DART Language

COURSE 3 (REV 5)

GAVRILUT DRAGOS

Agenda

- **Classes**
  General observations
- **Data members**
  const, static, null-safety
- **Constructors**
  Name, un-named, constant, factories
- **Properties**
  Getters and Setters
- **Methods**
  Overriding, static methods
- **Operators**
  Dynamic types
- **Inheritance**
  Mixins, abstract classes

Classes

Classes

Dart provides several mechanisms that can be used to create an object (an instance of a class).

The way a class is defined in Dart is a mix of properties derived from both Java, C++ and C# that include getter / setter, proxy constructors, operators, etc.

To create a simple class ➔ use the class keyword. All classes are implicitly derived from Object (meaning that each class has the following methods/properties:

`int hashCode`	`A hash code for this object`
`Type runtimeType`	`Type of the object`
`dynamic noSuchMethod(Invocation i)`	`Call whenever a non-existing property is called`
`String toString()`	`A string representation for current object`

Classes

A very simple example:

```
class Test
{
        // an empty class
}
void main() {
        var t = Test();
        print(t.runtimeType); // Test
        print(t.hashCode);    // 7552.....
        print(t.toString());  // Instance of 'Test’
}
```

Classes

A DART class consists in

- One or more constructors (including named constructors)
- Operators
- Getter and Setters (for properties)
- Data members (including static data members)
- Methods

A DART class does not have a destructor (this is not required as DART garbage collector takes care of this cases).

Data members

Data members

Defining a data member in a class can be done in the following way

```
class Test  {
        var v1[,v2,v3,… vn];               // one or multiple variables defined
}
```

Or using their type:

```
class Test {
        <type> v1[,v2,v3,… vn];            // one or multiple variables defined
}
```

The initial value of a data member can be specified at this point

```
class Test {
        <type or var> v = <value>;    // a data member with a value
}
```

Data Members

In this case, “y” type will be inferred, however “x” type will be a dynamic type (similar to what python has). As a result variable “x” can be changed (in terms of its type during runtime).

```
class Test {
	var x;
	var y = 10; // type will be inferred
}
void main() {
	var t = Test();
	print(t.x.runtimeType); // Null
	print(t.y.runtimeType); // int
	t.x = 10;
	print(t.x.runtimeType); // int
	t.x = "bla bla bla";
	print(t.x.runtimeType); // String
}
```

Data Members

However, the same logic does not work for inferred types. For example, “y” type is considered “int” and can not be changed during runtime.

```
class Test {
        var x;
        var y = 10; // type will be inferred as int
}
void main() {
        var t = Test();
        t.y = "bla bla bla";  // Error: A value of type 'String' can't be
                              // assigned to a variable of type 'int'.
}
```

Data Members

All data members (except for generic ones like the ones defined using var keyword must be initialized). To avoid this either initialize that data member or declare it using “?” specifier (this will set its value to null if not defined).

```
class Test {
   int x; // Field 'x' should be initialized because its type 'int’ doesn’t
          // allow null.
}
```

Data Members

All data members (except for generic ones like the ones defined using var keyword must be initialized). To avoid this either initialize that data member or declare it using “?” specifier (this will set its value to null if not defined).

```
class Test {
        int? x;
        int? y = 20;
        int z = 30;
}
void main() {
        var t = Test();
        print(t.x); // null
        print(t.y); // 20
        print(t.z); // 30
}
```

Data Members

All data member can be declared as read-only by using the final keyword. In this case, its value can’t be modified after its initialization.

```dart
class Test {
   final int x = 30;
}
void main() {
   var t = Test();
   print(t.x); // 30
   t.x = 40; // Error: The setter 'x' isn't defined for the class 'Test'.
}
```

In this case, the type of Dart requires a setter / getter (we will discuss more about this cases in the next slides).

Data Members

Dart also allows static data members (that belong to the class and not the instance). They can be defined by using the static keywork.

```dart
class Test {
        static int x = 30;
}
void main() {
        print(Test.x); // 30
}
```

As a difference between Dart and C++, a static member can not be accessed via an instance !

```dart
void main() {
   var t = Test();
   print(t.x); // Error: The getter 'x' isn't defined for the class 'Test'.
}
```

Data Members

A static member can however be defined as a static const (similar to **constexpr** format from C++). This will produce some compile time optimization where that variable will be replaced with its value.

```
class Test {
        static const int x = 30;
}
void main() {
        var t = Test();
        print(Test.x); // 30
}
```

Data Members

Dart does not have a private/public/protected concept similar to C++/Java. However, it does have some visibility limitation at the library level. To hide/restrict access to a data member outside its definition library, use the underscore (_) when defining its name.

KEEP IN MIND that this only limits the visibility outside its library. The next code will work as main function is defined in the same library (application) as class Test.

```
class Test {
       int _x = 30;
}
void main() {
       var t = Test();
       print(Test._x); // 30
}
```

Data Members

Use ?. operator to access a data member and check if the class was defined.

```
class Test {
        int x = 30;
}
void main() {
        Test? t;
        t?.x = 10;
}
```

As a general concept, Dart does not allow to create a null instance of an object (except from using it with “?” operator).

Constructors

Constructors

Constructors are instance specific methods that are called whenever an instance is created. A constructor is created in the following way:

```
class <class_name>  {
        <class_name>([v1,v2,v3,… vn]); // unnamed ctor
        <class_name.function_name>([v1,v2,v3,… vn]); // named ctor
}
```

As a difference from C++, there can only be only one unnamed constructor (meaning that even if we define multiple unnamed constructors with different parameters, Dart will produce a compile error and not allow this).

Constructors

Data members can be instantiated in the constructor (similar to how C++ allows it by adding : and then a list of data member = value after this

```
class <class_name>  {
       data_type d1,d2,d3… dn;
       <class_name>([v1,v2,v3,… vn]): d1=value1, d2=value2,… dn=valuen, { … }
}
```

```
class Test {
       int x,y;
       Test(): x=10, y = 20 { }
}
void main() {
       var t = Test();
       print("x=${t.x}, y=${t.y}"); // x=10, y=20
}
```

Constructors

It is also possible to initialize a variable directly and as part of the constructor. This will allow one to create some sort of default value that can be used if some constructors do not have an initialization code.

In this cases, the constructor code will take precedence and decide the value of a field.

```
class Test {
        int x=1;
        Test(): x=2 {}
}
void main() {
        var t = Test();
        print(t.x); // 2
}
```

Constructors

Multiple un-names constructors are not allowed:

```
class Test {
        int x;
        Test(): x = 10
        {
        }
        Test(int value): x = value
        {
        } // Error: 'Test' is already declared in this scope.
}
```

In this case, there is already an un-names constructor (with no parameters) so another one is not allowed.

Constructors

Event if a data member value can be changed in the constructor, its value MUST be instantiated before (after the constructor or at its definition).

```
class Test {
        int x; // Error: Field 'x' should be initialized because its type
               // 'int' doesn't allow null.
        Test() { x = 10; }
}
```

However, the following code will work:

```
class Test {
        int x = 2;
        Test() { x = 10; }
}
```

Constructors

Since un-names constructors are used to initialize data members, Dart allows some syntax sugar that specifies the data member that is initialized (via this keyword).

```
class Test {
        int x,y;
        Test(this.x, this.y);
}
void main() {
        var t = Test(10,20);
        print("x=${t.x}, y=${t.y}"); // x=10, y=20
}
```

In this case, this constructor is equivalent to:

```
Test(int value_x, int value_y): x=value_x, y = value_y { }
```

Constructors

If a constructor has an empty body { } there is a syntax sugar form that can be used (by adding ; after the constructor definition).

```
class Test {
        int x,y;
        Test(): x=10, y = 20 { }
}
```

is equivalent to

```
class Test {
        int x,y;
        Test(): x=10, y = 20;
}
```

Constructors

When initializing data members, this / other data members can not be used as part of the initialization expression. This is different that what C++ allows.

```
class Test {
       int x,y;
       Test(int value):
              y=value,
              x=this.y*this.y;      // Can't access 'this' in a field
                                    // initializer.
}
```

In this case, even if “this.y” is already instantiated and therefor x = this.y * this.y can be computed, Dart does not allow this.

Constructors

Dart classes may have multiple named constructors, as long as they have different names:

```
class Test {
        int x,y;
        Test.empty(): x=0,y=0 {}
        Test.withX(int value): x=value,y=0;
        Test.withXY(this.x, this.y);
}
void main() {
        var t = Test.empty();
        print("x=${t.x}, y=${t.y}"); // x=0, y=0
        t = Test.withX(10);
        print("x=${t.x}, y=${t.y}"); // x=10, y=0
        t = Test.withXY(1,2);
        print("x=${t.x}, y=${t.y}"); // x=1, y=2
}
```

Constructors

Just like C++, a constructor may proxy the initialization to another constructor (by using this keyword):

```
class Test {
        int x,y;
        Test(int value): y=value,x=value;
        Test.empty(): this(0);
}
void main() {
        var t = Test.empty();
        print("x=${t.x}, y=${t.y}"); // x=0, y=0
}
```

Constructors

Dart also have some compile time protections for cyclic redirections when using constructors.

```
class Test {
       int x,y;

       Test.ctor_1(): this.ctor_2();
       Test.ctor_2(): this.ctor_3();
       Test.ctor_3(): this.ctor_1();       // Error: Redirecting
                                           // constructors can't be cyclic.
}
```

Constructors

In case of objects that will not be modified during the execution, Dart allows creation of a constant constructor by adding const keyword in front of the definition. This will create a compile-time constant object. For this to be possible, data members from that object must be declared as final.

```
class Test {
        final int x,y;
        const Test.constantObject(): x=0,y=0;
}
void main() {
        var t = Test.constantObject();
}
```

Constructors

As a general concept, a constructor can not be static. A class however can have static methods (that work like a factory → for example in case of a Singleton pattern). In particular for Dart, there is a special keyword factory that allows one to create a named constructor that is static (does not have access to this , but returns an instance of the object).

```
class Singleton {
        static Singleton obj = Singleton(10);
        int x,y;
        Singleton(int value): x=value, y=value;
        factory Singleton.sameObject() { return obj; }
}
void main() {
        var t1 = Singleton.sameObject();
        var t2 = Singleton.sameObject();
        print(t1==t2); // true
}
```

Properties

Properties

In reality, Dart automatically creates two methods (a getter and a setter) for each data member (there are some exceptions → e.g. variable defined as final that do not have the setter method define → hence the error when trying to change such a variable: ***The setter '<variable name>' isn't defined for the class '<class name>'.***

```
class <class_name>  {
        <data_type> get <variable_name> { … return value; }
        set(<data_type> value) { … }
}
```

Dart provides 2 keywords: get and set (much like C#) that allow creating a setter and a getter for a data member.

Properties

A simple example:

```
class Grades {
        int mathGrade, englishGrade;
        Grades(this.mathGrade, this.englishGrade);
        int get average { return (mathGrade+englishGrade) >> 1; }
        set average(int value) { mathGrade = englishGrade = value; }
}
void main() {
        var st = Grades(10,8);
        print("Average = ${st.average}"); // Average = 9
        st.average = 10;
        print("Math=${st.mathGrade}, English=${st.englishGrade}");
        // Math=10, English=10
}
```

Properties

For simplicity, => operator can be used

```
class Grades {
       int mathGrade, englishGrade;
       Grades(this.mathGrade, this.englishGrade);
       int get average => (mathGrade+englishGrade) >> 1;
       set average(int value) => mathGrade = englishGrade = value;
}
void main() {
       var st = Grades(10,8);
       print("Average = ${st.average}"); // Average = 9
       st.average = 10;
       print("Math=${st.mathGrade}, English=${st.englishGrade}");
       // Math=10, English=10
}
```

Properties

To create a **read-only property**, only the get method should be defined. The following code will not compile !

```
class Grades {
        int mathGrade, englishGrade;
        Grades(this.mathGrade, this.englishGrade);
        int get average => (mathGrade+englishGrade) >> 1;
}
void main() {
        var st = Grades(10,8);
        print("Average = ${st.average}"); // Average = 9
        st.average = 10;        // The setter 'average' isn't defined for
                                // the class 'Grades'.
}
```

Properties

Getters and setters are often created to allow access to a private variable (from outside its library).

```
class Test {
        int _x = 0; // private (only visible within the library)
        set x(int value) => _x=value*2;
        int get x => _x;
}
void main() {
        var t = Test();
        t.x = 5;
        print(t.x); // 10
}
```

Properties

It is also possible to define only a setter, or to create a different setter and getter that access the same data member in a different way !

```
class Distance {
        int _m = 0; // the actual distance is store in meters
        set km(int value) => _m=value*1000;
        int get length => _m;
}
void main() {
        var t = Distance();
        t.km = 5;
        print(t.length); // 5000
}
```

Properties

Unary operators (such as ++) will trigger both the getter and the setter for a specific data member.

```
class Test {
        int _x = 0;
        set x(int value) {
                _x = value;
                print("Setter (x=${value})");
        }
        int get x {
                print("Getter (x=${_x})");
                return _x;
        }
}
void main() {
        var t = Test(); t.x++; // Getter (x=0) , Setter (x=1)
}
```

Methods

Methods

Methods are functions defined within a class that can access data members and act as an interface for producing different actions within a class instance.

```
class <class_name>  {
        <data_type> function_name ([v1,v2,v3,… vn]) { … }
        <data_type> function_name ([v1,v2,v3,… vn]) => <return value>;
}
```

Just like a regular function, a method has two forms (a standard one and a simplified one based on the => operator).

Methods

A simple example:

```
class Test {
        int x = 9;
        bool isOdd() {
                return x%2==0;
        }
        bool isEven() => x%2==1;
}
void main() {
        var t = Test();
        print(t.isOdd());     // false
        print(t.isEven());    // true
}
```

Methods

Dart methods DO NOT support overloading (from this point of view, Dart is more similar to Python than C++ or Java). As such, the following code will NOT compile.

```
class Test {
        int x = 9;
        int Add(int value) => x+value;
        int Add(int v1, int v2) => x+v1+v2;  // Error: 'Add' is already
                                             // declared in this scope.
}
```

And just like in Python, the solution is to create one function with multiple default / named parameters that can be called upon execution.

Methods

A possible solution for the previous code could look like this:

```
class Test {
        int x = 9;
        int Add(int v1, [int? v2]) => v2!=null ? x+v1+v2 : x+v1;
}
void main() {
        var t = Test();
        print(t.Add(5));       // 14
        print(t.Add(6,7));     // 22
}
```

In this case, v2 is a default variable that can be null if nor specified (because of its “?” symbol added after its type: int).

Methods

Methods in Dart can be overridden. Just like in Java, it is recommended to add a specific annotation @override in front of all overridden methods. Since all Dart classes are derived from object, a class can easily override toString to provide a string representation of itself.

```
class Test {
        int x;
        Test(this.x);
        @override
        String toString() => "Test with x = ${x}";
}
void main() {
        var t = Test(10);
        print(t); // Test with x = 10
}
```

Methods

Similarly, noSuchMethod can be overridden. However, that method will only work for dynamic types, meaning that for every dynamic variable, if a non-defined method or variable is accessed, *noSuchMethod* will be called instead !

```
class Test {
   int x;
   Test(this.x);
   @override
   void noSuchMethod(Invocation i) {
      if ((i.isMethod) && (i.memberName.toString().contains("write")))
         print("Test object with x = $x");
   }
}
void main() {
       dynamic t = Test(10); t.write(); }
```

Methods

The following code will NOT compile, because "t" is not a dynamic type, but a type Test inferred from its definition.

```
class Test {
        int x;
        Test(this.x);
        @override
        void noSuchMethod(Invocation i) { }
}
void main() {
        var t = Test(10);
        t.write();      // Error: The method 'write' isn't defined for the
                        // class 'Test'
}
```

Methods

Dart also supports static methods (specific to the class and not the instance). To create a static method, add static keyword in front of a method definition. A static method can only be called by the class and not by an instance and does not have access to this pointer.

```
class Test {
       int x = 10;
       static String getMyName() => "Test class";
}
void main() {
       print(Test.getMyName()); // Test class
}
```

```
var t = Test();
print(t.getMyName());        // Error: The method 'getMyName' isn't defined
                             // for the class 'Test'.
```

Operators

Operators

Operators are defined in a similar manner as with C++ language (by using a special keyword operator to identify a special method that will treat a specific operation).

```
class <class_name>  {
   <data_type> operator <oprator_type> ([v1,v2,v3,… vn]) { … }
   <data_type> operator <oprator_type> ([v1,v2,v3,… vn]) => <return value>
}
```

Supported operators are:

<	>	<=	>=	==
+	-	*	/	%
~/	^	&	|	~
>>	>>>	<<	[]	[]=

Operators

Some operators are not needed (for example != is similar to !(==) and as such there is no need for a specific overwrite in this case).

```
class Number {
        int x;
        Number(this.x);
        Number operator+ (Number n) => Number(x+n.x);
}
void main() {
        var n1 = Number(10);
        var n2 = Number(20);
        var n3 = n1+n2;
        print(n3.x); // 30
}
```

Operators

Just like in the case of method, operator overloading is not possible in Dart. As such, the following code will not compile:

```
class Number {
        int x;
        Number(this.x);
        Number operator+ (Number n) => Number(x+n.x);
        Number operator+ (int n) => Number(x+n);    // Error: '+' is already
                                                     // declared in this
                                                     // scope.
}
```

The main difference is that if method overloading can easily be replaced by using multiple methods with different name, the case of operator overloading is a little bit more complicated.

Operators

One way of solving operator overloading problem is to use a dynamic type:

```
class Number {
        int x;
        Number(this.x);
        Number operator+ (dynamic value) {
                if (value is Number) return Number(x+(value as Number).x);
                if (value is int) return Number(x+(value as int));
                return Number(x);
        }
}
void main() {
        print((Number(10)+20).x);               // 30
        print((Number(100)+Number(200)).x);  // 300
}
```

Inheritance

Inheritance

Inheritance concepts in Dart are:

- Class extension (similar to Java, using the keyword extends)
- Interfaces (a concept between Java and C++, realized using the keyword implements)
- Mixin (a way to extend an existing class with new functionality from another class).

Dart does not support multiple inheritance but can achieve similar functionality through interfaces and mixins.

To extend a class use extends after the class name and super keyword to access base class methods and data members.

```
class <class_name> extends <base_class> implements <interface1, interface2,…>
{
   . . .
}
```

Inheritance

A simple example of inheritance → class Derived has two variables "x" and "y" and one method GetX().

```
class Base {
        int x = 10;
        int GetX() => x*x;
}
class Derived extends Base {
        int y = 20;
}
void main() {
        var d = Derived();
        print("${d.y}, ${d.x}, ${d.GetX()}"); // 20, 10, 100
}
```

Inheritance

When deriving from a base class, both methods and data member can be overridden. It is still possible to access the base data member by using super keyword from a property/method.

```
class Base {
        int x = 10;
}
class Derived extends Base {
        int x = 20;
        int get baseX => super.x;
}
void main() {
        var d = Derived();
        print("x=${d.x}, base_x=${d.baseX}"); // x=20, base_x=10
}
```

Inheritance

When overriding a data member, make sure that the same type is used otherwise a compile error will be triggered.

```
class Base {
        int x = 10;
}
class Derived extends Base {
        double x = 20.5;        // Error: The return type of the method
                                // 'Derived.x' is 'double', which does not
                                // match the return type, 'int', of the
                                // overridden method, 'Base.x’.
}
```

It is also recommended to use annotation (@override) to explain the compiler that the data member was overridden on-purpose and not by mistake.

Inheritance

Method overridden works in a similar manner

```
class Base {
       int sum(int x, int y) => x+y;
}
class Derived extends Base {
       @override
       int sum(int x, int y) => x*y;
}
void main() {
       var d = Derived();
       print(d.sum(3,4)); // 12
}
```

Inheritance

And just like in the previous case, the method that is being overridden must have the same name, parameters and return type as the one from the base class.

```
class Base {
        int sum(int x, int y) => x+y;
}
class Derived extends Base {
        @override
        int sum(int x, int y, int z) => x+y+z;      // Error: The method
                                                    // 'Derived.sum' has more
                                                    // required arguments
                                                    // than those of
                                                    // overridden method
                                                    // 'Base.sum’.
}
```

Inheritance

Methods in Dart are virtual by nature (overriding one will change the behavior even if casting to the base class).

```
class Base {
       int op(int x, int y) => x+y;
}
class Derived extends Base {
       @override
       int op(int x, int y) => x*y;
}
int op(Base b, int x, int y) => b.op(x,y);
void main() {
       var d = Derived();
       print(op(d,3,4)); // 12
}
```

Inheritance

Keep in mind that final keyword in Dart does not have the same meaning as it has in Java (from this regard Dart is more close to C++ than to Java).

```
final class Base {
        int op(int x, int y) => x+y;
}
class Derived extends Base {
        @override
        int op(int x, int y) => x*y;
}
```

The following code will not compile (final is considered an invalid keyword to be used with class specifier).

Inheritance

Dart allows creation of abstract classes (NOT interfaces) that can be used to enforce overriding some methods. An abstract method can only be defined within an abstract class, but it is different from a regular method as it has ; operator at the end of its method definition.

```
abstract class Form {
        String getName() ;
}
class Circle extends Form {
        // The non-abstract class 'Circle' is missing implementations for
        // these members: - Form.getName
}
```

In this case, class Circle is not complete because it does not implement the abstract method getName from the class Form.

To create an abstract class, prefix its definition with abstract keyword.

Inheritance

At the same time, an abstract class can have non-abstract methods / data-members or properties.

```
abstract class Form {
       String getName();              // abstract method
       bool isForm() => true;         // non-abstract method
}
class Circle extends Form {
       @override
       String getName() => "Circle";
}
void main() {
       var c = Circle();
       print("Name=${c.getName()}, IsForm=${c.isForm()}"); // Name=Circle,
                                                            // IsForm=true
}
```

Inheritance

Abstract classes and methods are not necessary (but are very useful for code clarity). The previous example can be re-written without abstract classes in the following way.

```
class Form {
       String getName() => "Form";
       bool isForm() => true;
}
class Circle extends Form {
       @override
       String getName() => "Circle";
}
void main() {
       var c = Circle();
       print("Name=${c.getName()}, IsForm=${c.isForm()}"); // Name=Circle,
                                                           // IsForm=true
}
```

Inheritance

Abstract classes can have abstract properties.

Any data-member defined in the abstract class will be inherit in the derived class as well.

```
abstract class Form {
        String get name; // abstract property
        int x = 10;
}
class Circle extends Form {
        @override
        String get name => "Circle";
}
void main() {
        var c = Circle();
        print("${c.name}, ${c.x}"); // Circle, 10
}
```

Inheritance

A class cand also implement an interface (methods and properties) from another class (BUT no data members). The previous code will not compile if Circle implements Form !

```
abstract class Form {
        String get name; // abstract property
        int x = 10;
}
class Circle implements Form {
        @override
        String get name => "Circle";
}
void main() {
        var c = Circle();
        print("${c.x}"); // Error: The non-abstract class 'Circle' is
                        // missing implementations for these members: Form.x
}
```

Inheritance

A class can implement multiple interfaces:

```
abstract class Form { int get area; }
abstract class Name { String get name; }

class Circle implements Form, Name
{
        @override
        String get name => "Circle";
        @override
        int get area => 10;
}
void main() {
        var c = Circle();
        print("${c.name}, ${c.area}"); // Circle, 10
}
```

Inheritance

Any class (abstract or not) can be an interface for another class.

```
class Form { int get area => 0; }
class Name { String get name => "Form"; }

class Circle implements Form, Name
{
        @override
        String get name => "Circle";
        @override
        int get area => 10;
}
void main() {
        var c = Circle();
        print("${c.name}, ${c.area}"); // Circle, 10
}
```

Inheritance

When using multiple implements make sure that you don't have similar functions (by name and different parameters) that you are implementing.

```
abstract class Interface1 { int foo(int x); }
abstract class Interface2 { int foo(int x, int y); }
class Test implements Interface1, Interface2 {
       @override int foo(int x) => 2;
       @override int foo(int x, int y) => 2; // Error: 'foo' is already
                                             // declared in this scope.
}
void main() {
       var t = Test();
       print(t.foo);  // Error: Can't use 'foo' because it is declared more
                      // than once.
}
```

Inheritance

However, if two abstract methods are identical (same name, same return value, same parameters), implementing only one will suffice and the code will compile. The same logic applies for properties (different interfaces with the same property (name and type) can co-exist).

```
abstract class Interface1 { int foo(int x); }
abstract class Interface2 { int foo(int x); }
class Test implements Interface1, Interface2 {
	@override
	int foo(int x) => 2;
}
void main() {
	var t = Test();
	print(t.foo(10)); // 2
}
```

Inheritance

Sometimes it is required that the derived class implements a function slightly different (but with the same logic → e.g. a parameter that is derived from the one used in the base definition).

```
class Form { }
class Rectangle extends Form { }
abstract class Calculator { int Compute(Form f); }
class AreaCalculator implements Calculator {
        @override int Compute(Rectangle r) => 0;
        // Error: The parameter 'r' of the method 'AreaCalculator.Compute’
        // has type 'Rectangle', which does not match the corresponding type,
        // 'Form', in the overridden method, 'Calculator.Compute’.
}
void main() {
        var ac = AreaCalculator();
        print(ac.Compute(Rectangle()));
}
```

Inheritance

The solution in this case is to use the keywork covariant and allow other parameters to be used as part of the virtual methods, as long as they are derived from the base class (in this case, it has to be a parameter derived from Form class → e.g. Rectangle).

```
class Form { }
class Rectangle extends Form { }
abstract class Calculator { int Compute( covariant Form f); }
class AreaCalculator implements Calculator {
       @override
       int Compute(Rectangle r) => 0;
}
void main() {
       var ac = AreaCalculator();
       print(ac.Compute(Rectangle()));
}
```

Inheritance

Dynamic polymorphism can easily be achieved using implements keyword

```
abstract class Form { String get name; }

class Circle implements Form { String get name => "Circle"; }
class Square implements Form { String get name => "Square"; }
class Rectangle implements Form { String get name => "Rectangle"; }

void main() {
       var l = <Form>[Circle(), Square(), Rectangle()];
       l.forEach((e)=>print(e.name));      // Circle
                                           // Square
                                           // Rectangle
}
```

Inheritance

The same example will work in a similar way if we use extends and a non-abstract class as a base class, or if we combine those two cases (abstract class with extends keyword or regular class with implements keyword).

```
class Form { String get name => "Form"; }
class Circle extends Form { String get name => "Circle"; }
class Square extends Form { String get name => "Square"; }
class Rectangle extends Form { String get name => "Rectangle"; }

void main() {
       var l = <Form>[Circle(), Square(), Rectangle()];
       l.forEach((e)=>print(e.name));      // Circle
                                           // Square
                                           // Rectangle
}
```

Inheritance

extends and implements keywords can coexist when defining a derived class:

```
class Form {
        int x = 100;
}
abstract class Name {
        String get name;
}
class Circle extends Form implements Name {
        @override
        String get name => "Circle";
}
void main() {
        var c = Circle();
        print("${c.name}, ${c.x}"); // Circle, 100
}
```

Inheritance

When using implements keyword, all methods and properties from the inherit class MUST be implemented (even if they are already implemented in the base class). This is important as it differentiate an interface from a mixin.

```
class Interface {
        int sum(int x, int y) => x+y;
}
class my_int implements Interface {
        // Error: The non-abstract class 'my_int' is missing implementations
        // for these members: - Interface.sum
}
void main() {
        var i = my_int();
        print(i.sum(10,20));
}
```

Inheritance

To overcome previous limitation, Dart has introduced **mixins** (a class that has some methods defined that will be available as they are defined in the base class). To create a mixin use the mixin keyword and with keyword when deriving the new class.

```
mixin SumImplementation {
       int sum(int x, int y) => x+y;
}
class my_int with SumImplementation {
}
void main() {
       var i = my_int();
       print(i.sum(10,20)); // 30
}
```

Inheritance

Using the keyword mixin is not mandatory. A class can be derived using with keyword from another class as well (not necessary a mixin). However, if mixin keyword is used, that class can not be instantiated.

```
class SumImplementation {
       int sum(int x, int y) => x+y;
}
class my_int with SumImplementation {
}
void main() {
       var i = my_int();
       print(i.sum(10,20)); // 30
}
```

Inheritance

While not necessary, methods from a mixin can be overridden and they work just like every other virtual method. However, even if possible, this is not the intended usage for a mixin.

```
mixin SumImplementation {
        int sum(int x, int y) => x+y;
}
class my_int with SumImplementation {
        int sum(int x, int y) => x*y;
}
void main() {
        var i = my_int();
        print(i.sum(10,20)); // 200
        print((i as SumImplementation).sum(10,20)); // 200
}
```

Inheritance

Multiple mixins can be used at the same time. They need to respect the same rules as the ones used when using implements (e.g. no similar names with different parameters or return type).

```
mixin SumImplementation {
       int sum(int x, int y) => x+y;
}
mixin MulImplementation {
       int mul(int x, int y) => x*y;
}
class my_int with SumImplementation, MulImplementation { }
void main() {
       var i = my_int();
       print(i.sum(10,20)); // 30
       print(i.mul(10,20)); // 200
}
```

Inheritance

Another difference between using with and implements keywords is that data members are going to be present in the derived type (just like in the case of extends).

```
mixin SumImplementation {
       int sum(int x, int y) => x+y;
       int v = 10;
}
class my_int with SumImplementation {
}
void main() {
       var i = my_int();
       print(i.sum(10,20));  // 30
       print(i.v);           // 10
}
```

Inheritance

It is possible to derive from multiple mixins with the same data member. In this case, the last derivation will set the value for that variable in the derived class.

```
mixin M1 { int v = 10; }
mixin M2 { int v = 20; }
class my_int with M1, M2 {
}
void main() {
        var i = my_int();
        print(i.v); // 20
}
```

```
mixin M1 { int v = 10; }
mixin M2 { int v = 20; }
class my_int with M2, M1 {
}
void main() {
        var i = my_int();
        print(i.v); // 10
}
```

Inheritance

It is not possible to use two mixins with the same data member but of different types. If this is the case, a compile error will be triggered.

```
mixin M1 { int v = 10; }
mixin M2 { double v = 20.0; }
class my_int with M1, M2 {
  // Error: Applying the mixin 'M2' to 'Object with M1' introduces an
  // erroneous override of 'v’.
}
void main() {
       var i = my_int();
       print(i.v); // 20
}
```

Inheritance

A mixin can be restricted to be applied to a specific type or its descendants by using the keyword on upon definition. In the next example, only a class derived from Number can use the Sum mixin. As such, my_string will produce a compile error.

```
class Number { }
mixin Sum on Number { int sum(x,y) => x+y; }
class my_int extends Number with Sum { }
class my_string with Sum {
       // Error: 'Object' doesn't implement 'Number' so it can't be used with
       // 'Sum’.
}
void main() {
       var i = my_int();
       print(i.sum(1,2)); // 3
}
```

Inheritance

The following table presents an overview on what gets inherit for extends , implements and with keywords.

	extends	implements	with
Data members	Yes, will be inherit in the derived class	Will NOT be inherit in the derived class	Yes, will be inherit in the derived class
Methods	Can be overridden, but it is not required	MUST be overriden	Can be overridden, but it is not required
Properties	Can be overridden, but it is not required	MUST be overriden	Can be overridden, but it is not required
Abstract methods		Virtual, MUST be overridden	
Abstract properties		Virtual, MUST be overridden	
Multiple inheritance	NO	Yes	Yes

Inheritance

What to use and when:

- extends ➔ whenever you want to extend an existing class with new functionalities (but also keep and reuse the existing ones).
- implements ➔ whenever there is a common interface between multiple classes (a fix set of properties that all objects that have a common ancestor have).
- with ➔ whenever, there is some already implemented code (methods/properties) that need to be reused, but is not a general characteristic for all object with a specific ancestor

Q & A