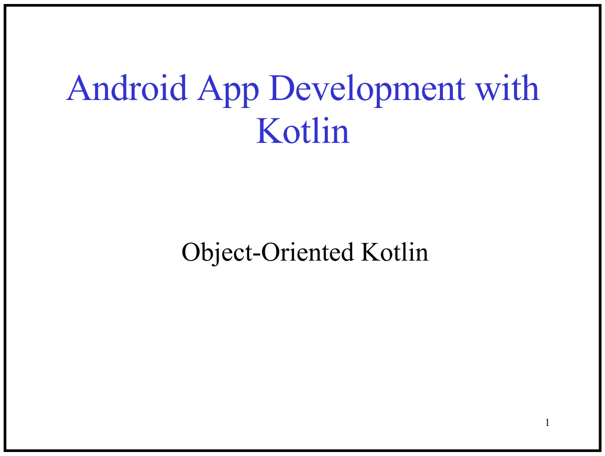
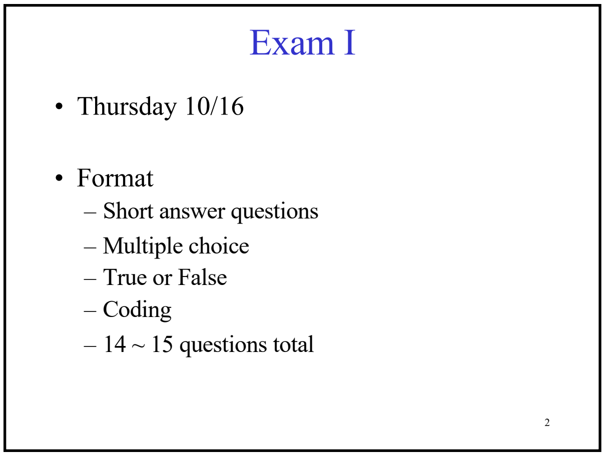


Android App Development with Kotlin

Object-Oriented Kotlin

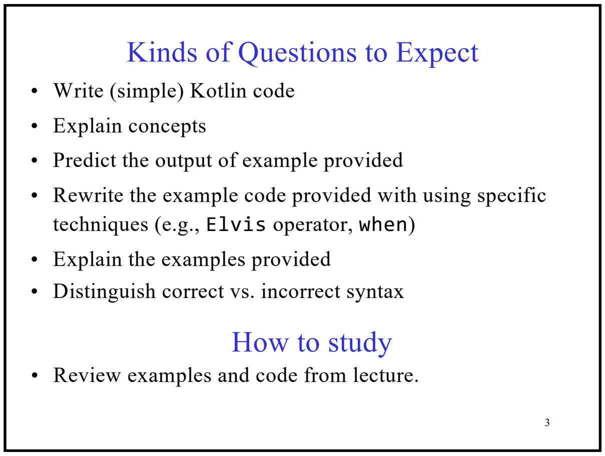
1



Exam I

- Thursday 10/16
- Format
  - Short answer questions
  - Multiple choice
  - True or False
  - Coding
  - 14 ~ 15 questions total

2



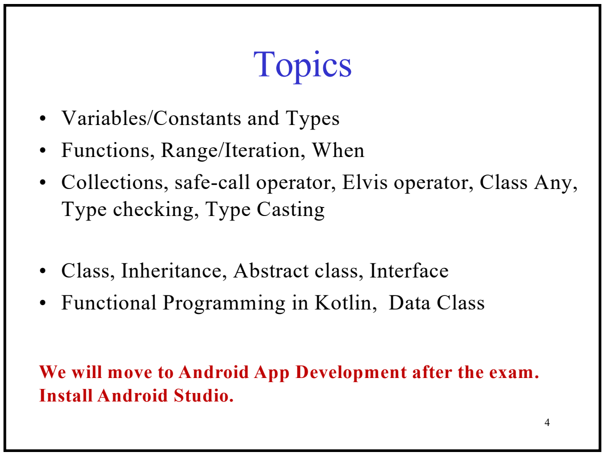
Kinds of Questions to Expect

- Write (simple) Kotlin code
- Explain concepts
- Predict the output of example provided
- Rewrite the example code provided with using specific techniques (e.g., `Elvis` operator, `when`)
- Explain the examples provided
- Distinguish correct vs. incorrect syntax

How to study

- Review examples and code from lecture.

3



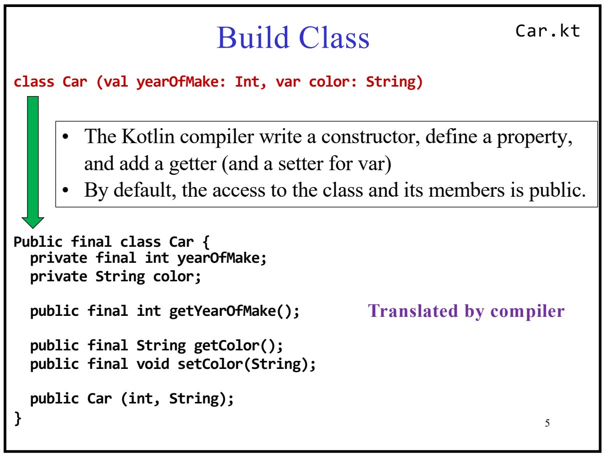
Topics

- Variables/Constants and Types
- Functions, Range/Iteration, When
- Collections, safe-call operator, Elvis operator, Class Any, Type checking, Type Casting

- Class, Inheritance, Abstract class, Interface
- Functional Programming in Kotlin, Data Class

We will move to Android App Development after the exam. Install Android Studio.

4



Build Class

Car.kt

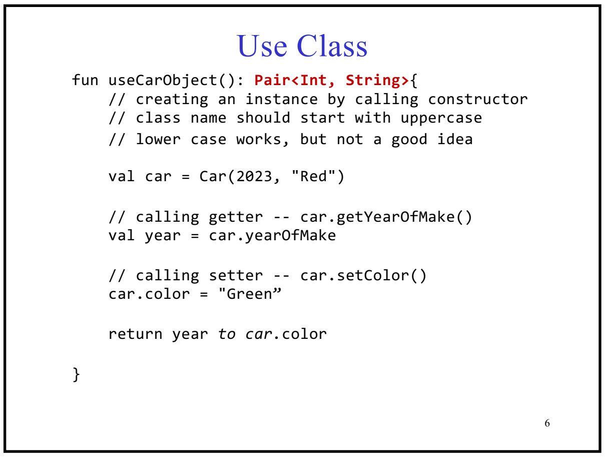
```
class Car (val yearOfMake: Int, var color: String)
```

- The Kotlin compiler write a constructor, define a property, and add a getter (and a setter for var)
- By default, the access to the class and its members is public.

```
Public final class Car {
  private final int yearOfMake;
  private String color;

  public final int getYearOfMake();

  public final String getColor();
  public final void setColor(String);

  public Car (int, String);
}
```

Translated by compiler

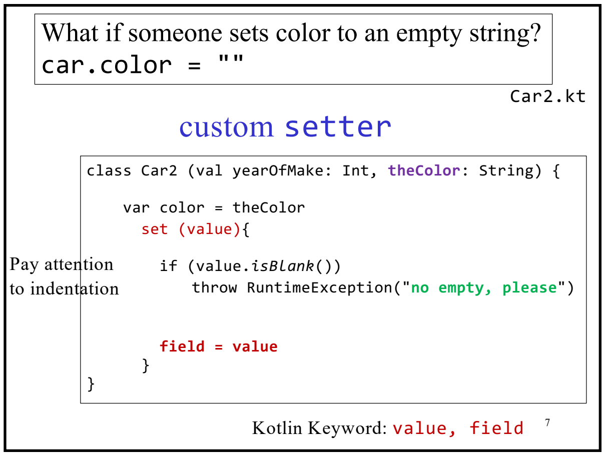
5



Use Class

```
fun useCarObject(): Pair<Int, String>{
    // creating an instance by calling constructor
    // class name should start with uppercase
    // lower case works, but not a good idea

    val car = Car(2023, "Red")

    // calling getter -- car.getYearOfMake()
    val year = car.yearOfMake

    // calling setter -- car.setColor()
    car.color = "Green”

    return year to car.color

}
```

6



What if someone sets color to an empty string?
`car.color = ""`

Car2.kt

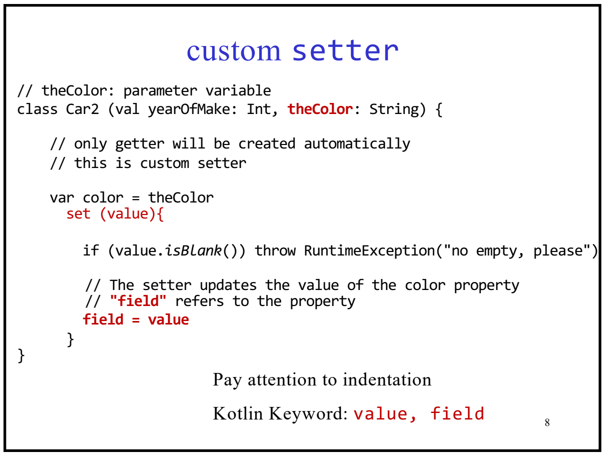
custom setter

```
class Car2 (val yearOfMake: Int, theColor: String) {

    var color = theColor
      set (value){

        if (value.isBlank())
            throw RuntimeException("no empty, please")


        field = value
      }
}
```

Pay attention to indentation

Kotlin Keyword: `value, field`

7



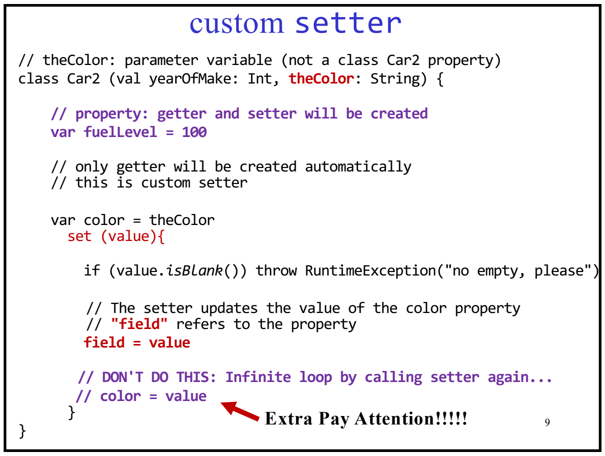
custom setter

```
// theColor: parameter variable
class Car2 (val yearOfMake: Int, theColor: String) {

    // only getter will be created automatically
    // this is custom setter

    var color = theColor
      set (value){

        if (value.isBlank()) throw RuntimeException("no empty, please")

        // The setter updates the value of the color property
        // "field" refers to the property
        field = value
      }
}
```

Pay attention to indentation

Kotlin Keyword: `value, field`

8

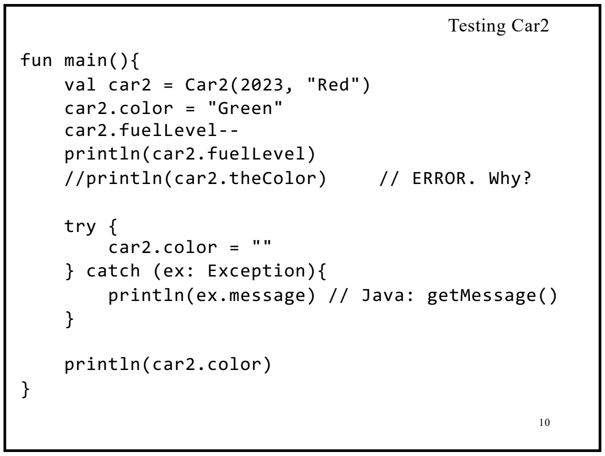


custom setter

```
// theColor: parameter variable (not a class Car2 property)
class Car2 (val yearOfMake: Int, theColor: String) {

    // property: getter and setter will be created
    var fuelLevel = 100

    // only getter will be created automatically
    // this is custom setter

    var color = theColor
      set (value){

        if (value.isBlank()) throw RuntimeException("no empty, please")

        // The setter updates the value of the color property
        // "field" refers to the property
        field = value

       // DON'T DO THIS: Infinite loop by calling setter again...
       // color = value
      }
}
```

Extra Pay Attention!!!!!

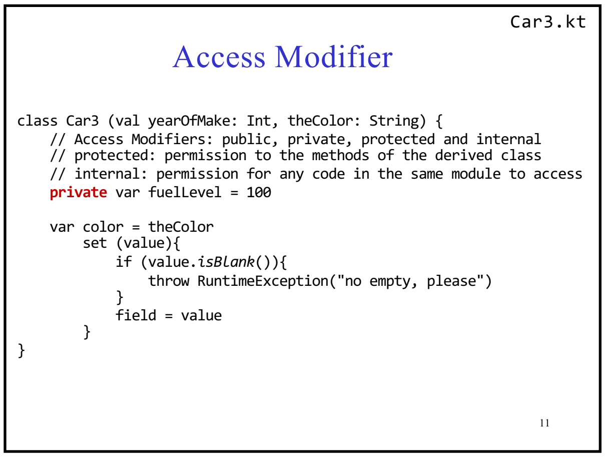
9



Testing Car2

```
fun main(){
    val car2 = Car2(2023, "Red")
    car2.color = "Green"
    car2.fuelLevel--
    println(car2.fuelLevel)
    //println(car2.theColor)     // ERROR. Why?

    try {
        car2.color = ""
    } catch (ex: Exception){
        println(ex.message) // Java: getMessage()
    }

    println(car2.color)
}
```

10

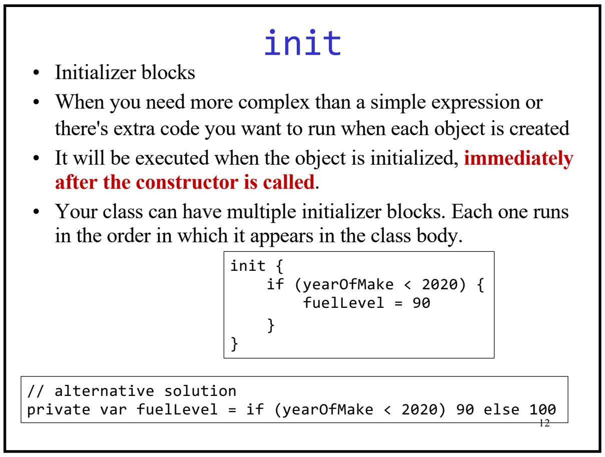


Car3.kt

Access Modifier

```
class Car3 (val yearOfMake: Int, theColor: String) {
    // Access Modifiers: public, private, protected and internal
    // protected: permission to the methods of the derived class
    // internal: permission for any code in the same module to access
    private var fuelLevel = 100

    var color = theColor
        set (value){
            if (value.isBlank()){
                throw RuntimeException("no empty, please")
            }
            field = value
        }
}
```

11



init

- Initializer blocks
- When you need more complex than a simple expression or there's extra code you want to run when each object is created
- It will be executed when the object is initialized, **immediately after the constructor is called**.
- Your class can have multiple initializer blocks. Each one runs in the order in which it appears in the class body.

```
init {
    if (yearOfMake < 2020) {
        fuelLevel = 90
    }
}
```

```
// alternative solution
private var fuelLevel = if (yearOfMake < 2020) 90 else 100
```

12

Person.kt

Secondary Constructor

- You may create more constructors, called secondary constructors.
- Your secondary constructors are required to either call the primary constructor or call one of the other secondary constructors.
- Secondary constructors' parameters can **NOT** be decorated with `val` or `var`
  - They don't define any properties.
  - Only the primary constructor and declarations within the class may define properties

13

Empty Constructor

- Whenever you define a class with no parameter, the compiler adds a default constructor to your compiled code

EmptyConstructor.kt

14

Custom `getter`

Dog.kt

- Add it to the property by writing it **immediately below** the property declaration
- Return type must match that of the property whose value you want to return.

```
// defines getter()
val weightInKgs:Double
    get()=weight / 2.2
```

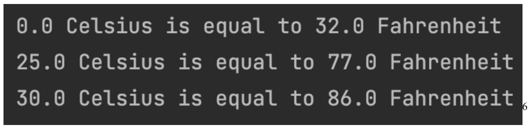
15

In-class Exercise

- Implement `class Temperature`

```
F = C * 1.8 + 32
C = (F - 32) * 5.0/9.0
```

```
fun main() {
    var temp = Temperature()
    println(temp) // default
    temp.celsius = 25.0
    println(temp)
    temp.fahrenheit = 86.0
    println(temp)
}
```



```
0.0 Celsius is equal to 32.0 Fahrenheit
25.0 Celsius is equal to 77.0 Fahrenheit
30.0 Celsius is equal to 86.0 Fahrenheit
```

16

```
class Temperature {
  var celsius = 0.0

  var fahrenheit
    get() = celsius * 1.8 + 32
    set(value) {
      celsius = (value - 32) * 5.0/9.0
    }

  override fun toString():String {
    return "$celsius Celsius is equal to $fahrenheit Fahrenheit"
  }
}

fun main() {
    val temp = Temperature()
    println(temp)           // default values
    temp.celsius = 25.0     // setter, C is 25 (default setter)
    // getter, C is 25 (default getter) and F is 77 --> 25*1.8+32
    println(temp)
    temp.fahrenheit = 86.0  // setter, C is 30 --> (86-32)*5/9
    // getter, C is 30 (default getter), F is 86 --> 30*1.8+32
    println(temp)
}
```

17

```
class Temperature {
    var celsius
        get() = (fahrenheit - 32) * 5.0/9.0
        set(value) {
            fahrenheit= value * 1.8 + 32
        }
    var fahrenheit = 32.0

    override fun toString():String {
        return "$celsius Celsius is equal to $fahrenheit Fahrenheit"
    }
}

fun main() {
    val temp = Temperature()
    println(temp)           // default values
    temp.celsius = 25.0     // setter, F is 77: 25*1.8+32
    // getter, C is 25: (77-32)*5/9 & F is 77 (default getter)
    println(temp)
    temp.fahrenheit = 86.0  // setter, F is 86 (default getter)
    // getter, C is 30: (86-32)*5/9, F is 86 (default getter)
    println(temp)
}
```

18

Misc.

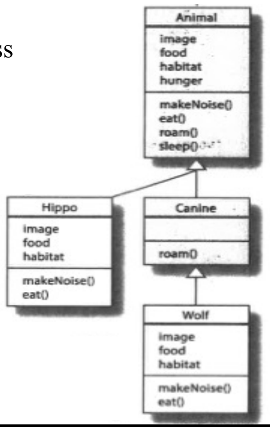
```
var myProperty:String
```

The compiler adds the following getters and setters when the code is compiled

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

You MUST initialize your properties in Kotlin before you try to use them. If not, You can prefix it with `lateinit`.

- Inheritance
- Abstract class
- Interface



Animal.kt

Inheritance

- Declare the superclass and its properties and functions as open
- To use a class as a superclass, it **MUST** be declared as open. Everything you want to override must also be open
- Calling the superclass (primary) constructor is mandatory.

```
// The Animal() after the : calls the Animal's constructor.
// This ensures that any Animal initialization code gets to run

class Hippo: Animal()
```

How about Java?

- In Kotlin, you can only inherit from super classes and override their properties and functions if they've been prefixed with **`open`**. **This is the opposite way round to how it works in Java.**
  - In Java, classes are open **by default**, and you use **`final`** to stop other classes inheriting from them or overriding their instance variables and methods.
- the **`open`** prefix makes it **explicit** as to which classes have been designed to be used as super classes, and which properties and functions can be overridden.

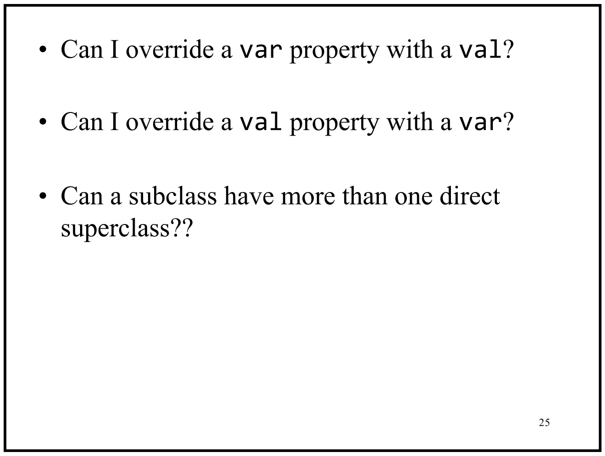
`final`

- What does `final` mean in Java?
- What does `final` mean in Kotlin?

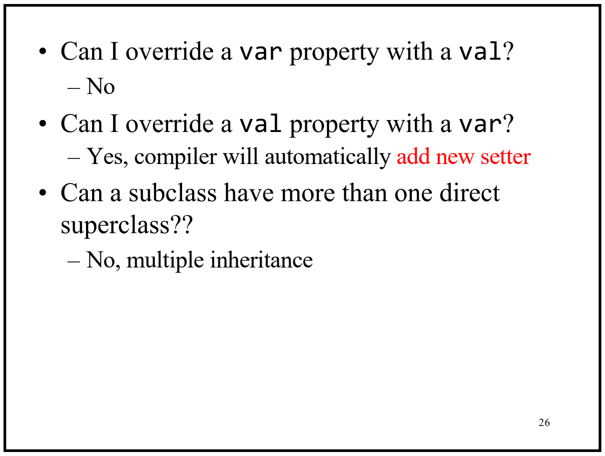
`final`

- What does `final` mean in Java?
- What does `final` mean in Kotlin?

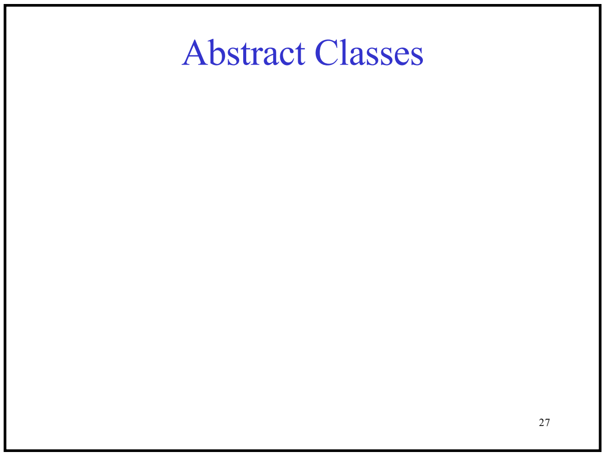
```
open class Wolf: Canine(){

  override val image = "wolf.jpg"
  override val food = "meat"
  override val habitat = "forests"

  override fun makeNoise()=println("Hooooowl!")

  // Declaring the function eat() as final in the Wolf class
  // means that it can no longer be overridden
  // in any of Wolf's subclass

  final override fun eat() = println("The Wolf is eating $food")
}
```



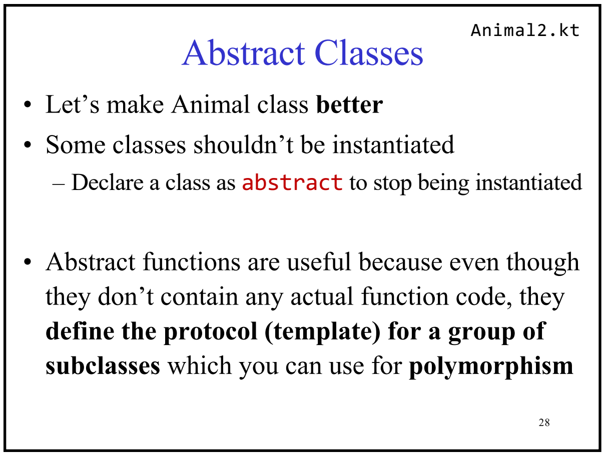
- Can I override a `var` property with a `val`?
- Can I override a `val` property with a `var`?
- Can a subclass have more than one direct superclass??



- Can I override a `var` property with a `val`?
  - No
- Can I override a `val` property with a `var`?
  - Yes, compiler will automatically add new setter
- Can a subclass have more than one direct superclass??
  - No, multiple inheritance



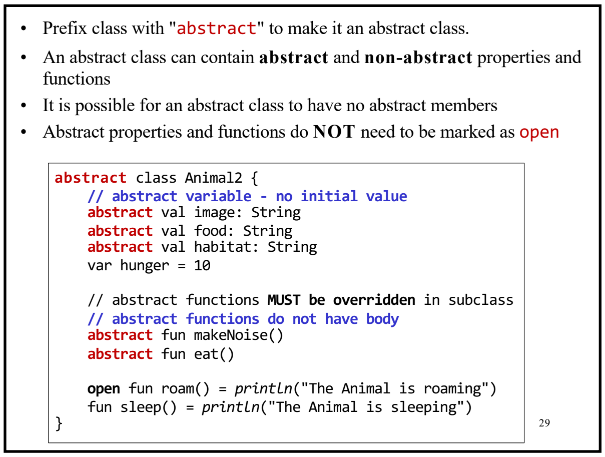
Abstract Classes



`Animal2.kt`

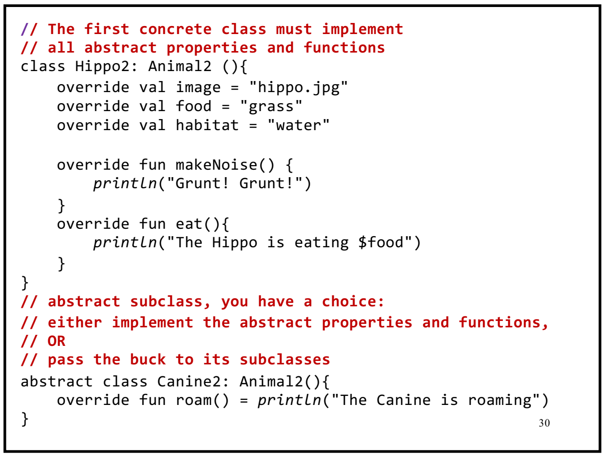
Abstract Classes

- Let's make Animal class **better**
- Some classes shouldn't be instantiated
  - Declare a class as `abstract` to stop being instantiated
- Abstract functions are useful because even though they don't contain any actual function code, they **define the protocol (template) for a group of subclasses** which you can use for **polymorphism**



- Prefix class with "`abstract`" to make it an abstract class.
- An abstract class can contain **abstract** and **non-abstract** properties and functions
- It is possible for an abstract class to have no abstract members
- Abstract properties and functions do **NOT** need to be marked as `open`

```
abstract class Animal2 {
    // abstract variable - no initial value
    abstract val image: String
    abstract val food: String
    abstract val habitat: String
    var hunger = 10

    // abstract functions MUST be overridden in subclass
    // abstract functions do not have body
    abstract fun makeNoise()
    abstract fun eat()

    open fun roam() = println("The Animal is roaming")
    fun sleep() = println("The Animal is sleeping")
}
```



```
// The first concrete class must implement
// all abstract properties and functions
class Hippo2: Animal2 (){
    override val image = "hippo.jpg"
    override val food = "grass"
    override val habitat = "water"

    override fun makeNoise() {
        println("Grunt! Grunt!")
    }
    override fun eat(){
        println("The Hippo is eating $food")
    }
}
// abstract subclass, you have a choice:
// either implement the abstract properties and functions,
// OR
// pass the buck to its subclasses
abstract class Canine2: Animal2(){
    override fun roam() = println("The Canine is roaming")
}
```

Abstract vs. open vs. default

- `abstract` keyword
- `open` keyword
- Default - No keyword used (no `abstract`, no `open`)

Abstract vs. open vs. default

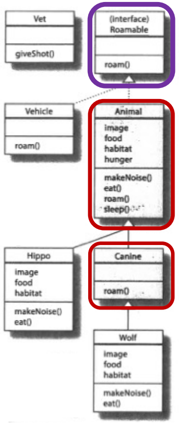
- `abstract` keyword
  - **Must** override
- `open` keyword
  - **May** override
- Default - No keyword used (no `abstract`, no `open`)
  - **Cannot** override

`Roamable.kt`

`Animal3.kt`

Interface

- When you add an abstract function to an interface, there is **no need** to prefix the function name with the `abstract` keyword.
  - With and interface, the compiler **automatically infers** that a function with no-body must be abstract, so you don't have to mark it as such.
- With an interface, you can override any of its properties and functions. So even if a function in an interface has a concrete implementation, you can still override it (even if it doesn’t have “open” or “abstract” keyword)
  - Interface cannot have constructors



Interface

Abstract class

Abstract subclass

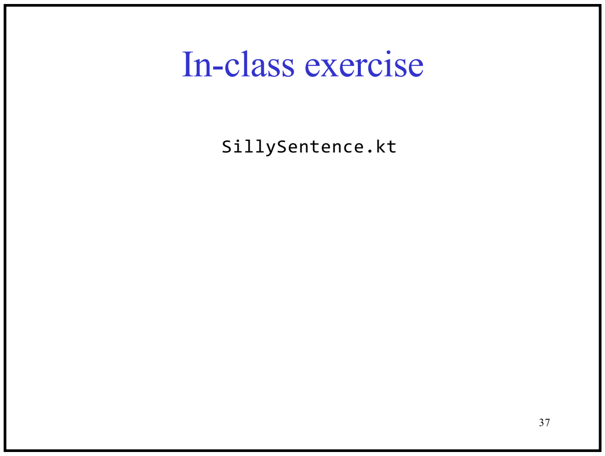
you can either implement the abstract properties and functions, OR pass the buck to its subclasses

First Concrete Class

Abstract class vs. Interface

Abstract class vs. Interface

- `Abstract classes` should be used primarily for objects that are closely related, whereas `interfaces` are best suited for providing a common functionality to unrelated classes.
- `Interfaces` are a good choice when we think that the API will not change for a while.



In-class exercise

`SillySentence.kt`