Object-Oriented Programming

Abstract Data Types ...

- package representation of data structure together with operations on the data structure
- encapsulate internal implementation details

+ **Inheritance** ...

- support defining new ADT as incremental change to previous ADT(s)
- share operations across multiple ADTs

+ **Dynamic Binding**

- run-time support for selecting right implementation of operation, depending on ADT argument(s)

Some OO languages:

- Simula 67: the original
- Smalltalk-80: popularized OO
- C++: OO for the hacking masses
- Java: OO for the webbing masses
- CLOS: powerful OO part of Common Lisp
- Emerald, Kaleidoscope, Cecil: OO languages from UW

Abstract data types

User-defined data structures along with user-defined operations

- hide implementation & representation from clients
  ⇒ allow it to be changed w/o affecting clients

Called a **class** in OO languages

- data structures called **objects** & **instances** of a class
- operations called **methods**; data called **instance variables**

Example: a stack class

Interface:

```
class Stack[T] {
  push(item:T):void
  pop():T
  top():T
  size():int
}
```

Client code:

```
s:Stack[int] := new Stack[int];
s.push(3); s.push(5);
print(s.pop());
```


An implementation:

```
class Stack[T] {
 private:
  items:array[10] of T;
  top:int := 0;
 public:
  push(item:T):void {
    items[top] := item; top := top + 1; }
  pop():T {
    top := top - 1; return items[top]; }
  size():int { return top; }
}
```

A better implementation:

```
class Stack[T] {
 private:
  items:list[T] := nil;
 public:
  push(item:T):void { items.add_first(item); }
  pop():T { return items.remove_first(); }
  size():int { return items.length(); }
}
```

Can upgrade implementation without modifying client code

Benefits of ADTs

Support good specification of interface to ADT,
hiding distracting implementation details

Hide aspects of implementation of ADT,
prevent undesired dependencies between client and ADT

Allow language to be extended with new types,
raising & customizing the level of the language

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 {
 private:
  center:Point;
  height, width:int;
 public:
  area():int { return height * width; }
  draw(screen:OutputDevice):void { ... }
  move(new_c:Point):void { center := new_c; }
  ...
}

class ColoredRectangle inherits Rectangle {
 private:
  color:Color;   // center, height, & width inherited
 public:
  // area, move, etc. inherited
  draw(screen:OutputDevice):void { ... }
}

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


Benefits of inheritance

Achieve more code sharing by **factoring** code into common superclass

- superclass can be **abstract**
- 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

Tracing control logic of code is harder

Simple extension & overriding may be too limited

- e.g. exceptions in classification hierarchies

Subclass polymorphism and dynamic binding

Subclass polymorphism:
allow instance of subclass to be used wherever instance of superclass expected

- code written for superclass reused for all subclasses

```
r :Rectangle := ...;
cr:ColoredRectangle := ...;
...
r := cr;     // legal, even though different types
```

Dynamic binding:
when invoke operations on object,
invoke appropriate operation for *dynamic* class of object,
not *static* class/type

```
r.draw();    // invokes ColoredRectangle draw
```

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

Method lookup

Given a message `obj.msg(args)`

Start with run-time class `C` of `obj` (the **receiver**)

- if `msg` is defined in `C`, then invoke it
- otherwise, recursively search in superclass of `C`
- if never find match, report run-time error
  ⇒ type checker guarantees this won't happen

Dynamic dispatching vs. static overloading

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: regular static overloading on arguments, too
  - CLOS, Cecil: resolve using all arguments
    (**multiple dispatching**)

Example: displaying shapes in a list

Without dynamic binding, use "typecase" idiom:

```
forall s:Shape in scene.shapes do
  if s.is_rectangle() then
    rectangle(s).draw();
  else if s.is_square() then
    square(s).draw();
  else if s.is_circle() then
    circle(s).draw();
  else
    error("unexpected shape");
  end
end
```

With dynamic binding, use message:

```
forall s:Shape in scene.shapes do
  s.draw();
end
```

What if add new `Shape` subclasses to library?

Benefits of dynamic binding

Allows subclass polymorphism and dynamic dispatching to class-specific methods

Allows new subclasses to be added without modifying clients

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

Pitfalls of dynamic binding

Makes logic of program harder to follow

Adds run-time overhead

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

Issues in object-oriented programming

Basic object model:

- hybrid vs. pure OO languages
- class-based vs. classless (prototype-based) languages
- single dispatching vs. multiple dispatching
- single inheritance vs. multiple inheritance
- the Beta model: nested, virtual classes

Static type checking:

- separating types from classes
- subtype-bounded polymorphism

Cecil

Purely object-oriented language

- all data structures through user-defined objects
  - **classless** object model
  - type safe, garbage collected, implicit pointers, ...
- all operations through user-defined methods
  - methods can dispatch on 0, 1, or several arguments
    $\Rightarrow$ **multimethods**
- all instance variables through user-defined fields
- all control structures through user-defined code manipulating first-class, lexically-nested, anonymous closure objects
- invoke methods, fields, constructors, & closures uniformly through messages

Optional polymorphic static type checking, with F-bounded and signature-bounded polymorphism

- no type inference
- (multiple) inheritance separate from multiple subtyping

Module system (only partially implemented)

Rich standard library

Procedures and variables

Use `method` to define regular procedures

- last expression returned
- can overload for different numbers of arguments

Use `let` to define local and global variables

- add `var` keyword to allow assignment, otherwise immutable
- must initialize at declaration

```
let var count := 0;
method foo(a, b, c) {
  count := count + 1;
  let var d := a + b;
  let e := frob(d, c);
  d := d + e;
  d + 5 }
method frob(x, y) { x - frob(y) + 1 }
method frob(x) { - x / 5 }
```


Closures: first-class functions

Code bracketed in braces is a 0-argument function value (called a closure in Cecil-speak)

```
let closure := { factorial(10) + 5 };
```

Evaluation of closure body delayed until invoked by `eval`:

```
eval(closure)    → 3628805
```

To allow arguments to closure, add `&(x,y,z)` prefix; invoke passing extra arguments to `eval`:

```
let closure2 := &(n){ factorial(n) + 5 };
...
eval(closure2, 10)  → 3628805
```

Just like `lambda`, `fn`, `\`:

- anonymous
- lexically scoped
- first-class

Glitch: returning closures

In current Cecil implementation, by default, closures cannot safely be returned out of their lexically enclosing scope

- a glitch in the Vortex implementation, not the Cecil language
  - can crash Vortex mysteriously
- prevents currying, `compose`, closures in data structures, ...

To be able to do this, use `&&` rather than `&`:

```
method add_x(x) { &&(y){ x + y } }

let add_2 := add_x(2);
let add_5 := add_x(5);

eval(add_2, 4)    → 6
eval(add_5, 4)    → 9
```


Using closures in control structures

All traditional (and many non-traditional) control structures implemented as regular Cecil procedures, with closures passed by callers supporting the necessary lazy evaluation

For simple lazy or repeated evaluation:

```
if(test, { then_value }, { else_value })
test1 & { test2 }
while({ test }, { body })
```

For iteration with arguments:

```
for(start, stop, &(i){ body })
do(array, &(elem){ body })
do_associations(table, &(key,value){ body })
```

For exception handling:

```
fetch(table, key, { if_absent })
```

For continuation-passing style code:

```
compare(i, j, { if_lt }, { if_eq }, { if_gt })
```

An example

```
-- this is a factorial method
method factorial(n) {
    if(n = 0,
       { 1 },
       { n * factorial(n - 1)}) }

-- call factorial here:
factorial(7)
```


Non-local returns

Support exiting a method early with a non-local return from a nested closure

- like a `return` statement in C
- like a limited kind of continuation in Scheme

```
{ ...; ^ result }
{ ...; ^ }
```

Example:

```
method fetch(table, key, if_absent) {
  table.do_associations(&(k, v){
    if(k = key, { ^ v });
  });
  eval(if_absent) }
method fetch(table, key) {
  fetch(table, key, {
    error("key " || print_string(key) ||
           " not found ") }) }

fetch(zips, "Seattle", { 98195 })
```


Objects, methods, and fields

To define a data structure, use `object` declaration

- to instantiate, use `object isa` expression

To add methods to an object, *specialize* the method to the object by adding `@Object` after the first argument (the receiver argument)

To add instance variables, use `field` declarations

- add `var` keyword to allow assignment, otherwise immutable
- fields can be given default initial values at declaration
- fields can be given initial values at instantiation

```
object Point;
var field x(p@Point) := 0;
var field y(p@Point) := 0;
method area(p@Point) { p.x * p.y }
method shift(p@Point, dx, dy) {
  p.x := p.x + dx;
  p.y := p.y + dy; }
method new_point() {
  object isa Point }
method new_point(x0, y0) {
  object isa Point { x := x0, y := y0 } }
```


Overloaded methods and dynamic dispatching

Can overload methods two ways:

- same name but different numbers of arguments
- same name & number of arguments, but different specializer objects

```
method area(p@Point) { p.x * p.y }
...
method area(c@Circle) { pi * square(c.radius) }
```

Specializer-based overloading resolved by using run-time class of receiver argument
(a.k.a. dynamic dispatching, message sending)

```
method print_area(x) {
  print(area(x));
}
let var p := new_point(3,4);
print_area(p); -- invokes Point's area method
    → 12
p := new_circle(5);
print_area(p); -- invokes Circle's area method
    → 78.539818
```

Fields accessed by messages

Field declarations implicitly produce 1 or 2 accessor methods:

- get accessor:
  given object, return field contents
- set accessor (for `var` fields):
  given object & new field contents, modify field

Manipulate field contents solely by invoking these methods

```
var field x(p@Point) := 0;
  ⇒
    method x(p@Point) {
      ... fetch p.x's contents, initially 0 ...}
    method set_x(p@Point, new_value) {
      ... update p.x to be new_value ... }
-- increment p.x:
set_x(p, x(p) + 1)
```

For syntactic convenience,
any call can be written using dot notation:

```
p.x              ⇔  x(p)
p.x := p.x + 1   ⇔  set_x(p, x(p) + 1)
p.shift(3,4)     ⇔  shift(p, 3, 4)
```

Infix & prefix operators (e.g. +) are really messages, too

Inheritance

Make new ADTs from old ones via `isa` inheritance link

- child/parent, a.k.a. subclass/superclass
- inherit all method & field declarations
- can add new fields & methods, specialized on child object
- can override fields & methods

```
object ColoredPoint isa Point;
-- inherit all @Point fields and methods
-- add some new ones:
field color(@ColoredPoint);
method new_colored_point(x0, y0, c0) {
  object isa ColoredPoint {
    x := x0, y := y0, color := c0 } }

let p := new_colored_point(3,4,"Blue");
print(p.color);  → "Blue"
p.shift(2,-2); -- invoke inherited Point method
print(p.x);      → 5
```


Overriding of methods

Parent and child objects can define overloaded methods

If both apply to a call, child's takes precedence
(a.k.a. overrides the parent's)

```
method draw(p@Point) {
  Display.plot_point(p.x, p.y);
}

method draw(p@ColoredPoint) {
  Display.set_color(p.color);
  Display.plot_point(p.x, p.y);
}

let var p := new_point(3,4);
p.draw; -- only Point's draw applies

p := new_colored_point(5,6,"Red");
p.draw; -- both draws apply, but
        -- ColoredPoint's overrides Point's
```


Resends

Often, overriding method includes overridden method as a subpiece

Can invoke overridden method from overriding method using `resend` (called `super` in some other languages)

Without `resend`:

```
method draw(p@Point) {
  Display.plot_point(p.x, p.y);
}
method draw(p@ColoredPoint) {
  Display.set_color(p.color);
  Display.plot_point(p.x, p.y);
}
```

With `resend`:

```
method draw(p@Point) {
  Display.plot_point(p.x, p.y);
}
method draw(p@ColoredPoint) {
  Display.set_color(p.color);
  resend;
}
```

Overriding of fields

Since fields accessed through accessor methods, can override accessor methods with regular methods, & vice versa

```
object Origin isa Point;
method x(@Origin) { 0 }
method y(@Origin) { 0 }
```

Because fields accessed through messages, like methods, clients can't tell how message implemented

- can differ in different child objects
- can change through program evolution & maintenance

```
let p := ...; -- Point or Origin object
print(p.x);   -- how is x implemented?
```


Factoring

Methods written in parent object can be reused via inheritance in child objects

Behavior of inherited methods can be customized for children by invoking operations on receiver formal argument

Example:
shared `area` method depends on `x` & `y` implementation

```
method area(p@Point) { p.x * p.y }
var field x(p@Point);
var field y(p@Point);
method x(@Origin) { 0 }
method y(@Origin) { 0 }
```

Another example:
code in `Point` that calls `draw` will do colored drawing when invoked on `ColoredPoint`s

Customizing like this is key to effectiveness of OO

Abstract objects

With factoring, can introduce objects whose sole purpose is to be inherited from, not directly used or instantiated

- called **abstract** as opposed to **concrete** objects/classes
- may only be partially implemented;
  may call **abstract methods** that should be defined for concrete children

```
abstract object Point;

  -- [abstract method isn't legal Cecil]
  abstract method x(p@Point);
  abstract method y(p@Point);
  abstract method rho(p@Point);
  abstract method theta(p@Point);

  method area(p@Point) { p.x * p.y }
  method distance_to_origin(p@Point) { ... }
  method rotate(p@Point, amount) { ... }
  ...
```


Concrete implementations

```
template object CartesianPoint isa Point;
  field x(p@CartesianPoint) := 0;
  field y(p@CartesianPoint) := 0;
  method rho(p@CartesianPoint) { ... }
  method theta(p@CartesianPoint) { ... }
  method new_cartesian_point(x0, y0) {
    concrete object isa CartesianPoint {
      x := x0, y := y0 } }

template object PolarPoint isa Point;
  field rho(p@PolarPoint) := 0;
  field theta(p@PolarPoint) := 0;
  method x(p@PolarPoint) { ... }
  method y(p@PolarPoint) { ... }
  method new_polar_point(rho0, theta0) { ... }

concrete object Origin isa Point;
  method x(@Origin) { 0 }
  method y(@Origin) { 0 }
  method rho(p@Origin) { 0 }
  method theta(p@Origin) { 0 }
```

Object roles

Can declare whether an object is
`abstract`, `template`, or `concrete`

`abstract`:

- potentially incomplete
- can be inherited from
- cannot be used directly or instantiated

`template`:

- complete
- can be inherited from
- can be instantiated
- not to be used directly

`concrete`:

- complete
- can be inherited from
- can be instantiated
- can be used directly

Classless object model

Most OO languages distinguish objects/instances from classes, distinguish subclassing from instantiation

Cecil doesn't: **classless** (a.k.a. **prototype-based**) object model

Individual objects have their own implementation directly

- methods specialized directly on object, not separate class thing
- objects with methods/fields specializing on them act like classes

Individual object can inherit behavior from other objects

- if no additional customization, then act like an instance
- if new/overriding implementation, then act like a subclass

Class-like objects can be used directly as instances, too

- e.g. the `Origin` object
- useful for constants, enumerated values, `true`, `false`, etc.

Use object creation expression w/ optional initialization instead of special constructor operation

A class-based model

If have explicit classes, typically:

- put field and method declarations with classes
  - class methods vs. instance methods, class fields vs. instance fields
- `new` *C*: creates instances of class *C*
- constructor methods in class called by `new`
- `this`/`self` to refer to implicit receiver of instance methods

```
class Point {
  var field x := 0;
  var field y := 0;
  method area { x * y }      -- self.x * self.y
  method shift(dx, dy) {
    x := x + dx;       -- self.x := self.x + dx;
    y := y + dy; }
  constructor(x0, y0) {
    x := x0; y := y0 }
  class field the_origin := new Point;
  class method origin() { the_origin }
}
let p := Point.origin();
p.shift(3,4);
```


Metaclasses

In pure, class-based OO languages (e.g. Smalltalk):

- every thing is a first-class object
- every object has a class

⇒ every class has a class

What's the class of a class? a **metaclass**

- stores the methods to handle messages sent to the class (e.g. `new`, other class methods)

What's the class of a metaclass?

Smalltalk-76 and Smalltalk-80 use different mechanisms on how to structure the metaclass instantiation hierarchy & break the infinite regress of metaclasses

- Smalltalk-80 more expressive (allows interesting class methods), but more complicated structure
- metaclasses are hardest part of ST to learn
  ⇒ led in part to development of classless languages

To inherit or not to inherit

Inheritance works well when

- subclass supports a superset of operations of superclass
  ⇒ can use subclass where superclass expected
- subclass reuses much of implementation of superclass
- subclass's representation extends rep of superclass
- subclass is a special kind of superclass, conceptually
  - subclass "is a" superclass

Inheritance is not appropriate when:

- class has another class as a component
  - class "has a" class
  - interfaces are not related
  - use a slot instead
- only part of other class's implementation is reused
- representation of other class needs to be altered

If in doubt, don't inherit,
or reorganize class hierarchy to make inheritance work better

Inheritance vs. forwarding

Forwarding:

- when invoke operation on object *A*, forward the operation to component object *B*

This is not inheritance:

- true inheritance remembers the original object,
  so that "sends to self" work (go back to *A*, not *B*)

Encapsulation

How to hide internal implementation details?

Design for Cecil:

- wrap declarations in a `module`
- annotate declarations as `public` or `private`
- `import` other modules, see only public decls

```
module PointMod {
  object Point;
  public get private set var field x(@Point);
  public get private set var field y(@Point);
  public method new_point(x0,y0) { ... }
  private method ...
}
module ShapeMod {
  public import PointMod;
  private import ShapeHelperMod;
  public object Shape;
  public field center(s@Shape);
  ...
}
```


Object/class-based encapsulation

An ADT-oriented alternative: `object` declaration is module

```
object Point {
  public get private set var field x;
  public get private set var field y;
  public constructor(x0,y0) { ... }
  private method ...
}
object Shape {
  public field center;
  protected field color;
  ...
}
object ShapeHelper {
  friend Shape;
  ...
}
```

`protected`: visible to subclasses but not regular clients

- hard to simulate in module-based encapsulation

`friend`: list objects that have private access

Not clear what analogue of `import` is