# The Self Programming Language

## CS 152 -- Programming Paradigms

## San José State University

**Michael McThrow**

**November 29, 2021**

# Prototype-Based Programming

- In traditional object-oriented programming, classes define objects.

```
public class Point {
  private int x, y;

  public Point(int x, int y) {
    self.x = x;
    self.y = y;
  }

  public void add(Point second) {
    x = second.getX();
    y = second.getY();
  }
}
```

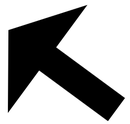
```
Point x = new Point(3,4);
Point y = new Point(5,6);
x.add(y);
```

- Object-oriented programming languages provide inheritance, creating a hierarchy of classes with instance variables and behavior (defined by methods) inherited from parent classes.

```
public class Point {
  private int x, y;

  public Point(int x, int y) {
    self.x = x;
    self.y = y;
  }

  public void add(Point second) {
    x = second.getX();
    y = second.getY();
  }
}
```



GUIPoint inherits Point

```
public class GUIPoint extends Point {
  private Color color;

  public GUIPoint(int x, int y, Color color) {
    super(x, y);
    self.color = color;
  }
}
```

```
Point x = new Point(3,4);
Point y = new Point(5,6);
x.add(y);
```

- A nice advantage of class-based programming languages is that each object has clear definitions regarding its state (e.g., variables) and its behavior (e.g., methods/ messages).

- Class-based programming languages can also be useful for performing compile-time type checks; we don't have to wait until the program is running to perform some type checking (for example, making sure that the object has the method defined).

- However, consider the following situation:
  - What happens if we want to assign a method to a particular object as a one-off special case?
    - In class-based object-oriented programming languages, this necessitates creating a new class that is a subclass of its parent, and then instantiating an object based on that subclass.

- Consider another situation: what if the behavior of a long-running class changes over time?

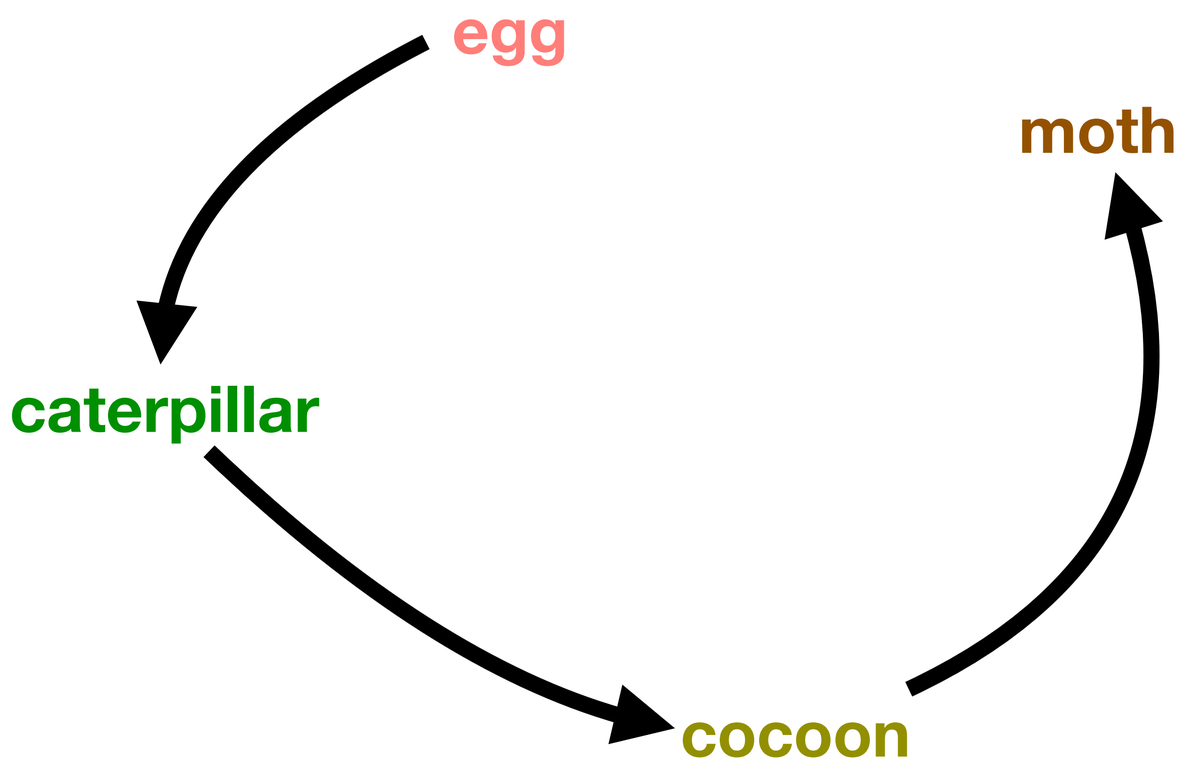
- In an environment where we are describing objects with fixed behavioral characteristics, class-based object-oriented programming seems to be a natural fit.

- Consider modeling situations such as driving a car, or making a transaction at a bank.

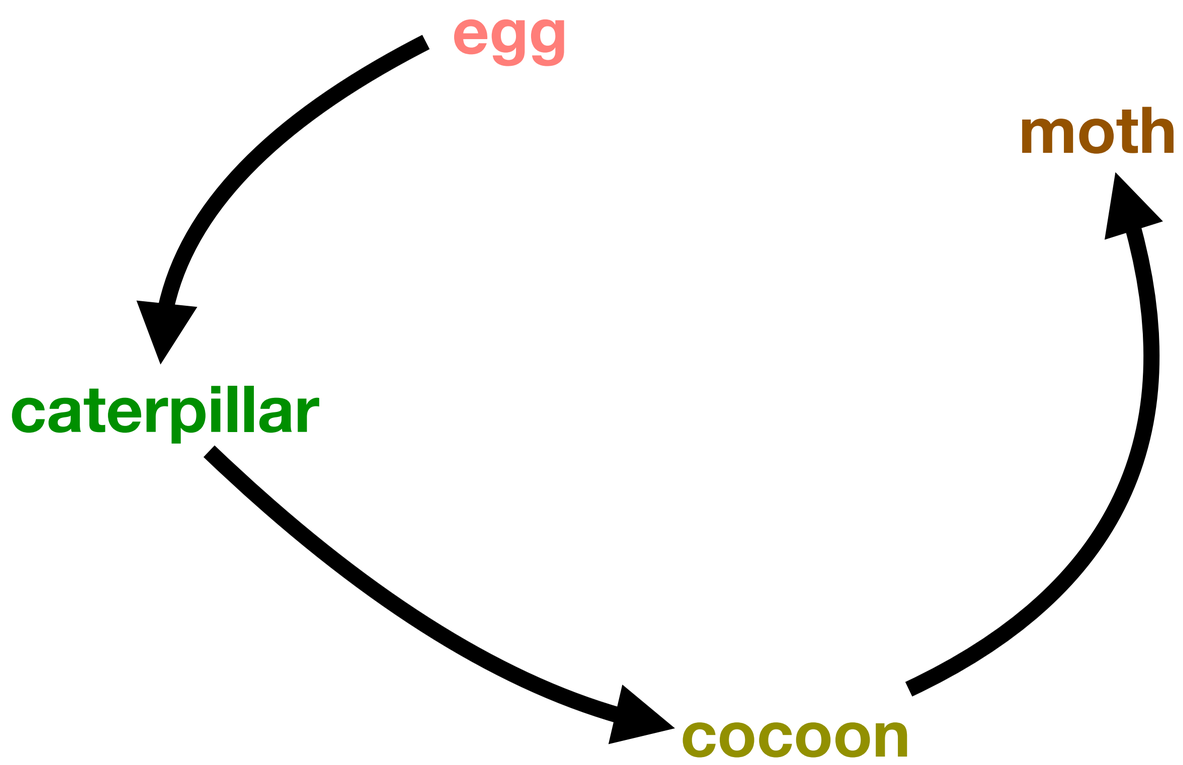
    - Generally, during the lifetime of the object (such a car), its behavior (in terms of the steps needed to carry out a procedure) is most likely not going to change.

However, not all objects behave the same throughout their lifetimes.

# Consider the metamorphosis of an insect.



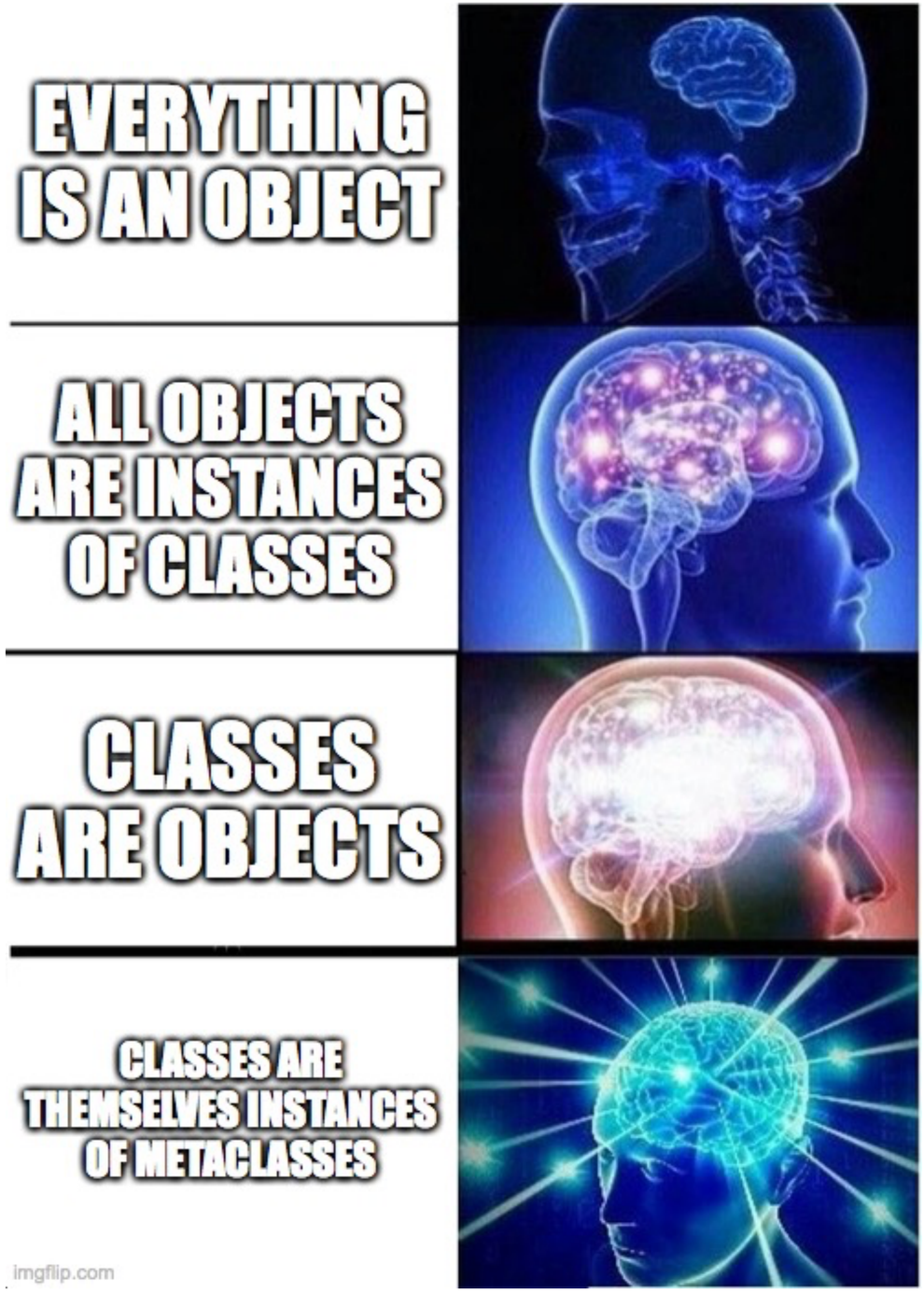
# Consider the metamorphosis of an insect.

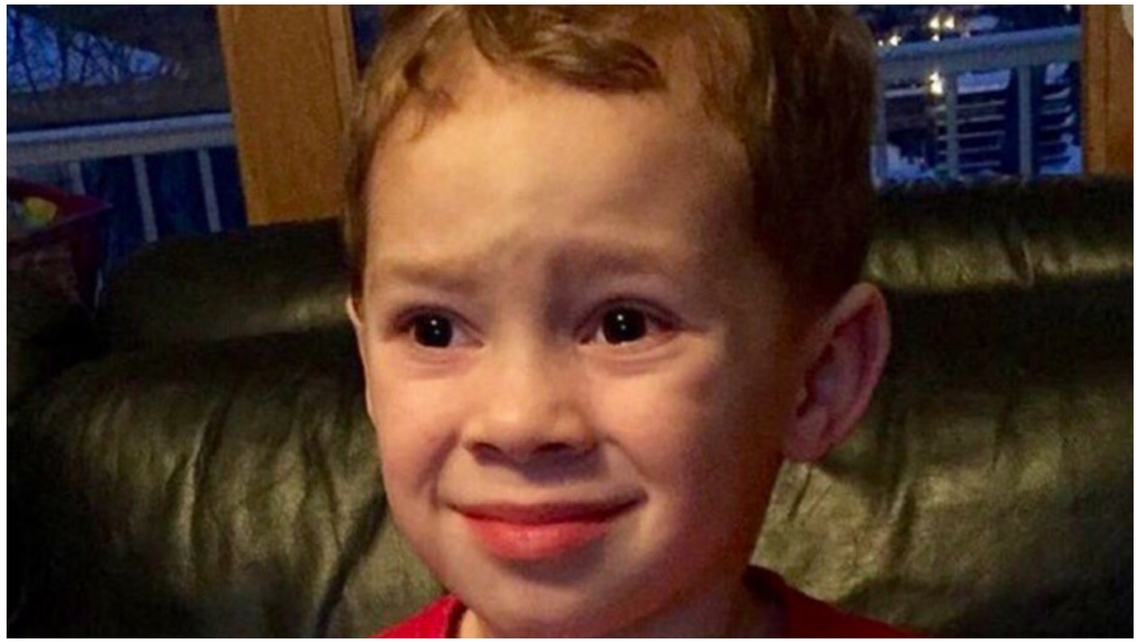



In a live system where objects have long lifetimes, objects could potentially change their behavior and exhibit new behaviors (or lose old behaviors).

# Limitations of Class-Based OO Programming

- In a class-based object-oriented system, how would we deal with new methods added over time?
  - For example, moths can reproduce, but eggs, caterpillars, and cocoons cannot.
- Also, the concept of metaclasses can be hard to grasp for beginners [Borning 1986].



Remember, Smalltalk was originally intended for **children**.

# Alan Kay and Metaclasses in Smalltalk-80

Alan Kay quotes from "The Early History of Smalltalk" [Kay 1993]:

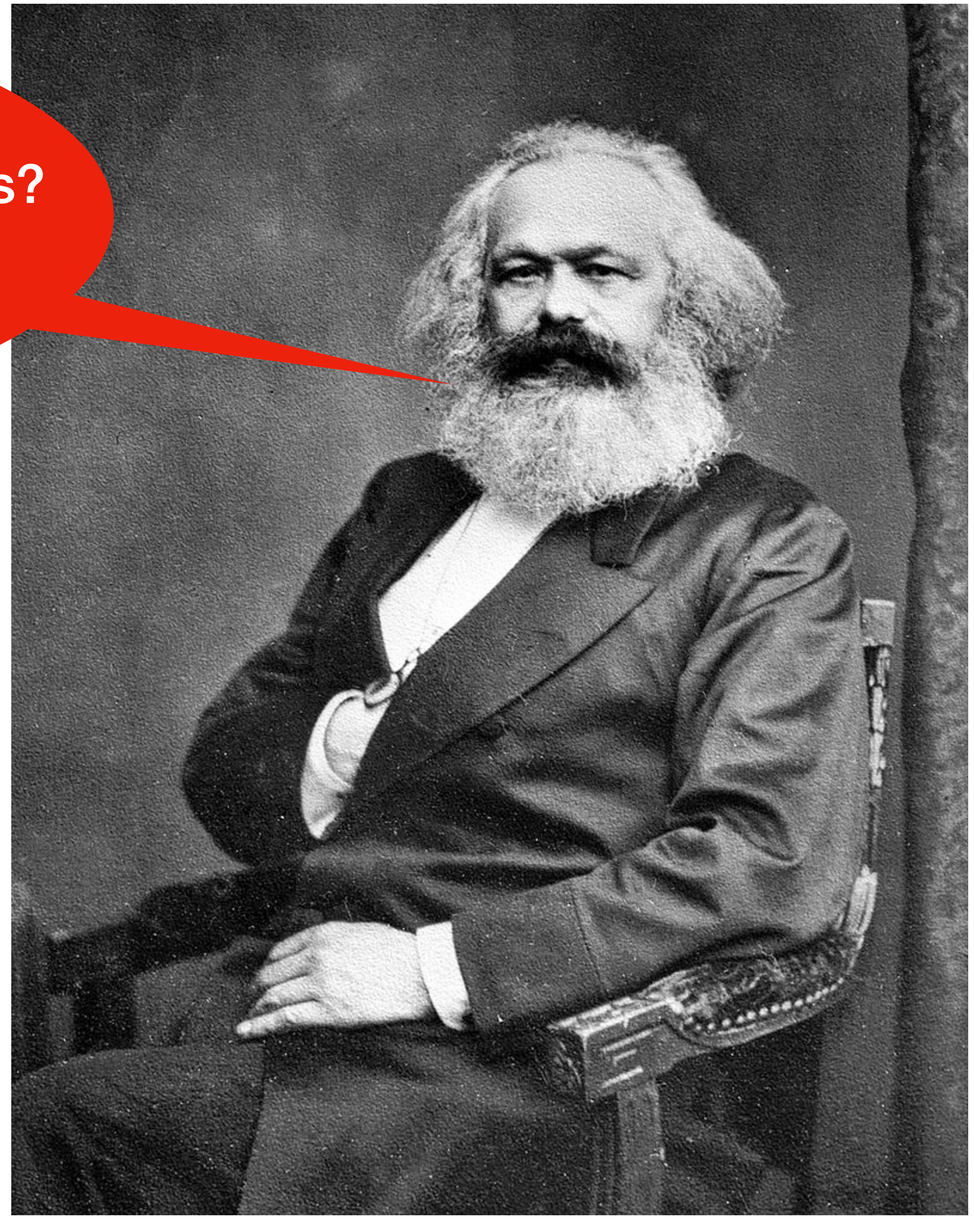
- "The most puzzling strange idea...was the introduction of metaclasses [in Smalltalk-80]".

- "[N]o child had programmed in any Smalltalk since Smalltalk-76 made its debut.  Xerox (and PARC) were now into 'workstations' as things in themselves--but I still wanted 'playstations.'"

One way of dealing with these types of issues is adopting class-less object-oriented programming.

In a sense, class-less object-oriented programming seems to be the logical conclusion of everything being an object, including classes.  If you think about it really hard, are classes truly necessary?

Why are objects defined by their classes?
Why have classes to begin with?



# *Deep Thoughts*

by Karl Marx

**NOTE:** This conversation is historically inaccurate.

It turns out that we don't need classes to do object-oriented programming.


EVERYTHING
IS AN OBJECT
ALL OBJECTS
RE INSTANCES
LASSES
CLASSE
ARE OBJECT
SES ARE
ELVES INSTANCES
OF METACLASSES
imgflip.com

# Self

# The Self Programming Language

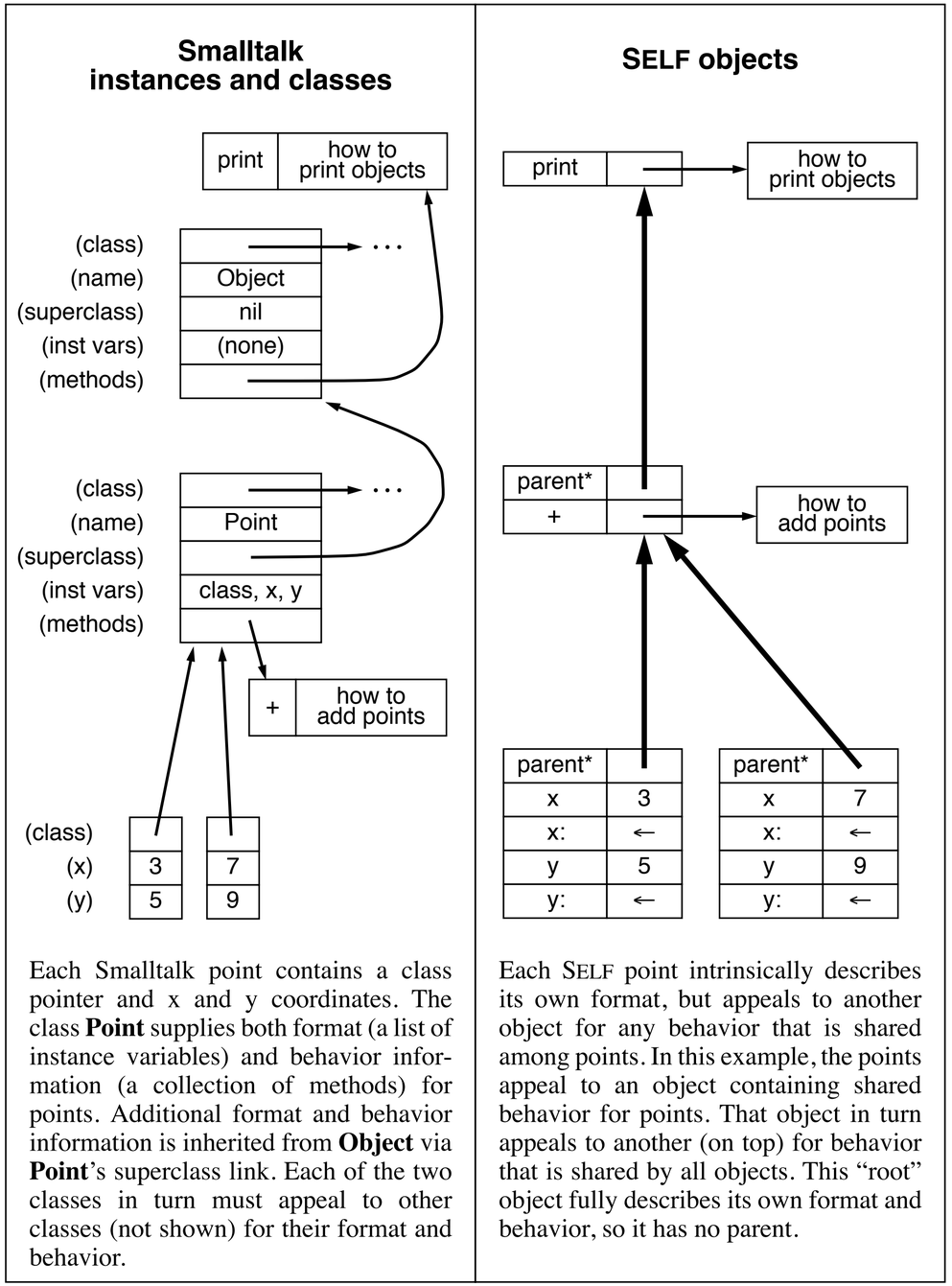
- Created by David Ungar.
  - His PhD thesis, under advisor David Patterson of RISC fame, was on writing a high-performance Smalltalk environment.
  - Self was originally done at Stanford University back when Ungar was a professor, before he left Stanford to go to Sun Microsystems, where he continued working on Self.
- Self is heavily influenced by Smalltalk
  - Syntax is very similar, and everything is an object.
  - Like most traditional Smalltalk implementations, Self is also a self-contained GUI environment.
- Self has no support for classes; this style of object-oriented programming is called prototype-based programming.

# The Self Programming Language

- Objects consists of *slots*.
  - This "slot" terminology is also in the Common Lisp Object System (which I will introduce in the next lecture) and Dylan (a language developed by Apple in the 1990s that is influenced by Lisp but has an Algol-style syntax).
  - Slots store either state (the equivalent of variables) or behavior (methods).
- There are no variables; instead, if an object wants to maintain state, it sends a message to itself (e.g., `self x: 5` creates a slot named `x` and stores the value `5`).
  - This is how Self got its name.
- Instead of constructing objects from class constructors, in Self we *clone* objects by copying other objects.
  - A *prototype* is an object that serves as an "example" for defining behavior.

# Inheritance in Self

- In Smalltalk and other class-based object-oriented programming languages, each object contains a pointer to its class.
- In Self, each object contains a pointer to its parent object.
- To perform inheritance in Self, "[i]f an object receives a message and it has no matching slot, the search continues via a *parent* pointer" [Ungar and Smith 1991, p. 3]

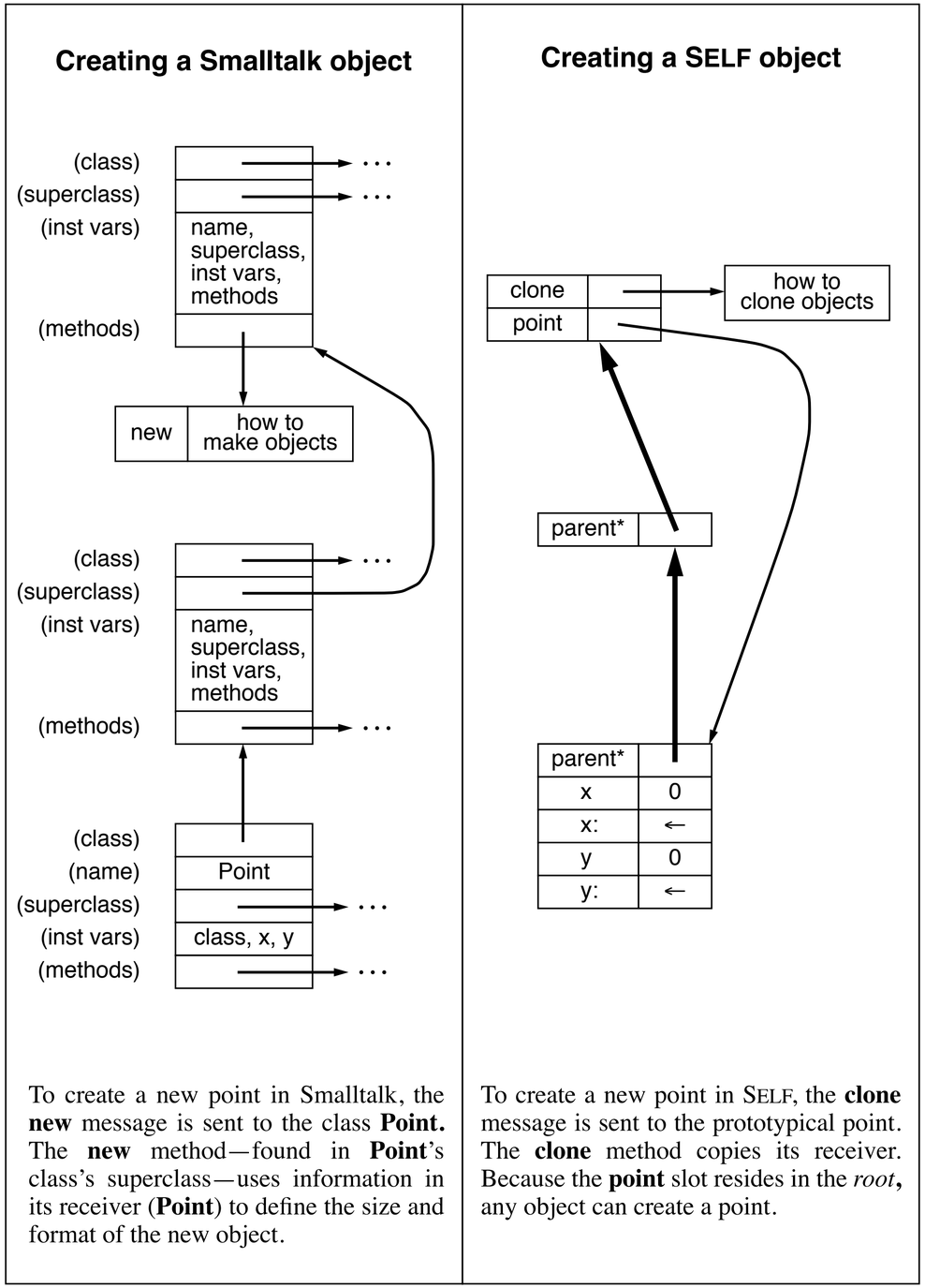



Each Smalltalk point contains a class pointer and x and y coordinates. The class **Point** supplies both format (a list of instance variables) and behavior information (a collection of methods) for points. Additional format and behavior information is inherited from **Object** via **Point**'s superclass link. Each of the two classes in turn must appeal to other classes (not shown) for their format and behavior.

Each SELF point intrinsically describes its own format, but appeals to another object for any behavior that is shared among points. In this example, the points appeal to an object containing shared behavior for points. That object in turn appeals to another (on top) for behavior that is shared by all objects. This "root" object fully describes its own format and behavior, so it has no parent.

Figure 1. A comparison of Smalltalk instances and classes with SELF objects.
At the bottom of each figure are two point objects that have been created by a user program.

Credits: Ungar and Smith 1991, p. 4




To create a new point in Smalltalk, the **new** message is sent to the class **Point.** The **new** method—found in **Point**'s class's superclass—uses information in its receiver (**Point**) to define the size and format of the new object.

To create a new point in SELF, the **clone** message is sent to the prototypical point. The **clone** method copies its receiver. Because the **point** slot resides in the *root*, any object can create a point.

Figure 2. Object creation in Smalltalk and in SELF.

Credits: Ungar and Smith 1991, p. 6

# Other Contributions of Self

- Many researchers who worked on Self helped develop virtual machines and compilers for improving the performance of Self and other dynamically-typed object-oriented programming languages.

- Much of the compiler/VM work for Self, as well as a strongly-typed variant of Smalltalk named Strongtalk, was applied by Sun Microsystems to develop the official Java VM (codenamed HotSpot).

- Research was also done on how to best organize programs without classes (I recommend the 1991 paper "Organizing Programs Without Classes" by Ungar et al.) and also on GUI-driven programming (see the 1995 paper "The Self-4.0 User Interface" by Smith, Maloney, and Ungar).

The language most influenced by Self is JavaScript.

JavaScript is one of the most deployed languages on Earth.

Yet JavaScript is the blunt of many jokes.

# What's 2 + "3" in JavaScript?

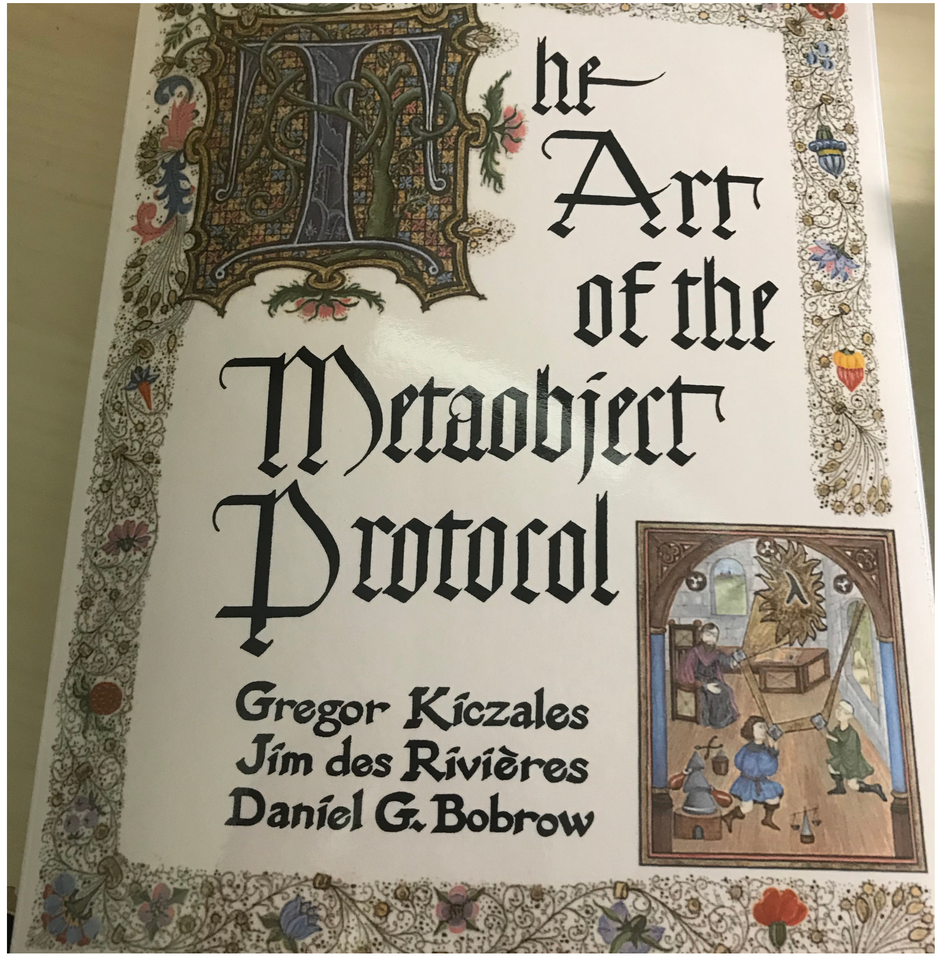

Credit: https://www.reddit.com/r/ProgrammerHumor/comments/621qrt/javascript_the_good_parts/

# Begin Rant

# Next Lecture Preview

## Common Lisp Object System

- Common Lisp is a multi-paradigm programming language that is the descendant of Lisp variants dating all the way back to the original version.
- CLOS has interesting features that are not in Java, JavaScript, Python, and other widely-used OO languages, such as multiple dispatch.
- Yes, CLOS has classes.
- Common Lisp is a highly flexible language. Aside from functions and macros, the metaobject protocol can be used to extend the language, and CLOS is implemented using MOP.




Alan Kay has praised this book. This book is quite a heavy read, however.