Scala

14 September 2023 11:29

Issues with Hadoop

- Issue with small files
  - Doesn't suit small data files, waste of space (high capacity design)
  - Small file (<128 mB) which is smaller than HDFS block
  - Too many small files cause problems for managing namenode
- Slow processing speed
  - Data distributed and processed over cluster in MapReduce which reduces processing speed
- Supports batch processing only, not streaming data
- No Delta Iteration
  - Not efficient for iterative processing
  - Does not support cyclic data flow

Apache Spark vs Apache Hadoop

Both are open source frameworks for big data processing

Apache Spark	Apache Hadoop
• Resilient distributed dataset RDD • No distributed file system, so mainly used for computation on top of Hadoop • Runs on local filesystem • Scalable file system • Does not need hadoop to run, but can be used to create distributed datasets	• Map Reduce • Distributed file system

Apache Spark vs Scala

Apache Spark	Scala
• Open source, distributed, general purpose framework for cluster computation • To increase efficiency of computation of Hadoop	• General purpose programming language providing support for functional programming and string static type system • Used for web applications, streaming data, distributed applications and parallel processing

What is Scala?

- Scalable language
- Multi paradigm programming language
- Integrates features of OOP and functional languages

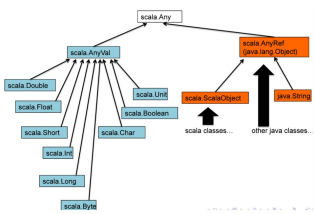
1. It is object oriented
   a. Every value is an object
   b. Types and behaviours described by classes and traits
   c. Subclassing, flexible mixin-based composition mechanism, (clean replacement for multiple inheritance)
2. It is functional
   a. Every function treated as value
   b. Lightweight syntax for defining anonymous functions
   c. Higher order functions
   d. Nesting
   e. Supports currying (functions with multiple arguments transformed to functions with single argument)
3. Everything immutable by default (can be made mutable)
4. Do not have to mention type of function output
5. Case classes, pattern matching provide functionality of algebraic types (used in many functional language)
6. Singleton objects: Provides convenient way of grouping functions that aren't members of a class
7. Ideal for building web services and data intensive applications

Scala is statically typed
It supports:

- Generic classes
- Variance annotations
- Upper and lower type bounds
- Inner classes and abstract type members as object members
- Compound types
- Explicitly typed self references
- Implicit parameters and conversions
- Polymorphic methods

Type inference means the user is not required to annotate code with redundant type information.
The Scala compiler can often infer the type of an expression so you don't have to declare it explicitly.

Scala class Hierarchy



Java vs Scala

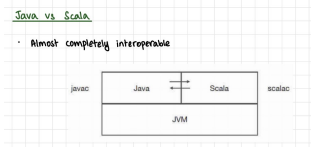
Compiled to byte codes and used JVM(Java virtual machine) to execute code

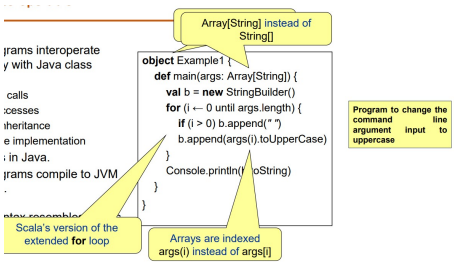
Scala	Java
• Lazy evaluation (can be specified by keyword lazy) • Supports operator overloading • Treats functions and methods as variables	• Does not support operator overloading • Treats functions as objects

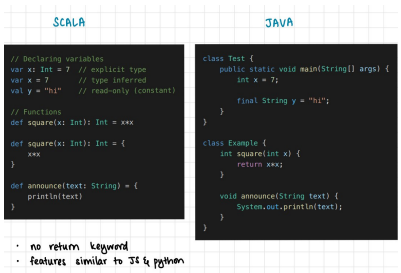
• Reduces number of lines of a java application by treating everything as an object and function passing • More complicated syntax • Limited backward compatibility	• Treats function as object • Has more lines of code • Less complicated syntax • Backward compatible

1. Scala is interoperable
   a. Scala programs interoperate seamlessly with java class libraries
      i. Method calls
      ii. Field accesses
      iii. Class inheritance
      iv. Interface implementation

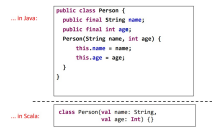
   All work as in java
   a. Scala programs compile to JVM bytecodes
   b. Syntax resembles java syntax (but there are some differences)

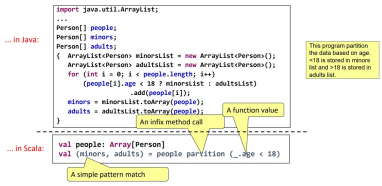




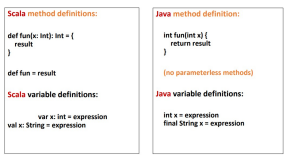


Classes



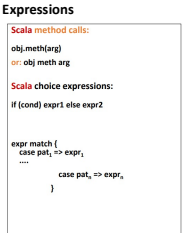


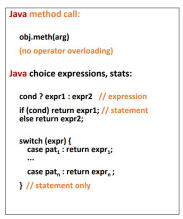
Method definitions



Val makes a variable immutable — like final in Java — and var makes a variable mutable.

Expressions





Traits

```
trait T {
  def abstractMeth(x: String): String

  def concreteMeth(x: String) =
    x+field

  var field = "!"
}
```

Scala mixin composition:

```
class C extends Super with T
```

```
interface T {
  String abstractMeth(String x)

  (no concrete methods)

  (no fields)
}
```

Java extension + implementation:

```
class C extends Super implements T
```

- Traits extend functionality of class using a set of methods

1. Minimal verbosity
2. Referential Transparency
   - type inferencing in Scala
   - compiler checks type of subexpressions, atomic values
3. Concurrency
   - Actor model
   - Akka — open-source framework for Actor-based concurrency
4. Functional Programming
   - Higher order functions that can return another function
   - Nested functions

1. Minimal Verbosity

- Getters & setters (note: ppt link does not work; objectmentor has been moved to http://cleancoder.com/)
- Java

```
class Person {
    private String firstName;
    private String lastName;
    private int age;

    public Person(String firstName, String lastName, int age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    public void setFirstName(String firstName) { this.firstName = firstName; }
    public String getFirstName() { return this.firstName; }

    public void setLastName(String lastName) { this.lastName = lastName; }
    public String getLastName() { return this.lastName; }

    public void setAge(int age) { this.age = age; }
    public int getAge() { return this.age; }
}
```

Scala (Automatic)

```
class Person(var firstName: String, var lastName: String, var age: Int)
```

2. Type Inferencing

- Java is statically typed
  - type errors caught by compiler
- Ruby & Python do not require declared types
  - harder to debug
  - not type safe
- Scala is statically typed but it uses type inferencing
  - type errors caught by compiler
  - https://docs.scala-lang.org/tour/type-inference.html

```
val collegeName = "PES University"      // const reference
def squareOf(x: Int) : x * x            // def method that cannot be reassigned
```

Consistency

- Java: every value is a type, except primitive types (int, bool) for efficiency reasons
- Scala: every value is an object; compiler turns into primitives for efficiency
- Java has operators & methods with different syntaxes
- In Scala, operators are methods and either syntax can be used

Concurrency

- Broadly speaking, concurrency can be either:
  - Fine-grained: Frequent interactions between threads working closely together (extremely challenging to get right)
  - Coarse-grained: Infrequent interactions between largely independent sequential processes (much easier to get right)
- Java 5 and 6 provide reasonable support for traditional fine-grained concurrency
  - Threads
- Scala has total access to the Java API
  - Hence, it can do anything Java can do
  - And it can do much more (see next slide)
- Scala also has Actors for coarse-grained concurrency
  - Sending messages (use the send ! Abstraction)

4. Functional Programming

- Problem with concurrency: acquire locks
- If prog language does not allow modification of variables, locks not required
- Functional programming languages use only immutable data (eg: ML, OCaml, Haskell, lisp)
- Difficult to learn
- Scala is an impure functional language — can program functionally but not forced upon you

- Features
  - Immutable
    - functional operations create new structures and do not modify existing structures

(b) Program implicitly captures data flow

(c) Order of operations unimportant

(d) Functions
- are objects
- arguments
- can be returned
- can operate on collections

Functional Programming – differences with java

Quicksort program in Scala – java style

Observe

Focus on HOW?

Explicitly iterate

Determine when and how to swap()

```
def sort(xs: Array[Int]) {
  def swap(i: Int, j: Int) {
    val t = xs(i); xs(i) = xs(j); xs(j) = t
  }
  def sort1(l: Int, r: Int) {
    val pivot = xs((l + r) / 2)
    var i = l; var j = r
    while (i <= j) {
      while (xs(i) < pivot) i += 1
      while (xs(j) > pivot) j -= 1
      if (i <= j) {
        swap(i, j)
        i += 1
        j -= 1
      }
    }
    if (l < j) sort1(l, j)
    if (j < r) sort1(i, r)
  }
  sort1(0, xs.length - 1)
```

Focus on solving the problem

Observe

Focus on WHAT, not HOW

Pick a pivot

Sort values smaller than the pivot

Sort values larger than the pivot

Concatenate the result

```
def sort(xs: Array[Int]): Array[Int] = {
  if (xs.length <= 1) xs
  else {
    val pivot = xs(xs.length / 2)
    Array.concat(
      sort(xs filter (pivot >)),
           xs filter (pivot ==),
      sort(xs filter (pivot <)))
  }
}
```

Original List is left unchanged

```
val list = List(1, 2, 3)
list.foreach(x => println(x)) // prints 1, 2, 3
list.foreach(println)         // same

list.map(x => x + 2)    // returns a new List(3, 4, 5)
list.map(_ + 2)         // same

list.filter(x => x % 2 == 1)// returns a new List(1, 3)
list.filter(_ % 2 == 1)     // same

list.reduce((x, y) => x + y) // => 6
list.reduce(_ + _)           // same
```

- Big data architectures leverage

  Parallel disk, memory and CPU in clusters
- Operations consist of independently parallel operations

  Similar to *map()* operator in a functional language
- Parallel operations have to be consolidated

  Similar to *aggregation()* operators in functional languages
- In order to achieve the desired efficiency for big data applications, Concurrency, Parallelism and Referential Transparency are key, and in a functional environment, all of them are naturally present.