Usare Flutter

tratto da:

https://docs.flutter.dev/get-started

Marino Segnan

Percorso globale Flutter

- Uso di Android studio e una prima occhiata a ‘Hello World’
- Spiegazione minimale del linguaggio Dart
- Creazione delle pagine di UI Layout dell’ App
- Navigazione tra le pagine di un’ App
- Gestione dello stato: semplice col package **getx**
- Accesso a risorse di rete in generale
- Cenni sull’accesso ai dati
- Tralasceremo molti dettagli: adattamento al singolo dispositivo, internazionalizzazione…
- L’attenzione e’ concentrata sugli aspetti peculiari di Flutter mentre non esamineremo i lati in cui e’ identico ad altri Framework

Passo 1: installare il plugin Flutter in Android Studio

- Flutter funziona con qualunque Editor piu' tool command line, ma noi useremo Android Studio.
- Cercate "download android studio", fate l'installazione per il vostro OS.
- poi da "File->Settings->Plugins->Marketplace" cercate ed installate "**flutter**" (puo' variare in base alla versione)
- Questo installa anche il linguaggio "**Dart**"

Passo 2: installare Flutter su PC

1. scaricate da https://docs.flutter.dev/get-started/install e seguite le istruzioni
2. importantissimo:
   non installate niente nelle cartelle di **sistema, es C:\Programs,** perche' richiede privilegi, **e non installate** nulla in cartelle dove il nome contiene **spazi o caratteri speciali**

Passo 2: installare Flutter su PC

lanciate poi il comando:

-> **flutter doctor**

che verifica la presenza di tutti i componenti.

se vedete un errore del tipo:

“`Android SDK is missing command line tools; download from https://goo.gl/XxQghQ`”

(Aprite Studio) Settings, cercate “sdk”, selezionate “sdk tools”, poi: check sulla terza box:

“Android sdk command line tools”

e riprovate **flutter doctor**

Errori flutter doctor

- Se il comando da' componenti mancanti, non tutto e' perduto.
- Ad esempio su Windows potrebbe dire che manca VScode, su IOS manca Xcode, ma noi usiamo android studio, quindi si puo' ignorare.
- provate a creare un progetto flutter come nelle slide successive, accettando tutti i default.
- Se viene generato correttamente, provate anche solo a farlo girare su Chrome.
- Inizialmente e' sufficiente.
- In parallelo, fuori dalla lezione, provate a eliminare gli altri errori.

Tools per Flutter

- mentre per Android nativo esiste praticamente solo Studio,
- se Studio non vi funziona, potete usare:
  VScode, Xcode, (che io non conosco)
- Oppure la linea di comando da terminale:

  a. <cd alla directory dove creare cartella>
  b. flutter create myproject
  c. cd myproject
  d. flutter run
  e. seguire le istruzioni

Tools per Flutter

- da terminale e' banale, ma non avete il supporto gestione del progetto, per editare i sorgenti dovete usare un qualunque editor di testo, che spesso vi da' un po' syntax highlighting
- ci sono molte opzioni per la command line, usando –h vi da' un lungo help
- per farlo girare su un device specifico, usare gli id che vi mostra:
- flutter run -d linux | chrome | 1ebd2679 …..
- notate che esiste una suite di tools (devtools) , andate al link che vi fornira' quando fate 'flutter run':
- The Flutter DevTools debugger and profiler on sdk gphone x86 is available at:
- http://127.0.0.1:9100?uri=http://127.0.0.1:37893/KOpTyNUCQk4=/

incollare nel browser qualcosa di simile

Primo Progetto (da Studio)

- File->new-> New Flutter project
- Si apre una nuova finestra, riselezionate “Flutter” a sx se non lo fosse gia’!!!!
- scegliete un nome a piacere, tipo “myapplication”,
- noterete che potete scegliere a sx per quale sistema generare l’App, lasciateli pure tutti attivi

Primo Progetto

- dovreste scegliere il Linguaggio per la parte Android, che di default e' Kotlin e non Java. Dopo e' difficile cambiarlo ma per il nostro corso non interessa
- Se per caso vi dice che Dart non e' configurato, andate in File->settings->'languages and frameworks'->**Flutter** e inserite il pathname del flutter sdk, quello sistema anche Dart.
- Se appare in alto una riga gialla tipo: "pub get has not been…" , voi cliccate su "**upgrade dependencies**"

Esecuzione su vari sistemi

- identica ad una App Android nativa su emulatore o smartphone
- potete pero’ scegliere se eseguire su un device Android o ad esempio su Chrome.
- per fare semplice testing del comportamento della GUI puo’ essere piu’ leggero provare su Chrome, anzi quasi raccomandabile (grazie al resize)
- Se invece dovete testare il comportamento globale, con un server remoto o affini, allora dovete verificare sul sistema effettivo, perche’ a volte vi sono differenze nei permessi di accesso/ security etc.

Esecuzione su Chrome

- Potete provare a lanciare come App web selezionando “Chrome(web)” anziche’ emulatore
- Anche se il nostro target ufficiale e’ Android, conviene anche testare su Chrome perche’ possiamo fare il resize della finestra e vedere come si comportano i vari Widget al cambiare delle dimensioni/rapporto lunghezza -altezza

Esecuzione su smartphone

- Collegate al PC il telefono col cavo USB
  - A seconda delle versioni/modello Android vi chiedera' se autorizzate l'accesso dal PC, dovete vedere voi
  - in Android Studio, apparira' il nome del telefono.
- cliccate il solito triangolo verde e verra' eseguito

Debug/ Hot reload (simili da command line)

- cliccate sullo spazio bianco a fianco del codice
  - appare un pallino rosa
- cliccate sull'insetto a fianco del triangolo verde
  - parte il debugger e si fermera' sul pallino rosa
- per fare Hot Reload, cliccare sul fulmine giallo:
  - il programma ricarica quasi istantaneamente lo stato
- Notate che il programma **non sempre** e' reinizializzato completamente:
  - se avete dubbi fate **stop** e **run** da capo per essere sicuri

un esempio di confronto col nativo

- estraiamo per dopo dallo zip su Moodle la cartella HelloAndroid
- contiene un esempio di progetto Android nativo che e' identico in funzionalita' a quello di Flutter e ci permettera' di vedere le differenze
- contiene solo un FloatingActionButton che se premuto aggiorna un contatore
- Lo scorriamo velocemente, conosciamo gia' la struttura generale del progetto

Struttura del progetto Flutter

- Ci sono cartelle comuni a tutte le versioni e cartelle specifiche per ogni OS.
- quella comune e' a livello 0 e contiene il file di configurazione "**pubspec.yaml**" che elenca le librerie da usare ed altre impostazioni generali
- altra comune e' "**lib**", che contiene i file sorgente dell' App
  - contiene **main.dart**, il sorgente iniziale del progetto
- le altre sono le cartelle "ios", "android",.... che contengono le impostazioni specifiche per i vari OS
- la cartella "**assets**" e' equivalente a "**res**" nativo

Avvertenze Android studio

- almeno su Linux, l' emulatore da' problemi con Flutter, mentre tutto funziona se provate con cavetto e smartphone.
- se ricevete un messaggio tipo "manca memoria", spegnete l'emulatore e scegliete dalle opzioni: "**cold boot now**"
- la cartella **android** del progetto e' evidenziata in rosso con errori di compilazione, che fortunatamente non danno problemi.
- se dovete modificarla, RightClick -> flutter-> open module in android studio.

Flutter usa Dart !

Dart e’ simile in sintassi a Java, C++ etc
Per ora, quando guardiamo il codice del progetto di partenza, lo scorriamo solo senza pretendere di capire tutto immediatamente.
Lo scopo e’ farci un’idea sommaria della struttura.
Appena terminata questa fase, faremo una panoramica delle caratteristiche del linguaggio

Il codice

Momentaneamente non guardiamo il codice generato ma eseguiamo il codice della slide successiva che e' semplificato.
Per fare questo:
1) lo copiamo.
2) in Studio new -> Dart File (scegliete un nome a caso, prova..)
3) incolliamo
4) right click -> run <nome scelto prima>

Il codice da copiare

```
import 'package:flutter/material.dart';
void main() {
  runApp(const MyApp());
}
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home:
      Center(
        child: Text('Ciao'),
      ),
    );
  }
}
```

Il codice

- il main() e' l'entry point
- abbiamo una class Widget (tutto e' un Widget) che **deve** implementare il metodo build().
- build() restituisce il contenuto del Widget:
- in questo caso scegliamo una App stile Google (MaterialApp) il cui contenuto (la schermata Home) e' un semplice Widget di testo, Centrato.
- Se togliete il Widget Center(), il testo finisce in alto!
- per rieseguire in fretta, cliccate sul lampo giallo in alto a dx , il cosiddetto "Hot Reload"

Linguaggio Dart

- Scorreremo brevemente i principali concetti di Dart
- Come per qualunque linguaggio, l'unico modo di impararlo bene e' usarlo
- Il nostro scopo e' solo quello di scrivere e capire programmi per Flutter
- A parere di molti, se si conosce un altro linguaggio, Dart diventa utilizzabile in poche ore.
- Se lo userete per general programming dovrete prima o poi dedicare qualche ora ad impararlo piu' organicamente

Documentazione Dart

- documentazione estesa su: https://dart.dev/guides

Per iniziare, guarderemo in breve solo alcune cose che ci servono per Flutter, tratte da qui:
https://dart.dev/samples
https://dart.dev/codelabs/dart-cheatsheet

In un secondo tempo potete per conto vostro approfondirne la conoscenza

Introduzione a Dart

- Dart e' Strongly-Typed come Java
- Per chi conosce Python puo' sembrare un ibrido con Java
- Cerca di essere coinciso e ridurre al minimo il codice **Boiler-Plate** e permette in alcuni casi una maggiore flessibilita' rispetto a Java.
- Ha tratto concisione da linguaggi come Python
- Possiede alcuni costrutti che semplificano la programmazione **concorrente**
- Esistono alcune aree che tuttavia contengono delle complessita' sintattiche, in particolare quella dei costruttori e delle inizializzazioni degli Oggetti.

Python, Java, Kotlin, Dart…?

- Tutti questi linguaggi purtroppo col passare del tempo tendono ad “arricchirsi” di costrutti, di sintassi, di keyword ed opzioni, perdere ironicamente leggibilita’ e la curva di apprendimento per l’uso ottimale tende a essere sempre piu’ lunga.
- Python in proporzione , pur essendo il piu’ antico, e’ rimasto relativamente comprensibile.
- Tutto male? No per fortuna. In particolare Kotlin e Dart hanno dei costrutti che semplificano la programmazione concorrente

Per le prove Dart

potete fare in tanti modi tra cui:

- create un <miosorgente>.dart in un progetto Android Studio e mette le print() che appariranno su console

- per altre prove, usate Dartpad: https://dartpad.dev/

Struttura di un modulo

Un modulo contiene **definizioni** di variabili, classi, funzioni (es il main())
Le funzioni e le variabili si possono definire fuori dalle classi:

```
void miafunzione() {
      print('ciao');
       }

int num = 12;
class MiaClasse{   }
```

Invece gli **statement non** si possono eseguire fuori da una funzione

```
 miafunzione();   // NON SI PUO'; solo definizioni a livello 0, non statements
 int num =32; // ok
num = 33;   // errore: ridefinizione
```

Null safety

- In Dart 2 di default le variabili non sono **mai** NULL
- Se voglio che possano essere NULL, lo devo dichiarare esplicitamente aggiungendo **?**

```
String s1 = 'ciao';  // sempre diversa da NULL

String? s2;  // annullabile
```

Null safety

```
String s1 = 'ciao';
s1 = null;  // errore, non puo' contenere NULL
String? s2 = 'ciao';  // annullabile
s2 = null; // ok, e annullabile
String s3; // non inizializzata
s2 = s3;  // errore s3 non e' inizializzata
s3 = null;  // errore ovviamente
String? s4;
s2 = s4; //ok assegno tra due annullabili
s1 = s4; // errore assegno a non nullable
var a;     // una var puo' contenere NULL, e non si
puo' fare var?
a = 'ciao';  // ok
a = null;      // ok
```

Identificatori

- in Dart c'e' la "**type inference**" applicata quando possibile.
- **Non** esistono **public, private, protected**, se una variabile e' privata a un modulo, si prefissa con **'_'** ma e' solo una convenzione
- Gli identificatori devono essere dichiarati usando un insieme di attributi : **<type>**, '**const**', **'final'** , **'var'**,**'dynamic'**.
- Tutto questo per facilitare il lavoro del compilatore e rendere piu' facili le ottimizzazioni

Identificatori

- **const** : e’ una costante calcolabile compile-time
- **final** : inizializzato una sola volta, ma il contenuto puo’ variare se e’ un oggetto complesso
- **var** : il compilatore determina il tipo
- **dynamic** : il compilatore non fa type-checking, rimandato a run-time
- **<Type>** : sostituite voi il tipo: int, String, Object, ……

Esempi di dichiarazioni

```
void funzione() {
 dynamic c = 10;
 c = 'ciao';  // ok
 var b = 20; // inferenza: dichiaro un 'int'
 b = 'ciao';  // errore
 var a;
 a = 10;
 a = 'ciao';  // ok!! come dynamic se non la inizializzo
 int d = 5;
 d = 'ciao'; // ovviamente errore
 final  List<int> ar = [0,1,2,3,4];
 ar.add(10);   // ok modificare il contenuto
 ar = [0,1,2,3,4,5]; // errore, cerco di riassegnarla
 const List<int> ar1 = [0,1,2,3]; // tutto l'interno e' costante
 ar1.add(10);   // errore fuorviante?:   'unsupported operation'
}
```

Funzioni in Dart

// notazione normale

```
bool isNoble(int atomicNumber) {
  return _nobleGases[atomicNumber] != null;
  }
```

// senza tipo parametro funziona lo stesso

```
isNoble(atomicNumber) {
  return _nobleGases[atomicNumber] != null;
  }
```

// per una funzione costituita da una sola **espressione** si puo' usare la notazione ⇒

```
bool isNoble(int atomicNumber) => _nobleGases[atomicNumber] != null;
```

Parametri delle Funzioni in Dart

//in Dart 3 tipi di parametri:
prima i posizionali,
poi seguiti da parametri ‘optional’ o ‘named’ (NON entrambi).
(il’?’ permette valore NULL)

```
void enableFlags({bool bold = false, required bool? hidden}) {}
```

```
i parametri ‘named’ sono racchiusi tra { }
il primo e’ ‘named’, opzionale con default,
il secondo e’ ‘named’, obbligatorio, anche NULL

al posto dei named posso avere parametri posizionali opzionali
racchiusi tra [ ]

// uno obbligatorio, secondo opzionale, anche null
void funz(param1, [int? param2]){}
```

Esempio funzione con named parameters

```
void func({required p ,q=0}) {
  print(p+q);
}
void main(){
func(p:3,q:8);
func(6);
      }
```

In Flutter moltissimi Widget hanno solo parametri ‘named’, di cui alcuni magari **required**

scorrete col mouse ad esempio sul costruttore di **Scaffold**

Classi

```
class worker{
  int _hours;
  double _pay;
  worker(this._pay,[ this._hours = 0]) {
  }

  double get earning => _hours*_pay;
  void set ore(int a) => _hours = a;
}
```

opzionale tra [ ]

- gli attributi sono accessibili tramite shortcut **get** e **set**

get e **set** possono essere funzioni con logica aggiuntiva e anche simulare la presenza di attributi

Classi

- per creare un oggetto:

```
var p = worker(10); // prima era: new worker(10)
p._hours = 8;
p.ore = 5;
print(p.earning);
```

- nella versione Dart 2 **‘new’** e’ opzionale
- hanno cercato di ridurre la ‘verbosita’ del linguaggio

Classi

- i costruttori hanno parametri di default, obbligatori, varie opzioni e sono una delle parti piu' elaborate del linguaggio
- tutti gli attributi devono essere inizializzati o nel costruttore o nella loro definizione
- ottimo riassunto:
- https://stackoverflow.com/questions/52449508/constructor-optional-params

riassunto del riassunto

- il costruttore (come una funzione) puo' avere diverse forme:
  - una sequenza di normali parametri posizionali
- a seguire:
  - una sequenza di parametri opzionali racchiusa tra [ ]
- oppure:
  - una sequenza di parametri 'named' racchiusa tra { }
  - i 'named' possono essere resi obbligatori con **required**

prima del body del costruttore si possono inizializzare altri attributi

costruttore con nome

```
class MyColor {
  int red,green,blue,_trasp=0;

  MyColor({required this.red, required this.green,  this.blue = 128}):_trasp=255;
  // named constructor

// posso definire costruttori aggiuntivi

  MyColor.coloridue({required this.red, required this.green}):blue=128{
    print('usocoloridue');
  }
}

var color = MyColor(red: 80, blue: 80,green: 80);
var color2 = MyColor.coloridue(red: 80, green: 80);
```

Factory constructor

la Keyword **factory** definisce un costruttore che non deve creare necessariamente una nuova istanza ma puo' restituirne una fissa , ad esempio

```
class MyColor {
  int red,green,blue;

  static MyColor gray = MyColor(red:128,green:128);

  MyColor({required this.red, required this.green,  this.blue = 128});

  factory MyColor.grigio(){return gray;}
}

MyColor c = MyColor.grigio();
```

flussi di controllo

il flusso di controllo e' il solito, come in Java, ci sono for while if…..

```
for (final object in workers) {
  print(object);
  }
```

Le Funzioni sono veri **Oggetti**

Sono usabili come parametri ed assegnabili a variabili

```
int f2(num){
 return num+3;
}
int funz(param1){
 return param1(4);
}
var id = f2;
print(funz(f2));
```

Le funzioni hanno un tipo derivato da **Function**

Le classi invece sono a se' stanti

Un identificatore di una classe invece e' un tipo vero e proprio e non puo' essere considerato un oggetto come tutti gli altri:

```
class C {
  static int call(x){return x;}
  C(){}
}

void ff() {
  var cc = C();
  print(cc(2));  // chiama call() sulla istanza
  var ee = C;
  var bb = ee(); // errore
  var dd = ee.call(2);  // errore, call() e' definito per le istanze ma non per il tipo
}
```

funzioni anonime (dette **lambda** in altri linguaggi)

e' una definizione di funzione ma manca il nome, si puo' usare come parametro o assegnare a una variabile:

<nome assente> `(parametri) => {statement}`

```
  var mialambda = (item) => item + 1;
  var mialambdae = (item) {return item - 1;}; // altra sintassi
  var ds = () => mialambda;
  print(mialambda(23)+mialambdae(16));
  print(ds()(33));
```

```
// java: int mialambda(int item){ return item+1;}
```

Lazy Initialization

- le variabili non-nullable devono essere inizializzate prima di essere usate
- la keyword **late** indica al compilatore che la variabile non e' inizializzata alla dichiarazione, ma lo sara' a runtime prima del suo uso.

```
class Weather {
  late int _temperature = _readThermometer();
}
```

- In questo caso viene inserito un check nel codice, per cui la inizializzazione viene fatta solo **SE** la variabile sara' usata

Lazy Initialization

- inoltre, con **late** si puo' accedere ai campi di **this** (l'oggetto) perche' a quel punto esso e' gia' stato inizializzato

```
class Point {
 double x = 0;
 double y = 0;
 late double z = x*y; // senza late errore
 Point(this.x, this.y);
}
```

espressioni condizionali

Sono shortcut sintattici equivalenti a quelli di Java:

```
var res = 10 > 15 ? "Greater" : "Smaller";
print(res);
var n1;
var n2 = 15;
var res2 = n1 != null ? n1 : n2;
var res1 = n1 ?? n2; // preferita
print(res1);
print(res2);
```

accesso condizionale

shortcut per testare se una variabile e' null prima di accedere ad un a suo attributo

```
var myObject;
var result = (myObject != null) ? myObject.someProperty :
null;
var result1 = myObject?.someProperty;
 // in cascata
var result2 = myObject?.someProperty?.someOtherProperty;
```

metodi in cascata tutti invocati sullo stesso oggetto

Se eseguo:

```
myObject.someMethod()
```

il valore sara' quello restituito da someMethod(), ma se scrivo:

```
myObject..someMethod()
```

il valore restituito sara' myObject. Posso fare quindi:

```
myObject..someMethod()
         ..someMethod1()
         ..someMethod2()    // etc etc
```

Strutture dati

Dart permette di definire Liste, Insiemi, Dizionari

```
final aListOfStrings = ['one', 'two', 'three'];
final aSetOfStrings = {'one', 'two', 'three'};
final aMapOfStringsToInts = { // Map sta per Dizionario
 'one': 1,
 'two': 2,
 'three': 3,
};
final aListOfInts = <int>[];
final aSetOfInts = <int>{};
final aMapOfIntToDouble = <int, double>{};
```

NB in Dart si usa sempre List, che e' modificabile, non ho trovato un vero Array tradizionale

Strutture dati

NB in Dart si usa sempre **List**, che e' modificabile, non ho trovato un vero Array tradizionale e funziona come la **ArrayList** in Java

```
List<int>? arr1 = [0,1,2,3];  // nullable
var arr = [0,1,2,3];  // equivalente not nullable
arr.insert(3,34);
print(arr);
arr1 = null;
arr = null; // errore
```

Supporto per Asincronia in Dart

future, async, await
scrivo codice che sembra sincrono in apparenza
Esistono poi i generatori, come in Python.
esistono sia generatori sincroni che asincroni.
restituiscono valori con ‘yield’
gli ‘stream’ sono sorgenti di dati asincrone.

Future

Una funzione puo’ restituire un valore **Future**, questo e’ un tipo di risultato non ancora pronto, che deve essere completato in seguito.
quando la funzione viene lanciata in esecuzione:
ritorna immediatamente un valore **Future.**
qualcuno in seguito completera’ il **Future** col valore di ritorno effettivo
Se l’operazione Future completa senza valori utili, e’ <void>

```
Future<void> fetchUserOrder() {
 return Future.delayed(const Duration(seconds: 2), () =>
print('Large Latte'));
}

void main() {
 fetchUserOrder();  // non aspettiamo !
 print('Fetching user order...'); // stampa prima questo
}
```

async ed await

per aspettare il risultato si usa **await**
per usare **await**, la funzione **deve** essere tipata **async**

```
Future<void> fetchUserOrder() {  // invariata!
 return Future.delayed(const Duration(seconds: 2), () =>
print('Large Latte'));
}

void main () async {   // aggiungo async perche' uso await
 await fetchUserOrder(); // aspetto! stampera' prima questo
 print('Fetching user order...');
}
```

await e **async**

Il Future prima o poi entrera' o in stato 'completed' o 'error'

```
Future<void> fetchUserOrder() {
// se per caso si verifica un errore
 return Future.delayed(const Duration(seconds: 2), ()
=> throw Exception('Logout failed: user ID is invalid'));
}

void main () async {
 await fetchUserOrder(); // aspettiamo
 print('Fetching user order...');
}
```

completare un future

abbiamo visto che un **future** si completera' con un valore, in generale dobbiamo noi stessi invocare la complete()

```
Future <dynamic> someFutureResult()  {
 final c = new Completer(); // contiene il future
 new Timer(Duration(seconds: 2), () {
   print("this line is printed after 2 seconds");
   c.complete("you should see me final");
 });
 return c.future;
}
```

await e **then()**

- await blocca l'esecuzione fino al completamento.
- se voglio maggior controllo uso **then**()

```
void main()  {
 Future <dynamic> res = someFutureResult();
 res.then((result) => print('$result'));
 print("you should see me first");
}
```

posso usare esplicitamente il Future<T> restituito dalla funzione, se scrivo <Future>.then(...) blocco solo il **ramo** relativo, posso installare error handler etc etc.

await e **then**

Due versioni simili, una blocca tutto e una solo un ramo

```
void main()  {
 Future <dynamic> res = someFutureResult();
 res.then((result) => print('$result'));
 print("you should see me first");
}

void main()  async {
 String res = await someFutureResult();
 print('$res');
 print("you should see me later");
}
```

Thread (flussi) in Dart

Non esistono.

- per eseguire codice in parallelo, si usa un **isolate**
- un **isolate** e' completamente autonomo, ha la sua area di memoria privata,
- **non** condivide memoria con gli altri isolate, cosi' non ci sono conflitti su risorse condivise che generano errori ansiogeni.
- ogni **isolate** ha il suo event loop
- Se ci sono piu' CPU, gli isolate possono girare in parallelo

Asincronia in Flutter

- in Flutter gli isolate servono sui dispositivi mobili per non bloccare l'interattivita' della UI.
- Una operazione lenta potrebbe bloccare le animazioni dello schermo o comunque alterare la regolarita' dell'interazione (come si diceva per Android nativo)
- le operazioni lunghe sono quindi delegate a degli **isolate**, che tramite il meccanismo **future .. await** al completamento restituiranno il risultato
- Nella maggioranza dei casi gli **isolate** non si usano esplicitamente ma sono eventualmente implementati dai plugin che usiamo

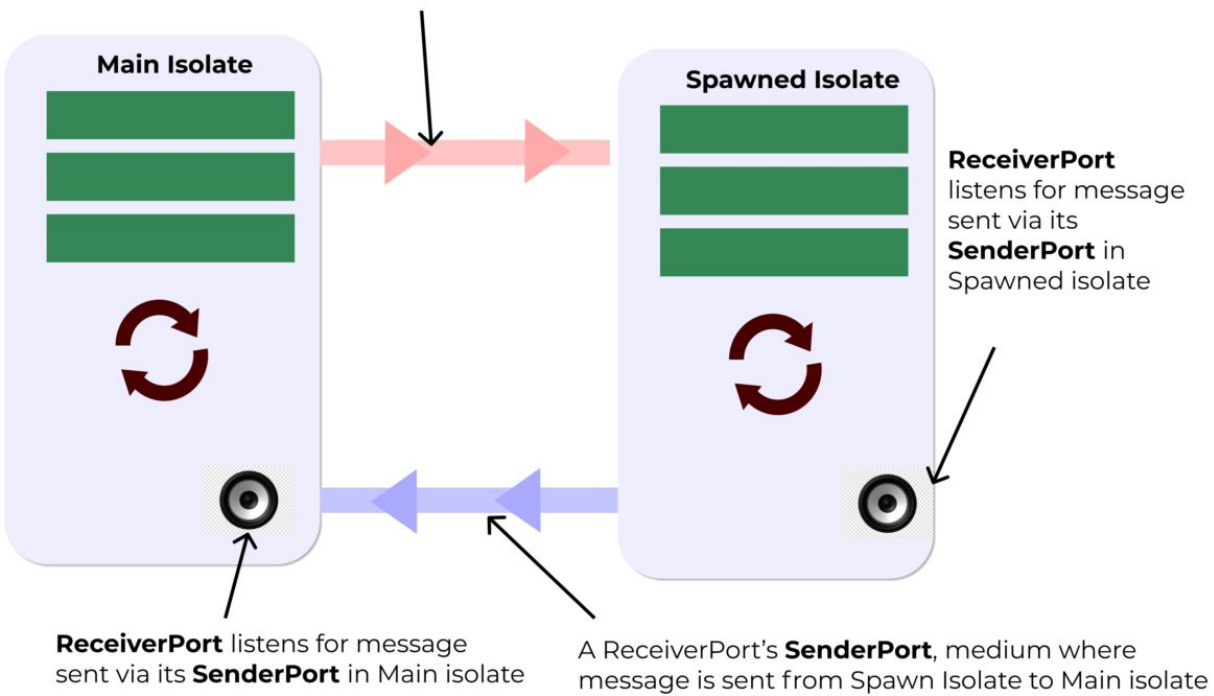
Isolate

- inizialmente , un programma Dart gira in un solo isolate, quello che esegue main()
- un isolate si puo' istanziare, gli si passa una funzione da eseguire tramite **Isolate.spawn()**
- Gli isolate possono scambiare messaggi usando **send()** tramite **SendPort** e **ReceivePort**
- gli isolate possono trasmettere Oggetti, ma non tutti, occorre leggere la documentazione della **send()**
  - Oggetti di classi utente contenenti tipi di base di solito si possono trasmettere.

Riassunto

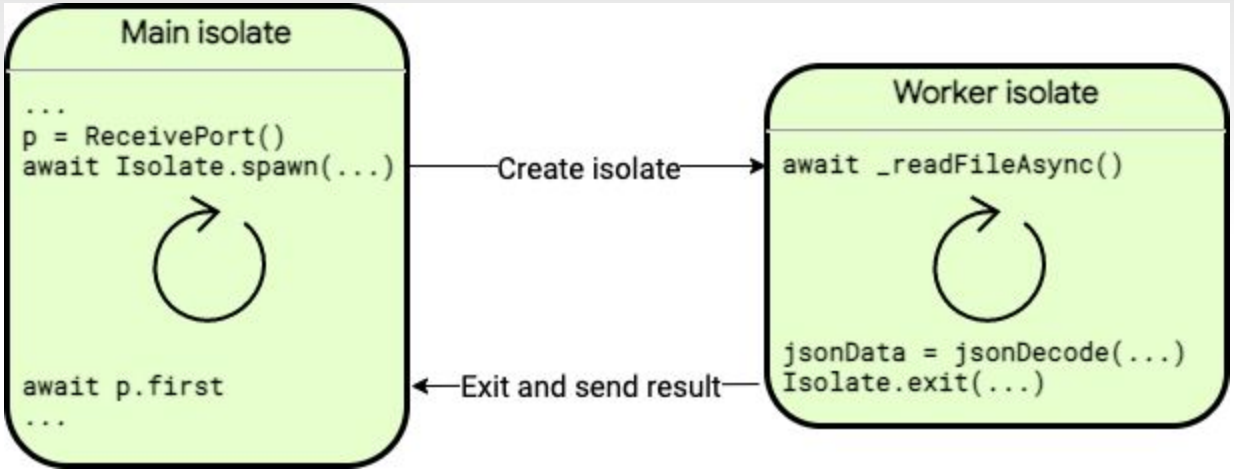
- il **main** crea un altro isolate tramite **spawn()** comunicandogli una **port** a cui ricevera' messaggi e si mette a fare una **listen**()
- **l'isolate** gli risponde (a quella **port**) comunicandogli a sua volta a quale **port** ricevera' messaggi
- la comunicazione e' stabilita
- ogni isolate usa la **listen**() per ascoltare i messaggi successivi e **send**() per rispondere, realizzando una sorta di **EventLoop**
- ogni isolate puo' crearne altri

comunicazione tra isolate



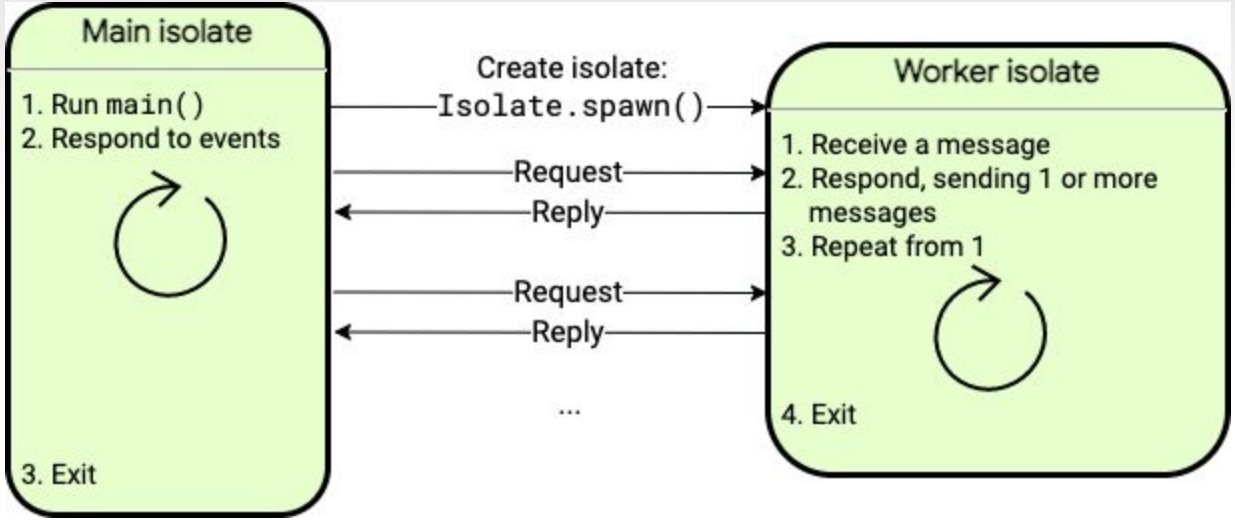
Isolate

Creare un isolate per eseguire una singola operazione lenta



Isolate

Scambio ripetuto di messaggi fra due isolate



Dart 3

Il linguaggio e' in evoluzione ed estensione, guardate ad esempio:

https://codelabs.developers.google.com/codelabs/dart-patterns-records#0

Sono concetti simili a quelli di Python

Sono feature sofisticate, ma esulano dall' uso di dart che facciamo in questo corso.