**Corso**

**Dart**

**Dedalo’s notes – Una serie di appunti da non prendere sul serio**

Serie di appunti sugli argomenti più svariati, fatti per imparare, apprendere e divertirsi, navigando per il magico mondo del…tutto!

Ogni errore o imprecisione sarà sistemata, oppure no 😊.

Attenzione, può contenere boiate, amenità o stramberie!

https://github.com/GiacomoBorsellino

Scritto da

**Giacomo Borsellino**

27-12-2024

**Cosa è?**

Dart è un linguaggio di programmazione **ad alto livello**, progettato per essere semplice e versatile. Le sue caratteristiche principali includono:

* **Tipizzazione forte e opzionale**: Dart supporta un sistema di tipi statici, ma consente anche di lavorare con tipi dinamici.
* **Orientato agli oggetti (OOP)**: Tutto in Dart è un oggetto, inclusi numeri, funzioni e null.
* **Compilazione mista**:
  + **Ahead-of-Time (AOT)**: Per creare applicazioni native ad alte prestazioni.
  + **Just-in-Time (JIT)**: Per velocizzare lo sviluppo con funzionalità come il **hot reload**.
* **Garbage collection**: Gestione automatica della memoria.
* **Multipiattaforma**: Ideale per sviluppare applicazioni per mobile, web e desktop.
* **Librerie standard e asincronicità**: Include strumenti avanzati per la gestione asincrona tramite Future e Stream.
* **Sintassi moderna**: Semplice e leggibile, simile a JavaScript, C# e Java.

Dart è un linguaggio **compilato**, ma può essere interpretato durante lo sviluppo. È progettato principalmente per lo sviluppo front-end, con particolare enfasi sul framework **Flutter**, per creare interfacce utente multipiattaforma.

Immagine che contiene Policromia, Elementi grafici, design

Descrizione generata automaticamente

Figura 1 Link al corso: https://www.youtube.com/watch?v=7hzlhtuIgGE&list=PLP5MAKLy8lP9VP6KL4Q9ClYCuGhet59Od&index=1

Immagine che contiene illustrazione, arte

Descrizione generata automaticamente con attendibilità mediaCapitolo 1 - Installazione

1. Installa **powershell** (se non già presente)
2. Installare **Chocolatey\*** (<https://chocolatey.org/install#individual>)
3. Installa sdk dart (**choco install dart-sdk** **\*\***)

**\*** **Chocolatey** è un gestore di pacchetti, al pari di apt su linux

**\*\*** Un **SDK** (Software Development Kit) è un insieme di strumenti, librerie, documentazione e esempi forniti da un fornitore per aiutare gli sviluppatori a creare applicazioni o integrare servizi specifici all'interno di un determinato ecosistema o piattaforma.

Capitolo 2 - Introduzione

Immagine che contiene porta, illustrazione, design

Descrizione generata automaticamente

**Per creare un nuovo progetto**

1. Apri Powershell

**dart create -h** (vedi tutti i template disponibili per creare un primo mprogetto in dart)

**dart create -t console-full corso\_dart** (creare un primo progetto base)

**Nota**: se non viene trovato il percorso per dart, controlla sotto C in tools

1. Apri con vsCode il nuovo progetto, struttura

File/ Cartelle:

* pubspec.yaml, pubspec.lock, .dart\_tool servono a importare/gestire i pacchetti

core di dart

Per installare i pacchetti basta andare su [**https://pub.dev/**](https://pub.dev/)

* analysis\_lint: gestione del linting
* test: cartella di testing
* bin: core del codice
* lib: cartella con funziona di riuso

1. Avvia da console con **dart run**

**Estensioni** utili per **VSCode**:

* Dart
* Dart Data Class Generator
* Pubspec Assist

Capitolo 3 - Variabili

Immagine che contiene vestiti, disegno, persona, calzature

Descrizione generata automaticamente

Definizione e gestione delle variabili

Dart è un linguaggio fortemente tipizzato

È possibile dichiarare e poi definire una variabile, specificando il tipo all’inizio della dichiarazione.

Es.

Immagine che contiene testo, Carattere, schermata, numero

Descrizione generata automaticamente

È possibile usare la keyword **dynamic**, che permette di definire ciò che si vuole

Es.

Immagine che contiene testo, Carattere, schermata, numero

Descrizione generata automaticamente

È possibile usare anche **var**, questo, se non definito fin da subito (es. var test = 5), funzionerà come dynamic,



differentemente il suo tipo verrà bloccato fin da subito!

Immagine che contiene testo, Carattere, schermata, numero

Descrizione generata automaticamente

È possibile usare anche **final**, che indica che un valore è nel suo stadio finale, non posso più cambiarne tipo né valore.

Valori immutabili che sono determinati a tempo di **runtime**





È possibile usare anche **const**, che però deve essere dichiarato e definiti contestualmente

Valori immutabili che sono determinati a tempo di **compilazione**



Al pari con le **liste**, non è possibile modificare liste con const, si con final, anche se, anche in questo caso non posso riassegnare!

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Gerarchia della **flessibilità** delle keyword:

**dynamic > var > final > const**

**NOTA**: E’ possibile verificare se c’è un errore in fase di runtime usando l’attributo runtimeType, che restituisce il tipo effettivo della variabile durante l'esecuzione.

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

**Tipologie di errore:**

**Errore di compilazione**

* **Quando si verifica:** Durante la fase di compilazione, prima che il programma venga eseguito.
* **Cosa lo causa:** Problemi nel codice che impediscono al compilatore di tradurre il programma in codice eseguibile.

**Errore di runtime**

* **Quando si verifica:** Durante l'esecuzione del programma.
* **Cosa lo causa:** Problemi logici o situazioni impreviste che emergono mentre il programma è in esecuzione.

Capitolo 4 - Sound NULL Safety

Immagine che contiene disegno, cartone animato, clipart, illustrazione

Descrizione generata automaticamente

Null è una assenza di valore, da dart v2.12 è stata posto per definizione che **nessuna variabile può essere null (sound NULL Safety)**, a meno che non lo si decida esplicitamente.

Per ciò, se si dichiarazze, ma non si definisse una variabile che poi verrà usata, ci sarà un errore,



Per cui bisognerà specificare la variabile come **nullable** (**?**)



Capitolo 5 – Tipi di dati integrati

Immagine che contiene calzature, testo, cartone animato, schermata

Descrizione generata automaticamente

Quali sono **integrati** in Dart?

**1. Tipi di base**

Questi sono i tipi fondamentali forniti dal linguaggio Dart:

* **int**: Numeri interi.
  + Esempio:

int a = 10;

* **double**: Numeri a virgola mobile (decimali).
  + Esempio:

double b = 3.14;

* **num**: Superclasse di int e double. Può contenere sia numeri interi che decimali.
  + Esempio:

num c = 2.5;

* **String**: Sequenze di caratteri (testo).
  + Esempio:

String s = "Hello, Dart!";

* **bool**: Valori booleani (true o false).
  + Esempio:

bool isReady = true;

**2. Collezioni**

Dart fornisce strutture dati per contenere più valori:

* **List**: Una lista ordinata di elementi (array).
  + Esempio:

List<int> numbers = [1, 2, 3];

List<dynamic> mixed = [1, "Hello", true];

* **Set**: Una collezione di elementi unici (non ordinati).
  + Esempio:

Set<int> uniqueNumbers = {1, 2, 3};

* **Map**: Una collezione di coppie chiave-valore.
  + Esempio:

Map<String, int> ages = {'Alice': 30, 'Bob': 25};

**3. Tipi speciali**

Questi tipi hanno scopi particolari:

* **dynamic**: Una variabile che può contenere valori di qualsiasi tipo e cambiare il tipo durante l'esecuzione.
  + Esempio:

dynamic x = 5;

x = "Ciao";

* **Object**: La superclasse di tutti i tipi in Dart (compresi null e dynamic).
  + Esempio:

Object obj = "Test";

* **Object?**: Una versione nullable di Object (può essere null).
* **void**: Indica che una funzione non restituisce nulla.
  + Esempio:

void greet() {

print("Hello");

}

* **Null**: Un tipo che ha un solo valore, null. Usato per indicare l'assenza di un valore.

**4. Tipi di supporto**

Questi tipi sono inclusi nella libreria di base di Dart e forniscono funzionalità specifiche:

* **Future**: Rappresenta un valore o un errore che potrebbe essere disponibile in futuro (operazioni asincrone).
  + Esempio:

Future<int> fetchNumber() async { return 42; }

 **Stream**: Una sequenza di valori asincroni.

* Esempio:

Stream<int> numberStream = Stream.fromIterable([1, 2, 3]);

 **Iterable**: Una sequenza generica di elementi che può essere iterata.

* Esempio:

Iterable<int> numbers = [1, 2, 3];

**5. Tipi di costanti**

* **const**: Valori immutabili che sono determinati a tempo di compilazione.
  + Esempio:

const pi = 3.14;

* **final**: Valori immutabili che sono determinati a runtime.
  + Esempio:

final currentTime = DateTime.now();

**6. Altri tipi**

* **Runes**: Rappresenta caratteri Unicode. Usato per manipolare stringhe a livello di carattere Unicode.
  + Esempio:

Runes emoji = Runes('\u{1F600}');

* **Symbol**: Rappresenta un simbolo nel codice Dart, utile per metaprogrammazione.
  + Esempio:

Symbol symbol = #mySymbol;

**7. Null safety**

Con Dart 2.12+ è stata introdotta la **null safety**, che distingue tra tipi nullable e non-nullable:

* **Non-nullable:** Una variabile deve avere sempre un valore.
  + Esempio:

int a = 10;

* **Nullable:** Una variabile può essere null.
  + Esempio:

int? a = null;

Capitolo 6 – Operazioni con i numeri

Immagine che contiene cartone animato, clipart, disegno, orologio

Descrizione generata automaticamente

Operazioni con i numeri

**Num** è una classe numero che è un insieme più grande di **int** e **double**.

E’ possibile effettuare operazioni tra loro.

Oltre a ciò, è possibile fare il **casting** del valore di un tipo ad un altro

Es.

num numero = 6.6;

int intero = 0;

double decimale = 1.1;

print("ciao " + numero.toString()); // ciao 6.6

Consersione da numero a stringa (.toString()

String stringa = '5';

print(double.parse(stringa)); // 5.0

Conversione da string a double (double.parse(stringa))

Con i metodi **ceil**, **round** e **floor** è possibile, rispettivamente, arrotondare per eccesso, arrotondare e arrotondare per difetto

  double qwerty = 6.6;

  print(qwerty.ceil()); // 7

  print(qwerty.round()); // 7

  print(qwerty.floor()); // 6

**Operatori aritmetici**:

* **+**
* **-**
* **/**
* **\***
* **%**
* **~/** (il resto dell’intero, es. 10 / 3 = 3.333, con questo operatore avremo 3)

**Operatori di assegnamento**:

* **+=**
* **-=**
* **\*=**
* **/=**

**Es.**

  int intero = 5;

  intero += 5;

  print(intero); // 10

**Operatori di incremento**: **\***

* **++intero**
* **intero++**

  int intero = 5;

  // intero++;

  ++intero;

  print(intero); // 6

**Operatori di decremento**: **\***

* **--intero**
* **Intero--**

  int intero = 5;

  // intero--;

  --intero;

  print(intero); // 4

**\*Nota:** Se è un **pre-incremento**, l’incremento avviene nella riga stessa, come prima operazione, mentre se messo in **post**, l’incremento avverrà nella riga successiva.

Es.

  int intero = 8;

  int qw = ++intero;

  print(qw); // 9

Risultato 9, intero è stato incrementato prima che venisse incamerato dalla variabile qw

Es 2.

  int intero = 8;

  int qw = intero++;

  print(qw); // 8

Risultato 8, intero **non** è stato incrementato prima che venisse incamerato dalla variabile qw, ma dopo.

Capitolo 7 – Operazioni con le stringhe

Immagine che contiene testo, cartone animato, schermata, linea

Descrizione generata automaticamente

Le stringhe sono definibile tramite **singoli apici** (‘’) o **doppi apici** (“”), eventuali singoli apici vengono inseriti tramite lo **slash** inveso (\) con l’escape dei caratteri

**Le stringhe sono**…

… composte da caratteri, *indicizzati*, naturalmente, a partire da 0

… *concatenabili* (string1 + string 2 = strin1string2)

…*interpolabili* con altri tipi, ovvero è possibile inserire le variabili in doppi apici, iterpolandole

Es.

String string4 = 'Ciao, sono in classe ';

int classe = 5;

print("${string4}${classe}"); // Ciao, sono in classe 5

…mandabili a capo con *\n*

Es.

String string5 = 'Ciao,\n mi trovo bene';

… o con le *multiline*, usando i 3 apici singoli,

Es.

String string6 = '''ciao

  come

  va

  ?''';

…o visualizzare in modo *raw* la stringa, usando come prefisso *r*

Es.

String string7 = r'Ciao,\n mi trovo bene';

**Metodi** **base**:

* indexof
* lastindexof
* touppercase
* tolowercase
* split
* substring

Es.

  String testing = "Sono una stringa";

  dynamic formatString0 = testing.contains('w');

  dynamic formatString1 = testing.indexOf('o');

  dynamic formatString2 = testing.lastIndexOf('o');

  dynamic formatString3 = testing.toUpperCase();

  dynamic formatString4 = testing.toLowerCase();

  dynamic formatString5 = testing.split(' ');

  dynamic formatString6 = testing.substring(1);

  print(formatString0); // false

  print(formatString1); // 1

  print(formatString2); // 3

  print(formatString3); // SONO UNA STRINGA

  print(formatString4); // sono una stringa

  print(formatString5); // [Sono, una, stringa]

  print(formatString6); // ono una stringa

Capitolo 8 – Booleani



È **true** o **false**, cos’altro vorresti sapere?

Capitolo 9 – List

Immagine che contiene forniture per ufficio, schermata, testo, grafica

Descrizione generata automaticamente

Struttura dati di dati, equivalenti agli array (es. in js)

Es.

  List<int> testListNumeri = [1, 2, 3];

Es 2.

List<Object> testListOggetti = [1, 2, 3, '', true, null];

**Nota**: in questo caso il tipo Oggetto fa funzionare la lista poiché, i dati inseriti, appartengono alle rispettive classi ttipo di appartenenza (es. true alla classe Boolean, ‘’ alla classe String, ecc…)

Differente è per **null**, che non può essere inserito, poiché assenza di valore e non apprtenente a Classe Object.

Casistiche peculiari:

  List<int> a = [1, 2, 3];

  List<int> b = [1, 2, 3, null];

  List<int?> c = [1, 2, 3, null];

  List<int>? d;

a) Nel caso **a** la lista è definite e dichiarata **corretamente**

b) Nel caso **b** la lista **non** può contenere un **null**, poiché esso non appartiene al tipo intero

c) Nel caso **c**, ponendo come nullable gli **interi** inseriti in lista, sarà possibile inserire un null

d) Nel caso **d**, non essendoci definizione, l’intera **lista** darà null, indi per cui bisognerà dichiarare la lista come nullable

Le liste sono **indicizzabili**, hanno una **lunghezza** e possono esser **concatenate** con altre liste, al pari di come si è già visto con le stringhe

  List<int> lista = [2, 4, 6, 8];

  List<int> lista2 = [1, 3, 5, 7];

  lista.add(100);

  lista.addAll(lista2);

  print(lista);

  print(lista[0]);

  print(lista.length);

  print(lista[lista.length - 1]);

  List<int> lista = [2, 4, 6, 8];

  List<int> lista2 = [1, 3, 5, 7, ...lista];

print(lista2); // [1, 3, 5, 7, 2, 4, 6, 8]

Nel caso sopra, **add** e **addAll**, permettono, rispettivamente, di aggiungere singoli **item o iterabili**, come altre liste, al pari dello **spread** **operator** [lista, …lista2] o rimuoverli (.remove(), .removeAt(), ecc…).

E’ possibile concatenare anche utilizzando i **..add(8)** a seguire.

Metodi utili:

* forEach
* indexOf
* clear()
* contains()
* map
* collection if **\***
* collection for **\*\***

…ecc

**\*** **collection if**: permette di inserire un if in una lista

  int qwerty = 5;

  List<String> lista = [

    'Agamennone',

    'Penelope',

    'Anassimandro',

    if (qwerty > 5) 'Anna',

    'Ettore'

  ];

  print(lista); // [Agamennone, Penelope, Anassimandro, Ettore]

**\*\*** **collection for**: permette di inserire un for in una lista

  List<int> prova = [1, 2, 3];

  List<String> lista2 = ['A', 'B', 'C', for (var i in prova) '${i}', 'D'];

  print(lista2); // [A, B, C, 1, 2, 3, D]

Capitolo 10 – Set



**Cosa sono?**

Struttura dati in cui una **lista** ha dei **valori** sono **unici** e **non ordinati**.

Es.

  var set = {1, 1, 2, 3, 4, 5, 6, 7, 8, 8};

// {1, 2, 3, 4, 5, 6, 7, 8}

Il set **non garantisce** l’ordine, quindi non è possibile usare il suo indice per selezionare un suo item.

Esistono vari modi di **dichiarare** un **Set**:

  var set1 = Set();

  Set<String> set2 = {"ciao", "hello"};

  var set3 = {"ciao", "hello"};

  var set4 = <int>{1, 2};

  print(set1); // {}

  print(set2); // {ciao, hello}

  print(set3); // {ciao, hello}

  print(set4); // {1}

* tramite un construttore Set()
* Dichiarandolo come tale
* ~~Usando le graffe~~
* Dichiando il tipo e poi le graffe

Il **terzo** modo è **errato**, perché, di default, il linguaggio usa dichiarazioni di questo come sintatticamente dei **Map**.

**Metodi utili:**

* add()
* addAll()
* …{} (**sprepad operator**)
* remove()
* removeAll()
* removeWhere()
* collection if
* collection for
* intersect **\***
* union **\*\***
* difference **\*\*\***

var set = <int>{1, 2, 3, 4, 5, 6};

var set2 = <int>{2, 88, 101};

**\*** **intersect**: Fa intersezione di valori uguali

  print(set.intersection(set2)); // {2}

**\*\*** **union**: fa l’unione dei due set

  print(set.union(set2)); // {1, 2, 3, 4, 5, 6, 88, 101}

**\*\*\*** **difference**: stampa i non differenti

  print(set.difference(set2)); // {1, 3, 4, 5, 6}

Capitolo 11 – Map



Struttura dati formata da una lista di item **chiave-valore** (key-value).

Es.

  var list = {

    "name": "Jack",

    "surname": "O'Lantern",

    "age": 28,

  };

Al pari degli altri, è possibile definire il tipo per la chiave e il tipo per il valore:

  var mapVuota = {};

  print(mapVuota.runtimeType); // \_Map<dynamic, dynamic>

  Map<String, int> test = {"numOggetti": 88};

  print(test); // {numOggetti: 88}

*// Esempio tipizzato*

**Aggiunta/modifica valore:**

  Map<String, int> test = {"numOggetti": 88};

  test["qualitaOggetti"] = 8;

  print(test); // {numOggetti: 88, qualitaOggetti: 8}

accedendo dinamicamente ad una proprietà.

**Nota**: al pari degli altri, collectionIf, add, spread operator, remove, foerEach, ecc…sono metodi ammessi e utilizzabili anche con i Map

Es.

  Map<String, int> test = {"numOggetti": 88, if (2 < 5) "tipologiaOggetti": 3};

  test["qualitaOggetti"] = 8;

  print(test); // {numOggetti: 88, tipologiaOggetti: 3, qualitaOggetti: 8}

  var prova = {"a": 1, "b": 2, "c": 3, "d": 4, "e": 5, "f": 6, "g": 7, "h": 8};

  prova.remove("e");

  prova.forEach((key, value) => print(key));

// a

// b

// c

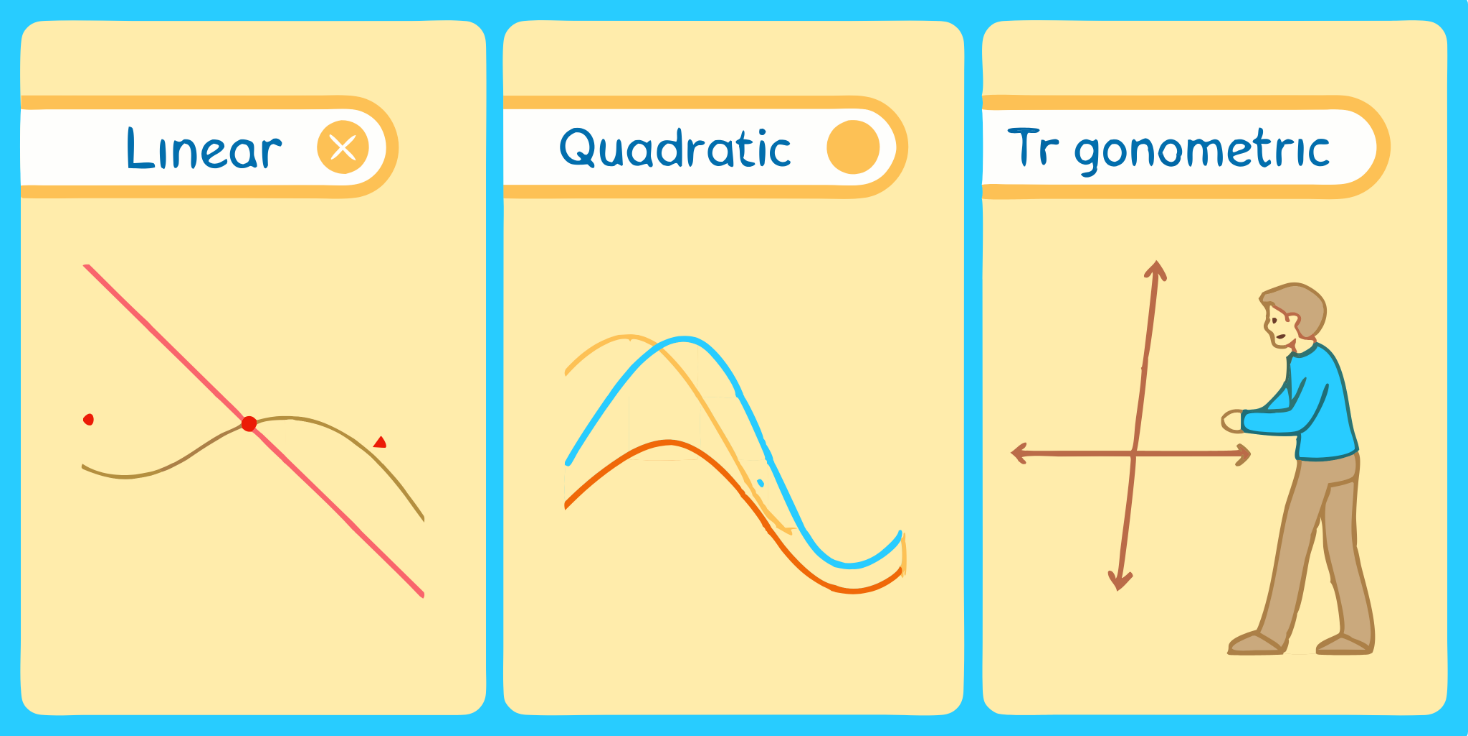
// d

// f

// g

// h

Capitolo 12 – Funzioni



Una **funzione** è un'unità di organizzazione del [codice](https://it.wikipedia.org/wiki/Codice_sorgente) che permette di raggruppare una sequenza di [istruzioni](https://it.wikipedia.org/wiki/Istruzione_(informatica)) in un unico blocco, caratterizzato da un nome, dei parametri in ingresso (detti argomenti) e uno o più [dati](https://it.wikipedia.org/wiki/Dato) restituiti in uscita.

void main(List<String> arguments) {

  // --- Funzioni:

  bananaPrint();

}

void bananaPrint() {

  print('banana');

}

Un a funzione accetta dei parametri (**arguments**), usati nello scope della funzione di codice stessa.

void main(List<String> arguments) {

  // --- Funzioni:

  var p = somma(6, 8);

  print(p); // 14

}

int somma(int a, int b) {

  int c = a + b;

  return c;

}

L’uso di **void** ammette alla funzione di effettuare il **return** di un valore **null**, altrimenti, il return di un null, se non esplicitato differentemente farà **crashare** la nostra applicazione.

**NOTA**: A differenza di altri linguaggi, in Dart le funzioni sono **oggetti**, definiti da una **classe astratta funzione**.

**Arrow function:** Una **arrow function** è una funzione concisa che utilizza la sintassi => per esprimere una singola espressione in modo compatto. È particolarmente utile per funzioni semplici, come i callback o le funzioni di una riga.

  …

var b = somma;

  var x = b(5, 10);

  print(x); //15

}

int somma(int a, int b) => a + b;

Capitolo 13 – Funzioni anonime

**Immagine che contiene disegno, illustrazione, Viso umano, schizzo

Descrizione generata automaticamente**

Una **funzione** **anonima** (o **lambda function**) è una funzione senza nome, utile quando devi passare una funzione come parametro o usarla inline senza definirla separatamente.

**Sintassi**

Le funzioni anonime possono essere dichiarate con due sintassi:

1. **Classica** con return
2. **Arrow Function** con => (se è a singola espressione)

// Sintassi classica

(parametri) {

// Corpo della funzione

};

// Sintassi arrow (se ha una sola espressione)

(parametri) => espressione;

**Esempi**

**1. Funzione anonima assegnata a una variabile**

var somma = (int a, int b) {

return a + b;

};

void main() {

print(somma(3, 4)); // Output: 7

}

Nell’esempio sopra abbiamo, selezionata, una **funzione** **anonima** e anche di tipo **arrow**.

**2. Arrow Function anonima**

var moltiplica = (int a, int b) => a \* b;

void main() {

print(moltiplica(3, 4)); // Output: 12

}

**3. Funzione anonima come callback**

Le funzioni anonime sono molto usate come callback, ad esempio con forEach:

void main() {

var numeri = [1, 2, 3, 4];

numeri.forEach((n) {

print('Numero: $n');

});

// Output:

// Numero: 1

// Numero: 2

// Numero: 3

// Numero: 4

}

**4. Funzione anonima con map**

void main() {

var numeri = [1, 2, 3, 4];

var doppi = numeri.map((n) => n \* 2).toList();

print(doppi); // Output: [2, 4, 6, 8]

}

**Differenza tra Funzioni Anonime e Arrow Functions**

* Le **funzioni anonime** possono avere più istruzioni nel loro corpo { ... }
* Le **arrow function** sono solo per espressioni singole (non supportano blocchi { ... })

**Esempio di funzione anonima con più istruzioni:**

var saluta = (String nome) {

print("Ciao, $nome!");

print("Come stai?");

};

void main() {

saluta("Luca");

// Output:

// Ciao, Luca!

// Come stai?

}

Capitolo 14 – Tipi di Parametri nelle Funzioni

**Immagine che contiene vestiti, testo, schermata, calzature

Descrizione generata automaticamente**

Esistono **quattro** tipi principali di parametri che puoi usare nelle funzioni per un totale di **16 tipi**:

1. **Parametri Posizionali** (obbligatori)
2. **Parametri Posizionali Opzionali**
3. **Parametri Nominali (o Nominati)**
4. **Parametri con Valore Predefinito**

### 1. Parametri Posizionali (Obbligatori)

Questi parametri devono essere passati nell'ordine corretto.

void stampaNome(String nome, int eta) {

print("Nome: $nome, Età: $eta");

}

void main() {

stampaNome("Luca", 25); // Output: Nome: Luca, Età: 25

}

### 2. Parametri Posizionali Opzionali

Si dichiarano tra **[ ]** e possono essere omessi. Se non forniti, il valore sarà null (a meno che non si assegni un valore predefinito).

void stampaNome(String nome, [int? eta]) {

print("Nome: $nome, Età: ${eta ?? 'Sconosciuta'}");

}

void main() {

stampaNome("Luca"); // Output: Nome: Luca, Età: Sconosciuta

stampaNome("Anna", 30); // Output: Nome: Anna, Età: 30

### 3. Parametri Nominali (o Nominati)

Si dichiarano tra { } e devono essere passati con il nome. Sono utili quando ci sono molti parametri per evitare confusione.

void stampaDati({required String nome, required int eta}) {

print("Nome: $nome, Età: $eta");

}

void main() {

stampaDati(nome: "Luca", eta: 25); // Output: Nome: Luca, Età: 25

}

**NOTA**: **required** indica che il parametro è obbligatorio. Se non lo metti, il parametro diventa opzionale.

### 4. Parametri con Valore Predefinito

Si usano con i parametri nominati { } e vengono assegnati valori di default.

void saluta({String nome = "Ospite"}) {

print("Ciao, $nome!");

}

void main() {

saluta(); // Output: Ciao, Ospite!

saluta(nome: "Luca"); // Output: Ciao, Luca!

}

**Combinazione di Tipi di Parametri**

Puoi combinare posizionali, opzionali e nominati, ma le regole sono:

* I **posizionali obbligatori** vanno sempre prima.
* Poi possono esserci **posizionali opzionali** [ ].
* I **nominati** { } vanno sempre dopo.

Es.

void mostraInfo(String nome, {int eta = 18, String? città}) {

print("Nome: $nome, Età: $eta, Città: ${città ?? 'Non specificata'}");

}

void main() {

mostraInfo("Luca"); // Output: Nome: Luca, Età: 18, Città: Non specificata

mostraInfo("Anna", città: "Roma"); // Output: Nome: Anna, Età: 18, Città: Roma

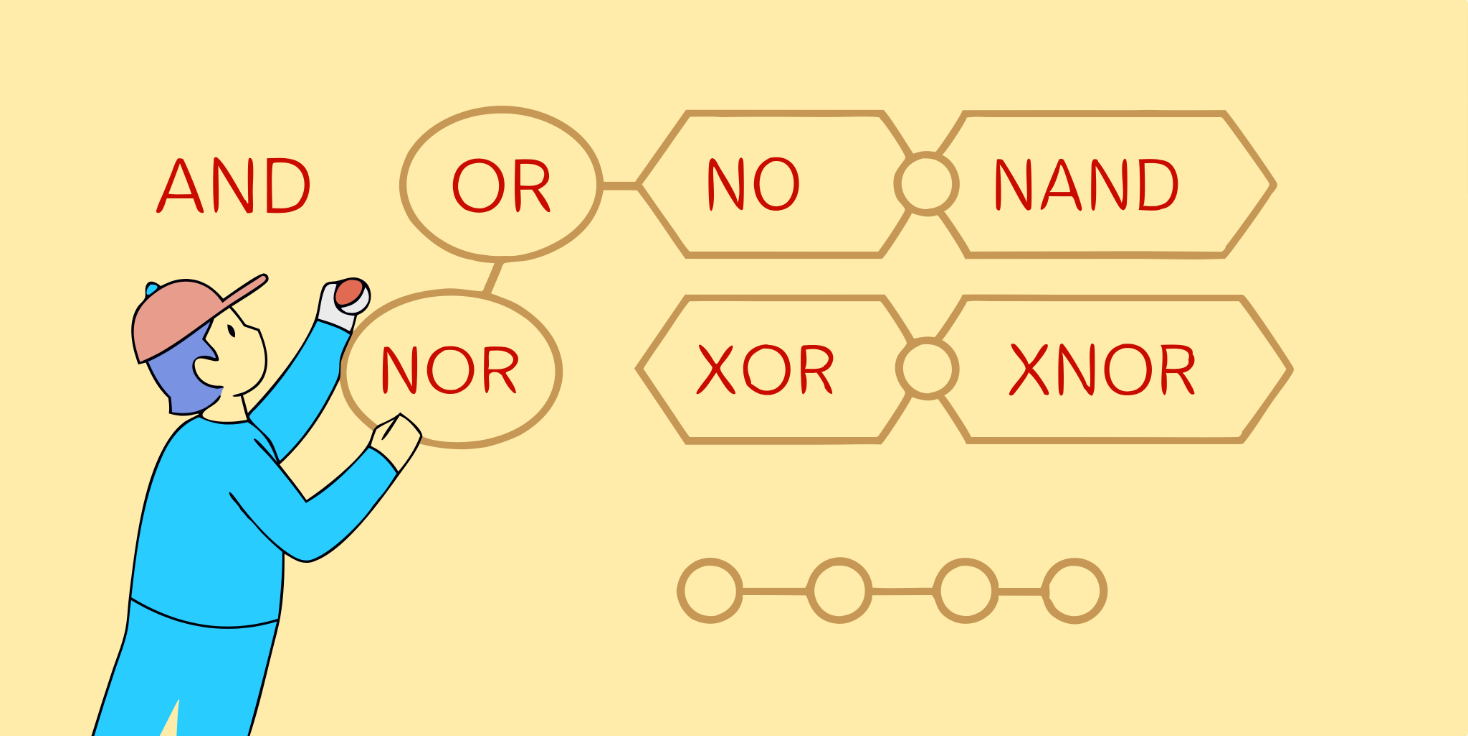
mostraInfo("Marco", eta: 30, città: "Milano"); // Output: Nome: Marco, Età: 30, Città: Milano

}

**In sintesi:**

| **Tipo di Parametro** | **Sintassi** | **Opzionale?** | **Richiede { } o [ ]?** | **Può avere un valore predefinito?** |
| --- | --- | --- | --- | --- |
| **Posizionale** | (String nome) | ❌ No | ❌ No | ❌ No |
| **Posizionale opzionale** | ([int? eta]) | ✅ Sì | ✅ [ ] | ✅ Sì (se specificato) |
| **Nominato** | ({String nome}) | ✅ Sì | ✅ { } | ✅ Sì (se specificato) |
| **Nominato obbligatorio** | ({required int eta}) | ❌ No | ✅ { } | ❌ No |

Capitolo 15 – Operatori logici e di comparazione



Gli **operatori logici** vengono utilizzati per combinare espressioni booleane e restituire un valore true o false. Sono molto utili nelle condizioni if, nei cicli e nelle espressioni di controllo di flusso.

**Operatori logici principali**

Dart fornisce i seguenti operatori logici:

| **Operatore** | **Nome** | **Descrizione** | **Esempio** |
| --- | --- | --- | --- |
| && | AND logico | Restituisce true se entrambe le condizioni sono vere | (true && false) → false |
| | | | OR logico | Restituisce true se una delle condizioni sono vere | OR logico |
| ! | NOT logico | Inverte il valore booleano | !true → false |

**🔹 Esempi di utilizzo**

1. **AND logico (&&)**

L'operatore && restituisce true solo se **entrambi** gli operandi sono true:

void main() {

bool a = true;

bool b = false;

print(a && b); // false

print(a && true); // true

}

1. **OR logico (||)**

L'operatore || restituisce true se almeno **uno** degli operandi è true:

void main() {

bool a = true;

bool b = false;

print(a || b); // true

print(b || false); // false

}

1. **NOT logico (!)**

L'operatore ! inverte il valore booleano:

void main() {

bool a = true;

print(!a); // false

print(!false); // true

}

**NOTA:** Operatore di **"short-circuiting"**

Dart utilizza il **"short-circuit evaluation"**, il che significa che:

* In un'operazione &&, se il primo operando è false, il secondo non viene valutato.
* In un'operazione ||, se il primo operando è true, il secondo non viene valutato.

Esempio:

void main() {

bool test() {

print("Funzione chiamata!");

return true;

}

print(false && test()); // Stampa solo "false", la funzione non viene chiamata.

print(true || test()); // Stampa solo "true", la funzione non viene chiamata.

}

Gli **operatori di confronto** servono per confrontare due valori e restituiscono un valore booleano (true o false). Sono comunemente usati in condizioni e cicli.

**Operatori di confronto principali**

| **Operatore** | **Nome** | **Descrizione** | **Esempio** |
| --- | --- | --- | --- |
| == | Uguale a | Restituisce true se i valori sono uguali | 5 == 5 → true |
| != | Diverso da | Restituisce true se i valori sono diversi | 5 != 3 → true |
| < | Minore di | Restituisce true se il primo valore è minore del secondo | 3 < 5 → true |
| <= | Minore o uguale a | Restituisce true se il primo valore è minore o uguale al secondo | 5 <= 5 → true |
| > | Maggiore di | Restituisce true se il primo valore è maggiore del secondo | 10 > 5 → true |
| >= | Maggiore o uguale a | Restituisce true se il primo valore è maggiore o uguale al secondo | 8 >= 10 → false |

**🔹 Esempi di utilizzo**

void main() {

int a = 10;

int b = 5;

print(a == b); // false

print(a != b); // true

print(a > b); // true

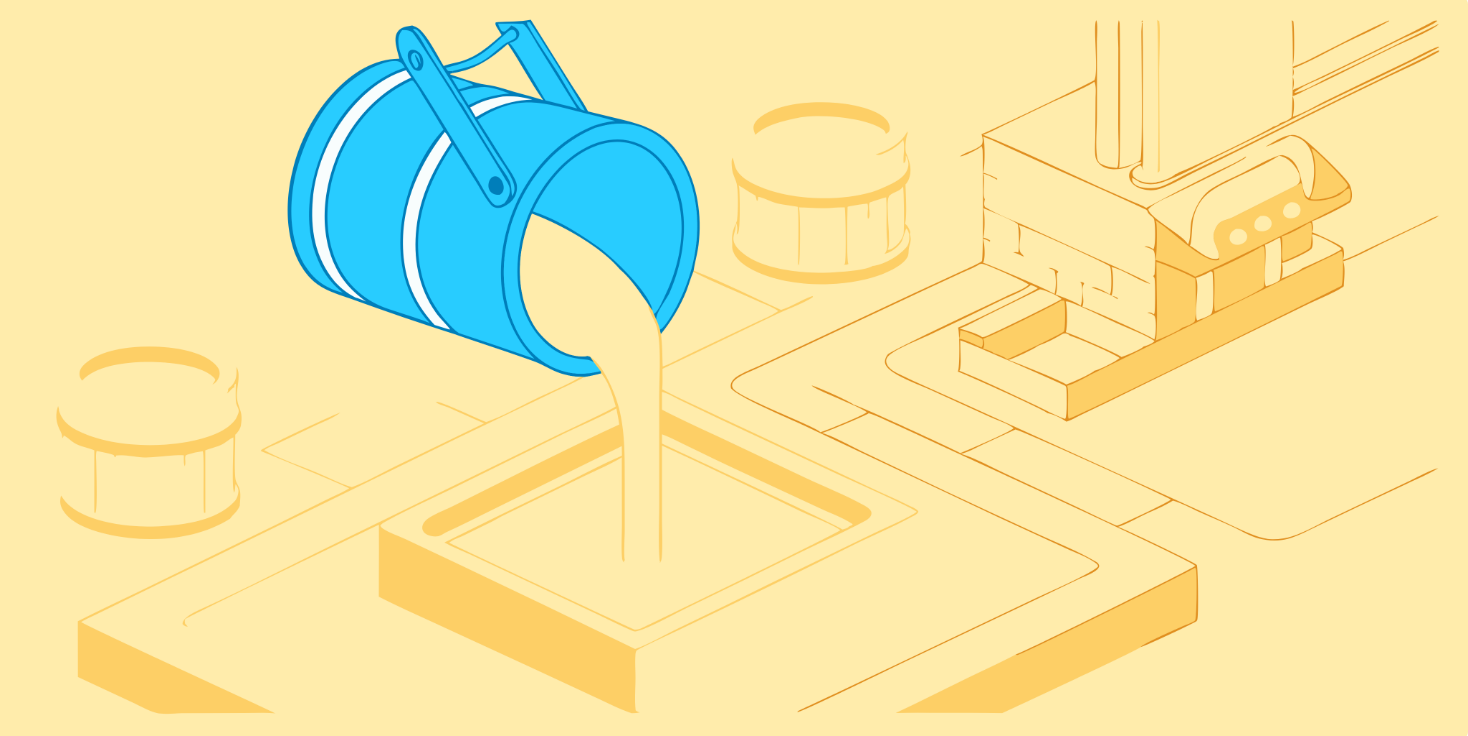
print(a < b); // false

print(a >= 10); // true

print(b <= 5); // true

}

Capitolo 16 – Operatori Type Test, AS IS, IS NOT



Gli **operatori Type Test** in Dart vengono usati per controllare il tipo di una variabile o per fare cast di tipo. Sono molto utili quando si lavora con **tipi dinamici** o quando si vuole assicurare che una variabile sia di un certo tipo prima di eseguire un'operazione su di essa.

| **Operatore** | **Nome** | **Descrizione** |
| --- | --- | --- |
| as | Type Cast | Converte un valore in un tipo specifico |
| is | Verifica il tipo | Restituisce true se l'oggetto è del tipo specificato |
| is! | Verifica il tipo negato | Restituisce true se l'oggetto **non** è del tipo specificato |

**Esempi di utilizzo**

* **is → Verifica se un oggetto è di un certo tipo**

L'operatore is restituisce true se una variabile è di un certo tipo.

void main() {

var valore = "Ciao";

print(valore is String); // true

print(valore is int); // false

}

* **is! → Verifica se un oggetto NON è di un certo tipo**

L'operatore is! restituisce true se la variabile **non** è di un certo tipo.

void main() {

var numero = 42;

print(numero is! String); // true

print(numero is! int); // false

}

* **as → Type Casting (conversione di tipo)**

L'operatore as viene usato per **convertire** un oggetto in un altro tipo, se il tipo è corretto.

void main() {

dynamic valore = "Hello";

// Casting sicuro, perché sappiamo che valore è una String

String testo = valore as String;

print(testo.toUpperCase()); // HELLO

}

⚠️ **Attenzione!** Se il cast **as** è usato su un valore del tipo sbagliato, genera un errore di runtime:

void main() {

dynamic numero = 42;

String testo = numero as String; // ERRORE: type 'int' is not a subtype of type 'String'

}

Per evitare errori, è consigliabile usare is prima di fare un cast:

void main() {

dynamic valore = "Dart";

if (valore is String) {

String testo = valore as String;

print(testo.toLowerCase()); // dart

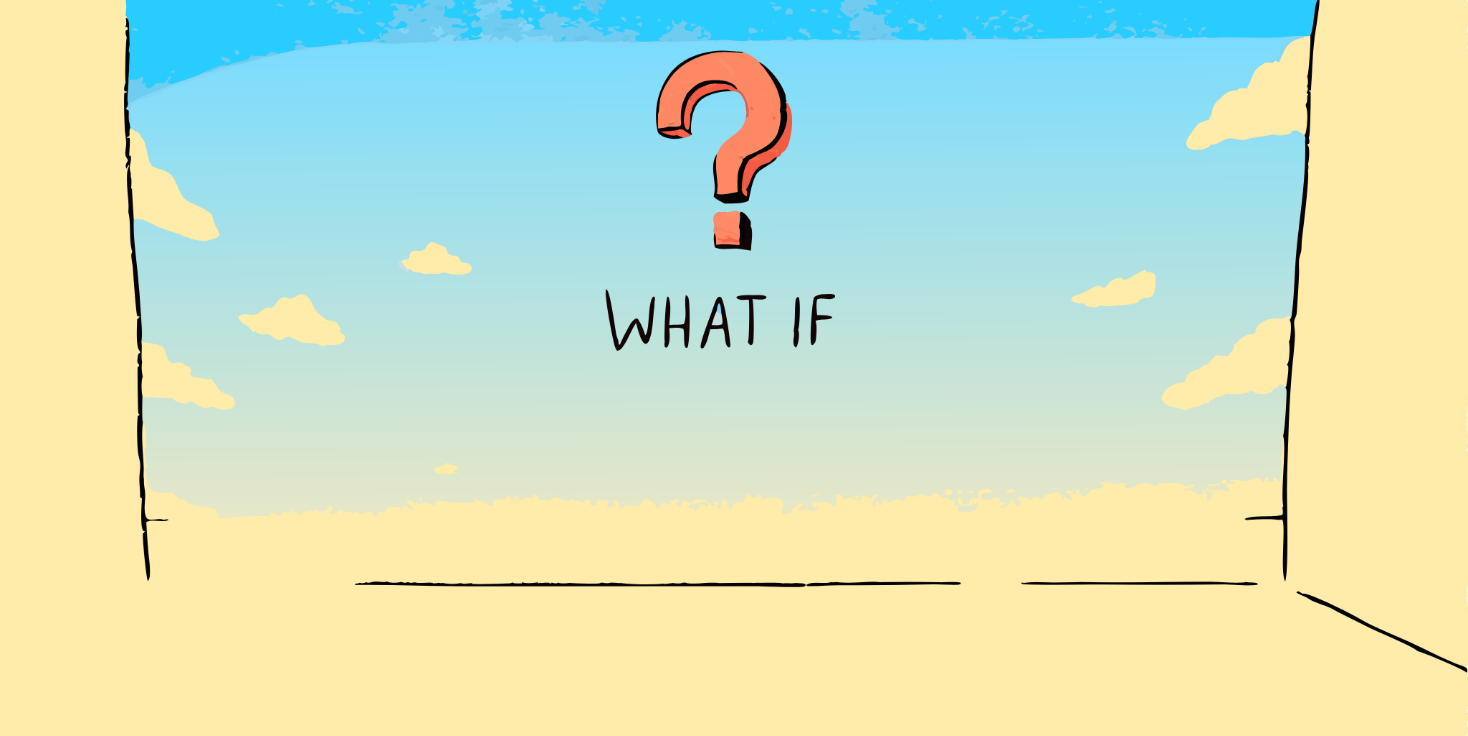
}

}

**🔹 Quando usarli?**

* **is e is!** per verificare il tipo di una variabile prima di usarla.
* **as** per eseguire un **cast di tipo**, ma solo se sei sicuro del tipo effettivo.

Capitolo 17 – IF ELSE e le Espressioni Condizionali



Le **istruzioni condizionali** servono per controllare il flusso del programma in base a determinate condizioni. Le principali strutture condizionali sono:

* **if**
* **if-else**
* **if-else if-else**
* **Espressioni condizionali (?: e ??)**
* **if statement**

La struttura di base if esegue un blocco di codice solo se la condizione è vera.

void main() {

int numero = 10;

if (numero > 5) {

print("Il numero è maggiore di 5");

}

} // Il numero è maggiore di 5

* **if-else statement**

Se la condizione è falsa, il blocco else viene eseguito.

void main() {

int numero = 3;

if (numero > 5) {

print("Il numero è maggiore di 5");

} else {

print("Il numero è minore o uguale a 5");

}

} // Il numero è minore o uguale a 5

**🔹 if-else if-else statement**

Se ci sono più condizioni, puoi usare else if per verificarle una alla volta.

void main() {

int voto = 85;

if (voto >= 90) {

print("Ottimo!");

} else if (voto >= 70) {

print("Buono!");

} else {

print("Devi migliorare.");

}

} // Buono!

**🔹 Espressioni condizionali (Ternary Operator)**

In Dart, puoi usare l'operatore **ternario (?:)** per scrivere condizioni compatte.

void main() {

int numero = 8;

String risultato = (numero > 5) ? "Maggiore di 5" : "Minore o uguale a 5";

print(risultato);

} // Maggiore di 5

**🔹 Espressione ?? (Null-aware Operator)**

L'operatore ?? restituisce il primo valore **non nullo**.

void main() {

String? nome;

// Se nome è null, usa il valore di default "Sconosciuto"

String risultato = nome ?? "Sconosciuto";

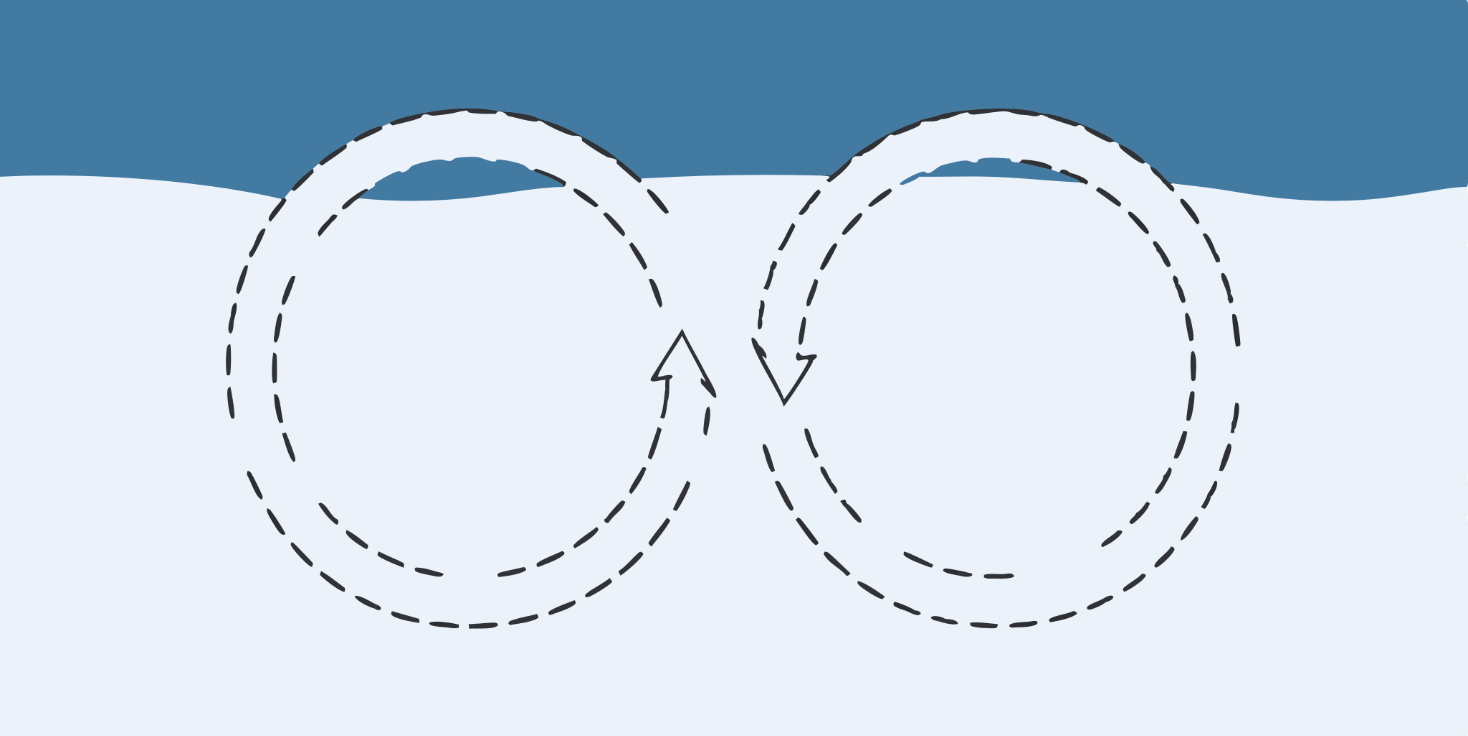
print(risultato);

} // Sconosciuto

**🔹 Quando usare quale?**

* Usa if-else quando hai più di una condizione complessa.
* Usa ?: quando la condizione è semplice e vuoi una scrittura più compatta.
* Usa ?? per gestire i valori nulli in modo pulito.

Capitolo 18 – IF ELSE e le Espressioni Condizionali



Ci sono diversi modi per iterare su una sequenza di elementi o ripetere un blocco di codice più volte. I cicli principali sono:

* **for**
* **for-in**
* **forEach**

**🔹 Ciclo for (Classico)**

Il ciclo for viene utilizzato quando conosci il numero esatto di iterazioni. Ha questa struttura:

for (inizializzazione; condizione; aggiornamento) {

// Blocco di codice da eseguire

}

**Esempio:**

void main() {

for (int i = 0; i < 5; i++) {

print("Iterazione numero $i");

}

}

// Iterazione numero 0

// Iterazione numero 1

// Iterazione numero 2

// Iterazione numero 3

// Iterazione numero 4

🔹 **Spiegazione**:

1. int i = 0; → Inizializzazione della variabile contatore.
2. i < 5; → Il ciclo continua finché la condizione è vera.
3. i++ → Incremento di i a ogni iterazione.

**🔹 Ciclo for-in (Per le collezioni)**

Il ciclo for-in è usato per iterare attraverso **liste, set e mappe** in modo più semplice rispetto al for classico.

**Esempio**

void main() {

List<String> nomi = ["Alice", "Bob", "Charlie"];

for (String nome in nomi) {

print("Ciao, $nome!");

}

}

// Ciao, Alice!

// Ciao, Bob!

// Ciao, Charlie!

🔹 **Vantaggi**:

* Più leggibile rispetto a for classico.
* Ideale per iterare su **liste e collezioni**.

**🔹 Ciclo forEach (Metodo delle liste)**

Il metodo forEach è un altro modo per iterare su una lista, usando una **funzione anonima o lambda**.

**Esempio**

void main() {

List<int> numeri = [1, 2, 3, 4, 5];

numeri.forEach((numero) {

print("Numero: $numero");

});

}

// Numero: 1

// Numero: 2

// Numero: 3

// Numero: 4

// Numero: 5

Puoi anche usare una funzione **arrow** per semplificare il codice:

numeri.forEach((numero) => print("Numero: $numero"));

🔹 **Vantaggi**:

* Ideale per **programmazione funzionale**.
* **Più compatto e leggibile** rispetto a for-in.

**🔹 Quando usare quale?**

| **Metodo** | **Quando usarlo** |
| --- | --- |
| **for** | Se devi gestire un **contatore manuale** o modificare la sequenza |
| **for-in** | Se vuoi **iterare su una lista** senza gestire l'indice |
| **forEach** | Se vuoi **mantenere il codice più pulito** e funzionale |

Capitolo 19 – Ciclo WHILE e DO WHILE



I cicli **while** e **do-while** vengono utilizzati quando non conosci il numero esatto di iterazioni in anticipo, ma vuoi ripetere un blocco di codice finché una condizione rimane vera.

**Ciclo while**

Il ciclo while esegue il blocco di codice **finché la condizione è vera**. Se la condizione è **falsa all'inizio**, il codice **non verrà mai eseguito**.

**Sintassi**

while (condizione) {

// Blocco di codice da eseguire

}

**Esempio**

void main() {

int numero = 0;

while (numero < 5) {

print("Numero: $numero");

numero++; // Incremento della variabile

}

}

// Numero: 0

// Numero: 1

// Numero: 2

// Numero: 3

// Numero: 4

**Spiegazione**:

* Il codice entra nel ciclo **solo se** numero < 5 è vero.
* A ogni iterazione, numero viene incrementato.
* Appena numero diventa **5**, il ciclo si interrompe.

**Ciclo do-while**

Il ciclo do-while è simile a while, con una differenza importante: **esegue il blocco almeno una volta**, indipendentemente dalla condizione.

**Sintassi**

do {

// Blocco di codice da eseguire

} while (condizione);

**Esempio**

void main() {

int numero = 5;

do {

print("Numero: $numero");

numero++;

} while (numero < 5);

} // Numero: 5

**Spiegazione**:

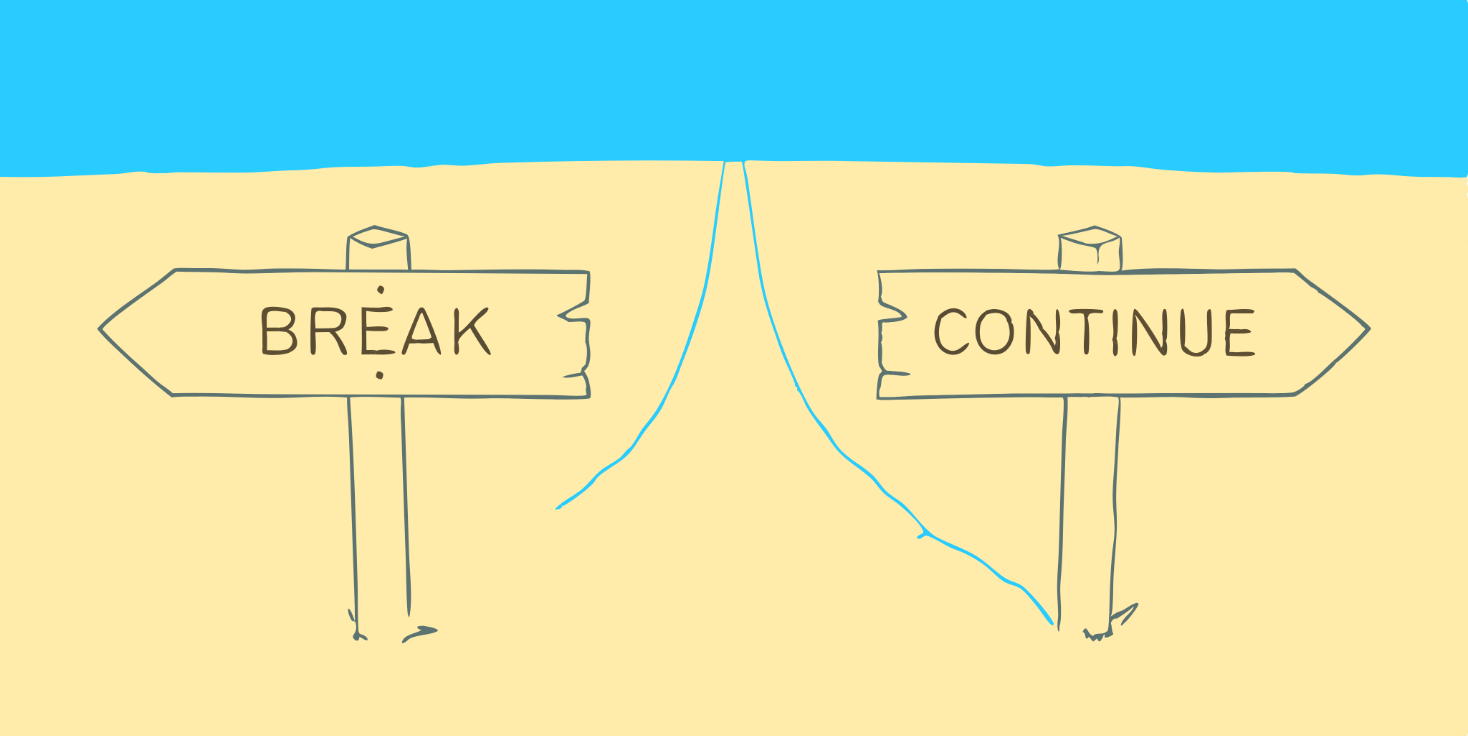
* Anche se numero < 5 è **falso fin dall'inizio**, il codice **viene comunque eseguito una volta** prima di controllare la condizione.
* **Differenze tra while e do-while**

| **Caratteristica** | **while** | **do-while** |
| --- | --- | --- |
| Controllo della condizione | All'inizio | Alla fine |
| Esecuzione garantita almeno una volta | ❌ No | ✅ Sì |
| Utilizzo consigliato | Quando vuoi eseguire il blocco **solo se** la condizione è vera | Quando vuoi eseguire il blocco **almeno una volta** |

**Quando usarli?**

* Usa while **quando non sei sicuro** che il blocco debba essere eseguito almeno una volta.
* Usa do-while **quando vuoi garantire** che il blocco venga eseguito almeno una volta (es. input utente con ripetizione fino a un valore valido).

Capitolo 20 – Break e Continue



Parole chiave che permettono di bloccare o far avanzare un ciclo.

**Esempio**

var lista = ['banane', 'carote', 'zucchine'];

for (var i = 0; i < lista.length; i++) {

    if (lista[i] == 'carote') {

      print(lista[i]);

      break;

    } else {

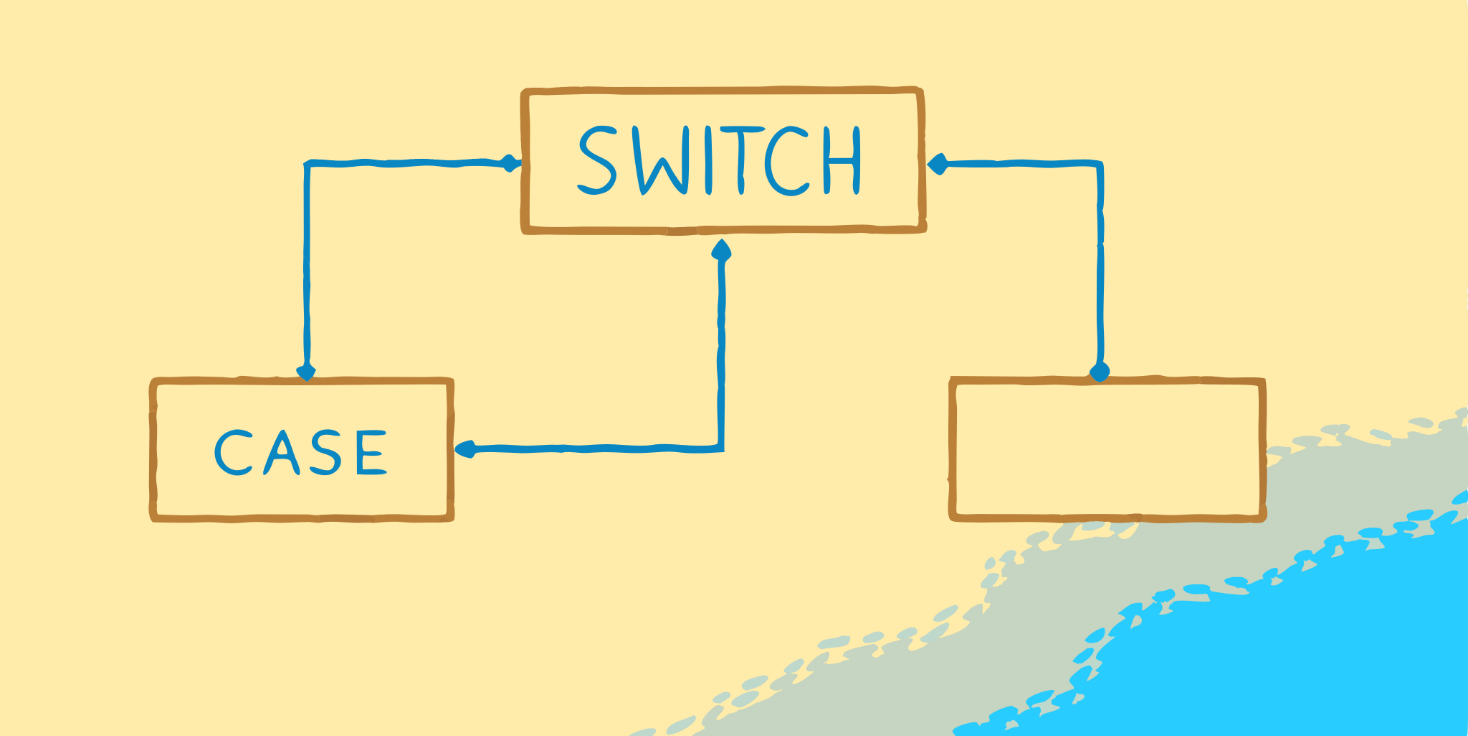
      print(lista[i]);

      continue;

    }

  } // banane carote

Capitolo 21 – Switch case



Il costrutto switch-case viene utilizzato per gestire più condizioni in modo più chiaro rispetto a una lunga serie di if-else if.

**Sintassi di switch-case**

switch (espressione) {

case valore1:

// Blocco di codice da eseguire

break;

case valore2:

// Blocco di codice da eseguire

break;

default:

// Blocco eseguito se nessun caso corrisponde

}

**Regole importanti**:

* Ogni case deve terminare con break, return o un altro comando che interrompe il flusso.
* Il default è opzionale, ma consigliato per gestire i casi non previsti.

**Esempio base**

void main() {

String giorno = "Lunedì";

switch (giorno) {

case "Lunedì":

print("Inizio della settimana!");

break;

case "Venerdì":

print("Quasi weekend!");

break;

case "Sabato":

case "Domenica":

print("È weekend!");

break;

default:

print("Giorno normale.");

}

} // Inizio della settimana!

**Spiegazione**:

* Il valore di giorno è "Lunedì", quindi viene eseguito il codice dentro case "Lunedì".
* Il break impedisce che vengano eseguiti altri casi.
* "Sabato" e "Domenica" condividono lo stesso codice perché non hanno break tra loro.
* **Switch con int**

Puoi usare switch-case anche con numeri interi:

void main() {

int voto = 8;

switch (voto) {

case 10:

print("Eccellente!");

break;

case 8:

print("Molto bene!");

break;

case 6:

print("Sufficiente.");

break;

default:

print("Voto non valido.");

}

}

// Molto bene!

* **Switch con enum (consigliato per categorie)**

Se hai categorie ben definite, puoi usare **enum** per rendere il codice più leggibile.

enum StatoOrdine { inLavorazione, spedito, consegnato }

void main() {

StatoOrdine stato = StatoOrdine.spedito;

switch (stato) {

case StatoOrdine.inLavorazione:

print("Il tuo ordine è in preparazione.");

break;

case StatoOrdine.spedito:

print("Il tuo ordine è stato spedito.");

break;

case StatoOrdine.consegnato:

print("Il tuo ordine è stato consegnato.");

break;

}

}

// Il tuo ordine è stato spedito.

**Vantaggi di usare enum**:

* **Codice più leggibile e mantenibile**.
* **Evita errori con stringhe errate** (es. "spedito" scritto male).
* **switch-case senza break**

Se ometti break, l'esecuzione continua nei casi successivi.

dart

CopiaModifica

void main() {

int numero = 2;

switch (numero) {

case 1:

print("Uno");

case 2:

print("Due");

case 3:

print("Tre");

default:

print("Altro numero");

}

}

// Due

// Tre

// Altro numero

**Spiegazione**:

* Poiché numero == 2, stampa "Due".
* **Mancando il break**, continua ad eseguire "Tre" e "Altro numero".

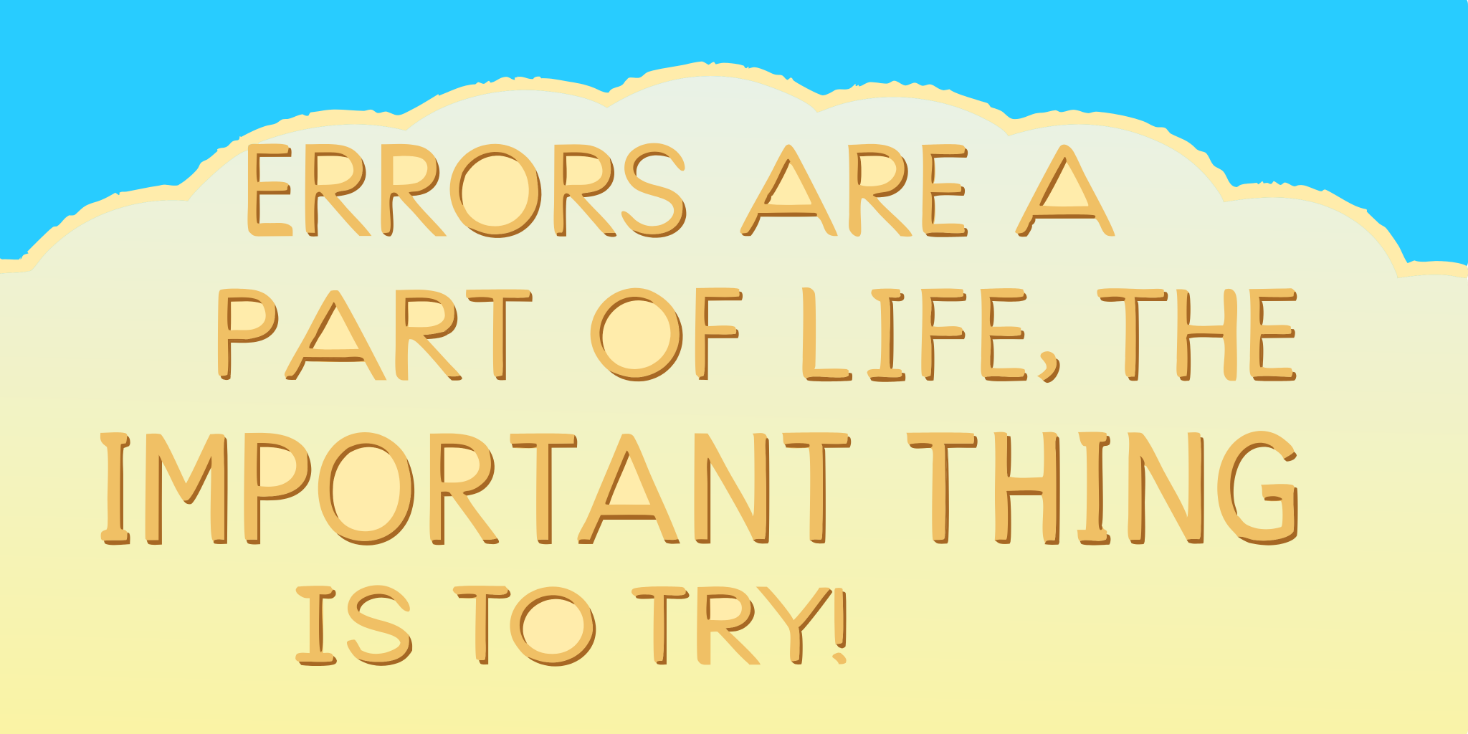
**Quando usare switch-case?**

| **Usa if-else quando...** | **Usa switch-case quando...** |
| --- | --- |
| Le condizioni sono **complesse** | Stai confrontando un **valore specifico** (int, string, enum) |
| Usi **operatori logici** (&&, ` |  |
| Hai solo **2-3 condizioni** | Vuoi un codice **più leggibile** per più opzioni |

**Conclusione**

* switch-case è ottimo per confrontare **valori esatti**.
* Usa enum per evitare errori con stringhe o numeri.
* Ricorda di **usare break** per evitare esecuzioni non desiderate.

Capitolo 22 – Try e Catch



La gestione degli errori avviene tramite try, catch, on, e finally, in modo simile ad altri linguaggi come JavaScript e Java. Vediamo come funzionano.

**1. try e catch**

Il blocco try viene usato per avvolgere il codice che potrebbe generare un'eccezione, mentre catch intercetta e gestisce l'errore.

void main() {

try {

int risultato = 10 ~/ 0; // Divisione per zero

print(risultato);

} catch (e) {

print('Errore catturato: $e');

}

}

💡 **Nota**: ~/ è l'operatore di divisione intera, e in Dart una divisione per zero genera un'eccezione (IntegerDivisionByZeroException).

**2. on per errori specifici**

Se vuoi gestire un tipo specifico di errore, puoi usare on:

void main() {

try {

int risultato = 10 ~/ 0;

print(risultato);

} on IntegerDivisionByZeroException {

print('Errore: divisione per zero!');

}

}

👉 **Quando usare on invece di catch?**

* Usa on se vuoi gestire un'eccezione specifica.
* Usa catch per gestire qualsiasi eccezione.

**3. catch con stack trace**

Se vuoi ottenere più dettagli sull'errore, puoi usare catch con due parametri:

void main() {

try {

List<int> numeri = [1, 2, 3];

print(numeri[5]); // Errore: indice fuori range

} catch (e, stackTrace) {

print('Errore: $e');

print('Stack trace: $stackTrace');

}

}

**4. finally**

Il blocco finally viene sempre eseguito, indipendentemente dal fatto che ci sia stato un errore o meno.

void main() {

try {

print('Apro una connessione...');

int risultato = 10 ~/ 0;

print(risultato);

} catch (e) {

print('Errore: $e');

} finally {

print('Chiudo la connessione.');

}

}

👉 **Utile per:**

* Chiudere file o connessioni
* Liberare risorse
* Eseguire codice che deve sempre essere eseguito

**5. Generare errori con throw**

Puoi lanciare un'eccezione personalizzata con **throw**:

void checkNumero(int numero) {

if (numero < 0) {

throw ArgumentError('Il numero non può essere negativo!');

}

print('Numero accettato: $numero');

}

void main() {

try {

checkNumero(-5);

} catch (e) {

print('Errore: $e');

}

}

**6. Creare eccezioni personalizzate**

Puoi definire una tua classe di eccezione:

class ErrorePersonalizzato implements Exception {

String messaggio;

ErrorePersonalizzato(this.messaggio);

@override

String toString() => 'ErrorePersonalizzato: $messaggio';

}

void main() {

try {

throw ErrorePersonalizzato('Qualcosa è andato storto!');

} catch (e) {

print(e);

}

}

**ErrorePersonalizzato** è un’estensione della classe **Exception**, che viene richiamata quando bisognoa restituire un messaggio di errore

**@override** è una keyword che permette di sovrascrivere un metodo della classe madre.

toString() è un metodo che converte un oggetto in una stringa

**Conclusione**

* try → Protegge il codice che potrebbe generare errori.
* catch (e) → Cattura qualsiasi errore.
* on TipoDiErrore → Cattura un errore specifico.
* catch (e, stackTrace) → Cattura l'errore e il suo stack trace.
* finally → Eseguito sempre, utile per cleanup.
* throw → Lancia un'eccezione.
* **Eccezioni personalizzate** → Migliorano la gestione degli errori nel codice.

Capitolo 23-27 – Programmazione Orientata agli oggetti

Immagine che contiene diagramma, Piano, mappa, linea

Il contenuto generato dall'IA potrebbe non essere corretto.

La **Programmazione Orientata agli Oggetti (OOP)** è un paradigma che organizza il codice attorno agli **oggetti** e alle loro **relazioni**. In Dart, tutto è un oggetto (persino i numeri e le funzioni).

L'OOP in Dart si basa su quattro pilastri fondamentali:

1. **Incamapsulamento** → protezione e organizzazione dei dati
2. **Ereditarietà** → riuso del codice e creazione di gerarchie
3. **Polimorfismo** → oggetti diversi possono comportarsi in modo simile
4. **Astrazione** → nascondere i dettagli complessi e mostrare solo ciò che serve

**1️⃣ Incapsulamento: Protezione e Organizzazione**

L'**incapsulamento** significa nascondere i dettagli interni di una classe e fornire un'interfaccia controllata. In Dart, i campi e metodi **privati** iniziano con \_ (underscore).

**Esempio: Classe con incapsulamento**

class ContoBancario {

String titolare;

double \_saldo; // Privato, accessibile solo all'interno della classe

ContoBancario(this.titolare, this.\_saldo);

// Getter per ottenere il saldo in modo controllato

double get saldo => \_saldo;

// Metodo per depositare denaro (con validazione)

void deposita(double importo) {

if (importo <= 0) throw Exception('Importo non valido');

\_saldo += importo;

}

// Metodo per prelevare denaro (con controllo)

void preleva(double importo) {

if (importo > \_saldo) throw Exception('Saldo insufficiente');

\_saldo -= importo;

}

}

void main() {

var conto = ContoBancario('Luca', 1000);

conto.deposita(500);

conto.preleva(300);

print(conto.saldo); // 1200

// conto.\_saldo = 5000; // ERRORE! \_saldo è privato

}

📌 **Cosa abbiamo fatto?**

* Il saldo \_saldo è privato: nessuno può modificarlo direttamente.
* Abbiamo fornito **metodi controllati** (deposita e preleva).
* Usiamo un **getter** (saldo) per accedere al saldo senza permettere modifiche dirette.

**2️⃣ Ereditarietà: Riutilizzo e Gerarchie**

L'**ereditarietà** permette a una classe di **estendere** un'altra per riutilizzarne il codice.  
In Dart si usa la parola chiave extends.

**Esempio: Classe base e classe derivata**

class Animale {

String nome;

Animale(this.nome);

void faiVerso() {

print('Verso generico');

}

}

class Cane extends Animale {

Cane(String nome) : super(nome); // super chiama il costruttore della classe base

@override

void faiVerso() {

print('$nome dice: Bau!');

}

}

void main() {

var fido = Cane('Fido');

fido.faiVerso(); // Fido dice: Bau!

}

📌 **Cosa succede qui?**

* Cane eredita da Animale, quindi **non serve riscrivere il costruttore**.
* super(nome); chiama il costruttore della classe base.
* Il metodo faiVerso() è **sovrascritto (@override)**.

💡 **Senza ereditarietà avremmo dovuto scrivere tutto da zero!**

**3️⃣ Polimorfismo: Stessa interfaccia, comportamenti diversi**

Il **polimorfismo** permette a una classe derivata di comportarsi in modo diverso pur mantenendo la stessa interfaccia.

**Esempio con liste di oggetti polimorfici**

abstract class Animale {

String nome;

Animale(this.nome);

void faiVerso(); // Metodo astratto (deve essere implementato nelle sottoclassi)

}

class Cane extends Animale {

Cane(String nome) : super(nome);

@override

void faiVerso() {

print('$nome dice: Bau!');

}

}

class Gatto extends Animale {

Gatto(String nome) : super(nome);

@override

void faiVerso() {

print('$nome dice: Miao!');

}

}

void main() {

List<Animale> animali = [Cane('Fido'), Gatto('Whiskers')];

for (var animale in animali) {

animale.faiVerso(); // Output diverso a seconda della classe

}

}

📌 **Cosa succede qui?**

* faiVerso() è un metodo **astratto**, quindi le sottoclassi **devono implementarlo**.
* Possiamo gestire **Cane e Gatto con un'unica lista** grazie al polimorfismo.

**4️⃣ Astrazione: Nascondere la Complessità**

L'**astrazione** aiuta a modellare solo le parti importanti di un oggetto, ignorando i dettagli interni.

In Dart possiamo usare **classi astratte (abstract)** e **interfacce (implements)**.

**Esempio: Classe astratta**

abstract class Veicolo {

void accendiMotore(); // Metodo astratto (senza corpo)

}

class Auto implements Veicolo {

@override

void accendiMotore() {

print('Motore acceso: Vroom!');

}

}

void main() {

var macchina = Auto();

macchina.accendiMotore(); // Motore acceso: Vroom!

}

📌 **Cosa succede qui?**

* Veicolo è un'**interfaccia**, Auto **deve implementare tutti i suoi metodi**.
* Possiamo scrivere codice che funziona per tutti i Veicolo, senza sapere esattamente quale veicolo sia.

💡 **Le classi astratte servono per progettare gerarchie robuste senza vincolare l'implementazione.**

**🔹 Altri Concetti Avanzati**

**Mixin: Riutilizzo senza ereditarietà**

I **mixin** permettono di aggiungere funzionalità senza creare una gerarchia complessa.

mixin Volante {

void vola() {

print('Sto volando!');

}

}

class Aquila with Volante {}

void main() {

var aquila = Aquila();

aquila.vola(); // Sto volando!

}

💡 **I mixin evitano i limiti dell'ereditarietà singola!**

**🔹 Conclusioni**

| **Concetto** | **Significato** | **Esempio** |
| --- | --- | --- |
| **Incamapsulamento** | Protegge i dati interni | \_saldo è privato |
| **Ereditarietà** | Riutilizza codice tra classi | Cane extends Animale |
| **Polimorfismo** | Stessa interfaccia, comportamento diverso | List<Animale> animali |
| **Astrazione** | Nasconde i dettagli complessi | abstract class e implements |
| **Mixin** | Riutilizzo senza ereditarietà | with Volante |

Capitolo 28 – Interfacce

**Immagine che contiene schermata, design

Il contenuto generato dall'IA potrebbe non essere corretto.Le interfacce servono per definire un contratto**: una classe che implementa un’interfaccia è obbligata a implementare tutti i suoi metodi e proprietà.  
A differenza di altri linguaggi come Java, **qualsiasi classe in Dart può essere usata come interfaccia**, grazie alla parola chiave implements.

**📌 Differenze tra ereditarietà e interfacce**

| **Concetto** | **Parola chiave** | **Può avere implementazione?** | **Può essere multipla?** |
| --- | --- | --- | --- |
| **Ereditarietà** | extends | Sì, può avere metodi concreti | ❌ No, eredita solo da una classe |
| **Interfacce** | implements | ❌ No, solo dichiarazione di metodi | ✅ Sì, può implementare più interfacce |
| **Mixin** | with | Sì, può avere metodi concreti | ✅ Sì, può combinare più mixin |

**1️⃣ Creare e Implementare un’Interfaccia**

Un’interfaccia è **qualunque classe** che viene implementata con implements.

**Esempio base: Interfaccia classica**

abstract class Animale {

void faiVerso(); // Metodo senza implementazione

}

class Cane implements Animale {

@override

void faiVerso() {

print('Bau!');

}

}

void main() {

var fido = Cane();

fido.faiVerso(); // Bau!

}

📌 **Cosa succede qui?**

* Animale è una **classe astratta**, che funge da interfaccia.
* Cane **deve implementare tutti i metodi dichiarati in Animale**.

💡 **Se una classe implementa un’interfaccia, è obbligata a definire tutto quello che c’è dentro!**

**2️⃣ Interfacce con più proprietà e metodi**

Un’interfaccia può avere **proprietà e metodi senza corpo**.

**Esempio: Interfaccia con metodi e proprietà**

abstract class Veicolo {

int get velocitaMassima;

void accendiMotore();

}

class Auto implements Veicolo {

@override

int get velocitaMassima => 200;

@override

void accendiMotore() {

print('Motore acceso! Vroom Vroom!');

}

}

void main() {

var ferrari = Auto();

print(ferrari.velocitaMassima); // 200

ferrari.accendiMotore(); // Motore acceso! Vroom Vroom!

}

📌 **Cosa succede?**

* Veicolo definisce un **getter** (velocitaMassima) che le classi implementanti devono fornire.
* Auto implementa sia la proprietà che il metodo.

💡 **Le interfacce garantiscono che tutte le classi implementanti rispettino una struttura precisa.**

**3️⃣ Implementare Più Interfacce**

A differenza dell’ereditarietà (extends), **una classe può implementare più interfacce**.

**Esempio: Multipla implementazione**

abstract class Volante {

void vola();

}

abstract class Nuotante {

void nuota();

}

class Anatra implements Volante, Nuotante {

@override

void vola() {

print('L’anatra vola!');

}

@override

void nuota() {

print('L’anatra nuota!');

}

}

void main() {

var anatra = Anatra();

anatra.vola(); // L’anatra vola!

anatra.nuota(); // L’anatra nuota!

}

📌 **Cosa succede qui?**

* Anatra **implementa due interfacce** (Volante e Nuotante).
* Deve fornire il corpo per **tutti i metodi dichiarati nelle interfacce**.

💡 **Questo è utile quando vuoi che una classe erediti comportamenti da più fonti.**

**4️⃣ Differenza tra implements e extends**

Molti fanno confusione tra implements ed extends.  
Vediamo la **differenza pratica**:

abstract class Animale {

void faiVerso() {

print('Verso generico');

}

}

class Cane extends Animale {

// Eredita il metodo faiVerso(), ma può sovrascriverlo

@override

void faiVerso() {

print('Bau!');

}

}

class Gatto implements Animale {

@override

void faiVerso() {

print('Miao!');

}

}

void main() {

var fido = Cane();

var whiskers = Gatto();

fido.faiVerso(); // Bau! (eredita e sovrascrive)

whiskers.faiVerso(); // Miao! (implementa tutto da zero)

}

📌 **Differenze:**

* Cane extends Animale: **eredita i metodi esistenti** e può modificarli.
* Gatto implements Animale: **deve scrivere tutto da zero**, anche se Animale avesse implementazioni.

💡 **Se vuoi riutilizzare il codice, usa extends. Se vuoi solo definire una struttura, usa implements.**

**5️⃣ Interfacce e Dependency Injection**

Le interfacce sono spesso usate per l’**inversione di controllo (IoC)** e la **dependency injection**, utile nei progetti grandi.

**Esempio: Dipendenza con interfaccia**

abstract class ServizioPagamento {

void paga(double importo);

}

class PayPal implements ServizioPagamento {

@override

void paga(double importo) {

print('Pagato $importo con PayPal');

}

}

class CartaDiCredito implements ServizioPagamento {

@override

void paga(double importo) {

print('Pagato $importo con Carta di Credito');

}

}

class Negozio {

final ServizioPagamento pagamento;

Negozio(this.pagamento);

void checkout(double totale) {

pagamento.paga(totale);

}

}

void main() {

var negozio1 = Negozio(PayPal());

negozio1.checkout(100); // Pagato 100 con PayPal

var negozio2 = Negozio(CartaDiCredito());

negozio2.checkout(50); // Pagato 50 con Carta di Credito}

📌 **Cosa succede qui?**

* ServizioPagamento è un’interfaccia.
* PayPal e CartaDiCredito la implementano con comportamenti diversi.
* Negozio riceve un ServizioPagamento, senza sapere quale sia.

💡 **Questo è il principio SOLID di "Dipendenza sulle astrazioni e non sulle implementazioni".**

**🔹 Riepilogo e Conclusione**

| **Concetto** | **Descrizione** | **Parola chiave** |
| --- | --- | --- |
| **Interfaccia** | Definisce un contratto per le classi | implements |
| **Ereditarietà** | Riutilizza codice di una classe base | extends |
| **Multipla implementazione** | Una classe può implementare più interfacce | ✅ Sì |
| **Uso in progetti grandi** | Struttura per dependency injection | ✅ Sì |
| **Differenza con mixin** | Mixin aggiunge codice riutilizzabile | with |

**Corso**

**Dart**

**Dedalo’s notes – Una serie di appunti da non prendere sul serio**

Serie di appunti sugli argomenti più svariati, fatti per imparare, apprendere e divertirsi, navigando per il magico mondo del…tutto!

Ogni errore o imprecisione sarà sistemata, oppure no 😊.

Attenzione, può contenere boiate, amenità o stramberie!

https://github.com/GiacomoBorsellino

Scritto da

**Giacomo Borsellino**

27-12-2024