Code reuse with 42

Many mains in 42

```
reuse L42.is/AdamTowel

Main0: Debug(S"Hello world1")

Main1: {
  Debug(S"Let's do something more")
  x=7Num*6Num
  Debug(S"Hello "++x)
  Library res= {
    class method S foo()=S"Surprise"
    }
  return res
  }
//Our computation can return code; 'code' in 42 is of type 'Library'

Main2: Debug(S"Hello "++Main1.foo())
//Main2 can use Main1, since is declared earlier.
//Classes are compiled top-bottom
```

Incremental compilation

```
//(STEP0)
A: {// a class with a method .ma()
  class method Library ma() = {class method Library k()={}}
  }}//.ma() just returns a library with a method .k()
B: A.ma()
C: B.k()
```

During execution becomes

```
//(STEP1)
A: {..} //as before
B: {class method Library k()={}}
C: B.k()
```

and finally

```
//(STEP2)
A: {..}//as before
B: {class method Library k()={}}
C: {}
```

The program can perform side effects while creating classes. Execution is just the production of all the classes.

My first class

```
reuse L42.is/AdamTowel
Point: Data <>< {Num x,Num y}
Main2: Debug(S"Hello "++Point(x: 4Num, y: 2Num))
//-lets dissect the code above:
{Num x,Num y}//(0) an unnamed library
Num x //(1) a final field x of type Num; it is equivalent to a getter
method Num x() //(2) the equivalent getter.  Note:  is abstract/no body
var Num x //(3)a non final field.  Equivalent to getter+setter
Point: Data <>< {..} //(4) this is just a main that returns code.
//That code will be the definition for class 'Point'
Data <>< {..} //(5) Data is a class decorator.
//In 42 operators are method calls; you can read it as if it was
Data.babelFish({Num x,Num y})
//Data adds features to the argument:  factory, equality, toS,..
Point(x: 4Num, y: 2Num) //(6) factory call.  Parameter names are required
```

Example 42, Item Rock and Wall

```
Item: {interface
 Point point    /*equivalent to*/ method Point point()
 method Item break()
 }
Rock: Data <>< {implements Item
  Num weight //it also inherit the 'Point point' field
  method break()=Rock(weight: this.weight()-1\, point: this.point())
  }
Wall: Data <>< {implements Item     Num height
  method break()=Rock(weight: 100\, point: this.point())
  }
..
myRock=Rock(weight: 15\ point: Point(x: 12\ y: 0\))
myRock.break()
myRock.weight()
myRock.point()
myRock.toS()
myRock==myRock
..
```

Traits!

Traits: a code reuse mechanism based on reusable units of code, that can be composed into either more traits or classes.

```
Part1: Trait({..})//has method a implemented, has method b abstract
Part2: Trait({..})//has method b implemented, has method a abstract
Result: Part1 & Part2 //has both a and b implemented
```

Many languages now support some variation of traits:

- Smalltalk: here is where the trait name first becomed popular.
- Rust has 'traits', but they are more like Haskell typeclasses.
- Scala has 'traits', but they are more like python multiple inheritance/C++ mixin layers
- Java8 interfaces with default methods can be used as if they was Smalltalk traits.

42 traits are implemented as a library over primitive code manipulation operators. Trait composition in 42 is similar to Smalltalk trait composition, but it is nested: Trait code can have nested classes; composition propagate recursivelly over nested classes with the same name.

My first Trait compostion

```
Part1: Trait({
  class method S message()

  class method S quoteMessage()=
    S"The message is: '"++this.message()++S"'."
  })

Part2: Trait({ class method S message()=S"42" })

Result: Part1 & Part2 & {}

Main: Debug(Result.quoteMessage())//S"The message is:  '42'."

// after evaluation, Result becomes:
Result: {//was Part1&Part2; flattening semantics:
  class method S message()=S"42"
  class method S quoteMessage()=
    S"The message is: '"++this.message()++S"'."
  }
```

Mental model

```
Library code={ class method Void foo()=Debug(S"hi") }
code.foo()//wrong, 'code' is Library, is not compiled yet
A: { class method Library code()={ method Void foo()=Debug(S"hi") } }
B: A.code()//here I select the code for compilation!
Main: B.foo()//this works and prints "hi"

A: {../*code of A, all manually written*/} //boring boilerplate

A: MyDecorator <>< {/*essential code of A*/}//compact, correct

APartial: {/*essential code of A*/}//wrong, the decorator needs
A: MyDecorator <>< APartial         //the code, not the class name

AT: Trait({/*essential code of A*/})//traits can be named
A: MyDecorator <>< AT.code() //their code can be recovered

{..} -->Library
Trait({..}) --> Trait  //Wraps the input code to support operations.
Trait(a).code() == a
Decorator <>< Library --> Library //Enrich library with more code
Trait & Trait --> Trait //symmetric composition, matches by member name
Trait & Library --> Library // -what are the possible errors?
```

Modularisation with traits

```
Game: {//example game code, NOT MODULARISED
  Item: {interface     Point point     method Item break()}//as before
  Rock: Data <>< {implements Item     Num weight      method break()=Rock(..)}
  Wall: Data <>< {implements Item     Num height      method break()=Rock(..)}

  Map: Data <>< {//map implementation by Bob
    method Item get(Point that)=..     method Void set(Item that)=..
    ..
    }
  class method Void run()=..this.load(..).. //implemented by Bob

  class method Map load(S fileName)={..//Alice writes load(_)
    //create empty map,
    //read from file and divide in lines,
    //for all the lines, call load(map,line)
    }
  class method Void load(Map map,S line)={//example line:  S"Rock 23 in 12, 7"
    ns=line.readNums()
    if line.startsWith(S"Rock") (
      map.set(Rock(weight: ns.get(0\),point: Point(x: ns.get(1\),y: ns.get(2\))))
      )
    if line.startsWith(S"Wall") (..)
    ..
    }
  }
Main: Game.run()
```

Modularised code: Alice <–> Bob

```
Bob: Trait({//all as before, but load(fileName) is abstract
  Item: {interface     Point point     method Item break()}
  Rock: Data <>< {implements Item     Num weight     method break()=Rock(..)}
  Wall: Data <>< {implements Item     Num height     method break()=Rock(..)}
  Map: Data <>< {/*map implementation by Bob*/}
  class method Void run()=..this.load(..).. //implemented by Bob
  class method Map load(S fileName)
  })
Alice: Trait({
  Map: {class method Map()     method Void set(Item that)}
  Item: {interface}
  Rock: {implements Item     class method Rock(Num weight, Point point) }
  Wall: {implements Item     class method Rock(Num height, Point point) }

  class method Map load(S fileName)=..//Alice writes load(_)
    /*exactly the same code of before call read(line)*/
  class method Void load(Map map,S line)={//example line:  S"Rock 23 in 12, 7"
    ns=line.readNums() //exactly the same code of before
    if line.startsWith(S"Rock") (
      map.set(Rock(weight: 1ns,point: Point(x: 2ns,y: 3ns)))
      )
    if line.startsWith(S"Wall") (..)
    ..                                    //Alice and Bob are now independent.
    }                                     //Map,Item,Rock inside of Alice are
  })                                      //unrelated with Map,Item,Rock inside Bob.
Game: Alice & Bob & {}         // & merges the members with the same name.
Main: Game.run()               //The flattened result is exactly the code of before.
```

Modularized code=testable code

```
Alice: Trait({/*as before*/})
AliceMock: Alice & {
  Item: {interface     Point point}
  Rock: Data <>< {implements Item     Num weight}

  Map: Trait(Collections.hashmap(key: Point, val: Item)) & {
    method Void set(Item that)=this.put(key: that.point(),val: that)
    }

  class method Void test(S fileName,S expected)={
    map=this.load(fileName: fileName)
    Debug.test(map, expected: expected)
    }
Test1: AliceMock.test(S"justARock.txt",S"HashMap[Point(..)->Rock(..)]"
Test2: ..
..
```

Those tests are completely independent of Bob code, and can be run even before Bob started writing any code. By composing independently testable code, you can put you project together after every component has been independently tested. You can (and should) do it also in Java. However, it is so much harder to do so.

Dependency injection Hell in Java

```
//common code:
interface Map{..} class Point{..}//most of those requires their own file
interface Item{..} interface Rock extends Item {..}..
interface ItemFactory{Rock makeRock(Point point, int weight); ..}
interface MapFactory{Map makeMap();}
//Alice code:
class MapLoader{
  ItemFactory items; MapFactory maps;
  MapLoader(ItemFactory i,MapFactory m){items=i;maps=m;}
  Map load(String fileName){..maps.makeMap()..}
  Void load(Map map,String fileName){..items.makeRock(..)..}
  }
//Alice mocking code
class MockMap implements Map{..}
class MockMapFactory implements MapFactory{//this may be a lambda in Java8
  public Map makeMap(){return new MockMap();}  }
class MockItemFactory implements ItemFactory{//but this can not
  public Rock makeRock(..){return new MockRock(..);}
  public Wall makeWall(..){return new MockWall(..);}  } ..
class MockRock implements Rock{..} class MockWall implements Wall{..}
class Tester{
  static void test(String fileName,String expected)={
    MapLoader m=new MapLoader(new MockMapFactory(),new MockItemFactory());
    Map map=m.load(fileName)
    assert map.toString().equals(expected);
    }
```

Redirect

```
MathTrait: Trait({
  N: AbstractNumeric &{}
  ListN: Collections.vector(of: N)
  class method ListN factorize(N that)=...
   })
MathNum: MapTrait"N"<=Num & {} /*alternative syntax:*/ MapTrait[\"N" into: Num]
MathDouble: MapTrait"N"<=Double  & {}

MapTest: MapTrait & {
  N: Data <>< { Num inner
    ..//forward '+', '/', '*', '-' etcetera to this.inner()
    //but do some checks and logging
   Log: {..}
  }
  class method Void test(N that)={
    N.Log.reset()
    res=This.factorize(that)
    Debug.test(res,expected: ..)
    Debug.test(N.Log.report(),expected: ..)
    }
  }
```

sum or redirect?

```
AbstractService: Trait({..}) //abstrat signatures (as in the trait 'Common')
ServiceImplT: AbstractService & Trait({..})//concrete implementation

AT: Trait({Ser: AbstractService &{} ..})
BT: Trait({Ser: AbstractService &{} ..})

//option 1:
A: AT & ServiceImplT & {}
B: BT & ServiceImplT & {}
//now A.Ser !=B.Ser, even if they may have the same implementation

//option 2:
Service: ServiceImplT &{}
A: AT"Ser"<=Service & {}
B: BT"Ser"<=Service & {}
//now neither A.Ser nor B.Ser exists; A and B both refers to Service
```

That is, in 42 you can modularize any arbitrary piece of code by just declaring abstract signatures.
If the code satisfying your requirement is inside of another trait, use `&`.
Otherwise, if the code you want is declared outside, use redirect.

More trait features

```
myTrait.abstract() //make the code all abstract and remove private members

myTrait[\"foo()" of: \"A" into: \"bar()";\"baz(x)"  into: \"beer(y)"]
//rename method A.foo() into A.bar(), and top-level baz(x) into beer(y)

myTrait[abstract: \"baz()"] //make method abstract

myTrait[hide: \"baz()"] //make method private

trait1.showOnly(trait2)//make everything private, except things in 'trait2'

//sometime it is convenient to examine code as in Java reflection.
//Traits offer methods for that, for example
retType=myTrait.method(\"foo()").returnType()

//and we can use this with redirect!
myTrait"T"<=retType
```

Excercise

Make a class decorator that behaves like **`Data`**, except it uses a specified field/no-arg-method to implement **`Concepts.ToS`**`.toS()`.

```
ToSTrait: Trait({implements Concepts.ToS
  T: {implements Concepts.ToS}
  T f
  method toS()=this.f().toS()
  })
Answer: Data <>< {Selector that
  method Library <><(Library right)={
    fieldType=Trait(right).method(this.that()).returnType()
    t=ToSTrait[\"f()" into: this.that()]"T"<=fieldType
    return Data <>< (t & right)
    // data will do its job, except for toS, since it is already defined
    }
  }

//usage
MyClass: Answer(\"inner()") <>< {Point inner}
Main: Debug(  MyClass(inner: Point(x: 3\,y: 7\))  ) //will print Point(x:  3,y:  7)
```

Compact and typesafe connection DB –> GUI

```
{reuse  L42.is/AdamsTowel
//load libraries
Db: Load <>< {reuse L42.is/DB}
Gui: Load <>< {reuse L42.is/2dGui}

//specialise the Db library to work over the VIC Students database
VicDb: Db"..my connection string.."

//VicDb knows all its tables and their shapes.  So we can have a vector of Students
StudentList: Collections.vector(of: VicDb.Student)
QueryFrom: VicDb(//prepared query
  query: Db.SQL"Select * from Student where country=@country"
  result: StudentList)//and have queries returning StudentList

//Gui lets us create a widget for a tabular representation of vectors of Data classe
StudentsView: Gui.table(title: S"selected students" content: StudentList)

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

Thanks!

Questions?