

Programming II Notes

Nigel Fernandez (IMT2013027)

Jan 2, 2014

1. Executable creation cycle
 - Source code in HLL
 - Intermediate compile to assembly code
 - Final compile to object code / machine language
 - Linking object files to executable
2. cc command does a combined compilation and linking step
3. Compilation phases: pre-processing → parsing → code generation
4. If an object file (binary) does not contain external references or binary dependencies to other binaries then it can directly be converted to an executable skipping the linking step
5. Compiler error if linking is not done: undefined reference to symbol
6. Library: collection of object files
7. Creation of final executable will include the linking of object files of the entire referenced library even if only a single object file of the library is referenced
8. Strip command will remove unused symbols
9. #include is part of the pre-processing step (not linking)
10. Handling of #include by pre-processor: looks for the directive delimiter → comments out directive → in place insertion of required code
11. Header files: collection of function prototypes
12. Function prototypes help the compiler to parse the function signature and flag errors in function calls
13. Functions like printf, scanf, etc are not part of the programming language but part of the libraries of external function provided to the language
14. #define is equivalent to string substitution
15. The original C compiler was written by Brian Kernighan and Dennis Ritchie (K&R) with the intention of not flagging errors to the user but simply converting correct code to correct output. Later ANSI C was introduced with additional syntax rules.
16. Process creation / Code-compile-link-load cycle:

Program (compiled and linked) -> executable (stored on disk) -> process (loaded by loader onto RAM)

17. Loading: Binding of variable to memory locations

18. Static binding: information determined during compile time

Ex – `int x` or `printf(“%d”, 10)`

19. Dynamic binding: information determined during run time

Ex – `int *ptr;`

`ptr = (int *)malloc(sizeof(int));`

20. Functions can only use static binding. Workaround: use function pointers

21. Executable bloat: slows down loading. Solution: strip command or dynamic loading.

22. Dynamic loading:

- Implemented by pointers to functions and libraries in hard disk which are loaded when needed
- Ex: Windows – DLL
- Reduces size of executable but may result in errors if the executable is moved from one platform to another
- Causes slow start-up of executable (loading DLL files) but later speeds up when object files are in memory

23. Programming languages:

- Imperative (mainly composed of sequence, conditional and iterative statements)
 - Structural (composed of subroutines) (C, Pascal, Fortran, COBOL)
 - Object oriented (C++, Java, Python)
- Functional (permutation of appropriate function calls) (LISP, Scheme)
- Logic (Uses set of relations and rules for output) (Prolog)

24. Structural paradigm: functions exchange data

25. OOP paradigm: objects (data) exchange messages (function calls)

Note – For more information please see my post on ‘Experiments of Jan 2’.

C++ Programming Class notes - Day 1

P.S.Srinivasan

IMT2013028

January 2, 2015

$\boxed{SourceCode} (Cant-be-understood-by-the-machine) \rightarrow (Compiling) \rightarrow$

$\boxed{Machine-Code \Leftrightarrow Object-Code} \rightarrow (Linking) \rightarrow \boxed{Executable}$

- The "gcc hello.c ..." is not a compile alone command. It is compile + link.
- It compiles files individually and links everything together.
- This Compiling has three parts
 - Parsing
 - Lexical Analysis
 - Code generation
- A program can be done without linking if every instruction is dumped in a single file.
- This "Udefined reference to symbol" comes from the linker \rightarrow Assume p2.0 is a file with the definition of the function f1. Suppose another file

p1.o calls the function f1 and the linker does a pretty bad job then, it throws out the error.

- $p1.o \downarrow p2.o \downarrow \underbrace{printf.o \downarrow scanf.o}_{\text{Executable}} \downarrow$

- Binaries of external authors (say printf.o) must also be linked. These object files are grouped as 'libraries'.
- The libraries are linked as entire files and not as individual functions.
- *#include* \neq *linking* " # include ..." is just to convey a message to the compiler for checking the functions calls whether the parameters have been passed properly so that the compiler can give intelligent errors. The pre-processing step just replaces the " # include ..." lines with all the function prototypes defined in the .h file as per the latest ASCII std. So theoretically speaking you should be able to compile without header files.
- Set of instructions to achieve a task, if resides in hard disk is called a program and if the same is in the RAM it is called a process.
- Source Code
↓
Compile and Link
↓
Hard Disk \Leftarrow Executable
↓
Loader
↓
RAM \Leftarrow Process

- Loading is nothing but binding of a variable to a memory location, There are two types of binding:
 - Static Binding: Things known at compile time. ex saying `int x = 4;`
 - Dynamic Binding: Things known only during the runtime of the program. ex `int* m; m = (int *) malloc(sizeof(int));`
 - Function pointers are also dynamically bound.
- Executable bloat: Exceptional growth in the size of an executable due to linking of large number of files. There are two ways around it
 - Removing unused symbols
 - Dynamic Loading: Say you are linking three files `p1.o` `p2.o` and `libx.o`. Now what dynamic loading does is it first created the executable without `libx` being actually loaded but just loads a pointer to the library file. So if `libx` is needed, it is dynamically loaded. This might increase the runtime performance but the size reduces. These dynamically loaded libraries are called as DLL.
- What is the major difference between object oriented programming and Structured programming? In structured programming the functions or the methods remain static and the data moves around thereby creating a flow whereas in object oriented programming, datas and functions are bound in a blackbox and is sitting untill they are triggered by a message which is usually a function call. These messages are moved around thereby creating the flow.

2nd Jan Notes

- Rajesh D M(IMT2013032)

Any source code – Written in High level language

This needs to be converted to machine code (object code) for the machine to be able to process it. This is done using the compiler.

The object code is then converted to an executable.

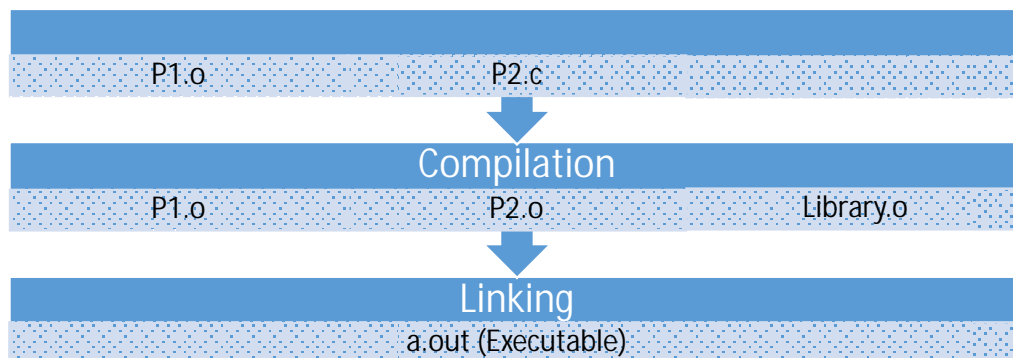
During Ordinary compilation – Ex: The command “cc hello.c” does both compiling and linking.

When only compilation is done – The .o file is generated (ex: hello.o – by running “cc -c hello.c”). This file can run by the machine if it does not require any resources from other files. But if it does, then the error undefined reference occurs -> it is the job of the linker to link all the resources required for running one file into one executable.

Only the .o file of the external resources required – source code not required.

If we have only the .o file of a resource – it is known as a library.

Usually while linking – the linker includes the whole library from where we are using a particular function or a few functions.



Pre – Processors

The line begins with a '#' and the keyword that follows is known as pre-processor directive.

Ex:

1. `#include<stdio.h>` simply replaces the code in the file `stdio.h` into the file where the statement `#include<stdio.h>` is present and the `#include` statement is commented.

2. `#define max 10` – This replaces all instances of `max` in the file by `10` present and the `#include` statement is commented

All this happens when the compilation begins (before the processing starts).

This can be viewed in the `.i` file.

Running an Executable:

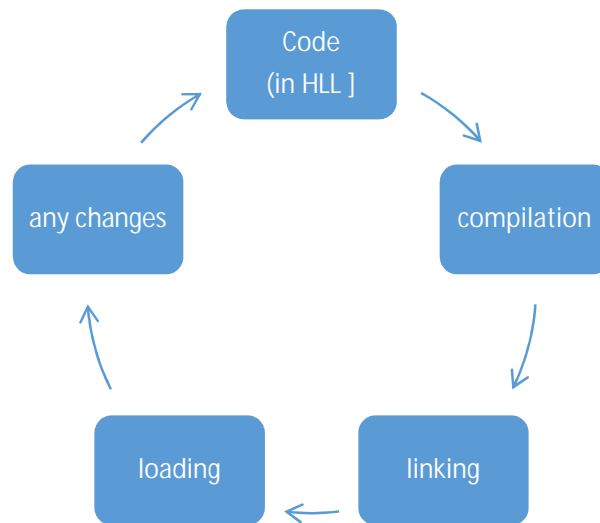
A program can't be run if it is in the disk – it has to be in the RAM.

A program is called a process when it is running.

The loader loads the executable from the disk on the RAM.

Any changes made to the code will not affect the running of the program until it is recompiled and reloaded into the RAM.

The whole process of running a program –



Loading: Binding of a variable to a memory location.

There are 2 types of binding:

1. Static Binding – is a binding which happens during compilation. It is also called early binding because binding happens before a program actually runs.
2. Dynamic Binding – is a binding which happens during run-time. It is also called late binding because it happens when the program is actually running.

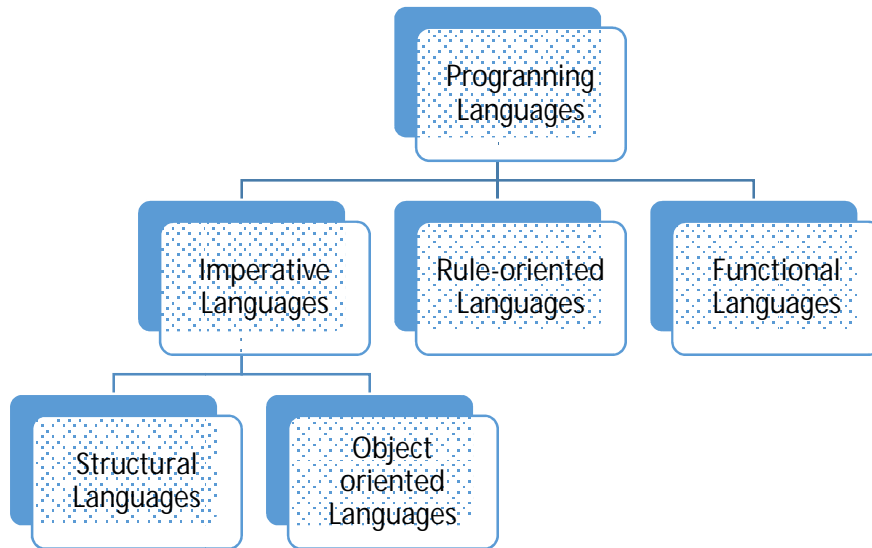
Executable Bloat: This occurs when the size of the executable becomes very large due to the inclusion of large number of big external libraries.

The main disadvantage of executable bloat is that the loading becomes very slow and the process becomes very slow.

There are 2 ways to overcome this –

1. Stripping the executable – Whenever the linker links all the files, it brings in all the information from the files which are being linked. Stripping the executable removes that extra code from the external files which are not being used.
2. Dynamic Loading – Linking is not done completely (especially libraries). There is only a pointer to these unlinked files. They are loaded into the RAM when the process is running and needs the resources from that particular library.
Disadvantage of dynamic loading: When the executable is run other machines and libraries are not transferred with the executable, and during the running of the executable if that particular library is required, execution terminates.

Programming Languages



Structural Programming Languages

Focuses on dividing the process into sub-routines and then repeatedly using them. Data is transferred between sub-routines.

Ex: C, PASCAL, FORTRAN

Object oriented languages

Focusses on the relationship between real world objects. Functions are used to interact between objects.

Ex: C++, Java, Python

2nd Jan Notes

- Rajesh D M(IMT2013032)

Any source code – Written in High level language

This needs to be converted to machine code (object code) for the machine to be able to process it. This is done using the compiler.

The object code is then converted to an executable.

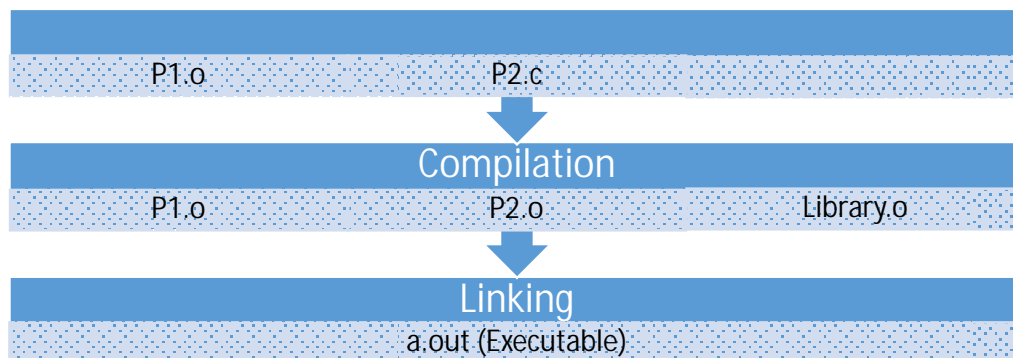
During Ordinary compilation – Ex: The command “cc hello.c” does both compiling and linking.

When only compilation is done – The .o file is generated (ex: hello.o – by running “cc -c hello.c”). This file can run by the machine if it does not require any resources from other files. But if it does, then the error undefined reference occurs -> it is the job of the linker to link all the resources required for running one file into one executable.

Only the .o file of the external resources required – source code not required.

If we have only the .o file of a resource – it is known as a library.

Usually while linking – the linker includes the whole library from where we are using a particular function or a few functions.



Pre – Processors

The line begins with a '#' and the keyword that follows is known as pre-processor directive.

Ex:

1. `#include<stdio.h>` simply replaces the code in the file `stdio.h` into the file where the statement `#include<stdio.h>` is present and the `#include` statement is commented.

2. `#define max 10` – This replaces all instances of `max` in the file by `10` present and the `#include` statement is commented

All this happens when the compilation begins (before the processing starts).

This can be viewed in the `.i` file.

Running an Executable:

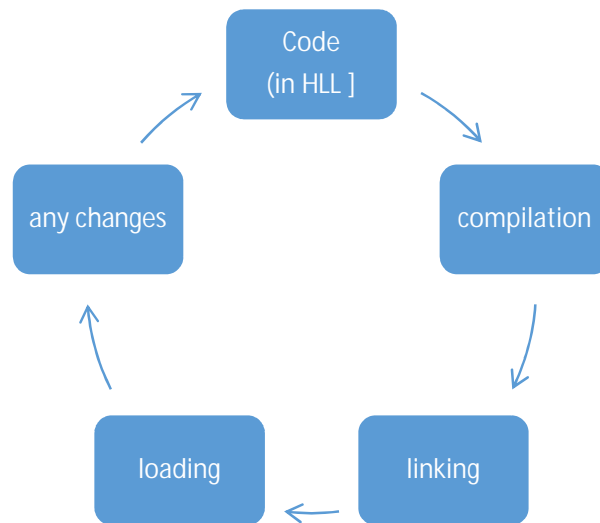
A program can't be run if it is in the disk – it has to be in the RAM.

A program is called a process when it is running.

The loader loads the executable from the disk on the RAM.

Any changes made to the code will not affect the running of the program until it is recompiled and reloaded into the RAM.

The whole process of running a program –



Loading: Binding of a variable to a memory location.

There are 2 types of binding:

1. Static Binding – is a binding which happens during compilation. It is also called early binding because binding happens before a program actually runs.
2. Dynamic Binding – is a binding which happens during run-time. It is also called late binding because it happens when the program is actually running.

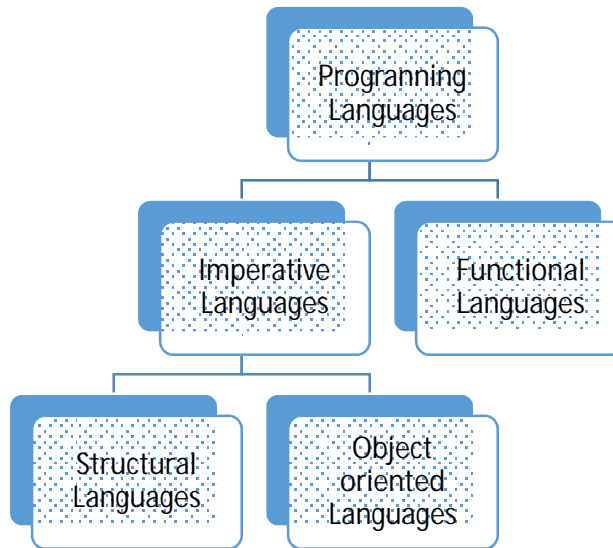
Executable Bloat: This occurs when the size of the executable becomes very large due to the inclusion of large number of big external libraries.

The main disadvantage of executable bloat is that the loading becomes very slow and the process becomes very slow.

There are 2 ways to overcome this –

1. Stripping the executable – Whenever the linker links all the files, it brings in all the information from the files which are being linked. Stripping the executable removes that extra code from the external files which are not being used.
2. Dynamic Loading – Linking is not done completely (especially libraries). There is only a pointer to these unlinked files. They are loaded into the RAM when the process is running and needs the resources from that particular library.
Disadvantage of dynamic loading: When the executable is run other machines and libraries are not transferred with the executable, and during the running of the executable if that particular library is required, execution terminates.

Programming Languages



Structural Programming Languages

Focuses on dividing the process into sub-routines and then repeatedly using them. Data is transferred between sub-routines.

Object oriented languages

Focusses on the relationship between real world objects. Functions are used to interact between objects.

Programming Languages – 02/01/15

- Way of communicating with the computer through instructions
- Source code → High Level Language → Compiled to object code → Linked to executable
- Compilation:
 - Typical applications have multiple source files
 - Compilation produces *.o file (object/machine code)
 - Compilation = Preprocessing + Parsing + Code generation
 - Object files are brought together by the linker
 - Linking is required when the binary corresponding to a function call resides external to the binary from where the function was called
 - Otherwise 'Undefined reference to symbol “___” ' error is generated by the linker
 - Binaries of external authors (say printf.o) must also be linked
 - These object files are grouped as 'libraries'
 - The corresponding machine code is generally put in a single binary
 - Binary file size > object file size (approx 7 times for hello world program)
- Preprocessor:
 - '#' designates the start of a preprocessor
 - 'include' is a directive to the processor to read the contents of the header file specified in <*.h> and replace the line (comment it out) with the contents of the entire header file
 - This can be viewed in the *.i file
 - #include is NOT the same as linking; It is a preprocessor directive
 - 'define' is another preprocessor directive that does string substitution (Ex. #define MAX 10)
- Header files:
 - Contains only function prototypes
 - Informs compiler of function signatures to aid compile time checking of function calls against function signatures
 - 'printf' is part of the library used by C, NOT part of the language
 - Kernighan & Ritchie (K&R) mode of compilation follows 'right code → right output' else gives 'segmentation fault core dumped'
 - To enforce stricter rules, ANSI C standard came about which enforced specification of function prototypes
 - Programs can be linked without specifying the preprocessor directives by invoking the K&R mode of compilation from command line
- Loader:
 - Running program = process
 - Program becomes a process as it moves from disk to RAM
 - While running a program, loader is invoked to transfer from disk to RAM
 - 'Code – Compile – Link – Load' is the cycle followed from code to execution
 - Loader binds variables to memory location
 - Two types – static and dynamic
 - Static – 'Memory location to which a symbol (variable/function) is bound' is determined at COMPILE TIME
 - Dynamic – 'Memory location to which a symbol (variable/function) is bound' is

determined during RUN TIME.

- Ex. `int x;` versus `int *x;`
 - First case – static binding; memory address for variable x is known
 - Second case – memory allocation is not known until '`malloc()`' is EXECUTED
 - In C programs all functions MUST be statically bound UNLESS they are specified as function pointers
 - By linking object files into a single executable there is high chance of occurrence of a situation termed as 'execution bloat'
 - Can be overcome by:
 - Removing unused symbols – 'stripping'
 - Use dynamic loading technique
 - Dynamic loading technique uses pointers to the libraries
 - Whenever required, the pointer is used to dynamically load the library into memory
 - Dynamic libraries are also called shared libraries (Ex. Windows DLL)
 - Since all libraries are not linked into a single executable, portability of the executable is affected (the dynamic libraries must be shifted too)
- Types of programming languages
 - Imperative – C, C++, Pascal
 - Functional – LISP, Scheme
 - Logic – PROLOG
 - Imperative languages have assignments, conditions and iterations as their constructs
 - Functional languages view everything as functions (including assignments iterations etc)
 - Imperative languages are of two types:
 - Structured – C, Pascal, FORTRAN, COBOL
 - Object Oriented – C++, Java
 - Structured languages follow the paradigm of data passing between functions
 - Object Oriented languages follow the reverse paradigm of functions passing between data

Object Oriented Programming:



Structured Programming [set of co-operating sub-routines]



Object Oriented Programming [set of co-operating objects which get things done]

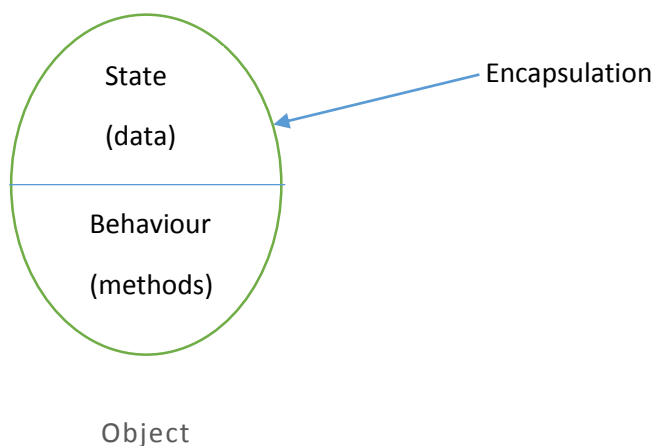
Object:

Object is an **encapsulation** of **state** and **behaviour**.

Function:

Function is an encapsulation of **behaviour** which takes some inputs and gives output.

[Behaviour is some kind of a task that is happening].



Encapsulation [Information Hiding]:

Helps in hiding the internal working with data. [Basically a bunch of things nicely covered up and easy to use. Client doesn't need to know what is going on inside it].

Methods:

Service provided by an object is called a method.

Messaging is an act of invoking the method.

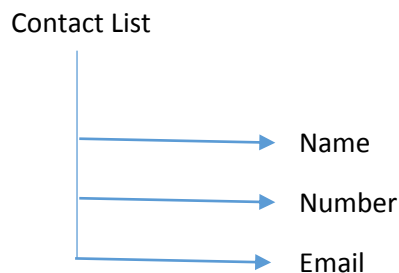
How is Object Oriented Programming approach more useful?

- 1) The Object already has the data that it needs and this data is pre-arranged in a way which is advantageous to the object for providing services to the client.
- 2) The impact of a change is more localized. The client doesn't have to do anything as the interface remains the same.

Abstraction:

Abstraction is creating a high-level view of certain low level-concepts.

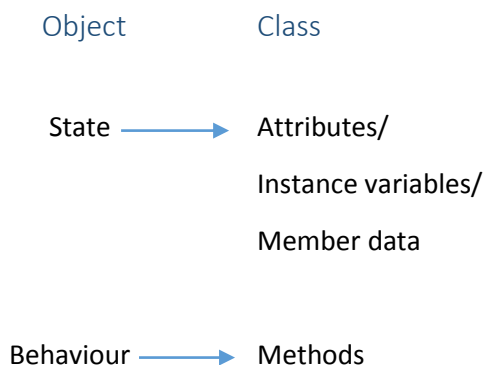
Eg. Contact list can be an abstraction of Name, number and e-mail.



Abstraction is achieved using complex objects.

Classes:

A class is a template for creating objects. It specifies the state and behaviour of every object.



You CANNOT create an object without a class.

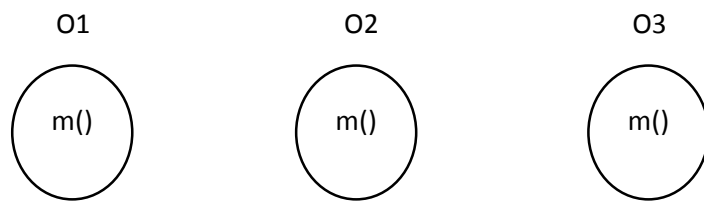
Constructor:

A constructor is a special method for initializing an object.

[Decides the state of the object initially to begin with].

Destructor:

A destructor is a special method for "cleaning up". This is the method where you perform any specific operation like free memory or write to file. It is called automatically at run-time for a dying object.



When invoking a method, you have to mention the object which is being referred to.

Eg we use O2.m() if we want to make changes to O2.

Object Oriented Design:

How to identify the classes for the given application?

- Typically, the classes are usually the nouns or the noun phrases in the set of requirements given for the application.
- This creates a confusion about these being classes or attributes.
- If it is non-atomic, it is invariably a class.
- If it is atomic, then usually it is an attributes.
- But if the amount of behaviour for this atomic data is large, then we make it a class.

Nouns/Noun Phrases → Classes/Attributes
Verbs → Methods

Egs. For verbs being made methods are:

Constructor → create

Destructor → destroy

Class names must be nouns and Singular.

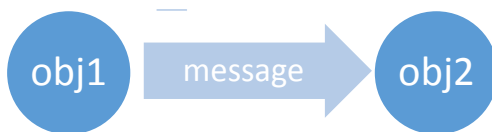
- - - x - - -

Structured Programming



A **function** is an **encapsulation** of **behavior** that takes some input and provides an output.

Object Oriented Programming



An **object** is an **encapsulation** in **state** and **behavior**.

State: It represents the data that is being stored in the object.

Behavior: Tasks to be done, driven by input output parameters.

Encapsulation: Binds the state and the behavior together into a single unit that cannot be modified by the external world.

The objects are inspired from real world objects with an interface to interact, without the need to know what's inside.

Interface: It is the point of contact for the user to the program. It showcases the encapsulated data to the user. The user can then invoke various methods/functions to process the data and get outputs without having to know how the data is being handled inside.

Eg. Suppose you have to go from place A to place B

Structured Programming: you have a car (data), you go to a driver, tell him where to go and he takes you there

Object Oriented Programming: you call a taxi service and book a taxi. The Taxi provides you with a car (data) and a driver to take you wherever you want to go.

The functions don't have parameters that take in the data as the data it has to operate upon in within the object itself.

The client doesn't have to bother as to how the data is being handled as long as the task is being completed.

In structured programming, if the functionality changes the client has to alter the way he handles the data. Whereas in object oriented programming the client need not worry about how the data is handled as long as the interface with which the client interacts with the program remains the same.

OOP enables encapsulation and abstraction.

Abstraction:

Creating a high level view with low level concepts. Eg contactList is an abstraction of contacts which itself is an abstraction of Name/Number/e-Mail.

In programming view the notion of classes was introduced to create objects.

A **Class** is a template for creating objects. The classes contain state and behavior.

State -> attributes, member data, data members, instance variables.

Behavior - > methods

A method is invoked in the context of an object. E.g. obj2.m() invokes the method m() of the object obj2 and uses the data in obj2 alone. It cannot alter the state of other objects unless explicitly allowed.

Special Methods:

Constructor

It is used to initialize the objects.

It is a method that is invoked when the object comes to life.

It has the same name as the class.

eg. contactList(struct contact *c, n)

```
{  
}
```

It is mandatory to have a constructor for every class.

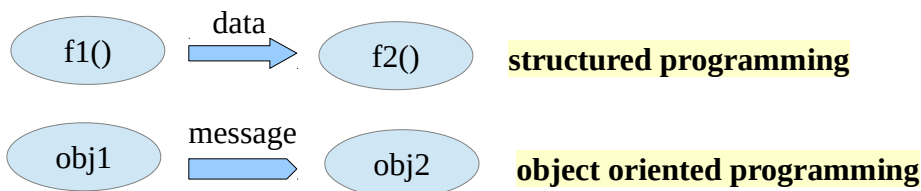
Destructor, a special method for cleaning up

When you are done with the object, you might want to perform some task like store the state of the object, delete memory, etc. All these tasks are specified within the destructor.

The destructor is automatically invoked before the object dies out.

OBJECT ORIENTED PROGRAMMING:

C++ is a better C. One of the reasons for this is that C++ supports Object Oriented Programming. Up until now, in structural programming, we have been using functions/subroutines to perform certain tasks and a program comprised of a set of coordinating functions/subroutines. However, let us take a slight paradigm shift. In object oriented programming, we view a software application as a set of coordinating **objects** to get things done. What are objects and why are they used?



OBJECT: An **object** is an **encapsulation** of **state** and **behaviour**. These three key words (in bold) define an object.

FUNCTION: A function is an **encapsulation** of behaviour which takes in some input and produces an output.

ENCAPSULATION: Encapsulation is essentially blackboxing. It prevents external clients from accessing internal objects. Encapsulation is synonymous to information hiding. It also enables the impact of any change to be localized. There is an interface between the client and the service provider so the client does not know what is happening internally.

Example. Let us take the simple example of driving a car. Calling a taxi service is analogous to encapsulation in object oriented programming. You as a passenger are not worried about driving or getting to the place. That is the job of the taxi driver. The taxi driver is not bothered about how the car is procured, etc. So the taxi service is sort of like a blackbox or a capsule whereas you as a passenger are the client which calls the taxi service when you need to.

ABSTRACTION: Abstraction is creating a high level view of a collection of low level objects. For example contacts is an abstraction of name, number, email address. You can have abstraction at multiple levels. Abstraction in object oriented programming is achieved using complex objects.

CLASS: A class is a template for creating objects. The template must specify state and behaviour.

State -> attributes/data members/instance variables

Behaviour -> methods

Note: The state and behaviour are associated with specific objects.

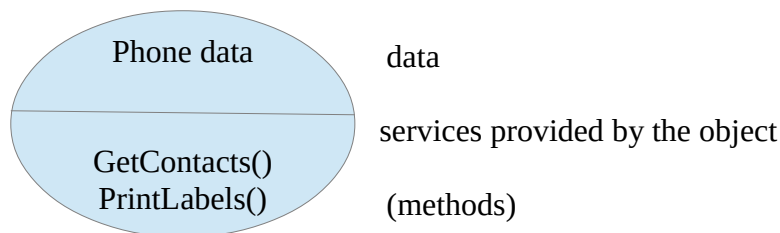
We can create multiple instances of a class. These instances are called objects. In C++, one CANNOT create an object without creating a class.

CONSTRUCTOR: A constructor is a very special method for initializing an object. Every class has a constructor. A constructor brings an object to life. It is responsible for taking data from a client and organizing this data using relevant data structures and initializes the objects.

DESTRUCTOR: A destructor is a very special method for 'cleaning up'. It is invoked just before an object dies. Constructors have external visibility whereas destructors do not.

WHAT ARE THE ADVANTAGES OF OBJECT ORIENTED PROGRAMMING?

Object oriented programming was designed to emulate the real world. In this model you have a client and a service provider. As a service provider, I have a better way to achieve the the behaviour because data is in my control. In order to pass a message, a method is invoked. It is entirely my call as the service provider as to how to organize the data (eg, data structures to use). This is decided by the type of services I would like to provide. For example, the inbuilt function `qsort()` only accepts arrays as the parameter. If you give a tree, it will not work. However, with object oriented programming, the client can give the data in the way he prefers and the job of converting it to the appropriate data structure is usually hidden from him or encapsulated. Object oriented programming essentially allows for outsourcing of a job to the service provider and the details of the implementation are hidden from the client because he doesnt care as long as his work is done.



*Functions need not have parameters because data is already present in the capsule

STEPS INVOLVED IN DESIGNING A PROGRAM USING OOP:

1. Identify classes for applications. Usually classes are nouns/noun phrases. Also, class names should be singular. However, a given noun can be a class or an attribute. How do we distinguish between them.

ATOMIC	NON - ATOMIC
Check for behaviour/ methods. If it has a lot of methods unique to the noun, it might help encapsulate them.	By Default it has to be a class
example, the string class in the standard template library. String is a single atomic noun with no further data members/ subdivisions but it has many methods which are specific only to strings	

2. Identify the methods. Usually the verbs are the methods.

KEYWORDS:

OBJECT
STATE
BEHAVIOR
METHODS
CONSTRUCTOR
DESTRUCTOR
DATA MEMBERS
CLASS

ENCAPSULATION
ABSTRACTION

Keywords/Buzzwords:

- Structured Programming
- Object Oriented Programming
- Object
- State
- Behavior
- Encapsulation
- Abstraction
- Class
- Constructor and Destructor

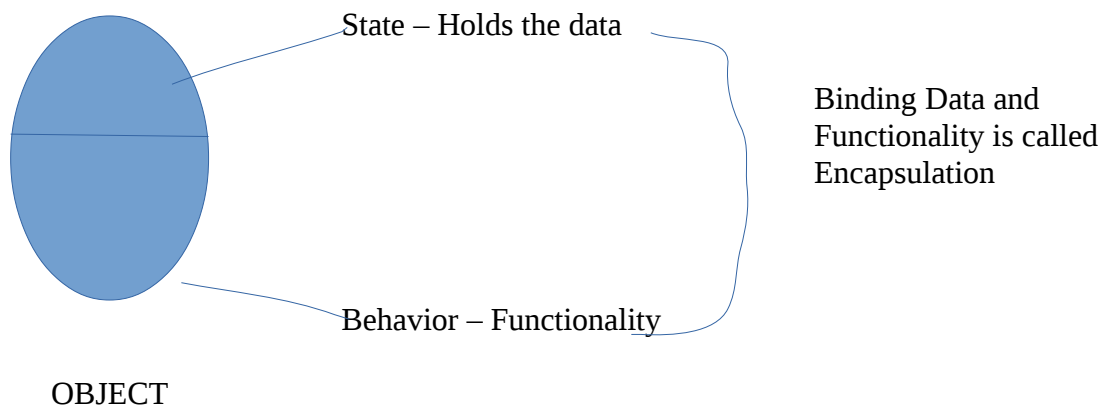


F1, F2 are certain function which accomplish certain tasks and they transport data to accomplish the task. Obj1, Obj2 are the Objects which transports the message to invoke a method to perform the given task.

Defn: A **software application** is a set of cooperating objects which gets things done by passing messages.

Defn: An **Object** is an encapsulation of state and behavior.

Defn: **Function/Method** is an encapsulation of behavior which takes some input and produces output.



A method of an Object can be invoked when a message is passed to it and it is important that you specify the Object to which you are sending this message because every Object will have a copy of the methods logically.

Interface is something which is showcase/represents the encapsulated data to the user. The user might not be knowing what kind of data is present and how it is handled. The point of contact to the user/client is the Interface.

Example to explain Encapsulation:

Aim is to travel from point A to point B.

In Structured Programming,

We can use a car and call a driver to drive us from A to B. Here, car acts as data and driver acts as method/function. We fetch the Car(data) and call the Driver(function).

By Encapsulation in OOP,

We could call a Taxi service. So, we have a ready service which serves our purpose to travel from A to B. Here, the phone call we make to the taxi services is the Interface. Taxi Service is where the car and driver is readily provided can be seen as encapsulation.

One Huge **Advantage** of Encapsulation is that the internal aspects is hidden from user/client and these internal aspects can be modified without affecting the interface. So, the impact of the change is localized inside the Object and thus, changes in OOP can be made with minimal impact on the system.

Encapsulation  **Information Hiding – Data is Protected & Covered**

Defn: Abstraction is the ability to collect small/low level data to a Higher level. Encapsulation is achieved by Data Abstraction. Basically, Data Collection is Abstraction.

OOP enables Encapsulation by Abstraction.

Defn: **Class** is a template for creating object.

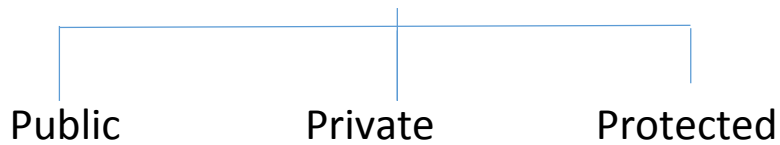
Rmk: State and behavior of an Object correspond to Attributes/Data members and methods respectively.

Defn: **Constructor** is special method for initializing an Object. It is a mechanism by which an Object is given life that is a Constructor gives a beginning state to a Object.

Defn: **Destructor** is a special method for “Cleaning up” the Object contents. Destructor is mainly important or used if you have to perform any function before deleting/removing the Object forever.

CLASS:

Class is made up of data and methods. Which are classified as access specifiers(scope of a member).It can be a data or a method.



The access specifiers are something that reflect the encapsulation of the members of the class.

Private:

A variable (data) or a method is a private then it can be accessible only by the methods **inside** the class.

Inside of a class:

Anything inside the scope of a class is called inside the class.

Public:

A variable (data) or a method is public if it can be accessible from inside and outside of the class.

Example:


```
class Account{
    private:
        int account_number;
    public:
        void printstatement();
}

int main()
{
    Account my-account;
```

```
my-account.account_number = 25;  
}
```

my-account.account_number = 25; is wrong because account_number is a private variable and can be only be accessed by the methods that are inside the class.

In classes there is a concept of “self” or “this” which refers to the object in which the function is called.

```
Account :: f()  
{  
    Account_number = 25;  Both are the same(Give the same effect)  
    [this-> account_number = 25;]  
}
```

But then a question arises that how to initialize an object. And therefore there is a special method called **Constructor**.

This is a method for initializing an object. There is no explicit call to the constructor and is called when the object is created.

There can be multiple constructors and the specific constructor is chosen by the compiler, based on how many parameters are specified while initializing the object.

There are two types of constructors.

1. Default constructor.
2. User defined constructor.

The concept of default constructor is invoked when there is no explicit call of the constructor. I-e the user has not specified a constructor.

The syntax for default constructor is as follows. <classname> ();

Even if one constructor is defined by the user then the default constructor is not invoked. I-e no constructor mentioned by the user => compiler generates default constructor.

Copy Constructor:

It is a special constructor when you want to initialize another object of the same class. Following is an example for a copy constructor.

```
Class Account {  
    Public:  
        Account(Account &a);  
};  
main()  
{  
    Account a1;  
    Account a2(a1); // copy constructor.  
    Account a3 = a2; // copy constructor.  
}  
Int f1(Account a1) // copy constructor.  
{  
    .....  
    .....  
    .....  
}
```

Copy constructor is also can be user defined. I-e the copy can be defined the way user wants it.

Default copy constructor initializes all the values in the new object same as the previous object.

Deep Copy:

Deep copy is a concept of copy constructor where ,if copied then all the objects in the links from the one starting will be duplicated.

Shallow Copy:

In shallow copy all the links are not duplicated. Only the head is duplicated.

Destructors:

Destructors are something that delete remove, delete the object.

Unlike constructors a class can have only one destructor.

In this case also the destructor can be user defined as well as default.

The default will only free the memory of the memory of the object that is asked to be destroyed. I-e it is a shallow delete.

Example:

```
main()
{
    Account a;
    ~a(); //destroys the object
}
```

New: constructor gets invoked; malloc: Constructor does not get invoked.

Delete: Destructor gets invoked; Free: Destructor does not get invoked

Class 3 Notes

Abhinav Chawla
IMT2013002

January 9, 2015

1 Access Specifiers

- Governs the scope of the data members(data+methods). It is intended for implementing encapsulation. There are three types of access specifiers
 - Private
 - Public
 - Protected

- **Private vs Public** These are two mainly used access specifiers. As from the name, private data members will be those members which can't be accessed by the client and they are hidden. They can't be used outside the class. Public data members can be accessed from outside the class. You can only affect the object data members if you have the permissions for it which are governed by these specifiers.

- **How to use Access Specifiers in C++?**

```
class Account{
private:
    int acc_num;
public:
    void printStatement(); //Doesnt take any argument as
                           //it takes the attributes of object
                           //directly to print
}
main(){
    Account myAcc;
    myAcc.acc_num=25; //This WILL GIVE AN ERROR.
                     //As you are not allowed to access private data
                     //from outside the class;
    myAcc.printStatement(); //This is perfectly fine as
                           //printStatement is a public function.
}
```

- **"This"** or **"Self"** refers to the object on which the method is invoked.

```
Account::f(){
    acc_num=25;
    //You can write as [this->acc_num=25;]
    //In Python we use self.acc_num and java we use this.acc_num
}
```

2 Constructors-User defined and Default

- Special public method for initializing an object.
- There can be any number of constructors, for a given class
- Naming convention of the constructor is the name of the class itself.
- As there can be any number of constructors, the compiler will itself find which is the best constructor to use.
- Constructors can be invoked only once.
- **Default Constructor** The above mentioned were for user defined constructors. The default constructor is implicit for every class and doesn't take Parameters. It will initialize every attribute to 0 or some random value. Default Constructor is created automatically by the compiler if there is NO user-defined constructor. Though it is advised not to use the default constructor, as you have less control over the program.

```
class Account{
    int acc_num;
    string holder_name;
    float balance;
public:
    Account(); //Default
    Account(int acc_num); //User Defined
    Account(int acc_num, string name, float balance); //User Defined
}
main(){
    Account a; // Invoking Default Constructors
    Account a1(25);
    Account a2(35,"John",500);
}
```

3 Copy Constructor

- To use the constructor of say , object 1, for the constructor of an object 2, we use the copy constructor.
- When you want to initialize an object with the use of another object of the same class.

- Different ways of invoking the copy constructor.

```

class Account{
public:
    Account(Account& a);
}
main(){
    Account a1;
    Account a2(a1); //First way
    Account a3=a1; //Second way
}

void f(Account a){
    ....
    ....
}
main(){
    Account a1,a2;
    f(a1); //Third way. This is passing by Value method and the
           //object gets copied.
    a2=a1; // This is ASSIGNMENT which is NOT EQUAL to INITIALIZATION
}

```

- The main difference in passing by value is that the whole value is copied while in Passing by reference only the pointer is being given.
- Non Primitive Data types should be passed by reference, for a better use.

4 Deep Copy v/s Shallow Copy

- Imagine a class like this where we are pointing a pointer of a class

```

class Account{
    int acc_num;
    float balance;
    customer *cust;
}

```

- Now we create an object a1 of the class Account. a1 will then point to customer class, let say the object c1. Now if we assign a new object such as [Account a2=a1;], it will create a member wise copy by default.
- The pointer also gets copied!! There is no new object of the class customer being created for a2. This doesn't traverse through the pointers.
- In deep copy, for every pointer in the object, it traverses through in the pointer and will create a copy for it and it will continue if even c2 has a pointer in it.
- We can consider deep copy as a linked list.
- The libraries are linked as entire files and not as individual functions.

- Whether to use deep copy or shallow copy, it depends on the symatics you want.

5 Destructor

- Special public methods invoked when an object is about to get destroyed/killed/deleted
- There can only be one destructor per class

```
class Account{
    int ..
    float ..
    public:
        ~Account(); //Can be default , can be user defined
}
```

- Default destruction is shallow. It will NOT traverse through the pointer and invoke the deep destructors.

```
main(){
    Account a;
}; //Destructor is invoked
```

```
main(){
    account *aptr = new Account();
} // Destructor is NOT invoked as it is a pointer;
```

```
main(){
    Account *aptr = new Account();
    .
    .
    .
    delete aptr; //Destructor is invoked
}
```

Class 3 Notes

Apoorv Vikram Singh

January 9, 2015

1 Class (A template for creating objects)

- Class Support - Access Specifiers (governs scope of members)
- Access Specifiers - public, private and protected.
- Access Specifiers intended for implementing encapsulation.

```
• class Account {  
  private:  
    int acc_number;  
  public:  
    void printStatement();  
};  
  
int main(){  
  Account myAcc;  
  myAcc.acc_number=25; //Not allowed  
  myAcc.printStatement(); //Methods can use private data  
}
```

- Convention in C++ - all data members are private by default

2 “Self” and “This”

- “self” and “this” refers to object on which method is invoked.

```
• Account::f();  
  Account a;  
  a.f();
```

- The object on which $f()$ is running is “this” object.

```
• Account::f()  
{  
  acc_number=25;  
  //actually what is happening is  
  //[this->acc_number=25]  
  //”this is a pointer”  
  //in JAVA it is this.acc_number and python self.acc_number  
}
```

3 Constructors (Special Public Methods for initializing)

3.1 User Defined

- There can be any number of constructors, based on the client need.
- Naming convention of the constructor in C++ is the class name itself.
- Examples of types of constructor -

```
class Account {
private:
    int acc_number;
    string holder_name;
    float balance;
public:
    Account(int acc_number);
    Account(int acc_number, string holder_name, float balance);
};

main() {
    Account a1(25);
    Account a2(25,"John",5000);
}
```

- C++ finds which one is the best constructor to use.
- Methods can be called in constructor.
- Constructor can be invoked only once, when the objects are created.

3.2 Default Constructor

- Implicit for every class and does not take parameters.
- If you want to use the default constructor write -

```
Account(); // In Class

Account a3; // In main(), invokes the default constructor
```

- Compiler does it anyway if and only if the class does not have ANY user-defined constructor. For example

```
class Account{
public:
    Account(int x);
}

main() {
    Account a(5); // Okay, user-defined constructor invoked

    Account b(); //NOT okay because there is no default constructor
                  //(neither explicit or compiler-generated)
}
```

- NOTE: As a good practice, never use the default constructor, to have more control over the program.

4 Copy Constructor (Special Constructor)

- Used when you need to initialize object 1 from object 2 belonging to the same class.
- Ways of invoking the copy constructor.

```
class Account {
public:
    Account(Account &a); //Copy Constructor
};

main() {
    Account a1;
    Account a2(a1); // First way of invoking copy constructor
    Account a3=a1; // Second way of invoking copy constructor
}

// The third way is passing by value

void f(Account a) {
    ...
    ...
}

main() {
    Account a1, a2;
    Account a;
    f(a); //Copy constructor gets invoked as it is pass by value
        //and object gets copied = Third way.

    a2=a1; // NOT copy constructor
        //it is an assignment statement and NOT initialization
        //Equivalent to memberwise copy
}
```

- Use always pass by reference (for non-primitive data-type) - lesser waste of memory.

4.1 Deep Copy v/s Shallow Copy

- Suppose we have a class like

```
class Account {
    int acc_number;
    float balance;
    Customer * cust; //Customer is some other class
};
```

- Suppose we create the object `a1` of type `Account` and `c1` of `Customer` type, and the third parameter in `Account a1` pointing to `c1`.
If we write `Account a2 = a1;`, it does a memberwise copy of `a1` to `a2`, and the address of `c1`, goes to the third parameter of `a2`, as well.
- This is shallow copy as it only copies the address, and doesn't traverse the pointers of the *Customer* `c1`.
- Deep copy traverses all the pointers, in `c1` and copies them to `a2`, unlike shallow copy.
- Advantage of deep copy is that if I delete some pointer, which the object was pointing to, it won't affect the client, as the copy is with the object itself.
- Advantages or Dis-advantages depends on the needs.

5 Destructor

- Special public methods invoked when an object is about to get destroyed / deleted / killed ...
- There can be only ONE destructor per class.
- Default destructor is shallow, it won't go to other pointers to delete it, ie (NO deep delete).
- Can be default (free up memory space) or user-defined.

```
class Account {
    int ...;
    float ...;
public:
    ~Account(); // Destructor
};
```

- Conditions for invoking destructor-

```
main() {
    Account a;
} // Destructor of a is invoked
```

```
main() {
    Account * aptr = new ... ;
    ...
    ...
    delete aptr; //Destructor invoked
}
```

- NOTE: Destructor is not invoked here :

```
main() {  
    Account * aptr = new Account();  
} // Destructor NOT invoked since it's a pointer
```

6 C++ v/s C

- C++ - new , delete
C - malloc , free
- Constructor gets invoked using new and not in malloc (therefore don't use malloc)
- Destructor gets invoked in delete , and not in free.

ACCESS SPECIFIERS (scope of members)

- Members include both data and methods
 - There are three type of access specifiers, public, private and protected.
 - Access specifiers are intended for implementing encapsulation
-

SYNTAX FOR defining a class Account:

```
class Account {  
  
private:  
    int acc_num // private data  
public:  
  
    void printstatement() // Public method. This function does not take any arguments.  
};
```

If you do not mention any access specifiers for the class members, it is by default taken as private.

```
Main  
{  
Account myAcc; // Declares an object of class Account.
```

```
MyAcc.acc-num=25; // This is not allowed since this is outside of class and acc-num was private  
myAcc.printstatement(); // This is allowed.  
}
```

Hence we can say that public methods become gateways to access private data.

CONSTRUCTOR

To initialize an object, constructors are used.

- Constructors are special public methods for initializing an object.

```
Class Account{  
  
    int acc_num  
    string holder_name  
    float balance  
  
    Public:  
        Account(); //Constructor  
        Account(int acc_num); //Constructor  
        Account(int acc_num, string name, float balance) // Constructor  
};
```


CLIENT CODE:

```
main()
{
Account a1(25)
Account a2(35,"John",500);
}
```

Note that constructors are invoked but there is no explicit calls to the constructors. They are invoked only once when the object is created.

There are two types of constructors:

- a). User defined constructors
- b). Default constructors

Copy constructor

Copy constructors are used when you want to initialize an object using another object belonging to the same class.

Eg:

```
Class Account{
    Public:
        Account(Account &a); // Copy constructor
};

main()
{
    Account a1;
    Account a2(a1); //Copy constructor. a2 initialized by a1
    Account a3=a1; //Copy constructor. a3 initialized by a1
}
```

```
main()
{
    Account a1;
    Account a2;
    a2=a1; // Here copy constructor is not invoked since the object a2 was already created. This is
           an assignment statement and not initializaion.
}
```

Two types of copy constructors:

Let an object a1 of a class Acc has a member which points to an object b1 of a class Cust.

a) Deep copy: In deep copy, if you create a copy of a1, a copy of b1 is also created and the member of the copied a1 points to the copied b1.

b) Shallow Copy: In shallow copy, if a copy of a1 is created then both a1 and its copy point towards the same object b1.

DESTRUCTORS

Destructor is a special public method invoked when object is about to get destroyed.

```
Class Account {  
    int ....  
    float ....  
  
    Public:  
        ~Account(); // Destructor. Can be default or user defined.  
};
```

CASES WHERE DESTRUCTORS ARE INVOKED.

```
main()  
{  
    Account a;  
} // Destructor invoked here
```

```
main()  
{  
    Account * deptr = new Account();  
} // Destructor not invoked here since deptr is just a pointer
```

```
main()  
  
{  
    Account * dept = new Account();  
    delete dept; // Destructor invoked  
}
```

NOTES: June 13 2015

- by Vikas Yadav

Inter Class Relationships

- Association
- Aggreation
- Composition
- Inheritance
- Realization

What is a UML Notation?

UML Notation Stands for Unified Modelling Language. It provides a standard way to visualize a software program

It has a set of Diagrams which are used to express a program:

Class Diagram

Used to express static structure of program

Sequence Diagram

Used to express behaviour of the program – it shows how objects communicate with each other

Activity Diagram

Used to express logic of the program – is somewhat similar to the flowcharts we use

Collaboration Diagram

Combination of both sequence diagrams and activity Diagram

Use case diagram:

Documenting features of the program

State Diagram

Class Notation in UML

Account
+ acc_number : int
+ balance : int
+ get_balance() : int
+ get_acc_number() : int

-> On top we have the name of the class

-> In middle we have all the (public) attributes related to the class

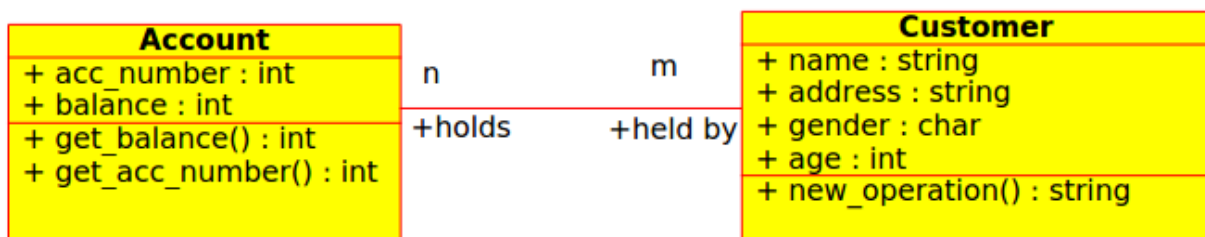
-> Third row has all the (public) methods attached to the class

Association

- Simple lines show that they are associated
- We give association a name

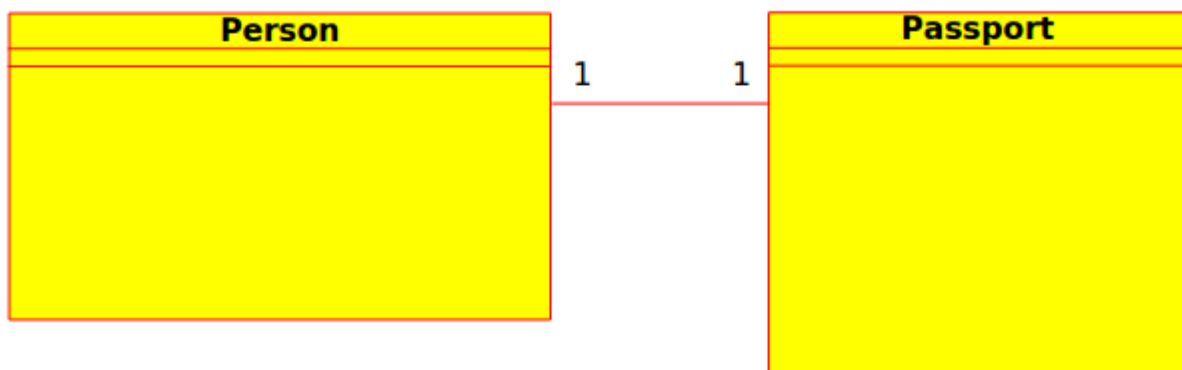
Cardinality

One to many

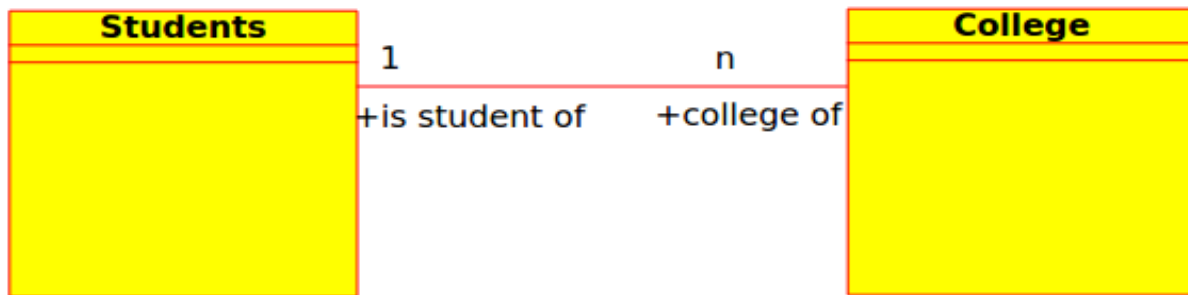


---This shows that each account can belong to more than one customer and each customer can hold more than one account

One to One



-- This shows that each person can have atmost one passport and each passport can belong to atmost one person.

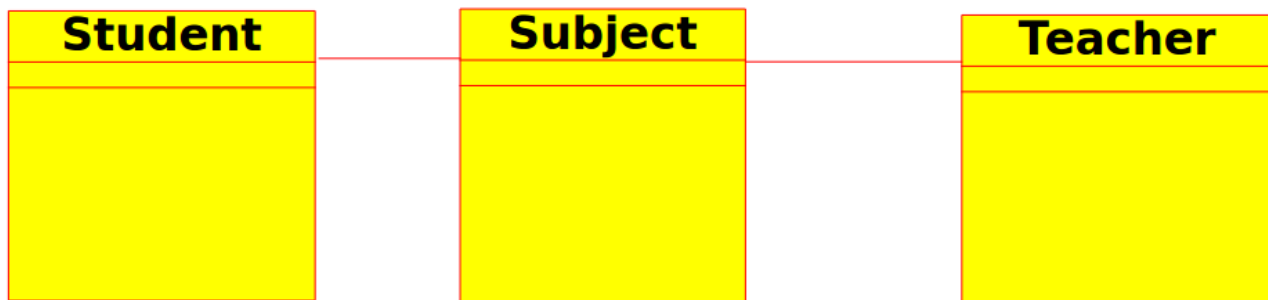


Degree

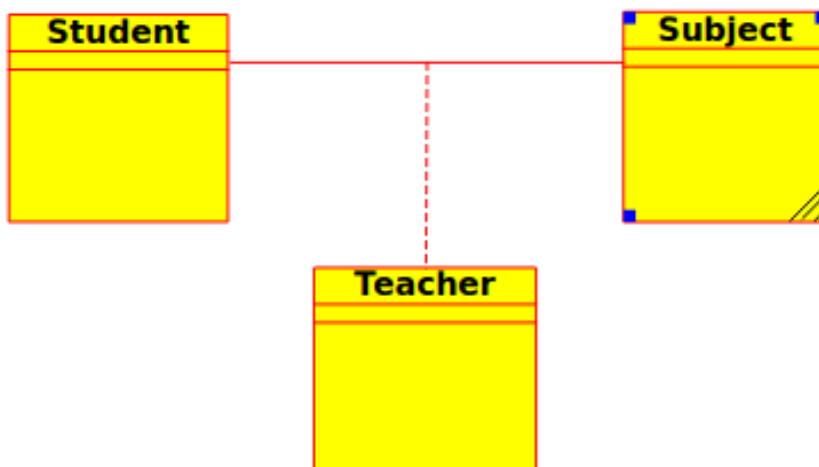
-Degree of association specifies number of unique classes involved in that association

1. Binary involves 2 classes
2. Ternary involves 3 classes
3. n-ary involves greater than 3 classes

Example of Binary



Ternary



Self Referential Association

A special kind of binary association where both classes are the same.

Example 1

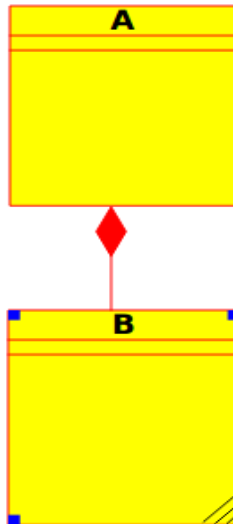
Class Person; and Association - Friend.

Each friend is a person, and each person can have multiple friends

Example 2

Class Employee; and Association Manager;

Composition



Class A -> Composite object/class

Class B -> Component object/class

Here A is composed of B
or, we can say
B is part of A

Existence Dependency:

Components cannot exist without being linked to a composite

Child Object cannot exist without parent

Component gets deleted when composite gets deleted

Inter-class relationship and its visual modeling

It is very important to know how objects are related in an Object Oriented Programming project. Some relationships between objects are Association, Aggregation, Composition, Inheritance, Realization etc. Relationship between object is better understood by pictures. UML (Unified Modelling Language) helps us to visually model our project so that everyone can understand its structure and how it is achieving a task given.

The basic things we need for visual modelling of our project are the diagrams used by Unified Modelling Language:

List of UML diagrams:

- Class Diagram
- Sequence Diagrams
- Activity Diagrams
- Collaboration Diagrams
- Use Case Diagrams
- State Diagrams
- Component Diagram
- Deployment Diagram
- Object Diagram
- Package Diagram
- Profile Diagram
- Composite Structure Diagram
- Communication Diagram
- Interaction Overview Diagram
- Timing Diagram

Explanation of some important UML diagrams:

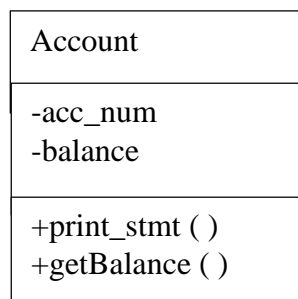
1. Class Diagram: This explains the static structure of the program. That is the relationships between the classes.
2. Sequence diagrams: This explains the behavior of our program
3. Activity Diagram: Its main emphasis is on the logical behavior of the program (For example Flow Chart)
4. Collaboration Diagram: This serves the same purpose as sequence diagram but looks different
5. Use case diagrams: This helps in documenting the features of the software

6. State diagrams: Behavior and state of objects in our project.

Now that we have seen what UML diagrams are, let's see how we denote classes and association between them:

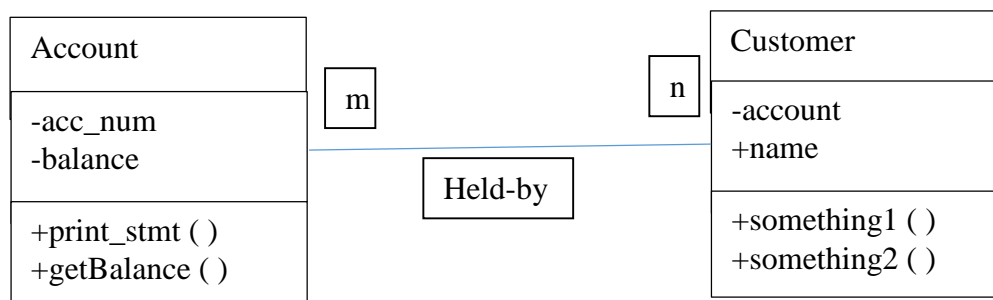
A CLASS is a rectangle with three sections, first section for name, second section for attributes and the third for methods. An optional thing is to put “-” for private members of class and “+” for public members.

For Example: There is an Account class with “acc_num”, “balance” attributes and print_stmt (), get Balance () methods. Then the class diagram of that account class will be as follows. Note the +, – symbols before members of the class.



An ASSOCIATION is a simple line drawn connecting two class diagrams. The line indicates that they are ASSOCIATED. We give the line (association) a name. Cardinality of association is mentioned to know how many objects each associated object can associate to.

For Example: There are two classes Account and Customer. We say Account is “held-by” customer. Every Account can be held-by many customers (n indicates this) and every customer can hold many accounts (m indicates this).

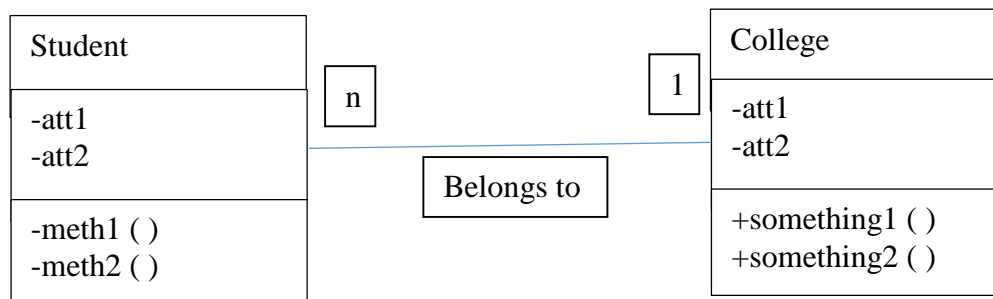


Default reading direction is assumed to be left to right and top to bottom. That is we say “Account is held by Customer”. If cardinality of an object1

associated with object 2 is one then we put 1 near object2. If it is greater than one, then we put n or m.

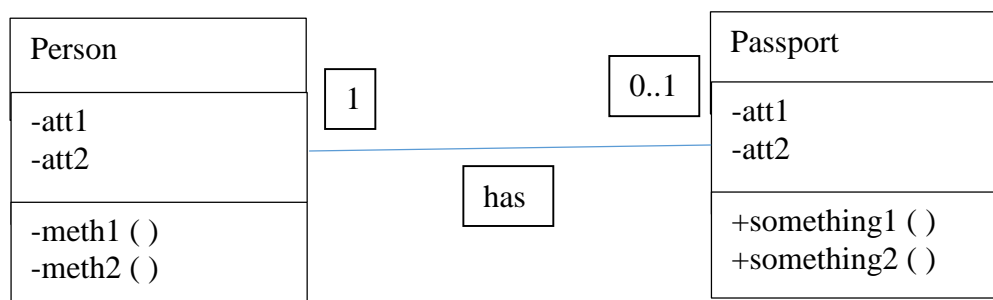
The previous example we talked is an example of many-to many-Association. There are other associations like one to one and one to many.

Example for one to many association:



In this case every student can study in only one college. We say “Student belongs to college”.

Example for one to one Association:



Here we put 0..1 instead of 1. It means that every person has at most one passport. In this case we don’t need an explicit class for Passport since every person has maximum one passport, but it passport is preferred to have independence existence than to merge into person since it can be used elsewhere

-Degree of association is the number of unique classes that are involved in the association.

Example: binary, ternary,

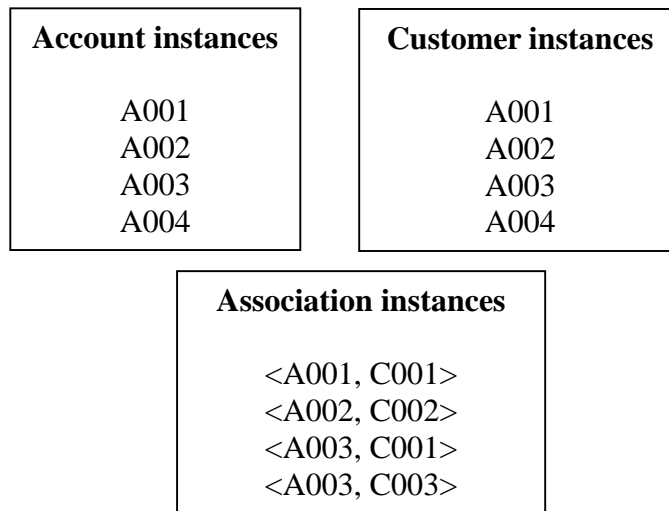
All the above examples have degree two since two types of classes are involved in the association.

-Account Instance

An account instance will help us understand degree of association even better.

Let’s say Account and Customer association between classes, Account and Customer, is denoted by <Account, Customer> is a binary association. Its Association instance will look like <A001, C001>,

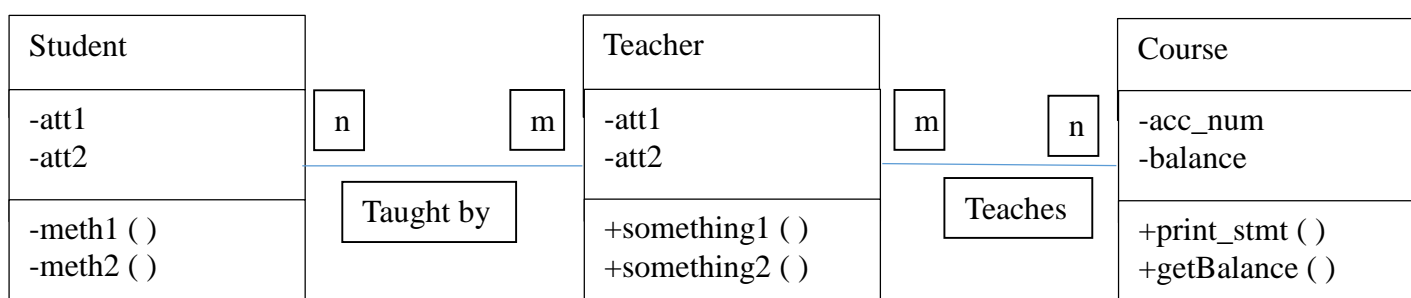
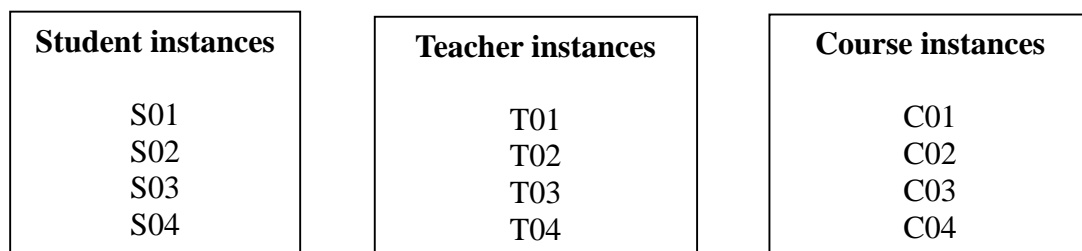
<A002, C002> <A003, C001> <A003, C003> where A001, A002, A003, A004 are instances of Account class and C001, C002, C003, C004 are instances of Customer class.

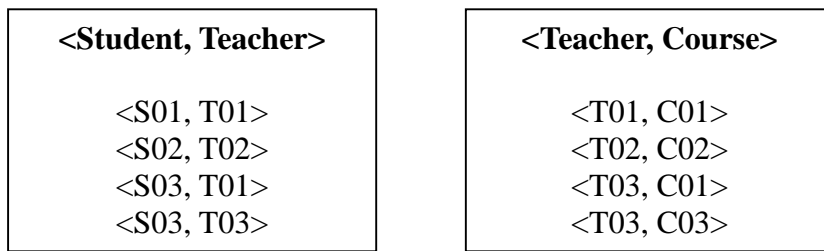


Then a ternary association will look something like <Account, Customer, Bank> .Till now we have only seen binary account instances in examples given above, but what is the use of a ternary association.

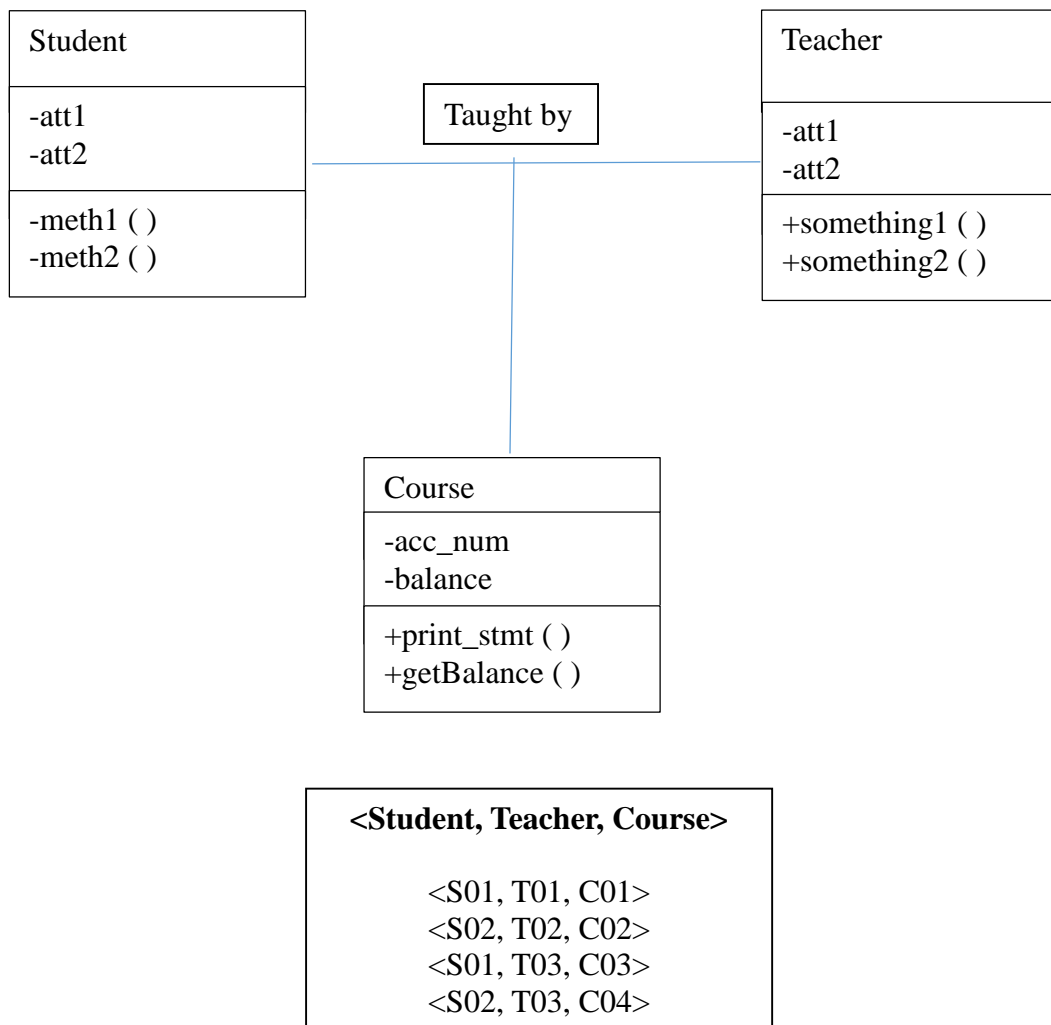
That is described by the following example:

Let's say there is pair wise association between student, teacher and subject classes as <student, teacher>, <student, subject> and <teacher, subject>.





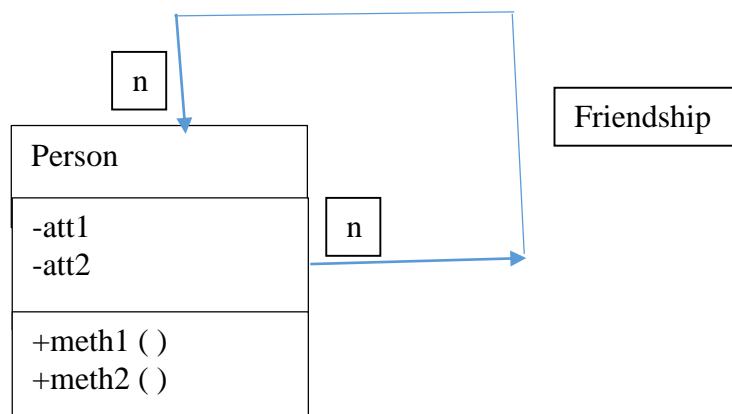
From this we can find which teachers teach a student and what all subjects a teacher is teaching is studying. But we can't say which teacher teaches a specific subject to a student. This is where **ternary association** comes in.



Now from the above association we can communicate the information we can communicate the information that we could not, using the binary instance form of the above ternary instance.

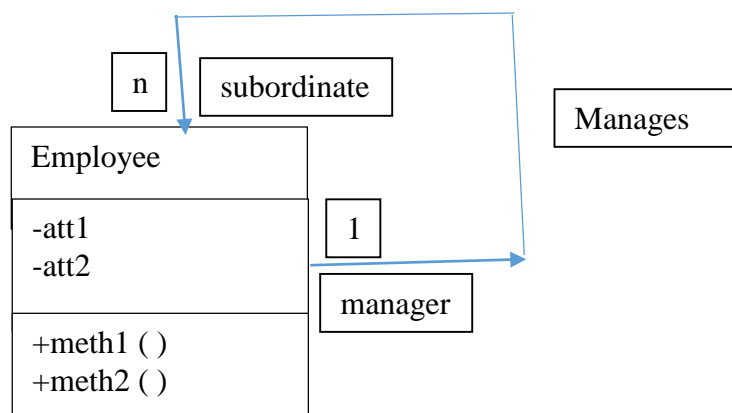
Self-referential association:

A special form of binary association in which both the objects involved will be of same class is called self-referential association. Associated objects may or may not be same but the classes of these objects should be same.



This is a many to many self-referential relationship. Every person can be friends with many other persons. There can be many to many and one to one self-referential relationships.

One to many self-referential relationship example: A manager (an object of employee class) manages many subordinates (Objects of employee class).

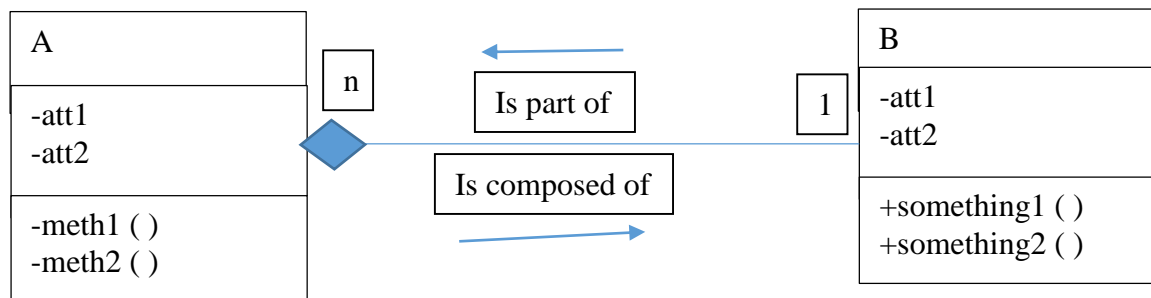


This is a one to many self-referential relationship. Every manager can manage many other subordinates.

A special type of association: Composition

This is a strict 1:n (one to many) or 1:1 (one to one) and cannot be m:n (many to many) association. Name of the association is “is-part-of” or “is-component-of” depending on the direction we read. The following diagram shows a composition association between classes A and B.

In this case we say A is composed of B and B is part of A.
A is composite and B is Component.



Existence dependency:

Components cannot exist without being linked to a composite or you can say Child object cannot exist without parent object. Component gets deleted when we delete composite

Inter class relationship

- Association
- Aggregation
- Composition
- Inheritance
- Realization

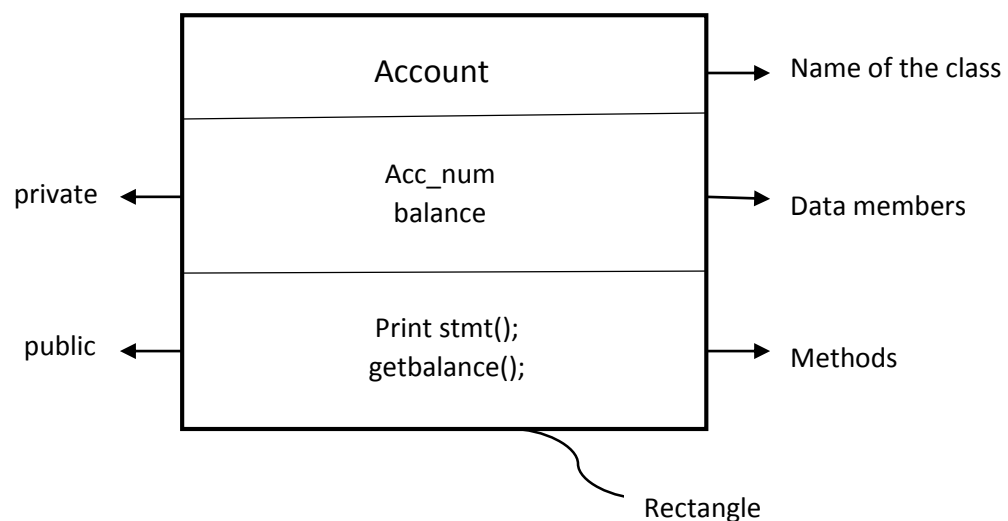
Notations:

We will work with UML(Unified Modelling Language)

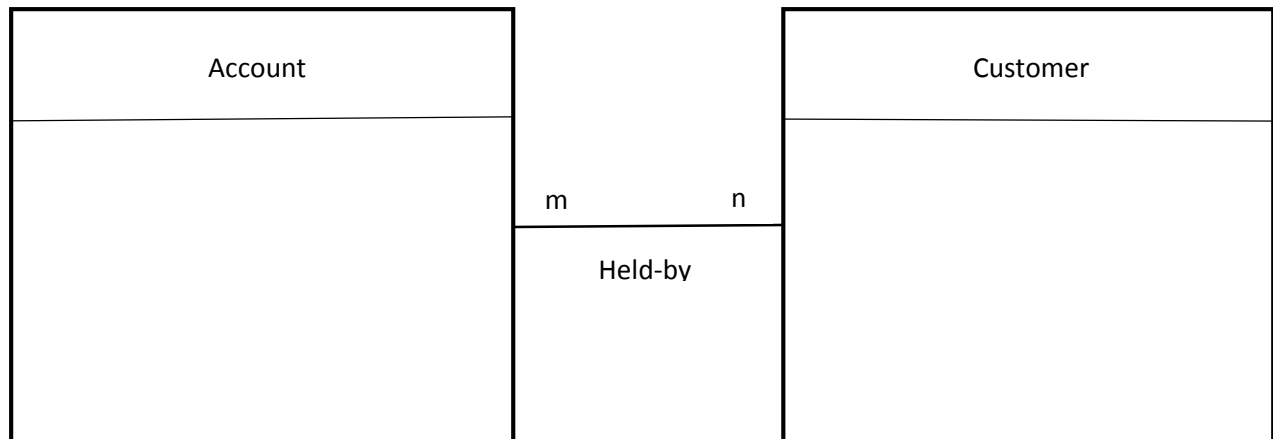
Visual modelling- communicating is easier, but we need a standard.

1. Class diagram (structure of application)
2. Sequence diagram (behaviour)
3. Activity diagram (behaviour, logic)→ similar to a flow chart.
4. Collaboration diagram (dual of sequence diagram)→ primary center of objects.
5. Use case diagram (documenting the features of the program)
6. State diagrams (behaviour/state)

Class notation



Association



Classes are connected only if there is need for object of class Customer to be used by object of class Account

Necessity in class diagram:

1. Line connecting the classes
2. Name of the association
3. Cardinality
4. Degree
5. Role (optional)

Cardinality

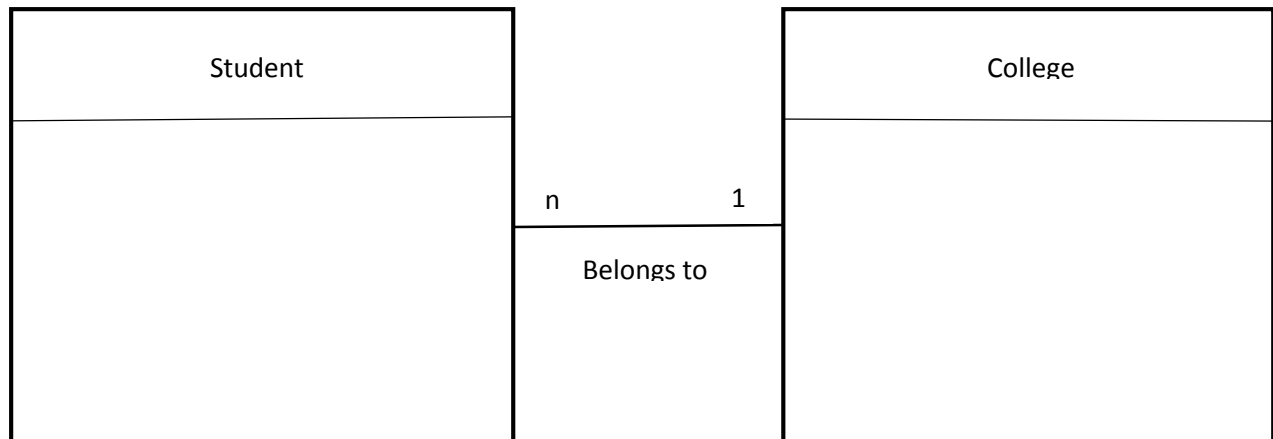
For every object of class Account, how many link object of class Customer is likely to be associated.

Example: For the same account there can be multiple customer (joint account). If it is more than '1' we write 'n' above the connecting line, else we write '1'.

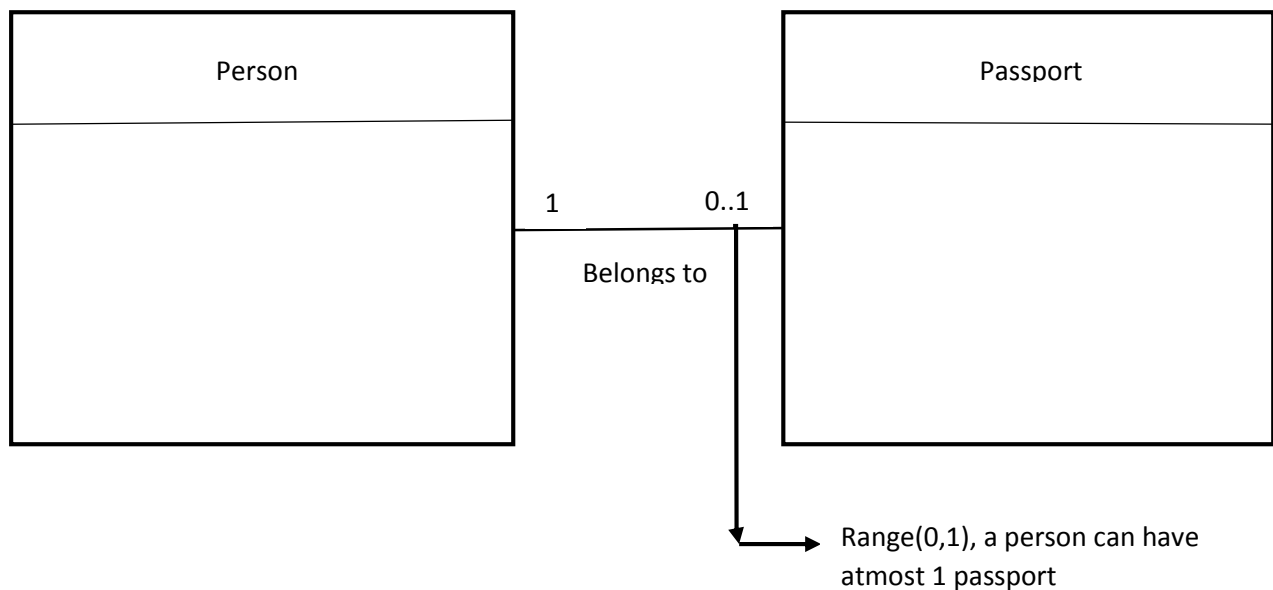
Similarly for every Customer how many Accounts are allowed, assume greater than 1. Therefore we write 'm'.

This is known as “**many-to-many**” association.

Similarly we can have **one-to-many** association.



Similarly, we can have **one-to-one** association



There is no harm in merging these two classes. There is small possibility of independent existence. It is more of a design thing.

Degree of association: number of unique classes involved in the association.

Binary involves 2 classes

Ternary involves 3 classes

n-ary > 3

Association instances:

Account instance (objects)

<A001, 30000>

<A002, 25000>

Customer instances

<"john", - - - >

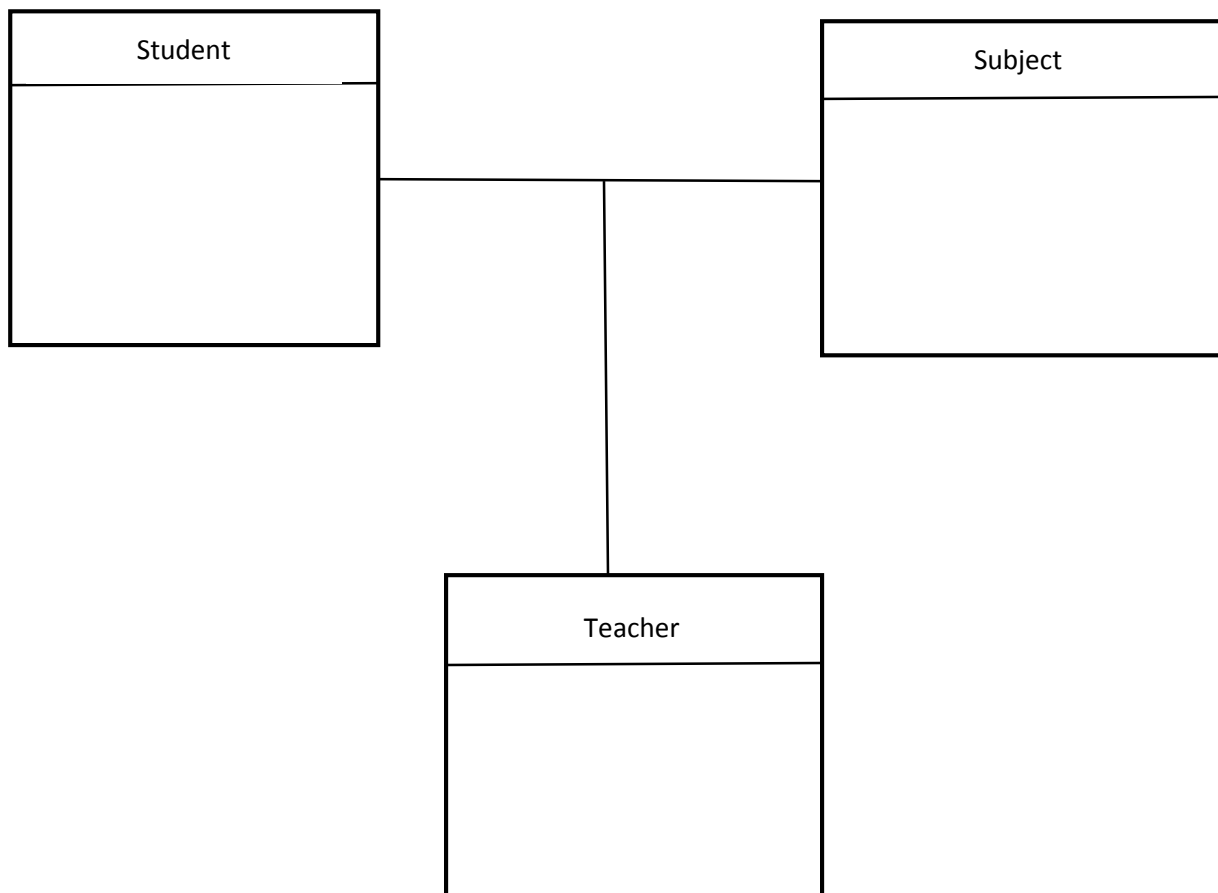
<"jane", - - - >

Association instances

<A001, "jane">

<A002, "jane">

<A001,"john">

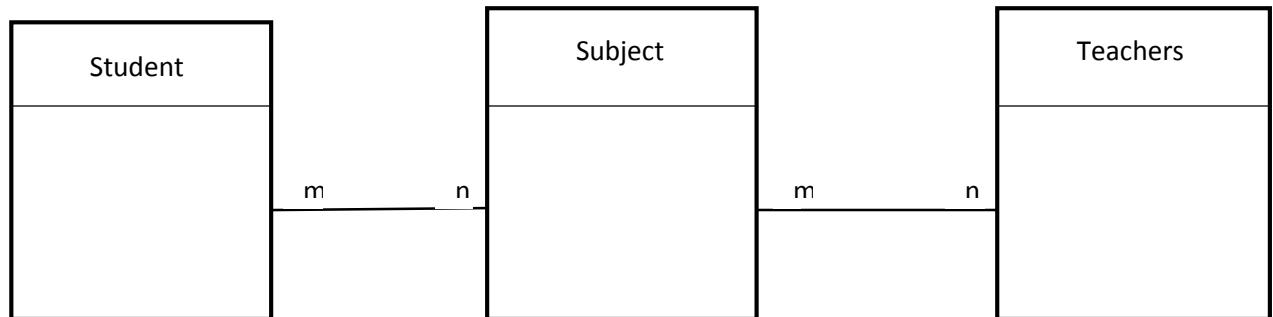


Degree =3

<S01, C01, T1>

<S02, C01, T2>

We know which student has taken
which course and is taught by which
teacher



Degree =2

<S01, C01>

<S02, C01>

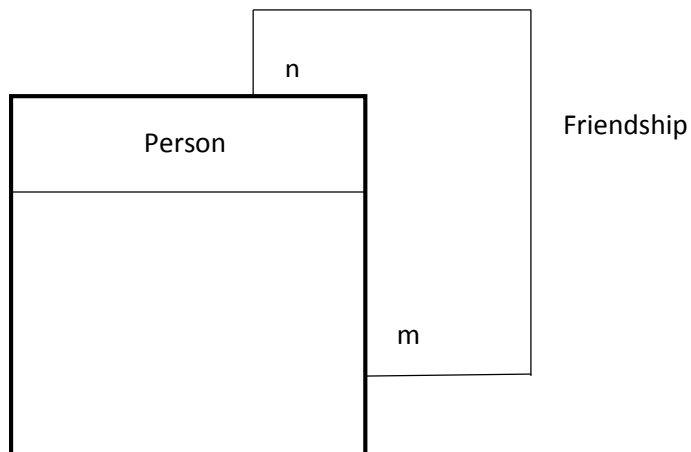
<C01, T1>

<C01, T2>

Here, both S01, S02 are studying C01, but both T1, T2 teach C01, we don't know S01 is taught by which teacher. Hence the design depends on the requirement of the client and 3 classes doesn't mean degree 3.

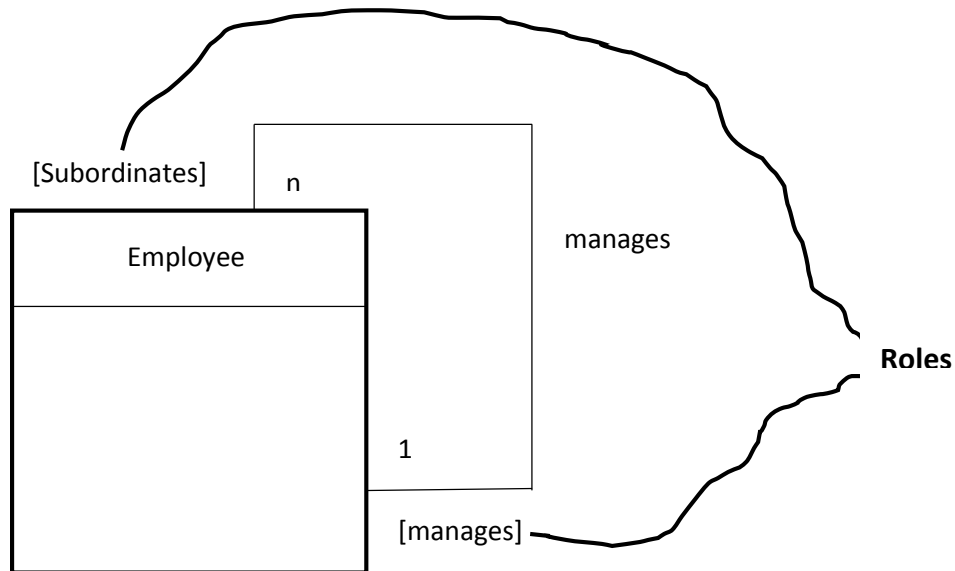
Self-referential association

A special binary association where both classes are the same



Note: NOT a unary relationship

1-n self-referential association



<e1,e2>

<e1,e3>

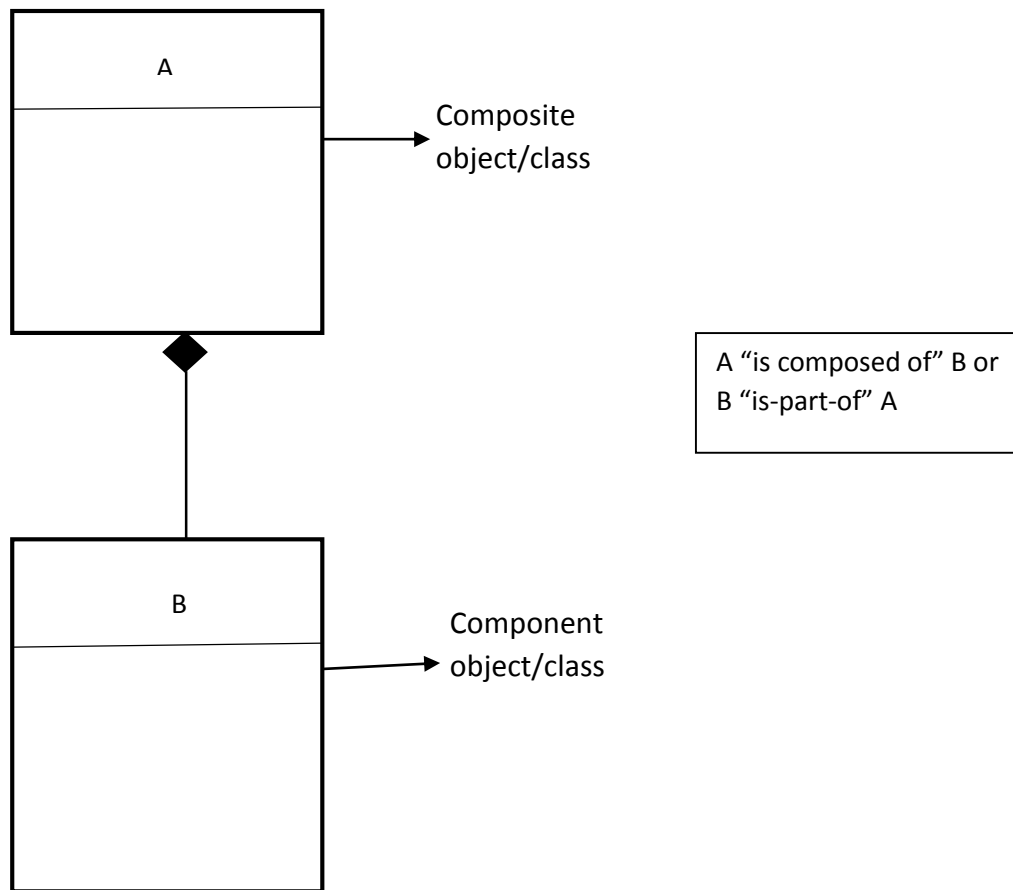
<e1,e5>

Composition

Special type of an association.

Properties:

- Name is fixed: “is-part-of” or “is-composed-of”.
- Is strictly 1:n or 1:1



Existence dependency

Component cannot exist without being linked to a composite. i.e. child object cannot exist without parent object

Components get deleted when composite gets deleted.

C++ NOTES: 13th Jan, 2015

Devarsh K U

IMT2013012

Inter-class relationships:

- Association
- Composition
- Aggregation
- Inheritance
- (Realization)

Notations: UML (Unified Modelling Language) is a notation used to describe any software.

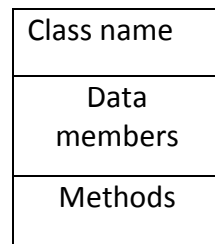
- Standard for Object Oriented Programming (OOP)
- Supports visual modelling i.e. its features expressed in pictures.

Diagrams of UML:

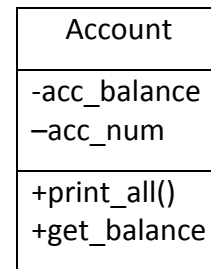
- 1) Class diagram: Collaborates classes i.e. shows classes, their types, contents and the relationships between them. So, describes structure.
- 2) Sequence diagram: Describes behaviour of classes. Emphasis is given to sequence of message calls and logic.
- 3) Activity diagram: Describes logic and data flow like a flowchart.
- 4) Collaboration diagram: It's a dual of sequence diagram. Here, emphasis is given to objects and the messages that facilitate their interaction.
- 5) Use case diagram: Documenting features of the software.
- 6) State diagram: Describes behaviour and state of objects and the transition between them.

Few other diagrams included in UML 2 are, Timing diagram, Package diagram, Object diagram, Interaction overview diagram, Deployment diagram, Composite structure diagram, Component diagram and Communication diagram.

Class notation:



General Structure



Example

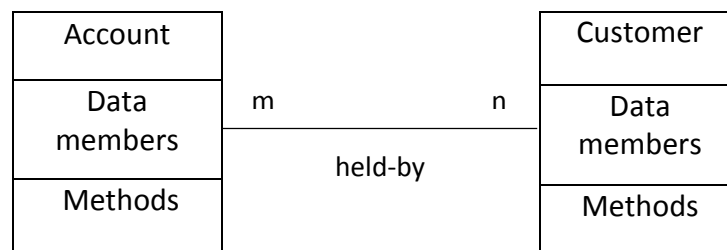
+ : public member

- : private member

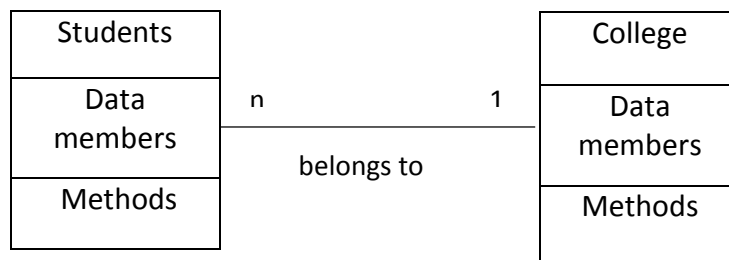
Association: It defines the relationship between classes that allow that interaction through objects. It is required only when there is a need to access a member of a class through the object of another class. In the example given below, object of class Account needs to access contents of objects of class Customer.

Features of an association:

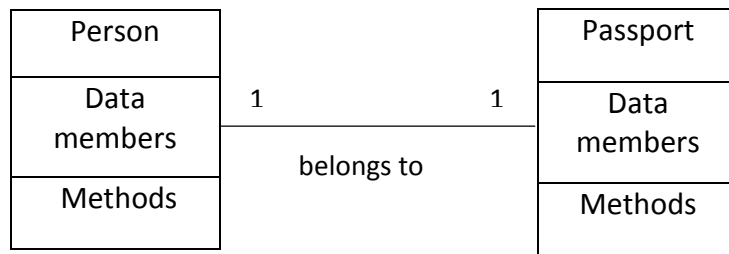
- 1) **Line connecting classes:** unless there is an arrow, association is bidirectional.
- 2) **Name of association**
- 3) **Cardinality:**
 - For every object of class Account, the number of objects of class Customer that are linked is say n, is written as shown.
 - For every object of class Customer, the number of objects of class Account that are linked is say m, is written as shown.



Many to many association



One to many association



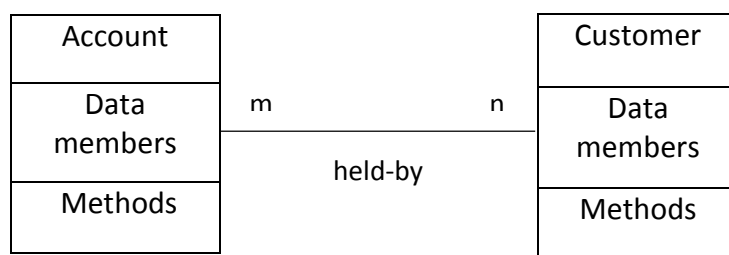
One to one association

When m, n or 1 is said to be the cardinality, it means at least m, n or 1 respectively. In case of 1, it can be written as 0.1.

4) Degree:

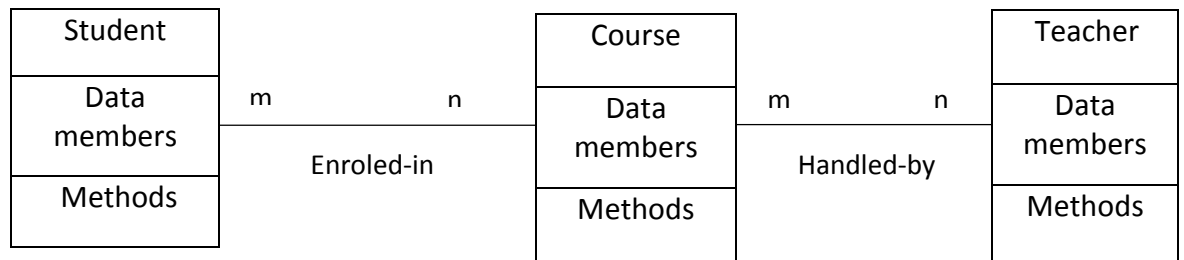
- Number of classes a class depends on in an association.
- **Association instance:** an object formed from multiple objects belonging to different or same class.
- It can also be said that degree is the number of objects required to form an association instance.

Binary: involving 2 classes



Association instances: eg: <Account1, Customer 2>
<Account2, Customer 2>

Ternary: involves 3 classes. Structure can be in many ways. Couple examples are



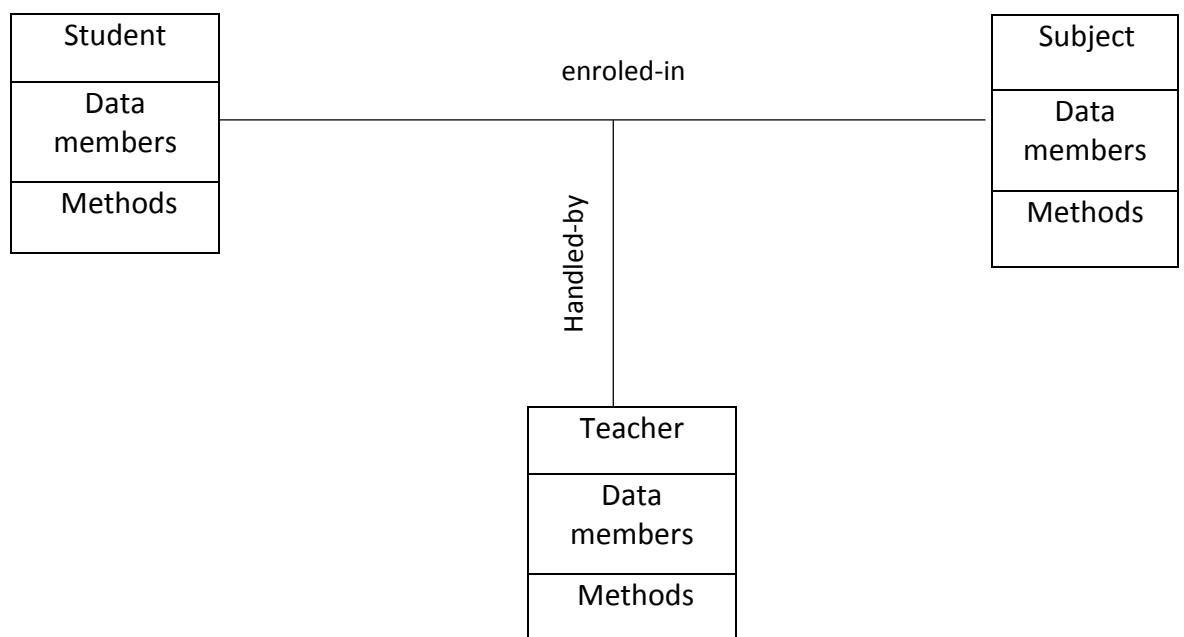
Association objects:

<Student 1, Course 1>

<Course 1, Teacher 1>

<Course 1, Teacher 2>

In the above case, degree is 2 and a problem arises when multiple teachers take the same course, so for student 1 taking course 1, the teacher can't be decided. So based on the requirement of the problem, the structure is constructed. An alternative to overcome this would be the following structure.

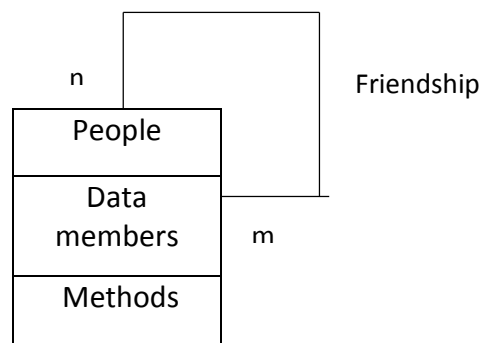


Association objects:

<Student 1, Course 1, Teacher 2>

In the above case, degree is 3 but there is no ambiguity as in the previous case. It is evident that teacher 2 handles the course 1, taken by student 1. For this example, this structure is more appropriate.

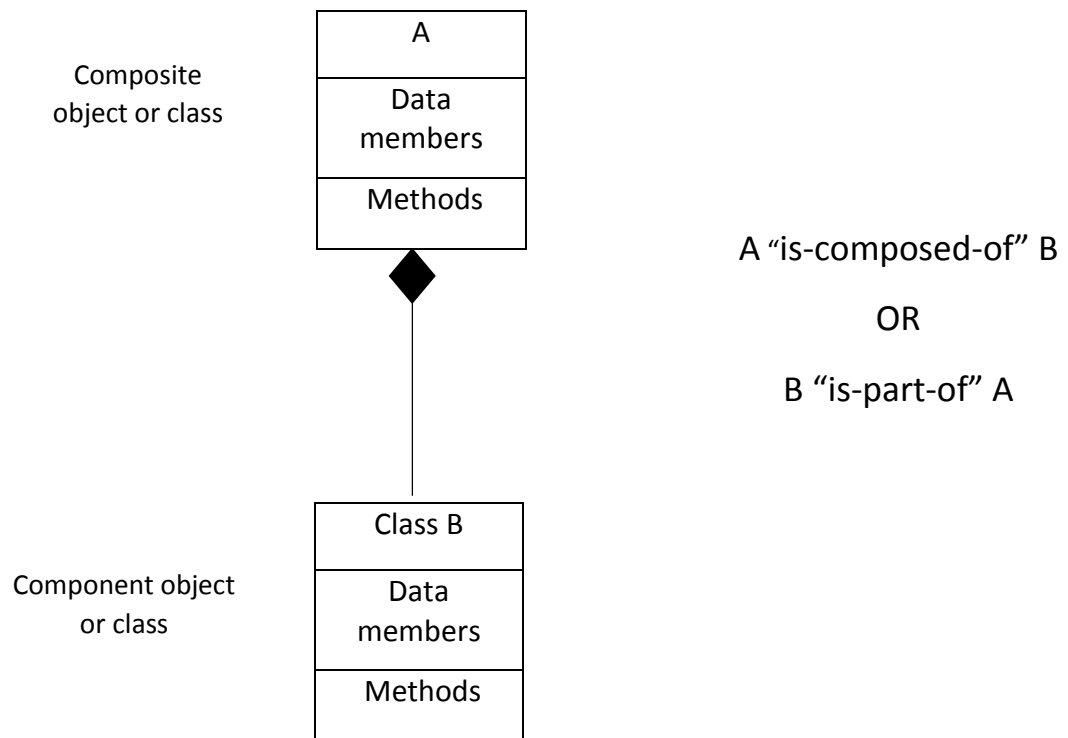
Self-referential association: A special binary association where both objects belong to the same class. An example would be friendship represented as:



- 5) **Role:** It defines the role of an object in an association object. This comes to importance in self-referential association.

Composition: A special case of association in which:

- Name is fixed and is, “is-part-of” or “is-composed-of”.
- Can never be many to many association.
- Represented by a small filled diamond.



Existence dependency:

- Components cannot exist without being linked to a composite.
- Component gets deleted when composite gets deleted.

Inter-class relationships

In addition to standalone classes, C++ supports more advanced class design based upon relationships between classes. There are broadly five types of inter-class relationships:

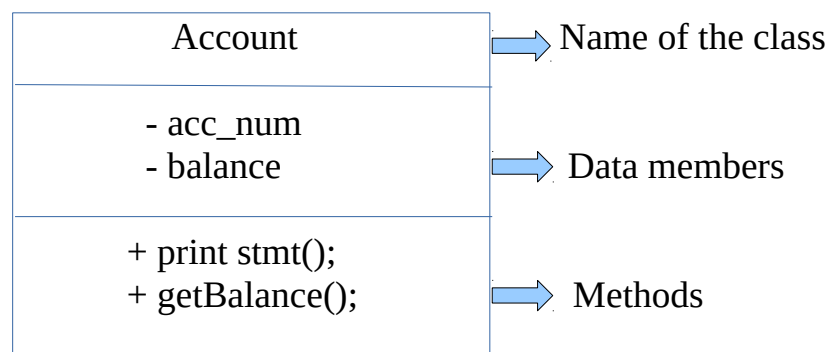
1. Association
2. Aggregation
3. Composition
4. Inheritance
5. (Realization)

UML (Unified Modelling Language) Notation:

UML is a standard notation for object oriented design that supports visual modelling. Several aspects of software can be described using UML:

1. Class Diagram: Expresses static structure of the entire application
2. Sequence Diagram: Used to describe behaviour
3. Activity Diagram: Similar to flowcharts, also used to describe behaviour (mainly logic)
4. Collaboration Diagram: It is a dual of sequence diagram. While in sequence diagram the emphasis is on the sequence of operations, in collaboration diagram the emphasis is on objects and the messages passed between them
5. Use-Case Diagram: Used to document features of software. Provides external view of how users are going to use your system
6. State Diagram: Depicts the possible states that a system can be in and the transitions that can take place from one state to another

Class Diagram:

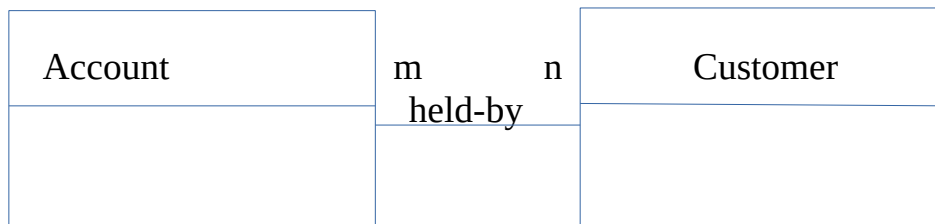


Class Notation

'+' indicates that the data members are public and

'-' indicates that the data members are private

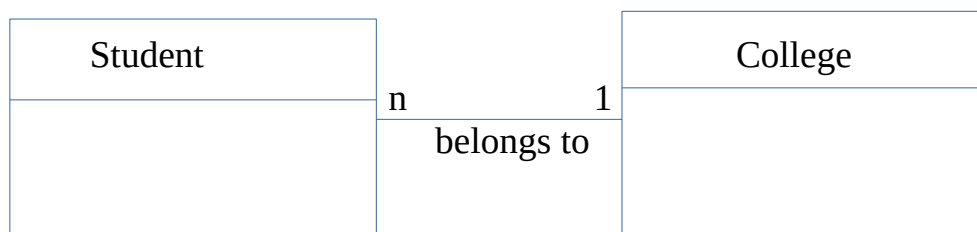
Association:



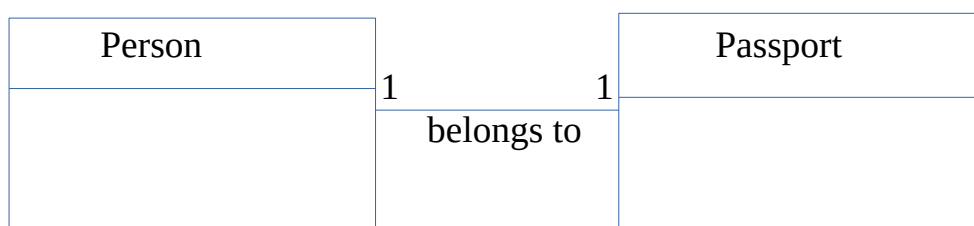
The **line** between the two classes denotes that they are linked to each other. The association is in addition given a **name** 'held-by' which means Account is held-by Customer.

The m and n above the line denotes the **cardinality** of the association.

In the above example every customer can hold more than one account (m) and each account can be held by more than one customer (n). This is hence an example of a **many to many** association.



This is an example of a **one to many** relationship. A student can belong to exactly one college, while a college can have many students.



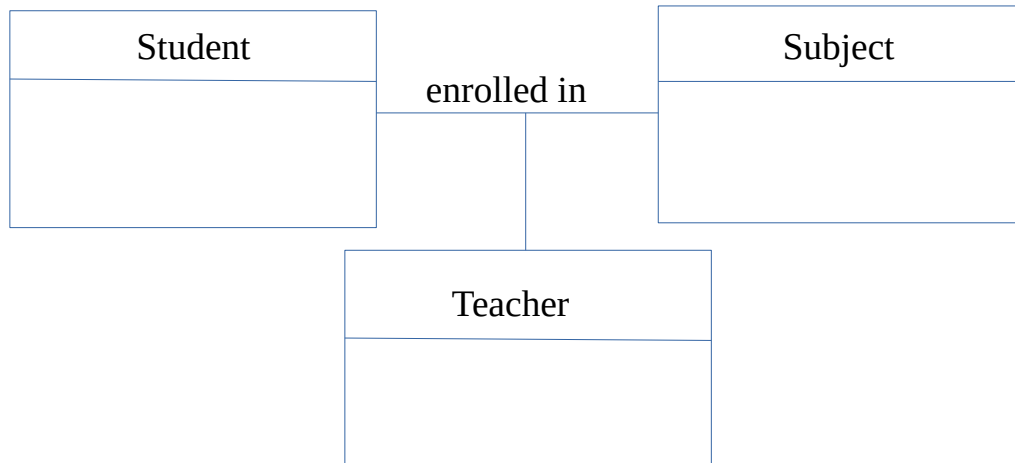
The above is a **one to one** relationship. One person can have exactly one unique passport and vice-versa.

Similarly, associations can also be classified based on the number of unique classes involved, called the **degree** of association.

All the above associations involve two classes and are examples of **binary** associations.

For e.g. in the account-customer association, if <A001, 30000>, <A002, 25000> are instances of account class and <"John", addr1>, <"Jane", addr2> are instances of the customer class. Instances of the binary association, are <A001, Jane>, <A002, John> etc.

Similarly, a **ternary** association involves three classes



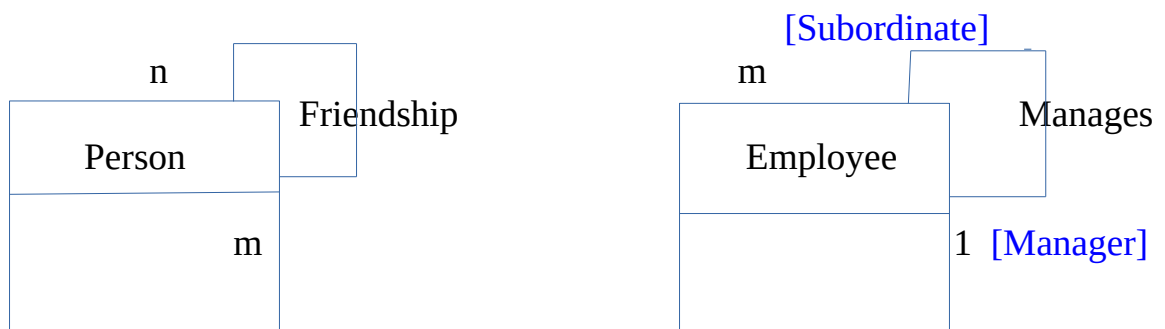
If S01, S02.. are instances of the class Student. C01, C02.. are instances of class Subject and T1, T2..are instances of the class Teacher.

Then, in order to find out the teacher of S01 for the course C01,if S01 takes the course C01 , the above association should be modelled as a ternary association with instances of the association taking form as <S01, C01, T1>, <S01, C01, T2> etc.

If the association involves more than three classes then it is called n-ary association.

Self-Referential Association:

It is a special **binary** association where both classes of the association are the same. For example,



The Person class self references itself when the association is friendship, it is a many to many self referential association. Similarly, the employee class self references itself when the association is manages (manager is also an employee). This is a one to

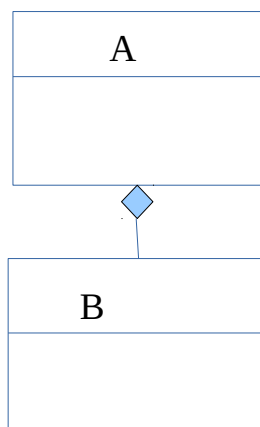
many self referential association.

In order to explicitly depict who is managing whom in the second example above, the **role** is specified in square brackets. This is optional and is almost always implied.

Special types of Associations:

1. Composition:

Name: is-part-of / is composed of



Read as: A is “composed of” B OR
B is “part of” A

A is called the **composite** object/class
B is called the **component** object/class

A composition association strictly **cannot** be **many to many** (m:n)

Existence dependency:

1. In such an association, components cannot exist without being linked to a composite or child objects cannot exist without parent.
2. Component gets deleted when composite gets deleted.

Scribe notes
15 january 2015

➤ Interclass relationships:

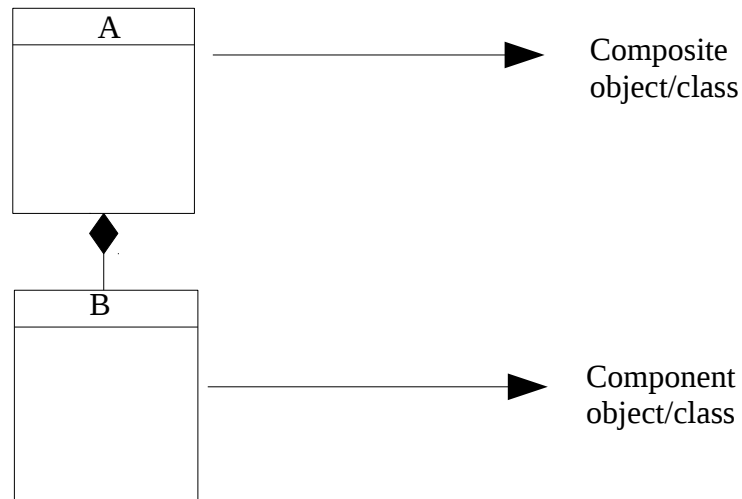
1. Composition:

It is a special type of association in which name is fixed.

Name: “is-part-of” or “is-composed-of”.

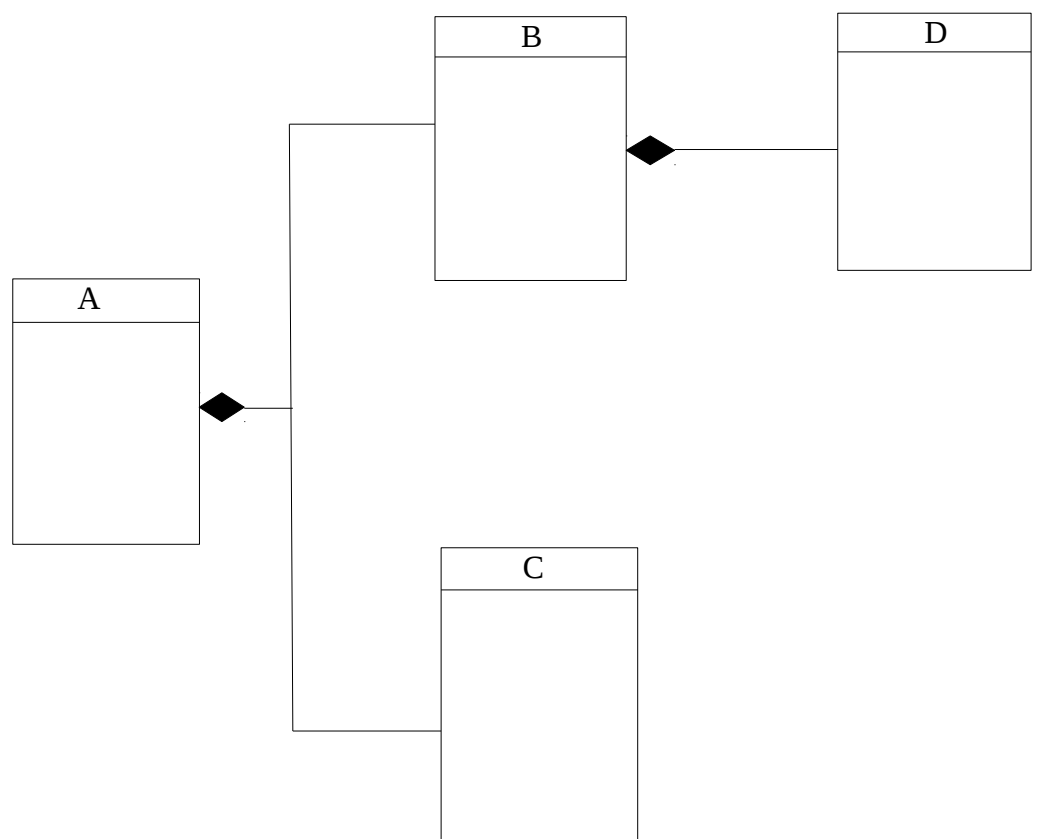
Cardinality is strictly 1 to n or 1 to 1, it cannot be m to n.

Notation:

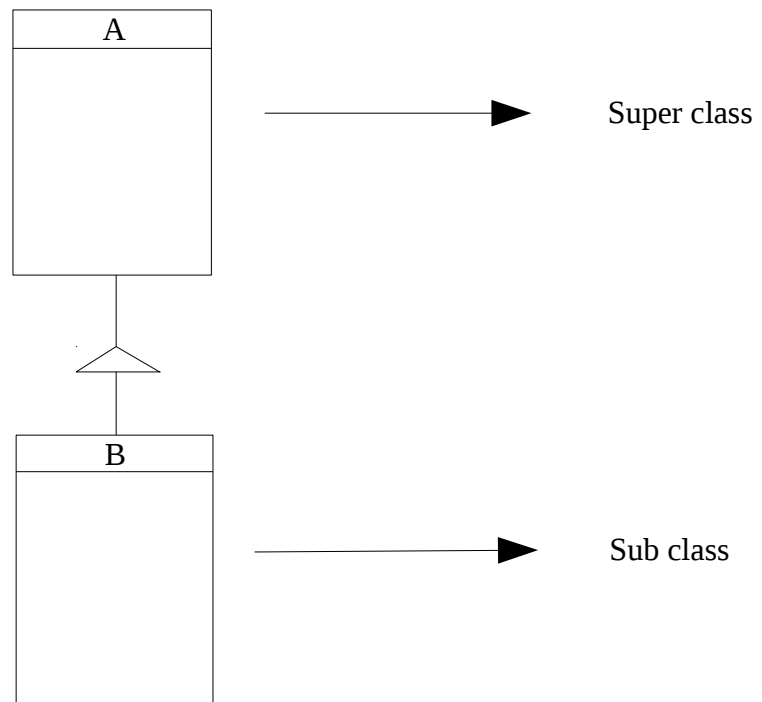


Existence dependency: components cannot exist without being linked to a composite. Child object cannot exist without parent object. Component object/class gets deleted when composite object/class gets deleted.

It can be used for containment hierarchy- components containing components and so on. Ex:



2. Inheritance: notation:



Implicit label name: “is-a”.

Inheritance hierarchy:- sub class of a super class will again have a sub class and so on.

The tip of the triangle always points towards the super class.

Important features:

1. Attributes get inherited: subclasses have the attributes of the superclass implicitly.
2. Methods get inherited: subclasses have the methods of the superclass implicitly.
3. Data which is present at subclass level is exclusive for the subclass.

In the subclass:

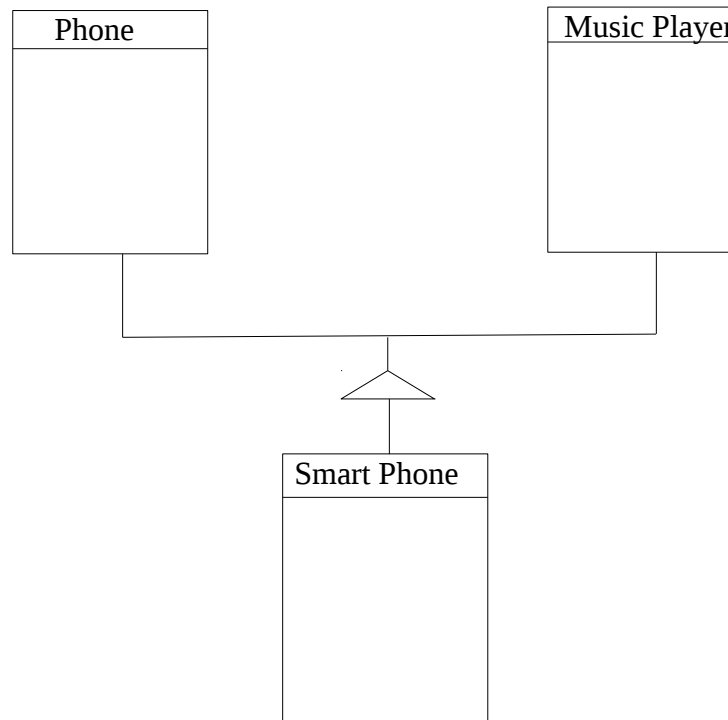
1. More data members can be added.
2. More methods can be added.
3. Methods can be modified.
4. Data members cannot be deleted (i.e.) if there are 4 data members in superclass then there will be those 4 data members in the subclass implicitly and we cannot delete any one of the four data members.
5. Superclass' methods can be changed in the sub class. This process is known as over-riding.
6. The methods inherited from the superclass into the subclass can be

hidden.

Types of Inheritance:

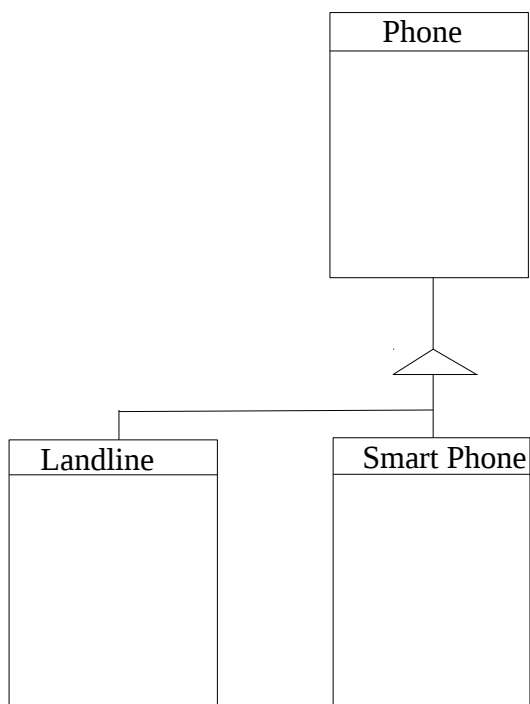
1. Single inheritance.
2. Multiple inheritance.

Multiple Inheritance: A subclass can have multiple superclasses. Ex:



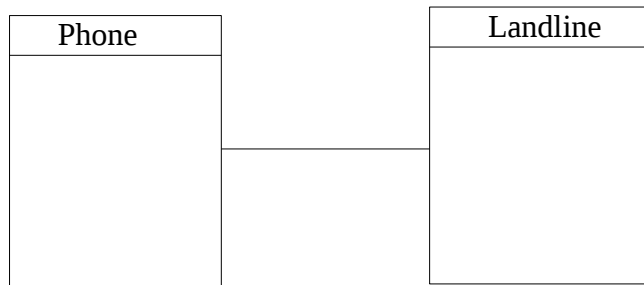
Subclass will have all the methods both from phone and music player.

Single inheritance:- A subclass can have only one superclass.ex:



Difference between Association and inheritance:

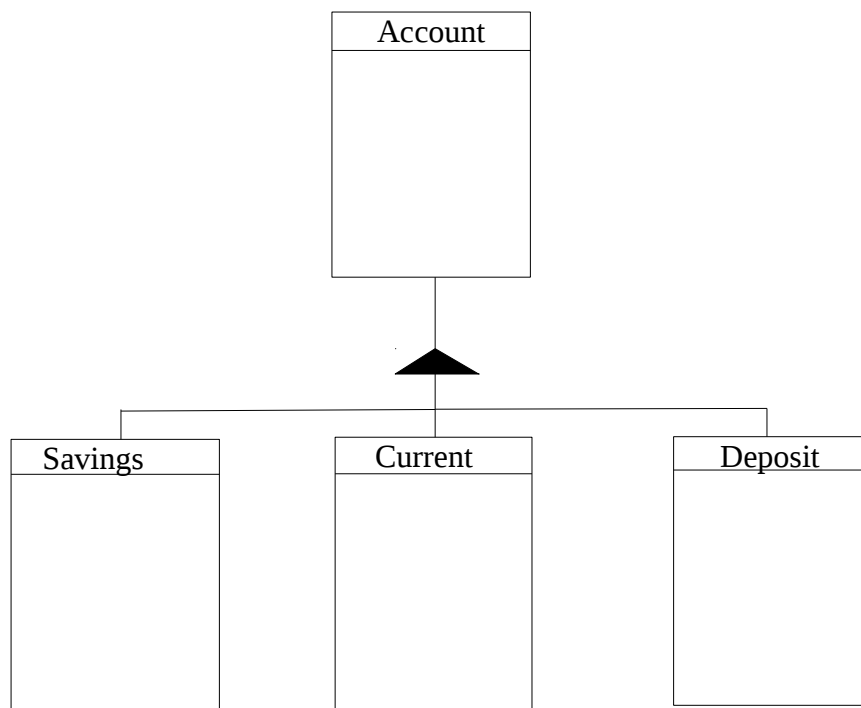
Association:- Ex:



Comparing the association example and inheritance example given before we can draw the following conclusion:

In association, we have to create two objects – for phone and for landline and then link the true objects whereas in inheritance all three steps are done in one step by creating an object of the superclass.

In inheritance there is only one addressable unit for both super class and sub class where as in association the two class objects are addressed separately.



If in the notation of inheritance, the filled triangle says that the subclasses partition the super class. In the above example an account can only be a savings account or a current account or a deposit account, the account cannot be of any other type.

Superclass of which stand alone instances cannot be created is called a abstract base class.ex: `a=new Account()` is not possible.

Guidelines:-

1. Cardinality check:- if the cardinality is anything other than 1 to 0-1, inheritance is

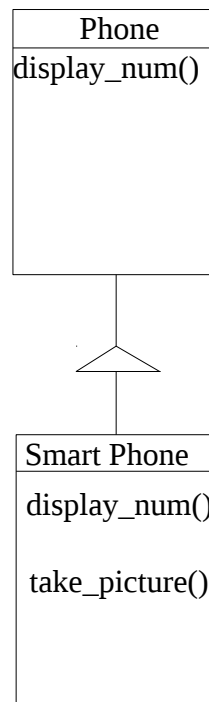
ruled out.

2. one necessary condition for subclass is that the subset property must hold but it is not sufficient condition.

3."is-a" - does it make sense.

Polymorphism:-

The ability to change or morph depending on the object is called polymorphism. Polymorphism is achieved using over-riding of methods in the subclass.



```
Phone *p1 = new phone();
```

```
SmartPhone *p2 = new SmartPhone();
```

```
Phone *p3 = new SmartPhone();
```

```
SmartPhone *p5 = new Phone();//error because SmartPhone is a subclass of  
Phone not super class;.
```

p1->display_num(); //Phone's method is invoked. At compile time it sees Phone type.

p2->display_num(); //SmartPhone's method is invoked. At compile time it sees SmartPhone type.

P3->display_num();//At compile time the compiler sees the Phone type but it binds the actual method of SmartPhone type at the run time.

P1->take_picture();//error is occurred because take_picture() is not present in

phone class.

P2->take_picture();//no error because take_picture() is present in phone class.

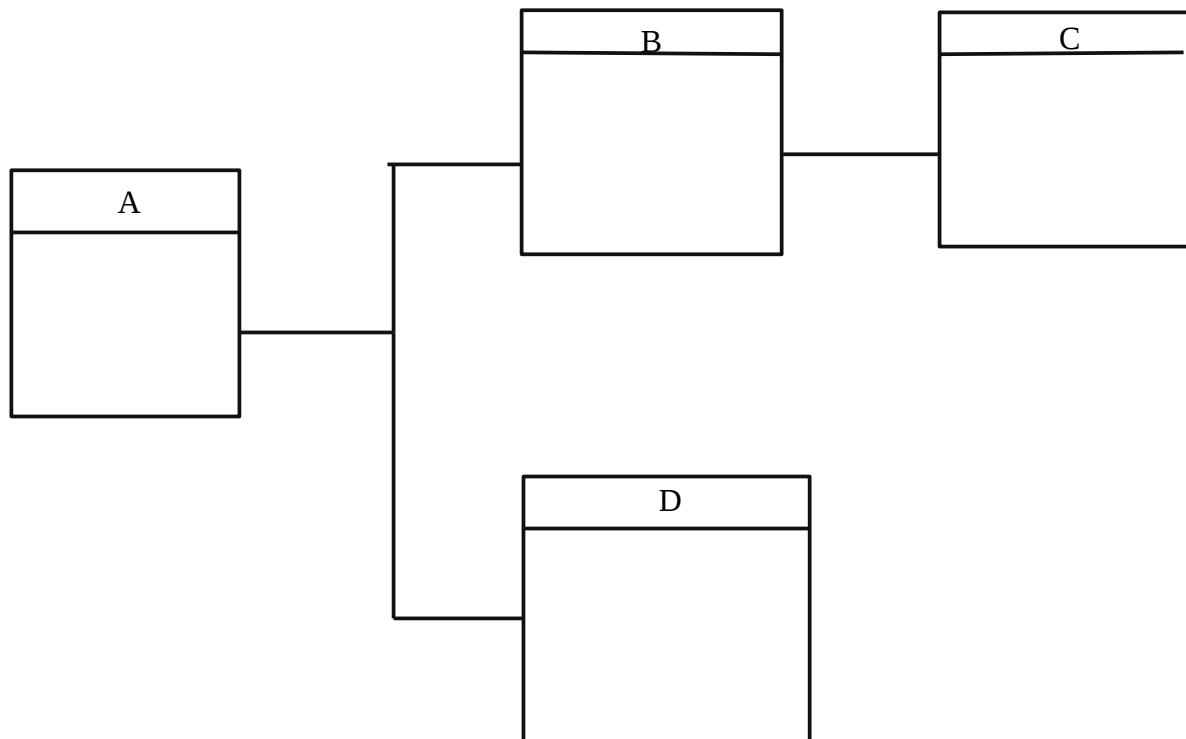
P3->take_picture();//error is occurred because take_picture() is not polymorphic that is take_picture() is not present in the Phone class.

A method is polymorphic when it is present in both parent class and the sub class.

SmartPhone *p4 = p3;

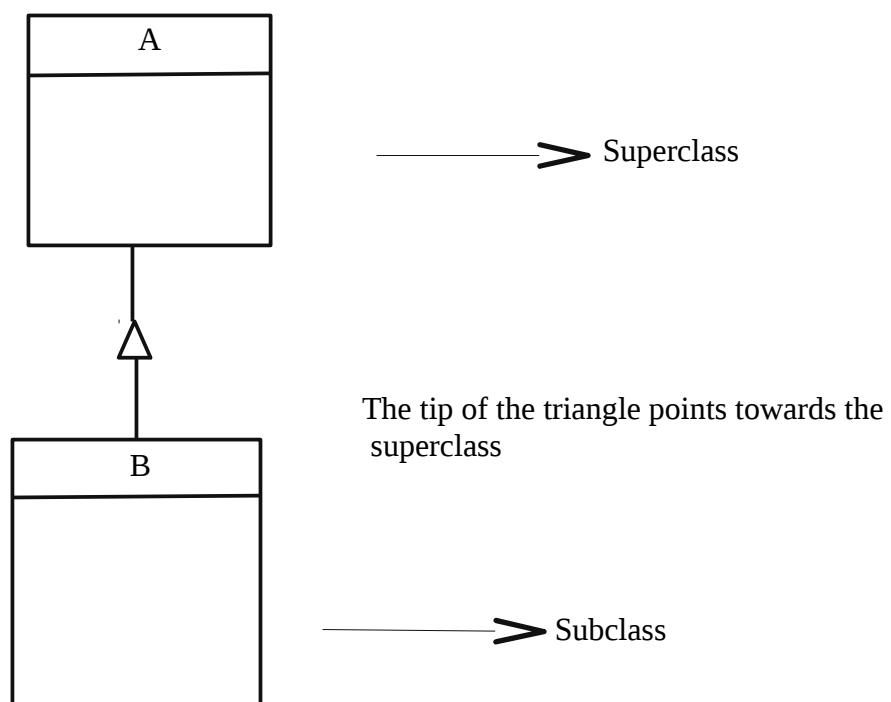
p4->take_picture();//no error

Using composition a containment hierarchy (a larger component made up of smaller components which in turn are made up of smaller components and so on) can be created. In other words a tree of components can be created



Here if an object of class A is deleted the entire tree gets deleted.

Inheritance

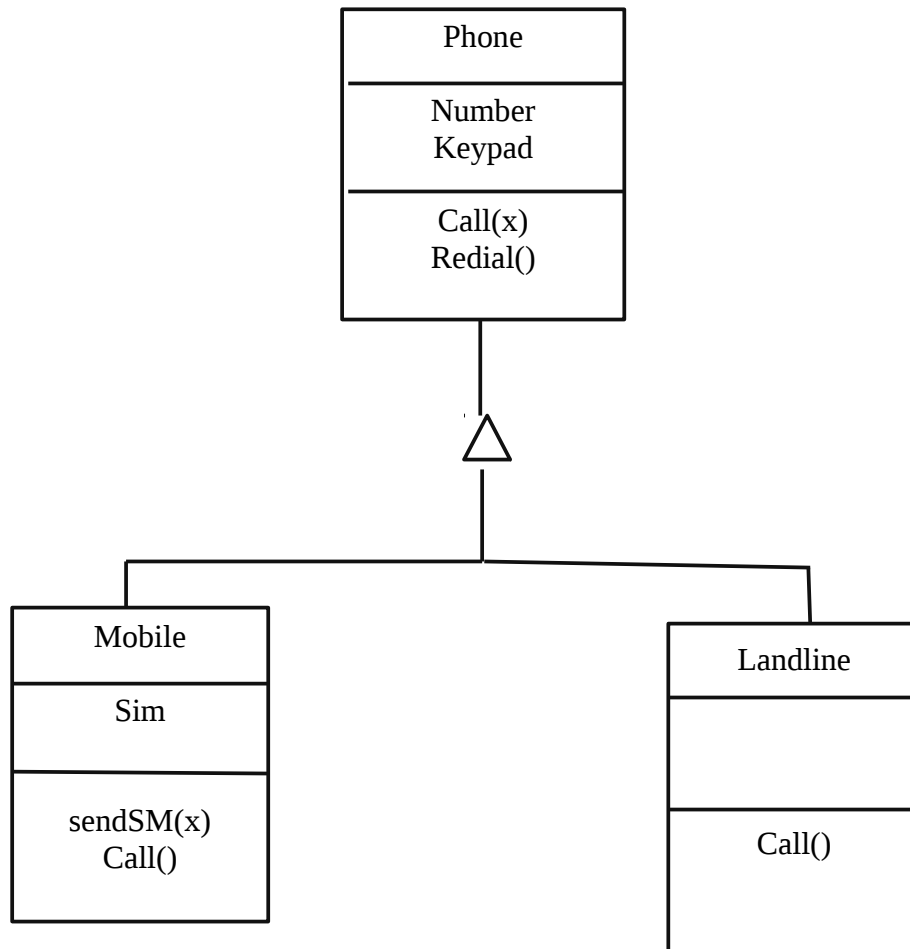


It is a special type of association whose implicit label is “is-a”. Here B “is-a” A.

A superclass can have more than one subclass. An inheritance hierarchy can be built (superclasses can have subclasses which in turn can have its own subclasses)

If B is a subclass of A then

1. the attributes get inherited. Subclasses have the attributes of the superclass implicitly. For example if the superclass has 3 attributes and the subclass has 2 attributes then the subclass effectively has 5 attributes.
2. the methods get inherited. All capabilities of the superclass are in the subclass.



Data members present in the sub level are exclusive to its hierarchy. In the above example, the data member **SIM** is exclusive to the Mobile hierarchy; that is, it is implicit to the subclasses of the Mobile superclass. It is not implicit to the subclasses of the Phone superclass. It isn't present in the Landline subclass.

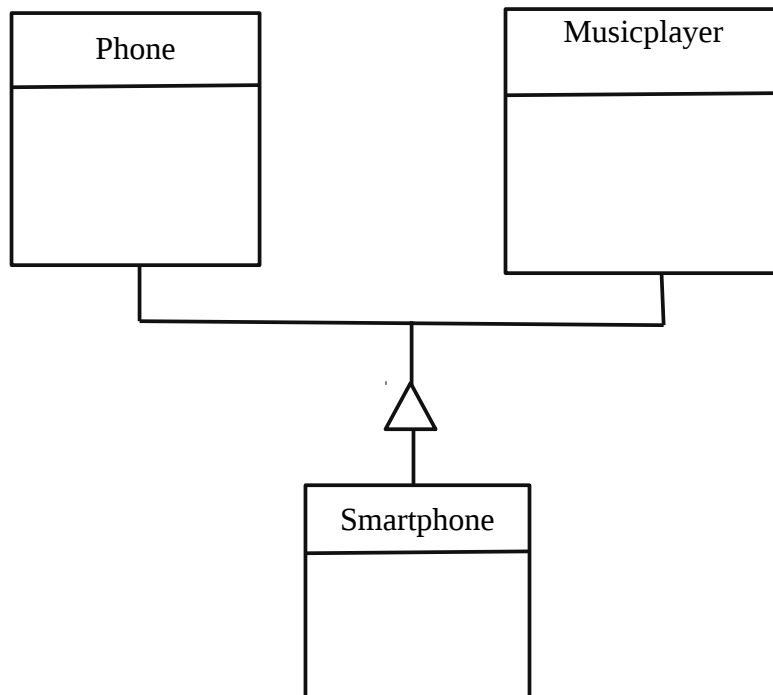
Methods are exclusive to the subclass. The subclass is not confined to the inherited data members and methods. The subclasses can have additional data members and methods. They can modify inherited methods but inherited data members can neither be modified nor deleted.

Methods can be **overridden**; that is, the entire method can be rewritten in the subclass. In the above example, the method **Call** is overridden. Inherited methods cannot be deleted but can be hidden.

Inheritance is of two types:

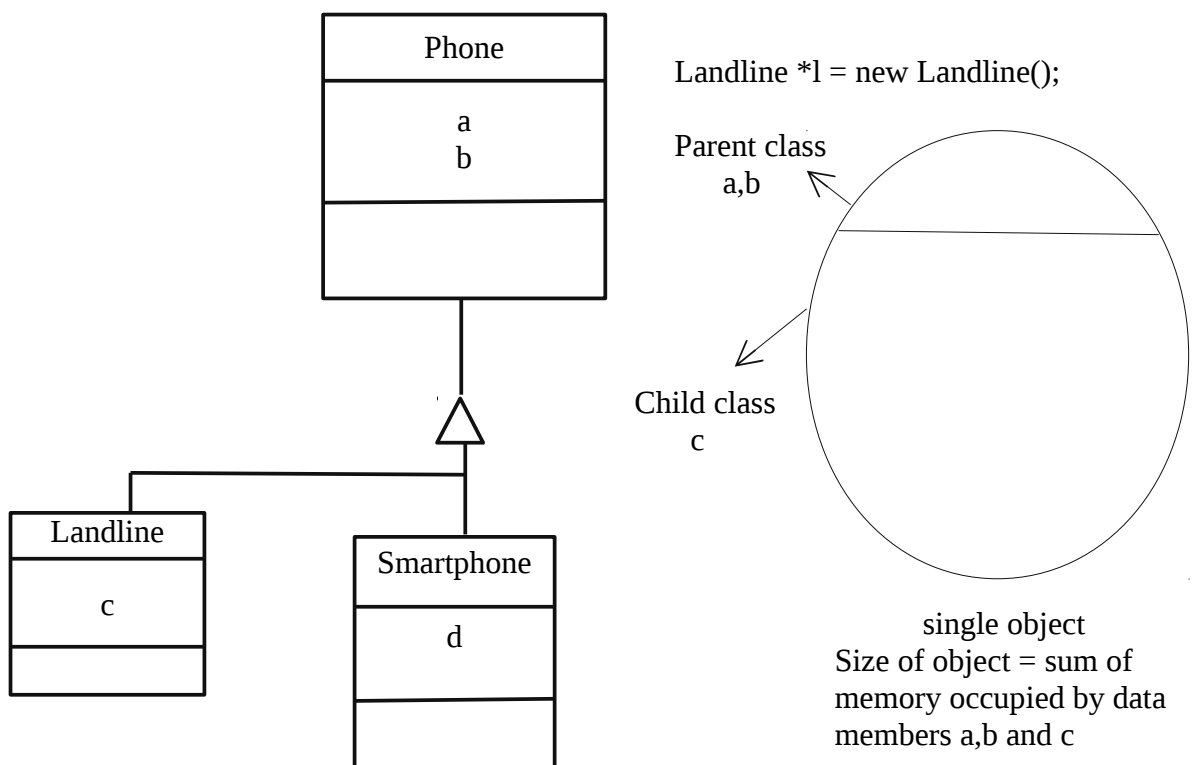
Single Inheritance: The subclass can have only one superclass in this type of inheritance.

Multiple Inheritance: The subclass can have more than one superclass in this type of inheritance. There is no restriction on the number of superclasses.



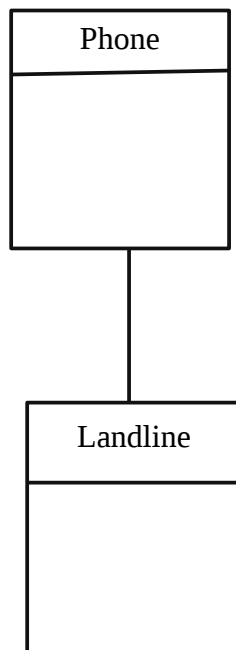
Data members and methods in the subclass get inherited from all its superclasses.

C++ supports while Java doesn't support multiple inheritance. It is generally avoided.



Object is a single addressable unit that is allocated a single chunk of memory. When inheritance is used

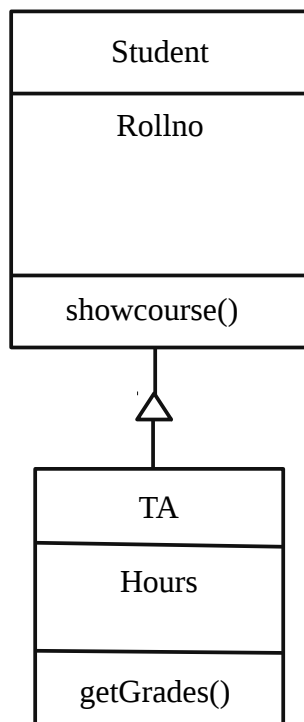
- 1.The whole chunk of memory is allocated in one shot. That is one object encapsulates the link between the subclass and the superclass.
- 2.Linking is implicit.
- 3.Memory allocation can be done in a single operation.
4. On deleting an object of the subclass the entire link is deleted.
5. Cardinality between superclass and subclass has to be 1:1 or more accurately 1:0..1.



In association

1. Two objects are created.
2. Explicit linking is required. In this example,an object of phone and object of landline has to be created and explicitly linked.
3. Cardinality can be one to one, one to many etc.

We can create instances of the superclass alone. In the following example we can create student objects which doesn't have to be a TA object.



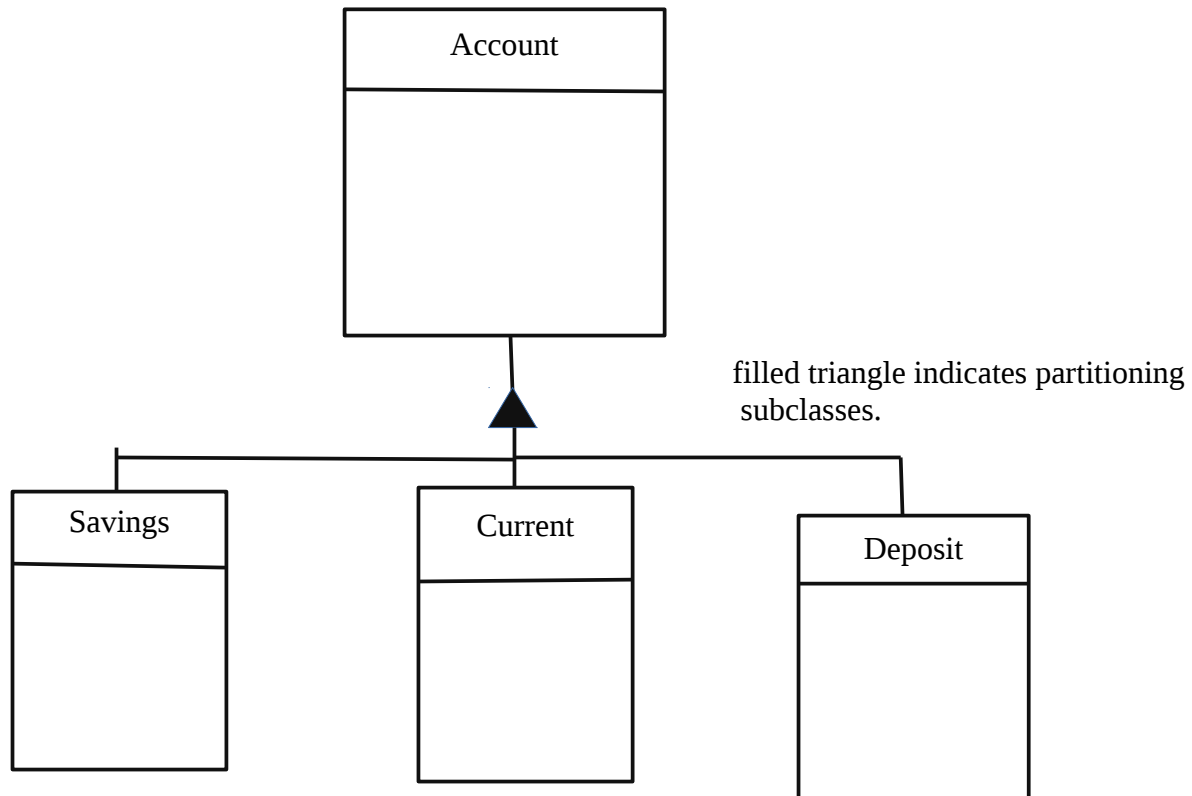
```
s = new Student(R001);
ta = new TA( R002,20);
s-> showcourse(); //invoked on s object
ta ->getGrades(); //invoked on ta object
ta->showcourse();//invoked on ta object
s->getGrades(); //error not allowed
```

From the above example it is clear that methods are invoked in the context of the object.

Set of instances of student is a superset of the set of instances of TA.

The set of instances of the superclass is a superset of the set of instances of the subclass.

Superclass for which standalone objects cannot be created is called abstract base class.



In the above example if there is a restriction that stand alone instances of the Account class cannot be created that is object should either be an instance of Savings class, Current class or Deposit class but not both then the Account class is called abstract base class.

```
a=new Account();//not allowed
```

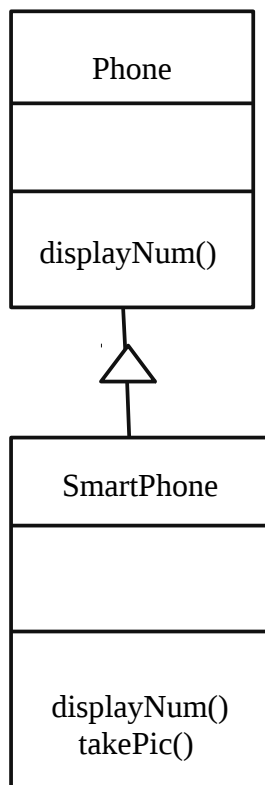
Guidelines for Inheritance

1. cardinality check
2. subset property
3. "is-a"

Implications of modelling as inheritance is one object can get only one instance of every attribute. Consider two classes Phone and Camera, it fails the cardinality check. A phone can have two cameras hence inheritance cannot be used in this case.

Polymorphism

Assuming that the subclasses is non partitioned polymorphism is achieved using overriding methods in subclass.



```

Phone *p = new Phone();
SmartPhone *p2 = new SmartPhone();
Phone *p3 = new SmartPhone(); //not allowed if inheritance wasn't used
SmartPhone *p4 = new Phone(); //not allowed
  
```

```

p1->displayNum();
p2->displayNum();
p3->displayNum(); //doesn't invoke phone's method, SmartPhone object is created.
  
```

Compiler doesn't resolve which method to call at runtime. It binds actual method to this call at runtime. It is an example of dynamic binding.

```

P3 = new Phone();
p3 -> displayNum(); //Phone's method is invoked
  
```

In the above two cases method call is same, but different methods are getting invoked. Method dynamically gets morphed (morphs/changes depending on object) during compile time.

```

p1->takePic(); //error
p2->takePic(); //allowed
p3->takePic(); //error
  
```

`takePic()` is not polymorphic. `takePic()` isn't present in the superclass. At compile time the compiler compares with type of pointer and checks if the method the object is trying to invoke is allowed. Method can be polymorphic if it is present in the superclass and subclass.

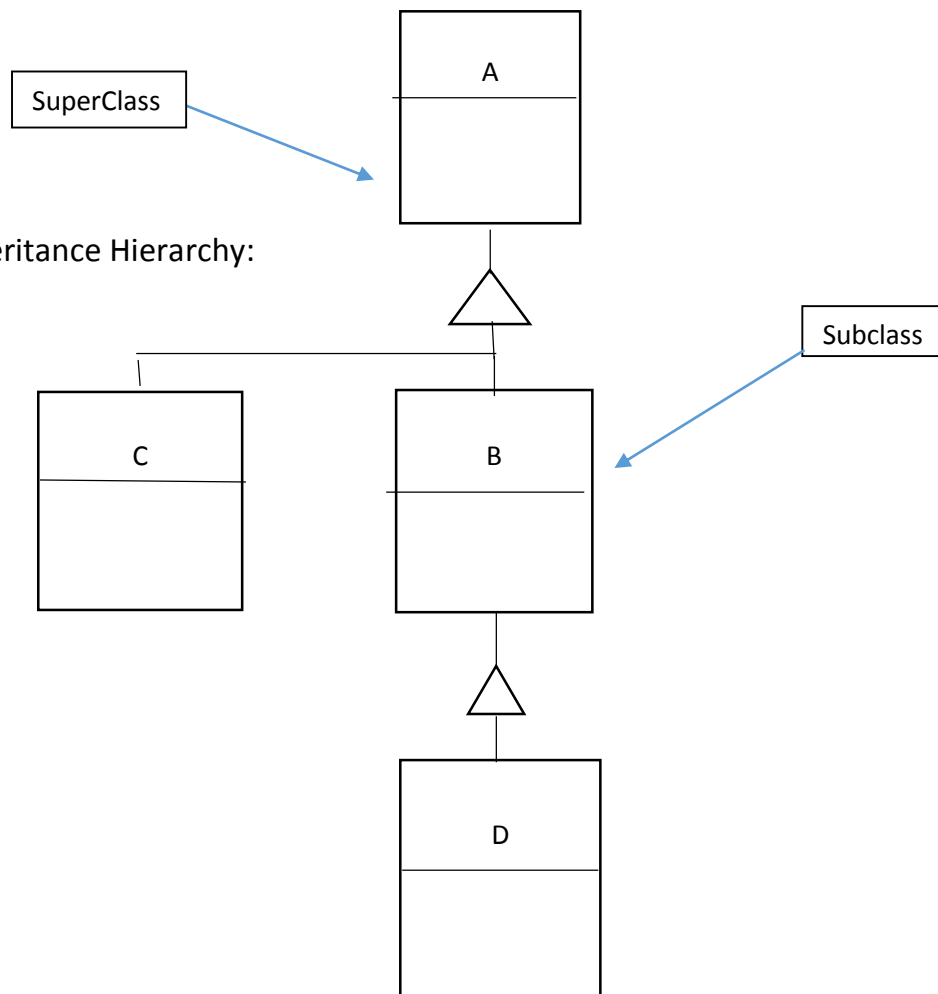
```

SmartPhone *p5 = p3;
p5->takePic(); // allowed since p5 is object of type SmartPhone
  
```

Pointer to a parent class is compatible with pointer to subclass. Polymorphism hides the implementation specifics of subclass and makes it more extensive.

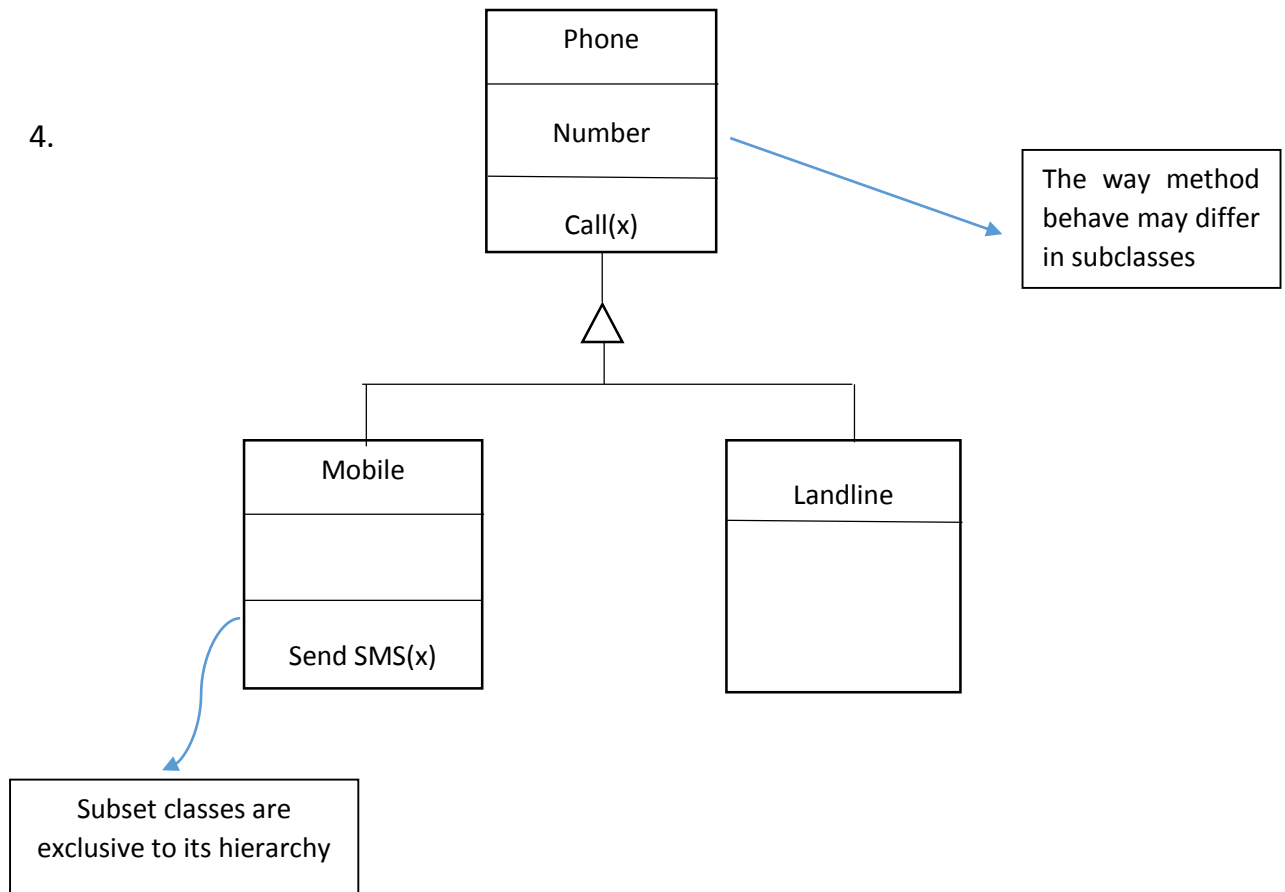
S.Sharanya
IMT2013038

1. Inheritance allows us to define a class in terms of another class.
 - Special type of association in which we don't put label of association and here label is B "is-a" A.
 - Tip of triangle is always towards the superclass.



3. What does inheritance mean?
 - Attributes get inherited: subclasses have the attributes of the superclass "implicitly".
 - Methods get inherited: this property makes it more powerful.

4.

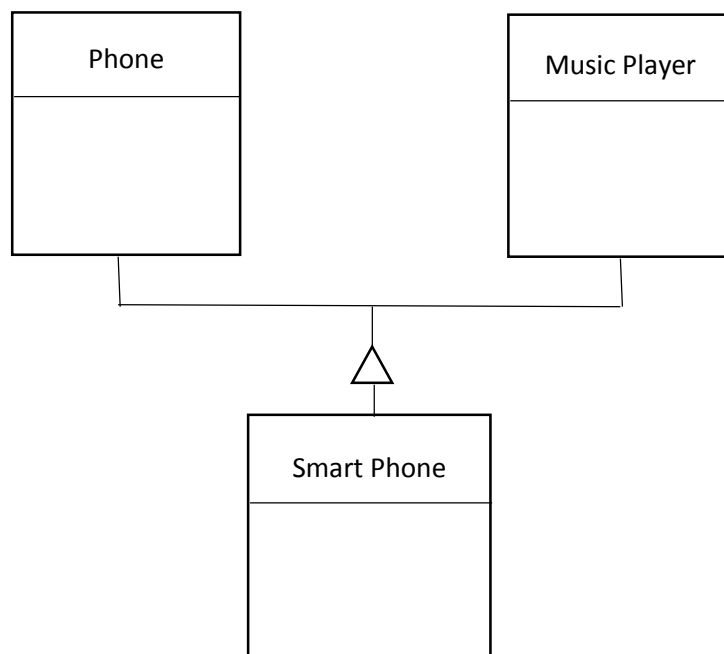


- Can't modify data members in subclasses
- Can delete methods
- Only methods can be overridden in subclasses and data can't.
- Can only add data members

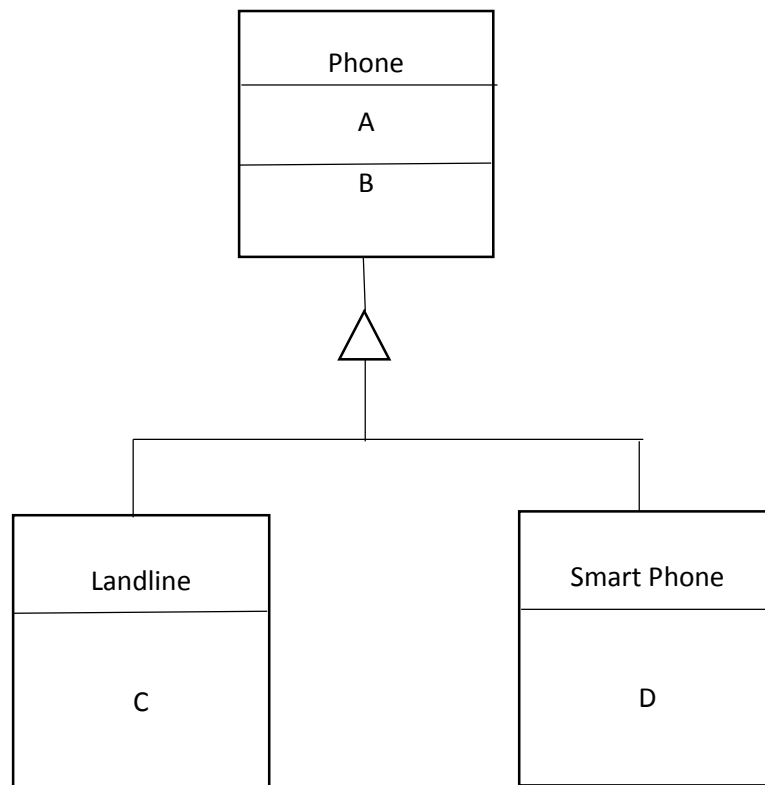
5. There are broadly two categories:

- Single Inheritance : a subclass can only have one superclass
- Multiple Inheritance : a subclass have more than one superclass

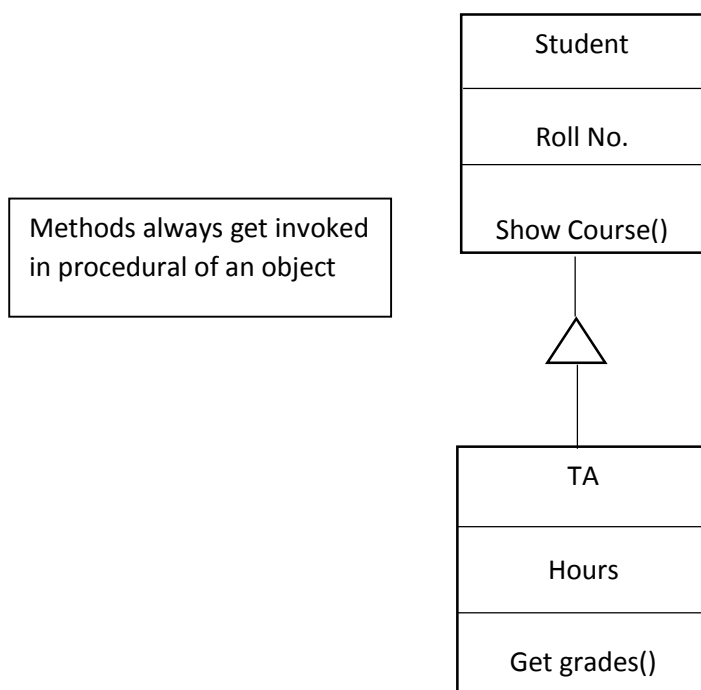
Eg. of multiple inheritance (there is no restriction on number of parents)



Eg: of single inheritance:



6. Whole chunk of memory is allocated in one shot when things are inherited and linking is implicit.
7. When dealing with inheritance we are dealing with one object so when object is deleted whole thing is deleted.
8. Cardinality between superclass and subclass is one to one or more accurately zero to one) because relating objects.



9. Student → Show Course() doesn't give an error

TA → Get Grades () doesn't give an error

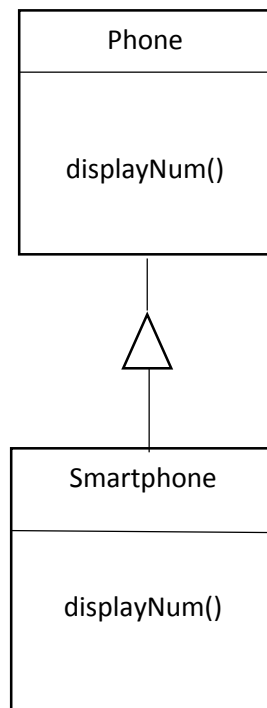
TA → Show Course () doesn't give an error

Student → get grades () gives an error

10. Guidelines:

- Subset property should hold : subset property should hold but it is not a sufficient condition.
- "is-a"
- Cardinality check: When one to many is found inheritance is out of context. In inheritance one to one is the condition.

11. Polymorphism: (achieved using over-riding of methods in sub-classes.



```
Phone * P1 = new Phone();
```

```
Smartphone * P2 = new Smartphone();
```

```
Phone * P3 = new Smartphone();
```

P1 → display Num(); : Here Phone method gets invoked (P1 is an instance of Phone)

P2 → display Num(); : Smartphone method gets invoked

P3 → display Num();

P1 → take picture(); : gives an error

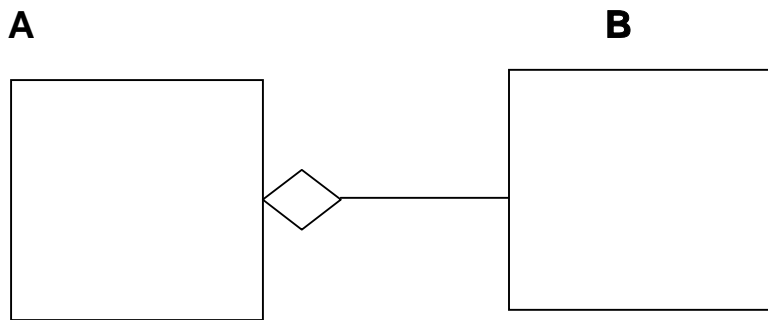
P2 → take picture(); : doesn't give an error

P3 → take picture(); : gives an error (take picture is not polymorphic it's not their in superclass)

Date: 19th Jan 2015:

Aggregation:

Notation:

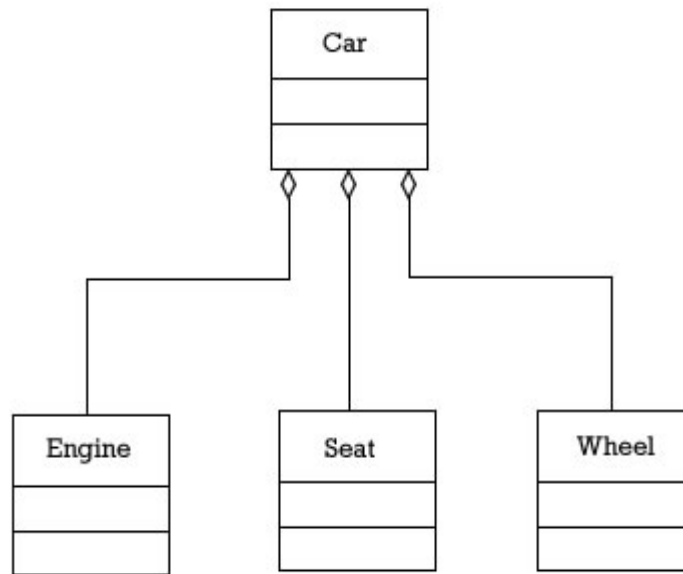


A has a B

Aggregation can be viewed as a special association with name of association as "Has a". In all the practical purposes this is same as association but it doesn't have any existence relationship which means, Let us suppose there are two classes A and B and they are interrelated by aggression which means A has a B if A is deleted then B won't be deleted and classes can exist individually or they are linked to only one class which means one to many relationship

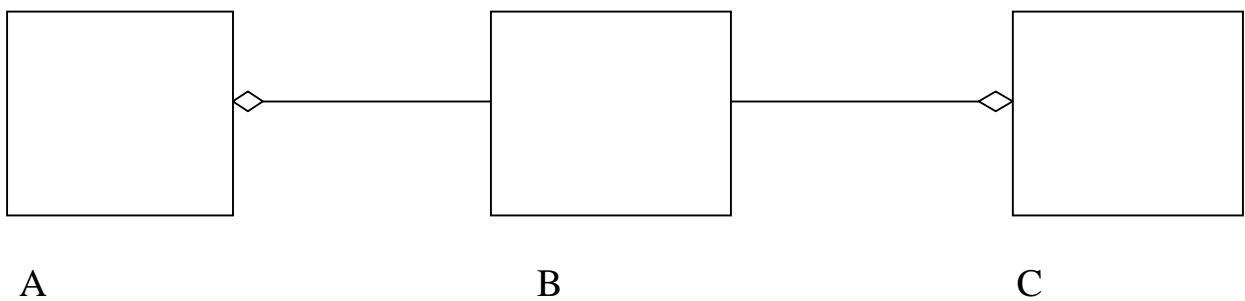
Ex: 1) A car class can be defined to contain other classes such as engine class, seat class, wheels class etc. The car class can define an engine class as one of its attributes. But the classes engine, seat, wheels can exist individually.

2) A camera can exist as a single entity or it may associate with others to form a big object



Instance of class cannot be interrelated to many classes by aggregation it can be only be related to one class

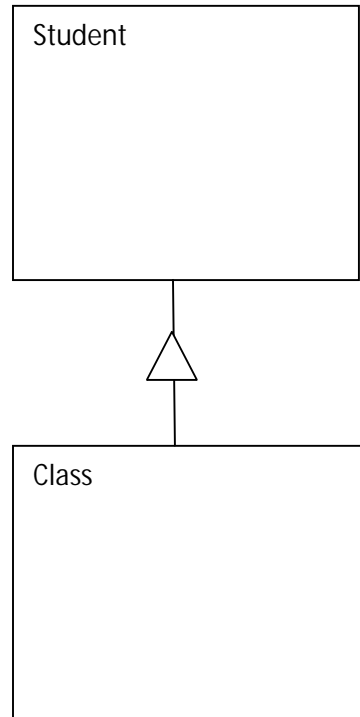
Wrong method:



The aggregation is hard to implement whereas association is easy to implement

Implementation of Interclass Relationships:

Inheritance:



C++ Implimentation:

```
class Student{  
  
};  
  
class TA : public Student{  
  
};
```

Java Implimentation:

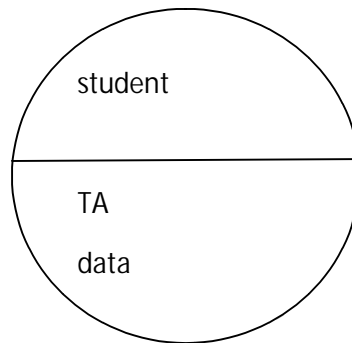
```
public class Student{  
  
};  
  
public TA extends student {  
  
};
```

Python Implementation:

```
class student:
```

```
class TA(student):
```

Memory Allocation:



Let data members of student occupies 10 bytes and TA object data members occupies 20 bytes if you declare TA object then 30 bytes memory was allocated function methods were declared then it will occupy a particular space in memory and if a method was called it will call the methods in that memory.

Memory allocation of methods are allocated class based not object based. Let's suppose a TA class has two methods each of size 1000 bytes and 500 bytes and data members of size 30 bytes if you call object of type TA multiple times memory of data types is allocated every time but the methods are called from the memory every time.

The notations of public private doesn't effect on the memory allocation it just controls the scope and access of the data members.

Specialization:

It's a strategy to identify the Inheritance phenomena. If the Identification is done on the basis of the Top-Down approach method is called Specialization.

Identification of the sub classes using the super class

Ex: Bank account can be classified as savings account and current account

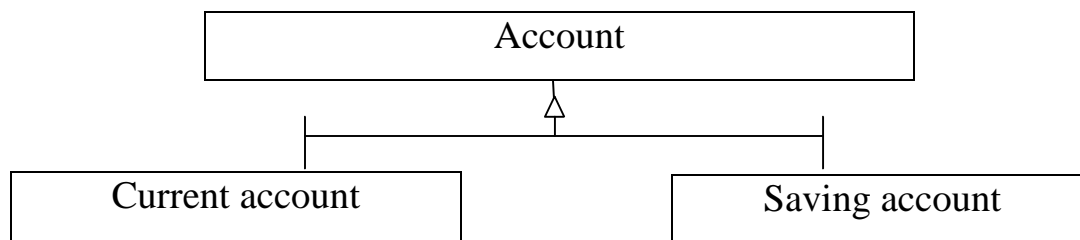
Generalization:

Another type of strategy for Identifying inheritance

Current account
Accountno, odlimit
Printstatement()

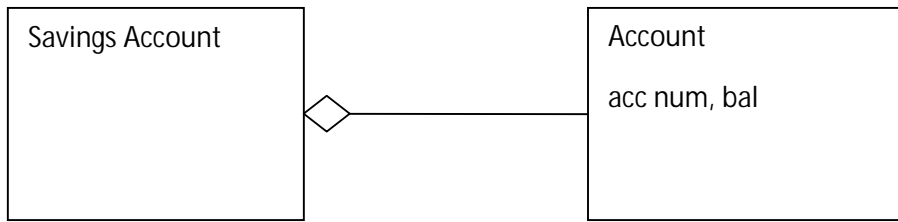
Savings account
Accountno, bal, minbal
Printstatement()

By identifying the sub classes identifying the super class and finding the inheritance link



Why can't we see above as association?

When we use association or composition we lose the property of overriding or polymorphism can't be done. It can be done only in inheritance. Composition will have one to many association which will violate property of inheritance.



C++ implementation:

```
class Savings{
    account a;

public:
    printstatement();

};
```

Implementation of print statement() in composition:

```
savingsAccount::Printstatement()
{
    a.printstatement();
    cout<<minbal<<endl;
    ....}
```

Implementation in inheritance:

```
_class savingsAccount{  
  
    min_bal;  
  
    printstatement(){  
  
        cout<<acc_num<<endl;  
  
        cout<<balance<<endl;  
  
        cout<<min_bal<<endl;  
  
    }  
}
```

the above will work if the data members of super class are protected.

```
Account::Printstatement(){  
  
    cout<<min-bal<<endl;  
  
}
```

since both look similar then why inheritance

if there is function which takes vector of account objects and print the statements

in inheritance it can be done as follows

```
for(elemets in vector) aptr{  
  
    aptr->printstatement()  
  
}
```

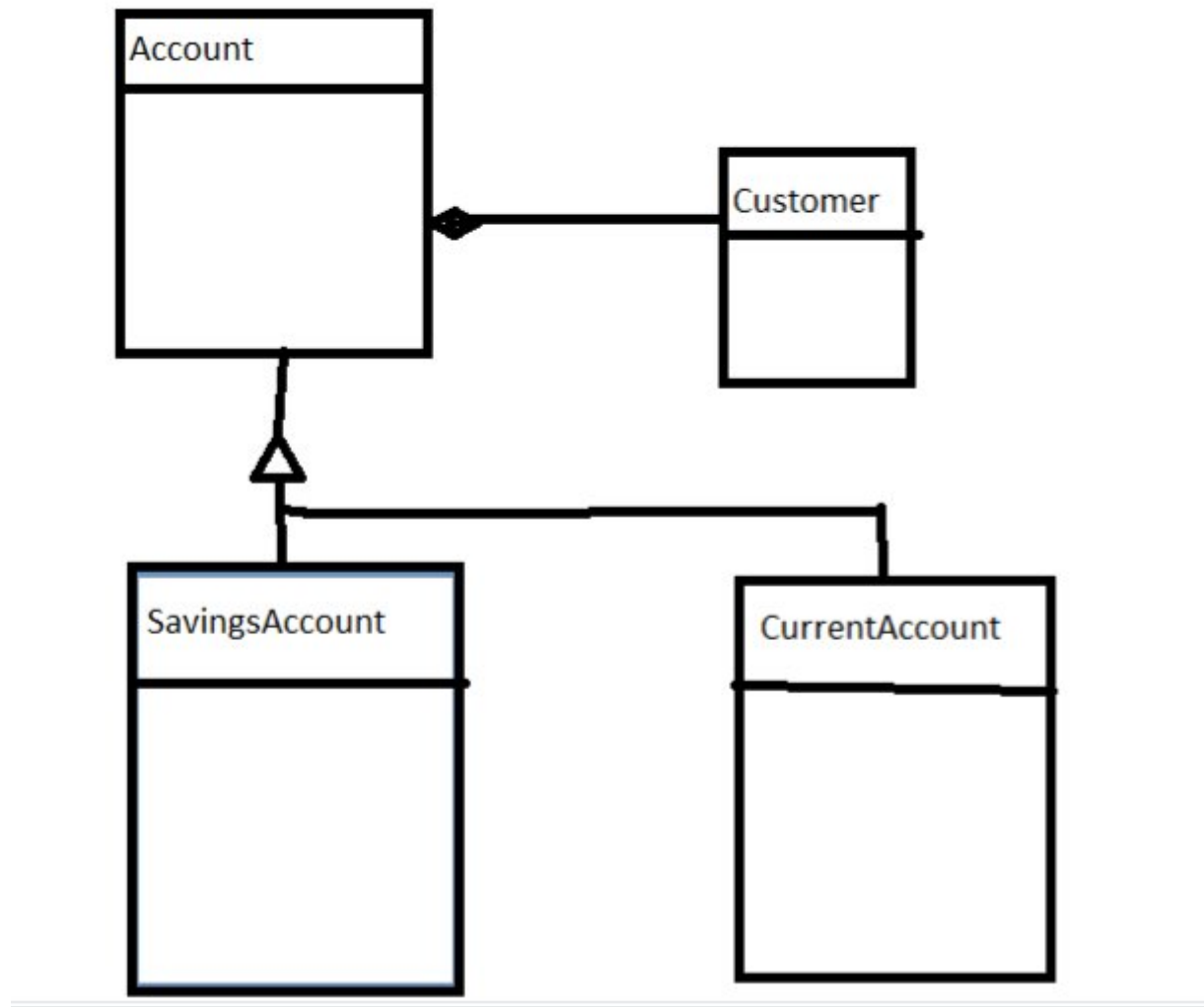
it will print all the details in print statement eventhough new things are added it will print all automatically but in composition you have to pass new vector of that thing and then you have to print.

ex:void Printallstatements(vector<cur acc*>v1, vector<sav acc*>)

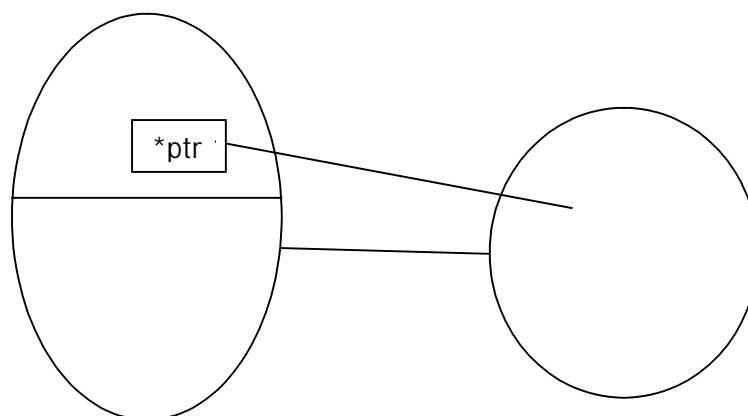
```
{  
for(each curacc ptr){  
curract->printstatement();  
}  
for(each savacc ptr){  
savaccptr->printstatement();  
}  
};
```

Object Graph:

Network of instances is called object graph



During inheritance along with the data and methods relations are also inherited
if there is a pointer in the super class it will be inherited in the subclass



Interfaces:

It is the collection of indented behaviors and unimplimented these are generally used in the GUI

Interface do not have data members

Ex:

GraphicsInterface

showpicture();

clicknext();

drawwindow();

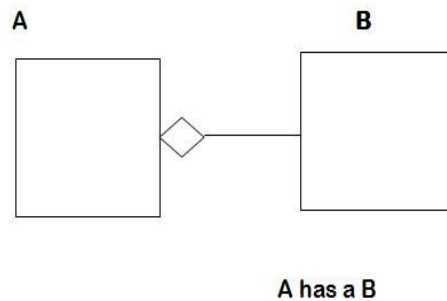
```
_7inchdisplay implements graphics interface{  
}
```

```
20inch-monitor implements graphics interface{  
}
```

it will be implimented according to the classes 7inchdisplay and 20inchmonitor

Aggregation:

Notation:



- 1) It is a type of association with the cardinality 1 to many.
- 2) A "has a" B
- 3) The difference between the composition and aggregation is that in aggregation there is no existence dependency of B on A i.e even if you delete A, B can exist independently.
- 4) Semantics of this one is hard to implement
- 5) There can't be two objects A and C such that both the objects have Aggregation relation with the same object B (A "has a" B and C "has a" B is not possible).
- 6) There is no specific syntax for implementing aggregation.

Implementation of Inheritance :

My super class is Account and my sub class is SavingsAccount.

In c++:

```
class Account{
//data members
//methods
};

class SavingsAccount : public Account{
//extra data members
//extra methods
};
```

// if you want to modify a method of the super class in your sub class you need to declare that method as virtual in the super class

In python:

```
class SavingsAccount(Account)
```

In java:

```
public class Account{
}

public class SavingsAccount extends Account{
}
```

```
//
*If you want to prevent the sub class to override a method of a super class use the
keyword "final" as show below.
//
    public class Account{
        final void test();
    }
```

Memory allocation for an object :

Suppose if there is a super class Account and a sub class SavingsAccount to it then when you create a object of type savings account the memory is allocated to all the data members of sub class as well as the super class (it doesn't depend whether they are declared public or private or protected).But memory is not allocated for methods. In a way the memory is allocated to all the data members of each object separately but not for the methods.

Ex:

```
account
data members :20 mb(assume the size of all data members is 20mb)
methods : 500mb
```

```
savings account
data members : 30 mb
methods :1000 mb
```

so when you create an object of type savings account a memory of 20+30 is allocated but not 500+1000+20+30

Two ways to Identify Inheritance:

1. specialization

Create a general purpose class with all the general data members and general methods. Now look at the special cases and all the additional data.Create a new classes for all these special cases. It is a top down approach.

2.Generalisation

Create all the classes. Now look at the common data members and common methods and create a super class with these data members and methods. This is a bottom up approach.

The main advantage of Inheritance over composition is polymorphism overwriting. If you are not using Polymorphism then using Inheritance doesn't make any major difference.

Interfaces & realizations:

Interfaces:

Interface is a collection of intended behavior

It has only methods.

No data members

(used in GUI programming)

Realization

These class will define the methods of interfaces in the way it is required.

ex: code

```
graphics Interface
    show Picture();
    Click next();
    draw Window();
```

class 7 inch display implements this graphics Interface

In realization the class will have data members.

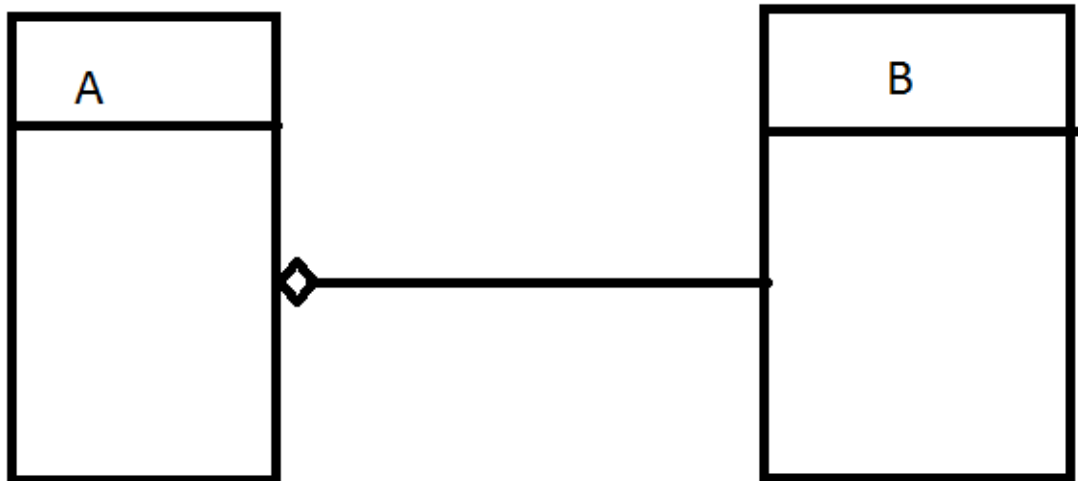
Multiple interface is possible in some languages and not in others.

They can also have extra methods which are not a part of the Interface.

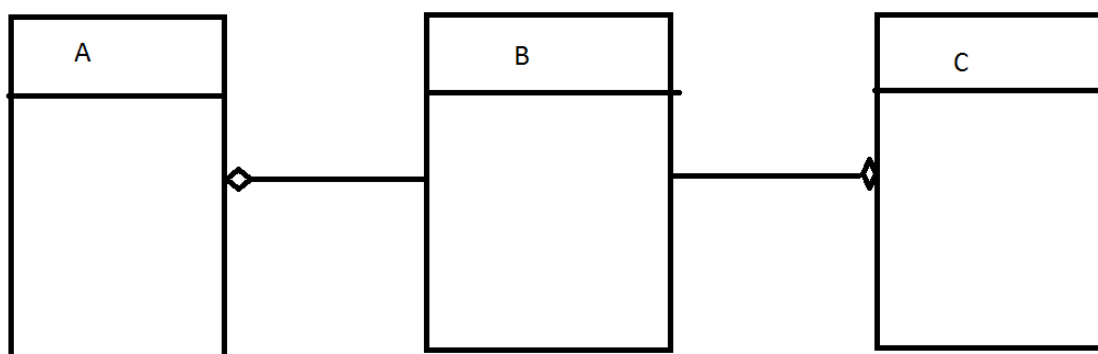
In java you can not implement multiple interface.

C++ does not have a built in concept of interface. But you can implement it using abstract classes.

AGGREGATION:

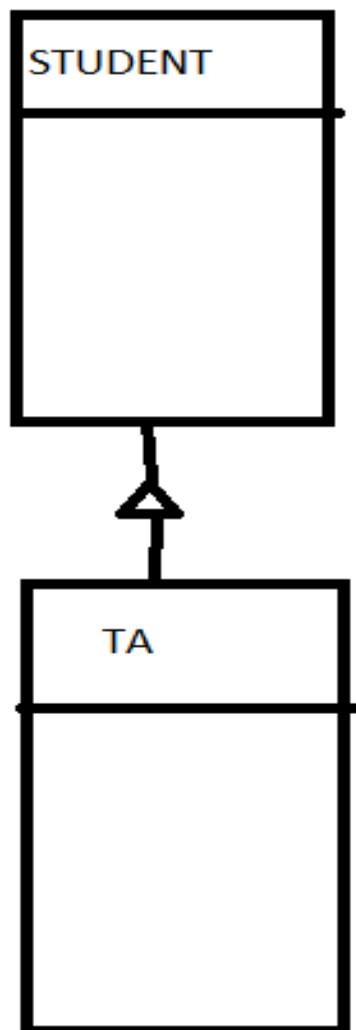


If we think Aggregation as a special form of Association then it is of the form A “has a ” B. It is similar to composition. The only difference between Composition and Aggregation is “there is no-existence dependency” of B on A. When we delete object of A, B does not get deleted. In this sense it behaves very much as Association. The cardinality is 1 to many. Difference between Association and Aggregation is, a component can belong only to one A or can stand alone. This concept is similar to composition but it is different from Association.



The above case is not allowed in case of Aggregation because both C and A has a B which is not possible in Aggregation. If there are individual objects we can make a component out of it. The logic of composition is easy to implement but, the exact logic of Aggregation is hard to implement. There is no real way to implement it. Direct support of programming language is present in both inheritance and composition but it is not present in Aggregation.

IMPLEMENTATION OF INHERITANCE:



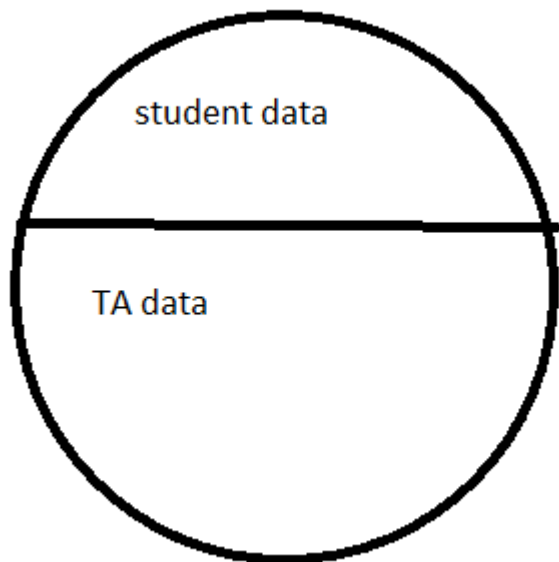
The syntax to implement inheritance in C++ is

```
Class TA : public Student{  
}
```

When we create object with particular structure like

TA x;

How does the object get created?



The data occupies memory but not methods. Let us consider that Student data occupies memory of 10 bytes and TA data occupies 20 bytes. When we create TA object it occupies 30 bytes because of inheritance. Now, if we consider Student class occupies 500 bytes method and TA class occupies 1000 bytes method. Then 1000 bytes is the overhead of TA class itself and 500 bytes is the overhead of Student class. In Imperative Programming the method is per class basis. In terms of public, private and protected; they do not show any bearing on memory allocated. On the other hand they are used only for the scope of the object. It doesn't mean that space is allocated only if it is protected.

How do you recognise the Inheritance Hierarchy?

Specialisation: It is one of the strategies to identify Inheritance. This strategy adopts the process of identifying inheritance in a top-down manner. It identifies if this class of students have other students who have special behaviour.

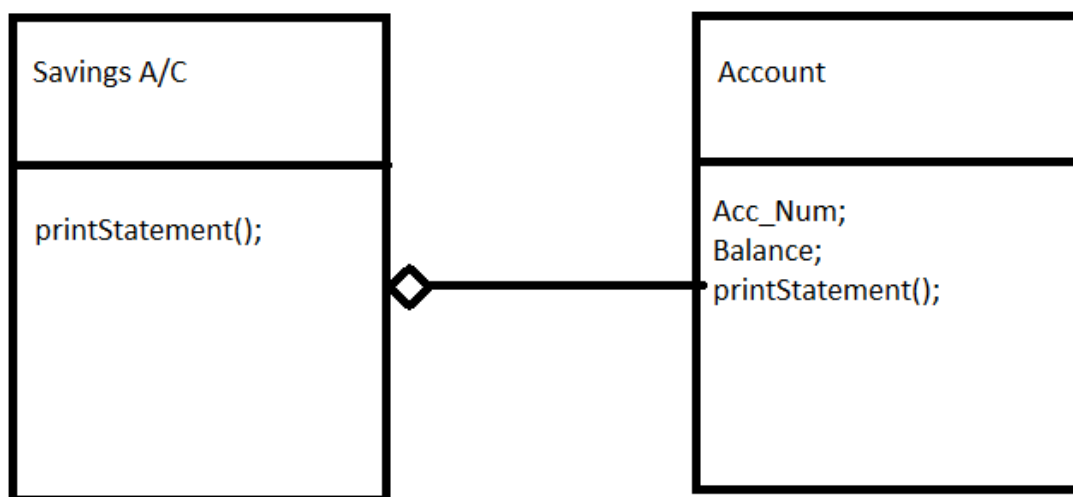
Generalisation : This is also a strategy to identify inheritance. It follows bottom-up method. It checks if both the classes have commonalities and place those data members in another class. By doing this we can avoid repetitions.

Polymorphism works only when we use inheritance.

IMPLEMENTATION OF COMPOSITION:

Inherently composition can be 1 to many but implicitly it has to be 1 to 1.

C++ way to implement composition for the below example is



```
Class SavingsAccount{
    Min_balance;
    Account a;

public:
    printStatement();
}

Savings::printStatement()
{
```

```

        a.printStatement();

        cout<<min_balance<<endl;
    }

```

Here, min_balance can be accessed only by SavingsAccount.

When we do it as inheritance.

printStatement(); in Account class should be virtual

```

class SavingsAccount{
    min_balance;

    public:
        printStatement();
        {
            Cout<<Acc_num<<balance<<min_balance;
        }
}

```

This is possible when the data members of Account class are protected.

If they are not protected then the syntax is

```
Account::printStatement();
```

```
Cout<<min_balance;
```

Syntax to print all the statements using inheritance is

```
Void printAllStatements(vector<Account*> alist)
```

```

{
    for(Range for on a list) aptr
    {
        aptr-> printStatement();
    }
}

```

```
    }  
}
```

Syntax to print all the statements using composition is

Void

```
printAllStatements(vector<CurrentAccount*>v1,vector<SavingsAccount*>v2)
```

```
{  
    for(each CurrentAccount_ptr)  
    {  
        CurrentAccount_ptr->printStatement();  
    }  
    For(each SavingAccount_ptr)  
    {  
        SavingsAccount_ptr->printStatement();  
    }  
}
```

If we want to exploit polymorphism then we should prefer inheritance over composition.

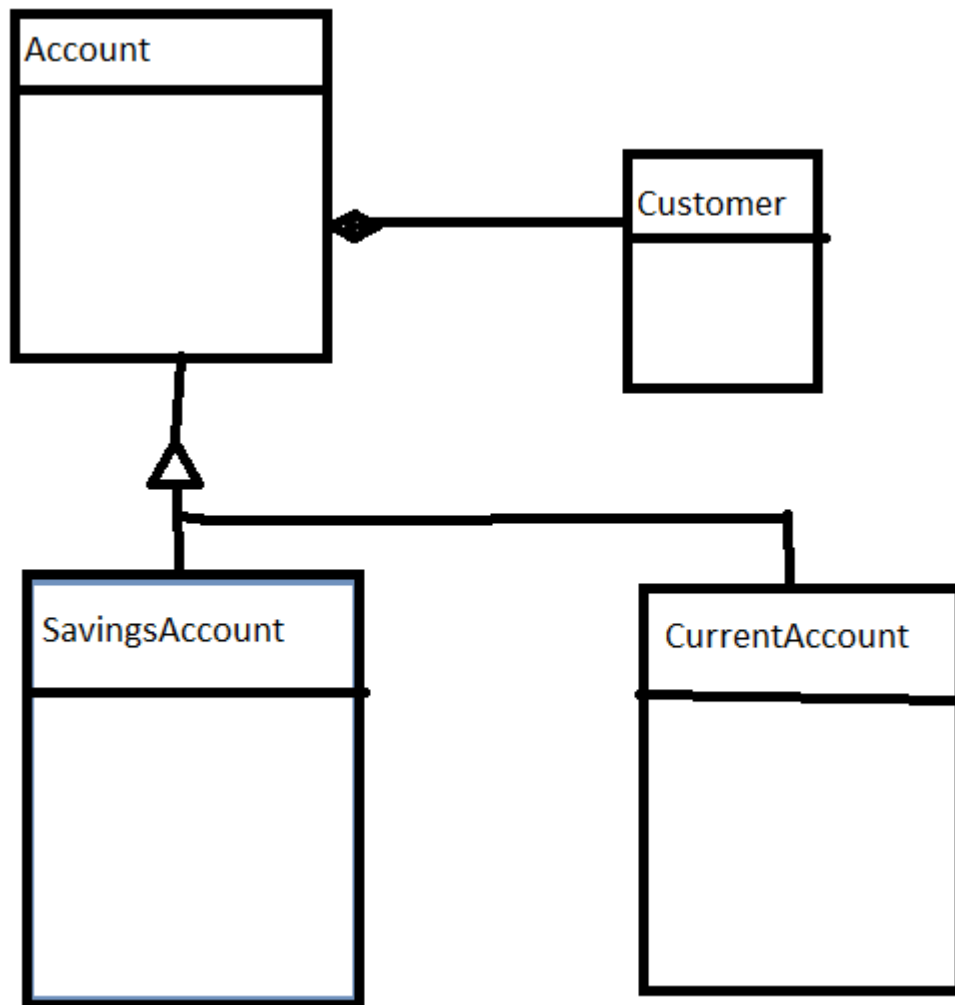
In C++ to override we should virtual before the methods.

In Java implicitly it will override, we should say explicitly not to override.

If virtual is not there in inheritance then always it calls the same class. Main benefit of inheritance is that it enables polymorphism.

OBJECT GRAPHS:

Object graphs plays a crucial role in Networking instances. Structure of object graph will be different compared to the actual class



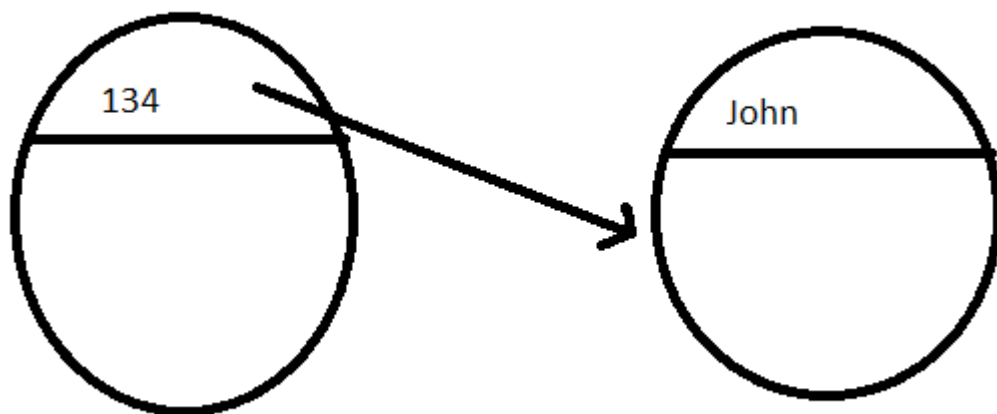
```
Account *a=new SavingsAccount(134);
```

```
Customer *c=new Customer("John");
```

```
a->addCustomer(c);
```

One object is created but it contains both the data.

The object graph is



Inheritance will inherit data, methods and relationships. In the above example Customer * inherits with SavingsAccount and CurrentAccount.

Interfaces and Realizations:

Interface is the unimplemented collection of intended behaviour.

Eg:

GraphicsInterface

showPicture();

clickNext();

drawWindow();

We should create realizations which can create these interfaces

Class

7 inch display implements graphics interface

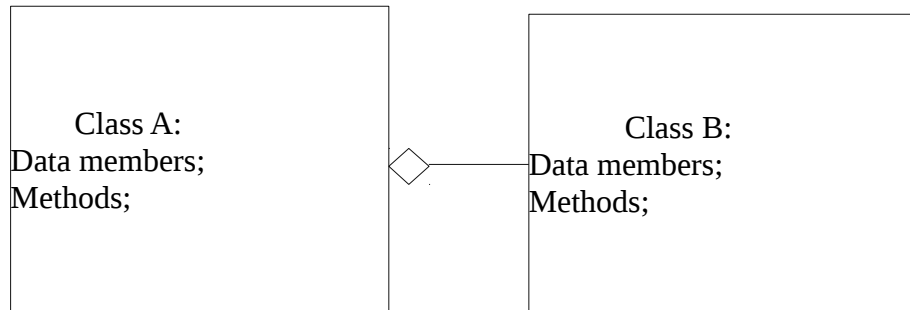
Body of the class is always realization.

In Java there is no multiple inheritance

One class can implement more than one interface. To have multiple inheritance we should create a class with more than one interfaces. In C++ to implement interface we have to use abstract class.

Scribe Notes:

Aggregation(read with reference to the diagram):



- It is an interclass relationship where there is a 1:n cardinality and the relationship is written as A (has-a) B.
- It is similar to an association because there is no existence dependency of B on A.
- However, it is different from an association because class B can belong to only one class (in this case A).
- Aggregation can be used to capture relationships where we use various components capable of independent existence to build some bigger object.

Examples:

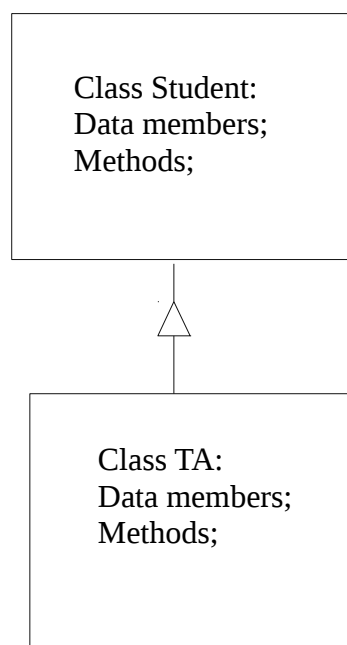
- using eggs to make cake,
- using hinges in the manufacture of a door.

Based on the example we see that for aggregation it doesn't make sense for a component to belong to more than one object.

Implementation:

We can't implement the exact semantics because although we can represent B belongs to A (with the associated cardinality) we can't model the notion 'B does not belong to any other class'. We have to utilise the specific constructs of the programming language to execute it.

Inheritance:



Implementation(based on the diagram):

a) C++

```
class Student{  
  
private: //you may use protected based on utility  
    string name;  
    vector<string> subjects;  
}  
class TA: public Student{  
//TA data  
private:  
    string Course;  
}
```

b) Java

```
public class Student{  
}  
public class TA extends Student{  
}
```

c) Python

```
class SubClass(Name_parentclass):  
    //adding the name of the class  
    that the subclass is inheriting  
    from within parenthesis.
```

Memory Allocation:

TA instance_TA; -- when a subclass is instantiated, the memory associated with data of both subclass and superclass is allocated, however the memory associated with the methods(along the hierarchy) is allocated only once(regardless of the number of objects).

The access specifiers – private, protected, public only define the scope of variables and methods and don't dictate memory allocation.

When the subclass is 'freed', then the superclass chain above it also 'freed'.

Identifying the Inheritance Hierachy:

- Specialization: If you have a general purpose class which has stand-alone meaningful existence. Now check if you have classes which are specialisation of this general class with some extra data or characteristic methods. This method is top-down because our starting point is a generalised parent class.
- Generalisation: If you have a number of classes, that have some commonalities then you could group them into a general class and override or add certain properties to make the subclasses. This approach is bottom-up because you look at common properties to establish the parent class. Ex: you look at professionals of an MNC(finance, BPO,backend support) and group them under a parent class of employees of company X.

Inheritance and Composition:

- We must use inheritance with an intention to exploit polymorphism.

Ex:

```
class Account // parent class with print statement as a method  
class SavingsAccount // sub class with print statement as a method  
class CurrentAccount // another subclass with print statement as method
```

If we have a vector <Account*> (a vector holding pointer to class of type account).

```
Account * A = new Account();
```

```
Account * B = new CurrentAccount;
```

```
Account * C = new SavingsAccount;
```

```
// we pushback these pointer into the vector
```

When we loop over them calling the print statement on each of them, then based on the type of Account the corresponding print statement is executed.

However if we implement the classes as an composition, then the print statement is not decided implicitly, we have to use multiple if statements which is tedious if we have a lot of sub-class types.
//

```
if(A == SavingsAccount)
    SavingsAccount->corresponding_print_method
if(A == CurrentAccount)
    CurrentAccount->corresponding_print_method
```

How to not override in Java:

```
class Student{
    public void greeting(){
        System.out.println("Hello");
    }
}
class TA{
    public void greeting(){
        System.out.println("Hello Everyone");
    }
}
```

a) when Student A = new Student();
Student B = new TA();

```
B.move();
//output showing implicit overriding
Hello Everyone
```

```
class Student{
    public void greeting(){
        System.out.println("HI");
    }
}
class TA{
    public void greeting(){
        super.greeting();
        System.out.println("Hey");
    }
}
```

b) when Student A = new Student();
Student B = new TA();

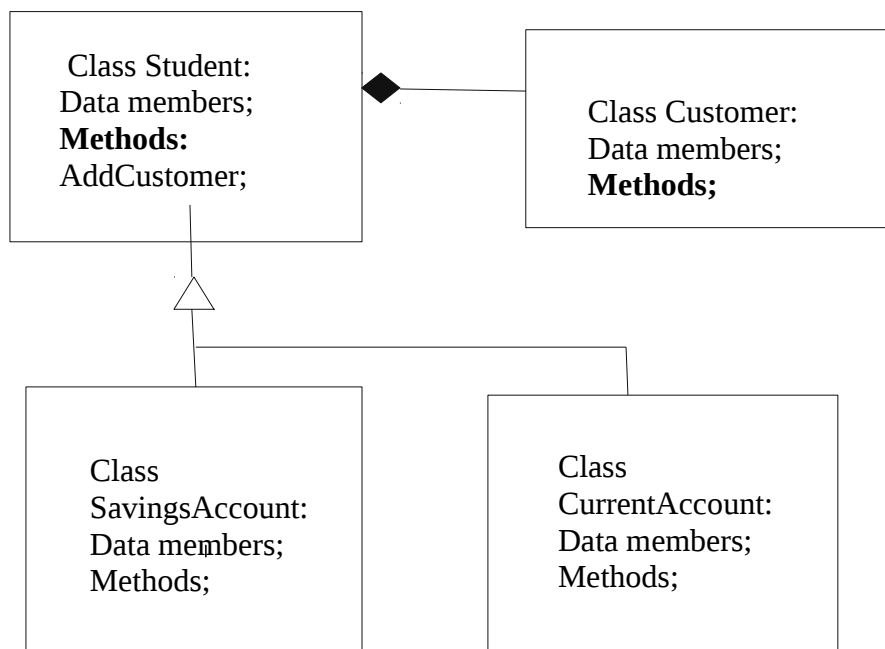
```
B.move();
//output prevents overriding
HI
```

Notions Associated with object graph:

During inheritance, the data, methods and relationships are inherited.

Example:

```
Account * A = new SavingsAccount(parameters); Customer * C = new Customer(parameters);
A->addCustomer(C);
```



Interface and Realisation:

It is a collection of intended behaviour. It is unimplemented. An interface has methods which forms a template/contract of sorts. A realisation then implements this contracted behaviour.

Ex:

Graphics interface:

```
showPicture();  
clickNext();  
drawWindow();
```

A 7inch display could then implement this graphics interface.

In C++ this is realised using inheritance.