

# Fundamentos de Programación Funcional y Concurrente

## Funciones y Datos

Juan Francisco Díaz Frias

Profesor Titular (1993-hoy)  
juanfco.diaz@correounivalle.edu.co  
Edif. B13 - 4009



**Universidad del Valle**

Octubre 2022

# Plan

- 1 Generalidades
- 2 Construyendo abstracciones de datos
  - Clases y Objetos
  - Control de acceso y de encapsulación
  - Evaluación y operadores
- 3 Construyendo abstracciones de datos incrementalmente
  - Jerarquía de clases
  - Despacho dinámico (control de acceso)
- 4 Organización de clases en Scala

# Plan

- 1 Generalidades
- 2 Construyendo abstracciones de datos
  - Clases y Objetos
  - Control de acceso y de encapsulación
  - Evaluación y operadores
- 3 Construyendo abstracciones de datos incrementalmente
  - Jerarquía de clases
  - Despacho dinámico (control de acceso)
- 4 Organización de clases en Scala

# Plan

- 1 Generalidades
- 2 Construyendo abstracciones de datos
  - Clases y Objetos
  - Control de acceso y de encapsulación
  - Evaluación y operadores
- 3 Construyendo abstracciones de datos incrementalmente
  - Jerarquía de clases
  - Despacho dinámico (control de acceso)
- 4 Organización de clases en Scala

# Plan

- 1 Generalidades
- 2 Construyendo abstracciones de datos
  - Clases y Objetos
  - Control de acceso y de encapsulación
  - Evaluación y operadores
- 3 Construyendo abstracciones de datos incrementalmente
  - Jerarquía de clases
  - Despacho dinámico (control de acceso)
- 4 Organización de clases en Scala

# Generalidades

- Hasta ahora nos hemos enfocado en construir **abstracciones funcionales**, combinando y componiendo funciones a partir de otras funciones. EL foco ha estado en la **abstracción de procesos**.
- Otro aspecto clave para la programación es la capacidad de **construir abstracciones de datos** a partir de otras abstracciones de datos más sencillas. Esto permite:
  - Elevar el nivel conceptual
  - Incrementar la modularidad
  - Fortalecer el poder expresivo del lenguaje
- La capacidad de construir abstracciones de datos compuestas nos permite **trabajar con los datos a un nivel conceptual más alto** que si trabajamos solamente con las abstracciones de datos primitivas del lenguaje.

# Generalidades

- Hasta ahora nos hemos enfocado en construir **abstracciones funcionales**, combinando y componiendo funciones a partir de otras funciones. EL foco ha estado en la **abstracción de procesos**.
- Otro aspecto clave para la programación es la capacidad de **construir abstracciones de datos** a partir de otras abstracciones de datos más sencillas. Esto permite:
  - Elevar el nivel conceptual
  - Incrementar la modularidad
  - Fortalecer el poder expresivo del lenguaje
- La capacidad de construir abstracciones de datos compuestas nos permite **trabajar con los datos a un nivel conceptual más alto** que si trabajamos solamente con las abstracciones de datos primitivas del lenguaje.

# Generalidades

- Hasta ahora nos hemos enfocado en construir **abstracciones funcionales**, combinando y componiendo funciones a partir de otras funciones. EL foco ha estado en la **abstracción de procesos**.
- Otro aspecto clave para la programación es la capacidad de **construir abstracciones de datos** a partir de otras abstracciones de datos más sencillas. Esto permite:
  - Elevar el nivel conceptual
  - Incrementar la modularidad
  - Fortalecer el poder expresivo del lenguaje
- La capacidad de construir abstracciones de datos compuestas nos permite **trabajar con los datos a un nivel conceptual más alto** que si trabajamos solamente con las abstracciones de datos primitivas del lenguaje.



# Ejemplo motivacional

Suponga que tenemos la tarea de diseñar un sistema para hacer aritmética con los números racionales

- **Elevar nivel conceptual:** uno podría pensar en modelar un número racional como dos enteros (el numerador y el denominador). Pero será mucho mejor si creamos una nueva abstracción denominada, por ejemplo, *Racional*, que nuestros programas puedan manipular sin importar cómo está implementada "por debajo".
- **Incrementar la modularidad:** si podemos manipular los números racionales directamente como tales, sin interesarnos en cómo están siendo ellos representados internamente, estaremos usando una poderosa herramienta de diseño denominada *abstracción de datos*. Esta técnica hace que los programas sean más fáciles de diseñar, mantener y modificar.
- **Fortalecer el poder expresivo del lenguaje:** Poder utilizar la nueva abstracción de datos como cualquier otra abstracción ya existente amplía el poder expresivo del lenguaje.

# Ejemplo motivacional

Suponga que tenemos la tarea de diseñar un sistema para hacer aritmética con los números racionales

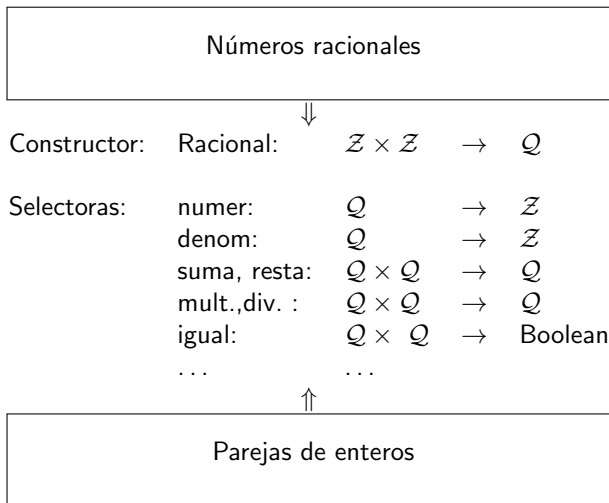
- **Elevar nivel conceptual:** uno podría pensar en modelar un número racional como dos enteros (el numerador y el denominador). Pero será mucho mejor si creamos una nueva abstracción denominada, por ejemplo, *Racional*, que nuestros programas puedan manipular sin importar cómo está implementada "por debajo".
- **Incrementar la modularidad:** si podemos manipular los números racionales directamente como tales, sin interesarnos en cómo están siendo ellos representados internamente, estaremos usando una poderosa herramienta de diseño denominada *abstracción de datos*. Esta técnica hace que los programas sean más fáciles de diseñar, mantener y modificar.
- **Fortalecer el poder expresivo del lenguaje:** Poder utilizar la nueva abstracción de datos como cualquier otra abstracción ya existente amplía el poder expresivo del lenguaje.

# Ejemplo motivacional

Suponga que tenemos la tarea de diseñar un sistema para hacer aritmética con los números racionales

- **Elevar nivel conceptual:** uno podría pensar en modelar un número racional como dos enteros (el numerador y el denominador). Pero será mucho mejor si creamos una nueva abstracción denominada, por ejemplo, *Racional*, que nuestros programas puedan manipular sin importar cómo está implementada "por debajo".
- **Incrementar la modularidad:** si podemos manipular los números racionales directamente como tales, sin interesarnos en cómo están siendo ellos representados internamente, estaremos usando una poderosa herramienta de diseño denominada *abstracción de datos*. Esta técnica hace que los programas sean más fáciles de diseñar, mantener y modificar.
- **Fortalecer el poder expresivo del lenguaje:** Poder utilizar la nueva abstracción de datos como cualquier otra abstracción ya existente amplía el poder expresivo del lenguaje.

# Sistema para hacer aritmética con los racionales



# Números racionales

- Diseñaremos un paquete para hacer aritmética con números racionales
- Un número racional  $\frac{x}{y}$  se representa por medio de dos números enteros:
  - El numerador,  $x$
  - El denominador,  $y$
- Representar un número racional, en el lenguaje de programación escogido, por medio de dos enteros, haría muy engorrosa la gestión de esos números.
- Una mejor alternativa consiste en **combinar numerador y denominador** en una misma estructura de datos. En este caso una **clase de Scala**.

# Números racionales

- Diseñaremos un paquete para hacer aritmética con números racionales
- Un número racional  $\frac{x}{y}$  se representa por medio de dos números enteros:
  - El numerador,  $x$
  - El denominador,  $y$
- Representar un número racional, en el lenguaje de programación escogido, por medio de dos enteros, haría muy engorrosa la gestión de esos números.
- Una mejor alternativa consiste en combinar numerador y denominador en una misma estructura de datos. En este caso una clase de Scala.

# Números racionales

- Diseñaremos un paquete para hacer aritmética con números racionales
- Un número racional  $\frac{x}{y}$  se representa por medio de dos números enteros:
  - El numerador,  $x$
  - El denominador,  $y$
- Representar un número racional, en el lenguaje de programación escogido, por medio de dos enteros, haría muy engorrosa la gestión de esos números.
- Una mejor alternativa consiste en combinar numerador y denominador en una misma estructura de datos. En este caso una clase de Scala.

# Números racionales

- Diseñaremos un paquete para hacer aritmética con números racionales
- Un número racional  $\frac{x}{y}$  se representa por medio de dos números enteros:
  - El numerador,  $x$
  - El denominador,  $y$
- Representar un número racional, en el lenguaje de programación escogido, por medio de dos enteros, haría muy engorrosa la gestión de esos números.
- Una mejor alternativa consiste en **combinar numerador y denominador** en una misma estructura de datos. En este caso una **clase** de Scala.



# Plan

- 1 Generalidades
- 2 Construyendo abstracciones de datos
  - Clases y Objetos
  - Control de acceso y de encapsulación
  - Evaluación y operadores
- 3 Construyendo abstracciones de datos incrementalmente
  - Jerarquía de clases
  - Despacho dinámico (control de acceso)
- 4 Organización de clases en Scala

# Clases

En Scala, construimos esa combinación, definiendo una **clase**:

```
0 class Racional(x:Int, y:Int) {  
1   def numer = x  
2   def denom = y  
3 }
```

Esta definición introduce dos entidades:

- Un nuevo **tipo** denominado *Racional*
- Un **constructor** también de nombre *Racional*, para crear elementos de este tipo.

Scala almacena las definiciones de tipos y valores en dos **espacios de nombres diferentes**. Por ello no hay conflicto entre esas dos definiciones con el mismo nombre.

# Objetos

- A los elementos de un tipo asociado a una clase se les denomina **objetos**

Para crear un nuevo objeto de una clase, se utiliza el operador **new**:

```
0 scala> new Racional(1,2)
1 val res0: Racional = Racional@2faf6e4a
```

- Los objetos de la clase *Racional* tienen 2 **miembros**: *numer* y *denom*. Los miembros de un objeto se seleccionan con el operador **'.'** (como en Java):

```
0 scala> val x=new Racional(1,2)
1 val x: Racional = Racional@990b86b
2
3 scala> x.numer
4 val res1: Int = 1
5
6 scala> x.denom
7 val res2: Int = 2
```

# Aritmética racional

La aritmética que debemos implementar es la siguiente:

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2}$$

$$\frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{n_1 d_2 - n_2 d_1}{d_1 d_2}$$

$$\frac{n_1}{d_1} \times \frac{n_2}{d_2} = \frac{n_1 n_2}{d_1 d_2}$$

$$\frac{n_1}{d_1} / \frac{n_2}{d_2} = \frac{n_1 d_2}{d_1 n_2}$$

$$\frac{n_1}{d_1} = \frac{n_2}{d_2} \equiv n_1 d_2 = d_1 n_2$$

# Implementando la aritmética racional

Las funciones para sumar dos racionales y para convertir un racional en cadena para poder visualizarlo son:

```
0 def sumaRacional(r: Racional, s: Racional): Racional =  
1   new Racional(  
2     r.numer * s.denom + r.denom * s.numer,  
3     r.denom * s.denom  
4   )  
5  
6 def convertirEnCadena(r: Racional) =  
7   s"$r.numer / r.denom"
```

Y el resultado al usarlo:

```
0 scala> val y= new Racional(2,3)  
1 val y: Racional = 2/3  
2  
3 scala> val x=new Racional(1,2)  
4 val x: Racional = 1/2  
5  
6 scala> sumaRacional(x,y)  
7 val res9: Racional = 7/6
```

# Métodos

- Se puede ir más lejos en la abstracción de los datos, y empaquetarlos en la misma abstracción junto con las funciones que los manipulan. Esas funciones se denominan **métodos**.
- Nuestros números racionales deberían tener, además de las funciones *numer* y *denom*, las funciones *suma*, *resta*, *mult*, *div*, *igual*.

# Métodos

- Se puede ir más lejos en la abstracción de los datos, y empaquetarlos en la misma abstracción junto con las funciones que los manipulan. Esas funciones se denominan **métodos**.
- Nuestros números racionales deberían tener, además de las funciones *numer* y *denom*, las funciones *suma*, *resta*, *mult*, *div*, *igual*.

# Métodos para *Racional*

El esquema de la implementación sería:

```
0  class Racional(x:Int, y:Int) {  
1    def number = x  
2    def denom = y  
3    def suma(r:Racional) =  
4      new Racional(  
5        number * r.denom + denom * r.number,  
6        denom * r.denom  
7      )  
8    def mult(r:Racional) = ...  
9    ...  
10   override def toString = s"number/ denom"  
11 }
```

Nótese que:

- Las funciones (métodos) como *suma* no tienen sino un argumento: el otro es la abstracción misma
- El modificador *override* declara que va a redefinir un método que ya existe: *toString*



# Invocación de métodos

- La abstracción con *suma* y *mul* terminadas se ve así:

```
0  class Racional(x:Int, y:Int) {  
1    def numer = x  
2    def denom = y  
3    def suma(r: Racional) =  
4      new Racional(  
5        numer * r.denom + denom * r.numer,  
6        denom * r.denom  
7      )  
8    def mult(r: Racional) =  
9      new Racional(  
10       numer * r.numer,  
11       denom * r.denom  
12     )  
13    override def toString = s"numer/denom"  
14  }
```

- Ahora podemos invocar las funciones (métodos) definidos:

```
0  scala> val x=new Racional(1,3)  
1      |   val y=new Racional(5,7)  
2      |   val z=new Racional(3,2)  
3      |   x.suma(y).mult(z)  
4  val x: Racional = 1/3  
5  val y: Racional = 5/7  
6  val z: Racional = 3/2  
7  val res10: Racional = 66/42
```

# Plan

- 1 Generalidades
- 2 **Construyendo abstracciones de datos**
  - Clases y Objetos
  - **Control de acceso y de encapsulación**
  - Evaluación y operadores
- 3 Construyendo abstracciones de datos incrementalmente
  - Jerarquía de clases
  - Despacho dinámico (control de acceso)
- 4 Organización de clases en Scala

# Mejorando la abstracción

- En el ejemplo anterior podemos ver que los números racionales no están siendo representados siempre en su forma más simple (¿Por qué?)
- Uno esperaría que los números racionales estuvieran *simplificados*: En lugar de 66/42 tener 11/7. ¿Cómo hacerlo?
- Hallamos el  $\text{mcd}(66, 42) = 6$  y dividimos tanto el numerador como el denominador por ese número.
- ¿Dónde incorporamos eso en la abstracción? ¿En cada operación?
- Una mejor alternativa, sería simplificar la abstracción en la clase, cada que el objeto se construye.

# Mejorando la abstracción

- En el ejemplo anterior podemos ver que los números racionales no están siendo representados siempre en su forma más simple (¿Por qué?)
- Uno esperaría que los números racionales estuvieran *simplificados*: En lugar de 66/42 tener 11/7. **¿Cómo hacerlo?**
- Hallamos el  $\text{mcd}(66, 42) = 6$  y dividimos tanto el numerador como el denominador por ese número.
- ¿Dónde incorporamos eso en la abstracción? ¿En cada operación?
- Una mejor alternativa, sería simplificar la abstracción en la clase, cada que el objeto se construye.

# Mejorando la abstracción

- En el ejemplo anterior podemos ver que los números racionales no están siendo representados siempre en su forma más simple (¿Por qué?)
- Uno esperaría que los números racionales estuvieran *simplificados*: En lugar de 66/42 tener 11/7. ¿Cómo hacerlo?
- Hallamos el  $\text{mcd}(66, 42) = 6$  y dividimos tanto el numerador como el denominador por ese número.
- ¿Dónde incorporamos eso en la abstracción? ¿En cada operación?
- Una mejor alternativa, sería simplificar la abstracción en la clase, cada que el objeto se construye.

# Mejorando la abstracción

- En el ejemplo anterior podemos ver que los números racionales no están siendo representados siempre en su forma más simple (¿Por qué?)
- Uno esperaría que los números racionales estuvieran *simplificados*: En lugar de 66/42 tener 11/7. ¿Cómo hacerlo?
- Hallamos el  $\text{mcd}(66, 42) = 6$  y dividimos tanto el numerador como el denominador por ese número.
- ¿Dónde incorporamos eso en la abstracción? ¿En cada operación?
- Una mejor alternativa, sería simplificar la abstracción en la clase, cada que el objeto se construye.

# Mejorando la abstracción

- En el ejemplo anterior podemos ver que los números racionales no están siendo representados siempre en su forma más simple (¿Por qué?)
- Uno esperaría que los números racionales estuvieran *simplificados*: En lugar de  $66/42$  tener  $11/7$ . ¿Cómo hacerlo?
- Hallamos el  $mcd(66, 42) = 6$  y dividimos tanto el numerador como el denominador por ese número.
- ¿Dónde incorporamos eso en la abstracción? ¿En cada operación?
- Una mejor alternativa, sería simplificar la abstracción en la clase, cada que el objeto se construye.

# Racionales con simplificación

- Simplificamos numerador y denominador calculando el *mcd*:

```
0  class Racional(x:Int, y:Int) {  
1    private def mcd(a:Int, b:Int):Int =  
2      if (b==0) a else mcd(b, a % b)  
3    private val m=mcd(x,y)  
4    def numer = x/m  
5    def denom = y/m  
6    ...  
7  }
```

- Nótese que:
  - Se definen dos nuevos miembros de la clase: *mcd* y *m*, y se etiquetan como **privados**. Esto limita el **acceso** a estos miembros
  - *m* fue definido por valor y no por nombre. Esto significa que se calcula una vez, y se usa de allí en adelante.



# Racionales con simplificación - variantes

- Se podría invocar *mcd* desde *numer* y *denom* sin necesidad de calcular *m*:

```
0 class Racional(x:Int, y:Int) {  
1   private def mcd(a:Int, b:Int):Int =  
2     if (b==0) a else mcd(b, a % b)  
3   def numer = x/mcd(x,y)  
4   def denom = y/mcd(x,y)  
5   ...  
6 }
```

Útil si *numer* y *denom* son invocadas con poca frecuencia

- También podríamos hacer que *numer* y *denom* sean valores y no funciones:

```
0 class Racional(x:Int, y:Int) {  
1   private def mcd(a:Int, b:Int):Int =  
2     if (b==0) a else mcd(b, a % b)  
3   val numer = x/mcd(x,y)  
4   val denom = y/mcd(x,y)  
5   ...  
6 }
```

Útil si *numer* y *denom* son invocadas con mucha frecuencia

- La **abstracción de datos** (separación de implementación y comportamiento) es una piedra angular de la ingeniería de software

# Ligadura dinámica

- Los **métodos son funciones que tienen un parámetro implícito**: el objeto dueño de los métodos que estamos invocando.
- Los lenguajes OO ofrecen siempre una sintaxis para referirse a ese objeto: *this* y *self* son las más usadas.
- A esta referencia se le conoce como **ligadura dinámica** porque los lenguajes de programación deciden a qué se refiere *this* durante la ejecución.

# Ligadura dinámica

- Los **métodos son funciones que tienen un parámetro implícito**: el objeto dueño de los métodos que estamos invocando.
- Los lenguajes OO ofrecen siempre una sintaxis para referirse a ese objeto: *this* y *self* son las más usadas.
- A esta referencia se le conoce como **ligadura dinámica** porque los lenguajes de programación deciden a qué se refiere *this* durante la ejecución.

# Ligadura dinámica

- Los **métodos son funciones que tienen un parámetro implícito**: el objeto dueño de los métodos que estamos invocando.
- Los lenguajes OO ofrecen siempre una sintaxis para referirse a ese objeto: **this** y **self** son las más usadas.
- A esta referencia se le conoce como **ligadura dinámica** porque los lenguajes de programación deciden a qué se refiere **this** durante la ejecución.

# Extendiendo los racionales

- Nos piden añadir las funciones *menorQue* y *max* a la clase *Racional*:

```
0  class Racional(x:Int, y:Int) {  
1    ...  
2    def menorQue(r:Racional)=  
3      numer * r.denom < denom * r.numer  
4    def max(r:Racional)=  
5      if (this.menorQue(r)) r else this  
6    ...  
7  }
```

Al interior de *Racional*, *this* hace referencia al objeto cuyo método está siendo ejecutado.

- Cualquier referencia dentro de la clase a un nombre de un miembro de la clase, por ejemplo *numer*, es realmente una referencia a *this.numer*.

El código de *menorQue* es una abreviación de:

```
0  class Racional(x:Int, y:Int) {  
1    ...  
2    def menorQue(r:Racional)=  
3      this.numer * r.denom < this.denom * r.numer  
4    ...  
5  }
```

# Precondiciones y aserciones

- En la clase *Racional* podemos requerir que el denominador sea siempre positivo. Podemos imponer esa condición con la **función predefinida** *require*:

```
0 class Racional(x:Int, y:Int) {  
1   require(y>0, "El denominador debe ser positivo")  
2   ...  
3 }
```

Si la condición impuesta en el *require* es evaluada a *false*, se lanza una excepción *IllegalArgumentException* con el mensaje que diga la cadena

- Otra función predefinida es ***assert***:

```
0 val x=sqrt(y)  
1 assert(x>=0)
```

Si la condición impuesta en el *assert* es evaluada a *false*, se lanza una excepción *AssertionError*

- Esto refleja una diferencia en la intención:
  - require* impone una **precondición** a quien invoca una función
  - assert* verificar una condición que uno quiere que se cumpla en ese punto de la ejecución

# Plan

- 1 Generalidades
- 2 **Construyendo abstracciones de datos**
  - Clases y Objetos
  - Control de acceso y de encapsulación
  - **Evaluación y operadores**
- 3 Construyendo abstracciones de datos incrementalmente
  - Jerarquía de clases
  - Despacho dinámico (control de acceso)
- 4 Organización de clases en Scala

# Extendiendo el modelo de substitución con clases

- Ya sabemos qué significa evaluar una invocación de una función, usando el modelo de substitución.
- Ahora necesitamos extender el modelo con la aparición de las clases. ¿Qué significa aplicar un método? Se extiende el modelo de substitución a clases y objetos.
- Suponga que tenemos una definición de clase

$$\text{class } C(x_1, \dots, x_m) \{ \dots \text{def } f(y_1, \dots, y_n) = b \dots \}$$

donde:

- Los parámetros formales de la clase son  $x_1, \dots, x_m$
- La clase contiene un método  $f$  cuyos parámetros formales son  $y_1, \dots, y_n$
- Por simplicidad se omiten los tipos de los parámetros



# Extendiendo el modelo de substitución con clases

- Ya sabemos qué significa evaluar una invocación de una función, usando el modelo de substitución.
- Ahora necesitamos extender el modelo con la aparición de las clases. ¿Qué significa aplicar un método? Se extiende el modelo de substitución a clases y objetos.
- Suponga que tenemos una definición de clase

$$\text{class } C(x_1, \dots, x_m) \{ \dots \text{def } f(y_1, \dots, y_n) = b \dots \}$$

donde:

- Los parámetros formales de la clase son  $x_1, \dots, x_m$
- La clase contiene un método  $f$  cuyos parámetros formales son  $y_1, \dots, y_n$
- Por simplicidad se omiten los tipos de los parámetros

# Extendiendo el modelo de substitución con clases

- Ya sabemos qué significa evaluar una invocación de una función, usando el modelo de substitución.
- Ahora necesitamos extender el modelo con la aparición de las clases. ¿Qué significa aplicar un método? Se extiende el modelo de substitución a clases y objetos.
- Suponga que tenemos una definición de clase

$$\text{class } C(x_1, \dots, x_m) \{ \dots \text{def } f(y_1, \dots, y_n) = b \dots \}$$

donde:

- Los parámetros formales de la clase son  $x_1, \dots, x_m$
- La clase contiene un método  $f$  cuyos parámetros formales son  $y_1, \dots, y_n$
- Por simplicidad se omiten los tipos de los parámetros

# Creación de objetos

Suponga que tenemos una definición de clase

$$\text{class } C(x_1, \dots, x_m) \{ \dots \text{def } f(y_1, \dots, y_n) = b \dots \}$$

¿Cuál es el resultado de evaluar  $\text{new } C(e_1, \dots, e_m)$  en el modelo de sustitución?

- Evalúe los argumentos  $e_1, \dots, e_m$ . Sean  $v_1, \dots, v_m$  los resultados de esas evaluaciones.
- El resultado de evaluar  $\text{new } C(e_1, \dots, e_m)$  es

$$\text{new } C(v_1, \dots, v_m)$$

# Creación de objetos

Suponga que tenemos una definición de clase

$$\text{class } C(x_1, \dots, x_m) \{ \dots \text{def } f(y_1, \dots, y_n) = b \dots \}$$

¿Cuál es el resultado de evaluar  $\text{new } C(e_1, \dots, e_m)$  en el modelo de sustitución?

- Evalúe los argumentos  $e_1, \dots, e_m$ . Sean  $v_1, \dots, v_m$  los resultados de esas evaluaciones.
- El resultado de evaluar  $\text{new } C(e_1, \dots, e_m)$  es

$\text{new } C(v_1, \dots, v_m)$

# Aplicación de métodos

Suponga que tenemos una definición de clase

$$\text{class } C(x_1, \dots, x_m) \{ \dots \text{def } f(y_1, \dots, y_n) = b \dots \}$$

y el objeto

$$\text{new } C(v_1, \dots, v_m)$$

¿Cuál es el resultado de evaluar  $\text{new } C(v_1, \dots, v_m).f(w_1, \dots, w_n)$  en el modelo de substitución?

El resultado consiste en **substituir en  $b$** :

- Los parámetros formales de  $f$ ,  $y_1, \dots, y_n$ , por los argumentos actuales  $w_1, \dots, w_n$
- Los parámetros formales de  $C$ ,  $x_1, \dots, x_m$ , por los argumentos actuales del objeto  $v_1, \dots, v_m$
- Las referencias a *this* por el valor del objeto:  $\text{new } C(v_1, \dots, v_m)$
- Estas tres substituciones se denotarán así:

$$[w_1/y_1, \dots, w_n/y_n][v_1/x_1, \dots, v_m/x_m][\text{new } C(v_1, \dots, v_m)/\text{this}] b$$

# Aplicación de métodos

Suponga que tenemos una definición de clase

$$\text{class } C(x_1, \dots, x_m) \{ \dots \text{def } f(y_1, \dots, y_n) = b \dots \}$$

y el objeto

$$\text{new } C(v_1, \dots, v_m)$$

¿Cuál es el resultado de evaluar  $\text{new } C(v_1, \dots, v_m).f(w_1, \dots, w_n)$  en el modelo de sustitución?

El resultado consiste en **substituir en  $b$** :

- Los parámetros formales de  $f$ ,  $y_1, \dots, y_n$ , por los argumentos actuales  $w_1, \dots, w_n$
- Los parámetros formales de  $C$ ,  $x_1, \dots, x_m$ , por los argumentos actuales del objeto  $v_1, \dots, v_m$
- Las referencias a *this* por el valor del objeto:  $\text{new } C(v_1, \dots, v_m)$
- Estas tres substituciones se denotarán así:

$$[w_1/y_1, \dots, w_n/y_n][v_1/x_1, \dots, v_m/x_m][\text{new } C(v_1, \dots, v_m)/\text{this}] b$$

# Aplicación de métodos

Suponga que tenemos una definición de clase

$$\text{class } C(x_1, \dots, x_m) \{ \dots \text{def } f(y_1, \dots, y_n) = b \dots \}$$

y el objeto

$$\text{new } C(v_1, \dots, v_m)$$

¿Cuál es el resultado de evaluar  $\text{new } C(v_1, \dots, v_m).f(w_1, \dots, w_n)$  en el modelo de sustitución?

El resultado consiste en **substituir en  $b$** :

- Los parámetros formales de  $f$ ,  $y_1, \dots, y_n$ , por los argumentos actuales  $w_1, \dots, w_n$
- Los parámetros formales de  $C$ ,  $x_1, \dots, x_m$ , por los argumentos actuales del objeto  $v_1, \dots, v_m$
- Las referencias a *this* por el valor del objeto:  $\text{new } C(v_1, \dots, v_m)$
- Estas tres substituciones se denotarán así:

$$[w_1/y_1, \dots, w_n/y_n][v_1/x_1, \dots, v_m/x_m][\text{new } C(v_1, \dots, v_m)/\text{this}] b$$

# Aplicación de métodos

Suponga que tenemos una definición de clase

$$\text{class } C(x_1, \dots, x_m) \{ \dots \text{def } f(y_1, \dots, y_n) = b \dots \}$$

y el objeto

$$\text{new } C(v_1, \dots, v_m)$$

¿Cuál es el resultado de evaluar  $\text{new } C(v_1, \dots, v_m).f(w_1, \dots, w_n)$  en el modelo de substitución?

El resultado consiste en **substituir en  $b$** :

- Los parámetros formales de  $f$ ,  $y_1, \dots, y_n$ , por los argumentos actuales  $w_1, \dots, w_n$
- Los parámetros formales de  $C$ ,  $x_1, \dots, x_m$ , por los argumentos actuales del objeto  $v_1, \dots, v_m$
- Las referencias a *this* por el valor del objeto:  $\text{new } C(v_1, \dots, v_m)$
- Estas tres substituciones se denotarán así:

$$[w_1/y_1, \dots, w_n/y_n][v_1/x_1, \dots, v_m/x_m][\text{new } C(v_1, \dots, v_m)/\text{this}] b$$



# Ejemplos de evaluaciones de aplicaciones de métodos

- *newRacional(1,2).numer*

→  $\llbracket [1/x, 2/y][\text{newRacional}(1,2)/\text{this}]x \rrbracket$

= 1

- *newRacional(1,2).menorQue(newRacional(2,3))*

→  $\llbracket [\text{newRacional}(2,3)/r][1/x, 2/y][\text{newRacional}(1,2)/\text{this}]\text{this.numer} * r.denom < \text{this.denom} * r.numer \rrbracket$

= *newRacional(1,2).numer \* newRacional(2,3).denom <*

*newRacional(1,2).denom \* newRacional(2,3).numer*

→  $1 * 3 < 2 * 2$

→ *true*

# Ejemplos de evaluaciones de aplicaciones de métodos

- *newRacional(1, 2).numer*  
 $\rightarrow [][1/x, 2/y][\text{newRacional}(1, 2)/\text{this}]x$   
 $= 1$
- *newRacional(1, 2).menorQue(newRacional(2, 3))*  
 $\rightarrow [\text{newRacional}(2, 3)/r][1/x, 2/y][\text{newRacional}(1, 2)/\text{this}]\text{this.numer} * r.denom < \text{this.denom} * r.numer$   
 $= \text{newRacional}(1, 2).numer * \text{newRacional}(2, 3).denom <$   
 $\text{newRacional}(1, 2).denom * \text{newRacional}(2, 3).numer$   
 $\rightarrow 1 * 3 < 2 * 2$   
 $\rightarrow \text{true}$

# Operadores binarios

- En principio, con la clase *Racional* podemos hacer aritmética tal cual la hacemos con los enteros, por ejemplo.
- Pero, para quien va a usar esta abstracción, hay una gran diferencia:
  - Escribimos  $x + y$  si  $x$  y  $y$  son enteros,
  - Pero, escribimos  $r.suma(s)$  si  $r$  y  $s$  son racionales.
- En Scala se puede eliminar esa diferencia:
  - **Notación infija** para operadores binarios
  - **Flexibilización** en el nombramiento de los **identificadores**

# Operadores binarios

- En principio, con la clase *Racional* podemos hacer aritmética tal cual la hacemos con los enteros, por ejemplo.
- Pero, para quien va a usar esta abstracción, hay una gran diferencia:
  - Escribimos  $x + y$  si  $x$  y  $y$  son enteros,
  - Pero, escribimos  $r.suma(s)$  si  $r$  y  $s$  son racionales.
- En Scala se puede eliminar esa diferencia:
  - Notación infija para operadores binarios
  - Flexibilización en el nombramiento de los identificadores

# Operadores binarios

- En principio, con la clase *Racional* podemos hacer aritmética tal cual la hacemos con los enteros, por ejemplo.
- Pero, para quien va a usar esta abstracción, hay una gran diferencia:
  - Escribimos  $x + y$  si  $x$  y  $y$  son enteros,
  - Pero, escribimos  $r.suma(s)$  si  $r$  y  $s$  son racionales.
- En Scala se puede eliminar esa diferencia:
  - **Notación infija** para operadores binarios
  - **Flexibilización** en el nombramiento de los **identificadores**

# Notación infija para operadores binarios

Cualquier método que tenga un sólo parámetro del mismo tipo que la clase (o sea, que pueda ser visto como **operador binario**), puede ser usado como operador infijo.

- $r.suma(s)$  puede escribirse  $r\ suma\ s$
- $r.mult(s)$  puede escribirse  $r\ mult\ s$
- $r.menorQue(s)$  puede escribirse  $r\ menorQue\ s$
- $r.max(s)$  puede escribirse  $r\ max\ s$
- Y así sucesivamente...

# Flexibilización en el nombramiento de los identificadores

Los símbolos de los operadores pueden ser usados en los identificadores.

Un identificador puede ser:

- **Alfanumérico**: empieza con una letra, y es seguido por una secuencia de letras o números.
- **Simbólico**: empieza con un símbolo de operador, y seguido por otros símbolos de operador.
- El carácter guión bajo (`_`) cuenta como letra
- Los identificadores alfanuméricos pueden terminar en un guión bajo (`_`) seguido de algunos símbolos de operadores.
- Ejemplos: `suma`, `suma2`, `*`, `+ ? % &`, `vector_+`, `contador_ =`, ...

# Flexibilización en el nombramiento de los identificadores

Los símbolos de los operadores pueden ser usados en los identificadores.

Un identificador puede ser:

- **Alfanumérico**: empieza con una letra, y es seguido por una secuencia de letras o números.
- **Simbólico**: empieza con un símbolo de operador, y seguido por otros símbolos de operador.
- El carácter guión bajo (`_`) cuenta como letra
- Los identificadores alfanuméricos pueden terminar en un guión bajo (`_`) seguido de algunos símbolos de operadores.
- Ejemplos: `suma`, `suma2`, `*`, `+%&`, `vector_++`, `contador_ =, ...`



# Flexibilización en el nombramiento de los identificadores

Los símbolos de los operadores pueden ser usados en los identificadores.

Un identificador puede ser:

- **Alfanumérico**: empieza con una letra, y es seguido por una secuencia de letras o números.
- **Simbólico**: empieza con un símbolo de operador, y seguido por otros símbolos de operador.
- El carácter guión bajo (`_`) cuenta como letra
- Los identificadores alfanuméricos pueden terminar en un guión bajo (`_`) seguido de algunos símbolos de operadores.
- Ejemplos: `suma`, `suma2`, `*`, `+%&`, `vector_++`, `contador_ =, ...`

# Flexibilización en el nombramiento de los identificadores

Los símbolos de los operadores pueden ser usados en los identificadores.

Un identificador puede ser:

- **Alfanumérico**: empieza con una letra, y es seguido por una secuencia de letras o números.
- **Simbólico**: empieza con un símbolo de operador, y seguido por otros símbolos de operador.
- El carácter guión bajo (`_`) cuenta como letra
- Los identificadores alfanuméricos pueden terminar en un guión bajo (`_`) seguido de algunos símbolos de operadores.
- Ejemplos: `suma`, `suma2`, `*`, `+%&`, `vector_++`, `contador_ =, ...`

# Flexibilización en el nombramiento de los identificadores

Los símbolos de los operadores pueden ser usados en los identificadores.

Un identificador puede ser:

- **Alfanumérico**: empieza con una letra, y es seguido por una secuencia de letras o números.
- **Simbólico**: empieza con un símbolo de operador, y seguido por otros símbolos de operador.
- El carácter guión bajo (`_`) cuenta como letra
- Los identificadores alfanuméricos pueden terminar en un guión bajo (`_`) seguido de algunos símbolos de operadores.
- Ejemplos: `suma`, `suma2`, `*`, `+`, `? % &`, `vector_++`, `contador_ =, ...`

# La clase *Racional* con operadores simbólicos

```
0 class Racional(x: Int, y: Int){
1   require (y>0, "El denominador debe ser positivo")
2   private def mcd(a: Int, b: Int) : Int = if (b==0) a else mcd(b, a % b)
3   private val m=mcd(math.abs(x),y)
4   def numer = x/m
5   def denom = y/m
6   def +(r: Racional) = new Racional(r.numer*denom + r.denom*numer, r.denom*denom)
7   def *(r: Racional) = new Racional(numer*r.numer, denom*r.denom)
8   def -(r: Racional) = new Racional(numer*r.denom - denom*r.numer, r.denom*denom)
9   def /(r: Racional) = new Racional(numer*r.denom, denom*r.numer)
10  def ==(r: Racional) = if (numer*r.denom == denom*r.numer) true else false
11  def <(r: Racional) = this.numer*r.denom < this.denom*r.numer
12  def max(r: Racional) = if (this < r) r else this
13  override def toString = s"numer/denom"
14 }
15 val r1 = new Racional(1,2)
16 val r2 = new Racional(2,3)
17 val r3 = r1 + r2
18 r3.numer
19 r3.denom
20 r3.toString
21 r1 + r2
22 r1 / r2
23 r1 - r2
24 r1 == (new Racional(2,4))
25 r1 < r2
26 r1 max r2
27 r1*r1 + r2*r2
```

# El concepto de Herencia

- Otra piedra angular de la ingeniería de software es la **reutilización** de código.
- La POO responde a esta necesidad con el mecanismo de la **herencia**.
- La **herencia** es en esencia la capacidad de construir nuevas clases (nuevas abstracciones de datos) a partir de clases existentes. O sea, la herencia es un mecanismo para la **construcción incremental de abstracciones de datos**.
- Para este fin, Scala provee:
  - Clases abstractas
  - Herencia simple
  - Rasgos (*traits*)
- Conceptos clave: grafo de herencia, control de acceso y control de encapsulación

# El concepto de Herencia

- Otra piedra angular de la ingeniería de software es la **reutilización** de código.
- La POO responde a esta necesidad con el mecanismo de la **herencia**.
- La **herencia** es en esencia la capacidad de construir nuevas clases (nuevas abstracciones de datos) a partir de clases existentes. O sea, la herencia es un mecanismo para la **construcción incremental de abstracciones de datos**.
- Para este fin, Scala provee:
  - Clases abstractas
  - Herencia simple
  - Rasgos (*traits*)
- Conceptos clave: grafo de herencia, control de acceso y control de encapsulación

# El concepto de Herencia

- Otra piedra angular de la ingeniería de software es la **reutilización** de código.
- La POO responde a esta necesidad con el mecanismo de la **herencia**.
- La **herencia** es en esencia la capacidad de construir nuevas clases (nuevas abstracciones de datos) a partir de clases existentes. O sea, la herencia es un mecanismo para la **construcción incremental de abstracciones de datos**.
- Para este fin, Scala provee:
  - Clases abstractas
  - Herencia simple
  - Rasgos (*traits*)
- Conceptos clave: grafo de herencia, control de acceso y control de encapsulación

# El concepto de Herencia

- Otra piedra angular de la ingeniería de software es la **reutilización** de código.
- La POO responde a esta necesidad con el mecanismo de la **herencia**.
- La **herencia** es en esencia la capacidad de construir nuevas clases (nuevas abstracciones de datos) a partir de clases existentes. O sea, la herencia es un mecanismo para la **construcción incremental de abstracciones de datos**.
- Para este fin, Scala provee:
  - Clases abstractas
  - Herencia simple
  - Rasgos (*traits*)
- Conceptos clave: grafo de herencia, control de acceso y control de encapsulación



# El concepto de Herencia

- Otra piedra angular de la ingeniería de software es la **reutilización** de código.
- La POO responde a esta necesidad con el mecanismo de la **herencia**.
- La **herencia** es en esencia la capacidad de construir nuevas clases (nuevas abstracciones de datos) a partir de clases existentes. O sea, la herencia es un mecanismo para la **construcción incremental de abstracciones de datos**.
- Para este fin, Scala provee:
  - Clases abstractas
  - Herencia simple
  - Rasgos (*traits*)
- Conceptos clave: grafo de herencia, control de acceso y control de encapsulación

# Clases abstractas (1)

- La estructura discreta más sencilla y fundamental es el **conjunto**. Dos operaciones básicas sobre los conjuntos son:
  - *insertar*( $e, C$ ) que dado un elemento  $e$  y un conjunto  $C$ , construye el conjunto resultante de insertar  $e$  en  $C$ .
  - *pertenece*( $e, C$ ) que dado un elemento  $e$  y un conjunto  $C$ , devuelve un booleano indicando si  $e \in C$  o no.
- Otras operaciones básicas sobre los conjuntos son la unión, la intersección, y la diferencia de conjuntos.
- Para implementar una abstracción de datos que represente este concepto de conjunto, es importante que los lenguajes de programación provean **mecanismos de abstracción** que permitan especificar requerimientos, sin necesidad de detalles de implementación.
- Scala provee para ello el concepto de **clase abstracta**

# Clases abstractas (1)

- La estructura discreta más sencilla y fundamental es el **conjunto**. Dos operaciones básicas sobre los conjuntos son:
  - *insertar*( $e, C$ ) que dado un elemento  $e$  y un conjunto  $C$ , construye el conjunto resultante de insertar  $e$  en  $C$ .
  - *pertenece*( $e, C$ ) que dado un elemento  $e$  y un conjunto  $C$ , devuelve un booleano indicando si  $e \in C$  o no.
- Otras operaciones básicas sobre los conjuntos son la unión, la intersección, y la diferencia de conjuntos.
- Para implementar una abstracción de datos que represente este concepto de conjunto, es importante que los lenguajes de programación provean **mecanismos de abstracción** que permitan especificar requerimientos, sin necesidad de detalles de implementación.
- Scala provee para ello el concepto de **clase abstracta**

# Clases abstractas (1)

- La estructura discreta más sencilla y fundamental es el **conjunto**. Dos operaciones básicas sobre los conjuntos son:
  - *insertar*( $e, C$ ) que dado un elemento  $e$  y un conjunto  $C$ , construye el conjunto resultante de insertar  $e$  en  $C$ .
  - *pertenece*( $e, C$ ) que dado un elemento  $e$  y un conjunto  $C$ , devuelve un booleano indicando si  $e \in C$  o no.
- Otras operaciones básicas sobre los conjuntos son la unión, la intersección, y la diferencia de conjuntos.
- Para implementar una abstracción de datos que represente este concepto de conjunto, es importante que los lenguajes de programación provean **mecanismos de abstracción** que permitan especificar requerimientos, sin necesidad de detalles de implementación.
- Scala provee para ello el concepto de **clase abstracta**

# Clases abstractas (1)

- La estructura discreta más sencilla y fundamental es el **conjunto**. Dos operaciones básicas sobre los conjuntos son:
  - *insertar*( $e, C$ ) que dado un elemento  $e$  y un conjunto  $C$ , construye el conjunto resultante de insertar  $e$  en  $C$ .
  - *pertenece*( $e, C$ ) que dado un elemento  $e$  y un conjunto  $C$ , devuelve un booleano indicando si  $e \in C$  o no.
- Otras operaciones básicas sobre los conjuntos son la unión, la intersección, y la diferencia de conjuntos.
- Para implementar una abstracción de datos que represente este concepto de conjunto, es importante que los lenguajes de programación provean **mecanismos de abstracción** que permitan especificar requerimientos, sin necesidad de detalles de implementación.
- Scala provee para ello el concepto de **clase abstracta**

# Clases Abstractas (2)

- Para modelar los **conjuntos de enteros** podemos definir la siguiente **clase abstracta**:

```
0  abstract class ConjEnt{  
1    def insertar(x: Int): ConjEnt  
2    def pertenece(x: Int): Boolean  
3  }
```

- La clase abstracta se llama *ConjEnt*
- Las clases abstractas contienen miembros sin la implementación (en este caso *insertar*, y *pertenece*)
- No se pueden crear **instancias** de una clase abstracta (no se puede usar *new*)

# Plan

- 1 Generalidades
- 2 Construyendo abstracciones de datos
  - Clases y Objetos
  - Control de acceso y de encapsulación
  - Evaluación y operadores
- 3 Construyendo abstracciones de datos incrementalmente
  - Jerarquía de clases
  - Despacho dinámico (control de acceso)
- 4 Organización de clases en Scala

# Extendiendo las Clases

Supongamos que queremos implementar los conjuntos de enteros por medio de árboles binarios ordenados.

- Hay dos tipos de árboles posibles: (1) un árbol vacío, y (2) un árbol con un elemento en la raíz y el resto de elementos en los subárboles izquierdo (los elementos menores que la raíz) y derecho (los elementos mayores que la raíz)
- La clase para representar árboles (conjuntos) vacíos:

```
0 class Vacio extends ConjEnt {  
1   def pertenece(x:Int):Boolean= false  
2   def insertar(x:Int): ConjEnt = new NoVacio(x,  new Vacio, new Vacio)  
3 }
```

- La clase para representar árboles (conjuntos) no vacíos:

```
0 class NoVacio(elem: Int, izq:ConjEnt, der: ConjEnt) extends ConjEnt{  
1   def pertenece(x:Int):Boolean = {  
2     if (x < elem) izq pertenece x  
3     else if (x > elem) der pertenece x  
4     else true  
5   }  
6   def insertar(x:Int):ConjEnt = {  
7     if (x < elem) new NoVacio(elem, izq insertar x, der)  
8     else if (x > elem) new NoVacio(elem, izq, der insertar x)  
9     else this  
10  }  
11 }
```



# Implementación (de clases abstractas) y anulación (de métodos)

- Las definiciones de *insertar* y *pertenece* en las clases *Vacio* y *NoVacio* **implementan** las funciones abstractas de la clase *ConjEnt*.
- Pero aún, si la clase de la que se hereda (abstracta o no) ya tuviera un método implementado, pero se desea redefinirlo en la nueva clase, es posible hacerlo usando la anotación de **anulación** (*override*, en inglés):

```
0  abstract class Base {  
1    def foo=1  
2    def bar:Int  
3  }  
4  class Sub extends Base {  
5    override def foo = 2  
6    def bar = 3  
7  }
```

# Definición de objetos

- En el ejemplo de *ConjEnt*, se puede argumentar que no se necesita realmente una clase que genere conjuntos vacíos porque sólo hay un conjunto vacío.
- Parece exagerado, entonces, que el usuario deba crear múltiples instancias de él. Esto se puede resolver con una definición de objeto:

```
0 object Vacio extends ConjEnt {  
1   def pertenece(x: Int): Boolean = false  
2   def insertar(x: Int): ConjEnt = new NoVacio(x, Vacio, Vacio)  
3 }
```

- Esta definición, define un **único objeto** denominado *Vacio*.

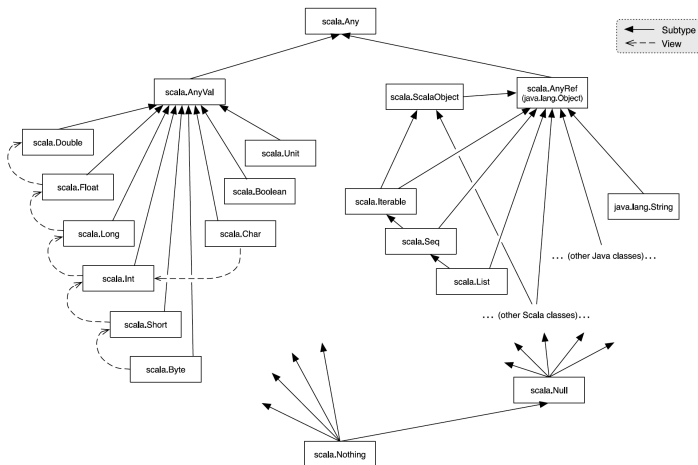
No se necesita, ni se puede, crear instancias de *Vacio*.

Los objetos únicos son valores, luego *Vacio* evalúa a sí mismo, pues ya es un valor.

# Vocabulario

- *Vacio* y *NoVacio* son clases que extienden la clase abstracta *ConjEnt*
- Se dice que los tipos *Vacio* y *NoVacio* **se ajustan** (conform, en inglés) al tipo *ConjEnt*: es decir, un objeto de tipo *Vacio* o *NoVacio* se puede usar en cualquier sitio donde se requiera un objeto de tipo *ConjEnt*
- *ConjEnt* se denomina la **superclase** de las clases *Vacio* y *NoVacio*.
- *Vacio* y *NoVacio* se denominan **subclases** de *ConjEnt*
- En Scala, toda clase definida por el usuario extiende alguna clase. Si no se explicita, extiende por defecto la clase *Object* de Java.
- Las superclases directas o indirectas de una clase *C* se denominan **clases base** de *C*. Las clases base de *NoVacio* son *ConjEnt* y *Object*

## Jerarquía de clases



# Traits (o sesgos)

- En Scala, la **herencia es sencilla** no múltiple. Una clase no puede heredar (extender) sino de una sola clase.
- Sin embargo, con frecuencia hay casos donde se requiere reutilizar código de varias fuentes, pues varias clases se ajustan a lo que se está diseñando. Para resolver esta necesidad, Scala provee los **Traits**
- Un **Trait** encapsula métodos y definiciones de campos que se reutilizan mezclándolos en las clases. Su sintaxis es muy similar a la de las clases abstractas:

```
0 trait Plano {  
1   def alto: Int  
2   def ancho: Int  
3   def area=alto * ancho  
4 }
```

# Un poco más sobre *Traits*

- Las clases, los objetos y los *traits* pueden heredar de a lo sumo una clase pero de muchos *traits*.

```
0 class Cuadrado extends Forma with Plano with Movable ...
```

- Los *traits*, se parecen a las interfaces en Java, pero son más poderosos porque pueden contener campos y métodos concretos.
- Los *traits* no pueden tener valores (val). Sólo se permiten a las clases.
- Como este curso no es de OO, no entraremos en más detalles sobre los *traits*.

# Plan

- 1 Generalidades
- 2 Construyendo abstracciones de datos
  - Clases y Objetos
  - Control de acceso y de encapsulación
  - Evaluación y operadores
- 3 Construyendo abstracciones de datos incrementalmente
  - Jerarquía de clases
  - Despacho dinámico (control de acceso)
- 4 Organización de clases en Scala

# Ligadura dinámica

- Los lenguajes OO (incluyendo a Scala) implementan lo que se denomina **despacho dinámico de métodos**
- Esto significa que el método que se ejecuta cuando se hace una invocación, depende del tipo que tiene el objeto al momento de esa ejecución.
- Usaremos el modelo de substitución para ilustrar este proceso:  
$$\text{Vacio.pertenece}(1)$$
$$\rightarrow [1/x][\text{Vacio}/\text{this}] \text{false}$$
$$= \text{false}$$



# Ligadura dinámica

- Los lenguajes OO (incluyendo a Scala) implementan lo que se denomina **despacho dinámico de métodos**
- Esto significa que el método que se ejecuta cuando se hace una invocación, depende del tipo que tiene el objeto al momento de esa ejecución.
- Usaremos el modelo de substitución para ilustrar este proceso:  
$$\text{Vacio.pertenece}(1)$$
$$\rightarrow [1/x][\text{Vacio}/\text{this}] \text{false}$$
$$= \text{false}$$

# Ligadura dinámica

- Los lenguajes OO (incluyendo a Scala) implementan lo que se denomina **despacho dinámico de métodos**
- Esto significa que el método que se ejecuta cuando se hace una invocación, depende del tipo que tiene el objeto al momento de esa ejecución.
- Usaremos el modelo de substitución para ilustrar este proceso:  
$$\text{Vacio.pertenece}(1)$$
$$\rightarrow [1/x][\text{Vacio}/\text{this}] \text{false}$$
$$= \text{false}$$

## Ligadura dinámica (2)

Otro ejemplo de evaluación, usando *NoVacio*:

```
new NoVacio(7, Vacio, Vacio).pertenece(7)
```

```
→ [7/x][7/elem, Vacio/izq, Vacio/der][newNoVacio(7, Vacio, Vacio)/this]
```

```
  if(x < elem) izq pertenece x else if(x > elem) der pertenece x else true
```

```
=   if(7 < 7) Vacio pertenece 7 else if(7 > 7) Vacio pertenece 7 else true
```

```
→ if(7 > 7) Vacio pertenece 7 else true
```

```
→ true
```

# Paquetes

- Las clases y los objetos se organizan en **paquetes**
- Para incluir una clase o un objeto dentro de un paquete, use la cláusula ***package*** al principio del archivo fuente:

```
0 package progfun.ejemplos
1
2 object Hello { ... }
```

Esto incluye a *Hello* en el paquete *progfun.ejemplos*.

- Ahora se puede invocar a *Hello* por su nombre *progfun.ejemplos.Hello*.

Por ejemplo para correr el programa:

```
0 scala> progfun.ejemplos.Hello
```

# Importaciones

Digamos que definimos la clase *Racional* en el paquete *progun.ejemplos*.

- Se puede usar la clase a través de su nombre completo:

```
0 val r = new progfun.ejemplos.Racional(1,2)
```

- De forma alternativa se puede usar un *import*:

```
0 import progfun.ejemplos.Racional  
1 val r = new Racional(1,2)
```

- Las importaciones se pueden hacer de diversas maneras:

```
0 import progfun.ejemplos.Racional           // importa solamente Racional  
1 import progfun.ejemplos.{Racional, Hello}  // importa Racional y Hello  
2 import progfun.ejemplos._                  // importa todo lo que haya en el paquete
```

# Importaciones automáticas

Algunas entidades se importan automáticamente en cualquier programa de Scala:

- Todos los miembros del paquete *scala*
- Todos los miembros del paquete *java.lang*
- Todos los miembros del objeto único *scala.Predef*

Estos son los nombres completos de algunos de los tipos y funciones que hemos usado:

0	<code>Int</code>	<code>scala.Int</code>
1	<code>Boolean</code>	<code>scala.Boolean</code>
2	<code>Object</code>	<code>java.lang.Object</code>
3	<code>require</code>	<code>scala.Predef.require</code>
4	<code>assert</code>	<code>scala.Predef.assert</code>