# COMSC-200 Lecture Topic 2
# Encapsulation

### Reference
Deitel Ch.9.1-4

### The 3 OOP Subject Areas
Encapsulation (data+functions together)
Inheritance (only-child subclasses)
Polymorphism (sibling-children subclasses)

### Object-Oriented Thinking
building programs in pieces
classes vs objects: concept vs reality
accessors or "getters"
mutators or "setters"
the "public interface"
e.g., an elevator simulation
  Rider, Floor, Elevator, Building

### Structure Definitions
"aggregate data types"
e.g.: `struct Time`
e.g.: `struct Elevator`
memory allocation for a `struct`
`struct` variables
`struct` brace initialization
`struct` references and pointers
`struct` arrays
`struct`s as parameters
  copy
  mutable reference
  immutable reference
`struct`s as return types
`class` vs `struct` keywords
  public vs. private

### Data Members
the dot-operator: `t.hour` -- *a.k.a. "member-of"*
dereferencing: `(*p).hour` -- *using dereference and member-of*
the arrow-operator: `p->hour` -- *a.k.a. "infix"*

### Member Functions
getters, setters, constructors, destructors, etc.

### Implementing `struct Time`
attributes: hour, minute, second
accessors: output in 12- and 24-hour formats
mutators: `setHour`, etc.

### Implementing `struct Elevator`
attributes: capacity, #riders, direction
accessors: hasRiders, isIdle, isGoingUp
mutators: load, unload, move

### Implementing Encapsulation
using the `public` keyword
move function prototypes into structure definition
  inline functions
  the *scope resolution operator*
using the `private` default for member data
  supplying "setter" functions
"constructor" functions for initialization
e.g., `class Time`
brace initialization still okay...
  until constructors are added!

### Data Hiding
e.g., changing how `class Time` stores its data
  `int h, m, s;` vs.
  `int s; // 0-86399`

### Arrays Of Objects
object arrays
  `Time ta[10];` *// requires default constructor*
    ...or no constructor at all
arrays of variable mutating pointers
  `Time* ta[10];`
  `ta[0] = new Time;` *// req. default constr. or none*
  `delete ta[0];`
arrays of mixed objects
  `void* ta[10];`
  `ta[0] = new Time;`
  `ta[1] = new Date;`
  `...((Time*)ta[0])->hour...` *uses type casting*
  `...((Date*)ta[1])->year...` *uses type casting*
  `delete (Time*)ta[0];` *allows destructor to be called*
  `delete (Date*)ta[1];` *allows destructor to be called*
object-type tracking
  `int taId[10] = {};`
  `taId[0] = TIME;` // const int TIME = 1;
  `taId[1] = DATE;` // const int DATE = 2;
  `for(int i = 0;...`
    `switch(taId[i])`

### Using Conditionals
`char* a =...` *(possibly zero)*
`int b = a ? atoi(a) : 0;` // requires cstdlib
*atoi(0) causes unpredictable results -- even fatal errors!*

### Common Mistakes
do NOT use scope resolution inside class definitions

```
class X
{
  ...
  void ~~X::~~fun(); // NO X::
};
```

**Inline Functions in OOP:**
Regular class member functions have prototypes inside the class definition, and their own definitions outside. Using H and CPP files, a class definition would be in the H file and function definitions in the CPP. But inline functions are fully written inside the class

definition in place of the prototype. They have no outside definition, and would not appear in a class' CPP. There is no need for the `inline` keyword, and it should not be used.

In this course, use inline functions when directed to do so. When not specified, you *may* use inline functions, but only for functions consisting of one statement.

### Non-inline member functions

```
class Time
{
  ...
  int getHour() const;
  void setHour();
};
...
int Time::getHour() const
{
  return hour;
}

void Time::setHour(int h)
{
  hour = h;
}
```

### Inline member functions

```
class Time
{
  ...
  int getHour() const {return hour;}
  void setHour() {hour = h;}
};
```