The software landscape today …
… resembles a tower of Babel with many little (or not so little) languages playing together.

E.g.
- JavaScript on the client
- Perl/Python/Ruby/Groovy for server side scripting
- JavaFX for the UI
- Java for the business logic
- SQL for database access
all cobbled together with a generous helping of XML.

This is both good and bad

Good: Every language can concentrate on what it's best at.

Bad: Cross language communication:
- complicated, fragile, source of misunderstandings.

Problematic: Cross language communication is controlled by a common type system (neither static nor dynamic).
- It's based on low-level representations such as XML trees or (worse) strings (as in JDBC database queries).

Alternative: Scalable languages

A language is scalable if it is suitable for very small as well as very large programs.

A single language for extension scripts and the heavy lifting.

Application-specific needs are handled through libraries and embedded DSL's instead of external languages.

Scala shows that this is possible.

Scala is a scripting language

It has an interactive read-eval-print-loop (REPL).

Types can be inferred.

Boilerplate is scrapped.

```
scala> var capital = Map("US" → "Washington", "France" → "Paris")
capital: Map[String, String] = Map(US → Washington, France → Paris)
scala> capital += ("Japan" → "Tokio")
scala> capital("France")
res7: String = Paris
```

Scala is the Java of the future

It has basically everything Java has now.
(sometimes in different form)

It has closures.
(proposed for Java 7, but rejected)

It has traits and pattern matching.
(I would not be surprised to see them in Java 8, 9 or 10)

It complies to classic files, is completely interoperable and runs about as fast as Java.
### Interoperability

Scala fits seamlessly into a Java environment.

- Can call Java methods, select Java fields, inherit Java classes, implement Java interfaces, etc.
- None of this requires glue code or interface descriptions.
- Java code can also easily call into Scala code.
- Scala code resembling Java is translated into virtually the same bytecode.

⇒ Performance is usually on a par with Java.

---

### Scala is a composition language

New approach to module systems:

- Component = class or trait
- Composition via mixins
- Abstraction through traits
- > parameters,
- > abstract members (both types and values),
- > self types
- Gives dependency injection for free

---

### Is Scala a “kitchen-sink language”?*

Not at all. In terms of feature count, Scala is roughly comparable to today’s Java and smaller than C# or C++.

But Scala is deep, where other languages are broad.

Two principles:

1. Focus on abstraction and composition, so that users can implement their own specialized features as needed.
2. Have the same sort of constructs work for very small as well as very large programs.

---

### Scala compared to Java

<table>
<thead>
<tr>
<th>Scala adds</th>
<th>Scala removes</th>
</tr>
</thead>
<tbody>
<tr>
<td>- a pure object system</td>
<td>- static members</td>
</tr>
<tr>
<td>- operator overloading</td>
<td>- primitive types</td>
</tr>
<tr>
<td>- closures</td>
<td>- break, continue</td>
</tr>
<tr>
<td>- mixin composition with traits</td>
<td>- special treatment of interfaces</td>
</tr>
<tr>
<td>- existential types</td>
<td>- wildcards</td>
</tr>
<tr>
<td>- abstract types</td>
<td>- raw types</td>
</tr>
<tr>
<td>- pattern matching</td>
<td>- enums</td>
</tr>
</tbody>
</table>

Modeled in libraries:
- assert, enums, properties, events, actors, using, queries, …

---

### Scala cheat sheet (1): Definitions

#### Scala method definitions:

```scala
def fun(x: Int): Int = {
    result
}
```  
#### Java method definition:

```java
int fun(int x) {
    return result;
}
```  
#### Scala variable definitions:

```scala
var x: Int = expression
val x: String = expression
```  
#### Java variable definitions:

```java
int x = expression
final String x = expression
```  
---

### Scala cheat sheet (2): Expressions

#### Scala method calls:

```scala
obj.meth(arg)
obj.methods
```  
#### Java method call:

```java
obj.meth(arg)
```  
#### Scala choice expressions:

```scala
if (cond) expr1 else expr2
```  
#### Java choice expressions, stmts:

```java
cond ? expr1 : expr2
```  
---

---
Scala cheat sheet (3): Objects and Classes

Scala Class and Object

```scala
class Sample(x: Int, val p: Int) {
  def instMeth(y: Int) = x + y
}
```

Java Class with statics

```java
class Sample {
  private final int x;
  public final int p;
  Sample(int x, int p) {
    this.x = x;
    this.p = p;
  }
  def instMeth(y) {
    return x + y;
  }
  static int staticMeth(int x, int y) {
    return x * y;
  }
}
```

Scala cheat sheet (4): Traits

Scala Trait

```scala
trait T {
  def abstractMth(x: String): Int
  def concreteMth(x: String) = 
}
```

Java Interface

```java
interface T {
  int abstractMth(String x)
}
```

Scala mixin composition:
```scala
class C extends Super with T
```

Java extension + implementation:
```java
class C extends Super implements T
```

Spring Cleaning

Scala’s syntax is lightweight and concise.

Due to:
- semicolon inference,
- type inference,
- lightweight classes,
- extensible APIs,
- closures as control abstractions.

Average reduction in LOC: ≥2

due to concise syntax and better abstraction capabilities

→ Scala feels like a cleaned up Java ...

... with one major difference

It’s x: Int instead of int x
Why the change?
Works better with type inference:
```scala
var capital = Map("US" -> "Washington",
  "Canada" -> "Ottawa")
capital += ("Japan" -> "Tokyo")
for (c <- capital.keys)
capital(c) = capital(c).capitalize
assert(capital("Canada") == "Ottawa")
```

Works better for large type expressions:
```scala
val x: HashMap[String, (String, List[Char])] = ...
```

Scalability demands extensibility

Take numeric data types
Today’s languages support int, long, float, double.
Should they also support BigInt, BigDecimal, Complex, Rational, Interval, Polynomial?

There are good reasons for each of these types
But a language combining them all would be too complex.

Better alternative: Let users grow their language according to their needs.

Adding new datatypes - seamlessly

For instance type BigInt:
```scala
def factorial(x: BigInt): BigInt =
  if (x == 0) 1 else x * factorial(x - 1)
```

Compare with using Java’s class:
```scala
import java.math.BigInteger
def factorial(x: BigInteger): BigInteger =
  if (x == BigInteger.ZERO)
    BigInteger.ONE
  else
    x.multiply(factorial(x.subtract(BigInteger.ONE)))
```
Implementing new datatypes - seamlessly

Here's how BigInt is implemented

```java
import java.math.BigInteger

class BigInt(val bigInteger: BigInteger)

+ is an identifier; can be used as a method name
Infix operations are method calls:
  a + b is the same as a.(b)
  a add b is the same as a.add(b)
```

```scala
extends java.lang.Number {
  def + (that: BigInt) =
    new BigInt(this.bigInteger add that.bigInteger)
  def - (that: BigInt) =
    new BigInt(this.bigInteger subtract that.bigInteger)
  ...
}
```

Adding new control structures

For instance using for resource control (proposed for Java 7)

```scala
val f = new BufferedReader(new FileReader(path))
try {
  println(f.readLine())
} finally {
  if (f != null) f.close()
}
```

Instead of:

```scala
using (new BufferedReader(new FileReader(path))) {
  f => println(f.readLine())
}
```

Implementing new control structures:

Here's how one would go about implementing using:

```scala
def using[T <: { def close() }](resource: T)(block: T => Unit) {
  try {
    block(resource)
  } finally {
    if (resource != null) resource.close()
  }
```

Break and continue

Scala does not have them. Why?
> They are a bit imperative; better use many smaller functions.
> Issues how to interact with closures.
> They are not needed!
We can support them purely in the libraries.

```scala
import scala.util.control.Breaks._
breakable {
  for (x <- elems) {
    println(x * 2)
    if (x > 0) break
  }
}
```

Getting back break and continue

What makes Scala scalable?

Many factors: strong typing, inference, little boilerplate,…
But mainly, its tight integration of functional and object-oriented programming.

<table>
<thead>
<tr>
<th>Functional programming:</th>
<th>Object-oriented programming:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Makes it easy to build interesting things from simple parts, using higher-order functions, algebraic types and pattern matching, parametric polymorphism.</td>
<td>Makes it easy to adapt and extend complex systems, using subtyping and inheritance, dynamic configurations, classes as partial abstractions.</td>
</tr>
</tbody>
</table>
Scala is object-oriented

Every value is an object
Every operation is a method call
Exceptions to these rules in Java (such as primitive types, statics) are eliminated.

```
scala> (1).hashCode
res8: Int = 1
scala> (1).+(2)
res10: Int = 3
```

Scala is functional

Scala is a functional language, in the sense that every function is a value.
Functions can be anonymous, curried, nested.
Many useful higher-order functions are implemented as methods of Scala classes. E.g:

```
scala> val matrix = Array(Array(1, 0, 0),
 |   Array(0, 1, 0),
 |   Array(0, 0, 1))
matrix: Array[Array[Int]] = Array([I@164da25,...
scala> matrix.exists(row => row.forall(0 ==))
res13: Boolean = false
```

Functions are objects

If functions are values, and values are objects, it follows that functions themselves are objects.
The function type S => T is equivalent to scala.Function1[S, T], where Function1 is defined as follows:

```
trait Function1[-S, +T] {
  def apply(x: S): T
}
```

So functions are interpreted as objects with apply methods.
For example, the anonymous successor function

```
(x: Int) => x + 1
```

is expanded to:

```
new Function1[Int, Int] {
  def apply(x: Int) = x + 1
}
```

Why should I care?

Since (=>) is a class, it can be subclassed.
So one can specialize the concept of a function.
An obvious use is for arrays, which are mutable functions over integer ranges.
A bit of syntactic sugaring lets one write:

```
a(i) = a(i) + 2 for a.update(i, a.apply(i) + 2)
```

Partial functions

Another useful abstraction are partial functions.
These are functions that are defined only in some part of their domain.
What's more, one can inquire with the isDefinedAt method whether a partial function is defined for a given value.

```
trait PartialFunction[-A, +B] extends (A => B) {
  def isDefinedAt(x: A):!Boolean
}
```

Scala treats blocks of pattern matching cases as instances of partial functions.
This lets one write control structures that are not easily expressible otherwise.

Developing new paradigms

Scala's flexibility makes it possible for users to grow the language into completely new paradigms.
Case in point: concurrent programming
Since Scala is interoperable, Java threads and concurrent libraries are available.
But it's also possible to explore completely new paradigms.
Erlang-style actors

Two principal constructs (adopted from Erlang):
Send (!) is asynchronous;
messages are buffered in an actor’s mailbox.
receive picks the first message in
the mailbox which matches any of
the patterns msgpat1.
If no patterns match, the actor
suspends.

// asynchronous message send
actor ! message

// message receive
receive {
  case msgpat1 => action1
  ... case msgpatn => actionn
}

A pattern matching block of type
PartialFunction[MessageType, ActionType]

A simple actor

```
case class Data(bytes: Array[Byte])
case class Sum(receiver: Actor)
val checkSumCalculator = actor {
  var sum = 0
  loop {
    receive {
      case Data(bs) => sum += hash(bs)
      case Sum(receiver) => receiver ! sum
    }
  }
}
```

Implementing receive

Using partial functions, it is straightforward to implement receive:
```
def receive[T] (f: PartialFunction[Message, T]): T = {
  self.mailBox.extractFirst(f.isDefinedAt)
  match {
    case Some(msg) =>
      f(msg)
    case None =>
      self.wait(messageSent)
  }
}
```

Here, self designates the currently
executing actor,
```
mailBox is its queue of pending messages, and
```
```
eventuallyFirst extracts first queue element matching given predicate.
```

Other Approaches to Scalability

C++
> Hard to scale down.
> Scaling up is possible for expert users.

.NET
> Many languages with common interoperability.
> Hard to do something that’s really different.

Java
> Lingua franca makes it easy to understand other people’s code.
> Not easy to scale down or up → pressure to add new languages.

Where are we now?

Scala
> Easy to scale down and up.
> Works well with a mix of expert users (for the framework) and non-
> experts (for the application code).
Scala solves the expressiveness challenge for doing this.
But does it also solve the safety issues?
> Problem: How to ensure that domain-specific code stays within its
domain-specific library/language?
> For instance: How to ensure that a query formulated in Scala is non-
> recursive?
Addressed by ongoing project: Pluggable type systems

The Scala community

50,000 downloads in 2008
300+ track contributors
20+ messages/day on the mailing lists
Industrial adoption has started, among
others at:
Twitter, Sony Pictures, Nature.com,
Reaktor, Mimesis Republic,
EDF Trading, ...
Scala talks in many conferences, next two
at QCon, London, March 10-12.
**Tool support**

- Standalone compiler: `scalac`
- Fast background compiler: `fsc`
- Interactive interpreter shell and script runner: `scala`
- Web framework: `lif`
- Testing frameworks: `Specs`, `ScalaCheck`, `ScalaTest`, `JUnit`, ...
- IDE plugins for:
  - Eclipse (supported by EDF)
  - IntelliJ (supported by JetBrains)
  - Netbeans (supported by Sun)

**Who's using it?**

- Open source projects:
  - Lift
  - Wicket
  - NetLogo
  - SPDE: Scala branch for Processing
  - Isabelle: GUI and code extractor

- Companies:
  - Twitter: infrastructure
  - Sony Pictures: middleware
  - Nature.com: infrastructure
  - Reaktor: many different projects
  - Mimesis Republic: multiplayer games

**Learning Scala**

To get started:
- First steps in Scala, by Bill Venners published in Scalazine at www.artima.com
- Scala for Java Refugees by Daniel Spiewack (great blog series)

To continue:
- Programming in Scala, by Odersky, Spoon, Venners, published by Artima.com
- Other books are in the pipeline.

**Thank You**

To try it out:
- scala-lang.org

Thanks also to the (past and present) members of the Scala team:

**Relationship between Scala and other languages**

Main influences on the Scala design:
- Java, C# for their syntax, basic types, and class libraries.
- Smalltalk for its uniform object model.
- Eiffel for its uniform access principle.
- ML, Haskell for many of the functional aspects.
- OCaml, PLT-Scheme, as other (less tightly integrated) combinations of FP and OOP.
- Pizza, Multi Java, Nice as other extensions of the Java platform with functional ideas.

Too many influences in details to list them all.

Scala also seems to influence other new language designs, see for instance the closures and comprehensions in LINQ/CLR 3.0.