UNIVERSITAT DE BARCELONA

Bases de dades avançades

curs 2017/2018

Enric Biosca Trias ebiosca@ub.edu

Dept. Matemàtica Aplicada i Anàlisi.

Universitat de Barcelona

Introducción a Scala

Bases de dades avançades curs 17/18

Enric Biosca Trias ebiosca@ub.edu

Dept. Matemàtica Aplicada i Anàlisi.

Universitat de Barcelona

- Scala
  - **scala?**
  - Hello world
  - Colecciones
  - Funciones
  - Clases, case clases y objetos

scala?

¿Qué es?

- Lenguaje de programación
- Compatible con Java

- Diseñado para expresar patrones de forma concisa

- Características principales
  - Orientado a objetos
  - **Programación Funcional Híbrida**
    - Full funcional
    - Mezclar con imperativa

scala?

Programación funcional

- Paradigma de programación
- Basado en el uso de funciones matemáticas

- Inmutabilidad
- Sin efectos colaterales

- Contrapunto a la programación imperativa
  - Python, java, C …

- Funciones típicas
  - Funciones anónimas (lambdas)
  - Filtros
  - Mapa
  - Reducir

scala?

Functional versus Object-Oriented Programming (ft. Martin Odersky)

scala?

Programación funcional: funciones típicas

```scala
val lista = List(1, 2, 3, 4, 5, 6)
// List(1, 2, 3, 4, 5, 6)
```

```scala
val pares = lista.filter(num => num%2 == 0)
// List(2, 4, 6)
```

```scala
val paresQuadrado = pares.map(x => x * x)
// List(4, 16, 36)
```

```scala
val suma = paresQuadrado.reduce( _ + _ )
// 56
```

```python
lista = [1, 2, 3, 4, 5, 6]
```

```python
pares = []
for num in lista:
    if num%2 == 0:
        pares.append(num)
```

```python
paresQuadrado = []
for x in pares:
    paresQuadrado.append(x * x)
```

```python
suma = 0
for x in paresQuadrado:
    suma += x
```

scala?

scala + Big Data: unión de éxito

Puntos destacados de la programación funcional:

- Inmutabilidad
- Sin efectos colaterales

Consecuencias:

- Fácilmente paralelizable
- No hay dependencias entre procesos
- Resultados determinísticos

¿Quién lo usa en Big Data?

- Frameworks
  - Apache Spark
  - Apache Flink
- Empresas
  - Twitter (todo el backend)
  - Coursera
  - Soundcloud

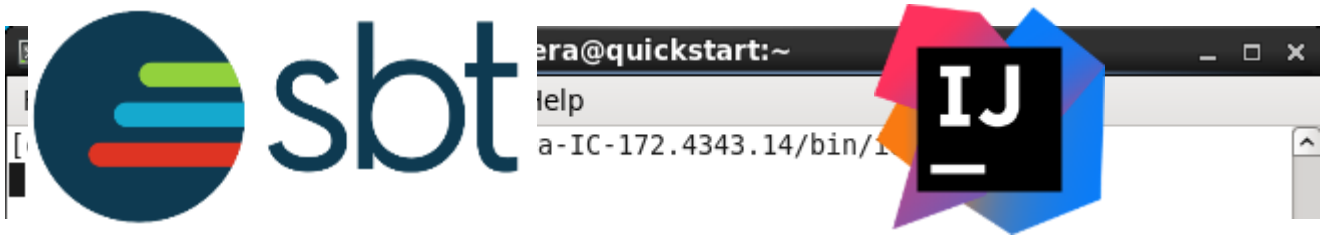
Scala

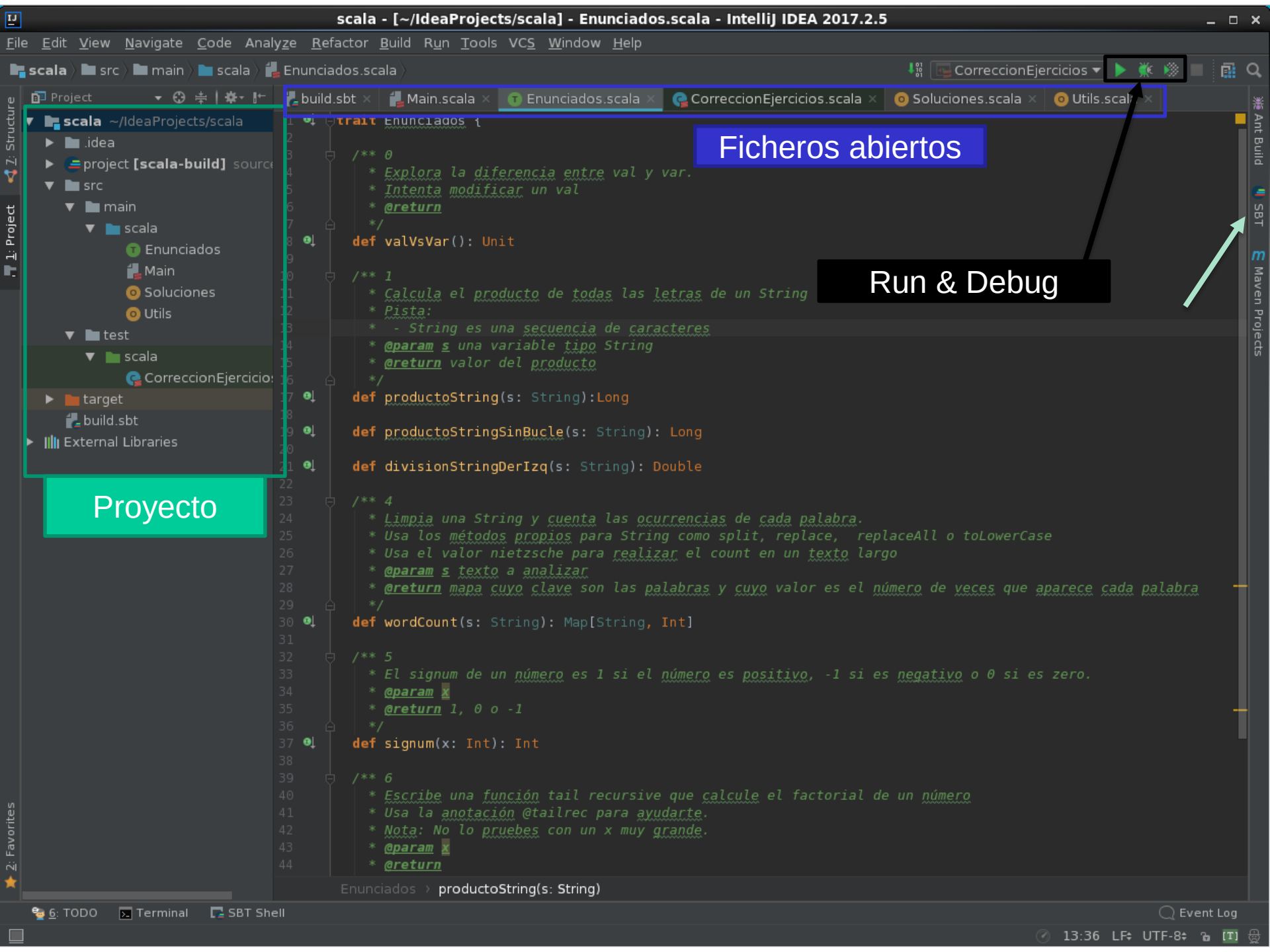
- scala?
- **Hello world**
- Colecciones
- Funciones
- Clases, case clases y objetos

Hello world

Entorno de trabajo

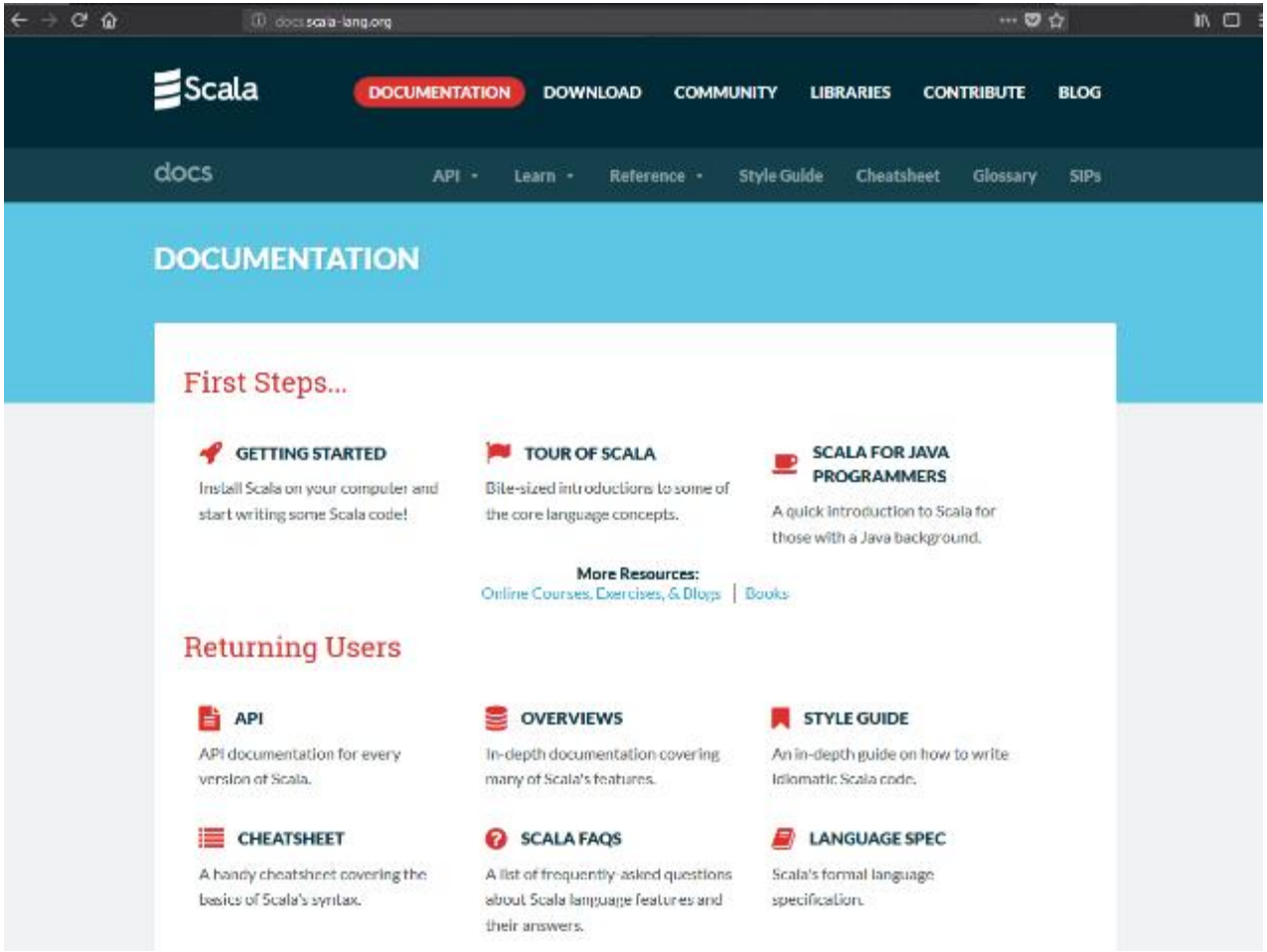
- SBT: *scala build tool*
  - Se encarga de descargar los paquetes y compilar
  - `$ sbt console` para obtener una consola de scala
- IntelliJ IDE
  - Detección de errores
  - Autocompletado




scala - [~/IdeaProjects/scala] - Enunciados.scala - IntelliJ IDEA 2017.2.5
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
scala src main scala Enunciados.scala
CorreccionEjercicios
Project
scala ~/IdeaProjects/scala
.idea
project [scala-build]
src
main
scala
Enunciados
Main
Soluciones
Utils
test
scala
CorreccionEjercicios
target
build.sbt
External Libraries
Proyecto
build.sbt
Main.scala
Enunciados.scala
CorreccionEjercicios.scala
Soluciones.scala
Utils.scala
Ficheros abiertos
Run & Debug
trait Enunciados {
/** 0
* Explora la diferencia entre val y var.
* Intenta modificar un val
* @return
*/
def valVsVar(): Unit
/** 1
* Calcula el producto de todas las letras de un String
* Pista:
* - String es una secuencia de caracteres
* @param s una variable tipo String
* @return valor del producto
*/
def productoString(s: String):Long
def productoStringSinBucle(s: String): Long
def divisionStringDerIzq(s: String): Double
/** 4
* Limpia una String y cuenta las ocurrencias de cada palabra.
* Usa los métodos propios para String como split, replace, replaceAll o toLowerCase
* Usa el valor nietzsche para realizar el count en un texto largo
* @param s texto a analizar
* @return mapa cuyo clave son las palabras y cuyo valor es el número de veces que aparece cada palabra
*/
def wordCount(s: String): Map[String, Int]
/** 5
* El signum de un número es 1 si el número es positivo, -1 si es negativo o 0 si es zero.
* @param x
* @return 1, 0 o -1
*/
def signum(x: Int): Int
/** 6
* Escribe una función tail recursive que calcule el factorial de un número
* Usa la anotación @tailrec para ayudarte.
* Nota: No lo pruebes con un x muy grande.
* @param x
* @return
Enunciados › productoString(s: String)
6: TODO
Terminal
SBT Shell
Event Log
13:36 LF UTF-8
Ant Build
SBT
Maven Projects
2: Favorites

Más sobre escala

Scala documentation web



Hello World

Antes de iniciar

Syntactic sugar: scala tiene formas de hacer el código más parecido al lenguaje humano.

```
a.map(square)
a map square
```

```
a :: b
b.::(a)
```

```
(x: Int) => x/2.0
x => x/2.0
_/2.0
```

```
a += 5
a = a + 5
```

```
val ls = List(1,2)
ls(0)
ls.apply(0)
```

```
val ar = Array(1,2)
ar(0) = 5
ar.update(0, 5)
```

```
ar.size
ar size
```

```
for (e <- ls) yield e*3
ls.map(_*3)
```

Hello world

Basic Types

Los tipos en scala **son clases** (al contrario que en java o Python).
No hay diferencia entre tipos primitivos y tipos creados en clases.
Esto permite invocar métodos sobre estos.

Byte	Int
Char	Long
Boolean	Float
Short	Double
java.lang.String	

Operadores: los operadores +, -, *… en realidad están definidos como métodos

```
scala> 1 + 1
scala> 1.+(1)
```

```
scala> 10.toString()
res0: String = 10

scala> "Hello".intersect("World")
res1: String = lo

scala> 1.to(10)
res2: …Range… = Range 1 to 10

scala> "31.2".toDouble
res3: Double = 31.2
```

str(10)

Hello world

Basics

scala no és solo un java que mola. Tiene que aprenderse con una mentalidad fresca.

Expresiones: casi todo es una

```
scala> 1 + 1
res0: Int = 2
```

Valores: Nombre de una expresión.

```
scala> val two = 1 + 1
two: Int = 2
```

Variables: si necesitas poder cambiar el valor, usa var

```
scala> var name = "steve"
name: java.lang.String = steve

scala> name = "marius"
name: java.lang.String = marius
```

Hello world

Basics

scala no requiere **;** al final de la línea, sólo se usa si definimos dos expresiones en la misma línea

Condicionales:

```
if (x>0)
if (x>=0) 1 else -1
```

Rangos:

```
1 to 5  //1, 2, 3, 4, 5
1 until 5  //1, 2, 3, 4
```

Loops: existen `while` y `do` como en java y C++. La sintaxis de `for` es distinta: es una iteración sobre una secuencia (lista, array, buffer …) . Comparable al `for` de python

no existe *break*

```
while (n > 0) {
  r = r * n
  n -= 1
}
```

```
for (i <- 1 to 10 if i%2 == 0)
  yield i
```

En scala no se usan mucho los *loops*. Muchas veces es mejor un `map` y/o un `reduce`

Hello world

Basics

Funciones: las funciones son *ciudadanos de primer nivel*, como clases y variables

```
scala> def sumaUno(x: Int): Int = x + 1
sumaUno: (x: Int)Int

scala> val three = sumaUno(2)
three: Int = 3
```

Funciones anónimas (*lambdas*): no siempre tenemos porque asignar un nombre a todas las funciones. Útil para colocarlas dentro de otras funciones (map, reduce, aggregate …). Hay *distintas sintaxis equivalentes*.

```
scala> (x: Int) => x + 1
res2: (Int) => Int = <function1>

scala> List(1, 2, 3).map( x => x + 1 )
res3: List[Int] =  List(1, 2, 3)

scala> List(1, 2, 3).map( _ + 1 )
```

_ es una *wildcard*, hace match con cualquier cosa

Hello world

Basics

Bloques de expresiones : para escribir un bloque de código que ocupa más de una línea, siempre se puede usar **{ }** para indicarlo. Su uso no se limita sólo a las clases y métodos, se puede ampliar a cualquier cosa. El valor del bloque será el valor de la última expresión

```
val distancia = {
  val dx = x - x0
  val dy = y - y0
  sqrt(dx * dx + dy * dy)
}

def sumaUno(x: Int): Int = {
  x + 1
}
```

```
def sumaUno(x: Int): Int = {
  x + 1
}

if (x > 0) {
  1
} else if (x == 0) {
  0
} else {
  -1
}
```

Como buena práctica, deja siempre una **línea en blanco después de cada método o bloque**
de código grande. Ayudará a la lectura y comprensión.

Scala

- scala?
- Hello world
- **Colecciones**
- Funciones
- Clases, case clases y objetos

Hello world

Collections

Arrays: conservan el orden, pueden contener duplicados y pueden mutar

```
scala> val numbers = Array(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
numbers: Array[Int] = Array(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)

scala> numbers(3) = 10
```

Listas: conservan el orden, pueden contener duplicados y son inmutables

```
scala> val numbers = List(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
numbers: List[Int] = List(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
scala> numbers.head // gives 1
scala> numbers.tail // gives List(2, 3, 4, 5, 1, 2, 3, 4, 5))
```

Hay más formas de trabajar con listas que se usan mucho en scala. Muchas cosas ya están implementadas. Mirad en scalaDocs sus métodos.

Sets: NO conservan el orden, NO contienen duplicados

```
scala> val numbers = Set(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
numbers: scala.collection.immutable.Set[Int] = Set(5, 1, 2, 3, 4)
```

Hello world

Collections

Tuples: grupos de elementos. Accede a los elementos mediante `._1`, `._2`, `._3` …

```
scala> val hostPort = ("localhost", 80)
hostPort: (String, Int) = (localhost, 80)
scala> hostPort._2
res1: Int = 80
```

Las tuplas son útiles, pero es mejor usar *case classes*

Mapas:

```
scala> val mapa = Map("one" -> 1, "two" -> 2)
mapa: scala.collection.immutable.Map[Int,String] = Map(one -> 1, two -> 2)
```

Option: es un contenedor que puede o no contener algo

```
scala> mapa.get("two")
res0: Option[Int] = Some(2)

scala> mapa.get("three")
res1: Option[Int] = None
```

Hello world

Collections – métodos comunes

Estas serán vuestras amigas y enemigas. Son métodos típicos en la programación funcional, y os los encontraréis en Spark. Se aplican a **iterables** (Array, List, Set, Map* …)

map: evalúa una función sobre cada elemento, devolviendo una nueva secuencia.

```
scala> val numbers = List(1, 2, 3, 4)

scala> numbers.map((i: Int) => i * 2)
res0: List[Int] = List(2, 4, 6, 8)
```

También podéis pasar una función

```
scala> def timesTwo(i: Int): Int = i * 2
timesTwo: (i: Int)Int

scala> numbers.map(timesTwo)
res0: List[Int] = List(2, 4, 6, 8)
```

Se expande a
`.map(timesTwo(_))`
que es equivalente a
`.map(i => timesTwo(i))`

¿Recordáis las funciones anónimas (alias lambdas)? ¿Y sus distintas sintaxis?

Hello world

Collections – métodos comunes

`foreach`: como `.map` pero no devuelve nada (clase Unit, como void en java). Útil para *side-effects*

```
scala> val nada = numbers.foreach(println)
nada: Unit = ()
```

`filter`: elimina los elementos dónde la función evalúa a falso

```
scala> def isEven(i: Int): Boolean = i % 2 == 0
isEven: (i: Int)Boolean
scala> numbers.filter(isEven)
res2: List[Int] = List(2, 4)
```

`zip`: agrega los contenidos de dos listas en una lista de pares (como una cremallera)

```
scala> List(1, 2, 3).zip(List("a", "b", "c"))
res0: List[(Int, String)] = List((1,a), (2,b), (3,c))
```

Hello world

Collections – reducciones

`reduce:` aplica una operación binaria (entre dos elementos) a toda la secuencia. Esta operación debe resultar en el mismo tipo que los elementos originales. *Ej: Int + Int => Int*

```
numbers.reduce(_ + _)
```

```
def sumar(x: Int, y: Int) = x + y
numbers.reduce(sumar)
```

Hay más formas de agregar y reducir los contenidos de un `Iterable`. En este curso sólo usaremos reduce, la más común y simple de todas. Las posibilidades las encontraréis en el scala docs, y esencialmente son los siguientes métodos: `reduceLeft`, `reduceRight`, `aggregate`, `foldLeft` y `FoldRight`

Scala

- scala?
- Hello world
- Colecciones
- **Funciones**
- Clases, case clases y objetos

Funciones

Recordemos

Las funciones son en realidad objetos.
Pueden pasarse como parámetros

Funciones: las funciones son ***ciudadanos de primer nivel***, como clases y variables

```
scala> def sumaUno(x: Int): Int = x + 1
sumaUno: (x: Int)Int

scala> val three = sumaUno(2)
three: Int = 3
```

Funciones anónimas (*lambdas*): no siempre tenemos porque asignar un nombre a todas las funciones. Útil para colocarlas dentro de otras funciones (map, reduce, aggregate …). Hay *distintas sintaxis equivalentes*.

```
scala> (x: Int) => x + 1
res2: (Int) => Int = <function1>

scala> List(1, 2, 3).map( x => x + 1 )
res3: List[Int] =  List(1, 2, 3)

scala> List(1, 2, 3).map( _ + 1 )
```

⚠ _ es una *wildcard*, hace match con cualquier cosa

Funciones

Parámetros

- De 0 a 22 parámetros
- Se debe indicar los tipos de los parámetros

```
def decora(s: String, left: String, right: String) = left + s + right
```

- Puedes definir valores por defecto en los parámetros

```
def decora(s: String, left: String = "[" , right: String = "]") = left + s + right
```

- Puedes cambiar el orden de los parámetros cuando llamas a la función si los nombras

```
scala> decora("hola", right= "<<")
res0: String = [hola<<

scala> decora(right= "-|", left= "|-", s="hola")
res1: String = |-hola-|
```

- Puedes definir funciones que toman un número variable de argumentos

```
def sum(args: Int*): Int =
  if (args.length==0) 0
  else args.head + sum(args.tail : _*)
sum(1,2,3,4,5,2,123,23,7,344,66,23)
```

Indica que es una secuencia de Ints

Transforma a secuencia de parámetros

las funciones en scala están implementadas como objetos

tail recursion

- A veces, la recursividad puede dar lugar a un bucle que conserva variables.
- Es muy usada en programación funcional
- Debes indicar el Tipo que devuelve

- Las funciones ***tail recursive*** **están optimizadas** en Scala. En ellas, lo **último que se hace es llamarse a si misma**. Usad la anotación @tailrec para que el IDE os ayude

```
def sum(xs: Seq[Int]): BigInt = {
   if (xs.isEmpty) 0
   else xs.head + sum(xs.tail)
 }
```

Tipo devuelto

Suma y llama a la función

```
 @tailrec
  def sumTailRec(xs: Seq[Int],
acumulador: BigInt): BigInt = {
    if (xs.isEmpty) acumulador
    else sumTailRec(xs.tail, xs.head +
acumulador)
  }
```

Sólo llama a la función

Funciones

Todo dentro de todo

Las funciones pueden estar dentro una clase, un objeto o incluso dentro de otras funciones.
Especialmente útil para implementar funciones recursivas, dónde tienes que iniciar el acumulador.

```
def sum(xs: Seq[Int]): BigInt = {

  @tailrec
  def sumTailRec(xs: List[Int], acumulador: BigInt): BigInt = {
    if (xs.isEmpty) acumulador
    else sumTailRec(xs.tail, xs.head + acumulador)
  }

  sumTailRec(xs, 0)
}
```

Índice

Scala

- scala?
- Hello world
- Colecciones
- Funciones
- **Clases, case clases y objetos**

Clases, case clases y objetos

Clases

```
class Counter {
  private var value = 0

  def this(startFrom: Int) = {
    this()
    value = startFrom
  }

  def increment() = value +=1
  def current = value
}
```

Constructor primario (código fuera de los métodos)

Constructor auxiliar

Métodos

- Se generan automáticamente los *getters* y *setters* para cada campo (con el mismo nombre que el campo)
- Puedes marcar campos como privados y privados para una clase/paquete en concreto
- Pueden heredar sólo de una clase

Clases, case clases y objetos

Objetos

Usa un `object` cuando necesitas una clase con una sola instancia o como lugar para guardar valores o funciones varias

Singleton: lugar para guardar los métodos estáticos

- Guarda funciones y constantes
- Cuando una instancia inmutable puede compartirse de modo eficiente
- Cuando necesitas coordinar algún servicio usando una sola instancia

```
object Accounts {
  private var lastNumber = 0
  def newUniqueNumber() = {
    lastNumber +=1
    lastNumber
  }
}
```

Se puede acceder desde cualquier sitio

Companion: cuando necesitas una clase con métodos propios de la instancia y otros estáticos

```
class Account {
  val id =
     Account.newUniqueNumber()
  private var balance = 0.0
  def deposit(x: Double) {...}
  ...
}

object Account {
  private var lastNumber = 0
  private def newUniqueNumber() =
    {...}
}
```

Sólo instancias de la clase Account pueden acceder

Clases, case clases y objetos

Application Objects

Todo programa de Scala debe empezar con un método `main` dentro de un objeto. O se puede extender el trait `App`.

```
object Hello {
  def main(args: Array[String]) {
    println("Hello World!")
  }
}
```

```
object Hello extends App {
  println("Hello World!")
}
```

Todo programa tiene que tener al menos una clase de estas!

Clases, case clases y objetos

Traits

Sólo se puede heredar de una clase pero de multiples traits

Un `trait` es una colección de campos y métodos que puedes extender o usar en tus clases
Una **clase puede extender múltiples traits**

```
trait Car {
  val brand: String
}

trait Shiny {
  val shineRefraction: Int
}

class BMW extends Car with Shiny {
  val brand = "BMW"
  val shineRefraction = 12
}
```

Añade múltiples traits con la keyword `with`

Un `trait` es **similar a las interfaces de Java**, pero también puede tener implementaciones de métodos y campos.

La única diferencia técnica entre traits y clases es que no puede tener parámetros en el constructor.

```
trait Car(brand: String) extends
Vehicle {...}
```

Clases, case clases y objetos

Case classes

Clases sin métodos, sólo tienen propiedades, similar a JavaBeans.

```
case class Phone (numero: Int, extension: Int)

case class Persona(dni: String, nombre: String, apellidos: String, telf: Phone)

val jane = Persona("0001A","Jane", "Doe", Phone(666666666, 34))

jane.dni
jane.telf.numero
```

Será extremadamente útil en Spark y SparkSQL. Si no lo entendéis, preguntádmelo

Clases, case clases y objetos

Pattern matching

Avanzado

Tiene múltiples aplicaciones:

- Estamentos *switch*
- Determinación de Tipo (en lugar de usar `isInstanceOf`)

Si no hace match con nada lanzará error. Usa la *wildcard* _ para evitarlo

```
obj match {
  case x: Int => x
  case s: String => Integer.parseInt(s)
  case _: BigInt => Int.MaxValue
  case _ => 0
}
```

```
map.get("key") match {
  case Some(v) => v
  case None => 0
}
```

- Desestructuración (de expresiones complejas)

```
case class PhoneExt(name: String, ext: Int)
val extensions = List(PhoneExt("steve", 100), PhoneExt("robey", 200))
scala> extensions.filter { case PhoneExt(name, extension) => extension < 200 }
res0: List[PhoneExt] = List(PhoneExt(steve,100))
```

También se puede usar para simplificar funciones! Ya sea con *case classes, nested classes, tuples* …

Clases, case clases y objetos

Pattern matching

Avanzado

Matching con valores

```
val times = 1

times match {
  case 1 => "one"
  case 2 => "two"
  case _ => "some other number"
}
```

Matching con *guards*

```
val times = 1

times match {
  case i if i == 1 => "one"
  case i if i == 2 => "two"
  case _ => "some other number"
}
```

Matching con miembros de una clase

```
case class Calculator(brand: String, model: String)

def calcType(calc: Calculator) = calc match {
  case Calculator("HP", "20B") => "financial"
  case Calculator("HP", "48G") => "scientific"
  case Calculator("HP", "30B") => "business"
  case Calculator(ourBrand, ourModel) => "Calculator: %s %s is of unknown
                                         type".format(ourBrand, ourModel)
}
```

Multiple alternatives:

```
case _ => ...
case Calculator(_, _) => ...
```