CSCI 334:
Principles of Programming Languages

Lecture 15: Object-Oriented Programming

Instructor: Dan Barowy

Williams

Announcements

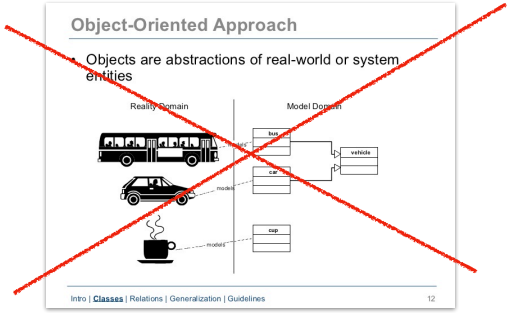
Will release HW7 later today.

Object-Oriented Programming

- OOP is both a language design philosophy and a way of working (OO design).
- OOP is possibly the most impactful development in the history of programming languages.

What OOP is Not

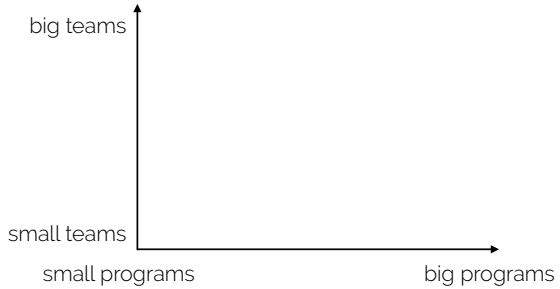
- Many, many instructors introduce OOP as a way of naturally simulating the world.



- While there is a natural affinity between real-world modeling and OOP, this misses the point entirely.

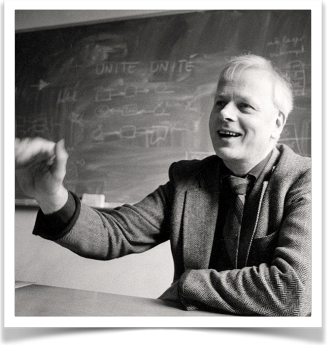
What OOP is

- Object-oriented programming is actually about scalability.
- Scalability in codebase size was the original motivation.
- But OO philosophy also has had a big effect on the scalability of programming teams.

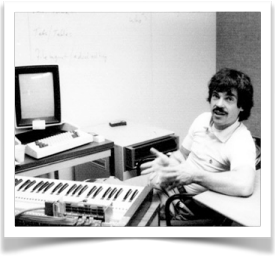


History

- First language recognizable as OO: Simula-67.
- Developed by Kristen Nygaard and others at the Norwegian Computing Center.
- Grew out of frustrations using ALGOL.
- Original plan was to add an "object" library, inspired by C.A.R. Hoare's "record classes".
- It was eventually realized that objects were a fundamentally different way of structuring a program; Simula became its own language.



History



- First mainstream success: Smalltalk
- Developed by Alan Kay, Dan Ingalls, and Adele Goldberg at Xerox PARC and later Apple Computer.
- Used to implement major components of the groundbreaking Xerox Alto computer.
- Highly influential. E.g., C++, Java, Ruby, etc.

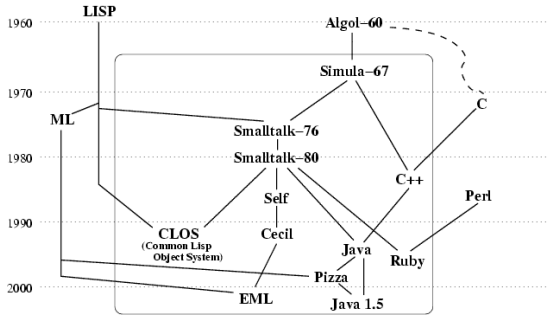
History

- https://www.youtube.com/watch?v=A09W93OxQ7U

History

And they showed me really three things. But I was so blinded by the first one I didn't even really see the other two. One of the things they showed me was object orienting programming they showed me that but I didn't even see that. The other one they showed me was a networked computer system… they had over a hundred Alto computers all networked using email etc., etc. I didn't even see that. I was so blinded by the first thing they showed me which was the graphical user interface… within you know ten minutes it was obvious to me that all computers would work like this some day.

History



OK, really, what is OO?

Object-oriented programming is composed primarily of four key language features:

1. Abstraction
2. Dynamic dispatch
3. Subtyping
4. Inheritance

Abstraction

- Similar concept to abstraction in the lambda calculus; a way of scoping bindings.
- More concretely: a way of "encapsulating" or hiding data.
- This data structure is called an *object*.
- Objects are *instantiated* from a template called a *class*.
- You are already familiar with this idea from Java.

```
class Circle {
  private int centerX = 0;
  private int centerY = 0;
  private int radius = 1;
  ...
}
```

Abstraction

```
class Circle {
  private int centerX = 0;
  private int centerY = 0;
  private int radius = 1;
  ...
}
```

- Users of this code cannot access field members by default.

```
Circle c = new Circle();
System.out.println(c.centerX);
```

```
Main.java:4: error: centerX has private access in Circle
    System.out.println(c.centerX);
                       ^
1 error
```

Abstraction

```
Object subclass: 'Circle'
  instanceVariableNames: 'centerX centerY radius'
```

- In Smalltalk, data stored inside an object (an "instance variable") is "private" to that object by default (unlike Java).

```
c := Circle new.
```

- In Smalltalk, there is nothing special about new. It it just a method.

Abstraction

```
Object subclass: 'Circle'
  instanceVariableNames: 'centerX centerY radius'
  new x: xvalue y: yvalue
    centerX := xvalue.
    centerY := yvalue.
    radius  := 1.
    ^ self.
```

```
c := (Circle new) x: 3 y: 4.
```

Abstraction

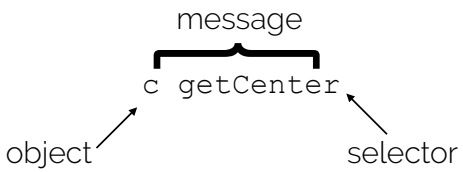
- Objects collect both data ("instance variable") and functions ("methods").
- The pairing of data and methods is specifically designed to aid in the evolution of the software system: objects encapsulate data, and the allowable operations on that data are defined by methods.

Abstraction

- In Smalltalk *everything* is an object.
- Every object is also, transitively, a subclass of the base class, `Object`.
- Java broke with this convention for performance reasons.
- It caused great pain when Java Generics were introduced.
- Scala, which was heavily influenced by both Java and Smalltalk, reverted to the everything-is-an-object model (Scala's base class is called `Any`).

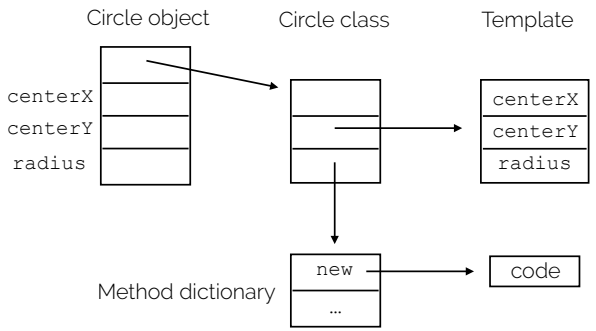
Dynamic Dispatch

- Dynamic dispatch is how functions are called.
- Unlike in SML, functions ("methods") are always tied to an object (or class).
- A method is called ("dispatched") by sending a "message" to the "selector" of an object.



Dynamic Dispatch

- Dynamic dispatch is an algorithm for finding an object's method corresponding to a given selector name.



Inheritance

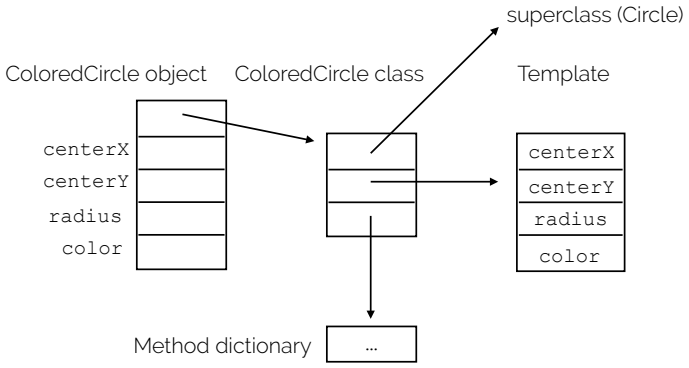
- Inheritance is a time-saving feature.
- It adds zero "power" to the language (does not change the set of programs that one can write).
- However, it makes code substantially easier to write and maintain.
- The idea is about *reuse of code*.

```
Circle subclass: 'ColoredCircle'
  instanceVariableNames: 'centerX centerY radius color'
```

- `ColoredCircle` will "inherit" the `new` method from `Circle`.
- No need to write method unless it is different.

Inheritance

- Inheritance relies on dynamic dispatch to find missing methods.



Subtyping

- ColoredCircle is a subtype of Circle because it can do all of the things that Circle can do (and then some).
- We often create subtypes using the inheritance mechanism.
- However, subtyping and inheritance are two *completely distinct* concepts.
- Subtyping is about the *logical relationship* between two types.
- Inheritance is a *mechanism* for code reuse.

Subtyping vs. Inheritance

- E.g., imagine you are going to implement a Dequeue, a Stack, and a Queue.
- Stack: add and remove from one end (e.g., "left").
- Queue: add from one end (e.g., "left"), remove from other (e.g., "right").
- Dequeue: add and remove from either end.
- Formally:

  Type **A** is a subtype of a type **B** if any context expecting an expression of type **B** may take any expression of type **A** *without introducing a type error*.

Subtyping vs. Inheritance

- Subtyping rule-of-thumb: can you substitute class A for class B?
- If so, **`A <: B`**.

```
c := Circle new.
moveCircle c.

cc := ColoredCircle new.
moveCircle cc.
```

- Thus, **`ColoredCircle <: Circle`**.

Subtyping vs. Inheritance

- In terms of code reuse, it makes perfect sense to implement a Stack and Queue on top a Dequeue. Dequeue has all the functionality needed.
- (Smalltalk allows one to "uninherit" methods from a superclass)
- But Stack and Queue are not subtypes of Dequeue!
- The converse is true!

```
Dequeue <: Stack
Dequeue <: Queue
```

Object-Oriented Extensibility

- Dan Ingalls developed a test for what qualifies as an "object-oriented" programming language.
- The test is about the ability to extend software *after* it has already been designed and written.
- E.g., suppose you have a class for a `ColoredRectangle`.
- Can you define a new kind of number (e.g., fractions), use your new numbers to define a new kind of rectangle, ask the system to color the rectangle, and have everything work? If so, you have an OO system.

OO vs Functional Tradeoff

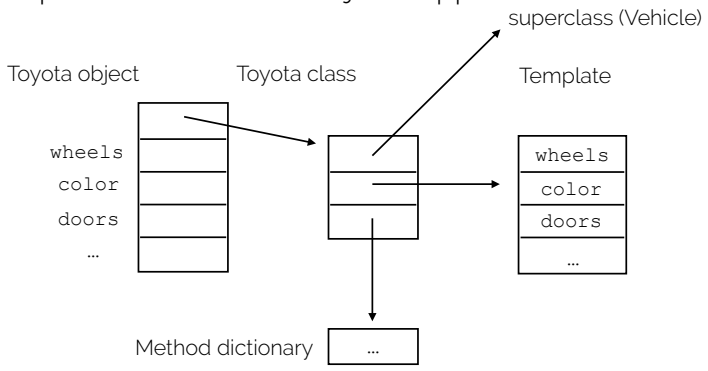
- OO offers a different kind of extensibility than functional (or function-oriented) languages.
- Suppose you're modeling a hospital.

Operation	Doctor	Nurse	Orderly
`Print`	`Print Doctor`	`Print Nurse`	`Print Orderly`
`Pay`	`Pay Doctor`	`Pay Nurse`	`Pay Orderly`

- Functional programming makes it easy to add operations.
- OO programming makes it easy to add data.

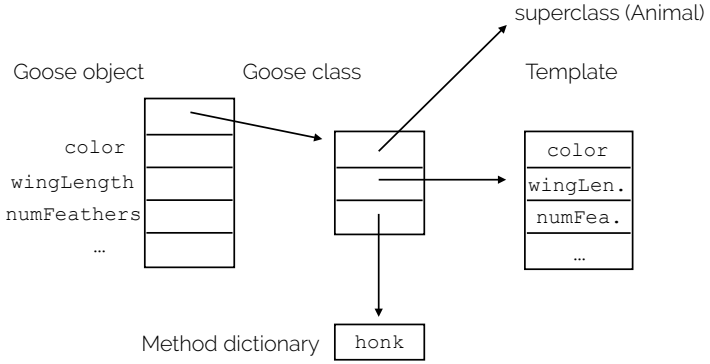
Polymorphism

- Dynamic dispatch allows for a kind of polymorphism.
- It does not matter whether a method exists because of a superclass or because it just happens to be there.



Polymorphism

- E.g,. both Geese and Toyotas can honk.
- Goose is not a subtype of Vehicle!
- Goose is does not inherit from Vehicle!



Object-Closure Duality

- Objects are kind of closure: can be simulated using activation records that bind over function names.
- But it is very difficult to implement subtyping and inheritance correctly without first-class support!