# A Bit of History

Some notable examples of early object-oriented languages and systems:

- Sketchpad (Ivan Sutherland's 1963 PhD dissertation) was the first system to use classes and instances (although Sketchpad is an application, not a programming language)
- First object-oriented programming language: Simula I, then Simula 67, created by Ole-Johan Dahl and Kristen Nygaard at the Norwegian Computing Center in Oslo.
- Smalltalk: developed at Xerox Palo Alto Research Center by the Learning Research Group in the 1970's (Smalltalk-72, Smalltalk-76, Smalltalk-80)
- Today: mature language paradigm. Some significant examples: C++, Java, C#, Python, Ruby.

# Ruby

Why Ruby?

Some basics of Ruby programs

- Syntax
- Classes, Methods
- Variables, fields, scope
- Dynamic typing
- The rep-loop, the main class, etc.

Note: Read Thomas book chapters 1–9 (or free first edition 1–8)

- Skip/skim regexps and ranges
- Not every detail: focus on OO, dynamic typing, blocks, mixins

# Principal Properties of Ruby

- *Pure* object-oriented: *all* values are objects
- Class-based
- Dynamically typed
- Convenient *reflection*

A good starting point for discussing what each of these means and what other languages look like.

| | dynamically typed | statically typed |
|---|---|---|
| functional | Scheme | Haskell |
| object-oriented | Ruby | Java |

# Ruby vs. Smalltalk

Smalltalk: language definition unchanged since 1980 (although lots of work on the environment and packages), is also pure OO, class-based, dynamically-typed.

- Smalltalk: tiny language (smaller than Scheme), elegant, regular, can learn whole thing
- Smalltalk: integrated into cool, malleable GUI environment
- Ruby: large language with a "why not?" attitude
- Ruby: scripting language (light syntax, some "odd" scope rules)
- Ruby: very popular, massive library support especially for strings, regular expressions, "Ruby on Rails." Won't be our focus at all.
- Ruby: *mixins* (a cool, advanced OO modularity feature)
- Ruby: blocks, libraries encourage lots of FP idioms

# Really key ideas

- Really, everything is an object (with constructor, fields, methods)
- Every object has a class, which determines how the object responds to messages.
- Dynamic typing (everything is an object)
- Dynamic dispatch
- Sends to `self` (a special identifier; Java's this)
- Everything is "dynamic" – evaluation can add/remove classes, add/remove methods, add/remove fields, etc.
- Blocks are *almost* first-class anonymous functions (later)
  - Can convert to/from real lambdas (class `Proc`)

(Also has some more Java/C like features – loops, return, etc.)

# Lack of variable declarations

If you assign to a variable in scope, it's mutation.

If the variable is not in scope, it gets created (!)

- Scope is the method you are in

Same with fields: an object has a field if you assign to it

- So different objects of the same class can have different fields (!)

This "cuts down on typing" but catches fewer bugs (misspellings)

- A hallmark of "scripting languages" (an informal term)

# Protection?

- Fields are inaccessible outside of instance
  - Define accessor/mutator methods as desired
    - Use `attr_read` and `attr_writer`
  - Good OO design: subclasses can override accessors/mutators
- Methods are public, protected, or private
  - protected: only callable from class or subclass object
  - private: only callable from `self`
- Later: namespace management, but no hiding

# Unusual syntax

Just a few random things (keep your own mental list):

- Variables and fields are written differently
  - `@` for fields
  - `@@` for class fields (Java's static fields)
- Newlines often matter — need extra semicolons, colons, etc. to put things on one line
- Message sends do not need parentheses (especially with 0 arguments)
- Operators like + are just message sends
- Class names must be capitalized

# Duck Typing

"If it walks like a duck and quacks like a duck, it's a duck."

A method might think, "I need an `Octopus`" but really it only needs an object that has similar enough methods that it acts enough like a `Octopus` that the method works.

Embracing duck typing: Methods that make method calls rather than assume the class of their argument.

Plus: More code reuse, very OO approach

- What messages can some object receive is all that matters

Minus: Almost nothing is equivalent

- `x+x` versus `x*2` versus `2*x`
- Callees may not want callers assuming so much

# Blocks and Iterators

Many methods in Ruby "take a block," which is a "special" thing separate from the argument list.

They are used very much like closures in functional programming; can take 0 or more arguments (see examples)

The preferred way for iterating over arrays, doing something $n$ times, etc.

They really are closures (can access local variables where they were defined).

Useful on homework: `each`, possibly `inject`

Useful in Ruby: many, many more

# Blocks vs. Procs

These block arguments can be used only by the "immediate" callee via the `yield` keyword.

If you really want a "first-class object" you can pass around, store in fields, etc., convert the block to an instance of `Proc`.

- `lambda {|x,y,z| e}`
- Instances of `Proc` have a method `call`
- This *really* is exactly a closure.

Actually, there is a way for the caller to pass a block and the callee convert it to a `Proc`.

- Look it up if you're curious.
- This is what `lambda` does
  (just a method in `Object` that returns the `Proc` it creates)

# Subclasses

Ruby is dynamically typed, so subclassing is *not* about what type-checks.

Subclassing is about *inheriting methods* from the superclass.

- In Java, it's about inheriting fields too, but we can just write to any field we want.

Example: `ThreeDPoint` inherits methods `x` and `y`.

Example: `ColorPoint` inherits `distFromOrigin` and `distFromOrigin2`.

# Overriding

If it were just inheritance, then with dynamic typing subclassing would just be avoiding copy/paste.

It's more.

But first, "simple" overriding lets us redefine methods in the subclass.

- Often convenient to use `super` to use superclass definition in our definition.

This is still "just" avoiding copy-paste.

Example: `distFromOrigin` and `initialize` in `ThreeDPoint`.

# Ruby-ish Digression

Why make a subclass when we could just add/change methods to the class itself?

- Add a color field to `Point` itself
- Affects all `Point` instances, even those already created (!)

Plus: Now a `ThreeDPoint` has a color field too.

Minus: Maybe that messes up another part of your program.

Fun example: Redefining `Fixnum`'s + to return 5.

# Late-Binding

So far, this OO stuff is very much like functional programming

- Fields are just like things in a closure's environment (remember simulating objects in Scheme)

But this is totally different:

- When a method defined in a superclass makes a `self` call it resolves to the method defined in the subclass (typically via overriding)

Example: `distFromOrigin2` in `PolarPoint` still works correctly!!!

Next batch of slides: Studying this very carefully.