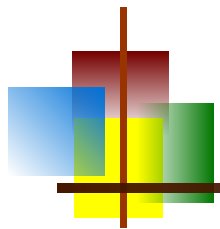




Java to Scala





Types

- Primitives

- char
- byte
- short
- int
- long
- float
- double
- boolean

- Objects

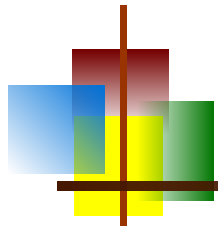
- String
- ... and many more ...

- Objects

- Char
- Byte
- Short
- Int
- Long
- Float
- Double
- Boolean
- String
- Function
- ... and many more ...

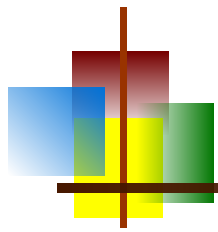
Scala doesn't have primitives, only objects

Behind the scenes, objects are converted to primitives as necessary, so there is no loss in efficiency



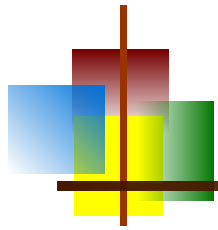
Type declarations

- `int x;`
- `final int Y = 0;`
- `int[] langs = {"C++", "Java", "Scala"};`
- `Set<String> langs = new Set<>();`
`langs.add("C++");`
`langs.add("Java");`
`langs.add("Scala");`
- `var x = 0`
 - The keyword `var` introduces a mutable variable
 - Variable declarations must include an initial value
 - The type of variable is inferred from the initial value
- `val y = 0`
 - The keyword `val` introduces an immutable variable
- `val langs = Array("C++", "Java", "Scala")`
- `val langs = Set("C++", "Java", "Scala")`



“Statements”

- Scala’s “statements” should really be called “expressions,” because *every statement has a value*
- The value of many statements, for example the while loop, is `()`
 - `()` is a value of type `Unit`
 - `()` is the *only* value of type `Unit`
 - `()` basically means “*Nothing to see here. Move along.*”
- The value of a `if` or `match` statement is the last value computed
- The value of a block, `{...}`, is the last value computed in the block
- A statement is ended by the end of the line (not with a semicolon) unless it is obviously incomplete, or if the next line cannot begin a valid statement
 - For example, `x = 3 * (2 * y +` is obviously incomplete
 - Because Scala lets you leave out a lot of unnecessary punctuation, sometimes a line that you *think* is complete really isn’t complete (or vice versa)
- You *can* end statements with semicolons, but that’s not good Scala practice

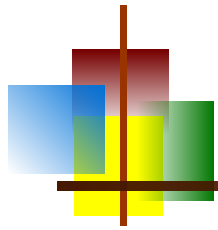


Constructors

- ```
class Point {
 private double x;
 private double y;

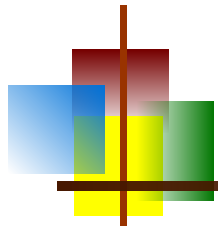
 public Point(double x, double y) {
 this.x = x;
 this.y = y;
 }
 ...
}
```
- ```
class Point(x: Double, y: Double) {  
    ...  
}
```

 - Unqualified parameters are effectively private final instance variables



Getters

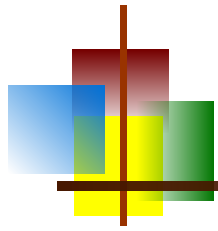
- Java: Make the instance variables private, and write getter methods
- ```
class Point {
 private double x, y;
 public double getX() {
 return x;
 }
}
```
- ```
Point p = new Point(3.6, 4.7);  
System.out.println(p.getX());
```
- Scala: Replace the primary constructor line with
- ```
class Point(val x: Double, val y: Double) {...}
```
- ```
val p = new Point(3.6, 4.7)  
println(p.x)
```



Both getters and setters

- Java: Write as methods
- ```
public double getX() {
 return x;
}
public void setX(Double x) {
 this.x = x;
}
```
- ```
p.setX(2 * p.getX());
```
- Scala: Replace the primary constructor line with

```
class Point(var x: Double, var y: Double)
```
- ```
p.x = 2 * p.x
```



# Auxiliary constructors

---

- Java: Can have multiple constructors, which may or may not refer to one another

- `public Point(double x, double y) {...}`

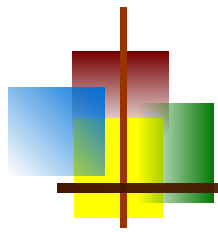
```
public Point() {
 this(0, 0);
}
```

- `class Point(x: Double, y: Double) {...}`

```
def this() {
 this(0, 0)
}
```

- Every auxiliary constructor must refer to a previously defined constructor

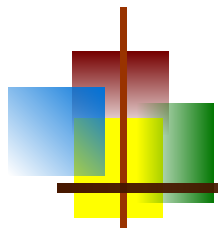




# Defining equality for objects

- `@Override`  

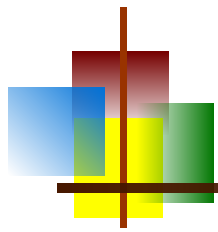
```
public boolean equals(Object other) {
 if (other instanceof Point) {
 Point that = (Point) other;
 return this.x == that.x && this.y == that.y;
 }
 return false;
}
```
- `final override def equals(Other: Any) = {`  
    `val that = other.asInstanceOf[Point]`  
    `if (that == null) false`  
    `else this.x == that.x && this.y == that.y`
- **OR:**  
`case class Point(x: Double, y: Double) {...}`



# Case classes in Scala

---

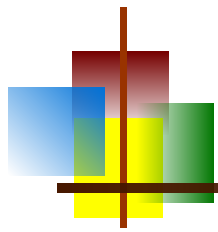
- A **case class** is just like a regular class, except:
  - The methods **equals**, **hashCode**, **toString**, and **copy** are automatically defined for you, based on the parameters to the primary constructor
  - Each of the constructor parameters becomes a **var** (so you have getters and setters)
  - An **apply** method is created—this lets you omit the word **new** when creating new objects
  - An **unapply** method is created, which allows you to pattern match on your objects



# Input and output

---

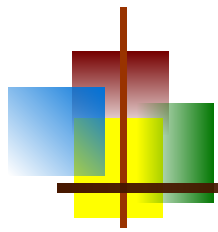
- `Scanner scanner = new Scanner(System.in);`  
`System.out.println("What is your name? ");`  
`String name = scanner.nextLine();`
  - `System.out.println("Hello, " + name);`
- 
- `val name = readLine("What is your name? ")`
  - `println("Hello, " + name)`



# Singleton objects

---

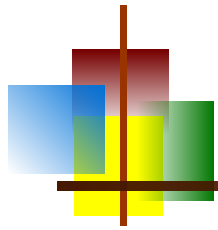
- ```
class Earth {  
    final double diameter = 7926.3352;  
    Earth earth = e;  
    private Earth() {}  
    public instanceOf() {  
        if (e == null) e = new Earth();  
        return e;  
    }  
}
```
- ```
object Earth {
 val diameter = 7926.3352
}
```



# Operators in Scala

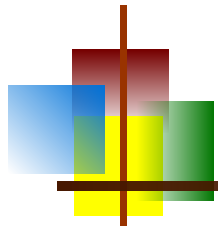
---

- Scala has the same arithmetic and logical operators as Java, except:
  - `++` and `--` have been removed
  - `test ? iftrue : iffalse` has been replaced by `if (test) iftrue else iffalse`  
which *is* an expression



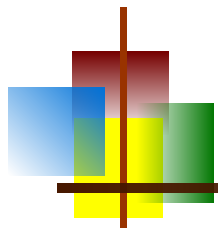
# Familiar statement types

- These are the same as in Java, but have a value of `( )`:
  - `variable = expression // also +=, *=, etc.`
  - `while (condition) { statements }`
  - `do { statements } while (condition)`
- These are the same as in Java, but may have a useful value:
  - `{ statements }`
    - The value of the block is the last value computed in it
  - `if (condition) { statements } else { statements }`
    - The value is the value of whichever block is chosen
    - If the value is to be used, both blocks should have the same type, otherwise the type of the result is the “least upper bound” of the two types
  - `if (condition) { statements }`
    - The value is the value of the last statement executed, but its type is `Any` – if you want a value, you really should use an `else`
- As in Java, braces around a single statement may be omitted



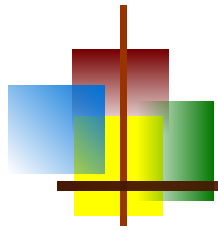
# The `for` comprehension

- Scala's `for` is much more powerful than Java's `for`
  - Consequently, it is used much more often than the other kinds of loops
- We will just cover some simple cases here
- `for (i <- 1 to 10) { println(i) }`
  - Prints the numbers 1 through 10
- `for (i <- 1 until 10) { println(i) }`
  - Prints the numbers 1 through 9
- `for (x <- 0 until myArray.length) { println(myArray(x)) }`
  - Prints all the values in `myArray`
- `for (x <- myArray) { println(x) }`
  - Prints all the values in `myArray`
- `for (x <- myArray  
    if x % 2 == 0) { println(x) }`
  - Prints all the even numbers in `myArray`



# for...yield

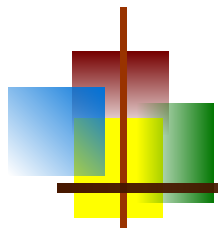
- `for` returns `Unit`, but `for...yield` returns a sequence of values
  - Where possible, it returns the same type of sequence as it operates on
  - ```
scala> for (i <- List(1, 2, 3)) yield { 2 * i }  
res1: List[Int] = List(2, 4, 6)
```
 - ```
scala> for (i <- 97 to 100) yield { i.toChar }
res2: scala.collection.immutable.IndexedSeq[Char] =
Vector(a, b, c, d)
```
  - ```
scala> for (ch <- "abcd" if ch != 'c') yield { ch.toInt }  
res3: scala.collection.immutable.IndexedSeq[Int] =  
Vector(97, 98, 100)
```

Explicit pattern matching

- Explicit pattern matching is done with the `match` method:

```
expression match {  
  case pattern1 => expressions  
  ...  
  case patternN => expressions  
}
```



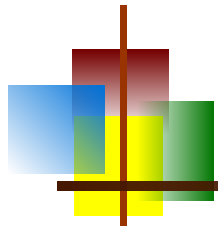
Pattern matching

- Pattern matching on literal values:

- ```
today match {
 case "Saturday" => println("Party! Party! Party!")
 case "Sunday" => println("Pray....")
 case day => println(day + " is a workday. :(")
}
```

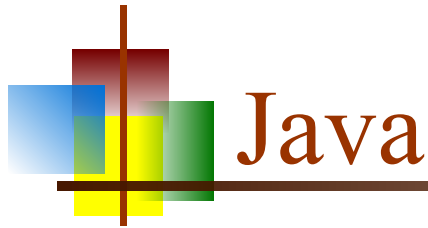
- Pattern matching on types:

- ```
something match {  
  case x: Int => println("I'm the integer " + x)  
  case x: String =>  
    println("I'm the String \"" + x + "\"")  
    println("My length is " + x.length)  
  case _ => println("I don't know what I am!  :( ")  
}
```

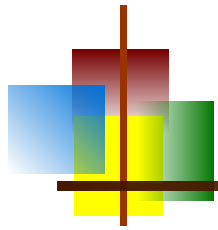


The Option type

- Scala has `null` because it interoperates with Java; it shouldn't be used any other time
- Instead, use an `Option` type, with values `Some(value)` and `None`
 - ```
def max(list: List[Int]) = {
 if (list.length > 0) {
 val big = list reduce {(a, b) => if (a > b) a else b}
 Some(big)
 } else {
 None
 }
}
```
  - ```
max(myList) match {  
  case Some(x) => println("The largest number is " + x)  
  case None => println("There are no numbers here!!!")  
}
```

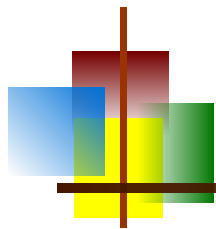


- What's wrong with Java?
 - Not designed for highly concurrent programs
 - The original Thread model was just *wrong* (it's been fixed)
 - Java 5+ helps by including `java.util.concurrent`
 - Verbose
 - Too much of `Thing thing = new Thing();`
 - Too much “boilerplate,” for example, getters and setters
- What's right with Java?
 - Very popular
 - Object oriented (mostly), which is important for large projects
 - Statically typed (more on this later)
 - The fine large library of classes
 - **The JVM!** Platform independent, highly optimized



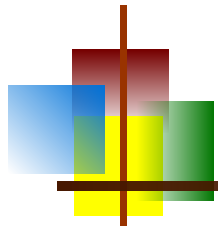
Pet peeves

- Here are some things that annoy me about Java but are fixed in Scala
 - `==` works for strings about 95% of the time
 - If you write a constructor for your class, the default constructor vanishes
 - For arrays, `length` is a variable; for Strings, it's a function
 - `ArrayList<String> strings = new ArrayList<String>();`
 - Special syntax for arrays (only), not for any other data structures
 - Adding elements to a list or a map, one at a time
 - Having to explain `++` and `--` to new students
 - By default, `assert` doesn't do anything
 - Having to write getters and setters to make my code “clean,” even when they aren't really needed
 - Having to create a `Scanner` to do simple input (for that matter, all file I/O)
 - Checked exceptions
 - No tuples!
 - `NullPointerExceptions`



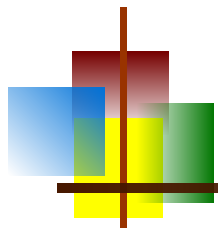
Scala is like Java, except when it isn't

- Java is a *good* language, and Scala is a lot like it
- For each difference, there is a *reason*--none of the changes are “just to be different”
- Scala and Java are (almost) completely interoperable
 - Call Java from Scala? No problem!
 - Call Scala from Java? Some restrictions, but mostly OK. No problem—if you can supply the right type of parameters
 - Scala compiles to **.class** files (a *lot* of them!), and can be run with either the **scala** command or the **java** command
- To understand Scala, it helps to understand the reasons for the changes, and what it is Scala is trying to accomplish



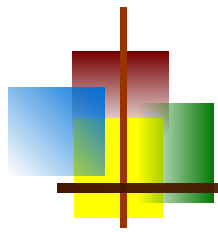
Consistency is good

- In Java, every value is an object--unless it's a primitive
 - Numbers and booleans are primitives for reasons of efficiency, so we have to treat them differently (you can't "talk" to a primitive)
- In Scala, all values are objects. Period.
 - The compiler turns them into primitives, so no efficiency is lost (behind the scenes, there are objects like [RichInt](#))
- Java has *operators* (+, <, ...) and *methods*, with different syntax
- In Scala, operators are just methods, and in many cases you can use either syntax



Type safety is good, verbosity is bad

- Java is **statically typed**--a variable has a type, and can hold *only* values of that type
 - You must specify the type of every variable
 - Type errors are caught by the compiler, not at runtime--this is a big win
 - However, it leads to a lot of typing (pun intended)
- Languages like Ruby and Python don't make you declare types
 - Easier (and more fun) to write programs
 - Less fun to debug, especially if you have even slightly complicated types
- Scala is *also* statically typed, but it uses **type inferencing**--that is, it figures out the types, so you don't have to
 - The good news: Less typing, more fun, type errors caught by the compiler
 - The bad news: More kinds of error messages to get familiar with



Verbosity

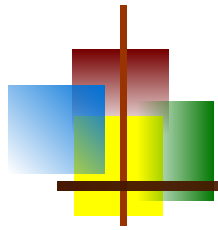
■ Java:

```
■ 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 void String getFirstName() { return this.firstName; }  
    public void setLastName(String lastName) { this.lastName = lastName; }  
    public void String getLastName() { return this.lastName; }  
    public void setAge(int age) { this.age = age; }  
    public void int getAge() { return this.age; }  
}
```

■ Scala:

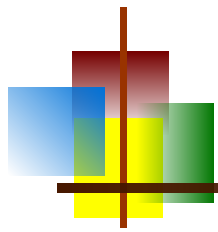
```
■ class Person(var firstName: String, var lastName: String, var age: Int)
```

■ Source: <http://blog.objectmentor.com/articles/2008/08/03/the-seductions-of-scala-part-i>



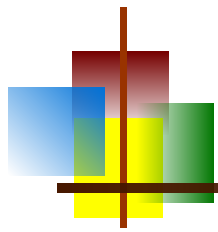
null in Scala

- In Java, any method that is *supposed* to return an object *could* return **null**
 - Here are your 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
 - <http://www.youtube.com/watch?v=u0-oinyjsk0>
- Yes, 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, which is either **Some(*theObject*)** or **None**
 - This forces you to use a **match** statement--but only when one is really needed!



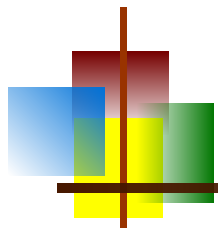
Uniform access

- In Java, `myString.length()` is a function, but `myArray.length` is a variable
- If `age` is a public field of `Person`, you can say:
`david.age = david.age + 1;`
but if `age` is accessed via methods, you would say:
`david.setAge(david.getAge() + 1);`
- You have to know whether a piece of data is *implemented* as a variable or as a function
- In Scala, if `age` is a public field of `Person`, you can say:
`david.age = david.age + 1;`
but if `Person` defines methods `age` and `age_`, you would say:
`david.age = david.age + 1;`
- In other words, 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
 - This is the **principle of uniform access**
 - Scala *won't let you* use parentheses when you call a function with no parameters



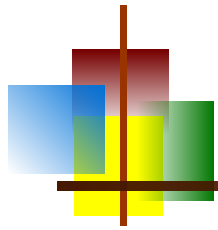
Concurrency

- “Concurrency is the new black.”
- 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
- 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



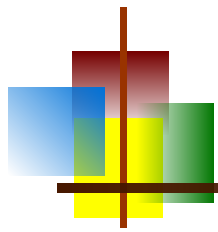
Scala is multiparadigm

- Scala is an attempt to blend object-oriented programming with functional programming
- Here's the difficulty:
 - Objects have state—that's practically their only reason for being
 - Functional programs are stateless
- Scala tries to bridge this gap
 - Functions in Scala are first-class objects
 - Scala encourages immutable objects
 - All the usual functional programming functions—**map**, **filter**, **fold**, etc.—are available in Scala



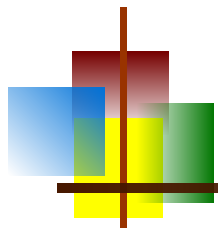
Functional languages

- The best-known functional languages are ML, OCaml, and Haskell
- Functional languages are regarded as:
 - “Ivory tower languages,” used only by academics (mostly but not entirely true)
 - Difficult to learn (mostly true)
 - The solution to all concurrent programming problems everywhere (exaggerated, but not entirely wrong)
- Scala is an “impure” functional language--you can program functionally, but it isn’t forced upon you



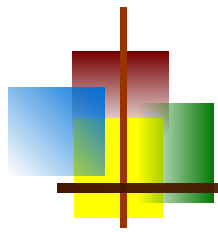
Scala as a functional language

- The hope--*my* hope, anyway--is that Scala will let people “sneak up” on functional programming (FP), and gradually learn to use it
 - This is how C++ introduced Object-Oriented programming
- Even a little bit of functional programming makes some things a lot easier
- Meanwhile, Scala has plenty of other attractions
- FP really is a different way of thinking about programming, and not easy to master...
- ...but...
- Most people that master it, never want to go back

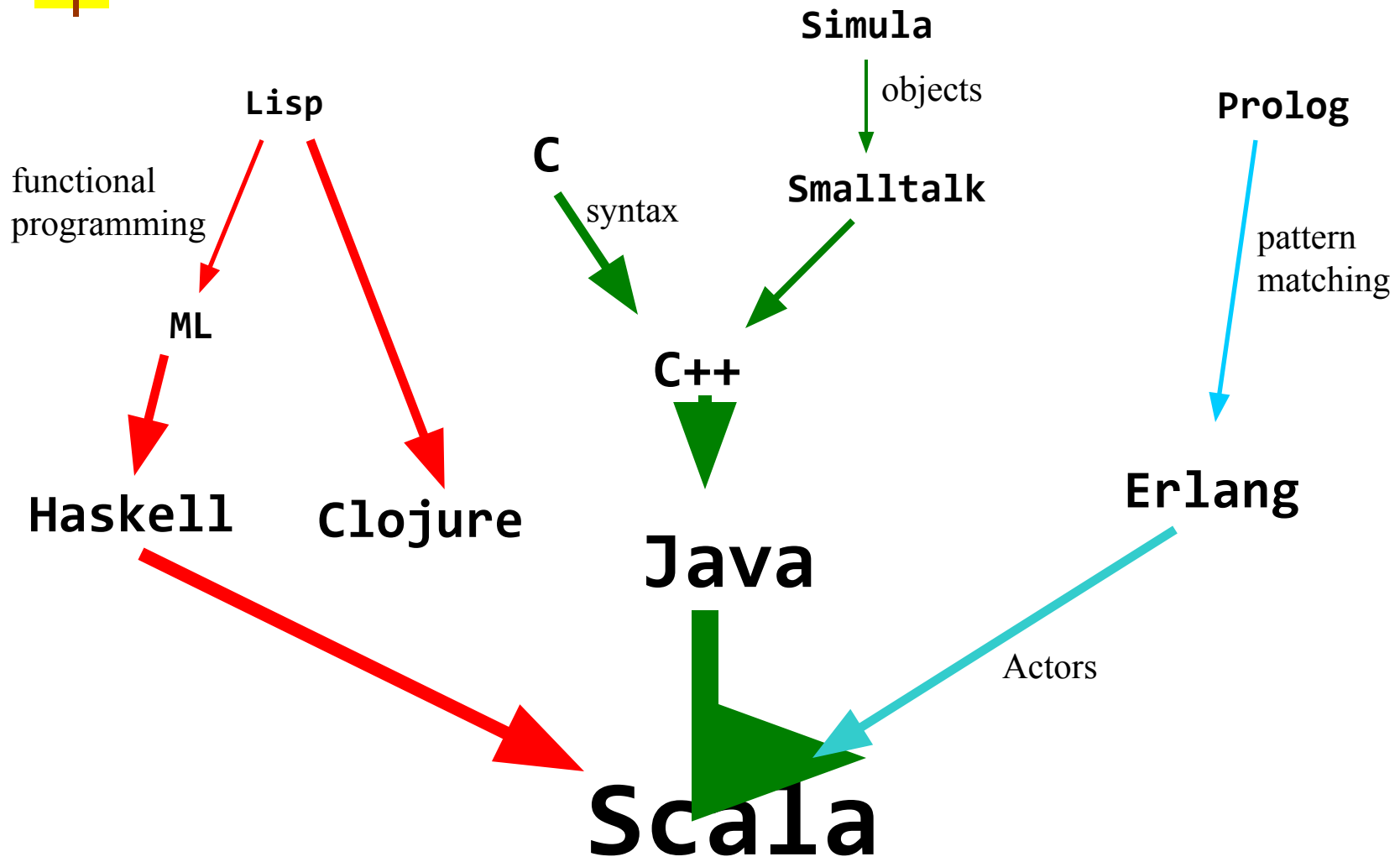


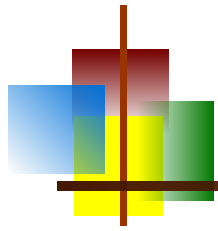
“You can write a Fortran program...”

- There’s a old saying: “You can write a Fortran program in any language.”
 - Some people quote this as “You can write a C program...,” but the quote is older than the C language
- People still say this, but I discovered recently that what they mean by it has changed (!)
- **Old meaning:** You can bring your old (Fortran) programming habits into the new language, writing exactly the same kind of program you would in Fortran, whether they make sense or not, and just totally ignore the distinctive character of the new language.
- **New meaning:** You can write a crappy program in any language.
- **Moral:** You can “write a Java program in Scala.” That’s okay 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.



Genealogy





The End

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

--James Gosling, creator of Java