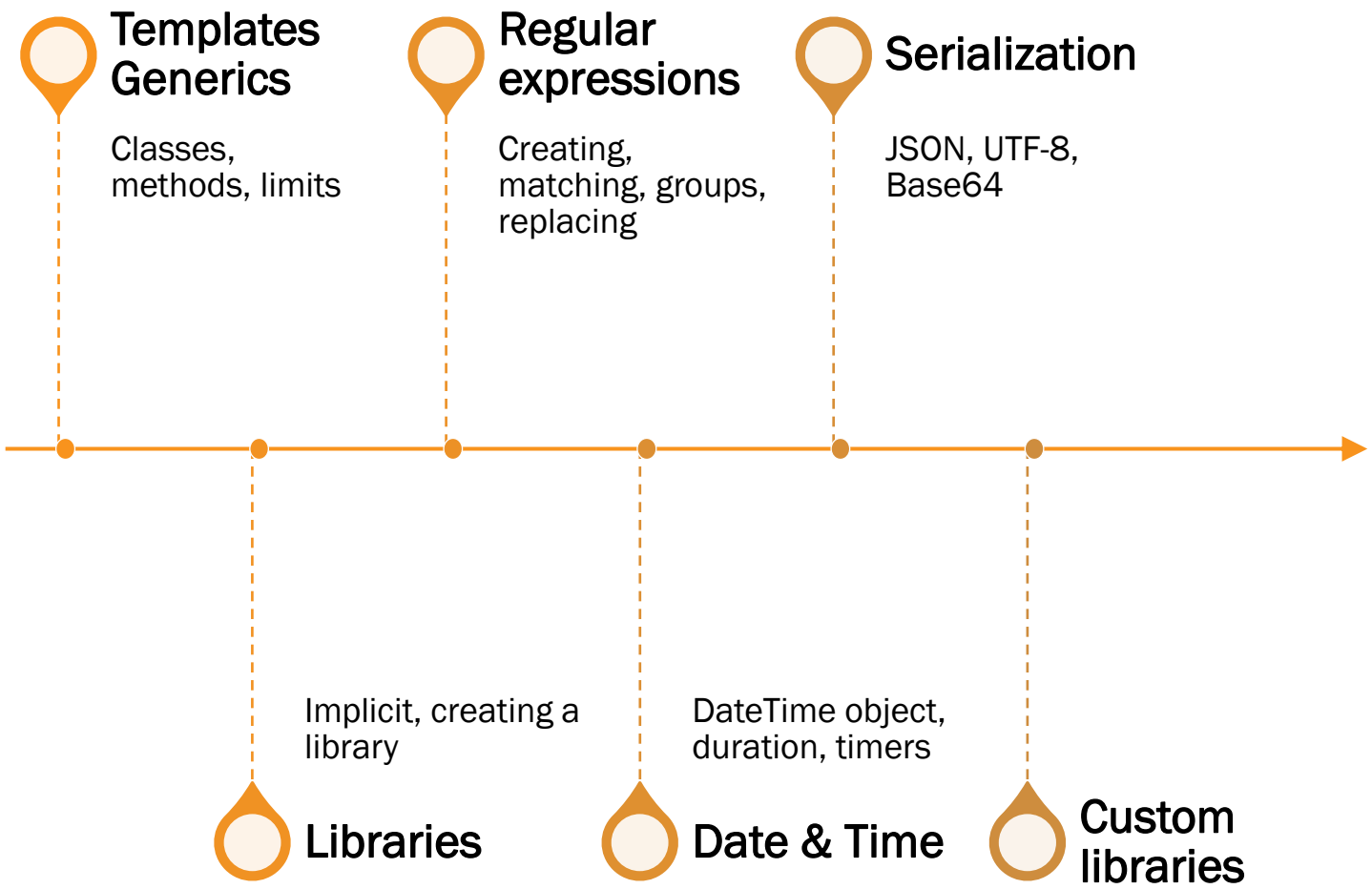
DART Language

COURSE 4 (REV 3)

GAVRILUT DRAGOS

Agenda


Templates
Generics
Classes, methods, limits
Regular expressions
Creating, matching, groups, replacing
Serialization
JSON, UTF-8, Base64
Implicit, creating a library
Libraries
DateTime object, duration, timers
Date & Time
Custom libraries

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));     // 0.49999999999999994
        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:

`File (String path)`	Creates a new File object
`File.fromRawPath(UInt8List path)`	Creates a new File object from a raw path
`File.fromUri(UInt8List path)`	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:

`bool RegExp.isCaseSensitive`	If the regular expression is case sensitive
`bool RegExp.isDotAll`	If '.' (dot) should represent all characters
`bool RegExp.isMultiLine`	If the pattern represent a multi-line object
`bool RegExp.isUnicode`	If the pattern is based on unicode

Libraries – Regular expressions

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

Methods:

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

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:

```
bool   String.contains(Pattern p, [int start = 0])
bool   String.startsWith(Pattern p, [int start = 0])
int    String.indexOf(Pattern p, [int start = 0])
int    String.lastIndexOf(Pattern p, [int start?])
String String.replaceAll(Pattern p, String replace)
String String.replaceFirst(Pattern p, String replace, [int start = 0])
String String.replaceAllMapped(Pattern p, String fnc(Match m))
String String.replaceFirstMapped(Pattern p, String fnc(Match m),[int start=0])
String String.splitMapJoin(Pattern p, {String onMatch(Match)?,
                                       String onNonMatch(String)?} )
```

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]?.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:

DateTime	DateTime.add (Duration d)
DateTime	DateTime.substract (Duration d)
int	DateTime.compareTo (DateTime d)
bool	DateTime.isAfter (DateTime d)
bool	DateTime.isBefore (DateTime d)
bool	DateTime.isAtSameMomentAs (DateTime d)
DateTime	DateTime.toLocal ()
DateTime	DateTime.toUtc ()

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:

DateTime	DateTime.parse (String txt)
DateTime?	DateTime.tryParse (String txt)

The following strings are accepted forms for the parser:

yyyy-MM-dd	yyyyMMdd hh:mm:ss
yyyy-MM-dd hh:mm:ss	yyyyMMddThhmmss
yyyy-MM-dd hh:mm:ss.msz	yyyyMMdd
yyyy-MM-dd hh:mm:ss,msz	yyyy-MM-ddThhZ

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<10000000;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 object 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.

```
import "dart:convert";

class Test {
        int x=10,y=20;
}
void main() {
        var s = JsonEncoder().convert(Test());      // Uncaught Error:
                                                    // Converting object to
                                                    // an encodable object
                                                    // failed: Instance of
                                                    // 'Test’
}
```

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("ăîșțĂÎȘȚ");
        print(diacritice); // [196, 131, 195, 174, 200, 153, 200, 155, 196,
                           // 130, 195, 142, 200, 152, 200, 154]
        txt = utf8.decode(diacritice);
        print(txt); // ăîșțĂÎȘȚ
}
```

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 is 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:

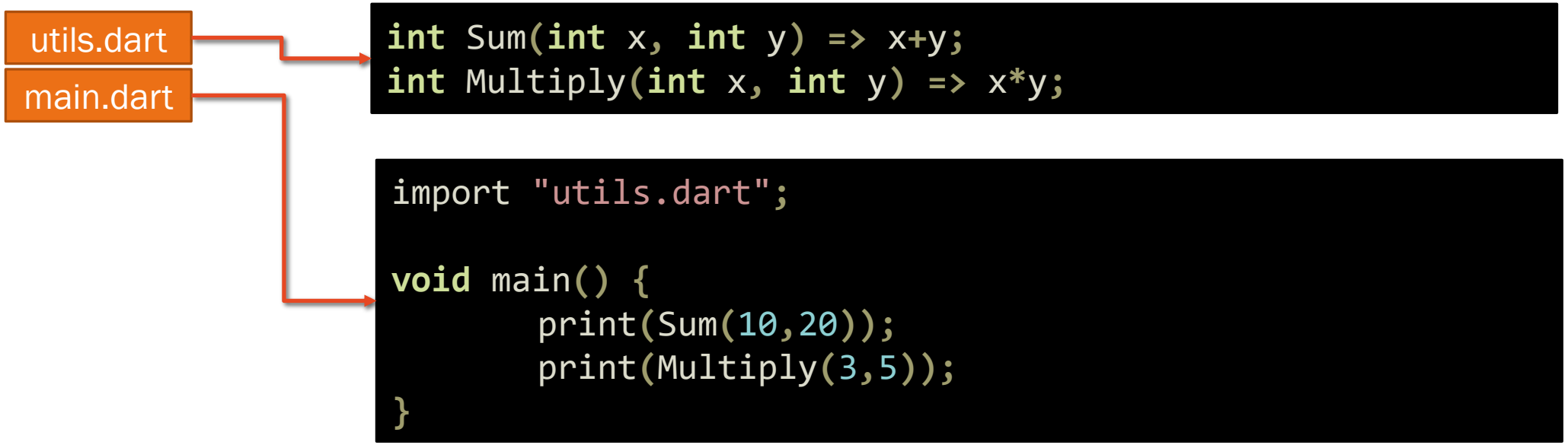
```
import "dart:convert";

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

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.Set(10);
        print(m.Add(20));
}
```

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:

src/utils.dart

main.dart

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

```
import “src/utils.dart";
void main() {
        var m = Math();
        m.Set(10);
        print(m.Add(20));
}
```

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.

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

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