Today - CS202 Lecture #1

cs.pdx.edu/~karlaf

1. Welcome! What to expect.
2. Syllabus
3. Approach for Programming
4. Introduction to OOP
5. Syntax: Inheritance Preview

Office Hours:
Wed 9-9:50am
Fri 1-1:50pm
FAB 120-19

Options

* You may watch this class online
* Lectures will be available Saturdays
* Once you watch lectures → Weekly Activities
* Keep current with D2L
* Important to watch each week

Programming Guidelines

1. Each program focusses on OOP
2. This means that each program will have multiple classes.
3. Each class needs to have a well defined purpose or "job"
4. Data Members must be private or protected
5. Consider waiting to intgrate the data structure(s) until after the OO design is known
6. NO Global variables
7. NO use of String class
8. All arrays MUST be dynamically allocated
9. All programs must use inheritance

Sample OO Design

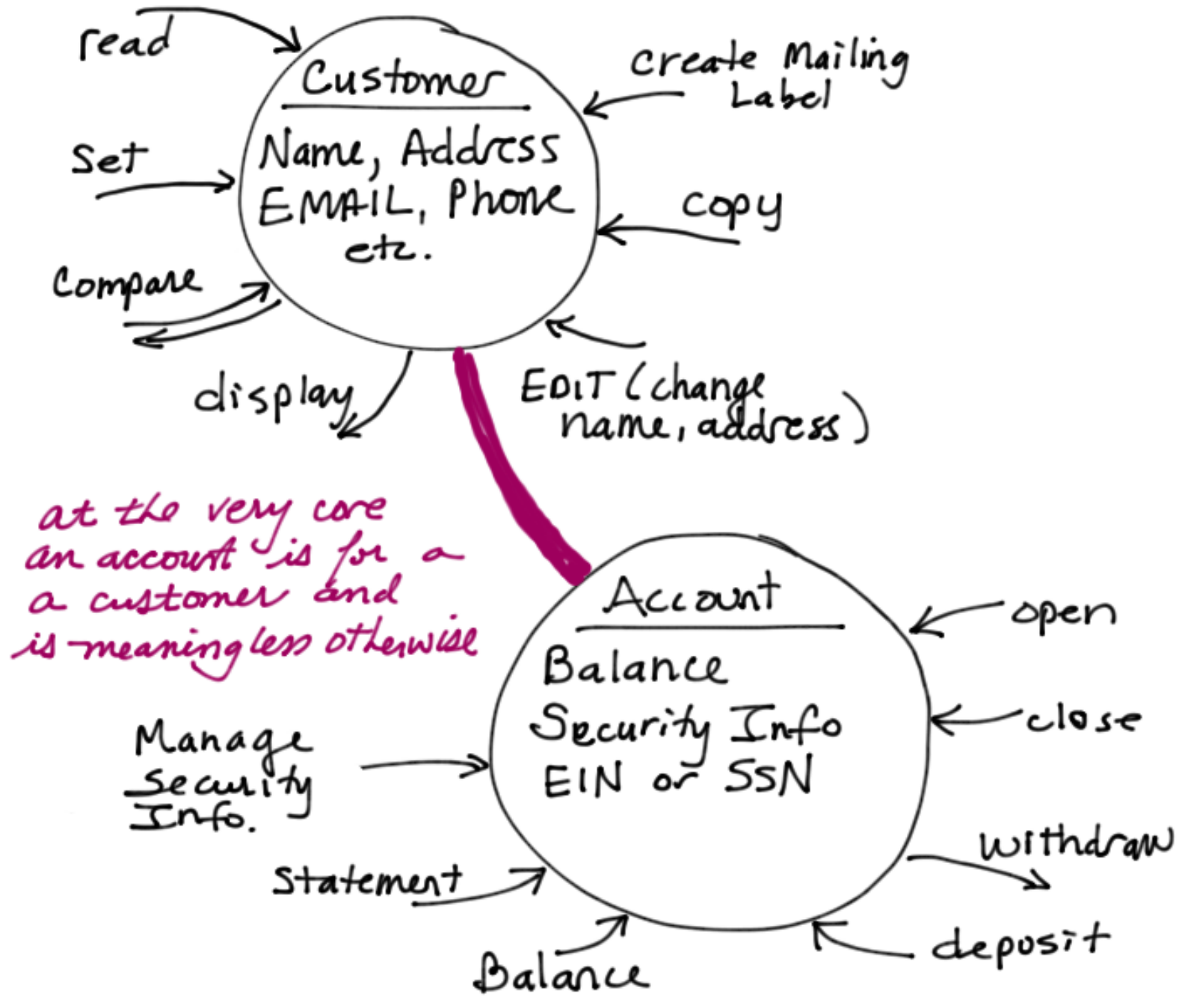
1. Understand the problem from an application perspective.
2. List all nouns (although they won't all be classes
3. Group Nouns together thinking about the job of each resulting class
4. Remember, with OOP classes work together to solve a problem!

Banking Application

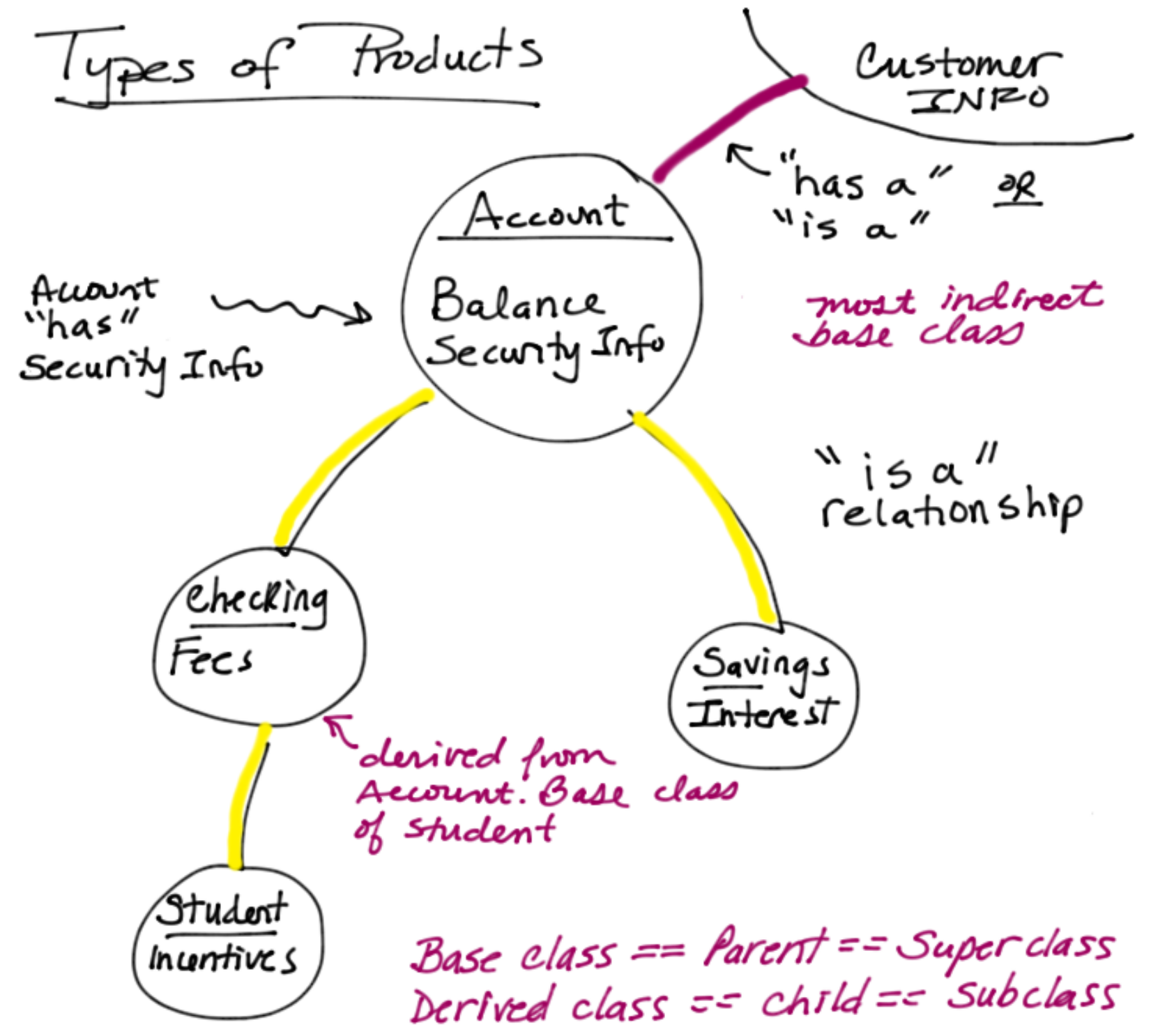
Products: Savings, Checking, CD's, Money Market
credit cards, Loans (equity, mortgages, car), Student Accounts

Group
Nouns: Account, Balance, customer, Transactions or history
Fees, interest, ...

Customer: Name, Address (Physical, Mailing), email
Password, Password questions, Maiden Name
credit score, application ...


read
Customer
Name, Address
EMAIL, Phone
etc.
create Mailing Label
Set
copy
Compare
display
EDIT (change name, address)
at the very core an account is for a a customer and is meaningless otherwise
Account
Balance
Security Info
EIN or SSN
open
close
Manage Security Info.
withdraw
Statement
deposit
Balance

Types of Products



Base class == Parent == Superclass
Derived class == child == Subclass

Syntax for Single Inheritance Hierarchies

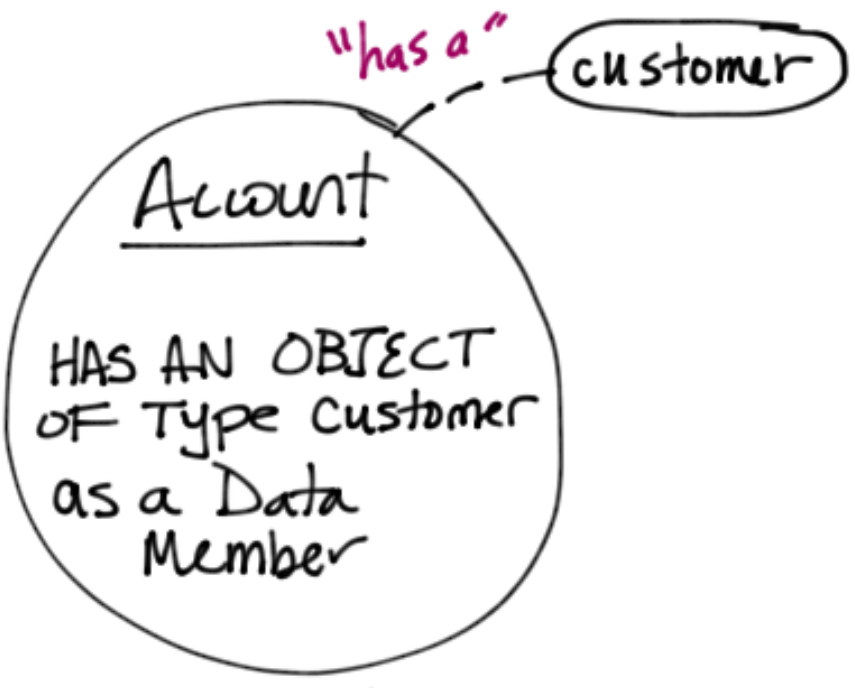
```
class Account
{ public:
   protected:
   private:
      float balance;
};

class Checking : public Account
{
    public:
   protected:
      // Fees, interest
   private:
};
```

Available for subclasses but not client programs

Derivation list

"Checking is an Account object plus more"

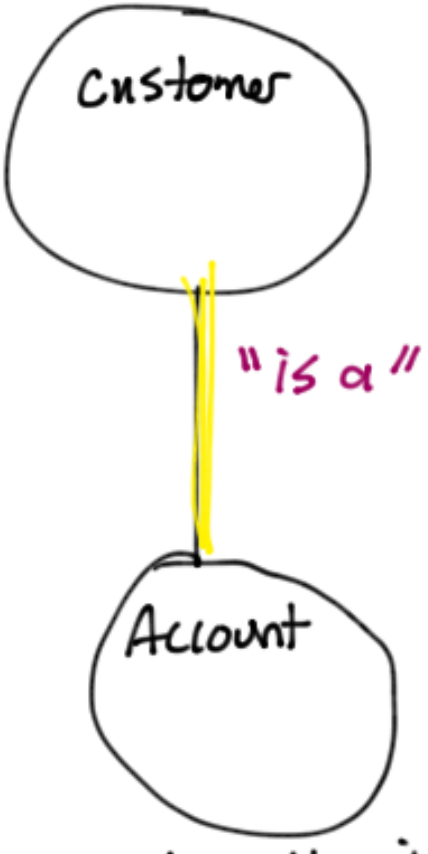
"has a" vs "is a"



object.open();
open would have to call customer member functions to work w/ name, email, etc

datamember.input();

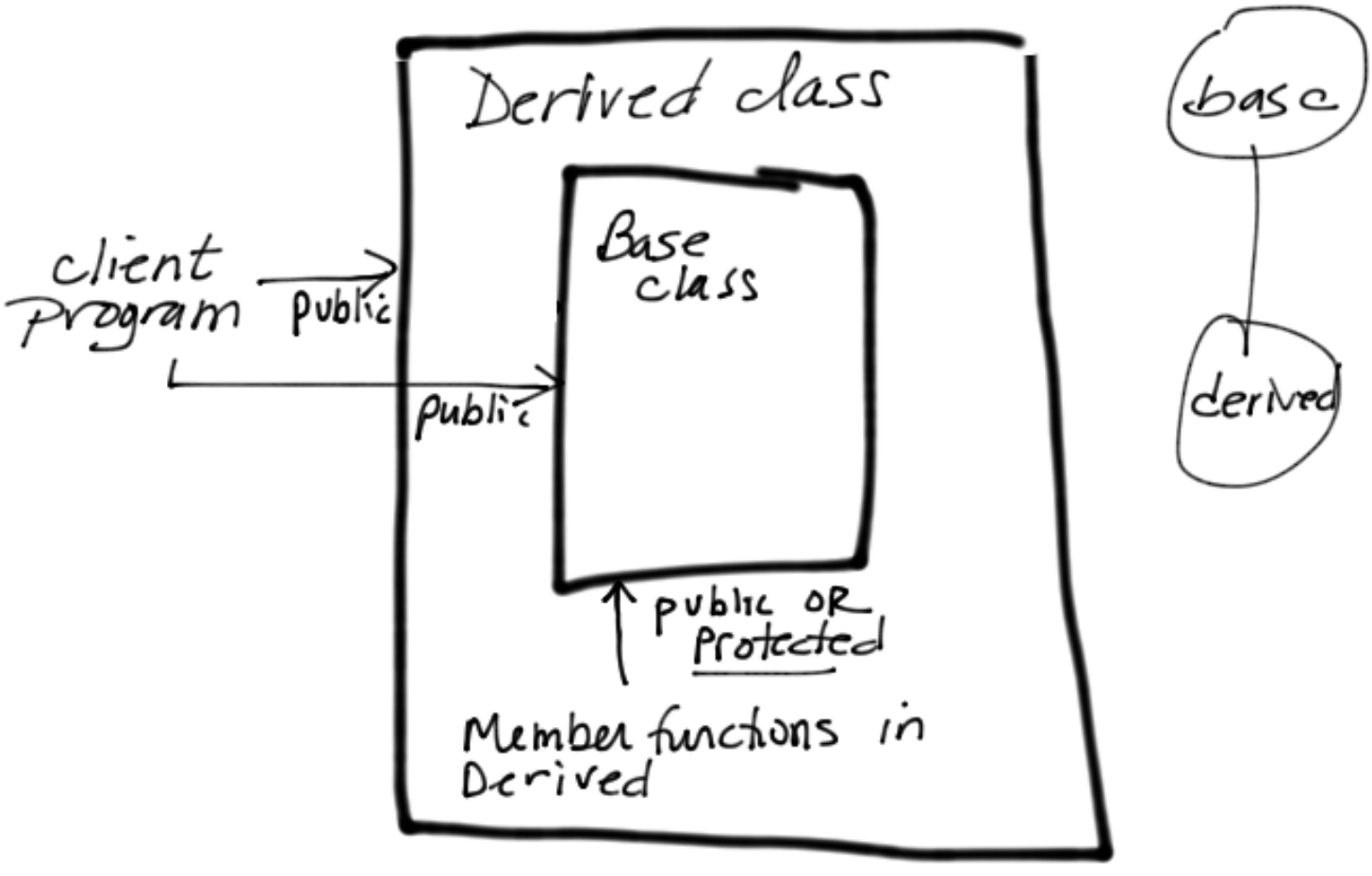
a customer class member function



Now, the input function can be called directly:

object.input();

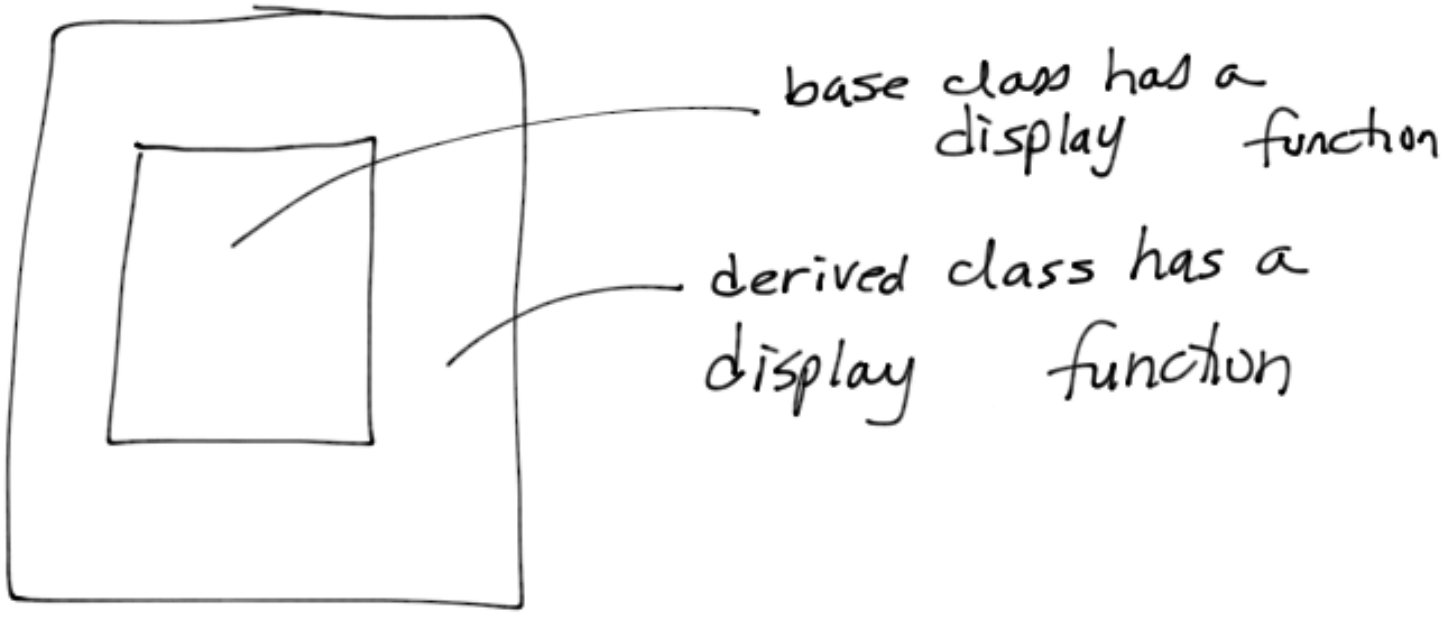
Think of Inheritance:



Derived Object;

Side Note: a member function of a derived class can call a public or protected member function of the base class.

What if.... Same function name?



Regardless of the argument lists, Function Overloading does not take place.

The chent can call the parent's public member function via:

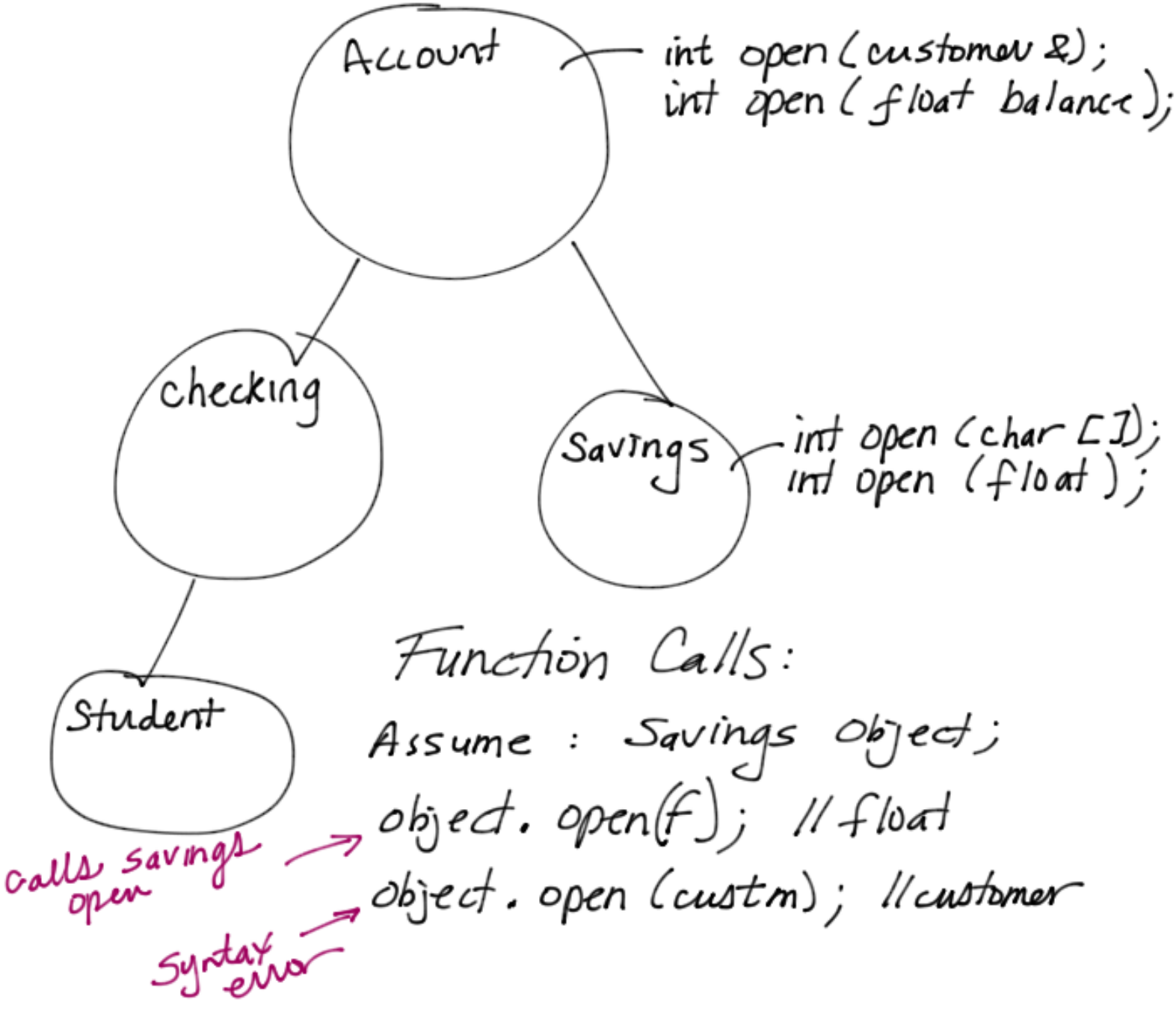
```
base_classname::display(args);
```

↑ Scope resolution operator

```
derived object;
object.base_classname::display(args);
```

Function Overloading...

only occurs within a given scope { } block



Function Calls:

Assume : Savings object;

object. open(f); // float

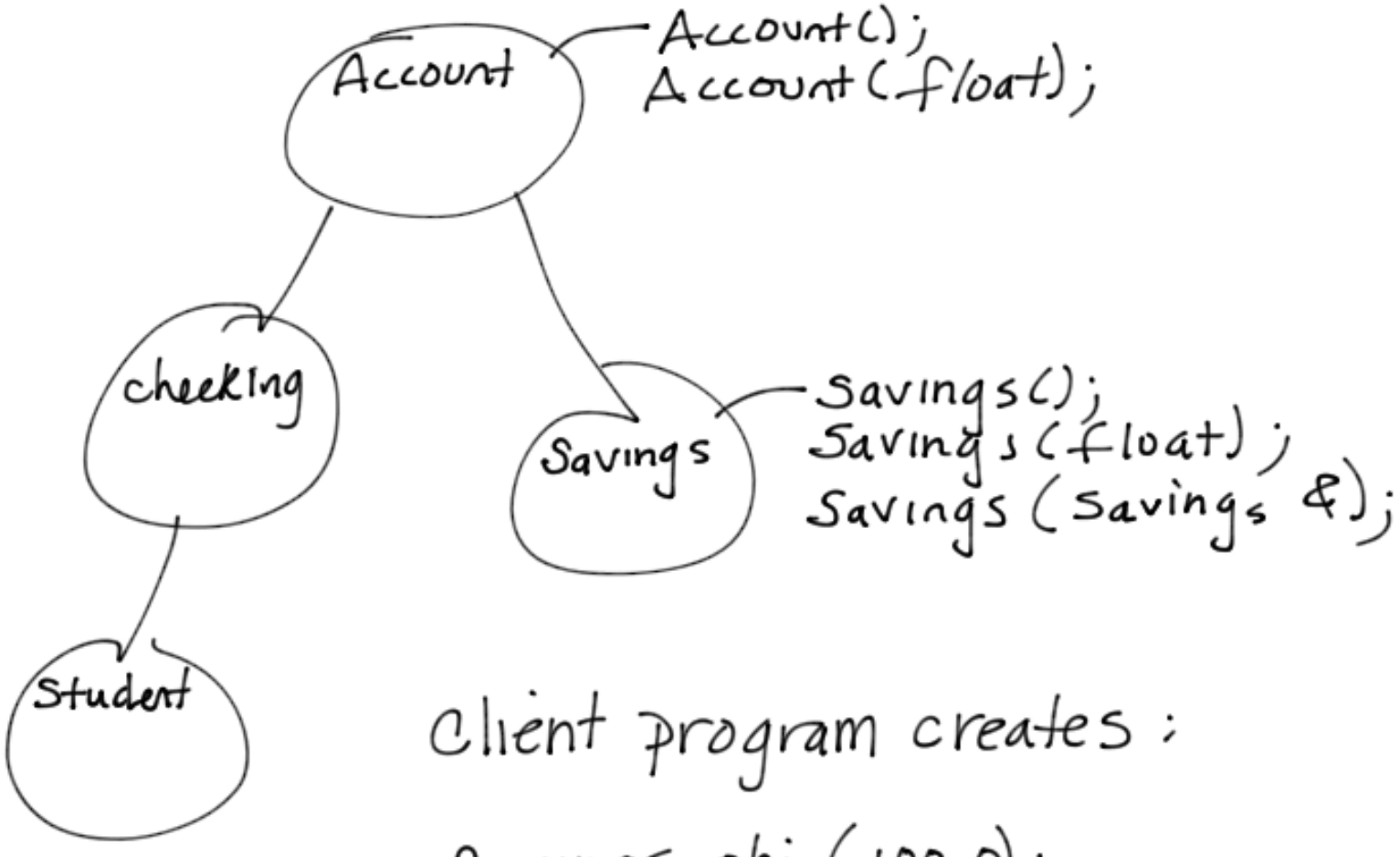
object. open (custm); // customer

calls savings open

Syntax error

* The derived class functions HIDE the base class functions of the same name.

Constructors with Arguments



Client program creates:

```
Savings obj(100.0);
```

1. First the default constructor for the Account class is implicitly called.

2. Then, the savings constructor with the float argument is called

So, the information is not passed up to the parent.....

Initialization Lists

SOLUTION

In the implementation of the constructors we can add inthalization lists.

They can be used to kick start the parents' constructor when arguments are involved

```
Savings::savings(float val) : Account(val), foo(9)
{
   ... // body of the function
}
```

initialization list

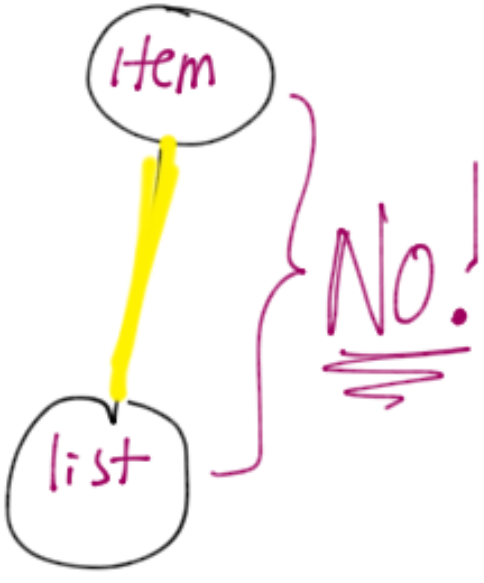
data member (Savings)

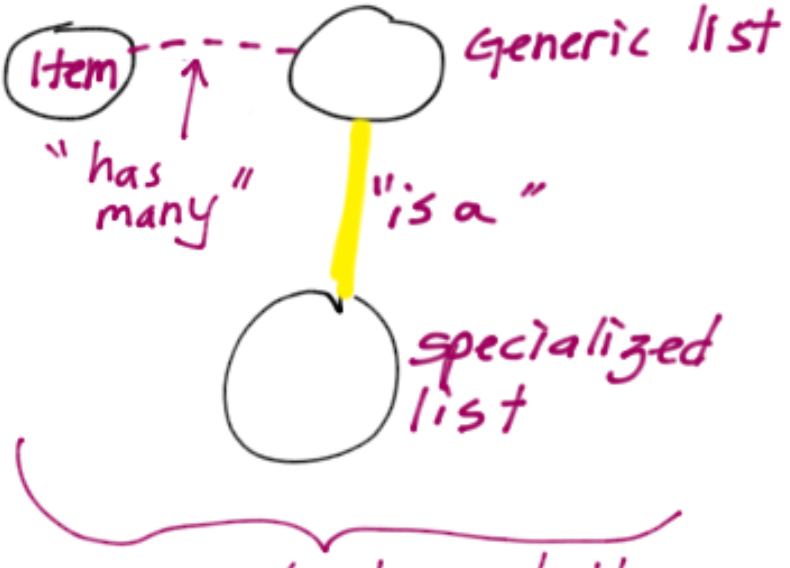
now the default constructor will not be involked when an object is created with a float passed as an argument

What about 1 vs. Many

1. Inheritance is not designed to take 1 instance and turn it into many.

So, if you had a "list", it should never be derived from (a) a node, or (b) an individual item





It is common to have both "has a" and "is a" relationships creating the solution.

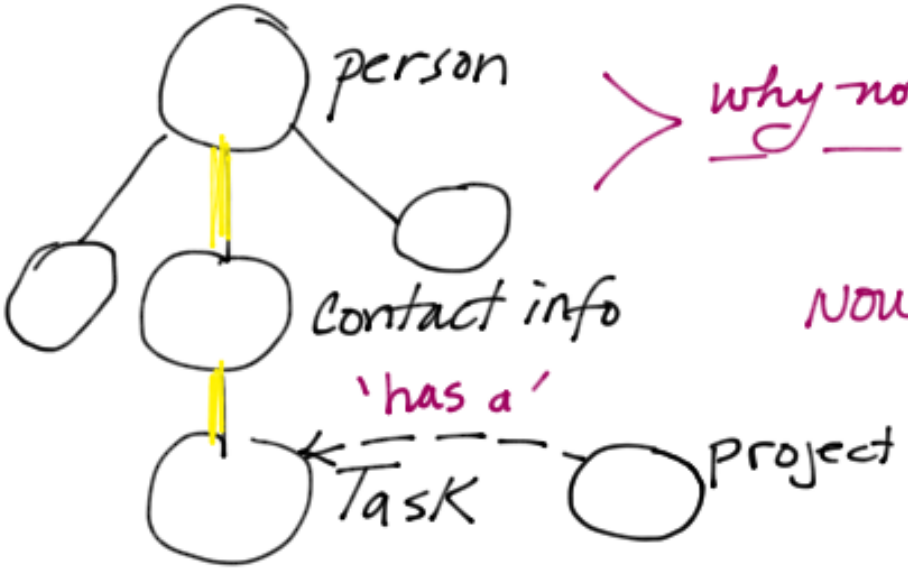
Example: To Do List

1. understand the problem
2. List Nouns, and Group them

Task
person
project
due date
priority

Person
name (first, last)
contact information

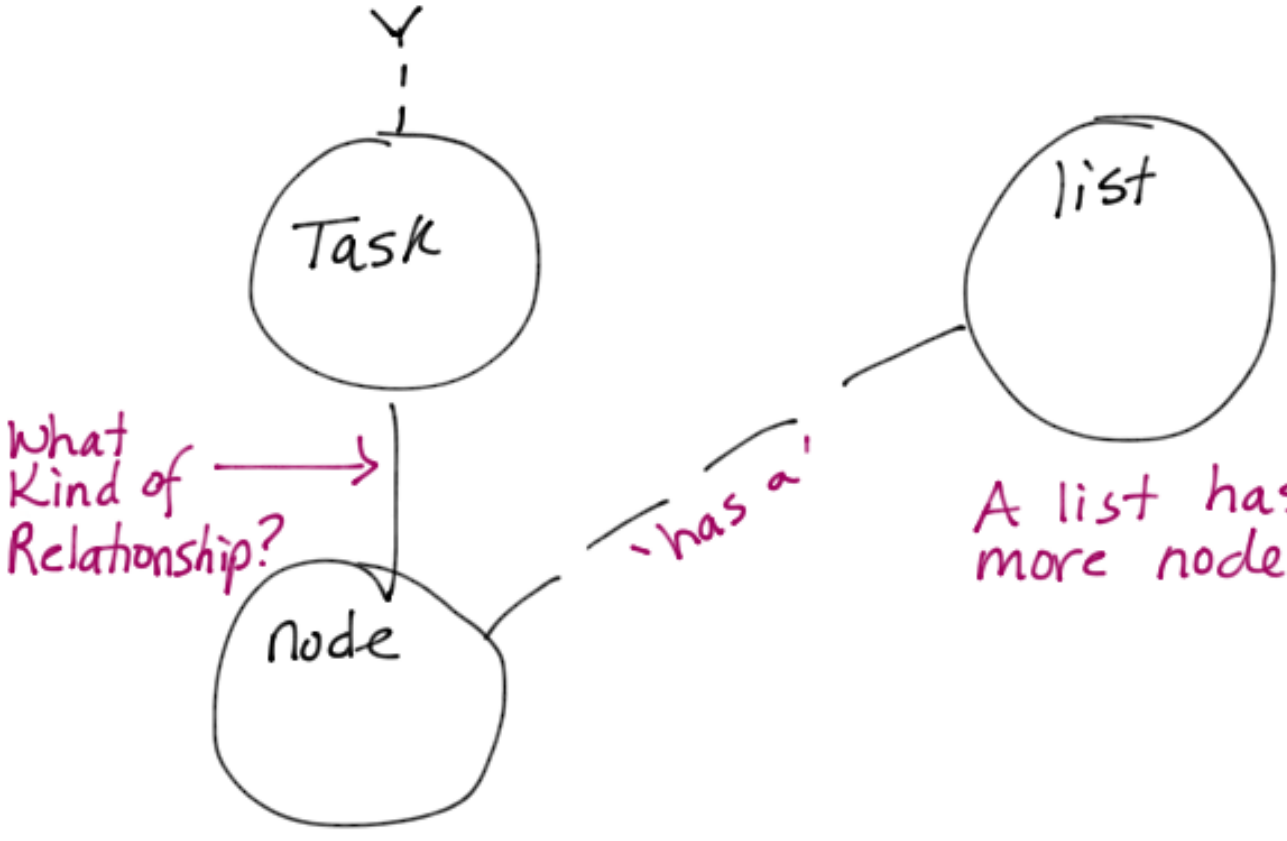
Contact Information
email address
phone #



why not opposite?

NOW, let's create the many

One Approach ....

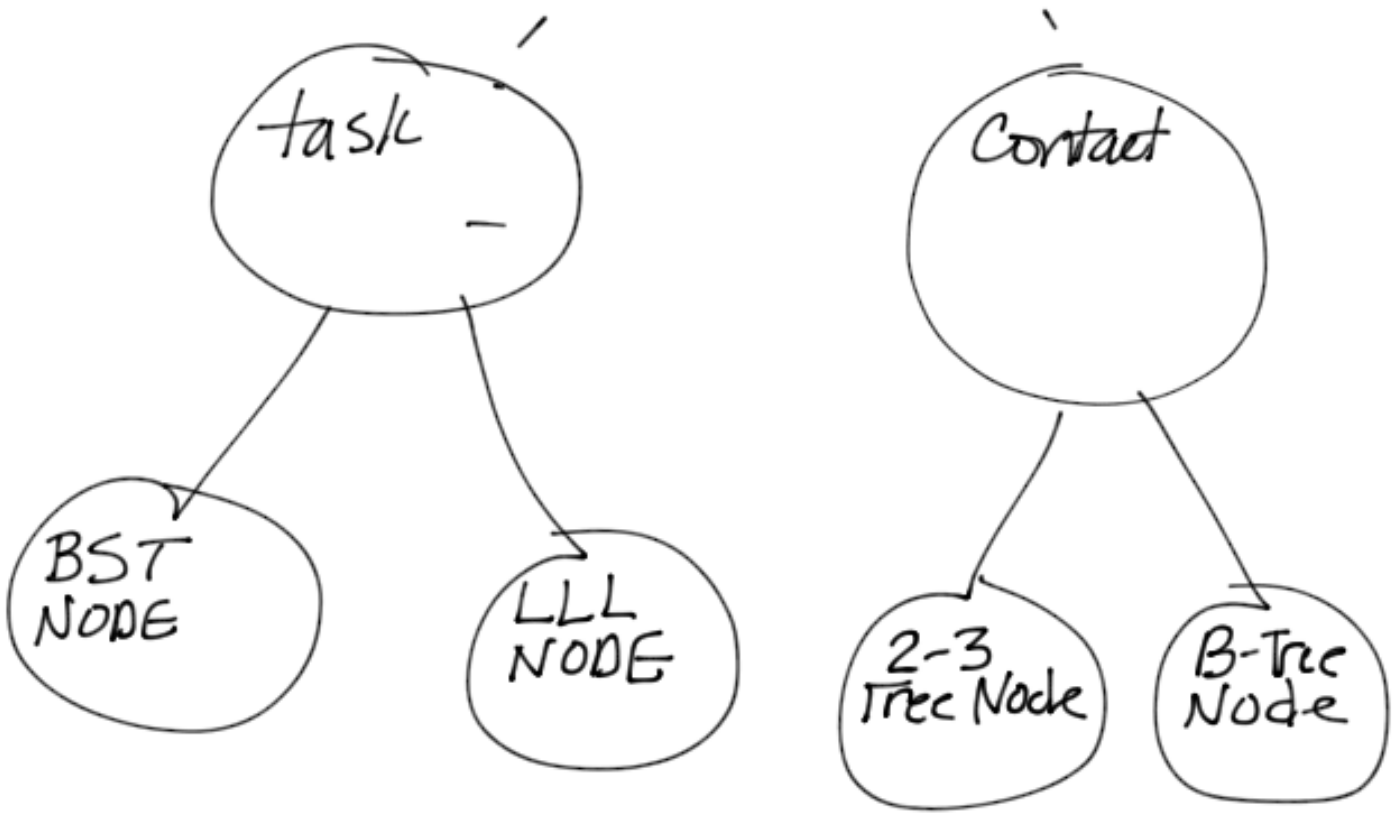


if "has a" (Task is a data member in a node), what happens if node is a class? It would need "wrapper" functions to pass information from the list class to the task.

if 'is a' (node is derived from task), the list could directly call task public functions

head → displaytask();

Opportunities:




```
class LLLNode : public Task
{
    :
};
```