Object-Oriented Programming

OOP =

Abstract Data Types ...
- package representation of data structure together with operations on the data structure
- encapsulate internal implementation details

+ **Inheritance** ...
- support defining a new ADT as an incremental change to previous ADT(s)
- share operations across multiple ADTs

+ **Subtype Polymorphism** ...
- variables can hold instances of different ADTs that have a common interface

+ **Dynamic Dispatching**
- run-time support for selecting right implementation of an operation, depending on argument(s)

Some OO languages

Simula 67: the original

Smalltalk-80: popularized OO
Self: Smalltalk-80 refinement ⇒ purest OO

C++: OO for the hacking masses; complex & powerful
Java, C#: cleaned up, more portable variants of C++

CLOS: powerful OO part of Common Lisp

Cecil, MultiJava, **EML**, **Diesel**:
OO languages from my research group

Emerald, Kaleidoscope: other OO languages from UW

Abstract data types

ADT: a user-defined data type along with operations for manipulating values of the type
- allow language to be extended with new types, raising & customizing the level of the language

"Abstract" means **encapsulated**
- exposes only **interface** to data structure & operations, hides internal implementation details
- presents simpler external view by hiding distracting internal details
- prevents undesired dependencies of clients on implementation
  - allows it to be changed w/o affecting clients

Called a **class** in (most) OO languages
- values of data type called **objects** or **instances** of the class
- operations called **methods**
- components of data type called **instance variables**
- each class implicitly defines a new type

Modules have similar benefits

Inheritance

Most recognizable aspect of OO languages & programs

Define new class as *incremental modification* of existing class
- new class is **subclass** of the original class (the **superclass**)
- by default, **inherit** superclass's methods & instance vars
- can add more methods & instance vars in subclass
- can **override** (replace) methods in subclass
  - but not instance variables, usually

Example

```
class Rectangle {
  Point center;
  int height, width;
  int area() { return height * width; }
  void draw(OutputDevice out) { ... }
  void move(Point new_c) { center = new_c; }
  ...
}

class ColoredRectangle extends Rectangle {
  // center, height, & width inherited
  Color color;
  // area, move, etc. inherited
  // draw overridden
  void draw(OutputDevice out) { ... }
}

Rectangle r = new Rectangle();
ColoredRectangle cr = new ColoredRectangle();
...
print(r.area()); print(cr.area());
r.draw(); cr.draw();
```


Benefits of inheritance

Can reuse ADT implementation code easily

- overriding allows customization of inherited code, allowing still more reuse

Supports key idiom of **factoring** code into common superclass

- superclass can be **abstract**
  - incomplete, fleshed out by subclasses
- encourages development of rich libraries of related data structures

May model real world scenario well

- use class to model different things
- use inheritance for classification of things: subclass is a special case of superclass

Pitfalls of inheritance

Inheritance often overused by novices

Code gets fragmented into small factored pieces

Simple extension & overriding may be too limited

- e.g. exceptions in classification hierarchies

A complex mechanism, with subtle static typechecking constraints

Subtype polymorphism

Subtyping: organize types into more general types (**supertypes**) and more specific types (**subtypes**)

- values of a supertype are a superset of values of a subtype
- each class is a type $\Rightarrow$ subclass is a subtype of superclass

Subtype polymorphism: allow value of subtype to be used wherever value of supertype is expected

- code written for superclass reused for all subclass

Example:

```
void client(Rectangle r) {
  ... r.draw(...) ... }

Rectangle r = ...;
client(r); // legal
ColoredRectangle cr = ...;
client(cr); // legal: ColoredRect. subtypes Rectangle
```

But in each call of `client`, what implementation of `draw` is invoked?

Dynamic dispatching

Dynamic dispatching:
when call an operation on an object,
invoke appropriate operation for *dynamic* class/type
of object, not *static* class/type

```
void client(Rectangle r) {
  ... r.draw(...) ... }

Rectangle r = ...;
client(r); // invokes Rectangle's draw method

ColoredRectangle cr = ...;
client(cr); // invokes ColoredRect.'s draw method
```

Dynamic dispatching also known as
dynamic binding,
message passing,
virtual function calling,
generic function application

"Most appropriate method"

For a dynamically dispatched call `obj.msg(args)`,
the most appropriate method is the one defined in or inherited by the run-time class of `obj` (the **receiver**)
- (in most OO languages)

An algorithm:
- start with run-time class `C` of `obj`
  - if a method named `msg` is defined in `C`, then invoke it
  - otherwise, recursively search in superclass of `C`
- if never find match, report run-time error
  ⇒ static type checker guarantees this won't happen

Precomputed virtual function tables are an optimized implementation that yields the same results as this algorithm

Factoring

Key programming idiom: abstract superclasses
- defines common interface across multiple implementations/variations/extensions
- provides (partial) default implementation
  - can be left abstract
  - others can be defined in terms of abstract "primitives"
  - default methods can be overridden by subclasses if desired

Example:

```
abstract class Shape {
  // draw in interface of all shapes,
  // but is implemented in subclasses
  abstract void draw(OutputDevice out);
  // drawConsole provides default implementation in terms of
  // draw (which can be called despite being abstract!)
  void drawConsole() { draw(Console); }
  ...
}

// lots of abstract and concrete subclasses of Shape
...
```


Example: displaying shapes in a list

Without dynamic dispatching, use "typecase" idiom:

```
foreach (Shape s in scene.shapes) {
  if (s.is_rectangle()) {
    ((Rectangle)s).draw();
  } else if (s.is_square()) {
    ((Square)s).draw();
  } else if (s.is_circle()) {
    ((Circle)s).draw();
  } else {
    error("unexpected shape");
  }
}
```

- similar: switch over enum tags

With dynamic dispatching, use single message:

```
foreach (Shape s in scene.shapes) {
  s.draw();
}
```

What if add new `Shape` subclasses to library?

Dynamic dispatching vs. static overloading

Example of overloaded methods:

```
void m(Rectangle r) { ... }
void m(ColoredRectangle cr) { ... }

ColoredRectangle cr = ...;
Rectangle r = cr;
m(r); // which overloaded method gets invoked?
```

Like overloading:

- multiple methods with same name, in different classes
- use class/type of argument to resolve to desired method

Unlike overloading:

- resolve using *run-time* class of argument, not *static* class/type
- consider only receiver argument, in most OO languages
  - C++, Java, C#: regular static overloading on arguments, too
  - CLOS, Cecil, EML, Diesel: resolve using all arguments (**multiple dispatching**)
  - MultiJava: support both multiple dispatching & static overloading

Benefits of dynamic dispatching

Makes overriding in the face of subtype polymorphism much more useful

- override won't be ignored
- greater potential for reuse

Allows more factoring of common code into superclass, since superclass code can be "parameterized" by "sends to **self**/**this**" that invoke subclass-specific operations

Dynamically dispatched calls are points of external extensibility

- unlike typecase idioms, switches over enum tags, etc.

Pitfalls of dynamic dispatching

Tracing flow of control of code is harder

- control can pop up and down the class hierarchy

Adds run-time overhead

- space for run-time class info
- time to do method lookup
  - but typically only an array lookup + indirect jump, not a search

Design space for object-oriented languages

Basic object model:

- hybrid vs. pure OO
- class-based vs. classless (prototype-based)

Inheritance and dispatching model:

- single inheritance vs. multiple inheritance
- nested, virtual classes?
- single dispatching vs. multiple dispatching

Static type checking:

- types vs. classes
- by-name vs. structural (sub)typing
- subtype-bounded polymorphism?

Self

A purely OO language
- developed as a Smalltalk successor by Ungar & Smith [87]

Theme: “the power of simplicity”
- à la λ-calculus, Scheme

Salient features:
- every thing is an object
  - including primitives like numbers, booleans, first-class functions
  - no classes, just objects
- every action is a message
  - including basic operations like +, <
  - including basic control structures like if, while
  - including invocation of first-class functions
  - including reading & assigning to instance variables
- nested scoping via inheritance
- dynamically typed
- interactive, reflective development environment

(A Self interpreter & some basic library code in available.)

Self objects

An object is just a sequence of *slots*

Each slot is a key/value pair
- the key is a name
- the value is a reference to another object

Examples:

```
( | x = 3. y = 4. | )
( | “no slots” | )
( “no slots” )
```


Accessing slots

The only thing you can do to an object is send it a message

To fetch the contents of a slot, send the slot’s name as a message to the object
- postfix syntax

Example:

```
val aPoint = (| x=3. y=4. |).
aPoint x      → 3
```

(`val` isn’t real Self. We’ll see later how to define top-level names.)

Methods

A method is just a special kind of object stored in a slot
- special because a method object has code as well as optional slots

Example:

```
val aPoint =
  (| x=3.
     y=4.
     distanceToOrigin = (
       (self x squared + self y squared) sqrt ).
  |).
aPoint distanceToOrigin    → 5
```

Message send semantics:
- find slot with name matching message
- if it contains an object without code (e.g. 3 or `aPoint`), then just return that object
- otherwise, evaluate the code, and return its result
  - code = a sequence of expressions, separated by periods

Syntax of messages

Unary messages: a simple identifier, written after the receiver expression
- highest precedence, right-associative
- `aPoint distanceToOrigin`
- `self x squared`
- `(...) sqrt`

Binary messages: a sequence of punctuation symbols, written between the receiver and the argument
- users can define their own operators!
- medium precedence, left-associative
- `self x squared + self y squared`
- `3 + 4 * 5`

Keyword messages: an identifier followed by a colon, written between the receiver and the argument
- (later: keywords with more than one argument)
- lowest precedence, non-associative
- `aPoint distanceTo: anotherPoint`
- `self x: 3`

Sends to self

In a method, the name of the receiver argument is `self`
- (like `this` in C++, Java, C#)

In a message send, can omit receiver expression
- defaults to `self`
- e.g. `self x squared` can be written `x squared`

E.g.

```
val aPoint =
  (| x=3.
     y=4.
     distanceToOrigin = (
       (x squared + y squared) sqrt ).
  |).
```

Method calls now as concise as (traditional) instance variable accesses!

Mutable slots

Slots can be immutable or mutable
- slots initialized with = are *immutable*
- slots initialized with `<-` are *mutable*

To change the contents of a slot named *x* in object *obj*, send the *x*`:` message to *obj* with the new value as the arg.
- returns the receiver, e.g. for additional assignments

Example:

```
val aPoint = (| x <- 3. y <- 4. |).
aPoint x: 5.  "updates aPoint's x slot to refer to 5"
aPoint x          → 5
aPoint y: aPoint y + 1.  "increments aPoint's y slot"
(aPoint x: 0) y: 0.      "reset aPoint to the origin"
```


Assignment slots

A mutable slot declaration *x* `<-` *e* actually generates two slots:
- a slot named *x* initialized to *e*
- a slot named *x*`:` initialized to the **assignment primitive**

When invoked, the assignment primitive stores its argument into the corresponding data slot

Aside from the special behavior of the assignment primitive, there's nothing special about assignment slots or assignment messages

Making new objects

Can make new objects by either:

- evaluating an object constructor expression like
  `(| x = some expr. y = another expr. ... |)`
- **cloning** an existing object
  - clone = shallow copy
  - clonee = the **prototype** a.k.a. the template
- not via instantiating a class!

A **primitive** unary message: `_Clone`

- (later: a nicer `clone` message in standard library)

Example:

```
val p1 = (| x <- 3. y <- 4. |).
val p2 = p1 _Clone.
p2 x: 6.
p1 y: 8.

val r1 = (| upperLeft = p1. lowerRight = p2. |).
val r2 = r1 _Clone.
r1 upperLeft x: 10.
r2 upperLeft x        → ??
```


Methods with arguments

A method object can specify that it takes an argument by declaring an **argument slot** of the form `:name`

- no initializer; value provided by the caller

Example:

```
val aPoint =
  (| x <- 3.
     y <- 4.
     + = (| :p | (_Clone x: x + p x) y: y + p y ).
     distanceTo: = (| :arg |
       ((x - arg x) squared +
        (y - arg y) squared) sqrt ).
  |).
aPoint distanceTo: (| x=5. y=7. |)    → 3.61
```

Syntactic sugar: put arg name with slot name, e.g.:

```
val aPoint =
  (| ...
     + p = ( (_Clone x: x + p x) y: y + p y ).
     distanceTo: arg = ( (...+...) sqrt ).
  |).
```


Multiple arguments

A method can take more than one argument;
just declare more than one argument slot, in order

A keyword message can have multiple keywords,
each followed by a corresponding argument expression

Example message:

```
aPoint copyX: newX Y: newY
```

- sends the `copyX:Y:` message to `aPoint`,
  with `newX` and `newY` as arguments

Example method definition:

```
val aPoint =
  (| ...
     copyX:Y: = (| :newX. :newY. |
       (_Clone x: newX) y: newY ).
  |).
```

Alternatively, using syntactic sugar:

```
val aPoint =
  (| ...
     copyX: newX Y: newY = (
       (_Clone x: newX) y: newY ).
  |).
```


Keyword message ambiguity

What should this parse as?

```
r1 upperLeft: aPoint copyX: x Y: y
```

What messages are being sent?

- `upperLeft:`, `copyX:`, and `Y:`
- `upperLeft:copyX:Y:`
- `upperLeft:` and `copyX:Y:`
- ...

Smalltalk-80: make longest message possible

- `upperLeft:copyX:Y:`
- bad choice if instance variable assignments via messages

Self: uncapitalized starts a message, capitalized continues; right-associative

- `upperLeft:` and `copyX:Y:`
- `r1 upperLeft: (aPoint copyX: x Y: y).`

A scenario

We define the first point as

```
val aPoint =
  (| x <- 3.
     y <- 4.
     + p = ( ... ).
     distanceTo: arg = ( ... ).
     copyX: newX Y: newY = ( ... ).
     "lots of other methods for points"
  |).
```

Then we make lots of other points via cloning

```
... aPoint _Clone ... p1 + p2 ...
```

Then we want to add a new method to all points

- but how?

Inheritance

Support sharing of common features across objects via inheritance

Put shared slots (e.g. methods) in one object

- conventionally called a **traits** object

Put object-specific slots (e.g. data slots) into another object

- conventionally called a **prototype** object

Have the prototype *inherit* from its traits object, via a **parent slot**

- mark a slot as a parent slot by putting * after its name
- initialize the parent slot to the object being inherited from

Clone the prototype to make new "instances"

- changes to the traits object now shared by all instances
- common code can be factored across many kinds of objects

Traits themselves can have parents, and so on

- leads to usual rich inheritance hierarchies

Example

```
val pointTraits =
  (| + p = ( ... ).
     distanceTo: arg = ( ... ).
     copyX: newX Y: newY = ( ... ).
     "lots of other methods for points"
  |).

val pointProto =
  (| x <- 0.
     y <- 0.
     parent * = pointTraits.
  |).

val p1 = (pointProto _Clone x: 3) y: 4.
```


Message lookup, revisited

If send a message *msg* to a receiver object *rcvr*:

- if *rcvr* has a slot named *msg* with contents *v*:
  - if *v* is the assignment primitive, assign the message's argument to the corresponding data slot
  - if *v* has code, invoke it
  - otherwise, return *v*
- otherwise, look for a slot marked as a parent
  - if find one, recursively search the object referenced by the parent slot
  - (later: what to do if find more than one)
- otherwise, report a message-not-understood error

If invoke a method object:

- clone the method object, to create an **activation record**
- initialize the clone's formal argument slots with the message's actual argument objects
- evaluate the expression(s) in the method's body
- return the result of the last expression

self

`self` is an implicit argument slot of every method

To what should it be initialized when the method is invoked?

- The original receiver of the message, *rcvr*?
- The object containing the method slot?
- Something else?

Example:

```
val pointTraits =
  (| + = (| ":self." :p. |
              ... "self" x + p x ... ).
  |).

val p1 =
  (| parent* = pointTraits. x <- 3. y <- 4. |).

p1 + p1
```


Local slots

Methods can declare additional slots (in addition to arg slots)
These act as local variables

Example:

```
val pointTraits =
  (| ...
     reflect = ( | temp <- 0 |
       temp: x.
       x: y.
       y: temp.
       self ).
  |).
```

Initializer of mutable data slot can be omitted

- defaults to nil

Example:

```
val pointTraits =
  (| ...
     reflect = ( | temp | ... ).
  |).
```


Accessing local slots

Access local slots (both arguments and local variables) using message without explicit receiver

- receiver is implicitly `self`

But local slots won't be found if start lookup with `self` object!

Special rule for implicit-self messages:
message lookup starts with current activation record object

- allows searching local "scopes"

To allow implicit-self messages to also be able to search `self` and its ancestors, e.g. to access instance variables, the implicit `self` argument slot is a parent slot

- if a message isn't found in a local slot of the activation record, then search the receiver object, then its parents, etc.
- activation record inherits from (i.e. *refines*) the receiver!

Example, revisited

```
val pointTraits =
  (| + = (| ":self*." :p. |
            (_Clone x: x + p x) y: y + p y ).
  |).

val p1 =
  (| parent* = pointTraits. x <- 3. y <- 4. |).

p1 + p1
```

Multiple inheritance

If an object has multiple parent slots, how to do method lookup?

Many plausible options,
leading to different rules for multiple inheritance

Option 1: don't allow more than one parent slot

- i.e., single inheritance (of implementation) only

Option 2: search all parents, recursively

- 2A: return first matching slot
  - what order to search parent slots?
- 2B: return matching slot if just one parent inherits a slot, otherwise, report message-ambiguous error
  - diamond-shaped inheritance?
- 2C: return matching slot if all parents inheriting slots inherit the same one, otherwise ambiguous [*Self interpreter*]
  - children override parents?
- 2D: return matching slot of ancestor inheriting from all other ancestors with matching slots, otherwise ambiguous
  - some parents more important than others?
- 2E: allow priorities to be given to parents (e.g., via more stars); return matching slot of highest-priority parent, otherwise ambiguous

Summary, so far

Saw syntax & semantics of

- defining objects
- assignment
- sending messages
- inheritance

Didn't see:

- classes
- primitive types
- constructors
- static methods & variables vs. instance methods & variables
- globals
- special syntax for instance variable access vs. method calls
- control structures
- exceptions
- ...

Class-based vs. classless programming

What are classes for? How does Self do without?

Define methods and instance variable layouts of their instances

- in Self, each object is self-describing and self-sufficient
- Self programmers use traits as a programming idiom to share methods (and data)
  - sharing instance variable layout isn't supported as well

Define static methods and variables

- Self programmers can define other objects, separate from traits, to hold these methods and variables

Define constructors

- Self programmers can define "creating" methods, using _Clone, in the objects holding other static methods

Allow inheritance from other classes

- in Self, objects inherit directly from other objects

Benefits of classless model

Much simpler language!

Avoids **meta-regress problem** and **metaclasses**

In Smalltalk:

- every object has a class
- every class is an object
- what's the class of a class? what's its class? etc.

Singleton (one-of-a-kind) objects are natural

- no special design pattern or other hacks needed

Objects can inherit interesting state from other objects

An object's parent slot can be mutable,
allowing its parent to be changed at run-time

- called **dynamic inheritance**
- elegant alternative to e.g. chain-of-responsibility & strategy patterns

Benefits of uniform messaging

Traditionally, instance variables and methods are accessed differently

Self accesses them both via messages
- difference is in target slot, not form of message

Benefits:
- hide implementation choice from external clients
  - can easily change between stored state and computed results
- inheriting objects can make different choices
  - can override data with code, and vice versa

Self's syntax makes messages just as concise as variable accesses
- benefits regular messages, too

C# properties are a clumsy version of uniform messaging

Benefits of uniform objects

Primitive values (`3`, `true`, `nil`, `'hi'`) are first-class objects
- inherit from predefined traits objects for their behavior
- programmers can add their own methods to these traits

Manipulated by sending them messages, just like other objects

To allow "natural" notation, infix operators are allowed, treated as binary messages
- programmers can define their own operator messages
  - but not precedence
- subsumes "operator overloading" of other languages

C# "auto-boxing" of primitives crudely approximates primitives as first-class objects

First-class functions

Self supports first-class, lexically-nested, anonymous functions called **blocks**
- blocks are objects, inheriting from their own predefined traits object

Written like a method, but using `[...]` instead of `(...)`

Examples:

```
[ 'hi there' printLine. ]
[| :arg1. :arg2. | arg1 + arg2 ]
```

Block doesn't execute until invoked

Invoke a block by sending it a `value` message with the right number of arguments
- 0 arguments: send `value`
- 1 argument: send `value:`
- 2 arguments: send `value:With:`
- 3 arguments: send `value:With:With:`
- etc.

Lexical scoping

Blocks can be written inside other methods (or blocks)

Nested block can access local slots of lexically enclosing method/block

Implemented by storing a reference to the lexically enclosing activation record when the block object is created (a closure!)
- when the block is invoked, the reference is stored into a parent slot in the block method's activation record
- (no `self` slot in a block activation record; `self` inherited from lexically enclosing method)

Lexical scoping is just inheritance!

Control structures using blocks

Self has *no* built-in control structures!

Instead, dynamically dispatched methods and blocks are used to program control structures

Example:

```
val true = (|
  parent* = boolTraits.
  ifTrue: trueBlock False: falseBlock = (
    trueBlock value ).
|).
val false = (|
  parent* = boolTraits.
  ifTrue: trueBlock False: falseBlock = (
    falseBlock value ).
|).

...
(x != 0) ifTrue: [ y/x ] False: [ -1 ]
```

Many methods in `boolTraits`, in terms of `ifTrue:False:`

More basic control structures

Implemented in `boolTraits`:

```
test ifTrue: [trueCode]
test ifFalse: [falseCode]

test and: [test2]
test or: [test2]
```

Implemented in `blockTraits`:

```
[body] loop
[test] whileTrue: [body]
[body] untilFalse: [test]
```


Iterators

ML provides higher-order operations over lists, e.g. map, fold

But OO languages offer many different kinds of collections

- lists, strings, association lists, ...

Provide family of iterator methods for each kind of collection

Most basic iterator: *`coll`* `do: [|:`*`elem`*`| ...]`
E.g.: `lst do: [|:elem| elem printLine.].`

Others:

- *`coll`* `doWithIndexes: [|:`*`index`*`. :`*`elem`*`.| ...]`
- *`coll`* `mapBy: [|:`*`elem`*`| ...]`

For association lists:

- *`assocLst`* `keysAndValuesDo: [|:`*`key`*`. :`*`value`*`.| ...]`

Non-local returns

In some cases, want to exit early from a loop

- e.g. `break` or `return` inside loop body

In Self, implement all early exiting using **non-local returns**

- `^` *`expr`* as last statement in a block's body
- causes immediate return from lexically enclosing method

Example:

```
val collectionTraits = (|
  ...
  includes: elem = (
    do: [|:e|
      e == elem ifTrue: [
        "we found it, so exit loop now, returning true"
        ^ true].
    ].
    "didn't find it, so return false"
    false ).
|).
```

A limited form of `call/cc` or exception

Exceptions using blocks

Use blocks in place of exceptions

- in place of handler, client passes in a block
- in place of throw, callee invokes block

Example:

```
val assocListTraits = (| ...
  at: key IfAbsent: absentBlock = (
    keysAndValuesDo: [|:k.:v.|
      k == key ifTrue: [
        "we found it, so exit loop now, returning value"
        ^ v].
    ].
    "didn't find it, so invoke and return absent block"
    absentBlock value ).
|).
```

Client can control whether exception is terminal or not:

```
... table at: key IfAbsent: [defaultValue] ...
... table at: key IfAbsent: [^ result] ...
... table at: key IfAbsent: [error: '...'] ...
```


Top-level environment

When running the read-eval-print loop, "implicit self" is the distinguished `globals` object

- represents the top-level global namespace
- its slots are the global names

By convention, all traits objects inherit (directly or indirectly) from `objectTraits`

- defines `clone`, `==`, `printString`, `printLine`, ... methods

`objectTraits` in turn inherits from `globals`

- gives all (normal) code access to global names
  - again, inheritance for lexical scoping!

Programming primitives

Self includes primitives to add or change slots of objects

obj `_AddSlots: (|` *slots* `|).`

- evaluates *obj* and *slots*, then adds all the *slots* to *obj*
  - changes any that already exist in *obj*
  - leaves any other slots in *obj* alone

obj `_AddSlotsIfAbsent: (|` *slots* `|).`

- evaluates *obj* and *slots*, then adds any new *slots* to *obj*
  - leaves alone any slots that already exist in *obj*

obj `_DefineSlots: (|` *slots* `|).`

- evaluates *obj* and *slots*, then makes *obj* contain only *slots*
  - any other slots in *obj* removed

No `val` declaration in Self; instead use these primitives, e.g.

```
globals _AddSlotsIfAbsent: (|
  pointTraits = (). |).
pointTraits _DefineSlots: (| ... |).
```