Dart Class, Interfaces & Inheritance

By: Laams Innovation Lab

Revision

- Data types, variables, functions, operators,
- Structural thinking, data structures, Json
- Relation Between Objects and Classes
- Types and parts of constructors
- Setters and Getters
- Create a user class with properties first, last name, and full name. Create a specific procedure to change each.

Content

- Class and all of its parts.
- Inheritance
- Interfaces and implementation
- Mixins
- Extension Methods
- Generics

All Parts of a Class

- Instance Related Parts:
  - Instance Variables => Object Properties
  - Instance Methods => Object Behaviors
  - Getters & Setters => Read and Write Access to Object Properties
  - Constructors => instantiates Objects.
  - Method Overrides
  - Operators
- Class Related Parts:
  - Static Variables
  - Static Methods

Method Overrides

You can **override** a defined method in a **super class**, by re-defining a method of the **same identifier**, the **same return type** (or sub-type) and **the same positional parameters type(s)** (or sub-types) in a **subclass**, marked with an **@override** annotation.

To override == operator, hashCode getter must also be overridden.

Classes and Privacy

To make a **class** or its **method**(s) and **variable**(s) private add an underscore (_)before their identifier(s).

Class Variables & Methods

Static variables and **methods** do not operate on a class **instance**, meaning they do not have access to ***instance*** *&* ***methods*** and the **this** Keyword, and they cannot be accessed as **properties** or **methods** once an **Object** is **instantiated** from the **class**; however, they are accessible to each other, and can also be used by the **instance**.

They act like **global variables**, and **top-level function**, defined inside a class.

Since, functions themselves are first-class citizens in dart, it is **recommended** to use **top-level functions** instead of **static methods** for **commonly** used utility **functions** in **Dart**.

Immutable Classes

Immutable classes are those whose **instance variables (states)** cannot be changed once they are **instantiated**. To create an immutable class in Dart, all its variables must be **final**.

To make a class a run-time constant a **const** keyword *must* be added at the beginning of **unnamed** or **named constructor**(s) in a class.

To change **immutable classes state(s)**, by convention, a **copyWith**() **method** is with the class **instance** return **type** is **defined** inside the **class**.

Class & Inheritance

Inheritance helps with **categorization** and **classification** of **classes** and their **instantiated objects** ,by declaring **generalize-able** variables (**properties**) and methods (**behaviors**) in a **super-class** (generalized, usually abstract) to be ***inherited*** by all the **sub-classes** (specific, usually concrete) **extending** the super-class, leading to a modular software, with less repeated code.

To **define** a **sub-class** inheriting from a **super-class**, use the **extends** keyword followed by the **identifier** of **super-class**, right after the **sub-class**' **identifier**.

Abstract Classes

Abstract classes are those classes which cannot be **instantiated**, meaning they do not create **objects** directly, unless it has **factory** or **static** **constructor(s)**, and **concrete** **sub-type(s)** to be **instantiated** from those constructors.

Abstract classes are created by adding an **abstract** keyword before a class **definition**.

Abstract classes are **useful** for creating **interfaces**, and a generalized **super class**, whose instantiation is not semantically logical.

Abstract classes often have **abstract methods**, which are **function declarations** with no **concrete implementation**. To create an abstract method, instead of the **function block**, a **semicolon** is added.

Interfaces and Implementation

An Interface, when **defined**, declares a **contract** to be **implemented** by its **sub-classes**, and when used, ***provides*** a clear **API** to the user.

In Dart every **concrete** or **abstract** **class** is implicitly an **interface**. Hence, implement any class as an interface use the keyword **implement**.

The difference between **implements** and **extends** keywords are that when **implementing**, the **sub-class** re-implements all **instance variables**, and **concrete** or **abstract methods**. While, by extending those are inherited.

Also, opposed to extends keyword, the class can implement multiple classes.

Mixins

To **reuse** codes in **multiple class hierarchies** we can use **Mixins**.

A class with no **constructors**, which **extends Object**, can be used as a **Mixin**.

To **create** a Mixin, use the keyword **mixin** with an **identifier** and a **block**, inside which variables and methods are defined.

To **use** a **Mixin** with a class use the **with** keyword after the **class** identifier followed by the **Mixin's** identifier.

To **limit** a Mixin's use to a specific **type**, use the **on** keyword after its identifier followed by the target **type's** identifier.

Extension Methods

You can add **functionality** to **class** without **extending** or **knowing** what functionalities the **class** has using **extension** methods.

Generics

All ready explained.

Module & Components

To create a software **module** or **component** you use a **combination** of **abstract**, **concrete**, **generic classes**, **class hierarchies**, **interfaces**, **static**, **concrete** and **asynchronous methods** and **functions**, and **static** or instance **variables** of different **types**.

Hence, Review the Dart Language Tour.