UNIVERSITÀ
DEGLI STUDI DI TRIESTE

Corso di Laurea in Ingegneria Clinica e Biomedica
Informatica Medica I

FLUTTER PER MOBILE DEVELOPMENT

Prof. Sara Renata Francesca Marceglia

INSTALL FLUTTER (ON MAC)

- Download flutter seguendo le istruzioni sulla pagina https://flutter.dev/docs/get-started/install/macos
- Modifica della variabile d'ambiente:
  - Aprire il terminale
  - Inserire il comando vim .bash_profile (apre il file bash_profile)
  - Premere «i» per attivare il comando di insert nel file
  - Inserire la riga

  export PATH="$PATH:[*PATH_TO_FLUTTER_GIT_DIRECTORY*]/flutter/bin"

  Inserendo al posto della […] il path alla cartella flutter
  - Premere esc seguito da :wq!
  - Chiudere il terminale e riaprirlo
  - Verificare che sia andato a buon fine il set del path digitando il commando

    flutter --version

INSTALL FLUTTER (ON WINDOWS)

- Download flutter seguendo le istruzioni sulla pagina https://flutter.dev/docs/get-started/install/windows
- Modificare la variabile ambientale “path”
  - Da pannello di controllo → Sistema→impostazioni avanzate → variabili d’ambiente
  - Aggiungere una nuova variabile [*PATH_TO_FLUTTER_GIT_DIRECTORY*]/flutter/bin
  - Inserendo al posto della […] il path alla cartella flutter
  - Aprire il prompt dei comandi
  - Verificare che sia andato a buon fine il set del path digitando il commando

    flutter --version

INSTALL ANDROID STUDIO

- Android Studio, version 3.0 or later
- Install the Flutter and Dart plugins:
  - Start Android Studio.
  - Open plugin preferences (**Configure > Plugins** as of v3.6.3.0 or later).
  - Select the Flutter plugin and click **Install**.
  - Click **Yes** when prompted to install the Dart plugin.
  - Click **Restart** when prompted.

TIPOLOGIE DI APPLICAZIONI MOBILE

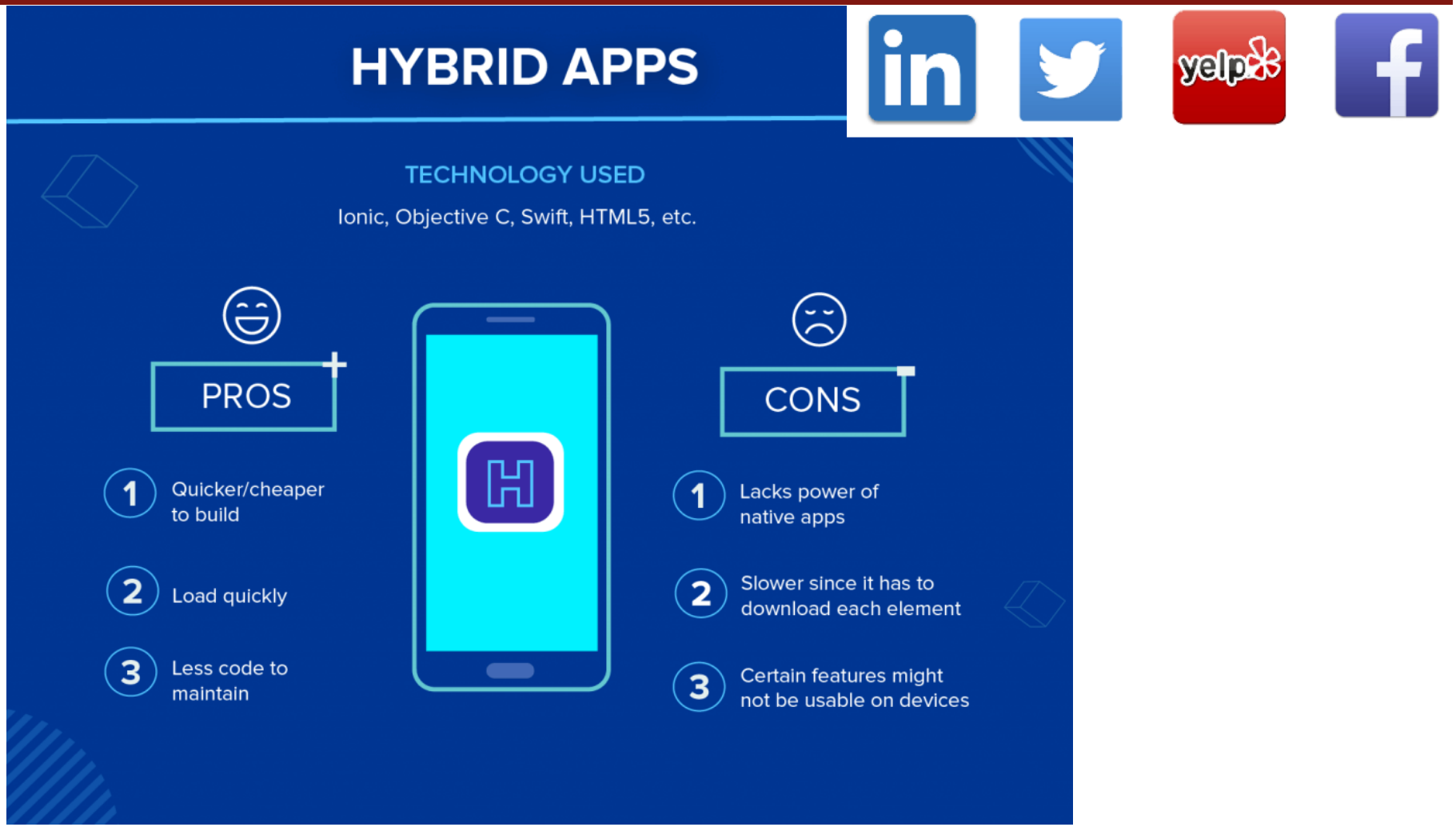
NATIVE APPS


NATIVE APPS
TECHNOLOGY USED
Java, Kotlin, Python, Swift, Objective C, etc.
PROS
1 Faster, better performance
2 Native UI
3 Can access device features
CONS
1 Higher cost to maintain
2 Takes up space in the device
3 Updates must be downloaded

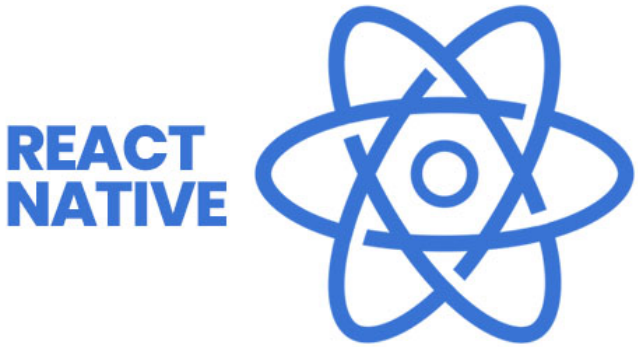
WEB APPS


WEB APPS
TECHNOLOGY USED
HTML5, CSS, JavaScript, Ruby, etc.
PROS
1 Web-based so performs on all devices
2 Easier to maintain
3 Users don't run out of storage
CONS
1 Dependent on a browser
2 Needs an internet connection
3 May not always integrate with device hardware

HYBRID APPS


HYBRID APPS
TECHNOLOGY USED
Ionic, Objective C, Swift, HTML5, etc.
PROS
1 Quicker/cheaper to build
2 Load quickly
3 Less code to maintain
CONS
1 Lacks power of native apps
2 Slower since it has to download each element
3 Certain features might not be usable on devices

CROSS-PLATFORM DEVELOPMENT

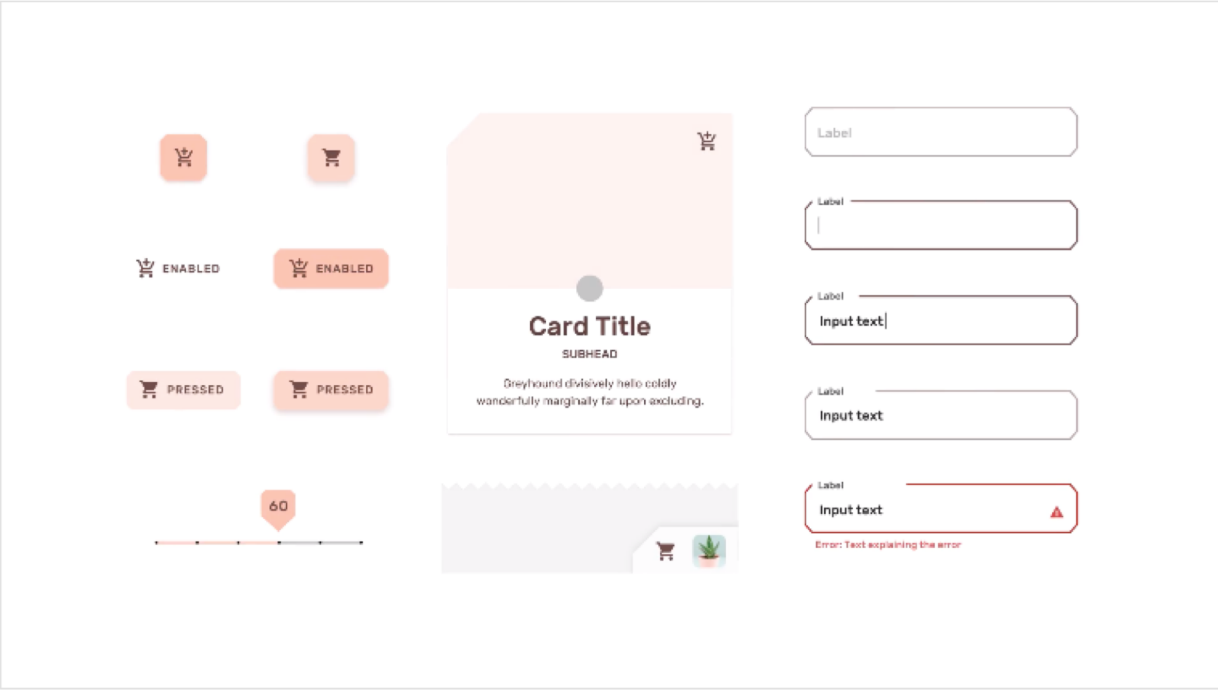


FLUTTER

- UI kit lanciato da Google nel 2018
- Permette di creare app native per Android e iOS
- Supporta I principi del «Material Design» (https://material.io/design/introduction)
- Basato su uno specifico linguaggio: Dart
- Il codice della app è scritto una volta sola per tutte le tipologie doi dispositivi

MATERIAL DESIGN

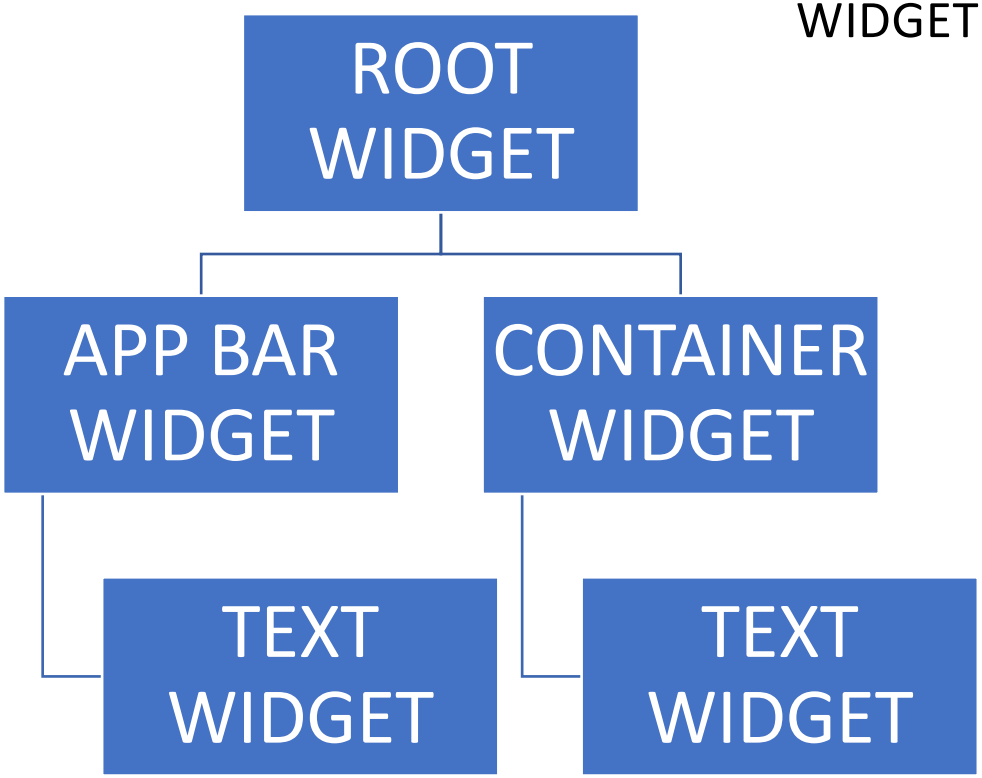
Material is a design system created by Google to help teams build high-quality digital experiences for Android, iOS, Flutter, and the web.



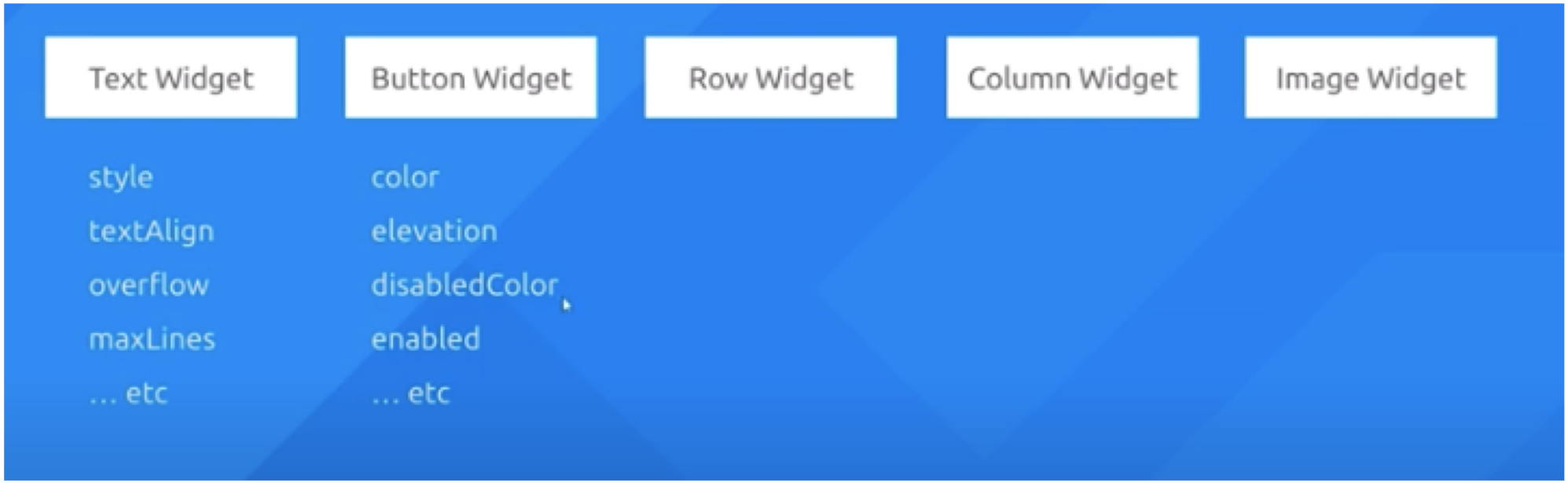
https://material.io/design/introduction#principles

FLUTTER WIDGETS

WIDGET TREE



FLUTTER WIDGETS



Ogni widget è una CLASSE
Il comportamento è definite mediante il linguaggio DART

STATELESS vs STATEFUL WIDGETS

- Stateless widgets:
  - Non modificano il loro stato
  - La visualizzazione non può essere modificata da azioni dell'utente o dal codice

- Stateful widgets:
  - Possono modificare il loro stato
  - Implementano un metodo che verifica il cambio di stato
  - Quando avviene un cambio di stato, viene invocato il build del widget e viene aggiornata la visualizzazione

STATEFUL WIDGETS

```
class page1 extends StatefulWidget {
  @override
  _page1State createState() => _page1State();
}
```

CLASSE page1→ si effettua un override del metodo che gestisce il cambio di stato creando un oggetto della classe _page1State

```
class _page1State extends State<page1> {
MaterialColor col = Colors.deepPurple;
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: col,
      appBar: AppBar(
        title: Text('Page 1'),
        centerTitle: true,
      ),
      body: Center(
child: FlatButton.icon (
…)),

    );
  }
}
```

_page1State: CLASSE PRIVATA CHE GESTISCE LO STATO →

- lo stato attuale della classe page1 è definito dalla classe page1State
- Il builder di questa classe ritorna un widget (Scaffold) che rappresenta lo stato corrente della pagina
- Quando le variabili/oggetti si modificano e il widget si modifica → viene rilanciato il builder di page1

STATEFUL WIDGETS

```
body: Center(
        child: FlatButton.icon(
          onPressed: () {
            setState(() {
              col = Colors.lightBlue;
            });
          },
          icon: Icon(Icons.color_lens),
          label: Text('Change background color')
        )),
```

La funzione **setState** permette di controllare le modifiche della UI

LINGUAGGIO DART: VARIABILI

- Dart è uno “Statically typed variable language ” → il tipo di dato va sempre dichiarato

```
int a = 5;
String b = “Sara”;
b = 7          //NON FUNZIONA
```

- Esistono anche variabili “dinamiche” che possono cambiare tipo in base al dato

```
Dynamic b = ”Sara”
b = 7          //FUNZIONA
```

LISTS

- Esiste un tipo lista che è intrinsecamente dinamico ma che può essere reso statico
- Il tipo lista contiene variabili necessariamente dello stesso tipo

```
List a = [2,4,5]
List names = ['Sara', 'Alessandro', 'Elisa']
List <String> names = ['Sara', 'Alessandro', 'Elisa']          //vincolato ad essere string
List a = [2,7,'Elisa']                                          //NON FUNZIONA
```

- Il tipo list è modificato mediante metodi:

```
names.add("Jim");
names.remove ("Sara");
```

- L'accesso alla lista avviene per indicizzazione

```
names[0]
```

MAPS

- “Map” è un tipo di dato simile a un dizionario Python
- È formato da coppie “chiave” : “valore”
- È dichiarato tra {…}

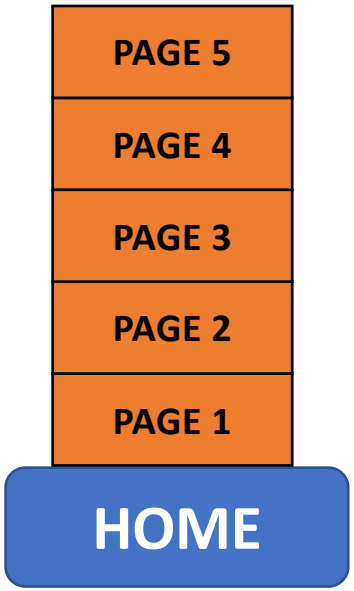
  Maps m = {“Key1”: 1, “Key 2”: 2, …, “Key N”: “Pippo”}
- Si accede ai valori delle single chiavi mediante [“NomeChiave”]

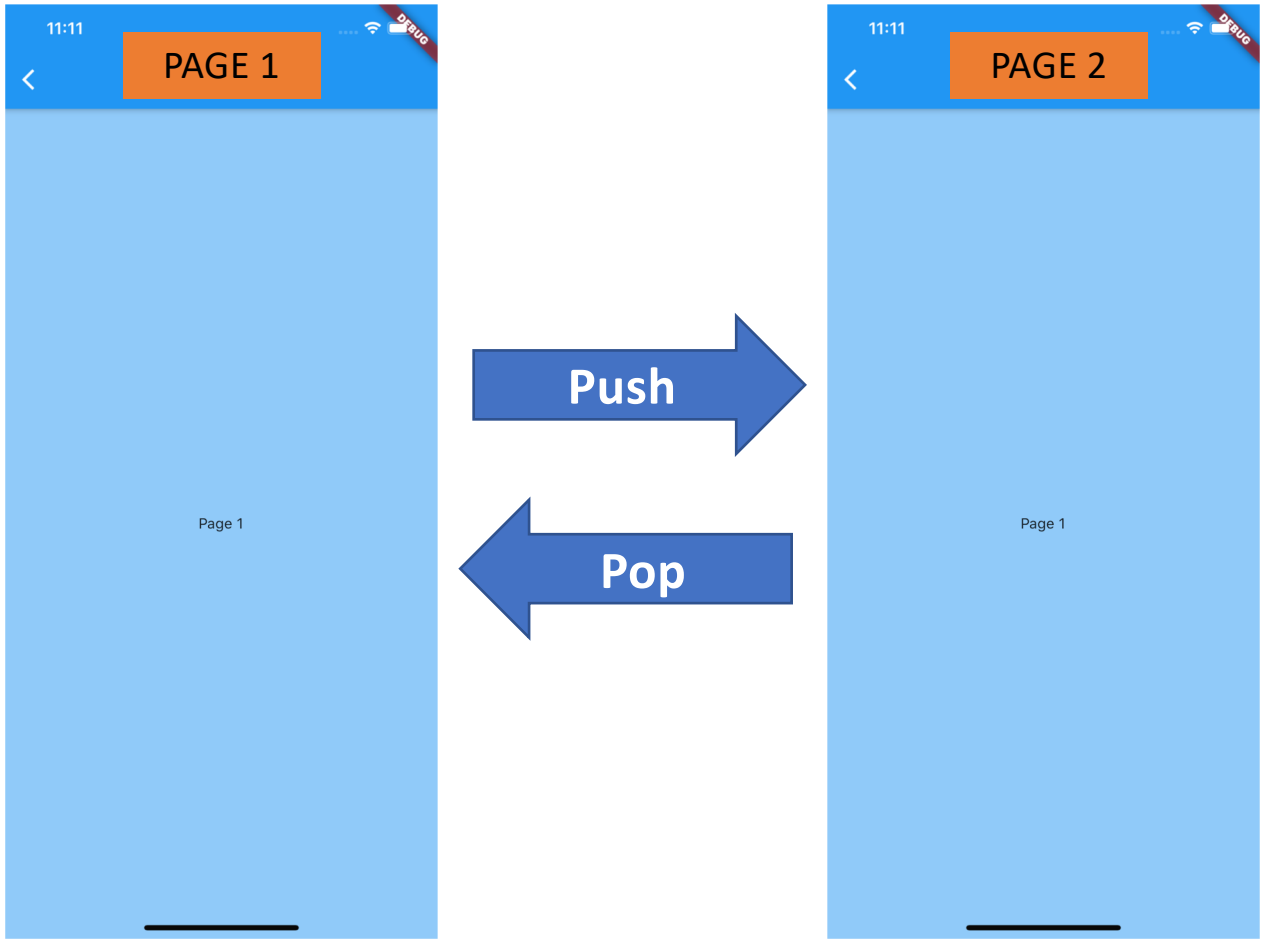
  print(m[”Key1”])

  >> 1

NAVIGAZIONE TRA PAGINE DELLA APP

APP page stack





ROUTING

- Le route di indirizzamento possono essere inserite in una variabile di **tipo map**

```
void main() {
  runApp(MaterialApp(
    initialRoute: '/', //redundant because it is default
    routes: {
      '/': (context) => Home(),
      '/page1': (context) => page1()
     '/page2': (context) => page2()
    },
  )
  );
}
```

Variabile che rappresenta il contesto in cui l'applicazione si trova (quale view)

Builder del widget che rappresenta la view a cui andare

ROUTING

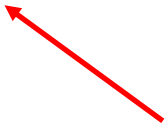
- I metodi di routing sono implementati nella classe Navigator

Navigator.*pushNamed*(context, <String> routeName)

Navigator.*push*(context, <route> route)

Navigator.*pop*(context)

Normalmente non è necessario perché si usa il back button della appBar



Si usa la classe MaterialPageRoute e si passa il builder del widget che rappresenta la view a cui andare

Dalla variabile map definita

ASYNCHRONOUS PROGRAMMING

- Flutter gestisce un unico thread di operazioni e non supporta multithreading → in caso di operazioni lente (es. Download di file, interrogazione di DB) l'esecuzione si bloccherebbe fino al termine dell'operazione
- Il blocco dell'esecuzione non è accettabile per la user experience
- Si implementano dei metodi che consentono la "programmazione asincrona"

  Await, Async (funzioni messe a disposizione da Dart)

  Future (API che mette a disposizione I metodi di programmazione asincrona)

  Then (funzione che fa parte della API Future che opera come async e await)

EXAMPLE

```
import 'dart:async';
void main() {
  print('Execution starts');
  printFileContent();
  print ('Execution ends');
}

printFileContent() {
  Future<String> fileContent = downloadFile();
  print ('This is the result: $fileContent');
  }

Future<String> downloadFile() {
  Future<String> result = Future.delayed(Duration(seconds: 7), () {
    return '123456789';
  });
  return result;
}
```

Funzione che riceve una “Future” string → promessa di ritorno di una variabile stringa

Funzione che ritorna il risultato dopo 7 sec

```
Console

  Execution starts
  This is the result: Instance of '_Future<String>'
  Execution ends
```

Nell’esecuzione non si attendono I 7 sec e quindi viene stampata la “promessa” di stringa

EXAMPLE

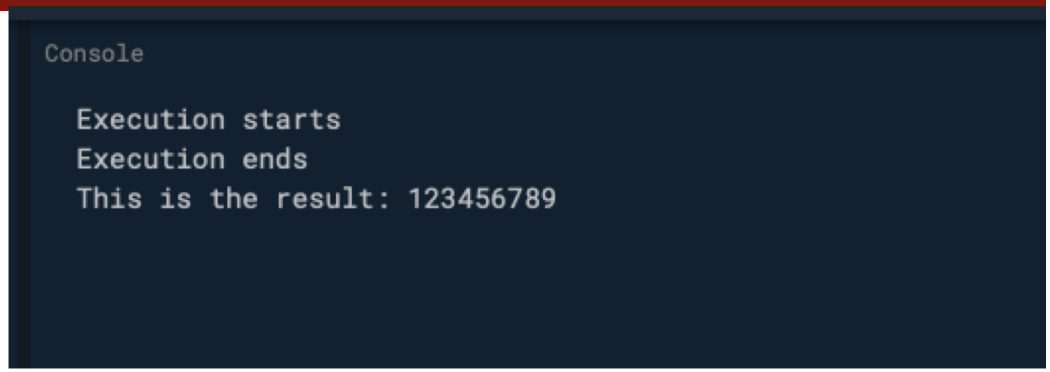
```
import 'dart:async';
void main() {
  print('Execution starts');
  printFileContent();
  print ('Execution ends');
}

printFileContent() async {
  String fileContent = await downloadFile();
  print ('This is the result: $fileContent');
  }

Future<String> downloadFile() {
  Future<String> result = Future.delayed(Duration(seconds: 7), () {
    return '123456789';
  });
  return result;
}
```

La funzione viene resa asincrona e aspetta il risultato di downloadFile()

Funzione che ritorna il risultato dopo 7 sec



Nell'esecuzione non si attendono I 7 sec e si passa comunque all'istruzione successive però il programma non termina finchè anche la funzione più lenta non è terminata

SQLITE IN FLUTTER: SQFLITE plugin

- SQFLITE è il plugin messo a disposizione in flutter per gestire la persistenza dei dati mediante un database SQL
- SQFLITE salva i dati su database SQLite in forma di oggetti MAP
- SQFLITE restituisce i dati in forma di oggetti MAP
- Servono quindi:
  - Una classe che rappresenti il MODELLO dei dati scambiati con il database e per gestire la conversione in map
  - Una classe che permetta la gestione delle operazioni di CREATE, QUERY, UPDATE, DELETE (CRUD)

ADD DEPENDENCIES

- Nel file pubspec.yaml devono essere aggiunte le dipendenze necessarie:
  - Sqflite: any //database plugin
  - Path_provider: any //package per la gestione dei path nel file system
  - Intl:^0.16.1 //per il formatting di numeri e date – controllare la versione attuale del package

CREAZIONE DEL MODELLO

- Il modello può essere incluso in un package (per rendere più leggibile il codice)
- Si crea la(/le) classe(/i) che rappresenta I dati provenienti dal database (ciascuna classe può rappresentare gli oggetti contenuti in una tabella)
- È conveniente che gli attributi che rappresentano le colonne della tabella siano privati → vanno costruiti i setter e i getter
- Si creano le funzioni per creare gli oggetti map e per creare un oggetto a partire da un dato di tipo map

ESEMPIO

```
class Patient {
  int _id;
  String _pat_ID;
  int _age;
  int _gender;

  Patient();  //to make one optional, put []
  Patient.declared (this._id,this._pat_ID,this._age,[this._gender]);  //named constructor

// getter
  int get id => _id;
  String get pat_ID => _pat_ID;
  int get age => _age;
  int get gender => _gender;

// setter
  set pat_ID (String newPat_id) {
    this._pat_ID = newPat_id; }

  set age (int newAge) {
    this._age = newAge;}

  set gender (int newGender) {
     if ((newGender == 1) ||
(newGender ==2)) {
     this._gender = newGender;  }
   }
```

ESEMPIO

```
//map conversion

Map<String, dynamic> toMap() {

    var newMap = Map<String, dynamic>();
    newMap['ID'] = _id;
    newMap['Pat_ID'] = _pat_ID;
    newMap['Age'] = _age;
    newMap['Gender'] = _gender;

  return newMap;

}
```

```
//named constructor

Patient.fromMap(Map <String,dynamic> map) {
    this._id = map['ID'];
    this._pat_ID = map['Pat_ID'];
    this._age = map['Age'];
    this._gender = map['Gender'];

  }
```

CREAZIONE DEL DB MANAGER

- Classe definita per:
  - Gestire l'apertura del database
  - Definire le operazioni necessarie sul DB (create, update, query, delete)

- Condizioni importanti:
  - L'istanza della classe deve essere un singleton
  - L'istanza del database deve essere un singleton
- Le query vengono gestite da metodi della classe Database
  - Rawquery → metodo che permette di passare la stringa SQL
  - Query → metodo che riceve i parametri della query come argomenti
  - Tutti i metodi ritornano Liste di Map

ESEMPIO

```
class DBmanager {

static DBmanager _DBmanager;
static Database _database;

DBmanager._createInstance();

factory DBmanager () {
  if (_DBmanager == null) {
    _DBmanager = DBmanager._createInstance();  }
  return _DBmanager;
}

Future <Database> get database async {
  if (_database == null) {
    _database = await initializeDatabase();   }
  return _database;
}
```

Singleton: _DBmanager viene istanziato solo se non esiste

Singleton: _database viene istanziato solo se non esiste

ESEMPIO

```
Future <List< Map <String, dynamic>>> getAllPatients() async {

    Database db = await this.database;

//using raw sql statement
var result = await db.rawQuery('SELECT * FROM $tableName');
//OPPURE using methods in sqflite
    // var result = await db.query(tableName);

return result;
}
```

Metodo asincrono

è una lista di maps fatte di key = stringa, value = dynamic