The Object-Oriented Programming Paradigm

Rounding of OOP

Bent Thomsen

SmallTalk -- Everything is an object

> Rule 1. Everything is an object.
> Rule 2. Every object is an instance of a class.
> Rule 3. Every class has a superclass.
> Rule 4. Everything happens by sending messages.
> Rule 5. Method lookup follows the inheritance chain.

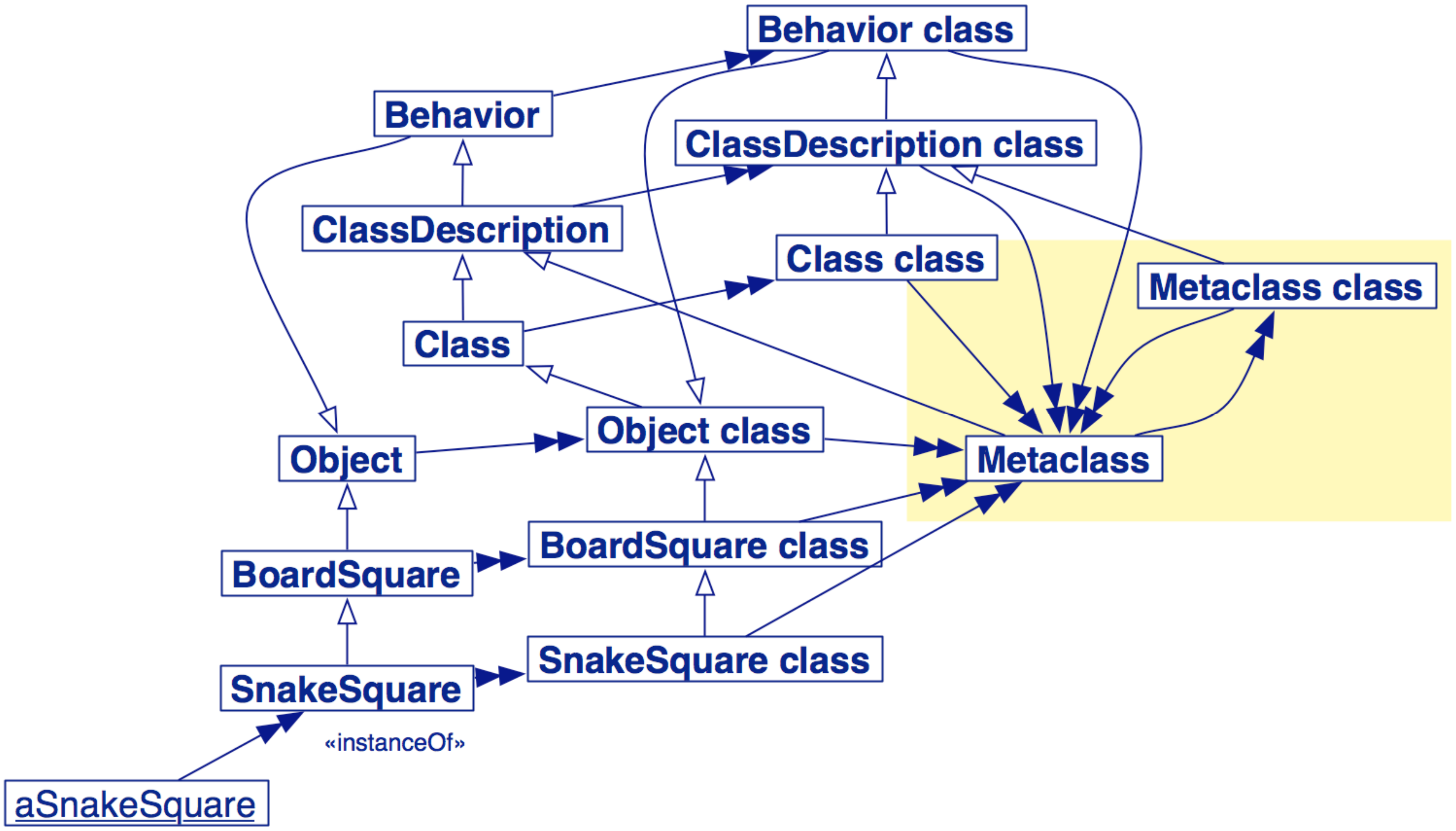
Metaclasses in 7 points

1. Every object is an instance of a class
2. Every class eventually inherits from Object
3. Every class is an instance of a metaclass
4. The metaclass hierarchy parallels the class hierarchy
5. Every metaclass inherits from Class and Behavior
6. Every metaclass is an instance of Metaclass
7. **The metaclass of Metaclass is an instance of Metaclass**

Navigating the metaclass hierarchy

```
3 class ->  SmallInteger
3 class class ->  SmallInteger class
3 class class class ->  Metaclass
3 class class class class ->  Metaclass class
3 class class class class class ->  Metaclass
```

7. The metaclass of Metaclass is an instance of Metaclass



SELF

> Stop the meta-madness!

>

> Prototype-based OOP in Self

SELF - History

- Prototype-based pure object-oriented language.
- Designed by Randall Smith (Xerox PARC) and David Ungar (Stanford University).
  - Successor to Smalltalk-80.
  - “Self: The power of simplicity” appeared at OOPSLA ‘87.
  - Initial implementation done at Stanford; then project shifted to Sun Microsystems Labs; Now on selflanguage.org
  - Vehicle for implementation research
    - *Lots of VM and compiler techniques*

Design Goals

- Conceptual economy
  - Everything is an object
  - Everything done using messages
  - No classes
  - No variables
- Concreteness
  - Objects should seem “real”
  - GUI to manipulate objects directly

Language Overview

- > Dynamically typed
- > Everything is an object
- > All computation via message passing
- > Creation and initialization: clone object
- > Operations on objects:
  - — send messages
  - — add new slots
  - — replace old slots
  - — remove slots

Objects and Slots

Object consists of named slots.

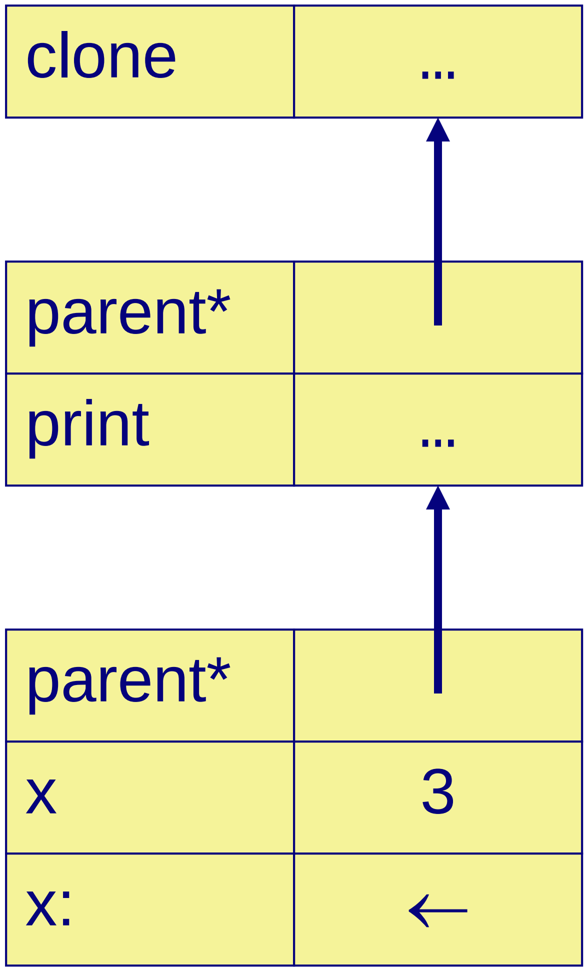
- Data
  - *Such slots return contents upon evaluation; so act like instance variables*
- Assignment
  - *Set the value of associated slot*
- Method
  - *Slot contains Self code*
- Parent
  - *Point to existing object to inherit slots*

Object Creation

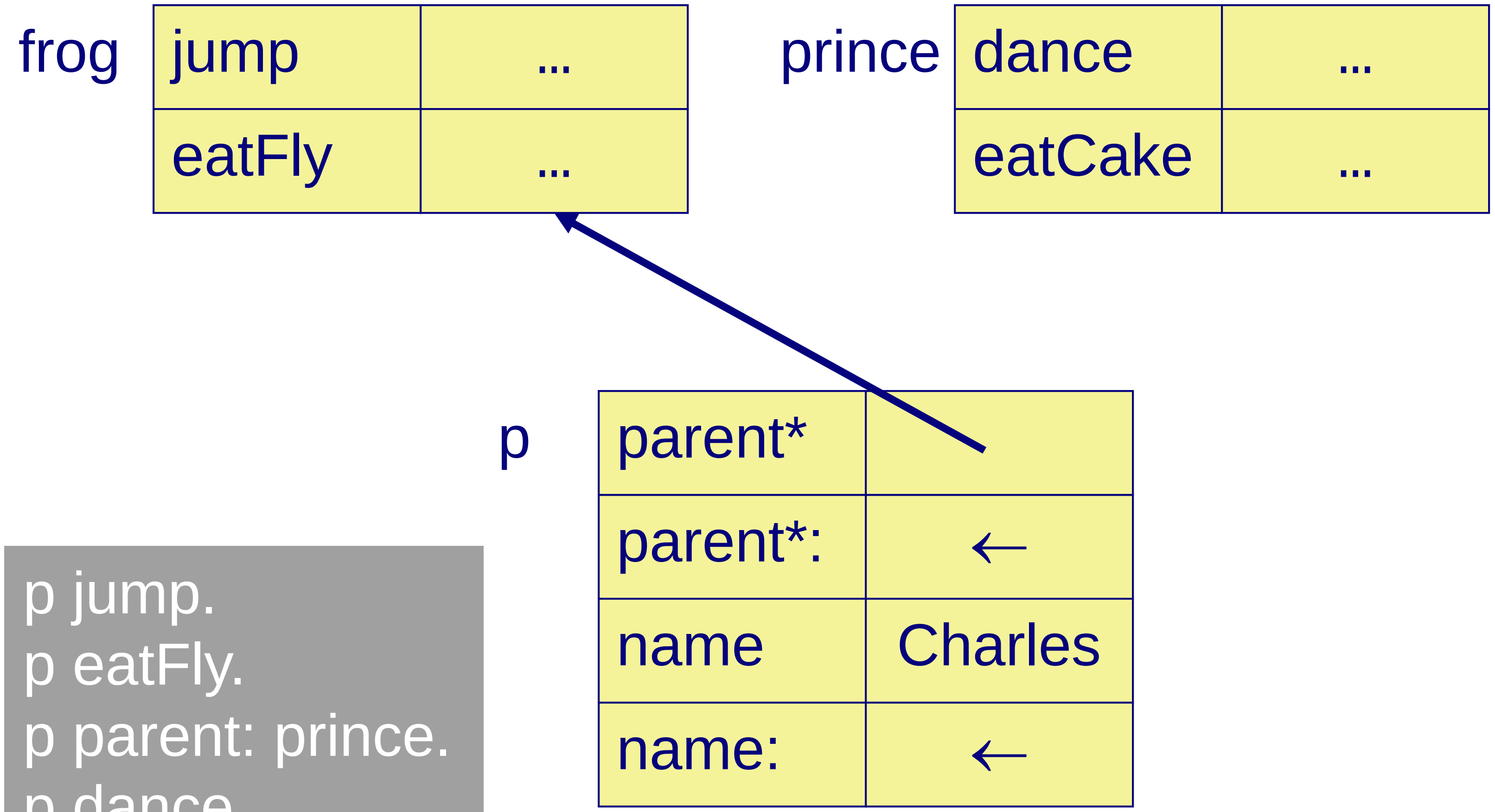
- To create an object, we copy an old one
- We can add new methods, override existing ones, or even remove methods
- These operations also apply to parent slots

Messages and Methods

- When message is sent, object searched for slot with name.
- If none found, all parents are searched.
  - Runtime error if more than one parent has a slot with the same name.
- If slot is found, its contents evaluated and returned.
  - Runtime error if no slot found.

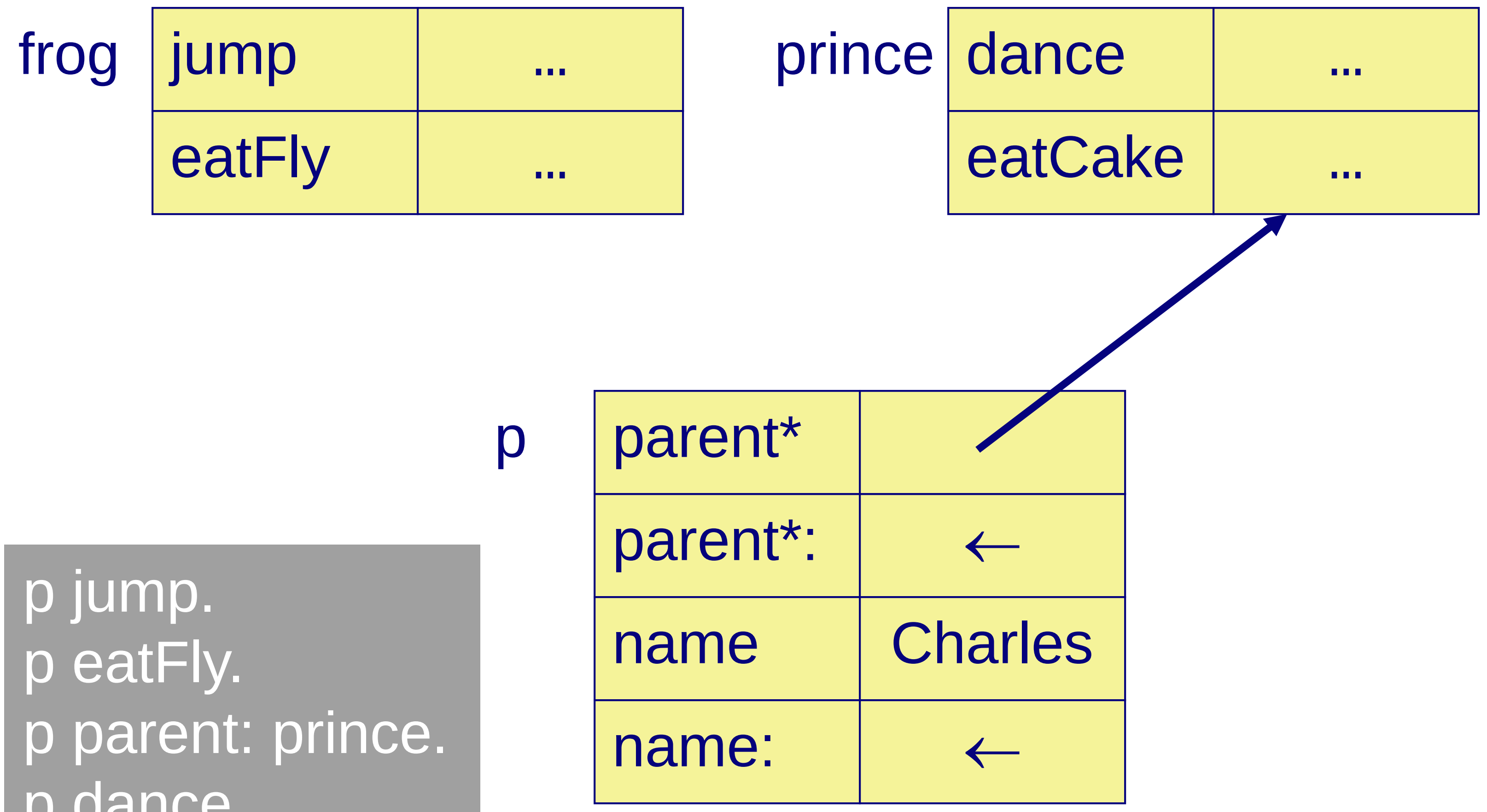


Changing Parent Pointers



```
p jump.
p eatFly.
p parent: prince.
p dance.
```

Changing Parent Pointers



```
p jump.
p eatFly.
p parent: prince.
p dance.
```

Lessons learned from SELF

> Classes require programmers to understand a more complex model.
  - To make a new kind of object, we have to create a new class first.
  - To change an object, we have to change the class.
  - Infinite meta-class regression.

> In Self it is common to structure programs with *traits*: objects that simply collect behavior for sharing.

Class based v.s classless Data Types

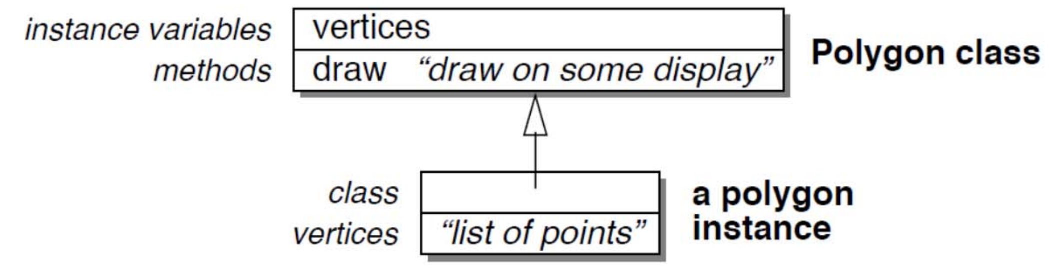


Figure 1a. Data types in a class-based language.

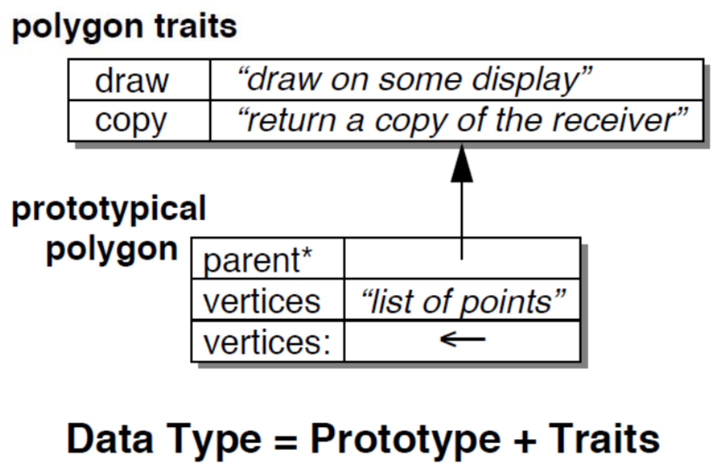
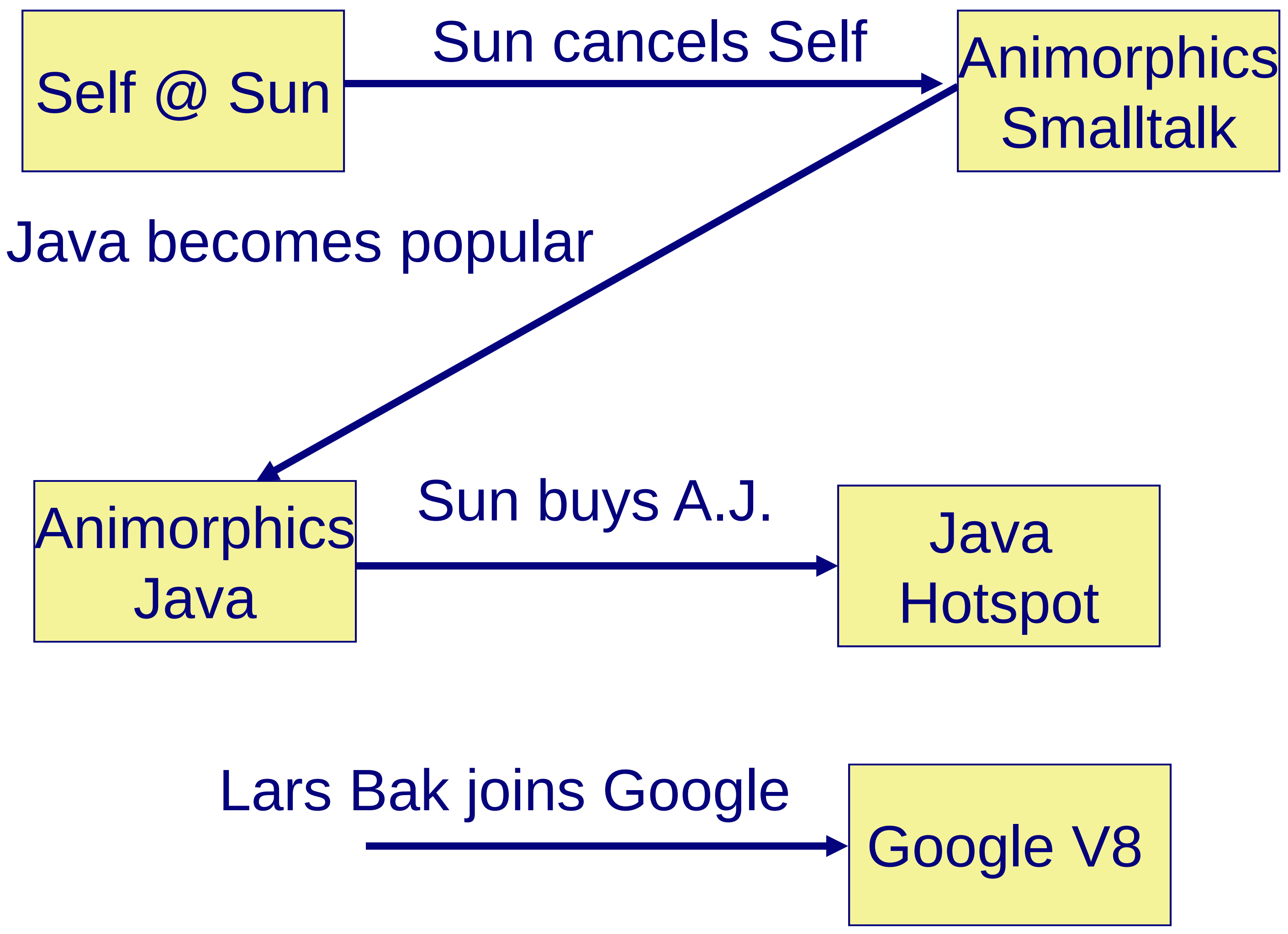

polygon traits
draw "draw on some display"
copy "return a copy of the receiver"
prototypical polygon
parent*
vertices "list of points"
vertices: ←
Data Type = Prototype + Traits

Figure 1b. Data types in a classless language.

Lessons learned from SELF

- But: Does Self require programmer to reinvent structure?
  - Common to structure Self programs with *traits*: objects that simply collect behavior for sharing.
- However: Prototype idea in widespread use: JavaScript
- The idea of traits in Scala (and Fortress and Pharo)

An indirect impact of Self



Ruby

A dynamically typed, class-based, pure OOP language with blocks and reflection

- — Inspired by Smalltalk

> Ruby less simple, but more “modern and useful”

Ruby

- *Pure object-oriented*: *all* values are *objects* (even numbers)
- *Class-based*: Every object has a class that determines behavior
  - *Mixins* (neither Java interfaces nor C++ multiple inheritance)
  - A bit like Traits with state
- *Dynamically typed*
- *Reflection*: Run-time inspection of objects
- Very *dynamic*: Can change classes during execution
- *Blocks* and libraries encourage lots of closure idioms
- Syntax, scoping rules, semantics of a "*scripting language*"
  - Variables "spring to life" on use
  - Very flexible arrays

Defining classes and methods

```
class Name
  def method_name1 method_args1
    expression1
  end
  def method_name2 method_args2
    expression2
  end
  …
end
```

> Method returns its last expression
  — Ruby also has explicit **`return`** statement
> Syntax note: Line breaks often required (else need more syntax), but indentation always only style

Creating and using an object

- **`ClassName.new`** creates a new object whose class is **`ClassName`**

- **`e.m`** evaluates **`e`** to an object and then calls its **`m`** method
  - Also known as "sends the **`m`** message"
  - Can also write **`e.m()`**

- Methods can take arguments, called like **`e.m(e1,…,en)`**
  - Parentheses optional in some places, but recommended

Method visibility

- Three *visibilities* for methods in Ruby:
  - **`private`**: only available to object itself
  - **`protected`**: available only to code in the class or subclasses
  - **`public`**: available to all code
- Methods are **`public`** by default
  - Multiple ways to change a method's visibility
  - Here is one way…

Method visibilities

```
class Foo =
# by default methods public
   …
 protected
# now methods will be protected until
# next visibility keyword
   …
 public
   …

 private
   …
end
```

Subclassing

```
class Point
  attr_accessor  :x, :y
  def initialize(x,y)
    @x = x
    @y = y
  end
  def distFromOrigin
    # direct field access
    Math.sqrt(@x*@x
              + @y*@y)
  end
  def distFromOrigin2
    # use getters
    Math.sqrt(x*x
              + y*y)
  end
end
```

```
class ColorPoint < Point
  attr_accessor  :color
  def initialize(x,y,c)
    super(x,y)
    @color = c
  end
end
```

An object has a class

```
p  = Point.new(0,0)
cp = ColorPoint.new(0,0,"red")
p.class                         # Point
p.class.superclass              # Object
cp.class                        # ColorPoint
cp.class.superclass             # Point
cp.class.superclass.superclass  # Object
cp.is_a? Point                  # true
cp.instance_of? Point           # false
cp.is_a? ColorPoint             # true
cp.instance_of? ColorPoint      # true
o = Object.new                  # Object
o.class                         # Object
```

> Using these methods is usually non-OOP style
  - Disallows other things that "act like a duck"
  - Nonetheless semantics is that an instance of **`ColorPoint`** “is a” **`Point`** but is not an “instance of” **`Point`**
  - Object is the top of the hierarchy

Changing classes

- Ruby programs (or the REPL) can add/change/replace methods while a program is running
- Breaks abstractions and makes programs very difficult to analyze, but it does have plausible uses
  - Simple example: Add a useful helper method to a class you did not define
    - *Controversial in large programs, but may be useful*
- Helps re-enforce “the rules of OOP”
  - Every object has a class
  - A class determines its instances’ behavior

Variables and Symbols

- Ruby is weakly typed. Variables receive their types during assignment.
- There is no boolean type, but everything has a value. False and nil are false and all other objects are true.
- Instance variables begin with the '@' sign.
  - @name, @age, @course
- Class variables begin with two '@' signs.
  - They are almost never used.
- Ruby has symbols (like Smalltalk). They begin with a colon.
  - :name, :age, :course
  - Symbols have a name (string) and value (integer) but no location.

Data Structures

- Arrays
  - Indexed with integers starting at 0.
  - Contents do not have to all be the same type.
  - Contents can be assigned in a list using square brackets.
    - *order = [“blue”, 6, 24.95]*
    - *Arrays are objects so must be instantiated with ‘new’.*
- Hash Tables
  - Key – value pairs
  - Keys are almost always symbols
  - Contents can be assigned in a list of key-value pairs using curly braces.
    - *order = {:color => “blue”, :size => 6, :price => 24.95}*
  - To retrieve an element, use square brackets
    - *@size = order[:size]*

Control Structures: Conditionals

```
if order[:color] == “blue”
          …
elsif order[:size] == 6
     …
else
     …
end
```

Control Structures: Iteration

- for, while and until

```
for item in order do
      puts item
```

- Iterator ‘each’

```
sum = 0
[1..10].each do |count|
   sum += count
end
puts sum
```

— count is a parameter to the block and has no value outside of it.

Exceptions

```
begin
    …
rescue
    …
rescue
    …
ensure
    …
end
```

- *rescue* and *ensure* are the same as *catch* and *finally*
- Ruby also has *throw* and *catch,* similar to Java

Blocks

Blocks are probably Ruby's strangest feature compared to other PLs

But *almost* just closures

- Normal: easy way to pass anonymous functions to methods for all the usual reasons
- Normal: Blocks can take 0 or more arguments
- Normal: Blocks use lexical scope: block body uses environment where block was defined

Examples:

```
3.times { puts "hi" }
[4,6,8].each { puts "hi" }
i = 7
[4,6,8].each {|x| if i > x then puts (x+1) end }
```

Blocks

- If a block consists of a single line, it is enclosed in curly braces.
- Usually blocks begin with a control statement and are terminated with the keyword, ‘end’.
- Indentation, usually two spaces, is used to indicate what is in the block. Common errors are to have either too few or too many ‘ends’.
- Variables within a block are local to the block unless they are instance variables starting with the ‘@’ sign.
- Methods begin with the keyword, ‘def’, and are terminated with an ‘end’.
- Parameters are enclosed with parentheses. If a method has no parameters, the parentheses are optional.

Some strange things

- Can pass 0 or 1 block with *any* message
  - Callee might ignore it
  - Callee might give an error if you do not send one
  - Callee might do different things if you do/don't send one
    - *Also number-of-block-arguments can matter*
- Just put the block "next to" the "other" arguments (if any)
  - Syntax: **`{e}`**, **`{|x| e}`**, **`{|x,y| e}`**, etc. (plus variations)
    - *Can also replace **{** and **}** with **do** and **end***
      - Often preferred for blocks > 1 line

More strangeness

- Callee does not give a name to the (potential) block argument
- Instead, just calls it with **`yield`** or **`yield(args)`**
  - Silly example:

```
def silly a
  (yield a) + (yield 42)
end
```

```
x.silly 5 { |b| b*2 }
```

- *Can ask **`block_given?`** but often just assume a block is given or that a block's presence is implied by other arguments*

Blocks are “second-class”

All a method can do with a block is **`yield`** to it

- — Cannot return it, store it in an object (e.g., for a callback), …
- — But can also turn blocks into real closures
- — Closures are instances of class **`Proc`**
  - – *Called with method **`call`***

This is Ruby, so there are several ways to make **`Proc`** objects ☺

- — One way: Proc.new which takes a block and returns the corresponding **`Proc:`**
  - – *proc_object = Proc.new {puts "I am a proc object"}*
  - – *proc_object.call*

Ruby – a story for another day

- Lots of support for string manipulation and regular expressions
- Popular for server-side web applications
  - Ruby on Rails
- Often many ways to do the same thing
  - More of a “why not add that too?” approach

So Smalltalk really is an improvement on most of it succesors ☺

ASL

- What if we combined the Prototype OO idea from SELF with Smalltalks Classes and a bit of Ruby like syntax?

- Abstraction Step Language (ASL)
  - Create objects directly
  - Add members and methods
  - Clone
  - Weakly categorize
    - *objects may borrow methods from each other*
  - Abstract – extract class from set of objects
  - Generalize – extract superclass from set of classes

  - Future work: What about types?

Types vs. Testing

- > Test driven development is an integral part of SmallTalk programming – rember exercise 3.7 and Sunit !!
- > Test driven development ought to be an integral part of programming in any dynamic language
  - — But often neglected because of lack of support
- > Studies show that test driven development can eliminate (almost) as many errors as a strong type system
  - — But of course testing can only show the presence of errors, not guarantee their absence
  - — And it takes more or less the same time to develop programs using rigorous testing as developing programs in a strongly typed language
- > Nevertheless, long war between dynamic and staticly typed languages ☹

Scala

- Scala is an object-oriented and functional language which is completely interoperable with Java
  - Developed by Martin Odersky, EPFL, Lausanne, Switzerland
- Uniform object model
  - Everything is an object
  - Class based, single inheritance
  - Mixins and traits
  - Singleton objects defined directly
- Higher Order and Anonymous functions with Pattern matching
- Genericity
- Type inference
- Extendible
  - All operators are overloadable, function symbols can be pre-, post- or infix
  - New control structures can be defined without using macros
  - SmallTalk inspired

Scala is Object Oriented

Scala programs interoperate seamlessly with Java class libraries:

- Method calls
- Field accesses
- Class inheritance
- Interface implementation

all work as in Java.

Scala programs compile to JVM bytecodes.

Scala's syntax resemble Java's, but there are a some

```
object Example {
  def main(args: Array[String]) {
    val b = new StringBuilder()
    for (i ← 0 until args.length) {
      if (i > 0) b.append(" ")
      b.append(args(i).toUpperCase)
    }
    Console.println( oString)
  }
}
```

var: Type instead of Type var

Scala's version of the extended **for** loop (use <- as an alias for ←)

Arrays are indexed args(i) instead of args[i]

Scala is functional

The last program can also be written in a completely different style:

- Treat arrays as instances of general sequence abstractions.
- Use higher-order functions instead of lo

map is a method of Array which applies the function on its right to each array element.

```
object Example2
  def main(args:      y[String]) {
    println(args
               map (_.toUpperCase)
               mkString " ")
  }
}
```

A clo   h applies the   to its

mkString is a method of Array which forms a string of all elements with a given separator between them.

Scala’s approach

> Scala applies Tennent’s design principles:
>
> — Concentrate on abstraction and composition capabilities instead of basic language constructs
>
> — Minimal orthogonal set of core language constructs

> But it is European ☹
>
> — But teamed up with Stanford Pervasive Parallelism Laboratory in 2009, so perhaps ☺
>
> — Now used in SPARK/Hadoop for MapReduce analysis of Big Data, claims to be up 100% faster than plain Hadoop

O’CAML

- CAML – french branch of ML
  - Categorical Abstract Machine Language
- Objective CAML
- O’CAML
- Based on record subtyping

```
class point =
  object
    val mutable x = 0
    method get_x = x
    method move d = x <- x + d
  end;;
class point :
  object val mutable x : int method get_x : int method move : int -> unit end
```

F#

- A .NET language (developed by Don Syme)
  - Connects with all Microsoft foundation technologies
  - 3rd official MS language shipped with VS2010
- Aims to combine the best of Lisp, ML, Scheme, Haskell, in the context of .NET
  - Actually based on O'Caml
- Functional, math-oriented, scalable
- Aimed particularly at the "Symbolic Programming" niche at Microsoft
- Has become very popular in the financial sector

F# on one slide

NOTE: type inferred

```
val data: int * int * int
```

```
val sqr: int -> int
```

```
> let data = (1,2,3)

> let sqr x = x * x

> let f (x,y,z) = (sqr x, sqr y, sqr z)

> let sx,sy,sz = f (10,20,30)

> print "hello world"; 1+2

> let show x y z =
>     printf "x = %d y = %d y = %d \n" x y z;
>     let sqrs= f (x,y,z) in
>     print "Hello world\n";
>     sqrs

> let (|>) x f = f x
```

NOTE: parentheses optional on application

NOTE: pattern matching

NOTE: sequencing

NOTE: local binding, sequencing, return

NOTE: pipelining operator

DART

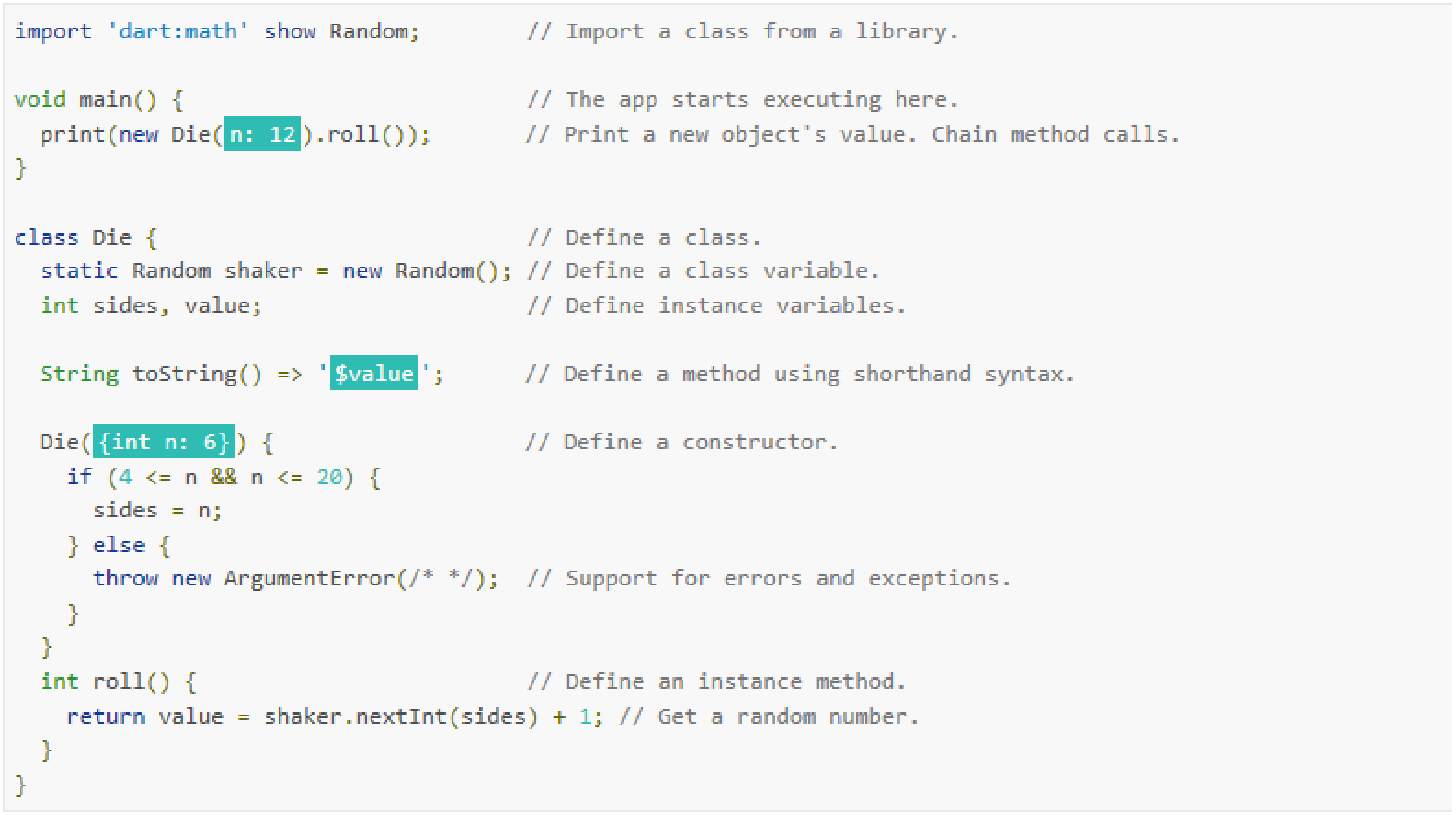
> New OOP/FP language from Google
> Development headed by Lars Bak at Google Århus

> Dynamic, optionally typed, class based OO language
  — Inspired by SmallTalk
> Allows you to write web client and server programs in one language

> Compiles to JavaScript for client side and JIT on server

Dart is new, yet familiar

You can probably already read and even write Dart code. See more samples.



```
import 'dart:math' show Random;         // Import a class from a library.

void main() {                           // The app starts executing here.
  print(new Die(n: 12).roll());         // Print a new object's value. Chain method calls.
}

class Die {                             // Define a class.
  static Random shaker = new Random(); // Define a class variable.
  int sides, value;                     // Define instance variables.

  String toString() => '$value';        // Define a method using shorthand syntax.

  Die({int n: 6}) {                     // Define a constructor.
    if (4 <= n && n <= 20) {
      sides = n;
    } else {
      throw new ArgumentError(/* */);  // Support for errors and exceptions.
    }
  }
  int roll() {                          // Define an instance method.
    return value = shaker.nextInt(sides) + 1; // Get a random number.
  }
}
```

Dart’s comprehensive libraries give you lots of choices

Every app can use lists, sets, and maps, all provided by Dart's core library.

```
Map famousDuos = { 'Han Solo': 'Chewbacca',      // Map literal.
                   'Bonnie': 'Clyde',
                   'Laurel': 'Hardy' };
List myFriends = [ 'Seth', 'Kathy', 'Shailen' ]; // List literal.

// Create lists and maps from Iterable objects.
List shuffledSidekicks = new List.from(famousDuos.values)..shuffle();
Map mixedDuos = new Map.fromIterables(famousDuos.keys, shuffledSidekicks);

// Iteration.
mixedDuos.forEach((k, v) { print('$k, $v'); });

// Some lists, maps, and sets are growable.
Set setOfMyFriends = new Set.from(myFriends);
Set famousPeople = new Set.from(famousDuos.values);
famousPeople.addAll(famousDuos.keys);

// Rich set of functionality for collections.
print(famousPeople.intersection(setOfMyFriends).isEmpty);
print(famousPeople.union(setOfMyFriends).length);
```

DART

- Everything you can place in a variable is an object, and every object is an instance of a class.
  - Even numbers, functions, and null are objects.
  - All objects inherit from the Object class.
- Dart supports top-level functions (such as main()), as well as functions tied to a class or object (static and instance methods, respectively).
  - You can also create functions within functions (nested or local functions).
- Specifying static types clarifies your intent and enables static checking by tools, but it's optional.
  - Dart has two runtime modes: production and checked.
    - *Production mode is the default runtime mode of a Dart program, optimized for speed. Production mode ignores assert statements and static types.*

DART

- Isolates

- Modern web browsers, even on mobile platforms, run on multi-core CPUs. To take advantage of all those cores, developers traditionally use shared-memory threads running concurrently. However, shared-state concurrency is error prone and can lead to complicated code.

- Instead of threads, all Dart code runs inside of isolates. Each isolate has its own memory heap, ensuring that no isolate’s state is accessible from any other isolate.

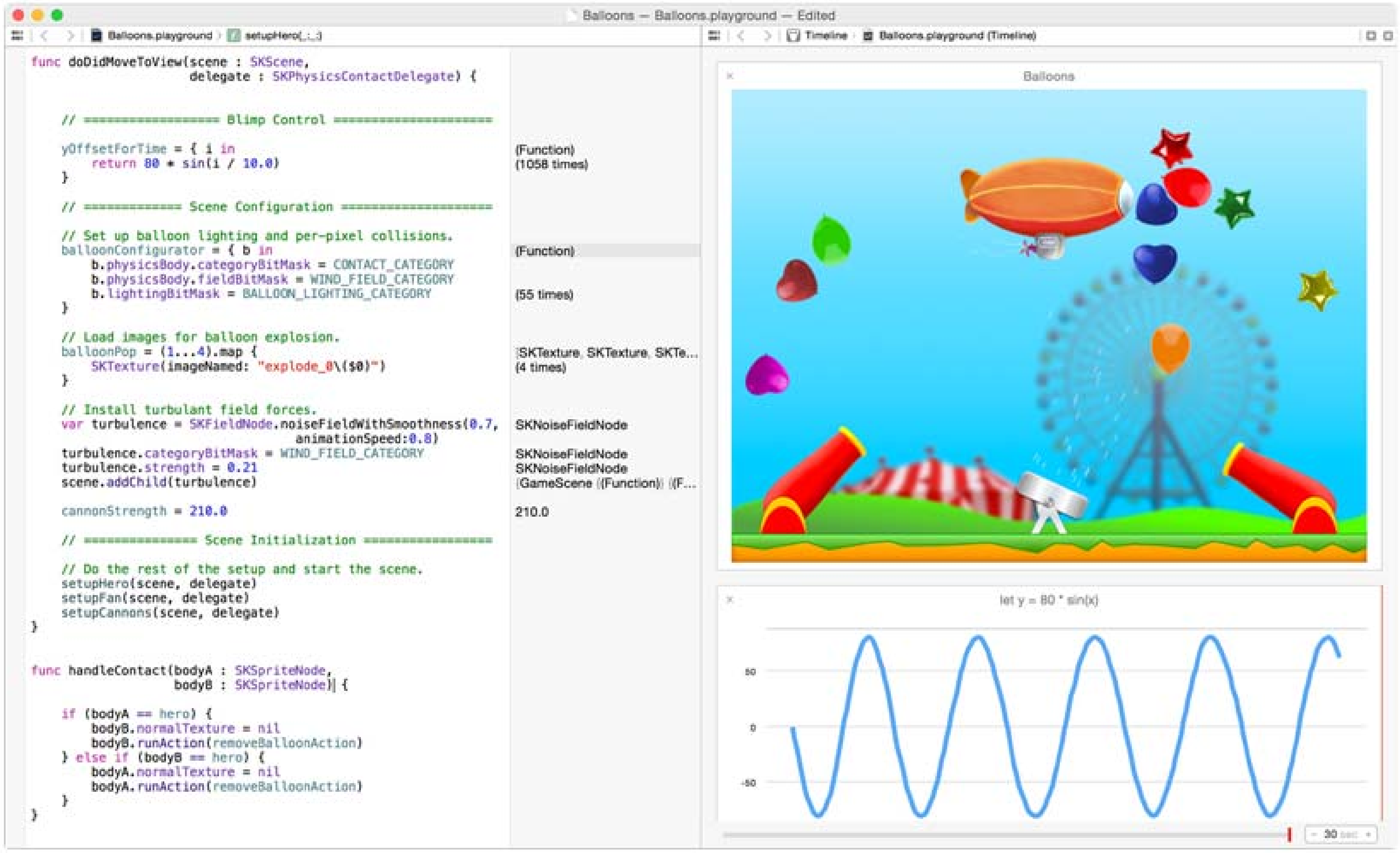
SWIFT

> A new language for iOS and OS X
  - Swift code works side-by-side with Objective-C

> OOP and Functional Language

> Inspired by SmallTalk
  - object-oriented features such as
    - *classes, protocols, and generics*
  - Closures unified with function pointers
  - Tuples and multiple return values
  - Generics
  - Fast and concise iteration over a range or collection
  - Structs that support methods, extensions, protocols.
  - Functional programming patterns, e.g.: map and filter
  - Type inference
  - Garbage collection

REPL interactive development environment



```
func doDidMoveToView(scene : SKScene,
                     delegate : SKPhysicsContactDelegate) {


    // =================== Blimp Control =====================

    yOffsetForTime = { i in
        return 80 * sin(i / 10.0)
    }

    // ============= Scene Configuration ====================

    // Set up balloon lighting and per-pixel collisions.
    balloonConfigurator = { b in
        b.physicsBody.categoryBitMask = CONTACT_CATEGORY
        b.physicsBody.fieldBitMask = WIND_FIELD_CATEGORY
        b.lightingBitMask = BALLOON_LIGHTING_CATEGORY
    }

    // Load images for balloon explosion.
    balloonPop = (1...4).map {
        SKTexture(imageNamed: "explode_0\($0)")
    }

    // Install turbulant field forces.
    var turbulence = SKFieldNode.noiseFieldWithSmoothness(0.7,
                                 animationSpeed:0.8)
    turbulence.categoryBitMask = WIND_FIELD_CATEGORY
    turbulence.strength = 0.21
    scene.addChild(turbulence)

    cannonStrength = 210.0

    // =============== Scene Initialization =================

    // Do the rest of the setup and start the scene.
    setupHero(scene, delegate)
    setupFan(scene, delegate)
    setupCannons(scene, delegate)
}


func handleContact(bodyA : SKSpriteNode,
                   bodyB : SKSpriteNode) {

    if (bodyA == hero) {
        bodyB.normalTexture = nil
        bodyB.runAction(removeBalloonAction)
    } else if (bodyB == hero) {
        bodyA.normalTexture = nil
        bodyA.runAction(removeBalloonAction)
    }
}
```

```
(Function)
(1058 times)

(Function)

(55 times)

[SKTexture, SKTexture, SKTe...
(4 times)

SKNoiseFieldNode

SKNoiseFieldNode
SKNoiseFieldNode
[GameScene [(Function)] [(F...

210.0
```

OOP vs. FP

- In functional (and procedural) programming, break programs down into functions that perform some operation
- In object-oriented programming, break programs down into classes that give behavior to some kind of data
  - These two forms of *decomposition* are so exactly opposite that they are two ways of looking at the same "matrix"
  - Which form is "better" is somewhat personal taste, but also depends on how you expect to *change/extend software*

The expression example

Well-known and compelling example of a common *pattern*:

- Expressions for a small language
- Different variants of expressions: ints, additions, negations, …
- Different operations to perform: **`eval`**, **`toString`**, **`hasZero`**, …

Leads to a matrix (2D-grid) of variants and operations

- Implementation will involve deciding what "should happen" for each entry in the grid *regardless of the PL*

	`eval`	**`toString`**	**`hasZero`**	**…**
`Int`				
`Add`				
`Negate`				
…				

Standard approach in ML

	`eval`	`toString`	`hasZero`	…
`Int`				
`Add`				
`Negate`				
…				

- Define a *datatype*, with one *constructor* for each variant
  - (No need to indicate datatypes if dynamically typed)
- “Fill out the grid” via one function per column
  - Each function has one branch for each column entry
  - Can combine cases (e.g., with wildcard patterns) if multiple entries in column are the same

```
datatype exp =
    Int    of int
  | Negate of exp
  | Add    of exp * exp

exception BadResult of string

(* this helper function is overkill here but will provide a more
   direct contrast with more complicated examples soon *)
fun add_values (v1,v2) =
    case (v1,v2) of
    (Int i, Int j) => Int (i+j)
      | _ => raise BadResult "non-values passed to add_values"

fun eval e = (* no environment because we don't have variables *)
    case e of
    Int _        => e
      | Negate e1    => (case eval e1 of
                Int i => Int (~i)
              | _ => raise BadResult "non-int in negation")
      | Add(e1,e2)   => add_values (eval e1, eval e2)

fun toString e =
    case e of
    Int i        => Int.toString i
      | Negate e1    => "-(" ^ (toString e1) ^ ")"
      | Add(e1,e2)   => "("  ^ (toString e1) ^ " + " ^ (toString e2) ^ ")"

fun hasZero e =
    case e of
    Int i        => i=0
      | Negate e1    => hasZero e1
      | Add(e1,e2)   => (hasZero e1) orelse (hasZero e2)
```

Standard approach in OOP

	`eval`	`toString`	`hasZero`	…
`Int`				
`Add`				
`Negate`				
…				

- Define a *class*, with one *abstract method* for each operation
  - (No need to indicate abstract methods if dynamically typed)
- Define a *subclass* for each variant
- So "fill out the grid" via one class per row with one method implementation for each grid position
  - Can use a method in the superclass if there is a default for multiple entries in a column

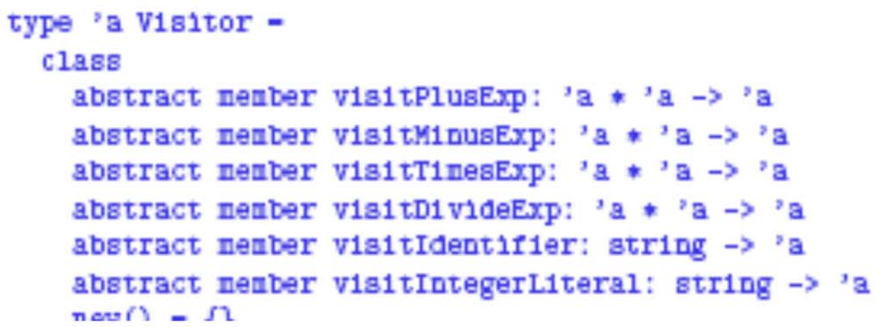
```
class Exp
  # could put default implementations or helper methods here
end

class Value < Exp
  # this is overkill here, but is useful if you have multiple kinds of
  # /values/ in your language that can share methods that do not make sense
  # for non-value expressions
end

class Int < Value
  attr_reader :i
  def initialize i
    @i = i
  end
  def eval # no argument because no environment
    self
  end
  def toString
    @i.to_s
  end
  def hasZero
    i==0
  end
end

class Negate < Exp
  attr_reader :e
  def initialize e
    @e = e
  end
  def eval
    Int.new(-e.eval.i) # error if e.eval has no i method
  end
  def toString
    "-(" + e.toString + ")"
  end
  def hasZero
    e.hasZero
  end
end

class Add < Exp
  attr_reader :e1, :e2
  def initialize(e1,e2)
    @e1 = e1
    @e2 = e2
  end
  def eval
    Int.new(e1.eval.i + e2.eval.i) # error if e1.eval or e2.eval has no i method
  end
  def toString
    "(" + e1.toString + " + " + e2.toString + ")"
  end
  def hasZero
    e1.hasZero || e2.hasZero
  end
end
```

A big course punchline

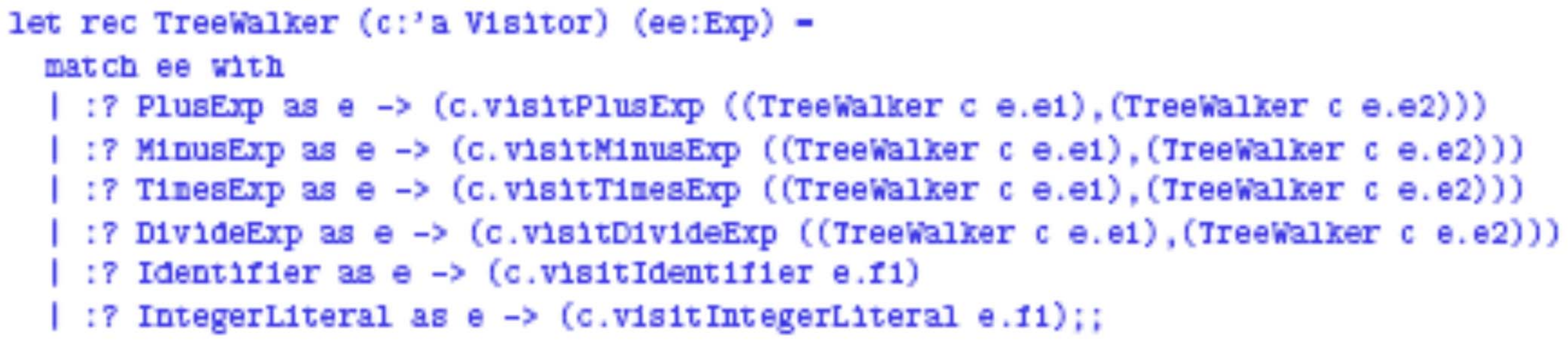
	`eval`	`toString`	`hasZero`	…
`Int`				
`Add`				
`Negate`				
…				

- FP and OOP often doing the same thing in *exact* opposite way
  - Organize the program “by rows” or “by columns”
- Which is “most natural” may depend on what you are doing (e.g., an interpreter vs. a GUI) or personal taste

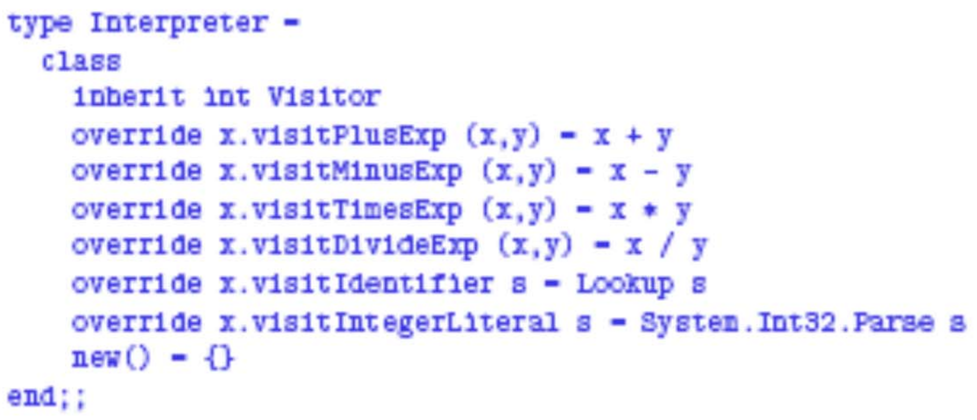
A generalized tree-walker in F#.



```
type 'a Visitor =
  class
    abstract member visitPlusExp: 'a * 'a -> 'a
    abstract member visitMinusExp: 'a * 'a -> 'a
    abstract member visitTimesExp: 'a * 'a -> 'a
    abstract member visitDivideExp: 'a * 'a -> 'a
    abstract member visitIdentifier: string -> 'a
    abstract member visitIntegerLiteral: string -> 'a
    new() = {}
```



```
let rec TreeWalker (c:'a Visitor) (ee:Exp) =
  match ee with
  | :? PlusExp as e -> (c.visitPlusExp ((TreeWalker c e.e1),(TreeWalker c e.e2)))
  | :? MinusExp as e -> (c.visitMinusExp ((TreeWalker c e.e1),(TreeWalker c e.e2)))
  | :? TimesExp as e -> (c.visitTimesExp ((TreeWalker c e.e1),(TreeWalker c e.e2)))
  | :? DivideExp as e -> (c.visitDivideExp ((TreeWalker c e.e1),(TreeWalker c e.e2)))
  | :? Identifier as e -> (c.visitIdentifier e.f1)
  | :? IntegerLiteral as e -> (c.visitIntegerLiteral e.f1);;
```



```
type Interpreter =
  class
    inherit int Visitor
    override x.visitPlusExp (x,y) = x + y
    override x.visitMinusExp (x,y) = x - y
    override x.visitTimesExp (x,y) = x * y
    override x.visitDivideExp (x,y) = x / y
    override x.visitIdentifier s = Lookup s
    override x.visitIntegerLiteral s = System.Int32.Parse s
    new() = {}
end;;
```

Combining OOP and FP

> Mixing OOP and FP brings the best of both worlds together

> See Mapping and Visiting in Functional and Object Oriented Programming paper

Combining OOP and FP

- A step towards more a declarative style of programming
- Say what you want, without saying how

Or as Anders Heilsberg, Inventor of C#, puts it:

”programmers need to talk less about how to do things and more about what they want done and have computers reason it out."

License

http://creativecommons.org/licenses/by-sa/3.0/