



Paradigmas de Programación (EIF-400) Introducción con el JDK8

CARLOS LORÍA-SÁENZ

JULIO 2017

EIF/UNA

Objetivos



2

- ▶ Un salto en el contexto del curso
- ▶ Un vistazo general a los Temas
- ▶ Noción de paradigma y relación con lenguajes y traducción
- ▶ Evolución de los lenguajes (tendencias, mercado, historia)
- ▶ Tour sobre distintos lenguajes y paradigmas
- ▶ Esbozo de historia de algunos
- ▶ Clases de Paradigmas y origen. Ejemplos
- ▶ Consideraciones de diseño. Balance implementación (compilación, IDE, concurrencia)
- ▶ Aspectos sobre compilación relevantes

Conceptos Énfasis

3

- ▶ Contexto
- ▶ Paradigma, Lenguaje, Programación
- ▶ Declarativo vs Operativo: Balance Qué versus Cómo
- ▶ Compilación/Interpretación: Parsing/Typing.
Dinámico/estático
- ▶ Máquinas virtuales.
- ▶ Introducción a la FP/LP
- ▶ Evolución paradigmas/lenguajes
- ▶ Patrones (OOP-)imperativo vs (OOP)-FP-declarativos
- ▶ Niveles de abstracción
- ▶ Perfil de lenguajes icónicos y su historia

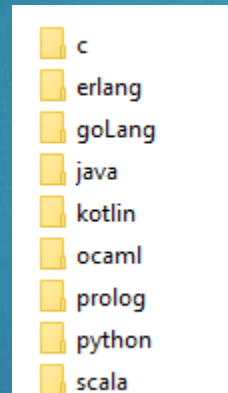
Ejercicios

- ▶ Se incluyen preguntas y ejercicios para realizar como complemento al contenido
- ▶ Tareas en distintas plataformas y herramientas (Java, Go, Kotlin, C, Python, Ocaml, Erlang Assembler)
- ▶ Herramientas Objdump, javap, dis
- ▶ Pueden ser potenciales quices/preguntas examen

Material de apoyo

5

- ▶ Ver material introductorio_2017.rar en el sitio



Contexto del curso

6

- ▶ Programa (carta al estudiante) del curso
- ▶ Dos sitios interesantes
 - ▶ Historia (genealogía)
 - ▶ Lista (> 2000 y sigue contando)
 - ▶ Popularidad (índices basados en Web)
- ▶ Revisemos los primeros en el ranking Tiobe

Jun 2017	Jun 2016	Change	Programming Language	Ratings	Change
1	1		Java	14.493%	-6.30%
2	2		C	6.848%	-5.53%
3	3		C++	5.723%	-0.48%
4	4		Python	4.333%	+0.43%
5	5		C#	3.530%	-0.26%
6	9	▲	Visual Basic .NET	3.111%	+0.76%
7	7		JavaScript	3.025%	+0.44%
8	6	▼	PHP	2.774%	-0.45%
9	8	▼	Perl	2.309%	-0.09%
10	12	▲	Assembly language	2.252%	+0.13%
11	10	▼	Ruby	2.222%	-0.11%
12	14	▲	Swift	2.209%	+0.38%
13	13		Delphi/Object Pascal	2.158%	+0.22%
14	16	▲	R	2.150%	+0.61%
15	48	▲▲	Go	2.044%	+1.83%

Very Long Term History

To see the bigger picture, please find below the positions of the top 10 programming languages of many years back. Please note that these are *average* positions for a period of 12 months.

Programming Language	2017	2012	2007	2002	1997	1992	1987
Java	1	1	1	1	12	-	-
C	2	2	2	2	1	1	1
C++	3	3	3	3	2	2	4
C#	4	4	7	17	-	-	-
Python	5	7	6	11	27	-	-
Visual Basic .NET	6	19	-	-	-	-	-
JavaScript	7	9	8	8	19	-	-
PHP	8	6	4	5	-	-	-
Perl	9	8	5	4	4	11	-
Assembly language	10	-	-	-	-	-	-
COBOL	25	28	17	9	3	10	8
Lisp	31	12	14	12	9	5	2
Prolog	33	35	26	15	18	14	3
Pascal	104	15	19	97	8	3	5

Programming Language Hall of Fame

The hall of fame listing all "Programming Language of the Year" award winners is shown below. The award is given to the programming language that has the highest rise in ratings in a year.

Year	Winner
2016	🏆 Go
2015	🏆 Java
2014	🏆 JavaScript
2013	🏆 Transact-SQL
2012	🏆 Objective-C
2011	🏆 Objective-C
2010	🏆 Python
2009	🏆 Go
2008	🏆 C
2007	🏆 Python
2006	🏆 Ruby
2005	🏆 Java
2004	🏆 PHP
2003	🏆 C++

[← New Style for User Groups](#)[Kotlin Future Features Survey Results →](#)

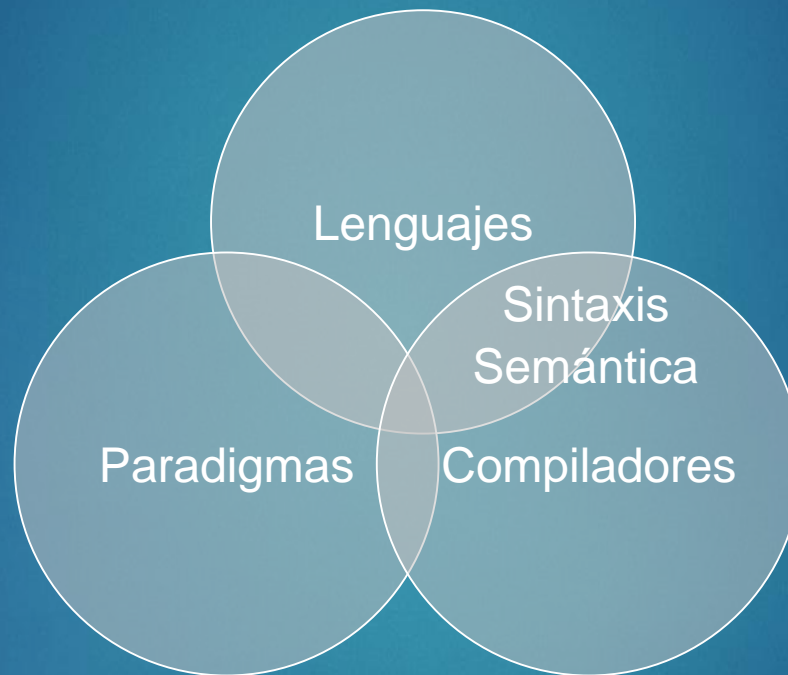
Kotlin on Android. Now official

Posted on May 17, 2017 by Maxim Shafirov



Today, at the Google I/O keynote, the Android team announced first-class support for Kotlin. We believe this is a great step for Kotlin, and fantastic news for Android developers as well as the rest of our community. We're thrilled with the opportunities this opens up.

Contexto: Paradigma, Lenguaje, Compilador



Compilador = Traductor

Problema: Barrera semántica

12



Conceptos
Humanos

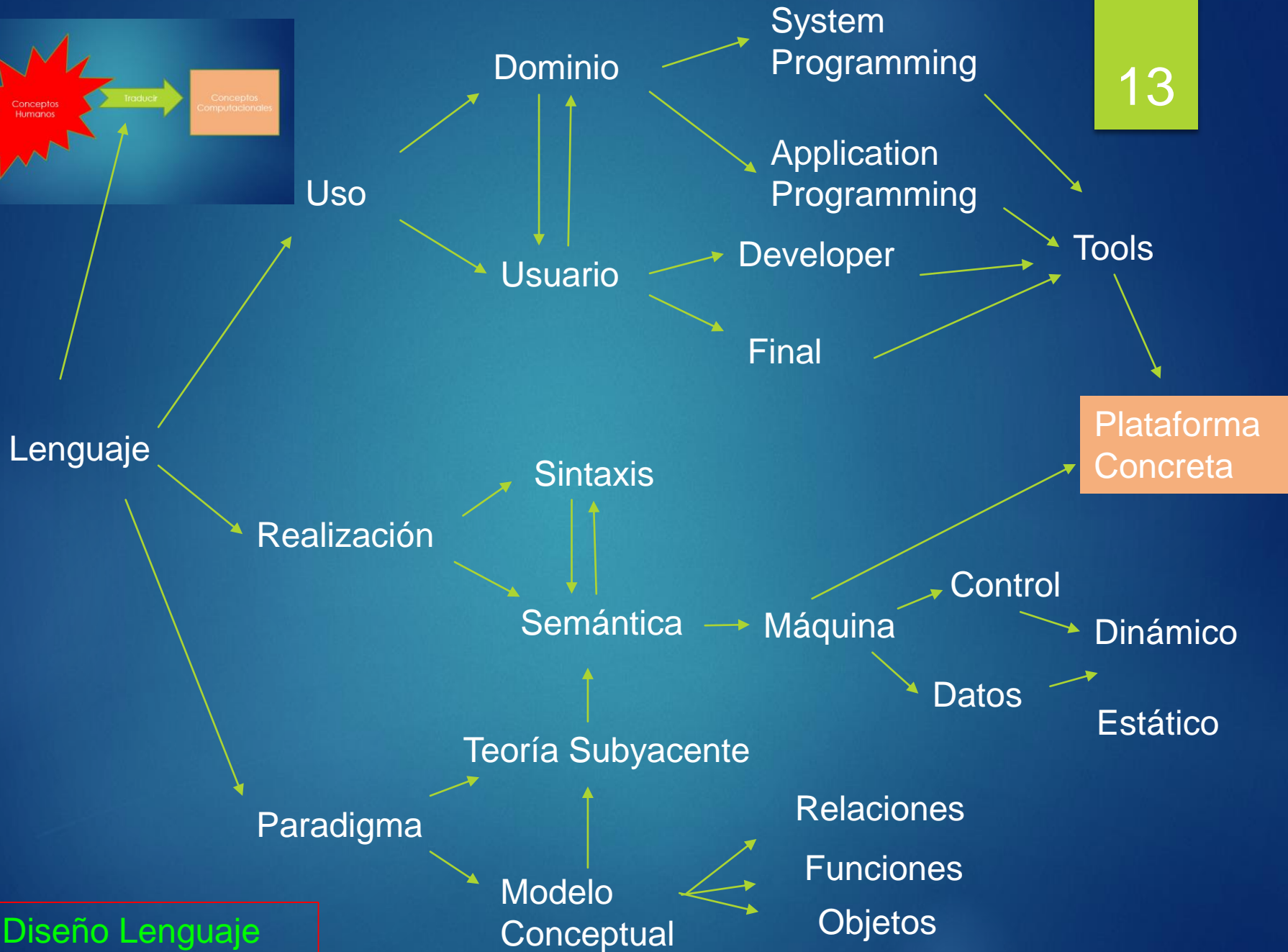
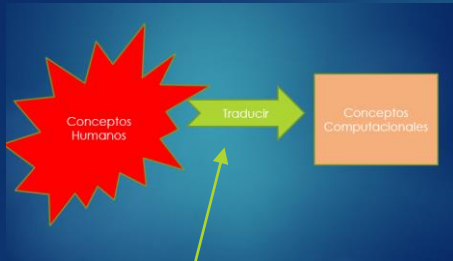
Expresar


Conceptos
Computacionales

Lenguaje de
Programación

Paradigma









WIKIPEDIA
La enciclopedia libre

[Portada](#)

Crear una cuenta  Ingresar

Artículo **Discusión**

Leer Editar Ver historial

Buscar 

Paradigma

El término **paradigma** significa «*ejemplo*» o «*modelo*». En todo el ámbito [científico](#), [religioso](#) u otro contexto [epistemológico](#), el término **paradigma** puede indicar el concepto de *esquema formal* de organización, y ser utilizado como sinónimo de *marco teórico* o *conjunto de teorías*.



Un **paradigma** es como unos anteojos que me permiten percibir una realidad: resolver un problema computacional

¿Cómo paso de *requerimientos* (muchas veces imprecisos) a *programas* (que son totalmente precisos)?

Paradigmas

15

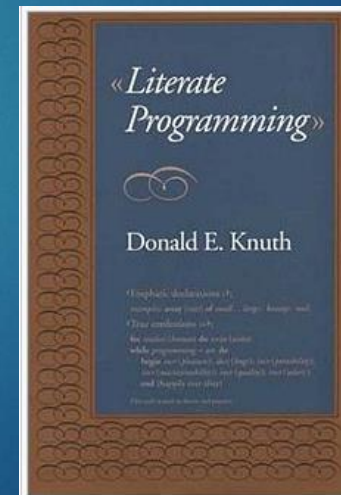
- ▶ Paradigmas en Programación: son enfoques computacionales para abstracción y representación del dominio de estudio (mundo)
- ▶ ¿Cómo modelar el mundo (requerimientos)?
- ▶ Conceptos para expresar computación
- ▶ Ejemplos: “objeto”, “función”, “regla”
- ▶ Una forma de “expresar las cosas” y patrones a seguir, métodos, enfoques, estilos acordes a ese enfoque
- ▶ El lenguaje de programación trata de *facilitar* esos conceptos en su *sintaxis* y *semántica*



¿Programar?

16

- ▶ ¿Qué es “programar”?
- ▶ Reflexione sobre esta frase:
- ▶ D. Knuth: “*Programming is the art of telling another human being what one wants the computer to do*”
- ▶ ¿Implicación sobre un lenguaje de programación?



- ¿Qué hace este programa?

```
#include <stdio.h>
#include <stdlib.h>
#define $ 0
#define $$ 1
void main(int $_, char* $__[]) {
    int _=$;
    if ($_==$+$$_){
        _=atoi($__[$$]);
    } else return;
    if (_>=$$);
    else return;
    int __=$$;
    for(int ____=_; ____;__*=(__--));
    printf("%d --> %d\n",_,__);
}
```

Datos + Control + Máquina = Programa

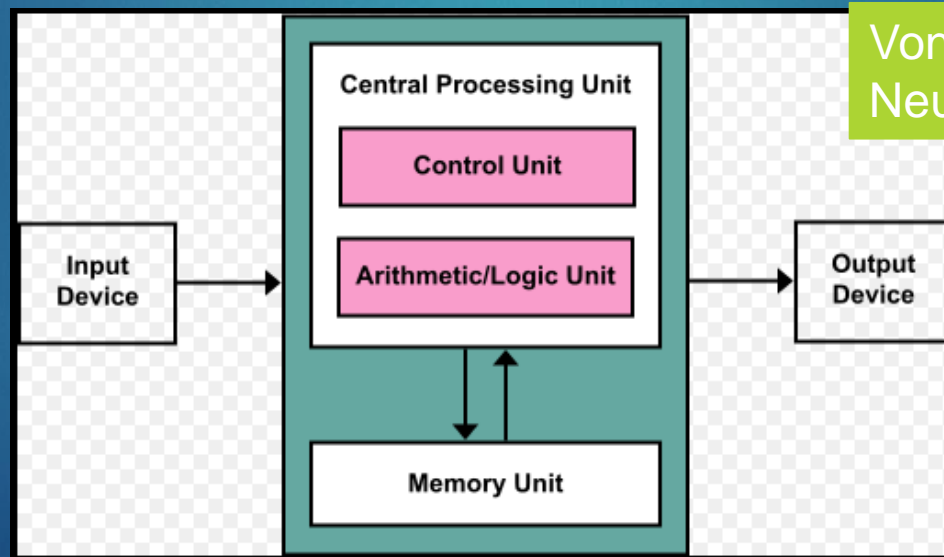
- ▶ Describa la arquitectura de una aplicación Web tres capas
- ▶ ¿En qué enfocarse?
- ▶ Datos = Sustantivos (memoria)
- ▶ Control = Verbos (operaciones)
- ▶ Máquina = Cliente+WebServer+DataServer



“Máquina Subyacente”

19

- ▶ Clásica Imperativa: Modelo Von Neumann (Turing Machine)
- ▶ Cálculo Lambda: Programación Funcional (FP)
- ▶ Prueba de Teoremas: Programación Lógica (LP)



Von
Neumann

De Wikipedia

Modelo Von Neumann

20

- ▶ Modelo conceptual que simplifica la arquitectura
- ▶ CPU (unidad de operaciones)
- ▶ Memoria(s) (áreas de almacenamiento)
- ▶ Memoria sirve para datos y programas
- ▶ “Traer” operación actual (fetch, decode, execute)
- ▶ Operaciones para transferir datos entre memoria(s) al/del CPU (load, store)
- ▶ Operaciones lógico-aritméticas (add, compare)
- ▶ Operaciones para bifurcar control a un punto (branch, jump)

Máquinas en FP/LP

21

- ▶ Modelos “poco convencionales” con respecto von Neumann
- ▶ **FP:** Máquina es un simplificador de expresiones
- ▶ Cumple “leyes” del cálculo Lambda
- ▶ Veremos después
- ▶ **LP:** Máquina busca una prueba de un teorema
- ▶ Cumple “leyes” de lógica

Ejemplo: “Secuencia”

22

- ▶ Natural: “Haga S_1 y después S_2 ”
- ▶ Imperativo: $S_1; S_2$
- ▶ Funcional : $S_2(S_1)$
- ▶ Lógico: $S_1 \wedge S_2$
- ▶ Razonar en “imperativo” es complejo

Ejercicio: Razonar/depurar

- Observe que `fibonacci(n)` falla en calcular `fibonacci(n)`
- Corrija la pulga de forma minimalista.
- Hágalo de dos maneras distintas

$$fib(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ fib(n-1) + fib(n-2) & \text{sino} \end{cases}$$

```
long fib( int n ) {
    return n <= 1 ? 1 : fib( n - 1 ) + fib ( n - 2 );
}
```

```
public long fibo( int n ){
    // returns fib(n)
    if ( n <= 1 )
        return 1;
    long a = 2, b = 1;
    for (int i = 2; i <= n; i++){
        long t = a;
        a += b;
        b = t;
    }
    return a;
}
```



Java Assertions

☐ Símbolo del sistema

```
PP:java -ea Fibo 5
fibonacci( 0 ) = 1 =? 1 = fibonacci( 0 )
fibonacci( 1 ) = 1 =? 1 = fibonacci( 1 )
Exception in thread "main" java.lang.AssertionError
    at Fibo.main(Fibo.java:26)
```

Facetas Control Alto nivel

24



- Se combinan con lo paradigmático: OOP, FP, LP

Tipos de Paradigmas:

imperativo no estructurado

- ▶ Imperativo con verbos lógico/aritméticos (add, cmp, goto, ...)
- ▶ No hay estructuras de control (while, if-then-else)
- ▶ Control: goto, eventualmente condicional
- ▶ Datos: celdas de memoria y registros de máquina (CPU)
- ▶ Ejemplares: Assembler, primeros versiones lenguajes “alto nivel” antes de 1970 (Fortran, Cobol)

Tipos de paradigmas: Imperativo (alto nivel)

- ▶ Imperativo estructurado en control:
 - ▶ Control son verbos a la máquina subyacente (implícita): `=`, `while`, `if-then-else`, etc
 - ▶ Abstracción de control: funciones (procedures, subrutinas) `call/return`
- ▶ Imperativo plano en datos:
 - ▶ Datos primitivos planos (`int`, `float`)
 - ▶ Abstracción Datos estructurados: `struct`, `[]`
 - ▶ Pointers explícitos `*`. Memoria global (salvo `struct`)
 - ▶ Estado mutable por doquier
- ▶ Control y Funciones conceptos muy separados
- ▶ Típicos ejemplares: `C` `Pascal`. `Fortran`/`Cobol`

Tipo de Paradigmas: imperativo OOP

- ▶ Orientado a objetos en datos (declarativo)
 - ▶ La memoria se reparte en objetos (generaliza struct)
 - ▶ Puede ser manejada automáticamente (colector de basura)
 - ▶ Eventualmente `class`
 - ▶ Eventualmente herencia/polimorfismo
- ▶ Control se abstrae en métodos que pertenecen a los objetos (verbos de objetos)
- ▶ Métodos internamente tienen funcionalidad imperativa tradicional
- ▶ Función no es de “primer piso”: no es tipo de objeto
- ▶ Ejemplares C++ (< C11-14), Java (< jdk8)

Excepción Interesante

28

- ▶ EL “assembler” (byte code) de la JVM es orientado a objetos
- ▶ ¿Por qué? Eso facilita la compilación de `.java` a `.class`

Tipo de Paradigmas: funcional (imperativo)

- ▶ Plano en datos: primitivos usuales, pero símbolos y listas
- ▶ La función es un tipo de dato (lambda)
- ▶ En control composición y recursión muy promovida
- ▶ Inmutabilidad es promovida (transparencia-referencial)
- ▶ Pero pueden ser imperativos/mutadores (impuros)
- ▶ Pueden ser dinámicos/estáticos. Último caso inferencia de tipos!
- ▶ Abstracción de datos usando “`data types`” (generaliza `struct`, `record`, pero no es `class`)
- ▶ *Pattern-matching*: un parámetro de una función puede ser una estructura de datos
- ▶ Ejemplares: `Lisp`/`Scheme`, `ML` (`Ocaml`, `F#`), `Haskell` (puro)

Data types (Ocaml)

30

- ▶ Árboles de Expresiones (expr) aritméticas y función to_string. Pruebe [acá](#)

```
type expr =  
  | Plus of expr * expr          (* means a + b *)  
  | Minus of expr * expr         (* means a - b *)  
  | Times of expr * expr         (* means a * b *)  
  | Divide of expr * expr        (* means a / b *)  
  | Value of string              (* "x", "y", "n", etc. *)  
;;  
  
let rec to_string e =  
  match e with  
  | Plus (left, right) ->  
    "(" ^ to_string left ^ " + " ^ to_string right ^ ")"  
  | Minus (left, right) ->  
    "(" ^ to_string left ^ " - " ^ to_string right ^ ")"  
  | Times (left, right) ->  
    "(" ^ to_string left ^ " * " ^ to_string right ^ ")"  
  | Divide (left, right) ->  
    "(" ^ to_string left ^ " / " ^ to_string right ^ ")"  
  | Value v -> v  
;;
```

Pattern-matching

Tipos de paradigmas: Lógico

- ▶ Datos usuales, simbólicos (listas)
- ▶ Declarativo uniforme en ambos datos/control: regla declara dato y operación (relación)
- ▶ Se promueve recursión
- ▶ Máquina no es Von-Neuman
- ▶ Control es automático (backtracking)
- ▶ Garbage-collected
- ▶ Algo como pattern matching: unificación
- ▶ Hay primariamente relaciones
- ▶ Puede ser OO. Puede ser imperativo/mutable
- ▶ Típico ejemplar: `Prolog`

Variantes/Combinaciones

32

- ▶ Funcional/OO: Scala
- ▶ Funcional concurrente (reactivo): Erlang, Elixir, Scala
- ▶ Imperativo concurrente no muy OO: Go
- ▶ OO/Funcional/Lógico/DSL: [Clips](#)



33



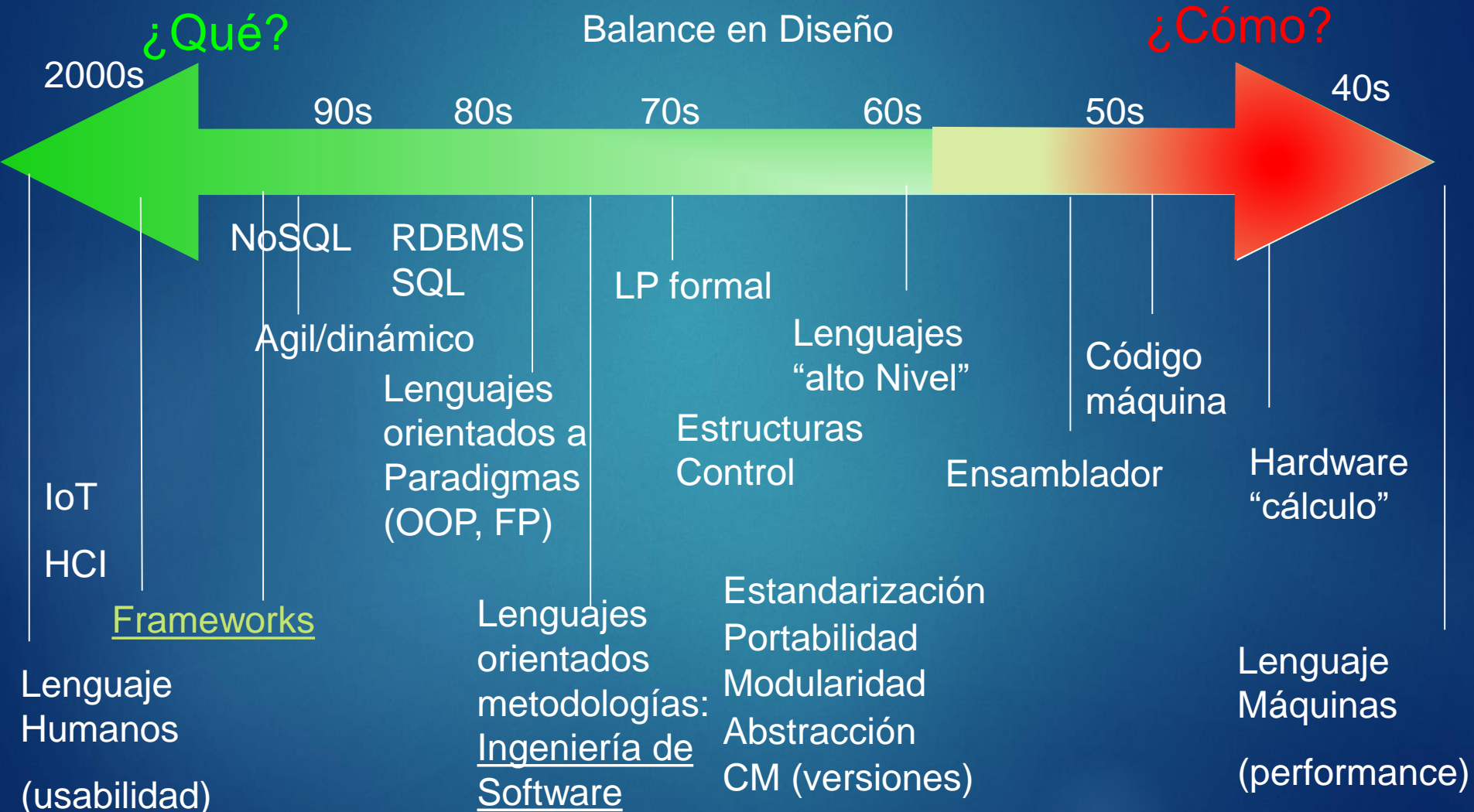
Capas de abstracción

34



Evolución Escala Semántica

35



Lenguaje de Programación

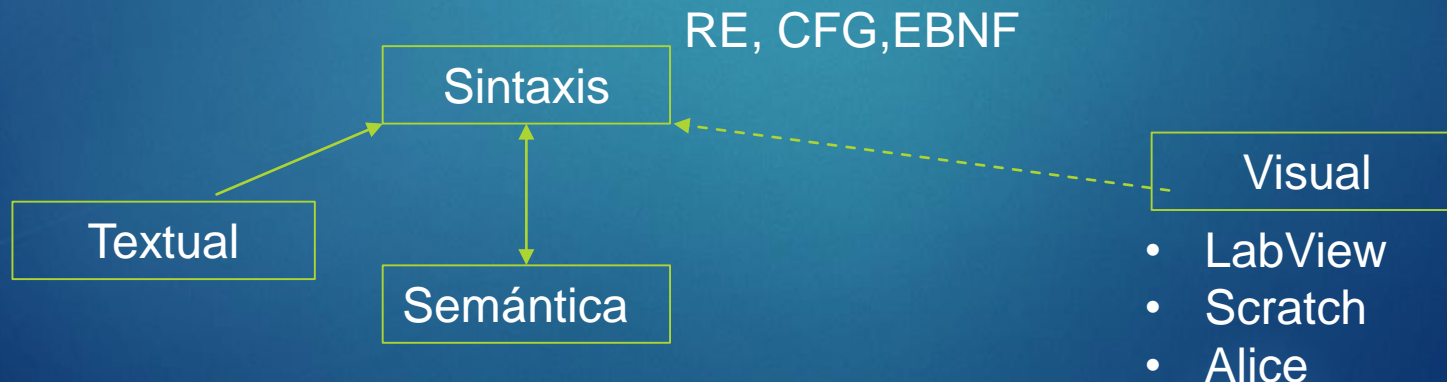
36

Article Talk

Programming language

From Wikipedia, the free encyclopedia

A **programming language** is a formal computer language or constructed language designed to communicate instructions to a machine, particularly a computer. Programming languages can be used to create programs to control the behavior of a machine or to express algorithms.



Lenguajes: áreas tipos

37

- ▶ Propósito general/específico: usuario
- ▶ Aplicaciones empresariales (usuario final)
 - ▶ Web, móvil
 - ▶ Office automation
 - ▶ Data Mining/Machine learning
- ▶ Programación de sistemas (developer)
 - ▶ Sistemas operativos (incluye empotrados)
 - ▶ Servidores (Web, DB)
 - ▶ IDE, tools, SDKs
 - ▶ Frameworks
 - ▶ Juegos

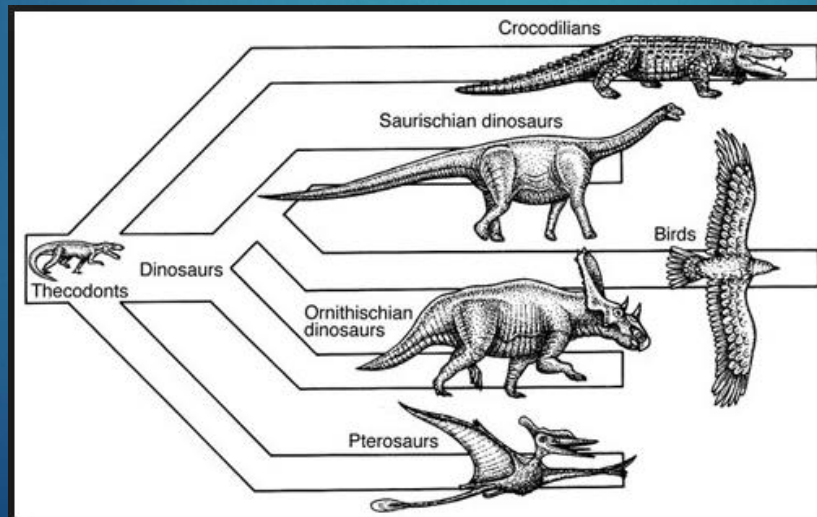
¿"Poder" de un Lenguaje de Programación?

- ▶ Reflexione sobre esta pregunta: ¿Es CSS un lenguaje de programación?
- ▶ ¿A qué paradigma se adhiere?
- ▶ ¿HTML?
- ▶ ¿Por qué es necesario JS?
- ▶ ¿SQL?
- ▶ ¿Cuál es la parte operacional de SQL?
- ▶ Concepto a investigar: Turing completo

Más de 2000 ¿Por qué tantos?

39

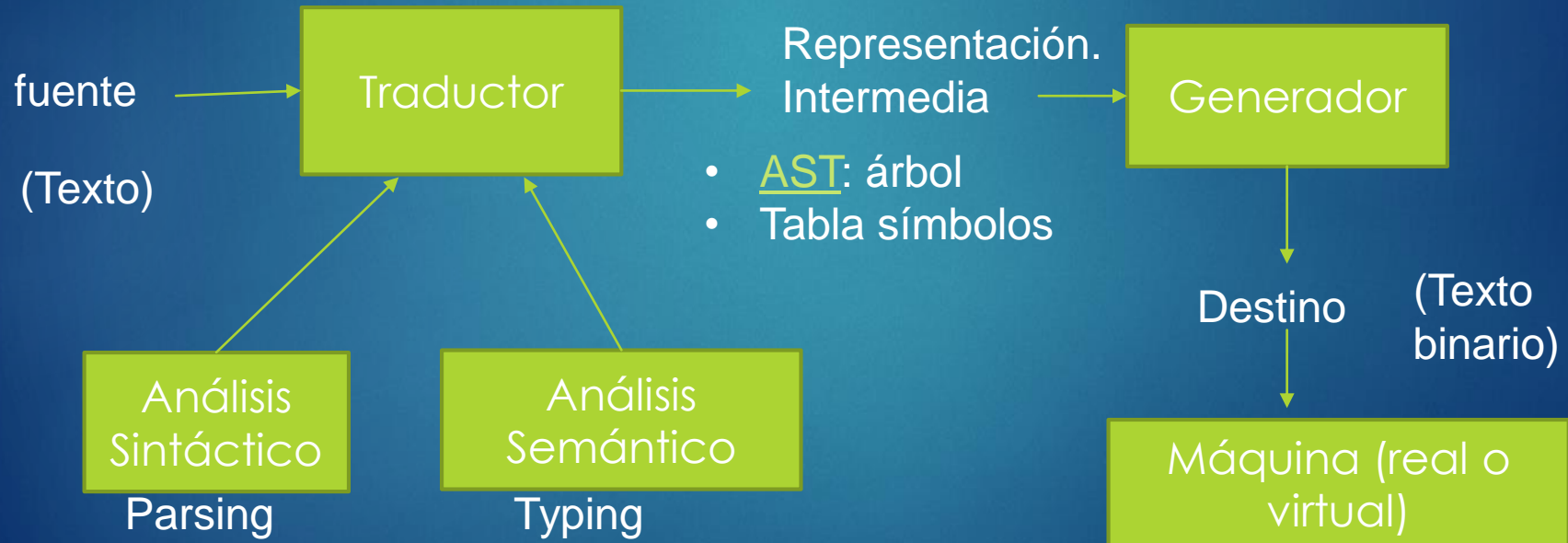
- ▶ Evolución social-histórico-tecnológica
- ▶ Dinosaurios que no se extinguen
- ▶ Creencias personales (Comunidades muy fuertes)
- ▶ Competencias entre la industria
- ▶ Influencia académica, nuevas ideas



Compilador/Intérprete

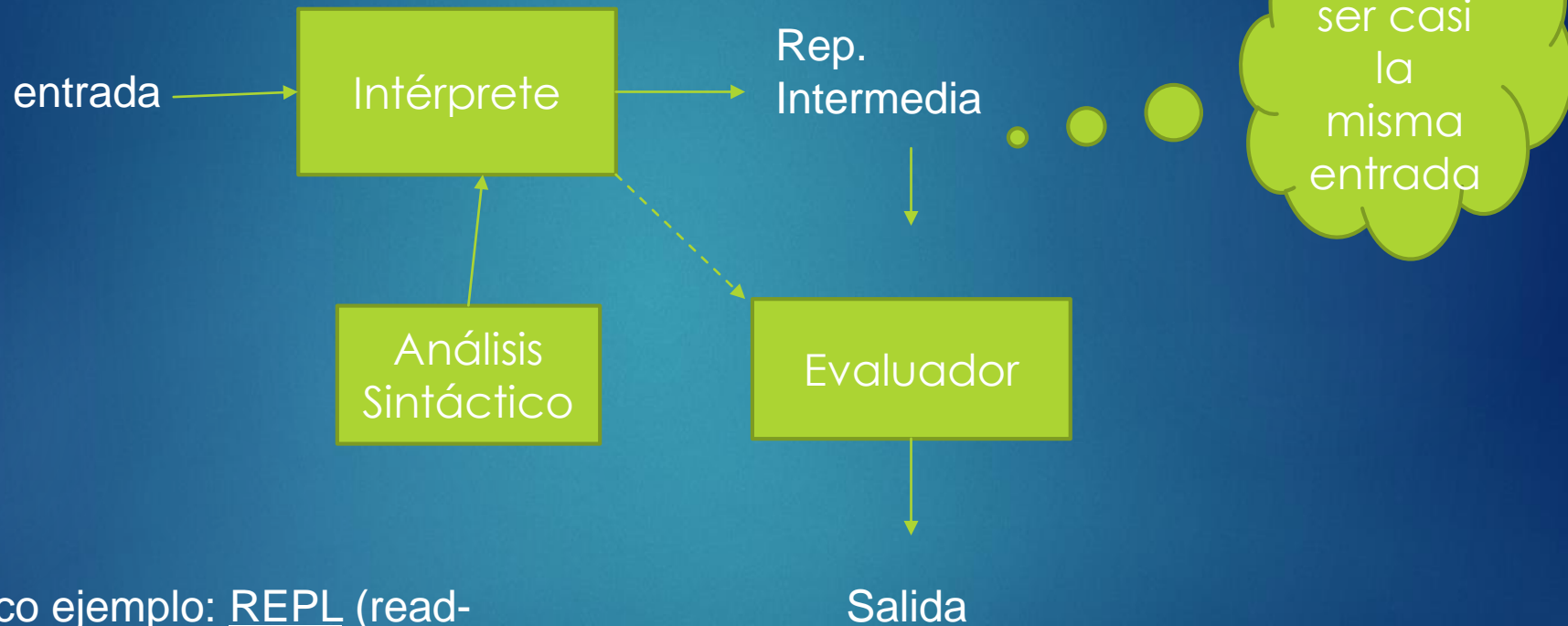
40

A **compiler** is a **computer program** (or a set of programs) that transforms **source code** written in a **programming language** (the source language) into another computer language (the target language), with the latter often having a binary form known as **object code**.^[1] The most common reason for converting source code is to create an **executable** program.



Intérprete (usualmente)

41



Típico ejemplo: REPL (read-eval-print loop)

Ejemplos: REPL aka Shells

42

- ▶ Node Shell
- ▶ Mongo Shell
- ▶ Nashorn (jjs) Shell
- ▶ Java9 Java Shell
- ▶ Kotlinc
- ▶
- ▶ También playgrounds:
 - ▶ Por ejemplo [go](https://play.golang.org/)
 - ▶ [Kotlin](https://play.kotlinlang.org/)
 - ▶ [jsfiddle](https://jsfiddle.net/)



```
GA. Símbolo del sistema - node

JS:node
> a = 666
666
> a + a
1332
>
```

```
GA. Símbolo del sistema - mongo -nodb

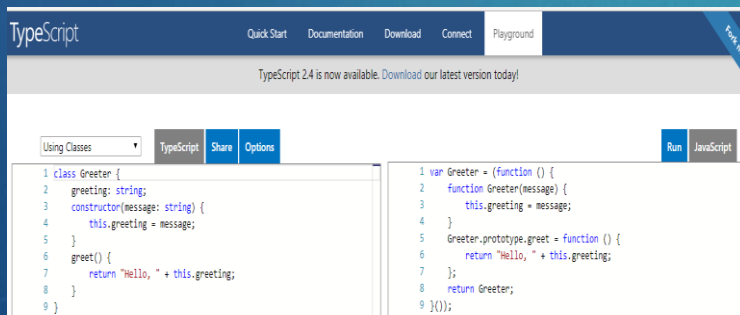
JS:mongo -nodb
MongoDB shell version: 3.2.7
> a = 666
666
> a + a
1332
>
```

```
< → ↺ Es seguro https://play.golang.org/
Aplicaciones ★ Bookmarks paradigms pers
The Go Playground Run Format Import
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     fmt.Println("Hello, playground")
9 }
10
```


Elemento interesante: Trans(compilación)

43

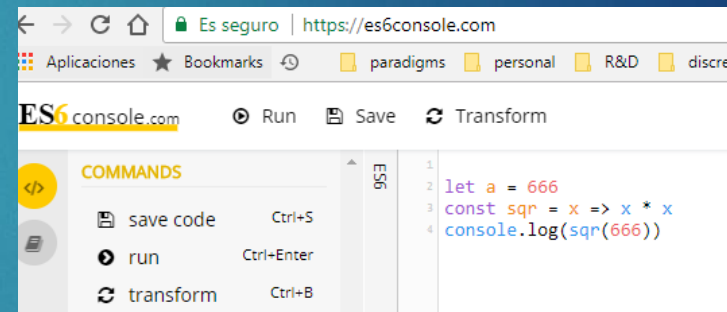
- ▶ Fuente y Destino son lenguaje fuente
- ▶ Ejemplos Less, Sass → CSS
- ▶ ES6 → ES5: Pruebe [acá ES6 Console](#)
- ▶ [Typescript](#) → Javascript



The screenshot shows the TypeScript Playground interface. On the left, the TypeScript source code defines a class `Greeter` with a `greeting` property, a `constructor` that sets `this.greeting`, and a `greet` method that returns a string. On the right, the compiled JavaScript output is shown, which uses a function constructor to create the `Greeter` object and its prototype.

```
1 class Greeter {  
2   greeting: string;  
3   constructor(message: string) {  
4     this.greeting = message;  
5   }  
6   greet() {  
7     return "Hello, " + this.greeting;  
8   }  
9 }
```

```
1 var Greeter = (function () {  
2   function Greeter(message) {  
3     this.greeting = message;  
4   }  
5   Greeter.prototype.greet = function () {  
6     return "Hello, " + this.greeting;  
7   };  
8   return Greeter;  
9 }());
```



The screenshot shows the ES6 Console web application. The browser address bar displays `https://es6console.com`. The interface includes a 'Run' button and a 'Transform' button. The 'COMMANDS' panel on the left lists 'save code' (Ctrl+S), 'run' (Ctrl+Enter), and 'transform' (Ctrl+B). The main editor shows ES6 code being transformed into ES5 code.

```
1 let a = 666  
2 const sqr = x => x * x  
3 console.log(sqr(666))
```

Sopa de Elementos Relevantes

- ▶ “Máquinas virtuales y frameworks” (multiplataforma vs nativo)
- ▶ JVM y CLR (.Net) (memoria manejada, GC)
- ▶ ART (Android)
- ▶ Compilación: JIT vs AOT (Java9)
- ▶ “*Write Once Run Anywhere*” (WORA)
- ▶ Los browsers: son la VM de JS de cliente
- ▶ Web basado en “lenguajes script” ¿Por qué no también el server? (Node.js)

Dinámico versus Estático

45

- ▶ Lenguajes compilados: dos tiempos
 - ▶ Tiempo compilación pesado (tipificación estática “pesada”)
 - ▶ Tiempo ejecución (tipificación dinámica “liviana”). Sólo existe si hubo compilación
- ▶ Lenguajes interpretados (dinámicos)
 - ▶ Tiempo “compilación” débil o hasta inexistente
 - ▶ Tiempo de ejecución con tipificación dinámica “fuerte” durante la corrida
- ▶ Términos a revisar: fuertemente-tipado, débilmente tipado, “duck-typing”, “late binding”, “inferencia”

Ejercicio: Explique la diferencia

- Compare las dos corridas: `-Xint` inhibe JIT

```
ca. Símbolo del sistema

PP:java Fibo 40 40
fibo( 40 ) = 267914296 =? 165580141 = fib( 40 )

Elapsed(max=40) 605 ms

PP:java -Xint Fibo40 40
fibo( 40 ) = 267914296 =? 165580141 = fib( 40 )

Elapsed(max=40) 9850 ms

PP:
```

Paradigma y Lenguaje

47

- ▶ ¿Por qué nace un “paradigma” P ?
- ▶ ¿Por qué nace un lenguaje L para P ?
- ▶ ¿Qué tan “puramente” refleja L a P ?
- ▶ ¿Por qué evoluciona?
- ▶ ¿Se reproduce?
- ▶ ¿Muere?

Tópicos entrelazados

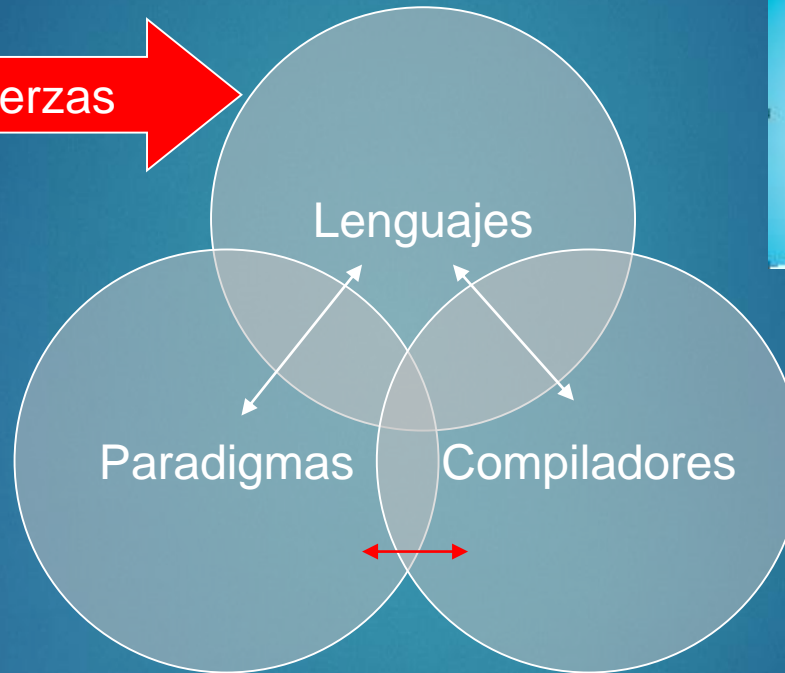
48

Avance
tecnológico

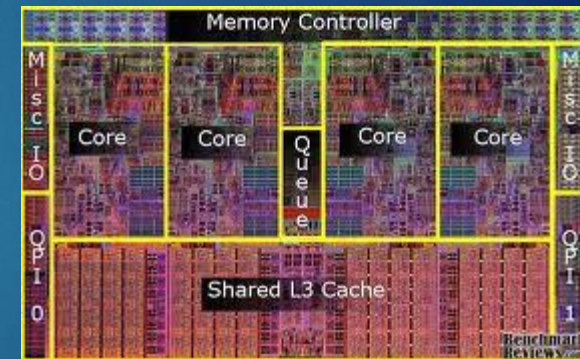
Necesidades
(mercado:
Data mining)

Desarrollo
veloz y ágil

Fuerzas

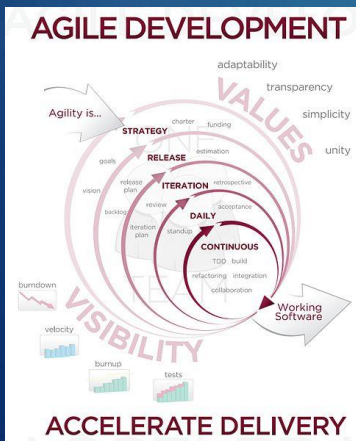


Web/ Móvil



Concurrencia
(multicore)

Java, .Net ART
Virtual machines



¿Y cloud computing?

49

- ▶ Nube == Sistema Operativo/Plataforma
- ▶ No toda "fuerza" tecnológica paradigmática crea un lenguaje
- ▶ Realización arquitecturas/plataformas/servicios más que con lenguajes
- ▶ No ha "permeado" tanto al nivel de lenguaje
- ▶ Puede implicar SO, Almacenamiento (infraestructura) , Contenedores de aplicaciones
- ▶ Ejercicio: mencionar tres plataformas para Cloud Computing

Escalable: Factor relacionado volumen

- ▶ Escalabilidad: capacidad de seguir funcionando cuando la volumen de trabajo crece
- ▶ Horizontal: “más nodos en un cluster”: el programa se puede distribuir fácilmente
- ▶ Vertical: Más cpus/memoria a un mismo nodo: el programa incorpora los nuevos recursos eficientemente

Factores adicionales

51

- ▶ Evolución de arquitecturas (desktop, móvil)
- ▶ Capacidad de compilación eficiente
- ▶ Ambiente de desarrollo (IDE)
- ▶ Productividad
- ▶ Mantenibilidad
- ▶ Curva de aprendizaje (novatos → avanzados)
- ▶ Estandarización (multiplataforma)
- ▶ Open-source (aporte comunitario)
- ▶ Uso en mega-empresas de Software

DSL: Otra faceta

52

- ▶ DSL: lenguajes de dominio específico
- ▶ Lenguajes que son para una clase concreta de aplicación
- ▶ Lo contrario de lenguaje de propósito general
- ▶ Por ejemplo: SQL, HTML, CSS , ANTLR, Latex/Tex (hay muchos)
- ▶ Hay lenguajes de propósito general que permiten una faceta DSL: C#, Groovy, Scala Kotlin

Patrones de Iteración y Control

- ▶ ¿Quién lleva el control”
- ▶ El lenguaje versus la colección
- ▶ Ejemplos: Implementar en JS/Python

$$s = \sum_{i=0}^n a_i$$

$$S = \{x^2 : \exists y (y \in N \wedge x = 2y + 1)\}$$

Resultado: Mezcla paradigmática

- ▶ Java8: lambdas: métodos como objetos de primera-clase) (métodos anónimas)
- ▶ A la Scheme/Lisp/ML
- ▶ ¿No es una función también un objeto?
- ▶ ¿Está el modelo OO “incompleto”?
- ▶ Formas de expresión más elegantes y compactas (“azúcar sintáctico”)
- ▶ Para “competir” con lenguajes script como JS, Python

Lambdas: ¿son clases?

55

- ▶ No es necesario tener lambdas directamente en Java
- ▶ Pero sale muy “verboso” (ceremonioso)
- ▶ Las lambdas son más sucintas (si se usan bien)
- ▶ Adecuado para programación por eventos (reactiva)
- ▶ Uso especial: Listeners de GUI
- ▶ Concurrencia

Ejercicio

56

- ▶ Estudie y compare `FrameTest.java` y `FrameTestLambda.java`
- ▶ Compile en consola y ponga a funcionar
- ▶ Note que el código es más simple (menos líneas de código)
- ▶ Menos imperativo más declarativo

Nuevas formas de control

57

- ▶ Asincronía/concurrencia
- ▶ Promesas
- ▶ Programación reactiva (no hay un control principal)
- ▶ Eventos y el programa reacciona a los mismos
- ▶ Iteradores/Generadores (Python, ES6)

Patrones control

58

- ▶ `for` → `for each` → `.forEach`
- ▶ Decisión: ¿quién tiene el control?
- ▶ ¿El lenguaje ó los objetos?
- ▶ ¿Qué es más orientado a objetos?
- ▶ ¿Qué es más declarativo?

Ejercicio

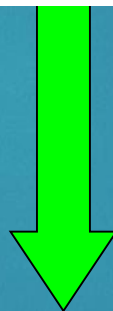
59

- ▶ Estudie y compile la clase IterationsPatterns.java

Conversión “bajo nivel” for-each

```
// 2) Estilo JDK 7
Naturals nats7 = new Naturals();
for(int n : nats7){
    if(n > 10) break;
    if(n%2==0)
        System.out.println(n);
}
```

Compilador
transforma



```
Naturals nats4 = new Naturals();
Iterator<Integer> iter = nats4.iterator();
while(iter.hasNext()){
    int n = iter.next();
    if(n > 10) break;
    if(n%2==0)
        System.out.println(n);
}
```


Poder Expresivo

61

- ▶ Requerimiento (establece un “qué”)
- ▶ *El máximo común divisor “ $\gcd(a, b)$ ” entre dos enteros positivos “ a ” y “ b ” es el mayor entero “ d ” que los divide a ambos.*
- ▶ No dice “cómo”- Se necesita definición más “operacional” (alias un “cómo”)
- ▶ Una definición recursiva de $\gcd(a, b)$ usando reglas:
 - ▶ $\gcd(a, a) = a$; $\gcd(0, b) = b$; $\gcd(a, 0) = a$
 - ▶ Si $a > b$ entonces $\gcd(a, b) = \gcd(a - b, b)$.
 - ▶ Si $a < b$ entonces $\gcd(a, b) = \gcd(a, b - a)$



```
// C plain
int gcd(int a, int b){
    if (a == 0) return b;
    if (b == 0) return a;
    while ( a != b ){
        if ( a > b ) a = a - b;
        else b = b - a;
    }
    return a;
}
```

```
// Go imperativo
func gcd(a int, b int) int {
    if a == 0 {
        return b
    }
    if b == 0 {
        return a
    }
    for a != b {
        if a > b {
            a = a - b
        } else {
            b = b - a
        }
    }
    return a
}
```

```
% Prolog reglas sabor imperativo
gcd(A, A, A) :- !.
gcd(0, B, B) :- !.
gcd(A, 0, A) :- !.
gcd(A, B, D) :- A > B, !,
                A1 is A - B, gcd(A1, B, D).
gcd(A, B, D) :- B1 is B - A,
                gcd(A, B1, D).
```

```
// Kotlin recursivo por reglas
fun gcd( a: Int, b: Int) : Int =
    when {
        a == b -> a
        a == 0 -> b
        b == 0 -> a
        a > b  -> gcd(a - b, b)
        else   -> gcd(a, b - a)
    }
```

Código x86 (MinGW)

63

```
00401460 <_gcd>:
401460: 55                push    %ebp
401461: 89 e5             mov     %esp,%ebp
401463: 83 7d 08 00       cmpl    $0x0,0x8(%ebp)
401467: 75 05             jne     40146e <_gcd+0xe>
401469: 8b 45 0c           mov     0xc(%ebp),%eax
40146c: eb 2c             jmp     40149a <_gcd+0x3a>
40146e: 83 7d 0c 00       cmpl    $0x0,0xc(%ebp)
401472: 75 1b             jne     40148f <_gcd+0x2f>
401474: 8b 45 08           mov     0x8(%ebp),%eax
401477: eb 21             jmp     40149a <_gcd+0x3a>
401479: 8b 45 08           mov     0x8(%ebp),%eax
40147c: 3b 45 0c           cmp     0xc(%ebp),%eax
40147f: 7e 08             jle     401489 <_gcd+0x29>
401481: 8b 45 0c           mov     0xc(%ebp),%eax
401484: 29 45 08           sub     %eax,0x8(%ebp)
401487: eb 06             jmp     40148f <_gcd+0x2f>
401489: 8b 45 08           mov     0x8(%ebp),%eax
40148c: 29 45 0c           sub     %eax,0xc(%ebp)
40148f: 8b 45 08           mov     0x8(%ebp),%eax
401492: 3b 45 0c           cmp     0xc(%ebp),%eax
401495: 75 e2             jne     401479 <_gcd+0x19>
401497: 8b 45 08           mov     0x8(%ebp),%eax
40149a: 5d                pop     %ebp
40149b: c3                ret
```

```
// C plain
int gcd(int a, int b){
    if (a == 0) return b;
    if (b == 0) return a;
    while ( a != b ){
        if ( a > b ) a = a - b;
        else b = b - a;
    }
    return a;
}
```

CS: Símbolo del sistema

```
JS: gcc gcd.c -o gcd.exe
JS: objdump -D gcd.exe > gcd.code
JS:
```

X86 Assembler

Código JVM (bytecode)

64

```
static int gcd(int, int);
```

Code:

0: iload_0	
1: ifne	6
4: iload_1	
5: ireturn	
6: iload_1	
7: ifne	12
10: iload_0	
11: ireturn	
12: iload_0	
13: iload_1	
14: if_icmpeq	36
17: iload_0	
18: iload_1	
19: if_icmple	29
22: iload_0	
23: iload_1	
24: isub	
25: istore_0	
26: goto	12
29: iload_1	
30: iload_0	
31: isub	
32: istore_1	
33: goto	12
36: iload_0	
37: ireturn	

a = #0 b=#1

```
static int gcd(int a, int b){
```

```
    if (a == 0) return b;
```

```
    if (b == 0) return a;
```

```
    while ( a != b ){
```

```
        if ( a > b ) a = a - b;
```

```
        else return b = b - a;
```

```
    }
```

```
    return a;
```

```
}
```

CA. Símbolo del sistema

```
OS:javac Gcd.java
```

```
OS:javap -c -l Gcd.class > Gcd.code
```

```
OS:
```

Ejercicio 8bits gcd

65

- ▶ Considere el simulador [8bit](#)
- ▶ Estudie el detalle del simulador [acá](#)
- ▶ Escriba un programa que calcule el gcd de dos números

Ejercicio Erlang gcd

66

- ▶ Estudie el case (pattern-matching) de Erlang
- ▶ Implemente gcd en gcd.erl (la versión del material tira una excepción)
- ▶ Esperado:

```
17/07/2017 14:02 <DIR> .
17/07/2017 14:02 <DIR> ..
17/07/2017 16:24      776 gcd.erl
                1 archivos      776 bytes
                2 dirs 20.963.966.976 bytes libres

PP:escript gcd.erl 20 24
gcd(20, 24)= 4

PP:escript gcd.erl 20
gcd(20, 0)= 20

PP:escript gcd.erl
gcd(0, 0)= 0
```


Historia Lenguaje (algunos)

67

- ▶ Fortran (54) (Primer compilado)
- ▶ Lisp (58) (Primer FP; IA)
- ▶ Cobol (59) (estandarizado)
- ▶ Simula (62) (origen OOP)
- ▶ Basic (64) (pre-PC)
- ▶ Logo (67)
- ▶ Algol(68) (prog. Estructurada, def. formal)
- Pascal (70)
- FORTH (70)
- Awk (70)
- Scheme (70)
- C (72)
- Smalltalk (72) (OOP)
- Prolog (72) (LP; IA)
- ML (73)
- Modula-2 (78)
- Ada (80) (DoD)
- SQL (86)

Historia (cont.)

68

- ▶ ADA (83) (diseño-por-contrato)
- ▶ C++ (83)
- ▶ Eiffel (85)
- ▶ Objective-C (86)
- ▶ Object-Pascal, Delphi (86, 95)
- ▶ Perl (87)
- ▶ Erlang(89)
- Haskell (90)
- VisualBasic (91)
- Python (91)
- Java (91)
- Lua (93) (Brasileño!)
- R (93) (Data mining)
- JavaScript (95)
- PHP (95)
- Ruby (95) (japonés)
- C# (2000) (El Java de MS)

Historia más reciente

69

- ▶ Scala (2003) (OOP y FP integrados en JVM). Mejor Java
- ▶ Groovy (2003) (Un Java-Python)
- ▶ F# (2005) (funcional corriendo en .Net)
- ▶ Clojure (2007) (“Scheme” en JVM)
- ▶ Node.js (2009) (JS en server)
- ▶ Go (2009) (google, concurrencia) ¿Algo entre C y Java?
- ▶ Dart (2011) (Un “mejor” JS)
- ▶ Typescript (MS, 2012, competidor de Dart). Pareja Angular
- ▶ Rust (2010) (¿Sustituto de C++?) Primer reléase mayo-2017!
- ▶ Kotlin (2011) Un mejor Scala (Android language)
- ▶ Elixir (2011) (Un Erlang renovado)
- ▶ Hack (2014) (Facebook, evolución de PHP)
- ▶ Swift (2014, Apple, MacOS, ios; reemplazo Objective-C)