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 in Lisp
- Kaleidoscope, Cecil: OO languages designed here

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
- encourages development of rich libraries of related data structures
- "**reuse**"

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

Dynamic binding

Allow subclass to be used wherever a superclass is expected,
to reuse superclass's code for subclass

- subclass can be assigned to variable of superclass type

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

When invoke operations on object,
locate and invoke right operation for object

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

A new kind of polymorphism: **subtype polymorphism**

Dynamic binding requires run-time class info for each object

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

Like **overloading**:

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

Unlike overloading:

- resolve using *run-time class* of argument, not *static 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:

```
forall s:Shape in scene 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:

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

What if add new `Shape` subclasses to library?

Benefits of dynamic binding

Allows subtype polymorphism and class-specific methods

Allows new subclasses to be added without modifying clients

More important than inheritance ?

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

Static type checking:

- separating types from classes
- subtype-bounded polymorphism