

Functional Programming and the Scala Language

Lecture 10

Eugene Zouev
Innopolis University
Spring Semester 2018

- Abstract **vars** & getters/setters
- Lazy **vals**
- Anonymous classes & Structural subtyping
- ~~Enumerations~~

Abstract Members: Introduction

From the previous lecture

Java, C#:

abstract classes, abstract methods

C++:

abstract classes, "pure virtual" methods.

Scala: **more general approach:**

- abstract methods
- abstract var-variables
- abstract val-variables
- abstract **types**

M.Odersky:

A member of a class or trait is abstract if the member does not have a complete definition in the class.

Abstract Members: Example

From the previous lecture

Abstract trait

```
trait Abstract
```

```
{
```

```
  type T
```

```
  def transform(x: T): T
```

```
  val initial: T
```

```
  var current: T
```

```
  val x: String
```

```
}
```

Abstract type: to be made concrete in subclass(es)

Abstract method: uses abstract type

Abstract variables: use abstract type

Abstract val: uses concrete type

```
class Concrete extends Abstract
```

```
{
```

```
  type T = String
```

```
  def transform(x: String) = x + x
```

```
  val initial = "hi"
```

```
  var current = initial
```

```
  val x = "hello"
```

```
}
```

Abstract vals

From the previous lecture

```
trait Abstract
{
  ...
  val x: String
}
```

Abstract val; **no initialization**.
In subclasses **x** might get different values (depending on the need of each subclass)

```
class Concrete1 extends Abstract
{
  ...
  val x = "Hello"
}
```

```
class Concrete2 extends Abstract
{
  ...
  val x = "Bye-bye"
}
```

Important remark:
declaration of **x** looks very similar to abstract method!

```
def x: String
```

"vars" in Java

Before "abstract vars", consider usual vars.


```
class Example Java
{
    ...
    public int member;
}
```

Potentially unsafe:
client code has uncontrolled
access to `member`



Solution: accessors ("getters" and "setters")

```
class Example Java
{
    ...
    private int member;
    public int getMember() { return member; }
    public void setMember(int newVal)
        { if (checkNewVal()) member = newVal; }
}
```



This code should check validity
of changing `member`'s value

C# Solution

Special syntax for accessors: **properties**

```
class Example C#
{
    ...
    private int _member;
    public int member {
        get { return _member; }
        set { if (checkNewVal()) _member = value; }
    }
}
```

Auto-generated properties
for simple cases like read-only feature

```
class Example
{
    ...
    public int member { get; private set; }
}
```

```
Example v = new Example();
...
int x = v.member;    // OK
v.member = 77;       // Error
```

C#

C# Solution

Properties under the hood

```
class Example
{
    ...
    private int _member;
    public int member {
        get { return _member; }
        set { if (checkNewVal()) _member = value; }
    }
}
```

C#

```
class Example
{
    ...
    private int _member;
    public int $get() { return _member; }
    public void $set(int value)
        { if (checkNewVal()) member = value; }
}
```

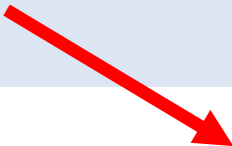
```
Example v =
    new Example();
...
int x = v.member;
v.member = 77;
```

```
Example v =
    new Example();
...
int x = v.$get();
v.$set(77);
```


vars: Scala Approach

The rule: for each var-member **two auxiliary methods** are associated with it: for reading and for writing.


```
class Time
{
  var hour = 12
  var minute = 0
}
```




The reading method has **the same name** as the var-member itself

The writing method has **the name** of form ***name*_** where *name* is the name of the var-member.


Two internal **vars** that keep values of **hour** & **minute**



private[this] means that the **var** can be updated only from within the class



Two methods for each **var** are generated automatically




A unique name

```
class Time
{
  private[this] var $1 = 12
  private[this] var $2 = 0

  def hour: Int = $1;
  def hour_=(x: Int) = { $1 = x }

  def minute: Int = $2
  def minute_=(x: Int) = { $2 = x }
}
```



vars: Scala Approach

User-defined getters & setters:

As a smarter replacement for a ~useless default solution

```
class Time
{
  private[this] var h = 12

  def hour: Int = h
  def hour_=(x: Int) = {
    require(0 <= x && x < 24)
    h = x
  }

  private[this] var m = 0

  def minute: Int = m
  def minute_=(x: Int) = {
    require(0 <= x && x < 60)
    m = x
  }
}
```

A reasonable equivalent
to the default
generation

C# vs Scala:

Scala provides the same
functionality but without
additional syntax
(M.Odersky)

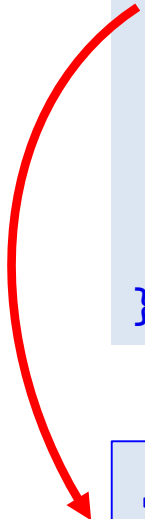
vars: Scala Approach

Getters/setters without members

```
class Thermometer
{
  var celsius: Float = _

  def fahrenheit = celsius*9/5 + 32
  def fahrenheit_= (f: Float) { celsius = (f-32)*5/9 }

  override def toString = fahrenheit + "F/" + celsius + "C"
}
```



```
...
var celsius: Float = _
...
```

`celsius` gets the default value: `0` for numeric types, `false` for `Boolean`, and `null` for reference types

Why underscore?

- To distinguish between non-abstract and abstract definition

```
var celsius: Float
```

This is abstract definition

Abstract vars

Abstract **vars**: only name & type are defined,
but not initial value.

Abstract vars...

```
trait AbstractTime
{
  var hour: Int
  var minute: Int
}
```



...get compiled to pairs of
getters/setters

```
trait AbstractTime
{
  def hour: Int
  def hour_=(x: Int)

  def minute: Int
  def minute_=(x: Int)
}
```

Lazy vals

How usual `vals` behave:

```
class Demo
{
  val m = "done"
}
```

Primary constructor
performs initialization



```
Demo d = new Demo()
```

m gets initialized by "done"



```
... d.m ...
```

m is equal to "done"



Illustration:

```
class Demo
{
  val m = { println("initializing m"); "done" }
}
```

```
Demo d = new Demo()
```

prints "initializing m"



```
... d.m ...
```

m is equal to "done"



Lazy vals

How usual `val`s behave:

```
class Demo
{
  val m = "done"
}
```

Primary constructor performs initialization

It happens while object creation, **before** any access to `m`

How "lazy" `val`s behave:

```
class Demo
{
  lazy val m = "done"
}
```

Initialization is performed while the first access to `m` is being done

```
Demo d = new Demo()
```

`m` is not initialized! - it's postponed until the first access to `m`

```
... d.m ...
```

`m` gets initialized just before the very first access to it

Lazy vals

How “lazy” vals behave:

```
class Demo
{
  lazy val m = { println("initializing m"); "done" }
}
```

```
Demo d = new Demo()
```

← No output: `m`'s initialization is not performed

```
... d.m ...
```

← `m` gets initialized here

What's the difference between lazy vals and defs?

```
class Demo
{
  def m = { println("initializing m"); "done" }
}
```

Lazy vals: pros & cons

Why “lazy” vals:

An important property of lazy vals is that **the textual order of their definitions does not matter**, because **values get initialized on demand**.

M. Odersky

Therefore, lazy vals can free you as a programmer from having to think hard how to arrange val-definitions to ensure that everything is defined when it is needed.

Problems with “lazy” vals:

“Laziness” is a typical feature of functional languages; e.g., Haskell is completely “lazy” 😊

...However, this advantage holds only as long as the initialization of lazy vals **neither produces side effects nor depends on them**. In the presence of side effects, initialization order **starts to matter**.

The task:

- Write an example with several **lazy** definitions so that the order of their initialization **differs** from the order of the definitions.
- Write an example when the order of definitions **matters**.

Structural Subtyping

Food and cows: The solution

```
class Food
abstract class Animal {
  type SuitableFood <: Food
  def eat(food: SuitableFood)
}
```

From the previous lecture

```
class Grass extends Food
class Cow extends Animal {
  type SuitableFood = Grass
  override def eat(food: Grass) = { }
}
```

The suitable food for cows is obviously grass

Another kind of use

Anonymous class representing **any** animal that eats grass - **not just Cow!!!!**

```
Animal { type SuitableFood = Grass }
```

Structural subtyping

The members in the curly braces specify (refine) the types of members from the base class.


Structural Subtyping

Having the **anonymous** class...

```
Animal { type SuitableFood = Grass }
```

...we can define another class:

```
class Pasture {  
  var animals:  
    List[Animal { type SuitableFood = Grass }] = Nil  
}
```



No need to give this class a name!

Structural Subtyping

One more example

```
def action[T, S](obj: T, operation: T=>S) =  
{  
  val result = operation(obj)  
  obj.close()  
  result  
}
```

The idea behind this code is to encapsulate two things: an **action** on an object and **closing** it after the action completes.

Here, `obj` can be of any type: no restrictions on `T`.

So, if `T` doesn't provide `close` method, the code won't compile!

```
def action[T <: { def close(): Unit }, S](obj: T, operation: T=>S) =  
{  
  val result = operation(obj)  
  obj.close()  
  result  
}
```

The solution: `T` must inherit any class implementing `close` method!