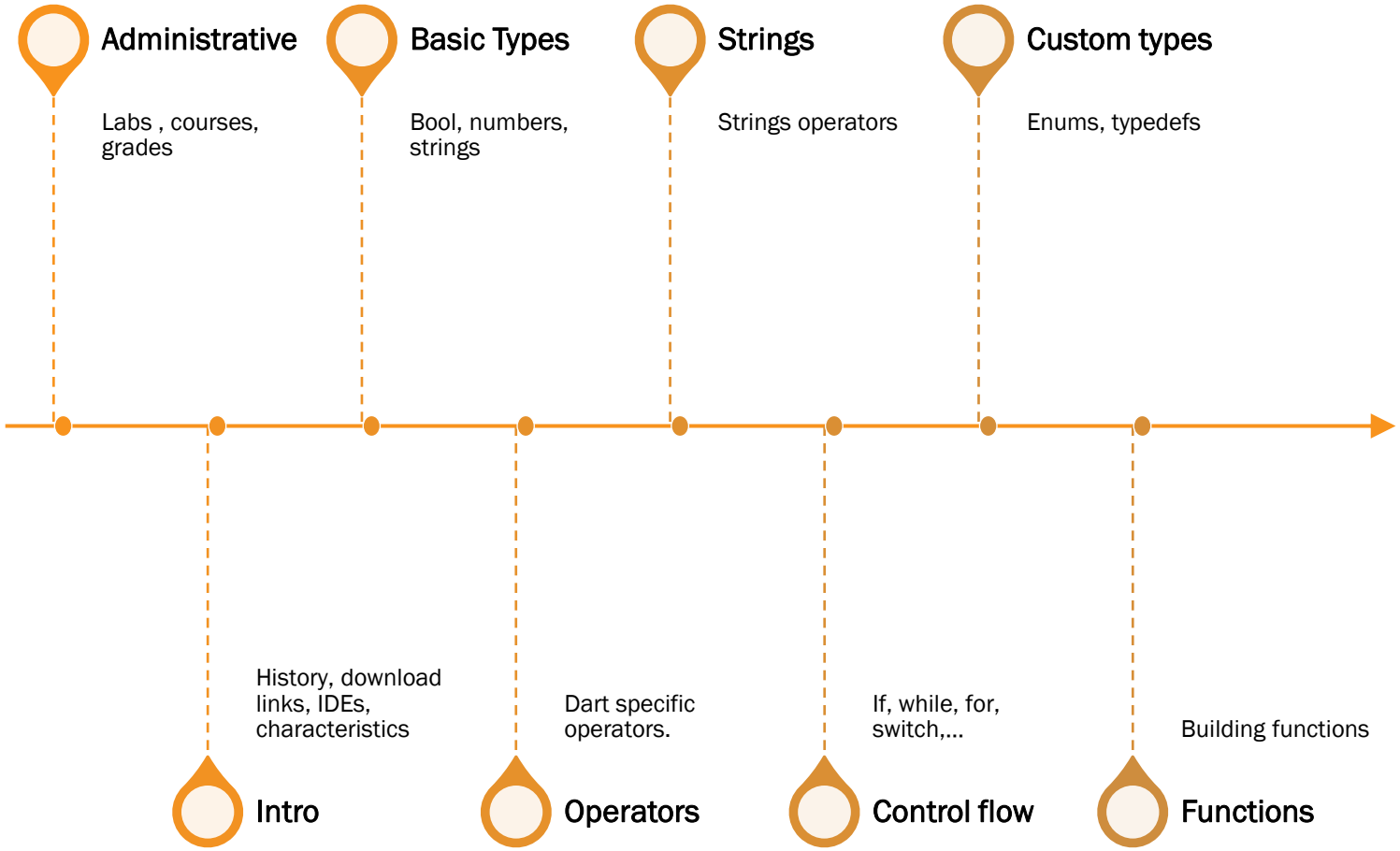
DART Language

COURSE 1 (REV 6)

GAVRILUT DRAGOS

Agenda


Administrative
Labs , courses, grades
Basic Types
Bool, numbers, strings
Strings
Strings operators
Custom types
Enums, typedefs
Intro
History, download links, IDEs, characteristics
Operators
Dart specific operators.
Control flow
If, while, for, switch,...
Functions
Building functions

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 programe

HelloWorld.dart

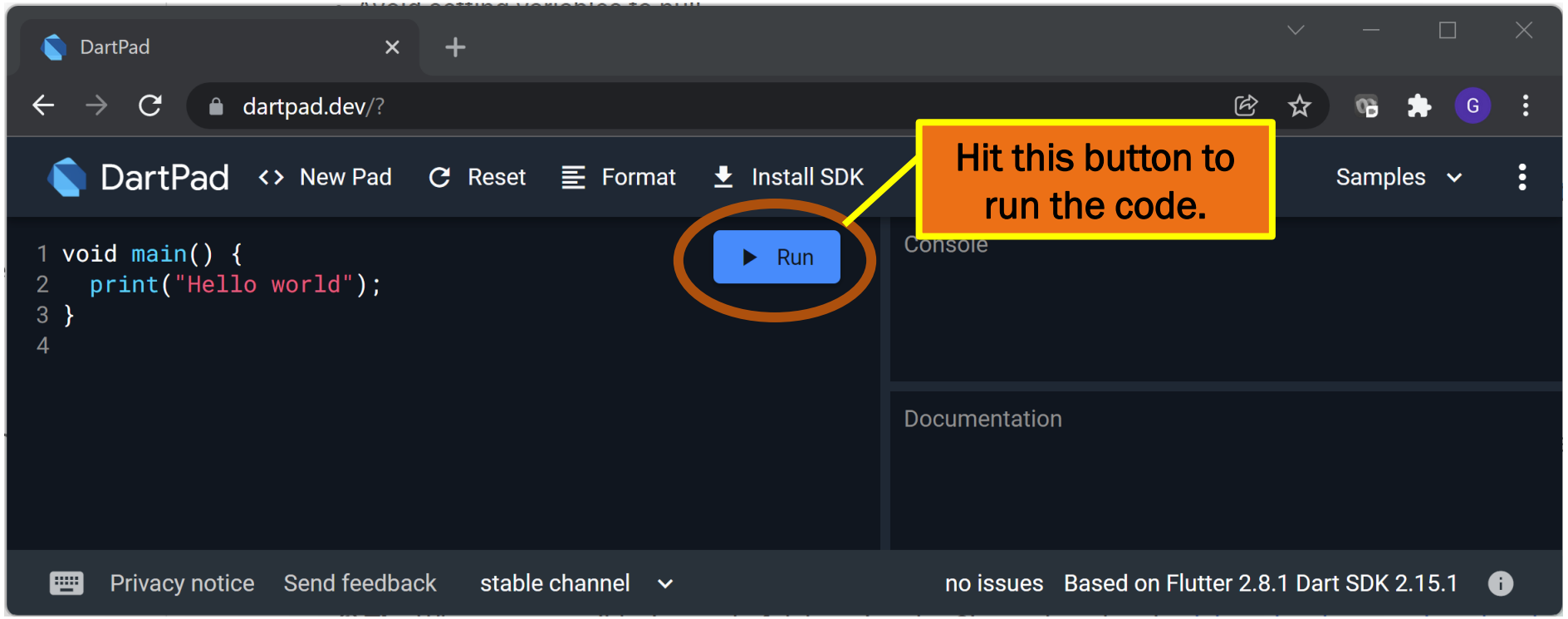
```
void main() {
        print("Hello, World");
}
```

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 programe

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 quottes
var s = "test";         // double quottes
var s = "te\nst";       // double quottes 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 exits) 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: $expression_1$?? $expression_2$
  The logic for this operator is as follows. If $expression_1$ is not null, the result will be expression 1, otherwise it will be $expression_2$

```
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 ***endsWidth*** 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 incrementations 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** valu1: … , **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** valu1: … , **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** valu1: … , **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> { $value_1$, $value_2$, … $value_n$}

- 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> { $value_1$, $value_2$, … $value_n$}

- 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> { $value_1$, $value_2$, … $value_n$}

- 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 expressions (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

```
void my_function({int x = 20, int y = 10}) {…}
```

```
int sum(int base, {int x=10, int y=20}) => base+x+y;
void main() {
        print(sum(10,x:5));          // 35 [10 + x=5 + y=20]
        print(sum(10,y:10));         // 30 [10 + x=10 + y=10]
        print(sum(10));              // 40 [10 + x=10 + y = 20];
        print(sum(10,y:5,x:3));      // 18 [10 + x=3 + y=5]
        print(sum(10,2,3));          // error (x,and y must be specified
                                     // by their name)
}
```

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)); // 2oo
        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)); // 2oo
        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