ECE 30862

Fall 2019 `, Prof. Midkiff

smidkiff@purdue.edu

Updated January 11, 2019, 9:10AM

We will use Piazza for out-of-class questions piazza.com/purdue/fall2018/ece30862/home`

- You can pose questions about course material, homework, questions, etc.
  - The TA and I will monitor it daily and answer questions
  - Students can also answer questions
- Questions sent by email will be answered on Piazza
- Questions can be asked anonymously
- Piazza gives us the benefit of being able to see other people's questions and answers

Permanent office hours will be announced soon

- You can make an appointment
- In general, I will hang around after class to answer questions
- The first homework has a doodle poll that will guide the me and the TAs in deciding when to put office hours

Read the syllabus. It’s got useful stuff

- The schedule of what we talk about is approximate.
- We’ll have three “midterms” and no final.
- Exams are at night (sigh!). I have at least 3 trips this semester and having evening exams will enable us to not miss to much. I hate them too. Life is hard.
- If you need to miss an exam let me know as soon as possible.
- Practice exams are posted on the course website.

Course website and blackboard

- Homework and grades will be posted on Blackboard. The latter keeps me out of prison.
- Course notes, old exams, other material will be posted on the course webpage. You can find it at https://engineering.purdue.edu/~smidkiff/ece30862/
- I'll try and hand out slides before the first class we cover them. These will be printed 4 or 6 up. I'll also post .pdf files for them on the course web page before class for those of you that like to annotate .pdfs with your notes.

Grades

90 – 100: A
80 – 90: B
70 – 80: C
60 – 70 D
Below 60: F

Red is changed from the handout!

When assigning grades, I will often make lower than a 90 an A, and so forth. But if you make in the ranges above you will get at least the grade indicated. I often assign plus/minus grades, but an A- will always go to a grade that would have been in the B range, a B- to a grade that would have been in the C range, and so forth. A+ may be given to the highest A grades, B+ to the highest B grades (after assigning A-), and so forth.

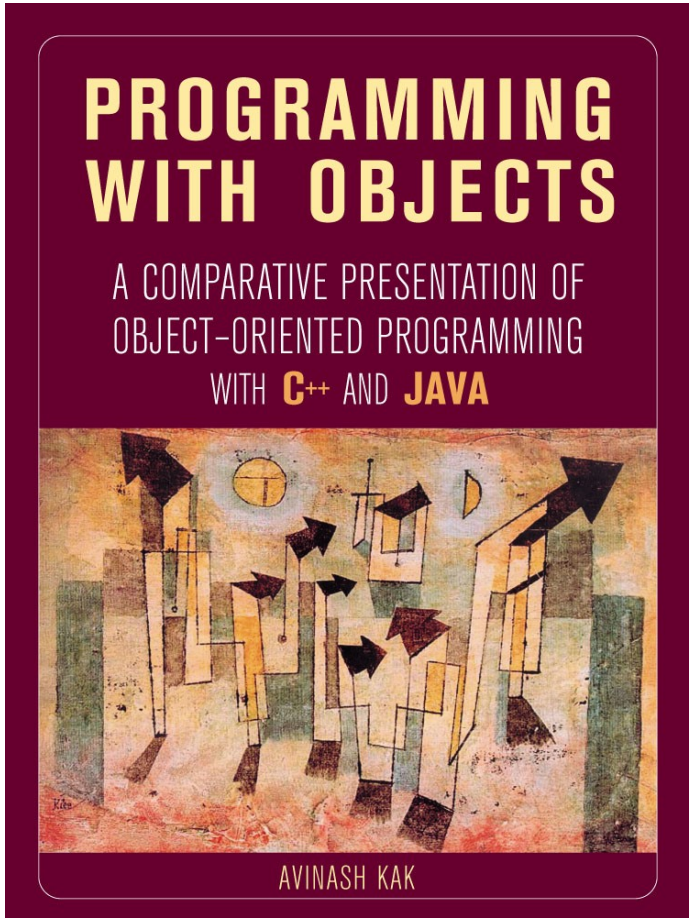
Tests: 3 tests will be given and each will be worth 14% of your grade, for a total of 42%.
Projects: The C++ project will be worth 25% of the grade, and the Java project 20%.
Homework: Homeworks, in total, will be worth 7% of your grade. At least 10 homeworks will be given.
Quizzes: In-class quizzes will be worth 6% of your grade. 10% of the quizz scores will be dropped.

Some additional info

- If you haven’t had 264 or its equivalent, talk to me.
- If you don’t understand pointers, talk to me.
- You should be able to write and compiler programs on a Unix-like machine (Linux, OS X) or Windows.
- Compilers are for catching syntax errors. I won’t test on syntax – syntax isn’t fundamental and I hope we’ll program enough and you’ll see enough code that you can at least read syntactically correct programs.
  - I hope to have some debugging homeworks. These may have syntax issues, especially for C++.

The textbook

- It is optional
- Learn to use Stack Overflow and other online resources.
- Don’t take online resources as truth. Also learn to write little test programs to check your understanding.



Course outcomes

1. write object-oriented programs of moderate complexity in Java.
2. write object-oriented programs of moderate complexity in C++.
3. understand the concepts of inheritance and polymorphism.
4. use template classes and the STL library in C++.
5. overload operators in C++.
6. incorporate exception handling in object-oriented programs.
7. understanding of the difference between function overloading and function overriding.
8. write programs with multiple threads and use synchronization among threads.

Emergency Preparedness

- Look here for general information: https://www.purdue.edu/ehps/emergency_preparedness/
- Look here for a quick reference guide: https://www.purdue.edu/ehps/emergency_preparedness/docs/6.8.17Quick%20Ref%20Guide-updated-Dec2016.pdf
- Look here for building information: https://www.purdue.edu/ehps/emergency_preparedness/bep/arms-bep.html
- Our greatest threat is tornadoes and fire. Go to the basement and/or interior corridors for tornadoes, get out of the building for fire.

Cheating

- I don’t like cheating. It puts students who do their own work at a disadvantage.
- Cheaters create work for the.
- These two things put me in a bad mood. To seek revenge for that, you will be turned in to the department and the Dean of Students for cheating.
- Tests are open notes and open book. Absolutely no electronics. This makes cheating hard.

Collaboration and Submitting work

- You can discuss lecture, homework, lab, or programming assignments with anyone. You can share **code *only*** with your programming partner (if you have one).
- You can have one and at most one partner for each programming assignment.
- All other coursework must be done by yourself only. If you discuss with anyone, please document it in your submission.
- The submission strategy will be announced later, but you should ensure your programs run on the MSEE 190/ECN Linux machines or can bring a machine to demo it on
- Having code similar to another project's code may lead to an F in the assignment and/or the course.

Collaboration and Submitting work (2)

- Piazza is something of a safe haven.
- You can answer questions about *classes* to use, etc., but be careful about posting code. If in doubt, don’t

Programming Assignments

- At least two larger programs
  - You can do each assignment alone or work with one (only one) classmate. You **may change** the group mate for each assignment.
- You can discuss programming assignments with anyone but you are not allowed to share code outside your group.
- There will be smaller programming assignments in the form of homework along the way

My goal is to help you learn

- It may not be obvious in the classroom, but I will spend a lot of time on thismclass. My reward for this is that some (and I would like to think all) of you will know significantly more on Dec. 8 about the subject than you do now. My interests are aligned with your long-term interests. I am not your enemy.
- I am a human and therefore have limited powers. Therefore I:
  - may make mistakes.
  - cannot always respond to your email immediately.
  - may be in a meeting when you want to talk. If you come by my office outside of office hours I will likely require you to make an appointment before talking to you.
  - has other responsibility, such as writing papers, working with grad students, committees, seminars, conferences …
- If you make an appointment, make sure we have a set time. When I send you a range of available times **we** need to decide on one -- you shouldn't just "show up" at one of the times.
- No knows all of C++ or Java. If you ask a question I don't know the answer to I will look up the anwer.

Communication

- Email (for things other than questions) is the best way to contact me and I'll use it to contact you for fast things with a copy in Piazza.
- Please put 30862 in the subject line of the email -- it will help it get in the right mailbox and for me to see it.
- If you use an email address other than your Purdue address, let me know that address *in an email from your Purdue address*.
- Don't call and leave a message. My office phone is often ignored.
- Use Piazza for questions.
- Any project/program questions asked in person should be directed to the TAs.

Why OO languages

- Earliest languages (Fortran and Cobol) were tailored to specific domains (matrix oriented numerical applications and business processing with record oriented data)
  - Relatively low level
  - ***Built-in types***, and operations on those types (+, *, array indexing, etc.)
  - Poor support for *encapsulation* -- a change in the data layout of a data structure could require changes throughout the program. Code making use of the data needed to know how the data was declared
- Algol 60 made control flow easier, messier function/subroutine semantics
  - good support for structured programming
  - no goto's, computed gotos, labels much rarer

- Early Basics were syntactically and semantically easy to understand as a language
- Two huge problems:
  - program control flow messed up completely with subroutine calls and gotos, programs hard to understand
  - All data was a a built-in data type. No structs, enums, unions, etc.
  - All variables were global and could be changed anywhere in the program!
- Worked great for simple programs, poorly for large programs

```
…
$a1 = 42
gosub func1
…
more user code
```

```
more user code
func1: if ($a1 > 0) …
return
…
more user code
```

Subroutines were *inline*. You could *gosub* to them or drop through to them

Variables were global. A$1 is accessible every where in the program

- Algol 60 had clean control structures, preceded Basic but was ignored because Basic was simpler to explain.
  - Algol is responsible for the C-like syntax we are all familiar with. Semicolons, nested control structure, etc., were popularized or introduced by Algol.
- Pascal was a rebellion against all primitive types -- introduced abstract types (essentially *structs* and *enums* in C. *Could not declare types and new operations on those types as a single entity.*
- PL/1 was IBM's horribly complicated attempt at a language that could do everything.
- Algol 68 mutated into something hideously complicated as it tried to do everything.
- A consensus began to form in the early 80s: instead of making languages complicated enough to do everything, let programmers add the data structures and operations they need to languages to essentially extend them to new domains. And do this in a way that encourages code re-use and control over how code is used. This is 95% of what OO languages were created for.

A true story

- There are 5,000 programmers working against the same code base
  - Read Fred Brooks, *The Mythical Man Month*
  - *Must* program to interfaces (black boxes), not to internal logic
  - *Must* be able to believe specs, cannot hope to understand what 5,000 other programmers are doing, or be able to talk to them to figure it out
- You have a code base of 1,000,000+ lines that you are debugging
  - OS/360 had 4 or 5 million lines, almost killed IBM
  - Windows XP had a code base of 45,000,000+ lines
    - Was a year or so late
    - Didn’t really faze Microsoft’s revenue stream
- *Object oriented languages are a way to deal with this complexity by programming to specifications using abstractions and specifications – program to the problem not to the machine or needs of the compiler*

- Most programs are big -- maybe not XP big, but big. And getting bigger.
- “Real” code lasts forever – much longer than you think it will when you write it.
- I saw Bill Joy give a talk in the early days of Java, and he said when he was working on BSD Unix while a student at Berkeley, he dreamed of a language like Java.
  - Whenever he changed a line of code in the OS, he never was sure if it would break something thousands of lines away
  - No *encapsulation*, data hiding hard, rogue pointer problems, etc., etc.
  - If he needed a bit, he had know where the bit was in a bit vector and extract it. Couldn’t just ask for the “page readable” bit.
- *Object oriented languages are a way to deal with this complexity by programming to specifications using abstractions and specifications – program to the problem not to the machine or needs of the compiler*

How do chip designers control billions of transistors?

Intel Broadwell, > 7 Billion transistors. IBM Z14 Storage controller, ~10 Billion

- Subcomponents are built and tied together
- Each of these subcomponents is a *black box*
- Designers of chips only care about the interfaces into the black box – not what is done behind the interface
  - If a bug or optimization of a sub-component is made, and the interface is preserved, the global design does not need to be changed.
- This makes designing multi-billion transistor chips possible

What is the greatest enabler of global trade?

- Not OO languages (although that would be a great selling point for the course)
- “The results are striking. In a set of 22 industrialised countries containerisation explains a 320% rise in bilateral trade over the first five years after adoption and 790% over 20 years. By comparison, a bilateral free-trade agreement raises trade by 45% over 20 years and GATT membership adds 285%.”
- More important than costs are knock-on effects on efficiency. In 1965 dock labour could move only 1.7 tonnes per hour onto a cargo ship; five years later a container crew could load 30 tonnes per hour (see table). This allowed freight lines to use bigger ships and still slash the time spent in port. The journey time from door to door fell by half and became more consistent. The container also upended a rigid labour force. Falling labour demand reduced dockworkers’ bargaining power and cut the number of strikes. And because containers could be packed and sealed at the factory, losses to theft (and insurance rates) plummeted. Over time all this reshaped global trade. Ports became bigger and their number smaller. More types of goods could be traded economically. Speed and reliability of shipping enabled just-in-time production, which in turn allowed firms to grow leaner and more responsive to markets as even distant suppliers could now provide wares quickly and on schedule. International supply chains also grew more intricate and inclusive. This helped accelerate industrialisation in emerging economies such as China, according to Richard Baldwin, an economist at the Graduate Institute of Geneva. Trade links enabled developing economies simply to join existing supply chains rather than build an entire industry from the ground up. But for those connections, the Chinese miracle might have been much less miraculous.
- http://www.economist.com/news/finance-and-economics/21578041-containers-have-been-more-important-globalisation-freer-trade-humble

Why do containers work?

- The form a black box around the goods they contain
- The define a single interface that ships, trucks and train cars can be designed against.

OO languages enable encapsulation

- This is one of their great strengths

Inherent and accidental complexity

- No silver bullets
- Languages cannot make the inherent complexity of a problem being solved by computer go away
- Languages can only not get in the way too much, i.e. reduce the accidental complexity of the solution
- To get a 10X increase in productivity, must have 90% of the complexity be accidental, and have a language that makes all of that go away
  - This is not going to happen
  - 10X increase is therefore unlikely
- What we can do is
  - allow the problem to be broken up into smaller problems that multiple teams can work on in parallel
  - allow previous and partial solutions to be reused
  - OO makes it easier to do these things.

From the Mythical Man Month (emphasis mine)

> **High-Level Languages.** Surely the most powerful stroke for software productivity, reliability, and simplicity has been the progressive use of high-level languages for programming. Most observers credit that development with at least a *factor of five in productivity*, and with concomitant gains in reliability, simplicity, and comprehensibility.

What does a high-level language accomplish? It frees a program from much of its accidental complexity. An abstract program consists of conceptual constructs: operations, data types, sequences, and communication. The concrete machine program is concerned with bits, registers, conditions, branches, channels, disks, and such. *To the extent that the high-level language embodies the constructs one wants in the abstract program and avoids all lower ones, it eliminates a whole level of complexity that was never inherent in the program al all.*

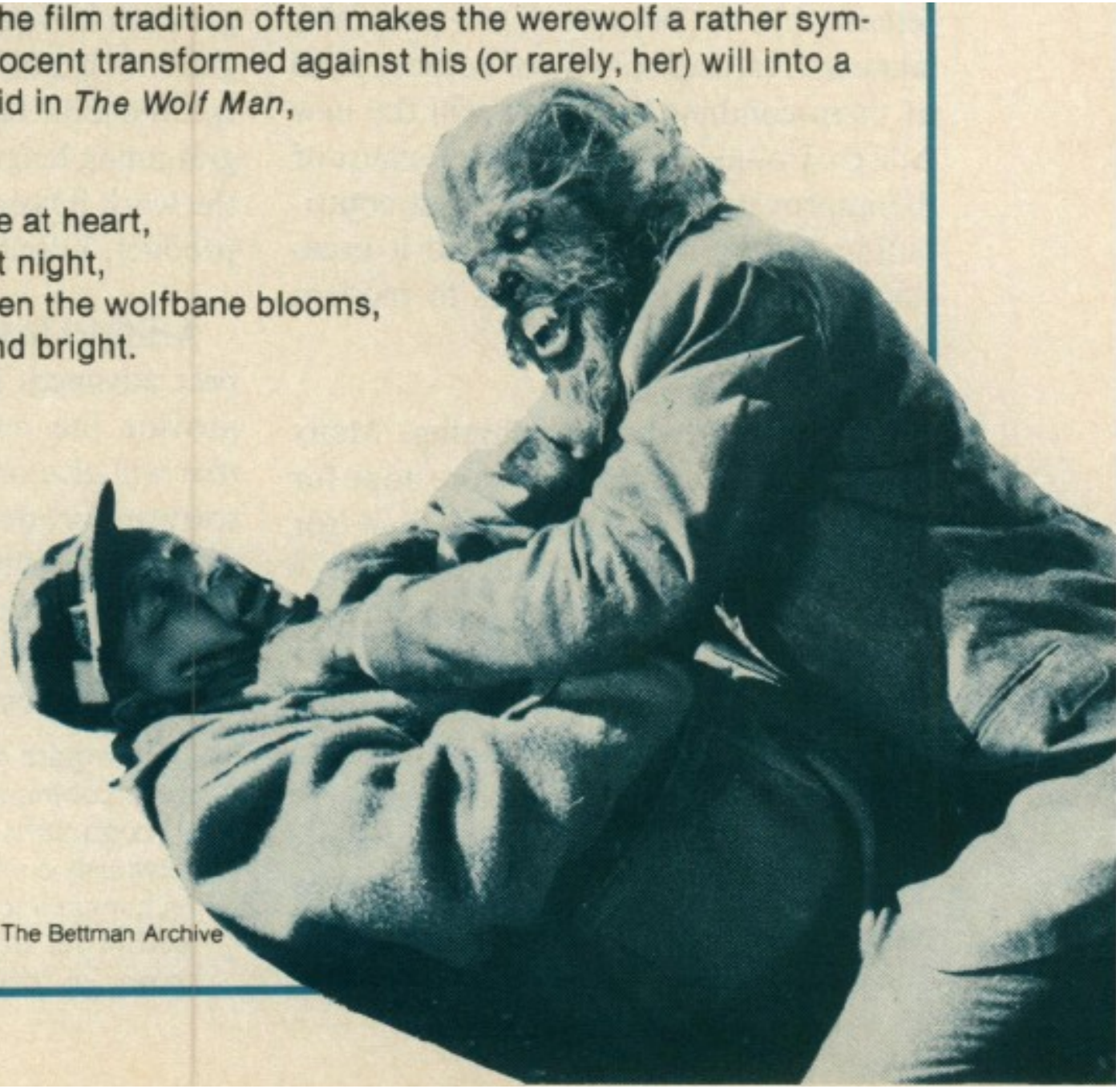
The most a high-level language can do is to furnish all the constructs that the programmer imagines in the abstract program. To be sure, the level of our thinking about data structures, data types, and operations is steadily rising, but at an ever decreasing rate. And language development approaches closer and closer to the sophistication of users.

Moreover, at some point the elaboration of a high-level language creates a tool mastery burden that increases, not reduces, the intellectual task of the user who rarely uses the esoteric constructs.

It is worth noting that the film tradition often makes the werewolf a rather sympathetic character, an innocent transformed against his (or rarely, her) will into a monster. As the gypsy said in *The Wolf Man*,

Even a man who is pure at heart,
And says his prayers at night,
Can become a wolf when the wolfbane blooms,
And the moon is full and bright.

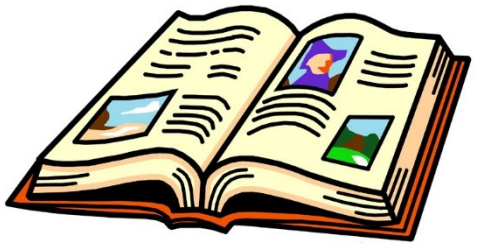
—*Nancy Hays*
Assistant Editor



The Bettman Archive

Objects

- An object can be a “concrete and tangible” entity that can be separated from other objects by **unique properties**:
  - you
  - your book
  - your car
  - my computer
  - Tom
  - Amy’s computer
  - your phone
  - Otto’s dog …



Objects . . .

- An object **can be abstract** and does not have to be tangible (i.e. it carries information but is not physically instantiated). For example, the mean value theorem or Moore's Law.
- An object can **contain** other objects:
  - a car = wheels + engine + door + windshield + …
  - a house = kitchen + bedrooms + living room + …
  - laptop = keyboard + display + processor + …

Objects have three properties

- Each object is unique and can be **identified** using a name, a serial number, a relationship with another object …
- Each object has a set of **attributes**, such as location, speed, size, address, phone number, on/off …
- Each object has unique **behaviors**, such as ring (phone), accelerate and move (car), take picture (camera), send email (computer), display caller (pager)
- Each object has **three** important properties:
  - **unique identity**
  - **attributes**, i.e. holds a state
  - **behavior** (action), verb (*methods* that act on the attributes and other data)

A simple way to think of objects in programs

- An object is a program entity that has:
  - A unique identity. This is often the address of the object in memory, or a 32 bit identity assigned by the programming language runtime
  - Attributes. These are variables that are declared to be part of the class. These variables may be visible outside the class, or private to the class. OO languages provide ways to do this.
  - Actions or methods. These are functions that are associated with the object that can operate on the attributes or variables of the object.

Objects interact

- You (object) press (action) the pedal (object) of your car (object). As a result, your car accelerates (action).
- When your phone (object) rings (action) and alerts (action) you (object) of an incoming call (state), you answer (action) the call (state).
- You submit (action) homework (object) and it is graded (action) with a score (state).

Objects as special case

- A person is an object. A student is also an object. A student is a **special case** of a person
- A student has **all attributes** of a person: name, home address, parents …
- A student has **all behavior** of a person: eat, sleep, talk …
- A student has something that a person may not have:
  - attributes: student ID, list of courses, classmates …
  - behavior/actions: submit a homework, take an exam …
- A particular person, Bob, is a special case of a person

Classes

- A class **describes the commonalities** of similar objects:
  - Person: you, David, Mary, Tom, Amy …
  - Car: your Toyota Camry, his Ford Explorer, Jennifer's Testarossa …
  - Classroom: EE170, EE117, EE129 …
  - Building: EE, MSEE, Purdue Bell Tower, Hovde Hall…
- A class describes both the attributes and the behavior:
  - Person: name, home … + sleep, eat, speak …
  - Car: engine size, year … + accelerate, brake, turn …

A simple way of thinking about classes in OO languages

- A class is a specification of a new type in the language. The new type has declarations for:
  - Attributes or variables
  - Methods or functions
- Objects are created by creating an instance of a class, i.e., by *instantiation*. In particular, an object of type X is is instantiated (or created) using the declaration that is class X.

A class can be a special case of another class

- A class can be a **special case** of another class:
  - Student is a special case of Person
  - Sedan is a special case of Car
  - Laptop is a special case of Computer
  - Computer is a special case of ElectronicMachine
- ⇒ This is called an **"is a"** relationship sometimes written ***isa*** or ***ISA***.
  - any Student object *is a* Person object
  - any Sedan object *is a* Car object
  - any Laptop object *is a* Computer object
  - any Computer object *is a* ElectronicMachine object

Class and Object

- An object is an **instantiation** (i.e. concrete example) of a class:
  - an object is unique
  - a class describes the common properties of many objects
- Each of you can be thought of as a unique student (and therefore an object) that is an instantiation (a concrete example) of the class PurdueStudent.
- An object A may contain an object B *as a field/variable/attribute*. This must be described in A's class. We can say that the A class **"has a"** B class.
- For example, an employee “has a” salary, therefore the Employee class has the ***hasa*** or ***HASA*** relationship with the Salary class. This relationship is best implemented by embedding an object of the Salary class in the Employee class.

ISA vs HASA

- ISA implies *inheritance*
  - Student ISA Person implies Student inherits from, extends or *subclasses* Person
  - Student has the capabilities of a Person because a Student ISA person
  - Student gets to reuse methods and attributes in (the interface of) Person as if they were part of Student
- HASA implies *declaration* of an attribute of the kind of object
  - If Student HASA Address the student has a field that is an Address object, or points to an Address object
  - Student gets to use the methods and attributes in the Address object

Class declaration and definition example in C++

Person.h

```
#include <string>
using namespace std;

class Person {
public:
   std::string name;

   Person( );
   ~Person( );

   virtual void setName(string);
   virtual string getName( );

};
```

See basicClass code, Person.h Person.cpp

Person.cpp

```
#include "Person.h"

Person::Person( ) { }
Person::~Person( ){ }

void Person::setName(string n) {
   name = n;
}

tring Person::getName( ) {
   return name;
};
```

Creating and using objects

```
#include "Person.h"
#include <iostream>

int main (int argc, char *argv[]) {

   Person* p1 = new Person( ); // create Person object, assign address to p1
   Person* p2 = new Person( ); // create Person object, assign address to p2

   p1->setName("Bill"); // call setName function on object pointed to by p1
   p2->setName("Maria"); // call setName function on object pointed to by p2



   std::cout << "p1:" << p1->getName( ) << std::endl; // print out name fields
   std::cout << "p2:" << p2->getName( ) << std::endl; // for the two objects
}
```

Inheritance = “ISA”

*Allows the problem to be broken up into smaller problems that multiple teams can work on in parallel AND **allows previous and partial solutions to be reused***

- Any Student object **is a** Person object. Student class is a **derived class** of Person. Person is the **base class**.
  - Person is more general, with fewer attributes and behaviors.
  - Student is more specific, with more attributes (school, major) and behaviors (submit homework, take exam).
- Any TabletPC object **is a** Computer object. TabletPC class is a **derived class** of Computer.
  - Computer is more general.
  - TabletPC is more specific, with more attributes (battery lifetime) and behavior (close or turn the screen)

A Student ISA Person

```
#include "Person.h"

class Student : public Person {
public:
   int id;

   Student( );
   virtual ~Student( );

   virtual void setID(int);
   virtual int getID( );

};
```

Student.h

```
#include "Student.h"

Student::Student( ) { }
Student::~Student( ) { }

void Student::setID(int i) {
   id = i;
}

int Student::getID( ) {
   return id;
};
```

Student.cpp

A student ISA person

```
#include "Student.h"
#include <iostream>

int main (int argc, char *argv[]) {

   Student* s = new Student( );
   s->setName("Maria"); // Call to inherited Person class setName( )!
   s->setID(1); // Call to function in Student class

   std::cout << "s: " << s->getName( );
   std::cout  << “, “ << s->getID( ) << std::endl;
}
```

Derived Class

- A class may have **multiple** derived classes:
  - Vehicle: Sedan, Truck, Sport Utility Vehicle, Sport Car ...
  - Computer: Laptop, Desktop, Server
  - Person: Student, Teacher, Father, Mother ...
- A derived class may also have derived classes:
  - Vehicle: Car, Bike ... Car: Sedan, SUV ...
  - Animal: Bird, Mammal ... Mammal: Dog, Cat ...
  - We will use "base" and "derived" classes. **I will not** use "super" and "sub" classes. A
- base class or a superclass is "smaller" (fewer attributes and behaviors)
  - ⇒ too confusing, but worth knowing the terminology

Encapsulation

- ***allows the problem to be broken up into smaller problems that multiple teams can work on in parallel*** *AND allows previous and partial solutions to be reused*
- An object can hide information (**attributes**) from being manipulated by or even be visible to other objects:
  - A person's name is given once when the object is created. This attribute is visible but cannot be changed.
- An attribute may only be modified through restricted channels to maintain consistency.
  - A person's address may have limited ways of changing that ensure the addresses validity is checked.
- Who can call methods can be controlled

Why Encapsulation?

- Prevent accidental modification of objects' attributes
  - If an attribute is private, we can only change it through class methods
- To hide implementation details
- To maintain data consistency (some attributes must be changed simultaneously)
  - If you want the address and phone number to always be updated together, can make the fields private and have a method that does the update and requires both items.

Why Encapsulation?

- To make it clear what maintainers of the class can change, and what they should not change
  - public method interfaces or actions cannot be changed without breaking code that uses it
  - protected may be able to be changed if we know who extends the class
  - private can be changed because we have access to all calls to that method
- To give those maintainers flexibility in their implementation
- One thing that has lead to the success of OOP:
  - large standard libraries providing a great deal of functionality
  - the algorithms used by these libraries can be improved and targeted to different underlying machines and operating systems without breaking the code that uses the libraries

What is the interface for a class’ object?

- A class' public attributes and methods form the *interface* for the objects of this class and all derived classes.
  - public attributes are attributes visible outside the class, i.e. to external users of the class
- if *func* is a *public* method, the object must be able to respond to a call of *func* as defined by the specification for the class.
  - The specification is a promise or contract that cannot be withdrawn (you, as a programmer, enforce this, not the compiler)
  - the specification is often informal (i.e. documentation or a manual) -- it is rarely a formal, executable thing, although this has been done.
  - a method is simply a function or subroutine that is a member of a class and that operates on the data declared by the class.
  - In Java, all functions are methods.
  - In C++, in addition to methods, you can have functions that exist outside of classes to maintain compatibility with C. These are functions, not methods.

Interfaces should be implementation independent

- Interfaces are what are visible to other programmers, and classes and objects
- Lets assume you have a class for a student record
- If you call getGPA( ) on a student object, you shouldn't care how it is computed, as long as it is correct
  - Could be pre-computed and stored as a fixed decimal value
  - Could be retrieved from a database
  - Could be computed from "total hours" and "total credits" fields
- If you want a GPA, all you really care about is that you get a valid GPA
- You do not need to know how the Student class implements the method -- you only need the interface to get the method from a student object

Interface ≠ Implementation

- Stated simply, the interface is the collection of methods and variables (attributes) in a class.
  - Often no variables are visible – variables are accessed via *getter* and *setter* functions, e.g., *getName( )*, *setName( )*.
  - Method prototypes, e.g., *findAddress(String name)* are visible and part of the interface.
- How *findAddress* is implemented should not be relied upon. Only the interface and associated documentation should be relied upon.
- How getName in Person is implemented should not matter – only that it returns the object's name as a string.

- Suppose you are creating a class to support searching for books in a library
- When the library has only hundreds of books, you can use a list to store each book (author, title, year …)
  - As the number of books grows, you may need to use a more sophisticated indexing scheme (e.g. a hash table), an SQL database, or ….
  - The class should provides the same interface to search books.
  - Users of functionality provided by the class should not care how the search is implemented, only that it is and that they don’t have to change their code when your class changes its implementation
- Encapsulation is the ability to hide the implementation of an interface.
  - enforced by compilers (no run-time surprises)

- Code reuse is enabled by promises that cannot be withdrawn.
  - Without the promise or irrevocable contract, you cannot count on what the method will do later in time
- Anything not in the interface (private attributes and methods) can be changed by the class maintainer
  - Provides flexibility to the class designer in implementation and maintenance
  - If you as a user of the class try and break in and use private attributes you are a bad (as in not good, as in making life a pain for yourself and others) programmer
- As a class designer - keep interfaces as small as possible.
  - Make visible only the essential functionalities required by the problem spec.
  - Hide all details that you can
  - Give yourself as much freedom as possible to make changes to the private stuff. It can save you days, weeks or years of work, and maybe save your job!
- And small interfaces help you – when there is a bug there are fewer ways variables can be changed.

Consider the Student class again

- We can write code using *Student* that looks like the code to the right.
- The program directly changes the *ID* and does not use *setID*.
- What if the writer of the Student class wants to perform checks on the *ID* in *setid*, and therefore wants to ensure that only *setID* sets the *ID*?

```
#include "Student.h"
#include <iostream>

int main (int argc, char *argv[]) {

  Student* s = new Student( );
  s->setName("Maria");
  s->id = -1;

  std::cout << "s: " << s->getName( );
  std::cout  << ", " << s->getID( ) << std::endl;
}
```

Encapsulation to the rescue!

```
#include "Person.h"
class Student : public Person {

public:
   Student( );
   virtual ~Student( );
   virtual void setID(int);
   virtual int getID( );

private:
   int id;

};
```

StudentE.h and
StudentE.cpp

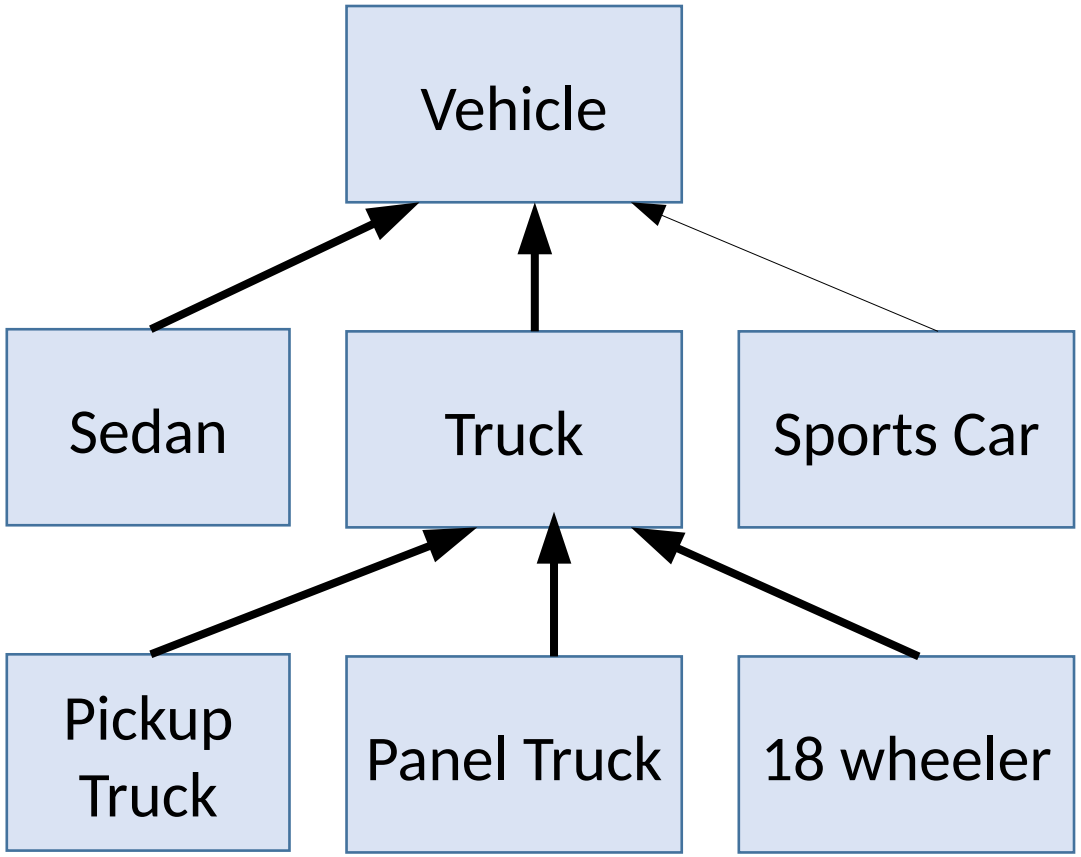
```
#include "Student.h"
#include <iostream>

Student::Student( ) { }
Student::~Student( ) { }

void Student::setID(int i) {
   if (i > 0) id = i;
   else std::cout << "Err, id < 0 " << i << std::endl;
}

int Student::getID( ) {
   return id;
};
```

Attempt to bypass setID gives an error

```
#include "StudentE.h"
#include <iostream>

int main (int argc, char *argv[]) {

  Student* s = new Student( );
  s->setName("Maria");
  s->id = -1;

  std::cout << "s: " << s->getName( ) << ", " << s->getID( ) << std::endl;
}
```

```
g++ mainStudentE.cpp
In file included from mainStudentE.cpp:1:0:
StudentE.h: In function ‘int main(int, char**)’:
StudentE.h:14:8: error: ‘int Student::id’ is private
   int id;
       ^
mainStudentE.cpp:8:7: error: within this context
   s->id = -1;
```

Some useful terminology



We say that a Truck is *derived from* (or *inherits from*) a Vehicle.

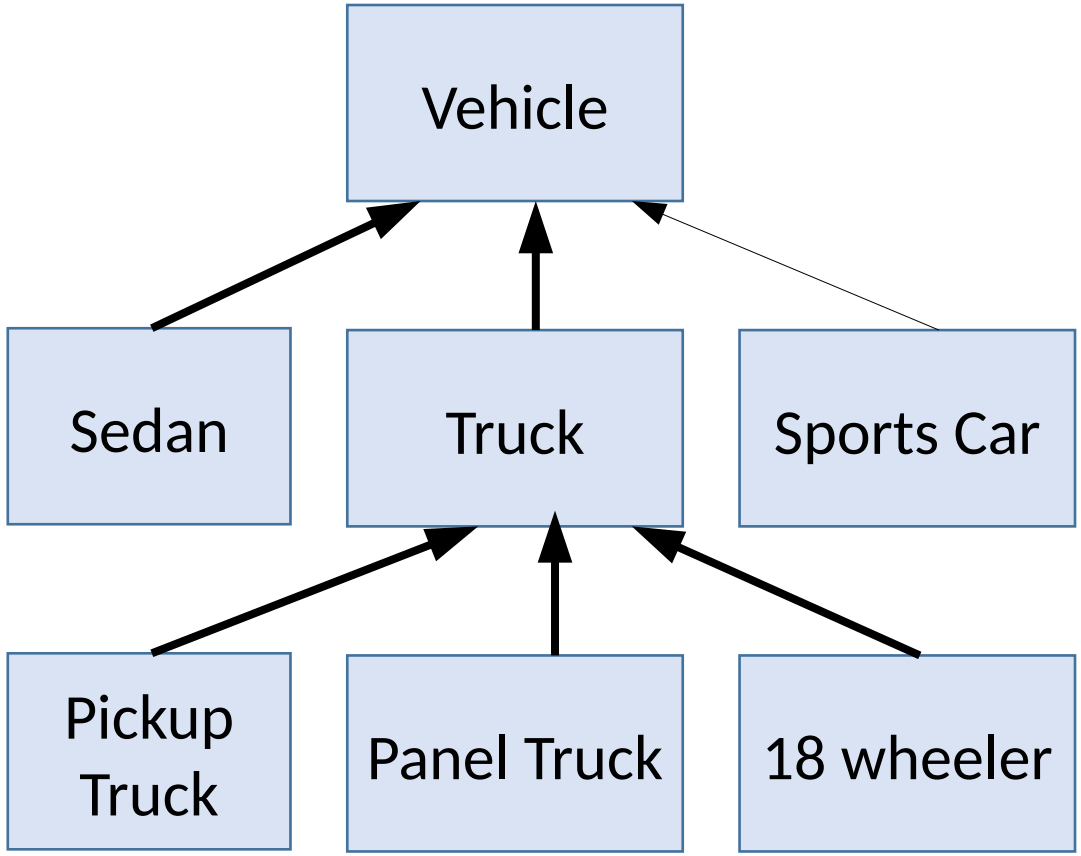
The same for Sedan and Sport Car

We can also say that a Panel Truck *is derived from* (or *inherits from*) a Truck.

The same is true for Pickup Truck and 18 Wheeler

The *Base Class* for all of these is the Vehicle class.

Some less useful terminology



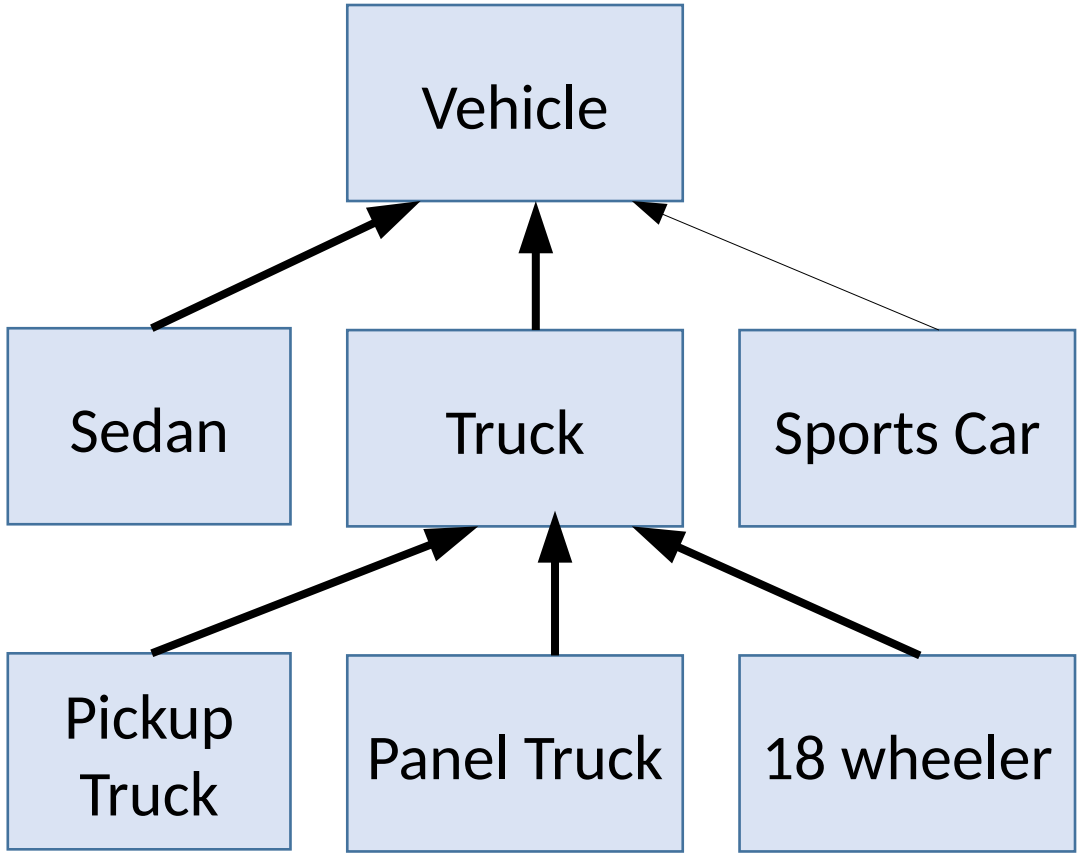
We say that a Truck is a *sub-class* of Vehicle. We can also say that a Panel Truck is a *sub-class* of a Truck.

We can say that Truck *subclasses* Vehicle, and that PanelTruck *subclasses* Truck

The *Superclass* for all of Sedan, Pickup truck, etc. is the Vehicle class.

This terminology works if you think of superclasses as being higher in the class hierarchy, and sub-classes being below them.

Some less useful terminology



This terminology doesn't work when you think of the number of methods (actions) and attributes (state or variables) defined by the class.

The available, publicly accessible methods of Truck are a superset of those of Vehicle.

What happens if a derived class and a base class both define a method m( )?

- If a class Derived inherits from a class Base, where Base declares a defines a method (function) m( )
  - Derived can redefine m( ) to act like it wants.
  - This is called *overriding*, i.e., *Derived overrides Base's m( )*.
- This allows us to use desired functionality from the Base class and redefine functionality that doesn't suit our purposes
- Allows better code reuse

Overriding summary – general OO (C++ and Java have a few wrinkles)

Program	Base	Derived	Object M called on	What is executed
HH.cpp	Implements M	Implements M	Base	Base M
HH.cpp	Implements M	Implements M	Derived	Derived M
HN.cpp	Implements M	Does not implement M	Base	Base M
HN.cpp	Implements M	Does not implement M	Derived	Base M
NH.cpp	Does not implement M	Implements M	Base	Error
NH.cpp	Does not implement M	Implements M	Derived	Derived M
NN.cpp	Does not implement M	Does not implement M	Base	Error
NN.cpp	Does not implement M	Does not implement M	Derived	Error

Program	Base	Derived	Object M called on	What is executed
HH.cpp	Implements M	Implements M	Base	Base M
HH.cpp	Implements M	Implements M	Derived	Derived M

```
class Base {
public:

   Base( );
   virtual ~Base( );
   virtual void m( );

};
```

```
#include <iostream>
#include "Base.h"
using namespace std;

Base::Base( ) { }
Base::~Base( ){ }

void Base::m( ) {
   std::cout << "Base:m( )" << std::endl;
}
```

```
Derived::Derived( ) { }
Derived::~Derived( ){ }

void Derived::m( )   {
   std::cout << "Derived:m( )" << std::endl;
}
```

```
class Derived : public Base {
public:

   Derived( );
   virtual ~Derived( );
   virtual void m( );

};
```

Not necessary here
because it is already made
virtual in Base

Program	Base	Derived	Object M called on	What is executed
HH.cpp	Implements M	Implements M	Base	Base M
HH.cpp	Implements M	Implements M	Derived	Derived M

```
#include "Base.h"
#include "Derived.h"
#include <iostream>

int main (int argc, char *argv[]) {
   Base* b = new Base( );
   Derived* d = new Derived( );
   Base* b2d = d;

   std::cout << "calling m( ) on b " << std::endl; b->m( );
   std::cout << "calling m( ) on d " << std::endl; d->m( );
}
```

```
calling m( ) on b
Base:m( )
calling m( ) on d
Derived:m( )
```

Program	Base	Derived	Object M called on	What is executed
HN.cpp	Implements M	Does not implement M	Base	Base M
HN.cpp	Implements M	Does not implement M	Derived	Derived M

```
class Base {
public:

   Base( );
   virtual ~Base( );
   virtual void m( );

};

Base::Base( ) { }
Base::~Base( ){ }

void Base::m( ) {
   std::cout << "Base:m( )" << std::endl;
}
```

```
class Derived : public Base {
public:

   Derived( );
   virtual ~Derived( );

};

Derived::Derived( ) { }
Derived::~Derived( ){ }
```

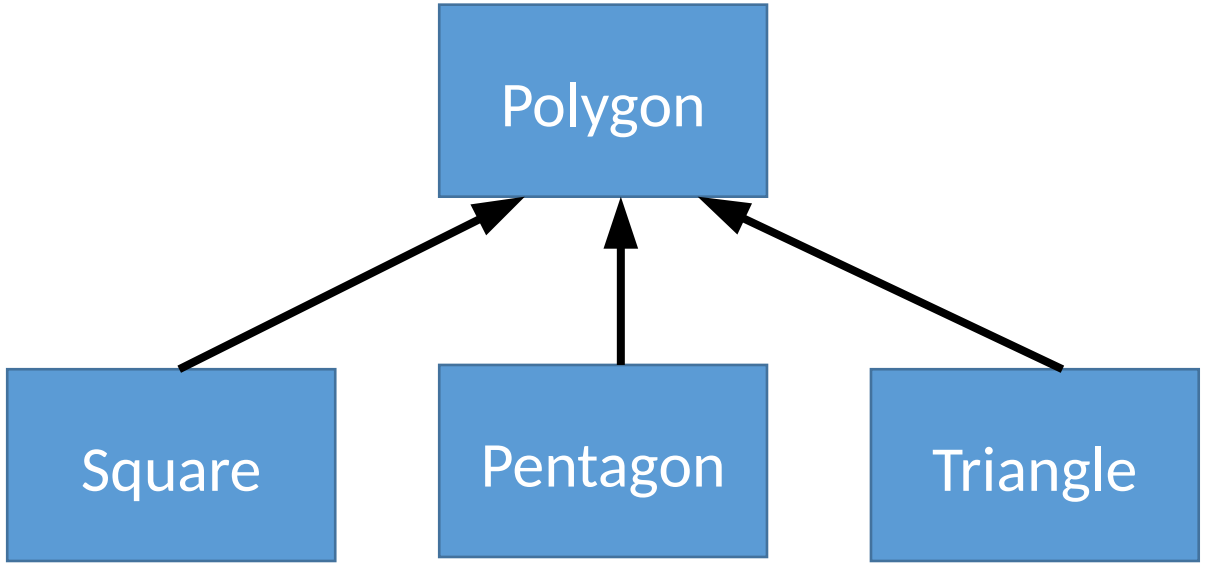
Program	Base	Derived	Object M called on	What is executed
HN.cpp	Implements M	Does not implement M	Base	Base M
HN.cpp	Implements M	Does not implement M	Derived	Base M

```
int main (int argc, char *argv[]) {
   Base* b = new Base( );
   Derived* d = new Derived( );
   Base* b2d = d;

   std::cout << "calling m( ) on b " << std::endl; b->m( );
   std::cout << "calling m( ) on d " << std::endl; d->m( );
}
```

```
calling m( ) on b
Base:m( )
calling m( ) on d
Base:m( )
```

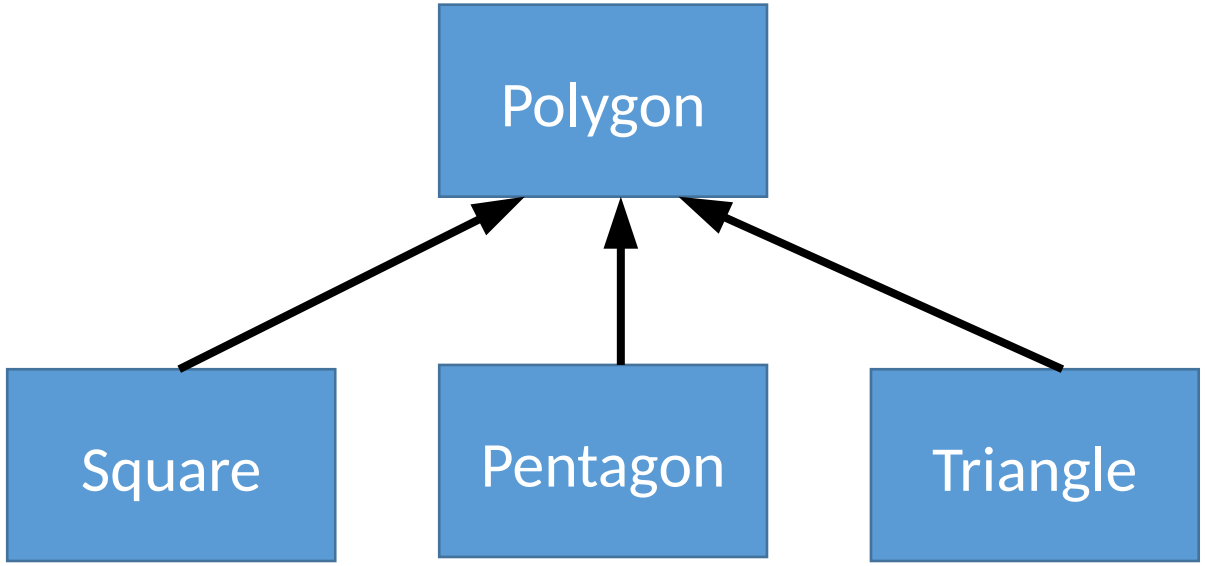
Polymorphism

- Literally means “many forms”
- It is the ability of an object to act like objects of other types, specifically, objects its derives from.
- Consider the class hierarchy to the right and the code on the following slides



Polymorphism

- What we would like to do is to create Square, Pentagon, Triangle and Polygon objects and have each define a getArea( ) method.
- Then, we would like to take a vector of these objects and find the total area in all of them
- Code to do this follows.



The Polygon class

```
#include <string>

class Polygon {
public:
  Polygon( );
  Polygon(int,float);
  virtual ~Polygon( );

  virtual float getArea( );

protected:
  int numSides;
  float lenSides;

};
```

```
#include "Polygon.h"
#include <math.h>

Polygon::Polygon(int n,float l) :
  numSides(n), lenSides(l) { }

Polygon::Polygon( ) { };
Polygon::~Polygon( ) { };

float Polygon::getArea( ) {
  float p = numSides*lenSides;
  float apo = numSides / 2. * tan(180. / numSides);
  return apo*p / 2.;
}
```

The Triangle class

```
#include "Polygon.h"
#include <string>

class Triangle : public Polygon {
public:
  Triangle(float);
  ~Triangle( );

  float getArea( );

};
```

```
#include "Triangle.h"
#include <math.h>

Triangle::Triangle(float l) : Polygon(3, l) { }

Triangle::~Triangle( ) { };

float Triangle::getArea( ) {
  return 0.433 * lenSides * lenSides;
}
```

Defines its own getArea class that overrides Polygon's getArea class.
Because the base Polygon class getArea is declared virtual, we don't need to do that in the derived Triangle class.

The Square class

```
#include "Polygon.h"
#include <string>

class Square : public Polygon {
public:
  Square(float);
  ~Square( );

  float getArea( );

};
#endif
```

```
#include "Square.h"
#include <math.h>

Square::Square(float l) : Polygon(4, l) { }

Square::~Square( ) { };

float Square::getArea( ) {
  return lenSides * lenSides;
}
```

Defines its own getArea class that overrides Polygon's getArea class.

The Pentagon class

```
#include "Polygon.h"
#include <string>

class Pentagon : public
Polygon {
public:
   Pentagon(float);
   ~Pentagon( );
};
```

```
#include "Pentagon.h"
#include <math.h>

Pentagon::Pentagon(float l) : Polygon(5, l)  { }
Pentagon::~Pentagon( ) { }
```

Uses the getArea method in the base Polygon class

The main function

```
// elided includes of .h files and iostream
int main (int argc, char *argv[]) {

   Polygon* p[4];
   float area = 0;

   p[0] = new Square(4.0); p[1] = new Triangle(4.0);
   p[2] = new Polygon(8, 4.0); p[3] = new Pentagon(4.0);

   for (int i=0; i <4; i++) {
      area += p[i]->getArea( );
      std::cout << "area of poly " << i << " = " << p[i]->getArea( ) << std::endl;
   }
   std::cout << "total area = " << area << std::endl;
}
```

```
area of poly 0 = 25.9164
area of poly 1 = 2.88036
area of poly 2 = 35.7025
area of poly 3 = 193.762
total area = 258.261
```

Fundamental OO concepts

- We'll spend the most of the rest of the semester looking at the details and variations of what we've covered so far:

  - **object and class**
  - **encapsulation**
  - **inheritance**
  - **polymorphism**