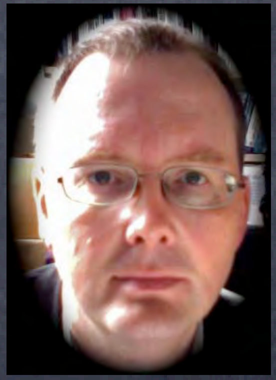
Grace

A New Educational Object-Oriented Programming Language

Andrew Black

Kim Bruce



James Noble


ECOOP 2010, Maribor, Slovenia

Supporters

- Peter Andreae, Victoria University of Wellington
- Gilad Bracha, Ministry of Truth
- John Boyland, University of Wisconsin, Milwaukee
- Pascal Constanza, Vrije Universiteit Brussel
- Sophia Drossopoulou, Imperial College, London
- Susan Eisenbach, Imperial College, London
- Michael Hicks, University of Maryland
- Michael Kölling, University of Kent at Canterbury
- Gary Leavens, University of Central Florida
- Shane Markstrum, Bucknell University
- Doug Lea, SUNY Oswego
- Dirk Riehle, Friedrich-Alexander-University of Erlangen-Nürnberg
- Ewan Tempero, The University of Auckland
- Dave Thomas, Bedarra Research Labs
- Laurence Tratt, Middlesex University
- Jan Vitek, Purdue University

Design by a really small committee

Supported by a **wide** community

Public Blog: http://www.gracelang.org

Obvious Questions:

- What is an educational language?
- Why not use a "real" language?
- Why not Java? Scala? Python?
- Why now?
- Non-Questions:
  - why start with objects?
  - why teach objects at all?

What is an educational programming language?

- Designed specifically for novices
- Can have limited or broad domain of application
  - We are interested in broad domain
- Main focus is on programming in the small, but some modularity features.

Teach Industrial- Strength Languages?

- Too much conceptual redundancy
- High overhead for simple programs
  - Too hard to read and write
- Conceptual clarity sacrificed for practicality
- Saddled w/backward compatibility

Why Not Java?

- Overloading
- Confusing subtyping with inheritance
- No user-defined operators
- Primitive & Object types
- No lambdas
- Weak support for Generics
- Covariant arrays
- Equality not automatic
- No definable control structures
- Synchronized

Why Not Scala?

- Too complex for novices
- Multiple ways of doing everything
- Weak generics
- Powerful, but complex, type system

Why Not Python?

- Weak encapsulation
- Can't teach typed programming
- Mismatch between method declarations & message sends
- __init__
- Implicit creation of fields

Why Now?

- Happy teaching Java next 3-5 years
- In 2015, Java will be 20 years old
- State of the art has advanced
  - patches look like ... patches
- New languages bring good ideas
  - ... but are for professionals, not students
- To be ready in 2015, we need to start now.

Our User Model

- First year students in OO CS1 or CS2
  - objects early or late,
  - static or dynamic types,
  - functionals first or scriptings first or ...
- Second year students
- Faculty & TAs — assignments and libraries

We are in the dog food business

User model:
Beginning
students

Customer:
experienced
instructors

The consumer is not the customer

The Big Question

- What do we hope the students learn?
  1. To program well in Grace?
  2. To understand and use the o-o model?
  3. To be prepared for other languages and models?
- My position: 3 is less important than 1 and 2

Features

- Uncluttered code; layout significant
- Structural typing
- Local type inference
- Subtyping separated from inheritance
- User-definable operators
- Sensible generics
- Lambdas
- Allows both static and dynamic typing
- Parallel programming
- Equals & hashcode work automatically
- v instead of getV() for access
- Minimize "incantations" ~~public static void main~~

Warning!

Warning!

- Design is in early phases

Warning!

- Design is in early phases
- Ambitious goals

Warning!

- Design is in early phases
- Ambitious goals
- Still disagree on many details

Grace Fundamentals

- Everything is an object
- Simple method dispatch
- Single inheritance via cloning and concatenation
- Language levels for teaching
- Extensible via Libraries (control & data)
- Java / C / Python / Scala programmers should be able to read Grace programs

Simple Grace Example

```
method average -> Number
// reads numbers from this stream and averages them
{  var total := 0
   var count := 0
   until {atEnd} do {
      count := count + 1
      total := total + readNumber }
   if (count = 0) then {return 0}
   return total / count }
```

Simple Grace Example

Might appear in a stream object

```
method average -> Number
// reads numbers from this stream and averages them
{  var total := 0
   var count := 0
   until {atEnd} do {
      count := count + 1
      total := total + readNumber }
   if (count = 0) then {return 0}
   return total / count }
```

Simple Grace Example

```
method average -> Number
// reads numbers from this stream and averages them
{  var total := 0
   var count := 0
   until {atEnd} do {
      count := count + 1
      total := total + readNumber }
   if (count = 0) then {return 0}
   return total / count }
```

Simple Grace Example

```
method average -> Number
// reads numbers from this stream and averages them
{  var total := 0
   var count := 0
   until {atEnd} do {
      count := count + 1
      total := total + readNumber }
   if (count = 0) then {return 0}
   return total / count }
```

implicit self.

Simple Grace Example

```
method average -> Number
// reads numbers from this stream and averages them
{  var total := 0
   var count := 0
   until {atEnd} do {
      count := count + 1
      total := total + readNumber }
   if (count = 0) then {return 0}
   return total / count }
```

Everything is an Object

- except for methods
- Functions are objects
  - as in Smalltalk, lambda expressions create objects that mimic functions

```
const welcomeAction := { print "Hello" }
```

Everything is an Object

- except for methods
- Functions are objects
  - as in Smalltalk, lambda expressions create objects that mimic functions

```
const welcomeAction := { print "Hello" }
```

Everything is an Object

- except for methods
- Functions are objects
  - as in Smalltalk, lambda expressions create objects that mimic functions

```
const welcomeAction := { print "Hello" }
```

```
object { method apply
                   { print "Hello" } }
```

Everything is an Object

- except for methods
- Functions are objects
    - as in Smalltalk, lambda expressions create objects that mimic functions

```
const welcomeAction := { print "Hello" }
```

Everything is an Object

- except for methods
- Functions are objects
  - as in Smalltalk, lambda expressions create objects that mimic functions

```
const welcomeAction := { print "Hello" }
```

Everything is an Object

- except for methods
- Functions are objects
  - as in Smalltalk, lambda expressions create objects that mimic functions

```
const welcomeAction := { print "Hello" }
```

```
welcomeAction.apply
```

Everything is an Object

- except for methods
- Functions are objects
  - as in Smalltalk, lambda expressions create objects that mimic functions

```
const welcomeAction := { print "Hello" }
```

Everything is an Object

- except for methods
- Functions are objects
  - as in Smalltalk, lambda expressions create objects that mimic functions

```
const orderingFunction := { a, b → a.name ≤ b.name }
```

Everything is an Object

- except for methods
- Functions are objects
  - as in Smalltalk, lambda expressions create objects that mimic functions

```
const orderingFunction := { a, b → a.name ≤ b.name }
```

Everything is an Object

- except for methods
- Functions are objects
  - as in Smalltalk, lambda expressions create objects that mimic functions

```
const orderingFunction := { a, b → a.name ≤ b.name }
```

```
object { method apply(a, b) {
                a.name ≤ b.name } }
```

Everything is an Object

- except for methods
- Functions are objects
  - as in Smalltalk, lambda expressions create objects that mimic functions

```
const orderingFunction := { a, b → a.name ≤ b.name }
```

Everything is an Object

- except for methods
- Functions are objects
  - as in Smalltalk, lambda expressions create objects that mimic functions

```
const orderingFunction := { a, b → a.name ≤ b.name }
```

Everything is an Object

- except for methods
- Functions are objects
  - as in Smalltalk, lambda expressions create objects that mimic functions

```
const orderingFunction := { a, b → a.name ≤ b.name }
```

```
if orderingFunction.apply(x, y) then { … }
```

Everything is an Object

- except for methods
- Functions are objects
  - as in Smalltalk, lambda expressions create objects that mimic functions

```
const orderingFunction := { a, b → a.name ≤ b.name }
```

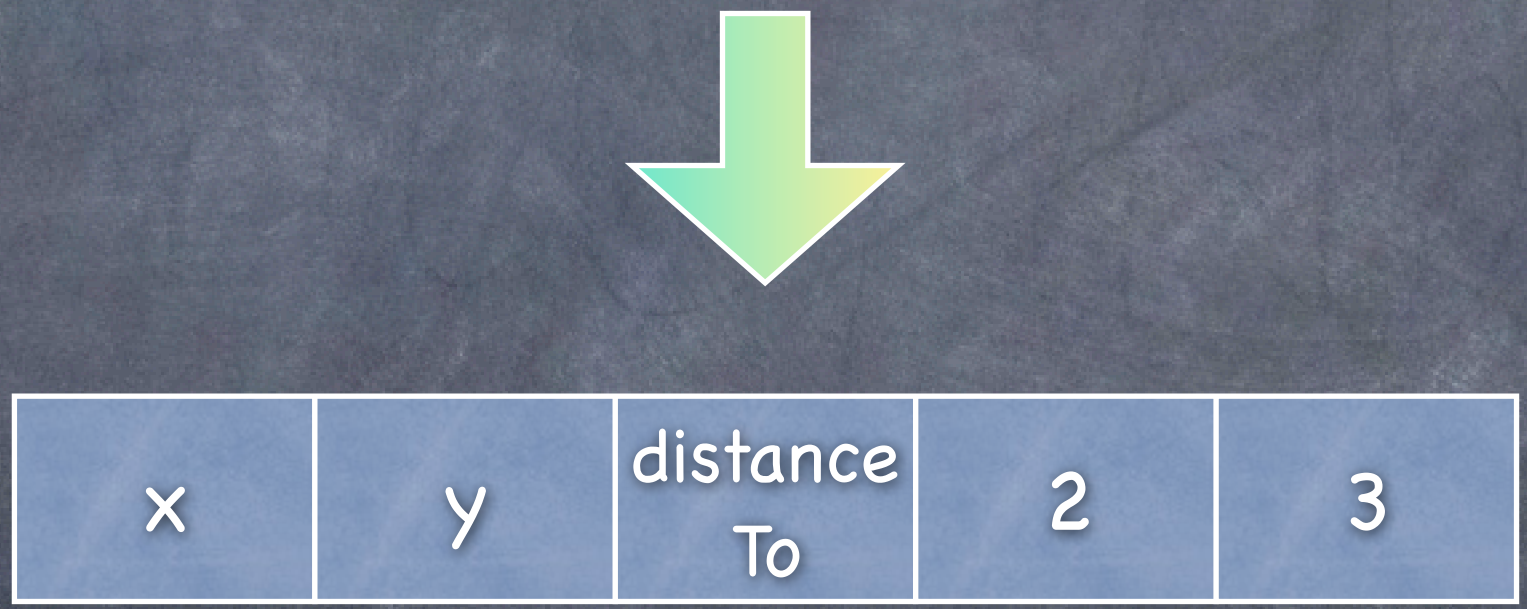
Everything is an Object

- But every object is not an instance of a class
- Instead: objects are self-contained
- Objects are created by executing an object constructor:

```
object {
   const x:Number := 2
   const y:Number := 3
   method distanceTo other:Point → Number {
      ((x - other.x)^2 + (y - other.y)^2) } }
```

```
object {
  const x:Number := 2
  const y:Number := 3
  method distanceTo other:Point → Number {
    ((x - other.x)^2 + (y - other.y)^2) } }
```

```
object {
  const x:Number := 2
  const y:Number := 3
  method distanceTo other:Point → Number {
    ((x - other.x)^2 + (y - other.y)^2) } }
```



```
object {
   const x:Number := 2
   const y:Number := 3
   method distanceTo other:Point → Number {
      ((x - other.x)^2 + (y - other.y)^2) } }
```

Design Decisions:

- fields and methods share the same namespace
- p.x might be a field access or a method request
- the implementation can replace a field by a method without the client knowing

What about classes?

- Pro
  - Instructors are familiar with classes
  - Classes capture a common pattern: a "factory" object that makes similar "instance" objects
  - Brevity
- Con
  - Unnecessary — just use objects
  - The common pattern usually lies in some way
  - Restrictive, e.g. Smalltalk's parallel hierarchies

Compromise Design

- Grace has classes; they resemble a block containing an object constructor
- We try to make the syntax familiar, but not so familiar that we lie
- Classes are restrictive, but the full power of object constructors is available to implement the general case

Point Class

```
const Point := class { x’: Number, y’:Number →
    const x:Number := x’
    const y:Number := y’
    method distanceTo other:Point → Number {
        ((x - other.x)^2 + (y - other.y)^2) }
    }
```

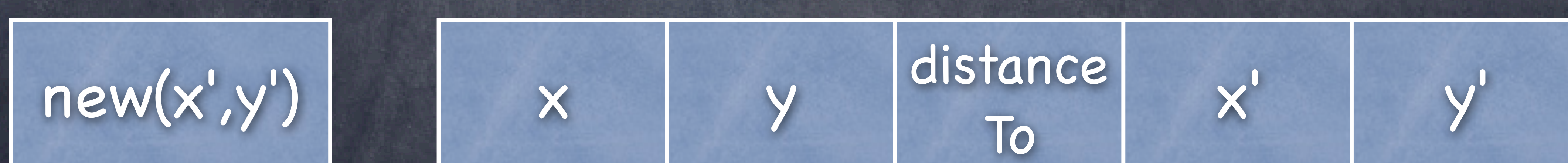
Point Class

```
const Point := class { x': Number, y':Number →
    const x:Number := x'
    const y:Number := y'
    method distanceTo other:Point → Number {
        ((x - other.x)^2 + (y - other.y)^2) }
    }
```

new(x',y')

Point Class

```
const Point := class { x′: Number, y′:Number →
    const x:Number := x′
    const y:Number := y′
    method distanceTo other:Point → Number {
        ((x - other.x)^2 + (y - other.y)^2) }
    }
```



Point Class

```
const Point := class { x': Number, y':Number →
    const x:Number := x'
    const y:Number := y'
    method distanceTo other:Point → Number {
        ((x - other.x)^2 + (y - other.y)^2) }
    }
```

Point Class

```
const Point = object {
  method new (x′:Number, y′:Number)  {
      object {
         const x:Number := x′
         const y:Number := y′
         method distanceTo other:Point→Number {
              ((x - other.x)^2 + (y - other.y)^2) }}}
}
```

Class: Summary

```
const Point := class { x': Number, y':Number →
   const x:Number := x'
   const y:Number := y'
   method distanceTo other:Point → Number {
      ((x - other.x)^2 + (y - other.y)^2) }
   }
```

```
const Point := object {
      method new (x':Number, y':Number)  {
         object {
            const x:Number := x'
            const y:Number := y'
            method distanceTo other:Point→Number {
               ((x - other.x)^2 + (y - other.y)^2) }}}
}
```

One true message send

- Like Smalltalk and Self:
  - no overloading
  - "method request" names the method and provides the arguments
  - "dynamic dispatch" selects the correspondingly-named method in the receiver
  - "method execution" occurs in the receiver
  - field access is via methods

One true message send

- Like Smalltalk and Self:
  - no overloading
  - "method request" names the method and provides the arguments
  - "dynamic dispatch" selects the correspondingly-named method in the receiver
  - "method execution" occurs in the receiver
  - field access is via methods

(I'm trying to learn <u>not</u> to say "message-send" or "method call".)

Example: a Contact Object

```
const andrewInfo := object {
   var firstName := "Andrew"
   const lastName := "Black"
   method printOn s:Stream {
      s.puts firstName
      s.puts ' '
      s.puts lastName }
}
```

Example: a Contact Object

```
const andrewInfo := object {
    var firstName := "Andrew"
    const lastName := "Black"
    method printOn s:Stream {
        s.puts firstName
        s.puts ' '
        s.puts lastName }
}
```

Creates a method lastname

Example: a Contact Object

Creates 2 methods: firstName and firstName:=

Creates a method lastname

```
const andrewInfo := object {
  var firstName := "Andrew"
  const lastName := "Black"
  method printOn s:Stream {
    s.puts firstName
    s.puts ' '
    s.puts lastName }
}
```

Contact Object Expanded

```
const andrewInfo := object {
   privar ¿firstName?
   method firstName -> String { ¿firstName? }
   method firstName:= s:String { ¿firstName? := s }
   ¿firstname? := "Andrew"
   priconst ¿lastName?
   method lastName -> String { ¿lastName? }
   ¿lastname? := "Black"
   method printOn s:Stream {
      s.puts firstName
      s.puts ' '
      s.puts lastName }
}
```

Contact Object Expanded

```
const andrewInfo := object {
   privar ¿firstName?
   method firstName -> String { ¿firstName? }
   method firstName:= s:String { ¿firstName? := s }
   ¿firstname? := "Andrew"
   priconst ¿lastName?
   method lastName -> String { ¿lastName? }
   ¿lastname? := "Black"
   method printOn s:Stream {
      s.puts firstName
      s.puts ' '
      s.puts lastName }
}
```

<u>Not</u> proposed for surface syntax

Contact Factory

```
const contact := object {
   method named (first, last) -> Contact {
      object {
         var firstName:String := first
         var lastName:String := last
         method printOn s:Stream {
            s.puts firstName
            s.puts ' '
            s.puts lastName } } }
   const database := MutableSequence.empty
   method add c:Contact {
      database addLast c }


}
```

Contact Factory

```
const contact := object {
  method named (first, last) -> Contact {
     object {
        var firstName:String := first
        var lastName:String := last
        method printOn s:Stream {
           s.puts firstName
           s.puts ' '
           s.puts lastName } } }
  const database := MutableSequence.empty
  method add c:Contact {
     database addLast c }

}
```

attributes of the outer "factory" object

Contact Factory

```
const contact := object {
  method named (first, last) -> Contact {
    object {
      var firstName:String := first
      var lastName:String := last
      method printOn s:Stream {
        s.puts firstName
        s.puts ' '
        s.puts lastName } } }
  const database := MutableSequence.empty
  method add c:Contact {
    database addLast c }



}
```

Contact Factory

```
const contact := object {
  method named (first, last) -> Contact {
    object {
      var firstName:String := first
      var lastName:String := last
      method printOn s:Stream {
        s.puts firstName
        s.puts ' '
        s.puts lastName } } }
  const database := MutableSequence.empty
  method add c:Contact {
    database addLast c }


}
```

returns a contact object initialized to (first, last)

Contact Factory

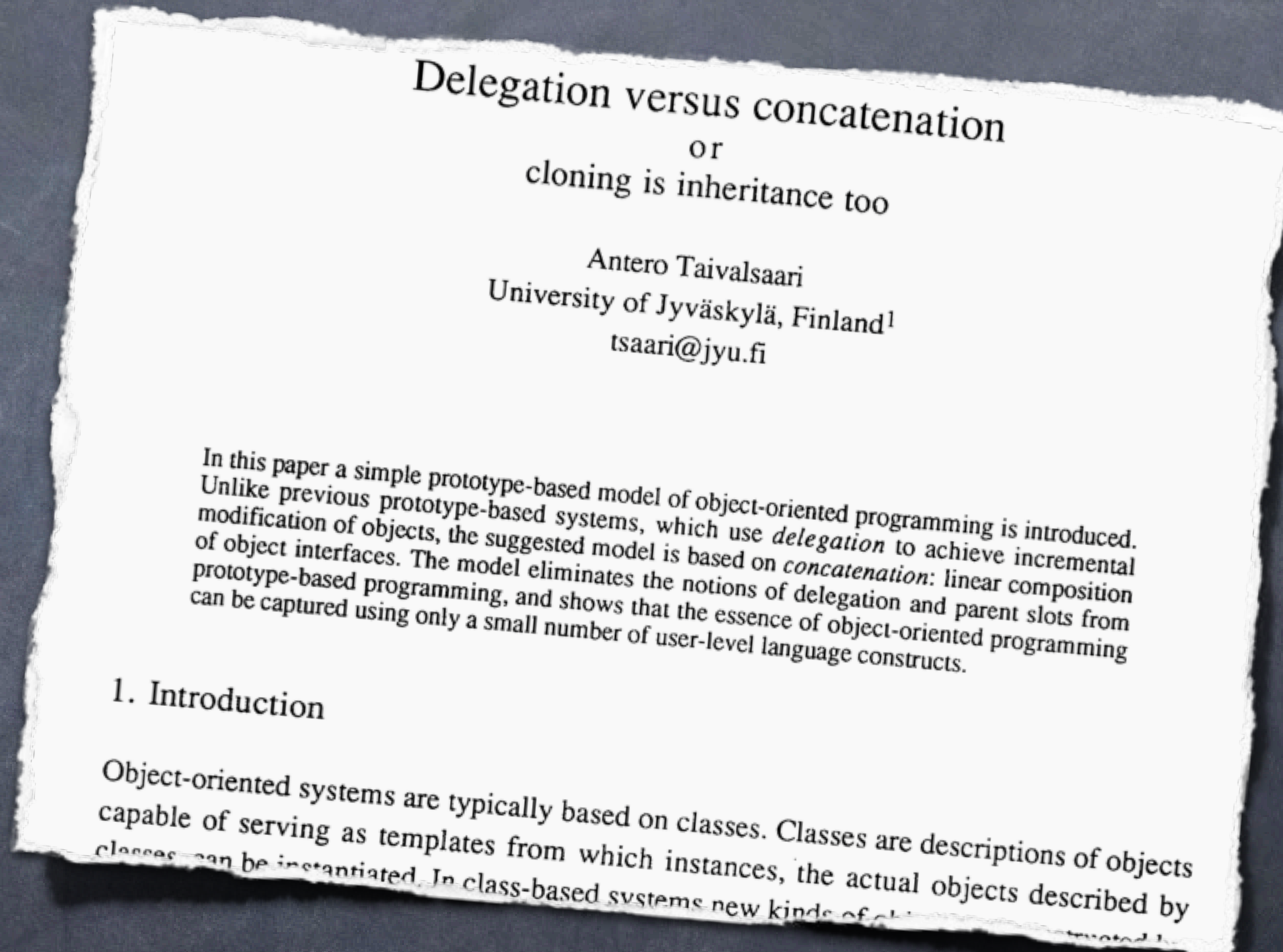
```
const contact := object {
  method named (first, last) -> Contact {
    object {
      var firstName:String := first
      var lastName:String := last
      method printOn s:Stream {
        s.puts firstName
        s.puts ' '
        s.puts lastName } } }
  const database := MutableSequence.empty
  method add c:Contact {
    database addLast c }



}
```

Sample client code

```
const host := contact.named("Graham", "Hutton")
const guest := contact.named("Andrew", "Black")
contact.database.add host
contact.database.add guest
```

Inheritance

- Grace's inheritance story is based on an old idea of Taivalsaari



Delegation versus concatenation
or
cloning is inheritance too

Antero Taivalsaari
University of Jyväskylä, Finland[1]
tsaari@jyu.fi

In this paper a simple prototype-based model of object-oriented programming is introduced. Unlike previous prototype-based systems, which use *delegation* to achieve incremental modification of objects, the suggested model is based on *concatenation*: linear composition of object interfaces. The model eliminates the notions of delegation and parent slots from prototype-based programming, and shows that the essence of object-oriented programming can be captured using only a small number of user-level language constructs.

1. Introduction

Object-oriented systems are typically based on classes. Classes are descriptions of objects capable of serving as templates from which instances, the actual objects described by classes, can be instantiated. In class-based systems new kinds of

- Cloning + Concatenation = inheritance

ACM SIGPLAN OOPS Messenger Volume 6 Issue 3, July 1995

Suppose that we have an object:

andrewInfo

first Name	last Name	last Name:=	printOn	"Andrew"	"Black"

Suppose that we have an object:

andrewInfo

first Name	last Name	last Name:=	printOn	"Andrew"	"Black"

We want to add a telephone number. The "delta" is:

Suppose that we have an object:

andrewInfo

first Name	last Name	last Name:=	printOn	"Andrew"	"Black"

We want to add a telephone number. The "delta" is:

andrewPhone

work phone	work phone:=	"+1 503 725 2411"

Suppose that we have an object:

andrewInfo

first Name	last Name	last Name:=	printOn	"Andrew"	"Black"

We want to add a telephone number. The "delta" is:

andrewPhone

work phone	work phone:=	"+1 503 725 2411"

1. Clone both objects

Suppose that we have an object:

andrewInfo

first Name	last Name	last Name:=	printOn	"Andrew"	"Black"

We want to add a telephone number. The "delta" is:

andrewPhone

work phone	work phone:=	"+1 503 725 2411"

1. Clone both objects

Suppose that we have an object:

andrewInfo

first Name	last Name	last Name:=	printOn	"Andrew"	"Black"

We want to add a telephone number. The "delta" is:

andrewPhone

work phone	work phone:=	"+1 503 725 2411"

1. Clone both objects

2. Concatenate the copies

andrewInfo

first Name	last Name	last Name:=	printOn	"Andrew"	"Black"

andrewPhone

work phone	work phone:=	"+1 503 725 2411"

andrewPhone extends andrewInfo

work phone	work phone:=	"+1 503 725 2411"	first Name	last Name	last Name:=	printOn	"Andrew"	"Black"

In code:

```
const andrewInfo := object {
   var firstName := "Andrew"
   const lastName := "Black"
   method printOn s:Stream {
      s.puts firstName
      s.puts ' '
      s.puts lastName }
}

const andrewPhone := object {
   var officePhone := "503 725 2411"
}

const andrewPhoneInfo := andrewPhone extends andrewInfo
```

No need to name intermediate objects:

```
const andrewPhoneInfo := object {
  var officePhone := "503 725 2411"
} extends contact.named ("Andrew", "Black")
```

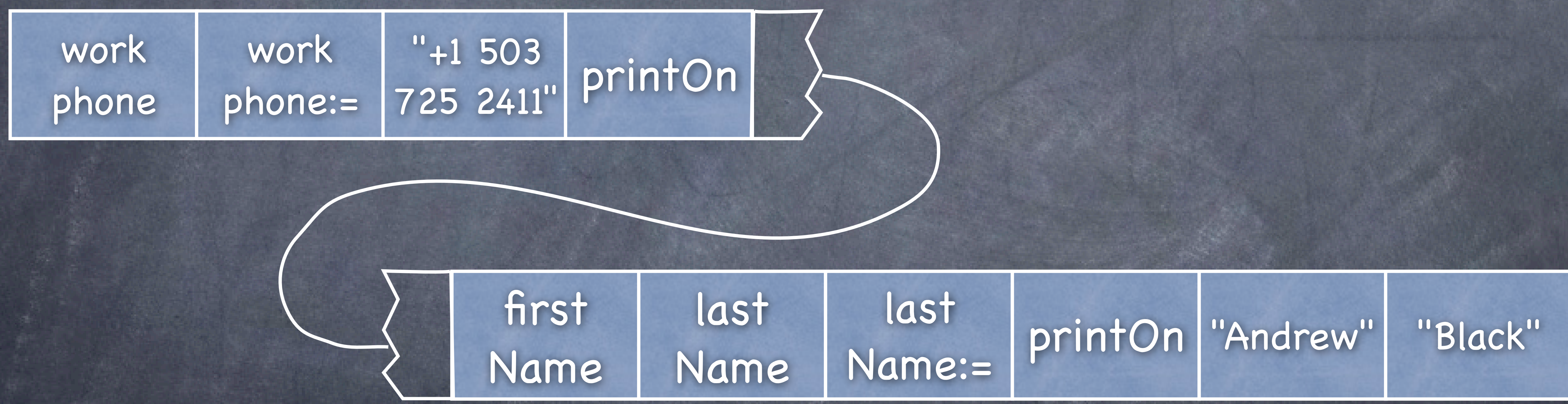
Notice what this means:

- clone means shallow copy:
- new object gets copies of the fields and the methods of the orignal objects
- it's possible for an object to have two or more methods with the same name

Let's fix printOn...

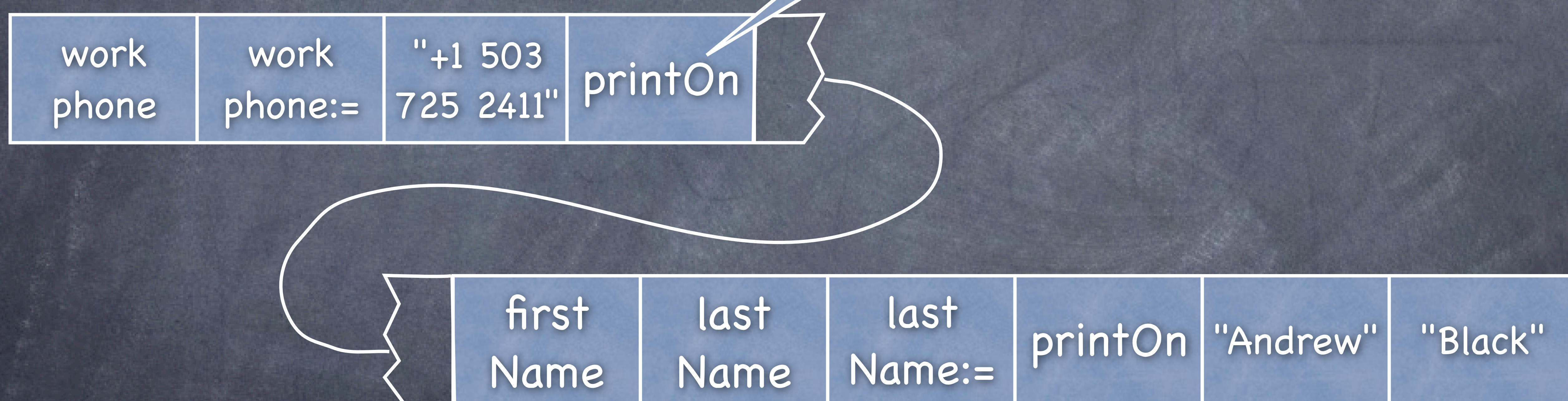
```
const andrewPhoneInfo := object {
   var officePhone := "503 725 2411"
   method printOn s:Stream {
      super.printOn s
      s.puts ' '
      s.puts officePhone
   }
} extends contact.new ("Andrew", "Black")
```

andrewPhoneInfo

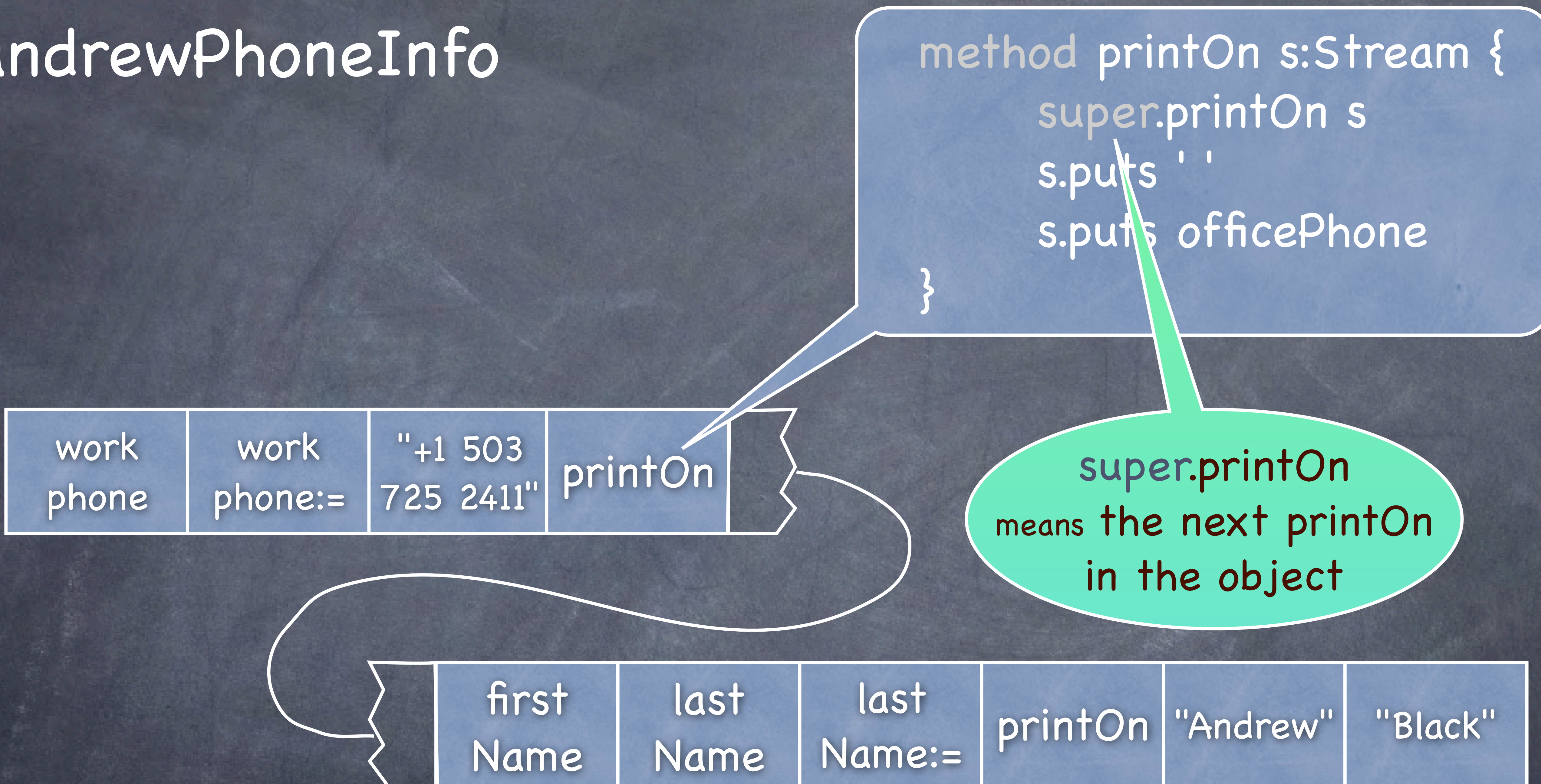


andrewPhoneInfo

```
method printOn s:Stream {
        super.printOn s
        s.puts ' '
        s.puts officePhone
}
```

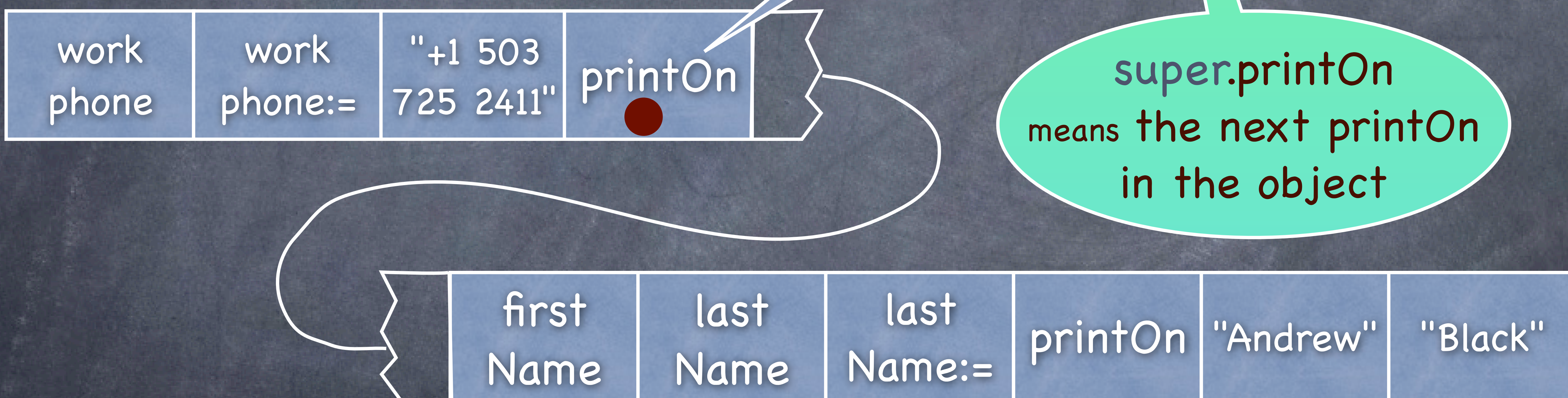


andrewPhoneInfo



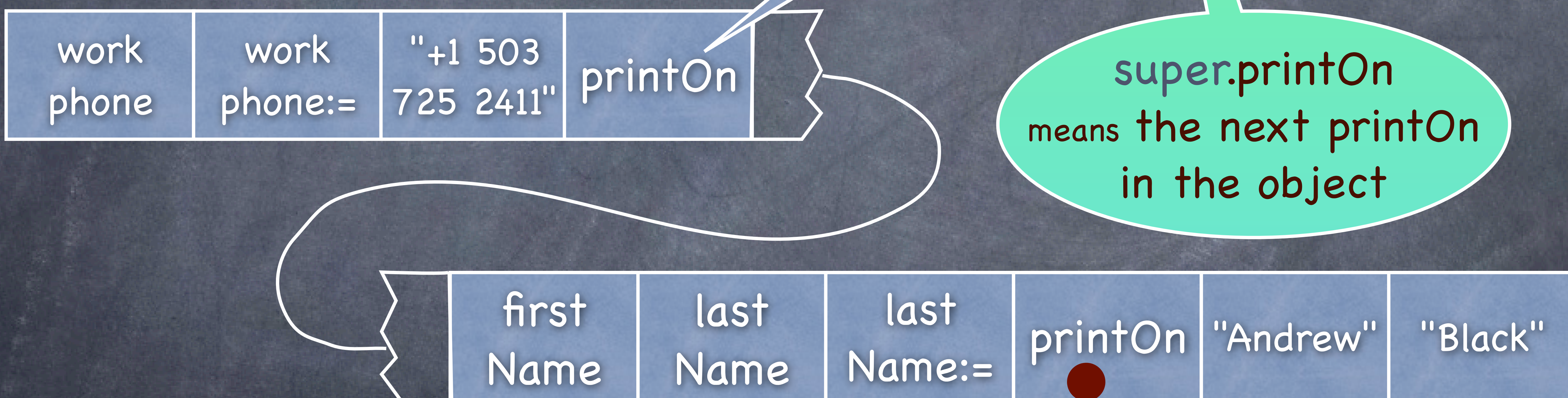
andrewPhoneInfo

```
method printOn s:Stream {
        super.printOn s
        s.puts ' '
        s.puts officePhone
}
```



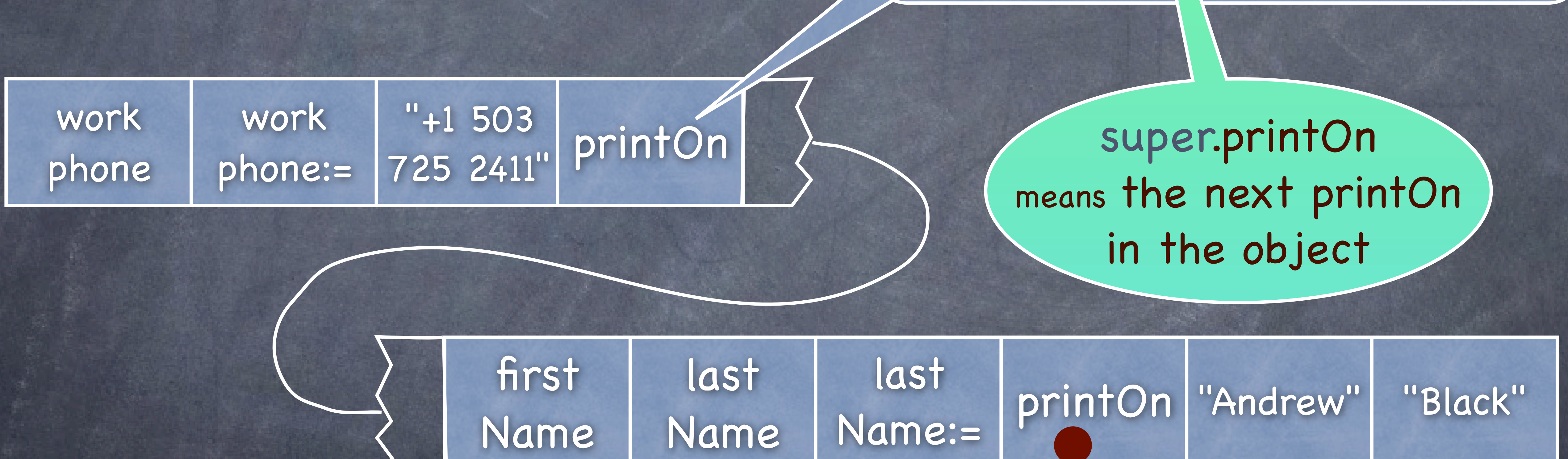
andrewPhoneInfo

```
method printOn s:Stream {
        super.printOn s
        s.puts ' '
        s.puts officePhone
}
```



andrewPhoneInfo

```
method printOn s:Stream {
    super.printOn s
    s.puts ' '
    s.puts officePhone
}
```



next might be a better keyword than super

Not your Grandfather's Inheritance

Not your Grandfather’s Inheritance

- What’s more important:
  - To have a simple, explicable, inheritance story?
  - To have an inheritance story that’s like the mainstream languages of the 1990’s?

Not your Grandfather’s Inheritance

- What’s more important:
  - To have a simple, explicable, inheritance story?
  - To have an inheritance story that’s like the mainstream languages of the 1990’s?
- Or:
  - Teachability, vs. familiar to instructors

Extensible via libraries

- Objects include data and actions
  - So it's essential to be able to define new control operations on new objects
- Example: a new kind of dictionary
  - It must be possible to define new iterators, lookups, etc. <u>with a syntax that's as convenient for the user as the "built in" objects</u>

We achieve this!

- Nothing is built in!
- As in SELF, all built-in objects are really defined in libraries, including the Booleans.
- if <condition> then <block> else <block>,
  while <block> do <block>, and
  with <collection> do <block>
  are all method requests.
- The methods are defined on object *Grace*, and inherited by all other objects

```
object Grace := {
   method if c:Boolean
              then t:Block[→α]
              else f:Block[→α] → α {
      c ifTrue t ifFalse f }

   method while c:Block[→Boolean]
              do a:NullaryBlock → void {
      c.apply ifTrue { a.apply; while c do a  }

   method until c:Block[→Boolean]
              do a:NullaryBlock → void {
      while {c.apply.not} do a }

}
```

```
object true := {
   method ifTrue t:Block[→α]
               ifFalse f:Block[→α] → α {
       t.apply } }

object false := {
   method ifTrue t:Block[→α]
               ifFalse f:Block[→α] → α {
       f.apply } }
```

```
object Grace = { ...
   method with c:Collection[ε]
              do a:Block[ε→void]  {
       c do a }

   method with c:Collection[ε]
              map a:Block[ε→α] → Collection[α] {
       c collect a }

   method with c: Collection[ε]
              select a:Block[ε→Boolean]
              → Collection[ε]  {
       c.select a }
}
```

```
class interval = {
   const start:Number
   const stop:Number
   const step:Number

   method do action:Block[Number→void] {
      var element
      var index := 0
      while {index < self size}
         do { element := start + (index × step)
               index := index + 1
               action.apply element  } }

… }
```

What about case?

- Pro
  - Instructors are familiar with case
  - Case is concise
  - Students will meet case in other languages
- Con
  - Unnecessary — just use method dispatch
    - Assume “open classes”
  - Case violates object encapsulation
    - “Tell, don’t ask”

What about case?

- Pro
  - Instructors are familiar with case
  - Case is concise
  - Students will meet case in other languages
- Con
  - Unnecessary — just use method dispatch
    - Assume "open classes"
  - Case violates object encapsulation
    - "Tell, don't ask"

Can we devise a simple, object-oriented dispatch?

What about case?

- Pro
  - Instructors are familiar with case
  - Case is concise
  - Students will meet case in other languages
- Con
  - Unnecessary — just use method dispatch
    - Assume "open classes"
  - Case violates object encapsulation
    - "Tell, don't ask"

Can we devise a simple, object-oriented dispatch?

Should we?

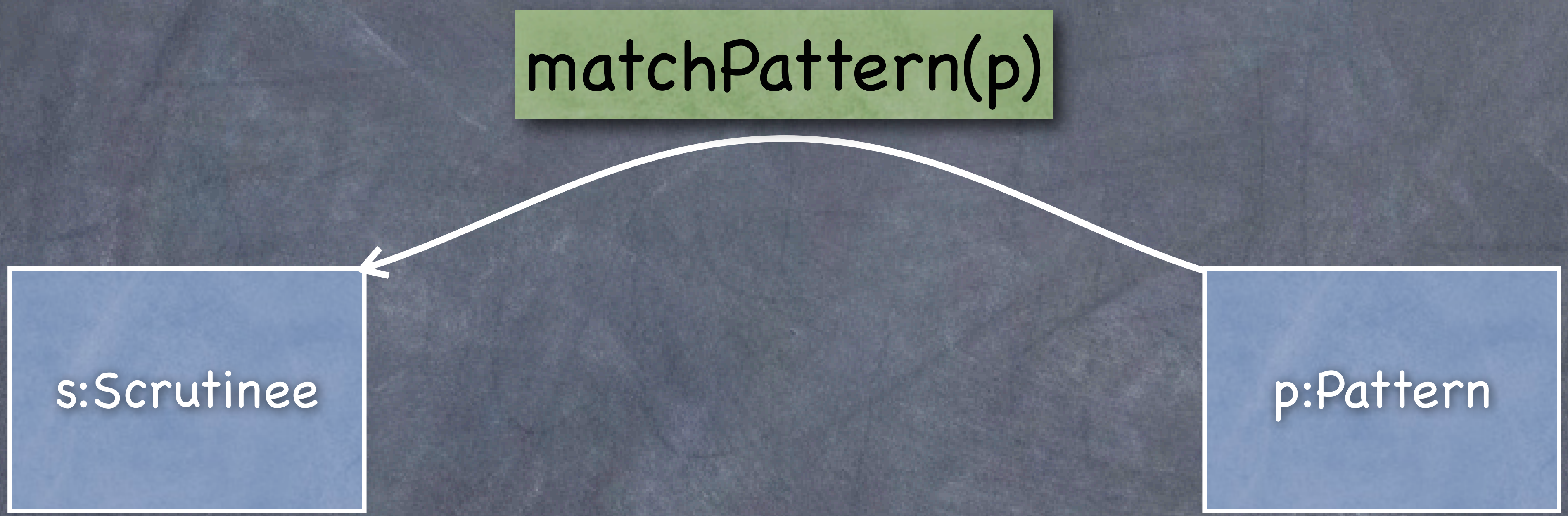
- How can we teach **case** without **case**?
  - Add algebraic types and pattern matching?
  - Adopt Newspeak-style quad-dispatch case?
  - Scala case-classes?

The last 2 men standing…

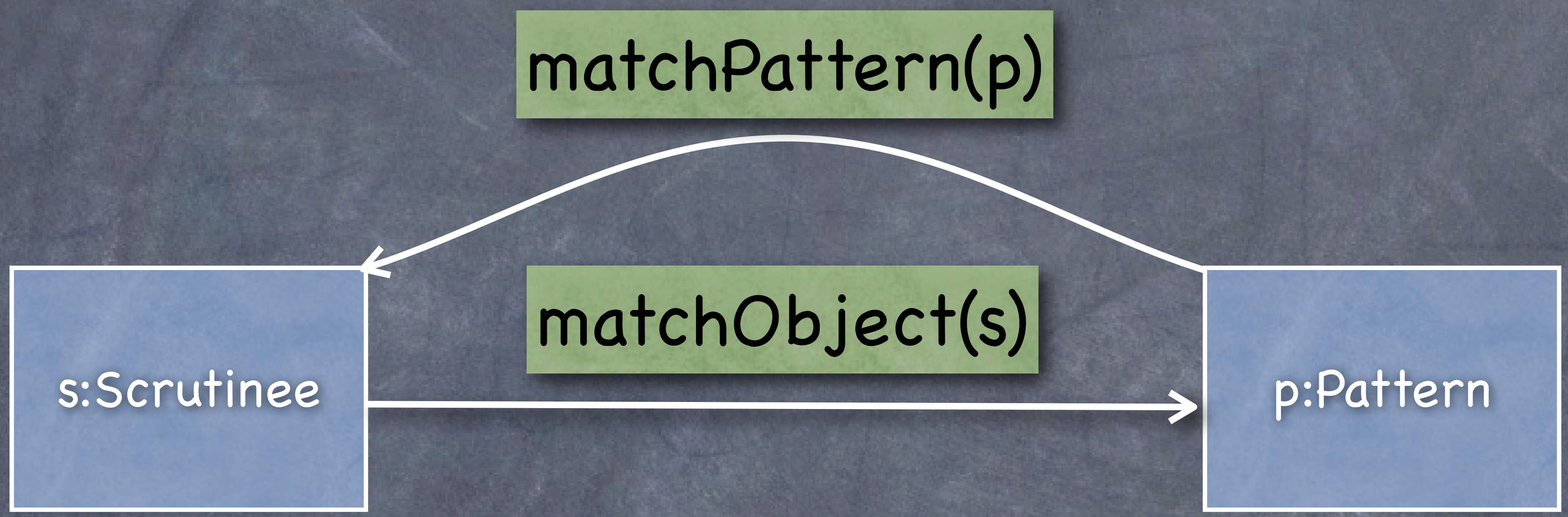
- Pattern-matching through method dispatch (James Noble, via Gilad Bracha)
- Case as object (Andrew Black, via Blume, Acar & Chae)

Pattern-matching through method dispatch

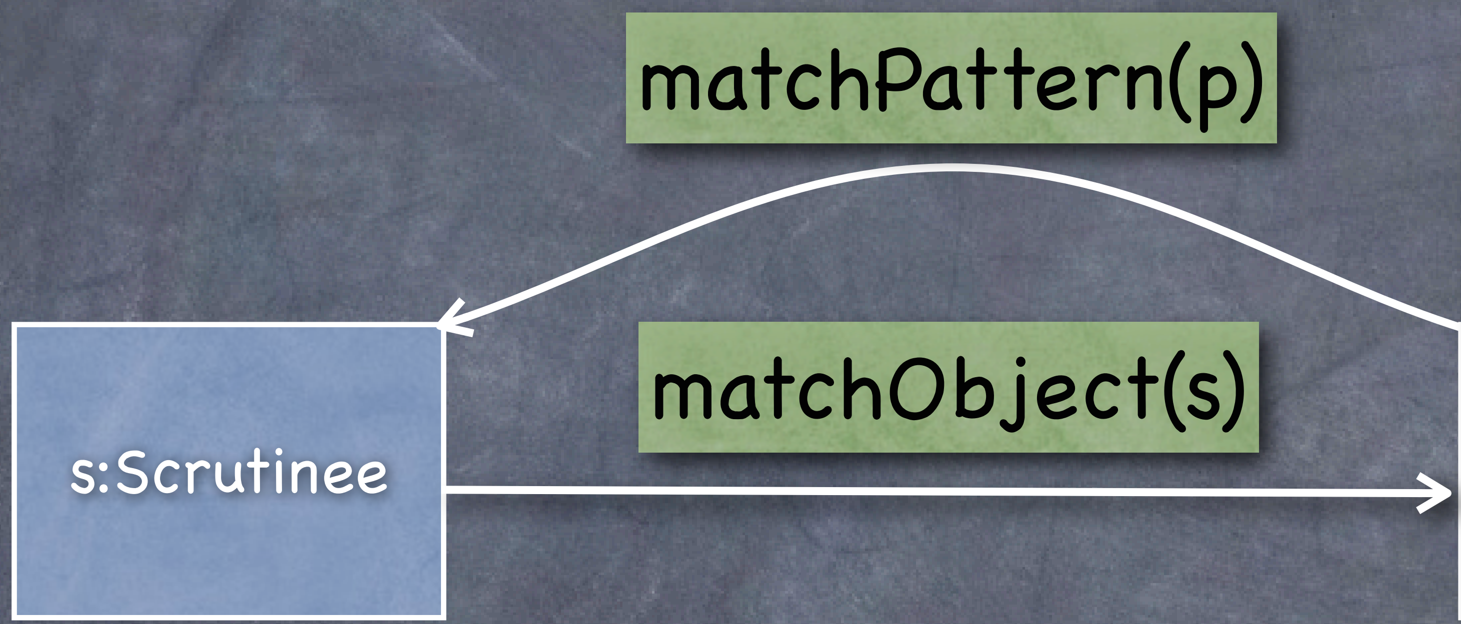
Pattern-matching through method dispatch



Pattern-matching through method dispatch



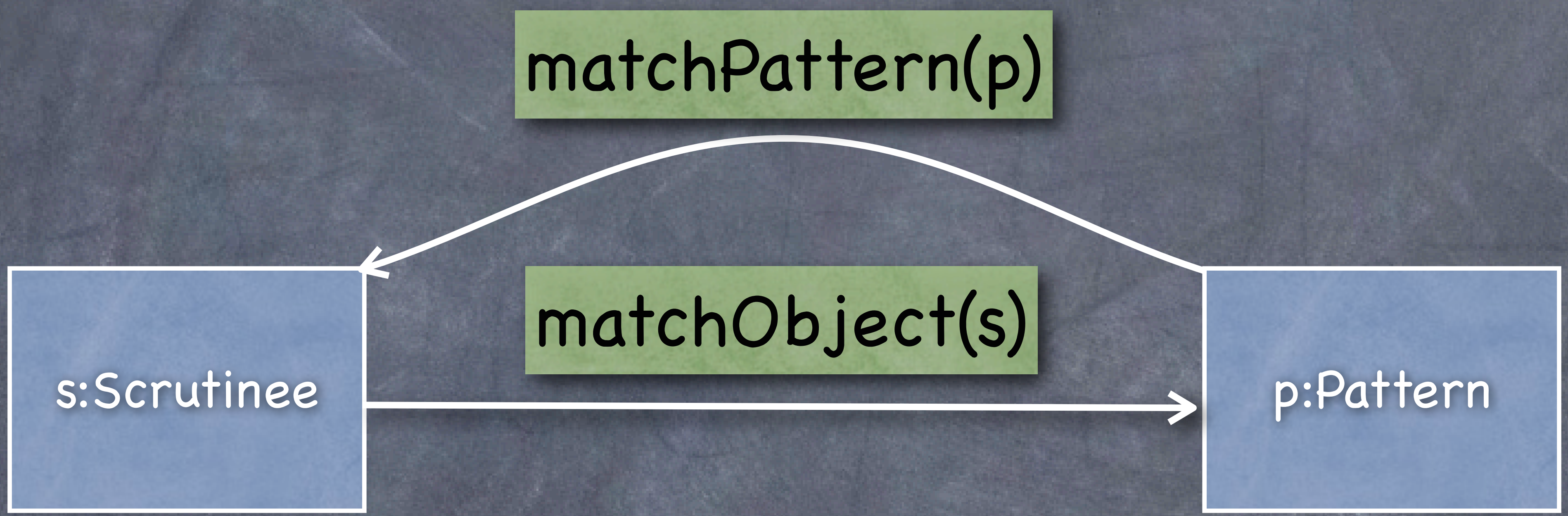
Pattern-matching through method dispatch



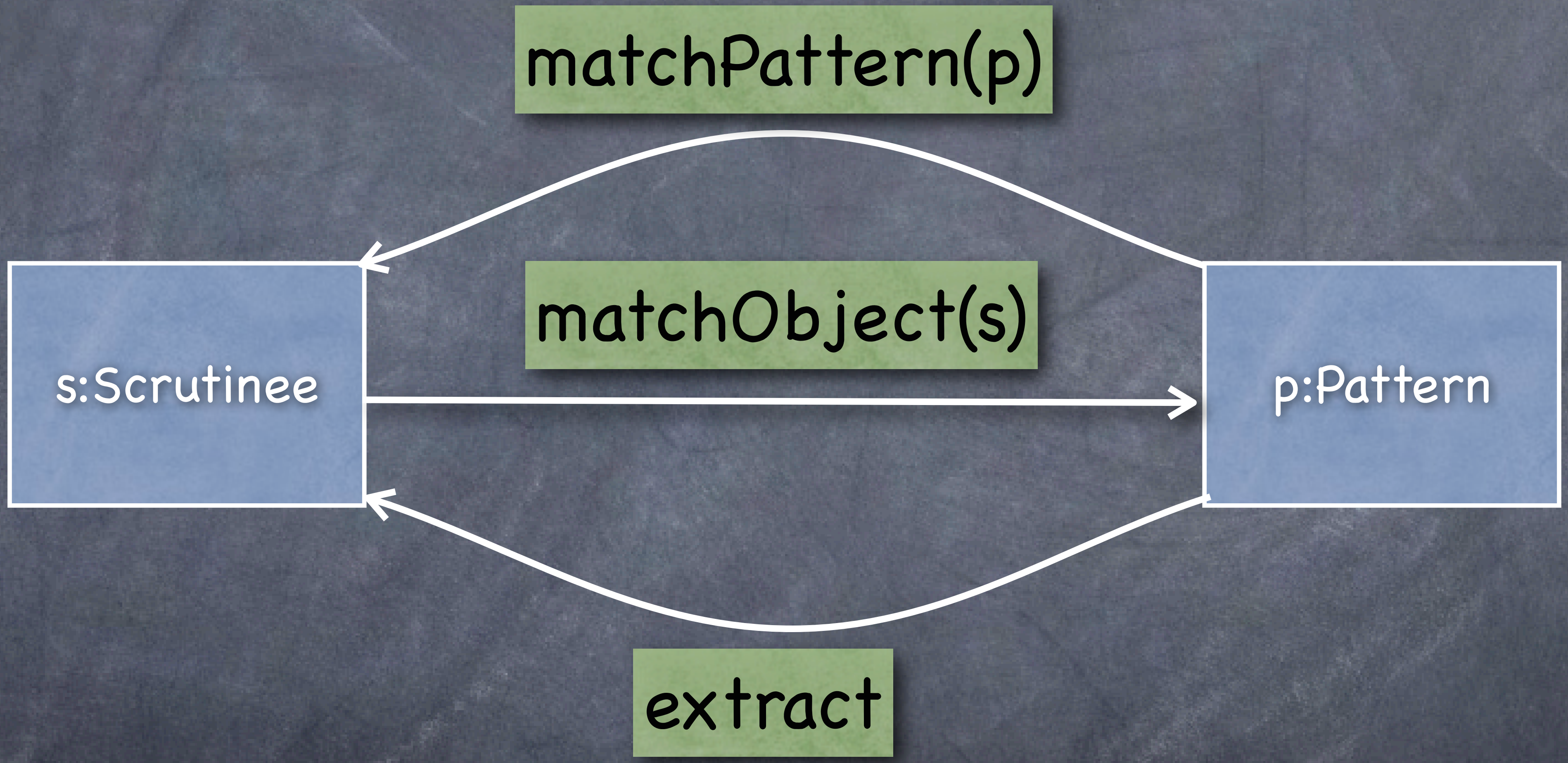
matchObject does different things in different patterns:

- Type patterns ask s for its type
- Literal patterns check for =
- etc

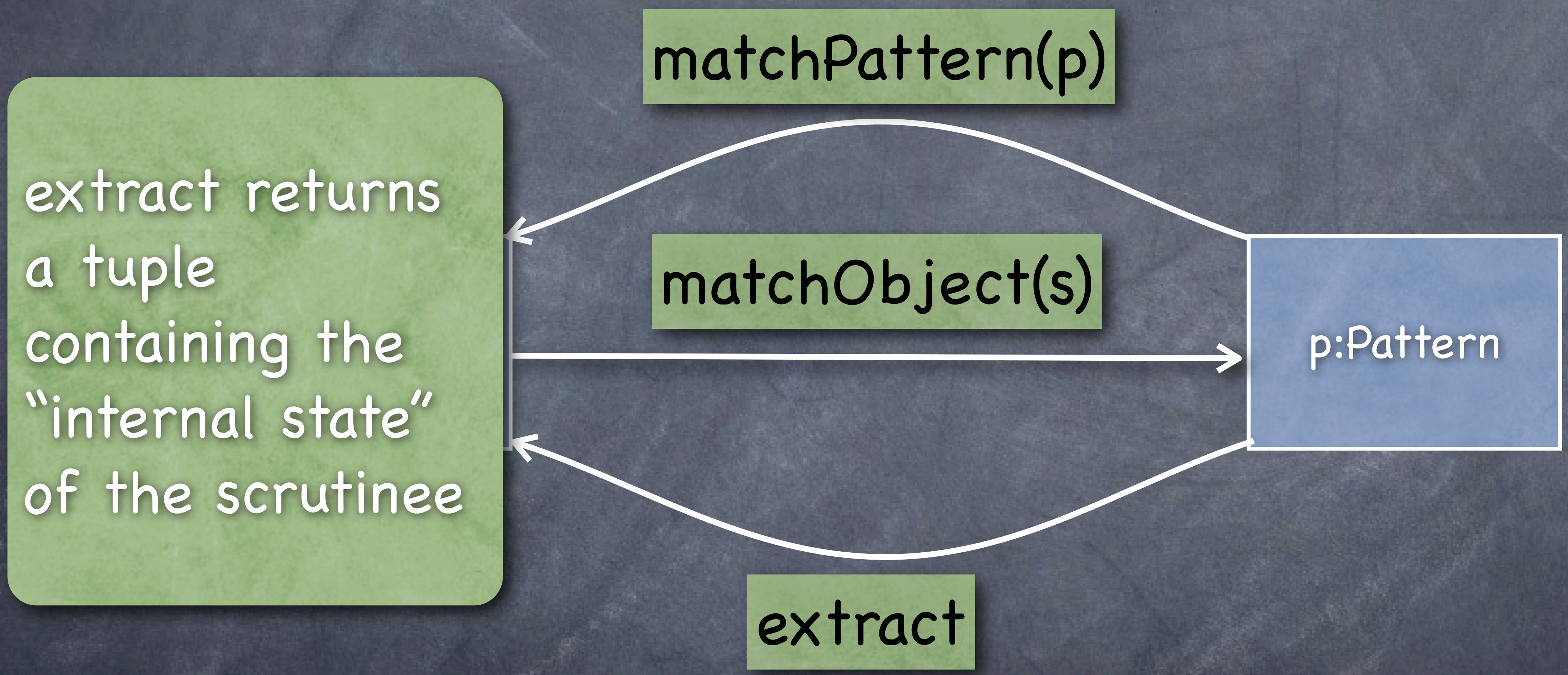
Pattern-matching through method dispatch



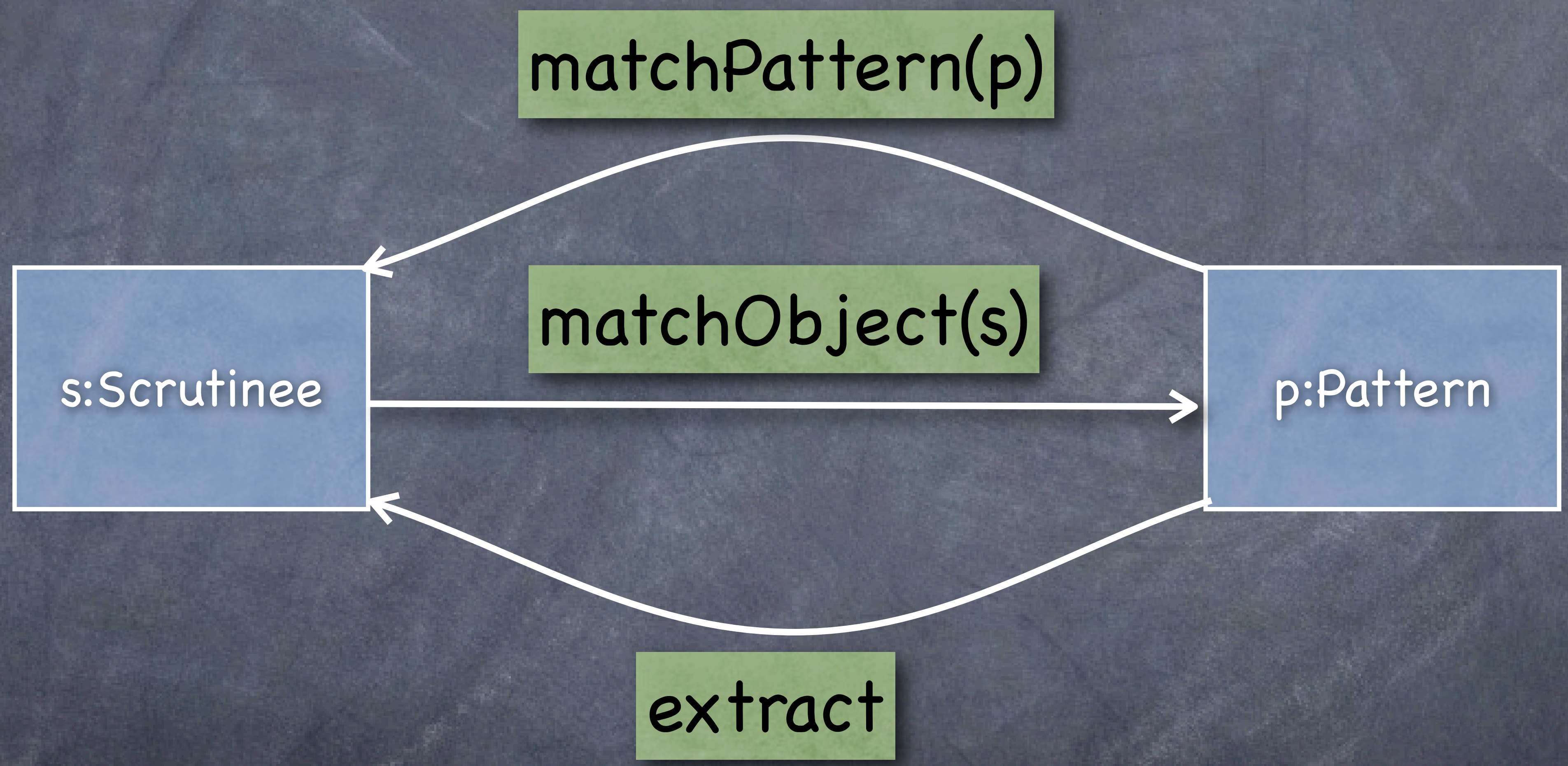
Pattern-matching through method dispatch



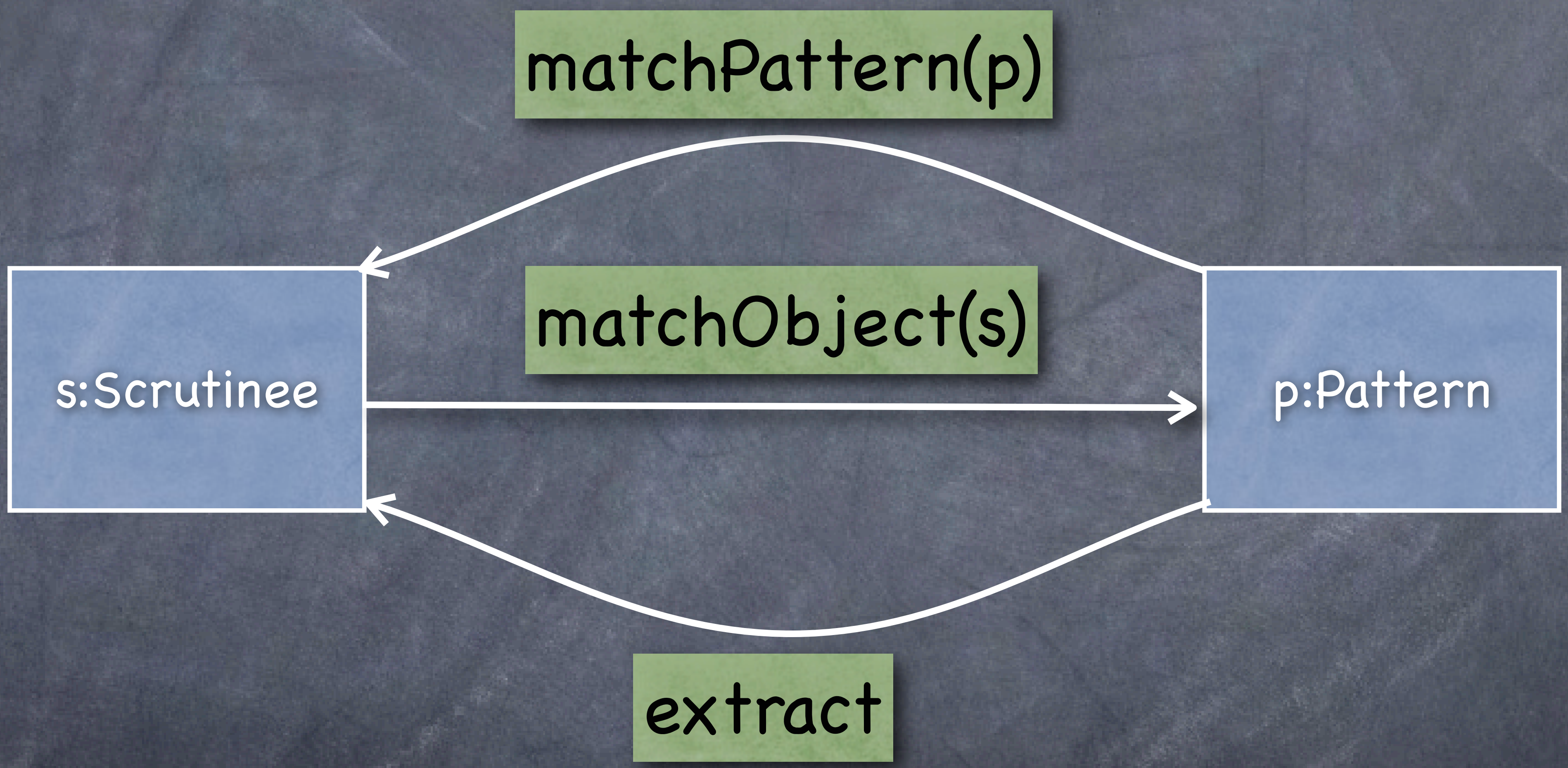
Pattern-matching through method dispatch



Pattern-matching through method dispatch



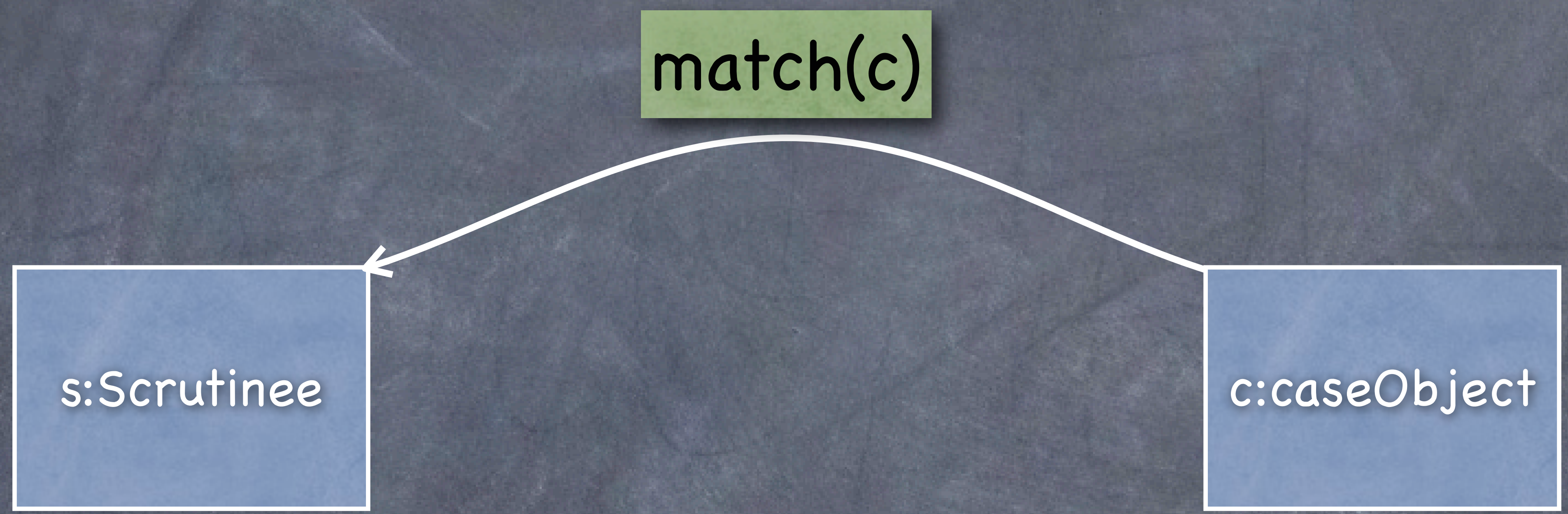
Pattern-matching through method dispatch



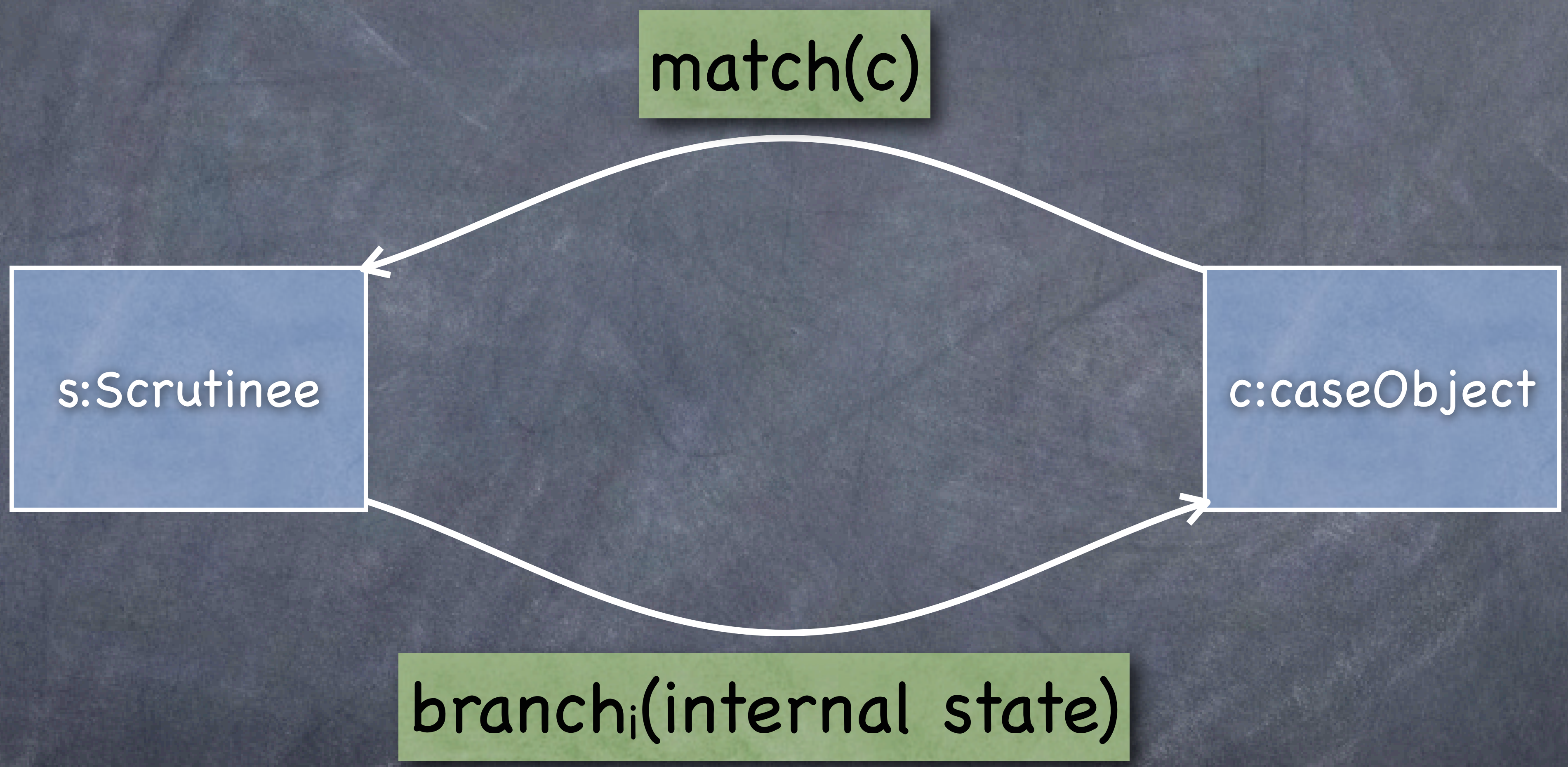
- Treat any lambda-expression as a pattern

Case as Object

Case as Object



Case as Object



Open Issues

- Statements as well as Expressions?
- case statement?

Open Issues

- Statements as well as Expressions?
- case statement?
- details of class syntax

Open Issues

- Statements as well as Expressions?
- case statement?
- details of class syntax

Open Issues

- Statements as well as Expressions?
- case statement?
- details of class syntax

Open Issues

- Statements as well as Expressions?
- case statement?
- details of class syntax

What **have** we decided?

- Types are optional
- Lambdas-expressions are supported
- Extensibility via libraries
- Types (= interfaces) are structural
- Classes define an interface corresponding to the operations on their instances
- Support for immutable objects
- Classes are open

Types are Optional

- This means more than inferring type declarations:
  - "Untyped semantics": types don't change the semantics of correct programs — and not the syntax either!
  - explicit type annotations are assertions
    - just like **assert** s.notEmpty

dynamic and static type-checking: two interpretations of the same program

dynamic and static type-checking: two interpretations of the same program

- The Laissez faire or George W. Bush interpretation:
  - do what you want, we won't try to stop you. If you mess up, the PDIC will bail you out.

dynamic and static type-checking: two interpretations of the same program

- The Laissez faire or George W. Bush interpretation:
  - do what you want, we won't try to stop you. If you mess up, the PDIC will bail you out.

Program debugger and interactive checker

dynamic and static type-checking: two interpretations of the same program

- The Laissez faire or George W. Bush interpretation:
  - do what you want, we won't try to stop you. If you mess up, the PDIC will bail you out.
- The "The Nanny State" or Harold Wilson interpretation.
  - We will look after you. If it is even remotely possible that something may go wrong, we won't let you try.

dynamic and static type-checking: two interpretations of the same program

- The Laissez faire or George W. Bush interpretation:
- The “The Nanny State” or Harold Wilson interpretation.

A third interpretation is useful:

- The Laissez faire or George W. Bush interpretation:
- The "The Nanny State" or Harold Wilson interpretation.
- The "Proceed with caution", or Edward R. Murrow, interpretation.
  - The checker has been unable to prove that there are no type errors in your program. It may work; it may give you a run-time error. Good night and good luck.

Three interpretations

- Under all three interpretations, an error-free program has the same meaning.
  - Under the Wilson interpretation:
    - some error-free programs won't be permitted to run
    - an erroneous program will result in a checked run-time error.

Three interpretations

- Under all three interpretations, an error-free program has the same meaning.
- Under the Bush interpretation, all checks will be performed at runtime.
    - Even those that are guaranteed to fail — because a counter-example is more useful than a type-error message

Three interpretations

- Under all three interpretations, an error-free program has the same meaning.
- Under the Bush interpretation, all checks will be performed at runtime.
- Under the Murrow interpretation, you will get a mix of compile-time <u>warnings</u> and run-time checks.
- Under the Wilson interpretation, you won't be permitted to run a program that <u>might</u> have a type-error

Help!

- Supporters
- Programmers
- Implementers
- Library Writers
- IDE Writers
- Testers
- Teachers
- Students
- Tech Writers
- Textbook Authors
- Blog editors
- Community Builders

Schedule

- 2011: 0.1, 0.2 and 0.5 language releases, hopefully prototype implementations
- 2012 0.8 language spec, some mostly complete implementations
- 2013 0.9 language spec, reference implementation, experimental classroom use
- 2014 1.0 language spec, robust implementations, textbooks, initial adopters for CS1/CS2
- 2015 ready for general adoption?

No conclusions —
we aren't done yet

Questions

Comments

Suggestions

Brickbats