Dart

1. Variables
   1. var/Object/Dynamic
      1. var varName = value;
      2. var can be of any type but cannot be changed once assigned a certain type.
      3. Object name = value;
      4. Object is used when the type is not declared yet
      5. Object is a root class for all other classes
      6. Object is good for type casting
      7. dynamic imPython = 'a';  
         imPython = 10;
      8. dynamic is almost like var but can be changed to any type once assigned a certain type
   2. null safety
      1. type? name
      2. ex:  
         String? myName;  
         //myName could be null or string  
         String myName2;  
         //myName2 has to be String
      3. without null safety, a variable must be assigned to a value. All null safety variable has default value of null
   3. late keyword
      1. lazy evaluation – value is not available until the variable itself is used
      2. allow non-null safety variable to not be assigned value at the time of declaration
      3. using a late variable that is not initialized will throw an error
      4. ex:  
         late int myInt; //this is legal  
         late int myInt2 = getRandomNumber();  
         //getRandomNumber is never called until there is a reference to myInt2  
         late int lateCheck = infLoop();  
         //assume infLoop will always raise an exception

if (true){//if it were false, it causes error

print('true');

}

else{

print(lateCheck);

}  
//when the whole code is run, there's no problem since infLoop is never called

* 1. final and const
     1. not allowing value change after initialization
     2. const gives instant-time compile
     3. const can also be used to create constant value – not just as declaration
     4. ex:  
        final name = 'bob';  
        const String myName = 'Joe';  
        var f = const [] //f can be changed

1. operator
   1. arithmetic
      1. +
      2. –
      3. \*
      4. /
      5. ~/ (integer division)
      6. %
   2. prefix/postfix increment
      1. --expr expr++
   3. equality
      1. ==
      2. >=
      3. <=
      4. !=
      5. >
      6. <
   4. type test
      1. as 🡪 typecast  
         used to cast an Object into a specific type  
         ex:  
         myCat as Cat
      2. is 🡪 true if the object is of specific type
      3. is! 🡪 is not
      4. ??= 🡪 assign value to a var if the var is null
   5. compound assignment supported
      1. a += b 🡪 a = a + b
   6. logical
      1. ! 🡪 not
      2. == 🡪 equal
      3. != 🡪 not equal
   7. bitwise operator
      1. & 🡪 AND
      2. | 🡪 OR
      3. ^ 🡪 XOR
      4. ~expr 🡪 unary bitwise not
      5. >> 🡪 shift right
      6. << 🡪 shift left
      7. >>> 🡪 unsigned shift right
   8. conditional expr
      1. condition ? exprTrue : exprFalse  
         ex:  
         var name = 1 ? 'Joe' : 'Jane';
      2. expr1 ?? expr2  
         if expr1 is not null, return its value, otherwise evaluate and returns value of expr2
   9. cascading
      1. allow shorthand syntax for accessing members of an object
      2. ex:  
         var human = Human();  
          ..name = 'Joe';  
          ..age = 12;
      3. if a value may be null ? can be used
      4. ex:  
         var girl = Girl();  
          ?..name = 'Jane';//check if girl is null  
          ..age = 12;
   10. other
       1. () 🡪 function call
       2. [] 🡪 subscript member access
       3. ?[] 🡪 check if the object if null; otherwise []
       4. . 🡪 member access
       5. ?. 🡪 conditional member access
       6. ! 🡪 non-null assertion (an object can't be null, otherwise an error is thrown)
2. Comment
   1. //single line
   2. /\*multi  
      lines\*/
3. importing
   1. syntax:  
      import packageName;
   2. ex:  
      import 'package:lib1/lib1.dart' as lib1;  
      import 'package:lib1/lib2.dart' show foo; //only foo from lib2.dart  
      import 'package:lib1/lib3.dart' hide foo; //all but foo from lib3.dart  
      import 'package:lib1/libLazy.dart' deferred;
4. structures in general
   1. all executed code must be inside of main() function

**Built-in types**

1. type records
   1. comma delimited fields enclosed in parenthesis
   2. can be used to define a type
   3. each type can have a name but must be enclosed in curly brackets
   4. to access the member value, use .name or .$position
   5. ex:  
      (int, int) myPair = (1, 2);  
      ({int age, String name}) myPerson = (age: 1, name:'Bob');  
      print('${myPair.$1}');  
      print('Hello ${myPerson.name}');
2. collections
   1. List<type>
      1. denoted using [] square brackets
      2. ex:  
         var myList1 = [];  
         List<int> myIntList = [1, 2, 3, ];  
         List<(int, int)> myPairList = [(1, 2), (2, 3)];  
         //trailing comma at the end is valid
   2. Set<type>
      1. initialization using <type>{} if using var
      2. ex:  
         Set<int> mySet1 = {};  
         var mySet2 = <int>{};
   3. Map<type1, type2>
      1. initialization using Map<type1, type2>() if using var
      2. to access member use myMap[key];
      3. ex:  
         Map<int, String> myMap = {0: 'zeroeth'};  
         myMap[1] = 'first';  
         Map<int, String> myMap = {0: 'zeroeth', };

myMap[1] = 'hello';

String? maybeNull = myMap[3];

maybeNull ??= 'ola';

print(myMap);

print(maybeNull);

* 1. spread operator
     1. use triple dots ...anotherlist
     2. support ? null-aware
     3. ex:  
        List<int> l1 = [4, 5, 6];  
        List<int> l2 = [0, 1, 2, 3, ...l1];

1. generic
   1. <t> is generic
   2. can be used to reduce duplicate
   3. ex:  
      abstract class Cache<T>{  
       //Some code  
      }  
      //T here is a place holder for type

Pattern

1. pattern matching
   1. dart supports pattern matching, similar to Haskell
   2. ex:  
      List<int> myList = [1, 2, 3, 4];

var [a, b, c, d] = myList;

switch(myList){

case [int \_, int \_]:

print('length of two');

case [int \_, int \_, int \_, int \_]:

print('length of 4');

}

print(a);  
//\_ means unnamed var

Function

1. function
   1. syntax:   
      type funcName(parameters){  
      //body  
      }
   2. to use named parameters:  
      type funcName({parameters}){  
      //body  
      }
   3. to use default value  
      type funcName({type param1 = value, ...}){  
      //body  
      }
   4. on named parameters, word required means it must be specified when calling the function
   5. ex:  
      int myNamedParam({int int1 = 0, int? int2}){

if (int2 == null){

return 0;

}

return int1 + int2;

}  
int myReq({required int int1}){

return int1;

}

OOP

1. Class
   1. syntax:  
      class ClassName{  
       //attributes  
       //constructor – optional  
       //methods  
      }
   2. attributes have initial value – if not specified by constructor when called
   3. ex:  
      class MyClass {

String? name;

int? age;  
 String? child;//child is always null by default

MyClass(String name, int age){

this.name = name;

this.age = age;

}

}

* 1. class could use generic
  2. ex:  
     class ListOfSomething<T> {

List<T> myList = [];

int length = 0;

String? desc;

T? getLastItem() {

if (length == 0) {

return null;

}

this.length -= 1;

return myList.removeAt(length);

}

void add(T item) {

this.myList.add(item);

this.length += 1;

}

void append(List<T> items) {

this.myList.addAll(items);

this.length += items.length;

}

ListOfSomething({required String desc, List<T> myList = const []}) {

this.desc = desc;

this.myList = myList;

this.length = myList.length;

}

}

1. OOP visibility
   1. static
      1. allow an attribute to be callable by class name instead of object
      2. ex:  
         //inside a class  
         class someClass{  
          static final int numOfInstance = 0;  
         }  
         //calling  
         int main(){  
          print(someClass.numOfInstance);  
         }
   2. private
      1. Limit the visibility of an attribute – can only be accessed within the same package.
      2. ex: when a class is imported, the importer cannot access private attribute, but when the caller is within the same file/package, it's accessible
      3. syntax:  
         type \_name = value;
2. Inheritance – class
   1. a class that supports another class and inherits the implementation
   2. constructor of parent class should be called by the child constructor
   3. a class can extends up to 1 classes
   4. ex: if class A extends Class B, class A can access all methods and attributes (depending on the visibility) from class B without having to override the methods.
3. Inheritance – Interface
   1. a class that supports another class but does not inherit the implementation – only having the members but doesn't have idea what they do
   2. constructor of parent class does not belong in the interface
   3. a class can implements more than 1 interfaces
   4. ex: if class A implements class B and class B has attributes of x,y,z and methods of f1 and f2, class A must have the same attributes and methods declared inside.
4. inheritance – mixin
   1. allow multiple use in multiple hierarchies
5. constructor
   1. constructor is a function
   2. support generative constructor
   3. a class may have more than 1 constructor
   4. example above is equivalent to below example:  
      class MyClass {

String? name;

int? age;  
 String? child;

MyClass(this.name, this.age);  
 MyClass.alt(){  
 this.name = 'Joe';  
 this.age = 0;  
 }

}