**Dart**

Last Update: October 2021.

• Data Types: •

Int

String

Double

Bools

<return type> Function(<…argument types>)

Dynamic: holds any type and has all the methods of primitive types.

Object: holds any type but doesn’t have any methods.

Int and double are subclasses of Nums.

Generic Types:

List

Map

Dart:typed\_data has many other types such as Uint64List and Float32List.

Use stringbuffer() and store characters in it using its methods. String is created when .toString() is called and the code is very performant using this approach.

Enum <name> {}

Is the same as in c++. Use the .index() method to get index of a stored enum.

• Variable types: final,var, const, late , required and dynamic.

Late is used when we want to use final but want to tell the compiler that we will initialize the variable before using it.

Var infers the type. If a variable is initialized when declared using var, it will only be of the inferred type and can’t change but if it is not initialized then it can hold any type and that will be changed whenever a new type’s value is assigned to the variable.

Final means it can be initialized later but only once.

Const means the variable cannot be changed and hence needs to be initialized whilst being declared. Const and final work together too. In classes, const is prefixed by static since there is no point in every instance having a const variable when the class itself can hold it. If all the attributes are final, const can be prefixed to the constructor which can give a minor performance boost.

Unlike final, every value being assigned to a const variable must be const. Even items in a list. That means if a class’ object is being stored, the object should be const as well. Meaning if a class has both a const and a non const constructor, there can’t be a const object for it if its initialized without the const constructor.

Required became a variable type in 2.10 , it was an annotation of @required before this update. Just used to mean that the variable must be initialized if this function is called and can only be used if the variable is inside the optional parameters.

• nnbd: non nullable by default, on since dart 2.10. Means nullable types have to be explicitly declared and normal ones can’t hold null value.

Int x;

Is hence an error.

Int? y; ? nullability operator has to be used to say y can be null.

Wherever y is to be used with an operator , like [] or … . use y?[] , y?.method() and so on.

• Type test operators: as, is, is!. Does what it says, as converts to the required type, is checks and is! Checks and inverts the bool.

• ?? is the null check operator. If the value of the variable before it is null return the next value instead.   
Int a;   
int b=a ?? 0; works and returns 0.

•RawString: Same as in any language, r””.

For example: r”\uaasd” is a raw string.

• Smart cast: if we check a variable with is in an if block, the variable is automatically converted to the other type.

For ex:

If(a is cow)

{ <a as cow> is automatically done.}

• for-in: for x in y loop, same as in python or c++.

• assert: check stuff at compile time. Asserts are ignored in release builds.

Assert(<expr>,<message>);

• In dart we can use one line functions, < func sig>() => <stuff>; stuff expressions’s value is automatically returned.

• ()=> <stuff>;or (){}; is declaration of anonymous function. After the {} adding () before; executes the anonymous function thereon. Idk about one line function’s execution without assignment.

• <iterable>.forEach((value)=> <stuf>); passes each value to any function(s). Recommended to not use anonymous functions. Type of value is inferred automatically but we can add it.

• \_ is used to say, I don’t want to use this value. It can be used but it’s symbolic for humans.

• Optional Parameter types for functions: These type of paramaters should only be declared at the end of argument list. We can only use one type of optional parameter type.  
[]: positional , meaning they are optional but they have to be passed in the same position as of their declaration. Empty middle positional parameters are not allowed.

So for,

Func s(int? a,[int? b, int? c]){}

S(1,,3) is not allowed but s(1,3) is and s(1) is .

{}: named optional parameter, meaning these parameters should be named whilst being passed in call.

For ex:

Func s(int? a,{int? b,int c=0}){}

S(2,c:3) .

• Dart allows functions within functions.

• Dart allows passing methods by reference and then automatically passes values to them.

• typedef: Same as in c++.

For functions:

Typedef lf=<return type> Function(<params>);

Works and the lf can by used to initialize variables to functions of the type.

Another feature added to typedef with 2.13 is type aliases.

Syntax:

Typedef <type> = <new name>;

We can use them with generic types too,

Typedef lol<x>=List<x>;

• .. is called the Cascade operator. It is used to access the properties of the class or object being assigned to another.  
For example,  
class A

{

Int d;

Int c;

A(this.a,this.b);

}

Can be initialized as  
A a=A(2,3)

..c=5;

The cascade operator accesses the class’ properties right there so it doesn’t need an initialized object in order to access it.

Useful in classes,like

Class B

{

A a=A(2,3);

a.c=5; will not work since a is not initialized to the compiler, but

A b=A(2,3)

..c=5  
..d=6; will work since here the anonymous object is getting initialized with the given changes

}

However as it may be apparent,   
a.c=3   
..d=4; will not work since 3 doesn’t have a property ‘d’. So in this case we can cascade ‘a’ entirely ,  
a..c=3  
..d=4; will work.

• For nullable types use, ?..

• import ‘<library location or name>’; to import a library.  
Libraries that come with the dart sdk use dart:<name> .  
Libraries that come with flutter sdk use package:flutter/<location and name>   
Libraries that are manually defined in pubspec use package:<location and name>

Inside the quotes of import.

• use library <name>; to name our library .

• use import ‘<name>’ as <another name>; to import libraries as a different name.  
• use import ‘<name>’ show <class name>; to only import the selected class from the library, we can use other attributes too , so long as they are on the global scope of the library.

• use import’<name>’ hide <class name>; to hide the selected classes/global attributes.

• In dart , we hide attributes or classes by prefixing their names with ‘\_’ at time of declaration. This way they are private. Even constructors can be made private, we use factory constructors in that case. They are private only to objects outside the file, for every object inside the same file this makes no difference.

• Constructor bodies are ran after the initialization phase, so if a variable is final it has to be initialized in the constructor’s declaration itself using <classname>(this.<attr1>,…); or using initializer lists, also called initializing formal.

• Initializer list: <constructor declaration>: <func calls or attr assignments>;

For example:

Class A

{

Int a;

A(int b): a=b; works. This is not the same as putting a=b inside the body as the initializer list is run in in initialization phase whereas the constructor body is ran after it.

}

• Since dart doesn’t have method overloading we use, named constructors. Like

<classname>.<name>(<…args>){};

• We can redirect constructors, much like delegating constructors in c++.

class A

{

int a;

int b;

A(this.a,this.b);

A.zero():this(0,0);

A.one(int d): this(1,d);

A.two(int c): this.one(c);

} is valid. This is called constructor redirection and it works only between same class constructors.

• Factory Constructor Redirection: Unlike normal constructor redirection, factory constructor redirection allows us to call external subclass constructors using factory constructors.

For example:

abstract class A{

A();

factory A.something(String name)=B;

}

class B extends A{

final String name;

B.\_(this.name);

factory B(String name){

return B.\_(name);

}

}

void main()

{

final a=A.something("al");

print(a.runtimeType);

}

• factory: used to return an instance of the class, which may or may not be a new instance if we decide so. Factory methods are on the class level instead of instance so they can’t access other non-static members.

For example:  
class A

{

Int a;

A.\_(this.a);

Factory A(int b)

{

Return A.\_(b); returns an instance of A with ‘a’ equals ‘b’;

}

factory A.yo(A a)

{

a..a=2

..b=3;

return a; this is where the power of factory constructors lie, modify the state of an instance and return it.

}

}

Since the default constructor here is private and unnamed it cant be accessed by objects from outside the file it is declared in.   
  
It wasn’t an issue in this case but factory A disables the default constructor. So we declared a named one, we can’t declare both a normal constructor and a normal factory method. But we can declare named constructors and named factory methods.   
Factory A.yo(<args>); hence works.

• When we extend, implement or with on a class, we can annotate if we override any methods by mentioning @override (it’s an empty class so it’s just used to say this).

• Annotations can be used anywhere, like @A.\_(2,3) works (no ; can be put after this, only after the next line, also the constructor of A being called must be const). This has no use in my opinion.

• Getters and Setters: Just like in c# declaring one or the other only provides that access to the variable, so only getter means read only and only setter means write only for the variable.

Getter: <type of variable> get <xyz name> => <variable to return> . We can use {} too.

Setter: set <xyz name>(<argument>){do whatever}. Doesn’t return anything, takes only 1 argument and then we can do whatever stuff we want inside. Usually used to assign variables some value based on some checks.

Though it’s just better to use final variable instead of a getter and use a setter only if there is some check.

• operator overloading: <classname> operator<operator to overload>(<obj to overload against>){};

Can’t overload same operator twice.

• call() overloading: Overload the () operator with , <return type> call(<…args>){};

Same, can be overloaded only once.

• cloning: By default, user-defined objects are passed by reference. So we create copy constructors just like in c++.

For ex.:

Class A

{

Int a;

A(this.a);

A copyWith({int newVal})

{

Return A(newVal ?? this.a);

}  
}

A a=A(2);

A b=a.copyWith(); works and returns a new copy instead of referencing the same obj.

• Inheritance: Dart doesn’t support multiple inheritance. Use ‘extend’ keyword to inherit a class. Default constructor or parameterized constructor must exist for a class to be inherited.

Class A

{}

Class B extends A{}

• All methods are virtual by default in dart.

• Upcasting is allowed, downcasting is not (since 2.9).

A obj =A();

B obj2=A(); not allowed

A obj3=B(); allowed.

In upcasting, overridden and extra attributes are sliced off.

• Use super to call parent class methods and attributes. If the parent class has a parameterized constructor, use initializer list and call super(<…args>) at its end.

• abstract class: We can use abstract class <name>{} to create an abstract class, meant to be extended. This type of class must have one method without a body. Can’t be instantiated, can have static members which can be directly accessed without instantiating. Unlike normal classes which must not have factory constructors in place of default/parameterized constructors , abstract classes can have default factory constructors though they can’t return the instance of the abstract class.

We can use , factory A()= B ; called factory constructor redirection.

• Abstract classes can have no non-body methods, but non-body methods must be inside abstract classes.

• covariant: allows a subtype to be used to instantiate an object.

For example:

Abstract class fruit

{}

Abstract class mammal

{

Void eat (fruit f);

}

Class banane extend fruit

{}

Class monke extends mammal

{

@override

Void eat(banane b){} ; gives error since eat needs Fruit. However we know that banane is inherited from fruit so it shouldn’t be an error. Just use, covariant banane b, to turn off static analysis for this variable. Alternatively , covariant fruit f, in the abstract class would do the same thing but allow any class to override with any subtypes.

}

• Interface: Use ‘implements’ to implement a class, this means that the class implementing another class must override every method of it. Normal classes can be implemented by other classes as well, so long as they have a single empty body method. (Empty body means {} , non-body would be no {} at all). A single class can implement multiple classes.

• Mixin: A class without a constructor that provides functionality. Mixins can have only non-body methods but they must be overridden by the class that uses them. Use mixin <name> on <classname>{} to make a mixin only be inherited by the given class.  
For example:

mixin Dance

{

void eat(String a);

}

class B

{

void speak()

{

print("ya");

}

}

abstract class C

{

void learn();

}

class A extends B with Dance implements C

{

void learn()

{

print("learn");

}

void eat(String a)

{

print("eat");

}

}

• When declaring a class, we put: class <classname> <extends> <mixins> <implements>{} in this order.

• extension: Used to add functionality to an already defined class. extension <name> on <classname>{}

For ex.

extension DoSome on String

{

Bool booboo()

{

Return this.isEmpty; this is the reference to the instance of String.

}

}

String a=”a”;

Print(a.booboo()); works and prints false.

• extension methods cannot be used on dynamic objects. Will compile but give runtime error. This also applies to the ‘var’ if the ‘var’ doesn’t have a value at the time of it’s declaration.

• Every immutable class (all attributes final) should override class Object’s methods, which are toString, bool operator==(<class obj>) , int get hashcode => <value>.hashCode . Hashes the value for the object and then passes it along to other methods like hashtable and hashmap.

• identical(<obj1>,<obj2>) , compares 2 objects and returns true if they have the same reference, i.e point to the same memory .

• class <x> implements Comparabale<x>{} , adds int compareTo(<object>) to be overridden. This method is used in sorting.

• to compare 2 doubles, use if((<first>-<second>).abs()< 1e-9) {} . This negates most rounding errors.

• throw randomclass() . Any object can be thrown. Even throw “na” works. Only null can’t be thrown.   
For example,  
try

{

Throw “s”;

}

On String catch (exc)

{

Print(exc)

}

Catch (e)

{…

}

Works, we have to specify the object’s class in the on and then we can deal with it. Like int for integers and <classname> for <classname> <object> . This is where overridden toString() is useful, since we can directly print the exception.

• <obj>.runtimeType returns the name of the class of the object.

• The try catch has, try{}on <classname> catch(<error obj>){}, catch(<error obj>) {} and finally{}.

• Alternatively, for user defined classes we can use implement the abstract class Exception and return the toString() to return a custom error.

Like,

Class Xerror implements Exception

{

Final String message;

Const Xerror(YClass obj) : message=obj.value;

@override

String toString() => message;

}

Now just throw Xerror(Yclass(2)) and it will work.

• Use rethrow to re throw an exception that has been caught, using throw resets stacktrace so the previous throw’s stack is discarded and that’s not efficient.

• Templates in dart are same as in cpp but without template declaration. Called Generics.

For ex:

Class Pop<T>{

Final T obj;

Const Pop(this.obj);

}

Works.

Pop p=Pop<int>(2);

• If a type is not assigned, the default type inferred is dynamic.

• If a class uses generics, then the subclasses must atleast declare that number of generics.

• In a case when a class uses generics, the subclasses should have the same name for the generics. For ex:

class A<T>

{

T obj;

A(this.obj);

}

class B<T> extends A<T>

{

B(T b):super(b);

}

Here, T can’t be anything else.

• Inheritance, abstract classes and mixins work as is with only <T> added (<>called diamond list). The number of arguments should be same and should be named the same.   
• To limit T to specific type, use T extends <classname> and then it can only be a subclass of the given class or the class itself. Num works.

For ex:

Class A<T extends String>{} means T can only be of type string.

• Generics are allowed in methods of non-generic classes.

Generic Collections

• List<T> : list of type T.   
final list 1=[0,1];

Final list 2=[2, …list1]; works, the spread operator opens the list . use …? For nullable types.

• Collection Statements:

We can perform checks or even loops inside lists.

Final list1=[

1,

2,

If(alphaIsTrue) 11

For(var i=3;i<10;++i) i

]; works and now for will put in each iterated item in the list. The if line works similarly, if we had used {}, we would have needed to use ‘,’ after the blocks.

Final list3=const<int>[0,1,2] works. But const needs constant objects, so if classes are used then they should have constant objects created using constant constructors.

• These are the types of List.

Growable: List<int>();

Fixed: List<int>(<size>);

Filled: List<int>.filled(<size>,<values>,growable:<bool>);

Unmodifiable: List<int>.unmodifiable(<iterable>); unmodifiablelistview doesn’t copy each element.

Generate: List<int>.generate(<size>,<generator>);

• Set<T>: No duplicates and sorted by default.

For ex: Set<int> a={1,2,3,4};

{} is map , <T>{} is set .

Unmodifiable doesn’t create a copy by default. Rest is same with list.

• Map<K,V>: Just like dictionary in python or map in c++.

Map<int,bool> a={1:true,2:false};

Unmodifiable and unmodifiablemapview exist.

Map.putifabsent(<key>, <value or function that returns value>);

Map[<key>] =<value> sets the value at key.

3 types of maps

LinkedHashMap<K, V>: default, the values at keys are returned in the order they are inserted.

HashMap<K, V>: null is a valid key, the order is not saved so random keys are returned.

SplayTreeMap<K, V>: keys are sorted and the pairs are saved in a self-balancing BST. Comparator must be passed for user defined type of keys or the class must implement/extend comparator class.

• Equatable is a package that provides prebuilt implementation of hashcode and == for immutable classes.

• Intermediates: These functions are called on containers in the middle of the call stack.  
These are:  
where(), returns an iterable satisfying the function provided to it.

Map(), this map is a function that runs a function on each element and then returns an iterable with the processed values. For ex: list<int> list1=[2,3,4] ; final list2=list1.map((value)=>value\*2)); will return 4,6,8 in the list2.

Skip(), skips given number of values and returns an iterable.

followedBy(), concats another iterable to an iterable and returns the final iterable.

• Terminals: These functions are called on containers at their tails.

These are:

toList(),toSet(),toString(),

every(), returns a Boolean if every element of the iterable satisfies the function.

Contains(),

Reduce((<elem1>,<elem2>)=>stuff), reduces entire container to a single value of the type of container.

Fold(<initial value>, <same as reduce>) , folds the container but uses the initial value in the function as well, also it can return a value of a type different than the container.

So ultimately, <container>.<intermediates>.<terminals>

• Future<T>: Asynchronously executes a task. Future<T>.value(stuff) asynchronously executes the stuff and returns either the final value or error. Future<T> obj=stuff, obj.then() executes function inside then after stuff has been processed.

.catchError(), will catch any error and pass it.

• obj.wait([]).then(…).catchError((e)=>…); waits for every future in the list to be completed.

• Since normal functions can’t have await, we use obj.then(<stuff>);

• Future<T>.delayed(<duration>,<function which returns value of type T>)

• Future<T>.error() completes the future and returns an error.

•Future<T>.value() creates a future object of a non-future object, though it returns the value immediately.

• Future<T>.sync() executes the function inside sync immediately.

• Async and await: Functions can be marked async and that allows them to return a Future<T>. We can use await <function call> to asynchronously execute the function. It is the same as then but less code. Unlike then however (where the rest of the code will continue and when then is finished it will do the stuff), the function is told to wrap it up at await. And everything after it executes in sync.

• sleep(<Duration>) sleeps the thread for the set duration.

• Streams and Generators: A generator is a function that returns a value and then continues execution. Stream<T> is returned by async generators and Iterable<T> for sync gennys. They are declared by async\* or sync\*respectively and use yield to return a value. Every stream emits a ‘done’ event after it completes, even single event streams.  
  
A stream is like a pipe connecting a subscriber and a generator. A genny sends data bit by bit and the subscriber consumes it, there can be multiple subscribers to a single genny but not vice versa.

For ex:

Stream<int> str() async\*

{

For(int i=0;i<10;++i)

{

Await Future<void>.delayed(Duration(seconds: 2));

Yield random.nextInt(10\*i);

Yield random.nextInt(20\*i);  
}

}

Yield doesn’t stop the function, so here the genny returns 2 values consecutively and then waits for 2 seconds and goes on. So basically, this function runs for 20 seconds with 2 values being streamed every 2 seconds.

• yield\*: There’s also yield\*, yield\* runs another function. Yield\* doesn’t affect the function so after it completes, the control pointer goes to yield.

Iterable<int> func(int x)

If(x>0)

Yield\* func(x-1)

Yield x;

Here say x is passed as 3, 3 goes in , then yield\* runs func again with x as x-1 until x>0 where it yields 0, then yields 1 then 2 then 3. In 1 call it has returned 4 values.

•Subscriber: At the very basic we can listen to a stream like this,

Await for(var a in str()) {} and then use it wherever.

Though we usually use the controller to declare and use a stream,

StreamController<T> <name>(onListen,onCancel,onPause,onResume);

We add events to the streamcontroller using the .add(<T>) method. To get the stream object we call the .stream() method on the controller.  
For its subscriber we copy its stream object and use .listen() method on it. The object that holds the listen() callback on stream is called a subscriber and we can call .cancel() on it which invokes the onCancel callback in the controller.

For ex:

import 'dart:async';

class Lox {

  int val = 0;

  StreamController<int> streamController;

  Timer timer;

  Lox() {

    streamController = StreamController<int>(

        onListen: \_startStream,

        onCancel: \_stopStream,

        onPause: \_stopStream,

        onResume: \_startStream);

  }

  Stream<int> call() {

    return streamController.stream;

  }

  void \_startStream() {

    timer = Timer.periodic(

      Duration(seconds: 5),

      (bkTime) {

        val++;

        streamController.add(val);

      },

    );

  }

  void \_stopStream() {

    timer.cancel();

    streamController.close();

    print('Stopped');

  }

}

Future<void> doStuff() async {

  final lox = Lox();

  final stream = lox();

  final subscriber = stream.listen(

    (event) {

      print(event);

    },

  );

  await Future<void>.delayed(Duration(seconds: 30));

  await subscriber.cancel();

}

void main() {

  Future<void>(doStuff).then((\_) => print('done'));

}

• Timer: Declared in the dart:async package, this class executes something at set periods or only once after a set duration.

Timer(<Duration>,<callback>); will execute callback after duration.

Timer.periodic(<duration>,<callback(Timer obj)>); will execute callback every duration units later.

Timer obj.cancel(); will cancel the timer.

• Stream<T>.periodic(<duration object>,<genny or function that returns a value of type T>);

•Stream<T>.value(); emits a single value of type T .

•Stream<T>.error(); same as value but emits an error.

•Stream<T>.fromIterable(); creates a stream from an iterable and emits each value each time it is called.

•Stream<T>.fromFuture(); creates a single event emitter, when the future completes the value/error is emitted.

•Stream<T>.empty(); just sends a ‘done’ event.

•stream obj.drain(); discards all events and emits done or error.

• stream obj.map(); just like normal map performs a function on each event, used to convert Stream<T> to Stream<A>.

•stream obj.skip(); discards the given number of events from the stream.

• Isolates: Dart doesn’t have threading, instead there are isolates which much like threads execute stuff on other threads. However they have their own memory so no data races and other issues.

We use isolate.spawn() to start an isolate. Alternatively, we use compute<Q,R>(<func>,<…args>) where Q is the type of parameter needed by the func and R is the return type.

• Microtask: Part of flutter, we use scheduleMicrotask((){}) to schedule microtasks. They have to be synchronous and must be quick tasks as they have higher priority than normal events.

• Completer: A class that produces Future objects and completes with either a value or an error. Useful for callbacks.

For example:

Completer completer= Completer<int>();

For a Future<void> doStuff()

{

return completer.future;

}

When we call doStuff(), completer.future returns a Future<int> that isn’t completed just yet. We have to await that instead.

The use of completer here is that it allows a future to be processed later on. The completer.future is an empty future that awaits on the context’s futures to complete.

Anywhere if we mention completer.future.then((){}); the then callback is available in the completer.complete(); callback. Similarly onError works.

import 'dart:async';

final completer= Completer<int>();

Future<int> doStuff(int val) async{

await doX(val);

return completer.future;

}

Future<void> doX(int val) async{

await Future.delayed(Duration(seconds:5),(){

print("yoo $val");

});

}

void main() async

{

final futura= doStuff(2);

futura.then((val){

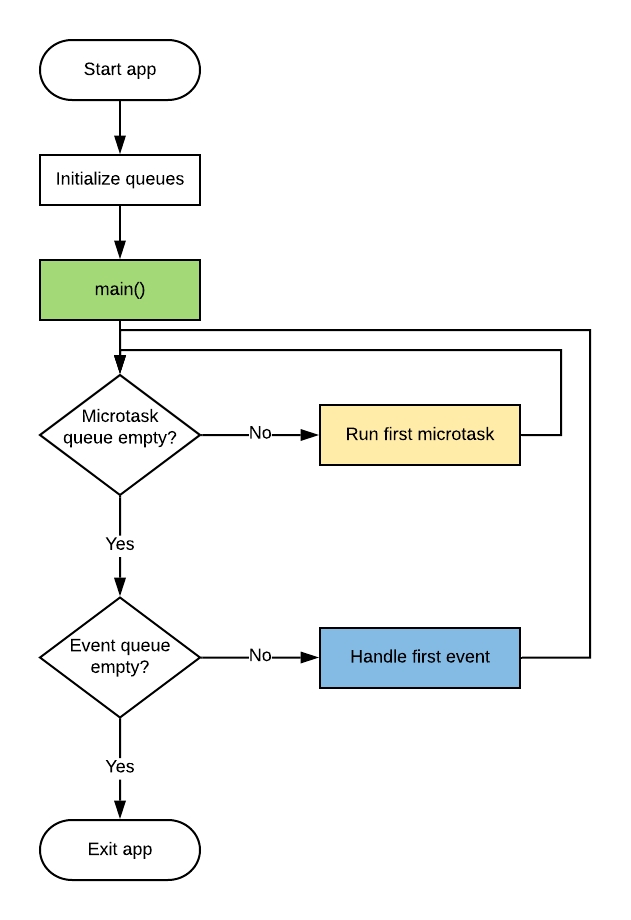
print("new val $val");

});

completer.complete(4);

}

Prints yoo 4\nnew val 4 after 5 seconds.   
Here, completer isn’t completed until doStuff() is complete.

• Dart Event Loop: 

• Design Principles:  
TDD: Test Driven development, write tests for all cases.  
DDD: Domain Driven design, identifier names should match the business domain.

• S.O.L.I.D:   
Single Responsibility Principle: Just like functions, classes should focus on one type of task so that they’re harder to break.

Open Closed Principle: New behaviors could be added without modifying the existing code.

Liskov Substitution Principle: If a subclass is made, it must override the superclass but maintain the logical correctness.

Interface Segregation Principle: Clients shouldn’t be forced to implement behaviors they don’t need. 8 interfaces with 1 method each are better than 1 interface with 8 methods.

Dependency Inversion Principle: Superclasses must not depend on children, both should depend on abstractions. Abstractions shouldn’t depend on children either.

• Dependency Injection: 2 classes shouldn’t be tightly coupled otherwise they will depend on each other meaning a single change in 1 class would necessitate change in the other and vice versa. Otherwise if a new method is introduced it would create dependency on the other class.

•Constructor Injection: We make our class be able to accept objects of acceptable classes. So we can inject constructors of any classes that are either the required class or a subclass of them. To do this we create an object of an abstract class and pass a class’s object to initialize it.

**•** <iterable>.Foreach(f): Does not return and function f is async then simply skips await calls until it finishes the parent block. Unlike how async works, as async functions finish up before the block is finished.

• <iterable>.map(f): Applies f to each element and returns a lazy iterable (Iterables that don’t finish up the processing of an item until each item is called). Unlike foreach it returns value at each element so await can be applied to each returned element.

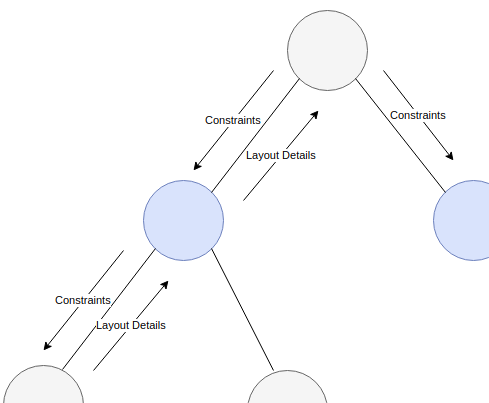
• Method injection: Same stuff but we have abstract class object in function instead of a class.

**Flutter**

• Flutter Rendering Phases: There are 4 phases through which our widgets are rendered in Flutter.

Layout Phase: In this phase flutter determines the dimensions of all objects, and where they will be displayed on the screen.

In Layout Phase, the parents pass contraints down to their children and receive layout details back. For example: A container passes max width to its child (constraints) and then the child only uses some part of the width and returns the details (layout details).



There are 2 types of layout protocols which define what the constraints and layout details are going to be. Box Protocol, is the simple layout protocol for width, height etc. needing widgets, it uses BoxConstraints for constraints. Sliver Protocol is the layout protocol for widgets that have scroll and contain scrolling information like scroll offset, overlap etc. They use SliverConstraints and return SliverGeometry.

Painting Phase: A canvas is provided to each widget to paint itself on it.

Once all widgets have returned layout details, the painting phase begins. In it, flutter passes a PaintingContext to all widgets which contains a Canvas, which is used by widgets to draw theirselves.

Compositing Phase: Flutter composits all the painted canvases and the final result screen is sent to the GPU for rendering it on screen.

Rasterizing Phase: In this final phase, the scene is displayed on the screen.

• Render Tree: There’s a widget tree which has all the widgets and a render tree which has RenderObjects. RenderObjects are created by RenderObjectWidget and added to the render tree.

RenderObject contains a constraints object, a parentData object which allows its parents to attach useful information on it, a performLayout object, a paint method etc. RenderObject is an abstract class and 2 of the classes that extend it are RenderBox and RenderSliver, which use the Box and Sliver protocol respectively.

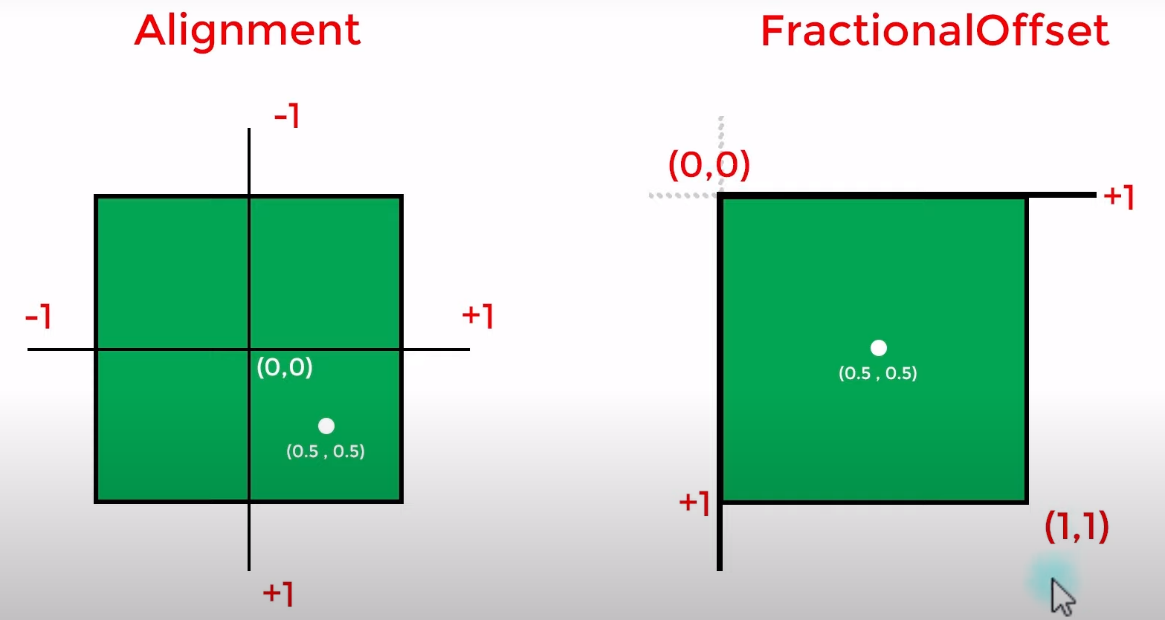
• Colors: Few ways to take colors,  
Hex: Hex is usually #<6hexdigits> but we use 0x<00 to ff for brightness><6 hex digits> in flutter.

Colors.<value>: Some preset colors.

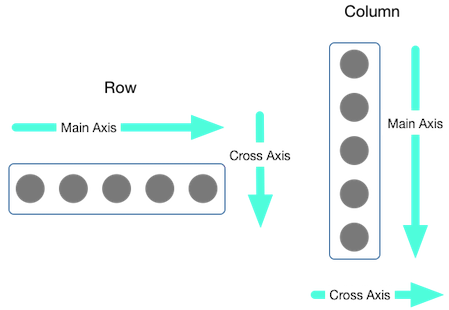
• SkSL Precompilation: The first time an app is ran, animations might be janky, known as shader compilation jank. To reduce this we can precompile shaders for our apps using SkSL warmup. To do that refer to <https://flutter.dev/docs/perf/rendering/shader>.

• Stack paints its children in order with the first child being at the bottom.

• Align: Use this widget to align widgets in various positions inside widgets. As given in the image below , Alignment(x,y) can be passed to align using the given co-ordinates. Alignment is a property available in widgets like container by default.  
For example, top left is Alignment(-1,-1).



MainAxis vs CrossAxis:



• Provider: A good way to manage state across the app.  
To create a provider there are many ways, for many types. They are:

1. Provider(  
   create: (context)=> <class>(),   
   child: <widget >(),  
   );

This is the most basic way of creating providers. It creates a provider of type <class> and then provides it’s value to every widget that calls for it. The class can be anything and doesn’t need any state logic.

1. ChangeNotifierProvider(

Create: (context) => <class that extends ChangeNotifier>(),

Child: <widget>(),

)

This is used to create a changenotifierprovider, the difference here is that a changenotifier extending class can choose when and for which functions it should call notifyListeners() function. This function notifies all the listeners of the change in the state of the class. A provider would notify listeners each time the state changes.

1. StreamProvider<T>(

Create: (context) => StreamObj,

initialData: TObj,

catchError: (context, Object error){},

child:<widget>,

)

Creates a provider that reads a stream object, i.e whenever a new value goes into the stream it notifies the listeners.

Like if T is int, then a stream that returns an int will pass an int object to provider as soon as it enters the stream. Since StreamProvider could have an initial value of null as a stream with a periodic duration starts with null, we can provide our own initial data. We can modify the values obtained in the stream by doing the following,

Create: (context)=> StreamObj.map((value) {do stuff with value then return it}), we can even throw errors which are caught and can be handled by the function given to the catchError parameter.

A good use of this provider is with firebase.onChanged.

1. FutureProvider<T>(…)

Same as streamprovider but for future, since future is returned only once this is equivalent to FutureBuilder but has additional parameters like initialvalue and catcherror.

1. ProxyProvider[optional int]<T,K,L…..>(  
   update:(context,TObj,KObj,LObj){

Return LObj;

}

initialValue: LObj,

create: (context)=> LObj,

child: <widget>,

)

Combines multiple provider values into 1 obj of the type that is at the end of the typelist inside diamond brackets.

The types before the last type should be type of providers. When a provider of the same type as any of them has an update in a state, the function passed to the update parameter is called which then notifies its listeners.

We can create an object with default values as well.

The optional int defines the proxyprovider’s count. There can be multiple proxy providers so they have to have an int signifying the same.

There is ChangeNotifierProxyProvider[optional int] and FutureProxyProvider[optional int] as well, the difference in them is that they the LObj in their case are of type ChangeNotifierProvider<LObj> and FutureProvider<LObj> respectively.

1. ValueNotifier<T>(TObj)

While it is not a provider, it’s still an advanced method for state management. A ValueNotifier object holds the value of type T and can be used to notify its listeners of a change in the value. Much like a normal provider. We can access the value in a valueNotifier object by the .value attribute. We can add a listener using the .addListener method.  
The ValueListenableBuilder(valueListenable: NotifierObj, builder(context, value, child) {}) is a widget that takes a valuenotifier and rebuilds its widgets whenever the valuenotifier is updated.

1. MultiProvider(providers:[…],child: <widget>);

Multiprovider is used to hold multiple provider widgets in it. Very useful with ProxyProviders since they need other providers anyway.

1. StateNotifierProvider<T,K>(  
   create: (context)=> TObj(),  
   child: <widget>,

);

A statenotifierprovider is just like a changenotifierprovider but the difference is that a statenotifier<K> extending class is needed. The said class will pass an initial value of type K to its base constructor. And in it where-ever we need to access the K value we can call state attribute. Whenever we cause a change to the state object in it, the listeners will be notified of the change.

1. XProvider<T>.value(

Value: TObj,

Child: <widget>,

)

Any provider can and should use this constructor instead if the object of type T is already initialized, i.e. we pass an instance of the object instead of initializing it here.

Listening to changes on providers:

1. Context.read<T>();   
   Returns the latest instance of the class once. We can then access the attributes wherever we want in the scope. T is the classname.
2. Provider.of<T>(context,listen:bool);

Returns either the latest instance or a Providerlistener , the former if listen is false which is true by default. We can then access the value anywhere by calling .value on the ProviderListener object.

1. Context.watch<T>();

Same as Provider.of<T>(context,listen:true);

1. Context.select<T,K>((TObj)=> TObj.KObj);

This is called a selector and only notifies of change in state if the value of type K changes. Since T can have multiple objects of K, we can specify which KObj it is in the T that will invoke the notifier.

1. Consumer<T>(

Child: <widget>,

Builder: (context, TObject, child){}

Consumer is a widget that rebuilds its children (which are returned by the builder) when the state changes. The child parameter can be null, it is used to give widgets that won’t require a change, so the consumer never re-builds them.

1. Selector<T,K>(

Selector: (context,TObj)=> TObj.KObj,

Child: <widget>,

Builder: (context, TObject,child){}

Same as context selector, except this one is a widget.

Modifying Provider states:

We can modify the state of a provider anywhere, given it is a listener.

That is, for object of Provider.of<T>(context,listen:true); , we can assign a value to obj.value.KObj =KObj; and it will notify listeners of the change. TObj in consumer and selectors work the same way.

• Multiple ways to get images, from network, file, cache, etc.:   
One such way to get image into a uint8list is:

final url =

        "<url>";

final networkImage = NetworkImage(url);

    await networkImage.obtainKey(ImageConfiguration()).then((value) async {

      var load = networkImage.load(value, (bytes,

          {allowUpscaling: true, cacheHeight, cacheWidth}) {

        log("size is ${bytes.lengthInBytes}");

        return ui.instantiateImageCodec(bytes);

      });

      load.addListener(ImageStreamListener((\_imageInfo, status) async {

        final imageData = await \_imageInfo.image.toByteData();

        final imageIntList = imageData!.buffer.asUint8List();

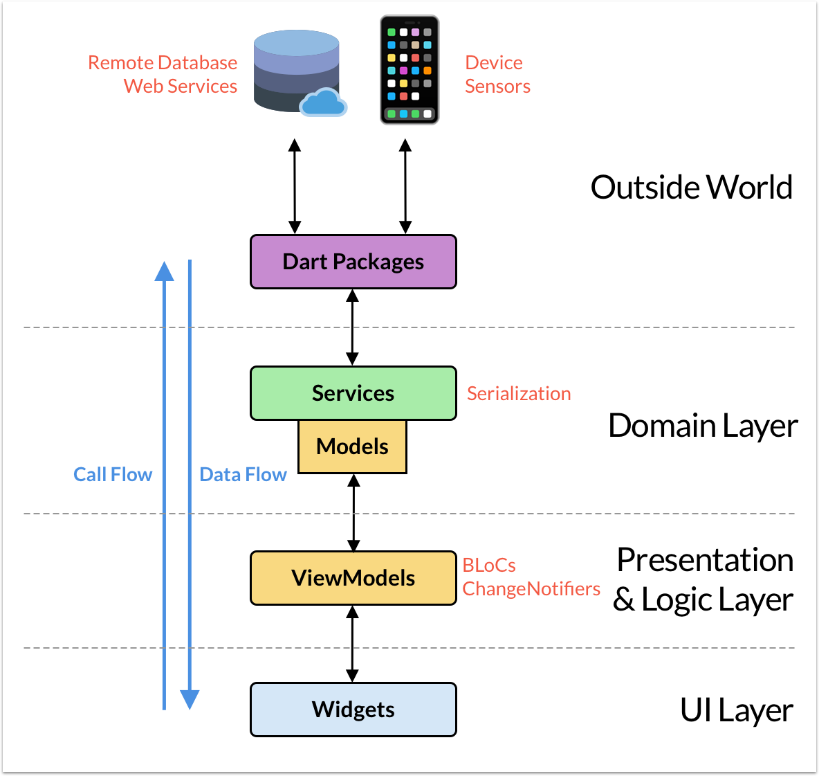
        log("well yea this is the image data too ${imageIntList.lengthInBytes}");

      }));

    });

• Architecture for flutter apps using Riverpod: An Architecture is used in conjunction with a design pattern and design patterns change for each app type.

An architecture by CodeWithAndrea:

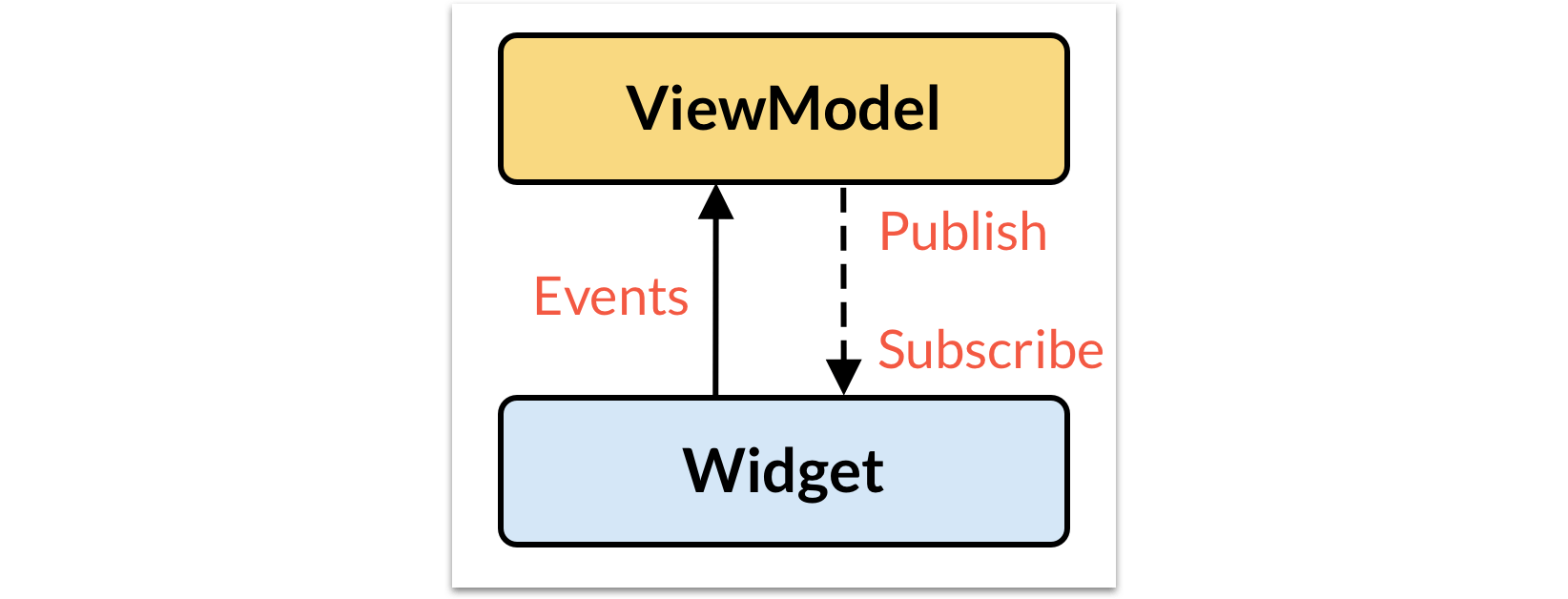


The data flow is unidirectional, meaning the components in the hierarchy do not know about components in layers below them.

Dart Packages: External dart and flutter packages.

Services and Models: Services are pure functional components written in pure dart and are used to transform external data into data usable by app using model classes, model classes are simple classes that are passed below and hold data. Service classes hence, return stongly-typed, immutable and domain-specific models.

ViewModels: ViewModels contain abstract classes for a widget’s state and presentation. Contain Business Logic for the app and contain mutable state classes. They don’t contain flutter code (like material.dart).



Widget: The UI of the app, widgets subscribe to ViewModel and ViewModels push changes to widgets, widgets then return events to ViewModels which changes state in the ViewModels and hence new changes get published, widgets rebuilt and voila!

• All API Widget Classes: Widget here is a class that provides some functionality to the app. Slightly different from a Widget that actually is inserted in a Widget tree.

W: Widget H: Helper class S: Service class B: Base/Abstract class

* Widget: A widget is an immutable description of a part of a UI, i.e., it defines what that part in the UI means, does and behaves like. A Widget isn’t what affects the rendering tree, that would be an Element which is itself made from a widget. Since widgets are immutable they don’t have a state, we give state using StatefulWidget. Every Widget that is inserted into a render tree transforms into an Element, and is given a Key. This Key determines what happens if another Widget comes to the same place as this widget. If the runtimeType and Key are same between a widget already in a tree and a widget replacing it then the Element gets updated with values (which is basically replacing the underlying Widget) but if the keys/type don’t match then the Element itself is removed and replaced.

B

* Element: An instantiation of a Widget for a particular location in a render tree. The widget for an element may change but the Element itself remains the same as the Elements form a tree.

B

* AbsorbPointer: If absorbing parameter is set to true, child widget of this widget will receive touch inputs.

W

* AlertDialog: Creates a dialog that has a title, content and action buttons. We use showDialog widget’s builder to show an alertdialog on the screen.

W

* Align: Used to align and size a child widget in parent widget. Alignment(x,y) and Alignment.<any of the 9 areas of a square> to align.

W

* Alignment: Alignment defines a point on the 2D plane, a point within a rectange with 0.0 as the center of the rectange and Alignment(0,0) being at the center.

S

* + AlignmentDirectional: A class just like Alignment but with an offset, it may take offset from left or right depending on the system’s text direction (ltr or rtl).

S

* AlignmentGeometry: Base class for Alignment.

B

* Accumulator: A simple class with an int attribute.

H

* Intent: In flutter Intent is an object that is bound to a keyboard key combination using Shortcuts widget and also to an Action, on the key combination this Action gets executed using ActionDispatcher. It’s an abstract class that only provides a constant constructor.

B

Class myIntent extends Intent{ const myIntent();}

* Action<T extends Intent>: Base class for actions, it sends a command whenever an intent is triggered.

B

Class MyAction extends Action<myIntent>{

Override add/removeActionListener and invoke method.

}

* ActionDispatcher: Invokes the actions and passes it the intent given to it.

H

ActionDispatcher().invokeAction(MyActionObj,myIntent())

* ActionListener: Allows us to listen to actions and intents passed to them, then automatically disposes them whenever necessary.

W

ActionListener(  
listener: (Action<Intent> action){

If(action.intentType==myIntent())

{

Dostuff

}

}  
);

Using ActionListener, ActionDispatcher, Action and Intent we can execute actions across the app in a vanilla android fashion.

* Actions: This widget uses ActionDispatcher and ActionListener internally and allows us to use Actions easily.

W

Actions(

actions: <Intent, Action<Intent>>{

<intent class>:<action class for it>}

child: <…stuff usually a builder>

);

Any child widget can call Actions.invoke(context,<intent class with new values>) and the mapped action runs its invoke method on the intent.

* ActivateIntent and ActivateAction: Used to activate the currently focused control which is usually Keyboard enter or space.
* AlignTransition: Animated widget for Align.

W

* AlwaysScrollableScrollPhysics: Always scrollable, even without content.

S

* AlwaysStoppedAnimation<T>: Animation that is always at a stop state and that state value is given at creation.

S

* AndroidView: Widget for embedding android view in a flutter app. Android view is just native view.

W

* AndroidViewSurface: Integrates a given android view with flutter’s touch handling.

H

* Animatable<T>: Returns a value of type T given an Animation<Double>.

S

* AnimatedAlign: Just like AlignTransition but a lot less verbose.

W

* AnimatedBuilder: Provides a builder that can apply animations to multiple widgets and manage their states.

W

* AnimatedContainer: Animated version of a container that allows any changes of the properties of container to start an animation.

W

* AnimatedCrossFade: Animates crossfading between 2 widgets, irrespective of their sizes.

W

* AnimatedCrossFadeBuilder
* AnimatedDefaultTextStyle: Animates changes between 2 text styles (except textAlign, softWrap, overflow, maxLines, textWidthBasis and textHeightBehavior properties).

W

* AnimatedList: Provides a list widget and animates insertions and deletions of items.

W

* AnimatedListItemBuilder
* AnimatedListRemovedItemBuilder
* AnimatedListState: Provides states for AnimatedList, we use this class to add or remove items from an animated list.

S

* AnimatedModalBarrier: A widget that obscures the route behind it, prevents touch inputs on it and animates the colors. Useful with ModalRoute (which is like alertdialog).

W

* AnimatedOpacity: Animates opacity of the given child widget, where it can go from fully invisible to fully visible.

W

* AnimatedPadding: Animates padding changes on a given child widget.

W

* AnimatedPhysicalModel: Animates changes on a physical model widget. A physical model is a widget that clips its children to a given shape.

W

* AnimatedPositioned: Animates changes as a positioned widget.

W

* AnimatedPositionedDirectional: Just like Animated Positioned but takes text direction into consideration and animates based on that.

W

* AnimatedSize: Animates between sizes.

W

* AnimatedSwitcher: A widget which does what AnimatedCrossfade does but provides more options and even allows different types of transitions in switching between widgets.

W

* AnimatedSwitcherLayoutBuilder
* AnimatedSwitcherTransitionBuilder
* AnimatedTransitionBuilder
* AnimatedWidget: Base class for building custom animation widgets like ScaleTransition/Fadetransition etc.

B

* AnimatedWidgetBaseState<T extends ImplicitlyAnimatedWidget>: This class calls build method of an ImplicitlyAnimatedWidget for each frame, we can use it to define custom behavior for builds.

H

* Animation<T>: Base class which consists of a value of type T and status which tells if the animation is going forward or backward. Tween is used to create various Animation subclasses.

B

* AnimationBehavior
* AnimationController: A controller for an animation, this controller controls how an animation plays out, setting the animation’s upper bound, lower bound and start value and some other properties. It needs a TickerProvider to be able to know when to go to the next state of the animation

S

* AnimationEagerListenerMixin
* AnimationLazyListenerMixin
* AnimationLocalListenersMixin
* AnimationLocalStatusListenersMixin
* AnimationMax/Min/Mean<T extends num>: These classes return the max, min or mean of type T from the given 2 animations of type T.

H

* AnimationStatus
* AnimationStatusListener
* AnimationWithParentMixin
* AnnotatedRegion <T extends Object>: Annotates a region of the layer tree with a value. Basically used to say this is the theme/style/etc. that should be the passed to the children to derive values from such as in themes.

H

* AppLifecycleState
* AppPrivateCommandCallback
* AspectRatio: A widget that sizes its childs according to a given ratio. For 16/9, it will say that the width can be 16x the smallest size and height will be the 9x the smallest size. The height is calculated by multiplying 16/9 to the max width of the child, where max width of the child is equal to ratio of max width to max height of the child.

W

* AssetBundle: A collection of resources such as images and strings that can be used by the app. We use this instead of rootbundle so that the platform can provide various types of AssetBundle handlers for various contexts.

S

* AssetBundleImageKey: Unique key used to identify an image in the image cache.

H

* AssetBundleImageProvider: Subclass of ImageProvider which has method for AssetBundles, useful to be used with AssetBundles.

S

* AssetImage: Fetches an image from AssetBundle given a context. Assets are chosen based on device pixel ratio and size, to directly get an image we use AssetBundleImageProvider. This class can automatically pick an asset based on device pixel ratio.

S and W

For example  
for a folder structure

Icons/1.0x/heart.png

Icons/2.0x/heart.png

The Pubspec will be

Flutter:

assets:

* Icons/heart.png

And AssetImage(‘icons/heart.png’) will be used. It will determine which asset to load based on device pixel density.

* AsyncSnapshot<T>: Contains the immutable representation/ result along with state of an async stream/future.

H

* AsyncWidgetBuilder
* AutocompleteFieldViewBuilder
* AutocompleteOnSelected
* AutocompleteOptionToString
* AutocompleteOptionsBuilder
* AutocompleteOptionsViewBuilder
* AutofillClient: This class provides autofill-related information of the given input field it represents.

S

* AutofillScope: A group of AutofillClients so that they can be connected and executed on the same textinput. If an AutofillClient doesn’t have a scope then it can only be executed on its own TextInputClient only.

S

* AutofillContextAction
* AutofillGroup: Widget for AutofillScope.

W

* AutofillGroupState: The state of an AutofillGroup widget. It can be used to register/de-register autofillClients on an AutofillGroup.

H

* AutofillHints: A collection of hints for values of the selected type such as AutofillHint.email taken from the platform itself.

S

* AutomaticKeepAlive: This widget allows its subtrees to be kept alive in the memory if the KeepAliveNotification is raised by any child widget. Requires children to use AutomaticKeepAliveClientMixin<T extends StatefulWidget> to raise and remove KeppAliveNotifications.

W

* AutomaticKeepAliveClientMixin
* AutomaticNotchedShape: Returns a NotchedShape given 2 ShapeBorders, this is also a shapeborder.

S

* AutovalidateMode
* Axis
* AxisDirection
* Route<T>: An abstract class used by a Navigator to build routes. The ‘routes’ used by a navigator are just pages or screens. The Route base class may provide visual elements which will be applied by Navigator to the routes. It may return a value, which will be of type T.

B

* RouteSettings: Configuration for a Route, this class is what is passed to a Route for a configuration.

S

* Page<T>: Implements a Route and provides configuration to it. Return type is T which is what a Route<T> would return. Used to create a route and also to be passed to Navigators.

B

* ModalRoute<T>: A route that blocks interaction with previous routes. Used to show modals, a modal is like an alertdialog but is a fully fledged page of itself.

W

* PageRoute<T>: A modal route that replaces the entire screen.

W

* MaterialPage<T>: Creates a material style Page, i.e., it can define properties like platform-adaptive transition whenever the route is removed from the current stack or pushed to the current stack.

W

* MaterialPageRoute<T>: For a material page route, i.e., a modal route with properties like transition.

W

* Navigator: A widget that manages multiple routes in a stack fashion. We push and pop routes from the stack, the one at the top of the stack is the current route. We can customize the route transitions using Navigator as well.

S

* Router<T>: A widget that facilitates routing of pages, i.e., it dispatches commands to close or open any page of an application. It uses Navigator internally and provides better functionality than using Navigator only. An app can have no router or many routers.

W

* RouterDelegate<T>: A delegate that is used by Router to build and configure a navigator. This class can also build a navigator for a Router.

S

* RouteInformation: Contains 2 props, a string path of a route and then an object which has state data for the route. If the Router owning a RouterInformation opts for state restoration then the object of the RouterInformation is serialized and stored and after app restart the config is deserialized and state is persisted.

S

* RouteInformationProvider: Provides RouteInformation to a router. It also notifies listeners of changes (usually the Router itself) to a particular RouteInformation.

S

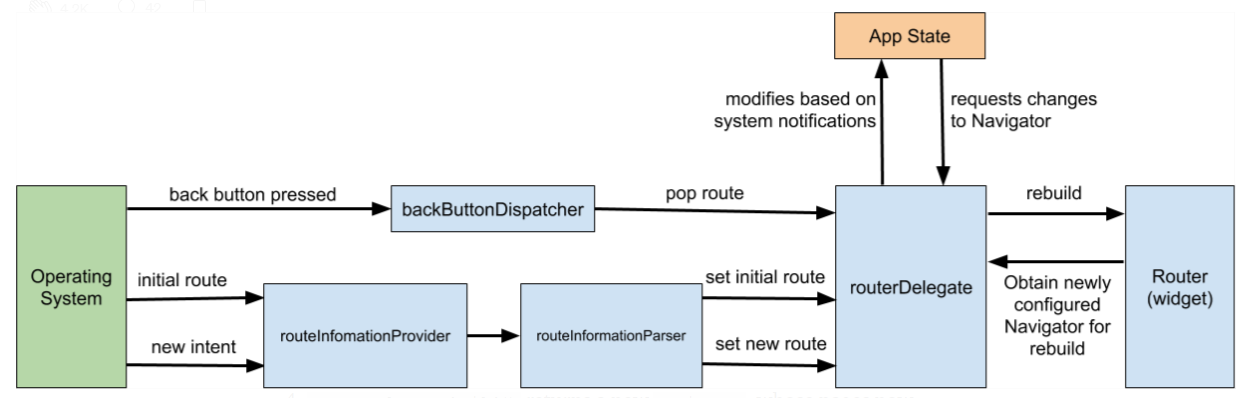
* BackButtonDispatcher: Reports back button events to a Router.

S

* ChildBackButtonDispatcher: When there are nested Routers, this class ensures that the BackButton event is passed for the current router and not the parent routers.

S

How Router works:



* RootBackButtonDispatcher: Default implementation of BackButtonDispatcher. It listens to platform pop route notifications and on event calls BackButtonDispatcher to handle the request.

S

* BackButtonListener: A widget that has a callback set for backbutton events which is fired when a RootBackButtonDispatcher sends a confirmation that the backbutton was pressed.

W

* BackdropFilter: Takes an imagefilter and applies it to child widgets.

W

* Simulation: A simulation class is used to simulate an object’s state for any given time given a position x and a velocity dx. The object contains state called isDone which will return true if the simulation is complete.

S

* ScrollActivity: Base class for scrolling activities like dragging and flinging.

B

* BallisticScrollActivity: Creates an activity that scrolls a view based on a simulation, the simulation is a retardation of velocity of scroll from the point of time when user leaves the screen and an initial velocity taken from the same.

S

* Banner: Shows a diagonal message on the corner of another widget.

W

* BannerLocation
* BannerPainter: Paints a banner.

S

* Baseline: A widget that sizes the box to contain its children and shifts the base of the children to be given pixels below the top edge of the widget above Baseline.

W

* BeveledRectangleBorder: A border that flattens the corners of the rectangle.

S

* BlendMode
* Semantics: A widget that annotates or provides a description to its children widgets. This meta-info is useful for accessibility tools and search engines to understand piece of UI.

W

* BlockSemantics: A Widget that prohibits the parent Semantics to be applied to its children.

W

* BlurStyle
* Border: A border of a box which has 4 sides. Each side is represented by a BorderSide object.

S

* BorderSide: A side of a border.

S

Border(top: BorderSide())

* BorderRadius: Used to provide radius for the corners of the border.

S

* BorderRadiusDirectional: Same as BorderRadius but if only 1 side of the Border has a BorderRadius then it will flip based on the textDirection.

S

* BorderRadiusGeometry: Base class for BorderRadius.

B

* Tween<T>: A linear interpolation class to interpolate between 2 values. This object can be plugged into an animation to let it animate each frame for each tick. T of tween can be of any type and many properties have a predefined tween class.

S

* BorderRadiusTween: Interpolation between 2 border radius.

S

* BorderDirectional: Just like Border except the left and right side may flip depending on textDirection.

S

* BorderStyle
* BorderTween: Interpolation between 2 borders.

S

* BottomNavigationBar: A widget displayed at the bottom with a few icons (3-5) which are typically used to show different screens on the same route.

W

* BottomNavigationBarItem: An interactive button used by a BottomNavigationBar to show items.

S

* BouncingScrollPhysics: Bouncing scroll physics which allows user to scroll outside bounds but bounces back in after user leaves touch.

S

* BouncingScrollSimulation: A simulation used for bouncing physics.

S

* BoxBorder: This is the class extended by Border and BorderDirectional to provide box borders that can be configured for widgets.

S

* Size: Holds a 2D floating-point size for an object, basically width, height and radius (if object is a circle ) from an origin at 0,0.

S

* Offset: A point in 2D space which is specified distance away from the origin.

S

* RenderObject: An object in the render tree. Render object is the most basic class for all rendering classes. We almost never need to create a class extending RenderObject directly but if we do we can define fine control properties like hit Testing, layout etc.

B

* RenderView: The root of a render tree, the view represents the total output surface area of the render tree and handles setting up the rendering pipeline. It has a unique child RenderBox which is what fills the output surface.

S

* RenderBox: An object in 2D co-ordinate system. The size is expressed as a width and height with 0,0 as the top left corner and max,max at the bottom right. Layout of the box widget is performed by a BoxConstraints object as its child which doesn’t exceed the layout set by the RenderBox’s parents.

B

* paintImage: Paints an image into the given rectangle on the canvas.

S

* paintBorder: Paints a border around the given rectangle.

S

* DecorationImage: An image for a BoxDecoration. Image is painted using paintImage.

S

* RenderImage: Creates a render box that displays an image. The image is painted using paintImage.

B

* RenderParagraph: A rendexbox that renders a paragraph of text.

B

* RenderProxyBox: A base class for render boxes that resemble their children, i.e., they mimic almost all the properties of their children.

B

* RenderOpacity: Makes its child partially transparent. The child will be of type RenderBox.

B

* RenderShiftedBox: Abstract class for one-child-layout render boxes that provide control over the child’s position.

B

* BoxConstraints: Immutable layout constraints for RenderBox.

S

* BoxConstraintsTransform
* BoxConstraintsTween: Interpolation between two boxconstraints.

S

* BoxDecoration: An object that defines the properties for painting a box.

S

* Decoration: Abstract class for all decoraations.

B

* BoxFit
* BoxPainter: Paints a particular Decoration object onto another class.

S

* Scrollable: Base class for a widget that scrolls, includes gesture recognition.

B

* Viewport: A widget that is bigger on the inside, i.e., what appears on the screen is always just a fraction of the complete widget. The center sliver holds the element which gets displayed.

W

* ScrollView: Creates a simple widget that scrolls, it internally uses Scrollable, Viewport and one or more slivers such as lists, grids etc.

W

* BoxScrollView: A scrollview that uses a single child layout model.

W

* BoxShadow: Defines a shadow cast by any box. Shape needn’t be rectangle.

S

* BoxShape
* BuildContext: Each widget has its own BuildContext, this object provides methods to modify the state for the widget. Even though each Widget has its own BuildContext, it cannot directly reference the parent BuildContext, only the ones that have already been returned on the tree,i.e., no widget in the build method can actually see the buildContext unless they have their own state and buildContext. To actually use generate a buildContext for the children we use Builders.

S

* BuildOwner: Manager class for widgets framework, i.e., it tracks states of widgets.

S

* Builder: This widget makes sure the parent widget is inserted into the tree and then passes the BuildContext to its builder method which the widgets can directly use, while being in the same build method.

W

* ButtonActivateIntent: Ab Intent that activates the currently focused button.

S

* CallbackAction<T exnteds Intent>: An Action which takes a callback to configure it instead of a diff subclass.

S

* PictureRecorder: Records a Picture containing a sequence of graphical operations for a canvas.

S

* Picture: An object that holds a sequence of graphical operations. We can use it on canvas to paint it.

S

* Scene: An opaque object that has a composited scene, which is basically a final Picture object given to a PictureRecorder in a Canvas.

S

* SceneBuilder: Builds a scene.

B

* Canvas: Canvas objects are used to create Picture for a PictureRecorder.

S

* Paint: A description of the style to use when drawing on canvas. We use Paint on canvas to describe style for the painter.

S

* InheritedWidget: Class that allows its children widgets to inherit data without explicitly passing the data thorugh constructors etc., just the context. To use it, first create a class that extends InheritedWidget, has const constructor that accepts Key and Widget then pass it to super then override bool updateShouldNotify(class old){…} and lastly has a static method class of(BuildContext context){

Final class? obj=context.dependOnInheritedWidgetOfExactType<class>(); assert(result!=null, ‘obj not found’);

Return obj;

}

Then insert this widget in the tree. And simply call class.of(context).<custom data property in the class> to get any stored entity.

B

* InheritedTheme: An Inherited Widget that is made specifically to work with visual properties (called themes). Allows us to capture a theme at any point of the widget tree and then wrap any widget down the tree with the same themeData.

B

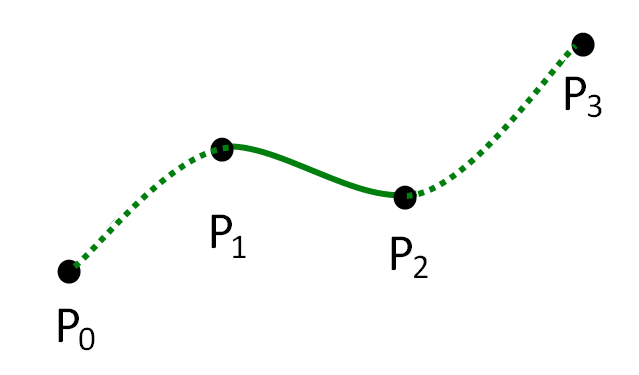
* CapturedThemes: Stores a list of captured InheritedThemes which can then be wrapped around a child to let it inherit themeData.

S

* CatmullRomCurve: A curve for any animation that uses Catmull-Rom spline.

S

* CatmullRomSpline: A type of mathematical spline that takes a list of control points and generates a CatmullRomSpline for it.



S

* Center: Widget that centers its child in the given constraints. If WidthFactor and HeightFactor are non null and constraints are null then multiplies the child widget width and height to the factors.

W

* ChangeNotifier: A class that can be extended or mixed in to provide an API that notifies on changes to the class.

B

* ValueNotifier<T>: A changenotifier that holds a single value, if the value gets changed then all listeners are notified.

S

* Characters: The characters of a string. This class provides an iterable that is used to iterate over the characters of a string, each character may be a different Characters object if operations are performed on it.

H

* CharacterRange: The iterable provided by Characters is a CharacterRange, this class can also be used to create CharacterRange for a given start and stop index.

H

* CheckedModeBanner: The annoying Debug ‘banner’ is a CheckedModeBanner widget. This widget does nothing in release mode.

W

* ChildIndexGetter
* CircleBorder: A border that fits a circle within the available space. Used with ShapeDecoration to draw circle.

S

* CircularNotchedRectangle: A rectangle with a smooth circular notch. Can return a Path object for the same.

S

* ClampingScrollPhysics: Scroll physics that prohibit the screen to drag beyond the scroll widget’s height.

S

* ClampingScrollSimulation: Simulates clamping scroll physics.

S

* Classes
* Clip
* PaintingContext: A place to paint. RenderObjects paint using PaintingContext, which uses multiple Canvas objects throughout its lifetime to composite layers and draw operations.

S

* ClipContext: Clip utilities used by a PaintingContext.

B

* ClipOval: A widget that clips its child in an oval (and subsequently a circle).

W

* ClipPath: Clip using a custom clipper of Path that allows custom shapes.

W

* ClipRRect: Clip using a rectangle.

W

* ClipRect: Clip using a rounded corner rectangle. All the Clip widgets take a clipper which can resize their children.
* ClipboardStatus
* ClipboardStatusNotifier: A value notifier whose value indicates whether the current contents of the clipboard can be pasted.

S

* Color: An immutable 32-bit color value in ARGB. Color can translate colors from hex, ARGB, hex ARGB and RGBO.

S

For hex: 0xZZYYYYYY,here ZZ is the opacity in hex. FF means highest opacity, 00 means lowest. YYYYYY is the hex code for the color.

* ColorFilter: A description of a color filter which takes 2 colors and outputs 1 color for each pixel of a given compositing layer.

S

* ColorFiltered: Applies a ColorFilter to its child.

W

* ColorProperty: DiagnosticsProperty that has a Color as value.

S

* ColorSwatch<T>: A class that has a color and a table of related colors, called ‘swatches’,
* ColorTween: Interpolation between 2 colors.

S

* ColoredBox: A widget that paints the area with a given Color and then draws its widget on top of it.

W

* Column: A widget that displays its children in a vertical array. Doesn’t scroll. Use Expanded widget to fill the available vertical space.

W

* ComponentElement: An Element that compose other Elements. It creates Elements instead of directly creating RenderObjects.

S

* CompositedTransformFollower
* CompositedTransformTarget
* CompoundAnimation: An interface that combines multiple Animations. That is, the values produced are a combination of multiple animations, the combination is describe by us.

B

* ConfirmDismissCallback
* ConnectionState
* ConstantTween: A tween whose begin and end value are the same.

S

* Constants
* ConstrainedBox: A widget that provides additional constraints on its child. We usually use BoxConstraints with it and define min and max height/width using ConstrainedBox.

W

* ConstrainedLayoutBuilder<T extends Constraints>: An abstract superclass for widgets that defer their building until layout. It is similar to Builder widget except it calls the builder function at layout time and provides the constraints to the child widgets. Subclasses must return a RenderObject that mixes in RenderContrainedLayoutBuilder.

B

* ConstraintsTransformBox: A container widget that applies an arbitrary transform to its constraints and sizes its child using the resulting BoxConstraints. This widget has a constraintsTransform parameter which applies Constraints to the widget itself.

W

* Container: The most basic widget in flutter, provides size, alignment for children, constraints etc. properties.

Usually a rectangle or a square.

W

* ContextAction: An abstract subclass of Action which adds an optional BuildContext to the invoke method.

B

* ContinuousRectangleBorder: A rectangular border with smooth continuous transition between straight sides and rounded corners.

S

* ControlsWidgetBuilder
* CreatePlatformViewCallback
* CreateRectTween
* CrossAxisAlignment
* CrossFadeState
* Curve: A parametric animation easing curve, i.e., a mapping of the unit interval to the unit interval. In other words, it allows different values to return between 0.0 and 1.0 following a given pattern, like Curves.easeIn would map value from 0.0 to like 0.8 in the 0.6 of the total duration and then take the 0.4 of the duration to give values from 0.8 to 1.0.

B

* Cubic: A cubic polynomial mapping of the unit interval, its basically a set of Curves that use Cubic interval curves.

H

* Curves: This class implements Curve and provides static methods for some common animation curves. For example: Curves.easeIn, Curves.elasticOut etc..

S

* Curve2D: Abstract class that defines an API for evaluating 2D parametric curves. This is different from Curve as the values are Offset values instead of double. That is, it can be used to affect position of a widget directly.

B

* Curve2DSample: A class that holds a sample of a 2D parametric curve, containing the Offset of the curve at the parametric value t.

H

* CurveTween: Transforms the value of the given animation by the given curve. The difference between CurveTween and CurvedAnimation is that it can be chained with another Tween before being given to an animation object.

S

* CurvedAnimation: An animation that applies a curve to another animation. Useful if we want to apply different curves for forward and reverse animations.

S

* CustomClipper<T>: An interface for providing custom clips. This class is used by Clip Widgets like ClipRect and ClipPath. We can use CustomClipper to customize the default clips. For ClipRect and ClipOval CustomClipper<Rect> can be used, for ClipRRect CustomClipper<RRect> can be used and for ClipPath CustomClipper<Path> can be used.

B

* CustomSingleChildLayout: A widget that defers the layout of its child to a delegate. The delegate can determine the layout constraints and position for the child and the parent.

W

* CustomMultiChildLayout: Similar to CustomSingleChildLayout but it is appropriate when there are complex relationships between size and positioning of multiple widgets.

W

* CustomPaint: This widget allows us to use a CustomPainter class to paint custom UI. The coordinates sent to the painter are of the widget itself and if the size is undefined then paint could go out of bounds hence it is ideal to wrap CustomPaint in widgets like ClipRect.

W

* CustomPainter: This interface is used by CustomPaint and RenderCustomPaint to define custom paint delegate. We extend this class and override some key methods like void paint(Canvas,Size). The Canvas object is what we paint on and the Size object can be used to define the canvas size and offset.  
  B
* CustomPainterSemantics: Creates semantics information describing a rectangle on a canvas.  
  S
* CustomScrollView: A scrollview that creates custom scroll effects using slivers. We can directly supply slivers to this scrollview to create scroll effets.  
  W
* Chip: A material design chip.

Chips are compact elements that represent an attribute, text, entity, or action.

W

* DebugCreator: A wrapper class for the Element that is the creator of a RenderObject.  
  Setting DebugCreator as RenderObject.debugCreator makes it return bettor error messages.  
  S
* DecoderCallback
* DecoratedBox: A widget that paints a Decoration either before or after its child paints.

W

* DecoratedBoxTransition: Animated version of a DecoratedBox that animates the different properties of its Decoration.

W

* DecorationImage: An image for a box decoration. Creates an image to show in a BoxDecoration.

S

* DecorationImagePainter: The painter for a DecorationImage.

S

* DecorationPosition
* DecorationTween: An interpolation between 2 Decorations.

S

* DefaultAssetBundle: A widget that determines the default asset bundle for its descendants.

We can use this widget to override what the current asset bundle is for its children widget, given we provide an asset bundle class to it.

W

* DefaultShaderWarmUp: Default way of warming up Skia shader compilations.

S

* DefaultTextEditingActions: An Actions widget that handles the default text editing behavior for Flutter on the current platform.

W

* DefaultTextEditingShortcuts: A Shortcuts widget with the shortcuts used for the default text editing behavior.

W

* DefaultTextHeightBehavior: Like DefaultTextStyle but only defines TextHeightBehavior. DefaultTextStyle with non-null TextHeightBehavior overrides this widget.

W

* DefaultTextStyle: This widget can be used to define a textstyle which will be used by descendant child widgets which don’t have their own text style.

W

* DefaultTextStyleTransition: Animated version of DefaultTextStyle that animates changes between its text styles.

W

* TransitionDelegate<T>: This abstract class is extended by classes to provide transition animation for pages being added or removed from Navigator.pages.

S

* DefaultTransitionDelegate<T>: The default implementation of TransitionDelegate that the Navigator will use if it doesn’t have a Navigator.transitionDelegate defined.

S

* DefaultWidgetsLocalizations: US English localizations for the widgets library.

S

* DesktopTextSelectionToolbarLayoutDelegate: A class that extendes SingleChildLayoutDelegate and is used by DesktopTextSelectionControls to position itself using this class. This class positions the toolbar at anchor (offset) if it fits, otherwise moves it so that it just fits fully on-screen.

S

* DevToolsDeepLinkProperty: Creates a diagnostics property that displays a deep link to Flutter DevTools.

S

* DiagnosticLevel
* DiagnosticsNode: Defines diagnostics data for a value. It provides a high quality diagnostics data.

S

* DirectionalFocusAction: An Action that moves the focus to the focusable node in the direction configured by the associated DirectionalFocusIntent.direction.

S

* DirectionalFocusIntent: An Intent that represents moving to the next focusable node in the given direction.

S

* DirectionalFocusTraversalPolicyMixin
* Directionality: A widget that determines the ambient directionality of text and text-direction-sensitive render objects.

W

* DismissAction: An action that dismisses the focused widget.

S

* DismissDirection
* DismissDirectionCallback
* DismissIntent: An Intent that dismisses the currently focused widget.

S

* Dismissible: A widget that can be dismissed by dragging in the indicated direction. It automatically animates the disappearing animation.

W

* DisposableBuildContext<T extends State of StatefulWidget>: Provides access to BuildContext of an active Element. That is, an object of this class must be created in initState of the stateful class and it must be disposed in the dispose of the stateful class.

S

* DoNothingAction: An Action, that doesn't perform any action when invoked. This action’s consumeKey can be configured to return false, in which case it will do nothing and stop propagation of key event to other widgets higher in the widget hierarchy.

S

* DoNothingAndStopPropagationIntent: This intent is bound to a DoNothingAction and it doesn’t perform any action nor does it allow propagation of events.

S

* DoNothingAndStopPropagationTextIntent: An Intent to send the event straight to the engine, but only if a TextEditingTarget is focused.

S

* DoNothingIntent: An Intent, that is bound to a DoNothingAction. It can’t be subclassed and it can be used with a Shortcuts to disable a keyboard shortcut.

S

* DragAnchor
* DragAnchorStrategy
* DragDownDetails: Details object for callbacks that use GestureDragDownCallback.

S

* DragEndCallback
* DragEndDetails: Details object for callbacks that use GestureDragEndCallback.

S

* Draggable: A widget that can be dragged to a DragTarget widget. It can display multiple child widgets for multiple states (idle, dragging and the widget left in place where the draggable widget got dragged from).

W

* DraggableDetails: Represents the details when a specific pointer event occurred on the Draggable. Details also include Velocity, Offset and whether DragTarget accepted it. This is also the Details object for callbacks that use DragEndCallback.

S

* DraggableScrollableSheet: This widget is a Draggable at first and when it is dragged till a given limit it acts like a Scrollable widget.

W

* DraggableScrollableActuator: A widget that can notify a descendent DraggableScrollableSheet that it should reset its position to the initial state.

W

* Notification: A notification that can bubble up the widget tree. We can determine the runtimeType of Notification (such as ScrollStartNotification) using the is operator and listen to Notifications using a NotificationListener. We can override the class for our own Notification types.

S and B

* NotificationListener<T extends Notification>: This widget can listen to notification of T type and once a Notification is caught, it is passed to the onNotification delegate which can return bool, true will stop bubbling. This widget just needs to be an ancestor widget to a Notification and not necessarily be a direct parent.

W

* DraggableScrollableNotification: A Notification released by a DraggableScrollableSheet. This is fired when extent ( size and scroll offset of the child widget ) of the widget changes.

S

* DragScrollActivity: The activity a scroll view performs when a the user drags their finger across the screen.

S

* DragSelectionUpdateCallback
* DragStartDetails: Details object for callbacks that use GestureDragStartCallback.

S

* DragTarget: A widget that receives data when a Draggable widget is dropped. This widget can choose to accept or reject a Draggable.

W

* DragTargetAccept
* DragTargetAcceptWithDetails
* DragTargetBuilder
* DragTargetDetails: Represents the details when a pointer event occurred on the DragTarget.

S

* DragTargetLeave
* DragTargetMove
* DragTargetWillAccept
* DragUpdateCallback
* DragUpdateDetails: Details object for callbacks that use GestureDragUpdateCallback.

S

* DraggableCanceledCallback
* DrivenScrollActivity: An activity that animates a scroll view based on animation parameters.

S

* DualTransitionBuilder: A transition builder that animates its child based on the AnimationStatus of the provided animation. It can define the forward animation and reverse animation for its child.

W

* EdgeInsets: An immutable set of offsets in each of the four cardinal directions. Usually used with Padding.

S

* EdgeInsetsDirectional: Same as EdgeInsets but it is affected by TextDirection.

S

* EdgeInsetsGeometry: Base class for EdgeInsets that allow text-direction aware resolution.

B

* EdgeInsetsGeometryTween: An interpolation between two EdgeInsetsGeometrys.

S

* EdgeInsetsTween: An interpolation between two EdgeInsetss.

S

* EditableText: A basic text input field. This widget presents a box where text can be input. It is a barebones version of TextField.

W

* EditableTextState: State for a EditableText.

S

* ElasticInCurve: An oscillating curve that grows in magnitude while overshooting its bounds. An instance of this class with period of 0.4 is available in Curves.

S

* ElasticInOutCurve: An oscillating curve that grows and then shrinks in magnitude while overshooting its bounds.

S

* ElasticOutCurve: An oscillating curve that shrinks in magnitude while overshooting its bounds.

S

* ElementVisitor
* Enums
* ErrorDescription:A detailed explanation of a problem.

S

* ErrorHint: An ErrorHint provides specific, non-obvious advice that may be applicable.

S

* ErrorSummary: A short description of an Error.

S

* ErrorWidget: A widget that renders an exception's message.

W

* ErrorWidgetBuilder
* ExactAssetImage: Fetches an image from an AssetBundle. It requires explicit final assetName and scale on construction unlike AssetImage that picks an appropriate image automatically.

S

* Exceptions
* ExcludeFocus: A widget that controls whether or not the descendants of this widget are focusable.

W

* ExcludeSemantics: A widget that drops all the semantics of its descendants. The opposite direction of BlockSemantics.

S

* ExpandSelectionLeftByLineTextIntent: An Intent to expand the selection left to the start/end of the current line.

S

* ExpandSelectionRightByLineTextIntent: ..
* ExpandSelectionToEndTextIntent: ..
* ExpandSelectionToStartTextIntent: ..
* Expanded: A widget that expands a child of a Row, Column, or Flex so that the child fills the available space. If multiple Expanded widget exist in the same space then the space is divided according to their flex factors.

W

* ExtendSelectionDownTextIntent: An Intent to extend the selection down by one line.

S

* ExtendSelectionLeftByLineTextIntent: ..
* ExtendSelectionLeftByWordAndStopAtReversalTextIntent: ..
* ExtendSelectionLeftByWordTextIntent: ..
* ExtendSelectionLeftTextIntent: An Intent to extend the selection left past the nearest word.

S

* ExtendSelectionRightByLineTextIntent: ..
* ExtendSelectionRightByWordAndStopAtReversalTextIntent: ..
* ExtendSelectionRightByWordTextIntent: ..
* ExtendSelectionRightTextIntent: ..
* ExtendSelectionUpTextIntent: ..
* Extensions
* FadeInImage: A widget that displays a local placeholder image until an image successfully loads from a network source.

W

* FadeTransition: A widget that lets us fade a widget in and out.

W

* FileImage: Decodes the given File object as an image, associating it with the given scale.

S

* FilterQuality
* FittedBox: A widget that scales and positions its child within itself according to fit.

W

* FixedColumnWidth: Sizes the column to a specific number of pixels.

S

* ScrollPosition: Determines which portion of the content is visible in a scroll view. It applies physics to scrolling, and is controlled by current ScrollActivity. ScrollPosition notifies listeners for a change in pixels. A single Scrollable might have multiple ScrollPositions overtime.

S and B

* FixedExtentMetrics: Metrics for a ScrollPosition to a scroll view with fixed item sizes. Creates an immutable snapshot of values associated with a ListWheelScrollView.

S

* FixedExtentScrollController: A controller for scroll views whose items have the same size. This controller is like ScrollController but allows us to go to item indices as well as scroll offsets.

S

* FixedExtentScrollPhysics: A snapping physics that always lands directly on items instead of anywhere within the scroll extent. Must be used alongside FixedExtentScrollController.

S

* FixedScrollMetrics: Creates an immutable snapshot of values associated with a Scrollable viewport.

S

* Flex: A widget that displays its children in a one-dimensional array. It can have multiple children, but they should be able to fit in the available space. Flex is used when we don’t know the main axis its children will be placed in (Horizontal/Vertical) in advance. We use Expanded on it’s children which we want expanding to fill the space left after all the non-flex factor children are laid out.

W

* FlexColumnWidth: Sizes the column by taking a part of the remaining space once all the other columns have been laid out.

S

* FlexFit: An enum that describes how a child is inscribed into the available space. FlexFit.loose means the child’s size takes precedence and FlexFit.tight means that the flex value is prioritised over child’s size and child will fill the available space according to the flex value.

S

* Flexible: A widget that controls how a child of a Row, Column, or Flex flexes. Unlike Expanded it doesn’t require it’s child to fill the available space.

W

* FlippedCurve: A curve that is the reversed inversion of its given curve.

S

* TweenSequence<T>: Enables creating an Animation whose value is defined by a sequence of Tweens. That is, a single animation with multiple tweens. Each Tween in a TweenSequence is defined using a TweenSequenceItem which has a weight and a tween property, the weight property defines the percentage of the total animation duration occupied by a tween.

S

* TweenSequenceItem: A simple holder for one element of a TweenSequence.

H

* FlippedTweenSequence: Enables creating a flipped Animation whose value is defined by a sequence of Tweens. It creates a TweenSequence.

S

* RenderFlow: A class that implements the flow layout algo.

B

* Flow: A widget that sizes and positions children efficiently, according to the logic in a FlowDelegate. Flow layouts are optimized for repositioning children using transformation matrices.

W

* FlowDelegate: A delegate that controls the appearance of a flow layout.

S

* FlowPaintingContext: A context in which a FlowDelegate paints. Provides information about the current size of the container and the children and a mechanism for painting children.

S

* FlutterError: Error class that reports Flutter-specific errors. Provides listeners for handling errors as well.

S

* FlutterErrorDetails: A class that holds an error that is caught by fluttererror. Used with it.

H

FlutterError.reportError(FlutterErrorDetails(

exception: error,

library: 'Flutter test framework',

context: ErrorSummary('while running async test code'),

));

* FlutterLogoDecoration: Creates a Decoration that knows how to paint Flutter’s logo.

S

* FlutterLogoStyle
* FocusNode: An obj that can be used by a stateful widget to obtain keyboard focus and also handle keyboard events.

S

* Focus: A widget that manages a FocusNode to allow keyboard focus be given to this widget and its children.

W

* FocusScope:
* FocusAttachment
* FocusHighlightMode
* FocusHighlightStrategy
* FocusManager
* FocusOnKeyCallback
* FocusOrder
* FocusScopeNode
* FocusTraversalGroup
* FocusTraversalOrder
* FocusTraversalPolicy
* FocusableActionDetector
* FontStyle
* FontWeight
* ForcePressDetails
* Form
* FormField
* FormFieldBuilder
* FormFieldSetter
* FormFieldState
* FormFieldValidator
* FormState
* FractionColumnWidth
* FractionalOffset
* FractionalOffsetTween
* FractionalTranslation
* FractionallySizedBox
* Functions
* FutureBuilder
* GenerateAppTitle
* GestureDetector
* GestureDragCancelCallback
* GestureDragDownCallback
* GestureDragEndCallback
* GestureDragStartCallback
* GestureDragUpdateCallback
* GestureForcePressEndCallback
* GestureForcePressPeakCallback
* GestureForcePressStartCallback
* GestureForcePressUpdateCallback
* GestureLongPressCallback
* GestureLongPressEndCallback
* GestureLongPressMoveUpdateCallback
* GestureLongPressStartCallback
* GestureLongPressUpCallback
* GestureRecognizerFactory
* GestureRecognizerFactoryConstructor
* GestureRecognizerFactoryInitializer
* GestureRecognizerFactoryWithHandlers
* GestureScaleEndCallback
* GestureScaleStartCallback
* GestureScaleUpdateCallback
* GestureTapCallback
* GestureTapCancelCallback
* GestureTapDownCallback
* GestureTapUpCallback
* GlobalKey
* GlobalObjectKey
* GlowingOverscrollIndicator
* Gradient
* GradientRotation
* GradientTransform
* GridPaper
* GridView
* GrowthDirection
* HSLColor
* HSVColor
* Hero
* HeroController
* HeroControllerScope
* HeroFlightDirection
* HeroFlightShuttleBuilder
* HeroMode
* HeroPlaceholderBuilder
* HitTestBehavior
* HoldScrollActivity
* HtmlElementView
* HttpClientProvider
* Icon
* IconData
* IconDataProperty
* IconTheme
* IconThemeData
* IdleScrollActivity
* IgnorePointer
* Image
* ImageCache
* ImageCacheStatus
* ImageChunkEvent
* ImageChunkListener
* ImageConfiguration
* ImageErrorListener
* ImageErrorWidgetBuilder
* ImageFiltered
* ImageFrameBuilder
* ImageIcon
* ImageInfo
* ImageListener
* ImageLoadingBuilder
* ImageProvider
* ImageRepeat
* ImageShader
* ImageSizeInfo
* ImageStream
* ImageStreamCompleter
* ImageStreamCompleterHandle
* ImageStreamListener
* ImplicitlyAnimatedWidget
* ImplicitlyAnimatedWidgetState
* IndexedSemantics
* IndexedSlot
* IndexedStack
* IndexedWidgetBuilder
* InheritedElement
* InheritedModel
* InheritedModelElement
* InheritedNotifier
* InheritedTheme
* InheritedWidget
* InitialRouteListFactory
* InlineSpan
* InlineSpanSemanticsInformation
* InlineSpanVisitor
* InspectorSelectButtonBuilder
* InspectorSelection
* InspectorSelectionChangedCallback
* InspectorSerializationDelegate
* IntTween
* Intent
* InteractiveViewer
* Interval
* IntrinsicColumnWidth
* IntrinsicHeight
* IntrinsicWidth
* KeepAlive
* KeepAliveHandle
* KeepAliveNotification
* Key
* KeyEventResult
* KeySet
* KeyedSubtree
* LabeledGlobalKey
* LayerLink
* LayoutBuilder
* LayoutChangedNotification
* LayoutId
* LayoutWidgetBuilder
* LeafRenderObjectElement
* LeafRenderObjectWidget
* LexicalFocusOrder
* LimitedBox
* LinearGradient
* ListBody
* ListView
* ListWheelChildBuilderDelegate
* ListWheelChildDelegate
* ListWheelChildListDelegate
* ListWheelChildLoopingListDelegate
* ListWheelElement
* ListWheelScrollView
* ListWheelViewport
* Listenable
* Listener
* LocalHistoryEntry
* LocalHistoryRoute
* LocalKey
* Locale
* LocaleListResolutionCallback
* LocaleResolutionCallback
* Localizations
* LocalizationsDelegate
* LogicalKeySet
* LongPressDraggable
* LongPressEndDetails
* LongPressMoveUpdateDetails
* LongPressStartDetails
* MainAxisAlignment
* MainAxisSize
* MaskFilter
* Matrix4
* Matrix4Tween
* MatrixUtils
* MaxColumnWidth
* MediaQuery
* MediaQueryData
* MemoryImage
* MergeSemantics
* MetaData
* MinColumnWidth
* Mixins
* ModalBarrier
* ModalRoute
* MouseCursor
* MouseRegion
* MoveSelectionDownTextIntent
* MoveSelectionLeftByLineTextIntent
* MoveSelectionLeftByWordTextIntent
* MoveSelectionLeftTextIntent
* MoveSelectionRightByLineTextIntent
* MoveSelectionRightByWordTextIntent
* MoveSelectionRightTextIntent
* MoveSelectionToEndTextIntent
* MoveSelectionToStartTextIntent
* MoveSelectionUpTextIntent
* MultiChildLayoutDelegate
* MultiChildRenderObjectElement
* MultiChildRenderObjectWidget
* MultiFrameImageStreamCompleter
* NavigationMode
* NavigationToolbar
* Navigator
* NavigatorFinderCallback
* NavigatorObserver
* NavigatorState
* NestedScrollView
* NestedScrollViewHeaderSliversBuilder
* NestedScrollViewState
* NestedScrollViewViewport
* NetworkImage
* NetworkImageLoadException
* NeverScrollableScrollPhysics
* NextFocusAction
* NextFocusIntent
* NotchedShape
* Notification
* NotificationListener
* NotificationListenerCallback
* NullableIndexedWidgetBuilder
* NumericFocusOrder
* ObjectKey
* Offset
* Offstage
* OnInvokeCallback
* OneFrameImageStreamCompleter
* Opacity
* OrderedTraversalPolicy
* Orientation
* OrientationBuilder
* OrientationWidgetBuilder
* OutlinedBorder
* Overflow
* OverflowBar
* OverflowBarAlignment
* OverflowBox
* Overlay
* OverlayEntry
* OverlayRoute
* OverlayState
* OverscrollIndicatorNotification
* OverscrollNotification
* Padding
* Page
* PageController
* PageMetrics
* PageRoute
* PageRouteBuilder
* PageRouteFactory
* PageScrollPhysics
* PageStorage
* PageStorageBucket
* PageStorageKey
* PageView
* Paint
* PaintImageCallback
* PaintingBinding
* PaintingContext
* PaintingStyle
* ParametricCurve
* ParentDataElement
* ParentDataWidget
* Path
* PathFillType
* PathOperation
* PerformanceOverlay
* PhysicalModel
* PhysicalShape
* Placeholder
* PlaceholderAlignment
* PlaceholderDimensions
* PlaceholderSpan
* PlatformRouteInformationProvider
* PlatformViewCreationParams
* PlatformViewLink
* PlatformViewSurface
* PlatformViewSurfaceFactory
* PointerCancelEvent
* PointerCancelEventListener
* PointerDownEvent
* PointerDownEventListener
* PointerEvent
* PointerMoveEvent
* PointerMoveEventListener
* PointerUpEvent
* PointerUpEventListener
* PopNavigatorRouterDelegateMixin
* PopPageCallback
* PopupRoute
* Positioned
* PositionedDirectional
* PositionedTransition
* PreferredSize
* PreferredSizeWidget
* PreviousFocusAction
* PreviousFocusIntent
* PrimaryScrollController
* PrioritizedAction
* PrioritizedIntents
* Properties
* ProxyAnimation
* ProxyElement
* ProxyWidget
* RRect
* RSTransform
* RadialGradient
* Radius
* RangeMaintainingScrollPhysics
* RawAutocomplete
* RawDialogRoute
* RawGestureDetector
* RawGestureDetectorState
* RawImage
* RawKeyEvent
* RawKeyboardListener
* RawScrollbar
* RawScrollbarState
* ReadingOrderTraversalPolicy
* RebuildDirtyWidgetCallback
* Rect
* RectTween
* RelativePositionedTransition
* RelativeRect
* RelativeRectTween
* RenderBox
* RenderComparison
* RenderConstrainedLayoutBuilder
* RenderNestedScrollViewViewport
* RenderObject
* RenderObjectElement
* RenderObjectToWidgetAdapter
* RenderObjectToWidgetElement
* RenderObjectWidget
* RenderSemanticsGestureHandler
* RenderSliverOverlapAbsorber
* RenderSliverOverlapInjector
* ReorderCallback
* ReorderItemProxyDecorator
* ReorderableDelayedDragStartListener
* ReorderableDragStartListener
* ReorderableList
* ReorderableListState
* RepaintBoundary
* RequestFocusAction
* RequestFocusIntent
* ResizeImage
* RestorableBool
* RestorableBoolN
* RestorableChangeNotifier
* RestorableDateTime
* RestorableDateTimeN
* RestorableDouble
* RestorableDoubleN
* RestorableInt
* RestorableIntN
* RestorableListenable
* RestorableNum
* RestorableNumN
* RestorableProperty
* RestorableRouteBuilder
* RestorableRouteFuture
* RestorableString
* RestorableStringN
* RestorableTextEditingController
* RestorableValue
* RestorationBucket
* RestorationMixin
* RestorationScope
* ReverseAnimation
* ReverseTween
* RichText
* RootBackButtonDispatcher
* RootRenderObjectElement
* RootRestorationScope
* RotatedBox
* RotationTransition
* RoundedRectangleBorder
* Route
* RouteAware
* RouteCompletionCallback
* RouteFactory
* RouteInformation
* RouteInformationParser
* RouteInformationProvider
* RouteListFactory
* RouteObserver
* RoutePageBuilder
* RoutePopDisposition
* RoutePredicate
* RoutePresentationCallback
* RouteSettings
* RouteTransitionRecord
* RouteTransitionsBuilder
* Router
* RouterDelegate
* Row
* SafeArea
* SawTooth
* ScaleEndDetails
* ScaleStartDetails
* ScaleTransition
* ScaleUpdateDetails
* ScrollAction
* ScrollActivity
* ScrollActivityDelegate
* ScrollAwareImageProvider
* ScrollBehavior
* ScrollConfiguration
* ScrollContext
* ScrollController
* ScrollDragController
* ScrollEndNotification
* ScrollHoldController
* ScrollIncrementCalculator
* ScrollIncrementDetails
* ScrollIncrementType
* ScrollIntent
* ScrollMetrics
* ScrollNotification
* ScrollNotificationPredicate
* ScrollPhysics
* ScrollPosition
* ScrollPositionAlignmentPolicy
* ScrollPositionWithSingleContext
* ScrollSpringSimulation
* ScrollStartNotification
* ScrollUpdateNotification
* ScrollView
* ScrollViewKeyboardDismissBehavior
* Scrollable
* ScrollableDetails
* ScrollableState
* ScrollableWidgetBuilder
* ScrollbarPainter
* SelectAction
* SelectIntent
* SelectionChangedCallback
* SelectionChangedCause
* SemanticIndexCallback
* Semantics
* SemanticsBuilderCallback
* SemanticsDebugger
* SemanticsGestureDelegate
* Shader
* ShaderCallback
* ShaderMask
* ShaderWarmUp
* Shadow
* ShapeBorder
* ShapeBorderClipper
* ShapeDecoration
* ShortcutManager
* ShortcutMapProperty
* Shortcuts
* ShrinkWrappingViewport
* Simulation
* SingleChildLayoutDelegate
* SingleChildRenderObjectElement
* SingleChildRenderObjectWidget
* SingleChildScrollView
* SingleTickerProviderStateMixin
* Size
* SizeChangedLayoutNotification
* SizeChangedLayoutNotifier
* SizeTransition
* SizeTween
* SizedBox
* SizedOverflowBox
* SlideTransition
* SliverAnimatedList
* SliverAnimatedListState
* SliverAnimatedOpacity
* SliverChildBuilderDelegate
* SliverChildDelegate
* SliverChildListDelegate
* SliverFadeTransition
* SliverFillRemaining
* SliverFillViewport
* SliverFixedExtentList
* SliverGrid
* SliverGridDelegate
* SliverGridDelegateWithFixedCrossAxisCount
* SliverGridDelegateWithMaxCrossAxisExtent
* SliverIgnorePointer
* SliverLayoutBuilder
* SliverLayoutWidgetBuilder
* SliverList
* SliverMultiBoxAdaptorElement
* SliverMultiBoxAdaptorWidget
* SliverOffstage
* SliverOpacity
* SliverOverlapAbsorber
* SliverOverlapAbsorberHandle
* SliverOverlapInjector
* SliverPadding
* SliverPersistentHeader
* SliverPersistentHeaderDelegate
* SliverPrototypeExtentList
* SliverReorderableList
* SliverReorderableListState
* SliverSafeArea
* SliverToBoxAdapter
* SliverVisibility
* SliverWithKeepAliveWidget
* SmartDashesType
* SmartQuotesType
* Spacer
* Stack
* StackFit
* StadiumBorder
* State
* StateSetter
* StatefulBuilder
* StatefulElement
* StatefulWidget
* StatefulWidgetBuilder
* StatelessElement
* StatelessWidget
* StatusTransitionWidget
* StepTween
* StreamBuilder
* StreamBuilderBase
* StringCharacters
* StrokeCap
* StrokeJoin
* StrutStyle
* SweepGradient
* SystemMouseCursors
* Table
* TableBorder
* TableCell
* TableCellVerticalAlignment
* TableColumnWidth
* TableRow
* TapDownDetails
* TapUpDetails
* TargetPlatform
* Text
* TextAffinity
* TextAlign
* TextAlignVertical
* TextBaseline
* TextBox
* TextDecoration
* TextDecorationStyle
* TextDirection
* TextEditingAction
* TextEditingActionTarget
* TextEditingController
* TextEditingValue
* TextHeightBehavior
* TextInputType
* TextLeadingDistribution
* TextOverflow
* TextPainter
* TextPosition
* TextRange
* TextSelection
* TextSelectionControls
* TextSelectionDelegate
* TextSelectionGestureDetector
* TextSelectionGestureDetectorBuilder
* TextSelectionGestureDetectorBuilderDelegate
* TextSelectionHandleType
* TextSelectionOverlay
* TextSelectionToolbarLayoutDelegate
* TextSpan
* TextStyle
* TextStyleTween
* TextWidthBasis
* Texture
* Threshold
* TickerCanceled
* TickerFuture
* TickerMode
* TickerProvider
* TickerProviderStateMixin
* TileMode
* Title
* Tolerance
* ToolbarBuilder
* ToolbarItemsParentData
* ToolbarOptions
* TrackingScrollController
* TrainHoppingAnimation
* Transform
* TransformProperty
* TransformationController
* TransitionBuilder
* TransitionDelegate
* TransitionRoute
* TraversalDirection
* Tween
* TweenAnimationBuilder
* TweenConstructor
* TweenSequence
* TweenSequenceItem
* TweenVisitor
* Typedefs
* UiKitView
* UnconstrainedBox
* UnfocusDisposition
* UniqueKey
* UniqueWidget
* UnmanagedRestorationScope
* UserScrollNotification
* ValueChanged
* ValueGetter
* ValueKey
* ValueListenableBuilder
* ValueNotifier
* ValueSetter
* ValueWidgetBuilder
* Velocity
* VertexMode
* VerticalDirection
* Viewport
* ViewportBuilder
* ViewportNotificationMixin
* Visibility
* VoidCallback
* Widget
* WidgetBuilder
* WidgetInspector
* WidgetInspectorService
* WidgetOrderTraversalPolicy
* WidgetSpan
* WidgetToRenderBoxAdapter
* WidgetsApp
* WidgetsBinding
* WidgetsBindingObserver
* WidgetsFlutterBinding
* WidgetsLocalizations
* WillPopCallback
* WillPopScope
* Wrap
* WrapAlignment
* WrapCrossAlignment
* applyBoxFit
* axisDirectionIsReversed
* axisDirectionToAxis
* childDragAnchorStrategy
* combineSemanticsInfo
* createLocalImageConfiguration
* debugAssertAllPaintingVarsUnset
* debugAssertAllWidgetVarsUnset
* debugCheckHasDirectionality
* debugCheckHasMediaQuery
* debugCheckHasOverlay
* debugCheckHasTable
* debugCheckHasWidgetsLocalizations
* debugChildrenHaveDuplicateKeys
* debugDescribeFocusTree
* debugDescribeTransform
* debugDisableShadows
* debugDumpApp
* debugDumpFocusTree
* debugDumpLayerTree
* debugDumpRenderTree
* debugFlushLastFrameImageSizeInfo
* debugHighlightDeprecatedWidgets
* debugImageOverheadAllowance
* debugInvertOversizedImages
* debugIsLocalCreationLocation
* debugItemsHaveDuplicateKeys
* debugNetworkImageHttpClientProvider
* debugOnPaintImage
* debugOnRebuildDirtyWidget
* debugPrint
* debugPrintBuildScope
* debugPrintGlobalKeyedWidgetLifecycle
* debugPrintRebuildDirtyWidgets
* debugPrintScheduleBuildForStacks
* debugPrintStack
* debugProfileBuildsEnabled
* debugWidgetBuilderValue
* decodeImageFromList
* defaultScrollNotificationPredicate
* factory
* flipAxis
* flipAxisDirection
* getAxisDirectionFromAxisReverseAndDirectionality
* hashList
* hashValues
* imageCache
* immutable
* kAlwaysCompleteAnimation
* kAlwaysDismissedAnimation
* mustCallSuper
* optionalTypeArgs
* paintBorder
* paintImage
* paintZigZag
* pointerDragAnchorStrategy
* positionDependentBox
* precacheImage
* primaryFocus
* protected
* required
* runApp
* showGeneralDialog
* textDirectionToAxisDirection
* transformDebugCreator
* visibleForTesting

• Animated Widgets:

AlignTransition

AnimatedAlign

AnimatedBuilder

AnimatedContainer

AnimatedCrossFade

AnimatedList

AnimatedModalBarrier

AnimatedOpacity

AnimatedPadding

AnimatedPhysicalModel

AnimatedPositioned

AnimatedPositionedDirectional

AnimatedSize

AnimatedSwitcher

DecoratedBoxTransition

DefaultTextStyleTransition

DualTransitionBuilder

• ScrollableScrollPhysics: Physics classes for scrollables.

AlwaysScrollableScrollPhysics

BouncingScrollPhysics

ClampingScrollPhysics

FixedExtentScrollPhysics

• Positional Widgets:

Align

Baseline

• Tween Widgets: Useful to interpolate between 2 values.

BorderTween

BorderRadiusTween

ColorTween

ConstantTween

DecorationTween

EdgeInsetsTween

EdgeInsetsGeometryTween