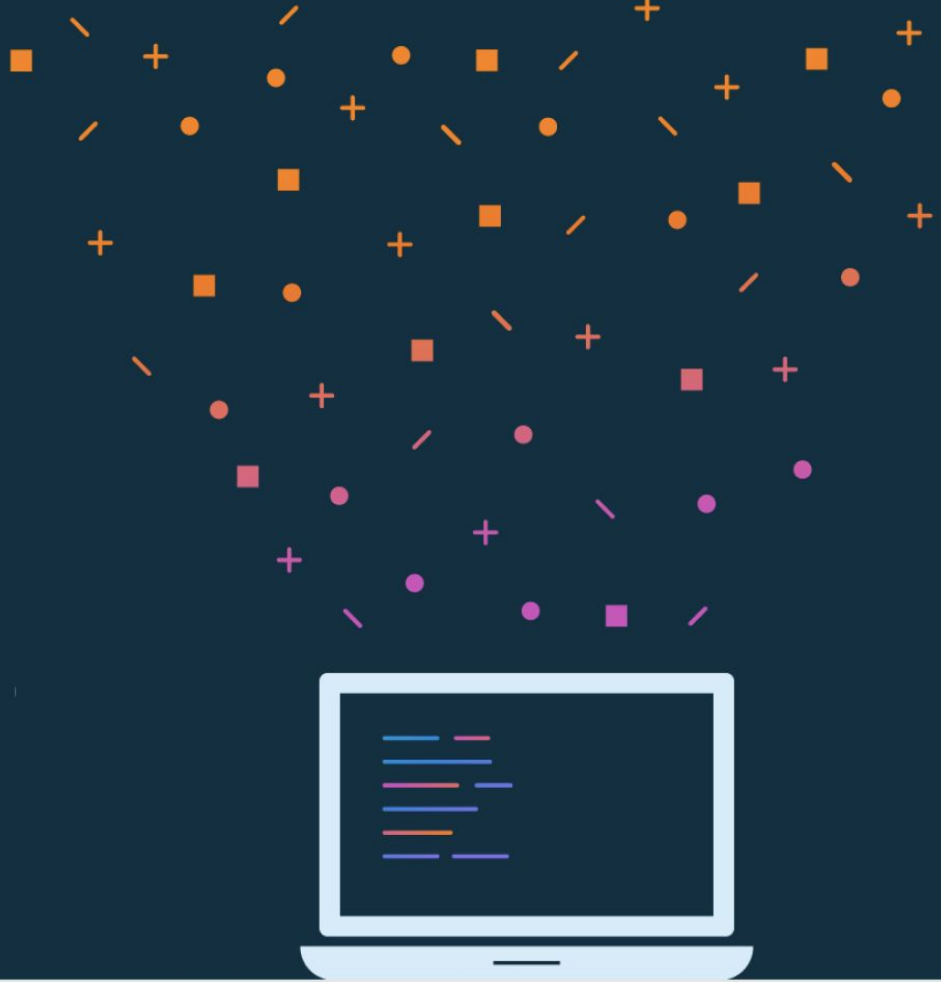
Lesson 3: Classes and objects



About this lesson

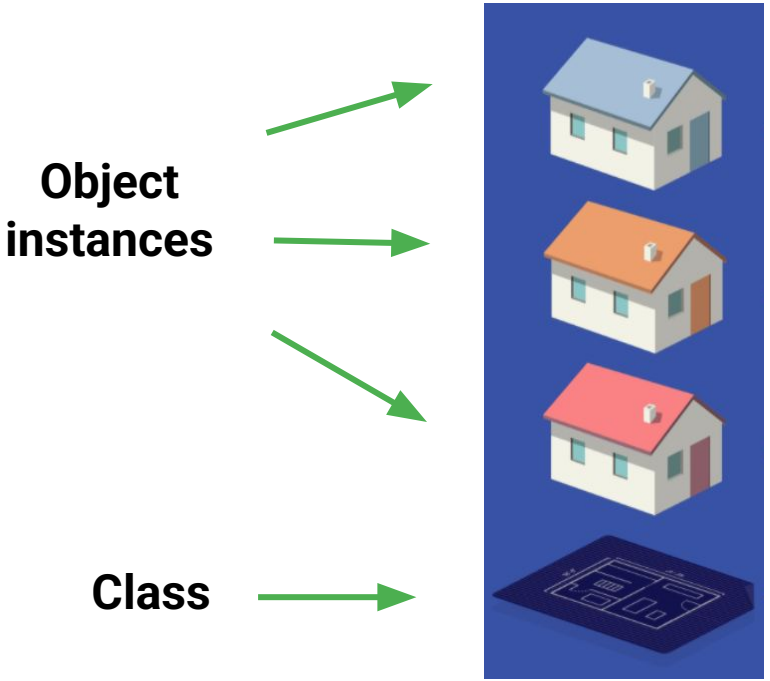
Lesson 3: Classes and objects

- Classes
- Inheritance
- Extension functions
- Special classes
- Organizing your code
- Summary

Classes

Class

- Classes are blueprints for objects
- Classes define methods that operate on their object instances



Class versus object instance

House Class

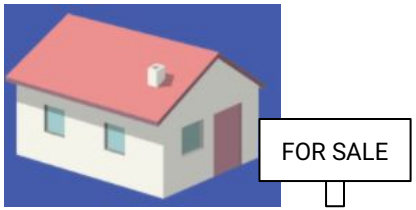
Data

- House color (String)
- Number of windows (Int)
- Is for sale (Boolean)

Behavior

- `updateColor()`
- `putOnSale()`

Object Instances



Define and use a class

Class Definition

```
class House {
  val color: String = "white"
  val numberOfWindows: Int = 2
  val isForSale: Boolean = false

  fun updateColor(newColor: String){...}
  ...
}
```

Create New Object Instance

```
val myHouse = House()
println(myHouse)
```

Constructors

When a constructor is defined in the class header, it can contain:

- No parameters

```
class A
```

- Parameters
  - Not marked with `var` or `val` → copy exists only within scope of the constructor

```
class B(x: Int)
```

  - Marked `var` or `val` → copy exists in all instances of the class

```
class C(val y: Int)
```

Constructor examples

```
class A
```

```
val aa = A()
```

```
class B(x: Int)
```

```
val bb = B(12)
println(bb.x)
=> compiler error unresolved
reference
```

```
class C(val y: Int)
```

```
val cc = C(42)
println(cc.y)
=> 42
```

Default parameters

Class instances can have default values.

- Use default values to reduce the number of constructors needed
- Default parameters can be mixed with required parameters
- More concise (don't need to have multiple constructor versions)

```
class Box(val length: Int, val width:Int = 20, val height:Int = 40)

val box1 = Box(100, 20, 40)
val box2 = Box(length = 100)
val box3 = Box(length = 100, width = 20, height = 40)
```

Primary constructor

Declare the primary constructor within the class header.

```
class Circle(i: Int) {
    init {
        ...
    }
}
```

This is technically equivalent to:

```
class Circle {
    constructor(i: Int) {
        ...
    }
}
```

Initializer block

- Any required initialization code is run in a special `init` block
- Multiple `init` blocks are allowed
- `init` blocks become the body of the primary constructor

Initializer block example

Use the `init` keyword:

```
class Square(val side: Int) {
    init {
        println(side * 2)
    }
}

val s = Square(10)
=> 20
```

Multiple constructors

- Use the `constructor` keyword to define secondary constructors
- Secondary constructors must call:
  - The primary constructor using `this` keyword

    OR

  - Another secondary constructor that calls the primary constructor
- Secondary constructor body is not required

Multiple constructors example

```
class Circle(val radius:Double) {
    constructor(name:String) : this(1.0)
    constructor(diameter:Int) : this(diameter / 2.0) {
        println("in diameter constructor")
    }
    init {
        println("Area: ${Math.PI * radius * radius}")
    }
}
val c = Circle(3)
```

Properties

- Define properties in a class using `val` or `var`
- Access these properties using
  dot . notation with property name
- Set these properties using
  dot . notation with property name (only if declared a `var`)

Person class with name property

```
class Person(var name: String)

fun main() {
    val person = Person("Alex")
    println(person.name)
    person.name = "Joey"
    println(person.name)
}
```

Access with .<property name>

Set with .<property name>

Custom getters and setters

If you don't want the default `get`/`set` behavior:

- Override `get()` for a property
- Override `set()` for a property (if defined as a `var`)

Format:

```
var propertyName: DataType = initialValue
        get() = ...
        set(value) {
            ...
        }
```

Custom getter

```
class Person(val firstName: String, val lastName:String) {
    val fullName:String
        get() {
            return "$firstName $lastName"
        }
}

val person = Person("John", "Doe")
println(person.fullName)
=> John Doe
```

Custom setter

```
var fullName:String = ""
    get() = "$firstName $lastName"
    set(value) {
        val components = value.split(" ")
        firstName = components[0]
        lastName = components[1]
        field = value
    }

person.fullName = "Jane Smith"
```

Member functions

- Classes can also contain functions
- Declare functions as shown in *Functions* in Lesson 2
  - `fun` keyword
  - Can have default or required parameters
  - Specify return type (if not `Unit`)

Inheritance

Inheritance

- Kotlin has single-parent class inheritance
- Each class has exactly one parent class, called a superclass
- Each subclass inherits all members of its superclass including ones that the superclass itself has inherited

If you don't want to be limited by only inheriting a single class, you can define an interface since you can implement as many of those as you want.

Interfaces

- Provide a contract all implementing classes must adhere to
- Can contain method signatures and property names
- Can derive from other interfaces

Format: `interface NameOfInterface { interfaceBody }`

Interface example

```
interface Shape {
    fun computeArea() : Double
}

class Circle(val radius:Double) : Shape {
    override fun computeArea() = Math.PI * radius * radius
}


val c = Circle(3.0)
println(c.computeArea())
=> 28.274333882308138
```

Extending classes

To extend a class:

- Create a new class that uses an existing class as its core (subclass)
- Add functionality to a class without creating a new one (extension functions)

Creating a new class

- Kotlin classes by default are not subclassable
- Use `open` keyword to allow subclassing
- Properties and functions are redefined with the `override` keyword

Classes are final by default

Declare a class

```
class A
```

Try to subclass `A`

```
class B : A
```

```
=>Error: A is final and cannot be inherited from
```

Use open keyword

Use `open` to declare a class so that it can be subclassed.

Declare a class

```
open class C
```

Subclass from C

```
class D : C()
```

Overriding

- Must use `open` for properties and methods that can be overridden (otherwise you get compiler error)
- Must use `override` when overriding properties and methods
- Something marked `override` can be overridden in subclasses (unless marked `final`)

Abstract classes

- Class is marked as `abstract`
- Cannot be instantiated, must be subclassed
- Similar to an interface with the added the ability to store state
- Properties and functions marked with `abstract` must be overridden
- Can include non-abstract properties and functions

Example abstract classes

```
abstract class Food {
    abstract val kcal : Int
    abstract val name : String
    fun consume() = println("I'm eating ${name}")
}
class Pizza() : Food() {
    override val kcal = 600
    override val name = "Pizza"
}
fun main() {
    Pizza().consume()    // "I'm eating Pizza"
}
```

When to use each

- Defining a broad spectrum of behavior or types? Consider an interface.
- Will the behavior be specific to that type? Consider a class.
- Need to inherit from multiple classes? Consider refactoring code to see if some behavior can be isolated into an interface.
- Want to leave some properties / methods abstract to be defined by subclasses? Consider an abstract class.
- You can extend only one class, but implement one or more interfaces.

Extension functions

Extension functions

Add functions to an existing class that you cannot modify directly.

- Appears as if the implementer added it
- Not actually modifying the existing class
- Cannot access private instance variables

Format: `fun ClassName.functionName( params ) { body }`

Why use extension functions?

- Add functionality to classes that are not open
- Add functionality to classes you don't own
- Separate out core API from helper methods for classes you own

Define extension functions in an easily discoverable place such as in the same file as the class, or a well-named function.

Extension function example

Add `isOdd()` to `Int` class:

```
fun Int.isOdd(): Boolean { return this % 2 == 1 }
```

Call `isOdd()` on an `Int`:

```
3.isOdd()
```

Extension functions are very powerful in Kotlin!

Special classes

Data class

- Special class that exists just to store a set of data
- Mark the class with the `data` keyword
- Generates getters for each property (and setters for vars too)
- Generates `toString()`, `equals()`, `hashCode()`, `copy()` methods, and destructuring operators

Format: `data class <NameOfClass>( parameterList )`

Data class example

Define the data class:

```
data class Player(val name: String, val score: Int)
```

Use the data class:

```
val firstPlayer = Player("Lauren", 10)
println(firstPlayer)
```

```
=> Player(name=Lauren, score=10)
```

Data classes make your code much more concise!

Pair and Triple

- `Pair` and `Triple` are predefined data classes that store 2 or 3 pieces of data respectively
- Access variables with `.first`, `.second`, `.third` respectively
- Usually named `data` classes are a better option (more meaningful names for your use case)

Pair and Triple examples

```
val bookAuthor = Pair("Harry Potter", "J.K. Rowling")
println(bookAuthor)
=> (Harry Potter, J.K. Rowling)

val bookAuthorYear = Triple("Harry Potter", "J.K. Rowling", 1997)
println(bookAuthorYear)
println(bookAuthorYear.third)
=> (Harry Potter, J.K. Rowling, 1997)
    1997
```

Pair to

`Pair`'s special `to` variant lets you omit parentheses and periods (infix function).

It allows for more readable code

```
val bookAuth1 = "Harry Potter".to("J. K. Rowling")
val bookAuth2 = "Harry Potter" to "J. K. Rowling"
```

```
=> bookAuth1 and bookAuth2 are Pair (Harry Potter, J. K. Rowling)
```

Also used in collections like Map and HashMap

```
val map = mapOf(1 to "x", 2 to "y", 3 to "zz")
```

```
=> map of Int to String {1=x, 2=y, 3=zz}
```

Enum class

User-defined data type for a set of named values

- Use `this` to require instances be one of several constant values
- The constant value is, by default, not visible to you
- Use `enum` before the `class` keyword

Format: `enum class EnumName { NAME1, NAME2, ... NAMEn }`

Referenced via `EnumName.<ConstantName>`

Enum class example

Define an `enum` with red, green, and blue colors.

```
enum class Color(val r: Int, val g: Int, val b: Int) {
    RED(255, 0, 0), GREEN(0, 255, 0), BLUE(0, 0, 255)
}

println("" + Color.RED.r + " " + Color.RED.g + " " + Color.RED.b)
=> 255 0 0
```

Object/singleton

- Sometimes you only want one instance of a class to ever exist
- Use the `object` keyword instead of the `class` keyword
- Accessed with `NameOfObject.<function or variable>`

Object/singleton example

```
object Calculator {
    fun add(n1: Int, n2: Int): Int {
        return n1 + n2
    }
}

println(Calculator.add(2,4))
=> 6
```

Companion objects

- Lets all instances of a class share a single instance of a set of variables or functions
- Use `companion` keyword
- Referenced via `ClassName.PropertyOrFunction`

Companion object example

```
class PhysicsSystem {
    companion object WorldConstants {
        val gravity = 9.8
        val unit = "metric"
        fun computeForce(mass: Double, accel: Double): Double {
            return mass * accel
        }
    }
}
println(PhysicsSystem.WorldConstants.gravity)
println(PhysicsSystem.WorldConstants.computeForce(10.0, 10.0))

=> 9.8100.0
```

Organizing your code

Single file, multiple entities

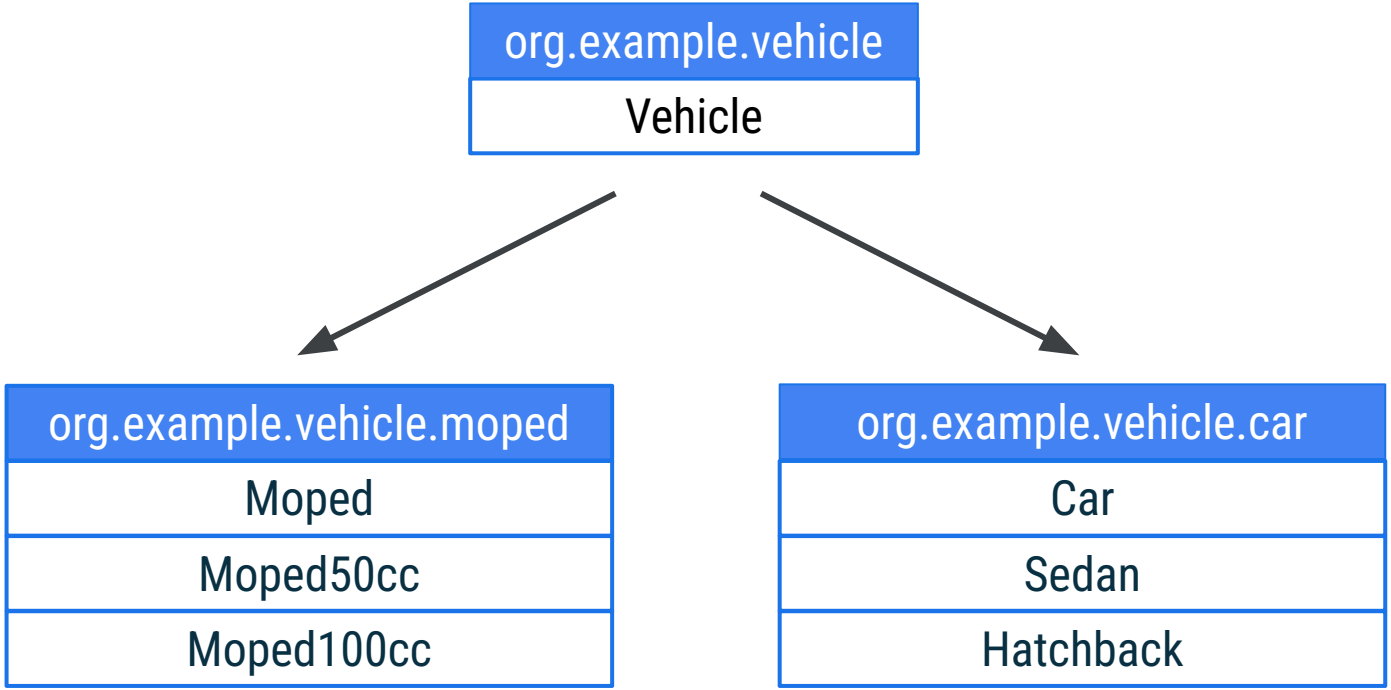
- Kotlin DOES NOT enforce a single entity (class/interface) per file convention
- You can and should group related structures in the same file
- Be mindful of file length and clutter

Packages

- Provide means for organization
- Identifiers are generally lower case words separated by periods
- Declared in the first non-comment line of code in a file following the `package` keyword

```
package org.example.game
```

Example class hierarchy



Visibility modifiers

Use visibility modifiers to limit what information you expose.

- `public` means visible outside the class. Everything is public by default, including variables and methods of the class.
- `private` means it will only be visible in that class (or source file if you are working with functions).
- `protected` is the same as `private`, but it will also be visible to any subclasses.

Summary

Summary

In Lesson 3, you learned about:

- Classes, constructors, and getters and setters
- Inheritance, interfaces, and how to extend classes
- Extension functions
- Special classes: data classes, enums, object/singletons, companion objects
- Packages
- Visibility modifiers

Pathway

Practice what you've learned by completing the pathway:

Lesson 3: Classes and objects