UNIVERSITY OF PADOVA
DEPARTMENT OF INFORMATION ENGINEERING

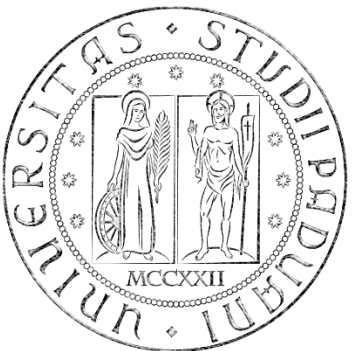Biomedical Wearable Technologies
for Healthcare and Wellbeing

Dart 101 – Part 1

A.Y. 2024-2025

Giacomo Cappon

Outline

➢ **Dart & Hello, world!**

➢ Variables

➢ Null-safety

➢ Built-in types

➢ Functions

➢ Operators

➢ Control flow constructs

➢ Exercises

➢ Homework

➢ Resources

Dart

➢ **What is Dart?**
- Dart is a object-oriented, open source language
- It is pretty new (2011)
- Cross-platform oriented

➢ **Why this choice?**
- State-of-the-art and Google-maintained
- Single codebase for iOS and Android (and Mac, Windows, Web)
- Relatively easy to learn
- Lots of examples
- Fastly growing job market
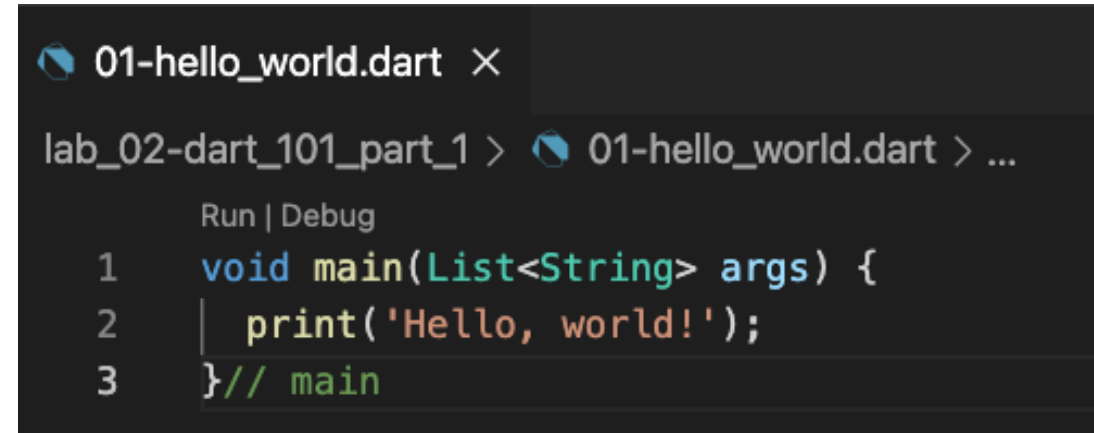- Nice hot reload feature

Hello, world!

➢ Let's start with the classic "Hello, world!" program.

➢ Preliminary steps:
- Create a folder
- Open VS Code and navigate to that folder
- Create a new file called "01-hello_world.dart"

➢ Write the snippet of code on the right

```dart
void main(List<String> args) {
    print('Hello, world!');
}// main
```

➢ Run it!
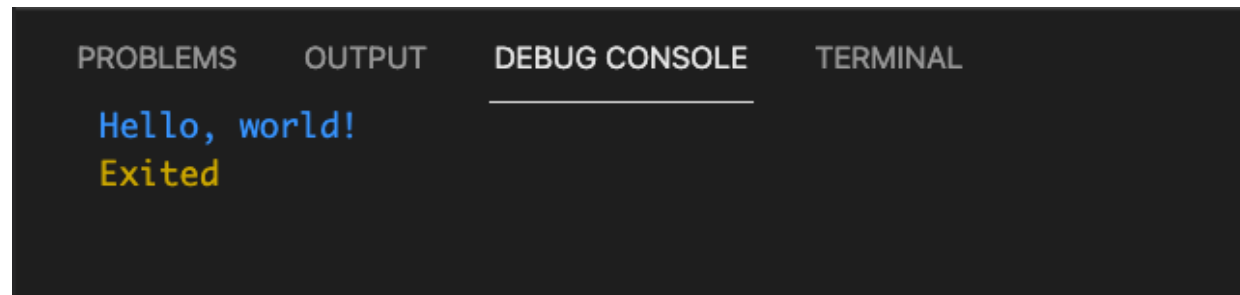- Option 1: Go to the upper menu and click the "Run > Start Debugging"
- Option 2: Open a terminal, cd to the folder containing "01-hello_world.dart", and type
  `dart 01-hello_world.dart`
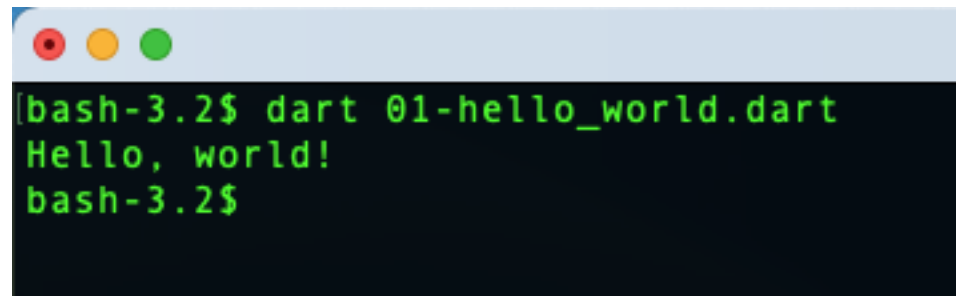
Hello, world!

➢ You will see something like

➢ Option 1:


```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL
    Hello, world!
    Exited
```

➢ Option 2:
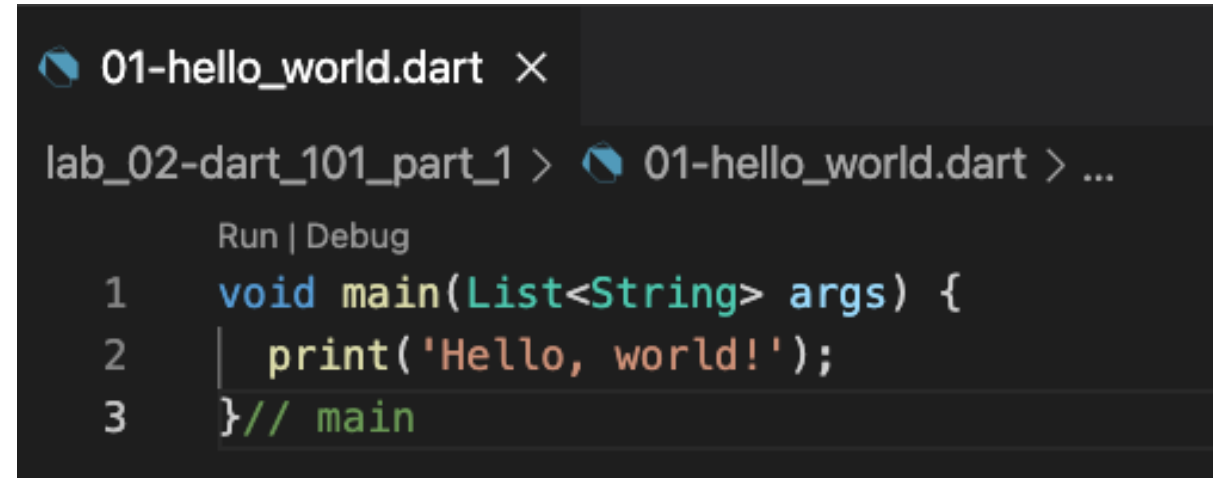

```
[bash-3.2$ dart 01-hello_world.dart
Hello, world!
bash-3.2$
```

Hello, world!

➢ Let's break it down:

- `main` is the starting point

- `main` is a function an accepts a `List` of strings

- `;` is necessary at the EOL

- It all starts from a `main`

- `print()` is an handy function to display output

- `// This is a comment`

```dart
01-hello_world.dart  ×
lab_02-dart_101_part_1 > 01-hello_world.dart > …
      Run | Debug
  1   void main(List<String> args) {
  2       print('Hello, world!');
  3   }// main
```

Full example in lab_02-dart_101_part_1/01-hello_world.dart

Outline

➢ Dart & Hello, world!
➢ **Variables**
➢ Null-safety
➢ Built-in types
➢ Functions
➢ Operators
➢ Control flow constructs

➢ Exercises
➢ Homework
➢ Resources

Variables

➢ **`var`** can be used to create and initialize a variable:
- `var number = 42; // number contains a **reference** to a number object with value 42`

➢ **`final`** can be used to create and initialize a constant (run-time):
- `final name = 'Bob';`

➢ **`const`** can be used to create and initialize a constant (compile-time):
- `const pi = 3.14;`

➢ Note that, the **type** of variables was inferred!

➢ Of course, type can be also specified:
- `final String anotherName = 'Jack'; //This will be a constant String`

Outline

➢ Dart & Hello, world!

➢ Variables

➢ **Null-safety**

➢ Built-in types

➢ Functions

➢ Operators

➢ Control flow constructs

➢ Exercises

➢ Homework

➢ Resources

Null safety

- In Dart, every variable has a nullable type
  - `int a; //Non-nullable int type`
  - `int? b; //Nullable int type`
- Un-initialized non-nullable typed variables must be assigned before they can be used.
- On the other hand, un-initialized nullable typed variables can (they have **null value by default**).
- This system, known as **null-safety** allows to be 100% sure whether an expression can be null or not at some point of the code.
- If you want to assign a value to a non-nullable variable using a nullable expression (that you know for sure that it is not null in that specific moment), you can add a `!` to assert that it is not null. Runtime will throw an exception if you are wrong.

Full example in lab_02-dart_101_part_1/03-null_safety.dart

Outline

➢ Dart & Hello, world!

➢ Variables

➢ Null-safety

➢ **Built-in types**

➢ Functions

➢ Operators

➢ Control flow constructs

➢ Exercises

➢ Homework

➢ Resources

Built-in types - Numbers

➢ Integer numbers (64-bit):
  - `int anInteger = 3;`

➢ Floating point numbers (64-bit floating point - IEEE 754 standard):
  - `double aDouble = 3.0;`
  - `print(double.nan); //Special constant 1: not-a-number`
  - `print(double.negativeInfinity); //Special constant 2: -inf`
  - `print(double.infinity); //Special constant 3: inf`
  - `print(double.minPositive); //Special constant 4: minimum representable number`
  - `print(double.maxFinite); //Special constant 5: maximum representable positive number`

Note: No need to memorize these numbers!

Built-in types - Numbers

➢ Useful thing to know: how to **parse** numbers to strings and viceversa

➢ Parse a String to an int

- ▪ ```var one = int.parse('1');```

➢ Parse a String to a double

- ▪ ```var onePointOne = double.parse('1.1');```

➢ Parse an int to a String

- ▪ ```String anIntegerAsString = anInteger.toString();```

➢ Parse double to a String

- ▪ ```String aDoubleAsString = aDouble.toString();```

Built-in types - Strings

➢ `var s1 = 'Single quotes work well for string literals.';`

➢ `var s2 = "Double quotes work just as well.";`

➢ `var s3 = 'It\'s easy to escape the string delimiter.';`

➢ `var s4 = "It's even easier to use the other delimiter.";`

➢ It is possible to interpolate an expression inside a String using `${expression}`

  ▪ `int a = 4;`
    `int b = 3;`
    `print('a + b = ${a+b}');`

Built-in types - Strings

➢ Strings can be concatenated using +

- ```
  final hello = 'Hello';
  final world = 'world';
  print(hello + ' ' + world + '!');
  ```

➢ As you imagine, the String class has many handy methods and variables. Here's some examples:

- ```
  final str = 'test string';
  print(str.length); //Gets the length of a
  string
  print(str.toUpperCase()); //Converts the whole
  string to upper case
  print(str.contains('str')); //Checks if a
  string contains a pattern
  print(str.indexOf('str')); //Tells where a
  pattern is within a string
  ```

➢ There's a lot more…

Built-in types - Booleans

➤ To represent booleans, Dart uses the `bool` type
- `bool flag = true; //or false`

Built-in types - Lists

➢ Arrays, in Dart, are represented as List objects.
- ```
  var listInferred = [1,2,3]; //Type inferred
  List<int> listNotInferred = [1,2,3]; //Type explicited
  assert(listInferred != listNotInferred); //However they are not equal!
  ```

➢ List elements can be accessed by index (they start from 0):
- ```
  print(listInferred[1]); //This will print '2'
  ```

➢ Length of a list can be accessed via the length instance variable
- ```
  print(listInferred.length); //This will print '3'
  ```

➢ As strings, Dart comes with handy features for lists
- ```
  print(listInferred.reversed); //This will obtain the list in reversed
  order
  ```
- ```
  listInferred.add(4); //This will add a 4 to the tail of the list
  ```
- ```
  listInferred = [0, ...listInferred]; //This will add a 0 to the head of
  the list (using the spread operator ...)
  ```
- ```
  print(listInferred);
  ```

➢ There's a lot more...

List - map() method

➢ A special "functionality" of the `List` type is the `map()` method.

➢ `map()` takes as input a function that applies "something" element-by-element to the given `List`:

  ▪ ```
    final mappedList = [1,2,3].map((element) {
      return element * 2;
    });
    print('${mappedList}'); //This will print [2,4,6]
    ```

Built-in types - Maps

➤ A useful collection type in Dart is the Map type. A Map is an object that associates keys and values.

➤ Each key occurs only once, values can occur multiple times.
- `var mapInferred = {2: 'helium', 10: 'neon', 18: 'argon'}; //Type inferred`
- `Map<int, String> mapNotInferred = {2: 'helium', 10: 'neon', 18: 'argon'}; //Type not inferred`
- `assert(mapInferred != mapNotInferred); //Again they are not equal!`

➤ Elements of a Map can be accessed via the key.
- `print(mapInferred[2]); //This will print 'helium'`

➤ It is possible to add key-value pairs to the Map by simply
- mapInferred[1] = 'hydrogen';

➤ Values of a Map can be overwritten
- mapInferred[1] = 'H';

➤ Handy features are also present for Maps

➤ `print(mapInferred.length); //This will print the length of the Map`

➤ `print(mapInferred.containsKey(2)); //This will check is a key exists in the Map`

➤ There's a lot more...

Full example in lab_02-dart_101_part_1/04-built_in_types.dart

Outline

➢ Dart & Hello, world!

➢ Variables

➢ Null-safety

➢ Built-in types

➢ **Functions**

➢ Operators

➢ Control flow constructs

➢ Exercises

➢ Homework

➢ Resources

Functions

- ➤ To implement a function you can use the syntax
  returnType functionName(typeParam1 nameParam1, typeParam2 nameParam2,…){functionBody}:

  - double sumTwoNumbers(double a, double b){
      return a + b;
    }//sumTwoNumbers

    void main(List<String> args) {
      print(sumTwoNumbers(3,4)); //This will print '7.0'
    }//main

- ➤ Dart is truly object-oriented: even functions are objects of type Functions. This means that they can be an input of another function:

  - double doSomethingWithNumbers(Function foo, double a, double b){
      return foo(a,b);
    }//doSomeThingWithNumbers
    void main(List<String> args) {
      print(doSomethingWithNumbers(sumTwoNumbers, 3, 4)); //This will print '7.0'
    }//main

- ➤ For functions that contain just one expression, you can use the 'arrow syntax':

  - double doSomethingWithNumbersArrow(Function foo, double a, double b) => foo(a,b);
    void main(List<String> args) {
      print(doSomethingWithNumbersArrow(sumTwoNumbers, 3, 4)); //This will print '7.0'
    }//main

Functions – Parameters

➤ A function can have any number of *required positional* parameters.

➤ These can be followed either by *named* parameters or by *optional positional* parameters (but not both).

➤ Named parameters can be specified using **braces {}**:
```
returnType functionName(typeParam1 nameParam1,
typeParam2 nameParam2,…,{typeNamedParam1
nameNamedParam1,…}){functionBody}
```

➤ Optional positional parameters can be specified using **brackets []**:
```
returnType functionName(typeParam1 nameParam1,
typeParam2 nameParam2,…,[typeOptPosParam1
nameOptPosParam1,…]){functionBody}
```

Functions – Named parameters

- ➤ Example of function with named parameters
  - ▪ ```
    double doSomethingWithNumbersNamed(Function foo, {double? a,
    double? b}) => foo(a,b);
    void main(List<String> args) {
      print(doSomethingWithNumbersNamed(sumTwoNumbers,a:3, b:4)); //
    This will print '7.0'
    }//main
    ```

- ➤ Note that the parameters are nullable because they are optional. So, if not specified, their default value is null. This means that the following will fail:
  - ▪ ```
    void main(List<String> args) {
      try{
            print(doSomethingWithNumbersNamed(sumTwoNumbers,a:3)); //
    This fails
      }catch(e){
            print('This fails because the default value of b is
    null.');
      }//catch
    }//main
    ```

Functions – Named parameters

➢ Named parameters can be marked as `required` to avoid them to be optional.

- ```
  double sumTwoNumbersNamed({required double a, required double
  b}) => a+b;
  void main(List<String> args) {
    print(sumTwoNumbersNamed(a:3, b:4)); // This will print '7.0'
  }//main
  ```

➢ Note that we can remove the nullable type now.

➢ We can specify default values for named parameters:

- ```
  double sumTwoNumbersDefaultNamed({required double a, double b =
  0}) => a+b;
  void main(List<String> args) {
    print(sumTwoNumbersDefaultNamed(a:3)); // This will print '3.0'
  }//main
  ```

Functions – Optional positional parameters

➢ **As anticipated, it is possible to specify optional positional parameters:**

```
String sayHi([String? name]){
  if(name != null){
        return 'Hi ' + name + '!';
  }//if
  return 'Hi!';
}//sayHi
void main(List<String> args) {
  print(sayHi()); //This will print 'Hi!'
  print(sayHi('Paul')); //This will print 'Hi Paul!'
}//main
```

➢ **We can specify default values for optional positional parameters:**

```
String sayHiDefault([String name = '']) => 'Hi ' + name;
void main(List<String> args) {
  print(sayHiDefault()); //This will print 'Hi'
  print(sayHiDefault('Paul')); //This will print 'Hi Paul'
}//main
```

25

Full example in lab_02-dart_101_part_1/05-functions.dart

Outline

➢ Dart & Hello, world!

➢ Variables

➢ Null-safety

➢ Built-in types

➢ Functions

➢ **Operators**

➢ Control flow constructs

➢ Exercises

➢ Homework

➢ Resources

Operators

➢ As any programming language, Dart has a set of operators:

- Arithmetic
- Prefix, postfix
- Relational
- Logical
- Type tester
- Many others…

Operators - Arithmetic

```dart
import 'dart:math';

void main(List<String> args) {

    double a = 4;
    double b = 5;

    // --- Arithmetic operators
    print(a+b); //Add: this will print '9.0'
    print(a-b); //Subtract: this will print '-1.0'
    print(a*b); //Multiply: this will print '20.0'
    print(a/b); //Divide: this will print '0.8'
    print(a~/b); //Integer division: this will print '0'
    print(a%b); //Remainder of the division: this will print '4.0'

    // --- Arithmetic operations using the dart:math library
    print(pow(a,b)); //Elevate: this will print '1024.0'
    print(log(a)); //Logarithm: this will print '1.38...'
    print(pi); //This will print the PI constant

}//main
```

Operators – Prefix, postfix

```
void main(List<String> args) {


    int m = 0;
    int n = 0;
    n = m++; //First assigns to n the value of m, then it increments m
    print(m); //This will print '1'
    print(n); //This will print '0'

    n = ++m; //First increments the value of m, then assigns to n the
    value of m
    print(m); //This will print '2'
    print(n); //This will print '2'

}//main
```

Operators – Relational

```
void main(List<String> args) {

    int x = 0;
    int y = 1;
    print(x == y); //This will print 'false'
    print(x != y); //This will print 'true'
    print(x > y); //This will print 'false'
    print(x < y); //This will print 'true'
    print(x >= y); //This will print 'false'
    print(x <= y); //This will print 'true'

}//main
```

Operators – Relational

```
void main(List<String> args) {

    bool flag1 = true;
    bool flag2 = false;
    print(flag1 && flag2); //AND: This will print 'false'
    print(flag1 || flag2); //OR: This will print 'true'
    print(!flag1); //NOT: This will print 'false'
    print(flag1 ? 'Hello' : 'World'); //Ternary operator:
        This will print 'Hello'

}//main
```

Operators – Type tester

```dart
void main(List<String> args) {

  double c = 0;
  print(c is double); //This will print 'true'
  print(c is! double); //This will print 'false'

}//main
```

Outline

➢ Dart & Hello, world!

➢ Variables

➢ Null-safety

➢ Built-in types

➢ Functions

➢ Operators

➢ **Control flow constructs**

➢ Exercises

➢ Homework

➢ Resources

Control flow

➢ As any programming language, Dart has a set of constructs to control the program flow:
- If-Else
- For
- While
- Do-While
- Break
- Continue
- Switch-Case
- Try-Catch-Finally

Control flow – If-Else

```
void main(List<String> args) {

    //The following will print
    //a has 0 value.
    int a = 0;
    if(a == 0){
        print('a has 0 value.');
    } else if(a < 0){
        print('a is negative.');
    } else {
        print('a is positive.');
    }//if-else


}//main
```

```
if(c1){
    run this if c1 is true;
} else if(c2){
    run this if c2 is true;
} else {
    run this otherwise;
}
```

```
void main(List<String> args) {

    //The following will print
    //1
    //2
    //3
    var list = [1,2,3];
    for (var i = 0; i < list.length; i++) {
        print(list[i]);
    }//for

    //The following will print
    //1
    //2
    //3
    for (var item in list) {
        print(item);
    }//for



}//main
```

Control flow – While

```
void main(List<String> args) {

    //The following will print
    //3
    //2
    //1
    var n = 3;
    while(n>0){
        print(n);
        n--;
    }//while

}//main
```

> **Note:** Use for over while when you know how many iterations are needed

Control flow – Do-While

```
void main(List<String> args) {

    //The following will print
    //3
    //2
    //1
    var m = 3;
    do{
        print(m);
        m--;
    }while(m>0);


}//main
```

Note: Use for over do-while when you know how many iterations are needed

Control flow – Break

```
void main(List<String> args) {

    //The following will print
    //5
    //4
    var x = 5;
    while(x>0){
        print(x);
        x--;
        if(x == 3){
            break;
        }//if
    }//while

}//main
```

Control flow – Continue

```
void main(List<String> args) {

    //The following will print
    //5
    //3
    //1
    int y = 6;
    while(y>0){
        y--;

        if(y % 2 == 0){
                continue;
        }else{
                print(y);
        }//if-else
    }//while

}//main
```

Control flow – Switch-Case

```
void main(List<String> args) {

    //The following will print
    //Open the gate!
    var command = 'OPEN';
    switch (command) {
        case 'OPEN':
                print('Open the gate!');
                break;
        case 'CLOSE':
                print('Close the gate!');
                break;
    }//switch-case

}//main
```

Control flow – Try-Catch-Finally

```
void main(List<String> args) {

    // --- Exceptions, try-catch-finally
    //The following will print
    //Impossible to parse!
    //Try to correct the code.

    String s = 'A string.';
    try{
        var n = int.parse(s);
    } on FormatException catch (e) {
        print('Impossible to parse!');
    } catch (e) {
        // No specified type, handles all other than FormatException
        print('Something really unknown: $e');
    } finally {
        //This is run no matter what.
        print(Bye!');
    }//try-catch-finally

}//main
```

Full example in lab_02-dart_101_part_1/07-control_flow.dart

Outline

➢ Dart & Hello, world!

➢ Variables

➢ Null-safety

➢ Built-in types

➢ Functions

➢ Operators

➢ Control flow constructs

➢ **Exercises**

➢ Homework

➢ Resources

Exercises

- ➢ **Exercise 01.01**
  - ▪ In the main function, given a number *n*, for all non-negative integers *i* < *n*, print *i^4*. Do it using a for loop and a while loop.

- ➢ **Exercise 01.02**
  - ▪ In the main function, given an integer number *n*, instantiate a list of *n* random integers, with possible maximum value 10. Then, for each element of the list, print it multiplied by 2. (Big hint: https://stackoverflow.com/questions/11674820/how-do-i-generate-random-numbers-in-dart)

- ➢ **Exercise 01.03**
  - ▪ Write a function that given a string, provided by the main function, returns the same text with swapped cases. Then print the result in the main function.

- ➢ **Exercise 01.04**
  - ▪ Write a function with an optional named parameter *up*, that, for a given string, provided by the main function, make it upper case if *up* is true, lower case otherwise. By default, *up* is false. Then print the result in the main function.

- ➢ **Exercise 01.05**
  - ▪ Given an integer, *n*, perform the following conditional actions:
    - • If *n* is odd, print 'Odd'
    - • If *n* is even and in the inclusive range of 2 to 5, print 'Small even'
    - • If *n* is even and in the inclusive range of 6 to 20, print 'Medium even'
    - • If *n* is even and greater than 20, print 'Big even'

Exercises

➢ **Exercise 01.06**
- Write a function that given a string provided by the main function, returns a boolean that is true if the string is palindrome. Then print the result in the main function.

➢ **Exercise 01.07**
- Given a number *n > 0*, print the first *n* numbers of the Fibonacci series. Bonus: do it recursively.

➢ **Exercise 01.08**
- An Armstrong number is a number that is the sum of its own digits each raised to the power of the number of digits. For example:
  - 9 is an Armstrong number, because 9 = 9^1 = 9
  - 10 is not an Armstrong number, because 10 != 1^2 + 0^2 = 1
  - 153 is an Armstrong number, because: 153 = 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153
  - 154 is not an Armstrong number, because: 154 != 1^3 + 5^3 + 4^3 = 1 + 125 + 64 = 190

In the main function, write some code to determine whether a given number *n* is an Armstrong number.

➢ **Exercise 01.09**
- Write a function that given a string provided by the main function, converts that string to its acronym. Then print the result in the main function.
  - For example: Portable Network Graphics will generate PNG.

Exercises

➢ Exercise 01.10

▪ Write a function that given a string provided by the main function containing brackets [], braces {}, parentheses (), or any combination thereof that follows the math expression rules, returns a boolean that is true if any and all pairs are matched and nested correctly. Suppose that only one pair of parentheses can be present for each type. Then print the result in the main function. For example:

• This is ok: {this[is(o)]}k

• This is not ok: T{hi[(sis)not}ok]

• This is not ok: {{this[is(notok)]}}

Outline

➢ Dart & Hello, world!

➢ Variables

➢ Null-safety

➢ Built-in types

➢ Functions

➢ Operators

➢ Control flow constructs

➢ Exercises

➢ **Homework**

➢ Resources

Homework

➢ (At least try to) Do all the exercises

➢ Get familiar with Dart 101 part 1

➢ **Aim**: to be able to do the exercise without too much effort

Outline

➤ Dart & Hello, world!

➤ Variables

➤ Null-safety

➤ Built-in types

➤ Functions

➤ Operators

➤ Control flow constructs

➤ Exercises

➤ Homework

➤ **Resources**

Resources

- Code repository of today's lesson and exercises solution
  - https://github.com/gcappon/bwthw/tree/master/lab_02-dart_101_part_1

- Dart language tour
  - https://dart.dev/guides/language/language-tour

- Dart samples
  - https://dart.dev/samples