http://www.youtube.com/watch?v=aboZctrHfK8

42 Programming Language Meta-programming as Default

Motivation

Libraries:

- collections of data and behavior
- have well-defined interfaces
- suitable to be used by multiple independent programs.

Library usage dramatically increase programmers' productivity.

Today a well-designed program can use up to 10 third-party libraries. However, when more than 10 libraries are used keeping trace of the issues arising from those libraries and their interplay become very difficult.

Today, a goal for tomorrow

Today a programmer can use a library containing millions of instructions without the need to understand each instruction individually, and instead can rely on the library interface.

With 42, a programmer will be able to use millions of libraries without the need to understand each library individually, and may rely on libraries managing and organizing other libraries.

This would alter the perception of what programming is: Normally the programming language is considered “the thing the programmers use to code”. The philosophy of 42 is different: the language provides the foundation for libraries, and libraries is what programmers use to code.

Smaller libraries

Today libraries tends to be massive chunks of code with tons of functionalities packed into.

This is because importing and managing libraries is hard, so is convenient for libraries to "expand" their domain in related areas.

With easy - transparent - libraries, There will be no reason to have so many functionalities packed all together.

Purpose

In 42 a *library* is a self-contained piece of code that is released by other programmers. A good library has a modularized documentation and contains a significant amount of code dedicated to try and deliver good error messages when used incorrectly with links to the relevant documentation section.

In 42 an *active library* is a library offering methods that generate other libraries. Such methods can compose code (other libraries) to obtain highly personalized libraries that can provide a clear dedicated semantic to the programmer.

Purpose: Primary features

- using (active) libraries is simple;
- developing/deploy (active) libraries is simple;
- using good (active) libraries produces good error messages;
- the language semantics/typing helps to isolate erroneous (and even bad) libraries;

Non-goals: trying to accomplish this goals would harm our core goals: size of the binary, memory footprint of the objects and performance predictability.

Decorators

Library composition is obtained using decorators, that are build upon basic decorators **`Use`** and **`Adapt`**.

Composing classes “as modules”,

Decorators

Library composition is obtained using decorators, that are build upon basic decorators **`Use`** and **`Adapt`**.

Composing classes “as modules”,
with a associative and commutative use/merge/sum operation:

Decorators

Library composition is obtained using decorators, that are build upon basic decorators **`Use`** and **`Adapt`**.

Composing classes “as modules”,
with a associative and commutative use/merge/sum operation:

- nested classes with same name are recursively merged;

Decorators

Library composition is obtained using decorators, that are build upon basic decorators **`Use`** and **`Adapt`**.

Composing classes “as modules”,
with a associative and commutative use/merge/sum operation:

- nested classes with same name are recursively merged;
- the set of implemented interfaces is merged;

Decorators

Library composition is obtained using decorators, that are build upon basic decorators **`Use`** and **`Adapt`**.

Composing classes “as modules”,
with a associative and commutative use/merge/sum operation:

- nested classes with same name are recursively merged;
- the set of implemented interfaces is merged;
- methods with same name are merged,

Decorators

Library composition is obtained using decorators, that are build upon basic decorators **`Use`** and **`Adapt`**.

Composing classes “as modules”,
with a associative and commutative use/merge/sum operation:

- nested classes with same name are recursively merged;
- the set of implemented interfaces is merged;
- methods with same name are merged,and conflict arises if both version have an implementation or if the headers are not compatible.

Decorators

Library composition is obtained using decorators, that are build upon basic decorators **`Use`** and **`Adapt`**.

Composing classes “as modules”,
with a associative and commutative use/merge/sum operation:

- nested classes with same name are recursively merged;
- the set of implemented interfaces is merged;
- methods with same name are merged,and conflict arises if both version have an implementation or if the headers are not compatible.

Good, but... how to merge fields/constructors?

Constructors and fields

Constructors and fields

What is exactly a field/constructor in 42,

Constructors and fields

What is exactly a field/constructor in 42,
since I can only access them as getters/setters/factories?

Constructors and fields

What is exactly a field/constructor in 42,
since I can only access them as getters/setters/factories?
What is the exact meaning of writing `Point:{(N x, N y)}`?

Constructors and fields

What is exactly a field/constructor in 42,
since I can only access them as getters/setters/factories?
What is the exact meaning of writing `Point:{(N x, N y)}`?
It is just sugar for

```
Point:{  '(N x, N y) desugar into
  type method Outer0 (N^ x, N^ y)
  method N x()
  method N y()
  }
```

That is, a class with just abstract methods.

```
Point:{  '(N x, N y) desugar into
  type method Outer0 (N^ x, N^ y)
  method N x()
  method N y()
  }
```

```
Point:{  '(N x, N y) desugar into
  type method Outer0 (N^ x, N^ y)
  method N x()
  method N y()
  }
```

In many languages, a concrete class have no abstract methods; In 42, a concrete class have a coherent set of abstract methods:

```
Point:{  '(N x, N y) desugar into
  type method Outer0 (N^ x, N^ y)
  method N x()
  method N y()
  }
```

In many languages, a concrete class have no abstract methods; In 42, a concrete class have a coherent set of abstract methods:

- A class with no abstract methods is concrete (as `java.lang.Math`)

```
Point:{  '(N x, N y) desugar into
  type method Outer0 (N^ x, N^ y)
  method N x()
  method N y()
  }
```

In many languages, a concrete class have no abstract methods; In 42, a concrete class have a coherent set of abstract methods:

- A class with no abstract methods is concrete (as `java.lang.Math`)
- A class with a single abstract **`type`** method returning **`Outer0`** whose parameters are all of placeholder type (as **`N^`** `x`) is concrete.

```
Point:{  '(N x, N y) desugar into
  type method Outer0 (N^ x, N^ y)
  method N x()
  method N y()
  }
```

In many languages, a concrete class have no abstract methods; In 42, a concrete class have a coherent set of abstract methods:

- A class with no abstract methods is concrete (as `java.lang.Math`)
- A class with a single abstract **`type`** method returning **`Outer0`** whose parameters are all of placeholder type (as **`N^`** `x`) is concrete.
- If such abstract **`type`** method is present, then, for each parameter, it is allowed to have abstract getters and setters, where the name maps back to the parameter name.

```
Point:{  '(N x, N y) desugar into
  type method Outer0 (N^ x, N^ y)
  method N x()
  method N y()
  }
```

In many languages, a concrete class have no abstract methods; In 42, a concrete class have a coherent set of abstract methods:

- A class with no abstract methods is concrete (as `java.lang.Math`)
- A class with a single abstract **`type`** method returning **`Outer0`** whose parameters are all of placeholder type (as **`N^`** `x`) is concrete.
- If such abstract **`type`** method is present, then, for each parameter, it is allowed to have abstract getters and setters, where the name maps back to the parameter name.

Take a deep breath, to me this is game changing and mind blowing. There are only methods. The constructor/getter implementation is not protected/hidden/secret/native. It is just not there.

```
Point:{  '(N x, N y) desugar into
  type method Outer0 (N^ x, N^ y)
  method N x()
  method N y()
  }
```

In many languages, a concrete class have no abstract methods; In 42, a concrete class have a coherent set of abstract methods:

- A class with no abstract methods is concrete (as `java.lang.Math`)
- A class with a single abstract **`type`** method returning **`Outer0`** whose parameters are all of placeholder type (as **`N^`** `x`) is concrete.
- If such abstract **`type`** method is present, then, for each parameter, it is allowed to have abstract getters and setters, where the name maps back to the parameter name.

Take a deep breath, to me this is game changing and mind blowing. There are only methods. The constructor/getter implementation is not protected/hidden/secret/native. It is just not there. The whole formal model works without requiring any difference between abstract methods and getters/setters/constructors.

Composing Constructors and fields

```
C:{
  type method Library point(){(N x, N y)
    method N distance(Outer0 that){
      x=this.x()-that.x()
      y=this.y()-that.y()
      return (x*x+y*y).sqrt()
      }

    }
  }
ColPoint:Use[C.point()]<{(N x, N y, Color color)
  type method Outer0 (N^ x,N^ y)
    Outer0(x:x,y:y,color:Color.black())
  }
```

Composing Constructors and fields

```
C:{
  type method Library point(){(N x, N y)
    method N distance(Outer0 that){
      x=this.x()-that.x()
      y=this.y()-that.y()
      return (x*x+y*y).sqrt()
      }
    method Outer0 clone(N x) Outer0(x:x,y:this.y())
    }
  }
ColPoint:Use[C.point()]<{(N x, N y, Color color)
  type method Outer0 (N^ x,N^ y)
    Outer0(x:x,y:y,color:Color.black())
  } 'now with sets the color to black :-(
```

Composing Constructors and fields

```
C:{
  type method Library point(){(N x, N y)
    method N distance(Outer0 that){
      x=this.x()-that.x()
      y=this.y()-that.y()
      return (x*x+y*y).sqrt()
      }
    method Outer0 clone(N x)
    }
  }
ColPoint:Data[]<<Use[C.point()]<{(N x, N y, Color color)
  type method Outer0 (N^ x,N^ y)
    Outer0(x:x,y:y,color:Color.black())
  } 'let data do its job
```

Traits, trivially

Traits are just groups of methods
(and they can implement interfaces)

```
{<:Foo,Bar ..method T x().. method Void foo().. }
```

Just add traits together to get other threads.

```
Use[Lib.a();Lib.b();..]
```

Sum a trait to a (concrete) class and get as a result a better (concrete) class, if such class offers all the methods required (i.e. abstract) in the thread.

```
Use[Lib.a();Lib.b();..]<{..}
```

Expression problem as matrix

	toString	eval

Num	`{`
	`Exp:{interface<:HasToString }`
	`Num:{(N that)<: Exp`
	`  method toString()`
	`  S""++this.that()`
	`  }}`
		`{`
		`Exp:{interface<:HasEval }`
		`Num:{(N that)<: Exp`
		`  method eval()`
		`  this.that()`
		`  }}`

Sum	`{`
	`Exp:{interface<:HasToString }`
	`Sum:{(Exp left, Expr right)<: Exp`
	`  method toString()`
	`  this.left().toString()`
	`  ++this.right().toString()`
	`  }}`
		`{`
		`Exp:{interface<:HasEval }`
		`Sum:{(Exp left, Expr right)<: Exp`
		`  method eval()`
		`  this.left().eval()`
		`  ++this.right().eval()`
		`  }}`

UMinus	`{`
	`Exp:{interface<:HasToString }`
	`UMinus:{(Exp that)<: Exp`
	`  method toString()`
	`  S"-"++this.that().toString()`
	`  }}`
		`{`
		`Exp:{interface<:HasEval }`
		`UMinus:{(Exp that)<: Exp`
		`  method eval()`
		`  this.that().eval()*-1N`
		`  }}`

Expression problem as matrix

Just sum the desired line/rows together

```
MySolution:ExprProblem(
  useRows:SList[S"uMinus";S"sum"]
  useColums:SList[S"eval";S"toString"]
  )
```

Adapt

Ok, **`Use`** is cool. What about **`Adapt`**? It can rename methods, nested classes, and can set members private. More than that, **`Adapt`** is cool since it can apply **`Use`** internally.

```
D:Adapt[Name"C" into: Name"K";Name"A" into: Name"K"]<{
  C:{method N c()}
  A:{method N a()}
  method C m(A a)
  }
```

rewrites to

```
D:{
  K:{method N c() method N a()}
  method K m(K a)
  }
```

Adapt

```
D:Adapt[ Name"C::A" into Name"Outer0" ] < {
  method N top()
  C:{method N c()
    A:{ method N a() }
    }
  }
```

rewrites to

```
D:{
  method N top()
  method N a()
  C:{method N c() }
  }
```

Should this be called **Move** instead of **Adapt**?

Adapt

```
D:Adapt[ Name"Outer0" into Name"C::A" ] < {..}
```

rewrites to

```
D:{ C:{ A:{..}}}
 'or D:{interface C:{interface A:{..}}}
```

Should this be called **Move** instead of **Adapt**? And finally,

```
D:Adapt[ Name"C" into Name(S) ] < {
  C:{.. B:{..}}
  method C m(C::B x)
  }
```

rewrites to

```
D:{
  method S m(S::B x)
  }
```

This last operation requires to check the structural type of **S** and **S::B** to be compatible with **C** and **C::B**. Should this be called **Replace** instead of **Adapt**?

Towel Intro

A towel is about the most massively useful thing a programmer can have. A towel has immense psychological value, and you should always know where your towel is.

```
{reuse L42.is/BasicTowel
Main:{
  Debug(S"Hello world")
  return ExitCode.normal()
  }
}
```

Here **`Debug`**, **`S`** and **`ExitCode`** are classes defined inside the towel BasicTowel. You can code without a towel, but this means starting from first principles, witch could be quite unpleasant; especially since the only primitive things that 42 offers are Library literals (code as first class entities), the constant **`void`**, and the types **`Library`**, **`Void`** and **`Any`**.

Towel Intro

The header reuse L42.is/BasicTowel imports the code from the URL L42.is/BasicTowel and places it at the start of the class. Code is(=will be) downloaded/updated from the Internet automatically. We generally call towels libraries that are expected to provide standard functionalities and types, such as numbers N, booleans Bool, strings S and various kinds of system errors (and Load, as discussed later). However, we do not expect all L42 programs to reuse the same towel. For hygienic reasons, in real life everyone tends to use their own towel. For similar reasons, any 42 program will use its own towel. We expect different programs to use massively different libraries for what in other languages is the standard library. That is, there is no such thing as ”the L42 standard library”

Towel Load

Most L42 libraries are not towels. All 42 libraries are closed code.
Thus, all non-towel 42 library is generic (have abstract classes/methods)

```
{reuse L42.is/BasicTowel
Gui:Load[]<{reuse L42.is/2dGui}
Main:{
  Gui.alert(S"Hello world")
  return ExitCode.normal()
}
```

`Load[]<{..}` would intuitively behave like

`Adapt[Name"N"into:Name(N);Name"S"into:Name(S);...]<{..}`

`Load` uses `Adapt` to redirect such abstract classes to their incarnation in BasicTowel, assuming they respect some structural interface.

Towel Multiple Towels

Towels shines when multiple towels are used at the same time.

```
{reuse L42.is/AdamsTowel
 ' here you can access to lots of
 ' utility classes declared inside the towel
 ' including numbers, strings and so on.
C:{reuse L42.is/FordTowel
    'here you can access a different set of classes.
    'For example, N would refer to the number in FordTowel
    'and to see the number declared in AdamTowel
    'you have to write Outer1::N
   }
}
```

Different code parts reason about different set of classes; including those predefined in other languages.
Useful for code that reasons on code; that is a very common task in L42.

Towel Multiple Towels

Define and deploy our own towel: like BasicTowel, with Gui as before.

```
{reuse L42.is/DeploymentTowel
Main:Deploy[URL"../GuiTowel";permissions:S"..."]<{
  reuse L42.is/BasicTowel
  Gui:Load[]<{reuse L42.is/2dGui}
  }
}
```

DeploymentTowel is just some variation of BasicTowel, with Deployment functionalities already included. Here **`Load`** comes from BasicTowel, and the nested code literal results in closed code, thus **`Deploy`** will be able to correctly export it and save it in the provided URL, if we have permission.

Towel Embroidery

We can mix and match different implementations for numbers and strings (as an example). This requires some special care:

- strings should know their length (as a number)
- numbers should be comparable (into true/false booleans)
- booleans (and numbers) should be representable (as strings)

```
{reuse L42.is/DeploymentTowel
Main:
  Deploy[URL"../MyTowel";permissions:S"..."]<<
  Adapt[Name"NAlt" into:Name"N"; Name"SAlt" into:Name"S"]<<
  Abstract[Name"N";Name"S"]< 'now N and S are abstract
    { 'code with alternative versions for N and S
    reuse L42.is/BasicTowel
    NAlt:Load[]<{reuse L42.is/InfinitePrecisionNatural}
    SAlt:Load[]<{reuse L42.is/UTF8Strings}
    }
```

Library Deployment

Non-towel libraries can also be deployed in a similar way:

```
{reuse L42.is/DeploymentTowel
Main:
  Deploy[URL"../MyTowel";permissions:S".."]<<
  Optimize::RemoveUnreachableCode[]<<
  ExposeAsLibrary[Name"MyLib"]<{
    reuse  L42.is/BasicTowel
    MyLib:...  'this refers to the file MyLib.L42
    } }
```

Where `ExposeAsLibrary[Name"MyLib"]<{..}` could be:

```
  Adapt[makePrivate:Name"PrImpl"]<<
  Adapt[Name"PrImpl::N" into:Name"N";Name"PrImpl::S" into:..]<<
  Adapt[Name"PrImpl::MyLib" into:Name"Outer0"]<<
  Adapt[Name"Outer0" into:PrImpl]<<
  Abstract[Name"N";Name"S";Name"Bool";..]<{..}
```

Support for decent size applications

```
{reuse L42.is/AnotherDeploymentTowel
Local:UrlData.readOnly(prefix:URL"file:///../myProjects/")
Remote:UrlData.writeOnly(prefix:URL"L42.is/byMarco/",permission:S"..")
Web:UrlData.writeOnly(prefix:URL"www.myWebsite.com/",permission:S"..")
Header:Deployer[]<{()
    method DeployResult() {
      var acc=Project(Local"header")
      b= Benchmark(Local"headerBenchmark")
      acc :=b.optimize(acc, inlining:10Minute)
      acc :=b.optimize(acc, parallel:1Minute)
      return acc.deploy(Remote"header")
      }
  }
GraphicEngine:Deployer[]<{(Header h)
  method DeployResult() {
    var acc=Project(Local"graphicEngine")
       'inside reuse L42.is/byMarco/header
    b1=Benchmark(Local"graphicEngineBenchmark1")
    b2=Benchmark(Local"graphicEngineBenchmark2")
    acc =b1.optimize(acc, inlining:10Minute)
    acc =b2.optimize(acc, parallel:10Minute)
    return acc.deploy(Remote"graphicEngine")
   }
 }
```

Support for decent size applications

```
MyGame:Deployer[]<{(Header h GraphicEngine e)
  method DeployResult() {
    var acc=Project(Local"myGame")
    b=Benchmark(Local"myGameBenchmark")
    acc =b.optimize(acc, inlining:10Minute)
    acc =b.optimize(acc, parallel:10Minute)
    return acc.deploylist()[
        windowsExe: acc Web"myGame.exe";
        linuxExe: acc   Web"myGame";
        javaExeJar: acc Web"myName.jar";
        htmlPage: Local"index.html".asText())
          name: Web"index.html";

      ]
    }
  }
Main:Build(MyGame) 'Build compute dependency graph and check for changes
```

Getting started with Active Libraries

```
reuse L42.is/AdamsTowel
AgeInput: Gui.input(receive:N, request:S"insert age")
 'class AgeInput is the first class we declare!
 'is generated by method "Gui.input"
Main:{
  Gui::Alert(S"your age is "+AgeInput().asText())
   'class AgeInput principal method (the one with no name)
   'ask the user to insert a number N
  return ExitCode.normal()
  }
```

Getting started with Active Libraries

```
reuse  L42.is/base
BirthInput: Gui.input(
  title:S"Date of birth"
  receive:Date  'we ask a widgets for the type Date
  request:S"insert your date of birth")
Main:{
  Date userDate= BirthInput()
  Gui::Alert(S"your date of birth is "+userDate.asText())
  return ExitCode.normal()
  }
```

`Gui.input` can generate widgets for each type.

Getting started with Active Libraries

```
reuse  L42.is/base
Tables: DB(connection:DBConnection"....")
    'create a class with a nested for each db table
    'plus utility features to connect with the db
...
me=Tables::Student(id:....)
    'get student from id, queries also available.
....me.name().....
```

DB turns a db into classes. If the db is updated, and `name` is renamed in `fullName`, 42 will point to the error place and we will be guided into correspondingly updating our program.

```
reuse  L42.is/base
Tables: DB(connection:....) ' with the updated db-schema
...
me=Tables::Student(id:....)
....me.name().... 'now I get a red sign here!
```

Visualize DB

```
{reuse  L42.is/AdamsTowel
Db: Load[]<{reuse L42.is/Db}
Gui: Load[]<{reuse L42.is/2dGui}
Tables: DB(S"...my connection string...")
StudentList: Collections.list(Tables::Student)
StudentsView: Gui.tableOf(StudentList)
QueryFrom: Tables::Query[result:StudentList](
  DB::SQL"Select * from Student where country=@country")
Main:{
  connection=Tables.connect()
  fromItaly=QueryFrom(connection, country:S"Italy")
  Gui.show(StudentsView(fromItaly))
  return ExitCode.normal()
  }
}
```

Visualize DB

```
DB: Load[]<{L42.is/DB}
Gui: Load[]<{L42.is/2dGUI}
Tables: DB(S"...my connection string...")
StudentList: Collections.list(Tables::Student)
StudentsView: Gui.tableOf(StudentList)
QueryFrom: Tables::Query[result:StudentList](
  DB::SQL"Select * from Student where country=@country")
```

- **`Load[]<{L42.is/..}`** **`Load`** is loading the code of a library and, according to the conventions in *L42.is/AdamsTowel*, is linking numbers, strings and other general purposes classes to their correspondent inside of *L42.is/AdamsTowel*.
- **`DB(S"...my connection string...")`** creates classes to reify tables in the database.
- **`StudentList: Collections.list(Tables::Student)`** creates a collection for students.

Visualize DB

```
StudentsView: Gui.tableOf(StudentList)
QueryFrom: Tables::Query[result:StudentList](
  DB::SQL"Select * from Student where country=@country")
```

- `StudentsView: Gui.tableOf(StudentList)` create a widget visualizing lists of students. Every collection can be seen as a table, if the object supports iteration over its fields and its class supports iteration over its field names and types. `Data[]` can easily add those methods for you.
- `QueryFrom: Tables::Query[result:StudentList](..)` creates a (class that can only perform a) prepared query. If the query is of the form `"Select * from TableName where ..."` then it can produce a list of instances of a class under `Table::`. Otherwise, it would also generate a nested class for the result.

Visualize DB

```
Main:{
  connection=Tables.connect()
  fromItaly=QueryFrom(connection, country:S"Italy")
  Gui.show(StudentsView(fromItaly))
  return ExitCode.normal()
  }
```

- `Tables.connect()` connects to the database.
- `fromItaly=QueryFrom(connection, country:S"Italy")` Using `QueryForm` we ask for students coming from Italy. Notice how `country:` is part of the method signature. The name of the parameter was extracted while generating the class. A good IDE autocompletion would take advantage of it.
- `Gui.show(StudentsView(fromItaly))` close the circle, showing a table view of our Italian students.

Maintainability?

```
{reuse  L42.is/AdamsTowel
Db: Load[]<{reuse L42.is/Db}
Gui: Load[]<{reuse L42.is/2dGui}
Tables: DB(S"...my connection string...")
StudentList: Collections.list(Tables::Student) ' HERE!!
StudentsView: Gui.tableOf(StudentList)
QueryFrom: Tables::Query[result:StudentList](
  DB::SQL"Select * from Student where country=@country")
Main:{
  connection=Tables.connect()
  fromItaly=QueryFrom(connection, country:S"Italy")
  Gui.show(StudentsView(fromItaly))
  return ExitCode.normal()
  }
}
```