Aplikacje „webowe” z wykorzystaniem

Scala i Lift

Opracował: Mikołaj Sochacki

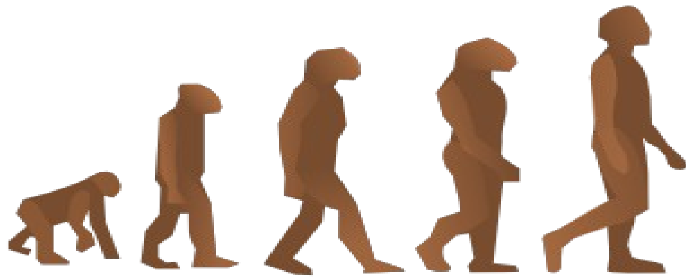
Jaki język będzie dla mnie najlepszy?

Dynamiczny

+ <u>szybkość pisania</u>
+ krótki czas potrzebny na opanowanie
- więcej pisania testów
- więcej trudnych do wykrycia błędów
- trudniej znaleźć typ parametru

Statyczny

+ <u>dużo błędów wyłapywanych na etapie kompilacji</u>
+ mniej pisania testów
- czas oczekiwania na kompilację
- <u>wolniejsze tworzenie kodu</u>



Ewolucja

C++ → fajny język, ale brak wygodnego frameworka do tworzenia stron internetowych (Wt?)

Perl → słaba czytelność kodu

C# → zamknięta technologia (.NET), słabe Mono

Python → co przyjmuje parametr funkcji?
brak znaku końca zakresu, mała szybkość działania programu

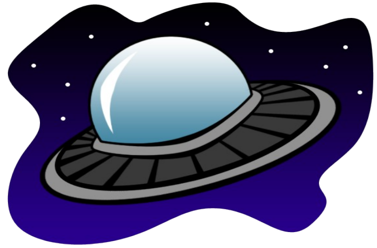
Scala

Dlaczego Scala?

- w pełni obiektowy
- imperatywno – funkcyjny
- ładna składnia – inferencja typów
- przeładowanie operatorów - jakich operatorów? ;)
- bezpośredni dostęp do bibliotek i technologii Java bez użycia ubogiego języka Java
- możliwość pisania dla .NET
- interpreter „pythono-podobny”
- inne: DSL, XML, skalowalność

Martin Odersky

- główny projektant Scali
- profesor EPFL w Lozanie - obecnie
- główny projektant generics w Java
- inżynier JVM i kompilatora w Sun Microsystems
- twórca języka Pizza – pierwowzoru Scala



Obiekty

- Wszystko (oprócz metod) jest obiektem:

```
3.toString
```

- Metody mogą mieć prawie dowolne nazwy, nie ma pojęcia operatora:

```
(1).+(2) // w skrócie 1 + 2
```

- Nie ma pól statycznych w zamian wykorzystuje się wzorzec Singleton – każda klasa może mieć obiekt o tej samej nazwie będący singletonem:

```
object MyClass {
   def witaj = "Witaj Scala”
}
MyClass.witaj
```

Obiekty - c.d.

→ Funkcje anonimowe jako obiekty:

```
val myFunct = (x:Int,y:Int) => x*y
myFunct(34,64)
```

→ Dziedziczenie jednobazowe, ale można użyć traits:

```
class MyClass extends Parent with MyTrait
```

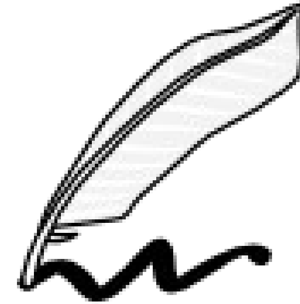
→ Przy nadpisaniu metod w dziecku obowiązkowe override

```
override def toString = "Nic"
```

→ Generics - wygodna składnia, możliwość definiowania własnych szablonów:

```
case class myClass[A](var in:A)
```

Funkcyjnie czy imperatywnie?



- Funkcyjnie:

```
(1 to 10).foreach(println)
```

- Imperatywnie:

```
for (i <- 1 to 10) println(i)
```

- Zazwyczaj i tak uzyskujemy ten sam bytecode
- Używanie niezmiennych referencji (immutable):

```
val i = 255 //niezmienna referencja
var j = 255 //zmienna referencja
```

- Ważna rola wbudowanych kolekcji

```
val list = List('h','e','l','l','o')
val list2 = list.map(x => x.toByte)
```

Biblioteki Java

→ Pełen dostęp do bibliotek napisanych w Java:

```
import java.io._
val in = new BufferedReader(
            new FileReader("plik.txt"))
val s = in.read()
```

→ Korzystanie z kodu Scali w Javie tylko pod warunkiem nie używania specyficznych własności języka:

```
def ++(list:List[String]):List[String] = {
     ... }  //nie da rady :)
```

→ Specyficzne nazwy klas i funkcji w kodzie Java:

```
object App == public static final class App$
```

„Syntactic sugar"

- lazy valuses:

```
lazy val x = {println("Inicjacja”);”Napis”}
```

- tuples: `val pair = (19,”Napis”)`
- funkcje anonimowe: `(s:String) => s.split(",")`
- wzorce: match, try - catch:

```
val s = x match {
        case 12 => "Jest dwanaście"
        case x:Int => "Jest " + x.toString }
```

- domknięcia
- partially applied function
- tail recursion
- parametry domyślne
- i wiele innych ...

Praca z XML

- węzeł jako wartość:

```
val link = "http://brosbit4U.net”
def toXML = <a href={link}>kliknij link</a>
```

- prosta serializacja (j.w.)
- wyciąganie informacji z XML:

```
val xml = <wezel atr="wartosc"> <wewn>Text
          </wewn> </wezel>
val str = xml \ @atr  //daje "wartosc"
val w = xml \ "wewn" //daje <wewn>Text</wewn>
val t = w.text //daje "Text"
<a><b><c>A</c><b></a> \\ "c"  //daje <c>A</c>
```

- prosta deserializacja (j.w.)

Actors - wątki

→ wykorzystanie modelu aktorów z Erlanga

```
val echoAktor = actor {
   loop {
      react {
         case msg => println("Otrzymałem: " +
                                   msg)
      }
   }
}
echoAktor ! "Witaj!"
```

→ brak współdzielenia danych - wysyłamy tylko stałe - nie ma zakleszczeń

→ zawsze można korzystać bezpośrednio z wątków Java :)

Scala frameworki webowe

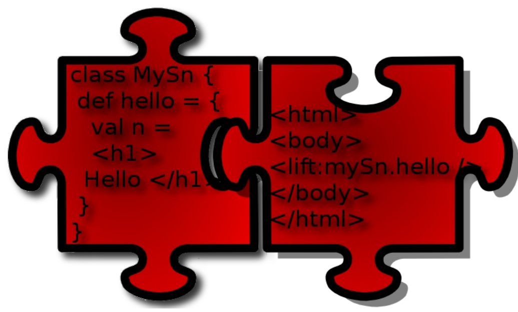
- Lift – wzorowany na Wicket, Django, Rails, Seaside
- SweetScala – wzorowany nieco na GAE i Django
- Scalatra - wzorowany na Sinatra
- Pinky – wykorzystuje Guice Servlet
- wszystkie frameworki Javy

David Pollack

- programuje zawodowo od 1977 roku
- od 1996 roku zajmuje się prawie wyłącznie programowaniem webowym
- przez wiele lat używał Ruby on Rails
- fan i propagator języka Scala
- w 2007 roku rozpoczął pracę nad Liftem (nazwa robocza Scala with Sails)

Manifest Lifta

- prosty i szybki CRUD oraz użycie MVC
- ORM bez konieczności użycia SQL
- wbudowany AJAX i Comet
- komponenty
- utrzymywanie komponentów w pamięci
- bezpieczeństwo
- wielowątkowość
- zintegrowany framework do testów
- łatwe wdrożenie i uaktualnienie aplikacji



Szablony i snippety

→ szablon:

```
<lift:surround with="default" at="content">
<lift:nazwaSnippetu.metoda form=”POST”>
<h1>kod xhtml</h1>
<znacznik:znacznik2 /> </lift:surround>
```

→ snippet:

```
class NazwaSnippetu {
  def metoda(node:NodeSeq):NodeSeq = {
      var str = ""
      bind( node, "znacznik”,
      Shtml.text(str = _),atrbut -> "atr" ) }
```

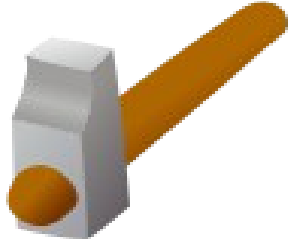
→ cechy: wielokrotne przetwarzanie XML, kod AJAX w bind, views, embed, tagi i snippety Comet, stacjonarne snippety

ORM

- mapper – dla baz SQL
- record – dla baz NoSQL (docelowo podstawowy model)
- dodatkowy kod przetwarzania w obiektach modelu
- konfiguracja bazy w klasie konfiguracyjnej aplikacji
- ProtoUser trait – obsługa użytkownika
- operacje na bazie bez SQL
- dodatkowe typy danych jak adres e-mail itp.
- równoczesna obsługa wielu baz danych

Technologie

- elementy formatek wbudowane w obiekty frameworka
- LiftActor – obsługa wątków (głównie Comet)
- użycie ścisłego XHTML i możliwość zmiany na HTML5
- bogaty system zarządzania URL
- zarządzanie sesją
- hierarchiczny system uwierzytelniania
- skrypty JS pisane klasami Lifta, obsługa JSON
- proste i szybkie budowanie aplikacji z Comet
- możliwość integracji z JPA
- integracja usługami: PayPal, XMPP, Facebook, AMQP, OpenID
- widgety



Wdrożenie

- edytory: Eclipse, NetBeans, IntelliJ IDEA,
  inne (tylko podświetlanie składni)
- kompilacja: Maven, Ant, SBT, Gradle
  (dla Mavena polecenie deploy)
- testowanie: konsola (mvn, sbt), scalaTest, S.notice
- serwer: Tomcat, Jetty, Glassfish, Jboss
  (najczęściej stosowany Tomcat + Nginx)
- hosting: Amazon.com (stax.net), GAE (tylko JPA),
  każdy obsługujący wymienione serwery,
  „dedyk”, własny serwer :)

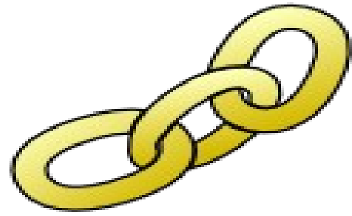
Σ Podsumowanie

ZALETY:

- wygoda i szybkość pisania i duże możliwości (+Scala)
- szybkość i wydajność aplikacji (nie ma konieczności łączenia z C/C++)
- bogactwo narzędzi i bibliotek Java
- bezpieczeństwo aplikacji – zastosowanie dla instytucji finansowych

WADY:

- mniejsza ilość źródeł wiedzy (młody framework)
- słaby hosting w Polsce
- problemy z narzędziami (np wtyczka Eclipse – choroby wieku dziecięcego :))



Linki

Oficjalna strona Scali:

http://www.scala-lang.org/

Oficjalna strona Lift:

http://www.liftweb.net/

Wiki:

http://scalatutorial.com/

http://scala.sygneca.com/

http://www.simplyscala.com/ - interpreter

http://www.assembla.com/wiki/show/liftweb/

Dziękuję za uwagę!

def **Pytania_?**(s:String):String