Classes and Traits

Nancy Hitschfeld
Matías Toro

Outline

1. Classes
2. Traits
3. Exercise

CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

Outline

1. **Classes**
2. Traits
3. Exercise

Classes

Classes are constructors for objects. Some example are:

```
class Person(var name: String, var vocation: String)
class Book(var title: String, var author: String, var year: Int)
class Movie(var name: String, var director: String, var year: Int)
```

A constructor is responsible to properly initialize an object

When you write **new** Person(**“Robert”**, **“Harmonica”**):

1 - the memory is allocated (object creation)

2 - the new object is initialized

Specified by the programmer (author of Person)

Done automatically by the virtual machine

Pay attention that having a construction does not mean your object will be well initialized.

Example of incomplete initialization

```
class ColorPoint(private var x: Double,
                 private var y: Double) {
  private var color: Color = null

  def setColor(aColor: Color): Unit = {
    color = aColor
  }
}
```

Classes

Once you have an instance of a class such as p, you can access its fields, which in this example are all constructor parameters:

```
val p = new Person("Robert Allen Zimmerman", "Harmonica Player")
p.name        // "Robert Allen Zimmerman"
p.vocation    // "Harmonica Player"
```

As the parameters were created as **`var`**, then can be mutated:

```
p.name = "Bob Dylan"
p.vocation = "Musician"
```

Default parameters

Class constructor parameters can also have default values:

```
class Socket(val timeout: Int = 5000, val linger: Int = 5000) {
  …
}
```

```
val s = new Socket() // timeout: 5000, linger: 5000
val s = new Socket(2500) // timeout: 2500, linger: 5000
val s = new Socket(10000, 10000) // timeout: 10000, linger: 10000
val s = new Socket(timeout = 10000) // timeout: 10000, linger: 5000
val s = new Socket(linger = 10000) // timeout: 5000, linger: 10000
```

Auxiliary constructors

Classes can have multiple constructors, and may invoke each other

```
// [1] the primary constructor
class Student(var name: String,  var govtId: String) {
  private var _applicationDate: Option[LocalDate] = None
  private var _studentId: Int = 0

  // [2] a constructor for when the student has completed
  // their application
  def this(name: String, govtId: String, applicationDate: LocalDate) = {
    this(name, govtId)
    _applicationDate = Some(applicationDate)
  }

  // [3] a constructor for when the student is approved
  // and now has a student id
  def this(name: String, govtId: String, studentId: Int) = {
    this(name, govtId)
    _studentId = studentId
  }
}
```

The keyword **this** is used for that purpose. Note that this “**this**”, used in to invoke constructor, has nothing to do with the “this” pseudo variable we will later see.

Auxiliary constructors

The constructors can be called like this:

```
val s1 = new Student("Mary", "123")
val s2 = new Student("Mary", "123", LocalDate.now)
val s3 = new Student("Mary", "123", 456)
```

Classes

Classes can also have methods and additional fields that are not part of constructors. They are defined in the body of the class. The body is initialized as part of the default constructor:

```
class Person(var firstName: String, var lastName: String) {

  println("initialization begins")
  val fullName = firstName + " " + lastName

  // a class method
  def printFullName: Unit = println(fullName)

  printFullName
  println("initialization ends")
}
```

What does the
new Person(**"John"**, **"Doe"**)
program prints?

Classes

Alternatively, using an auxiliary constructor:

```
class Person {
  private var firstName: String = null
  private var lastName: String = null
  private var fullName: String = null

  def this(firstName: String, lastName: String) = {
    this()
    this.firstName = firstName
    this.lastName = lastName

    println("initialization begins")
    fullName = firstName + " " + lastName

    // a class method
    printFullName()
    println("initialization ends")
  }
  def printFullName() = println(fullName)
}
```

Exercise

What does the following program prints? **abc2c3**

```
new A()
```

```
class A(val x: Int){
  print(“a")

  def this(x: String) = {
    this(x.toInt)
    print("c2")
  }

  def this() = {
    this("0")
    print("c3")
  }

  print("b")
}
```

Outline

1. Classes
2. **Traits**
3. Exercise

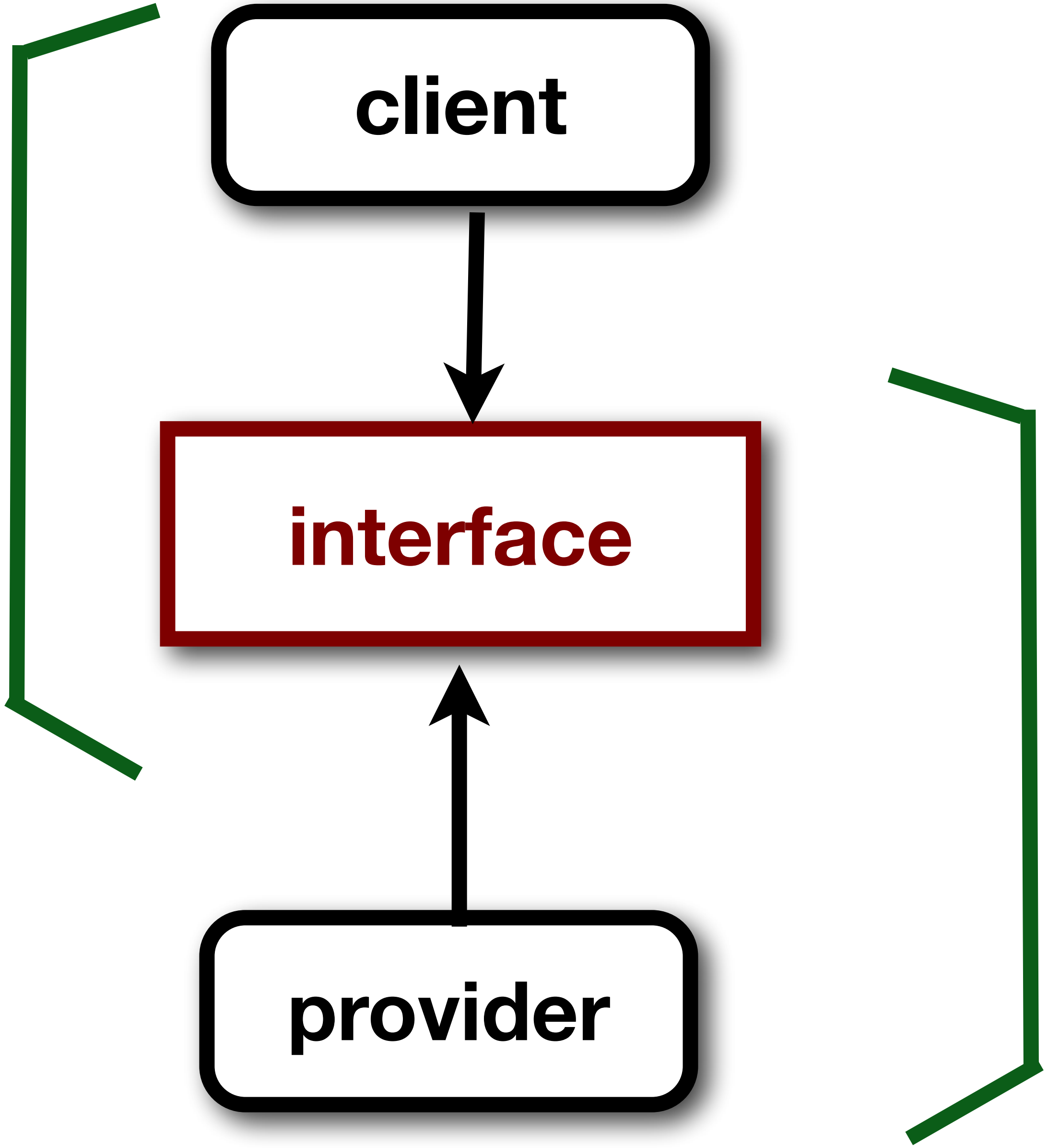
Scala Traits

“Methods form the object's interface with the outside world; the buttons on the front of your television set, for example, are the interface between you and the electrical wiring on the other side of its plastic casing. You press the "power" button to turn the television on and off.” — docs.oracle.com

In practice, Scala traits are often used to *abstract a domain variation*

Simple rule: Whenever you need more than one kind of objects, then you need to use interfaces

Interfaces as a Contract



Modular checking

Scala Traits

Traits are used to define specifications/interfaces in this course.

An interface is a group of related methods, and defines an abstract type

A class that extends or “implements” a trait must implement all abstract fields and methods it inherits. What happens if we don’t?

```
trait HasLegs {
  val numLegs: Int
  def walk(): Unit
}
class IrishSetter(name: String) extends HasLegs{
  val numLegs = 4
  def walk() = println("I’m walking")
}
```

Scala Traits

A class may implements more than one trait:

```
trait HasTail {
  def tailColor: String
}
class IrishSetter(name: String) extends HasLegs with HasTail {
  val numLegs = 4
  def tailColor = "Red"
  def walk() = println("I'm walking")
}
```

Implementing a trait allows a class to become more formal about the behavior it promises to provide.

Interfaces form a *contract between the class and the outside world*, and this contract is enforced at build time by the compiler.

Scala Traits

Traits are not instantiated, but rather implemented

In practices, it often happens that *abstract classes* (which we will see soon) implements interfaces

An interface may have 0, 1 or more super interfaces

```
trait PrintColors
trait RainbowColors
trait LotsOfColors extends
       RainbowColors with PrintColors { … }
```

Scala Traits

Refer to objects by their traits

parameters, return values, variables, and fields should ALL be declared using trait types

“if you get into the habit of using trait as types, your program will be much more flexible”

Scala Traits

One can now write:

```
val r1: IrishSetter = new IrishSetter("Bob")
val r2: HasLegs = new IrishSetter("Bob")
```

Use traits as types instead of classes => it leads to better design of programs

Which is better?:

```
def putShoes(x: IrishSetter) = { ... }
def putShoes(x: HasLegs) = { ... }
```

[Mitchell]

Subtype Polymorphism

Subtyping

Type B is a subtype of type A if any context expecting an expression of type A may take an expression of type B without errors.

Subtyping is about substitutability

Corresponds directly to the containment relation on object interfaces

Subtype polymorphism allow a single term to have many types

Implicit Subtype Polymorphism

In dynamically-typed languages,
subtype polymorphism is implicit

Python

```
def foo(o):
  o.m1()
  print(o.m2())
  return o.m3()
```

any object that understands
AT LEAST
m1,m2,m3 will do

“if it walks like a duck and quacks like a duck, then it’s a duck!”

Explicit Subtype Polymorphism

In most statically-typed languages, the subtype relation has to be explicitly declared.

If class A defines all methods of class B, A is not a subtype of B!!

```
class Cat { def talk: String = { ... } }
class Show {
  def present(c: Cat): Unit = { display(c.talk) }
}
class Robot { def talk: String = { ... } }


val tvShow = new Show
tvShow.present(new Robot)
```

Explicit Subtype Polymorphism

is a:

```
trait ICanTalkAndWalk {
  def talk: String
  def walk(): Unit
}
```

also a:

```
trait ICanTalk {
  def talk(): String
}
```

?

Remember! relation has to be explicitly declared:

```
trait ICanTalkAndWalk extends ICanTalk { ... }
```

Outline

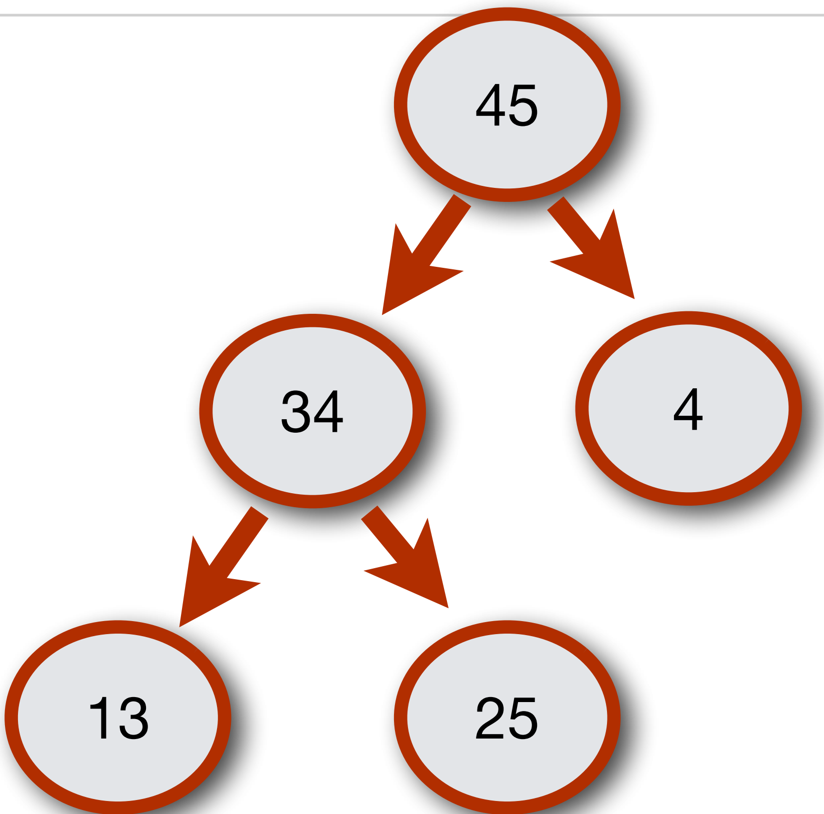
1. Classes
2. Traits
3. **Exercise**

CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

Exercise

Define binary trees with:

- sum all value
- get the min value
- get the max value



do it!

Note: try to respect the principles of the OOP paradigm

A Solution?

Java

```
class Tree(var value: Int, var left: Tree, var right: Tree) {
  def sum(): Int = {
    val right_sum = if(right == null) 0 else right.sum()
    val left_sum = if(left == null) 0 else left.sum()
    value + right_sum + left_sum
  }
  def min(): Int = {
    if (left == null && right == null)
      value
    else if (right == null)
      Math.min(value, left.min())
    else if (left == null)
      Math.min(value, right.min())
    else
      Math.min(value, Math.min(right.min(), left.min()));
  }
  ...
}
```

REMEMBER:

“any programming model that allows inspection of the representation of more than one abstraction at a time is NOT object oriented”.

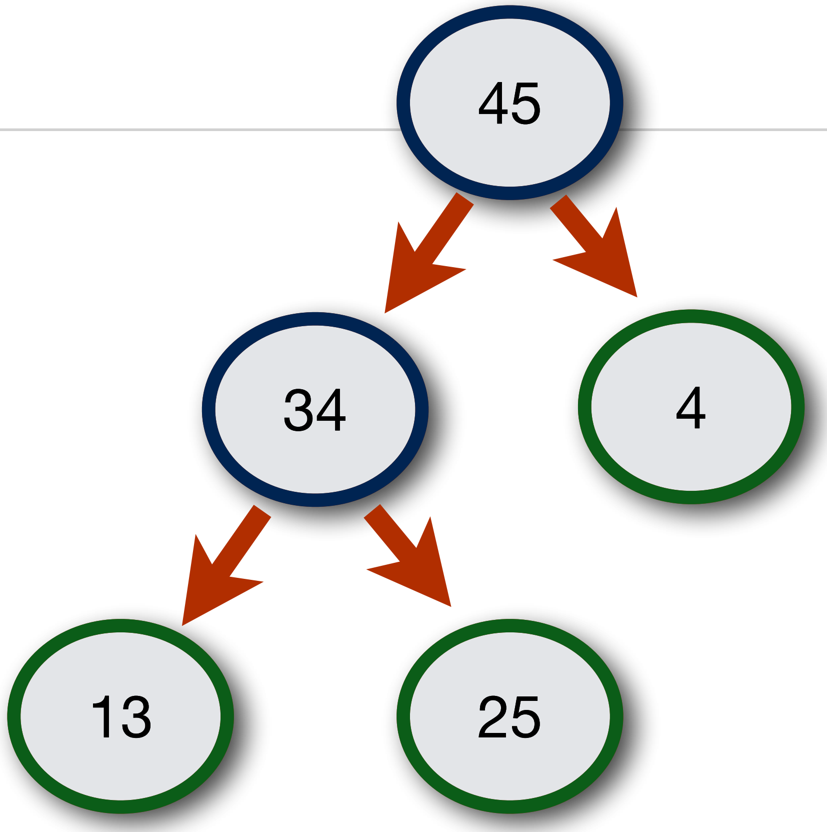
[Cook]

Looking if an object is == null is looking at its representation…

Insight

Define binary trees with:

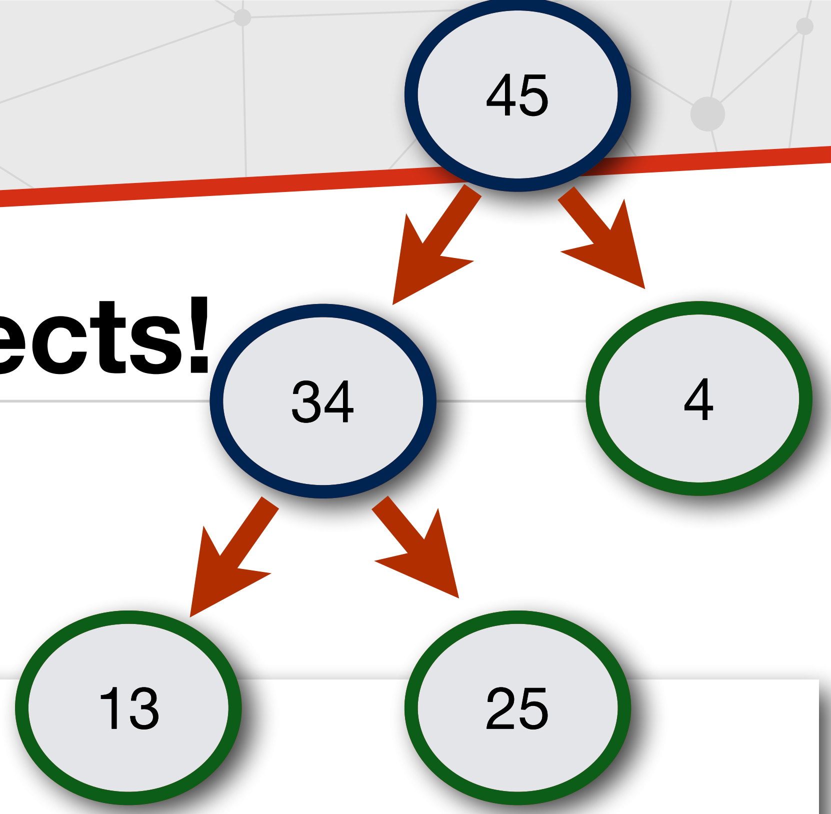
- sum all value
- get the min value
- get the max value



Different Kinds of Objects!

Note: try to respect the principles of the OOP paradigm

Two different kind of objects!



Same interface: sum/min/max

Different implementation

```
trait Tree{
  def sum(): Int
  def min(): Int
}
class Leaf(value: Int) extends Tree{
  def sum(): Int = value
  def min(): Int = value
}
class Node(value: Int, left: Tree, right: Tree) extends Tree{
  def sum(): Int = this.value + right.sum() + left.sum()
  def min(): Int = Math.min(this.value, Math.min(right.min(), left.min()))
}
val n: Tree = new Node(45,
  new Node(34, new Leaf(13), new Leaf(25)),
  new Leaf(4)
)
```

License

Algunas diapos corresponden a Éric Tanter dcc

CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

www.dcc.uchile.cl

/ DCCUCHILE