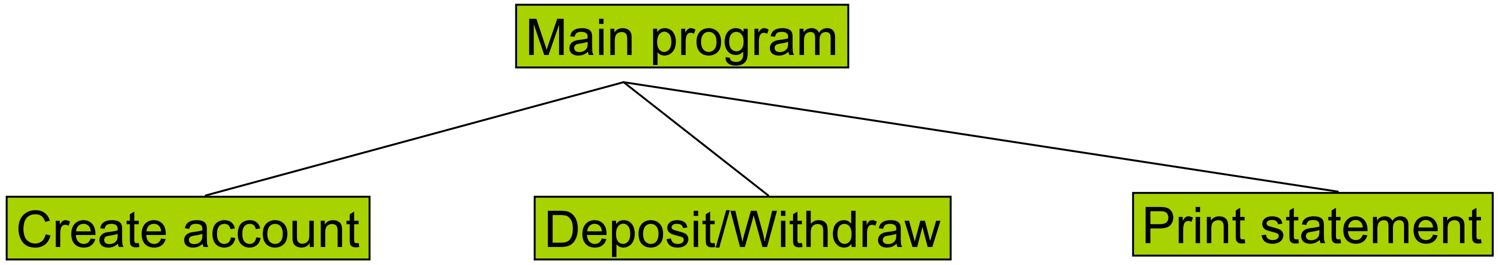
Modularity and Object-oriented Abstractions

Encapsulation, Dynamic binding, Subtyping and Inheritance

Modularity

- When we program, we try to solve a problem by
  - Step1: decompose the problem into smaller sub-problems
  - Step2: try to solve each sub-problem separately
  - Each solution is a separate component that includes
    - Interface: types and operations visible to the outside
    - Specification: intended behavior and property of interface
    - Implementation: data structures and functions hidden from outside
- Example: a banking program



Basic Concept: Abstraction

- An abstraction separates interface from implementation
  - Hide implementation details from outside (the client)
- Function/procedure abstraction
  - Client: caller of the function
  - Implementation: function body
  - Interface and specification: function declaration
  - Enforced by scoping rules
- Data abstraction
  - Client: Algorithms that use the data structure
  - Implementation: representation of data
    - Priority queue can be binary search tree or partially-sorted array
  - Interface and specification: operations on the data structure
  - Enforced by type system
- Modules
  - A collection of related data and function abstractions

Example: A Function Abstraction

- Hide implementation details of a function
  - Interface: float sqrt (float x)
  - Specification: if x>1, then sqrt(x)*sqrt(x) ≈ x.
  - Implementation details

```
float sqrt (float x){
   float y = x/2; float step=x/4; int i;
   for (i=0; i<20; i++){
     if ((y*y)<x) y=y+step;
          else y=y-step;
     step = step/2;
  }
   return y;
}
```

Example: A Data Abstraction

- Hide details of data structure (ML)

```
abstype complex = C of real*real with
  fun complex(x,y:real) = C(x,y)
  fun  x_coord(C(x,y)) = x
  fun y_coord(C(x,y)) = y
  fun add(C(x1,y1),C(x2,y2)) = C(x1+x2,y1+y2)
end
```

  - No outside operations can use C(x,y) to access internals of a complex value
  - Only data are members of abstraction
  - Access functions are global functions
    - Function names are bound in enclosing block

Modules: Combination Of Data And Function Abstractions

- General Support For Information Hiding
  - Hide implementation of related data and functions
    - Interface: a set of names and their types
      - Include both variable and function declarations
    - Implementation
      - Implementation for every entry in the interface
      - Additional declarations that are hidden
  - Can define multiple data or function abstractions
- Modules in different languages
  - ML: signatures, structures and functors (will skip)
  - C++ namespaces
  - Object-oriented abstractions
    - Java interfaces and classes; C++ classes
  - C++ templates (generic abstractions)

Global Names And Name Spaces

- Global names in C/C++
  - A name whose scope is the entire program
    - Global types, global data, global functions
  - Problems with global names
    - They might not need to be always visible and may conflict with other global names
- Namespace of global names
  - Grouping of global types, data, and functions
  - Inside namespace: use the local name
  - Outside namespace: namespace + local name
- Namespace as an abstraction
  - Interface: declarations of member variables/functions
  - Implementation: implementations of members
  - Separation of concern: file inclusion

Example: Global vs. Local names

- Java class:

```
class vehicle {
  protected: double speed =0, fuel = 0;
  public void start(double x) {speed = x;}
  public void refuel (double x) { fuel = fuel + x; }
};
vehicle a = new vehicle; a.start(5);
```

- ML abstype

```
abstype vehicle = V of real ref * real  ref with
        fun mk_vehicle() = V(ref 0.0, ref 0.0);
        fun vehicle_start (V(speed,fuel), x) = speed := x;
        fun vehicle_refuel (V(speed,fuel), x) = fuel := !fuel + x
   end;
val a = mk_vehicle(); vehicle_start(a,5.0);
```

Summary of Abstractions

- Abstractions
  - Information hiding: interface and implementation details
    - Function and data abstractions
- Modules: grouping of related data and functions
  - Types, variables, constants, functions
  - Interface: declarations visible to the outside
- Abstractions in different languages
  - ML abstype: data abstraction (hide data representation);
    - all access functions are in the global scope.
  - C++ namespaces: a group of related data and functions;
    - No explicit access control (separation through file inclusion);
    - Not a data type (cannot build values of name spaces)
  - C++/Java classes: data abstraction + module
  - What about Java interfaces? (no implementation)

Object Oriented Abstractions

- Programming methodology for building extensible systems
  - Organize concepts into objects and classes
- An OO abstraction is a data abstraction and a module
  - Is a module: a group of related data structures and functions
  - Is a data type: can be instantiated to produce objects/values
- Encapsulation (access control)
  - Separate members into interfaces and implementations
- Dynamic binding of methods (function pointers)
  - Implementations of functions are looked up at runtime
- Subtype polymorphism (relations between types)
  - Can have subtype relations with other OO abstractions
- Inheritance (inherit and modify behavior of base classes)
  - Subtype inheritance: inheriting abstraction interface
  - Implementation inheritance: inheriting method implementation

Encapsulation

- Use access control to support abstractions
  - Hide implementation details from outside
    - Implementation code: operate on data representation
    - Client code: invoke only interface operations
  - Access control: only a few functions can access private data
    - Supported by the type system of the language
    - Example: ML abstypes, C++/Java classes
- Compare to using blocks to support abstractions
  - Hide implementation detail inside each block
    - Variables can be accessed only by functions within the same block
    - Return interface functions to the outside
  - Difference: implementation

Encapsulation vs. Function Closure

- Garbage collect activation records

```
fun mk_vehicle () =
  let  val speed = ref 0.0;  val fuel = ref 0.0 in
    { start = (fn x=> speed := x),
      refuel = (fn x => fuel := !fuel + x)}
  end;
```

- Object oriented encapsulation

```
class vehicle {
    private double speed =0, fuel = 0;
    public void start(double x) {speed = x;}
    public void refuel (double x) { fuel = fuel + x; }
 };
```

Dynamic Binding of Methods

- In object-oriented programming,
  - object->message (arguments)
  - Example: x->add(y)

hidden data	
msg1	method1
. . .	. . .
msgn	methodn

- In conventional programming,
  - Operation(operands): e.g. add(x,y)
  - Impl of operation is always the same
    - e.g., ML abstype functions are treated as global functions
- Implementing Dynamic Binding of methods
  - An object may contain both data and functions
    - Instance variables, also called member variables
    - Functions, also called methods or member functions
  - Put all the name-value bindings into a table
    - Content of table can be changed, just like the activation record of a function

Static vs. dynamic lookup

- What about operator overloading (ad hoc polymorphism)?
  - int add(int x, int y) { return x + y; }
  - float add(float x, float y) { return x + y; }
- Very important distinction
  - Overloading is resolved at compile time
  - Dynamic lookup is resolved at run time
  - Difference: flexibility vs. efficiency
- Statically bound functions
  - C++ non-virtual functions, Java static functions, global overloading of operators
- Dynamically bound functions
  - C++ virtual functions, Java non-static functions

Static Binding of Methods

- C++ class: non-virtual member functions
  - Essentially global functions with an implicit env parameter

```
class vehicle {
   protected: double speed, fuel;
   public: vehicle() : speed(0),fuel(0) {}
           void start(double x) {speed = x;}
};
vehicle* a = new vehicle; a->start(5);
```

- Java/C++: Static Methods/Variables
  - Essentially global functions/variables in a name space

```
class vehicle {
   static protected double speed, fuel;
   public static void start(double x) {speed = x;}
};
Vehicle.start(3.0);
```

Subtyping And Inheritance

- In C++/Java, classes can declare other classes as base classes, which means
  - The derived class is a subtype of the base class (how does it relate to the union types in C and ML?)
  - The derived class can inherit both interface and implementation of the base classes
- Goal: separate classes into groups
  - Members of the same group share some structural property
  - What properties?
    - Interface: the external view of an object
    - Implementation: the internal representation of an object
- Subtyping: relation between interfaces
- Inheritance: relation between implementations

Subtype Polymorphism

- A function can often operate on many types of values

```
void diagonal-move(MovableThing& a, int len)
{
    for (int step = 0; step < len; ++step)
        a.move(1,1);
}
```

  - Diagonal-move can be applied to all movable things
- Subtyping: if interface A contains interface B, then A objects can also be used as B objects
  - The interface of an object is its type.

Subtyping vs. Inheritance

- Subtyping and inheritance often occur simultaneously
- Subtype inheritance
  - Categorize data into related types
  - Java: implementing interfaces, inheriting a base class
  - C++: public inheritance from one or more base classes
- Implementation inheritance
  - Sharing of implementation details (not necessarily interface)
  - C++: private and protected inheritance
- Why not just invoke members of other classes?
  - When to inherit (is-a vs. has-a relations)?
  - Do they support the same interface (subtype relation)?
  - Need to change dynamic binding of base methods?
  - Need to access protected members of the other class?

C++/Java Subtyping

- Java/C++ subtype polymorphism

```
class MovableThing
      { virtual void move(int,int) = 0; }
class MovableThing1 : public MovableThing
   { … void move(int x, int y) { … }… };
class MovableThing2 : public MovableThing
   { … void move(int x, int y) { … }… };

void diagonal-move (MovableThing& a, int len)
{
    for (int step = 0; step < len; ++step)
        a.move(1,1);
}
```

ML Subtype Polymorphism

- ML subtype polymorphism

```
abstype MovableThing =
  MovableThing1 of … | MovableThing2 of …  with
   fun move(MovableThing1(…), int x, int y) = …
             | move(MovableThing2(…), int x, int y) = …
  end;

fun diagonal-move (MovableThing a, len) =
    if  len > 0 then
             (move(a, 1,1); diagonal-move(a, len-1))
```

- Difference: have to know all the subtypes when defining MovableThing

Designing The Class Hierarchy

- What is the subtype relation?

```
datatype element=Sym of string
                | Num of int
                | List of elements
    and elements = Empty
                | Multi of element * elements
```

- How to implement the subtyping relations via class inheritance?
  - Base types: element and elements
  - Derived types: Sym, Num, List, Empty, Multi

Varieties of OO languages

- Class-based languages (C++, Java, …)
  - Behavior of object determined by its class
- Object-based (Self)
  - Objects defined directly
- Multi-methods (CLOS)
  - Operation depends on all operands
- This course: class-based languages
- History
  - Simula: Object concept used in simulation 1960's
  - Smalltalk: Object-oriented design in systems 1970's
  - C++: Adapted Simula ideas to C 1980's
  - Java: embedded programming, internet 1990's

Summary

- Abstractions and object-oriented design
- Primary object-oriented language concepts
  - dynamic lookup
  - encapsulation
  - inheritance
  - subtyping
- Program organization
  - class hierarchy
- Comparison
  - Objects as closures?