

IT1050- Object Oriented Concepts

Lecture-01 – Introduction to C++

Agenda

- Introduction to Module
 - Recalling C
 - C to C++

Introduction to Module

Learning Outcomes

- Understand and apply the basic concepts of Object Oriented Programming
- Design solutions by identifying the classes and relationships (Object Oriented Analysis and Design)
- Implement a solution to the given problem using the C++ Language

Delivery

- Lectures
 - 1 Hour per week
- Tutorial
 - 1 Hour per week
- Labs
 - 2 Hours per week

Assessment Criteria

- Continuous Assessment
 - Assignment 1 - Practical 10%
 - Assignment 2 - Group work (case study) 10%
 - Mid Term Examination 20%
- Final Examination 60%

Content

- **Introduction to C++**
- **Introduction to OOP Concepts**
 - Abstraction
 - Encapsulation
 - Information Hiding
- **Identifying classes and objects**
- **Object Oriented Design**
 - Noun Verb Analysis
 - CRC Cards
- **Introduction to Object Oriented Programming**
- **Advanced Object Oriented Concepts**
 - Relationships
 - Polymorphism

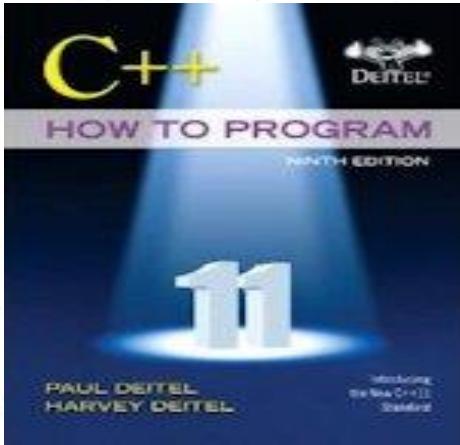
Teaching Learning Activities

- Case Study - Library System
- Home work - Watching Videos
- Group work - Assignment 2

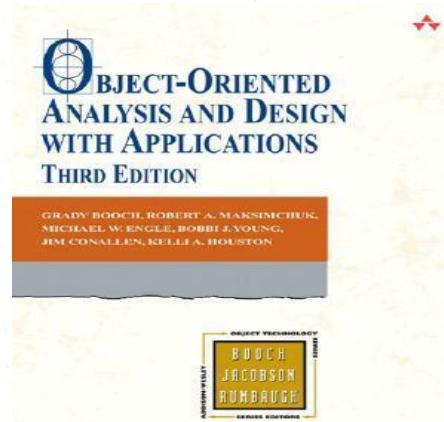
Academic Integrity Policy

- Are you aware that following are not accepted in SLIIT???
- Plagiarism - using work and ideas of other individuals intentionally or unintentionally
- Collusion - preparing individual assignments together and submitting similar work for assessment.
- Cheating - obtaining or giving assistance during the course of an examination or assessment without approval
- Falsification – providing fabricated information or making use of such materials
- From year 2018 the committing above offenses come with serious consequences !
- See General support section of Courseweb for full information.

Reference



Deitel & Deitel's (2016),
C++ How to Program, 9th
Edition



Grady Booch (2008), Object-
Oriented Analysis and Design
with Application,
3rd Edition

C++

- One of the most powerful and popular programming languages
- Evolve from C
- Developed by Bjarne Stroustrup in 1979 at Bell Laboratories
- Provide capabilities for Object Oriented Programming
- Current Version – C++ 17



C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows away your whole leg.

— Bjarne Stroustrup —

AZ QUOTES

C vs C++

```
// C Program  
  
#include <stdio.h>  
  
void main ( void )  
{  
  
    printf ("Hello World ! \n");  
  
}
```

```
// C++ Program  
#include <iostream>  
  
int main ( )  
{  
    std::cout<< "Hello World !";  
    std::cout<< std::endl;  
  
    return 0;  
}
```

Output :

Hello World !

First C++ Program

```
// C++ Program : prg_01.cpp
//Printing a String
#include <iostream> // allows program to output data to the screen

int main ( ) // Function main begins program execution
{
    std::cout<< "Hello World !"; // Display message
    std::cout<< std::endl; // New line

    return 0; // indicate that program ended successfully

} // End of main function
```

Comments

```
// C++ Program : prg_01.cpp  
//Printing a String
```

- Comments provide information to the people who read the program
- Comments are removed by the preprocessor, therefore the compiler ignores them
- In C++, there are two types of comments
 - Single line comments //
 - Delimited comments /* */ for comments with more than one line.

Preprocessing Directives

#include <iostream>

- Lines begin with # are processed by the preprocessor before the program is compiled.
- Notifies the preprocessor to include in the program the content of the input/output stream header <iostream>
- “iostream” is a header file containing information used by the compiler when compiling a program with output data to screen or input data from the keyboard using c++ input/output stream

The main function

```
int main()
{
}
```

- C++ programs begin executing at function `main`.
- It is the main building block of a program.
- `int` indicates that `main` returns an integer value.
- `{` (left brace) indicates the begin of the main body and `}` (right brace) indicates the end of the function's body.

Output Statement

```
std:: cout<< "Hello World ! " ;
```

- **cout** : to indicate the computer to output something on screen
- **<<** : is the stream insertion operator used to send information to cout
- **"Hello World ! "** : String / String Literal. What you need to display on screen
- **;** : statement terminator

New Line

```
std:: cout<< endl ;
```

- **endl** : to go to a new line (same as “\n”)
eg : std::cout<<“\n”;

```
std:: cout<< “Hello World !”<< endl ;
```

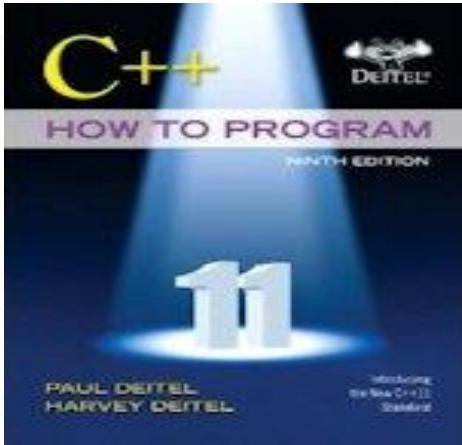
Output :

Hello World !

Exercise

- Write a C++ program to display your name and address in 3 lines.

Reference



Chapter 01 & 02

Deitel & Deitel's (2016), C++ How to Program,
9th Edition

IT1050- Object Oriented Concepts

Lecture – 02 - C++

Learning Outcomes

At the end of the Lecture students should be able to
Write a C++ program including :

- Namespaces
- Variables
- Sequence
- Selection
- Repetition
- Using Input Commands and Formatting Output

The std namespace

- Let's have a look at the iostream.h header file (a simplistic view of the actual file)

```
// iostream.h header file
// this is inserted to your program when you use the command #include <iostream>

namespace std {

    // various commands related to input and output are defined here

    ofstream cout;

    //
    ifstream cin;

    char endl = '\n';
}
```

To access cout, cin, endl outside the namespace we have to explicitly use

std::cout
std::cin
std::endl

Everything defined in the iostream header file is defined under a namespace called std

namespaces are used to avoid naming collisions

```
// I could write my code for example  
// using the namespace FOCSLIIT
```

```
// Imagine a namespace to be a folder  
// in your computer.
```

```
namespace FOCSLIIT {  
    int data;  
    void graphics(int x, int y);  
}
```

```
namespace Graphics {  
    void graphics(int x, int y);  
    int mygraphics;  
    int data;  
}
```

```
// Since FOCSLIIT and Graphics are  
// two separate namespaces (folders)  
// variables, functions with the same  
// name can exist without issues
```

FOCSLIIT::data
FOCSLIIT::graphics(10,20);

Graphics::data
Graphics::graphics(10,20);

namespace1.cpp

namespace2.cpp

The std namespace

- The `std ::` before `cout` is required when we use names that we've brought into the program by the preprocessing directives `#include <iostream>`
- Means `cout` belongs to namespace `std`

* * *
* * *
* * *
* * *
* * *
* * *
* * *
* * *
* * *
* * *

```
// C++ Program
#include <iostream>

int main ( )
{
    std::cout<< "Hello ";
    std::cout<<"World !";
    std::cout << std::endl;

    return 0;
}
```

```
// C++ Program
#include <iostream>
using namespace std;

int main ( )
{
    cout<< "Hello ";
    cout<<"World !";
    cout << endl;

    return 0;
}
```

namespace std

using namespace std;

- The keyword `namespace` defines a scope
- `std` is a namespace defined by C++
- `cout` is included in `std`
- Using the above statement will omit having to use `std::` (`::` - scope resolution operator) with every member (directive/keyword) of the `std` namespace

Chaining multiple << operators together

- Instead of using only one << for each cout. We can chain multiple insertion operators in the same line. Each of the data can be of different types.

```
// C++ Program
#include <iostream>
using namespace std;

int main ( )
{
    cout << " My Score is ";
    cout << 70;
    cout << endl;

    return 0;
}
```

```
// C++ Program
#include <iostream>
using namespace std;

int main ( )
{
    cout << "My Score is " << 70 << endl;
    return 0;
}
```

C++ Keywords

Table 4 — Keywords

alignas	continue	friend	register	true
alignof	decltype	goto	reinterpret_cast	try
asm	default	if	return	typedef
auto	delete	inline	short	typeid
bool	do	int	signed	typename
break	double	long	sizeof	union
case	dynamic_cast	mutable	static	unsigned
catch	else	namespace	static_assert	using
char	enum	new	static_cast	virtual
char16_t	explicit	noexcept	struct	void
char32_t	export	nullptr	switch	volatile
class	extern	operator	template	wchar_t
const	false	private	this	while
constexpr	float	protected	thread_local	
const_cast	for	public	throw	

Use of Variables

- Same as in C

```
// prg_02.cpp
//Program that adds two numbers
#include <iostream>
int main ( )
{
    int number1 = 25;
    int number2 = 32;
    int sum;
    sum = number1 + number2;
    cout<< "Sum is : " << sum << endl; // Display value of
sum
    return 0;
}
```

Recall.....

- C++ Rules for making identifies
 - Consists with letters, digits, and underscore character
 - Starts with a letter
 - Cannot contain spaces, special characters, operators and reserve words / keywords
 - Cannot contain more than 31 characters



Input from Keyboard

```
// C Program  
#include <stdio.h>  
  
void main ( void)  
{  
    int num;  
    printf ("Input Number : ");  
    scanf("%d", &num);  
    printf("Number : %d\n", num);  
}
```

```
// C++ Program  
#include <iostream>  
using namespace std;  
  
int main ( )  
{  
    int num;  
    cout<< "Input Number :";  
    cin >> num;  
    cout<<"Number is : "<< num  
                     << endl;  
  
    return 0;  
}
```

cin command

- `cin` is the stream input in the C++ standard library.
- The Extraction operator `>>` will skip leading whitespace
 - (blank, tab, newline) characters and start the value with the first non-whitespace characters.
- A value is terminated by whitespace.
- e.g:

Output

```
cout<< "Input length and width :";  
cin >> length >> width;
```

Input length and width : 7.5 8.5

Exercise 01

- Write a C++ program to input the length and the width of a rectangle and calculate and print the perimeter.

Try this in **repl.it** using your account. Copy your solution's url to chat window

Formatting Output – <iomanip>

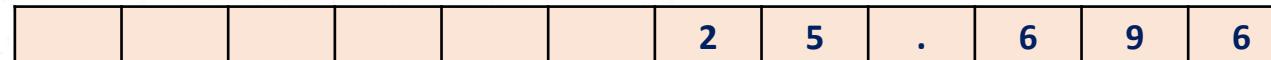
- These allow you to send control signals to cout to control how the output is displayed.
- **setw (n)**
 - Specifies number of spaces used to display a number
- **setiosflags (ios::fixed)** – Specifies that the number should be printed as floating point number with decimal places e.g. 345.67
- **setprecision (n)**
 - When used with ios::fixed, controls the number of decimal places that will be printed.

Formatting Output cont....

```
cout<< setw(12) << setiosflags(ios::fixed) << 25.695789<< endl;
```



```
cout<< setw(12) << setprecision (3) << 25.695789<< endl;
```



```
cout<< setw(12) << setprecision (5) << 25.695789 << endl;
```



iomanip.cpp

Exercise 02

- Modify the program that was written to calculate the perimeter of the rectangle to display the results using two decimal places.

Try this in **repl.it** using your account. Copy your solution's url to Slido.com

Selection Control Structure

- if
 - if- else
 - switch

- ```
if(a >b)
{
 cout <<a <<“is the largest”<<endl;
}
```
- ```
if ( a > b )
    cout<< a << “is greater than”<< b<<endl;
else
    cout<<b <<“is greater than”<<a<< endl;
```

Selection Control Structure

```
if( score == 4 )
    cout << "Excellent" << endl;
else
    if ( score == 3 )
        cout << "Good" << endl;
    else
        if ( score == 2 )
            cout << "Average" << endl;
        else
{
    cout << "Below Average" << endl;
    cout << "Needs Improvement";
}
```

```
switch( score )
{
    case 4 :cout << "Excellent" << endl;
               break;
    case 3 :cout << "Good" << endl;
               break;
    case 2 :cout << "Average" << endl;
               break;
    default :
               cout << "Below Average" << endl;
               cout << "Needs Improvement";
}
```

Exercise 03

- Write a C++ program to input the total price to be paid by a customer and calculate the discount according to the chart below.

Total Price	Discount Rate
> 10000	25%
10000 - 5000	15%
5000 - 3000	10%

Try this in **repl.it** using your account. Copy your solution's url to chat window

Iteration Control Structure

- while
- do – while

- for

```
int count=1;
while ( count <=10 )
{
    cout<<count<<endl;
    count++;
}
```

```
int count=1;
do
{
    cout<<count<<endl;
    count++;
} while ( count <=10 );
```

```
for(int count=1; count <=10 ; count++)
    cout<<count<<endl;
```

Exercise 04

- Display number 1000,900,800,700,... 100

Using a while loop, do while loop and a for loop

- within the same program.

- i.e. Display these number series three times. One for each repetition structure.

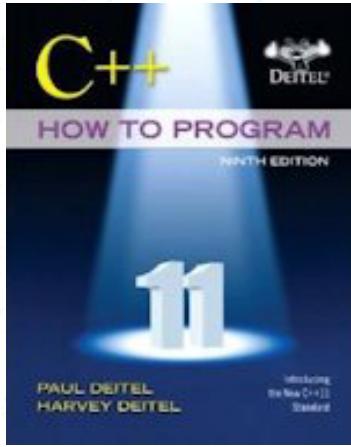
Try this in **repl.it** using your account. Copy your solution's url to chat window

Exercise 05

- Consider Exercise 03
- Modify the program to input details of 3 customers and calculate the total discount amount given.
- What would you do if you want to continue entering prices until -1 is entered ?
- What would you do if you want to enter data until user enters 'y' to continue and 'n' to stop?

Try this in **repl.it** using your account. Copy your solution's url to chat window

Reference



Chapter 01 & 02

Deitel & Deitel's (2016), C++ How to Program,
9th Edition





SLIIT

Discover Your Future

CSIT1050- Object Oriented Concepts

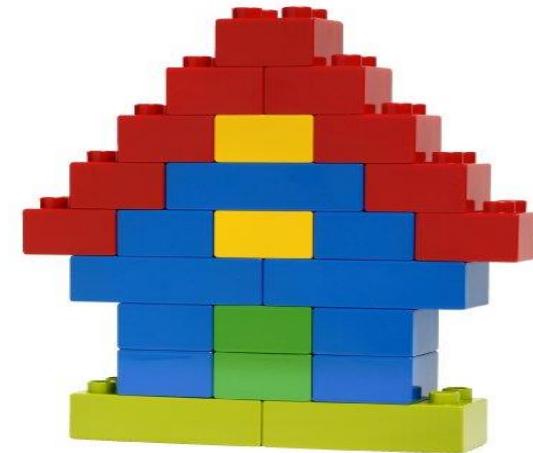
Lecture 03 – Object Oriented Concepts

Learning Outcomes

- At the end of the Lecture students should be able to
 - Understand Abstraction
 - Understand, describe and identify Objects and Classes
- • •
- • •
- • •
- • •
- • •
- • •
- • •

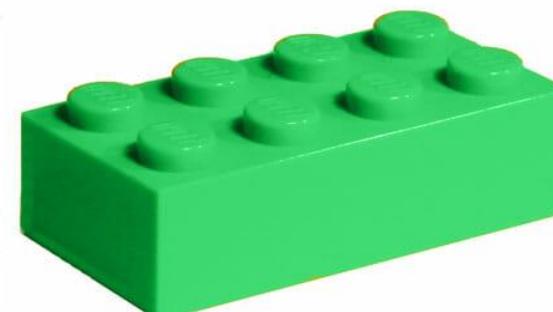
Object Oriented Programming

- Object Oriented Programming is a method of implementation in which programs are organized as a collection of objects which cooperate to solve a problem.
- Allows to solve more complex problems easily.

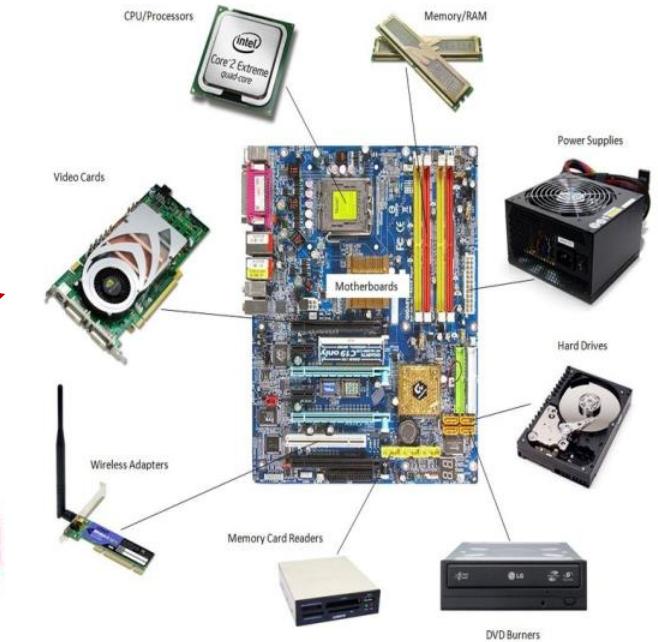
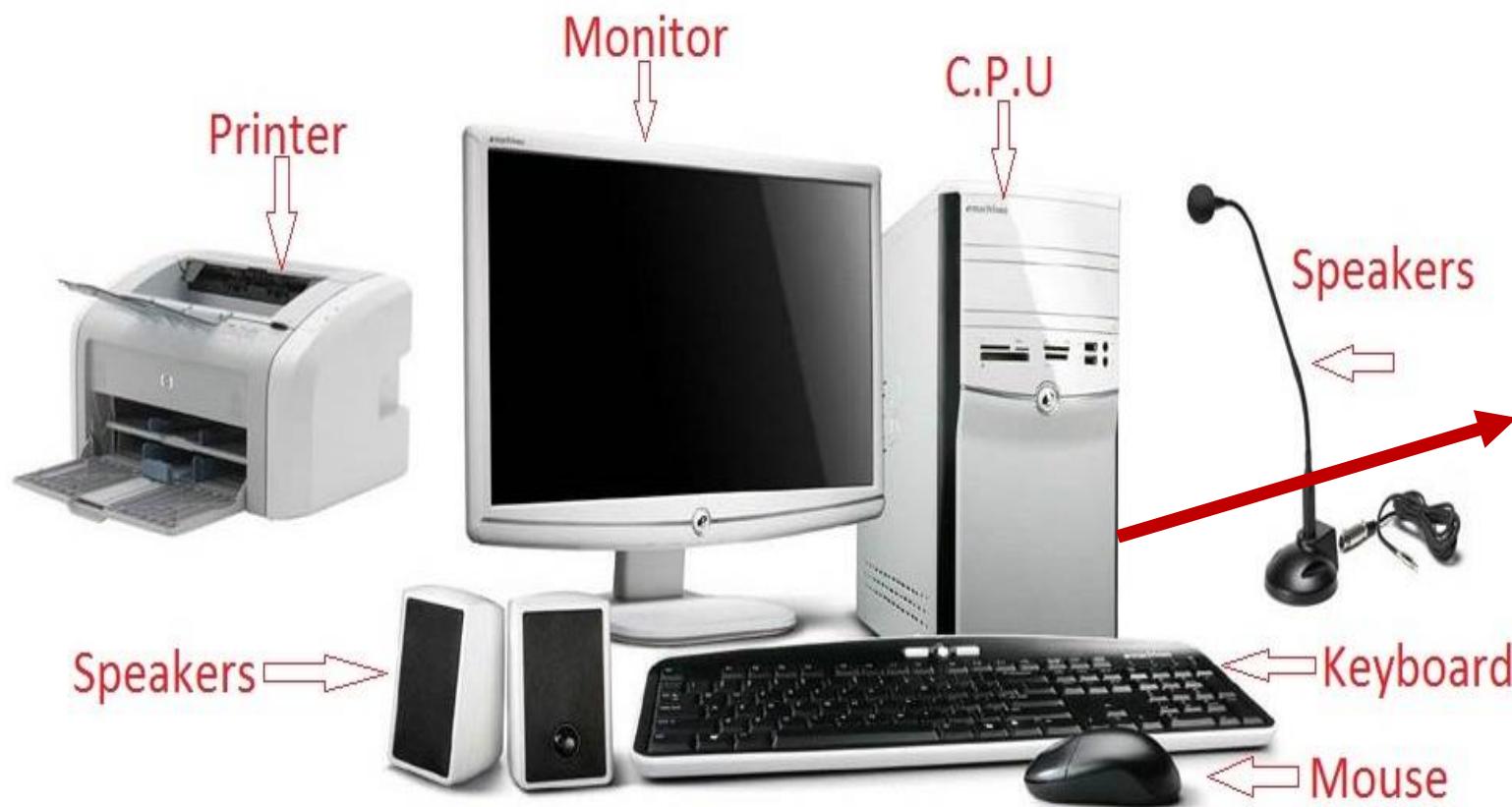


Object Oriented Programming

- A complex system is developed using smaller sub systems.
- Sub systems are independent units containing their own data and functions.
- Can reuse these independent units to solve many different problems.



A Computer System



Basic parts of a Computer

Object – General Meaning

object

noun

/'ɒbdʒɛkt, 'ɒbdʒɪkt/ 

1. a material thing that can be seen and touched.

"he was dragging a large object"

synonyms: thing, article, item, piece, device, gadget, entity, body

Oxford Dictionary

Objects in the Real World



(c) 2017 Monique Snoeck, KU LEUVEN

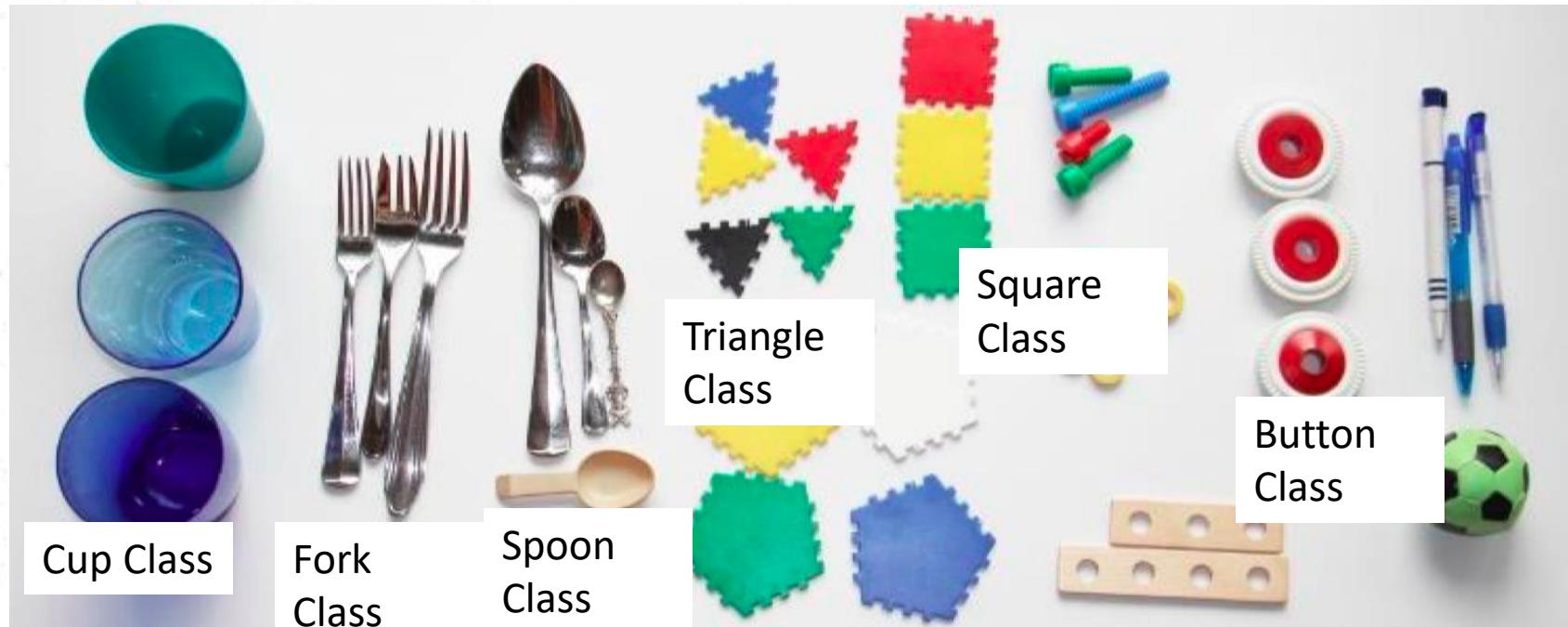
Grouping related things together



(c) 2017 Monique Snoeck, KU LEUVEN

Classes

We can classify objects into concepts. To do this we focus on the essential properties of an Object.
Classes are Concepts.



(c) 2017 Monique Snoeck, KU LEUVEN

Abstraction

- Distinguish between different Objects
- Classify Objects into Concepts
- Focus on the common properties



(c) 2017 Monique Snoeck, KU LEUVEN

Abstraction

- Distinguish between different Objects
- Classify Objects into Concepts
- Focus on the common properties



Cat Class



Dog Class

(c) 2017 Monique Snoeck, KU LEUVEN

Activity 1 – Identify Objects and Classes

- Dr. Pradeepa
- IWT
- Dushantha
- OOC
- Prof. Chandimal
- Lalani
- SPM
- Dr. Malitha
- Theja

Activity 1 – Identify Objects, Classes

Lecturer	Subject	Student
Dr. Pradeepa	IWT	Dushantha
Prof. Chandimal	OOC	Lalani
Dr. Malitha	SPM	Theja

Properties

- A class has a set of properties (attributes).
 - i.e. What do we need to store to describe a student?
- Activity - 2
 - What are the properties of a Student?
 - i.e.
 - Name
 - Age
 - ...
 - ..



Activity - 3

- Payroll system

Class: Employee

What are the properties needed ?

Employee number	Marital status	Age	Designation
OT Hours	Basic Salary	Height	Loan Installment
OT Rate	weight	Bonus	Name
Address	Hobbies	Insurance payment	Allowance
			Number of children
			Favourite Movie



Activity 3

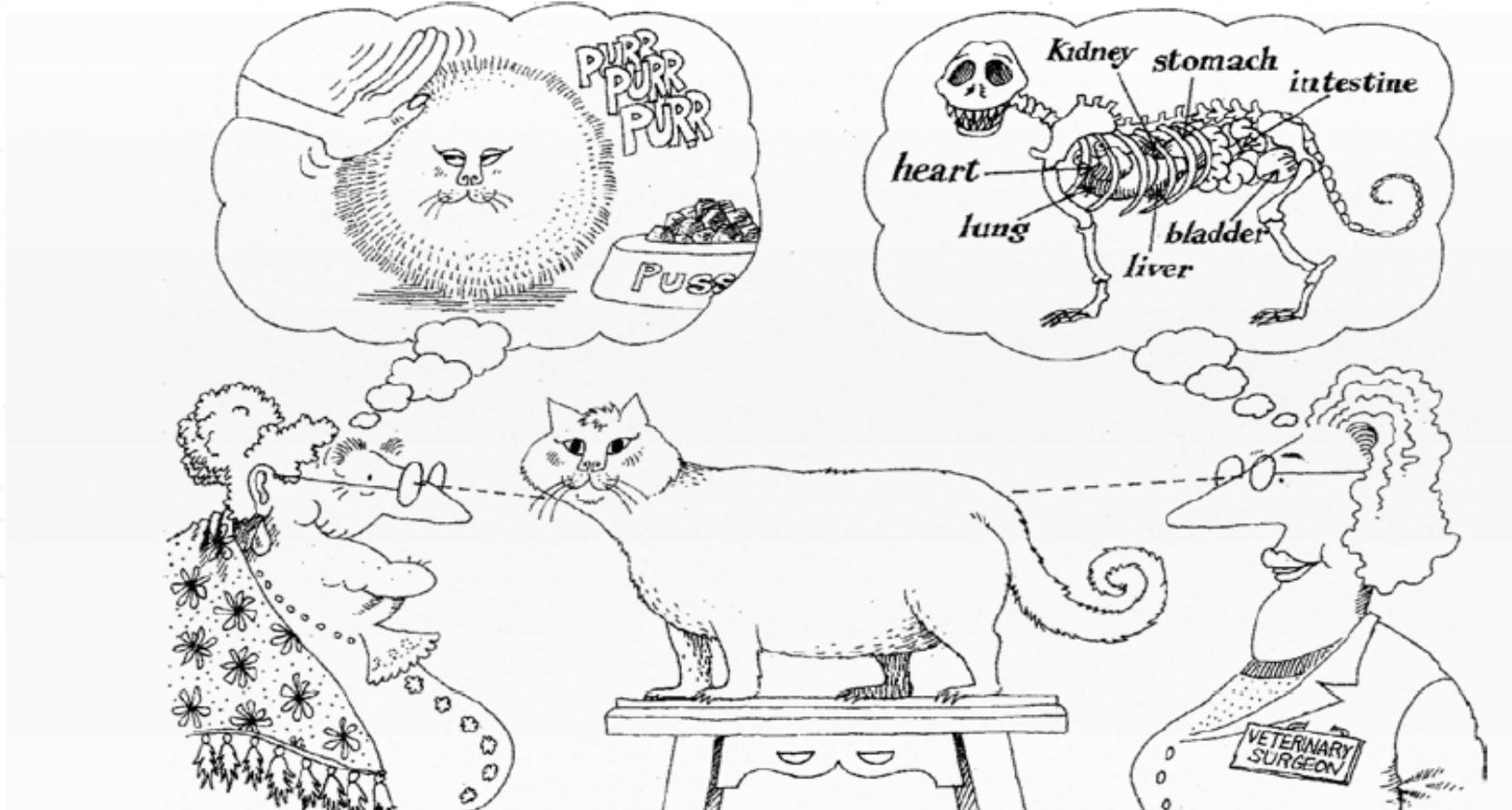
- What are the necessary properties of an employee for a Payroll system of Company ?
- What are the necessary properties of an employee for a Insurance scheme system of Company ?

Payroll System	Insurance scheme information system
Employee Number	Employee Number
Name	Name
Designation	Age
Basic Salary	Basic Salary
Allowance	Height
Bonus	Weight
OT Hours	Marital Status
OT Rate	Number of Children
Loan Installment	
Insurance Payment	

Abstraction

- Abstraction is the process of removing characteristics from ‘something’ in order to reduce it to a set of essential characteristics that is needed for the particular system.
- • •
- • •
- • •
- • •
- • •

Abstraction



Abstraction focuses on the essential characteristics of some object, relative to the perspective of the viewer.

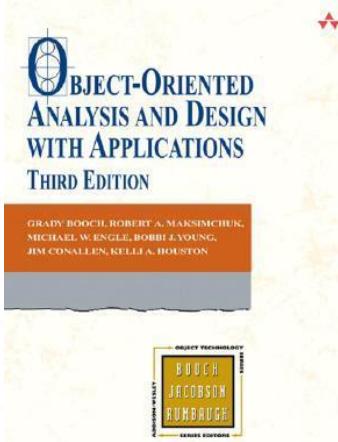
Abstraction

- An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to perspective of the viewer.
- • •
- • •
- • •
- • •
- • •

(Reference : Grady Booch, eta (2008), Object Oriented Analysis and Design with Applications 3rd Edition, pg 44)

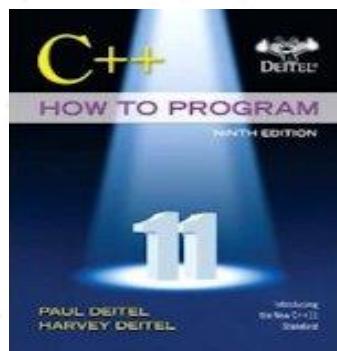


Reference



Chapter 01 & 02

Grady Booch (2008), Object-Oriented Analysis and Design with Application,
3rd Edition



Chapter 03

Deitel & Deitel's (2016), C++ How to Program,
9th Edition



SLIIT

Discover Your Future

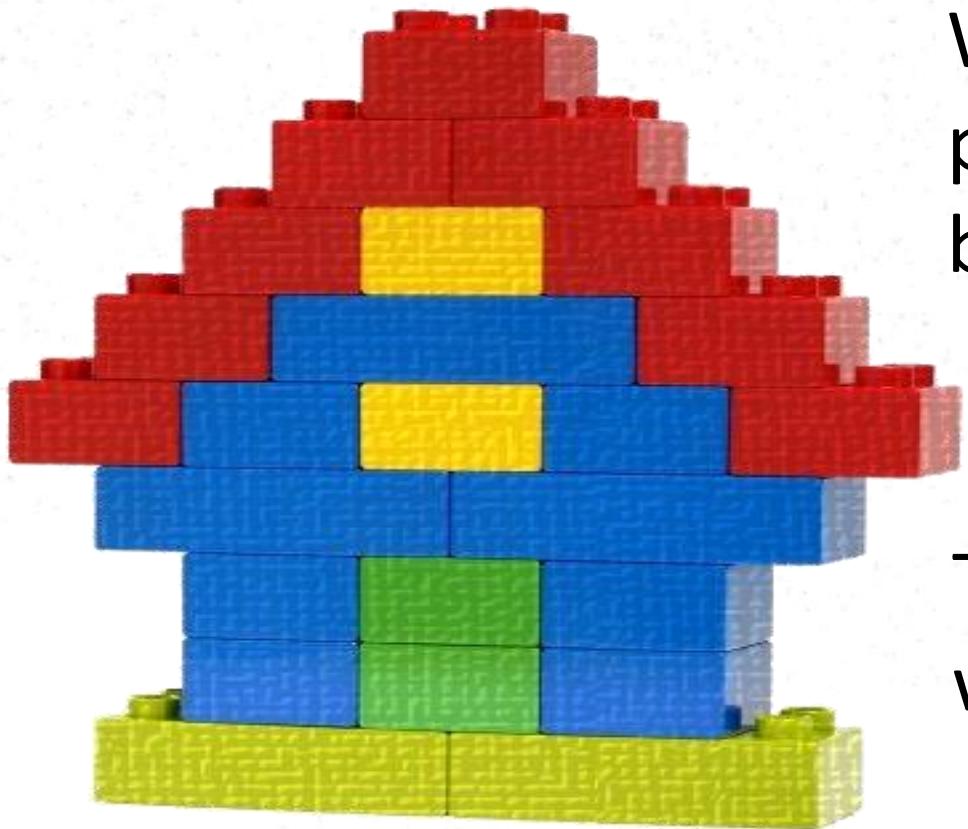
Object Oriented Concepts

Lecture-04
Classes & Objects – Part 2

Learning Outcomes

- At the end of the Lecture students should be able to
 - Understand, identify and describe Classes, Objects, Properties and Methods
 - Describe Encapsulation, Information Hiding, and Interfaces

How do we develop an OO Program ?



We look at the problem that needs to be solved

This is the building we want to build

e.g. In a real world scenario this could be a Student Information System (SIS)

• • •

Identifying Objects needed



These are the Blocks (Objects) that we need.

e.g. In SIS objects could be details of OOC, IWT, students called Manoj, Gayani

• • •

How do you create these Blocks (Objects)?



What if we needed to manufacture these blocks. How could we do this? What do we need to make first?

...
...

A Mould (Class)



Once we make a Mould (Class) we can make as many blocks (Object) that we need.

How to group these Blocks (Objects)?



How can we group these Blocks?

...

We could do it by Shape



A Square Mould (class)



A Rectangle Mould
(class)

e.g. In SIS it could be Student Class, Subject Class

Creating Blocks (Objects) from Moulds (Classes)



A Square Mould (class)

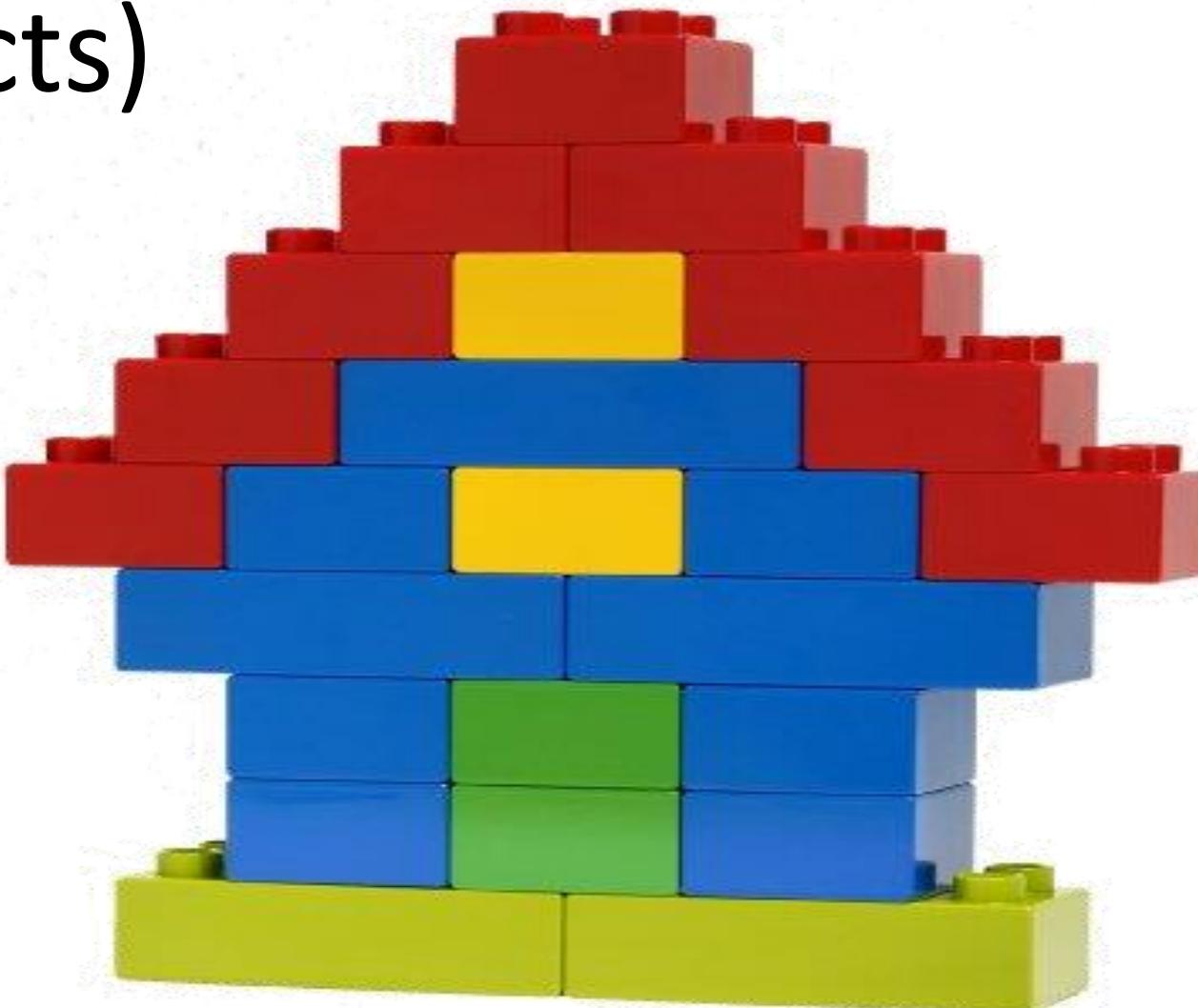


A Rectangle Mould (class)

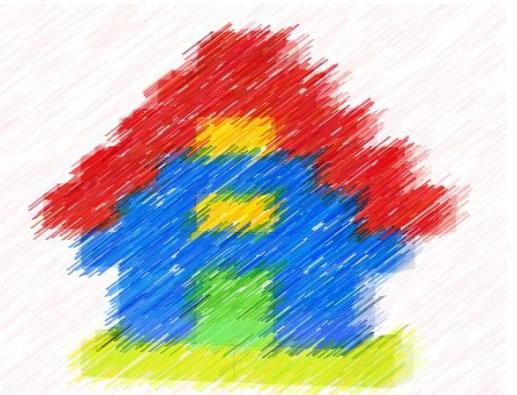


Blocks (Objects) made

Final Product – Assembling Blocks (Objects)



We can now assemble the Objects and create our final solution:



Problem to Solve



Identify Objects that are needed



Identify Classes through Abstraction



Assemble Objects to create the
solution



Create Objects from
Classes

I am dreaming of

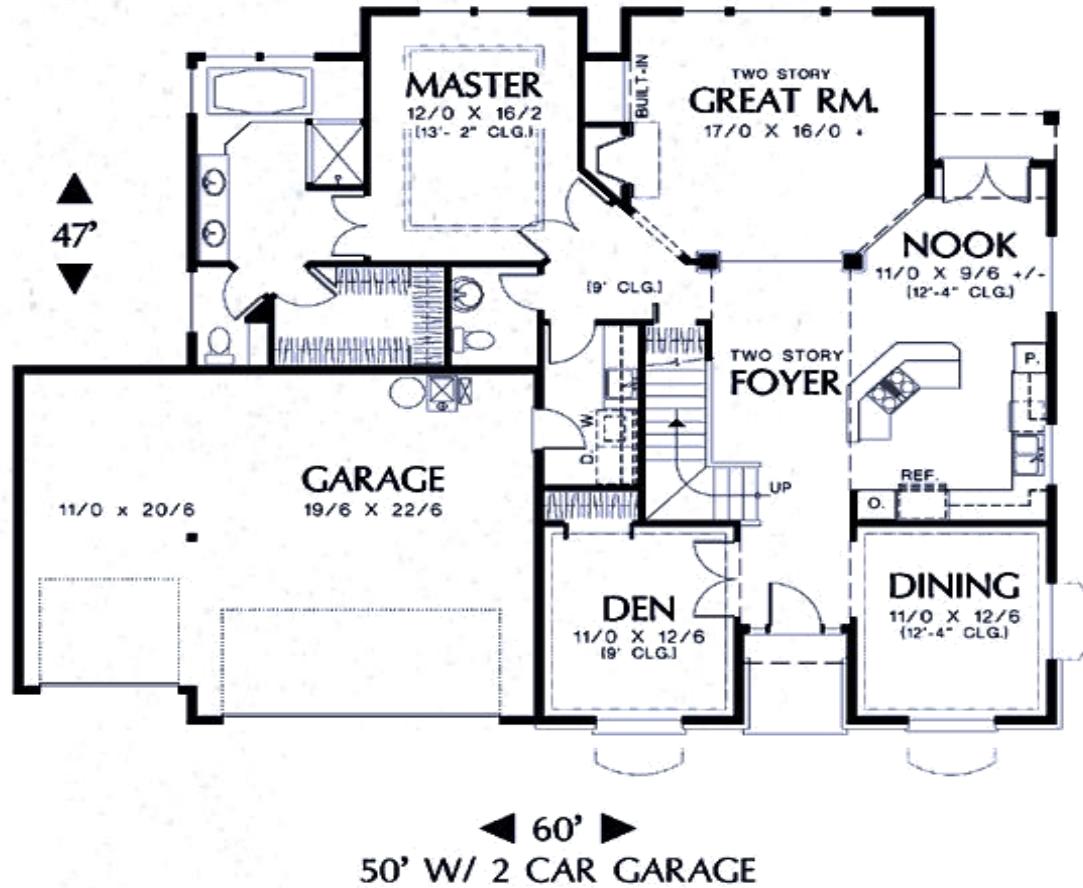


How do you build a house?



Meeting an Architect to make a House Plan (Blue Print)

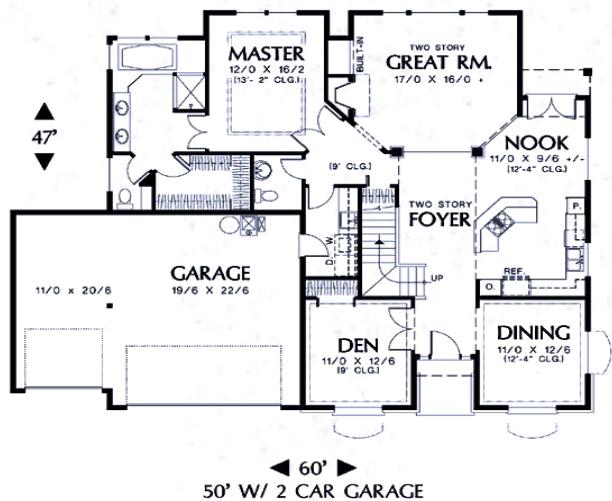
Blue Print – House Plan (Class)



Your Dream
House

Your Dream House

- With the House Plan – Blue Print (Class) you can now get a contractor to build your dream house (Object)



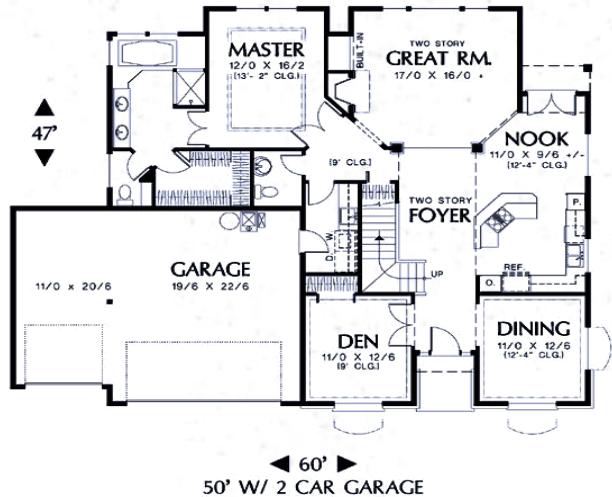
Class House



Object

Classes and Objects

- An Object is a specific instance (variable) of the data type (class)
- A class is a blue print of an object.



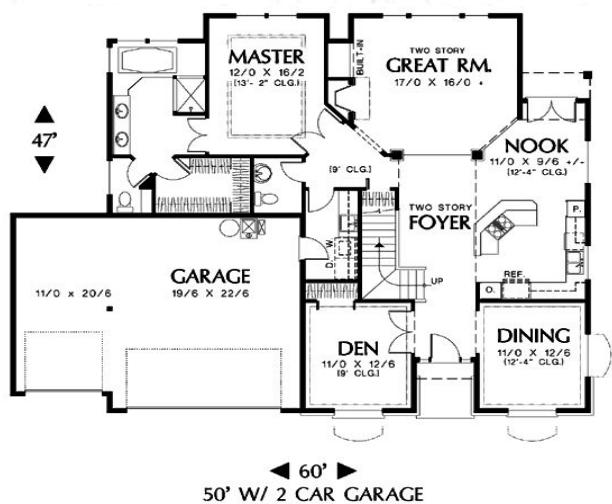
Class House



Object

Classes and Objects

- You can make as many houses as you want from a single house plan – Blue Print (Class)



Class House



House1

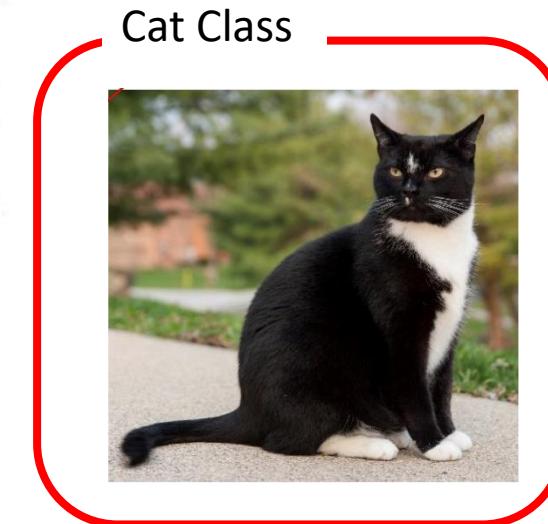
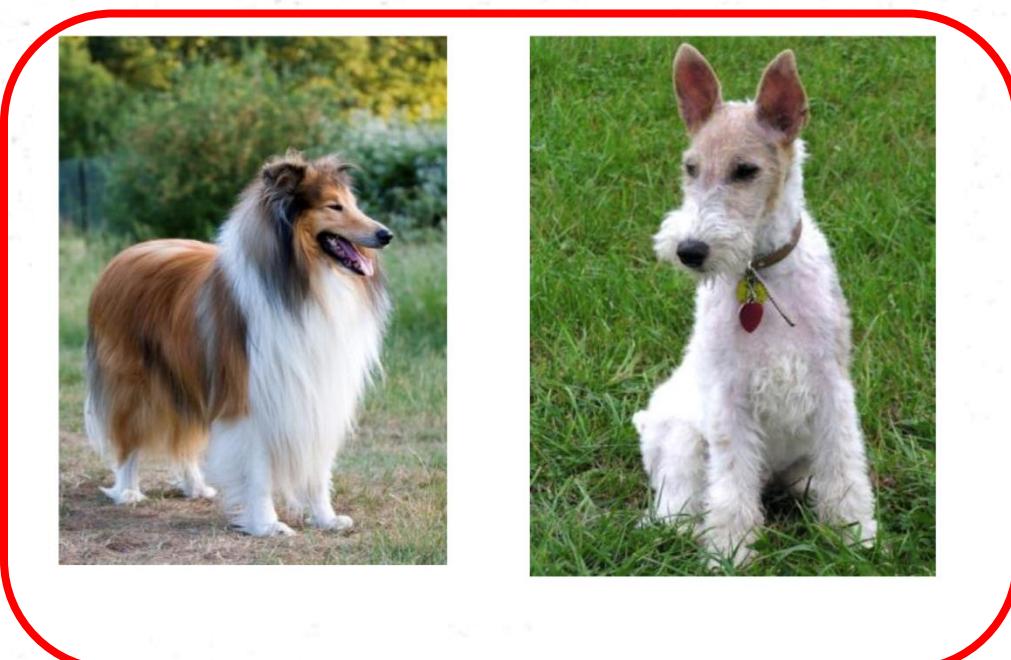
House2



Objects

Classes – Captures Behavior as well

- A concept (class) has both properties and behavior.
- We know that dogs and cats behave differently



(c) 2017 Monique Snoeck, KU LEUVEN

Example – Behaviour is captured as functions

Dog Class

name
owner
breed
work (*e.g. police dog*)

Bark()
Fetch()

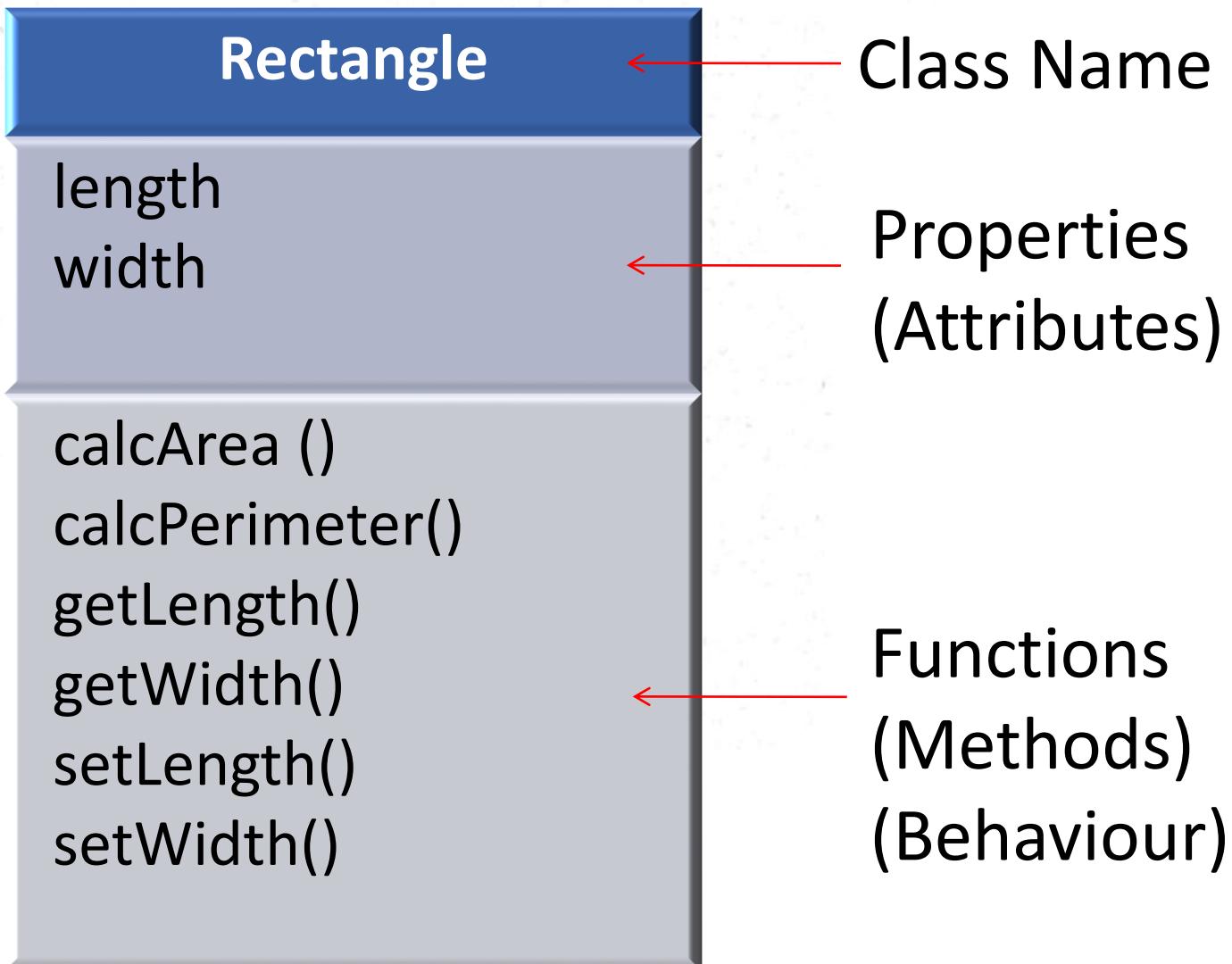
Cat Class

name
owner
breed

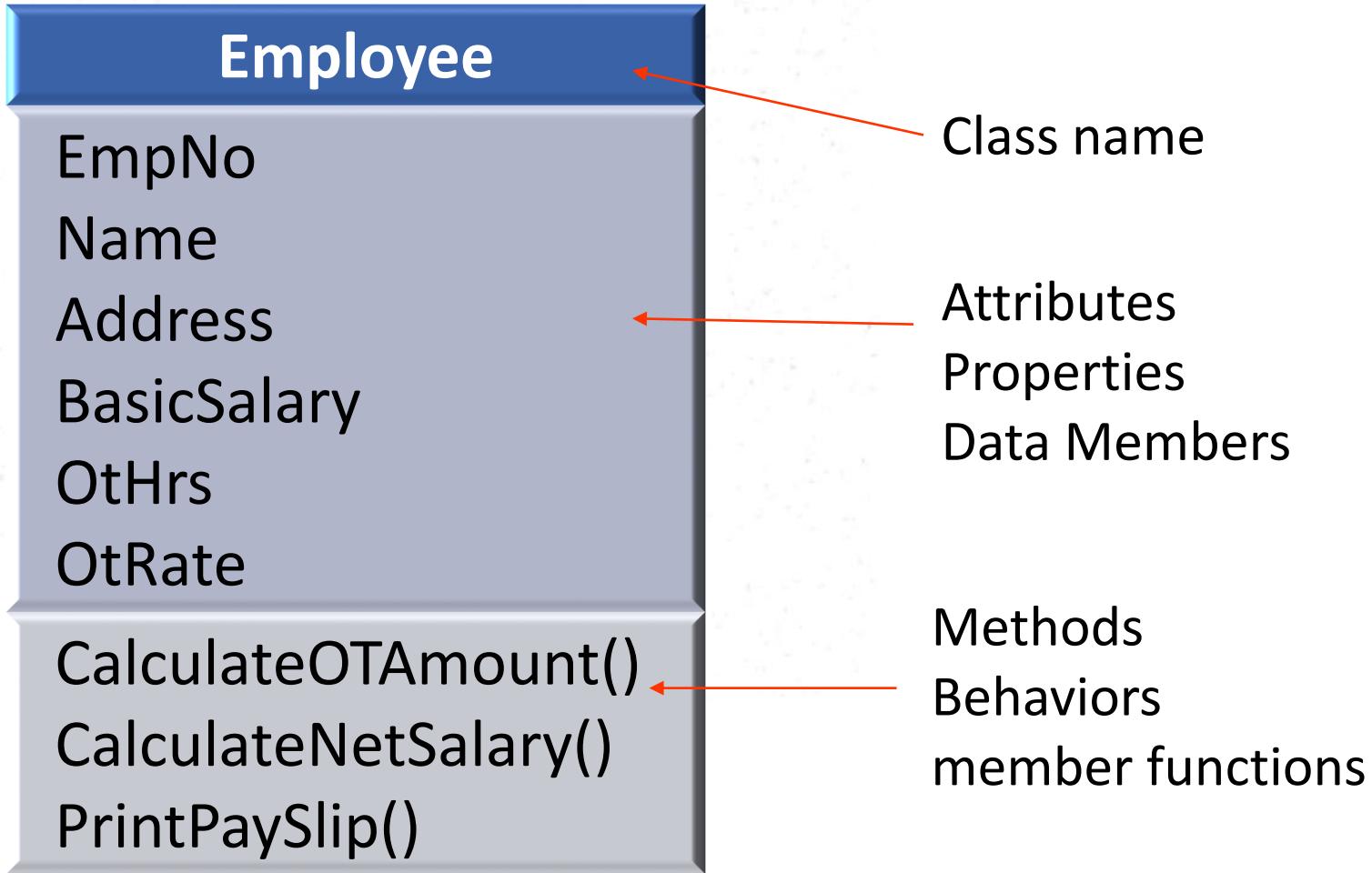
Meow()
Purr()

Grouping properties and functions together is called Encapsulation

Rectangle Class



Employee Class



Restricted Access

- All properties and some functions of a Class have restricted access (private) and can be accessed only through public functions.
- Why is this necessary?
- Let's look at an example.

A Jewelry Shop





Jewelry can be accessed only through a Sales Person

JewelleryShop Class



Private
(Restricted)

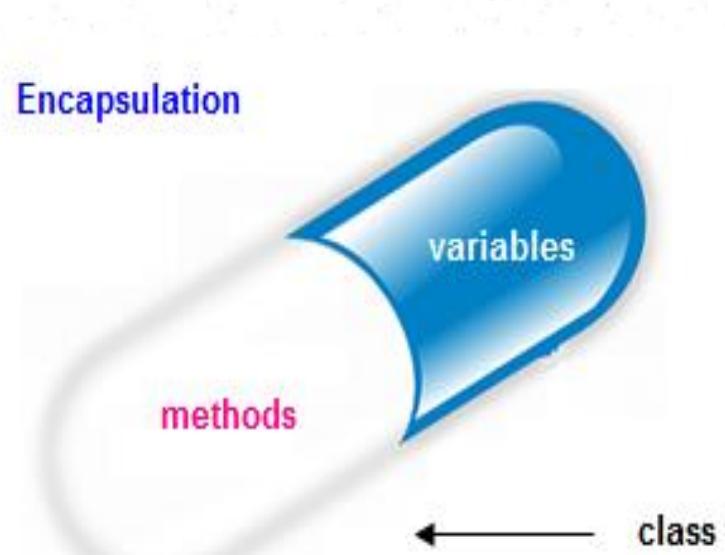
Public
(Salesman
do this)

Information Hiding

- Hide certain information or implementation decision that are internal to the encapsulation structure (class)
- The only way to access an object is through its public interface (public functions)
 - Public – anyone can access / see it
 - Private – no one except the class can see/ use it

Encapsulation

- It is the process of grouping related attributes and methods together, giving a name to the unit and providing an interface (public functions) for outsiders to communicate with the unit.

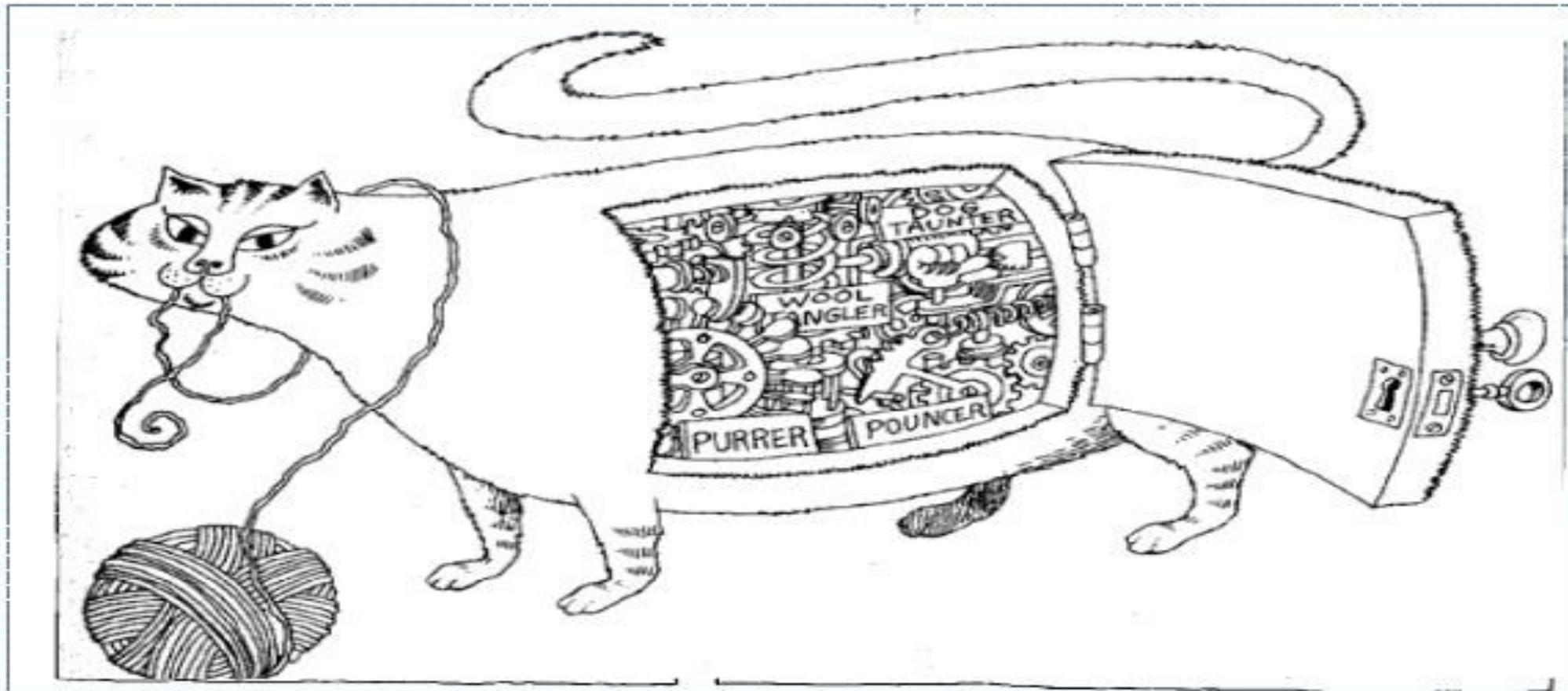


Encapsulation

- The implementation of the TV is hidden from us. Your TV could be OLED, LCD, Plasma or an old CRT one.
- You can control any such TV using the same commands in your remote (interface).



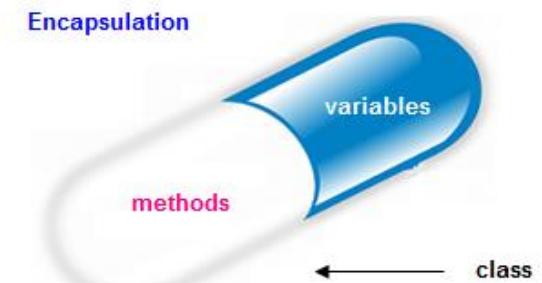
Encapsulation hides the details of the implementation of an object



(Reference : Grady Booch, eta (2008), Object Oriented Analysis and Design with Applications 3rd Edition, pg 52)

Encapsulation

- Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation.

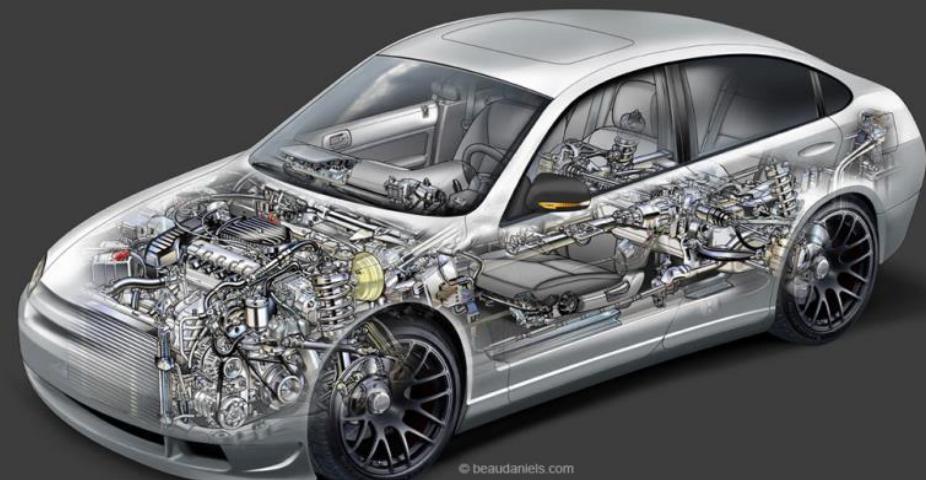


- (Reference : Grady Booch, eta (2008), Object Oriented Analysis and Design with Applications 3rd Edition, pg 52)

Interface – Public Functions

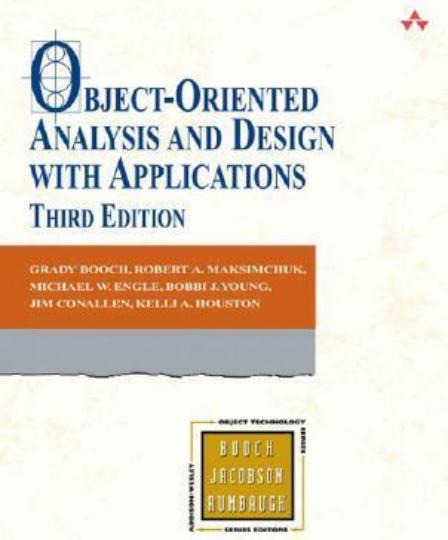


Full Vehicle cutaway



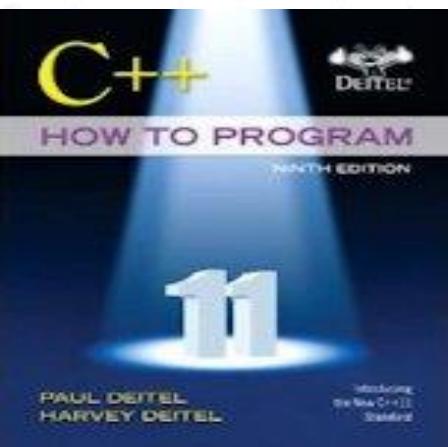
You interact with some object through its interfaces.
Your car can be Gasoline, Hybrid or Electric but you drive it the same way.

Reference



Chapter 03

Grady Booch (2008), Object-Oriented Analysis and Design with Application,
3rd Edition



Chapter 09

Deitel & Deitel's (2016), C++ How to Program,
9th Edition



SLIIT

Discover Your Future

CSIT1050- Object Orientation

Lecture-05
Classes and Objects in C++

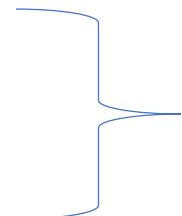
Learning Outcomes

At the end of the Lecture students should be able to;

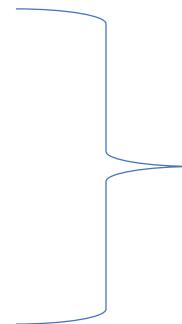
- Implement a class and create objects using C++.

Recalling Steps in OOP

- Analyse the Problem
- Identify Objects
- Develop Classes
- Create Objects
- Build the Solution



OO Analysis and
Design

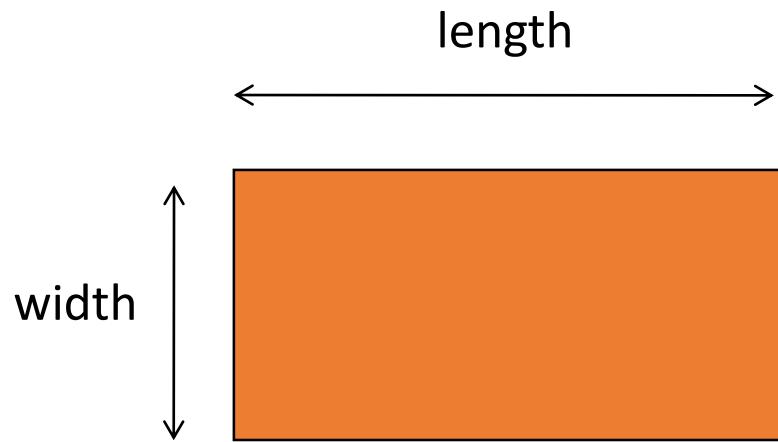
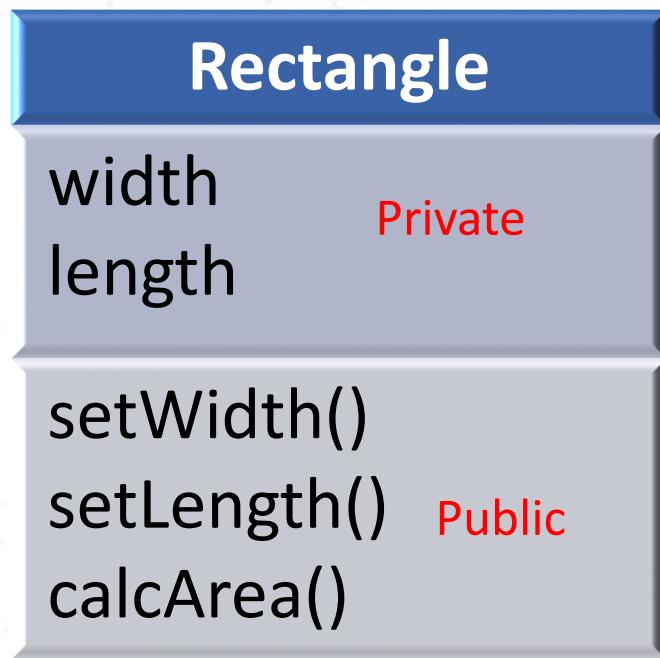


OO Programming

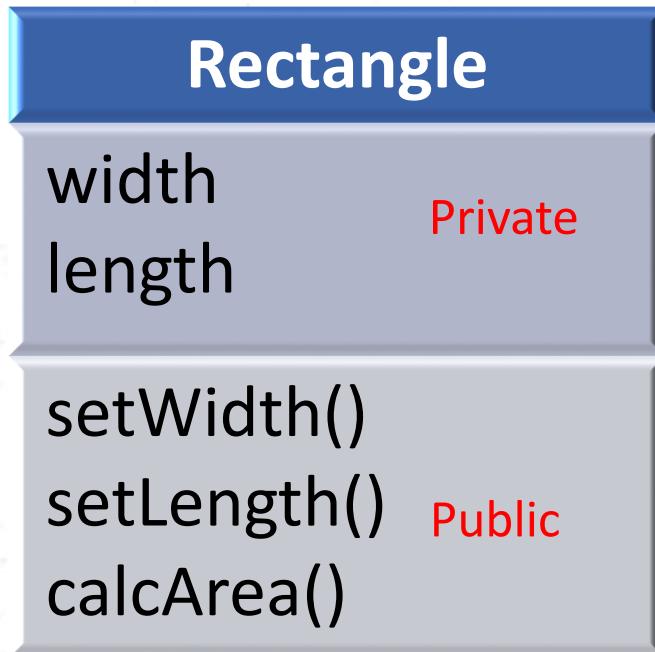
How to write an Object Oriented Program ?



Example-1 - Rectangle Class



Rectangle Class in C++

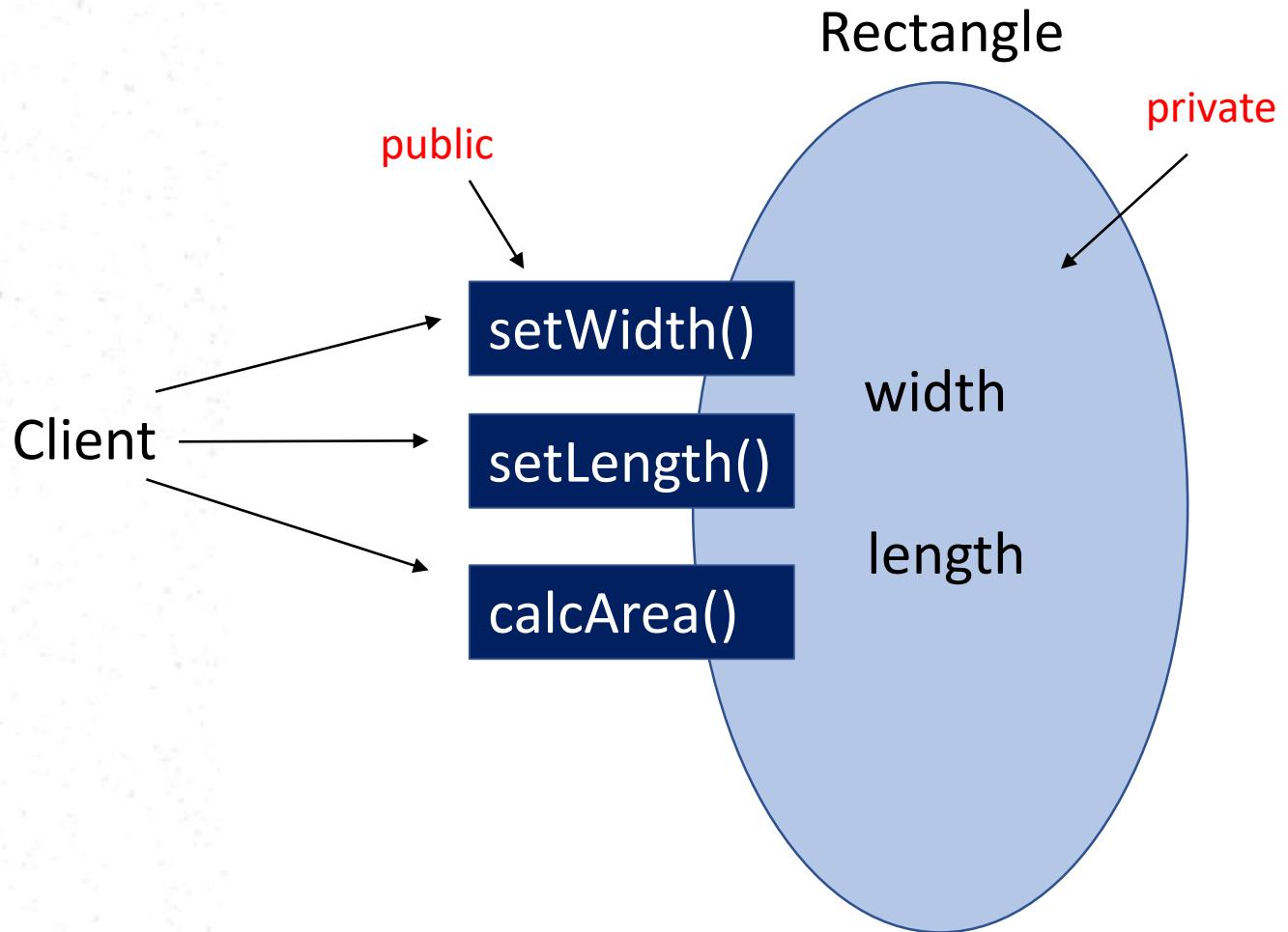


```
//C++ coding for Rectangle class
class Rectangle {
    private:
        int width;
        int length;
    public:
        void setWidth(int no);
        void setLength(int no);
        int calcArea();
};
```

Private & Public

- The **private** part of the definition specifies the properties (data members) of a class.
- These are hidden from outside the class and can only be accessed through the methods (operations/functions) defined for the class.
- The **public** part of the definition specifies the methods as function prototypes.
- These methods as they are called, can be accessed by the main

Private & Public



Rectangle Class in C++

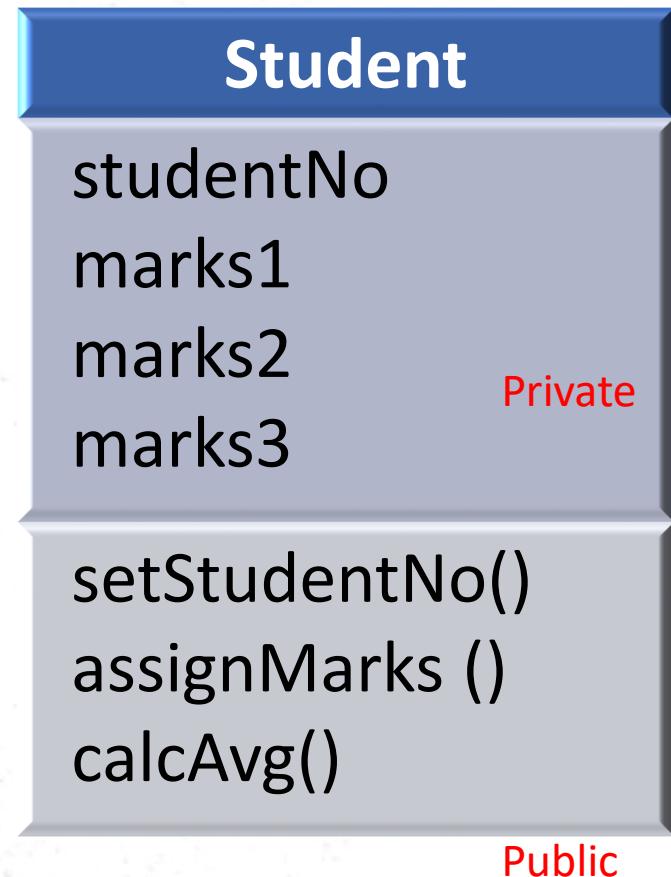
```
//C++ coding for Rectangle
class Rectangle {
    private :
        int width;
        int length;
    public:
        void setWidth(int no);
        void setLength(int no);
        int calcArea();
};
```

```
void Rectangle::setWidth(int no) {
    width = no;
}

void Rectangle::setLength(int no) {
    length= no;
}

int Rectangle::calcArea() {
    int area = length * width;
    return area;
}
```

Student class – Activity 1



Implement the Student class in C++.

Answer : Student Class in C++

```
//C++ coding for Student
class Student {
    private :
        int StudentNo;
        int marks1;
        int marks2;
        int marks3;
    public:
        void setStudentNo(int no);
        void assignMarks(int n1, int n2, int n3);
        float calcAvg();
};
```

```
void Student::setStudentNo(int no) {  
    width = no;  
}  
  
void Student :: assignMarks(int n1, int n2, int n3);{  
    marks1 = n1;  
    marks2 = n2;  
    marks3 = n3;  
}  
float Student ::calcAvg() {  
    float average = (marks1+marks2+marks3)/3.0;  
    return average;  
}
```

Creating Objects

Class_name Object_name;

e.g: Rectangle rect1; // single object

Rectangle rect1, rect2; // multiple objects

Rectangle rectangles[5]; // array of objects

Note : Use C++ rules for identifiers when naming objects

Creating Objects

```
Rectangle rec1, rec2;
```

rec1 : Rectangle

width = 10
length = 20

rec2 : Rectangle

width = 5
length = 10

Accessing Public Methods

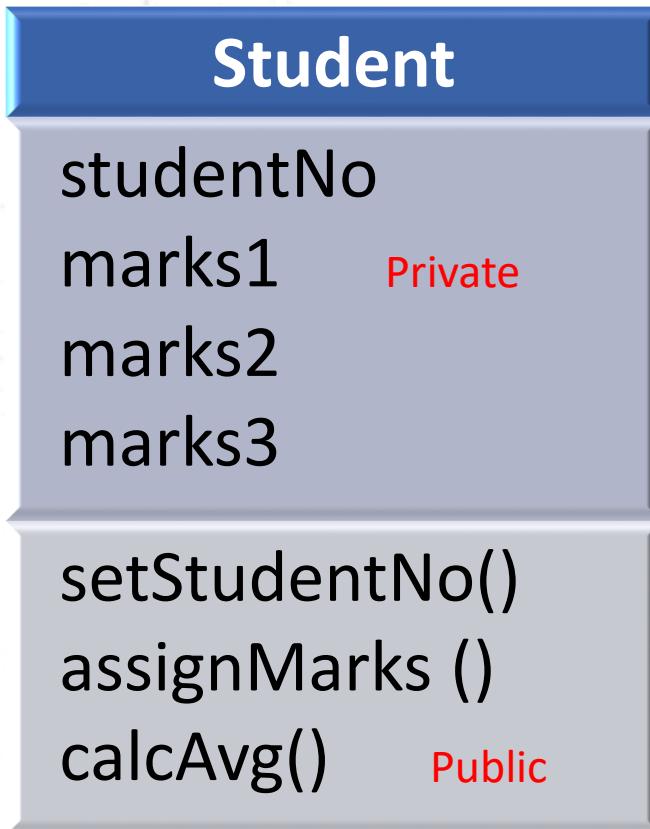
```
#include <iostream>
using namespace std;
int main() {
    Rectangle rec1, rec2;

    rec1.setWidth(10);
    rec1.setLength(20);

    rec2.setWidth(5);
    rec2.setLength(10);

    cout << rec1.calcArea() << endl;
    cout << rec2.calcArea() << endl;
    return 0;
}
```

Student class – Activity 2



Create two Student Objects.
Calculate and print their averages.

Student Objects

std1: Student

studentNo = 1023
marks1= 50
marks2= 60
marks3 = 70

std2: Student

studentNo= 2345
marks1= 70
marks2= 80
marks3 = 75

```
#include <iostream>
using namespace std;
int main() {
    Student std1, std2;

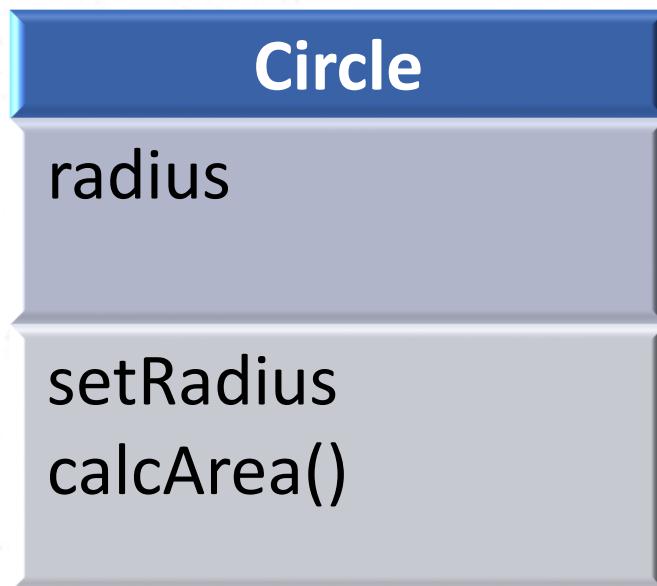
    std1.setStudentNo(1023);
    std1.assignMarks(50,60,70);

    std2.setStudentNo(2345);
    std2.assignMarks(70,80,75);

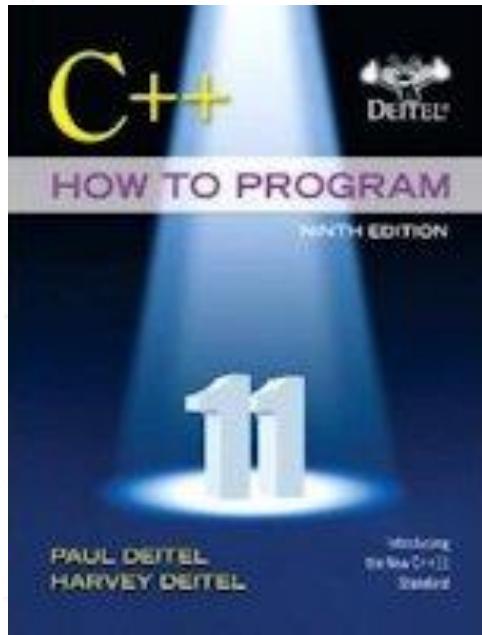
    cout <<“Average of student1:”<< std1.calcAvg() << endl;
    cout <<“Average of student2:”<< std2.calcAvg() << endl;
    return 0;
}
```

Activity 3

Implement the Circle class and write a client (main) program to calculate and print the area of a circle.



Reference



Chapter 03

Deitel & Deitel's (2016), C++ How to Program,
9th Edition

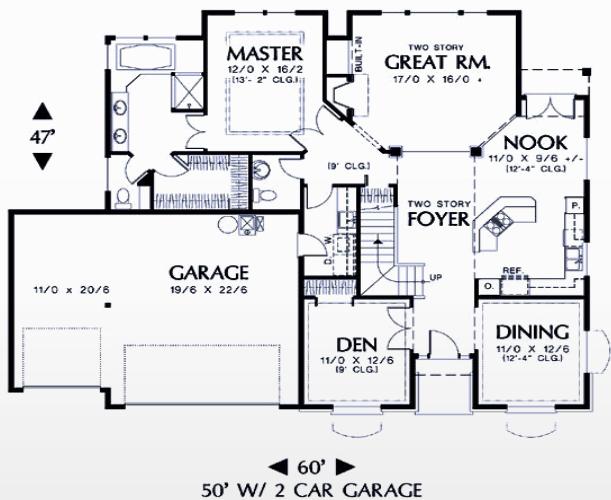
Object Oriented Concepts

Lecture 06
Classes in C++

Learning Outcomes

- At the end of the Lecture students should be able to
 - Have a better understanding of the differences between classes and objects (in C++ coding)
 - Use setters and getters in a Class
 - Write Object Oriented Programs
 - Use header files with classes

Classes and Objects



Class House
Blue Print of a House

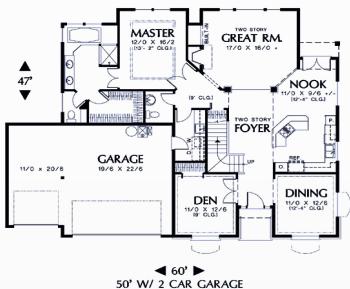
```
House
- length
- width
- height
- area
+ paint()
```

```
class House {
    private:
        int length;
        int width;
        int height;
        int area;
    ...
public:
    void paint();
    ...
}
```

Classes and Objects

```
class House {  
    private:  
        int length;  
        int width;  
        int height  
        int area;  
        ...  
    public:  
        void paint();  
        ...  
}
```

Class



Objects



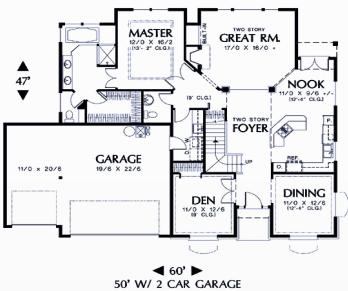
myHouse1

```
int main() {  
    House myHouse1;  
}  
4
```

Classes and Objects

```
class House {  
    private:  
        int length;  
        int width;  
        int height  
        int area;  
        ...  
    public:  
        void paint();  
        ...  
}
```

Class



Objects



myHouse1



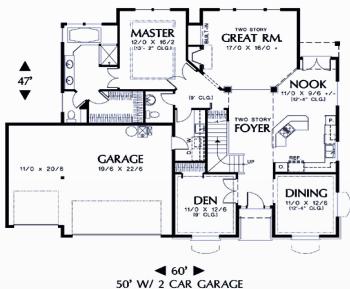
myHouse2

```
int main() {  
    House myHouse1;  
    House myHouse2;  
}  
}
```

Classes and Objects

```
class House {  
    private:  
        int length;  
        int width;  
        int height;  
        int area;  
        ...  
    public:  
        void paint();  
        ...  
}
```

Class



Objects



myHouse1



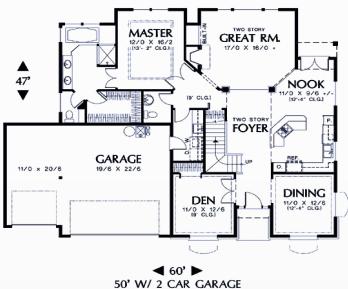
myHouse2

```
int main() {  
    House myHouse1;  
    House myHouse2;  
    myHouse1.paint(green);  
}  
}
```

Classes and Objects

```
class House {  
    private:  
        int length;  
        int width;  
        int height;  
        int area;  
        ...  
    public:  
        void paint();  
        ...  
}
```

Class



Objects



myHouse1



myHouse2

```
int main() {  
    House myHouse1;  
    House myHouse2;  
    myHouse1.paint(green);  
    myHouse2.paint(blue);  
}
```

Classes and Objects

```
class House {  
    private:  
        int length;  
        int width;  
        int height;  
        int area;  
        ...  
    public:  
        void paint();  
        ...  
}
```

Class



These attributes are protected
We can't access these
in the main function()

We interact with objects using
The public methods.

Objects



myHouse1

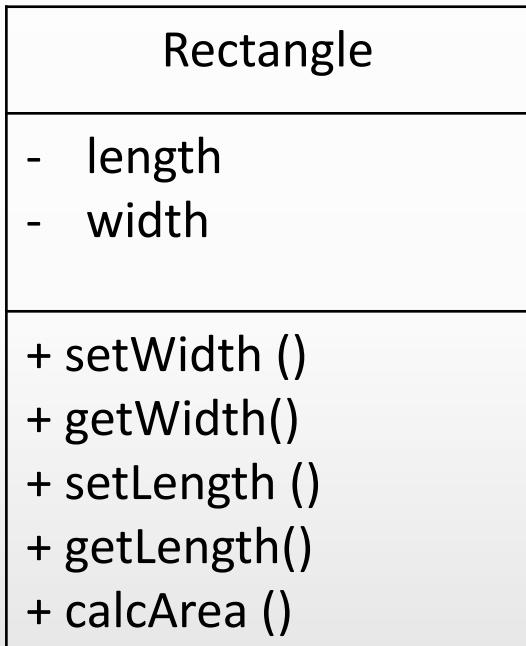


myHouse2

```
int main() {  
    House myHouse1;  
    House myHouse2;  
    myHouse1.paint(green);  
    myHouse2.paint(blue);
```

} We use the dot Operator to
access methods.

Rectangle Class



```
class Rectangle {  
    private:  
        int width;  
        int length;  
    public:  
        void setWidth(int w);  
        int getWidth();  
        void setLength(int l);  
        int getLength();  
        int calcArea();  
};
```

The diagram on the left side is a UML (Unified Modeling Language) Class Diagram. Here – means private and + means public

Rectangle Class

Rectangle
- length - width
+ setWidth () + getWidth() + setLength () + getLength() + calcArea ()

```
class Rectangle {  
    private:  
        int width;  
        int length;  
    public:  
        void setWidth(int w); ← A Setter  
        int getWidth(); ← A Getter  
        void setLength(int l);  
        int getLength();  
        int calcArea();  
};
```

Since the properties length and width are protected and cannot be accessed from the main function we usually write two methods per property. A set method (setters) and a get method (getters).

Setters (Mutators)

```
class Rectangle {  
    private:  
        int width;  
        int length;  
    public:  
        void setWidth(int w);  
        int getWidth();  
        void setLength(int l);  
        int getLength();  
        int calcArea();  
};
```

We can do validations

In a setter

Here we are assuming that the default width is 10.

We know a Rectangle's width cannot be zero or negative.

If someone sets a negative width
The Rectangle width will be set to 10.

// A Setter starts with the word set
// followed by the name of the property
// e.g. setWidth()
// setters are always methods that don't
// return values (void functions)

```
void Rectangle::setWidth(int w) {  
    if (w > 0)  
        width = w;  
    else  
        width = 10;  
}
```

Getters (Accessors)

```
class Rectangle {  
    private:  
        int width;  
        int length;  
    public:  
        void setWidth(int w);  
        int getWidth();  
        void setLength(int l);  
        int getLength();  
        int calcArea();  
};
```

Getters always contain the
Following code.

return property;

// A Getter starts with the word get
// followed by the name of the proeprty
// e.g. getWidth()
// getters always have the return type of
// the property.

```
int Rectangle::getWidth(){  
    return width;  
}
```

Exercise - 1

StopWatch
- minute
- second
+ setMinute()
+ getMinute()
+ setSecond()
+ getSecond()
+ start()
+ stop()

Write a setter and getter for the property minute.

Exercise – 1 – Class definition (Not part of the answer)

StopWatch
- minute
- second

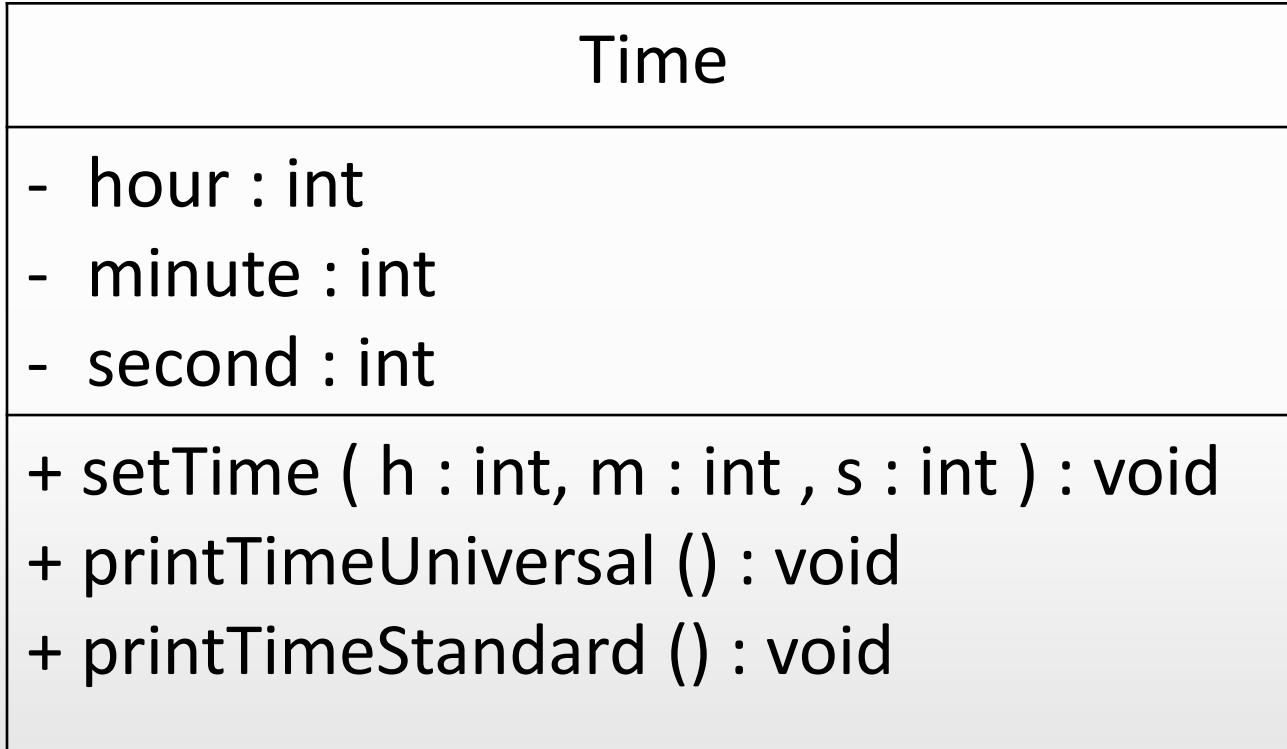
+ setMinute()
+ getMinute()
+ setSecond()
+ getSecond()
+ start()
+ stop()

```
class StopWatch {  
    private :  
        int minute;  
        int second;  
    public:  
        void setMinute(int min);  
        int getMinute();  
        void setSecond(int sec);  
        int getSecond();  
        void start();  
        void stop();  
};
```

Setter and getters for minute property

```
void StopWatch::setMinute(int min) {  
    minute= min;  
}  
// or with validations  
void StopWatch::setMinute(int min) {  
    if (min >= 0 && min <= 59)  
        minute = min;  
    else  
        minute = 0;  
}  
int StopWatch::getMinute() {  
    return minute;  
}
```

Implementing a Class.. Example..



We can represent datatypes and parameters in UML class diagrams as shown above. The datatype or return type is given after the property or method. A colon is used as a separator

Exercise - 2

Time

- hour : int
- minute : int
- second : int

- + setTime (h : int, m : int , s : int) : void
- + printTimeUniversal () : void
- + printTimeStandard () : void

Implement this class. Here the time is represented in a 24 hour clock Format. Universal time is a 24 hour clock. Standard time is a 12 hour Clock, we also need to print AM, PM. In this class for simplicity we have not included setters and getters.

Time class in C++

```
class Time {  
    private :  
        int hours;  
        int minute;  
        int second;  
    public:  
        void setTime(int h, int m, int s);  
        void printTimeUniversal();  
        void printTimeStandard();  
};
```

Implement methods

```
void Time::setTime(int h, int m, int s)
{
    hour = h;
    minute = m;
    second = s;
}

void Time::printTimeUniversal()
{
    cout<<setw(2)<<hour<<":"<<setw(2)
    <<minute<< ":"<<setw(2)<<second;
}
```

Setfill is used to have leading zeros

```
void Time:: printTimeStandard()
{
    cout<<setfill('0')<<setw(2);
    if( hour == 0 || hour == 12)
        cout<<12;
    else
        cout<<hour%12;

    cout<<":"<<setw(2)<<minute<<":"<<setw(2)<<second;
    if ( hour < 12)
        cout << " AM" <<endl;
    else
        cout << " PM" << endl;
}
```

Client Program – Exercise 2

OUTPUT :

Input Hour :13

Input Minutes :27

Input seconds :6

13:27:06

01:27:06 PM

Write a main program to input values for hours, minutes and seconds in a 24 hour clock format and print the time both in universal time and standard time

Client Program

OUTPUT :

Input Hour :13

Input Minutes :27

Input seconds :6

13:27:06

01:27:06 PM

```
int main()
{
    Time t; // static object
    int hou, min, sec;

    cout<<"Input Hour :";
    cin >> hou;
    cout<<"Input Minutes :";
    cin >> min;
    cout<<"Input seconds :";
    cin >> sec;

    t.setTime (hou, min, sec);
    t.printTimeUniversal ();
    t.printTimeStandard ();

}
```

Static Object

```
Time t;
```

Methods are accessed using dot (.) operator

```
t.setTime ( 13, 27, 6);
```

How Classes are implemented

- In C++ we generally separate each class implementation into two files.
- A Header file containing the class definitions. e.g. Time.h
- A .cpp file containing the implementation of the methods of the class e.g. Time.cpp
- The client program is the main program that is used to create objects of the classes we have previously implemented

Time.h

Time.cpp

client.cpp

Definition of the class

Implementation of the class

Main program

How Classes are implemented

- This approach allows us to reuse a class in many applications.
- This is a standard practice when writing C++ code.
- The header file only contain the definitions of the class, including the interfaces (public methods)

Time.h

Time.cpp

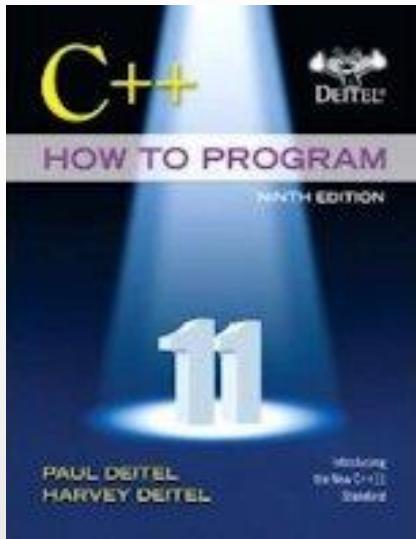
client.cpp

Definition of the
class

Implementation of
the class

Main program

Reference



Chapter 09

Deitel & Deitel's (2016), C++ How to Program,
9th Edition



SLIIT

Discover Your Future

IT1050 - Object Oriented Concepts

Lecture-07 Constructors and Destructors

Learning Outcomes

- At the end of the Lecture students should be able to
 - Use Overloading
 - Use constructors and destructors.
 - Use dynamic objects.
- • •
- • •
- • •
- • •
- • •
- • •
- • •

Overloading Functions

```
// Prototype functions  
void print();  
void print(char msg[]);  
void print(char msg[], int no);
```

- In C++, you can have multiple functions with the same name, but having different types of parameters.
- This is called function overloading.
- In overloading the parameters of each function should be different. The return type and the function name have to remain the same.

```
void print (dataType1, dataType2, dataType3);
```

Overloading Functions

```
// Prototype functions
void print();
void print(char msg[]);
void print(char msg[], int no);

// Prototype function implementation
void print() {
    cout << "Hello " << endl;
}
void print(char msg[]) {
    cout << "Hello " << msg << endl;
}
void print(char msg[], int no) {
    cout << msg << ":" << no << endl;
}
```

```
int main() {  
    print();  
    print("SLIIT");  
    print("Age ", 19);  
    return 0;  
}
```

Output

Hello
Hello SLIIT
Age : 19



Employee Class

```
class Employee {  
    private:  
        int empno;  
        char name[20];  
        double basicSal;  
        double allowance;  
        double salary;  
    public:  
        void assignDetails(int pempno, char  
                           pname[], double pbasicSal);  
        void setAllowance(double pallowance);  
        void calcSalary();  
        void printPaySlip();  
};
```

```
int main() {  
    Employee emp;  
    emp.printPaySlip();  
    return 0;  
}
```

Output

EmpNo	:	142
Name	:	
Basic Salary	:	2.07322e-317
Allowance	:	6.95252e-310
Net Salary	:	0

How do we assign values to the emp object when it is created? The output shown above displays random values.

Employee Class – From Lab

```
class Employee {  
    private:  
        int empno;  
        char name[20];  
        double basicSal;  
        double allowance;  
        double salary;  
    public:  
        void assignDetails(int pempno, char  
                           pname[], double pbasicSal);  
        void setAllowance(double pallowance);  
        void calcSalary();  
        void printPaySlip();  
};
```

```
int main() {  
    Employee emp;  
    emp.assignDetails(0, "", 0, 0);  
    emp.printPaySlip();  
    return 0;  
}
```

Output

EmpNo	:	0
Name	:	
Basic Salary	:	0
Allowance	:	6.9528e-310
Net Salary	:	0

We could call the assignDetails() method before printing,
But we are not handling the allowance in this method.

What if we use another assignDetails() method to set values at the beginning (initialize)

```
class Employee {  
    private:  
        int empno;  
        char name[20];  
        double basicSal;  
        double allowance;  
        double salary;  
    public:  
        void assignDetails();  
        void assignDetails(int pempno, char  
                          pname[], double pbasicSal);  
        void setAllowance(double pallowance);  
        void calcSalary();  
        void printPaySlip();  
};
```

```
void Employee::assignDetails() {  
    empno = 0;  
    strcpy(name, "");  
    basicSal = 0;  
    allowance = 0;  
}  
  
void Employee::assignDetails(int pempno,  
                           char pname[], double pbasicSal) {  
    empno = pempno;  
    strcpy(name, pname);  
    basicSal = pbasicSal;  
}
```

void assignDetails(int pempno, char pname, pbasicSal) is now an overloaded method.

Using assignDetails() method

```
class Employee {  
    private:  
        int empno;  
        char name[20];  
        double basicSal;  
        double allowance;  
        double salary;  
    public:  
        void assignDetails();  
        void assignDetails(int pempno, char  
                          pname[], double pbasicSal);  
        void setAllowance(double pallowance);  
        void calcSalary();  
        void printPaySlip();  
};
```

```
int main() {  
    Employee emp;  
    emp.assignDetails();  
    emp.printPaySlip();  
    return 0;  
}
```

Output

```
-----  
EmpNo : 0  
Name :  
Basic Salary : 0  
Allowance : 0  
Net Salary : 0
```

Now it seems to be okay. But what if we want to do this automatically when an object is created.

Initializing the Attributes

- Initialize means setting values to variables (attributes) at the beginning (usually when they are declared).
- Although we add another method to the class called “assignDetails()”, it does not initialize the attributes automatically, instead when called it assigns or overwrites the garbage values that are already in the attributes.
- We could automatically initialize attributes in a class by using a special kind of methods called “**Constructors**”

Using Constructors

```
class Employee {  
private:  
    int empno;  
    char name[20];  
    double basicSal;  
    double allowance;  
    double salary;  
public:  
    Employee();  
    Employee(int pempno, char  
             pname[], double pbasicSal);  
    void setAllowance(double pallowance);  
    void calcSalary();  
    void printPaySlip();  
};
```

```
// default constructor  
Employee::Employee() {  
    empno = 0;  
    strcpy(name, "");  
    basicSal = 0;  
    allowance = 0;  
}  
// Constructor with parameters  
Employee::Employee(int pempno,  
                    char pname[], double pbasicSal) {  
    empno = pempno;  
    strcpy(name, pname);  
    basicSal = pbasicSal;  
}
```

Constructors don't have a return type. They have the same name as the Class. **Employee()** is called the **default constructor** And **Employee(int pempno, char ..)** is the **overloaded constructor**

Constructor

- The constructor is used to initialize the object when it is declared.
- The constructor does not return a value, and has no return type (not even void)
- The constructor has the same name as the class.
- There can be default constructors or constructors with parameters
- When an object is declared the appropriate constructor is executed.

Constructors

- Default Constructors
 - Can be used to initialized attributes to default values

```
Employee::Employee() {  
    empno = 0; strcpy(name, "");  
    basicSal = 0; allowance = 0;  
}
```

- Overloaded Constructors (Constructors with Parameters)
 - Can be used to assign values sent by the main program as arguments

```
Employee::Employee(int pempno, char pname[], double pbasicSal) {  
    empno = pempno; strcpy(name, pname);  
    basicSal = pbasicSal;  
}
```

Using Default Constructor

```
class Employee {  
    private:  
        int empno;  
        char name[20];  
        double basicSal;  
        double allowance;  
        double salary;  
    public:  
        Employee();  
        Employee(int pempno, char  
                 pname[], double pbasicSal);  
        void setAllowance(double pallowance);  
        void calcSalary();  
        void printPaySlip();  
};
```

```
int main() {  
    Employee emp;  
    emp.printPaySlip();  
    return 0;  
}
```

Output

```
-----  
EmpNo : 0  
Name :  
Basic Salary : 0  
Allowance : 0  
Net Salary : 0
```

When we create the emp object in the main() program,
the default constructor is automatically executed.

Using Overloaded Constructor

```
class Employee {  
private:  
    int empno;  
    char name[20];  
    double basicSal;  
    double allowance;  
    double salary;  
  
public:  
    Employee();  
    Employee(int pempno, char  
            pname[], double pbasicSal);  
    void setAllowance(double pallowance);  
    void calcSalary();  
    void printPaySlip();  
};
```

```
int main() {  
    Employee emp(100,  
                "Niranjan", 5000);  
    emp.printPaySlip();  
    return 0;  
}
```

Output

```
-----  
EmpNo      :100  
Name       :Niranjan  
Basic Salary :5000  
Allowance   :0  
Net Salary  :0  
-----
```

When we create the emp object in the main() program,
the overloaded constructor is automatically executed.



Exercise 1 - Constructors

Rectangle	
-	length
-	width
+	setWidth ()
+	getWidth()
+	setLength ()
+	getLength()
+	calcArea ()

```
class Rectangle {  
    private:  
        int width;  
        int length;  
    public:  
        void setWidth(int w);  
        int getWidth();  
        void setLength(int l);  
        int getLength();  
        int calcArea();  
};
```

Implement the default and overloaded constructors

Solution

```
// default Constructor
Rectangle::Rectangle () {
    length = 0;
    width = 0;
}

// Constructor with parameters
Rectangle::Rectangle (int l, int w) {
    length = l;
    width = w;
}
```

```
class Rectangle {  
    private:  
        int width;  
        int length;  
    public:  
        Rectangle();  
        Rectangle(int l, int w );  
        void setWidth(int w);  
        int getWidth();  
        void setLength(int l);  
        int getLength();  
        int calcArea();  
};
```

Exercise 2 – Create two objects using the two Constructors.

```
class Rectangle {  
    private:  
        int width;  
        int length;  
    public:  
        Rectangle();  
        Rectangle( int l, int w);  
        void setWidth(int w);  
        int getWidth();  
        void setLength(int l);  
        int getLength();  
        int calcArea();  
};
```

Rectangle Class Solution

```
18 Rectangle::Rectangle() {
19     length = 0;
20     width = 0;
21 }
22 Rectangle::Rectangle(int l, int w) {
23     length = l;
24     width = w;
25 }

* * *
43 int main() {
44     Rectangle rec1;
45     Rectangle rec2(10, 5);
46
47     cout << "Rectangle 1 = length - "
48         << rec1.getLength()
49         << ", width - "
50         << rec1.getWidth()
51         << endl;
52
53     cout << "Rectangle 2 = length - "
54         << rec2.getLength()
55         << ", width - "
56         << rec2.getWidth()
57         << endl;
58
59     return 0;
60 }
```

```
4 class Rectangle {
5     private:
6         int width;
7         int length;
8     public:
9         Rectangle();
10        Rectangle(int l, int w );
11        void setWidth(int w);
12        int getWidth();
13        void setLength(int l);
14        int getLength();
15        int calcArea();
16 };
```

Output

```
Rectangle 1 = length - 0, width - 0
Rectangle 2 = length - 10, width - 5
```

Destructor

- A class's destructor is called implicitly when a object is destroyed.
- An object gets destroyed when a program execution terminates or leaves the scope in which the object was created.
- A destructor can be used to release memory of attributes that were created dynamically when the object was created.
- The name of the destructors is as same as the class name with a ~ (tilde) at the beginning
- It does not specifies any parameters or a return type.

Destructor

```
Rectangle::~Rectangle () {  
    cout << "Destructor runs" << endl;  
}
```

```
class Rectangle {  
private:  
    int width;  
    int length;  
public:  
    Rectangle();  
    Rectangle( int w, int l );  
    void setWidth(int w);  
    int getWidth();  
    void setLength(int l);  
    int getLength();  
    int calcArea();  
    ~Rectangle();  
};
```

Destructors used in Rectangle Class

```
27 Rectangle::~Rectangle() {
28     cout << "Destructor Runs for Rec with Len = "
29         << length << " and width = " << width << endl;
30 }
31
32 int main() {
33     Rectangle rec1;
34     Rectangle rec2(10, 5);
35
36     cout << "Rectangle 1 = length - "
37         << rec1.getLength()
38         << ", width - "
39         << rec1.getWidth()
40         << endl;
41
42     cout << "Rectangle 2 = length - "
43         << rec2.getLength()
44         << ", width - "
45         << rec2.getWidth()
46         << endl;
47
48     return 0;
49 }
```

```
4 class Rectangle {
5     private:
6         int width;
7         int length;
8     public:
9         Rectangle();
10        Rectangle(int l, int w );
11        void setWidth(int w);
12        int getWidth();
13        void setLength(int l);
14        int getLength();
15        int calcArea();
16        ~Rectangle();
17 };
```

Output

```
Rectangle 1 = length - 0, width - 0
Rectangle 2 = length - 10, width - 5
Destructor Runs for Rec with Len = 10 and width = 5
Destructor Runs for Rec with Len = 0 and width = 0
```

Static Objects

- Using Default Constructor

```
Rectangle R1;
```

R1
length - 0
width – 0

- Using constructor with Parameters

```
Rectangle R2 ( 100, 50) ;
```

R2
length - 100
width – 50

Static Objects- Accessing methods

```
Rectangle R1;  
  
R1.setLength( 100 );  
R1.setWidth(50);
```

The dot (.) operator is used to access the public methods of a static object

Dynamic Objects

```
Rectangle *r;  
  
r = new Rectangle ();  
  
r -> setWidth(100);  
r -> setLength(50);  
cout<<"Area is : "<< r -> calcArea();  
  
delete r;
```

The arrow (->) is used to access the public methods of a dynamic object

Dynamic Objects

- Most programming languages only support Dynamic Objects.
- You need to delete the allocated memory in C++.
- The new command is used to allocate memory for an object.
- The delete command needs to be used to deallocate memory (release memory) once we have finished using the objects.

Dynamic Objects

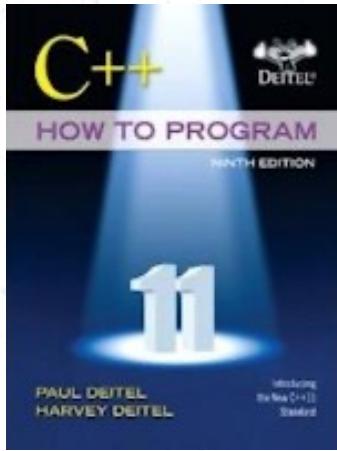
```
64 Rectangle *rec3, *rec4;
65
66 rec3 = new Rectangle();
67 rec4 = new Rectangle(20, 10);
68
+     cout << "Rectangle 3 = length - "
+         << rec3->getLength()
+         << ", width - "
+         << rec3->getWidth()
+         << endl;
74
+     cout << "Rectangle 4 = length - "
+         << rec4->getLength()
+         << ", width - "
+         << rec4->getWidth()
+         << endl;
80
81     delete rec3;
82     delete rec4;
83
```

```
4 class Rectangle {
5     private:
6         int width;
7         int length;
8     public:
9         Rectangle();
10        Rectangle(int l, int w );
11        void setWidth(int w);
12        int getWidth();
13        void setLength(int l);
14        int getLength();
15        int calcArea();
16        ~Rectangle();
17 }
```

Output

```
Rectangle 3 = length - 0, width - 0
Rectangle 4 = length - 20, width - 10
Destructor Runs for Rec with Len = 0 and width = 0
Destructor Runs for Rec with Len = 20 and width = 10
+
```

Reference



Chapter 09 & 10

Constructors and Destructors

Deitel & Deitel's (2016), C++ How to Program,
9th Edition



SLIIT

Discover Your Future

IT1050-Object Oriented Concepts

Noun Verb Analysis and CRC Cards
Lecture-08



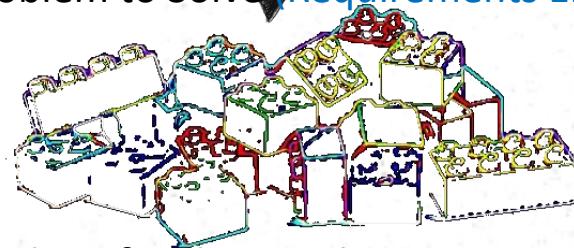
Agenda

- Object Oriented Analysis
 - Identifying Classes
 - Noun and Verb Method
- •
- •
- •
- •
- •
- •
- •
- •

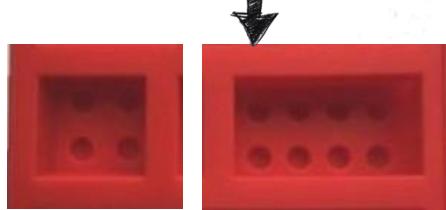
Software Engineering Coding



Problem to Solve (Requirements Engineering)

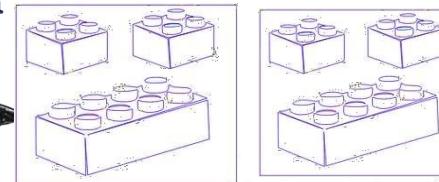


Identify Objects that are needed
(Object Oriented Analysis and Design)



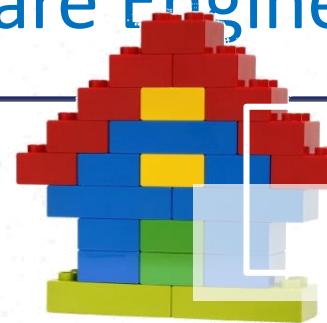
Identify Classes through Abstraction
and Implementation (writing Block class)

```
class Block {  
private:
```



```
Block block1, block2, block3, block4
```

Create Objects from Classes (In main function)



```
block1.assemble(block2);  
block2.assemble(block3);  
block3.assemble(block4);
```

Assemble Objects
to create the
solution (in main
function)

```
Block1.setColor(yellow);  
block2.setColor(blue);  
block3.setColor(yellow);  
block4.setColor(red);
```

SDLC – Software Development Life Cycle

- Requirements Gathering
 - Describe the Application
 - Requirements should ideally be describing the new application that needs to be built, not how things are done manually
- Analysis
- Design
- Implementation (Coding)

OOA is part of the SDLC

OOA

- Requirements are generally represented as
 - Use Case Diagrams
 - Use Case Scenarios
- • Or
- • as User Stories
- In Object Oriented Analysis we take the requirements captured as above and try to develop a Class Diagram

Object Oriented Analysis

- Discover Classes for the requirements.
 - 1. Noun/Verb Analysis
 - 2. CRC Method
 -
 -
 -
 -
 -
 -
 -

Noun/Verb Analysis

Noun/Verb Analysis

- Remember that, in general, classes correspond to nouns, which are objects— people, places, and things.
- Steps:
 - Identify objects in our problem statement by looking for nouns and noun phrases.
 - Each of these can be underlined and becomes a candidate for an object in our solution.
 - We can eliminate some objects by some simple rules.



Common Nouns and Proper Nouns

- Common Nouns – Correspond to **Classes**
 - A common noun (e.g., **Person**) is a name of a class of beings or things.
- Proper Nouns – Correspond to **Objects**
 - A name used for an individual person, place, or organization, spelled with an initial capital letter,
 - e.g. **Jagath, Dehiwala, and Keells**

Activity-1

Library System – a Description

- In the Library a member can borrow, return Books
- The people using the library can also search for Books.
- The users of the library from the Faculty of Computing belong to the Department of IT/CSSE/CSE
- Each Book has an ISBN
- The SLIIT Librarian is Ms Pushpamala Perera.

Library System (Nouns in Red)

- In the Library a member can borrow, return Books
- The people using the library can also search for Books.
- The users of the library from the Faculty of Computing belong to the Department of IT/CSSE/CSE
- Each Book has an ISBN
- The SLIIT Librarian is Ms Pushpamala Perera.

Rules for Rejecting Nouns

Think about Library system of a University.

1. Redundant - In a Library system member and user refers the same person.
2. An event or an operation – Search book is the operation of library system.
3. Outside scope of system – University Department (CSSE/IT) is outside scope of library.
4. Meta-language (Meta language is words or symbols for talking about language itself.)- In a library system people who are using library system can call as member.
5. An attribute – ISBN of a book is an attribute.

Note : 1 and 2 are similar in most situations

Activity 2- Identify Nouns

- In a DVD rental store there are two types of users, a registered member can borrow up to 3 DVDs at a time. These members have already paid a deposit and only need to pay 50/= per DVD.
- Unregistered members can also borrow DVDs at the rate of 75/= per DVD. They are required to provide their id card for this purpose.
- Members can keep the DVD for three days and when they are returned appropriate fines may be calculated.

Activity 2

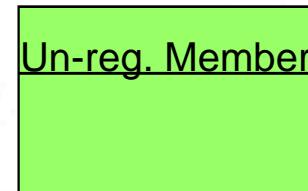
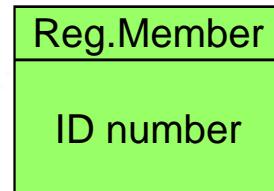
- In a **DVD rental store** there are two types of **users**, a **registered member** can borrow up to 3 DVDs at a time. These members have already paid a **deposit** and only need to pay 50/= per DVD.
- **Unregistered members** can also borrow DVDs at the **rate** of 75/= per DVD. They are required to provide their **id card** for this purpose.
- **Members** can keep the DVD for three days and when they are returned appropriate **fines** may be calculated.

Activity 2 Answer:

- DVD class
- User/Member Redundant
- Unregistered member Class
- Registered Class
- Member
- ID Card ID number is an attribute
 Attribute
- Fine Attribute
- Deposit Attribute / operation
- Store Out of scope
- Rate Attribute

Final Classes ?

- DVD
- Reg. Member
- Un reg. Member

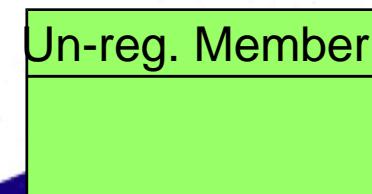
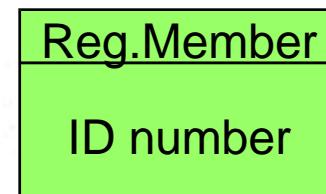


Activity 2 Answer:

- DVD class
 - User
 - Unregistered member } Redundant
 - Member (registered) Meta language
 - ID Card ID number is an attribute Attribute
 - Fine
 - Deposit Attribute / operation
 - Store Out of scope
 - Rate Attribute

Final Classes ?

- DVD
 - Reg. Member
 - Un reg. Member



Activity 3 – Online Order System

- A customer in an online store needs to first register providing details such as name, address.
- The online store administrator can add new items to the store, restock (increase quantity), generate a list of items that need to be restocked.
- A Customer can place an Order from an online store. An Order consists of multiple items.
 - The customer can see the status of the Orders placed, and get a list of previous orders made.
 - The customer specifies a payment method (credit card, debit card, pay pal) for each order.
 - Once the customer confirms the order and the payment is validated the order is placed and items are updated.

Activity 3 – Online Order System

- A **customer** in an online **store** needs to first register providing details such as name, address.
- The online **store administrator** can add new **items** to the **store**, restock (increase quantity), generate a **list of items** that need to be restocked.
- A **Customer** can place an **Order** from an online store. An **Order** consists of multiple **items**.
- The **customer** can see the **status** of the **Orders** placed, and get a **list of previous orders made**.
- The **customer** specifies a **payment** method for each order. e.g. **credit card, debit card, paypal**
- Once the **customer** confirms the **order** and the **payment** is validated the **order** is placed and **items** are updated.

Activity 3 – Online Order System

- list of items - **Item**, (paypal, credit card, debit card) - **Payment**, list of previous order - **Order** – **Redundant**
- Administrator – **Outside Scope of System**, this is actually a user of the system (An Actor)
- Store – the system itself (**outside the scope**)
- Customer - **Class**
- Payment - **Class**
- Order – **Class**
- Item – **Class**
- Name, address, status – **Attributes of Book**

Verbs

- Definition
 - a word used to describe an action, state, or occurrence, and forming the main part of the predicate of a sentence, such as *hear, become, happen.*
- Maps to methods in a class

Activity 4 – Identify Verbs (potential methods)

- A customer in an online store needs to first register providing details such as name, address.
- The online store administrator can add new items to the store, restock (increase quantity), generate a list of items that need to be restocked.
- A Customer can place an Order from an online store. An Order consists of multiple items.
- The customer can see the status of the Orders placed, and get a list of previous orders made.
- The customer specifies a payment method (credit card, debit card, paypal) for each order.
- Once the customer confirms the order and the payment is validated the order is placed and items are updated.

Activity 4 – Identify Verbs (potential methods)

- A customer in an online store needs to first **register providing** details such as name, address.
- The online store administrator can **add** new items to the store, **restock** (increase quantity), **generate** a list of items that need to be restocked.
- A Customer can **place** an Order from an online store. An Order consists of multiple items.
- The customer can **see** the status of the Orders placed, and **get** a list of previous orders made.
- The customer **specifies** a payment method (credit card, debit card, pay pal) for each order.
- Once the customer **confirms** the order and the payment is **validated** the order is **placed** and items are **updated**.

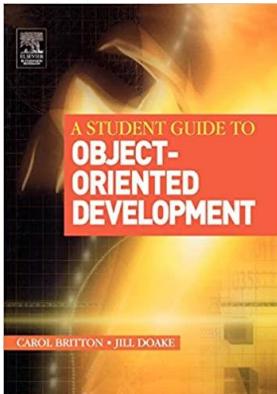
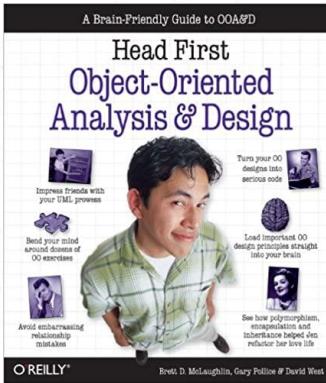
Activity 4 – Verbs are methods

- A customer in an online store needs to first register providing details such as name, address.
- The online store administrator can add new items to the store, restock (increase quantity), generate a list of items that need to be restocked.
- A Customer can place an Order from an online store. An Order consists of multiple items.
• • •
• The customer can see the status of the Orders placed, and get a list of previous orders made.
- The customer specifies-a payment method (credit card, debit card, paypal) for each order.
• • •
• Once the customer confirms the order and the payment is validated the order is placed and items are updated.

Activity 4 - Methods

- Customer – Register
- Payment - Validated
- Order – “Place an Order”, “See status of Order Placed”, Confirm
- Item – Add, Restock, Updated
- Report – “Generate List of Items”, “List of Previous orders”
- Sometimes it is not clear which class should contain which method. We can use CRC cards in such situations.

References



- Head First Object-Oriented Analysis and Design,
1st Edition – Chapters 4.
- A Student Guide to Object Oriented Development,
1st Edition – Chapter 5



Discover Your Future

Lecture 09

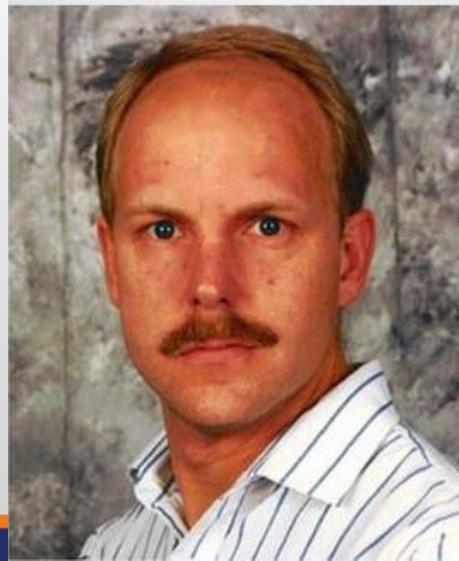
Object Oriented Analysis Using CRC Cards

Agenda

- Object Oriented Analysis
 - Class Responsibility and Collaboration (CRC) Cards.

What is CRC Cards Method?

- Class Responsibility and Collaboration (CRC) cards:
 - Developed by Kent Beck and Ward Cunningham (1989).
 - Used to explore class relationships between classes in a particular use- case scenario.



A CRC Card

A CRC is a 4×6 inch card divided into three sections.

- The **name** of the class on the top of the card.
- The **responsibilities** of the class on the left of the card.
- The class **collaborations** on the right of the card.
 - (the list of other classes with which the class collaborates to realize its responsibilities)

Class name:	
Responsibilities:	Collaborations:

Identifying the Classes' Responsibilities

Responsibilities relate to actions. You can generally identify responsibilities by selecting the verbs from the summary of the requirements.

The list of responsibilities on the CRC card is **not the list of methods** in the class.

- A responsibility may be realized by several methods.
- e.g. An **Item** class in a Supermarket may have the responsibility of **“Store Details of Item”**
 - We may need the following methods to realize the responsibility
 - **addItems(), updateItems(), deleteItems()**

What is a Collaboration?

- To identify the collaborations, we need to study the responsibilities and determine what other classes the object interacts with.
- A class may not be able to act upon its responsibility on its own.
- It may require some interaction with other classes.(need help)
 - these helper classes are the **collaborators**.

e.g. An **ItemReport** class that has responsibility of “producing a list of **Items** which needs to be **reordered**” would need to collaborate with the **Item** class

Activity 1:

Scenario step :

" You (*the student*) should e-mail your CV to my (*Lecturer's*) boss "



Write few CRC cards for this

Steps

1. Take these as the *classes* for (the first) cards.
Student, CV, Boss, Lecturer
2. The only verb you need to make this scenario happen is
"send an e-mail with CV"
3. Add that as a *responsibility* to the "Student" card. However, Student can not do this activity alone. (because the card doesn't have enough information).

So, Student needs two *collaborators/helpers* :

Have to attach **The CV** > student should have a **CV**
Find **boss's e-mail address**. > student should ask **Lecturer**

Activity 1: Basic CRC Cards

Student	
Responsibility	Collaborators
e-mail CV to a given e-mail address.	CV
find e-mail address.	lecturer

CV	
Responsibility	Collaborators

Boss	
Responsibility	Collaborators

Lecturer	
Responsibility	Collaborators
Know boss's e-mail address.	

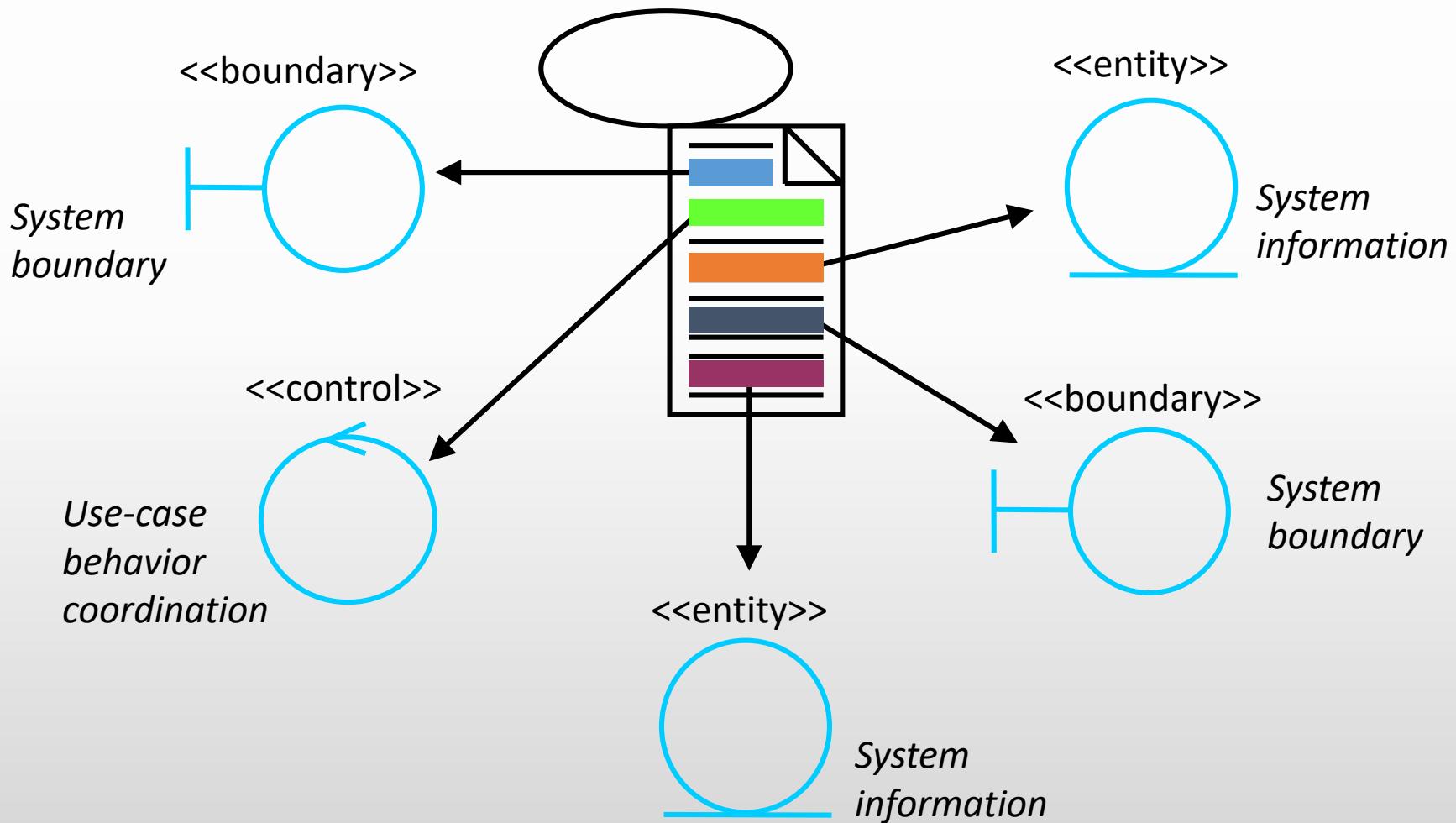
Activity 2

- In a DVD rental store there are two types of users, a registered member can borrow up to 3 DVDs at a time. These members have already paid a deposit and only need to pay 50/= per DVD.
- Unregistered members can also borrow DVDs at the rate of 75/= per DVD. They are required to provide their id card for this purpose.
- Members can keep the DVD for three days and when they are returned appropriate fines may be calculated.

Activity 2 - Solution

DVD Class		Registered Member Class		Unregistered Member Class	
<u>Responsibilities</u>	<u>Collaborations</u>	<u>Responsibilities</u>	<u>Collaborations</u>	<u>Responsibilities</u>	<u>Collaborations</u>
Store Details of DVDs		Pay Deposit Borrow DVD Return DVD Pay Fine	DVD DVD	Borrow DVD Return DVD Pay Fine	DVD DVD

What Is an Analysis Class?



Types of Analysis Classes

- Entity Classes – Classes that we have considered upto now.
- Boundary Classes – Interaction classes, Forms, Reports. We can one Boundary Class per Actor.
- Control classes – In a complex use case, the use case itself can be a class. Typically we can have one control class per complex use case.

Actors as Entity Classes

- Actors are users of the system in a use case diagram.
- Actors if they only have login credentials in the system will typically not be classes. e.g. user accounts, administrator.
- However if the Actor needs to do provide his/her data and that is directly relevant to the application domain then such actors would be classes.
- e.g. A Customer in an Online Store, has to register and provide details of shipping address, billing address, contact details would be a class.

Activity 3 – Online Order System

- A **customer** in an online **store** needs to first register providing details such as **name, address**.
- The online **store administrator** can add new **items** to the **store**, restock (increase quantity), generate a **list of items** that need to be restocked.
- A **Customer** can place an **Order** from an online **store**. An **Order** consists of multiple **items**.
- The **customer** can see the **status** of the **Orders** placed, and get a **list** of previous orders made.
- The **customer** specifies a **payment** method (**credit card, debit card, paypal**) for each order.
- Once the **customer** confirms the **order** and the **payment** is validated the **order** is placed and **items** are updated.

Activity 3

- Customer - Class
- Payment - Class
- Order – Class
- Item – Class

Activity 3

Step 2 –

Identify Responsibilities
For each Class

Responsibilities

For Item Class

- Add New Items Items
- Restock
- ...

Activity 3

Step 3 –

Identify Collaborations for
each Class

Collaborations

For Order Class

- We need to elaborate with the Payment class

Solution

Customer Class	
<u>Responsibilities</u>	<u>Collaborations</u>
Register Details	

Item Class	
<u>Responsibilities</u>	<u>Collaborations</u>
Add Items Restock List of Restock Items Update Items	

Order Class	
<u>Responsibilities</u>	<u>Collaborations</u>
Place Order Status of Order List of previous Order Confirm Order	Payment

Payment Class	
<u>Responsibilities</u>	<u>Collaborations</u>
Store Payment Details Validate	

Handling list of Objects – Some Informal Guidelines

- A list of Previous Orders typically needs to be handled a collection class, not the Order class.
- Similarly the list of items to be reordered ideally should not be in the Item Class.
- An Entity class is typically responsible for handling one object at a time.

Activity 3

Step –

Refine CRC Cards

Solution

Customer Class	
<u>Responsibilities</u>	<u>Collaborations</u>
Register Details	

Item Class	
<u>Responsibilities</u>	<u>Collaborations</u>
Add Items Restock Update Items	

Order Class	
<u>Responsibilities</u>	<u>Collaborations</u>
Place Order Status of Order Confirm Order	Payment

Payment Class	
<u>Responsibilities</u>	<u>Collaborations</u>
Store Payment Details Validate	

Report Class	
<u>Responsibilities</u>	<u>Collaborations</u>
List of Restock Items List of Previous Order	Item Order, Customer

References

- Grady Booch, et al (2007), Object Oriented Analysis and Design with Applications 3rd Edition – Chapters 5.
- Matt Weisfeld, The Object-Oriented Thought Process 3rd Edition

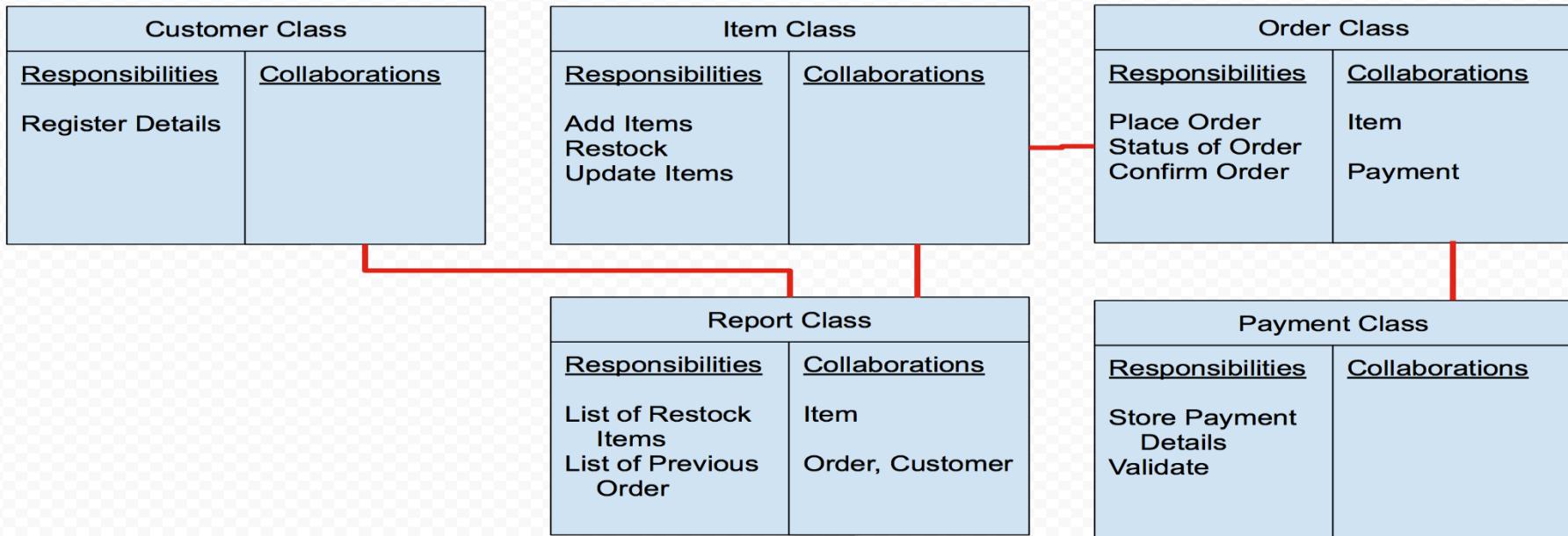
IT1050-Object Oriented Concepts

Relationships and Class Diagram
Lecture-10

Learning Outcomes

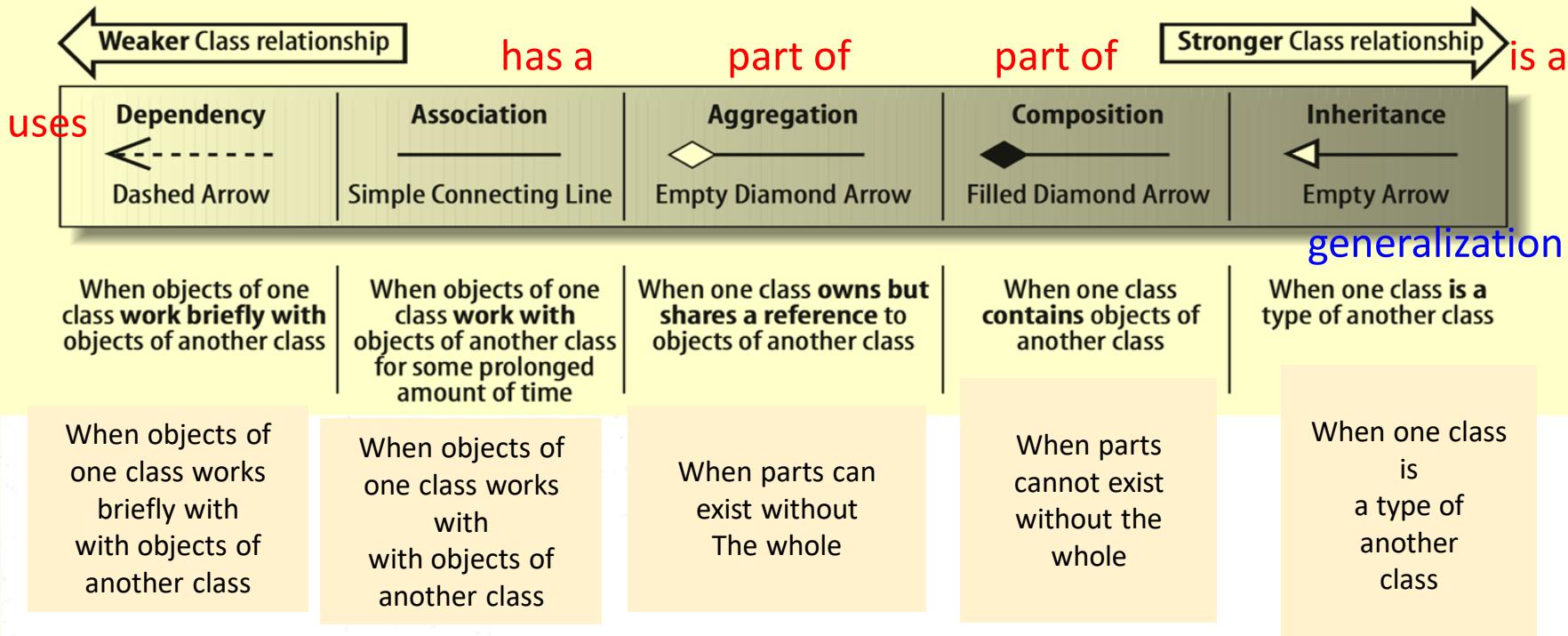
- At the end of the lecture, students should be able to
 - Identify Dependency, Association, Aggregation, Composition, and Inheritance relationships between classes
 - Create a class diagram in UML notation with the Dependency, Association, Aggregation, Composition and Inheritance relationships

Relationships between classes



We can see that there are relationships between classes when we draw CRC cards. We can divide all relationships into five categories

Relationships Between Classes



Inheritance

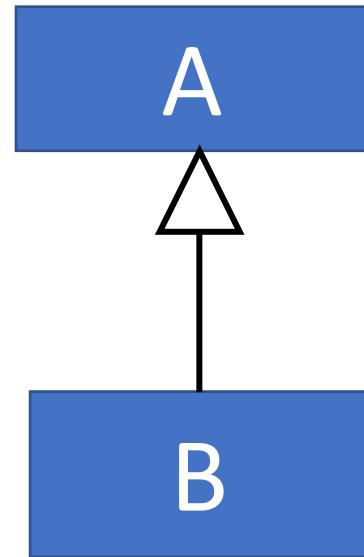
- Otherwise Known as Generalization.
- Inheritance represents a “**is-a-kind-of**” relationship.
- Inheritance is a relationship between a general thing (superclass/parent) and a more specific kind of a thing (subclass/child).
- Graphically, it is rendered as an empty block arrow.



Inheritance

- Class A is a **super class** (parent) of class B if B directly inherits from A.
- Class B is a **sub class** (child) of class A id B directly inherits from A.
- Class A is an **ancestor** of class B if A is above B in the inheritance hierarchy
- Class B is a **descendant** of class A if B is below A in the inheritance hierarchy

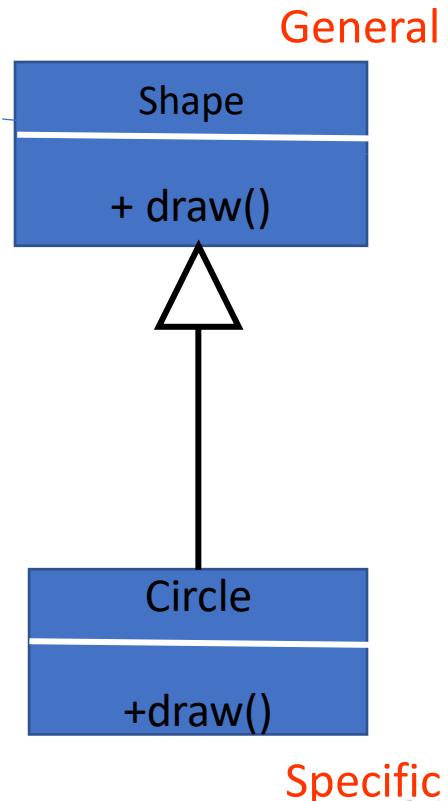
Super class
Ancestor



Sub class
Descendent

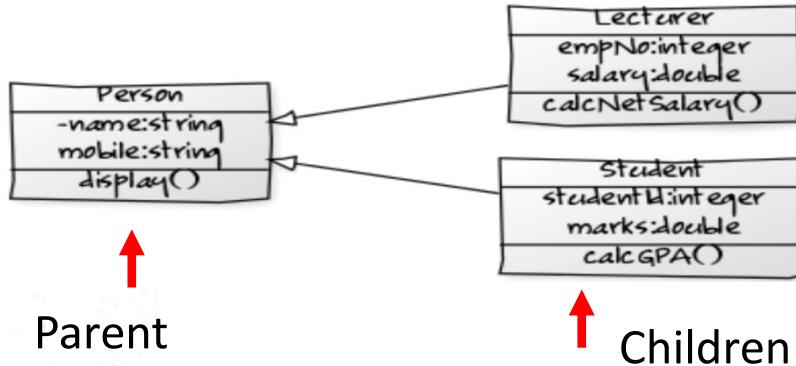
Inheritance

- Generalization takes place from sub-class to super-class: the super-class is a **generalization** of the sub-class.
- Specialization takes place from super-class to sub-class: the sub-class is a **specialization** of the super-class.
- The functionality of the child should be a specialization of the functionality of the parent.
- **e.g. the functionality of the draw method in Circle is more specific than the one in Shape**



Inheritance

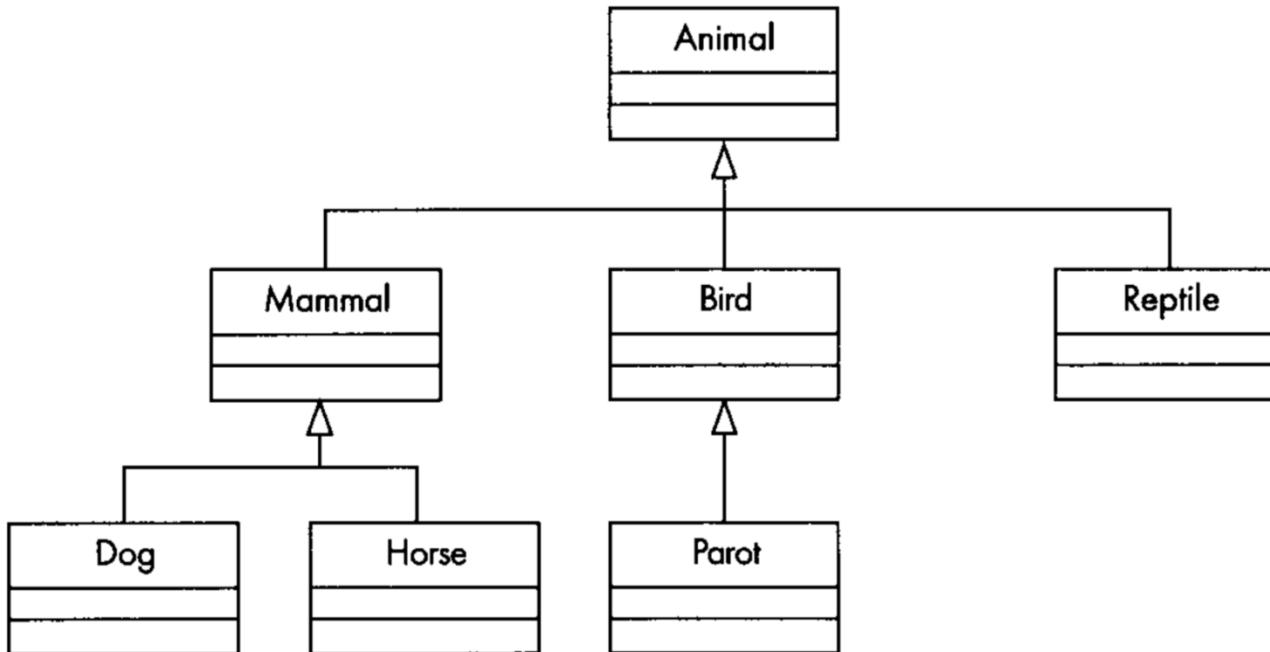
- Super Class / Parent : Person
- Sub Classes / Children : Student, Lecturer
- Children (Student and Lecturer) inherit the properties of its parent's attributes, operations, responsibilities, etc.).



Note : Normally
this diagram is
drawn vertically

Inheritance – is A

Animal, Mammal, Bird, Reptile, Dog, Horse, Parrot



Whole – Part Relationships

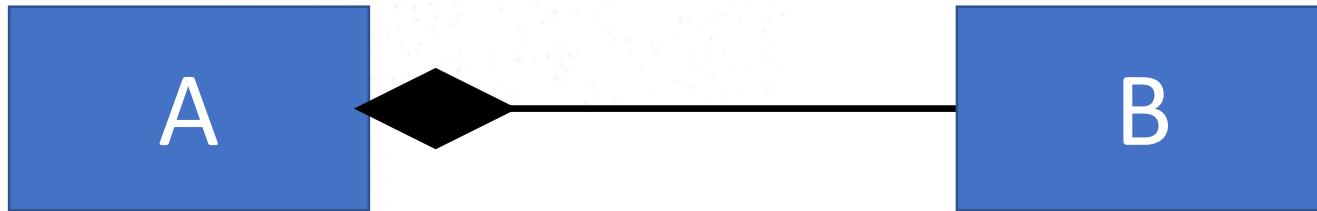
- Aggregation
- Composition

Whole – Part Relationships

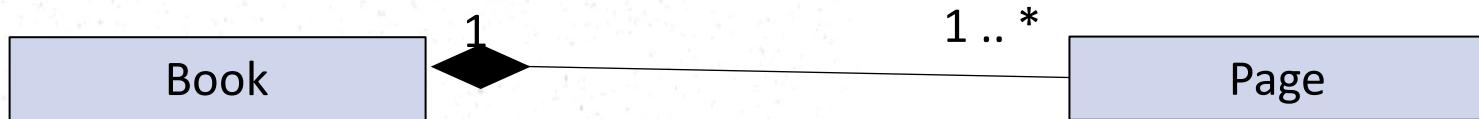
- A "whole/part" relationship refers to a fairly strong connection between two classes.
- One class represents a larger thing ("whole"), which consists of smaller things ("parts").
- This means "part of" relationship. Meaning that an object of the whole has objects of the part.
- Two types;
 - Aggregation (Relatively Weak)
 - Composition (Relatively Strong)

Composition

- Composition is a strong form of whole-part relationship
- Graphically, a dependency is rendered as a filled diamond arrow.
- Composition should have a relationship with a multiplicity of “1 .. ”



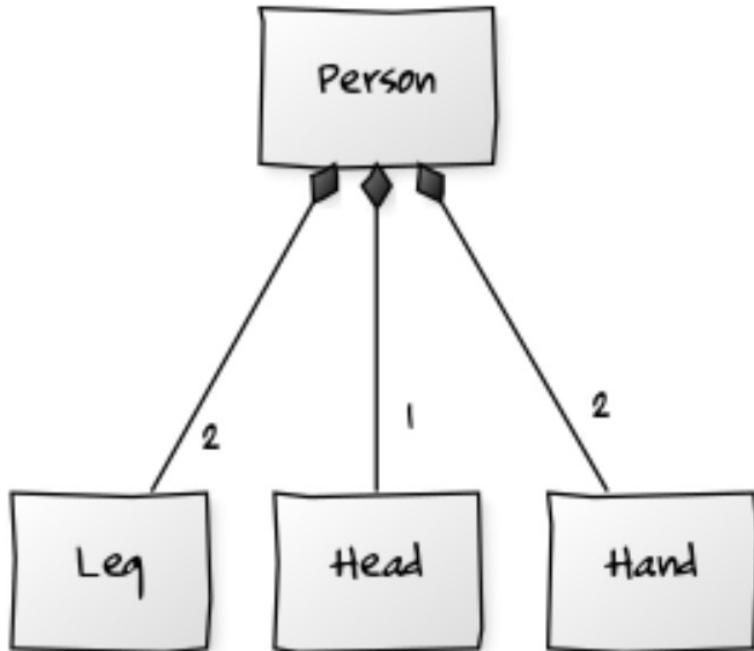
Composition



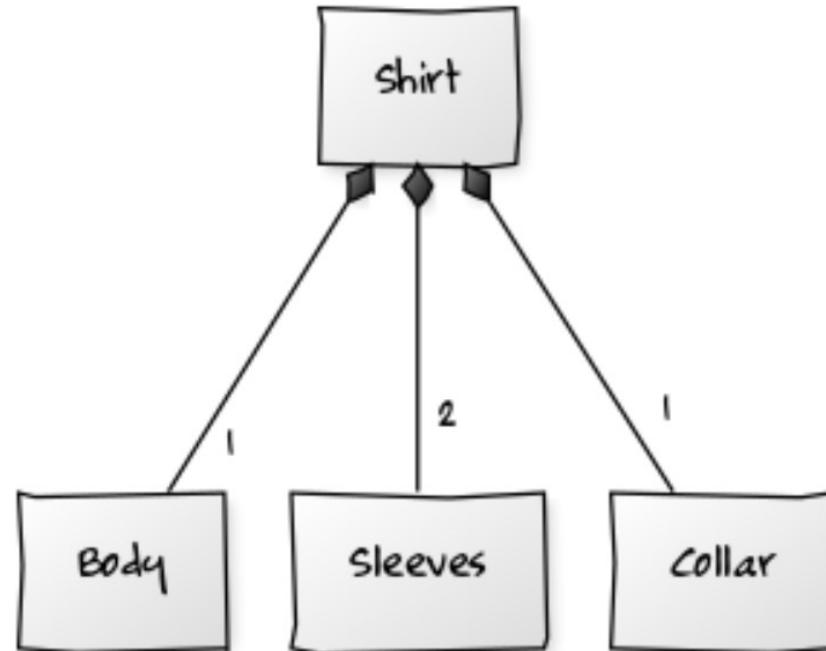
- Whole : Book
- Part : Page
- A Book has pages,
- A Page cannot exist without the Book.
- Implies that the “Part cannot exist without Whole”

Composition

Person, Head, Leg, Hand



Shirt, Body, Sleeve, Collar



Aggregation

- Aggregation is just a special kind of association
- Aggregation is a weak form of whole-part relationship
- Graphically, a dependency is rendered as an empty diamond arrow.
- This means that A aggregates B.

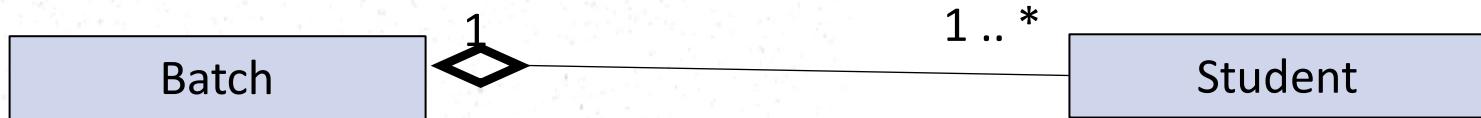


Aggregation

- The same rule for multiplicity can be applied for aggregation relationship.
- In some situations Aggregation can also have a multiplicity of “0..”
- e.g. : A batch consists of **one or more students**



Aggregation



- Whole : Batch
- Part : Student
- A Student can exist without the Batch.
- This implies that the Part can exist without the Whole.

Aggregation

Sentence, Word, Letter



Association

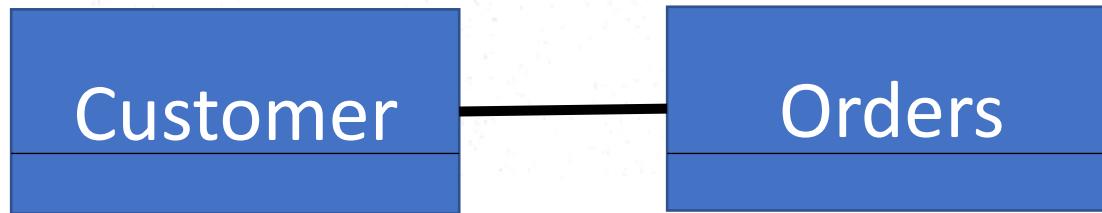
- Association defines a **has a** relationship.
- Association connects one instance of a class with an instance of another class.

- The relationship can be **bi-directional (two way)** or **uni-directional (one way)**

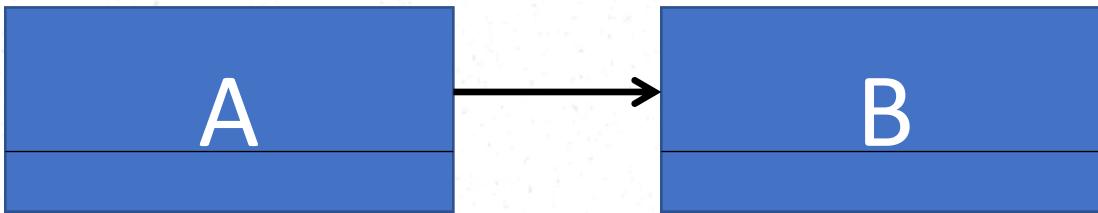
- If there is an arrowhead, it means there is a one-way relationship.

Association

- Example: “A Customer **has** Orders” (A list of previous orders made)

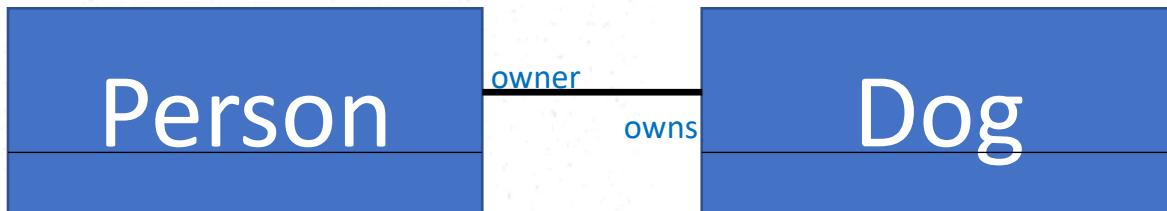


Association (one way)



- Here it means that;
 - Class A is associated with class B
 - Class A uses and contains one instance of class B, but B does not know about it or contain any instances of class A.
- In an Association relationship, the dependent class (A) defines an instance of the associated class (B).

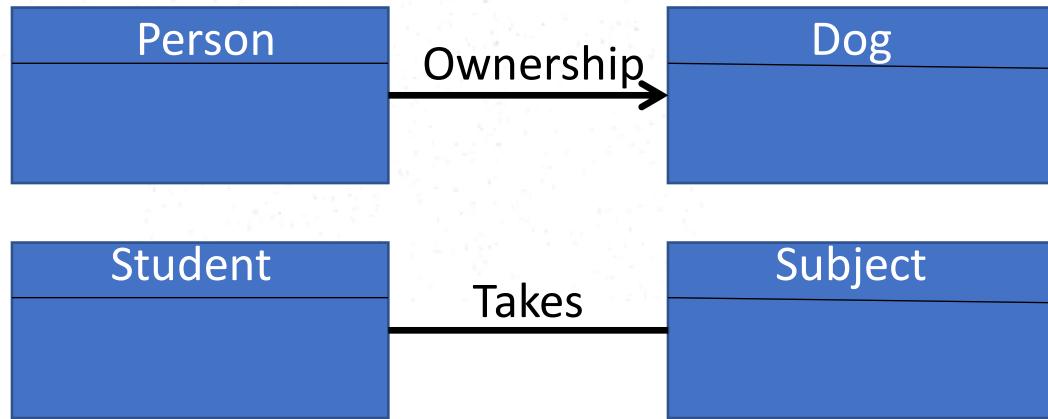
Association (Two way)



- The person class and the Dog class are associated.
- Here. It is **bi-directional**
 - The person is related to the dog in some way
 - The Dog is also related to the person in some way

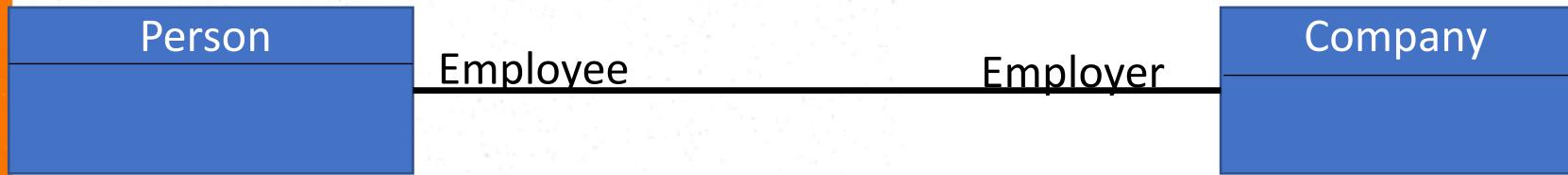
Association

- Association **Name** :
- An Association can be given a name:
(This is only shown if it helps to clarify the association)



Association

- Association **Role**:
- When a class participates in an association, it has a specific role that it plays in that relationship;
- A role is just the face the class at the near end of the association presents to the class at the other end of the association.



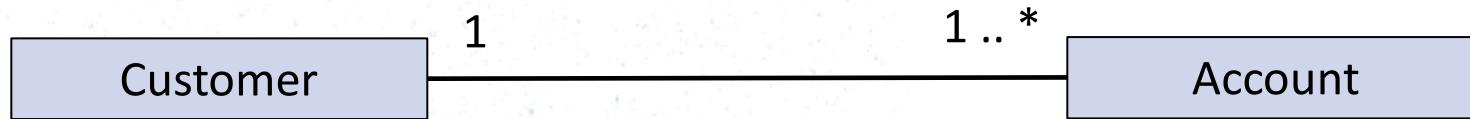
A Person playing the role of **employee** is associated with a Company playing the role of **employer**.

Association

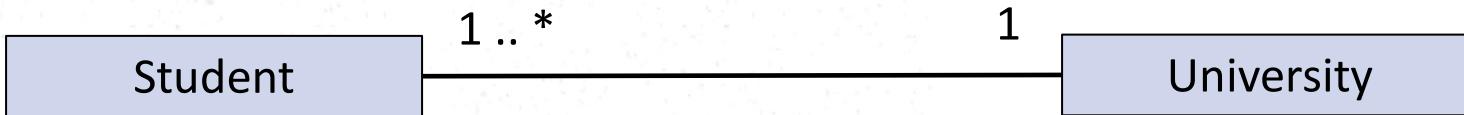
- Association **multiplicity**:
 - Multiplicity defines the number of objects associated with an instance of the association
 - This “How many” is the multiplicity of an association’s role.

Format	Meaning
1	Exactly one
n	Exactly n
0 .. 1	Zero or one
n .. m	Between n and m (inclusive)
0 .. *	Zero or more
1 .. *	One or more
0 .. 1, 3 .. 5	Zero or one, or between 3 and 5 inclusive

Association (Two Way)



Customer can own **1 or more** Accounts. Account belongs to **only one** customer.



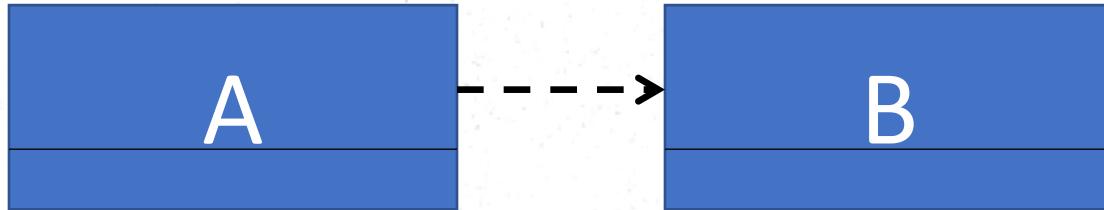
University class has **1 or more** students and a student is attached to **only one** university.

Dependency

- Dependency is a weaker form of relationship which indicates that one class depends on another because it uses it at some point in time.
- It implies that **a change to one class may affect the other but not vice versa.**

Dependency

- Graphically, a dependency is rendered as a dashed directed line, directed to the thing being depended on.

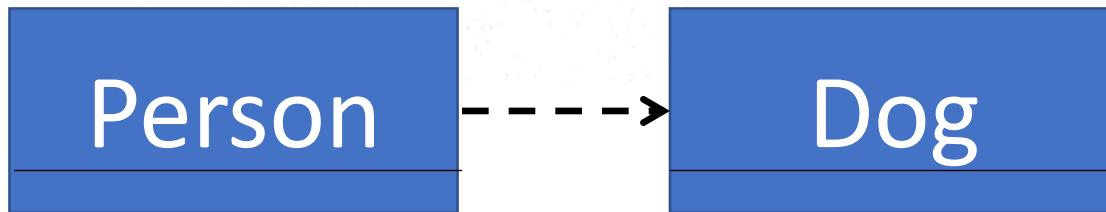


- It means that;
 - A uses B, but A does not contain an instance of B as part of its own state.

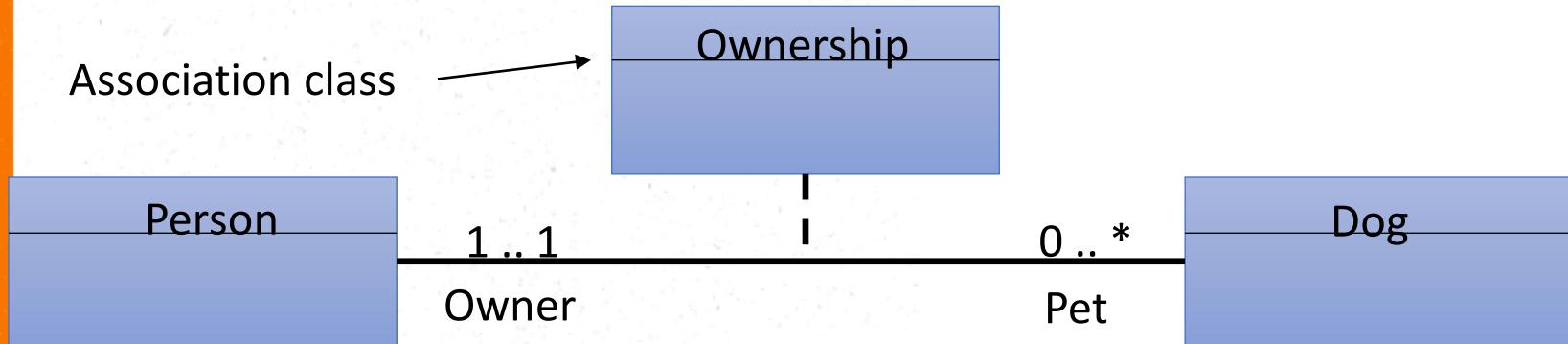
Dependency

- e.g :
- The “Person” class depends on “Dog” class. Changes to “Dog” class may affect the “Person” class, but not vice versa.
- A “Person” object uses a “Dog” object somewhere as a parameter in one of its methods.

`void Person::walk(Dog dog)`

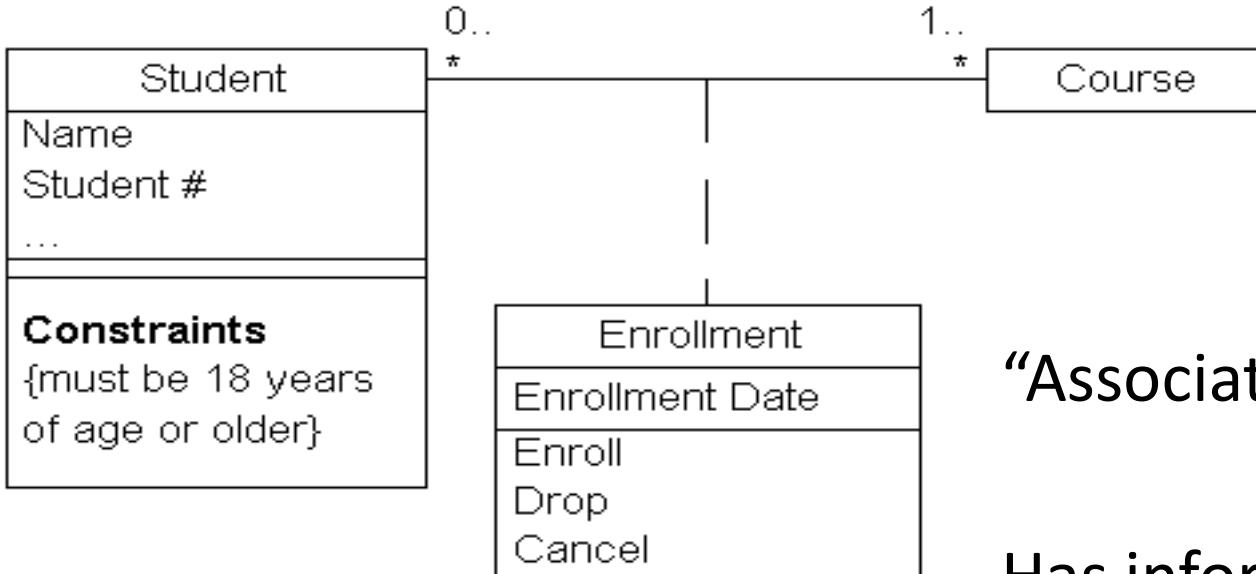


Association class



- The association can also be promoted to a class
- This places the responsibility for maintaining information pertaining to the association with the Ownership class.

Association



“Association Class”

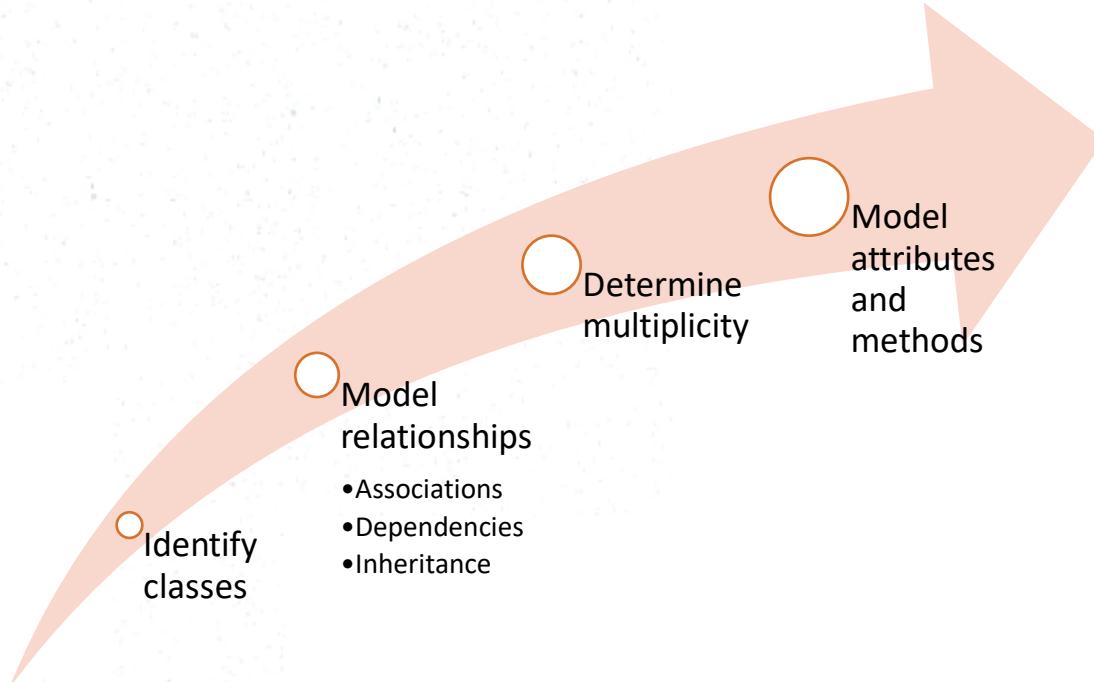
Has information related to the association.

Deciding on Relationships

- Is-A relationship – Inheritance
- Part of – Part cannot exist without the whole – Composition
 - e.g. Pages are part of a Book
- Part of – Part can exist without the whole – Aggregation
 - e.g. Students are part of a Batch
- Has a – Not a direct Part of Relationship - Association
 - e.g. Customer has multiple Orders
 - Airplane can carry (has) multiple Passengers
- Uses - An object is used as a parameter of a method – Dependency
 - e.g. void Player::ThrowDice(Dice mydice);

Here the ThrowDice method in the Player Class has a parameter of the Dice Class.

Creating a Class Diagram



Class Diagram – The Case

- An Exam Paper has many instructions and has one or more Questions. Each Question has a solution.
- Here you can assume that Questions can only be in Exam papers.

Class Diagram – Identifying Classes

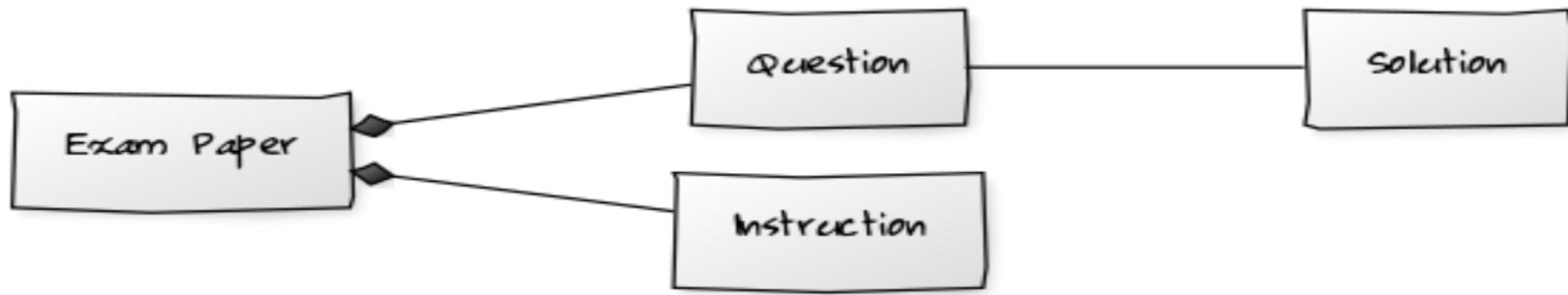
Exam Paper

Instruction

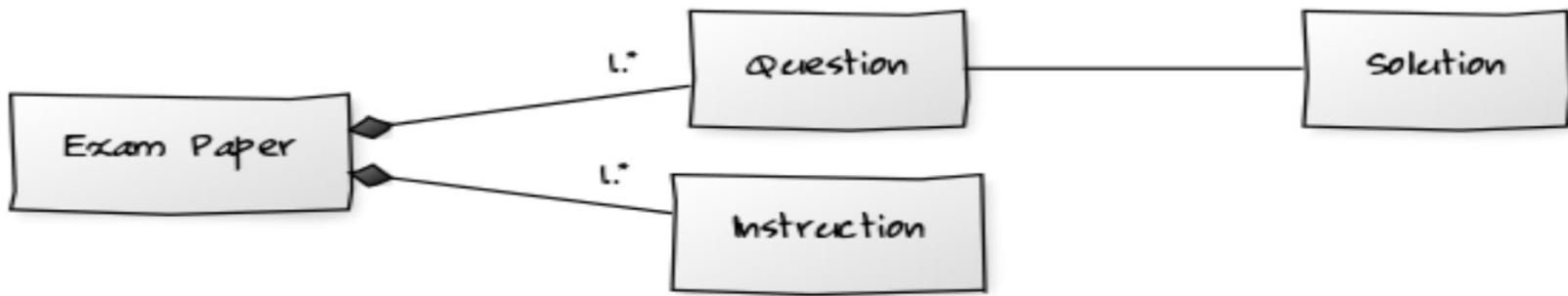
Question

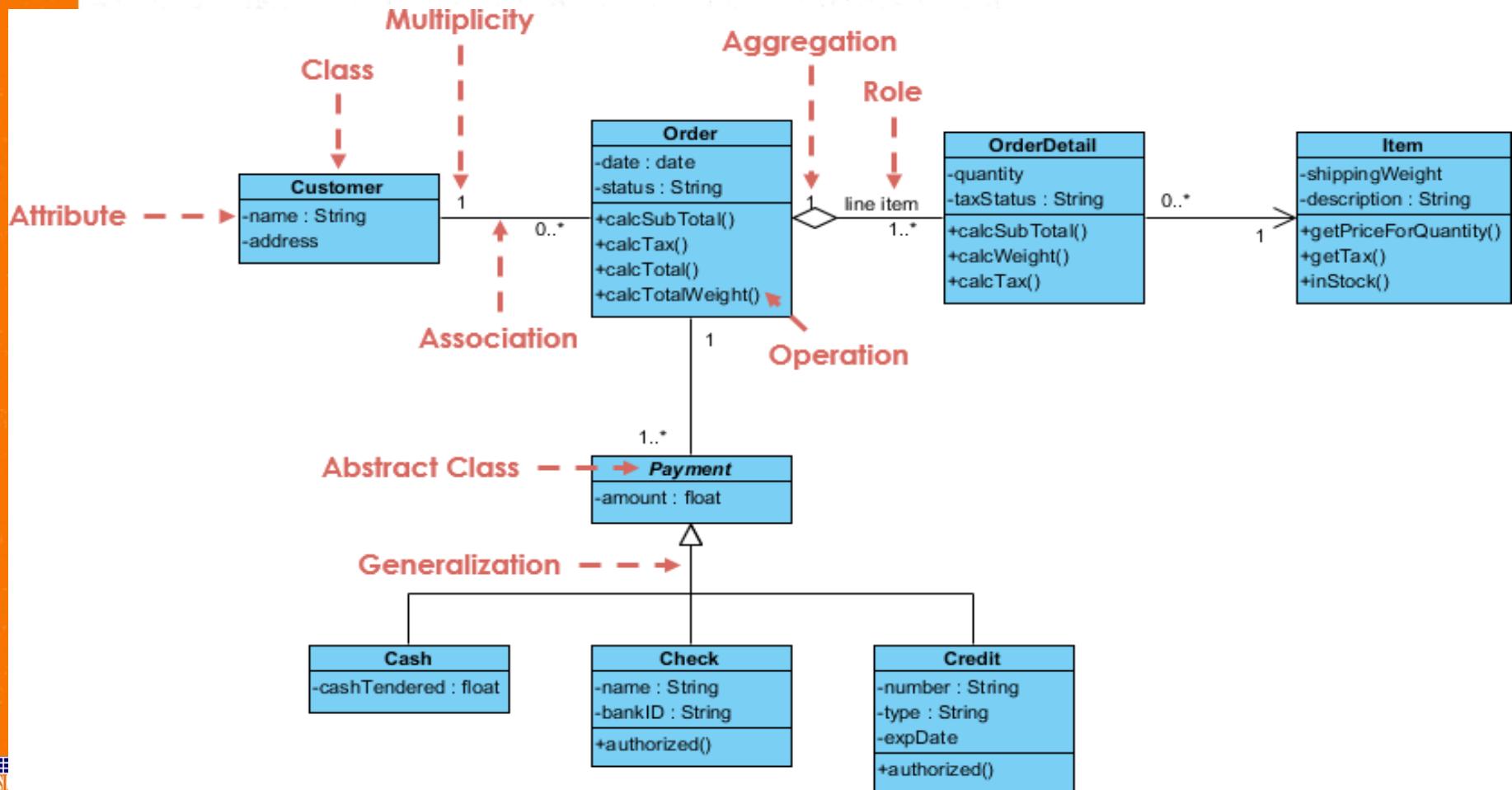
Solution

Class Diagram – Modeling Relationships



Class Diagram – Multiplicity





References

- UML Distilled by Martin Fowler, chapters 3 and 5
- Fundamentals of Object-Oriented Design in UML by Page-Jones, M (2000). Chapters 4 & 12.



SLIIT

Discover Your Future

Object Oriented Concepts

Lecture-12

Implementation of Inheritance using
C++

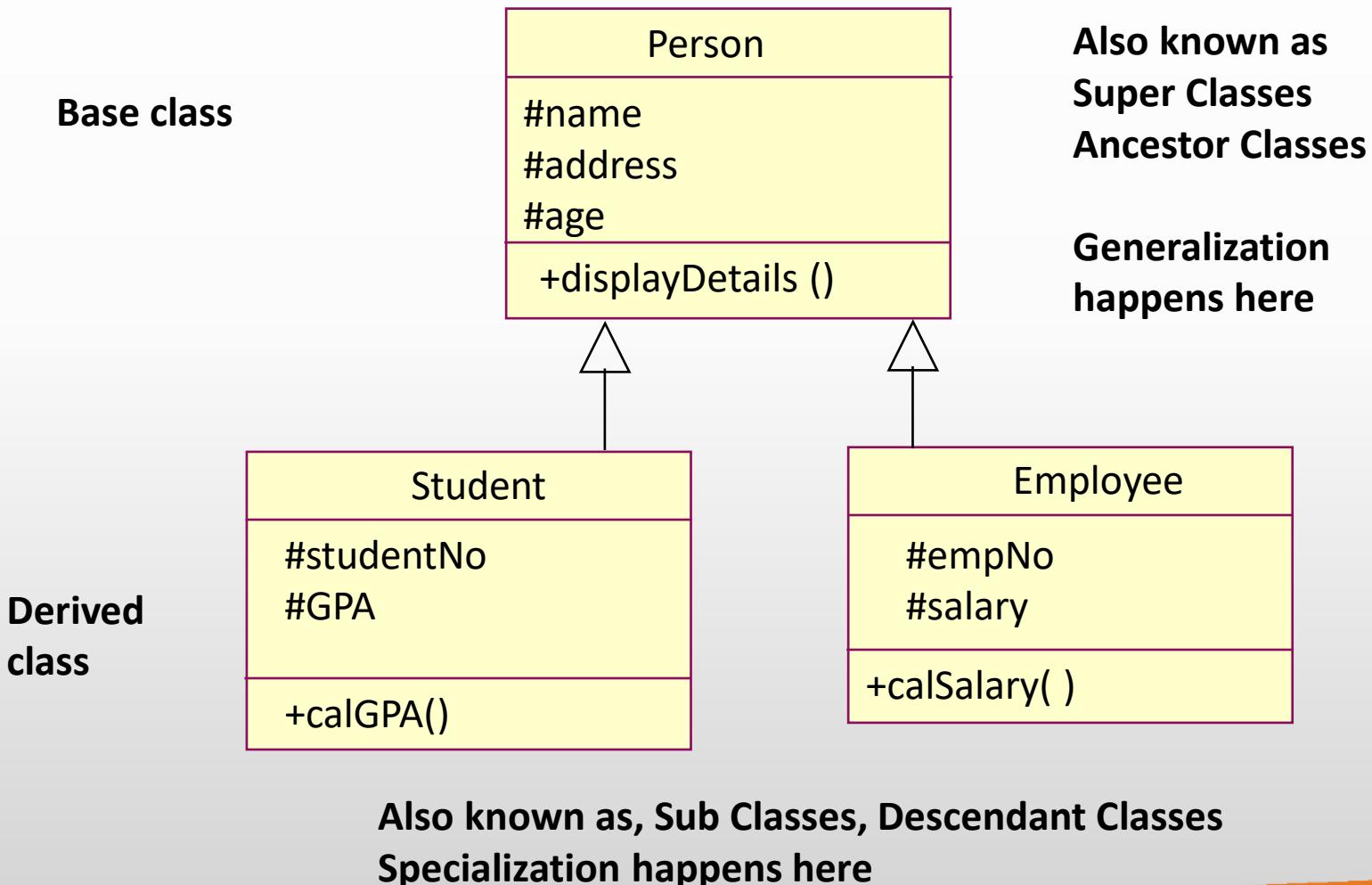
Learning Outcomes

- At the end of the lecture, students should be able to
 - Implement Inheritance, virtual functions
 - Understand and apply polymorphism

Inheritance

- Inheritance is the process by which one object can acquire the properties of another object.
- Instead of writing completely new data members and member functions, you can specify that the new class should inherit the members of existing class.
- The existing class is called the base class and the new class is called the derived class.

Base class and derived class



C++ code

Derived Classes

Base Class

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

class Person{
protected :
    char name[20];
    char address[20];
    int age;
public:
    Person() {};
    void display() {
        cout << "this is person class" << endl;
    }
    void displayDetails();
};

}
```

```
class Student : public Person{
protected :
    int studentNo;
    double GPA;
public:
    Student() {};
    void display() {
        cout << "this is student class. derived class
from person" << endl;
    }
    void calGPA();
};

class Employee : public Person{
protected :
    int empNo;
    double salary;
public:
    Employee() {};
    void display() {
        cout << "this is employee class.
Derived class from person" << endl;
    }
    void calSalary();
};

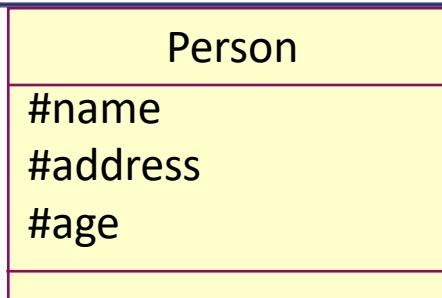
};
```

Access Rules

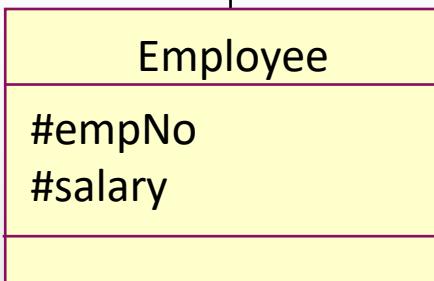
	Public +	Protected #	Private +
Same class	Yes	Yes	Yes
Derived class	Access	Yes	No
Outside class	Yes	No	No

Multilevel Inheritance

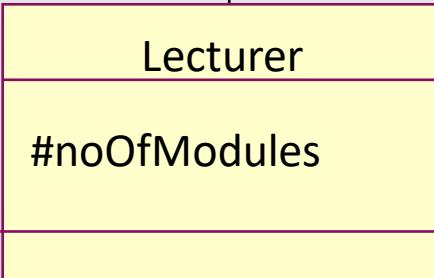
Base class



Intermediate Class



Derived class



C++ code

Derived Classes

Base Class

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

class Person{
protected :
    char name[20];
    char address[20];
    int age;
public:
    Person(){};
    void display() {
        cout << "this is person class" << endl;
    }
    void displayDetails();
};

class Employee : public Person{

protected :
    int empNo;
    double salary;
public:
    Employee() {};
    void display() {
        cout << "this is employee class.
                Derived class from person" << endl; }

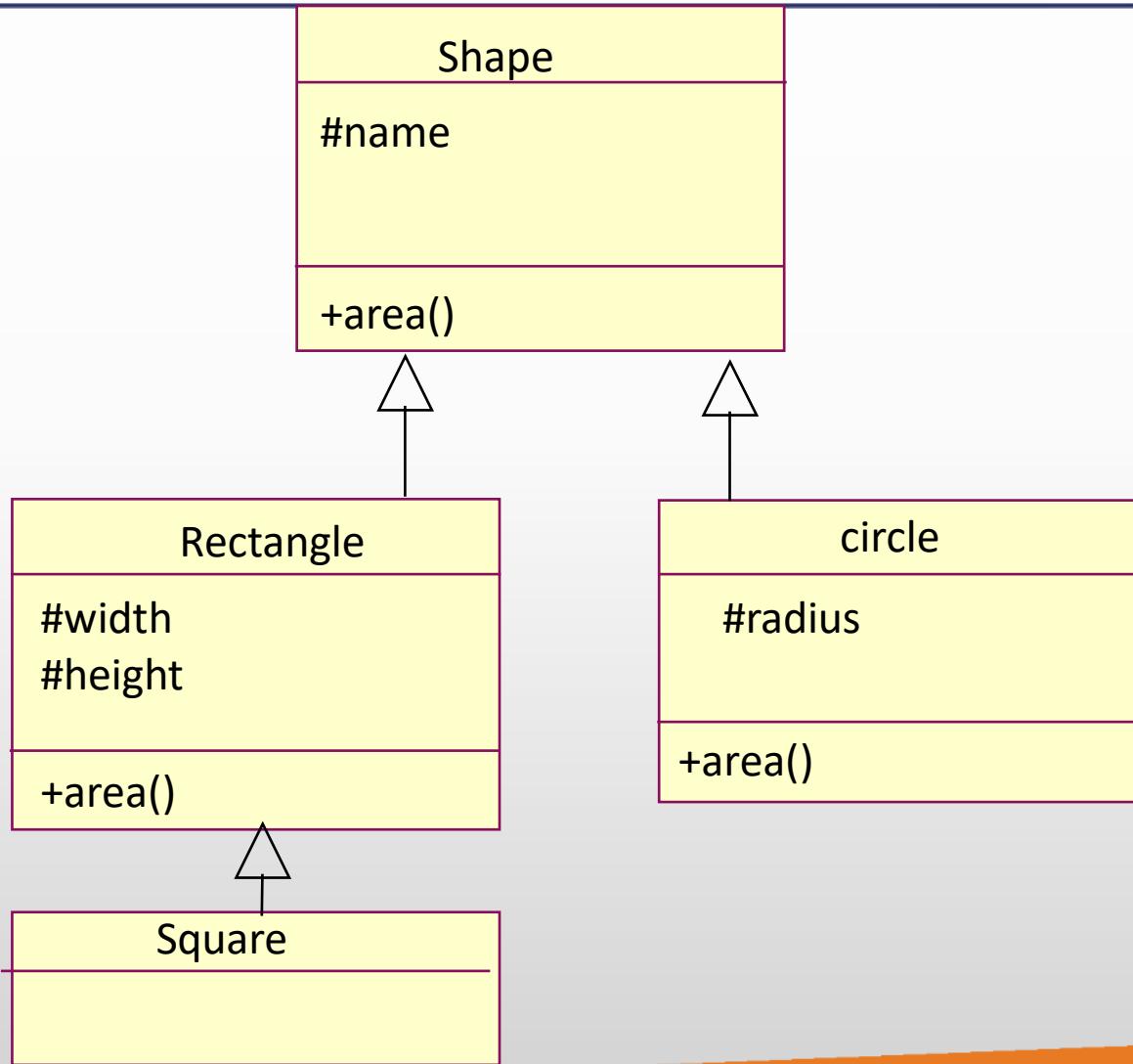
    void calSalary();
};

class Lecturer: public Employee{

protected :
    int noOfModules;
public :
    Lecturer( ) {};
    void display(){
        cout << "this is employee class.
                Derived class from person" << endl;
    }
};
```

Sample Code

Handling constructors



Default Constructors

```
class Shape
{
protected:
    char name[20];
public:
    Shape() {
        cout << "Shape
called" << endl
    }
};
```

```
class Rectangle : public Shape
{
protected:
    int length, width;
public:
    Rectangle() {
        cout << "Rectangle
called" << endl
    }
};
```

The compiler will insert code in the derived class (Rectangle) class constructor to call the Shape class constructor. This is the first instruction that will be executed in the Rectangle class constructor.

Rectangle rec; // will produce the following output

Shape Called

Rectangle Called

Sample Code

Overloaded Constructors

Base class

```
class Shape
{
    protected:
        char name[20];
    public:
        Shape();
        Shape ( char tname[])
    {
        name = tname;
    }
};
```

Derived class

```
class Rectangle: public Shape{
    protected:
        int length;
        int width;
    public:
        Rectangle (char tname[] ,
                    int l, int w) : Shape ( tname)
    {
        length = l;
        width = w;
    }
};
```

Sample Code

Overriding methods

Base class

```
class Shape
{
    protected:
        char name[20];
    public:
        Shape();
        ...
        int area() {
            return 0;
        }
    };
}
```

Derived class

```
class Rectangle: public Shape{
    protected:
        int length;
        int width;
    public :
        Rectangle();
        ...
        int area() {
            return length * width;
        }
    };
}
```

Sample Code

Overriding

- In the previous example we saw that the Rectangle class redefines the area method()

```
int area()
```

- This definition is exactly the same as in the Shape class. This is called overriding.
- If a sub class has a different behavior of a method we can override it and redefine the code associated.
- Please Note that in overloading there is a difference in parameters and overloading usually happens within the same class.

Overriding

- Consider the following code

```
Shape sh("myshape");
Rectangle rec("whiteboard", 10,5);
cout << sh.area(); // will produce 0, Shape::area() called
cout << rec.area(); // will produce 50, Rectangle::area() called
```

Situations where overriding doesn't work properly in C++ and how to fix it

```
Shape *sh;
```

```
sh = new Rectangle("whiteboard", 10, 5);
```

```
cout << sh->area();
```

- // produces 0, Shape::area() called
- In the above code a Rectangle type object is created in the second line, but when we run the code the Shape classes area() function is called

Virtual functions as a fix

- We can define the function that we are overriding as a virtual function. This enables dynamic binding where the overridden methods are called correctly at runtime.
- A virtual function is a member function that is declared with in the base class and redefined by a derived class.
- The function in the base class must precede the keyword **virtual**
- **Virtual functions** support dynamic polymorphism
- e.g. we have to add the word virtual to the area method of the Shape class and everything will work properly.

```
virtual int area() {  
    return 0;  
}
```

Sample Code

Polymorphism

- Greek meaning “*having multiple forms*”
- Ability to assign a different meaning or usage to something in different contexts
- Specifically, to allow an entity such as a variable, a function, or an object to have more than one form
- Overriding is a type of polymorphism, here we generally expect dynamic binding to work as well (use of virtual functions)

An example

- Consider the request (analogues to a method)
“please cut this in half” taking many forms



For a cake:

- Use a knife
- Apply gentle pressure

For a cloth:

- Use a pair of scissors
- Move fingers in a cutting motion

“please cut this in half”

- Imagine this task is automated...
- Without polymorphism
 - Need to tell the computer how to proceed for each situation
- With polymorphism
 - Just tell *please cut this in half*
 - The computer will handle the rest!

Polymorphism

- Polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.
- C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Abstract Classes

- In some situations we want to prevent the creation of objects of a given class. e.g. We may want to restrict the creation of Shape type objects.
- In C++ we can create an abstract class by including at least one pure virtual method
- We can make a method a pure virtual method by assigning zero to the virtual method.
- e.g. in the Shape Class we can do the following

virtual int area() = 0;

- You cannot declare an instance (object) of an abstract base class; you can use it only as a base class when declaring other classes.

Abstract Classes

Base class

```
class Shape
{
    protected:
        char name[20];
    public:
        Shape();
        ...
    virtual int area() = 0;
};
```

Derived class

```
class Rectangle: public Shape{
    protected:
        int length;
        int width;
    public :
        Rectangle();
        ...
    int area() {
        return length * width;
    }
};
```

Now the following code will generate a compilation error
Shape myshape;

However we can do the following

Shape *shp;
Shp = new Rectangle();

Example – Polymorphism

```
class Animal {  
  
protected:  
  
    char name[20];  
  
public:  
  
    Animal() {}  
  
    Animal(char tname[]) {  
  
        strcpy(name, tname);  
  
    }  
  
    virtual void speak() {}  
  
    void song() {  
  
        cout << name << "'s Song " << endl;  
  
        speak();  
  
        cout << "la la la la" << endl;  
  
        speak();  
  
        cout << "la la la la" << endl;  
  
        speak();  
  
    }  
};
```

Sample Code

Animal Example

```
class Cat : public Animal {  
public:  
    Cat() {}  
    Cat(char tname[]) : Animal(tname) {}  
    void speak() {  
        cout << "Meow... Meow..." << endl;  
    }  
};  
  
class Dog : public Animal {  
public:  
    Dog() {}  
    Dog(char tname[]): Animal(tname) {}  
    void speak() {  
        cout << "Bow... Bow..." << endl;  
    }  
};
```

Animal Example

```
class Cow: public Animal {  
public:  
    Cow() {}  
    Cow(char tname[]) : Animal(tname) {}  
    void speak() {  
        cout << "Moo... Moo..." << endl;  
    }  
};  
int main()  
{  
    Animal *ani[4];  
    ani[0] = new Cat("Micky the Cat");  
    ani[1] = new Dog("Rover the Dog");  
    ani[2] = new Cow("roo the Cow");  
    ani[3] = new Animal("no name");  
    for (int r=0;r<4; r++)  
        ani[r]->sing();  
  
    char ch;  
    cin >> ch;  
    return 0;  
}
```

IT1050- Object Oriented Concepts

Lecture-13
Implementation of relationships
among classes using C++

Learning Outcomes

- At the end of the lecture, students should be able to
 - Implement Composition, Aggregation, Association and Dependency

Composition



- Whole : University
- Part : Room
- A University is composed of at least one Room,
- If there are no rooms, there is no university
- Implies that the “part cannot exist without the whole”

Composition

- In composition, the objects have coincident lifetimes.
- if parent (whole) object gets deleted, then all of it's child (part) objects will also be deleted.
- If the University object is deleted, the class room objects will get deleted automatically.
- Child (part) objects are created in the parent (whole) class.

Composition – C++ Implementation

```
class ClassRoom {  
private:  
    int roomno;  
public:  
    ClassRoom() {};  
    ClassRoom(int no) {  
        roomno = no;  
    };  
    void Display() {  
        cout << "Class Room " << roomno << endl;  
    };  
    ~ClassRoom() {  
        cout << "Deleting Room " << roomno << endl;  
    }  
};
```

Sample Code

Composition – C++ Implementation

```
class University {  
    private:  
        ClassRoom *room[SIZE];  
    public:  
        University() {  
            room[0] = new ClassRoom(101);  
            room[1] = new ClassRoom(102);  
        };  
        University(int no1, int no2) {  
            room[0] = new ClassRoom(no1);  
            room[1] = new ClassRoom(no2);  
        };  
        void DisplayClassRooms() {  
            for (int i=0; i<SIZE; i++)  
                room[i]->Display();  
        };  
        ~University() {cout << "Univesity shutting down" << endl;  
        for (int i=0; i <SIZE; i++)  
            delete room[i];  
        cout << "the End" << endl;  
    };
```

Composition – C++ Implementation

```
int main()
{
    University *myUniversity;
    myUniversity. = new University(501, 502);
    myUniversity >DisplayClassRooms();

    return 0;
}
```

Class room objects are created inside the university class and when the University destructor is called all the class room objects are deleted.

Output

Class Room 501

Class Room 502

University shutting down

Deleting Room 501

Deleting Room 502

the End

Aggregation



- Whole : Department
- Part : Employee
- A Department has one or more employees
- This implies that the **Part can exist without the Whole.**

Aggregation

- In aggregation the objects have their own life cycles, but there is a ownership.
- The Department and Employee objects have their own life cycles.
- If the Department object is deleted, still the Employee objects can exist.
- If the Employee objects is deleted, still the Department object can exist.

Aggregation – C++ implementation

```
class Employee
{
private :
    string empID;
    string name;
public :
    Employee(string pempID, string pname)
    {
        empID = pempID;
        name = pname;
    }
    void displayEmployee()
    {
        cout << "empID = " << empID << endl;
        cout << "name = " << name << endl;
        cout << "*****" << endl;
    }
    ~Employee(){cout << "Deleting Employee" << empID << endl;
    }
};
```

Sample Code

Aggregation - C++ implementation

```
class Department
{
private:
    Employee *emp[2];
public:
    Department() {};
    void addEmployee(Employee *emp1, Employee *emp2)
    {
        emp[0] = emp1;
        emp[1] = emp2;
    }
    void displayDepartment()
    {
        for(int i = 0; i < SIZE; i++)
            emp[i]->displayEmployee();
    }
}
```

```
~Company() {cout << "Department shutting down" << endl; }
```

Aggregation - C++ implementation

```
int main()
{
    Department*ABC = new Department();
    Employee *e1 = new Employee("E001",
"Nimal");
    Employee *e2 = new Employee("E002",
"Jagath");
    ABC->addEmployee(e1, e2);
    ABC->displayDepartment();
    delete ABC;
    e1->displayEmployee();
    e2->displayEmployee();
    return 0;
}
```

After the company ABC is deleted the two employees exists.

Output

```
empID = E001
name = Nimal
*****
empID = E002
name = Jagath
*****
Company shutting down
empID = E001
name = Nimal
*****
empID = E002
name = Jagath
*****
```

Aggregation - C++ implementation

```
int main(){
    Department*ABC = new Department();
    Employee *e1 = new Employee("E001", "Nimal");

    Employee *e2 = new Employee("E002", "Jagath");

    ABC->addEmployee(e1, e2);
    delete e1;
    delete e2;

    Employee *e3 = new Employee("E003", "Kamal");

    Employee *e4 = new Employee("E004", "Lal");
    ABC->addEmployee(e3, e4);
    ABC->displayDepartment();

    return 0;
}
```

After E001 and E002 is deleted still the company exist and new employees can be added.

Output

Deleting EmployeeE001

Deleting EmployeeE002

emplID = E003

name = Kamal

emplID = E004

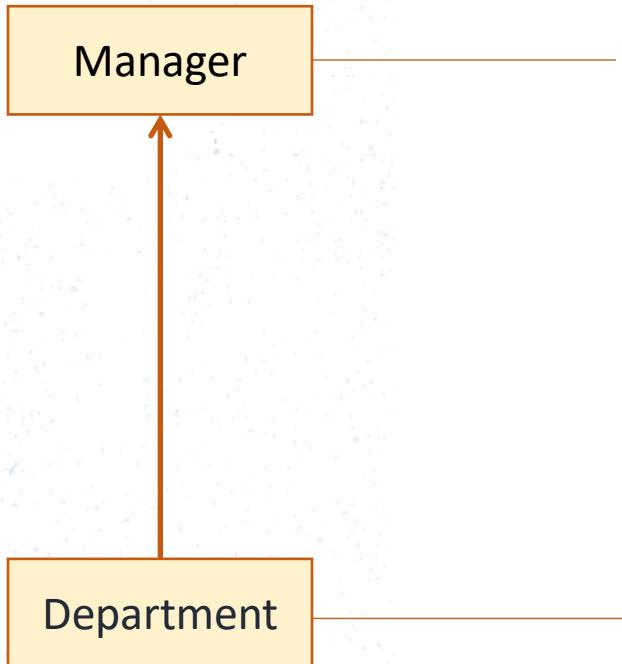
name = Lal

Association

- An association between two classes indicates that objects at one end of an association “recognize” objects at the other end and may send messages to them.
- Example: “A Customer has many Orders”



Uni-directional association



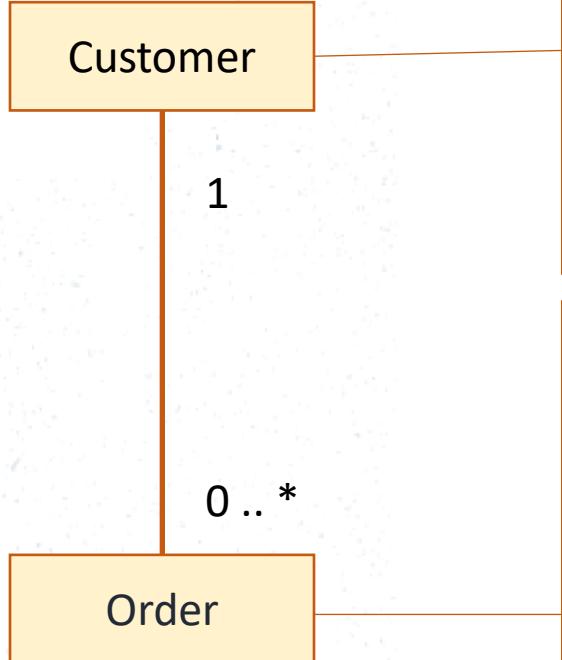
```
class Manager
{
public:
    Manager();
    ~Manager();
};
```

```
# include "Manager.h"
class Department
{
private:
    Manager* mgr;

public:
    Department();
    ~Department();
};
```

Sample Code

Bi-directional association

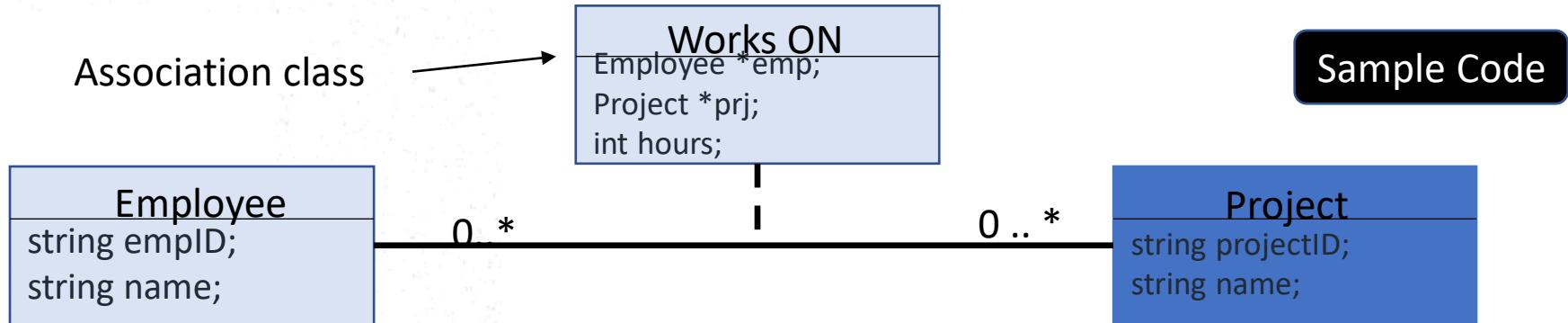


```
class Customer
{
    private:
        string name;
        string address;
        Order *order[SIZE];
        int noOfOrders;
    public:
        Customer();
        Customer( string pname, string
paddress );
        void addOrder(Order *O);
        void displayCustomer();
};
```

```
class Order
{
    private:
        string orderID;
        Customer *Cus;
    public:
        Order (string porderID, Customer
*pCus);
        void displayOrders();
};
```

Sample Code

Association class



- An association class is a class that is part of an association relationship between two other classes.
- An association class provides additional information about the relationship.
- If an employee works for more than one project and if each project is assigned to more than one employee , the additional information (number of hours each employee spend on the a project) can be store in Works ON class.

Dependency

- Dependency is a weaker form of relationship which indicates that one class depends on another because it uses it at some point in time.
- It implies that **a change to one class may affect the other but not vice versa.**

Dependency



- The Sales Person class depends on a Product class because the Product class is used as a parameter for an add operation in the Sales Person class.

```
void SalesPerson::addSales(int qty , Product *P)
{
    salesAmount = qty * P->getPrice();
}
```

Dependency

```
class Product
{
    private:
        string productID;
        string name;
        double price;
    public:
        Product(){}
        Product(string pID, string pname,double pPrice){
            productID = pID;
            name = pname;
            price = pPrice;
        }
        float getPrice(){
            return price;
        }
        void display()
        {
            cout << " Product ID =" << productID << endl;
            cout << " Product name =" << name << endl;
            cout << " Price = " << price << endl;
        }
};
```

Dependency

```
class SalesPerson
{
    private:
        string name;
        double salesAmount;
public:
    SalesPerson(string pname) {
        name = pname;
        salesAmount = 0;
    }
    void addSales(int qty , Product *P){
        salesAmount = qty * P->getPrice();
    }
    void display()
    {
        cout << "name = " << name << endl;
        cout << "Sales Amount = " << salesAmount << endl;
    }
};
```

Dependency

```
int main()
{
    Product *P1 = new Product("P001", "Mugs" , 200.00);
    SalesPerson *SP = new SalesPerson("Ajith");
    SP->addSales(10, P1);
    SP->display();
}
```