

# Computer Programming with Scala

## Week 2: Dealing with Complexity (OOP)

Martin Quinson  
November 2015



école  
normale  
supérieure

# Remember Last Week

## Computer Science and Informatics

- ▶ **Science of Abstraction:** building hierarchies of symbols and concepts  
Programming computers: surface activity, but the easiest to practice with
- ▶ **Computational Sciences:** simulation as third pillar (with observation & theory)

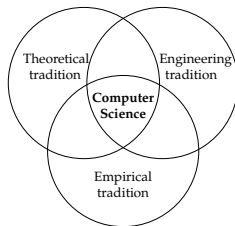
# Remember Last Week

## Computer Science and Informatics

- ▶ **Science of Abstraction:** building hierarchies of symbols and concepts  
Programming computers: surface activity, but the easiest to practice with
- ▶ **Computational Sciences:** simulation as third pillar (with observation & theory)

## The Historical Heritages of Computer Science

- ▶ **Maths:** proves necessary facts
- ▶ **Natural Sciences:** tests contingent facts
- ▶ **Engineering:** solves problems



**Programming Complex Systems is at the core of the discipline**  
(That's the topic of this course :)

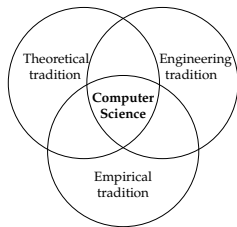
# Remember Last Week

## Computer Science and Informatics

- ▶ **Science of Abstraction:** building hierarchies of symbols and concepts  
Programming computers: surface activity, but the easiest to practice with
- ▶ **Computational Sciences:** simulation as third pillar (with observation & theory)

## The Historical Heritages of Computer Science

- ▶ **Maths:** proves necessary facts
- ▶ **Natural Sciences:** tests contingent facts
- ▶ **Engineering:** solves problems



**Programming Complex Systems is at the core of the discipline**  
(That's the topic of this course :)

**Scala:** Nice little language, that turns out to be a multi-paradigm beauty (beast?)

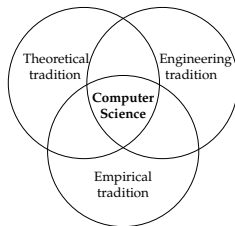
# Remember Last Week

## Computer Science and Informatics

- ▶ **Science of Abstraction:** building hierarchies of symbols and concepts  
Programming computers: surface activity, but the easiest to practice with
- ▶ **Computational Sciences:** simulation as third pillar (with observation & theory)

## The Historical Heritages of Computer Science

- ▶ **Maths:** proves necessary facts
- ▶ **Natural Sciences:** tests contingent facts
- ▶ **Engineering:** solves problems



**Programming Complex Systems is at the core of the discipline**  
(That's the topic of this course :)

**Scala:** Nice little language, that turns out to be a multi-paradigm beauty (beast?)

This week: **How to cope with complexity in programs?**

# What is the **Right Solution** for my Problem?

---

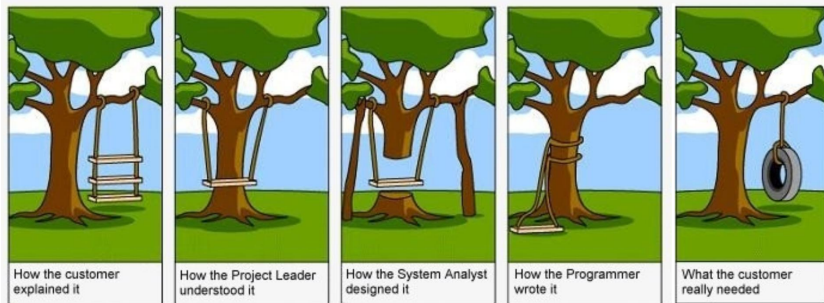
- ▶ The **Correct** one: provides the right answer
- ▶ The **Efficient** one: fast, use little memory, but also: fast write
- ▶ The **Simple** one: KISS! (acronym for *keep it simple, silly*)

# What is the **Right Solution** for my Problem?

- ▶ The **Correct** one: provides the right answer
- ▶ The **Efficient** one: fast, use little memory, but also: fast write
- ▶ The **Simple** one: KISS! (acronym for *keep it simple, silly*)

## Real Problems ain't easy

- ▶ They are **complex**: composed of several interacting entities
- ▶ They are **dynamic**: the specification evolves with the understanding



Turning the obvious into the useful is a living definition of the word *frustration*.

Alan J. Perlis

# Dealing with Complexity: Reductionism

*Divide each difficulty into as many parts as is feasible and necessary to resolve it.*

– René Descartes (1596-1650)

Composite Structure: Software System composed of manageable pieces

- ▶ The smaller the component, the simpler it is 😊
- ▶ The more parts, the more possible interactions ☹



# Dealing with Complexity: Reductionism

*Divide each difficulty into as many parts as is feasible and necessary to resolve it.*

– René Descartes (1596-1650)

Composite Structure: Software System composed of manageable pieces

- ▶ The smaller the component, the simpler it is 😊
- ▶ The more parts, the more possible interactions ☹

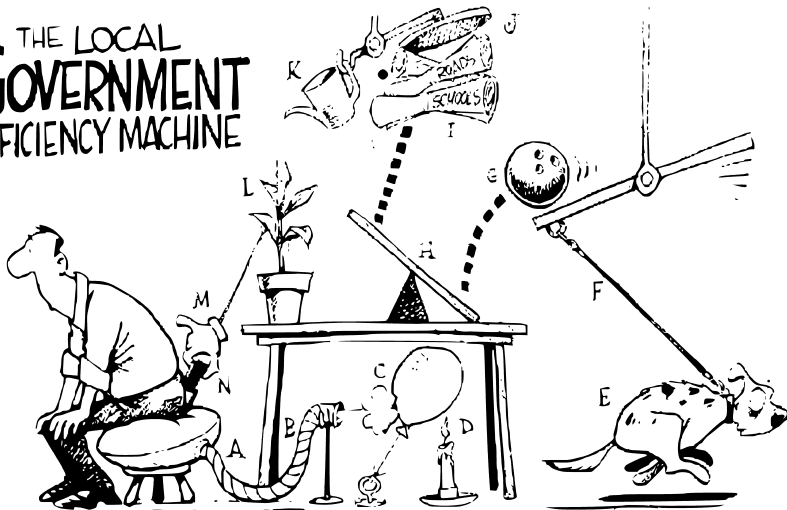
The Complexity Balance is important!

Good Composite Systems?

- ▶ Each component has a carefully specified function
- ▶ Components are easily integrated together
- ▶ Example: Audio speakers easily connected to the amplifier

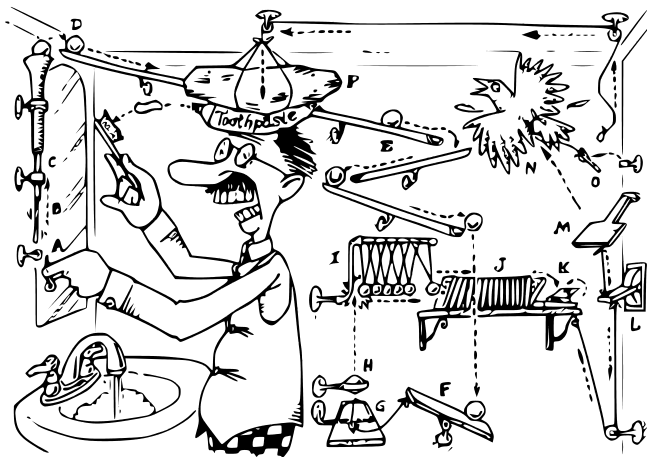
# Bad Composite Systems: Rube Goldberg Machines

## THE LOCAL GOVERNMENT EFFICIENCY MACHINE



- ▶ Utterly complex interactions to reach the point
- ▶ No definitive rule to avoid bad designs. Only bad smells.

# The Rube Goldberg's Toothpaste Dispenser

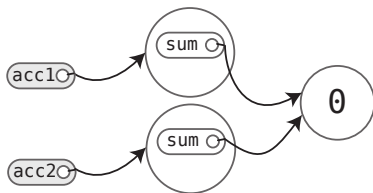


- ▶ Such **over engineered** solutions should obviously remain jokes
- ▶ **Right level of abstraction**: focus on relevant properties
- ▶ **Right composition**: focus on relevant components, that are easily integrated

# First OOP Principle: Object Encapsulation

- ▶ Group things that go together. Example: (x,y) of the point
- ▶ Give meaning to raw data. Example: an accumulator is (more than) an integer

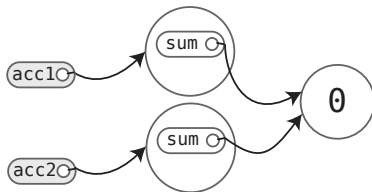
```
class Checksum {  
  var sum = 0  
}  
val acc1 = new Checksum  
val acc2 = new Checksum
```



# First OOP Principle: **Object Encapsulation**

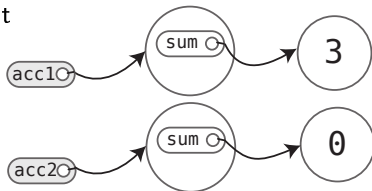
- ▶ Group things that go together. Example: (x,y) of the point
- ▶ Give meaning to raw data. Example: an accumulator is (more than) an integer

```
class Checksum {  
  var sum = 0  
}  
val acc1 = new Checksum  
val acc2 = new Checksum
```



- ▶ You can change the value of the content

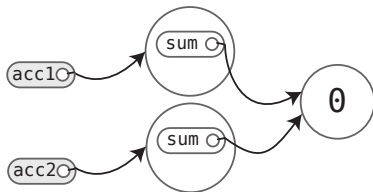
```
acc1.sum = 3
```



# First OOP Principle: Object Encapsulation

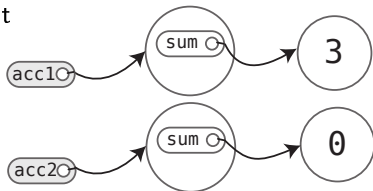
- ▶ Group things that go together. Example: (x,y) of the point
- ▶ Give meaning to raw data. Example: an accumulator is (more than) an integer

```
class Checksum {  
  var sum = 0  
}  
val acc1 = new Checksum  
val acc2 = new Checksum
```



- ▶ You can change the value of the content

```
acc1.sum = 3
```



- ▶ You still cannot change the `value` itself

```
scala> acc1 = new Checksum  
<console>:12: error: reassignment to val
```

# More OOP Syntax

## First version

```
class Checksum {  
  var sum = 0  
  def add(b: Int): Unit = {  
    sum += b  
  }  
  def checksum(): Int = {  
    return ~ (sum & 0xFF) + 1  
  }  
}
```

- ▶ Each Checksum object:  
1 variable + 2 methods

```
scala> val acc = new Checksum  
scala> acc.add(346634554)  
scala> println(acc.checksum)  
-58
```

# More OOP Syntax

## First version

```
class Checksum {  
  var sum = 0  
  def add(b: Int): Unit = {  
    sum += b  
  }  
  def checksum(): Int = {  
    return ~(sum & 0xFF) + 1  
  }  
}
```

- Each Checksum object:  
1 variable + 2 methods

```
scala> val acc = new Checksum  
scala> acc.add(346634554)  
scala> println(acc.checksum)  
-58
```

## Better version: less sugar

```
class Checksum {  
  var sum = 0  
  def add(b: Int): Unit = sum += b  
  def checksum(): Int = ~(sum & 0xFF) + 1  
}
```



# More OOP Syntax

## First version

```
class Checksum {  
  var sum = 0  
  def add(b: Int): Unit = {  
    sum += b  
  }  
  def checksum(): Int = {  
    return ~(sum & 0xFF) + 1  
  }  
}
```

- ▶ Each Checksum object:  
1 variable + 2 methods

```
scala> val acc = new Checksum  
scala> acc.add(346634554)  
scala> println(acc.checksum)  
-58
```

## Better version: less sugar

```
class Checksum {  
  var sum = 0  
  def add(b: Int): Unit = sum += b  
  def checksum(): Int = ~(sum & 0xFF) + 1  
}
```

## Even better version: hide your data

```
class Checksum {  
  private var sum = 0  
  def add(b: Int): Unit = sum += b  
  def checksum(): Int = ~(sum & 0xFF) + 1  
}
```

The **private** keyword hides class content from the outer world

```
scala> acc.sum = 3  
<console>:13: error: variable sum in class Checksum cannot be accessed in Checksum
```

# Some OOP Vocabulary

```
class Checksum {  
  private var sum = 0  
  def add(b: Int): Unit = sum += b  
  def checksum(): Int = ~(sum & 0xFF) + 1  
}
```

- ▶ `sum`: **field** or **member**
- ▶ `add`: **procedure** (does not return a value)
- ▶ `checksum`: **method** (returns a value)

# Some OOP Vocabulary

```
class Checksum {  
  private var sum = 0  
  def add(b: Int): Unit = sum += b  
  def checksum(): Int = ~(sum & 0xFF) + 1  
}
```

- ▶ `sum`: **field** or **member**
- ▶ `add`: **procedure** (does not return a value)
- ▶ `checksum`: **method** (returns a value)

```
acc.add(346634554)
```

- ▶ The whole is a **method call**
- ▶ `346634554`: method **parameter** (or explicit parameter)
- ▶ `acc`: method call **receiver** (or implicit parameter)

# Some OOP Vocabulary

```
class Checksum {  
  private var sum = 0  
  def add(b: Int): Unit = sum += b  
  def checksum(): Int = ~(sum & 0xFF) + 1  
}
```

- ▶ **sum**: **field** or **member**
- ▶ **add**: **procedure** (does not return a value)
- ▶ **checksum**: **method** (returns a value)

```
acc.add(346634554)
```

- ▶ The whole is a **method call**
- ▶ **346634554**: method **parameter** (or explicit parameter)
- ▶ **acc**: method call **receiver** (or implicit parameter)
- ▶ **Public Interface**: what you can do from outside
- ▶ **Class**: type of object of acc
- ▶ **Instance**: one of the Checksums' object

## Don't mix Class vs. Instance

- ▶ It's *Concept* vs. *Object*
- ▶ e.g.: *Car model* vs. *actual car* or to *Human being* vs. *an individual*

# Reducing the Syntactic Sugar further

- Scala allows to omit the `.` and the `()`

```
acc.add(346634554)
```

becomes

```
acc add 346634554
```

# Reducing the Syntactic Sugar further

- Scala allows to omit the `.` and the `()`

```
acc.add(346634554)
```

becomes

```
acc add 346634554
```

- This is particularly cool if you define a method called `+`

```
class Checksum {  
  private var sum = 0  
  def +(b: Int): Unit = sum += b  
  def checksum(): Int = ~(sum & 0xFF) + 1  
}  
  
val acc = new Checksum  
acc + 346634554    // nice, isn't it?
```

# Singletons: When you need only one instance

```
object checksum {  
  private var sum = 0  
  def +(b: Int): Unit = sum += b  
  def checksum(): Int = ~(sum & 0xFF) + 1  
}  
  
checksum + 346634554
```

- ▶ You can directly use checksum as an instance of the class
- ▶ You cannot create several instances

# Functional Objects: objects with no mutable state

---

- ▶ `sum` is a variable, so Checksums may change without notice
- ▶ **Side effects**  $\leadsto$  harder to reason about the object (particularly if multi-threaded)



# Functional Objects: objects with no mutable state

- ▶ `sum` is a variable, so Checksums may change without notice
- ▶ **Side effects**  $\leadsto$  harder to reason about the object (particularly if multi-threaded)

```
class Rational(n: Int, d: Int) {  
  println("Created " + n + "/" + d)  
}
```

```
scala> new Rational(1, 2)  
Created 1/2  
res0: Rational = Rational@424c0bc4
```

- ▶ **Problem:** `Rational@424c0bc4` is neither nice looking nor informative

# Functional Objects: objects with no mutable state

- ▶ sum is a variable, so Checksums may change without notice
- ▶ **Side effects**  $\leadsto$  harder to reason about the object (particularly if multi-threaded)

```
class Rational(n: Int, d: Int) {  
  println("Created "+ n + "/" + d)  
}
```

```
scala> new Rational(1, 2)  
Created 1/2  
res0: Rational = Rational@424c0bc4
```

- ▶ **Problem:** Rational@424c0bc4 is neither nice looking nor informative

## Redefining the toString() method

```
class Rational(n: Int, d: Int) {  
  override def toString() = n + "/" + d  
}
```

```
scala> new Rational(1, 2)  
res0: Rational = 1/2
```

- ▶ Notice the override keyword, because toString() is redefined

# Functional Objects: objects with no mutable state

- ▶ sum is a variable, so Checksums may change without notice
- ▶ **Side effects**  $\leadsto$  harder to reason about the object (particularly if multi-threaded)

```
class Rational(n: Int, d: Int) {  
  println("Created "+ n + "/" + d)  
}
```

```
scala> new Rational(1, 2)  
Created 1/2  
res0: Rational = Rational@424c0bc4
```

- ▶ **Problem:** Rational@424c0bc4 is neither nice looking nor informative

## Redefining the toString() method

```
class Rational(n: Int, d: Int) {  
  override def toString() = n + "/" + d  
}
```

```
scala> new Rational(1, 2)  
res0: Rational = 1/2
```

- ▶ Notice the override keyword, because toString() is redefined

## Checking preconditions

```
scala> new Rational(5, 0)  
res0: Rational = 5/0
```

- ▶ We want to forbid this

# Functional Objects: objects with no mutable state

- ▶ sum is a variable, so Checksums may change without notice
- ▶ **Side effects**  $\leadsto$  harder to reason about the object (particularly if multi-threaded)

```
class Rational(n: Int, d: Int) {  
  println("Created "+ n + "/" + d)  
}
```

```
scala> new Rational(1, 2)  
Created 1/2  
res0: Rational = Rational@424c0bc4
```

- ▶ **Problem:** Rational@424c0bc4 is neither nice looking nor informative

## Redefining the toString() method

```
class Rational(n: Int, d: Int) {  
  override def toString() = n + "/" + d  
}
```

```
scala> new Rational(1, 2)  
res0: Rational = 1/2
```

- ▶ Notice the override keyword, because toString() is redefined

## Checking preconditions

```
scala> new Rational(5, 0)  
res0: Rational = 5/0
```

```
class Rational(n: Int, d: Int) {  
  require(d != 0)  
  override def toString() = n + "/" + d  
}
```

- ▶ We want to forbid this
- ▶ new Rational(5,0) will now fail with an IllegalArgumentException

# Adding methods to the Rational class

```
class Rational(n: Int, d: Int) {  
  require(d != 0)  
  override def toString() = n+"/"+d  
  def *(that: Rational): Rational =  
    new Rational(  
      n * that.n, d * that.d)  
}
```

# Adding methods to the Rational class

```
class Rational(n: Int, d: Int) {  
  require(d != 0)  
  override def toString() = n+"/"+d  
  def *(that: Rational): Rational =  
    new Rational(  
      n * that.n, d * that.d)  
}
```

## Compilation error

error: value d is not a member of Rational

- Indeed, d is not a field of Rational (that's a constructor parameter)

# Adding methods to the Rational class

```
class Rational(n: Int, d: Int) {  
  require(d != 0)  
  override def toString() = n+"/"+d  
  def *(that: Rational): Rational =  
    new Rational(  
      n * that.n, d * that.d)  
}
```

## Compilation error

error: value d is not a member of Rational

- Indeed, d is not a field of Rational (that's a constructor parameter)

## Second try

```
class Rational(n: Int, d: Int) {  
  require(d != 0)  
  val num: Int = n  
  val den: Int = d  
  override def toString() = num+"/"+den  
  def *(that: Rational): Rational =  
    new Rational(  
      this.num * that.num,  
      this.den * that.den  
    )  
}
```

```
scala> val oneHalf = new Rational(1, 2)  
oneHalf: Rational = 1/2
```

```
scala> val twoThirds = new Rational(2,3)  
twoThirds: Rational = 2/3
```

```
scala> oneHalf * twoThirds  
res0: Rational = 2/6
```

- Much better looking.
- (`vals` don't need to be private)
- (`this`: current object; `that`: param)

# More flesh to the Rational class

## Auxiliary Constructors

```
scala> val five = new Rational(5)
five: Rational = 5/1
```

```
class Rational(n: Int, d: Int) {
  require(d != 0)
  val num: Int = n
  val den: Int = d
  // auxiliary constructor
  def this(n: Int) = this(n, 1)

  override def toString() = num+"/"+den
  def *(that: Rational): Rational =
    new Rational(this.num * that.num,
                  this.den * that.den)
}
```



# More flesh to the Rational class

## Auxiliary Constructors

```
scala> val five = new Rational(5)
five: Rational = 5/1
```

```
class Rational(n: Int, d: Int) {
  require(d != 0)
  val num: Int = n
  val den: Int = d
  // auxiliary constructor
  def this(n: Int) = this(n, 1)

  override def toString() = num+"/"+den
  def *(that: Rational): Rational =
    new Rational(this.num * that.num,
                  this.den * that.den)
}
```

## Private Fields and Methods

```
scala> new Rational(66,42)
res0: Rational = 11/7
```

```
class Rational(n: Int, d: Int) {
  require(d != 0)
  private val g = gcd(n.abs,d.abs)
  val num: Int = n / g
  val den: Int = d / g
  ...
  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
}
```

# Mixing Rationals and Integers

```
scala> val x = new Rational(2/3)
x: Rational = 2/3

scala> x * 2
res0: Rational = 4/3
```

```
class Rational(n: Int, d: Int) {
  ...
  def *(that: Rational): Rational =
    new Rational(this.num * that.num,
                  this.den * that.den)
  def *(that: Int): Rational =
    new Rational(this * that.num, this.den)
}
```

# Mixing Rationals and Integers

## First try

```
scala> val x = new Rational(2/3)
x: Rational = 2/3
```

```
scala> x * 2
res0: Rational = 4/3
```

```
class Rational(n: Int, d: Int) {
  ...
  def *(that: Rational): Rational =
    new Rational(this.num * that.num,
                  this.den * that.den)
  def *(that: Int): Rational =
    new Rational(this * that.num, this.den)
}
```

```
scala> 2 * x
error: overloaded method value * with alternatives:
  (x: Double)Double <and>
  (x: Float)Float <and>
  (x: Long)Long <and>
  (x: Int)Int <and>
  (x: Char)Int <and>
  (x: Short)Int <and>
  (x: Byte)Int
cannot be applied to (Rational)
```

Indeed, no method  
\*(x: Rational)  
in class Int

# Mixing Rationals and Integers

## First try

```
scala> val x = new Rational(2/3)
x: Rational = 2/3
```

```
scala> x * 2
res0: Rational = 4/3
```

```
class Rational(n: Int, d: Int) {
  ...
  def *(that: Rational): Rational =
    new Rational(this.num * that.num,
                  this.den * that.den)
  def *(that: Int): Rational =
    new Rational(this * that.num, this.den)
}
```

```
scala> 2 * x
error: overloaded method value * with alternatives:
  (x: Double)Double <and>
  (x: Float)Float <and>
  (x: Long)Long <and>
  (x: Int)Int <and>
  (x: Char)Int <and>
  (x: Short)Int <and>
  (x: Byte)Int
cannot be applied to (Rational)
```

Indeed, no method  
\*(x: Rational)  
in class Int

## Second try

```
implicit def intToRational(x: Int) = new Rational(x)
```

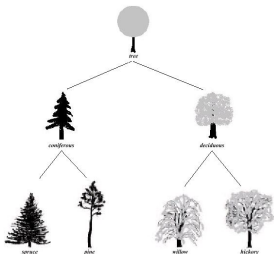
```
scala> 2 * x
res0: Rational = 4/3
```

- Removes much of the Caml sugar (!), at the price of implicit actions (F34R)

# Designing a good OOP Composition

## Abstraction

- ▶ Dealing with components and interactions without worrying about details  
Not “vague” or “imprecise”, but focused on few relevant properties
- ▶ Eliminate the irrelevant and amplify the essential
- ▶ Capture commonality between different things

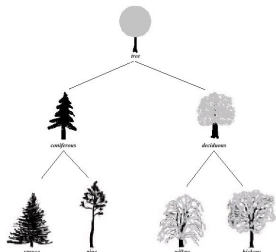


# Designing a good OOP Composition

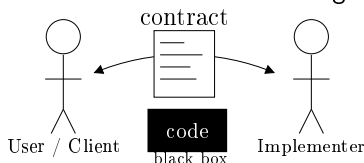
## Abstraction

- ▶ Dealing with components and interactions without worrying about details  
Not “vague” or “imprecise”, but focused on few relevant properties
- ▶ Eliminate the irrelevant and amplify the essential
- ▶ Capture commonality between different things

## Abstraction in programming



- ▶ Think about what your components should do before
- ▶ Abstract their **interface** before coding



- ▶ Show your interface, hide your implementation

# Good Property: Cohesion

A class = A concept

- ▶ Good cohesion if all parts of Public Interface are related to the concept
- ▶ Counter-example:

```
class CashRegister {  
  def enterPayment(dollars: Int, quarters: Int, dimes: Int,  
                  nickels: Int, pennies: Int): Unit = ...  
  val NICKEL_VALUE = 0.05  
  val DIME_VALUE = 0.1  
  val QUARTER_VALUE = 0.25  
}
```

- ▶ There is two concepts: `CashRegister` and `Coins`
- ▶ There must be (at least) two classes

# Coupling

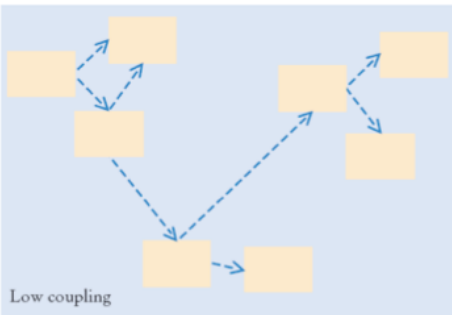
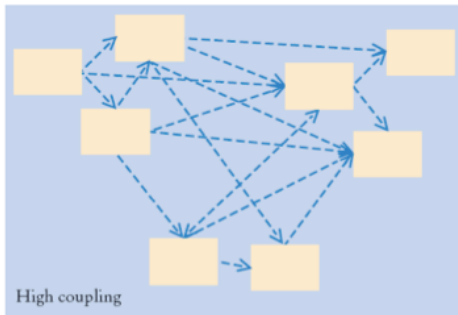
- ▶ A class *depends* on another if it utilizes it
  - ▶ CashRegister depends on Coin, Coin does not depend on CashRegister
- ▶ **Low Coupling** (few inter-class dependencies) better than **Tight Coupling**
  - ▶ Thinking of components in isolation is easier



# Coupling

- ▶ A class *depends* on another if it utilizes it
  - ▶ CashRegister depends on Coin, Coin does not depend on CashRegister
- ▶ **Low Coupling** (few inter-class dependencies) better than **Tight Coupling**
  - ▶ Thinking of components in isolation is easier

## Representing the coupling with boxes



- ▶ Cycles in coupling diagrams would have a **bad smell**
- ▶ UML is the standard way of doing it (but don't get too picky!)

# Beyond Encapsulation

- ▶ A class or object can be seen as a "component" with values and operations

<b>Vehicule</b>
maker: String
start(): Unit

- ▶ You can build other "components" for other classes

<b>Vehicule</b>
maker: String
start(): Unit

<b>Tire</b>
diameter: Float
inflate(): Unit

<b>Driver</b>
name: String
sayHello(): Unit

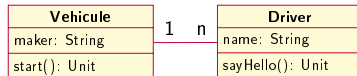
<b>Truck</b>
axleCount: Int
harness(t: Tow): Unit

- ▶ To combine them, you can either go for **association** or **inheritance**

# Class Association and Inheritance

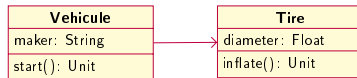
## Reciprocal Association

- ▶ A "have some" B
- ▶ B also "have some" A
- ▶ You can specify the amount of A it has (but rarely need to)



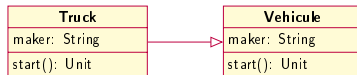
## Unidirectional Association

- ▶ A "have some" B
- ▶ but B don't have any A



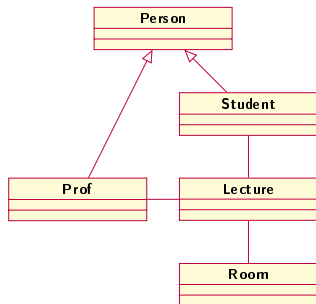
## Class inheritance

- ▶ A "is a" B
- ▶ (B cannot "be a" A, or  $A=B$ )

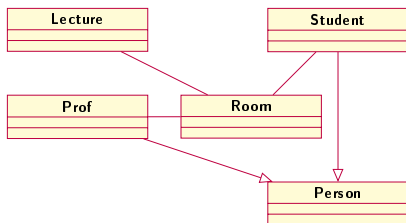


Don't worry if you forget the arrow shape: I always do too

# Quizz: Which Design is Preferable?

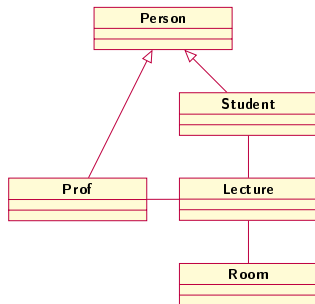


(A)

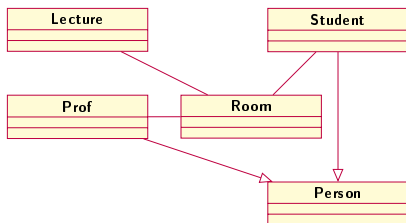


(B)

# Quizz: Which Design is Preferable?



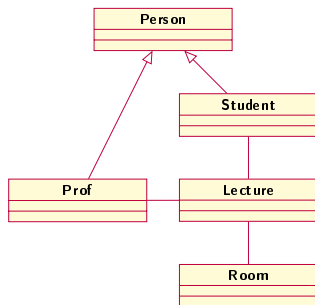
(A)



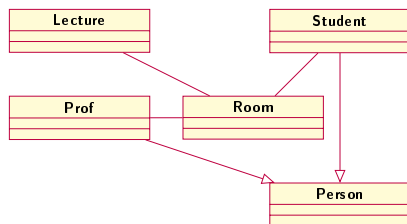
(B)

- ▶ **Theory** (n) coherent set of concepts allowing to speak of something
- ▶ These boxes quickly get boring, but notation helps thinking about large systems

# Quizz: Which Design is Preferable?



(A)



(B)

- ▶ **Theory** (n) coherent set of concepts allowing to speak of something
- ▶ These boxes quickly get boring, but notation helps thinking about large systems

## So? What is a *Good Design*?

- ▶ Much of personal taste involved, even if we framed a bit the idea
- ▶ But how would you define a *Good Proof*?

# Association and Inheritance in Scala

## Association

- ▶ A "has a" B simply means that B is a field of A

```
class A {  
  val x: Set[B]  
}
```

```
class B {  
  ...  
}
```

# Association and Inheritance in Scala

## Association

- ▶ A "has a" B simply means that B is a field of A

```
class A {  
  val x: Set[B]  
}
```

```
class B {  
  ...  
}
```

## Inheritance

- ▶ B "is a" A means that B extends A

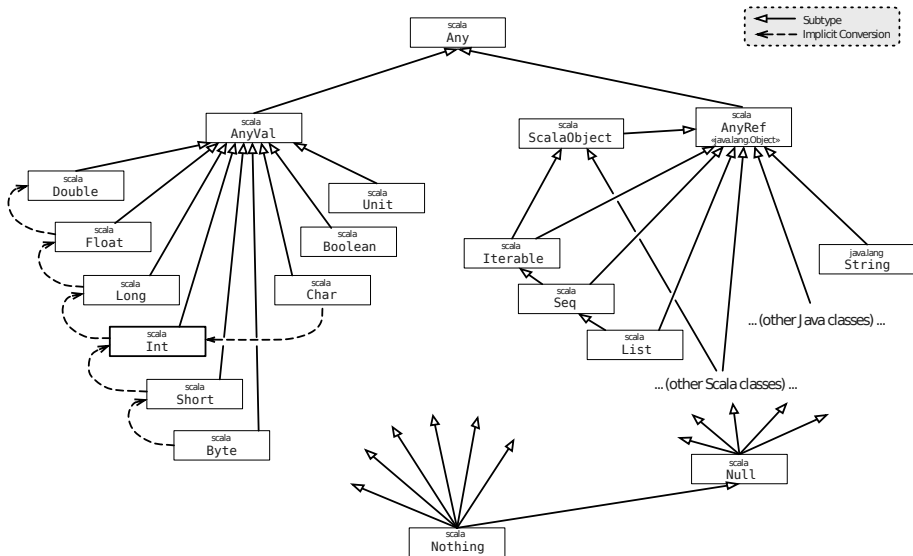
```
class A {  
  ...  
}
```

```
class B extends A {  
  ...  
}
```

- ▶ Methods and fields of A are also in B (toString() was already in Rational)
- ▶ That's a very powerful tool to **factorize code and complexity**



# Scala Class Diagram



# Polymorphism: Factorizing Complexity

## Overriding content

- If B extends A, it can **override** (redefine) definitions of A

```
class A {  
  def fun = println("I'm a A")  
}  
class B extends A {  
  override def fun = println("I'm a B")  
}
```

```
scala> val a = new A; a.fun  
I'm a A  
scala> val b = new B; b.fun  
I'm a B
```

# Polymorphism: Factorizing Complexity

## Overriding content

- ▶ If B extends A, it can **override** (redefine) definitions of A

```
class A {  
  def fun = println("I'm a A")  
}  
class B extends A {  
  override def fun = println("I'm a B")  
}
```

```
scala> val a = new A; a.fun  
I'm a A  
scala> val b = new B; b.fun  
I'm a B
```

- ▶ The code selection depends on the static and dynamic types

## Abstract class

- ▶ When a class is only there to factorize code but shouldn't be used directly

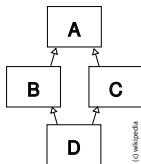
```
abstract class Ordered {  
  def <(that:Ordered):Boolean  
  def >(that:Ordered) =  
    !(that < this)  
}
```

- ▶ You cannot instantiate that class
- ▶ The class can contain *undefined def* further factorization opportunities
- ▶ Concrete sub-classes must implement them

# Multiple Inheritance

**Diamond Problem:** Multiple Inheritance is not easy

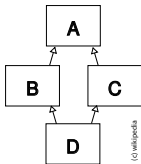
- ▶ D inherits of B and C, which both inherit of A
- ▶ Both B.fun and C.fun override the same method A.fun
- ▶ D.fun is called.
  - ▶ Which to call between B.fun & C.fun ? If both, order? A.fun called twice?



# Multiple Inheritance

**Diamond Problem:** Multiple Inheritance is not easy

- ▶ D inherits of B and C, which both inherit of A
- ▶ Both B.fun and C.fun override the same method A.fun
- ▶ D.fun is called.
  - ▶ Which to call between B.fun & C.fun ? If both, order? A.fun called twice?



**Simple Solution:** don't do that.

- ▶ Scala and Java forbid multiple class inheritance
- ▶ Simpler, but missed factorizations opportunities

**Safe multiple *is-a*:** the Java interfaces

- ▶ **Syntax:**  $\approx$  abstract classes without code any code (no diamond problem)
- ▶ Enables several implementations of the same interface
- ▶ In practice, abstract classes and interfaces are different:
  - ▶ abstract class to factorize code; interface = contract between implementer/user

# Scala Traits: Mixing Orthogonal Concerns

- ▶ Interface providing concrete members / Abstract class with multiple inheritance
- ▶ Not exactly a class: cannot take constructor parameters

```
abstract class IntQueue {  
  def get(): Int  
  def put(x: Int)  
}  
  
class BasicIntQueue extends IntQueue {  
  private val buf = new ArrayBuffer[Int]  
  def get() = buf.remove(0)  
  def put(x: Int) { buf += x }  
}
```

```
trait Doubling extends IntQueue {  
  abstract override def put(x: Int) {  
    super.put(2 * x) }  
}
```

```
scala> val queue = new BasicIntQueue  
queue: BasicIntQueue = BasicIntQueue@2465  
scala> queue.put(10)  
scala> queue.put(20)  
scala> queue.get()  
res9: Int = 10  
scala> queue.get()  
res10: Int = 20
```

# Scala Traits: Mixing Orthogonal Concerns

- ▶ Interface providing concrete members / Abstract class with multiple inheritance
- ▶ Not exactly a class: cannot take constructor parameters

```
abstract class IntQueue {  
  def get(): Int  
  def put(x: Int)  
}  
  
class BasicIntQueue extends IntQueue {  
  private val buf = new ArrayBuffer[Int]  
  def get() = buf.remove(0)  
  def put(x: Int) { buf += x }  
}
```

```
trait Doubling extends IntQueue {  
  abstract override def put(x: Int) {  
    super.put(2 * x) }  
}
```

```
trait Filtering extends IntQueue {  
  abstract override def put(x: Int) {  
    if (x >= 0) super.put(x) }  
}
```

```
scala> val queue = new BasicIntQueue  
queue: BasicIntQueue = BasicIntQueue@2465  
scala> queue.put(10)  
scala> queue.put(20)  
scala> queue.get()  
res9: Int = 10  
scala> queue.get()  
res10: Int = 20
```

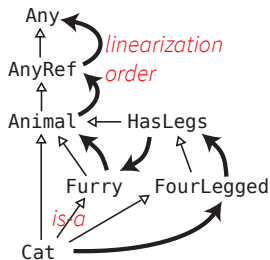
```
scala> class DQ extends  
  BasicIntQueue with Doubling  
scala> val q = new DQ; q.put(10); q.get()  
res12: Int = 20
```

```
scala> class DFQ extends BasicIntQueue\  
  with Doubling with Filtering  
scala> val q = new DFQ; q.put(-1); q.put(10)  
scala> q.get()  
res12: Int = 20
```

- ▶ Evaluation Order  $\approx$  traits further to the right take effect first

# Traits and Diamond Problem

```
class Animal
trait Furry extends Animal
trait HasLegs extends Animal
trait FourLegged extends HasLegs
class Cat extends Animal with Furry with FourLegged
```



## Linearization

- ▶ Strictly defined (but complex) order of traits and classes
- ▶ First found implementation wins – other method candidates are ignored
- ▶ super is not who you think, but it's for the best



# Dealing with Complexity

## Some classical design principles

- ▶ **Composition**: split problem in simpler sub-problems and compose pieces
- ▶ **Abstraction**: forget about details and focus on important aspects

## Object Oriented Programming

- ▶ Data are the central element
- ▶ **Encapsulation**: Divide complexity into manageable units
- ▶ **Heritage**: Factorize behavior between related units
- ▶ **Polymorphism**: Use a specialized unit instead of the expected one

## Functional Programming

- ▶ Functions and behaviors are the central elements
- ▶ Usually produces programs that are easier to reason about
- ▶ Somehow harder to write when not used to

No holy war needed: Scala has both :)

## "A cat catches a bird and eats it"



How would you design/organize/split this code?

# "A cat catches a bird and eats it"



How would you design/organize/split this code?

As a OOP programmer

- ▶ There is **two nouns: cat and bird**
- ▶ Cat has two verbs associated:  
catch and eat

```
class Bird
class Cat {
  def catch(b: Bird): Unit = ...
  def eat(): Unit = ...
}
val cat = new Cat
val bird = new Bird
cat.catch(bird)
cat.eat()
```

# "A cat catches a bird and eats it"



How would you design/organize/split this code?

## As a OOP programmer

- ▶ There is **two nouns: cat and bird**
- ▶ Cat has two verbs associated: catch and eat

```
class Bird
class Cat {
  def catch(b: Bird): Unit = ...
  def eat(): Unit = ...
}
val cat = new Cat
val bird = new Bird
cat.catch(bird)
cat.eat()
```

## As a FP programmer

- ▶ There is **two verbs: catch and eat**
- ▶ They are composed and apply to typed values

```
trait Cat
trait Bird
trait Catch
trait FullTummy
def catch(hunter: Cat, prey: Bird):
  Cat with Catch = ...
def eat(consumer: Cat with Catch):
  Cat with FullTummy = ...
val story = (catch _) andThen (eat _)
story(new Cat, new Bird)
```

Example and Code from *Scala in Depth*

# OOP vs. FP

So, do you prefer nouns or verbs? Well, both.

## Object-Oriented Programming

Composition of objects (nouns)  
Encapsulated stateful interaction  
Iterative algorithms  
Imperative flow  
Explicit Memory Layout (HW-like)

## Functional Programming

Composition of functions (verbs)  
Deferred side effects  
Recursive algorithms and continuations  
Lazy evaluation  
Pattern matching

But they seem somehow incompatible ... until Scala

- ▶ Scala is a convincing attempt to mix them
- ▶ Everything is an object (including functions)
- ▶ Best practices: prefer immutable values, even if mutables exist
- ▶ Large OOP systems were introducing FP in Java anyway (with ugly hacks)  
Now the language and compiler helps (but still that Frankenstein smell)

# Conclusion

## Computer Science is the Science of Abstraction

- ▶ But sorting the concepts require some technics
- ▶ Computer Scientists are engineers terraforming ideas and concepts :)

# Conclusion

## Computer Science is the Science of Abstraction

- ▶ But sorting the concepts require some technics
- ▶ Computer Scientists are engineers terraforming ideas and concepts :)

## Object-Oriented concepts are meant to help

- ▶ Encapsulation and abstraction to design objects; Association to compose them
- ▶ Inheritance to factorize objects; Abstract class to further factorize concepts
- ▶ **Dynamic binding**: complex problem (more to come in practical)
- ▶ Want more factorization while avoiding the Diamond Problem
  - ▶ Traits goes further than Java's interface without the C++ Diamond madness

## Functional Programming: orthogonal approach

- ▶ Focuses on verbs instead of nouns
- ▶ Which is best suited depends on the problem (and programmer)

# Conclusion

## Computer Science is the Science of Abstraction

- ▶ But sorting the concepts require some technics
- ▶ Computer Scientists are engineers terraforming ideas and concepts :)

## Object-Oriented concepts are meant to help

- ▶ Encapsulation and abstraction to design objects; Association to compose them
- ▶ Inheritance to factorize objects; Abstract class to further factorize concepts
- ▶ **Dynamic binding**: complex problem (more to come in practical)
- ▶ Want more factorization while avoiding the Diamond Problem
  - ▶ Traits goes further than Java's interface without the C++ Diamond madness

## Functional Programming: orthogonal approach

- ▶ Focuses on verbs instead of nouns
- ▶ Which is best suited depends on the problem (and programmer)

## Scala gives you both OOP and FP

- ▶ Everything is an object (even functions), and you want to use immutable objects