

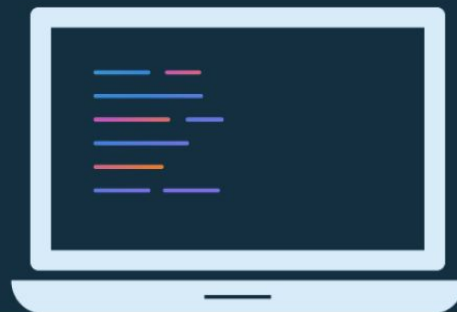


# Lezione 1.3

## Classi e oggetti



[Automaton - Jamiroquai](#)



# Questa lezione

## Lezione 1.3 - Classi e oggetti

- Classi
- Ereditarietà
- Extension functions
- Classi speciali
- Organizzare il codice

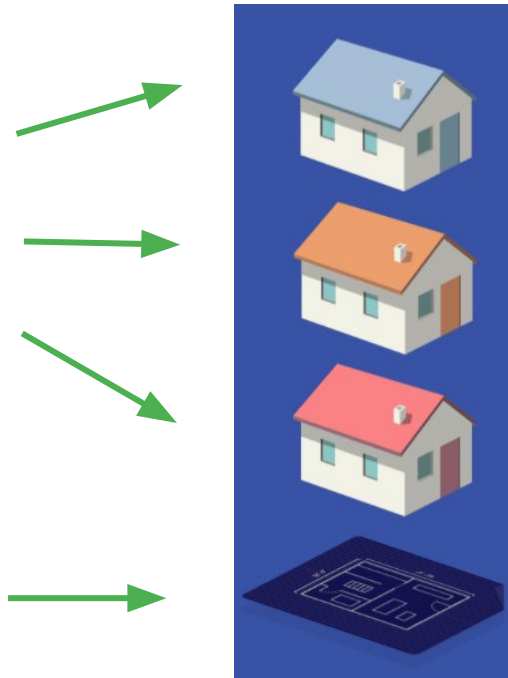
# Classi

# Classi

- Le classi sono le “planimetrie” (blueprints) o i prototipi degli oggetti
- Definiscono metodi che operano sulle loro occorrenze, o istanze (object instances)

**Object  
instances**

**Class**



# Class vs. object instance

## House Class

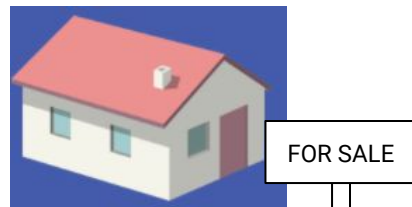
### Data

- Colore (String)
- Numero di finestre (Int)
- È in vendita (Boolean)

### Behavior

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

## Object Instances



# Definizione ed uso di una classe

## Definizione di una classe

```
class House {  
    val color: String = "white"  
    val numberOfWindows: Int = 2  
    val isForSale: Boolean = false  
  
    fun updateColor(newColor: String){...}  
    ...  
}
```

## Creazione di un'istanza

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

# Costruttori

Un costruttore viene definito nell'intestazione della classe e può contenere:

- Nessun parametro

```
class A
```

- Parametri

- Non marcati con `var` or `val` → la variabile esiste nell'ambito del costruttore o nel corpo della funzione ma solo per inizializzare delle variabili

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

- Marcati `var` or `val` → la variabile è trattata come una proprietà della classe

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



# Costruttori: esempi

```
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

Le istanze di una classe possono avere valori di default

- Sintassi più concisa (si evitano varianti multiple dei costruttori)
- I default parameters possono essere usati assieme a parametri obbligatori

```
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)
```

# Costruttore principale

Il costruttore principale si dichiara nell'intestazione della classe:

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

Questo è tecnicamente equivalente a:

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

# Blocco di inizializzazione

- Se necessario è possibile eseguire il codice di inizializzazione in uno speciale blocco `init`
  - Sono consentiti più blocchi `init`
  - I blocchi `init` divengono il corpo del costruttore principale
  - I blocchi `init` sono eseguiti nell'ordine in cui appaiono, eventualmente mescolati con dichiarazioni di proprietà
- (<https://kotlinlang.org/docs/classes.html#constructors>)

# Blocco di inizializzazione: esempio

Si utilizza la keyword `init`:

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

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

# Costruttori multipli

- Si usa la keyword `constructor` per definire dei costruttori secondari
- Questi devono chiamare (delegare):
  - Il costruttore principale usando la keyword `this`  
OPPURE
  - Un altro costruttore secondario che a sua volta chiamerà il costruttore primario
- Non è obbligatorio definire il corpo di un costruttore secondario

# Costruttori multipli: esempio

```
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)
```

# Proprietà

- Sono variabili definite in una classe. Possono essere dichiarate `val` o `var`
- L'accesso alle proprietà in lettura avviene con la dot notation (esempio: `variabile.proprietà`)
- La modifica delle proprietà avviene con la dot notation (solo se dichiarata `var`)



# Classe Person con proprietà name

```
class Person(var name: String)

fun main() {
    val person = Person("Alex")
    println(person.name) ← Accesso con .<property name>
    person.name = "Joey" ← Modifica con .<property name>
    println(person.name)
}
```

# Getters e setters personalizzati

Se non si vuole l'approccio di default per leggere/modificare:

- Override `get()` per una proprietà
- Override `set()` per una proprietà (se definita `var`)

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

# getter personalizzato

```
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
```

# setter personalizzato

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

*Backing field* che rappresenta la variabile su cui stiamo operando

```
person.fullName = "Jane Smith"
```

# Member functions

- Le classi possono contenere funzioni
- La dichiarazione avviene nel modo classico:
  - Keyword `fun`
  - Possono avere parametri di default o obbligatori
  - Specificano un tipo di ritorno (se non è `Unit`)

# Proprietà lateinit

Normalmente, le proprietà dichiarate non-nullable devono essere inizializzate nel costruttore, ma questo a volte non è conveniente (ad esempio se le inizializzo in un metodo specifico o tramite dependency-injection).

Kotlin permette la definizione di proprietà non-nullable che possono non essere inizializzate, utilizzando la keyword `lateinit`

```
lateinit var subject: TestSubject
```

# Proprietà lateinit

Condizioni:

- La proprietà deve essere `var` (altrimenti non potremmo inicializzarla dopo)
- La proprietà è dichiarata nel corpo della classe (e non nel costruttore primario)
- Il tipo deve essere non nullable
- Il tipo non deve essere “primitivo” (nel senso di Java, ossia `Int`, `Double`,...)

# Proprietà lateinit: esempio

```
class Dipartimento(val nome: String)

class Persona(val nome: String){

    lateinit var dipart: Dipartimento

    fun config(){
        dipart = Dipartimento("DII")
    }
    //O tramite injection dall'esterno
    fun config(d: Dipartimento){
        dipart = d
    }
}
```




# Proprietà lateinit: esempio

```
class Dipartimento(val nome: String)

class Persona(val nome: String){

    lateinit var dipart: Dipartimento

    fun config(d: Dipartimento){
        dipart = d
    }
    fun check() = if (::dipart.isInitialized) true else false
}
```



Per verificare se la proprietà è stata inizializzata.  
Nota: si può usare solo nella classe o nei sottotipi

# Interfacce

# Interfacce

- Forniscono un contratto a cui devono aderire tutte le classi che la implementano
- Può contenere le signature dei metodi e i nomi di proprietà
- Può derivare da altre interfacce

**Formato:** `interface` NameOfInterface { interfaceBody }

# Interfacce: esempio

```
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
```

# Interfacce

- Possono includere proprietà astratte oppure proprietà non astratte a patto che si definiscano dei metodi getter o setter
- Possono includere dichiarazione di metodi astratti o la loro implementazione

```
interface X {  
    val x: Int  
    get() = 5  
    fun foo() {  
        println("hello")  
    }  
    fun foo2()  
}
```

# Interfacce

Una classe può implementare più di una interfaccia, ma possono avvenire conflitti se esistono più implementazioni di un metodo. In questi casi occorre fare override del metodo indicando quale scegliere (o reimplementandolo).

```
interface A {  
    fun foo() { print("A") }  
    fun bar()  
}  
  
interface B {  
    fun foo() { print("B") }  
    fun bar() { print("bar") }  
}
```

```
class D : A, B {  
    override fun foo() {  
        super<A>.foo()  
        super<B>.foo()  
    }  
  
    override fun bar() {  
        super<B>.bar()  
    }  
}
```

# Ereditarietà

# Ereditarietà

- Kotlin ha un approccio single-parent class all'ereditarietà
- Ogni classe ha esattamente una classe genitore, chiamata superclasse
- Ogni sottoclasse eredita tutti i membri della sua superclasse inclusi quelli eventualmente ereditati dalla superclasse

Per superare il limite della singola superclasse, è possibile usare le interfacce, perché una classe può implementarne più di una.



# Estensione di classi

Per estendere una classe:

- Creare una nuova classe che usa una classe esistente come base (sottoclasse)
- Aggiungere funzionalità ad una classe senza crearne una nuova (extension functions)

# Creazione di una nuova classe

- Le classi in Kotlin sono `final` (non subclassable) per default
- Si può usare la keyword `open` per consentire il subclassing
- Proprietà e funzioni possono essere ridefinite tramite la keyword `override`

# Le classi sono final by default

Dichiariamo una classe:

```
class A
```

Proviamo a definire una sottoclasse:

```
class B : A()
```

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

# Uso di open


Se usiamo `open` per dichiarare una classe sarà possibile definirne sottoclassi.

Dichiarazione:

```
open class C
```

Sottoclasse di C:

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



Come per le interfacce, ma qui si usano le parentesi perché occorre chiamare il costruttore.

# Overriding

- Bisogna usare `open` per le proprietà ed i metodi che possono essere sovrascritti (altrimenti viene generato un errore in compilazione)
- Bisogna usare `override` quando si sovrascrivono proprietà e metodi
- Ogni cosa indicata come `override` può essere sovrascritta nelle sottoclassi (a meno che non sia marcata come `final`, ed in quel caso non potrà essere più ereditata)

# Superclassi ed interfacce

Se una classe eredita implementazioni multiple dello stessa funzione, ad esempio da una classe e da un'interfaccia, occorre effettuare l'override

```
open class Rectangle {  
    open fun draw() { /* ... */ }  
}  
  
interface Polygon {  
    fun draw() { /* ... */ } // interface members are 'open' by default  
}  
  
class Square() : Rectangle(), Polygon {  
    // The compiler requires draw() to be overridden:  
    override fun draw() {  
        super<Rectangle>.draw() // call to Rectangle.draw()  
        super<Polygon>.draw() // call to Polygon.draw()  
    }  
}
```

# Superclassi ed interfacce

...ma se l'interfaccia non avesse fornito l'implementazione, l'override non sarebbe stato necessario perché Square ha già una implementazione della funzione, fornita da Rectangle

```
open class Rectangle {  
    open fun draw() { /* ... */ }  
}  
  
interface Polygon {  
    fun draw()  
}  
  
class Square() : Rectangle(), Polygon {  
}
```

# Classi astratte

- La classe viene marcata come `abstract`
- Non possono essere istanziate ma devono essere estese da sottoclassi
- Simili ad un'interfaccia ma con la capacità di memorizzare dati (cioè posseggono uno stato)
- Proprietà e funzioni marcate `abstract` devono essere sovrascritte
- Possono includere proprietà e funzioni non astratte



# Classi astratte: esempio

```
abstract class Food {  
    abstract val kcal : Int  
    abstract val name : String  
    val canBeEaten = true  
    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"  
}
```

# Quando usare cosa?

- Viene definito un largo insieme di behavior o tipi? Considera un'interfaccia.
- Il behavior sarà specifico a quel tipo? Considera una classe.
- Devi ereditare da classi multiple? Considera un refactoring per vedere se qualche behavior può essere isolato in un'interfaccia.
- Vuoi lasciare qualche proprietà / metodo astratto per lasciarlo definire dalle sottoclassi? Considera una classe astratta.

# Classi speciali

# Data class

- Una classe creata per memorizzare un insieme di dati
- E' sufficiente marcare la classe con la keyword `data`
- Genera automaticamente i metodi getter per ogni proprietà (ed i setter per le variabili `var`)
- Genera automaticamente i metodi `toString()`, `equals()`, `hashCode()`, `copy()`, e gli operatori di destructuring

**Formato:** `data class` <NameOfClass>( parameterList )

# Data class: esempio

Definiamo una data class:

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

Utilizziamo la data class:

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

```
println(firstPlayer)
```

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

Viene implicitamente chiamato toString() che per le data class formatta automaticamente i valori

Le data class rendono il codice molto più conciso!

# Data class

Se una proprietà viene dichiarata nel corpo, non verrà valutata nell'esecuzione di toString, equals, hashCode e copy

```
data class Person(val name: String) {  
    var age: Int = 0  
}
```

```
val p1 = Person("Pippo")  
p1.age = 25  
val p2 = Person("Pippo")  
println(p1 == p2)  
=> true
```

Nota: per testare  
l'uguaglianza tra due  
oggetti data class, oltre  
ad equals si può usare ==

# Data class

Il costruttore copy consente di copiare un oggetto in un altro

```
data class Person(val name: String, var age: Int = 0)
val p = Person("Geoffrey", 40)
val p2 = p.copy()
```

È possibile anche alterare parametri val durante la copia

```
val p2 = p.copy(name = p.name + "Smith")
```

# Pair & Triple

- `Pair` e `Triple` sono `data class` predefinite che memorizzano 2 o 3 dati rispettivamente
- Si accede alle variabili con `.first`, `.second`, `.third` rispettivamente
- Tipicamente le normali `data classes` sono una miglior opzione (si possono usare nomi più significativi per gli attributi)



# Pair & Triple: esempi

```
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

La variante `to` di `Pair` consente di omettere le parentesi e il punto (notazione infissa), consentendo di scrivere codice più leggibile:

```
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)
```

Viene usata anche nelle collection come `Map` e `HashMap`

```
val map = mapOf(1 to "x", 2 to "y", 3 to "zz")  
=> map of Int to String {1=x, 2=y, 3=zz}
```

# Pair & Triple: destructuring declarations

Pair e Triple possono essere utilizzati per ospitare i valori delle proprietà di una data class

```
data class Person(val name: String, val surname: String, val age: Int)
val bugo = Person("Christian", "Bugatti", 48)
val (one,two,three) = bugo
```

Questo approccio si chiama *destructuring declaration*, e può essere utile anche per ritornare più un valore da una funzione. Se un elemento non serve può essere indicato con un underscore

```
val (result,status) = function(...)
val (result, _) = function(...)
```

# Destructuring declarations

In generale il meccanismo funziona per qualsiasi oggetto per cui si possa chiamare il metodo `componentN()`, che ritorna l'n-esima proprietà dell'oggetto:

`val (name, age) = person` equivale a `val name = person.component1()`  
`val age = person.component2()`

Si può usare anche per iterare un map:

```
for ((key, value) in map) { ... }
```

# Enum class

Un tipo di dato definito dall'utente per un insieme di valori finiti

- Utilizza `this` per richiedere che le istanze siano uno dei valori costanti di una lista
- Il valore costante non è visibile di default
- Utilizzare `enum` prima della keyword `class`

**Formato:** `enum class` EnumName { NAME1, NAME2, ... NAME<sub>n</sub> }

Si può accedere tramite EnumName.<ConstantName>

# Enum class: esempio

Defizione di un `enum` con tre colori:

```
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

- A volte vogliamo che possa esistere una singola istanza di una classe
- Utilizziamo la keyword `object` invece di `class`
- Si accede con `NameOfObject.<function o variable>`
- Possono essere incluse in altre classi

# Object/singleton: esempio

```
object Singleton{  
    var variableName = "I am Var"  
  
    fun printVarName(){  
        println(variableName)  
    }  
}  
  
...  
  
Singleton.variableName = "Nuovo valore"
```



# Companion objects

- E' un `object` ma dentro una classe
- Consente a tutte le istanze di una classe di condividere una istanza di un insieme di variabili o funzioni (come fa la keyword "static" in Java)
- Si utilizza la keyword `companion`
- Si accede come `ClassName.PropertyOrFunction` (come se fosse una porzione "statica" di una classe)
- Approfondimento: <https://www.youtube.com/watch?v=Dt8zTBdDv5w>

# Companion object: esempio

```
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
```

# Organizzare il codice

# Single file, multiple entities

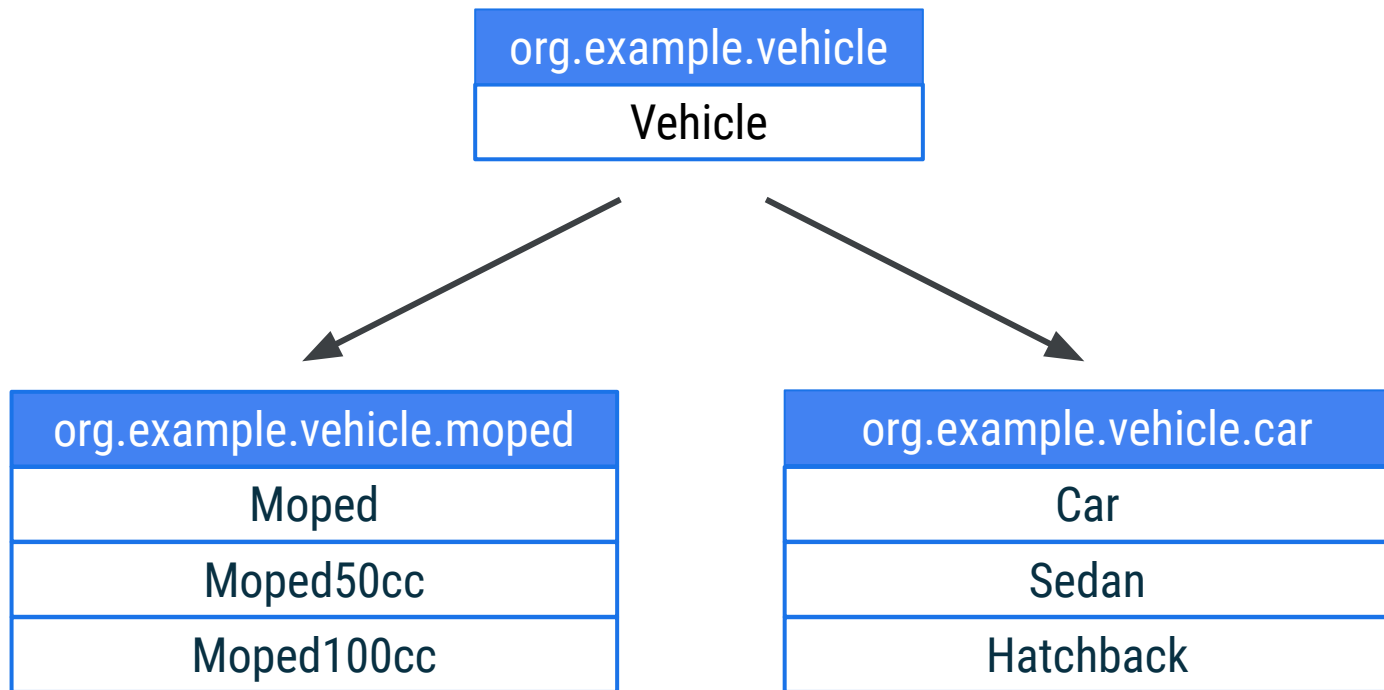
- Kotlin NON impone la convenzione di definire una singola entità (classe/interfaccia) per file
- E' possibile (e consigliato) raggruppare strutture correlate nello stesso file
- Attenzione alla lunghezza dei file e al disordine

# Packages

- Forniscono un mezzo per organizzare un progetto
- Gli identificatori sono in genere parole in minuscolo separate da punti
- Vengono dichiarati nella prima linea non commentata di codice del file preceduti dalla keyword `package`

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

# Esempio di gerarchia di classi



# Packages

- Vengono importati di default `kotlin.*`, `kotlin.annotation.*`, `kotlin.collections.*`, `kotlin.comparisons.*`, `kotlin.io.*`, `kotlin.ranges.*`, `kotlin.sequences.*`, `kotlin.text.*`
- Ad essi se ne aggiungono altri in base al target di compilazione (ad es: `java.lang.*` e `kotlin.jvm.*` se per JVM)

# Modificare la visibilità

Utilizza le seguenti keyword per limitare l'informazione che viene esposta:

- `public` = visibile al di fuori della classe. Ogni informazione è `public` di default, incluse le variabili ed i metodi della classe.
- `private` = visibile soltanto nella classe (o nel file sorgente se siamo al di fuori di una classe).
- `protected` = come `private`, ma sarà visibile alle sottoclassi
- Nota: non c'è un modificatore relativo esclusivamente ai package (a parte `sealed` per le classi che sono possono essere estese solo da altre classi dentro lo stesso package)