Dart Basics

Dart Fundamentals You Should Know!

Important
This document is just a short summary of the basics taught in this module! If you want to learn more/ all about Dart immediately (instead of throughout the course lectures), you can scroll to the **end of this document**. There, you find a link to further resources!

What is Dart?

Dart is an object-oriented programming language developed by Google. Whilst technically not restricted, it's primarily used for creating frontend user interfaces for the web (with AngularDart or Flutter for Web) and mobile apps (Flutter).

It's under active development, compiled to native machine code (when used for building mobile apps), inspired by modern features of other programming languages (mainly Java, JavaScript, C#) and strongly typed.

As already mentioned, Dart is a compiled language. That means, that your code isn't executed like you write it, but instead, a compiler parses + transform it (to machine code).

Variables, Functions, Types

Like pretty much all programming languages, Dart has a couple of core features it supports - for example:

- **Variables** to temporarily store data
- **Functions** to executed code "on demand"
- **Types** to analyse your code before you run/ test it

Variables are a core building block that allows you store results of other operations - or any other data you want to use at a later point of time.

```
var myName = 'Max';
print(myName); // Outputs 'Max'
```

In the above snippet, we see how a variable is created (with the help of the `var` keyword), how a value is assigned (with the help of the `=` sign) and how you can then use the variable - for example to simply out (`print`) it to the system log.

Regarding the naming, it's common that you use **camelCase** notation to name your variables - and you want to name such that it's clear what's "inside of the variable".

The `myName` variable is of type **string**. Why? Because we initially assign a value of type string to the variable. Instead of `var`, you could also use the type name to initialise the variable.

```
String myName = 'Max';
print(myName); // Outputs ‘Max’
```

Since Dart is able to infer the type though (because you initialise the variable with a value - ‘Max’), it’s considered a better practice to use `var` instead of `String`. Technically, both would work though.

But what are types? Types describe the type of data of something - e.g. of a variable (you can also use types in other places, see below). Types are a useful feature because they force you to write clean code and avoid mistakes/ bugs. For example, an addition might only work with two numbers but not with two strings. With types, you can ensure that a certain code expression only accepts numbers and the compiler “yells at you” if you pass wrong data.

Here are some of the core types Dart offers - please note that this list only includes important types used in this module. Not all types Dart knows!

Type	Example	More Information
String	String myName = 'Max';	Holds text. You can use single or double quotes - just be consistent once you made your choice.
num, int, double	int age = 30; double price = 9.99	num refers to “number” - there are two types of numbers in Dart: Integers (numbers without a decimal place) and doubles (numbers with decimal places)
object	Person = Person()	Generally, everything in Dart is an object - even an integer. But objects can also be more complex, see below.

Besides variables and types of values, you also have another core concept in Dart (and in basically any other programming language as well): Functions.

Functions allow you to define code snippets which you can call whenever and as often as you want.

```
void sayHello(String name) {
  print('Hello ' + name);
}
```

Just as variables, functions use camelCase notation for the naming. The name typically also can be read like a short phrase, describing what the function does (sometimes also describing when the function is used).

In the above snippet, you also see another example of types being used: We define both the return type (`void`) and the argument type (`String`) of the function.

What does that mean? “Return” and “argument”?

Functions can (but don’t have to) return values and they also can (but don’t have to) accept input - so called “arguments” or “parameters”.

The example function above does accept one argument (it could accept more than one if you needed that behavior) and it returns nothing (=> `void` is a type that signals that nothing is returned).

Here’s a function example that takes two arguments and also returns something:

```
double addNumbers(double n1, double n2) {
  return n1 + n2;
}
```

This function accepts two `double` values as input and returns the sum of them.

Arguments (i.e. the input to a function) are basically variables which are only available inside of the function body (the part between the `{ }`).

Objects & Object-orientation

Dart is an object-oriented programming language - that means that every value in Dart is an object. Even a simple number. The idea behind object-orientation is, that you think of all data structures as objects. A bit like in the “real world”. You have a car which can drive. In programming, you might have a user who can logIn().

What are “objects” though?

Objects are data structures - you find them in a lot of programming languages. In Dart, every value is an object, even primitive values like text (= String) or numbers (= Integers and Doubles). But you also have more complex built-in objects (e.g. Lists of data) and you can build your own objects.

You often build your own objects if you want to express more complex relations between data or if you want to encapsulate certain functionality in “one building block”.

Objects are created with the help of “Classes” because every object needs a blueprint (=> the class) based on which you can then create (“instantiate”) it.

Here's an example class definition:

```
class Person {
  var name = 'Max';
  var age = 30;

  void greet() {
    print('Hi, I am ' + name + ' and I am ' + age.toString() + ' years old!';
  }
}
```

In this example, we define a `Person` class which has two class-level variables (also called "instance fields" or "properties") and one class-level function (also called "method").

As you can see, we also use types in classes - for both properties (variables) and methods (functions).

You can also see, that inside of the `greet` method, we can access the class properties `name` and `age` without issues (`age.toString()` is used to covert the integer value to a string whilst outputting it in a longer string).

The class only serves as a blueprint though! On its own, it does not give you an object! Instead, you can now create objects based on this class:

```
class Person {
  var name = 'Max';
  var age = 30;

  void greet() {
    print('Hi, I am ' + name + ' and I am ' + age.toString() + ' years old!';
  }
}

void main() {
  var myself = Person();
  print(myself.name); // use the . to access class properties & methods
}
```

As a side note: The `main` function is a special function in Dart - it's the function which Dart will execute first, when your app starts.

Inside of `main`, we then create a new object based on `Person` by using `Person()`. This process is called "instantiating the class", hence we create "an instance of `Person`". The **type of** `myself` would then be `Person` because **classes always also act as types**!

Constructor Functions

It might look strange that you create an instance of a class (i.e. you create an object) by calling the class like a function.

That makes more sense, however, once you understand that each class has a special method - which you can (but don't have to) add to it (if you don't add it, it invisibly has an empty default method): The constructor method.

Here's how you would add a constructor to `Person`:

```
class Person {
  String name;
  int age;

  Person(String name, int age) {
    this.name = name; // this.name refers to the “name” property
    this.age = age;
  }

  void greet() {
    print('Hi, I am ' + name + ' and I am ' + age.toString() + ' years old!';
  }
}
```

A constructor method is added by repeating the class name inside of the class and by basically writing it like a normal class method (without a return type though).

The constructor can (but doesn't have to) take arguments which you then can use in the constructor body. Typically, these arguments are then used to initialise some class properties (in this example `name` and `age`). Using such a constructor has the advantage, that you can create different, individually configured instances of this class - all based on the same class but with different values for `name` and `age`.

```
var mySelf = Person('Max', 30);
var someoneElse = Person('Michael', 45);
```

It can be confusing, that the constructor takes two arguments `name` and `age`, when we also have to class properties with the same names.

It's important to understand, that Dart has a concept called “Scoping”. It scopes (= “limits the usage”) name and age received as arguments to inside the constructor method. I.e., you can only use name and age inside the constructor method.

`name` and `age` on the class level are available in the whole class though. To tell them apart, you can use `this.name` and `this.age` to make it clear that you're referring to the class-level properties. Without the special `this` keyword, Dart assumes that you're referring to local (method-level) variables named `name` and `age`. Only if such variables would not exist, Dart would fall back to the class-level properties.

Classes can also have more than one constructor method! You can define as many “utility constructors” (not an official term) as you want - typically to create differently pre-configured instances of a class:

```
class Person {
  String name;
  int age;

  Person(String name, int age) {
    this.name = name; // this.name refers to the “name” property
    this.age = age;
  }

  Person.sixty(String name) {
    this.name = name;
    this.age = 60; // not configured dynamically!
  }

  void greet() {
    print('Hi, I am ' + name + ' and I am ' + age.toString() + ' years old!';
  }
}
```

`Person.sixty` is an additional constructor in this case. You simply add extra constructors by repeating the class name and then adding `.anyNameYouWant`.

When using Flutter, you’ll find quite a lot of classes (built into Flutter) that take advantage of such extra constructors!

Last but not least, you can also improve and shorten the code a little bit (and you should):

```
class Person {
  final String name;
  final int age;

  Person(this.name, this.age);

  ...
}
```

When you have properties (like `name` and `age`) which you only set inside of the constructor and which never change thereafter (i.e. to which you never assign a new value thereafter), you should mark them as `final`.

`final` is a special keyword in Dart that will cause an error if you then accidentally do change these properties - hence it improves code quality as it forces you to be clear about your intentions.

In addition, constructors, where you only accept arguments to them assign them to class properties, can be shortened. By using the shortcut you see above (no constructor body, `;` after the constructor, `this.` in front of the argument, no types being used, arguments match property names) you instruct Dart to automatically store the argument values in the fitting properties of the class.

Positional & Named Arguments

No matter if you're working with constructor methods, normal methods or normal functions, you can always accept two types of arguments: Positional and named arguments.

Thus far (in the above examples), we only used positional arguments - i.e. the first value you pass into the function when calling it, is passed into the first argument. The second value is passed to the second argument etc.

As an alternative, you can use named arguments - those are especially useful if you got functions with many arguments. Remembering the position of each argument (when calling the function) can be difficult - using names can be easier, especially since IDEs like Visual Studio Code help you with completion.

```
void doSomething({String name, double price, String description}) { ... }
```

In this example, `name`, `price` and `description` are named arguments - simply by wrapping the argument list with `{ }`.

You can also mix positional and named arguments, the positional arguments have to come first though:

```
void doSomething(String name, {double price, String description}) { ... }
```

How do you call a function with named (or mixed) arguments?

```
doSomething('A Book', price: 9.99, description: 'An awesome book!');
```

You "target" a named argument by using it's name and a colon (`:`) and then the value you want to assign.

Named arguments are always optional by default! That means, that you could call a function/ method without providing a value of the argument. Therefore, you should ensure that your function works if no value is provided. Or you set a default value:

```
void doSomething(String name, {double price = 5.99}) { ... }
```

When working in a Flutter app, you can also add a so-called “Decorator” to a named argument that marks it as “required” (it adds metadata behind the scenes that causes the Dart compiler to “yell at you” if you are not providing a value when calling the method):

```
void doSomething(String name, {@required double price}) { ... }
```

Side-note: You can also have optional positional arguments - also with default values if you want to:

```
void doSomething(String name, [double price, String desc = 'Default']) { ... }
```

Inheritance

Back to classes and objects. One important concept, which you’ll see a lot when working with Flutter, is “Inheritance”.

“Inheritance” means that a class can be based on another class (and only one one other class, not on multiple classes).

```
class Person {
  void greet() {
    print('Hi!');
  }
}

class Friend extends Person {
  void greetFriendly() {
    print('Amazing to see you!');
  }
}

void main() {
  var myFriend = Friend();
  myFriend.greet(); // works and prints “Hi”
  myFriend.greetFriendly(); // also works and prints “Amazing to see you!”
}
```

In this example, `Friend` is a class that inherits from `Person` (by using the `extend` keyword). The `Friend` class now has all properties and methods of `Person` but it can also add its own properties and methods. On every instance of `Friend`, you can therefore use everything you could use on a `Person` instance, and you can of course access all features of `Friend`. Every instance of `Friend` also automatically is of type `Friend` and of type `Person`!

You can also override methods (and properties) of base/ parent classes:

```
class Friend extends Person {
  @override
  void greet() {
    print('Hello there!');
  }

  void greetFriendly() {
    print('Amazing to see you!');
  }
}
```

Using `@override` is recommended (technically, it's not required though) since it makes it really clear that you are deliberately (and not accidentally) overriding a method of the base class.

final & const

Dart knows three types of variables/ properties:

1. **"Real variables"** (created via `var` or the type name)
2. **Final variables** (created via `final`, `final` + type name when creating uninitialised variables)
3. **Constants** (created via `const`)

The difference between "real variables" (I invented this term, it's not an official term) and final or constant variables should be relatively clear: "Real variables" can change their values, final or constant variables can't.

This causes an error for example:

```
final myName = 'Max';
myName = 'Manu';
```

Dart would complain about a final variable being re-assigned.

It makes sense to mark variables as final if you don't intend to change them - a scenario you'll encounter quite often in Flutter programs.

But how does `final` differ from `const`?

`final` values are runtime constants, whereas `const` values are compile-time constants.

That means that a `final` value doesn't have to be known exactly at the time you're writing the code. For example, a value entered by an user of the app might not change thereafter (i.e. it's `final`) but you don't know it when you write the code for the app.

`const` values on the other hand are known at the point of time you're writing the code. For example some text you want to output in a widget - it might not change at runtime and hence is compile-time constant.

```
const title = 'Your Products'; // this can't change at runtime!
```

"if" Statements & Ternary Expressions

Quite often in coding, you have certain code that should only run if some condition is met (or not met).

```
var age = getFromUserInput(); // assume that this is entered by a user

if (age > 30) {
  runCodeForOldUser();
} else if (age < 25) {
  runCodeForVeryYoungUser();
} else {
  runCodeForUserBetween25and30();
}
```

`if` statements allow you to run different code snippets based on one or more conditions (multiple conditions can be combined with a logical AND - `&&` - or a logical OR - `||` -).

You can also define `else` or `else if` blocks in an if statement to define code that runs if the condition is not met (`else`) or if it's not met and some other condition is true (`else if`).

Where to learn more

This document only touches on some of the basics of Dart - and also only on some basics that are shown in this module.
Throughout the course, you'll learn more and more about Dart.

In case you want to learn way more about it right now already - or in case you want to dive a bit deeper into the things mentioned above - the official docs are a great place to learn more: https://dart.dev/guides/language/language-tour