• Out of town this afternoon thru Monday

• HW1 due next Thursday 9/9
Types

• Last time:

• **strongly typed** if no type errors occur at run time

• **statically typed languages** rule out type errors at compile time
  • requires programmers to write type declarations

• **dynamically typed languages** rule out type errors at run time
  • dynamic checks before operations
Records

A **record** is a value consisting of a sequence of named fields.

Declaring a record:
- C: `struct { int a; float b; }`
- Pascal: `record a: integer; b: real; end`

Support access operations via field names:
- `x.a, x.b`
Records as ADTs

```c
struct IntList {
    int head;
    struct IntList *tail;  // must be a pointer!
}
```
Records as ADTs

```c
struct IntList {
    int head;
    struct IntList *tail; // must be a pointer!
}

int get(IntList *xs, int i) {
    if (i == 0) return xs->head;
    else get(xs->tail, i-1);
}

void append(IntList *xs, IntList *ys) {
    if (xs->tail == NULL) xs->tail = ys;
    else append(xs->tail, ys);
}
```
Problems

• Implementation exposed.

• Only one implementation supported.

• Clients of the ADT must all use same implementation.
Objects

- Focus is on *data* rather than on *processes*.
- Programs composed of self-contained, interacting objects.
  - vs. as a list of tasks to perform
- Bundle data with the operations on that data
- Encapsulate (hide) implementation from clients.
  - Can only access data through **public** interface.
Objects

Source code for class defines concrete type (implementation)

Interface defined by **public** variables and methods of a class

class IntList {
    private int head; private IntList tail;
    public int get(int i) {
        if (i == 0) return this.head;
        else return get(this.tail, i-1); }
    public void append(IntList ys) {
        if (this.tail == null) this.tail = ys;
        else this.tail.append(ys); }
}
OO languages

Not an exhaustive survey

Simula 67 → C++ → Java → C#

Smalltalk → C→ Objective C

LISP → CLOS

ML → OCaml

Scala
**OO languages**

- Two kinds of OO languages
  - class-based
    - Simula, Smalltalk, C++, Java, C#, Scala
  - prototype-based
    - Self, Cecil, JavaScript
Class-based languages

- A class defines a template for creating objects
  - defines what members an object of class C ("instance") has
    - fields (aka member variables, instance variables)
    - methods (aka member functions)

- All instances have the same structure
  - same methods, same fields

- We’ll ignore static members for now
Inheritance of classes

• A *subclass* (derived class) inherits from (extends) one or more *superclasses* (base class)

• As if members of the superclass were copied down into the subclass declaration

• Instance of subclass contains all members of an instance of the superclass + new members defined by the subclass
class Point {
  int x, y;
  void move(int dx, int dy) {
    x += dx; y += dy; }
}

class ColorPoint extends Point {
  Color c;
  void redden() {
    c = Color.RED; }
}
Prototype-based languages

• No notion of class

• Inheritance is by *delegation*

• To create a new object:
  • create an object from nothing (ex nihilo), or
  • clone another object (the *prototype*)
  • modify the new object
  • maintain a reference (*delegate*) to the original
// create two objects
var p = {x: 1, y: 2};
var cp = {color: "red"};

// make cp extend p
cp.__proto__ = p;

cp.x // 1
cp.y // 2
cp.color // "red"
Delegation

• If object does not contain a given field, check its prototype

• Behaves similarly to inheritance
Multiple implementations

OO languages let you have multiple implementations of the same specification:

class List {
    int length();
    int get(int i);
    List append(List x);
}

class ArrayList extends List { ... }
class LinkedList extends List { ... }
class ConcList extends List { ... }
Dispatching problem

**Problem**: don’t know what code to run at compile time

List a = ...;

a.length();

• ArrayList.length or LinkedList.length?

• Objects must “know” their implementation at run time
Compiling objects

- Add to each object an extra pointer to a dispatch vector (aka virtual table, vtable) with pointers to method code

- Code receiving $x : \text{List}$ only knows $x$ has an initial dispatch vector pointer
Polymorphism

• Code can use values with more than one type

• Object oriented languages support *subtype polymorphism*

• Good for heterogeneous data structures containing different implementations of the same interface

• Can mix different Animal implementations in the same list
  • (Cat, Duck, Cow, Moose, TRex, Human, Sponge)
Type relationships

- Classes and their superclasses are related by a *subtype* relationship
  - `ArrayList <: List`
  - `LinkedList <: List`
Subtypes

One type **extends** another by allowing more operations

```java
class Point {
    int x();
    int y();
}

class ColorPoint {
    int x();
    int y();
    Color color();
}
```

ColorPoint <: Point

“is a subtype of”
(also: \(\leq\))
Subtyping

Predicate view of types
• A type is a predicate on values
• T1 is a subtype of T2 if T1’s predicate implies T2’s

• Barney is a Dinosaur => Barney is an Animal

Set-theoretic view of types
• A type is a set of values
• T1 is a subtype of T2 if T1’s set of values is a subset of T2’s

• The set of Dinosaurs is a subset of the set of Animals

• Note: it’s a subset of values not a subset of operations
Substitution principle

Can always substitute an instance of a subtype for an instance of a supertype and the program will have no type errors

Bird ---> Penguin

Liskov Substitution Principle:
• if $P(x)$ is true about objects $x$ of type $T$, then $P(y)$ is true for objects $y$ of type $S$, a subtype of $T$
• “behavioral subtyping”
• usually too strong to be enforceable
Subtype relation

- Subtyping is a *binary relation* on types

- Notation:
  - $T_1 <: T_2$ – $T_1$ is a subtype of $T_2$

- $<:$ is:
  - reflexive: $T <: T$
  - transitive: if $T_1 <: T_2$ and $T_2 <: T_3$, then $T_1 <: T_3$
  - antisymmetric: if $T_1 <: T_2$ and $T_2 <: T_1$, then $T_1 = T_2$
Type-checking assignment

if x has type T1 and e has type T2

then require T2 <: T1

x = e has type T1

Let T <: S

S s = ...;
T t = ...;

s = t; // allowed

// not allowed
Type-checking assignment

- Let $T <: S$
- $S \ s = \ldots$
- $T \ t = \ldots$
- $s = t; //$ allowed
- $t = s; //$ not allowed
Subtyping in Java

- Classes:
  - class C
    - C <: Object
  - class C extends D
    - C <: D

- Interfaces:
  - interface I
    - I <: Object
  - class C implements I
    - C <: I

- Primitives:
  - <: is =
    - int <: int
    - Why not int <: long?

- Arrays:
  - int[] <: Object
  - C[] <: Object
  - if T1 <: T2 then T1[] <: T2[]
    - Note: this rule is broken
    - Why?
Primitives

• Why are primitive types different?

• What is the representation of a value of type int?
• Type long?

• Type Object?
Subtyping in Java

- **Classes:**
  - class C
    - C <: Object
  - class C extends D
    - C <: D

- **Interfaces:**
  - interface I
    - I <: Object
  - class C implements I
    - C <: I

- **Primitives:**
  - <: is =
  - int <: int
  - *Why not* int <: long?

- **Arrays:**
  - int[] <: Object
  - C[] <: Object
  - if T1 <: T2 then T1[] <: T2[]
    - Note: this rule is broken
    - Why?
Covariant array subtyping

- In Java:
  - `Integer[] <= Object[]`

- `Integer[] x = new Integer[10];`
- `Object[] y = x;`
- `y[0] = “this is a string, not an integer”;

- Will cause an `ArrayStoreException` at run-time