Object-Oriented Programming

1

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
- + **Subclass Polymorphism**
  - allow variables to hold instances of different ADTs
- + **Dynamic Binding**
  - run-time support for selecting right implementation of operation, depending on argument(s)

2

Some OO languages

- Simula 67: the original
- Smalltalk-80: popularized OO
- C++: OO for the hacking masses
- Java, C#: cleaned up, more portable variants of C++
- CLOS: powerful OO part of Common Lisp
- **Self**: very pure OO language
- **Cecil**, MultiJava, **EML**: OO languages from my research group
- Emerald, Kaleidoscope: other OO languages from UW

3

Abstract data types

- User-defined data structures along with user-defined operations
  - Support good specification of **interface** to ADT, hiding distracting implementation details
  - Prevent undesired dependencies between client and ADT, allowing implementation to change w/o affecting clients
  - Allow language to be extended with new types, raising & customizing the level of the language
- Called a **class** in OO languages
  - data structures called **objects**, or **instances** of the class
  - operations called **methods**; data called **instance variables**
- Modules have similar benefits

4

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

5

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
  void draw(OutputDevice out) { ... }  // override!
}
```

6

Benefits of inheritance

- Achieve more code sharing by **factoring** code into common superclass
  - superclass can be **abstract**
    - no direct instances, just reusable unit of implementation
  - encourages development of rich libraries of related data structures
- May model real world scenarios well
  - use classes 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 real-world classification hierarchies

Subclass polymorphism

- Allow instance of subclass to be used wherever instance of superclass expected
  - client code written for superclass also works/is reusable for all subclasses

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

ColoredRectangle cr = …;
… client(cr) …
// legal, because ColoredRectangle is a subclass of Rectangle

// but what version of draw is invoked?
```


Dynamic binding

- When invoke operations on object, invoke appropriate operation for *dynamic* class of object, not *static* class/type

```
ColoredRectangle cr = … ;
Rectangle r = cr;  // OK, because CR subclass of R
r.draw(); // invokes ColoredRectangle::draw!
```

- 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, MultiJava: resolve using all arguments (**multiple dispatching**)

Example

- Without dynamic binding, use "typecase" idiom:

```
forall Shape s 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, send message:

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

- What happens if a new `Shape` subclass is added?

13

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
  - "Template method" design pattern

14

Pitfalls of dynamic binding

- 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 only an array lookup (or equivalent), not a search

15

Issues in object-oriented language design

- Object model:
  - hybrid vs. pure OO languages
  - class-based vs. classless (prototype-based) languages
  - single inheritance vs. multiple inheritance
- Dispatching model:
  - single dispatching vs. multiple dispatching
- Static type checking:
  - types vs. classes
  - subtyping
  - subtype-bounded polymorphism

16

Self

17

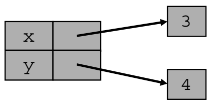
Self

- A purely object-oriented language, developed as a Smalltalk successor in 1986/7
  - Every thing is an object
    - including primitives like numbers, booleans, etc.
    - no classes, only prototypes
    - first-class functions (aka blocks) are objects
    - even methods are objects
  - Every action is a message
    - operations on primitives
    - control structures
    - access & assignment to instance variables
  - Scoping is inheritance
- Theme: simplicity (uniformity) yields power

18

Self objects

- An object is just a list of *slots*
  - A slot is a key/value pair
  - The contents of a slot is (a reference to) another object
- Example: `(| x = 3. y = 4. |)`



Accessing slots

- The only thing you can do with an object is send it a message
- To fetch the contents of an object's slot, send the slot's name as a message to the object
- Example:

```
let aPoint = ( | x = 3.  y = 4. | )
aPoint x   "send x to aPoint, yielding 3"
```


Methods

- A method is just a special kind of object stored in a slot
  - Special because it has code that runs when it's looked up in a slot
- Example:

```
let aPoint =
  ( | x = 3.
      y = 4.
      distanceToOrigin = (
          (self x squared + self y squared) sqrt )
    | )
aPoint distanceToOrigin   "yields 5"
```


Syntax of messages

- Unary messages: a simple identifier written *after* the receiver
  - right-associative
  - `aPoint` **`distanceToOrigin`**
  - `self` **`x`**
  - `self x` **`squared`**
  - `(…)` **`sqrt`**
- Binary messages: punctuation symbol(s) written *between* its two arguments
  - any sequence of punctuation symbols allowed; user-defined operators J
  - lower precedence than unary messages
  - all binary messages have same precedence and are left-associative L
  - `x squared` **`+`** `y squared`
  - `3 + 4 * 5` yields `35` L
- Keyword messages: later…

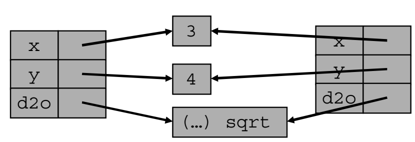
Sends to self

- In a method, the name of the receiver of the message is `self`
- If no receiver specified, then it's implicitly `self`
- E.g.
  - `self x squared` can be written `x squared`
  - `distanceToOrigin = (`
    `(x squared + y squared) sqrt )`
- Makes method calls as concise as (traditional) instance variable access

Making new objects

- Can make new objects by:
  - writing them down explicitly (as we've done), or
  - *cloning* an existing object (the *prototype*)
    - a shallow copy of the object
- Example:

```
let otherPoint = aPoint clone.
```



Mutable slots

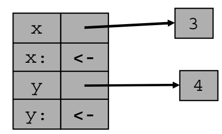
- Slots initialized with `=` are immutable
- Slots initialized with `<-` are mutable
- To change a slot named `x`, send the object the `x:` message with the new value
  - returns the receiver, e.g. for additional updates
- Example:

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

25

Assignment slots

- When a mutable slot named `x` is declared, two slots are created in the object:
  - one named `x` referring to the slot's (current) contents
  - one named `x:` referring to the assignment primitive
- Example: `(| x <- 3.  y <- 4. |)`



26

Keyword messages

- A keyword message is an identifier followed by a colon
  - It takes an argument after the colon
  - `aPoint` **`x:`** `5`
    - The message is **`x:`**
    - The receiver is (the result of evaluating) `aPoint`
    - The argument is (the result of evaluating) `5`
- Also have keyword messages that can take more than one argument (later…)

27

Methods with arguments

- A method object can take one or more arguments by declaring slots whose names begin with colons
  - One argument slot for each argument that can be accepted according to the slot name
    - 1 for binary messages
    - 1 or more for keyword messages
- Example:

```
(| …
   + = (| :p | (clone x: x + p x) y: y + p y )
 |)
```

- Shorthand: put argument name in slot name

```
   + p = ( (clone x: x + p x) y: y + p y )
```

28

A scenario…

- We define the first point as

```
let aPoint =
  ( | x = 3.
      y = 4.
      distanceToOrigin = ( … )
      + p = ( … )
      …  "lots of other methods on points"
    | )
```

- Then we make lots of other points via cloning
  `… aPoint clone … p3 + p9 …`
- Then we want to add a new method to all points
  - how?

29

Inheritance

- Introduce sharing through inheritance
- Put shared slots (e.g. methods) into one object (called a *traits* object)
- Put object-specific slots (e.g. instance vars) into another object (called a *prototype* object)
- Have the prototype *inherit* the traits
  - By adding a slot marked as a *parent* slot using an asterisk
- Clone the prototype to make new objects
  - They'll also inherit the same traits object

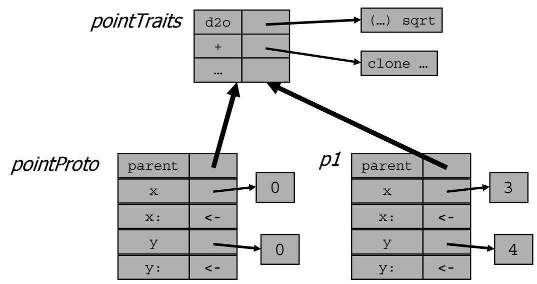
30

Example

```
let pointTraits =
   ( | distanceToOrigin = ( … )
       + p = ( … )
       …   "lots of other methods on points"
     | )
let pointProto =
   ( | x <- 0.   "default initial coordinates"
       y <- 0.
       parent* = pointTraits.   "inherit shared code"
     | )
let p1 = (pointProto clone x: 3) y: 4.
```

31

The result



32

Message lookup, revisited

- If a message *msg* is sent to an object:
  - If the object contains a slot named msg, get the object referenced by the slot
    - If it's the assignment primitive, do an assignment to the corresponding data slot
    - If it's a method, run its body and return the result (more later)
    - Otherwise, just return the contents
  - Otherwise, look for a slot marked as a parent slot
    - If found, then recursively look up the message in the object referred to by the parent slot
      (Parents can contain their own parent slots, etc.)
    - Otherwise, report a "message not understood" error

33

Invoking a method

- To run a method object:
  - Clone the method object to make a method activation object
    - a stack frame!
  - Initialize the argument slots of the cloned method to the argument objects
  - Evaluate the expression(s) in the body
  - Return the result of the last expression
- But what about `self`?

34

Self

- `self` is an implicit argument slot of every method
- What is the `self` argument?
  - The original object that received the message?
  - Or the object containing the method?
  - Or something else?
- Consider a message `p1 + p1`; in the `+` method inherited from `pointTraits`, what's `self` bound to?

35

Local slots

- Methods can have slots
  - E.g. argument slots
  - Plus regular slots which act like local variables
- Sends to implicit self actually start the message lookup in the currently executing method activation object
  - The `self` slot of the method is a parent slot, so that lookup continues to search the receiver if a message to implicit self doesn't match a local slot
    - The method activation is a *refinement* of the receiver
- Example:

```
+ = (| ":self*" :p | … x + p x … )
```

36

Multiple arguments

- To send a message with multiple arguments, use extended keyword messages
  - Interleave keyword message part with argument expressions
  - In Self, the first keyword message part must start with a lower-case letter; the rest must start with an upper-case letter
- Example:
  - `pointTraits newX: x Y: y`
    - message/slot name: `newX:Y:`
    - receiver: `pointTraits`
    - arguments: `x` and `y`
  - `pointTraits = (| newX: x Y: y = ( … ). … |)`

Summary, so far

- Saw syntax & semantics of declaring objects, sending messages, inheritance, assignment
- Didn't see classes…
- Didn't see constructors…
- Didn't see static methods & variables vs. instance methods & variables…
- Didn't see different syntax for accessing instance variables vs. calling methods…

What do classes usually offer?

- Can define the methods and instance variables of its instances
  - Self lets each object be self-sufficient & self-describing
  - Self programmers use shared traits objects as a way to share things across all instances of a class
    - (Doesn't work as well for instance variables)
- Can inherit from other classes
  - Self allows individual objects to inherit directly from other objects
    - Self inheritance is used for both class inheritance and class instantiation
- Can have static/class methods and instance variables
  - Self programmers can define separate objects (e.g. factories) if they want these things
- Can define constructors
  - Self programmers define regular methods which use clone to do this

Benefits of prototype model

- Self is much simpler by not having separate class and instance concepts
- Also:
  - makes singleton objects natural
  - avoids the problem of "what's the class of a class? and what's its class? and …"
    - no metaclasses
  - allows instances to inherit run-time state from other instances
  - allows inheritance to be changed at run-time, by making parent slots assignable
    - called *dynamic inheritance*

Benefits of uniform messaging

- Traditionally, instance variables and methods are accessed differently
- Self accesses them both via messages
  - Easy to change implementation from data to code or vice versa, without affecting clients
  - Easy to override data, and override code with data
  - Still syntactically concise
- C#'s attributes are a clumsy version of this

Benefits of uniform objects

- Primitive values are first-class objects
  - Inherit from predefined traits objects, etc., for their behavior
- Send them messages just like other objects
  - To make this work using expected syntax, syntax of "operations" are available to all objects
- Can add user-defined methods on them, just like other objects

First-class functions

- Self includes first-class functions as objects
  - Called "blocks"
- Written like a method, except use `[ … ]` instead of `( … )`
- Invoke a block by sending it the `value` message (or `value:` if it takes an argument, or `value:With:With:` if it takes 3 arguments)
  - `[| :arg1. :arg2 | code ]` means
    `(| value: arg1 With: arg2 = ( code ) |)`

Lexical scoping

- Blocks can be nested in methods
  - Can access slots of lexically enclosing method
- Implemented by giving block activation objects an implicit anonymous parent slot that inherits from the lexically enclosing method activation object
- Lexical scoping is just inheritance!

Control structures using blocks

- Self has *no* built-in control structures
- Instead, use objects, messages, and blocks to program them all, entirely in (extensible) user code
- Example:

```
let true =
  (| parent* = boolTraits.
     ifTrue: trueBlock False: falseBlock = (
       trueBlock value ) |)
let false = (| … falseBlock value … |)
(x < 0) ifTrue: ['neg'] False: ['non-neg']
```


Iterators

- To preserve abstraction of collections, each defines one (or more) iterator methods
- Most basic: `do:`
  - `aList do: [ |:elem| elem print. ].`
- Others: `keysAndValuesDo:`, `includes:`, `includesKey:`, `filterBy:`, `map:`, …

Example: association list

```
let assocListTraits = (|
   parent* = orderedCollectionTraits.   "lots of cool methods"
   assoc = (| key. value. next. |). "implicit <-nil"
   assocsDo: aBlock = (
       | assoc <- head |
       [assoc != nil] whileDo: [
          aBlock value: assoc.
           assoc: assoc next. ] ).
   keysAndValuesDo: aBlock = (
       assocsDo: [|:assoc|
           aBlock value: assoc key With: assoc value] ).  |).
   at: k Put: v = (   "should check for existing assoc, too"
       assocsDo: [|:assoc|
           (assoc key = k) ifTrue: [
               assoc value: v. ^self ] ].    "^ does early return"
       head: ((assoc clone key: k) value: v) next: head. ).
let assocListProto = (| parent* = assocListTraits.
                         head. |).
```


A client

```
let phoneBook = assocListProto clone.
phoneBook at: 'Sylvia' Put: '123-4567'.
phoneBook at: 'Andrei' Put: 'unlisted'.
…
phoneBook keysAndValuesDo: [|:name. :number|
  ('calling ' + name + '...').print.
  number makeCrankCall.
].
```

Top-level environment

- There's a distinguished object that's the top-level environment
- It defines or inherits slots for all "global" names, e.g. pointTraits, assocListProto, …
- A Self read-eval-print interpreter executes expressions in the context of this object
  - It's the implicit self of the read-eval-print loop

Updating existing objects

- Introduce (true) primitives to modify existing objects
  - `obj _AddSlots: (| slots |)`
    - adds *`slots`* to `obj`, replacing any that already exist
  - `obj _DefineSlots: (| slots |)`
    - like *_AddSlots:*, plus removes all others from *obj*
- No need for special `let` construct
  - `let pointTraits = (|…|)` is really `_AddSlots: (| pointTraits = (|…|) |)`

Object-Oriented Programming

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
- + **Subclass Polymorphism**
  - allow variables to hold instances of different ADTs
- + **Dynamic Binding**
  - run-time support for selecting right implementation of operation, depending on argument(s)

Some OO languages

- Simula 67: the original
- Smalltalk-80: popularized OO
- C++: OO for the hacking masses
- Java, C#: cleaned up, more portable variants of C++
- CLOS: powerful OO part of Common Lisp
- **Self**: very pure OO language
- **Cecil**, MultiJava, **EML**: OO languages from my research group
- Emerald, Kaleidoscope: other OO languages from UW

Abstract data types

- User-defined data structures along with user-defined operations
  - Support good specification of **interface** to ADT, hiding distracting implementation details
  - Prevent undesired dependencies between client and ADT, allowing implementation to change w/o affecting clients
  - Allow language to be extended with new types, raising & customizing the level of the language
- Called a **class** in OO languages
  - data structures called **objects**, or **instances** of the class
  - operations called **methods**; data called **instance variables**
- 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
   void draw(OutputDevice out) { ... }  // override!
}
```


Benefits of inheritance

- Achieve more code sharing by **factoring** code into common superclass
  - superclass can be **abstract**
    - no direct instances, just reusable unit of implementation
  - encourages development of rich libraries of related data structures
- May model real world scenarios well
  - use classes 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 real-world classification hierarchies

Subclass polymorphism

- Allow instance of subclass to be used wherever instance of superclass expected
  - client code written for superclass also works/is reusable for all subclasses

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

ColoredRectangle cr = ...;
... client(cr) ...
// legal, because ColoredRectangle is a subclass of Rectangle

// but what version of draw is invoked?
```


Dynamic binding

- When invoke operations on object, invoke appropriate operation for *dynamic* class of object, not *static* class/type

```
ColoredRectangle cr = ... ;
Rectangle r = cr;   // OK, because CR subclass of R
r.draw(); // invokes ColoredRectangle::draw!
```

- 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

61

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, MultiJava: resolve using all arguments (**multiple dispatching**)

62

Example

- Without dynamic binding, use "typecase" idiom:

```
forall Shape s 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, send message:

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

- What happens if a new `Shape` subclass is added?

63

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
  - "Template method" design pattern

64

Pitfalls of dynamic binding

- 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 only an array lookup (or equivalent), not a search

65

Issues in object-oriented language design

- Object model:
  - hybrid vs. pure OO languages
  - class-based vs. classless (prototype-based) languages
  - single inheritance vs. multiple inheritance
- Dispatching model:
  - single dispatching vs. multiple dispatching
- Static type checking:
  - types vs. classes
  - subtyping
  - subtype-bounded polymorphism

66