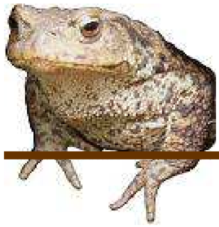
OO History: Simula and Smalltalk

Principles of Software System Construction

Jonathan Aldrich and Charlie Garrod

Fall 2014



Learning Goals

- Know the motivation for, precursors of, and history of objects
- Understand the design of a pure object-oriented language
- Recognize key design patterns used in Smalltalk
  - Including the double dispatch pattern (new)
- Understand the key benefits of objects that drove adoption

Outline

- The beginnings of objects
  - Simulation in Simula 67
  - Demonstration: the first OO language
- Pure OO in Smalltalk
  - Historical context and goals
  - Demonstration: a pure object model
  - Design patterns in Smalltalk
- The benefits and adoption of objects

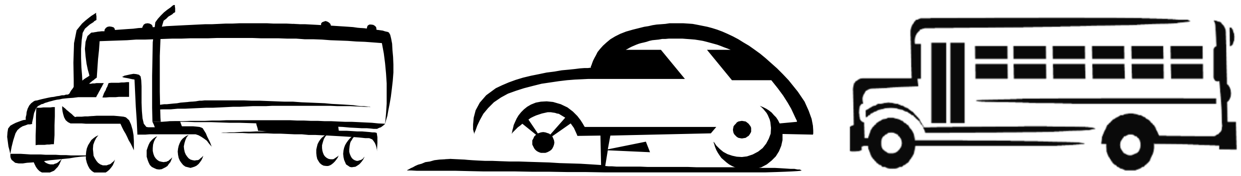
Simulation at the NCC in 1961

- Context: Operations research
  - Goal: to improve decision-making by **simulating complex systems**
    - Discrete-event simulations like Rabbit world, but in domains like traffic analysis
  - Kristin Nygaard and Ole-Johan Dahl at the Norwegian Computing Center

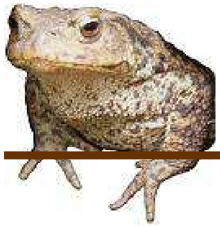
Dahl and Nygaard at the time of Simula's development

- Development of SIMULA I
  - Goal: SIMULA "should be **problem-oriented** and not computer-oriented, even if this implies an appreciable increase in the amount of work which has to be done by the computer.“
  - Modeled simulations “as a variable **collection of interacting processes**"
  - Design approach: "Instead of deriving language constructs from discussions of the described systems combined with implementation considerations, **we developed model system properties** suitable for portraying discrete event systems, considered the implementation possibilities, **and then settled the language constructs**."

SIMULA: a Motivating Problem



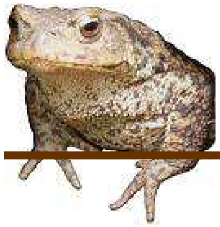
- Need to store vehicles in a toll booth queue.
- Want to store vehicles in a linked list to represent the queue
- Each vehicle is either a car, a truck, or a bus.
- Different kinds of vehicles interact with the toll booth in different ways



Needs Motivating OOP

- Issues with SIMULA I
  - Since each object in a simulation was a process, it was awkward to get **attributes** of other objects
  - "We had seen many useful applications of the process concept to represent **collections of variables and procedures**, which functioned as natural units of programming" motivating more direct support for this
  - "When writing simulation programs we had observed that processes often **shared a number of common properties**, both in data attributes and actions, but were structurally different in other respects so that they had to be described by separate declarations."
  - "**memory space** [was] our most serious bottleneck for large scale simulation."

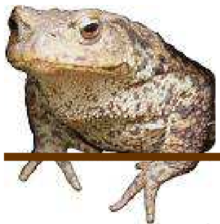
[source: Kristen Nygaard and Ole-Johan Dahl, The Development of the SIMULA Languages, History of Programming Languages Conference, 1978]



Needs Motivating OOP

- Issues with SIMULA I
  - Since each object in a simulation was a process, it was awkward to get **attributes** of other objects
  - "We had seen many useful applications of the process concept to represent **collections of variables and procedures**, which functioned as natural units of programming" motivating more direct support for this
  - "When writing simulation programs we had observed that processes often **shared a number of common properties**, both in data attributes and actions, but were structurally different in other respects so that they had to be described by separate declarations."
  - "**memory space** [was] our most serious bottleneck for large scale simulation."

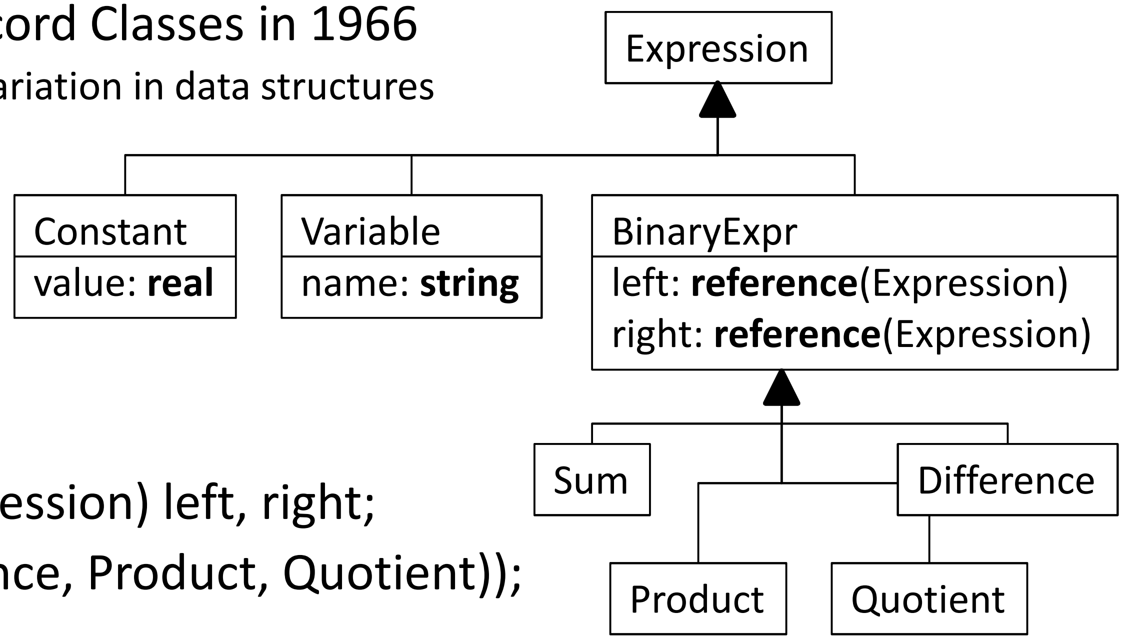
Garbage collection was a good technology for the memory problem. The others required new ideas.



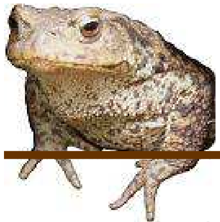
Hoare’s Record Classes

- C. A. R. Hoare proposed Record Classes in 1966
  - Goal: capture similarity and variation in data structures

record class Expression (
    subclasses
        Constant(**real** value),
        Variable(**string** name),
        BinaryExpr(**reference**(Expression) left, right;
            subclasses Sum, Difference, Product, Quotient));

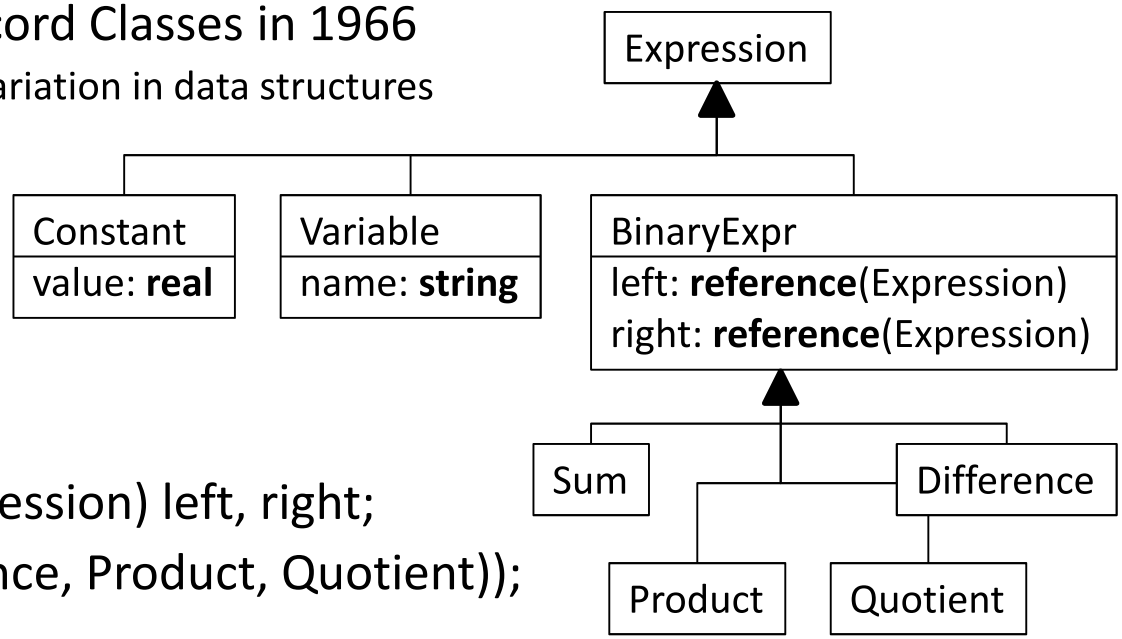


- Each **class** described a particular record structure
- A **subclass** shared fields from its parent
- Variables could take any type in the subclass hierarchy
- A **record class discriminator** provided case analysis on the record type



Hoare's Record Classes

- C. A. R. Hoare proposed Record Classes in 1966
  - Goal: capture similarity and variation in data structures

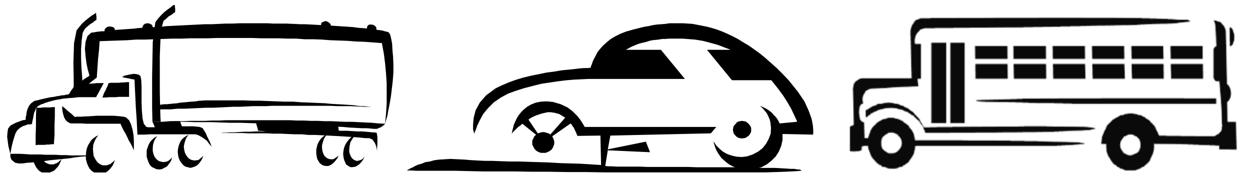


record class Expression (
  subclasses
    Constant(**real** value),
    Variable(**string** name),
    BinaryExpr(**reference**(Expression) left, right;
      subclasses Sum, Difference, Product, Quotient));

Dahl and Nygaard's observations on record classes:

- "We needed subclasses of processes with...actions...not only of pure data records"
- "We also needed to group together common process properties in such a way that they could be applied <u>later</u>, in a variety of different situations not necessarily known in advance"

SIMULA 67's Class Prefix Idea



- Create a Link class to represent the linked list
- Add the Link class as a **prefix** to vehicles, which are subclasses
  - Today we would say this is not a good design—but it nevertheless was enough to motivate a good idea
- As in Hoare's design, subclassing is hierarchical
  - Car, Truck, etc. are subclasses of Vehicle
- Unlike Hoare's classes, Simula classes can have **virtual procedures**
  - Allows subclasses to override behavior for the toll booth
- Unlike in Hoare's design, each class was **declared separately**
  - Link could be reused for other linked lists, not just lists of vehicles
  - Supports extensibility: can add RVs later as a subclass of Vehicle

Hello World in Simula (Demo)

```
begin
    OutText("Hello, world!");
    OutImage
end
```

writes text to the current **image** (line) being created

writes the current image to standard output

Simulating Vehicles (Demo)

```
begin
  class Vehicle;
    virtual: procedure sound is procedure sound;;
  begin
  end;

  Vehicle class Car;
  begin
    procedure sound;
    begin
      OutText("Beep beep!");
      OutImage;
    end;
  end;

  Vehicle class Bike;
  begin
    procedure sound;
    begin
      OutText("Ding ding!");
      OutImage;
    end;
  end;
```

virtual methods can be overridden in subclasses (equiv. of non-**final** in Java)

overriding the sound method in a subclass

Car and Bike are subtypes of Vehicle

Dispatches to code in the car and bike

A size 2 array of references to Vehicles

```
  ref (Vehicle) array vehicles (1 : 2);
  Integer i;
  vehicles (1) :- new Car;
  vehicles (2) :- new Bike;
  for i := 1 step 1 until 2 do
    vehicles(i).sound
end;
```

Co-routines (Demo)

```
begin
   ref (Car) aCar;
   ref (Truck) aTruck;

   class Car;
   begin
      Integer N;

      detach;
      for N := 1 step 1 until 10 do
      begin
         OutText("Driving me insane!");
         OutImage;
         resume(aTruck);
      end;
   end;
```

each class has code that runs when objects are created

we immediately suspend execution until set up is done

continue the care simulation

let the truck take a step in the simulation

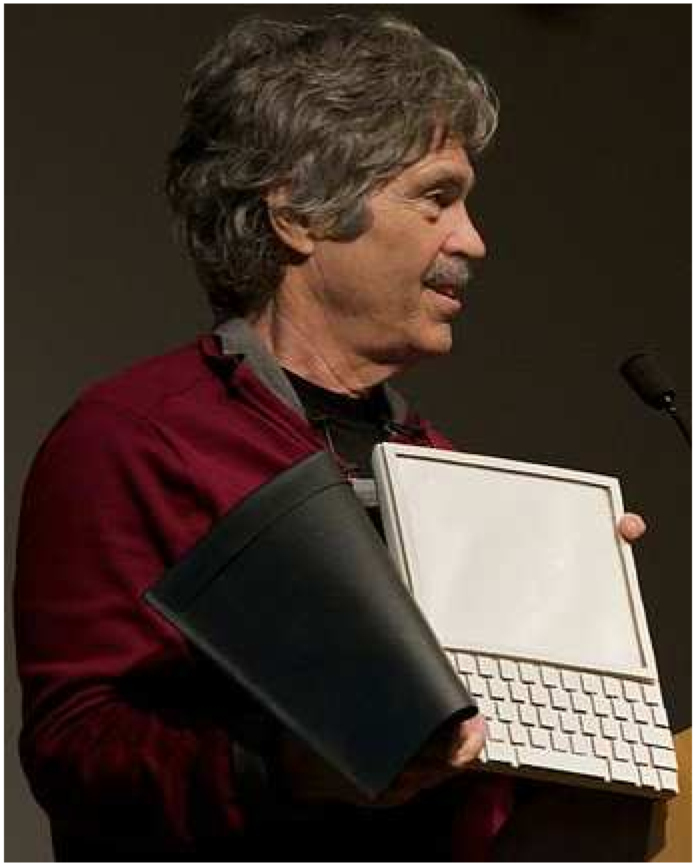
```
   class Truck;
   begin
      Integer N;

      detach;
      for N := 1 step 1 until 10 do
      begin
         OutText("Keep on truckin'!");
         OutImage;
         resume(aCar);
      end;
   end;

   aCar :- new Car;
   aTruck :- new Truck;
   resume(aCar);
end;
```

create the car and truck

start the simulation with the car

Smalltalk Context: Personal Computing

- The Dynabook at Xerox PARC:
  "A Personal Computer for Children of All Ages"
- Funded by US Govt (ARPA, the folks who brought you the internet) to facilitate portable maintenance documentation
- Alan Kay's goal
  - Amplify human reach
  - Bring new ways of thinking to civilization
    (*Still a goal of CS research – e.g. see computational thinking work at CMU*)



Alan Kay with a Dynabook prototype

Smalltalk and Simula

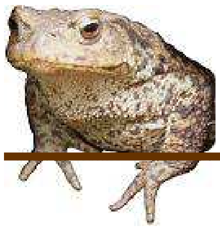
"What I got from Simula was that you could now replace bindings and assignment with *goals*. The last thing you wanted any programmer to do is mess with internal state even if presented figuratively. Instead, the objects should be presented as sites of higher level behaviors more appropriate for use as dynamic components."

- Alan Kay, The early history of Smalltalk. In History of programming languages—II, 1993.

Smalltalk

- The name
  - "Children should program in…"
  - "Programming should be a matter of…"
- Pure OO language
  - Everything is an object (including true, "hello", and 17)
  - All computation is done via sending messages
    - 3 + 4 sends the "+" message to 3, with 4 as an argument
    - To create a Point, send the "new" message to the Point class
      - Naturally, classes are objects too!
- Garbage collected
  - Following Lisp and Simula 67
- Reflective
  - Smalltalk is implemented (mostly) in Smalltalk
    - A few primitives in C or assembler
  - Classes, methods, objects, stack frames, etc. are all objects
    - You can look at them in the debugger, which (naturally) is itself implemented in Smalltalk

Smalltalk Demo



The Double Dispatch Pattern

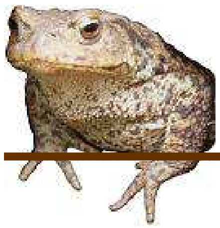
- Problem: behavior depends on two different classes

<u>Result Type for Addition Operation</u>

Right Operand

Left Operand

	Integer	Fraction	Float	Complex
Integer	Integer	Fraction	Float	Complex
Fraction	Fraction	Fraction	Float	Complex
Float	Float	Float	Float	Complex
Complex	Complex	Complex	Complex	Complex



The Double Dispatch Pattern

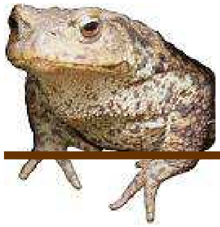
- Problem: behavior depends on two different classes
- Solution: dispatch twice

```
class Fraction
   method + aNumber
      | n d d1 d2 |
      aNumber isFraction ifTrue:
         [d := denominator gcd: aNumber denominator…].
      ^ aNumber adaptToFraction: self andSend: #+

class Integer
   method adaptToFraction: rcvr andSend: selector
      ^ rcvr perform: selector with: (Fraction numerator: self denominator: 1)
```

First dispatch to Fraction's + method: if both numbers are fractions, we compute the greatest common denominator (GCD) and proceed…

otherwise we ask the other number to turn itself into a fraction, and then add self to it

Second dispatch to Integer's adaptToFraction:andSend method Integer does so by creating a fraction with itself as the numerator and a denominator of 1. perform is a reflective method that calls '+' (the selector) in this case



The Double Dispatch Pattern

- Problem: behavior depends on two different classes
- Solution: dispatch twice

```
class Fraction
   method + aNumber
      | n d d1 d2 |
      aNumber isFraction ifTrue:
         [d := denominator gcd: aNumber denominator…].
      ^ aNumber adaptToFraction: self andSend: #+

class Float
   method adaptToFraction: rcvr andSend: selector
      ^ rcvr asFloat perform: selector with: self
```

if both numbers are fractions, we compute the greatest common denominator (GCD) and proceed…

otherwise we ask the other number to turn itself into a fraction, and then add self to it

On the other hand, Float says “no, actually a fraction should adapt to me before addition.”

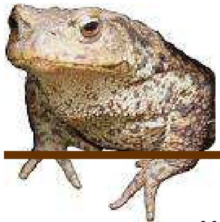
Smalltalk: Classes as Factories

"Creating different kinds of collections with a factory method"

```
OrderedCollection newFrom: #(3 2 2 1).
SortedCollection newFrom: #(3 2 2 1).
Set newFrom: #(3 2 2 1).
```

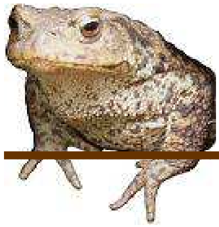
"Classes – and thus the factories – are first-class. We can assign them to a factory object and then use it to create different kinds of collections."

```
factoryObj := Set.
factoryObj newFrom: #(3 2 2 1).
factoryObj := OrderedCollection.
factoryObj newFrom: #(3 2 2 1).
```



Smalltalk, according to Alan Kay

- "In computer terms, Smalltalk is a recursion on the notion of computer itself. Instead of dividing "computer stuff" into things each less strong than the whole—like data structures, procedures, and functions which are the usual paraphernalia of programming languages—**each Smalltalk object is a recursion of the entire possibilities of the computer**.
- "…everything we describe can be represented by the recursive composition of a single kind of behavioral building block that hides its combination of state and process inside itself and can be dealt with only through the exchange of messages.
- "Thus [Smalltalk's] semantics are a bit like having thousands and thousands of computers all hooked together in a very fast network."



Dan Ingalls' perspective

- Computing should be viewed as an intrinsic capability of objects that can be uniformly invoked by sending messages... Instead of a bit-grinding processor raping and plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires.
  – Daniel Ingalls, Design Principles Behind Smalltalk (1981)

Impact of Smalltalk and Simula

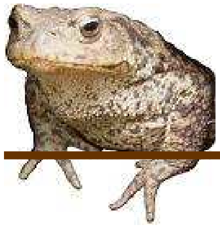
- Mac (and later Windows): inspired by Smalltalk GUI
- GUI frameworks
  - Smalltalk MVC → MacApp → Cocoa, MFC, AWT/Swing, …
- C++: inspired by Simula 67 concepts
- Objective C: borrows Smalltalk concepts, syntax
- Java: garbage collection, bytecode from Smalltalk
- Ruby: pure OO model almost identical to Smalltalk
  - All dynamic OO languages draw from Smalltalk to some extent

- Design and process ideas impacted by Smalltalk
  - Patterns, Refactoring, Extreme programming/Agile movement

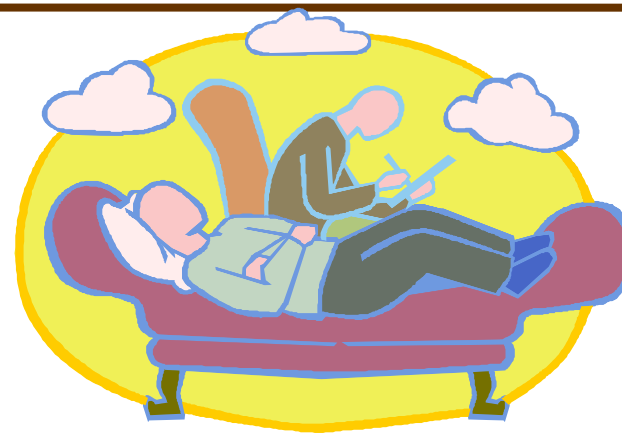
Why has OOP been successful?

- Discuss your answer with your neighbors and write it down

Material from “The Power of Interoperability: Why Objects Are Inevitable”
by Jonathan Aldrich, Onward! Essay, 2013.



Why has OOP been successful?

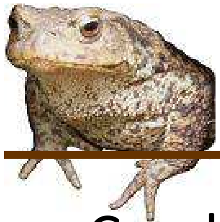


"the object-oriented paradigm...is consistent with the natural way of human thinking"

- [Schwill, 1994]

OOP may have psychological benefits.

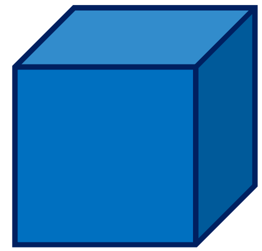
But is there a technical characteristic of OOP that is critical for modern software?



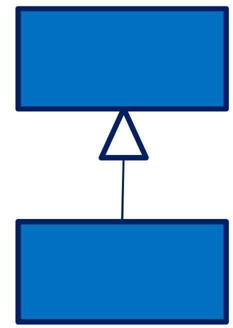
What Makes OOP Unique?

Candidates: key features of OOP

- Encapsulation?
  – Abstract data types (ADTs) also provide encapsulation



- Inheritance?

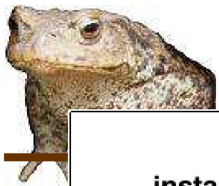


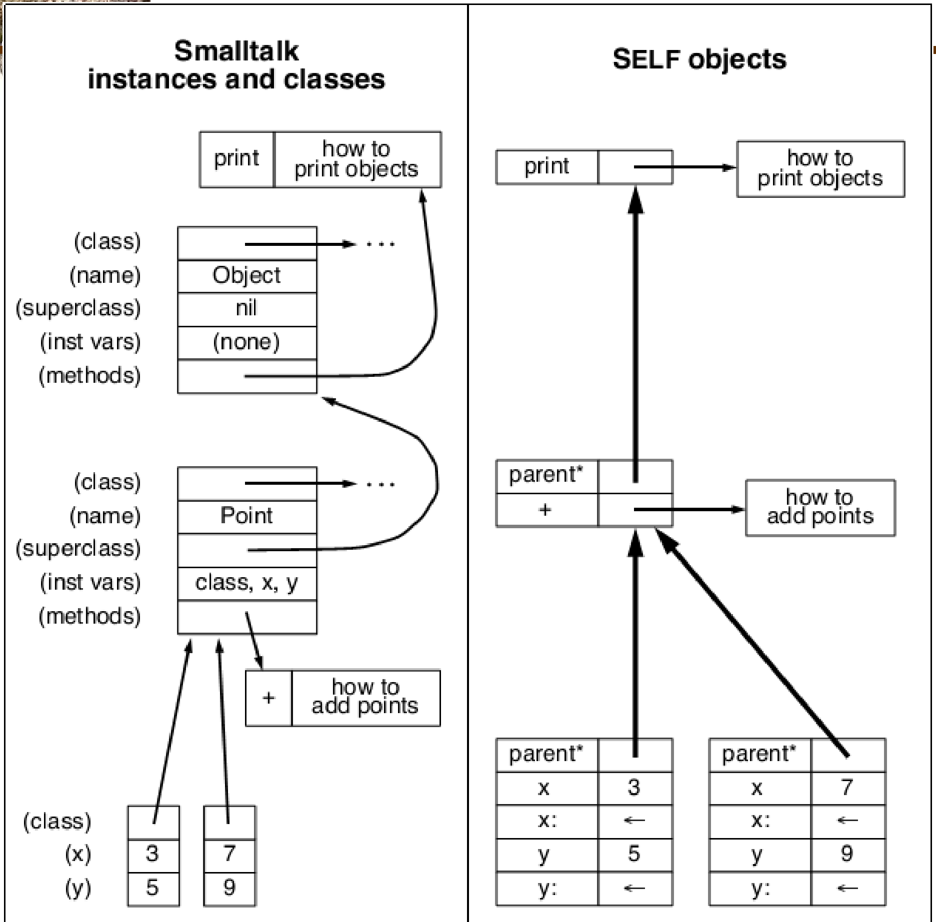
Not all OO Languages Have Inheritance

- A Modern Example: Go
  - Provides encapsulation, interfaces, dynamic dispatch
  - But no inheritance of code from a superclass

- An Alternative: Delegation
  - Supported in Self, JavaScript, others
  - There are no classes, only objects. To get a new object:
    - Create an empty object
      - Add things to it
      - Optionally, delegate to an existing object via a *parent* field
        - » If you call method *m* on an object, and *m* is not defined, the system will look for it in the parent object
    - Clone an existing object

Inheritance vs. Delegation, Graphically





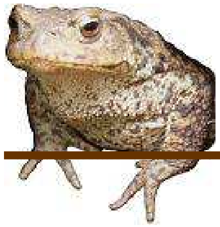
Each Smalltalk point contains a class pointer and x and y coordinates. The class **Point** supplies both format (a list of instance variables) and behavior information (a collection of methods) for points. Additional format and behavior information is inherited from **Object** via **Point**'s superclass 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.

The Self project also had a big impact on optimization of dynamic compilers

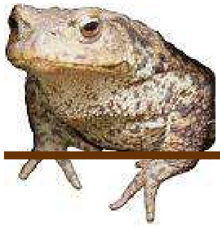
- E.g. [Chambers & Ungar, 1989]
- Used in Java, JavaScript, etc.

Source: Ungar and Smith. Self: The Power of Simplicity. *Lisp and Symbolic Computation*, 1991.



Inheritance has Benefits, Drawbacks

- Benefits
  - No easier way to reuse a partial implementation of an abstraction
  - Alternative requires forwarding each method individually
  - Especially useful when subclass and superclass call each other
    - E.g. a class with both super calls and a template method
    - Implementing Template Method, Factory is awkward in Go [Schmager, Cameron, and Noble 2010]
- Drawbacks
  - Tight coupling between subclass and superclass
    - E.g. fragile base class problem
  - Drawbacks mitigated by careful methodology



Fragile Base Class Problem

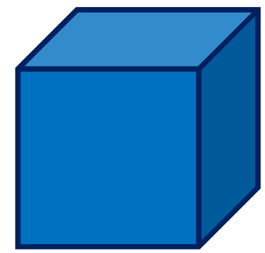
```
class List {
  private Link head;
  public void add(int i) {…}
  public void addAll(List l) {…}
  public int size() {
    … // traverses the list
  }
}

class CachedSizeList extends List {
  private int cachedSize;
  public int size() { return cachedSize; }
  public void add(int i) {
    cachedSize++;
    super.add(i);
  }
  // do we need to override addAll?
}
```

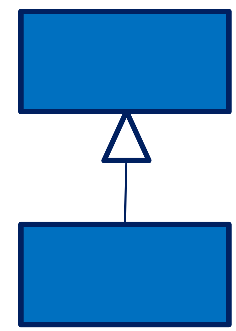
- Correct impl of subclass depends on the base class implementation
  - Couples classes, breaks modularity
- Worse: if the base class changes, the subclass will be broken
- What causes this coupling *is also what makes the template method pattern work!*
- Some solutions
  - Document internal method calls that can be intercepted
    - Document whether addAll() calls add()
  - Only make self-calls to `abstract` or `final` methods
  - Selective open recursion – language feature describes which methods are used for downcalls [Aldrich and Donnelly, 2004]

What Makes OOP Unique?

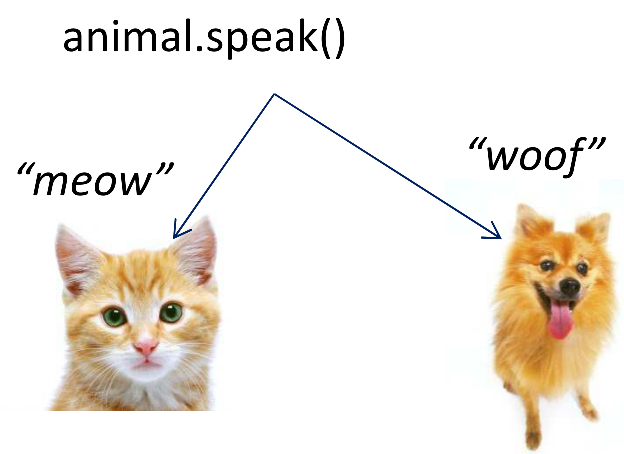
Candidates: key features of OOP

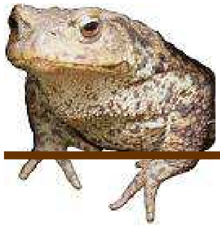
- Encapsulation?
  - Abstract data types (ADTs) also provide encapsulation





- Inheritance?
  - Neither universal nor unique in OOPLs
  - Worth studying, but not our focus
- Polymorphism/Dynamic dispatch?
  - Every OOPL has dynamic dispatch
  - Distinguishes objects from ADTs





Dynamic Dispatch as Central to OOP

Significant grounding in the OO literature

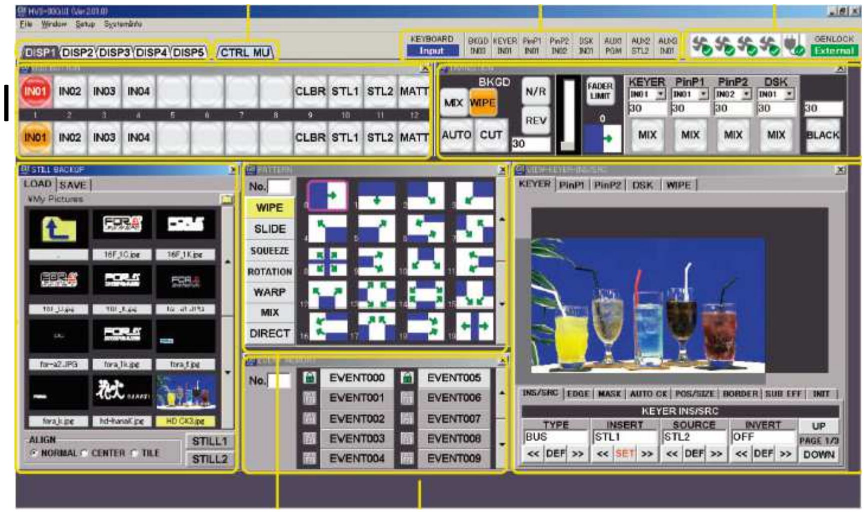
- Cook's 2009 Onward! essay
  - Object: "value exporting a procedural interface to data or behavior"
  - Objects are self-knowing (*autognostic*), carrying their own behavior
  - Equivalent to Reynolds' [1975] *procedural data structures*

- Historical language designs
  - "the big idea [of Smalltalk] is messaging" [Kay, 1998 email]

- Design guidance
  - "favor object composition over class inheritance" [Gamma *et al.* '94]
  - "black-box relationships *[based on dispatch, not inheritance]* are an ideal towards which a system should evolve" [Johnson & Foote, 1988]

Interoperability of Widgets

- Consider a Widget-based GUI
  - Concept notably developed in Smalltal



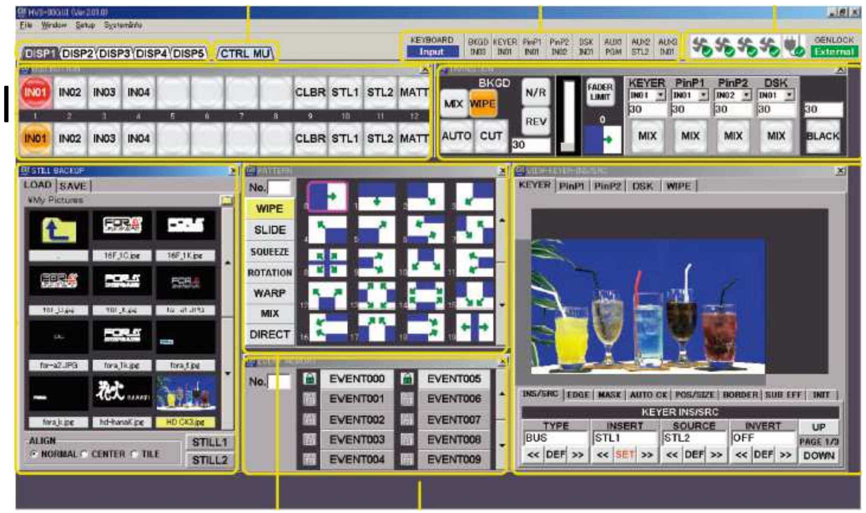
Source: http://www.for-a.com/products/hvs300hs/hvs300hs.html

```
interface Widget {
   Dimension getSize();
   Dimension getPreferredSize();
   void setSize(Dimension size);
   void paint(Display display);

   … /* more here */ }            // based on ConstrainedVisual from Apache Pivot UI framework
```

- Nontrivial abstraction – not just paint()
  - A single first-class function is not enough

Interoperability of Composite Widgets

- Consider a **Composite GUI**
  – Concept notably developed in Smalltal



Source: http://www.for-a.com/products/hvs300hs/hvs300hs.html

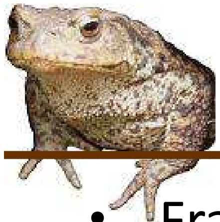
```
class CompositeWidget implements Widget {
  Dimension getSize();
  Dimension getPreferredSize();
  void setSize(Dimension size);
  void paint(Display display);
  void add(Widget widget)
  … /* more here */ }          // based on Container from Apach
```

Object-oriented dispatch supports ***interoperability*** between different Widgets in a Composite

- Nontrivial abstraction – not just paint()
  – A single first-class function is not enough
- Composite needs to store diverse subcomponents in a list
  – Can't do this with type classes, generic programming
- Composite needs to invoke paint() uniformly on all subcomponents
  – Also breaks type classes, generic programming

Software Frameworks

- A framework is "the skeleton of an application that can be customized by an application developer" [Johnson, 1997]
- Frameworks uniquely provide **architectural reuse**
  - Reuse of "the edifice that ties components together" [Johnson and Foote, 1988]
  - Johnson [1997] argues can reduce development effort by 10x
- As a result, frameworks are ubiquitous
  - GUIs: Swing, SWT, .NET, GTK+
  - Web: Rails, Django, .NET, Servlets, EJB
  - Mobile: Android, Cocoa
  - Big data: MapReduce, Hadoop



Frameworks need Objects

- Frameworks define **abstractions** that extensions implement
  - The developer "supplies [the framework] with a set of components that provide the application specific behavior" [Johnson and Foote, 1988]
  - Sometimes the application-specific behavior is just a function
  - More often, as we will see, these abstractions are **nontrivial**

- Frameworks require **modular extensibility**
  - Applications extend the framework without modifying its code
    - Frameworks are typically distributed as binaries or bytecode
    - *cf.* Meyer's [1988] open-closed principle
  - Framework developers cannot anticipate the details of extensions
    - Though they do plan for certain kinds of extensions

- Frameworks require **interoperability**
  - Plugins often must **interoperate** with each other
  - Frameworks must **dynamically** manage diverse plugins
  - We have already seen this for GUI widgets – let's look at other examples

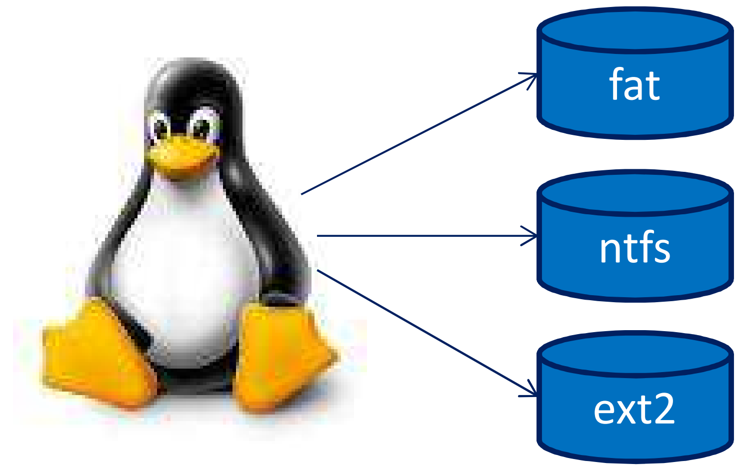
Web Frameworks: Java Servlets

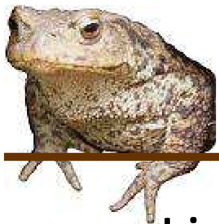
```
interface Servlet {
  void service(Request req, Response res);
  void init(ServletConfig config);
  void destroy();
  String getServletInfo();
  ServletConfig getServletConfig();
}
```

- Nontrivial abstraction
  - Lifecycle methods for resource management
  - Configuration controls
- Modular extensibility
  - Intent is to add new Servlets
- Interoperability required
  - Web server has a list of diverse Servlet implementations
  - Dispatch is required to allow different Servlets to provide their own behavior

Operating Systems: Linux

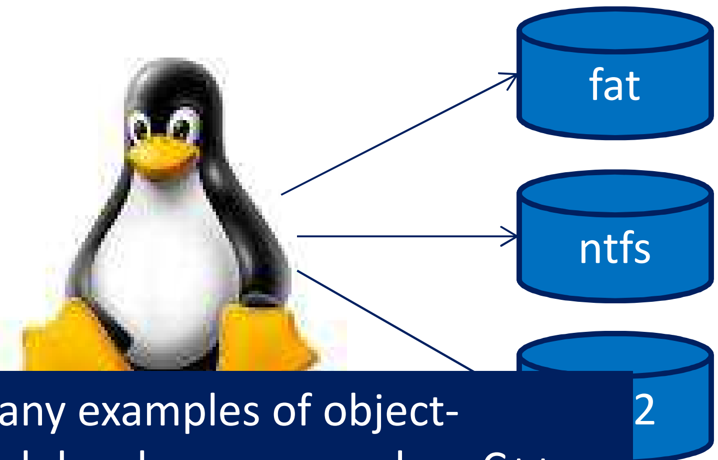
- Linux is an OO framework!
  - In terms of design—not implemented in an OO language
- File systems as objects
  - Interface is a struct of function pointers
  - Allows file systems to interoperate
    - E.g. symbolic links between file systems
- Not just file systems
  - Many core OS abstractions are extensible
  - ~100 object-like abstractions in the kernel





Operating Systems: Linux

- Linux is an OO framework!
  - In terms of design—not implemented in an OO language
- File systems as service abstractions
  - Interface is a struct of function



People often miss this, or even deny it, but there are many examples of object-oriented programming in the kernel. Although the kernel devel-opers may shun C++ and other explicitly object-oriented languages, thinking in terms of objects is often useful. The VFS [Virtual File System] is a good example of how to do clean and efficient OOP in C, which is a language that lacks any OOP constructs.

- Robert Love, *Linux Kernel Development (2nd Edition)*

Software Ecosystems

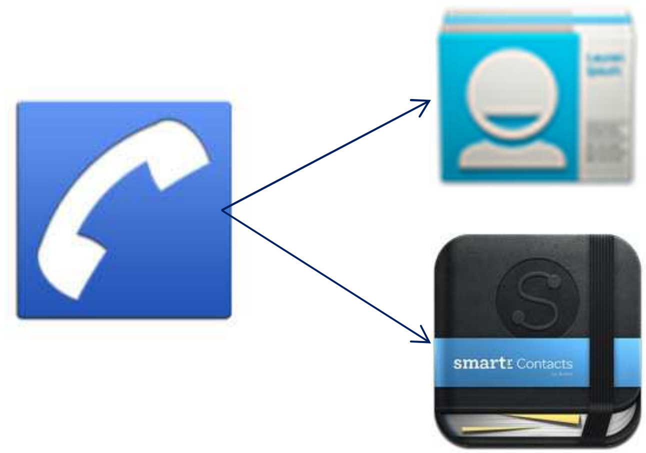
- A ***software ecosystem*** is a "set of software solutions that enable, support, and automate the activities...[of] actors in the associated social or business ecosystem" [Bosch, 2009]
  – Examples: iOS, Android, Windows, Microsoft Office, Eclipse, Amazon Marketplace, …

- Ecosystems have enormous **economic impact**
  – Driven by network effects [Katz and Shapiro, 1985]
  – Top 5 tech firms control or dominate an ecosystem
    - Apple, Microsft, IBM, Samsung, Google

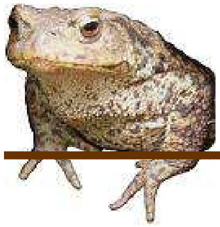
- Ecosystems require **interoperability**
  – Critical to achieving benefit from network effects
  – "the architecture provides a formalization of the rules of interoperability and hence teams can, to a large extent, operate independently" [Bosch, 2009]

Mobile Devices: Android

```
class ContentProvider {
  abstract Cursor query(Uri uri, ...);
  abstract int insert(Uri uri, ContentValues vals);
  abstract Uri update(Uri uri, ContentValues vals, ...);
  abstract int delete(Uri uri, ...);
  ... // other methods not shown
}
```



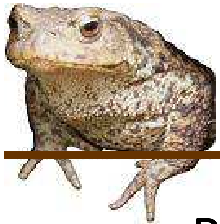
- Network effects (apps) give Android value
- Apps build on each other
  - Example: contact managers
    - Smartr Contacts is a drop-in replacement for the default contact manager
    - Phone, email apps can use Smartr Contacts *without preplanning*
  - Enabled by service abstraction interfaces
    - Android keeps a list of heterogeneous ContentProvider implementations



Conclusions: Why Objects Were Successful

- The essence of objects is **dispatch**
- Dispatch provides **interoperability**
- First-class interoperability is critical to **frameworks** and **ecosystems**
- Frameworks and ecosystems are **economically critical** to the software industry

- Likely a significant factor in objects' success
  - Future study is warranted to validate the story above
  - Other factors (psychology, benefits of inheritance) are worth exploring too



Sample Exam Questions

- By making each class an object, Smalltalk supports what design pattern?
- Name at least one of the designers of Simula or Smalltalk
- Multiple choice: the primary designer of Smalltalk compared objects to:
  - Records
  - Functions
  - Networked computers
  - Cars
- Which of the following ideas were new in Simula 67?
  - Subclasses with inherited fields
  - The ability to define subclasses separately from the superclass
  - Dynamic dispatch
  - Multiple inheritance
  - Garbage collection

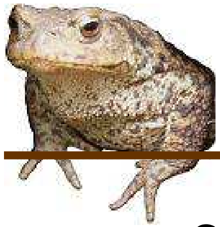
Sample Exam Questions

- Explain how Smalltalk can add different kinds of numbers together, always producing the right kind of number as a result
- What feature of object-oriented programming was likely most important to its success?

Takeaways and Next Week

- Today: The Past of Objects
  – Origins in simulation and Hoare's Record Classes
  – Inheritance and virtual procedures in Simula 67
  – Everything as an object in Smalltalk
  – Smalltalk's impact: GUIs, frameworks
  – Role of dispatch, frameworks in adoption of OO technology
- Next Week
  – The Present of Objects: Java 8*
  – The Future of Objects: Scala*

    *these are illustrative examples



Resources

- Squeak – a modern Smalltalk implementation
  - http://www.squeak.org/
  - Alan C. Kay. The Early History of Smalltalk. Proc. History of Programming Languages, 1993. http://portal.acm.org/citation.cfm?id=155364

- GNU Simula
  - https://www.gnu.org/software/cim/
  - An Introduction to Programming in Simula. Rob Pooley. http://www.macs.hw.ac.uk/~rjp/bookhtml/

- The Power of Interoperability: Why Objects Are Inevitable. Jonathan Aldrich. In Onward! Essays, 2013.
  - http://www.cs.cmu.edu/~aldrich/papers/objects-essay.pdf