Object-Oriented Languages and Object-Oriented Design
What is an OO language?

• In Ada and Modula 2 one can define objects
  – encapsulate a data structure and relevant operations in a construct
  – data structure hidden to clients
• One can define ADTs
• Possible to define generic objects and ADTs
• *Is this enough to call them object-oriented?*
…but what is an OO language?

OO languages must support:

• abstract data type definitions
• inheritance
• (inclusion) polymorphism
• dynamic binding of method invocations to bodies (also called dynamic dispatching)
  – the method called on an object depends on the object’s dynamic type, although its static type is used for type checking to ensure type safety
Jargon

• Often procedures/functions encapsulated in an object are called
  – methods, or member functions
and procedure/function call
  – sending a message
To respond to a message sent to an object O, object O invokes an appropriate method
Inheritance (in C++)

class stack {
    public:
        stack(); {top = 0;} //constructor
        void push(int) {s[top++] = i;};
        int pop() {return s[--top];};
    protected:
        int top;
    private:
        int s[100];
};
class counting_stack : public stack {
    public:
        int size(){return top;}; // # of elem. on the stack
};
Polymorphism

• A stack reference can refer to a counting stack (but not vice-versa)

```cpp
stack* sp = new stack;
counting_stack* csp = new counting_stack;
... 
sp = csp;  //ok

csp = sp;  //compiler error (strong typing: csp -> size???)
```

• WARNING: in C++ different behavior for automatic objects (unlike “pure” OO languages)

```cpp
stack s; counting_stack cs;
s = cs;  //ok but with COERCION!!!
cs = s;  // compiler error
```
Dynamic binding

stack* sp = new stack;
counting_stack* csp = new counting_stack;
...
sp->push(…); //stack::push
csp->push(…); //counting_stack::push
sp = csp;
sp->push(…); //which push????

The operation to be invoked should be the one defined by the object’s dynamic type
Inheritance and type system

• We wish a subclass to define a subtype
• NB: type, set (better: algebra), class should be the same thing but ...
• The relation between a type and a subtype is substitutability
• If we wish to achieve strong typing and polymorphism, the subtype can
  – add new operations
  – redefine operations preserving contravariance of input (new domain is a superset of old one) and covariance of output (new range is a subset of old one) parameters
    • in C++, Java, Object Pascal, and Modula3 they must be the same type; (NB: Java 1.5 admits covariance of output)
    • Eiffel and Ada require covariance of both parameter and result
Virtual functions and dynamic binding

class student{
public:
...
virtual void print(){...};
};

class college_student: public student{
    void print() {
        ... // specific print for college_student
    }
};

student* s;
college_student* cs;
...
s->print(); //calls student::print()
s = cs; // okay
s->print(); //calls college_student::print()
Inheritance

• Types organized in a hierarchy
• Subtype “is-a” parent type
  – counting_stack is-a stack
• Subtypes can add features and redefine virtual member functions
• Multiple vs single inheritance
Multiple inheritance

It can be necessary to redefine (name clashes) and undefine (to avoid duplication of inherited entities)

- C++ supports multiple inheritance
- Java does not (but can “inherit” from multiple interfaces)
Abstract classes

class shape{
public:
    void draw() = 0; // these are pure virtual functions
    void move(...) = 0;
    void hide() = 0;
    point center;
};
class rectangle: public shape{
private:
    float length, width; // specific data for rectangle
public:
    void draw() {...}; // impl. for derived pure virtual function
    void move(...) {...};
    void hide() {...};
};
Class structure

class C {
    public:
        // accessible to the general public
    protected:
        // accessible to members and friends and
        // to members and friends of derived classes only
        // and (for Java) to classes of the same package
    private:
        // accessible to members and friends only
};
Pure OO design

• Decomposition is based ONLY on classes
• A class is both a module and a type
• Classes may be generic
• Classes may be abstract
• Relationships among modules (classes)
  – USES
  – INHERITS
Pure OO language

• Ada and Modula 2 are not object oriented
  – they are sometimes called “object based”
• C++ is not purely object oriented: it supports also functions and functional decomposition
• Java is purely object oriented
Smalltalk

• Fully dynamic: objects carry type at r.t.
• Syntax for method invocation suggestive of message sending
  myPoint xMove
  T lowerBound: 6 upperBound: 95
• Class variables and methods
• Single inheritance
Variables and dynamic typing

• Variables are uniformly refs to objects
• Refs are untyped: messages sent ok if object bound to ref can respond

→ Dynamic polymorphism
Pervasive role of objects

• 1 is an object
• a class is an object (of class class) that can respond to instantiation
  – myPoint <- point new
    • msg new sent to class point; assignment to myPoint
• control structures are objects
  – programs viewed as data
Problem of size of private types

- If stack is an ADT which exports a type stack, e.g.
  - In Ada and Modula, a module that exports a type stack
  - In C++, it is the class itself

- How to handle a client declaration:
  s: stack;     //how much memory to allocate?
ADTs and modularity

\[ s: \text{stack} \]
\[ \text{define stack and export ops} \]
Size of an ADT...

- Compiler has to know the size at client side even though it is not part of spec
  - C++ and Ada require the representation of private type in the specification
  - Java uses pointers to ADTs, pointer is of fixed size
  - Modula’s opaque export allows access through pointer without revealing representation
Problem of assignment of private types

- Assignment and equality tests are fundamental operations of types. Does an arbitrary ADT support them?
  - C++ provides default (memberwise) operations but user can redefine the operations
  - Ada declares a type as private (supports assignment and equality) and limited private (does not)
The case of Java

- All objects are implicitly accessible via pointers
- Copying an object copies the pointer, i.e., object is shared