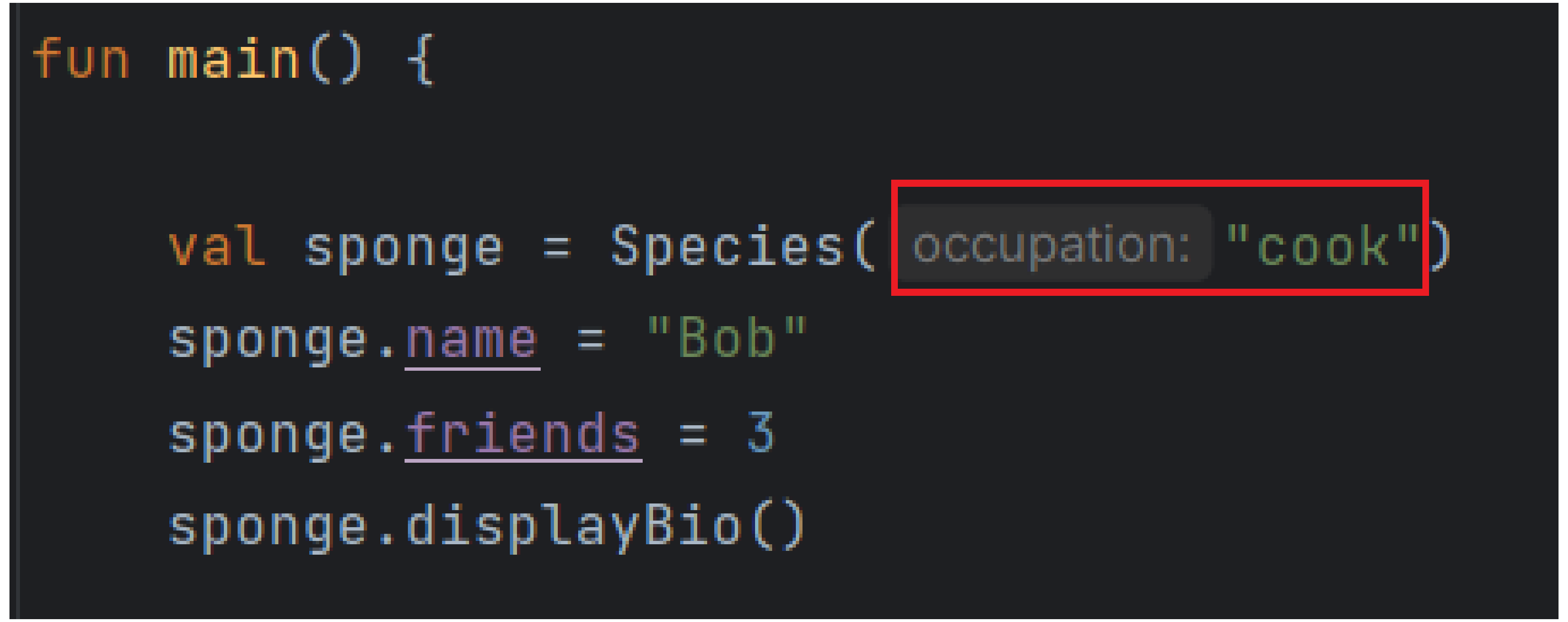
- 或在 *init* 函数内部使用
  - 每当实例化该类时，都会执行 init 代码块



```
class Species constructor(occupation:String){

    var name: String = ""
    var friends: Int = 0
    var job: String = ""

    init {
        job = occupation
    }
```

Constructors (cont.)

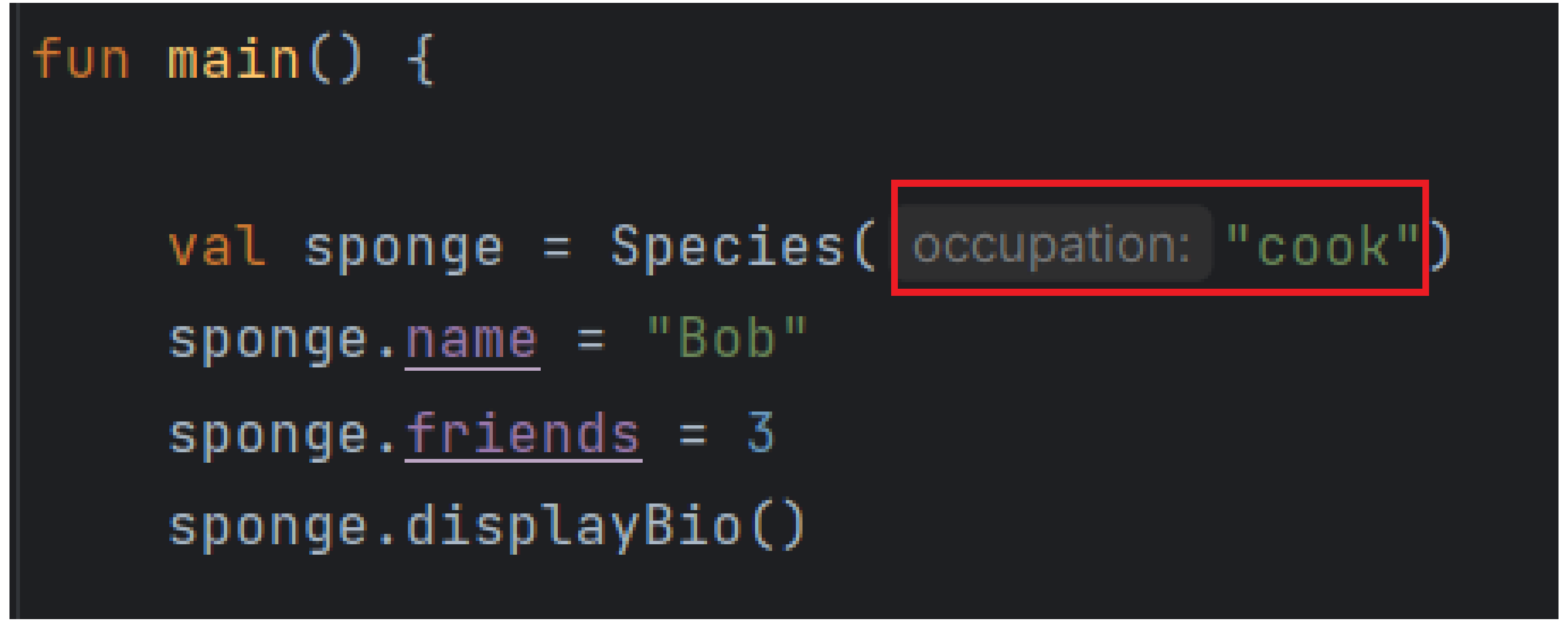
- We then can pass in arguments when we initialize a new object



```
fun main() {

    val sponge = Species( occupation: "cook")
    sponge.name = "Bob"
    sponge.friends = 3
    sponge.displayBio()
```

构造函数（续）

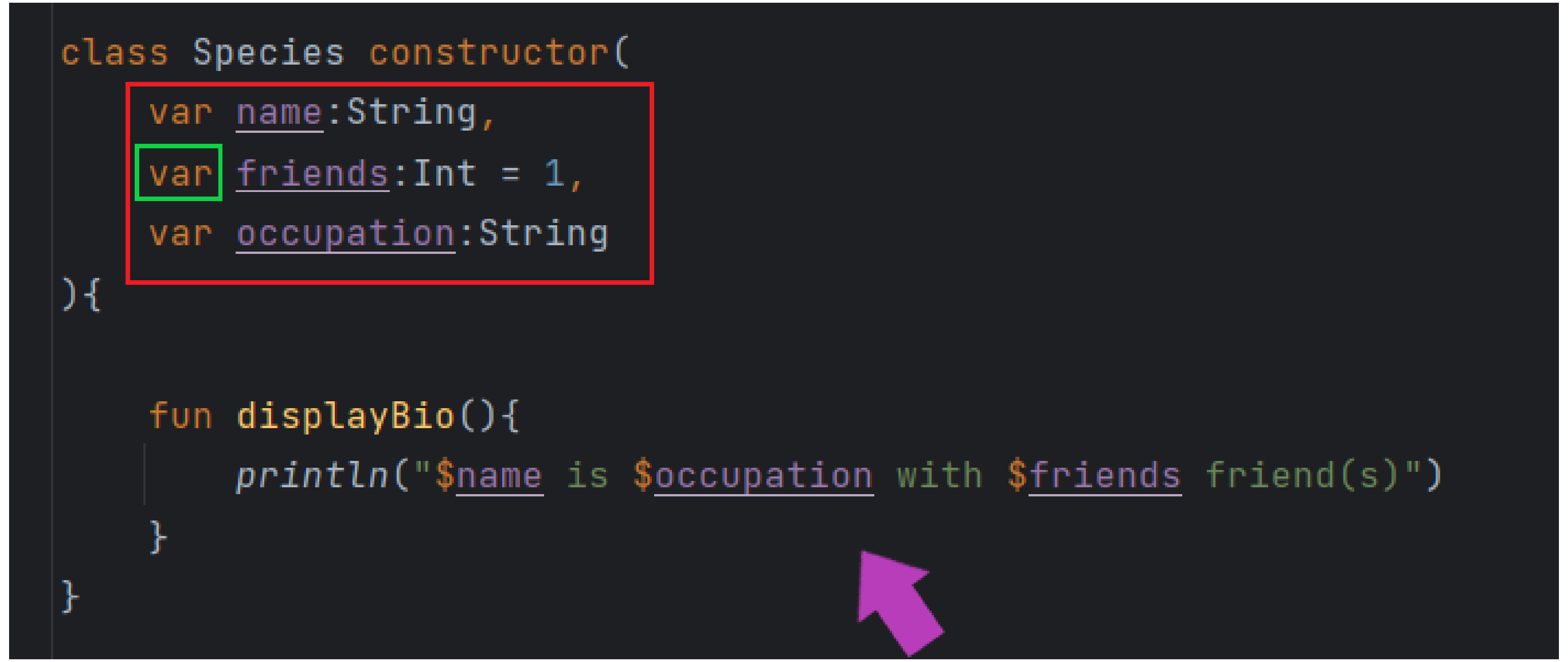
- 我们在初始化一个新对象时可以传入参数



```
fun main() {

    val sponge = Species( occupation: "cook")
    sponge.name = "Bob"
    sponge.friends = 3
    sponge.displayBio()
```

Constructors (cont.)

- We can also create properties as parameters in a constructor
  - The difference to regular parameters is declaring them with val or var

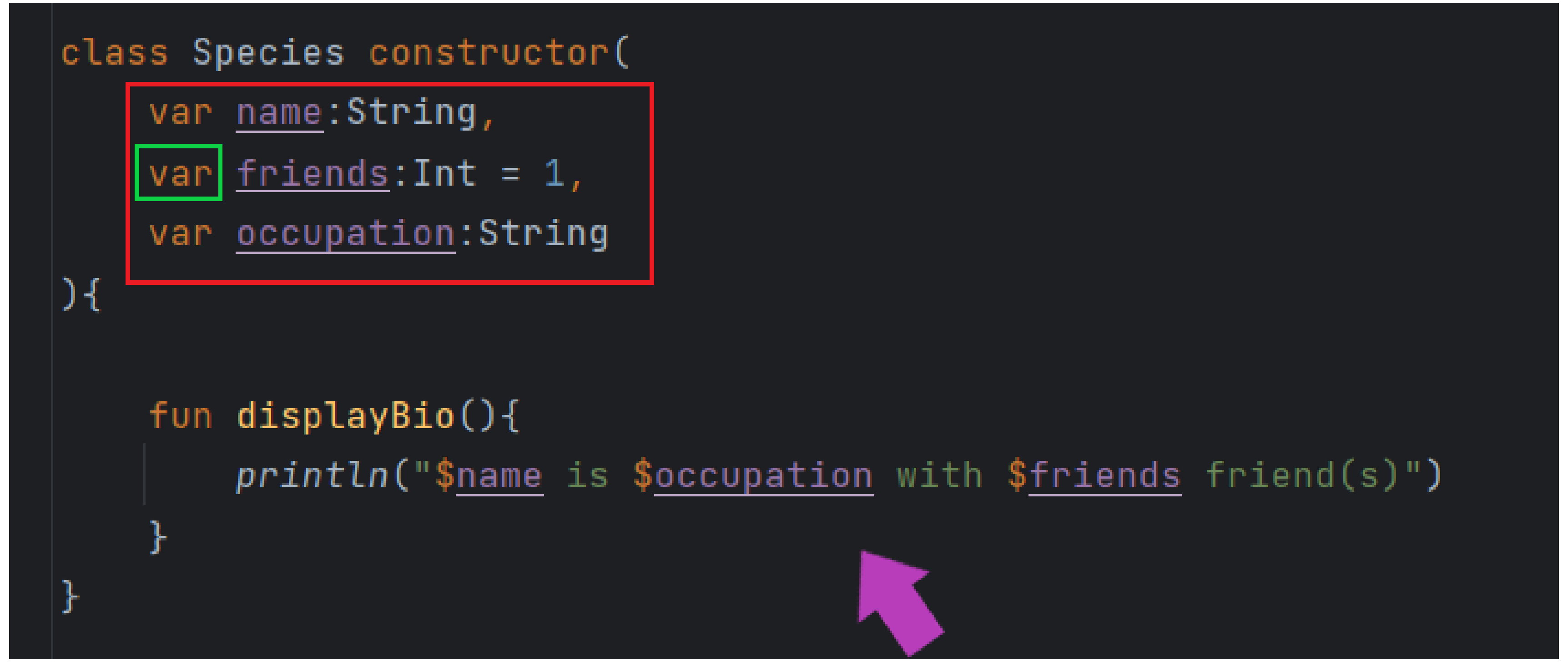


```
class Species constructor(
    var name:String,
    var friends:Int = 1,
    var occupation:String
){

    fun displayBio(){
        println("$name is $occupation with $friends friend(s)")
    }
}
```

- Since they are properties, they can be used everywhere in the class

构造函数（续）

- 我们还可以在构造函数中将属性作为参数进行创建
  - 与普通参数的区别在于使用 val 或 var 来声明它们

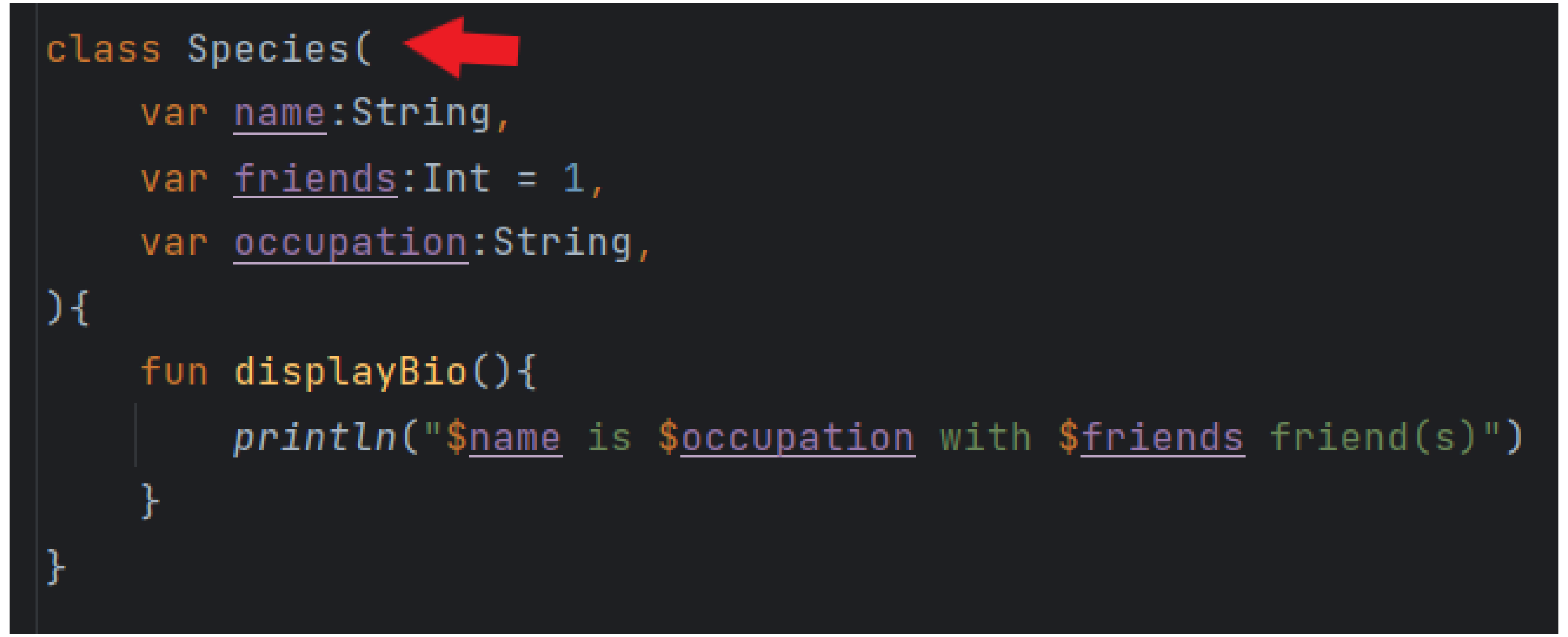


```
class Species constructor(
    var name:String,
    var friends:Int = 1,
    var occupation:String
){

    fun displayBio(){
        println("$name is $occupation with $friends friend(s)")
    }
}
```

- 由于它们是属性，因此可以在类的任何地方 使用

Constructors (cont.)

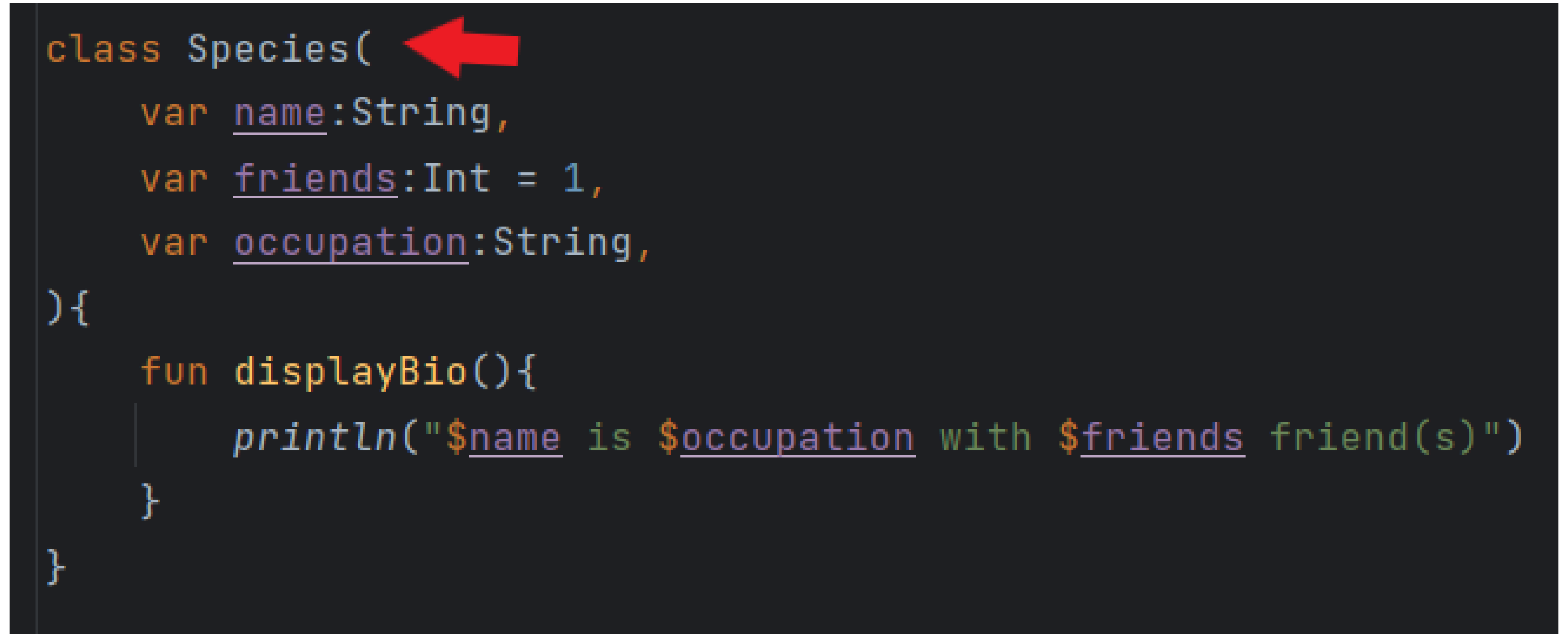
- The constructor keyword can also be dropped



```
class Species(
    var name:String,
    var friends:Int = 1,
    var occupation:String,
){
    fun displayBio(){
        println("$name is $occupation with $friends friend(s)")
    }
}
```

构造函数（续）

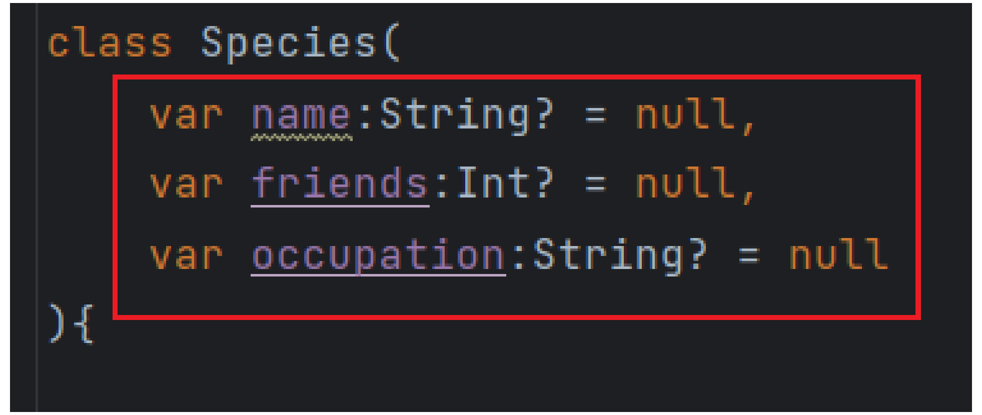
- 也可以省略constructor关键字



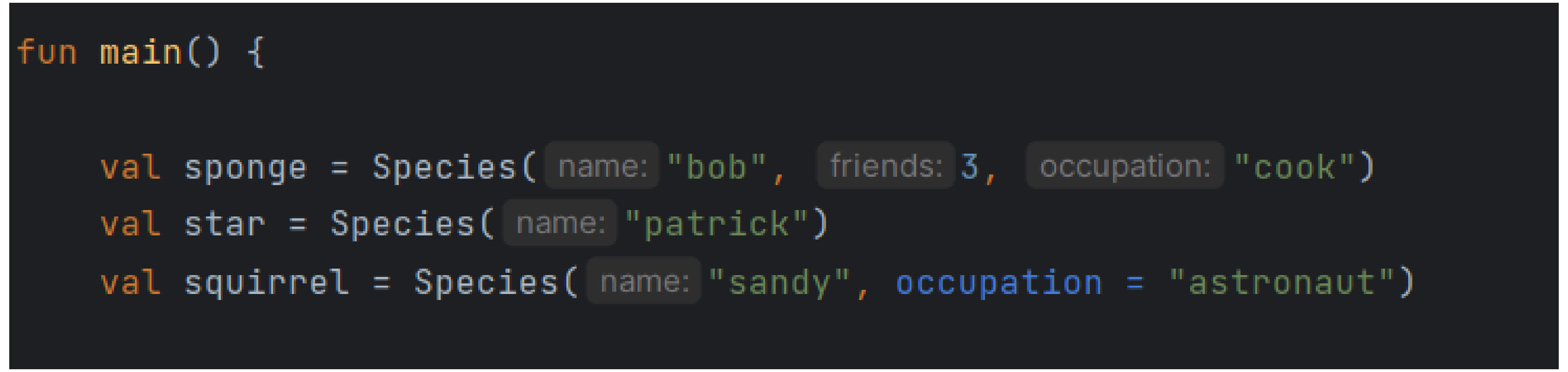
```
class Species(
    var name:String,
    var friends:Int = 1,
    var occupation:String,
){
    fun displayBio(){
        println("$name is $occupation with $friends friend(s)")
    }
}
```

Object instantiation

- By using default values for parameters, we can provide multiple ways to create a new object



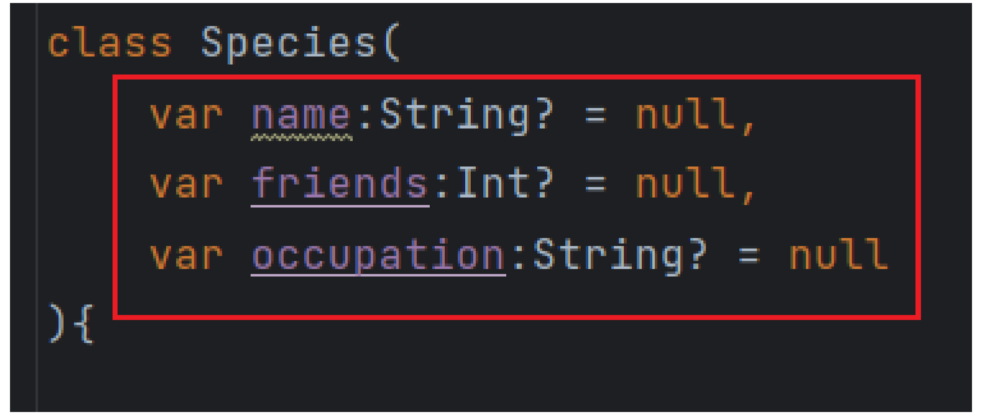
```
class Species(
    var name:String? = null,
    var friends:Int? = null,
    var occupation:String? = null
){
```



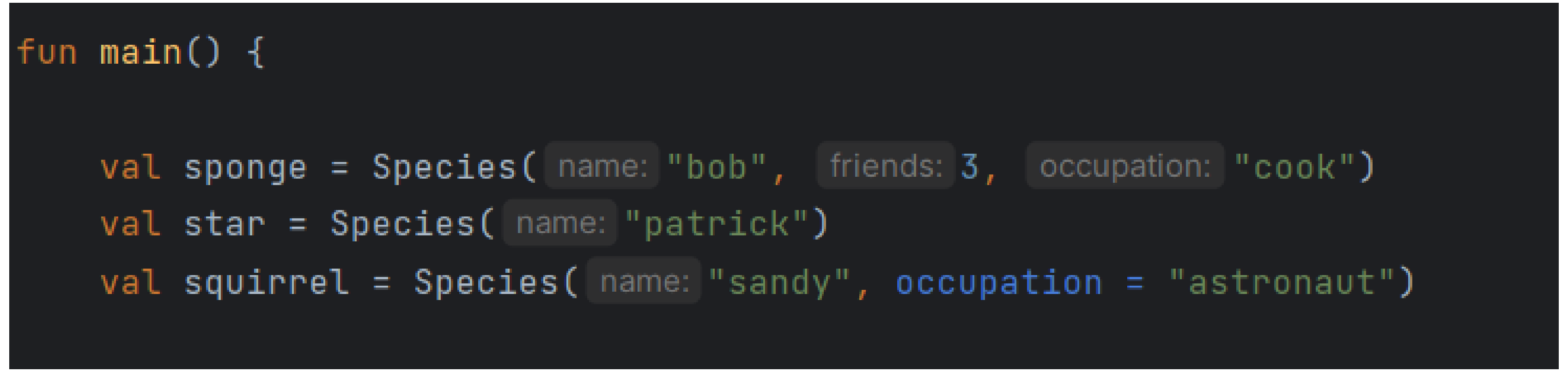
```
fun main() {

    val sponge = Species(name: "bob", friends: 3, occupation: "cook")
    val star = Species(name: "patrick")
    val squirrel = Species(name: "sandy", occupation = "astronaut")
```

对象实例化

- 通过使用 参数的默认值，我们可以提供多种方式来创建一个新对象



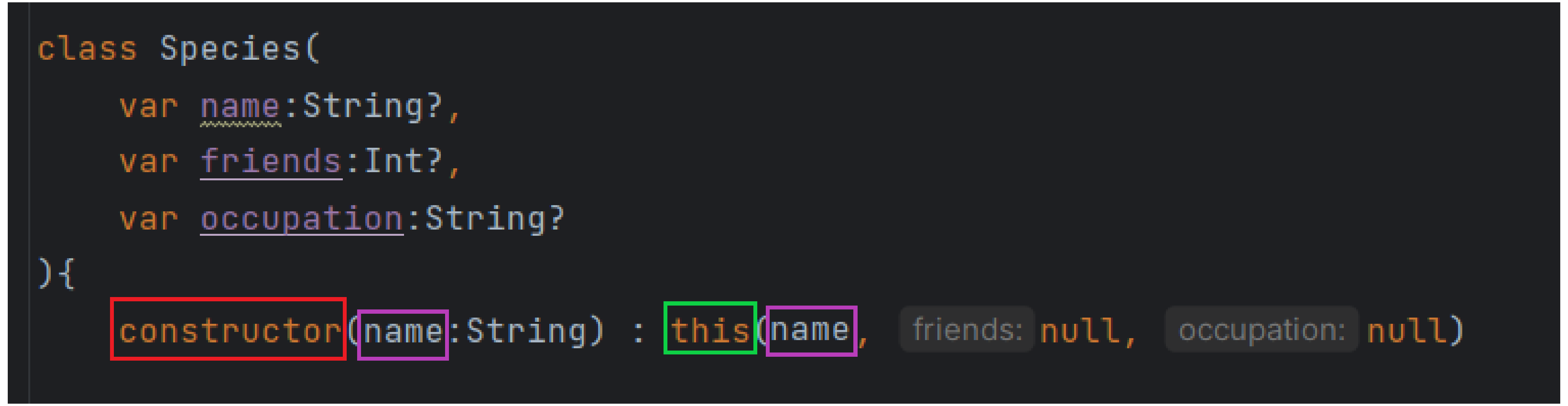
```
class Species(
    var name:String? = null,
    var friends:Int? = null,
    var occupation:String? = null
){
```



```
fun main() {

    val sponge = Species(name: "bob", friends: 3, occupation: "cook")
    val star = Species(name: "patrick")
    val squirrel = Species(name: "sandy", occupation = "astronaut")
```

Multiple constructors (cont.)

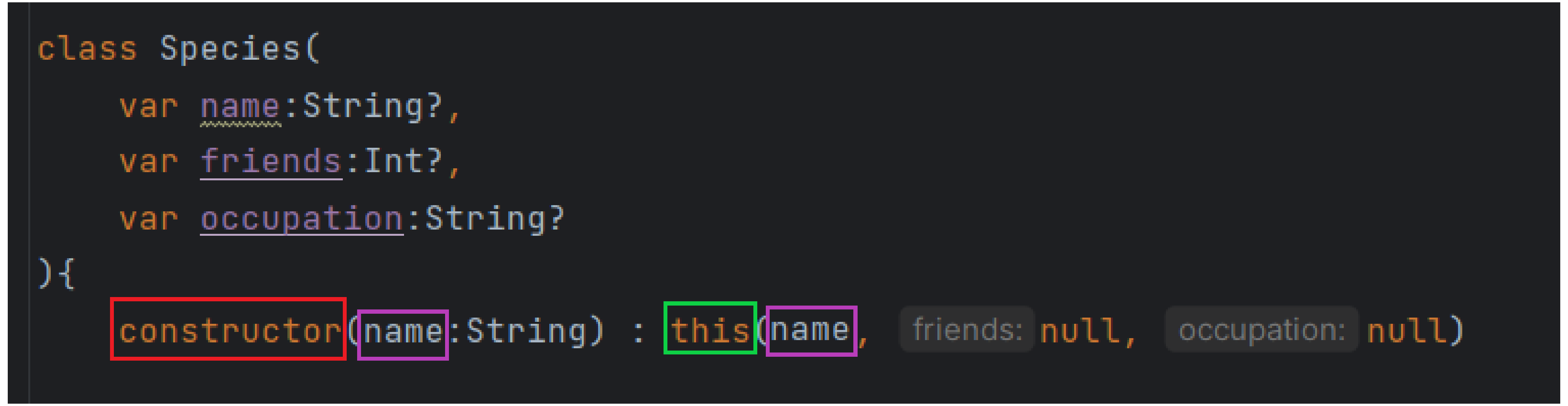
- Another way to accomplish the previous logic is to create a secondary constructor
- The parameter we want to isolate gets passed into the primary constructor using the syntax below



```
class Species(
    var name:String?,
    var friends:Int?,
    var occupation:String?
){
    constructor(name:String) : this(name, friends: null, occupation: null)
```

多个构造函数（续）

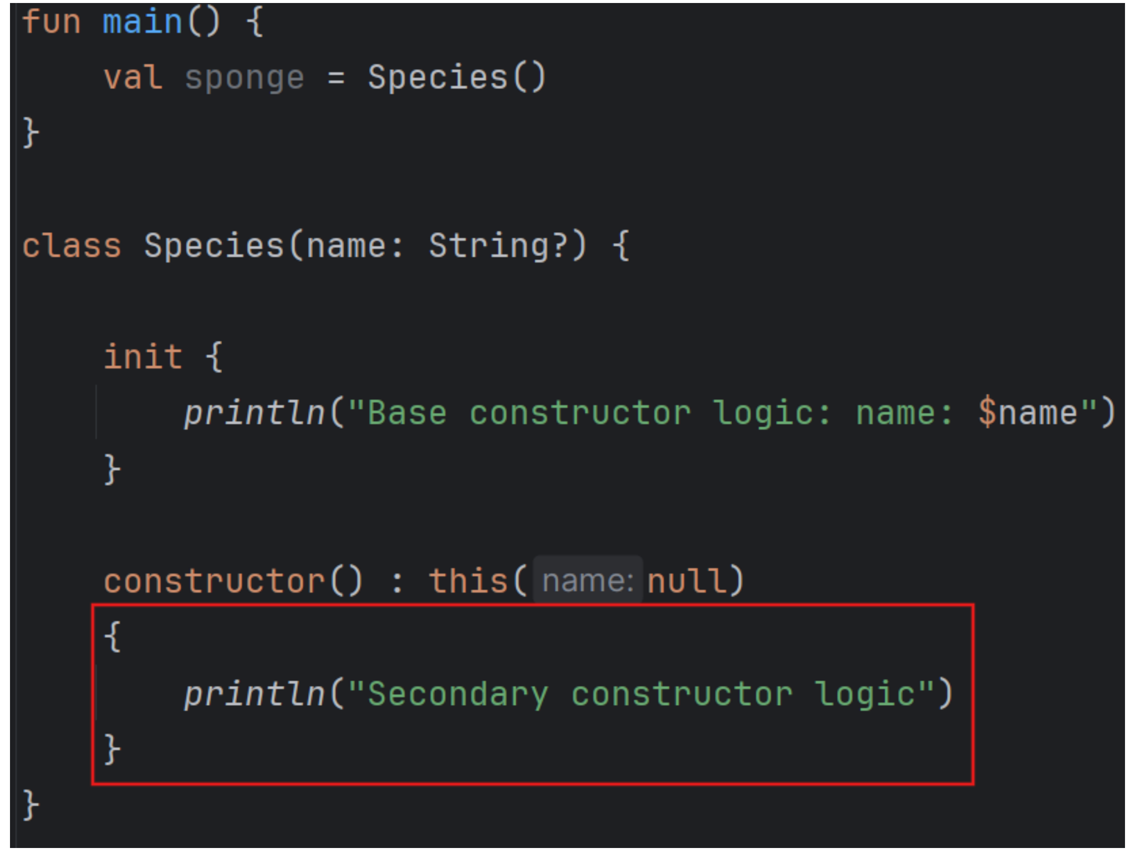
- 实现前述逻辑的另一种方法是创建一个次要构造函数
- 我们想要隔离的参数通过以下语法传递给主构造函数。



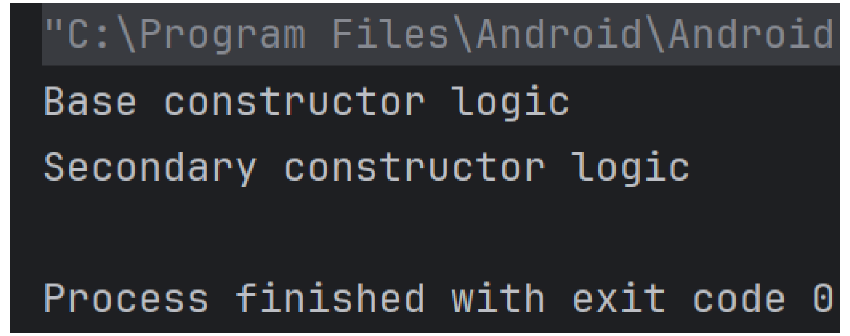
```
class Species(
    var name:String?,
    var friends:Int?,
    var occupation:String?
){
    constructor(name:String) : this(name, friends: null, occupation: null)
```

Multiple constructors (cont.)

- You can have separate initialization logic for each constructor



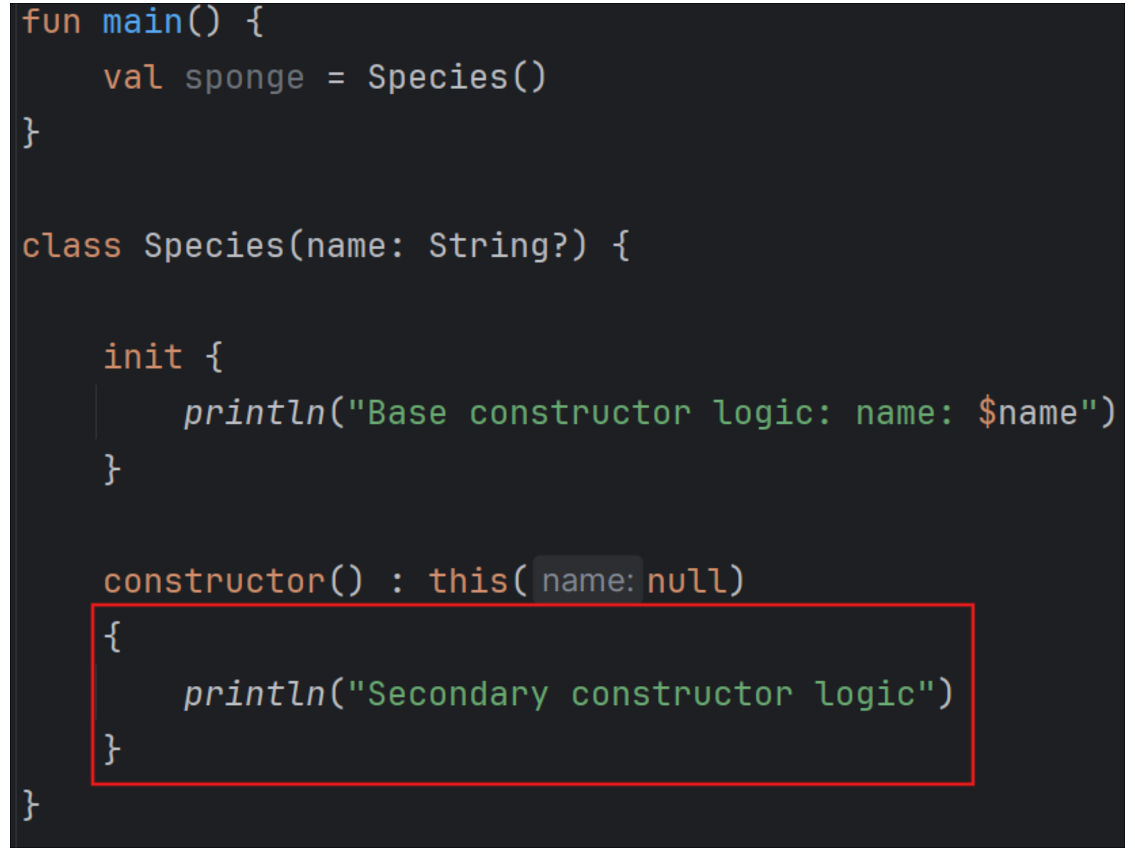
```
fun main() {
    val sponge = Species()
}

class Species(name: String?) {

    init {
        println("Base constructor logic: name: $name")
    }

    constructor() : this( name: null)
    {
        println("Secondary constructor logic")
    }
}
```



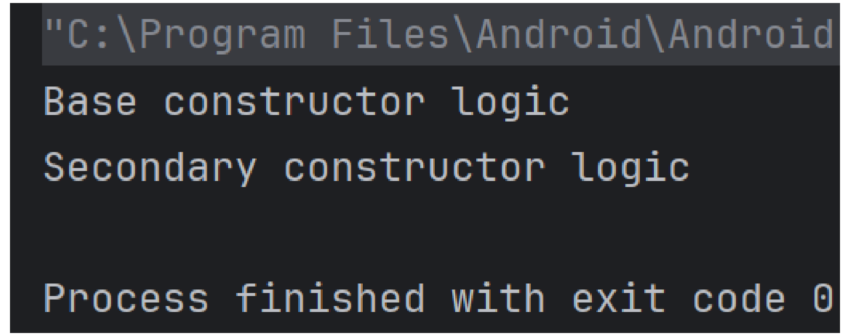
```
"C:\Program Files\Android\Android
Base constructor logic
Secondary constructor logic

Process finished with exit code 0
```

多个构造函数（续）

- 你可以为 每个构造函数设置独立的初始化逻辑



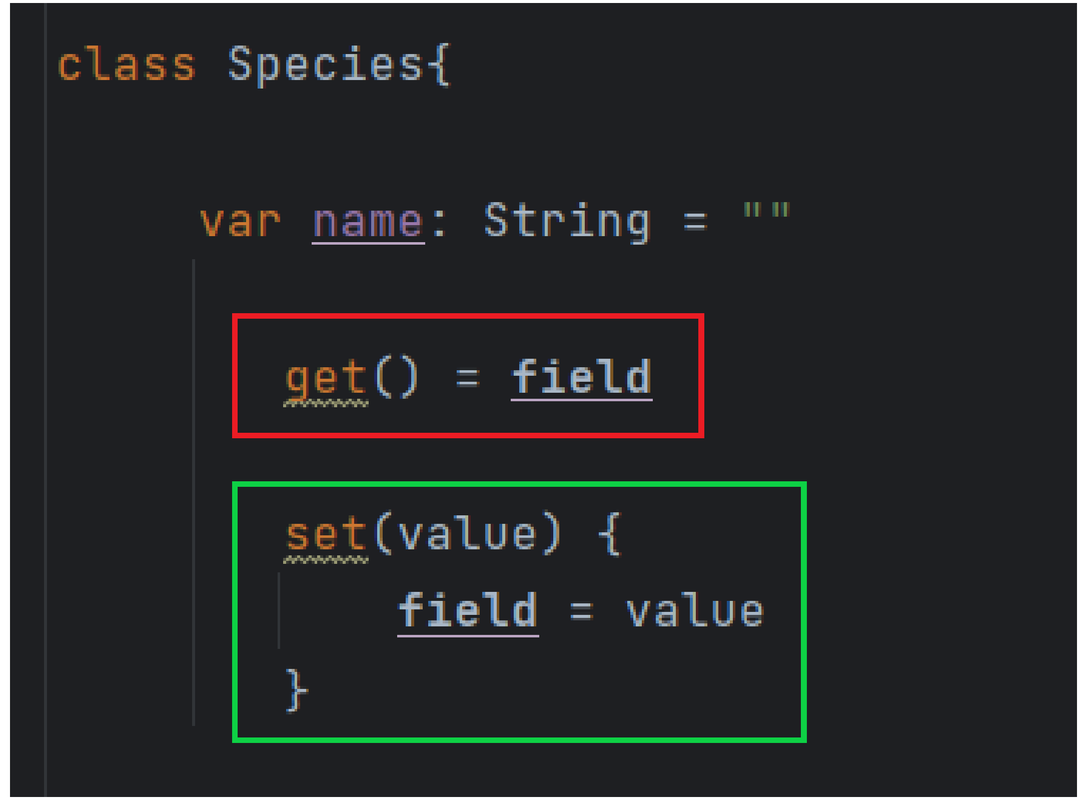
```
fun main() {
    val sponge = Species()
}

class Species(name: String?) {

    init {
        println("Base constructor logic: name: $name")
    }

    constructor() : this( name: null)
    {
        println("Secondary constructor logic")
    }
}
```



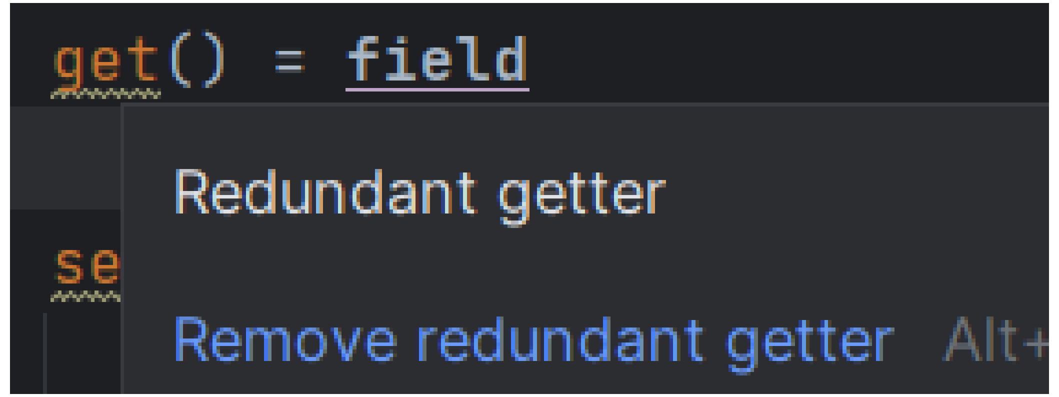
```
"C:\Program Files\Android\Android
Base constructor logic
Secondary constructor logic

Process finished with exit code 0
```

Getters & setters

- Getters and setters are auto generated and hidden



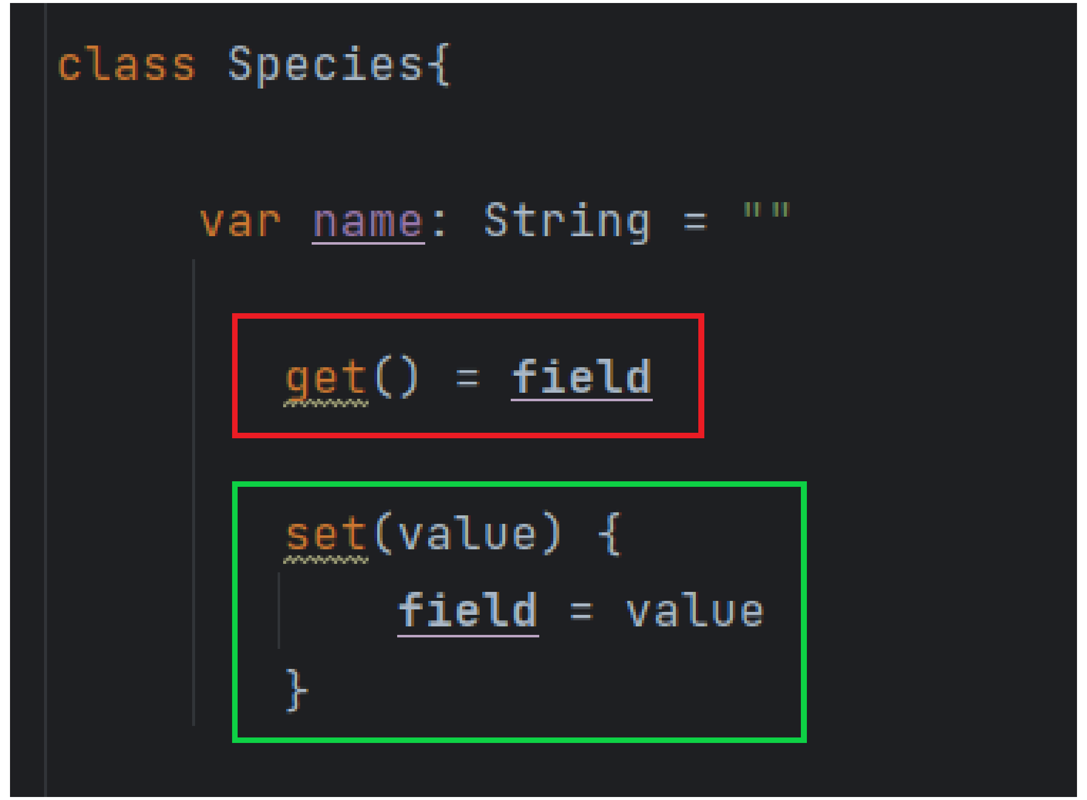
```
class Species{

    var name: String = ""
        get() = field
        set(value) {
            field = value
        }
```



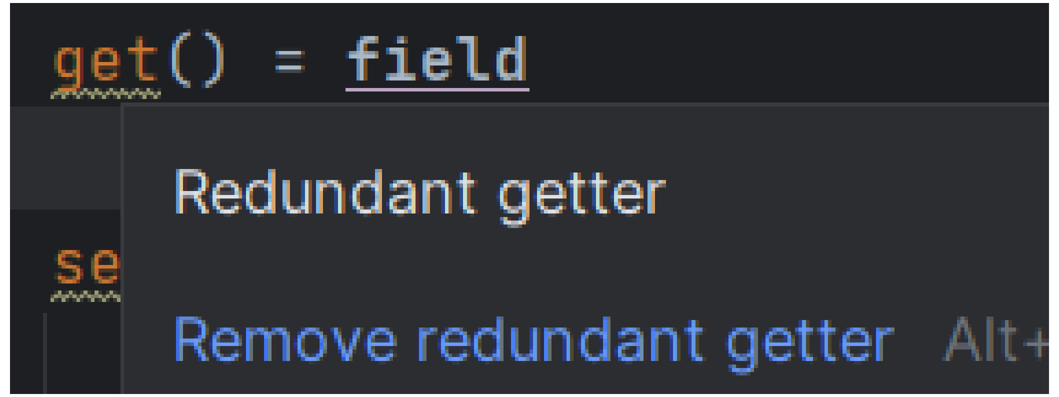
```
get() = field
se
   Redundant getter

   Remove redundant getter   Alt+
```

Getter 和 Setter

- Getter 和 Setter 是自动生成且隐藏的



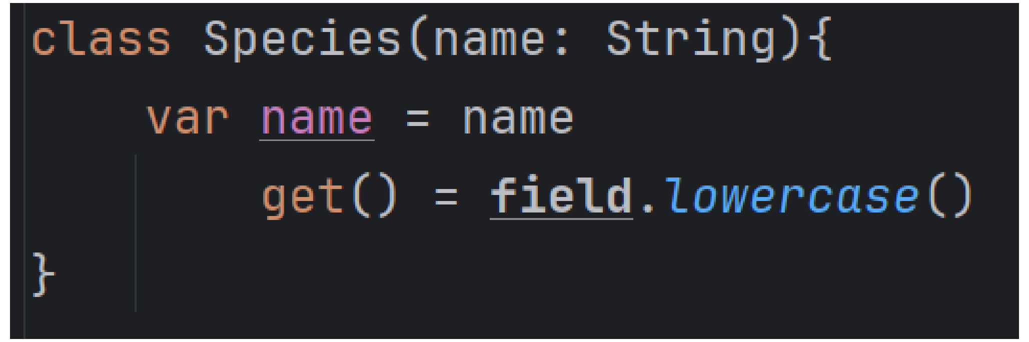
```
class Species{

    var name: String = ""
        get() = field
        set(value) {
            field = value
        }
```



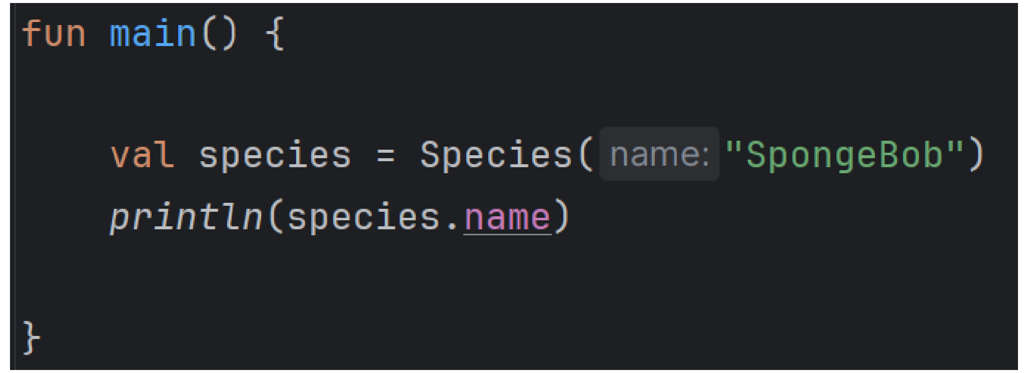
```
get() = field
se
   Redundant getter

   Remove redundant getter   Alt+
```

Getters & setters (cont.)

- Getters and setters help promote strong encapsulation within our class
  - Ex. We can provide controlled rules



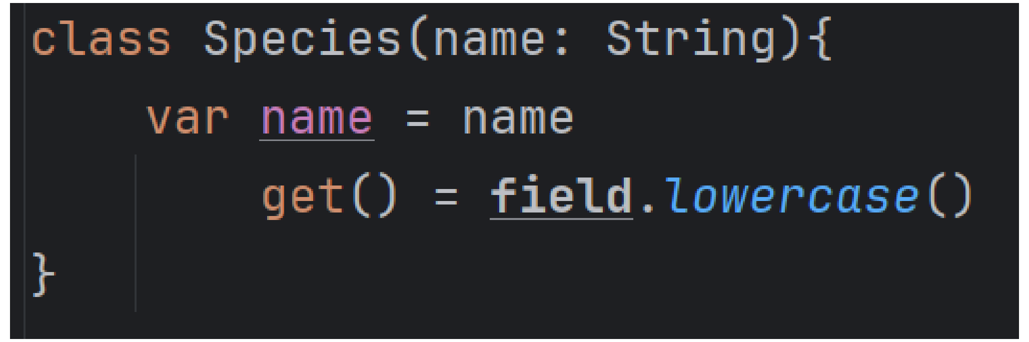
```
class Species(name: String){
    var name = name
        get() = field.lowercase()
}
```



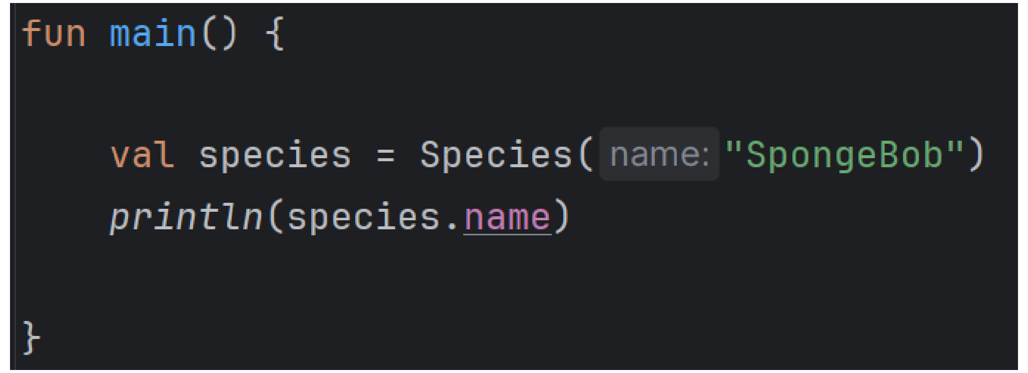
```
fun main() {

    val species = Species( name: "SpongeBob")
    println(species.name)

}
```

Getter 和 Setter（续）

- Getter 和 Setter 有助于在我们的类中实现良好的封装性
  - 例如，我们可以提供受控的规则



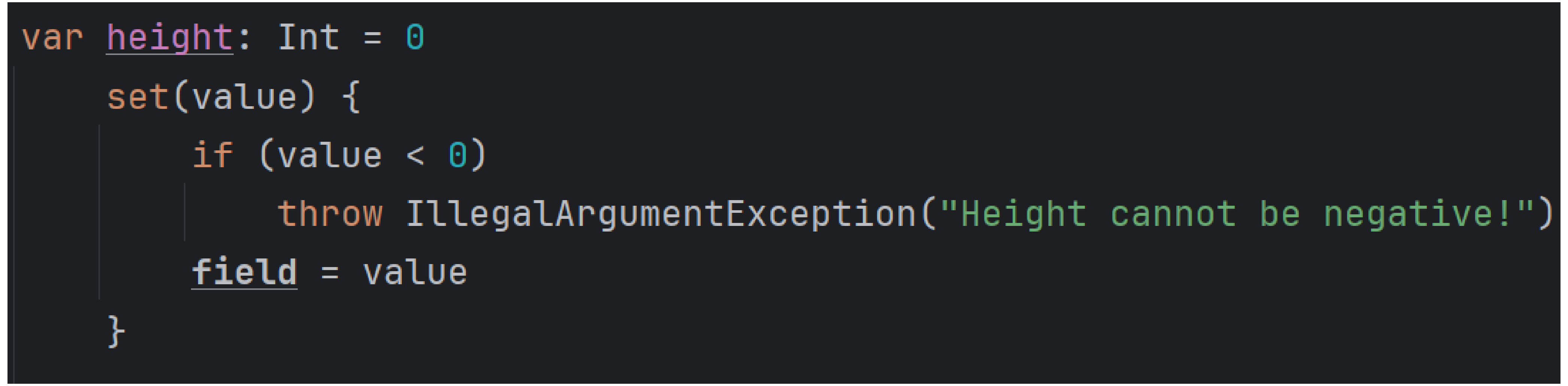
```
class Species(name: String){
    var name = name
        get() = field.lowercase()
}
```



```
fun main() {

    val species = Species( name: "SpongeBob")
    println(species.name)

}
```

Getters & setters (cont.)

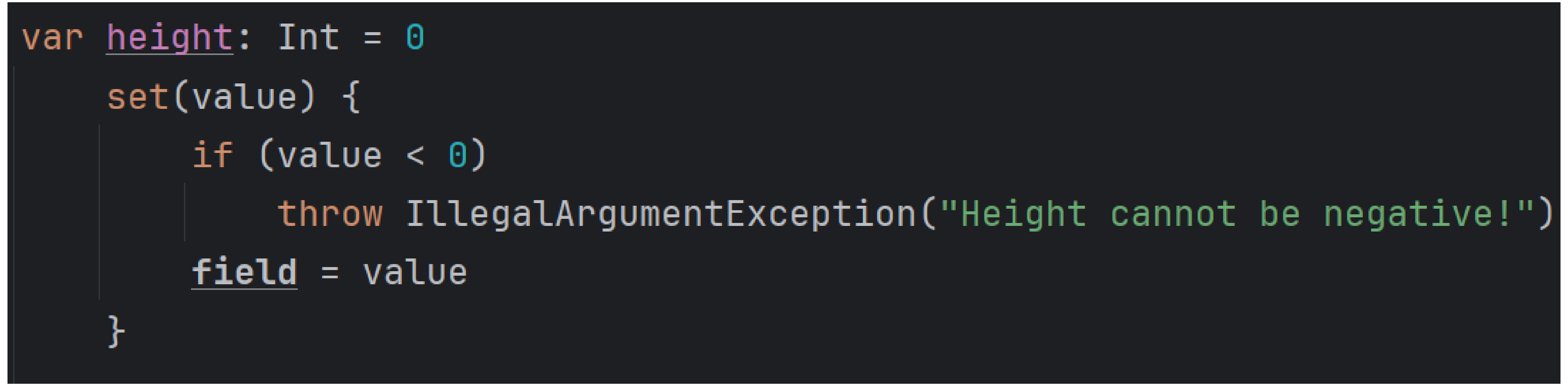
- Here we are encapsulating the validation of our data



```kotlin
var height: Int = 0
    set(value) {
        if (value < 0)
            throw IllegalArgumentException("Height cannot be negative!")
        field = value
    }
```

取值器与赋值器（续）

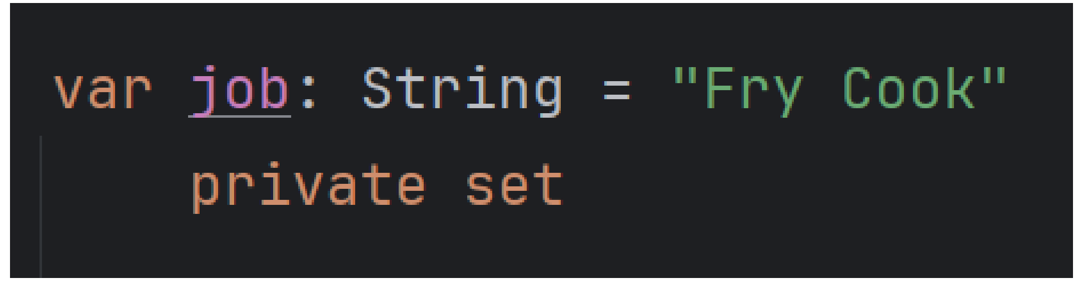
- 在这里，我们正在封装数据的验证



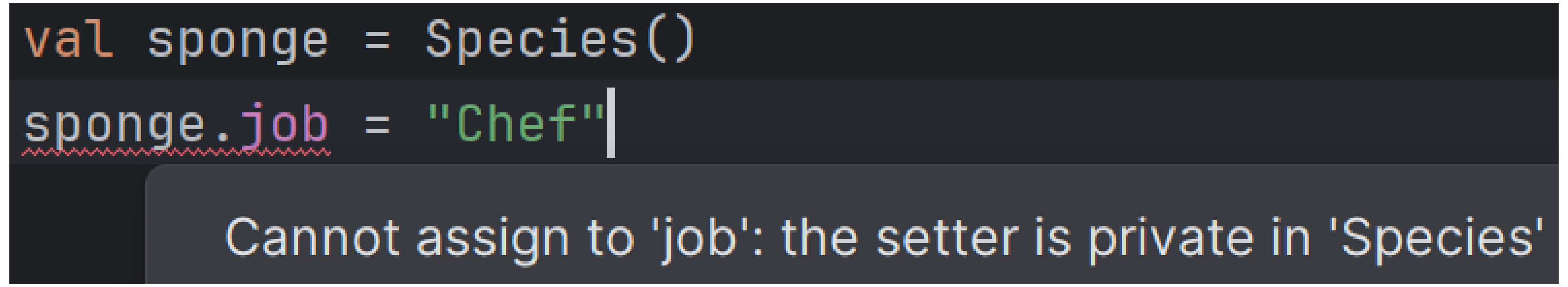
```kotlin
var height: Int = 0
    set(value) {
        if (value < 0)
            throw IllegalArgumentException("Height cannot be negative!")
        field = value
    }
```

Getters & setters (cont.)

- We can also restrict access to data
  - Ex. With a private setter, we can only modify the value internally



```
var job: String = "Fry Cook"
    private set
```

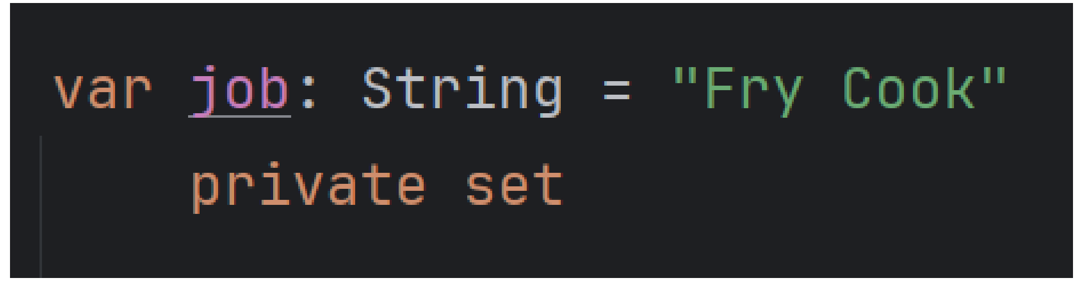


```
val sponge = Species()
sponge.job = "Chef"
```

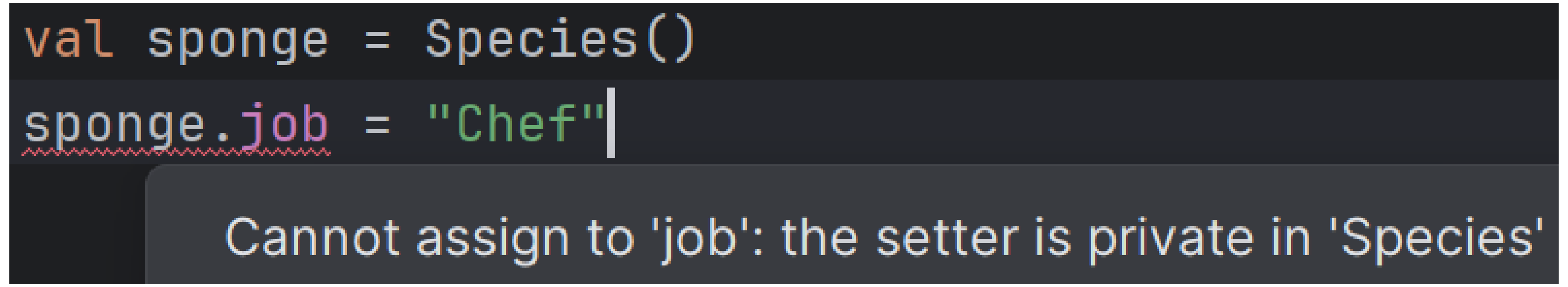
Cannot assign to 'job': the setter is private in 'Species'

获取器与设置器（续）

- 我们还可以限制对数据的访问
  - 例如，使用私有设置器时，我们只能在内部修改该值



```
var job: String = "Fry Cook"
    private set
```



```
val sponge = Species()
sponge.job = "Chef"
```

Cannot assign to 'job': the setter is private in 'Species'

Visibility modifiers

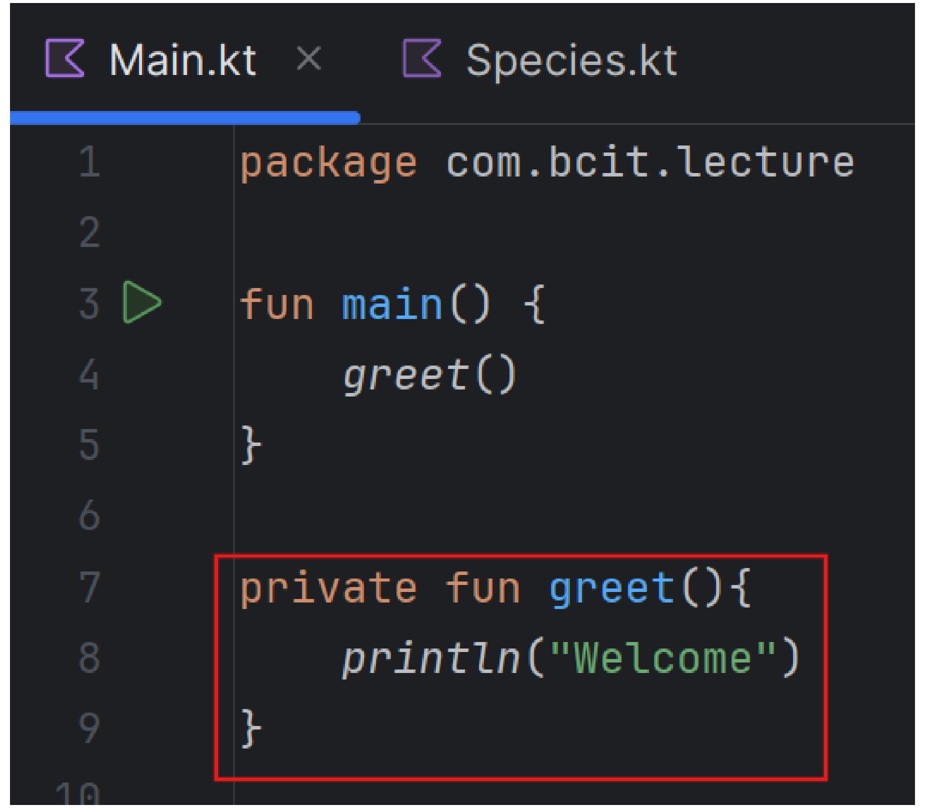
- There are 4 visibility modifiers
  - *public*: Default and can be accessed anywhere
  - *private*: Available only inside the same file or class
  - *protected*: Available only inside same class and subclasses
  - *internal*: available anywhere in the same module

可见性修饰符

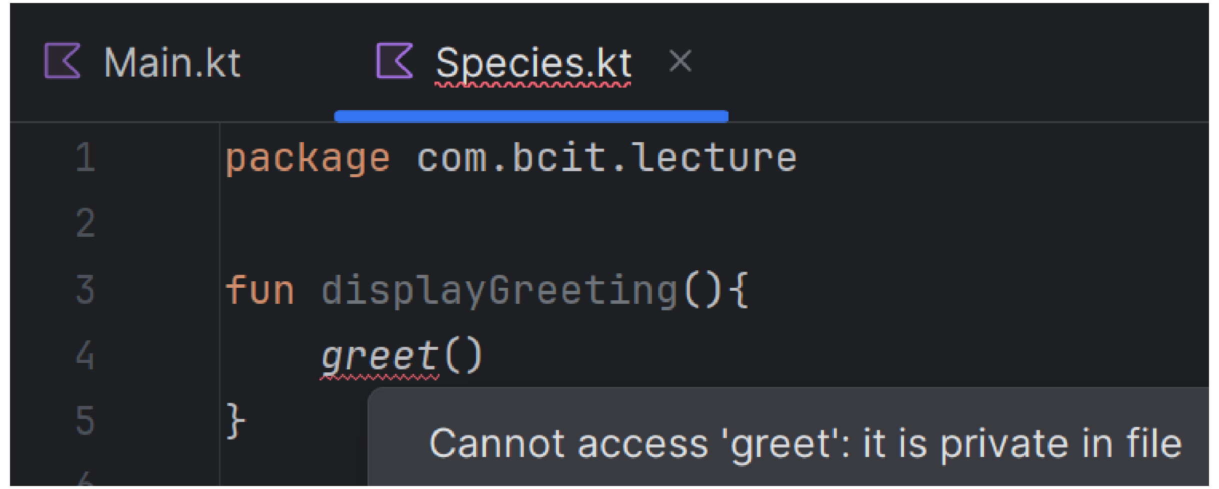
- 共有 4 种可见性修饰符
  - *public*：默认修饰符，可在任何地方访问
  - *private*：仅在同一个文件或类中可用
  - *protected*：仅在同一个类及子类中可用
  - *internal*：在同一个模块中的任何位置都可用

Visibility modifiers (cont.)

- Members that are private and declared outside classes are restricted to that file



```
package com.bcit.lecture

fun main() {
    greet()
}

private fun greet(){
    println("Welcome")
}
```

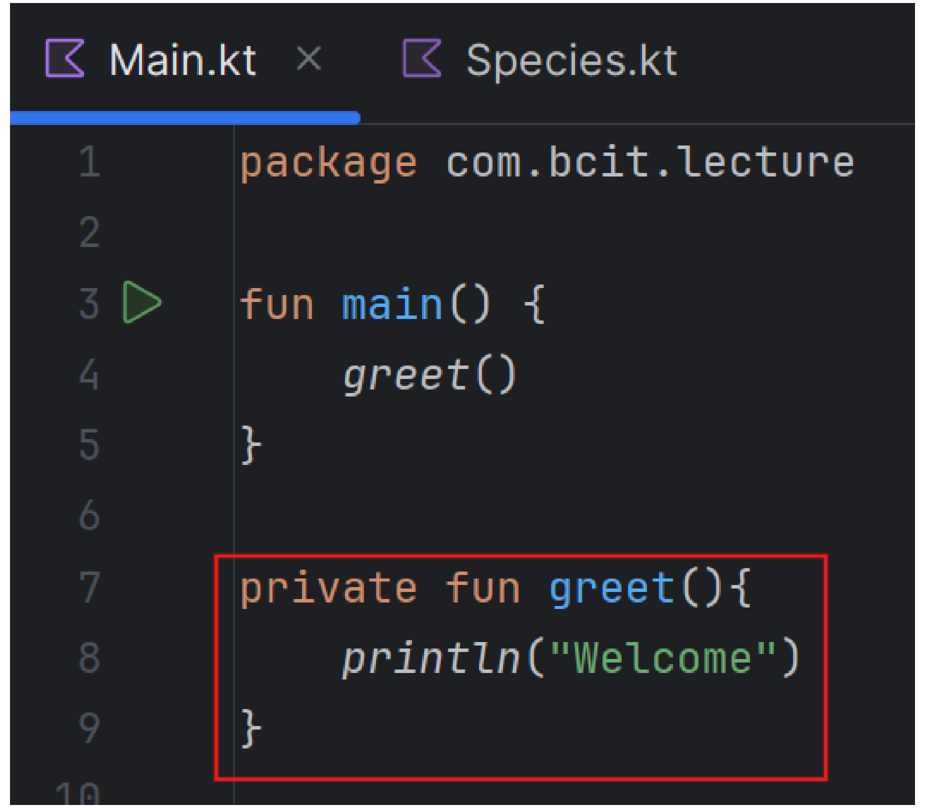


```
package com.bcit.lecture

fun displayGreeting(){
    greet()
}
```

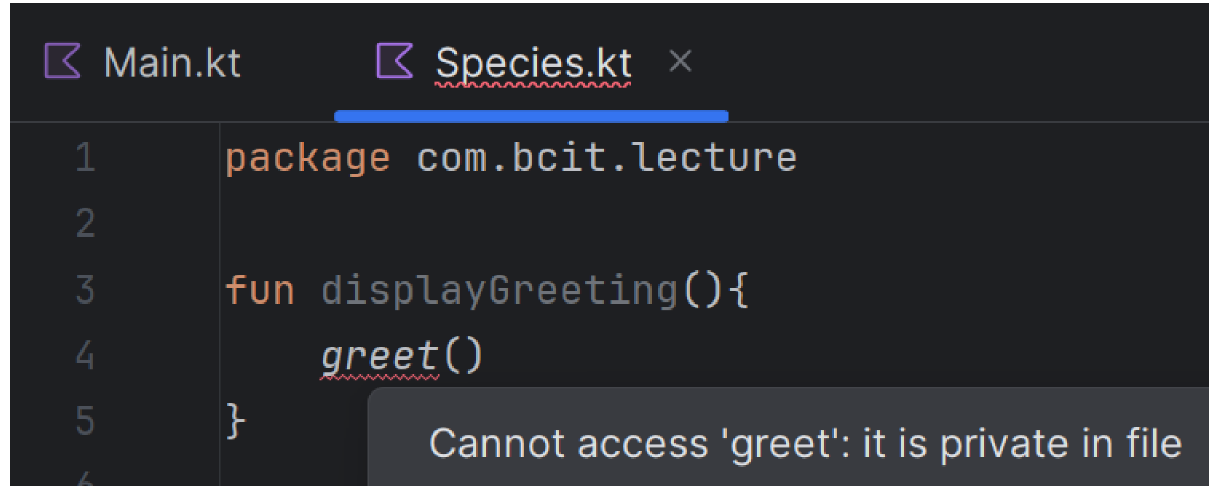
Cannot access 'greet': it is private in file

可见性修饰符（续）

- 在类外部声明的私有成员 outside classes 仅限于该文件内访问


Main.kt ×
Species.kt

```
package com.bcit.lecture

fun main() {
    greet()
}

private fun greet(){
    println("Welcome")
}
```

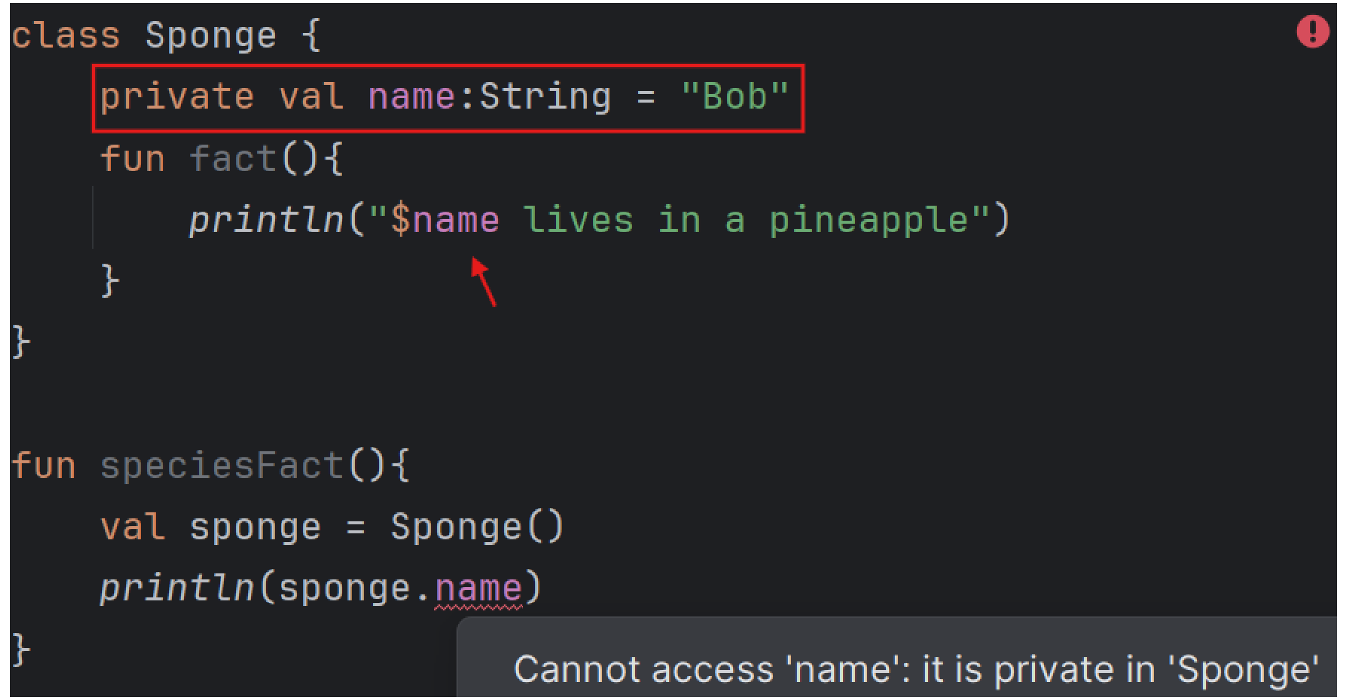

Main.kt
Species.kt ×

```
package com.bcit.lecture

fun displayGreeting(){
    greet()
}
```

Cannot access 'greet': it is private in file

Visibility modifiers (cont.)

- Members that are private and declared inside classes are restricted to that class

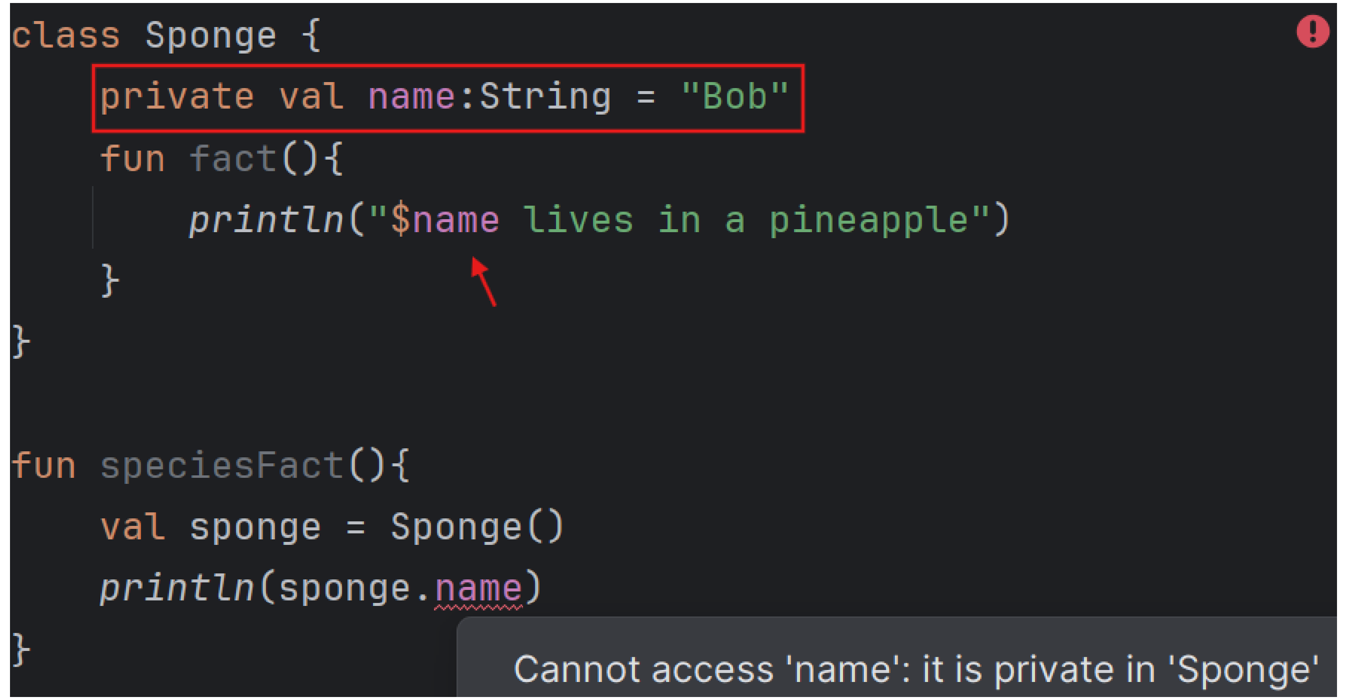


```
class Sponge {
    private val name:String = "Bob"
    fun fact(){
        println("$name lives in a pineapple")
    }
}

fun speciesFact(){
    val sponge = Sponge()
    println(sponge.name)
}
```

Cannot access 'name': it is private in 'Sponge'

可见性修饰符（续）

- 在类内部声明的私有成员 在该类中受到限制



```
class Sponge {
    private val name:String = "Bob"
    fun fact(){
        println("$name lives in a pineapple")
    }
}

fun speciesFact(){
    val sponge = Sponge()
    println(sponge.name)
}
```

Cannot access 'name': it is private in 'Sponge'

Inheritance

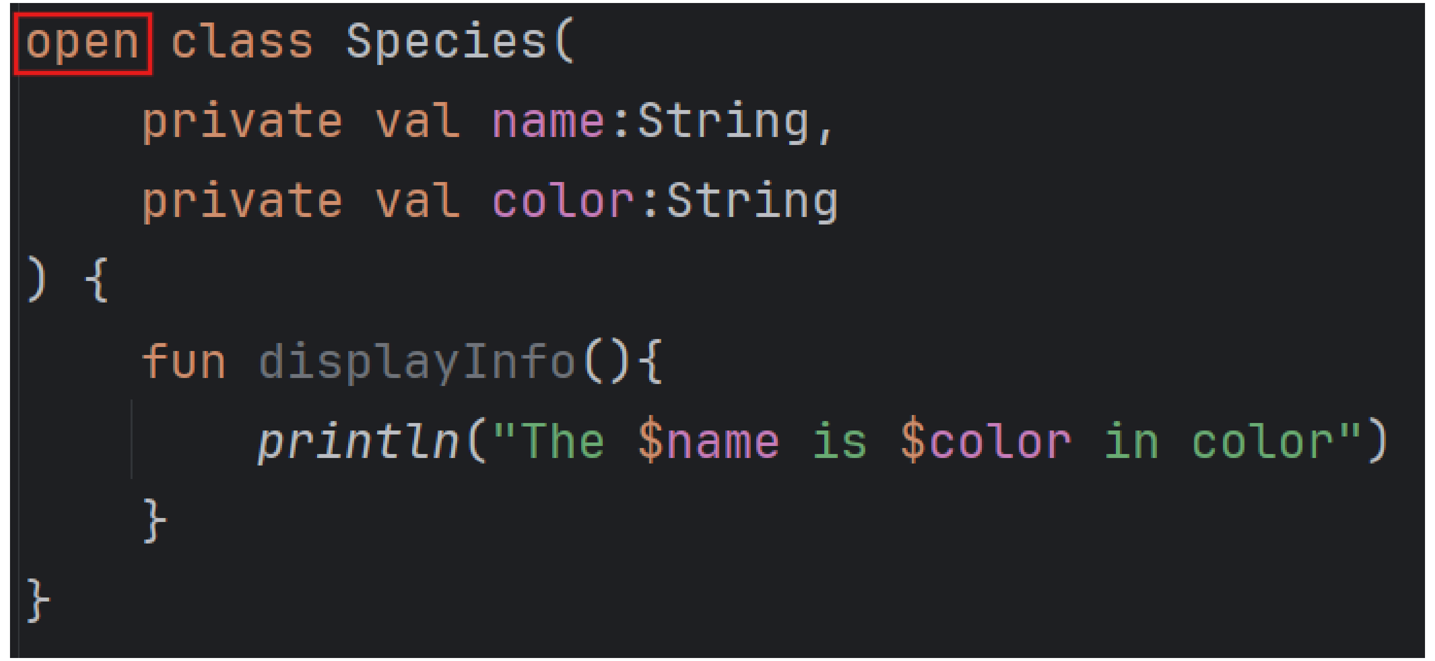
- Inheritance allows us to "inherit“ all members from another class
- The class being inherited from is called the *parent, super or base* class
- The class that is inheriting the parent is called the *child* or *subclass*

继承

- 继承使我们能够从另一个类“继承”所有成员
- 被继承的类称为父类、超类或基类
- 继承父类的类称为子类或派生类

Inheritance (cont.)

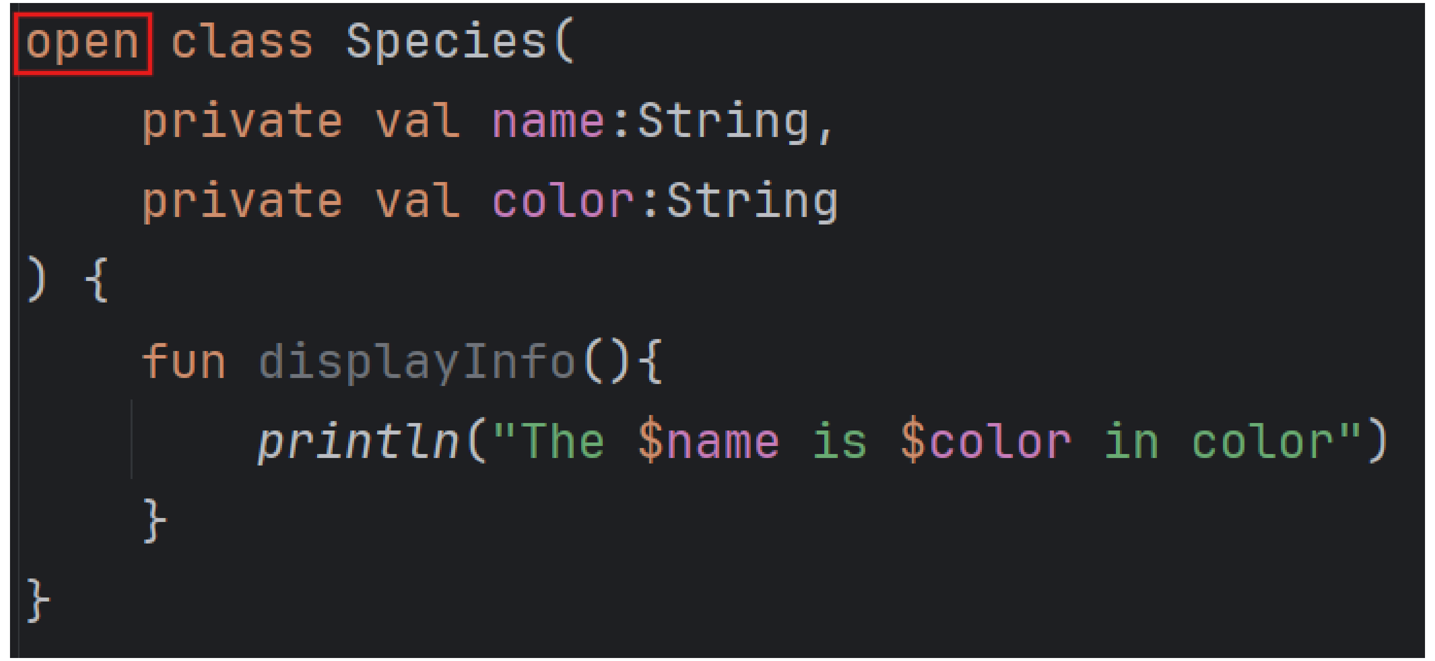
- All classes in Kotlin are *final* by default
- To allow a class to be inherited we need to define it with the open



```
open class Species(
    private val name:String,
    private val color:String
) {
    fun displayInfo(){
        println("The $name is $color in color")
    }
}
```

继承（续）

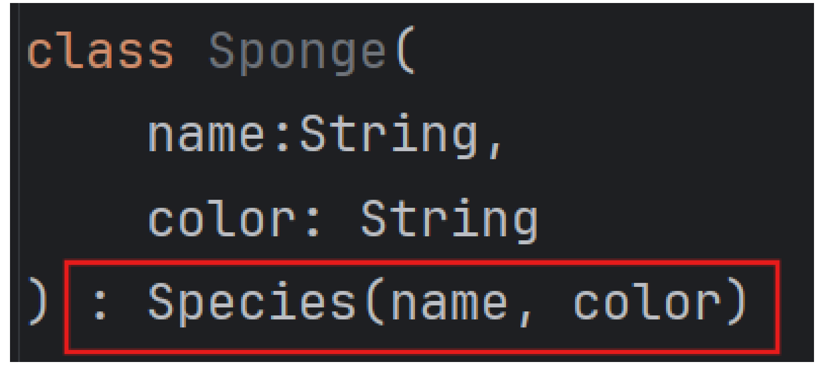
- Kotlin 中的所有类默认都是 *final* 的
- 要允许一个类被继承，我们需要用open关键字来定义它



```
open class Species(
    private val name:String,
    private val color:String
) {
    fun displayInfo(){
        println("The $name is $color in color")
    }
}
```

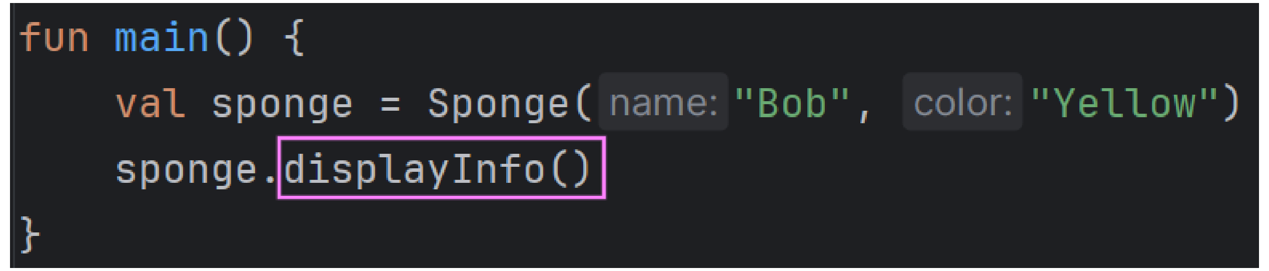
Inheritance (cont.)

- When inheriting another class, we use the syntax below



```
class Sponge(
    name:String,
    color: String
) : Species(name, color)
```

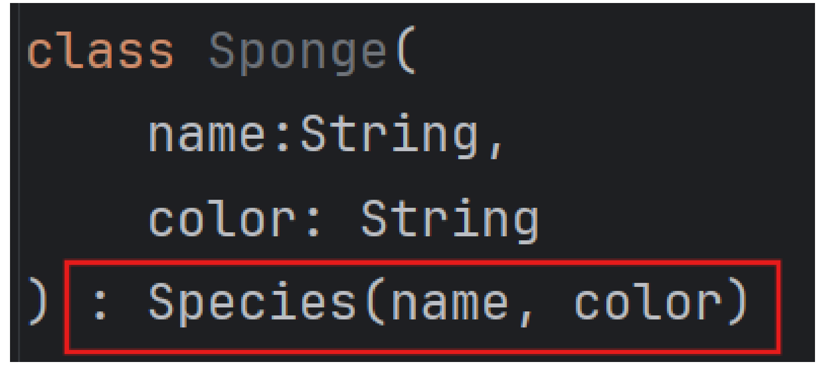
- Now when we create a new object, the child class can use parent class members



```
fun main() {
    val sponge = Sponge( name: "Bob",  color: "Yellow")
    sponge.displayInfo()
}
```

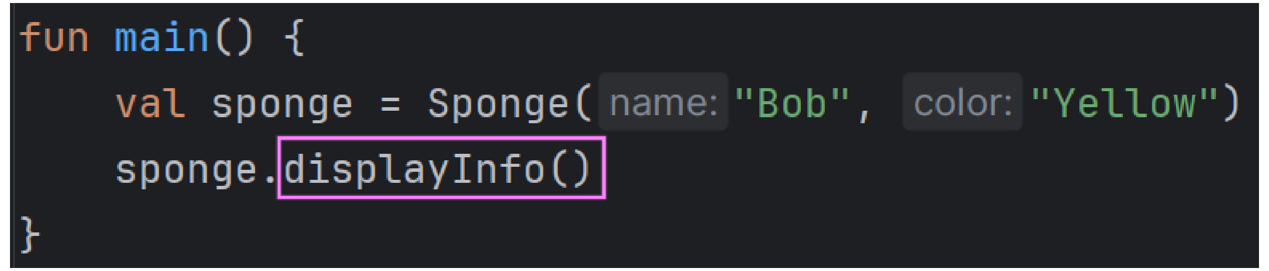
继承（续）

- 继承另一个类时，我们使用以下语法



```
class Sponge(
    name:String,
    color: String
) : Species(name, color)
```

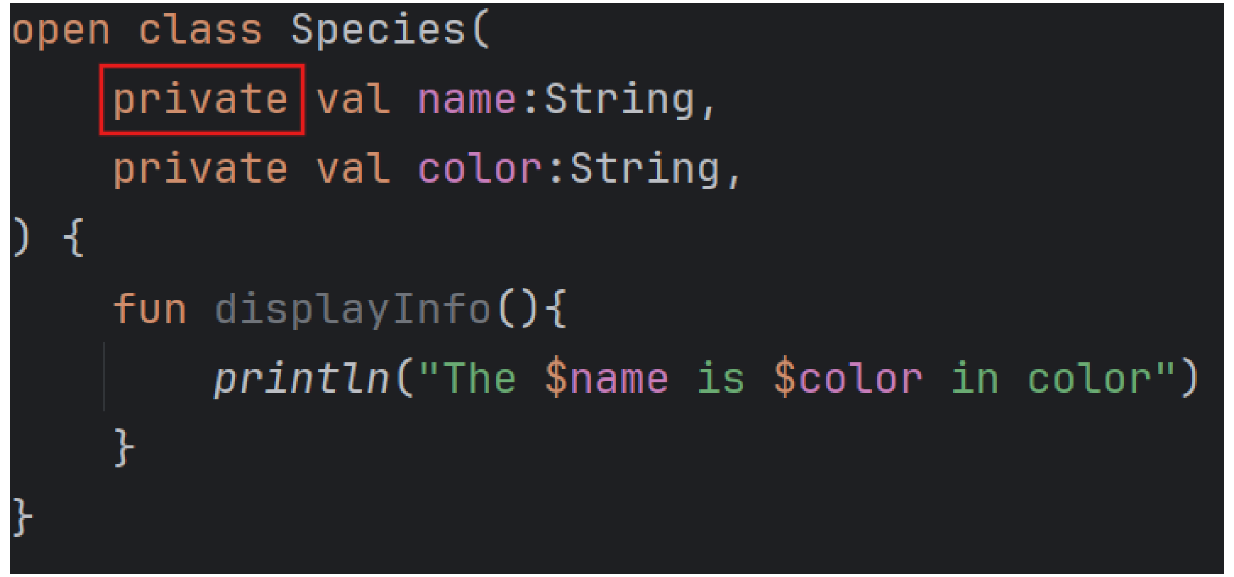
- 现在当我们创建一个新对象时，子类 可以使用父类成员



```
fun main() {
    val sponge = Sponge( name: "Bob",  color: "Yellow")
    sponge.displayInfo()
}
```

Private vs protected

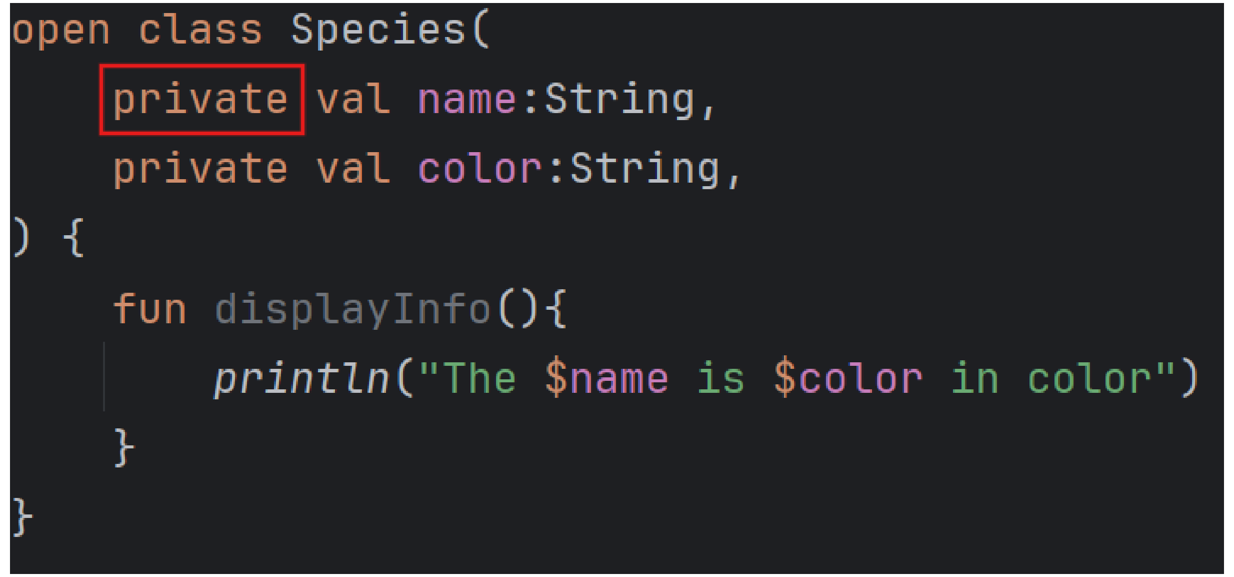
- If any member in the parent class is private



```
open class Species(
    private val name:String,
    private val color:String,
) {
    fun displayInfo(){
        println("The $name is $color in color")
    }
}
```

私有与受保护

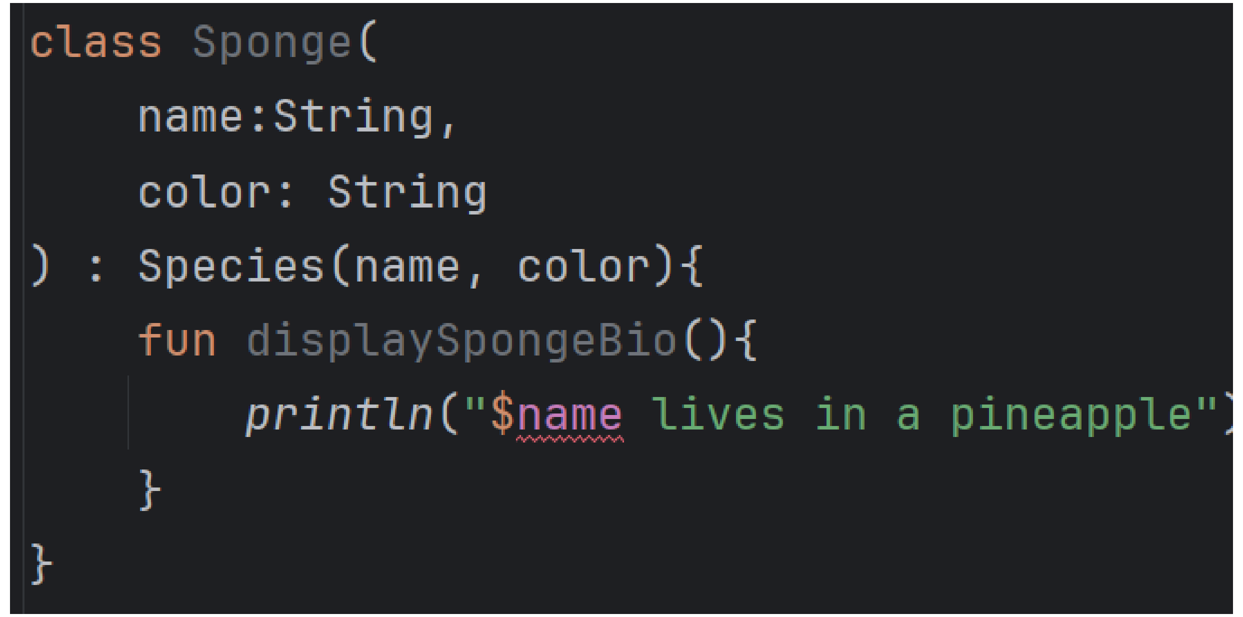
- 如果父类中的任何成员是 私有的



```
open class Species(
    private val name:String,
    private val color:String,
) {
    fun displayInfo(){
        println("The $name is $color in color")
    }
}
```

Private vs protected (cont.)

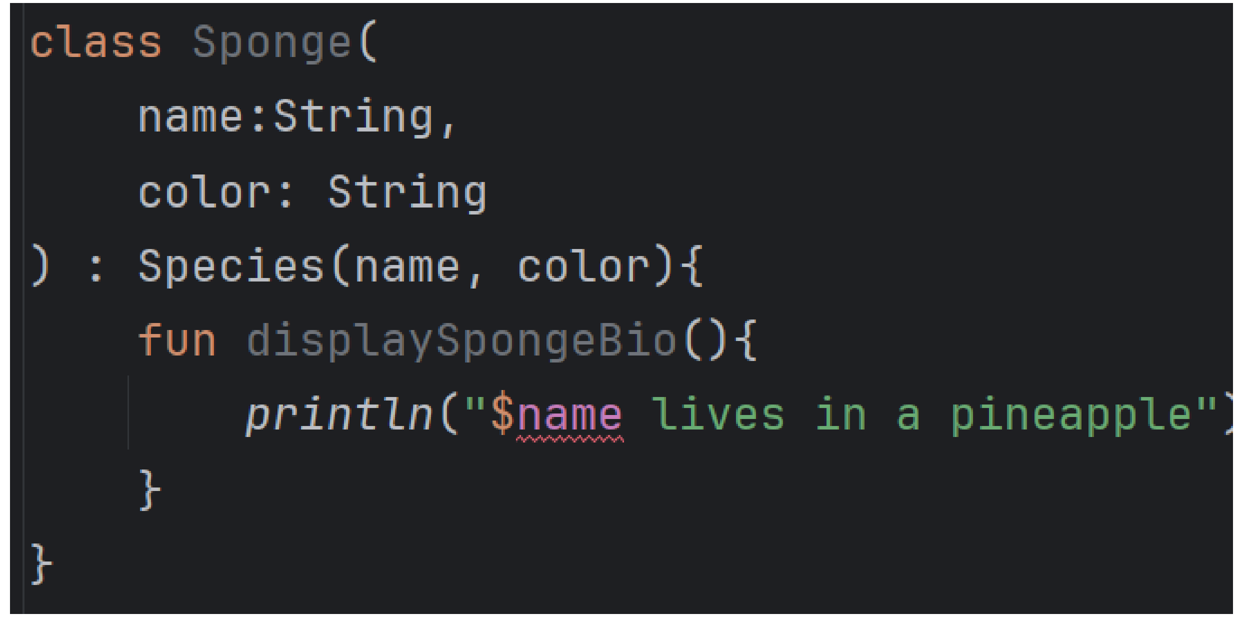
- The child class can't access them



```
class Sponge(
    name:String,
    color: String
) : Species(name, color){
    fun displaySpongeBio(){
        println("$name lives in a pineapple")
    }
}
```

私有与受保护（续）

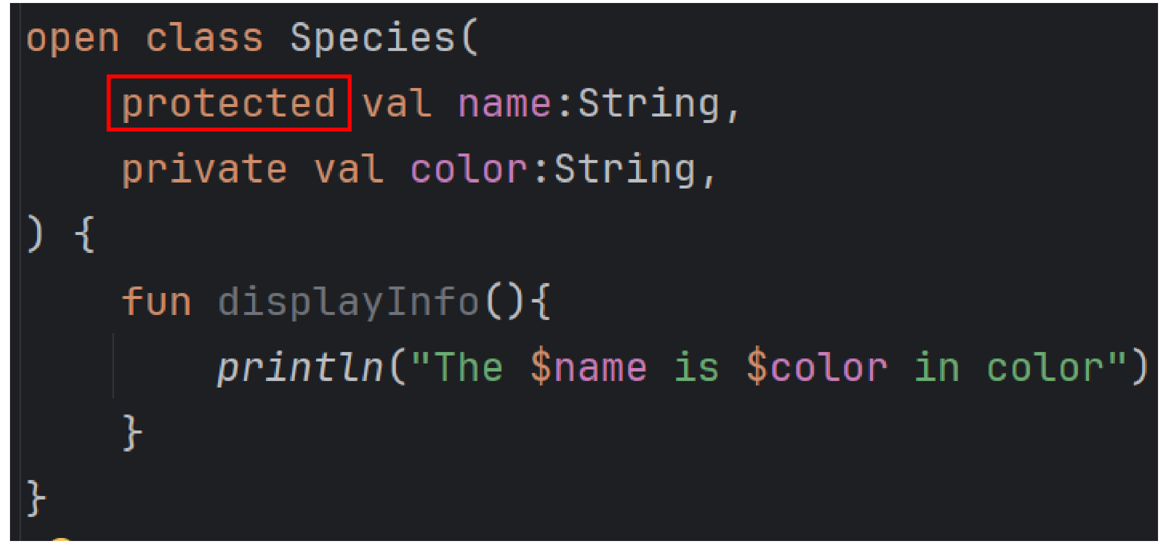
- 子类无法访问它们



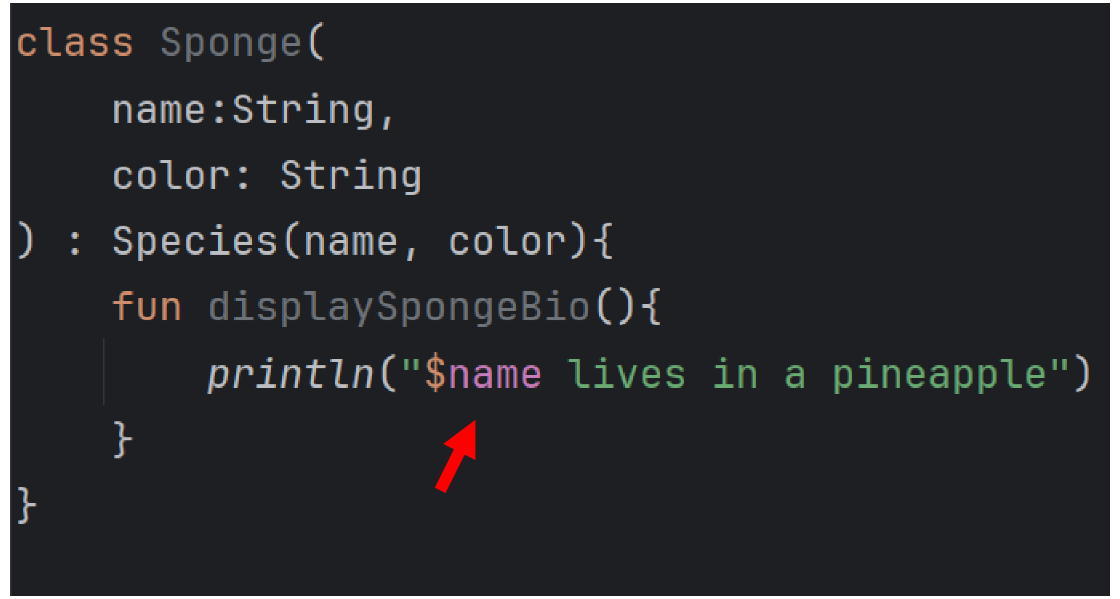
```
class Sponge(
    name:String,
    color: String
) : Species(name, color){
    fun displaySpongeBio(){
        println("$name lives in a pineapple")
    }
}
```

Private vs protected (cont.)

- When we use the visibility modifier protected
  - We can keep that member private but available to all subclasses



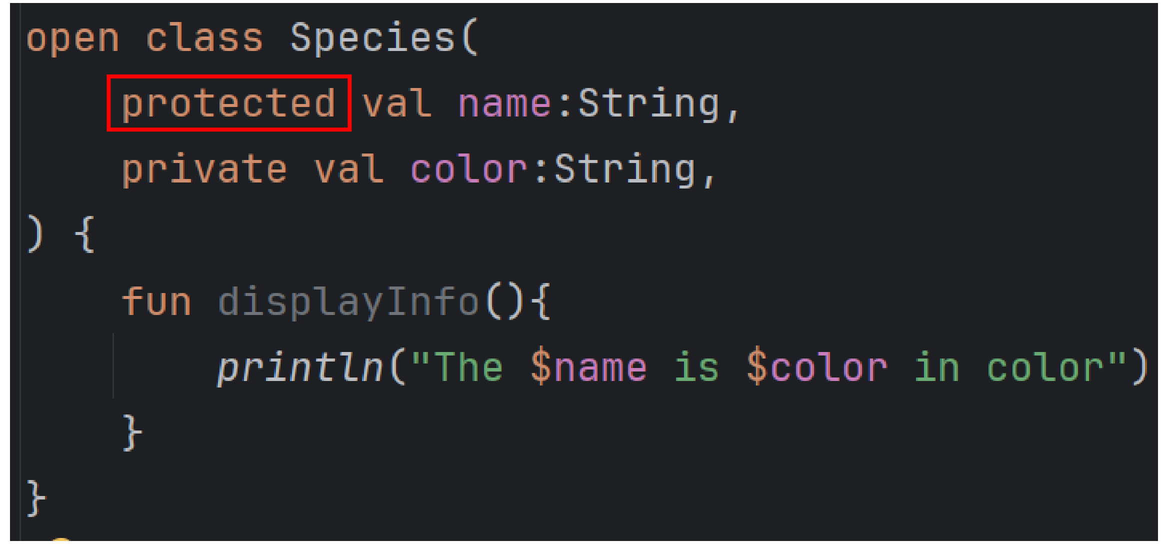
```
open class Species(
    protected val name:String,
    private val color:String,
) {
    fun displayInfo(){
        println("The $name is $color in color")
    }
}
```



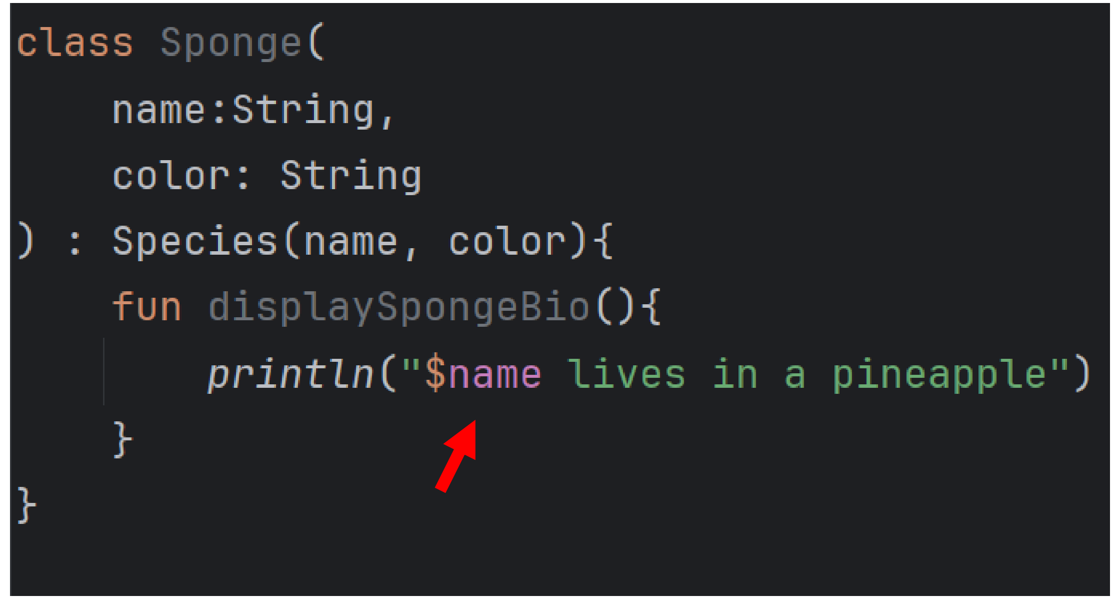
```
class Sponge(
    name:String,
    color: String
) : Species(name, color){
    fun displaySpongeBio(){
        println("$name lives in a pineapple")
    }
}
```

私有与受保护（续）

- 当我们使用可见性修饰符 protected
  - 我们可以使该成员保持私有，但对所有子类可见

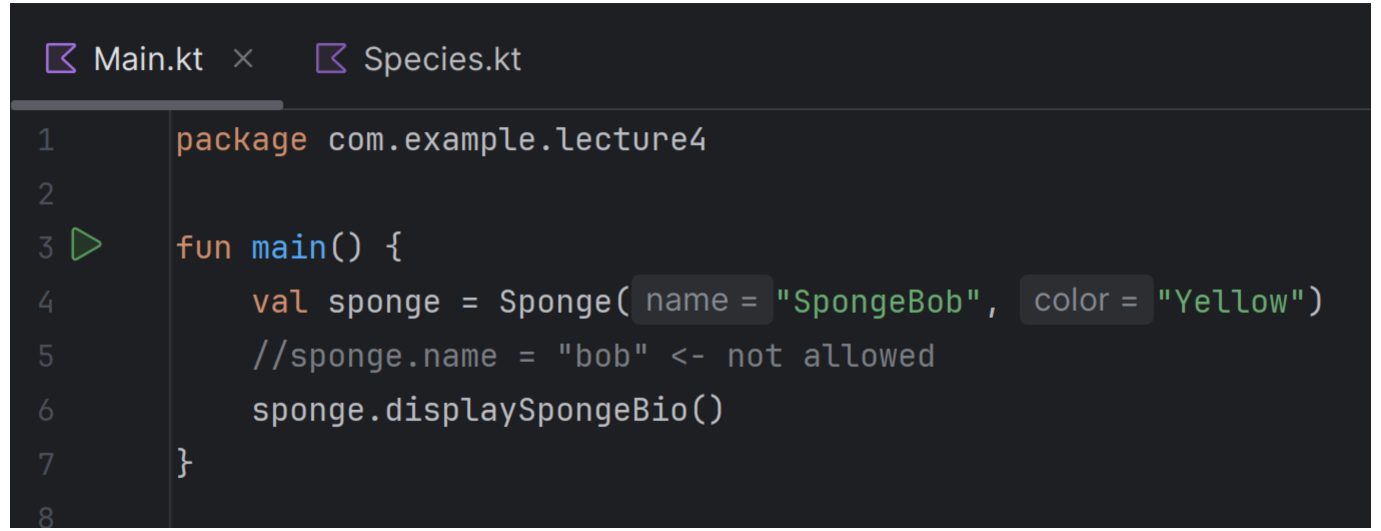


```
open class Species(
    protected val name:String,
    private val color:String,
) {
    fun displayInfo(){
        println("The $name is $color in color")
    }
}
```

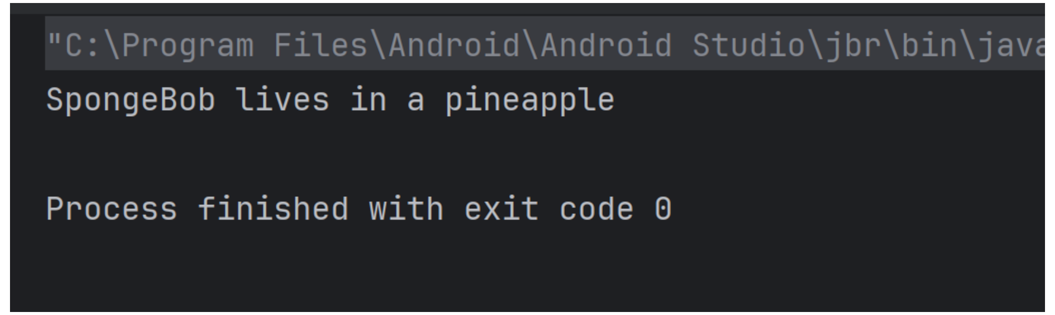


```
class Sponge(
    name:String,
    color: String
) : Species(name, color){
    fun displaySpongeBio(){
        println("$name lives in a pineapple")
    }
}
```

Private vs protected (cont.)



```
package com.example.lecture4

fun main() {
    val sponge = Sponge( name = "SpongeBob", color = "Yellow")
    //sponge.name = "bob" <- not allowed
    sponge.displaySpongeBio()
}
```



```
"C:\Program Files\Android\Android Studio\jbr\bin\java
SpongeBob lives in a pineapple

Process finished with exit code 0
```

私有与受保护（续）

```
package com.example.lecture4

fun main() {
    val sponge = Sponge( name = "SpongeBob", color = "Yellow")
    //sponge.name = "bob" <- not allowed
    sponge.displaySpongeBio()
}
```

```
"C:\Program Files\Android\Android Studio\jbr\bin\java
SpongeBob lives in a pineapple

Process finished with exit code 0
```

Abstract classes

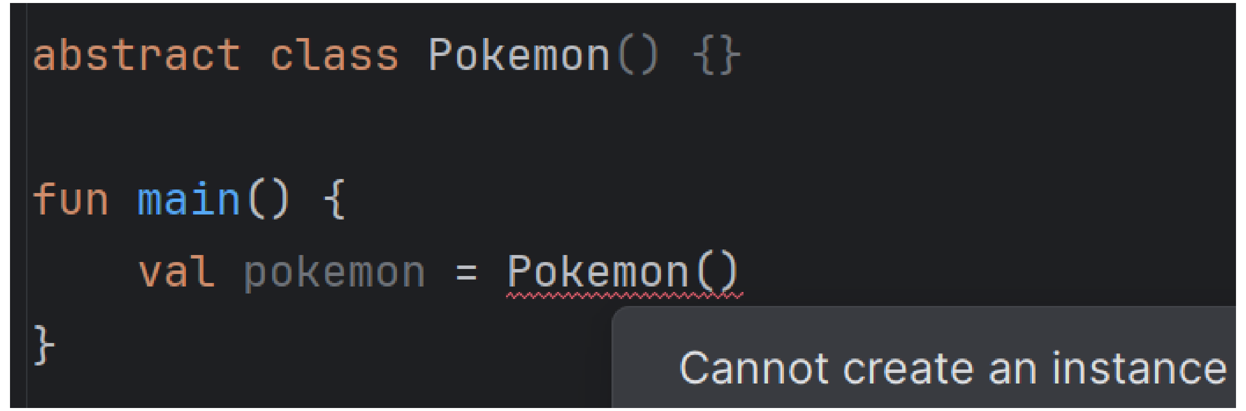
- An abstract class allows us to define a class with abstract members
- This allows us to create member definitions that subclasses must fulfill (aka. a contract)
- This allows us to provide a common interface in our code while hiding the subclass implementation (aka. Abstraction)

抽象类

- 抽象类允许我们定义包含抽象成员的类
- 这使我们能够创建子类必须实现的成员定义（即契约）
- 这使我们能够在代码中提供一个公共接口，同时隐藏子类的实现细节（即抽象）

Abstract classes (cont.)

- We cannot create an instance of an abstract class

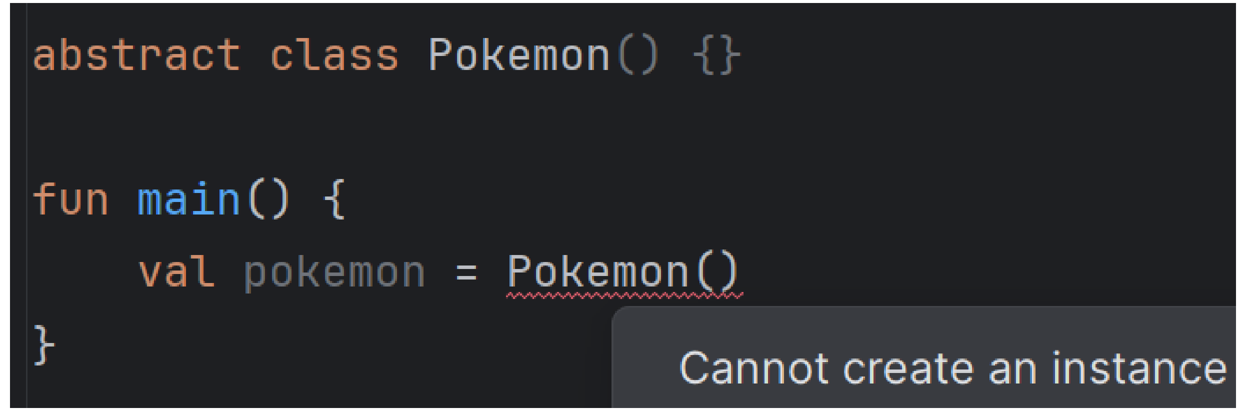


```
abstract class Pokemon() {}

fun main() {
    val pokemon = Pokemon()
}
```

Cannot create an instance

抽象类（续）

- 我们不能创建一个 抽象类的实例

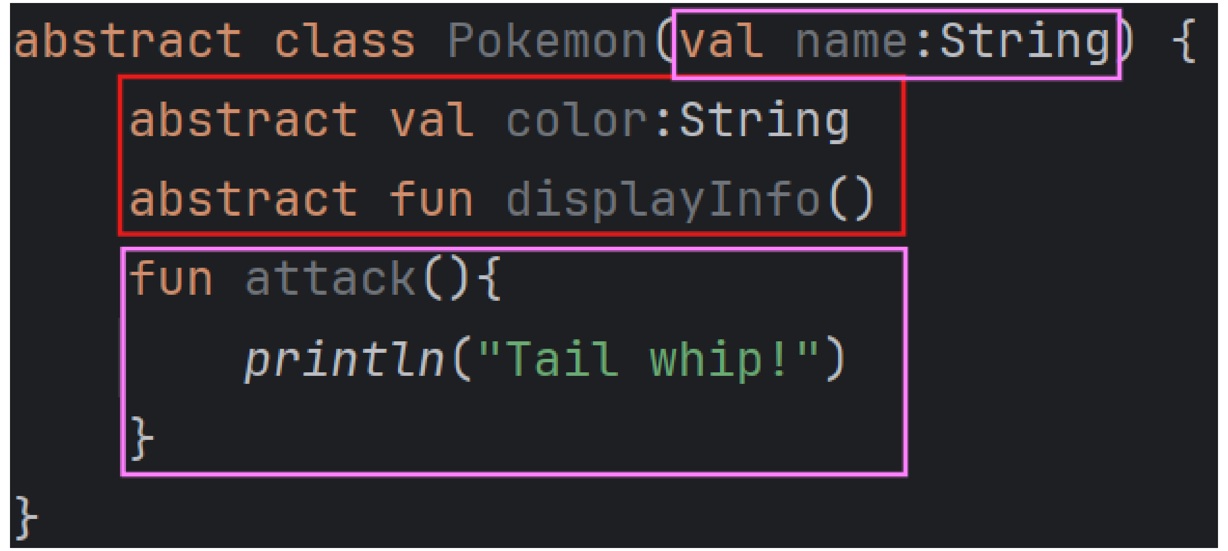


```
abstract class Pokemon() {}

fun main() {
    val pokemon = Pokemon()
}
```

Cannot create an instance

Abstract classes (cont.)

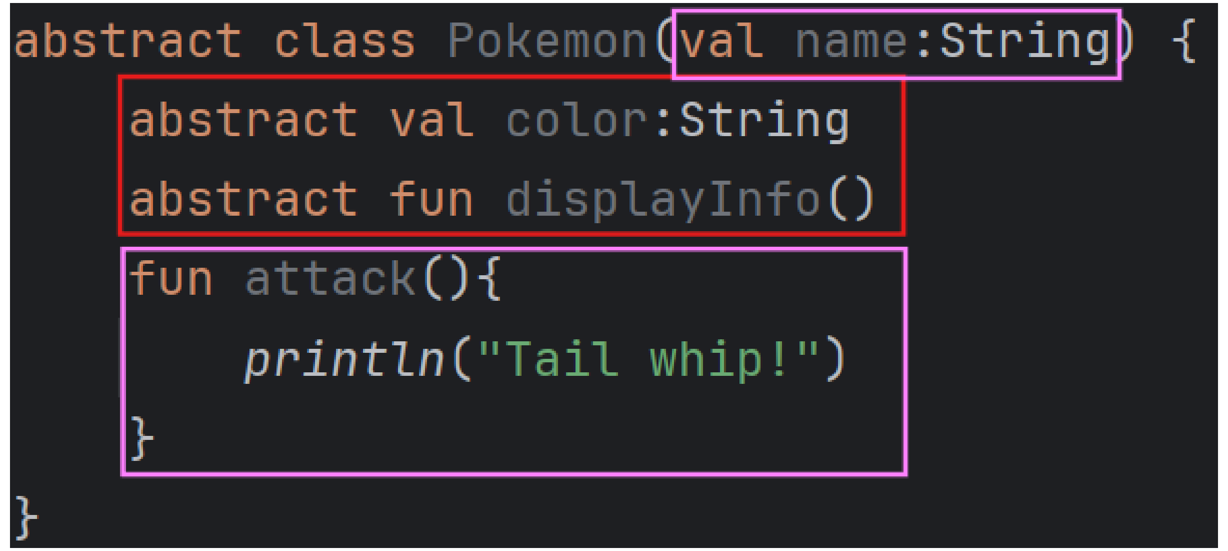
- Abstract members are defined using abstract, abstract classes can also have state and regular methods



```
abstract class Pokemon(val name:String) {
    abstract val color:String
    abstract fun displayInfo()
    fun attack(){
        println("Tail whip!")
    }
}
```

抽象类（续）

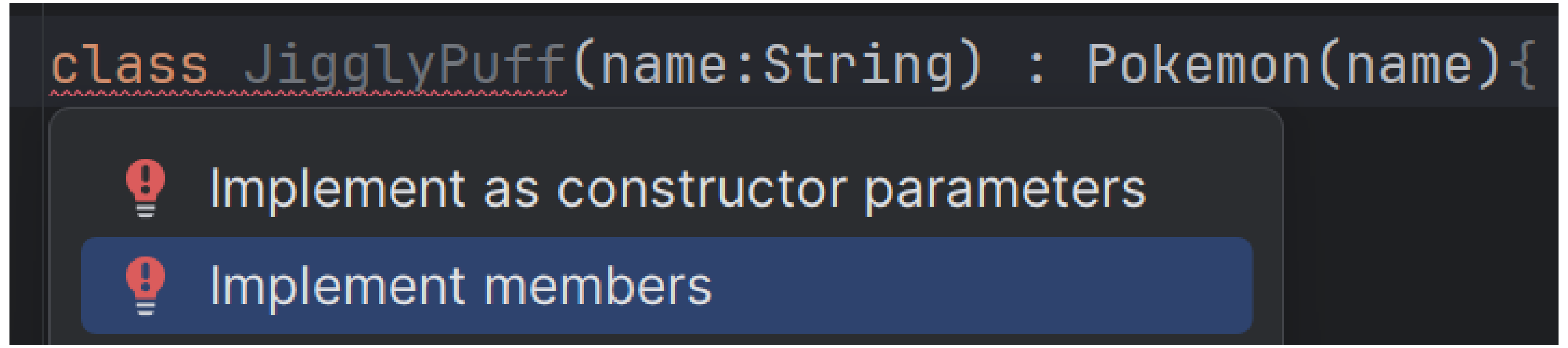
- 抽象成员使用 abstract，抽象类还可以具有 状态和普通方法



```
abstract class Pokemon(val name:String) {
    abstract val color:String
    abstract fun displayInfo()
    fun attack(){
        println("Tail whip!")
    }
}
```

Abstract classes (cont.)

- When we inherit our abstract class, we must implement the abstract members



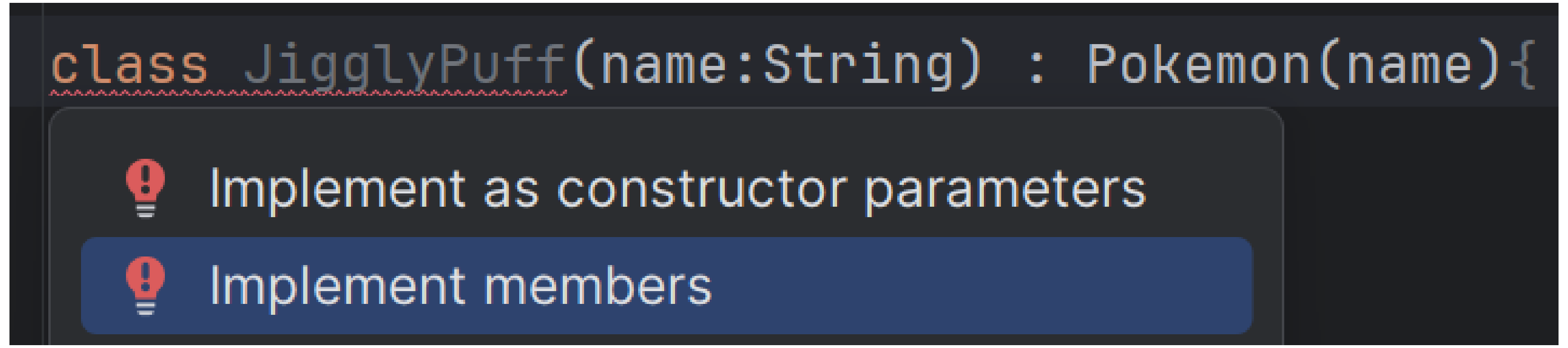
```
class JigglyPuff(name:String) : Pokemon(name){
```

Implement as constructor parameters

Implement members

抽象类（续）

- 继承抽象类时，必须实现其抽象成员



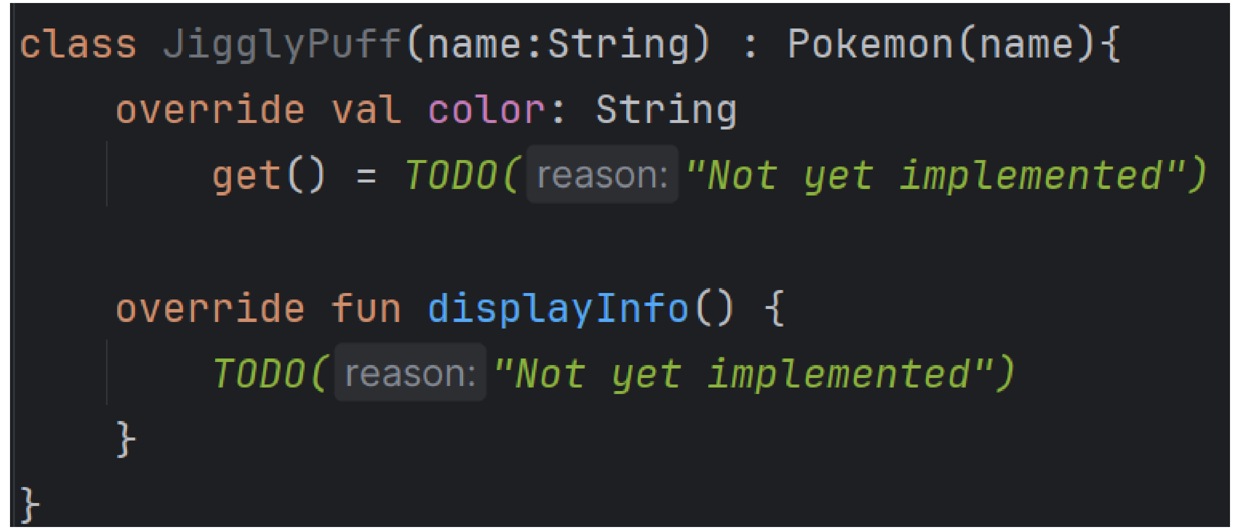
```
class JigglyPuff(name:String) : Pokemon(name){
```

Implement as constructor parameters

Implement members

Abstract classes (cont.)

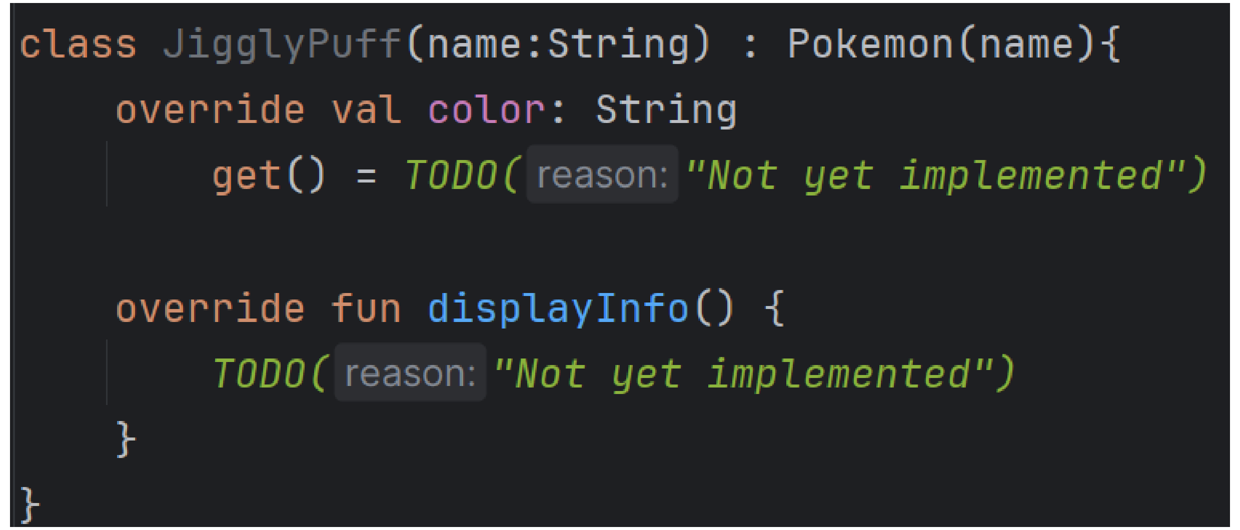
- We implement abstract members by using the override keyword



```
class JigglyPuff(name:String) : Pokemon(name){
    override val color: String
        get() = TODO( reason: "Not yet implemented")

    override fun displayInfo() {
        TODO( reason: "Not yet implemented")
    }
}
```

抽象类（续）

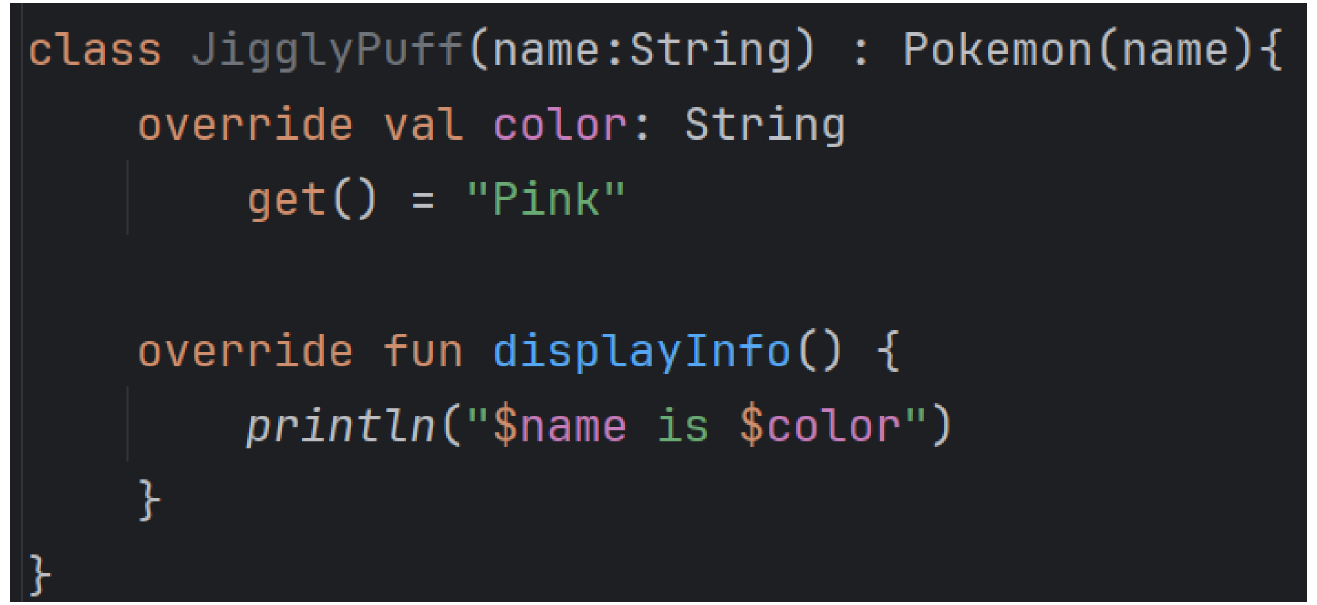
- 我们通过使用 override 关键字来实现抽象成员



```
class JigglyPuff(name:String) : Pokemon(name){
    override val color: String
        get() = TODO( reason: "Not yet implemented")

    override fun displayInfo() {
        TODO( reason: "Not yet implemented")
    }
}
```

Abstract classes (cont.)

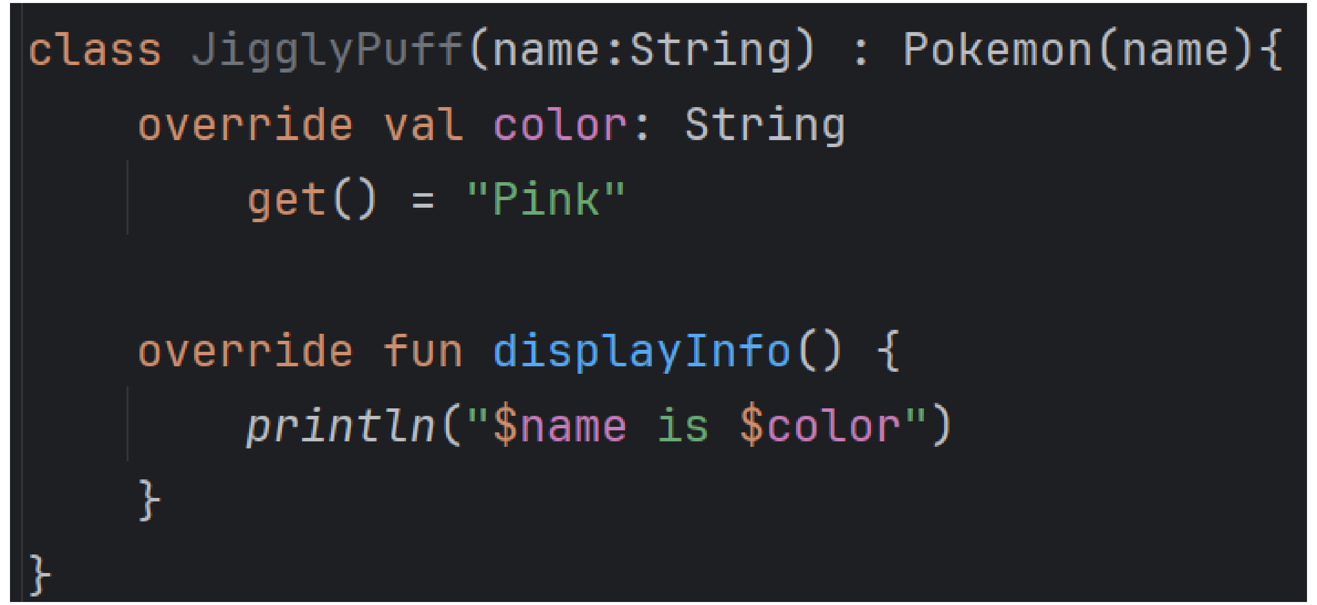
- The subclass can then provide its own implementation of the members



```
class JigglyPuff(name:String) : Pokemon(name){
    override val color: String
        get() = "Pink"

    override fun displayInfo() {
        println("$name is $color")
    }
}
```

抽象类（续）

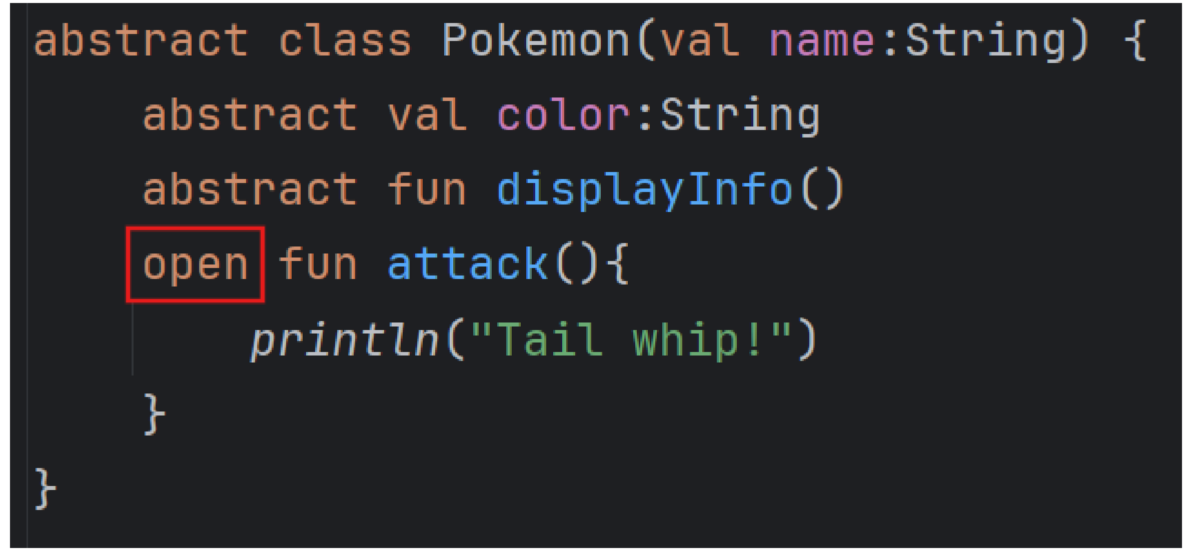
- 子类随后可以为其成员提供自己的实现



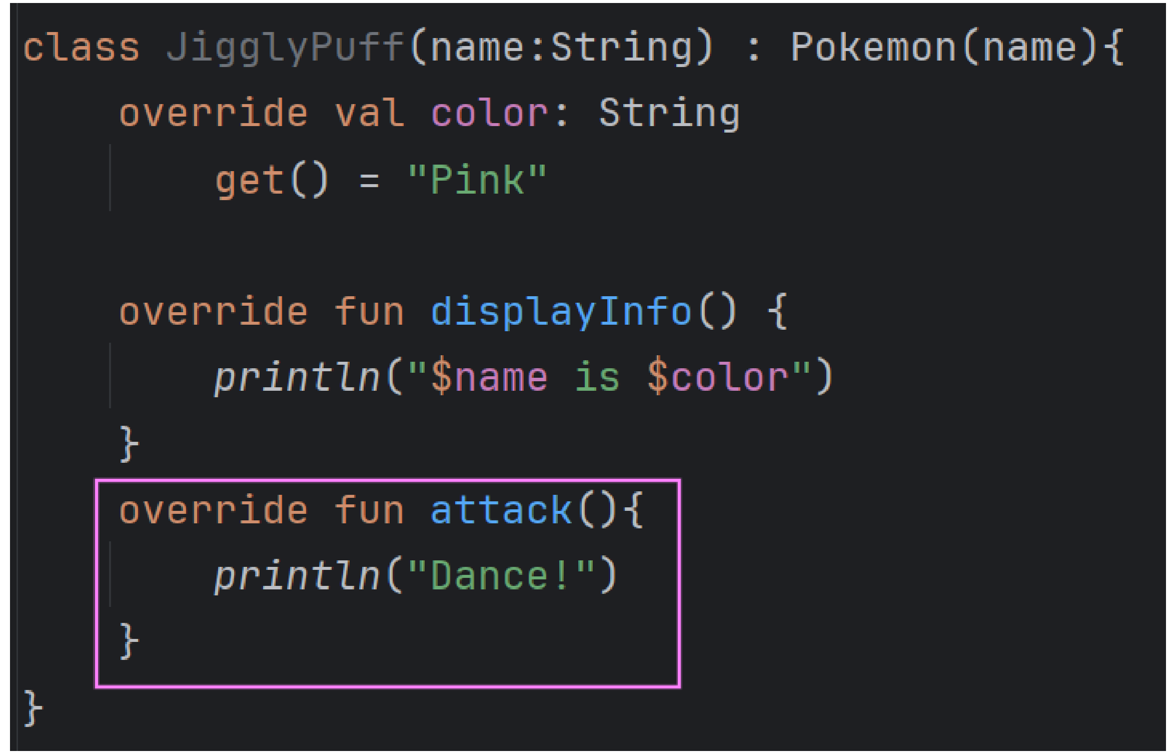
```
class JigglyPuff(name:String) : Pokemon(name){
    override val color: String
        get() = "Pink"

    override fun displayInfo() {
        println("$name is $color")
    }
}
```

Overriding members (cont.)

- To override a non abstract member, we use the open keyword



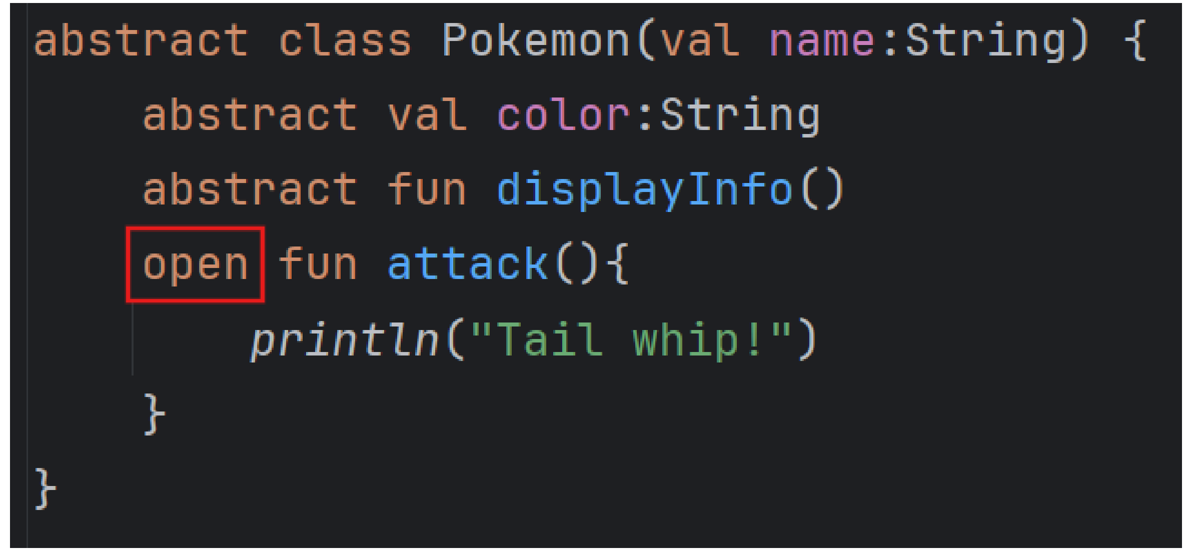
```
abstract class Pokemon(val name:String) {
    abstract val color:String
    abstract fun displayInfo()
    open fun attack(){
        println("Tail whip!")
    }
}
```



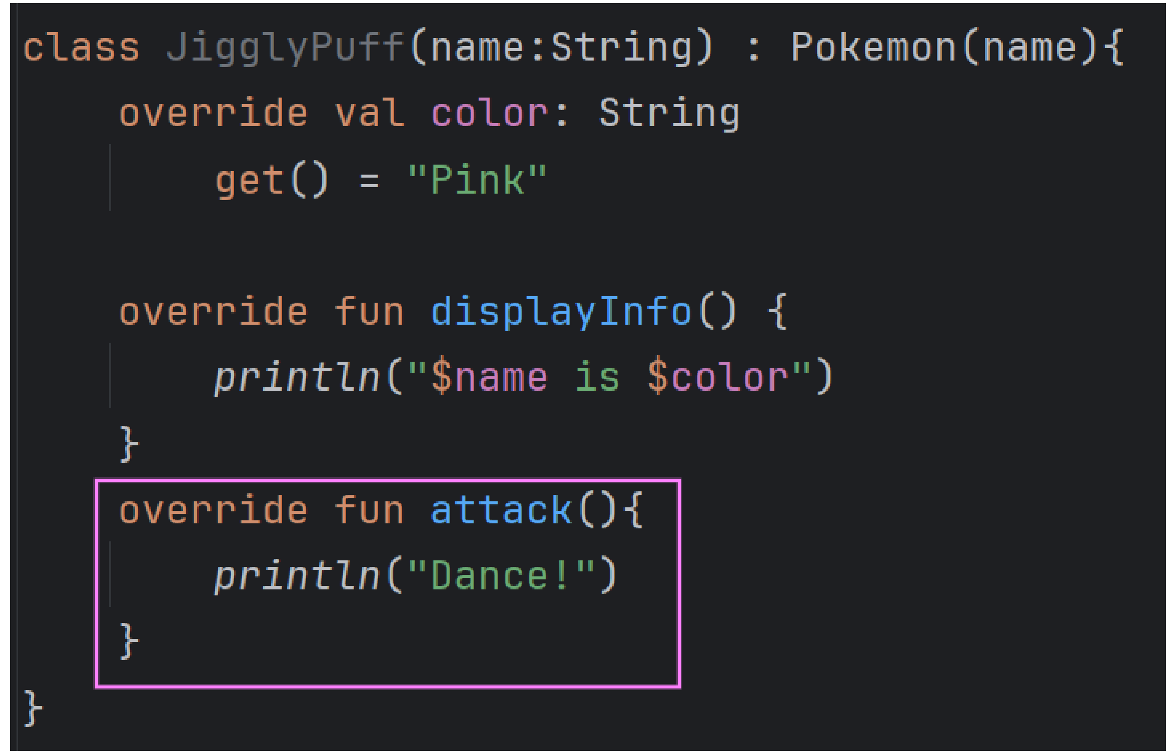
```
class JigglyPuff(name:String) : Pokemon(name){
    override val color: String
        get() = "Pink"

    override fun displayInfo() {
        println("$name is $color")
    }
    override fun attack(){
        println("Dance!")
    }
}
```

重写成员（续）

- 要重写一个非抽象成员，我们使用open关键字



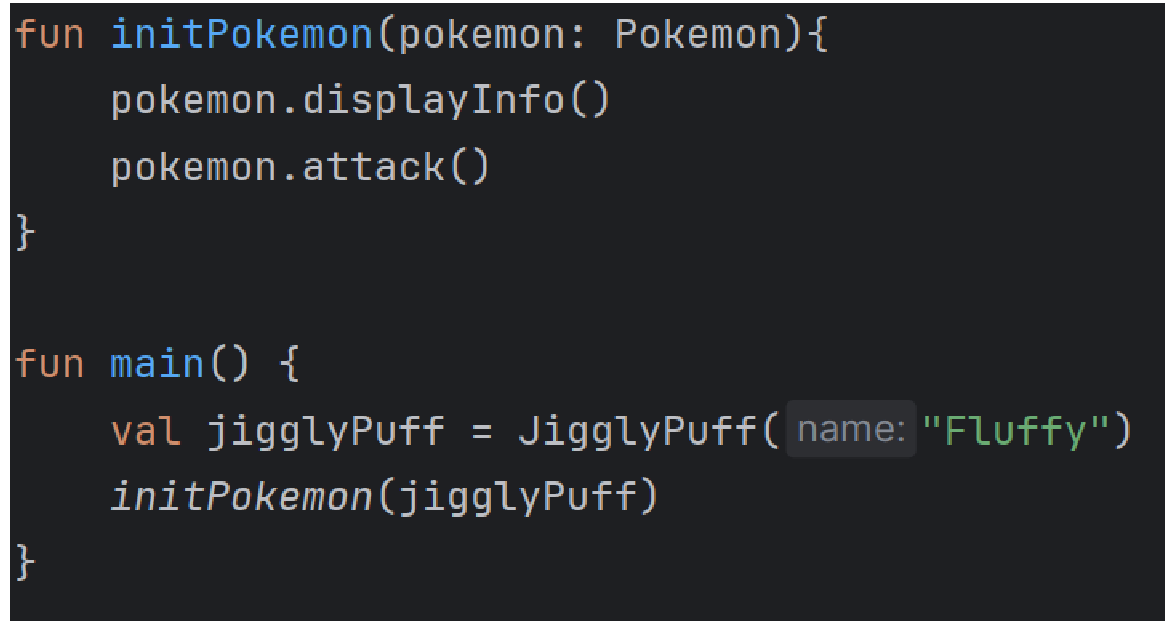
```
abstract class Pokemon(val name:String) {
    abstract val color:String
    abstract fun displayInfo()
    open fun attack(){
        println("Tail whip!")
    }
}
```



```
class JigglyPuff(name:String) : Pokemon(name){
    override val color: String
        get() = "Pink"

    override fun displayInfo() {
        println("$name is $color")
    }
    override fun attack(){
        println("Dance!")
    }
}
```

Abstraction (cont.)

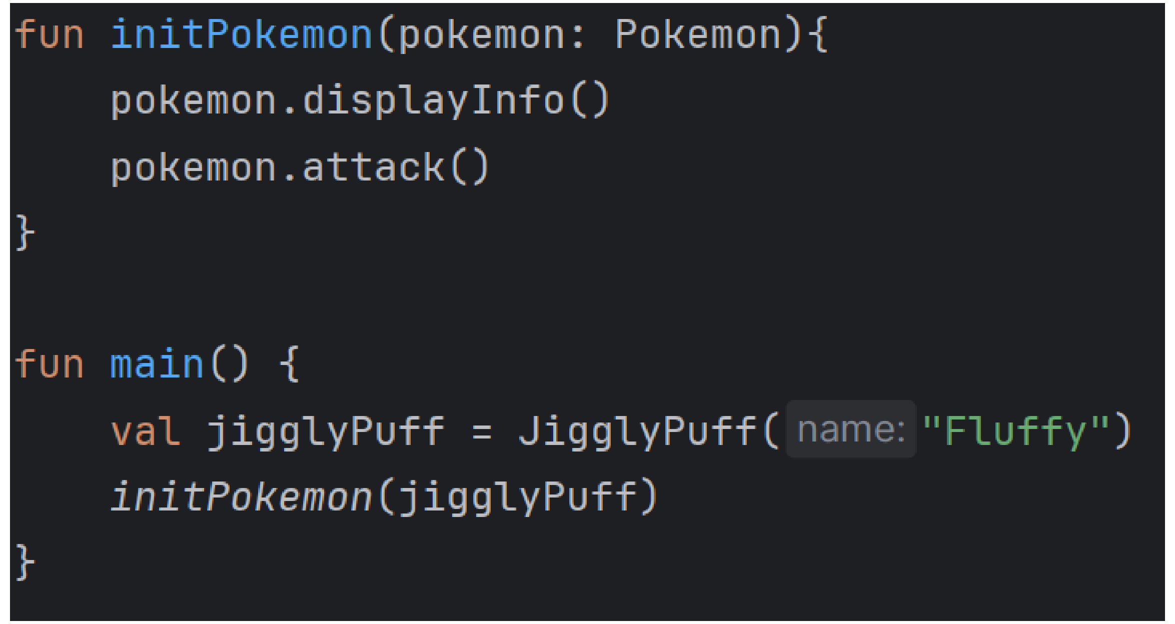
- Here we use the common interface with our *initPokemon* function
  - All it is concerned about is that it’s a Pokemon
  - It has not clue it is a JigglyPuff



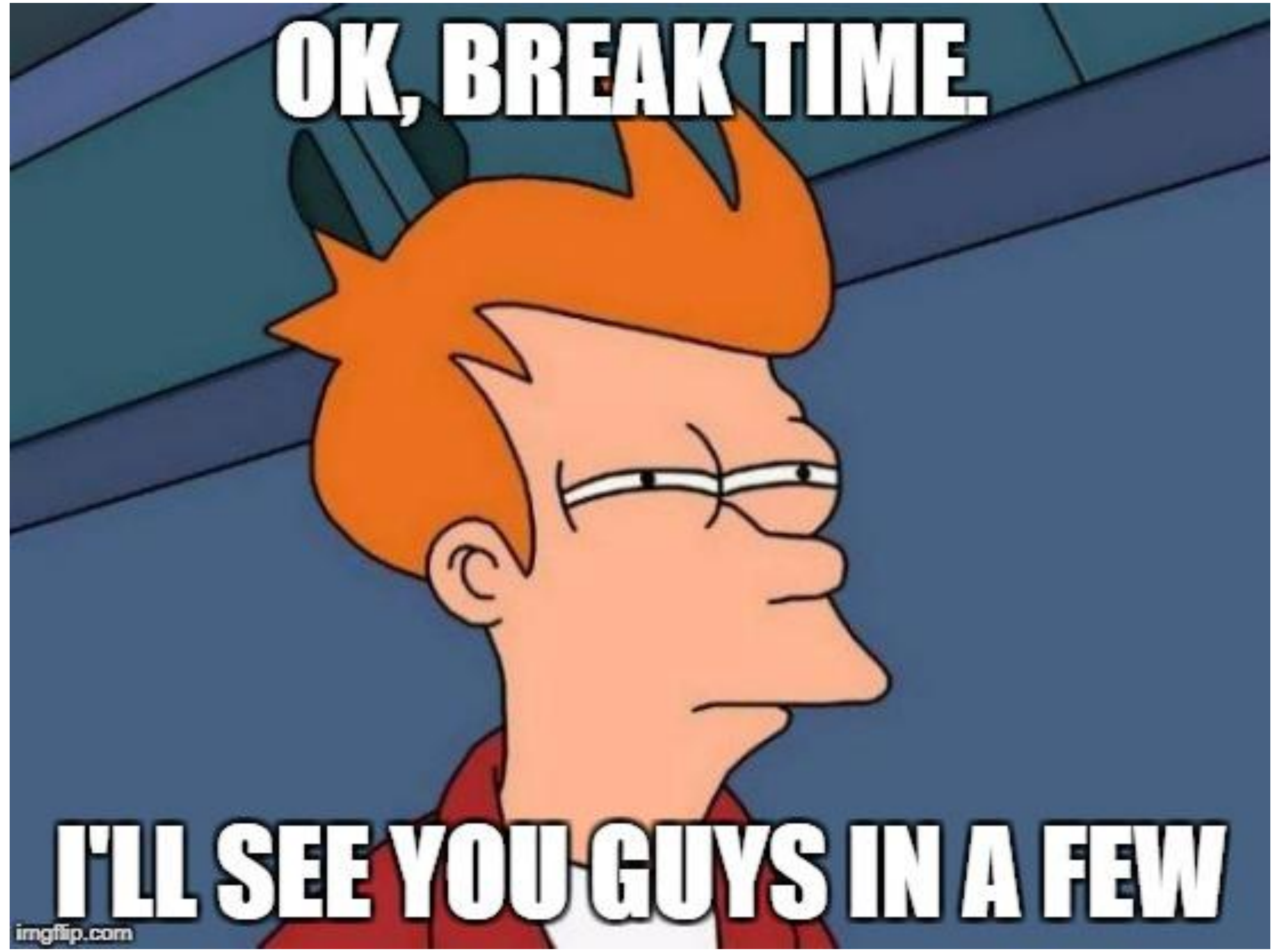
```
fun initPokemon(pokemon: Pokemon){
    pokemon.displayInfo()
    pokemon.attack()
}

fun main() {
    val jigglyPuff = JigglyPuff( name: "Fluffy")
    initPokemon(jigglyPuff)
}
```

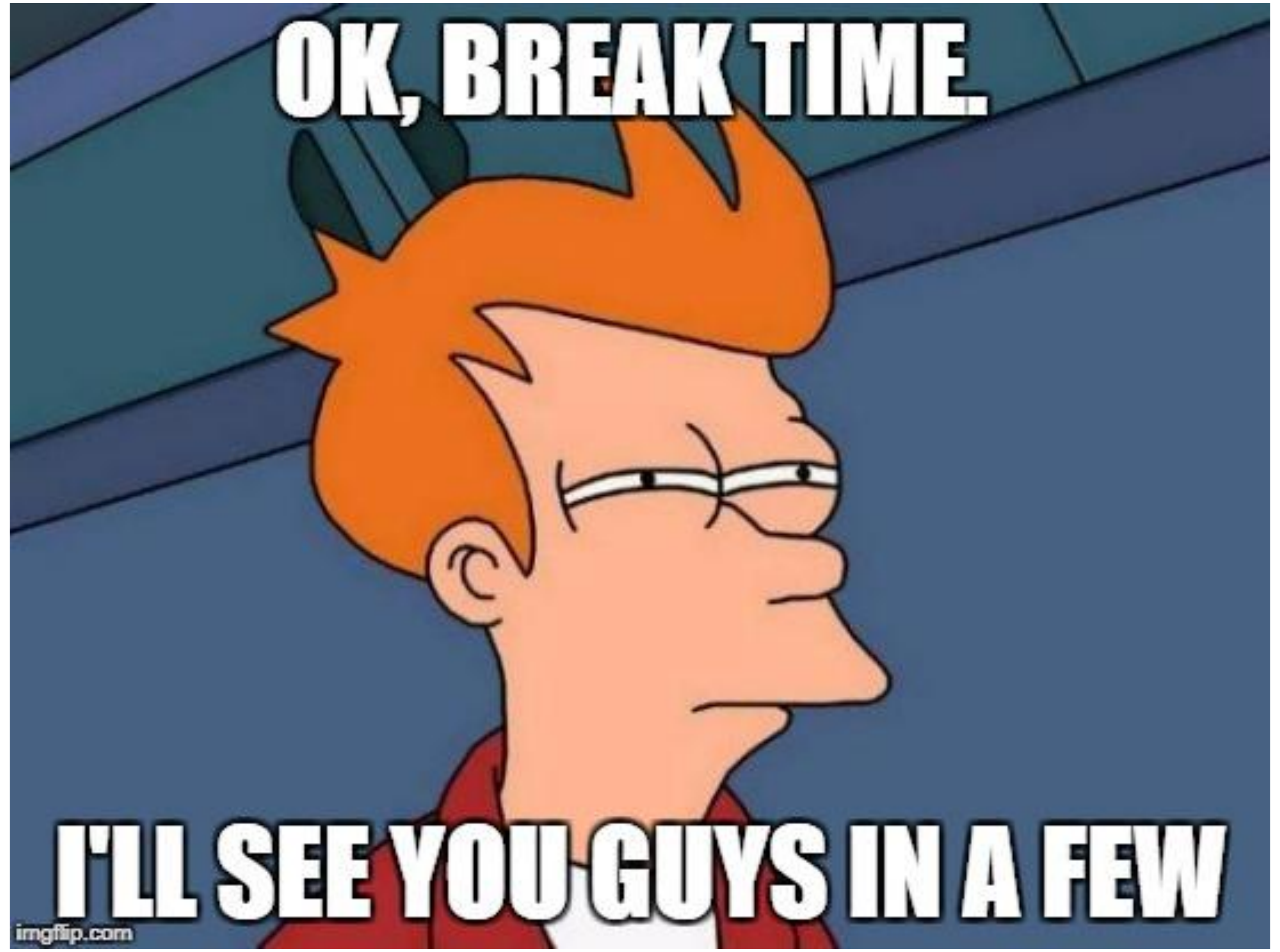
抽象（续）

- 我们在此使用带有 *initPokemon* 函数的通用接口
  - 它只关心这是一个宝可梦
  - 它完全不知道这是一只胖丁



```
fun initPokemon(pokemon: Pokemon){
    pokemon.displayInfo()
    pokemon.attack()
}

fun main() {
    val jigglyPuff = JigglyPuff( name: "Fluffy")
    initPokemon(jigglyPuff)
}
```


OK, BREAK TIME.
I'LL SEE YOU GUYS IN A FEW


OK, BREAK TIME.
I'LL SEE YOU GUYS IN A FEW

Interfaces

- The problem with abstract classes is we can only inherit from one

- It was a design choice when creating Kotlin to not allow multiple class inheritance
  - It can be problematic involving multiple parent class initializations

- An interface is a type of abstract class that allows us to maintain abstraction without the problems of multiple class inheritance

接口

- 抽象类的问题在于我们只能继承一个

- 在创建 Kotlin 时不允许一个类多重继承，这是一个设计上的选择继承
  - 涉及多个父类初始化时可能会带来问题

- 接口是一种抽象类，它使我们能够在避免多重类继承问题的同时保持抽象性
抽象而不会带来多重类继承的问题

Interfaces (cont.)

- Interfaces do everything an abstract class does, except
  - They don’t have a constructor/init method
  - They don’t have state

- A class can implement multiple interfaces

- But why would we even want to inherit from multiple classes?
  - When a program starts to scale and become more complex, we need ways to contain the chaos

接口（续）

- 接口能完成抽象类所做的所有事情，除了
  - 它们没有构造函数或初始化方法
  - 它们没有状态

- 一个类可以实现多个接口

- 但我们为何需要从多个类继承呢?
  - 当程序开始扩展并变得更加复杂时，我们需要方法来控制混乱

Interfaces (cont.)

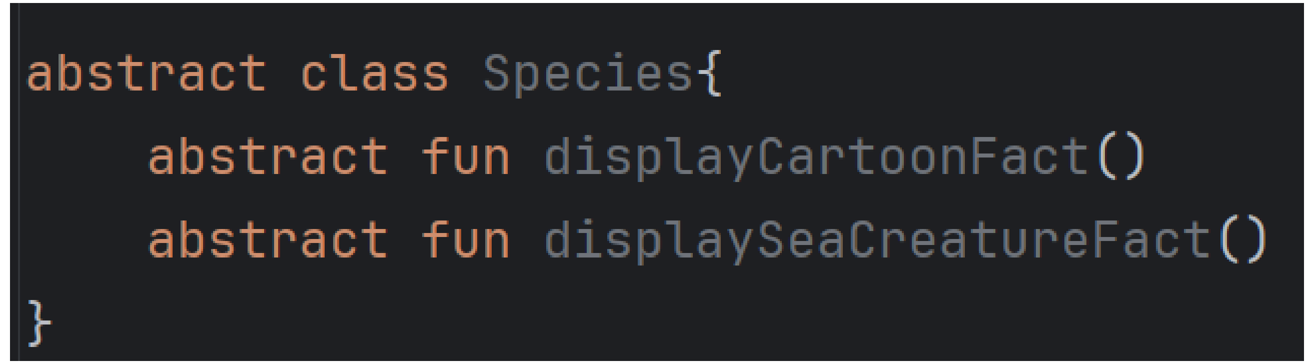
- Let’s assume we are creating an educational SpongeBob program
- Corporate wants us to have an educational aspect to the program so the kids can learn more about sea creatures
- They decided they want some real facts about the actual sea creatures in the show
    - Ex. A crab has a hard outer shell and walks on four legs

接口（续）

- 假设我们正在创建一个教育类的海绵宝宝程序
- 公司希望我们的节目具有教育意义，让孩子们能了解更多关于海洋生物的知识
- 他们决定希望加入一些关于节目中真实海洋生物的科学事实
    - 例如：螃蟹有坚硬的外壳，用四条腿行走

Interfaces (cont.)

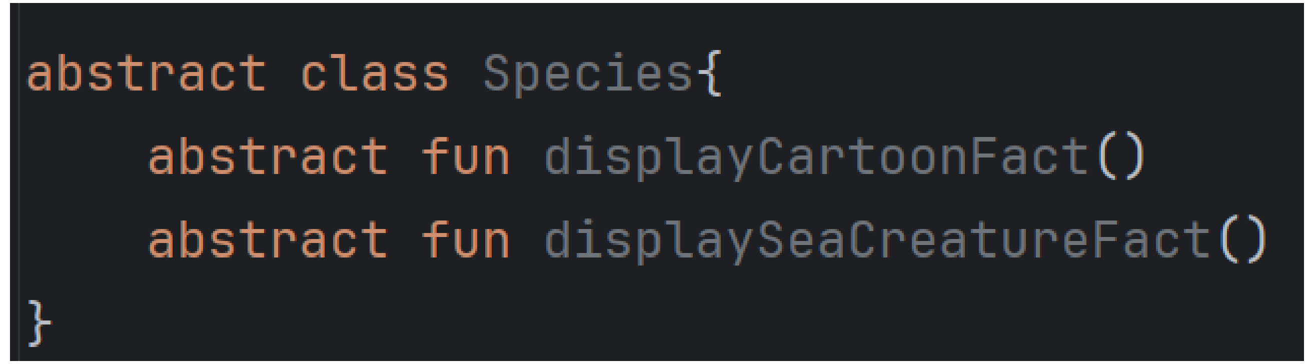
- We could create an abstract Cartoon class and define some abstract members



```
abstract class Species{
    abstract fun displayCartoonFact()
    abstract fun displaySeaCreatureFact()
}
```

接口（续）

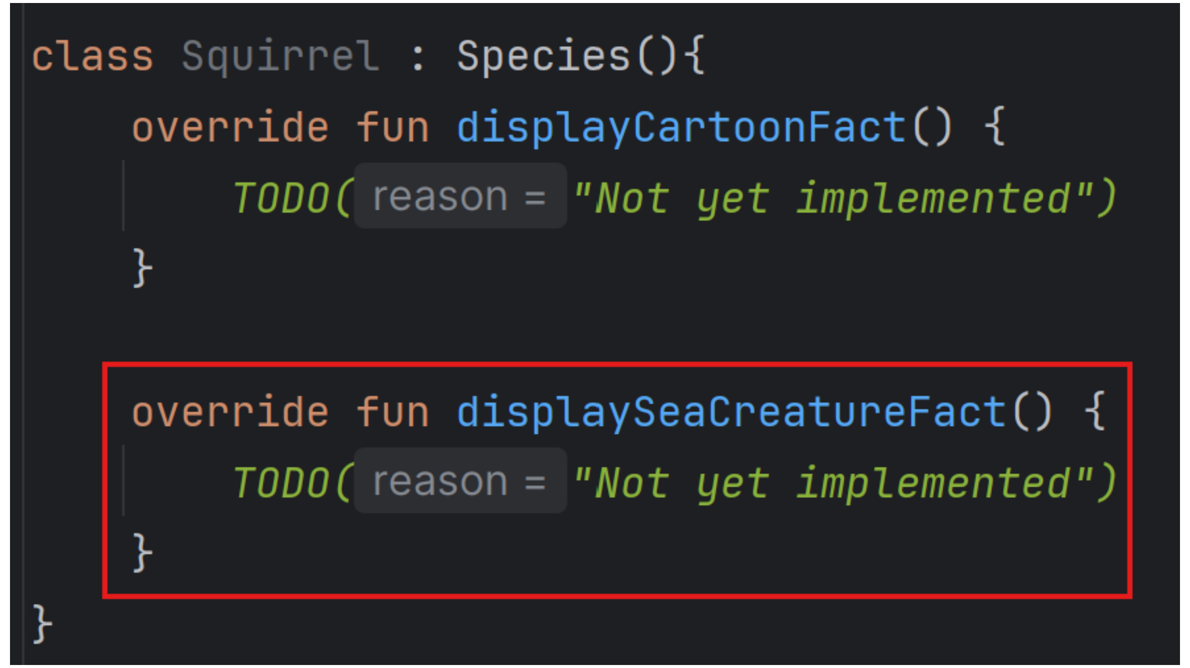
- 我们可以创建一个抽象的 Cartoon 类并定义一些抽象成员



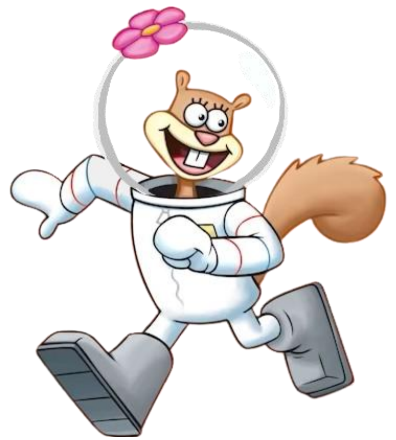
```
abstract class Species{
    abstract fun displayCartoonFact()
    abstract fun displaySeaCreatureFact()
}
```

Interfaces (cont.)

- The problem though is some species in the show aren't actual sea creatures which adds unnecessary complexity

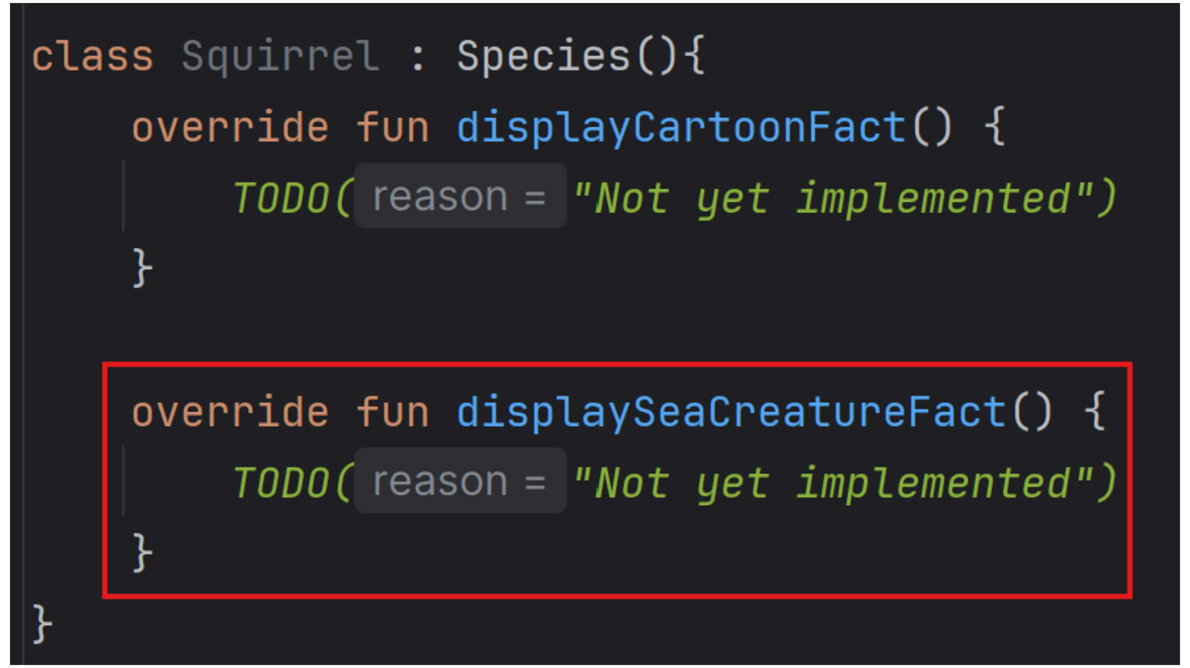


```
class Squirrel : Species(){
    override fun displayCartoonFact() {
        TODO( reason = "Not yet implemented")
    }

    override fun displaySeaCreatureFact() {
        TODO( reason = "Not yet implemented")
    }
}
```

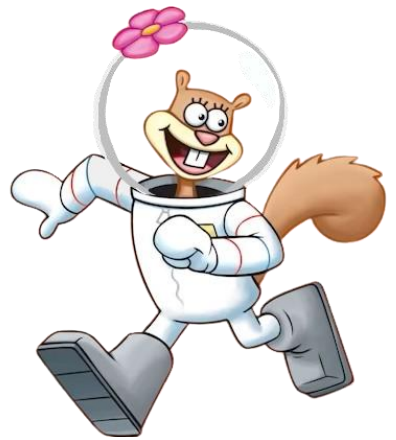


接口（续）

- 但问题在于，剧中的一些物种并非真正的海洋生物，这增加了不必要的复杂性

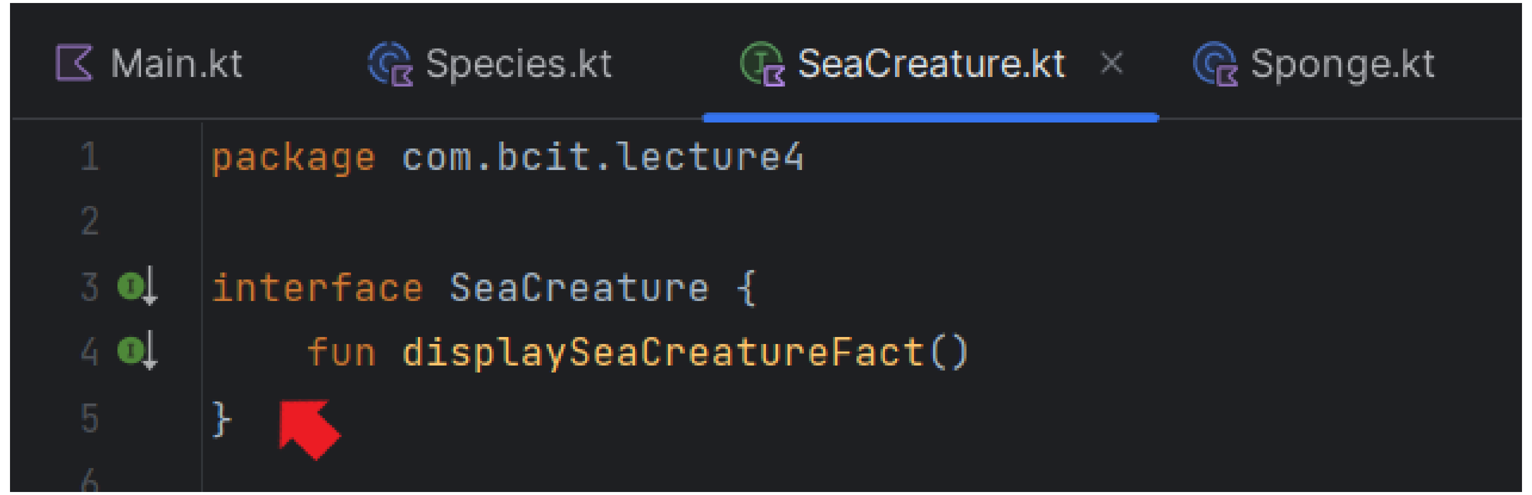


```
class Squirrel : Species(){
    override fun displayCartoonFact() {
        TODO( reason = "Not yet implemented")
    }

    override fun displaySeaCreatureFact() {
        TODO( reason = "Not yet implemented")
    }
}
```



Interfaces (cont.)

- To solve this problem and maintain abstraction we can use an interface

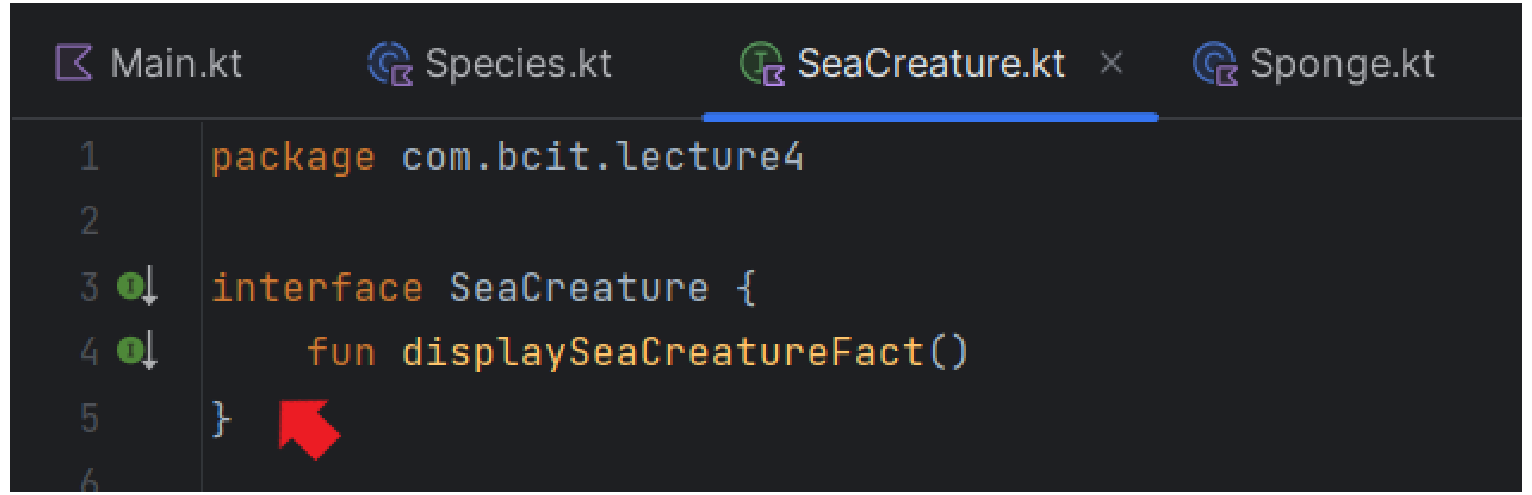


```
package com.bcit.lecture4

interface SeaCreature {
    fun displaySeaCreatureFact()
}
```

- If an interface member doesn't have an implementation, the abstract keyword is inferred

接口（续）

- 为了解决这个问题并保持抽象性，我们可以使用一个接口

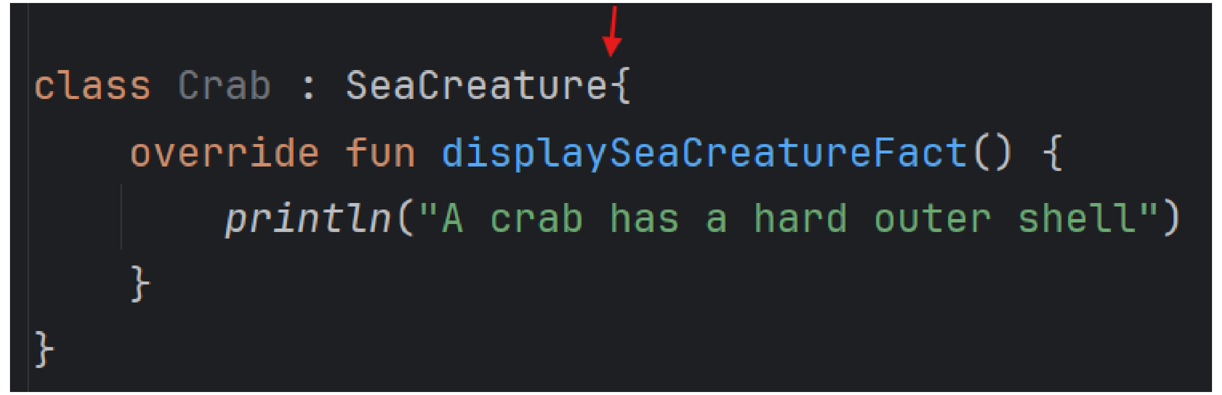


```
package com.bcit.lecture4

interface SeaCreature {
    fun displaySeaCreatureFact()
}
```

- 如果接口成员没有实现，则会自动推断为 abstract 关键字

Interfaces (cont.)

- Here, our Crab class is implementing the *SeaCreature* interface

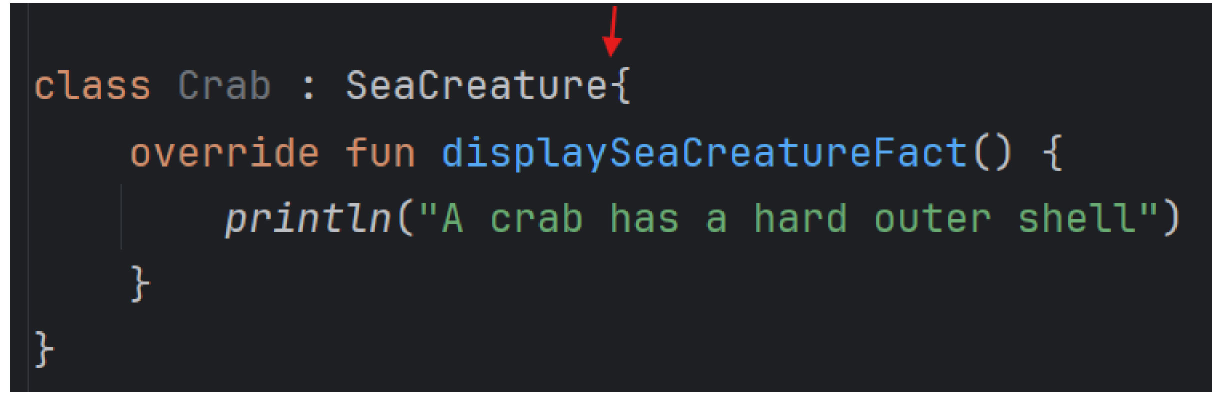


```
class Crab : SeaCreature{
    override fun displaySeaCreatureFact() {
        println("A crab has a hard outer shell")
    }
}
```

- Since interfaces don't have constructors, we don't use the ( ) brackets

接口（续）

- 这里，我们的 Crab 类正在实现 *SeaCreature* 接口

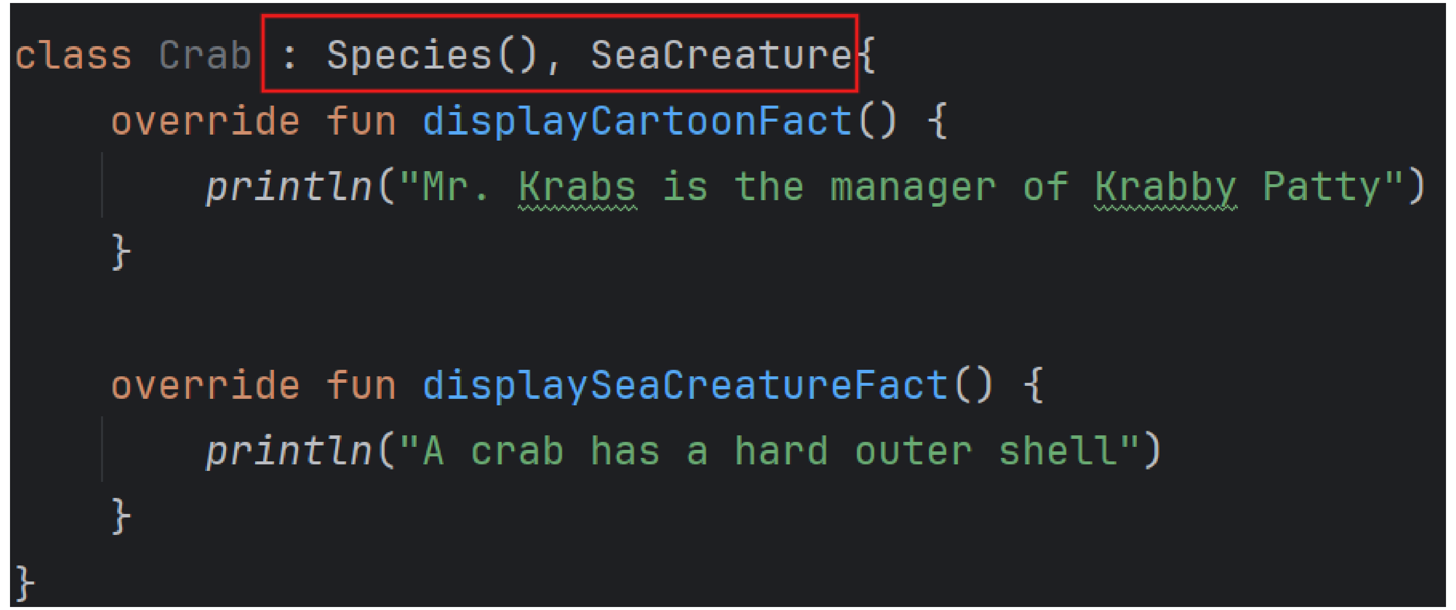


```
class Crab : SeaCreature{
    override fun displaySeaCreatureFact() {
        println("A crab has a hard outer shell")
    }
}
```

- 由于接口没有构造函数，我们 不使用 ( ) 括号

Interfaces (cont.)

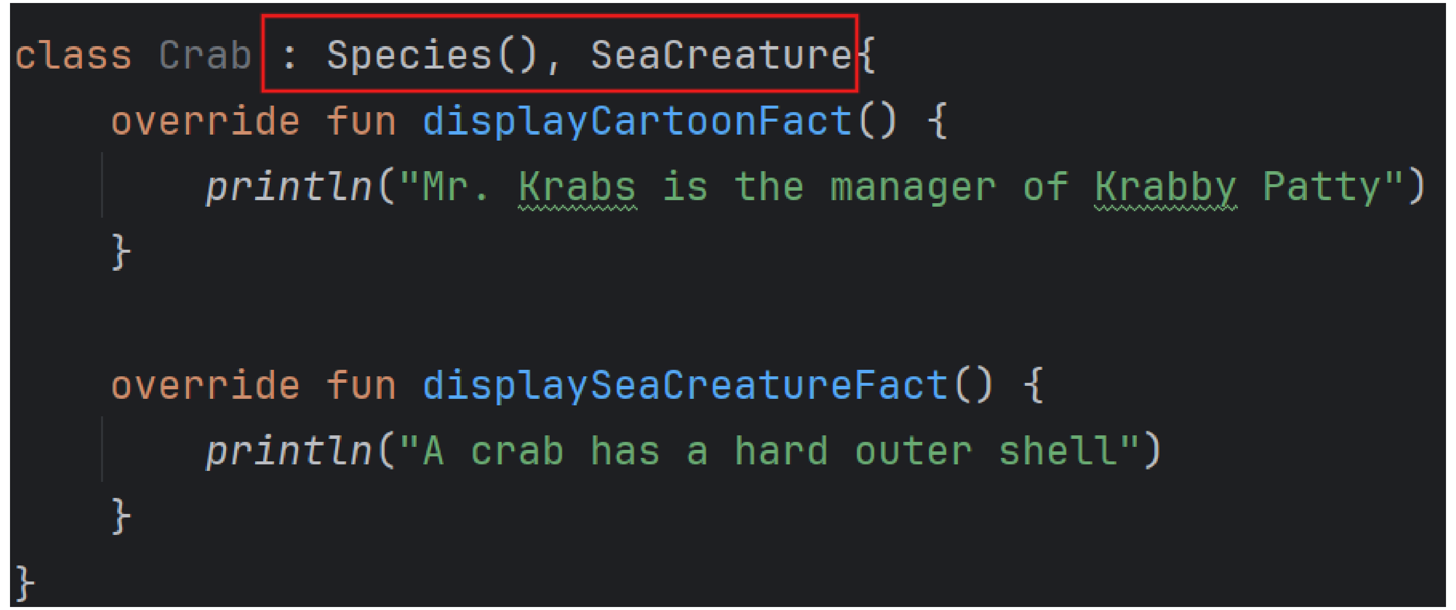
- When inheriting a class and/or implementing interfaces we separate them with commas



```
class Crab : Species(), SeaCreature{
    override fun displayCartoonFact() {
        println("Mr. Krabs is the manager of Krabby Patty")
    }

    override fun displaySeaCreatureFact() {
        println("A crab has a hard outer shell")
    }
}
```

接口（续）

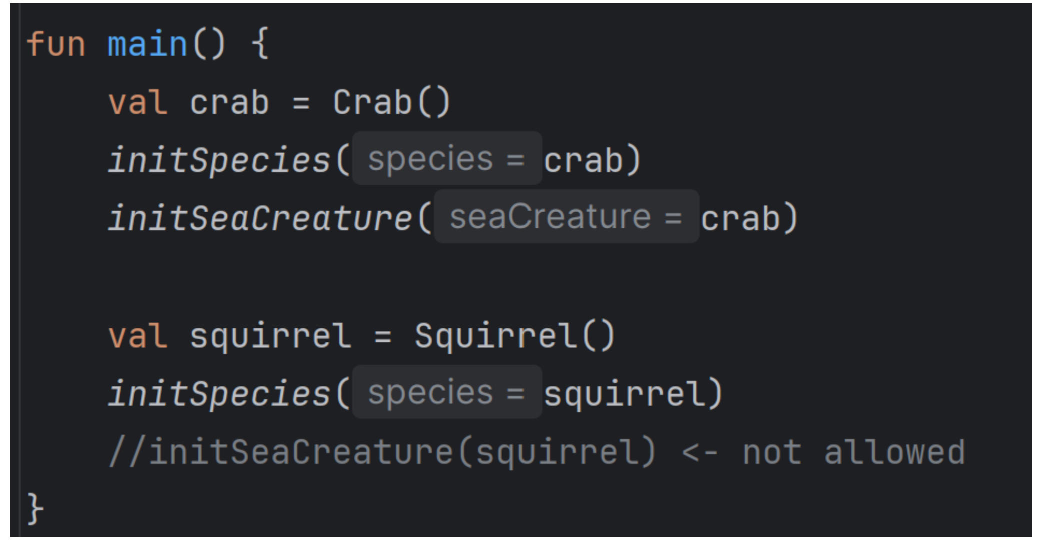
- 在继承类和/或实现接口时，我们使用逗号分隔它们



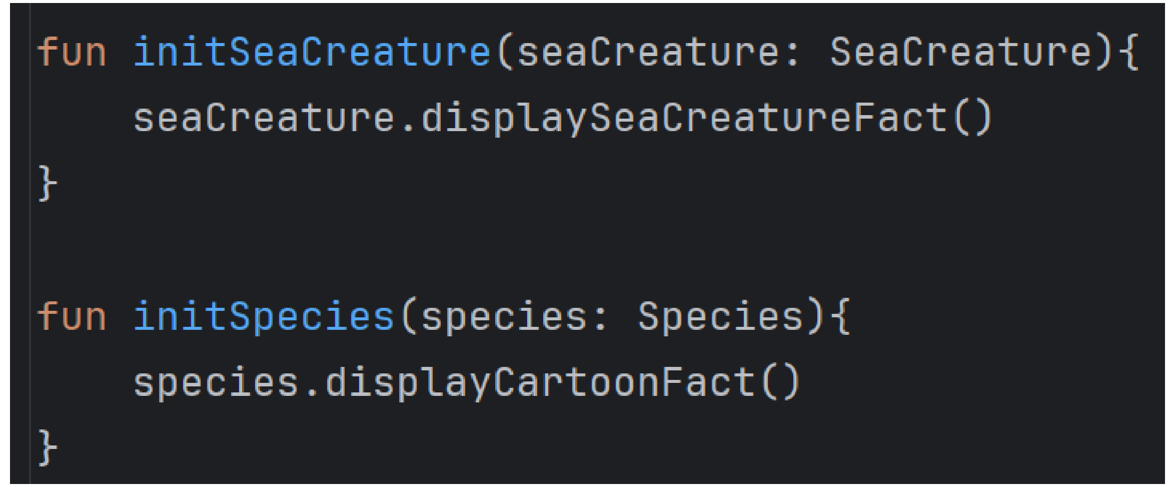
```
class Crab : Species(), SeaCreature{
    override fun displayCartoonFact() {
        println("Mr. Krabs is the manager of Krabby Patty")
    }

    override fun displaySeaCreatureFact() {
        println("A crab has a hard outer shell")
    }
}
```

Interfaces (cont.)

- We have now decoupled a SeaCreature from Species while maintaining abstraction



```
fun main() {
    val crab = Crab()
    initSpecies( species = crab)
    initSeaCreature( seaCreature = crab)

    val squirrel = Squirrel()
    initSpecies( species = squirrel)
    //initSeaCreature(squirrel) <- not allowed
}
```

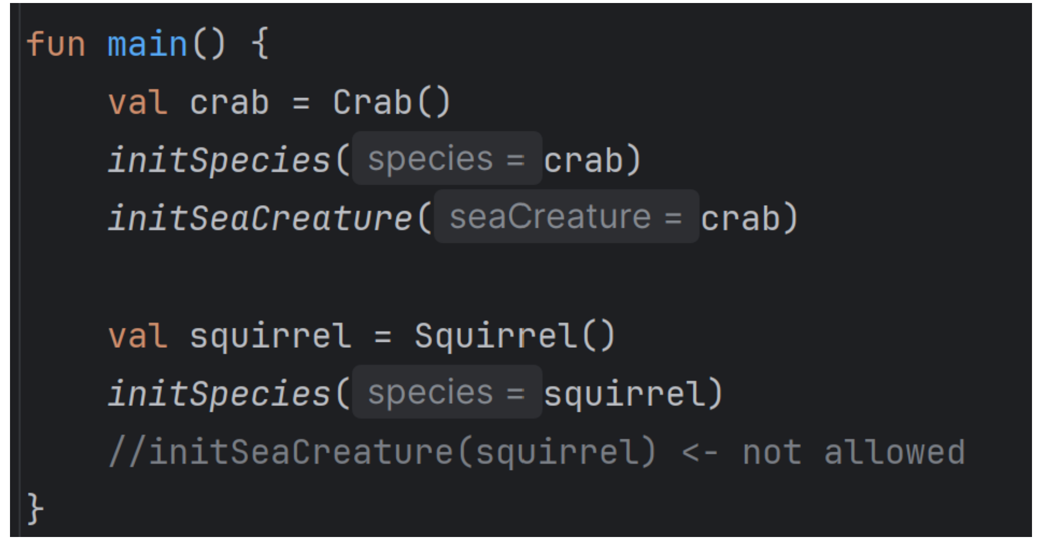


```
fun initSeaCreature(seaCreature: SeaCreature){
    seaCreature.displaySeaCreatureFact()
}

fun initSpecies(species: Species){
    species.displayCartoonFact()
}
```

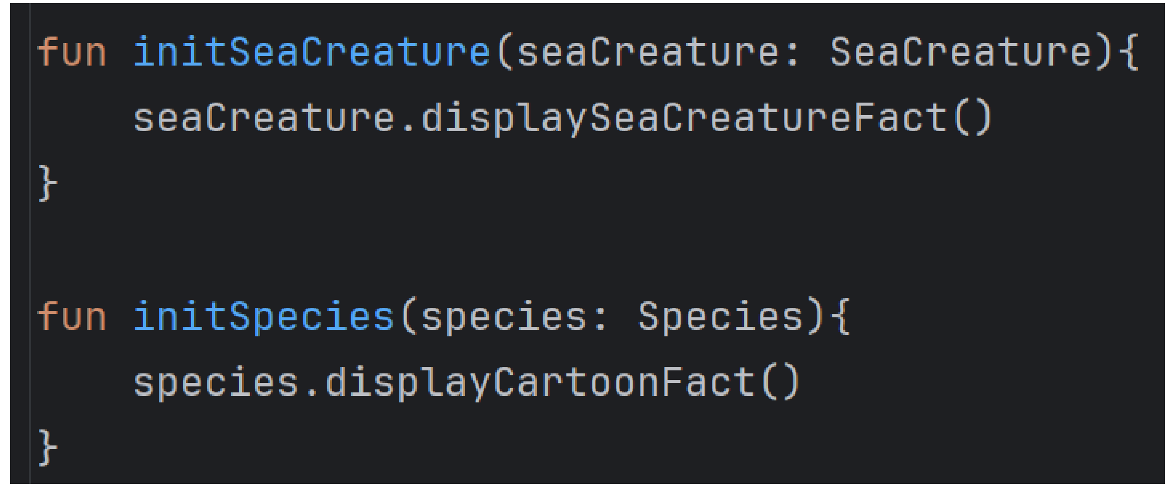
- Fun fact: this example illustrates the *Interface segregation principle*

接口（续）

- 我们现在已将 SeaCreature 与 Species 解耦，同时保持抽象性



```
fun main() {
    val crab = Crab()
    initSpecies( species = crab)
    initSeaCreature( seaCreature = crab)

    val squirrel = Squirrel()
    initSpecies( species = squirrel)
    //initSeaCreature(squirrel) <- not allowed
}
```

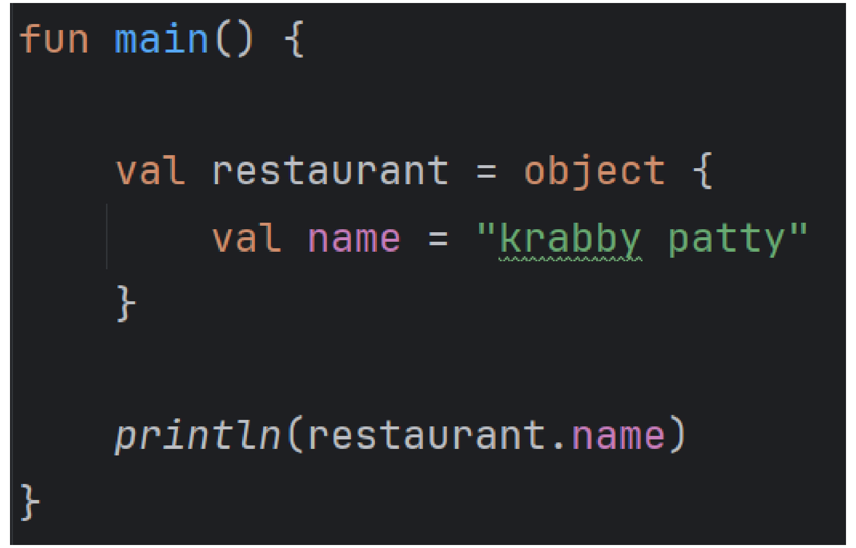


```
fun initSeaCreature(seaCreature: SeaCreature){
    seaCreature.displaySeaCreatureFact()
}

fun initSpecies(species: Species){
    species.displayCartoonFact()
}
```

- 有趣的是：此示例说明了 接口隔离原则

Anonymous Class

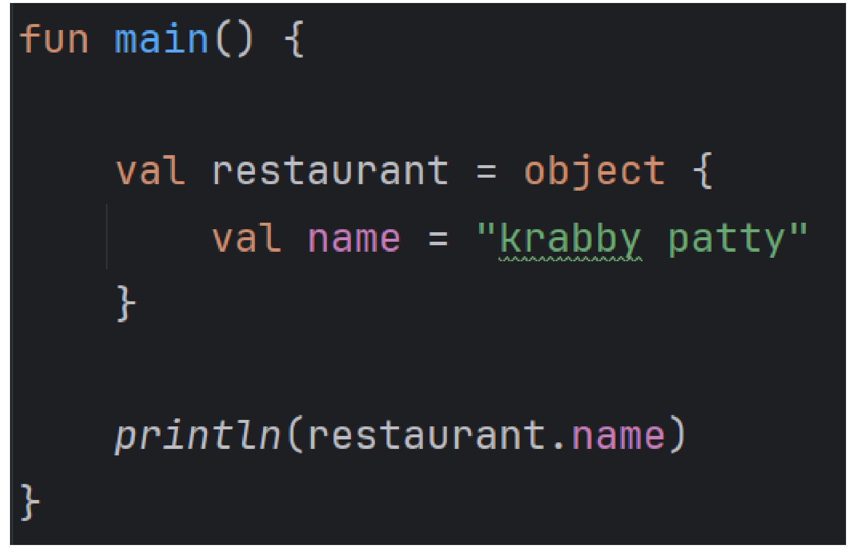
- Anonymous classes are declared using object expression
  - There is no class definition



```
fun main() {

    val restaurant = object {
        val name = "krabby patty"
    }

    println(restaurant.name)
}
```

匿名类

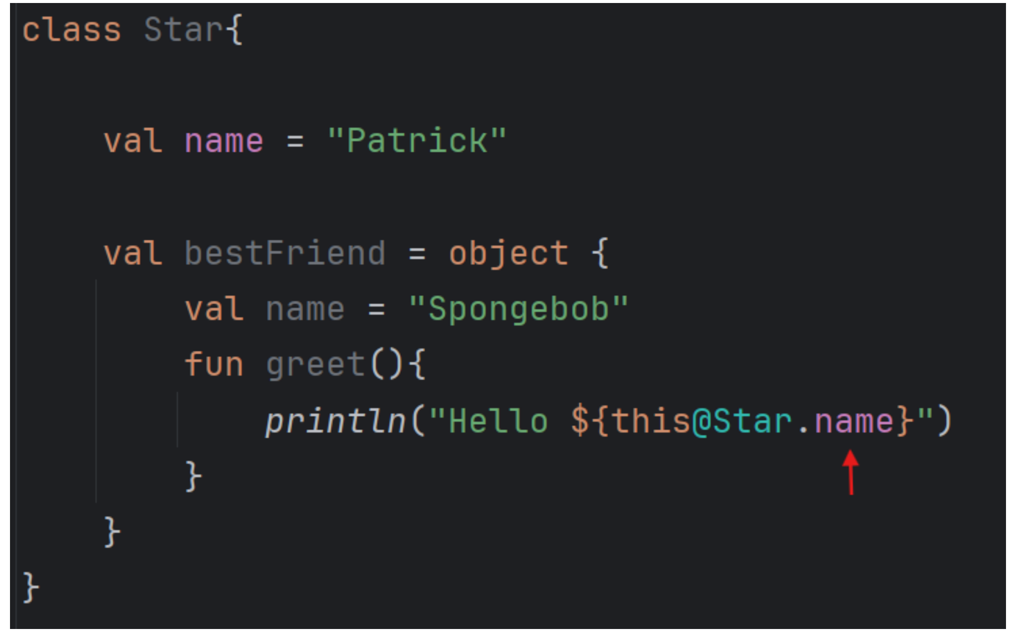
- 匿名类通过对象表达式声明
  - 没有类定义



```
fun main() {

    val restaurant = object {
        val name = "krabby patty"
    }

    println(restaurant.name)
}
```

Anonymous Class (cont.)

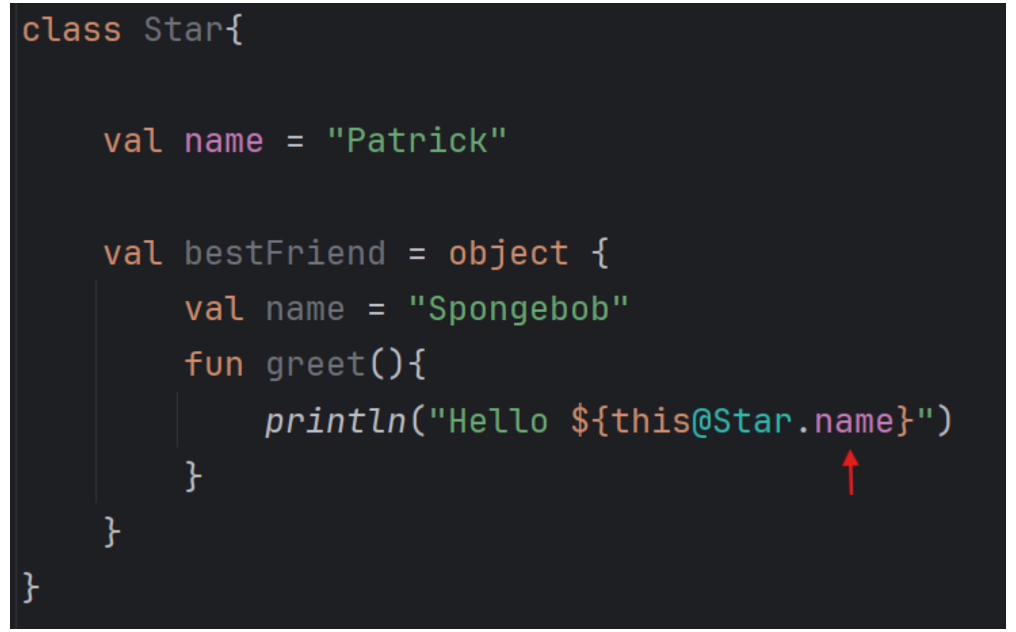
- By default, anonymous classes are *inner*
  - Classes marked or defined as *inner* can access outer class members



```
class Star{

    val name = "Patrick"

    val bestFriend = object {
        val name = "Spongebob"
        fun greet(){
            println("Hello ${this@Star.name}")
        }
    }
}
```

匿名类（续）

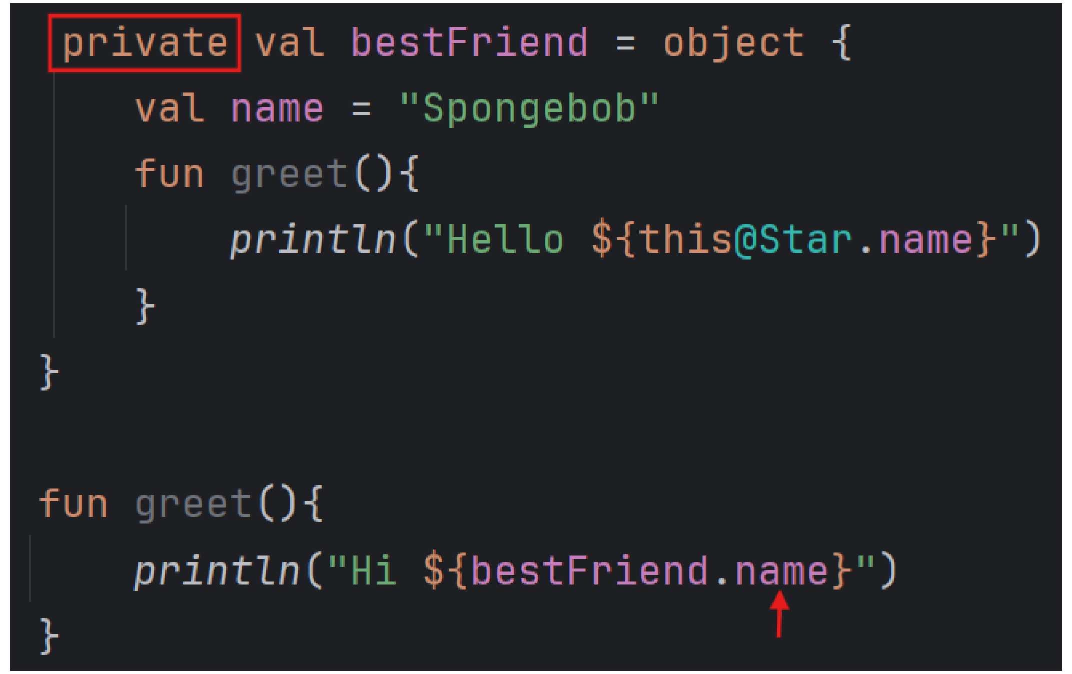
- 默认情况下，匿名类是内部的
  - 标记为或定义为内部的类可以访问外部类的成员



```
class Star{

    val name = "Patrick"

    val bestFriend = object {
        val name = "Spongebob"
        fun greet(){
            println("Hello ${this@Star.name}")
        }
    }
}
```

Anonymous Class (cont.)

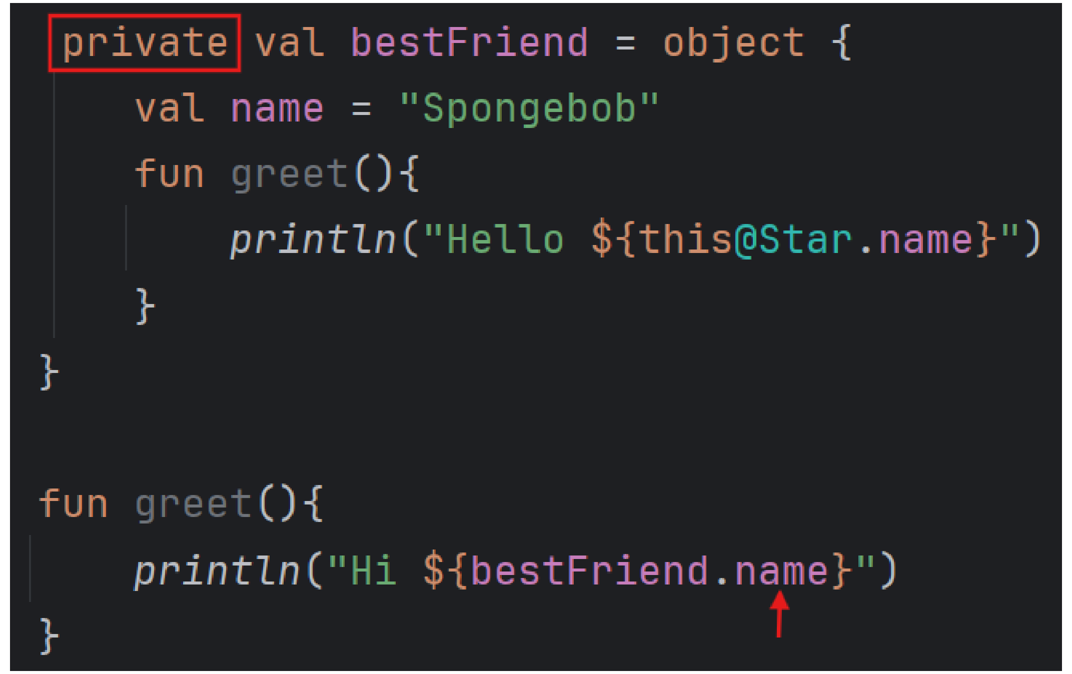
- When declaring an anonymous class as a class member (or at file level), it must be private for its full type to be preserved



```
private val bestFriend = object {
    val name = "Spongebob"
    fun greet(){
        println("Hello ${this@Star.name}")
    }
}

fun greet(){
    println("Hi ${bestFriend.name}")
}
```

匿名类（续）

- 将匿名类声明为类成员（或文件级别）时，必须将其设为私有，以确保其完整类型得以保留



```
private val bestFriend = object {
    val name = "Spongebob"
    fun greet(){
        println("Hello ${this@Star.name}")
    }
}

fun greet(){
    println("Hi ${bestFriend.name}")
}
```

Anonymous Class (cont.)

- Anonymous classes can also inherit from other classes and implement interfaces

- Let's look at a broader example using an interface and an anonymous class to illustrate how this would work

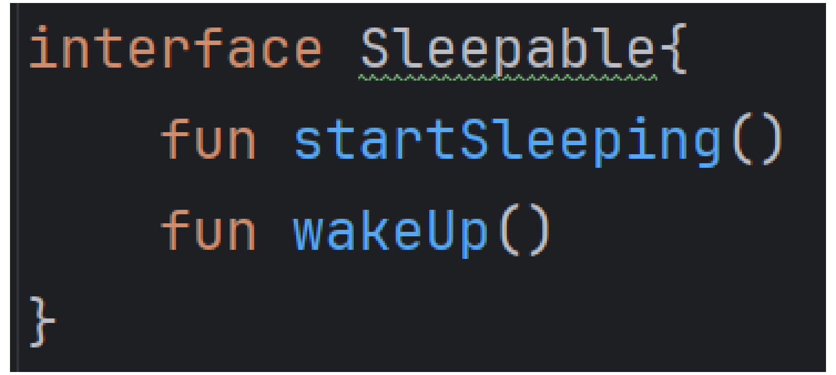
匿名类（续）

- 匿名类还可以继承其他类并实现接口

- 让我们来看一个更完整的例子，使用一个接口和一个匿名类来说明这将如何工作

Anonymous Class (cont.)

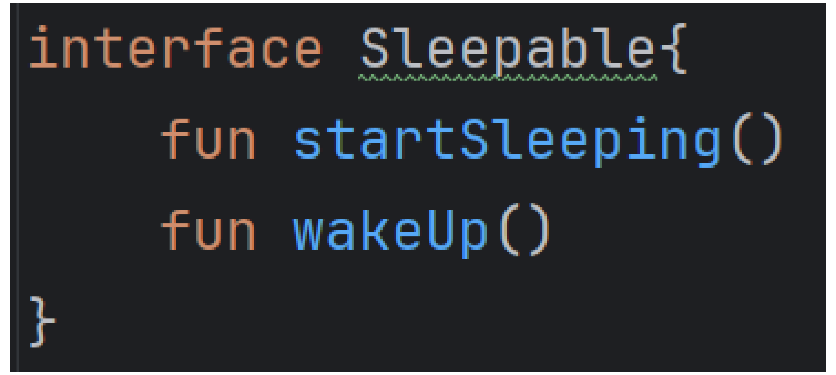
- First let's create a *Sleepable* interface



```
interface Sleepable{
    fun startSleeping()
    fun wakeUp()
}
```

匿名类（续）

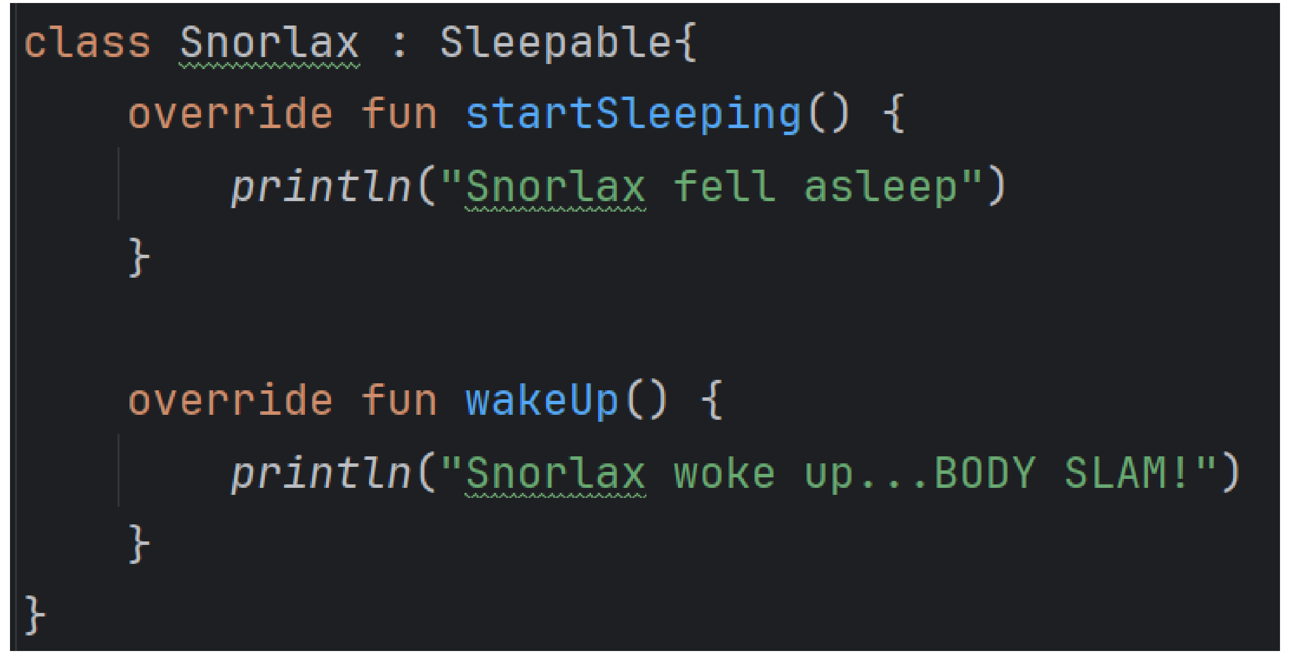
- 首先让我们创建一个 *Sleepable* 接口



```
interface Sleepable{
    fun startSleeping()
    fun wakeUp()
}
```

Anonymous Class (cont.)

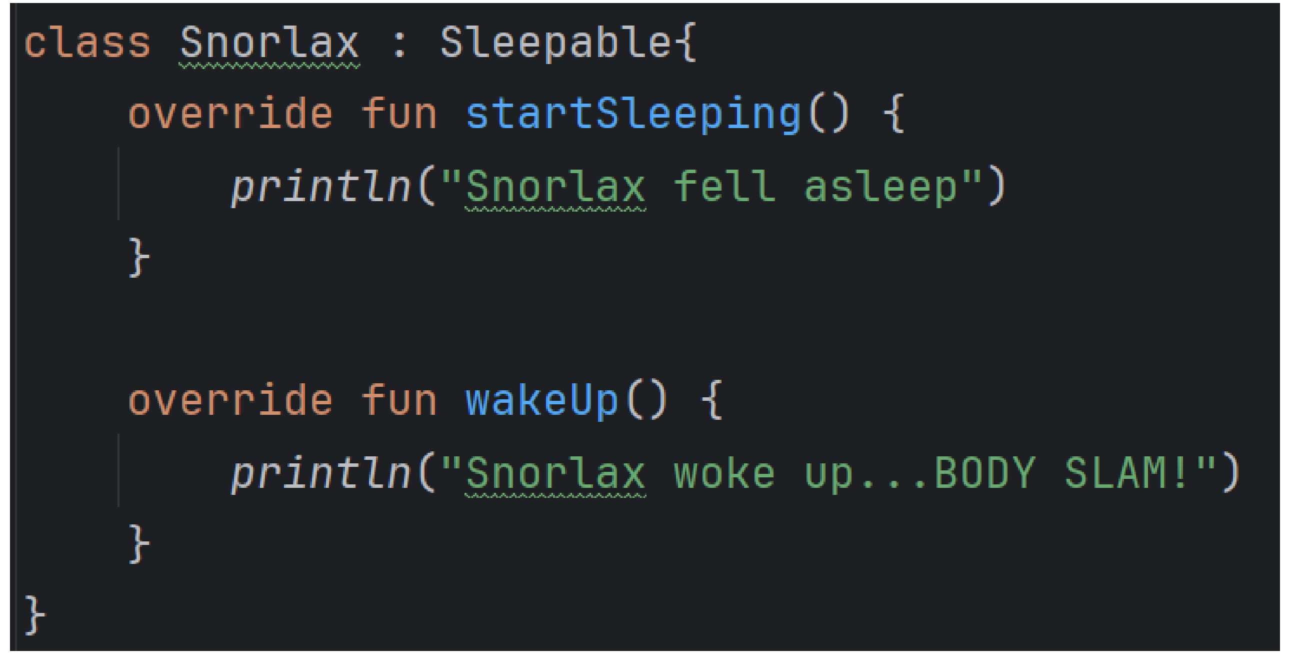
- Then let’s create a class that implements *Sleepable*



```
class Snorlax : Sleepable{
    override fun startSleeping() {
        println("Snorlax fell asleep")
    }

    override fun wakeUp() {
        println("Snorlax woke up...BODY SLAM!")
    }
}
```

匿名类（续）

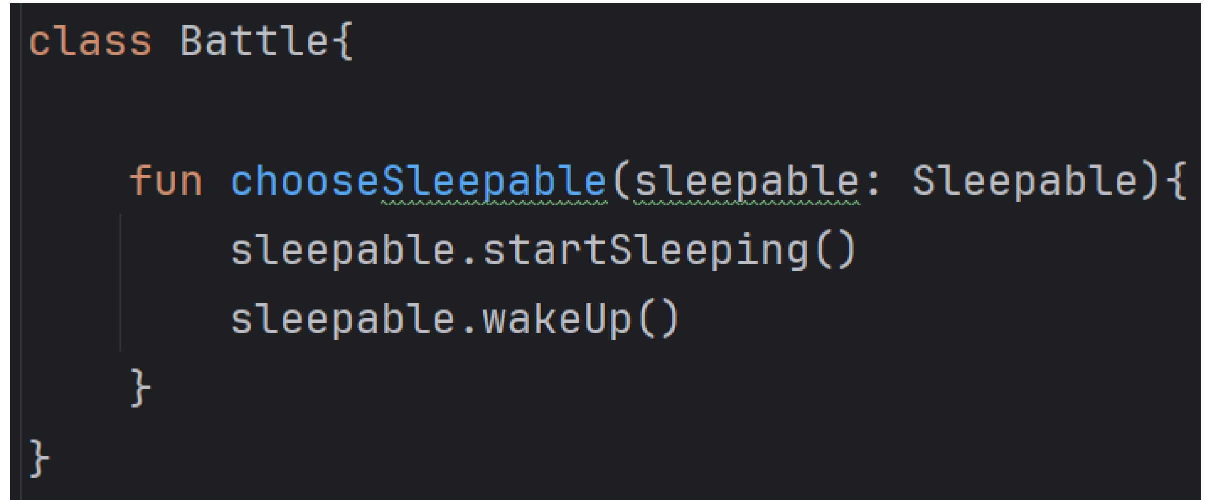
- 接下来让我们创建一个实现 *Sleepable* 的类



```
class Snorlax : Sleepable{
    override fun startSleeping() {
        println("Snorlax fell asleep")
    }

    override fun wakeUp() {
        println("Snorlax woke up...BODY SLAM!")
    }
}
```

Anonymous Class (cont.)

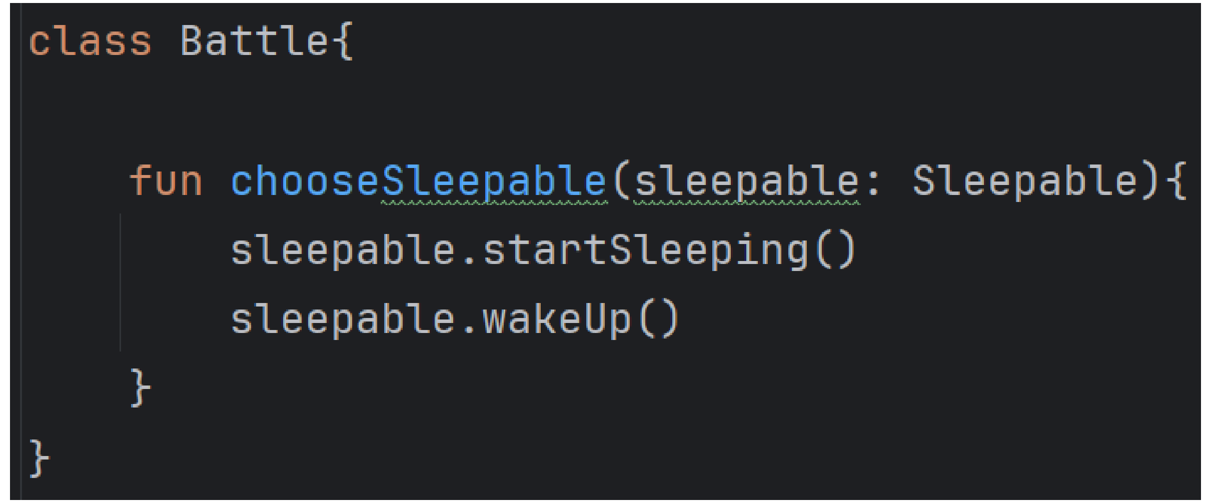
- Next, let’s create a class that uses a *Sleepable*



```
class Battle{

    fun chooseSleepable(sleepable: Sleepable){
        sleepable.startSleeping()
        sleepable.wakeUp()
    }
}
```

匿名类（续）

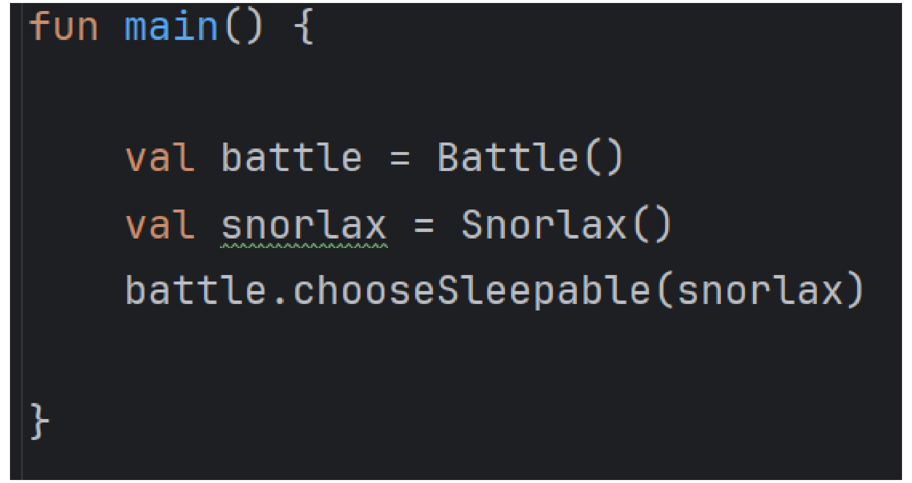
- 接下来，让我们创建一个使用*Sleepable*的类



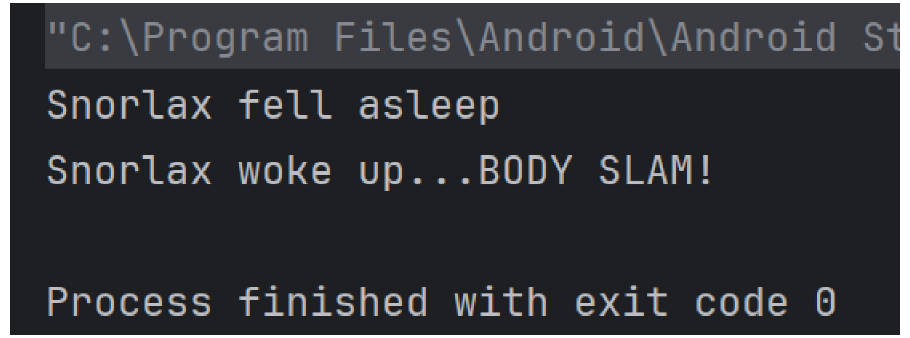
```
class Battle{

    fun chooseSleepable(sleepable: Sleepable){
        sleepable.startSleeping()
        sleepable.wakeUp()
    }
}
```

Anonymous Class (cont.)

- Now we can use it all together



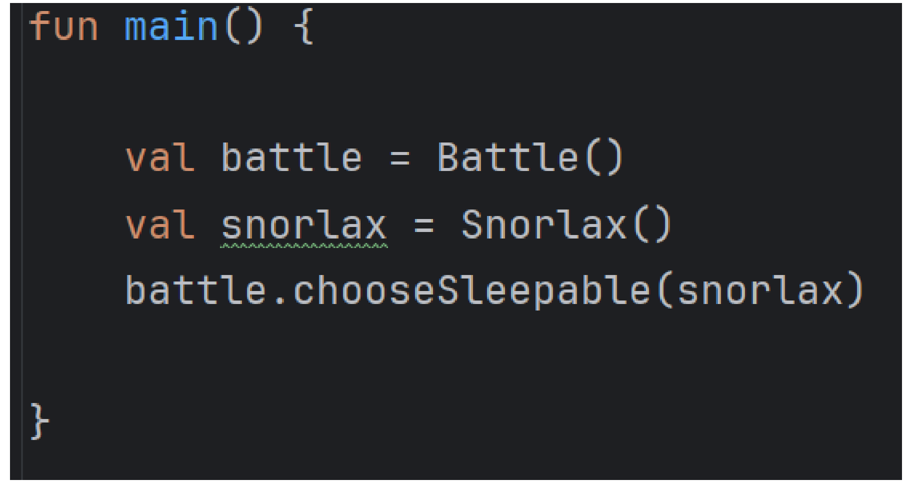
```
fun main() {

    val battle = Battle()
    val snorlax = Snorlax()
    battle.chooseSleepable(snorlax)

}
```



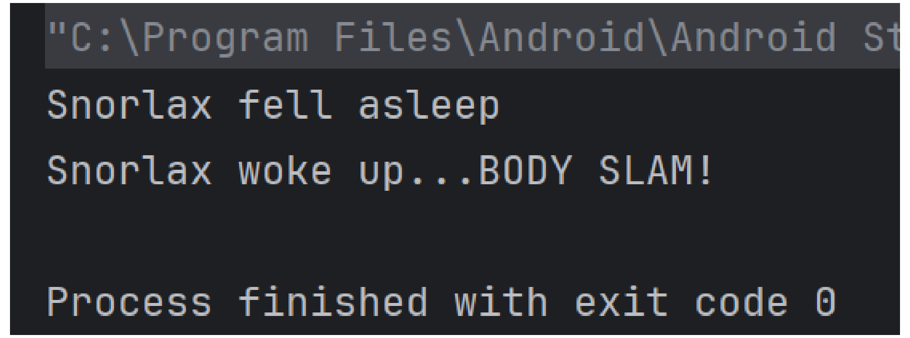
```
"C:\Program Files\Android\Android St
Snorlax fell asleep
Snorlax woke up...BODY SLAM!

Process finished with exit code 0
```

匿名类（续）

- 现在我们可以将其全部组合使用



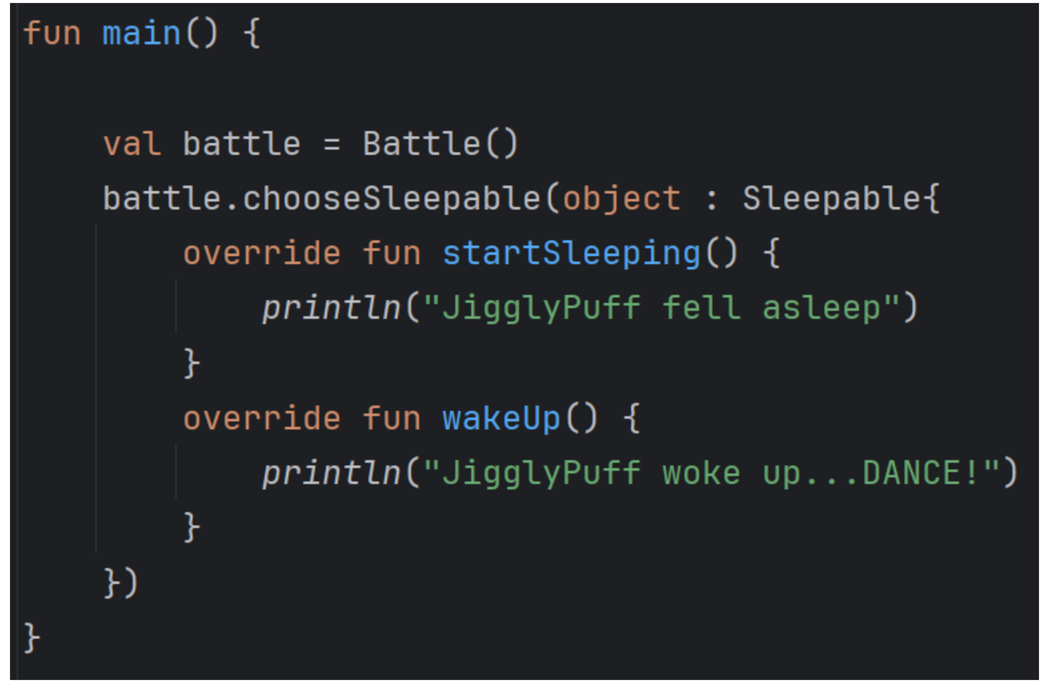
```
fun main() {

    val battle = Battle()
    val snorlax = Snorlax()
    battle.chooseSleepable(snorlax)

}
```



```
"C:\Program Files\Android\Android St
Snorlax fell asleep
Snorlax woke up...BODY SLAM!

Process finished with exit code 0
```

Anonymous Class (cont.)

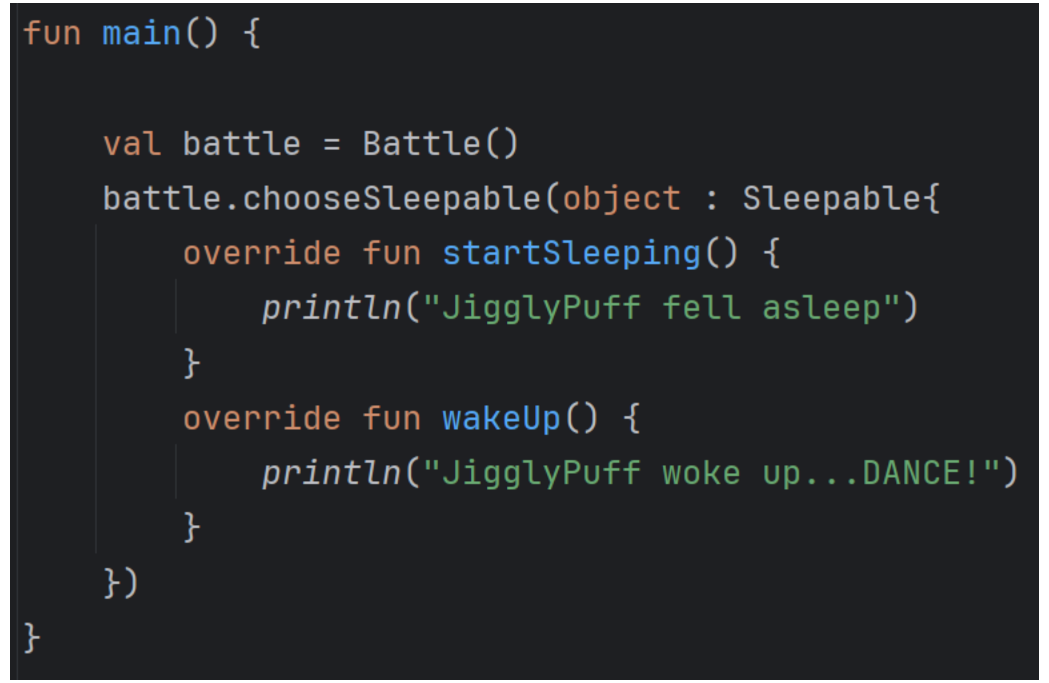
- An anonymous class comes in handy if we want to create a simple class quickly



```kotlin
fun main() {

    val battle = Battle()
    battle.chooseSleepable(object : Sleepable{
        override fun startSleeping() {
            println("JigglyPuff fell asleep")
        }
        override fun wakeUp() {
            println("JigglyPuff woke up...DANCE!")
        }
    })
}
```

匿名类（续）

- 如果我们想快速创建一个简单的类，匿名类会非常方便



```kotlin
fun main() {

    val battle = Battle()
    battle.chooseSleepable(object : Sleepable{
        override fun startSleeping() {
            println("JigglyPuff fell asleep")
        }
        override fun wakeUp() {
            println("JigglyPuff woke up...DANCE!")
        }
    })
}
```