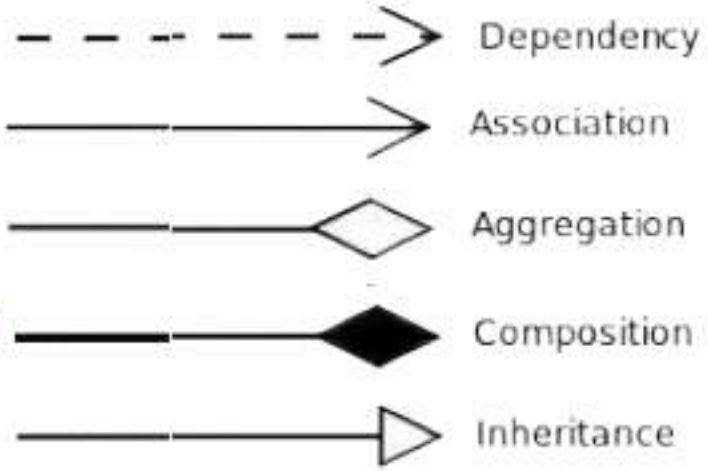
Relationship between classes and Objects

1. Dependency (use-a)
2. Association (use-a)
3. Aggregation (has-a)
4. Composition (whole-part)
5. Inheritance (is-a)



Dependency (use-a) Example

- ostream and istream objects are used in operator functions.
  - `friend istream& operator>> (istream& , Point&);`
  - `friend ostream& operator<< (ostream& , const Point&);`
- ostream and istream object are neither created inside class object, nor they are related to the object
- Life time (creation and destruction) of Point, ostream and istream is independent

- Unidirectional
  - istream and ostream classes are unaware of existence of Point class and its objects,
  - but Point class is aware of the use in operator functions

Association (use-a)

- Weak relation, no ownership of objects is involved
- Object of one class can be associated with object(s) of other class(s) for performing some tasks
  1. one-to-one,
  2. one-to-many
  3. many-to-many
- Objects have independent life time (creation and destruction)
- Objects are unrelated to one another
- Objects may or may not know about the existence of the object
  - Unidirectional
  - Bidirectional

Association

- **Kinds of Association:**
  There are two main types of association which are then further subdivided i.e
  - 1. Class Association
  - 2. Object Association

Association

Class Association

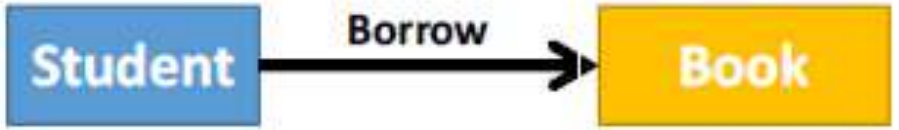
- Class association is implemented in terms of Inheritance. Inheritance implements generalization/specialization relationship between objects. Inheritance is considered class association.
  - In case of public inheritance it is “IS-A” relationship.

Association

Object Association

- It is the interaction of stand alone objects of one class with other objects of anther class.
- It can be of one of the following types,
  - Composition
  - Aggregation

Association (use-a) Examples



- One-to-Many relation, one student can borrow many books from a library.
- No owner ship and lifetime is involved in this relationship.
- A list of ids of borrowed books can be added to student.

```
class student{
private:
    int  sId;
    int *borrowedBooks;
    //Maintain the list of borrowed
    books
public:
    void borrowABook(const int & bid);
    void ReturnABook(const int & bid);


};
```

Simple Association

- The two interacting objects have no intrinsic relationship with other object. It is the weakest link between objects. It is a reference by which one object can interact with some other object.
  - Customer gets cash from cashier
  - Employee works for a company
  - Ali lives in a house
  - Ali drives a car

Simple Association

Simple association can be categorized in two ways,

- With respect to direction (navigation)
- With respect to number of objects (cardinality)

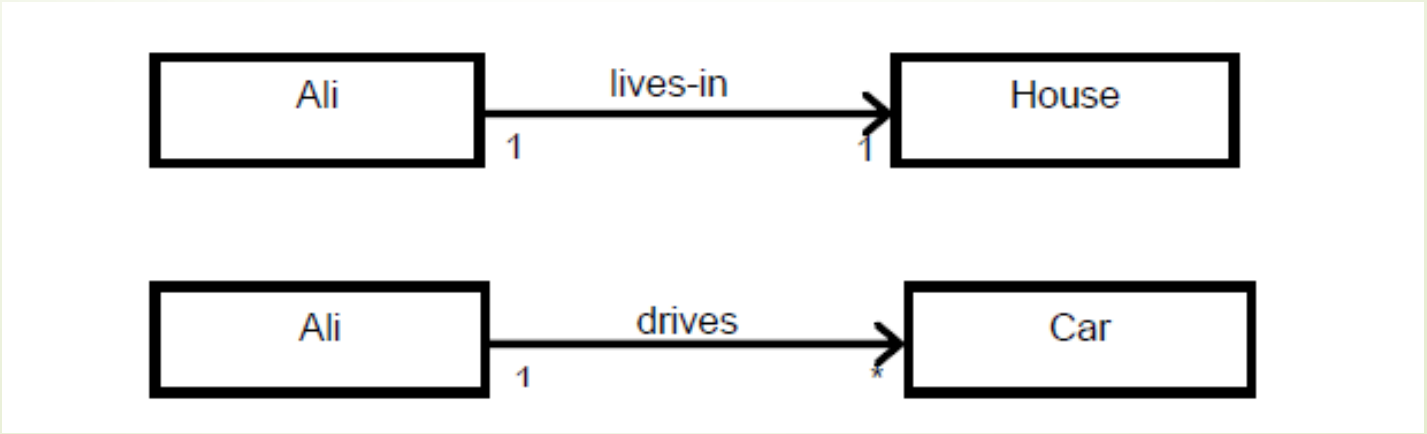
Kinds of Simple Association w.r.t Navigation
With respect to navigation association has the following types,

a. One-way Association
b. Two-way Association

a. One-way Association

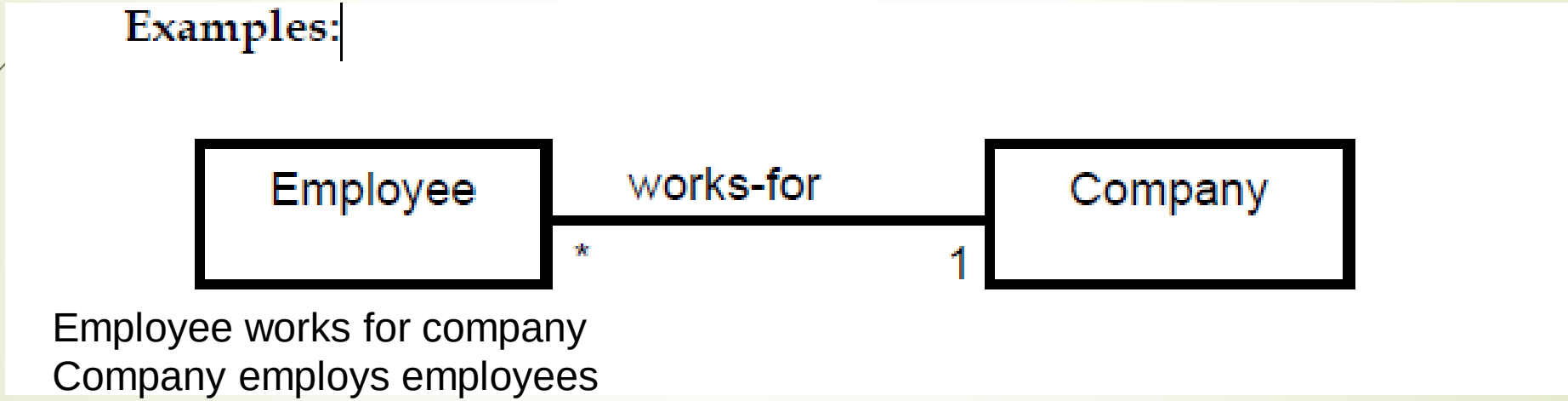
In One way association we can navigate along a single direction only, it is denoted by an arrow towards the server object.

Examples:



Simple Association

Two-way Association
In two way association we can navigate in both directions, it is denoted by a line between the associated objects
Examples:



For example Managers and Employees, multiple employees may be associated with a single manager and a single employee may be associated with multiple managers.

Kinds of Simple Association w.r.t Cardinality

With respect to cardinality association has the following types,

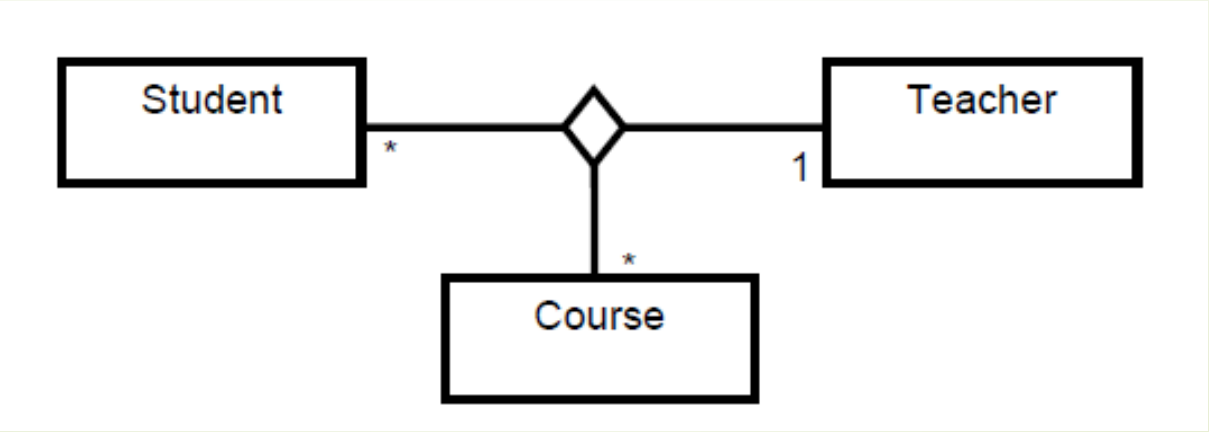
a. Binary Association
b. Ternary Association
c. N-ary Association

Binary Association

It associates objects of exactly two classes; it is denoted by a line, or an arrow between the associated objects.

Ternary Association

It associates objects of exactly three classes; it is denoted by a diamond with lines connected to associated objects.

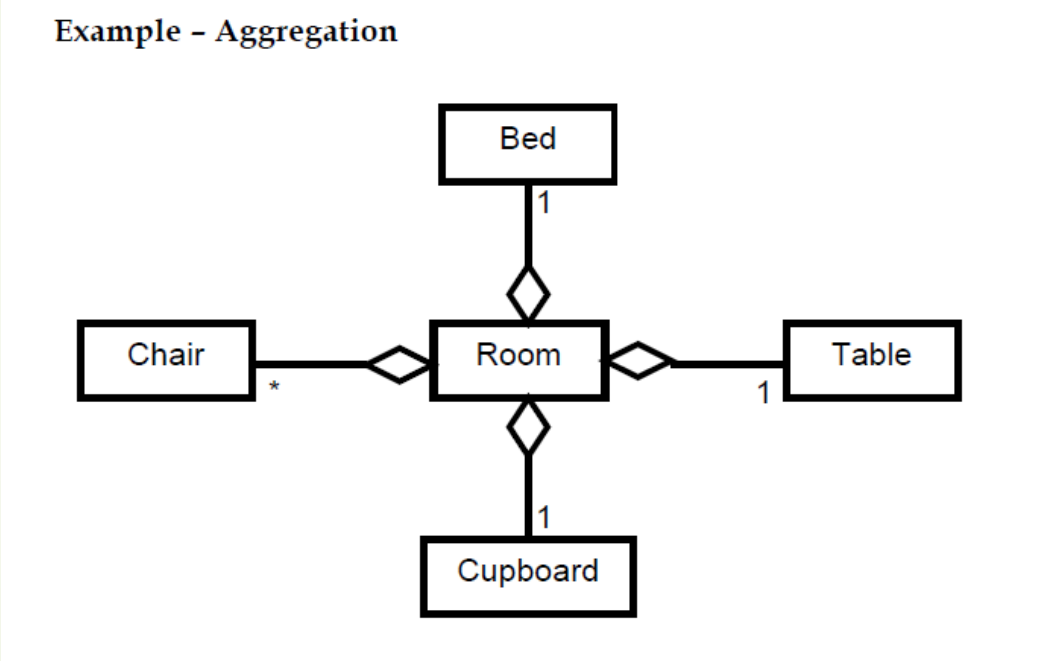


Aggregation (has-a)

- Subset of association relation where ownership is involved
- Weak relation
- Object of one class can contain object(s) of other class(s) for specific amount of time
  1. one-to-one,
  2. one-to-many
- Unidirectional object of container class knows about its parts
- Objects have independent life time (creation and destruction)

Aggregation

- An object may contain a collection (aggregate) of other objects, the relationship between the container and the contained object is called aggregation, Aggregation is represented by a line with unfilled-diamond head towards the container

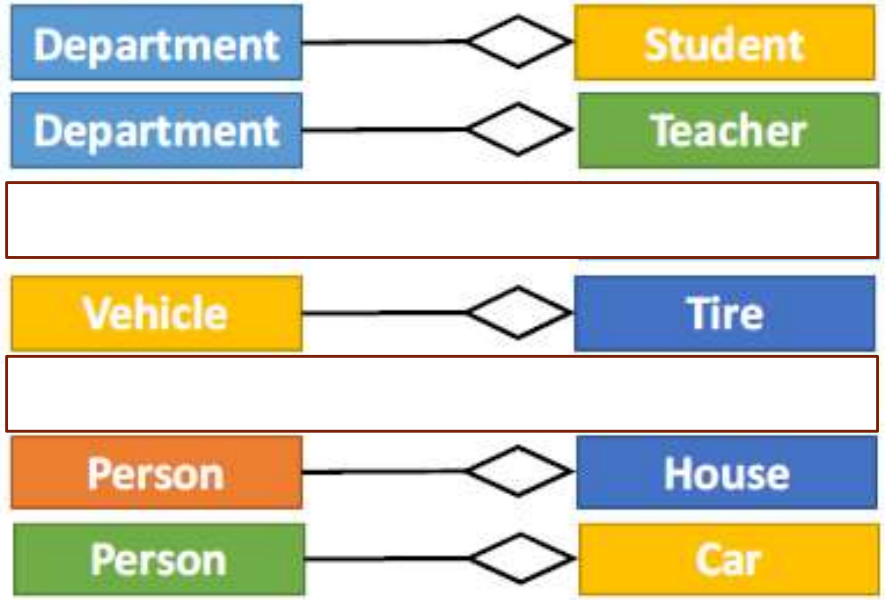


We can say it is a direct association among the objects. In Aggregation, the direction specifies which object contains the other object. There are mutual dependencies among objects.

For example, departments and employees, a department has many employees but a single employee is not associated with multiple departments. Employees may exist without a department. Here, department can be called an owner object and the employee can be called a child object.

Aggregation (has-a)Examples

- One department has many students.
- A department has many teachers.
- A University has many departments.
- A vehicle has many tires.
- A car has an engine.
- A person owns a house.
- A person owns many cars.



```
class B{
    int b;
    public:
    B(int b=0){ this->b=b;}
};
class A{
    int a;
    B * objB; //pointer
public:
    A(int a=0){ this->a=a;}
    void addB(B *b){ this->objB = b;}

    void removeB(){ objB = nullptr;}
    Void changeB(B*b){ objB= b;}
    ~A(){ objB=nullptr;}
    //nothing to do with objB
};
```

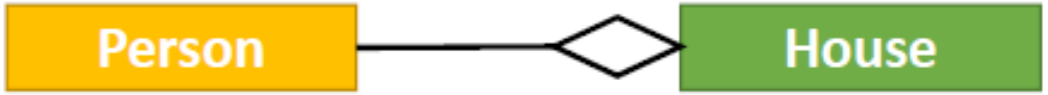
Use a pointer to aggregate class object(s).

A B

```
void main(){
    A a(1), a2;
    B b, b2(3);
    a.addB(&b);
    a.changeB(&b2);
    a2.addB(&b);
}
```

Aggregation

Aggregation (has-a) Implementation



- One to one
- House pointer in person class points to aggregate class object.

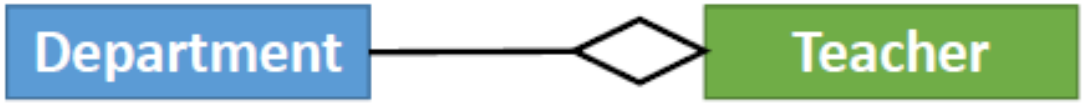
```
class House{
    int hid;
public:
    House(int h=0){ this->hid=h;}
};

void main(){
    House * h  = new House(1);
    Person p(1, h);
    p.removeHouse();
    delete h;
}
```

```
class Person{
    int  pid;
    House * hptr; //pointer for house
public:
    Person(int pid, House * hptr){
        this->hptr =hptr;
        this->pid = pid;
    }
    void changeHouse(House * h){
        hptr = h;
    }
    void removeHouse(){ hptr = nullptr;}
    ~Person(){
        hptr = nullptr;
    }
};
```

Aggregation

Aggregation (has-a) Implementation



- **One to many**

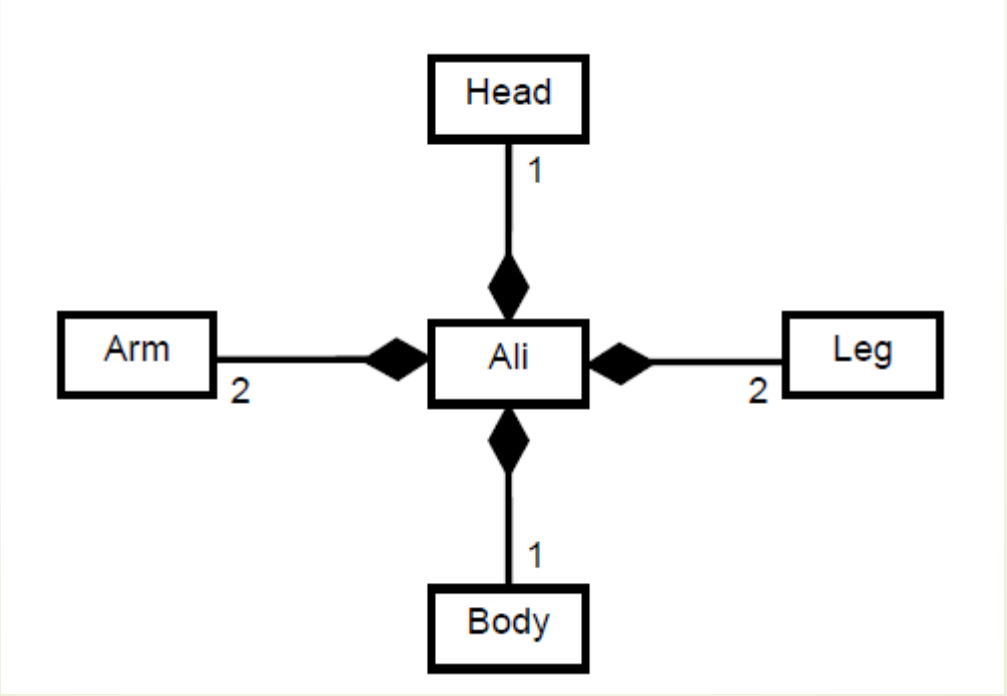
```
class Teacher{
    int tid;
    char * name;
public:
    Teacher(int t=0, char*n=nullptr){
        tid=t;
        name = nullptr;
        if(n!=nullptr){
          name = new char[strlen(n)+1];
          strcpy(name, n);
        }
    }
    ~Teacher(){
        if(name != nullptr)
        delete [] name;
    }
};
```

```
class Department{
    int  did, noofteachers, current;
    Teacher ** tList; //pointers list
public:
    Department(int id = 0, int noofteachers = 10){
        this->noofteachers= noofteachers;
        tList = new Teacher * [noofteachers];
        current = 0;
    }
    void AddTeacher(Teacher * t){ tList[current++] = t;
    void RemoveTeacher(int tid);
    ~Department(){
        for(int i=0; i< noofteachers;i++}
            tList[i] = nullptr;
        delete[] tlist;
    }
```

Composition

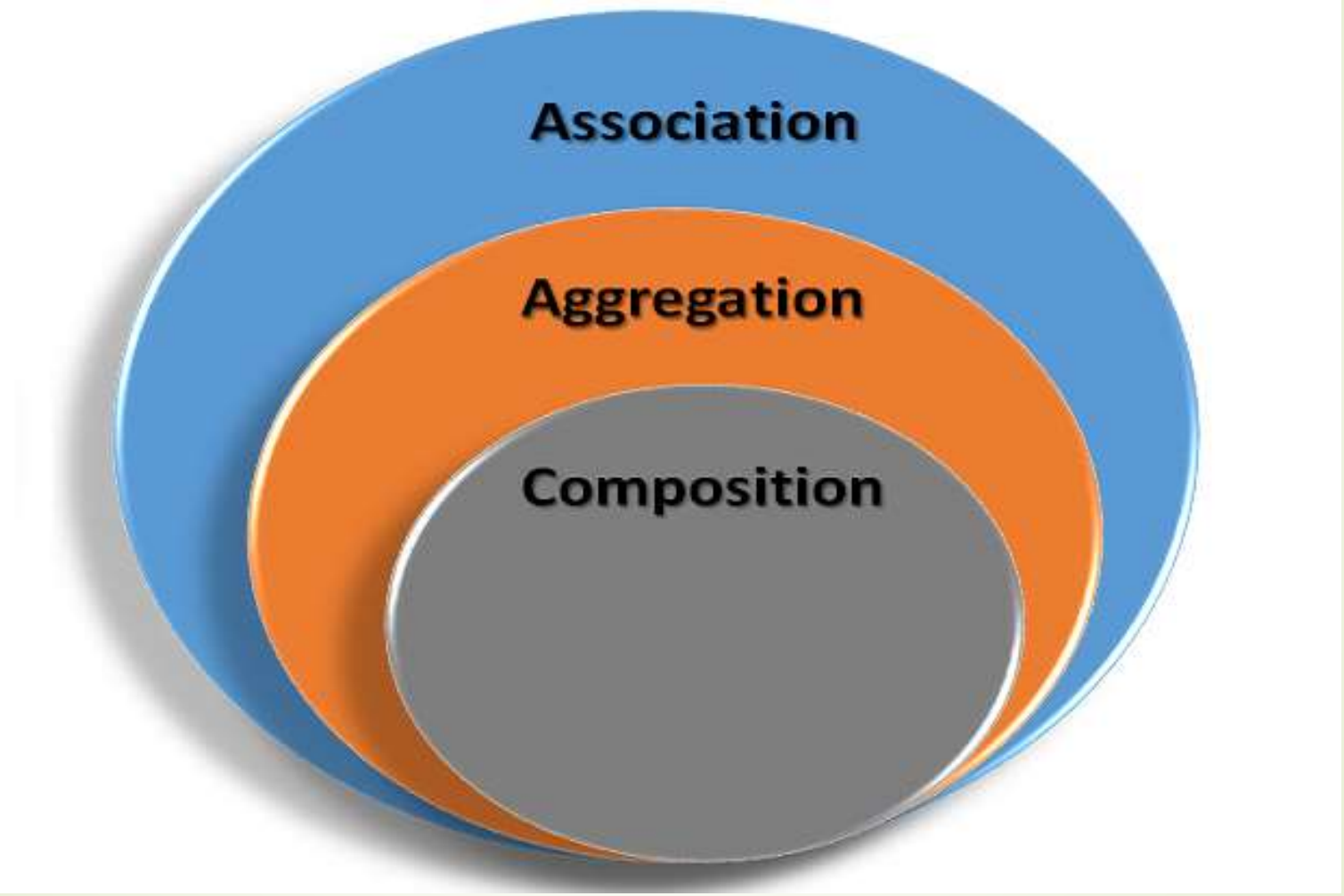
An object may be composed of other smaller objects, the relationship between the "part" objects and the "whole" object is known as Composition. Composition is represented by a line with a filled-diamond head towards the composer object

Example – Composition of Ali

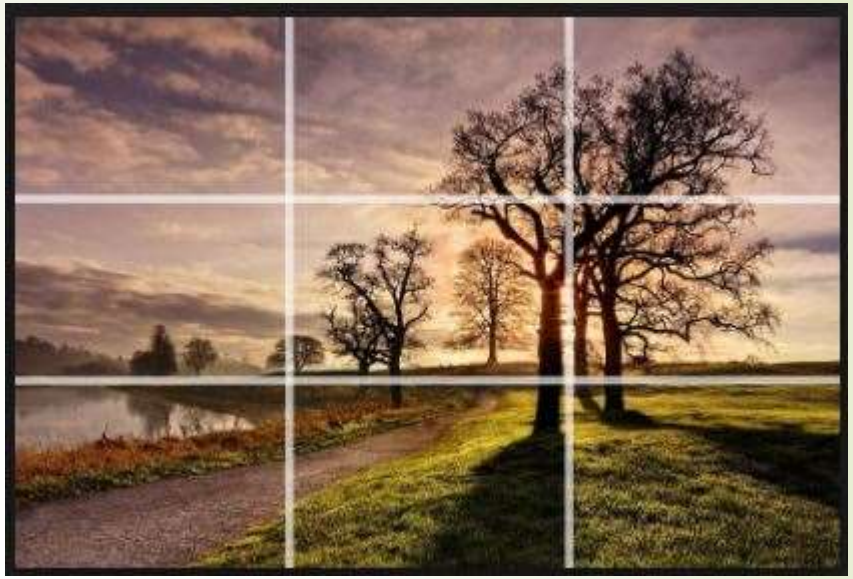


Composition

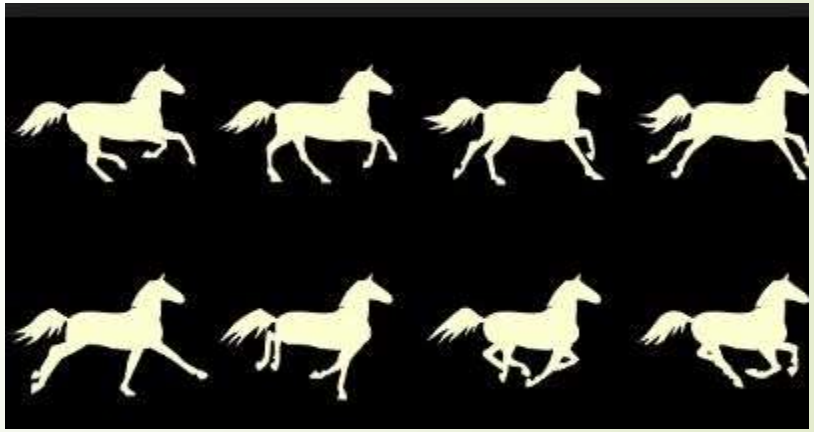
- **Composition is stronger relationship:**
  - Composition is a stronger relationship, because
  - Composed object becomes a part of the composer
  - Composed object can't exist independently


Association
Aggregation
Composition

Composition: The following picture is image composition i.e. using individual images making one image.



collection of image in single location



Composition (whole-part) Example

Person

- Single class person controls every thing

```
class Person{
    int  pid;
// Name
    char * fname;
    char * lname;
//Date of Birth
    int day;
    int mon;
    int year;
//Address
    char * city;
    char *country;
    int streetNo;
    int houseNo;
};
```

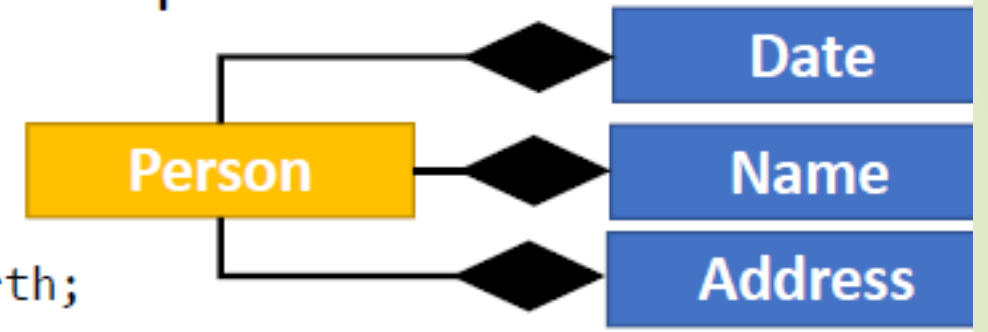
- Not scalable
- Error prone
- Not reusable in other class
- Redefine all attributes and functions separately for other classes
- For example student, doctor and teacher, patient

Composition (whole-part) Example

- Design separate classes

```
class name{
    char * fname;
    char * lname;
};
class date{
    int day;
    int mon;
    int year;
};
class address
    char * city;
    char *country;
    int streetNo;
    int houseNo;
};
```

```
class person{
    int  pid;
    name pname;
    date dateofBirth;
    address paddress;
};
```



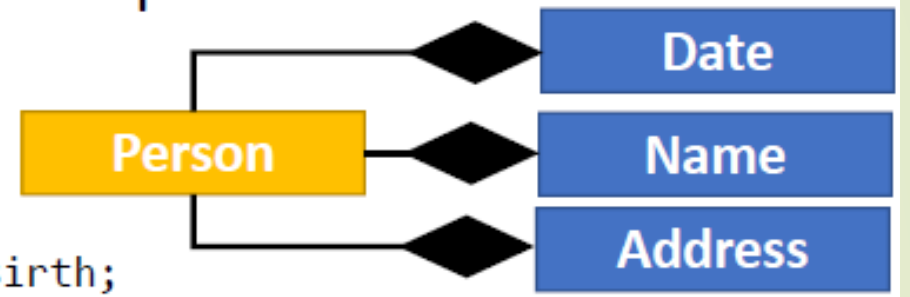
- Add objects as variables in class
- Scalable
- Less Error prone
- Reusable in other classes such as student, doctor and teacher, patient
- No need to redefine all attributes and functions separately for other classes

Composition (whole-part) Example

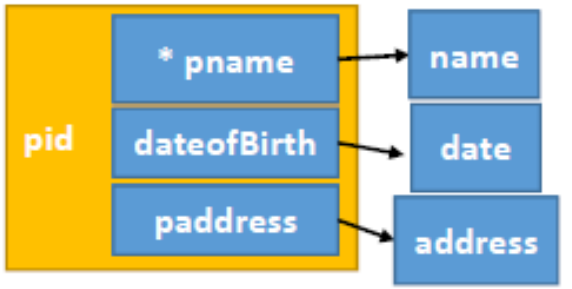
- Design separate classes

```
class name{
    char * fname;
    char * lname;
};
class date{
    int day;
    int mon;
    int year;
};
class address
    char * city;
    char *country;
    int streetNo;
    int houseNo;
};
```

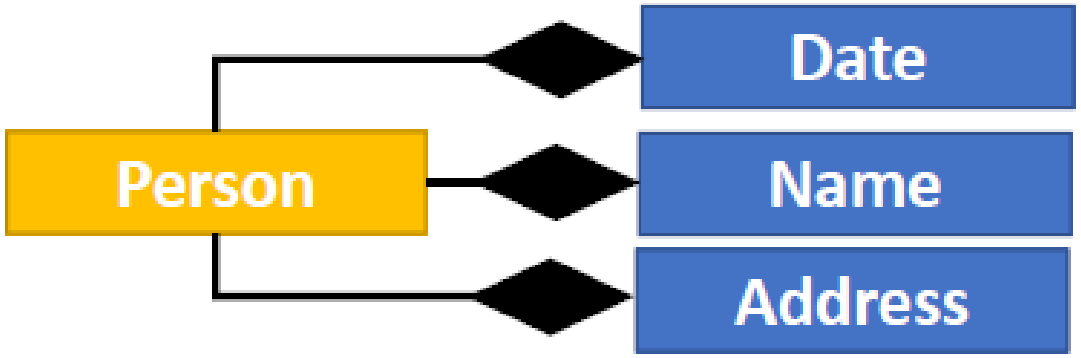
```
class person{
    int  pid;
    name * pname;
    date * dateofBirth;
    address * paddress;
};
```



- Add objects as pointer to variables in class
- Scalable Less Error prone
- Reusable in other classes such as student, doctor and teacher, patient
- No need to redefine all attributes and functions separately for other classes



Composition (whole-part) Example



- Call functions of composed classes

```
class person{
    int  pid;
    name pname;
    date dateofBirth;
    address paddress;
public:
    person(int pid, char*fn, char*ln, int d, int m, int y, char*city, char*country,
    int street, int house)
    :pname(fn,ln), dateofBirth(d,m,y), paddress(city, country, street, house)
    {
        this->pid=pid;
    }
//call parameterized constructors for object separately
};
```