Enums for Dart

We plan to introduce enumerated types after the 1.0 release. An enumerated type has the form:

enum E {id_1, … id_n}

It is a top level declaration, and introduces a class E into the surrounding scope. E may not be subclassed, mixed in, implemented or instantiated. There are exactly *n* instances of E, available via the static constants id_1, .., id_n. We may want to refine this so that enums support methods like >, >=, <, <=.

We expect to support iteration over enum values

```
for (var e in E.values) { print(e.index); }
```

Where values is a list of all the enum instances, in the order they were declared.

Each enum instance has an ordinal value determined by its position in the sequence id_1, … id_n. The value can be retrieved via a getter index. It is thus possible to convert an integer into an enum of the corresponding ordinal value by indexing the list of values of the enum type

E.values[n] // should return the n'th enum

It should be possible to switch over enums, and we plan to issue warnings if the switch does not cover the entire set of enum values. Each enum value has a getter index that provides its ordinal.

This is almost like sugar for

```
class E /* implements Comparable<E>? */ {
 final int index;
 final String _name;
 const E(this.index, this._name);
 String toString() => '$E.$_name';
static const E id1 = const E(0, 'id1');
```

...
static const E id_n = **const** E(n-1, 'id_{n-1}');

static const List<E> values = **const** <E>[id_1 … id_n];
// equality and relational operators?
}

except for the following differences:

- The field _name cannot be accessed.
- There is additional support in switch.
- E can not be instantiated or used as a mixin, superinterface or superclass.

Proposed Specification

We might choose to modify the sections on inheritance and import/export to localize or repeat the restrictions on enums.

Enums

An *enumerated type*, or *enum*, is used to represent a fixed number of constant values.

enumType:
metadata **enum** *id '{'id [',' id]* [','] '}'*
;

The declaration of an enum of the form **enum** E = id_1, … id_n;
has the same effect as a class declaration

metadata **class** E {
final int index;
final String _name;
const E(**this**.index, **this**._name);
static const E id_1 = **const** E(0, 'id_1');
…
static const E id_n = **const** E(n - 1, 'id_{n-1}');

static const List<E> values = **const** <E>[id_1 … id_n];

```
  String toString() => 'E.$_name';
}
```

Except for the following differences: It is a compile-time error to subclass, mix-in or implement an enum. It is also a compile-time error to explicitly instantiate an enum via **new** or **const** or to access its private fields.

Switch

The *switch statement* supports dispatching control among a large number of cases.

switchStatement:
    switch *'('* *expression* *')' '{' switchCase* defaultCase? '}'*
  ;

switchCase:
    label (***case** *expression ':') statements*
  ;

defaultCase:
    *label** **default** *':' statements*
  ;

Given a switch statement of the form **switch** (*e*) { $label_{11}$ … $label_{1j1}$ **case** e_1**:** s_1 … $label_{n1}$ **..**$label_{njn}$ **case** e_n**:** s_n **default:** s_{n+1}} or the form **switch** (*e*) { $label_{11}$ … $label_{1j1}$ **case** e_1**:** s_1 … $label_{n1}$ **..**$label_{njn}$ **case** e_n**:** s_n}, it is a compile-time error if the expressions e_k are not compile-time constants, for all 1 <= *k* <= *n*. It is a compile-time error if values of the expressions e_k are not either:

- instances of the same class *C*, for all 1 <= *k* <= *n*, or
- instances of a class that implements int, for all 1 <= *k* <= *n*.
- instances of a class that implements String, for all 1 <= *k* <= *n*.

In other words, all the expressions in the cases evaluate to constants of the exact same user defined class, or are of certain known types. Note that the values of the expressions are known at compile-time, and are independent of any static type annotations.

It is a compile-time error if the class *C* has an implementation of the operator == other than the one inherited from Object unless the value of the expression is a string or integer.
The prohibition on user defined equality allows us to implement the switch efficiently for user defined types. We could formulate matching in terms of identity instead with the same efficiency. However, if a type defines an equality operator, programmers would find it quite surprising that equal objects did not match.

The **switch** statement should only be used in very limited situations (e.g., interpreters or scanners).

Execution of a switch statement of the form **switch** (*e*) {label$_{11}$ **..**label$_{1j1}$ **case** e_1**:** s_1 … label$_{n1}$ **..**label$_{njn}$ **case** e_n**:** s_n **default:** s_{n+1}} or the form **switch** (*e*) { label$_{11}$ … label$_{1j1}$ **case** e_1**:** s_1 … label$_{n1}$ **..**label$_{njn}$ **case** e_n**:** s_n} proceeds as follows:

The statement **var** *id* = *e*; is evaluated, where *id* is a variable whose name is distinct from any other variable in the program. In checked mode, it is a run time error if the value of *e* is not an instance of the same class as the constants e_1 … e_n.

Note that if there are no case clauses (*n = 0*), the type of *e* does not matter.

Next, the case clause **case** e_1**:** s_1 is executed if it exists. If **case** e_1**:** s_1 does not exist, then the default clause is executed by executing s_{n+1}**.**

A case clause introduces a new scope, nested in the lexically surrounding scope. The scope of a case clause ends immediately after the case clause's statement list.

Execution of a **case** clause **case** e_k**:** s_k of a switch statement **switch** (*e*) {label$_{11}$ **..**label$_{1j1}$ **case** e_1**:** s_1 … label$_{n1}$ **..**label$_{njn}$ **case** e_n**:** s_n **default:** s_{n+1}} proceeds as follows:

The expression e_k == *id* is evaluated to an object *o* which is then subjected to boolean conversion yielding a value *v.*

If *v* is not **true**, the following case, **case** e_{k+1}**:** s_{k+1} is executed if it exists. If **case** e_{k+1}**:** s_{k+1} does not exist, then the **default** clause is executed by executing s_{n+1}**.**
If *v* is **true**, let *h* be the smallest integer such that *h* >= *k* and s_h is non-empty. If no such *h* exists, let h = n + 1. The sequence of statements s_h is then executed. If execution reaches the point after s_h then a runtime error occurs, unless *h = n + 1*.

Execution of a **case** clause **case** e_k**:** s_k of a switch statement **switch** (*e*) {label$_{11}$ **..**label$_{1j1}$ **case** e_1**:** s_1 … label$_{n1}$ **..**label$_{njn}$ **case** e_n**:** s_n**}** proceeds as follows:

The expression e_k == *id* is evaluated to an object *o* which is then subjected to boolean conversion yielding a value *v.*

If *v* is not **true**, the following case, **case** e_{k+1}**:** s_{k+1} is executed if it exists.
If *v* is **true**, let *h* be the smallest integer such that *h* >= *k* and s_h is non-empty. The sequence of statements s_h is executed if it exists. If execution reaches the point after s_h then a runtime error occurs, unless *h* = *n*.

In other words, there is no implicit fall-through between cases. The last case in a switch (default or otherwise) can 'fall-through' to the end of the statement.

It is a static warning if the type of *e* may not be assigned to the type of e_k. It is a static warning if the last statement of the statement sequence s_k is not a break, continue, return or throw statement.

The behavior of switch cases intentionally differs from the C tradition. Implicit fall through is a known cause of programming errors and therefore disallowed. Why not simply break the flow implicitly at the end of every case, rather than requiring explicit code to do so? This would indeed be cleaner. It would also be cleaner to insist that each case have a single (possibly compound) statement. We have chosen not to do so in order to facilitate porting of switch statements from other languages. Implicitly breaking the control flow at the end of a case would silently alter the meaning of ported code that relied on fall-through, potentially forcing the programmer to deal with subtle bugs. Our design ensures that the difference is immediately brought to the coder's attention. The programmer will be notified at compile-time if they forget to end a case with a statement that terminates the straight-line control flow. We could make this warning a compile-time error, but refrain from doing so because do not wish to force the programmer to deal with this issue immediately while porting code. If developers ignore the warning and run their code, a run time error will prevent the program from misbehaving in hard-to-debug ways (at least with respect to this issue).

The sophistication of the analysis of fall-through is another issue. For now, we have opted for a very straightforward syntactic requirement. There are obviously situations where code does not fall through, and yet does not conform to these simple rules, e.g.:

```
switch (x) {
 case 1: try { … return;} finally { … return;}
}
```

Very elaborate code in a case clause is probably bad style in any case, and such code can

always be refactored.

It is a static warning if all of the following conditions hold:

- The switch statement does not have a **default** clause.
- The static type of *e* is an enumerated typed with elements id_1, … id_n.
- The sets $\{e_1, \ldots, e_k\}$ and $\{ id_1, \ldots id_n\}$ are not the same.

In other words, a warning will be issued if a switch statement over an enum is not exhaustive.