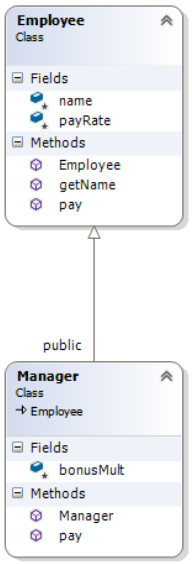
Inheritance implements IS_A

```cpp
#include "employee.h"
using namespace std;
Employee::Employee(string theName, float thePayRate)
{
  name = theName;
  payRate = thePayRate;
}

string Employee::getName() const
{
  return name;
}

float Employee::pay(float hoursWorked) const
{
  return hoursWorked * payRate;
}
```

```cpp
#include "manager.h"
using namespace std;

Manager::Manager(string theName,
                 float thePayRate,
                 int bonusMult)
  : Employee(theName, thePayRate), bonusMult(bonusMult)
{
}

float Manager::pay(float hoursWorked) const
{
 return bonusMult*(Employee::pay(hoursWorked));
}
```




```cpp
class Employee {
public:
  Employee(std::string theName, float thePayRate);

  std::string getName() const;
  float pay(float hoursWorked) const;

protected:
  std::string name;
  float payRate;
};
```

```cpp
#include "employee.h"
class Manager : public Employee {
public:
  Manager(std::string theName,
          float thePayRate,
          int bonusMult);

float pay(float hoursWorked) const;

protected:
  int bonusMult;
};
```

Manager inherits all base class members and data

Calling base class

Manager must call base class if needed

See constructor IL

Employee::pay(hoursworked)

(Scope it)

Pay is in 2 places do not even need pay in manager, but may want to change how pay works override pay to get diff behavior

Or just use base implementation (code reuse).

You must scope or get infinite recursion (remove Employee:: in manager)

If do not call base class employee in constructor. Compiler will attempt to create default constructor to call, if base does not have one will not compile.

Demo remove employee IL from manager

Do not need to call base class in destructor, compiler handles it.

Protected:

Way for derived classes to get at innards of base class (member vars and functions) without exposing implementation details to world.
Why? Without it no derived class can get at base class members

In general, it's a good idea to prefer less inheritance. Use containment wherever possible, and inheritance only in the specific situations in which it's needed. Large inheritance hierarchies in general, and deep ones in particular, are confusing to understand and therefore difficult to maintain. Inheritance is a design-time decision and trades off a lot of runtime flexibility.

Start here 10/22/15
Start on virtyual next time

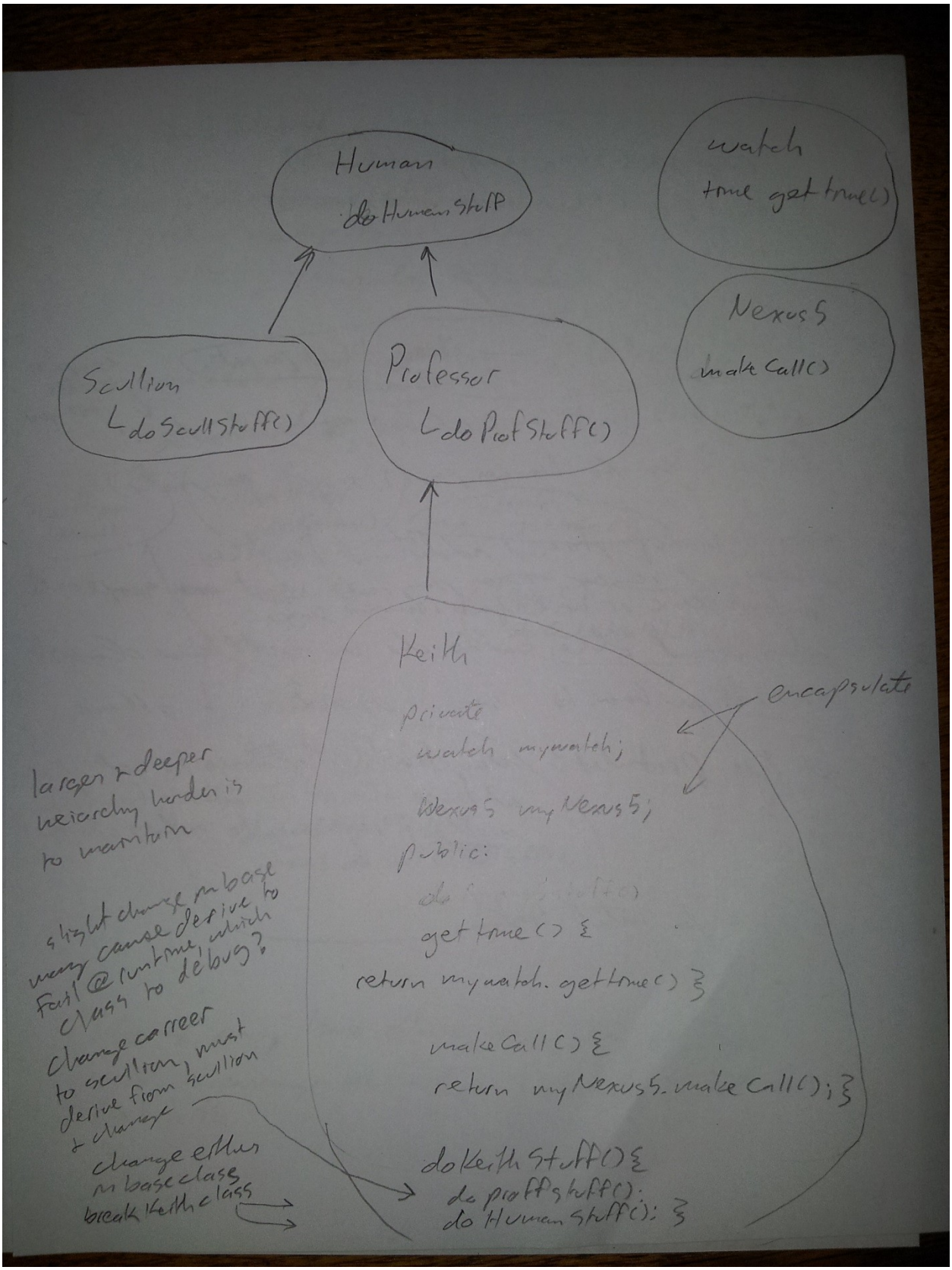
Virtual:

```
void outputPay(Employee *pEmp){
        // Assume all employees worked 40 hours this period.
        cout << "For Employee:" << endl;
        cout << "Name: " << pEmp->getName() << endl;
        cout << "Pay: " << pEmp->pay(40.0) << endl;
}
```

As long as not virtual;

Will always call employee::pay() will never go to most derived class
Unless you make it virtual in header of where virtual starts, every class derived therafter is virtual by default;
AS A COURTESY TO FUTURE DEVELOPERS PUT VIRTUAL IN ALL DERIVED CLASSES

Does it with a V-Table


Human
doHumanStuff
watch
time gettime()
Nexus5
makeCall()
Scullion
doScullStuff()
Professor
doProfStuff()
Keith
private
watch mywatch;
Nexus5 myNexus5;
public:
getTime() {
return mywatch.getTime() }
makeCall() {
return myNexus5.makeCall(); }
doKeithStuff() {
doProffStuff();
doHumanStuff(); }
encapsulate
larger + deeper
heirarchy harder is
to maintain
slight change in base
may cause derive to
fail @ runtime, which
class to debug?
change career
to scullion, must
derive from scullion
+ change
change either
in base class,
break Keith class

Composition:

When an object posseses something

I have a watch

I have a Nexus5

Which to Use:

Think of composition as a **has a** relationship. A car "has an" engine, a person "has a" name, etc.
Think of inheritance as an **is a** relationship. A car "is a" vehicle, a person "is a" mammal, etc.