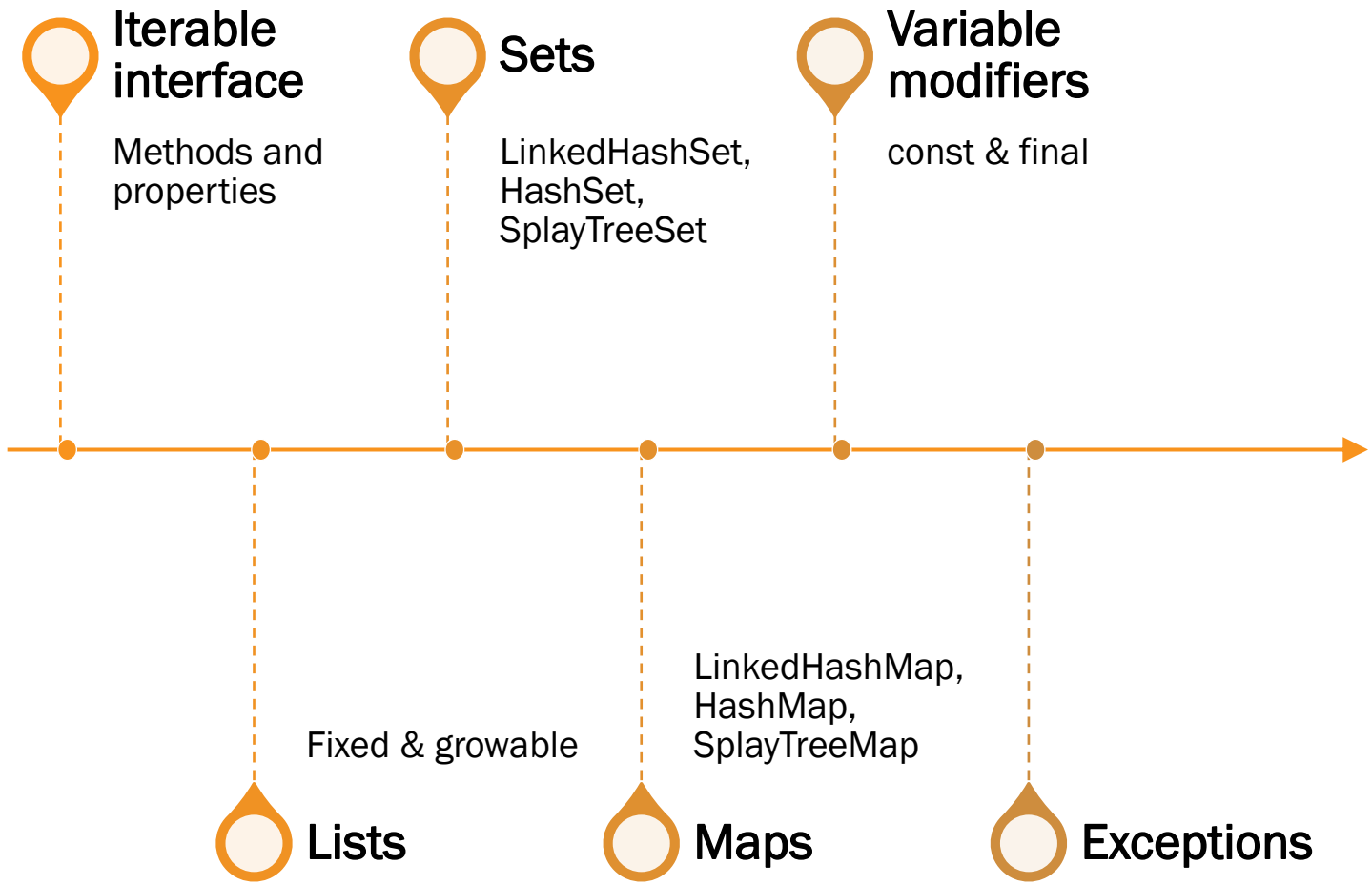
DART Language

COURSE 2 (REV 6)

GAVRILUT DRAGOS

Agenda


Iterable interface
Methods and properties
Sets
LinkedHashSet, HashSet, SplayTreeSet
Variable modifiers
const & final
Fixed & growable
Lists
LinkedHashMap, HashMap, SplayTreeMap
Maps
Exceptions

Iterable

Iterable

Dart provides an interface (abstract class) called ***Iterable*** that offers the basic functionality of every container. Lists / sets and maps are templates derived from ***Iterable.*** One can not create an object of type ***Iterable***, however a list/set or map can be casted to an ***Iterable*** object.

Properties:

`T Iterable<T>.first`	Returns the first element from the collection or StateError exception otherwise
`T Iterable<T>.last`	Returns the last element from the collection or StateError exception otherwise
`T Iterable<T>.single`	Returns the first element from the collection if the collection has only ONE element or StateError exception otherwise
`int Iterable<T>.length`	Number of elements in the collection
`bool Iterable<T>.isEmpty`	True if the collection is empty, false otherwise
`bool Iterable<T>.isNotEmpty`	False if the collection is empty, true otherwise

Iterable

Casting methods (used to convert from one iterable to another one).

`Iterable<U> Iterable<T>.cast<U> ()`	Creates a new iterable where every T element is casted out to a U element.
`Set<T> Iterable<T>.toSet ()`	Creates a Set of type T from the existing Iterable
`List<T> Iterable<T>.toList ()`	Creates a List of type T from the existing Iterable

Iterable methods (used to create another Iterable from the existing one based on some conditions).

`Iterable<T> Iterable<T>.skip(int count)`
`Iterable<T> Iterable<T>.skipWhile(bool validate(T object))`
`Iterable<T> Iterable<T>.take(int count)`
`Iterable<T> Iterable<T>.takeWhile(bool validate(T object))`
`Iterable<T> Iterable<T>.where(bool retainIfTrueFunction(T object))`

Iterable

Mapping values methods (used to convert from one iterable to another type (including another iterator) by converting all of its values).

```
Iterable<U> Iterable<T>.map<U> (U convertFunction(T object))
Iterable<T> Iterable<T>.expand(Iterable<T> convertToIterable(T object))
T           Iterable<T>.reduce(T comboneFunction(T value, T object))
bool        Iterable<T>.any(bool validate(T object))
bool        Iterable<T>.every(bool validate(T object))
String      Iterable<T>.join([Separator = ""])
```

Iterable also provides a generic .forEach method to easily iterate through all elements from a collection.

```
void Iterable<T>.forEach (void processElement(T element))
```

Iterable

Finding values methods (used to find an element or check the existence of elements in a collection).

```
bool Iterable<T>.contains(T? object)
T    Iterable<T>.elementAt(int index)
T    Iterable<T>.firstWhere(bool validate(T object)), {T orElse()?})
T    Iterable<T>.lastWhere(bool validate(T object)), {T orElse()?})
```

We will discuss more about these methods when discussion one of their implementations (List).

Lists

Lists

DART has two type of lists:

- Fixed sized
- Growable

Characteristics:

- Dart lists hold elements of the same type and use “[...]” to define them.
- There is a special type of parametrized list that allows storing elements of different types.
- Dart lists are based on templates/generics. This mean that upon construction, list element type can be provided. However, if not provided, Dart can infer it from initialization parameters.
- Dart lists work based on references to an object (exception for this case are basic type (number, bool and string) that are copied.

Lists

Creating an int list with some values:

```
void main() {
        var l = [1,2,3];
        print(l);
}
```

Creating an empty int list :

```
var l = <int>[]; // type cannot be inferred so it has to be specified
```

A list is a template of type List<T>. As such, List<T> can also be used to construct an object. However, in case of list there is no constructor but several static methods that can be used to create an object.

```
void main() {
        var l = [1,2,3];
}
```

is equivalent to

```
void main() {
    var l = List<int>.from([1,2,3]);
}
```

Lists

Dar lists have several properties that can be used: .first , .last , .length, .reversed , .isEmpty , .isNotEmpty

```
void main() {
        var l = ["hello","dart","zzz","alongstring"];
        print(l.first);        // hello
        print(l.last);         // alongstring
        print(l.isEmpty);      // false
        print(l.isNotEmpty);   // true
        print(l.reversed);     // (alongstring, zzz, dart, hello)
        print(l.length);       // 4
}
```

Lists

List constructors / static methods that can be used to create a list:

```
List<T>.empty({bool growable = false})
List<T>.filled(int len, T value, {bool growable = false})
List<T>.generate(int len, T generatorFnc(int index), {bool growable = false})
List<T>.of(Iterable<T> elements, {bool growable = false})
List<T>.from(Iterable elements, {bool growable = false})
List<T>.unmodifiable(Iterable elements)
```

Keep in mind that List<T> is not always necessary as dart will attempt to infer type T from parameters. This is why there are two very similar version (**.from** and **.of** → they can help inference process understand type T);

Lists

All object from dart are derived from one singular dynamic type (Object). As such if a type can not be inferred, it is considered to be the base object (dynamic).

```dart
void main() {
    var l1 = [1,2,3];
    print(l1.runtimeType);
    var l2 = [];
    print(l2.runtimeType);
}
```

List<int>
List<dynamic>

```dart
void main() {
    var l = List<int>.from([1,2,3]);
    print(l.runtimeType);  // List<int>
    var l2 = List.empty();
    print(l2.runtimeType); // List<dynamic>
    var l3 = List<int>.empty();
    print(l3.runtimeType); // List<int>
}
```

List<int>
List<dynamic>
List<int>

OBS: The output type will be different if you run the same examples in dart pad (JavaScript based) or compile them directly.

Lists

The main difference between .from and .of constructors if the .from takes in consideration the parametrize type while .of considers the initialization values as well for the inference process.

```
void main() {
	var l1 = List<int>.from([1,2,3]);
	print(l1.runtimeType);			// l1 = List<int>
	var l2 = List.from([1,2,3]);
	print(l2.runtimeType);			// l2 = List<dynamic>
//-----------------------------------------------------------------
	var l3 = List<int>.of([1,2,3]);
	print(l3.runtimeType);			// l3 = List<int>
	var l4 = List.of([1,2,3]);
	print(l4.runtimeType);			// l4 = List<int>
}
```

Lists

To populate a list based on a mathematical relation between list index and its value, use generate constructor. This is similar to map paradigm from Python. Both lambdas and functions can be used.

```
int myFunction(int index) {
        return index % 3;
}
void main() {
        var l = List<int>.generate(5,(i)=>i*i); // i=int (inferred)
        print(l); // [0, 1, 4, 9, 16]
        var l2 = List<int>.generate(5,(int i)=>i*i*i); // i=int (declared)
        print(l2); // [0, 1, 8, 27, 64]
        var l3 = List<int>.generate(5,myFunction);
        print(l3); // [0, 1, 2, 0, 1]
}
```

Lists

The existence of *List<dynamic>* allows one to create a heterogenous type list.

```
void main() {
        var l = [10, "test", 1.5];
        print(l);
        print(l[0].runtimeType);
        print(l[1].runtimeType);
        print(l[2].runtimeType);
}
```

```
[10, test, 1.5]
int
String
double
```

```
void main() {
        var l = <int>[10, "test", 1.5];
}
```

In this case a compile error will be triggered as “l” has to contain int elements

Lists

To add an element to a list use add or addAll methods:

```
List<T>.add(T value)
```

```
List<T>.addAll(Iterable<T> elements)
```

Operator += can also be used (similar to Python lists). However, the behavior is different as += is an operator defined in the following way: T+=V ⇔ T = T + V. This means that the new element(s) is/are not added to the existing list, but a new list is created that is the sum of the first two.

```
void main() {
    var l = [1,2,3];
    l.add(4);
    print(l); // [1,2,3,4]
    l.addAll([10,20,30]);
    print(l); // [1,2,3,4,10,20,30]
}
```

or

```
void main() {
    var l = [1,2,3];
    l += [4];
    print(l); // [1,2,3,4]
    l += [10,20,30];
    print(l); // [1,2,3,4,10,20,30]
}
```

Lists

So why use add or addAll instead of += operator ?

```
void main() {
        var l = List<int>.from([1,2,3], growable: false );
        print(l);
        l.add(5);
}
```

OBS: List “l” is defined as a fixed list (non-growable). This means that adding new elements to the list will throw an error.

Upon execution an “**Unsupported operation: Cannot add to a fixed-length list**” will be thrown.

Lists

So why use add or addAll instead of += operator ?

Now let's write the previous example, but use += instead of add

```
void main() {
        var l = List<int>.from([1,2,3], growable: false );
        l += [4];
        l.add(5);
        print(l); // [1,2,3,4,5]
}
```

The following syntax: l += [4]; will be execute as l = l + [4]. Meaning that a new list will be created that is the result of the sum between the original "l" list and [4]. The new list will be growable. As a result, l.add(5) will work as expected !

Lists

The “growable” parameter is extremely useful for array creation. The following code creates an array of 5 int elements. Methods like add or addAll can not be used. However, elements can be accessed and modified via the [index] operator.
To create an array-like list use the List<T>.filled(int len, T value) constructor.

```
void main() {
        var l = List<int>.filled(5,0, growable: false);
        print(l); // [0, 0, 0, 0, 0]
        l[0] = 1;
        l[1] = 2;
        l[3] = 3;
        print(l); // [1, 2, 0, 3, 0]
        l[10] = 100; // index out of bounds exception during runtime
}
```

Lists

The same technique can be used to create a bi-dimensional array.

```
void main() {
        var l = List<List<int>>.filled(
                        5, // 5 rows
                        List<int>.filled(
                                3, // 3 columns
                                0, // default value for all elements
                                growable: false),
                        growable: false);
        print(l); // [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
}
```

This code will create a 5x3 matrix (5 rows and 3 columns).

Lists

Another way to create a list is to use the cascade operator "..." with the method "addAll"

```
void main() {
        var l1 = <int>[]..addAll([1,2,3]);
        print(l1.runtimeType); // List<int>
        var l2 = []..addAll([1,2,3]);
        print(l2.runtimeType); // List<dynamic>
}
```

If using this method and a type-specific list is desired, make sure that you prefix the empty list creation with the type you want !

In the previous example, l1 is a List<int>, while l2 is a List<dynamic>

Lists

To insert an element into a list use insert or insertAll methods:

```
List<T>.insert(int index, T value)
```

```
List<T>.insertAll(int index, Iterable<T> elements)
```

Use List<T>.length to insert an element at the end of the list.

```
void main() {
        var l = ["A","B","C","D"];
        l.insert(2,"**");
        print(l); // [A, B, **, C, D]
        l.insertAll(0,["Q","W","E"]);
        print(l); // [Q, W, E, A, B, **, C, D]
        l.insert(l.length,"END");
        print(l); // [Q, W, E, A, B, **, C, D, END]
}
```

Lists

To insert an element into a list use insert or insertAll methods:

```
List<T>.insert(int index, T value)
List<T>.insertAll(int index, Iterable<T> elements)
```

Using a position outside the boundaries of the list will trigger a runtime exception:

```
void main() {
        var l = ["A","B","C","D","E","F","G","H","I","J"];
        l.insert(100,"XX");   // runtime expection:
                              // RangeError: Invalid value:
                              // Not in inclusive range 0..9: 100
}
```

Lists

To remove an element from a list, the following methods can be used:

```
bool List<T>.remove(T? object)
T    List<T>.removeAt(int index)
void List<T>.removeRange(int start, int end)
void List<T>.removeWhere(bool removeIfTrueFunction(T object))
void List<T>.retainWhere(bool retainIfTrueFunction(T object))
T    List<T>.removeLast()
void List<T>.clear()
```

What is interesting to observed is that removeAt and removeLast also return the object that was removed from the list. This feature actually makes them easily to use for stacks or queues (as they simulate a pop instruction).

Use .clear to remove all elements from a list.

Lists

```
void main() {
        var l = [1,2,3,4,5,6,7,8,9,10];
        print(l.removeLast());          // 10
        print(l);                       // [1, 2, 3, 4, 5, 6, 7, 8, 9]
        print(l.removeAt(2));           // 3
        print(l);                       // [1, 2, 4, 5, 6, 7, 8, 9]
        l.removeRange(1,3);
        print(l);                       // [1, 5, 6, 7, 8, 9]
        l.removeWhere((i)=>i%2==0);
        print(l);                       // [1, 5, 7, 9]
        print(l.remove(1000));          // false
        print(l.remove(1));             // true
        print(l);                       // [5, 7, 9]
}
```

Lists

To find or check the existence of an element in a list use the following APIs:

```
bool List<T>.contains(T? object)
int  List<T>.indexAt(T object, [int start = 0])
int  List<T>.lastIndexAt(T object, [int start?])
int  List<T>.indexWhere(bool validate(T object)), [int start = 0])
int  List<T>.lastIndexWhere(bool validate(T object)), [int start?])
T    List<T>.firstWhere(bool validate(T object)), {T orElse()?})
T    List<T>.lastWhere(bool validate(T object)), {T orElse()?})
```

The return of index based functions is either -1 if the element was not found or the position in the list.

Lists

```
void main() {
        var l = [1,2,3,4,5,6,7,8,9,10];
        print(l.indexOf(1000));                    // -1 (1000 is not
                                                   // in the list)
        print(l.indexOf(4));                       // 3 (4 has index 3 on
                                                   // a 0-based list)
        print(l.indexWhere((i)=>i%2==0));          // 1 (l[1]==2)
                                                   // (first odd number)
        print(l.lastIndexWhere((i)=>i%2==0));      // 9 (l[9]==10)
                                                   // (last odd number)


        print(l.contains(5));                      // true
        print(l.contains(1000));                   // false
}
```

Lists

```
void main() {
        var l = ["Dart","hello","World"];

        var e = l.firstWhere((i)=>i.length==4);
        print(e); // Dart
        var f = l.firstWhere((i)=>i.length==10,orElse:()=>"Not found");
        print(f); // Not found
        var g = l.lastWhere((i)=>i.length==5,orElse:()=>"Not found");
        print(g); // World
        var j = l.firstWhere((i)=>i.length==100);   // runtime exception
                                                    // Uncaught Error: Bad
                                                    // state: No element
}
```

Lists

Just like Python that has several methods to automatically process a list and obtain another one (such as filer or map) , Dart has some similar methods as well:

```
Iterable<T> List<T>.getRange(int start, int end)
Iterable<U> List<T>.map<U> (U convertFunction(T object)))
Iterable<T> List<T>.skip(int count)
Iterable<T> List<T>.take(int count)
Iterable<T> List<T>.where(bool retainIfTrueFunction(T object)))
List<T>     List<T>.sublist(int start, [int? end])
Iterable<T> List<T>.expand(Iterable<T> convertToIterable(T object))
T           List<T>.reduce(T comboneFunction(T value, T object))
```

As each one of those methods produce an object, they can be used in a chained expression.

Lists

```
void main() {
        var l = [1,2,3,4,5];
        var l2 = l.map<int>((i)=>i*i);       // (1, 4, 9, 16, 25)
        var l3 = l2.skip(2);                 // (9, 16, 25)
        var l4 = l3.where((i)=>i%2!=0);      // (9, 25)
}
```

or we can obtain the same results via a chain of commands such as:

```
void main() {
        var l = [1,2,3,4,5];
        var l2 = l.map<int>((i)=>i*i)
                  .skip(2)
                  .where((i)=>i%2!=0);
        print(l2);
}
```

Lists

Keep in mind that the previous methods (except for sublist) do not create a list but an Iterable object. To create a list use .toList() method or .toList(growable:true) for a fix sized list.

```
void main() {
 var l = [1,2,3,4,5];
 var l2 = l.map<int>((i)=>i*i).skip(2).where((i)=>i%2!=0).toList(); //[9,25]
}
```

Dart lists also inherit a .forEach method that can vbe used to quickly iterate through all elements:

```
void List<T>.forEach(void action(T object))
```

```
void main() {
   var l = [1,2,3,4,5];
   int s = 0;
   l.forEach( (i) => s+=i );
   print(s); // 15
}
```

➔ the same result can be achieved via .reduce method

```
void main() {
  var l = [1,2,3,4,5,6];
  print(l.reduce((sum,i)=>sum+=i)); // 21
}
```

Lists

To sort a list use sort method:

```
List<T>.sort([int compareFunction(T object1, T object2)?])
```

Compare function (if present) must return “1” if object1 is bigger than object2, “-1” if object1 is smaller than object2 or “0” if they are equal.

```
void main() {
        var l = ["hello","dart","zzz","alongstring"];
        l.sort();
        print(l); // [alongstring, dart, hello, zzz]
        l.sort((i,j)=>i.length.compareTo(j.length));
        print(l); // [zzz, dart, hello, alongstring]
}
```

Lists

To sort a list use sort method:

```
List<T>.sort([int compareFunction(T object1, T object2)?])
```

Compare function (if present) must return “1” if object1 is bigger than object2, “-1” if object1 is smaller than object2 or “0” if they are equal.

```
void main() {
        var l = ["hello","dart","zzz","alongstring"];
        l.sort();
        print(l); // [alongstring, dart, hello, zzz]
        l.sort((i,j)=>i.length.compareTo(j.length));
        print(l); // [zzz, dart, hello, alongstring]
}
```

Lists

A dart list store references. In the following example, “l.add(t)” will copy a reference to variable “t” into the list, and not a copy of that variable. The solution in this case is to make sure that you make a copy of an object before copying it into a list.

```
class Test {
        int x = 0;
}
void main() {
        var t = Test();
        t.x = 10;
        var l = <Test>[];
        l.add(t);
        t.x = 20;
        print(l[0].x); // 20
}
```

```
class Test {
    int x = 0;
    Test clone() { return Test()..x = x; }
}
void main() {
        var t = Test();
        t.x = 10;
        var l = <Test>[];
        l.add(t.clone());
        t.x = 20;
        print(l[0].x); // 10
}
```

Lists

All command line arguments are sent as a list of strings.

```
void main(List<String> arguments)
{
        print(arguments); // [1, 2, 3, 4]
}
```

To teste the previous program, compile it view “dart.exe compile exe <name>” an them runt the newly obtain executable file like this: “<name>.exe 1 2 3 4”.

There are some specialized modules for parsing command line (similar to what python has). We will discuss more about them when talking about libraries.

Sets

Sets

Sets are containers where each elements is unique. There are several implementations for sets:

- HashSet
- LinkedHashSet
- SplayTreeSet
- Set

Characteristics:

- To create a set use "{...}" to define them.
- Sets are of type ***Iterable*** meaning that will inherit all their properties

Sets

Creating an int set with some values:

```
void main() {
        var s = {1,1,5,6,7,5,3};
        print(s); // {1, 5, 6, 7, 3}
}
```

Creating an empty int set :

```
var s = <int>{}; // type cannot be inferred so it has to be specified
    or
var s = Set<int>();
```

Sets

Constructors / static methods that can be used to create a set:

```
Set<T>.identity()
Set<T>.of(Iterable<T> elements)
Set<T>.from(Iterable elements)
Set<T>.unmodifiable(Iterable elements)
```

```
void main() {
        var s1 = Set<int>.identity(); // s1 = <int>{}
        var s2 = Set<int>.from([1,2,3,1,2,3]); // s2 = <int>{1,2,3}
}
```

Sets

To add an element to a set use add or addAll methods:

```
Set<T>.add(T value)
Set<T>.addAll(Iterable<T> elements)
```

As a difference from List, '+=' can not be used as Sets don't have the operator + defined !

```
void main() {
    var s = {1,2,3};
    s.add(4);
    print(s); // {1,2,3,4}
    s.addAll([1,3,5]);
    print(s); // {1,2,3,4,5}
}
```

Set

To remove an element from a set, the following methods can be used:

```
bool Set<T>.remove(T? object)
void Set<T>.removeAll(Iterable<T>? elements)
void Set<T>.removeWhere(bool removeIfTrueFunction(T object))
void Set<T>.retainWhere(bool retainIfTrueFunction(T object))
void Set<T>.retainsAll(Iterable<T>? elements)
void Set<T>.clear()
```

Use .clear to remove all elements from a set.

The rest of the methods work in a similar manner as they work for a list.

Sets

```
void main() {
        var s = {1,2,3,4,5,6,7,8};
        print(s); // {1, 2, 3, 4, 5, 6, 7, 8}
        s.removeWhere( (e)=>e%2==0 );
        print(s); // {1, 3, 5, 7}
        s.removeAll({1,3,9,10,11});
        print(s); // {5, 7}
}
```

Retain methods work in a similar manner.

Set

Sets specific operations:

```
Set<T>  Set<T>.intersection(Iterable<T?> elements)
Set<T>  Set<T>.difference(Iterable<T?> elements)
Set<T>  Set<T>.union(Iterable<T?> elements)
```

There is no method for asymmetric difference.

```
void main() {
        var s1 = {1,2,3,4};
        var s2 = {3,4,5,6};
        print(s1.intersection(s2));  // {3, 4}
        print(s1.union(s2));         // {1, 2, 3, 4, 5, 6}
        print(s1.difference(s2));    // {1, 2}
        print(s2.difference(s1));    // {5, 6}
}
```

Set

To check if an element is present in a set, use the following methods:

```
T?   Set<T>.lookup(T element)
bool Set<T>.containsAll(Iterable<T?> elements)
bool Set<T>.contains(T? element)
```

```
void main() {
        var s = {1,2,3,4};
        print(s.contains(2));                  // true
        print(s.contains(200));                // false
        print(s.lookup(2));                    // 2
        print(s.lookup(200));                  // null
        print(s.containsAll({1,2,3}));         // true
        print(s.containsAll([1,2,1,2,1,2])); // true
        print(s.containsAll({1,2,4,5}));       // false
}
```

Maps

Maps

Maps are (key,value) containers where each key is unique. There are several implementations for maps:

- HashMap
- LinkedHashMap
- SplayTreeMap
- Map

Characteristics:

- To create a map use "{ ... : ... }" to define them (similar to Python format)
- Maps inherit ***Iterable*** meaning that will inherit all their properties and method

Maps

Creating an <String,int> map with some values:

```
void main() {
        var m = {"Popescu":10,"Ionescu":9,"Georgescu":7};
        print(m); // {Popescu: 10, Ionescu: 9, Georgescu: 7}
}
```

Creating an empty <String,int> map :

```
var s = <String,int>{}; // type cannot be inferred so it has to be specified
    or
var s = Map<String,int>();
```

Maps

Constructors / static methods that can be used to create a map:

```
Map<K,V>.identity()
Map<K,V>.of(Map<K,V> otherMap)
Map<K,V>.from(Map otherMap)
Map<K,V>.unmodifiable(Map otherMap)
```

```
void main() {
        var m1 = Map<String,int>.identity(); // m1 = <String,int>{}
        var m2 = Map<int,int>.from({1:1,2:4}); // m2 = <int,int>{1:1,2:4}
}
```

Maps

Constructors / static methods that can be used to create a map:

```
Map<K,V>.fromEntries(Iterable<MapEntry<K,V>> entries)
Map<K,V>.fromIterables(Iterable<K> keys, Iterable<V> values)
Map<K,V>.fromIterable(Iterable elements, { K key(dynamic elem),
                                           V value(dynamic elem) })
```

A *MapEntry* is similar to pair from STL and is constructed in the following way:

```
MapEntry<K,V>(K key,V value)
```

```
MapEntry(K key,V value)
```

A *MapEntry* has the following properties:

`K MapEntry<K,V>.key`	Returns the key from the (key,value) pair
`V MapEntry<K,V>.value`	Returns the value from the (key,value) pair

Maps

Creating a map using .fromEntries:

```
void main() {
        var m = Map.fromEntries([ MapEntry("Popescu",10),
                                  MapEntry("Georgescu",7),
                                  MapEntry("Ionescu",9) ]);
        print(m); // {Popescu: 10, Georgescu: 7, Ionescu: 9}
}
```

Creating a map using .fromIterables

(OBS: Make sure that keys and values iterables have the same length).

```
void main() {
        var m = Map.fromIterables(["Popescu","Georgescu","Ionescu"],[10,8,6]);
        print(m); // {Popescu: 10, Georgescu: 8, Ionescu: 6}
}
```

Maps

Creating a map using .fromIterable:

```
void main() {
        var list = [
                ["Popescu"  , 10],
                ["Georgescu",  8],
                ["Ionescu“  ,  6]
        ];
        var m = Map.fromIterable(list,
                                 key:(e)=>e[0],      // first entry  = key
                                 value:(e)=>e[1]);   // second entry = value
        print(m); // {Popescu: 10, Georgescu: 8, Ionescu: 6}
}
```

Maps

To add an element to a map use addEntries , addAll methods or [] operator:

```
Map<K,V>.addEntries(Iterable<MapEntry<K,V>> entries)
```

```
Map<K,V>.addAll(Map<K,V> otherMap)
```

```
void main() {
        var m = <String,int>{};
        m["Popescu"] = 9;
        print(m); // {Popescu: 9}
        m.addEntries([MapEntry("Ionescu",10),MapEntry("Georgescu",7)]);
        print(m); // {Popescu: 9, Ionescu: 10, Georgescu: 7}
        m.addAll({"Teodorescu":7});
        print(m); // {Popescu: 9, Ionescu: 10, Georgescu: 7, Teodorescu: 7}
        m["Georgescu"] = 6;
        print(m); // {Popescu: 9, Ionescu: 10, Georgescu: 6, Teodorescu: 7}
}
```

Maps

To update an element to from the map use update , updateAll , putIfAbsetnt methods or [] operator:

```
V    Map<K,V>.putIfAbsent(K key, V ifAbsentFunction() )
V    Map<K,V>.update(K key, V updateFunction(V value), {V ifAbsent()?})
void Map<K,V>.updateAll(V updateFunction(K key, V value))
```

```
void main() {
        var m = {"Pop":10,"Geo":8};
        m.putIfAbsent("Ion",()=>5);   // m = {Pop: 10, Geo: 8, Ion: 5}
        m.update("Pop",(v)=>v-2);     // m = {Pop: 8, Geo: 8, Ion: 5}
        m.update("Ionel",(v)=>10,ifAbsent: ()=>8);  // m = {Pop: 8, Geo: 8,
                                                    //      Ion: 5, Ionel: 8}
        m.updateAll((k,v)=>v-4);      // m = {Pop: 4, Geo: 4, Ion: 1,
                                      //      Ionel: 4}
}
```

Maps

To remove an element from a map, the following methods can be used:

```
bool Map<K,V>.remove(K? key)
```

```
void Map<K,V>.removeWhere(bool removeIfTrueFunction(K key, V value))
```

```
void Map<K,V>.clear()
```

```
void main() {
        var m = {"Raul":10,"Alina":8,"Ion":5,"Georgiana":7};
        print(m); // {Raul: 10, Alina: 8, Ion: 5, Georgiana: 7}
        m.remove("Ion");
        print(m); // {Raul: 10, Alina: 8, Georgiana: 7}
        m.removeWhere((k,v)=>k.length>6);
        print(m); // {Raul: 10, Alina: 8}
}
```

Maps

To check if an element is present in a map, use the following methods:

```
bool Map<K,V>.containsKey(K? key)
```

```
bool Map<K,V>.containsValue(V? value)
```

```
void main() {
        var m = {"Alin":7,"Marilena":8,"Dragos":5};
        print(m.containsKey("Alin"));        // true
        print(m.containsValue(5));           // true
        print(m.containsKey("John"));        // false
        print(m.containsValue(2));           // false
}
```

Maps

A map object has the following properties:

`Itareble<MapView<K,V>>`	`Map<K,V>.entries`	List of (key,value) entries
`Itareble<K>`	`Map<K,V>.keys`	List of key entries
`Iterable<V>`	`Map<K,V>.values`	List of key values

```
void main() {
        var m = {"Alin":7,"Marilena":8,"Dragos":5};
        print(m.values); // (7, 8, 5)
        print(m.keys); // (Alin, Marilena, Dragos)
        for (var i in m.entries) {
                print(i);
        }
}
```

```
MapEntry(Alin: 7)
MapEntry(Marilena: 8)
MapEntry(Dragos: 5)
```

Variable modifiers

Variable modifiers

There are several variable modifiers in Dart that can be used to declare a variable:

- const
- final

A **const** variable will be treated like a value that is determined and evaluated during compile time (similar to what **constexpr** is in C++).

A **final** variable is a variable that can not be changed (similar to **final** in Java or **const** in C/C++). The main difference between a const and a final variable is that const variable must be initialized with a compile-time known value, while a final var could be initialized with any value.

Variable modifiers

Defining a const or final value:

```
void main() {
        const int x = 10;
        const y = 20;
        print(x); // 10
        print(y); // 20
}
```

```
void main() {
        final int x = 10;
        final y = 20;
        print(x); // 10
        print(y); // 20
}
```

A variable can NOT be declared as both **const** and **final**:

```
void main() {
        final const int x = 10; // Error: Members can't be declared to be both
                                // 'const' and 'final'.
}
```

Variable modifiers

A const or a final variable can not be modified:

```
void main() {
   const int x = 10;
   x = 20; // Error: Can't assign
           // to the const variable
           // 'x'.
}
```

```
void main() {
   final int x = 10;
   x = 20; // Error: Can't assign
           // to the final variable
           // 'x'.
}
```

Both **const** and **final** can be use with classes (we will further discuss this when talking about classes). When dealing with object, a const object Is different from a final object because the fields of a final object can be changed, but the fields from a const object can not.

Variable modifiers

Some difference between const and final:

```
import 'dart:math';

void main() {
   final x = Random().nextInt(100);
   print(x);
}
```

```
import 'dart:math';

void main() {
   const x = Random().nextInt(100);
   print(x);
}
```

The second code (the one that uses **const** will not compile, as a random value is not something that can be evaluated at compile time.

Exceptions

Exceptions

Dart exceptions have the following format:

```
try {
  …
}
catch (e) {
  …
}
```

```
try {
  …
}
on <ExceptionType1> [catch (e)] {
  …
}
on <ExceptionType2> [catch (e)] {
  …
}
…
catch (e) {
  …
}
```

```
try {
  …
}
on <ExceptionType1> [catch (e)] {
  …
}
…
catch (e) {
  …
}
finally (e) {
  …
}
```

Exceptions

Example:

```
void main() {
        dynamic x = "Test";
        try {
                x+=2;
        } catch (e) {
                print("Error=>$e");   // Error=>type 'int' is not a subtype of
                                      // type 'String' of 'other'
        }
}
```

Exceptions

Example:

```
void main() {
        dynamic x = "Test";
        try {
                x+=2;
        } catch (e) {
                print("Error=>$e");   // Error=>type 'int' is not a subtype of
                                      // type 'String' of 'other’
        } finally {
                print(“Done");
        }
}
```

Exceptions

To throw and exception use throw keyword:

```
void foo(int x) {
        if (x>10)
                throw "Invalid x = ${x}";
}
void main() {
        try {
                foo(20);
        } catch (e) {
                print("Error = $e"); // Error = Invalid x = 20
        }
}
```

Exceptions

catch can receive 2 arguments (exception and the stack trace):

```
void foo(int x) {
        if (x>10)
                throw "Invalid x = ${x}";
}
void main() {
        try {
                foo(20);
        } catch (e,s) {
                print("Error = $
                print(s);
        }
}
```

```
Invalid x = 20
    at Object.wrapException (<anonymous>:352:17)
    at Object.throwExpression (<anonymous>:366:15)
    at main (<anonymous>:2561:11)
    at <anonymous>:3128:7
    at <anonymous>:3111:7
    at dartProgram (<anonymous>:3122:5)
    at <anonymous>:3130:3
    at replaceJavaScript (https://dartpad.dev/scripts/frame.js:19:19)
    at messageHandler (https://dartpad.dev/scripts/frame.js:91:13)
```

Exceptions

Use rethrow from a catch statement to re-send an exception to the next outer try...catch block.

```
void main() {
        try {
                try {
                        throw "Some exception";
                } catch (e) {
                        print("Error = $e"); // Error = Some exception
                        rethrow;
                }
        } catch (e) {
                print("Second catch"); // Second catch
        }
}
```

Q & A