



Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

Null Safety & il tipo Optional

Corso di Laurea in Ingegneria Informatica
Anno accademico 2021/2022

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

RAPPRESENTARE L'ASSENZA

- Spesso ci si trova di fronte a situazioni in cui occorre *rappresentare l'assenza di un oggetto*
 - risultati non disponibili (es. ricerche che non hanno dato esito)
 - risultati impossibili (es. nelle matrici, **extractMinor** con indici di riga/colonna «esterni» o in presenza di matrice non quadrata)
 - argomenti «opzionali» che l'utente potrebbe fornire o non fornire
- Come gestire *opportunamente* tutto ciò?
- Un classico modo di farlo è *passare/restituire null*
 - se non c'è risultato (es. "elemento non trovato" in una ricerca)
 - se quel certo argomento in quel certo caso "non c'è" o "non serve"
- ma è un approccio *naif* che *inietta fragilità* nel software
 - in effetti, è il perfetto generatore di NullPointerException (NPE)

PERCHÉ NON null?

- Lasciamolo dire direttamente a Tony Hoare:



I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years. [wikipedia]

PERCHÉ NON `null`?

- Perché `null` è inopportuno?
 - se una funzione può restituire `null`, il cliente è *obbligato a verificare sempre il risultato* prima di poterlo usare (altrimenti, NPE)
 - quindi, il codice del chiamante si riempie di `if(... != null)` e diventa illeggibile: la *business logic* si perde in mezzo ai controlli
 - *alla minima svista esplode tutto e lo fa nel posto sbagliato*: le NPE sono *un incubo*, rendono il debugging faticoso e time-consuming
 - last but not least, *non funziona coi valori primitivi*: non essendo oggetti, non hanno un "null" (per i soli reali, al limite, c'è `NaN`...)
- Non a caso i linguaggi più moderni cercano di *evitare o comunque circoscrivere/disciplinare* l'uso di `null`
- Nuovo obiettivo: *NULL SAFETY*

RAPPRESENTARE L'ASSENZA DI VALORE NEI REALI

- Le due classi di libreria `Float` e `Double` fungono (anche) da "librerie di supporto" per i tipi `float` e `double`, rispettivamente
- Fra le altre cose, definiscono *tre costanti pubbliche*

NaN	A constant holding a Not-a-Number (NaN) value of type float.
NEGATIVE_INFINITY	A constant holding the negative infinity of type float.
POSITIVE_INFINITY	A constant holding the positive infinity of type float.

Java	~C#
Kotlin	~Scala

- NaN** (Not A Number) rappresenta *l'assenza del valore atteso*
 - è il valore restituito dalle *forme indeterminate* ($\infty - \infty$ o ∞ / ∞)
 - può ben essere sfruttato anche per *rappresentare l'assenza di risultato* in funzioni che restituiscono `float` o `double`
 - le due librerie `Float` e `Double` offrono la *funzione statica* `isNaN` per verificare se un dato valore reale sia "assente"

NUMERI REALI: NaN e ∞

```
double nan = Double.NaN;
double plusInf = Double.POSITIVE_INFINITY;
double minusInf = Double.NEGATIVE_INFINITY;
```

Java

```
double diffPlusInf = plusInf - plusInf;
System.out.println(diffPlusInf); // NaN
System.out.println(diffPlusInf==nan); // false: NaN non è mai uguale a sé stesso!
System.out.println(Double.isNaN(diffPlusInf)); // true
System.out.println(Double.isNaN(nan)); // true
//-----
System.out.println(plusInf + 3.0); // Infinity
System.out.println(minusInf + 3.0); // -Infinity
//System.out.println(7/0); // ESPLODE fra interi
System.out.println(7.0/0); // Infinity
System.out.println(Double.isInfinite(-7/0.0)); // true
```

```
NaN
false
true
true
Infinity
-Infinity
Infinity
true
```

```
val nan = Double.NaN;
val plusInf = Double.PositiveInfinity;
val minusInf = Double.NegativeInfinity;

val diffPlusInf = plusInf - plusInf;
println(diffPlusInf); // NaN
println(diffPlusInf==nan); // false perché
println(diffPlusInf.isNaN()); // true
println(nan.isNaN()); // true
//-----
println(plusInf + 3.0); // Infinity
println(minusInf + 3.0); // -Infinity
// println(7/0); // ESPLODE fra
println(7.0/0); // Infinity
println((-7/0.0).isInfinite()); // true
```

```
NaN
false
true
true
Infinity
-Infinity
Infinity
true
```

Scala

```
double nan = Double.NaN;
double plusInf = Double.PositiveInfinity;
double minusInf = Double.NegativeInfinity;

double diffPlusInf = plusInf - plusInf;
Console.WriteLine(diffPlusInf); // NaN
Console.WriteLine(diffPlusInf==nan); // false
Console.WriteLine(Double.IsNaN(diffPlusInf)); // true
Console.WriteLine(Double.IsNaN(nan)); // true
//-----
Console.WriteLine(plusInf + 3.0); // Infinity
Console.WriteLine(minusInf + 3.0); // -Infinity
//Console.WriteLine(7/0); // ESPLODE fra interi
Console.WriteLine(7.0/0); // Infinity
Console.WriteLine(Double.IsInfinite(-7/0.0)); // true
```

```
NaN
False
True
True
Infinity
-Infinity
Infinity
True
```

C#

```
fun main() {
    val nan = Double.NaN;
    val plusInf = Double.POSITIVE_INFINITY;
    val minusInf = Double.NEGATIVE_INFINITY;
    val diffPlusInf = plusInf - plusInf;

    println(diffPlusInf); // NaN
    println(diffPlusInf==nan); // false: NaN non è mai uguale a sé stesso!
    println(diffPlusInf.isNaN()); // true
    println(nan.isNaN()); // true
    //-----
    println(plusInf + 3.0); // Infinity
    println(minusInf + 3.0); // -Infinity
    //println(7/0); // ESPLODE fra interi
    println(7.0/0); // Infinity
    println((-7/0.0).isInfinite()); // true
}
```

Kotlin

```
NaN
false
true
true
Infinity
-Infinity
Infinity
true
```

RAPPRESENTARE L'ASSENZA DI VALORE NEI REALI: NaN

- Quindi, nei reali, se non si è potuto calcolare il risultato, ha senso restituire **NaN**
 - Esempio: nelle matrici, se la matrice non è quadrata il determinante non esiste, quindi ha perfettamente senso restituire **NaN**:

```
public double det() {  
    return isSquared() ? calcDet() : Double.NaN;  
}
```

Java

C#

~Kotlin

~Scala

- Punto chiave: il cliente che riceve **NaN** *non esplode*, perché riceve comunque "un risultato" (non "il nulla")
 - ovviamente, **NaN** si propaga nelle operazioni successive, che daranno quindi tutte come risultato **NaN**...
 - ...ma questo è logico: se manca un valore, le operazioni successive perdono di senso, non possono "far finta" di averlo!



RAPPRESENTARE L'ASSENZA DI VALORE NEI REALI: NaN

- Ovviamente, di conseguenza, *anche i test* devono considerare i casi in cui il risultato potrebbe essere "assente"

```
@Test
public void testDetNaN() {
    Matrix m = new Matrix(new double[][] { { 1, 0, 0, 1 }, { 0, 1, 0, 1 }, { 0, 0, 1, 1 } });
    assertTrue(Double.isNaN(m.det()));
}
```

Java

- Ma.. per tutti gli altri oggetti, come possiamo fare, se non vogliamo usare `null`...?



RAPPRESENTARE L'ASSENZA DI OGGETTI

- Assodato che passare/restituire `null` *inietta fragilità* indesiderate nel software
 - e causa *`NullPointerException`* difficili da debuggare
- prendiamo atto che, come già nei reali con NaN, *l'assenza va sempre esplicitamente rappresentata*
 - non va *elusa* o "nascosta dietro un `null`", anzi:
si deve vedere quando e dove un oggetto possa mancare
 - come con lo 0 in matematica, serve un *modo esplicito* per *denotare l'assenza* di elementi: non basta "non scrivere che ci sono"
 - con le rappresentazioni implicite *non si esprimono concetti né vincoli, non si computa*: la realtà va sempre *esplicitamente rappresentata*

OBIETTIVO: NULL SAFETY

- Linguaggi diversi adottano approcci diversi
 - Java, Scala **agiscono a livello di libreria**, introducendo il concetto di tipo **Optional** (in Scala: `Option`)
 - Kotlin: **agisce a livello di linguaggio**, ritenendo i normali tipi (classi, interfacce) a priori «*non-nullable*»
 - viene vietato l'uso di `null` negli assegnamenti
 - sono vietate chiamate di metodi su oggetti *potenzialmente nulli*
 - si introducono *operatori speciali* «*appesantiti*» per le (rare) situazioni in cui si voglia consentirlo, in deroga al principio generale
 - nascono **specifici tipi «nullable»** a complemento dei tipi «non nullable»
 - C#: da C# 8.0, sostanzialmente come Kotlin
 - prima, occorreva costruirsi a mano un analogo all'`Optional` di Java



IL TIPO “OPZIONALE”

- NUOVO APPROCCIO: esprimere esplicitamente l'idea che un oggetto *possa mancare, ovvero sia opzionale*
 - NON manipolare direttamente un oggetto *che possa essere null*
 - MA incapsularlo in un adapter **Optional<T>** (Scala: **Option[T]**) che in quanto tale non sia mai null
 - il **null**, nel caso, è *dentro* all'adapter
- SINTASSI: **Optional<T>** **Java** / **Option[T]** **Scala**
 - il "vero" oggetto che ci interessa, di tipo **T** (eventualmente **null**) è incapsulato in un oggetto-wrapper ausiliario *optional*
 - l'oggetto-wrapper **Optional** *esiste sempre: non è mai null*
 - quindi, può essere usato, passato, fatto circolare, testato... *senza rischio che spari una NPE a tradimento ☺*



IL TIPO `Optional<T>` in JAVA

- **`Optional<T>`** incapsula un possibile valore di tipo **`T`** Java
 - per incapsulare un valore: **`Optional.of`**
 - per esprimere il "nessun valore": **`Optional.empty`**
 - per accertare la presenza/assenza: **`isPresent` / `isEmpty`**
 - per estrarre il valore (se presente): **`get` / `orElse`**
- Vantaggi
 - **PUNTO CHIAVE:** si esplicita nella signature della funzione che un risultato possa non esserci o che un argomento sia facoltativo
 - **si elimina così la sorgente di fragilità** dovuta al *rischio che un oggetto inaspettatamente non esista* e conseguente imprevista NPE
 - i clienti sanno di avere tra le mani *solo oggetti che esistono sempre*
 - solo per accedere al contenuto essi dovranno verificare preventivamente che l'oggetto opzionale "ci sia" (metodo **`isPresent`**)



IL TIPO `Option[T]` in SCALA

- `Option[T]` incapsula un possibile valore di tipo `T` Scala
 - è una classe astratta con due sottoclassi concrete: `Some` e `None`
 - per incapsulare un valore: `Some(valore)`
 - per esprimere il "nessun valore": `None`
 - o, con un comodo factory method: `Option(valoreOrNull)`
 - per accertare la presenza del valore: `isDefined / isEmpty`
 - per estrarre il valore (se presente): `get / getOrElse`
- Impostazione analoga
 - tecnicamente, un po' più verbosa per l'uso di `Some/None`, ma mitigata dal factory method `Option(...)` che ne limita l'uso diretto
 - impianto generali e metodi disponibili molto simili

ESEMPIO

Optional nel compito BikeRent

Nel compito BikeRent del 12/9/2018

Java

- la classe **Rate** rappresenta una *tariffa*, caratterizzata da nome della città, costo e durata del primo periodo e dei periodi successivi, **eventuale durata max e orario di rientro limite**, ammontare della sanzione per violazioni
- una città potrebbe *non prevedere alcuna durata max e/o alcun orario limite di rientro*: per questo sono **Optional<Duration>** e **Optional<LocalTime>**
- il costruttore prevede quindi due argomenti **Optional**
- gli accessor corrispondenti restituiscono quindi anch'essi due **Optional**

Rate	
-	citta: String
-	durataMax: Optional<Duration>
-	orarioMax: Optional<LocalTime>
-	primo: Periodo
-	sanzione: double
-	successivi: Periodo
<hr/>	
+	getCitta(): String
+	getCostoPeriodiSuccessivi(): double
+	getCostoPrimoPeriodo(): double
+	getDurataMax(): Optional<Duration>
+	getDurataPeriodiSuccessivi(): Duration
+	getDurataPrimoPeriodo(): Duration
+	getOrarioMax(): Optional<LocalTime>
+	getSanzione(): double
+	Rate(String, Periodo, Periodo, Optional<Duration>, Optional<LocalTime>, double)
+	toString(): String



Optional PER TIPI PRIMITIVI

- `Optional<T>` incapsula un possibile valore di tipo `T`
MA il tipo `T` dev'essere una classe o un'interfaccia
- Quindi, *in Java non può essere un tipo primitivo*, perché i tipi primitivi non sono oggetti (sigh...)
- Per ovviare, sono state definite **tre classi ausiliarie** Java
`OptionalInt`, `OptionalLong`, `OptionalDouble`
per i tre tipi primitivi più usati, `int`, `long`, `double`
 - i corrispondenti metodi per recuperare il valore primitivo interno si chiamano `getAsInt`, `getAsLong`, `getAsDouble`
- In Scala e Kotlin, dove non esistono tipi primitivi perché *everything is an object*, il problema ovviamente non si pone

ESEMPIO

Optional nel compito CambiaValute

Nel compito CambiaValute del 12/1/2016:

Java

- la classe `CambiaValute` rappresenta un'agenzia cambiavalute
- i due metodi `acquisto` e `vendita` calcolano l'equivalente di un importo da/verso **una valuta estera specificata come stringa (USD, GBP, etc.)**
- poiché un'agenzia non è ovviamente tenuta a trattare tutte le valute del mondo, il risultato dei metodi `acquisto` e `vendita` è un **OptionalDouble**

se l'agenzia *non tratta quella specifica valuta*, il risultato ovviamente non esiste.

CambiaValute	
+ acquisto(siglaValutaEsteri :String, importo :double) :	OptionalDouble
+ arrotonda(importo :double) :	double
+ CambiaValute(nomeAgenzia :String, elencoValute :Map<String,TassiDiCambio>)	
+ convertiInDouble(importo :String) :	double
+ formatta(importo :double) :	String
+ getElencoValute() :	Map<String,TassiDiCambio>
+ getNomeAgenzia() :	String
+ toString() :	String
+ vendita(siglaValutaEsteri :String, importo :double) :	OptionalDouble



Optional: ESEMPIO 1

```
OptionalDouble aliquota = OptionalDouble.of(8.6);
```

Java

```
if (aliquota.isPresent()) System.out.println(aliquota.getAsDouble());  
else System.out.println("aliquota indefinita");
```

```
OptionalDouble detrazione = OptionalDouble.empty();
```

```
if (detrazione.isEmpty()) System.out.println("nessuna detrazione");  
else System.out.println(detrazione.getAsDouble());
```

In Java, i tre tipi primitivi più usati (int, long, double) hanno il loro wrapper Optional specializzato

8.6
nessuna detrazione

```
val aliquota : Option[Double] = Some(8.6); // o anche: Option(8.6)
```

```
if (aliquota.isDefined) println(aliquota.get);  
else println("aliquota indefinita");
```

Factory method
di comodità

Scala

```
val detrazione : Option[Double] = None;
```

```
if (detrazione.isEmpty) println("nessuna detrazione");  
else println(detrazione.get);
```

8.6
nessuna detrazione



Optional: ESEMPIO 2

In Java, i tre tipi primitivi più usati (int, long, double) hanno il loro wrapper Optional specializzato

```
OptionalDouble aliquota = OptionalDouble.of(8.6);  
System.out.println(aliquota.orElse(10.6));  
  
OptionalDouble aliquota2 = OptionalDouble.empty();  
System.out.println(aliquota2.orElse(10.6));
```

Java

```
8.6  
10.6
```

In Scala, non esistono tipi primitivi: si usa il wrapper Option generico

Factory method di comodità

```
val aliquota : Option[Double] = Some(8.6); // anche: Option(8.6)  
println(aliquota.orElse(Some(10.6))); // anche: Option(10.6)  
println(aliquota.orElse(Some(10.6)).get);  
  
val aliquota2 : Option[Double] = None;  
println(aliquota2.orElse(Some(10.6))); // anche: Option(10.6)  
println(aliquota2.orElse(Some(10.6)).get);
```

Scala

```
Some(8.6)  
8.6  
Some(10.6)  
10.6
```



Optional: ESEMPIO 3

```
String nome = null;
```

```
Optional<String> optString1 =  
    (nome==null) ? Optional.empty() : Optional.of(nome);
```

```
Optional<String> optString2 = Optional.ofNullable(nome);
```

```
System.out.println(optString1);
```

```
System.out.println(optString2);
```

Java

Due forme
equivalenti

Optional.empty
Optional.empty

```
val nome : String = null;
```

```
val optString1 : Option[String] =  
    if(nome==null) None else Some(nome);
```

```
val optString2 : Option[String] = Option(nome);
```

```
println(optString1);
```

```
println(optString2);
```

Scala

Qui il factory method è molto
comodo: equivale a `ofNullable`

None
None



Optional FOREVER...?

- NO! `Optional` non va usato «solo perché esiste»
- Ci sono **due casi d'uso tipici**:
 - **incapsulare un `null` di ritorno**, per restituire un *esito lecito* di assenza di risultato
 - **passare argomenti (realmente) opzionali** a una funzione
- NON è invece inteso per
 - **incapsulare un `null` dovuto a *violazione di precondizioni***
 - scenario tipico: clienti che non passano argomenti che avrebbero dovuto passare (la soluzione giusta sarà *lanciare eccezione*)
 - ***evitare di validare l'input***: che un utente sbagli a immettere un dato è prevedibile, ma non significa che quel dato sia «opzionale»!
(e neppure che siamo in presenza di una *situazione eccezionale*)



OPTIONAL: USO PRATICO

- Come mostrato negli esempi precedenti, esistono anche **metodi di utilità che catturano casi d'uso tipici**, evitando noiose trafile di **if/else**
 - in Java, **ofNullable** è una comoda variante di **of**, che incorpora la creazione di **empty** se l'elemento è **null**
 - in Scala, il factory method **Option(...)** svolge la stessa funzione
 - in entrambi, **orElse** è una comoda variante di **get** che specifica un elemento T alternativo, da restituire se l'optional è vuoto
- Altri metodi sono pensati per favorire l'integrazione con le lambda expression (che vedremo presto..):
 - in entrambi, **filter** accetta una lambda che restituisce un boolean
 - in Java, **ifPresent** accetta una lambda che consuma un valore
 - in entrambi, **orElseGet** accetta una lambda che fornisce un valore



Optional: DETTAGLI

- In particolare:

- In Java, **ofNullable(...)** è una comoda variante di **of**, che incorpora la creazione di **empty** se l'elemento è **null**

Java

`Optional.ofNullable(x)`

equivale a

`(x==null) ? Optional.empty() : Optional.of(x);`

- In Scala, **Option(...)** fa la stessa cosa

Scala

`Option(x)`

equivale a

`if(x==null) None else Some(x);`



Optional: DETTAGLI

- In particolare:

- In Java, **orElse** è una comoda variante di **get** che incorpora un elemento T alternativo, da restituire se l'optional è vuoto (ossia incorpora **null**)

Java

```
x.orElse(y)
```

equivale a

```
x.isPresent() ? x.get() : y
```

- In Scala, **orElse** fa la stessa cosa, ma maneggia dei **Some (...)** anziché direttamente i valori contenuti

Scala

```
x.orElse(y)
```

equivale a

```
if(x.isDefined) x.get else y
```

Optional: ESEMPIO 4

```
String s1 = null;
String s2 = "ciao";

Optional<String> opt1 = Optional.ofNullable(s1);
Optional<String> opt2 = Optional.ofNullable(s2);

String value1 = opt1.orElse("boh");
String value2 = opt2.orElse("buh");

System.out.println(value1);
System.out.println(value2);
```

Java

opt1 è vuoto → scatta l'else
opt2 è presente → si estrae

boh
ciao

```
val s1 : String = null;
val s2 : String = "ciao";

val opt1 : Option[String] = None
val opt2 : Option[String] = Some(s2);
val opt0 : Option[String] = Some(s1);

val value1 = opt1.orElse(Some("boh")); println(value1);
val value2 = opt2.orElse(Some("buh")); println(value2);
val value0 = opt0.orElse(Some("void")); println(value0);
```

Scala

Pessima pratica! Non si dovrebbe mai usare **Some** per incapsulare un null! Con **Option(s1)** si evita

Some(boh)
Some(ciao)
Some(null)



NULL SAFETY IN KOTLIN e C# 8.0

- In Kotlin, la null safety è *embedded nel linguaggio*
 - non c'è quindi alcun tipo *Optional* offerto come libreria
- Innanzitutto, il linguaggio distingue fra:
 - tipi *non-nullable* (quelli «standard»), che non possono essere *null*
ESEMPLI: `String`, `Int`, `Counter`, etc.
 - tipi *nullable*, che possono essere *null* ma *devono essere denotati esplicitamente come tali tramite il suffisso ?*
ESEMPLI: `String?`, `Int?`, `Counter?`, etc.

```
// val s1 : String = null; // rejected, cannot be null
val s1 : String? = null;   // OK, uses a nullable type
val s2 : String = "ciao";  // OK, is not null
```

Kotlin



NULL SAFETY IN KOTLIN

- Sui tipi *nullable* non è possibile effettuare chiamate o selezioni di campi con la classica dot notation .
- Si può agire su di essi solo:
 - tramite *safe calls* → operatore *?.*
 - tramite *non-null asserted call* → operatore *!!.*
- Una *safe call* effettua la chiamata *solo* se il riferimento non è nullo: altrimenti, restituisce null *senza chiamare alcunché*
 - di fatto, incorpora il check `if (... != null)`, evitando così le NPE
- Una *non-null asserted call* effettua *comunque* la chiamata, tacitando il sistema di null safety → rischio di NPE
 - approccio «for the NPE lovers»: se a run-time il riferimento è null, esplode tutto con NPE

NULL SAFETY IN KOTLIN

- al posto del metodo `orElse` degli `Optional`, Kotlin introduce un operatore detto, per la sua forma, *Elvis Operator* `?:`:



ESEMPIO

```
// val s1 : String = null; // rejected
val s1 : String? = null; // OK with nullable type
val s2 : String = "ciao";

println(s1); // it's null, prints "null"
// println(s1.length) // rejected, because it's unsafe
println(s1?.length) // OK, safe call --> it's null, prints "null"
println(s1?.length ?: -1) // OK, safe call + Elvis operator for alternative --> prints -1
println(s1!!.length) // non-null asserted call --> NPE at runtime

println(s2); // it's "ciao", prints "ciao"
println(s2.length) // accepted, because it's safe --> prints 4
println(s2?.length) // OK, (useless) safe call --> prints 4
println(s2?.length ?: -1) // OK, (useless) safe call + Elvis operator for alternative --> prints 4
println(s2!!.length) // non-null asserted call --> possible NPE at runtime, but here prints 4
```

Kotlin



NULL SAFETY IN C# 8.0

- Stessa sintassi di Kotlin per i nullable types *Tipo?* C#
- Come in Kotlin, *si può agire sui tipi nullable* solo:
 - tramite *safe calls* → operatore *?.* (= Kotlin)
 - tramite *non-null asserted call* → operatore *!.* (≠ Kotlin)
 - tramite l'equivalente dell'*Elvis operator* → operatore *??* (≠ Kotlin)
- La direttiva **#nullable** controlla il comportamento del compilatore in presenza di nullable types:

- `#nullable enable`: Imposta il contesto di annotazione nullable e il contesto di avviso Nullable su Enabled.
- `#nullable disable`: Imposta il contesto di annotazione nullable e il contesto di avviso Nullable su disabled.
- `#nullable restore`: Ripristina il contesto di annotazione nullable e il contesto di avviso Nullable nelle impostazioni del progetto.
- `#nullable disable warnings`: Impostare il contesto di avviso Nullable su disabled.
- `#nullable enable warnings`: Impostare il contesto di avviso Nullable su Enabled.
- `#nullable restore warnings`: Ripristina il contesto di avviso Nullable nelle impostazioni del progetto.
- `#nullable disable annotations`: Impostare il contesto di annotazione Nullable su disabled.
- `#nullable enable annotations`: Impostare il contesto di annotazione Nullable su Enabled.
- `#nullable restore annotations`: Ripristina il contesto di avviso delle annotazioni nelle impostazioni del progetto.

NULL SAFETY IN C# 8.0

ESEMPIO

```
#nullable enable // not mandatory but strongly suggested
```

Direttiva (non obbligatoria)

C#

```
// val s1 : String = null; // rejected  
string? s1 = null; // OK with nullable type  
string s2 = "ciao";
```

```
Console.WriteLine(s1);           // it's null, prints "null"  
// println(s1.Length);          // rejected, because it's unsafe  
Console.WriteLine(s1?.Length);    // OK, safe call --> it's null, prints empty line  
Console.WriteLine(s1?.Length ?? -1); // OK, safe call + ?? operator for alternative --> prints -1  
// Console.WriteLine(s1!.Length); // non-null asserted call --> NPE at runtime  
  
Console.WriteLine(s2);           // it's "ciao", prints "ciao"  
Console.WriteLine(s2.Length);     // accepted, because it's safe --> prints 4  
Console.WriteLine(s2?.Length);    // OK, (useless) safe call --> prints 4  
Console.WriteLine(s2?.Length ?? -1); // OK, (useless) safe call + ?? operator for alternative --> prints 4  
Console.WriteLine(s2!.Length);    // non-null asserted call --> possible NPE at runtime, but here prints 4
```

Se il riferimento è nullo, C# stampa una riga vuota anziché la stringa «null»

```
-1  
ciao  
4  
4  
4  
4
```

Operatori per nullable types:
?. ?? !?