**Specialization Hierarchy** – has the constraint that every subclass participates as a subclass in only one class/subclass relationship, i.e. that each subclass has only one parent. This results in a tree structure.

**Specialization Lattice** – has the constraint that a subclass can be a subclass of more than one class/subclass relationship. The figure shown above is a specialization lattice, because Engineering\_Manager participates has more than one parent classes.

**Java Class File:-**

Java Class files are identified by the following 4 byte header (in hexadecimal): CA FE BA BE (the first 4 entries in the table below).



**malloc():-** Allocates requested size of bytes and returns a pointer to the first byte of allocated space.

**calloc():-** Allocates space for an array elements, initializes to zero and then returns a pointer to memory.

**free():-**dellocate the previously allocated space.

**realloc():-**Change the size of previously allocated space

**Array of pointers**

An array of pointers is an array, where the elements are pointers to some memory location.

for eg- it would be something like int\* parr[10]; here parr[0],parr[1],etc are pointer variables

**Pointer to an Array**

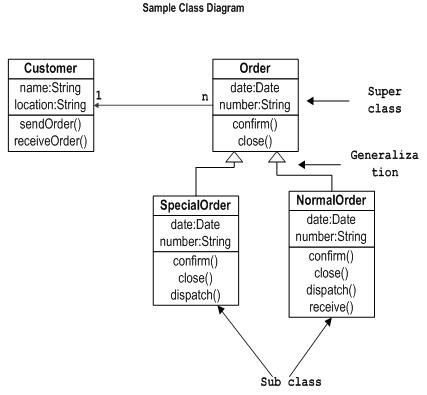
A pointer to an array is when a pointer is pointing to the starting address of an array.

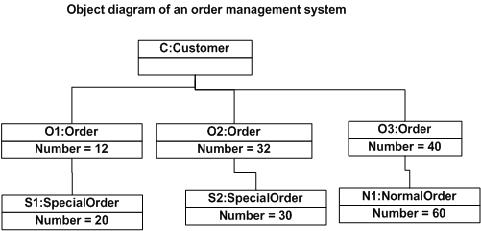
eg, let int arr[10] be an array. A pointer that points to arr[0], would be a pointer to an array

For eg:- int (\*ptr) [10] is ptr is a pointer to an array of 10 integers.

**Class diagram** shows a collection of declarative (static) model elements, such as classes, types, and their contents and relationships.

**Object diagram** encompasses objects and their relationships at a point in time. It may be considered a special case of a class diagram or a collaboration diagram





***QUES:- Which diagram provides a formal graphic notation for modelling objects, classes and their relationships to one another ?***

(A) Object diagram (Ans) (B) Class diagram

(C) Instance diagram (D) Analysis diagram

**PARALLEL PROCESSING**

**Amdahl's Law**

Execution time after improvement =

Execution time unaffected +

Execution time affected / Amount of improvement

TI = TU + TA/Improvement

TI = TU + TA/Speedup

**More About Amdahl's Law :-**

speedup is a function of the fraction of the code that can be parallelized and the number of processors

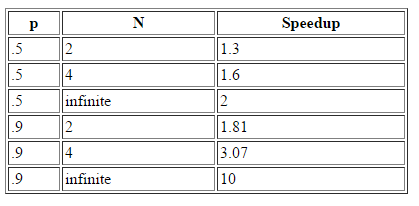
non-parallel sections do not gain performance

Speedup(N) = 1/((1-p)+p/N)

where p= portion of the code that can be made parallel

N=number of processor. For example:

N = The part which performance needs to be improved.



***The speed up of a pipeline processing over an equivalent non-pipeline processing is defined by the ratio :***

(A) S = n tn/ (k + n – 1)tp(yes)

(B) S = n tn/ (k + n + 1)tp

(C) S = n tn/ (k – n + 1)tp

(D) S = (k + n – 1)tp/ n tn

Where n → no. of tasks; tn→ time of completion of each task;

k → no. of segments of pipeline; tp→ clock cycle time;

S → speed up ratio;

***Without pipeline one task needs tn time.***

*• n tasks need ntn time.*

***With pipeline,***

• First task needs k cycles to finish. So time will be k tp

• Other n-1 tasks needs tp time only to finish.

• Total time = (k+ n -1 ) tp

***Speed up = T pipeline / T without pipeline = n t n / (k+n-1) t p***

***A set of processors P1, P2, ......, Pk can execute in parallel if Bernstein's conditions are satisfied on a pair wise basis; that is P1 || P2 || P3 || ..... || Pk if and only if:***

(A) Pi || Pj for all i ≠ j(yes) (B) Pi || Pj for all i = j+1

(C) Pi || Pj for all i ≤ j (D) Pi || Pj for all i ≥ j

***What is the result of the expression (1&2)+(3/4) ?***

(A) 1 (B) 2 (C) 3 (D) 0(yes)

***When one-dimensional character array of unspecified length is assigned an initial value :***

(A) an arbitrary character is automatically added to the end of the string

(B) ‘o’ is added to the end of the string(yes)

(C) length of the string is added to the end of the string

(D) ‘end’ is added to the end of the string

## Calculating Constant Failure Rates

In order to measure failure rates, you need a sample of identical components or systems that can be observed over time. For example, suppose you had five light bulbs connected to an automatic circuit that you could then turn on and off once per hour for 1,000 hours, giving you the following data:

* Bulb 1 burned out after 422 hours.
* Bulb 2 burned out after 744 hours
* Bulb 3 burned out after 803 hours
* Bulb 4 burned out after 678 hours
* Bulb 5 stayed lit for 1000 hours

This gives you 4 failures over a total of 3,647 hours.

To calculate the failure rate, divide the number of failures by the total number of hours, such as 4/3,647 = 0.0011 failures per hour.

In this example, the failure rate per hour is so small that it is almost insignificant. Multiplying the number by 1,000 would make it more meaningful to someone thinking about buying a light bulb, which would be 1.1 failures per 1,000 hours. Since there are 8,760 hours in one year, you can divide 3,647 by 8,760 to get 0.41 failures per year, or about 2 failures every five years.

## Calculating MTBF

Another way to express failure rates is by using the Mean Time Between Failures. MTBF is usually used in high-quality systems where failures are expected to be rare and need to be minimized, like the guidance system on a commercial aircraft or the air bags in a passenger car. Knowing the MTBF allows manufacturers to recommend how often components should be inspected, maintained and replaced.

To calculate the MTBF, you divide the number of hours by the number of failures. In the case of the five light bulbs that were tested, which had a failure rate of 4 per 3,647, you determine the MTF as 3,647/4 = 909. The MTBF is therefore 909 hours.