Object Orientated Programming

```
Course Code: ELEE1146

Course Name: Mobile Applications for Engineers

Credits: 15

Module Leader: Seb Blair BEng(H) PGCAP MIET MIEEE MIHEEM FHEA
```

Download as a PDF

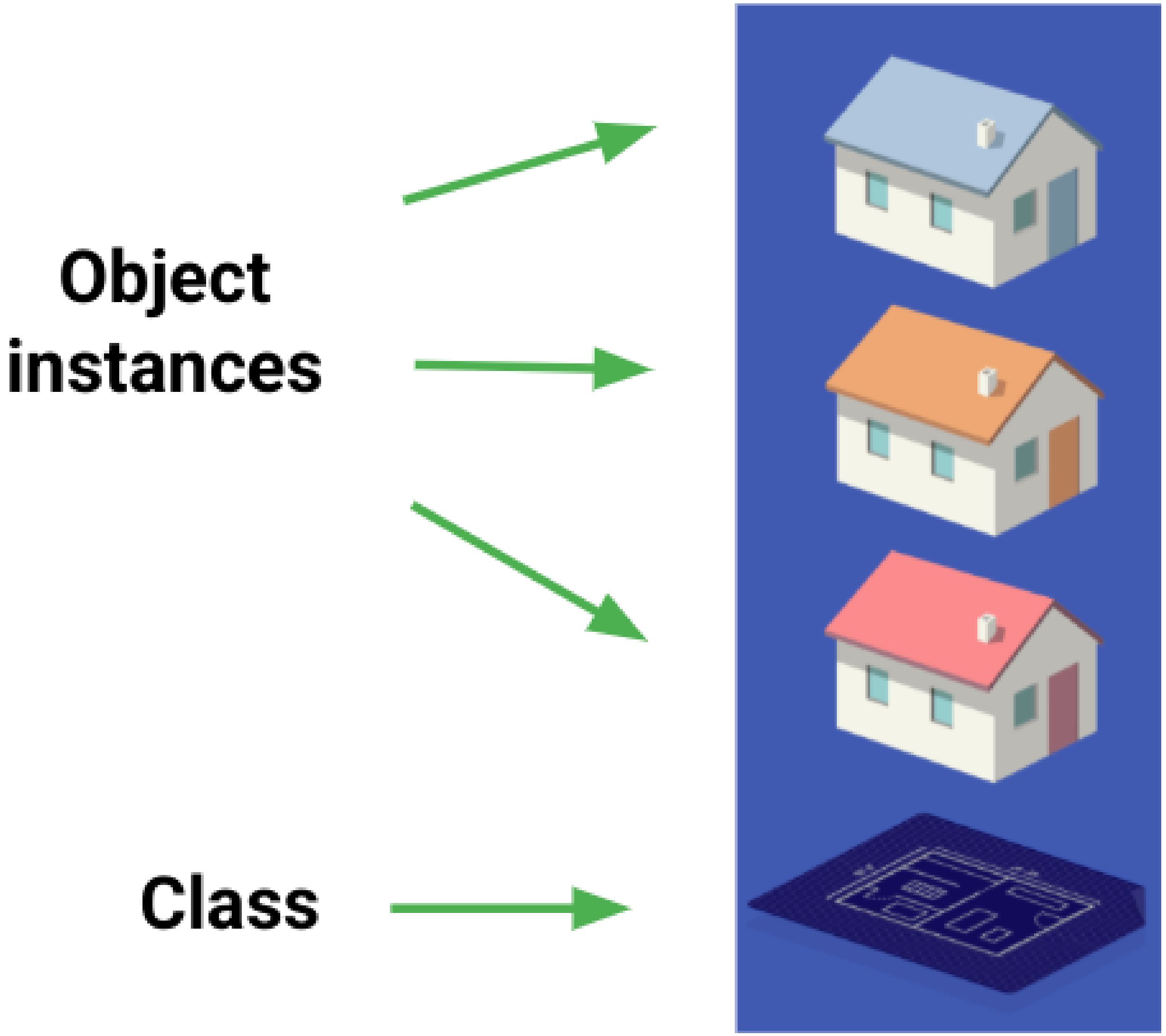
OOP Key Concepts

- Classes and Objects
- Functions and Methods
- Encapsulation
- Inheritance
- Polymorphism
- Interfaces

Classes

Classes are software programming models - abstractions of the real world or system entities.

Classes define methods that operate on their object instances



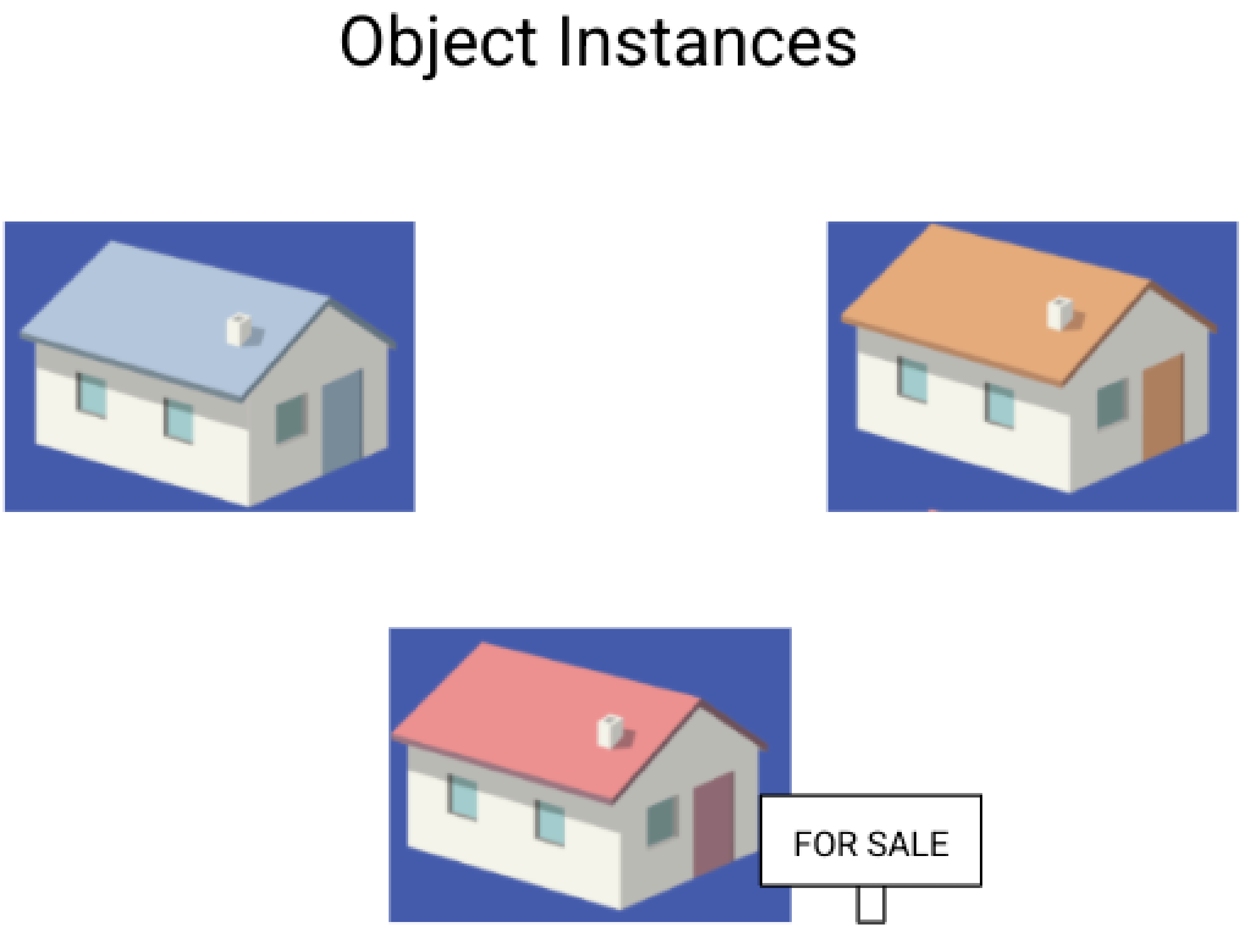
Object instances

Class

Classes vs Objects (2)

House Class

- Data
  - House color (`String`)
  - Number of windows (`Int`)
  - Is for sale (`Boolean`)
- Behavior
  - `updateColor()`
  - `putOnSale()`



Class - an Example

Class Definition

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

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

Object Creation

```kotlin
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)
    ```

Constructors 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
```

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) {
        ...
    }
}
```

Initialiser 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

```
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
- Secondary constructor body is not required

```
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`)

```kotlin
class Person(var name: String)
fun main() {
    val person = Person("A Name")
    println(person.name)           // Access with .<property name>
    person.name = "Your Name"           // Set with .<property name>
    println(person.name)
}
```

Setters and Getters

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`)

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

```
val person = Person("Your", "Name")
println(person.fullName)
=> Your Name
```

Custom Setter

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

```
person.fullName = "Marshall Mathers"
```

Inhertiance

- 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

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

Classes are Final

- Declare a class

  ```
  class A
  ```

- Try to subclass A

  ```
  class B : A
  ```

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

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

    ```
    open class C
    ```

  - Subclass from C

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

Abstraction

- 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

Abstraction Example

```kotlin
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"
}
```

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

Define the data class:

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

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

Pair and Triple ~~Tuple~~

- 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)

```
val bookAuthor = Pair("Prox Transmissions", "Dustin Bates & Peter David")
println(bookAuthor)
=> (Prox Transmissions, Dustin Bates & Peter David)

val bookAuthorYear = Triple("Prox Transmissions", "Dustin Bates & Peter David", 2017)
println(bookAuthorYear)
println(bookAuthorYear.third)
=> (Prox Transmissions, Dustin Bates & Peter David, 2017)
    2017
```

Pair to..

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

More readable

```
val bookAuth1 = "Prox Transmissions".to("Dustin Bates & Peter David")
val bookAuth2 = "Prox Transmissions" to "Dustin Bates & Peter David"
=> bookAuth1 and bookAuth2 are Pair (Prox Transmissions, Dustin Bates & Peter David)
```

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

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.GREEN.g + " " + Color.BLUE.b)
=> 255 255 255
```

Companion objects

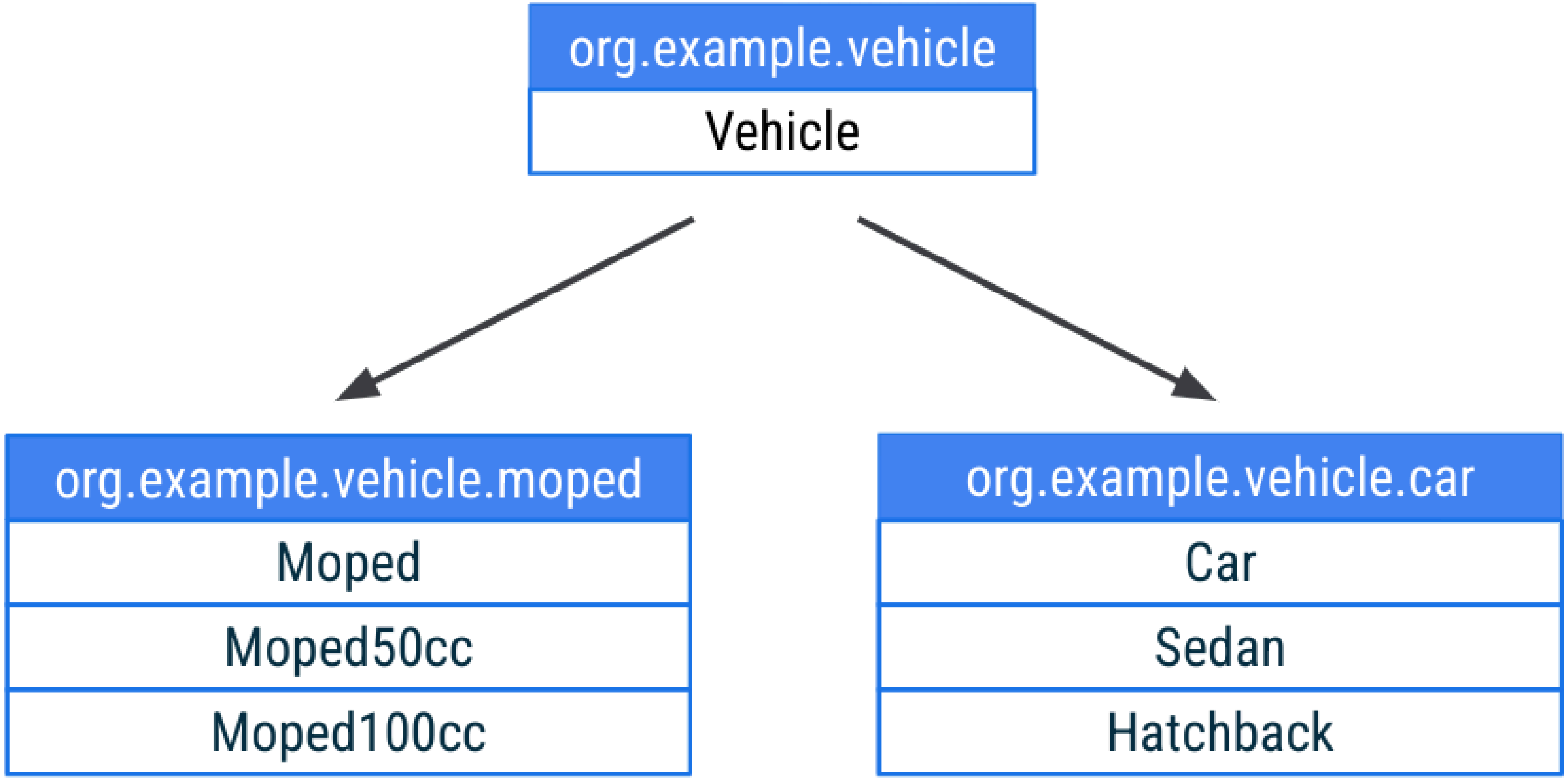
- Lets all instances of a class share a single instance of a set of variables or functions
- Use `companion` keyword
- Referenced via `ClassName.PropertyOrFunction`

```
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
```

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



org.example.vehicle

Vehicle

org.example.vehicle.moped

Moped

Moped50cc

Moped100cc

org.example.vehicle.car

Car

Sedan

Hatchback