



جامعة دمشق
الكلية التطبيقية
قسم تقنيات الحاسوب

البرمجيات النقالة

- ١ -

What is Flutter ?

- Flutter is an open-source UI software development kit created by Google.
- It is used to develop applications for Android, iOS, Windows, Mac, Linux, Google Fuchsia and the web.

Deep Into Mobile Platforms



Mobile Platforms



Mobile Development Approaches



Native App Approach



Web App Approach



Hybrid Approach



APACHE
CORDOVA™



Phone**Gap**

Other popular Hybrid Approach



JavaScript

Damascus University

So what is Flutter ?

- Flutter uses hybrid approach
- Flutter is Google's UI toolkit for building beautiful, natively compiled applications for mobile, web, and desktop from a single codebase.



So what is Flutter ?

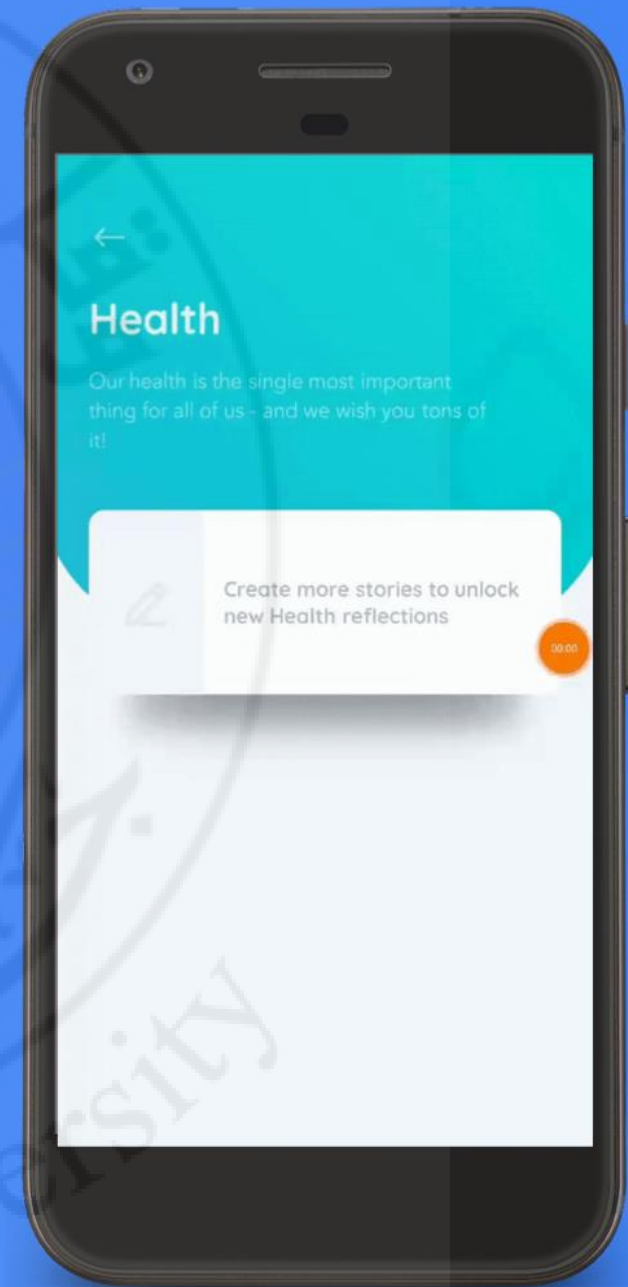
- Flutter is the next step in App development

1. Expressive, beautiful UIs
2. Beautiful Animations
3. Fast
4. Productive



1-Expressive, beautiful UIs

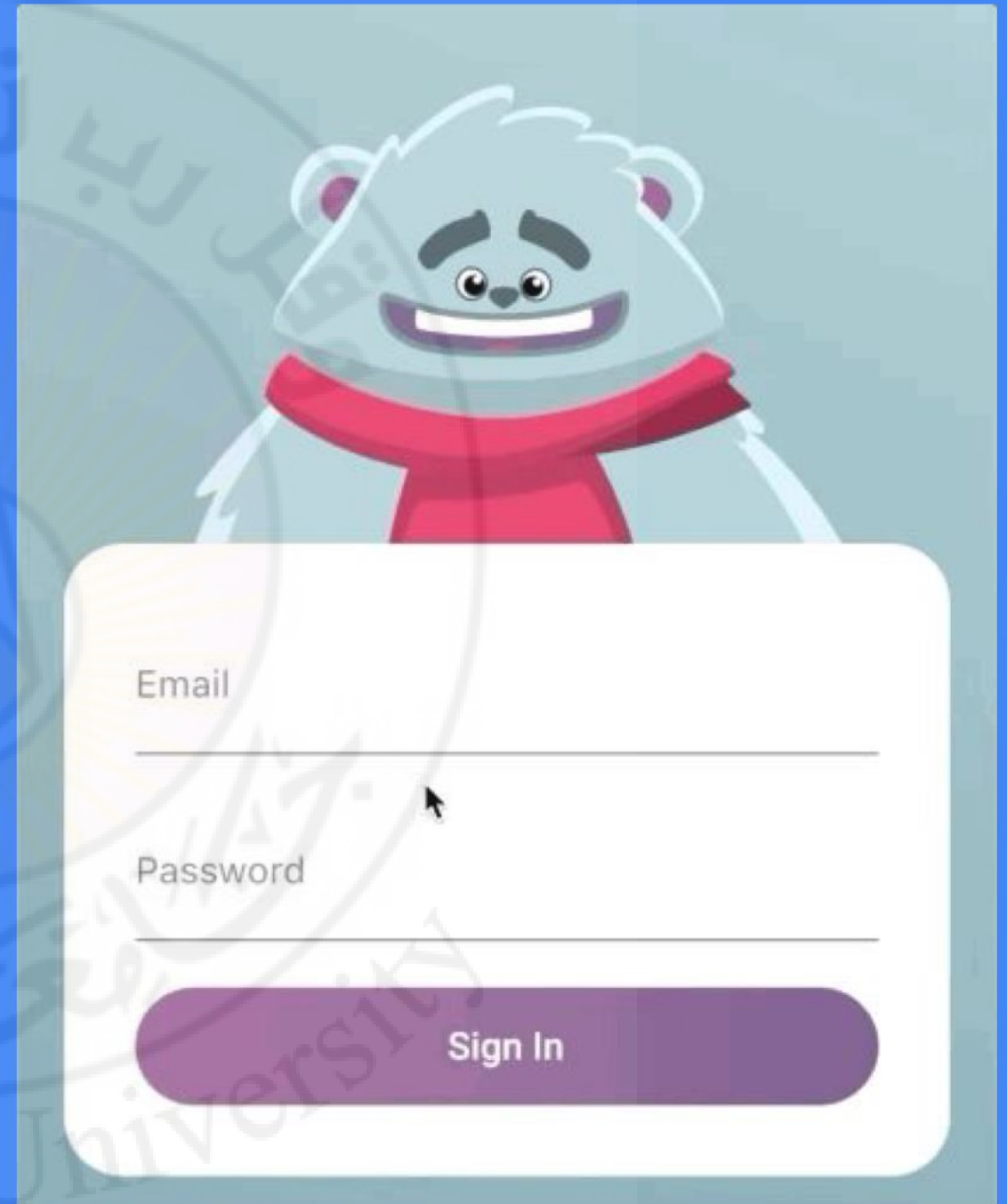
- Control every pixel on the screen
- Make your brand come to life
- Never say "no" to your designer
- Stand out in the marketplace
- Win awards with beautiful UI



2-Beautiful Animations

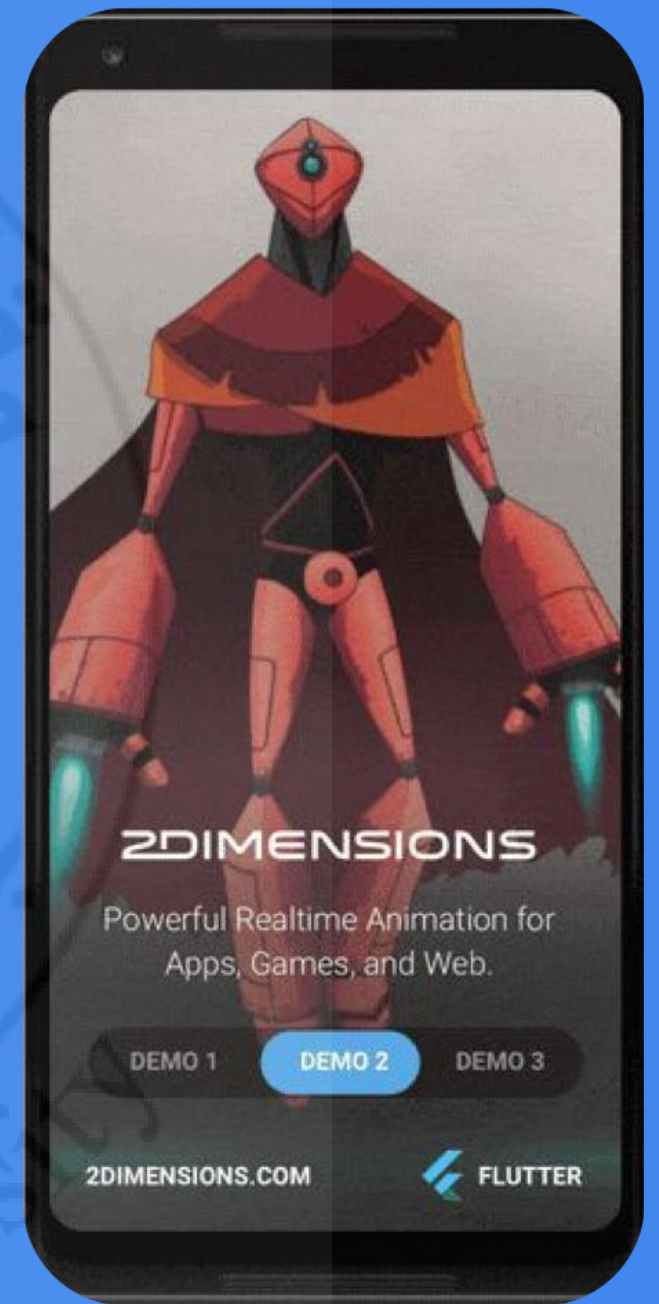
Flutter support many types of Animations

- Tween
- Hero
- Sliver
- Transform
- FadeInWidget
- Animation Builder
- AnimatedOpacity
- Physics-based animation



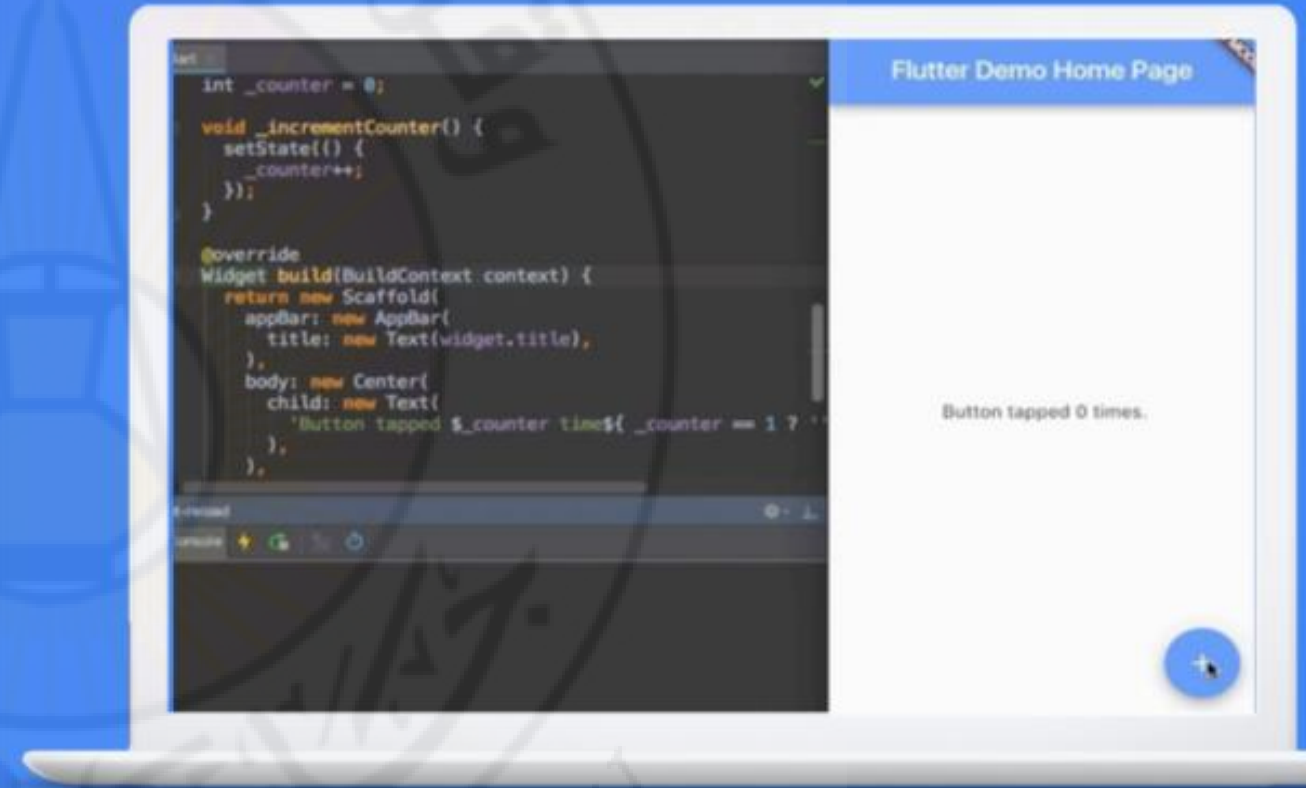
3-Fast

- Brings the power of a games engine to user experience development
- 60fps, GPU accelerated
- Compiled to native machine code



4-Productive

- Sub-second reload times
- Paint your app to life
- Iterate rapidly on features
- Test hypotheses quicker than ever
- More time to experiment & test features
- Single-codebase for faster collaboration
- 3X Productivity Gains



What makes Flutter unique?

- Compiles to Native Code (ARM Binary code)
- No reliance on OEM widgets
- No bridge needed
- No markup language (only Dart)



What language is Flutter built with?



Dart

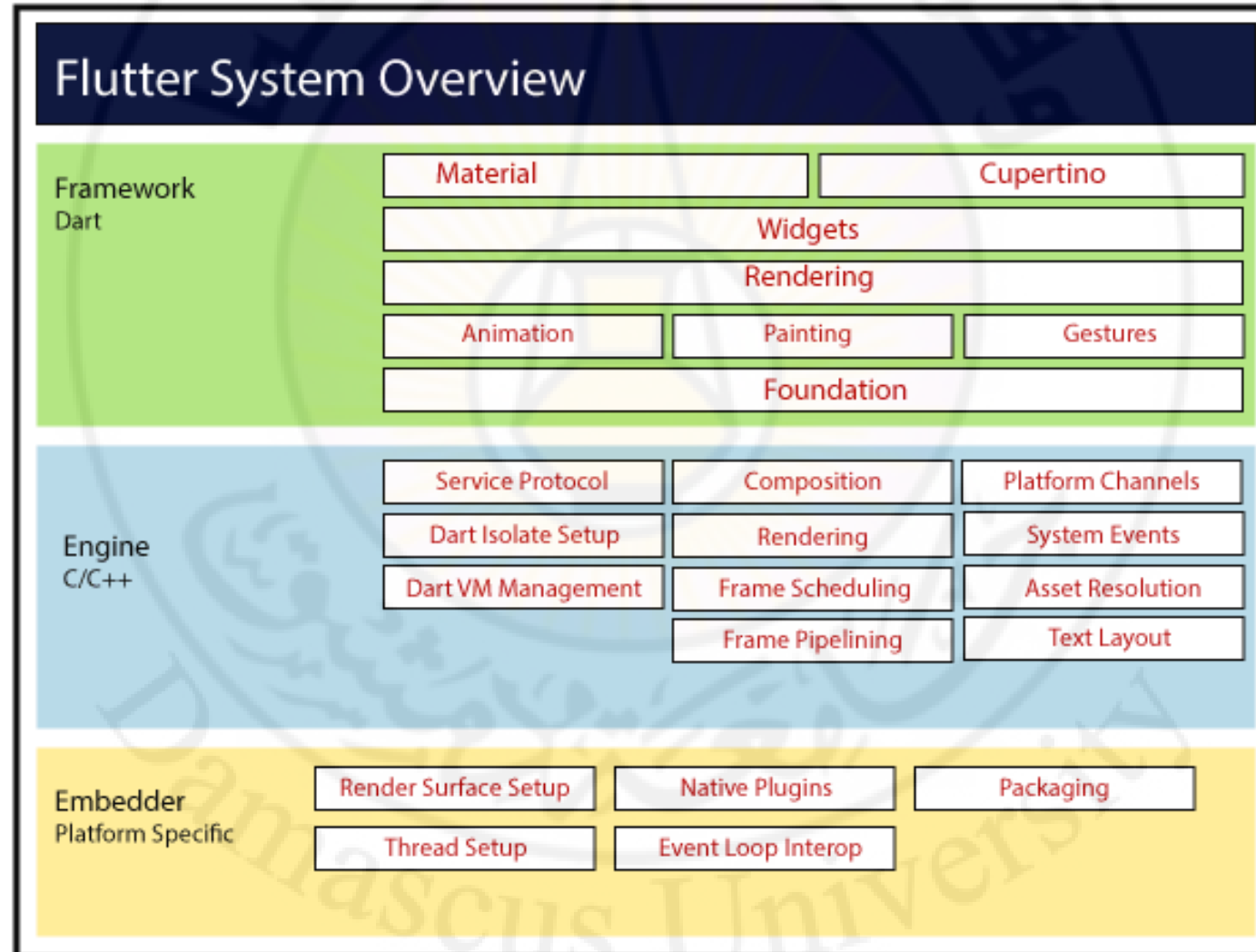
- Dart is a client-optimized language for fast apps on any platform! (Web - Desktop - Mobile - Embedded)?
- Language and Libraries
- Packages manager <https://pub.dev>
- Virtual machine
- Compile to Javascript `dart2js`



Deep Into Flutter



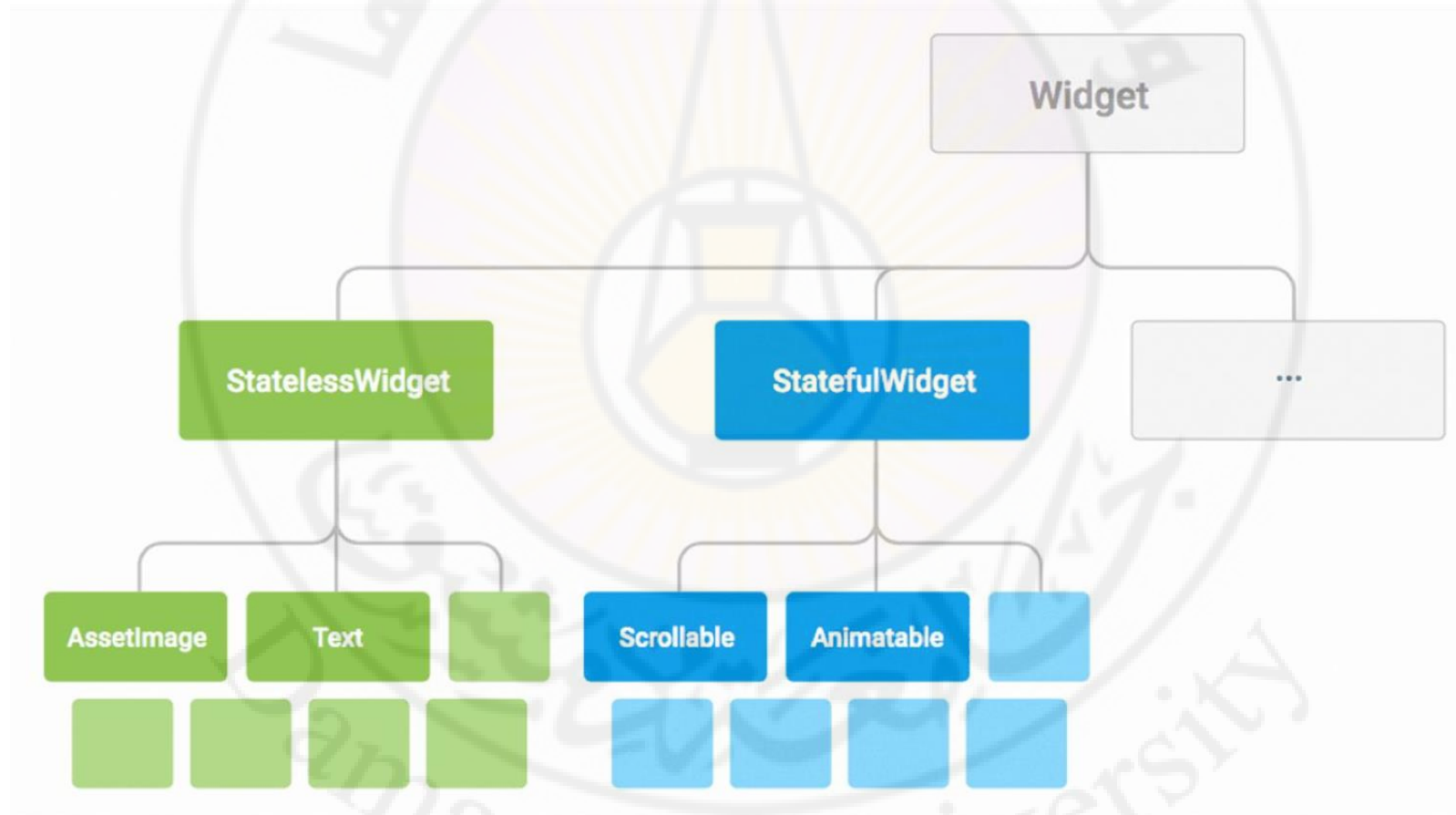
Flutter Architecture



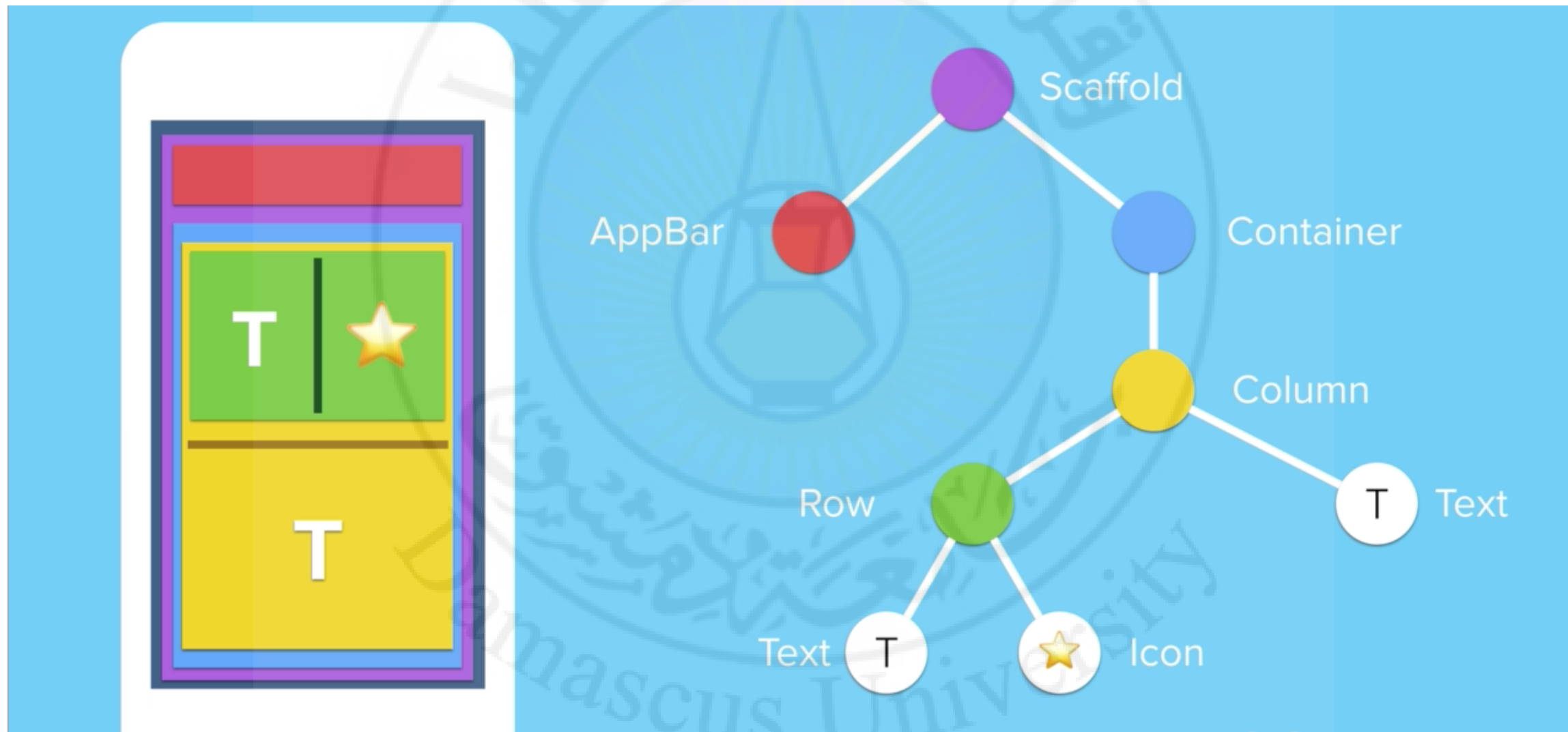
What are Widgets in Flutter?

- Everything in flutter consist of Widgets including but not limited to, visible Screen(s),
- text(s),
Button(s),
Material Design(s),
Application Bar(s)
as well as invisible Container(s) and Layout(s)

Everything is a Widget



Everything is a Widget



Stateful Widget vs. Stateless Widget



A single StatelessWidget can build in many different BuildContexts

A StatefulWidget creates a new State object for each BuildContext

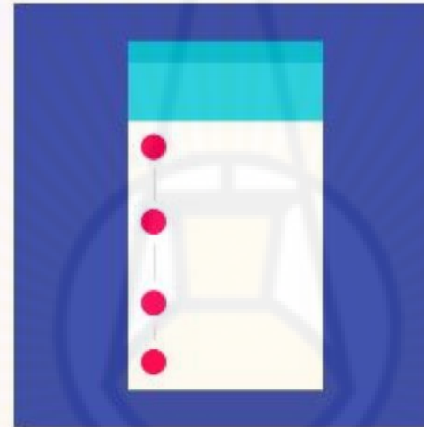
Layout & Components



ListTile

A single fixed-height row that typically contains some text as well as a leading or trailing icon.

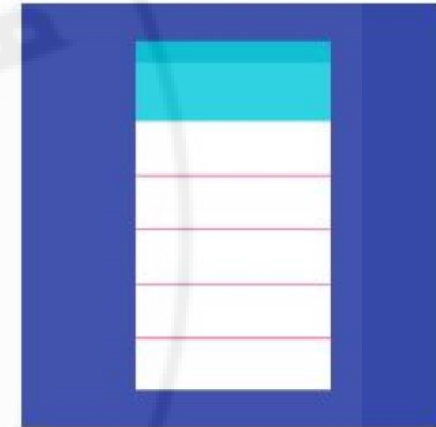
[Documentation](#)



Stepper

A material stepper widget that displays progress through a sequence of steps.

[Documentation](#)



Divider

A one logical pixel thick horizontal line, with padding on either side.

[Documentation](#)

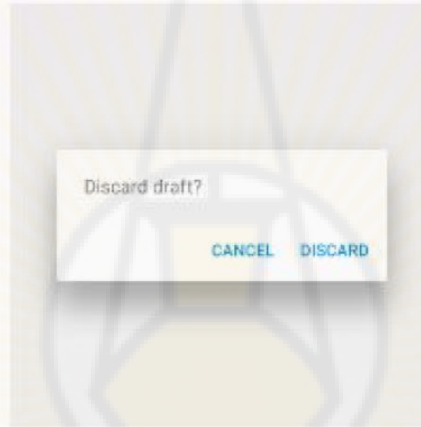
Layout & Components



SimpleDialog

Simple dialogs can provide additional details or actions about a list item. For example they can display avatars icons clarifying subtext or orthogonal actions...

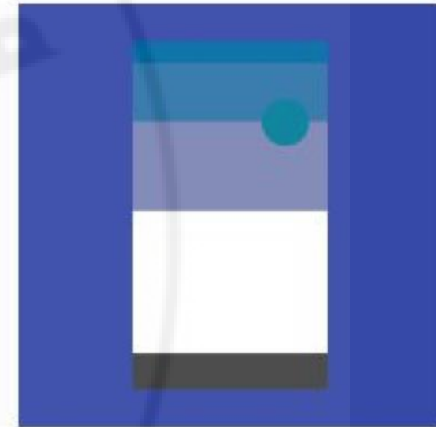
[Documentation](#)



AlertDialog

Alerts are urgent interruptions requiring acknowledgement that inform the user about a situation. The AlertDialog widget implements this component.

[Documentation](#)



BottomSheet

Bottom sheets slide up from the bottom of the screen to reveal more content. You can call `showBottomSheet()` to implement a persistent bottom sheet or...

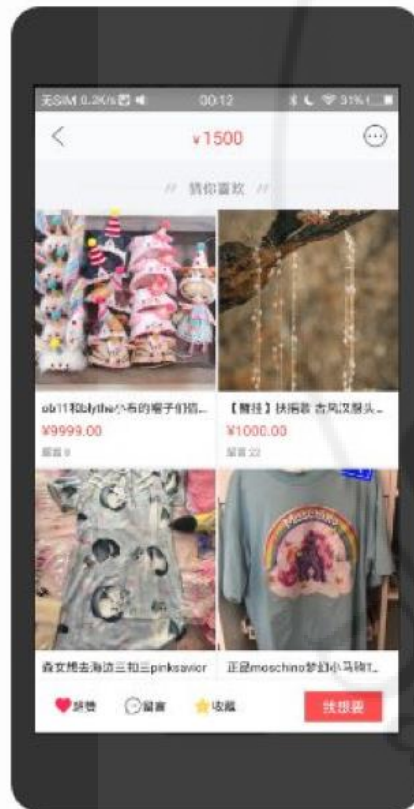
[Documentation](#)

IDE Support ?

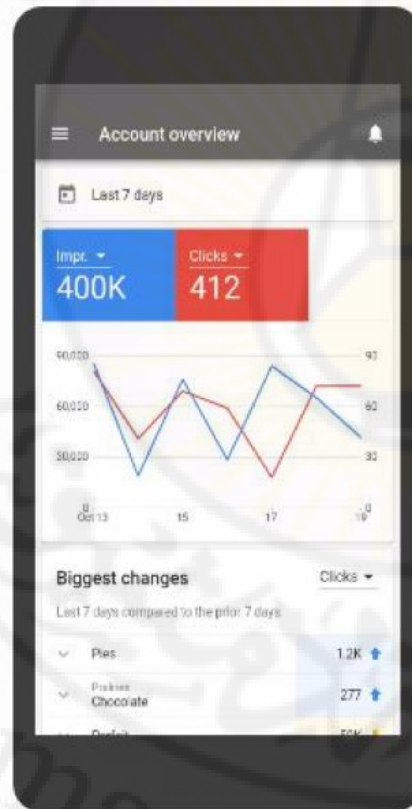


Flutter Showcase

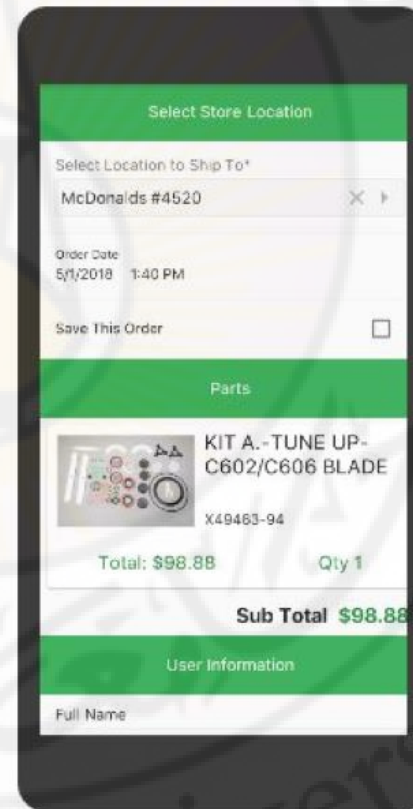
Alibaba



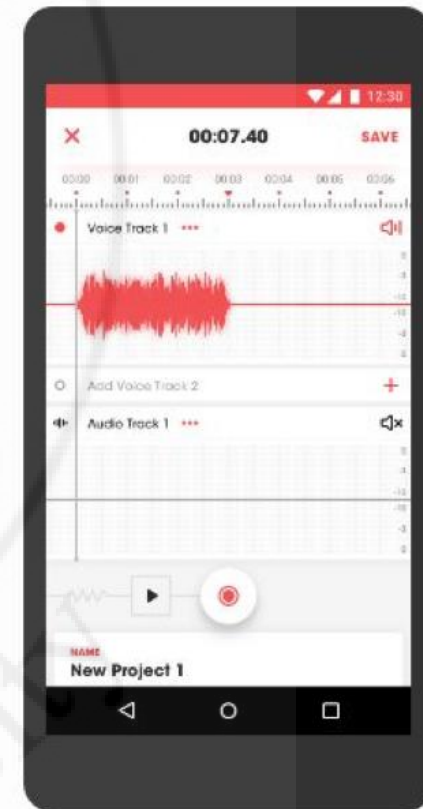
Google Ads



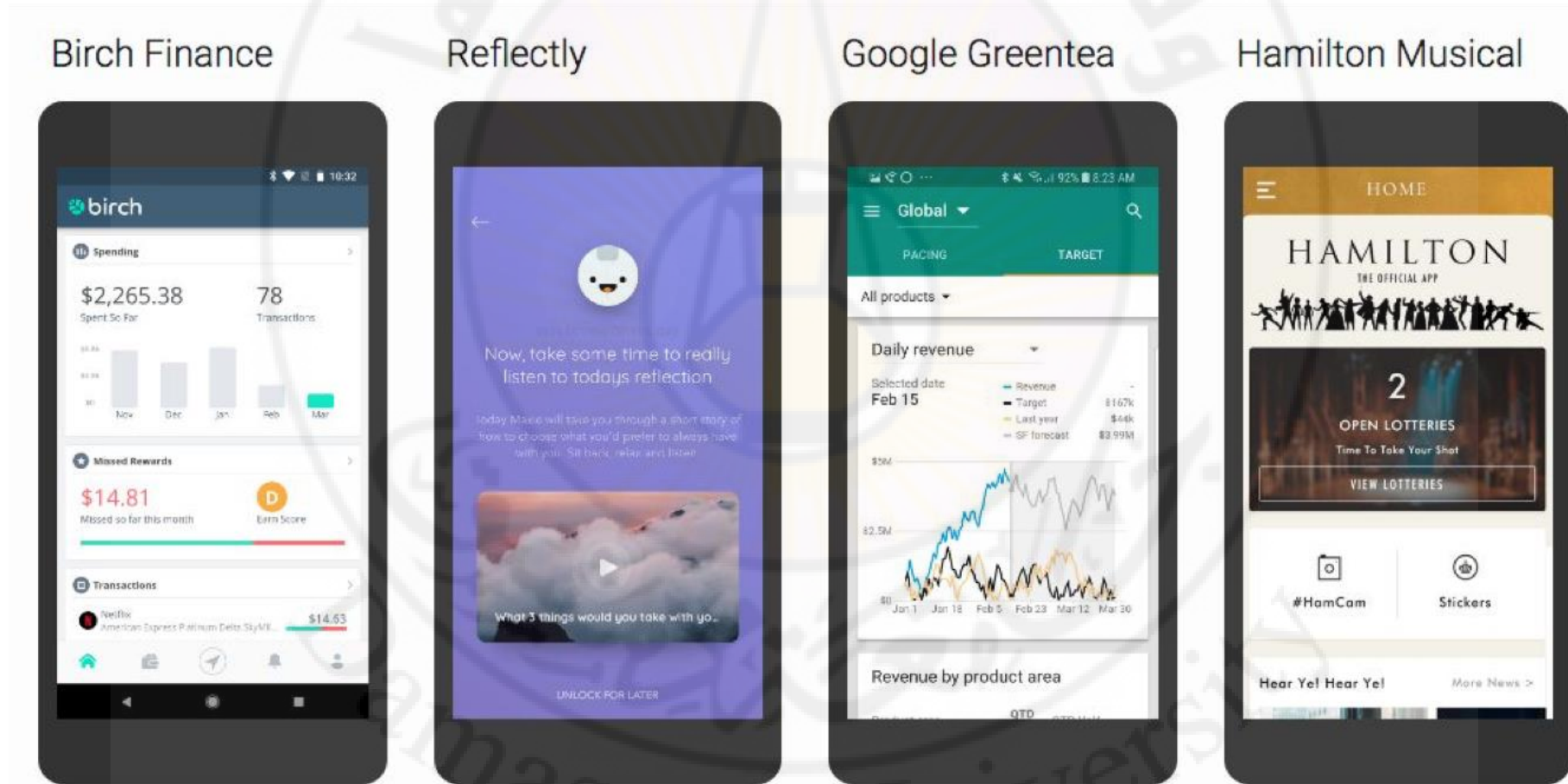
AppTree



Topline



Flutter Showcase





جامعة دمشق
الكلية التطبيقية
قسم تقنيات الحاسوب

البرمجيات النقالة

- ٢ -

Dart Language

- The Dart language, developed by Google, is a programming language that can be used to develop web, desktop, server-side, and mobile applications.
- Dart is the programming Language used to code Flutter apps, enabling it to provide the best experience to the developer for the creation of high-level mobile applications. So, let's explore what Dart provides and how it works so we can later apply what we learn in Flutter.

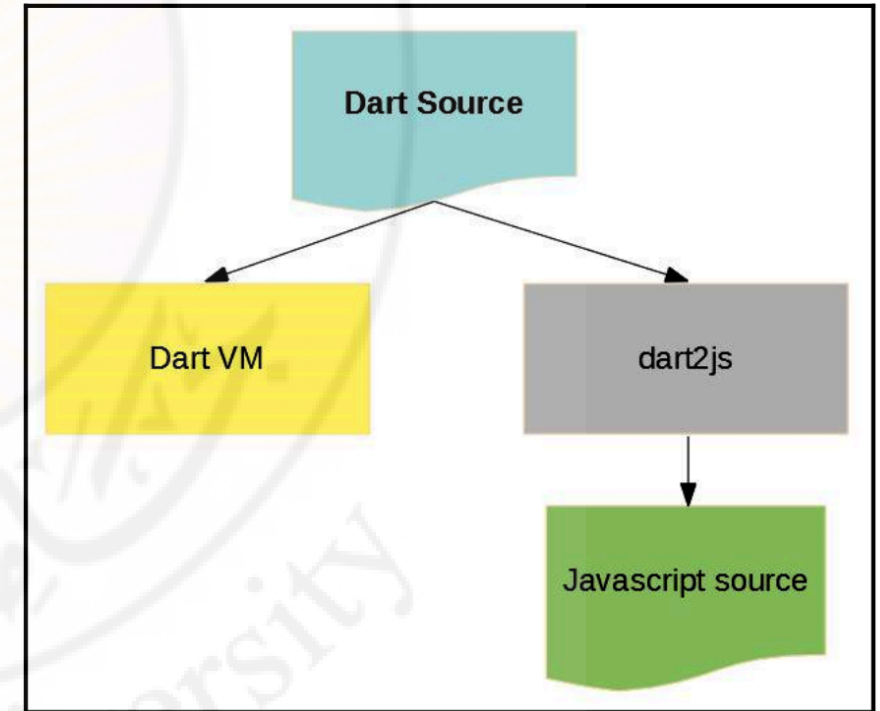
Dart Language

- Dart aims to aggregate the benefits of most of the high-level languages with mature language features, including the following:
 - Productive tooling: This includes tools to analyze code, integrated development environment (IDE) plugins, and big package ecosystems.
 - Garbage collection: This manages or deals with memory deallocation (mainly memory occupied by objects that are no longer in use).
 - Type annotations (optional): This is for those who want security and consistency to control all of the data in an application.
 - Statically typed: Although type annotations are optional, Dart is type-safe and uses type inference to analyze types in runtime. This feature is important for finding bugs during compile time.
 - Portability: This is not only for the web (transpiled to JavaScript), but it can be natively compiled to ARM and x86 code.

How Dart works

- To understand where the language's flexibility came from, we need to know how we can run Dart code. This is done in two ways:
 - Dart Virtual Machines (VMs)
 - JavaScript compilations

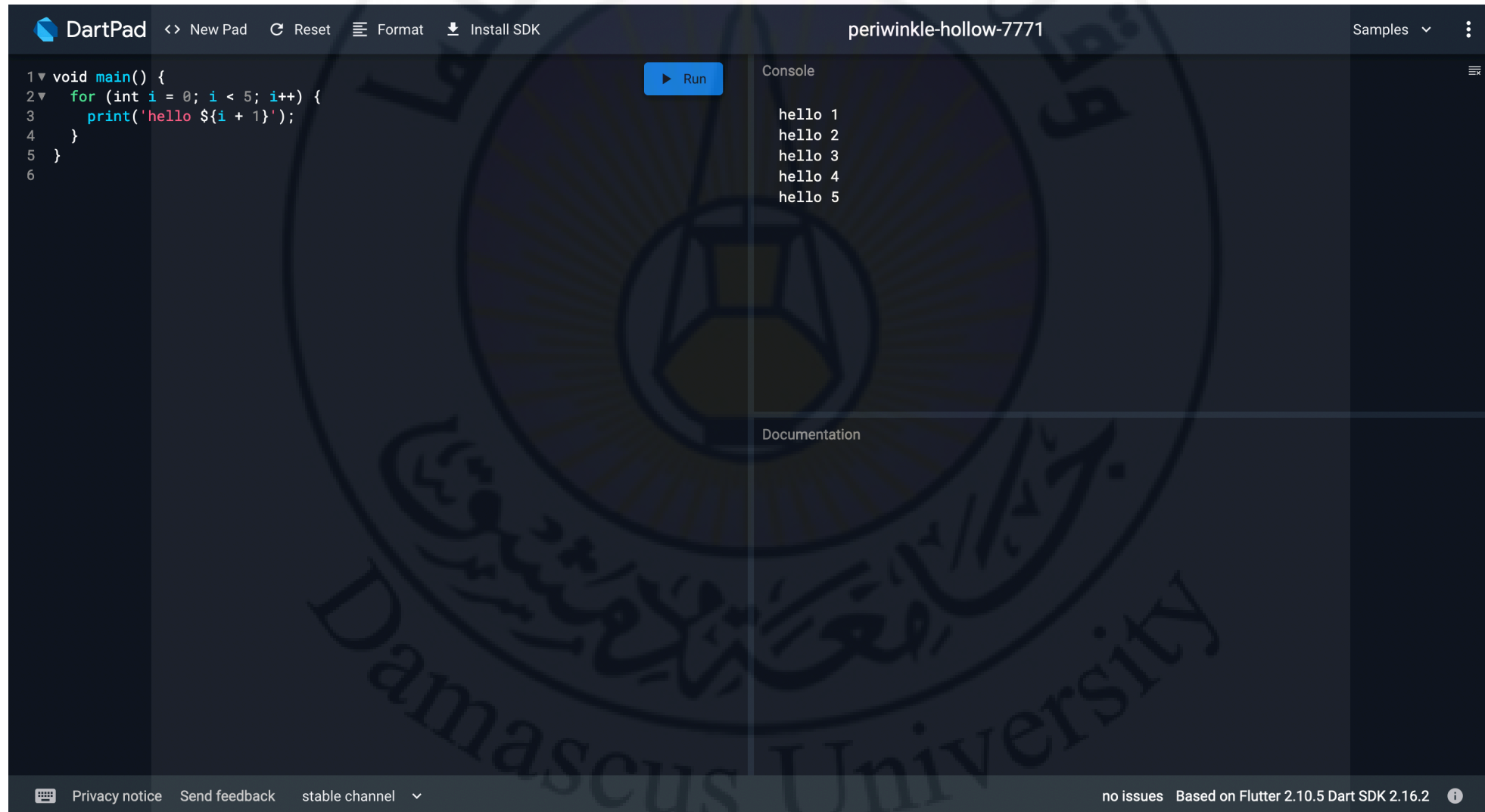
Have a look at the following diagram:



Dart development tools

- DartPad is a perfect way to start experimenting with the language without any extra effort.
Since you will soon want to learn advanced things such as writing on files or using custom libraries, you'll need to have a development environment configured for that.
- The most common IDEs used for Dart and Flutter development are Visual Studio Code or VS Code (for the web and Flutter) and Android Studio or any JetBrains IDE such as WebStorm (which is web-focused). All of the Dart functionalities of these IDEs are based on official tools, so it doesn't matter what you choose—the provided tools will be mostly the same. The Dart SDK provides specialized tools for each development ecosystem, such as web and server-side programming.

DartPad



Hello world

- The following code is a basic Dart script, so let's take a look:

```
main() { // the entrypoint of an Dart app
    var a = 'world'; // declaring and initializing variable
    print('hello $a'); // call function to print to display output
}
```

- This code contains some basic language features that need highlighting:
 - Every Dart app must have an entry point top-level function), that is, the main() function.
 - Although Dart is type-safe, type annotations are optional. Here, we declare a variable with no type and assign a String literal to it.

Hello world

- The following code is a basic Dart script, so let's take a look:

```
main() { // the entrypoint of an Dart app
    var a = 'world'; // declaring and initializing variable
    print('hello $a'); // call function to print to display output
}
```

- This code contains some basic language features that need highlighting:
 - A String literal can be surrounded with single or double quotes, for example, 'hello world' or "hello world".
 - To display output on the console, you can use the print() function (which is another top-level function).
 - With the string interpolation technique, the \$a statement inside a String literal resolves the value of the a variable. Dart calls the object's toString() method

Dart operators

- In Dart, operators are nothing more than methods defined in classes with a special syntax.
- So, when you use operators such as `x == y`, it is as though you are invoking the `x.==(y)` method to compare equality
- This concept means that operators can be overridden so that you can write your own logic for them. Again, if you have some experience in Java, C#, JavaScript, or similar languages, you can skip most of the operators, as they are very similar in several languages

Dart operators

- Dart has the following operators:
 - Arithmetic
 - Increment and decrement
 - Equality and relational
 - Type checking and casting
 - Logical operators
 - Bits manipulation
 - Null-safe and null-aware (modern programming languages provide this operator to facilitate null value handling)

Arithmetic operators

- Dart comes with many typical operators that work like many languages; this includes the following:
 - `+`: This is for the addition of numbers.
 - `-`: This is for subtraction.
 - `*`: This is for multiplication.
 - `/`: This is for division.
 - `~/`: This is for integer division. In Dart, any simple division with `/` results in a double value. To get only the integer part, you would need to make some kind of transformation (that is, type cast) in other programming languages; however, here, the integer division operator does this task.
 - `%`: This is for modulo operations (the remainder of integer division).
 - `-expression`: This is for negation (which reverses the sign of expression).

Increment and decrement operators

- The increment and decrement operators are also common operators and are implemented in number type, as follows:
 - ++var or var++ to increment 1 into var
 - --var or var-- to decrement 1 from var
- The Dart increment and decrement operators don't have anything different to typical languages. A good application of increment and decrement operators is for count operations on loops

Equality and relational operators

- The equality Dart operators are as follows:
 - `==`: For checking whether operands are equal
 - `!=`: For checking whether operands are different
- For relational tests, the operators are as follows:
 - `>`: For checking whether the left operand is greater than the right one
 - `<`: For checking whether the left operand is less than the right one
 - `>=`: For checking whether the left operand is greater than or equal to the right one
 - `<=`: For checking whether the left operand is less than or equal to the right one

Type checking and casting

- Dart has optional typing, as you already know, so type checking operators may be handy for checking types at runtime:
 - `is`: For checking whether the operand has the tested type
 - `is!`: For checking whether the operand does not have the tested type
- There's also the `as` keyword, which is used for typecasting from a supertype to a subtype, such as converting `num` into `int`.

Logical operators

- Logical operators in Dart are the common operators applied to bool operands; they can be variables, expressions, or conditions. Additionally, they can be combined with complex expressions by combining the results of the expressions. The provided logical operators are as follows:
 - !expression: To negate the result of an expression, that is, true to false and false to true
 - ||: To apply logical OR between two expressions
 - &&: To apply logical AND between two expressions

Bits manipulation

- Dart provides bitwise and shift operators to manipulate individual bits of numbers, usually with the num type. They are as follows:
 - &: To apply logical AND to operands, checking whether the corresponding bits are both 1
 - |: To apply logical OR to operands, checking whether at least one of the corresponding bits is 1
 - ^: To apply logical XOR to operands, checking whether only one but not both of the corresponding bits is 1
 - ~operand: To invert the bits of the operand, such as 1s becoming 0s and 0s becoming 1s
 - <<: To shift the left operand in x bits to the left (this shifts 0s from the right)
 - >>: To shift the left operand in x bits to the right (discarding the bits from the left)
- Like arithmetic operators, the bitwise ones also have shortcut assignment operators, and they work in the exact same way as the previously presented ones; they are <<=, >>=, &=, ^=, and |=.

Null-safe and null-aware operators

- Following the trend on modern OOP languages, Dart provides a null-safe syntax that evaluates and returns an expression according to its null/non-null value.
- The evaluation works in the following way: if expression1 is non-null, it returns its value; otherwise, it evaluates and returns the value of expression2: `expression1 ?? expression2`.
- In addition to the common assignment operator, `=`, and the ones listed in the corresponding operators, Dart also provides a combination between the assignment and the null-aware expression; that is, the `??=` operator, which assigns a value to a variable only if its current value is null.
- Dart also provides a null-aware access operator, `?.`, which prevents accessing null object members.

Dart types and variables

- You probably already know how to declare a simple variable, that is, by using the var keyword followed by the name. One thing to note is that when we did not specify the variable's initial value, it assumed null no matter its type.

final and const

- A variable will never intend to change its value after it is assigned, and you can use the final and const ways for declaring this:
 - `final value = 1;`
- The value variable cannot be changed once it's initialized:
 - `const value = 1;`
- Just like the final keyword, the value variable cannot be changed once it's initialized, and its initialization must occur together with a declaration.
- In addition to this, the const keyword defines a compile-time constant. As a compile-time constant, the const values are known at compile time. They also can be used to make object instances or Lists immutable, as follows:
 - `const list = const [1, 2, 3]`
// and
`const point = const Point(1,2)`
- This will set the value of both variables during compile time, turning them into completely immutable variables.

Built-in types

- Dart is a type-safe programming language, so types are mandatory for variables. Although types are mandatory, type annotations are optional, which means that you don't need to specify the type of a variable when declaring it. Dart performs type inference, and we will examine more of this in the *Type inference – bringing dynamism to the show* section.
- Here are the built-in data types in Dart:
 - Numbers (such as num, int, and double)
 - Booleans (such as bool)
 - Collections (such as lists, arrays, and maps)
 - Strings and runes (for expressing Unicode characters in a string)

Built-in types

Type	Literal example
int	10, 1, -1, 5, and 0
double	10.1, 1.2, 3.123, and -1.2
bool	true and false
String	"Dart", 'Dash', and <code>"""multiline String"""</code>
List	[1, 2, 3] and ["one", "two", "three"]
Map	{"key1": "val1", "b": 2}

Type inference – bringing dynamism to the show

- Var keyword

```
import 'dart:mirrors';

main() {
  var someInt = 1;
  print(reflect(someInt).type.reflectedType.toString()); // prints: int
}
```

Type inference – bringing dynamism to the show

- Dynamic keyword

```
main() {  
    var a; // here we didn't initialized var so its  
           // type is the special dynamic  
    a = 1; // now a is a int  
    a = "a"; // and now a String  
  
    print(a is int); // prints false  
    print(a is String); // prints true  
    print(a is dynamic); // prints true  
    print(a.runtimeType); // prints String  
}
```


Control flows and looping

- We've reviewed how to use Dart variables and operators to create conditional expressions. To work with variables and operators, we typically need to implement some control flow to make our Dart code take the appropriate direction in our logic.
- Dart provides some control flow syntax that is very similar to other programming languages; it is as follows:
 - if-else
 - switch/case
 - Looping with for, while, and do-while break and continue
 - Asserts
 - Exceptions with try/catch and throw



جامعة دمشق
الكلية التطبيقية
قسم تقنيات الحاسوب

البرمجيات النقالة

- ٣ -

Introduction to OOP in Dart

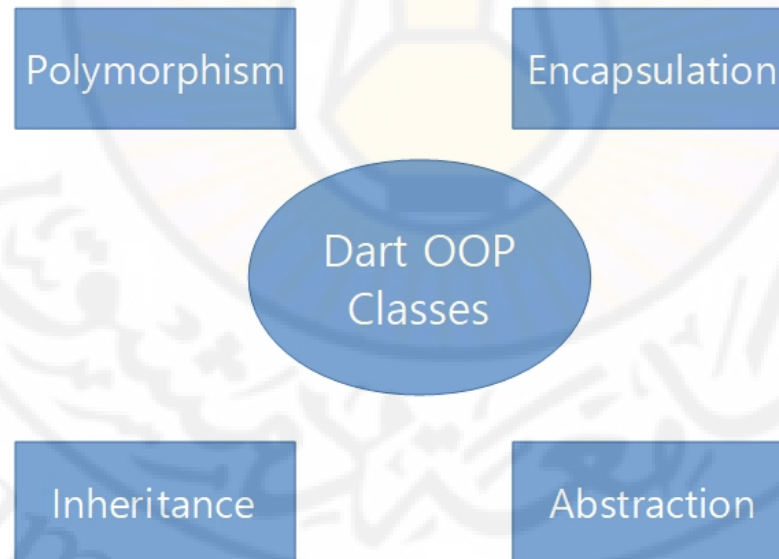
- In Dart, everything is an object, including the built-in types. Upon defining a new class, even when you don't extend anything, it will be a *descendant of an object*. Dart implicitly does this for you.
- Dart is called a **true object-oriented** language. Even functions are objects, which means that you can do the following:
 - Assign a function as a value of a variable.
 - Pass it as an argument to another function.
 - Return it as a result of a function as you would do with any other type, such as String and int.
- This is known as having **first-class functions** because they're treated the same way as other types.

Introduction to OOP in Dart

- Another important point to note is that Dart supports *single inheritance* on a class, similar to Java and most other languages, which means that a class can inherit directly from only a single class at a time.
- Here are the main OOP artifacts that are presented in the Dart language:
 - **Class:** This is a blueprint for creating an object.
 - **Interface:** This is a contract definition with a set of methods available on an object. Although there is no explicit interface type in Dart, we can achieve the interface purpose with abstract classes.
 - **Enumerated class:** This is a special kind of class that defines a set of common constant values.
 - **Mixin:** This is a way of reusing a class's code in multiple class hierarchies.

Introduction to OOP in Dart

- Every programming language can provide the OOP paradigm in its own way, with partial or full support, by applying some or all of the following principles:



Objects and classes

- The starting point of OOP, objects, are instances of defined classes. In Dart, as has already been pointed out, everything is an object, that is, every value we can store in a variable is an instance of a class. Besides that, all objects also extend the Object class, directly or indirectly:
 - Dart classes can have both instance members (methods and fields) and class members (static methods and fields).
 - Dart classes do not support constructor overloading, but you can use the flexible function argument specifications from the language (optional, positional, and named) to provide different ways to instantiate a class. Also, you can have named constructors to define alternatives.

Inheritance and composition

- Inheritance allows us to extend an object to specialized versions of some abstract type. In Dart, by simply declaring a class, we are already extending the Object type implicitly. The following also applies:
 - Dart permits single direct inheritance.
 - Dart has special support for mixins, which can be used to extend class functionalities without direct inheritance, simulating multiple inheritances, and reusing code.
 - Dart does not contain a final class directive like other languages, that is, a class can always be extended (have children).

Polymorphism

- **Polymorphism** is achieved by inheritance and can be regarded as the ability of an object to behave like another; for example, the `int` type is also a `num` type. The following also applies:
 - Dart allows overriding parent methods to change their original behavior.
 - Dart does not allow **overloading** in the way you may be familiar with. You cannot define the same method twice with different arguments. You can simulate overloading by using flexible argument definitions (that is, optional and positional, as seen in the previous *Functions* section) or not use it at all.

Encapsulation

- Dart does not contain access restrictions explicitly, like the famous keywords used in Java—protected, private, and public. In Dart, encapsulation occurs at the library level instead of at the class level (this will be discussed further in the following chapter). The following also applies:
 - Dart creates implicit getters and setters for all fields in a class, so you can define how data is accessible to consumers and the way it changes.
 - In Dart, if an identifier (class, class member, top-level function, or variable) starts with an underscore(`_`), it's private to its library.

Dart classes and constructors

- Dart classes are declared by using the class keyword, followed by the class name, ancestor classes, and implemented interfaces. Then, the class body is enclosed by a pair of curly braces, where you can add class members, that include the following:
 - **Fields:** These are variables used to define the data an object can hold.
 - **Accessors:** Getters and setters, as the name suggests, are used to access the fields of a class, where get is used to retrieve a value, and the set accessor is used to modify the corresponding value.
 - **Constructor:** This is the creator method of a class where the object instance fields are initialized.
 - **Methods:** The behavior of an object is defined by the actions it can take. These are the object functions.

Dart classes and constructors

```
class Person {  
    String firstName = "";  
    String lastName = "";  
  
    String getFullName() => "$firstName $lastName";  
}  
  
main() {  
    Person somePerson = new Person();  
    somePerson.firstName = "Clark";  
    somePerson.lastName = "Kent";  
    print(somePerson.getFullName()); // prints Clark Kent  
}
```

Dart classes and constructors

- Now, let's take a look at the Person class declared in the preceding code and make some observations:
 - To instantiate a class, we use the new (*optional*) keyword followed by the constructor invocation. As we advance in this book, you will notice that this keyword is used less.
 - It does not have an ancestor class explicitly declared, but it does have one, the object type, as already mentioned, and this inheritance happens implicitly in Dart.
 - It has two fields, firstName and lastName, and a method, getFullName(), which concatenates both by using string interpolation and then returns.
 - It does not have any get or set accessor declared, so how did we access firstName and lastName to mutate it? A default get/set accessor is defined for every field in a class.
 - The *dot* class.member notation is used to access a class member, whatever it is—a method or a field (get/set).
 - We have not defined a constructor for the class, but, as you may be thinking, there's a default empty constructor (no arguments) already provided for us.

The cascade notation

- We've seen that Dart provides the dot notation to access a class member. In addition to that, we can also use the double dot/cascade notation, **syntactic sugar**, which allows us to chain a sequence of operations on the same object:

```
main() {  
    Person somePerson = new Person()  
        ..firstName = "Clark"  
        ..lastName = "Kent";  
  
    print(somePerson.getFullName()); // prints Clark Kent  
}
```

- The result is the same as when employing the typical approach. It's just a good way to write succinct and legible code.

The enum type

- The enum type is a common type used by most languages to represent a set of finite constant values. In Dart, it is no different. By using the enum keyword, followed by the constant values, you can define an enum type:
 - ```
enum PersonType {
 student, employee
}
```
- Note that you define just the value names. enum types are special types with a set of finite
- values that have an index property representing its value. Now, let's see how it works. First, we add a field to our previously defined Person class to store its type:
  - ```
class Person {  
    ...  
    PersonType type;  
    ...  
}
```


The enum type

- Then, we can use it just like any other field:

```
main() {  
    print(PersonType.values); // prints [PersonType.student,  
                                //PersonType.employee]  
  
    Person somePerson = new Person();  
    somePerson.type = PersonType.employee;  
    print(somePerson.type); // prints PersonType.employee  
    print(somePerson.type.index); // prints 1  
}
```

- You can see that the index property is zero, based on the declaration position of the value.
- Also, you can see that we are calling the values getter on the PersonType enum directly. This is a static member of the enum type that simply returns a list with all of its values. We will examine this further soon.

Constructors

- To instantiate a class, we use the new keyword, followed by the corresponding constructor with parameters, if required. Now, let's change the Person class and define a constructor with parameters on it:

Constructors

```
class Person {  
    String firstName = "";  
    String lastName = "";  
  
    Person(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
  
    String getFullName() => "$firstName $lastName";  
}  
  
main() {  
    // Person somePerson = new Person(); this would not compile as we  
    //defined mandatory parameters on constructor  
    Person somePerson = new Person("Clark", "Kent");  
    print(somePerson.getFullName());  
}
```

Constructors

- If you look in our constructor body, it uses the `this` keyword. Furthermore, the constructor parameter names are the same as the field ones, which could cause ambiguity. So, to avoid this, we prefix the object instance fields with the `this` keyword during the value assign step.
- Dart provides another way to write a constructor like the one provided in the example, by using a shortcut syntax:

```
// ... class fields definition

// shortcut initialization syntax
Person(this.firstName, this.lastName);
```

- We can omit the constructor body as it only sets the class field values without any additional setup applied to it.

Named constructors

- Unlike Java and many other languages, Dart does not have overloading by redefinition, so, to define alternative constructors for a class, you need to use the named constructors:

```
// ... class fields definition  
// other constructors  
  
Person.anonymous() {}
```

- A named constructor is how you define alternative constructors for a class. In the preceding example, we defined an alternative constructor for a Person class without a name.

Factory constructors

- Another useful syntax in Dart is the factory constructor, which helps to apply the factory pattern, a creation technique that allows classes to be instantiated without specifying the exact resulting object type.

Field accessors – getters and setters

- As mentioned previously, getters and setters allow us to access a field on a class, and every field has these accessors, even when we do not define them. In the preceding Person example, when we execute `somePerson.firstName = "Peter"`, we are calling the `firstName` field's set accessor and sending "Peter" as a parameter to it. Also in the example, the get accessor is used when we call the `getFullName()` method on the person, and it concatenates both names.

Field accessors – getters and setters

```
class Person {
    String firstName = "";
    String lastName = "";

    Person(this.firstName, this.lastName);

    Person.anonymous() {}

    String get fullName => "$firstName $lastName";
    String get initials => "${firstName[0]}. ${lastName[0]}.";
}

main() {
    Person somePerson = new Person("clark", "kent");

    print(somePerson.fullName); // prints clark kent
    print(somePerson.initials); // prints c. k.

    somePerson.fullName = "peter parker";
    // we have not defined a setter fullName so it doesn't compile
}
```

Field accessors – getters and setters

- The following important observations can be made regarding the preceding example:
 - We could not have defined a getter or setter with the same field names: `firstName` and `lastName`. This would give us a compile error, as the class member names cannot be repeated.
The initials getter would throw an error for a person instantiated by
 - the anonymous named constructor, as it would not have `firstName` and `lastName` values (equates to null).
 - We do not need to always define the pair, get and set, together, as you can see that we have only defined a `fullName` getter and not a setter, so we cannot modify `fullName`. (This results in a compilation error, as indicated previously.)

Static fields and methods

- As you already know, fields are nothing more than variables that hold object values, and methods are simple functions that represent object actions. In some cases, you may want to share a value or method between all of the object instances of a class. For this use case, you can add the static modifier to them

- ```
class Person {
 // ... class fields definition
 static String personLabel = "Person name:";
 static String personCount = 0;

 String get fullName => "$personLabel $firstName $lastName";
 // modified to print the new static field "personLabel"
}
```

# Class inheritance

- In addition to the implicit inheritance to the Object type, Dart allows us to extend defined classes by using the extends keyword, where all of the members of the parent class are inherited, except the constructors.
- Now, let's check out the following example, where we create a child class for the existent Person class:



# Class inheritance

```
class Student extends Person {
 String nickName;

 Student(String firstName, String lastName, this.nickName)
 : super(firstName, lastName);

 @override
 String toString() => "$fullName, also known as $nickName";
}

main() {
 Student student = new Student("Clark", "Kent", "Kal-El");
 print(student); // same as calling student.toString()
 // prints Clark Kent, also known as Kal-El
}
```

# Class inheritance

- The following observations can be made regarding the preceding example:
  - Student: The Student class defines its own constructor. However, it calls the Person class constructor, passing the required parameters. This is done with the super keyword.
  - @override: There's an overridden toString() method on the Student class. This is where inheritance makes sense—we change the behavior of a parent class (Object, in this case) on the child class.
  - print(student): As you can see in the print(student) statement, we are not calling any method; the toString() method is called for us implicitly.