Flutter Fundamentals Short recap day #1

Peter Kassenaar –
info@kassenaar.com

Agenda - details

- Introduction – overview of the Flutter landscape
- Flutter tooling – installation
- Hello World –the structure and architecture of Flutter apps.
- Zooming in on Flutter:
  - Components – Stateless vs. Stateful

Agenda – cont'd

- Widgets - introduction
- The Flutter layout system – `Scaffold()` and more
- Using Images and assets
- More layout widgets
    - `Button()`, `Icon()`, `Container()`, `Padding()`
    - `Row()`, `Column()`, properties, children and more
- Optional: working with data
    - `ListView()`, `Card()`, Designing layouts

“Flutter is Google’s UI toolkit for building beautiful, natively compiled applications for mobile, web, and desktop from a single codebase.”

Installation – recommended order

1. Install Visual Studio
2. Install IntelliJ
3. Install Flutter plug-in for IntelliJ
4. Install Flutter SDK
5. Update Windows PATH variable
6. Run `flutter doctor`, fix any possible problems

Default code – recognize the structure

```
import 'package:flutter/material.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
        visualDensity: VisualDensity.adaptivePlatformDensity,
      ),
      home: MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}
…
```

(Yours might be slightly different, due to updates)

Study the default code. It has useful comments.

Flutter == Dart in action

- Important widgets
  - Scaffold()
  - AppBar()
  - Themes, fonts & colors
  - Image()
  - Icon()
  - Row(), Column()
  - ListView()
  - Container()
  - Expanded()

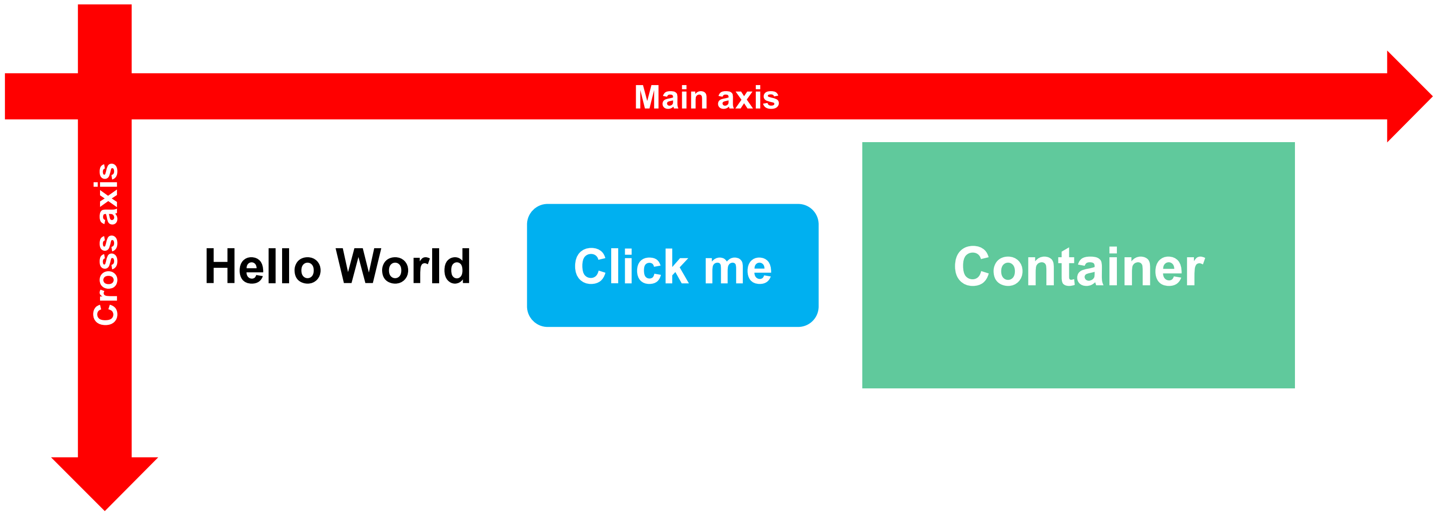
“Every Flutter App

is composed as a

tree of widgets”

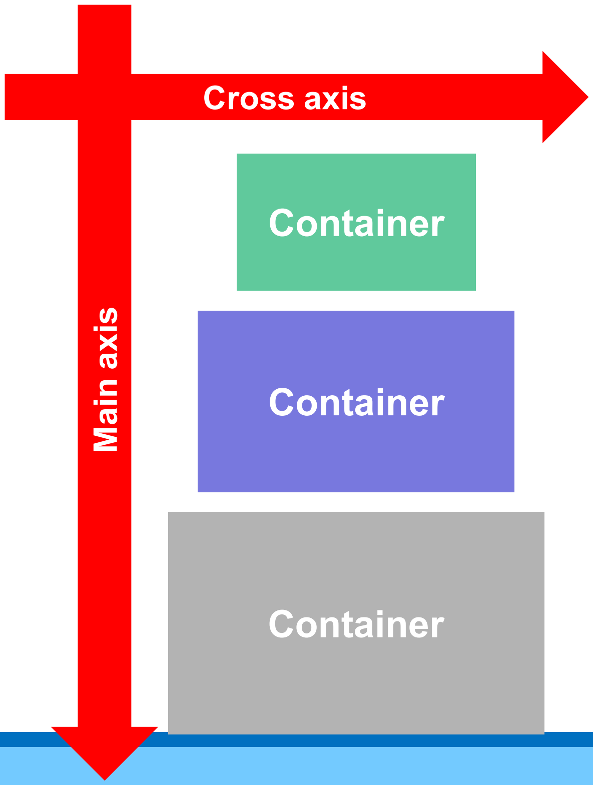
Alignment in Rows/Columns

- In Rows:
  - Use `MainAxisAlignment` for horizontal layout
  - Use `CrossAxisAlignment` for vertical layout

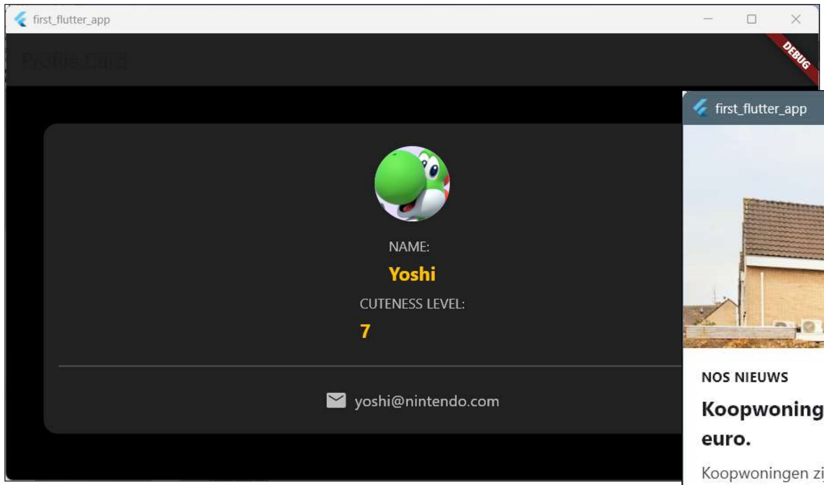


Alignment

- In Columns – opposite to Rows:
  - Use `MainAxisAlignment` for vertical layout
  - Use `CrossAxisAlignment` for horizontal layout

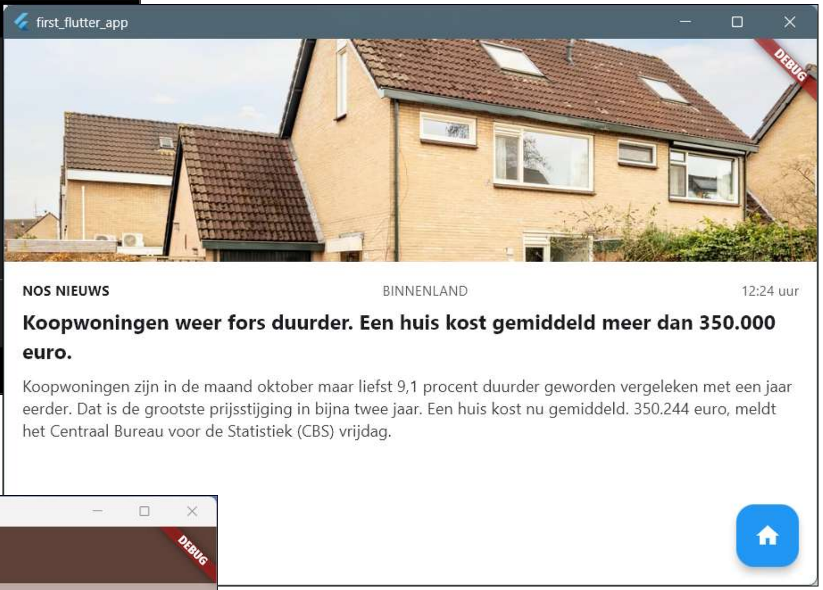


Creating various layouts by combining Widgets



NAME:
Yoshi
CUTENESS LEVEL:
7

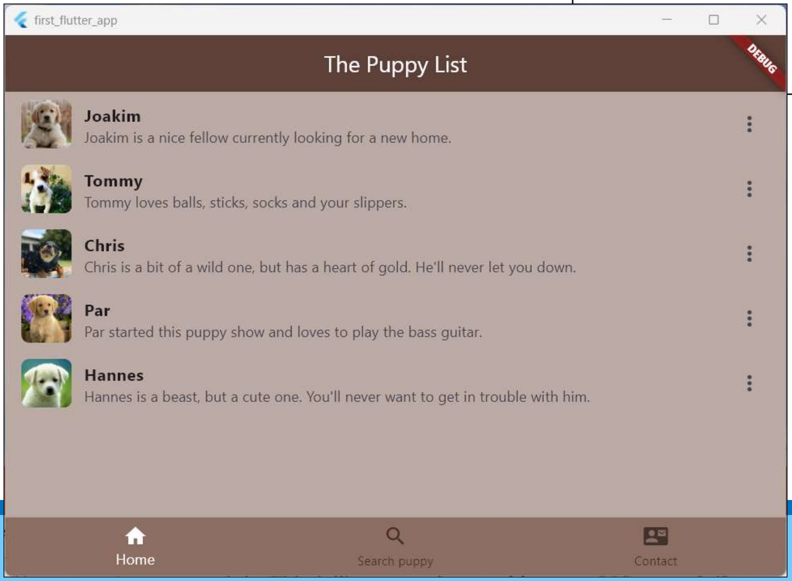
yoshi@nintendo.com



NOS NIEUWS BINNENLAND 12:24 uur

Koopwoningen weer fors duurder. Een huis kost gemiddeld meer dan 350.000 euro.

Koopwoningen zijn in de maand oktober maar liefst 9,1 procent duurder geworden vergeleken met een jaar eerder. Dat is de grootste prijsstijging in bijna twee jaar. Een huis kost nu gemiddeld. 350.244 euro, meldt het Centraal Bureau voor de Statistiek (CBS) vrijdag.



The Puppy List

Joakim
Joakim is a nice fellow currently looking for a new home.

Tommy
Tommy loves balls, sticks, socks and your slippers.

Chris
Chris is a bit of a wild one, but has a heart of gold. He'll never let you down.

Par
Par started this puppy show and loves to play the bass guitar.

Hannes
Hannes is a beast, but a cute one. You'll never want to get in trouble with him.

Questions:

- Using `const` or not in code?
- Use `const` in situations where the following is true:
  - The object is entirely immutable.
  - Its parameters (if any) are *also compile-time constants*.
  - You want to *avoid recreating the object unnecessarily* every time a widget rebuilds.
- E.g: consider this snippet:

```
@override
Widget build(BuildContext context) {
  return const Placeholder();
}
```

- Removing the keyword `const` is less optimal!
- It still works because `Placeholder()` is immutable by design.
- However, removing const makes a difference in performance!
  - Without const, a *new instance* of `Placeholder()` will be created every time this widget rebuilds.
  - With `const`, the *same compile-time constant instance is reused* across rebuilds, improving performance by avoiding unnecessary object creation!

```
Widget build(BuildContext context) {
  return Placeholder(); // also works!
}
```

Private, public, protected?

- Dart does NOT have traditional access modifier keywords like `private`, `public` and `protected`.
- Instead uses conventions and scoping rules to determine visibility
- Private
  - A member is `private` if its name starts with an underscore (for instance `_puppyList[]`)

Cont’d

- Public
  - Any variable without a leading underscore is public by default
- Protected
  - Dart does not have an explicit `protected` modifier.
  - Instead, you use inheritance and overriding to achieve similar behavior for class members.

Controlled access? Write `class` + getters/setters

- If you need additional control over how variables are accessed or modified, create a `class` and use `get` and `set`.
  - For Example

```
class PuppyListClass {
  late List<String> _puppies;

  List<String> get puppies => _puppies;

  set puppies(List<String> newList) => _puppies = newList;
}
```

Const vs. final

- Both keywords are used for variables that cannot be reassigned
- They differ in how and *when* the value is evaluated and used
- A variable declared as `const` is a compile-time constant
  - its value is determined at compile time and cannot change.
- The value of a `const` variable must be known and fixed at the time of compilation.
  - You can't assign the result of runtime operations to a `const`.

```
const int maxItems = 100;
const double pi = 3.14159;
const String greeting = "Hello, world!";
```

final

- A variable declared as `final` can be assigned only once, but its value is determined at runtime.
    - Once assigned, it cannot be changed, but it doesn't need to be known at compile time.

```
final DateTime now = DateTime.now(); // Value is determined at runtime
```

verdict

- Use `const` when the value is completely fixed and known at compile-time.
  - This could include compile-time constants or immutable values like widget configurations.
- Use `final` when the value cannot change after it is assigned, but its assignment might depend on runtime conditions or calculations.

other

Questions?

Today:

- State management in various ways
  - Stateful widgets – using models/custom classes
  - passing parameters, passing functions
- Communicating with external API’s
  - http / other methods
- More state management
  - bloc pattern, bloc + cubit implementation
  - payload – emit multiple events

Tomorrow

- TextFields
- Routing / Navigation
- Complete applications
- gRPC
- Gestures
- Publication (executables/packages)
- Evals & goodbye
- …