Lezione 2

Singleton → oggetto che restituisce sempre la stessa istanza.

Solitamente è molto complesso da realizzare; per questo motivo utilizziamo direttamente il tipo object.

La singola istanza si ha, come in java, a runtime.

La singoleton è utile, in quanto all'interno di ogni classe possiamo avere un singleton. Il companion object introduce i metodi **statici**. Essi infatti hanno lo stesso valore per la classe.

I metodi statici possono essere accetudi chiamando solamente il nome della classe.

Per aggiungere un nuovo metodo della classe, allora basta chiamare la classe, il nome della funzione che vogliamo aggiunge.

Le extstansion function sono problematici, in quanto l'ide deve scannerizzare tutto il codice per dover suggerire il metodo che estendiamo, l'ide deve scannerizzare tutto il codice.

Di default, un classe classica non può essere **estesa**. Infatti, essa deve essere annotata come "open". Anche i metodi devono essere firmati come open per essere sovrascritti.

In java, se volevamo che una classe non sia estendibile, allora possiamo tipizzare quella classe come final, al contrario, in kotlin, possiamo tipizzare la classe con closed, per cercare di estenderla.

La differenza tra ereditarietà e interface, sta nel fatto che con l'ereditarietà dobbiamo inserire gli attributi all'iterno della classe che estendiamo, invece, con le interfacce, basta solmanete scrivere il nome dell'interfaccia.

```
open class Rectangle(var width: Double, var height: Double) {
open fun draw() { /* ... */ }
}
interface Polygon {
```

```
fun area(): Double // interface members are 'open' by default
}
class Square(side: Double) : Rectangle(side, side), Polygon {
// Both draw() and area must be overridden:
override fun draw() {
super<Rectangle>.draw() // call to Rectangle.draw()
}
override fun area() = width*height
}
```

Data class

Un data class è un contenitore che non ha metodi, e non ha metodi, non può essere estesa, o inner, o seald. Essa deve avere un costruttore.

```
//dato questo codice:
class Simple(val a:String,val age:Number)


fun main() {
    val v1 = Simple("Joe",24)
    val v2 = Simple("Joe",24)
    println(v1)
    println(v2)
    print(v1===v2)
}
//output:
/*
Simple@3941a79c
Simple@b1bc7ed
false
*/
```

```
//dato questo codice:
data class Simple(val a:String,val age:Number)
```

```
fun main() {
    val v1 = Simple("Joe",24)
    val v2 = Simple("Joe",24)
    println(v1)
    println(v2)
    print(v1===v2)
}
//output:
/*
Simple(a=Joe, age=24)
Simple(a=Joe, age=24)
true
*/
```

Con il data class si ha un working implementation del metodo equal e metodo toString.

```
class Simple(val name:String,val age:Number){
    override fun hashCode():Int {
        return this.name.hashCode()*31 + this.age.hashCode();
    }
    //mettiamo il "?" in quanto possiamo comparare un oggetto con un null.
    override fun equals(other:Any?):Boolean {
        if(other == null) return false
        else{
            //=== abbiamo lo stesso PUNTATORE. ovvero se abbiamo lo stesso punta
            if(other === this) return true;
            //ogni classe proprietà ha una proprietà chiamata javaClass, la quale essa
            else if (other.javaClass != this.javaClass) return false
            else {
                val o = other as Simple
                return (this.name == other.name && this.age == other.age)
            }
```

```
        }
    }
}


fun main() {
    val v1 = Simple("Joe",24)
    val v2 = Simple("Joe",23)
    println(v1)
    println(v2)
    print(v1==v2)
}
```

Esempi risolti

```
data class Simple(val name:String,val age:Number)
fun main() {
    val v1 = Simple("Jill",23);
    val v2 = v1.copy();

    println(v1)
    println(v2)

    println(v1 == v2)
    //gli oggetti sono identici?
    println(v1 === v2)
    //copy con un valore, allora copia un valore cambiando il valore passato in inpu
    val v3 = v1.copy(age = 32);
    println(v3)
    //decostruisco la classe: sarebbe component1(),..., componentN()
    val (n,age) = v1;
    println(n);
    println(age);
}
```

Altri tipi di classi possono essere **enum class.**

Esse forniscono una serie di field.

Una enum può anche avere un valore, il quale può anche essere scritto da tutti gli elementi della enum.

Ad esempio:

```
enum class HttpStatusCode(val value: Int) {
OK(200), CREATED(201), NOT_FOUND(404),
ACCESS_DENIED(403), INTERNAL_SERVER_ERROR(500)
}
```

L'enum è utile, in quanto può essere utilizzata facilemte dalla keywork `When` .

Le enum hanno una limitazione, ovvero che ogni elemento deve incorporare il comportamento comune della enum. Ovvero che nella enum, tutti gli elemnti devono avere lo stesso numero di attributi. Per questo motivo si usa la **sealed class**. Essa è una classe che può essere estesa all'interno dello stesso **codice sorgente**.

Ad esempio, in questo caso:

```
sealed class Result
data class Success(val data: List) : Result()
data class Failure(val error: Throwable?) : Result()
fun processResult(result: Result): List = when (result) {
is Success → result.data
is Failure → listOf()
}
```

La classe sealed result, può essere estesa **solo all'interno del file.** Infatti, la classe Success, e Failure, estendono la classe result, all'interno della classe stessa. La classe result, però, può essere utilizzata ovunque.

Con la sealed class si possono eseguire Some e None come in rust.

Generics

Come in java, anche Kotlin implementa i generics.

All’intero essa può essere chiamata attraverso le “<>”.

Possiamo avere classi generiche o funzioni generiche.

Supponiamo di avere due box, una box di number, e una box di integer.

Le keyword out e in, indicano che stiamo cercando una sottoclasse (in) o una superclass (nel caso dell’out).

Out → se inserisco un tipo specifico, allora allora ottengo un tipo più specifico.

In → se inserisco un tipo generico, allora ottengo un tipo più generico.

💡 Chat Risponde:

🔷 Variance in Generic Classes in Kotlin

La **variance** nei generics in Kotlin riguarda **come i tipi generici vengono propagati attraverso l'ereditarietà** e come possono essere usati in input o output.

In particolare, Kotlin introduce i modificatori in e out per controllare il comportamento dei tipi generici nelle classi.

1️⃣ Il Problema della Variance

Immaginiamo di avere queste classi:

```
open class Animal
class Dog : Animal()
class Cat : Animal()
```

Ora, supponiamo di avere una classe generica:

```
class Box<T>
```

La domanda è: **se Dog è un sottotipo di Animal, allora Box<Dog> è un sottotipo di Box<Animal>?**

👉 **La risposta è NO!** Kotlin non lo permette per evitare problemi di tipo.

2️⃣ Out-Variance (out T)

✔️ Covariance (output)

Quando un tipo generico è **covariante**, significa che **può essere usato come output**, ma **non come input**.

In Kotlin, si usa il modificatore out per indicare che un tipo generico **può solo produrre valori di tipo T, ma non consumarli**.

```
interface Producer<out T> {
    fun produce(): T
```

```
}
```

🔷 **Esempio**:

```
class DogProducer : Producer<Dog> {
    override fun produce(): Dog = Dog()
}

val animalProducer: Producer<Animal> = DogProducer() // ✅ Funzi
ona grazie a `out`
```

✅ **Perché funziona?**

- DogProducer produce un Dog, che è un sottotipo di Animal.
- **Quindi è sicuro trattarlo come Producer<Animal>**, perché tutto ciò che produce è **almeno** un Animal.

🚨 **Regola: Un tipo contrassegnato con out può essere usato solo come valore restituito da funzioni, non come parametro di input.**

Ad esempio, non possiamo scrivere:

```
interface Producer<out T> {
    fun consume(value: T) // ❌ Errore! Non possiamo usare `T` in inp
ut se è `out`
}
```

3️⃣ **In-Variance (in T)**

✔️ **Contravariance (input)**

Quando un tipo generico è **contravariante**, significa che **può essere usato solo come input**, ma **non come output**.

In Kotlin, si usa in per indicare che un tipo **può solo consumare valori di tipo T, ma non produrli**.

```
interface Consumer<in T> {
    fun consume(value: T)
```

```
}
```

🔷 **Esempio**:

```
class AnimalConsumer : Consumer<Animal> {
    override fun consume(value: Animal) {
        println("Consuming an animal")
    }
}

val dogConsumer: Consumer<Dog> = AnimalConsumer() // ✅ Funzi
ona grazie a `in`
dogConsumer.consume(Dog()) // Accetta un Dog
```

✅ **Perché funziona?**

- AnimalConsumer può consumare qualsiasi Animal, quindi è sicuro assegnarlo a una variabile che consuma Dog (perché un Dog è un Animal).

🚨 **Regola: Un tipo contrassegnato con in può essere usato solo come parametro di funzione, non come valore restituito.**

Ad esempio, non possiamo scrivere:

```
interface Consumer<in T> {
    fun produce(): T // ❌ Errore! Non possiamo usare `T` in output se
è `in`
}
```

4️⃣ **Caso Speciale: Invarianza (nessun in o out)**

Se **non specifichiamo in o out**, la classe è **invariante**, cioè Box<Dog> **non** è un sottotipo di Box<Animal>.

```
class Box<T>(private val value: T) {
    fun get(): T = value
```

```
    fun set(value: T) { /* ... */ }
}
```

🔷 **Esempio**:

```
val dogBox: Box<Dog> = Box(Dog())
// val animalBox: Box<Animal> = dogBox // ❌ Errore! Invarianti non sono compatibili
```

Perché?

- Box<T> può **sia consumare che produrre** T, quindi Kotlin non può garantire sicurezza nei tipi.

🔷 **Riepilogo**

Modificatore	Significato	Esempio	Sicuro?
out T	Covariante (solo output)	List<out Animal> = List<Dog>() ✅	✅ Sicuro per la lettura
in T	Contravariante (solo input)	Consumer<in Dog> = Consumer<Animal>() ✅	✅ Sicuro per la scrittura
Nessuno (T)	Invariante (né input né output)	Box<Dog> **NON** è sottotipo di Box<Animal> ❌	🚨 No, può causare problemi

🎯 **Conclusione**

- out T → **Usato quando la classe genera valori (Producer)**
- in T → **Usato quando la classe consuma valori (Consumer)**
- Se non usiamo in o out, la classe è **invariante** e segue regole più restrittive.

📌 **Regola mnemonica:**

- out → **output** (solo produzione)
- in → **input** (solo consumo)

> Se vuoi garantire sicurezza nei tipi nei tuoi generics, **usa in e out in base al loro utilizzo!** 🚀

Scoped function

???

Se creiamo una estention type per ogni tipo function, allora possiamo applicare con T.apply, o T.run ecc... .

Kotlin fornisce già funzioni preimpostate.

- let → accetta una funzione (lambda) per fare ispezione del suo contenuto
- apply → serve se vogliamo manipolare l'oggetto stesso.
- run → usato per fare alcune operazioni con oggetti nullable.
- with → esegue operazioni su non null object

l'operatore as casta l'oggetto, un tipo specifico.

Però se il tipo su cui vogliamo castare non è il tipo deisiderato, allora il codice crasha. Per evitare questo, allora utilizziamo l'operatore ?, ad esempio as?.

```
open class C1(val i:Int)
class C2(i:Int):C1(i){
    fun m() = i
}

fun main() {
    var a:Any? = null;

    a = C2(65);

    println(a is C1); //true
    println(a is C2); //true
}
```

In questo caso a è sia di classe C1 e di classe C2.

smart casting

```
open class C1(val i:Int)
class C2(i:Int):C1(i){
    fun m() = i
}

fun main() {
    var a:Any? = null;

    a = C2(65);

    //esso è chiamato smart casting
    if(a is C2){
        a.m()
    }
}
```

Se sposto la variabile in globale, allora essa lancia un errore, infatti:

```
open class C1(val i:Int)
class C2(i:Int):C1(i){
    fun m() = i
}

var a:Any? = null;

fun main() {

    a = C2(65);

    //esso è chiamato smart casting
    if(a is C2){
        a.m() //errore
```

```
    }
}
```

La differenza tra list e set, è che un set non contiene elementi duplicati.

Possiamo operare con le list attravero una functional form.

map si fa con:

```
 val m = mapOf("one" to 1,"two" to 2,"three" to 3,"four" to 4,"five" to 5);
```

Una sequence è un contenitore lazy, e le operazioni vengono eseguite in modo asyncrono.

```
val words = "The quick brown fox jumps over the lazy dog".split(" ")
val lengthsList = words.filter { it.length > 3 }
.map { it.length }
.take(4) //prendi solamente 4 elementi
```

ci sono 23 operazioni, oppure posso fare:

```
val words = "The quick brown fox jumps over the lazy dog".split(" ")
val wordsSequence = words.asSequence()
val lengthsList = wordsSequence.filter {it.length > 3 }
.map { it.length }
.take(4)
.toList()
```

Adesso lo guardo come un sequence ovvero un lazy iterable collection. ?????