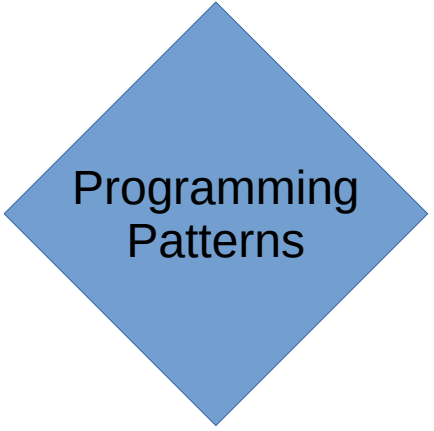
SWEN423 - Lecture 15



Nested Trait composition

42: Nested Traits composition

- 42 (Marco S): a new programming language designed in VUW ECS.
  It is in good company: Grace (James N), Whiley (David P), DeepJava (Thomas K), Wivern (Alex P)

- The goal of 42 is to allow safe interactions of many different libraries at the same time.
- So, safety + code reuse
- Here we are going to focus on the code reuse part.

42: Hello world

```
reuse [AdamTowel]
Main1= Debug(S"Hello world")
Main2= Debug(S"Hello world, again")
```

- First, we import AdamTowel. Towels are just like standard libraries; but since there are many different ones, you name the one you are using.
- Crucially, it is easy to make new towels; usually by building over existing ones.
- Modules developed by a specific towel can be reused on a range of other compatible towels.

42: Hello world

```
reuse [AdamTowel]
Main1= Debug(S"Hello world")
Main2= Debug(S"Hello world, again")
```

- Then, we have our main Main1.
- Note how the string literal report the type of the string.
- In 42 many classes accept the “string literal” syntax, and ‘S’ is just one of those (e.g. `Url"www.google.com"`).
- Debug prints on standard output. There is a simple File system module to work with other files.

42: Hello world

```
reuse [AdamTowel]
Main1= Debug(S"Hello world")
Main2= Debug(S"Hello world, again")
```

- Finally, we have a SECOND MAIN
- In 42 you can have as many mains you want, just list them one after another.
- A typical use of multiple mains is for testing, where every main can run a different test suite.

42: Hello world

```
reuse [AdamTowel]
Point=Data:{Num x, Num y}
Points=Collection.list(Point)
Main0=(
  p1=Point(x=3Num,y=5Num)
  p2=Point(x=8Num,y=12Num)
  Debug(p1)        //prints Point(x=3, y=5)
  ps=Points[p1;p2;p1]
  Debug(ps)        //prints [Point(x=3, y=5); Point(x=8, y=12); Point(x=3, y=5)]
  var acc=S""
  for p in ps ( acc++=p.x().toS() )
  Debug(acc)       //prints 383
  )
```

- Here you can see more code
- We declare a Point class with x and y fields of type Num
- We declare a Points class as a list of Points

42: Compilation process

```
reuse [AdamTowel]
Point=Data:{Num x, Num y}
Points=Collection.list(Point)
Main0=(..)
```

- How is this working?
- Is `Data` a keyword? Is `Collection` a keyword?
- No, they are just a normal class names, like `s` and `Num`.
- `Data:{Num x, Num y}` looks like a class declaration of sort, but `Collection.list(Point)` looks like a static method call.
- The syntax to declare a Main looks very similar to the one to declare a class.
- What is going on here?

42: Compilation process

```
reuse [AdamTowel]
Point=Data:{Num x, Num y}
Points=Collection.list(Point)
Main0=(..)
```

- `{Num x, Num y}` is a Library literal: a unit of code, like the body of a class, but without a name.
- In 42 Libraries are first class values: they are objects that can be passed around.
- 42 supports operator overloading, (like Python,C#,C++ etc); thus ':' is just another way to write a method call
- `Data:{Num x, Num y}` is similar to `Data.decorate({Num x, Num y})`
- Data is a 'class decorator', and ':' is the decorator operator.
- A decorator is just a class with a ':' method that can take code and improve it the way it prefers.
- In the case of Data, it adds toS(), constructors, ==, != and many other utility methods that just depend from the set of fields.

42: Compilation process

```
reuse [AdamTowel]
Point=Data:{Num x, Num y}
Points=Collection.list(Point)
Main0=(..)
```

- The code above have 3 mains: Point, Points and Main0.
- The mains return Library literals
- The mains are executed in order.
- The expression in the main Point produce the code for the Point class.
- The next main can USE the code of Point to create Points.
- The last main can use the code of Point and Points.

Iterative Compilation

```
reuse [AdamTowel]
Point=Data:{Num x, Num y}                <--
Points=Collection.list(Point)
Main0=(..)
```

- First some mains are executed. They produce code.
- If their code can be typed (dependencies are available), it is type checked.
- A mains can use all the code already produced/typechecked.
- There is no difference between compilation and execution.

Iterative Compilation

```
reuse [AdamTowel]
Point={Num x, Num y, class method This(Num x, Num y) ..}
Points=Collection.list(Point)          <--
Main0=(..)
```

- First some mains are executed. They produce code.
- If their code can be typed (dependencies are available), it is type checked.
- A mains can use all the code already produced/typechecked.
- There is no difference between compilation and execution.

Iterative Compilation

```
reuse [AdamTowel]
Point={Num x, Num y, class method This(Num x, Num y) ..}
Points={class method This() method Num size().. }
Main0=(..)                                    <--
```

- First some mains are executed. They produce code.
- If their code can be typed (dependencies are available), it is type checked.
- A main can use all the typechecked code above.
- This is different from the Java/C model of first compile, then execute.

Iterative Compilation

```
MyDecorator=.. //first you define a decorator that help you to define more code
A=MyDecorator:{..}//then you can use it to define the classes you actually need
B=MyDecorator:{..}//the decorator can generating the repetitive code for you
C=MyDecorator:{..}
Main0=(../*uses A,B,C*/)
```

- Decorators serves a similar task of “macros”, they helps you to define large classes by providing much smaller code in input.
- However, they are just normal classes/methods written in the base language.
- They are designed so that it is easy/easier to track down when something goes wrong, since they work at a very high level, they do not touch the code directly, but work using operations similar to extends and refactoring, provided by the “Trait” decorator

Traits in 42

```
T1= Trait:{ class method S msg()=S"Hello world" }
T2= Trait:{
   class method Void printMsg()=Debug(this.msg())
   class method S msg() //no body == abstract method
   }
T=T1:T2 //T is another trait
C=Class:T1:T2   //equivalent to Class:(T1:T2), equivalent to Class:T
                //similar to Class().decorate(T1().decorate(T2()))
Main0=C.printMsg() //prints hello world
```

- Trait is a decorator, that wraps a Library into a Trait instance, offering useful operations to compose code.
- A trait can decorate other traits or Library by summing its code with the code of the other library.
- Class is another decorator, that takes a Trait and return the corresponding library. It also checks that the result is coherent (similar to non-abstract in Java)

Traits in 42

```
T1= Trait:{ class method S msg()=S"Hello world" }
T2= Trait:{
   class method Void printMsg()=Debug(this.msg())
   class method S msg() //no body == abstract method
   }
T=T1:T2['printMsg()=>'myMsg()]  //rename is another trait operation
C=Class:T[hide='msg()]  //hide is a form of rename;
//operator precendece:   a:b[c] == a:(b[c])
Main0=C.myMsg() //prints hello world
```

- Trait can tweak the code in many different ways:
  strong rename, (as eclipse refactoring)
  weak rename, (leave the abstract signature behind)
  hide, (make private)
  clear (make abstract)
  and many more

Html Nodes

```
Base=Trait:{
  Node={interface}
  Nodes=Collection.list(Node)
  P=Data:{[Node] S text}
  H1=Data:{[Node] S text}
  Div=Data:{[Node] Nodes nodes}
  Divs=Collection.list(Div)
  Head=Data:{[Node]}
  Body=Data:{[Node] Divs divs}
  Html=Data:{[Node] Head head,Body body}
  A=Data:{[Node]S href, S text}
  }
```

- We can define a composite very easily in 42.
- Just use Data on all the datavariants.
- The syntax “[Node]” is like the Java “implements Node”
- However, in this way we do not have the Flyweight pattern.

Html Nodes

- Data supports the flyweight, but we need to pass the right parameters.
- Oh, no, how the code is horrible and repetitive!

```
Base=Trait:{
  Node={interface}
  Nodes=Collection.list(Node)
  P=Data('This,autoNorm=Bool.true()):{[Node] S text}
  H1=Data('This,autoNorm=Bool.true()):{[Node] S text}
  Div=Data('This,autoNorm=Bool.true()):{[Node] Nodes nodes}
  Divs=Collection.list(Div)
  Head=Data('This,autoNorm=Bool.true()):{[Node]}
  Body=Data('This,autoNorm=Bool.true()):{[Node] Divs divs}
  Html=Data('This,autoNorm=Bool.true()):{[Node] Head head,Body body}
  A=Data('This,autoNorm=Bool.true()):{[Node]S href, S text}
  }
```

Html Nodes

- Data supports the flyweight, but we need to pass the right parameters.
- Oh, no, how the code is horrible and repetitive!

```
Base=Trait:{
  Node={interface}
  Nodes=Collection.list(Node)
  P=Data('This,autoNorm=Bool.true()):{[Node] S text}
  H1=Data('This,autoNorm=Bool.true()):{[Node] S text}
  Div=Data('This,autoNorm=Bool.true()):{[Node] Nodes nodes}
  Divs=Collection.list(Div)
  Head=Data('This,autoNorm=Bool.true()):{[Node]}
  Body=Data('This,autoNorm=Bool.true()):{[Node] Divs divs}
  Html=Data('This,autoNorm=Bool.true()):{[Node] Head head,Body body}
  A=Data('This,autoNorm=Bool.true()):{[Node]S href, S text}
  }
```

Html Nodes

```
Case=Decorator:{
  method Trait decorate(Trait trait)[Message.Guard]
    =Data('This,autoNorm=Bool.true()):trait
  }

Base=Trait:{
  Node={interface}
  Nodes=Collection.list(Node)
  P=Case:{[Node] S text}
  H1=Case:{[Node] S text}
  Div=Case:{[Node] Nodes nodes}
  Divs=Collection.list(Div)
  Head=Case:{[Node]}
  Body=Case:{[Node] Divs divs}
  Html=Case:{[Node] Head head,Body body}
  A=Case:{[Node] S href, S text}
  }
```

- Using the Decorator decorator, we can define our own decorator, delegating to Data with parameters.
- Syntax [Message.Guard] is like throws Exception in java
- Now we can define a fully flyweight composite very easily.
- How can we add operations and cases in a library setting?

Html Nodes

```
ToHtml=Base:{
  Node={method S toHtml()}
  P={method S toHtml()=S"<P>%this.text()</P>"}
  H1={method S toHtml()=S"<H1>%this.text()</H1>"}
  Div={method S toHtml()={
    var ss=S""
    for e in this.nodes() (ss++=e.toHtml())
    return S"<Div>%ss</Div>"
    }}
  Head={method S toHtml()=S"<Head></Head>"}
  Body={method S toHtml()={
    var ss=S""
    for e in this.divs() (ss++=e.toHtml())
    return S"<Body>%ss</Body>"
    }}
  Html={method S toHtml()={
    h=this.head().toHtml()
    b=this.body().toHtml()
    return S"<Div>%h %b</Div>"
    }}
  }
```

- From an dynamically typed perspective,
  this should work!
- Just compose with code with nested classes with all the cases implemented for the method toHtml()
- Sadly, **it do not work in 42** because P do not know it will have a method .text() and so on for all the other cases.
- We could just repeat all the abstract signatures but it would be very repetitive code.

Introducing, the self method pattern

```
TA=Trait:{
  method This self()=this //self method
  method S baz(Num that)=(..)
  method Num beer()=(..)
  ..//more methods here
  }
TB=Trait({
  A=Class:TA
  method A self() //abstract self method
  method S doStuff()=
    this.self().baz(this.self().beer())
  ..//more methods here
  })['A=>'This]//renames A to the top level
```

- We introduce a method call self that returns ‘This’ in the trait that we want to reuse.
- Then, we can just import the Trait as a nested class, and we declare a self() method returning such nested class.
- Now calling the abstract method ‘self()’ we can convince the type system that ‘this’ is also of type ‘A’.
- Finally we rename A to the top level, so that the abstract self method is implemented by the one in A.
- A variation of this pattern is also very common with Java Generics.

Introducing, the self method pattern

```
Case=Decorator:{
  HasSelf=Trait:{method This self()=this}
  method Trait decorate(Trait trait)[Message.Guard]
    =Data('This,autoNorm=Bool.true()):HasSelf:trait
  }
Base=Trait:{//base is defined exactly as before
  Node={interface}
  Nodes=Collection.list(Node)
  P=Case:{[Node] S text}
  H1=Case:{[Node] S text}
  Div=Case:{[Node] Nodes nodes}
  Divs=Collection.list(Div)
  Head=Case:{[Node]}
  Body=Case:{[Node] Divs divs}
  Html=Case:{[Node] Head head,Body body}
  A=Case:{[Node] S href, S text}
  }
Abs=Base[clear='This]//a fully abstract Base
```

- We can implement self() for all the cases by simply tweaking the ‘Case’ decorator.
- We also define a variation of Base, where all the methods has been made abstract. This will help to compose implementations from multiple sources

```
ToHtml=Trait({
  Dom=Class.Relax:Abs:{Node={method S toHtml()}}
  P={method Dom.P self()
    method S toHtml()=S"<P>%this.self().text()</P>"}
  H1={method Dom.H1 self()
    method S toHtml()=S"<H1>%this.self().text()</H1>"}
  Div={method Dom.Div self()
    method S toHtml()={
      var ss=S""
      for e in this.self().nodes() (ss++=e.toHtml())
      return S"<Div>%ss</Div>"}}
  Head={method S toHtml()=S"<Head></Head>"}
  Body={method Dom.Body self()
    method S toHtml()={
      var ss=S""
      for e in this.self().divs() (ss++=e.toHtml())
      return S"<Body>%ss</Body>"}}
  Html={method Dom.Html self()
    method S toHtml()={
      h=this.self().head()   b=this.self().body()
      return S"<Div>%h.toHtml() %b.toHtml()</Div>" }}
})['Dom=>'This]
```

ToHtml

- Then, for all the cases we only implement the single method self, where P self returns Dom.P, H1 self returns Dom.H1
  and so on.
- It is not perfect; there is some repetition across the cases, but it is manageable.

```
ToHtml=Trait({
  Dom=Class.Relax:Abs:{Node={method S toHtml()}}
  P={method Dom.P self()
    method S toHtml()=S"<P>%this.self().text()</P>"}
  H1={method Dom.H1 self()
    method S toHtml()=S"<H1>%this.self().text()</H1>"}
  Div={method Dom.Div self()
    method S toHtml()={
      var ss=S""
      for e in this.self().nodes() (ss++=e.toHtml())
      return S"<Div>%ss</Div>"}}
  Head={method S toHtml()=S"<Head></Head>"}
  Body={method Dom.Body self()
    method S toHtml()={
      var ss=S""
      for e in this.self().divs() (ss++=e.toHtml())
      return S"<Body>%ss</Body>"}}
  Html={method Dom.Html self()
    method S toHtml()={
      h=this.self().head()   b=this.self().body()
      return S"<Div>%h.toHtml() %b.toHtml()</Div>" }}
  })['Dom=>'This]
```

ToHtml

- Then, for all the cases we only implement the single method self, where P self returns Dom.P, H1 self returns Dom.H1
  and so on.
- It is not perfect; there is some repetition across the cases, but it is manageable.

```
ToHtml=Trait({
  Dom=Class.Relax:Abs:{Node={method S toHtml()}}
  P={method Dom.P self()
    method S toHtml()=S"<P>%this.self().text()</P>"}
  H1={method Dom.H1 self()
    method S toHtml()=S"<H1>%this.self().text()</H1>"}
  Div={method Dom.Div self()
    method S toHtml()={
      var ss=S""
      for e in this.self().nodes() (ss++=e.toHtml())
      return S"<Div>%ss</Div>"}}
  Head={method S toHtml()=S"<Head></Head>"}
  Body={method Dom.Body self()
    method S toHtml()={
      var ss=S""
      for e in this.self().divs() (ss++=e.toHtml())
      return S"<Body>%ss</Body>"}}
  Html={method Dom.Html self()
    method S toHtml()={
      h=this.self().head()   b=this.self().body()
      return S"<Div>%h.toHtml() %b.toHtml()</Div>" }}
})['Dom=>'This]
```

ToHtml

- Then, for all the cases we only implement the single method self, where P self returns Dom.P, H1 self returns Dom.H1
  and so on.
- It is not perfect; there is some repetition across the cases, but it is manageable.

```
ToHtml=Trait({
  Dom=Class.Relax:Abs:{Node={method S toHtml()}}
  P={method Dom.P self()
    method S toHtml()=S"<P>%this.self().text()</P>"}
  H1={method Dom.H1 self()
    method S toHtml()=S"<H1>%this.self().text()</H1>"}
  Div={method Dom.Div self()
    method S toHtml()={
      var ss=S""
      for e in this.self().nodes() (ss++=e.toHtml())
      return S"<Div>%ss</Div>"}}
  Head={method S toHtml()=S"<Head></Head>"}
  Body={method Dom.Body self()
    method S toHtml()={
      var ss=S""
      for e in this.self().divs() (ss++=e.toHtml())
      return S"<Body>%ss</Body>"}}
  Html={method Dom.Html self()
    method S toHtml()={
      h=this.self().head()   b=this.self().body()
      return S"<Div>%h.toHtml() %b.toHtml()</Div>" }}
  })['Dom=>'This]
```

ToHtml

- Then, for all the cases we only implement the single method self, where P self returns Dom.P, H1 self returns Dom.H1
  and so on.
- It is not perfect; there is some repetition across the cases, but it is manageable.

```
ToHtml=Trait({
  Dom=Class.Relax:Abs:{Node={method S toHtml()}}
  P={method Dom.P self()
    method S toHtml()=S"<P>%this.self().text()</P>"}
  H1={method Dom.H1 self()
    method S toHtml()=S"<H1>%this.self().text()</H1>"}
  Div={method Dom.Div self()
    method S toHtml()={
      var ss=S""
      for e in this.self().nodes() (ss++=e.toHtml())
      return S"<Div>%ss</Div>"}}
  Head={method S toHtml()=S"<Head></Head>"}
  Body={method Dom.Body self()
    method S toHtml()={
      var ss=S""
      for e in this.self().divs() (ss++=e.toHtml())
      return S"<Body>%ss</Body>"}}
  Html={method Dom.Html self()
    method S toHtml()={
      h=this.self().head()   b=this.self().body()
      return S"<Div>%h.toHtml() %b.toHtml()</Div>" }}
})['Dom=>'This]
```

ToHtml

- Then, for all the cases we only implement the single method self, where P self returns Dom.P, H1 self returns Dom.H1
  and so on.
- It is not perfect; there is some repetition across the cases, but it is manageable.

```
ToHtml=Trait({
  Dom=Class.Relax:Abs:{Node={method S toHtml()}}
  P={method Dom.P self()
    method S toHtml()=S"<P>%this.self().text()</P>"}
  H1={method Dom.H1 self()
    method S toHtml()=S"<H1>%this.self().text()</H1>"}
  Div={method Dom.Div self()
    method S toHtml()={
      var ss=S""
      for e in this.self().nodes() (ss++=e.toHtml())
      return S"<Div>%ss</Div>"}}
  Head={method S toHtml()=S"<Head></Head>"}
  Body={method Dom.Body self()
    method S toHtml()={
      var ss=S""
      for e in this.self().divs() (ss++=e.toHtml())
      return S"<Body>%ss</Body>"}}
  Html={method Dom.Html self()
    method S toHtml()={
      h=this.self().head()   b=this.self().body()
      return S"<Div>%h.toHtml() %b.toHtml()</Div>" }}
})['Dom=>'This]
```

ToHtml

- Then, for all the cases we only implement the single method self, where P self returns Dom.P, H1 self returns Dom.H1
  and so on.
- It is not perfect; there is some repetition across the cases, but it is manageable.

Adding H2 and composing all together

```
WithH2=Trait({
  Dom=Class.Relax:Abs
  H2=Case:{[Dom.Node] S text}
  })['Dom=>'This]


JustToHtml=Class:Base:ToHtml

JustH2=Class:Base:WithH2

All=Class:Trait({
  Dom=Class.Relax:Base:WithH2:ToHtml
  H2={method Dom.H2 self()
    method S toHtml()
      =S"<H2>%this.self().text()</H2>"
    }
  })['Dom=>'This]
```

- To add a new datavariant, we can follow the same pattern
- It is then easy to get the Base with toHtml or the Base with H2.
- In order to add both toHtml and H2 we need to also specify how to turn H2 into html.

```
Clone=Trait({..})['Dom=>'This]
```

Clone

- Can not fit well on a slide. Let see it in pieces

```
Clone=Trait({
  Dom=Class.Relax:Abs:{
    Node={interface method This op()}
    Div={[Node] method This op()}
    Head={[Node] method This op()}
    Body={[Node] method This op()}
    }
..
  })['Dom=>'This]
```

Clone

- We define Dom as usual, and we add an ‘op()’ method.
- This method return type is refined by some cases, and we need to list them.
- ‘op()’ is our clone method

```
Clone=Trait({
  Dom=Class.Relax:Abs:{
    Node={interface method This op()}
    Div={[Node] method This op()}
    Head={[Node] method This op()}
    Body={[Node] method This op()}
    }
  Extra={
    class method S text(S that)=that
    class method S url(S that)=that
    }
..
  })['Dom=>'This]
```

Clone

- Extra will contains method to handle commo subcase cases that are not Nodes.
- This is not as natural as the encoding in the Java clone visitors

```
Clone=Trait({
  Dom=Class.Relax:Abs:{..}
  Extra={..}
  P={method Dom.P self()
    method Dom.Node op()=
      Dom.P(text=Extra.text(this.self().text()))}
  H1={method Dom.H1 self()
    method Dom.Node op()=
      Dom.H1(text=Extra.text(this.self().text()))}
  Div={method Dom.Div self()
    method Dom.Div op()={
      ns=this.self().nodes()
      return Dom.Div(nodes=
        Dom.Nodes()(for e in ns \add(e.op()))
      }}
  ..
  })['Dom=>'This]
```

Clone

- Then we define op for all the dataviariants.
- This is very similar to what we did for toHtml().
- The 42 code in Div serves a similar role to a python list comprehension.

```
Clone=Trait({
  Dom=Class.Relax:Abs:{..}
  Extra={..}
  P={..}
  H1={..}
  Div={..}
  Head={method Dom.Head self()
   method Dom.Head op()=this.self()}
  Body={method Dom.Body self()
    method Dom.Body op()=Dom.Body(divs=Dom.Divs()(
      for e in this.self().divs() \add(e.op())
    ))}
  Html={method Dom.Html self()
    method Dom.Html op()=Dom.Html(
      head=this.self().head().op()
      body=this.self().body().op())}
  A={method Dom.A self()
    method Dom.Node op()=Dom.A(
      href=Extra.url(this.self().href())
      text=Extra.text(this.self().text()))}
  })['Dom=>'This]
```

Clone

- Here we define the rest of the cases.
- Note how we follow a predictable pattern
- Finally, as usual, we merge Dom with the top level

Use Clone to define transformations

```
ToItalian=Trait({
  Dom=Class.Relax:Clone
    ['Node.op()=>'Node.toItalian()]
    [clear='Extra.text(that)]
  Extra={ //we override Extra.text
    class method S text(S that)
      =S"Pizza!!!"++that
    }
  })['Dom=>'This][hide='Extra]
```

- As happened in Java, once we have a way to clone with all the handles available, we can easily define transformations by overriding.
- Note how we manually ‘clear’ Extra.text(that)
- We also hide Extra, so that ToItalian can be merged with other operations

Use Clone to define transformations

```
PInDiv=Trait({
  Dom=Class.Relax:Clone
    ['Node.op()=>'Node.pInDiv()]
    ['P.pInDiv()->'P.superPInDiv()]
  P={method Dom.P self()//we override P.pInDiv
    method Dom.Node pInDiv(){//calling super
      p=this.self().superPInDiv()
      return Dom.Div(nodes=Dom.Nodes[p])
      }
    }
  })['Dom=>'This][hide='Extra]
```

- PInDiv pushes all Ps in an extra level of Div.
- Note how we use the soft rename to encode ‘super’

Concluding

- Still not as compact as the python multimethod solution, but it is in a statically typed setting.
- Benefit: the user of our operations see all methods as belonging to Node:

```
N=Class:Base:ToItalian:PInDiv
Main2=(
  dom=N.Div(nodes=N.Nodes[N.P(text=S"Hello");O.H1(text=S"World")])
  Debug(dom.toItalian())
  //prints N.Div(nodes=[N.P(text="Pizza!!!Hello"); N.H1(text="Pizza!!!World")])
  Debug(dom.pInDiv())
  //prints N.Div(nodes=[N.Div(nodes=[N.P(text="Hello")]); N.H1(text="World")])
  )
```

- Using more decorators we could make it a little more compact, but I’m trying to limit the content to 1 lecture.