Implementation of Mixins in DART Programming Language

A Project Report

by

Purna ChatterjeePatra

email: chatp.99@gmail.com

Advisor : Dr. Cay Horstmann (email: horstmann@cs.sjsu.edu)

August 2012

ABSTRACT

In Object Oriented Programming Language, a Mixin is a class which allows certain functionalities to be inherited or used by the subclass, but, not to be instantiated. Mixins can be thought as a form of "Multiple Inheritance", because, a class or object may inherit most or all of the functionalities from one or more mixins. In practice, mixins encourage "Code Reuse" and avoid some well-known problems associated with "Multiple Inheritance".
Mixin can be also considered similar to Interface with implemented methods. Mixins do not need to implement the interface. When a class includes a mixin, the class itself implements the interface and also includes all of the mixin's attributes and methods. The definition and binding of the methods can be deferred till the runtime.
My project goal is to implement Mixin support in the DART programming language, which would allow to reduce code for developers.

Table of Contents

1. Introduction

1.1 What is Dart

Dart is Google's structured web programming language to solve current JavaScript limitations and problems. Dart promises easier development for cross-platform applications for the web as well as Androids and other mobile platforms.

1.2 Dart Goals and Features

- Create a structured yet flexible language for Web programming.
- Make Dart feel familiar and natural to programmers and thus easy to learn.
- Simple and unsurprising Object oriented programming language for the web
- Class-based single inheritance with interfaces
- Optional static types
- Real lexical scoping
- Single-threaded
- Familiar syntax

1.3 Purpose of the Project

Mixins can be thought as a form of "Multiple Inheritance", because, a class or object may inherit most or all of the functionalities from one or more mixins. In practice, mixins encourage "Code Reuse" and avoid some well-known problems associated with "Multiple Inheritance". My project goal is to implement Mixin support in the DART programming language, which would allow to reduce code for developers.

1.4 Report Structure

In this report, in Chapter 2, I will introduce the concepts of Multiple Inheritance, Mixins and Traits, and we will observe the differences between these concepts.

Chapter 3 will describe the proposal to introduce Traits as Mixin in Dart Language.

In Chapter 4, I present the specification of the proposed Trait implementation and the parser implementation of a small prototype code.

Finally, Chapter 5 will provide the future work for CS 298 section.

2. Trait as Mixin vs. Multiple Inheritance and Classic Mixin

Traits were originally defined as “composable units of behavior” : reusable groups of methods that can be composed together to form a class. Trait composition can be thought of as a more robust alternative to multiple inheritance. Traits may provide and require a number of methods. Required methods are like abstract methods in OO class hierarchies: their implementation should be provided by another trait or class.

The main difference between traits and alternative composition techniques such as multiple inheritance and mixin based inheritance is that upon trait composition, name conflicts should be explicitly resolved by the composer. This is in contrast to multiple inheritance and mixins, which define various kinds of linearization schemes that impose an implicit precedence on the composed entities, with one entity overriding all of the methods of another entity. While such systems often work well in small reuse scenarios, they are not robust: small changes in the ordering of classes/mixins somewhere high up in the inheritance/mixin chain may impact the way name clashes are resolved further down the inheritance/mixin chain. In addition, the linearization imposed by multiple inheritance or mixins precludes a composer to give precedence to both a method m1 from one class/mixin A and a method m2 from another class/mixin B: either all of A’s methods take precedence over B, or all of B’s methods take precedence over A.

Traits allow a composing entity to resolve name clashes in the individual components by either excluding a method from one of the components or by having one trait explicitly override the methods of another one. In addition, the composer may define an alias for a method, allowing the composer to refer to the original method even if its original name was excluded or overridden. Name clashes that are never explicitly resolved will eventually lead to a composition error. Depending on the language, this composition error may be a compile-time error, a runtime error when the trait is composed, or a runtime error when a conflicting name is invoked on a trait instance.

Trait composition is declarative in the sense that the ordering of composed traits does not matter. In other words, unlike mixin-based or multiple inheritance, trait composition is commutative and associative. This tremendously reduces the cognitive burden of reasoning about deeply nested levels of trait composition. In languages that support traits as a compile-time entity (similar to classes), trait composition can be entirely performed at compile-time, effectively “flattening” the composition and eliminating any composition overhead at runtime.

3. Proposal to introduce Trait as Mixin in Dart

3.1 Mixins

As their name reveals, Traits are usually used to represent a distinct feature or aspect that is normally orthogonal to the responsibility of a concrete type or at least of a certain instance. Therefore, the functionality of a Trait may be required by completely different types that have nothing in common or aren't even members of the same type hierarchy.

Let us say we want to model the ability to sing as such that: it could be applied to **`Birds`**, **`Persons`** (well, not all) or even to **`Radios`** (with just a little bit of imagination).

In Dart, I could come up with an Interface in order to express this "**`trait`**":

```
interface Singer{
   void sing();
}
```

Now every type which is also a singer may implement this interface and give an appropriate implementation:

```
class Bird implements Singer{
   ….
   void sing(){ … }
}
   ….....................
class Cicada extends Singer{
   ….
   void sing(){ … }
}
```

Now all of them (and their subclasses) can be treated as **`Singers`** even – as said – a **`Cicada`** is not a **`Bird`** nor a **`Radio`**.
If some of them (or all) should sing the same way, I could use 'copy n paste' or place a default implementation and use composition. But nevertheless, I have to provide a definition of sing() in every of that classes – if only because to delegate to that implementation.
No matter which option I choose, the proportion of boilerplate code isn't just small (Inheritance may be no option at all, because there may be no common parent class. Particularly, placing sing() into a too general supertype would mean that ALL subtypes would be singers).

3.2 Mixing traits 'statically'

I plan to provide an elegant way to define what it means to sing (at least a default implementation) and reuse it quite independently by separating that feature as a trait and using that trait as a mixin.

That said, I can (but don't have to) provide a definition for some or all methods of that trait and mix that trait into every type I want to:

```
trait Singer{
        sing(){
             print("Singing ...");
        }
}

class Bird extends Singer {...}
```

As we can see, the class definition of class Bird has mixed trait Singer into it's own definition using keyword 'extends'.
Now Bird has mixed in all methods (and all other members of the trait) into its own definition as if class Bird would have defined method sing() on its own – no boilerplate delegation code necessary.
Of course we now can ask every instance of a Bird to sing.

A last word on keyword 'extends': we also (or normally) use it to let a class inherit from a superclass. In case of a trait I only use it if I don't inherit from a superclass and then only for mixin in the first trait. All following traits (should I want to mix in more than one trait) are mixed in using keyword 'with':

```
class Insect {...}
class Cicada extends Insect with Singer {...}

class Bird extends Singer with Flyer {...}
```

3.3 Mixing traits 'dynamically'

In case of class Person, I face another special problem: I only want some instances of Person to be singers. I can't implement interface Singer on class Person since this would turn every instance of Person into a singer.

Fortunately, I can allow to mix in a trait 'dynamically' when creating a new instance of a class. In that case, only that special instance will be a singer and provide the methods of that trait:

```
class Person{
   tell () { print ( "here's a little story ..." ) ;}
}
var SingingPerson = new Person with Singer;
```

Here we've created a new instance of type Person, saying that this instance is also a Singer by using keyword 'with'.
Actually, we'll receive an instance of a new anonymous class that is a Person as well as a Singer.

3.4 Conclusion

Dart's proposed traits are an elegant way to separate concerns. Every feature may be separated within an own trait and can than be mixed into every type or instance that should posses that trait.

4. Implementation of Traits in Dart

4.1 Specification

a) Trait Definition:

```
trait identifier superclass? '{'classMemberDefinition*'}'
```

b) ClassMemberDefinition:

```
declaration ';'
methodSignature
functionBody
```

c) MethodSignature:

```
factoryConstructorSignature
static?
functionSignature
getterSignature
setterSignature
operatorSignature
constructorSignature
initializers?
```

d) Trait Declaration:

```
trait T1 {
        m1(String p1); //abstract method
}
```

no need to declare the method as abstract—an unimplemented method in a trait is automatically abstract. A subclass can provide an implementation:

```
class C1 extends T1{
        m1(String p1) { ... }
}
```

If I need more than one trait, I can add the others using the with keyword:

```
class C2 extends T1 with T2 with T3
```

A class can have only one superclass but any number of traits.The methods of a trait need not be always abstract. For example, I can make our class C1 into a trait:

```
trait C1 {
        m1(String p1) { ... }
```

```
}
```

The trait C1 provides a method with an implementation

I can add a trait to an individual object when I construct it. To set up an example, I will use the trait C which comes with a do-nothing implementation:

```
trait T2 {
        m1(String p1) { }
}
```

Let's use that trait in a class definition:

```
class C2 extends C3 with T2 {
        m2(Double p2) {
                if (...) m1("Testing");
                else ...
        }
        ...
}
```

Now, nothing gets done, which might seem pointless. But I can "mix in" a better trait when constructing an object. The trait T3 extends the trait T2:

```
trait T3 extends T2 {
        m1(String p3) { print(p3); } // overrides method m1 of
trait T2
}
```

I can add this trait when constructing an object:

```
var O1 = new C2 with T3;
```

When calling m1 on the O1 object, the m1 method of the T3 trait executes. Of course, another object can add in a different trait:

```
var O2 = new C2 with T4;
```

I can add, to a class or an object, multiple traits that invoke each other starting with the last one. This is useful when I need to transform a value in stages.

```
trait T4 extends T3{
        m1(String p4) {super.m1("Another " + p4);}
}

trait T5 extends T3{
        m1(String p5) {super.m1("One More " + p5);}
}
```

Note that each of the m1 methods passes a modified message to super.m1. With traits, super.m1 does not have the same meaning as it does with classes. Instead, super.m1 calls the next trait in

the trait hierarchy, which depends on the order in which the traits are added. Generally, traits are processed starting with the last one.
To see why the order matters, compare the following two examples:

```
var O3 = new C2 with T3 with T4 with T5;

var O4 = new C2 with T3 with T5 with T4;
```

A field in a trait can be concrete or abstract. If I supply an initial value, the field is concrete, otherwise it's abstract.

```
trait T5 extends T3 {
        num x = 15; // A concrete field
...
}
```

A class that mixes in this trait acquires an x field. In general, a class gets a field for each concrete field in one of its traits. These fields are not inherited; they are simply added to the subclass.
An uninitialized field in a trait is abstract and must be overridden in a concrete subclass.

```
trait T5 extends T3 {
        num x; // An abstract field
        m1 (String p5) {
                super.m1(if (p5.length <= x) p5 else p5 +
"..."); }
...
}
```

When I use this trait in a concrete class, I must supply the x field:

```
class C2 extends C1 with T2 with T5 {
        num x = 20; // No override necessary
...
}
```

This way of supplying values for trait parameters is particularly handy when I construct objects on the fly.

```
var O3 = new C2 with T3 with T4 with T5 {
        num x = 20;
}
```

4.2 Implementation

Right now I have implemented the parser part, so that the test file having the code inserted :

```
interface Shape {
   num perimeter();
}
trait Rectangle implements Shape {
   final num height, width;
   Rectangle(num this.height, num this.width); // Compact
constructor syntax.
   num perimeter() => 2*height + 2*width; // Short function
syntax.

}
class Square extends Rectangle {
   Square(num size) : super(size, size);
}
```

will not give any error when dart2js compiler tries to convert this code into javascript. The "trait" here is considered just as another class right now – so the javascript converted code is becoming

```
Isolate.$defineClass("Rectangle", "Object", [], {
 perimeter$0: function() {
  return $.mul(2, this.height) + $.mul(2, this.width);
 }
});

Isolate.$defineClass("Square", "Rectangle", ["width",
"height"], {
});

.
.
.

$.Square$1 = function(size) {
  return new $.Square(size, size);
};
```

To achieve this I had to make changes in 3 files – all of them in **~/dart/dart-sdk/lib/dart2js/lib/compiler/implementation/scanner** directory.

The first one is **keyword.dart**, where I included this code part

```
class Keyword implements SourceString {
   .
   .
   .
   static final Keyword TRAIT = const Keyword("trait");
   static final Keyword WITH = const Keyword("with");
   .
   .
   .
}
.
.
.
static final List<Keyword> values = const <Keyword> [
   .
   .
   .
   TRAIT,
   .
   WITH];
```

The next file is **`listener.dart`**, where I included this part:

```
void beginTraitDeclaration(Token token) {
}

void endTraitDeclaration(int interfacesCount, Token
                         beginToken, Token extendsKeyword,
                         Token implementsKeyword, Token
                         endToken) {
}
```

And, the most important change till now is in the **`parser.dart`** file, with the following code inclusion :

```
Token parseTopLevelDeclaration(Token token) {
   .
   .
   .

   } else if (value === 'trait') {
      print('TraitToken');
      return parseTrait(token);
    }
   .
   .
   .
}
```

```
Token parseTrait(Token token) {
    Token begin = token;
    listener.beginTraitDeclaration(token);
    if (optional('abstract', token)) {
      // TODO(ahe): Notify listener about abstract modifier.
      token = token.next;
    }
    token = parseIdentifier(token.next);
    token = parseTypeVariablesOpt(token);
    Token extendsKeyword;
    if (optional('extends', token)) {
      extendsKeyword = token;
      token = parseType(token.next);
    } else {
      extendsKeyword = null;
      listener.handleNoType(token);
    }
    Token implementsKeyword;
    int interfacesCount = 0;
    if (optional('implements', token)) {
      do {
        token = parseType(token.next);
        ++interfacesCount;
      } while (optional(',', token));
    }
    token = parseTraitBody(token);
    listener.endTraitDeclaration(interfacesCount, begin,
extendsKeyword, implementsKeyword, token);
    return token.next;
  }

  Token parseTraitBody(Token token) {
    return parseClassBody(token);
  }
```

5. Future Work

Right now, the implementation of trait is just running without any compile-time error. But, it is not yet achieving the required goal, specially the mixin part. Right now, parser is treating the trait just as another class. So, all the trait is doing right now is implementing the single inheritance. In CS298, I am going to implement the actual mixin part, activating the multiple inheritance and code reuse.

To achieve this goal, I am planning to take advantage of the inheritance in javascript. I will use the dart2js dart-to-javascript convertor to resolve necessary conflicts in the mixin and convert the trait dart code into inheritance code into javascript. From there on, javascript compiler will work on its own to resolve the inheritance.

Reference

[1] Ancona, D., Lagorio, G., & Zucca, E. (2003). Jam--designing a java extension with mixins. ACM Transactions on Programming Languages & Systems, 25(5), 641.

[2] Bergel, A., Ducasse, S., Nierstrasz, O., & Wuyts, R. (2008). Stateful traits and their formalization. Computer Languages, Systems & Structures, 34(2), 83. doi:10.1016/j.cl.2007.05.003

[3] Cassou, D., Ducasse, S., & Wuyts, R. (2009). Traits at work: The design of a new trait-based stream library. Computer Languages, Systems & Structures, 35(1), 2. doi:10.1016/j.cl.2008.05.004

[4] Gilad Bracha, The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance, Ph.D. dissertation, Dept. of Computer Science, University of Utah 1992.

[5] Gilad Bracha and David Griswold. Extending Smalltalk with Mixins. OOPSLA96 Workshop on Extending the Smalltalk Language.

[6] Gilad Bracha and William Cook. Mixin-based inheritance. In Proc. of the Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming, October 1990 .

[7] Lars Bak, Gilad Bracha, Steffen Grarup, Robert Griesemer, David Griswold and Urs Hoelzle. Mixins in Strongtalk. ECOOP02 Workshop on Inheritance.

[8] Strout, J. (2005). Mixins without multiple inheritance. Dr.Dobb's Journal: Software Tools for the Professional Programmer, 30(1), 37.

[9] Wikipedia contributors. "Mixin." Wikipedia, The Free Encyclopedia, 6 Jan. 2012. http://en.wikipedia.org/w/index.php?title=Mixin&oldid=469952544

[10] Zdun, U., Strembeck, M., & Neumann, G. (2007). Object-based and class-based composition of transitive mixins. Information & Software Technology, 49(8), 871. doi:10.1016/j.infsof.2006.10.001