

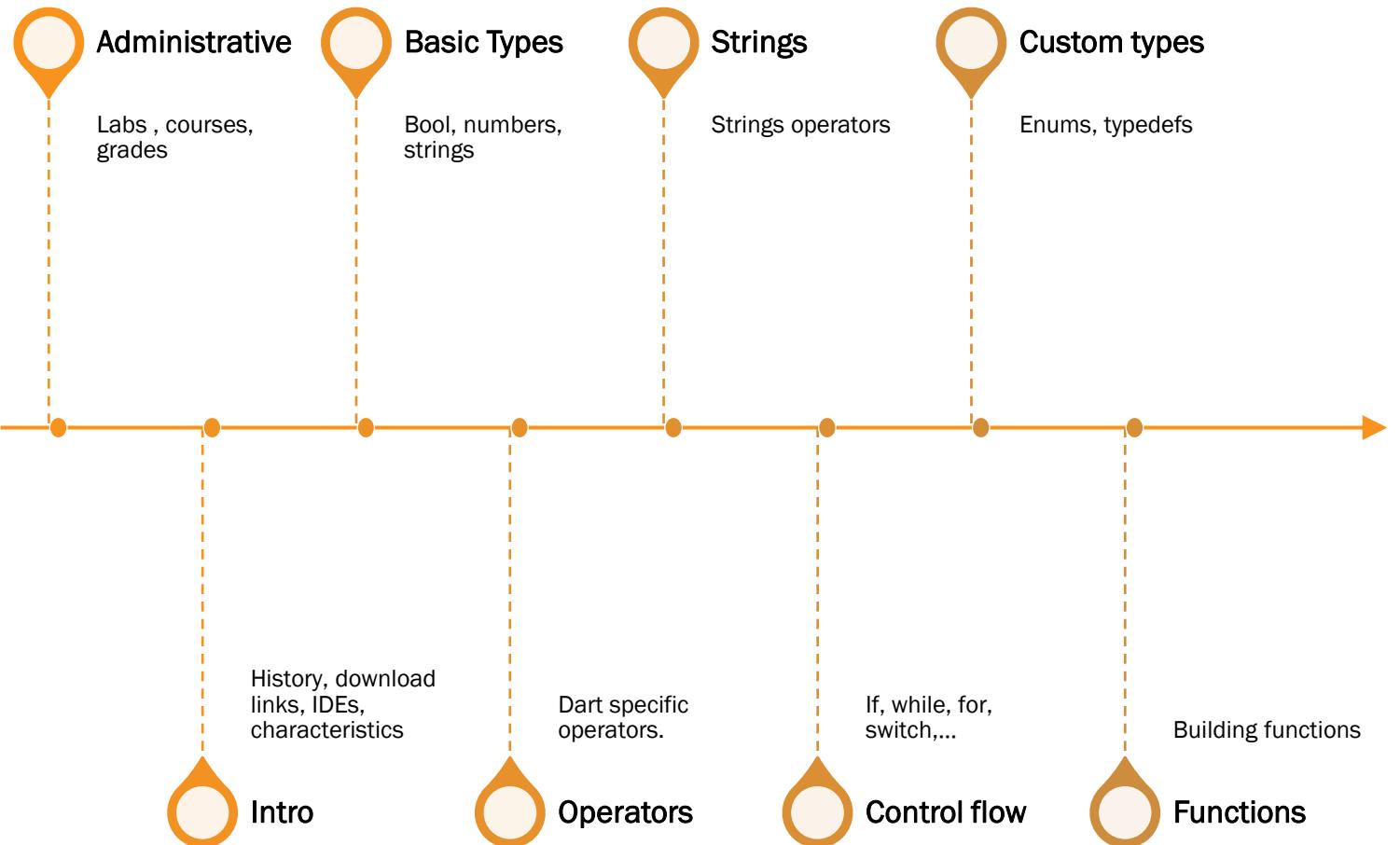


DART Language

COURSE 1 (REV 6)

GAVRILUT DRAGOS

Agenda



Administratives

Administratives

We will study **DART** language and **FLUTTER** framework for mobile development.

Course web page: gdt050579.github.io/dart_course_fii/

Grading: Gauss-like system (check out our Administrative page for more details)

Examination type:

- A lab project → 60 points
- Course examination → 30 points
- Lab activity → 10 points

Minimal requirements:

- Lab project → 20 points
- Course examination → 10 points

Intro

What is DART

DART is a programming language designed by Google that can be used to create both mobile and web applications.

History:

- Announced in 2012, October
- Dart v2.0 launched in August 2018
- Current version: Dart 3.3 (January 15, 2024)

Docs & Install:

- Documentation: <https://dart.dev/guides>
- Install: <https://dart.dev/get-dart>
- Download standalone kit: <https://dart.dev/get-dart/archive>

DART IDEs

The easiest way to quickly test a DART program (online) is via *DartPAD*: <https://dartpad.dev/>?

Other IDEs:

- NotePad++ → <https://notepad-plus-plus.org/downloads/>
- Visual Studio Code → <https://code.visualstudio.com/download>
- IntelliJ / Android Studio → download plugin from: <https://dart.dev/tools/jetbrains-plugin>
- Eclipse → plugin can be found: <https://github.com/eclipse/dartboard>
- VIM → plugin can be found: <https://github.com/dart-lang/dart-vim-plugin>
- Sublime → plugin can be found: <https://github.com/guillermooo/dart-sublime-bundle>

DART Characteristics

1. C-Like language
2. Typing : Strong (inferred) and optional
3. Object oriented
4. Garbage collector
5. Compiles to both native (Ahead of Time) and JavaScript
6. It also contains a stand-alone VM that can be used to run a Dart code
7. Can be used (together with Flutter framework) to create apps for bot Android and iOS

First Dart programme

HelloWorld.dart

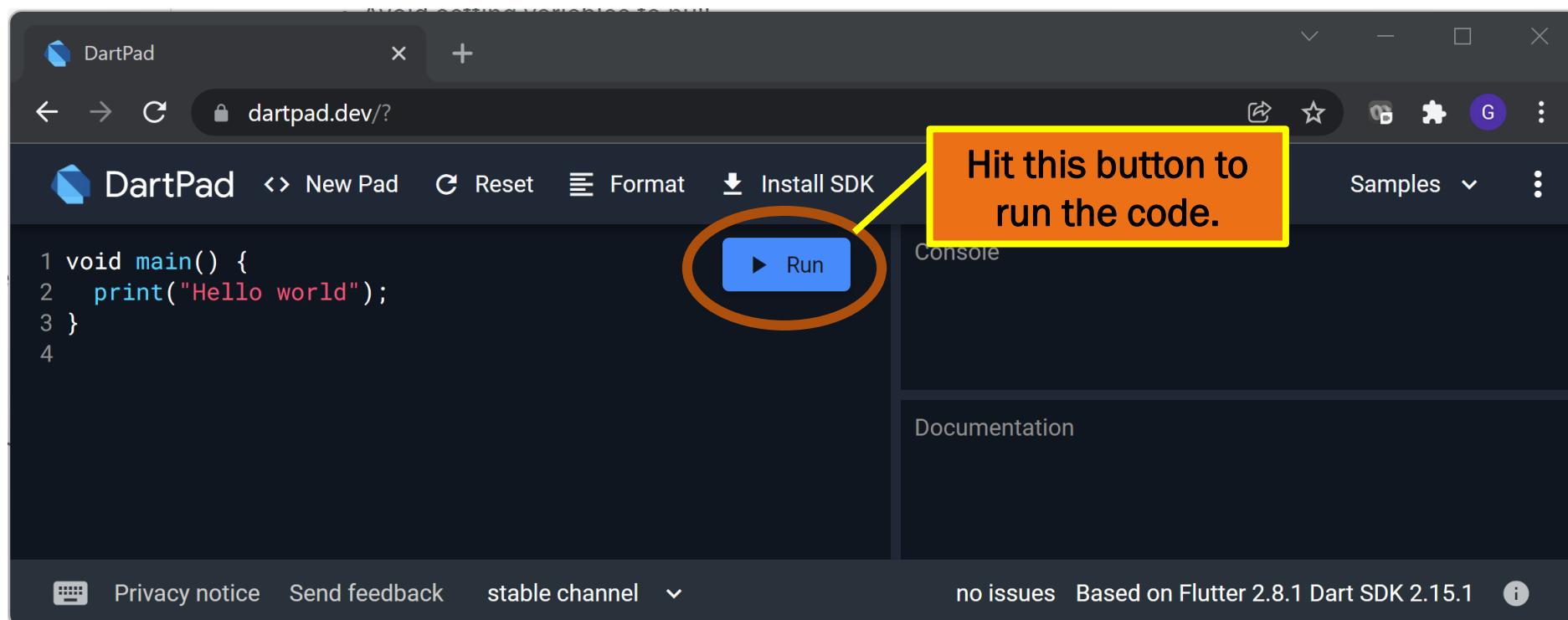
```
void main() {  
    print("Hello, World");  
}
```

To run this code:

1. run “*dart.exe HelloWorld.dart*”
2. run “*dart.exe compile exe HelloWorld.dart*” then execute “*HelloWorld.exe*” that was created after the compiling ends
3. run “*dart2js.bat HelloWorld.dart -o HelloWorld.js*” then load “*HelloWorld.js*” into a browser and execute it. To load in different browser please consult: <https://dart.dev/tools/dart2js>
4. try <https://dartpad.dev/> , then paste the previous code and hit “Run” button

First Dart programme

HelloWorld.dart → dartpad example



Basic Types

Basic Types

1. Boolean
2. Numeric
3. String
4. Dynamic

Basic Types

1. Boolean

2. Numeric

3. String

Strong typed.

```
void main() {  
    bool b;  
    b = true;  
    print(b);  
}
```

Inferred type

```
void main() {  
    var b = false;  
    print(b);  
}
```

Optional

```
void main() {  
    var b;  
    print(b);  
    b = false;  
    print(b);  
}
```

Output:
null
false



Basic Types

1. Boolean
2. Numeric
3. String
4. Dynamic

One type (**num**) with two sub-types:

1. **int** (64 bit signed integer) → equivalent to “long long” type from C/C++. Depending on the platform, an **int** value can be up to 64 bit (but lower on web).

```
void main() {  
    int x = 10; // x = int  
    var y = 123; // y = int  
    num z = 1; // z = int  
    var t = int.parse("5"); // t = int  
    print(x); print(y);  
    print(z); print(t);  
}
```

Basic Types

1. Boolean
2. Numeric
3. String
4. Dynamic

One type (**num**) with two sub-types:

2. **double** (double precision floating point number – IEE 754 standard). It is equivalent to “double” type from C/C++. Uses 64 bit for storage.

```
void main() {  
    double x = 10; // x = double  
    var y = 123.2; // y = double  
    num z = 1.5; // z = double  
    var t = double.parse("1.5"); // t = double  
    print(x); print(y);  
    print(z); print(t);  
}
```

Basic Types

1. Boolean
2. Numeric
3. String
4. Dynamic

A string is immutable (UTF-16 format).

```
void main() {  
    String s1 = "test";    // s1 = string  
    var s2 = "test";      // s2 = string  
    print(s1); print(s2);  
}
```

A string can be formatted in multiple ways

```
var s = 'test';           // single quotes  
var s = "test";          // double quotes  
var s = "te\nst";        // double quotes with escaped chars  
var s = r"te\nst";       // raw string  
var s = """Multi-line  
string""";               // multi-line string
```

Basic Types

1. Boolean
2. Numeric
3. String
4. Dynamic

A string can format a variable or expression if it is inserted into string using \$ character in the following format:

- \$var → for a specific variable
- \${expression} → for both expression and variables

```
var x = 10;  
var y = 20;  
var s1 = "x=$x"; // x=10  
var s2 = "y=${y}"; // y=10  
var s3 = "sum=${y+x}"; // sum=30
```

Use r character to force a string to be a raw string if you want to ignore the \$ character usage.

```
var x = 10;  
var s1 = r"x=$x"; // x=$x
```

Basic Types

1. Boolean
2. Numeric
3. String
4. Dynamic

If what's after the `$` character is an invalid expression (e.g. a variable that does not exist) the code will **NOT** compile

```
var x = 10;  
var y = 20;  
var s1 = "x=$z";      // Error
```



```
Error: Undefined name 'z'.  
var s1 = "x=$z";  
^  
Error: Compilation failed.
```

Basic Types

1. Boolean
2. Numeric
3. String
4. Dynamic

All types from Dart are derived from one type “**dynamic**”. Because of this, if a variable is declared **dynamic**, it can change its type during runtime (similar to how Python typeless variables work).

```
void main() {  
    dynamic v = 10;  
    print(v); // 10  
    v = "Test";  
    print(v); // Test  
    v = true;  
    print(v); // true  
}
```

Basic Types

1. Boolean
2. Numeric
3. String
4. Dynamic

“dynamic” and “var” are two different things.

- dynamic → this is a type that implies that every variable / object can be assigned to an object
- var → this is a keyword that states that the type of a variable must be inferred from its value. If the inference fails, the type will be considered dynamic.

```
void main() {  
    var v = 10;  
    print(v);      // 10  
    v = "Test";   // compiler error as v is int and a  
                  // string ("Test") can not be  
                  // assigned to it.  
}
```

Basic Types

1. Boolean
2. Numeric
3. String
4. Dynamic

The following code will work. Variable “v” is defined using “var” keyword. However, as no value is assigned to it (as part of its initialization), the inference process will not be able to find a suitable type and will use **dynamic** instead.

```
void main() {  
    var v;  
    v = 10;  
    print(v); // 10  
    v = "Test";  
    print(v); // Test  
    v = true;  
    print(v); // true  
}
```

Basic Types

Operator **is** can be used to check if a variable is of a specific type.

```
void main() {  
    int x = 10;  
    print(x is double);  
    print(x is int);  
    print(x is String);  
}
```

However, keep in mind that the results vary depending on the platform:

- On web (JavaScript) the result will be “true, true, false”
- For native builds, the result will be “false, true, false”

Operators

Operators

1. Arithmetic
2. Bitwise
3. Assignment
4. Relational
5. Logical
6. Others

Operators

1. Arithmetic

2. Bitwise

3. Assignment

4. Relational

5. Logical

6. Others

Similar to the ones from C/C++:

- Binary: + | * | / | % | ~/
- Unary: ++ | -

```
void main() {  
    var x = 10;      // int  
    var y = 3;       // int  
    print(x+y);    // 13  
    print(x*y);    // 30  
    print(x/y);    // 3.333333  
    print(x~/y);   // 3  
}
```

- Operator ~/ is used to return the integer result of a division (as a general notion, / operator returns the mathematical result of a division (meaning that even if we divide two integers the result might be a double)).

Operators

1. Arithmetic

2. Bitwise

3. Assignment

4. Relational

5. Logical

6. Others

Similar to the ones from C/C++:

- Binary: & | ^ >> << >>>
- Unary: ~

```
void main() {  
    var x = 10;      // int  
    var y = 3;       // int  
    print(x&y);    // 2  
    print(x|y);    // 11  
    print(x<<y);  // 80  
    print(x>>>y); // 1  
}
```

- Operator **>>>** performs a right shift but for the unsigned value (**>>** and **<<** perform the right and left shift for the signed value).

Operators

1. Arithmetic
2. Bitwise
3. Assignment
4. Relational
5. Logical
6. Others

Similar to the ones from C/C++:

- Binary: `= += -= *= /= %= >>= <<= >>>= ~/= &= |= ^= ??=`

```
void main() {  
    var x = 10;      // int  
    var y = 3;       // int  
    x += y;         // 13  
    x *= y;         // 39  
    x &= y;         // 3  
}
```

- Operator `??=` assigns the value of the right expression only if the left expression is null

Operators

1. Arithmetic
2. Bitwise
3. Assignment
4. Relational
5. Logical
6. Others

Similar to the ones from C/C++:

- Binary: > >= == != < <= is is!

```
void main() {  
    var x = 10;           // int  
    var y = 3;            // int  
    var z = x>y;         // true  
    var t = x is int;    // true  
    var u = x is String; // false  
}
```

- Operator is and is! (pronounced is not) checks if an object is of a certain type

Operators

1. Arithmetic
2. Bitwise
3. Assignment
4. Relational
5. Logical
6. Others

Similar to the ones from C/C++:

- Binary: `&&` `||` `!`
- The only difference between Dart and C++ in this case is that logical operators can not be used with numbers → they have to be used with bool values

```
void main() {  
    var x = 10;                      // int  
    var y = 3;                        // int  
    var z = x && y;                  // error  
}
```

Error: A value of type 'int' can't be assigned to a variable of type 'bool'.
`var z = x && y;`

^

Operators

1. Arithmetic
2. Bitwise
3. Assignment
4. Relational
5. Logical
6. Others

Conditional:

- Form 1: condition ? value if true : value if false

```
void main() {  
    var x = 10;           // int  
    var y = 3;            // int  
    var z = x > y ? 2 : 3; // 2  
}
```

- Form 2: expression₁ ?? expression₂

The logic for this operator is as follows. If expression₁ is not null, the result will be expression₁, otherwise it will be expression₂

```
var x;           // null  
var y = x ?? 10; // y = 10
```

Operators

1. Arithmetic
2. Bitwise
3. Assignment
4. Relational
5. Logical
6. Others

Index / subscript access:

- Form 1: object/array [index]

```
void main() {  
    var x = [1,2,3];  
    var y = x[1];  
    print(y);  
}
```

- Form 2: object/array ?[index]

This will only compute the element from the subscript if the object/array is not null. If this is the case, the value returned is null. If ?[...] is used as the left part of an expression, the assignment will be ignored if it translates to null.

```
var x;           // null  
var y = x?[1]; // y = null
```

```
var x;           // null  
x?[1] = 10; // OK (but x will  
             not be changed)
```

Operators

1. Arithmetic
2. Bitwise
3. Assignment
4. Relational
5. Logical
6. Others

Member access:

- Form 1: object.member

```
void main() {  
    print("test".contains("te"));  
}
```

- Form 2: object?.member

This form will allow access to a member only if the object is not null.

```
void main() {  
    var s = "test";  
    print(s.contains("te"));      // true  
    var t;  
    var x = t?.contains("te");  
    print(x);                  // null  
}
```

Operators

1. Arithmetic
2. Bitwise
3. Assignment
4. Relational
5. Logical
6. Others

Member access:

- Form 2: object?.member

This form will allow access to a member only if the object is not null.

Keep in mind that member evaluation is done dynamically (depending on the type of the object) in this case.

```
void main() {  
    var o; // my object  
    o?.membr1 = 10;  
    o?.member2 = 20;  
    o?.run(10,20);  
}
```

Operators

1. Arithmetic
2. Bitwise
3. Assignment
4. Relational
5. Logical
6. Others

Cascade operator:

- Form 1: object.member..member..member ..member

This form allows access to multiple methods and data members of the same object. More on this topic on-classes.

```
class Test {  
    int x = 10;  
    int y = 20;  
}  
void main() {  
    var t = new Test();  
    t  
        ..x = 200  
        ..y = 100;  
    print(t.x); print(t.y);  
}
```

Operators

1. Arithmetic
2. Bitwise
3. Assignment
4. Relational
5. Logical
6. Others

Cascade operator:

- Form 2: object?.member ..member ..member ..member

This form allows access to multiple methods and data members of the same object if the object is non-null. Can be used with `. Operator`

```
void main() {  
    var t; // an object  
    t  
        ?.x = 200  
        ..y = 100;  
}
```

In this case, access to “x” data member is only granted if “t” is not null. This means that code will stop from validating “y” as it will stop on `? .x`

Operators

1. Arithmetic
2. Bitwise
3. Assignment
4. Relational
5. Logical
6. Others

Cast operator:

- Form: object **as** type

```
void main() {  
    var t = 1;  
    print(t as double);  
}
```

in this case it is possible to convert an int to a double. However, if the casting was not possible, an exception will be thrown. The previous code will run correctly in DartPad (javascript version as int and double can be converted one to another in this case). However, if build as a native code, it will not compile and show the following error: **type 'int' is not a subtype of type 'double' in type cast**

Strings

Strings

Strings in Dart have the following properties:

Concatenation

```
void main() {  
    print("Da"+"rt"); // Dart  
}
```

toString() → every object has a *toString* method that can be used to transform it to a string

```
void main() {  
    var x = 10;  
    print("X = "+x.toString());  
}
```

or

```
void main() {  
    var x = 10;  
    print("X = ${x}");  
}
```

Strings

Multiplication

```
void main() {  
    print("Da"+"r" * 5 + "t"); // Darrrrrt  
}
```

Operator [] can be used to access a character from a specific index

```
void main() {  
    print("Dart"[2]); // r ➔ the 3rd character from string Dart  
    print("Dart"[20]); // exception  
}
```

To find out the length of a string, use .length method

```
print("Dart".length); // will print 4
```

Strings

Casing (two methods: *toUpperCase* and *toLowerCase*)

```
void main() {  
    var s = "Dart";  
    print(s.toLowerCase()); // dart  
    print(s.toUpperCase()); // DART  
}
```

Trimming (three methods: *trim* , *trimLeft* and *trimRight*)

```
void main() {  
    var s = " Dart ";  
    print("[ "+s.trim()+" ]");      // [Dart]  
    print("[ "+s.trimLeft()+" ]"); // [Dart ]  
    print("[ "+s.trimRight()+" ]");// [ Dart]  
}
```

Strings

Substrings (one method: *substring* with two implementations)

- *substring (indexStart);*
- *substring (indexStart , indexEnd);* → *indexEnd* must be bigger or equal to *indexStart*

```
void main() {  
    var s = "Dart language";  
    print(s.substring(5));      // Language  
    print(s.substring(3,10));   // t Langu  
    print(s.substring(3,4));   // t  
    print(s.substring(3,1));   // error (exception)  
}
```

Strings

Testing to see if a string contains a string (several methods: ***startsWith*** , ***contains*** with two forms

- method (stringToSearch);
- method (stringToSearch , index);

and ***endsWith*** with only one form).

```
void main() {  
    var s = "Dart language";  
    print(s.startsWith("Dart")); // true  
    print(s.startsWith("rt",2)); // true  
    print(s.endsWith("ge")); // true  
    print(s.contains("Dart")); // true  
    print(s.contains("Dart",2)); // false  
}
```

Strings

Finding the index of a string in another string (two methods: *indexOf* , *lastIndexOf* with two forms

- method (stringToSearch);
- method (stringToSearch , index);

```
void main() {  
    var s = "Dart language";  
    print(s.indexOf("rt"));           // 2  
    print(s.indexOf("rt",10));        // -1  
    print(s.lastIndexOf("lang"));     // 5  
    print(s.lastIndexOf("lang",2));   // -1  
}
```

If the method is successful, the result is the index of the search string. Otherwise, -1 will be returned.

Strings

Comparing two strings ca be done via

- method compareTo(stringToCompare); → returns 0 if the two strings are equal, 1 if the first string (this) is bigger than the second one (the parameter), and -1 otherwise
- operator ==

```
void main() {  
    print("abc" == "abc"); // true  
    print("abc" == "ABC"); // false  
    print("zzz" > "aaa"); // compile error  
    print("zzz".compareTo("aaa"))); // 1  
    print("zzz".compareTo("zzz"))); // 0  
    print("aaa".compareTo("zzz"))); // -1  
}
```

Strings

Splitting a string can be done via *split* method

- method `split(stringToSplit);`

```
void main() {  
    var s = "Red,Green,Blue";  
    var l = s.split(",");  
    print(l[0]); // Red  
    print(l[1]); // Green  
    print(l[2]); // Blue  
    print(l.length);  
}
```

Strings

To replace a string with another one, use one of the following methods:

- replaceAll (stringToSearch , stringToReplaceWith);
- replaceFirst (stringToSearch , stringToReplaceWith);
- replaceFirst (stringToSearch , stringToReplaceWith , startIndex);
- replaceRange (startIndex, endIndex, stringToReplaceWith);

```
void main() {  
    var s = "D_rt progr_mming";  
    print(s.replaceAll("_","a"));          // Dart programming  
    print(s.replaceFirst("_","a"));         // Dart progr_mming  
    print(s.replaceFirst("_","a",4));        // D_rt programming  
    print(s.replaceRange(0,2,"DAA"));       // DAArt programming  
}
```

Besides these methods, there are another two (*replaceAllMapped* and *replaceFirstMapped*) that are going to be further discussed when talking about regular expressions

Strings

A string can also be created using a **StringBuffer** object (similar to the one that Java has). The most important methods are:

- write
- writeAll
- clear

```
void main() {  
    var sb = new StringBuffer();  
    sb.write("I");  
    sb.write(" like");  
    sb.write(" Dart");  
    var s = sb.toString();  
    print(s); // I like Dart  
}
```

Control Flow Instructions

Control Flow Instructions

1. if
2. while
3. do..while
4. for
5. switch

Control Flow Instructions

1. if
2. while
3. do..while
4. for
5. switch

Format:

- if (condition) <then_part>
- if (condition) <then_part> else <else_part>

```
void main() {  
    var a = 10;  
    if (a>10) a = a + 1;  
    if (a<10) {  
        a = a + 2;  
        a = a - 3;  
    } else {  
        a = a * 5;  
    }  
    print(a); // 50  
}
```

OBS: condition **MUST** use boolean values !

Control Flow Instructions

1. if
2. while
3. do..while
4. for
5. switch

Format:

- **if** (condition) <then_part>
- **if** (condition) <then_part> **else** <else_part>

OBS: condition **MUST** use boolean values !

```
void main() {  
    var a = 10;  
    if (a)  
        a = a + 1;  
}
```

Error: A value of type 'int' can't be assigned to a variable of type 'bool'.
if (a)

^

Error: Compilation failed.

This type of condition would have been evaluated to true and would have compiled if a C/C++ compiler would have been used.

Control Flow Instructions

1. if
2. while
3. do..while
4. for
5. switch

Format:

- **while** (condition) <do_part>
- **break** and **continue** can be used while in the **while-loop**

OBS: condition **MUST** use boolean values !

```
void main() {  
    var c = 0, s = 0;  
    while (c < 100) {  
        c++;  
        if (c%2==0) continue;  
        s+=c;  
    }  
    print(s); // 2500  
}
```

Control Flow Instructions

1. if
2. while
3. do..while
4. for
5. switch

Format:

- do { ... } while (condition);
- break and continue can be used while in the do..while-loop

OBS: condition **MUST** use boolean values !

```
void main() {
    var c = 0, s = 0;
    do {
        c++;
        if (c%2==0) continue;
        s+=c;
    } while (c < 100);
    print(s); // 2500
}
```

Control Flow Instructions

1. if
2. while
3. do..while
4. for
5. switch

Format:

- **for** (initialization ; condition ; increment) { ... };
- **break** and **continue** can be used while in the **for**

```
void main() {
    var s = 0;
    for (var i=0;i<100;i++)
    {
        s+=i;
    }
    print(s); // 4950
}
```

```
void main() {
    var s = 0, i = 0;
    for (;i<100;i++)
    {
        s+=i;
    }
    print(s); // 4950
}
```

- Just like the “for” from C/C++, all three components (initialization, condition and increment are optional).

Control Flow Instructions

1. if
2. while
3. do..while
4. for
5. switch

Format:

- **for** (initialization ; condition ; increment) { ... };
- **break** and **continue** can be used while in the **for**
- Just like the “for” from C/C++, all three components (initialization, condition and increment are optional).

```
void main() {
    var s = 0, i = 0;
    for (;;) i++ {
        if (i>=100)
            break;
        s+=i;
    }
    print(s); // 4950
}
```

```
void main() {
    var s = 0, i = 0;
    for (;;) {
        if (i>=100)
            break;
        s+=i++;
    }
    print(s); // 4950
}
```

Control Flow Instructions

1. if
2. while
3. do..while
4. for
5. switch

Format:

- **for** (initialization ; condition ; increment) { ... };
- **break** and **continue** can be used while in the **for**
- Just like the “for” from C/C++, several initialization and multiple increments can be added into a single **for** definition.

```
void main() {
    for (var i=0,j=5;i*j<30;i++,j++) {
        print("$i,$j");
    }
}
```


Output:
0 , 5
1 , 6
2 , 7
3 , 8

Control Flow Instructions

1. if
2. while
3. do..while
4. for
5. switch

Format:

- `for (var <name> in <iterable object>) { ... };`
- This form is similar to a for-each parser and will further be discussed when talking about iterable objects.

```
void main() {  
    var s = "I like Dart";  
    for (var w in s.split(" "))  
    {  
        print(w);  
    }  
}
```

Output:
I
like
Dart

Control Flow Instructions

1. if
2. while
3. do..while
4. for
5. switch

Format:

- `switch (expression) { case valu1: ... , case value2: ... };`
- Similar to the format from C/C++ but with some differences that underline some limitation in terms of performance !

```
void main() {
    var x = 1;
    switch (x) {
        case 1: print("one"); break;
        case 2: print("two"); break;
        default: print("something else"); break;
    }
}
```

Control Flow Instructions

1. if
2. while
3. do..while
4. for
5. switch

Format:

- switch (expression) { case value1: ... , case value2: ... };
- Some differences from C/C++;

“break” can not be omitted if a fall through is needed.

```
void main() {
    var x = 1;
    switch (x) {
        case 1: print("one");
        case 2: print("two"); break;
        default: print("something else"); break;
    }
}
```

Error: Switch case may fall through to the next case.
case 1: print("one");

Control Flow Instructions

1. if
2. while
3. do..while
4. for
5. switch

Format:

- switch (expression) { case value1: ... , case value2: ... };
- Some differences from C/C++;

The solution is to use “continue” to a label defined before a case value:

```
void main() {
    var x = 1;
    switch (x) {
        case 1: print("one"); continue case_no_2;
case_no_2:
        case 2: print("two"); break;
        default: print("something else"); break;
    }
}
```

Control Flow Instructions

1. if
2. while
3. do..while
4. for
5. switch

Format:

- switch (expression) { case valu1: ... , case value2: ... };
- Some differences from C/C++;

Using continue allows one to skip directly to a separate different case in a program execution (not necessarily the next case value).

```
void main() {
    var x = 1;
    switch (x) {
        case 1: print("one"); continue case_no_3;
        case 2: print("two"); break;
        case_no_3:
            case 3: print("three"); break;
    }
}
```

Control Flow Instructions

1. if
2. while
3. do..while
4. for
5. switch

Format:

- switch (expression) { case value1: ... , case value2: ... };
- Some differences from C/C++;

One other difference from C/C++ is that strings can be used in a switch... case statement. This implies that the switch is not optimized (in this case).

```
void main() {
    var color = "Red";
    switch (color) {
        case "Red": print("R"); break;
        case "Blue": print("B"); break;
        case "Green": print("G"); break;
    }
}
```

Custom types

Custom types

1. enums
2. typedefs

Custom types

1. enums

2. typedefs

Format: enum <name> { value₁, value₂, ... value_n}

- Enums are however different than C/C++ implementation. An enum object has several properties such as:
 - name: the name of that value
 - index: the index order (0-based) of that value

```
enum food { apple, orange, carrot }
void main() {
    var a = food.apple;
    print(a);
    print(a.name);
    print(a.index);
}
```

Output:
food.apple
apple
0

Custom types

1. enums

2. typedefs

Format: enum <name> { value₁, value₂, ... value_n}

- Enums can be iterated and all of their values listed

```
enum food { apple, orange, carrot }
void main() {
    for (var i in food.values) {
        print(i.toString()+"=>"+i.index.toString());
    }
}
```

Output:
food.apple=>0
food.orange=>1
food.carrot=>2

Custom types

1. enums

2. typedefs

Format: enum <name> { value₁, value₂, ... value_n}

- An enum however can not be of a specific type (similar to the C/C++ definition of enum class <name>:type)

```
enum food: int { apple, orange, carrot } // error
```

- Enums can not have values with specific numerical value associated (similar to C/C++)

```
enum food { apple, orange = 5, carrot } // error
```

- Enums can not be casted to an int value (as it is possible with typeless enums from C/C++);

```
enum food { apple, orange, carrot }
int i = food.apple; // error
```

Custom types

1. enums
2. **typedefs**

Format: `typedef <name> = existing_type`

- Similar to `typedef` and `using` from C/C++

```
typedef i64 = int;
void main() {
    i64 x = 10;
    print(x);
}
```

- This is in particular useful (just like in C/C++) for templates (generics). As such , `typedefs` will further be discuss at that point.

Functions

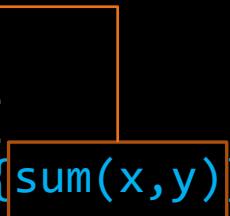
Functions

Basic form (similar to C/C++)

```
return_type <function_name>(<parameters>) { ... [return value] }
```

<return_type> can be void. If this is the case, return statement does not have to be used.

```
int sum(int x, int y) {
    return x+y; ←
}
void print_numbers(int x, int y) {
    print("x = $x, y=$y, sum=${sum(x,y)}");
}
void main() {
    print_numbers(10,20);
}
```



Functions

Basic form (similar to C/C++)

```
return_type<function_name>(<parameters>) { ... [return value] }
```

<return_type> can be omitted. In this case, the return type will be inferred from the return statement expression.

```
sum(int x, int y) { return x+y; } ←  
void print_numbers(int x, int y) {  
    print("x = $x, y=$y, sum=${sum(x,y)}");  
}  
void main() {  
    print_numbers(10,20);  
}
```

Functions

Simplified form for when the result of a function is an expression (a statement can't be used).

Pretty much , everything between “=>” and “;” must be an expression.

```
return_type <function_name>(<parameters>) => expression;
```

or “sum(int x, int y) => x+y;”



```
int sum(int x, int y) => x+y;
void print_numbers(int x, int y) {
    print("x = $x, y=$y, sum=${sum(x,y)}");
}
void main() {
    print_numbers(10,20);
}
```

Functions

Function parameters can be:

- Positional (similar to C/C++)
- Optional (with default values → similar to C/C++). These parameters must be included between [...]
- Named (similar to what Python has). Named parameters are some sort of optional parameters. These parameters must be included between {...}

A function can either use Optional or Named parameters (but can not used both types).

If Optional or Named parameters exists, they must be defined after the positional parameters (if they exist).

In case of Named parameters, a required keyword can be used to make it mandatory.

Functions

- Function with no parameters:

```
void my_function() {...}
```

- Function with positional parameters

```
void my_function(int x, int y) {...}
```

- Function with positional and optional parameters

```
void my_function(int x, [int y = 10]) {...}
```

- Function with optional parameters

```
void my_function([int x = 20, int y = 10]) {...}
```

Functions

- Function with optional parameters (**without type** that is inferred from the default value):

```
void my_function([x = 20, y = 10]) {...} // x and y are int
```

- Function with optional parameters without a default value can be used if “?” symbol is used after the type. This is translated that the specific parameter can be a null or something of its type: “int? x” means that “x” can either be an *int* value or a *null* value. In this case, the default value is not mandatory as it is implied that if don't use it, that variable will be set to null.

```
int sum([int? x, int? y]) {  
    if ((x is int) && (y is int)) return x+y;  
    if (x is int) return x;  
    return 0;  
}  
void main() { print(sum(10,20)); print(sum(10)); print(sum()); }
```


Functions

- Function with named parameters (x is required)

```
void my_function({required int x, int y = 10}) {...}
```

- Function with named parameters using “?” symbol (without default value).

```
void my_function({int? x, int? y}) {...}
```

- Function with named parameters of a defined type / class

```
void my_function({Car c, Aeroplane a}) {...}
```

In this case, if “c” and “a” can be null, a default value will be considered null (even if not specified). We will talk more about this when discussing about classes.

Functions

(lambdas – anonymous functions)

Lambdas are defined as follows

```
(parameters) {...}
```

or

```
(parameters) => expression ;
```

With parameters being defined just like in the case of regular functions.

```
void main() {
    var sum = (int x, int y) { return x+y; };
    var mul = (int x, int y) => x*y;
    print(sum(10,20));
    print(mul(5,3));
}
```

Functions

(lambdas – anonymous functions)

Instead of lambdas, the previous example can be written with inner functions:

```
void main() {
    int sum(int x, int y) {
        return x+y;
    }
    int mul(int x, int y) {
        return x*y;
    }

    print(sum(10,20));
    print(mul(5,3));
}
```

Functions (closures)

A function can be used to return a lambda. To do this, a special keyword **Function** should be used as a return type. If the return lambda uses parameters or local values, those values will be captured and used even if the function from where they were captured ends.

```
Function GetIsDivisibleBy(int n) {  
    return (int x) => x % n==0;  
}  
  
void main() {  
    var f = GetIsDivisibleBy(7);  
    print(f(21));// true  
}
```

```
Function GetIsDivisibleBy(int n) {  
    var ndiv = n+1;  
    return (int x) => x % ndiv==0;  
}  
  
void main() {  
    var f = GetIsDivisibleBy(5);  
    print(f(24));// true  
}
```

Generic functions

Very similar to a template-based function from C/C++, however more generic (no type specifier exists).

```
sum(x,y) {  
    return x+y;  
}  
void main() {  
    var v = sum(10,20);  
    print(v.runtimeType); // int  
    var v2 = sum("test","abc");  
    print(v2.runtimeType); // String  
    var v3 = sum(1.5,10);  
    print(v3.runtimeType); // double  
}
```

Generic functions

These type of function may contain some parameters that are defined with a type (including the return type) and some parameters that are define in a more generic way.

What is important is that the operation that these parameters are doing is possible:

```
sum(x,int y) {  
    return x*y;  
}  
void main() {  
    print(sum(10,20)); // 200  
    print(sum("test",3)); // testtesttest  
}
```

This example works because multiplication between two `ints` and between a `String` and an `int` are possible in Dart.

Generic functions

The evaluation for these type of functions is done at runtime. This means that parameters matching will be checked upon execution of this function. From this point of view, these type of functions are very similar to what Python has (in terms of no types for functions).

```
sum(x,int y) {  
    return x*y;  
}  
void main() {  
    print(sum(10,20)); // 200  
    print(sum(true,3)); // runtime error  
}
```

Unhandled exception:
NoSuchMethodError: Class 'bool' has no instance method '*'.

Generic functions

This means that one can use these functions to return different types (similar to what Python does). The following example demonstrates this ability.

```
foo(int x) {
    if (x>100)
        return "result";
    else
        return true;
}
void main() {
    print(foo(200)); // result
    print(foo(10)); // true
}
```

Generic functions

The same property can be achieved using dynamic type as a return type.

```
dynamic foo(int x) {
    if (x>100)
        return "result";
    else
        return true;
}
void main() {
    print(foo(200)); // result
    print(foo(10)); // true
}
```

Q & A



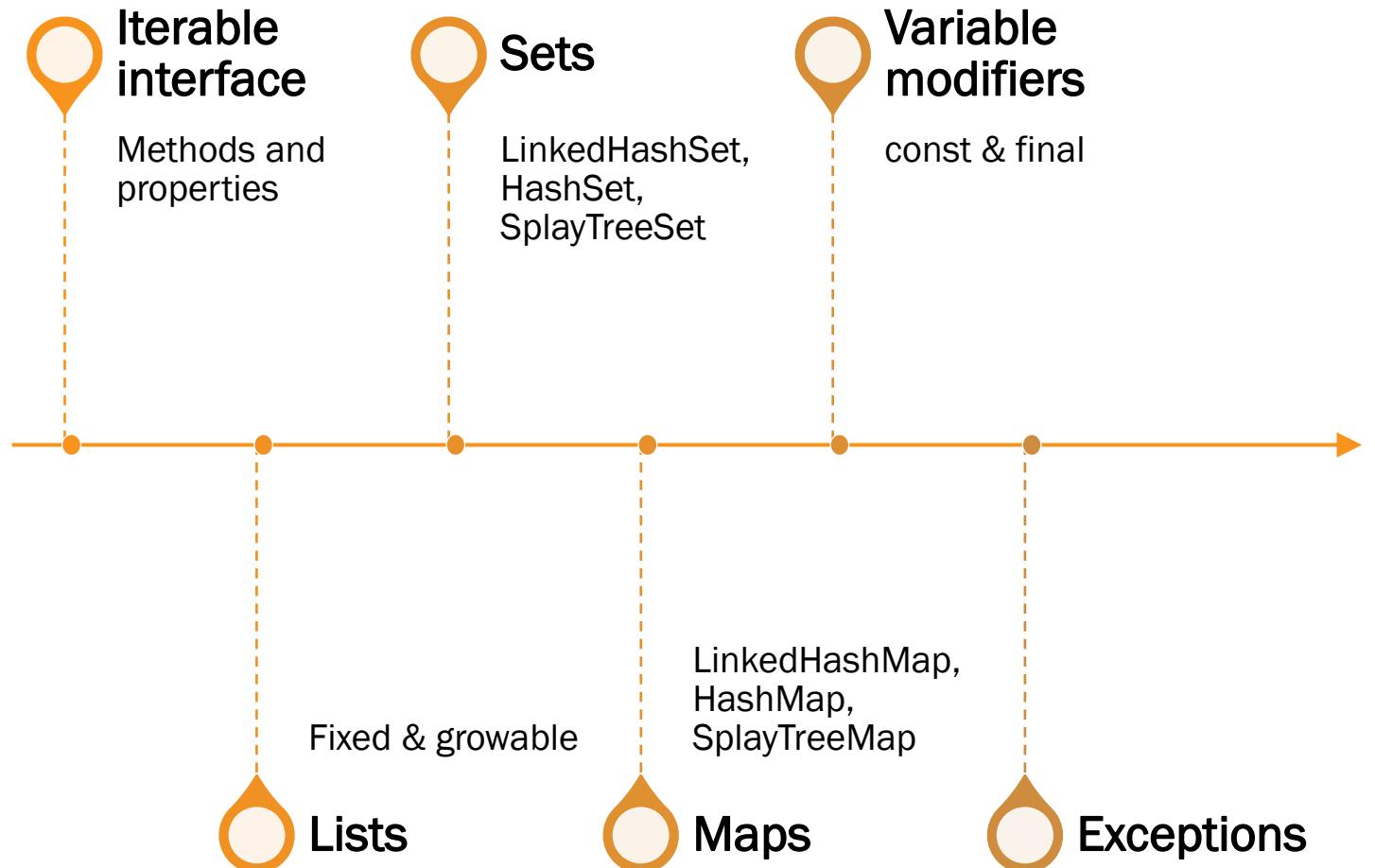


DART Language

COURSE 2 (REV 6)

GAVRILUT DRAGOS

Agenda



Iterable

Iterable

Dart provides an interface (abstract class) called ***Iterable*** that offers the basic functionality of every container. Lists / sets and maps are templates derived from ***Iterable***. One can not create an object of type ***Iterable***, however a list/set or map can be casted to an ***Iterable*** object.

Properties:

T	Iterable<T>.first	Returns the first element from the collection or StateError exception otherwise
T	Iterable<T>.last	Returns the last element from the collection or StateError exception otherwise
T	Iterable<T>.single	Returns the first element from the collection if the collection has only ONE element or StateError exception otherwise
int	Iterable<T>.length	Number of elements in the collection
bool	Iterable<T>.isEmpty	True if the collection is empty, false otherwise
bool	Iterable<T>.isNotEmpty	False if the collection is empty, true otherwise

Iterable

Casting methods (used to convert from one iterable to another one).

<code>Iterable<U> Iterable<T>.cast<U> ()</code>	Creates a new iterable where every T element is casted out to a U element.
<code>Set<T> Iterable<T>.toSet ()</code>	Creates a Set of type T from the existing Iterable
<code>List<T> Iterable<T>.toList ()</code>	Creates a List of type T from the existing Iterable

Iterable methods (used to create another Iterable from the existing one based on some conditions).

<code>Iterable<T> Iterable<T>.skip(int count)</code>
<code>Iterable<T> Iterable<T>.skipWhile(bool validate(T object))</code>
<code>Iterable<T> Iterable<T>.take(int count)</code>
<code>Iterable<T> Iterable<T>.takeWhile(bool validate(T object))</code>
<code>Iterable<T> Iterable<T>.where(bool retainIfTrueFunction(T object))</code>

Iterable

Mapping values methods (used to convert from one iterable to another type (including another iterator) by converting all of its values).

```
Iterable<U> Iterable<T>.map<U> (U convertFunction(T object))
```

```
Iterable<T> Iterable<T>.expand(Iterable<T> convertToIterable(T object))
```

```
T Iterable<T>.reduce(T combineFunction(T value, T object))
```

```
bool Iterable<T>.any(bool validate(T object))
```

```
bool Iterable<T>.every(bool validate(T object))
```

```
String Iterable<T>.join([Separator = ""])
```

Iterable also provides a generic .forEach method to easily iterate through all elements from a collection.

```
void Iterable<T>.forEach (void processElement(T element))
```

Iterable

Finding values methods (used to find an element or check the existence of elements in a collection).

```
bool Iterable<T>.contains(T? object)
T Iterable<T>.elementAt(int index)
T Iterable<T>.firstWhere(bool validate(T object)), {T orElse()?}
T Iterable<T>.lastWhere(bool validate(T object)), {T orElse()?}
```

We will discuss more about these methods when discussion one of their implementations (List).

Lists

Lists

DART has two type of lists:

- Fixed sized
- Growable

Characteristics:

- Dart lists hold elements of the same type and use “[...]” to define them.
- There is a special type of parametrized list that allows storing elements of different types.
- Dart lists are based on templates/generics. This mean that upon construction, list element type can be provided. However, if not provided, Dart can infer it from initialization parameters.
- Dart lists work based on references to an object (exception for this case are basic type (number, bool and string) that are copied).

Lists

Creating an `int` list with some values:

```
void main() {  
    var l = [1,2,3];  
    print(l);  
}
```

Creating an empty `int` list :

```
var l = <int>[]; // type cannot be inferred so it has to be specified
```

A list is a template of type `List<T>`. As such, `List<T>` can also be used to construct an object. However, in case of list there is no constructor but several static methods that can be used to create an object.

```
void main() {  
    var l = [1,2,3];  
}
```

is equivalent to

```
void main() {  
    var l = List<int>.from([1,2,3]);  
}
```

Lists

Dar lists have several properties that can be used: `.first` , `.last` , `.length`, `.reversed` , `.isEmpty` , `.isNotEmpty`

```
void main() {  
    var l = ["hello", "dart", "zzz", "alongstring"];  
    print(l.first);          // hello  
    print(l.last);           // alongstring  
    print(l.isEmpty);        // false  
    print(l.isNotEmpty);     // true  
    print(l.reversed);       // (alongstring, zzz, dart, hello)  
    print(l.length);          // 4  
}
```

Lists

List constructors / static methods that can be used to create a list:

```
List<T>.empty({bool growable = false})  
List<T>.filled(int len, T value, {bool growable = false})  
List<T>.generate(int len, T generatorFnc(int index), {bool growable = false})  
List<T>.of(Iterable<T> elements, {bool growable = false})  
List<T>.from(Iterable elements, {bool growable = false})  
List<T>.unmodifiable(Iterable elements)
```

Keep in mind that `List<T>` is not always necessary as dart will attempt to infer type `T` from parameters. This is why there are two very similar version (`.from` and `.of` → they can help inference process understand type `T`);

Lists

All objects from Dart are derived from one singular **dynamic** type (`Object`). As such if a type can not be inferred, it is considered to be the base object (`dynamic`).

```
void main() {  
    var l1 = [1,2,3];  
    print(l1.runtimeType);  
    var l2 = [];  
    print(l2.runtimeType);  
}
```

```
void main() {  
    var l = List<int>.from([1,2,3]);  
    print(l.runtimeType); // List<int>  
    var l2 = List.empty();  
    print(l2.runtimeType); // List<dynamic>  
    var l3 = List<int>.empty();  
    print(l3.runtimeType); // List<int>  
}
```

List<int>
List<dynamic>

OBS: The output type will be different if you run the same examples in Dart Pad (JavaScript based) or compile them directly.

List<int>
List<dynamic>
List<int>

Lists

The main difference between .from and .of constructors if the .from takes in consideration the parametrize type while .of considers the initialization values as well for the inference process.

```
void main() {
    var l1 = List<int>.from([1,2,3]);
    print(l1.runtimeType);                      // l1 = List<int>
    var l2 = List.from([1,2,3]);
    print(l2.runtimeType);                      // l2 = List<dynamic>
//-----
    var l3 = List<int>.of([1,2,3]);
    print(l3.runtimeType);                      // l3 = List<int>
    var l4 = List.of([1,2,3]);
    print(l4.runtimeType);                      // l4 = List<int>
}
```

Lists

To populate a list based on a mathematical relation between list index and its value, use `generate` constructor. This is similar to map paradigm from Python. Both lambdas and functions can be used.

```
int myFunction(int index) {
    return index % 3;
}
void main() {
    var l = List<int>.generate(5,(i)=>i*i); // i=int (inferred)
    print(l); // [0, 1, 4, 9, 16]
    var l2 = List<int>.generate(5,(int i)=>i*i*i); // i=int (declared)
    print(l2); // [0, 1, 8, 27, 64]
    var l3 = List<int>.generate(5,myFunction);
    print(l3); // [0, 1, 2, 0, 1]
}
```

Lists

The existence of `List<dynamic>` allows one to create a heterogenous type list.

```
void main() {  
    var l = [10, "test", 1.5];  
    print(l);  
    print(l[0].runtimeType);  
    print(l[1].runtimeType);  
    print(l[2].runtimeType);  
}
```

[10, test, 1.5]
int
String
double



```
void main() {  
    var l = <int>[10, "test", 1.5];  
}
```



In this case a compile error will be triggered as "l" has to contain int elements

Lists

To add an element to a list use `add` or `addAll` methods:

```
List<T>.add(T value)
```

```
List<T>.addAll(Iterable<T> elements)
```

Operator `+=` can also be used (similar to Python lists). However, the behavior is different as `+=` is an operator defined in the following way: $T+=V \Leftrightarrow T = T + V$. This means that the new element(s) is/are not added to the existing list, but a new list is created that is the sum of the first two.

```
void main() {  
    var l = [1,2,3];  
    l.add(4);  
    print(l); // [1,2,3,4]  
    l.addAll([10,20,30]);  
    print(l); // [1,2,3,4,10,20,30]  
}
```

or

```
void main() {  
    var l = [1,2,3];  
    l += [4];  
    print(l); // [1,2,3,4]  
    l += [10,20,30];  
    print(l); // [1,2,3,4,10,20,30]  
}
```

Lists

So why use `add` or `addAll` instead of `+=` operator ?

```
void main() {  
    var l = List<int>.from([1,2,3], growable: false);  
    print(l);  
    l.add(5);  
}
```

OBS: List “l” is defined as a fixed list (non-growable).
This means that adding new elements to the list will
throw an error.

Upon execution an “**Unsupported operation: Cannot add to a fixed-length list**” will be thrown.

Lists

So why use `add` or `addAll` instead of `+=` operator ?

Now let's write the previous example, but use `+=` instead of `add`

```
void main() {  
    var l = List<int>.from([1,2,3], growable: false );  
    l += [4];  
    l.add(5);  
    print(l); // [1,2,3,4,5]  
}
```

The following syntax: `l += [4];` will be execute as `l = l + [4]`. Meaning that a new list will be created that is the result of the sum between the original “`l`” list and `[4]`. The new list will be growable. As a result, `l.add(5)` will work as expected !

Lists

The “**growable**” parameter is extremely useful for array creation. The following code creates an array of 5 int elements. Methods like **add** or **addAll** can not be used. However, elements can be accessed and modified via the **[index]** operator.

To create an array-like list use the **List<T>.filled(int len, T value)** constructor.

```
void main() {
    var l = List<int>.filled(5,0, growable: false);
    print(l); // [0, 0, 0, 0, 0]
    l[0] = 1;
    l[1] = 2;
    l[3] = 3;
    print(l); // [1, 2, 0, 3, 0]
    l[10] = 100; // index out of bounds exception during runtime
}
```

Lists

The same technique can be used to create a bi-dimensional array.

```
void main() {
    var l = List<List<int>>.filled(
        5, // 5 rows
        List<int>.filled(
            3, // 3 columns
            0, // default value for all elements
            growable: false),
        growable: false);
    print(l); // [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
}
```

This code will create a 5x3 matrix (5 rows and 3 columns).

Lists

Another way to create a list is to use the cascade operator “`..`” with the method “`addAll`”

```
void main() {  
    var l1 = <int>[]..addAll([1,2,3]);  
    print(l1.runtimeType); // List<int>  
    var l2 = []..addAll([1,2,3]);  
    print(l2.runtimeType); // List<dynamic>  
}
```

If using this method and a type-specific list is desired, make sure that you prefix the empty list creation with the type you want !

In the previous example, l1 is a `List<int>`, while l2 is a `List<dynamic>`

Lists

To insert an element into a list use `insert` or `insertAll` methods:

```
List<T>.insert(int index, T value)
```

```
List<T>.insertAll(int index, Iterable<T> elements)
```

Use `List<T>.length` to insert an element at the end of the list.

```
void main() {
    var l = ["A", "B", "C", "D"];
    l.insert(2, "**");
    print(l); // [A, B, **, C, D]
    l.insertAll(0, ["Q", "W", "E"]);
    print(l); // [Q, W, E, A, B, **, C, D]
    l.insert(l.length, "END");
    print(l); // [Q, W, E, A, B, **, C, D, END]
}
```

Lists

To insert an element into a list use `insert` or `insertAll` methods:

`List<T>.insert(int index, T value)`

```
List<T>.insertAll(int index, Iterable<T> elements)
```

Using a position outside the boundaries of the list will trigger a runtime exception:

Lists

To remove an element from a list, the following methods can be used:

```
bool List<T>.remove(T? object)
T List<T>.removeAt(int index)
void List<T>.removeRange(int start, int end)
void List<T>.removeWhere(bool removeIfTrueFunction(T object))
void List<T>.retainWhere(bool retainIfTrueFunction(T object))
T List<T>.removeLast()
void List<T>.clear()
```

What is interesting to observe is that `removeAt` and `removeLast` also return the object that was removed from the list. This feature actually makes them easily to use for stacks or queues (as they simulate a pop instruction).

Use `.clear` to remove all elements from a list.

Lists

```
void main() {
    var l = [1,2,3,4,5,6,7,8,9,10];
    print(l.removeLast());           // 10
    print(l);                      // [1, 2, 3, 4, 5, 6, 7, 8, 9]
    print(l.removeAt(2));          // 3
    print(l);                      // [1, 2, 4, 5, 6, 7, 8, 9]
    l.removeRange(1,3);
    print(l);                      // [1, 5, 6, 7, 8, 9]
    l.removeWhere((i)=>i%2==0);
    print(l);                      // [1, 5, 7, 9]
    print(l.remove(1000));         // false
    print(l.remove(1));            // true
    print(l);                      // [5, 7, 9]
}
```

Lists

To find or check the existence of an element in a list use the following APIs:

```
bool List<T>.contains(T? object)
int List<T>.indexAt(T object, [int start = 0])
int List<T>.lastIndexAt(T object, [int start?])
int List<T>.indexWhere(bool validate(T object)), [int start = 0]
int List<T>.lastIndexWhere(bool validate(T object)), [int start?]
T List<T>.firstWhere(bool validate(T object)), {T orElse()?}
T List<T>.lastWhere(bool validate(T object)), {T orElse()?}
```

The return of index based functions is either -1 if the element was not found or the position in the list.

Lists

```
void main() {
    var l = [1,2,3,4,5,6,7,8,9,10];
    print(l.indexOf(1000));                                // -1 (1000 is not
                                                          // in the list)
    print(l.indexOf(4));                                  // 3 (4 has index 3 on
                                                          // a 0-based list)
    print(l.indexWhere((i)=>i%2==0));                  // 1 (l[1]==2)
    print(l.lastIndexWhere((i)=>i%2==0));              // 9 (l[9]==10)
                                                          // (Last odd number)
    print(l.contains(5));                                // true
    print(l.contains(1000));                            // false
}
```

Lists

Lists

Just like Python that has several methods to automatically process a list and obtain another one (such as filter or map), Dart has some similar methods as well:

```
Iterable<T> List<T>.getRange(int start, int end)
Iterable<U> List<T>.map<U> (U convertFunction(T object))
Iterable<T> List<T>.skip(int count)
Iterable<T> List<T>.take(int count)
Iterable<T> List<T>.where(bool retainIfTrueFunction(T object))
List<T>     List<T>.sublist(int start, [int? end])
Iterable<T> List<T>.expand(Iterable<T> convertToIterable(T object))
T           List<T>.reduce(T combineFunction(T value, T object))
```

As each one of those methods produce an object, they can be used in a chained expression.

Lists

```
void main() {  
    var l = [1,2,3,4,5];  
    var l2 = l.map<int>((i)=>i*i);          // (1, 4, 9, 16, 25)  
    var l3 = l2.skip(2);                      // (9, 16, 25)  
    var l4 = l3.where((i)=>i%2!=0);          // (9, 25)  
}
```

or we can obtain the same results via a chain of commands such as:

```
void main() {  
    var l = [1,2,3,4,5];  
    var l2 = l.map<int>((i)=>i*i)  
        .skip(2)  
        .where((i)=>i%2!=0);  
    print(l2);  
}
```

Lists

Keep in mind that the previous methods (except for `sublist`) do not create a list but an Iterable object. To create a list use `.toList()` method or `.toList(growable:true)` for a fix sized list.

```
void main() {  
    var l = [1,2,3,4,5];  
    var l2 = l.map<int>((i)=>i*i).skip(2).where((i)=>i%2!=0).toList(); // [9, 25]  
}
```

Dart lists also inherit a `.forEach` method that can vbe used to quickly iterate through all elements:

```
void List<T>.forEach(void action(T object))
```

```
void main() {  
    var l = [1,2,3,4,5];  
    int s = 0;  
    l.forEach( (i) => s+=i );  
    print(s); // 15  
}
```

→ the same result can be achieved via `.reduce` method

```
void main() {  
    var l = [1,2,3,4,5,6];  
    print(l.reduce((sum,i)=>sum+=i)); // 21  
}
```

Lists

To sort a list use `sort` method:

```
List<T>.sort([int compareFunction(T object1, T object2)?])
```

Compare function (if present) must return “1” if object1 is bigger than object2, “-1” if object1 is smaller than object2 or “0” if they are equal.

```
void main() {
    var l = ["hello", "dart", "zzz", "alongstring"];
    l.sort();
    print(l); // [alongstring, dart, hello, zzz]
    l.sort((i,j)=>i.length.compareTo(j.length));
    print(l); // [zzz, dart, hello, alongstring]
}
```

Lists

To sort a list use `sort` method:

```
List<T>.sort([int compareFunction(T object1, T object2)?])
```

Compare function (if present) must return “1” if object1 is bigger than object2, “-1” if object1 is smaller than object2 or “0” if they are equal.

```
void main() {
    var l = ["hello", "dart", "zzz", "alongstring"];
    l.sort();
    print(l); // [alongstring, dart, hello, zzz]
    l.sort((i,j)=>i.length.compareTo(j.length));
    print(l); // [zzz, dart, hello, alongstring]
}
```

Lists

A dart list store references. In the following example, “`l.add(t)`” will copy a reference to variable “`t`” into the list, and not a copy of that variable. The solution in this case is to make sure that you make a copy of an object before copying it into a list.

```
class Test {  
    int x = 0;  
}  
void main() {  
    var t = Test();  
    t.x = 10;  
    var l = <Test>[];  
    l.add(t);  
    t.x = 20;  
    print(l[0].x); // 20  
}
```

```
class Test {  
    int x = 0;  
    Test clone() { return Test()..x = x; }  
}  
void main() {  
    var t = Test();  
    t.x = 10;  
    var l = <Test>[];  
    l.add(t.clone());  
    t.x = 20;  
    print(l[0].x); // 10  
}
```

Lists

All command line arguments are sent as a list of strings.

```
void main(List<String> arguments)
{
    print(arguments); // [1, 2, 3, 4]
}
```

To teste the previous program, compile it view “`dart.exe compile exe <name>`” an them runt the newly obtain executable file like this: “`<name>.exe 1 2 3 4`”.

There are some specialized modules for parsing command line (similar to what python has). We will discuss more about them when talking about libraries.

Sets

Sets

Sets are containers where each elements is unique. There are several implementations for sets:

- HashSet
- LinkedHashSet
- SplayTreeSet
- Set

Characteristics:

- To create a set use “[...]” to define them.
- Sets are of type ***Iterable*** meaning that will inherit all their properties

Sets

Creating an `int` set with some values:

```
void main() {  
    var s = {1,1,5,6,7,5,3};  
    print(s); // {1, 5, 6, 7, 3}  
}
```

Creating an empty `int` set :

```
var s = <int>{}; // type cannot be inferred so it has to be specified  
or  
var s = Set<int>();
```

Sets

Constructors / static methods that can be used to create a set:

```
Set<T>.identity()  
Set<T>.of(Iterable<T> elements)  
Set<T>.from(Iterable elements)  
Set<T>.unmodifiable(Iterable elements)
```

```
void main() {  
    var s1 = Set<int>.identity(); // s1 = <int>{}  
    var s2 = Set<int>.from([1,2,3,1,2,3]); // s2 = <int>{1,2,3}  
}
```

Sets

To add an element to a set use `add` or `addAll` methods:

```
Set<T>.add(T value)
```

```
Set<T>.addAll(Iterable<T> elements)
```

As a difference from List, '`+=`' can not be used as Sets don't have the operator `+` defined !

```
void main() {
    var s = {1,2,3};
    s.add(4);
    print(s); // {1,2,3,4}
    s.addAll([1,3,5]);
    print(s); // {1,2,3,4,5}
}
```

Set

To remove an element from a set, the following methods can be used:

```
bool Set<T>.remove(T? object)
void Set<T>.removeAll(Iterable<T>? elements)
void Set<T>.removeWhere(bool removeIfTrueFunction(T object))
void Set<T>.retainWhere(bool retainIfTrueFunction(T object))
void Set<T>.retainsAll(Iterable<T>? elements)
void Set<T>.clear()
```

Use `.clear` to remove all elements from a set.

The rest of the methods work in a similar manner as they work for a list.

Sets

```
void main() {
    var s = {1,2,3,4,5,6,7,8};
    print(s); // {1, 2, 3, 4, 5, 6, 7, 8}
    s.removeWhere( (e)=>e%2==0 );
    print(s); // {1, 3, 5, 7}
    s.removeAll({1,3,9,10,11});
    print(s); // {5, 7}
}
```

Retain methods work in a similar manner.

Set

Sets specific operations:

```
Set<T> Set<T>.intersection(Iterable<T?> elements)
Set<T> Set<T>.difference(Iterable<T?> elements)
Set<T> Set<T>.union(Iterable<T?> elements)
```

There is no method for asymmetric difference.

```
void main() {
    var s1 = {1,2,3,4};
    var s2 = {3,4,5,6};
    print(s1.intersection(s2)); // {3, 4}
    print(s1.union(s2));      // {1, 2, 3, 4, 5, 6}
    print(s1.difference(s2)); // {1, 2}
    print(s2.difference(s1)); // {5, 6}
}
```

Set

To check if an element is present in a set, use the following methods:

```
T? Set<T>.lookup(T element)
```

```
bool Set<T>.containsAll(Iterable<T?> elements)
```

```
bool Set<T>.contains(T? element)
```

```
void main() {
    var s = {1,2,3,4};
    print(s.contains(2));                // true
    print(s.contains(200));              // false
    print(s.lookup(2));                 // 2
    print(s.lookup(200));               // null
    print(s.containsAll({1,2,3}));      // true
    print(s.containsAll([1,2,1,2,1,2])); // true
    print(s.containsAll({1,2,4,5}));    // false
}
```

Maps

Maps

Maps are (key,value) containers where each key is unique. There are several implementations for maps:

- HashMap
- LinkedHashMap
- SplayTreeMap
- Map

Characteristics:

- To create a map use “[... :]” to define them (similar to Python format)
- Maps inherit **Iterable** meaning that will inherit all their properties and method

Maps

Creating an `<String,int>` map with some values:

```
void main() {  
    var m = {"Popescu":10,"Ionescu":9,"Georgescu":7};  
    print(m); // {Popescu: 10, Ionescu: 9, Georgescu: 7}  
}
```

Creating an empty `<String,int>` map :

```
var s = <String,int>{}; // type cannot be inferred so it has to be specified  
or  
var s = Map<String,int>();
```

Maps

Constructors / static methods that can be used to create a map:

```
Map<K, V>.identity()
```

```
Map<K, V>.of(Map<K, V> otherMap)
```

```
Map<K, V>.from(Map otherMap)
```

```
Map<K, V>.unmodifiable(Map otherMap)
```

```
void main() {
    var m1 = Map<String, int>.identity(); // m1 = <String, int>{}
    var m2 = Map<int, int>.from({1:1, 2:4}); // m2 = <int, int>{1:1, 2:4}
}
```

Maps

Constructors / static methods that can be used to create a map:

```
Map<K, V>.fromEntries(Iterable<MapEntry<K, V>> entries)
```

```
Map<K, V>.fromIterables(Iterable<K> keys, Iterable<V> values)
```

```
Map<K, V>.fromIterable(Iterable elements, { K key(dynamic elem),  
                                              V value(dynamic elem) })
```

A *MapEntry* is similar to pair from STL and is constructed in the following way:

```
MapEntry<K, V>(K key, V value)
```

```
MapEntry(K key, V value)
```

A *MapEntry* has the following properties:

<code>K MapEntry<K, V>.key</code>	Returns the key from the (key,value) pair
<code>V MapEntry<K, V>.value</code>	Returns the value from the (key,value) pair

Maps

Creating a map using `.fromEntries`:

```
void main() {
    var m = Map.fromEntries([ MapEntry("Popescu",10),
                            MapEntry("Georgescu",7),
                            MapEntry("Ionescu",9) ]);
    print(m); // {Popescu: 10, Georgescu: 7, Ionescu: 9}
}
```

Creating a map using `.fromIterables`

(OBS: Make sure that keys and values iterables have the same length).

```
void main() {
    var m = Map.fromIterables(["Popescu","Georgescu","Ionescu"],[10,8,6]);
    print(m); // {Popescu: 10, Georgescu: 8, Ionescu: 6}
}
```

Maps

Creating a map using `.fromIterable`:

```
void main() {
    var list = [
        ["Popescu" , 10],
        ["Georgescu", 8],
        ["Ionescu" , 6]
    ];
    var m = Map.fromIterable(list,
                            key:(e)=>e[0],      // first entry = key
                            value:(e)=>e[1]);   // second entry = value
    print(m); // {Popescu: 10, Georgescu: 8, Ionescu: 6}
}
```

Maps

To add an element to a map use `addEntries` , `addAll` methods or `[]` operator:

```
Map<K,V>.addEntries(Iterable<MapEntry<K,V>> entries)  
Map<K,V>.addAll(Map<K,V> otherMap)
```

```
void main() {  
    var m = <String,int>{};  
    m["Popescu"] = 9;  
    print(m); // {Popescu: 9}  
    m.addEntries([MapEntry("Ionescu",10),MapEntry("Georgescu",7)]);  
    print(m); // {Popescu: 9, Ionescu: 10, Georgescu: 7}  
    m.addAll({"Teodorescu":7});  
    print(m); // {Popescu: 9, Ionescu: 10, Georgescu: 7, Teodorescu: 7}  
    m["Georgescu"] = 6;  
    print(m); // {Popescu: 9, Ionescu: 10, Georgescu: 6, Teodorescu: 7}  
}
```

Maps

To update an element to from the map use `update` , `updateAll` , `putIfAbsent` methods or `[]` operator:

```
V Map<K,V>.putIfAbsent(K key, V ifAbsentFunction() )
V Map<K,V>.update(K key, V updateFunction(V value), {V ifAbsent()?})
void Map<K,V>.updateAll(V updateFunction(K key, V value))
```

```
void main() {
    var m = {"Pop":10,"Geo":8};
    m.putIfAbsent("Ion", ()=>5); // m = {Pop: 10, Geo: 8, Ion: 5}
    m.update("Pop", (v)=>v-2); // m = {Pop: 8, Geo: 8, Ion: 5}
    m.update("Ionel", (v)=>10, ifAbsent: ()=>8); // m = {Pop: 8, Geo: 8,
                                                    //           Ion: 5, Ionel: 8}
    m.updateAll((k,v)=>v-4); // m = {Pop: 4, Geo: 4, Ion: 1,
                            //           Ionel: 4}
}
```

Maps

To remove an element from a map, the following methods can be used:

```
bool Map<K,V>.remove(K? key)
void Map<K,V>.removeWhere(bool removeIfTrueFunction(K key, V value))
void Map<K,V>.clear()
```

```
void main() {
    var m = {"Raul":10,"Alina":8,"Ion":5,"Georgiana":7};
    print(m); // {Raul: 10, Alina: 8, Ion: 5, Georgiana: 7}
    m.remove("Ion");
    print(m); // {Raul: 10, Alina: 8, Georgiana: 7}
    m.removeWhere((k,v)=>k.length>6);
    print(m); // {Raul: 10, Alina: 8}
}
```

Maps

To check if an element is present in a map, use the following methods:

```
bool Map<K,V>.containsKey(K? key)
```

```
bool Map<K,V>.containsValue(V? value)
```

```
void main() {
    var m = {"Alin":7,"Marilena":8,"Dragos":5};
    print(m.containsKey("Alin"));           // true
    print(m.containsValue(5));             // true
    print(m.containsKey("John"));          // false
    print(m.containsValue(2));             // false
}
```

Maps

A map object has the following properties:

<code>Iterable<MapView<K,V>> Map<K,V>.entries</code>	List of (key,value) entries
<code>Iterable<K> Map<K,V>.keys</code>	List of key entries
<code>Iterable<V> Map<K,V>.values</code>	List of key values

```
void main() {
    var m = {"Alin":7,"Marilena":8,"Dragos":5};
    print(m.values); // (7, 8, 5)
    print(m.keys); // (Alin, Marilena, Dragos)
    for (var i in m.entries) {
        print(i);
    }
}
```

MapEntry(Alin: 7)
MapEntry(Marilena: 8)
MapEntry(Dragos: 5)



Variable modifiers

Variable modifiers

There are several variable modifiers in Dart that can be used to declare a variable:

- const
- final

A **const** variable will be treated like a value that is determined and evaluated during compile time (similar to what **constexpr** is in C++).

A **final** variable is a variable that can not be changed (similar to **final** in Java or **const** in C/C++).

The main difference between a **const** and a **final** variable is that **const** variable must be initialized with a compile-time known value, while a **final** var could be initialized with any value.

Variable modifiers

A const or a final variable can not be modified:

```
void main() {  
    const int x = 10;  
    x = 20; // Error: Can't assign  
            // to the const variable  
            // 'x'.  
}
```

```
void main() {  
    final int x = 10;  
    x = 20; // Error: Can't assign  
            // to the final variable  
            // 'x'.  
}
```

Both **const** and **final** can be used with classes (we will further discuss this when talking about classes). When dealing with objects, a **const** object is different from a **final** object because the fields of a **final** object can be changed, but the fields from a **const** object can not.

Variable modifiers

Some difference between const and final:

```
import 'dart:math';

void main() {
    final x = Random().nextInt(100);
    print(x);
}
```

```
import 'dart:math';

void main() {
    const x = Random().nextInt(100);
    print(x);
}
```

The second code (the one that uses `const` will not compile, as a random value is not something that can be evaluated at compile time.

Exceptions

Exceptions

Dart exceptions have the following format:

```
try {  
  ...  
} catch (e) {  
  ...  
}
```

```
try {  
  ...  
}  
on <ExceptionType1> [catch (e)] {  
  ...  
}  
on <ExceptionType2> [catch (e)] {  
  ...  
}  
...  
catch (e) {  
  ...  
}
```

```
try {  
  ...  
}  
on <ExceptionType1> [catch (e)] {  
  ...  
}  
...  
catch (e) {  
  ...  
}  
finally (e) {  
  ...  
}
```

Exceptions

Example:

Exceptions

Example:

```
void main() {
    dynamic x = "Test";
    try {
        x+=2;
    } catch (e) {
        print("Error=>$e"); // Error=>type 'int' is not a subtype of
                            // type 'String' of 'other'
    } finally {
        print("Done");
    }
}
```

Exceptions

To throw and exception use **throw** keyword:

```
void foo(int x) {
    if (x>10)
        throw "Invalid x = ${x}";
}
void main() {
    try {
        foo(20);
    } catch (e) {
        print("Error = $e"); // Error = Invalid x = 20
    }
}
```

Exceptions

catch can receive 2 arguments (exception and the stack trace):

```
void foo(int x) {  
    if (x>10)  
        throw "Invalid x = ${x}";  
}  
void main() {  
    try {  
        foo(20);  
    } catch (e,s) {  
        print("Error = ${e}");  
        print(s);  
    }  
}
```



Invalid x = 20
at Object.wrapException (<anonymous>:352:17)
at Object.throwExpression (<anonymous>:366:15)
at main (<anonymous>:2561:11)
at <anonymous>:3128:7
at <anonymous>:3111:7
at dartProgram (<anonymous>:3122:5)
at <anonymous>:3130:3
at replaceJavaScript (<https://dartpad.dev/scripts/frame.js>:19:19)
at messageHandler (<https://dartpad.dev/scripts/frame.js>:91:13)

Exceptions

Use **rethrow** from a catch statement to re-send an exception to the next outer try...catch block.

```
void main() {
    try {
        try {
            throw "Some exception";
        } catch (e) {
            print("Error = $e"); // Error = Some exception
            rethrow;
        }
    } catch (e) {
        print("Second catch"); // Second catch
    }
}
```

Q & A



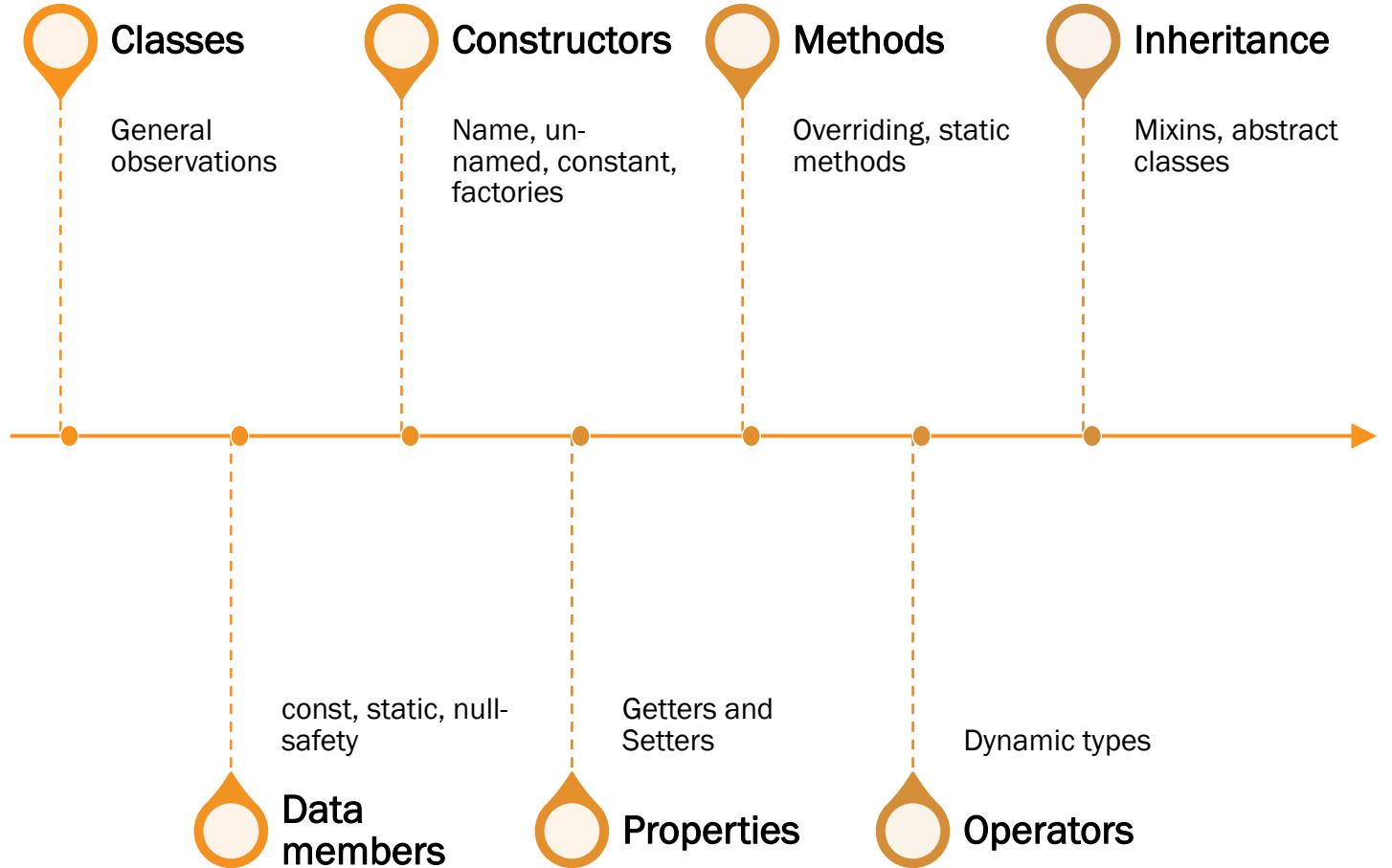


DART Language

COURSE 3 (REV 5)

GAVRILUT DRAGOS

Agenda



Classes

Classes

Dart provides several mechanisms that can be used to create an object (an instance of a class).

The way a class is defined in Dart is a mix of properties derived from both Java, C++ and C# that include getter / setter, proxy constructors, operators, etc.

To create a simple class → use the **class** keyword. All classes are implicitly derived from Object (meaning that each class has the following methods/properties:

<code>int hashCode</code>	A hash code for this object
<code>Type runtimeType</code>	Type of the object
<code>dynamic noSuchMethod(Invocation i)</code>	Call whenever a non-existing property is called
<code>String toString()</code>	A string representation for current object

Classes

A very simple example:

```
class Test
{
    // an empty class
}
void main() {
    var t = Test();
    print(t.runtimeType); // Test
    print(t.hashCode);   // 7552.....
    print(t.toString()); // Instance of 'Test'
}
```

Classes

A DART class consists in

- One or more constructors (including named constructors)
- Operators
- Getter and Setters (for properties)
- Data members (including static data members)
- Methods

A DART class does not have a destructor (this is not required as DART garbage collector takes care of this cases).

Data members

Data members

Defining a data member in a class can be done in the following way

```
class Test {  
    var v1[,v2,v3,... vn];           // one or multiple variables defined  
}
```

Or using their type:

```
class Test {  
    <type> v1[,v2,v3,... vn];           // one or multiple variables defined  
}
```

The initial value of a data member can be specified at this point

```
class Test {  
    <type or var> v = <value>;    // a data member with a value  
}
```

Data Members

In this case, “y” type will be inferred, however “x” type will be a dynamic type (similar to what python has). As a result variable “x” can be changed (in terms of its type during runtime).

```
class Test {  
    var x;  
    var y = 10; // type will be inferred  
}  
void main() {  
    var t = Test();  
    print(t.x.runtimeType); // Null  
    print(t.y.runtimeType); // int  
    t.x = 10;  
    print(t.x.runtimeType); // int  
    t.x = "bla bla bla";  
    print(t.x.runtimeType); // String  
}
```

Data Members

However, the same logic does not work for inferred types. For example, “y” type is considered “int” and can not be changed during runtime.

```
class Test {  
    var x;  
    var y = 10; // type will be inferred as int  
}  
void main() {  
    var t = Test();  
    t.y = "bla bla bla"; // Error: A value of type 'String' can't be  
                          // assigned to a variable of type 'int'.  
}
```

Data Members

All data members (except for generic ones like the ones defined using `var` keyword must be initialized). To avoid this either initialize that data member or declare it using “`?`” specifier (this will set its value to `null` if not defined).

```
class Test {  
    int x; // Field 'x' should be initialized because its type 'int' doesn't  
           // allow null.  
}
```

Data Members

All data members (except for generic ones like the ones defined using `var` keyword must be initialized). To avoid this either initialize that data member or declare it using “`?`” specifier (this will set its value to `null` if not defined).

```
class Test {  
    int? x;  
    int? y = 20;  
    int z = 30;  
}  
void main() {  
    var t = Test();  
    print(t.x); // null  
    print(t.y); // 20  
    print(t.z); // 30  
}
```

Data Members

All data member can be declared as read-only by using the final keyword. In this case, its value can't be modified after its initialization.

```
class Test {  
    final int x = 30;  
}  
void main() {  
    var t = Test();  
    print(t.x); // 30  
    t.x = 40; // Error: The setter 'x' isn't defined for the class 'Test'.  
}
```

In this case, the type of Dart requires a setter / getter (we will discuss more about this cases in the next slides).

Data Members

Dart also allows static data members (that belong to the class and not the instance). They can be defined by using the `static` keyword.

```
class Test {  
    static int x = 30;  
}  
void main() {  
    print(Test.x); // 30  
}
```

As a difference between Dart and C++, a static member can not be accessed via an instance !

```
void main() {  
    var t = Test();  
    print(t.x); // Error: The getter 'x' isn't defined for the class 'Test'.  
}
```

Data Members

A static member can however be defined as a **static const** (similar to **constexpr** format from C++). This will produce some compile time optimization where that variable will be replaced with its value.

```
class Test {  
    static const int x = 30;  
}  
void main() {  
    var t = Test();  
    print(Test.x); // 30  
}
```

Data Members

Dart does not have a **private/public/protected** concept similar to C++/Java. However, it does have some visibility limitation at the library level. To hide/restrict access to a data member outside its definition library, use the underscore (`_`) when defining its name.

KEEP IN MIND that this only limits the visibility outside its library. The next code will work as `main` function is defined in the same library (application) as class `Test`.

```
class Test {  
    int _x = 30;  
}  
void main() {  
    var t = Test();  
    print(Test._x); // 30  
}
```

Data Members

Use ?. operator to access a data member and check if the class was defined.

```
class Test {  
    int x = 30;  
}  
void main() {  
    Test? t;  
    t?.x = 10;  
}
```

As a general concept, Dart does not allow to create a null instance of an object (except from using it with “?” operator).

Constructors

Constructors

Constructors are instance specific methods that are called whenever an instance is created. A constructor is created in the following way:

```
class <class_name> {  
    <class_name>([v1, v2, v3, ... vn])); // unnamed ctor  
    <class_name.function_name>([v1, v2, v3, ... vn])); // named ctor  
}
```

As a difference from C++, there can only be only one unnamed constructor (meaning that even if we define multiple unnamed constructors with different parameters, Dart will produce a compile error and not allow this).

Constructors

Data members can be instantiated in the constructor (similar to how C++ allows it by adding `:` and then a list of `data member = value` after this

```
class <class_name> {
    data_type d1,d2,d3... dn;
    <class_name>([v1,v2,v3,... vn]): d1=value1, d2=value2,... dn=valuen, { ... }
}

class Test {
    int x,y;
    Test(): x=10, y = 20 { }
}
void main() {
    var t = Test();
    print("x=${t.x}, y=${t.y}"); // x=10, y=20
}
```

Constructors

It is also possible to initialize a variable directly and as part of the constructor. This will allow one to create some sort of default value that can be used if some constructors do not have an initialization code.

In this cases, the constructor code will take precedence and decide the value of a field.

```
class Test {  
    int x=1;  
    Test(): x=2 {}  
}  
void main() {  
    var t = Test();  
    print(t.x); // 2  
}
```

Constructors

Multiple un-names constructors are not allowed:

```
class Test {
    int x;
    Test(): x = 10
    {
    }
    Test(int value): x = value
    {
    } // Error: 'Test' is already declared in this scope.
}
```

In this case, there is already an un-names constructor (with no parameters) so another one is not allowed.

Constructors

Even if a data member value can be changed in the constructor, its value MUST be instantiated before (after the constructor or at its definition).

```
class Test {  
    int x; // Error: Field 'x' should be initialized because its type  
           // 'int' doesn't allow null.  
    Test() { x = 10; }  
}
```

However, the following code will work:

```
class Test {  
    int x = 2;  
    Test() { x = 10; }  
}
```

Constructors

Since un-names constructors are used to initialize data members, Dart allows some syntax sugar that specifies the data member that is initialized (via `this` keyword).

```
class Test {  
    int x,y;  
    Test(this.x, this.y);  
}  
void main() {  
    var t = Test(10,20);  
    print("x=${t.x}, y=${t.y}"); // x=10, y=20  
}
```



In this case, this constructor is equivalent to:

```
Test(int value_x, int value_y): x=value_x, y = value_y { }
```

Constructors

If a constructor has an empty body {} there is a syntax sugar form that can be used (by adding ; after the constructor definition).

```
class Test {  
    int x,y;  
    Test(): x=10, y = 20 { }  
}
```

is equivalent to

```
class Test {  
    int x,y;  
    Test(): x=10, y = 20;  
}
```

Constructors

When initializing data members, `this` / other data members can not be used as part of the initialization expression. This is different than what C++ allows.

```
class Test {  
    int x,y;  
    Test(int value):  
        y=value,  
        x=this.y*this.y;      // Can't access 'this' in a field  
                               // initializer.  
}
```

In this case, even if “`this.y`” is already instantiated and therefore `x = this.y * this.y` can be computed, Dart does not allow this.

Constructors

Dart classes may have multiple named constructors, as long as they have different names:

```
class Test {  
    int x,y;  
    Test.empty(): x=0,y=0 {}  
    Test.withX(int value): x=value,y=0;  
    Test.withXY(this.x, this.y);  
}  
void main() {  
    var t = Test.empty();  
    print("x=${t.x}, y=${t.y}"); // x=0, y=0  
    t = Test.withX(10);  
    print("x=${t.x}, y=${t.y}"); // x=10, y=0  
    t = Test.withXY(1,2);  
    print("x=${t.x}, y=${t.y}"); // x=1, y=2  
}
```

Constructors

Just like C++, a constructor may proxy the initialization to another constructor (by using `this` keyword):

```
class Test {
    int x,y;
    Test(int value): y=value,x=value; ◀
    Test.empty(): this(0); →
}
void main() {
    var t = Test.empty();
    print("x=${t.x}, y=${t.y}"); // x=0, y=0
}
```

Constructors

Dart also have some compile time protections for cyclic redirections when using constructors.

Constructors

In case of objects that will not be modified during the execution, Dart allows creation of a constant constructor by adding `const` keyword in front of the definition. This will create a compile-time constant object. For this to be possible, data members from that object must be declared as `final`.

```
class Test {  
    final int x,y;  
    const Test.constantObject(): x=0,y=0;  
}  
void main() {  
    var t = Test.constantObject();  
}
```

Constructors

As a general concept, a constructor can not be static. A class however can have static methods (that work like a factory → for example in case of a Singleton pattern). In particular for Dart, there is a special keyword `factory` that allows one to create a named constructor that is static (does not have access to `this`, but returns an instance of the object).

```
class Singleton {  
    static Singleton obj = Singleton(10);  
    int x,y;  
    Singleton(int value): x=value, y=value;  
    factory Singleton.sameObject() { return obj; }  
}  
void main() {  
    var t1 = Singleton.sameObject();  
    var t2 = Singleton.sameObject();  
    print(t1==t2); // true  
}
```

Properties

Properties

In reality, Dart automatically creates two methods (a getter and a setter) for each data member (there are some exceptions → e.g. variable defined as final that do not have the setter method define → hence the error when trying to change such a variable: ***The setter ‘<variable name>’ isn’t defined for the class ‘<class name>’.***

```
class <class_name> {  
    <data_type> get <variable_name> { ... return value; }  
    set(<data_type> value) { ... }  
}
```

Dart provides 2 keywords: **get** and **set** (much like C#) that allow creating a setter and a getter for a data member.

Properties

A simple example:

```
class Grades {  
    int mathGrade, englishGrade;  
    Grades(this.mathGrade, this.englishGrade);  
    int get average { return (mathGrade+englishGrade) >> 1; }  
    set average(int value) { mathGrade = englishGrade = value; }  
}  
void main() {  
    var st = Grades(10,8);  
    print("Average = ${st.average}"); // Average = 9  
    st.average = 10;  
    print("Math=${st.mathGrade}, English=${st.englishGrade}");  
    // Math=10, English=10  
}
```

Properties

For simplicity, `=>` operator can be used

```
class Grades {  
    int mathGrade, englishGrade;  
    Grades(this.mathGrade, this.englishGrade);  
    int get average => (mathGrade+englishGrade) >> 1;  
    set average(int value) => mathGrade = englishGrade = value;  
}  
void main() {  
    var st = Grades(10,8);  
    print("Average = ${st.average}"); // Average = 9  
    st.average = 10;  
    print("Math=${st.mathGrade}, English=${st.englishGrade}");  
    // Math=10, English=10  
}
```

Properties

To create a read-only property, only the `get` method should be defined. The following code will not compile !

Properties

Getters and setters are often created to allow access to a private variable (from outside its library).

```
class Test {  
    int _x = 0; // private (only visible within the Library)  
    set x(int value) => _x=value*2;  
    int get x => _x;  
}  
void main() {  
    var t = Test();  
    t.x = 5;  
    print(t.x); // 10  
}
```

Properties

It is also possible to define only a setter, or to create a different setter and getter that access the same data member in a different way !

```
class Distance {  
    int _m = 0; // the actual distance is stored in meters  
    set km(int value) => _m=value*1000;  
    int get length => _m;  
}  
void main() {  
    var t = Distance();  
    t.km = 5;  
    print(t.length); // 5000  
}
```

Properties

Unary operators (such as `++`) will trigger both the getter and the setter for a specific data member.

```
class Test {
    int _x = 0;
    set x(int value) {
        _x = value;
        print("Setter (x=${value})");
    }
    int get x {
        print("Getter (x=${_x})");
        return _x;
    }
}
void main() {
    var t = Test(); t.x++; // Getter (x=0) , Setter (x=1)
}
```

Methods

Methods

Methods are functions defined within a class that can access data members and act as an interface for producing different actions within a class instance.

```
class <class_name> {  
    <data_type> function_name ([v1, v2, v3, ... vn]) { ... }  
    <data_type> function_name ([v1, v2, v3, ... vn]) => <return value>;  
}
```

Just like a regular function, a method has two forms (a standard one and a simplified one based on the => operator).

Methods

A simple example:

```
class Test {
    int x = 9;
    bool isOdd() {
        return x%2==0;
    }
    bool isEven() => x%2==1;
}
void main() {
    var t = Test();
    print(t.isOdd());      // false
    print(t.isEven());    // true
}
```

Methods

Dart methods DO NOT support overloading (from this point of view, Dart is more similar to Python than C++ or Java). As such, the following code will NOT compile.

```
class Test {  
    int x = 9;  
    int Add(int value) => x+value;  
    int Add(int v1, int v2) => x+v1+v2; // Error: 'Add' is already  
                                         // declared in this scope.  
}
```

And just like in Python, the solution is to create one function with multiple default / named parameters that can be called upon execution.

Methods

A possible solution for the previous code could look like this:

```
class Test {
    int x = 9;
    int Add(int v1, [int? v2]) => v2!=null ? x+v1+v2 : x+v1;
}
void main() {
    var t = Test();
    print(t.Add(5));        // 14
    print(t.Add(6,7));     // 22
}
```

In this case, `v2` is a default variable that can be null if nor specified (because of its “`?`” symbol added after its type: `int`).

Methods

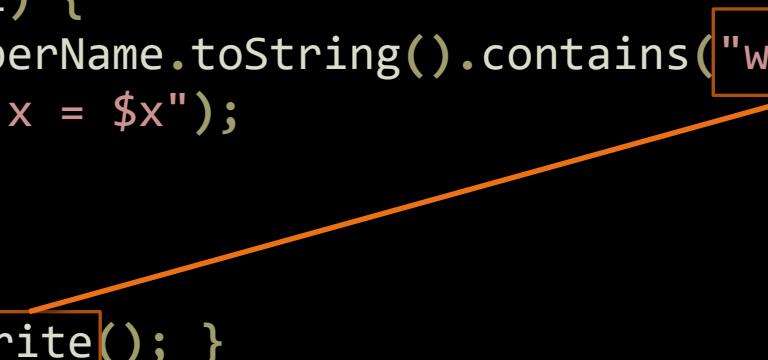
Methods in Dart can be overridden. Just like in Java, it is recommended to add a specific annotation `@override` in front of all overridden methods. Since all Dart classes are derived from `object`, a class can easily override `toString` to provide a string representation of itself.

```
class Test {  
    int x;  
    Test(this.x);  
    @override  
    String toString() => "Test with x = ${x}";  
}  
void main() {  
    var t = Test(10);  
    print(t); // Test with x = 10  
}
```

Methods

Similarly, `noSuchMethod` can be overridden. However, that method will only work for dynamic types, meaning that for every dynamic variable, if a non-defined method or variable is accessed, `noSuchMethod` will be called instead !

```
class Test {  
    int x;  
    Test(this.x);  
    @override  
    void noSuchMethod(Invocation i) {  
        if ((i.isMethod) && (i.memberName.toString().contains("write")))  
            print("Test object with x = $x");  
    }  
}  
void main() {  
    dynamic t = Test(10); t.write(); }
```



Methods

The following code will NOT compile, because “t” is not a dynamic type, but a type Test inferred from its definition.

```
class Test {  
    int x;  
    Test(this.x);  
    @override  
    void noSuchMethod(Invocation i) { }  
}  
void main() {  
    var t = Test(10);  
    t.write();    // Error: The method 'write' isn't defined for the  
                // class 'Test'  
}
```

Methods

Dart also supports static methods (specific to the class and not the instance). To create a static method, add `static` keyword in front of a method definition. A static method can only be called by the class and not by an instance and does not have access to `this` pointer.

```
class Test {  
    int x = 10;  
    static String getMyName() => "Test class";  
}  
void main() {  
    print(Test.getMyName()); // Test class  
}
```

```
var t = Test();  
print(t.getMyName());          // Error: The method 'getMyName' isn't defined  
                             // for the class 'Test'.
```

Operators

Operators

Operators are defined in a similar manner as with C++ language (by using a special keyword **operator** to identify a special method that will treat a specific operation).

```
class <class_name> {  
    <data_type> operator <operator_type> ([v1, v2, v3, ... vn]) { ... }  
    <data_type> operator <operator_type> ([v1, v2, v3, ... vn]) => <return value>  
}
```

Supported operators are:

<	>	<=	>=	==
+	-	*	/	%
~/	^	&		~
>>	>>>	<<	[]	[]=

Operators

Some operators are not needed (for example != is similar to !(==) and as such there is no need for a specific overwrite in this case).

```
class Number {  
    int x;  
    Number(this.x);  
    Number operator+ (Number n) => Number(x+n.x);  
}  
void main() {  
    var n1 = Number(10);  
    var n2 = Number(20);  
    var n3 = n1+n2;  
    print(n3.x); // 30  
}
```

Operators

Just like in the case of method, operator overloading is not possible in Dart. As such, the following code will not compile:

```
class Number {  
    int x;  
    Number(this.x);  
    Number operator+ (Number n) => Number(x+n.x);  
    Number operator+ (int n) => Number(x+n);    // Error: '+' is already  
                                                // declared in this  
                                                // scope.  
}
```

The main difference is that if method overloading can easily be replaced by using multiple methods with different name, the case of operator overloading is a little bit more complicated.

Operators

One way of solving operator overloading problem is to use a dynamic type:

```
class Number {
    int x;
    Number(this.x);
    Number operator+ (dynamic value) {
        if (value is Number) return Number(x+(value as Number).x);
        if (value is int) return Number(x+(value as int));
        return Number(x);
    }
}
void main() {
    print((Number(10)+20).x);          // 30
    print((Number(100)+Number(200)).x); // 300
}
```

Inheritance

Inheritance

Inheritance concepts in Dart are:

- Class extension (similar to Java, using the keyword `extends`)
- Interfaces (a concept between Java and C++, realized using the keyword `implements`)
- Mixin (a way to extend an existing class with new functionality from another class).

Dart does not support multiple inheritance but can achieve similar functionality through interfaces and mixins.

To extend a class use `extends` after the class name and `super` keyword to access base class methods and data members.

```
class <class_name> extends <base_class> implements <interface1, interface2, ...>
{
    . . .
}
```

Inheritance

A simple example of inheritance → class Derived has two variables “x” and “y” and one method `GetX()`.

```
class Base {  
    int x = 10;  
    int GetX() => x*x;  
}  
class Derived extends Base {  
    int y = 20;  
}  
void main() {  
    var d = Derived();  
    print("${d.y}, ${d.x}, ${d.GetX()}"); // 20, 10, 100  
}
```

Inheritance

When deriving from a base class, both methods and data member can be overridden. It is still possible to access the base data member by using `super` keyword from a property/method.

```
class Base {  
    int x = 10;  
}  
class Derived extends Base {  
    int x = 20;  
    int get baseX => super.x;  
}  
void main() {  
    var d = Derived();  
    print("x=${d.x}, base_x=${d.baseX}"); // x=20, base_x=10  
}
```

Inheritance

When overriding a data member, make sure that the same type is used otherwise a compile error will be triggered.

```
class Base {  
    int x = 10;  
}  
class Derived extends Base {  
    double x = 20.5;          // Error: The return type of the method  
                            // 'Derived.x' is 'double', which does not  
                            // match the return type, 'int', of the  
                            // overridden method, 'Base.x'.  
}
```

It is also recommended to use annotation (`@override`) to explain the compiler that the data member was overridden on-purpose and not by mistake.

Inheritance

Method overridden works in a similar manner

```
class Base {  
    int sum(int x, int y) => x+y;  
}  
class Derived extends Base {  
    @override  
    int sum(int x, int y) => x*y;  
}  
void main() {  
    var d = Derived();  
    print(d.sum(3,4)); // 12  
}
```

Inheritance

And just like in the previous case, the method that is being overridden must have the same name, parameters and return type as the one from the base class.

Inheritance

Methods in Dart are virtual by nature (overriding one will change the behavior even if casting to the base class).

```
class Base {  
    int op(int x, int y) => x+y;  
}  
class Derived extends Base {  
    @override  
    int op(int x, int y) => x*y;  
}  
int op(Base b, int x, int y) => b.op(x,y);  
void main() {  
    var d = Derived();  
    print(op(d,3,4)); // 12  
}
```

Inheritance

Keep in mind that `final` keyword in Dart does not have the same meaning as it has in Java (from this regard Dart is more close to C++ than to Java).

```
final class Base {  
    int op(int x, int y) => x+y;  
}  
class Derived extends Base {  
    @override  
    int op(int x, int y) => x*y;  
}
```

The following code will not compile (`final` is considered an invalid keyword to be used with class specifier).

Inheritance

Dart allows creation of **abstract** classes (NOT interfaces) that can be used to enforce overriding some methods. An abstract method can only be defined within an abstract class, but it is different from a regular method as it has **;** operator at the end of its method definition.

```
abstract class Form {  
    String getName() ;  
}  
class Circle extends Form {  
    // The non-abstract class 'Circle' is missing implementations for  
    // these members: - Form.getName  
}
```

In this case, class Circle is not complete because it does not implement the abstract method **getName** from the class Form.

To create an abstract class, prefix its definition with **abstract** keyword.

Inheritance

At the same time, an abstract class can have non-abstract methods / data-members or properties.

Inheritance

Abstract classes and methods are not necessary (but are very useful for code clarity). The previous example can be re-written without abstract classes in the following way.

Inheritance

Abstract classes can have abstract properties.

Any data-member defined in the abstract class will be inherit in the derived class as well.

```
abstract class Form {  
    String get name; // abstract property  
    int x = 10;  
}  
class Circle extends Form {  
    @override  
    String get name => "Circle";  
}  
void main() {  
    var c = Circle();  
    print("${c.name}, ${c.x}"); // Circle, 10  
}
```

Inheritance

A class can also implement an interface (methods and properties) from another class (BUT no data members). The previous code will not compile if Circle implements Form !

```
abstract class Form {  
    String get name; // abstract property  
    int x = 10;  
}  
class Circle implements Form {  
    @override  
    String get name => "Circle";  
}  
void main() {  
    var c = Circle();  
    print("${c.x}"); // Error: The non-abstract class 'Circle' is  
                    // missing implementations for these members: Form.x  
}
```

Inheritance

A class can implement multiple interfaces:

```
abstract class Form { int get area; } •←
abstract class Name { String get name; } •←

class Circle implements Form, Name
{
    @override
    String get name => "Circle";
    @override
    int get area => 10;
}

void main() {
    var c = Circle();
    print("${c.name}, ${c.area}"); // Circle, 10
}
```

Inheritance

Any class (abstract or not) can be an interface for another class.

```
class Form { int get area => 0; }
class Name { String get name => "Form"; }

class Circle implements Form, Name
{
    @override
    String get name => "Circle";
    @override
    int get area => 10;
}
void main() {
    var c = Circle();
    print("${c.name}, ${c.area}"); // Circle, 10
}
```

Inheritance

When using multiple implements make sure that you don't have similar functions (by name and different parameters) that you are implementing.

```
abstract class Interface1 { int foo(int x); }
abstract class Interface2 { int foo(int x, int y); }
class Test implements Interface1, Interface2 {
    @override int foo(int x) => 2;
    @override int foo(int x, int y) => 2; // Error: 'foo' is already
                                         // declared in this scope.
}
void main() {
    var t = Test();
    print(t.foo); // Error: Can't use 'foo' because it is declared more
                  // than once.
}
```

Inheritance

However, if two abstract methods are identical (same name, same return value, same parameters), implementing only one will suffice and the code will compile. The same logic applies for properties (different interfaces with the same property (name and type) can co-exist).

```
abstract class Interface1 { int foo(int x); }
abstract class Interface2 { int foo(int x); }
class Test implements Interface1, Interface2 {
    @override
    int foo(int x) => 2;
}
void main() {
    var t = Test();
    print(t.foo(10)); // 2
}
```

Inheritance

Sometimes it is required that the derived class implements a function slightly different (but with the same logic → e.g. a parameter that is derived from the one used in the base definition).

```
class Form { }
class Rectangle extends Form { }
abstract class Calculator { int Compute(Form f); }
class AreaCalculator implements Calculator {
    @override int Compute(Rectangle r) => 0;
    // Error: The parameter 'r' of the method 'AreaCalculator.Compute'
    // has type 'Rectangle', which does not match the corresponding type,
    // 'Form', in the overridden method, 'Calculator.Compute'.
}
void main() {
    var ac = AreaCalculator();
    print(ac.Compute(Rectangle()));
}
```

Inheritance

The solution in this case is to use the keyword **covariant** and allow other parameters to be used as part of the virtual methods, as long as they are derived from the base class (in this case, it has to be a parameter derived from Form class → e.g. Rectangle).

```
class Form { }
class Rectangle extends Form { }
abstract class Calculator { int Compute(covariant Form f); }
class AreaCalculator implements Calculator {
    @override
    int Compute(Rectangle r) => 0;
}
void main() {
    var ac = AreaCalculator();
    print(ac.Compute(Rectangle()));
}
```

Inheritance

Dynamic polymorphism can easily be achieved using implements keyword

Inheritance

The same example will work in a similar way if we use extends and a non-abstract class as a base class, or if we combine those two cases (abstract class with `extends` keyword or regular class with `implements` keyword).

Inheritance

`extends` and `implements` keywords can coexist when defining a derived class:

```
class Form {  
    int x = 100;  
}  
abstract class Name {  
    String get name;  
}  
class Circle extends Form implements Name {  
    @override  
    String get name => "Circle";  
}  
void main() {  
    var c = Circle();  
    print("${c.name}, ${c.x}"); // Circle, 100  
}
```

Inheritance

When using **implements** keyword, all methods and properties from the inherit class MUST be implemented (even if they are already implemented in the base class). This is important as it differentiate an interface from a mixin.

```
class Interface {  
    int sum(int x, int y) => x+y;  
}  
class my_int implements Interface {  
    // Error: The non-abstract class 'my_int' is missing implementations  
    // for these members: - Interface.sum  
}  
void main() {  
    var i = my_int();  
    print(i.sum(10,20));  
}
```

Inheritance

To overcome previous limitation, Dart has introduced **mixins** (a class that has some methods defined that will be available as they are defined in the base class). To create a mixin use the **mixin** keyword and with keyword **when** deriving the new class.

```
mixin SumImplementation {
    int sum(int x, int y) => x+y;
}
class my_int with SumImplementation {
}
void main() {
    var i = my_int();
    print(i.sum(10,20)); // 30
}
```

Inheritance

Using the keyword `mixin` is not mandatory. A class can be derived using `with` keyword from another class as well (not necessary a mixin). However, if `mixin` keyword is used, that class can not be instantiated.

```
class SumImplementation {
    int sum(int x, int y) => x+y;
}
class my_int with SumImplementation {
}
void main() {
    var i = my_int();
    print(i.sum(10,20)); // 30
}
```

Inheritance

While not necessary, methods from a mixin can be overridden and they work just like every other virtual method. However, even if possible, this is not the intended usage for a mixin.

```
mixin SumImplementation {
    int sum(int x, int y) => x+y;
}
class my_int with SumImplementation {
    int sum(int x, int y) => x*y;
}
void main() {
    var i = my_int();
    print(i.sum(10,20)); // 200
    print((i as SumImplementation).sum(10,20)); // 200
}
```

Inheritance

Multiple mixins can be used at the same time. They need to respect the same rules as the ones used when using implements (e.g. no similar names with different parameters or return type).

```
mixin SumImplementation {
    int sum(int x, int y) => x+y;
}
mixin MulImplementation {
    int mul(int x, int y) => x*y;
}
class my_int with SumImplementation, MulImplementation { }
void main() {
    var i = my_int();
    print(i.sum(10,20)); // 30
    print(i.mul(10,20)); // 200
}
```

Inheritance

Another difference between using `with` and `implements` keywords is that data members are going to be present in the derived type (just like in the case of `extends`).

```
mixin SumImplementation {
    int sum(int x, int y) => x+y;
    int v = 10;
}
class my_int with SumImplementation {
}
void main() {
    var i = my_int();
    print(i.sum(10,20)); // 30
    print(i.v);          // 10
}
```

Inheritance

It is possible to derive from multiple mixins with the same data member. In this case, the last derivation will set the value for that variable in the derived class.

```
mixin M1 { int v = 10; }
mixin M2 { int v = 20; }
class my_int with M1, M2 {
}
void main() {
    var i = my_int();
    print(i.v); // 20
}
```

```
mixin M1 { int v = 10; }
mixin M2 { int v = 20; }
class my_int with M2, M1 {
}
void main() {
    var i = my_int();
    print(i.v); // 10
}
```

Inheritance

It is not possible to use two mixins with the same data member but of different types. If this is the case, a compile error will be triggered.

```
mixin M1 { int v = 10; }
mixin M2 { double v = 20.0; }
class my_int with M1, M2 {
    // Error: Applying the mixin 'M2' to 'Object with M1' introduces an
    // erroneous override of 'v'.
}
void main() {
    var i = my_int();
    print(i.v); // 20
}
```

Inheritance

A mixin can be restricted to be applied to a specific type or its descendants by using the keyword `on` upon definition. In the next example, only a class derived from `Number` can use the `Sum` mixin. As such, `my_string` will produce a compile error.

```
class Number { }
mixin Sum on Number { int sum(x,y) => x+y; }
class my_int extends Number with Sum { }
class my_string with Sum {
    // Error: 'Object' doesn't implement 'Number' so it can't be used with
    // 'Sum'.
}
void main() {
    var i = my_int();
    print(i.sum(1,2)); // 3
}
```

Inheritance

The following table presents an overview on what gets inherit for `extends` , `implements` and `with` keywords.

	extends	implements	with
Data members	Yes, will be inherit in the derived class	Will NOT be inherit in the derived class	Yes, will be inherit in the derived class
Methods	Can be overridden, but it is not required	MUST be overriden	Can be overridden, but it is not required
Properties	Can be overridden, but it is not required	MUST be overriden	Can be overridden, but it is not required
Abstract methods		Virtual, MUST be overridden	
Abstract properties		Virtual, MUST be overridden	
Multiple inheritance	NO	Yes	Yes

Inheritance

What to use and when:

- **extends** → whenever you want to extend an existing class with new functionalities (but also keep and reuse the existing ones).
- **implements** → whenever there is a common interface between multiple classes (a fix set of properties that all objects that have a common ancestor have).
- **with** → whenever, there is some already implemented code (methods/properties) that need to be reused, but is not a general characteristic for all object with a specific ancestor

Q & A



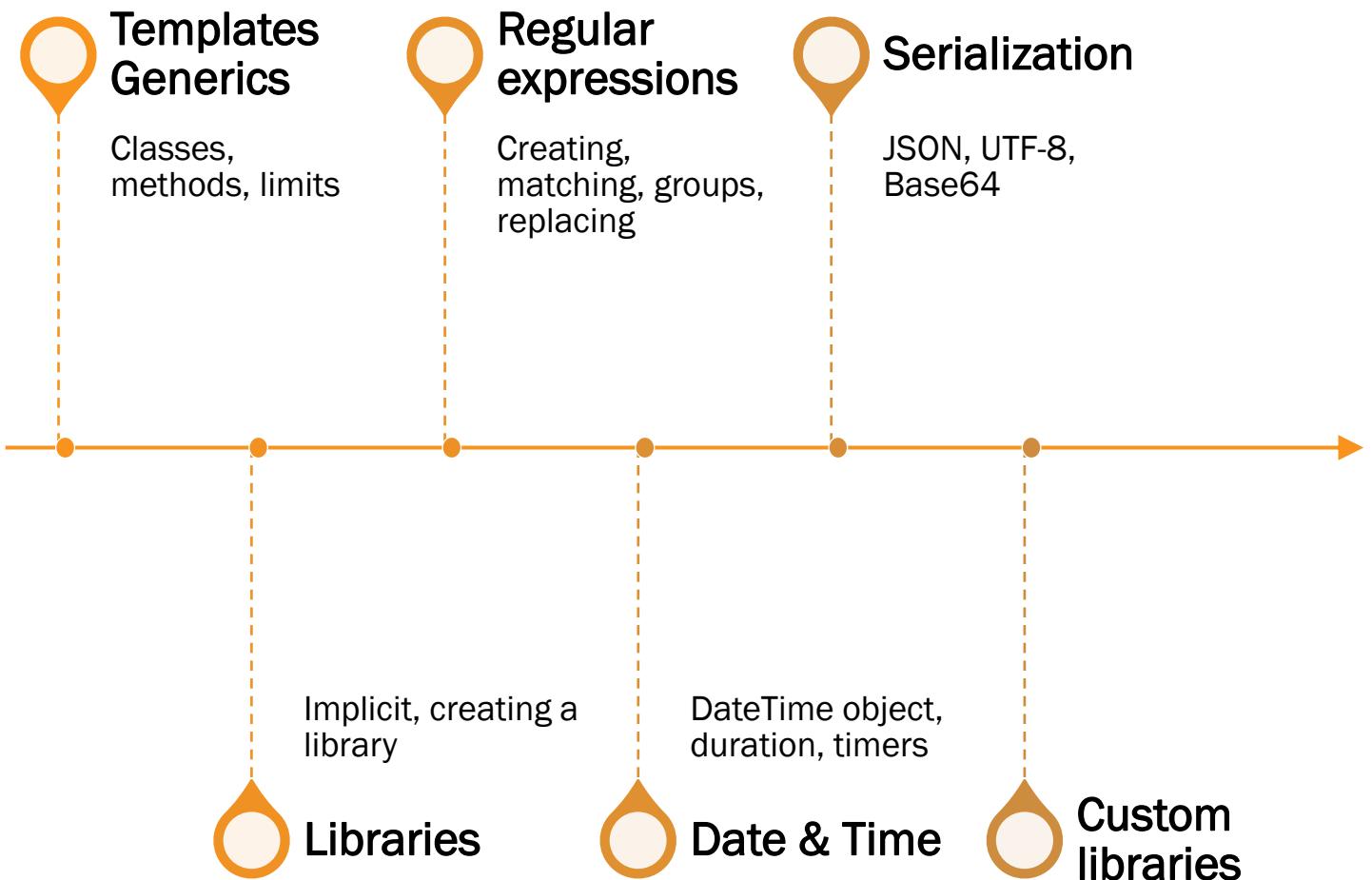


DART Language

COURSE 4 (REV 3)

GAVRILUT DRAGOS

Agenda



Templates / Generics

Templates

Templates (or generics!?) – because there are some differences on how a template/generic is compiled in different languages) are used to avoid code duplication. Just like in C++ templates can be used for both methods and classes.

The general form a template is created:

```
class class_name <T1, [T2, ..., Tn]>
{
    // described the class and use T1..n within it
}
```

The concept is a little bit more generic as T_{1..n} can be a type or a type that express an inheritance relationship:

- Example: T = type
- Example: T = type **extends** base_type

Templates

A very simple example:

```
class MyTemplate<T,G> {
    T obj_1;
    G obj_2;
    MyTemplate.init(T obj1, G obj2): obj_1 = obj1, obj_2 = obj2;
    void Print() {
        print("MyClass(obj_1 = ${obj_1.toString()},
                  obj_2 = ${obj_2.toString()}"));
    }
}
void main() {
    var m = MyTemplate<int,String>.init(10,"Text");
    m.Print(); // MyClass(obj_1 = 10, obj_2 = Text)
}
```

Templates

However, if in C++, templates are evaluated at runtime based on parameters types it, Dart language validates the template without looking into the type of parameters.

```
class MyTemplate<T> {
    T Sum(T obj1, T obj2) {
        return (obj1+obj2) as T;      // Error: The operator '+' isn't
                                      // defined for the class
                                      // 'Object?'.
    }
}
```

The previous code will not compile as type not all possible `T` have a plus operator (`operator+`) defined. In reality `T` is seen as something derived from `Object`, and not all `Object` objects have `operator+` defined. This means that even if we write something like “`MyTemplate<int>`” the code will not compile.

Templates

The solution for the previous case is to use extended keyword to limit the type of a parametrized type.

```
class MyTemplate<T extends num> {
    T Sum(T obj1, T obj2) {
        return (obj1+obj2) as T;
    }
}
void main() {
    var m = MyTemplate<int>();
    print(m.Sum(10,20)); // 30
}
```

Templates

This technique also works as a way to limit the types that can be used in a template / generic. In the following example, as `T` must be a sub-class of `num`, one can not use `String` create a template / generic based on `MyTemplate`.

```
class MyTemplate<T extends num> {
    T Sum(T obj1, T obj2) {
        return (obj1+obj2) as T;
    }
}
void main() {
    var m = MyTemplate<String>();
    print(m.Sum(10,20)); // Error: The argument type 'int' can't be
                        // assigned to the parameter type 'String'.
}
```

Templates

A more complex example:

```
abstract class PrintInterface { void Print(); }
class MyTemplate<T extends PrintInterface> {
    void Print(T obj) => obj?.Print();
}
class MyInt extends PrintInterface {
    int i;
    MyInt(this.i);
    @override
    void Print() => print(i);
}
void main() {
    var m = MyTemplate<MyInt>();
    m.Print(MyInt(10)); // 10
}
```

Templates

However, extends keyword must be used. The same example will not compile if T just implements an interface. In this case, a compiler error will be raised. The same logic applies for mixins as well.

```
abstract class PrintInterface { void Print(); }
class MyTemplate<T implements PrintInterface> {
    void Print(T obj) => obj?.Print();
}
class MyInt extends PrintInterface {
    int i;
    MyInt(this.i);
    @override
    void Print() => print(i);
}
```

Templates

It is also possible to create generic/templatized method for a regular class.

```
class MyString {
    String data = "";
    void From<T>(T obj) => data = obj.toString();
}
void main() {
    var m = MyString();
    m.From<int>(10);
    print(m.data); // 10
    m.From<double>(1.23);
    print(m.data); // 1.23
}
```

Libraries

Libraries

Like any language, DART has a way to add additional functionality via external modules (called libraries). Dart comes with a set of predefined libraries, but custom libraries can be created as well. To use functionality from another library, use the keyword `import`.

```
import "uri";  
import "uri" as <alias>;
```

It is also possible to import only a part of library (just some components)

```
import "uri" show <component>;
```

or to import an entire library but exclude some components.

```
import "uri" hide <component>;
```

Libraries

The “uri” used by the import system has the following format:

1. “**dart:<name>**” → for regular dart libraries
2. “**package:<path>**” → for dart packages
3. “**<path>**” → for a local files

The most common dart default packages:

Library	Functionality
dart:math	Mathematical functions, constants, random number generator
dart:collection	Collections (queues, stacks, trees, etc)
dart:io	I/O support for non-web apps (file, sockets, http, etc)
dart:convert	Data representation (e.g. JSON)
dart:ffi & dart:typed_data	Foreign function integration (e.g. with C/C++ code) and fixed-sized data (e.g. 8-bit integers)
... and other	

Libraries – math module

Math module provides access to a lot of mathematical functions (sin, cos, tan, sqrt, etc), mathematical constants (pi, e, etc) and some geometrical structures (Point, Rectangle, etc).

```
import 'dart:math' as mt;

void main() {
    print(mt.pi);                      // 3.141592653589793
    print(mt.e);                        // 2.718281828459045
    print(mt.sin(30 * mt.pi / 180.0)); // 0.4999999999999999
    print(mt.Random().nextInt(100));   // 12
    print(mt.sqrt(25));                // 5
    print(mt.min(10, 20));              // 10
    print(mt.Point(10, 20));            // Point(10, 20)
}
```

Libraries – I/O

Dart io module has a File object that can be used for file operators.

Constructors:

<code>File (String path)</code>	Creates a new File object
<code>File.fromRawPath(UInt8List path)</code>	Creates a new File object from a raw path
<code>File.fromUri(UInt8List path)</code>	Creates a new File object from an URI

File I/O operation can be synchronous and asynchronous. While both of them are supported by Dart, we will only discuss about synchronous operations in this course.

Libraries – I/O

A very simple example of creating a file and writing some text into it.

```
import "dart:io";
void main() {
    File("a.txt").openWrite()..write("A new file was created")..close();
}
```

Or written in a more familiar way:

```
import "dart:io";
void main() {
    var f = File("a.txt").openWrite();
    f.write("A new file was created");
    f.close();
}
```

Libraries – I/O

A File also has some properties (like its path, directory, URI, etc).

```
import "dart:io";

void main() {
    var f = File("E:\\Lucru\\Dart\\a.txt");
    print(f.path);          // E:\Lucru\Dart\a.txt
    print(f.parent);        // Directory: 'E:\Lucru\Dart'
    print(f.uri);           // file:///E:/Lucru/Dart/a.txt
}
```

For programs that use File object it is best to compile them natively so that they can work with the existing OS functions.

Libraries – I/O

A File object actually offers a set of quick functions that can provide different file operations:

For example, the following code can be used to read the entire content of a File and display-it as a list of lines (`List<String>`)

```
import "dart:io";
void main() {
    for(var line in File("E:\\Lucru\\Dart\\a.txt").readAsLinesSync())
        print(line);
}
```

Similarly, to write a content to a file, the following can be used:

```
File("a.txt").writeAsStringSync("Dart programming");
File("b.bin").writeAsBytesSync(<int>[1,2,3,4,5]);
```

Libraries – I/O

A list of all methods supported by File object can be found here:

<https://api.dart.dev/stable/<version>/dart-io/File-class.html>

To simplify:

- Read related methods: *readAsBytes*, *readAsBytesSync*, *readAsLines*, *readAsLinesSync*, *readAsString*, *readAsStringSync*
- Write related methods: *writeAsBytes*, *writeAsBytesSync*, *writeAsString*, *writeAsStringSync*
- OS file related methods: *rename*, *renameSync*, *exists*, *existsSync*, *delete*, *deleteSync*, *copy*, *copySync*
- File information: *lastAccesed*, *lastAccesedSync*, *lastModified*, *lastModifiedSync*, *length*, *lengthSync*

For each of these methods there are two forms (**Sync** or not (meaning asynchronously)). We will talk more about these forms when we discuss asynchronously support in Dart.

Libraries – dart:core

A series of libraries are already available in the dart:core module:

- Different types: String, num, Map, etc
- Date time objects
- Regular expression support
- URI support
- Runes
- Exception and Error support
- Async support (Future template)

Regular expressions

Libraries – Regular expressions

Dart has an object `RegExp` that is used to express a regular expression.

Constructor:

```
RegExp (String regexp, {  
    bool multiLine = false,  
    bool caseSensitive = true,  
    bool unicode = false,  
    bool dotAll = false}  
)
```

Creates a new regular expression object.
There are several named parameters that can control
string format:

- Multi-line support
- If it is case sensitive
- Unicode format
- ...

Properties:

<code>bool RegExp.isCaseSensitive</code>	If the regular expression is case sensitive
<code>bool RegExp.isDotAll</code>	If ‘.’ (dot) should represent all characters
<code>bool RegExp.isMultiLine</code>	If the pattern represent a multi-line object
<code>bool RegExp.isUnicode</code>	If the pattern is based on unicode

Libraries – Regular expressions

Dart has an object `RegExp` that is used to express a regular expression.

Methods:

<code>Iterable<RegExpMatch> RegExp.allMatches(String input, [int start = 0])</code>
<code>RegExpMatch? RegExp.firstMatch(String input)</code>
<code>bool RegExp.hasMatch(String input)</code>
<code>Match? RegExp.matchAsPrefix(String input, [int start = 0])</code>
<code>String RegExp.stringMatch(String input)</code>

All of these methods can be used to test regular expression matchings.

OBS: usually, the `input` string is a raw string (to avoid de-duplication of '\\' characters)

Libraries – Regular expressions

A simple example:

```
void main() {
    var r = RegExp(r"[0-9]+");
    print(r.hasMatch("Hello 1234 world"));      // true
    print(r.stringMatch("Hello 1234 world"));    // 1234
    print(r.matchAsPrefix("Hello 1234 world")); // null

    var res = r.matchAsPrefix("Hello 1234 world",6);
    if (res != null)
    {
        print("${res.start}, ${res.end}"); // 6, 10
    }
}
```

Libraries – Regular expressions

If we want to find all matches we can use the `.allMatches` method. The next example finds all words from a sentence.

```
void main() {  
    var r = RegExp(r"\w+");  
    var s = "Hello world in Dart language";  
    for (var i in r.allMatches(s)) {  
        print("${i.start},${i.end} => ${i.group(0)}");  
    }  
}
```

Output

0,5 => Hello
6,11 => world
12,14 => in
15,19 => Dart
20,28 => language

Libraries – Regular expressions

Similarly, some method from String object allow using regular expressions. One such method is split that can be used with regular expressions.

```
void main() {  
    var r = RegExp(r"\s+");  
    var s = "Hello world      in      Dart      language";  
    for (var i in s.split(r)) {  
        print(i);  
    }  
}
```

Output
Hello
world
in
Dart
language

As an alternative, one can write things like this:

```
for (var i in "Hello world in      Dart language" .split(RegExp(r"\s+"))) {  
    print(i);  
}
```

Libraries – Regular expressions

Beside “**Split**” other methods from class String that use Regular expression as parameters are:

Libraries – Regular expressions

A “Pattern” class is a class that can represent a form of string data use for various operation (comparation, finding, etc).

A pattern instance can be both:

1. A regular string
2. A regular expression

as it implements both RegExp and String interfaces.

Libraries – Regular expressions

This example shows how replace function works with regular expression.

The second case actually uses the result of the regular expression (`m[0]` is actually the string that matched).

```
void main() {
    var r = RegExp(r"[0-9]+");
    var s = "100 kilos of apples cost 25 dolars";
    print(s.replaceAll(r,"<NUM>")); // <NUM> kilos of apples cost <NUM>
                                    // dolars
    r = RegExp(r"\w+");
    print(s.replaceAllMapped(r,(m)=>m[0]?m[0].toUpperCase(): ""));
    // 100 KILOS OF APPLES COST 25 DOLARS
}
```

Libraries – Regular expressions

Groups are also available and one case use the `[]` operator to access each representative of a group. There is also a `.groupCount` property that tells you how many groups are there (keep in mind that the total number of groups is larger by 1 unit as `[0]` will always be the entire match.

```
void main() {  
    var r = RegExp(r"([0-9]{1,3})\.( [0-9]{1,3})\.( [0-9]{1,3})\.( [0-9]{1,3})");  
    var s = "The server IP is 127.5.9.255";  
    var m = r.firstMatch(s);  
    if (m != null) {  
        print("Groups = ${m.groupCount}");  
        for (var i =0;i<=m.groupCount;i++) {  
            print(m[i]);  
        }  
    }  
}
```

Output
Groups = 4
127.5.9.255
127
5
9
255

Date & Time

Libraries – Date & Time

For date/time operations Dart has two objects: `DateTime` and `Duration`

Constructor:

```
DateTime (int year, [int month = 1, int day = 1, int hour = 0,  
                     int minute = 0, int second = 0, int millisecond = 0,  
                     int microsecond = 0])
```

```
DateTime.fromMicrosecondsSinceEpoch(int microsecondsSinceEpoch,  
                                      {bool isUtc = false})
```

```
DateTime.fromMillisecondsSinceEpoch(int millisecondsSinceEpoch,  
                                      {bool isUtc = false})
```

```
DateTime.now()
```

```
DateTime.utc(int year, [int month = 1, int day = 1, int hour = 0,  
                      int minute = 0, int second = 0, int millisecond = 0,  
                      int microsecond = 0])
```

Libraries – Date & Time

A simple example:

```
void main() {  
    var d = DateTime.now();  
    print(d); // yyyy-mm-dd hh:mm:ss.msec  
    d = DateTime(1900,1,1,12,30,45);  
    print(d); // 1900-01-01 12:30:45.000  
    d = DateTime.fromMicrosecondsSinceEpoch(10000);  
    print(d); // 1970-01-01 02:00:00.010  
    print("Year = ${d.year}, Day=${d.day}"); // Year = 1970, Day=1  
    print("Day of week = ${d.weekday}"); // Day of week = 4  
}
```

In term of properties a DateTime object has: year, month, day, weekday, hour, minute, second, millisecond, millisecondsSinceEpoch, microsecond and microsecondsSinceEpoch.

Libraries – Date & Time

For date/time operations Dart has two objects: `DateTime` and `Duration`

Methods:

<code>DateTime</code>	<code>DateTime.add (Duration d)</code>
<code>DateTime</code>	<code>DateTime.subtract (Duration d)</code>
<code>int</code>	<code>DateTime.compareTo (DateTime d)</code>
<code>bool</code>	<code>DateTime.isAfter (DateTime d)</code>
<code>bool</code>	<code>DateTime.isBefore (DateTime d)</code>
<code>bool</code>	<code>DateTime.isAtSameMomentAs (DateTime d)</code>
<code>DateTime</code>	<code>DateTime.toLocal ()</code>
<code>DateTime</code>	<code>DateTime.toUtc ()</code>

Libraries – Date & Time

For **Duration** object is used to compute differences between different moments in time:

Constructor:

```
const Duration ({int days = 0, int hours = 0, int minutes = 0,  
                int seconds = 0, int milliseconds = 0,  
                int microseconds = 0})
```

This is a const constructor → meaning that the resulted object will be **const** as well. A duration object supports various operators (+, - < <= > >= == !=) and has the following properties:

int Duration.inDays	int Duration.inSeconds
int Duration.inHours	int Duration.inMilliseconds
int Duration.inMinutes	int Duration.inMicroseconds

Libraries – Date & Time

A simple example:

```
void main() {  
    var d1 = DateTime(2000,1,1,12,30);  
    var d2 = d1.add(Duration(days:3,minutes:10));  
    print(d1); // 2000-01-01 12:30:00.000  
    print(d2); // 2000-01-04 12:40:00.000  
    print(d1.compareTo(d2)); // -1  
    print(d2.difference(d1).inMinutes); // 4330  
}
```

Libraries – Date & Time

`DateTime` object also has some static methods that can be used to parse a string and obtain the date time that corresponds to that textual format:

Static methods:

<code>DateTime</code>	<code>DateTime.parse (String txt)</code>
<code>DateTime?</code>	<code>DateTime.tryParse (String txt)</code>

The following strings are accepted forms for the parser:

<code>yyyy-MM-dd</code>	<code>yyyyMMdd hh:mm:ss</code>
<code>yyyy-MM-dd hh:mm:ss</code>	<code>yyyyMMddThhmmss</code>
<code>yyyy-MM-dd hh:mm:ss.ms</code>	<code>yyyyMMdd</code>
<code>yyyy-MM-dd hh:mm:ss,ms</code>	<code>yyyy-MM-ddThhZ</code>

Libraries – Date & Time

A simple example:

```
void main() {
    print(DateTime.parse("2000-05-10"));           // 2000-05-10 00:00:00.000
    print(DateTime.parse("2000-05-10 12:30"));     // 2000-05-10 12:30:00.000
    print(DateTime.parse("20000510 11:20"));       // 2000-05-10 11:20:00.000
    print(DateTime.parse("20000510T112345"));     // 2000-05-10 11:23:45.000
}
```

Make sure that you respect the format. For example, the following example will throw an exception:

```
void main() {
    print(DateTime.parse("2000/05/10"));           // runtime EXCEPTION
}
```

Libraries – Date & Time

To measure the time that passes while doing different operations, Dart has another class called Stopwatch.

Methods:

void	Startwatch. start ()
void	Startwatch. stop ()
void	Startwatch. reset ()

Properties:

Duration	Startwatch. elapsed
int	Startwatch. elapsedMicroseconds
int	Startwatch. elapsedMilliseconds
bool	Startwatch. isRunning

Libraries – Date & Time

A simple example:

```
void main() {
    var timer = Stopwatch();
    timer.start();
    var counter = 0, mod = 2;
    for (var i = 0;i<100000000;i++) {
        if ((i % mod)==0) {
            counter++;mod++;
            if (mod>10) mod = 2;
        }
    }
    timer.stop();
    print("Algorithm time: ${timer.elapsed.inSeconds} seconds,
          found $counter values");
}
```

Data serialization

Data serialization

Dart has a library: `dart:convert` that has multiple classes designed to allow conversion between different data types:

- JSON
- Base64
- Ascii
- UTF8
- Latin1
- HTML escape

These objects allow quick processing (via strings) for different types of text based formats.

JSON

Two classes (`JsonEncoder` and `JsonDecoder`) and binding class (`JsonCodec`).

Constructors:

```
JsonEncoder ([Object? toEncodable(dynamic obj)?])
```

```
JsonEncoder.withIndent (String? indent, [Object? toEncodable(dynamic obj)?])
```

```
JsonDecoder ([Object? reviver(Object? key, Object? value)?])
```

Methods:

```
String JsonEncoder.convert (Object? obj)
```

```
dynamic JsonDecoder.convert (String input)
```

JSON

A simple example (serialize):

```
import "dart:convert";

void main() {
    var json = JsonEncoder();
    var x = [1,2,3,4];
    var s = json.convert(x);
    print(s.runtimeType);           // String
    print(s);                      // [1,2,3,4]
}
```

JSON

To pretty format the out, use the `.withIndent` named constructor:

```
import "dart:convert";

void main() {
    var json = JsonEncoder.withIndent(" ");
    var x = [1,2,3,4];
    var s = json.convert(x);
    print(s);    // [
                  //   1,
                  //   2,
                  //   3,
                  //   4
                  // ]
}
```

JSON

The same logic works for maps as well. In case of maps, the keys **MUST** be of type string, otherwise a runtime error will be thrown

```
import "dart:convert";

void main() {
    var json = JsonEncoder.withIndent(" ");
    var x = {"Marilena":10, "Alin":9, "Dragos": 8};
    var s = json.convert(x);
    print(s);    // {
                  //   "Marilena": 10,
                  //   "Alin": 9,
                  //   "Dragos": 8
                  // }
}
```

JSON

The same logic works for maps as well. In case of maps, the keys **MUST** be of type string, otherwise a runtime error will be thrown.

In this example we have changed the map to use integer keys. The code will compile but will throw a runtime error.

```
import "dart:convert";

void main() {
    var json = JsonEncoder.withIndent(" ");
    var x = {10:"Marilena", 9:"Alin", 8:"Dragos"};
    var s = json.convert(x);      // Uncaught Error: Converting object to
                                // an encodable object failed
    print(s);
}
```

JSON

The following code decodes a JSON from a string. As a general idea, use `Map<String,dynamic>` whenever these objects are decoded as the values can be anything!

```
import "dart:convert";
void main() {
    var json = JsonDecoder();
    var s = """{
        "Name": "Dragos",
        "Grades": {
            "Dart": 10,
            "Math": 9
        }
    }""";
    Map<String,dynamic> res = json.convert(s);
    print("Name=${res['Name']}, Math=${res['Grades']['Math']}");
}
```

JSON

Serializing an object to JSON needs some adjustments. Simply sending an object of some sort to a JSON encoder will not work.

JSON

First solution for this problem is to provide a `toEncodable` function when creating the `JsonEncoder` object.

```
import "dart:convert";
class Test { int x=10,y=20; }
Object? EncodeSomething(dynamic obj) {
    if (obj is Test) {
        var t = obj as Test;
        return {"Test":"x=${t.x},y=${t.y}"};
    }
    return null;
}
void main() {
    var s = JsonEncoder(EncodeSomething).convert(Test());
    print(s); // {"Test":"x=10,y=20"}
}
```

JSON

The second solution is to add a `.toJson` method in the object that you want to serialized as a JSON.

```
import "dart:convert";

class Test {
    int x=10,y=20;

    String toJson() => "x=${x}, y=${y}";
}

void main() {
    var s = JsonEncoder().convert({"MyKey":Test()});
    print(s); // {"MyKey": "x=10, y=20"}
}
```

JSON

To decode an object from a Json, one may move the decoding logic into the class itself by creating a constructor that receives the string format from the Json object.

```
import "dart:convert";
class Test {
    int x = 0, y = 0;
    Test.fromJson(String s) {
        var parser = RegExp(r"x=(\d+),\s+y=(\d+)").firstMatch(s);
        x = int.parse(parser?.group(1) ?? "0");
        y = int.parse(parser?.group(2) ?? "0");
    }
}
void main() {
    var m = JsonDecoder().convert(''{"MyKey": "x=10, y=20"}'');
    var t = Test.fromJson(m["MyKey"]); print("${t.x}, ${t.y}"); // 10,20
}
```

JSON

Another solution is to use a reviver function and to pass it as a parameter to the constructor of JsonDecoder class.

```
import "dart:convert";
class Test { int x = 0,y = 0;
    Test.fromJson(String s) { /* same method from the previous slide */ }
}
Object? MyConvertor(Object? key, Object? value) {
    if ((key!=null) && (key.toString() == "MyKey")) // a Test object
        return Test.fromJson(value!=null?value.toString(): "");
    return value;
}
void main() {
    var m = JsonDecoder(MyConvertor).convert("""{"MyKey":"x=10, y=20"}""");
    print((m["MyKey"] as Test).x); // 10
}
```

UTF-8

Two classes (`Utf8Encoder` and `Utf8Decoder`) and binding class (`Utf8Codec`).

Constructors:

```
Utf8Encoder ()
```

```
Utf8Decoder ({bool allowMalformed = false})
```

Methods:

```
UInt8List Utf8Encoder.convert (String input, [int start = 0, int? end])
```

```
String     Utf8Decoder.convert (List<int> input, [int start = 0, int? end])
```

There is also a constant object define (`utf8` of type `Utf8Codec`) that can be used so that there is no need to create an encoder / decoder. A codec has two method: `.encode` and `.decode` that pretty much call `encoder.convert(...)` and `decoder.convert(...)`.

UTF-8

A simple example:

```
import "dart:convert";

void main() {
    var u8 = Utf8Encoder().convert("românește");
    print(u8); // [114, 111, 109, 195, 162, 110, 101, 200, 153, 116, 101]
    var txt = Utf8Decoder().convert(u8);
    print(txt); // românește
    var diacritice = utf8.encode("ăiștĂÎŞT");
    print(diacritice); // [196, 131, 195, 174, 200, 153, 200, 155, 196,
                      // 130, 195, 142, 200, 152, 200, 154]
    txt = utf8.decode(diacritice);
    print(txt); // ăiștĂÎŞT
}
```

Ascii

Two classes (`AsciiEncoder` and `AsciiDecoder`) and binding class (`AsciiCodec`).

Constructors:

```
AsciiEncoder ()
```

```
AsciiDecoder ({bool allowInvalid = false})
```

Methods:

```
UInt8List AsciiEncoder.convert (String input, [int start = 0, int? end])
```

```
String     AsciiDecoder.convert (List<int> input, [int start = 0, int? end])
```

There is also a constant object define (`ascii` of type `AsciiCodec`) that can be used so that there is no need to create an encoder / decoder. A codec has two method: `.encode` and `.decode` that pretty much call `encoder.convert(...)` and `decoder.convert(...)`.

Ascii

A simple example:

```
import "dart:convert";

void main()
{
    var s = "Hello world";
    var e = ascii.encode(s);
    print(e); // [72, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100]
    print(ascii.decode(e)); // Hello world
}
```

Base64

Two classes (`Base64Encoder` and `Base64Decoder`) and binding class (`Base64Codec`).

Constructors:

`Base64Encoder ()`

`Base64Decoder ()`

Methods:

`String Base64Encoder.convert (List<int> obj)`

`UInt8List Base64Decoder.convert (String input, [int start = 0, int? end])`

There is also a constant object define (`base64` of type `Base64Codec`) that can be used so that there is no need to create an encoder / decoder.

Base64

To decode / encode from strings it is easier to use the utf8 object.

```
import "dart:convert";
void main() {
    var b64 = Base64Encoder().convert(utf8.encode("Hello Dart"));
    print(b64);    // SGVsbG8gRGFydA==
    var bytes = Base64Decoder().convert(b64);
    print(bytes); // [72, 101, 108, 108, 111, 32, 68, 97, 114, 116]
    var s = utf8.decode(bytes);
    print(s);      // Hello Dart
}
```

The same result will be available if **base64** constant object is used with methods encode and decode.

```
var b64 = base64.encode(utf8.encode("Hello Dart"));
var bytes = base64.decode(b64);
```

HTML escape

Dart provides a class `HtmlEscape` that is useful to translate non-HTML characters such as < & “ / into their escaped value that can be used make sure that an HTML is valid.

Constructor:

```
HtmlEscape ([HtmlEscapeMode mode = HtmlEscapeMode.unknown])
```

Methods:

```
String     HtmlEscape.convert (String input)
```

There is also a constant object define (`htmlEscape` of type `HtmlEscape`) that can be used so that there is no need to create an object.

HTML escape

A simple example:

```
import "dart:convert";

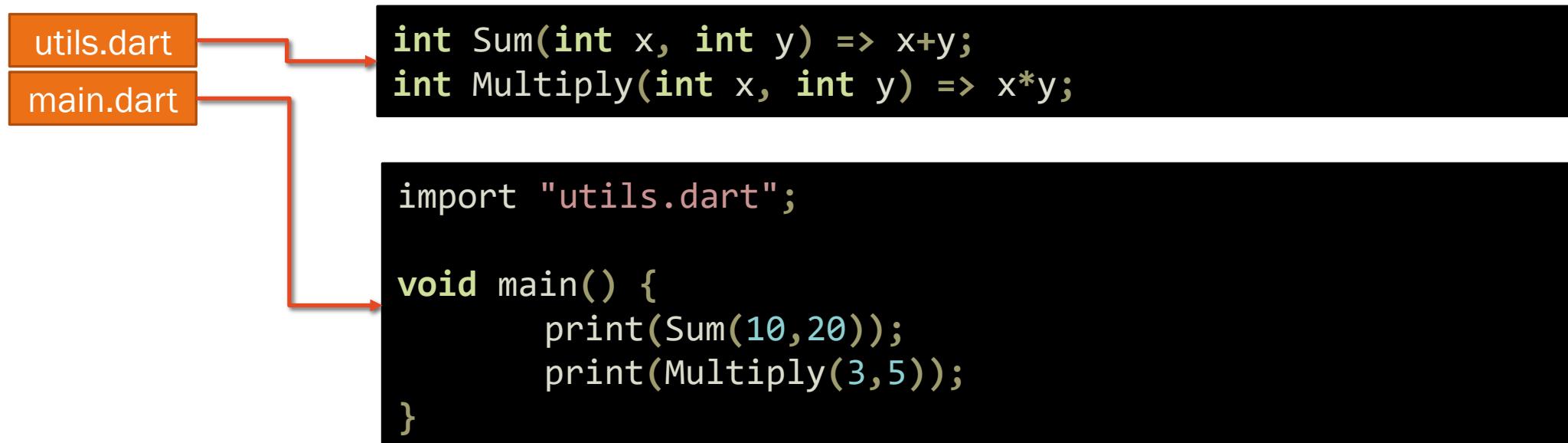
void main()
{
    var s = "10 & 20 > 30 & 40";
    print(HtmlEscape().convert(s));        // 10 & 20 > 30 & 40
    print(htmlEscape.convert(s));         // 10 & 20 > 30 & 40
}
```

In this case ‘&’ is converted to &, ‘>’ is converted into >

Custom libraries

Custom libraries

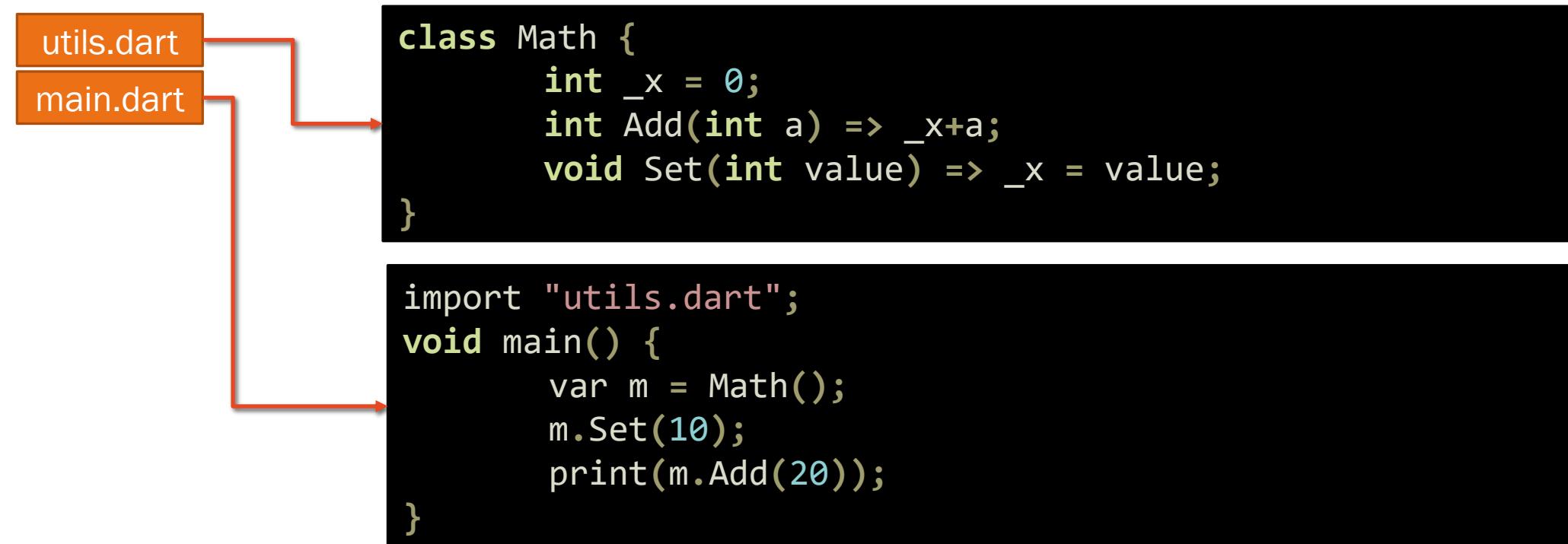
Dart can import a dart file from different locations. Let's consider the following disk organization:



You can run this code via “`dart.exe main.dart`” and It will print on the screen 30 and 15

Custom libraries

The same code applies for classes as well:



Upon execution, this code will write 30 on the screen.

Custom libraries

In this case, “`_x`” is considered private and can not be used outside its file.

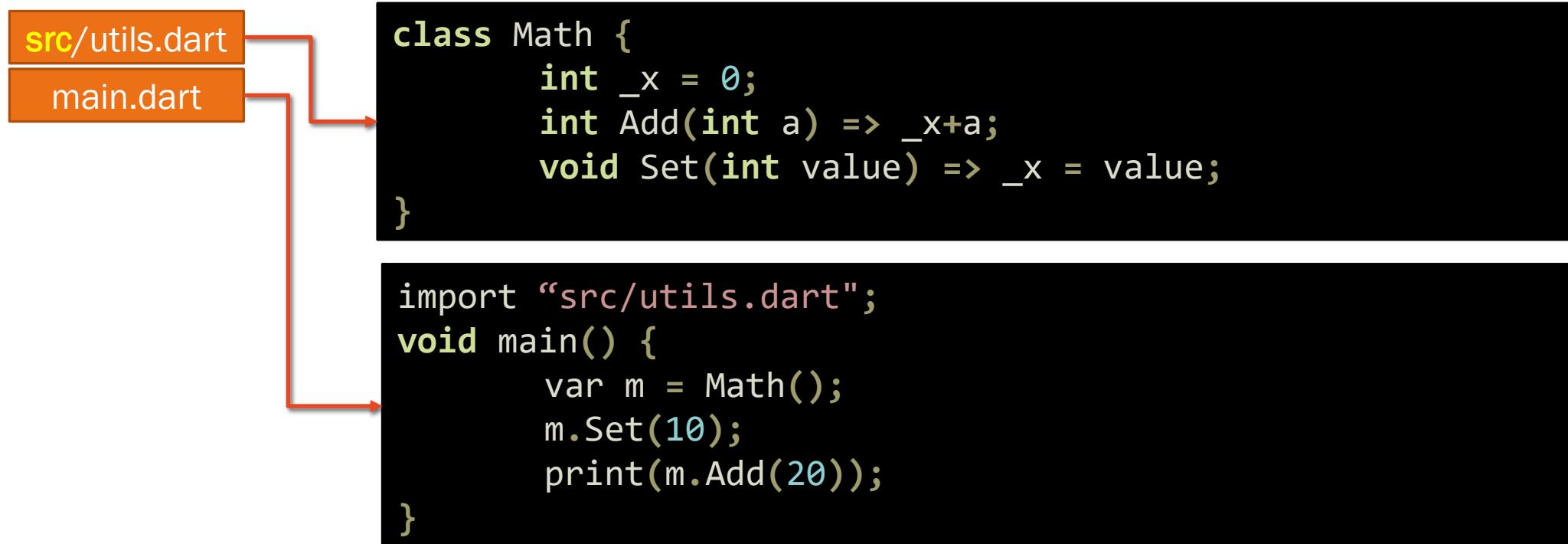
utils.dart
main.dart

```
class Math {  
    int _x = 0;  
    int Add(int a) => _x+a;  
    void Set(int value) => _x = value;  
}
```

```
import "utils.dart";  
void main() {  
    var m = Math();  
    m._x = 10; // Error: The setter '_x' isn't defined  
              // for the class 'Math'.  
}
```

Custom libraries

The same code will work if you use a tree-like folders/sub-folders distribution:



Upon execution, this code will write 30 on the screen.

Custom libraries

A class can also be private if its name is preceded by **_** character.

utils.dart
main.dart

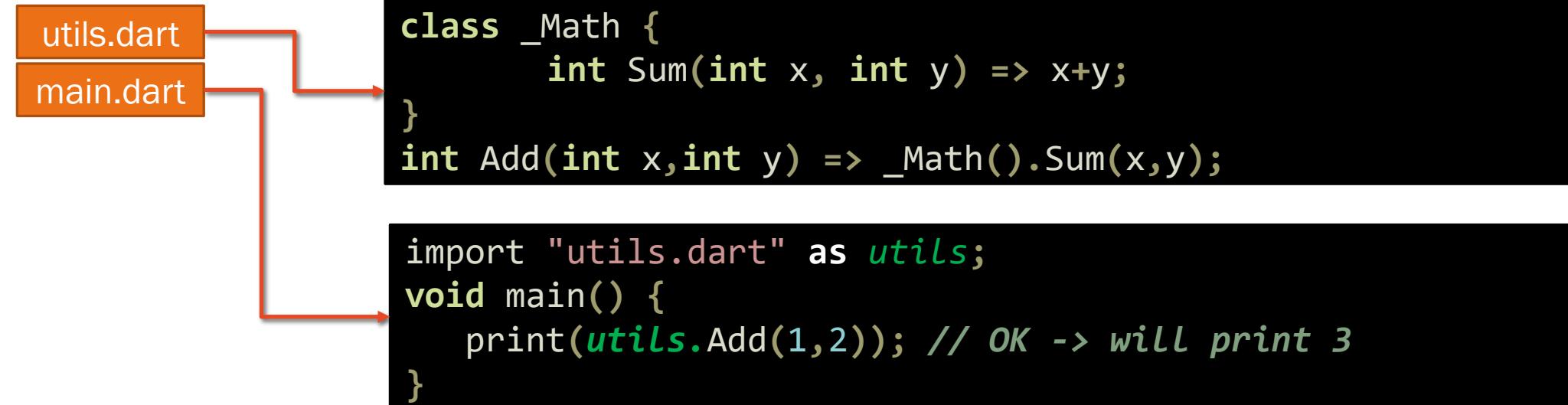
```
class _Math {  
    int Sum(int x, int y) => x+y;  
}  
int Add(int x,int y) => _Math().Sum(x,y);
```

```
import "utils.dart";  
void main() {  
    var m = _Math(); // Compiler error (_Math is private)  
}
```

```
import "utils.dart";  
void main() {  
    print(Add(1,2)); // OK -> will print 3  
}
```

Custom libraries

A class can also be private if its name is preceded by **_** character.



There are no namespaces in Dart. However, when importing from a dart file, an alias can be created and all classes / methods from that file will be associated with that alias creating some sort of namespace.

Q & A

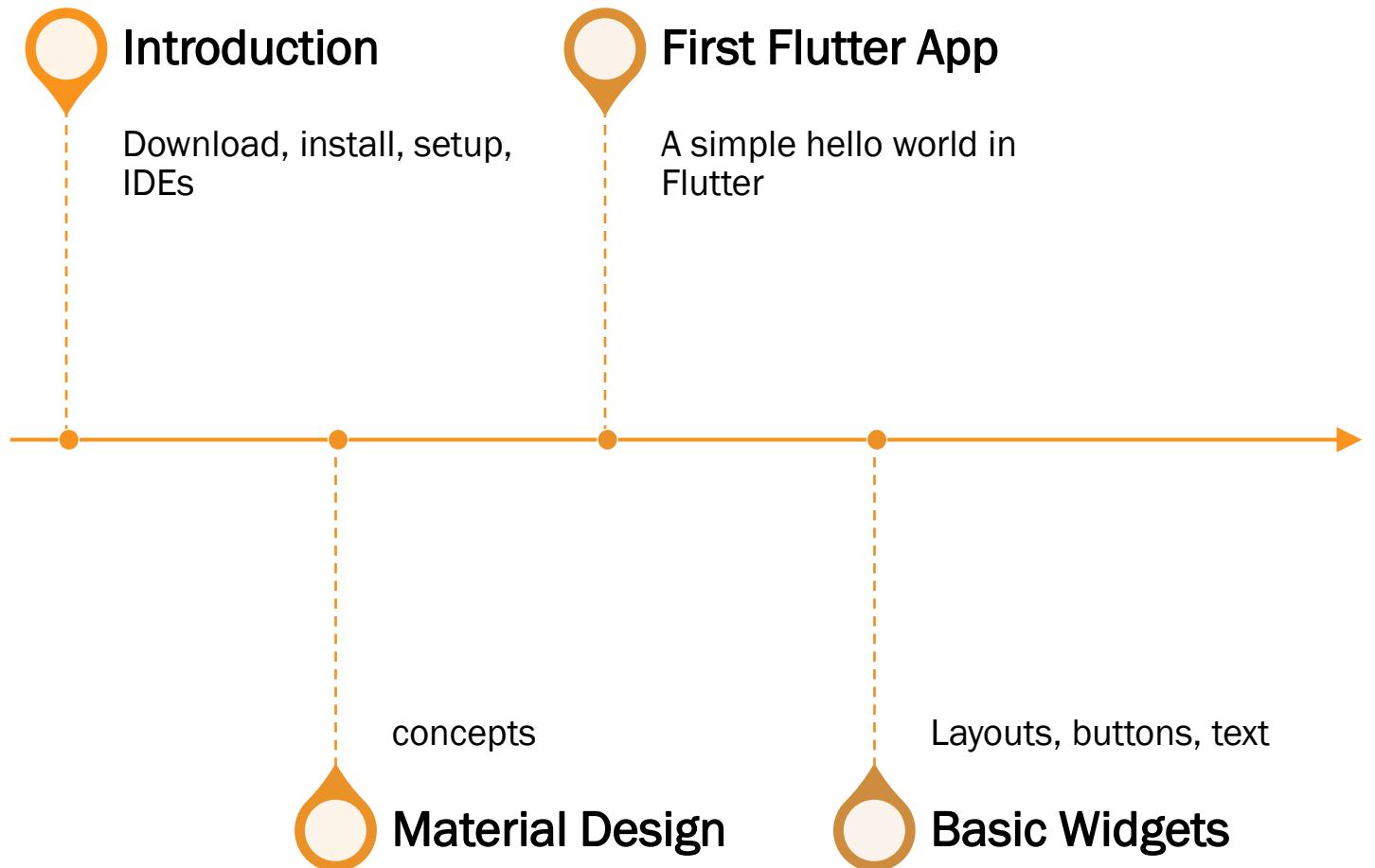


Flutter Framework

COURSE 5 (REV 4)

GAVRILUT DRAGOS

Agenda



Introduction

Introduction

A framework build by Google design to create UX apps for:

- Mobile devices (Android and iOS)
- Desktops (Windows, Linux , MAC/OSX, ChromeOS)
- Web
- Embedded devices

Characteristics:

- Flutter designed was started in 2015 and it was available for usage in 2017
- Based on the paradigm: “**code once, deploy everywhere**”
- It is based on Dart language (meaning that usage of Flutter is easily done via Dart). However, the core engine of Flutter is written in C/C++ for high performance.
- Based on Material design concepts (<https://material.io/>)
- Current version: 2.10.4

Introduction

Website: <https://flutter.dev/> and <https://flutter.dev/development>

Download & Install:

- Windows development : <https://docs.flutter.dev/get-started/install/windows>
- Linux development: <https://docs.flutter.dev/get-started/install/linux>
- MAC/OSX development: <https://docs.flutter.dev/get-started/install/macos>

A separate setup is required for

- Android development
- Windows app development
- Web development

IDEs

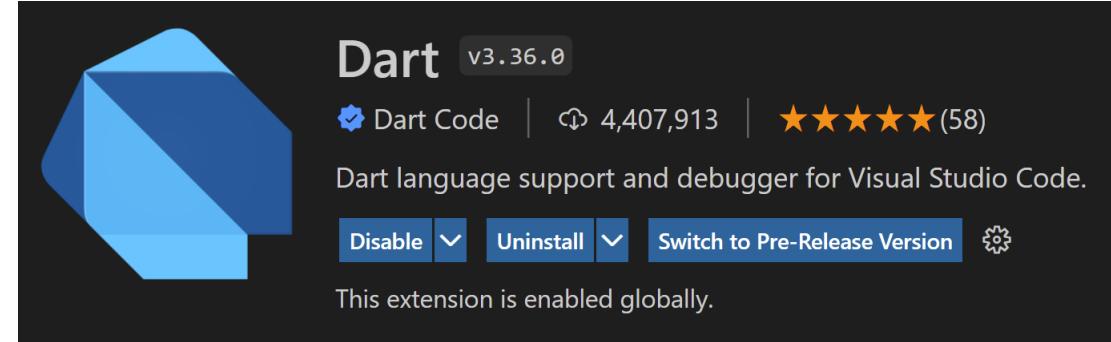
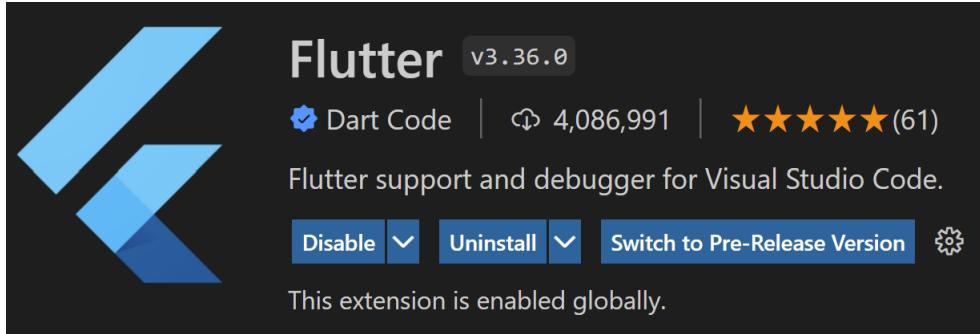
There are multiple IDEs that can be used for Flutter development:

1. Visual Studio Code
2. Android Studio / IntelliJ
3. Emacs
4. DartPad (<https://dartpad.dev>)

For all these editors a plugin(s) will be required to allow integration with Flutter framework (the only exception to this rule being DartPad).

Visual Studio Code

For Visual Studio Code there are two extensions that need to be installed:



With this install you will have access to some new commands (accessible via **Ctrl+Shift+P** (Command Palette) from Visual Studio Code such as:

1. Flutter: New project
2. Flutter: Launch emulator
3. Flutter: Upgrade packages
4.

Flutter CLI

Once flutter is installed, you should see the following files in the <Flutter_installed_folder>/bin

```
cache  
internal  
mingit  
dart  
dart.bat  
flutter  
flutter.bat
```

If your using Windows OS for development use `flutter.bat` otherwise `flutter` (sh version).

It is usually best to make sure that this folder is located in the PATH environment variable and can be used directly from different locations.

Flutter CLI

Flutter CLI can be used to run several commands that are useful to automate build / checking of flutter-based applications:

Generic commands for Flutter SDK

- `channel` List or switch Flutter channels.
- `config` Configure Flutter settings.
- `doctor` Show information about the installed tooling.
- `upgrade` Upgrade your copy of Flutter.

Flutter CLI

Flutter CLI can be used to run several commands that are useful to automate build / checking of flutter-based applications:

Generic commands for Flutter projects

- `analyze` Analyze the project's Dart code.
- `assemble` Assemble and build Flutter resources.
- `build` Build an executable app or install bundle.
- `clean` Delete the build/ and `.dart_tool/` directories.
- `create` Create a new Flutter project.
- `pub` Commands for managing Flutter packages.
- `run` Run your Flutter app on an attached device.

Flutter CLI

Flutter CLI can be used to run several commands that are useful to automate build / checking of flutter-based applications:

Generic commands for tools/devices

- attach Attach to a running app.
- devices List all connected devices.
- emulators List, launch and create emulators.
- install Install a Flutter app on an attached device.
- logs Show log output for running Flutter apps.
- screenshot Take a screenshot from a connected device.

Flutter CLI

Flutter doctor: `flutter.bat doctor` is a very useful command to check the status of flutter framework.

```
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, 2.10.3, on Microsoft Windows [Version 10.0.22000.556], locale en-US)
[✗] Android toolchain - develop for Android devices
    ✗ Unable to locate Android SDK.
        Install Android Studio from: https://developer.android.com/studio/index.html
        On first launch it will assist you in installing the Android SDK components.
        (or visit https://flutter.dev/docs/get-started/install/windows#android-setup for detailed instructions).
        If the Android SDK has been installed to a custom location, please use
        `flutter config --android-sdk` to update to that location.

[✓] Chrome - develop for the web
[✓] Visual Studio - develop for Windows (Visual Studio Community 2022 17.2.0 Preview 2.1)
[!] Android Studio (not installed)
[✓] Connected device (3 available)
[✓] HTTP Host Availability

! Doctor found issues in 2 categories.
```

In this example, there are two problems: Android toolchain is not installed, and android studio is not installed. However, since Visual Studio is present, building a Windows app is possible. Make sure that you have the latest toolchain (latest update of Visual Studio).

Flutter CLI

To create a project, you can either use an option from the IDE or type “`flutter.bat create <name>`” from command line. For example: running “`flutter create my_project`” will should create the following output.

```
>flutter create my_project
Creating project my_project...
Running "flutter pub get" in my_project...
Wrote 96 files.

1,374ms

All done!
In order to run your application, type:

$ cd my_project
$ flutter run

Your application code is in my_project\lib\main.dart.
```

Flutter CLI

To create a project, you can either use an option from the IDE or type “`flutter.bat create <name>`” from command line. For example: running “`flutter create my_project`” will should create the following output. You should see a new folder “my_project” that has the following content:

```
.dart_tool
.idea
android
ios
lib
test
web
windows
.gitignore
.metadata
.packages
analysis_options.yaml
my_project.iml
pubspec.lock
pubspec.yaml
README.md
```

Flutter CLI

To create a project, you can either use an option from the IDE or type “`flutter.bat create <name>`” from command line. For example: running “`flutter create my_project`” will should create the following output. You should see a new folder “my_project” that has the following content:

```
.dart_tool
.idea
android
ios
lib
test
web
windows
.gitignore
.metadata
.packages
analysis_options.yaml
my_project.iml
pubspec.lock
pubspec.yaml
README.md
```

← Folder for Android build (manifest, resource, class
wrappers, etc)

Flutter CLI

To create a project, you can either use an option from the IDE or type “`flutter.bat create <name>`” from command line. For example: running “`flutter create my_project`” will should create the following output. You should see a new folder “my_project” that has the following content:

```
.dart_tool  
.idea  
android  
ios  
lib  
test  
web  
windows  
.gitignore  
.metadata  
.packages  
analysis_options.yaml  
my_project.iml  
pubspec.lock  
pubspec.yaml  
README.md
```

Folder for iOS build (pList, xcode files, etc)

Flutter CLI

To create a project, you can either use an option from the IDE or type “`flutter.bat create <name>`” from command line. For example: running “`flutter create my_project`” will should create the following output. You should see a new folder “my_project” that has the following content:

```
.dart_tool  
.idea  
android  
ios  
lib  
test  
web  
windows  
.gitignore  
.metadata  
.packages  
analysis_options.yaml  
my_project.iml  
pubspec.lock  
pubspec.yaml  
README.md
```

Folder for web browsers build (index.html,
manifest.json, etc)

Flutter CLI

To create a project, you can either use an option from the IDE or type “`flutter.bat create <name>`” from command line. For example: running “`flutter create my_project`” will should create the following output. You should see a new folder “my_project” that has the following content:

```
.dart_tool
.idea
android
ios
lib
test
web
windows
.gitignore
.metadata
.packages
analysis_options.yaml
my_project.iml
pubspec.lock
pubspec.yaml
README.md
```

Folder for Windows build (Cpp and header files,
cmake file, resources)

Flutter CLI

To create a project, you can either use an option from the IDE or type “`flutter.bat create <name>`” from command line. For example: running “`flutter create my_project`” will should create the following output. You should see a new folder “my_project” that has the following content:

```
.dart_tool  
.idea  
android  
ios  
lib  
test  
web  
windows  
.gitignore  
.metadata  
.packages  
analysis_options.yaml  
my_project.iml  
pubspec.lock  
pubspec.yaml  
README.md
```

The actual dart code (one file: main.dart)

This is the code that will be compiled once
and deploy on Android, iOS, Windows or web

Flutter CLI

To build a flutter-based application, one can use an IDE command or just enter the project folder and run “`flutter.bat build <system>`” from command line, where `<system>` can be “`windows`”, “`android`”, etc.

For example: running “`flutter build windows`” from the folder “`my_project`” created on the previous step should result in the following:

```
\my_project>flutter build windows  
Building with sound null safety  
Building Windows application...
```

And you should see the following files in `my_project\build\windows\runner\Release`

```
data  
flutter_windows.dll  
my_project.exe
```

Flutter CLI

To run a flutter project you can:

1. Run a command from the IDE (e.g. F5 from Visual Studio Code)
2. Build and run the executable from `<project folder>\build\windows\runner\Release` (this is specific for Windows only → for other environments the location is different or you might need to install on a specific device first)
3. Type “`flutter.bat run`” in the project folder from the command line. If multiple options are available, you will be asked to select one. For example: if both windows and web based are possible, you should see something like this:

```
Multiple devices found:  
Windows (desktop) • windows • windows-x64     • Microsoft Windows [Version 10.0.22000.556]  
Chrome (web)       • chrome   • web-javascript • Google Chrome 99.0.4844.51  
Edge (web)        • edge     • web-javascript • Microsoft Edge 97.0.1072.55  
[1]: Windows (windows)  
[2]: Chrome (chrome)  
[3]: Edge (edge)  
Please choose one (To quit, press "q/Q"): 1  
Launching lib\main.dart on Windows in debug mode...  
Building Windows application...  
Syncing files to device Windows... 119ms
```

Material design

Material Design

A concept design and adopted by Google related to how an application (either for web or mobile) should be constructed (in terms of visibility, accessibility, usability, etc)

Characteristics:

- Material Design was announced in 2014
- Since then it has been adopted in all major Google platforms (such as gmail, google drive, google docs)
- Website (<https://material.io/>)
- Current version: 3
- It provides a sort of rules (“best practices”) on how these widgets should be used.
- It is available for:
 - Android
 - iOS
 - Web
 - Flutter

Material Design

For example, for a Button, you can read more on how-to-use on the following link:

<https://material.io/components/buttons>

Usage

Buttons communicate actions that users can take. They are typically placed throughout your UI, in places like:

- Dialogs
- Modal windows
- Forms
- Cards
- Toolbars

Material Design

For example, for a Button, you can read more on how-to-use on the following link:

<https://material.io/components/buttons>

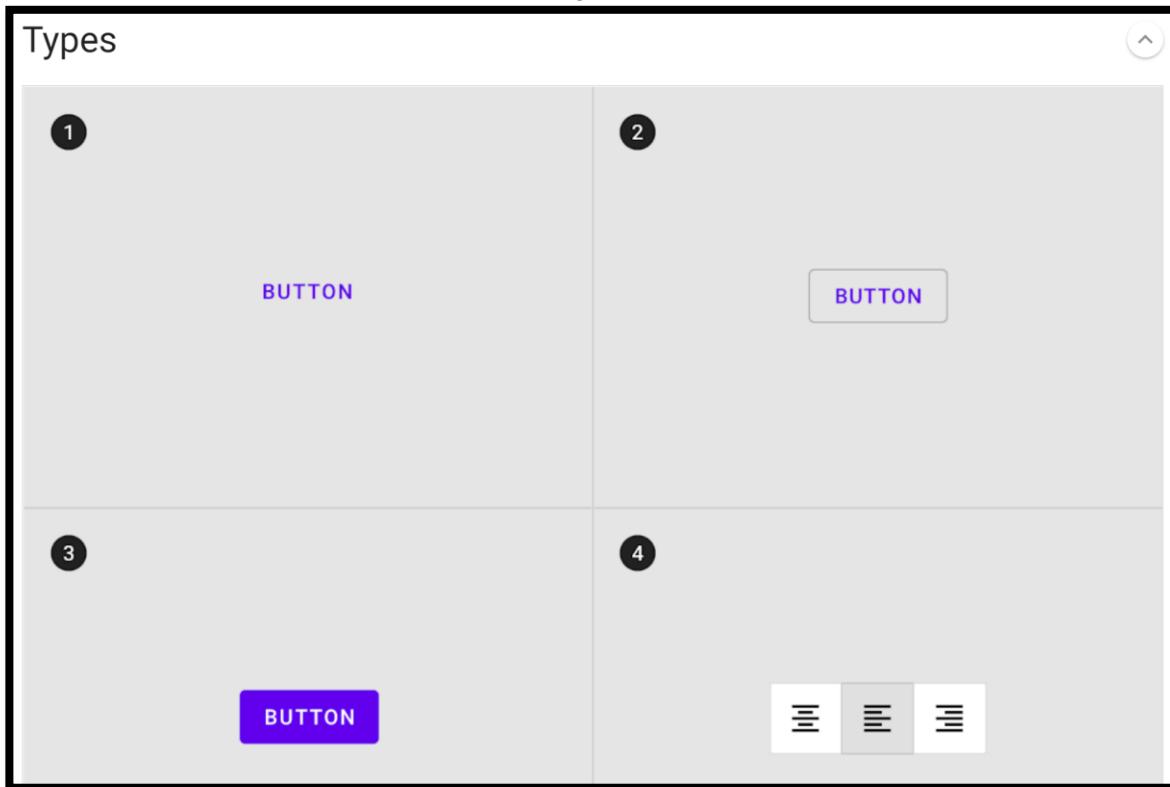
Principles

The image shows three cards, each featuring a large central button icon with a plus sign inside, set against a pink background. The first card, titled 'Identifiable', shows the button surrounded by other UI elements like a switch and an info icon. The second card, titled 'Findable', shows the button being magnified by a large circular magnifying glass. The third card, titled 'Clear', shows the button with a purple shadow effect.

Identifiable	Findable	Clear
Buttons should indicate that they can trigger an action.	Buttons should be easy to find among other elements, including other buttons.	A button's action and state should be clear.

Material Design

For example, for a Button, you can read more on how-to-use on the following link:



1. Text button (low emphasis)

Text buttons are typically used for less important actions.

2. Outlined Button (medium emphasis)

Outlined buttons are used for more emphasis than text buttons due to the stroke.

3. Contained button (high emphasis)

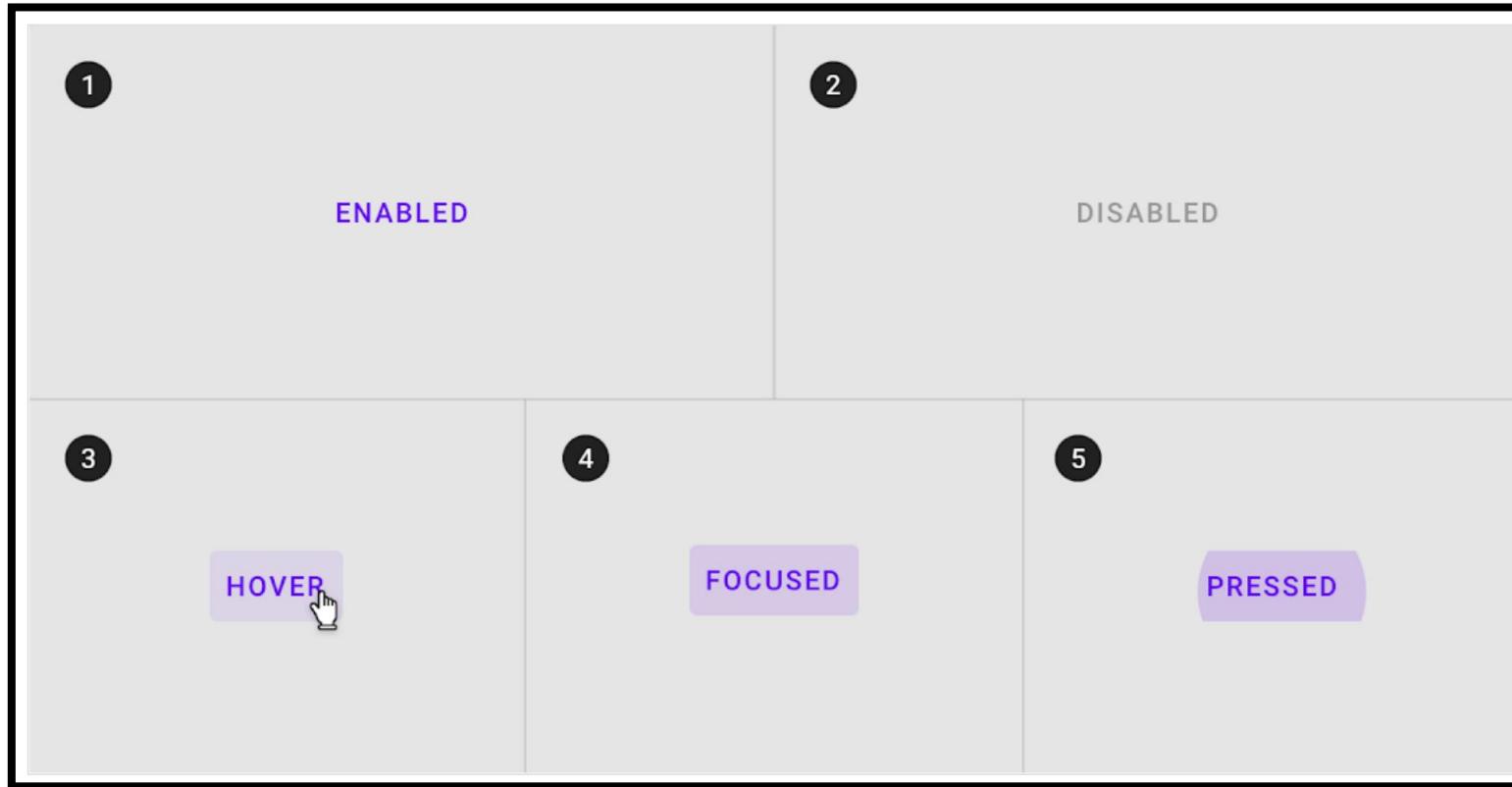
Contained buttons have more emphasis, as they use a color fill and shadow.

4. Toggle button

Toggle buttons group a set of actions using layout and spacing. They're used less often than other button types.

Material Design

For example, for a Button, you can read more on how-to-use on the following link:



Material Design

For example, for a Button, you can read more on how-to-use on the following link:



First flutter App

First flutter App

Create a new flutter app and replace the code from main.dart with the following one:

```
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();
  }
}
```

Run “flutter run” and try both windows and browser versions. In both cases you should see an empty window or an empty chrome tab.

First flutter App

Create a new flutter app and replace the code from main.dart with the following one:

```
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();
    }
}
```

The main function usually consists in a `runApp` method that receives a Widget

Every UX control in Flutter is a Widget.

First flutter App

Create a new flutter app and replace the code from main.dart with the following one:



The most important method a **StatelessWidget** or a **StatefulWidget** is:

Widget build (BuildContext context)

This is the method that is used by flutter framework to create a tree of widgets that describe how elements are going to be draw on the screen.

First flutter App

Create a new flutter app and replace the code from main.dart with the following one:

```
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();
  }
}
```

In our example, the method build returns a new Widget of type MaterialApp().

First flutter App

MaterialApp constructor looks has multiple parameters, the most important ones being:

```
MaterialApp (  
    Widget? home,  
    TransitionBuilder? builder,  
    String title = '',  
    Color? color,  
    ThemeData? theme,  
    ThemeMode? themeMode = ThemeMode.system,  
    Locale? locale,  
    bool debugShowMaterialGrid = false,  
    bool showPerformanceOverlay = false,  
    bool checkerboardRasterCacheImages = false,  
    bool checkerboardOffscreenLayers = false,  
    bool showSemanticsDebugger = false )
```

Out of these, the most important one is the `home` parameter that describes the top-level widget of the app.

This mean that at minimum, we need to add one widget to display something on our app.

First flutter App

So ... let's change the previous app code and add another widget (a very simple one):

```
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: const Text("Hello, world"));
  }
}
```

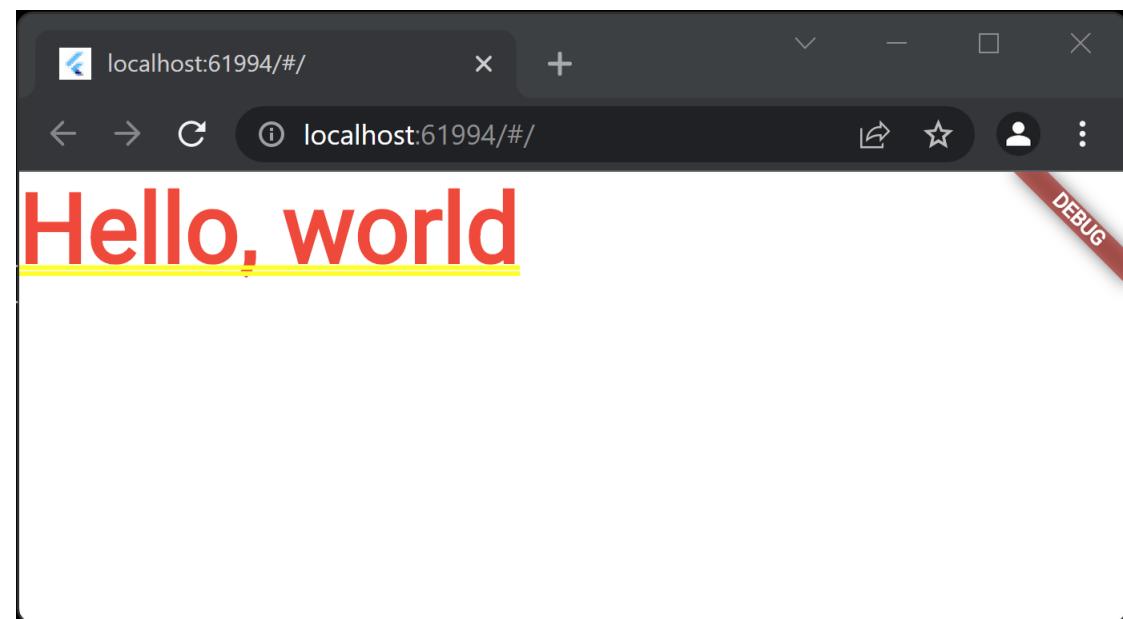
In this case we have used a Text widget to print the classical text “Hello, world”

First flutter App

When we execute the previous code via “`flutter run`” command we get:



or



Window app

The difference in this case is that the windows app uses system theme (Dark mode) while the browser just renders in white.

Chrome browser

First flutter App

You can also test the same code in DartPad (if you don't have the flutter framework installed).

The screenshot shows the DartPad interface with the following details:

- Title Bar:** DartPad
- Address Bar:** dartpad.dev/?
- Toolbar:** DartPad, New Pad, Reset, Format, Install SDK, crimson-lotus-1697, Samples, More
- Code Editor:** Displays the following Dart code:

```
1 import 'package:flutter/material.dart';
2
3 void main() {
4   runApp(const MyApp());
5 }
6
7 class MyApp extends StatelessWidget {
8   const MyApp({Key? key}) : super(key: key);
9
10  @override
11  Widget build(BuildContext context) {
12    return MaterialApp(home: const Text("Hello, world"));
13  }
14 }
15
```
- Output Area:** Shows the text "Hello, world" in red, with a yellow underline. A red ribbon in the top right corner says "DEBUG".
- Bottom Navigation:** Console, Documentation
- Bottom Status Bar:** info line 12 • Prefer const with constant constructors. (view docs), Privacy notice, Send feedback, stable channel, 1 issue hide, Based on Flutter 2.10.4 Dart SDK 2.16.2

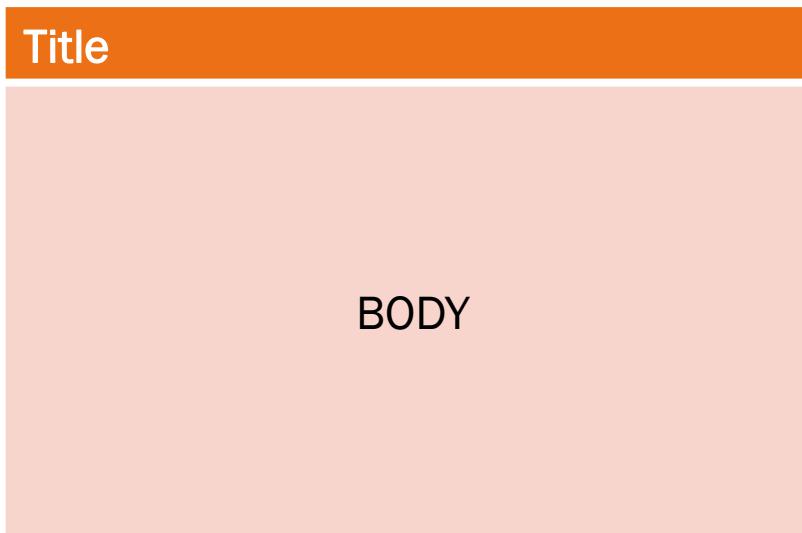
Basic widgets

Scaffold widget

A scaffold widget is a widget that allows creating a material view for your app that has:

- An application bar
- A body

This is very similar to how Android apps or in general web base google apps look like:

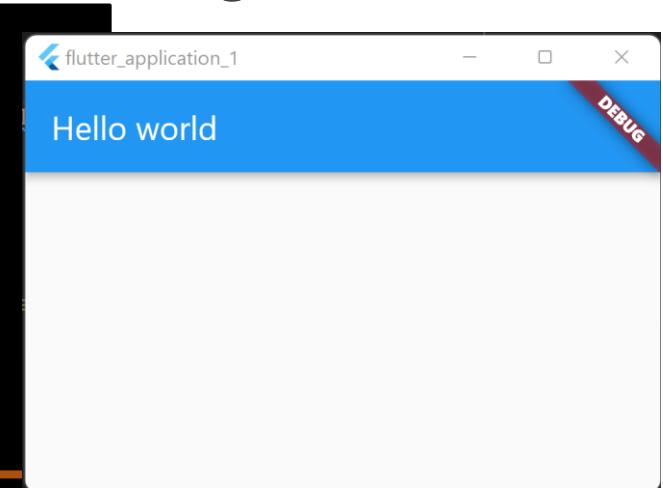


Scaffold widget

Create a new flutter app and replace the code from main.dart with the following one:

```
import 'package:flutter/material.dart';
void main() { runApp(const MyApp()); }

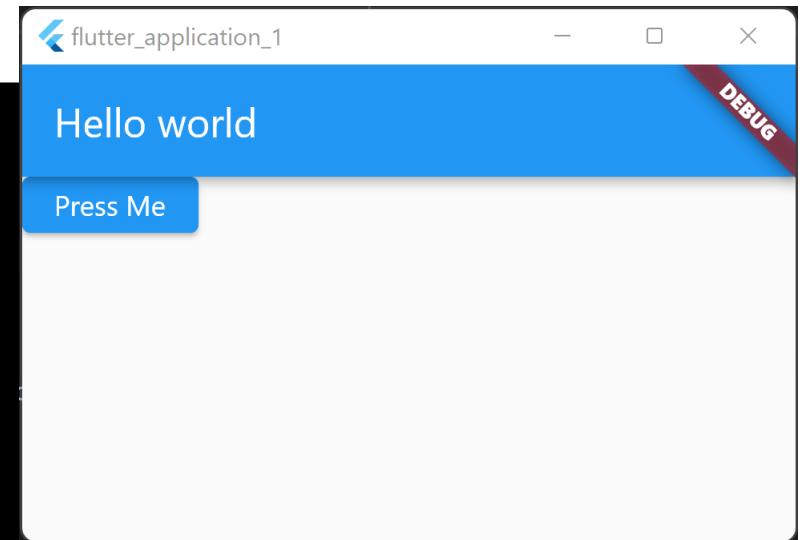
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("Hello world")),
      )
    );
  }
}
```



ElevatedButton widget

Let's add a body with a button:

```
import 'package:flutter/material.dart';
void main() { runApp(const MyApp()); }
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
  Widget GetBody() {
    return ElevatedButton( child: Text("Press Me"),
                          onPressed: () => {});
  }
  @override
  Widget build(BuildContext context) {
    return MaterialApp( home: Scaffold(
      appBar: AppBar(title: Text("Hello world")), body: GetBody()));
  }
}
```



ElevatedButton widget

To simplify the code even more, let's consider the `getBody` the method we are going to change from this moment on, and the rest of the code will remain the same.

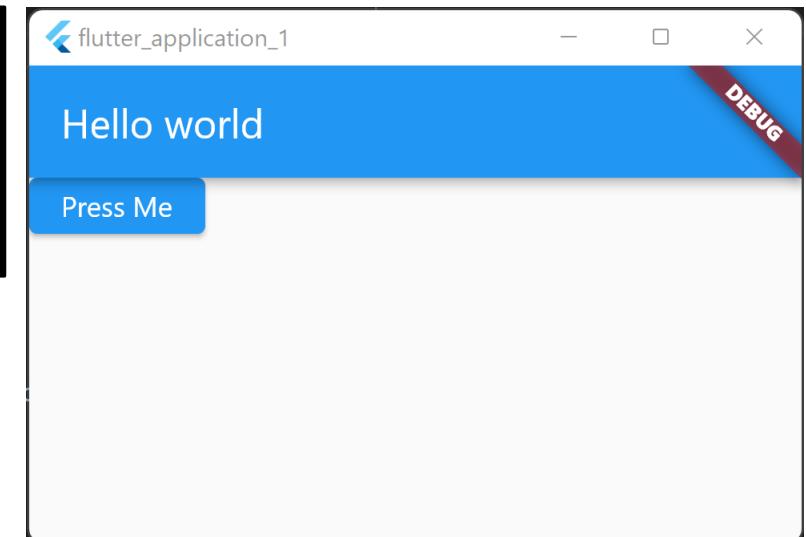
```
import 'package:flutter/material.dart';
void main() { runApp(const MyApp()); }
class MyApp extends StatelessWidget {
    const MyApp({Key? key}) : super(key: key);
    Widget GetBody() { ... }
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("Hello world")),
                body: GetBody()));
    }
}
```

ElevatedButton widget

Let's add a body with a button:

```
Widget GetBody() {  
    return ElevatedButton( child: Text("Press Me"),  
                          onPressed: () => {});  
}
```

One observation in this case is that the button is located in the top-left corner. To change this behavior we need to use various containers and layouts.



Layout widgets

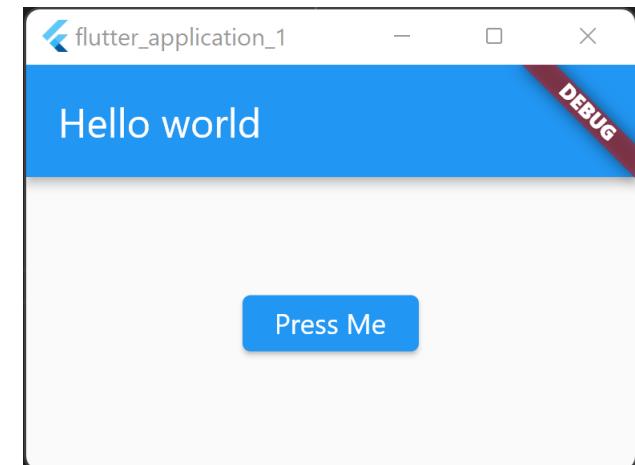
Flutter has multiple layouts – the most common one being:

1. Center → the entire layout is position to the center of the app
2. SizedBox → allows one to specify the size of a container
3. Padding → allows one to add a margin
4. Column → multiple elements are added in a vertical flow
5. Row → multiple widgets are added in a horizontal flow
6. Container → a generic container
7. Positioned → to select a specific location for another widget
8. FractionallySizedBox → Similar to SizedBox with the only difference that the width and height are provided as percentages from the parent

Center widget

A center widget has one child, and its child will be center to the parent of the center widget (in this case the body from the scaffold widget from the app.):

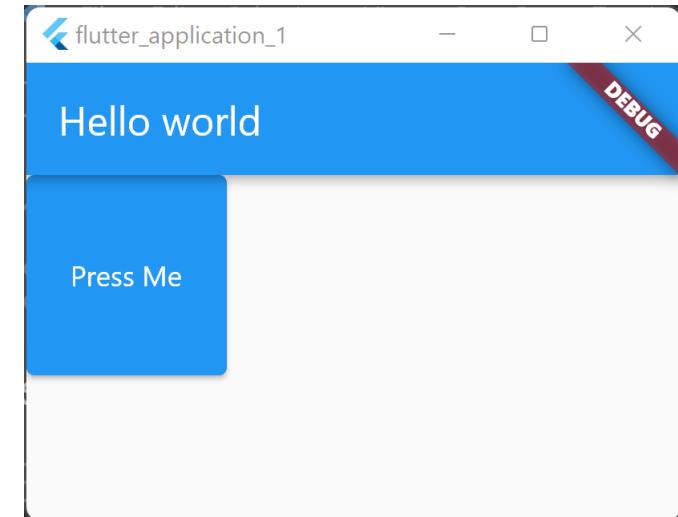
```
Widget GetBody() {  
    return Center(  
        child: ElevatedButton(  
            child: Text("Press Me"),  
            onPressed: () => {},  
        ));  
}
```



SizeBox widget

A `SizeBox` widget has one child, a `width` and a `height` property. In this example, `width` and `height` are set to 100.

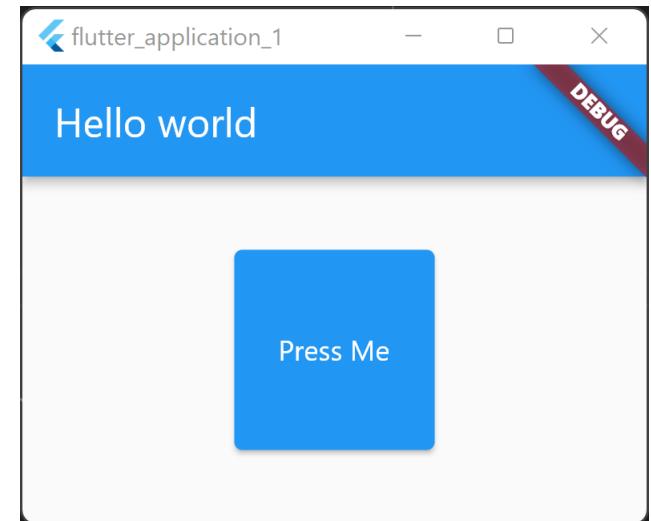
```
Widget GetBody() {  
  return SizedBox(  
    width: 100,  
    height: 100,  
    child: ElevatedButton(  
      child: Text("Press Me"),  
      onPressed: () => {},  
    ),  
};
```



SizeBox widget

All containers / layouts can be used together. This means that a SizedBox can be the child of a Center widths, or vice-versa.

```
Widget GetBody() {  
  return Center(  
    child: SizedBox(  
      width: 100,  
      height: 100,  
      child: ElevatedButton(  
        child: Text("Press Me"),  
        onPressed: () => {},  
      )),  
  );  
}
```

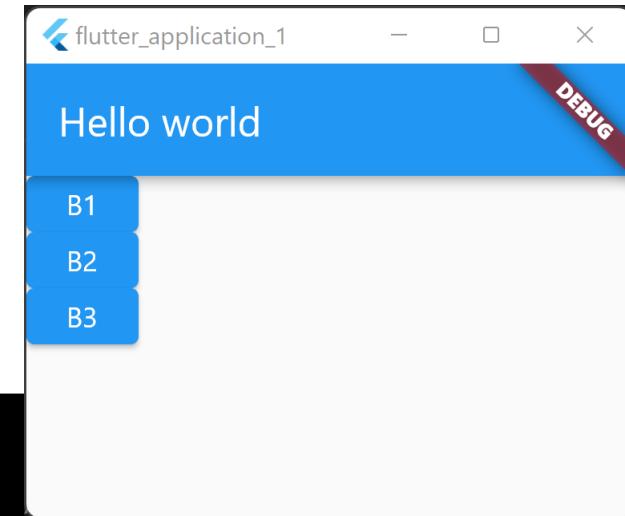


Now we have a 100x100 button centered.

Column widget

A Column or a Row widgets have multiple children. This means that their main property is children (that is a list of other widgets that are going to be added one after another (just like in a stack). In case of Column the stack will be vertical, while in case of Raw the stack will be displayed horizontal.

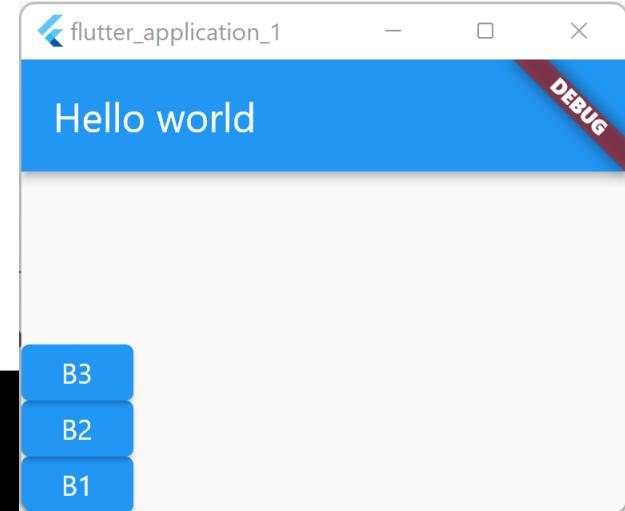
```
Widget GetBody() {  
    return Column(children: [  
        ElevatedButton(child: Text("B1"), onPressed: () => {}),  
        ElevatedButton(child: Text("B2"), onPressed: () => {}),  
        ElevatedButton(child: Text("B3"), onPressed: () => {})  
    ]);  
}
```



Column widget

Another property of Column widget is `VerticalDirection` with two possible value (up or down). If set to `up` the children are going to be arrange from the bottom to top the reverse order (last one will be on the bottom, previous one will be on top of the last one, and so on ...)

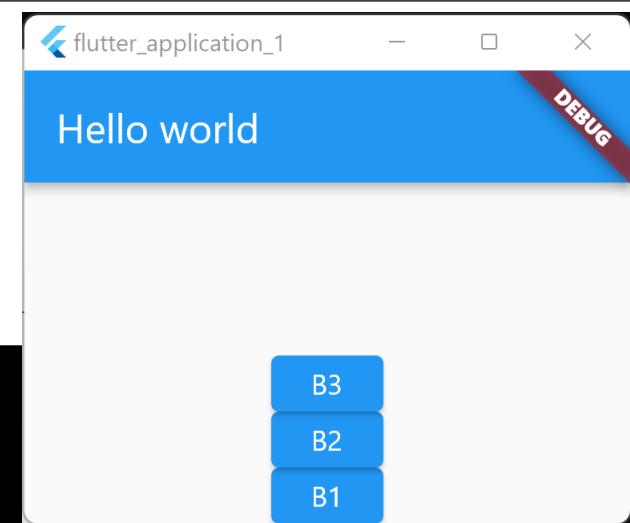
```
Widget GetBody() {  
    return Column(children: [  
        ElevatedButton(child: Text("B1"), onPressed: () => {}),  
        ElevatedButton(child: Text("B2"), onPressed: () => {}),  
        ElevatedButton(child: Text("B3"), onPressed: () => {})  
    ], verticalDirection: VerticalDirection.up);  
}
```



Column widget

We can use a Center widget to center the column stack horizontally. Keep in mind that the column stack is as large as the height of its parent (in this case the body of the scaffold object).

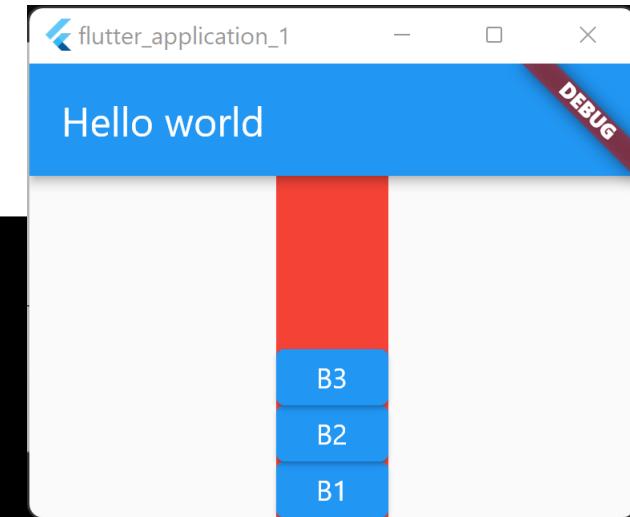
```
Widget GetBody() {  
    return Center(  
        child: Column(children: [  
            ElevatedButton(child: Text("B1"), onPressed: () => {}),  
            ElevatedButton(child: Text("B2"), onPressed: () => {}),  
            ElevatedButton(child: Text("B3"), onPressed: () => {})  
        ], verticalDirection: VerticalDirection.up));  
}
```



Column widget

To validate the previous assumption, lets use a container widget with a background color of red set up.

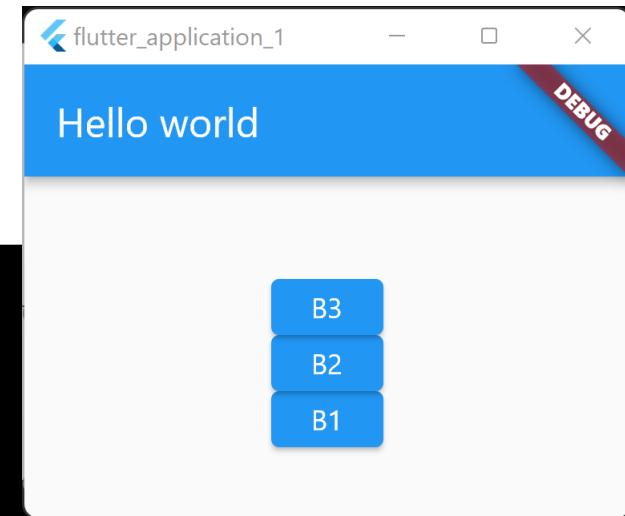
```
Widget GetBody() {  
    return Center(  
        child: Container(  
            color: Colors.red,  
            child: Column(children: [  
                ElevatedButton(child: Text("B1"), onPressed: () => {}),  
                ElevatedButton(child: Text("B2"), onPressed: () => {}),  
                ElevatedButton(child: Text("B3"), onPressed: () => {})  
            ], verticalDirection: VerticalDirection.up)));  
}
```



Column widget

We can use a `SizeBox` between `Center` widget and `Column` widget to limit the height to 100. This will somehow center all of the buttons (as long as their combine height is 100 ... or close to it !).

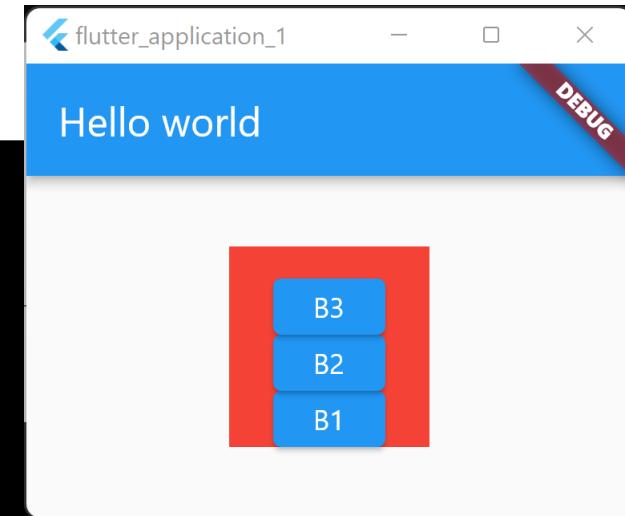
```
Widget GetBody() {  
    return Center(  
        child: SizedBox(  
            width: 100,  
            height: 100,  
            child: Column(children: [  
                ElevatedButton(child: Text("B1"), onPressed: () => {}),  
                ElevatedButton(child: Text("B2"), onPressed: () => {}),  
                ElevatedButton(child: Text("B3"), onPressed: () => {})  
            ], verticalDirection: VerticalDirection.up)));  
}
```



Column widget

The reason we are formulating that it will not be perfect can be checked with a Container with color property set.

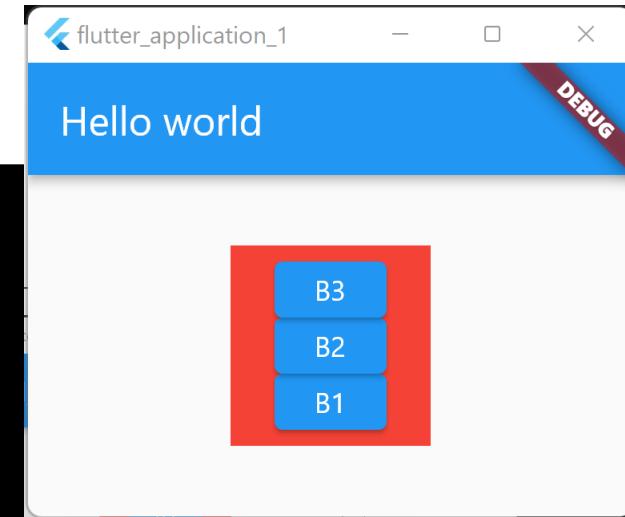
```
Widget GetBody() {  
    return Center(  
        child: SizedBox(  
            width: 100,  
            height: 100,  
            child: Container(  
                color: Colors.red,  
                child: Column(children: [  
                    ElevatedButton(child: Text("B1"), onPressed: () => {}),  
                    ElevatedButton(child: Text("B2"), onPressed: () => {}),  
                    ElevatedButton(child: Text("B3"), onPressed: () => {})  
                ], verticalDirection: VerticalDirection.up))));  
}
```



Column widget

But, what if we want to make sure that the pile of buttons are perfectly centered → we can use the mainAxisAlign property.

```
Widget GetBody() {  
    return Center(  
        child: SizedBox( width: 100, height: 100,  
            child: Container( color: Colors.red,  
                child: Column(  
                    children: [  
                        ElevatedButton(child: Text("B1"), onPressed: () => {}),  
                        ElevatedButton(child: Text("B2"), onPressed: () => {}),  
                        ElevatedButton(child: Text("B3"), onPressed: () => {})  
                    ],  
                    verticalDirection: VerticalDirection.up,  
                    mainAxisAlignment: MainAxisAlignment.center))));  
}
```

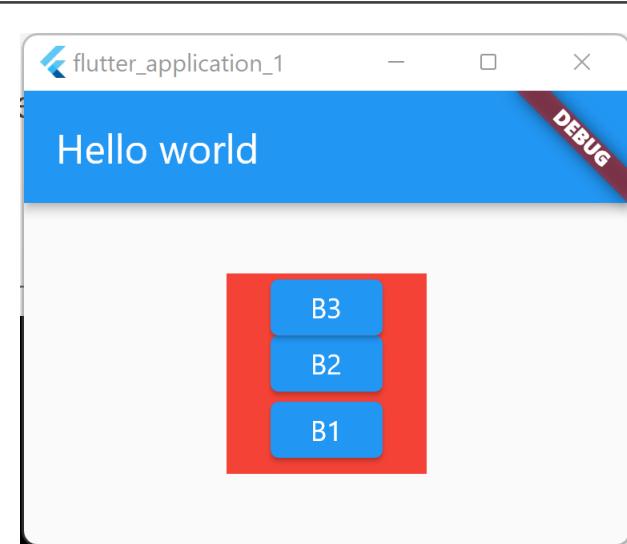


Padding widget

If we want to add some margins between widgets, we can use the Padding widget and select a specific margins (for left, right, bottom or top).

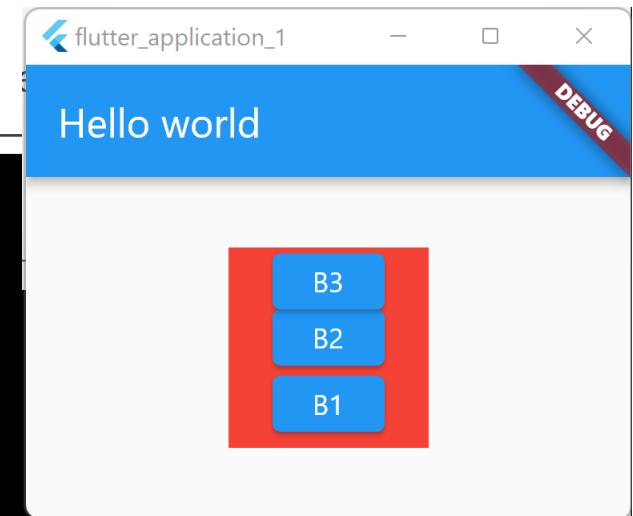
After we do this (the code will be displayed in the next slide) the outcome should be similar as the one from the next picture (you can see that button B1 has some margins between him and the rest of the bottoms).

To do this, use the property padding from the Padding widget.



Padding widget

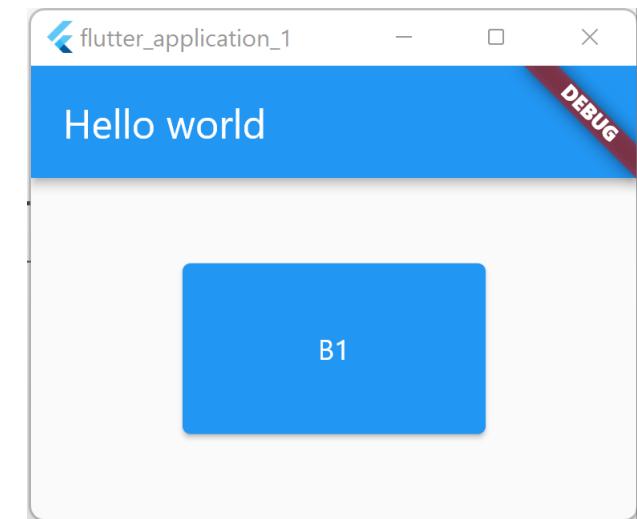
```
Widget GetBody() {  
    return Center(  
        child: SizedBox( width: 100,height: 100,  
            child: Container( color: Colors.red,  
                child: Column(  
                    children: [  
                        Padding(  
                            padding: EdgeInsets.fromLTRB(5, 5, 5, 5),  
                            child: ElevatedButton(  
                                child: Text("B1"), onPressed: () => {}),  
                                ElevatedButton(child: Text("B2"), onPressed: () => {}),  
                                ElevatedButton(child: Text("B3"), onPressed: () => {})  
                            ],  
                            verticalDirection: VerticalDirection.up,  
                            mainAxisAlignment: MainAxisAlignment.center,)))));  
}
```



FractionallySizedBox widget

Using FractionallySizedBox can provide a way to select the size of an object based on percentages from the width and height of its parent (in this case 50% or width and 50% of height)

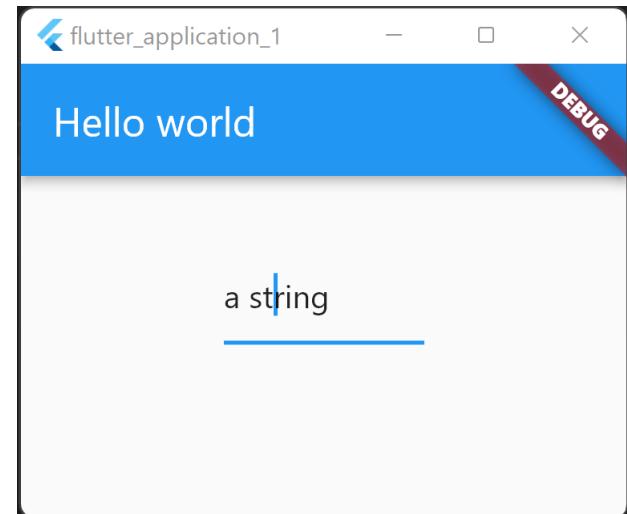
```
Widget GetBody() {  
    return Center(  
        child: FractionallySizedBox(  
            widthFactor: 0.5,  
            heightFactor: 0.5,  
            child: ElevatedButton(  
                child: Text("B1"),  
                onPressed: () => {}),  
        ));  
}
```



TextField widget

The TextField widget can be used to write / arrange or edit a text. It can also be used as a label if we set the read only property.

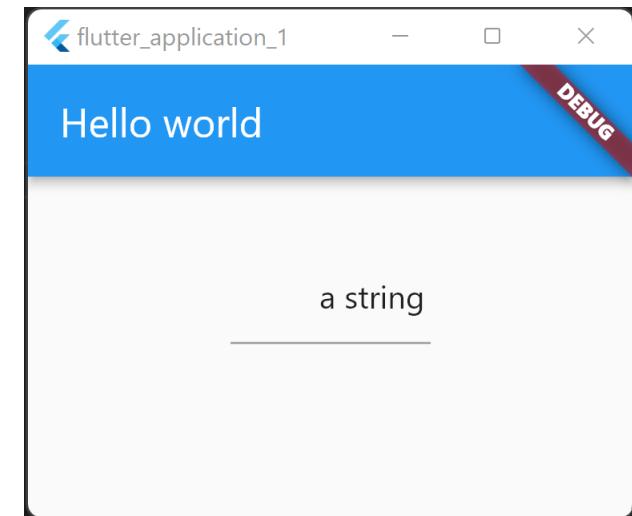
```
Widget GetBody() {  
  return Center(  
    child: SizedBox(  
      width: 100,  
      height: 100,  
      child: TextFormField(  
        initialValue: "a string",  
      )),  
}
```



TextField widget

We can also align the text (left or right) or make it read-only.

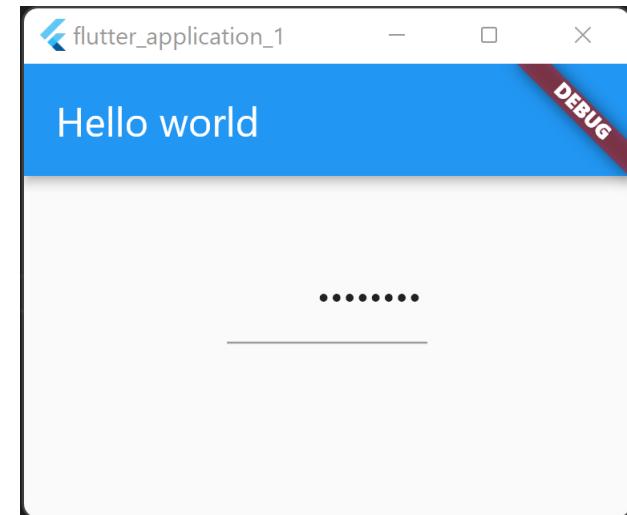
```
Widget GetBody() {  
  return Center(  
    child: SizedBox(  
      width: 100,  
      height: 100,  
      child: TextFormField(  
        initialValue: "a string",  
        textAlign: TextAlign.end,  
        readOnly: true,  
      )),  
}
```



TextField widget

We can make it work like a password control (via obscureText parameter).

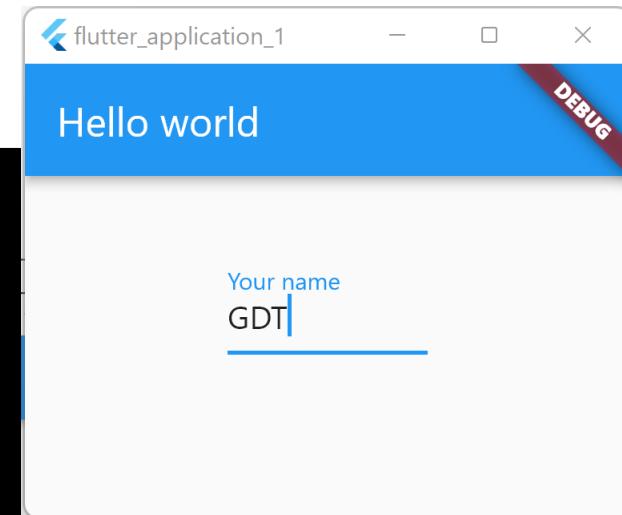
```
Widget GetBody() {  
    return Center(  
        child: SizedBox(  
            width: 100,  
            height: 100,  
            child: TextField(  
                initialValue: "a string",  
                textAlign: TextAlign.end,  
                obscureText: true,  
            )));  
}
```



TextField widget

You can also use an InputDecorator to add a label that explain what that text form is all about.

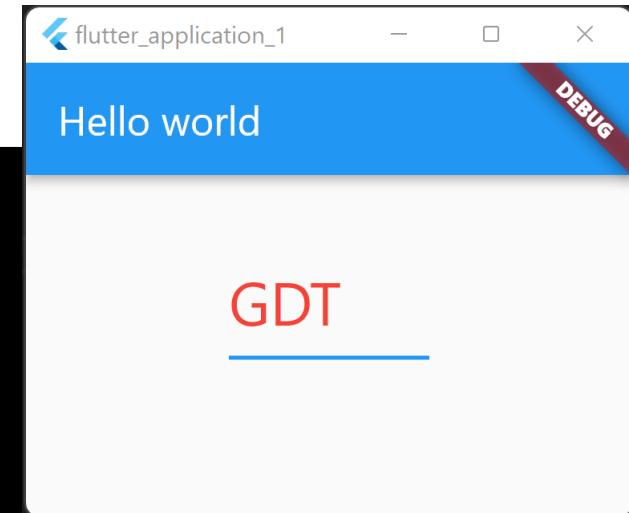
```
Widget GetBody() {  
    return Center(  
        child: SizedBox(  
            width: 100,  
            height: 100,  
            child: TextFormField(  
                initialValue: "GDT",  
                decoration: const InputDecoration(labelText: "Your name"),  
            )));  
}
```



TextField widget

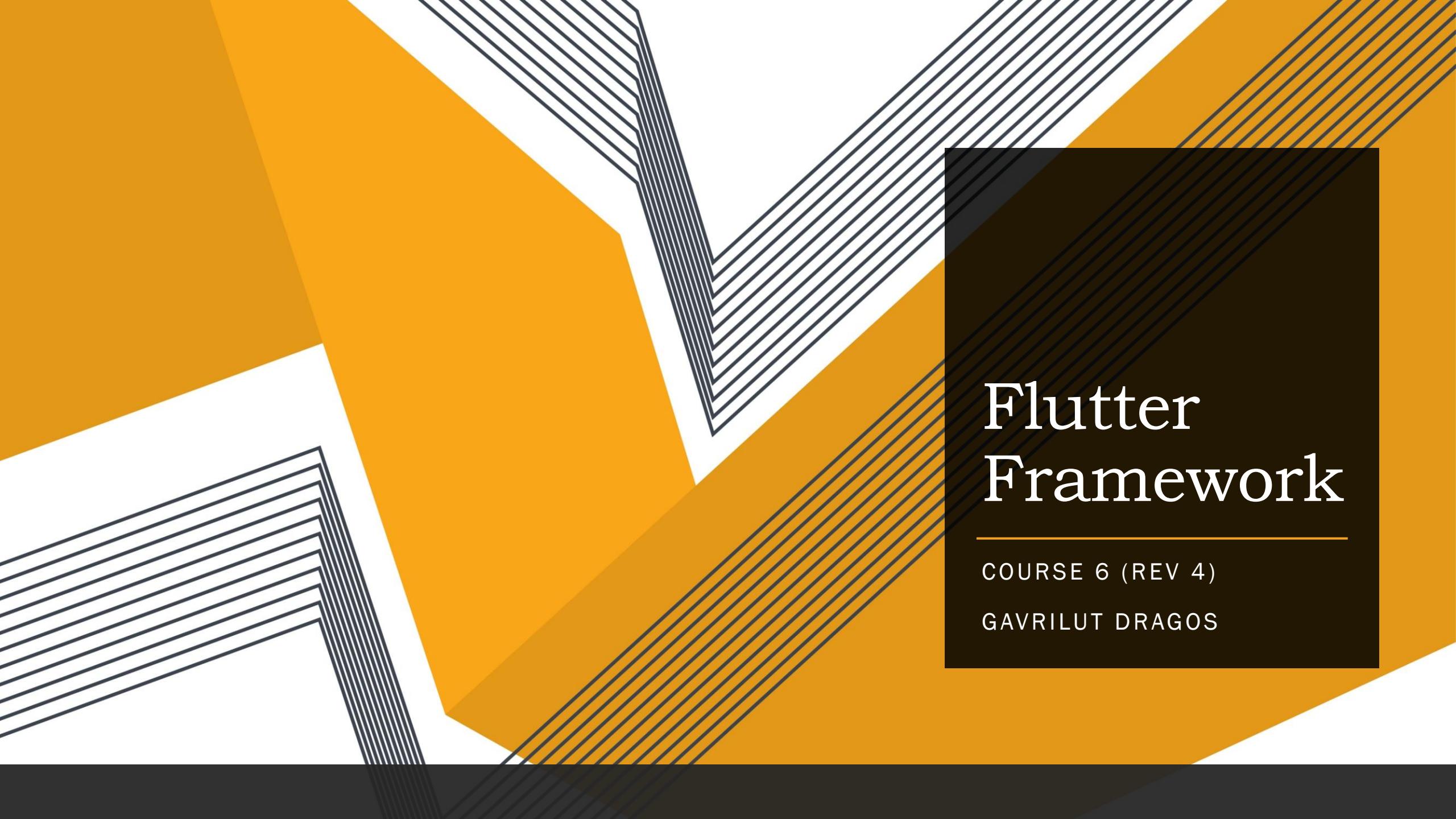
You can also change the font, size, color and a bunch of other font/text properties via style parameter.

```
Widget GetBody() {  
    return Center(  
        child: SizedBox(  
            width: 100,  
            height: 100,  
            child: TextFormField(  
                initialValue: "GDT",  
                style: TextStyle(color: Colors.red, fontSize: 30),  
            )));  
}
```



Q & A



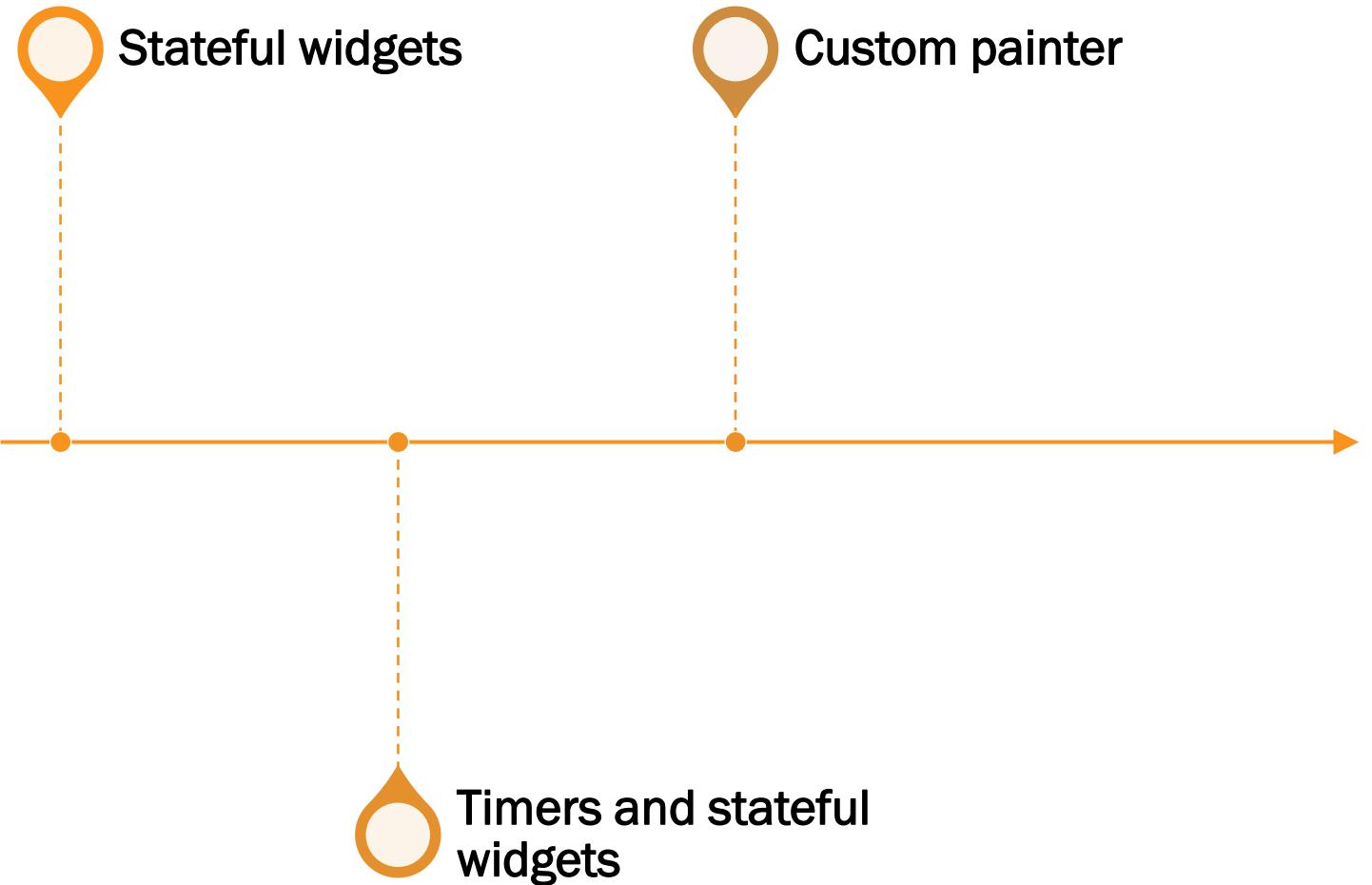


Flutter Framework

COURSE 6 (REV 4)

GAVRILUT DRAGOS

Agenda



Stateful widget

Stateful widget

Regular applications have data members / variable that often change as a result of the interaction with the user. Some of them have an UI reflection (meaning that the change of that specific variable should be reflected in a change in UI).

For example, → if we have a timer that is printed on the screen, whenever the timer changes, the new time has to be reprinted to reflect the change.

In Flutter, this behavior is called a stateful widget → or to simplify a widget that has to be redrawn when its internal state changes (e.g. something else to display).

Stateful widget

A stateful application has the following format:

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

void main() => runApp(MyApp()); ← We still need a main function to start the program
                                         (the actual app)

class MyApp extends StatefulWidget {
  @override
  State<MyApp> createState() => MyAppState();
}

class MyAppState extends State<MyApp> {
  // data members
  @override
  Widget build(BuildContext context) => MaterialApp(...);
}
```

Stateful widget

A stateful application has the following format:

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

void main() => runApp(MyApp());

class MyApp extends StatefulWidget {
  @override
  State<MyApp> createState() => MyAppState();
}

class MyAppState extends State<MyApp> {
  // data members
  @override
  Widget build(BuildContext context) => MaterialApp(...);
}
```

However, the app extends StatefulWidget (that allows the App to create a State object)

Stateful widget

A stateful application has the following format:

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

void main() => runApp(MyApp());

class MyApp extends StatefulWidget {
  @override
  State<MyApp> createState() => MyAppState();
}

class MyAppState extends State<MyApp> {
  // data members
  @override
  Widget build(BuildContext context) => MaterialApp(...);
}
```

The state object is created via the `createState` method (that has to be overridden)

Stateful widget

A stateful application has the following format:

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

void main() => runApp(MyApp());

class MyApp extends StatefulWidget {
  @override
  State<MyApp> createState() => MyAppState();
}

class MyAppState extends State<MyApp> {
  // data members
  @override
  Widget build(BuildContext context) => MaterialApp(...);
}
```

The state object (in our case `MyAppState`) has a `build` method that will be called whenever the new Widget / view has to be updated/created.

Stateful widget

The “MyAppState” usually looks like this:

```
class MyAppState extends State<MyApp> {
    // some data member
    Widget GetBody() => ...

    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("<Title>")),
                body: GetBody()));
    }
}
```

This is where the data (the state values) are defined. They are usually regular variables / data members from this class.

Stateful widget

The “MyAppState” usually looks like this:

```
class MyAppState extends State<MyApp> {  
    // some data member  
  
    Widget GetBody() => ...  
  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            home: Scaffold(  
                appBar: AppBar(title: Text("<Title>")),  
                body: GetBody()));  
    }  
}
```

This is where the widgets and the interaction that leads to a change in the state happens.

Stateful widget

Let's build a very simple stateful app:

1. Has a button, on the center of the screen that says “Random Values”
2. Whenever that button is being pressed, a new random value will be generated and the text of that button will be change to reflect that value.
3. We will create an internal data member (called “value”) to store the randomly generated value
4. The initial value of that field will be -1 and the generated values will be between 0 and 100

Stateful widget

The “MyAppState” usually looks like this:

```
class MyAppState extends State<MyApp> {  
    int value = -1;
```

First we need to create that state value

Stateful widget

The “MyAppState” usually looks like this:

```
class MyAppState extends State<MyApp> {  
    int value = -1;  
  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            home: Scaffold(  
                appBar: AppBar(title: Text("Test")),  
                body: GetBody()));  
    }  
}
```

Then we need to add the **build** method

Stateful widget

The “MyAppState” usually looks like this:

```
class MyAppState extends State<MyApp> {  
    int value = -1;  
  
    @override  
    Widget build(BuildContext context) => ...  
  
    Widget GetBody() {  
        return Center(  
            child: ElevatedButton(  
                child: Text(value == -1 ? "Random" : "Value: ${value}"),  
                onPressed: onButtonPressed));  
    }  
}
```

After this we create the GetBody method
that creates an ElevatedButton that has a
text form out of either “Random” or
“Value:...”



Stateful widget

The “MyAppState” usually looks like this:

```
class MyAppState extends State<MyApp> {  
    int value = -1;  
  
    @override  
    Widget build(BuildContext context) {  
        Widget GetBody() {  
            return Center(  
                child: ElevatedButton(  
                    child: Text(value == -1 ? "Random" : "Value: ${value}"),  
                    onPressed: onButtonPressed));  
        }  
    }  
}
```

We also define a callback called `onButtonPressed` that should be called whenever that button is being pressed.

Stateful widget

The “MyAppState” usually looks like this:

```
class MyAppState extends State<MyApp> {  
    int value = -1;  
  
    @override  
    Widget build(BuildContext context) => ...  
  
    Widget GetBody() => ...  
  
    void onButtonPressed() {  
        setState(() {  
            value = Random().nextInt(100);  
        });  
    }  
}
```

Then we need to add the **build** method

Stateful widget

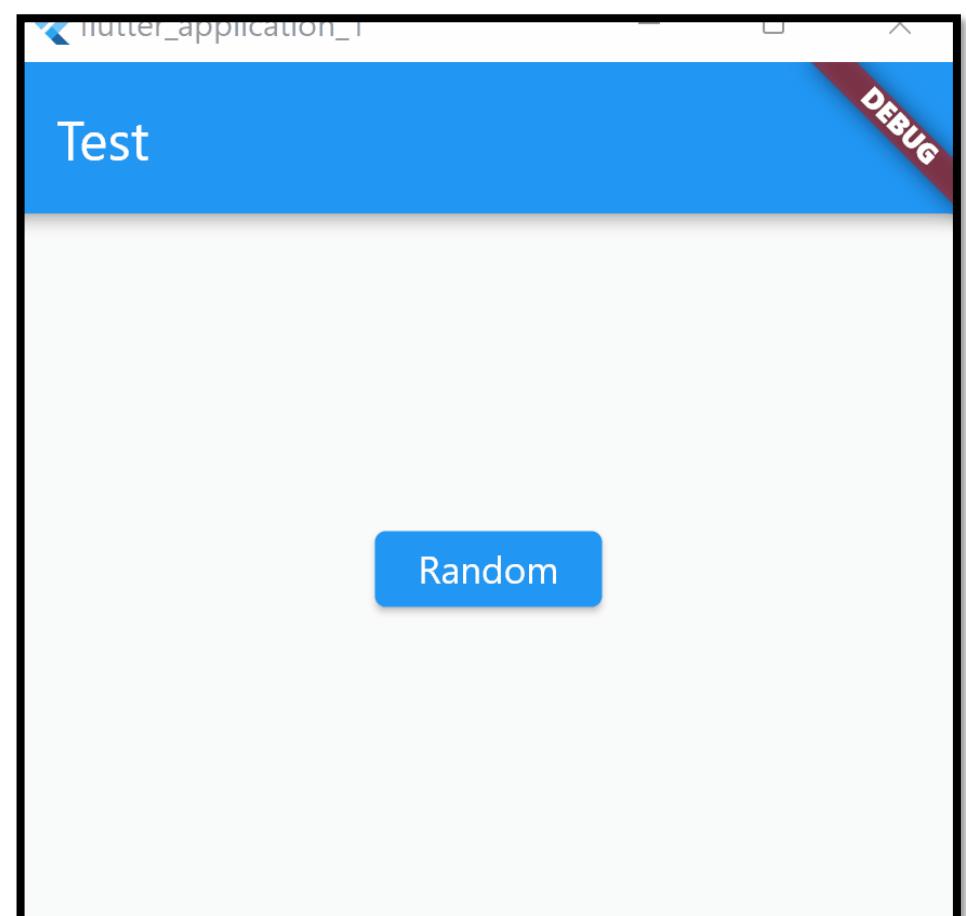
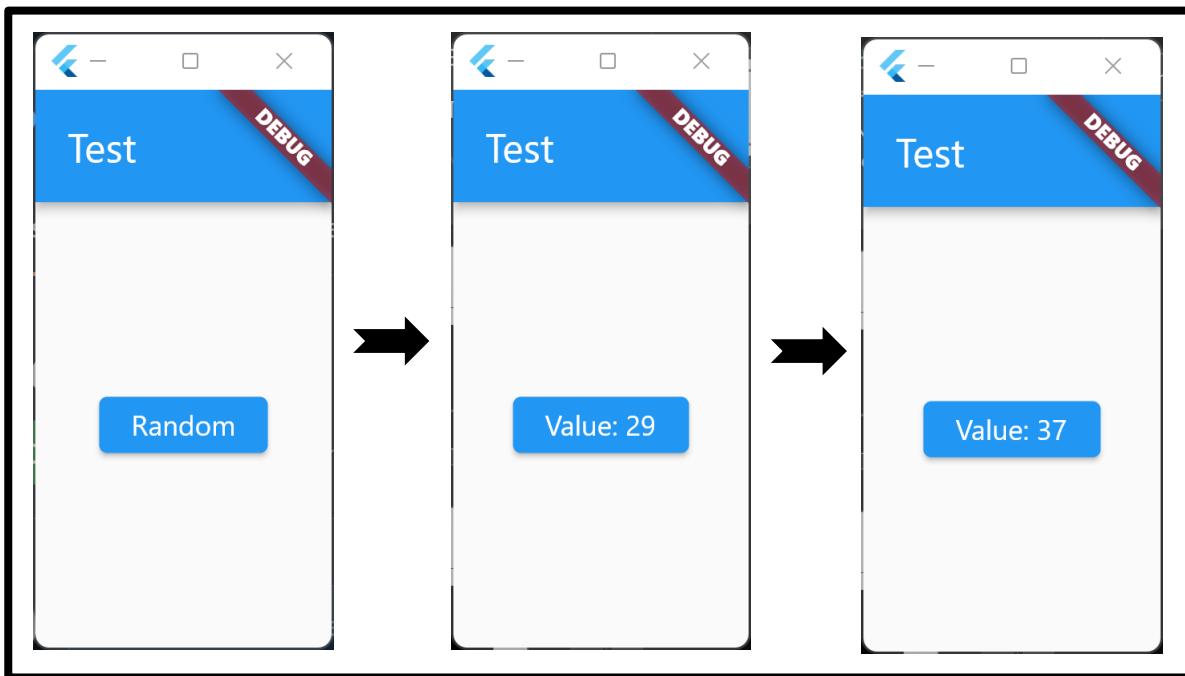
The “MyAppState” usually looks like this:

```
class MyAppState extends State<MyApp> {  
    int value = -1;  
  
    @override  
    Widget build(BuildContext context) => ...  
  
    Widget GetBody() => ...  
  
    void onButtonPressed() {  
        setState(() {  
            value = Random().ne...  
        });  
    }  
}
```

It's very important to change the value of the state within a `setState(...)` call. This makes sure that `.build` method is called after the value is changed, and as a result the text from the button will be changed as well.

Stateful widget

Upon execution, the app should behave like in the following images:



Using timers
with stateful
widgeds

Stateful widget & Timers

A stateful widget can be used with a Timer object to create an animation logic (a loop that will be called at a specific period of time).

A very simple example will increase a value and print it every “x” seconds

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

void main() => runApp(MyApp());

class MyApp extends StatefulWidget {
  @override
  State<MyApp> createState() => MyAppState();
}
class MyAppState extends State<MyApp> {
...
}
```

Let's consider this a template
for a very simple program.

Stateful widget & Timers

A stateful widget can be used with a Timer object to create an animation logic (a loop that will be called at a specific period of time).

A very simple example will increase a value and print it every “x” seconds

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

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

class MyApp extends StatelessWidget {
  @override
  State<MyApp> createState() => MyAppState();
}

class MyAppState extends State<MyApp> {
  ...
}
```

This means that in the next slides
we will focus on the **MyAppState** class

Let's consider this a template
for a very simple program.

Stateful widget & Timers

```
class MyAppState extends State<MyApp> {  
    int value = 0;
```

```
}
```

First, we need to create a value (the timer value) that will be increase every seconds.

Stateful widget & Timers

The app consists in a **Center** layout that contains a **Text** object with the value (“Value: \${value}”), practically showing the value of **value** data member

```
int value = 0,
```



```
@override  
Widget build(BuildContext context) {  
  return MaterialApp(  
    home: Scaffold(  
      appBar: AppBar(title: Text("Test")),  
      body: Center(child: Text("Value: ${value}"))));  
}
```

Next build method has to be created.

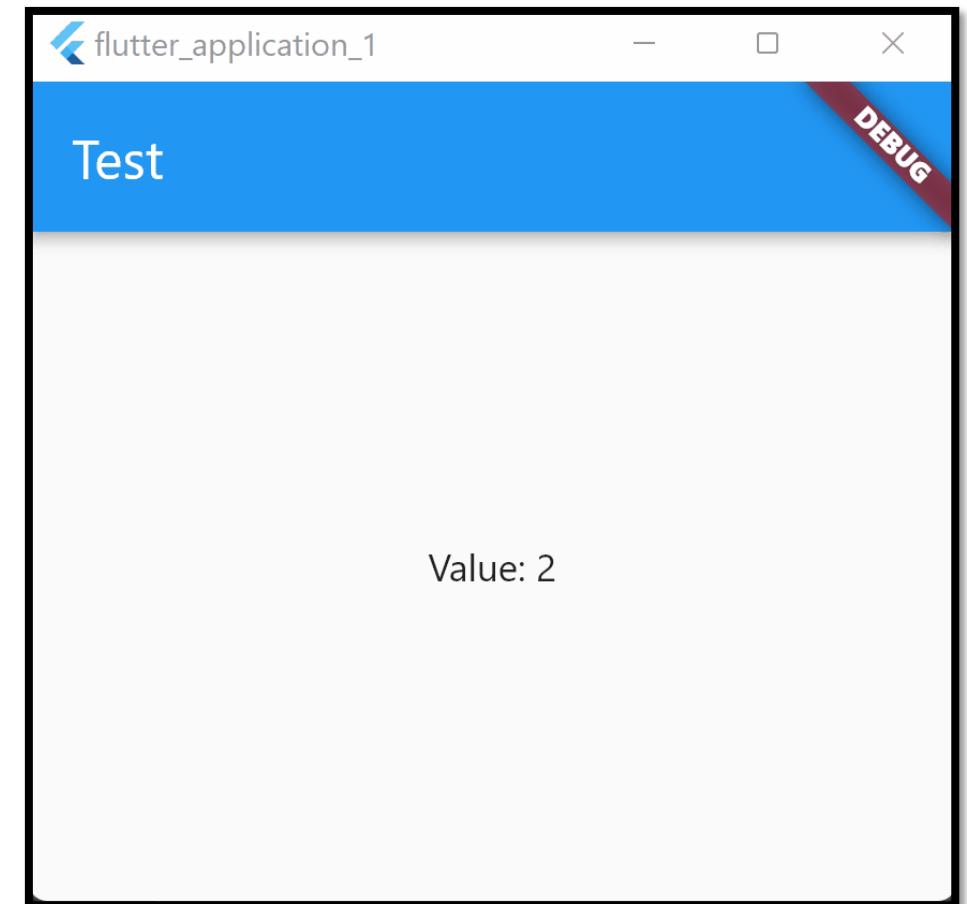
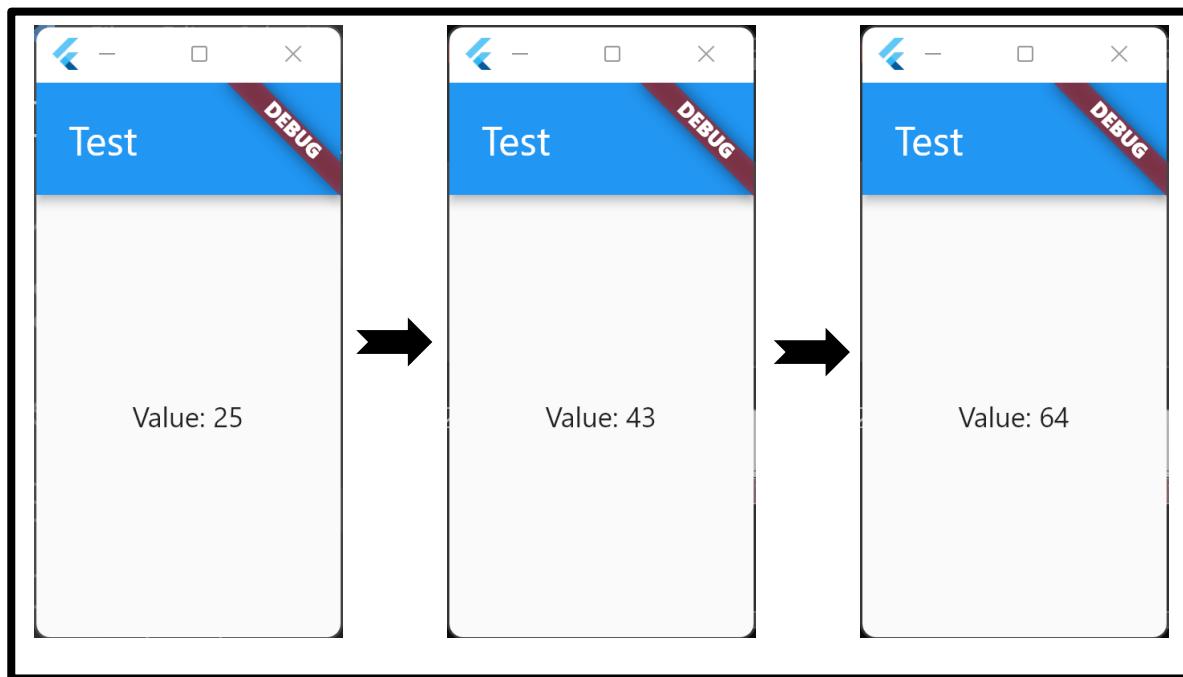
Stateful widget & Timers

```
class MyAppState extends State<MyApp> {  
    int value = 0;  
  
    @override  
    Widget build(BuildContext context) {...}  
  
    MyAppState() {  
        Timer.periodic(  
            Duration(seconds: 1),  
            (t) => setState(() {  
                value++;  
            }));  
    }  
}
```

Finally, the constructor creates a Timer object that will be triggered every one second, and uses **setState** to increment the value.

Stateful widget & Timers

Upon execution, the app should behave like in the following images:



Custom paint

Custom paint

Every UX has to have a component that can be used for custom drawing. In Flutter this is materialized through the `CustomPaint` widget that is usually organized in the following way:

Custom paint

Every UX has to have a component that can be used for custom drawing. In Flutter this is materialized through the `CustomPaint` widget that is usually organized in the following way:

```
class MyAppState extends State<MyApp> {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar(title: Text("<Title>")),  
        body: CustomPaint(painter: <painter>,  
                           child: Container()))),  
  }  
}
```

A *painter* parameter must be provided
(this will include the actual custom
drawing).

Custom paint

Every UX has to have a component that can be used for custom drawing. In Flutter this is materialized through the `CustomPaint` widget that is usually organized in the following way:

```
class MyAppState extends State<MyApp> {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar(title: Text("<Title>")),  
        body: CustomPaint(painter: <painter>,  
                           child: Container()))  
  }  
}
```

A `child` parameter will reflect the widget over which the drawing will happen.

Custom paint

The painter object has to be derived from `CustomPainter` (not to be confused with `CustomPaint` widget) where two methods have to be overridden:

```
class <my_painter> extends CustomPainter {  
  @override  
  void paint(Canvas canvas, Size size) { ... }  
  
  @override  
  bool shouldRepaint(CustomPainter oldDelegate) { ... }  
}
```

Custom paint

The painter object has to be derived from `CustomPainter` (not to be confused with `CustomPaint` widget) where two methods have to be overridden:

```
class <my_painter> extends CustomPainter {  
    @override  
    void paint(Canvas canvas, Size size) { ... }  
  
    @override  
    bool shouldRepaint(CustomPainter oldDelegate) { ... }  
}
```

This is the actual method where the custom drawing will happen

Custom paint

The painter object has to be derived from `CustomPainter` (not to be confused with `CustomPaint` widget) where two methods have to be overridden:

```
class <my_painter> extends CustomPainter {  
    @override  
    void paint(Canvas canvas, Size size) { ... }  
  
    @override  
    bool shouldRepaint(CustomPainter oldDelegate) { ... }  
}
```

This is called whenever a new instance of the class is created. The idea is to check if a repaint should be performed or not.

Custom paint

The painter object has to be derived from `CustomPainter` (not to be confused with `CustomPaint` widget) where two methods have to be overridden:

```
class <my_painter> extends CustomPainter {  
  @override  
  void paint(Canvas canvas, Size size) { ... }  
}
```

the `.paint` method has two parameters:

1. A canvas object that will be used for the actual drawing
2. The size of the space where the drawing will occur

Custom paint

A canvas is a generic interface for drawing. It is similar to the capabilities of a SVG format and it includes methods for:

- Primitives (drawing a line, a rectangle, a circle, etc)
- Paths
- Clipping support
- Scale / skew / various transformations
- Image manipulations

A complete list of all these methods can be found here:

<https://api.flutter.dev/flutter/dart-ui/Canvas-class.html>

Custom paint

Let's put all of these together and build an app that uses custom paint to draw a smiling face 😊
First → we will use the previous template for the basic app, and we will focus on the **MyAppState** class.

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

void main() => runApp(MyApp());

class MyApp extends StatefulWidget {
  @override
  State<MyApp> createState() => MyAppState();
}

class MyAppState extends State<MyApp> {
...
}
```

Custom paint

The `MyAppState` class creates the widget hierarchy. Its body contains of a `CustomPaint` widget that uses `MyPainter` class to draw a smiley face over a `Container` object. This means that the drawing will happen over the entire app space.

Custom paint

The MyPainter class overrides:

1. `paint` method
2. `shouldRepaint` method

```
class MyPainter extends CustomPainter {  
  @override  
  void paint(Canvas canvas, Size size) {  
    ...  
  }  
  
  @override  
  bool shouldRepaint(CustomPainter oldDelegate) { return false; }  
}
```

This mean that the entire drawing logic should happen in the `.paint` method:

Custom paint

Let's start by creating a method (`DrawSmileyFace`) that will receive a coordinate at the screen and will draw a smiley face there. We will draw a smiley face at the center of the screen.

```
class MyPainter extends CustomPainter {  
    void DrawSmileyFace(Offset pos, Canvas canvas) { ... }  
  
    @override  
    void paint(Canvas canvas, Size size) {  
        Offset center = Offset(size.width / 2, size.height / 2);  
        DrawSmileyFace(center, canvas);  
    }  
  
    @override  
    bool shouldRepaint(CustomPainter oldDelegate) { return false; }  
}
```

Custom paint

Let's start by creating a method (`DrawSmileyFace`) that will receive a coordinate at the screen and will draw a smiley face there. We will draw a smiley face at the center of the screen.

```
class MyPainter extends CustomPainter {  
    void DrawSmileyFace(Offset pos, Canvas canvas) {  
        var paint = Paint()  
            ..color = Colors.yellow  
            ..style = PaintingStyle.fill;  
        canvas.drawCircle(pos, 100, paint);  
  
    }  
}
```

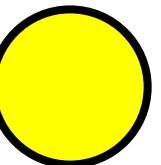


First, we will draw a yellow circle with the ray of 100. The paint object describe how painting has to be performed.

Custom paint

Let's start by creating a method (`DrawSmileyFace`) that will receive a coordinate at the screen and will draw a smiley face there. We will draw a smiley face at the center of the screen.

```
class MyPainter extends CustomPainter {  
    void DrawSmileyFace(Offset pos, Canvas canvas) {  
        var paint = Paint()...  
        canvas.drawCircle(pos, 100, paint);  
  
        paint  
            ..color = Colors.black  
            ..strokeCap = StrokeCap.round  
            ..strokeWidth = 5  
            ..style = PaintingStyle.stroke;  
        canvas.drawCircle(pos, 100, paint);  
    }  
}
```



Secondly, we will draw a black line circle of width 5 around the existing yellow circle.

Custom paint

Let's start by creating a method (`DrawSmileyFace`) that will receive a coordinate at the screen and will draw a smiley face there. We will draw a smiley face at the center of the screen.

```
class MyPainter extends CustomPainter {  
    void DrawSmileyFace(Offset pos, Canvas canvas) {  
        ...  
        paint  
            ..color = Colors.black  
            ..style = PaintingStyle.fill;  
        canvas.drawCircle(Offset(pos.dx - 33, pos.dy - 10), 20, paint);  
        canvas.drawCircle(Offset(pos.dx + 33, pos.dy - 10), 20, paint);  
    }  
}
```

Similarly, we draw the eyes



Custom paint

Let's start by creating a method (`DrawSmileyFace`) that will receive a coordinate at the screen and will draw a smiley face there. We will draw a smiley face at the center of the screen.

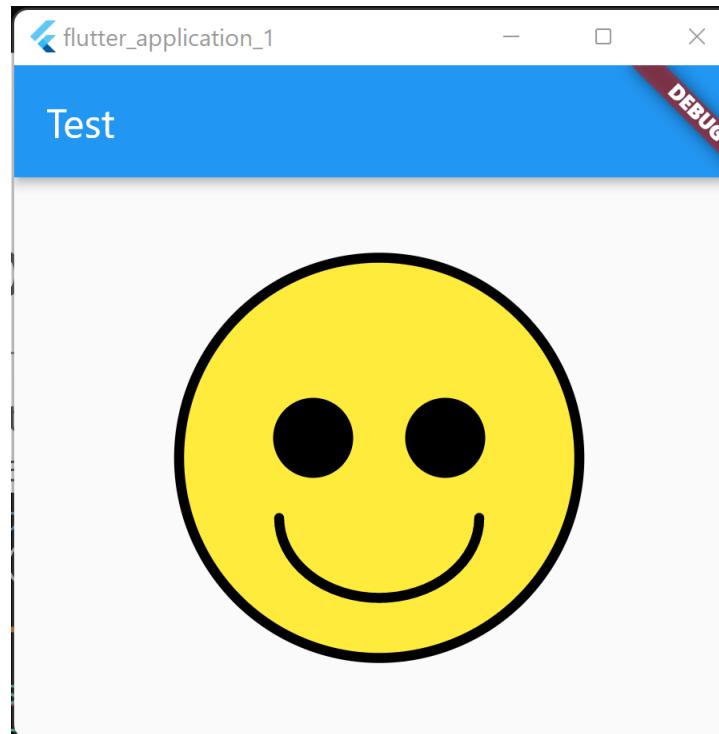
```
class MyPainter extends CustomPainter {  
    void DrawSmileyFace(Offset pos, Canvas canvas) {  
        ...  
        paint  
            ..color = Colors.black  
            ..strokeCap = StrokeCap.round  
            ..strokeWidth = 5  
            ..style = PaintingStyle.stroke;  
        canvas.drawArc(Rect.fromCenter(center: Offset(pos.dx, pos.dy + 30),  
                                     width: 100, height: 80), 3.14, -3.14, false, paint);  
    }  
}
```

Finally, we draw the mouth



Custom paint

Upon execution the image should look like the following one. Keep in mind the changed the parameters / distances can be used to change the way this image looks like.



Custom paint

What if we want to create a simple animation → for example to move a smiley face within the screen. First, we will change the **MyPainter** class in the following way:

```
class MyPainter extends CustomPainter {  
    double x, y;  
    MyPainter(this.x, this.y);  
    void DrawSmileyFace(Offset pos, Canvas canvas) {...}  
  
    @override  
    void paint(Canvas canvas, Size size) {  
        DrawSmileyFace(Offset(size.width * x, size.height * y), canvas);  
    }  
  
    @override  
    bool shouldRepaint(CustomPainter oldDelegate) => true;  
}
```

Custom paint

What if we want to create a simple animation → for example to move a smiley face within the screen. First, we will change the **MyPainter** class in the following way:

```
class MyPainter extends CustomPainter {  
    double x, y;  
    MyPainter(this.x, this.y);  
    void DrawSmileyFace(Offset pos, Canvas canvas) {...}  
  
    @override  
    void paint(Canvas canvas, Size size) {  
        DrawSmileyFace(Offset(size.width * x, size.height * y), canvas);  
    }  
  
    @override  
    bool shouldRepaint(CustomPainter oldDelegate) => true;  
}
```

MyPainter will now receive the (x,y) coordinates of the Smiley object as percentages (0 .. 1) [0 to 100%]

Custom paint

What if we want to create a simple animation → for example to move a smiley face within the screen. First, we will change the **MyPainter** class in the following way:

```
class MyPainter extends CustomPainter {  
    do  
    MyP  
    void drawSmileyFace(Canvas canvas, Offset offset) {  
        canvas.draw...  
  
        @override  
        void paint(Canvas canvas, Size size) {  
            DrawSmileyFace(Offset(size.width * x, size.height * y), canvas);  
        }  
  
        @override  
        bool shouldRepaint(CustomPainter oldDelegate) => true;  
    }  
}
```

The .paint method just calls **DrawSmileyFace** with coordinates based on “X” percentage and “y” percentage.



Custom paint

What if we want to create a simple animation → for example to move a smiley face within the screen. First, we will change the **MyPainter** class in the following way:

```
class MyPainter extends CustomPainter {  
    double x, y;  
    MyPainter(this.x, this.y);  
    void DrawSmileyFace(Offset pos, Canvas canvas) {...}  
  
    @override  
    Finally, the .shouldRepaint method will return true to tell the app that it  
    should redraw whenever a new MyPainter object is created.  
    , canvas);  
}  
  
@override  
bool shouldRepaint(CustomPainter oldDelegate) => true;  
}
```

Custom paint

Let's see how the state class should look like:

```
class MyAppState extends State<MyApp> {  
    double x = 0.3, y = 0.4;  
    double addX = 0.05, addY = 0.05;  
}  
}
```

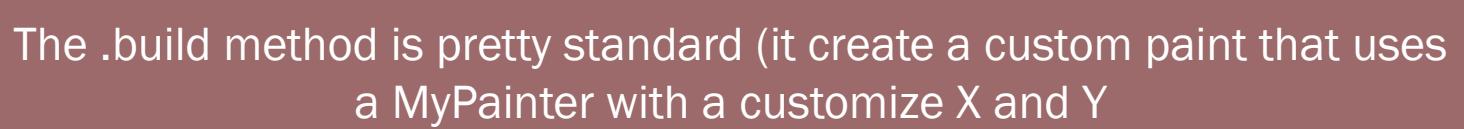
We define a start-up position (30% x, 40% y) and addX /addY with values to be added on X/Y axes.

Custom paint

Let's see how the

```
class MyAppState extends State<MyApp> {
    double x = 0.5, y = 0.4,
    double addX = 0.05, addY = 0.05;
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("Test")),
                body: CustomPaint(painter: MyPainter(x, y), child: Container())));
    }
}
```

The .build method is pretty standard (it creates a custom paint that uses a MyPainter with a customize X and Y)



}

Custom paint

Let's see how the state class should look like:

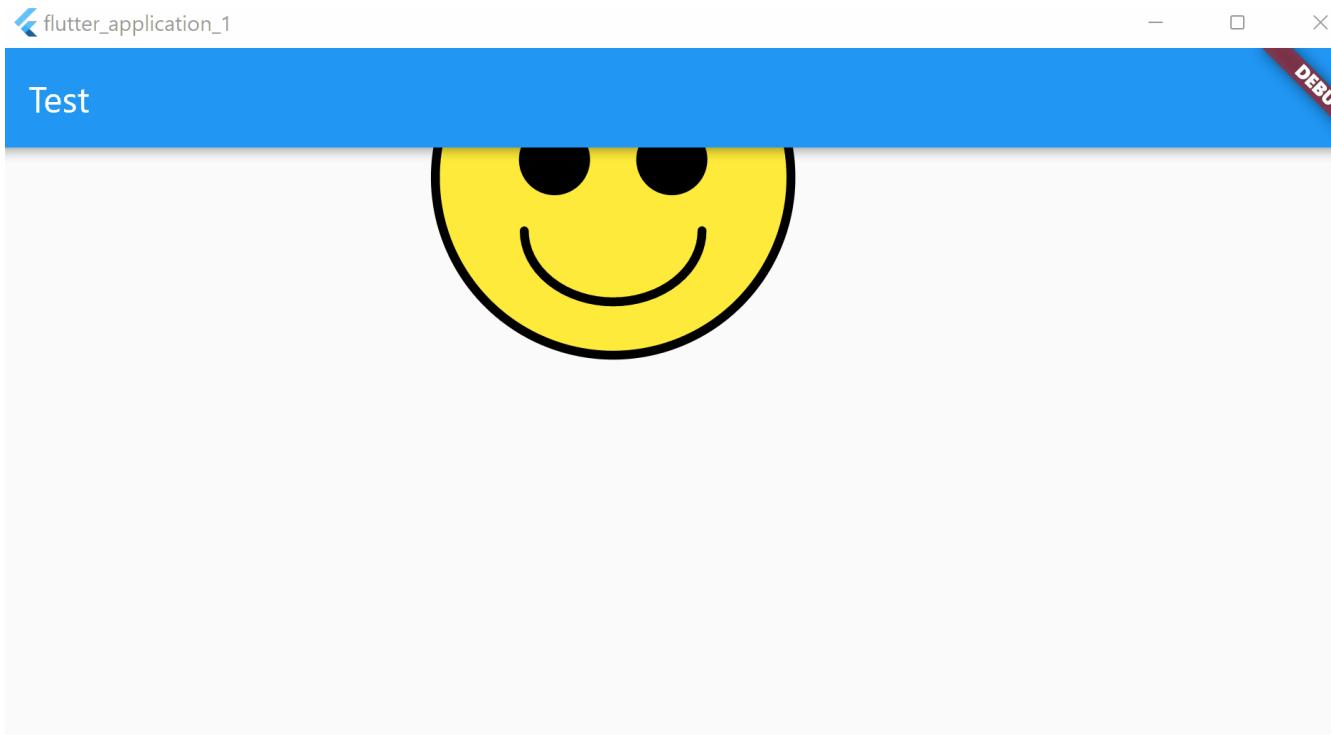
```
class MyAppState extends State<MyApp> {
    double x = 0;
    double y = 0;
    double addX = 1;
    double addY = 1;

    The constructor uses a timer object set up to be triggered every 100
    milliseconds that changes the X and Y with a very simple algorithm.

    @override
    Widget build(BuildContext context) {
        MyAppState() {
            Timer.periodic( Duration(milliseconds: 100), (t) => setState(() {
                x += addX;
                y += addY;
                if ((x >= 1) || (x <= 0)) addX = -addX;
                if ((y >= 1) || (y <= 0)) addY = -addY;
            }));
        }
    }
}
```

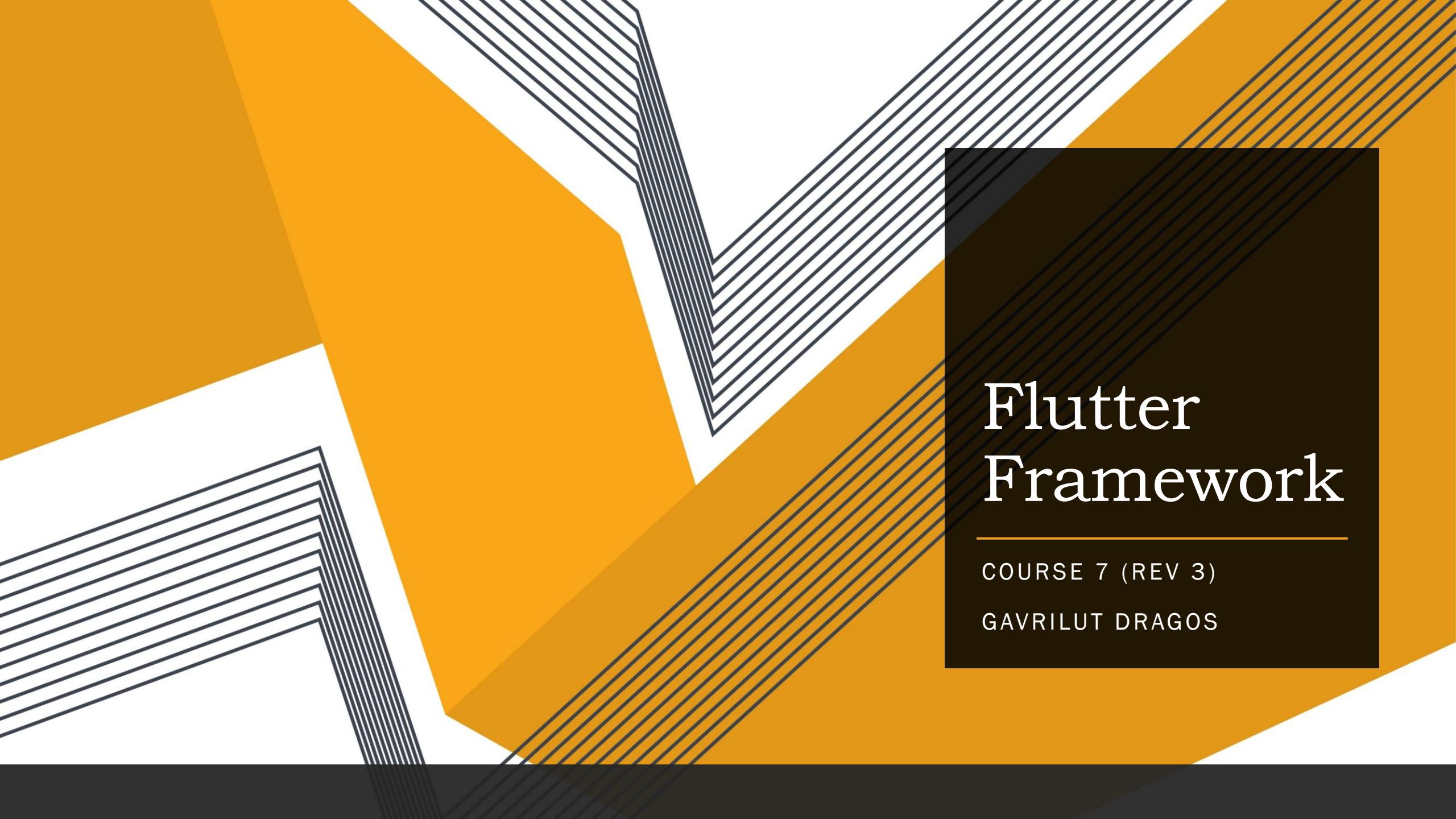
Custom paint

Upon execution the image should look like the following one with a smiley face moving.



Q & A



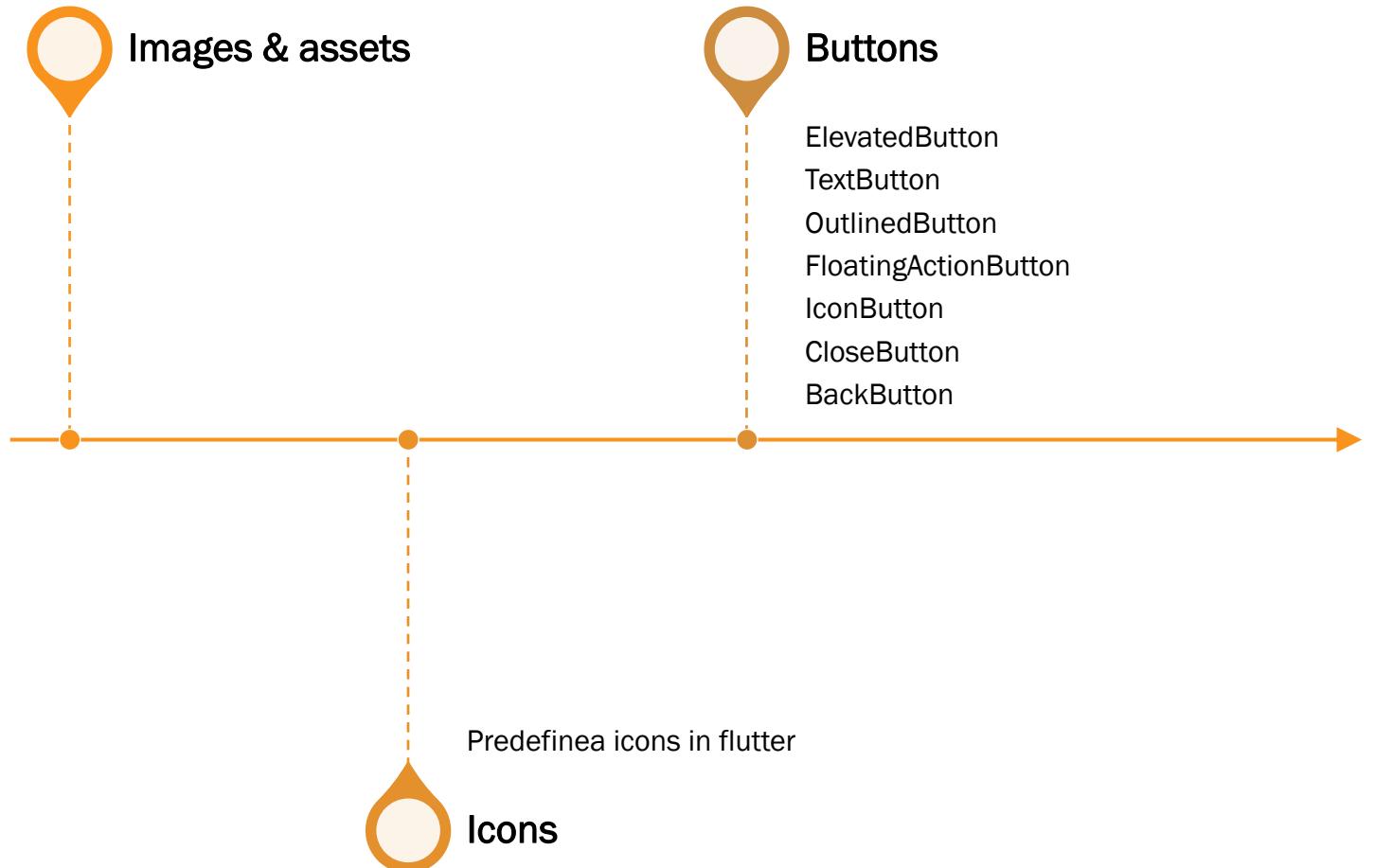


Flutter Framework

COURSE 7 (REV 3)

GAVRILUT DRAGOS

Agenda



Images

Images

Assets (and in particular images) are an integral part of any application or game.

In flutter, assets have to be defined in the `pubspec.yaml` file.

To add an image to your application, follow the next steps:

1. Create a folder (or a hierarchy of folders) into the root folder of your application. One common usage is to create a folder named “assets” and in that folder create another folder named “images” where all your images will be stored
2. Copy the images that you want to use in that folder.

Images

```
..  
.dart_tool  
.idea  
.vscode  
android  
assets  
build  
ios  
lib  
test  
web  
windows  
.gitignore  
.metadata  
.packages  
analysis_options.yaml  
flutter_application_1.iml  
pubspec.lock  
pubspec.yaml  
README.md
```

First, create a folder assets in the root of the application.

For simplicity, consider the root of the application
the folder where the file **pubspec.yaml** is located

Images

```
..  
.dart_tool  
.idea  
.vscode  
android  
assets  
build  
ios  
lib  
test  
web  
windows  
.gitignore  
.metadata  
.packages  
analysis_options.yaml  
flutter_application_1.iml  
pubspec.lock  
pubspec.yaml  
README.md
```

Create the following process/hierarchy:

- assets
- images

Copy all of the images you want to use in your project in this folder.

Lets consider that the following image **fii_logo.png** was copied !

Images

Once the images have been copied in their asset folder, modify the `pubspec.yaml` file in the following way:

```
flutter:  
  uses-material-design: true  
  assets:  
    - assets/images/fii_logo.png
```



If these items already exists just add the relative location of your images



It is also possible to add just the location of the folder (e.g. `assets/images`) if you want to include all existing images.

Images

After this modify the `main.dart` file from the application in the following way.

```
import 'package:flutter/material.dart';
void main() => runApp(MyApp());
class MyApp extends StatefulWidget {
  @override
  State<MyApp> createState() => MyAppState();
}
class MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp( home: Scaffold(
      appBar: AppBar(title: Text("Test")),
      body: Image.asset('assets/images/fii_logo.png')));
  }
}
```

Images

After this modify the `main.dart` file from the application in the following way.

```
import 'package:flutter/material.dart';
void main() => runApp(MyApp());
class MyApp extends StatefulWidget {
  @override
  State<MyApp> createState() => MyAppState();
}
class MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp( home: Scaffold(
      appBar: AppBar(title: Text("Test")),
      body: Image.asset('assets/images/fii_logo.png')));
  }
}
```

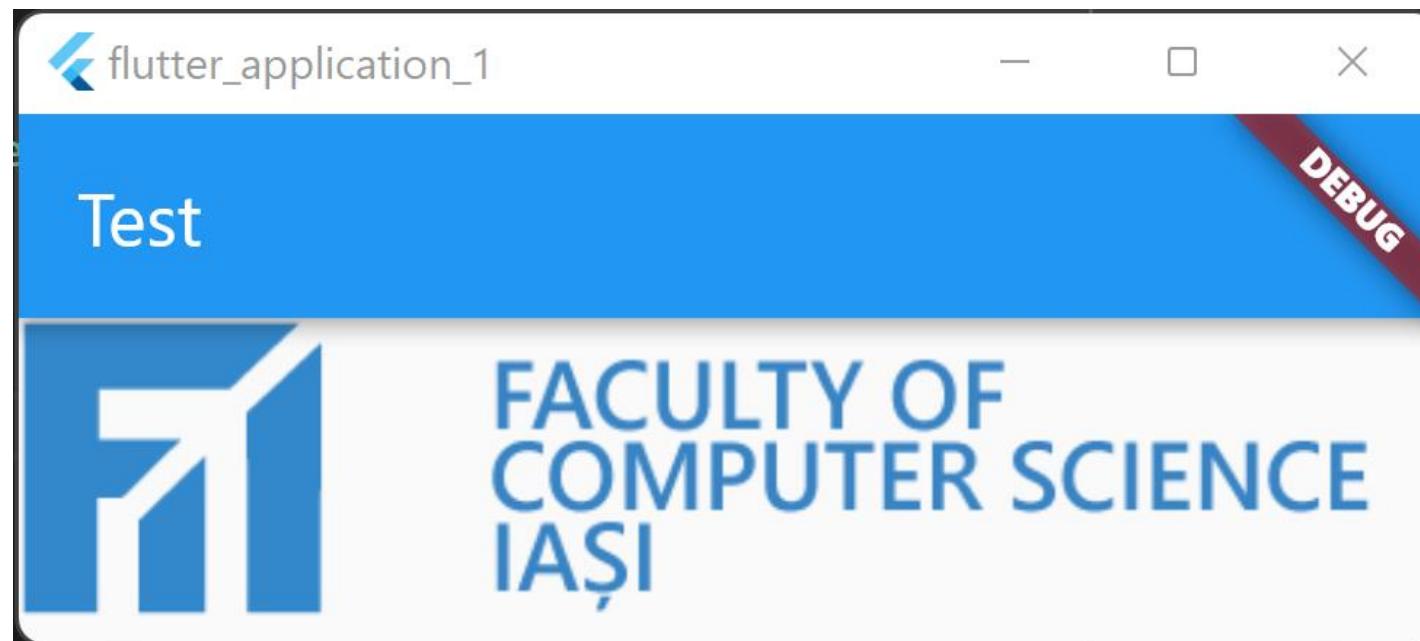
The widget `Image` loads an image from a stream / memory / file or asset. In case of assets, the relative path from the root of the application has to be provided.



```
body: Image.asset('assets/images/fii_logo.png')));
```

Images

Finally, when execution this application, you should see something that looks like the next picture.



Images

Image widget has four name constructors:

<code>Image.asset (String name, {...})</code>	A hash code for this object
<code>Image.file (File file, {...})</code>	Type of the object
<code>Image.memory (Uint8list bytes, {...})</code>	Call whenever a non-existing property is called
<code>Image.network (String uri, {...})</code>	A string representation for current object

The optional parameters include:

- Color
- Blending
- Scale

More information can be found on: <https://api.flutter.dev/flutter/widgets/Image-class.html>

Images

For example, in the previous example if we change the build method in the following way:

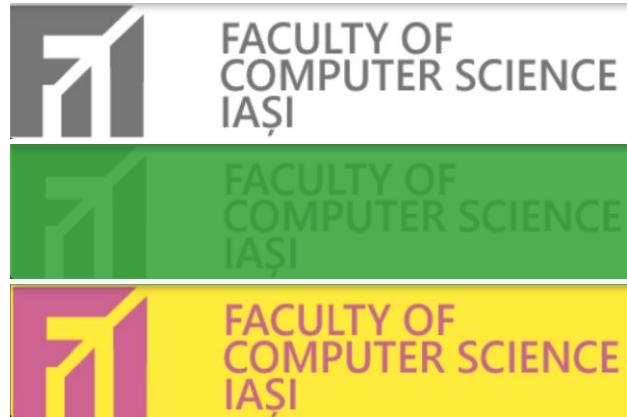
```
import 'package:flutter/material.dart';
void main() => runApp(MyApp());
class MyApp extends StatefulWidget { ... }
class MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("Test")),
        body: Image.asset('assets/images/fii logo.png',
          colorBlendMode: BlendMode.color, color: Colors.black)));
  }
}
```



Images

Other combinations:

- `colorBlendMode: BlendMode.color,`
`color: Colors.white`
- `colorBlendMode: BlendMode.color,`
`color: Colors.green`
- `colorBlendMode: BlendMode.difference,`
`color: Colors.red`

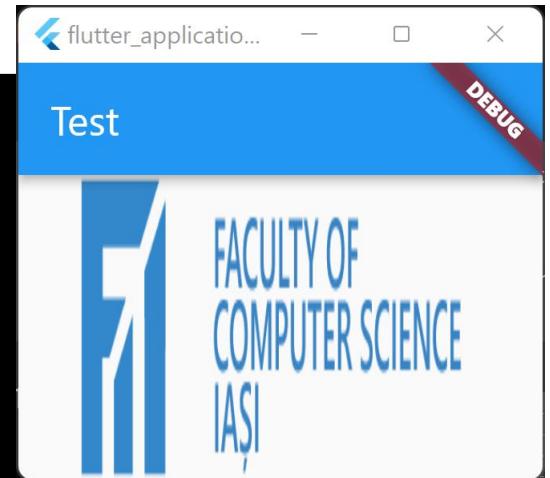


More details on how blend mode and color can be combined to change the way an image looks like can be found here: <https://api.flutter.dev/flutter/dart-ui/BlendMode.html>

Images

Another useful property is fit and can be used to stretch/skew and image:

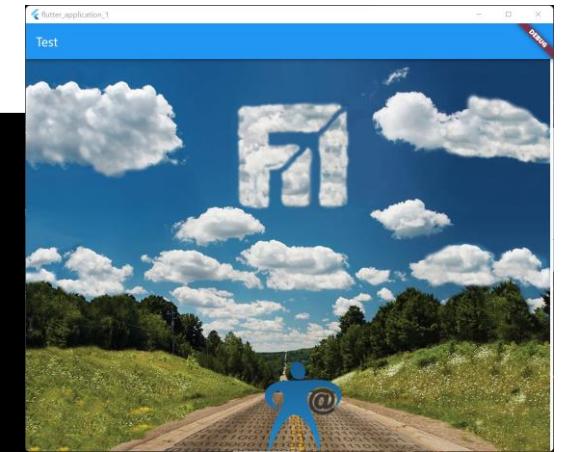
```
class MyAppState extends State<MyApp> {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar(title: Text("Test")),  
        body: Center(  
          child: Container(  
            child: Image.asset('assets/images/fii_logo.png',  
              fit: BoxFit.fill,),  
            width: 200,  
            height: 200))));  
  }  
}
```



Images

To load an image from an URI location use `Image.network` constructor:

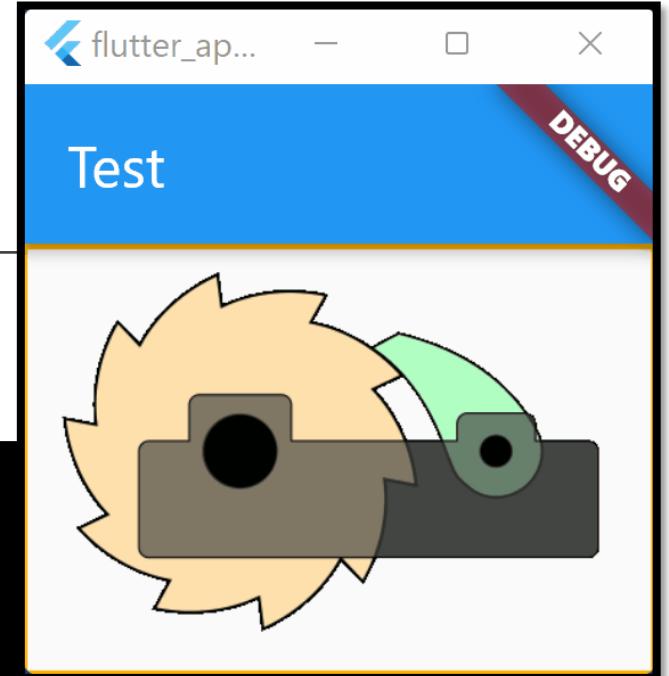
```
class MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("Test")),
        body: Image.network(
          'https://scontent.fias1-1.fna.fbcdn.net/v/t39.30808-
          6/278101604_4917352475029667_7419691009543583291_n.jpg?_nc_cat=108&ccb=1-
          5&_nc_sid=730e14&_nc_ohc=SaGdsZnIGWcAX-BjoiR&_nc_ht=scontent.fias1-
          1.fna&oh=00_AT_B-qyyXlHvcKBGDvxVNWFaoIVI0xA2yTJle6F3DbthQ&oe=626CF29A'));
  }
}
```



Images

Image widget also works with animated gifs
(either from the network or from your assets):

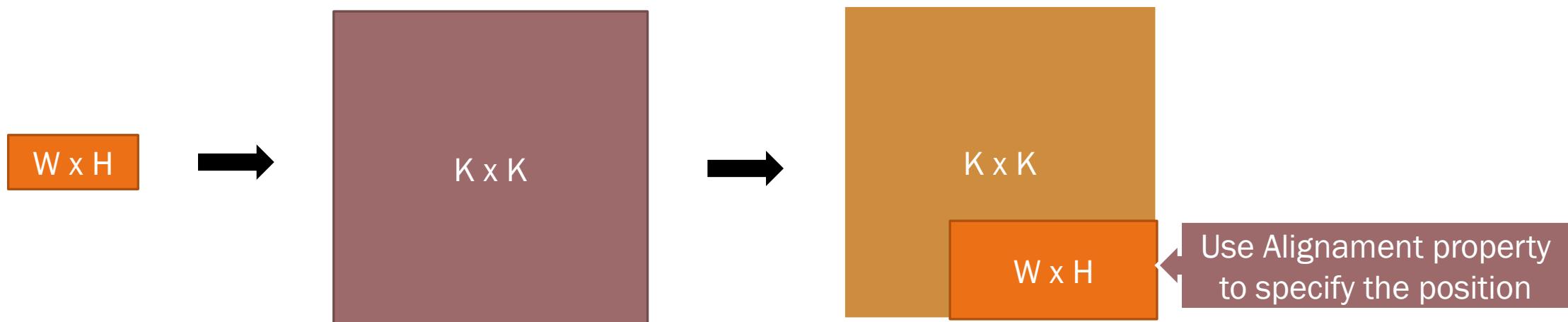
```
class MyAppState extends State<MyApp> {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar(title: Text("Test")),  
        body: Image.network(  
          'https://upload.wikimedia.org/wikipedia/commons/7/7e/Ratchet_Drawing_Animation.gif')));  
  }  
}
```



Images

Another interesting properties are width and height that can be used to set up a width and/or a height for an image. Its important to understand that the image will retain its width/height aspect ratio.

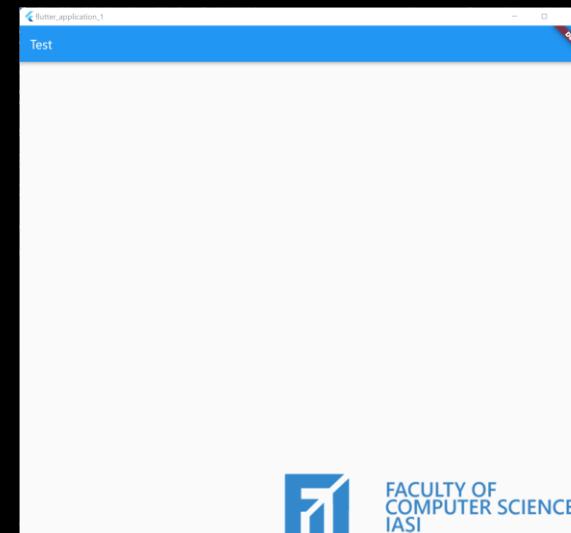
This means that in reality, only one size (either width or height) will be applied and the other one will be computed so that the aspect ratio is maintained.



Images

Using width and height:

```
class MyAppState extends State<MyApp> {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar(title: Text("Test")),  
        body: Image.asset(  
          'assets/images/fii_logo.png',  
          width: 1000,  
          height: 1000,  
          alignment: Alignment.bottomRight,  
        )),  
    );  
  }  
}
```

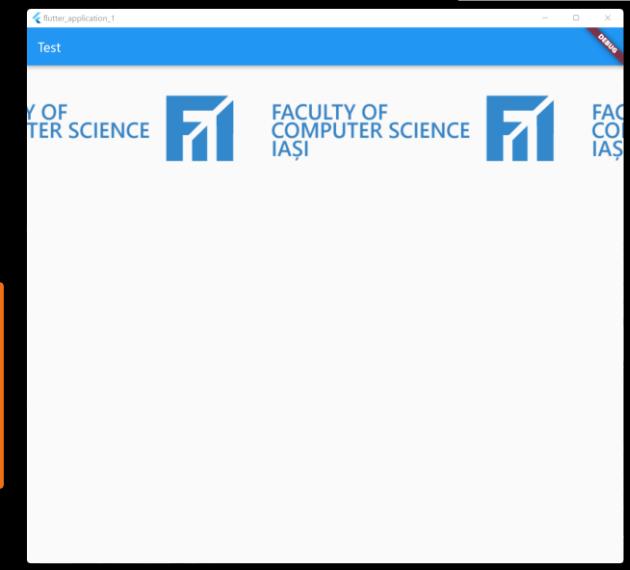


FACULTY OF
COMPUTER SCIENCE
IAŞI

Images

You can also use the repeat function to duplicate an image multiple times. Usually this option has to be combined with width or height property to increase the area where the image will be replicated. Possible values for repeat are `repeatX` or `repeatY` (to repeat an image on axex X or Y) or just simple `repeat` to repeat an image across the entire space (both X and Y axes).

```
class MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("Test")),
        body: Image.asset('assets/images/fii_logo.png',
          width: 1000, repeat: ImageRepeat.repeatX));
  }
}
```

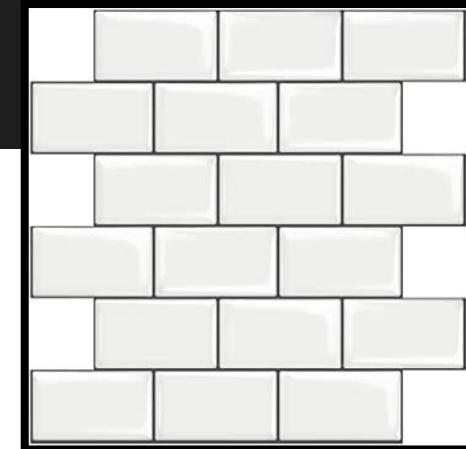


Images

The repeat property is often used with tiles (square images) that put together form a pattern of some sort. Let's consider that we add another image like this in our pubspec.yaml file (and let's consider that its name is tiles.jpg).

```
flutter:  
  uses-material-design: true  
  assets:  
    - assets/images/fii_logo.png  
    - assets/images/tiles.jpg
```

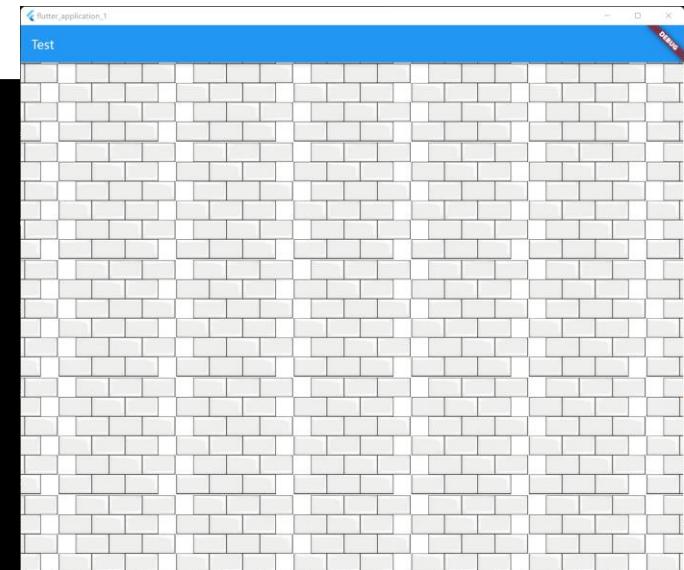
In this example we will use a brick-like image to simulate a wall.



Images

Using repeat property:

```
class MyAppState extends State<MyApp> {  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            home: Scaffold(  
                appBar: AppBar(title: Text("Test")),  
                body: Image.asset('assets/images/tiles.jpg',  
                    width: 1000,  
                    height: 1000,  
                    scale: 2,  
                    repeat: ImageRepeat.repeat)));  
    }  
}
```

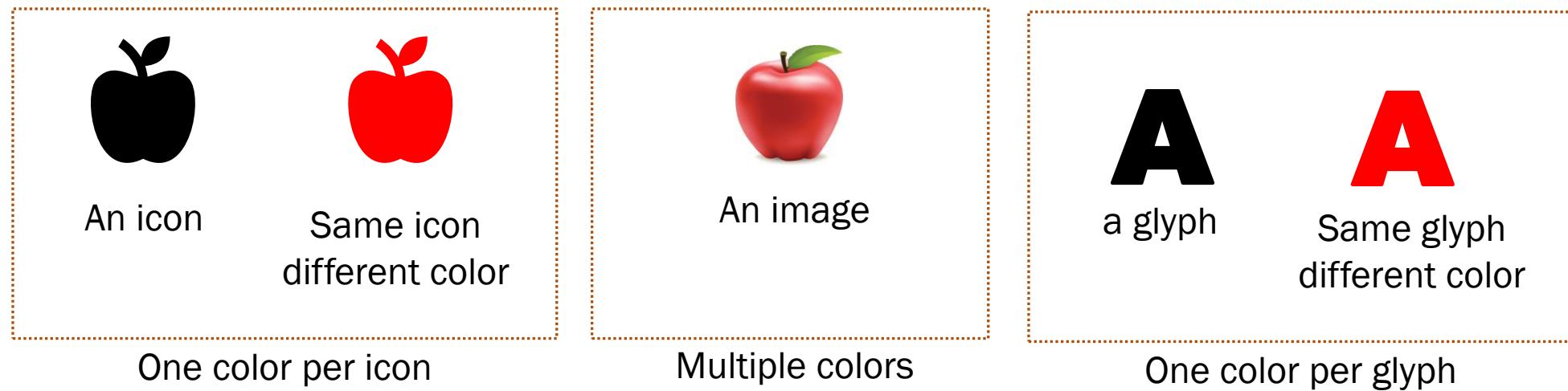


Icons

Icons

Icons are also an integral part of any application.

Icons (as a concept in Flutter) are glyphs (meaning that they are similar to a character representation from a font). In other words, an icon is a an image with only one color (that can be set up) and a transparent background.



Icons

The widget that draws an icon is called **Icon** and has the following constructor:

```
Icon (IconData? icon, {Key? key,  
                      double? size,  
                      Color? color,  
                      String? semanticLabel,  
                      TextDirection? textDirection})
```

Out of this:

- The `icon` parameter is one of `Icons.xxx` existing values
- The `size` parameter is the size of icon in logical pixels
- The `color` parameter allows one to change the color of the existing icon / glyph

Icons

Flutter has more than **8300** predefined icons that can be used (related to accessibility, operations, zooming, calls, messages, social, etc).

You can find a full list of predefined icons here:

<https://api.flutter.dev/flutter/material/Icons-class.html>

Some examples:

`alarm` → const IconData



– material icon named "alarm".

`backup_rounded` → const IconData



– material icon named "backup" (round).

`check_circle_rounded` → const IconData



– material icon named "check circle" (round).

`handyman_rounded` → const IconData



– material icon named "handyman" (round).

`notifications_sharp` → const IconData



– material icon named "notifications" (sharp).

`zoom_in_outlined` → const IconData

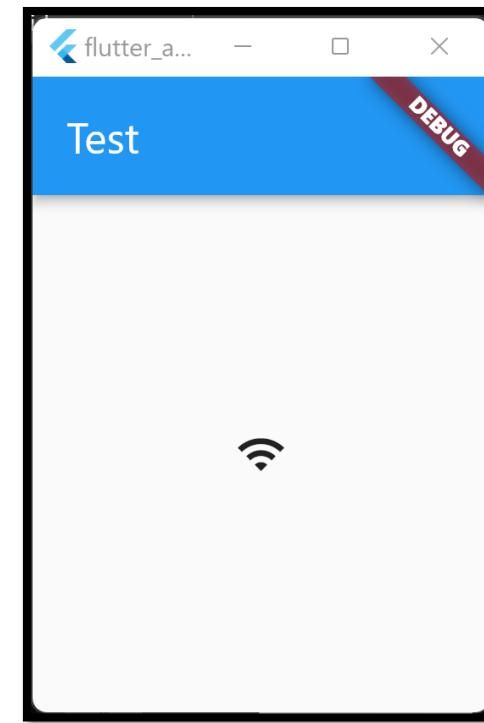


– material icon named "zoom in" (outlined).

Icons

The next example shows the icon for `wifi` (`Icons.wifi`) in the center of the app.

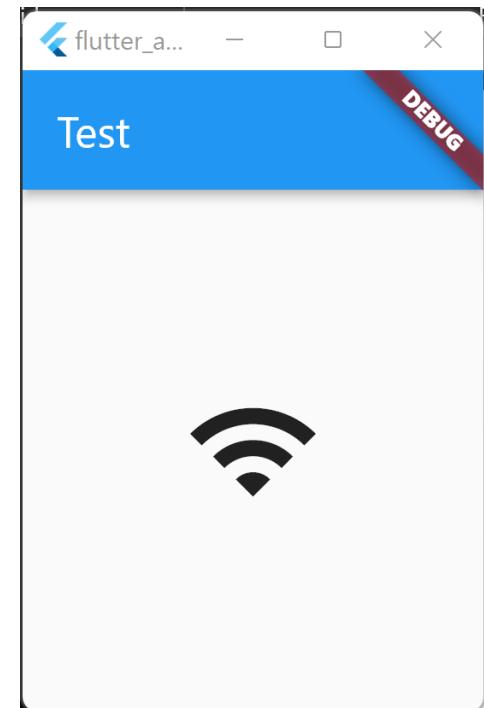
```
class MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("Test")),
        body: Center(child: Icon(Icons.wifi))),
    );
}
```



Icons

The next example shows the icon for `wifi` (`Icons.wifi`) in the center of the app, but sized to 64 logical pixels.

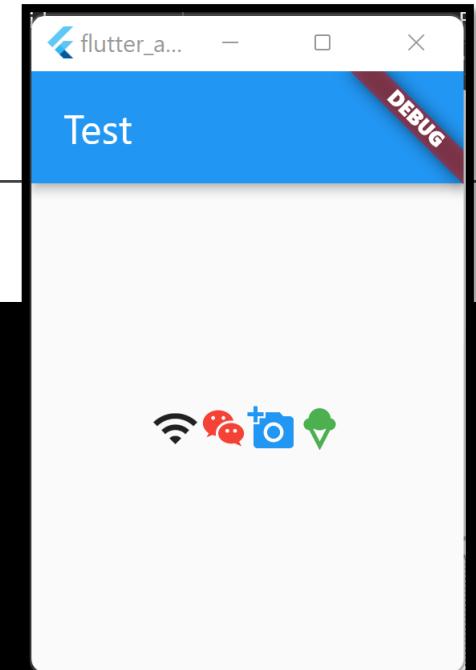
```
class MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("Test")),
        body: Center(child: Icon(Icons.wifi, size: 64)),
      );
    }
}
```



Icons

Multiple icons with different colors.

```
Widget build(BuildContext context) {
  return MaterialApp(
    home: Scaffold(
      appBar: AppBar(title: Text("Test")),
      body: Center(
        child: Row(
          children: [
            Icon(Icons.wifi),
            Icon(Icons.wechat, color: Colors.red),
            Icon(Icons.add_a_photo, color: Colors.blue),
            Icon(Icons.icecream, color: Colors.green)
          ],
          mainAxisSize: MainAxisSize.min,)));
}
```



Icons

For custom icons, Flutter provides two classes that can be used with various assets: `ImageIcon`

```
ImageIcon (ImageProvider<Object>? image, {Key? key,  
        double? size,  
        Color? color,  
        String? semanticLabel})
```

and `AssetImage`

```
AssetImage (String assetName, {AssetBundle? bundle, String? package})
```

The most common usage is to combine these two methods as follows:

```
ImageIcon (AssetImage ("<path to asset image>"))
```

Icons

Let's create a custom icon. We will use a PNG image, however for clarity it is recommended to copy these images in another folder (not images) → for example icons

1. First, we need to add an image to our assets folder (let's name it 'cat.png'). Make sure that the image has transparency.



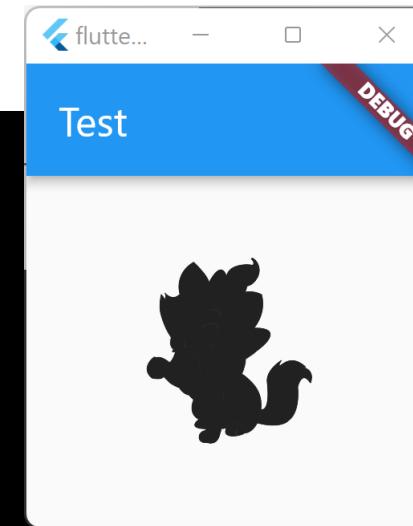
2. Second, add the new image to the `pubspec.yaml` file

```
flutter:  
  uses-material-design: true  
assets:  
  - assets/icons/cat.png
```

Icons

Now, let's draw a custom icon:

```
class MyAppState extends State<MyApp> {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar(title: Text("Test")),  
        body: Center(  
          child: ImageIcon(AssetImage("assets/icons/cat.png"),  
                        size: 96))));  
  }  
}
```



Icons

So why did the icon look like this:



and not like the actual image:



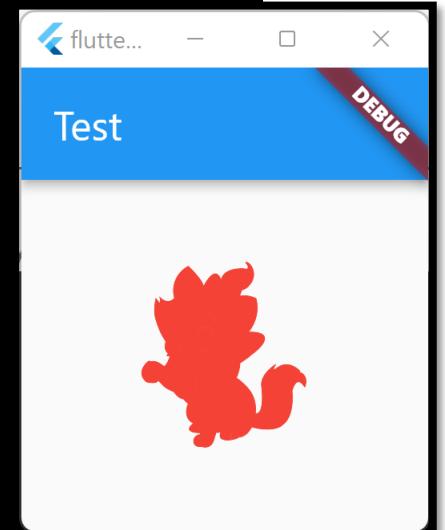
?

Well ... the reason is that we wanted to create an icon (a glyph) and not another image. As such **ImageIcon** object will convert the existing image into one color format image that can be used as a glyph. In our case, all pixels that were not transparent (pretty much the entire image of a cat) will have only one color (and appears as a shadow when painted).

Icons

Now that we know how custom icons can be created, we can apply other colors on them.

```
class MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("Test")),
        body: Center(
          child: ImageIcon(AssetImage("assets/icons/cat.png"),
            size: 96,
            color: Colors.red)));
  }
}
```

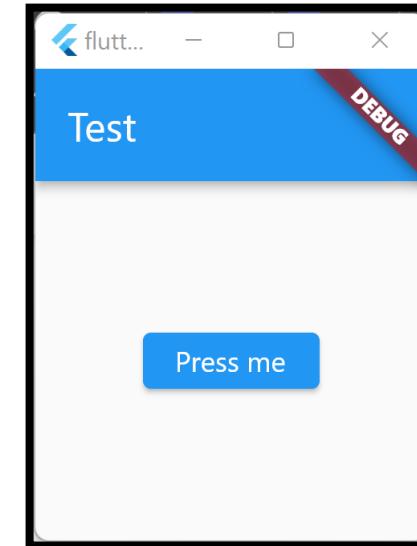


Buttons

Buttons

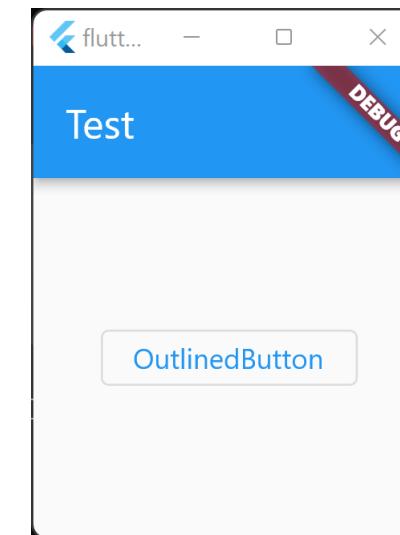
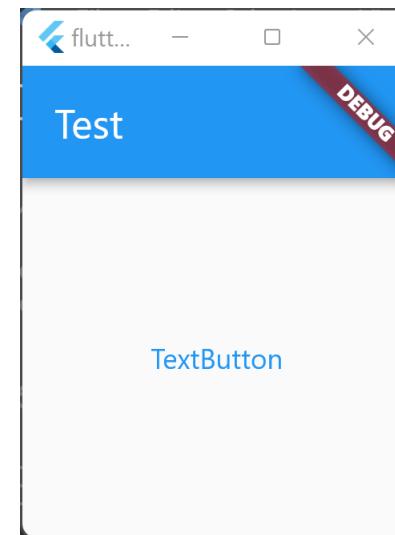
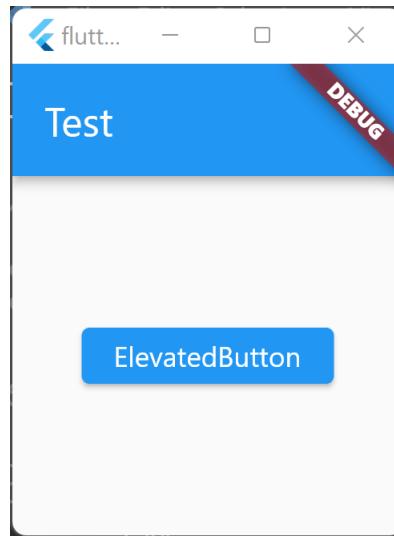
Buttons are an integrated part of any flutter application. There are 3 types of widgets defined in Flutter: **ElevatedButton**, **TextButton** and **OutlinedButton**

```
class MyAppState extends State<MyApp> {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar(title: Text("Test")),  
        body: Center(  
          child: ElevatedButton(  
            onPressed: () => {},  
            child: Text("Press me"))));  
  }  
}
```



Buttons

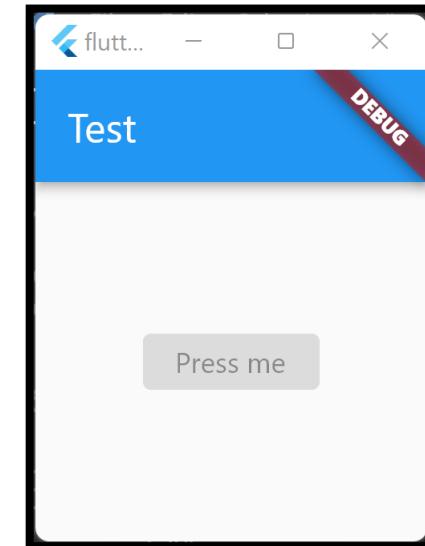
Buttons are an integrated part of any flutter application. There are 3 types of widgets defined in Flutter: **ElevatedButton**, **TextButton** and **OutlinedButton**. The main difference lies in how these type of Buttons are painted.



Buttons

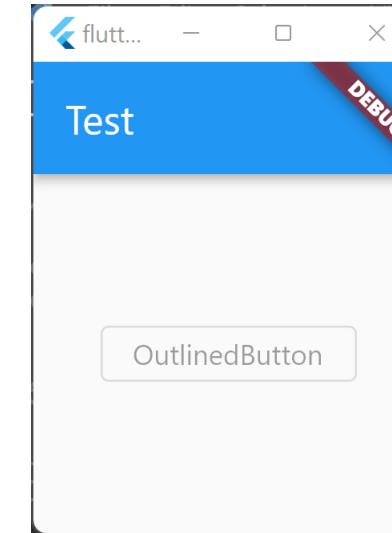
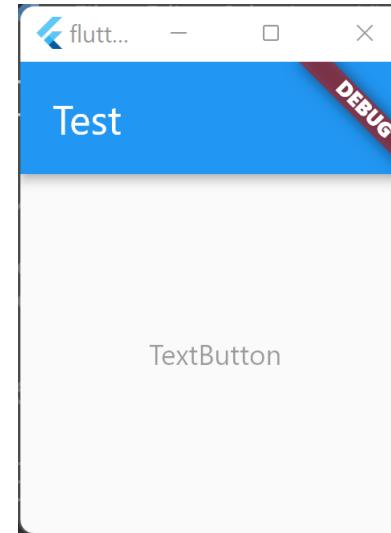
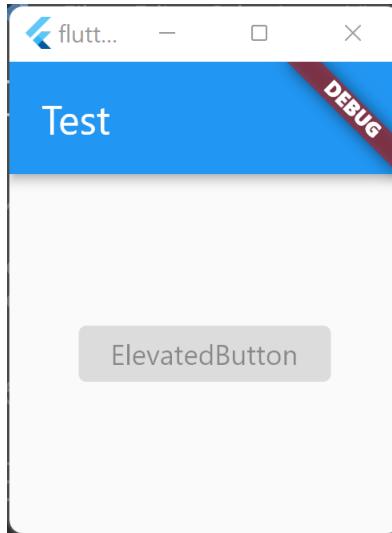
To disable a button , just set the `onPressed` property to null.

```
class MyAppState extends State<MyApp> {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar(title: Text("Test")),  
        body: Center(  
          child: ElevatedButton(  
            onPressed: null,  
            child: Text("Press me"))));  
  }  
}
```



Buttons

To disable a button , just set the `onPressed` property to null.



Buttons

What if we want to create a button that just shows an image.

1. First, we need to add an image to our assets folder (let's name it 'yes_logo.png'). Make sure that the image has transparency.



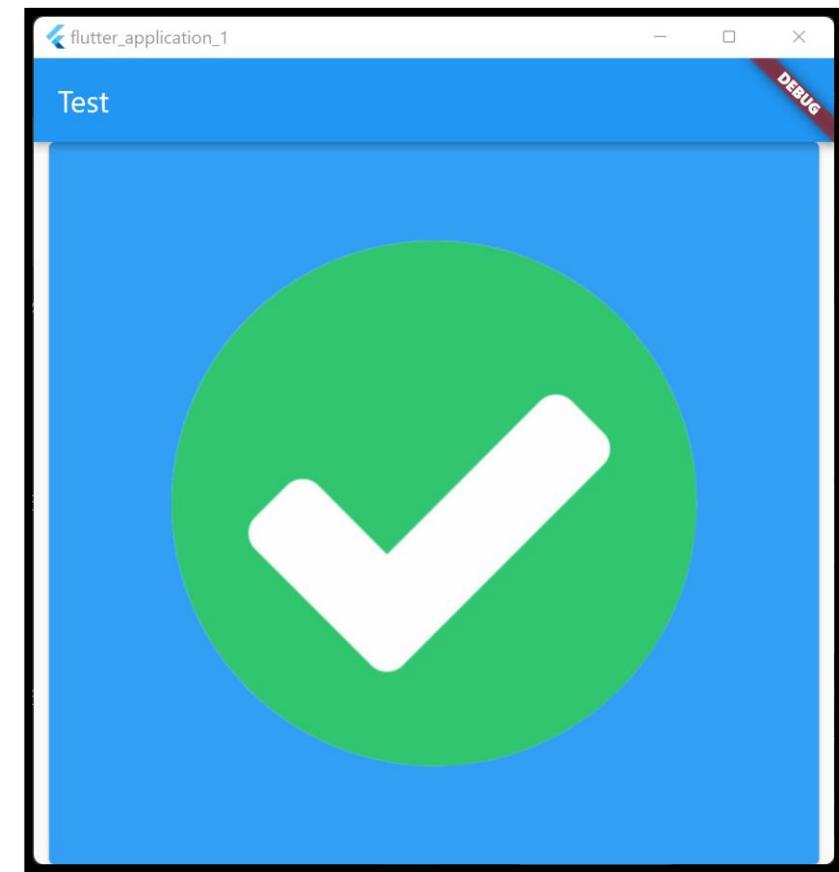
2. Second, add the new image to the `pubspec.yaml` file

```
flutter:  
  uses-material-design: true  
  assets:  
    - assets/images/yes_logo.png
```

Buttons

What if we want to create a button that just shows an image.

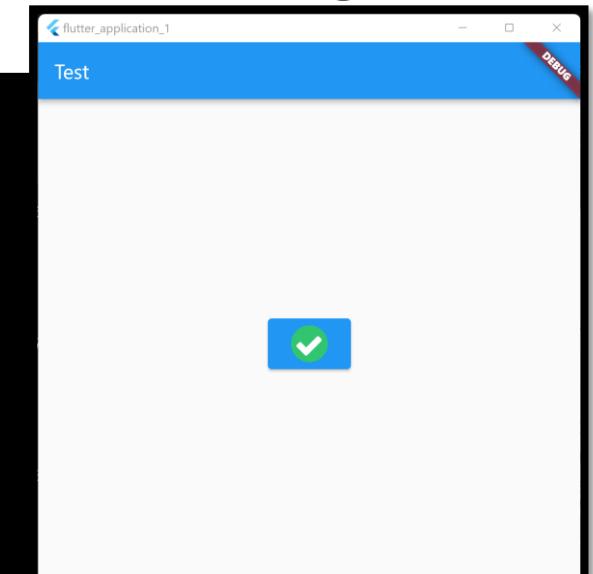
```
class MyAppState extends State<MyApp> {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar(title: Text("Test")),  
        body: Center(  
          child: ElevatedButton(  
            onPressed: () => {},  
            child:  
              Image.asset("assets/images/yes_logo.png"))));  
  }  
}
```



Buttons

In this particular case, the image size was 700 x 700 pixels and as such it increased the size of the button to almost the entire app screen. To solve this, we can use the `.width` and `.height` parameters in the `Image` constructor.

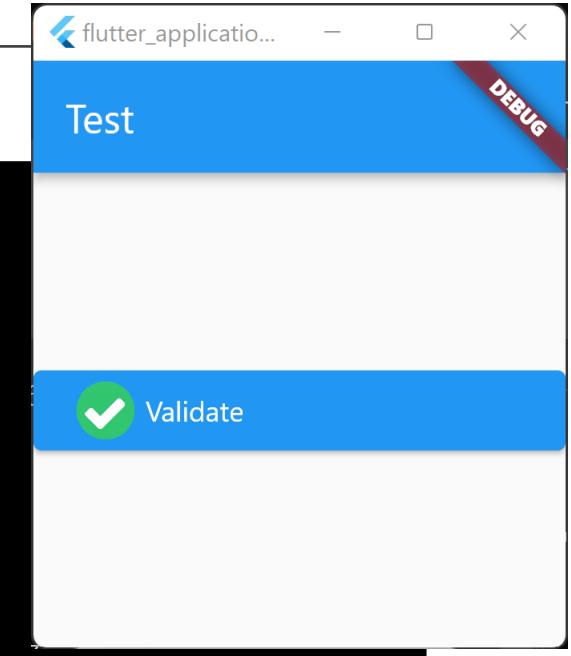
```
class MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("Test")),
        body: Center(
          child: ElevatedButton(
            onPressed: () => {},
            child: Image.asset("assets/images/yes_logo.png",
              height: 50, width: 50))));}
}
```



Buttons

But what if we want to add both an image and a text to a button ?

```
Widget build(BuildContext context) {  
    return MaterialApp(  
        home: Scaffold(  
            appBar: AppBar(title: Text("Test")),  
            body: Center(  
                child: ElevatedButton(  
                    onPressed: () => {},  
                    child: Row(children: [  
                        Image.asset("assets/images/yes_logo.png",  
                            height: 40, width: 40),  
                        Text("Validate")  
                    ]))),);  
}
```

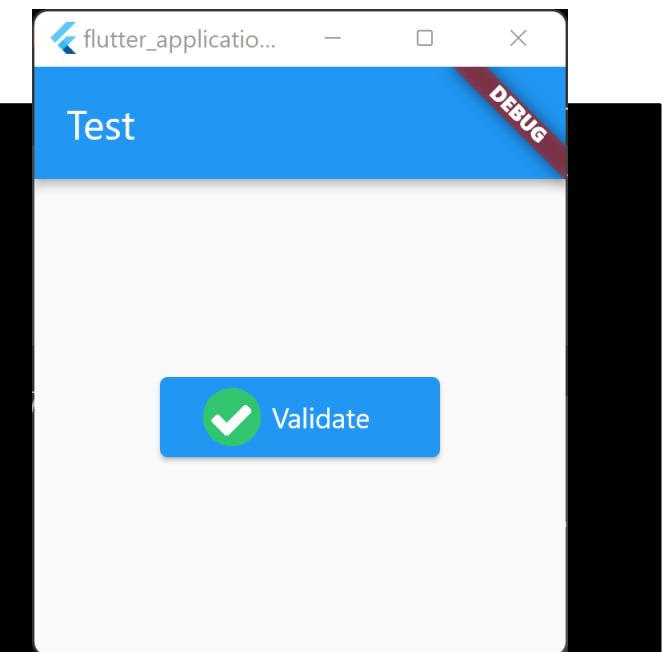


The main issue here is that the Row widget extends to the entire app space

Buttons

One way around the Row widget problem is to use a Container:

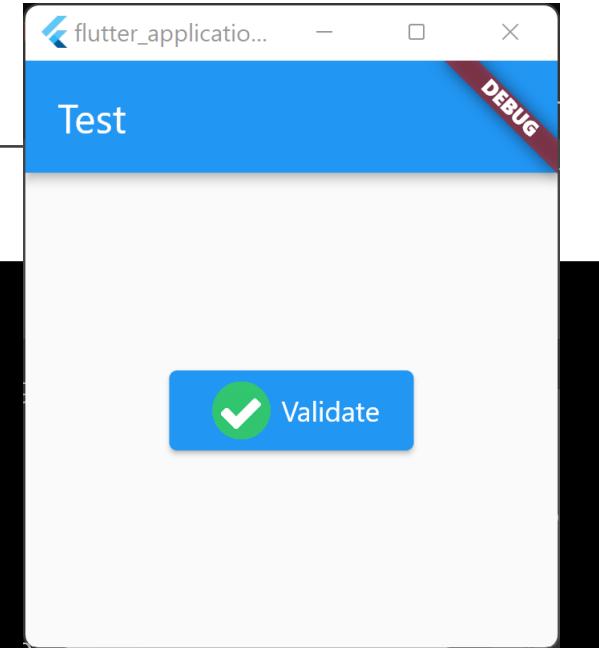
```
Widget build(BuildContext context) {
  return MaterialApp(
    home: Scaffold(
      appBar: AppBar(title: Text("Test")),
      body: Center(
        child: Container(
          width: 140,
          height: 40,
          child: ElevatedButton(
            onPressed: () => {},
            child: Row(children: [
              Image.asset("assets/images/yes_logo.png", height: 40, width: 40),
              Text("Validate")]))));
}
```



Buttons

Another solution is to set the property `mainAxisSize` to `MainAxisSize.min`

```
Widget build(BuildContext context) {
  return MaterialApp(
    home: Scaffold(
      appBar: AppBar(title: Text("Test")),
      body: Center(
        child: ElevatedButton(
          onPressed: () => {},
          child: Row(mainAxisSize: MainAxisSize.min,
                    children: [
                      Image.asset("assets/images/yes_logo.png",
                        height: 40, width: 40),
                      Text("Validate")
                    ])));
}
```



Buttons

However, this approach comes with other issues as well (for example, one needs to compute the size of the Container so that it fits the button).

Since this type of buttons (with an icon and a text) are very common, a named constructor (.icon) was added to all three forms of buttons to easily describe a button with an image and a text.

```
ElevatedButton.icon (required Widget icon, required Widget label, [...])
```

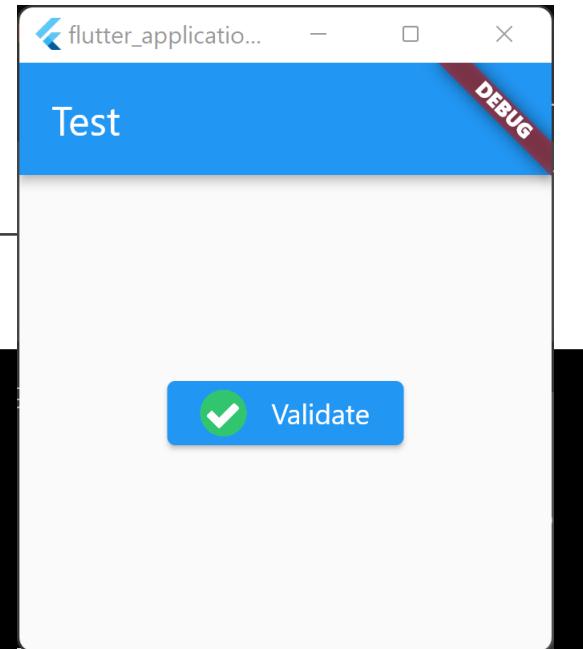
```
TextButton.icon (required Widget icon, required Widget label, [...])
```

```
OutlinedButton.icon (required Widget icon, required Widget label, [...])
```

Buttons

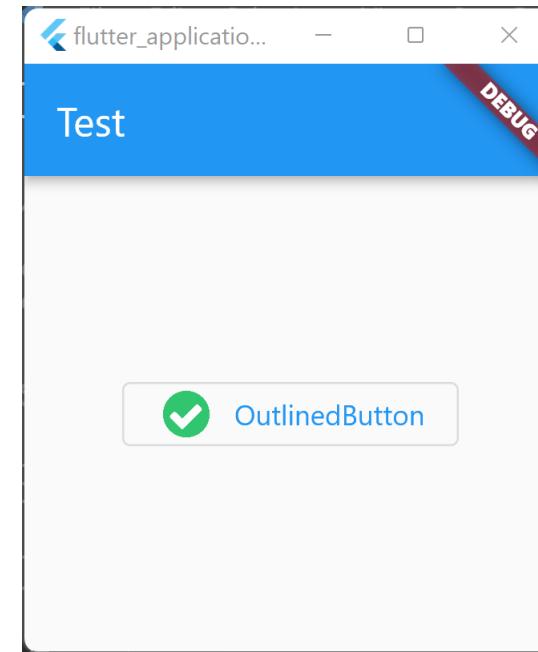
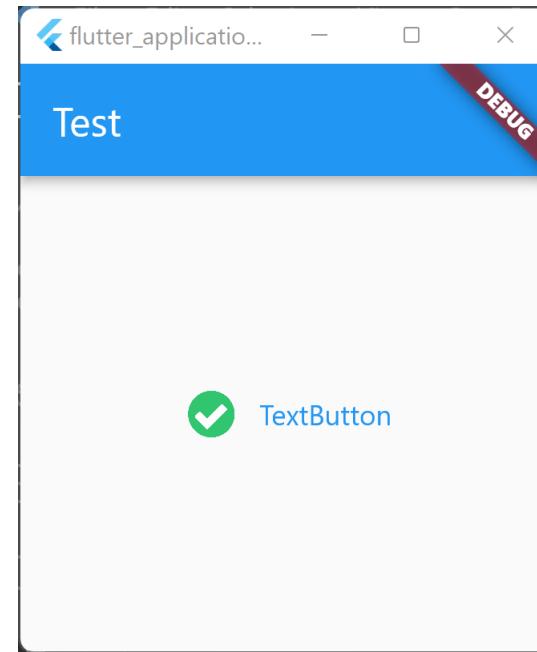
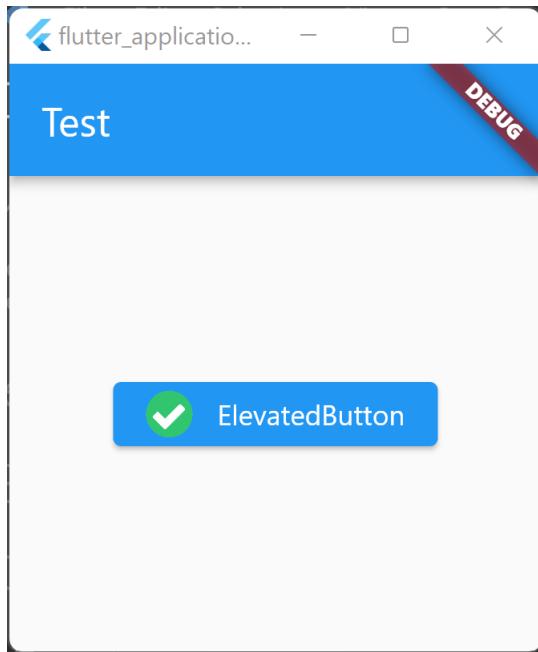
As such, the previous code will now look like this:

```
Widget build(BuildContext context) {
  return MaterialApp(
    home: Scaffold(
      appBar: AppBar(title: Text("Test")),
      body: Center(
        child: ElevatedButton.icon(
          onPressed: () => {},
          icon: Image.asset("assets/images/yes_logo.png",
            width: 32, height: 32),
          label: Text("Validate"),
        )));
}
```



Buttons

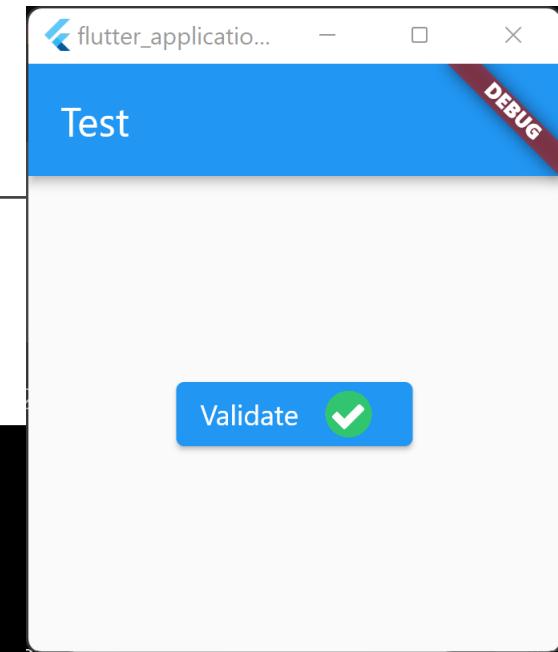
As such, the previous code will now look like this:



Buttons

Since both label and icon properties are in fact Widgets, one can replace them with any type of widget. One simple effect being that we can create a right-side icon with the text on the left side).

```
Widget build(BuildContext context) {
  return MaterialApp(
    home: Scaffold(
      appBar: AppBar(title: Text("Test")),
      body: Center(
        child: ElevatedButton.icon(
          onPressed: () => {},
          label: Image.asset("assets/images/yes_logo.png",
            width: 32, height: 32),
          icon: Text("Validate"),
        )));
}
```



Buttons

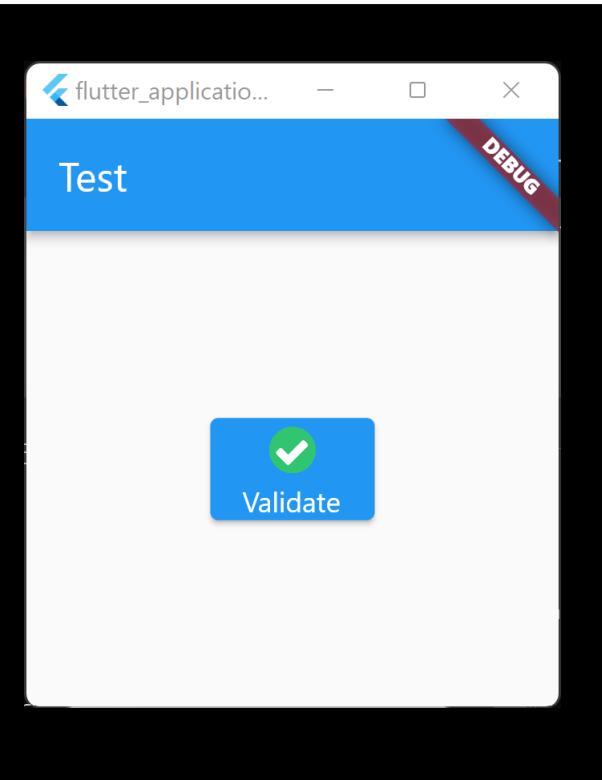
But what if we want to create an even more custom button (something that we can re-use). We can try to extend something from the existing classes.

```
class MyButton extends ElevatedButton {  
    MyButton(String iconAsset, String label, Function()? onPressedCallback)  
        : super( onPressed: onPressedCallback,  
            child: Column(children: [  
                Image.asset(iconAsset,  
                    width: 32,  
                    height: 32),  
                Text(label)  
            ],  
            mainAxisSize: MainAxisSize.min)  
        )  
    {}  
}
```

Buttons

But what if we want to create an even more custom button (something that we can re-use). We can try to extend something from the existing classes.

```
class MyAppState extends State<MyApp> {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar(title: Text("Test")),  
        body: Center(  
          child: MyButton(  
            "assets/images/yes_logo.png",  
            "Validate",  
            () => {}))));  
  }  
}
```



Buttons

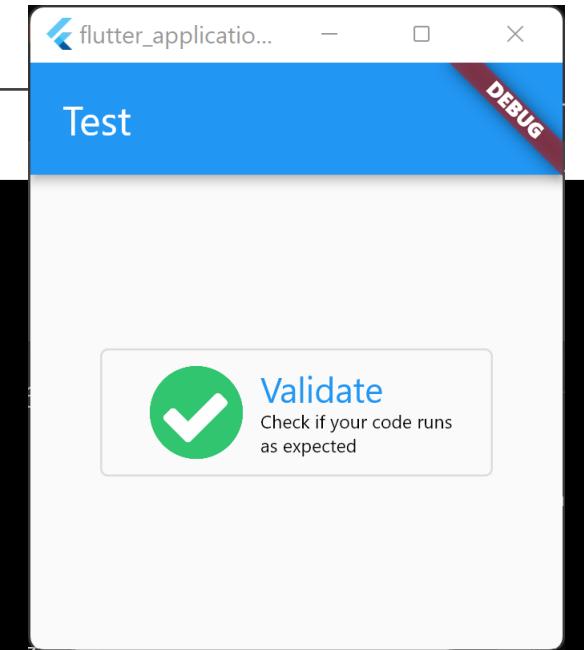
Using this technique, complex widgets can be created and reuse.

```
class MyButton extends OutlinedButton {  
    MyButton(String iconAsset, String label, String label2, Function()? fnc)  
        : super( onPressed: fnc,  
            child: Row(children: [  
                Image.asset(iconAsset, width: 64, height: 64),  
                Container(  
                    width: 100, height: 48,  
                    child: Column(children: [  
                        Container(width: 100, height: 22,  
                            child: Text(label, textScaleFactor: 1.25)),  
                        Text(label2, textScaleFactor: 0.66,  
                            style: TextStyle(color: Colors.black))  
                    ]))  
            ], mainAxisSize: MainAxisSize.min)) {}  
}
```

Buttons

Using this technique, complex widgets can be created and reuse.

```
class MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("Test")),
        body: Center(
          child: MyButton("assets/images/yes_logo.png",
            "Validate",
            "Check if your code runs as expected",
            () => {})
        )));
  }
}
```



Buttons

Besides these three type of buttons, flutter framework also provides a round button (based on icon) called `FloatingActionButton` (that is more easily customizable).

```
FloatingActionButton ({Key? key,  
                    Widget? child,  
                    String? tooltip,  
                    Color? foregroundColor,  
                    Color? backgroundColor,  
                    Color? focusColor,  
                    Color? hoverColor,  
                    Color? splashColor,  
                    required VoidCallback? onPressed,...})
```

```
FloatingActionButton.extended(...)
```

```
FloatingActionButton.large(...)
```

```
FloatingActionButton.small(...)
```

Buttons

Just like in the previous cases, let's add an image (a black and white icon) to our assets.

1. First, we need to add an image to our assets folder (let's name it 'thumbs_up.png'). Make sure that the image has transparency.



2. Second, add the new image to the `pubspec.yaml` file

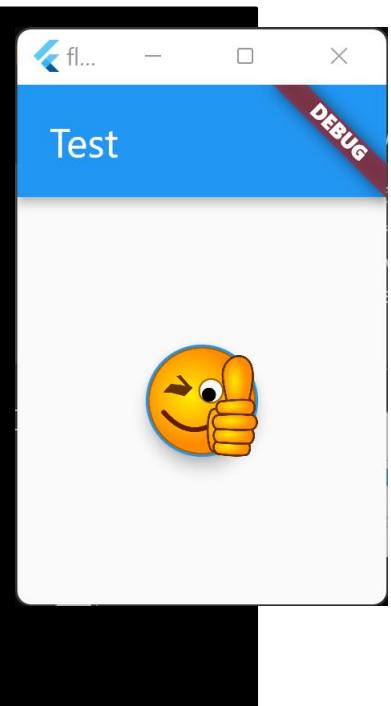
```
flutter:  
  uses-material-design: true  
  assets:  
    - assets/images/thumbs_up.png
```

Buttons

Then, let's create a simple `FloatingActionButton` using that icon.

If the image is bigger than the actual button, the image will be automatically scaled down to fit the button.

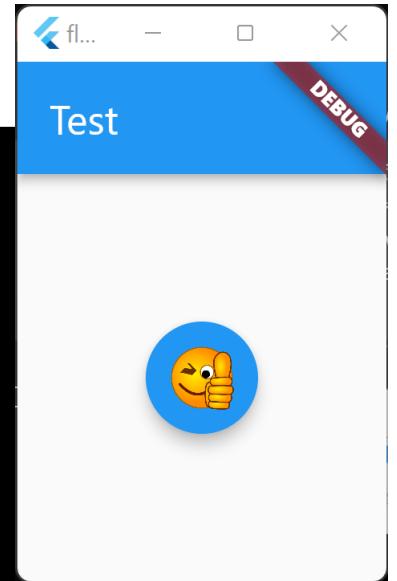
```
class MyAppState extends State<MyApp> {  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            home: Scaffold(  
                appBar: AppBar(title: Text("Test")),  
                body: Center(  
                    child: FloatingActionButton(  
                        child: Image.asset("assets/images/thumbs_up.png"),  
                        onPressed: () => {}))));  
    }  
}
```



Buttons

Then, let's create a simple `FloatingActionButton` using that icon. We can however, programmatically resize the icon to be smaller than the actual button:

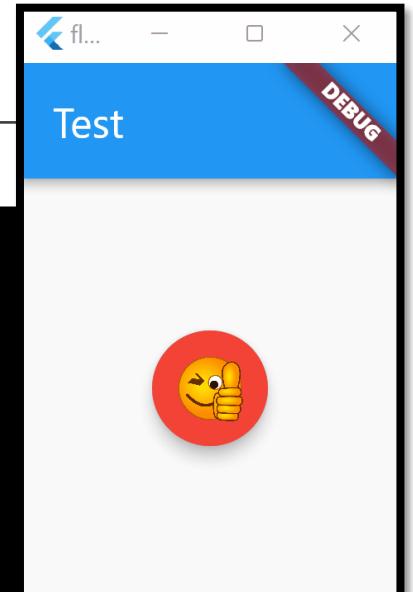
```
class MyAppState extends State<MyApp> {
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("Test")),
                body: Center(
                    child: FloatingActionButton(
                        child: Image.asset("assets/images/thumbs_up.png",
                            width: 32, height: 32),
                        onPressed: () => {})));
    }
}
```



Buttons

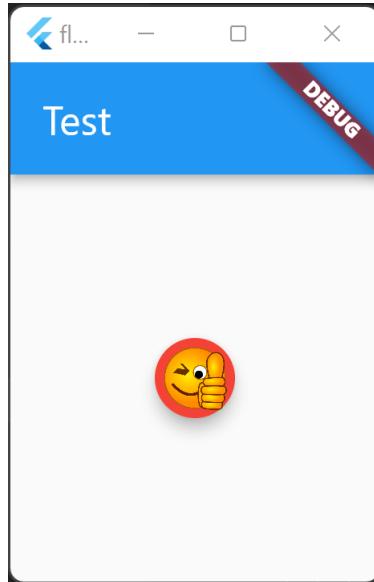
We can also customize the colors used for background or hover ...

```
Widget build(BuildContext context) {
  return MaterialApp(
    home: Scaffold(
      appBar: AppBar(title: Text("Test")),
      body: Center(
        child: FloatingActionButton(
          child: Image.asset("assets/images/thumbs_up.png",
            width: 32, height: 32),
          onPressed: () => {},
          backgroundColor: Colors.red,
          hoverColor: Colors.orange,
        ))));
}
```

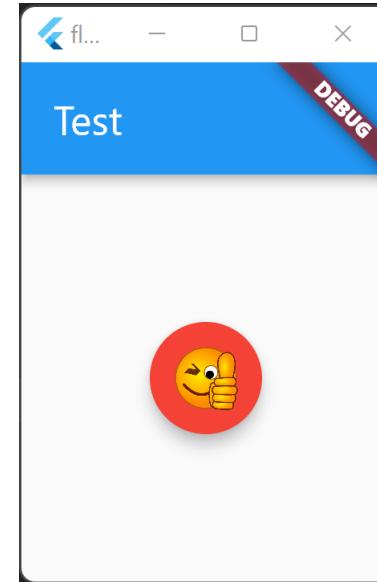


Buttons

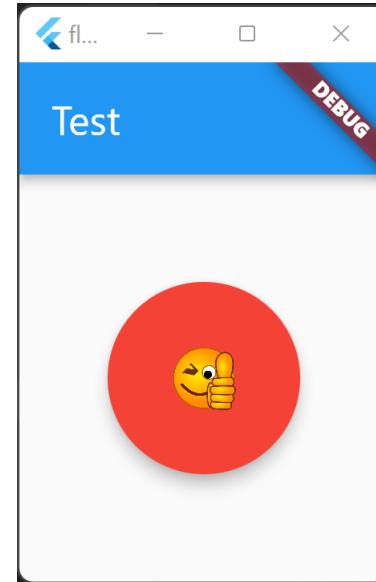
.small and .large named constructors make the button size smaller or bigger.



FloatingActionButton.small(...)



FloatingActionButton(...)

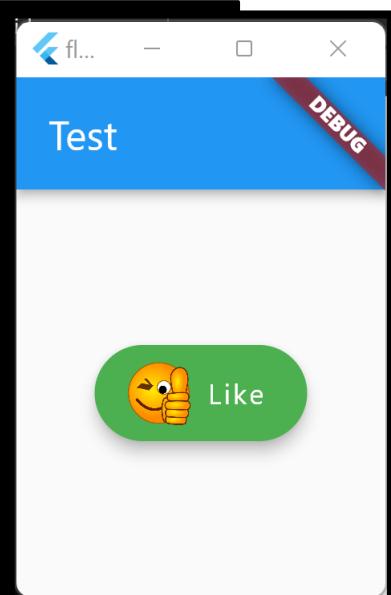


FloatingActionButton.large(...)

Buttons

The `.extended` named constructor can be used to create a button with text and icon.

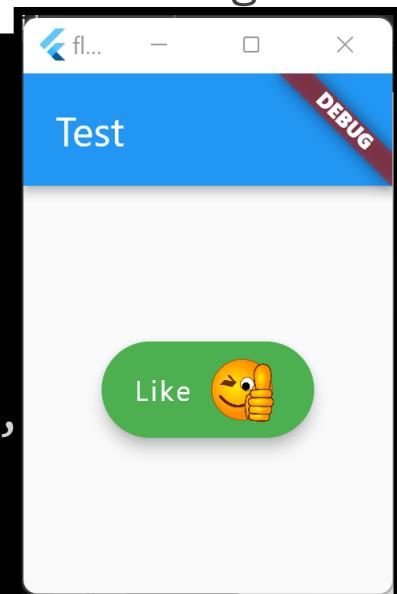
```
Widget build(BuildContext context) {
  return MaterialApp(
    home: Scaffold(
      appBar: AppBar(title: Text("Test")),
      body: Center(
        child: FloatingActionButton.extended(
          icon: Image.asset("assets/images/thumbs_up.png",
              width: 32, height: 32),
          label: Text("Like"),
          onPressed: () => {},
          backgroundColor: Colors.green,
          hoverColor: Colors.orange,
        ))));
}
```



Buttons

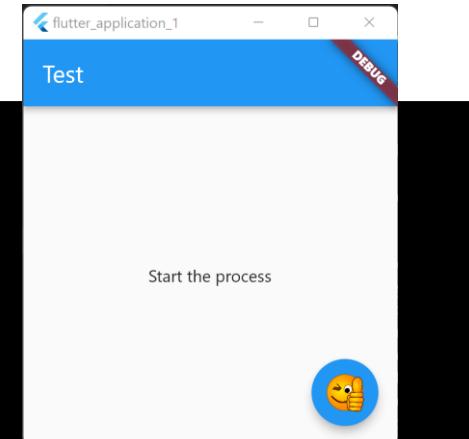
And as expected, since label and icon properties are widgets, they can be used with other widgets (in this example we've just reversed them to create a button where text is on the left and icon on the right)

```
Widget build(BuildContext context) {  
    return MaterialApp(  
        home: Scaffold(  
            appBar: AppBar(title: Text("Test")),  
            body: Center(  
                child: FloatingActionButton.extended(  
                    label: Image.asset("assets/images/thumbs_up.png",  
                        width: 32, height: 32),  
                    icon: Text("Like"),  
                    onPressed: () => {},  
                    backgroundColor: Colors.green,  
                    hoverColor: Colors.orange))));  
}
```



Buttons

Finally, the Scaffold widget has a property called `floatingActionButton` that can be set up with such a bottom (that will be located at the bottom-right side of the application).

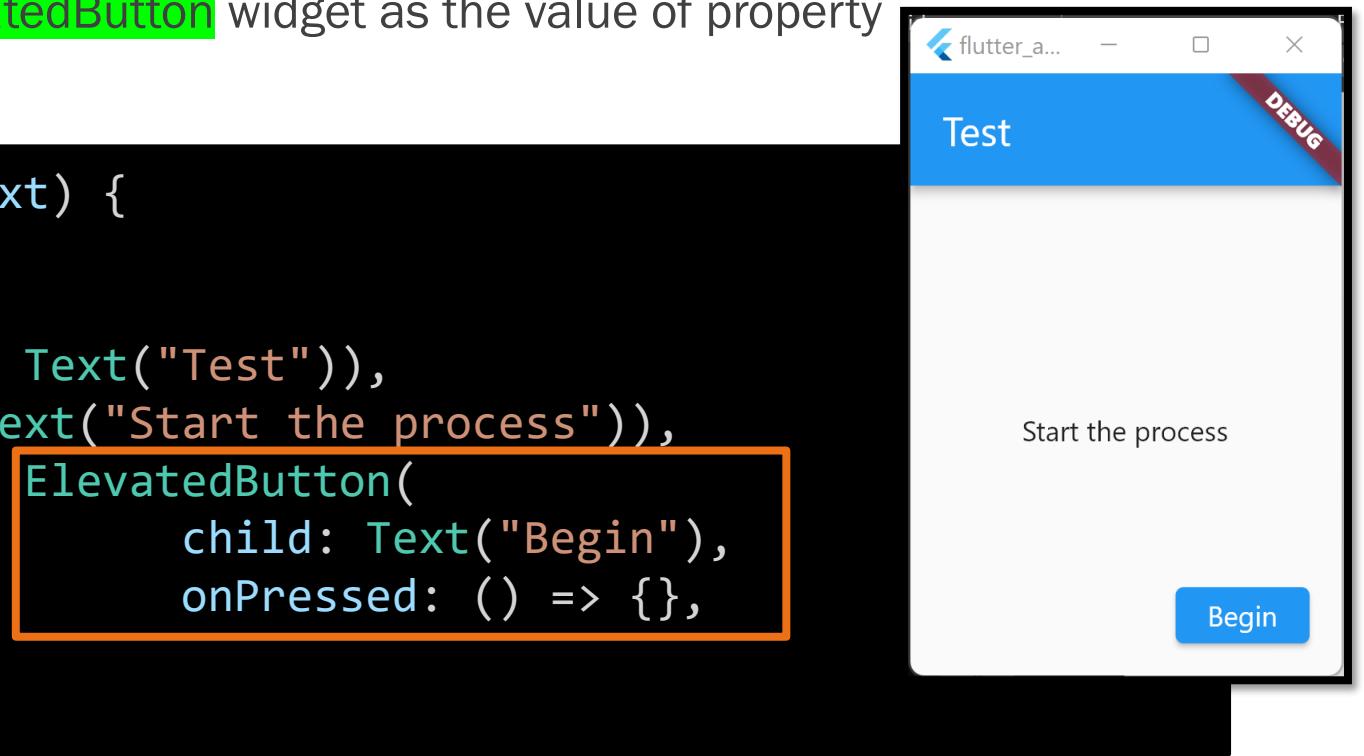


```
Widget build(BuildContext context) {
  return MaterialApp(
    home: Scaffold(
      appBar: AppBar(title: Text("Test")),
      body: Center(child: Text("Start the process")),
      floatingActionButton: FloatingActionButton(
        child: Image.asset("assets/images/thumbs_up.png",
          width: 32, height: 32),
        onPressed: () => {},
        backgroundColor: Colors.blue,
      )));
}
```

Buttons

The `floatingActionButton` property is also a Widget. This mean that even if the name suggests that the widget used in this case should be a `FloatingActionButton`, in reality it could be any other widget. For example, the next example uses an `ElevatedButton` widget as the value of property `floatingActionButton`.

```
Widget build(BuildContext context) {  
    return MaterialApp(  
        home: Scaffold(  
            appBar: AppBar(title: Text("Test")),  
            body: Center(child: Text("Start the process")),  
            floatingActionButton: ElevatedButton(  
                child: Text("Begin"),  
                onPressed: () => {},  
            )),  
    );  
}
```



Buttons

Another type of button available on Flutter framework is IconButton. This type of button is similar to the previous ones, with the exception that it uses icons (glyphs) instead of images.

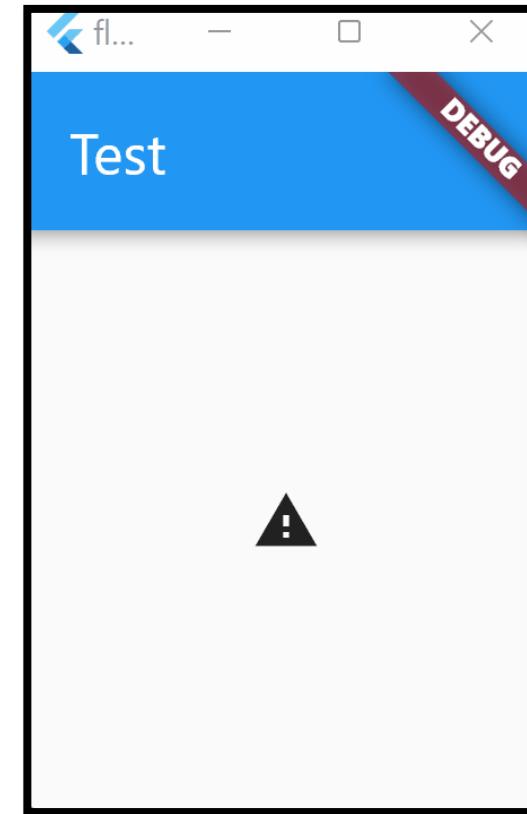
```
IconButton ({Key? key,  
          required VoidCallback? onPressed,  
          required Widget? icon,  
          double? iconSize,  
          Color? color,  
          Color? disabledColor,  
          Color? focusColor,  
          Color? hoverColor,  
          Color? splashColor,  
          ,...})
```

Just like previous cases, this type of button is more customizable, and it is designed to work with existing icons.

Buttons

The next example creates a very simple (centered) icon button with the alert sign.

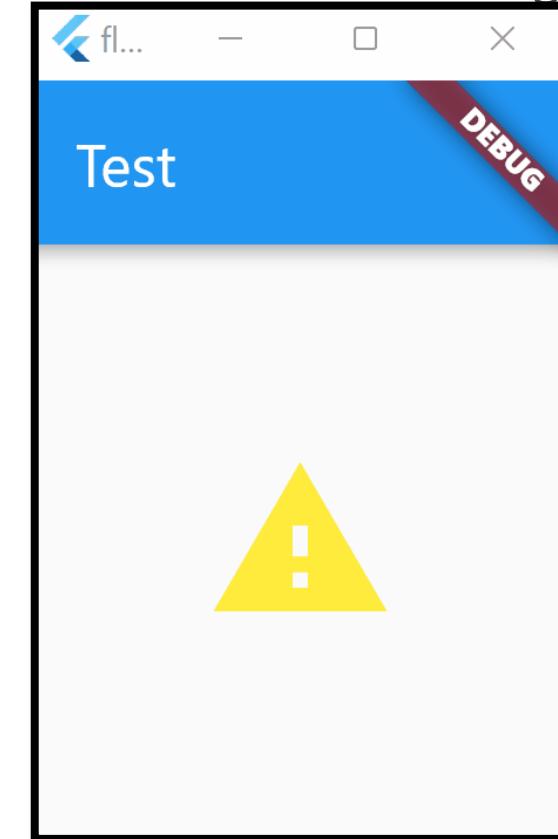
```
class MyAppState extends State<MyApp> {
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("Test")),
                body: Center(
                    child: IconButton(
                        icon: Icon(Icons.warning),
                        onPressed: () => {})));
    }
}
```



Buttons

The next example creates a customized icon button with a larger icon and red hovered background.

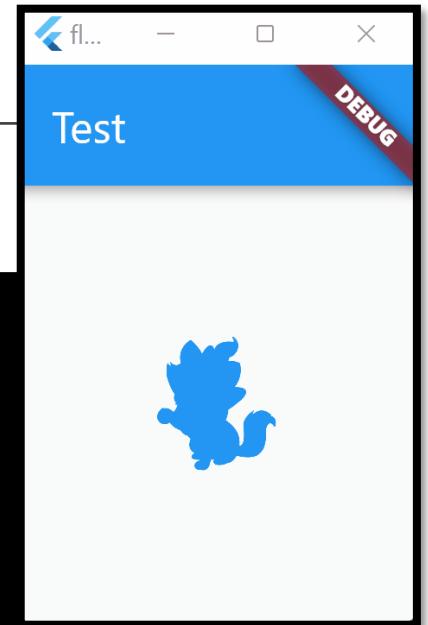
```
Widget build(BuildContext context) {
  return MaterialApp(
    home: Scaffold(
      appBar: AppBar(title: Text("Test")),
      body: Center(
        child: IconButton(
          icon: Icon(Icons.warning),
          onPressed: () => {},
          iconSize: 64,
          color: Colors.yellow,
          hoverColor: Colors.red,
        ))));
}
```



Buttons

We can also use a custom icon (via `ImageIcon` and `AssetImage` classes), with a size of 64 logical pixels, blue color and yellow hover color.

```
Widget build(BuildContext context) {
  return MaterialApp(
    home: Scaffold(
      appBar: AppBar(title: Text("Test")),
      body: Center(
        child: IconButton(
          icon: ImageIcon(AssetImage("assets/icons/cat.png")),
          onPressed: () => {},
          iconSize: 64,
          color: Colors.blue,
          hoverColor: Colors.yellow,
        ))));
}
```



Buttons

For convenience, there two extra buttons defined in Flutter:

1. **CloseButton** (a class that simulates an IconButton with the close icon)
2. **BackButton** (a class that simulates an IconButton with the back button)

```
CloseButton ({Key? key, Color? color, void Function? onPressed})
```

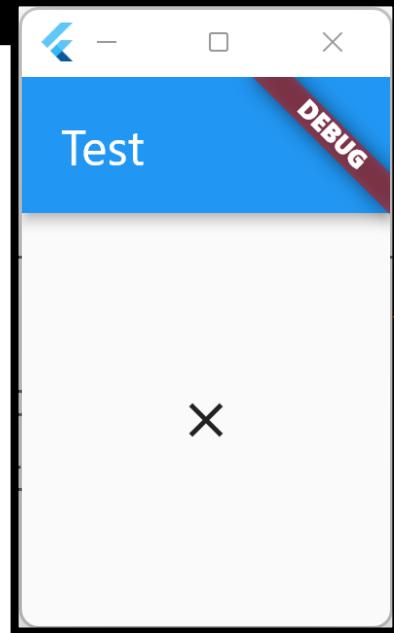
```
BackButton ({Key? key, Color? color, void Function? onPressed})
```

These types of button don't have so many customization → just the color and the onPressed callback.

However, since these types of buttons are often use, and they do have a clear usage and icon defined in material view, its easier to use this way.

Buttons

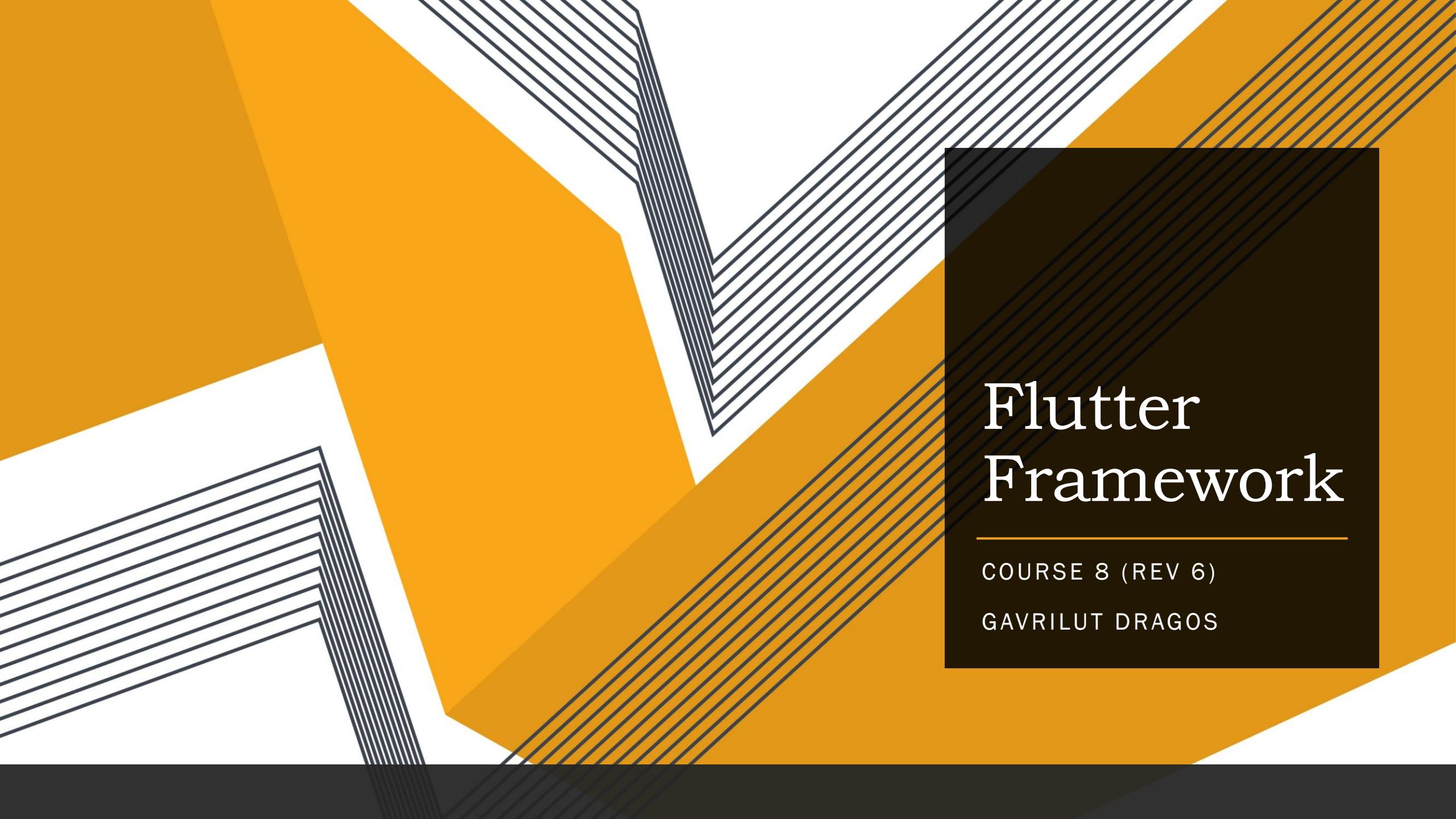
```
Widget build(BuildContext context) {  
    return MaterialApp( home: Scaffold(  
        appBar: AppBar(title: Text("Test")),  
        body: Center(child: CloseButton())));  
}
```



```
Widget build(BuildContext context) {  
    return MaterialApp( home: Scaffold(  
        appBar: AppBar(title: Text("Test")),  
        body: Center(child: BackButton())));  
}
```

Q & A



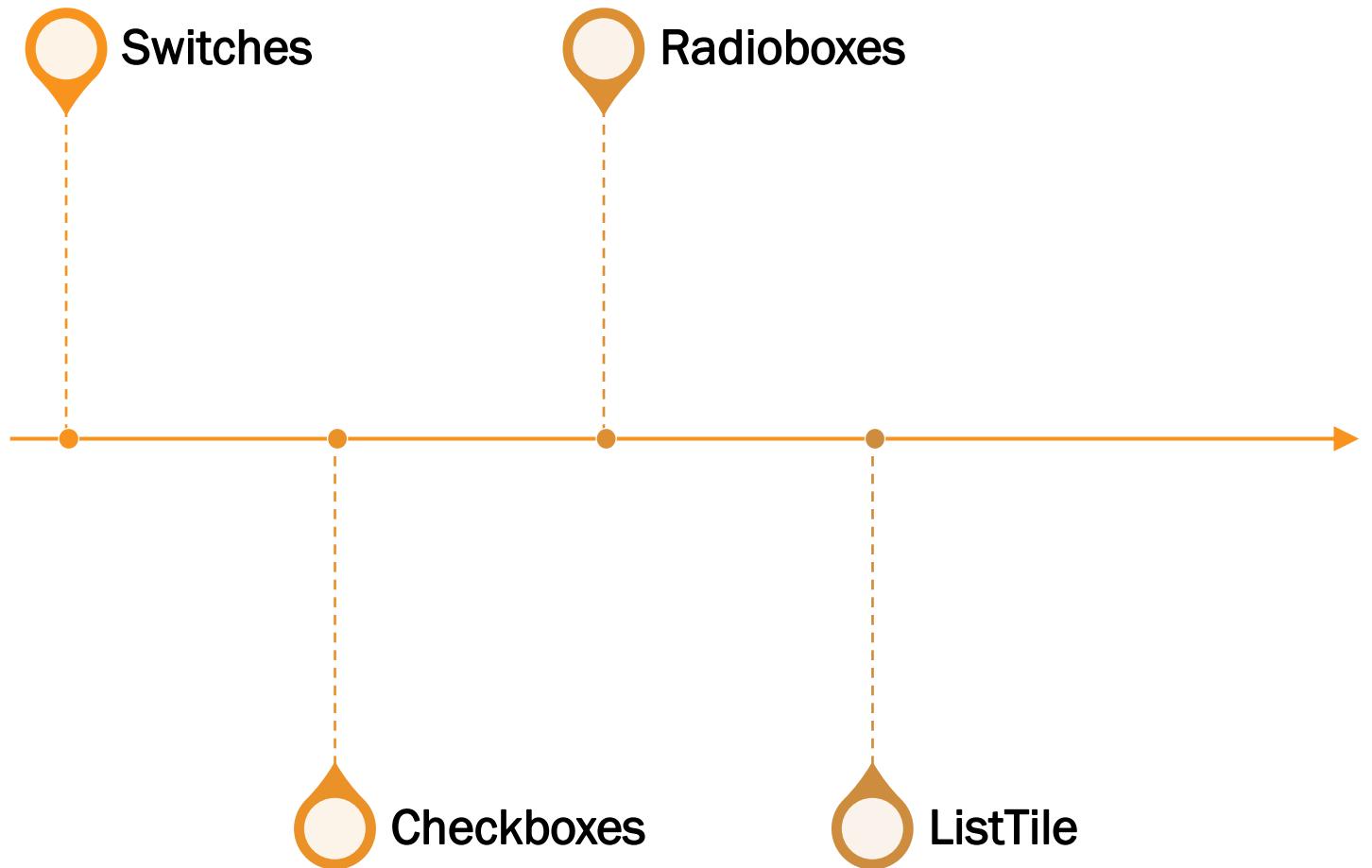


Flutter Framework

COURSE 8 (REV 6)

GAVRILUT DRAGOS

Agenda



Switches

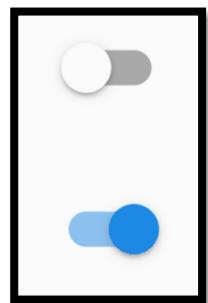
Switches

Switches are UX object that reflect an on/off state (true / false) value.

It is usually recommended to use a switch when the change happens immediately → meaning that once you enable it, the change that reflects it will be associated (for example: enabling Wifi)

Flutter has one widget (call **Switch**) that can be used for this kind of tasks.

A switch looks like in the following way:



← Un-checked (OFF or false)

← Checked (ON or true)

One main difference is that these widgets don't have an associated label (a text that explain what that checkbox represents). This means that usually, a Widget like this will be used as part of a Raw widget (so that it can incorporate a Text widget).

Switch

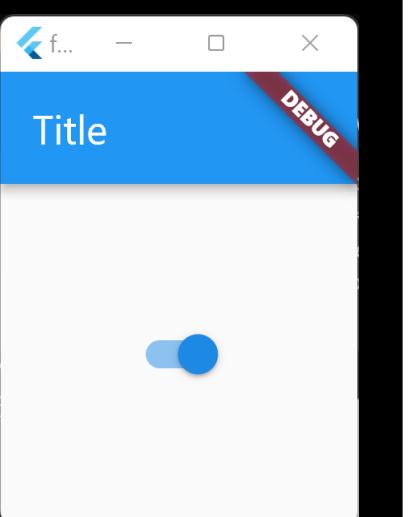
Constructor:

```
Switch({  
    required bool value,  
    required void Function(bool)? onChanged,  
    Color? activeColor,  
    Color? activeTrackColor,  
    Color? inactiveThumbColor,  
    Color? inactiveTrackColor,  
    Color? focusColor,  
    Color? hoverColor,  
    ... }  
)
```

Switch

Let's see a very simple example.

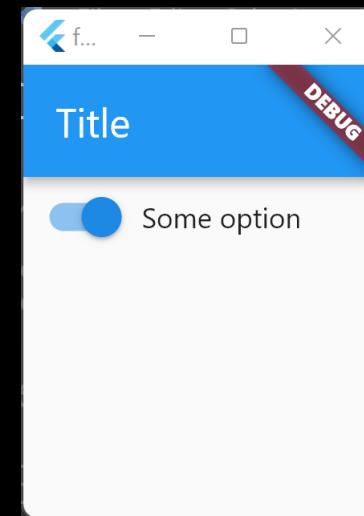
```
import 'package:flutter/material.dart';
void main() => runApp(MyApp());
class MyApp extends StatefulWidget {
  @override
  State<MyApp> createState() => MyAppState();
}
class MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("Title")),
        body: Center(child: Switch(onChanged: (b) => {}, value: true)));
  }
}
```

A screenshot of a Flutter application running in an emulator. The title bar says "Title" and has a "DEBUG" watermark. The main content area contains a single blue Switch widget with its switch button moved to the right, indicating the value is true. The entire code block is displayed on the left, with the line containing the Switch widget highlighted by a red rectangle.

Switch

So ... now we have a switch, but not text (label) associated with it. Let's see how we can add one.

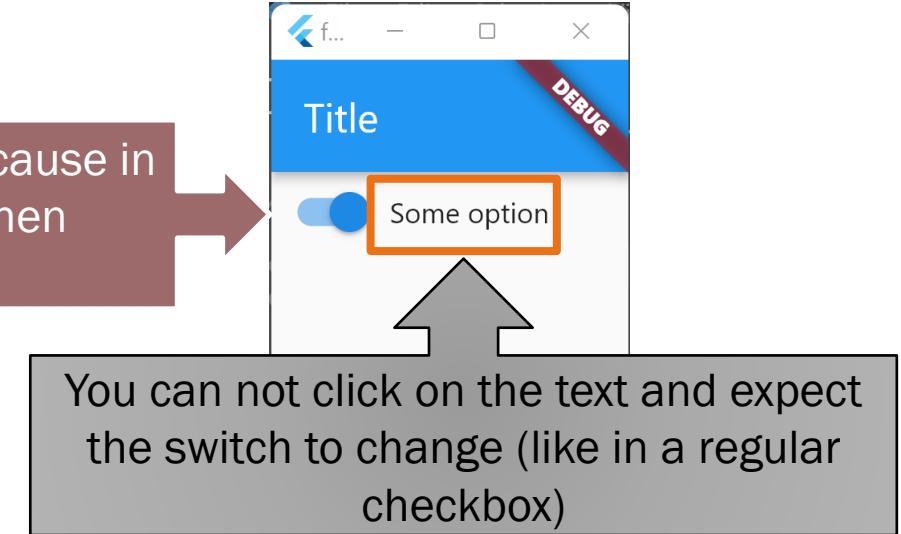
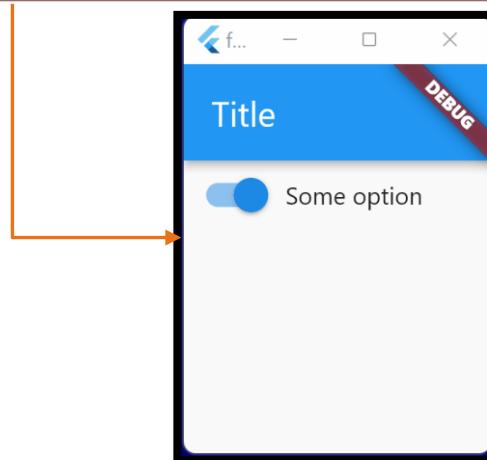
```
class MyAppState extends State<MyApp> {  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            home: Scaffold(  
                appBar: AppBar(title: Text("Title")),  
                body: Row(children: [  
                    Switch(onChanged: (b) => {}, value: true),  
                    Text("Some option")  
                ]));  
    }  
}
```



Switch

There are however a couple of things you might notice once .

You can click on the switch, but nothing happens. That's because in the code, there is no state (no variable) that changes when `onChanged` callback is called.

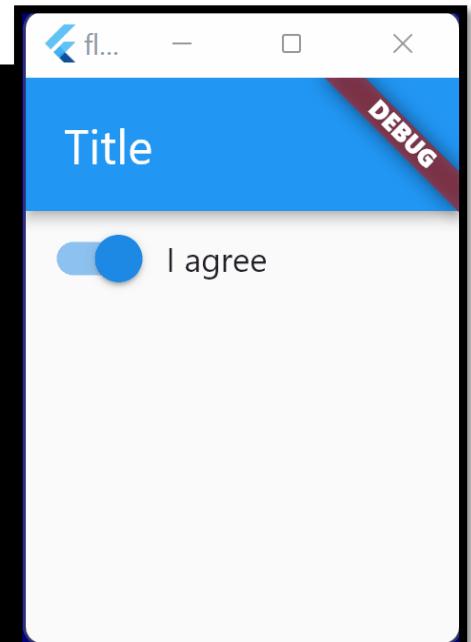


You can not click on the text and expect the switch to change (like in a regular checkbox)

Switch

A functional example:

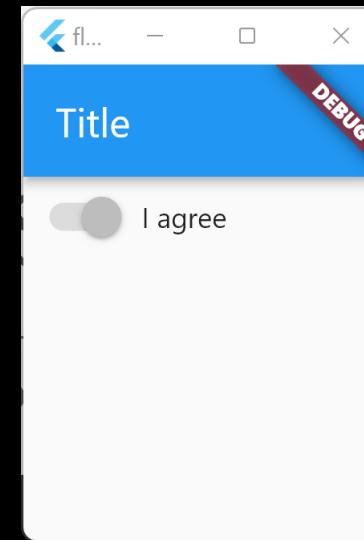
```
class MyAppState extends State<MyApp> {
    bool optValue = true;
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("Title")),
                body: Row(children: [
                    Switch(
                        onChanged: (b) => setState(() => optValue = b),
                        value: optValue),
                    Text(optValue ? "I agree" : "I disagree")
                ]));
    }
}
```



Switch

To disable a switch, just change the `onChanged` parameter value to null. Notice that just the switch will be disabled, the associated text will not.

```
class MyAppState extends State<MyApp> {
    bool optValue = true;
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("Title")),
                body: Row(children: [
                    Switch(onChanged: null, value: optValue),
                    Text(optValue ? "I agree" : "I disagree")
                ]));
    }
}
```



Switch

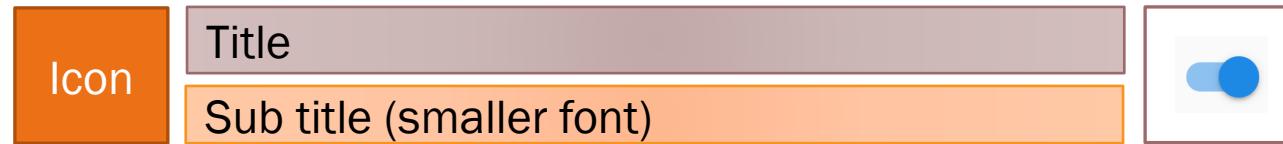
The next example sets the color of the text to look like something that was disabled.

```
class MyAppState extends State<MyApp> {
    bool optValue = true;
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("Title")),
                body: Row(children: [
                    Switch(onChanged: null, value: optValue),
                    Text(optValue ? "I agree" : "I disagree",
                        style: TextStyle(color: Colors.black45))
                ]));
    }
}
```



SwitchListTile

This is one way of trying to have a switch that also have a text associated. However, this does not solve the problem with clicking on the text. And since this is a very often case, a derived widget that has a label, an icon and a subtext was created → **SwitchListTile**



SwitchListTile

Constructor

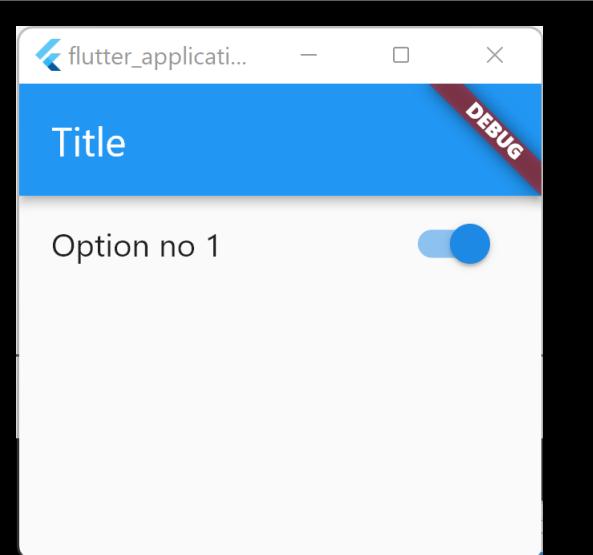
```
SwitchListTile({  
    required bool value,  
    required void ValueChanged<bool>? onChanged,  
    Widget? title,  
    Widget? subtitle,  
    Widget? secondary,  
    Color? activeColor,  
    Color? tileColor,  
    Color? inactiveThumbColor,  
    Color? inactiveTrackColor,  
    Color? selectedTileColor,  
    Color? hoverColor,  
    ... })
```

Title label
Subtitle text
Icon

SwitchListTile

A very simple example

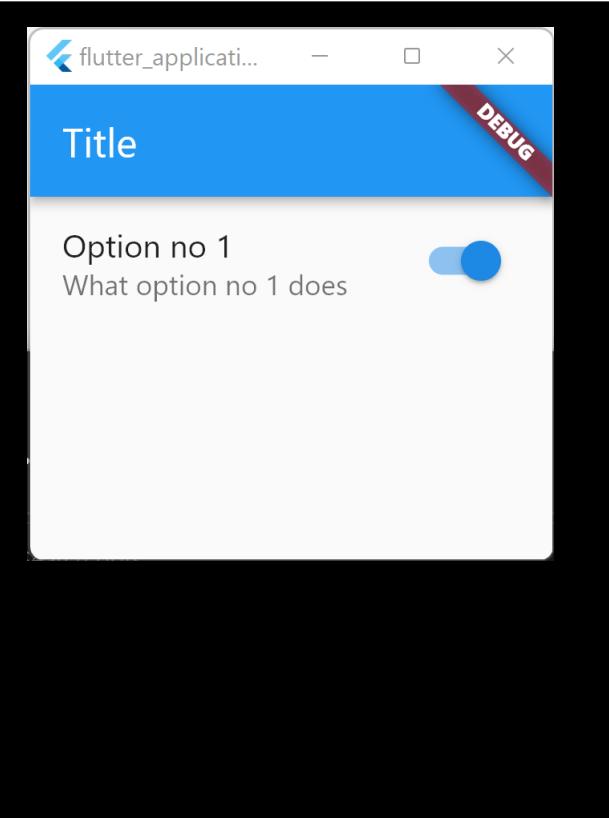
```
class MyAppState extends State<MyApp> {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar(title: Text("Title")),  
        body: SwitchListTile(  
          value: true,  
          onChanged: (b) => {},  
          title: Text("Option no 1"),  
        )),  
    );  
  }  
}
```



SwitchListTile

The `subtitle` parameter is usually an explanation of the option.

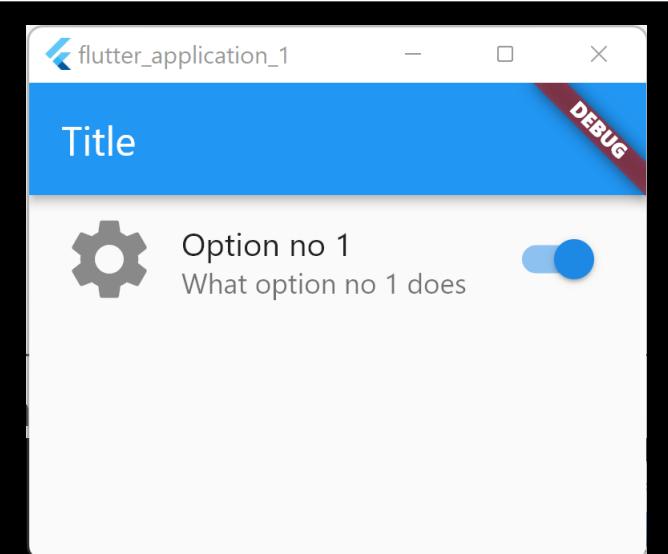
```
class MyAppState extends State<MyApp> {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar(title: Text("Title")),  
        body: SwitchListTile(  
          value: true,  
          onChanged: (b) => {},  
          title: Text("Option no 1"),  
          subtitle: Text("What option no 1 does"),  
        )),  
    );  
  }  
}
```



SwitchListTile

To show an icon in front of the option, use the `secondary` parameter.

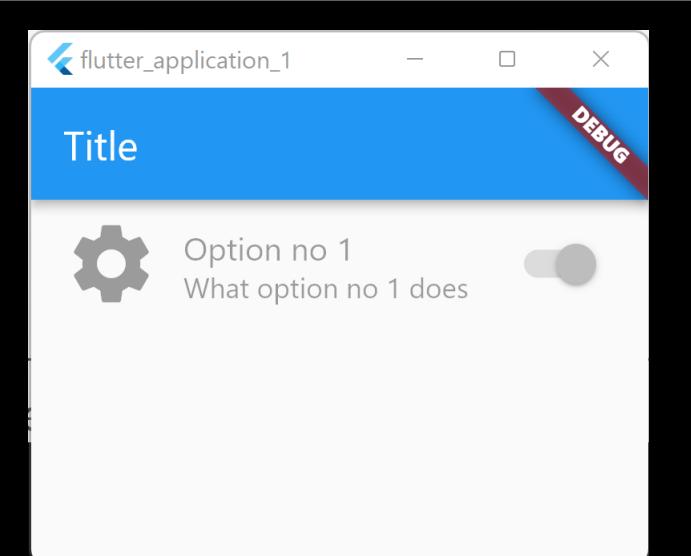
```
class MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("Title")),
        body: SwitchListTile(
          value: true,
          onChanged: (b) => {},
          title: Text("Option no 1"),
          subtitle: Text("What option no 1 does"),
          secondary: Icon(Icons.settings, size: 48),
        )));
}
```



SwitchListTile

To disable the widget (icon, title, subtitle), just set the value of `onChanged` parameter to null.

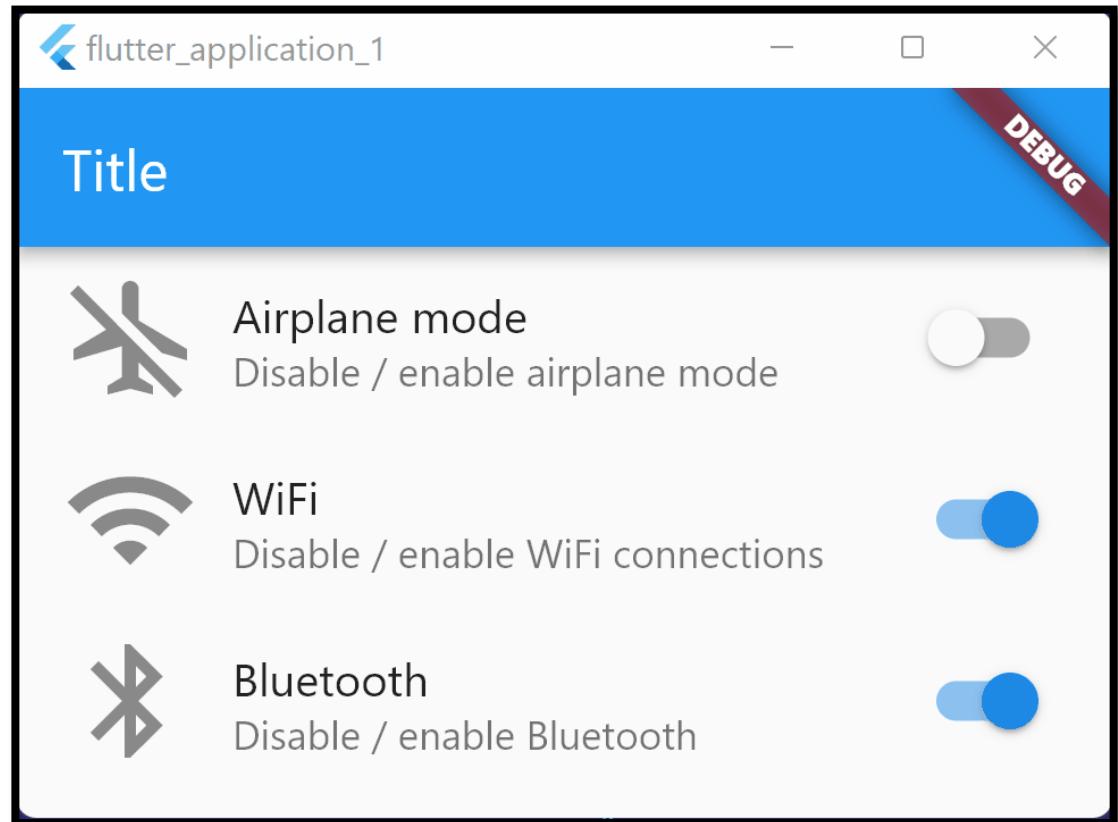
```
class MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("Title")),
        body: SwitchListTile(
          value: true,
          onChanged: null,
          title: Text("Option no 1"),
          subtitle: Text("What option no 1 does"),
          secondary: Icon(Icons.settings, size: 48),
        )));
  }
}
```



SwitchListTile

Let's build a more complex example:

- 3 options (Airplane mode, wifi and Bluetooth)
- If Airplane mode is not checked, then wifi and Bluetooth will be disabled (not checkable)
- All 3 options should have icons
- The Airplane mode icon should be different pending on airplane mode status



SwitchListTile

A more complex example

```
class MyAppState extends State<MyApp> {  
    bool airplaneMode = false, enablewifi = true, enableBluetooth = true;  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            home: Scaffold(  
                appBar: AppBar(title: Text("Title"),),  
                body: ...  
            ),  
        );  
    }  
}
```



First, we need some local Boolean variable to hold the state for the three options

SwitchListTile

A more complex example

```
class MyAppState extends State<MyApp> {
    bool airplaneMode = false, enablewifi = true, enableBluetooth = true;
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("Title")),
                body: Column(
                    children: [
                        SwitchListTile(...),
                        SwitchListTile(...),
                        SwitchListTile(...),
                    ])));
    }
}
```

We will use a Column widget with three
children, one for each options

SwitchListTile

A more complex example

```
class MyAppState extends State<MyApp> {
    bool airplaneMode = false, enableWifi = true, enableBluetooth = true;
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("My App")),
                body: Column(
                    children: [
                        SwitchListTile(
                            value: airplaneMode,
                            onChanged: (b) => setState(() { airplaneMode = b; }),
                            title: Text("Airplane mode"),
                            subtitle: Text("Disable / enable airplane mode"),
                            secondary: Icon(airplaneMode ? Icons.airplanemode_active
                                : Icons.airplanemode_inactive,
                                size: 48),
                            hoverColor: Colors.amber,
                        ),
                        SwitchListTile(...),
                        SwitchListTile(...),
                        SwitchListTile(...),
                    ])));
    }
}
```



SwitchListTile

A more complex example

```
class MyAppState extends State<MyApp> {
    bool airplaneMode = false, enableWifi = true, enableBluetooth = true;
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("Title")),
                body: Column(
                    children: [
                        SwitchListTile(...),
                        SwitchListTile(...) // This line is highlighted with an orange box
                        SwitchListTile(...),
                    ])));
    }
}
```



```
SwitchListTile(
    value: enableWifi,
    onChanged: airplaneMode ? null
        : (b) => setState(() {
            enableWifi = b;
        }),
    title: Text("WiFi"),
    subtitle: Text("Disable / enable WiFi connections"),
    secondary: Icon(Icons.wifi, size: 48),
)
```

SwitchListTile

A more complex example

```
class MyAppState extends State<MyApp> {
    bool airplaneMode = false, enableWifi = true, enableBluetooth = true;
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("Title")),
                body: Column(
                    children: [
                        SwitchListTile(...),
                        SwitchListTile(...),
                        SwitchListTile(...)
                    ]));
    }
}
```

```
SwitchListTile(
    value: enableBluetooth,
    onChanged: airplaneMode ? null
        : (b) => setState(() {
            enableBluetooth = b;
       }),
    title: Text("Bluetooth"),
    subtitle: Text("Disable / enable Bluetooth"),
    secondary: Icon(Icons.bluetooth, size: 48),
)
```

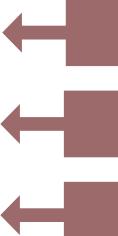
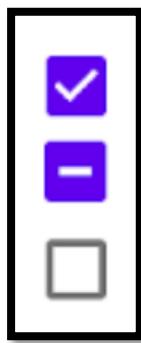
Checkboxes

Checkbox

Checkboxes are UX objects that can have usually two states (checked or unchecked). In some cases, a checkbox has a 3rd state (unknown).

A checkbox is similar to a switch in terms of how it works, interface, etc.

It is recommended to use a checkbox when you are setting multiple options (on or off) that are going to be applied when a button will be pressed (similar to a configuration panel).



Checked (ON or true)

Unknown state (null)

Un-checked (OFF or false)

One main difference is that these widgets don't have an associated label (a text that explain what that checkbox represents). This means that usually, a Widget like this will be used as part of a Raw widget (so that it can incorporate a Text widget).

Checkbox

Constructor:

```
Checkbox({  
    required bool value,  
    required void Function(bool?)? onChanged,  
    Color? activeColor,  
    Color? checkColor,  
    Color? focusColor,  
    Color? hoverColor,  
    bool tristate = false  
    ... }  
)
```

Checkbox

A very simple example:

```
class MyAppState extends State<MyApp> {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar(title: Text("Title")),  
        body: Checkbox(  
          value: true,  
          onChanged: (b) => {},  
        )),  
    );  
  }  
}
```

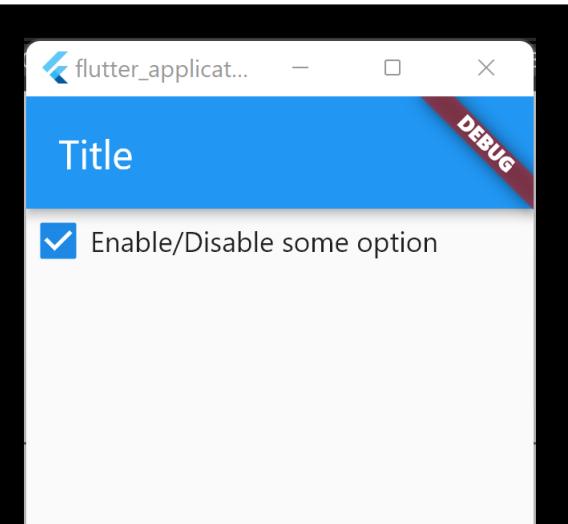


Just like in the case of Switches,
there aren't any labels
associated with a checkbox

Checkbox

To add a text, a similar technique like in the case of switches can be applied (use a Row with a checkbox and a text). However, the same limitations applies: disabling the checkbox will not disable the text, text can not be clicked, etc.

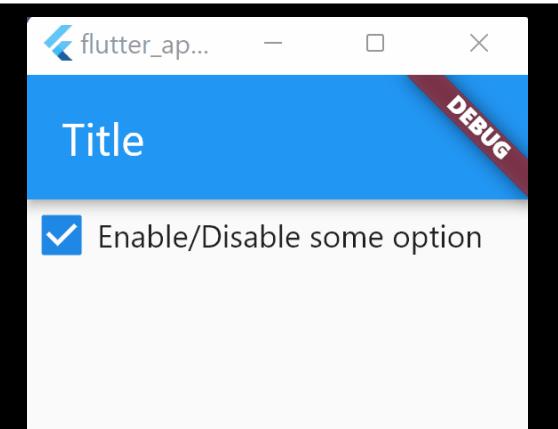
```
class MyAppState extends State<MyApp> {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar(title: Text("Title")),  
        body: Row(children: [  
          Checkbox(value: true, onChanged: (b) => {},),  
          Text("Enable/Disable some option")  
        ]));  
  }  
}
```



Checkbox

A more complex example:

```
class MyAppState extends State<MyApp> {
    bool option_1 = true;
    @override
    Widget build(BuildContext context) {
        return MaterialApp( home: Scaffold(
            appBar: AppBar(title: Text("Title")),
            body: Row(children: [
                Checkbox(
                    value: option_1,
                    onChanged: (b) => setState(() { option_1 = b ?? false; }),
                ),
                Text("Enable/Disable some option")
            ]));
    }
}
```



Checkbox

A more complex example:

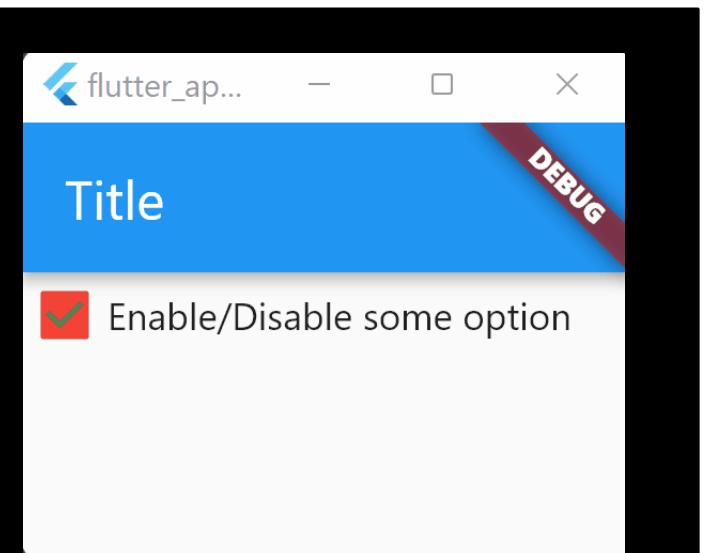
```
class MyAppState extends State<MyApp> {
    bool option_1 = true;
    @override
    Widget build(BuildContext context) {
        return MaterialApp( home: Scaffold(
            appBar: AppBar(title: Text("Title")),
            body: Row(children: [
                Checkbox(
                    value: option_1,
                    onChanged: (b) => setState(() { option_1 = b ?? false; })),
                Text("Enable/Disable some option"),
            ]));
    }
}
```

Another difference from a Switch is that the parameter for `onChange` is `bool?` (to reflect a possible tristate).

Checkbox

A more complex example (with colors):

```
class MyAppState extends State<MyApp> {
  bool option_1 = true;
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("Title")),
        body: Row(children: [ Checkbox(
          value: option_1,
          hoverColor: Colors.amber,
          activeColor: Colors.red,
          checkColor: Colors.green,
          onChanged: (b) => setState(() { option_1 = b ?? false; }),
        ), Text("Enable/Disable some option")
      )));}}
```



Checkbox

A tri-state example:

```
class MyAppState extends State<MyApp> {
    bool? option_1 = true;
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("Title")),
                body: Row(children: [
                    Checkbox(
                        value: option_1,
                        tristate: true,
                        onChanged: (b) => setState(() { option_1 = b; })),
                    Text("Enable/Disable some option")
                ])));
}
```

Checkbox

A tri-state example:

```
class MyAppState extends State<MyApp> {
    bool? option_1 = true;
    @override
    Widget build(BuildContext context) {
        return Scaffold(
            appBar: AppBar(
                title: Text("Title")),
            body: Row(children: [
                Checkbox(
                    value: option_1,
                    tristate: true,
                    onChanged: (b) => setState(() { option_1 = b;}),
                ),
                Text("Enable/Disable some option")
            ]));
    }
}
```

First, the option_1 is no longer a bool, it is now a `bool?` to reflect all 3 possible values:

[`true` for checked, `false` for unchecked and `null` for unknown]

Checkbox

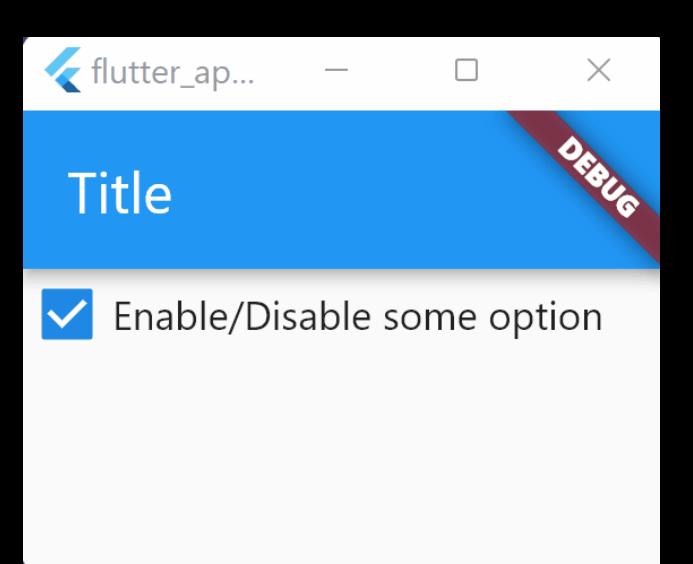
A tri-state example:

```
class MyAppState extends State<MyApp> {
    bool? option_1 = true;
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("Title")),
                body: Row(children: [
                    Checkbox(
                        value: option_1,
                        tristate: true,  
                        onChanged: (b) => setState(() { option_1 = b; })),
                    Second, tristate parameter must be set to true. option")
                ]));
}
```

Checkbox

A tri-state example:

```
class MyAppState extends State<MyApp> {
    bool? option_1 = true;
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("Title")),
                body: Row(children: [
                    Checkbox(
                        value: option_1,
                        tristate: true,
                        onChanged: (b) => setState(() { option_1 = b; })),
                    Text("Enable/Disable some option")
                ])));
}
```

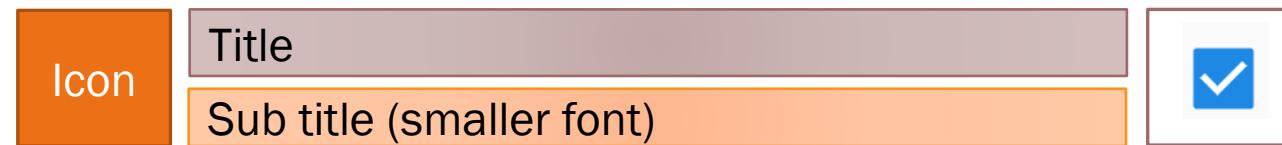


Third, on the `onChanged` callback just copy the value of `b` to `option_1`.

CheckboxListTile

Similar to a switch where there is a special widget called `SwitchListTile`, there is a similar form for a Checkbox as well (called `CheckboxListTile`) that has the same properties:

- You can click on the entire space, and this action will trigger an a check or uncheck for the object
- There is a title (and a subtitle)
- There is an icon



CheckboxListTile

Constructor

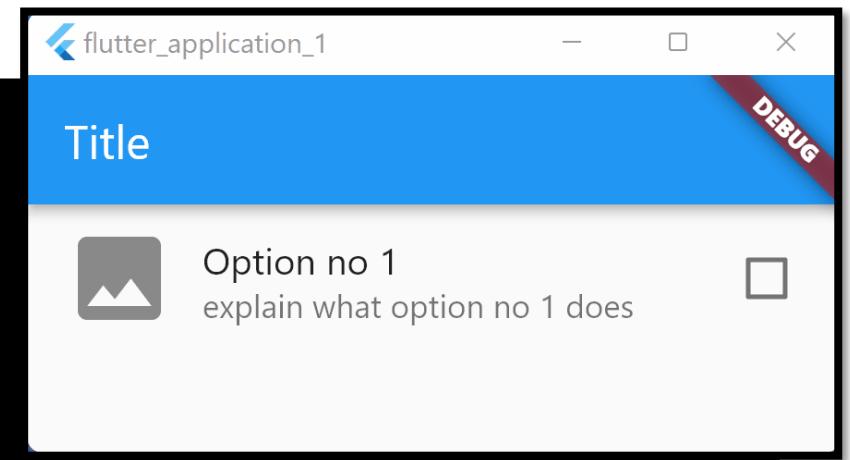
```
CheckboxListTile({  
    required bool value,  
    required void ValueChanged<bool?>? onChanged,  
    Widget? title,  
    Widget? subtitle,  
    Widget? secondary,  
    Color? activeColor,  
    Color? tileColor,  
    Color? checkColor,  
    Color? selectedTileColor,  
    bool tristate = false  
    ... })
```

Title label
Subtitle text
Icon

CheckboxListTile

A simple example

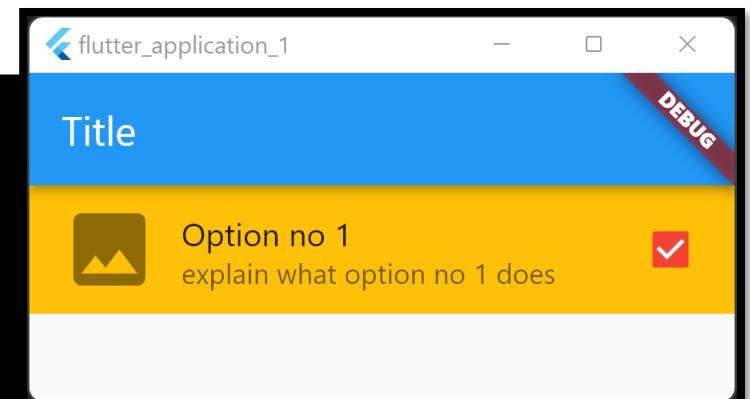
```
class MyAppState extends State<MyApp> {
  bool? option_1 = true;
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("Title")),
        body: CheckboxListTile(
          title: Text("Option no 1"),
          subtitle: Text("explain what option no 1 does"),
          secondary: Icon(Icons.photo, size: 48),
          onChanged: (b) => setState(() { option_1 = b; }),
          value: option_1));
  }
}
```



CheckboxListTile

A simple example with colors

```
Widget build(BuildContext context) {  
  return MaterialApp(  
    home: Scaffold(  
      appBar: AppBar(title: Text("Title")),  
      body: CheckboxListTile(  
        title: Text("Option no 1"),  
        subtitle: Text("explain what option no 1 does"),  
        secondary: Icon(Icons.photo, size: 48),  
        tileColor: Colors.amber,  
        activeColor: Colors.red,  
        onChanged: (b) => setState(() { option_1 = b; }),  
        value: option_1));  
}
```

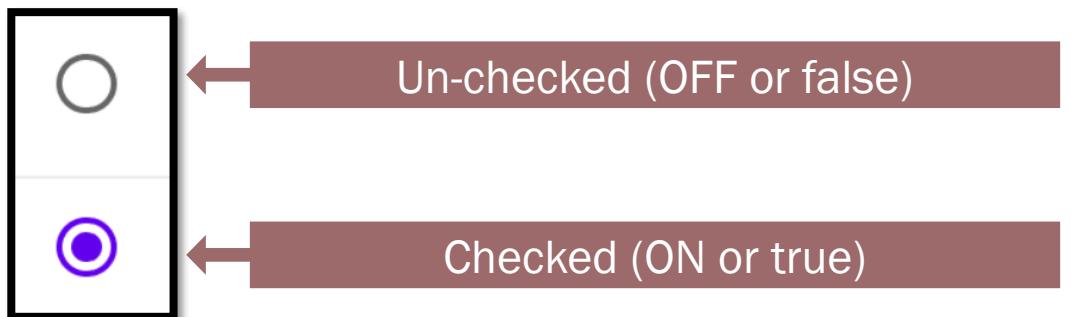


Radioboxes

Radioboxes

Radioboxes are UI widgets that can have only one of them being checked at some moment of time. Radioboxes use a **groupValue** to compare their value with the one from the group. The one that has the same value as the one from the **groupValue** will be checked, the rest will not be.

Flutter has one widget (called Radio) that is defined as a template/generic. This is very helpful when dealing with enum values.



One main difference is that these widgets don't have an associated label (a text that explain what that radiobox represents). This means that usually, a Widget like this will be used as part of a Raw widget (so that it can incorporate a Text widget).

Radio

Constructor

```
Radio<T>({  
    required T value,  
    required T? groupValue,  
    required void ValueChanged<T?>? onChanged,  
    Color? activeColor,  
    Color? focusColor,  
    Color? hoverColor,  
    Color? overlayColor,  
    ... }  
)
```

Radio

A regular usage for a Radio will be as follows

```
class MyAppState extends State<MyApp> {  
    String s = "2";
```

First let's create a local variable that will be the groupValue for several Radio widgets.

In our case it will be a string variable with three possible values:
“1”, “2” and “3”

```
}
```

Radio

A regular usage for a Radio will be as follows

```
class MyAppState extends State<MyApp> {  
    String s = "2";  
  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            home: Scaffold(  
                appBar: AppBar(title: Text("Title")),  
                body: Row(children: [  
                    Radio<String>(...),  
                    Radio<String>(...),  
                    Radio<String>(...)  
                ]));  
    }  
}
```

Second, we will create a Row widget with 3 Radio widgets.

Notice that each Radio widget is a template of type String

Radio

A regular usage for a Radio will be as follows

```
class MyAppState extends State<MyApp> {
    String s = "2";

    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("Radio Example")),
                body: Row(children: [
                    Radio<String>(...),
                    Radio<String>(...),
                    Radio<String>(...)
                ])));
    }
}
```

The code shows a `Row` widget containing three `Radio` widgets. The first `Radio` is highlighted with an orange box. A callout box points to a detailed view of its properties:

```
Radio<String>(
    value: "1",
    groupValue: s,
    onChanged: (b) => setState(() { s = "1"; })
)
```

Radio

A regular usage for a Radio will be as follows

```
class MyAppState extends State<MyApp> {  
    String s = "2";  
  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            home: Scaffold(  
                appBar: AppBar(title: Text("Radio Example")),  
                body: Row(children: [  
                    Radio<String>(...),  
                    Radio<String>(...),  
                    Radio<String>(...),  
                ]));  
    }  
}
```

The diagram illustrates the state flow in the code. A red box highlights the variable `s` in the `onChanged` callback. An arrow points from this box to the `groupValue: s` parameter of the first `Radio` widget in the `Row`. Another arrow points from the `groupValue: s` parameter to the `value: "1"` parameter of the same `Radio` widget, indicating that when `s` is "1", this radio button will be checked.

This translates in **when string variable `s` has the value “1”, this Radio will be checked.**

Radio

A regular usage for a Radio will be as follows

```
class MyAppState extends State<MyApp> {  
    String s = "2";  
  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            home: Scaffold(  
                appBar: AppBar(title: T  
                body: Row(children: [  
                    Radio<String>(...),  
                    Radio<String>(...),  
                    Radio<String>(...)  
                ]));  
    }  
}
```

```
    )  
    Radio<String>(  
        value: "1",  
        groupValue: s,  
        onChanged: (b) => setState(() {  
            s = "1";  
        })  
    )
```

When click, change the value of variable “s” to “1” so that this Radio will be checked.

```
{ s = "1"; }
```

Alternatively, one can also write { s = b; }

Radio

A regular usage for a Radio will be as follows

```
class MyAppState extends State<MyApp> {
    String s = "2";

    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("Title")),
                body: Row(children: [
                    Radio<String>(...),
                    Radio<String>(...), // This Radio is highlighted with an orange box
                    Radio<String>(...),
                ])));
    }
}
```

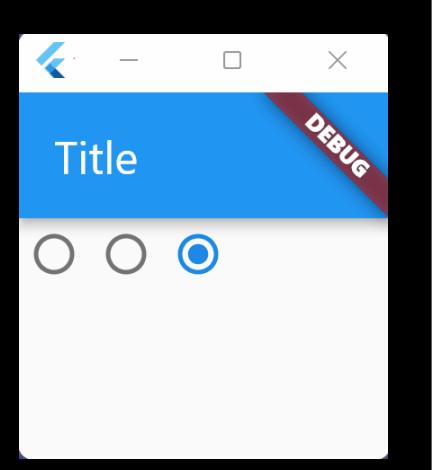
The diagram illustrates the state flow in the code. It shows a callout pointing from the highlighted `Radio<String>(...)` in the `Row` to the `onChanged` callback of the `Radio` widget. Another callout points from this `onChanged` callback to the `setState` call in the `build` method, which then updates the `s` variable.

```
graph TD
    Radio[Radio<String>(...)] --> onChanged[onChanged: (b) => setState(() { s = "2"; })]
    onChanged --> setState[setState(() { s = "2"; })]
```

Radio

A regular usage for a Radio will be as follows

```
class MyAppState extends State<MyApp> {  
    String s = "2";  
  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            home: Scaffold(  
                appBar: AppBar(title: Text("Title")),  
                body: Row(children: [  
                    Radio<String>(...),  
                    Radio<String>(...),  
                    Radio<String>(...)  
                ]));  
    }  
}
```

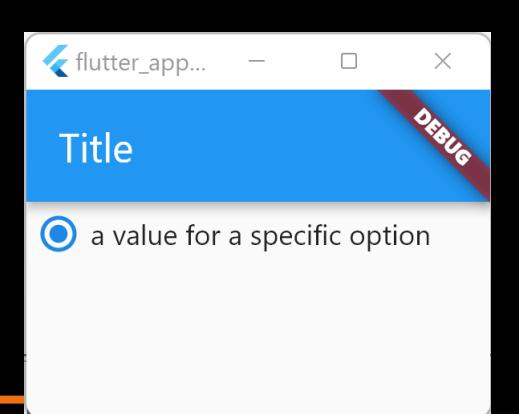


```
Radio<String>(  
    value: "3"  
    groupValue: s,  
    onChanged: (b) => setState(() { s = "3"; })  
)
```

Radio

One way of adding a label to a radio box is to put it in a Row widget and add a Text after it.

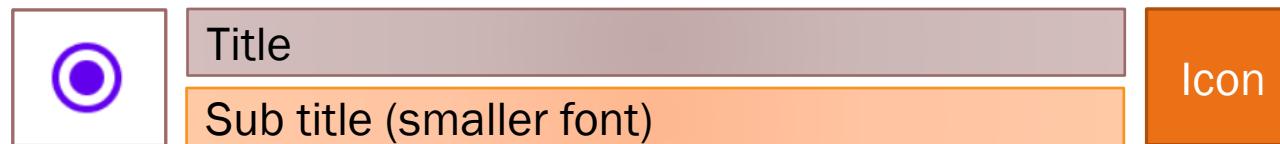
```
class MyAppState extends State<MyApp> {
    String s = "2";
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("Title")),
                body: Row(children: [
                    Radio<String>(
                        value: "1",
                        groupValue: s,
                        onChanged: (b) => setState(() { s = "1"; })),
                    Text("a value for a specific option")
                ])); }
}
```



RadioListTile

The other option is to use a RadioListTile (similar to the one from Checkbox and Switch). This widget has the same parameters as with the CheckboxListTile or SwitchListTile:

- A title
- A subtitle
- A secondary parameter that usually serves as an icon



Another observation here is that by default, the radio button is placed on the left side (as different from the checkbox that by default is position on the right side).

RadioListTile

Constructor

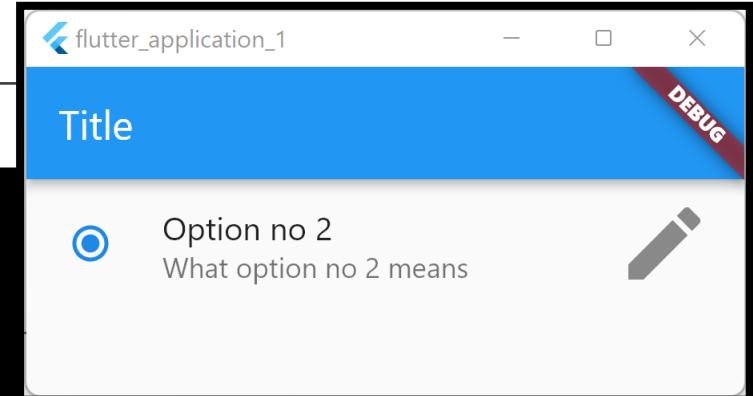
```
RadioListTile<T>({  
    required T value,  
    required T? groupValue,  
    required void ValueChanged<T?>? onChanged,  
    Widget? title,  
    Widget? subtitle,  
    Widget? secondary,  
    Color? activeColor,  
    Color? tileColor,  
    Color? selectedTileColor,  
    ... }  
)
```

Title label
Subtitle text
Icon

RadioListTile

A very simple example:

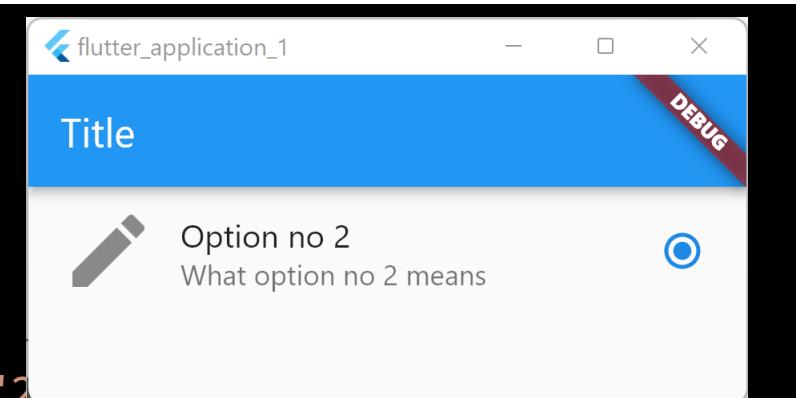
```
class MyAppState extends State<MyApp> {
    String s = "2";
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold( appBar: AppBar(title: Text("Title")),
                body: RadioListTile<String>(
                    value: "2",
                    onChanged: (b) => setState(() { s = "2"; }),
                    groupValue: s,
                    title: Text("Option no 2"),
                    subtitle: Text("What option no 2 means"),
                    secondary: Icon(Icons.edit, size: 48),
                )));
}
```



RadioListTile

We can change the position of the Radio button by using `controlAffinity` parameter.

```
Widget build(BuildContext context) {
  return MaterialApp(
    home: Scaffold(
      appBar: AppBar(title: Text("Title")),
      body: RadioListTile<String>(
        value: "2",
        onChanged: (b) => setState(() { s = "2", });
        groupValue: s,
        title: Text("Option no 2"),
        subtitle: Text("What option no 2 means"),
        secondary: Icon(Icons.edit, size: 48),
        controlAffinity: ListTileControlAffinity.trailing,
      )));
}
```



RadioListTile

Let's see how we can use a RadioListTile with an enum

```
enum Transport { Bike, Taxi, Airplane }
```

```
class MyAppState extends State<MyApp> {  
    Transport? transport;
```

We will also create a local variable of `Transport` type

Let's consider
`Transport` enum
with 3 values.

```
}
```

RadioListTile

Let's see how we can use a RadioListTile with an enum

```
enum Transport { Bike, Taxi, Airplane }

class MyAppState extends State<MyApp> {
    Transport? transport;
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(appBar: AppBar(title: Text("Transport Selection")),
                body: Column(children: [
                    RadioListTile<Transport>(...),
                    RadioListTile<Transport>(...),
                    RadioListTile<Transport>(...)])));
    }
}
```

The diagram shows a callout arrow pointing from the first `RadioListTile` in the code snippet to its corresponding detailed definition. The detailed definition is enclosed in a box and includes the `value`, `onChanged`, `groupValue`, `title`, `subtitle`, `secondary`, and `controlAffinity` properties.

```
RadioListTile<Transport> (
    value: Transport.Bike,
    onChanged: (b) => setState(() {
        transport = Transport.Bike;
    }),
    groupValue: transport,
    title: Text("Use a bike"),
    subtitle: Text("More exercise for you"),
    secondary: Icon(Icons.bike_scooter, size: 48),
    controlAffinity: ListTileControlAffinity.trailing,
)
```

RadioListTile

Let's see how we can use a RadioListTile with an enum

```
enum Transport { Bike, Taxi, Airplane }

class MyAppState extends State<MyApp> {
    Transport? transport;
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(appBar: AppBar(title: ...
                body: Column(children: [
                    RadioListTile<Transport>(...),
                    RadioListTile<Transport>(...),
                    RadioListTile<Transport>(...),
                ])));
    }
}
```

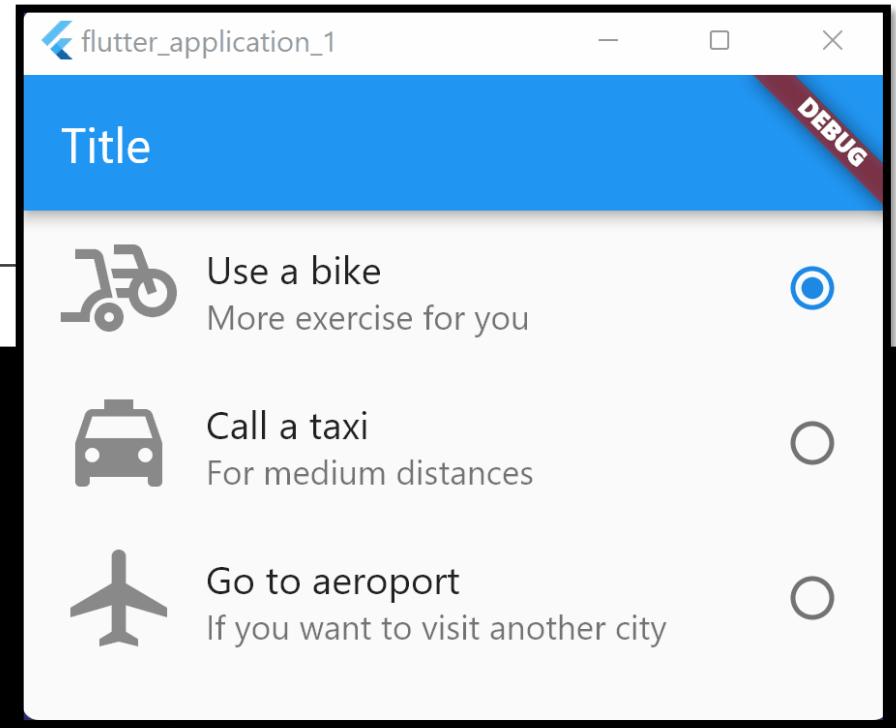
```
RadioListTile<Transport> (
    value: Transport.Taxi,
    onChanged: (b) => setState(() {
        transport = Transport.Taxi;
    }),
    groupValue: transport,
    title: Text("Call a taxi"),
    subtitle: Text("For medium distances"),
    secondary: Icon(Icons.local_taxi, size: 48),
    controlAffinity: ListTileControlAffinity.trailing,
)
```

RadioListTile

Let's see how we can use a RadioListTile with an enum

```
enum Transport { Bike, Taxi, Airplane }

class MyAppState extends State<MyApp> {
    Transport? transport;
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(appBar: AppBar(title:
                body: Column(children: [
                    RadioListTile<Transport>(...),
                    RadioListTile<Transport>(...),
                    RadioListTile<Transport>(...)]));
    }
}
```



```
RadioListTile<Transport> (
    value: Transport.Airplane,
    onChanged: (b) => setState(() {
        transport = Transport.Airplane;
    }),
    groupValue: transport,
    title: Text("Go to aeroport"),
    subtitle: Text("If you want to visit another city"),
    secondary: Icon(Icons.airplanemode_active,
        size: 48),
    controlAffinity: ListTileControlAffinity.trailing,
)
```

RadioListTile

Make sure that you put different values for all RadioListTiles in a group ! If you don't you might get to check on and end up having 2 or more selected !

```
class MyAppState extends State<MyApp> {
    int opt = 1;
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text('')),
                body: Column(children: [
                    RadioListTile<int>(...),
                    RadioListTile<int>(...),
                    RadioListTile<int>(...),
                ])));
    }
}
```



```
RadioListTile<int>(
    value: 1,
    onChanged: (b) => setState(() { opt = 1; }),
    groupValue: opt,
    title: Text("Option 1"),
)
```

RadioListTile

Make sure that you put different values for all RadioListTiles in a group ! If you don't you might get to check on and end up having 2 or more selected !

```
class MyAppState extends State<MyApp> {
    int opt = 1;
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("RadioListTile Example")),
                body: Column(children: [
                    RadioListTile<int>(...),
                    RadioListTile<int>(...), // This line is highlighted with an orange box
                    RadioListTile<int>(...),
                ])));
    }
}
```

The correct value should have been 2 and not 1

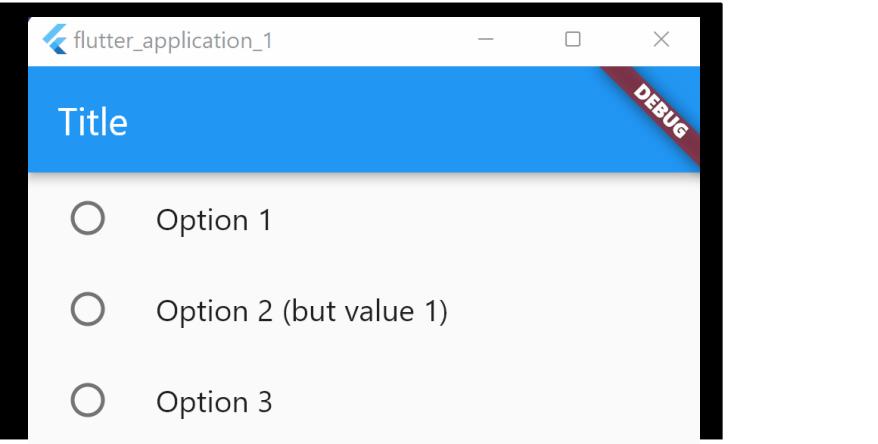
```
RadioListTile<int>(
    value: 1,
    onChanged: (b) => setState(() { opt = 2; }),
    groupValue: opt,
    title: Text("Option 2 (but value 1)"),
)
```

RadioListTile

Make sure that you put different values for all RadioListTiles in a group ! If you don't you might get to check on and end up having 2 or more selected !

```
class MyAppState extends State<MyApp> {
    int opt = 1;
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("Title")),
                body: Column(children: [
                    RadioListTile<int>(...),
                    RadioListTile<int>(...),
                    RadioListTile<int>(...),
                ])));
    }
}
```

```
RadioListTile<int>(
    value: 3,
    onChanged: (b) => setState(() { opt = 3; }),
    groupValue: opt,
    title: Text("Option 3"),
)
```



ListTile

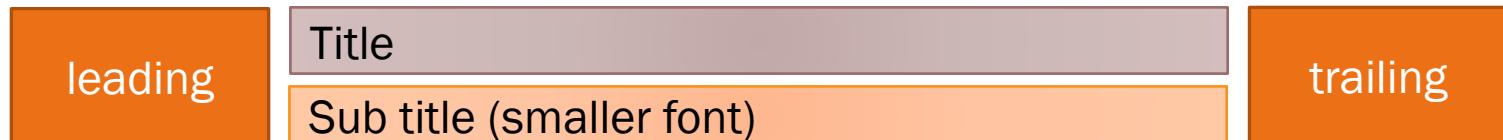
ListTile

A **ListTile** is a widget that can act like a button (an item) in a list of items.

It is in particular useful if you want to create a list of elements that contain several actions. List tiles (as a concept) is used with another widgets such as: Switch, Checkbox, Radiobox.

As a general format, a list tile is form out of:

- A leading widget
- A trailing widger
- A title
- A sub-title



ListTile

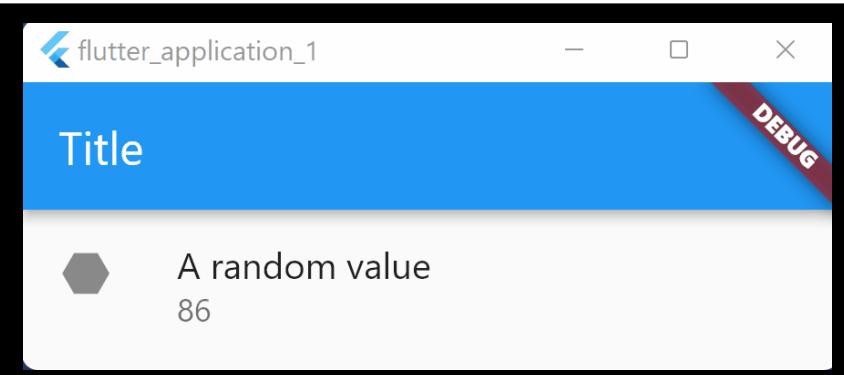
Constructor

```
ListTile ({  
    GestureTapCallback? onTap,  
    GestureLongPressCallback? onLongPress,  
    Widget? leading,  
    Widget? title,  
    Widget? subtitle,  
    Widget? trailing,  
    Color? iconColor,  
    Color? tileColor,  
    Color? selectedTileColor,  
    Color? selectedColor,  
    Color? hoverColor,  
    ... })
```

ListTile

A simple example for Random numbers generation

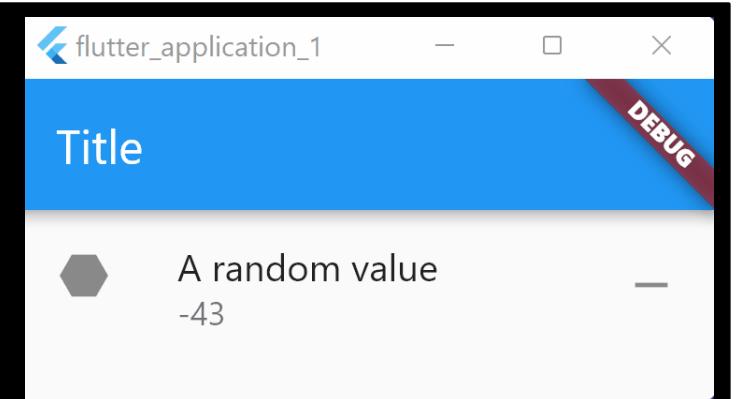
```
class MyAppState extends State<MyApp> {
    int rand = Random().nextInt(100);
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("Title")),
                body: ListTile(
                    title: Text("A random value"),
                    subtitle: Text("$rand"),
                    onTap: () => setState(() { rand = Random().nextInt(100); }),
                    leading: Icon(Icons.hexagon),
                )));
    }
}
```



ListTile

Let's build a slightly more complex example (one that generates a number between `-50` and `+49` and shows an icon with a plus (if the number is positive) or a minus if the number is negative.

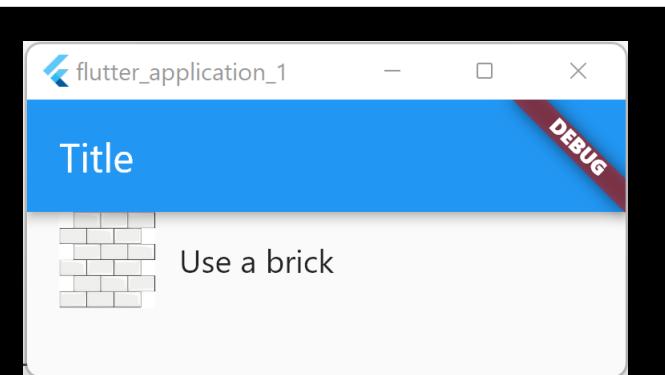
```
Widget build(BuildContext context) {
  return MaterialApp(
    home: Scaffold(
      appBar: AppBar(title: Text("Title")),
      body: ListTile(
        title: Text("A random value"),
        subtitle: Text("$rand"),
        onTap: () => setState(() { rand = Random().nextInt(100) - 50; }),
        leading: Icon(Icons.hexagon),
        trailing: Icon(rand >= 0 ? Icons.add : Icons.remove),
        hoverColor: Colors.amber,
      )));
}
```



ListTile

Images can also be used (instead of icons). The next example assumes that tiles.jpg is located in “assets/images” and it is also added in pubspec.yaml file.

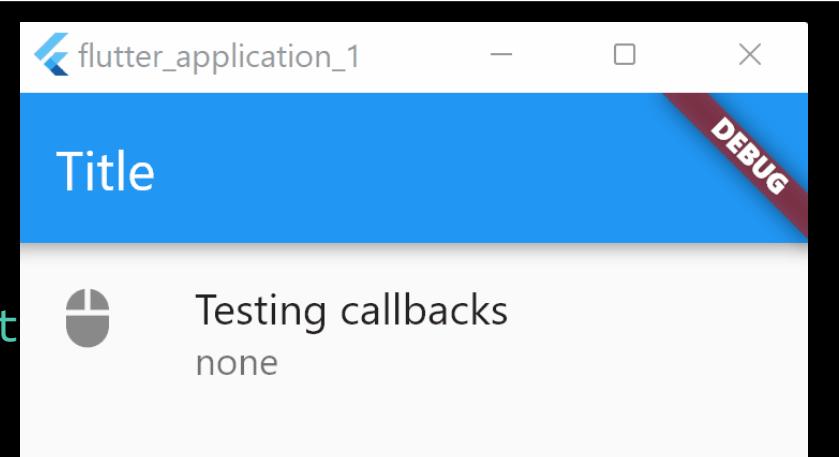
```
class MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("Title")),
        body: ListTile(
          title: Text("Use a brick"),
          onTap: () => {},
          leading: Image.asset("assets/images/tiles.jpg"),
          hoverColor: Colors.amber,
        )));
  }
}
```



ListTile

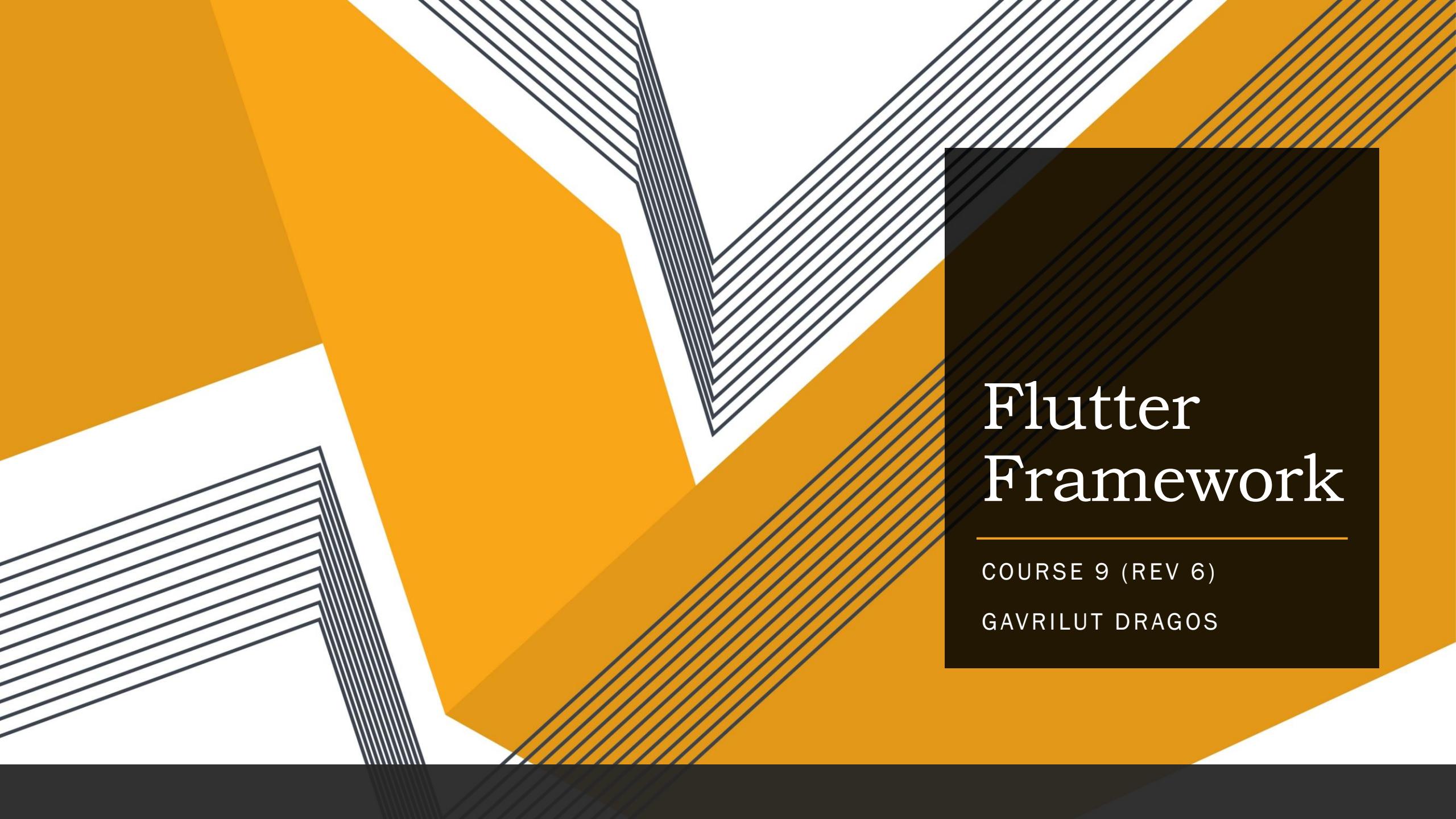
A simple example to test `onTap` and `onLongPress` callbacks.

```
class MyAppState extends State<MyApp> {
  String msg = "none";
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold( appBar: AppBar(title: Text("Title")),
        body: ListTile(
          title: Text("Testing callbacks"),
          subtitle: Text(msg),
          onTap: () => setState(() { msg = "onTap"; }),
          onLongPress: () => setState(() { msg = "onLongPress"; }),
          leading: Icon(Icons.mouse),
          hoverColor: Colors.lightBlue,
        )));
}
```



Q & A



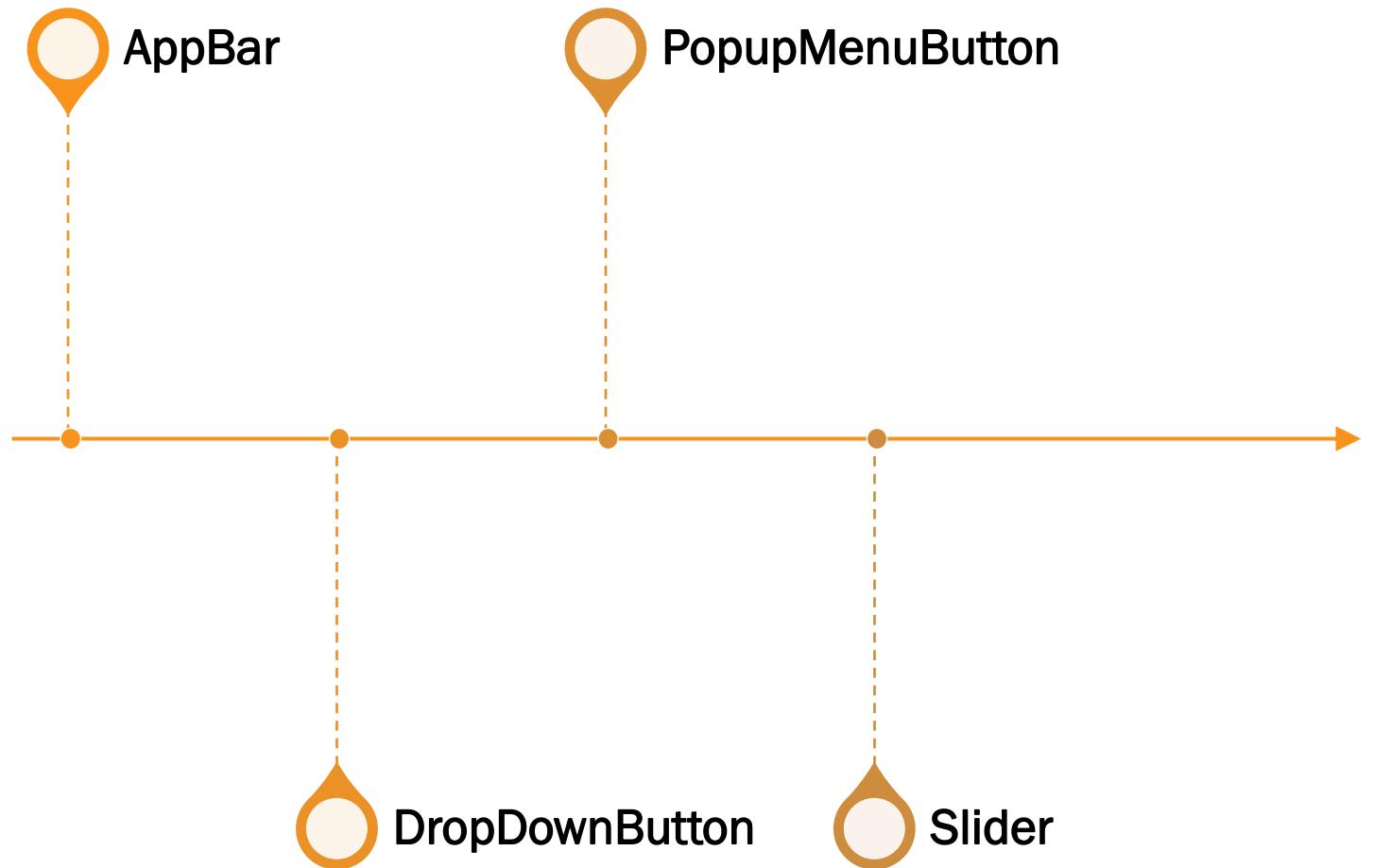


Flutter Framework

COURSE 9 (REV 6)

GAVRILUT DRAGOS

Agenda

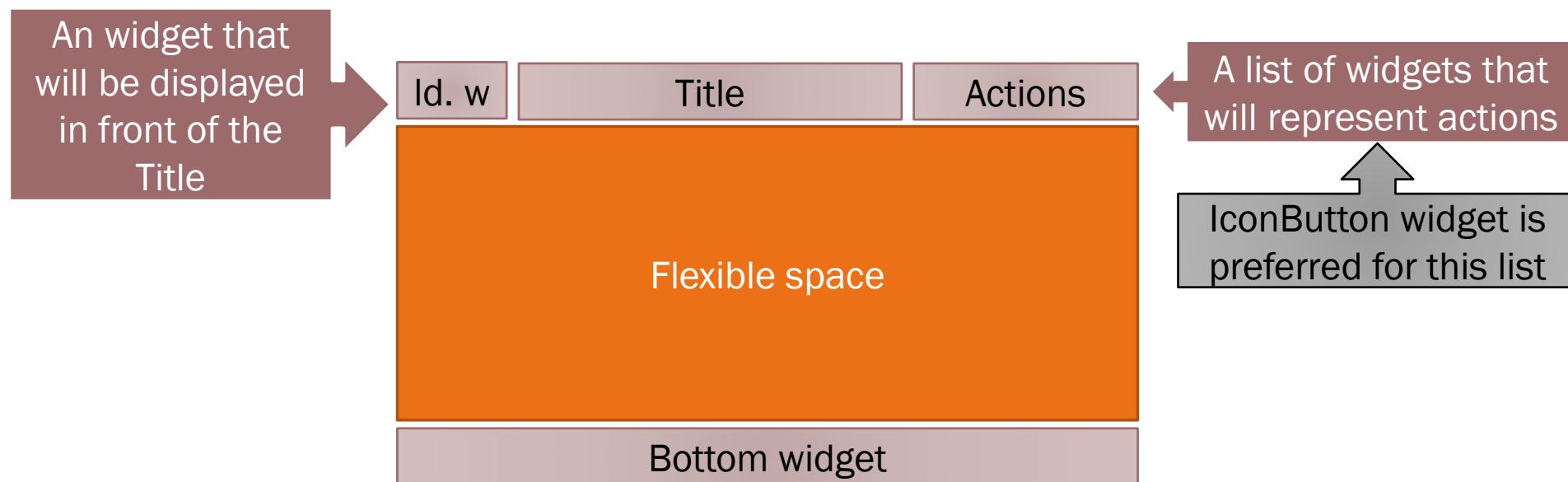


AppBar

AppBar

AppBar component is the main component of a Scaffold (and it holds a form of organization for the current view).

An AppBar has some components:



AppBar

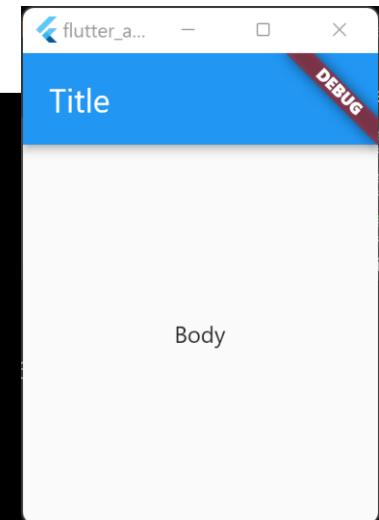
Constructor:

<pre>AppBar({ Widget? leading, Widget? title, Widget? flexibleSpace, PreferredSizeWidget? bottom, List<Widget>? actions, Color? foregroundColor, Color? backgroundColor, ... })</pre>	<p>The leading widget Title The flexible space widget The bottom widget A list of widgets for actions</p>
--	---

AppBar

Let's see a very simple example.

```
import 'package:flutter/material.dart';
void main() => runApp(MyApp());
class MyApp extends StatefulWidget {
  @override
  State<MyApp> createState() => MyAppState();
}
class MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("Title")),
        body: Center(child: Text("Body")));
  }
}
```

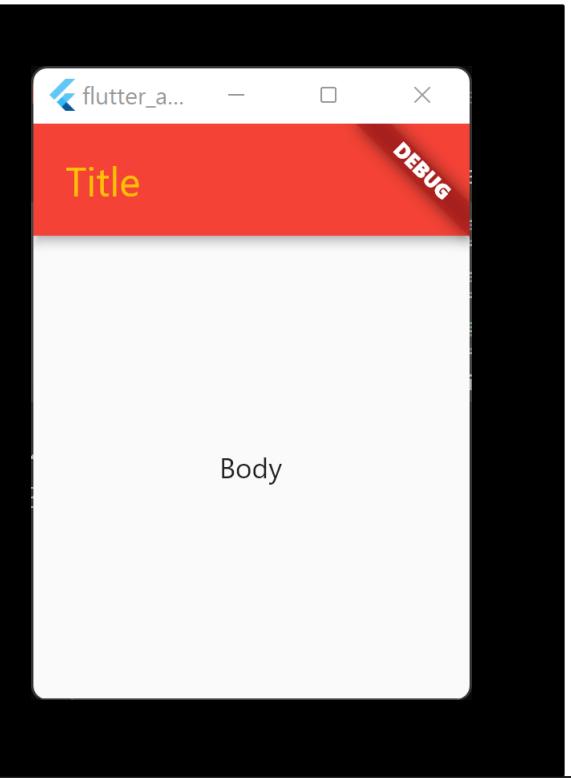


For the next part of this course, we will focus on the build method and different parameters that can be set up for the AppBar

AppBar

To set up the background and title text color use the `.foregroundColor` and `.backgroundColor` properties.

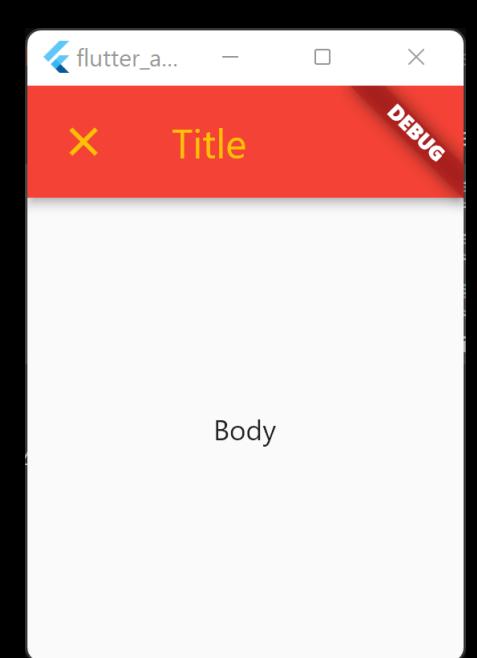
```
class MyAppState extends State<MyApp> {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar(  
          title: Text("Title"),  
          foregroundColor: Colors.amber,  
          backgroundColor: Colors.red,  
        ),  
        body: Center(child: Text("Body"))));  
  }  
}
```



AppBar

The “leading” widget is usually a close button (however, it can be any widget).

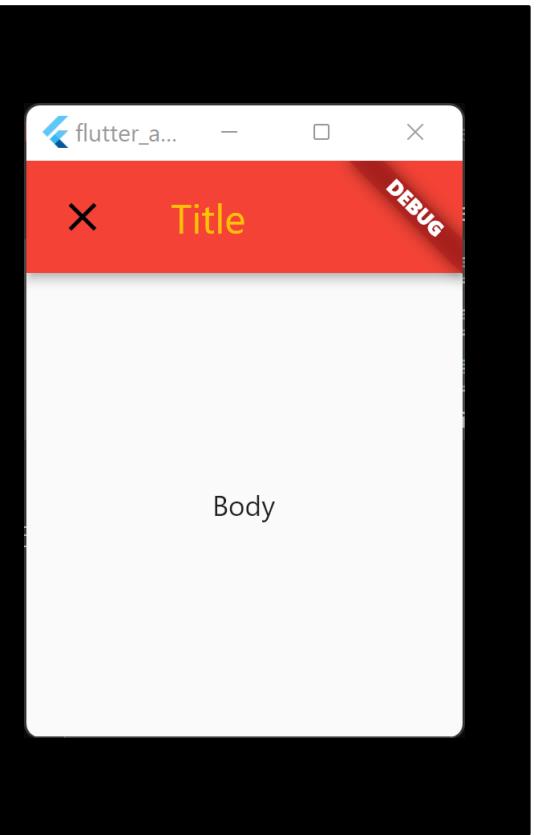
```
class MyAppState extends State<MyApp> {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar(  
          title: Text("Title"),  
          foregroundColor: Colors.amber,  
          backgroundColor: Colors.red,  
          leading: CloseButton(),  
        ),  
        body: Center(child: Text("Body"))));  
  }  
}
```



AppBar

The “leading” widget is usually a close button (however, it can be any widget).

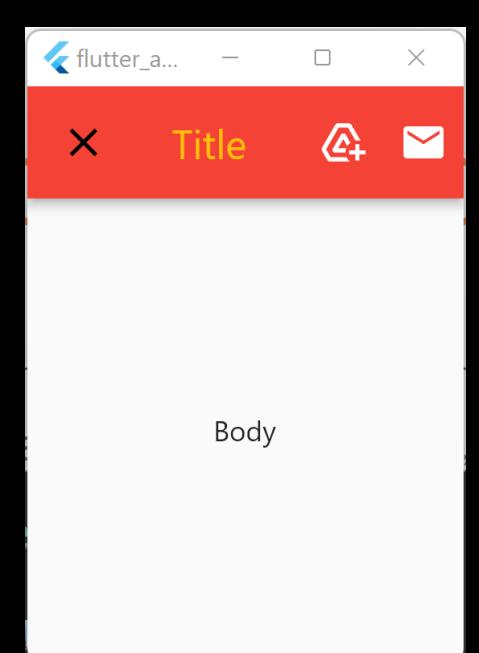
```
class MyAppState extends State<MyApp> {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar(  
          title: Text("Title"),  
          foregroundColor: Colors.amber,  
          backgroundColor: Colors.red,  
          leading: CloseButton(color: Colors.black),  
        ),  
        body: Center(child: Text("Body"))));  
  }  
}
```



AppBar

The “leading” widget is usually a close button (however, it can be any widget).

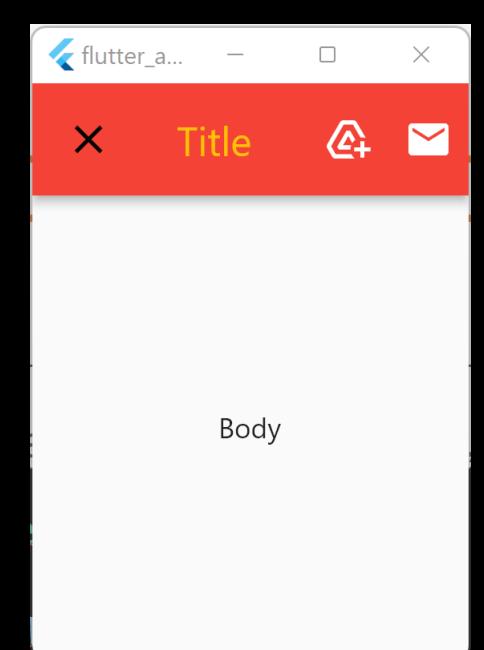
```
Widget build(BuildContext context) {
  return MaterialApp(
    debugShowCheckedModeBanner: false,
    home: Scaffold(
      appBar: AppBar(
        title: Text("Title"),
        foregroundColor: Colors.amber,
        backgroundColor: Colors.red,
        leading: CloseButton(color: Colors.black),
        actions: [ IconButton(...), IconButton(...) ],
      ),
      body: Center(child: Text("Body")));
}
```



AppBar

The “leading” widget is usually a close button (however, it can be any widget).

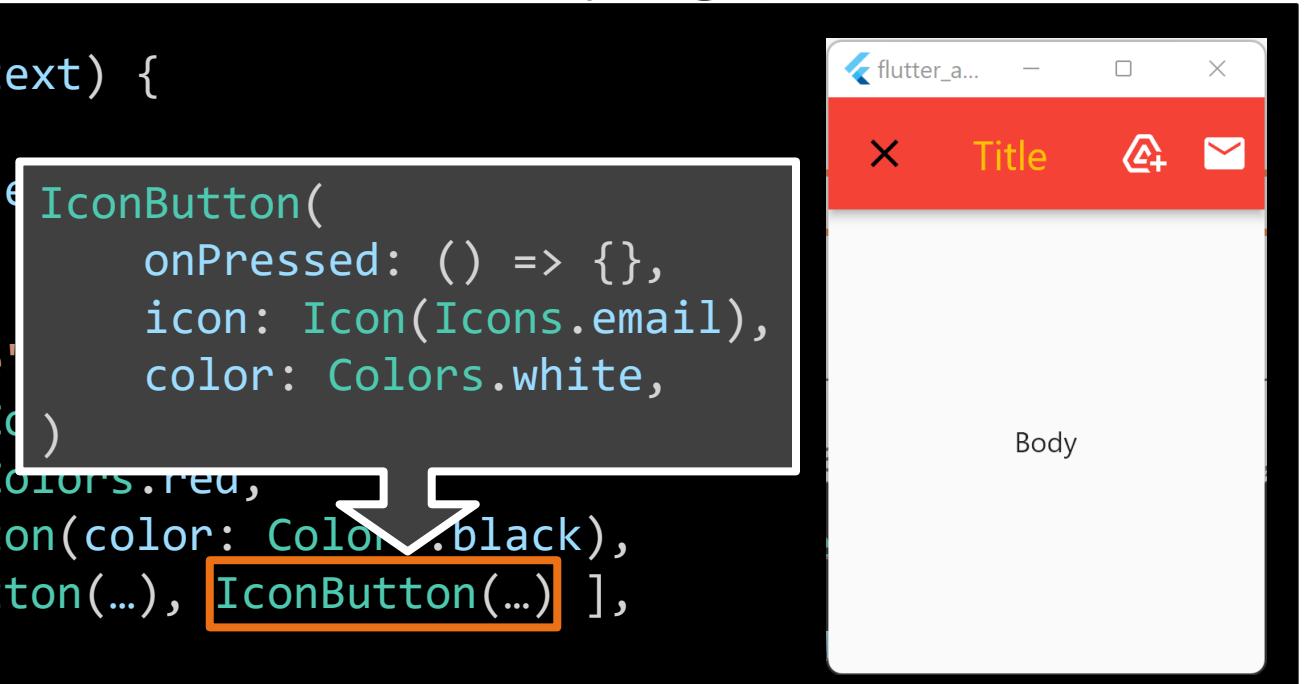
```
Widget build(BuildContext context) {
  return MaterialApp(
    debugShowCheckedModeBanner: false,
    home: Scaffold(
      appBar: AppBar(
        title: Text("Title"),
        actions: [
          IconButton(onPressed: () {}, icon: Icon(Icons.add_to_drive), color: Colors.white),
          IconButton(onPressed: () {}, icon: Icon(Icons.add_to_drive), color: Colors.white),
        ],
        backgroundColor: Colors.red,
        leading: CloseButton(color: Colors.black),
      ),
      body: Center(child: Text("Body")));
}
```



AppBar

The “leading” widget is usually a close button (however, it can be any widget).

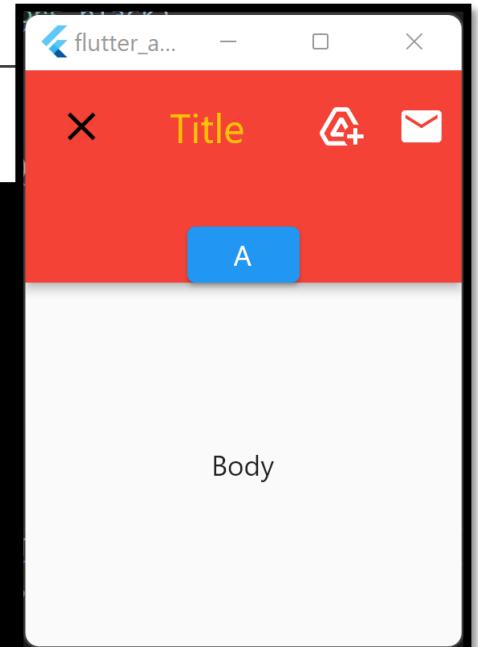
```
Widget build(BuildContext context) {  
    return MaterialApp(  
        debugShowCheckedModeBanner: false,  
        home: Scaffold(  
            appBar: AppBar(  
                title: Text("Title"),  
                foregroundColor: Colors.white,  
                backgroundColor: Colors.red,  
                leading: CloseButton(color: Colors.black),  
                actions: [ IconButton(...), IconButton(...) ],  
            ),  
            body: Center(child: Text("Body"))));  
}
```



AppBar

The “bottom” widget can be used to increase the size of the AppBar

```
Widget build(BuildContext context) {
  return MaterialApp(
    debugShowCheckedModeBanner: false,
    home: Scaffold(appBar: AppBar(
      title: Text("Title"),
      foregroundColor: Colors.amber,
      backgroundColor: Colors.red,
      leading: CloseButton(color: Colors.black),
      actions: [ IconButton(...), IconButton(...) ],
      bottom: PreferredSize(
        child: ElevatedButton(child: Text("A"), onPressed:()=>{}),
        preferredSize: Size(300, 50)),
    ),
    body: Center(child: Text("Body"))));
}
```



DropdownButton

DropdownButton

A dropdown button (or combobox) is a widget that allows you to choose from several existing options.

Just like in the case of Radio buttons, a dropdown button is based on a template that reflects the selection. Because of this, it is very useful for enums.

There are two classes relevant for this widget:

1. `DropdownButton` (the actual button)
2. `DropdownMenuItem` (the actual item, also a template that holds the value and the widget that reflects that value)

DropdownButton

Constructors:

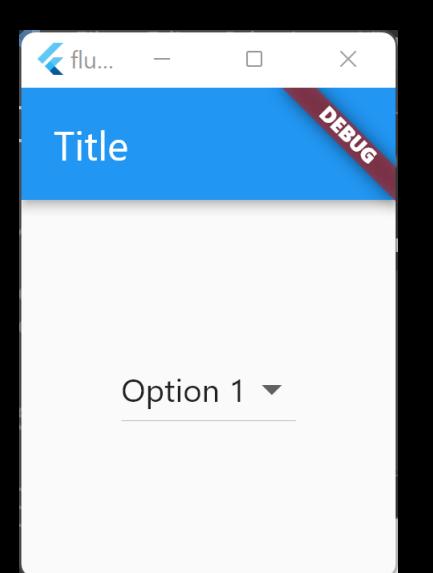
```
DropdownButton<T>({  
    Widget? hint,  
    Widget? disabledHint,  
    Widget? underline,  
    Widget? icon,  
    double iconSize = 24.0,  
    T? value,  
    required List<DropdownMenuItem<T>>? items,  
    required void Function(T?)? onChanged,  
    Color? iconDisabledColor,  
    Color? iconEnabledColor,  
    Color? dropdownColor,  
    Color? focusColor,  
    ... } )
```

```
DropdownMenuItem<T>({  
    required Widget child,  
    T? value,  
    bool enabled = true,  
    AlignmentGeometry alignment = ,  
        AlignmentGeometry.centerStart  
    ... }  
)
```

DropdownButton

A very simple example.

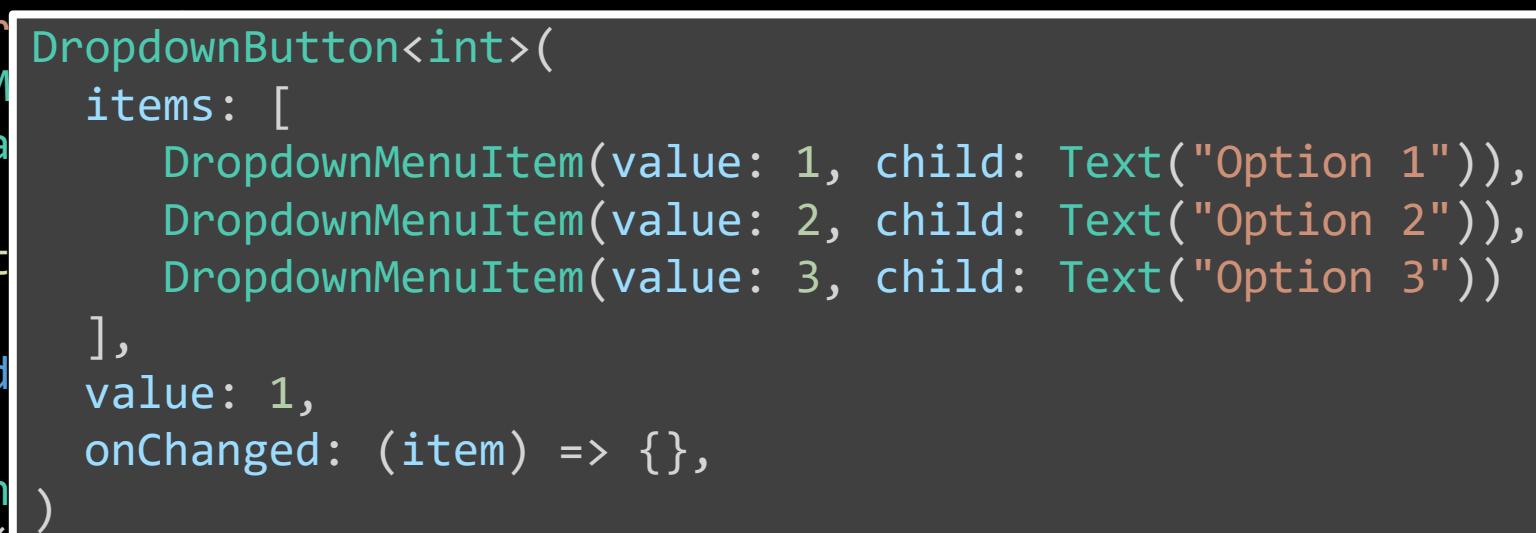
```
import 'package:flutter/material.dart';
void main() => runApp(MyApp());
class MyApp extends StatefulWidget {
  @override
  State<MyApp> createState() => MyAppState();
}
class MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("Title")),
        body: Center(child: DropdownButton<int>(...)));
  }
}
```



DropdownButton

A very simple example.

```
import 'package:flutter/material.dart';
void main() => runApp(MyApp());
class MyApp extends StatelessWidget {
  @override
  State<MyApp> createState() => _MyAppState();
}
class _MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("Title")),
        body: Center(child: DropdownButton<int>(
          items: [
            DropdownMenuItem(value: 1, child: Text("Option 1")),
            DropdownMenuItem(value: 2, child: Text("Option 2")),
            DropdownMenuItem(value: 3, child: Text("Option 3"))
          ],
          value: 1,
          onChanged: (item) => {},
        )));
  }
}
```

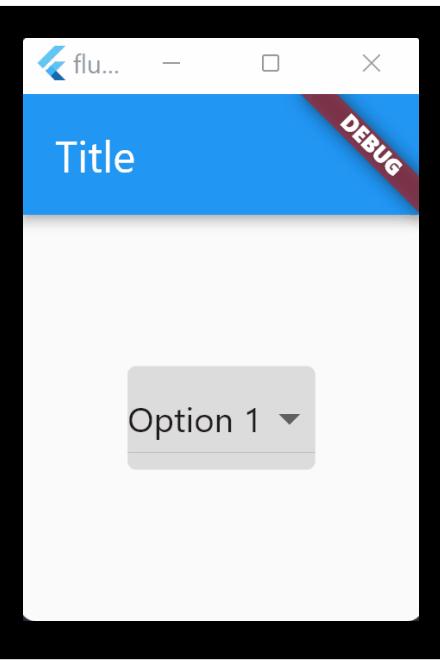


DropdownButton

To make a drop-down button work, the following steps must be followed:

1. Create a variable that will store the selected value (in our care it variable `v` of type `int`)
2. Make sure that the property `value` from `DropdownButton` ctor is set to variable `v`
3. In `onChange` callback use `setState` to copy the selected value into variable `v`

```
class MyAppState extends State<MyApp> {  
    int? v = 1;  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            home: Scaffold(  
                appBar: AppBar(title: Text("Title")),  
                body: Center(  
                    child: DropdownButton<int>(...),  
                )));  
    } }
```



DropdownButton

To make a drop-down button work, the following steps must be followed:

1. Create a variable
2. Make sure to import `DropdownButton`
3. Implement `onChanged`

```
class MyAppState extends State<MyApp> {  
    int? v = 1;  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            home: Scaffold(  
                appBar: AppBar(title: Text("Dropdown Button Example")),  
                body: Center(  
                    child: DropdownButton<int>(...),  
                ))));  
    } }
```

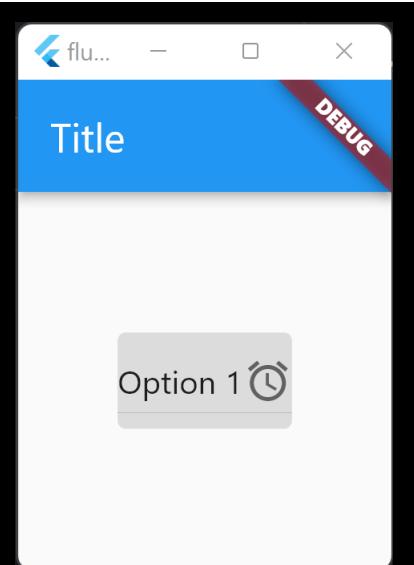
```
DropdownButton<int>(  
    items: [  
        DropdownMenuItem(value: 1, child: Text("Option 1")),  
        DropdownMenuItem(value: 2, child: Text("Option 2")),  
        DropdownMenuItem(value: 3, child: Text("Option 3"))  
    ],  
    value: v,  
    onChanged: (item) => setState(() { v = item; })
```



DropdownButton

To add an icon to the dropdown use the `icon` property.

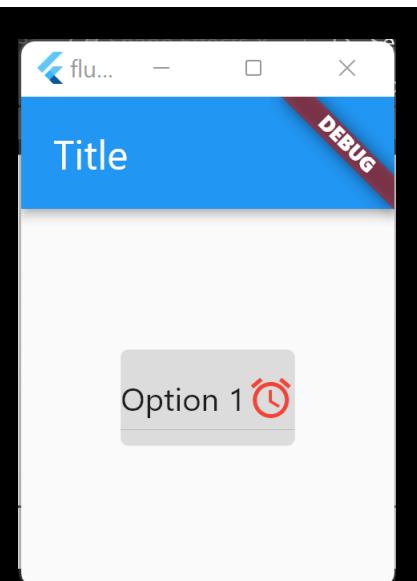
```
class MyAppState extends State<MyApp> {
    int? v = 1;
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("Title")),
                body: Center(
                    child: DropdownButton<int>(
                        items: [...],
                        value: v,
                        onChanged: (item) => setState(() { v = item; }),
                        icon: Icon(Icons.access_alarm))));}
}
```



DropdownButton

You can also set the enable and disable color for the icon:

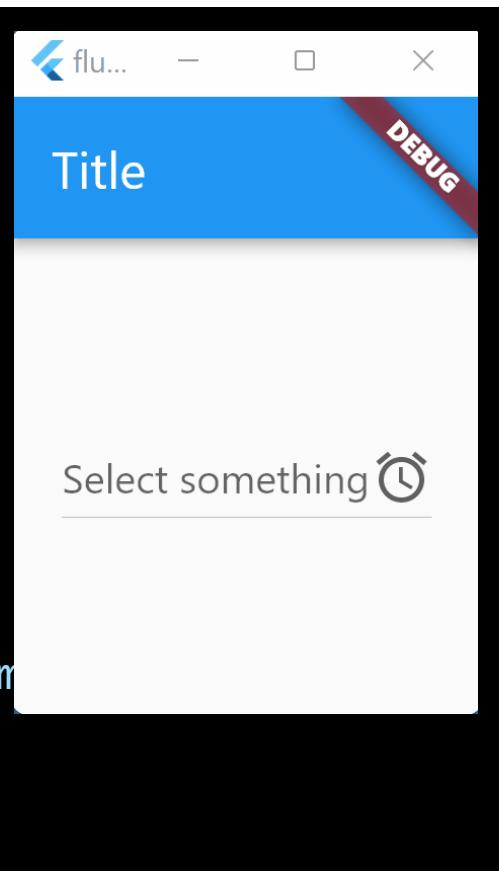
```
class MyAppState extends State<MyApp> {
    int? v = 1;
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("Title")),
                body: Center(
                    child: DropdownButton<int>(
                        items: [...],
                        value: v,
                        onChanged: (item) => setState(() { v = item; }),
                        icon: Icon(Icons.access_alarm),
                        iconEnabledColor: Colors.red)));
    }
}
```



DropdownButton

If value is `null`, a the widget is `enabled`, the widget from `hint` is used (if exists).

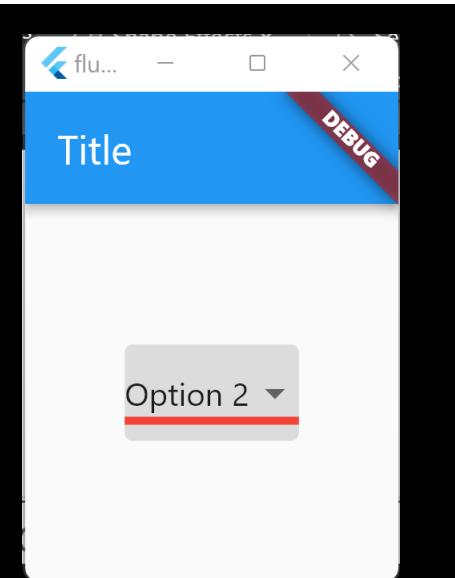
```
class MyAppState extends State<MyApp> {
    int? v = null;
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("Title")),
                body: Center(
                    child: DropdownButton<int>(
                        items: [...],
                        value: v,
                        onChanged: (item) => setState(() { v = item;
                        hint: Text("Select something"),
                        icon: Icon(Icons.access_alarm))))));
    }
}
```



DropdownButton

You can also use underline property to draw something under the widget

```
class MyAppState extends State<MyApp> {
    int? v = 1;
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("Title")),
                body: Center(
                    child: DropdownButton<int>(
                        items: [...],
                        value: v,
                        onChanged: (item) => setState(() { v = item; }),
                        underline: Container(height: 4, color: Colors.red),
                    )));
    }
}
```



DropdownButton

But what if we want to create something more complex.

For example, a dropdown menu that has a different icon for each item (and maybe different colors, or different picture, etc).

In this case, the child property for the DropdownMenuItem has to be a complex widget.



DropdownButton

So ... let's see a code that will create such a widget.

```
class MyAppState extends State<MyApp> {
    int? v = 1;
    DropdownMenuItem<int> CreateMenuItem(String text, int val, IconData iconData) {...}
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("Title")),
                body: Center(
                    child: DropdownButton<int>(
                        items: [...],
                        value: v,
                        onChanged: (item) => setState(() { v = item; })));
    }
}
```

DropdownButton

So ... let's see a code that will create such a widget.

```
class MyAppState extends State<MyApp> {
    int? v = 1;
    DropdownMenuItem<int> CreateMenuItem(String text, int val, IconData iconData) {...}
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                body: Center(
                    child: DropdownButton(
                        value: v,
                        child: Row(
                            children: [
                                Icon(iconData),
                                SizedBox(width: 10),
                                Text(text)
                            ]
                        )
                    )
                )
            )
        );
    }
}
```

DropdownButton

So ... let's see a code that will create such a widget.

```
class MyAppState extends State<MyApp> {
    int? v = 1;
    DropdownMenuItem<int> CreateMenuItem(String text, int val, IconData iconData) {...}

items: [
    CreateMenuItem("Start", 1, Icons.start),
    CreateMenuItem("Fast forward", 2, Icons.fast_forward),
    CreateMenuItem("Stop", 3, Icons.stop),
],
    body: Center(
        child: DropdownButton<int>(
            items: [...],
            value: v,
            onChanged: (item) => setState(() { v = item; })));
}
```

PopupMenuButton

PopupMenuButton

A popup menu button is similar to a DropdownButton, but it has no selection (meaning that after you click on an item, you will just receive a callback to notify you that a specific item has been clicked, but no change in the interface on how that button looks like).

Its main face can be represented by an icon or a child widget (but not both). If none is provided a default icon that looks like 3 vertical points () will be used.

This means that it is possible to make it look like a DropdownButton if you change its properties upon building.

PopupMenuButton is often used with the AppBar to show the default menu for that application.

PopupMenuButton

Constructor:

```
PopupMenuButton<T>({  
    Widget? child,  
    Widget? icon,  
    double? iconSize,  
    T? initialValue,  
    required List<PopupMenuEntry<T>> Function(BuildContext) itemBuilder,  
    void Function(T)? onSelected,  
    void Function()? onCancel,  
    String? toolTip,  
    Color? color,  
    ... }  
)
```

PopupMenuButton

Constructor:

```
PopupMenuButton<T>({  
    Widget? child,  
    Widget? icon,  
    double? iconSize,  
    T? initialValue,  
    required List<PopupMenuEntry<T>> Function(BuildContext) itemBuilder,  
    void Function(T)? onSelected,
```

A `PopupMenuEntry` is a base class for a menu item.

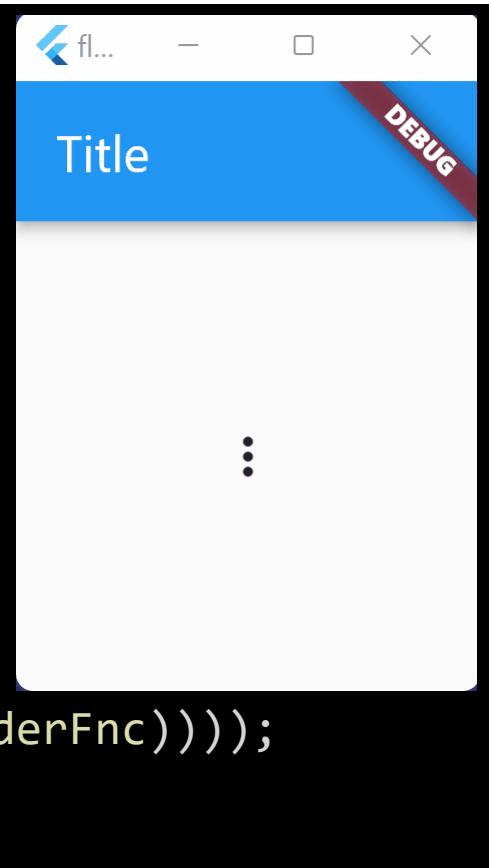
There are several classes that implements this class

- `PopupMenuEntry` → a menu item
- `CheckedPopupMenuEntry` → a menu item with checkmark
- `PopupMenuDivider` → a vertical line that separates two menu items

PopupMenuButton

So ... let's see a code that will create such a widget.

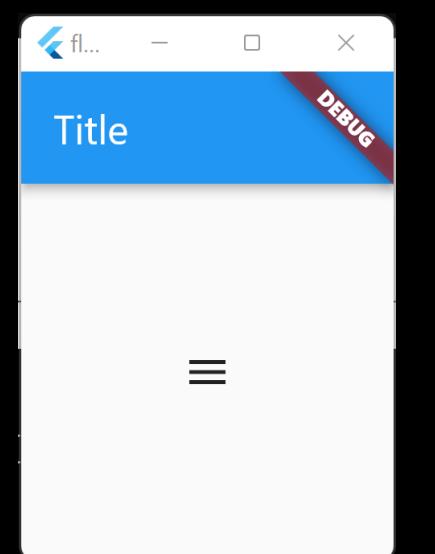
```
class MyAppState extends State<MyApp> {
    List<PopupMenuEntry<int>> itemBuilderFnc(context) => [
        PopupMenuItem<int>(value: 1, child: Text("opt 1")),
        PopupMenuItem<int>(value: 2, child: Text("opt 2")),
        PopupMenuItem<int>(value: 3, child: Text("opt 3"))
    ];
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("Title")),
                body: Center(
                    child: PopupMenuButton<int>(itemBuilder: itemBuilderFnc)));
    }
}
```



PopupMenuButton

We can change the default icon by using the `icon` property:

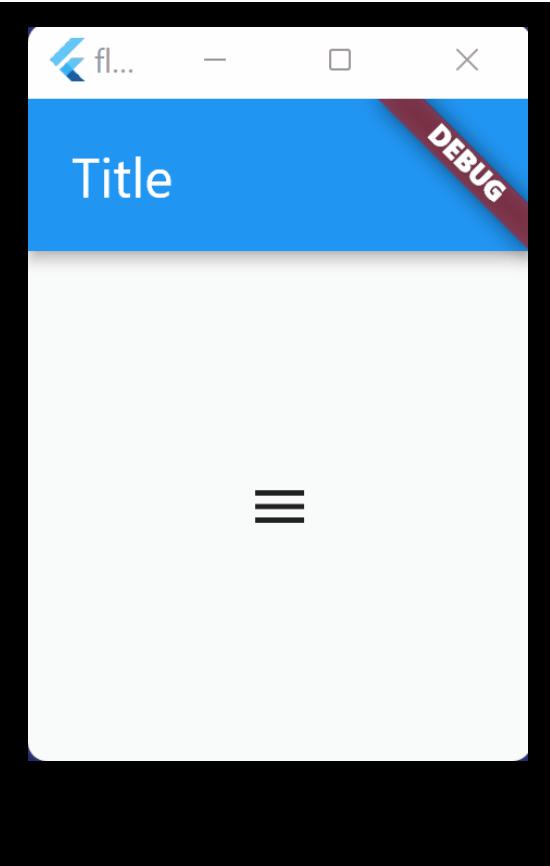
```
class MyAppState extends State<MyApp> {
    List<PopupMenuEntry<int>> itemBuilderFnc(context) => [...];
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("Title")),
                body: Center(
                    child: PopupMenuButton<int>(
                        itemBuilder: itemBuilderFnc,
                        icon: Icon(Icons.menu))});
    }
}
```



PopupMenuButton

We can also change the color of the menu with the `color` property:

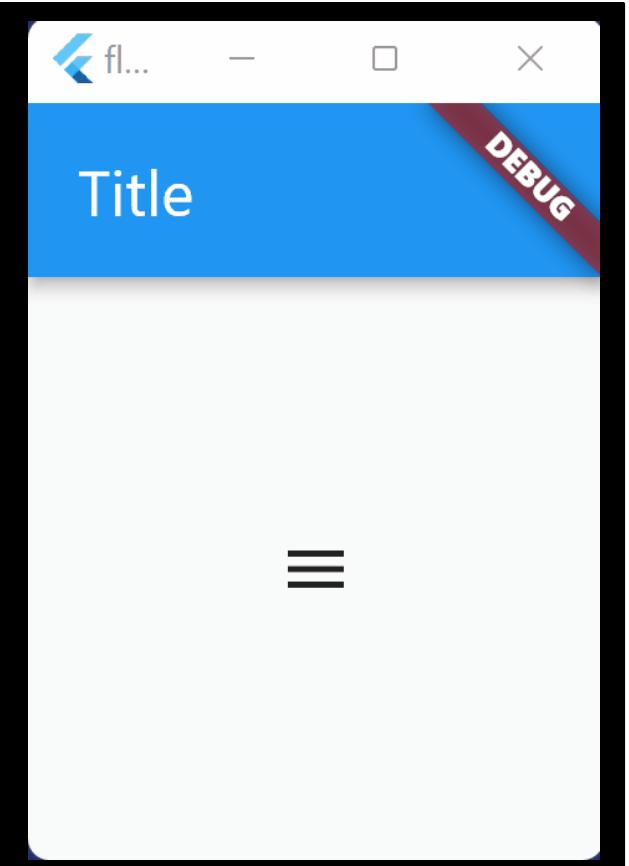
```
class MyAppState extends State<MyApp> {
  List<PopupMenuEntry<int>> itemBuilderFnc(context) => [...];
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("Title")),
        body: Center(
          child: PopupMenuButton<int>(
            itemBuilder: itemBuilderFnc,
            icon: Icon(Icons.menu),
            color: Colors.lightBlue,
          ))));
  }
}
```



PopupMenuButton

Or, we can use the `toolTip` property to set a new toolTip (the default one is Show menu):

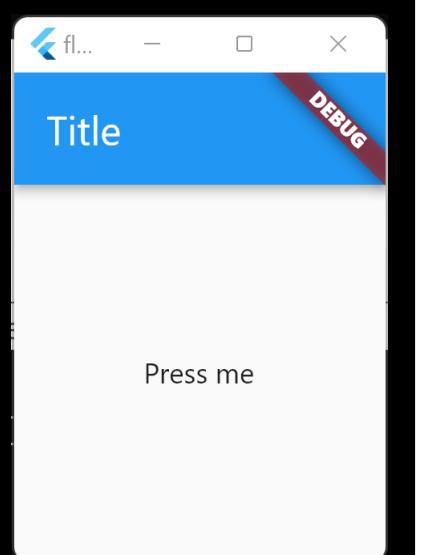
```
class MyAppState extends State<MyApp> {
    List<PopupMenuEntry<int>> itemBuilderFnc(context) => [...];
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text("Title")),
                body: Center(
                    child: PopupMenuButton<int>(
                        itemBuilder: itemBuilderFnc,
                        icon: Icon(Icons.menu),
                        tooltip: "Press to see a menu",
                    ))));
    }
}
```



PopupMenuButton

You can also use the `child` property if you want to set up a text or something different than an icon

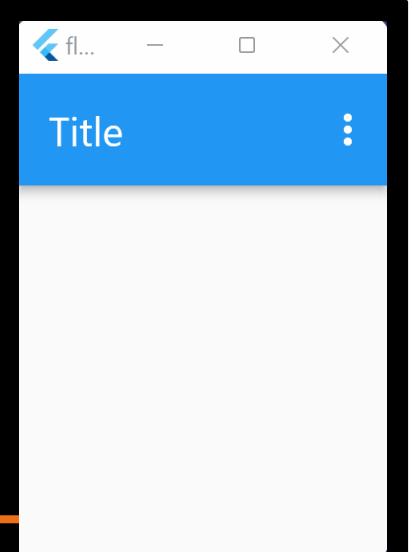
```
class MyAppState extends State<MyApp> {
  List<PopupMenuEntry<int>> itemBuilderFnc(context) => [...];
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("Title")),
        body: Center(
          child: PopupMenuButton<int>(
            itemBuilder: itemBuilderFnc,
            child: Text("Press me"),
          ))));
  }
}
```



PopupMenuButton

Usually, a PopupMenuItem is being used with the AppBar as one of its actions button:

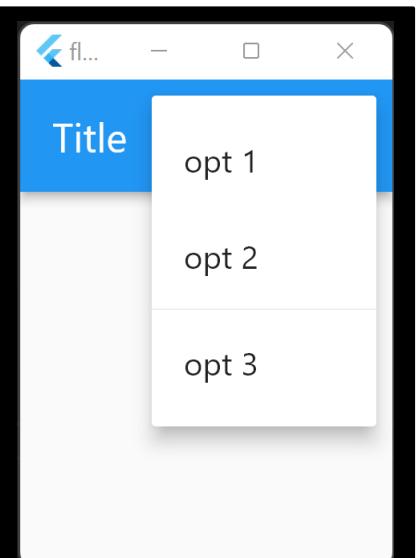
```
class MyAppState extends State<MyApp> {
    List<PopupMenuEntry<int>> itemBuilderFnc(context) => [...];
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            debugShowCheckedModeBanner: false,
            home: Scaffold(
                appBar: AppBar(
                    title: Text("Title"),
                    actions: [
                        PopupMenuItem<int>(itemBuilder: itemBuilderFnc),
                    ],
                )));
    }
}
```



PopupMenuButton

For the menu, we can use a PopupMenuItem to separate different items from the menu (in our case, “opt 1” and “opt 2” from “opt 3”)

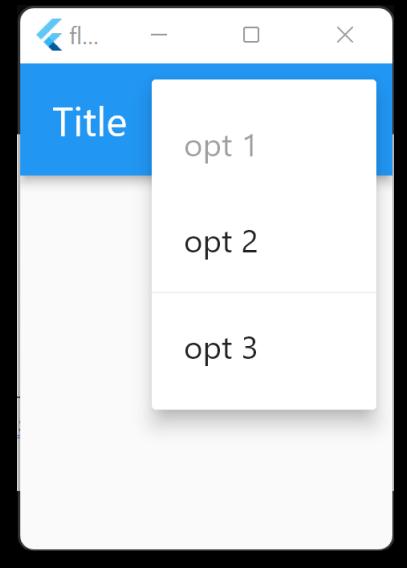
```
class MyAppState extends State<MyApp> {
    List<PopupMenuEntry<int>> itemBuilderFnc(context) => [
        PopupMenuItem<int>(value: 1, child: Text("opt 1")),
        PopupMenuItem<int>(value: 2, child: Text("opt 2")),
        PopupMenuDivider(height: 5),
        PopupMenuItem<int>(value: 3, child: Text("opt 3"))
    ];
    @override
    Widget build(BuildContext context) {...}
}
```



PopupMenuButton

For PopuMenuItem instances, there is a property called enable that can be use to disable or enable an item.

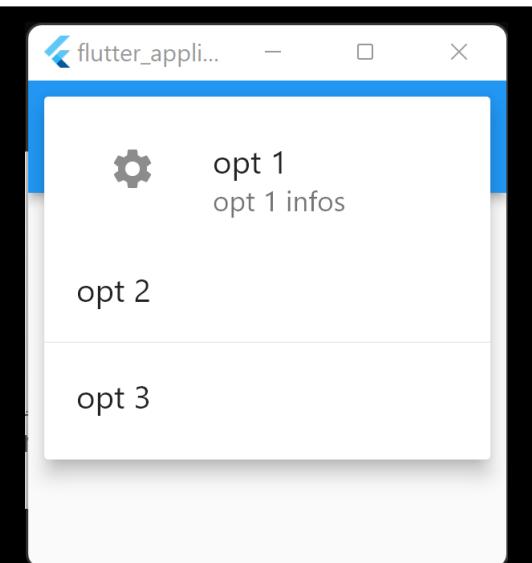
```
class MyAppState extends State<MyApp> {
  List<PopupMenuEntry<int>> itemBuilderFnc(context) => [
    PopupMenuItem<int>(value: 1, enabled: false, child: Text("opt 1")),
    PopupMenuItem<int>(value: 2, child: Text("opt 2")),
    PopupMenuDivider(height: 5),
    PopupMenuItem<int>(value: 3, child: Text("opt 3"))
  ];
  @override
  Widget build(BuildContext context) {...}
}
```



PopupMenuButton

To create a more complex menu item (for example something with an icon) one can use a Row or a ListTile for the child property.

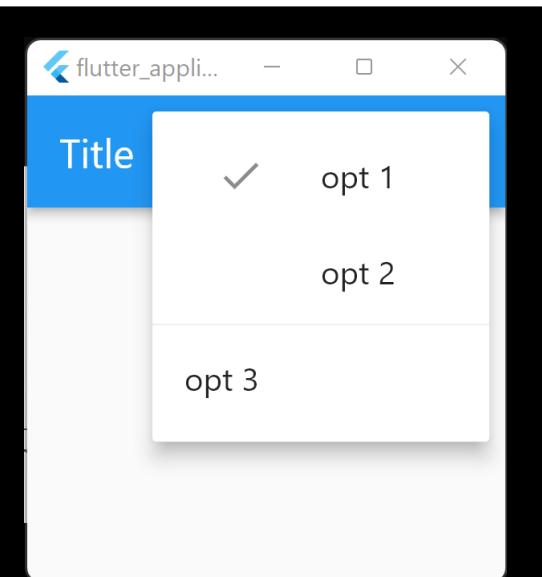
```
List<PopupMenuEntry<int>> itemBuilderFnc(context) => [
    PopupMenuItem<int>(
        value: 1,
        child: ListTile(
            leading: Icon(Icons.settings),
            title: Text("opt 1"),
            subtitle: Text("opt 1 infos"),
        )),
    PopupMenuItem<int>(value: 2, child: Text("opt 2")),
    PopupMenuDivider(height: 5),
    PopupMenuItem<int>(value: 3, child: Text("opt 3"))
];
```



PopupMenuButton

And you can use a `CheckedPopupMenuItem` if some of the items from your menu should be checked or not (via checked property).

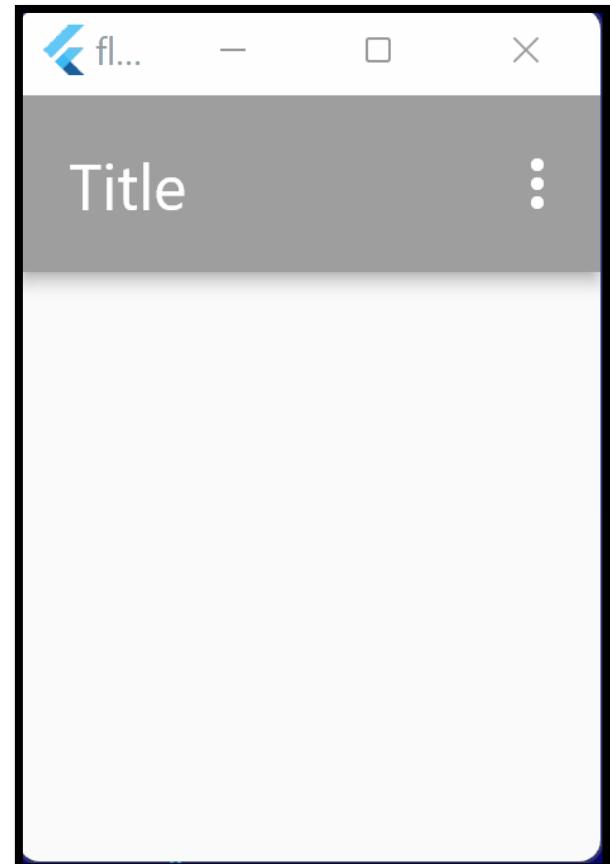
```
List<PopupMenuEntry<int>> itemBuilderFnc(context) => [
    CheckedPopupMenuItem<int>(
        value: 1,
        checked: true,
        child: Text("opt 1")),
    CheckedPopupMenuItem<int>(
        value: 2,
        checked: false,
        child: Text("opt 2")),
    PopupMenuDivider(height: 5),
    PopupMenuItem<int>(value: 3, child: Text("opt 3"))
];
```



PopupMenuButton

Let's see a more complex example where we have a menu with 3 options (red, green and blue) that if pressed will change the color of the AppBar accordingly.

The initial color of the AppBar should be different (e.g. a grey)



PopupMenuButton

Step 1 → create two variables (“c” for the actual color, and cList – an array with red, green and blue)

```
class MyAppState extends State<MyApp> {  
  var cList = [Colors.red, Colors.green, Colors.blue];  
  Color c = Colors.grey;  
  
}  
}
```

PopupMenuButton

Step 2 → create the itemBuilderFnc to be used when constructing a PopupMenuItem

```
class MyAppState extends State<MyApp> {  
  var cList = [Colors.red, Colors.green, Colors.blue];  
  Color c = Colors.grey;  
  List<PopupMenuEntry<int>> itemBuilderFnc(context) => [  
    PopupMenuItem<int>(value: 0, child: Text("Red")),  
    PopupMenuItem<int>(value: 1, child: Text("Green")),  
    PopupMenuItem<int>(value: 2, child: Text("Blue"))  
};
```

↑
Notice that the value of each
item is an index in **cList**
array to the color it
represents.

```
}
```

PopupMenuButton

Step 3 → create the build method with an AppBar that uses “c” variable for its background color.

```
class MyAppState extends State<MyApp> {
    var cList = [Colors.red, Colors.green, Colors.blue];
    Color c = Colors.grey;
    List<PopupMenuEntry<int>> itemBuilderFnc(context) => [...]
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            debugShowCheckedModeBanner: false,
            home: Scaffold(
                appBar: AppBar(title: Text("Title"),
                    backgroundColor: c,
                    actions: [ PopupMenuButton<int>(...)]))
            ));
    }
}
```

PopupMenuButton

Step 3 → create the build method with an AppBar that uses “c” variable for its background color.

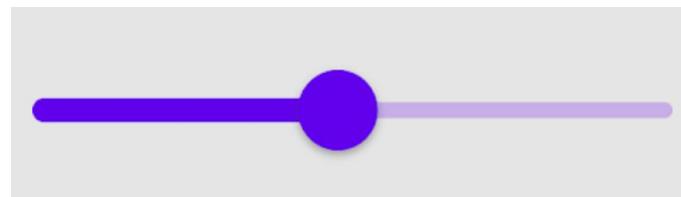
```
class MyAppState extends State<MyApp> {
    var cList = [Colors.red, Colors.green, Colors.blue];
    Color c = Colors.grey;
    List<PopupMenuEntry<int>> itemBuilderFnc(context) => [...]
    @override
    Widget build(Bui PopupMenuItem<int>(
        itemBuilder: itemBuilderFnc,
        debugShowC onSelected: (i) { setState(() { c = cList[i]; }); }
        home: Scaf ) )
        appBar: AppBar(circ text: title, backgrounColor: c,
        actions: [ PopupMenuItem<int>(...)] )
    )));
}
```

Slider

Slider

A slider is a widget that can be used to select a numerical value within a specific interval.

It is usually used for things like music / SFX volume, color (RGB / Saturation) selection, brightness selection, etc.



Slider

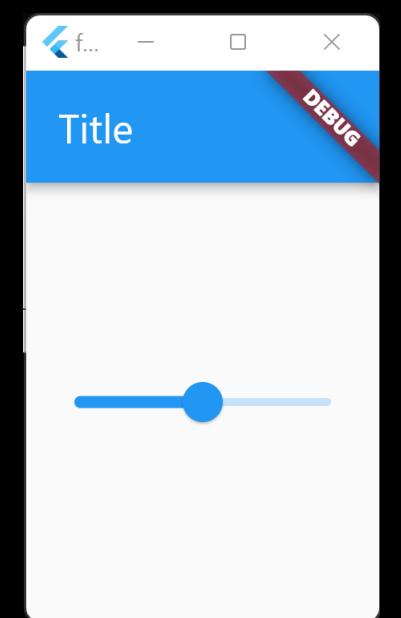
Constructor:

```
Slider({  
    required double value,  
    required void Function(double)? onChanged,  
    double min = 0.0,  
    double max = 1.0,  
    int? divisions,  
    String? label,  
    Color? activeColor,  
    Color? inactiveColor,  
    ... }  
)
```

Slider

A very simple example:

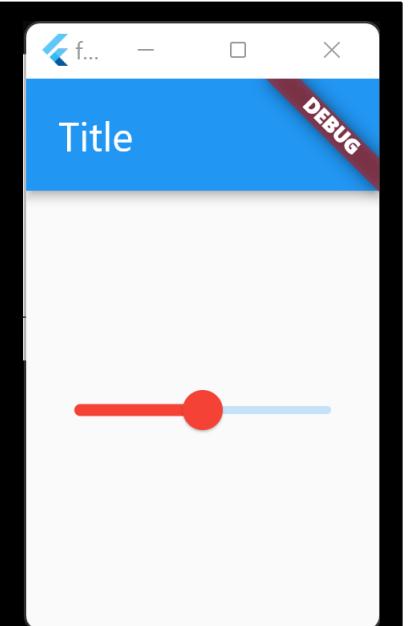
```
class MyAppState extends State<MyApp> {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar( title: Text("Title")),  
        body: Center(  
          child: Slider(value: 20,  
                        min: 10,  
                        max: 30,  
                        onChanged: (v) => {})),  
        ));  
  }  
}
```



Slider

Use activeColor property to change slider color.

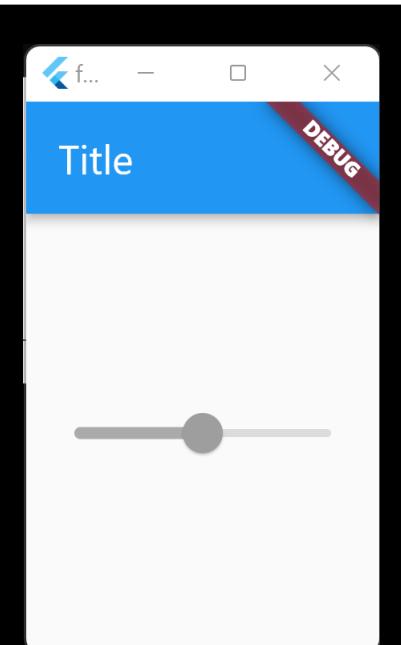
```
class MyAppState extends State<MyApp> {  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            home: Scaffold(  
                appBar: AppBar( title: Text("Title")),  
                body: Center(  
                    child: Slider(value: 20,  
                                min: 10,  
                                max: 30,  
                                activeColor: Colors.red ,  
                                onChanged: (v) => {})),  
            ));  
    }  
}
```



Slider

To disable a slider, set the onChange callback to null.

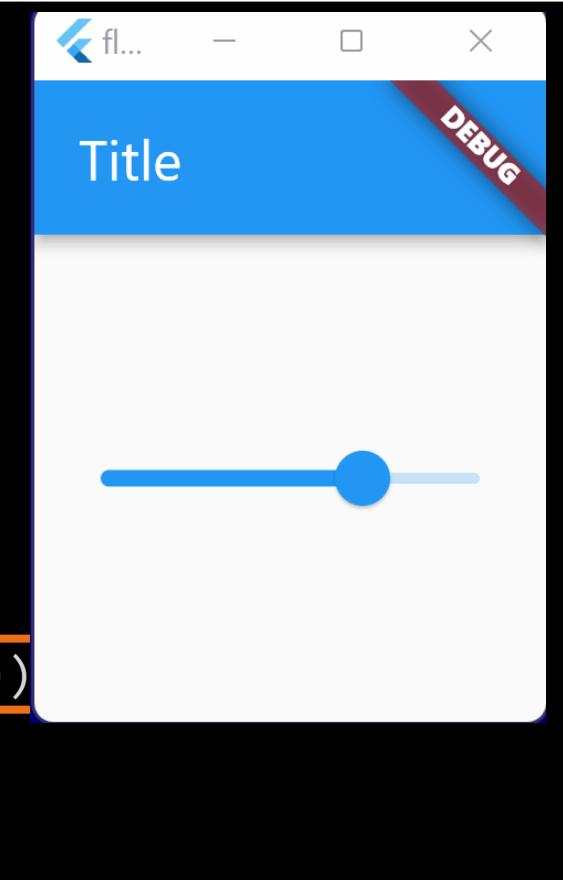
```
class MyAppState extends State<MyApp> {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar( title: Text("Title")),  
        body: Center(  
          child: Slider(value: 20,  
                        min: 10,  
                        max: 30,  
                        onChanged: null)),  
        ));  
  }  
}
```



Slider

The value of the slider has to be used via the onChanged callback

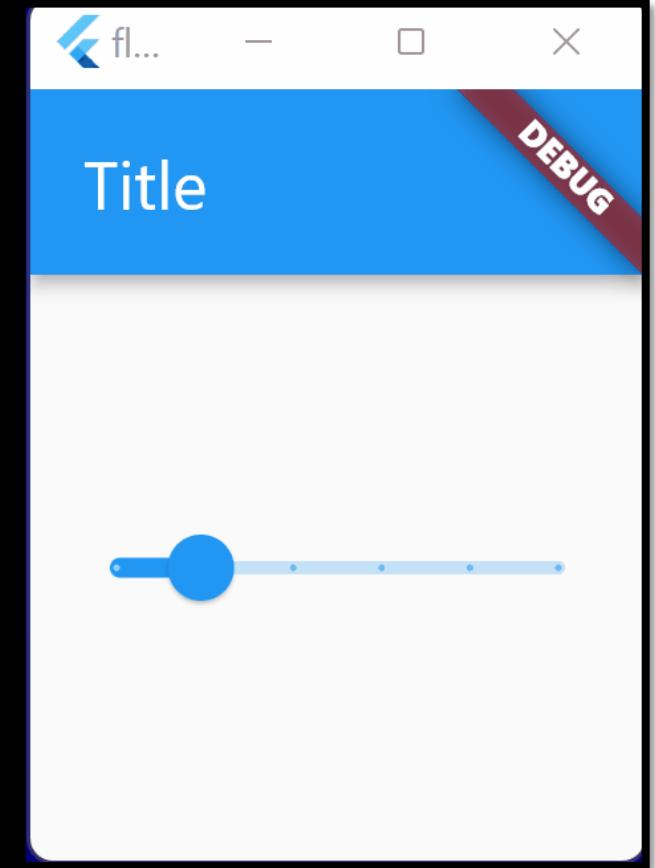
```
class MyAppState extends State<MyApp> {  
    double v = 20;  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            home: Scaffold(  
                appBar: AppBar( title: Text("Title")),  
                body: Center(  
                    child: Slider(  
                        value: v,  
                        min: 10, max: 30,  
                        onChanged: (val) => setState(() { v = val; })))  
            ));  
    }  
}
```



Slider

To show the label, use division and label together (just like in this example).

```
Widget build(BuildContext context) {  
  return MaterialApp(  
    home: Scaffold(  
      appBar: AppBar( title: Text("Title")),  
      body: Center(  
        child: Slider(  
          value: v,  
          min: 10,  
          max: 30,  
          divisions: 5,  
          label: v.toString(),  
          onChanged: (val) => setState(() { v = val; })  
        ),  
      )),  
  );  
}
```



Q & A



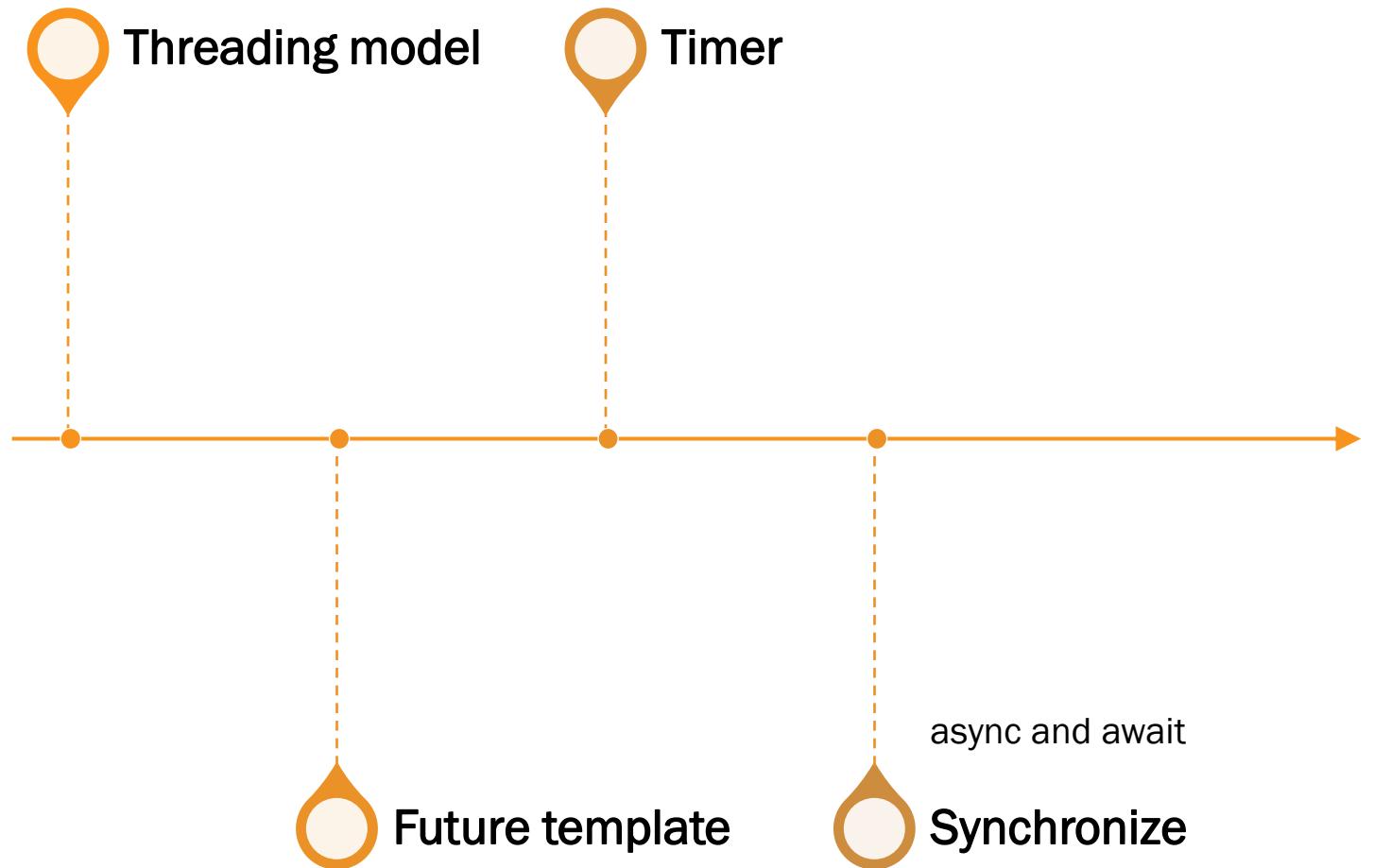


DART Language

COURSE 10 (REV 3)

GAVRILUT DRAGOS

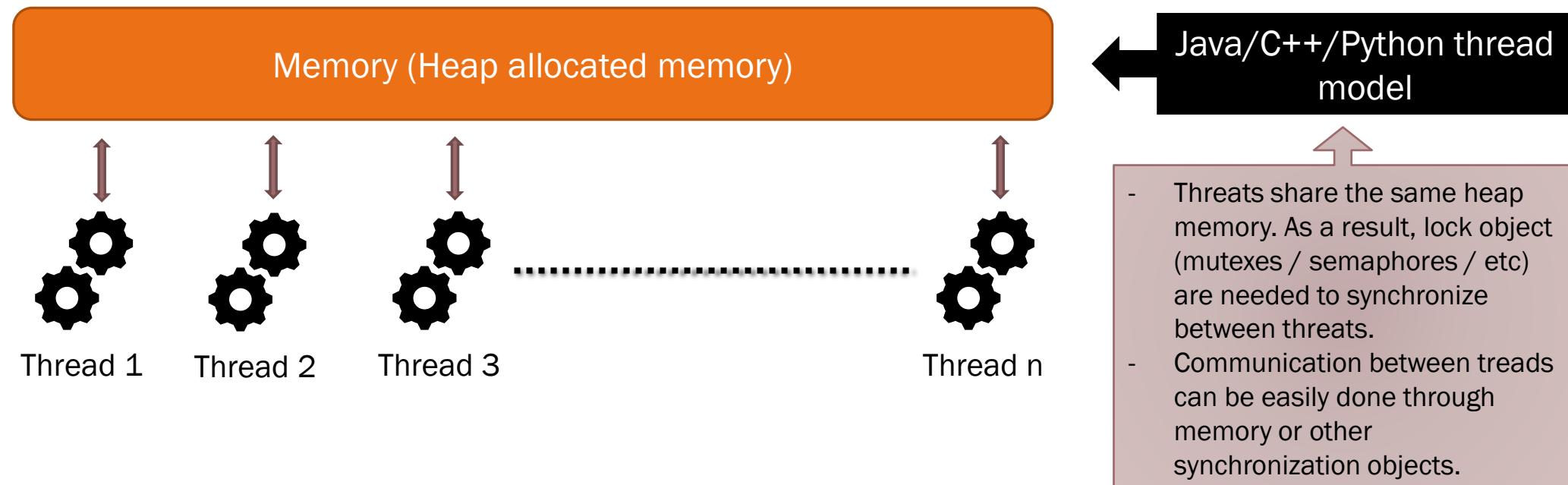
Agenda



Threading model

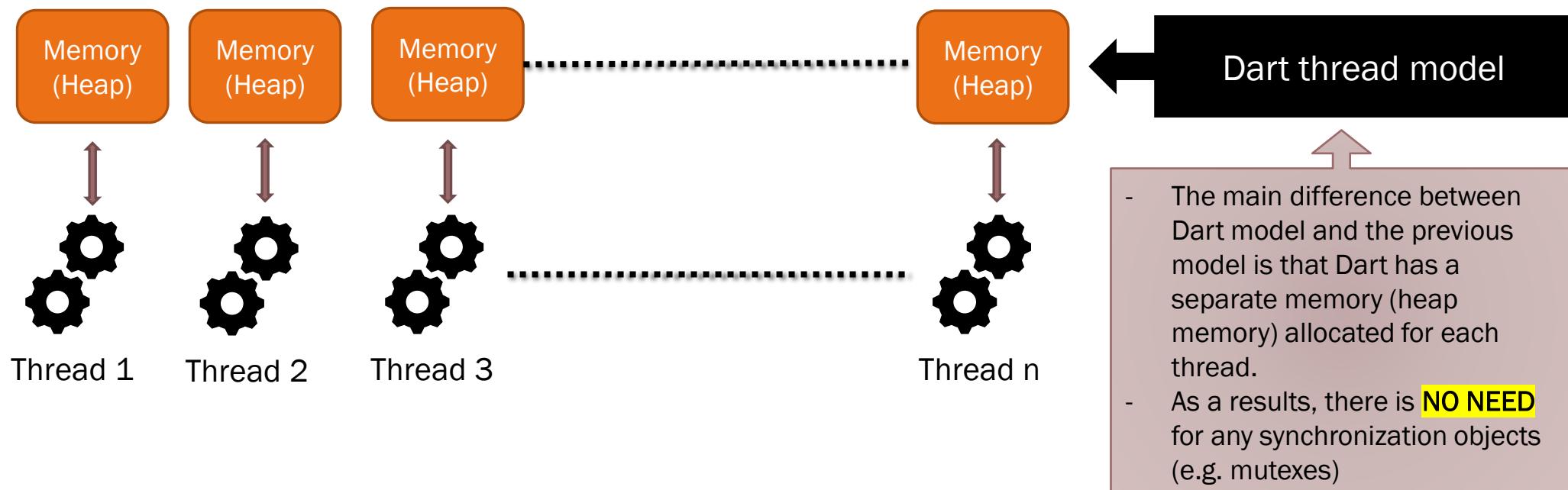
Threading model

Generically, most programming languages have a multi-threading support. Dart has one too, but with some differences.



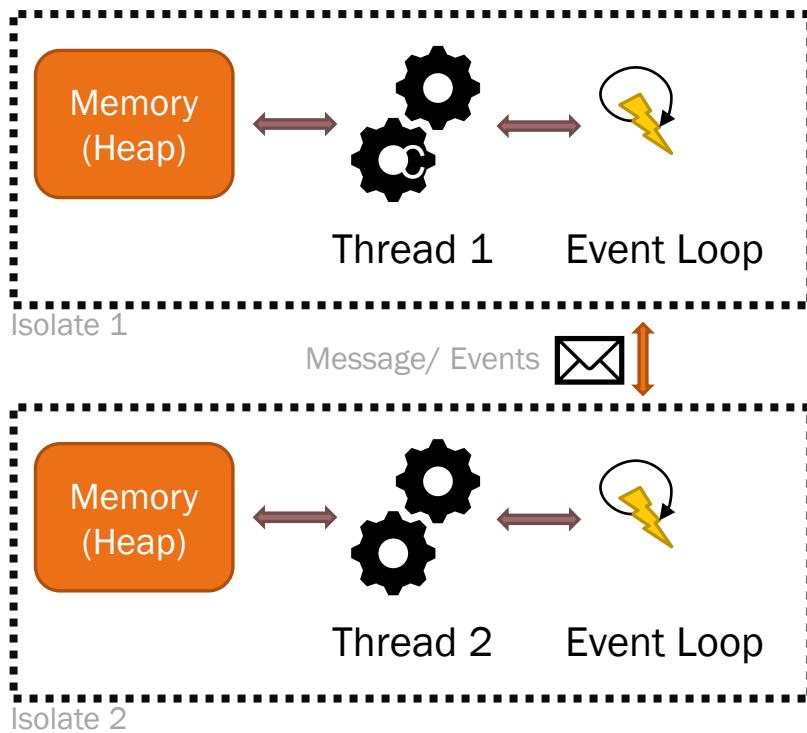
Threading model

Generically, most programming languages have a multi-threading support. Dart has one too, but with some differences.



Threading model

Generically, most programming languages have a multi-threading support. Dart has one too, but with some differences.



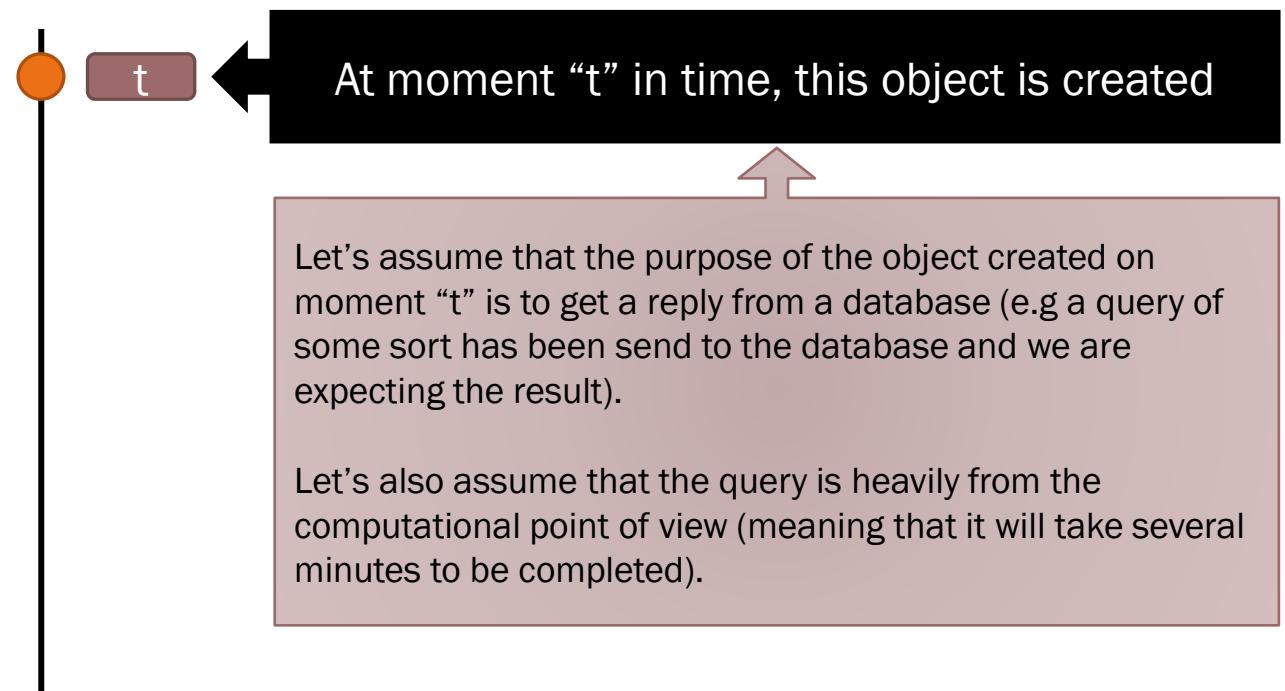
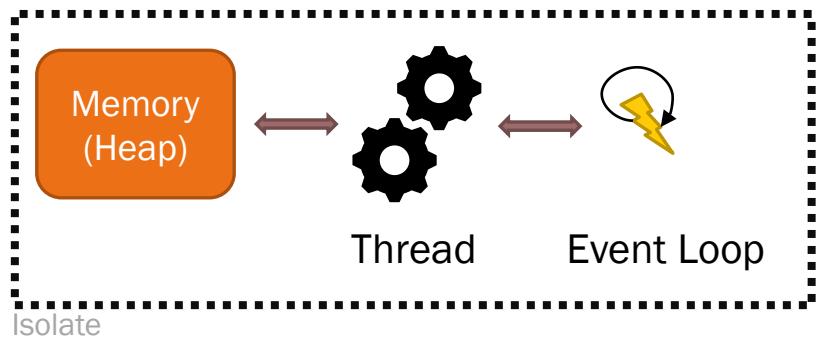
An Isolate

- An isolate uses an event loop to process messages or I/O events
- Multiple isolates can communicate between them in a similar manner (by sending an event to the event loop).
- The memory used by an isolate is his own (meaning that another isolate can not access any object created by another isolate).
- Any DART program has at least (an most of the time) only one isolate (the one that represent the function **main**).

Future object

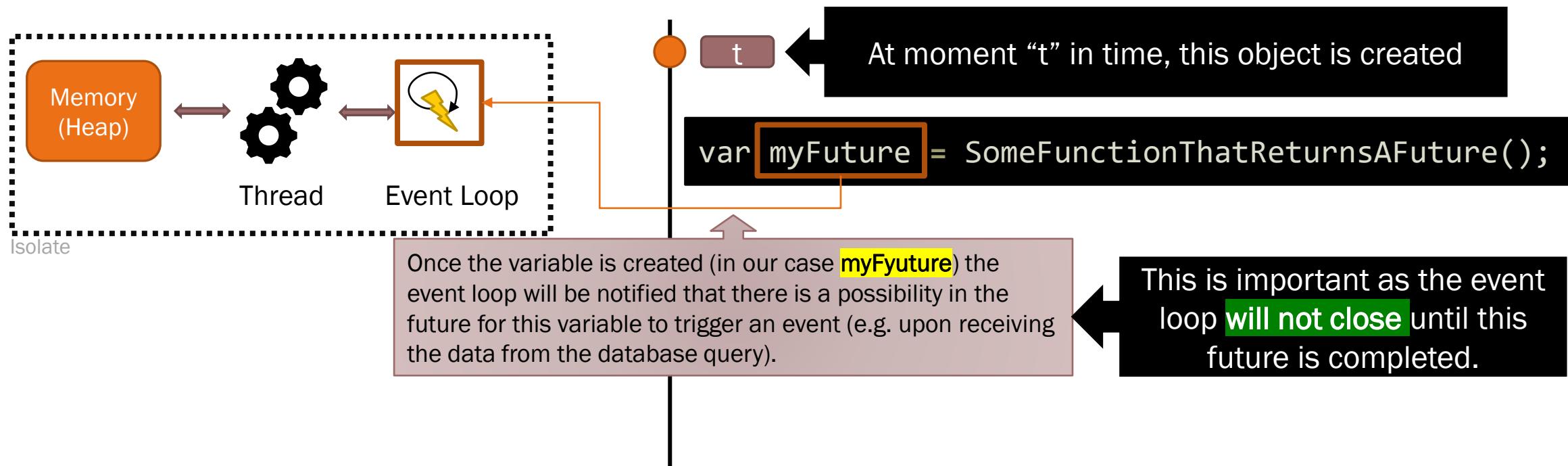
Futures

A future object is a place-holder (a container) that will contain that object you are requesting not immediately, but sometimes in the future.



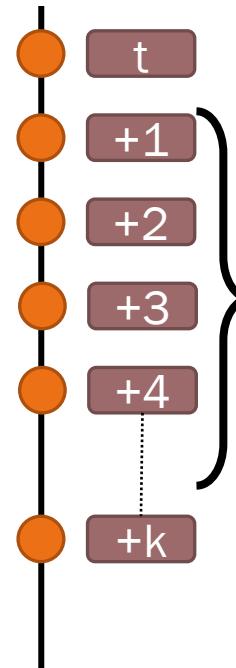
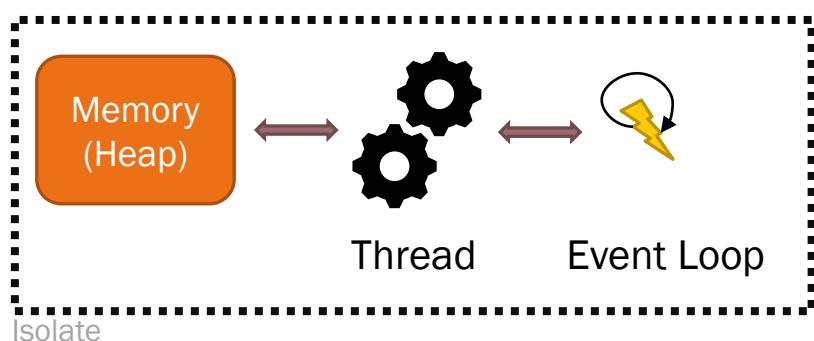
Futures

A future object is a place-holder (a container) that will contain that object you are requesting not immediately, but sometimes in the future.



Futures

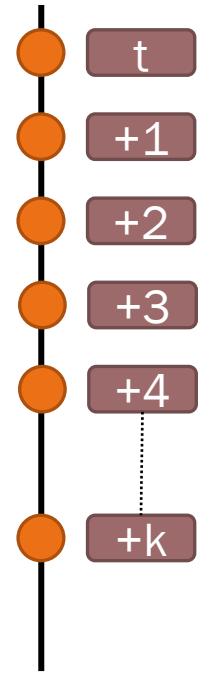
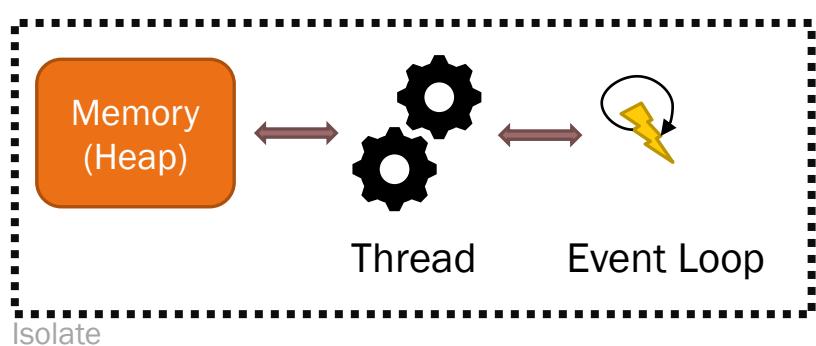
A future object is a place-holder (a container) that will contain that object you are requesting not immediately, but sometimes in the future.



For the next “ k ” periods of time nothing happens (in fact the event loop will process other messages)

Futures

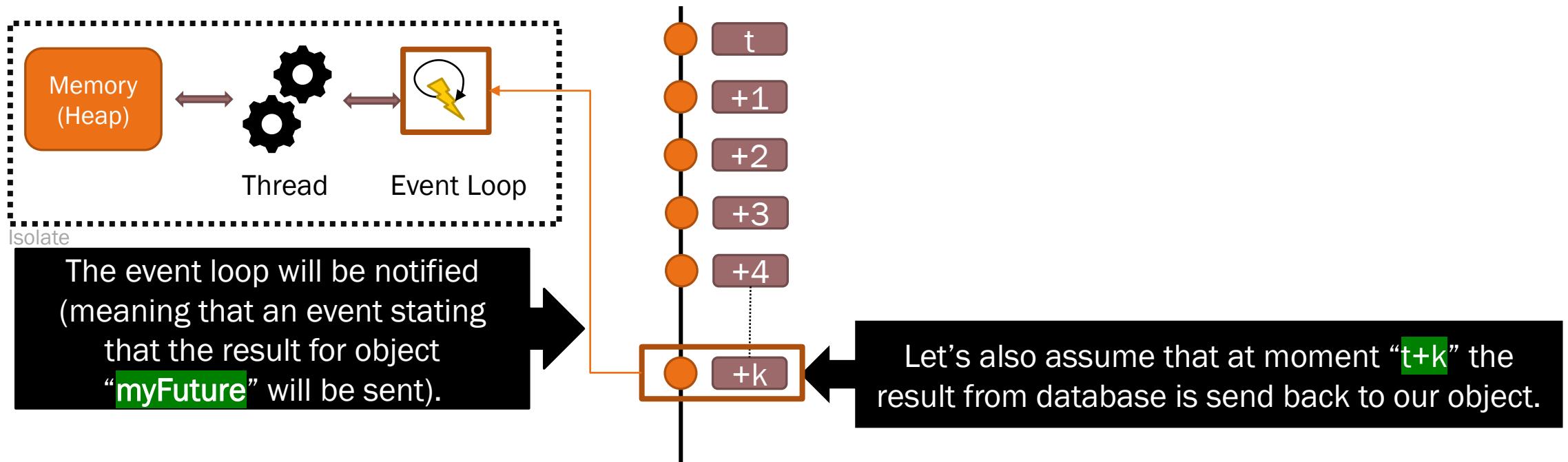
A future object is a place-holder (a container) that will contain that object you are requesting not immediately, but sometimes in the future.



Let's also assume that at moment " $t+k$ " the result from database is send back to our object.

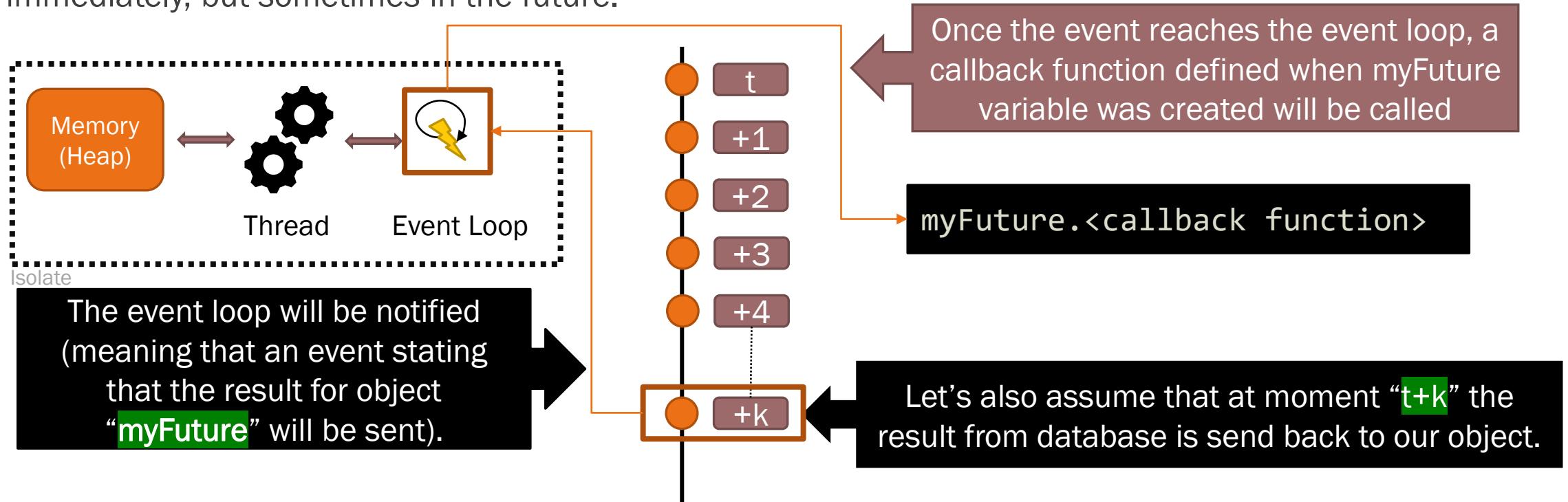
Futures

A future object is a place-holder (a container) that will contain that object you are requesting not immediately, but sometimes in the future.



Futures

A future object is a place-holder (a container) that will have that object you are requesting not immediately, but sometimes in the future.



Futures

A future object is a template and can be constructed in the following way:

```
Future<T> (FutureOr<T> computation())
Future<T>.delayed (Duration duration, [FutureOr<T> computation()])
Future<T>.sync (FutureOr<T> computation())
Future<T>.value ([FutureOr<T>? value])
```

Where `FutureOr<T>` can be either a value of type `T` or an object of type `Future<T>`.

Dart compiler will throw an error if any class tries to extend / implement or mix a `FutureOr<>` class.

Out of the above constructors, `Future<T>.delayed` implies that the computation code will be executed after a specific period of time.

To use/create a future object import “`dart:async`” library.

Futures

A Future example:

```
import "dart:async";

int CodeToBeExecuted() {
    print("Future code called");
    return 10;
}
void main() {
    print("Start main code");
    var futureObj = Future<int>.delayed(
        Duration(seconds: 2),
        CodeToBeExecuted);
    print("End main code");
}
```

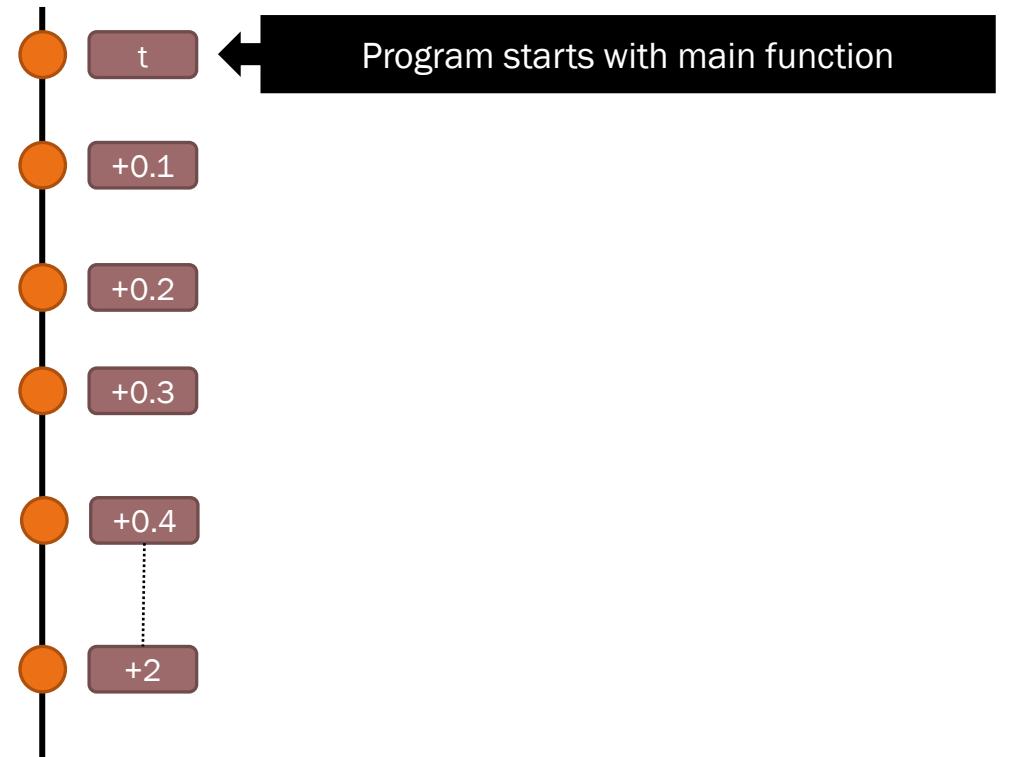
Futures

A Future example:

```
import "dart:async";

int CodeToBeExecuted() {
    print("Future code called");
    return 10;
}

void main() {
    print("Start main code");
    var futureObj = Future<int>.delayed(
        Duration(seconds: 2),
        CodeToBeExecuted);
    print("End main code");
}
```



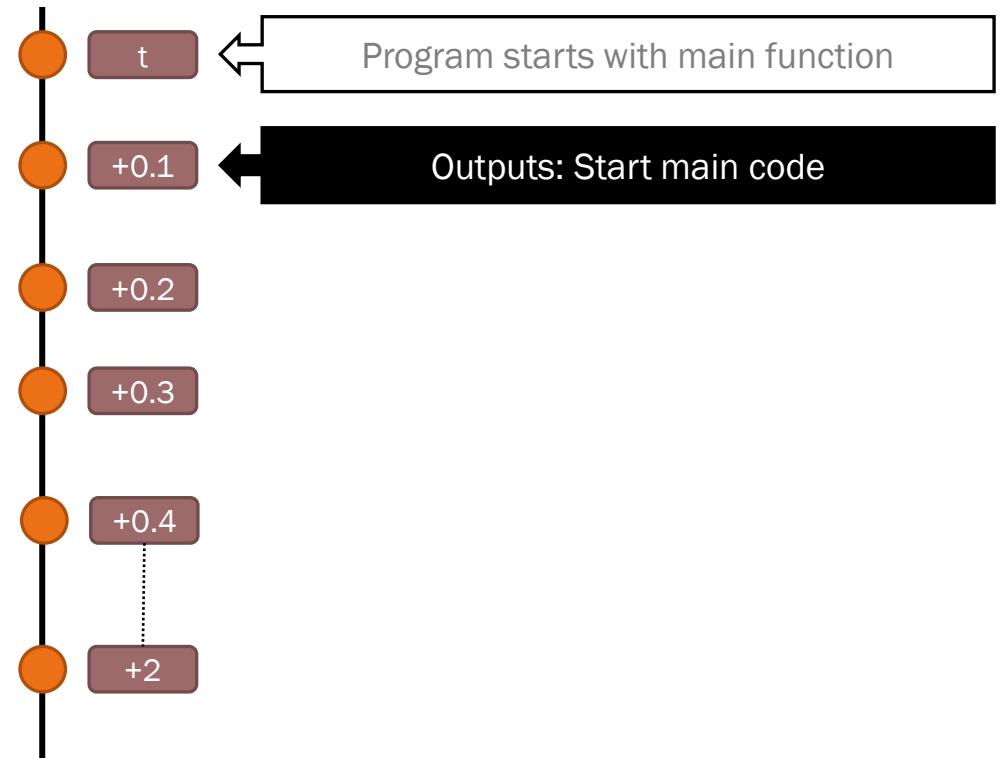
Futures

A Future example:

```
import "dart:async";

int CodeToBeExecuted() {
    print("Future code called");
    return 10;
}

void main() {
    print("Start main code");
    var futureObj = Future<int>.delayed(
        Duration(seconds: 2),
        CodeToBeExecuted);
    print("End main code");
}
```



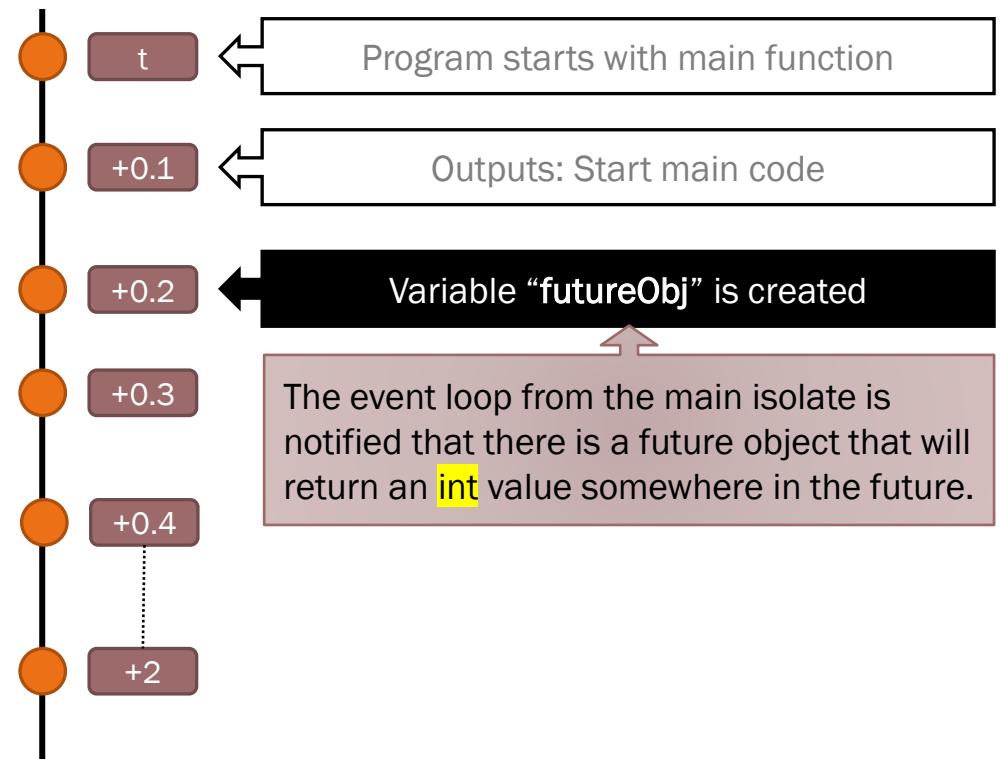
Futures

A Future example:

```
import "dart:async";

int CodeToBeExecuted() {
    print("Future code called");
    return 10;
}

void main() {
    print("Start main code");
    var futureObj = Future<int>.delayed(
        Duration(seconds: 2),
        CodeToBeExecuted);
    print("End main code");
}
```



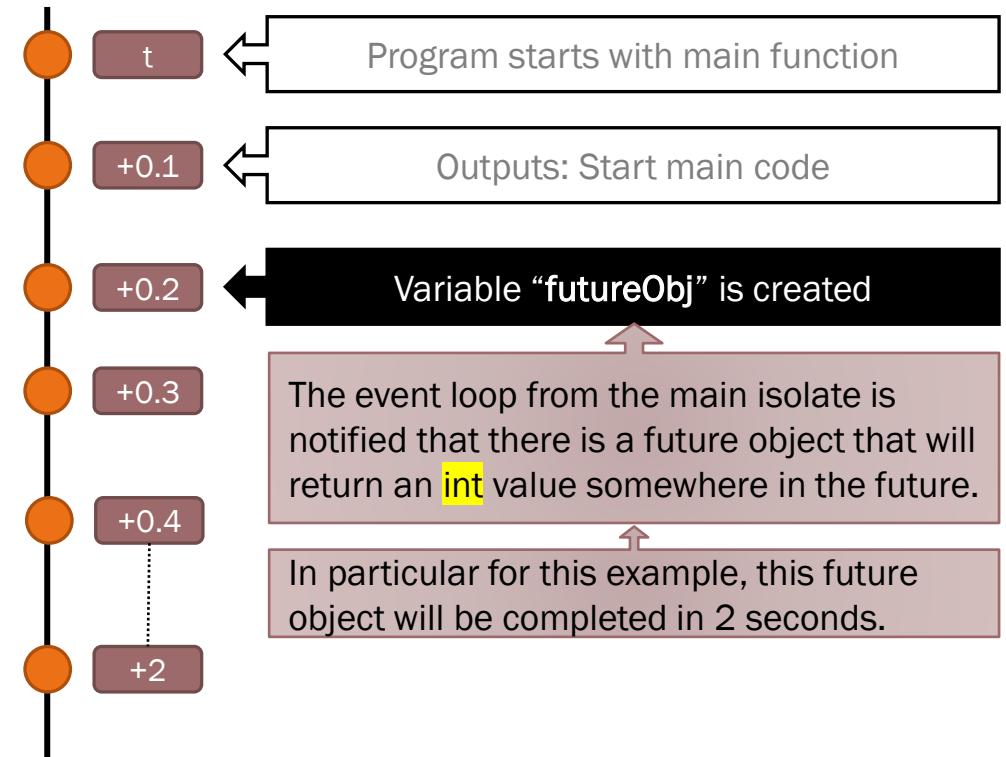
Futures

A Future example:

```
import "dart:async";

int CodeToBeExecuted() {
    print("Future code called");
    return 10;
}

void main() {
    print("Start main code");
    var futureObj = Future<int>.delayed(
        Duration(seconds: 2),
        CodeToBeExecuted);
    print("End main code");
}
```



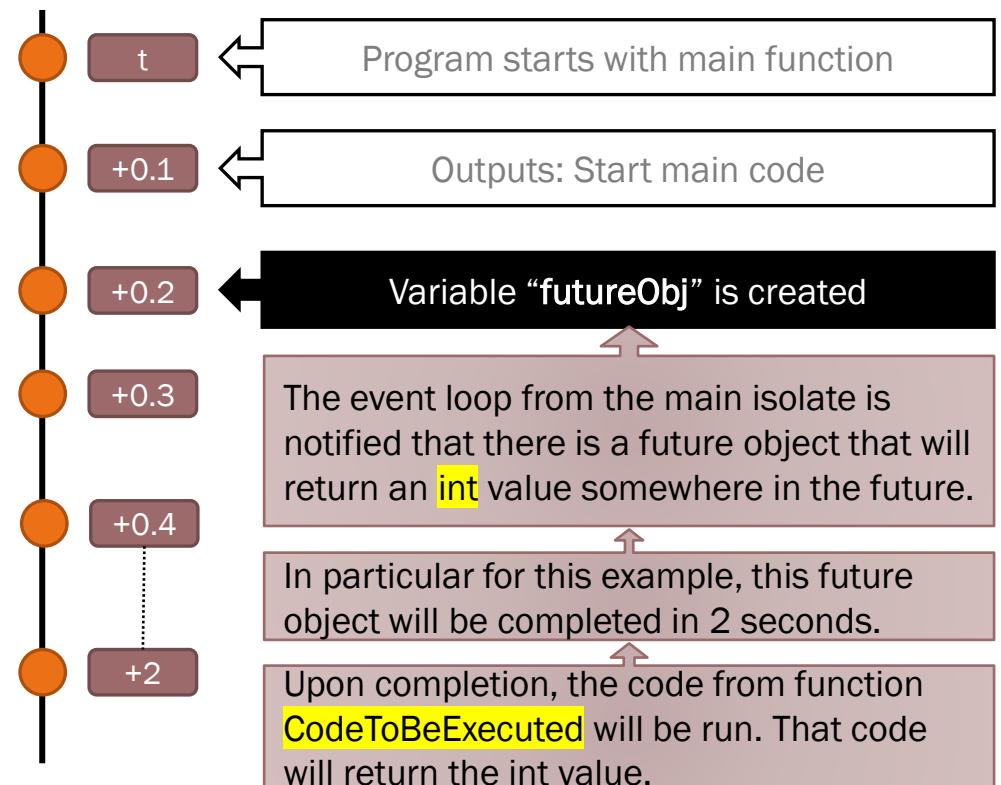
Futures

A Future example:

```
import "dart:async";

int CodeToBeExecuted() {
    print("Future code called");
    return 10;
}

void main() {
    print("Start main code");
    var futureObj = Future<int>.delayed(
        Duration(seconds: 2),
        CodeToBeExecuted);
    print("End main code");
}
```



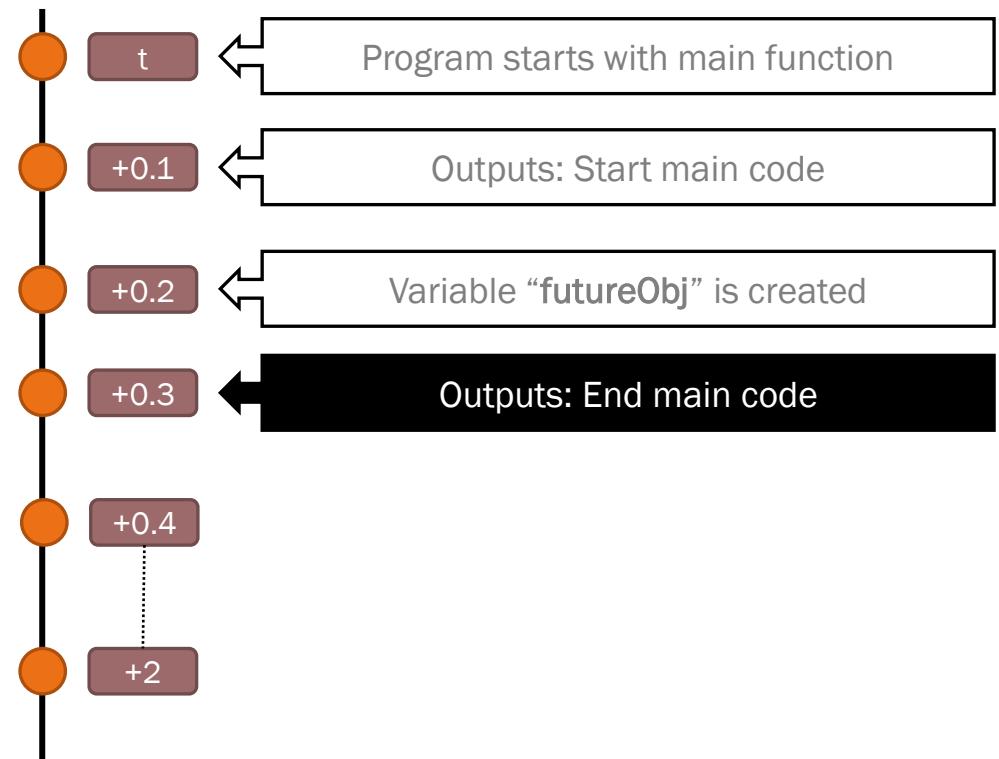
Futures

A Future example:

```
import "dart:async";

int CodeToBeExecuted() {
    print("Future code called");
    return 10;
}

void main() {
    print("Start main code");
    var futureObj = Future<int>.delayed(
        Duration(seconds: 2),
        CodeToBeExecuted);
    print("End main code");
}
```



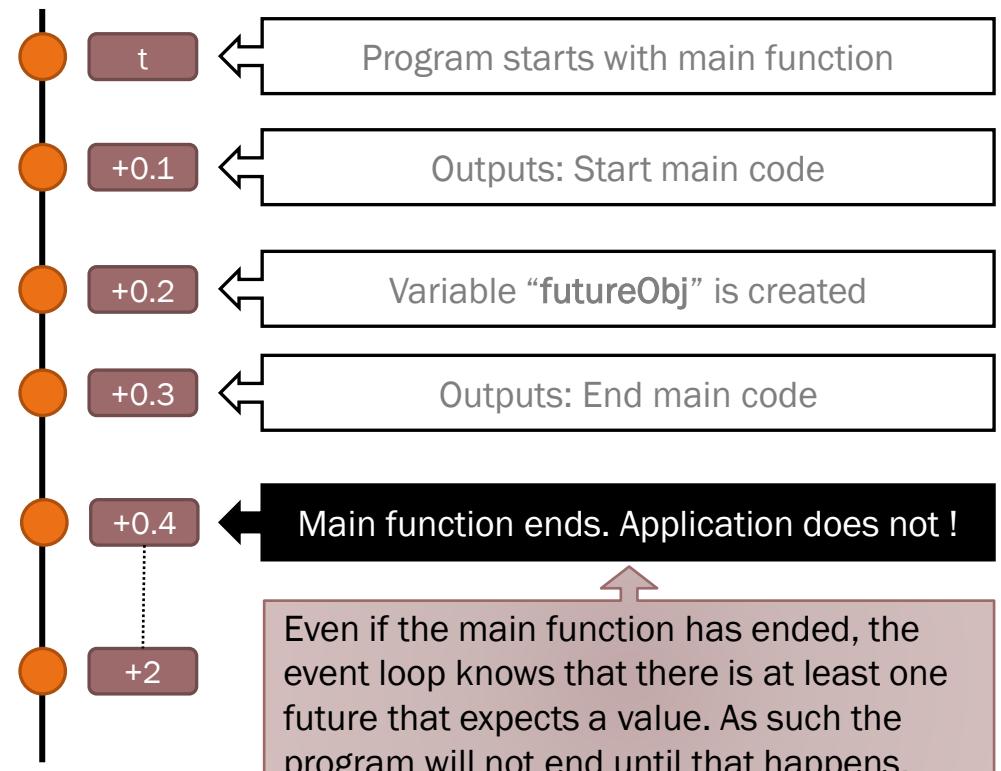
Futures

A Future example:

```
import "dart:async";

int CodeToBeExecuted() {
    print("Future code called");
    return 10;
}

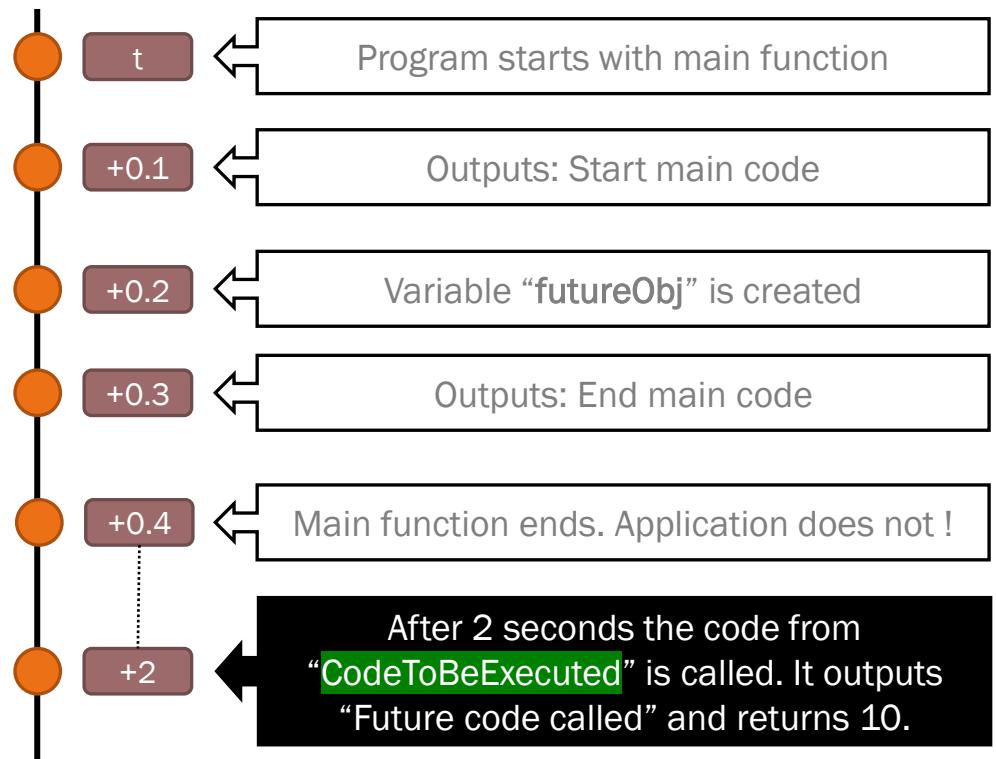
void main() {
    print("Start main code");
    var futureObj = Future<int>.delayed(
        Duration(seconds: 2),
        CodeToBeExecuted);
    print("End main code");
}
```



Futures

A Future example:

```
import "dart:async";  
  
int CodeToBeExecuted() {  
    print("Future code called");  
    return 10;  
}  
  
void main() {  
    print("Start main code");  
    var futureObj = Future<int>.delayed(  
        Duration(seconds: 2),  
        CodeToBeExecuted);  
    print("End main code");  
}
```



Futures

To get a callback/be notified when a futures completes, use the method **then**

```
Future<T> then<T> (FutureOr<T> onValue(T value), {Function? onError})
```

This method will be called when the future object completes and the value it return is passed on to the **onValue** callback.

```
import "dart:async";
void processValue(int value) {
    print("Value received is ${value}");
}
void main() {
    print("Start main code");
    var futureObj = Future<int>.delayed(Duration(seconds: 2), ()=>10);
    futureObj.then(processValue);
    print("End main code");
}
```

Output:

Start main code
End main code
Value received is 10

Futures

Method `.then(...)` can be used to linked a future with another one.

```
import "dart:async";
FutureOr<int> SecondFuture(int value) {
    print("Second future -> value=${value}");
    return 0;
}
FutureOr<int> FirstFuture(int value) {
    print("First future -> value=${value}");
    return Future<int>.delayed(Duration(seconds:value), ()=>2)
        .then(SecondFuture);
}
void main() {
    print("Start");
    Future<int>.delayed(Duration(seconds: 2), ()=>4).then(FirstFuture);
}
```

Output:

Start
First future -> value=4
Second future -> value=2

Futures

Dart `Future<T>.value` named constructor can be used to return a value (but after the current execution ends).

```
import "dart:async";
import "dart:io";

void main() {
    print("Start");
    stdin.readLineSync();
    for (var i=0;i<10;i++) {
        Future<int>.value(i).then((value) => print(value));
    }
    stdin.readLineSync();
    print("End");
}
```

Output:

Start

End

0

1

2

3

4

5

6

7

8

9

Futures

Let's analyze the following piece of code and work with the assumption that on each `stdin.readLineSync()` we will wait **10 seconds** before we introduce a string and press Enter. The following code was compiled and tested with: **Dart SDK version: 2.15.1 (stable)**

```
import "dart:async";
import "dart:io";

void main() {
    print("Start");
    stdin.readLineSync();
    Future<int>.delayed(Duration(seconds: 2), ()=>2)
        .then((value) => print(value));
    stdin.readLineSync();
    print("End");
}
```

Futures

Let's analyze the following piece of code and work with the assumption that on each `stdin.readLineSync()` we will wait **10 seconds** before we introduce a string and press Enter. The following code was compiled and tested with: **Dart SDK version: 2.15.1 (stable)**

```
import "dart:async";
import "dart:io";

void main() {
    print("Start");
    stdin.readLineSync();
    Future<int>.delayed(Duration(seconds: 2), ()=>2)
        .then((value) => print(value));
    stdin.readLineSync();
    print("End");
}
```

1. Start is outputted to the screen

Futures

Let's analyze the following piece of code and work with the assumption that on each `stdin.readLineSync()` we will wait **10 seconds** before we introduce a string and press Enter. The following code was compiled and tested with: **Dart SDK version: 2.15.1 (stable)**

```
import "dart:async";
import "dart:io";

void main() {
  print("Start");
  stdin.readLineSync();
  Future<int>.delayed(Duration(seconds: 2), ()=>2)
    .then((value) => print(value));
  stdin.readLineSync();
  print("End");
}
```

1. Start is outputted to the screen

2. Wait for a line to be typed from the keyboard
followed by the ENTER key

Futures

Let's analyze the following piece of code and work with the assumption that on each `stdin.readLineSync()` we will wait **10 seconds** before we introduce a string and press Enter. The following code was compiled and tested with: **Dart SDK version: 2.15.1 (stable)**

```
import "dart:async";
import "dart:io";

void main() {
  print("Start");
  stdin.readLineSync();
  Future<int>.delayed(Duration(seconds: 2), ()=>2)
    .then((value) => print(value));
  stdin.readLineSync();
  print("End");
}
```

1. Start is outputted to the screen
2. Wait for a line to be typed from the keyboard followed by the ENTER key
3. A new Future is created. It should be triggered after 2 seconds and when it will be triggered it will print value 2 on the screen.

Futures

Let's analyze the following piece of code and work with the assumption that on each `stdin.readLineSync()` we will wait **10 seconds** before we introduce a string and press Enter. The following code was compiled and tested with: **Dart SDK version: 2.15.1 (stable)**

```
import "dart:async";
import "dart:io";

void main() {
  print("Start");
  stdin.readLineSync();
  Future<int>.delayed(Duration(seconds: 2), ()=>2)
    .then((value) => print(value));
  stdin.readLineSync();
  print("End");
}
```

1. Start is outputted to the screen
2. Wait for a line to be typed from the keyboard followed by the ENTER key
3. A new Future is created. It should be triggered after 2 seconds and when it will be triggered it will print value 2 on the screen.
4. Wait for ENTER to be pressed (more than 10 seconds).

In theory, those 2 seconds from that Future object would pass, and the value 2 should be printed ! In practice nothing happens.

Futures

Let's analyze the following piece of code and work with the assumption that on each `stdin.readLineSync()` we will wait **10 seconds** before we introduce a string and press Enter. The following code was compiled and tested with: **Dart SDK version: 2.15.1 (stable)**

```
import "dart:async";
import "dart:io";

void main() {
  print("Start");
  stdin.readLineSync();
  Future<int>.delayed(Duration(seconds: 2), ()=>2)
    .then((value) => print(value));
  stdin.readLineSync();
  print("End");
}
```

1. Start is outputted to the screen
2. Wait for a line to be typed from the keyboard followed by the ENTER key
3. A new Future is created. It should be triggered after 2 seconds and when it will be triggered it will print value 2 on the screen.
4. Wait for ENTER to be pressed (more than 10 seconds).
5. End is outputted to the screen

Futures

Let's analyze the following piece of code and work with the assumption that on each `stdin.readLineSync()` we will wait **10 seconds** before we introduce a string and press Enter. The following code was compiled and tested with: **Dart SDK version: 2.15.1 (stable)**

```
import "dart:async";
import "dart:io";

void main() {
  print("Start");
  stdin.readLineSync();
  Future<int>.delayed(Duration(seconds: 2), ()=>2)
    .then((value) => print(value));
  stdin.readLineSync();
  print("End");
}
```

1. Start is outputted to the screen
2. Wait for a line to be typed from the keyboard followed by the ENTER key
3. A new Future is created. It should be triggered after 2 seconds and when it will be triggered it will print value 2 on the screen.
4. Wait for ENTER to be pressed (more than 10 seconds).
5. End is outputted to the screen
6. Main code ends. The event loop checks the future, 2 seconds have passed, and it prints 2

Futures

Let's see how this work:



Futures

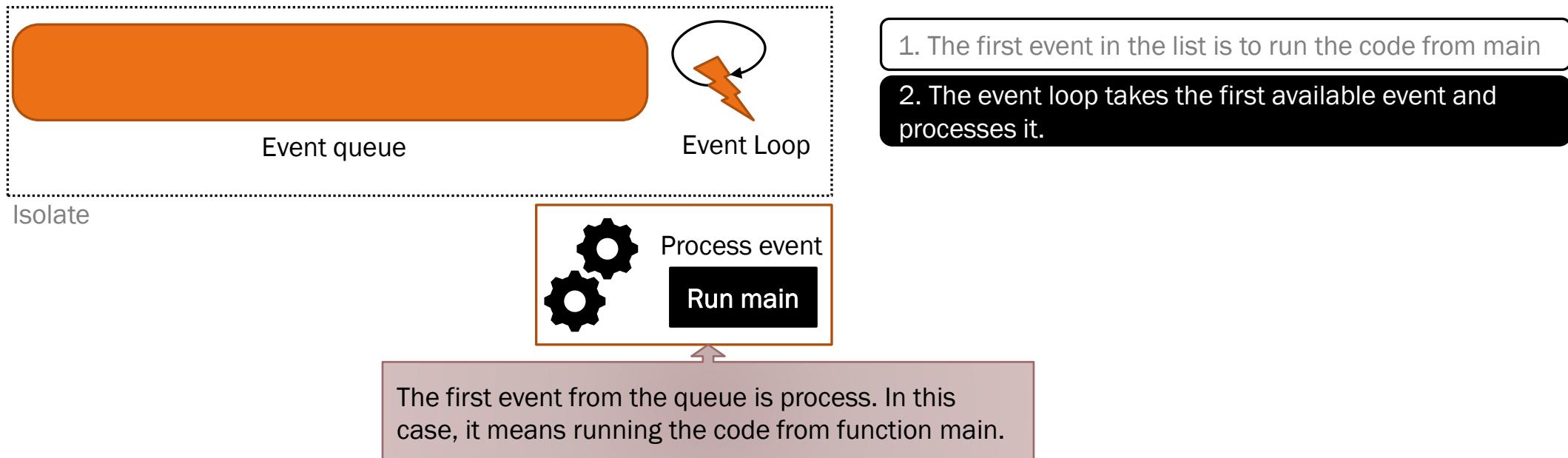
Let's see how this work:



1. The first event in the list is to run the code from main

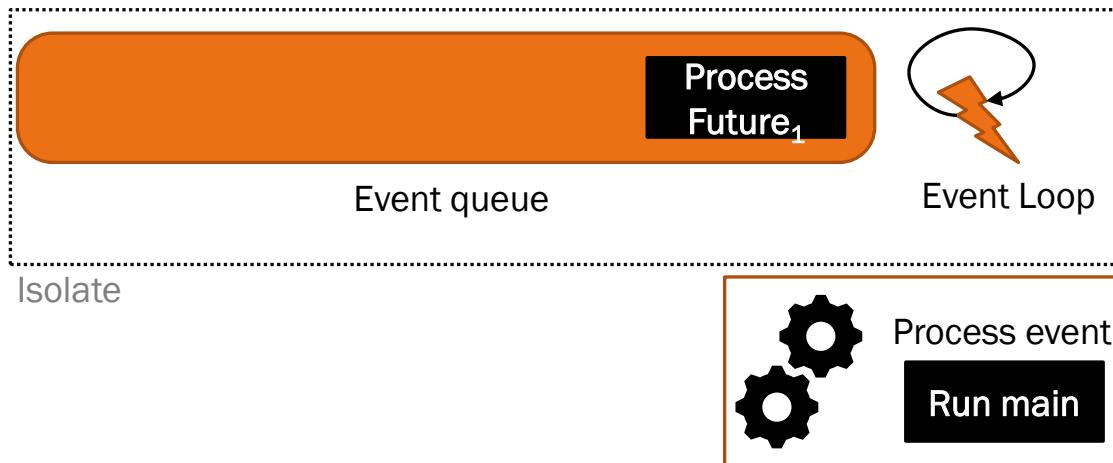
Futures

Let's see how this work:



Futures

Let's see how this work:

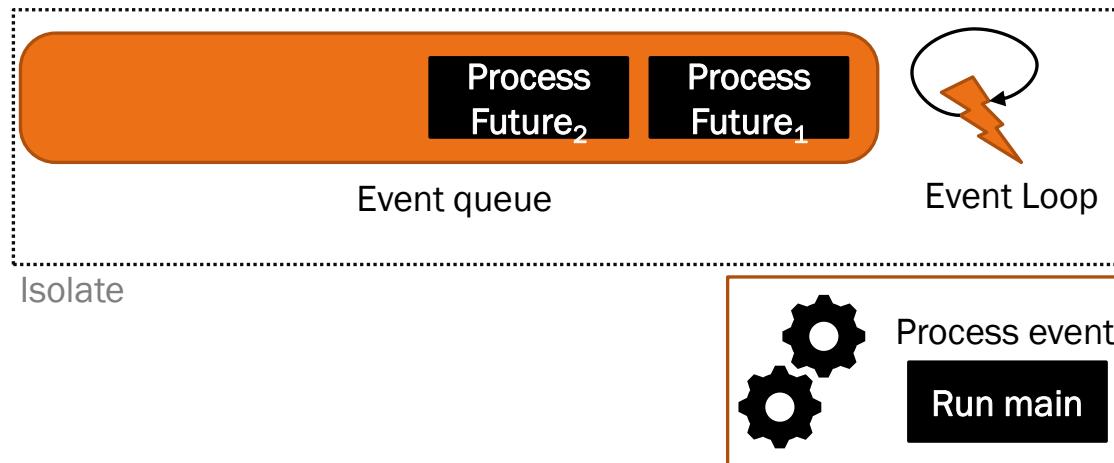


1. The first event in the list is to run the code from main
2. The event loop takes the first available event and processes it.
3. Upon execution of the cod from main a Future object is created → lets call it Future₁

As a result, an event to process that Future (to test if it is completed will be created and pushed in the queue)..

Futures

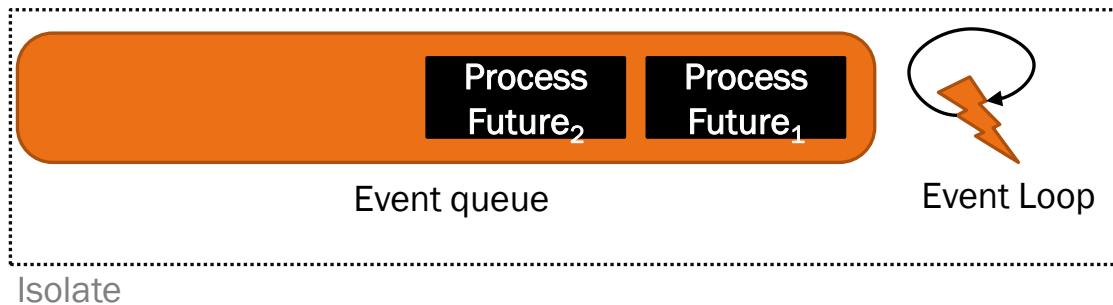
Let's see how this work:



1. The first event in the list is to run the code from main
2. The event loop takes the first available event and processes it.
3. Upon execution of the cod from main a Future object is created → lets call it Future₁
4. After a while, another future object (let's call it Future₂) is created by the code from main

Futures

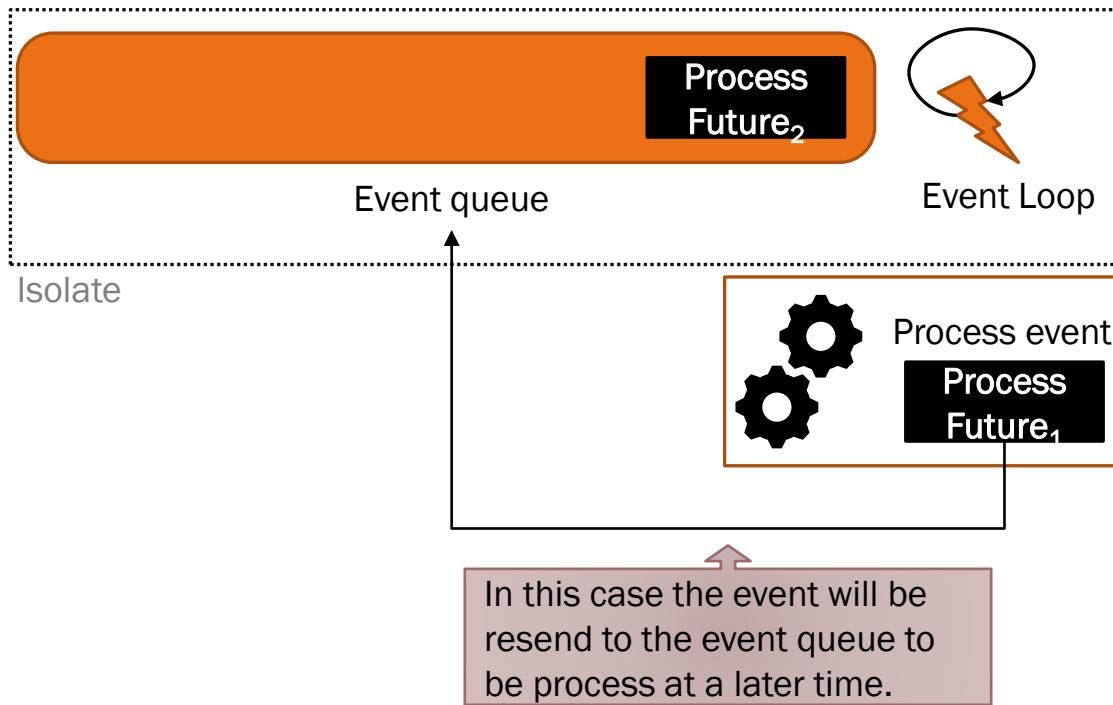
Let's see how this work:



1. The first event in the list is to run the code from main
2. The event loop takes the first available event and processes it.
3. Upon execution of the cod from main a Future object is created → lets call it Future₁
4. After a while, another uture object (let's call it Future₂) is created by the code from main
5. The code from main ends (meaning that the first event “Run main” was completed).

Futures

Let's see how this work:

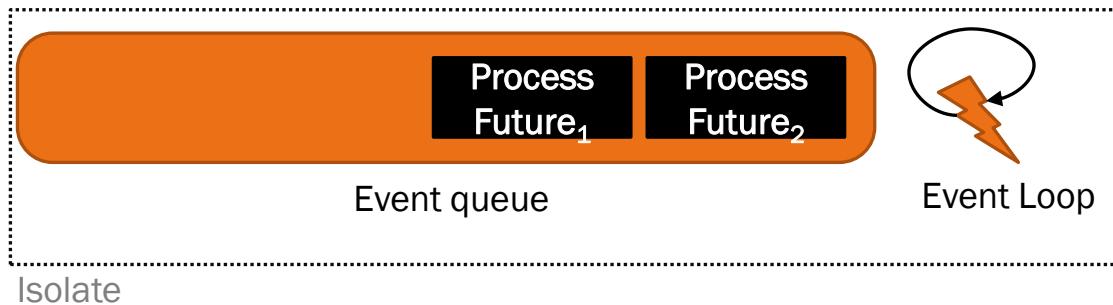


1. The first event in the list is to run the code from main
2. The event loop takes the first available event and processes it.
3. Upon execution of the cod from main a Future object is created → lets call it Future₁
4. After a while, another future object (let's call it Future₂) is created by the code from main
5. The code from main ends (meaning that the first event “Run main” was completed).
6. The event loop extracts the next event from the queue and start processing it.

Let's assume that the event was not completed (e.g. the URL content was not read, the time that should have passed for a Future<T>,delayed did not pass, etc)

Futures

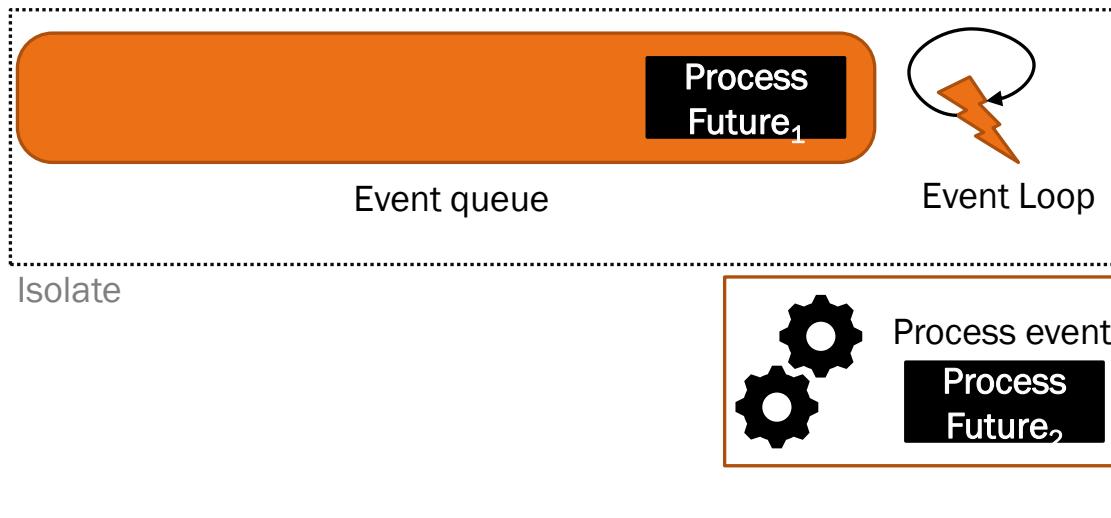
Let's see how this work:



1. The first event in the list is to run the code from main
2. The event loop takes the first available event and processes it.
3. Upon execution of the cod from main a Future object is created → lets call it Future₁
4. After a while, another future object (let's call it Future₂) is created by the code from main
5. The code from main ends (meaning that the first event “Run main” was completed).
6. The event loop extracts the next event from the queue and start processing it.
7. The event loop now extracts the next event from the queue and starts processing it.

Futures

Let's see how this work:



1. The first event in the list is to run the code from main

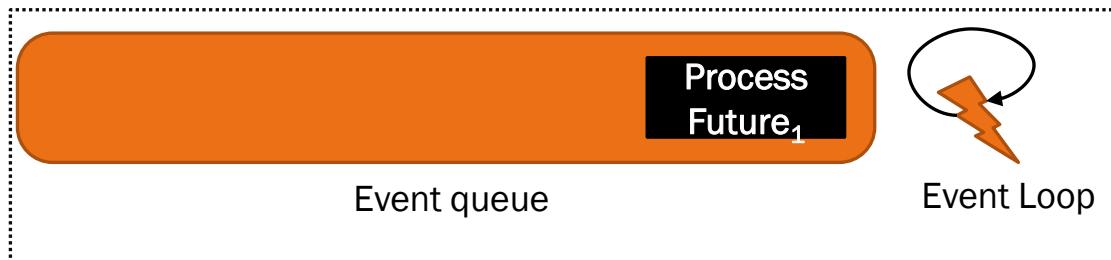
7. The event loop now extracts the next event from the queue and starts processing it.

8. The next event refers to Future₂. **Event loop will start process this one.**

Assuming that Future₂ is completed, the callback that was set through `Future<T>.then(...)` method will be called.

Futures

Let's see how this work:



1. The first event in the list is to run the code from main

7. The event loop now extracts the next event from the queue and starts processing it.

8. The next event refers to Future₂. Event loop will start process this one.

9. After the previous future has ended, the event is removed from the event ques and the process is restarted and the next event is process.

OBS: This is an approximation behavior (in reality the code behind the event loop si more complex and subject to change from version to version).

Futures

Other methods that can be used for a Future.

<code>Stream<T> Future<T>.asStream()</code>	Creates a stream based on a future
<code>Future<T>.catchError(Function onError, {bool test(Object error)?})</code>	Set a callback to be used when an error is raised.
<code>Future<T>.whenComplete(FutureOr<void> action())</code>	Call when a future is complete (regardless of the future outcome - complete or error).
<code>Future<T>.timeout(Duration timeLimit, {FutureOr<T> onTimeout()}?)</code>	Sets a timeout for a Future object

Futures

An example using `catchError` and `whenComplete`.

```
import "dart:async";

int runCode() {
    print("Run code");
    throw "some exception";
}

int onErrorCallback(error) { print("Error: ${error}"); return 0; }

void main() {
    Future<int>.sync(runCode)
        .catchError(onErrorCallback)
        .then((value) => print("Value:${value}"))
        .whenComplete(() => print("Done"));
}
```

Output:

Run code
Error: some exception
Value: 0
Done

Futures

A simple example that uses a http package to download our website (info.uaic.ro) and find out the name of our dean.

```
import 'package:http/http.dart' as http;
import 'dart:io';

main() {
  http.get("https://www.info.uaic.ro/conducere/").then((response) {
    var s = r'Decan</strong>:[\w\s\.]*<span class="wikilink">([\w\s]*)</span>';
    var r = RegExp(s);
    var m = r.firstMatch(response.body);
    print(m[1]);
  });
}
```

Futures

To make this code work, the following steps must be performed:

1. Create a `pubspec.yaml` file in the same folder where your dart file is located. Add the following content to pubspec.yaml file

```
name: my_example
version: 1.2.3
dependencies:
  http: ^0.12.1
environment:
  sdk: '>=2.10.0 <3.0.0'
```

2. Run the following command: ``dart pub get`` → where ‘dart’ refers to the dart executable (e.g `<install_folder>\dart-sdk\bin\dart.exe`)
3. Compile ``dart compile exe <file_name>.dart``

Timer

Timer

Dart also has an object (Timer) that works like an event that is being called after a specific period of time.

Constructors:

<code>Timer (Duration timeLimit, void callback())</code>	Creates a timer that will be triggered only once
<code>Timer.periodic (Duration timePeriod, void callback(Timer timer));</code>	Creates a timer that will be triggered periodically

Properties:

<code>int Timer.tick</code>	Number of timePeriods that have passed
<code>bool Timer.isActive</code>	True if the timer is active, false otherwise

The un-named constructor is similar to a `Future<T>.delayed` object.

Timer

A simple example that uses a timer that will be triggered after 2 seconds and will print a message when those 2 seconds have passed.

```
import 'dart:async';

main() {
  print("start");
  Timer(Duration(seconds: 2), ()=>print("Timer was triggered"));
  print("end");
}
```

Output:

start

end

Timer was triggered

Timer

A timer also has a method (`.cancel()`) that can be used to stop it. This is useful for periodic timers.

```
import 'dart:async';

void timerCallback(Timer t) {
    if (t.tick>=4)
        t.cancel();
    print("Timer called: Tick = ${t.tick}");
}

main() {
    print("start");
    Timer.periodic(Duration(seconds: 2), timerCallback);
    print("end");
}
```

Output:

start

end

Timer called: Tick = 1

Timer called: Tick = 2

Timer called: Tick = 3

Timer called: Tick = 4

Synchronization (await/async)

Synchronization

Let's analyze the following code:

```
import "dart:async";

void main() {
  print("start");
  var f = Future<int>.delayed(Duration(seconds: 3),
    ()=>5)
    .then((value)=>print("received: ${value}"));
  print("end");
}
```

Output:
start
End
Received: 5

We know that the output will be **start** , **end** and **received: 5** (as a result on how event loop works) → meaning that main code gets executed first, then the future code is executed.

But what if we want to wait until “f” completes and only then move to the next event code from main ?

Synchronization

First there are some observations to be made here:

1. We need a way to tell the event loop that the current method / function that the event loop is execution can be stop until a future object completes (or to be more precise, we need a way to tell the event loop that current functions works asynchronously).
2. We need a way to specify at what point (location/locations) of the code from we should stop executing the code and cease control to the event loop until the future object we are interested in completes.

Synchronization

Dart has introduced 2 keywords to resolve the previously described problems: `await` and `async`

- “`async`” → should be used for any function that can be interrupted so that its execution waits for one or multiple futures to be completed.
- “`await`” → can only be used int functions that are defined using “`async`” keyword and specify the future we want to wait to complete.

```
import "dart:async";  
  
Future<returnType> MyFunction(...) async  
{  
    // some code  
    // a future object (let's call it my_future) is created/obtained  
    await my_future;  
    // some other code  
}
```

This tells the event loop that *MyFunction* should be treated asynchronously (meaning that the execution can be paused until a future completes).

This command pauses the execution of *MyFunction* and wait for *my_future* object to be completed.

Synchronization

Dart has introduced 2 keywords to resolve the previously described problems: `await` and `async`

- “`async`” → should be used for any function that can be interrupted so that its execution waits for one or multiple futures to be completed.
- “`await`” → can only be used int functions that are defined using “`async`” keyword and specify the future we want to wait to complete.

```
import "dart:async";
Future<returnType> MyFunction(...) async
{
    // some code
    // a future object (let's call it my_future) is created/obtained
    var res = await my_future; // res will be of type returnType
    // some other code
}
```

The return type should be a `Future` or `void` for any `async` functions.

Synchronization

Let's analyze the previous code with async/await:

```
import "dart:async";

void main() async {
  print("start");
  var f = Future<int>.delayed(Duration(seconds: 5),
    ()=>5)
    .then((value)=>print("received: ${value}"));
  await f;
  print("end");
}
```

Output:
Start
received: 5
end

Now the output will be **start** , **received: 5** and **end** (as a result on how event loop works) → meaning that main code gets executed first, then the future code is executed.

Also, the code will wait for 5 seconds until **"f"** is complete and only then will run **"print("end");"** command

Synchronization

`await` keyword can be used to get the value that a future returns.

```
import "dart:async";

void main() async {
  print("start");
  var f = Future<int>.value(123);
  var result = await f;
  print(result);
  print("end");
}
```

Output:
start
123
end

As a general concept: “`var <variable_name> = await <future>`” will copy the value returned by a future into a new variable.

Synchronization

This technique can be used to sync multiple future objects that are linked together. Let's assume that we have Future₁ that returns a value that will be used by Future₂ to compute something.

```
import "dart:async";
void main() {
    Future<int>.value(10)
        .then((val) => Future<int>.value(val*val)
            .then((value) => print(value)));
}
```

or we can write it with `async` and `await` like this:

```
import "dart:async";
void main() async {
    int x = await Future<int>.value(10);
    print(await Future<int>.value(x*x));
}
```

Synchronization

You can also use a `try...catch` or `try...catch...finally` block for cases where a Future could not complete with a value, but with an error of some sort (e.g. we are waiting to download some content from the internet and the server we are downloading from suddenly stop responding).

```
import "dart:async";

void main() async {
    try {
        int x = await Future<int>.sync(() => throw "My error");
    }
    catch (err) {
        print("Exception: ${err}"); // will print Exception: My error
    }
}
```

Synchronization

Because any function that is created with `async` keyword returns a Future object (except for void functions), this can be another way to create a Future object.

```
import 'dart:async';

Future<int> GetAFuture(int value) async => value * value;
main() {
  print("start");
  GetAFuture(10).then((value) => print(value));
  print("end");
}
```

Output:
start
end
100

Q & A

