Lecture 4

COMP 3717- Mobile Dev with Android Tech

Classes & objects

- Like other OOP languages, kotlin uses classes and objects

```
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 = ""
}
```

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)")
    }
}
```

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
```

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

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
```

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
    }
```

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()
```

Constructors (cont.)

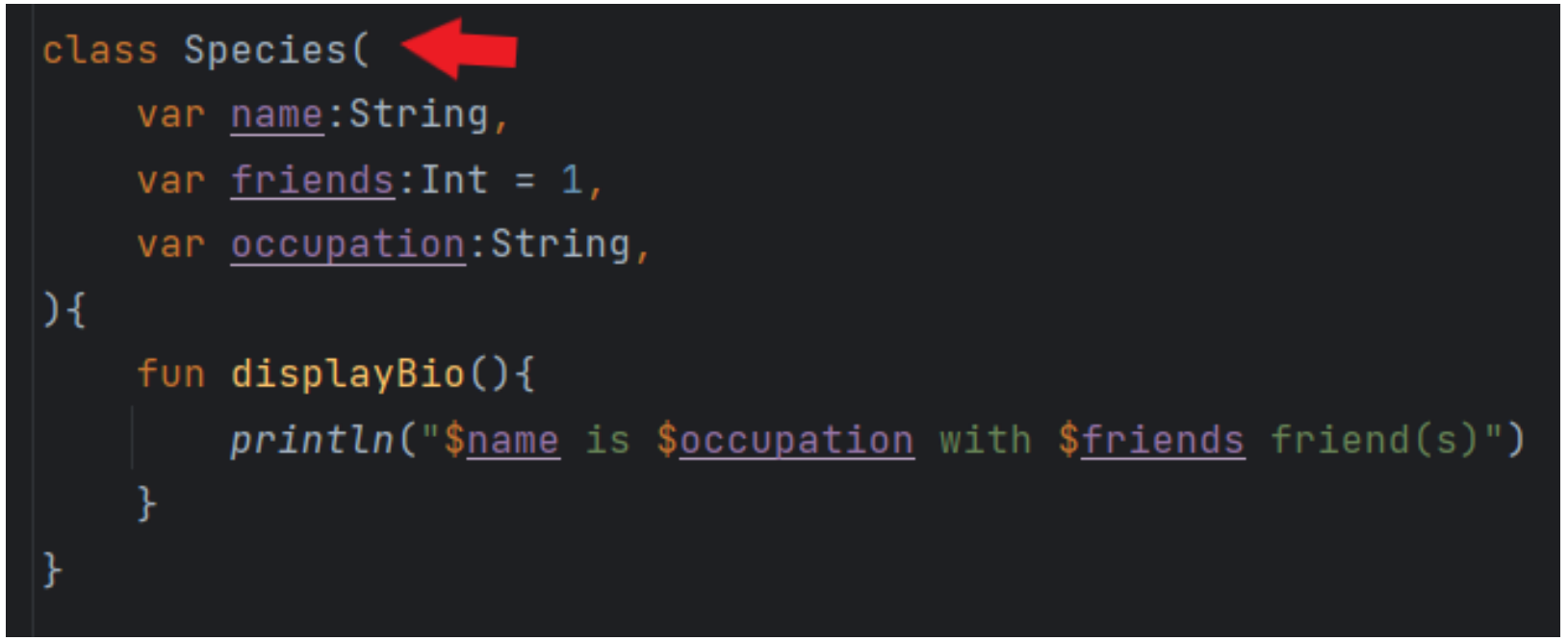
- We can also create properties as parameters in a constructor
  - The difference to regular parameters is declaring them with val or var

```
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

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)")
    }
}
```

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")
```

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)
```

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
```

Getters & setters

- Getters and setters are auto generated and hidden

```
class Species{

    var name: String = ""

        get() = field

        set(value) {
            field = value
        }
```

```
get() = field

    Redundant getter
se
    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)

}
```

Getters & setters (cont.)

- Here we are encapsulating the validation of our data

```
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"
```

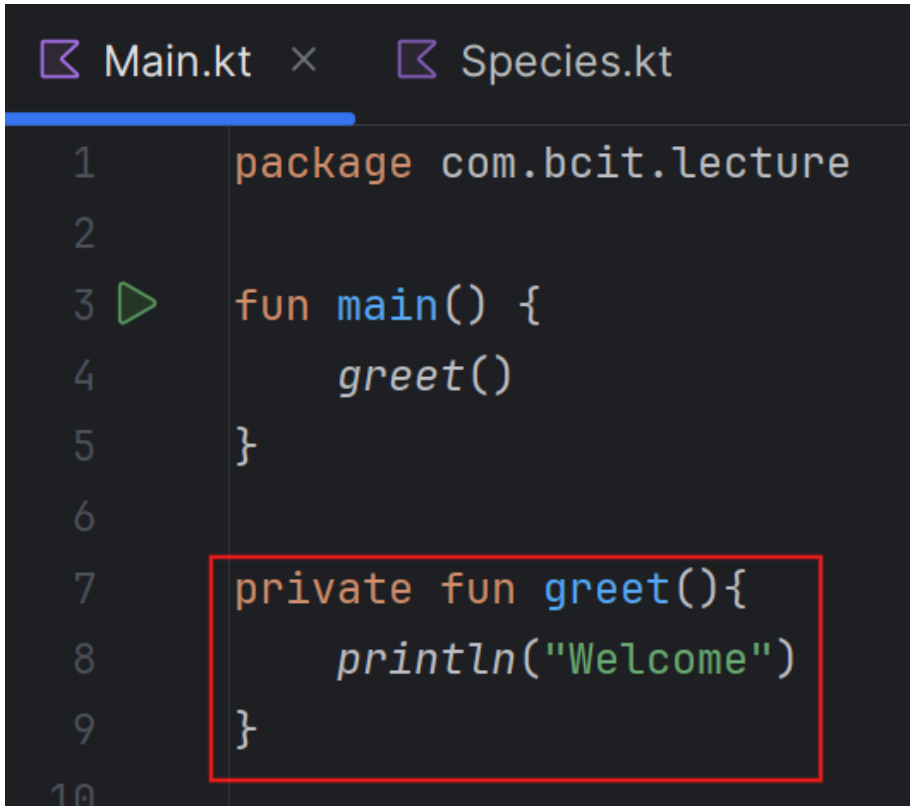
Cannot assign to 'job': the setter is private in 'Species'

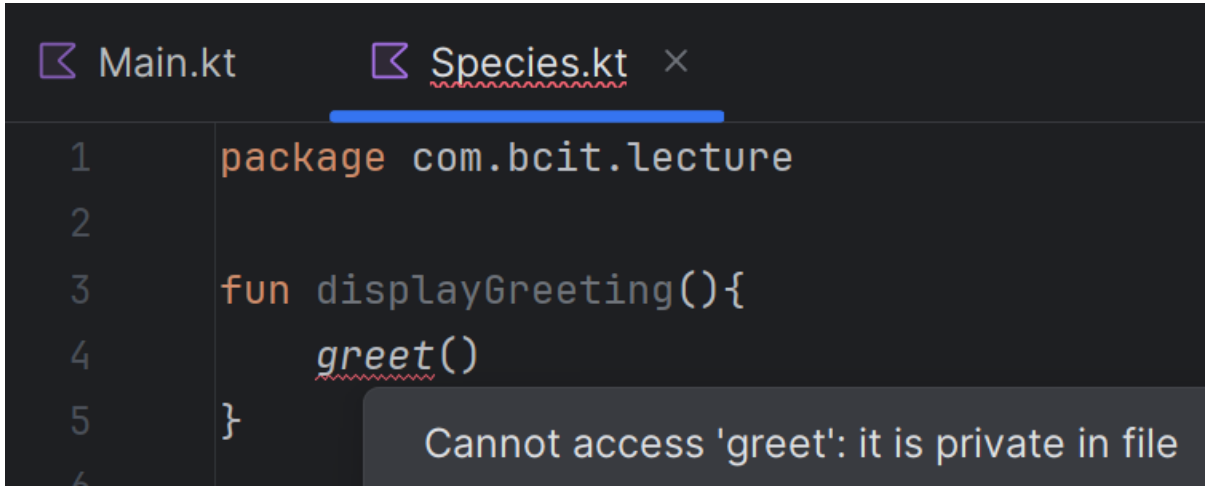
Visibility modifiers

- 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

Visibility modifiers (cont.)

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

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*

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")
    }
}
```

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()
}
```

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")
    }
}
```

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")
    }
}
```

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")
    }
}
```

Private vs protected (cont.)

Main.kt × Species.kt

```
1  package com.example.lecture4
2
3  fun main() {
4      val sponge = Sponge( name = "SpongeBob",  color = "Yellow")
5      //sponge.name = "bob" <- not allowed
6      sponge.displaySpongeBio()
7  }
8
```

```
"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)

Abstract classes (cont.)

- We cannot create an instance of an abstract class

```
abstract class Pokemon() {}

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

Cannot create an instance

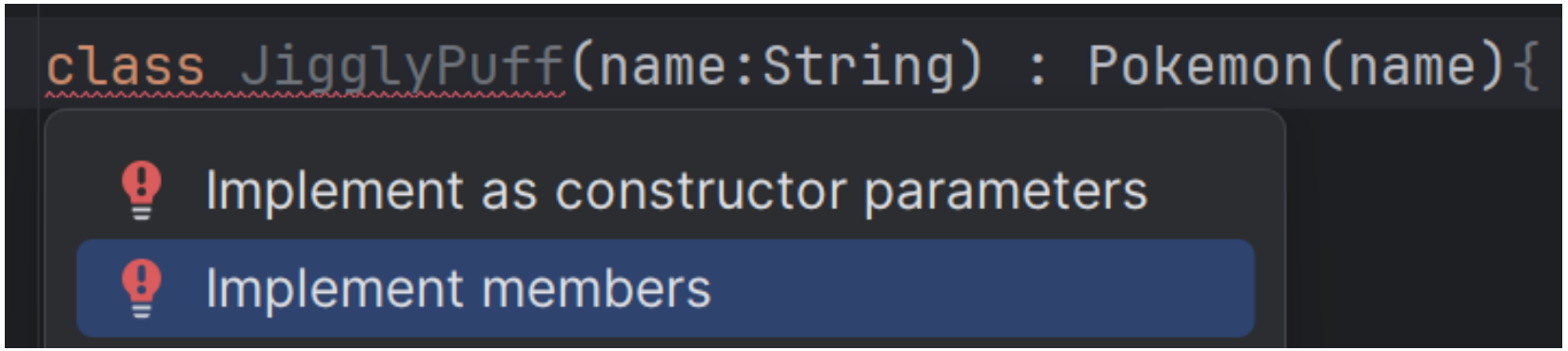
Abstract classes (cont.)

- 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!")
    }
}
```

Abstract classes (cont.)

- When we inherit our abstract class, we must implement the abstract 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")
    }
}
```

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")
    }
}
```

Overriding members (cont.)

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

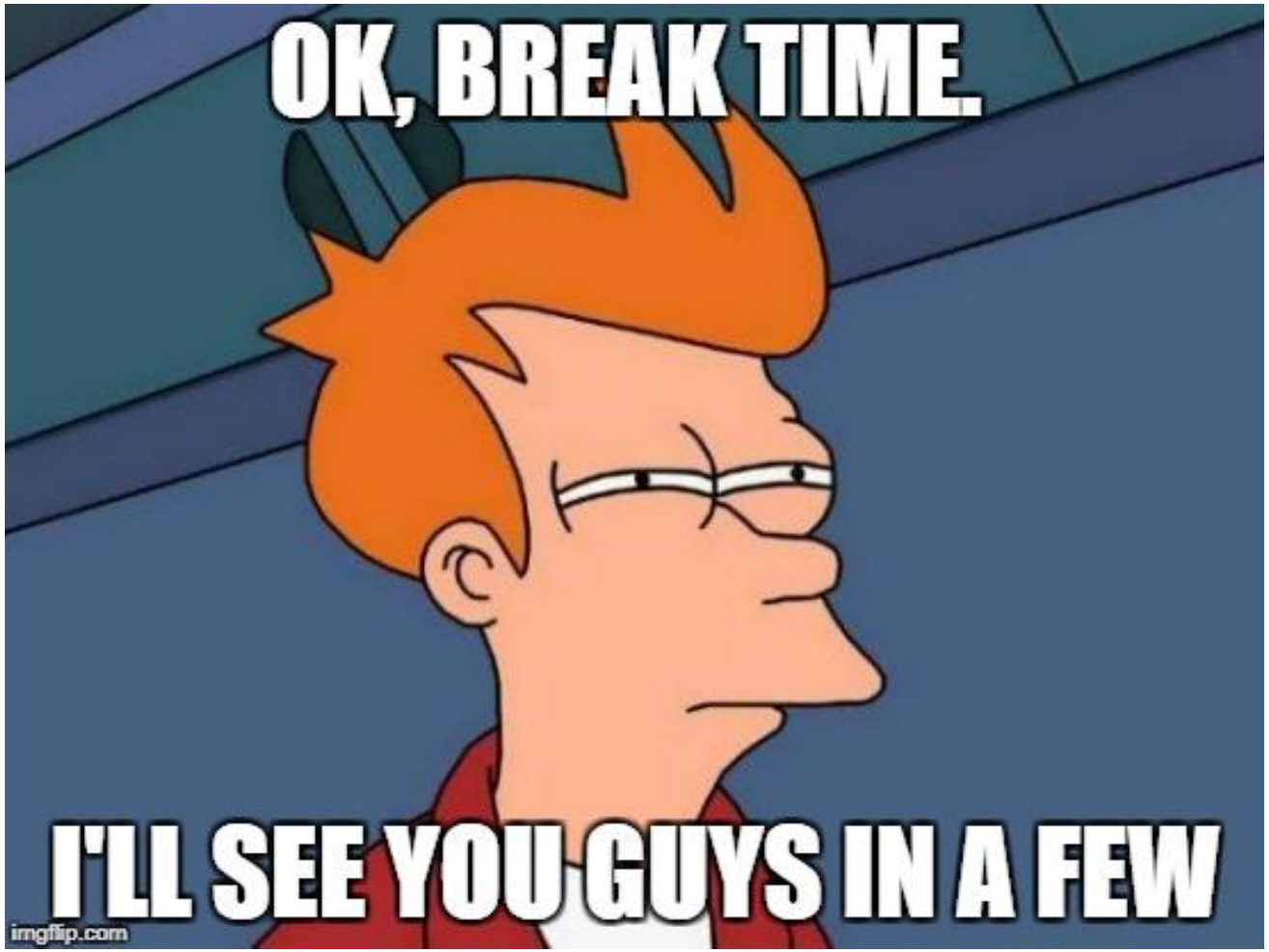
```
abstract class Pokemon(val name:String) {
    abstract val color:String
    abstract fun displayInfo()
    open fun attack(){
        println("Tail whip!")
    }
}
```

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

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

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)
}
```


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

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

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

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

Interfaces (cont.)

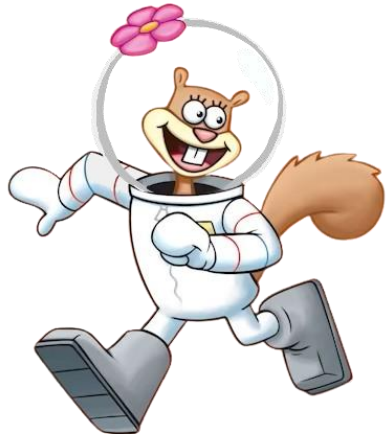
- We could create an abstract Cartoon class and define some abstract members

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

Interfaces (cont.)

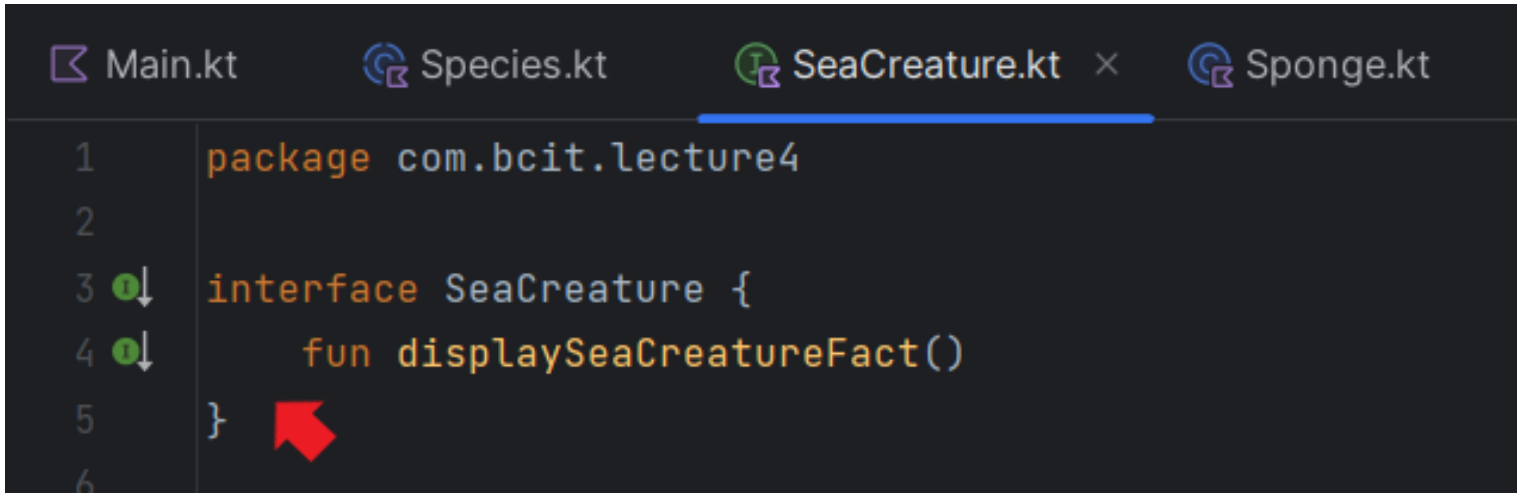
- The problem though is some species in the show aren't actual sea creatures which adds unnecessary complexity

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

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



Interfaces (cont.)

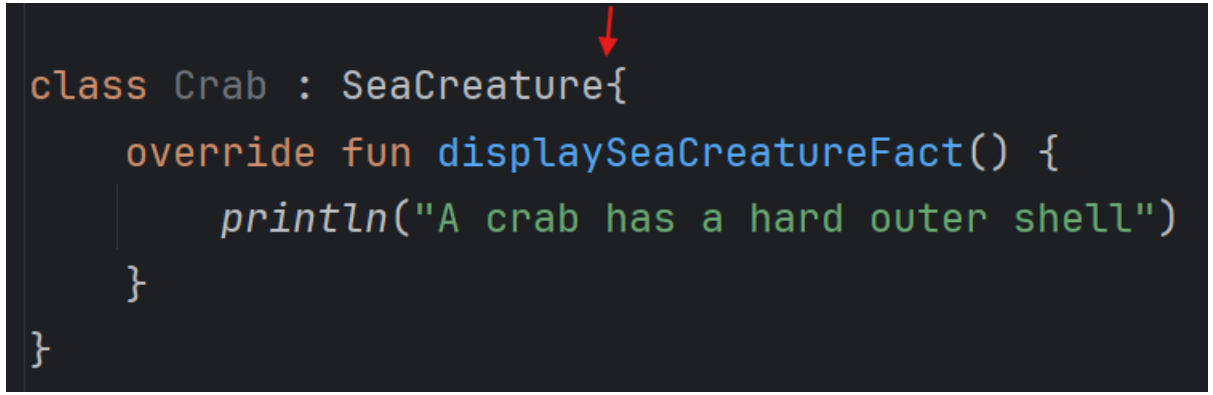
- To solve this problem and maintain abstraction we can use an interface



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

Interfaces (cont.)

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



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

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")
    }
}
```

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*

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)
}
```

Anonymous Class (cont.)

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

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

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

Anonymous Class (cont.)

- First let's create a *Sleepable* interface

```
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!")
    }
}
```

Anonymous Class (cont.)

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

```
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
```

Anonymous Class (cont.)

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

```
fun main() {

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