“If I were to pick a language to use today other than Java, it would be Scala.”
– James Gosling, creator of Java

“No other language on the JVM seems as capable of being a “replacement for Java” as Scala, and the momentum behind Scala is now unquestionable.”
– Charles Nutter, creator of JRuby
Success Stories

- Twitter maintains several millions lines of Scala (35% of its code base)
  - Travis Brown, “Scala at Scale at Twitter” (OSCON 2015)

- Scala drives its core social graph service: 380-400 M transactions per day
  - Martin Odersky, “From Scala to Typesafe” (Venture Ideas Symposium 2011)

- EU’s largest energy firm migrated a 300K lines contract modeling platform from Java to Scala
  - Lee Momtahan, “Scala at EDF Trading” (CUFP 2009)
The work on Scala was motivated by two hypotheses:

**Hypothesis 1:**
- A general-purpose language needs to be scalable; the same concepts should describe small as well as large parts.

**Hypothesis 2:**
- Scalability can be achieved by unifying and generalizing functional and object-oriented programming concepts.
Unify FP & OO

Both have complementary strengths for composition:

**Functional Programming**

Makes it easy to build interesting things from simple parts, using

- higher-order functions,
- algebraic types and pattern matching,
- parametric polymorphism

**Object Oriented Programming**

Makes it easy to adapt and extend complex systems, using

- subtyping and inheritance,
- dynamic configurations,
- classes as partial abstractions
Functional Programming

- Key idea: do everything by writing functions and composing them
  - no mutable state variables
  - no side effects:
    - no modification to any state variable when evaluating an expression

```
<table>
<thead>
<tr>
<th>Function</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>old value</td>
<td>function₁</td>
</tr>
</tbody>
</table>
```

- Emphasis is on "transformation" rather than on "state"
- Functional programs are timeless: if a program produces a result for a given set of arguments, it will always produce the same result for the same arguments
Key idea: do everything by writing functions and composing them
- no mutable state variables
- no side effects:
  no modification to any state variable when evaluating an expression

Emphasis is on “transformation” rather than on “state”

Functional programs are timeless:
if a program produces a result for a given set of arguments, it will always produce the same result for the same arguments
• Recursion takes the place of iteration
• Functions are first-class citizens:
  • Functions can take functions as arguments and can return functions as results
Functional Programming

• Recursion takes the place of iteration
• Functions are first-class citizens:
  • Functions can take functions as arguments and can return functions as results

Pros and Cons

• John Hughes: “Why Functional Programming Matters”
• Read Norman Ramsey’s comment on Pitfalls/Disadvantages of Functional Programming
  • http://stackoverflow.com/questions/1786969/
• Some advantages:
  - functions can be reasoned about in isolation
  - facilitates parallel and distributed programming
Scala: Alternative for Java

Scala

Java
(9M developers)

scalac

javac

JVM

Java runs on more than 850 million PCs worldwide (Oracle, 2015)
In Java:

```java
public class Person {
    public final String name;
    public final int age;
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```
In Java:

```java
public class Person {
    public final String name;
    public final int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

In Scala:

```scala
class Person(val name:String, val age:Int)
```
Usage

In Java:

```java
import java.util.ArrayList;
...
Person[] people;
Person[] minors;
Person[] adults;
ArrayList<Person> minorsList = new ArrayList<Person>();
ArrayList<Person> adultsList = new ArrayList<Person>();
for (int i = 0; i < people.length; i++)
    (people[i].age > 18 ? adultsList : minorsList)
        .add(people[i]);
minors = minorsList.toArray(people);
adults = adultsList.toArray(people);
```
**Usage**

**In Java:**

```java
import java.util.ArrayList;
...
Person[] people;
Person[] minors;
Person[] adults;
ArrayList<Person> minorsList = new ArrayList<Person>();
ArrayList<Person> adultsList = new ArrayList<Person>();
for (int i = 0; i < people.length; i++)
    (people[i].age > 18 ? adultsList : minorsList)
        .add(people[i]);
minors = minorsList.toArray(people);
adults = adultsList.toArray(people);
```

**In Scala:**

```scala
val people = Array[Person]
val (minors, adults) = people partition (_.age ≤ 18)
```

*An infix method call*

*A function value*

*A simple pattern match*
Guy Steele: “Switching from Java to Scala reduced size of Fortress typechecker by a factor of 4”.
Guy Steele: “Switching from Java to Scala reduced size of Fortress typechecker by a factor of 4”.

Question: Does it matter? Doesn’t Eclipse write the extra lines for me?
Guy Steele: “Switching from Java to Scala reduced size of Fortress typechecker by a factor of 4”.

**Question:** Does it matter? Doesn’t Eclipse write the extra lines for me?

**Answer:** This does matter:

- Eye-tracking experiments: For program comprehension, average time spent per word of source code is constant.
- Roughly, half the code means half the time necessary to understand
• In Scala, variables are functions
• In Java, if age is a public field of Person, you can say:
  ```java
  zack.age = zack.age + 1;
  ```
• but if age is accessed via methods, you would say:
  ```java
  zack.setAge(zack.getAge() + 1);
  ```
Referential Transparency

- In Scala, variables are functions
- In Java, if age is a public field of Person, you can say:
  ```java
  zack.age = zack.age + 1;
  ```
- but if age is accessed via methods, you would say:
  ```java
  zack.setAge(zack.getAge() + 1);
  ```
- In Scala, if age is a public field of Person, you can say:
  ```scala
  zack.age = zack.age + 1
  ```
- but if Person defines getter and setter methods age and age_, you would say:
  ```scala
  zack.age = zack.age + 1
  ```
Referential Transparency

- In Scala, variables are functions
- In Java, if age is a public field of Person, you can say:
  ```java
  zack.age = zack.age + 1;
  ```
- but if age is accessed via methods, you would say:
  ```java
  zack.setAge(zack.getAge() + 1);
  ```
- In Scala, if age is a public field of Person, you can say:
  ```scala
  zack.age = zack.age + 1
  ```
- but if Person defines getter and setter methods age and age_,
you would say:
  ```scala
  zack.age = zack.age + 1
  ```

**Principle of Uniform Access:**

- If you want to access a piece of data in Scala, you don’t have to know whether it is:
  - computed by a method, or,
  - held in a simple variable
- Scala won’t let you use parentheses when you call a function with no parameters
Class Hierarchies and ADTs

- Unify class hierarchies and abstract data types (ADTs)
- Introduce pattern matching for objects
- Concise manipulation of immutable data structures

Class hierarchy for binary trees:

```scala
abstract class Tree[T]
case object Empty extends Tree[Nothing]
case class Binary[T](elem:T,left:Tree[T],right:Tree[T]) extends Tree[T]
def inOrder[T](t:Tree[T]): List[T] = t match {
  case Empty => List()
  case Binary(e,l,r) =>
    inOrder(l) ::: List(e) ::: inOrder(r)
}
```
Functions and Collections

- First-class functions make collections more powerful
- Cascading operations on immutable collections

Example

- Check if there are two or more adults with the same name in the `people` class

```scala
class Person(val name: String, val age: Int)
people.filter(_.age > 18)
  .groupBy(_.name)
  .mapValues(_.size)
  .exists(_._2 > 1)
```
Functions and Collections

• First-class functions make collections more powerful
• Cascading operations on immutable collections

Example

• Check if there are two or more adults with the same name in `people`

```scala
class Person(val name: String, val age: Int)

people.filter(_.age > 18)
  .groupBy(_.name)
  .mapValues(_.size)
  .exists(_._2 > 1)
```
“You can write Fortran in any language” (old programming quote)

- You can bring your old programming habits into the new language
- Write exactly the same kind of program you would in Fortran, whether they make sense or not
- Totally ignore the distinctive character of the new language!
“You can write Fortran in any language” (old programming quote)

- You can bring your old programming habits into the new language
- Write exactly the same kind of program you would in Fortran, whether they make sense or not
- Totally ignore the distinctive character of the new language!

Moral

- “You can write a Java program in Scala”
- OK at first – you have to start out with what you know, which is Java
- After that, you have a choice:
  - You can gradually learn the Scala way, or
  - You can keep writing crappy Scala programs.
“I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn’t resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.”
null in Scala

In Java:

- Any method that is supposed to return an object could return null
- Some options:
  - Always check for null
  - Always put your method calls inside a try ... catch
  - Make sure the method can’t possibly return null
  - Ignore the problem and depend on luck
null in Scala

In Java:

- Any method that is supposed to return an object could return null
- Some options:
  - Always check for null
  - Always put your method calls inside a try ... catch
  - Make sure the method can’t possibly return null
  - Ignore the problem and depend on luck

In Scala:

- Scala has null—but only so that it can talk to Java
- In Scala, if a method could return “nothing,” write it to return an Option object
- Option: either Some(theObject) or None
- This forces you to use a match statement—but only when one is really needed!