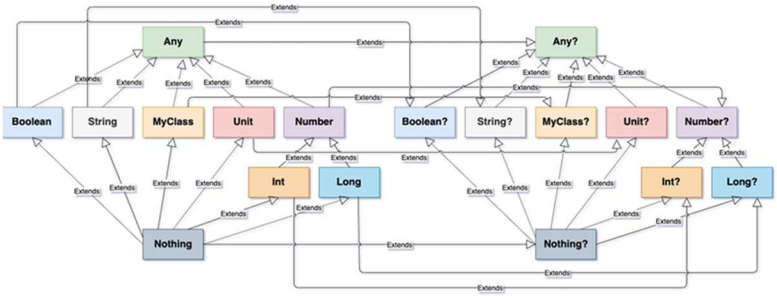
KOTLIN

Linguaggio staticamente **tipato**.
Sistema di inferenza di kotlin decide il tipo del valore nel caso in cui non lo si specifica.

- kotlin prova a capire automaticamente il tipo della variabile
    - Se voglio specificare il tipo → var i: Int = 5;

TIPI:

- Int, Short, Long, Byte, Float, Double, Char, Boolean
- Class **Any** → root della gerarchia
- Class **Nothing** → bottom della gerarchia
    - contiene null
- Class **Nothing?** → Non può contenere nulla, lì per completezza



Se seguiti dal ? sono nullabili, altrimenti no

NOTA: Se dichiaro var b=null; lo legge come un Nothing?

?. → safe-call operator: se il valore non è nullo, si traduce in . ; Altrimenti ritorna null.
?: → se quello di sx e null, valore dx

```
var a: String = "abc"

// a = null // compilation error

var b: String? = "abc"

b = null  //ok

var len: Int? = b?.length // len: null

var c: String = b ?: "default" // c: "default"

var len2:Int = b!!.length // Null-Pointer Exception
```

Dichiarazione di variabili

- **var** i=5; → dichiaro variabile e il tipo viene dedotto
- **val** i=5; → value non riassegnabile → costante

NOTE:

- Le variabili sono salvate **sull'heap**.
- Posso cambiare il valore delle variabili ma non il tipo (Anche se non l'ho specificato)
    - Posso fare var i=5; i=2;
    - Non posso fare var i=4; i=2.2;
- Esiste un tipo **Unit** che viene usato per le funzioni che non tornano nulla

```
var age: Int = 25
age = 26 // value of age is changed from 25 to 26

val name: String = "John"
// name = "Jane" // Error: val cannot be reassigned

var x = 10 // type of x is inferred as Int
val y = "Hello" // type of y is inferred as String

val seq: CharSequence = "Jane" // this works because a String is also a CharSequence

val l: MutableList<String> = mutableListOf() // l is an empty mutable list
l.add(name)   // now l contains ["John"]
l.add("Jane") // now l contains ["John", "Jane"]
```

NOTE ULTERIORI

- **println**(i) → stampo valore della variabile i
- **val** equivale ad una costante ma lo Stringbuilder permette di fare operazioni particolari come l'append
- **val** s= StringBuilder("Hello"); s.append(" world"); println(s); -→ Hello world
- se dichiaro una variabile var al di fuori delle funzioni questa sarà **globale** e vivrà sempre.

Funzioni

```
fun test(i: Int): String {
    return ("test($i)");
}
println(test(42));
```

Funzione test a cui passo un intero e torna una stringa
Appende l'intero alla parola "test"

***chiamata** della funzione*

```
fun test(i: Int, l: Double =0.0): String {
    return ("test($i; $l)");
}
```

valore opzionale→ se non viene passato l, assume valore 0.0

NOTA: in fase di chiamata della funzione, posso specificare i nomi dei parametri a cui sto assegnando il valore

```
fun reformat(str: String,                                    // Mandatory
             normalizeCase: Boolean = true,                  // Optional
             wordSeparator: Char = ' '): String { … }        // Optional

reformat("Hello Kotlin!", wordSeparator = '_')
```

Posso **assegnare una funzione ad una variabile** (f) in base al valore di una variabile. Quindi, usando f, chiamo la funzione decisa nell'if `val f= if(i>5)::test2 else ::test;` `println(f(42, 1.1));`

NOTA: *assegnando un valore ad una variabile, perdo la possibilità di usare i valori opzionali*

\ *-slide 15, terzo punto*

Function literal: blocco fun anonimo, invocato tramite variabile

```
val sum = fun(a:Int, b:Int) { return a + b }
val result = sum( 5, 3 ) // 8
```

LAMBDA expression → `val l1 : (Int) -> Int = { n -> n+1 }` *val f= {i:Int, d:Double -> "Test3{$i*d}"};*

NOTA: Se ho un solo parametro per la lambda, posso non esprimere in modo esplicito il nome e il tipo del parametro e chiamarlo it `val l2 : (Int) -> Int = { it + 1 }`

```
fun combine(a: Int, b: Int, operation: (Int, Int) -> Int): Int {
    return operation(a, b)
}

val result = combine(5, 10) { x, y -> x + y }  // → 15
```

Posso specificare operazione in fase di chiamata

lambda function hanno la capacità di tirare fuori gli elementi

funzione che riceve un int, un double e una funzione e torna una stringa

- definizione
- chiamata

```
fun useStrategy(i:Int, d:Double, f: (i:Int, d:Double) -> String){
    println(f(i,d));
}    useStrategy(42, 3.5) {i:Int, d:Double -> "Test3 ${i*d}" }
```

Control statements

```
fun describeNumber(num: Int): String {
    return when (num) {
        0 -> "Zero"
        1 -> "One"
        2 -> "Two"
        else -> "Other" //mandatory!
    }
}

println(describeNumber(1)) // → "One"
println(describeNumber(7)) // → "Other"
```

```
fun describe(obj: Any): String =
  when {
    obj is String && obj.length < 10 ->
                    "Short string"
    obj is String && obj.length >= 10 -
>
                    "Long string"
    obj is List<*> ->
                    "List (${obj.size})"
    else -> "Other"
  }

println(describeLength("Hello"))
// → "Short string"
println(describeLength(listOf(1,2)))
// → "List (2)"
```

Loops and iterators

- The **for** statement only operates with ***iterators***, ***ranges***, or ***iterables***
  - The traditional, C-like, syntax has been removed

```
val iter = intArrayOf(1,2,4,8,16,32,64).iterator();

for (item in data)
  println(item)
```

```
for (i in 1..10)  // inclusive range
  println(i)
```

```
val array = arrayOf("a","b","c")
for ( (index, value) in array.withIndex() ) // iterable
  println( "array[${index}]: ${value}")
```

CLOSURE: funzioni in grado di catturare lo stato dell'environment (incluse variabile e oggetti)

- Estende ciclo di vita delle variabili → continuano ad esistere anche quando funzione ritorna

```
fun createGenerator(): () -> Int {
  var c = 0
  return { ++c }
}
val g1 = createGenerator()
val g2 = createGenerator()
```

```
println( g1() ) // 1
println( g1() ) // 2

println( g2() ) // 1
println( g2() ) // 2
```

USER DEFINED TYPES

- Classi
- Interface

NOTA: se non si specifica la classe padre → assegnato Any

Non ci sono getter e setter → tutto ciò che abbiamo sono **proprietà**

```
class Car{
    var brand: String = ""
        set(v) { if (v.length <3) throw RuntimeException("wrongValue")
            else field= v;
            println("Field was changed")}
}


fun main(){
    val c= Car();
    c.brand= "fiat"

}
```

Primary constructor: dichiarazione delle minime cose che devo fornire per creare un qualcosa → parametri della funzione necessari

- Devo passare determinati valori per creare
  (in questo caso var brand: String)

```
class Car(var brand: String){
}

fun main(){
    val c= Car("fiar");
}
```

```
class Car(eletric: Boolean){
    val engine: String
    init {
        if (eletric) engine= "Eletric"
        else engine="Thermic"
    }
}

fun main(){
    val c= Car(true);
    println(c.engine)
}
```

Se non specifico il tipo (var/val), quello che passo è un semplice parametro che sarà usato nell'init → body del costruttore

Constructor secondari:

- Introdotti da constructor
  - Delega il primo step al costruttore primario
  - Svolge altre azioni secondarie

```
class Car(eletric: Boolean){
    val engine: String
    constructor(): this(false){
        println("Secondary constructor was involved")
    }
    init {
        if (eletric) engine= "Eletric"
        else engine="Thermic"
    }
}
```

Es: viene usato nel caso in cui non si passa nulla alla funzione

Volendo posso avere un costruttore primario privato

```
class Car private constructor (eletric: Boolean){
    val engine: String
```

```
class Person constructor(val name: String) {

    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }

        constructor(name: String, parent: Person, age: Int): this(name) {
        parent.children.add(this)
        this.age=age;
    }

    var children: MutableList<Person> = mutableListOf<Person>();
    var age = 0;
}
```

Se serve fare inizializzazioni all'interno, devo creare un blocco init

```
init {
    engine = if (eletric) "Eletric" else "Thermic"
}
```

Metodi

```
class Car (eletric: Boolean){
    val engine: String
    fun doSomething(){
        println(this.engine)
    }

    constructor(): this(false){
```

```
fun main(){
    val c= Car(false);
    c.doSomething()
}
```

PROPRIETÀ

- **val** → read/write
- **var** → read/only
- getter e setter possono essere esplicitamente overriden se necessario aggiungendo altri get() o set(...) dopo la dichiarazione

```
class Circle(var radius: Double) {
    val area: Double
        get() = Math.PI * radius * radius
}


val c = Circle(2.0)  // Instantiates a circle with radius 2.0
c.radius *= 0.5      // modifies the radius property, setting it to 1.0

println(c.area)      // access the (computed) property area and prints 3.141592…
```

Vari tipo di **visibilità**:

- Se omessa → **public**
- **Private** → acceduta solo dentro lo stesso file/classe che l'ha definita
- **Protected** → classe o sottoclasse *(non fuori dalla gerarchia di classe)*
- **internal** → stesso modulo

```
class Document(_summary: String) {
    var summary: String = _summary
        set(value) {  field = value; hashCode = computeHash() }
    var hashCode: Int = computeHash()
        private set
    private fun computeHash(): Int { println("Hashing..."); return summary.hashCode() }
}

val d = Document("abc") // → Hashing…
d.summary = "qwerty" // → Hashing…
```

NOTA informale: per ogni classe max 7 attributi altrimenti impazzisco e non mi ricordo più niente

LATEINIT var : usati quando il valore della variabile non è disponibile quando costruisco l'oggetto

- Classi creati dai framework (activity, etc)
- Costruttore di queste classi può essere senza parametri

Perché c'è un momento in cui esistono ma nessuno ancora le conosce, sa a cosa servono
→ prima che siano completamente funzionali

Diverso da null in quanto sono non disponibili all'inizio ma quando lo diventano, non potranno essere null → perciò non posso dichiararle come nullabili → le dichiaro come lateinit

Es: La mia attività ha sicuramente un bottone ma non posso ancora inizializzarlo

```
class MyActivity: Activity(){
      private lateinit var button: Button //this will be initialized later

      override fun onCreate(bundle: Bundle?) {
          super.onCreate(bundle)
          button = Button(this)  //initialization occurs here
      }
}
```

DELEGATES

Le proprietà sono sofisticate in android, ogni istanza della classe car stora una stringa con il nome del brand.
Ci sono delle extra keyword che possono essere delegate → **mantenute da qualche altra parte.**

Comportamento completamente differente:

- variabile **lazy**: se store value è molto costoso da computing e nella maggior parte dei casi non è mai usato
    - funzione che accetta una lamda come parametro
        - non computa il tutto, fa solo se serve
        - la prima volta che provo ad accedere al contenuto della lazyValue, questa invoca la funzione lambda che riceve, stora il valore e poi mi da il valore

```
val lazyValue: String by lazy {
    print("computed!")
    "Hello"
}

fun main() {
    println(lazyValue) // → computed!Hello
    println(lazyValue) // → Hello
}
```

- **observable**: posso creare un qualcosa, devo fornire un valore iniziale e una lambda che sarà invocata automaticamente quando cambia il valore
    - reagisce facilmente al cambiamento
    - lambda ha 3 parametri:
        - proprietà
        - old value
        - new value

```
class User {
    var name: String by Delegates.observable("<no name>") {
        prop, old, new -> println("$old -> $new")
    }
}

fun main() {
    val user = User()
    user.name = "first"   // <no name> -> first
    user.name = "second"  // first -> second
}
```

- **vetoable**: lambda invocata prima del cambiamento del valore
    - per esempio, può bloccare la modifica di un valore
- **custom delegate**
    - posso creare i miei delegati che fanno ciò che voglio io

```
class Example {
  var p: String by MyCustomDelegate()
}

class MyCustomDelegate {
  operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
    // compute the value…
  }

  operator fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {
    // store the value…
  }
```

METODI: funzioni definiti con lo scope di una classe

- definiscono comportamento delle istanze della classe
- invocabili con **dot notation**→ someObject.someMethod(param1, param2)
- possono avere una rappresentazione short → fun sum(A:Int, b:Int) = a+b

OPERATOR **OVELOADING**

Ridefinire il comportamento di un operatore

```
class Vector2D(val x: Int, val y: Int) {
    operator fun plus(other: Vector2D): Vector2D {
        return Vector2D(x + other.x, y + other.y)
    }
}
val res = Vector(1,2) + Vector(2,3) // → Vector(3,5)
```

In questo caso cambia il significato dell'operatore +

COMPARISON

Equity

- **==** → valori
- **===** → stesso oggetto (pointer)
- Equality interagisce con la logica cìgenerale e con l'inheritance
- Si può implementare un **equals ad hoc**, facendo attenzione che rispetti le proprietà base
- Ogni classe che si crea deve essere in grado di essere comparata con Any

```
class Car (var eletric: Boolean){
    override fun equals(other: Any?): Boolean{
        if (other === this) return true
        if (other === null) return false
        if (!(other is Car)) return false
        val car = other as Car  //casto other come Car, posso vedere dentro
        return this.eletric==car.eletric

    }
}

fun main(){
    val c1= Car(true);
    val c2= Car(true);
    println(c1.equals(c2)) //oppure c1==c2 --> invoca equals
}
```

Nota: questa classe non è estendibile, se voglio estenderla, devo dichiararla open

```
open class Car (var eletric: Boolean){
```

→dichiaro come open quindi estendibile

```
class ElectricCar: Car(true){
    val brand="fsfs"
```

→ classe ElectricCar dichiarata come estensione della classe Car

```
open class Car (var eletric: Boolean){
    override fun equals(other: Any?): Boolean{
        if (other === this) return true
        if (other === null) return false
        if (this.javaClass != other.javaClass) return false
        val car = other as Car  //casto other come Car, posso vedere dentro
        return this.eletric==car.eletric

    }
```

```
class ElectricCar: Car(true){
    val brand="fsfs"
     override fun equals(other: Any?): Boolean{
        if (other === this) return true
        if (other === null) return false
        if (this.javaClass != other.javaClass) return false
        val car = other as ElectricCar
        return this.eletric==car.eletric && this.brand==car.brand
     }
```

Usare JavaClass è possibile se si usa Java ma non Javascript

HASHCODE: ridurre il contenuto di un oggetto ad un numero

- Implementare containers come hashMap o hashSet che beneficiano del fatto che possiamo **ridurre lo spazio di ricerca** per scoprire se un elemento è presente controllando direttamente l'hashcode.
- Implementare hashcode causa la responsabilità di garantire che ogni volta che due elementi sono uguali, lo devono essere anche gli hashcode *(non è detto il reverse)*

ORDER TESTING

I tipi primitive possono essere comparati usando i normali operatori di ordine che effettuano un ordinamento:

- Numerico per numeri
- Alfabetico per stringhe

Altri tipi possono essere comparati se implementano l'interfaccia Comparable:
se due object sono equals → comparTo deve ritornare 0

the `Comparable<T>` interface
- `operator fun compareTo(other: T): Int`

SINGLETON OBJECT
In alcune situazioni, vogliamo essere sicuri che per **una data classe esiste una sola istanza**.
Se non ha parametri e non mi interessa averlo Lazy, posso dichiarare come un **oggetto**.

```
object MySingleton {
  private var _counter = 0
  fun increment() { ++ _counter }
  val counter = _counter
}
```

```
MySingleton.increment()
println(MySingleton.counter) // → 1

MySingleton.increment()
println(MySingleton.counter) // → 2
```

In alcuni casi vogliamo che singleton sia Lazy: non posso semplicemente usare object invece che class

COMPANION OBJECT → definisce l'insieme di proprietà e metodi statici di una classe

- Creato come viene creata la classe

```
class Person(val name: String, var age: Int) {
    companion object {
        val defaultName = "Unknown"
    }
}
println(Person.defaultName) // → Unknown
```

EXTENSION FUNCTION → possono essere usate per aggiungere funzionalità ad una classe esistente senza modificare il codice della classe sorgente.

```
fun <T> List<T>.second(): T? {
    if (this.size<2) return null
    return this.get(1)
}

fun main() {
    val l= listOf("alfa", "beta", "gamma", "delta", "epsilon", "zeta")
    println(l.second())
}
```

```
beta
```

INHERITANCE
Ogni cosa deriva dalla classe Any se non nullable; dalla classe Any? Se nullable.
Di default, inheritance è disabilitata → una classe non può essere estesa se non lo dico esplicitamente dichiarandola come open.

Quando faccio comparazione, devo controllare se l'altro ha la mia stessa classe. Nel caso in cui lui sia una sottoclasse della mia, ho problemi perché restituirebbe vero nel controllo se è nella mia classe ma in realtà non siamo proprio nella stessa classe. (io sono un frutto generico, lui è una mela).
Questo problema si ha solo con le classi dichiarate open, perché nel caso opposto non puoi creare sottoclassi.

```
open class Base(val p: Int) {
  open fun method1(): Int = p

  fun method2(): String = "Cannot be overridden"
}

class Derived(p: Int) : Base(p) {
  override fun method1(): Int = p*2

  // override fun method2(): String = "Error" // method2 is final
                                              // and cannot be overridden
```

Classe derivata può rimpiazzare l'implementazione dei metodi dichiarati come open, usando la keyword ovverride. Se un metodo non è dichiarato come open, non si può ovveridare.

Ogni classe ha una e una sola superclasse e 0+ interfacce implementate:

- Superclassi e interfacce vanno specificate dopo il costruttore primario

```
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
}
```

DataClasses:
classe designata a **storare dati**

- Deve avere almeno (at least) 1 parametro
- Tutti i parametri dei costruttori primari devono essere marcati con val o var
- Non possono essere abstract, open, sealed, ineer

Forniscono molte **funzionalità**:

- Equals(...) basato sui valori effettivi e non su indirizzi
- hashCode()
- toString
- component1(), … , componentN()
- copy(...)

```
data class Person(val name:String, val Age: Int

fun main() {
   val d1 = Person("a", 32);
   val d2 = Person("a", 32);
   println(d1.equals(d2))  //torna true
}
```

```
data class Point( var x: Int, var y: Int)

val p1 = Point(1)
val p2 = Point(1)
println("Same point? ${p1 == p2}")

>>>> "Same point? true"

The new definition of Point as a data class automatically includes overrides for
methods, as equals, whose implementation depend on the properties of the class.For
example, the data class version of Point contains an equals method that is
equivalent to this:
override fun equals(o: Any?): Boolean {
  // If it's not a Point, return false. Note that null is not a Point
  if (o !is Point) return false
  return x == o.x && y == o.y
}
```

```
fun main() {
   val d1 = Person("a", 32);
   val (nome,anni) = d1
   println("name: $nome has age $anni")
}
```

```
name: a has age 32
```

Enum Class
Per definire un insieme di possibili valori
Fittano molto bene con le istruzioni when

```
fun getSuccessfulCode(code: HttpStatusCode): Int? = when (code) {
         HttpStatusCode.OK -> code.value
         HttpStatusCode.CREATED -> code.value
     HttpStatusCode.NOT_FOUND -> null
     HttpStatusCode.ACCESS_DENIED -> null
     HttpStatusCode.INTERNAL_SERVER_ERROR -> null
}
```

```
enum class GymActivity {
     BARRE, PILATES, YOGA, FLOOR, SPIN, WEIGHTS
}

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

Nota: se accio un when con un enum e successivamente aggiungo un elemento ad enum e non al when → compile error finchè non riallineo

SEALED CLASS: classi che possono essere estese solo da un'altra classe definita nello stesso file sorgente

- Open di default per le classi definite nel file sorgente corrente
- Può avere delle sottoclassi (Success, failure)

Una sealed class è un'unione dei domini delle sue sottoclassi. Se una sottoclasse non ha attributi, può essere rimpiazzata dai data object.

```
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()
}
```

```
sealed class Optional<out T>
data class Some<out T>(val data: T) : Optional<T>()
data object None : Optional<Nothing>()

var o1: Optional = Optional.Some(10)
println( o1 ) // → Some(data=10)
o1 = None
println( o1 ) // → None
```

GENERICS:

Paradigma di programmazione che permette di definire una classe o una funzione che può lavorare con diversi tipi di dato:

- Non cambia se ho numeri, stringhe o altro
- Functions, classes, interfaces fanno riferimento ai loro parametri/proprietà senza far riferimento al tipo specifico ma a <T>.

Due implementazioni:

- Qualche nome → come fa C
  - Compilatore genera una versione della mia classe generica, specializzata per il tipo che ho appena dichiarato
  - Se uso una lista di 3 diversi tipi, il compilatore genera 3 copie della funzione indipendenti tra loro
- **Type erasure** → cosa fa kotlin
  - Kotlin genera codice per Any?
    - Quando compilo posso prendere nota del tipo specifico per controllare validità del programma
    - Quando faccio runtime non posso far nulla a riguardo
      - Non riesco a capire se è una lista di numeri o char

```
fun <T> sort(l: List<T>) where T : CharSequence, T : Comparable<T> { … }

sort( listOf( "three", "two", "one" ) ) //ok
//String implements both Comparable<String> and CharSequence

// sort( listOf( 1, 2, 3 ) )
// Type mismatch: inferred type is IntegerLiteralType[Int,Long,Byte,Short]
// but CharSequence was expected
```

Posso specificare che voglio Char Sequence → funzione accetta solo charSequence

```
class Box<T>(private val content: T) {
    fun getContent(): T {
        return content
    }
}
```

```
val b1 = Box("hello")
val b2 = Box(12)
val s: String = b1.getContent() // "hello"
val i: Int = b2.getContent() // 12
// val b = b1 is Box<Int>
```

Tipo per classe.

Variance in classe generiche, volendo, si può specificare l'uso del tipo

- **In** → classe prende T come un input
- **Out** → classe sputa fuori T come un ouput

```
interface Source<out T> {
    fun next(): T
}
```

```
interface Sink<in T> {
    fun put(item: T)
}
```

Nota: se non specificato, può essere usato in entrambi i modi

SCOPED FUNCTION

Funzioni che eseguono un blocco di codice nel contesto dell'oggetto

- Let …
- Run …
- With

```
var name: String? = null
// …name is updated…

// let is used to prevent a
// NullPointerException from occurring
name?.let {
  println(it.reversed)
  println(it.length)
}
```

```
class Car{ var name:String = "",
           var maker:String = "" }

// apply is used to configure an object
val car = Car().apply {
  name = "Panda"
  maker = "Fiat"
}
```

```
val car = Car()
// configure car…

// with operates on a non-null object
with (car) {
    println(name)
    println(maker)
}
```

Eseguono un blocco di codice e modificano l'oggetto

- **Also**
- **Apply**

```
data class Person(val name:String, val Age: Int)

fun main() {
   val d1 = Person("a", 32).apply {println(this.Age)}
}
```

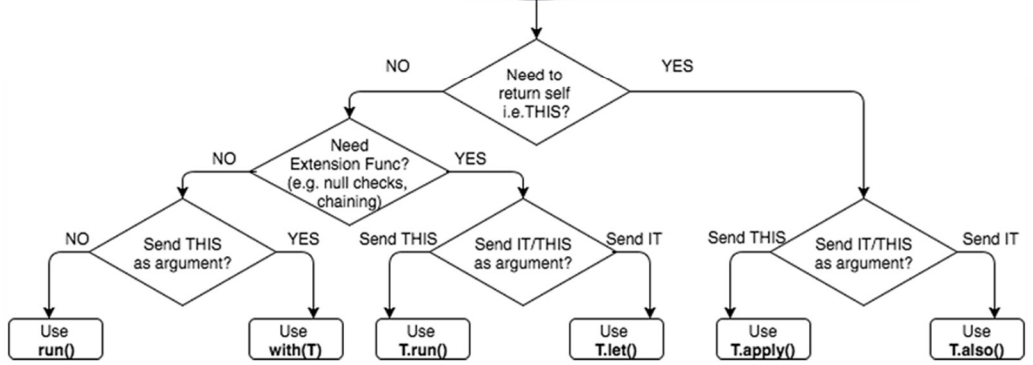
```
32
```

```
data class Person(val name:String, val Age: Int)

fun main() {
   val d2=3
   val d1 = Person("a", 32).run {println(this); d2}
   println(d1)
}
```

```
Person(name=a, Age=32)
3
```

```
data class Person(val name:String, val Age: Int)

fun main() {
   val d1 = Person("a", 32).also {p -> println(p)}
}
```

```
Person(name=a, Age=32)
```



Checking e casting →

In alcuni casi è necessario accedere al tipo del valore salvato in una variabile

- Operatore **is**/**!is** `if (v is String) { … }`

Se si vuole castare un valore ad un tipo differente si può usare l'operatore **as**

- Se possibile, trasforma nel cast richiesto; altrimenti throw an exception.

Se si vuole usare un modo più safe, si può usare **AS?:**

- ritorna un puntatore del tipo corretto se ok; altrimenti ritorna null

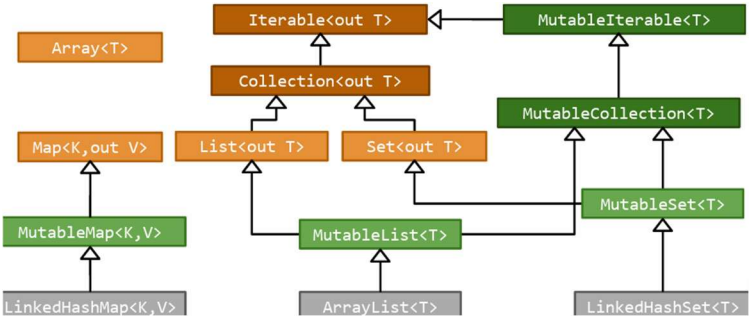
SMART CAST:
applicato solo in qualche situazione, casta automaticamente.
Solo se il compilatore può garantire che il campo non può essere mutato dopo il suo check

- Val local variables → sempre fatto
- Val properties → se la proprietà è provata o interna o il check è svolto nello stesso modulo in cui è dichiarata tale proprietà
- Val local variables → se la variabile non è modificata tra il check e l'uso, e non è catturata da una lambda che la modifichi
- Var properties → mai svolto

Collection library:

Si basa principalmente su 3 classi java:

- **Map**
- **List**
- **Set**



Top dela gerarchia → **Iterable** → fornisce gli elementi uno alla volta (valore successivo o null se finito)

- Può vedere il contenuto ma non modificarlo

Mutable iterable → può modificare l'oggetto su cui sto iterando

Collection → posso vedere più cose, dettagli (es: size)

Mutable collection → posso rimuovere/Aggiungere elementi alla collezione senza iterarci

Tipi di collection

- **List**: garantisce l'ordine, posso avere duplicati
  - Mutable se modificabile (può crescere o dimunuire)
- **Set** : non ho controllo sull'ordine degli elementi, non possono esistere duplicati
  - Mutable se modificabile (può crescere o dimunuire)

Map: associa chiave a valore

- Mutable se modificabile (può crescere o dimunuire)

CREAZIONE

- Immutabili
  - Per map → Pamela to 26
    - `listOf(values...)`, `setOf(values…)`, `mapOf(values…)`, `arrayOf(values…)`, `listOfNotNull(collection)`, `emptyList()`, `emptySet()`, `emptyMap()`
- Mutabili `mutableListOf(values...)`, `mutableSetOf(values…)`, `mutableMapOf(values…)`

Metodi di manipolazione → non cambiano l'originale ma ce ne **forniscono una copia** modificata

- Metodo per filtrare

```
fun main() {
    val l=listOf(9,8,7,6,5,4,3,2,1,0)
    println(l.filter{it>5}) //[9,8,7,6]
}
```

FILTER

`filter(predicate: (T) -> Boolean): List<T>`
- returns a list containing only elements matching the predicate
- `filterNot(...)` does the reverse
- `filterNotNull(...)` removes all of the nulls from a collection

```
fun main() {
    val l=listOf(9,8,7,6,5,4,3,2,1,0)
    println(l.find{it>5}) //9 (torna primo che soddisfa)
}
```

FIND

returns the first element matching the given predicate, or `null` if none is found

```
fun main() {
    val l=listOf(9,8,7,6,5,4,3,2,1,0)
    println(l.min())
}
```

```
0
```

```
fun main() {
    val l=listOf(9,8,7,6,5,4,3,2,1,0)
    println(l.max())
}
```

```
9
```

- **MIN**
- **MAX**

```
fun main() {
    val l=listOf(9,8,7,6,5,4,3,2,1,0)
    println(l.fold(0){a,b -> a+b})
} //0 = valore iniziale dell'accumulatore
```

```
45
```

- **FOLD** → valori intermedi andando avanti fino alla fine della collezione quando li avrò letti tutti

```
fun main() {
    val l=listOf(9,8,7,6,5,4,3,2,1,0)
    println(l.fold(10000){a,b -> a+b})
} //10000 = valore iniziale dell'accumulatore
```

```
10045
```

```
fun main() {
    val l=listOf(9,8,7,6,5,4,3,2,1,0)
    println(l.map{"s$it"})
}
```

```
[s9, s8, s7, s6, s5, s4, s3, s2, s1, s0]
```

MAP

- `map(transform: (T) -> R): List<R>`
  - Transforms all the elements of the collection by applying them the given function
  - It does not mutate the collection to which it is applied, but it returns a new list
- `mapIndexed(transform: (index: Int, T) -> R): List<R>`
  - The transforming function is supplied with both the element and the corresponding index
- `mapNotNull(transform: (T) -> R?): List<R>`
  - returns a list of all non-null outcomes obtained applying the supplied transform function to each element of the original collection

```
val doubles: List = listOf(1.0, 2.0, 3.0, null, 5.0)
val squares: List = doubles.mapNotNull { it?.pow(2) }
//squares = [1.0, 4.0, 9.0, 25.0]
```

FLAT MAP

- `flatMap(transform: (T) -> Iterable<R>): List<R>`
  - Applies a transformation function to each element of the original collection, and then flattens the resulting collections into a collection of plain values

```
val list: List> = listOf(listOf(1, 2, 3, 4), listOf(5, 6))
val flatList: List = list.flatMap { it.filter {v -> v%2 == 0} }
//       The variable flatList will have the value [ 2, 4, 6].
```

MAP:

- **Map**
- **mapIndexed**:
  - index → numero
  - element
- **mapNotNull** → applica una funziona, se la funziona ritorna null → elimina elemento

GROUP BY: opera sulla collezione, ritorna un Map

- **groupBy(keySelector: (T) -> K): Map<K, List<T>>**
  - groups elements having the same key into a list and returns a map where each key is associated to its list

```
val numbers = listOf(1, 20, 18, 37, 2)
val groupedNumbers = numbers.groupBy {
    when {
        it < 20 -> "less than 20"
        else -> "greater than or equal to 20"
    }
}
// The variable groupedNumbers now contains a Map>. The map has two keys,
“less than 20” whose value is [1, 18, 2], and “greater than or equal to 20”
whose value is [20, 37]
```

PARTITION → divide in maniera booleana → 2 sottogruppi

```
partition(predicate: (T) -> Boolean):
                        Pair<List<T>, List<T>>
```

METODI BOOLEANI:

- any: `any(predicate: (T) -> Boolean) : Boolean`
- all: `all(predicate: (T) -> Boolean): Boolean`
- none(...) `none(predicate: (T) -> Boolean): Boolean`
- none() → ritorna vero solo se la collezione è completamente vuota

```
none() : Boolean
```

```
val nums = listOf(10, 20, 100, 5)
val isAny = nums.any() // true

val isAnyOdd = nums.any { it % 2 > 0 } // true

val isAnyBig = nums.any { it > 1000} // false

val areAllEven = nums.all { it % 2 == 0 } // false

val isNone = nums.none() // false

val isNone4 = nums.none { it == 4 } // true
```

SEQUENCE

Container iterabile che opera Lazy, potenzialmente infiniti elementi

- quando una sequence va sotto un processo multi-step, gli elementi sono estratti uno alla volta

CREARE SEQUENZA:

- Da elementi
- Da ogni iterable
- Da una funzione
- Da chunk

```
val numbers = sequenceOf( "three",  "two", "one" )

    val numbers = listOf("three", "two", "one").asSequence()

    val numbers = generateSequence { try { readLine().toInt() }
                                     catch (e:Exception) { null } }

val primes = sequence {
      yield(1)
      yieldAll( listOf(2,3,5) )
}
```

→ lavora solo quando arriva a toList(). Finchè non capisce che serve non fa niente.

```
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()  //consumer
```